ECTI Transactions on Computer and Information Technology
Journal homepage: https://ph01.tci-thaijo.org/index.php/ecticit/
Published by the ECTI Association, Thailand, ISSN: 2286-9131

Model-Based Design Optimization using CDFG for Image Processing on FPGA

Surachate Chumpol[1], Panadda Solod[2], Krerkchai Thongnoo[3] and Nattha Jindapetch[4]

ABSTRACT

As the automotive industry moves toward autonomous driving and ADAS (Advanced Driver Assistance Systems), Model-Based Design (MBD) is a practical design methodology. It can be used to develop rapid prototyping by using MATLAB and Simulink. The MBD method still has limitations for handling complex models. This paper uses the Control Data Flow Graph (CDFG), an intermediate representation for analyzing complex algorithms, so that suitable optimizations for image processing applications can be implemented on an FPGA. The experimental results show that the proposed CDFG method improved both the area and speed of the edge detection case study compared with the MathWorks Vision HDL toolbox.

Article information:

Keywords: Model-Based Design (MBD), Control Data Flow Graph (CDFG), image processing, FPGA

Article history:
Received: April 18, 2023
Revised: June 24, 2023
Accepted: October 5, 2023
Published: October 21, 2023 (Online)

DOI: 10.37936/ecti-cit.2023174.252417

1. INTRODUCTION

Autonomous driving and ADAS are vital technologies for next-generation vehicles. These significantly requires data from many sensors, such as radar, LiDAR, and vision sensors. Even high-performance microcontrollers cannot handle the massive amounts of data in real-time. For these requirements, high-performance parallel processors such as FPGAs (Field Programmable Gate Arrays) have the potential to support this automotive trend.

Recently, FPGA applications have become more complicated due to their higher degree of system integration. Traditional FPGA development methods by hardware description language (HDL such as VHDL or Verilog) take time to develop and validate, and they also need exceptional skilled FPGA engineers. Recently, high-level synthesis (HLS) tools offer more straightforward and faster development solutions for the increasing needs of complex systems.

In literature [1], [2], and [3], the authors proposed alternative approaches by using a Model-Based Design (MBD) that is a practical design methodology and able to develop rapid prototyping by using MATLAB/Simulink. In literature [4] and [5], the authors proposed optimization, such as fixed-point optimization, to reduce the area consumption for a lower-cost FPGA board with limited resources. Many applications employed only a few conditions without nested loops [1], [2], [3], [4], and [5]. So, this literature cannot be a good candidate for complex models. On the other hand, a new approach used a control data flow graph (CDFG), which is an intermediate representation, to analyze complex algorithms and perform speed optimization at high-level abstractions such as C programming [6]. Recently, commercial tools are focusing on algorithm designs at high-level synthesis methodology for complex FPGA applications. There are limitations in that the MathWorks HDL coder cannot stream a loop if there are two or more nested loops at the same hierarchy level within another loop [7]. Some automated optimization tools allow designers to add pragmas and optimization directives to perform pipelining [8]. However, there is room to make the circuit even faster.

In this paper, we propose an alternative method to solve this issue. Model-Based Design (MBD) is one of the alternative methods that have the potential to solve complex models. We aim to investigate this method and find practical processes to develop FPGA applications by this method using high-level synthesis tools, such as the HDL coder provided by

[1,3]The authors are with Toyota Tsusho Nexty Electronics (Thailand) co., ltd., Bangkok, Thailand 10330, E-mail: surachate@th.nexty-ele.com and krerkchai.thongnoo@gmail.com
[2]The author is with Faculty of Industrial Education and Technology, Rajamangala University of Technology Srivijaya, Songkhla, Thailand 90000, E-mail: panadda.solod@gmail.com
[4]The author is with Department of Electrical Engineering, Faculty of Engineering, Prince of Songkla University, Songkhla, Thailand 90110, E-mail: nattha.s@psu.ac.th
[4]The corresponding author: nattha.s@psu.ac.th

MathWorks. Due to the excellent characteristics of MBD, we can simulate and prove our design concept in advance, so that developers can focus on algorithm design instead of FPGA implementation. For complex model synthesis, such as image processing, we need to investigate and apply CDFG to achieve the optimized goals.

2. MODEL-BASED DESIGN

In the automotive industry, MBD is widely used to develop embedded software as shown in Figure 1. After the algorithm has been designed according to the specification requirements, the corresponding behavioral model is created and validated in the first stage. The implementation model can then be created and validated. Next, the quality code is generated the actual hardware.

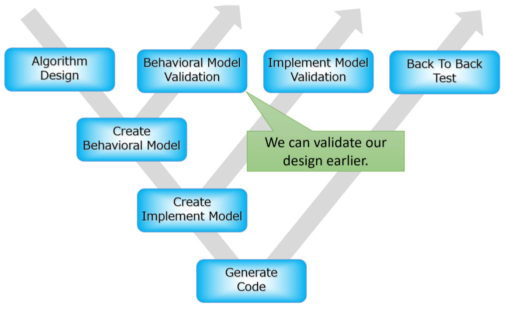


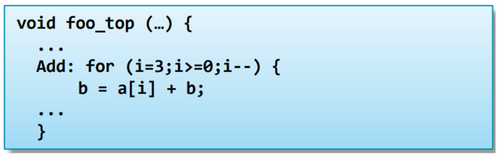
Fig.1: *Model-Based Design approach in the V model.*

Finally, a system test or back-to-back test is performed. As the design can be validated earlier, development time can be shortened, and there is no need to wait for hardware development.

2.1 MBD Speed Optimizations

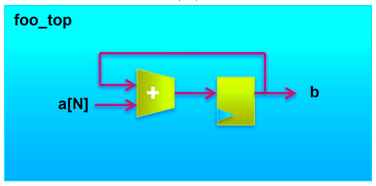
The critical path is the longest delay path that determines the speed of the design. When the target speed is not satisfied, speed optimizations can improve the timing of the design on the target FPGA by optimizing the critical path. The critical path estimation is obtained from the static timing analysis using timing data from target-specific timing databases. Model-level speed optimizations have two significant methods: pipelining and loop unrolling.

Loop unrolling parallels multiple instances of the loop body to accelerate the loop operations. A loop in function foo_top of Figure 2(a) uses one adder and one register, shown in Figure 2(b), taking four cycles. When the loop unrolling used four adders and one register, shown in Figure 2(c), the latency became one cycle. The faster design needs more parallel resources. Therefore, the trade-off between speed and area must be done according to the options illustrated in Figure 2(d). There are three options: option 1 uses one adder takes four cycles; option 2 uses two adders takes two cycles; and option 3 uses four adders takes one cycle.

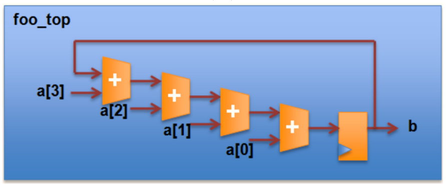


```
void foo_top (…) {
  ...
  Add: for (i=3;i>=0;i--) {
      b = a[i] + b;
  ...
}
```

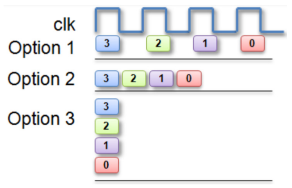
(a)



(b)

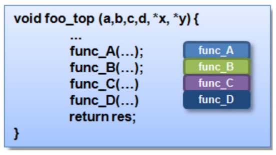


(c)



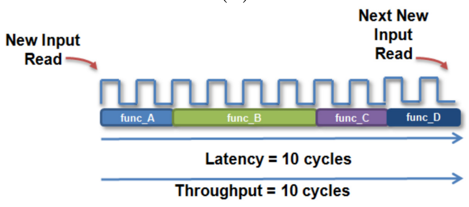
(d)

Fig.2: *Loop unrolling [9].*

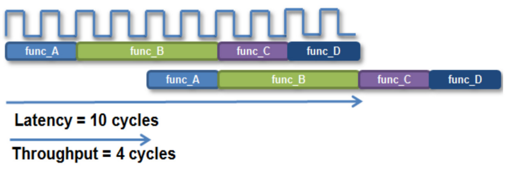


```
void foo_top (a,b,c,d, *x, *y) {
    ...
    func_A(...);
    func_B(...);
    func_C(...)
    func_D(...)
    return res;
}
```
func_A
func_B
func_C
func_D

(a)

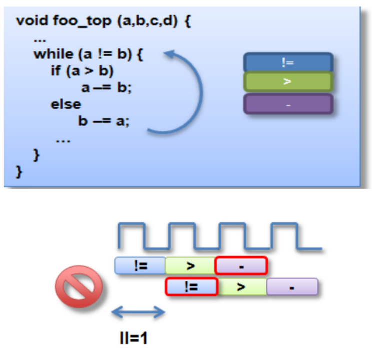


(b)



(c)

Fig.3: *Function pipelining [9].*



```
void foo_top (a,b,c,d) {
  ...
  while (a != b) {
    if (a > b)
        a -= b;
    else
        b -= a;
     ...
  }
}
```

Fig.4: *Feedback algorithm limits pipelining [9].*

Registers are inserted at the input, output, or ports of certain function blocks to perform pipelining. The critical path is then determined by the most prolonged delay among two registers, resulting in a higher throughput for the design. Figure 3 (a) shows the foo_top function, which contains four functions sequentially operated, resulting in a latency of 10 cycles and a throughput of 10 cycles, as shown in Figure 3 (b). After pipelining, the throughput was optimized to four cycles, as shown in Figure 3 (c).

2.2 Problem Statements

Although pipelining and loop unrolling can effectively optimize the design, they remain obstacles to the actual design. Algorithms with nested loops or nested if-else conditions are prone to complicate critical path estimation. Algorithms with feedback outputs to inputs may limit full pipelining and loop unrolling. As shown in Figure 4, the comparison operation (a != b) must wait the subtraction (b −= a) of the previous iteration. Therefore, such problems state that a more straightforward analysis method is needed to determine data dependence and suggest the optimal solution.

3. OPTIMIZATION PROCEDURE

Improving system performance by using the MBD approach can be done by improving algorithms, reducing some operations, parallelizing, unrolling loops, and pipelining. In this paper, we propose using CDFG as a medium to analyze the system to help select the proper optimization method for each subsystem.

3.1 CDFG Analysis Procedure

CDFGs are intermediate representations that combine control flows and data flow graphs (DFGs) in the same graph. A CDFG and a DFG are explicitly defined in [10, 11] as definitions 1 and 2. A CDFG contains one DFG or more and a control flow. A CDFG may contain one DFG or groups of DFGs without control flow.

Definition 1: A DFG-unit is a triple $\Omega = (V, E, O)$, where V is a set of operation nodes, E is a set of edges between operation nodes, and O is a set of operations defined for each node. The edge set E corresponds with the transfer of data from one operation to another.

Definition 2: A CDFG-unit is a 4-tuple $\Gamma = (X, Y, Z, E)$, where X is a control node, Y is a conditional node, Z is the set of nodes in Γ except for X and Y, where Z is said a child block of the Γ's control node. E is a set of edges between nodes.

Figure 5 shows the CDFGs of the loop (repeated from C=0 to C<Max) and the if-else conditions. A CDFG has at least one DFG, which is the data flow via additions, multipliers, bit-wise operations, or any arithmetic and logic operations. The DFG contains no conditions such as loops or if-else statements. A CDFG containing inner CDFGs and DFGs is called a hierarchical CDFG.

The hierarchical CDFG raises the observability of data dependence among DFGs within the CDFG or among CDFGs, so that scheduling and pipelining optimizations can be done. Therefore, this paper proposed a procedure to analyze a CDFG, which is shown in Figure 6. The data dependence among the CDFGs at the same hierarchy level suggests performing either pipelining or parallel processing. Similarly, the data dependence among the operations in the same DFG suggests performing either pipelining or loop unrolling that allows parallelism. Among DFGs in an if-else CDFG, if there is a DFG in another branch containing the same operation, resource sharing can be performed to minimize the area. For example, DFG1 and DFG2 of the if-else CDFG in Figure 5 can share resources because they never occur simultaneously. For each DFG, algorithm optimizations and partial optimizations can be further considered.

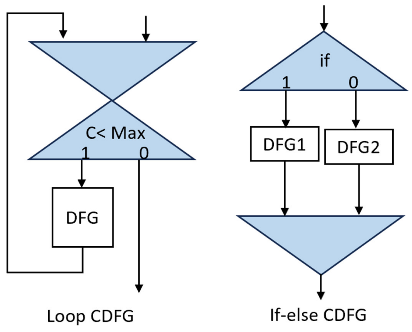


Fig.5: *CDFGs of the loop and the if-else statement.*

To illustrate the data dependence of the CDFG analysis procedure in Figure 6, a part of an algorithm can be decomposed into the data dependence of CDFGs shown in Figure 7. Given algorithms described in C/C++ are decomposed into CDFGs. The arrows denote the data flow from one CDFG to another. If two CDFGs have no dependence data, no arrow is drawn. CDFG L0_1 and CDFG L0_2 are the

```
Procedure: CDFG Analysis
 1: For each loop or if statement, do
 2:     create CDFG LX_N with node-level assignment
 3: For each CDFG LX_i where i=1, 2, …, N in the
      same level
 4:     Determine data dependence
 5:     if (data dependence)
 6:         Pipelining
 7:     else
 8:         perform parallel
 9: For each DFG in a CDFG
10:     algorithm optimization
11:     partial optimization
12: For each if-else CDFG
13:     resource sharing among DFGs
```

Fig.6: *CDFG analysis procedure.*

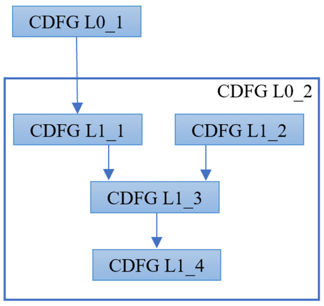


Fig.7: *Data dependence of CDFGs.*

outermost CDFGs, namely parent CDFGs. CDFG L0_1 has data dependence with CDFG L0_2 because the results of CDFG L0_1 are used in CDFG L1_1, which is an inner CDFG of CDFG L0_2, namely a child CDFG. In this case, pipelining may be applied to increase speed. CDFG L1_1 and CDFG L1_2 are independent, so that both CDFGs can parallel. The pipelining may be applied between CDFG L1_1 to CDFG L1_3, CDFG L1_1 to CDFG L1_3, and CDFG L1_3 to CDFG L1_4 because of data dependence on each other.

CDFGs can easily highlight nested loops, nested if-else, and feedback loops inhibiting optimizations. Paths containing feedback loops are unbreakable paths. When there is feedback data from the outputs to any operations, pipelining cannot be performed. The algorithm should be re-designed to avoid such feedback loops. In the nested loop case, we suggest adjusting the algorithm to flatten the loop. In the nested if-else case, we suggest using the switch-case statement instead.

3.2 Algorithm Optimization

Some algorithms in DFGs may be too complicated to implement in a low-budget hardware platform. For example, Gradient and Threshold in (1) combine the differences between the horizontal (Gx) and vertical (Gy) axis differences to have a single gradient magnitude (G). The square root function is not suitable for FPGA. Therefore, G is approximated by using the absolute function for the FPGA implementation, as shown in (2).

$$|G| = \sqrt{Gx^2 + Gy^2} \tag{1}$$

$$|G| = |Gx| + |Gy| \tag{2}$$

3.3 Partial Optimization

Some operations in DFGs may be reduced if they are multiplied by zero. For example, a Sobel filter uses two 3x3 kernels: the first kernel is used to determine the horizontal difference (Gx), and the second is used to determine the vertical difference (Gy), as shown in Figure 8. The multiplications can be reduced from 18 to 12 because six operations multiplied by zero. In addition, some multiplications that multiply the same coefficient can share a multiplier. For example, -1×P11 + -2×P12+ -1×P13 gives the same result as -1×(P11+P13) + -2×P12 with reduced multipliers.

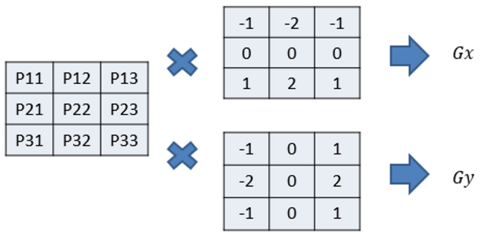


Fig.8: *Sobel filter using two 3×3 kernels (Gx/Gy).*

3.4 Pipeline balancing

Pipelining is a standard method to accelerate data-dependence operations inside a CDFG or among CDFGs. Pipelining is performed by adding registers between CDFGs. CDFGs containing nested if-else, nested loop, or feedback prevent pipeline. As an example, Figure 9 shows three data-dependence CDFGs with the total processing time of 10.214 ns (3.703 ns+1.048 ns+5.463 ns), obtaining a throughput of 1/10.214 or 97.9 mega-pixels per second (MPPS). Adding registers between CDFGs to perform pipelining can increase throughput to 1/5.463 or 183.05 MMPS. The throughput should align with the worst-case stage delay, i.e., the maximum delay, for the correct answer.

On the other hand, if we decompose the CDFGs into DFGs, we see data-dependence DFGs. Only the last part is two DFGs under a sub-CDFG with an if-else condition, as shown in Figure 10. We balance the pipeline by adding registers between DFGs that give almost the same delay at each stage, as shown in Figure 10. Finally, we obtain the throughput of 1/3.500 or 285.71 MPPS.

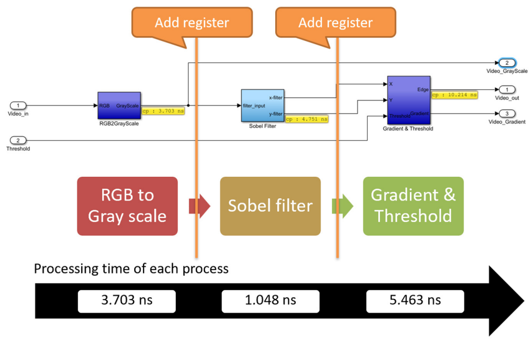


Fig.9: *Pipelining.*

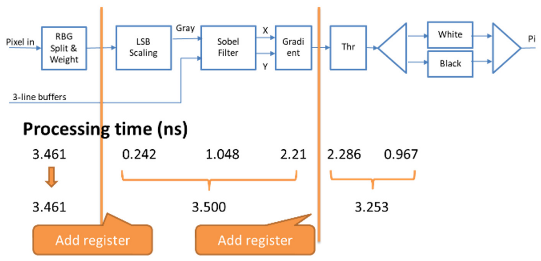


Fig.10: *Pipeline balancing.*

Note that since image processing applications are structurally diverse in their algorithms, some structures, such as nested if-else, nested loop, or feedback, cannot be pipelined. CDFG analysis should be done to visualize this structure prior to pipelining.

4. EXPERIMENTAL RESULTS

To demonstrate the usefulness of the proposed CDFG analysis, an edge detection was selected as a case study. CDFGs can express the data dependence among RGB, Sobel, and Gradient and Threshold approximation process blocks. The CDFG analysis method can suggest a more suitable optimization method. Here, the experimental results are presented in two studies. The first study is a case study of edge detection that illustrates how to apply partial optimizations and a CDFG-based analysis to minimize processing time and resources. The second study compared an edge-detection model using the proposed method and the Vision HDL toolbox, which is a toolbox designed for image processing using an architecture optimized for HDL provided by MATLAB/Simulink.

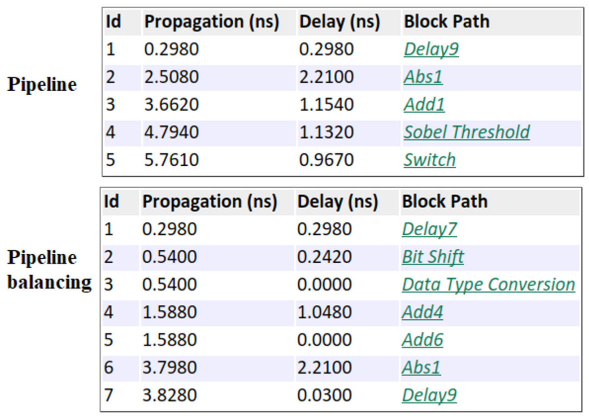
4.1 Case Study: Edge Detection

After CDFG analysis, the CDFG of the functions is decomposed into DFGs and sub-CDFGs so that we can consider the suitable optimization for each DFG. Figure 11 shows the edge detection algorithm in Simulink model. Figure 11 (a) is a grayscale subsystem model using an unsigned integer. Figure 11(b) is a Sobel filter sub-system with the partial optimization described Section 3.3. Figure 11 (c) is Gradient and Threshold approximation with the algorithm optimization described Section 3.2. Figure 11(d) shows images from the original input image, grayscale, gradient after Sobel filter, and edge detection. If there is data dependence among DFGs, pipeline balancing will be applied.

Table 1: *Synthesized Results of Sobel filter.*

Results	Original model	Optimized model
Multipliers	18	0
Adders/Subtractors	16	10
Propagation delay (ns)	3.796	1.092

Table 2: *Synthesized Results of Pipelining.*

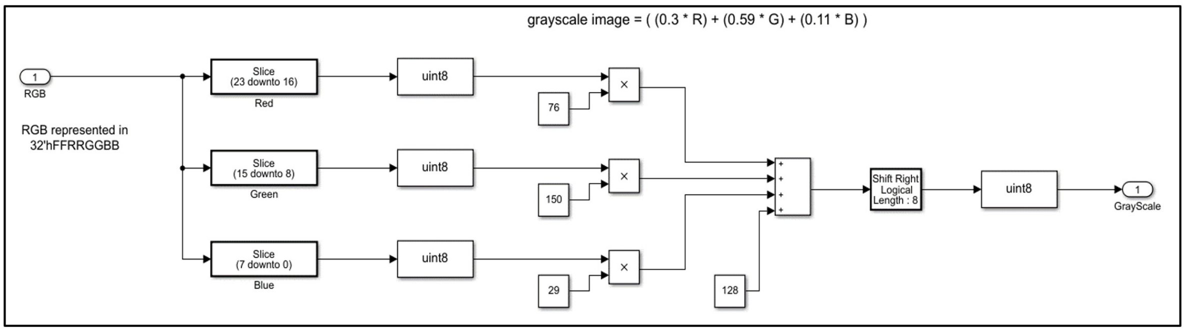


	Id	Propagation (ns)	Delay (ns)	Block Path
Pipeline	1	0.2980	0.2980	Delay9
	2	2.5080	2.2100	Abs1
	3	3.6620	1.1540	Add1
	4	4.7940	1.1320	Sobel Threshold
	5	5.7610	0.9670	Switch
Pipeline balancing	1	0.2980	0.2980	Delay7
	2	0.5400	0.2420	Bit Shift
	3	0.5400	0.0000	Data Type Conversion
	4	1.5880	1.0480	Add4
	5	1.5880	0.0000	Add6
	6	3.7980	2.2100	Abs1
	7	3.8280	0.0300	Delay9

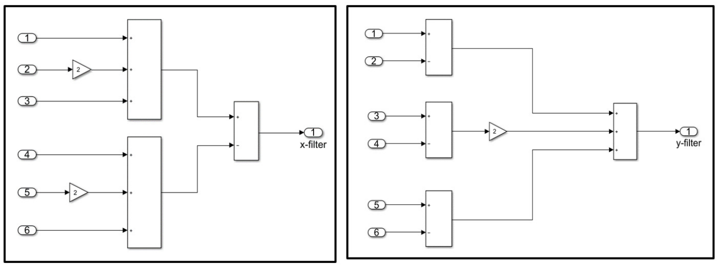
Once the HDL codes of the models in Figure 11 were generated by the HDL coder, the Vivado was used to synthesize the circuit for the target device (AMD's xa7z010clg225-1). Table 1 shows the synthesized results of the Sobel filter. The partial optimization explained in Section 3.3 can minimize the resources to a zero multiplier and fewer adders. In addition, operations multiplied by -2, 2, 1, and -1 are further reduced. The shift-right operation is used instead of a multiplied-by-2 operation. The multiplied-by-1 operation uses an adder. Consequently, the circuit also improved speed from 3.796 ns to 1.092 ns.

Table 2 shows the processing time improvement of the pipeline balancing of Figure 10 compared to the pipelining of Figure 9. The results of the three processing DFGs have the output image of each DFG, as illustrated in Figure 11. The processing time of three DFGs are as follows: 1) Converting a color image to a grayscale image or a grayscale image (RGB to Grayscale) 3.703 ns, 2) Sobel filter 1.048 ns and 3) Gradient & Threshold 5.463 ns. These delays formed the critical path of 10.214 ns, as shown in Figure 9.

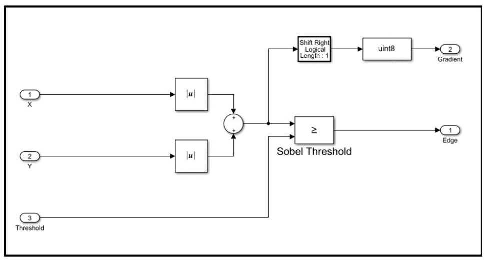
Pipeline optimization reduced overall critical path execution time from 10.214 ns to 5.761 ns, which is 5.463 ns of processing time from the third DFG (Gradient & Threshold) plus the 0.2980 ns delay of added registers for pipelining. The improvement in processing time achieved by using pipeline optimization. The insertion of pipeline-stage registers between DFGs re-


grayscale image = ((0.3 * R) + (0.59 * G) + (0.11 * B))
1 RGB
RGB represented in 32'hFFRRGGBB
Slice (23 downto 16) Red
Slice (15 downto 8) Green
Slice (7 downto 0) Blue
uint8
76
150
29
128
Shift Right Logical Length : 8
uint8
1 GrayScale

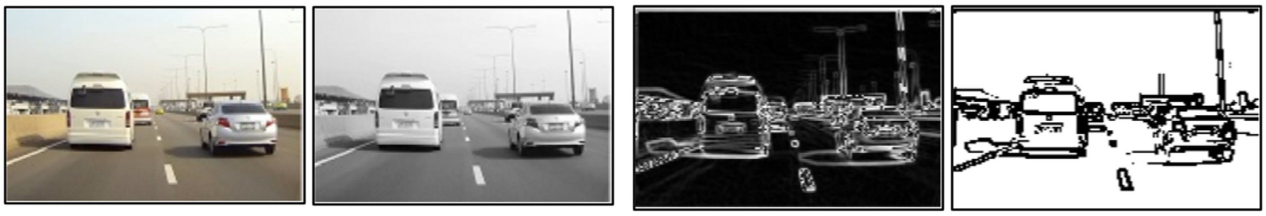
(a) Grayscale sub-system in Simulink model.



(b) Sobel filter sub-system in Simulink model (x-axis and y-axis).

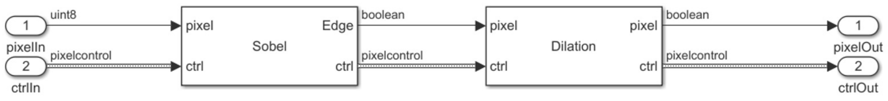


(c) Gradient and Threshold approximation in Simulink model.

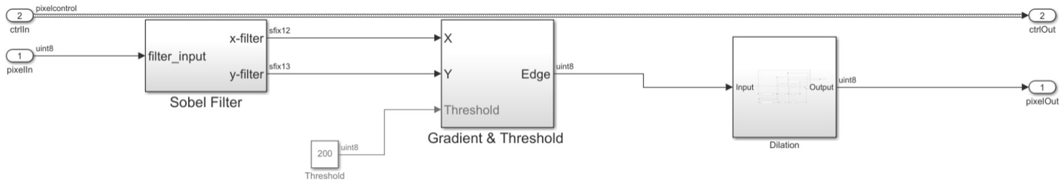


(d) Images of Original, Grayscale, Gradient after Sobel filter and Edge detection.

Fig.11: *Pipeline balancing.*

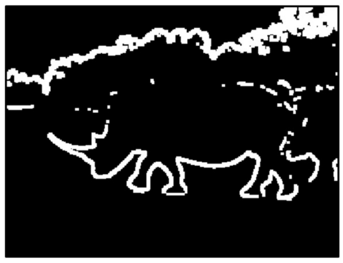


(a) Vision HDL Library model.

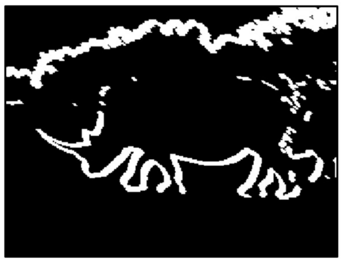


(b) Implementation I.

(c) Input Imagge.

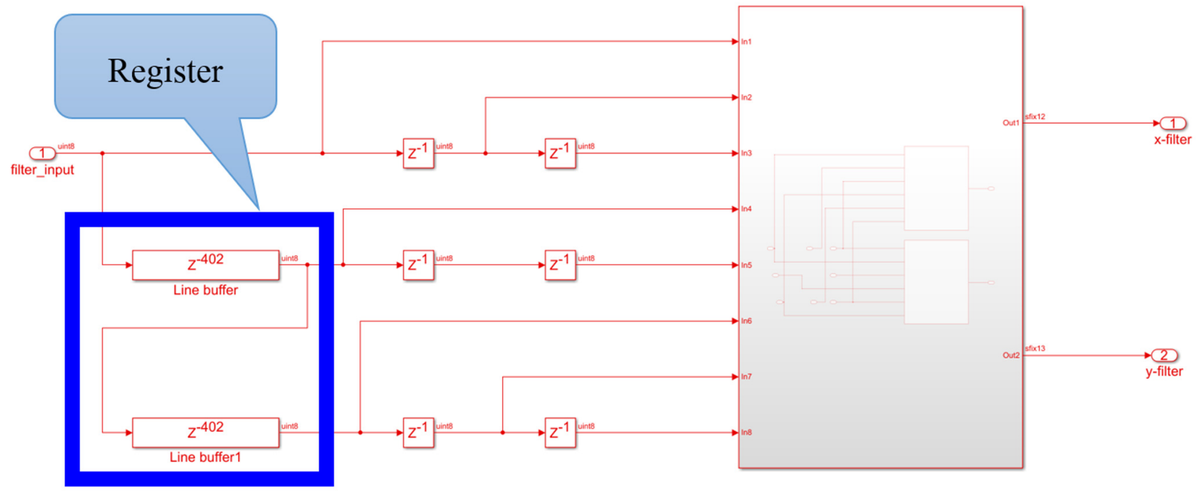


(d) Output Image: Vision HDL.

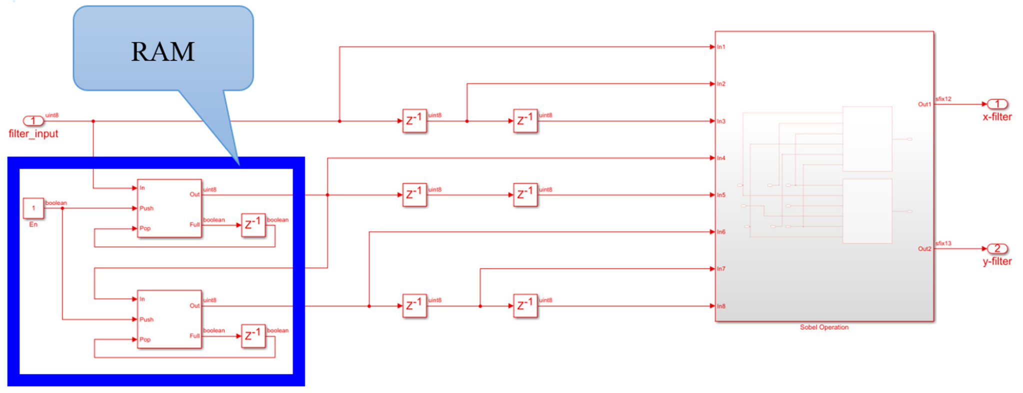


(e) Output Image: Implementation I.

Fig.12: *Edge Detection applications.*



(a) Implementation I with register buffering.



(b) Implementation II with RAM buffering.

Fig.13: *Edge Detection applications using register buffering and RAM buffering.*

sults in a reduction in the critical path processing time. Note that the static timing analysis tool mentioned in Section 2 determines the critical path from the maximum delay of the data transfer between any two registers, i.e., the data transfer at the register transfer level (RTL).

4.2 Comparison with the Vision HDL Toolbox

This section reports the comparative results of edge detection implementations between the proposed optimization method and the Vision HDL toolbox. In the Vision HDL toolbox, each multiplier has two pipeline stages on each input and two on each output. The adder is a pipelined tree structure. HDL code generation uses symmetric, unity, or zero-value coefficients to reduce the number of multipliers [12]. However, pipeline balancing is not considered.

Figure 12 shows the implementations of edge detection algorithms. Figure 12(a) is the implementation model using the Vision HDL library, and Figure 12(b) is the proposed implementation model. From the input image of Figure 12(c), the output images of both implementations are almost the same, as shown in Figures 12(d) and (e).

Table 3: *Resource usage results.*

Resources	**Vision HDL Toolbox**	**Imp. I (Register)**	**Imp. II (RAM)**
Multipliers	2	0	0
Adders/Subtractors	39	13	29
Registers	413	1615	**44**
Total 1-Bit Registers	1792	12920	**316**
RAMs	4	0	**4**
Multiplexers	126	13	53
I/O Bits	23	30	30

Table 4: *Resource utilization on Zynq Z-7010 FPGA.*

Resources	**Available**	**Vision HDL Toolbox**	**Imp. I (Register)**	**Imp. II (RAM)**
Slice LUTs	17600	715	417	426
LUT as Logic	17600	695	183	426
LUT as Memory	6000	20	234	0
Slice Registers	35200	1480	489	238
Register as Flip Flop	35200	1480	489	238
Block RAM (RAMB18)	120	4	0	4
DSPs	80	2	0	0

Table 5: *Critical Path Estimation.*

	Id	Propagation (ns)	Delay (ns)	Block Path
Vision HDL Toolbox	1	0.2980	0.2980	Edge Detector
	2	11.1030	10.8050	Dilation
Imp. I (Reg.)	1	0.2980	0.2980	Delay4
	2	1.3900	1.0920	Add3
	3	3.6000	2.2100	Abs1
	4	4.7540	1.1540	Add
	5	5.8860	1.1320	Sobel Threshold
	6	6.8530	0.9670	Switch
	7	6.8830	0.0300	Delay1
Imp. II (RAM)	1	3.5510	3.5510	HDL FIFO
	2	4.5990	1.0480	Add1
	3	4.5990	0.0000	Gain
	4	4.5990	0.0000	Add3
	5	6.8090	2.2100	Abs1
	6	7.9630	1.1540	Add
	7	9.0950	1.1320	Sobel Threshold
	8	10.0620	0.9670	Switch
	9	10.4750	0.4130	HDL FIFO

The resource usage results are shown in Table 3. The first column shows the resources in the number of multipliers, adders/subtractors, registers, total 1-bit registers, RAMs, multiplexers, and I/O bits. The second column shows the results of the Vision HDL library model. The third column shows the results of the proposed implementation using registers, namely Implementation I. The last column shows the results of the proposed implementation using RAMs, namely Implementation II.

Similarly, Table 4 shows the resource utilization on an AMD Xilinx Zynq Z-7010 FPGA. Adders/Subtractors and multiplexers were implemented by LUTs. Registers were implemented by flip-flops. RAMs were implemented by block RAMs. Multipliers and consecutive adders were implemented by DSP blocks, also known as Multiply And Accumulate (MAC) blocks.

We can see that the proposed Implementation I and Implementation II used no multiplier, whereas the Vision HDL library implementation used two multipliers. Adders/subtractors are three times reduced in the proposed Implementation I, and ten adders/subtractors decreased in the proposed Implementation II. Multiplexers drastically decreased in both proposed implementations compared to the Vision HDL library implementation.

The proposed Implementation I, shown in Figure 13 (a), uses many registers to store data instead of RAMs. Alternatively, we can choose RAMs as in the proposed Implementation II, as shown in Figure 13(b). RAMs affect a lower speed, whereas registers affect a larger area.

Table 5 shows the critical path estimation results. The first column shows the number of components in the critical path. The second column shows the path propagation delays, and the third shows the delay of each component in the critical path. The last column is the component name, i.e., block path. The critical path estimation of the Vision HDL library implementation is the top path of Table 5. The edge detector block consumed 0.2980 ns, plus the dilation block consumed 10.8050 ns, for 11.1030 ns. The proposed Implementation I consumed 6.883 ns, and the proposed Implementation II consumed 10.475 ns.

Obviously, the proposed implementation I achieved the fastest circuit. Since FPGAs have a limited number of registers, a trade-off between speed and area should be considered.

5. CONCLUSIONS

In this paper, a method for MBD optimization using CDFG for image processing has been presented. The CDFG analysis procedure raised the observability of data dependencies among DFGs within the CDFG so that suitable optimizations could be obtained. For each DFG, partial optimization techniques such as algorithm optimization and operation reduction can minimize resource usage and increase speed. CDFG was also used to analyze the processing time of each process to segment each process into small DFGs to average the processing time of each DFG group equally (balancing process time) to improve the critical path when inserting a register between each group of DFGs. The pipeline balancing resulted in better throughput than pipeline optimization among the process blocks.

In this paper, the examples are not diverse enough to find ways to improve the efficiency of digital systems. The example is also not complex enough. In future work, more samples and more complex systems will be investigated.

ACKNOWLEDGEMENT

This research has been supported by the Intelligent Electronics for the Automotive Electrical Industry program, Next Generation Higher Education, Ministry of Higher Education, Science, Research, and Innovation. The project is a collaboration between the Department of Electrical Engineering, Faculty of Engineering, Price of Songkla University, and Toyota Tsusho Nexty Electronics (Thailand) Co., Ltd.

References

[1] J.C. Ting Hai, O. Chee Pun, and T. Wooi Haw, “Accelerating Video and Image Processing Design for FPGA using HDL Coder and Simulink,” in *2015 IEEE Conference on Sustainable Utilization and Development in Engineering and Technology (CSUDET)*, pp. 28-32, 2015.

[2] S. Titri, C. Larbes and K. Y. Toumi, “Rapid prototyping of PVS into FPGA: From Model-Based Design to FPGA/ASICs Implementation,” *2014*

9th International Design and Test Symposium (IDT), Algeries, Algeria, pp. 162-167, 2014.

[3] N. Othman, M. H. Jabbar, A. K. Mahamad and F. Mahmud, "Luo Rudy Phase I Excitation Modeling towards HDL Coder Implementation for Real-time Simulation," *2014 5th International Conference on Intelligent and Advanced Systems (ICIAS)*, Kuala Lumpur, Malaysia, pp. 1-6, 2014.

[4] N. Othman, F. Mahmud, A. Kadir Mahamad, M. Hairol Jabbar and N. Atiqah Adon, "Cardiac Excitation Modeling: HDL Coder Optimization towards FPGA Stand-alone Implementation," in *2014 IEEE International Conference on Control System, Computing and Engineering*, pp. 507-512, 2014.

[5] M. Besbes, S. H. Saïd and F. M'Sahli, "FPGA Implementation of High Gain Observer for Induction Machine Using Simulink HDL Coder," 2015 3rd International Conference on Control, Engineering & Information Technology (CEIT), Tlemcen, Algeria, pp. 1-6, 2015.

[6] S. Cheng, and J. Wawrzynek, "High-Level Synthesis with a Dataflow Architectural Template," in *2nd International Workshop on Overlay Architectures for FPGAs (OLAF2016)*, pp. 288-293, 2016.

[7] MathWorks, "Limitations for MATLAB Loop Optimization," *HDL coder user guide*, pp. 8-30, 2023.

[8] Advanced Micro Devices, *Vitis High-Level Synthesis User Guide (UG1399)*, `https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Creating-Additional-Solutions`, 2023.

[9] Xilinx, *Improving Performance*, 2016.

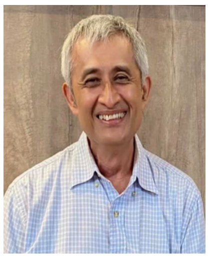
[10] E. Kim, J. Lee, and D. Lee, "Automatic Process-Oriented Asynchronous Control Unit Generation from Control Data Flow Graphs," *IEICE Trans. on Fundamentals*, vol. E84-A, no. 8, pp. 2014-2028, August 2001.

[11] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[12] The MathWorks, Inc., *Vision HDL Toolbox^{TM} Reference*, 2023.

Surachate Chumpol received the B.Eng. degree in EE from Prince of Songkla University, Thailand, in 1997, and the M.Eng. degree in EE from Prince of Songkla University, Thailand, in 2023. He is currently the General Manager of Software Development Department, Toyota Tsusho Nexty Electronics (Thailand) Co., Ltd., and the Director of Toyota Tsusho Denso Electronics (Thailand) Co., Ltd. His research interests include embedded software development, Advanced Driver Assistance Systems (ADAS), and Model Based Design (MBD).

Panadda Solod received the B.Eng. degree in Mechatronics Engineering from Prince of Songkla University (PSU), Thailand, in 2017, and the M.Eng. degree in Electrical Engineering from PSU, Thailand, in 2021. She is currently a lecturer at Faculty of Industrial Education and Technology, Rajamangala University of Technology Srivijaya. Her research interests include FPGAs, embedded systems, image processing, and Model Based Design (MBD).



Krerkchai Thongnoo received his B.Eng. in Electrical Engineering from PSU, Thailand in 1980. He received his M.Eng.Sci. in Computer Science and Ph.D. in Electrical Engineering and Computer Science, in 1987 and 1991 respectively, from University of New South Wales, Australia. He has been employed as Associate Professor at PSU where he retired in 2017. He is currently a research consultant at Toyota Tsusho Nexty Electronics (Thailand) Co., Ltd. (NETH) where his current research is mainly in embedded systems design.

Nattha Jindapetch received the B.Eng. degree in Electrical Engineering (EE) from Prince of Songkla University (PSU), Thailand, in 1993, the M.Eng. degree in information technology, and the Ph.D. degree in interdisciplinary course on advanced science and technology from the University of Tokyo, Japan, in 2000 and 2004, respectively. She is currently an Associate Professor with the Department of EE, PSU. Her research interests include FPGAs, embedded systems, Model-Based Design (MBD), and sensor networks.