Self-generating clock using an augmented distribution network

Gerard M Blair

Department of Electrical Engineering
The University of Edinburgh
The King's Buildings
Edinburgh, EH9 3JL
Scotland, UK
tel: UK+131 650 5592
fax: UK+131 650 6554
email: gerard@ee.ed.ac.uk

ABSTRACT

VLSI sub-modules may be designed in a synchronous style but with a local clock generated using its own distribution network as a ring oscillator. If a long critical path exists, a delay can be added to the oscillator path by implementing the critical path (alone) in self-timed logic. This technique has many of the advantages of self-timed logic with lower circuit overheads and a conventional design style. The technique is illustrated by a self-timed square-root module.

1. Introduction

The reduction in the feature size of silicon fabrication technology leads to faster transistors but slower interconnect lines. Thus the distribution of signals within a component becomes relatively slow and best performance is achieved if the majority of communication is performed within small regions of the component. Applied to the flow of data, this implies local processing and storage of information; applied to the flow of control and clocking signals, this implies that the traditional global-clocking scheme must be replaced. We need to develop new design disciplines based upon semi-autonomous sub-modules with either local clock generation or asynchronous logic.

The Stanford pipelined iterative multiplier (SPIM) published by Santoro and Horowitz[1] uses a local clock generator: simple ring-oscillator whose feed-back path is controlled by external signals to include different numbers of buffer delays. This allows the clock to be digitally tuned according to the critical path delay. The clock signal is generated for the number of cycles needed to perform the (possibly pipelined) multiplications. The general clock architecture is illustrated in Figure 1: the module is seen as a collection combinatorial logic separated by state elements driven by a synchronous clock signal generated by an independent ring oscillator.

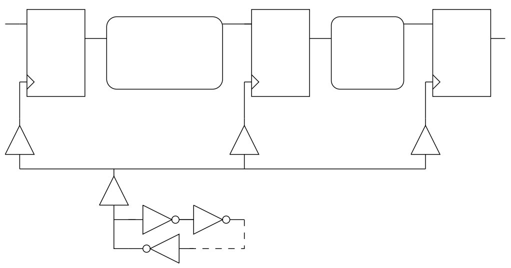

Figure 1: Clock generated by independent ring-oscillator

A different approach is to use asynchronous or self-timed logic. Each stage is enhanced to produce a "completion" signal and to generate its own "clock" from the completion signals of all stages which consume its outputs. The computation thus proceeds by hand-shaking between adjacent logic functions: there is no clock, rather a sequence of local control signals. This is illustrated in figure 2. An example of a circuit using this approach is the self-timed CMOS divider published by Williams and Horowitz[2] in 1991 which forms a ring of such elements to implement the iterative operations for the division.

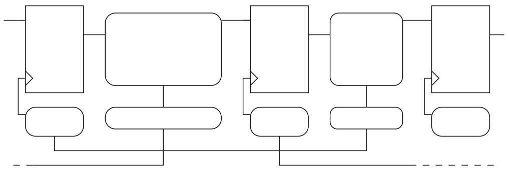

Figure 2: Self-timed or asynchronous logic

Instead of using an independent ring-oscillator as in the SPIM design, it is possible to use the local clock-distribution network as the oscillator. This was used in 1991 by the author to implement a bit-serial correlator[3]. In this design the clock is distributed to different sub-modules in the reverse order to the data-flow, which reduces power spikes in comparison to purely synchronous schemes. The clock architecture is illustrated in Figure 3. The advantages of forming the ring-oscillator from the clock distribution network are that the circuit overhead is low and the oscillator noise is only that associated with the clock distribution. The disadvantages are that the logic must have a critical path delay which is less than the period of the clock distribution network, and that the clock circuit must still be designed with a margin to avoid race hazards with the critical path delay.

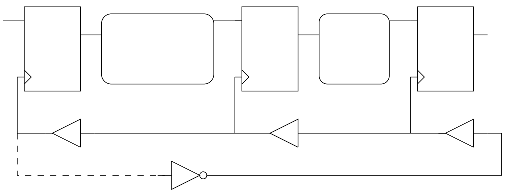

Figure 3: Ring-oscillator formed by clock distribution network

All three designs have a similar interface to the rest of the system. The module is activated by a *Start* signal which indicates that input data is available, the internal clock or hand-shaking protocol is initiated, and when the computation is complete the activity within the module stops and a *Done* signal is raised. This supports low-power operation since power is consumed only during the computation, and the logic and clock are suspended at other times.

In section 2, we present a new clock generation architecture, illustrated by a component design introduced in section 3. Section 4 looks more closely at the clock generation circuitry and the results of the fabricated test component are given in section 5.

2. Augmented clock-generation architecture

The limiting factor in clock frequency is the delay of the critical path: the longest delay between adjacent latches. In a module with a clearly identifiable critical path, we can use the critical path itself to define the clock rate. Thus the basic idea is to include completion logic on *only* the critical path and to use that as part of the clock-generation oscillator. The remainder of the circuit (including all the state-elements) is designed as a conventional synchronous module.

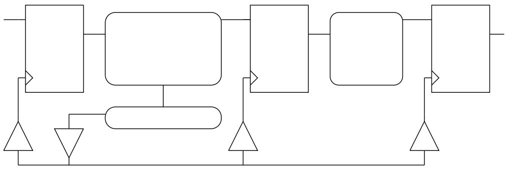

Figure 4: Clock generated by distribution network and completion logic on critical path

The architecture is illustrated in figure 4. The oscillator is formed using the clock distribution network which is driven by the output of the completion logic for the critical path. Unlike the first architecture in section 1, the clock frequency is automatically determined; unlike the second architecture, the logic overheads are low; unlike the third architecture, long critical paths can be accommodated. On the other hand, the advantages new architecture combine many of those of the architectures in section 1:

- the local clock speed is independent of the circuits external to the module and automatically runs with the highest frequency possible for the given circuit
- the internal clock speed tracks the process and environmental variations of the component (as with asynchronous logic)
- circuit overheads are limited to the hand-shake interface and the inclusion of completion logic on the critical path only
- for the most part, well-known (synchronous) techniques can be used in the module design

3. The square-root algorithm

We now consider an example design. The objective is to implement an independent square-root function with minimal logic overhead beyond conventional synchronous design using the clock generation architecture described in section 2. Thus we will use an architecture which includes a full carry-propagate chain (as the critical path) rather than a faster, though larger, circuit using "SRT" techniques as in [2].

The square-root function is a natural function in digital signal processing algorithms but is less commonly used due to the complexity of its implementation. With special purpose processors, there is commonly an efficient multiply function and the square-root can be implemented, for example, by modifying the Newton-Raphson algorithm as a nonlinear digital filter[4]. For systems without fast multiplication hardware, algorithms have been developed using only shift, add and (sometimes) table look-up. We consider an implementation based on a 1962 paper by Meggitt[5] which uses only shift and subtraction.

Given a positive integer S, we seek an integer R such that $(R+1)^2 > S \geq R^2$. If S can be expressed in a fixed field of s-bits, then R can be expressed in *(s+1)/2* -bits. The algorithm proceeds by successive approximation in which the bits of an estimate for the square-root are set to "1" in turn (starting with the most-significant-bit) and reset if the square of the new estimate is greater than S. If X is the current estimate and Y is 2^i where i is the index of the bit under test, then the new estimate is $(X + Y)$ and the test condition may be rewritten as:

$$S < (X+Y)^2 = X^2 + 2XY + Y^2 \qquad (1)$$

or equivalently as

$$REM \;=\; S - X^2 < Y(2X+Y) \qquad (2)$$

where REM is the current remainder (REM): the difference between S and the square of the current estimate. If equation 2 is true then the estimate $(X+Y)$ is too large and so rejected; if false then the estimate is updated and the remainder is decreased by $Y(2X+Y)$. The algorithm (for an 8-bit square-root) can be expressed in C as:

```
x = 0;
rem = S;
for (y = 1 << 7; y > 0; y = y >> 1) {
  diff = rem − y * (2 * x + y);
  if (diff > 0) {
    rem = diff;
    x = x + y; } }
```

We avoid multiplication since Y is a power of two and so the multiplications in $Y(2X+Y)$ can be seen as shifts or simply bit-alignments. Furthermore, we know by the manner in which X is calculated that all bits which are less significant than, and including, the current test bit are zero; therefore $Y(2X+Y)$ has the pattern:

{known bits of X, 0, 1, zero or more 0s}

The algorithm can also be expressed in Verilog (as shown below for an 8-bit root).

```
input [15:0] S;
reg   [ 7:0] x;
reg   [23:0] rem;
wire  [10:0] diff;

assign  diff =  rem[23:14] - {x, 2'b01};

initial begin
  x   = 0;
  rem = S;

  repeat (9)
    #1
    if (diff[10] == 1) begin // if diff negative
      rem <= {rem[21:0], 2'b00};
      x   <= {x[6:0], 1'b0};
      end
    else begin
      rem <= {diff[7:0], rem[13:0], 2'b00};
      x   <= {x[6:0], 1'b1};
      end
  end
```

To perform the necessary alignments of the operands for the subtraction to calculate *diff*, we define two shift register:

x: (8-bits) contains only the known bits of X right-aligned (initially zero). This shifts left one-bit each cycle with a "1" or "0" inserted according to the results of equation 2.

rem: (24 bits) contains the remainder (initially S). For alignment, this shifts left two-bits each cycle (once to compensate for the shift of x and once because Y in equation 2 causes one less shift to $(2X+Y)$ each cycle). Its value updates to either the value (REM) of the previous cycle, or the difference $REM - Y(2X + Y)$, shifted left by two bits.

Notice that due to the pattern of $Y(2X + Y)$, the subtraction forming *diff* does not involve the lower bits of *rem* (from which 0s are subtracted); thus *diff* is implemented by a 10-bit subtracter. The final estimate is correctly aligned in x while the remainder is $rem[23:16]$.

4. Clock generation

The critical path of this design is the carry-propagation in the subtraction forming *diff*, which is implemented in Domino Logic[6] with the complementary signals *carry* and $\overline{carry}$ generated separately according to the formulae:

$$carry = a.b.\bar{c} + a.c + b.c \qquad (3)$$

$$\overline{carry} = \bar{a}.\bar{b}.c + \bar{a}.\bar{c} + \bar{b}.\bar{c} \qquad (4)$$

where c and $\bar{c}$ are the *carry* and $\overline{carry}$ outputs from the previous stage of the carry chain. The use of complementary Domino Logic functions in self-timed logic can be found in [2]. During precharge phase, the output of both functions is HIGH; when the function is resolved, one of the outputs becomes LOW. Thus the NOR of the complementary outputs provides a completion signal for the function.

Generally, in asynchronous circuits, this completion circuit is used for single function gates; we extend its use to the whole carry-chain and so also take advantage of Domino Logic's high-speed. Because c or $\bar{c}$ are part of each product term in equations (3) and (4), each stage can only resolve after either the *carry* or $\overline{carry}$ output from the previous stage has become LOW. Essentially, the complementary functions of each bit are themselves completion logic for the preceding stages; and the completion of the whole carry-propagation is monitored by a single NOR gate on the complementary outputs of the most-significant bit.

In general, there are two drawbacks to Domino Logic: first, no logic evaluation occurs during the precharge phase; second, the precharging of a LOW output consumes power even if the output does not change in value. However, in this design the domino logic is used to generate the clock which results in the smallest practical precharge phase: it represents a small proportion of the cycle time and this is offset by the higher speed of Domino Logic compared to static logic. As to power consumption, the Domino Logic is used only on the critical path yet enables power saving by allowing the clock is be stopped in the whole module when the function is not being used.

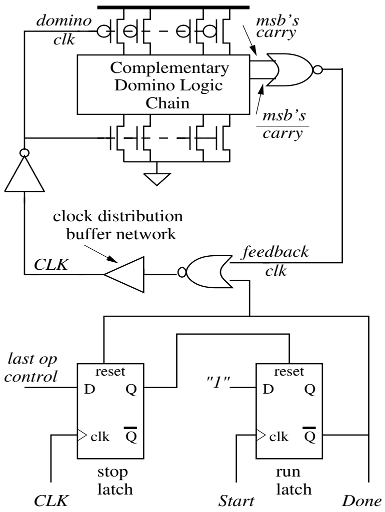

Figure 5: Clock generation and interface logic

The full clock generation and interface logic is shown in figure 5. The square-root operation is initiated by a rising edge of the *Start* signal which clocks a HIGH into the *run latch* and so connects (through the NOR gate) the ring oscillator path. On the first rising (internal) clock edge, the shift-registers are initialized; thereafter *CLK* oscillates until, on the penultimate cycle, the *last op control* line is raised. Then, on the next rising *CLK* edge, the *stop latch* is set HIGH which in turn resets the *run latch* raising its $\bar{Q}$ signal to break the ring-oscillator path and to reset the *stop latch.* The $\bar{Q}$ signal also provides the *Done* signal for the handshaking protocol. The internal control signals are generated from a state machine implemented by a simple shift-register.

Figure 6 shows an HSPICE simulation performed on the extracted layout to demonstrate the internal clock signal. The results in the upper graph show the *Start* signal rising followed by nine internally generated *CLK* pulses (with an asymmetric duty cycle due to the critical-path delay). The lower graph shows the *Done* (*READY*) signal and the output of the first bit of the X (*SQRT*) register as the result is shifted through.

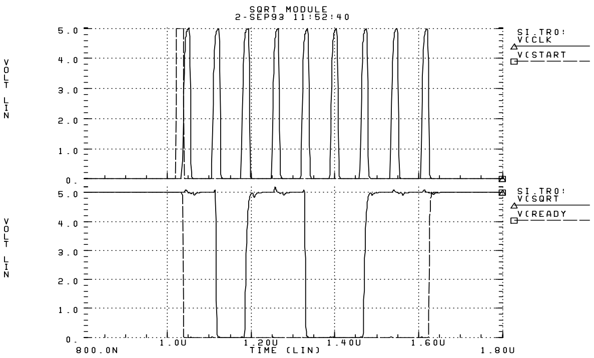

Figure 6: HSPICE simulation

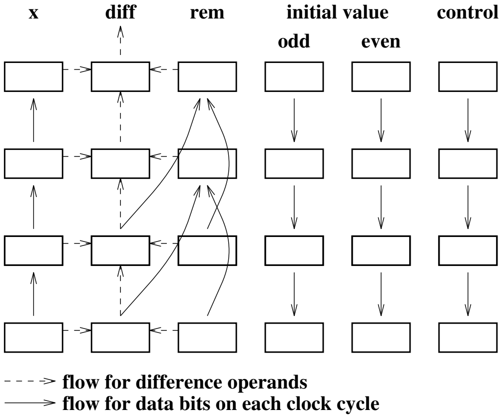

Figure 8: Circuit floor plan

5. Test component

To validate this design and clocking technique, a module which transforms a 16-bit input to its 8-bit square-root has been designed (see figure 7) in a 1.5μ 2-metal CMOS (ES2's ECPD15) process with an active area of $1mm^2$ (1.06mm * 0.94mm). The fabricated component has been tested and performs a square-root on a 16-bit number in 280ns.

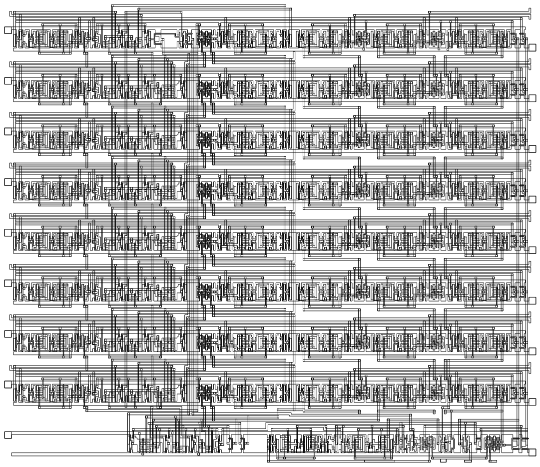

Figure 7: Circuit layout

The circuit is laid out as repeated rows as shown in figure 8. The control logic (which counts the number of iterations) is implemented as a simple shift register and so the design can be expanded to a different word size by adding extra rows. The lower 16-bits of the *REM* register use two shift registers in parallel (for odd and even bits). The bottom row of figure 7 (not shown in figure 8) contains the extra logic to achieve the handshaking protocol, the clock generation, and the initial two bits of the subtracter.

The implementation of the square-root function with full carry-propagation was chosen to illustrate the clock design: larger, though faster, architectures using SRT techniques were not used in favour of the simpler design. However, a recent paper by Matsubara *et al*[7] reports a self-timed 55-bit square-root module which functions in 30ns at 0.3μ technology: with adjustments for bit size and processing technology (with delay scaling $\propto$ length$^{1.5}$), this is only 2-3 time faster than the design reported here.

6. Conclusions

We have demonstrated a self-timing architecture which combines a self-generated clock with the combinatorial logic of a module's critical path. Because the clock and the critical path overlap, the design problem of matching their delays to avoid synchronization failure is avoided. The technique allows the majority of the design to be undertaken with traditional synchronous design techniques: only the critical path needs to be modified to produce a resolution-complete signal.

References

1. M R Santoro and M A Horowitz, "SPIM: A pipelined 64x64-bit iterative multiplier", *IEEE Journal of Solid-State Circuits*, vol. 24, no. 2, pp. 487-493, Apr 1989.

2. T E Williams and M A Horowitz, "A Zero-Overhead Self-Timed 160-ns 54-b CMOS Divider", *IEEE Journal of Solid-State Circuits*, vol. 26, no. 11, pp.

1651-1661, Nov 1991.

3. G.M. Blair, ‘‘Bit-serial Correlator with Novel Clocking Scheme’’, *Proc ESSCIRC*, pp. 157-160, 1991.

4. N Mikami, M Kobayashi, and Y Yokoyama, ‘‘A New DSP-Orientated Algorithm for Calculation of the Square Root Using a Nonlinear Digital Filter’’, *IEEE Trans on Signal Processing*, vol. 40, no. 7, pp. 1663-9, July 1992.

5. J E Meggitt, ‘‘Pseudo Division and Pseudo Multiplication Processes’’, *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 210-26, April 1962.

6. R H Krambeck, C M Lee, and H-F S Law, ‘‘High-Speed Compact Circuits with CMOS’’, *IEEE Journal of Solid-State Circuits*, vol. SC-17, no. 3, pp. 614-619, June 1982.

7. G Matsubara, N Ide, H Tago, S Suzuki, and N Goto, ‘‘30-ns 55-b shared radix 2 division and square root using self-timed circuit’’, *Proc 12th Symposium on Computer Arithmetic*, pp. 98-105, 1995.