Implementation of a Low-power Embedded Processor for IoT Applications and Wearables

Kareem Mansour, *Member, IEEE* and Ahmed Saeed, *Member, IEEE*

Abstract—Embedded processors are key building blocks for IoT platforms. Such processors should provide flexible computing and low-power consumption for the small form factor devices to have better battery life. This paper introduces an implementation of a new design for a 32-bit RISC embedded processor optimized for low-power budget and targeting IoT applications. The proposed processor is capable to execute a small set of simple instructions in few cycles, and hence, efficient for low-power embedded applications. The instruction set is inspired by the state-of-the-art Thumb-2 ISA by ARM. The performance of the processor is analyzed in terms of delay and power. The design is described in VHDL, implemented and simulated on Vivado and tested using Nexys 4 DDR board featuring Xilinx's Artix-7 FPGA.

Keywords—Embedded Processor, FPGA, IoT, Low-Power, MAC, Multiplier, Vedic, VHDL.

I. Introduction

INTERNET of Things (IoT) refers to a giant network that extends to everyday objects, namely 'Things'. Things can be sensors, actuators, wearables, mechanical machines, home appliances or even persons. These objects, while not considered computers, are able to communicate with other objects and computers through the Internet via embedded systems without human intervention. Fig. 1 depicts the components of an IoT device. Sensors in each IoT device are used to monitor and collect data from the surrounding environment; local *processor* or *microcontroller* (MCU) is used to process these data and interface to a wireless device for connectivity [1].

The internal design of the processor, which describes the hardware components and their interconnections, is known as the *microarchitecture* [2]. The microarchitecture is usually divided into two main parts: the datapath and the control unit. The datapath comprises structures — memories, registers, ALUs, etc. and the multiplexers that connect them together. The control unit is responsible for generating the driving signals that control these structures in order to define the dataflow. Fig. 2 abstracts the components that make up an ARM core; one of the most pervasive processors in the world that are embedded in a wide range of products from cell phones to vehicles [3].

A processor must execute a sequence of instructions, where each instruction performs some primitive operation, such as adding two numbers [4]. The instructions supported by a particular processor and their binary encodings are known as its *instruction-set architecture* (ISA). In computer organization, the processor architecture is a combination of its microarchitecture and ISA.These instructions are nothing but the basic data-processing, load/store, and branch instructions which are able to build a more complex program.

The Power consumption in mobile IoT devices depends on the type of sensors, *processor* and radio transceiver within the device [5]. Typically, IoT Devices are powered through

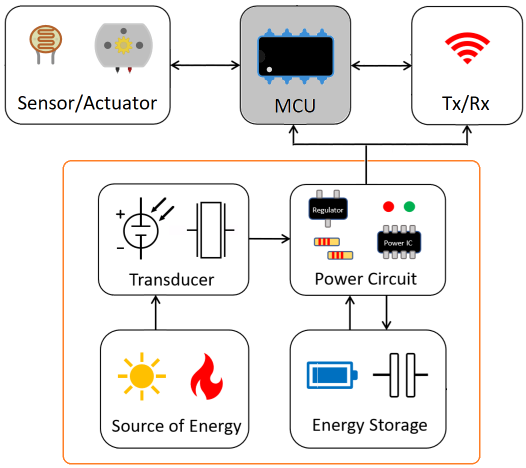


Fig. 1: Block diagram of IoT device.

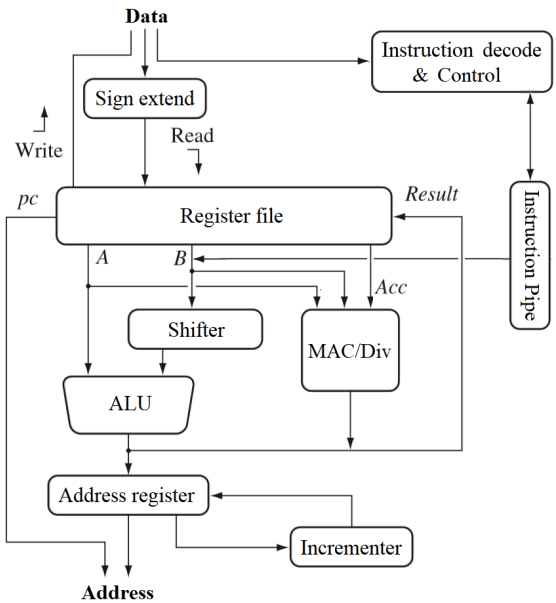


Fig. 2: ARM core functional units and dataflow model.

K. Mansour is with the Department of Microsystems Engineering - IMTEK, Albert-Ludwigs-Universität Freiburg, Freiburg, Germany. e-mail: mansour@tf.uni-freiburg.de.

A. Saeed is with the Electrical Engineering Department, Future University in Egypt, Cairo, Egypt. e-mail: asaeed@fue.edu.eg.

Manuscript received January 31, 2019; revised October 20, 2019.

rechargeable batteries and/or harvested energy for at least a portion of the time [6]. This is one of the reasons that power dissipation has become an important concern, where it is essential for devices to use minimal power while providing a good performance. Therefore, the power-performance trade-off drives the need for a new type of low-power processors.

Reduced Instruction Set Computer (RISC) processors can perform data transfer and set up communications efficiently [4]. On the other hand, digital signal processors (DSPs) are well suited to process the signals with lower latency and power. Combining RISC and DSPs into a single processor core would reduce the cost, area and complexity and meet the energy goals.

This paper introduces a new design for a low-power 32-bit embedded processor combining DSP and RISC. This architecture is able to execute a small set of simple instructions in a few cycles and hence, efficient for low-power embedded applications. The instruction set is inspired by the state-of-the-art Thumb-2 ISA by ARM [7].

The rest of the paper is organized as follows: section II introduces the architecture of the proposed processor, section III explains the design in details, the simulation and results are discussed in section IV. Finally, section V concludes the paper.

II. Processor Architecture

In this section, the architecture of the proposed processor is explained showing the purpose and the behavior of the different components of the processor. Fig. 3 depicts the microarchitecture of the proposed processor. It is also worth pointing out that the data lines are marked in black while control signals are marked in red.

The architecture of the proposed processor is based on a 32-bit RISC microarchitecture and a rich 32-bit instruction set. The data-path follows the Harvard architecture in which the instruction storage and data storage are physically separate. It consists of a register file, program counter (PC), instruction memory, data memory, Arithmetic-Logic Unit (ALU), barrel shifter, and a powerful Multiply-accumulate (MAC) unit. These structures are connected through multiplexers that enable multiple selections to the direction of dataflow. The processor features a three-stage fetch-decode-execute pipeline to improve the throughput of the processor by overlapping multiple instructions during execution.

The instruction set has a very regular encoding that supports several data-processing, load-store instructions, and branch instructions. Most instructions contain dedicated fields for: the operation to be performed, two source operands, a shift-control over the second operand, and a destination operand. The first source operand is always located in the register file whereas the second one can be an operand located in the register file or an immediate operand encoded into the instruction. All instructions are executed unconditionally, except the conditional branch instruction, which enables the program to bypass subroutines if a condition was not satisfied. The instruction set architecture of the proposed processor is found in appendix A.

A. Data-processing units

ALU is the fundamental building block of the processor which carries out most of the data-processing operations. It generates status codes, known as flags, which are useful for detecting errors and making comparisons and decisions in conditional codes. The ALU operations of the proposed processor are not limited to the basic arithmetic and logic operations, but they are also extended to include digital signal processing, single-input multiple-data (SIMD), and packing and unpacking operations which can significantly improve the performance of the processor. These operations can be performed on data stored in registers only.

The arithmetic instructions are divided into basic arithmetic and parallel arithmetic instructions. For basic arithmetic, a single operation is performed on a pair of operands of data type word. However, parallel arithmetic supports SIMD operations in which two or four addition, subtraction operations or a mix of them are done on a pair of two half-words or four bytes in one cycle. Fig. 4 illustrates the concept of addition/subtraction in SIMD. Hence, the processor requires less time to solve more complex problems at maximum speed and performance. The logic instructions refer to bitwise logical operations which are performed on 32-bit words of data.

The second operand of the ALU can be optionally shifted by a barrel shifter connected directly to the ALU. The barrel shifter is capable of performing five different types of 32-bit position shift and rotate operations. The shift operations are not restricted to shift instructions. However, the second operand can be shifted before executing some data-processing and data-transfer instructions in order to support address scaling and constructing immediate values. In this way, the barrel shifter can increase the number of operations performed per instruction, reduce the number of cycles required and improve the performance of the processor.

For the load instructions, the address is always calculated from a base register value, an immediate operand, PC value or another shifted base register value. A word of data at the calculated address is then read from the data memory and written into a destination register. The same goes for the store instructions, except that a PC value cannot be used to calculate the address and the word of data is read from a source register and written to a memory location.

DSP applications are typically performed by an optimized ***Multiply-Accumulate unit*** which multiplies two numbers and accumulates the result onto an accumulator. The proposed processor includes a new design for a low-power MAC unit capable of performing several 16-bit, dual 16-bit, and 32-bit MAC operations in which up to three operands are involved. The two operands are the multiplier and multiplicand while the third operand is used for accumulation purposes. Like the ALU, MAC operations can be performed on data stored in registers only. The proposed MAC operations can be carried out on signed or unsigned operands and the result can be a word of 32-bits for short multiplication or double-words of 64-bits in case of long multiplication. In order to maximize the performance, all the MAC operations are executed in one cycle. However, for long multiplication, two cycles are

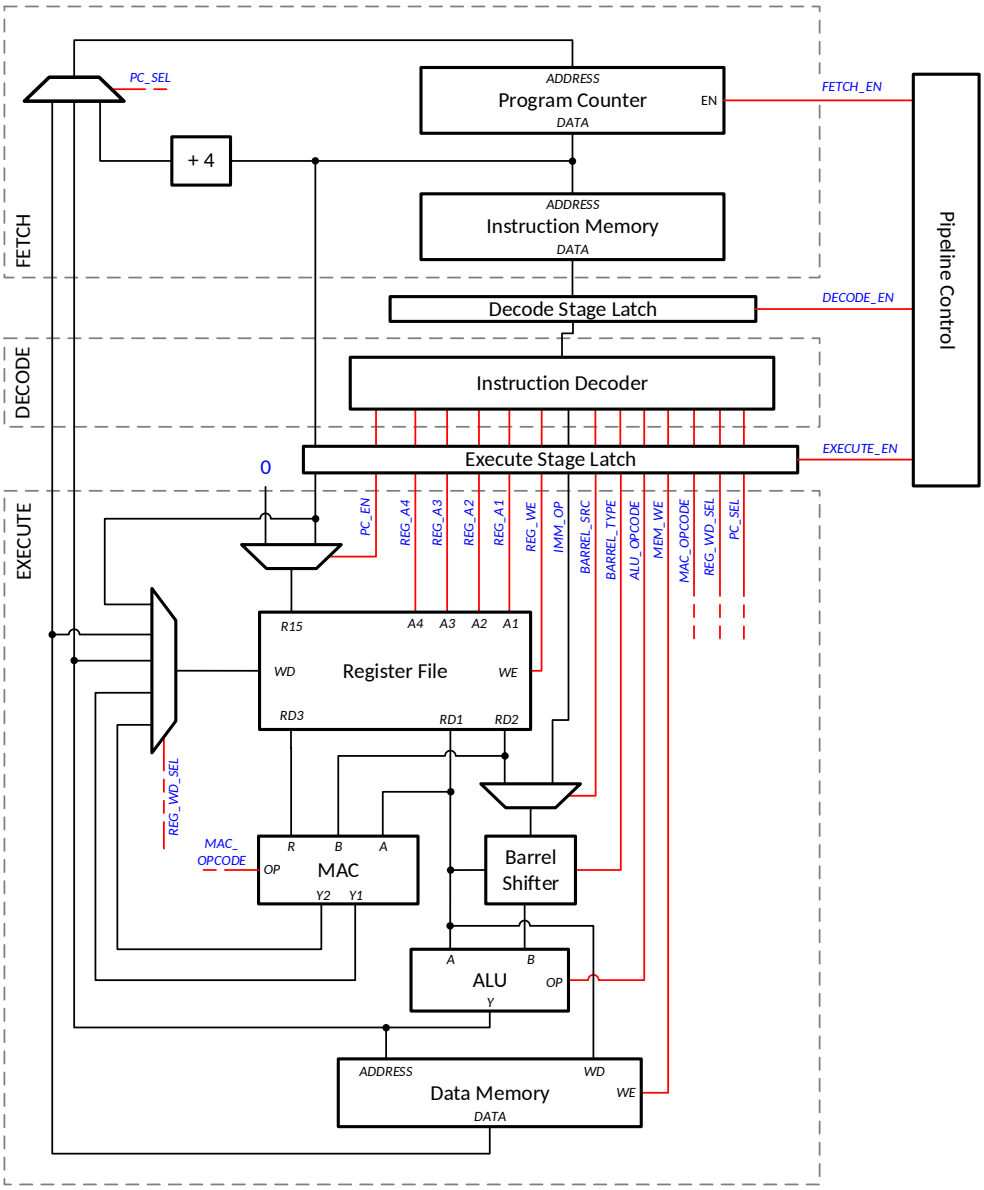


Fig. 3: The Microarchitecture of the Proposed Processor

required to store the result in the registers, as the register file is able to write only one register at a time. DSP instructions include saturation instructions that can be used with signed and unsigned operands to round their value to the maximum value supported by the processor and prevent overflow. MAC unit is a fundamental block that maximizes the performance of the processor.

Both ALU and MAC have no direct access to operands stored in memory. Memory can be accessed only by load-store instructions. Therefore at least two instructions are required to perform an operation on memory operands, or to perform an operation and store its result in memory. Data-processing and load-store instructions can read one of their operands from the PC. However, these instructions cannot change its value; only branch instructions can change the contents of PC to change the program flow.

B. Register File

The register file of the proposed processor consists of sixteen resisters. The registers R0 to R12 are general purpose registers that can hold all data types. Registers R13 to R15 are special purpose registers which can only hold the data essential for specific operations: R13 is a stack pointer (SP) that keeps the address of the last program request in a stack; R14 is a link register (LR) which contains the address to return to after finishing a subroutine or a function call completes; and R15 is the program counter that holds the address of the next instruction to be fetched. The program status register (PSR) is another special purpose register that stores the flags used in conditional execution. However, PSR is not a part of the

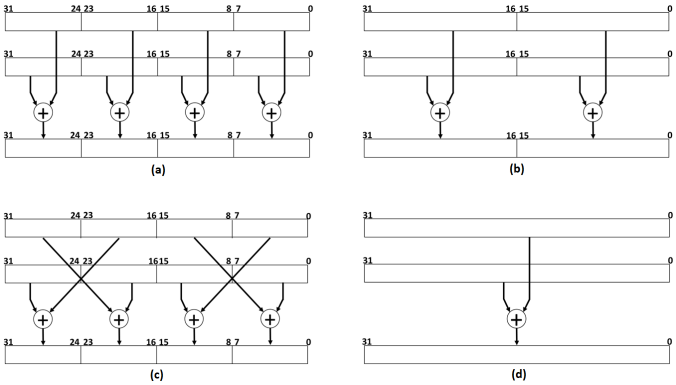


Fig. 4: Addition/Subtraction in SIMD: a) byte operands, b) halfwords, c) bytes interchanged and d) complete word

register file.

Unlike general purpose registers, only certain instructions are able to access special purpose registers and changing the contents of these registers will likely cause a change in the program flow. A register specifier to R15 can have different meanings according to the instruction. For branch and load/store instructions, this register specifier lets data to be read and written into the PC. However, in most of the data-processing instructions, it indicate a second operand of zero when used as a source register, or that the result of the operation is not to be written when used as a destination register.

The proposed processor is able to address memory of up to 4GB. The storage is distributed among two memory elements: an instruction memory that stores the program code and a data memory for data operands. The processor memory is the primary storage that contains the data and instructions of the program in operation. The instruction memory is accessed every cycle to load a new instruction into the pipeline. The data memory is read or written only when a load/store instruction is performed, which is responsible for the data transfer between the data memory and registers.

C. Pipeline Stages

Each stage of the pipeline consists of a stage-latch to hold the data of the stage and block any change from the previous stages. The latch is followed by a combinational circuit that performs operations on data within the stage. The enable signals of these latches are synchronously generated every cycle by the control unit. Fig. 5 illustrates the pipeline stages.

The pipeline stages are organized as follows:

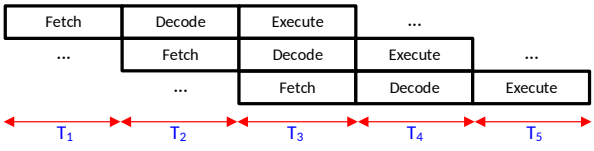


Fig. 5: The three-stage pipeline of the proposed processor

- **Fetch Stage:** An instruction, in a memory location stored in the PC, is read from the instruction memory and the next instruction address is calculated. The results of the stage are passed to the decode-stage latch.
- **Decode Stage:** The instruction is decoded by the instruction decoder, which generates all the control signals required to switch on and off the components of the processor and control the selection of the multiplexers to define the dataflow. It generates the addresses of the operands to be read from, or written into registers and memory. Furthermore, it decodes immediate operand expressions if available within the instruction. The generated control signals and the immediate operand are connected to the execute-stage latch.
- **Execute Stage:** The control signals generated from the decode stage are allowed to drive the components of the processor. At the beginning of the cycle, the register file is read, shift type and shift operand are applied to the barrel shifter, operands are processed by data-processing units (ALU or MAC unit). The source of the next instruction is selected such that the PC is updated to the address of the following instruction in the sequence, or switch to another one in case of executing a branch instruction. Finally, the results are written into the register file or data memory by the end of the cycle.

D. Control Unit

The control unit is responsible for directing the response of the processor components. Once an instruction is fetched, the control unit receives the current instruction from the datapath and tells the datapath how to execute that instruction. It includes the instruction decoder along with the extra logic that provide timing and control signals according to the instruction. Specifically, it generates the driving signals for multiplexer select, register enable, and memory write to control the operation of the datapath. The control unit plays an important role as well in controlling the stages of the pipeline to prevent hazards.

III. Processor Design

A. Register File

The register file can read up to three registers through three read-ports, namely ‘RD1’, ‘RD2’ and ‘RD3’. Each read-port is associated with a 4-bit register specifiers namely, ‘A1’, ‘A2’ and ‘A3’, to determine the source registers. It can write in one register at a time through one write-port ‘WD’ associated with a register specifier ‘A4’. Source operands are accessed through the register file read ports at the beginning of every cycle, while destination operands are are applied to the write-port by the end of the same cycle. Data is only written into the specified registers at the rising edge of the following clock cycle when a write enable signal WE is asserted. Clock-gating low-power technique is used to reduce power consumption by blocking the clock signal from the registers that are not in use. Fig.6 depicts the design of the register file.

The dataflow in the register file is controlled by a 2-phase non-overlapping clocks, generated from the main clock.

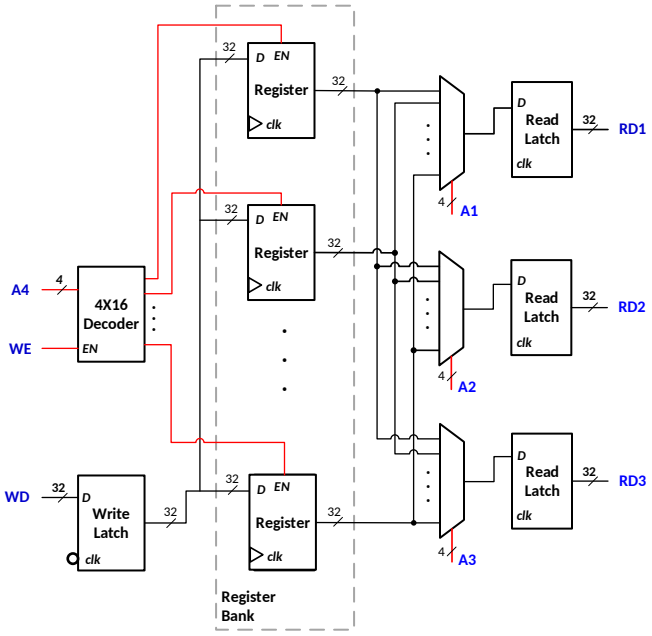


Fig. 6: Block Diagram of the Register File

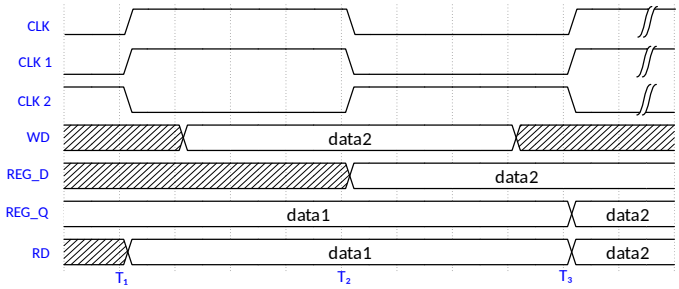


Fig. 7: 2-phase Non-Overlapping Clock Scheme

During phase 1, the read-port latches are activated and data stored in the specified source registers are allowed on the read-ports. The write-port, however, is not active and hence, any change in its data will not be applied to the inputs of the registers. On the other hand, the read-port latches are deactivated during phase 2 and any change in the data stored inside the registers will not affect the read-ports until the next clock cycle. The write-port latches are enabled and the destination operands are allowed on the input of the registers. At the next rising clock edge, the write-port data are written in the specified register and are ready to be accessed. In this way, we can easily achieve the three-stage pipeline and an extra writeback stage is no longer required. Fig.7 illustrates the 2-phase non-overlapping clock scheme.

Memories are accessed using addresses of 32-bits and data bus width of 32-bits.The storage space is distributed among two memories whose addresses do not overlap, one for instructions and another for data. The implemented instruction memory is a 4KB asynchronous read-only memory. The data memory, on the other hand, is a synchronous 1MB read/write memory. Data can be written only in the data-memory when a write enable signal 'WD' is asserted.

B. ALU

The ALU as shown in Fig. 8 is divided into three parts: arithmetic, saturation and logic parts. The design of the arithmetic block consists of four 8-bit adders which can be used separately or together, through multiplexers, to form two 16-bit adders or a single 32-bit adder. The result can be saturated or divided by 2 according to the instruction. The reason behind this design is to utilize and reuse the same components to execute a wide range of DSP instructions with minimum hardware reducing the power consumption. The design of arithmetic block is shown in Fig. 9 and Table. I.

The saturation block can perform signed or unsigned saturation. For signed saturation, the overflow flag is inspected. In case of no overflow, the result is not saturated. Otherwise, the result will be saturated to the largest signed positive number if the overflowed result is negative and to the smallest signed negative number if positive. For unsigned saturation, the carry-in and carry-out of the arithmetic operation are inspected. The carry-in indicates whether the operation was addition or subtraction. The unsigned value is not saturated when the two

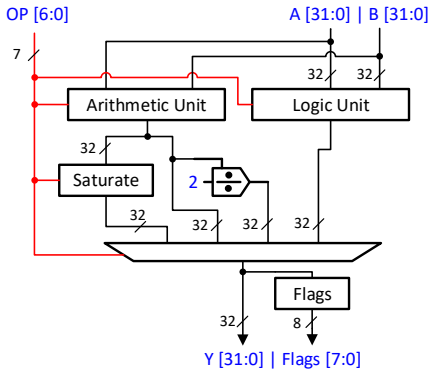


Fig. 8: The design of the ALU

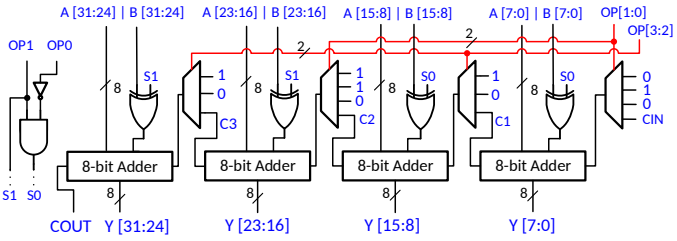


Fig. 9: The Design of the Arithmetic Block

TABLE I: The Design of the Arithmetic Block

OP[1:0]	OP[3:2]	Adders Inputs	Operation
00	00	A, B	32-bit Adder
01	00	A, B	Two 16-bit Adders
10	00	A, $\bar{B}$	Two 16-bit Subtractors
11	00	A, $\bar{B}$	16-bit Adder and 16-bit Subtractor
01	01	A, B	Four 8-bit Adders
10	10	A, $\bar{B}$	Four 8-bit Subtractors

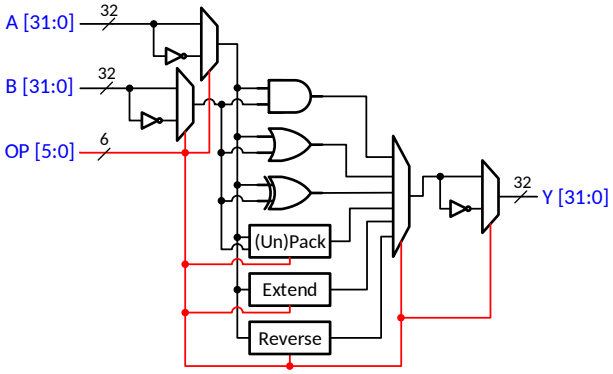


Fig. 10: Brief illustration of the Logic Block Design

carries are equal. Otherwise, the unsigned value is saturated to the largest unsigned positive number in case of addition and to zero in case of subtraction.

The logic block performs the basic logic operations. It is also designed to support packing/unpacking, extend, and reverse instructions. Packing/unpacking instructions deal with half-words and bytes to extract 16-bit and 8-bit operands from 32-bit words, or to combine several 16-bit and 8-bit operands into 32-bit words. Extend instructions can extend signed and unsigned byte or halfword into a 32-bit operand. Reverse instructions are able to change the order or bytes, halfwords, or bits within a word of data. A brief illustration of the logic block is depicted in Fig. 10.

The logic unit is not used in arithmetic operations while, the adders are not used in logic, packing/unpacking, extend, and reverse operations. Therefore, the different parts are designed to be switched off when they are not in use to reduce power consumption. Guard evaluation low-power technique [8] is used to block the change in inputs to these blocks; Hence, saving dynamic power due to transitions.

C. Barrel Shifter

The second operand passes through the barrel shifters before it enters the ALU. The barrel shifter is capable of performing five different shift operations which are: logic shift right, logic shift left, arithmetic shift right, rotate, and rotate with carry. The number of shift positions ranges between 0 and 32 and is defined by a register based operand or immediate operand. Each shift operation is performed by a specific combinational block. The operand is applied to the five blocks at the same time and the desired output is selected by a multiplexer whose selection is controlled by the instruction decoder. Guard evaluation low-power technique is used to switch off the blocks that are not in use according to the instruction.

D. MAC Unit

The multiplicand and multiplier operands are denoted by A and B, respectively, while the third operand is denoted by R, and the result is denoted by Y. For demonstration, the main idea of the proposed MAC unit operations is listed below:

The proposed MAC unit consists of four 16x16 bit array multipliers whose inputs and outputs are connected to adders and multiplexers. Fig. 11 shows the block diagram of the proposed MAC unit and demonstrates its dataflow architecture.

– **Multiply words and Accumulate**
$$Y = R \pm A \times B$$
– **Multiply Halfwords and Accumulate**
$$Y = R \pm A(\text{halfword}) \times B(\text{halfword})$$
– **Multiply Word by Halfword and Accumulate**
$$Y = R \pm A \times B(\text{half word})$$
– **Dual Multiply Halfwords, Add/Subtract, Accumulate**
$$Y = R \pm A(\text{halfword}) \times B(\text{halfword}) \pm A(\text{other halfword}) \times B(\text{other halfword})$$

For 16-bit MAC operations, only one multiplier is required to multiply two halfwords. In case of dual 16-bit operations, two multipliers are involved in multiplying two pairs of standalone halfwords and their result is then added or subtracted. The four multipliers are connected together along with adders to form a 32x32 bit vedic multiplier in 32-bit operations. The result of multiplication is optionally accumulated on another operand by means of a 32-bit and/or a 64-bit adder.

Multiplexers are responsible for choosing the desired inputs and outputs of the multipliers and adders according to the operation. They are used to select the desired words and halfwords of the input and resulting operands. In addition, select the type of multiplication, whether signed or unsigned. In this way, the same hardware components can be reused to perform a wide range of operations while consuming less area and power by avoiding component replication.

The multipliers used are unsigned by default, and hence, an extra hardware is required to perform signed multiplication. For unsigned multiplication, the operands are allowed into the multipliers without any change. However, for signed multiplication, the absolute value of operands is fed into the multipliers, unsigned multiplication is performed, and finally the sign is separately calculated and added to the result.

The multiplier blocks, denoted by ‘M0’, ‘M1’, ‘M2’ and ‘M3’, are 16x16 bit unsigned array multipliers that use digital combinational circuits to perform parallel multiplication. Array multipliers outperform serial multiplication schemes in terms of speed and performance. The design of an array multiplier is based upon partial product generation, shifting and addition. The partial product is generated by the multiplication of the multiplicand with one multiplier bit. Each partial product is shifted according to its bit position. Finally, the result is obtained by adding the shifted partial products.

In order to maximize the performance while maintaining minimum power and area and enable hardware reuse, the vedic scheme is used to construct a 32x32 bit multiplier. Fig. 12 shows an example of 4x4 bit vedic multiplier. The same methodology can be extended to construct 32x32-bit vedic multiplier [9].

In the proposed design, ‘M0’ is used to multiply the lower halfwords of the input operands. ‘M3’ is used to multiply the upper halfwords. The other two multipliers ‘M1’ and ‘M2’ are used to multiply the lower halfword of the first operand by the upper halfword of the second operand and vice-versa.

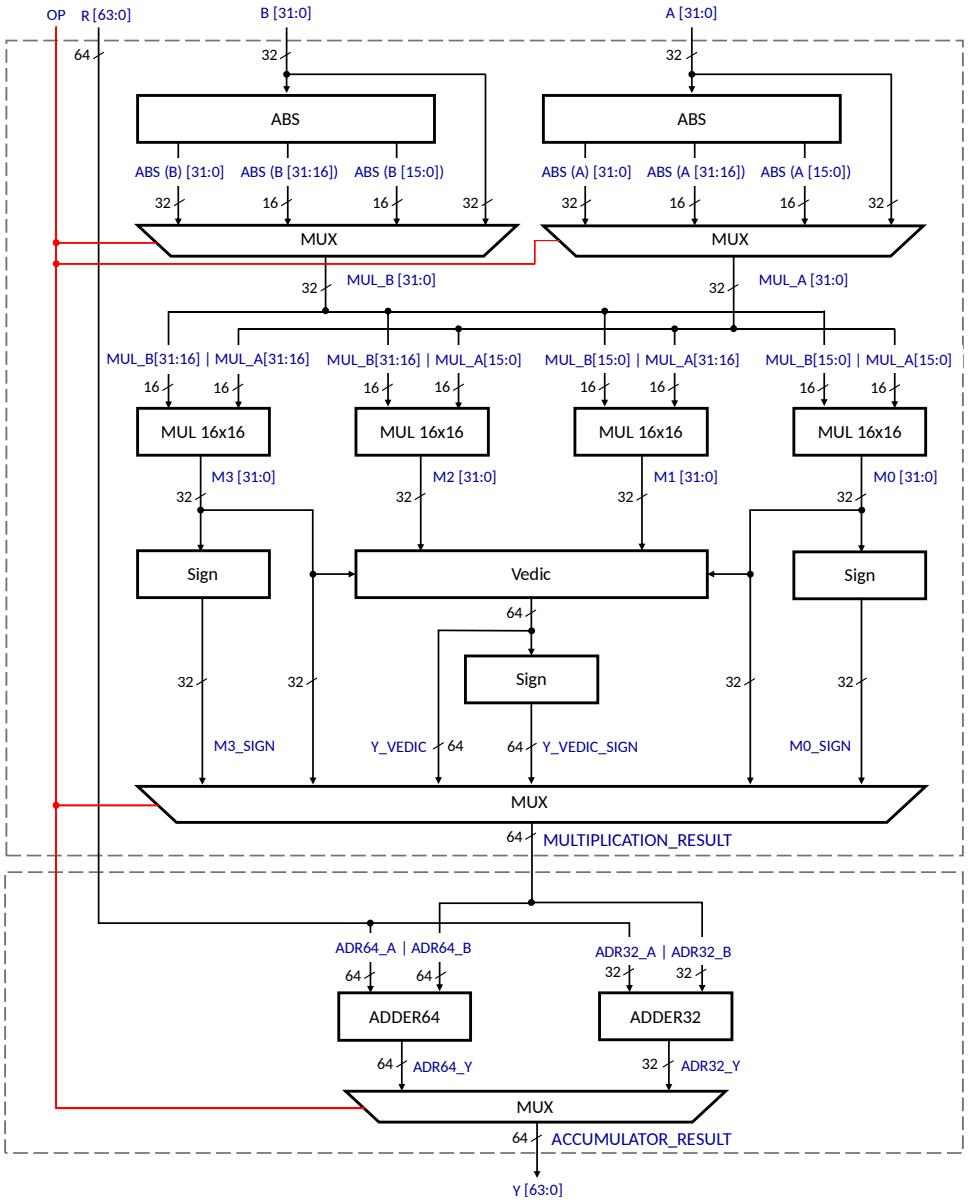


Fig. 11: The block diagram of the proposed MAC unit.

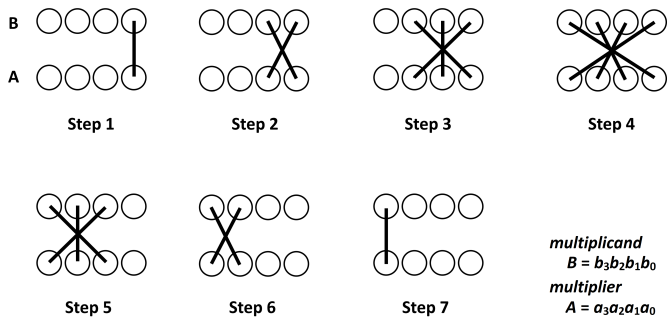


Fig. 12: vedic scheme for 4x4-bit multiplier.

In this way, the result of multiplying the different halfwords is obtained at the same time and can be used in operations where two standalone multiplication results are required. The result selection depends on the operation and is done by means of multiplexers. In order to use the same hardware to perform 32x32 bit multiplication, the concept of vedic multiplication was used. The results of the four multipliers are connected to adders to extend the multiplication process. Fig. 13 depicts the connection of adders in the vedic block.

The 'ABS' block is used to obtain the absolute value of the signed operands and is required only in signed multiplication operations. To obtain the absolute value, the most-significant bit of the operand is inspected. If it was '1' then the operand is negative and the absolute value is the two's complement of the operand. Otherwise, the operand is positive and the absolute value is equal to the operand itself. The obtained absolute value is ready to be used with the unsigned multipliers. Fig. 14 depicts the design of both the 'ABS' block.

The 'Sign' block is required to calculate the sign of the result in signed multiplication. The resulting operand should be negative only if A and B were of different signs. In

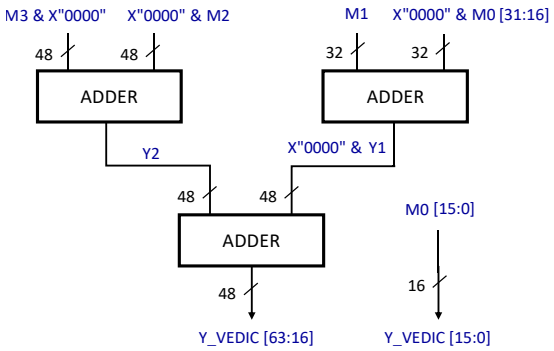


Fig. 13: The design of the 'Vedic' block.

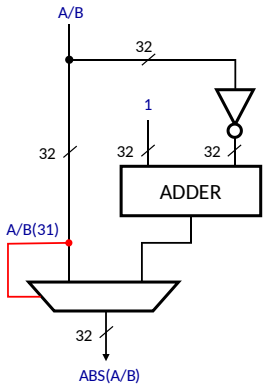


Fig. 14: The design of the *ABS* block.

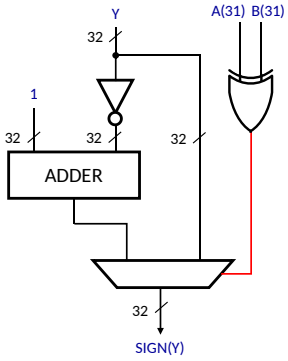


Fig. 15: The design of the 'Sign' block.

order to achieve this, the most-significant bit of A and B are inspected. The signed result is the two's complement of the unsigned multiplication result when the most-significant bits are different. Otherwise, the signed result is equal to the unsigned multiplication result. Fig. 15 depicts the design of both the 'Sign' block.

Regarding the low-power consumption, one MAC operation does not use all the components at a time. Only some specific components are necessary to operate when executing a certain operation and the rest can be switched off. Therefore, guard evaluation low-power technique is used again to block the change in inputs to these blocks.

E. Control Unit

The control unit includes the instruction decoder and the logic circuit that controls the stages of the pipeline to prevent hazards. The instruction decoder is a combinational circuit which decodes the instruction into its components and accordingly, drive the control signals of the different components of the processor. On the other hand, the pipeline control circuit is based on a finite-state machine in which each state controls the three stages of the pipeline. Fig. 16 depicts the finite state machine representation of the control unit.

The processor starts at state 'C' whenever a reset signal is received. In this state, the decoding and execution latches are cleared and disabled. The fetch stage is enabled so that the processor can read a new instruction. In the following cycle, the control unit changes the state to 'D' to enable the decode stage. However, the instruction is still not ready to be executed. Finally, the control unit changes the state to 'A' in the following cycle to enable the execution stage. Unless a branch, load/store or conditional instruction or a reset signal is received, the control unit remains in state 'A'.

When a branch instruction reaches the execution stage, there should be two other instructions waiting in the pipeline. These two instructions should be flushed away in order to prevent them from being executed. Therefore, for unconditional branch, the processor enters state 'B' to execute the branch instruction, then changes state to 'C' to clear the pipeline. For conditional codes, the processor moves to state D in case the condition was not satisfied to stall the pipeline and skip executing the instruction. Otherwise, it will remain in state 'A' or enter state 'B' in case of conditional branch.

IV. Simulation and Results

The proposed processor was implemented using VHDL using Vivado software tool by Xilinx. The testing is divided into two parts: a simulation part where all the testing is done on the simulation tool ISim integrated with Vivado, and a

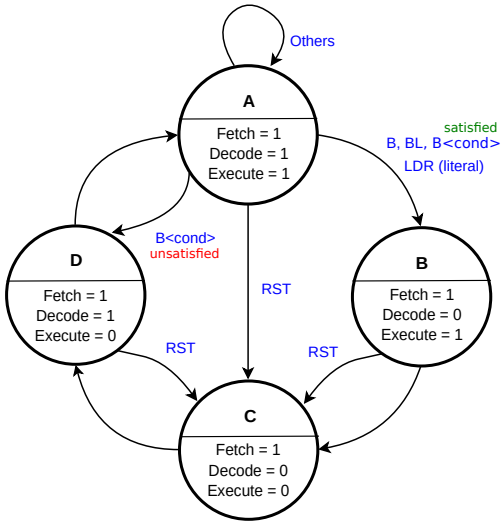


Fig. 16: The finite-state machine of the control unit.

hardware part where the implemented design is uploaded on Digilent Nexys 4 FPGA development kit. The estimated power report of the processor was obtained from Xilinx Vivado.

The on-chip power report of the processor generated at clock speed of 100MHz is shown in Figure 17. The report defines the estimated values for the dynamic power and static power. Static power, as well known, is the power consumption in the steady state resulting from intrinsic leakage of transistors. The static power is a constant value of 91 mW representing 42% of the total power. The internal switching activity and the clock frequency influencing directly the dynamic power. The dynamic power of the whole design is about 129mW which is 52% of the total power consumption. These results reflects that the proposed low-power 32-bit RISC processor provides an ultimate solution for battery-powered IoT applications.

For the simulation part, a test-bench was made for every component to separately test its functionality in simulation. In order to test the functionality of the processor as whole, a test-code with a wide range of instructions was preloaded into the instruction memory and the output of every component was verified after assembly. The processor simulation was performed at a clock speed of 100MHz. A small part of the test-code is listed below for demonstration.

```
MOV R1, #2
MOV R3, #3
ADD R2, R1, R3
SUBS R0, R3, R2 <LSL,#1>
BL <P>  0x00000000
MOV R0, #0
BL 0x000c0000
```

The test-code starts by moveing two immediate operands of value ‘2’ and ‘3’ into registers R1 and R3 respectively. Then, it adds the register value of R1 to that of R3 and writes the result into R2. The operand in R2 is then shifted by 1 position to the left and subtracted from the operand stored in R3. The result is written into R0 and the flags are updated. The the value of the ‘N flag’ is checked, and if the flag was clear (i.e. the result of the subtraction operation was positive,) the program branches to the first address in the instruction memory. If the condition was not satisfied, the program will move an operand of ’0’ into R0 and finally, branch unconditionally to the address ‘0x000c0000’ linking the current instruction address to ‘SP’.

Figure 18 shows the obtained simulation results. According to the figure, the control unit started at state ‘C’ after receiving

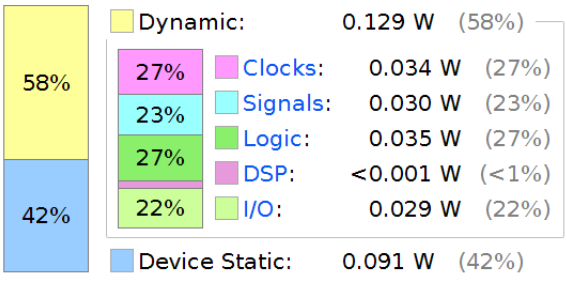


Fig. 17: The Power Report of the Processor

the simulated reset signal. The first immediate operand ‘2’ was written into the register R1 by the end of the third cycle as expected. Then, the second immediate operand ‘3’ was written into R3. The two register values were added resulting in an operand of value ‘5’ and the operand was stored in R2. The shifted value of R2 was then subtracted from R3 resulting in a negative value ‘FFFFFFF8’ which is written to R0 and the N flag was set. In the next cycle, the branch condition was not satisfied and the control unit changed the state to ‘D’ to disable the execution latch. In the following cycle, an immediate operand of ‘0’ was written into R0. Finally, the processor branched to address ‘0x000c0000’ after writing the current instruction address ‘0x00000001C’ into SP. The control unit changed the state to ‘B’ in order to execute the branch, adding two empty bubbles to refill the pipeline.

V. Conclusions and Future Work

Efficient hardware architecture for low-power 32-bit RISC embedded processor has been designed, implemented and tested in this work to fit the IoT power budget. The proposed architecture is implemented using VHDL featuring Xilinx’s FPGA. The DSP performance is improved by designing 32-bit MAC unit optimized for low-power operations which is capable to execute all operations in one cycle. A testbench was created for every component of the proposed design to separately test its functionality on ISim simulator integrated on Vivado. The whole processor is tested on Nexys 4 development board using some test codes with wide range of instructions. The test results obtained from simulation show that power consumption is very low at clock frequency of 100 MHz. Future work will focus on improving the processor capabilities by including deep-sleep mode and some encryption modules to fulfill the need of security measures.

References

[1] Pallavi Sethi and Smruti R. Sarangi. “Internet of Things”. In: *JECE* 2017 (2017), pp. 6–. ISSN: 2090-0147. DOI: 10.1155/2017/9324035.

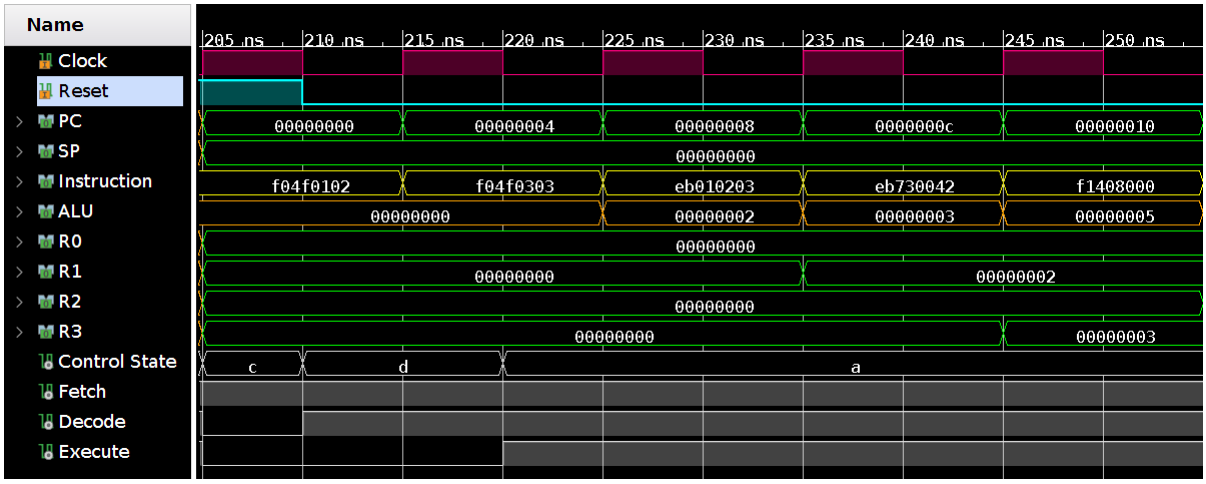
[2] Sarah Harris and David Harris. *Digital Design and Computer Architecture: ARM Edition*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2015. ISBN: 0128000562, 9780128000564.

[3] Andrew Sloss, Dominic Symes, and Chris Wright. *ARM System Developer’s Guide: Designing and Optimizing System Software*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004. ISBN: 1558608745.

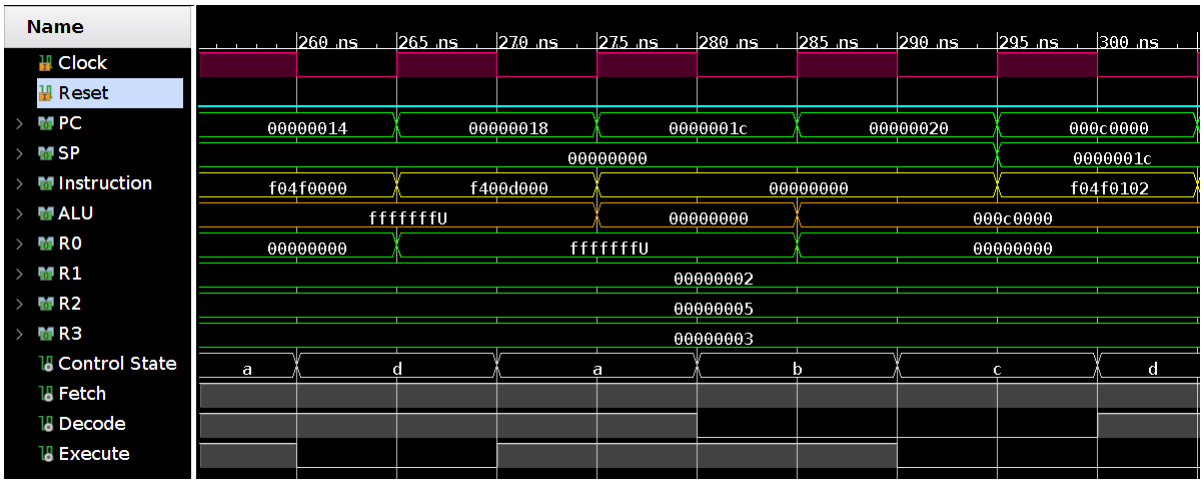
[4] Randal E. Bryant and David R. O’Hallaron. *Computer Systems: A Programmer’s Perspective*. 3rd. Pearson, 2015. ISBN: 0134092669, 9780134092669.

[5] Rajkumar Buyya and Amir Vahid Dastjerdi. *Internet of Things: Principles and Paradigms*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016. ISBN: 012805395X, 9780128053959.

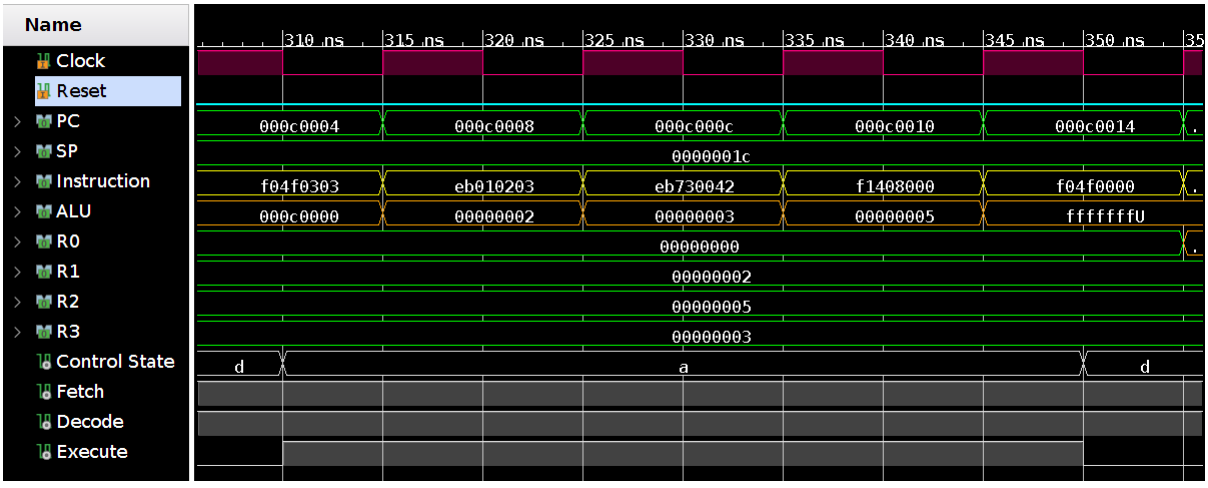
[6] Claire Rowland et al. *Designing Connected Products: UX for the Consumer Internet of Things*. 1st. O’Reilly Media, Inc., 2015. ISBN: 1449372562, 9781449372569.



(a)



(b)



(c)

Fig. 18: The Simulation Results of the Processor

[7] Joseph Yiu. *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors, Third Edition*. 3rd. Newton, MA, USA: Newnes, 2013. ISBN: 0124080820, 9780124080829.

[8] Vivek Tiwari, Sharad Malik, and Pranav Ashar. "Guarded Evaluation: Pushing Power Management to Logic Synthesis/Design". In: *Proceedings of the 1995 International Symposium on Low Power Design*. New York, NY, USA: ACM, 1995, pp. 221–226. ISBN: 0-89791-744-8. DOI: 10.1145/224081.224120.

[9] Yogita Bansal and Charu Madhu. "A Novel High-speed Approach for 16X16 Vedic Multiplication with Compressor Adders". In: *Comput. Electr. Eng.* 49 (2016), pp. 39–49. ISSN: 0045-7906. DOI: 10.1016/j.compeleceng.2015.11.006.

APPENDIX A
INSTRUCTION SET ARCHITECTURE OF THE PROPOSED PROCESSOR

The processor is capable of performing several operations, which include:

- Conditional and unconditional branching
- Load/store instructions
- Logical and arithmetic shifting in both directions, and rotation
- Addition and subtraction (with and without carry)
- Short multiplication instructions (Long multiplication instructions are the same, but with double the operand size)
- Logic instructions

The following tables summarize some of the most important instructions:

TABLE II: Branch Instructions

Instruction	Function
Unconditional Branch (B)	Loads PC with an address obtained from a register value or an immediate operand to execute a subroutine .
Branch and Link (BL)	Saves the current PC to SP then loads PC with an address obtained from a register value or an immediate operand .
Conditional Branch (B <cond >)	Branches only if a condition is satisfied. Status flags are used to check if the condition is true or false. Conditions can be Equal, Not Equal, Greater than, Less than, Negative, Positive, Zero, Overflow, etc

TABLE III: Load/Store Instruction Operations

Instruction	Operation
LDR (Immediate)	$R_t = M\left[R_n \pm \#imm\right]$
LDR (Register)	$R_t = M\left[R_n \pm LSL\left(R_m, \#imm\right)\right]$
LDR (Literal)	$R_t = M\left[R_n \pm PC\right]$
STR (Immediate)	$M\left[R_n \pm \#imm\right] = R_t$
STR (Register)	$M\left[R_n \pm LSL\left(R_m, \#imm\right)\right] = R_t$

TABLE IV: The types of Barrel Shifter Operations

Instruction	Operation	Flags
LSL <n>	Logic Shift Left by n-bit positions.	N C Z
LSR <n>	Logic Shift Right by n-bit positions.	N C Z
RSR <n>	Arithmetic Shift Right by n-bit positions.	N C Z
ROR	The least significant bit is rotated off the right end and inserted into the most significant bit position on the left.	N C Z
RRX	Shift right one bit with the old carry inserted in the most significant bit.	-

TABLE V: Signed/Unsigned Basic and Parallel Arithmetic Instructions (optionally saturated or divided by 2)

Instruction	Operation	Flags
ADD SUB	Word Addition/Subtraction $R_d = R_n \pm R_m$	N, C, Z, V
ADC SBC	Addition/Subtraction with Carry $R_d = R_n \pm R_m + carry$	N, C, Z, V
ADD16 SUB16	Half-word Addition/Subtraction $R_d\left[31:16\right] = R_n\left[31:16\right] \pm R_m\left[31:16\right]$ $R_d\left[15:0\right] = R_n\left[15:0\right] \pm R_m\left[15:0\right]$	G
ASX SAX	Exchanged Half-word Addition/Subtraction $R_d\left[31:16\right] = R_n\left[31:16\right] \pm R_m\left[15:0\right]$ $R_d\left[15:0\right] = R_n\left[15:0\right] \mp R_m\left[31:16\right]$	G
ADD8 SUB8	Byte Addition/Subtraction $R_d\left[31:24\right] = R_n\left[31:24\right] \pm R_m\left[31:24\right]$ $R_d\left[23:16\right] = R_n\left[23:16\right] \pm R_m\left[23:16\right]$ $R_d\left[15:8\right] = R_n\left[15:8\right] \pm R_m\left[15:8\right]$ $R_d\left[7:0\right] = R_n\left[7:0\right] \pm R_m\left[7:0\right]$	G
QADD QSUB	Signed Saturate Add/Subtract $R_d = \text{SAT } R_n \pm R_m$	Q
QDADD QDSUB	Signed Saturate, Double, Add/Subtract $R_d = \text{SAT } (R_n \pm \text{ SAT } (R_m))$	Q

TABLE VI: MAC Unit Instructions

Instruction	Operation
MUL, MLA, MLS	(Un)Signed Multiply and Accumulate/Subtract $R_d = R_a \pm R_n \times R_m$
(U/S)MULxy	(Un)Signed Multiply Accumulate Halfwords xy=BB: $R_d = R_a \pm R_n\left[15:0\right] \times R_m\left[15:0\right]$ xy=BT: $R_d = R_a \pm R_n\left[15:0\right] \times R_m\left[31:16\right]$ xy=TB: $R_d = R_a \pm R_n\left[31:16\right] \times R_m\left[15:0\right]$ xy=TT: $R_d = R_a \pm R_n\left[31:16\right] \times R_m\left[31:16\right]$
(U/S)MULWx	(Un)Signed Multiply Accumulate Word by Halfword x=B: $R_d = R_a \pm R_n \times R_m\left[15:0\right]$ x=T: $R_d = R_a \pm R_n \times R_m\left[31:16\right]$
SMUADx	Signed Dual Multiply, Add or Subtract, Accumulate $R_d = R_a \pm R_n\left[15:0\right] \times R_m\left[15:0\right] \pm R_n\left[31:16\right] \times R_m\left[31:16\right]$ $R_d = R_a \pm R_n\left[15:0\right] \times R_m\left[31:16\right] \pm R_n\left[31:16\right] \times R_m\left[15:0\right]$

TABLE VII: Basic Logic Instructions

Instruction	Operation	Flags
AND	$R_d = R_n \wedge R_m$	N, C, Z
BIC	$R_d = R_n \wedge (\neg R_m)$	N, C, Z
ORR	$R_d = R_n \vee R_m$	N, C, Z
ORN	$R_d = R_n \vee (\neg R_m)$	N, C, Z
EOR	$R_d = R_n \oplus R_m$	N, C, Z

TABLE VIII: Packing and Unpacking Instructions

Instruction	Operation
SEL	Select Byte If GE = 1, $R_d = R_n$, Else, $R_d = R_m$
(S/U)XTAH	(Un)Signed Extend Add Half-wrod $R_d = R_n +$ Ext($R_m\,[15:0]$)
(U/S)XTAB16	(Un)Signed Extend Add two bytes $R_d\,[31:16] = R_n\,[31:16] +$ Ext($R_m\,[23:16]$) $R_d\,[15:0] = R_n\,[15:0] +$ Ext($R_m\,[7:0]$)
(U/S)XTAB	(Un)Signed Extend Add Byte $R_d = R_n +$ Ext($R_m\,[7:0]$)

TABLE IX: Miscellaneous Instructions

Instruction	Operation
REV	Reverse Bytes in a Word
REV16	Reverse Bytes in Each Half-words
REVSH	Reverse Bytes in the Lower Half-word and Sign Extend the Result
RBIT	Reverse Bits in a Word
CLZ	Count Leading Zeros

Kareem Mansour is currently a graduate student pursuing his MSc degree in Microsystems Engineering at the microsystems engineering department (IMTEK), Albert-Ludwigs-Universität Freiburg, Germany. He received his BSc degree from Future University in Egypt in Electronics and Communication Engineering in 2018. Kareem is interested in digital system design, integrated circuit design and fabrication and robotics. He has several skills in programming and CAD design. His graduation project was about a design and implementation of a low-power embedded processor on FPGA. In 2017, he was a trainee at Ingeniarius Lda, Portugal, where he worked on several projects using Robot Operating Systems in the laboratories of Coimbra university. Kareem was an active member in the CHEF (Club of Hobby Electronics at Future University) where he gave some short tutorials to the participating students in Arduino. He is also a member in the Egyptian Syndicate of Engineers.

Ahmed Saeed is assistant professor in the electrical engineering department, Future University in Egypt. He received his PhD in Electronics Engineering from Ainshams University and his BSc and MSc in Communications and Electronics Engineering from Helwan University, Egypt. He was a visiting researcher at System and Architecture Lab in Faculty of Engineering, University of Porto, Portugal in 2014. Saeed was visiting staff at the School of Engineering, University of Central Lancashire, UK and Faculty of Engineering, University of Porto, Portugal. Ahmed assisted and supervised a lot of graduation/commercial projects in the field of signal processing, communication systems implementation, and digital system design. He is member in the organization/steering committee for the scientific meetings and events held in Faculty of Engineering. His research interests include low-power implementation of IoT, signal processing functions and wired/wireless communication systems on FPGA and ASIC, signal analysis and modeling in high-speed interconnects, signal integrity. Ahmed is member in IEEE and Egyptian Syndicate of Engineers. He published some papers in efficient implementation of the DSP blocks on an FPGA.