Received January 31, 2022, accepted March 1, 2022, date of publication March 8, 2022, date of current version June 3, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3157879

Novel VLSI Architectures and Micro-Cell Libraries for Subscalar Computations

KUMAR SAMBHAV PANDEY[1,2] AND HITESH SHRIMALI[2], (Senior Member, IEEE)
[1]Department of Electronics and Communication Engineering, National Institute of Technology Hamirpur, Hamirpur 177005, India
[2]School of Computing and Electrical Engineering, Indian Institute of Technology Mandi, Mandi 175005, India

Corresponding author: Hitesh Shrimali (hitesh@iitmandi.ac.in)

This work was supported by the Ministry of Electronics & Information Technology, Government of India, through the Ph.D. Visweswaraya and Young Faculty Research Fellowships (PHD-1120).

ABSTRACT Parallelism is the key to enhancing the throughput of computing structures. However, it is well established that the presence of data-flow dependencies adversely impacts the exploitation of such parallelism. This paper presents a case for a new computing paradigm namely subscalar digital arithmetic which is aimed at mitigating this issue. It proposes to break up atomic data and atomic operations thereon into sub-atomic data fragments and sub-atomic partial operations. Such a break-up exposes hitherto unexploited levels of parallelism by way of allowing overlap of operations even if data-dependent. Surprisingly this enhanced exploitation of latent parallelism comes with a favorable impact on the area-power characteristics of corresponding computing structures which is contrary to common sense. The paper also proposes a novel micro cell library with logic primitives at corresponding subscalar levels. The synthesized circuits for several sequential benchmarks show an order of magnitude improvement in their area-throughput figure-of-merit (FOM).

INDEX TERMS Bit-level parallelism, digital arithmetic, pipelining.

I. INTRODUCTION

Performance of digital systems cannot be arbitrarily enhanced solely by way of exploiting parallelism at data-word boundaries in presence of data-flow dependencies. This phenomenon is nicely captured by one of the cardinal principles of industrial engineering as:

"Unless something is produced, it cannot be consumed."

A deeper inspection of the architectures of arithmetic computing structures, however, reveals that not all the bits of the result are produced simultaneously nor do all the bits of operands are consumed simultaneously in a given operation. Thus it is possible to partially overlap the execution of sequences of operations, even if data-dependent, thereby exposing opportunities for exploitation of sub-word and sub-instruction level parallelism. In this context, a relaxed but still logically correct version of the above quote could be:

"Unless something is fully produced, it cannot be fully consumed; but if something is partially produced it can be partially consumed."

The associate editor coordinating the review of this manuscript and approving it for publication was Thomas Canhao Xu.

Keeping the above principle in perspective, this paper proposes a new methodology for designing computing structures for applications in the field of Very Large Scale Integrated (VLSI) circuits. Data level parallelism, per se, is not a new concept in computer architecture. It has been a topic of academic research [1] as well as it has been applied in many successful industrial implementations [2]–[4]. These efforts are largely targeted at packing similar operations on multiple sets of operands having lesser precision in a single instruction. Our approach, however, does not propose to process lesser precision data in a packed larger atomic operation. We propose to process full precision data by way of some novel sequences of sub-atomic operations. The fields of computer architecture and digital VLSI architecture are intricately interwoven and many common techniques like pipelining and parallelism have been extensively applied in both. However, there is nothing analogous to Single Instruction Multiple Data (SIMD) architectures in digital VLSI. Proponents of SIMD architectures are driven by the desire to pack multiple smaller precision data on wider register files and to pack operations on them in single instructions. In the case of digital VLSI, there is no concept of "instruction" and the circuits are always "application-specific". Moreover, one may note that the operands and results, even if smaller than the size of registers wherein they are densely packed are still atomic.

Operating on sub-atomic data fragments is, however, non-trivial. Though, bit-wise logical operations can be trivially performed in parallel at any sub-atomic data boundary, yet, the case of arithmetic or shift operations is not as straight-forward due to the issue of carry/borrow propagation. This problem of carry/borrow propagation has been studied at length and various designs have been proposed [5]–[11]. The focus of all of these efforts has been on reducing the latency and/or increasing the throughput, albeit at the cost of increased area and power. In contrast, this paper proposes to circumvent the issue altogether. The motivation behind the research becomes obvious in a small example given below in eqn (1), where, data-flow dependencies adversely infringe upon any potential gains of parallelism or pipelining or both. Without loss of generality, assume that a, b, c and s are all 32-bit unsigned integers and the computation is performed using two instances of standard 32-bit adders.

$$s = a + b + c \tag{1}$$

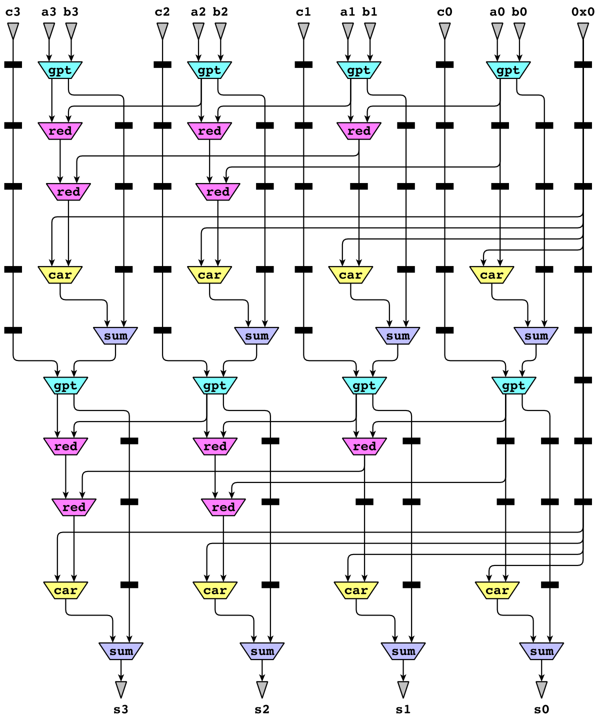

FIGURE 1. Example computation with state-of-the-art parallel prefix adders [5].

Evidently, unless the intermediate result $a + b$ is known, c cannot be added to it to get the value of s. The architecture for the given computation is, therefore, a cascade connection of two 32-bit adders.

One such implementation, where state-of-the-art parallel prefix adders having 8-bit valencies are deployed, is shown in Fig. 1. The semantics of blocks *gpt*, *red*, *car* and *sum* have been borrowed from [5], while the dark small rectangles are pipeline registers. Block *gpt* computes three signals Generate(g_i), Propagate(p_i) and Transmit(t_i) locally at individual bit positions i. For 1-bit valency, they are defined in equations (2, 3 and 4). Block *red* reduces these signals in a tree-like fashion to compute sinals Generate($g_{i..i-n}$) and Propagate($p_{i..i-n}$) over a group of bits i to $i - n$. For 1-bit valency, they are defined in equation (5 and 6). The block *car* computes Carry_out(C_{i+1}) as a function of signals Carry_in(C_i), Generate(g_i) and Propagate(p_i) at individual bit positions i. Once again for 1-bit valency, they are defined in equation (7). Block *sum* computes all the sum bits in parallel according to euation (8). For higher valencies the behaviors are more complex. One can note that the architecture is composed of 34 logic blocks of comparable complexities and 64 8-bit registers. The resulting architecture performs the computation in 10 cycles.

$$g_i = a_i \cdot b_i \tag{2}$$
$$p_i = a_1 + b_i \tag{3}$$
$$t_i = a_i \oplus b_i \tag{4}$$
$$g_{i \cdots i-1} = g_i + p_i \cdot g_{i-1} \tag{5}$$
$$p_{i \cdots i-1} = p_i \cdot p_{i-1} \tag{6}$$
$$C_{i+1} = g_i + p_i \cdot C_i \tag{7}$$
$$s_i = t_i \oplus C_i \tag{8}$$

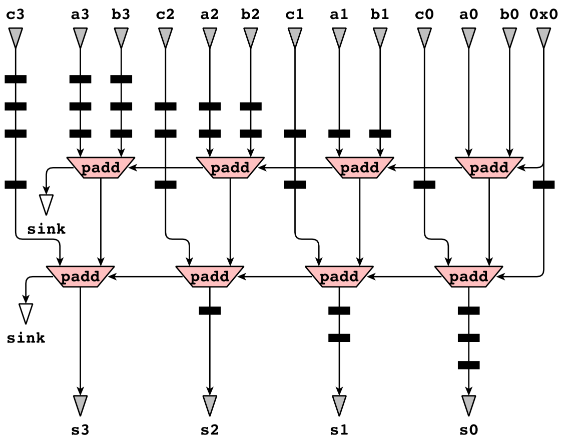

FIGURE 2. Example computation with subscalar adders.

In contrast to this state-of-the-art deeply pipelined architecture, if the same computation is performed using simple 32-bit carry ripple adders having the same valency of 8-bits, the implementation shown in Fig. 2 is obtained. The blocks captioned *padd* are 8-bit full adders and detailed in Section II. They have similar complexity as the logic blocks of the deeply pipelined implementation and the dark small rectangles are the same 8-bit pipeline registers. The resulting implementation not only performs the computation in 5 cycles but also consumes much less Silicon resources. In fact, only 8 logic blocks of comparable complexities and only 29 8-bit registers are needed. Both the example implementations use two adders. Suitable data steering and control logic may be added to implement the examples with fewer resources. The data-path in such a case for the former implementation will need at least 4 instances of block *gpt*, 3 instances of block *red*, 4 instances of block *car*, and 4 instances of block *sum*. In comparison, the later implementation will need only 2 instances of block *padd*!

The area-latency characterization of these logic blocks were estimated with open-source digital Application Specific Integrated Circuit (ASIC) implementation flow OpenLane using sky130_fd_sc_hd standard cell library and are presented in Table 1.

TABLE 1. Area-latency characterization of the example blocks.

Block	Area ($\mu\, m^2$)	Latency ($n\ sec$)
gpt	2180	60
red	2950	40
car	1690	60
sum	560	10
padd	3170	60

Both the designs will run at the same frequency as the slowest stage in any of the implementations takes 60 *nanoseconds*. As the former takes 10 cycles and the later takes only 5 cycles to complete one iteration, the former is slower by a factor of two. The area estimates are 0.06494 mm^2 and 0.02536 mm^2 respectively which is also better by a factor of almost two and a half.

The idea of operating on operands with less precision has an inherent appeal of being faster and at the same time consuming less Silicon resources. Often, even the exactness can be compromised, in case the resulting inexactness is tolerable and the gains in power consumption or Silicon resources are disproportionately higher [12], [13]. Tensor Processing Units [14], [15] and Graphics Processing Units [16] are classic case studies of recent times, where less precise data are used in massively parallel systems. All of these architectures compromise on data width in one way or the other. To the best of our knowledge, there are no systems that preserve the data width and still process smaller fragments of full-width data gainfully to reduce the complexities either in space, in time, or both, with one exception of On-line Arithmetic [17]–[19]. It operates on full precision data in a bit-serial fashion. Both MSB first and LSB first have been proposed. The possibility of overlapped execution of data-dependent operations was explored in these proposals.

Our proposal on subscalar digital arithmetic differs from that on the above mentioned On-line arithmetic in four significant ways.

1) The On-line arithmetic uses redundant representations for the data while in our proposal we use their conventional 2's complement binary representation. This is more intuitive and better suited for hardware implementations. On-line arithmetic even proposes an on-the-fly conversion from redundant representation to conventional representation while our proposal is natively conventional.
2) Individual slices for bit-wise computations are themselves more complex as compared to the very simple logic used in the definition of our processing units. The slices described in On-line Arithmetic necessarily have a multiplexer to select the result bits. This multiplexer can quickly become prohibitively oversized in the case of higher radix implementations. The critical path of computation as well as the Silicon resources used to do so are higher and, thus, may offset the potential gains of overlapped execution of data-dependent operations. The proposed scheme in this paper is devoid of any such issues and, hence, can scale up better for higher radices.
3) The On-line Arithmetic is based on digit recurrence and is capable of computing any general function, whereas our proposal only attempts to implement simple arithmetic operations. Computing any general function using our proposal, therefore, needs to be carried out with combinations of these small arithmetic primitives.
4) The On-line arithmetic is bit-serial, while our proposal is more like systolic arrays [20], where the data is processed parallelly while the computation progresses fragment-by-fragment. The same is obvious in Fig. 2.

Bit-serial On-line Arithmetic, as well as the proposed novel subscalar digital arithmetic, have circumvented the issue of serializing data-dependent operations in similar ways by overlapping their execution bit-by-bit or fragment-by-fragment, thus ensuring that the carry/borrow propagates cycle-by-cycle. Sequential circuits are necessarily recursive in their structure. Pipelining alone cannot help in their high throughput realizations whereas they stand to gain by the use of either of these techniques.

The term ''Subscalar'' for our novel computing paradigm is adapted from a similar term ''Superscalar'' [21] used heavily in the realm of computer architecture. Superscalar processors execute multiple instructions (operations) in a single cycle. In juxtaposition, we propose to perform a single atomic operation in multiple cycles, gainfully though. Consequently, it would be prudent to propose this term.

The rest of the paper is organized as follows. Section II sets the stage by describing subscalar computation as a concept. Corresponding novel micro-cell libraries are presented in section III. Evaluation of the proposal is presented in section IV, while section V concludes the paper and points to future research possibilities.

II. SUBSCALAR COMPUTATION

Subscalar computation is introduced in this section using the following three operations. They are purposefully chosen. It may be noted that the operations op_1 and op_2 are independent of each other, while the operation op_3 is data dependent on the operation op_2.

op_1: $x = f(a, b)$
op_2: $y = f(c, d)$
op_3: $z = f(y, e)$

Without loss of generality, it is assumed that:

1) The data types of the operands and results of all the operations are unsigned integers of 32-bit width.
2) Only one instance of the unit implementing the operation f is available.
3) The unit implementing f can be pipelined in four stages.
4) The function f may (as in case of add, subtract, multiply, etc.) or may not (as in case of bit-wise logical operations) have carry/borrow propagation.

Fig. 3 presents the progress of the computation stated above with time. The progress in case of an unpipelined implementation of the function unit f is presented in Fig. 3(a). As only one instance of the unit is available, all three operations have to be necessarily serialized. This case is taken as the base implementation against which the other implementations are compared.

If the unit implementing function f is pipelined in 4 stages, the operation op_2 can start execution as soon as the first stage of the unit is free after it has executed the first phase of the operation op_1. The execution of operation op_3, however, has to stall till the operation op_2 has finished execution and produced the value of the result y. It is presented in Fig. 3(b). The throughput in the case of this implementation is higher as expected, but due to the presence of data-flow dependence in the computation between operations op_2 and op_3, it is less than the theoretically achievable limit.

The cases of unpipelined and pipelined implementations of the class of functions where no bit to bit signal (carry, borrow, etc.) are propagated across the 4 sub-atomic data fragments (operands as well as results) are shown in Fig. 3(c) and Fig. 3(d) respectively. Such functions could be bit-wise logic operations or even SIMD-like operations on smaller width operands. In these figures, a valency of 8 bits is assumed. Thus a total of 4 such data fragments are shown. Obviously, as the operands are assumed to have a valency of 8 bits, the respective implementations consume less time to finish in comparison to the case where they have a full-width valency of 32 bits. The unpipelined version in Fig. 3(c) executes the operations serially, while the pipelined version executes the operations as shown in Fig. 3(d). It is clarified that these figures are not drawn to scale and the intent is not to convey that the operations on 8-bit data are twice as fast in comparison to the operations on 32-bit data. However, it is universally true that operations on smaller precision operands are always faster to execute as compared with similar operations on larger precision operands.

The last two executions as shown in Fig. 3(e) and Fig. 3(f) are subscalar. In the given case, it is assumed that the function f is such that some signals (carry/borrow etc.) propagate bit-by-bit during its execution. Here also we assume an execution valency of 8 bits, hence, four data fragments and a

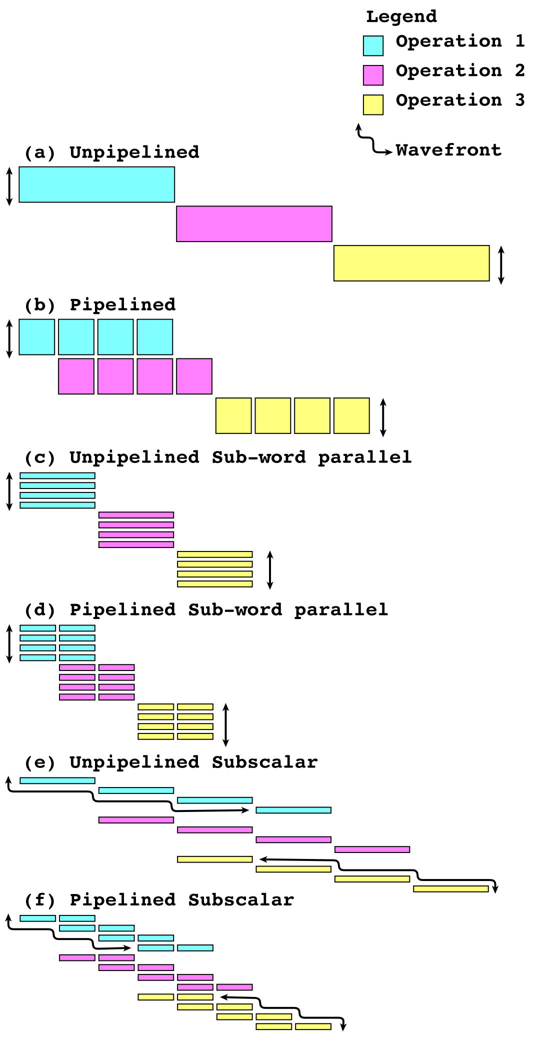

FIGURE 3. Pipelining, sub-word parallelism and subscalar computation.

four-stage execution pipeline. As soon as the least significant data fragment of the function f for operation op_1 gets computed and the carry-like signal from it becomes available, its next more significant data fragment starts getting executed. At the same time, the resource used for computing the least significant data fragment becomes available and thus takes up the execution of the least significant data fragment of the function f for operation op_2. This continues even for the function f for operation op_3 even if there is a data-flow dependence on operation op_2. It may be noted that this was not hitherto possible because the operand and result data were restricted to be atomic. In the case of subscalar computation, this restriction is relaxed. The pipelined subscalar implementation in Fig. 3(f) is obviously swifter as compared to its unpipelined implementation Fig. 3(e). In this context, a new concept of data wavefront is introduced. It is defined as the temporal shape of the data in which it is consumed or produced. It may be noted that the data wavefront in the case of conventional implementations of functional units is a vertical straight line. In the case of subscalar computations, however, it is like a staircase for operands of integer type. It may have any arbitrary step shape for other types of data like floating points, posits, or any other compound data types. Only the following two restrictions need to be applied.

1) The shape of the data wavefront should be the same for all the operands consumed and all the results produced by every processing element.
2) It is preserved throughout any given design entity. Only when the entity's boundaries are crossed the wavefront needs to be reshaped by inserting suitable synchronizing registers.

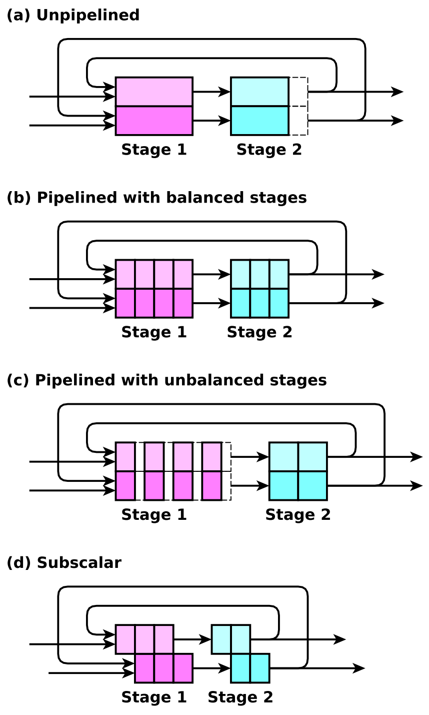

FIGURE 4. Subscalar architecture for a sequential circuit explained.

It may be argued that pipelined implementations also have a potential of increasing the throughput. This is, however, not true for sequntial circuits. Consider for example the case shown in Fig. 4. It is assumed that the forward path of the given sequential circuit is composed of two cascade connected satges. *Stage*1 is assumed to have a latency of 4 time units and *Stage*2 is assumed to have a latency of 3 time units. In synchronous realizations both the stages are clocked at the speed of the slowest stage. In the presence of the feedback path, therefore, successive iterations can only be initiated with an interval of 8 time units as shown in Fig. 4(*a*). If the two stages are pipelined in sub-stages that have latencies of 1 time unit for both the stages, successive iterations can be initiated with an interval of 7 time units. This is shown in Fig. 4(*b*). Unfortunately, if the pipelined sub-stages are unbalanced as shown in Fig. 4(*c*), the throughput of the resulting pipelined implementation may deteriorate. In this case it is assumed that *Stage*2 is pipelined in two sub-stages, each having a latency of 1.5 time units. The pipeline sub-stages of *Stage*1 will have to be slowed down correspondingly. Evidently, successive iterations on the resulting implementation can only be initiated in an interval of 9 time units. A Subscalar implementation of the same circuit is shown in Fig. 4(*d*). Here it is conservatively assumed that the lower half of the operands are operated upon in 3 time units and *carry*/*borrow* like signals are passed on to the part of the unit operating on the upper half the operands in a pipelined way. For the sake of fair comparison, it is again assumed that the total latency of *Stage*1 is 4 time units and that of *Stage*2 is 3 time units. This implementation achieves far better throughput of 5 time units per iteration.

With these foundation and illustrations of subscalar computation as a concept, the reader is invited to revisit Fig. 2. There are two data-dependent additions. We have used two instances of carry-ripple adders having a valency of 8-bits. The synchronizing latches are inserted at the top as well as the bottom where the so-called design entity boundaries are crossed. The shape of the data wavefront produced by the first adder and consumed by the second adder are all the same.

III. MICRO-CELLS: DEFINITION AND APPLICATIONS

Electronics Design Automation (EDA) tool vendors as well as Silicon foundries provide optimized pre-defined layouts for standard logic gates (also known as standard cells) and pre-designed self-contained logic modules (also known as macro cells) in a given technology. In this paper, we propose novel logic blocks at an abstraction that is higher than standard cells but lower than macrocells. We call them micro-cells. These micro-cells may be connected in various combinations and topologies to create useful arithmetic and logic modules like adders, shifters, multiplexers, comparators, etc., from which complete data-paths can be synthesized.

A. DESIGN CONSIDERATIONS

The following two subsections present the design considerations for micro-cells and their applications as constituent blocks of commonly used data-path elements respectively.

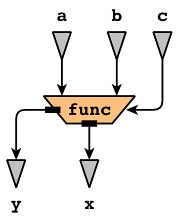

FIGURE 5. A typical micro-cell.

Typically, data-path elements operate on two operands and produce one result for most of the arithmetic and logic functions of interest. As subscalar computation units operate on sub-atomic data and produce partial results, we need to

TABLE 2. Functional semantics of micro-cells.

Cell	Connections	Functional Semantics	Implementation Logic
logic	$c = func$ $b = data\_in\_0$ $a = data\_in\_1$ $x = data\_out$ $y = func$	$if(func_{1..0} = 0)$ $data\_out_{1..0} = (data\_in\_0)'_{1..0}$ $else\, if(func_{1..0} = 1)$ $data\_out_{1..0} = (data\_in\_1 \ \& \ data\_in\_0)_{1..0}$ $else\, if(func_{1..0} = 2)$ $data\_out_{1..0} = (data\_in\_1 \ \| \ data\_in\_0)_{1..0}$ $else\, if(func_{1..0} = 3)$ $data\_out_{1..0} = (data\_in\_1 \oplus data\_in\_0)_{1..0}$	$x_0 = b_0c_1c_0' + a_0b_0c_1'c_0 + b_0'c_1'c_0' +$ $a_0'b_0c_1 + a_0b_0'c_1$ $x_1 = b_1c_1c_0' + a_1b_1c_1'c_0 + b_1'c_1'c_0' +$ $a_1'b_1c_1 + a_1b_1'c_1$ $y_0 = c_0$ $y_1 = c_1$
shift	$b = data\_in$ $c = shamt$ $x = data\_out_0$ $y = data\_out_1$	$data\_out\_0_{1..0} = (data\_in_{1..0} << shamt)_{1..0}$ $data\_out\_1_{1..0} = (data_in)_{1..0} << shamt)_{3..2}$	$x_1 = a_0b_0$ $y_0 = a_1b_0$
mask	$c = func$ $b = data\_in$ $x = data\_out$	$if(func_{1..0} = 1)$ $data\_out = data\_in_0, data\_in_0$ $else\, if(func_{1..0} = 2)$ $x = data\_in_1, data\_in_1$	$x_0 = b_1c_1c_0 + b_0c_1'c_0$ $x_1 = b_1c_1c_0 + b_0c_1'c_0$
padd	$c = carry\_in$ $b = augend$ $a = addend$ $x = sum$ $y = carry\_out$	$sum_{1..0} = (addend_{1..0} + augend_{1..0} +$ $carry\_in_0)_{1..0}$ $carry\_out_0 = (addend_{1..0} + augend_{1..0} +$ $carry\_in_0)_2$	$x_0 = a_0b_0'c_0' + a_0'b_0c_0' + a_0'b_0'c_0$ $x_1 = a_1b_1'b_0'c_0' + a_1'b_1b_0'c_0' + a_1a_0'b_1'c_0' +$ $a_1'a_0'b_1c_0' + a_1a_0'b_1'b_0' + a_1'a_0'b_1b_0' +$ $a_1'b_1'b_0c_0 + a_1'a_0b_1'c_0 + a_1'a_0b_1'b_0$ $y_0 = b_1b_0c_0 + a_1b_0c_0 + a_0b_1c_0 + a_1a_0c_0 +$ $a_0b_1b_0 + a_1a_0b_0 + a_1b_1$
psub	$c = borrow\_in$ $b = subtrahend$ $a = minuend$ $x = difference$ $y = borrow\_out$	$difference_{1..0} = (minuend_{1..0} - subtrahend_{1..0} -$ $borrow\_in_0)_{1..0}$ $borrow\_out_0 = (minuend_{1..0} - subtrahend_{1..0} -$ $borrow\_in_0)_2$	$x_0 = a_0b_0'c_0' + a_0'b_0c_0' + a_0'b_0'c_0 + a_0b_0c_0$ $x_1 = a_1b_1'b_0'c_0' + a_1a_0b_1'c_0' + a_1a_0b_1'b_0' +$ $a_1'b_1b_0'c_0' + a_1'a_0b_1c_0' + a_1'a_0b_1b_0' +$ $a_1'b_1'b_0c_0 + a_1b_1b_0c_0 + a_1'a_0'b_1'c_0 +$ $a_1a_0'b_1c_0 + a_1'a_0'b_1'b_0 + a_1a_0'b_1b_0$ $y_0 = a_1'b_1'b_0c_0 + a_1b_1b_0c_0 + a_1'a_0'b_1'c_0 +$ $a_1a_0'b_1c_0 + a_1'a_0'b_1'b_0 + a_1a_0'b_1b_0 +$ $a_0'b_1$
pmlt	$c = augend$ $b = multiplier$ $a = multiplicand$ $x = product\_low$ $y = product\_high$	$product\_low_{1..0} = (multiplicand_{1..0} \times$ $multiplier_{1..0} + augend_{1..0})_{1..0}$ $product\_high_{1..0} = (multiplicand_{1..0} \times$ $multiplier_{1..0} + augend_{1..0})_{3..2}$	$x_0 = a_0b_0c_0' + b_0'c_0 + a_0'c_0$ $x_1 = a_1'a_0b_1'b_0c_1'c_0 + a_1a_0b_1'c_1c_0 +$ $a_0'b_1b_0c_1c_0 + a_1a_0b_1b_0c_1c_0' +$ $a_1'a_0b_1c_1'c_0' + a_1b_1'b_0c_1'c_0' +$ $a_1a_0'b_1b_0'c_1 + a_1'b_1'c_1c_0' +$ $a_1b_1b_0c_1'c_0 + a_0b_1b_0c_1'c_0' +$ $a_1a_0'b_0c_1' + b_1'b_0'c_1 + a_1'a_0'c_1 +$ $y_0 = a_1a_0'b_1c_1' + a_1a_0b_1'b_0c_0 + a_1'a_0b_1b_0c_0 +$ $a_1a_0'b_1b_0'c_1 + a_1b_1b_0'c_1' + a_1'a_0b_1c_1 +$ $a_1b_1'b_0c_1 + a_0b_0c_1c_0$ $y_1 = a_1a_0b_1c_1 + a_1b_1b_0c_0 + a_1a_0b_1b_0$
comp	$c = comp\_so\_far$ $b = data\_in\_0$ $a = data\_in\_1$ $y = data\_out$	$if(data\_in\_0 = data\_in\_1 \ \& \ comp\_so\_far = 00)$ $data\_out_{1..0} = 00$ $else\, if(data\_in\_0 > data\_in\_1 \ \|$ $data\_in\_0 = data\_in\_1 \ \& \ comp\_so\_far = 01)$ $data\_out_{1..0} = 01$ $else\, if(data\_in\_0 < data\_in\_1 \ \|$ $data\_in\_0 = data\_in\_1 \ \& \ comp\_so\_far = 10)$ $data\_out_{1..0} = 10$	$y_0 = a_1'b_1c_0' + a_1'a_0'b_0c_0' + a_0'b_1b_0c_0' +$ $a_1'a_0'c_1'c_0 + a_0'b_1c_1'c_0 + a_1'b_1c_1' +$ $a_1'b_0c_1'c_0 + b_1b_0c_1'c_0$ $y_1 = b_1'b_0'c_1c_0 + a_1b_0'c_1c_0' + a_1b_1'c_0' +$ $a_0b_1'c_1c_0' + a_1a_0c_1c_0' + a_1b_1'c_1' +$ $a_0b_1'b_0'c_1'c_0' + a_1a_0b_0'b_0c_1'$
mux	$c = select$ $b = data\_in\_0$ $a = data\_in\_1$ $x = data\_out$	$if(select = 00)$ $data\_out = data\_in\_0$ $elseif(select = 11)$ $data\_out = data\_in\_1$	$x_0 = a_0b_1'b_0c_1'c_0' + a_1'b_1b_0c_1'c_0' + a_0c_1c_0 +$ $a_1'b_0c_1'c_0'$ $x_1 = a_1a_0'b_0c_1'c_0' + a_1c_1c_0 + b_1c_1'c_0'$ $y_0 = c_0$ $y_1 = c_1$
demux	$c = select$ $b = data\_in$ $x = data\_out\_0$ $y = data\_out\_1$	$if(select = 00)$ $data\_out\_0 = data\_in$ $elseif(select = 11)$ $data\_out\_1 = data\_in$	$x_0 = b_0c_1'c_0'$ $x_1 = b_1c_1'c_0'$ $y_0 = b_0c_1c_0$ $y_1 = b_1c_1c_0$

have provisions for one more operand (*carry_in*) and one more result (*carry_out*). In the best interest of regular VLSI layouts, we propose micro-cells having uniform interfaces of 3 inputs *a*, *b*, and *c* and 2 outputs *x* and *y* as shown in Fig. 5. Both the outputs are latched in output registers so that they may be used in asynchronous formalism. The valency of all the operands and all the results are kept constant throughout which, in general, may be a bit, pair (2-bit), nibble (4-bit), byte (8-bit), or even half-word (16-bit).

Table 2 details the functional semantics and implementation logic for a micro-cell library having pair valency. The primitives are chosen such that the library is complete (any logic can be realized) and efficient (dedicated cells for commonly used data-path elements). It may be noted that all the

TABLE 3. Area-latency characterization of the proposed micro-cells at various valencies.

Block	Area ($\mu\, m^2$)			Latency ($n\ sec$)		
	2-bit	4-bit	8-bit	2-bit	4-bit	8-bit
logic	330	430	630	30	30	30
shift	330	430	630	30	30	30
mask	330	430	630	30	30	30
padd	450	920	12,680	30	50	60
psub	450	920	12,680	30	50	60
pmlt	1,180	5,620	24,320	60	120	220
comp	80	160	320	90	150	250
mux	180	340	660	20	20	20
demux	180	340	660	20	20	20

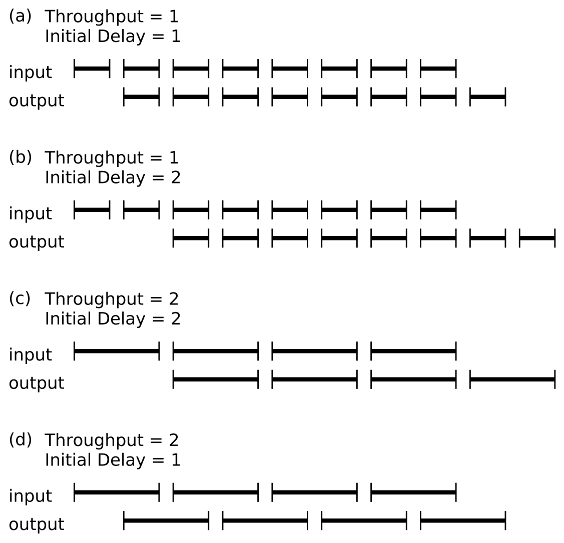

FIGURE 6. Latency, throughput and initial delay.

primitives in the table have 3 inputs and 2 outputs, which may not necessarily be all connected. For other valencies like a nibble, byte, half-word, etc. the implementation logic can be easily derived.

The area-latency characterization of these logic blocks at pair, nibble and byte valencies were estimated with open-source digital ASIC implementation flow OpenLane using sky130_fd_sc_hd standard cell library and are presented in Table 3.

In all, we have defined nine micro-cells. The micro-cells *logic*, *shift* and *mask* are used in realizing bit-wise logical operations. The cell *logic* implements bit-wise logical functions *not*, *and*, *or* and *xor* of the input pairs connected to a and b depending upon whether c is 00, 01, 10 or 11 respectively. The result is read from x and output y is just a delayed copy of the input connected at c. This is useful in creating bit-wise logical circuits at higher data widths as explained in the next subsection. In the case of the operation *not*, the input a is ignored. Any other logic function can be trivially emulated by combinations of the first three. In the case of a nibble or higher valencies, these may even be directly encoded in the input c. The cell *shift* is specially designed to function both as left-shift and right-shift. The input a is left unconnected while b is connected to the data to be shifted and c is connected to the shift amount which is specified in 2's complement representation, +ve for left-shift and −ve for right-shift. The outputs x and y are the least significant bit of the input data padded with a 0 on its right and the most significant bit of the input data padded with a 0 on its left respectively. This cell can be connected as described in the next subsection to implement the *shift* operation for larger data widths. In the case of the cell, *mask* the input a and the output y are left unconnected. The output x is the least(most) significant bit of the input data connected to b replicated in both the bits if the input to c is 01(10). This cell is useful for sign extension operations and is also used as a constituent block for shift operation. Commonly used arithmetic operations *add*, *subtract* and *multiply* are emulated by combinations of the micro-cells *padd*, *psub* and *pmlt*. The micro-cells *padd*(*psub*) are just adders(subtractors) having pair data width and provisions for *carry_in*(*borrow_in*) and *carry_out*(*borrow_out*). The micro-cell *pmlt* performs multiply-add operation on operands having pair valency. The data connected to the inputs a and b are multiplied and their product is added with the data connected at the input c. The output at x is the lower pair and at y is the higher pair of this multiply-add operation. This special design is innovatively used to emulate the *multiply* operation on data of higher widths as described in the next subsection. In VLSI, control flow is achieved through comparators, multiplexers, and demultiplexers. Micro-cells *comp*, *mux*, and *demux* are provided exactly for this purpose. They have conventional functional semantics, except that the second output y is just the delayed third input c. This novel idea comes very handy in creating relevant control flow implementations.

B. APPLICATIONS IN DATA-PATH ELEMENT SYNTHESIS

The micro-cells proposed above may not appear to be useful as stand-alone units but all sorts of useful data-path elements can be emulated by their various novel combinatorial topologies.

To characterize the subscalar designs presented in this subsection we use the same definitions of latency, throughput, and initial delay as used in On-line Arithmetic [17]. However, we augment their definitions to include higher valencies of pair, nibble, and byte along with the valency of single bit used therein. While we describe the subscalar implementations, we use these definitions to highlight any potential gains. Figure 6 presents four different situations in which the definitions are exemplified. In the context of this paper they are formally defined as:

Latency:
Latency is defined as the number of cycles (or equivalently, the total time elapsed) between the first fragment (valency number of bits) of the operands applied and the last fragment of the results produced.
In Figure 6(a) - (d) latencies are 8 cycles each.

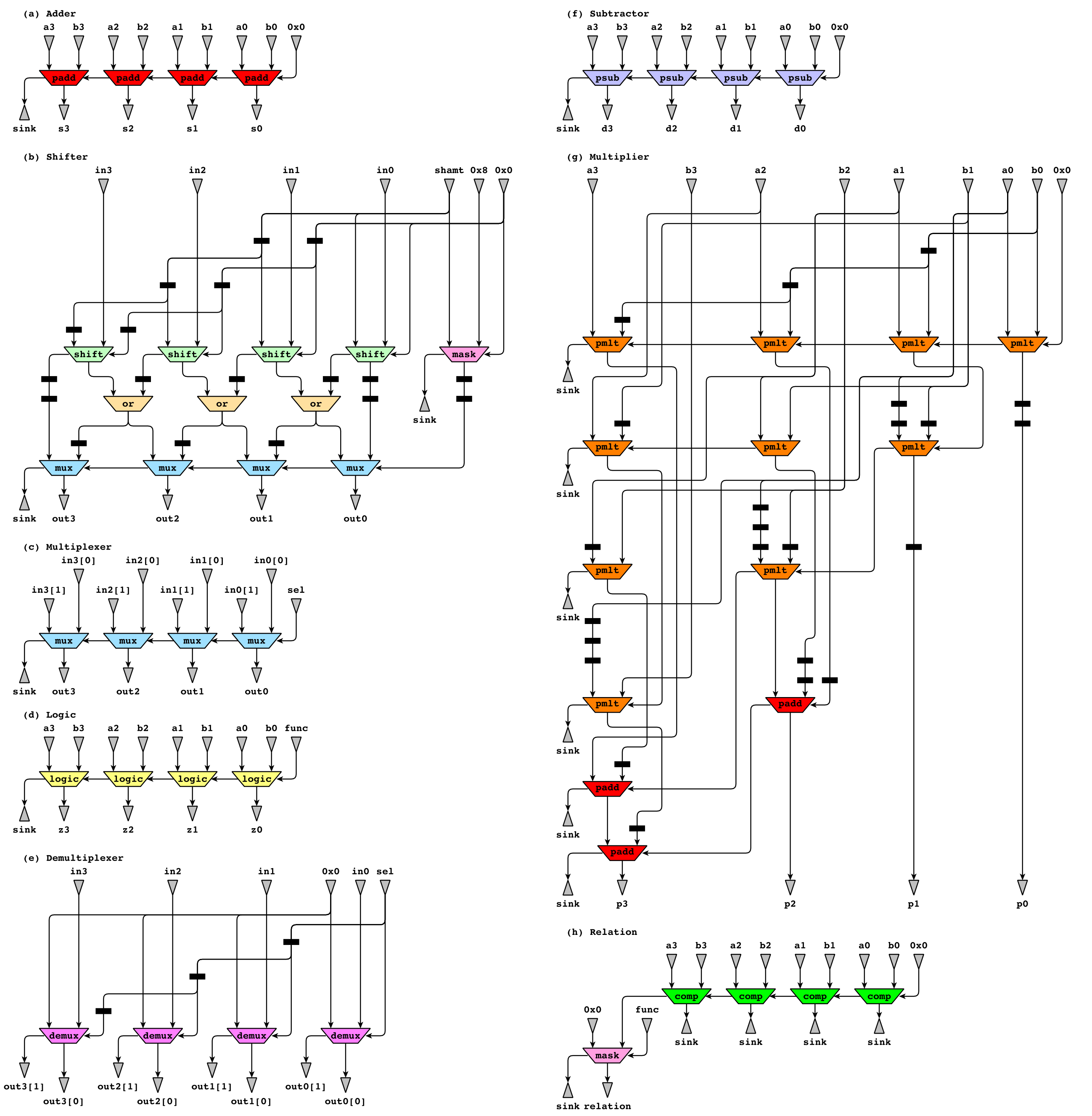

FIGURE 7. Common data-path elements composed of micro-cells.

Throughput:
Throughput is defined as the number of cycles (or equivalently, the total time elapsed) per fragment of opearnds consumed or equivalently, per fragment of results produced. In Figure 6(a) & (b) throughputs are 1 cycle per fragment and in Figure 6(c) & (d) throughputs are 2 cycles per fragment.
Initial Delay:
Initial Delay is defined as the number of cycles (or equivalently, the total time elapsed) between the first fragment of the operands applied and the first fragment of the results produced.
In Figure 6(a) & (d) initial delays are 1 cycle each and in Figure 6(b) & (c) initial delays are 2 cycles each.

Subscalar adder(subtractor) is just a carry(borrow) ripple design at higher valencies as shown in Fig. 7(*a*) and Fig. 7(*f*) respectively. These simple designs have a latency of 4 cycles. But the throughput and the initial delay are just 1 cycle each. This means that the designs are capable of producing results every clock cycle block by block while

TABLE 4. Execution time and area characteristics.

Benchmark	Implementation	Execution time (m sec)			Area (mm^2)		
		8-bit	16-bit	32-bit	8-bit	16-bit	32-bit
diffeq	Unpipelined	0.30580	1.33440	2.66880	0.22212	0.78208	2.89464
	Pipelined-pair	0.39198	0.62550	1.05918	0.17972	0.64856	2.15840
	Pipelined-nibble	0.68944	0.78396	1.25100	0.19016	0.63720	2.05552
	Pipelined-byte	0.30580	1.03416	1.43726	0.25597	0.69402	2.42960
	Subscalar-pair	**0.15846**	**0.19182**	**0.26412**	**0.11353**	**0.40647**	**1.54003**
	Subscalar-nibble	**0.24696**	**0.31692**	**0.38364**	**0.13904**	**0.45476**	**1.63556**
	Subscalar-byte	**0.30580**	**0.45276**	**0.58102**	**0.20151**	**0.57326**	**1.89405**
ellipf	Unpipelined	0.00924	0.01078	0.01232	0.09594	0.22698	0.52650
	Pipelined-pair	0.01584	0.01848	0.02112	0.08788	0.35828	0.48724
	Pipelined-nibble	0.02464	0.03080	0.03696	0.08840	0.33280	0.48880
	Pipelined-byte	0.02792	0.03696	0.04620	0.11518	0.30706	0.84422
	Subscalar-pair	**0.00462**	**0.00462**	**0.00462**	**0.04680**	**0.09360**	**0.18720**
	Subscalar-nibble	**0.00770**	**0.00770**	**0.00770**	**0.04784**	**0.09568**	**0.19136**
	Subscalar-byte	**0.00924**	**0.00924**	**0.00924**	**0.08242**	**0.16484**	**0.32968**
gcd	Unpipelined	0.24948	0.29106	0.33264	0.03829	0.08333	0.18061
	Pipelined-pair	0.28980	0.33120	0.38924	0.03134	0.09778	0.15146
	Pipelined-nibble	0.33120	0.38640	0.44160	0.02874	0.08748	0.14096
	Pipelined-byte	0.24948	0.49680	0.57960	0.03254	0.07983	0.20391
	Subscalar-pair	**0.20790**	**0.31878**	**0.58160**	**0.02344**	**0.04688**	**0.09376**
	Subscalar-nibble	**0.25410**	**0.34650**	**0.53130**	**0.02094**	**0.04188**	**0.08376**
	Subscalar-byte	**0.24948**	**0.30492**	**0.41580**	**0.02624**	**0.05248**	**0.10496**
Kalman	Unpipelined	2.85696	3.33312	3.80928	17.34844	62.54508	235.02924
	Pipelined-pair	1.40160	2.61888	3.05536	13.97560	51.15672	174.61656
	Pipelined-nibble	1.52768	2.80320	4.80128	14.89208	50.42736	166.30016
	Pipelined-byte	2.85696	3.05536	5.13920	20.26164	55.29468	195.94180
	Subscalar-pair	**0.53568**	**1.60704**	**2.41056**	**8.66012**	**32.18564**	**124.61332**
	Subscalar-nibble	**0.49104**	**1.07136**	**1.60704**	**10.82312**	**36.28544**	**132.72592**
	Subscalar-byte	**0.47616**	**0.98208**	**1.07136**	**15.88388**	**45.87336**	**153.68236**
qsort	Unpipelined	1.95432	2.28004	2.60576	0.62436	1.31892	2.78292
	Pipelined-pair	1.00590	1.20708	1.40826	0.55060	1.46624	2.47384
	Pipelined-nibble	0.95002	1.30288	1.62860	0.52282	1.35764	2.36168
	Pipelined-byte	0.96566	1.65542	2.06928	0.56197	1.27734	3.01488
	Subscalar-pair	**0.90658**	**0.92031**	**0.93405**	**0.46844**	**0.93688**	**1.87376**
	Subscalar-nibble	**1.51096**	**1.53385**	**1.55675**	**0.44170**	**0.88340**	**1.76680**
	Subscalar-byte	**1.81316**	**1.84063**	**1.86810**	**0.49645**	**0.99290**	**1.98580**

also consuming the operands every clock cycle block by block. In the case of state-of-the-art pipelined parallel prefix designs, the throughput is 1 cycle but the latency and initial delay are both 5 cycles. Therefore two data-dependent additions(subtractions) will necessarily take 10 cycles to complete, while in the case of subscalar designs they will complete in 5 cycles only (Section I). Moreover, the pipelined parallel prefix designs are composed of 17 logic blocks, while the subscalar designs are composed of only 4 logic blocks of comparable complexity. Huge throughput-area gains are, thus, achievable. The subscalar multiplier architecture is presented in Fig. 7(g). The design is a little irregular in a couple of least significant blocks but is capable of achieving latency of 6 cycles, throughput of 1 cycle, and initial delay of 3 cycles and uses 13 blocks which are all better figures than any conventional design of multipliers. Subscalar bit-wise logical operator can be trivially implemented as a cascade connection of *logic* micro-cells and is presented in Fig. 7(d). It achieves throughput and initial delay of 1 cycle and a latency of 4 cycles while it is composed of 4 blocks. The design of the subscalar shifter presented in Fig. 7(b) is a bit tricky. Recall from the previous subsection that both the left-shift and right-shift partial results are produced at the same time on the two outputs of the micro-cell *shift*. These outputs have zeroes padded on either side. The outputs from adjacent micro-cells are then logically ored and relevant shifted outputs are finally selected by using micro-cells *mux*. This novel scheme works well even for higher valency designs. It achieves a throughput of 1 cycle, a latency of 7 cycles, and an initial delay of 4 cycles. Data-path elements to implement control flow namely comparators, multiplexers and demultiplexers are trivially implemented as cascade compositions of micro-cells *comp*, *mux* and *demux* as presented in Fig. 7(h), Fig. 7(c) and Fig. 7(e) respectively. All of these elements achieve throughput and initial delay of 1 cycle and latency of 5, 4, and 4 cycles respectively.

The solid rectangles in all the data-path elements presented in Fig. 7 are synchronizing latches and the micro-cell *sink* is only a placeholder and meant to indicate that the respective port is left unconnected.

It is worth mentioning that the latencies of data-path elements lose their significance in subscalar designs and are thus irrelevant when used in larger algorithmic data-paths. The throughputs for all the subscalar elements described above are 1 cycle each which helps in achieving overall throughput of 1 cycle for the entire data-path irrespective of the fact whether the sequences of operations are interdependent or not. The initial delay is also small. In the case of dependent operations and the case of loops higher initial delay has a detrimental effect. This does not impact the overall gains too much as evident in the results presented in section IV. All the circuits, however, consume much less area and consequently much less power as compared to their state-of-the-art high-speed implementations.

IV. PERFORMANCE EVALUATION

To estimate possible area-throughput gains, five benchmark circuits were chosen from MCNC Benchmark Suite [22] and Mibench Benchmark Suite [23]. They were chosen to have a judicious mix of both control-dominated and dataflow-dominated circuits. These benchmark circuits are diffeq, gcd, Kalman, ellipf, and qsort. They were synthesized at data-path widths of byte (8-bit), half-word (16-bit) and word (32-bit) and their layouts were generated using open-source digital ASIC implementation flow OpenLane [24], [25] using sky130_fd_sc_hd standard cell library. Their performances concerning die area and total time to execute the benchmarks with their standard input test vectors were recorded. These areas and execution times were taken as the base against which the respective pipelined and subscalar implementations at pair, nibble, and byte valencies were compared.

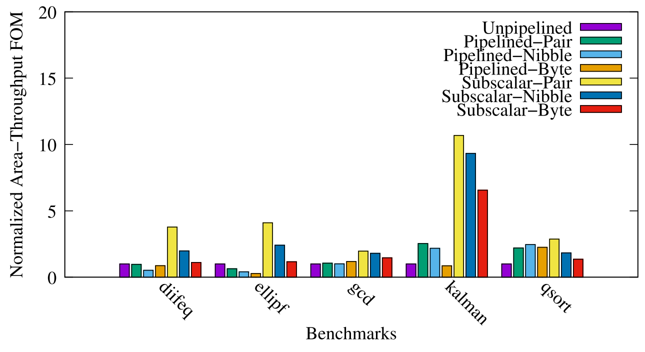

FIGURE 8. **Area-throughput figure-of-merit for the chosen benchmarks for 8-bit data-path.**

The areas and execution times so recorded are presented in Table 4. It may at once be noted that the subscalar realizations are consistently better than their pipelined counterparts both in terms of throughput as well as chip area. It may be noted that the wider the data-paths the higher are the gains. Subscalar designs, however, consistently show higher throughput coupled with smaller Silicon footprints. Often it is impossible to achieve simultaneous improvements in both. This fact for ‘‘Subscalar Computation’’ is highlighted in Table 4.

The area-throughput figure-of-merit (FOM) for the unpipelined, pipelined, and subscalar implementations at pair, nibble, and byte valencies of the chosen benchmark circuits are plotted as histograms in Fig. 8 for an 8-bit data-path width. The corresponding area-throughput FOM for 16-bit

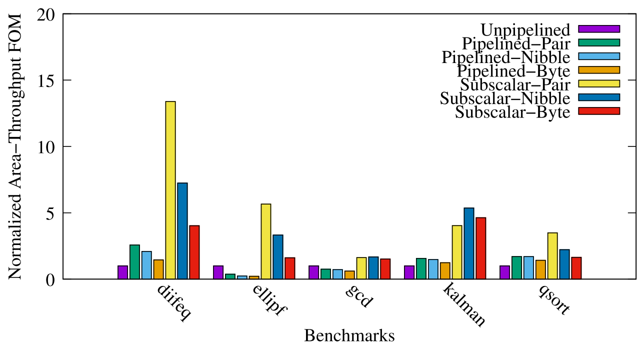

FIGURE 9. **Area-throughput figure-of-merit for the chosen benchmarks for 16-bit data-path.**

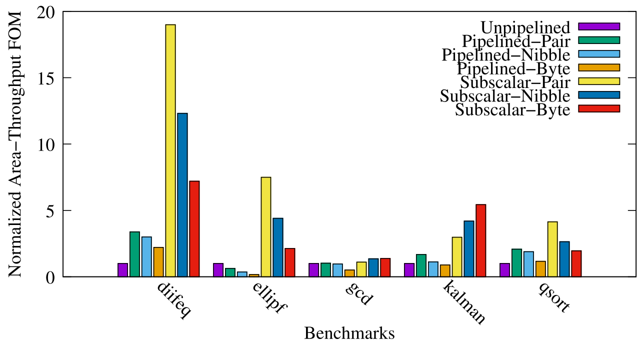

FIGURE 10. **Area-throughput figure-of-merit for the chosen benchmarks for 32-bit data-path.**

and 32-bit data-path widths are presented in Fig. 9 and Fig. 10 respectively. While this FOM is consistently better, in the case of wider data-paths the gains are outstanding. Circuits with wider data-paths of 64-bits or higher will potentially achieve much higher FOM.

V. CONCLUSION

A novel digital hardware synthesis paradigm namely ‘‘Subscalar Computation’’ is proposed and evaluated in this paper. Major contributions of the paper and pointers for future research are summarized below:

1) New data-path synthesis methodology of a partial processing of data at a sub-word boundary is presented.
2) Cell library namely micro-cell library at an intermediate level of complexity and functionality between standard gates and macrocells is proposed. All the cells in the library have a 3-input 2-output interface. They can be implemented as hardwired circuits or as lookup tables or even as coarse grain reconfigurable logic.
3) Designs of a few commonly used data-path elements composed of elements chosen from the proposed micro-cell library are presented. The designs are our initial proposals and by no means the only possibilities. One may think about improvements over them and may even extend the designs with more functions of use like

dividers, floating-point processors, function evaluators, GPUs, etc.

4) The research can be extended towards the development of new EDA tools and can also be adapted to be used in reconfigurable devices.
5) The proposal has been evaluated on several standard benchmark circuits and corresponding area-throughput gains are presented. The idea can also be potentially deployed in general-purpose computing systems ranging from embedded to data center scale.

REFERENCES

[1] W. D. Hillis, "The connection machine," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., MIT, Cambridge, MA, USA, 1985.

[2] G. Conte, S. Tommesani, and F. Zanichelli, "The long and winding road to high-performance image processing with MMX/SSE," in *Proc. 5th IEEE Int. Workshop Comput. Archit. Mach. Perception*, Sep. 2000, pp. 302–310.

[3] R. B. Lee, "Subword parallelism with MAX-2," *IEEE Micro*, vol. 16, no. 4, pp. 51–59, Aug. 1996.

[4] M. Tremblay, J. M. O'Connor, V. Narayanan, and L. He, "VIS speeds new media processing," *IEEE Micro*, vol. 16, no. 4, pp. 10–20, Aug. 1996.

[5] K. S. Pandey, D. Kumar, N. Goel, and H. Shrimali, "An ultra-fast parallel prefix adder," in *Proc. IEEE 26th Symp. Comput. Arithmetic (ARITH)*, Jun. 2019, pp. 125–134.

[6] I. Koren, *Computer Arithmetic Algorithms*, 2nd ed. Natick, MA, USA: A. K. Peters, 2001.

[7] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*. New York, NY, USA: Oxford Univ. Press, 2000.

[8] M. D. Ergecovac and T. Lang, *Digital Arithmetic*. New York, NY, USA: Morgan Kaufmann, 2003.

[9] T. Han and D. A. Carlson, "Fast area-efficient VLSI adders," in *Proc. IEEE 8th Symp. Comput. Arithmetic*, Como, Italy, May 1987, pp. 49–56.

[10] S. Knowles, "A family of adders," in *Proc. 14th IEEE Symp. Comput. Arithmetic*, Apr. 2001, pp. 277–281.

[11] A. Beaumont-Smith and C.-C. Lim, "Parallel prefix adder design," in *Proc. 15th IEEE Symp. Comput. Arithmetic*, Jun. 2001, pp. 218–225.

[12] S. Leyffer, S. M. Wild, M. Fagan, M. Snir, K. Palem, K. Yoshi, and H. Finkel, "Doing Moore with less–leapfrogging Moore's law with inexactness for supercomputing," in *Proc. 3rd Int. Workshop Post-Moore's Era Supercomput. (PMES)*, 2017, pp. 1–9.

[13] P. Düben, J. Schlachter, P. S. Yenugula, J. Augustine, C. Enz, K. Palem, and T. N. Palmer, "Opportunities for energy efficient computing: A study of inexact general purpose processors for high-performance and big-data applications," in *Proc. Des., Autom. Test Eur. Conf. Exhib. (DATE)*, 2015, pp. 764–769.

[14] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, and S. Bates, "In-datacenter performance analysis of a tensor processing unit," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 1–12.

[15] J. Dean, D. Patterson, and C. Young, "A new golden age in computer architecture: Empowering the machine-learning revolution," *IEEE Micro*, vol. 38, no. 2, pp. 21–29, Mar. 2018.

[16] S. Mittal and S. Vaishay, "A survey of techniques for optimizing deep learning on GPUs," *J. Syst. Archit.*, vol. 99, Oct. 2019, Art. no. 101635. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762119302656

[17] M. D. Ercegovac, "On-line arithmetic: An overview," in *Real-Time Signal Processing VII*, vol. 495, K. Bromley, Ed. Bellingham, WA, USA: SPIE, 1984, pp. 86–93, doi: 10.1117/12.944012.

[18] M. D. Ercegovac, "On left-to-right arithmetic," in *Proc. 51st Asilomar Conf. Signals, Syst., Comput.*, Oct. 2017, pp. 750–754.

[19] Z. Huang and M. D. Ercegovac, "High-performance low-power left-to-right array multiplier design," *IEEE Trans. Comput.*, vol. 54, no. 3, pp. 272–283, Mar. 2005.

[20] H. T. Kung, "Why systolic architectures?" *Computer*, vol. 15, no. 1, pp. 37–46, Jan. 1982.

[21] M. Johnson, *Superscalar Microprocessor Design*. New York, NY, USA: Prentice-Hall, 1991.

[22] K. Koźmiński, "Benchmarks for layout synthesis–evolution and current status," in *Proc. 28th Conf. ACM/IEEE Des. Autom. Conf.*, 1991, pp. 265–270.

[23] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proc. IEEE Int. Workshop Workload Characterization*, Sep. 2001, pp. 3–14, doi: 10.1109/WWC.2001.15.

[24] M. Shalan and T. Edwards, "Building OpenLANE: A 130 nm openroad-based tapeout-proven flow," in *Proc. 39th Int. Conf. Comput.-Aided Des.*, Nov. 2020, pp. 1–6.

[25] R. T. Edwards, M. Shalan, and M. Kassem, "Real silicon using open-source EDA," *IEEE Design Test*, vol. 38, no. 2, pp. 38–44, Apr. 2021.

KUMAR SAMBHAV PANDEY received the B.Tech. (Hons.) degree in electrical engineering & electronics from the Malaviya National Institute of Technology Jaipur, the M.Eng. degree in computer science & engineering from the Indian Institute of Science, Bengaluru, and the joint M.T.D. degree in embedded systems from the National University of Singapore and the Eindhoven University of Technology, Eindhoven, The Netherlands. He is currently pursuing the Ph.D. degree in the area of processor design with the Indian Institute of Technology Mandi.

He is also working as an Associate Professor with the Department of Electronics & Communication Engineering, National Institute of Technology Hamirpur. He has guided more than two dozen graduate theses and published about 60 papers at national and international levels. His research interests include the domains of computer architecture, reconfigurable computing, CAD of digital systems, embedded systems, digital arithmetic, performance evaluation, and formal verification.

HITESH SHRIMALI (Senior Member, IEEE) was born in Ahmedabad, India. He received the Bachelor of Engineering (B.E.) and Master of Technology (M.Tech.) degrees in instrumentation engineering from IIT Kharagpur, India, and the Ph.D. degree in analog and mixed-signal VLSI design from IIT Delhi, India. He was with the Analog and Mixed-Signal Group, TRnD Department, Analog-To-Digital Converter Team, STMicroelectronics, India, for two years, from August 2011 to June 2013. From June 2013 to December 2014, he was a Postdoctoral Researcher with the University of Milan. From December 2014 to May 2019, he has worked as an Assistant Professor with IIT Mandi, where he is currently working as an Associate Professor. His current research interests include design and testing of radiation hard circuits (CMOS silicon detectors), analog and mixed-signal VLSI design (ADCs), the modeling of radiation effects on analog and mixed signal circuits, and on-chip instrumentation.

He has served as a TPC Member for IEEE EDAPS and VLSID 2018. He has served as an Organizing Committee Member and a Fellowship Chair for IEEE EDAPS 2018 and VDAT 2019, respectively. He was a recipient of the Young Faculty Research Fellowship (YFRF), MeitY, Government of India, from January 2019 to December 2024, and the Distinguished Alumni Award 2017 from the Instrumentation Engineering Department, Nirma University, Ahmedabad. He has served as a Reviewer for the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—I: REGULAR PAPERS (TCAS-I), the *Journal of Circuits, Systems and Signal Processing* (Springer), IETE, IEEE ISCAS, IEEE VLSID, and MWSCAS.