An Algorithm for Nano-pipelining of Circuits and Architectures for a Nanotechnology

Pallav Gupta and Niraj K. Jha
Department of Electrical Engineering
Princeton University
Princeton, NJ 08544
{pgupta, jha}@ee.princeton.edu

***Abstract*— In this paper, we describe an algorithm to post-process a register-transfer level (RTL) architecture to enable gate-level pipelining or nano-pipelining for the nanotechnology based on resonant tunneling diodes (RTDs). Nano-pipelining offers the opportunity to obtain massive throughput and, therefore, has applications in data-intensive algorithms such as digital signal processing (DSP). Since RTDs are a self-latching nanotechnology, nano-pipelining is an implicit property that should be exploited for this technology. The novelty of this work lies in exploring and demonstrating the benefits of nano-pipelining and presenting an algorithm for architectural nano-pipelining.**

I. Introduction

In accordance with Moore's Law, complementary metal-oxide semiconductor (CMOS) chips have continued to provide the high degree of integration required to sustain the ever-increasing high-performance computing needs of end users. As predicted by the Semiconductor Industries Association (SIA) roadmap, this trend is likely to continue for another 10-15 years [1]. However, developments in the material science and device community have enabled the creation of nanoscale devices that offer the possibility of improving digital systems further in terms of area and speed. Consequently, research is being done to understand the properties of these devices to see how they can be used in novel circuits and architectures [2]–[5].

There are three main reasons why RTDs are currently the most promising nanotechnology to make their way into mainstream electronics. First, RTDs can be grown with great precision and uniformity using molecular-beam epitaxy [2]. Second, logic circuits containing RTDs and heterostructure field-effect transistors (HFETs) have been demonstrated to work at high clock frequencies [3], [6], [7]. Finally, RTDs implement threshold logic which provides improved computational functionality through smaller circuit logic depth, fewer devices, and shorter wiring [8].

While progress in the material and physical understanding of RTDs continues to be made, there is a need for work at the logic and architectural levels to fully harness the potential of RTDs. The authors in [2] describe an RTD-FET logic family that is expected to operate five times faster than circuits utilizing III-V FETs alone. In [3], the monostable-bistable transition logic element (MOBILE) is introduced which is a self-latching RTD-HFET threshold gate. Because of the MOBILE's self-latching capability, computation can be nano-pipelined. The authors in [7]–[9] take advantage of this fact to present nano-pipelined designs of a digital correlator and various adder circuits. However, there is no work to date on exploiting this level of computation granularity at the architecture level.

The goal of this work is to present an algorithm for nano-pipelining of an architecture. The novel contributions of this work are as follows:

- This is the first algorithm for architectural nano-pipelining.
- We present a case for further research in nano-pipelined architectures by demonstrating its benefits.
- We present nano-pipelined designs of a few arithmetic circuits and perform a theoretical analysis of their gate count and latency characteristics.

The remainder of this paper is organized as follows. We discuss some preliminaries in Section II. In Section III, we present a motivational example to demonstrate the benefits of architectural nano-pipelining. We describe our nano-pipelining algorithm in Section IV and discuss nano-pipelined designs of some arithmetic circuits in Section V. Section VI provides some insights into open research areas in nano-pipelining. Our conclusions are presented in Section VII.

II. Preliminaries

In this section, we present some background material which will be helpful in understanding the ideas presented in the remainder of this paper.

A. Threshold Logic

A linear threshold function is a multi-input function in which each input, $x_i \in \{0,1\}$, $i \in \{1,2,\ldots,n\}$, is assigned a weight, w_i, such that the function assumes a value of 1, if and only if the weighted sum of the inputs is greater than or equal to the function's threshold, T. That is,

$$f = \begin{cases} 1 & \text{if } \mathbb{X} \geq T \\ 0 & \text{if } \mathbb{X} < T \end{cases} \tag{1}$$

$$\mathbb{X} = \sum_{x=1}^{n} w_i x_i. \tag{2}$$

In theory, the only restriction on the weights and threshold is that they must be real-valued. In practice, however, they are integer-valued and constrained within a finite range due to technological considerations. Therefore, $w_i \in \{0, \pm1, \ldots, \pm w_{max}\}$ and $T \in \{0, \pm1, \ldots, \pm T_{max}\}$. We will use the weight-threshold vector $\langle w_1, w_2, \ldots, w_n; T\rangle$ to denote the weights and threshold of the threshold function.

B. Monostable-Bistable Transition Logic Element (MOBILE)

The authors in [3] have demonstrated that a MOBILE can operate at frequencies of several GHz. Shown in Fig. 1, a MOBILE is a rising edge-triggered, current-controlled gate that implements a threshold function. It consists of a serially connected load and driver RTD. The RTD-HFETs connected in parallel to the load and driver RTDs perform a positive and negative weighting of the inputs, respectively.

Acknowledgments: This work was supported in part by NSF under grant No. CCR-0303789.

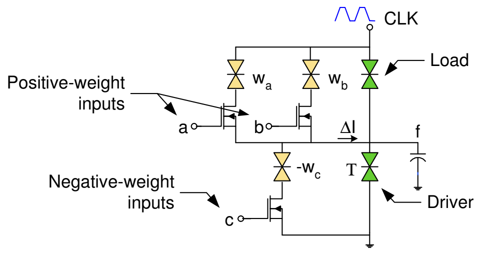

Fig. 1. A monostable-bistable transition logic element (MOBILE).

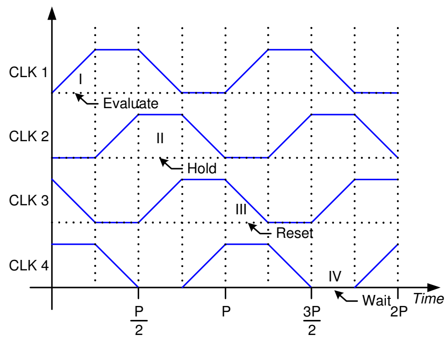

Fig. 2. Four-phase overlapping clocking scheme for MOBILE circuits.

1) Device characteristics: A MOBILE switches to the monostable or bistable mode of operation depending upon its bias voltage, V_{CLK}, which oscillates between 0V and V_{DD} [9]. That is,

$$V_{OUT} = \begin{cases} monostable & \text{if } V_{CLK} < 2V_P \\ metastable & \text{if } V_{CLK} \approx 2V_P \\ bistable & \text{if } V_{CLK} > 2V_P, \end{cases} \quad (3)$$

where V_{OUT} and V_P are the MOBILE output voltage and the RTD peak voltage, respectively. A typical value for V_P is 0.3V [7]. The metastable state is achieved when $V_{CLK} > 0.55$V [9] and the digital states are achieved when a MOBILE is in the bistable mode of operation.

When the device is in the metastable region, the modulation current, ΔI, at the output node determines what digital state the device transitions to as it enters the bistable region. The modulation current is obtained from Kirchoff's Current Law and is given as,

$$\Delta I = \sum_{i=1}^{N_p} w_i I(V_{gs}) - \sum_{i=1}^{N_n} w_i I(V_{gs}), \quad (4)$$

where N_p and N_n are the number of positive and negative weighted inputs, respectively, and $I(V_{gs})$ is the peak current of a minimum-size RTD [9]. If the gate-to-source voltage of an input, x_i, exceeds the threshold voltage of the HFET, the current $w_i I(V_{gs})$ is supplied either to the load or driver RTD current. The net RTD current for the load and driver is $I_T = TI(V_{gs})$, respectively. Consequently, V_{OUT} is logic high if $\Delta I - I_T$ is positive and logic low otherwise.

2) Four-phase clocking scheme: A MOBILE is self-latching because its output is valid only when the clock is high. This property can be exploited to achieve a nano-pipeline by constructing a cascaded network of MOBILEs. However, a four-phase overlapping clocking scheme, shown in Fig. 2, is required to ensure correct operation [9]. In this scheme, each stage of the clock has an equal period of $\frac{P}{4}$ and a 90° phase delay. During the *evaluation* phase, the output of a gate is computed. In the *hold* (i.e., self-latching) phase, the result is valid while the subsequent pipeline stage begins its computation. In the *reset* phase, the load capacitance is discharged

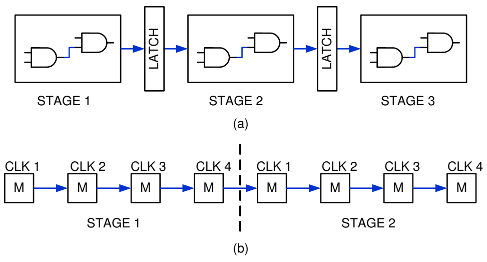

Fig. 3. (a) Conventional pipeline, and (b) nano-pipeline.

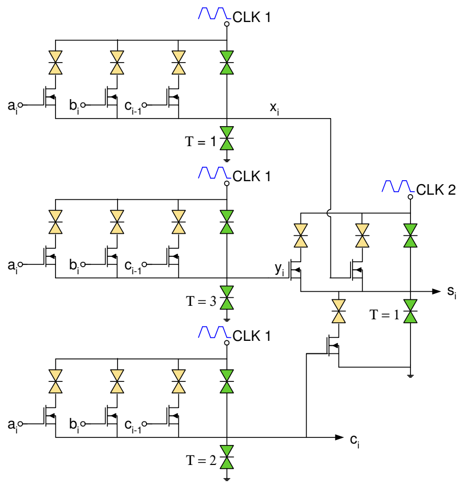

Fig. 4. A nano-pipelined full-adder using MOBILEs.

and the gate returns to the monostable mode of operation. Finally, in the *wait* phase, the inputs of the gate are loaded with the results obtained from the previous pipeline stage.

Fig. 3 shows the main difference between a conventional CMOS pipeline and an RTD-HFET nano-pipeline. In Fig. 3, M stands for a MOBILE gate. Note that in the latter, latches[1] are not needed between pipeline stages because MOBILEs are self-latching. However, latches may be needed in other parts of a circuit for synchronization. Furthermore, the depth of a nano-pipeline stage is four MOBILEs due to the overlapping clocking scheme.

C. A Nano-pipelined Full-Adder

A full-adder can be implemented with two threshold gates. If a_i, b_i, c_{i-1} and c_i, s_i are the inputs and outputs, respectively, then the weight-threshold vectors of the full-adder are as follows:

$$c_i(a_i, b_i, c_{i-1}) : \langle 1, 1, 1; 2 \rangle \quad (5)$$
$$s_i(a_i, b_i, c_{i-1}, c_i) : \langle 1, 1, 1, -2; 1 \rangle . \quad (6)$$

Because MOBILE circuits use an overlapping clocking scheme, the input operands must be latched in order to compute s_i in the second stage. This causes circuit overhead and does not allow for an efficient nano-pipelined operation.

To remove the above inefficiency, it is necessary to remove the dependency of s_i on a_i, b_i, and c_{i-1} [9]. This can be achieved by dividing the full-adder into two stages. The first stage computes

[1]An RTD-HFET latch is simply a buffer, i.e., with weight-threshold vector $\langle 1; 1 \rangle$.

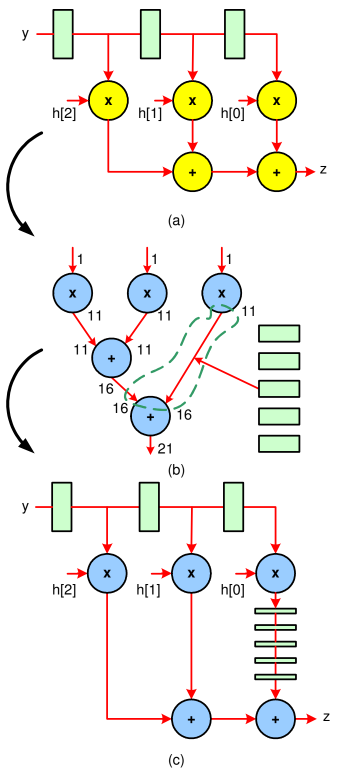

Fig. 5. A four-bit FIR filter to demonstrate nano-pipelining. The circles represent functional units.

$a_i + b_i + c_{i-1}$ (arithmetic sum) as follows:

$$x_i(a_i, b_i, c_i) : \langle 1, 1, 1; 1\rangle \quad (7)$$

$$c_i(a_i, b_i, c_i) : \langle 1, 1, 1; 2\rangle \quad (8)$$

$$y_i(a_i, b_i, c_i) : \langle 1, 1, 1; 3\rangle. \quad (9)$$

The second stage detects if $a_i + b_i + c_{i-1} \in \{1, 3\}$. Thus,

$$s_i(x_i, c_i, y_i) : \langle 1, -1, 1; 1\rangle. \quad (10)$$

Now s_i is dependent on the intermediate variables, x_i, y_i, and c_i which are computed in the previous stage. Therefore, there is no need to latch the inputs. Fig. 4 shows a nano-pipelined full-adder that uses MOBILEs.

III. Motivational Example

In this section, we present a nano-pipelined finite impulse response (FIR) filter as an example to motivate the need for our nano-pipelining algorithm. We also demonstrate the massive throughput that can be obtained with nano-pipelining.

A. Demonstration of Nano-pipelining

The initial RTL architecture of a four-bit FIR filter is shown in Fig. 5(a). The first step in nano-pipelining is to replace each functional unit in the architecture with its nano-pipelined equivalent. During this mapping phase, it is clear that a library of nano-pipelined modules, similar to a CMOS RTL module library, must exist. The components in this library will have their associated area, latency, power, etc., characteristics.

After the mapping phase, the next step is to determine when the inputs and outputs of each of the modules become available. Assuming a four-bit nano-pipelined ripple carry adder (RCA) and array multiplier have latencies of 5 and 10 clock phases (as shown later), respectively, the arrival times (in clock phases) for the signals are annotated next to the modules in Fig. 5(b). Since the inputs to the multipliers come from latches, they are available at clock phase 1. The constants, $h[i]$, are always available and the output, z, is available after 21 clock phases.

The final step in architectural nano-pipelining is to synchronize all paths from the inputs to the outputs so that a traversal along any path will always lead to the same latency. This is achieved by inserting latches on the required paths. For example, the inputs to the last RCA in Fig. 5(b) arrive at the 11th and 16th clock phases, respectively. As shown in Fig 5(b), five latches need to be inserted to synchronize the arrival of the inputs. The nano-pipelined FIR filter is shown in Fig. 5(c).

B. Throughput Analysis

In this subsection, we wish to demonstrate the primary advantage of nano-pipelining — the massive throughput that can be achieved. To do so, we again consider the FIR filter of Fig. 5(a). For a CMOS implementation, we assume that the multiplier has two pipeline stages while the adder is non-pipelined and carry-lookahead (CLA). The critical path of the filter is a single stage of the multiplier and two CLAs. The pipeline stage of the multiplier contains an RCA consisting of three full-adders (in our case, a full-adder consists of two cascaded XOR gates). Assuming that the full-adders are realized using the four-NAND implementation, the critical path of the RCA has 12 gates (six gates for the first full-adder and three gates each for the second and third full-adders). A CLA has six gates in its critical path (one gate to compute the generate/propagate signal, two gates to compute the carry lookahead, and three gates for the second-level XOR gate to compute the sum). Therefore, the critical path of the filter is 24 gates and the total delay is $24D_{cmos}$ where D_{cmos} is the typical delay of a CMOS gate. In the nano-pipelined filter, the critical path has four MOBILEs and the delay is $4D_{mobile}$ where D_{mobile} is the delay of a MOBILE.

The increase in throughput in the nano-pipelined implementation is $6\frac{D_{cmos}}{D_{mobile}}$. Assuming D_{cmos} and D_{mobile} are comparable, the nano-pipelined filter offers 6X higher throughput than its CMOS counterpart. The overall conclusion is that nano-pipelining can drastically increase throughput and data-intensive applications stand to gain the maximum benefits.

IV. Nano-pipelining Algorithm

We present our algorithm for nano-pipelining of a pipelineable RTL architecture in this section. We model the RTL architecture as a directed acyclic graph (DAG), $G(V, E)$, where V and E represent the modules and their interconnections in the architecture, respectively. $F_{in}(v)$ and $F_{out}(v)$ denote the set of inputs and outputs of node v, respectively. It is assumed that there exists a library which contains nano-pipelined versions of every module present in the architecture that is to be nano-pipelined. Furthermore, we assume that the modules are implemented with MOBILEs, although nano-pipelining, in general, is applicable to any nanotechnology that is self-latching. Fig. 6 outlines our nano-pipelining algorithm.

In lines 1-3 of the algorithm, we replace each node (module) in the network with its nano-pipelined equivalent from the module library. The complexity of this step is $O(|V|)$. In line 4, topological sorting of the network is performed so that we can compute the arrival times of each input and output signal of a node in lines 5-6. It is assumed that the input arrival times to the network are known. The output time is simply the sum of the maximum of the input arrival times and the node delay. The complexity of sorting and computing the arrival times is $O(|V + E|)$ and $O(|E|)$, respectively.

Once the arrival times have been computed, it is necessary to insert latches to synchronize all the paths from the inputs to outputs.

```
Require: $G(V,E)$
    // replace each module with nano-pipelined version from library
 1: for all $v \in G(V)$ do
 2:     $v_p \leftarrow$ module_library($v$)
 3:     replace $v$ with $v_p$ in $G$
 4: $G' \leftarrow$ topological sort of $G$
    // compute arrival time, AT, of each signal
 5: for all $e = (u,v) \in G'(E)$ do
 6:     AT($e$) $\leftarrow$ max{AT($k$), $k \in F_{in}(u)$} + delay($u$)
    // insert latches to synchronize all paths from inputs to outputs
 7: for all $v \in G'(V)$ do
 8:     max_AT $\leftarrow$ max{AT($k$), $k \in F_{in}(v)$}
 9:     $k' \leftarrow$ edge that has max_AT
10:     for all $k \in F_{in}(v), k \neq k'$ do
11:         insert max_AT – AT($k$) latches on edge $k$
    // remove extra latches on edges with fanout $> 1$
12: for all $v \in G'(V)$ do
13:     if $F_{out}(v) > 1$ then
14:         min_latches $\leftarrow$ min{AT($k$), $k \in F_{in}(v)$}
15:         $k' \leftarrow$ edge that contains min_latches
16:         for all $k \in F_{out}(v), k \neq k'$ do
17:             remove min_latches on edge $k$, if possible
```

Fig. 6. The nano-pipelining algorithm.

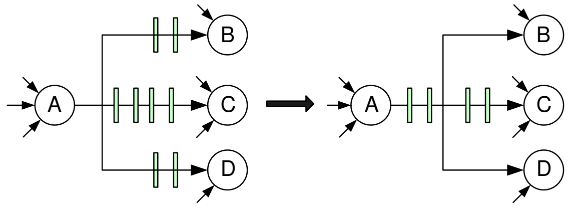

Fig. 7. Relocation of latches from the “branches” to the “stem” of an output node with multiple fanout.

This is achieved by visiting the network nodes in order. As shown in lines 7-11, we determine the maximum arrival time of all inputs to a node. For every input to this node that has its arrival time less than the maximum arrival time, we insert the appropriate number of latches. The architecture is fully nano-pipelined once this has been accomplished. The complexity of this step is $O(|V| \cdot |E|)$.

The above step is a little sloppy in inserting the latches. Fig. 7 shows a scenario in which an output node has multiple fanout. The way the algorithm works, lines 7-11 will insert latches on the “branches” of an output, rather than on its “stem”. To prevent this, lines 12-17 look at nodes with multiple fanout. If an output is detected that contains latches on all its “branches”, then it is possible to relocate the number of common latches on all “branches” to the “stem” of the output. The complexity of this step is $O(|E|)$ and the overall complexity of the algorithm is $O(|V| \cdot |E|)$.

V. Examples

In this section, we present the design of a nano-pipelined array multiplier and a non-restoring divisor. These circuits can be part of the library of nano-pipelined modules required in our nano-pipelining algorithm. One can also use nano-pipelined versions of the designs presented in [10], [11]. We perform a similar analysis later of some DSP architectures when our nano-pipelining algorithm is applied to them.

A. *Nano-pipelined Array Multiplier*

The design of a four-bit nano-pipelined array multiplier is shown in Fig. 8. It consists of the cells FA, FX, FX0, FX1, and FX2. FA is the full-adder discussed in Section II-C while FX is a single threshold gate implementing the product term X_0Y_0. The circuit also contains input and output latches that are required to achieve synchronization for nano-pipelining. A square in Fig. 8 with number y in it represents a series of y latches. The remaining squares represent a series of two latches.

To understand the operation of the nano-pipelined multiplier, we first look at the design of its cells FX0, FX1, and FX2 shown in Fig. 9. It can be seen that one of the inputs to the full-adder in FX0 is always zero while one (two) of the inputs to the full-adders in FX1 (FX2) are constrained by the logical AND of d, e (a, b and d, e). This gives us the opportunity to implement the outputs of the cells in Fig. 9 in a compact way using threshold logic. The Boolean logic function of p for the various cells are as follows:

$$\begin{aligned}
p_{FX0} &= \underbrace{ab(\bar{d}+\bar{e})}_{g} + \underbrace{(\bar{a}+\bar{b})de}_{h} \\
p_{FX1} &= \underbrace{\bar{a}b(\bar{d}+\bar{e})}_{i} + \underbrace{a\bar{b}(\bar{d}+\bar{e})}_{j} + \underbrace{\bar{a}\bar{b}de}_{k} + \underbrace{abde}_{l} \\
p_{FX2} &= \underbrace{abdef}_{m} + \underbrace{ab(\bar{d}+\bar{e})\bar{f}}_{n} + \underbrace{(\bar{a}+\bar{b})de\bar{f}}_{o} + \underbrace{(\bar{a}+\bar{b})\bar{d}f}_{q} + \\
&\quad + \underbrace{(\bar{a}+\bar{b})\bar{e}f}_{r}. \qquad (11)
\end{aligned}$$

Similarly, the Boolean logic function of c for the various cells are as follows:

$$\begin{aligned}
c_{FX0} &= abde \\
c_{FX1} &= ab + (a+b)de \\
c_{FX2} &= \underbrace{ab(de+f)}_{s} + \underbrace{def}_{t}. \qquad (12)
\end{aligned}$$

The functions in (11) are not threshold because many variables appear in both phases in the function’s expression. A well-known necessary condition for a function to be threshold is that it must be unate [12] (the converse is not true, however). In addition, c_{FX2} is also not a threshold function even though it is unate. However, if we let the variables g through o and q through t be as designated above, then it is possible to convert (11) and (12) into threshold functions as follows:

$$\begin{aligned}
p_{FX0}(g,h) &: \langle 1,1;1 \rangle \qquad (13) \\
g(a,b,d,e) &: \langle 2,2,-1,-1;3 \rangle \\
h(a,b,d,e) &: \langle -1,-1,2,2;3 \rangle \\
p_{FX1}(i,j,k,l) &: \langle 1,1,1,1;1 \rangle \qquad (14) \\
i(a,b,d,e) &: \langle -2,2,-1,-1;1 \rangle \\
j(a,b,d,e) &: \langle 2,-2,-1,-1;1 \rangle \\
k(a,b,d,e) &: \langle -1,-1,1,1;2 \rangle \\
l(a,b,d,e) &: \langle 1,1,1,1;4 \rangle \\
p_{FX2}(m,n,o,q,r) &: \langle 1,1,1,1,1;1 \rangle \qquad (15) \\
m(a,b,d,e,f) &: \langle 1,1,1,1,1;5 \rangle \\
n(a,b,d,e,f) &: \langle 2,2,-1,-1,-2;3 \rangle \\
o(a,b,d,e,f) &: \langle -1,-1,2,2,-2;3 \rangle \\
q(a,b,d,f) &: \langle -1,-1,-2,2;1 \rangle \\
r(a,b,e,f) &: \langle -1,-1,-2,2;1 \rangle \\
c_{FX0}(a,b,d,e) &: \langle 1,1,1,1;4 \rangle \qquad (16) \\
c_{FX1}(a,b,d,e) &: \langle 2,2,1,1;4 \rangle \qquad (17) \\
c_{FX2}(s,t) &: \langle 1,1;1 \rangle \qquad (18) \\
s(a,b,d,e,f) &: \langle 3,3,1,1,2;8 \rangle \\
t(d,e,f) &: \langle 1,1,1;3 \rangle.
\end{aligned}$$

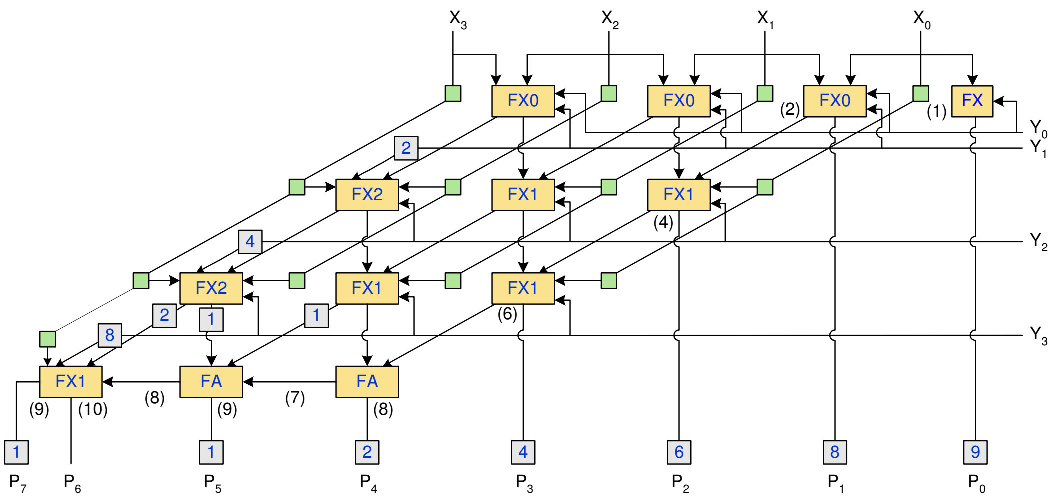

Fig. 8. Schematic diagram of a four-bit nano-pipelined array multiplier.

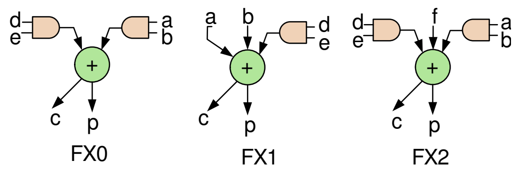

Fig. 9. The cells that comprise the nano-pipelined array multiplier. The circle represents a full-adder.

In converting the Boolean functions into threshold functions, we have taken care to prevent input dependencies from exceeding a single nano-pipeline stage. In this way, we perform a computation and use its results immediately in the next stage. In cell FX2, it takes two clock phases to calculate p and c. On the other hand, it takes two and one clock phases to calculate p and c, respectively, in cells FX0 and FX1. However, both p and c are required at the same time in the next stage. Thus, a latch must be inserted on the path realizing c to equalize its latency with p (this latch is assumed to be inside the FX0 and FX1 cells in Fig. 8 and is thus not shown explicitly). Additional latches are required in the last stage to synchronize the addition, since the partial products arrive earlier than the carry from the full-adders. The clock phase at which each bit of the product term is generated is shown in parenthesis in Fig. 8.

The latency, L_m, in clock phases of an N-bit nano-pipelined array multiplier is the sum of the latencies of the $N-1$ levels in the multiplier and the $N-2$ full-adders and the FX1 cell in the last level. The latency of the carry from the full adder and the sum of the FX1 cell is one and two clock phases, respectively. Therefore,

$$L_m = 2(N-1) + (N-2) + 2 = 3N - 2. \tag{19}$$

The total number of threshold gates (including latches), G_m, required is,

$$\begin{aligned} G_m = {} & G_{FX} + (N-1)G_{FX0} + (N^2 - 4N + 5)G_{FX1} \\ & + (N-2)G_{FX2} + (N-2)G_{FA} + (3N-2)(N-1) \\ & + \frac{(L_m - 2N)(L_m - 2N + 1) + 2}{2} + (N-1)(L_m - N) \\ & + (2L_m - 1) + (N-2)^2, \end{aligned} \tag{20}$$

where G_{FX}, G_{FX0}, G_{FX1}, G_{FX2}, and G_{FA} are the threshold gate counts of the various cells and are equal to one, five (four for the logic and one for the latch), seven (six for the logic and one for the latch), nine, and four, respectively. Table I shows the gate count and latency for nano-pipelined array multipliers with bit-widths of 4, 8, 16, 32, and 64.

B. Nano-pipelined Non-Restoring Divisor Circuit

The relevant portion of a nano-pipelined, non-restoring divisor circuit is shown in Fig. 10. The dividend is loaded in the lower half of the remainder latch while the divisor and its complement are stored in other latches. We store the complement as well because if we did not, an extra nano-pipeline stage would be necessary for this computation each time. This would involve inserting latches in all the other parts of the circuit and would cause significant overhead. It is simpler just to compute the complement in the beginning and have it available throughout the division process.

The nano-pipelined RCA operates on the upper half of the remainder latch. The shifting of the remainder is performed implicitly by the wiring. The most significant bit (MSB) from the RCA is used to select whether the divisor or its complement gets added in the next stage. In addition, the MSB is inverted to output the quotient bit and serves as the carry-in to the RCA in the next stage. The latency, L_d, of the divisor in clock phases is,

$$L_d = 1 + (L_{RCA_N} + 2)(N+1), \tag{21}$$

where L_{RCA_N} is the latency of an N-bit nano-pipelined RCA. We observe that to reduce the latency of the divisor, it is important to optimize the RCA. For this analysis, we use an N-bit nano-pipelined RCA whose latency is simply $N+1$ clock phases. One can also use the more sophisticated nano-pipelined adders presented in [9] to improve the latency of this circuit. The total number of required threshold gates (including latches), G_d, is,

$$\begin{aligned} G_d = {} & (N+1)G_{RCA_N} + N^2 \cdot G_{MUX} + L_{RCA_N}\frac{9N^2 + 6N}{4} \\ & + 4N(N+1), \end{aligned} \tag{22}$$

where G_{MUX} is the number of threshold gates needed to implement a one-bit 2:1 multiplexer, and is equal to three (two gates for logic and one latch), L_{RCA_N} is as mentioned above, and G_{RCA_N} is the number of threshold gates needed for an N-bit nano-pipelined RCA and is,

$$G_{RCA_N} = 4N + 1.5(N^2 - N) + 1. \tag{23}$$

Table I shows the threshold gate count and latency for nano-pipelined non-restoring divisor circuits of different bit-widths.

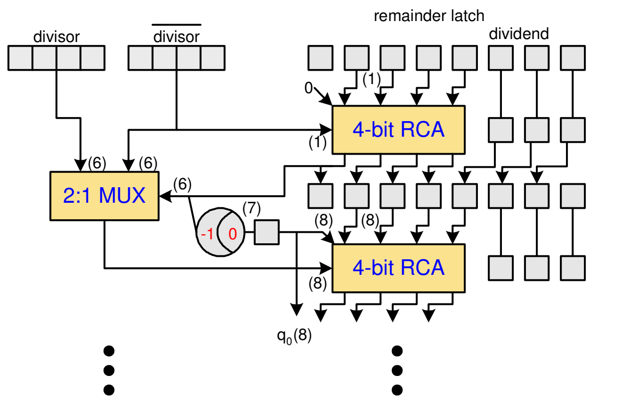

Fig. 10. A portion of a four-bit nano-pipelined non-restoring divisor circuit.

TABLE I
GATE COUNT AND LATENCY CHARACTERISTICS OF NANO-PIPELINED ARITHMETIC CIRCUITS

	Mult	Div	Mult	Div
N	# Threshold gates		Latency (clock phases)	
4	152	513	10	36
8	726	2,937	22	100
16	3,170	19,281	46	324
32	13,242	138,273	94	1,156
64	54,122	1,044,033	190	4,356

C. Examples of Nano-pipelined DSP Architectures

In this subsection, we present the threshold gate count, latch overhead, latency, and throughput for some nano-pipelined DSP architectures. The word size of the architecture is assumed to be four bits. In calculating the throughput offered by a CMOS and a nano-pipelined implementation of an architecture, we make the same assumptions about the modules and technology characteristics as those presented in Section III-B.

Table II presents the results of our analysis. The examples consist of direct and symmetric FIRs, and direct infinite impulse response (IIR) filter [13]. The tap size of a filter is just the number of data points that are used for calculating the result. We applied our algorithm to obtain the nano-pipelined version of these architectures. The gate count and latencies were calculated using the equations developed in previous subsections. The latch overhead represents the number of latches that are inserted in the architecture (not the sub-modules) to synchronize the paths needed for nano-pipelining. As can be seen, nano-pipelined architectures have the potential to offer massive throughput.

VI. FUTURE DIRECTIONS

We devote this section to discussing possible enhancements and developments to the initial algorithm for nano-pipelining. A major problem is to try to decrease the circuit overhead that is accrued during nano-pipelining. This can only be achieved through efficient design of circuits based on RTD-HFET gate technology. Second, nano-pipelining is an idea that should be exploited in the high-level synthesis paradigm. Although one can find examples of architectures that are not nano-pipelineable, many architectures of interest are and it is important to address this concept in high-level design for nanotechnologies. It will also be helpful to develop a design automation tool for design space exploration and synthesis of nano-pipelined architectures. Finally, it will be necessary to develop communication mechanisms for interfacing with the external environment.

TABLE II
FOUR-BIT NANO-PIPELINED DSP ARCHITECTURES

Tap	# Threshold gates	Latch overhead	Latency (clock phases)	Throughput increase
	Direct FIR			
2	339	0	15	4.5X
4	713	15	25	7.5X
6	1,087	50	35	10.5X
8	1,461	105	45	13.5X
16	2,957	525	85	25.5X
	Symmetric FIR			
2	187	0	15	4.5X
4	409	0	20	6X
6	596	5	25	7.5X
8	853	15	30	9X
16	1,741	115	50	15X
	Direct IIR			
2	526	0	20	6X
4	1,274	20	30	9X
6	2,022	80	40	12X
8	2,770	155	50	15X
16	5,762	915	90	27X

VII. CONCLUSIONS

In this paper, we presented nano-pipelined designs of some arithmetic circuits. We also introduced the first algorithm to enable architectural nano-pipelining. This is achieved by first replacing each module in the RTL architecture with its nano-pipelined version and then inserting latches, where needed, to synchronize all paths from the inputs to the outputs. We presented examples and demonstrated that it is possible to achieve high throughput using nano-pipelined architectures. Furthermore, we showed the application of RTD-HFET nanotechnology to nano-pipelining. We hope that this work has laid the foundation for further research in exploring novel nano-pipelined architectures.

REFERENCES

[1] "Semiconductor Industries Association Roadmap." http://public.itrs.net

[2] R. H. Mathews *et al.*, "A new RTD-FET logic family," *Proc. IEEE*, vol. 87, no. 4, pp. 596–605, Apr. 1999.

[3] K. Maezawa *et al.*, "High-speed and low-power operation of a resonant tunneling logic gate MOBILE," *IEEE Electron Device Lett.*, vol. 19, no. 3, pp. 80–82, Mar. 1998.

[4] J. P. Sun *et al.*, "Resonant tunneling diodes: Models and properties," *Proc. IEEE*, vol. 86, no. 4, pp. 641–661, Apr. 1998.

[5] K. F. Goser *et al.*, "Aspects of systems and circuits for nanoelectronics," *Proc. IEEE*, vol. 85, no. 4, pp. 558–572, Apr. 1997.

[6] W. Williamson *et al.*, "12 GHz clocked operation of ultralow power interband resonant tunneling diode pipelined logic gates," *IEEE J. Solid-State Circuits*, vol. 32, no. 2, pp. 222–230, Feb. 1997.

[7] P. Mazumder *et al.*, "Digital circuit applications of resonant tunneling devices," *Proc. IEEE*, vol. 86, no. 4, pp. 664–686, Apr. 1998.

[8] C. Pacha and K. Goser, "Design of arithmetic circuits using resonant tunneling diodes and threshold logic," in *Proc. Wkshp. Innovative Circuits & Systems for Nanoelectronics*, Sept. 1997, pp. 83–93.

[9] C. Pacha *et al.*, "Resonant tunneling device logic circuits," University of Dortmund and Gerhard-Mercator University of Duisburg, Tech. Rep., July 1999.

[10] L. Ciminiera and A. Serra, "Efficient serial-parallel arrays for multiplication and addition," in *Proc. Int. Conf. Computer Arithmetic*, June 1985, pp. 28–35.

[11] L. Dadda, "Fast multipliers for two's complement numbers in serial form," in *Proc. Int. Conf. Computer Arithmetic*, June 1985, pp. 57–63.

[12] Z. Kohavi, *Switching and Finite Automata Theory*. New York, NY: McGraw-Hill, 1978.

[13] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing: Principles, Algorithms, and Applications*. Upper Saddle River, NJ: Prentice-Hall, 1996.