IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received January 29, 2021, accepted February 19, 2021, date of publication March 2, 2021, date of current version March 24, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3063238

RISC-V3: A RISC-V Compatible CPU With a Data Path Based on Redundant Number Systems

MARC REICHENBACH, JOHANNES KNÖDTEL, SEBASTIAN RACHUJ, AND DIETMAR FEY
Department of Computer Science, Computer Architecture, Faculty of Engineering, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 91058 Erlangen, Germany

Corresponding author: Marc Reichenbach (marc.reichenbach@fau.de)

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) through the Joint Research in the Project IMBRA under Grant FE 412/8-1 and Grant RE 4182/2-1.

ABSTRACT Redundant number systems (RNS) are a well-known technique to speed up arithmetic circuits. However, in a complete CPU, arithmetic circuits using RNS were only included on subcircuit level e.g. inside the Arithmetic Logic Unit (ALU) for realization of the division. Still, extending this approach to create a CPU with a complete data path based on RNS can be beneficial for speeding up data processing, due to avoiding conversions in the ALU between RNS and binary number representations. Therefore, with this paper we present a new CPU architecture called RISC-V3 which is compatible to the RISC-V instruction set, but uses an RNS number representation internally to speed up instruction execution times and therefore increase the system performance. RISC-V is very suitable for RNS because it does not have a flags register which is expensive to calculate when using an RNS. To present reliable performance numbers, arithmetic circuits using RNS were realized in different semiconductor technologies. Moreover, an instruction set simulator was used to estimate system performance for a benchmark suite (Embench). Our results show, that we are up to 81% faster with the RISC-V3 architecture compared to a binary one, depending on the executed benchmark and CMOS technology.

INDEX TERMS Arithmetic and logic units, processor architectures, RISC, ternary arithmetic.

I. INTRODUCTION

For building fast and energy efficient computer systems, one possibility is to consider alternative number representations other than traditional 2's complement. An approach proposed in the past is to use redundant number systems [1]. These number systems allow the usage of more than r values per digit for a radix $r > 1$. While this looks very inefficient on the first glance in terms of hardware resources especially for storing data, arithmetic calculations can be performed asymptotically faster with regards to the word width when using these numbers. The reason for this improvement lies within the limited carry propagation of arithmetic circuits implementing such number systems.

Parhami [2] proposed GSD (General Signed-Digit) number systems as a framework for different redundant number representations and its arithmetic operations (e.g. additions) on it. As special cases for radix-2 based systems ($r = 2$) BSD (binary signed-digit), with the values $\{-1, 0, +1\}$, and BSC (binary stored-carry) with the values $\{0, +1, +2\}$ are covered by this framework. By utilizing these number representations, additions can be executed in constant time, independent of the word length. This fact can be exploited to speed up arithmetic in modern CPUs. Moreover, addition can be seen as the basis of most arithmetic operations. Optimizing this operation is potentially able to increase a CPUs performance dramatically. For example some floating-point units are utilizing RNS in their internal architecture to perform the SRT division algorithm [3], [4]. Finally, limiting carry propagation and increasing data locality decrease CMOS net activity for sufficiently large word sizes and is therefore able to save energy.

In this paper, especially BSD and BSC number representations are used, because these systems allow the implementation with traditional full adder structures. Using full adder cells promises a short critical path because highly optimized versions are often provided in most standard cell libraries. Fig. 1 shows an example for the addition of two numbers in BSC representation. As discussed, a number in this number representation may not only contain "1"s and "0"s as

The associate editor coordinating the review of this manuscript and approving it for publication was Sun Junwei.

$$\begin{array}{cccccccc} & 0 & 0 & 1 & 1 & 1 & = & 7 \\ + & 1 & 1 & 1 & 1 & 0 & = & 30 \\ \hline & 1 & 2 & 1 & 0 & 1 & = & 37 \end{array}$$

FIGURE 1. Addition example in BSC. The reason that this example can be calculated in constant time is that the carries can only affect at most two digits in the result. Note that is possible to continue calculating with the resulting number without eliminating "2"s from the result and still retaining the constant time delay, as shown in section II.

digits; additionally, we allow the "2" as digit value here. Consequently, instead of propagating a carry, the carry will be included in the appropriate digit, which will be possible by allowing a "2" as digit for the number representation. Due to the possibility to allow three different states per digit, we will call these number representations in this paper *ternary* number systems.

In the area of redundant number systems a lot of research has been done. Parhami [5] first proposed a recode mechanism for BSD numbers, to allow a carry free-addition. For BSC, Weinberger [6] proposed the 4-2-Adder Module, which was re-analyzed several times in context of GSD [7], [8]. This circuit is usually known as a carry-save adder. Based on these fundamental findings more recent work has been done: For example in [9] and [10] new FPGA implementations of arithmetic circuits using RNS have been developed. Moreover, various applications could be sped up using RNS. Examples inclue complex DSP calculations using CORDIC [11] or the fast computation of cryptographic functions [12].

As shown, redundant number representations might be able to increase arithmetic subcircuit performance significantly and are therefore used in rare cases, e.g. for the SRT division in the arithmetic logic unit (ALU) of a CPU. However, due to the non-unique representation of a number in RNS meaning that a value z can be represented by different bit vectors in RNS, a time-consuming (non-constant time) conversion becomes necessary for further arithmetic operations using this value in classical 2's complement. For example, in a CPU that uses classical 2's complement representation in general but also a RNS-based arithmetic subcircuits for division, a conversion is needed for each division instruction inside the ALU. As Parhami already stated, "Whenever long sequences of computations are to be performed on particular pieces of data, the one-time conversion and reconversion effort to/from the OSD [ordinary signed digit] representation is more than compensated for by the gain in computation speed." [2]. Hence, it is useful to increase the length of these sequences. Therefore, for a further performance increase, this implies not only using RNS based arithmetic subcircuits, but rather to completely rely on ternary encodings on the data path in the CPU. Consequently, conversions between/to registers and the ALU in a CPU will be avoided. Finally, all instructions in a CPU are working on words in RNS representation which avoids the conversation completely (except of Input/Output of the CPU). In Fig. 2 a comparison between a traditional CPU and a ternary CPU as presented in this paper is shown. While in the traditional CPU only a small subcircuit (SRT) inside the ALU works with RNS (marked in red), in the ternary CPU the complete data path is implemented based on RNS. There are still circuits relying on binary representations, as they are unavoidable, e.g. bit-wise operations such as bit-wise and, but signals between functional units are encoded entirely in RNS.

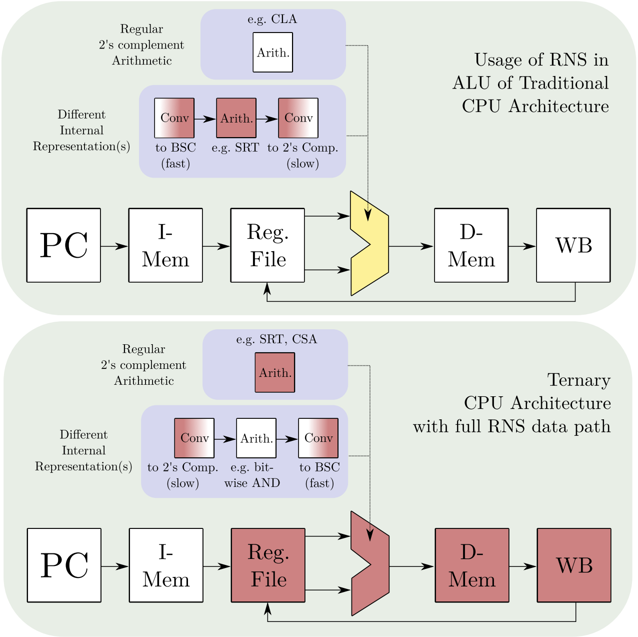


FIGURE 2. RNS CPU compared to traditional ones. The upper part shows a conventional CPU data path where only a few functional units (like SRT) use a RNS (marked in red). In the lower part a CPU is depicted that uses uses RNS as the default number representation (also marked in red) and only converts back for certain operations (like logical operations).

Although, the above mentioned method promises a huge performance gain, different challenges arise:

1) Due to more bits having to be stored for RNS numbers than actually required for conventional binary ones, register files inside the CPU increase in size as well as energy consumption.
2) While arithmetic circuits such as addition and multiplication will become faster and more efficient using RNS, other operations such as shifts, bit-wise logical operations, and sign detection are more complex than their counterparts using 2's complement.
3) By executing several consecutive arithmetic operations using RNS, results might need more digits to be stored. While this might look as an integer overflow, the corresponding value is still in the valid range. This topic was covered in previous works by different authors and the problem was coined *pseudo-overflow* by Timmermann and Hosticka [13].

To really benefit from the ternary CPU, the above shown challenges (1) to (3) need to be addressed in detail and will be analyzed in this paper. Therefore, we want to investigate the additional overhead which is needed to solve these problems. Fortunately, research activities to address problem (1) have been done before. To overcome it, the authors of [14] propose to use novel emerging memories technologies such as ReRAM, which are able to store multiple states in one memory cell and therefore drastically improve the memory density.

However, problem (2) is a crucial limitation for processor designers and one of the main reasons, why this number representation is not used more often. While addition and other operations based on addition can be executed very fast in redundant number representations as seen before, other operations such as bit-wise logic operations, overflow detection as well as sign detection cannot be executed in constant time, i.e. independent of the word length. In most cases a so-called renormalization becomes necessary which can be simply described as the conversion back into a conventional number respresentation. However, this needs the same amount of time as an addition in traditional 2's compliment representation. Nevertheless, even if a few operations cannot be sped up, a performance improvement of programs running on the system as a whole is still possible. We will prove this claim in this paper by providing a system simulator which evaluates the overall performance of the ternary CPU compared with a binary one.

In ternary arithmetic subcircuits with binary external interfaces, problem (3) does not occur, because the number is converted back to the traditional 2's complement representation after the calculation which is equivalent to a complete renormalization. Unfortunately, in a CPU with a ternary data path an arbitrary number of arithmetic operations can be done on a register without any renormalization process. Due to their redundant behavior, this will result in a large number of leading digits [15] which does not fit in the digits provided by the registers although the corresponding binary value would fit. In this paper, we will propose an adapted normalization process to solve this problem. It can be executed in constant time inside the adder circuit, independent from the word length.

As one can see, a deep analysis of ternary CPU architectures is needed. On the one hand side, RNS promises a huge performance gain due to optimized arithmetic, on the other hand side, new drawbacks arise which did not exist in traditional computers using 2's complement computers. Therefore, in this paper we will provide an extended analysis of RNS data paths and their impact on complete CPUs. This will include a methodology on how to address the above mentioned challenges. Finally, we create a new CPU architecture, based on RISC-V, with a ternary data path, utilizing BSC and BSD RNS as well as an evaluation methodology to evaluate chances and risks of using RNS in CPUs.

This paper contains 6 Sections, organized in an order, that follows a bottom-up approach. In this Section, we provided an introduction and gave a historical background. In Section II, we first design and present our basic arithmetic circuits using RNS (including conversion functions to 2's complement) and show their performance. This enables a quantitative analysis of these RNS in terms of timing. Afterwards, in Section III, we present our RISC-V system architecture using a ternary data path and map the determined timings, found in the previous section, to this architecture. Using this approach, we are able to run simulations to estimate timing behavior of the whole processor. This will be focused on in Section IV. There, we present our complete simulation environment and how it is possible to run real (benchmark) programs on the new ternary architecture. Finally, in Section V, we do an extended evaluation and compare the measured results using our new processor architecture with traditional ones using 2's complement. Section VI will summarize and conclude this paper.

Summarizing, our contributions in this paper are:

- A study on ternary circuit implementations containing arithmetic circuits for different RNS and their mapping to different CMOS technologies. This resulted in a characterized VHDL IP library.
- A concept of a CPU architecture with ternary data path based on the RISC-V ISA.
- A simulation environment to execute benchmark programs on the ternary CPU architecture.
- An evaluation methodology, i.e. providing a set of experiments to estimate the overall system performance. This methodology is able to translate knowledge about arithmetic performance to system level performance metrics.

II. CIRCUIT DESIGN FOR RNS ARITHMETIC

As described above, in this chapter we will design our arithmetic circuits in different redundant number systems. Due to the fact that conventional CMOS technology does not offer the means for processing ternary values, the functionality has to be realized using binary signals. Thus, we first present different encoding schemes to realize numbers in RNS with conventional binary technologies. After that, we derive the arithmetic circuits for addition on these number representations and finally we present our synthesis results, which shows the benefits regarding timing behavior over the binary representation.

A. NUMBER REPRESENTATIONS AND ENCODINGS

The BSD and BSC number representations are both based on the radix $r = 2$. Due to the redundant nature of these number representations, each digit d_i in a BSC or BSD number z, has to hold one out of three possible states, which needs to be encoded with binary signals (bits). For the encoding of three possible states, at least two bits are required. With an encoding table we will define, how a set of two bits is interpreted in their respective number representation. This encoding heavily influences the arithmetic circuits as well as the synthesis results.

In BSD representation each digit d_i is defined as $d_i \in \{-1, 0, +1, \}$, resulting in a BSD number

$$z_{BSD} = (d_{n-1}, d_{n-2}, \ldots, d_0). \tag{1}$$

Now, each d_i needs to be encoded by two bits

$$d_i = \left(d_i^{(1)}, d_i^{(2)}\right), \quad d_i^{(1)}, d_i^{(2)} \in \{0, 1\} \tag{2}$$

TABLE 1. Encoding table for BSC and BSD number representation.

Bits		Encoded Values			
$d_i^{(1)}$	$d_i^{(2)}$	BSD-PN	BSD-SV	BSD-SUM	BSC-2C
0	0	0	0	−1	0
0	1	−1	1	0	1
1	0	1	0	0	1
1	1	0	−1	1	2

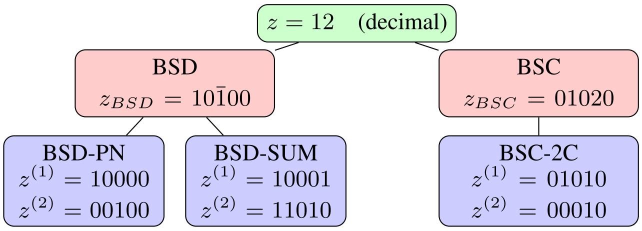


FIGURE 3. Example for different encodings with different in RNS.

which leads us to three encodings: (i) the so called positive/negative (BSD-PN) encoding, (ii) sign/value (BSD-SV) encoding and (iii) encoding we called the BSD-SUM encoding, as described by [2], [16]. The name BSD-SUM is used, because its encoded value can be derived directly from the sum of both individual bits (i.e. $d^{(1)}+d^{(2)}$), which means that $d = (0, 1)$ and $d = (1, 0)$ encoding the same encoded value (namely 0). For our experiments we will only use BSD-PN and BSD-SUM encoding, as it has the advantage, that the inversion of the number is simply done by inversion the bits of the digit. BSD-SV is not further investigated in this paper.

For BSC representation a number z is defined similarly with

$$z_{BSC} = (d_{n-1}, d_{n-2}, \ldots, d_0)\,. \tag{3}$$

Further, each digit d_i is defined as $d_i \in \{0, +1, +2\}$. As encoding we choose here the sum of the two representing bits, leading to $d_i = d_i^{(1)} + d_i^{(2)}$. We will call this encoding BSC-2C in the following. All of the discussed encodings are formally described in Table 1. In Fig. 3 an example is shown, how a number z can be described in different RNS with different encodings. Furthermore, we define $z^{(1)}$ and $z^{(2)}$ as bit vectors in the following way:

$$\begin{aligned} z^{(1)} &= \left(d_n^{(1)}, d_{n-1}^{(1)}, \ldots, d_0^{(1)}\right) \\ z^{(2)} &= \left(d_n^{(2)}, d_{n-1}^{(2)}, \ldots, d_0^{(2)}\right). \end{aligned} \tag{4}$$

While for BSD numbers, it is easy to represent negative numbers due to the presence of negative digits, for numbers in BSC representation an additional explanation how to represent negative numbers is needed. According to the fact, that in BSC-2C encoding, a digit is encoded as the sum of the two corresponding bits, also the overall number z_{BSC} can be interpreted as the sum of the two vectors $z^{(1)}$ and $z^{(2)}$ resulting in $z_{BSC} = z^{(1)} + z^{(2)}$. By interpreting the numbers $z^{(1)}$ and $z^{(2)}$ in two's complement, it becomes possible to represent negative numbers.

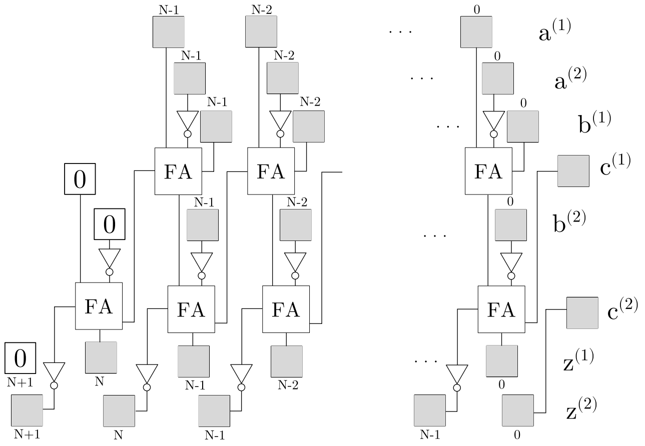


FIGURE 4. Adder circuit for BSD-PN encoding.

B. BASIC ARITHMETIC CIRCUITS (ADDERS)

Based on the three different encodings (BSD-PN, BSD-SUM und BSC-2C) adders can now be defined on circuit level. In Figure 4, Figure 5 and Figure 6 the resulting adder circuits are shown for performing a calculation of $z = a + b$. As basic element a classical full adder (FA) is used with some inverters. In contrast to a traditional adder (like a ripple carry adder) carry chains are limited to only one connection between FAs of one result digit in this implementation, resulting in the aforementioned constant critical path independent of the word length. Nevertheless, two additional bits (carry-in) $c^{(1)}$ and $c^{(2)}$ are also defined, which can be used for additional arithmetic operations (e.g. subtractions). Furthermore, it can be seen that the circuit for BSD-SUM and BSC-2C encoding looks very similar except of the most significant digit.

While the presented arithmetic circuits allow performing an addition, the subtraction can also be implemented. Based on the knowledge of traditional adders, a subtraction is performed by adding the negated number, i.e. $z = a-b = a+(-b)$. Defining the negation of a number z as $\text{neg}(z) = -z$, the calculation of $\text{neg}(z_{\text{BSD}})$ can, as mentioned previously for BSD-PN and BSD-SUM, easily be performed by evaluating

$$\overline{z^{(1)}} = \left(\overline{d_n^{(1)}}, \overline{d_{n-1}^{(1)}}, \ldots, \overline{d_0^{(1)}}\right) \tag{5}$$

and $\overline{z^{(2)}}$ respectively which can be performed by using simple XOR gates. The negation of z_{BSC} will be slightly more complex. As defined $z_{\text{BSC}} = z^{(1)} + z^{(2)}$ results in

$$\begin{aligned} \text{neg}(z_{\text{BSC}}) &= \text{neg}(z^{(1)}) + \text{neg}(z^{(2)}) \\ &= (\overline{z^{(1)}} + 1) + (\overline{z^{(2)}} + 1) \end{aligned} \tag{6}$$

by applying two's complement for the negation of $z^{(1)}$ and $z^{(2)}$. The constant values (1) can be used as input c in the represented adder circuit.

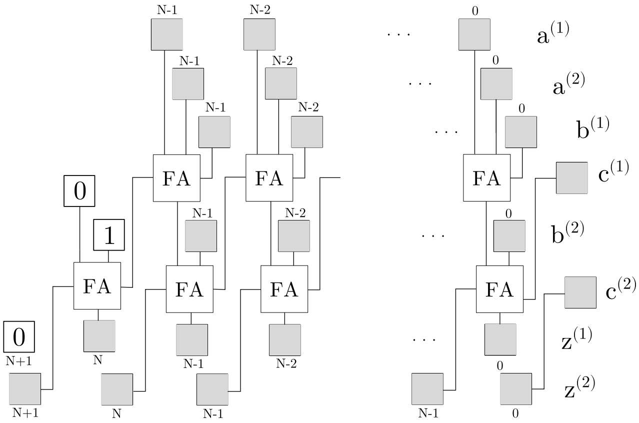


FIGURE 5. Adder circuit for BSD-SUM encoding.

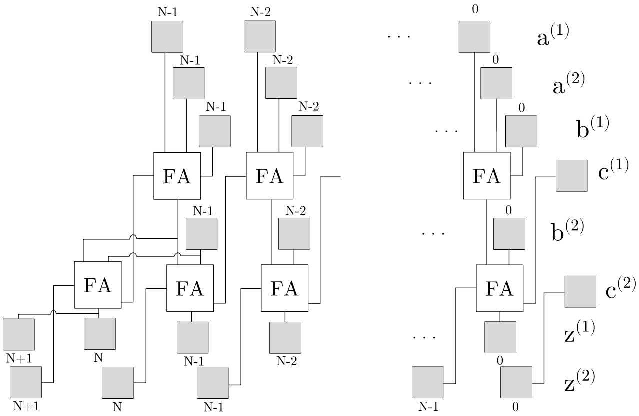


FIGURE 6. Adder circuit for BSC-2C encoding.

An even more efficient variant exists for BSD-PN, by swapping the constituent bits of each digit:

$$\begin{aligned}\text{neg}(z_{\text{BSD}-\text{PN}}) &= \text{neg}(z^{(1)}_{\text{BSD}-\text{PN}}, z^{(2)}_{\text{BSD}-\text{PN}})\\ &= (z^{(2)}_{\text{BSD}-\text{PN}}, z^{(1)}_{\text{BSD}-\text{PN}})\end{aligned}\tag{7}$$

C. DESIGN AND CHARACTERIZATION OF ADDERS

In order to get quantitative results we characterized the delay of the presented circuits in standard cell based designs by post-synthesis static timing analysis (STA). The methodology of this characterization is as follows: First the designs were synthesized and afterwards analyzed by static timing using an industry-standard synthesis tool. This is done by applying different constraints to force the tool to yield delay-optimal results. These timing constraints for finding the lowest critical path were found by binary search: this search is stopped after a number of steps where the maximum error of the search was smaller than 3 picoseconds, which is well within the error margin of the STA and synthesis. We found that due to the nature of the optimization algorithms in the synthesis tool, we do not get a constant delay for different word length. Nevertheless, the uncertainties are minor and therefore can be ignored.

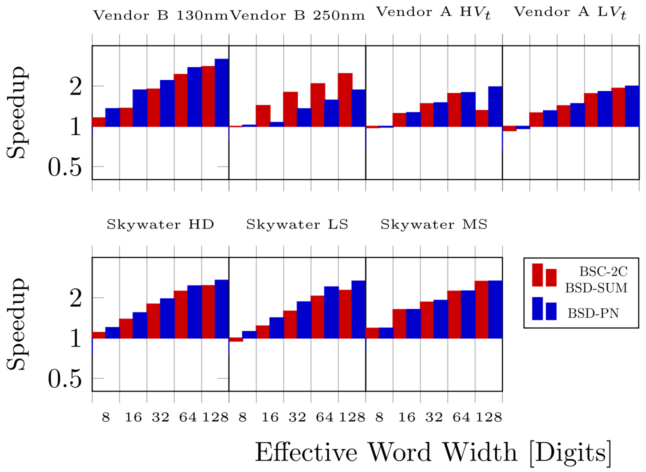


FIGURE 7. Speedup of RNS adders compared to a binary adder.

Our experiments are carried out utilizing different PDKs:

- Vendor A at 150nm (Low V_t (High Speed), High V_t (Low Power))
- Vendor B at 250nm, 130nm
- Skywater 130nm (Medium Speed, Low Speed, High Density)

Due to contractual obligations we cannot disclose the identity of Vendor A and Vendor B, but we are using regular standard cell libraries provided to us.

The results of this delay characterization can be found in Figure 7. It shows the speedup relative to the fastest binary adder circuit for different word widths for each technology (design kit). Due to the fact, that the adders for BSC-2C and BSD-SUM result in a similar circuit with the same critical path these two graphs are merged. It can be seen, that speedups up to 2.5x are possible for 128 bit word width in Skywater PDK.

In contrast to intuition, counting gates in the critical path is not representative for the delay of the associated timing arcs, as this is highly dependent on standard cell properties and capacities of interconnects between cells and input ports. These synthesis results suggest that e.g. the BSD-PN adder is faster than the BSD-SUM/BSC-2C adder although it has additional inverters.

D. CONSECUTIVE ADDITIONS AND NORMALIZATION

While in the previous chapters a simple addition was realized, the question arises whether such circuits can now be placed easily in the ALU of a CPU. Therefore, in this chapter we want to elaborate on consecutive additions and their impact on the corresponding adder circuits.

In Figure 8 an example is shown for two consecutive additions in BSC representation for $n = 5$ digits. Initially a and b are given by $a = 1$ and $b = -1$ and shown in their corresponding representation with $a^{(1)}$ and $a^{(2)}$, and $b^{(1)}$ and $b^{(2)}$ respectively. After calculating $c = a + b$ with the aforementioned adder circuits, the corresponding vectors

$$\begin{array}{rcrr}
 & & \text{Bits} & \text{Value} \\
a^{(1)} & = & 00001 = & 1 \\
a^{(2)} & = & 00000 = & 0 \\
b^{(1)} & = & 11111 = & -1 \\
b^{(2)} & = & 00000 = & 0 \\
c^{(1)} & = & \underline{1}\underline{1}11100 = & -4 \\
c^{(2)} & = & \underline{0}\underline{0}00100 = & 4 \\
d^{(1)} & = & \underline{1}\underline{1}10001 = & -15 \\
d^{(2)} & = & \underline{0}\underline{0}10000 = & 16 \ \text{or} -16?
\end{array}$$

FIGURE 8. Example of consecutive additions in a BSC-2C encoding with the initial values $a = 1$ and $b = -1$. In the first step $c = a + b$, in the second $d = c + a = 1 + (-1) + 1 = 1$ is calculated. The results contains two leading digits (underlined), which cannot be omitted.

$$\begin{array}{rcrr}
 & & \text{Bits} & \text{Value} \\
a^{(1)} & = & 10110101 = & -75 \\
a^{(2)} & = & 11010111 = & -41 \\
b^{(1)} & = & 00100101 = & 37 \\
b^{(2)} & = & 11010011 = & -45 \\
c^{(1)} & = & \underline{0}\underline{0}11111110 = & 254 \\
c^{(2)} & = & \underline{1}\underline{0}10000110 = & -378 \\
d^{(1)} & = & 11111111 = & -1 \\
d^{(2)} & = & 11111110 = & -2 \\
e^{(1)} & = & \underline{\underline{0}}\underline{\underline{0}}\underline{1}\underline{1}10000101 = & 901 \\
e^{(2)} & = & \underline{\underline{1}}\underline{\underline{0}}\underline{1}\underline{1}11111100 = & -1028 \\
e_n^{(1)} & = & \underline{\underline{1}}\underline{\underline{1}}\underline{1}\underline{0}10000101 = & -379 \\
e_n^{(2)} & = & \underline{\underline{0}}\underline{\underline{0}}\underline{0}\underline{0}11111100 = & 252
\end{array}$$

FIGURE 9. Example of consectuively additions in BSC representation with handling of leading digits. Initially the constants are set to $a = -116$ and $b = -8$. Afterwards $c = a + b = -124$ is calculated by needing two leading digits (underlined). Finally, we set $d = -3$ and calculate $e = c + d = -127$. It can be seen, that e requires again two additional leading digits (double underlined). Fortunately, the four additional leading digits (two underlined and two double underlined) can be condensed to only two leading digits, by performing a so-called constant-time partial normalization e_n = norm$_s$(e) and manipulating only the four leading bit.

$c^{(1)}$ and $c^{(2)}$ are also shown, which are yielding the correct result. In order to stress the effects to be obvious here, $c^{(1)}$ and $c^{(2)}$ contain two leading digits which are not necessary for the result to be correct. Finally, $d = c + a$ is calculated. Also here, $d^{(1)}$ and $d^{(2)}$ has two leading digits. In contrast to the calculation of c it can be seen, that these two digits cannot be omitted without altering the resulting value of the number.

This example leads to a problem which was already described before in [15] as "carries may ripple out into positions which may not be needed to represent the final value of the result and, thus, a certain amount of leading guard digits are needed to correctly determine the result". This means, that although if n digits are enough to encode a value, the result of an addition is a number z with the same value, but which requires $n + 2$ digits due to the structure of the adder circuit and the redundant nature of the number system.

For building a CPU with ternary data path this fact will cause a problem which has to be analyzed very carefully: Will the effect of leading digits accumulate over consecutive additions? This question is analyzed by a second example, shown in Figure 9. The two leading digits (underlined) for c cannot be removed easily: to remove them, a total renormalization of the whole number is needed. Total renormalization means, to convert the result completely to its non-redundant binary form and therefore allow storing the information in a binary compatible form in one of the vectors. Unfortunately, this is of course expensive and should be avoided. The delay of the total renormalization circuits is dominated by the adder or similar component used for the conversion to binary. To put these costs into perspective: The ternary adder is up to 4 times faster than the renormalization circuit.

Summarizing, it is clear, that it is not useful to perform a total renormalization after each addition. Therefore, it is a good idea to accept the additional generated leading digits for now, but investigate if they will increase after an additional addition step. Fortunately this is not the case as they will not accumulate over consecutive additions, after performing a so-called partial normalization after each addition step, taking only a constant amount of time. Referring again to Figure 9 the leading digits (underlined, as shown in the number e) can be condensed by a constant-time normalization (e_n = norm$_s$(e)) which only needs to consider a constant number of leading digits, independently of the word width.

For a formal description, two functions are introduced: head($z^{(i)}$) provides the four leading digits of a given vector $z^{(i)}$ while tail($z^{(i)}$) provides the remaining ones. Furthermore, if $+\!\!+$ is defined as the concatenation of two bit-vectors, then we can define z_n = norm$_s$(z) as follows and apply it to determine e_n as shown in the example.

$$\begin{aligned} z_n &= (z_n^{(1)}, z_n^{(2)}) = \mathrm{norm}_s(z) \\ &= (\mathrm{head}(z^{(1)}) + \mathrm{head}(z^{(2)})) +\!\!+ \mathrm{tail}(z^{(1)}), \\ &\quad 0000 +\!\!+ \mathrm{tail}(z^{(2)})) \end{aligned} \tag{8}$$

In [15] the leading-digits problem and its normalization is addressed in a mathematical way; also [13] shows a specific algorithm for SD numbers. Unfortunately, to our best knowledge, an RTL or lower level implementation is missing but needed for the investigation of these adders in a ternary CPU. Therefore in addition to the basic adder circuits, we provide normalization circuits for dealing with leading digits in this paper. Nevertheless, in the work of [15] and [13] multiple mathematical descriptions for constant-time normalization are shown, so the following three different circuits were implemented:

1) Adder-Normalization: as defined by norm$_s$(z); A generic method by us, which requires two leading digits. Can be used for BSD-PN, BSD-SUM and BSC-2C representation. This is loosely based on the proofs of [15].

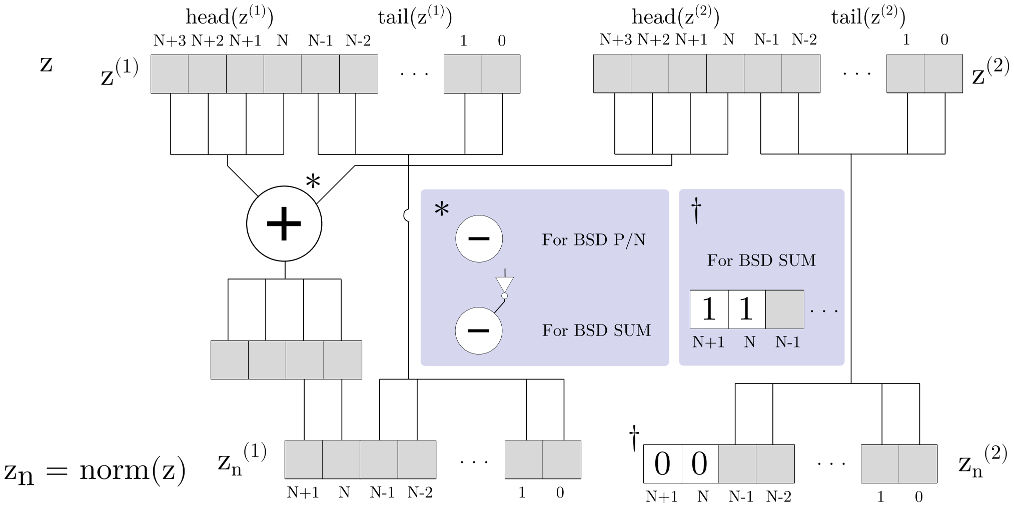


FIGURE 10. Adder-based normalization scheme.

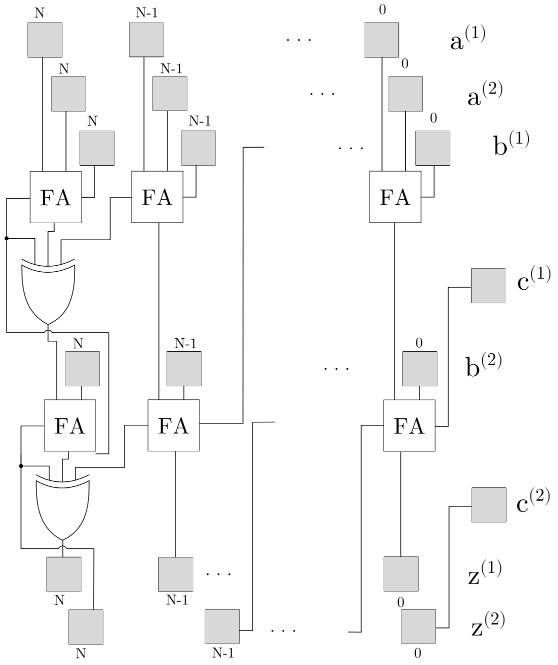


FIGURE 11. XOR-normalized adder for BSC-2C.

2) XOR-Normalization: Method presented in [15] (Corollary 9, similar to [17], [18]) which requires only one leading digit. Can only be used for BSC-2C.
3) MUX-Normalization: Method presented in [13] which requires also one leading digit. Can only be used for BSD-PV and BSD-SUM.

The described normalization circuits are shown in Figures 10, 11 and 12. Finally the MUX-Normalizer and the Adder-Normalizer need to be placed directly behind the basic adder circuit to reduce the length of $n+3$ (three leading digits) to $n+1$ (one leading digits) after each addition. The XOR-Normalizer is placed after each layer of FAs inside of the adder circuit and is therefore integrated into it.

TABLE 2. Overview about all investigated circuits. Left different RNS and its encodings, at the top different normalization circuits. A checkmark means that this combination will be considered.

	Adder-Norm	XOR-Norm	MUX-Norm
BSD-PV	✓		✓
BSD-SUM	✓		✓
BSC-2C	✓	✓	

Our findings result in Table 2. Here, all investigated circuits (combination of an adder and a normalization method) are shown. Summarizing, a constant time normalization should be performed after each addition step to make further calculations possible. However, full renormalization takes very long but is only needed in some special cases, especially when the binary representation of a number is exactly needed. That will be the case for example, if the result of an addition will be used by bit manipulating operators.

Finally, as presented before, the resulting circuits (adder with constant time normalization) have been evaluated in the same way, as the basic adder circuits. In Fig. 13 the results are shown. It is clear, that the speedup decreases by using these normalization circuits. However, such circuits are needed and definitively have to be considered when constructing a ternary CPU. Nevertheless, even a speedup of around two is still achievable for 128 bits registers. It can be seen, that BSD-SUM and BSD-PN with MUX-Normalization will give the best results, especially for Vendor B 130nm process. Therefore, we will focus on these numbers for building the ternary CPU.

When analyzing the other technologies, the results are worse compared to Vendor B 130nm technology. The main reason for these discrepancies are differences in standard cell libraries optimized for speed or other metrics: Here especially the delays incurred from gates and their output nets are different. This is hard to analyse since delays are affected by capacitances, but we saw this effect rather consistently.

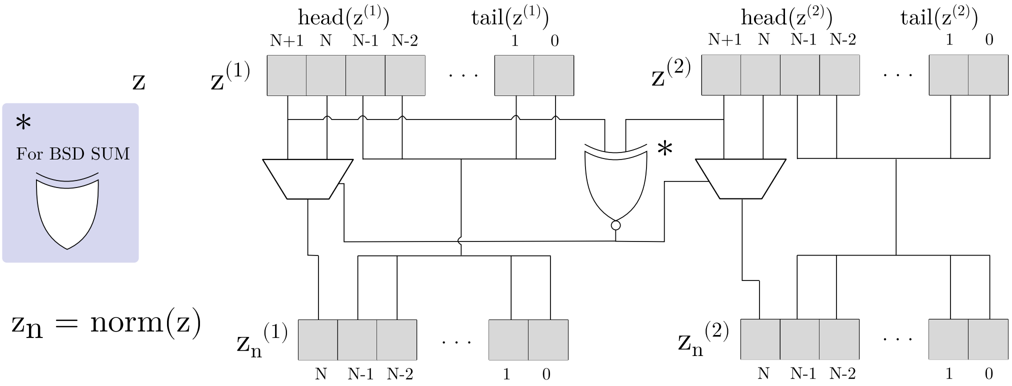


FIGURE 12. Mux-based normalization scheme.

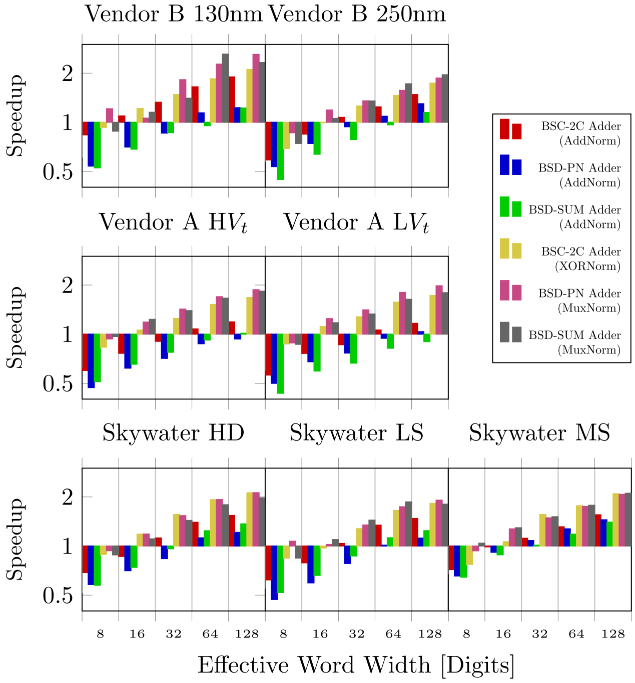


FIGURE 13. Speedup over binary adder for normalized redundant adders.

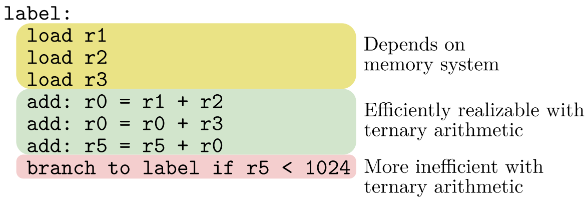
Thereby, the greater speedup of Vendor B 130nm can be explained.

E. IP LIBRARY

As one contribution of this paper, we build our own IP library containing the presented arithmetic circuits (with different encodings) and their corresponding total and partial normalization as RTL code in VHDL. Using this IP library, it is easily possible, to "ternarize" a given circuit to benefit from a redundant number representation. Due to high configurability of the arithmetic IP cores, the internal encoding of them can be changed easily. This can be used for design-space exploration, which number representation/encoding fits best for a given application. Converter IPs from/to classical 2's complement are also available for easy using of these IPs.

III. RISC-V3: A TERNARY RISC-V ARCHITECTURE

In the previous section, the ternary arithmetic used in this paper was presented and characterized. Now, the application of the ternary arithmetic within a CPU is of interest. Thus, a detailed discussion about the challenges of integrating a ternary data path into a conventional processor pipeline is needed and presented in this chapter. As shown, the presented arithmetic circuits are especially efficient for performing additions and subtractions. Unfortunately, additional overhead is required for comparison operations since determination of the sign from the two bit vectors of a ternary number is not possible in constant time. Moreover, logical operations which are working on single bits need a conversion to 2's complement (in other words: a complete renormalization). In summary, while most arithmetic operations of the CPU become faster, additional time has to be spent for total renormalization, if an instruction needs the binary representation of a number. Thus, it is expected that a ternary processor pipeline is capable of executing some instructions faster but some slower than a binary processor pipeline. In Fig. 14 a pseudo assembler code is shown that implements a simple loop as it could be found in real-world applications. Here, the shown instructions are grouped in three annotated categories describing if they can be efficently implemented in a ternary CPU pipline. Consequently from our findings on which operations are efficient and which are inefficient with ternary arithmetic, it can be for example identified, that conditional branches are mostly inefficient due to the determination of the sign which implies a full renormalization. On the other hand, the arithmetic operations really benefit from a ternary data path. It is expected that the improvements of the arithmetic operations carry more weight than the slowness of the conditional branches. The efficiency of stores is depending on the memory system and whether it is capable of handling ternary addresses and data. When using ternary data paths the data to store and the addresses must be converted back to a binary representation when using a conventional binary memory system making it more expensive than before. Avoiding this conversion is done by assuming a ternary memory system for the processor.



```
label:
  load r1
  load r2
  load r3
  add: r0 = r1 + r2
  add: r0 = r0 + r3
  add: r5 = r5 + r0
  branch to label if r5 < 1024
```

Depends on memory system

Efficiently realizable with ternary arithmetic

More inefficient with ternary arithmetic

FIGURE 14. An example pseudo code of a simple loop showing instructions that benefit and that don't benefit from ternary arithmetic.

Calculating condition codes, e.g. if a number is greater or smaller than another number or if a number is equal to zero has more complexity when using a ternary data path. Hence, instructions set architectures (ISAs) using a flags register which has to be updated for every arithmetic operation are more difficult to extend with a ternary data path. For this paper, the RISC-V ISA is used which only needs condition codes calculated for conditional branch instructions. It is an open ISA that makes it very suitable for utilizing it within research [19], [20]. The RISC-V processor architecture of this paper with the adjusted arithmetic is called RISC-V3 since it is compatible to the RISC-V ISA but contains a ternary data path. Since the main focus of this paper lies on the integer data path, the RV64IMC variant of the instruction set is realized. It includes the basic integer instructions with a register width of 64 bits (RV64I), multiplication and division operations (M), and implements the compressed instruction extension (C) that allows instructions to have different byte lengths. With this ISA variant, it is possible to run many real-world applications or benchmarks that use integer arithmetic. In the following, an analysis about the effects of these changes on the instructions offered by a RISC-V CPU is presented.

A. IMPACT ON INSTRUCTIONS

Table 3 groups the instructions offered by RV64IMC into categories that define how a traditional RISC-V pipeline is affected when adding support for a ternary data path. Similar instructions that differ only depending on their operand type (like `add`, `addi`, `addw` and so on) are abbreviated by a star ($\star$).

Instructions that do not read a register are *unaffected* by a changed data path. Writes to the register file without reads from it can directly be realized due to binary numbers being also valid ternary numbers. This allows setting the target register with just the binary number as it would be done in case of a binary data path.

As soon as an instruction uses the contents of a register as an *address* to memory, it depends on the memory system whether a special handling is required. Additionally, for *stores* the data that is saved has to be in a format that is compatible with the memory, in addition to the address. Loads do not suffer from this disadvantage because all numbers in 2's complement representation are trivially convertible to our representations. For example for BSC-2C this is done by the means of constant 0's and direct wiring without incurring any significant latency. Due to the fact that the program counter addresses instructions, the source operand register used by `jalr` might have to be converted to binary, if a memory system is used that does not support ternary addresses. But for this paper, it is assumed that a ternary addressable memory is present for the RISC-V3 processor that also allows saving ternary data. This can be achieved for example by offering additional memory cells.

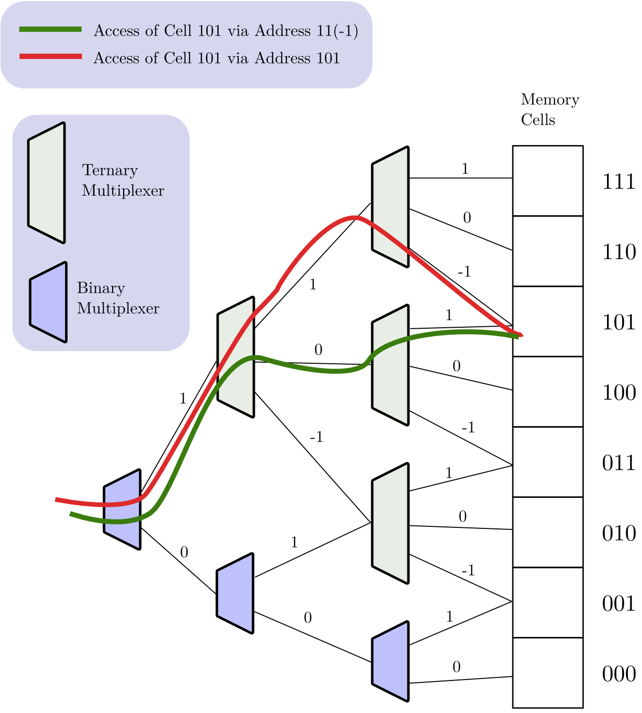


FIGURE 15. A decoder for a ternary memory system, here shown for BSD.

Additionally for the memory being able to use ternary numbers as addresses, we assume an approach based on a ternary memory decoder from [21], as shown in Fig. 15. The depicted decoder selects a ternary memory cell from the available memory cells. In comparison to a conventional decoder, it requires multiplexers with three inputs to build the multiplexer tree instead of multiplexers with two inputs. The only exception are the bottom-most multiplexers that can still be left with only two inputs, since we assume only non-negative memory addresses. In the figure, two example paths are displayed that point to the same memory cell. While the effective address is exactly the same, the actual ternary number representation is different. Still, this decoder is capable of finding the correct cell with only the small overhead of using bigger multiplexers. Due to the fact that this kind of memory system benefits a ternary CPU tremendously, we assumed the usage of it within this work.

Directly Changed Arithmetic includes the basic arithmetic operations (addition and subtraction) including the `auipc` instructions that adds an immediate to the program counter. These instructions do not need special treatment and can be directly realized with a ternary ALU. Since RISC-V does not require a flags register, like e.g. aarch64 and x86_64 do, the information about whether a result is zero, less than zero or greater than zero is not required to be computed during these operations. This would require an additional on-demand calculation of the register bits as soon as the flags register is read. Therefore, RISC-V is a good fit for using a ternary data path internally.

When a *condition* has to be evaluated (e.g. for a conditional jump), the information if a number is greater than, less than or equal to another number might be necessary. With a binary data path, greater or less comparisons against zero usually mean checking the sign bit and if all bits of a number are zero. In case a number has to be compared against another number, a subtraction can be done beforehand to reduce the problem again to a comparison against zero. If it is of importance if two numbers are equal or not equal, all bits can pairwise be compared with AND gates. A ternary data path does not allow checking a sign bit or comparing bits pairwise since multiple representations might represent the same number. Additionally, the ternary number representations introduced in this paper do not offer a separate sign bit. Adding a sign bit is also not possible without detriment to the complexity class of the circuit delays of the ternary arithmetic. Thus, a final conversion back to a binary number needs to be done. To still benefit from the fast ternary additions, this transformation should be done after the comparing subtraction if it is required. Since a real world program generally contains more arithmetic instructions than jumps, the additional overhead should not make the RISC-V3 CPU much slower. In Sec. V a detailed analysis of the speedup for diverse benchmark programs is performed.

Logical instructions and shifts cannot easily be executed on ternary numbers. They have to be converted to binary numbers due to a different bit representation. This conversion is more expensive than the usual binary logic operation would be. However, especially shifts are often used to avoid the usage of a multiplication instructions which are very slow on a binary processor. On a ternary processor, multiplication can be used since it can be faster than the logical instructions (it has a smaller slowdown).

Moreover, *left shifts* can be realized for ternary numbers with some limitations. Small shift lengths that do not overflow the number can easily be done. Since many shifts only have shift lengths up to three (meaning a multiplication with eight which is the word length on RV64I), shifts on ternary numbers can oftentimes safely be executed. Only in case of an overflow, the calculation would be wrong. To prevent this problem, it is possible to use larger word widths for small shifting lengths. When adding 4 digits, all shifts resulting from offset calculations which are performed in the benchmarks used in this paper can be efficiently realized. By generating instruction logs that give exact statistics for which benchmarks this assumption is correct, it can be concluded that this optimization can be performed on most programs in our benchmark suite. In fact, all the cases for which this was not possible fall into similar categories of programs relying on bit-wise operations and shifts, e.g. cryptography routines. In the evaluation section (V) these cases are handled separately as logic intensive benchmarks to make sure that program correctness is not affected.

The focus of this paper lies on integrating ternary addition and subtraction into processor architectures. Optimizing *multiplication* and *division* instructions is not the target because there is already much prior work that analyzed these operations in great detail [22]–[24]. Some of the division algorithms are even based on the same circuit structures as the adders presented in this paper. For example, dividers relying on the BSD number system are available [25]. Regardless, these operations are necessary in many programs making an integration of function units for them essential. To still have the functionality available, an efficient *binary* multiplier and divider with conversion stages for the input and output numbers are deployed. This approach reaches a speedup of 0.77 and 0.97 for multiplication and division respectively. An implementation adjusted for the use case with ternary numbers is expected to yield a slight improvement over a pure binary circuit. Therefore, the speedup results presented in Sec. V can be seen as a lower bound on the optimal improvement possible with a ternary data path while still being rather accurate. Even without further optimization, the results show an overall speedup for arithmetic-intensive algorithms, since the addition and subtraction instructions in these benchmarks make a greater impact than the branches, logical instructions, multiplications, and divisions. An optimized version investigated in future work is capable of increasing the speedup even more.

Another considered possibility is to use a software implementation for multiplication and division. Since the commonly used approaches are either very logic intensive (using many shifts and bit-wise and) or inefficient (using many additions), these were not considered for this paper in order to avoid skewing the final results.

B. SIMPLE PROCESSOR PERFORMANCE MODEL

With the knowledge of the impact on each instruction, a simple performance model for the RISC-V3 CPU can be defined allowing a performance comparison later on. Since the model should be used to find speedups between processors based on binary and ternary data paths, different arithmetic circuits (as defined in Sec. II) were synthesized and the delay of their critical paths determined. With this data, a speedup (or slowdown in some cases) can be assigned to each group of instructions. For the Vendor B 130nm PDK, the corresponding speedups are shown in the last column of Tab. 3. Address and Store instructions are considered to be unaffected due to the assumed ternary memory. Left shifts are given in the unoptimized (equal to the other logical instructions) and optimized (for small constant shift widths) version as described above. It is apparent again that ternary arithmetic outperforms the binary one for basic calculation operations. Only in the case of logical operations, a binary data path is still faster than the ternary variant. The overall speedup S of a program under test with c_i instructions of type i is determined by (9) where

TABLE 3. Grouping of RV64IMC instructions (based on [19]) and the effects on the data path. The stars (*) abbreviate different instruction variants (like `add`, `addi`, `addiw`, etc).

Group	Instruction	Effect	Speedup with Vendor A 130nm $\left(\frac{t_{\text{binary}}}{t_{\text{ternary}}}\right)$
Unaffected	`lui`, `fence`, `ecall`, `ebreak`, `jal`	None	1.00
Address	`jalr`, `ld/lw*/lh*/lb*`	Depends on the memory	1.00
Stores	`sd/sw/sh/sb`	Depends on the memory	1.00
Directly Changed Arithmetic	`add*/sub*`, `auipc`,	Speedup	2.27
Conditions	`slt*`, `beq/bne/blt*/bge*`	Slowdown	0.69
Logical Instructions	`and*/or*/xor*`, `srl*/sra*`	Slowdown	0.10
Left Shift	`sll*`	None if optimized	1.00
M Extension	`mul*/div*/rem*`	Not investigated in detail	-

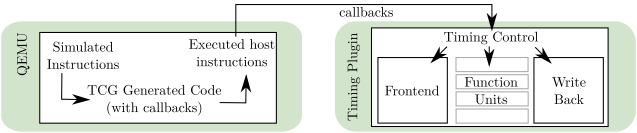


FIGURE 16. Components of the simulator and their interactions.

s_i is the speedup of an instruction of this type:

$$S = \frac{\sum\limits_i c_i s_i}{\sum\limits_i c_i} \tag{9}$$

With this abstract processor model, no exact runtime can be determined. Nevertheless, it is used to find out if the approach of ternary data paths would be beneficial in principle. A detailed analysis of this model applied to a benchmark suite is given in the evaluation section of this paper.

IV. SIMULATION ENVIRONMENT

After the introduction of the abstract architecture model, the next step is to integrate RISC-V3 into a processor simulator to get more exact information about the timing behavior. This includes effects like pipeline hazards that reduce the benefits obtained by instruction parallelism. Simulation can be done on different levels of accuracy trading speed against more accurate results [26], [27]. Therefore, for the investigation presented here, an instruction-accurate processor simulator based on QEMU [28] is extended with a custom cycle-approximate model. Using simulators is a well known technique to estimate nonfunctional properties like the runtime of hardware systems. When using QEMU prediction errors of 8% [29] to 20% [30] on average were previously published. This allows a good estimation of the effects introduced by the changes applied to the microarchitecture.

A. SIMULATION INFRASTRUCTURE

As a basic infrastructure for simulating the RISC-V3 processor, a Just-In-Time compiling emulator called QEMU is deployed [28]. It translates the instructions executed on the simulated CPU into instructions for the processor running the simulator making the emulation very fast in comparison to interpreting the simulated instructions one by one. However, by default it does not contain any means of determining the runtime a real RISC-V3 processor would take for a certain program. Thus, callback mechanisms are utilized to track the executed instructions and update the architectural state of a timing plugin as shown in Fig. 16. This approach is well-known and was already applied before in different kinds of research projects [27], [29], [31]. The mechanism is usually used to execute a callback when a complex instruction is encountered that is not easily implemented with the help of the Tiny Code Generator (TCG) is responsible to dynamically translate guest to host instructions that can be natively run on the host processor. These callbacks tell the timing plugin that a new instruction is encountered requiring updating its internal state. It is responsible for finding out how much time elapsed since the last instruction entered the pipeline and modify the timing information of the frontend, the function units, and the write back stage which contains a reorder buffer. These units are implemented separately and keep track of the details of specific architectural components (like the instruction fetch in the frontend, the ALU in the function units and the common data bus in the writeback stage). The concept of this simulator and its architecture is agnostic with regard to the arithmetic used for the data paths which makes it predestined for the analysis of ternary processors. Unfortunately, instrumenting each instruction by introducing a callback leads to a tremendous slow-down in emulation speed, that is not avoidable, as analyzed before [27]. But it is still much faster than a full RTL-level simulation.

The timing plugin implements a so-called mechanistic model meaning the knowledge about the internal layout of a processor is used instead of a statistical black box approach [32, chap. 4]. It extends the simple architectural model shown in Fig. 2 with an execution scheme that is based on Tomasulo's algorithm [33]. Thereby, the simulator correctly takes timing behaviors of data dependencies, jumps and occupied function units into account. Additionally, the different function units can be modelled and exchanged as required. This allows the creation of "split" function units with special subunits for logical operations, ternary arithmetic and so on. With the exception of the level 1 caches which are oftentimes part of the actual processor pipeline, the latency of the memory hierarchy is currently disregarded. These features allow the implementation of a ternary data path which is described in the following.

B. TERNARY ARITHMETIC

The simulation infrastructure was extended with additional function units that can be configured to use binary arithmetic or ternary arithmetic depending on the investigated arithmetic. Challenges regarding overflow handling and sign detection (required for comparisons and conditional branches) are taken care of as described in the sections above. Additionally, a ternary register file is implemented next to the binary registers in QEMU. This allows a functional verification of the ternary processor due to having binary reference registers but working on ternary arithmetic in the timing plugin. Thereby, it is possible to show that the proposed RISC-V3 architecture is capable of correctly running unchanged RISC-V programs.

Equipped with advanced logging mechanisms, the simulator was used to create detailed statistics about the instruction mixes of the executed programs. These build also the foundations of the previously introduced abstract processor model of Sec. III that requires the instruction counts to calculate a rough speedup estimation. The logs can also be taken advantage of to investigate in more detail about why a benchmark performs well or poorly with ternary arithmetic. An analysis of this aspect is done in Sec. V. Moreover, with the results of this analysis, future optimizations with regard to the code generation of compilers for ternary processors can be implemented. The newly introduced ternary arithmetic is also evaluated to find out potential speed improvements or degradations.

C. TIMING PARAMETERS

The mechanistic model that is used in the simulation framework is a generic model that can be configured by parameters to change its timing behavior. Configurable features include well-known architectural components that are usually found in CPUs exhibiting instruction level parallelism [34, chap. 2]. It is possible to define different properties of the frontend like its width (when a multi-issue system is simulated) and its length (the latency until an instruction reaches its reservation station). The parameters of Tomasulo's algorithm can also be defined. These include the size of the reservation stations, the reorder buffer but also allows disabling elements like the register renaming in case no real out-of-order processor is simulated.

Timings of the function units can also be given to the simulator. These include latencies for binary and for ternary arithmetic which can be switched as required. This allows evaluating different latencies that might result from a different design kit in a real-world processor. While the results created with the simulator are only theoretical, a good estimation about the behavior can be given for the function units with the new arithmetic.

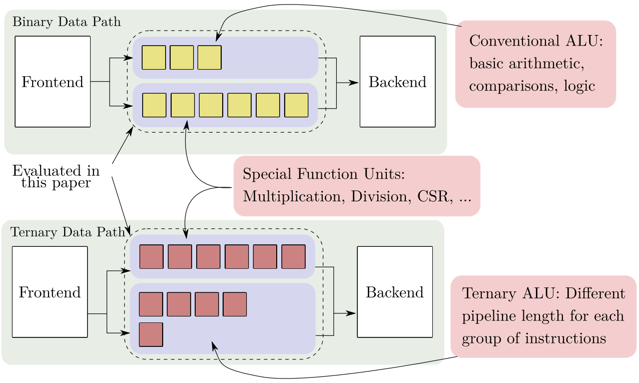

Binary Data Path
Frontend
Backend
Conventional ALU: basic arithmetic, comparisons, logic
Evaluated in this paper
Special Function Units: Multiplication, Division, CSR, ...
Ternary Data Path
Frontend
Backend
Ternary ALU: Different pipeline length for each group of instructions

FIGURE 17. Brief design of the functional units of the processor architectures with binary and with ternary data paths.

TABLE 4. The amount of pipeline stages of the functional units.

Data Path	Function Unit	Amount of Stages
Binary	ALU Multiplication Divison	3 11 128
Ternary	Addition/Subtraction Comparisons / Logical Operations Left Shift (optimized) Multiplication Divison	1 4 1 15 132

The resulting architectures are depicted in Fig. 17. The focus of this paper lies on the function units while the frontend and backend are used without any changes. On the top part of the figure, the processor with a binary data path can be seen. It is assumed that the binary ALU needs multiple pipeline stages (pictured as rectangles) to calculate basic arithmetic, comparisons for branches and the logic. In contrast, the ternary data path shown at the lower part of the figure requires only one pipeline stage for basic arithmetic, but more stages than the binary variant for comparisons and logic. Special function units are not affected in such a great manner and are only displayed for completeness. The amount of stages required for each operation is determined by the circuit delay results of Sec. II when trying to reach a frequency of 2 GHz using the Vendor A 130nm PDK and is given in Tab. 4. These results are utilized for binary and for ternary data paths making the results comparable and allowing the speedup to be calculated. However, this means that the functional units might run in a higher frequency than the surrounding components of the processors. Instructions that calculate an address like `auipc` are also capable of using the function unit for addition and subtraction due to an assumed ternary memory which makes them also more performant on the RISC-V3.

The remaining parts of the CPU are configured roughly based on the Ariane CPU [35] which contains a single-issue out-of-order execute and in-order commit pipeline. Ariane is very suitable for our investigation because it is not a very complex design but still contains important components like a scoreboard. Additionally, a comparison with real hardware (at least for the binary microarchitecture) is possible due to having the Verilog sources of Ariane available. The configuration of the simulated frontend and backend is adjusted for the results of the Vendor A 130nm technology as analyzed in Sec. II which were already utilized to create the relative runtimes for the simple model in Sec. III and the stage count of the function units. In comparison to the 22nm FDSOI technology used by Zaruba and Benini [35], it is a rather big PDK but still allows determining the increased performance of a ternary data path. With the selected parameters, a speedup evaluation was done in the following section.

V. EVALUATION

In this section, an evaluation of our idea of a ternary data path in a processor based on the circuit and arithmetic algorithms presented in Sec. II is performed. For this, the simple model of Sec. III and the simulation model of Sec. IV are investigated. In order to measure performance, we do not rely on static metric such as peak operations per second, but on benchmarks. This allows us to assess the performance in different scenarios and create the opportunity for qualitative investigations (i.e. why are some sequences of operations performed faster or slower in comparison to pure binary microarchitectures).

A. EMBENCH

The benchmarking suite used for the evaluation is Embench [36]. Its main advantage is the diverse set of benchmarks implementing real-world algorithms, which can be analyzed independently. This allows identifying applications that benefit from ternary arithmetic and applications that are executed slower on such a processor. In Tab. 5 the benchmarks are listed and a small comment about their underlying algorithm is given. Since the ternary data path approach only focuses on the integer pipeline of a processor, the `minver`, `nbody`, `st`, and `cubic` benchmarks could not be analyzed since they require floating point arithmetic.

By running the suite in the simulator with its logging facilities, it is possible to create a histogram of instructions. Different instructions are impacted in different ways by the changes to the processor arithmetic (compare Sec. III). Therefore, the first step is to look into the ratio of instructions performing better (giving a speedup), worse (resulting in a slowdown), or without a difference (not affected) with the new arithmetic. Fig. 18 shows these portions for each benchmark.

With the exception of `aha-mont64` most benchmarks exhibit more instructions with a speedup than instructions with a slowdown. This facilitates the hypothesis that an overall speedup of these programs is possible with the presented approach. However, since the slowdown and speedups are not the same for all kinds of instructions, no accurate projection about the exact behavior can be given. Interestingly, some benchmarks like the `nettle` algorithms which are expected to perform worse due to being logic-intensive algorithms still have a lot more instructions with a speedup than with a slowdown. Since the slowdown of a logic instruction is bigger than the speedup of an arithmetic one, it might still come to a slowdown in total.

TABLE 5. Benchmarks of Embench and their implemented algorithms.

Name	Description
aha-mont64	Montgomery Multiplication, 128-bit arithmetic
crc32	CRC32 Checksum Calculation
cubic	Cubic Polynomial Solver
edn	MDH WCET Benchmark Suite (Vector Multiplication, FIR and IIR Filters, Lattice Synthesis, Vocoder Codebook Search, JPEG Discrete Cosine Transform)
huffbench	Huffman Compression
matmult-int	Integer Matrix Multiplication
minver	Floating Point Matrix Inversion
nbody	Floating Point N-Body Problem
nettle-aes	Advanced Encryption Standard
nettle-sha256	Secure Hash Algorithm 2
nsichneu	Pr/T-Net Execution
picojpeg	JPEG Decode
qrduino	QR Code Generator
sglib-combined	Data Structure Benchmark (Array, Linked List, Hash Table, Queue, RB Tree)
slre	Regular Expressions
st	Statistic Algorithms
statemate	Electric Window Control
ud	LU Decomposition
wikisort	Sorting Algorithm

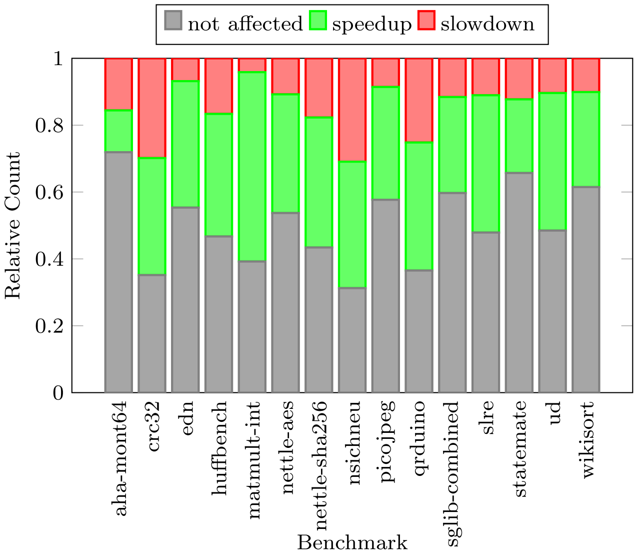


FIGURE 18. Ratios of instructions for each benchmark.

B. PERFORMANCE ESTIMATION (SIMPLE MODEL)

With the extracted instruction counts, the simple model previously introduced can be applied. Since it does not take hazards into account, a simple multiplication of the delays of a certain instruction and the amount of occurrences of it is enough to determine the overall runtime introduced by the opcode. The summation of all of these products gives an overall runtime. Since the delays for the binary and ternary data

TABLE 6. Speedups of the logic intensive algorithms on the simple model.

Design Kit	aha-mont64	crc32	nettle-aes	nettle-sha256	picojpeg
Vendor B 250nm	0.51	0.82	0.68	0.81	0.85
Vendor B 130nm	0.55	0.97	0.78	0.96	1.02
Vendor A HV_t 150nm	0.51	0.81	0.68	0.80	0.84
Vendor A LV_t 150nm	0.52	0.84	0.70	0.83	0.87
Skywater High Density 130nm	0.51	0.83	0.69	0.82	0.86
Skywater Low Speed 130nm	0.52	0.85	0.71	0.84	0.88
Skywater Medium Speed 130nm	0.49	0.79	0.66	0.79	0.83

path are different (see Sec. III), a speedup of the ternary data path in comparison to the binary data path can be calculated.

Due to the expectation that logic intensive algorithms perform worse on the RISC-V3 processor than arithmetic-intensive algorithms, the results are divided into two groups: In Tab. 6 the speedups for all logic intensive algorithms of Embench are shown for the different technologies. It confirms the hypothesis of bad results for this benchmark class. The Vendor B 130nm technology gives the best results when switching from a binary to a ternary processor. However, the `aha-mont64` only reaches around half the speed. As can be seen in Tab. 18, this benchmark has one of the worst beneficial and detrimental instruction ratio. For `picojpeg` even a small speedup for one of the used technologies can be observed.

Although the first results do not show much improvements, in Tab. 7a) the remaining benchmarks are shown. As discussed before, they are arithmetic intensive meaning that the advantages of the ternary arithmetic can be demonstrated. By using the Vendor B 130nm libraries, all of the benchmarks experience a speedup greater than one with a ternary arithmetic in comparison to a binary one. This means that with this simple model, the presented approach is beneficial for the speed of a processor. Yet for the other technologies, much less speedup can be observed. Still in case of a slowdown as e.g. it happened to `wikisort` for all other libraries, the numbers are not far away from one.

A noteworthy observation is the fact that `slre`, `nsichneu`, and `statemate` receive the greatest speedups. These algorithms are no classical arithmetic benchmarks (as e.g. `matmult-int` would be), but contain state machines with complex state transitions. While they do not have a compact loop kernel, they have more conditions and random memory accesses. They benefit tremendously from the fast ternary address calculation. The more well-known arithmetic algorithms with usual loop kernels also profit from the ternary memory but have more offset calculations including left shifts. Thus, they directly follow these state machines (like `edn`). Due to the unoptimized multiplication instructions, `matmult-int` does not perform as well as other algorithms on the ternary CPU. But on the best technology, a speedup can still be observed. Still, sorting has the worst speedup of these benchmarks due to a short loop body with many comparisons.

In Sec. III an optimization was presented that allows left shifts directly on the ternary bit vectors. When realizing this adjustment, much better results can be observed as seen in Tab. 7b). Usually, left shifts are used to calculate byte offsets that are multiples of a power of two. These small shifts can be implemented efficiently even with ternary arithmetic as long as no overflow occurs which is most of the time the case. Especially `matmult-int` and `ud` benefit tremendously from this improvement which are very arithmetic intensive and access series of addresses that have to be calculated. But also the other benchmarks have an increased speedup. Now, only `qrduino` is left with slowdowns on certain technologies. On the other hand, the previously very well performing algorithms `statemate`, `nsichneu`, and `slre` are only slightly affected. However, they already perform great without this optimization when running them on a ternary data path. From these findings, it can be concluded, that there are some algorithms that do not benefit from the proposed approach while others experience a noteworthy boost in speed. Thus, using a ternary data path can be seen as an application-specific acceleration.

C. PERFORMANCE ESTIMATION (SIMULATION MODEL)

While the simple performance model already gives a rough estimation of the expected speedups, it still does not completely map real-world challenges within processors like pipeline hazards or out-of-order execution. Thus, the more accurate simulation model presented in Sec. IV is applied. For this model, only the latencies of the best performing technology (Vendor B 130nm) were employed. Likewise with the simple model, the binary and ternary results were independently determined and then the speedup was calculated. Tab. 7c) shows the results given by the simulation. The left shift optimization is already used.

The results are similar to the results of the simple models with some benchmarks experiencing higher and other lower speedups. However, the numbers are more realistic since processor characteristics like pipelining, hazards, branch prediction, caches and out-of-order execution are implemented in the simulation model. The simulator used for this comparison has an average error of 10% for the used binary configuration in comparison to the Verilog model of the Ariane CPU. This error was determined by manually comparing the runtimes determined by the Verilog model with the runtimes estimated by the simulator for a binary data path. Since only the parameters affected by the ternary data path are adjusted, the comparison between the binary and ternary CPUs gives a very good estimate about the runtime changes that can be expected in a real CPU when performing the proposed change to the processor architecture. This can be assumed because the same error affects the binary as well as the ternary data path. Most benchmarks that were performing very well with the simple model exhibit less speedup since the dependencies dominate the maximum achievable speed while some of the worse performing benchmarks get

TABLE 7. Speedup of arithmetic intensive algorithms on a) Simple model with binary shifts, b) Simple model with ternary shifts and c) Cycle-approximate simulation with ternary shifts.

Sim. Type	Design Kit	edn	huffbench	matmult-int	nsichneu	qrduino	sglib-combined	slre	statemate	ud	wikisort	Median	Mean
a) Binary Shifts	Vendor B 250nm	1.09	1.06	0.96	1.20	0.97	1.04	1.27	1.18	1.02	0.95	1.05	1.07
	Vendor B 130nm	1.39	1.34	1.19	1.57	1.19	1.31	1.70	1.55	1.27	1.17	1.33	1.37
	Vendor A HV$_t$ 150nm	1.07	1.04	0.95	1.18	0.95	1.03	1.26	1.17	1.00	0.94	1.04	1.06
	Vendor A LV$_t$ 150nm	1.12	1.08	0.99	1.23	0.99	1.07	1.31	1.22	1.04	0.98	1.08	1.10
	Skywater High Density 130nm	1.11	1.08	0.98	1.23	0.98	1.07	1.31	1.22	1.04	0.97	1.07	1.10
	Skywater Low Speed 130nm	1.12	1.09	1.00	1.22	1.00	1.07	1.30	1.21	1.05	0.98	1.08	1.10
	Skywater Medium Speed 130nm	1.07	1.04	0.94	1.20	0.95	1.03	1.27	1.18	1.00	0.94	1.04	1.06
	Median	1.11	1.08	0.98	1.22	0.98	1.07	1.30	1.21	1.04	0.97		
	Mean	1.14	1.10	1.00	1.26	1.00	1.09	1.34	1.25	1.06	0.99		
b) Ternary Shifts	Vendor B 250nm	1.42	1.26	1.57	1.20	0.98	1.20	1.28	1.19	1.46	1.15	1.23	1.27
	Vendor B 130nm	1.97	1.68	2.29	1.57	1.21	1.57	1.72	1.55	2.06	1.48	1.63	1.71
	Vendor A HV$_t$ 150nm	1.39	1.25	1.54	1.18	0.97	1.18	1.27	1.17	1.44	1.13	1.22	1.25
	Vendor A LV$_t$ 150nm	1.46	1.30	1.62	1.23	1.00	1.23	1.32	1.22	1.51	1.18	1.27	1.31
	Skywater High Density 130nm	1.46	1.30	1.63	1.23	1.00	1.23	1.32	1.22	1.51	1.18	1.26	1.31
	Skywater Low Speed 130nm	1.45	1.29	1.62	1.22	1.01	1.22	1.31	1.21	1.50	1.18	1.26	1.30
	Skywater Medium Speed 130nm	1.41	1.26	1.57	1.20	0.96	1.20	1.28	1.18	1.46	1.14	1.23	1.27
	Median	1.45	1.29	1.62	1.22	1.00	1.22	1.31	1.21	1.50	1.18		
	Mean	1.51	1.33	1.69	1.26	1.02	1.26	1.36	1.25	1.56	1.20		
c) CA	Vendor B 130nm	1.81	1.65	1.74	1.67	1.32	1.65	1.75	1.43	1.54	1.46	1.65	1.60

a higher speedup because of the architectural optimizations as mentioned before. Hence, the ternary data path results in an improved runtime in comparison to a binary data path for the selected algorithms.

We believe that with larger word widths the speedups and slowdowns when using our operations increase, since conventional binary addition becomes more expensive, while ternary addition is unaffected. An equivalent binary system is affected in a way that makes the critical path of all additions longer. Thus, processors made with a ternary data path allow a much wider word width without having any impact on the clock frequency. This allows special purpose CPUs for digital signal processors that have to work on very wide data registers with the same performance as already available processors. With the increased accuracy this makes it possible to replace for example floating point units with ternary integer arithmetic.

VI. DISCUSSION AND CONCLUSION

In this paper, we presented a RISC-V based CPU called RISC-V3 that uses a ternary data path internally. Beginning with basic arithmetic circuits, we made quantitative evaluations about the performance gain in contrast to binary arithmetic. Afterwards, we constructed a complete processor and presented different performance models. Finally we evaluated the performance of our RISC-V3 architecture using a benchmark suite and applied these models. It was possible to show the strengths as well as the challenges of a complete ternary data path inside a processor pipeline.

Our experiments show that on one hand, ternary arithmetic can really boost a CPU's performance, especially if arithmetic intense programs are executed on the architecture. While on the other hand, when logic- and branch-intensive applications run, a slowdown is experienced. We proof this claim by providing quantitative results derived from an RTL implementation.

This paper can be seen as a baseline for a first CPU architecture with ternary data path. For further performance improvements, in this chapter we would like to discuss some approaches how to deal with slowdowns especially for branch- and logic-intensive algorithms. The ideas presented here, will be a good starting point for further investigations.

A. EXTRAPOLATE RESULTS

While in this paper we use benchmarks to evaluate our ternary architecture, a holistic evaluation for more advanced use cases is still missing. We plan to measure speedup also for complex applications such as image processing or video decoding algorithms. Moreover, the focus on this paper was mainly on timing. For a complete implementation also area and energy have to be considered as well.

B. HYBRID NUMBER REPRESENTATIONS

As the results show, branches and logic operations are very slow. Therefore, one possibility would be to provide a hybrid number representation within the CPU with two different data paths. The goal is to use a set of binary registers which are only used for the control path (e.g. variables for counting in a loop), while all other data registers will be ternary as presented in this paper.

C. FLEXIBLE SIMD INSTRUCTIONS

Due to the fact, that the critical path remains constant by changing the word width, more flexible SIMD processing units in the ALU become possible. Today's SIMD units are only able to work on data with a fixed size (e.g. 8, 16 or 32 bit). Obviously, different applications such as the inference of deep neural networks can benefit from more fine grained SIMD parallelization by increasing the speedup while remaining a constant clock frequency for all data widths.

D. ADDITIONAL TERNARY INSTRUCTIONS

Benefiting from a ternary data path also requires providing specialized instructions that exploit the new circumstances. For example, arithmetic shifts can also be done on a ternary number when the digits that are shifted out of range are preserved and can be inspected. This would provide the possibility to create a fast software multiplication implementation.

E. TERNARY AWARE COMPILERS

Current compilers are optimized to benefit from traditional 2's complement arithmetic. E.g. multiplications will be replaced by shifts if possible or bit operations will be used to speed up complex arithmetic calculations. When introducing ternary CPUs also compilers need to be changed: As the results show different kinds of instructions take more time while others become faster. Compilers need to prefer the fast instructions over the slow ones if they can be exchanged. When additional ternary instructions can be provided, they should also be taken into consideration.

With this paper, a good baseline for further investigations was created. Using ternary arithmetic inside RISC CPUs has a great potential to improve the performance of modern CPUs. Other applications of the ternary data paths include accelerators that don't need to use the bad performing instructions. But due to comprehensive impact of the introduced ternary data path, further investigations as described are required.

REFERENCES

[1] A. Avizienis, "Signed-digit numbe representations for fast parallel arithmetic," *IEEE Trans. Electron. Comput.*, vol. EC-10, no. 3, pp. 389–400, Sep. 1961.

[2] B. Parhami, "Generalized signed-digit number systems: A unifying framework for redundant number representations," *IEEE Trans. Comput.*, vol. 39, no. 1, pp. 89–98, Jan. 1990.

[3] T. E. Williams and J. Horowitz, "SRT division diagrams and their usage in designing custom integrated circuits for division," Comput. Syst. Lab., Dept. Elect. Eng., Stanford University, Stanford, CA, USA, Tech. Rep. CSL-TR-87-328, 1998.

[4] D. L. Harris, S. F. Oberman, and M. A. Horowitz, "SRT division architectures and implementations," in *Proc. 13th IEEE Symp. Comput. Arithmetic*, Jul. 1997, pp. 18–25.

[5] B. Parhami, "Carry-free addition of recoded binary signed-digit numbers," *IEEE Trans. Comput.*, vol. 37, no. 11, pp. 1470–1476, Nov. 1988.

[6] A. Weinberger, "4-2 carry-save adder module," IBM Tech. Discl. Bull., Tech. Rep., 1981.

[7] P. Kornerup, "Reviewing 4-to-2 adders for multi-operand addition," *J. VLSI Signal Process. Syst. Signal, Image, Video Technol.*, vol. 40, no. 1, pp. 143–152, May 2005.

[8] M. D. Ercegovac and T. Lang, *Digital Arithmetic* (The Morgan Kaufmann Series in Computer Architecture and Design). San Francisco, CA, USA: Morgan Kaufmann, 2004, pp. 50–135.

[9] Y. Tanaka, Y. Suzuki, and S. Wei, "Novel binary signed-digit addition algorithm for FPGA implementation," *J. Circuits, Syst. Comput.*, vol. 29, no. 09, Jul. 2020, Art. no. 2050136.

[10] R. Thakur, S. Jain, and M. Sood, "FPGA implementation of unsigned multiplier circuit based on quaternary signed digit number system," in *Proc. 4th Int. Conf. Signal Process., Comput. Control (ISPCC)*, Sep. 2017, pp. 637–641.

[11] H. Mahdavi and S. Timarchi, "Improving architectures of binary signed-digit CORDIC with Generic/Specific initial angles," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 7, pp. 2297–2304, Jul. 2020.

[12] S. Wei, "New residue signed-digit addition algorithm," *Adv. Intell. Syst., Comput.*, vol. 1070, pp. 390–396, Jun. 2020.

[13] D. Timmermann and B. J. Hosticka, "Overflow effects in redundant binary number systems," *Electron. Lett.*, vol. 29, no. 5, pp. 440–441, Mar. 1993.

[14] D. Fey, M. Reichenbach, C. Söll, M. Biglari, J. Röber, and R. Weigel, "Using memristor technology for multi-value registers in signed-digit arithmetic circuits," in *Proc. 2nd Int. Symp. Memory Syst.*, New York, NY, USA, Oct. 2016, pp. 442–454.

[15] P. Kornerup and J.-M. Muller, "Leading guard digits in finite precision redundant representations," *IEEE Trans. Comput.*, vol. 55, no. 5, pp. 541–548, May 2006.

[16] G. Jaberipur, B. Parhami, and M. Ghodsi, "An efficient universal addition scheme for all hybrid-redundant representations with weighted bit-set encoding," *J. VLSI signal Process. Syst. Signal, Image Video Technol.*, vol. 42, no. 2, pp. 149–158, Feb. 2006.

[17] T. Noll and E. De Man, "Anordnung zur bitparallen addition von binärzahlen mit carry-save überlaufkorrektur," U.S. Patent 0 249 132 B1, Aug. 14, 1993.

[18] T. G. Noll, "Carry-save architectures for high-speed digital signal processing," *J. VLSI Signal Process. Syst. Signal, Image Video Technol.*, vol. 3, nos. 1–2, pp. 121–140, Jun. 1991.

[19] A. Waterman and K. Asanović, *The RISC-V Instruct Set Manual*, vol. 1. San Francisco, CA, USA: RISC-V Foundation, 2019.

[20] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for RISC-V," EECS Dept., Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2014-146, 2014.

[21] A. Hagelauer, R. Weigel, M. Reichenbach, and D. Fey, "Integrierte memristor-basierte Rechner-Architekturen (IMBRA)," Res. DFG Proposal 53.01–05/16, 2017.

[22] B. Parhami, "Tight upper bounds on the minimum precision required of the divisor and the partial remainder in high-radix division," *IEEE Trans. Comput.*, vol. 52, no. 11, pp. 1509–1514, Nov. 2003.

[23] B. Parhami, "Precision requirements for quotient digit selection in high-radix division," in *Proc. 34th Asilomar Conf. Signals, Syst. Comput.*, 2001, pp. 1670–1673.

[24] C. Yu Hung and B. Parhami, "Generalized signed-digit multiplication and its systolic realizations," in *Proc. 36th Midwest Symp. Circuits Syst.*, Aug. 1993, pp. 1505–1508.

[25] T. Nishiyama and S. Kuninobu, "Divider and arithmetic processing units using signed digit operands," U.S. Patent 4 935 892, Jun. 19, 1990.

[26] S. Rachuj, C. Herglotz, M. Reichenbach, A. Kaup, and D. Fey, "A hybrid approach for runtime analysis using a cycle and instruction accurate model," in *Proc. Int. Conf. Archit. Comput. Syst.* Cham, Switzerland: Springer, 2018, pp. 85–96.

[27] S. Rachuj, D. Fey, and M. Reichenbach, "Impact of performance estimation of fast processor simulators," in *Proc. Simulation Tools, Techn. (SIMUtools)*. Berlin, Germany: Springer, 2020.

[28] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX Annu. Tech. Conf.*, vol. 41, 2005, p. 46.

[29] S.-H. Kang, D. Yoo, and S. Ha, "TQSIM: A fast cycle-approximate processor simulator based on QEMU," *J. Syst. Archit.*, vols. 66–67, pp. 33–47, May 2016.

[30] Y. Luo, Y. Li, X. Yuan, and R. Yin, "QSim: Framework for cycle-accurate simulation on out-of-order processors based on QEMU," in *Proc. 2nd Int. Conf. Instrum., Meas., Comput., Commun. Control*, Dec. 2012, pp. 1010–1015.

[31] D. Thach, Y. Tamiya, S. Kuwamura, and A. Ike, "Fast cycle estimation methodology for instruction-level emulator," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2012, pp. 248–251.

[32] L. Eeckhout, *Computer Architecture Performance Evaluation Methods* (Synthesis Lectures on Computing Architecture). San Rafael, CA, USA: Morgan & Claypool, 2010.

[33] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. Res., Develop.*, vol. 11, no. 1, pp. 25–33, 1967.

[34] J. Hennessy and D. Patterson, *Computing Architecture: A Quantitative Approach*, 4th ed. Amsterdam, The Netherlands: Elsevier, 2007.

[35] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 11, pp. 2629–2640, Nov. 2019.

[36] Embench. *Open Benchmarks for Embedded Platforms*. Accessed: Sep. 17, 2020. [Online]. Available: https://github.com/embench/embench-iot

MARC REICHENBACH received the Diploma degree in computer science from Friedrich-Schiller University Jena, in 2010, and the Ph.D. degree from the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), in 2017. He currently works as a Postdoctoral Researcher with the Chair of Computer Architecture, FAU. His research interests include novel computer architectures, memristive computing, and smart sensor architectures for varying application fields.

JOHANNES KNÖDTEL received the M.Sc. degree in computer science from the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), in 2016. He started as a Scientific Staff with the Chair for Computer Architecture, in 2017. His research interests include influence of arithmetic circuits on different levels of abstraction in the domain of processor design as well as the usage of memristive technologies in such designs.

SEBASTIAN RACHUJ received the B.Sc. degree in computer science from the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), in 2015, and the M.Sc. degree in computer science from the Chair of Computer Architecture, Erlangen, in 2016. Since 2016, he has been working as a Scientific Staff with the Chair of Computer Architecture. His research interests include different approaches of processor simulation offering different speeds and accuracies and on the simulation of heterogeneous systems.

DIETMAR FEY received the Diploma degree in computer science and the Ph.D. degree with his work on using optics in computer architectures from the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), in 1992, and the Habilitation degree from Friedrich-Schiller-University Jena. From 1994 to 1999, he researched at Friedrich-Schiller-University Jena. From 1999 to 2001, he worked as a Lecturer with Siegen University before he became a Professor of computer engineering with Jena University. Since 2009, he has been leading the Chair for Computer Architecture, FAU. His research interests include parallel computer architectures, programming environments, embedded systems, and memristive computing.

• • •