Dependency-Resolving Intra-Unit Pipeline Architecture for High-Throughput Multipliers

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

***Abstract*—In this paper, we propose two dependency-resolving intra-unit pipeline architectures to design high-throughput multipliers. Simulation results show that the proposed multipliers achieve approximately $2.4\times$ to $4.3\times$ execution time reduction at a cost of 4.4% area and 3.7% power overheads for highly-dependent multiplications.**

I. Introduction

Compute-intensive applications such as data compression, encryption/decryption, and matrix multiplications require high-throughput architectures for short execution time. In addition, various emerging architectures such as three-dimensional stacked memory provide very wide memory bandwidth and significantly reduces the memory bottleneck problem [1], [2]. Thus, memory-intensive applications such as video signal and graph processing also need high-throughput architectures to process large data loaded from memory in time.

One of the most challenging problems in the high-performance architecture design is how to resolve data dependencies [3], [4]. To resolve data dependencies, researchers have proposed various hardware techniques such as data forwarding (bypassing), out-of-order execution, and register renaming [5]. Pipelining is also commonly used for throughput improvement in almost all high-performance architectures [6]–[8]. In addition, superscalar and very long instruction word (VLIW) architectures increase the instruction-level parallelism for further throughput improvement.

Multi-cycle instructions such as multiplications and divisions take multiple clock cycles to finish an instruction. If two k-cycle instructions ($k > 1$) of the same type are executed in a non-pipelined execution unit, the total execution time is $2k$ cycles. On the contrary, if they are sequentially executed in a pipelined execution unit that has k pipeline stages, the total execution time is $k+1$, which is significantly shorter than $2k$. If there is a dependency between the instructions, however, the total execution time becomes $2k$ and the intra-unit pipeline does not improve the throughput.

In this paper, we propose dependency-resolving intra-unit pipeline architectures for high-throughput multipliers. Briefly speaking, the architectures generate partial results in the intra-unit pipeline stages and forward them to their earlier pipeline stages so that dependent instructions can be injected into the execution unit with a minimum wait time [9]. The intra-unit forwarding paths are similar to system-level forwarding (bypassing) paths, but the forwarding occurs inside the execution unit. Design of the intra-unit forwarding paths requires reorganization of the architecture of the execution unit or development of a new architecture. In this paper, we reorganize multiplier architectures to resolve dependencies.

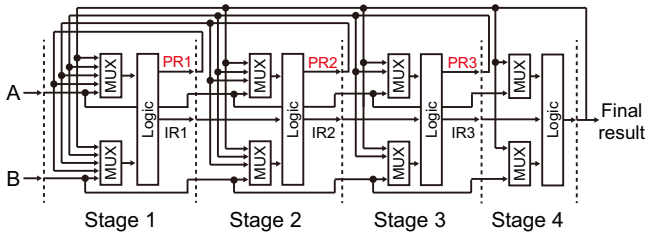

Fig. 1. A four-stage dependency-resolving pipeline architecture. IR: Intermediate result. PR: Partial result.

II. Dependency-Resolving Pipeline Architecture

In this section, we present general dependency-resolving intra-unit pipeline architectures.

A. Dependency-Resolving Pipeline Architecture

The main idea of the dependency-resolving pipelined execution unit architecture is to generate partial results in the pipeline stages and forward them to their earlier pipeline stages so that dependent instructions can be injected into the unit with a minimum wait time. Figure 1 shows a four-stage dependency-resolving pipeline architecture. In the figure, two inputs A and B are passed to the first stage, in which the execution unit generates an intermediate result $IR1$ and a partial result $PR1$. $IR1$, A, and B are passed to the second stage for further computation, whereas $PR1$ is forwarded to the first stage so that the next instruction can use it if required. Similarly, the second stage generates an intermediate result $IR2$ and a partial result $PR2$. $IR2$, A, and B are passed to the third stage, whereas $PR2$ is forwarded to the first and second stages. We control the multiplexers in the pipeline stages to choose proper operands.

In Figure 1, all the stages generate partial results and all the partial results are forwarded to all the earlier pipeline stages. However, it is also possible that only a few pipeline stages generate partial results and some of the partial results are forwarded to only a few earlier stages.

B. Dependency Types and Distance, Forwarding Distance

We define three dependency types for a two-operand operation $Z = OP_1 * OP_2$ as follows:

978-3-9819263-2-3/DATE19/©2019 EDAA

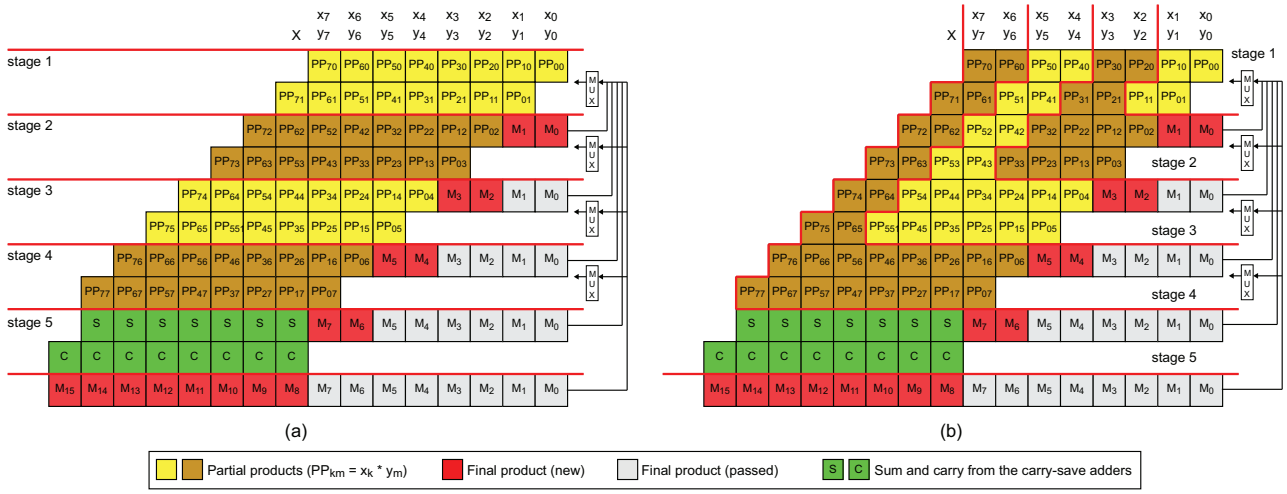

Fig. 2. Dependency-resolving 8-bit pipelined multipliers. (a) Arch1 resolving Type01/10 dependencies. (b) Arch2 resolving Type01/10/11 dependencies.

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

We also define the dependency distance for a dependent instruction as follows. Suppose instructions are executed sequentially starting from i_1. If instruction i_m is dependent on instruction i_k ($k < m$), the dependency distance of instruction i_m is $m - k$. For example, if three instructions i_1, i_2, i_3 are executed in this order and i_3 is dependent on i_1, the dependency distance from i_3 to i_1 is 2. We call a Type01 or Type10 dependency of distance d a Dd dependency. A Type11 dependency has two dependency distances. We also define the forwarding distance of a forwarding path in Figure 1 as follows. Suppose the partial result of the pipeline stage m is forwarded to the pipeline stage k ($k \leq m$). Then, the forwarding distance of the forwarding path is $m - k + 1$. We call a forwarding path whose forwarding distance is f a Df forwarding path in this paper.

III. Dependency-Resolving Pipelined Multiplier

In this section, we propose two dependency-resolving pipelined multipliers.

A. Arch1: Type01/10-Dependency-Resolving Architecture

Figure 2(a) shows an 8-bit 5-stage pipelined multiplier with full forwarding paths. The two inputs are $x_{[7:0]}$ and $y_{[7:0]}$ and the final output is $M_{[15:0]}$. The first four stages are carry-save addition stages and each of them adds two 8-bit partial products using carry-save adders. The fifth stage is a carry-propagated addition stage. The four stages accumulates the eight partial products and the output of the fourth stage has two operands to be added. Thus, the fifth stage adds the two operands using a high-speed carry-propagated adder. The first stage calculates $x_{[7:0]} * y_{[1:0]}$ and generates intermediate sum and carry bits and $M_{[1:0]}$. The intermediate sum and carry bits are passed to the next stage and added to the next two partial products. If the second multiplication is dependent on the first multiplication, the multiplexer in the first stage chooses $M_{[1:0]}$ for the second multiplication.

The second stage calculates $x_{[7:0]} * y_{[3:2]}$ and generates intermediate sum and carry bits and $M_{[3:2]}$. Notice that $M_{[1:0]}$ is passed from the first stage to the second stage so that it can be forwarded to the first stage to handle D1 dependencies. Similarly, the third and the fourth stage calculate $x_{[7:0]} * y_{[5:4]}$ and $x_{[7:0]} * y_{[7:6]}$ and generate $M_{[5:4]}$ and $M_{[7:6]}$ as their partial results, respectively. The partial results are forwarded to all the previous stages to resolve D1, D2, D3 dependencies.

Suppose the first instruction is $i_1 : Y_{[15:0]} = A_{[7:0]} * B_{[7:0]}$ and the second instruction is $i_2 : Z_{[15:0]} = C_{[7:0]} * Y_{[7:0]}$. Table I shows the progresses of i_1 and i_2 in the multiplier stages. At time 0, i_1 is injected to the first stage of the multiplier with two independent inputs $A_{[7:0]}$ and $B_{[7:0]}$. Although the whole bits of B are available at time 0, only $B_{[1:0]}$ is necessary for the partial product accumulation in the first stage. The first stage generates a partial result $Y_{[1:0]}$ and a partial product accumulation result. The former is used to resolve dependencies if required and the latter is used for the partial product accumulation in the later stages. At time 1, $Y_{[3:2]}$ is newly generated from adding the previous accumulation result and $A_{[7:0]} * B_{[3:2]}$. $Y_{[3:2]}$ is a newly-generated result and $Y_{[1:0]}$ is passed from the first stage, so $Y_{[3:0]}$ is available at the end of the second stage. At the same time, i_2 is injected into the first stage of the multiplier because the first stage requires only $C_{[7:0]}$ and $Y_{[1:0]}$ as its inputs. As shown in the table, $Y_{[1:0]}$ is forwarded from the second stage. i_2 generates $Z_{[1:0]}$ in the first stage for its partial result, which could be used for other instructions dependent on i_2. Overall, i_2 is injected into the multiplier at time 1, so the total execution time of i_1 and i_2 is six clock cycles.

B. Arch2: Type01/10/11-Dependency-Resolving Architecture

Figure 2(b) shows an 8-bit 5-stage pipelined multiplier with full forwarding paths that can resolve Type01, Type10, and Type11 dependencies. For a given instruction $M_{[15:0]} =$

TABLE I
TYPE01. $i_1 : Y_{[15:0]} = A_{[7:0]} * B_{[7:0]}$. $i_2 : Z_{[15:0]} = C_{[7:0]} * Y_{[7:0]}$. T: TIME. S: STAGE. I: REQUIRED INPUTS. P: PARTIAL RESULTS.

T	S	I	P	S	I	P
0	1	$A_{[7:0]}$, $B_{[1:0]}$	$\mathbf{Y_{[1:0]}}$	-	-	-
1	2	$A_{[7:0]}$, $B_{[3:2]}$	$\mathbf{Y_{[3:0]}}$	1	$C_{[7:0]}$, $\mathbf{Y_{[1:0]}}$	$Z_{[1:0]}$
2	3	$A_{[7:0]}$, $B_{[5:4]}$	$\mathbf{Y_{[5:0]}}$	2	$C_{[7:0]}$, $\mathbf{Y_{[3:2]}}$	$Z_{[3:0]}$
3	4	$A_{[7:0]}$, $B_{[7:6]}$	$\mathbf{Y_{[7:0]}}$	3	$C_{[7:0]}$, $\mathbf{Y_{[5:4]}}$	$Z_{[5:0]}$
4	5	-	$Y_{[15:0]}$	4	$C_{[7:0]}$, $\mathbf{Y_{[7:6]}}$	$Z_{[7:0]}$
5	-	-	-	5	-	$Z_{[15:0]}$

TABLE II
TYPE11. $i_1 : X_{[15:0]} = A_{[7:0]} * B_{[7:0]}$. $i_2 : Y_{[15:0]} = C_{[7:0]} * D_{[7:0]}$. $i_3 : Z_{[15:0]} = X_{[7:0]} * Y_{[7:0]}$ T: TIME. S: STAGE. I: REQUIRED INPUTS. P: PARTIAL RESULTS.

T	I	P	I	P	I	P
0	$A, B_{[1:0]}$	$\mathbf{X_{[1:0]}}$	-	-	-	-
1	$A, B_{[3:0]}$	$\mathbf{X_{[3:0]}}$	$C, D_{[1:0]}$	$\mathbf{Y_{[1:0]}}$	-	-
2	$A, B_{[5:0]}$	$\mathbf{X_{[5:0]}}$	$C, D_{[3:0]}$	$\mathbf{Y_{[3:0]}}$	$\mathbf{X}, \mathbf{Y_{[1:0]}}$	$Z_{[1:0]}$
3	$A, B_{[7:0]}$	$\mathbf{X_{[7:0]}}$	$C, D_{[5:0]}$	$\mathbf{Y_{[5:0]}}$	$\mathbf{X}, \mathbf{Y_{[3:0]}}$	$Z_{[3:0]}$
4	-	$X_{[15:0]}$	$C, D_{[7:0]}$	$\mathbf{Y_{[7:0]}}$	$\mathbf{X}, \mathbf{Y_{[5:0]}}$	$Z_{[5:0]}$
5	-	-	-	$Y_{[15:0]}$	$\mathbf{X}, \mathbf{Y_{[7:0]}}$	$Z_{[7:0]}$
6	-	-	-	-	-	$Z_{[15:0]}$

$x_{[7:0]} * y_{[7:0]}$, the first stage computes $x_{[1:0]} * y_{[1:0]}$ using carry-save adders and generates intermediate sum and carry bits and $M_{[1:0]}$. The second stage computes $x_{[3:0]} * y_{[3:0]}$ with the result of $x_{[1:0]} * y_{[1:0]}$, which is passed from the first stage, and generates partial result $M_{[3:0]}$ where $M_{[3:2]}$ is a new result and $M_{[1:0]}$ is passed from the first stage. Similarly, the third stage computes $x_{[5:0]} * y_{[5:0]}$ with the result of $x_{[3:0]} * y_{[3:0]}$ and generates $M_{[5:0]}$. Finally, the fourth stage computes $x_{[7:0]} * y_{[7:0]}$ with the result of $x_{[5:0]} * y_{[5:0]}$ and generates $M_{[7:0]}$. The fifth stage is the same as the fifth stage of Arch1 in Figure 2(a). Table II shows an example in which three instructions $i_1 : X_{[15:0]} = A_{[7:0]} * B_{[7:0]}$, $i_2 : Y_{[15:0]} = C_{[7:0]} * D_{[7:0]}$, $i_3 : Z_{[15:0]} = X_{[7:0]} * Y_{[7:0]}$ are executed.

IV. SIMULATION RESULTS

We designed Arch1 and Arch2 using VHDL and synthesized them using Synopsys Design Compiler and the Nangate 45nm Open Cell Library. We used Kogge-Stone adders for the carry-propagated additions [10]. We also designed Arch1 and Arch2 without the forwarding paths to compare overheads and execution times of Arch1 and Arch2. We call the Arch1 and Arch2 designs without the forwarding paths Base1 and Base2 designs, respectively.

A. Clock period, Area, and Power

Table III show the clock periods, area, and power of the 8-, 16-, 32-, and 64-bit Arch1 and Arch2 designs. S is the number of carry-save addition stages and the last stage is always a carry-propagated addition, so the total number of pipeline stages of each design is $S + 1$. The numbers shown in the parentheses are the ratios between Arch1 (or Arch2) and Base1 (or Base2) designs, respectively.

The clock periods of the Arch1 designs are slightly longer than those of the Base1 designs on average. The clock period

TABLE III
CLOCK PERIOD, AREA, AND POWER CONSUMPTION OF N-BIT $S + 1$-STAGE ARCH1 AND ARCH2 DESIGNS. S IS THE NUMBER OF CARRY-SAVE ADDITION STAGES. THE NUMBERS IN THE PARENTHESES SHOW THE VALUES OF THE ARCH1 (ARCH2) DESIGNS SCALED TO THE BASE1 (BASE2) DESIGNS.

		Arch1 (resolving Type01/10 dependencies)		
N	S	Clock (ns)	Area (um^2)	Power (uW)
8 bits	1	1.14 (1.036)	672 (1.062)	28 (1.062)
	2	0.70 (1.045)	790 (1.055)	36 (1.039)
	4	0.50 (1.220)	1,080 (1.055)	53 (1.064)
16 bits	1	2.25 (1.023)	2,320 (1.018)	99 (1.020)
	2	1.29 (1.032)	2,537 (1.048)	123 (1.051)
	4	0.77 (1.116)	3,092 (1.050)	171 (1.082)
32 bits	1	4.45 (1.007)	8,713 (1.017)	371 (1.017)
	2	2.42 (1.021)	8,961 (1.019)	438 (1.021)
	4	1.40 (1.094)	9,852 (1.015)	555 (0.996)
64 bits	1	9.02 (1.023)	30,668 (0.960)	1,372 (0.974)
	2	4.62 (1.013)	33,444 (0.997)	1,682 (0.998)
	4	2.50 (1.050)	35,079 (0.997)	2,073 (1.009)
Geo. mean		(1.055)	(1.024)	(1.027)
		Arch2 (resolving Type01/10/11 dependencies)		
N	S	Clock (ns)	Area (um^2)	Power (uW)
8 bits	1	1.01 (1.098)	825 (1.083)	32 (1.092)
	2	0.87 (1.101)	912 (1.079)	39 (1.048)
	4	0.66 (1.179)	1,228 (1.126)	57 (1.102)
16 bits	1	1.77 (1.035)	2,978 (1.061)	117 (1.076)
	2	1.33 (1.090)	2,955 (1.045)	138 (1.088)
	4	0.95 (1.092)	3,554 (1.088)	173 (1.054)
32 bits	1	3.17 (1.033)	11,169 (1.010)	430 (1.019)
	2	2.07 (1.030)	10,697 (1.045)	467 (1.007)
	4	1.37 (1.079)	11,526 (1.046)	582 (1.020)
64 bits	1	6.25 (1.016)	40,883 (1.007)	1,588 (1.010)
	2	3.84 (1.143)	37,615 (0.934)	1,792 (0.978)
	4	2.18 (1.058)	40,064 (1.014)	2,058 (0.960)
Geo. mean		(1.078)	(1.044)	(1.037)

overhead is due to the multiplexer delays. However, the multiplexer delays are much shorter than those of the carry-save adders or the carry-propagated adders, so the actual overhead compared to the Base1 designs is 5.5% on average and 22.0% in the worst case. We observe similar trends in the Arch2 designs. The clock period overhead is 7.8% on average and 17.9% in the worst case compared to the Base2 designs.

The Arch1 and Arch2 designs also have area overhead compared to the Base1 and Base2 designs. For example, the area overhead of the Arch1 designs is 2.4% on average and 6.2% in the worst case compared to the Base1 designs. Similarly, the area overhead of the Arch2 designs is 4.4% on average and 12.6% in the worst case compared to the Base2 designs. The area overhead is also due to the multiplexer area. In addition, as the number of stages goes up, we should insert more multiplexers and more complex multiplexers. For example, the 8-bit 3-stage Arch1 design (N=8,S=2) needs two 4-bit 3:1 multiplexers and two 4-bit 2:1 multiplexers, but the 8-bit 5-stage Arch1 design (N=8,S=4) needs two 2-bit 5:1 multiplexers, two 2-bit 4:1 multiplexers, two 2-bit 3:1 multiplexers, and two 2-bit 2:1 multiplexers. Similarly, the Arch1 and Arch2 designs have higher power consumption than the Base1 and Base2 designs. The power overheads of the Arch1 and Arch2 designs are 2.7% and 3.7% on average and 8.2% and 10.2% in the worst case compared to the Base1 and Base2 designs, respectively. The power overhead is also mainly due to the power consumption of the multiplexers.

B. Execution Time

Adding the forwarding paths to the pipelined designs slightly increases the clock periods of the designs due to the multiplexer delay overhead. In this simulation, therefore, we compare actual execution times of the Arch1, Arch2, Base1, Base2 designs. The simulation methodology is as follows. First, we randomly generate a set of 10,000 multiplication instructions for an N-bit architecture having $S+1$ pipeline stages where S stages are carry-save addition stages and the last stage is a carry-propagated addition stage. When we generate the instructions, some of the instructions are dependent (Type01/10/11) instructions and the other instructions are independent instructions. The ratio of the number of dependent instructions to the total number of instructions is r and we vary r from 0% (i.e., all the instructions are independent) to 100% (i.e., all the instructions except the first instruction are dependent). In addition, we assign a random integer between 1 and S to each Type01/10 dependent instruction and two random integers between 1 and S to each Type11 dependent instruction for their dependency distances. We compute the total execution time for a set of instructions executed in a multiplier by $e \cdot T$ where e is the end time of the last instruction and T is the clock period of the multiplier.

Figure 3 shows the execution times of the Base1, Base2, Arch1, and Arch2 designs. For the 8-bit multiplications, if all the instructions are independent (Dep(0%)), the execution times of the Arch1 and Arch2 designs are 7% to 60% longer than the Base1 designs due to the clock period overhead. As the dependency ratio increases from 25% to 100%, however, the execution times of the Arch1 and Arch2 designs decrease significantly. If all the instructions are dependent (Dep(100%)), the execution times of the Arch1 and Arch2 designs are shorter than the Base1 designs by 39% and 41%, respectively. We find similar trends in the 32-bit multiplier designs. When all the instructions are dependent, the Arch1 and Arch2 designs have 58% and 77% execution time reduction ($2.4\times$ and $4.3\times$ throughput improvement), respectively, compared to the Base1 designs. In summary, as more dependent instructions are executed, the proposed architectures achieve shorter execution time.

V. Conclusion

In this paper, we have proposed two pipelined multipliers resolving data dependencies. The main idea is to generate partial results in the pipeline stages and forward them to earlier pipeline stages so that instructions dependent on the results can be injected into the multiplier with a minimum delay penalty. Arch1 resolves Type01/10 dependencies, whereas Arch2 resolves all types of dependencies. The Arch1 and Arch2 designs achieve $2.4\times$ to $4.3\times$ execution time reduction at a cost of 4.4% area overhead and 3.7% power overhead on average for highly-dependent instructions.

Acknowledgement

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

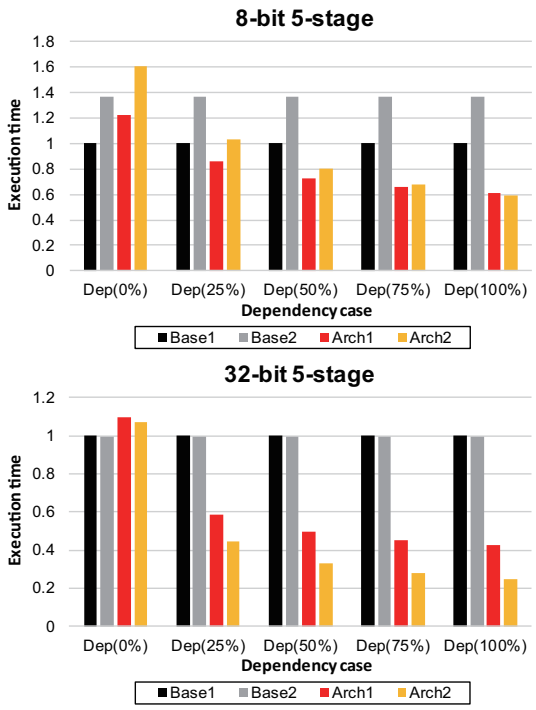

Fig. 3. Comparison of the execution times of the Base1, Base2, Arch1, and Arch2 designs for 8-bit 5-stage (N=8,S=4) and 32-bit 5-stage (N=32,S=4) multipliers. Dep(r) denotes that $r\%$ of the instructions are dependent instructions. All the execution time values are scaled to the Base1 designs.

References

[1] D. H. Woo, N. H. Seong, D. L. Lewis, and H. S. Lee, "An Optimized 3D-Stacked Memory Architecture by Exploiting Excessive, High-Density TSV Bandwidth," in *Proc. IEEE Int. Symp. on High-Performance Computer Architecture*, Jan. 2010, pp. 429–440.

[2] D. H. Kim, K. Athikulwongse, M. B. Healy, M. M. Hossain, M. Jung *et al.*, "Design and Analysis of 3D-MAPS (3D Massively Parallel Processor with Stacked Memory)," in *IEEE Trans. on Computers*, vol. C64, no. 1, Jan. 2015, pp. 112–125.

[3] M. H. Lipasti and J. P. Shen, "Exceeding the Dataflow Limit via Value Prediction," in *Proc. Annual Int. Symp. Microarchitecture*, Dec. 1996, pp. 226–237.

[4] A. Sodani and G. S. Sohi, "Dynamic Instruction Reuse," in *Proc. IEEE Int. Symp. on Computer Architecture*, Jun. 1997, pp. 194–205.

[5] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2011.

[6] P. Y. T. Hsu, J. T. Rahmeh, E. S. Davidson, and J. A. Abraham, "TIDBITS: Speedup via time-delay bit-slicing in ALU design for VLSI technology," in *Proc. IEEE Int. Symp. on Computer Architecture*, 1985, pp. 29–35.

[7] A.Hartstein and T. R. Puzak, "The Optimum Pipeline Depth for a Microprocessor," in *Proc. IEEE Int. Symp. on Computer Architecture*, May 2002.

[8] A. Hartstein and T. R. Puzak, "Optimum Power/Performance Pipeline Depth," in *Proc. Annual Int. Symp. Microarchitecture*, Dec. 2003, pp. 117–126.

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

[10] 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.