electronics

MDPI

Article

Design and Evaluation of Open Source Soft-Core Processors

Mario Gazziro[1,2], Jecel Mattos de Assumpção Junior[2], Oswaldo Hideo Ando Junior[3], Marco Roberto Cavallari[4] and João Paulo Carmo[2,*]

1 Information Engineering Group, Department of Engineering and Social Sciences (CECS), Federal University of ABC (UFABC), Av. dos Estados, 5001, Santo André 09210-580, Brazil; mario.gazziro@ufabc.edu.br (M.A.G.)
2 Group of Metamaterials Microwaves and Optics (GMeta), Department of Electrical Engineering (SEL), University of São Paulo (USP), Avenida Trabalhador São-Carlense, Nr. 400, Parque Industrial Arnold Schimidt, São Carlos 13566-590, Brazil; jcarmo@sc.usp.br (J.P.C.)
3 Academic Unit of Cabo de Santo Agostinho (UACSA), Federal Rural University of Pernambuco (UFRPE), Cabo de Santo Agostinho 54518-430, Brazil; oswaldo.ando@ufrpe.br (O.H.A.)
4 Department of Electronics and Biomedical Engineering (DEEB). School of Electrical and Computer Engineering (FEEC), State University of Campinas (UNICAMP), Campinas 13083-852, Brazil; mrcavall@unicamp.br (M.R.C.)
* Correspondence: jcarmo@sc.usp.br

Abstract: The advantage of FPGAs lies in the ability to implement a fully hardware solution for interfacing with various input/output (I/O) devices. Each block can work in parallel with all the others, simplifying the satisfaction of timing constraints. However, this hardware utilization consumes FPGA resources that could otherwise be allocated to the primary project. An alternative involves employing a small "soft-core" processor implementing I/O in software. With the goal to design and evaluate a new tiny soft-core processor optimized for FPGA resources in I/O , a novel processor named Baby8 is developed. It is an 8-bit CISC soft-core processor optimized for reduced FPGA resources including program size for 8-bit applications. The number of instructions is not large but any instruction can access arbitrary memory locations. The performance and resource utilization of the newly designed processor are evaluated and compared with a variety of other soft-core processors. The results demonstrate its competitive performance, achieving an average maximum clock frequency of approximately 57 MHz and a power consumption of around 2mW. Furthermore, it conserves nearly half of the FPGA resources in implementation.

Keywords: optimization; soft-core; processor; FPGA; ASIC; RISC-V; CISC.

Citation: Gazziro, M., Assumpção, J., Ando, O., Cavallari, M., Carmo, J. Design and Evaluation of Open Source Soft-Core Processors. Electronics 2024. https://doi.org

Received: date
Revised: date
Accepted: date
Published: date

1. Introduction

FPGAs offer the flexibility to implement a fully hardware solution for interfacing with diverse I/O devices, allowing blocks to operate in parallel, simplifying timing constraint fulfillment. However, the drawback is the consumption of FPGA resources that could be directed towards the main project. A viable alternative is employing a small "soft-core" processor for software-based I/O implementation. This approach saves resources in two ways: block RAM bits are smaller compared to separate flip-flops in configurable blocks, and a processor can time multiplex operations like adders, unlike the hardware solution with multiple unused copies. The performance only needs to be sufficient for protocol implementation, making the trade-off between fewer resources and speed. To achieve this goal, a new processor called Baby8, designed for FPGA I/O, is developed using Yosys Synthesis software [1][2].

Baby8 is a von Neumann 8-bit CISC soft-core processor (already available in author's GitHub repository in [3]) optimized for reduced FPGA resources, including program size for 8-bit applications. Although the instruction set is not extensive, each instruction can access arbitrary memory locations. Features like post-increment addressing, while not faster than RISC (instructions use simple register-level [4]), enhance program compactness and may reduce the need for block RAMs. The paper presents a comparison of Baby8 with almost a dozen other soft-core processors, demonstrating competitive performance in terms of implementation results, and comparisons with ASIC, NAND gates, and various FPGA models.

1.1. *ASICs*

Application Specific Integrated Circuits are those designed to be used in one particular product, as opposed to standard integrated circuits that are sold to many companies to be used in many products. While the cost of each ASIC is normally a fraction of the cost of an FPGA capable of handling the same design, the production of the ASIC has some high NRE (Non-Recurring Engineering) costs so a reasonably high volume is needed for the ASIC to be more viable economically than a FPGA. Additionally, ASICs may operate at higher frequencies and lower power than FPGAs, impacting product considerations. Comparing processor cores involves assessing area, operating frequency, and power usage for a given fabrication technology.

1.2. *NAND gates*

It is possible to build any digital circuits entirely with NAND logic gates or with just NOR gates (the case for the AGC, the Apollo Guidance Computer which landed on the Moon). Before the Field Programmable Gate Arrays (FPGAs) many projects used Gate Arrays. These were chips which had a large number of NAND gates, the same for all clients, and the metal layer was specific for each client. Translating designs to NAND gates provides insights into design complexity using the OnlyNANDYosysSynth complement script to Yosys Synthesis software.

1.3. *FPGAs*

Field Programmable Gate Arrays can implement any digital circuit up to a size that depends on the particular FPGA device Using hardware description languages like Verilog, VHDL, Chisel, SpinalHDL, and others, a high-level processor description can be translated into a netlist of basic blocks. The placement tool assigns blocks to specific locations, and routing utilizes the FPGA's configurable connection network. Routing uses the configurable connection network in the FPGA (a little like early telephone exchange systems) to actually connect the placed blocks. This is encoded into a "bit file" that is loaded by the FPGA each time it is turned on to actually implement the circuit.

This comparison considers the usage of basic blocks (LUT, Registers, DSP, Distributed memory, Block memory) in soft-cores when translated with the open-source tool Yosys002E. The basic blocks are:

- LUT: LookUp Tables implement all logic in FPGAs and can be categorized by the number of address lines they require. A LUT4 has 16 words of a single bit each and needs 4 bit addresses. A LUT6 has 64-bits and needs 6 address lines. A larger LUT can always do the job of a smaller one by either tying unused address lines to 0 or 1 or else duplicating the bits such that the output doesn't depend on that address line. Combining smaller LUTs into a larger one is facilitated by special "mux" blocks in some FPGAs (XC4000), enhancing efficiency.
- Registers: the LUTs are purely combinational , and an optional flip-flop circuit at the output enables the implementation of sequential circuits.. Normally one register is associated with one LUT, but there tend to be some extra registers as part of the I/O pads.
- DSP: Digital Signal Processing blocks are hardware implementations of multiplication circuits. Otherwise a very large number of LUTs would be

required to implement this operation (which has many more uses beyond digital signal processing).

- Distributed memory: Distributed memory: each LUT is actually a very small Random Access Memory (RAM), generally unaltered after the initial FPGA configuration. An additional circuit allows the use of all LUTs or a fraction as read/write memories).
- Block memory: the area needed to store a bit in a register or even in a LUT is very large compared to a dedicated RAM circuit. Since the 1990s FPGAs have included a number of memory blocks that can efficiently handle a medium to large number of bits.

Other FPGA blocks include input and output buffers, clock buffers, and carry circuits converting LUTs into adder circuits, along with multiplexers for combining small LUTs and FPGA-specific circuits unique to each type.

1.3.1. FPGA Families

Xilinx 7: The 28nm generation of Xilinx FPGAs remains popular, even after AMD's acquisition and the introduction of two subsequent generations. It is common for soft-core processor implementation projects to make use of FPGAs manufactured by Xilinx [5]. Cyclone V: The Cyclone family was the low end of Altera FPGAs, with the V generation being the last before Intel's acquisition of Altera. ICE40: The startup Silicon Blue took advantage of the expiration of key FPGA patents to introduce their own very basic offering. Their focus was on smaller FPGAs with low cost and low energy requirements. They were bought by Lattice Semiconductor and a second generation was designed moving from the original 65nm process to a more modern 40nm further reducing the energy use. GoWin: This Chinese company achieved success with various FPGA variations globally, introducing unique features like using spare LUTs to aid routing. How much this is used varies from one project to the next and this is why the numbers are not always relatively the same compared to other types of FPGAs. ECP5: An evolution of the earlier ECP, ECP2 and ECP3 FPGAs, the ECP5 family is a cost-effective mid-range option with high-speed serial interfaces.

1.4. *RISC-V Soft-cores*

The rising popularity of RISC-V [6] in the industrial and academic space resulted in a plethora of open-source RISC-V implementations [7][8]. Openness of the standard does not guarantee openness of specific cores, but those considered in this paper are open In addition, only cores implemented in Verilog or the subset of System Verilog handled by Yosys were considered. RISC-V cores have a tremendous range in performance and complexity [9], from tiny 32-bit microcontrollers to out-of-order 64-bit application cores for datacenters. Only the low end was studied here. darkRISC-V: The original darkRISC-V was created by Samsoniuk in a single night of development to evaluate the advantages and disadvantages of the RISC-V instruction set compared to others. It can be optionally made smaller by reducing the number of registers as per RV32E. vexRISC-V: The vexRISC-V in SpinalHDL is meant to show off the advantages of using that language, with many configuration options where it is even easy to change the number of pipeline stages. Many projects use the translated Verilog version of the processor, like the management system in the repository for GlobalFoundries 180nm version of the Caravel "harness" for open-source ASIC design. Glacial: Glacial trades off performance for size by being an 8-bit processor which emulates a 32-bit RISC-V. One inspiration was the low end of the original IBM S/360 family which used microcode and an 8-bit datapath to implement the architecture. PicoRV32: An early compact RISC-V core, the goal of the PicoRV32 was to fit in the tiny ICE40 FPGAs first targeted by the ICE Storm open-source FPGA tool, of which Yosys was a key component. SERV: SErial Risc V, also trades off performance for size, in this case by having a completely serial implementation. This means that any operation requires 32 clock cycles as the operation handles only a single bit in each cycle. Serial computers were a little more common in the days of vacuum tubes when every single component had a significant cost and added to the construction cost as well. They became far rarer in the integrated circuit days but are one way to save FPGA resources best left for other parts of a project.

1.5. *Other Soft-cores*

Even with all variation possible with the RISC-V instruction set, there are applications where other designs are a better option. That is particularly true when executing programs. Baby8: Designed to help interface to adapt FPGA projects to specific boards, the goal of **Baby8** is to use as few FPGA resources as possible to leave more for the user's actual project (available in author's GitHub repository in [3]). An 8-bit architecture is a good match for the applications of interfacing to PS/2 or USB keyboards, mice and game controllers as well as providing an abstract interface to files on a FAT32 formatted SD memory card. In an FPGA, distributed memory built from LUTs is denser than individual flip-flops. **Baby8** capitalizes on this density, albeit at the cost of reduced performance, by incorporating the program counter into the register bank.. In an ASIC, there is no advantage in doing this, as the flip-flops would be the same either way. The nomenclature of the processor was chosen in honor of the Manchester Baby, an 32-bit Small-Scale Experimental Machine, the first electronic stored-program computer in 1948 [10]. A complete description of the features and particularities of the **Baby8** processor will be presented in the methodologies section of this paper. 6502 and UKP: A NES (Famicom) emulator for the Sipeed Tang Nano 20K FPGA board includes two processors. The 6502 is needed to run the actual games while the limited UKP handles USB mice, keyboards or game controllers. Femto16: In the 8-Bit Workshop online video game development system there is an option to design games at the hardware level using Verilog. The examples grow in complexity, introducing two simple processors, the 8-bit Femto8 and the 16-bit Femto16. Games are then converted from pure hardware to assembly programs for these processors. J0: Describing the J1 soft core optimized for small programs in the Forth language was the inspiration for projects like SwapForth by the same author and the Forth CPU computer system. The Gameduino project adds FPGA-based video output for Arduino boards, incorporating the J0 processor as a slight modification of the J1. MCPU: With only 4 instructions and addressing only 64 bytes of memory (similar to Xilinx PicoBlaze), the MCPU is remarkably small yet Turing complete . ZPU: The 'ZPU Avalanche' was designed to use the least FPGA possible while being fully compatible with all the GNU programming tools, including GCC. The concept revolves around treating C as more of a scripting language on an FPGA, with heavy processing handled by hardware blocks. The Avalanche project translated the original VHDL implementation to System Verilog. The System Verilog files were copied from the original repository in the top directory.

2. Objectives

The main goal is to design and evaluate a brand new soft-core processor optimized for reduced FPGA resources, including program size for 8-bit applications.

3. Materials and Methods

3.1. Processor Specifications

3.1.1. State Registers

The processor features 16 internal 8-bit state registers, as shown in Figure 1. While not accessible to programmers, exceptions include the registers W, X, Y and the external memory, which can be directly read and written by the user program. Flags Z (ZERO), N (NEGATIVE), C (CARRY) and V (OVERFLOW) do not persist between instructions but can be saved with TEST instructions.

The complete list of 8-bit state registers is provided in the following, where H denotes high-byte and L denotes low-byte in case of 16-bit addressing:

- **PH/PL:** 16-bit (8H/8L) program counter in normal execution mode;
- **MH/ML:** 16-bit (8H/8L) pointer for indirect “zero page” operands;
- **IH/IL:** 16-bit (8H/8L) program counter in interrupt mode;
- **LH/LL:** 16-bit (8H/8L) address saved in last call instruction;
- **ZH/ZL:** 16-bit (8H/8L) address of the “zero page” operand;
- **TH/TL:** 16-bit (8H/8L) timer to define the number of cycles to pause before next instruction;
- **K:** 8-bit single register for ’cascades’ - values between pairs of instructions;
- **W, X, Y:** 8-bit single registers accessible to the programmers for reading and writing.

3.1.2. Basic Syntax

The processor is designed with a two-address architecture, where one address indicates both the destination and one operand, and the second address specifies the other operand. The assembly language syntax is C-like, with an addition represented as:

rD += rS

3.1.3. K-Cascade

The "cascade" feature changes the destination to be the first operand of the following instruction. A sequence that is incremented by 'the value pointed to by location 20' looks like:

X += *20
W &= X

It can be implemented with cascade as:

K = X + *20
Y = W & K

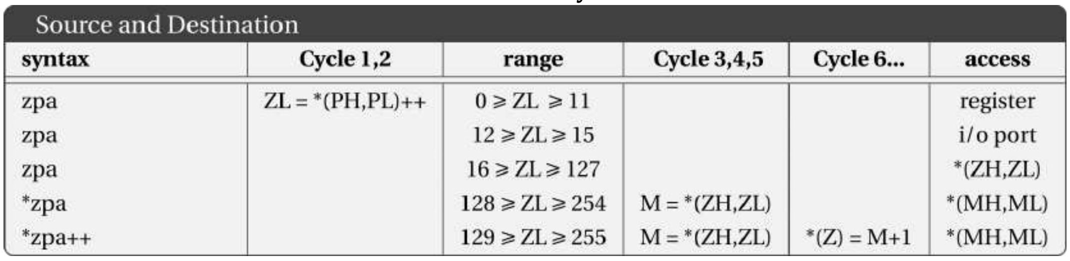
Unlike the original code fragment, these two instructions do not erase the value in X. While the architecture is generally two-address, it can function as three addresses (actually four) when needed. The use of “K” in the instruction distinguishes it from a move with some complicated expression to be calculated at assembly time.

3.1.4. Source and Destination

Table 1 shows the source and destination directives in the processor architecture. In theory, the zero page allows access to bytes 0 to 127 of the 256-byte page selected by register ZH. In practice, addresses 12 to 15 access the four input and four output ports. More I/O ports can be memory-mapped to other addresses if needed.

Zero page addresses 0 to 11 access the internal registers W to IH, allowing LL and LH to be saved and restored if more than one level of subroutines are needed. Since ZH can also be changed, the “zero page” can be relocated to any memory location, similar to the 6809 and 65816 processors. When TL is changed the timer is started if TL and TH are not both zero and it is stopped if they are zero.

Table 1. Source and destination directives in **Baby8** architecture.



Source and Destination

syntax	Cycle 1,2	range	Cycle 3,4,5	Cycle 6...	access
zpa	ZL = *(PH,PL)++	0 ⩾ ZL ⩾ 11			register
zpa		12 ⩾ ZL ⩾ 15			i/o port
zpa		16 ⩾ ZL ⩾ 127			*(ZH,ZL)
*zpa		128 ⩾ ZL ⩾ 254	M = *(ZH,ZL)		*(MH,ML)
*zpa++		129 ⩾ ZL ⩾ 255	M = *(ZH,ZL)	*(Z) = M+1	*(MH,ML)

The two-bit ***ss*** and ***dd*** fields (see Table 2) in the instruction use values 0, 1 and 2 to encode the registers W, X and Y respectively. A value of 3 indicates that a byte follows, encoding the actual address.

When both ***ss*** and ***dd*** are 3, the first extra byte is for the source, and the second is for the other source/destination. In the case of immediate instructions, the immediate value is the first extra byte, followed by the destination..

Next, Tables 2, 3, 4, and 5 will be presented, describing respectively:

- The basic operations opcodes;
- The immediate instructions opcodes;
- The control flow instructions opcodes;
- And the conditional tests.

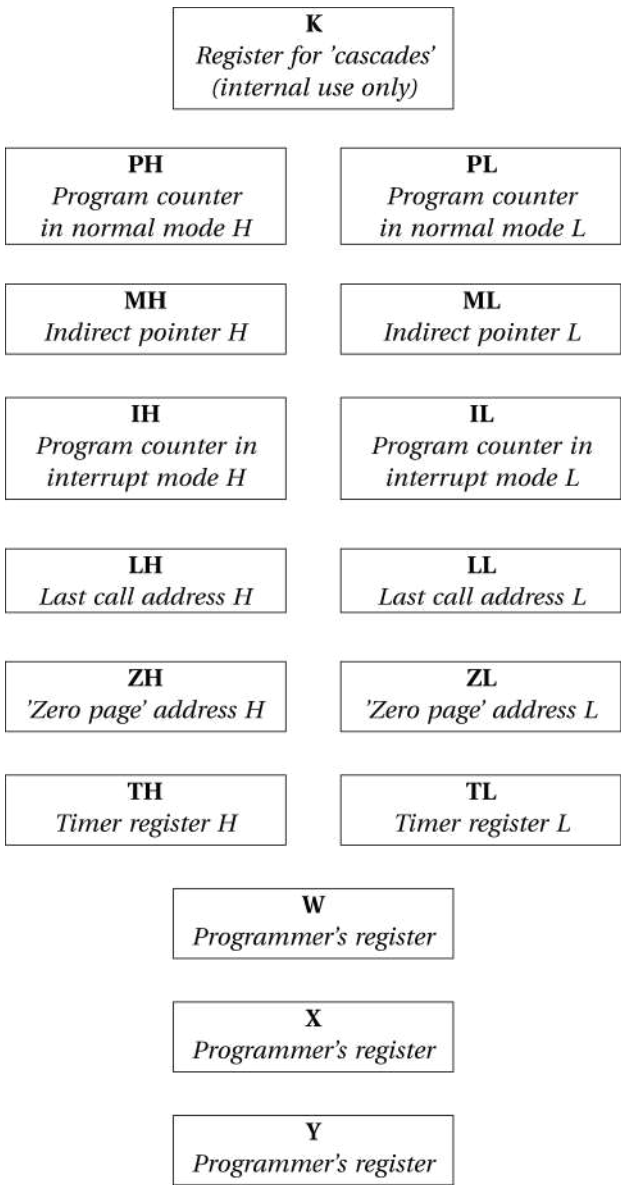


Figure 1. Baby8 processor 8-bit state registers. **H**igh and **L**ow forming 16-bit addresses.

Table 2. Regular and K-cascade basic instructions in **Baby8**.

Basic Instructions				
opcode	**binary**	**syntax**	**K-binary**	**K-syntax**
add	0000 ssdd	D += S	0001 ssdd	K = D + S
subtract	0010 ssdd	D -= S	0011 ssdd	K = D - S
move	0100 ssdd	D = S	0101 ssdd	K = D = S
test	0110 cccc	cond ?	0111 cccc	K = cond
and	1000 ssdd	D &= S	1001 ssdd	K = D & S
or	1010 ssdd	D |= S	1011 ssdd	K = D | S
exclusive or	1100 ssdd	D ˆ = S	1101 ssdd	K = D ˆ S
See next table	111f ffdd			

Table 3. Immediate instructions in **Baby8**.

Immediate Instructions		
opcode	**binary**	**syntax**
add	1110 00dd	D += #
subtract	1110 01dd	D -= #
move	1110 10dd	D = #
See next table	1110 11ff	
and	1111 00dd	D &= #
or	1111 01dd	D |= #
exclusive or	1111 10dd	D ˆ = #
See next table	1111 11ff	

Table 4. Control flow instructions in **Baby8**.

Control Flow Instructions			
opcode	**binary**	**syntax**	**internal operation**
jump	1110 1100	>>>> expr	PH,PL := ##
call	1110 1101	>>>>$ expr	LH,LL := PH,PL; PH,PL := ##
nop	1110 1110	~	
return	1110 1111	<<<<	PH,PL := LH,LL
jump	1111 1100	>>>> *zpa	PH,PL := *zpa
call	1111 1101	>>>>$ *zpa	LH,LL := PH,PL; PH,PL := *zpa
branch	1111 1110	>>>> #expr	PH,PL += #
table	1111 1111	<<<< zpa	PH,PL := [LH,LL + zpa]

Table 5. Conditional Tests in **Baby8**.

Conditional Tests							
conditional	**name**	**code**	**syntax**	**a.s.***	**code**	**syntax**	**a.s.***
Z	Equal	011k 0000	==	Z	011k 0001	!=	!Z
C	Greater equal	011k 0010	>=	C	011k 0011	<	!C
N	Negative	011k 0100	<0	N	011k 0101	>=0	!N
V	Overflow	011k 0110		V	011k 0111		!V
C&!Z	Greater	011k 1000	>		011k 1001	<=	
N==V	Signed greater equal	011k 1010	$>=		011k 1011	$<	
!Z&N==V	Signed greater	011k 1100	$>		011k 1101	$<=	
1	TRUE	011k 1110	true		011k 1111	false	

* ***alternative syntax***

3.1.5. Shifts and Rotations

The missing multiplication and division instructions are to be expected for a very small processor (though they can be added as an I/O device if needed), but the lack of shift and rotate instructions might seem limiting. Adding a value to itself is equivalent to a one-bit shift to the left, with the carry indicating the removed bit. Shifting to the right (logical or arithmetic) an 8-bit value by N bits can be achieved by extending (zero or sign) to 16-bits, shifting that to the left by 8-N bits and using the top byte as the result (Listing 1). Sending a byte with the least significant bit first might seem to need 7 such steps per bit, but the same result can be achieved by shifting a one-bit mask to the left once per step to check each bit from least to most significant (Listing 2).

```
; code to shift X right by 5 bits arithmetically
W = #0 ; default sign
K = X & #128
C ?
    W = #255 ; extended sign for V
W += W ; shift left top half
X += X ; shift left bottom half
C ?
    W += #1 ; move carry to bottom bit
W += W
X += X
C ?
    W += #1 ; second bit
W += W
X += X
C ?
    W += #1 ; third bit. Now W is X $>> 5
```

Listing 1. Shifts and rotations syntax example in **Baby8**.

```
    W = #1 ; bit mask, start at least significant
SendBit:
    K = X & W
    C ?
        >>>> sendOne
SendZero:
    ...
    W += W
    C ?
        >>>> sendBit
```

Listing 2. Shifts and rotations syntax example in **Baby8**.

3.1.6. Interrupt

The interrupt mode uses IH,IL for instruction fetches instead of PH,PL (bit 2 of the register addresses is 1), and zero page addresses go from 144 to 255 instead of from 16 to 127 (bit 7 of addresses is set). The interrupt mode is entered on the next instruction fetch after the interrupt line goes high unless the previous instruction was a cascade or the current instruction is supposed to be skipped. The interrupt mode ends on the next instruction fetch after the interrupt line goes low with the same restrictions as described before.

Most processors have interrupt handlers start at fixed addresses or, more often, indicated in some table. In Baby8, the address in IH,IL is simply the instruction following the one that caused the interrupt line to go low at the end of the previous interrupt. This is more like a coroutine scheme where an explicit "yield" instruction is executed. With careful programming it is possible to speed up response time by having the processor ready to execute in different places for different situations. Any registers used by the interrupt handler should be saved and then restored right before the yield. Having use of the second half of the zero page makes this easier. Note that the first half can still be accessed with indirect addressing.

3.1.7. Timer

Every instruction is executed in a known number of clock cycles, but to make precise delays even easier TH,TL implements a 16-bit timer. The timer mode is entered on the next instruction fetch after a write to TL which results in TH or TL having a non-zero value. The restrictions are the same as entering or leaving the interrupt mode.

Instead of fetching instructions the processor simply decrements TL by one on every clock cycle. When TH,TL reaches zero, instruction fetch resumes. If TH is not 0 but TL is, then TL is decremented by two (going to 254), and on the next cycle, TH is decremented. If an interrupt arrives while in the timer mode, the interrupt is handled, and execution is normal until the interrupt ends. The interrupt handler may reset TH,TL or decrease their values if encountering a longer delay than expected in the user code is an issue.

3.2. Custom Processor Design

An advanced digital design and simulation tool named DIGITAL, developed by Neemann, is employed for the design process. The Baby8 processor and its peripherals (RAM block and terminal) are shown in Figure 2, while its internal blocks are given in Figure 3.

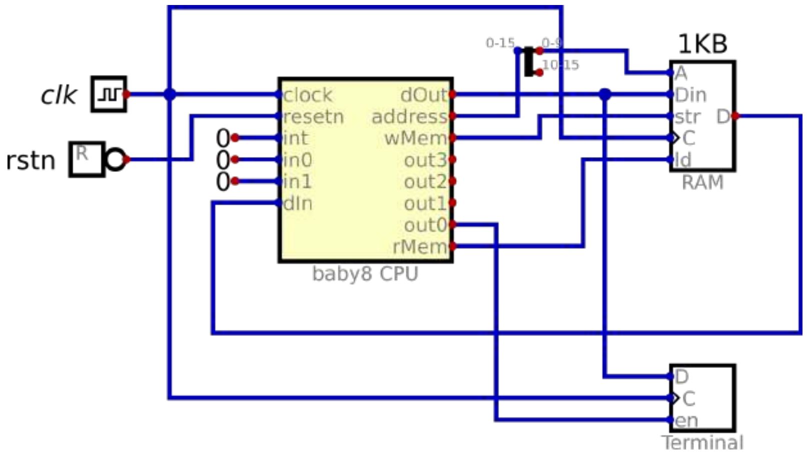


Figure 2. Baby8 CPU and its *peripherals RAM block and terminal*.

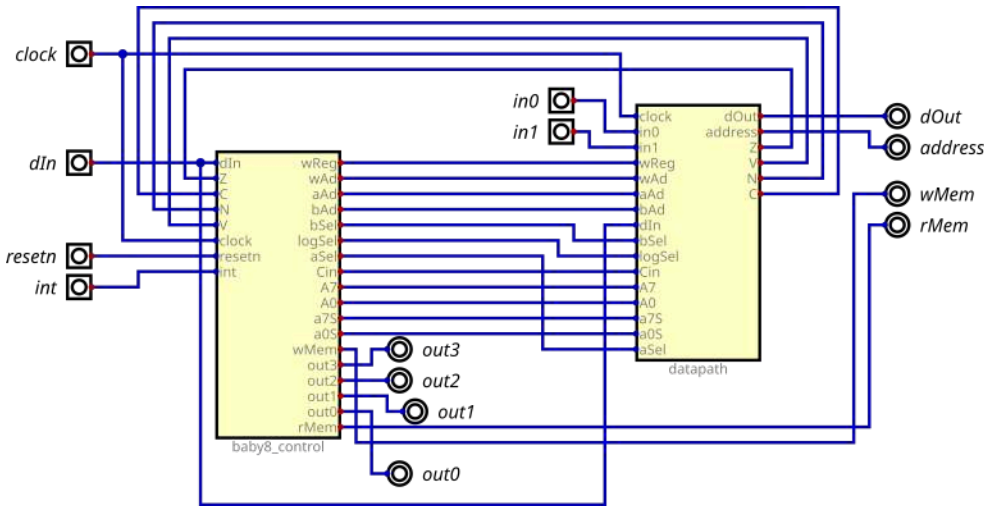


Figure 3. The combination of the DATAPATH and the CONTROL UNIT forms the CPU.

3.2.1 DATAPATH

The heart of the DATAPATH is the ALU (arithmetic logic unit). It can perform a logical operation (AND, OR, XOR or NOT) between 8-bit inputs ***A*** and ***B***, or modifying input ***A***. Figure 4 provides an in-depth view of the DATAPATH architecture and its internal components, including ALU, register bank, multiplexers, address modifier (Address Modification), and data output register.

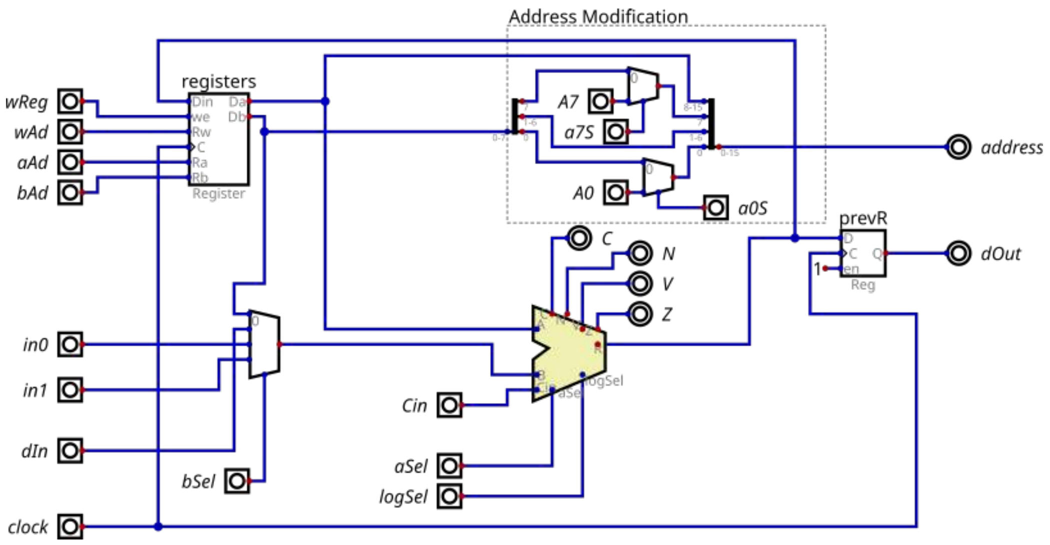


Figure 4. The DATAPATH defines the logic to execute the instructions.

During the native data flow, operand **B** of the ALU typically originates from the output ***Db*** of the register bank. However, it may eventually come from memory, or from one of the processor's two input ports. In turn, the ***Da*** and ***Db*** outputs of the register bank together form a 16-bit address. Within the region of combinational logic circuits called Address Modification, the binary values of the addresses are processed. Bit 7 indicates whether the address is direct or indirect, while bit 0 signifies whether it should be used or not incremented.

The address from memory is not always desired, and it can be created from internal operands and/or ALU operations. Logical address modification circuits also serve this purpose. The first 127 memory addresses are allocated for normal programs, and the range between 128 and 255 is utilized by routines called by interrupts. Additionally, a data output register is necessary to write to external memory, requiring 24 bits for this operation – 16 address bits and another 8 data bits, concatenated at the clock edge, using 2 clock cycles for this operation. Although this operation slightly reduces performance, it enables the utilization of fewer FPGA hardware resources, aligning with the primary objective.

3.2.2 CONTROL UNIT

The core of the Baby8 processor control unit is the state machine defined by the ROMfsm block in Figure 5. Internally, it implements the internal ROM generating control signals for decoding (microcode), as specified in Figure 6 and Table 6. This connected finite state machine generates signals needed to use the data path to execute instructions.

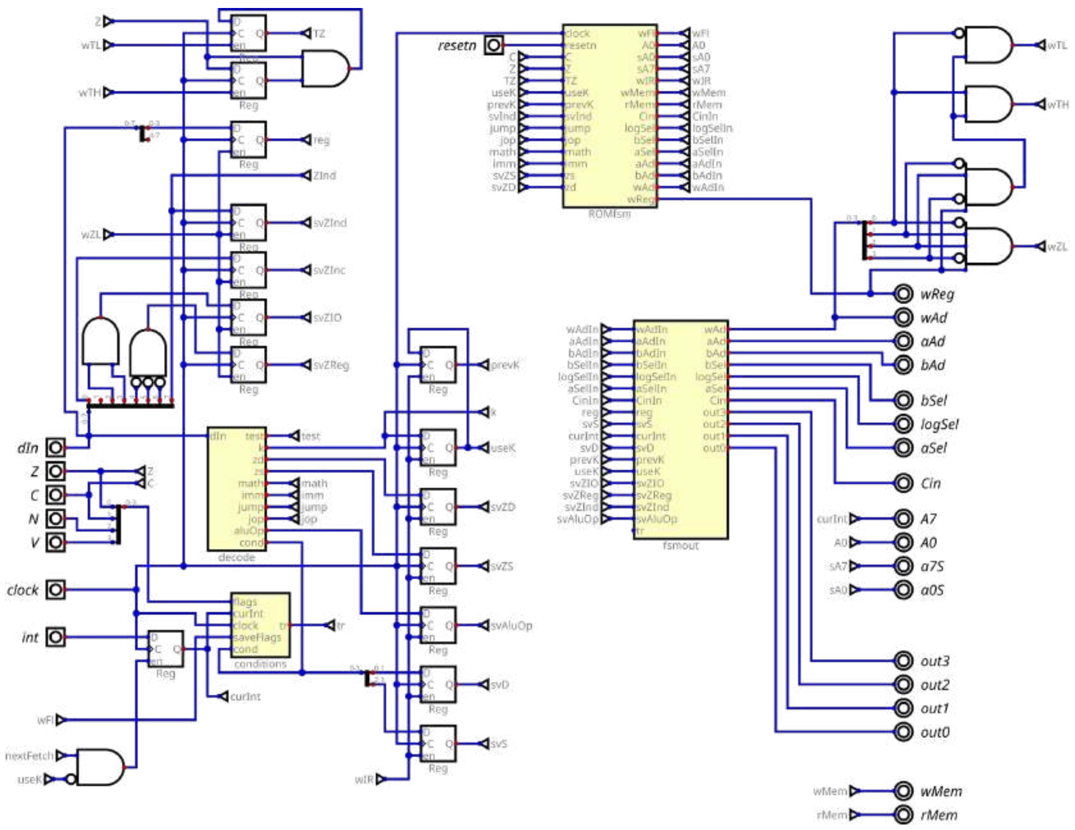


Figure 5. Decodification combinatorial circuit inside ***CONTROL*** unit.

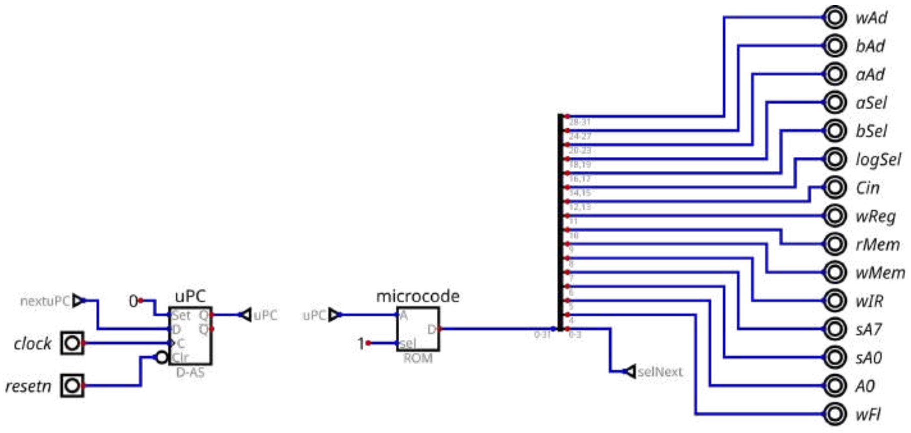


Figure 6. Internal details of the ***ROMfsm*** block, containing the ***uPC*** microprogram counter register and the microcode ROM whose contents are described in Table 6.

Table 6. Internal ROM generating control signals by decoding **OPCODES**.

Address (8-bits)	Control (32-bits)	Address (8-bits)	Control (32-bits)
0x00	0xE3308801	0x40	0x30380803
0x01	0xF3308801	0x41	#N/DISP
0x02	0x43308801	0x42	0x303A6812
0x03	0x53308800	0x43	0xFF30180A
0x04	0x0	0x44	0xFF30180A
0x05	0x0	0x45	0x303A6811
0x06	0x0	0x46	#N/DISP
0x07	0x0	0x47	0x3CD00200
0x08	0x44311800	0x48	0x3300800
0x09	0x55311801	0x49	0x3300000

0x0A	0x44310800	0x4A	0
0x0B	0x0	0x4B	0x8E300801
0x0C	0xEEF01C05	0x4C	0x9F300809
0x0D	0xFF30190F	0x4D	0xEC300801
0x0E	0x3330010F	0x4E	0xFD300800
0x0F	0x0	0x4F	0x0
0x10	0xEEF01C05	0x50	0xC0880805
0x11	0xFF30180E	0x51	0xD9301802
0x12	0x3330000E	0x52	0xD9300801
0x13	0xEE301805	0x53	0xCCD01C05
0x14	0xFF301801	0x54	0xE3340802
0x15	0xEE301800	0x55	0xE3340802
0x16	0xFF301800	0x56	0x3CD00402
0x17	0x0	0x57	0xDD301802
0x18	0x3330B800	0x58	0xF3340800
0x19	0x0	0x59	0x3CD00401
0x1A	0x21026810	0x5A	0xF3340800
0x1B	0x0	0x5B	0x0
0x1C	0x8E300801	0x5C	0x201A6810
0x1D	0x9F300801	0x5D	0x0
0x1E	0xEEF01C05	0x5E	#N/DISP
0x1F	0xC3340802	0x5F	0x3CD00200
0x20	0xC3340802	0x60	0x3300800
0x21	0xFF301801	0x61	0x3300000
0x22	0x3EF00401	0x62	#N/DISP
0x23	0xF3340801	0x63	0x0
0x24	0xEC300800	0x64	0x30380809
0x25	0x0	0x65	#N/DISP
0x26	0xE8300801	0x66	0x3CD00200
0x27	0xF9300800	0x67	0x3300800
0x28	0x0	0x68	0x3300000
0x29	0xEEF01C05	0x69	0x0
0x2A	0x13166812	0x6A	0x0
0x2B	0x13166810	0x6B	0x0
0x2C	0xFF301800	0x6C	0x0
0x2D	0x0	0x6D	0x0
0x2E	0x0	0x6E	0x0
0x2F	0x0	0x6F	0x0
0x30	0x0	0x70	0xEEF01C05
0x31	0x0	0x71	0x63340802
0x32	0x0	0x72	#N/DISP
0x33	0x0	0x73	#N/DISP
0x34	0x0	0x74	0x36700408
0x35	0x0	0x75	0x367004C1
0x36	0x0	0x76	0xC3340801
0x37	0x0	0x77	0x367004E1
0x38	0x0	0x78	#N/DISP
0x39	0x0	0x79	0x3C301005
0x3A	0x0	0x7A	0x367002C2
0x3B	0x0	0x7B	0x367002C3
0x3C	0x0	0x7C	0x3D301001
0x3D	0x0	0x7D	0x367002E1
0x3E	#N/DISP	0x7E	0x3CD00408
0x3F	#N/DISP	0x7F	0x0

3.2.3 ALU

In the design of the ALU, several factors were meticulously considered to achieve a balance between performance and resource efficiency. The initial step involved defining the logical and arithmetic operations necessary for the application. Another critical aspect focused on the format of the operands to be manipulated.

Ultimately, priority was given to minimizing the area, optimizing the layout, with the reduction of the FPGA circuit area established as a primary project objective. This approach, tailored for FPGA implementation rather than TTL, led to specific decisions contributing to circuit reduction in this particular context. Choosing multiplexers at the input of the adders, instead of at the output, was one such decision. Motivated by the presence of a 6-input lookup table on the FPGA, this choice makes the circuit more compact for this specific case.

The ALU, showcased in Figure 7, was designed around an 8-bit adder, receiving two 8-bit data and a ***Cin*** (carry in) bit. This setup results in a 9-bit output, where 8 bits represent the operation result, and 1 bit is the ***C*** (carry out).

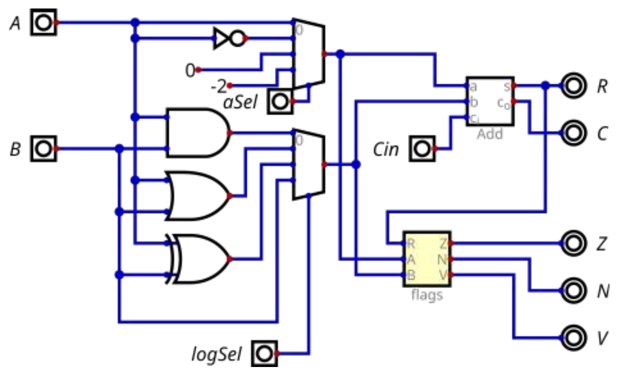


Figure 7. The ALU combines two operands in different ways by the instructions.

The operations performed by the adder are determined by two multiplexers. The first multiplexer, in short, defines the arithmetic operations. Using the ***aSel*** selector, it is possible to choose the operand ***A***, the inverse of ***A***, 0 or -2.

The second multiplexer is mainly responsible for logical operations. Using the logSel selector, it is possible to choose between ***A*** and ***B***, ***A*** or ***B***, ***A*** or ***B***, or just ***B***.

In this way, with this simple circuit, we can perform all the necessary operations in our ALU. To perform a logical operation, we insert bit 0 into the first multiplexer, which will be added to the logic defined by the second multiplexer: AND, OR, XOR. In this way, it is as if the adder were removed and the result of the logical operation was passed directly to the output.

Now, to perform arithmetic operations, we set the second multiplexer to send the operand ***B*** directly to the adder, while the first multiplexer can determine the operation. When we pass ***A*** directly, we perform an addition; when passing the inverse of ***A***, we perform a subtraction, but to do so we send 1 in ***Cin*** so that we have the two's complement.

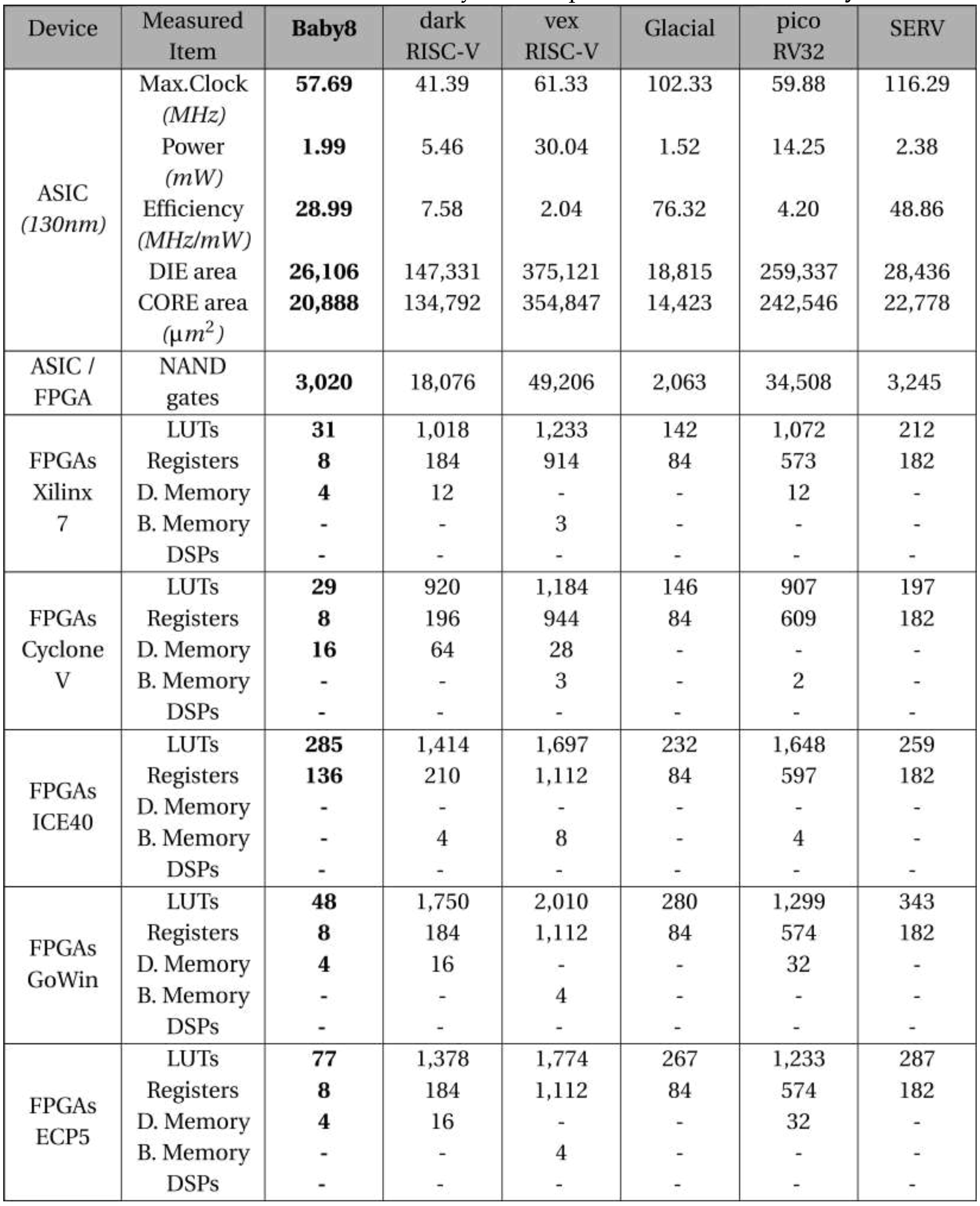
4. Results

All the results presented in the Tables 7 and 8 were evaluated using both scripts and data available by the authors in their Github repository. The results are divided into two tables divided into RISC-V™ and Non-RISC-V™ compatibility.

We show the main results classified as ASIC, NAND gates and FPGA. For ASIC we inform the maximum clock frequency, the power consumption, the efficiency and the effective area of the CORE and DIE (DIE is a small block of semiconducting material on which a given functional circuit is fabricated).

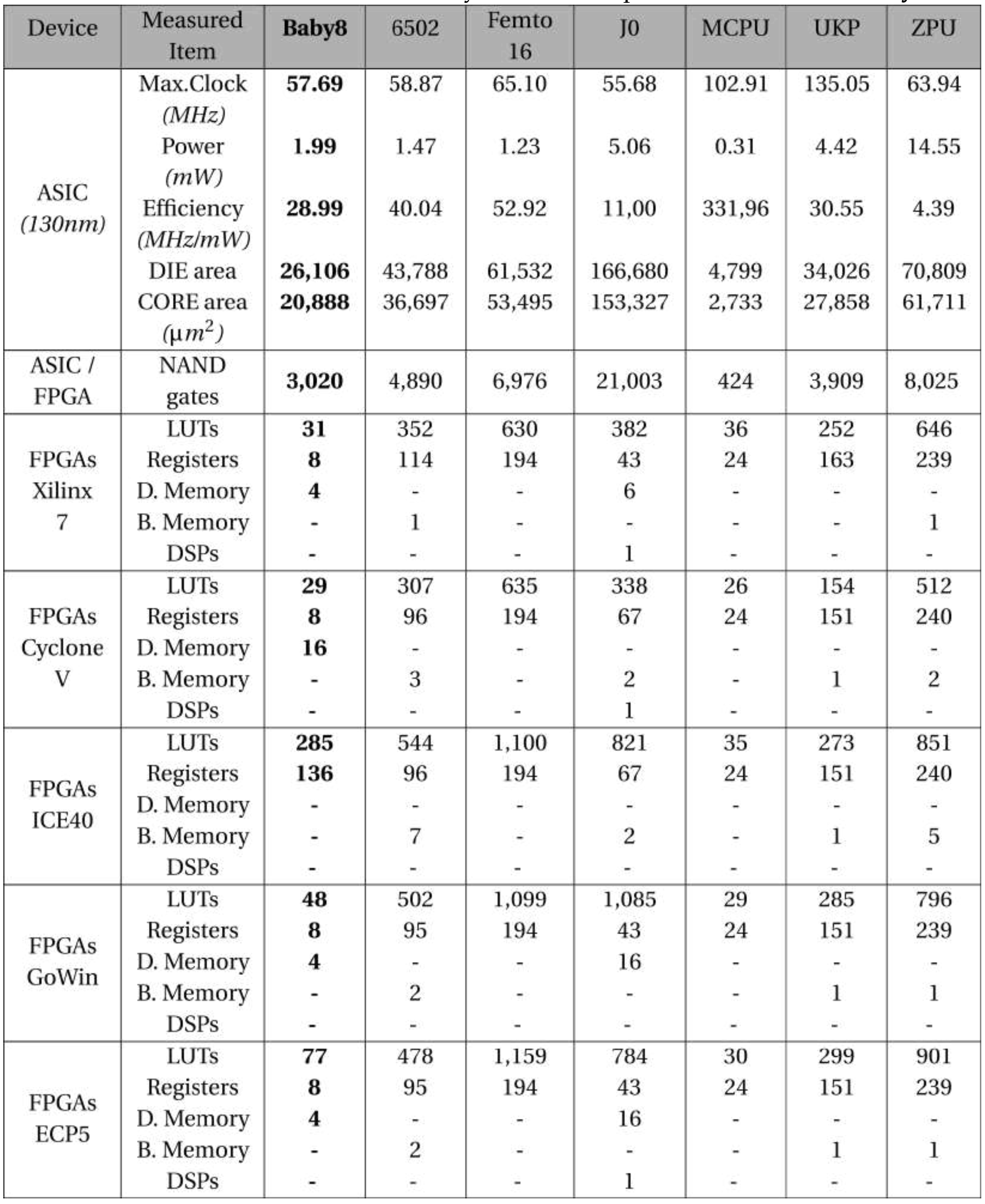
For NAND gates we inform the plain number of gates used. And for FPGAs we inform the number of LUTs, registers, distributed memories, block memories and DSP units used for each family of FPGA (Xilinx 7, Cyclone V, ICE 40, Gowin and ECP 5).

Table 7. ASIC and FPGA resources used by RISC-V processors related to **Baby8**.



Device	Measured Item	**Baby8**	dark RISC-V	vex RISC-V	Glacial	pico RV32	SERV
ASIC *(130nm)*	Max.Clock *(MHz)*	**57.69**	41.39	61.33	102.33	59.88	116.29
	Power *(mW)*	**1.99**	5.46	30.04	1.52	14.25	2.38
	Efficiency *(MHz/mW)*	**28.99**	7.58	2.04	76.32	4.20	48.86
	DIE area	**26,106**	147,331	375,121	18,815	259,337	28,436
	CORE area *(μm^2)*	**20,888**	134,792	354,847	14,423	242,546	22,778
ASIC / FPGA	NAND gates	**3,020**	18,076	49,206	2,063	34,508	3,245
FPGAs Xilinx 7	LUTs	**31**	1,018	1,233	142	1,072	212
	Registers	**8**	184	914	84	573	182
	D. Memory	**4**	12	-	-	12	-
	B. Memory	-	-	3	-	-	-
	DSPs	-	-	-	-	-	-
FPGAs Cyclone V	LUTs	**29**	920	1,184	146	907	197
	Registers	**8**	196	944	84	609	182
	D. Memory	**16**	64	28	-	-	-
	B. Memory	-	-	3	-	2	-
	DSPs	-	-	-	-	-	-
FPGAs ICE40	LUTs	**285**	1,414	1,697	232	1,648	259
	Registers	**136**	210	1,112	84	597	182
	D. Memory	-	-	-	-	-	-
	B. Memory	-	4	8	-	4	-
	DSPs	-	-	-	-	-	-
FPGAs GoWin	LUTs	**48**	1,750	2,010	280	1,299	343
	Registers	**8**	184	1,112	84	574	182
	D. Memory	**4**	16	-	-	32	-
	B. Memory	-	-	4	-	-	-
	DSPs	-	-	-	-	-	-
FPGAs ECP5	LUTs	**77**	1,378	1,774	267	1,233	287
	Registers	**8**	184	1,112	84	574	182
	D. Memory	**4**	16	-	-	32	-
	B. Memory	-	-	4	-	-	-
	DSPs	-	-	-	-	-	-

Table 8. ASIC and FPGA resources used by non-RISC-V processors related to **Baby8**.



Device	Measured Item	**Baby8**	6502	Femto 16	J0	MCPU	UKP	ZPU
ASIC *(130nm)*	Max.Clock *(MHz)*	**57.69**	58.87	65.10	55.68	102.91	135.05	63.94
	Power *(mW)*	**1.99**	1.47	1.23	5.06	0.31	4.42	14.55
	Efficiency *(MHz/mW)*	**28.99**	40.04	52.92	11,00	331,96	30.55	4.39
	DIE area	**26,106**	43,788	61,532	166,680	4,799	34,026	70,809
	CORE area (μm^2)	**20,888**	36,697	53,495	153,327	2,733	27,858	61,711
ASIC / FPGA	NAND gates	**3,020**	4,890	6,976	21,003	424	3,909	8,025
FPGAs Xilinx 7	LUTs	**31**	352	630	382	36	252	646
	Registers	**8**	114	194	43	24	163	239
	D. Memory	**4**	-	-	6	-	-	-
	B. Memory	-	1	-	-	-	-	1
	DSPs	-	-	-	1	-	-	-
FPGAs Cyclone V	LUTs	**29**	307	635	338	26	154	512
	Registers	**8**	96	194	67	24	151	240
	D. Memory	**16**	-	-	-	-	-	-
	B. Memory	-	3	-	2	-	1	2
	DSPs	-	-	-	1	-	-	-
FPGAs ICE40	LUTs	**285**	544	1,100	821	35	273	851
	Registers	**136**	96	194	67	24	151	240
	D. Memory	-	-	-	-	-	-	-
	B. Memory	-	7	-	2	-	1	5
	DSPs	-	-	-	-	-	-	-
FPGAs GoWin	LUTs	**48**	502	1,099	1,085	29	285	796
	Registers	**8**	95	194	43	24	151	239
	D. Memory	**4**	-	-	16	-	-	-
	B. Memory	-	2	-	-	-	1	1
	DSPs	-	-	-	-	-	-	-
FPGAs ECP5	LUTs	**77**	478	1,159	784	30	299	901
	Registers	**8**	95	194	43	24	151	239
	D. Memory	**4**	-	-	16	-	-	-
	B. Memory	-	2	-	-	-	1	1
	DSPs	-	-	-	1	-	-	-

The Figures 8 and 9 show the total utilization of the DIE area and the number of NAND gates used by each processor.

As one can see in Figure 10, the number of LUTs can vary greatly depending on the chosen FPGA family for the same processor. The best use of resources is often linked to the improvement of development tools and the integration of the synthesis results generated to their own internal hardware. So a project decision can't be based only on lower prices of the FPGA chips but also on their efficiency in the synthesis due to a greater use of resources based on its company technology.

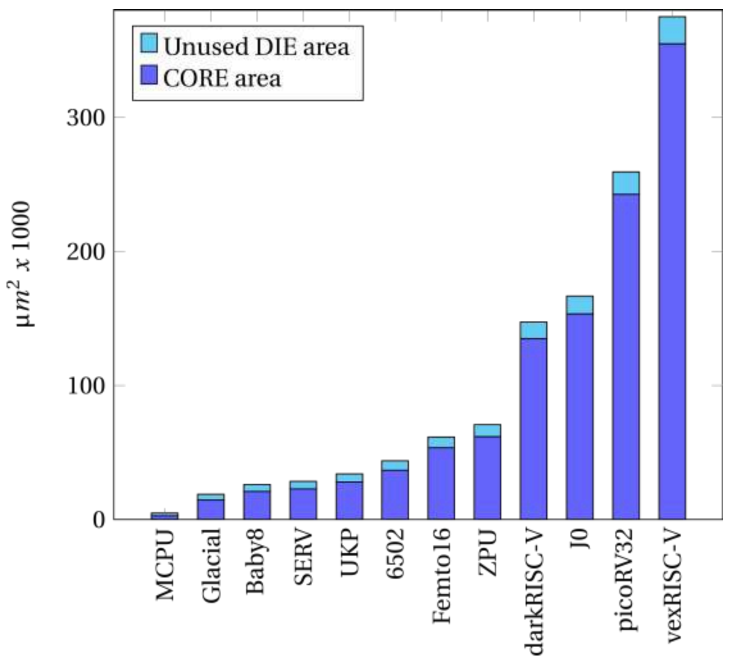


Figure 8. Utilization of the total DIE area by processor CORES..

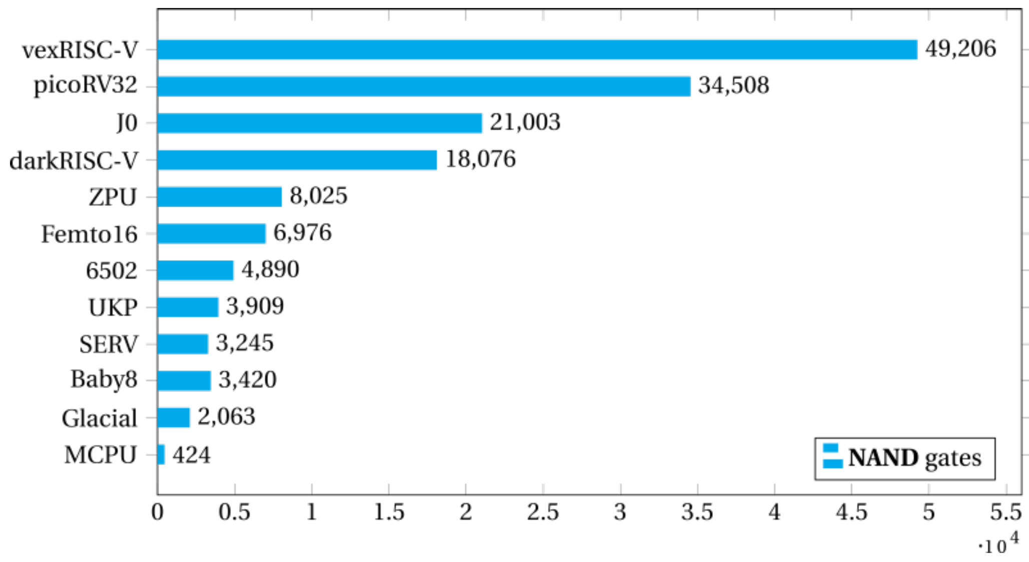


Figure 9. Soft-core processor resources utilization for ASIC / FPGA based on the number of NAND gates.

Figure 11 shows an extremely low use of FPGA resources for the I/O generic purpose processor **Baby8** (that was exactly our initial goal in this project) in comparison with Figures 12 (darkRISCV) and 13 (J0). Only the FPGA family ICE40 synthesized the Baby8 soft-core processor with a high use of resources.

One can notice that the same processor darkRiscV (Figure 10) could be synthesized using more registers or more LUTs, depending on the FPGA family chosen.

The J0 is the only processor in this study that used FPGA DSP units and for the FPGAs Xilinx 7, Cyclone V and ECP 5 only. This is the reason why FPGA ICE40 and GoWin are consuming much more LUTs and registers than other ones in J0 on the radar plot of Figure 9. And despite the fact that J0 processor consumes almost the same number of NAND gates of darkRISCV processor (Figure 8), it consumes much less FPGA's LUTs and registers due to use of these FPGA's DSP units.

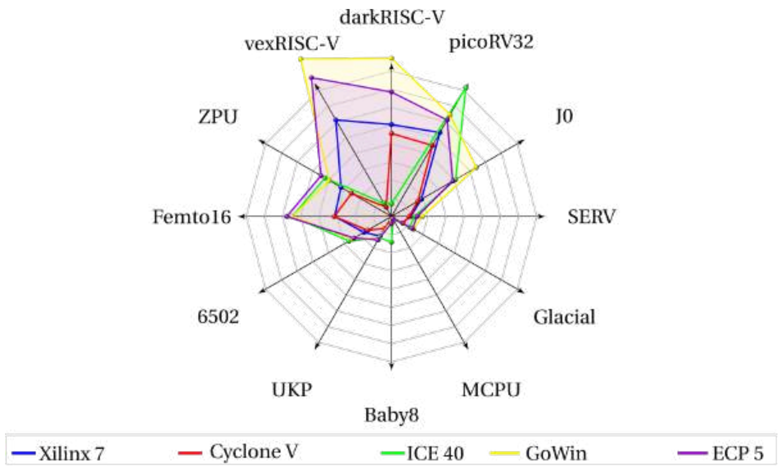


Figure 10. Proportion of LUTs allocated per soft-core processor and per FPGA model.

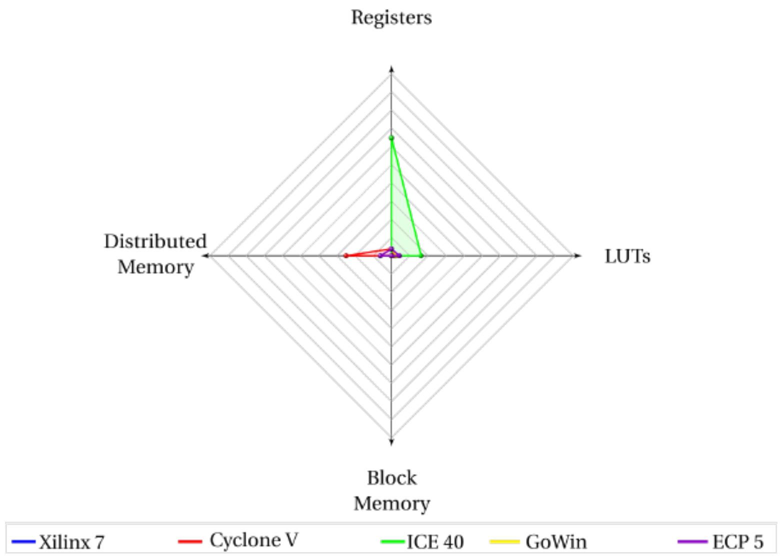


Figure 11. Proportion of LUTs, Registers, Distributed Memory, Block Memory and DSP allocated for the Baby8 processor per FPGA model.

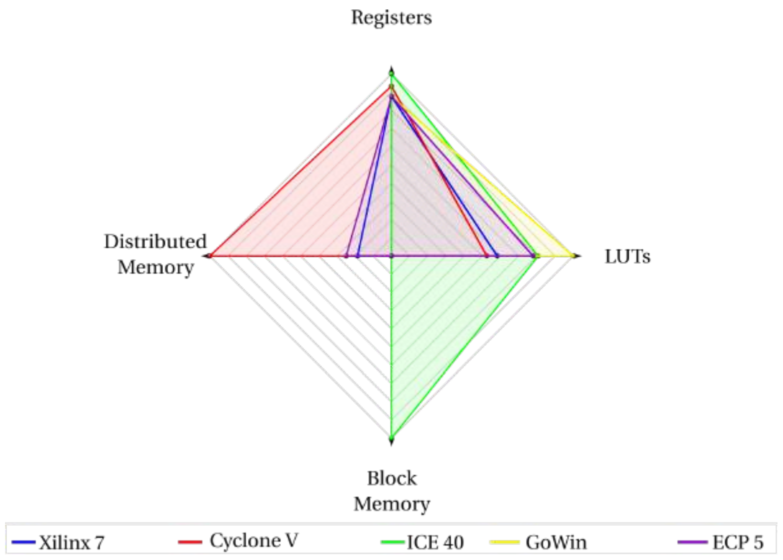


Figure 12. Proportion of LUTs, Registers, Distributed Memory, Block Memory and DSP allocated for darkRISC-V processor per FPGA model.

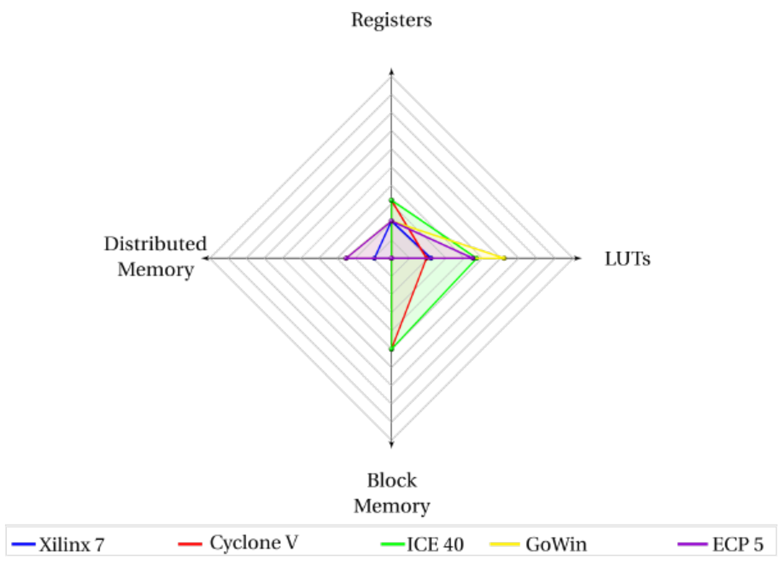


Figure 13. Proportion of LUTs, Registers, Distributed Memory, Block Memory and DSP allocated for J0 processor per FPGA model.

.

4.1. Performance

Although performance is not a measure of resource utilization per se, resource contention certainly impacts directly on the final performance of a processor design.

As much as the objective of this paper is to develop a processor with low resource usage (in terms of internal components for FPGAs or even area for ASICs), we must show in this section a comparative analysis of the final performance of the developed processor compared to the others analyzed.

By Figure 14 we can see that our processor **Baby8** successfully achieves an average performance both in maximum clock frequency as efficiency in power consumption among all other processors.

The frequency measurements were performed with OpenLane2 [11][12]. For each project, the OpenLane2 was configured to synthesize using the Sky130A PDK from Skywater and to use a 100ns clock period (10MHz) for the timing analysis.

This analysis is done several times during the chip generation process including after placement and routing. The timing analysis is done for the best case (low temperature and high supply voltage), the typical case and the worst case (high temperature and low supply voltage). In addition, the analysis is done for the minimum, nominal and maximum delays for the components.

The limiting factor of how high the clock for the device can be is indicated by the worst case setup time slack for the maximum delay with high temperature and low supply voltage. Subtracting this value from the clock period gives us a higher clock than the original 10MHz.

Synthesis is run again and the same circuit should result, but the timing analysis will give a different setup slack for the worst case. It might seem that the new number would be zero, but there are several complicating factors and the actual result will be smaller than the first time but still a positive value. The new number is subtracted from the new clock period and the process is repeated. This allows a successive approximation to the actual maximum clock possible of each processor.

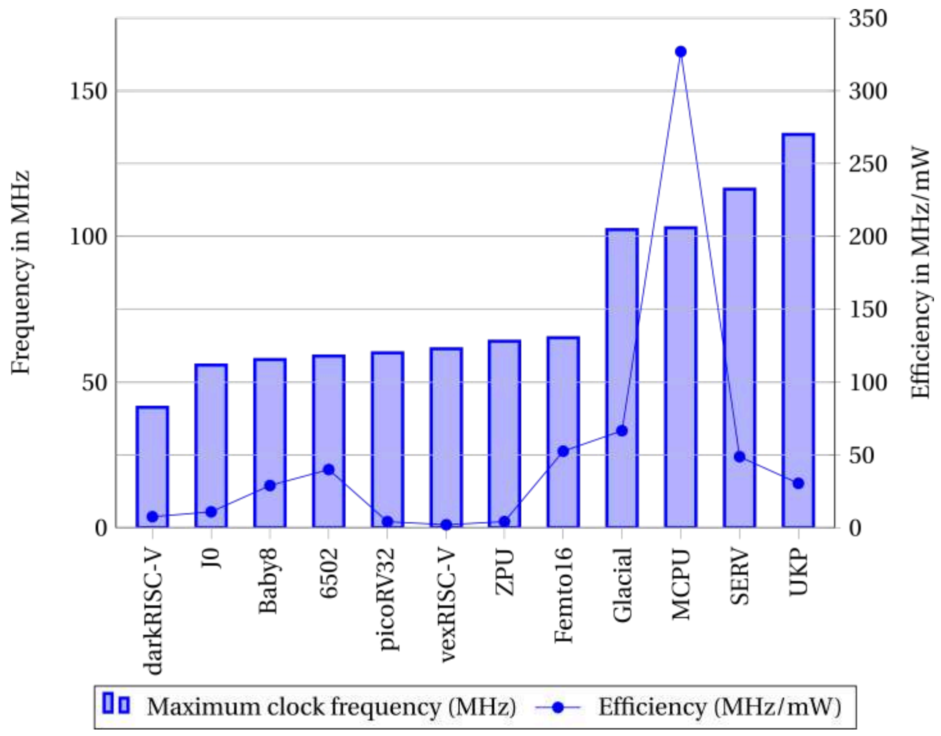
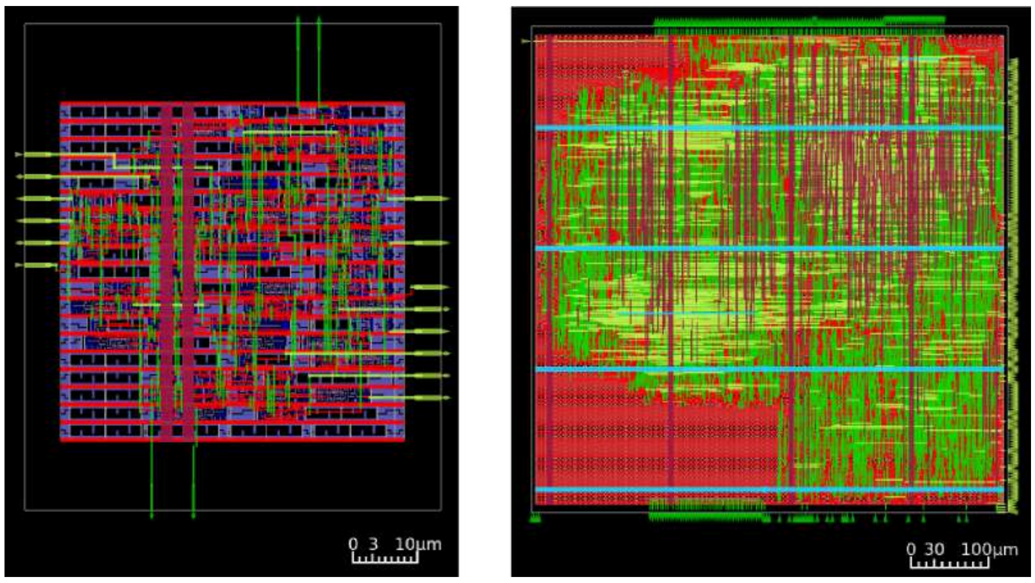


Figure 14. Maximum clock frequency (MHz) vs Efficiency (MHz/mW) in an ASIC simulation with *130nm* process and using worst case scenario: 1.6 volts at temperature of 100ºC.

4.2. Layouts

The chip layouts were also produced by the Open Lane 2 tool [26] and the layouts of the smallest area (MCPU) and largest area (vexRISC-V) processors implemented are shown in Figure 15.



(a) MCPU (b) vexRISC-V

Figure 15. The smallest area (MCPU) and the largest area (vexRISC-V) chip layouts.

5. Discussion

Although our developed **Baby8** soft-core processor achieves very good results in performance (~57 MHz) and energy consumption (~2mW), we need to make sure it can meet the initially proposed requirements of using few FPGA internal resources.

In order to do this analysis **Baby8** results were compared with 6502 results, both implemented in the ICE40 FPGA family.

Baby8 uses practically half of LUTs (285) than 6502 (544) in the same FPGA. It still uses a bit more registers - 136 - against 96 of 6502, but doesn't use any additional memory block, despite the fact that 6502 uses 7 blocks or memory RAM.

Since both processors are very similar in all other aspects (both are CISC 8-bits, with same power consumption and same maximum achieved frequency clock), we can consider that **Baby8** architecture could save up to half of FPGA resources compared with 6502 for the same features in the project.

6. Conclusions

Our developed soft-core processor architecture successfully achieves the goals of stay tiny small and saves nearly half of the resources in FPGA implementation, keeping the same performance and power consumption from similar processors

Author Contributions: Conceptualization, methodology, software: Jecel Mattos de Assumpção Junior; formal analysis, investigation, data curation: Mario Gazziro; review, supervision, project administration and funding acquisition: Oswaldo Hideo Ando Junior, Marco Roberto Cavallari and João Paulo Carmo.

Funding: This research was funded by the Brazilian Ministry of Science, Technology and Innovations by providing 5 undergraduate grants in PMN-Design microelectronics program 2B, managed by Eldorado Institute.

Data Availability Statement: All data and software are available online at author's Github repository..

Acknowledgments: Brazilian Ministry of Science, Technology and Innovations, Eldorado Institute, Chandler Klüser, Alejandra Victoria Herrera, Hugo Puertas, Beatriz Aimee Teixeira Furtado Braga, Gabriel Valizi, Gabriel Victor Santana, Samuel Rocha Silva, Vinicius Caffeu, Maximiliam Luppe.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
FPGA	Field Programmable Gate Array
I/O	Input / Output
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer

References

1. Glaser, J.;Wolf, C. Methodology and Example-Driven Interconnect Synthesis for Designing Heterogeneous Coarse-Grain Reconfigurable Architectures. In Proceedings of theModels,Methods, and Tools for Complex Chip Design; Haase, J., Ed., Cham, 2014; pp. 201–221.
2. Shah, D.; Hung, E.; Wolf, C.; Bazanski, S.; Gisselquist, D.; Milanovic, M. Yosys+nextpnr: An Open Source Framework from Verilog to Bitstream for Commercial FPGAs. In Proceedings of the 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom ComputingMachines (FCCM), Los Alamitos, CA, USA, may 2019; pp. 1–4. https://doi.org/10.1109/FCCM.2019.00010.
3. Assumpção, J. Baby8. https://github.com/jeceljr/baby8, 2023.
4. Patterson, D.A.; Hennessy, J.L. Computer organization and design ARM edition: the hardware software interface;Morgan kaufmann, 2016. .
5. Hiremath, S.; Chickerur, S.; Dandin, J.; Patil, M.;Muddinkoppa, B.; Adakoli, S. Open-source Hardware: Different Approaches to Softcore implementation. In Proceedings of the 2022 International Conference on Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER), 2022, pp. 76–83. https://doi.org/10.1109/DISCOVER55800.2022.9974836.
6. SiFive. SiFive. https://www.sifive.com/about, 2023.
7. Heinz, C.; Lavan, Y.; Hofmann, J.; Koch, A. A Catalog and In-Hardware Evaluation of Open-Source Drop-In Compatible RISC-V Softcore Processors. In Proceedings of the 2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig), 2019, pp. 1–8. https://doi.org/10.1109/ReConFig48160.2019.8994796.
8. Gruin, A.; Carle, T.; Cassé, H.; Rochange, C. Speculative Execution and Timing Predictability in an Open Source RISC-V Core. In Proceedings of the 2021 IEEE Real-Time Systems Symposium(RTSS), 2021, pp. 393–404. https://doi.org/10.1109/RTSS52674.2021.00043.
9. Coluccio, A.; Ieva, A.; Riente, F.; Roch,M.R.; Ottavi,M.; Vacca, M. RISC-Vlim, a RISC-V Framework for Logic-in-Memory Architectures. Electronics 2022, 11. https://doi.org/10.3390/electronics11192990.
10. Copeland, B. TheManchester Computer: A Revised History Part 2: The Baby Computer. IEEE Annals of the History of Computing 2011, 33, 22–37. https://doi.org/10.1109/MAHC.2010.2.
11. Ghazy, A.A.; Shalan,M. OpenLANE: The Open-Source Digital ASIC Implementation Flow. 2020.
12. Wang, H.; Li, T.; Li, Y.; Chen, L.; Sima, C.; Liu, Z.;Wang, B.; Jia, P.;Wang, Y.; Jiang, S.; et al. OpenLane-V2: A Topology Reasoning Benchmark for Unified 3D HDMapping. In Proceedings of the Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track, 2023.