2012 IEEE 18th International Symposium on Asynchronous Circuits and Systems

An Asynchronous Divider Implementation

Navaneeth Jamadagni
Maseeh College of
Computer Science and Engineering
Portland State Univeristy
Portland, OR 97201
Email: navaja@cecs.pdx.edu

Jo Ebergen
Oracle Labs
Redwood Shores, CA 94065
Email: jo.ebergen@oracle.com

***Abstract*—We present an asynchronous implementation of a novel division algorithm previously patented in [1]. Our implementation exploits the average-case behavior of the algorithm and uses the versatility of GasP circuits to implement the data-dependent latencies in the algorithm. On average, the delay per quotient bit for our implementation is 6.3 FO4 gate delays compared to 9.5 FO4 gate delays for a similar SRT divider implementation.**

***Keywords*—Asynchronous Divider; GasP; Division Algorithm H; Variable Latency;**

I. Introduction

Division is one of the most complex arithmetic operations performed in microprocessors. Although division occurs less frequently than addition or multiplication, Oberman shows that an efficient division implementation is necessary for good system performance [2]. There are several division algorithms available to implement in hardware. These algorithms can be broadly classified into digit recurrence algorithms and multiplicative algorithms. The multiplicative algorithms include Newton-Raphson's method and Goldschmidt's algorithm [3]. The digit recurrence algorithms include SRT division [4], F-Division algorithm [5], algorithm E in [6], and algorithm H in [1].

The SRT division algorithm is the most frequently implemented algorithm in general purpose processors. The name SRT comes from the initials of Sweeny, Robertson and Tocher; they independently developed this algorithm at approximately the same time. Harris presents an analysis of various SRT divider architectures and circuit styles in [7]. In this paper we present an analysis of a novel asynchronous implementation of algorithm H in [1]. We show that by exploiting the average-case behavior of asynchronous circuits, we can achieve an average delay per quotient bit of 6.3 FO4 delays compared to 9.5 FO4 delays in a comparable SRT implementation as shown by Harris [7]. The simulations show that the average energy consumption per division of 25-bit operands in a 90nm CMOS technology is 182pJ.

We briefly summarize algorithm H in Section 2. In section 3 we describe the hardware implementation which exploits the average-case behavior of algorithm H. We discuss the spice simulation results and compare the results with a similar SRT implementation in section 4.

II. Division Algorithm

In a simple binary long division process, each repetition step involves either left shifting the partial remainder by one, or subtracting the divisor from the partial remainder followed by a left-shift by one. The selection of one of these two alternatives depends on whether the partial remainder is less than or at least the divisor. In each step a quotient digit is accumulated from the digit set $\{0, 1\}$, where 0 corresponds to a shift operation and 1 corresponds to a subtract & shift operation. Several enhancements can speed up the division process, for example by replacing carry-propagate additions with carry-save additions. These improved algorithms must select one of at least three alternatives for each repetition step. For example, a standard radix-2 SRT algorithm with carry-save addition executes one of three alternatives in each iteration. The three alternatives are addition of divisor followed by shift, shift, and subtraction of divisor followed by shift. For each operation, the SRT algorithm selects a corresponding quotient digit from the digit set $\{-1, 0, 1\}$. Typically, the selection of an alternative and quotient digit relies on the most significant bits of the partial remainder. For an SRT division, the selection logic often inspects the three or four most-significant bits of the partial remainder. Algorithm E in [6] and H in [1] simplify the selection logic by inspecting only the two most-significant bits of the partial remainder rather than three or four. This simplification comes at a cost of selecting from more alternatives in each repetition step. Algorithm E in [6] executes one of six alternatives in each repetition step, and Algorithm H in [1] executes one of seven alternatives in each repetition step. Like a simple SRT algorithm, Algorithm E retires one quotient digit in each repetition step. Algorithm H, however, can retire one or two quotient digits per repetition step. We show that simplifying the selection logic overcomes the extra cost of selecting among more alternatives and leads to a shorter average latency per quotient bit.

Rather than just considering division, we consider computing the result of multiplication and division at the same time. That is, $Q = C * (R/D)$, where Q is the quotient, $C * R$ is the dividend and D is the divisor; if $C = 1$, Q is the quotient of a simple division operation. However if $C \neq 1$, then Q is the result of a division along with multiplication. In [1] and [6], the authors present generalizations of various division algorithms

1522-8681/12 $26.00 © 2012 IEEE
DOI 10.1109/ASYNC.2012.17

to perform both division and multiplication at the same time. The following section summarizes algorithm H from [1].

A. *Algorithm H*

Because we are considering a hardware implementation, we make some assumptions about the ranges for C, R, and D which are binary numbers with fractional bits.

$$C \in [0, 2^K] \tag{1}$$
$$R \in [-2^{K+1}, 2^{K+1}) \tag{2}$$
$$D \in [2^K, 2^{K+1}) \tag{3}$$

For IEEE 754 single and double precision formats, $K = 0$. The number of repetitions required per division is determined by the number of fractional bits, L, in a floating point number. For IEEE single precision format $L = 23$ and for IEEE double precision format $L = 52$.

Algorithm H appears below.

```
1: rs:=R; rc:=0; qs:=0; qc:=0; c:=C; n:=0
2:   while(n ≤ K+L+2) do
3:     if ((rs, rc) in 2X) then
4:        rs, rc := rs⋆2, rc⋆2;
5:        c:= c/2; n:=n+1;
6:     elseif ((rs, rc) in 2X*) then
7:        rs, rc := rs⋆2, rc⋆2;
8:        invert(K+1, rs, rc);
9:        c:= c/2; n:=n+1;
10:    elseif ((rs, rc) in 4X*) then
11:       rs, rc := rs⋆4, rc⋆4;
12:       invert(K+1, rs, rc);
13:       c:= c/4; n:=n+2;
14:    elseif ((rs, rc) in ADD1 & 2X*) then
15:       rs, rc := add(rs, rc, D);
16:       qs, qc := add(qs, qc, -c);
17:       rs, rc := rs⋆2, rc⋆2;
18:       invert(K+1, rs, rc);
19:       c:= c/2; n:=n+1;
20:    elseif ((rs, rc) in SUB1 & 2X*) then
21:       rs, rc := add(rs, rc, -D);
22:       qs, qc := add(qs, qc, c);
23:       rs, rc := rs⋆2, rc⋆2;
24:       invert(K+1, rs, rc);
25:       c:= c/2; n:=n+1;
26:    elseif ((rs, rc) in ADD2 &  2X*) then
27:       rs, rc := add(rs, rc, 2D);
28:       qs, qc := add(qs, qc, -2c);
29:       rs, rc := rs⋆2, rc⋆2;
30:       invert(K+1, rs, rc);
31:       c:= c/2; n:=n+1;
32:    elseif ((rs, rc) in SUB2 & 2X*) then
33:       rs, rc := add(rs, rc, -2D);
34:       qs, qc := add(qs, qc, 2c);
35:       rs, rc := rs⋆2, rc⋆2;
36:       invert(K+1, rs, rc);
37:       c:= c/2; n:=n+1;
38:    endif
39:  endwhile
```

We briefly explain the program. For a complete correctness proof, please see [1], [6]. The program variables qs and qc represent the partial quotient q in carry-save form, where $q = qs + qc$. The program variables rs and rc represent the partial remainder r in carry-save form, where $r = rs+rc$. The function $add(x,\ y,\ z)$ represents carry-save addition. It takes three inputs x, y, and z and produces two results $add_s(x,\ y,\ z)$ and $add_c(x,\ y,\ z)$ such that

$$add_s(x,\ y,\ z) + add_c(x,\ y,\ z) = x + \ y + \ z$$

The result $add_s(x,\ y,\ z)$ is the bit-wise parity of x, y, and z, and the result of $add_c(x,\ y,\ z)$ is the bit-wise majority of x, y, and z left shifted by one. The parity bits are also called partial sum bits. The majority bits, left shifted by one, are also called partial carry bits. In line 15 of the program above, the statement $rs,\ rc := add(rs,\ rc,\ D)$, means that the result of $add_s(rs,\ rc,\ D)$ is assigned to rs and the result of $add_c(rs,\ rc,\ D)$ is assigned to rc.

The function $invert(K+1,\ rs,\ rc)$ in lines 8, 12, 18, 24, 30, and 36 inverts the $K+1^{st}$ bit in rs and rc, where $K+1$ is the position of the most-significant bit. This operation represents a translation over $(2^{K+1}, -2^{K+1})$ or $(-2^{K+1}, 2^{K+1})$ in the (rs, rc) plane (see Fig. 1) and keeps the value of $rs + rc$ unchanged. The repetition of algorithm H maintains the following invariants

$$(qs + qc) * D + c * (rs + rc) = C * R \tag{4}$$
$$rs \in [-2^{K+1}, 2^{K+1}) \tag{5}$$
$$rc \in [-2^{K+1}, 2^{K+1}) \tag{6}$$

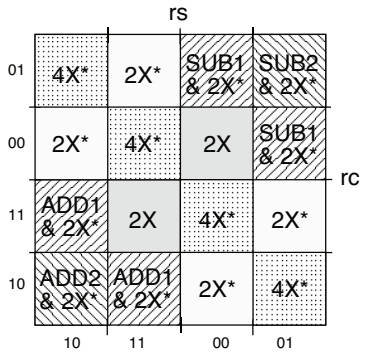

Fig. 1. The rs and rc plane: An illustration of various regions with their corresponding actions for Algorithm H. Both rs and rc are in two's complement form and their two most significant bits are indicated along the axes. The sum of rs and rc is the partial remainder r. Therefore a point in the center of the figure denotes $r = 0$. A point on the diagonal across the squares labeled 4X* has a value of r very close to 0. The value of r increases as a point moves towards the Northeast in the figure and the value of r decreases as a point moves towards the Southwest in the figure.

Algorithm E of [6] is very similar to Algorithm H, but replaces all 4X* operations by 2X* operations resulting in bigger 2X* regions in the second and fourth quadrants in Figure 1.

The seven alternatives, 2X, 2X*, 4X*, ADD1 & 2X*, ADD2 & 2X*, SUB1 & 2X*, and SUB2 & 2X* in the program correspond to the regions of Figure 1. Choosing the correct alternative relies only on the two most significant bits of rs and rc. The * in the alternatives indicates the inversion of the most-significant bit of rs and rc.

Algorithm H computes two quotient digits when the algorithm does the 4X* operation and one quotient digit for all other operations. This means that the number of iterations per division varies depending on how many times the algorithm executes the 4X* operation, which in turn depends on the input operands. A traditional radix-2 SRT algorithm and algorithm E retire one quotient bit per repetition step and take a fixed number of repetitions to perform a division operation. Figure 2 shows the probability distribution of the number of iterations per division for a radix-2 SRT algorithm, for algorithm E, and for algorithm H. For a division of 25-bit numbers, the SRT algorithm takes 25 iterations to compute the result. Algorithm E takes 26 iterations to obtain the same accuracy, because the inaccuracy in the result of Algorithm E is slightly larger than that of SRT division. Algorithm H takes on average between 22 to 23 iterations to compute the result with the same accuracy.

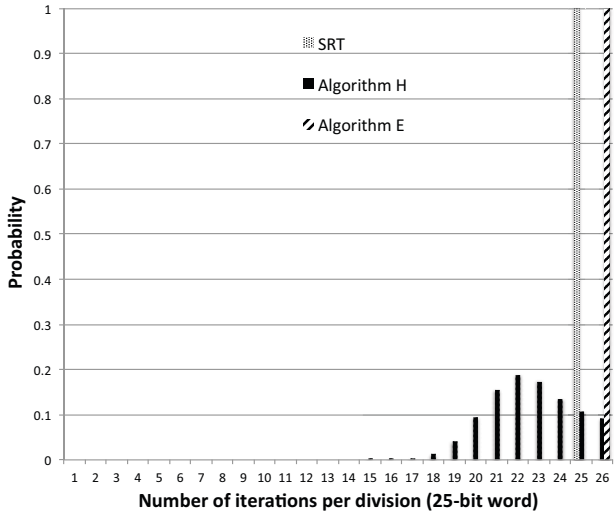

Fig. 2. Probability distribution of number of repetitions per division for Algorithm E, Algorithm H, and SRT algorithm. For random input operands, Algorithm H on average takes 22.6 iterations per division.

As Figure 2 shows, the number of iterations for Algorithm E ranges from 13 to 26 with an average that is less than the number of iterations in an SRT algorithm. Furthermore, from statistical analysis we found that on average 66% of all repetitions in an SRT division consist of an addition and a shift, where a subtraction counts as an addition [6]. In Algorithm H only 48% of all repetitions consist of an addition plus shift. In Algorithm E this percentage is 42% [6]. This difference in number of additions can be exploited further in each repetition step if the latency in a path with only shifts is smaller than the latency in a path containing an addition followed by a shift.

III. IMPLEMENTATION

In this section we describe an asynchronous implementation of algorithm H. The divider is implemented as a stage in a pipeline, where an input FIFO delivers operands to the division stage and an output FIFO takes the results from the division stage. The implementation of the division stage consists of a data path (see Fig.3) and a control path (see Fig.6). The data path implements various computational blocks such as a carry-save adder (CSA), a selection logic component (SLC), and multiplexers along with the necessary registers. The registers use latches rather than flip-flops. The control path generates the control signals for the registers in the data path at appropriate times. The control path also includes a counter that keeps track of the number of iterations and terminates the division operation at the appropriate time. The counter is an asynchronous down counter. Sections III-A and III-B provide the implementation details of the data path and the control path respectively.

A. *Data Path*

The data path for the divider appears in Figure 3. The cloud labeled `init` implements the following initialization statements (line 1) of algorithm H and generates the select control signals for the first iteration. The generation of select control signals is explained in section III-B.

```
1: rs:=R; rc:=0; qs:=0; qc:=0; c:=C;
```

In Figure 3, the register labelled `ENTER` receives the new operands, R, D, and C from `FIFO-A`. The `init` cloud executes the initialization statements and computes the select signals for the first iteration. The register `MERGE` is a 2:1 multiplexing register that selects the data from either the `init` cloud or from the register `TRUE`, depending on its input control signals. In each repetition step the data path computes the partial remainder and partial quotient for the next iteration. When the division operation terminates, the implementation sends the result of the division operation in carry-save form, qs and qc, to `FIFO-B` from register `FALSE` to register `EXIT`.

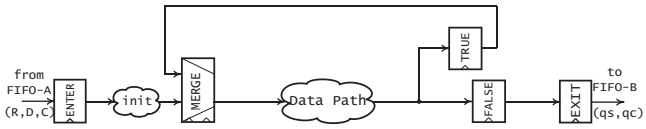

Fig. 3. Divider Data Path. The registers `ENTER` and `EXIT` serve to embed the divider in a pipeline.

The data path is segregated into a remainder data path and a quotient data path. The remainder data path implements the seven alternatives and the selection logic component (SLC).

The SLC determines the appropriate operation to execute on the partial remainder and the partial quotient. The quotient data path implements the actions performed on the partial quotient.

1) ***Remainder Data Path****:* Figure 4 shows an implementation of the remainder data path. The implementation includes one carry-save adder, indicated by CSA, multiplexers, labelled Mux A, Mux B, etc., the selection logic, indicated by SLC, and the shift operations of the algorithm indicated by ovals with label, 2X, 2X*, and 4X*. The ovals with label 2X* and 4X* implement left shift by 1 and 2 respectively followed by the inversion of most-significant bit of rs and rc. The oval with label 2X implements only a left shift by 1. Note that the only difference between 2X and 2X* is the inversion of the most-significant bit of rs and rc.

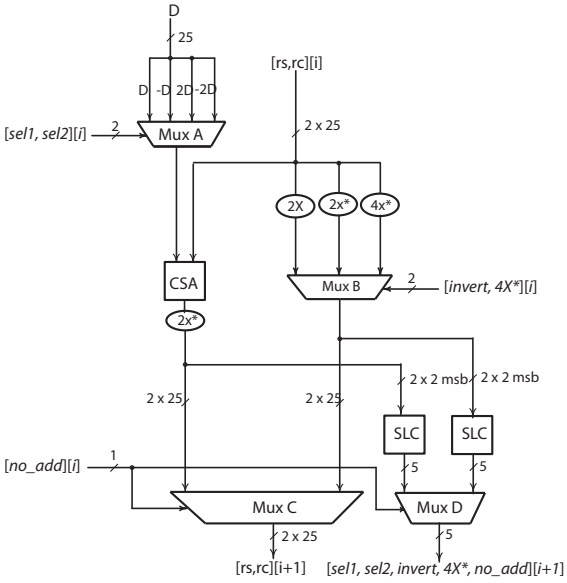

Fig. 4. An implementation of the Remainder Data Path.

The implementation computes one of the four addition operations and one of the three shift operations, and then selects the correct partial remainder for the next repetition. Computing one of the four addition operations involves selecting the correct multiple of the divisor D. In Figure 4, Mux A selects the appropriate multiple of D to perform one of the four additions followed by a 2X* operation, that is, ADD1& 2X*, SUB1 & 2X*, ADD2 & 2X*, or SUB2 & 2X*; Mux B selects one of three shift operations, that is 2X, 2X*, or 4X*; Multiplexers C and D choose the correct partial remainder and the select signals for the multiplexers for the next iteration respectively.

2) ***Selection Logic****:* The selection logic (SLC) determines which alternative to execute in the next repetition step. The selection logic takes the two most significant bits of the partial remainder in carry-save form, rs and rc, as input and produces five output signals; *sel1, sel2, invert, 4X*,* and *no_add*.

The signals *sel1* and *sel2* select the appropriate multiple of D for the carry-save addition. The combination of *invert* and *4X** select one of three shift operations: 2X, 2X*, and 4X*. The *no_add* signal selects the result from the addition or the result of Mux B. The SLC implements the following functions to generate corresponding signals for iteration *i+1*, where *i* is the iteration index. In the equations below, rs_0 and rs_1 are the most and second most significant bits of the partial sum respectively and rc_0 and rc_1 are the most and second most significant bits of the partial carry respectively. The symbols $\oplus$ denotes exclusive-OR operation.

$$4X^*_{i+1} = (rs_0 \oplus rc_0) \cdot (rs_1 \oplus rc_1) \quad (7)$$

$$invert_{i+1} = (rs_0 \cdot rs_1 \cdot rc_0 \cdot rs_1) + (\overline{rs_0} \cdot \overline{rs_1} \cdot \overline{rc_0} \cdot \overline{rc_1}) \quad (8)$$

$$no\_add_{i+1} = invert_{i+1} + (rs_0 \oplus rc_0) \quad (9)$$

$$sel2_{i+1} = \overline{(rs_1 \oplus rc_1)} \quad (10)$$

$$sel1_{i+1} = \overline{rs_0} \quad (11)$$

For the very first iteration the value of rc is 0 and the selection logic implements the following functions in the `init` cloud of Figure 3.

$$4X^*_1 = (rs_0) \cdot (rs_1) \quad (12)$$

$$invert_1 = (\overline{rs_0} \cdot \overline{rs_1}) \quad (13)$$

$$no\_add_1 = invert_1 + (rs_0) \quad (14)$$

$$sel2_1 = (rs_1) \quad (15)$$

$$sel1_1 = \overline{rs_0} \quad (16)$$

Tables 1(a) to 1(c) summarize various signals and the corresponding operation performed on the partial remainder.

invert	4X*	Type of *shift* operation
0	0	2X
0	1	Invalid
1	0	2X*
1	1	4X*

(a)

sel1	*sel2*	Type of *add* operation
0	0	ADD1
0	1	SUB1
1	0	ADD2
1	1	SUB2

(b)

no_add	Select result from
0	Output of CSA & 2X*
1	Output of Mux B

(c)

TABLE I
VARIOUS PARTIAL REMAINDER ACTIONS ASSOCIATED WITH THE SELECT SIGNALS FROM THE SLC

*3) **Quotient Data Path**:* The quotient data path appears in Figure 5. Its implementation is similar to the remainder data path implementation. Recall that in each repetition step, algorithm H performs one of five operations on the partial quotient along with right shifting the c operand by one or by two. Four out of five operations on the partial quotient are addition operations and they inversely correspond with the addition operations in the remainder data path; that is, when D is added to the partial remainder, c is subtracted from the partial quotient; when D is subtracted from the partial remainder, c is added to the partial quotient. However, when the remainder data path executes one of the three shift operations, the quotient accumulates one or two zeros by right shifting c by one or two respectively. Note that in this implementation, for values of c not equal to 1, the quotient is the result of both multiplication and division.

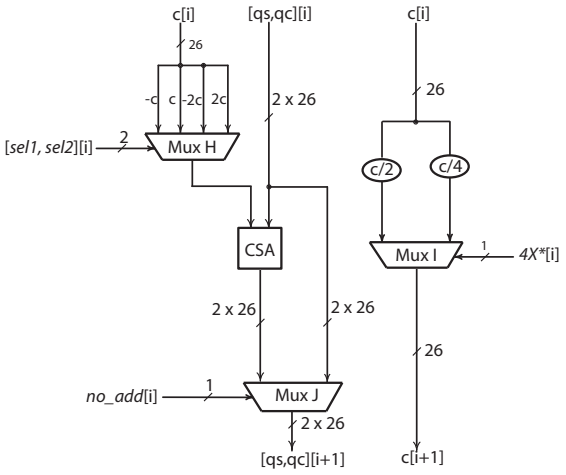

Fig. 5. A high-level implementation of Quotient Data Path

The quotient data path produces the quotient in carry-save form, which is a redundant representation. For further processing, the carry-save representation is usually converted to a unique binary representation. This conversion can be done by a carry-propagate adder, for example in the stage between the FALSE and EXIT registers. We have omitted this step in our implementation.

B. Control Path

Figure 6 shows the control path for the divider. The control path consists of GasP modules and an asynchronous down counter. The GasP modules were first introduced by Sutherland and Fairbanks in [10]. These modules generate pulses to enable the proper registers at appropriate times. We refer to these pulses as *fire* pulses. In Figure 6, the wires that connect two GasP modules are called state wires. For example, wires pred, enter and req_new are state wires. Whenever a GasP module *fires*, the module sets its output state wires HI and clears its input state wires LO. In this memo, we represent logic high as HI and logic low as LO.

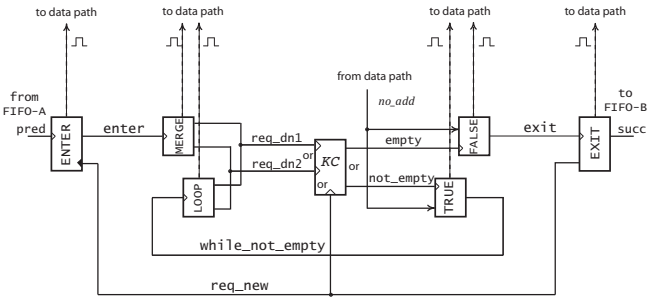

Fig. 6. Control path implementation for the divider

The pred and succ state wires connect GasP modules ENTER and EXIT with other GasP modules in FIFO-A and FIFO-B respectively. The ENTER module produces a fire pulse when both the pred and req_new state wires are HI. When the ENTER module fires, it sets the enter state wire HI and clears the pred and req_new state wires LO. When the enter state wire goes HI, the GasP module MERGE fires and sets either req_dn1 or req_dn2 state wire HI, requesting the down counter to decrement by 1 or 2 respectively. Furthermore, the MERGE module clears the enter state wire LO.

The box with label *KC*, implements the asynchronous down counter. The counter can decrement the count value by 1 or 2. It reports whether the count value is zero or not in a bounded response time of about 3.5 FO4 gate delays. The implementation of the down counter is a slight modification of the down counter presented in [11] . The req_new state wire going HI initializes the counter to a fixed value of $K+L+2$. The counter responds to decrement requests, either by setting the not_empty state wire HI, denoting a non-zero count value, or by setting the empty state wire HI, denoting a zero count value.

When the not_empty state wire is set HI, the GasP module TRUE fires. This makes the register labelled TRUE in the data path briefly transparent, and sets the while_not_empty state wire HI, requesting the GasP module LOOP to fire. The firing of the LOOP module enables the MERGE register in the data path to select the data from the TRUE register and sets either the req_dn1 or req_dn2 state wire HI, requesting the counter to decrement by 1 or 2 respectively.

When the counter sets the empty state wire HI, the GasP module FALSE fires. This makes the register labelled FALSE in the data path briefly transparent, and sets the exit state wire HI, requesting the GasP module EXIT to fire. The fire pulse from the EXIT module enables the EXIT register and sets the succ and req_new state wires HI. This allows FIFO-B to receive the computed result and makes the ENTER module ready to accept new operands from FIFO-A.

Figure 7 shows the complete top-level implementation of the divider. The dotted lines in the figure carry the *fire* signals.

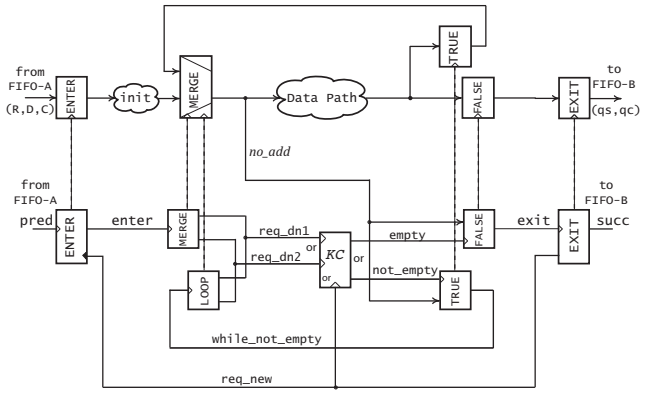

Fig. 7. Top-level schematic of the divider. The dotted lines carry the *fire* pulses.

C. *Variable Latency*

In the remainder data path, the latency of the *shift* path is smaller than the latency of the *add & shift* path. In our implementation, the control path exploits this disparity in the data path latencies by making the latencies of the GasP modules `TRUE` and `FALSE` data dependent. This means that the delay from the firing of the `MERGE` or `LOOP` module to the firing of the `TRUE` or `FALSE` module varies according to the *no_add* signal from the data path. As a result, the latency per iteration is a variable rather than fixed worst-case latency. Furthermore, because the counter can decrement by 1 or 2, our implementation also does a variable number of iterations per division.

IV. Results

We simulated Algorithm H in a TSMC 90nm process technology. We measured the delays on the critical paths from HSPICE simulations, where we normalize delays to fanout-of-4 (FO4) inverter delays. In TSMC 90nm technology one FO4 gate delay is 25ps. We used a custom static design style and the gates were sized using logical effort for equal delay rather than for the smallest possible path delay [12]. The simulation netlist included wire delay models. Part of the wire delay models used wire lengths from back annotated layout and the rest of the wire delay models used estimated wire lengths from a plausible floorplan.

In the implementation, the remainder data path is the critical path and it determines the latency per iteration. Table II provides the delay of various components used in the remainder data path of Figure 4. In our design, multiplexers C and D are built into the registers `TRUE` and `FALSE`, and hence we omit them from Table II. If not included in the registers, multiplexers C and D take 1 FO4 delay.

Components	Delay in FO4
Mux A, L_{MuxA}	2
Mux B, L_{MuxB}	2
CSA, L_{CSA}	2.5
SLC, L_{SLC}	4

TABLE II
Delay of various components in Figure 4

A. *Latency*

We compute average delay per quotient bit from average latency per iteration and average number of iterations per division. As discussed in the previous section, the latency per iteration varies because of the difference in the latencies of the add & shift and shift paths. Using the delay values from Table II, the *add&shift* latency $L_{add\&shift}$ is,

$$L_{add\&shift} = L_{MuxA} + L_{CSA} + L_{SLC} \\ = 2 + 2.5 + 4 = 8.5 \text{ FO4 } delays$$

and the *shift* path latency is,

$$L_{shift} = L_{MuxB} + L_{SLC} \\ = 2 + 4 = 6 \text{ FO4 } delays$$

The average latency per iteration, $L_{avg/iter}$ is,

$$L_{avg/iter} = (P_{add} \cdot L_{add\&shift} + P_{shift} \cdot L_{shift})$$

where, P_{add} is the probability of choosing the result from the carry save adder and P_{shift} is the probability of choosing the result from Mux B. For algorithm H, P_{add} and P_{shift} are approximately equal to 0.5. Therefore, $L_{avg/iter}$ is 7.25 FO4 delays. Note that this number excludes the sequencing overhead per iteration $T_{seq/iter}$, which is 7.5 FO4 delays for our implementation. The sequencing overhead is the delay incurred in the registers.

In [7], Harris, Oberman, and Horowitz compare various SRT implementation schemes for delay per quotient bit and area per bit per cycle. The non-overlapped radix-4 SRT implementation in [7] is similar in architecture and circuit family to our data path implementation technique. The delay per quotient bit for this implementation is 9.5 FO4 delays [7]. This number excludes sequencing overhead per iteration. In our implementation we compute the average delay per quotient bit, $D_{avg/bit}$ as follows, taking into account that the number of iterations per division varies and the latency per iteration varies.

$$D_{avg/bit} = (L_{avg/iter} \cdot N_{avg}) \ / \ Number\ of\ quotient\ bits$$

where, N_{avg} is the average number of iterations per division, which is 22.6 for algorithm H, $L_{avg/iter}$ is 7.25 FO4 delays, and the number of quotient bits accumulated

is 26. Therefore, $D_{avg/bit}$ = 6.30 FO4 delays. This number also excludes sequencing overhead. Even, if we include a delay of 1FO4 for multiplexers C and D, which are hidden in registers, our design compares favorable to the 9.5 FO4 delays of other static designs.

Ted Williams's work in [8] reports on several divider architectures which all use domino circuits. Domino circuits occupy more area but can be 1.5-1.7x faster than the static circuits that we use. Williams's architecture with a non-overlapping quotient selection logic, like ours, has a delay of 16.8 FO1. This corresponds to a delay between 5.6 to 6.72 FO4 per quotient bit assuming 1 FO4 is between 3 to 2.5 FO1 [7][12]. Because Williams's design uses domino circuits and a ring of 5 stages, in order to hide the sequencing overhead, the area overhead for this design is very large. Renaudin's work presented in [13] uses a similar idea as Williams, but instead of domino circuits [13] uses LDCVSL circuits and rings of only three stages. The delay per quotient bit for [13] is about 16 FO4.

Like average latency per iteration, we can also calculate the average latency per division, $T_{avg/div}$ for our implementation as follows:

$$T_{avg/div} = (L_{avg/iter} + T_{seq/iter}) \cdot N_{avg} + L_{f,ENTER} + L_{f,EXIT}$$

where, $L_{f,ENTER}$ and $L_{f,EXIT}$ are the forward latencies of GasP modules `ENTER` and `EXIT`. In our design $L_{f,ENTER}$ and $L_{f,EXIT}$ are 6 FO4 delays each. The GasP modules `MERGE`, `LOOP`, `TRUE` and `FALSE` are designed such that the latency per iteration in the control path matches the latency per iteration plus the sequence overhead in the data path. Therefore, the average latency per division, $T_{avg/div}$ is approximately 345 FO4 delays. Table III summarizes the best-case, average-case and worst-case delays per quotient bit and latency per division. For the best-case, $L_{best/iter}$ is 6.75 FO4 delays because the 4X* alternative is executed on the partial remainder and the algorithm takes 13 iterations to complete. For the worst-case, the algorithm takes 26 iterations to complete and all the iterations are one of the add&shift operations with $L_{worst/iter}$ being 8.5 gate delays.

Cases	Delay per bit without $T_{seq/iter}$ (FO4)	Latency per division (FO4)	Delay per bit with $T_{seq/iter}$ (FO4)
Best-case, N = 13	3	187.5	6.75
Worst-case, N = 26	8.5	428	16
Average-case, N = 22.6	6.30	345	12.82

TABLE III
EXPECTED DELAY PER QUOTIENT BIT AND LATENCY PER DIVISION FOR DIFFERENT NUMBER OF ITERATIONS N OF ALGORITHM H

The measured average latency per division from 50 simulations was approximately 330 FO4 delays. This is different from the expected result for following reasons:

- For the 50 simulations, the average number of iterations per division was 22.2 instead of 22.6.
- The average number of additions per division was 40% rather than 48%.
- $L_{add\&shift} + T_{seq/iter}$ was 15.8 FO4 delays instead of the expected 16 (=8.5+7.5) FO4 delays.

If Algorithm H was implemented as a synchronous divider rather than as an asynchronous divider, we expect the latency per division and delay per quotient bit to increase. This is because the synchronous implementation fails to exploit the variable latency per iteration. However, the synchronous implementation can certainly take advantage of the variable number of iterations per division feature of algorithm H. The latency per iteration for a synchronous implementation will be equal to the latency of the critical path which is the latency of the add & shift path. Therefore, the latency per iteration L_{iter} in a synchronous implementation will be:

$$L_{iter} = L_{add\&shift} = 8.5 \text{ FO4 } delays$$

Using L_{iter} we can compute delay per quotient bit and latency per division for a synchronous implementation. Table IV summarizes the best-case, average-case and worst-case delays per quotient bit and latency per division for a synchronous implementation.

Cases	Delay per bit without $T_{seq/iter}$ (FO4)	Latency per division (FO4)	Delay per bit with $T_{seq/iter}$ (FO4)
Best-case, N = 13	4.25	220	8
Worst-case, N = 26	8.5	428	16
Average-case, N = 22.6	7.39	373.6	13.90

TABLE IV
EXPECTED DELAY PER QUOTIENT BIT AND LATENCY PER DIVISION FOR DIFFERENT NUMBER OF ITERATIONS N OF ALGORITHM H (SYNCHRONOUS IMPLEMENTATION)

In [14] Liu and Nannarelli report that the latency per division of 24-bit operands using a standard radix-4 SRT implementation in STM 90nm technology is 13 ns, which translates to 520 FO4 gate delays per division. Although this number includes converting redundant representation to binary representation and final rounding, it is significantly higher than the expected result of a synchronous implementation and the simulated result of our asynchronous implementation of Algorithm H.

B. Energy

We applied random input test vectors to the divider circuit in 50 simulations and used NanoSim to calculate the average energy per division. The average energy per division of 25-bit

operands for our implementation in a 90nm CMOS technology is approximately 182pJ. Renaudin et al. [13] report an average energy per division of 32-bit operands in a 0.5μm CMOS technology of 3nJ. The energy per division of 24-bit operands using a standard radix-4 SRT [14] implementation in STM 90nm technology is 112.5 pJ. Liu and Nannarelli [14] also present a low-power implementation of the radix-4 SRT divider which consumes 93.6 pJ per division.

V. Conclusion

This paper presents an example of exploiting the average-case behavior of asynchronous circuits. Our example is an asynchronous implementation of a multiply-divide circuit, which on average takes less delay per quotient bit than a similar SRT divider implementation. There are two reasons for this improvement. First, Algorithm H, on average, takes fewer iterations per division than radix-2 SRT algorithm, which both asynchronous and synchronous design styles can exploit. Second, each iteration has variable latency, which only asynchronous circuits can make use of. Although this paper uses GasP circuits, other asynchronous circuit design styles such as Delay-Insensitive (DI) and Quasi-Delay Insensitive (QDI) circuits [15] can also be used to implement Algorithm H.

Our main goal was to design a simple, straightforward implementation using static logic. Obviously, there are several ways to improve our implementation. First, the data path can be optimized by exploiting a more parallel architecture like an overlapping digit selection logic. We expect that this improvement will reduce the average latency per iteration at a small cost of extra area and energy consumption. Second, the sequencing overhead per iteration can be reduced. The current implementation has a sequencing overhead per iteration of 7.5 FO4, which is large when compared to the average latency per iteration of 7.25 FO4. Because the feedback path in our repetition cycle performs no computation, inserting another data computation in the feedback path will mitigate the effect of the sequencing overhead. Such an implementation will produce at least two quotient digits per iteration for the same sequencing overhead. Third, we can try to hide the sequencing overhead completely by using domino circuits and a self-timed ring, like Williams's design. But this solution comes at the cost of much more area and energy consumption.

If we want to implement a division only, i.e., $C = 1$, we can further reduce the energy consumption by simplifying the quotient data path. The quotient data path then implements an addition of -2, -1, 0, 1, or 2 and can do an on-the-fly conversion of the redundant quotient into a unique binary representation at the same time [9].

Acknowledgment

We thank Ivan Sutherland for his contributions to this work. This work was partially sponsored by the Defense Advanced Research Projects Agency (DARPA) under grant number HR0011-10-1-0069 for the technical proposal "Fleet Studies" and partially sponsored by Oracle Labs.

References

[1] J. Ebergen, I. Sutherland, and D. Cohen, "Method and apparatus for performing a carry-save division operation," U.S. Patent 7 660 842, February 9, 2010.

[2] S. F. Oberman and M. J. Flynn, "Design Issues in Division and Other Floating-Point Operations," *IEEE Trans. Comput.*, vol. 46, pp. 154–161, February 1997. [Online]. Available: http://dx.doi.org/10.1109/12.565590

[3] R. E. Goldschmidt, "Applications of Division by Convergence," Master's thesis, Massachusetts Institute of Technology, 1964.

[4] J. E. Robertson, "A New Class of Digital Division Methods," *Electronic Computers, IRE Transactions on*, vol. EC-7, no. 3, pp. 218 –222, sept. 1958.

[5] M. Ercegovac and J. Muller, "Digit-recurrence algorithms for division and square root with limited precision primitives," in *Signals, Systems and Computers, 2003. Conference Record of the Thirty-Seventh Asilomar Conference on*, vol. 2, nov. 2003, pp. 1440 – 1444 Vol.2.

[6] J. Ebergen, I. Sutherland, and A. Chakraborty, "New division algorithms by digit recurrence," in *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Eighth Asilomar Conference on*, vol. 2, nov. 2004, pp. 1849 – 1855 Vol.2.

[7] D. L. Harris, S. F. Oberman, and M. A. Horowitz, "SRT Division Architectures and Implementations," in *Proceedings of the 13th Symposium on Computer Arithmetic (ARITH '97)*, ser. ARITH '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 18–25. [Online]. Available: http://dl.acm.org/citation.cfm?id=786448.786534

[8] T. Williams and M. Horowitz, "A Zero-Overhead Self-Timed 160-ns 54-b CMOS Divider," *Solid-State Circuits, IEEE Journal of*, vol. 26, no. 11, pp. 1651 –1661, nov 1991.

[9] S. F. Oberman and M. J. Flynn, "Division Algorithms and Implementations," *IEEE Trans. Comput.*, vol. 46, pp. 833–854, August 1997. [Online]. Available: http://dx.doi.org/10.1109/12.609274

[10] I. Sutherland and S. Fairbanks, "GasP: a minimal FIFO control," in *Asynchronous Circuits and Systems, 2001. ASYNC 2001. Seventh International Symposium on*, 2001, pp. 46 –53.

[11] J. Ebergen and A. Megacz, "Asynchronous Loadable Down Counter," U.S. Patent 8 027 425, September 27, 2011.

[12] Sutherland, Ivan and Sproull, Bob and Harris, David, *Logical effort: designing fast CMOS circuits*. Morgan Kaufmann Publishers Inc., 1999.

[13] M. Renaudin, B. Hassan, and A. Guyot, "A new asynchronous pipeline scheme: application to the design of a self-timed ring divider," *Solid-State Circuits, IEEE Journal of*, vol. 31, no. 7, pp. 1001 –1013, jul 1996.

[14] W. Liu and A. Nannarelli, "Power dissipation in division," in *Signals, Systems and Computers, 2008 42nd Asilomar Conference on*, oct. 2008, pp. 1790 –1794.

[15] P. Beerel, M. Ferretti, and R. Ozdag, *A Designer's Guide to Asynchronous VLSI*. Cambridge University Press, 2010.