2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)

High-Throughput Multiplier Architectures Enabled by Intra-Unit Fast Forwarding

Jihee Seo and Dae Hyun Kim
School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA, USA
jihee.seo@wsu.edu, daehyun@eecs.wsu.edu

***Abstract*—In this paper, we propose a pipelined multiplier architecture that can resolve data dependencies. The proposed architecture generates partial results in the pipeline stages of the multiplier and forwards the partial results back to the pipeline stages through so-called fast-forwarding paths, thereby enabling an execution of dependent multiplications with a minimum delay penalty. We apply the architecture to a normal binary multiplier (NBBE-2) and two redundant binary multipliers (RBBE-4 and CRBBE-4) and compare the execution time, clock period, area, and power consumption of the multipliers. The simulation results show that the proposed architecture achieves up to 30% execution time reduction.**

***Keywords*-Multiplier; Multiplication; Fast Forwarding;**

I. Introduction

The performance of arithmetic units such as multipliers and dividers has a great impact on the throughput of processing units utilizing the arithmetic units. Thus, an enormous amount of effort has been put into the design of high-speed arithmetic units [1]–[4]. Especially, integer multiplication is heavily used in almost all applications. Thus, many researchers have developed and proposed various high-speed multiplier architectures [1], [3], [5]–[8].

Multiplier architectures can be generally decomposed into three steps, partial product generation (PPG), partial product reduction (PPR), and carry-propagate addition (CPA). The PPG step generates partial products to be added. The PPR step adds the partial products generated in the PPG step until only two rows (a sum and a carry rows) are left. The CPA step adds the two final rows by a carry-propagate adder to produce the final multiplication result. Thus, improving the performance of a multiplier tries to reduce the number of partial products [9], reduce the computation time of the PPR and CPA steps [10], or use different number representations such as the redundant binary (RB) representation [11]. All these methodologies reduce the delay of the multiplier, but not its throughput (# multiplications per cycle). Thus, some processors use pipelining to increase the throughput at a cost of increased chip area and delay due to flip-flops. However, pipelining does not increase the throughput if dependencies exist in the incoming multiplications. Processors resolve the dependency problem using various techniques such as out-of-order execution. In many cases, however, those techniques cannot fully utilize the pipelined multipliers due to the dependencies.

In this paper, we apply an intra-unit fast-forwarding architecture to pipelined multipliers to improve the throughput. The main idea is to generate partial results in all or some of the pipeline stages and forward the results back to previous pipeline stages. If incoming multiplications are independent of each other, the forwarding paths are not used. If they are dependent, however, the partial results of earlier multiplications being executed in the multiplier are used in some of the previous pipeline stages for later multiplications, thereby increasing the throughput.

II. Intra-Unit Fast-Forwarding Architecture

Similar to the data bypassing architectures in datapath architectures, the general intra-unit fast-forwarding architecture consists of partial results generation, fast forwarding, and multiplexing [12]. In this section, we briefly review intra-unit pipelined architectures and present the general intra-unit fast-forwarding architecture.

A. Definitions and Terminologies

Two multiplications are *independent* if the output of one of them is not used as an input of the other. Two multiplications M_1 and M_2 are *dependent* if they are not independent. In this case, M_2 is dependent on M_1 if one of the inputs of M_2 is the output of M_1. Notice that multiple multiplications $(M_2, ..., M_j)$ can be dependent on the same multiplication (M_1). In addition, a multiplication can be dependent on two multiplications. For example, suppose $M_1 : C = A * B$, $M_2 : F = D * E$, $M_3 : G = C * F$. In this case, the two inputs of M_3 are the outputs of M_1 and M_2, so M_3 are dependent on both M_1 and M_2. In addition, when a multiplication M_A is dependent on another multiplication M_B, the output of M_B could be used for M_A as a multiplicand and/or a multiplier. Thus, we define the following three dependency types for $Y = OP_1 * OP_2$:

- Type01 dependency: OP_1 is independent and OP_2 is dependent.
- Type10 dependency: OP_1 is dependent and OP_2 is independent.
- Type11 dependency: Both OP_1 and OP_2 are dependent.

We also define the dependency distance between two dependent multiplications as follows:

- Dependency distance: When j multiplications $M_1, ..., M_j$ are executed in this order, if M_j is

978-1-7281-3366-9/19/$31.00 ©2019 IEEE
DOI 10.1109/ARITH.2019.00036

Table I

COMPARISON OF A NON-PIPELINED, AN n-STAGE PIPELINED, AND AN n-STAGE PIPELINED FULL FAST-FORWARDING (FF) ARCHITECTURES. L: LOGIC DELAY. δ_{F}: A D FLIP-FLOP DELAY. δ_{M}: A MULTIPLEXER DELAY. "EXECUTION TIME (I)" AND "EXECUTION TIME (D)" ARE THE EXECUTION TIMES FOR k INDEPENDENT AND k SEQUENTIALLY DEPENDENT OPERATIONS, RESPECTIVELY.

	Non-pipelined	Pipelined	Pipelined + Full FF
Stage delay	L	$L/n + \delta_{\mathrm{F}}$	$L/n + \delta_{\mathrm{F}} + \delta_{\mathrm{M}}$
Execution time (I)	$k \cdot L$	$\frac{k}{n} \cdot (L + n \cdot \delta_{\mathrm{F}}) + (n-1) \cdot \delta_{\mathrm{F}} + (L - \frac{L}{n})$	$\frac{k}{n} \cdot (L + n \cdot (\delta_{\mathrm{F}} + \delta_{\mathrm{M}})) + (n-1) \cdot (\delta_{\mathrm{F}} + \delta_{\mathrm{M}}) + (L - \frac{L}{n})$
Execution time (D)	$k \cdot L$	$k \cdot (L + n \cdot \delta_{\mathrm{F}})$	

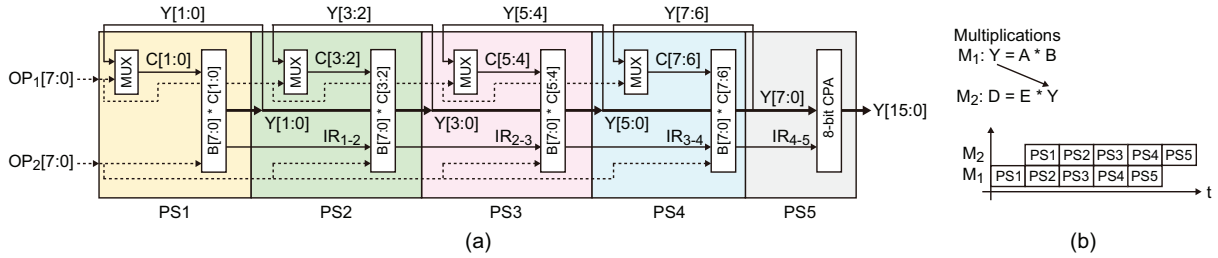

Figure 1. (a) An intra-unit pipelined multiplication architecture with D1 fast-forwarding paths. (b) A timing diagram for the architecture shown in (a).

dependent on M_1, the *dependency distance* between M_j and M_1 is defined as $j-1$.

Sequentially dependent multiplications are a stream of dependent multiplications (except a few leading multiplications) whose dependency distances are all one. We also define the D-s dependency as follows:

- D-s dependency: If M_j is dependent on M_i, M_j has a D-s (Distance-s) dependency where $s = j - i$.

For example, if M_4 is dependent on M_1, M_4 has a D-3 dependency. Notice that a multiplication can have two dependencies if it has a Type11 dependency. For example, if M_5 is dependent on M_2 and M_4, M_5 has a D-3 and a D-1 dependencies.

B. Intra-Unit Pipeline Architecture

Pipelined architectures are widely used for throughput improvement in high-performance arithmetic units. Table I compares a non-pipelined and a pipelined architectures for executing k independent operations and k sequentially dependent operations. The non-pipelined architecture has one logic stage and the delay of the logic is L. For this architecture, executing k independent operations does not differ from executing k dependent operations. Thus, the total execution time of k operations is $k \cdot L$.

The pipelined architecture, however, has a shorter stage delay than the non-pipelined architecture. An ideal pipelining is to split the logic into n equal-delay stages. In this case, the delay of a stage is $L/n + \delta_{\mathrm{F}}$ where δ_{F} is the delay of a flip-flop. If k independent operations are executed in this architecture, an operation can be issued every clock cycle, so the total execution time is $\frac{k}{n} \cdot (L + n \cdot \delta_{\mathrm{F}}) + (n-1) \cdot \delta_{\mathrm{F}} + (L - \frac{L}{n})$ as shown in the table. Assuming k is sufficiently large and n is sufficiently small, the total execution time is reduced by $1/n$ compared to the non-pipelined architecture. However, if the operations are sequentially dependent, the total execution time becomes $k \cdot (L + n \cdot \delta_{\mathrm{F}})$, which is longer than that of the non-pipelined architecture. In addition, if the pipeline depth n goes up, both the flip-flop delay and area overheads increase. Thus, n should be small enough to minimize the overheads, but large enough to maximize the throughput.

C. Intra-Unit Fast-Forwarding Architecture

The motivation of this paper is to reduce the total execution time regardless of their dependency patterns. A solution for the execution time reduction is to generate some partial results for a multiplication M_1 in the pipeline stages of a multiplier. If a multiplication M_2 is dependent on M_1, the multiplier uses the partial results of M_1 to execute M_2 with a minimum delay penalty.

Figure 1(a) shows an example of an 8-bit 5-stage multiplier. Suppose $A[7:0]$ and $B[7:0]$ are fed into the multiplier for $M_1 : Y = A * B$. In the first stage, the multiplexer selects the input A and the logic calculates $B[7:0] * A[1:0]$ and generates two outputs. The first output is a partial result $Y[1:0]$ denoted by the thick line in the figure and the second output is an intermediate output (if necessary) denoted by IR_{1-2}. Some or all of the primary inputs A and B, the partial result $Y[1:0]$, and the intermediate output are passed to the next stage. In the second stage, the multiplexer selects $A[3:2]$ and the logic calculates $B[7:0] * A[3:2]$ and generates $Y[3:0]$ (actually $Y[3:2]$ is newly generated and $Y[1:0]$ is just concatenated). At the same time, the second stage also generates an intermediate output IR_{2-3}. The third and fourth stages are similar to the second stage. At the end of the fourth stage, a partial result $Y[7:0]$ and an

intermediate output IR_{4-5} are generated. The intermediate output is fed into the 8-bit carry-propagation adder (CPA), which generates the final result $Y[15:8]$. Concatenating the output of the 8-bit CPA and $Y[7:0]$ produces $Y[15:0]$.

Now, suppose the next multiplication is $M_2 : D = E * Y$, which has a Type01 dependency, and E and Y are fed into OP_2 and OP_1, respectively. In this case, M_2 is issued one clock cycle after M_1 is issued. In the first stage for M_2, the multiplexer selects the partial result coming from the next stage. Thus, the first stage for M_2 computes $E[7:0] * Y[1:0]$. During the next clock cycle, the third stage forwards $Y[3:2]$ back to the second stage and computes $B[7:0] * A[5:4]$. At the same time, the multiplexer in the second stage selects $Y[3:2]$ and the logic in the stage computes $E[7:0] * Y[3:2]$ for M_2. Thus, the two multiplications can be executed with a minimum delay penalty. Figure 1(b) shows a timing diagram for M_1 and M_2.

Before we present detailed multiplier architectures in the next section, we define a D-s (Distance-s) fast-forwarding path and discuss properties of the fast-forwarding architecture shown in Figure 1(a).

- D-s fast-forwarding path: If the partial result of the j-th pipeline stage is forwarded to the i-th pipeline stage, we call the forwarding path *a D-s fast-forwarding (or just forwarding) path* where s is $j - i + 1$.

In Figure 1(a), for example, all the forwarding paths are D-1 forwarding paths because the partial result generated at stage j is fed back to stage j for $j = 1, ..., 4$ (through the flip-flops between stage j and stage $j+1$). The D-1 forwarding paths enable the multiplier to execute multiplications having D-1 dependencies with a minimum delay penalty. Thus, the execution time for M_1 and M_2 in Figure 1(a) is six cycles as shown in Figure 1(b).

D. Dependencies, Fast Forwarding, and Overheads

In general, if a multiplier has D-j forwarding paths, it can resolve D-j dependencies. However, whether a dependency could be resolved by fast forwarding or not is strongly dependent on available forwarding paths in the multiplier and the actual dependencies among incoming multiplications. For example, if the multiplier has only D-s forwarding paths, it cannot resolve D-j dependencies for $j \neq s$. Figures 2(b)-(f) compare several architectures for the two dependency patterns shown in Figure 2(a). For the comparison, we assume that the flip-flop and multiplexer delays are negligible and the multiplier has four pipeline stages. In Case 1, i_2, i_3, and i_4 have D-1 Type01 dependencies. In Case 2, i_3 and i_4 have D-2 Type01 dependencies. For the non-pipelined multiplier shown in Figure 2(b), the total execution time is $4T$. For the four-stage pipelined multiplier shown in Figure 2(c), the total execution time for independent multiplications is $1.75T$, whereas the total execution times for Case 1 and Case 2 are $4T$ and $2.25T$, respectively. If the pipelined multiplier has D-1 forwarding paths as shown in Figure 2(d), the total execution time for independent multiplications and the dependent multiplications in Case 1 is $1.75T$, which is better than the architecture in Figure 2(c). However, it cannot resolve D-2 dependencies, so the execution time for Case 2 is still $2.25T$. The pipelined multiplier with D-2 forwarding paths shown in Figure 2(e) has shorter execution time ($1.75T$) than the architectures shown in Figure 2(c) and (d), but it has longer execution time ($2.5T$) for Case 1. Lastly, the multiplier in Figure 2(f) has full (D-1, D-2, D-3) forwarding paths, so its execution time is $1.75T$ for all the independent and dependent multiplications. The fourth column in Table I shows the execution time for independent and sequentially dependent operations. As long as some operations are dependent, the pipelined architecture with full forwarding paths has shorter execution time than the one without forwarding paths.

As shown above, having full forwarding paths will minimize the execution time regardless of the dependency patterns. However, adding forwarding paths has two problems. First, adding forwarding paths requires multiplexers and more routing resources, which leads to area and power overheads. We can resolve this problem by adding only the most-demanding forwarding paths. Second, multiplier architectures that can support fast forwarding for Type01/10/11 dependencies should be developed, which is the main topic of this paper.

III. High-Throughput Multiplier Architectures

In this section, we propose high-throughput multiplier designs by inserting fast forwarding paths to several multiplier architectures.

A. Intra-Unit Fast-Forwarding Paths

Figure 3 shows two 8-bit 5-stage pipelined unsigned multiplier architectures having intra-unit fast-forwarding paths. The architecture (*Arch1*) shown in Figure 3(a) resolves Type01 and Type10 dependencies, whereas the architecture (*Arch2*) shown in Figure 3(b) resolves Type01, Type10, and Type11 dependencies. The first stage of Arch1 calculates $A[7:0] * B[1:0]$ and generates two intermediate outputs and a partial result. The intermediate outputs are the sum and carry-out rows of $A[7:0] * B[1:0]$. The partial result is $M[1:0]$ where M is the final result. Thus, the first stage of Arch1 requires a complete 8-bit input for A, but a partial 2-bit input ($B[1:0]$) for B and generates a partial output $M[1:0]$. Thus, the partial result can be forwarded back to the first stage so that multiplications having D-1 dependencies can use the forwarding path. Similarly, the second stage of Arch1 calculates $A[7:0] * B[3:2]$ and the intermediate output of the first stage using carry-save adders. The outputs of the second stage are an intermediate output and a partial result $M[3:2]$. The partial result $M[1:0]$ generated in the first stage is also passed to the second stage, so the second stage actually has $M[3:0]$.

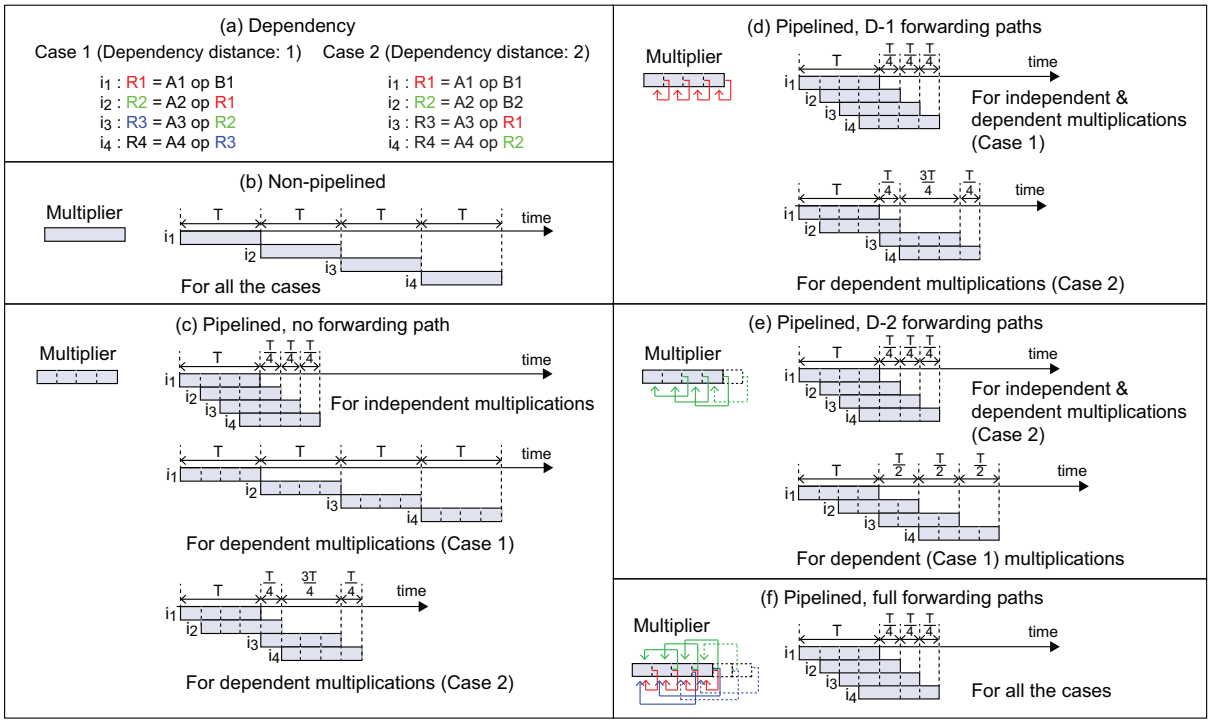

Figure 2. Timing diagrams of non-pipelined and pipelined architectures.

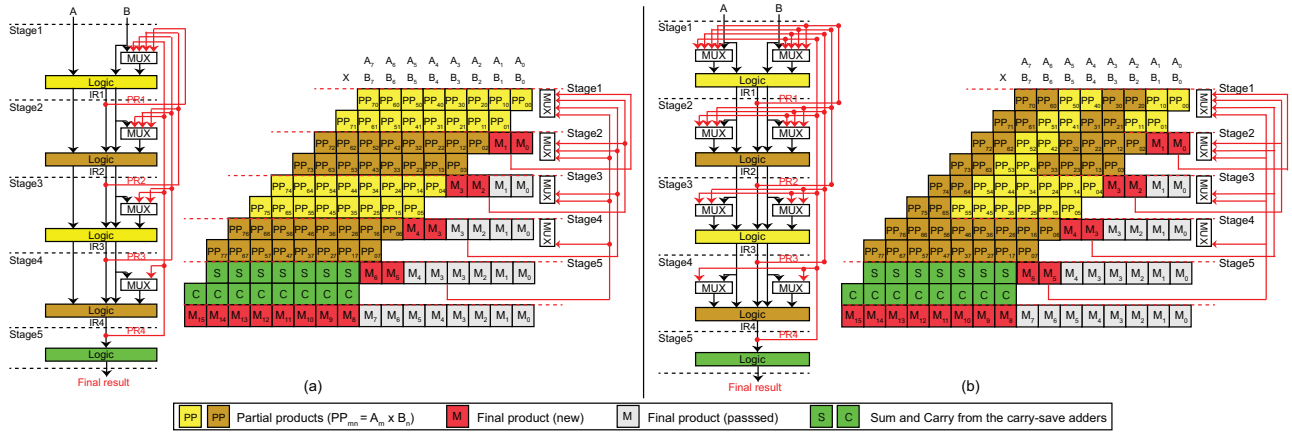

Figure 3. Two 8-bit 5-stage pipelined unsigned multipliers. (a) Arch1 resolving Type01 and Type10 dependencies. (b) Arch2 resolving Type01, Type10, and Type11 dependencies.

$M[3:2]$ is forwarded back to the second stage and $M[1:0]$ is forwarded back to the first stage. The third and fourth stages have similar architectures as shown in the figure. The fifth stage is a carry-propagation addition stage, which adds the intermediate output (the sum and carry-out rows) of the fourth stage using a high-speed parallel adder. The fifth stage generates the final result $M[15:0]$. Note that some additional flip-flops are needed after the fifth stage to hold the final result and forward it to the previous stages to resolve D-2 to D-5 dependencies. Note also that we can resolve Type10 dependencies by swapping the inputs.

Arch2 in Figure 3(b) resolves all the three dependency types as follows. The first stage of Arch2 adds $A[1:0]*B[0]$ and $A[1:0]*B[1]$ and generates an intermediate output and a partial result $M[1:0]$. Arch2 requires partial inputs for the first stage computation, whereas Arch1 requires a complete input and a partial input. Thus, Arch2 can also resolve Type11 dependencies. The second, third, and fourth stages of Arch2 are similar to the first stage as shown in the figure. The fifth stage is a carry-propagation addition stage.

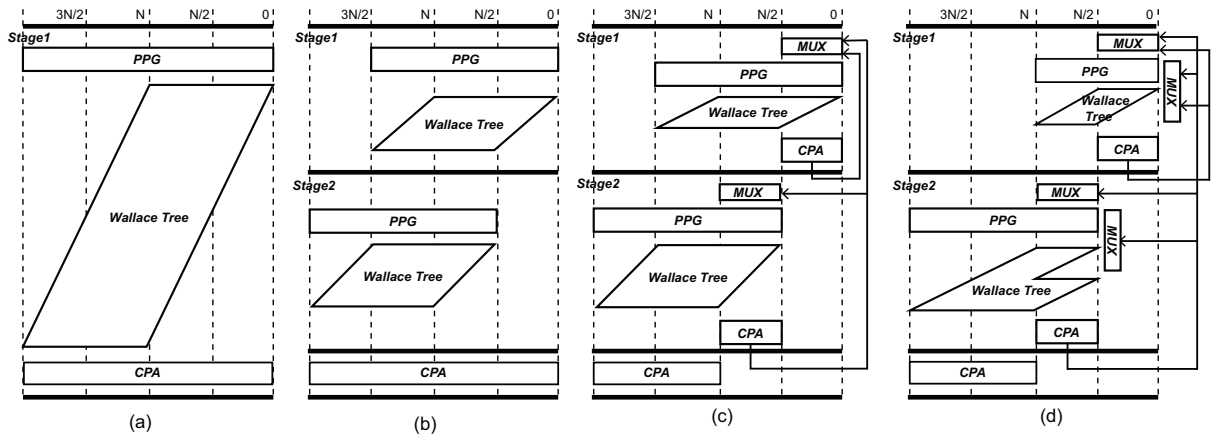

Figure 4. General N-bit three-stage signed multiplier architectures. (a) Non-pipelined. (b) Pipelined. (c) Pipelined + fast forwarding paths (Arch1) resolving Type01/10 dependencies. (d) Pipelined + fast forwarding paths (Arch2) resolving Type01/10/11 dependencies.

Note that the number of partial products to add in each stage is fixed in Arch1, but linearly increases in Arch2. However, the maximum number of bits to add at each column in each stage of Arch2 is fixed to 4 in the figure and $2 \cdot N/(S-1)$ in general where N and S are the number of operand bits and the number of pipeline stages, respectively. If the multiplier in Figure 3 is a signed multiplier, there will be more bits to be added, but the same architectures can still be used for signed multiplications.

Figure 4 compares general N-bit three-stage signed multiplier architectures compared in this paper. All the architectures are composed of three stages, partial product generation, partial product reduction, and carry-propagate addition. The non-pipelined multiplier in Figure 4(a) uses the Wallace tree in the partial product reduction stage, but it can use different partial product reduction architectures such as the Dadda tree [13]. The three-stage pipelined architecture shown in Figure 4(b) splits the partial product generation and partial product reduction stages into two stages. If it has s pipeline stages, the first $s-1$ stages will consist of the partial product generation and reduction stages and the last stage will be the carry-propagate addition stage. The three-stage pipelined architecture in Figure 4(c) has D-1 and D-2 forwarding paths to resolve Type01 and Type10 dependencies. The first and second stages contain small carry-propagate adders to generate partial results. Thus, the last stage carry-propagate adder is an N-bit adder. Since the Wallace tree is just split into upper and lower stages, this architecture (Arch1) resolves only Type01 and Type10 dependencies. On the other hand, Arch2 shown in Figure 4(d) splits the Wallace tree in V-shaped form, thereby resolving Type11 dependencies too.

B. Applications to Existing Multipliers

In this section, we apply the general fast forwarding paths to three multiplier architectures, normal binary based multiplication using Radix-4 Booth encoding (NBBE-2) [9], redundant binary based multiplication using Radix-16 Booth encoding (RBBE-4) [14], [15], and redundant binary based multiplication using covalent Radix-16 Booth encoding (CRBBE-4) [1]. Figure 5 shows partial product reduction trees used for NBBE-2, RBBE-4, and CRBBE-4.

1) NBBE-2: Figure 5(a) shows a partial product reduction tree architecture we propose for NBBE-2 Arch1. We use the sign extension prevention method proposed in [16] in the architecture. We also use the Booth encoder and decoder used in [3] for Radix-4 Booth encoding. The figures show two partial product reduction stages (Pipeline Stage 1 and 2) and a carry-propagate addition stage (Pipeline Stage 3). The first pipeline stage adds the partial products in two sub-stages, Stage 1a and Stage 1b. While we add the higher bits (Bit 11 to Bit 4) and generate an intermediate output in Stage 1b, we add the lower bits (Bit 3 to Bit 0) by a carry-propagate adder (CPA), generate a partial result, and forward it to the first stage (the forwarding paths are not shown in the figure). The second pipeline stage is composed of four sub-stages, Stage 2a to Stage 2d. In this stage, we add the intermediate output passed from the first pipeline stage and the partial products generated in the second pipeline stage. The last sub-stage (Stage 2d) generates a partial result by a CPA. The third pipeline stage adds the intermediate output passed from the second pipeline stage by a CPA and generates the final result. Figure 5(b) shows a partial product reduction tree architecture we propose for NBBE-2 Arch2. It is similar to the Arch1 design, but the first pipeline stage adds only the partial products of $A[3:0] * B[3:0]$. Stage 1b uses a CPA to generate the partial result $Y[3:0]$. Similarly, Stage 2d uses a CPA to generate the partial result $Y[7:4]$.

2) RBBE-4 and CRBBE-4: Figure 5(c) and (d) show the partial product reduction trees for the RBBE-4 Arch1 and Arch2 designs, respectively. Similarly, Figure 5(e) and (f) show the partial product reduction trees for the CRBBE-4

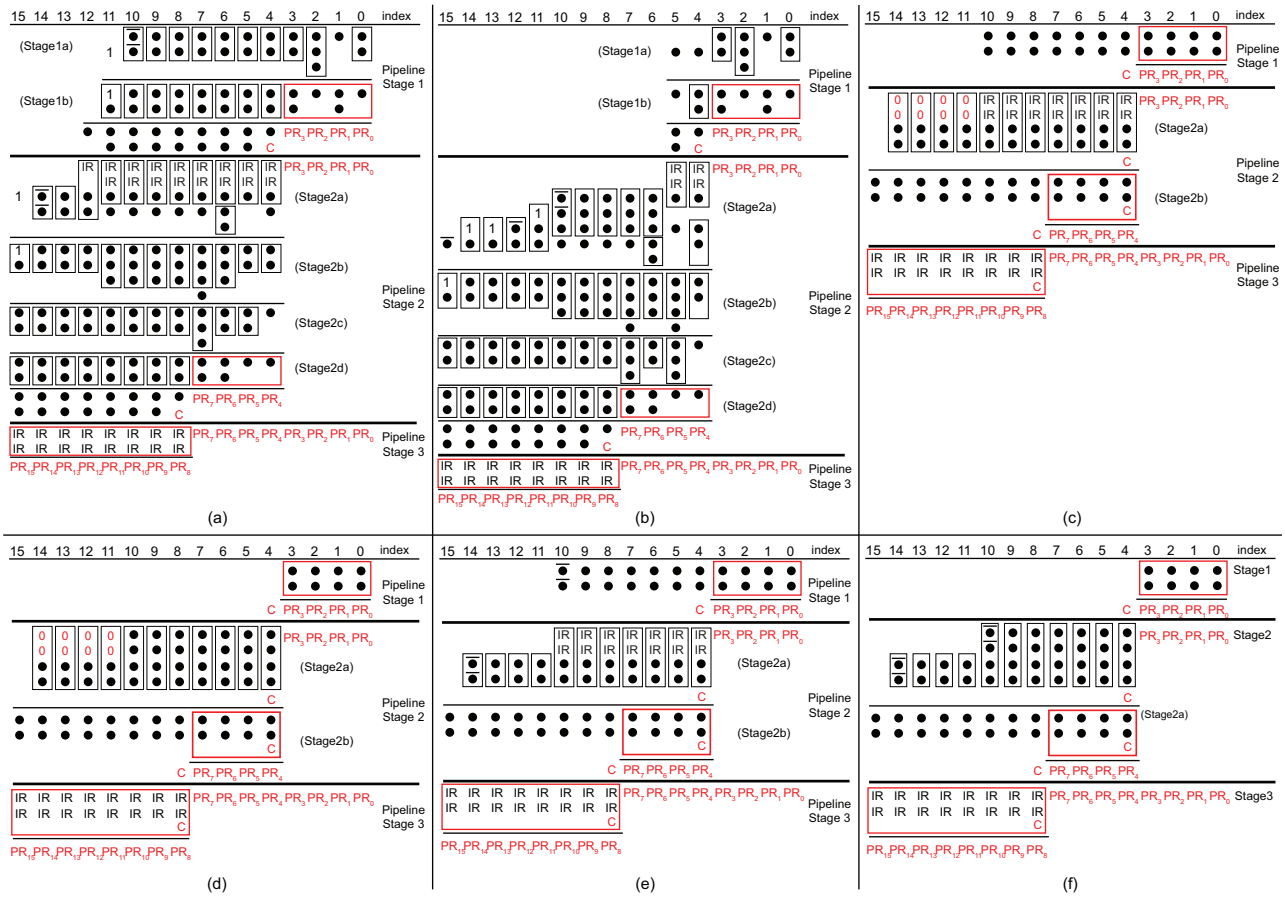

Figure 5. Partial product reduction trees and carry-propagate addition. (a) NBBE-2 (Arch1), (b) NBBE-2 (Arch2), (c) RBBE-4 (Arch1), (d) RBBE-4 (Arch2), (e) CRBBE-4 (Arch1), (f) CRBBE-4 (Arch2).

Arch1 and Arch2 designs, respectively. In the first stage of RBBE-4 and CRBBE-4, we add the RB partial products while generating a partial result by a redundant binary addition followed by redundant binary to normal binary conversion. The intermediate outputs are passed to the second pipeline stage in which the four rows of partial products are added and some partial results are generated. The width of the CPA is $\frac{N}{S-1}$ where S is the total number of pipeline stages.

IV. Simulation Results

In this section, we present simulation results of the three multipliers. We implemented the multipliers using VHDL and synthesized them using Synopsys Design Compiler and the Nangate 45nm open cell library [17]. For the carry-propagate addition, we used the Kogge-Stone adder [18]. The following lists the multiplier architectures we compare:

- N-P: Non-pipelined multipliers.
- Base: Pipelined multipliers without forwarding paths. The pipelining scheme is the same as that of Arch1.
- Arch1: Pipelined multipliers with forwarding paths resolving Type01/10 dependencies.
- Arch2: Pipelined multipliers with forwarding paths resolving Type01/10/11 dependencies.

A. Clock Period

Table II shows the clock periods of 32- and 64-bit multiplier designs. S is the number of pipeline stages for the Base, Arch1, and Arch2 designs. The clock periods of the Base designs are greater than L/S where L is the clock period of the N-P designs. The reason is that the delays of the N-P designs are not evenly distributed throughout the pipeline stages. In addition, pipelining adds flip-flop delays to the logic delay. The clock periods of the Arch1 and Arch2 designs are greater than those of the Base designs because the Arch1 and Arch2 designs have multiplexers, which add additional delays into the logic. Moreover, the Arch2 designs have more multiplexers than the Arch1 delays, so the Arch2 designs have longer clock periods than the Arch1 designs in general. If all multiplications are independent, the Base designs will have shorter execution time than the Arch1

Table II
CLOCK PERIOD, AREA (um^2), AND POWER (uW). S: # PIPELINE STAGES (FOR BASE, ARCH1, AND ARCH2). THE LAST PIPELINE STAGES ARE CARRY-PROPAGATE ADDITION.

N	S	Design	Clock period (ns)				Area (um^2)				Power (uW)			
			N-P	Base	Arch1	Arch2	N-P	Base	Arch1	Arch2	N-P	Base	Arch1	Arch2
32 bits	2	NBBE-2	1.64	1.19	1.56	1.58	8534.61	9299.09	8346.81	8356.92	381.4	429.2	359.3	359.4
		RBBE-4	1.75	1.43	1.64	1.68	12107.52	13860.73	10870.36	10732.83	533.6	645.7	476.4	472.1
		CRBBE-4	1.70	1.17	1.64	1.67	14701.29	15568.98	14326.23	14309.74	677.2	716.3	662.3	664.5
	3	NBBE-2	-	1.07	1.37	1.54	-	9797.31	8984.95	9306.81	-	501.8	446.8	472.4
		RBBE-4	-	1.24	1.41	1.56	-	14454.71	11447.58	13099.17	-	737.7	572.1	622.8
		CRBBE-4	-	1.15	1.47	1.69	-	15246.85	13049.43	14707.67	-	773.6	655.2	694.7
	5	NBBE-2	-	0.87	1.12	1.22	-	11316.44	10866.63	10853.33	-	618.2	609.9	585.8
		RBBE-4	-	1.02	1.19	1.41	-	16204.19	13035.33	13699.00	-	896.0	734.9	755.2
		CRBBE-4	-	0.95	1.24	1.38	-	16865.46	15191.26	14205.46	-	926.6	845.6	777.2
64 bits	2	NBBE-2	2.13	1.50	2.00	1.93	32414.76	33075.77	32381.78	29135.51	1521	1546	1519	1258
		RBBE-4	2.24	1.78	1.96	1.99	48937.62	49468.29	50097.11	37070.56	2332	2349	2387	1643
		CRBBE-4	2.19	1.46	1.99	2.05	50800.68	54205.48	52081.47	51764.93	2401	2525	2441	2433
	3	NBBE-2	-	1.24	1.68	1.82	-	35691.88	33316.23	31405.56	-	1862	1745	1635
		RBBE-4	-	1.53	1.82	1.97	-	51622.89	44016.88	42408.91	-	2637	2241	2131
		CRBBE-4	-	1.42	1.74	1.98	-	54540.11	52447.75	47676.24	-	2762	2626	2416
	5	NBBE-2	-	1.11	1.39	1.55	-	38162.49	32318.20	34394.86	-	2199	1859	1995
		RBBE-4	-	1.28	1.46	1.75	-	46740.46	40813.98	45787.38	-	2577	2337	2691
		CRBBE-4	-	1.23	1.51	1.76	-	57956.88	44993.37	52365.03	-	3265	2578	3070

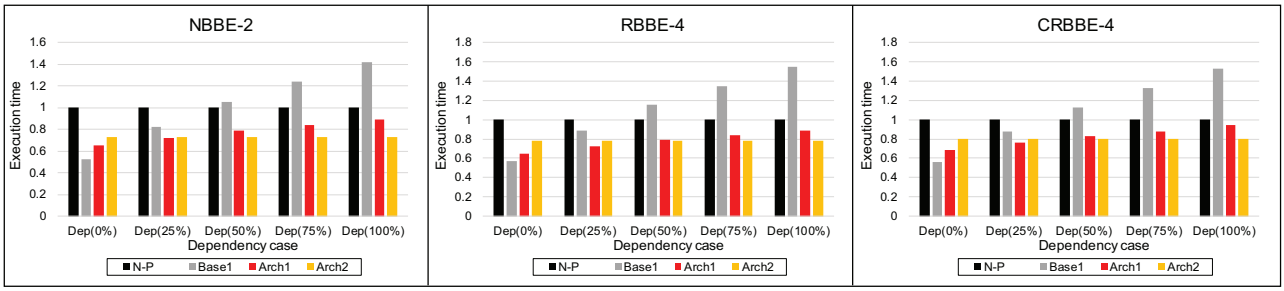

Figure 6. Comparison of the execution times of the 64-bit 5-stage N-P, Base, Arch1, and Arch2 designs. Dep (r) denotes that $r\%$ of the multiplications are dependent. All the execution times are scaled to the N-P designs.

and Arch2 designs because the latter have the clock period overheads. If the dependency goes up, however, the Arch1 and Arch2 designs will have shorter execution than the Base designs. We present actual execution times in Section IV-B. Regarding the number of pipeline stages, it is possible to increase the number of stages further, but the overhead due to additional flip-flops will go up. In addition, the size of the multiplexers and routing complexity in the Arch1 and Arch2 designs will also go up if full fast forwarding paths are to be added. Thus, increasing the pipeline depth over five might be unrealistic.

B. Execution Time

The actual execution time of a given sequence of multiplications depends on both the clock period and the dependency patterns of the multiplications. Thus, we also compared the total execution times for different sequences of multiplications. Our simulation methodology is as follows:

- Five sequences of 10,000 64-bit multiplications are randomly generated. Each sequence is named Dep (r).
- We assign two random integers d_1 and d_2 ($0 \leq d_1, d_2 \leq 4$) to the multiplications. d_1 and d_2 are the dependency distances for the multiplicand and multiplier, respectively, of each multiplication. For example, ($d_1 = 0, d_2 = 3$) means that the multiplicand is independent and the multiplier has a D-3 dependency.
- When we generate d_1 and d_2, we controlled the ratio (r) of the number of dependent multiplications to the number of independent multiplications. Sequence Dep (r) means that r of the 10,000 multiplications in the sequence are dependent. The five sequences are Dep (0%), Dep (25%), Dep (50%), Dep (75%), and Dep (100%).
- For each multiplication simulation, we obtain the end time T_e. Then, we obtain the actual execution time from $T_e \cdot T_{clk}$ for each design where T_{clk} is the clock period of the design in Table II.

Figure 6 shows the execution times of the 64-bit N-P and 5-stage Base, Arch1, and Arch2 designs for NBBE-2, RBBE-4, and CRBBE-4. For the independent multiplications (Dep (0%)), all the pipelined designs have shorter execution times than the N-P designs. In addition, the Base designs have the shortest execution times because they do not have multiplexers and carry-propagate adders

in the carry-save addition stages. As the dependency ratio increases, however, the execution times of the Base designs go up significantly and are 40% to 53% longer than the N-P designs due to the clock period overhead. On the other hand, the Arch1 and Arch2 designs still have shorter execution times than the N-P designs by almost 5% to 35%. Especially, the Arch2 designs consistently show approximately 20% and 30% shorter execution times than the N-P designs. Thus, we clearly see the impact of the fast forwarding path architecture on the throughput improvement. On the other hand, the execution times of the Arch1 designs go up as the dependency ratio increases. This is because more Type11 dependencies are added to the multiplications as the dependency ratio goes up.

C. Area and Power

Table II also shows the area and power consumption of the designs. The Base designs have larger area than the N-P designs for both 32- and 64-bit multipliers. However, some of the Arch1 and Arch2 designs such as the 64-bit 5-stage Arch1 designs have smaller area than the N-P designs. The reason is that the delay values are unevenly distributed throughout the five stages, so the synthesis tool uses small cells for the stages that have large timing margin. Comparing NBBE-2, RBBE-4, and CRBBE-4, the NBBE-2 designs have the smallest area because they have the simplest architecture. On the other hand, the CRBBE-4 designs have the largest area. We also observe similar trends for the power consumption. The Base designs consume about 15% to 58% more power than the N-P designs because of the additional flip-flops for pipelining. However, the Arch1 and Arch2 designs have lower power overhead than the Base designs. In addition, some of the Arch1 and Arch2 designs consume less power than the N-P designs. This is because the Arch1 and Arch2 architectures use carry-propagate adders for partial result generation, which helps reduce the number of partial product terms.

V. Conclusion

In this paper, we developed intra-unit fast-forwarding architectures to design high-throughput multipliers. The main idea of the fast-forwarding architectures is to generate partial results in the pipeline stages of the multipliers and forward them to previous pipeline stages to resolve data dependencies. The simulation results show that the multipliers with the fast-forwarding paths achieve up to 35% execution time reduction.

Acknowledgment

This work was supported by the Defense Advanced Research Projects Agency (DARPA) Young Faculty Award under Grant D16AP00119.

References

[1] Y. He and C.-H. Chang, "A New Redundant Binary Booth Encoding for Fast 2^n-Bit Multiplier Design," in *IEEE Trans. on Circuits and Systems*, vol. 56, no. 6, 2009, pp. 1192–1201.

[2] J. Rupley, J. King, E. Quinnell, F. Galloway, K. Patton *et al.*, "The Floating-Point Unit of the Jaguar x86 Core," in *Proc. IEEE Int. Symp. on Computer Arithmetic*, Apr. 2013, pp. 7–16.

[3] X. Cui, W. Liu, X. Chen, Earl E. Swartzlander Jr., and F. Lombardi, "A Modified Partial Product Generator for Redundant Binary Multipliers," in *IEEE Trans. on Computers*, vol. 65, no. 4, Apr. 2016, pp. 1165–1171.

[4] J. D. Bruguera, "Radix-64 Floating-Point Divider," in *Proc. IEEE Int. Symp. on Computer Arithmetic*, Jun. 2018, pp. 84–91.

[5] W.-C. Yeh and C.-W. Jen, "High-Speed Booth Encoded Parallel Multiplier Design," in *IEEE Trans. on Computers*, vol. 49, no. 7, Jul. 2000, pp. 692–701.

[6] F. Elguibaly, "A Fast Parallel Multiplier-Accumulator Using the Modified Booth Algorithm," in *IEEE Trans. on Circuits and Systems – II: Express Briefs*, vol. 47, no. 9, Sep. 2000, pp. 902–908.

[7] J.-Y. Kang and J.-L. Gaudiot, "A Simple High-Speed Multiplier Design," in *IEEE Trans. on Computers*, vol. 55, no. 10, Oct. 2006, pp. 1253–1258.

[8] S.-R. Kuang, J.-P. Wang, and C.-Y. Guo, "Modified Booth Multipliers with a Regular Partial Product Array," in *IEEE Trans. on Circuits and Systems – II: Express Briefs*, vol. 56, no. 5, May 2009, pp. 404–408.

[9] A. D. Booth, "A Signed Binary Multiplication Technique," in *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, Jan. 1951, pp. 236–240.

[10] C. S. Wallace, "A Suggestion for a Fast Multiplier," in *IEEE Trans. on Electron Computers*, Feb. 1964, pp. 14–17.

[11] N. Takagi, H. Yasuura, and S. Yajima, "High-Speed VLSI Multiplication Algorithm with a Redundant Binary Addition Tree," in *IEEE Trans. on Computers*, vol. 32, no. 9, Sep. 1985, pp. 789–796.

[12] G. Govindu, P. Gupta, S. Pitkethly, and G. J. Rozas, "Execution Pipeline Data Forwarding," in *US Patent US9569214B2*, 2017.

[13] L. Dadda, "Some Schemes for Parallel Multipliers," in *Alta Frequenza*, vol. 34, Mar. 1965, pp. 349–356.

[14] H. Makino, Y. Nakase, H. Suzuki, H. Morinaka, H. Shinohara *et al.*, "An 8.8-ns 54x54-Bit Multiplier with High Speed Redundant Binary Architecture," in *IEEE Journal of Solid-State Circuits*, vol. 31, no. 6, 1996, pp. 773–783.

[15] N. Besli and R. G. Deshmukh, "A Novel Redundant Binary Signed-Digit(RBSD) Booth's Encoding," in *Proc. IEEE SoutheastConf*, Apr. 2002, pp. 426–431.

[16] D. P. Agrawal and T. R. N. Rao, "On Multiple Operand Addition of Signed Binary Numbers," in *IEEE Trans. on Computers*, vol. c27, no. 11, Nov. 1978, pp. 1068–1070.

[17] Nangate, "Nangate 45nm Open Cell Library," http://www.nangate.com.

[18] P. M. Kogge and H. S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," in *IEEE Trans. on Computers*, vol. C-22, no. 8, Aug. 1973, pp. 786–793.