DESIGN TECHNIQUES FOR HIGH PERFORMANCE ASYNCHRONOUS ARITHMETIC OPERATORS

Xingcha Fan, Richard G. Burford and Neil W. Bergmann
School of Information Science & Technology
Flinders University, Adelaide, AUSTRALIA

ABSTRACT
High performance asynchronous arithmetic operator design techniques are proposed, which adopt some of the techniques commonly used in synchronous systems such as fast precharged logic and efficient latch design, while maintaining the features of localized and elastic pipelining control inherent in asynchronous design. A pipelined sixteen bit multiplier designed using these techniques is presented and its performance compared with several previously reported asynchronous and synchronous designs.

1. INTRODUCTION

Asynchronous logic design removes the global clock signal and hence the global timing constraints of conventional synchronous VLSI systems. The flow of data is dictated by local timing considerations. This attribute is becoming increasingly important as VLSI feature size is reduced and chip complexity increases. Other potential advantages of asynchronous design include lower power consumption, simplified system level design, and great product longevity. In recent years, asynchronous design is finding its way into high performance VLSI systems for Digital Signal Processing (DSP) applications.

Arithmetic operators are often the major building blocks and performance limiting factors for DSP and other numerical processing VLSI systems. In recent years, many asynchronous arithmetic operators have been reported [4, 14, 6, 7, 9], and some of them exhibit better overall performance or superiority in some performance metrics than comparable synchronous designs. For those asynchronous arithmetic operators demonstrating performance superiority, the performance advantages are mainly achieved by the exploitation of data-dependent *operational redundancies* in the arithmetic operation.

A typical example is a *carry-completion sensing* ripple carry adder. By detecting the actual longest carry propagation in the adder, a carry-completion sensing adder can achieve an average delay in the order of $O(\log_2 n)$ [5], while in a synchronous ripple carry adder the addition delay must account for the worst-case carry propagation delay in the order of $O(n)$. Another reported successful asynchronous arithmetic operator is a 54-bit self-timed divider based on a radix-2 SRT division algorithm [15, 14]. In this asynchronous divider, an *early-done* detection technique is used to terminate the division iterations and generate the division "*done*" signal as soon as the remainder repeats. Special techniques are applied to the pipelined ring structure of the divider to achieve *zero-overhead* performance. This divider has been shown to be much faster than the commercial synchronous divider chips.

However, for many arithmetic operations or implementation styles of arithmetic operators, data-dependent operational redundancies do not exist or often the performance gain of exploiting the operational redundancies is outweighed by the performance overhead introduced by doing so. An example is a fully pipelined array multiplier in which each carry save adder stage is of fixed delay without data-dependent operational redundancy. On the other hand, zero-overhead is not achievable for pipelined structures which do not have a ring structure. For a straight pipeline, zero-overhead can be achieved on latency, but at the penalty of about 50% throughput decrease.

A pipelined structure is commonly used in high performance arithmetic operator design. This paper will focus on the design of asynchronous pipelined arithmetic operators where no data-dependent operational redundancies are exploited.

It is generally believed that asynchronous pipelined arithmetic operators are slower and occupy a larger chip area than their synchronous counterparts and this is supported by several published studies [10]. We contend, however, that asynchronous pipelined arithmetic operators can be designed with performance similar or equal to that of equivalent synchronous ones by using design techniques proposed in this paper. When a *degradation factor* (typically 50% or greater [3]) is applied to the clock speed of synchronous designs to allow for temperature, supply voltage and fabrication process variations, asynchronous designs can exhibit a significant performance advantage.

In this paper, we propose design techniques for very high performance asynchronous arithmetic operators, which adopt some of the design techniques commonly seen in high speed synchronous VLSI design. These design techniques will be elaborated by the presentation of a sixteen bit fixed point multiplier design, which achieves a performance level similar to that of non-derated synchronous designs.

2. ASYNCHRONOUS PIPELINED STRUCTURES

In this section, we review some of the techniques used for asynchronous pipelined arithmetic operator designs.

2.1 Micropipeline

Micropipelines [11] is an asynchronous self-timed circuit or system design style using *two-cycle* or *non-return-to-zero* self-timed signaling and a *bundled data format*. Each bit of data is carried over a single wire. Arbitrary data widths are bundled and the flow of data is controlled by the exchange of common request and acknowledge handshake signals.

To indicate the completion of its operation, the computation delay of a computation block in the micropipeline is accounted for by introducing an explicit delay element in the control path which models or matches the delay of its critical path. This technique of *delay modeling* or *delay matching* must allow for the worst-case delay of each computation block. The delay matching technique is free of the completion detection delay incurred by other design techniques. It is efficient for completion indication in asynchronous system design where the delay of the computation block is data-independent, such as in a carry save adder stage of a pipelined multiplier.

In order to use dynamic precharged logic, a precharge period between successive evaluation phases is required. Since two-cycle signaling does not have a return-to-zero phase which can be used for precharging it is difficult or impossible to incorporate precharged logic into a micropipeline in its pure form. Computation within the pipeline is usually implemented in static complementary logic (CMOS). This results in larger and slower computation blocks than achievable using dynamic precharged logic because of the need for more complementary P type transistors.

In micropipeline structure, data latching between pipeline stages is controlled by signal transitions (either low-to-high or high-to-low). A special transition data latch structure [11] is required, which may limit the use of high speed and simple latch designs as can be used in synchronous or level controlled design.

2.2 Asynchronous pipeline using DCVSL

Differential Cascode Voltage Switch Logic (DCVSL) [2, 6] is a widely used logic style for self-timed computation block design. Data is *dual-rail* encoded, i.e. two wires are used to encode each data bit. Outputs are inverted so that both outputs are low at the precharge phase. This is particularly necessary when DCVSL gates are cascaded to form a computation block with a common $Precharge/Evaluation$ signal, to ensure that evaluation of DCVSL gate can commence only when the outputs of the preceding gate have settled. This introduces an inverter delay between each gate, which should be avoided in high performance arithmetic operator design.

DCVSL provides complementary outputs which can be used for completion detection by simply *ORing* the output pair. For a multi-bit computation block, completion detection of the output bundle can be done by a C-element tree or an AND-gate tree taking the *completion* signal of each output bit as inputs, or by simply *wired-ORing* their complements. Completion detection can produce a performance improvement when there is a data-dependent variation in computation delay. However, even then, the time and area overhead for completion detection may outweigh its advantages.

Meng proposed an asynchronous *full-handshake* pipeline structure in [6] using four-cycle self-timed signaling protocol and DCVSL computation block. By inserting data latches between computation blocks and carefully designing the handshaking control circuit, adjacent DCVSL blocks can perform evaluation on different data concurrently. Ignoring the C-element delays in the control circuit, the throughput ($\mathcal{T}$) and latency ($\mathcal{L}$) of a n stage full-handshake pipeline can be represented as

$$\mathcal{T} = 1/(D_e + D_{e:cd} + D_p + D_{p:cd} + 2D_l) \qquad (1)$$

$$\mathcal{L} = (D_e + D_{e:cd} + D_l) \times n \qquad (2)$$

where D_e is the evaluation delay of a DCVSL computation block, D_p is its precharging delay, $D_{e:cd}$ is the completion detection delay for the evaluation, $D_{p:cd}$ is the completion detection delay for the precharging, and D_l is the delay of data latch. Equation 2 is obtained assuming all the DCVSL blocks are identical.

From Equation 2, we can see that completion detection delay $D_{e:cd}$ contributes to the performance overhead of both throughput and latency. Throughput overhead is further caused by an extra precharging cycle i.e. $(D_p + D_{p:cd} + D_l)$, which is required by DCVSL logic and speed-independent four-cycle handshake signaling. To achieve a similar performance level to synchronous design, these overheads need to be reduced.

Williams in [13] proposed a technique to overlap the completion detection delay of a DCVSL block with the evaluation delay of its succeeding block. Data latches are removed between DCVSL computation blocks to further improve the latency. The throughput and latency of a n-stage pipeline using this technique (ignoring the control circuit delays) are

$$\mathcal{T} = 1/(3D_e + D_p + D_{e:cd} + D_{p:cd}) \qquad (3)$$

$$\mathcal{L} = D_e \times n \qquad (4)$$

From Equation 4, we can see that no delay overhead is introduced into the latency of this pipeline, therefore this is called a *zero-latency-overhead* pipeline. However, the throughput of this pipeline is largely degraded due to the removal of data latches and the character of DCVSL gate that inputs must remain unchanged to hold the output state. These two factors mean that any three adjacent stages in the pipeline cannot evaluate concurrently. This technique has been successfully used in a self-timed divider with ring structure, where the latency of the pipeline is the only performance deciding factor while the overall throughput ($\mathcal{T}$) of the divider is $1/\mathcal{L}$. For a normal pipeline, this technique is not very useful because of poor throughput performance, particularly when computation blocks consist of multiple stages of DCVSL gate, or D_e is large.

Improvement can be made by incorporating output holding function into the DCVSL gate. By inserting a cut-off transistor between the NMOS evaluation tree and each of the output node which connects its gate to the opposite output node, a modified DCVSL circuit is able to hold its output state after the evaluation completed and allow the free change of its inputs. By using DCVSL circuits modified in such a way and carefully designing the control circuit, the throughput of the pipeline can be improved to

$$\mathcal{T} = 1/(2D_e + D_p + D_{e:cd} + D_{p:cd}) \qquad (5)$$

while achieving the same latency. However, due to the use of cut-off transistors which introduce an extra stage of serial transistors in the NMOS tree, the evaluation delay, D_e, of the modified DCVSL gate is increased.

Another logic family, *Latched Differential Pass Transistor Logic* (LDPL), is proposed by Salomon *et al* [9] for their fully pipelined multiplier design. By using pass transistor logic for the NMOS evaluation tree and cut-off transistors between evaluation tree and output nodes which connects their gate to the precharge/evaluation signal, it allows the evaluation of the tree to be carried out as soon as the inputs are set up even when the LDPL circuit is still in precharge phase. Output latching is also incorporated, although in a different sense from the modified DCVSL gate that output state holds when LDPL circuit is in the precharging phase. With this output latching feature, a LDPL circuit stage can start precharging while its following stage is still evaluating on its outputs. The use of LDPL can further improve the throughput of pipeline. However, this may be outweighed by the complexity and the speed sacrifice of LDPL. A multiplier designed using LDPL will be compared later in the paper.

3. DESIGN TECHNIQUES FOR HIGH PERFORMANCE ASYNCHRONOUS PIPELINES

The asynchronous pipeline structures and the design techniques described in the last section have limitations and introduce various performance and hardware cost overheads compared with synchronous pipelines. For example, the two-phase handshaking signaling protocol of the micropipeline structure excludes the use of high-speed dynamic precharged logic for the processing block, and requires more complex and slower event-controlled registers. The performance of normal four-phase handshaking pipeline with DCVSL processing logic is affected by completion detection delays and the pipeline throughput is further affected by the extra return-to-zero phases required by the four-phase signaling protocol. Although delay-matching completion indication technique can be applied to DCVSL processing logic block which reduces the $D_{e:cd}$ and $D_{p:cd}$ delays to approximately *zero*, DCVSL processing logic still has the disadvantages of longer evaluation delay due to the *inverters* interposed between DCVSL gates, and higher hardware requirement due to the dual-rail signal encoding. Although techniques can be used to achieve zero latency overhead, these improvements are usually achieved at the expense of throughput.

In conclusion, the reported asynchronous pipeline structures and design techniques are not sufficient to design high-performance pipelined arithmetic operators in which operational redundancy is *not* exploited within processing blocks. The idea behind our design technique for high-performance asynchronous pipelines is that *to achieve similar performance level to synchronous designs, techniques commonly used in high performance synchronous design should be extensively adopted.* The key points of our design technique is described as follows.

First, delay matching techniques should be used for indicating the operation completion of processing blocks (pipeline stages). By using delay matching completion indication, the performance of processing blocks can be exploited to *at least* the same extent as in synchronous design. Thus, considering only the processing block, an asynchronous design should achieve at least equivalent performance to a synchronous design. In order to guarantee correct operation over the required temperature and supply voltage range and to allow for fabrication process variations, a degradation factor is generally applied to the clock speed of synchronous design. In contrast, the delay matching technique allows an asynchronous design to achieve optimal performance for given operating conditions by tracking the variation in computational delay.

Second, to achieve very high performance as seen in synchronous designs, high-speed logic circuit design techniques, such as dynamic precharged logic are used. This excludes the use of the micropipeline structure and DCVSL processing logic.

Third, data latching is done in a similar manner to synchronous design, but with a locally generated *latching signal pulse* instead of a global clock signal. This pulse is of fixed width, initiated by the completion indication signal of each processing block. The width of the pulse does not contribute to the handshaking loop which determines the pipeline throughput. High-speed latch designs are used.

Inherent in our design technique is the assumption that logic gate and interconnection wire delays are *bounded* and can be modeled (i.e. *bounded wire delay* model), much in the same way as implied in synchronous design. We sacrifice speed-independence or delay-insensitivity in order to achieve higher performance. The design technique is demonstrated in the next section, through the design of a very high-performance asynchronous pipelined multiplier.

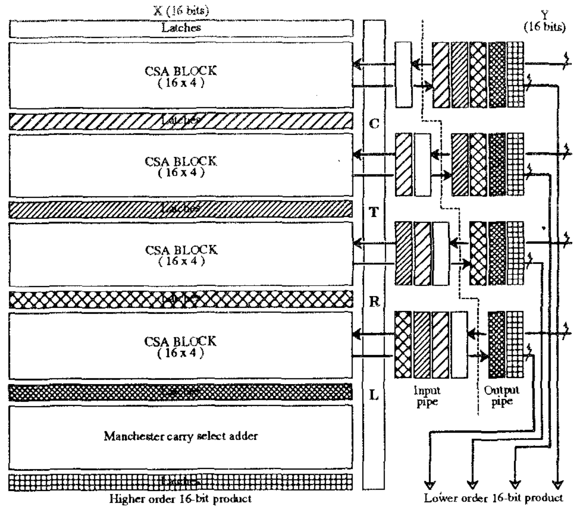

Figure 1: The floorplan of the 16×16 pipelined multiplier

4. A 16-BIT ASYNCHRONOUS PIPELINED MULTIPLIER

In this section, we present an 185MMPS (Million Multiplications Per Second) 16×16 asynchronous pipelined unsigned multiplier to demonstrate the high-performance asynchronous pipeline design technique. The multiplier is implemented in an $1.2\mu m$ single-poly double-metal CMOS process [8].

4.1 Overall structure of the multiplier

The multiplier is composed of *sixteen* stages of 16-bit carry save adders (CSA) and a combination of Manchester carry adders and carry select adders to merge the final two vectors of carry and partial sum values generated by the CSA array. It is implemented as a *five* stage pipeline, with the top four pipeline stages each consisting of four cascaded 16-bit carry save adders (each of these four pipeline stages is called a **CSA block**), and the last pipeline stage consisting of the Manchester carry select adder. Data latches are used between pipeline stages to store the partial products and operand data flowing through the pipeline. The selection of this five stage pipeline is the result of a trade-off between throughput, latency, and chip area. It has also been based on the consideration that delays of pipelined stages are balanced and evenly distributed. Increasing the number of stages of pipelining increases the number of interstage latches, and input latches for staggering the **Y** operand (or *multiplier*). This results in greater throughput at the expense of a longer latency time and larger chip area. Five stages of pipelining allow us to maximize the multiplier throughput while not exceeding the allowable active die area (excluding pads) of $3.24mm^2$ for the chosen prototype fabrication process (Orbit Semiconductor's Tiny Chip).

The floorplan of the pipelined multiplier is shown in Figure 1. The **X** operand (or *multiplicand*) is input at the top of the CSA array and flows down through the interstage latches. **Y** operand is input from the right, and is staggered through the pipelined input latches (**input pipe**) so that **Y** bits (in the form of 4-bit digits) reach CSA blocks at the same time as the respective **X** operand. Each CSA block generates four bits of lower order product which are output through the pipelined output latches (**output pipe**). The final 32 bit product is output at the last pipelined latches in parallel. The interstage latches also latch the intermediate carry and sum

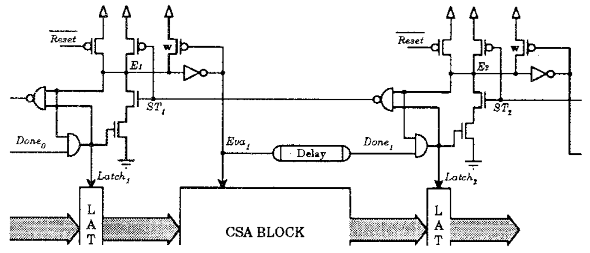

Figure 2: Pipelining and data latching control circuit

results from CSA blocks. In Figure 1, latches filled with the same cross-hatch pattern are controlled by the same latching signal. The handshaking between pipeline stages are controlled by the control path (**CTRL**) which occupies a vertical strip between the adder array and the input/output pipes. The control path also generates the latching signals for the latches.

4.2 The control path

The control path circuit which performs the pipelining control and generates local data latching signals is shown in Figure 2. It generates the data latching signal, $\boldsymbol{Latch}$, to latch the outputs of a CSA block, as well as **X** operand and digits of **Y** operand from previous latches, *after* the CSA block completed evaluation, and *when* its following CSA block is empty or has had its output latched. Therefore, the control path provides an *elastic* pipelining control. The data latching signal is locally generated and, similar to the clock for a synchronous pipeline, is a positive pulse of *fixed* width.

The operation of the control circuit in Figure 2 is explained as follows. Before the start of pipelined operation, the control path is reset ($\overline{Reset}$ low), so that the original state of E_1, E_2, ST_1, ST_2 are all high, while $Done_0$, $Done_1$ are low, and the pipeline is empty. When $Done_0$ goes high after the previous CSA block completes its evaluation, $\boldsymbol{Latch_1}$ goes high, latching its output data into the latch. E_1 is then pulled down to low starting the evaluation of the current CSA block by driving $\boldsymbol{Eva_1}$ to high. The drop of E_1 also drives $\boldsymbol{Latch_1}$ to low to complete the fixed width pulse of $\boldsymbol{Latch_1}$.

While E_1 is low, it prevents the latching of new data into the data latch until the current CSA block has completed its evaluation and latched its output. Here, the rising of $\boldsymbol{Latch_2}$ will generate a negative pulse of ST_1 which sets E_1 to high, so that a new latching cycle of $Latch_1$ may start. The delay element in Figure 2 matches the delay of CSA block to indicate the completion of its evaluation.

As can be seen, this control circuit implementation is not speed-independent. However, the delay assumptions that have been made, such as on the width of $\boldsymbol{Latch}$ and ST signal pulses, are reasonable and reliable.

4.3 Progressive evaluation

To achieve higher performance, dynamic precharged logic is used for the carry save adders, as shown in Figure 3. The sum circuit performs the logic function $S_o = S_i \oplus C_i \oplus P$ and the carry circuit performs the logic function $C_o = S_iP + C_iP + S_iC_i$, where S_i and C_i are the sum and carry signals from the preceding carry save adder stage, P is obtained by the logic *AND* of the corresponding bits in the **X** and **Y** operands. No carry propagation occurs in the horizontal direction in each 16-bit carry save adder. Complementary inputs and outputs are used to eliminate the need of *inverters* for obtaining complementary signals. This minimizes the delay of each

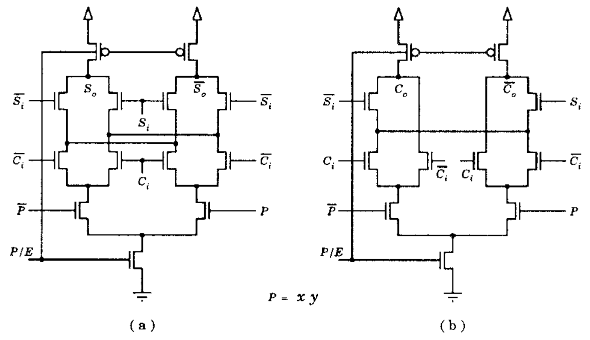

Figure 3: Carry save adder circuits: (a) sum (b) carry

carry save adder without sacrificing much of the chip area because of the simplicity of the circuit. Simulations using normal SPICE device parameter models for the ORBIT process [8] show that both sum (S_o or $\overline{S}_o$) and carry (C_o or $\overline{C}_o$) outputs which evaluate to a low state reach 50% of rail voltage in 0.6ns and 25% within 0.8ns.

Four stages of such carry save adders are cascaded to form a CSA block. To ensure that the NMOS pull down tree evaluates correctly, it is essential that evaluation of a given carry save adder stage does not commence until all its inputs are valid and stable. One technique to ensure this is to interpose *inverters* between each carry save adder while using a common *P/E* signal for all stages, like in a normal DCVSL processing block. However, this increases the delay of CSA blocks. We use a **Progressive Evaluation** technique, in which adjacent evaluation stages are released from precharge after the output of the preceding stage has settled, similar to a multi-phase, synchronously-clocked, precharged-logic system.

The complete control path incorporating the Progressive Evaluation control is re-drawn in Figure 4. The delay element is composed of a string of *NAND* gates and *inverters* which model the delay of carry save adder stages. Timing for the precharge and evaluation phases is derived from taps in the delay element. The evaluation of a carry save adder can be commenced once any low level inputs have settled below the threshold of the NMOS evaluation tree. A delay of 0.8ns is allowed between successive evaluations (i.e. *P/E* signals). This delay allows outputs of a carry save adder, which evaluate to low state, reach below 25% of the rail voltage, i.e. approximately 1.25V. Although this voltage is still a fraction higher than the *on-threshold* voltage of NMOS transistors (typically 1V), only an insignificant amount of stored charge in its succeeding carry save adder is discharged before the associated NMOS transistors are turned off by the further drop of low level output signals. Correct evaluation of its following carry save adder stage is ensured.

The carry save adders in a CSA block are precharged concurrently after the outputs of the CSA block are safely latched. The precharging of the CSA block is driven by the $\boldsymbol{Pre}$ signal, which goes low slightly before the $\boldsymbol{Latch}$ signal goes high. The *P/E* signals are pulled down *one NAND* gate delay *behind* the rising of $\boldsymbol{Latch}$ signal. This delay, plus the precharging delay of carry save adders, together with the special character of our *self-latching* latches guarantee the reliable latching of CSA block output, while allowing overlap between data latching and CSA block precharging. Similarly, the signal generating the negative pulse of ST is derived one stage early to improve the pipeline throughput. The $\boldsymbol{Latch}$ and

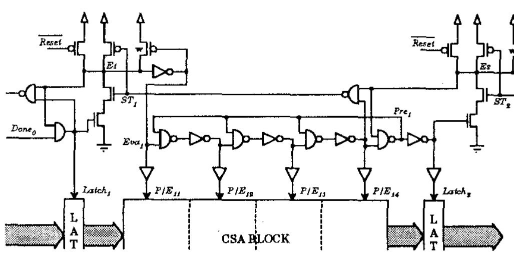

Figure 4: Control path for Progressive Evaluation and data latching

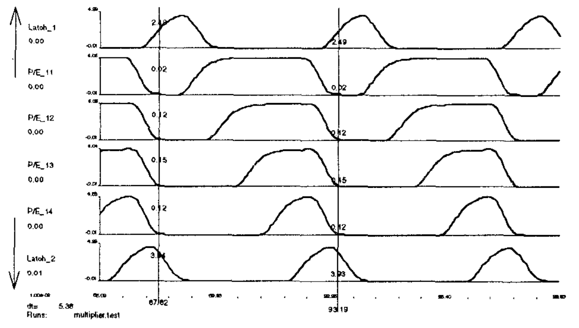

Figure 5: Data latching and Progressive Evaluation control signals

P/E signals of one pipeline stage, obtained from a SPICE simulation for the whole multiplier control path with the full load of all the $Latch$ and *P/E* signals is shown in Figure 5.

4.4 Manchester carry select adder

The final pipeline stage of the multiplier resolves the higher-order sixteen bits of the product by adding the sum and carry vectors (both are 16 bits) generated by the carry save adder array. To avoid becoming the performance bottleneck and to match the delay of the other pipeline stages, i.e. CSA blocks of approximately 3.3ns, a combination of Manchester carry adders and carry select adders [12] are used, referred to as a **Manchester carry select adder**.

We use a precharged logic carry lookahead circuit generally called a **Manchester carry chain (MCC)** [12]. *Four* bits of carry signals are generated in each MCC to restrict the number of serial transistors in the evaluation path. Dynamic precharged logic is used for the **Generate Propogate (GP)** circuits, MCCs, and the circuits generating the carry signals. Progressive Evaluation is again used. The sum blocks use normal static complementary CMOS logic, because their delay is not critical due to overlap with the evaluation of the carry signals.

SPICE simulation shows the delay of the Manchester carry select adder, i.e. the delay from GP block starting evaluation to final carry becoming valid is about 3.4ns which is close to the delay of other pipeline stages. The multiplexing for the output sum selection is incorporated into the data latching stage.

Salomon *et al.* have also designed a 16-bit pipelined adder, using a combination of Manchester carry chain and the carry select technique, as the final stage of a 16 $\times$ 16 multiplier [9] to merge the partial products of the pipelined carry-save multiplier core. The pipelined adder is implemented with 4 pipeline stages using a dynamic pass transistor logic family called LDPL (*Latched Differential Pass Transistor Logic*). Their simulations show that the adder implemented with $1.0\mu m$ CMOS technology has the addition latency of approximately 13ns.

Our Manchester carry select adder is much faster than the one reported by Salomon *et al.*, because of the use of dynamic precharged logic with Progressive Evaluation, as well as careful transistor sizing and layout design.

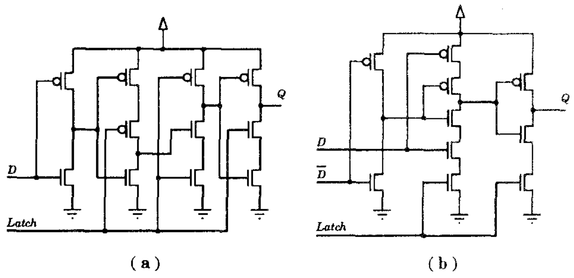

Figure 6: Data latches: (a) positive edge triggered flip-flop (b) self-latching register

4.5 Data latches

Data latches for **X** and **Y** operands, inputs and output pipes use a 9-transistor single-phase positive edge triggered D flip-flop [1]. The circuit, shown in Figure 6(a) can be implemented with fewer transistors than micropipeline-style event-controlled registers [11] and exhibits near zero data hold time. An input *inverter* is used to give a non-inverting latch, resulting in eleven transistors for each latch. Because this input inverter is not in the critical delay path of the CSA blocks and **X**, **Y** operands are always ready well before the latching signals in the interstage latches, it will not introduce any performance loss. Simulations indicate the latch has a setup time of 0.8ns and total delay of 1ns under normal operating conditions.

Carry and sum outputs of CSA blocks are latched using a so called **self-latching** register shown in Figure 6(b). The complementary outputs of carry and sum are used as the input of the register. When the latch signal $Latch$ is high and D is not equal to $\overline{D}$, the output Q will equal D. When $Latch$ is low, or D equals $\overline{D}$ (i.e. when the block is precharged or still in evaluation) the output will held. This relaxes constraints of $Latch$ signal width, and the timing between $Latch$ signal and *P/E* signals. Simulation results shown a delay of 0.9ns from input to output.

4.6 Performance analysis and comparison

The chip layout of the multiplier implemented on a 40-pin Orbit Semiconductor Tiny Chip is shown in Figure 7. The large block at the upper-left of the layout is the CSA array (four pipeline stages), and the less regular part of the layout at the bottom is the Manchester carry select adder. The right strip of the layout is the input and output pipe for the staggered **Y** operand and lower order product bits. In between the CSA array and input/output pipes is the control path (CTRL) of the multiplier. The effective chip area occupied by the multiplier, excluding the input and output multiplexing circuit required by the limited pin number, is $3.03mm^2$. The control path occupies only 4.7% of the effective multiplier area.

SPICE simulations using normal device parameter models for the Orbit $1.2\mu m$ CMOS process at an operating temperature of 27°C indicate the multiplier is capable of accepting input operands

Design Style	DCVSL[6]	LDPL[9]	Mpipe.[9]	Sync.[9]	**Prog.Eva**
Feature size (μm)	1.6	1.0	1.0	1.0	**1.2**
Area (mm^2)	8.1	2.59	2.64	2.53	**3.03**
Area (scaled to $1\mu m$)	3.16	2.59	2.64	2.53	**2.1**
Throughput (MMPS)	26.3	156	104	172	**185**
Latency (ns)	38	64	-	-	**23**

Table 1: Performance comparison of several multipliers

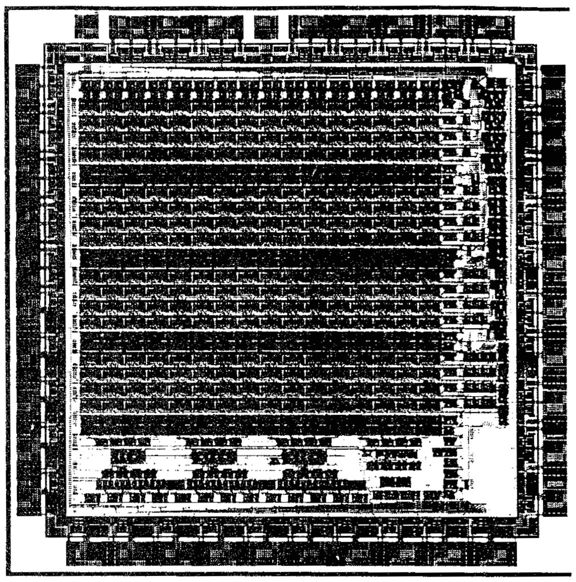
Figure 7: Chip layout of the 16×16 asynchronous multiplier

every 5.4ns or an effective throughput rate of 185 MMPS (million multiplications per second). Latency of the multiplier is 23ns.

Table 1 compares the performance of several 16 × 16 multiplier designs. The "DCVSL multiplier" is reported by Meng [6]. It is a non-pipelined Booth-encoded multiplier core, which outputs two 16-bit carry and sum vectors (like the output from our CSA array), instead of the final product. The "LDPL multiplier" is a fully pipelined multiplier designed by Salomon *et al.* [9] using *Latched Differential Pass Transistor Logic* (LDPL). The micropipelined and synchronous multiplier designs are also cited from [9]. The entries shown for "Area(scaled)" have been scaled to $1\mu m$ feature size for comparison purposes. As shown in the table, our design technique of Progressive Evaluation (Prog.Eva) demonstrates the best performance as well as the smallest chip area.

A clocked synchronous design using the circuit design techniques similar to those presented here (including Progressive Evaluation) may be able to achieve performance equal or perhaps slightly better than our design (without allowing for a degradation factor). Such a design would not, however, have the *elastic* pipeline properties, and requires the distribution of a 185MHz global clock signal which is not a easy task.

5. CONCLUSIONS

High performance asynchronous arithmetic operator design techniques have been presented in this paper. An asynchronous pipelined multiplier designed using the proposed techniques demonstrates a performance level similar to that of a non-derated synchronous design, while maintaining the inherent advantages of asynchronous design such as elastic pipelining control. The proposed design techniques are applicable to many asynchronous, pipelined VLSI design problems.

REFERENCES

[1] M. Afghahi and C. Svensson. A unified single-phase clocking scheme for VLSI systems. *IEEE J. Solid-State Circuits*, SC-25(1):225 – 233, February 1990.

[2] K.M. Chu and D.L. Pulfrey. A comparison of CMOS circuit techniques: Differential cascode voltage switch logic versus conventional logic. *IEEE J. Solid-State Circuits*, SC-22(4):528–532, August 1987.

[3] M.E. Dean. STRiP: A self-timed RISC processor. Technical Report CSL-TR-92-543, Computer System Laboratory, Stanford University, July 1992.

[4] J.D. Garside. A CMOS VLSI implementation of an asynchronous ALU. *Proc. IFIP Working Conference on Asynchronous Design Methodologies, Manchester, England*, March 1993.

[5] K. Hwang. *Computer Arithmetic: Principles, Architecture, and Design*. John Wiley & Sons, 1979.

[6] T.H.-Y. Meng. *Synchronization Design for Digital Systems*. Kluwer Academic Publishers, 1991.

[7] C.D. Nielsen and A.J. Martin. A delay-insensitive multiply-accumulate unit. Technical Report Caltech-CS-TR-92-03, Computer Science Department, California Institute of Technology, 1992.

[8] Orbit Semiconductor Inc., Sunnyvale, CA. *Foresight Users Manual*, rev 1.4 edition, July 1991.

[9] O. Salomon and H. Klar. Self-timed fully pipelined multiplier. *Proc. IFIP Working Conference on Asynchronous Design Methodologies, Manchester, England*, March 1993.

[10] J. Sparsø, C.D. Nielsen, L.S. Nielsen, and J. Staunstrup. Design of self-timed multipliers: A comparison. *Proc. IFIP Working Conference on Asynchronous Design Methodologies, Manchester, England*, March 1993.

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

[12] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design, A System Perspective*. Addison-Wesley, 1985.

[13] T.E. Williams. Analyzing and improving the latency and throughput performance of self-timed pipelines and rings. *Proc. 1992 IEEE International Symposium on Circuits and Systems*, pages 665–669, May 1992.

[14] T.E. Williams and M.A. Horowitz. A zero-overhead self-timed 160-ns 54-b CMOS divider. *IEEE J. Solid-State Circuits*, 26(11):1651–1661, Nov. 1991.

[15] T.E. Williams, M.A. Horowitz, R.L. Alverson, and T.S. Yang. A self-timed chip for division. *Proc. 1987 Stanford Conference on Avanced Research in VLSI*, pages 75–96, March 1987.