Design of synchronous and asynchronous variable-latency pipelined multipliers

Mauro Olivieri *

Abstract: **This paper presents a novel variable-latency multiplier architecture, suitable for implementation as a self-timed multiplier core or as a fully synchronous multi-cycle multiplier core. The architecture combines a 2nd order Booth algorithm with a split carry save array pipelined organization, incorporating multiple row skipping and completion-predicting carry-select final adder. The paper reports the architecture and logic design, CMOS circuit design and performance evaluation. In 0.35 μm CMOS, the expected sustainable cycle time for a 32-bit synchronous implementation is 2.25 ns. Instruction level simulations estimate 54% single-cycle and 46% two-cycle operations in SPEC95 execution. Using the same CMOS process, the 32-bit asynchronous implementation is expected to reach an average 1.76 ns throughput and 3.48 ns latency in SPEC95 execution.**

I. INTRODUCTION

Fast integer multipliers are a key topic in the VLSI design of high-speed microprocessors. Recent results have shown that through a careful full-custom CMOS design a 54x54 bit multiplication in less than 3 ns is possible [21]. However, with commonly available CMOS processes, micro-architectures with 2 ns cycle time are commercially available [28]. As a result, due to the registers' setup and hold times, even a fast 32 bit multiplication may not fit in a single cycle, and the design of pipelined multi-cycle multipliers is a common design choice to avoid the whole microarchitecture be limited by a relatively slow multiplier.

Data dependency always puts a limitation to the throughput of pipelined arithmetic units [22], due to idle cycles between consecutive dependent operations. To overcome this, synchronous variable-latency pipelined *addition* units have recently been proposed in DSP industrial design [30]. A variable latency unit operates as a normal pipelined unit, but for most operands it can complete its operation in a single cycle, thus avoiding idle cycles insertion and improving the average throughtput. A synchronous signal flags in which cycle the operation has completed. A more aggressive implementation of this idea is inherent in asynchronous design, with self-timed units capable of an average response faster than the worst case [6][9][14] [25][29][39][52].

* The author is with the Dept. of Electronic Engineering - University of Rome "La Sapienza" – Italy. Email: olivieri@die.ing.uniroma1.it

In a self-timed arithmetic unit, a totally asynchronous completion signal flags in which instant the operation has completed. A practical problem of such a totally asynchronous unit is the interface with a synchronous microprocessor architecture, due to the metastability effects of signal synchronization [2]. This fact has given an impulse to the design of totally asynchronous microarchitectures, where important results have already been achieved [16][17][43][44][47][48]. While conceptually similar, synchronous variable-latency units and self-timed units substancially differ in their architecture and VLSI design[1].

Though variable-latency multiplication algorithms are historically well known (e.g. shifting over zeros after Booth encoding [22][4]), modern high-speed VLSI synchronous implementations have addressed fixed-time multipliers [19][21], for they best fit the conceptual design of a synchronous, fixed-latency instruction set architecture [22]. In fact, variable latency is present in some non-pipelined multi-cycle multiply units based on an iterative sequential algorithm, targeting low-cost CPUs [35]. Another example of this approach is the 8-bit multiplier recently presented in [45]. *Synchronous* variable-latency has also been proposed for addition [32] and implemented in a high-speed pipelined VLSI adder [30]. No specific work has addressed synchronous variable-latency multipliers targeting high speed.

On the other hand, several research works have addressed *asynchronous* VLSI multipliers. An early example of the variable latency concept with an asynchronous implementation is in [36]. Some studies address serial asynchronous multipliers, with a low speed target [13][46], while several studies target asynchronous (i.e. unclocked) design but not variable latency [1][33][7][11][38] (with the goals of reducing power consumption, avoiding clock distribution, etc.), or they partially implement variable latency, usually in the final carry propagation adder of an array multiplier [31][37][10]. A few works do primarily address variable latency [40][20][26]: in [40] and [20], the target is not absolute speed, but rather the trade off between area, power and speed. In [26], Kearney and Bergmann present a design with conceptual similarities with this work, i.e. variable-latency carry-save array and variable-latency final adder.

This paper presents an integer (2's complement negative coding) pipelined multiplier architecture, which combines several algorithmic and design techniques to allow the VLSI implementation as a self-timed multiplier core or as a fully synchronous variable-latency multiplier core. The synchronous version is essentially a novel design, while the asynchronous version is a substancial evolution of [26] in the architecture (use of Booth encoding, different data-dependent carry-save array, different final adder) with additional differences in the implementation (different micropipeline scheme, full-custom core vs cell-based design). The goal of the proposed designs is worst-case speed comparable with the fastest existing multipliers, and appreciably better performance in the average case. The paper reports a comprehensive

[1] For simplicity, this paper uses the term "latency" for both synchronous and asynchronous units, meaning the time to have the result ready (either measured in cycles

instruction level analysis on the statistical effectiveness of the variable latency, in conjunction with circuit level simulations, showing the expected effectiveness of the design.

II. Background on parallel multiplier architecture

The basic add-shift multiplication algorithm of two n-bit integers A and B is expressed by the following pseudo-code:

```
for $j = 0 .. n-1$ loop
    $product <= product + A$ AND $b_j$ ;
    A <= 2*A ;
end loop ;
```

where b_j is the jth bit of B and the notation A AND bj indicates the bit vector resulting from the AND of each bit of A with the bit b_j.

An *array multiplier* can take advantage of Carry Save Adders (CSA) in order to avoid the carry propagation at each step of the add-shift loop [22]. Fig. 1 sketches the structure of a CSA array multiplier; each row of the array is composed of full adders (half-adders for the first row) with no horizontal carry propagation, while the last row at the bottom is a carry propagation adder (CPA) that resolves the carry propagation to obtain the final sum. In Fig. 1 the shift operations among consecutive CSA rows are implicit, to keep the picture simple. The CSA array, having O(n) delay, can be transformed into a radix-3 tree structure with O(log n) delay, without affecting the correctness of the final result (Wallace tree, [22]). The irregular structure of such tree can be overcome by introducing *4:2 compressors* [49], i.e. 4-input 2-output adders that allow a more regular radix-2 tree. Another possibility is to split the CSA array into a pair of arrays that operate in parallel [22], thus obtaining a regular rectangular structure mostly employing simple full adders, but only reducing the CSA delay to O(n/2).

Table 1

bj+1	bj	bj-1	operation	sj	dj	pj	mj
0	0	0	no op.	0	1	0	0
0	0	1	+ A	1	0	1	0
0	1	0	+ A	1	0	1	0
0	1	1	+ 2A	0	1	1	0
1	0	0	- 2A	0	1	0	1
1	0	1	- A	1	0	0	1
1	1	0	- A	1	0	0	1
1	1	1	no op.	0	1	0	0

or in ns). This causes no ambiguity for the scope of this paper.

Independently of the CSA structure, the number of partial products to be added together can be reduced by applying *2nd order Booth encoding* to the multiplier operand [22][4]. 2nd order Booth encoding exploits the presence of 1's or 0's in the multiplier operand to merge two additions in a single operation, as specified in Table 1. The encoding in Table 1 is applied for $j = 0, 2, 4, \ldots, n-2$, thus reducing to $n/2$ the number of partial products to be considered. Table 1 shows a specific implementation of the encoding bits, *sign select Booth encoding* [19]. Essentially, the bits p_j and m_j indicate whether a "plus" or a "minus" operation has to be performed, while s_j and d_j indicate whether to use a "single" or "doubled" value of A as the partial product. Following that implementation, the add-shift pseudo-code multiplication algorithm becomes

```
for $j$ = 0 .. $n$-2 step 2 loop
    $product$ <= $product$ + ($A$ AND $p_j$ – $A$ AND $m_j$) AND $s_j$
                        +(2*$A$ AND $p_j$ – 2*$A$ AND $m_j$) AND $d_j$ ;
    $A$ <= 4*$A$ ;
end loop ;
```

In a CSA array implementation of the above algorithm, a single set of encoding bits p_j, m_j, s_j and d_j controls a CSA row operation, except for the first row of the array which needs two sets of encoding bits because it directly processes two partial products.
Finally, pipeline registers can be inserted in the multiplier data-path to improve the throughtput.

III. Architecture and logic design of the proposed multipliers

The proposed architecture is based on sign select 2nd order Booth encoding and a split CSA array. For n-bit operands there are $n/2$ partial products, i.e. $(n/2)(1/2) - 2$ rows of full-adders in each of the two CSA arrays (Fig. 2). A 4:2 compressor is used to merge the results of the split CSA array, with the view of allowing a regular layout, though this work does not focus on layout design. Each row of full-adders is conventionally numbered with the index j of the Booth encoding bits controlling that row, thus starting with $j = 4$ for the array on the left, and $j = n/2 + 4$ for the array on the right. Such row numbering convention simplify the logic equations in the following.
A straightforward fixed-latency implementation of the scheme in Fig. 2 has the following delay components:
$Delay = D_{Booth\text{-}enc} + D_{Booth\text{-}sel} + D_{ha} + (n/4 - 2)\, D_{fa} + D_{4:2} + D_{cpa}$,
where the delay contributions are: $D_{Booth\text{-}enc}$: Booth encoding logic producing the encoding bits p_j, m_j, s_j, d_j; $D_{Booth\text{-}sel}$: logic that selects the operation of each CSA row according to the encoding bits; D_{ha} : half-adder (CSA row); D_{fa} : full-adder (CSA row), $D_{4:2}$: 4:2 compressor; D_{cpa} : final CPA. The latter is intended as a fixed-latency adder.

In order to enforce a variable (and shorter) latency in the CSA array operation, the proposed architecture includes a set of *skip* signals. Referring to one of the two CSA arrays, each $skip_{hk}$ signal indicates whether *all* of the consecutive CSA rows numbered h, h+2, h+4, … k have to perform *no operation*. The $skip_{hk}$ signals express the possibility of bypassing not only a single CSA row in case of no operation, but also multiple consecutive CSA rows.

Fig. 3 illustrates the detail of the skip operation in the CSA rows. The Booth selector chooses among the possible operations to be performed, while the $skip_{hk}$ signals force the data to bypass the CSA rows that perform no effective operation (i.e. should add a zero). The bypass operation occurs in a single multiplexer delay even for multiple row bypass, thanks to the multiple $skip_{hk}$ signals covering all the consecutive row bypass combinations.

In [26], Kearney and Bergmann have shown that bypassing a CSA row in an array multiplier implies the cost of forwarding to the final CPA an unresolved carry+sum bit pair for the least significant bit of the row, instead of a single final-product bit. In a *non-Booth-encoding* architecture, this causes the final CPA size to be 2n bits instead of n bits required by a conventional CSA array [26]. However, in a *Booth-encoding* architecture a 2n bit final CPA is anyway needed, to manage the extra carries produced by 2's complement Booth subtractions [21], so that introducing variable-latency does not increase the size of the final CPA. Fig. 3 details the mechanism used to exploit this property, which distinguishes the proposed architecture from previous "row-skipping" CSA arrays: a non-skipped row may forward the least significant carry caused by a 2's complement subtraction; a skipped row forwards the least significant carry coming from the preceding row.

The logic design of the $skip_{hk}$ signal generation is a peculiar aspect of the proposed architecture and deserves special attention. From Table 1, the expression of the signals $skip_{hk}$, $h \leq k$, is

$skip_{hk} = (\sim p_h \cdot \sim m_h) \cdot (\sim p_{h+2} \cdot \sim m_{h+2}) \cdot \ldots \cdot (\sim p_k \cdot \sim m_k)$.

where the symbol ~ indicates logic inversion.

For the operation to be efficient, it is necessary to generate the $skip_{hk}$ signals in parallel, rather than iteratively (such as $skip_{hk} = skip_{h,k-1} \cdot \sim p_k \cdot \sim m_k$). Moreover, to have the fastest generation of the $skip_{hk}$ signals and to avoid increasing the fan-out of the Booth encoders, it is favorable to obtain the $skip_{hk}$ signals directly from the bits of the operand B, rather than from the Booth encoders output p_h and m_h. The logic expression of a generic $skip_{hk}$ signal becomes (from Table 1):

$skip_{hk} =$

$(\sim b_{h+1} \cdot \sim b_h \cdot \sim b_{h-1} + b_{h+1} \cdot b_h \cdot b_{h-1}) \cdot$

$(\sim b_{h+3} \cdot \sim b_{h+2} \cdot \sim b_{h+1} + b_{h+3} \cdot b_{h+2} \cdot b_{h+1}) \cdot$

…

$(\sim b_{k+1} \cdot \sim b_k \cdot \sim b_{k-1} + b_{k+1} \cdot b_k \cdot b_{k-1})$,

which, by De Morgan transformations, can be reduced to

$\sim skip_{hk} =$

$(\sim b_{h+1} \cdot b_h + \sim b_{h+1} \cdot b_{h-1} + b_{h+1} \cdot \sim b_h + \sim b_h \cdot b_{h-1} + b_{h+1} \cdot \sim b_{h-1} + b_h \cdot \sim b_{h-1}) +$

…

$(\sim b_{k+1} \cdot b_k + \sim b_{k+1} \cdot b_{k-1} + b_{k+1} \cdot \sim b_k + \sim b_k \cdot b_{k-1} + b_{k+1} \cdot \sim b_{k-1} + b_k \cdot \sim b_{k-1})$.

This NOR-of-AND expression allows to produce the $skip_{hk}$ bits concurrently with the Booth encoding bits m_h and p_h. On the other hand, it obviously introduces redundant logic.

By some algebraic passages we can observe that the number of $skip_{hk}$ signals to detect all the combinations of consecutive rows among x rows is $x(x+1)/2$. For some CSA array sizes it could be impractical to generate the complete set of $skip_{hk}$ signals, because of the interconnection complexity (see Fig. 3). In a 32×32 bit implementation, a full-bypass architecture would need 26 $skip_{hk}$ signals and 8-input multiplexers on the final CSA row. Such complexity is likely to be useless because the first CSA row consists of relatively fast half-adders, and because a full skip of both CSA arrays is reasonably very rare ($0.25^{14} = 4{\cdot}10^{-9}$ probability, assuming uniform operand distribution). One possibility is to partition a CSA array into blocks of limited size that represent the maximum number of simultaneously bypassable CSA rows. In the proposed 32×32 bit implementation, both 7-row arrays are divided into two blocks of 3 bypassable rows each, plus a non-bypassable row of half-adders (first-row). Actually, this choice limits to 6 $skip_{hk}$ signals the complexity of each bypassable block; alternatively, three 2-row blocks would vanish most of the potential of multiple row skipping, while a 4-row block followed by a 2-row block would result in a very unbalanced block complexity.

For the final CPA adder, the proposed architecture employs a Carry Select (CS) scheme [5], whose data dependent completion time has been investigated and proven in previous works [15][29][30]. Specifically, the size of the CS groups in the final 64-bit CPA follows the scheme 8,8,8,8,8,6,6,5,4,3 bits, considering the most significant digit on the left. With this sizing choice, after a delay equivalent to 7 full-adders the output carries of the 3-, 4-, 5- and 6-bit groups are surely valid, while the 8-bit groups *may* have their output carry valid (*anticipated carry* [15]). In that case, an early CPA completion occurs. In the worst case, an additional delay of 5 multiplexers is needed to complete the carry selection chain. The CPA sizing choice aims at having a similar latency variation (in nanoseconds) in the CPA and in the CSA array.

The resulting multiplier latency is a generic expression of the form

$Delay = D_{Booth\text{-}enc} + D_{Booth\text{-}sel} + D_{ha} + N(B)\cdot(D_{fa} + D_{mux}) + b\cdot D_{mux} + D_{4:2} + D_{cpa}(A,B)$,

where D_{mux} is the delay of a multiplexer, b is the number of bypassable blocks of rows in a CSA array, $N(B)$ is the actual number of partial products added/subtracted by the two CSA arrays (actually the maximum between the two). The notations $N(B)$, $D_{cpa}(A,B)$ indicate dependency on the operand values. In particular, the formula assumes the overlapped $skip_{hk}$ signal computation, i.e. $D_{skiplogic} < D_{Booth\text{-}enc} + D_{Booth\text{-}sel} + D_{ha}$, where $D_{skiplogic}$ is the time to generate the $skip_{hk}$ signals.

The functionality of the proposed architecture scheme has been validated through a bit level executable algorithmic model [24].

IV. VLSI DESIGN OF SYNCHRONOUS VARIABLE-LATENCY IMPLEMENTATION

Fig. 4 shows the block diagram of the synchronous 32-bit implementation of the variable latency architecture. Each of the two CSA arrays includes two 3-row bypassable blocks. Six $skip_{hk}$ signals operate on each bypassable block.

The multiplier is pipelined by a register at the output of the split CSA array. Considering the worst case delay, the approximate critical paths of the two stages are:

$Delay\_1^{st}|_{worst} = D_{Booth\text{-}enc} + D_{Booth\text{-}sel} + D_{ha} + 6\ D_{fa} + 6\ D_{mux}$,

$Delay\_2^{nd}|_{worst} = D_{mux} + D_{4:2} + D_{cpa} \approx D_{mux} + D_{4:2} + 7\ D_{fa} + 5\ D_{mux}$;

where the carry propagation and the sum generation in a full-adder are both indicated as D_{fa} for simplicity.

The multiplier can complete in either one or two clock cycles, depending on the actual operand values. To complete the operation in one cycle, a multiplexer allows the partial results of the CSA array to override the pipeline register in case there is a chance to traverse the whole multiplier logic path in a single cycle.

To implement single-cycle/two-cycle completion prediction, we have to establish a set of conditions leading to a total latency roughly equivalent to half the worst case total latency, and then size the VLSI cycle time to fit such reduced latency. In the proposed implementation, the conditions for having the result ready after one cycle are that each of the CSA arrays processes at most two partial products, *and* the second pipe stage is not busy (processing a previous 2-cycle multiplication), *and* all of the 8-bit CS groups in the final CPA anticipate their carries. When all those conditions are met, the actual critical path of *both* stages together is nominally *Delay_singlecycle* = $D_{Booth\text{-}enc} + D_{Booth\text{-}sel} + D_{ha} + 2\,(D_{fa} + D_{mux}) + 2\,D_{mux} + D_{mux} + D_{4:2} + 7\,D_{fa} + D_{mux}$. The conditions are defined with the aim of a relatively easy logic detection of them; the VLSI implementation of the multiplier must take care that the single-cycle-operation delay is roughly comparable with the worst case delays of the two single pipe stages ($Delay\_1^{st}|_{worst}$ and $Delay\_2^{nd}|_{worst}$). The VLSI cycle time will be sized accordingly, as shown later.

Functionally, the signal *busy* detects if the second pipe stage is busy, the signals *unresolved-carry*$_i$ detect if the CPA cannot anticipate its internal carries, the signals *no-skip*$_i$ detect CSA rows that cannot be skipped, and the *onecycle* signals flags if the operation is completed in the current cycle. The “>2” blocks check if there are more than two partial products processed in either CSA arrays, while the “>0” block (OR operator) detects if there is at least one 8-bit CS groups in the final adder with unresolved carry.

Supposing the operands are presented at the multiplier inputs at the i^{th} clock falling edge, in case the result is ready in one cycle the signal *onecycle* is high at the i+1th clock falling edge; otherwise the *onecycle* signal is low, and the result is ready at the i+2th rising clock edge. It is important that the *onecycle* signal is synchronous with the clock, i.e. it is stable within the current clock cycle, and is active only in case of early completion of the multiplication.

Fig. 5 shows the full-custom VLSI design of the “skip logic” for a 3-row block. All the 3-row blocks in Fig. 4 and the corresponding “skip logic” are identical to this. Fig. 6 shows the VLSI design of the final logic producing the *onecycle* signal. Fig. 7 details the VLSI design of a carry select block in the carry-completion-sensing CPA. All the logic in the multiplier is designed in dynamic “domino” CMOS style, with inverters between cascaded logic blocks [50]. Transistor sizing is accomplished according to the “logical effort” optimization methodology [42], though with simplifying assumptions concerning circuit branching. In accordance with the logical effort method, the number of CMOS stages in the critical path of the main circuit blocks has been optimized. In particular, in the “skip logic” circuits (Fig. 5), there are 2 static inverting buffers on each input bit (not shown), and the NOR-of-AND function is split into 2 domino stages. In Fig. 5, the critical path is from the $b_3 \ldots b_9$ input bit to the *skip*$_{48}$ output signal. In Fig. 6, the critical inputs of the logic circuit are the *unresolved*$_j$ bits, because they come from the carry select blocks in the final CPA and therefore are later than the *no-skip*$_h$ signals, which directly come from the skip logic (Fig. 4). The critical delay condition for the signal *onecycle* occurs when only 2 *no-skip*$_h$ are active (i.e. the CSA array processes 2 partial products), but there is some *unresolved*$_j$ active. The resulting critical delay of the whole logic that produces the signal *onecycle* is approximately $D_{onecycle} = D_{Booth\text{-}enc} + D_{Booth\text{-}sel} + D_{ha} + 2\,(D_{fa} + D_{mux}) + 2$

$D_{mux} + D_{mux} + D_{4:2} + 7\, D_{fa} + D_{detect}$, where D_{detect} is the delay of the circuit in Fig. 6 from the $unresolved_j$ bits to the *onecycle* output signal.

The signals in the circuits are pre-charged high during the low clock-half-cycle, except for the *busy* signal which is ready immediately after the falling edge of the clock because it is produced by the falling-edge-triggered register.

The following timing parameters, augmented by the register and precharge overhead, define the cycle time sustainable by the multiplier: the worst-case $Delay\_1^{st}\,|_{worst}$ and $Delay\_2^{nd}\,|_{worst}$, the delay in case of single cycle operation *Delay_singlecycle*, and the prediction delay $D_{onecycle}$.

V. VLSI DESIGN OF SELF-TIMED VARIABLE-LATENCY IMPLEMENTATION

The scheme of the variable-latency asynchronous multiplier is based on the micropipeline paradigm [41]. The multiplier is designed as a two-stage micropipeline interfaced with the external environment through a pair of request/acknowledge 2-phase signals at the input and another pair at the output of the multiplier. The scheme uses double edge triggered memory elements, realized according to [51], which reduce the micropipeline interconnections and switching activity.

Fig. 8 shows the micropipeline implementation. The essential feature of the design is that the delay of each of the two micropipeline stages is data dependent, thanks to the $skip_{hk}$ signals in the CSA and the completion-sensing CPA. To this end specialized circuits are responsible for producing the completion signals of the CSA arrays and of the CPA. For the CSA array, a dummy logic path controlled by the $skip_{hk}$ signals reproduces the data-dependent logic levels of the two CSA arrays. The signals $Z1$ and $Z2$ switch nominally at the same time as the corresponding CSA array output are ready. This time depends on the actual data values, through the $skip_{hk}$ signals. For the final CPA, the completion signal is generated by a speculative completion technique [52], i.e. by selecting one of two delays for the completion signal: a shorter delay in case no $unresolved_i$ signal is active, a longer delay in the opposite case. Double edge-triggered monostable elements provide the precharge signal for the dynamic logic gates of the multiplier. In both pipe stages a programmable delay element is inserted, to compensate for the pre-charge time and to adjust possible derating factors in the control signal delay, as already shown in [14] and [10].

In the micropipelined implementation, it is neither necessary to provide the logic to override the pipeline register, nor to check if the second stage is busy when the first stage's result is ready, because the register control signal is governed by the data-dependent hand-shaking of the two pipe stages. So the elastic behavior of the micropipeline automatically implements the variable response time of the multiplier.

All the data-path components have the same circuit implementation as in the synchronous version, except for the double-edge-triggered pipeline register.

The average delays of the two variable-latency stages are nominally expressed by

$Delay\_1^{st}|_{avg}$ =

$D_{Booth\text{-}enc} + D_{Booth\text{-}sel} + D_{ha} + app\cdot (D_{fa} + D_{mux}) +2\, D_{mux}$,

$Delay\_2^{nd}|_{avg}$ =

$D_{4:2} + \mathrm{E}\{D_{cpa}\} \approx D_{4:2} + 7\, D_{fa} + pncp\cdot D_{mux} + (1 - pncp)\cdot 5\, D_{mux}$;

where *app* is the average number of non-zero partial products to be added/subtracted by the split CSA array and *pncp* is the probability that the final CPA needs no carry propagation, because all of the 8-bit CS groups anticipate their output carry.

The timing parameters that define the performance of the micropipeline multiplier implementation are the worst and best delay of the stages, and their average value $Delay\_1^{st}|_{avg}$ and $Delay\_2^{nd}|_{avg}$, to obtain the multiplier throughput and latency.

VI. Statistical instruction level relevance of variable latency

The design of a variable latency unit should be supported by evidence that its average operation latency in a real application is shorter than the worst-case latency. As Booth encoding affects the distribution of 0’s and 1’s in the operands, we need a direct analysis of the bit-level behavior of the specific architecture executing real operations. Here I present a study based on simulating a MIPS-like instruction set architecture, through an instrumented version of the Simplescalar tool set [8], considering two multiplication instructions: MULT (32-bit integer multiply) and MULTU (unsigned 32-bit integer multiply). The MULTU instruction is less significant as it occurs either rarely or never at all. Operand traces where extracted and used with a bit-level C-language model of the variable-latency architecture [24], obtaining the statistical parameters in Tables 2 and 3. Table 2 refers to a sequence of multiplications with theoretical standard distributions of the operands. More significantly, Table 3 refers to the execution of the SPEC95 benchmark suite with reference input files [23]. Tables 2 and 3 report the parameters *app* , *pncp* and the parameter 1*cyc*, i.e. the probability of meeting the conditions for single-cycle operation in the synchronous implementation.

Table 2

instruction ->	MULT			MULTU		
parameter ->	app	1cyc	pncp	app	1cyc	pncp
Uniform ops	4.51	0.01	0.40	4.50	0.01	0.39
Gauss ops	2.90	0.02	0.03	2.99	0.01	0.03
Poisson ops	3.89	0.02	0.09	3.90	0.02	0.09

Table 3

instruction ->	MULT			MULTU		
parameter ->	app	1cyc	pncp	app	1cyc	pncp
applu	1.65	0.17	0.17	1.81	0.05	0.06
compress	0.06	0.95	0.95	--	--	--
gcc	1.48	0.45	0.79	--	--	--
go	0.07	0.97	0.97	--	--	--
ijpeg	3.79	0.00	0.00	--	--	--
li	0.09	0.92	0.92	--	--	--
m88ksim	0.03	0.97	0.96	6.00	0.00	0.00
mgrid	1.11	0.00	0.00	1.86	0.23	0.35
swim	0.32	0.88	0.88	1.95	0.14	0.17
tomcatv	0.20	0.80	0.80	--	--	--
turb3d	1.20	0.00	0.00	2.67	0.24	0.40
vortex	1.55	0.30	0.30	5.99	0.00	0.35
wave5	0.78	0.65	0.65	--	--	--
AVERAGE	1.01	0.54	0.57	0.38	0.11	0.22

VII. PERFORMANCE RESULTS

Fig. 9 shows a sample of the HSpice level 49 simulation of the synchronous multiplier referring to a 0.35 μm CMOS process, assuming typical operating conditions. The mask layout of the circuits has not been fully designed; however, long metal line parasitic capacitances were manually coded into Spice netlists, based on a hypothetical layout of the circuits. In all the circuits, no serious charge sharing problems occur, probably because the capacitance of the internal nodes that may cause charge redistribution effects are actually small, compared to the load capacitance and the parasitic capacitance of the output nodes. In Fig. 9, from left to right, first a two-cycle multiplication case and then a single-cycle case are shown. Data values are sampled by the registers on the falling edge of the clock. Register delay is overlapped with the subsequent precharge phase of the dynamic combinational circuits.

Table 4 shows the delay characterization obtained for the relevant circuit modules of the multiplier; in the asynchronous version, pre-charging starts after the precharge control signal has traversed a proper buffer

tree in order to drive all the dynamic circuits in the pipe stage; in the synchronous version, clock buffering delay is overlapped with circuit operation, and pre-charging starts right at the falling edge of the clock signal. For this reason, in the self-timed implementation the actual time taken by pre-charging is longer.
Power dissipation has not been thoroughly investigated; as in any parallel multiplier, the switching activity is strongly dependent on the sequence of input operands. A rough estimate based on the transistor overhead may lead to assume less than 20% power dissipation overhead with respect to a fixed-latency array multiplier architecture.

A. *Synchronous implementation*

The cycle time limit for the synchronous implementation results to be the completion prediction delay $D_{onecycle}$ of 1.84 ns. Considering the the hold time for the precharge signal, the expected cycle time sustainable by the multiplier is nominally 2.25 ns (with asymmetric clock shape). The area overhead can be estimated in terms of transistor count with respect to the implementation of the same synchronous multiplier architecture with fixed latency: the total transistor count of the proposed implementation is 22609, and the transistor count of all the logic dedicated to variable latency is 3329. The overhead is approximately 17%. The delay overhead due to the variable latency management is essentialy due to the multiplexers inserted in the CSA arrays. Referring to Fig. 4, circuit simulation shows that the sum of the delays of the pass-transistor multiplexers causes a 0.32 ns overhead in the first (and slowest) pipeline stage. Thus a fixed-latency implementation of the proposed architecture is expected to sustain 1.75 ns cycle time and 3.50 ns *fixed* latency.

Table 4

synchronous	[ns]	**self-timed**	[ns]
$Delay1^{st}\|_{worst}$	1.65	$Delay1^{st}\|_{worst}$	1.65
$Delay2^{nd}\|_{worst}$	1.23	$Delay2^{nd}\|_{worst}$	1.20
pipe register	0.29	*pipe register*	0.32
precharge hold time	0.40	*precharge* delay	0.78
$D_{skiplogic}$	0.55	$D_{skiplogic}$	0.55
D_{Booth_enc}	0.34	D_{Booth_enc}	0.34
D_{Booth_sel}	0.19	D_{Booth_sel}	0.19
D_{ha} *in 1st CSA row*	0.08	D_{ha} *in 1st CSA row*	0.08
Delay_singlecycle	1.83	$Delay1^{st}\|_{best}$	0.76
$D_{onecycle}$	1.84	$Delay2^{nd}\|_{best}$	0.81
		D_{fa} *in CSA row*	0.11
		D_{mux} *in CPA*	0.08

Table 5 – (n.a. = not applicable; N.A. = not available)

type	SA	PPC	PIII	NEC	M·CORE (best lat.)	M·CORE (worst lat.)	this work (best lat.)	this work (worst lat.)
variable-latency statistics (SPEC95 multiplications)	n.a.	n.a.	n.a.	n.a.	N.A.	N.A.	54%	46%
latency [cycles]	2	3	4	1	2	18	1	2
latency [ns]	9.60	6.44	6.67	2.73	60.00	540.00	2.25	4.50
throughput [ns]	4.80	2.14	1.67	2.73	60.00	540.00	2.25	2.25
feature size [μm]	0.35	0.22	0.18	0.25	0.36		0.35	

Table 6 – (source for performance of compared designs: [27])

type	BFB	SK	KB	this work
word size [bits]	16×16	16×16	16×16	32×32
latency [ns]	25.00	64.00	15.00	3.48
throughput [ns]	5.51	6.40	10.80	1.76
supply voltage [V]	5.00	5.00	5.00	3.30
feature size [μm]	1.20	1.00	0.60	0.35

Fig. 10 shows a comparison with existing state-of-the-art synchronous 32-bit multipliers. The labels "M·CORE", "PIII", "SA" and "PPC" refer to 33MHz M·CORE MMC2001 (incorporating a variable latency multiplier), 600MHz PentiumIII, 206MHz StrongARM 1110 and 466 MHz PowerPC 750, respectively. The label "NEC" refers to NEC's ultra-fast 2.7 ns multiplier architecture reported in [21], here estimated for 32-bit design instead of 54-bit. The original delay contribution reported in [21] are 0.53 ns Booth encoders/selectors, 0.95 ns 4:2 compressor tree, and 1.22 ns 108-bit CarryLookahead CPA. Scaling the compressor tree and the final CPA for 32-bit multiplier operands leads to an estimated 2.33 ns total latency and 2.73 ns cycle time including pre-charge time.

Considering the 54% statistical probability of single-cycle operations in SPEC95 execution (Table 3), the average expected latency of the proposed multiplier remains slightly longer than NEC's design, due to more advanced circuit optimization and technology. As a whole, the expected-performance figures of the synchronous variable-latency design are surely remarkable.

B. Self-timed implementation

From Table 4, the worst and best case latencies of the self-timed implementation result to be 4.41 ns and 3.13 ns, respectively, while the average delays of the two micropipeline stages referring to SPEC95 execution are

$Delay\_1st|_{avg} = 0.76\text{ ns} + 1.01(D_{fa} + D_{mux}) = 0.94\text{ ns}$,

$Delay\_2^{nd}|_{avg} = 0.76\text{ ns} + 0.57\, D_{mux} + 0.43 \cdot 5\, D_{mux} = 0.98\text{ ns}.$

The resulting approximate average latency of the whole micropipeline is 0.78 + 0.94 + 0.78 + 0.98 = 3.48 ns and the average throughput is dictated by the delay of the second stage, 0.98 + 0.78 = 1.76 ns. The architecture is organized to overlap the pre-charge phase of the second stage with the propagation delay through the micropipeline register.

The variable-latency complexity overhead in the self-timed implementation is similar to the synchronous case, as the essential logic blocks are the same.

Fig. 11 shows a comparison with published asynchronous multipliers performance. The compared designs are all 16-bit multipliers. The labels “BFB” and “SK” respectively refer to the fixed-latency self-timed designs reported in [7] and [38], while the label “KB” refers to the variable-latency design reported in [26]. Technology parameters for all the designs are reported.

VIII. Conclusions

The proposed architecture and VLSI design demonstrates that a variable latency multiplier, in either synchronous or asynchronous implementations, can overcome the performance offered by fast fixed-latency multipliers.

The presented design address the use of edge-triggered registers, to reduce the complexity of the register-overriding logic design in the synchronous implementation and of the interconnection routing in the self-timed implementation. Investigation of the performance impact of using latches may be an area of further work. Similarly, further work may address the performance trade-offs of the number of pipe stages in the self-timed micropipelined implementation.

IX. Acknowledgements

The author is grateful to Prof. David Kearney for his valuable suggestions and to Prof. Alessandro De Gloria for his support.

X. References

[1] Acosta A. J., Jiménez R., Barriga A., Bellido M. J., Valencia M., and Huertas J. L., Design and characterisation of a CMOS VLSI self timed multiplier architecture based on a bit level pipelined array structure. IEE Proceedings, Circuits, Devices and Systems, 145(4):247 253, August 1998.

[2] Afgahi M. and Svensson C., "Performance of Synchronous and Asynchronous Design Scheme for VLSI Systems", IEEE Trans. on Computers, 41(7):838 872, July 1992.

[3] Alvarez J., Barkin E., Chai Chin Chao., Johnson B., D'Addeo M., Lassandro F., Nicoletta G., Patel P., Reed P., Reid D., Sanchez H., Siegel J., Snyder M., Sullivan S., Taylor S. and Minh Vo, 450 MHz PowerPCTM microprocessor with enhanced instruction set and copper interconnect , IEEE International Solid State Circuits Conference, Piscataway, NJ, USA., 1999., p.96 7.

[4] Baer J. L., Computer systems architecture, Computer Science Press, Rockville, Maryland, 1980, pp. 100 106.

[5] Bedrij, O.J., "Carry Select Adder", IRE Trans. on Electronic Comp., 11:340 346, 1962.

[6] Briley, B. "Some new results on average worst case carry", IEEE Trans. on Computers,C 22:459 463, 1973.

[7] Burford R. G., Fan X., and Bergmann N. W., An 180 MHz 16 bit multiplier using asynchronous logic design techniques. In Proc. IEEE Custom Integrated Circuits Conference, pages 215 218, 1994.

[8] Burger, D. and Austin, T. M. , The Simplescalar Tool Set, Version 2.0, Tech. Rep. #1342, Univ. of Wisconsin Madison, June 1997. Available at http://www. cs.wisc.edu/~mscalar/simplescalar.html

[9] Burks, A.W. and Goldstine, H. and Von Neumann, J. , "Preliminary discussion on the logical design of an electronic computing instrument", Tech. Report, The Institute of Advanced Study", Princeton, NJ, 1947.

[10] Chandramouli V., Brunvand E., Smith KF, Self timed design in GaAs case study of a high speed, parallel multiplier, IEEE Transactions on VLSI Systems.vol.4, no.1., March 1996., p.146 9.

[11] Chiang Jen Shiun and Liao Jun Yao. A novel asynchronous control unit and the application to a pipelined multiplier. In Proc. International Symposium on Circuits and Systems, volume 2, pages 169 172, June 1998.

[12] Christensen K. T., Jensen P., Korger P., and Sparsø J., The design of an asynchronous Tiny RISC TR4101 microprocessor core. In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 108 119, 1998.

[13] De Angel E., Swartzlander E. Jr. and Abraham J., A new asynchronous multiplier using enable/disable CMOS differential logic. In Proc. International Conf. Computer Design (ICCD). IEEE Computer Society Press, October 1994.

[14] De Gloria A. and Olivieri M., "Statistical Carry Lookahead Adders", IEEE Trans. on Comp., 45(3):340 347, Mar. 1996.

[15] De Gloria A. and Olivieri M., "Completion-detecting Carry Select Addition", IEE Proc. – Comput. and Dig. Techn. , 147(2):93-100, Mar. 2000.

[16] Furber S.B., Garside J. D., and Gilbert D.A., AMULET3: A high performance self timed ARM microprocessor. In Proc. International Conf. Computer Design (ICCD), October 1998.

[17] Furber S.B., Garside J.D., Riocreux P., Temple S., Day P., Liu J., Paver N.C., AMULET2e: An asynchronous embedded controller. Proc. of the IEEE, 87(2):243 256, Feb. 99.

[18] Goodman R. M. and McAuley A. J., An efficient asynchronous multiplier. In K. Bromley, S. Y. Kung, and E. Swartzlander, editors, Proceedings of the Second International Conference on Systolic Arrays, pages 593 599. IEEE Computer Society Press, May 1988.

[19] Goto g., Inoue A., Kashiwakura S., Mitarai S, Tsuru T. and Izawa T., A 4.1 ns 54x54 b Multiplier utilizing sign select Booth Encoders, IEEE Jour. of Solid State Circuits, 32(11):1676 1681, Nov. 1997.

[20] Haans J., van Berkel K., Peeters A., Schalij F, Asynchronous multipliers as combinational handshake circuits IFIP Transactions A (Computer Science and Technology).vol.A 28., 1993., p.149 63.

[21] Hagihara Y., Inui S., Yoshikawa A., Nakazato S., Iriki S., Ikeda R., Shibue Y., Inaba T., Kagamihara M., Yamashina M, A 2.7 ns 0.25 mu m CMOS 54*54 b multiplier, IEEE International Solid State Circuits Conference. IEEE, New York, NY, USA., 1998., p.296-7

[22] Hennessy, J.L., Patterson, D.A., Computer Architecture: A Quantitative Approach, Morgan Kaufmann, Palo Alto, CA, 1990.

[23] Official SPEC web site: http://www.spec.org

[24] Model source code at ftp://ss1.ing.uniroma1.it/pub/reports/mul*.c

[25] Johnson, D. and Akella, V., Design and Analysis of Asynchronous Adders, IEE Proceedings Comp. and Dig. Tech., 145(1), 1998.

[26] Kearney D., Bergmann NW, Bundled data asynchronous multipliers with data dependent computation times, Intern. Symp. on Advanced Research in Asynchronous Circuits and Systems. IEEE Comput. Soc. Press, Los Alamitos, CA, USA., 1997., p.186 97.

[27] Kearney, K. and Bergmann, N., 1997. "VLSI Design of an Asynchronous Multiplier with Data Dependent Processing Times". Proc. 14th Australian Microelectonics Conference. IEEE. pp. 282-287.

[28] Kessler R. E., The Alpha 21264 Microprocessor, IEEE Micro, 19(2), pp. 24 36, March /Apr 1999.

[29] Kinnement, D. J. , An Evaluation of Asynchronous Addition, IEEE Trans. on VLSI Systems, 4(1):137 140, Mar 1996.

[30] Kondo, Y. , Ikumi N., Ueno K., Mori J. and Hirano M., An Early Completion Detecting ALU for a 1GHz 64b Datapath, IEEE Intern. Solid State Circuits Conference. IEEE, New York, NY, USA., 1997.

[31] Lau CH., Renshaw D., Mavor J, A self timed wavefront array multiplier, IEEE International Symposium on Circuits and Systems. IEEE, New York, NY, USA., 1989., p.138 41 vol.1.

[32] Lee J. and Asada K., “A synchronous completion prediction adder”, IEICE Transactions on Fundamentals of Electronics, Comm. and Comp. Sci., vol. E80-A, no. 3, pp. 606-609, March 1997.

[33] Marc R., El Hassan Bachar, A low power, 100 MHz 12*18+30 b multiplier accumulator operating in asynchronous and synchronous modes, ESSCIRC '94.Twentieth European Solid State Circuits Conference. Editions Frontieres, Gif sur Yvette, France., 1994.

[34] Martin A. J., Lines A., Manohar R., Nystroem M., Penzes P., Southworth R., and Cummings U., The design of an asynchronous MIPS R3000 microprocessor. In Advanced Research in VLSI, pages 164 181, Sept. 1997.

[35] Motorola Inc., M•CORE™ MMC2001 reference Manual, 1998., available at http://www.motorola.com

[36] Noaks DR., Burton DP, A high speed, asynchronous, digital multiplier, Radio and Electronic Engineer, vol.36, no.6., Dec. 1968., p.357 66.

[37] Rao VM., Nowrouzian B, Design and implementation of asynchronous parallel multiply accumulate arithmetic architectures, 38th Midwest Symposium on Circuits and Systems. IEEE, New York, NY, USA., 1996., p.761 4 vol.2.

[38] Salomon O., Klar H, Self timed fully pipelined multipliers, IFIP Transactions A (Computer Science and Technology), vol.A 28., 1993., p.45 55.

[39] Seitz C.L., “System Timing”, in C.Mead and L. Conway, Introduction to VLSI Systems Design, pp. 218 262. Addison Wesley, 1980.

[40] Sparsø J., Nielsen C. D., Nielsen L. S., and Staunstrup J., Design of self timed multipliers: A comparison. In S. Furber and M. Edwards, editors, Asynchronous Design Methodologies, volume A 28, IFIP Trans., pages 165-179. Elsevier Science Publishers, 1993.

[41] Sutherland I.E., Micropipelines. Communications of the ACM, 32(6):720 738, June 1989.

[42] Sutherland, I., Sproull, B., Harris, D., *Logical Effort: Designing fast CMOS circuits*, Morgan Kaufmann, San Francisco, CA, 1999.

[43] Takamura A., Kuwako M., Imai M., Fujii T., Ozawa M., Fukasaku I., Ueno Y., and Nanya T., TITAC 2: An asynchronous 32 bit microprocessor based on scalable delay insensitive model. Intern. Conference on Computer Design (ICCD), pp. 288-294, Oct. 1997.

[44] Terada H., Miyata S., and Iwata M., DDMP's: Self timed super pipelined data driven multimedia processors. Proceedings of the IEEE, 87(2):282 296, Feb. 1999.

[45] Tang, T.Y., Choy, C.S., Siu, P.L., and Chan, C.F., "Design of Self-timed Asynchronous Booth’s multiplier", Proc. of Asia & South Pacific Design Automation Conference, Jan. 2000, pp. 15-16.

[46] Tosic M. B., Stojcev M. K., Maksimovic D. M., and Djordjevic G. L., The asynchronous counterflow pipeline bit serial multiplier, Journal of Systems Architecture, 44(12):985 1004, Dec. 1998.

[47] Van Berkel K., Burgess R., Kessels J., Peeters A., Roncken M., and Schalij F., A fully asynchronous low power error corrector for the DCC player. IEEE Journal of Solid State Circuits, 29(12):1429 1439, December 1994.

[48] Van Gageldonk H., Baumann D., Van Berkel K., Gloor D., Peeters A., and Stegmann G., An asynchronous low power 80c51 microcontroller. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 96 107, 1998.

[49] Weinberger A., 4-2 carry-save adder module, IBM-Technical-Disclosure-Bulletin, vol.23, no.8; Jan. 1981; p. 3811-14.

[50] Weste, N.H. and Eshraghian, K., Principles of CMOS VLSI Design, Addison Wesley, 1993.

[51] Yun K. Y., Beerel P. A., and Arceo J., High-Performance Two-Phase Micropipeline Building Blocks: Double Edge-Triggered Latches and Burst-Mode Select and Toggle Circuits , IEE Proceedings-Circuits, Devices and Systems. pp 282-288. Vol. 143, No. 5, October 1996.

[52] Yun Y. K., Nowick S.M, Beerel P. A., Dooply A. E., Speculative Completion for the Design of High Performance Asynchronous Dynamic Adders, Proceedings of ASYNC 97, Eindhoven, The Netherlands, 1997.

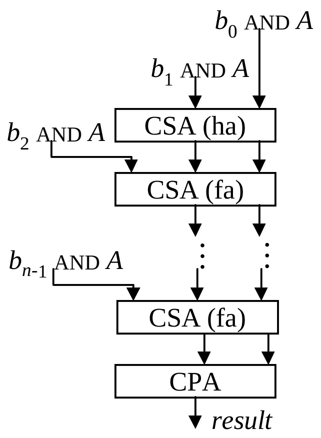

Fig. 1 – Basic scheme of array multiplier. ha= Half Adders, fa=Full Adders.

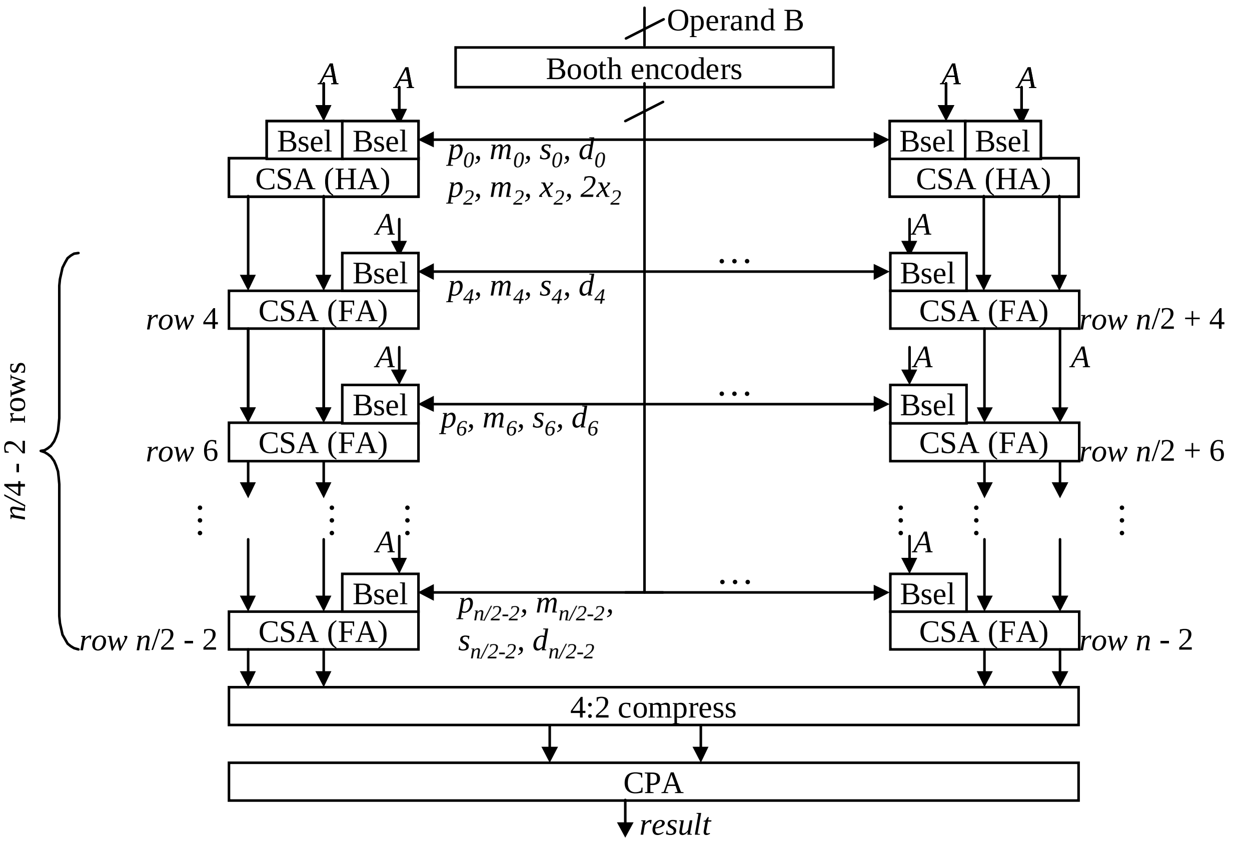

Fig. 2 – Architecture scheme of split-array multiplier with Booth encoding. Bsel = Booth selector, HA= Half Adders, FA=Full Adders.

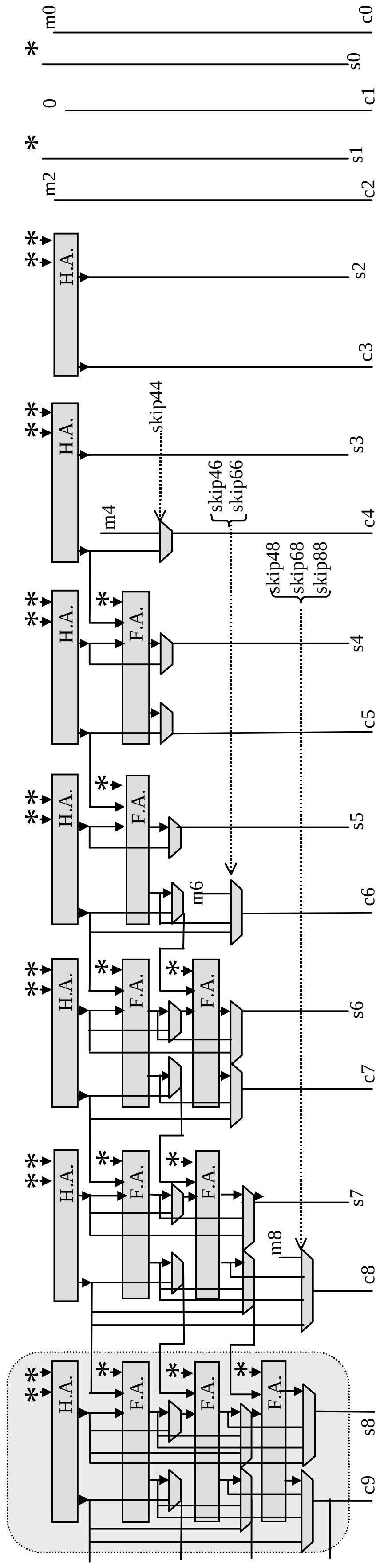

Fig. 3 – Detail of the row bypass architecture. Each row of multiplexers is driven a single set of skip signals. The symbol * indicates a partial-product-bit generator. The highlighted part at the bottom is detailed in Fig. 5 at transistor level.

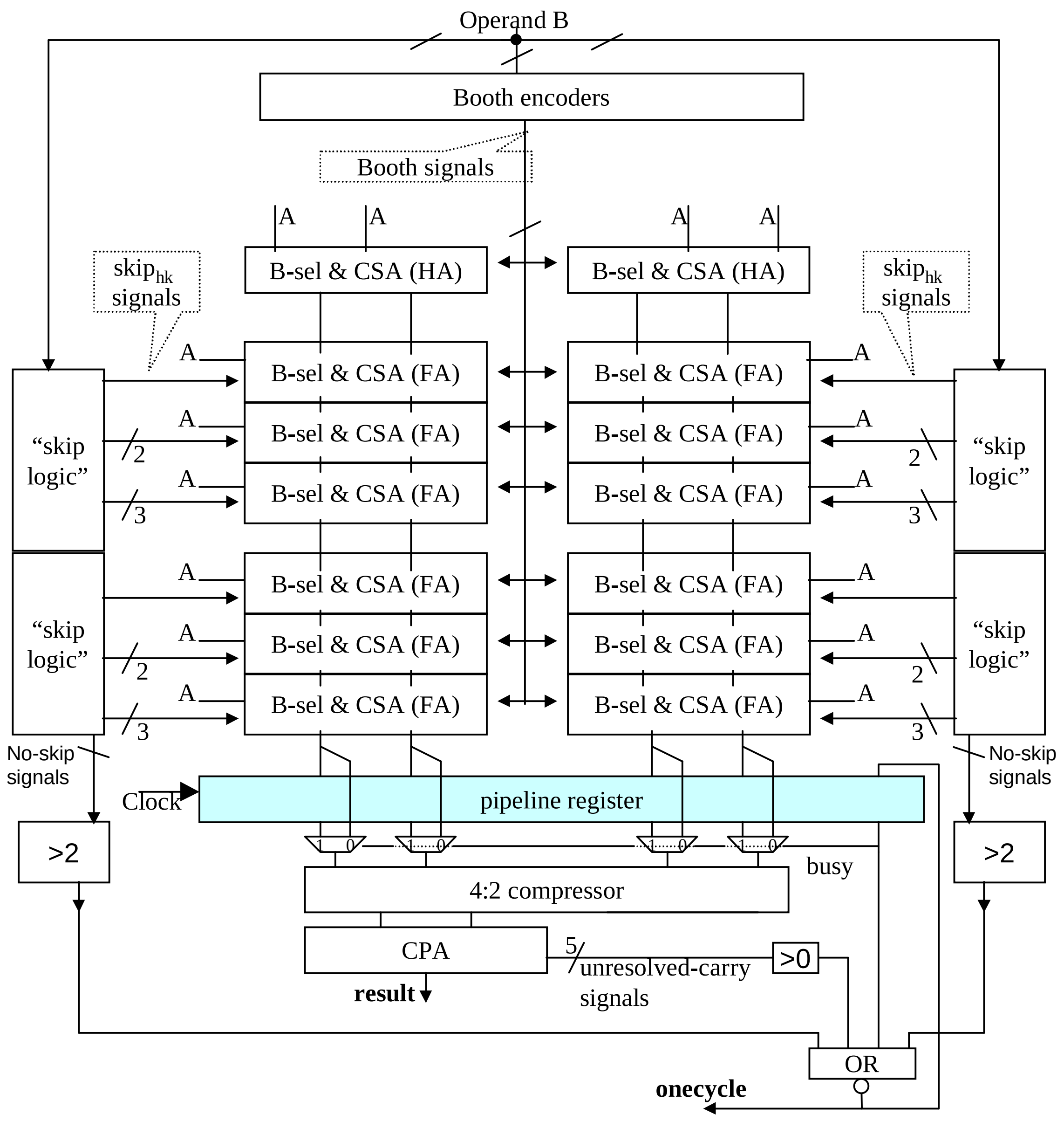

Fig. 4 – 32-bit synchronous implementation of the variable latency multiplier

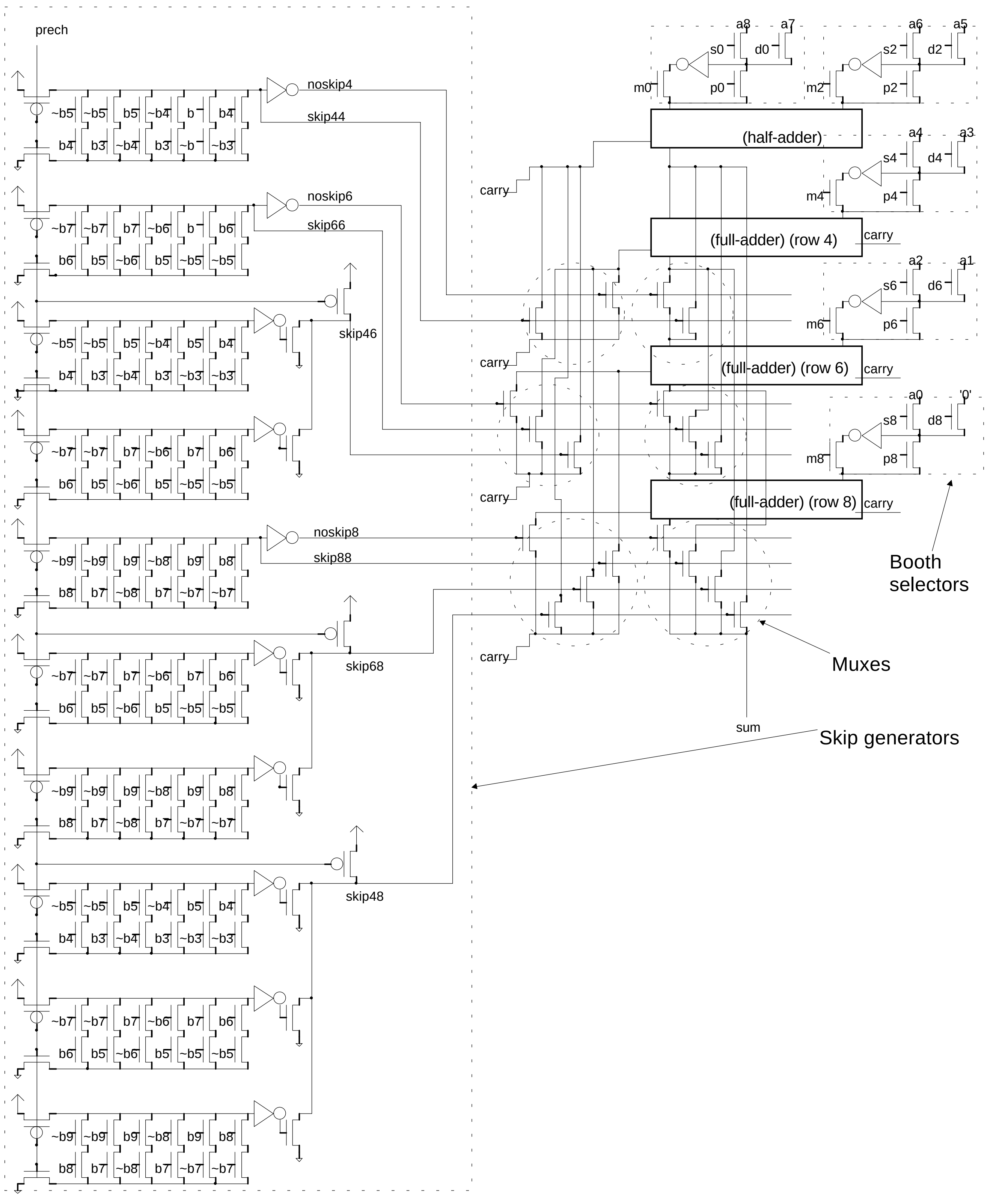

Fig. 5 - Full-custom VLSI design of the "skip logic" for a 3-row block in the CSA array

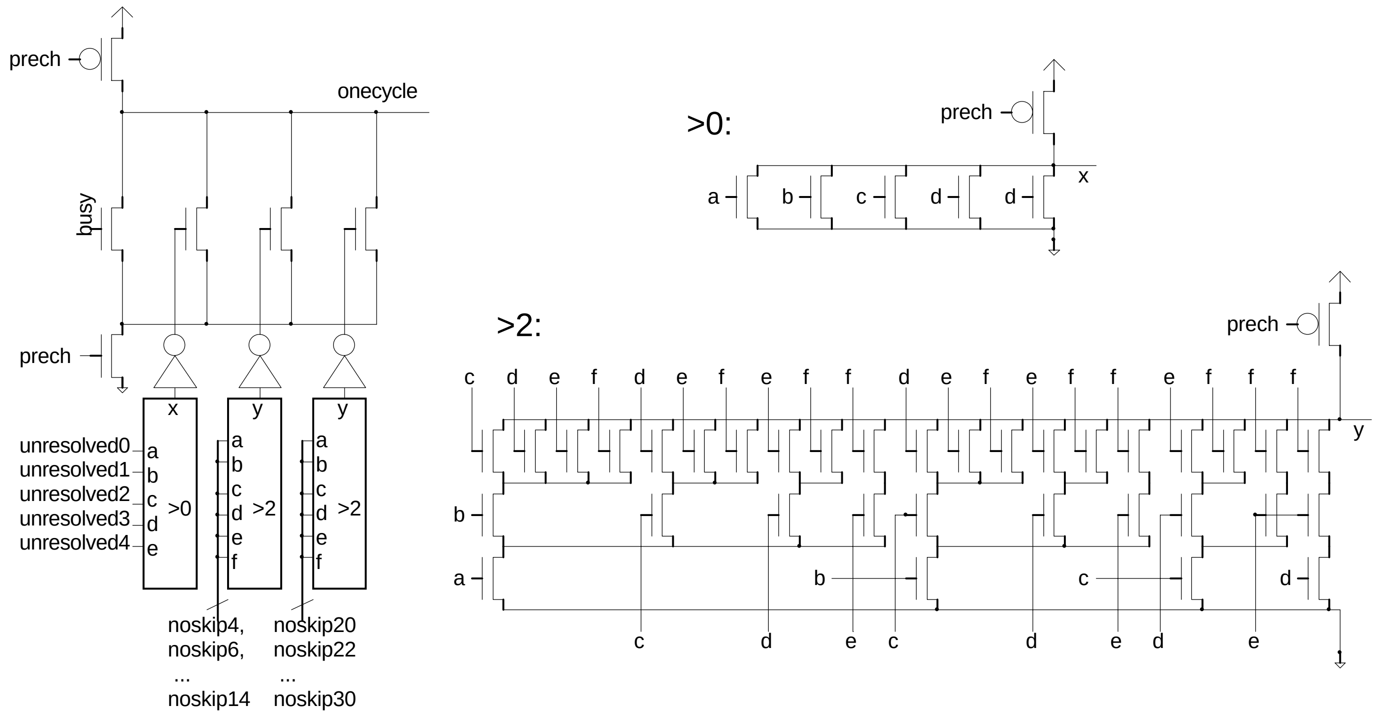

Fig. 6 - VLSI design of the logic producing the "onecycle" signal. At the top-right, the '>0' subcircuit is detailed; at the bottom-right, the '>2' subcircuit is detailed

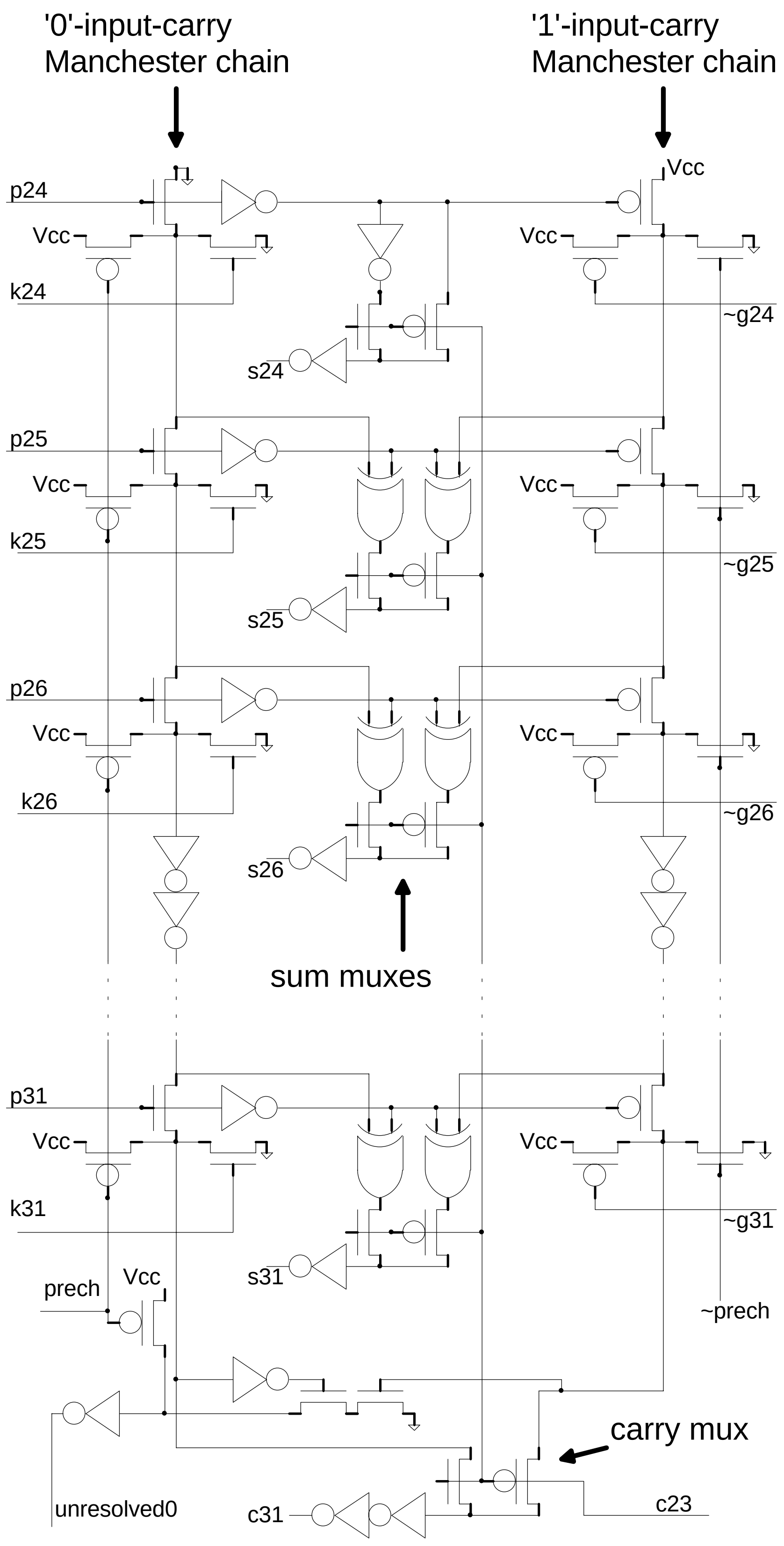

Fig. 7 - Full custom VLSI design of a Carry Select block in the final adder, with unresolved carry detection circuitry

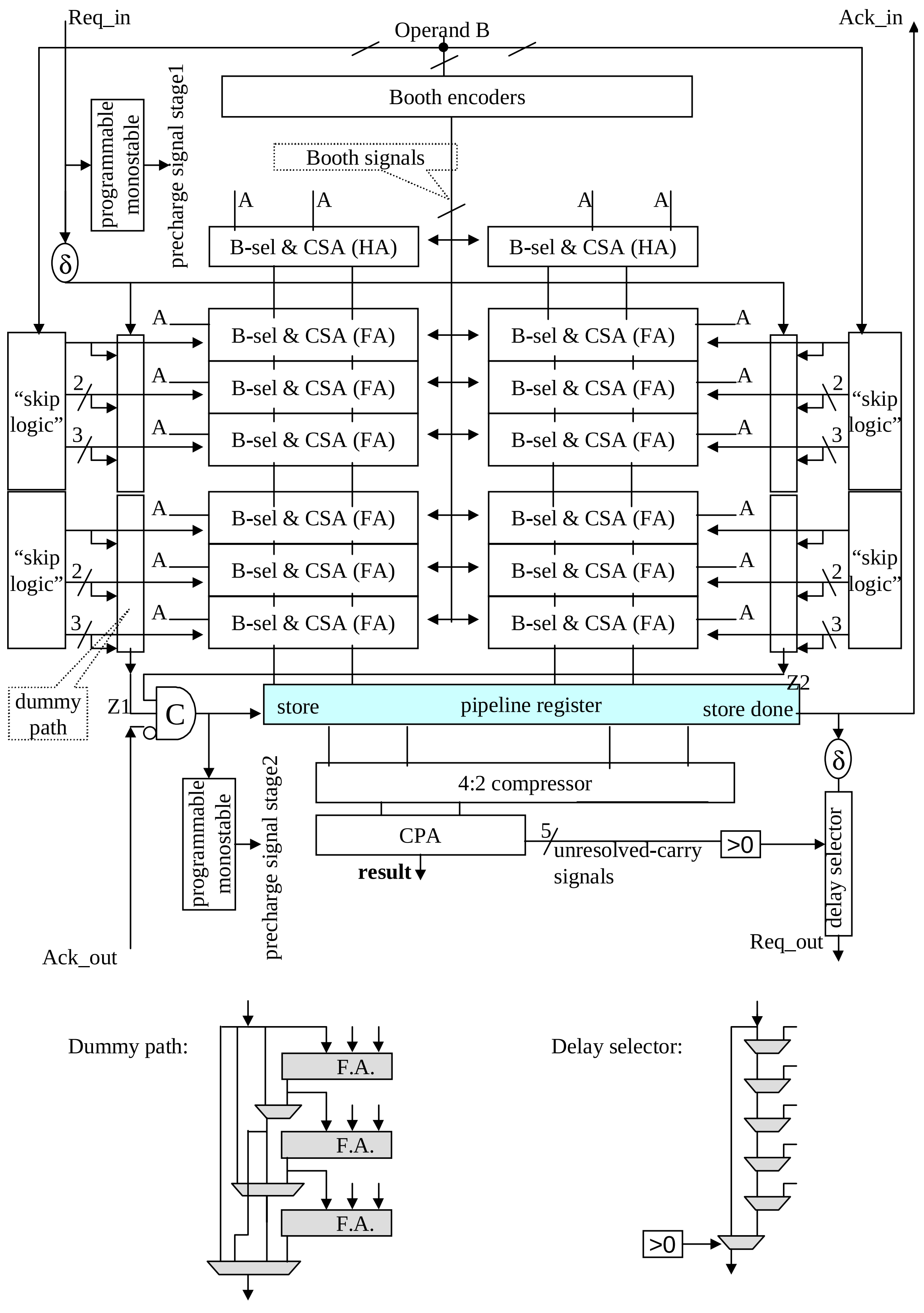

Fig. 8 – 32-bit self-timed implementation of the variable latency multiplier. δ = programmable delay element

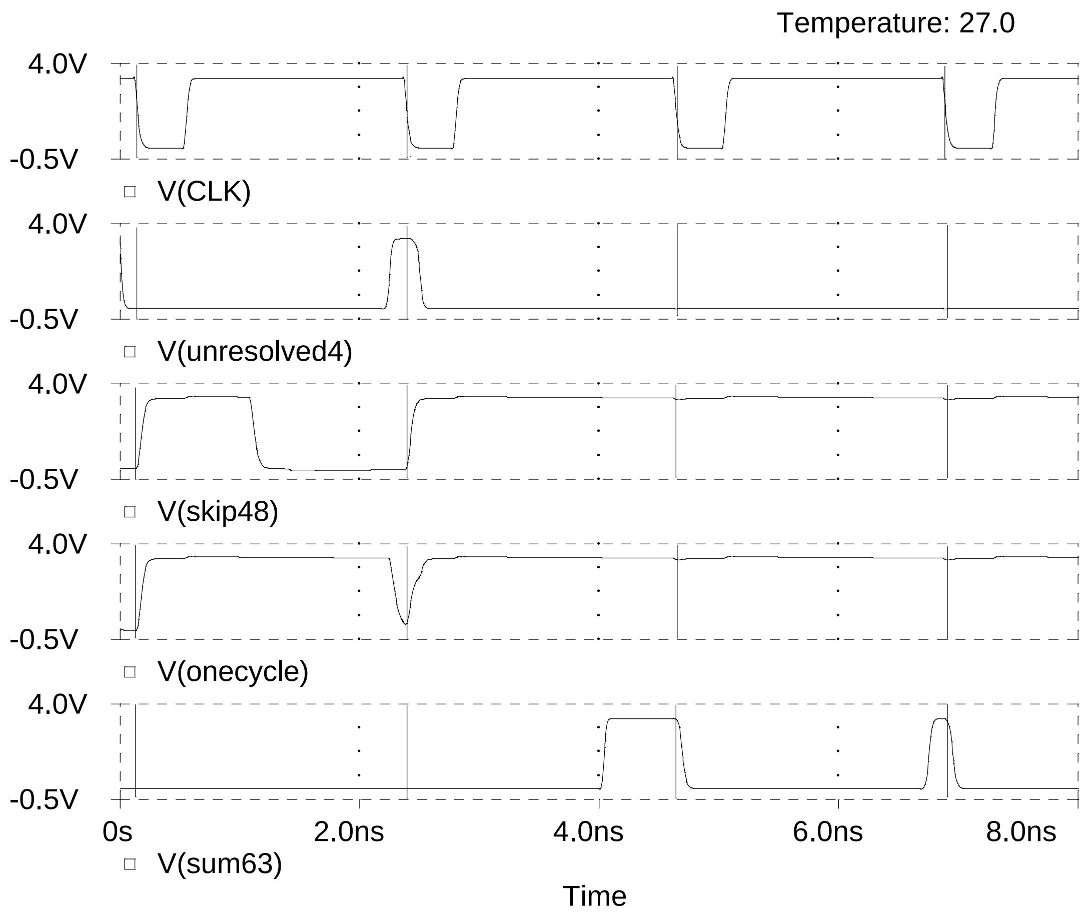

Fig. 9 - Spice simulation of the synchronous variable-latency multiplier CMOS implementation. Clock cycle borders are highlighted.