VAST-LP: CLOCK GATING IN HIGH-LEVEL SYNTHESIS

BY

ZELEI SUN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

ABSTRACT

High-level synthesis (HLS) promises high-quality hardware with minimal development effort. In this thesis, we evaluate the current state-of-the-art HLS engine VAST and propose a method to generate clock-gating-friendly RTL code for downstream logic synthesis tools. We use one-hot-key encoding method to build the state transition in hardware, and we use the state registers along with main clock signal to generate subclock signals. By analyzing the usage of each register when the finite state machine is in different states, we assign the corresponding subclock signals to the register and reduce the unnecessary toggle of the registers when they are not in use. CHStone benchmarks in different application categories are used to verify the functionality and test the performance of the designs. The area and power data are measured using downstream commercial state-of-the-art tools during logic synthesis. We gain 5% to 20% dynamic power saving with -6% to 2% area increase.

To Xiaotian and my parents, for their love and support. To Prof. Chen for the guidance and help with my research.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

As the development of modern circuits advances, more transistors as well as complex heterogeneous architectures are integrated in to a single circuit, leading to a much more complex design in the design space. This increases the difficulty for hardware engineers to keep working at register-transfer level (RTL). The increased development time and weak ability for design space exploration lead people to move beyond RTL design and head into electronic system level (ESL) design methodology. It offers a great improvement in productivity for integrated circuit design [1]. High-level synthesis (HLS) performs the transition from high-level behavior languages (C,C++,etc.) to RTL descriptions (Verilog, VHDL, etc.). It seeks to create a bridge from the high-productivity software paradigm to the hardware world. It is hard to fully appreciate the advantage and the potential brought about by HLS. Development cost and time have been dramatically reduced by HLS engines, not to mention the time and effort saved on the test and debug phase. Yet these designs can still leverage the performance and efficiency of hardware implementation. However, with less effort invested in the details of a hardware design, the door to many customized optimizations may be shut.

In the meantime, as the technology node keeps scaling down driven by the Moore's law, the limitations begin to appear. Power consumption is becoming the dominant constraint over area limitations and performance requirements. In the past, mobile and portable platforms required major consideration for power. However, recently, people's demand for low-power computation is no longer limited to these devices. Even for computationally intense high-performance applications, such as convolution neural network applications on supercomputers, people are taking low-power solutions into consideration [2]. How to design an energy-friendly circuit is then a critical question. Clock gating [3] has been a promising and important method to reduce dynamic power consumption. The key of clock gating is reducing the dynamic power consumption by eliminating switching of the logic gates in the combinational logics when the logic gates are supposed to be idle. This is done through fixing the input to these gates. Registers feeding to these inputs will be disabled, resulting in their outputs being fixed at a certain state. For CMOS logics, only static power is consumed when the output does not change. Compared to the dynamic power consumed by these gates when the output is flipping, the static power is usually smaller. This is also the reason why the switch activity is very essential to accurately estimate circuit power. Clock gating effectively reduces the switching activity of both combinational and sequential logics in some situations, thereby effectively reducing the power consumption in these cases. There are some scenarios where some writes to a register would not change the behavior for down-stream variables or have identical value in consecutive clock cycles. From the downstream perspective, we call this kind of condition Observability Don't Care. On RTL level, there is a good deal of research on power reduction [4, 5].

As the design methodology goes from RTL to ESL, we found new opportunities to further utilize the clock-gating technology. HLS flow can be summarized in the following stages:

1. Compiling. In this stage, the HLS engine will compile the input language, analyze the relationship between instructions, and convert the operations into CDFG (control-datapath flow graph). In CDFG, there are scheduling units which stand for the operations and registers.

2. Scheduling. The sequence of operations is decided at this stage as we put the scheduling units into target clock cycles based on the resources we have. The state transition graph is also generated at this stage to define the finite state machine we have.

3. Binding. This is the stage where we bind the actual resources to the operations

and variables. Some of the operations may share the same function module, and some of the variables may share the same register.

4. RTL code generation. In this stage we generate the RTL code based on the given binding and scheduling information.

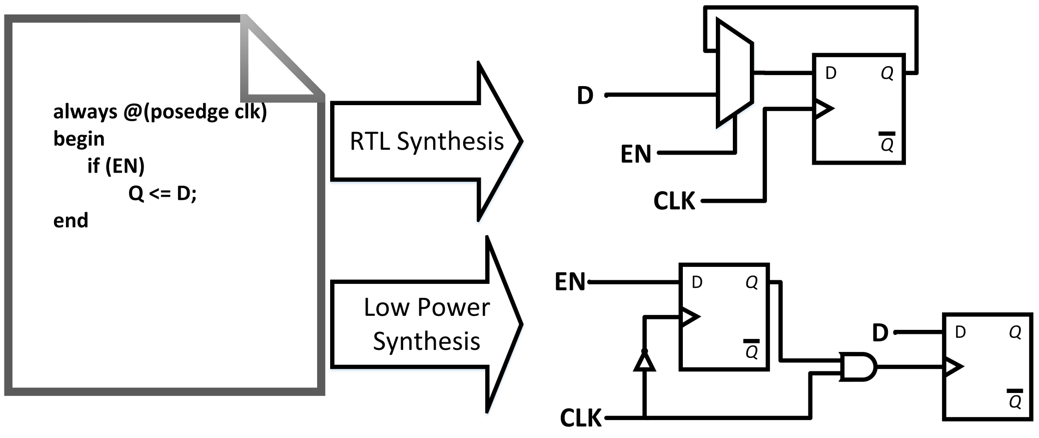
One way of clock gating comes from the finite state machine. After the binding stage, we have got the information of which registers will be used at what state. Based on this, we can actually let the clock signal for the registers toggle only when the finite state machine is at the corresponding state. In general there are many states in the HLS generated RTL code, and with only one of them on at a given time, we should be able to shut down the dynamic power for the registers which are used at other states and thus save a good amount of power.

In this thesis, we will implement the above clock-gating feature in VAST, which is one of the most powerful state-of-the-art HLS engines. We will explore different optimization choices and analyze the pros and cons for this technology.

CHAPTER 2

BACKGROUND

Clock gating has been introduced in circuit designs for years. The basic idea is to identify the conditions where the clock signal for the registers can be changed so that it does not toggle every clock cycle. In this way, we can avoid unnecessary register value changes and reduce the dynamic power consumption from the flip-flop as well as the resulting combinational logic circuit. Two basic clock-gating choices are very popular in the design. One is based on the enable signal for some registers. As shown in Fig. 2.1, in low-power design we can change the multiplexer into an AND gate to reduce the toggle rate. Another way of clock gating is using the XOR gate to detect data changes as shown in Fig. 2.2.



```
always @(posedge clk)
begin
    if (EN)
        Q <= D;
end
```

Figure 2.1: An example of enable based clock gating.

More advanced clock gating at RTL level has been well studied. Wu et al. [3] discussed the clock gating in a sequential circuit. The conditions for gating the

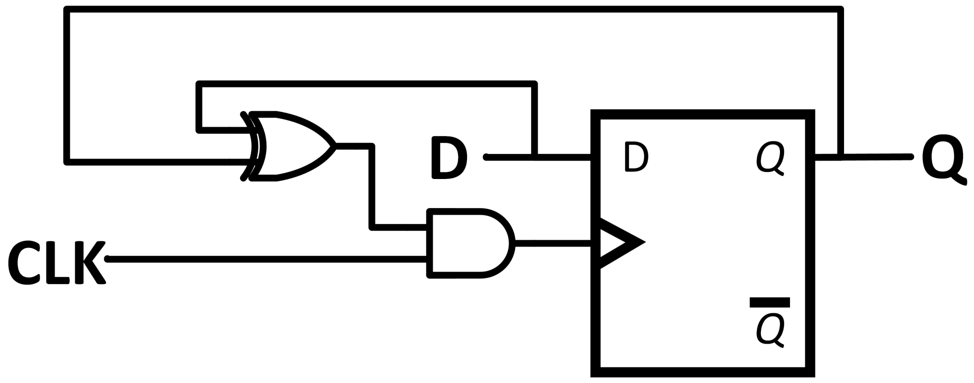


Figure 2.2: An example of XOR gate based clock gating.

master lock are derived by the relation between the clock and the transition behaviors of the triggered flip-flops, and a quaternary variable is used to model the behavior. Observability Don't Care based clock gating is also an important approach.

However, due to the limitation of the RTL design, these clock-gating methods mostly focus on local optimization without global consideration, and thus miss power saving opportunities on the micro-architecture level. With the help of behavior vision in HLS, we will be able to explore the relationships between modules and different function blocks more easily, and provide clock gating based on a higher-level abstraction.

CHAPTER 3

CLOCK GATING IN HLS

Although clock-gating techniques have been well defined in the RTL level implementation, there is not much optimization being explored in the area of high-level synthesis. However, there are actually more opportunities for clock gating in HLS. In the era of RTL code, some engineers will also add some clock-gating blocks at the module level manually. In high-level synthesis with the extra information on the dependency between different modules, we can analyze the hardware design at a micro-architectural level and thus implement clock gating in this level automatically.

The HLS flow will analyze a high-level input language such as C and SystemC, and build the state transition graph based on the compiled backend information. In the meantime, it also generates the resource list such as the registers that are in use as well as the combinational logic circuits. With this information in well written structures, we can easily find out the states where a register is used, and generate a subclock signal that only toggles at that state. We will then use this subclock as the input clock signal for that register; in this way, the register will only toggle when it is needed. Given the fact that in HLS there are hundreds of states and only one of them will be on at each clock cycle, the chance of power saving is actually very high.

Resource sharing is very common nowadays in the hardware design. A register can be used in different states. We can analyze and detect these cases, and generate a unique subclock for that register if we want to reach maximum power saving. Here in our implementation, we do not clock-gate such registers due to timing concern as the unique subclock may need more combinational logic to generate and thus tighten the timing constraint.

We transform the finite state machine into one-hot-key format encoding [6], where we have a register for each state, and only when we reach that state will the corresponding register output 1. With this, the delay of the generated subclock signal can be reduced to a minimum to be the delay of only one AND gate. With the information of which registers are used at what states, we will generate subclock signals based on different states and thus reduce the dynamic power of unnecessary toggle.

There are multiple choices for state machine encoding, such as traditional binary state encoding and one-hot-key encoding. Binary encoding uses the numbers from 0 to $N-1$ to represent all the states, where N is the total number of states. In this way, we only need $log_2(N)$ flip-flops. This could use a minimum of registers to store the state and thus reduce the area; however, while using the states to drive the datapath, more combinational logic is needed to decode the states into the required signals and thus it is actually not power optimal or area optimal. One-hot-key encoding, on the other hand, provides more straightforward instruction signals to the datapath despite using more flip-flops to store the states. This method creates one register for each state, and only one of them will be 1 at a time. During each state transition, only one flip-flop will be on. Sutter et al. [7] evaluated the power saving for different state encodings for circuits with different numbers of states, and it turns out that one-hot-key encoding is better for large circuits with over 16 states. It is very common for HLS tools to generate hundreds of states.

One-hot-key encoding is also good for our implementation. One possible concern of our method comes from the skew between the main clock signal and subclock signal. This may lead to an asynchronous data problem. To reduce this effect, the combinational logic used to generate the subclock signal needs to be as simple as possible. With binary state encoding, this logic will be much more complicated than one-hot-key encoding, where the combinational logic will simply be an AND gate like in Fig. 3.1. The subclock is generated by the AND gate, the clock signal and the signal from the latch.

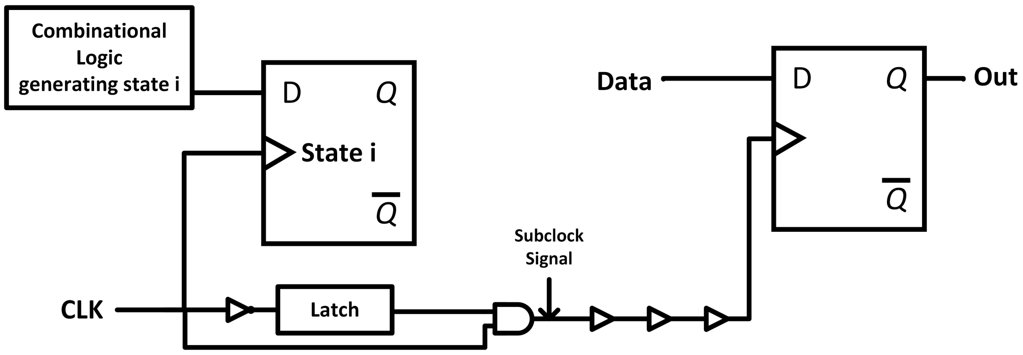


Figure 3.1: Clock gating based on one-hot-key state machine.

CHAPTER 4

IMPLEMENTATION

Our design will be implemented on top of the VAST HLS engine. VAST is one of the state-of-the-art HLS engines and is used widely for HLS research [8, 9, 10]. VAST uses the Shang high-level synthesis framework, which is implemented as a LLVM backend. The VAST engine takes C specification as input and generates Verilog RTL hardware description from LLVM-IR. It works on the LLVM machine code layer and makes full use of the information to perform optimizations on high-level synthesis specific operations.

The scheduling stage will take the operations generated by the LLVM compiler and schedule them while inserting sufficient delay between instructions to respect the data dependency and the duration of the operations. The schedule will be subject to some hardware resource and timing restrictions. Meanwhile, it will attempt to minimize some criteria. These criteria can be area, resource, delay or a combination. The binding stage will bind variables to registers and operations to functional hardware or combinational logic based on the scheduled result. After the binding is completed, a piece of hardware description code will be generated describing the data path and state machines resulting from the binding. A hardware compiler will take the generated RTL from the HLS engine and create target-specific gate level implementation. One-hot-key encoding for the states is naturally supported in VAST, so we can directly use that signal for our design.

We implement our clock-gating feature at the very end stage of the VAST engine right before generating the RTL code. In VAST engine, we have the following data generated for analyzing:

1. VASTOp: This is the class where we translate each instruction in high level

into internal operations; it contains the operands that are used and the output it generates. Each operand will have its corresponding register.

2. VASTRegister: This is the class where we instantiate the registers; some of the registers may be shared by different operands from different VASTOps.
3. VASTSlot: This is the class for each state in the finite state machine. In each VASTSlot, there are multiple VASTOps to be executed.

The general steps of the clock-gating flow are as follows:

1. Analyze each state and annotate the states where each register is written.
2. Sweep the registers and annotate the states where we need to generate the corresponding subclock signal.
3. Generate the subclock signal and modify the register block during the RTL code generation flow.

A state set is created for each register. This will be used to record how many states are using this register. And we then maintain a state set for the whole module to indicate how many subclocks are generated. Algorithm 1 shows each step of the work.

Algorithm 1 Inserting Clock-Gating Edges

```
1: for Each $VASTSlot\ S_i$ do
2:     for Each $VASTOperation\ OP_i \in S_i$ do
3:         for Each $Operand \in OP_i$ do
4:             Get corresponding VASTRegister $R_i$ for $Operand$
5:             $R_i$'s $SlotSet.insert(S_i)$
6:         end for
7:     end for
8: end for
9: $sub_clock_state_set.clear()$
10: for Each $VASTRegister\ R_i$ do
11:     if $R_i.set$ has only one slot in it then
12:         $sub_clock_state_set.insert(R_i.slot)$
13:     end if
14: end for
15: for Each $VASTSlot \in sub_clock_state_set$ do
16:     Generate subclock wire for $VASTSlot$
17: end for
18: for Each VASTRegister $R_i$ do
19:     if $R_i.set$ has only one slot in it then
20:         Generate
21:     end if
22: end for
```

The time complexity of this algorithm is $O(N + M + R)$, where N is the total number of the operands used in the module, M is the number of slots and R is the number of registers. Since $N > M$ and $N > R$ (there are multiple operations in each slot and the operands may share registers), the time complexity can be further simplified as $O(N)$. With linear time complexity, the algorithm will finish in a short time.

CHAPTER 5

EXPERIMENTAL RESULTS

Our experiment is combined with the VAST HLS engine and downstream power measurement flow. A set of CHSTONE testbenches [11] is used to test our flow and compared with the VAST flow that does not enable this low-power feature. To fully test the technique in different application domains, we used benchmarks from arithmetic application (dfmul, dfsin, etc.), encryption domain (blowfish) and media application (mpeg2). All of these benchmarks involve computationally heavy tasks and are well suited for HLS design.

We first verified the behavior of the new circuit to ensure that the functionality of the circuit is not changed. VAST engine auto-generates a Verilog testbench for output RTL code. Newly generated RTL code with state clock gating is verified using this testbench. The new designs are verified with all the CHStone benchmarks and could generate correct hardware outputs.

After getting the RTL Verilog code, we use state-of-the-art commercial tools to measure the power consumption for both versions. Design Compiler is used to perform logic synthesis and generate the netlist with the IBM 45nm technology library. Provided with RTL level simulation waveform, we use Primetime to obtain the power consumption for the circuit. RTL level clock gating is enabled in Design Compiler in both cases to maximize power saving as we are trying to simulate the condition in reality. And given that there are two stages where we can insert the clock-gating feature (HLS flow and logic synthesis), there are possibilities that clock-gating opportunities discovered at higher levels in the design flow might also be discovered by downstream tools during the RTL clock-gating process. By enabling RTL level clock gating, we can eliminate the power saving found in both stages and thus analyze purely the benefits from HLS clock gating. Area reports and timing reports are also generated during the process of logic synthesis.

From the result in Table 5.1 we can see that we reduced average dynamic power 7% to 20% on different benchmarks. With extra subclock signals, the net switching power is increased. However, since we avoided unnecessary register changes in idle states, not only did we save the flip-flop toggle power, but we also reduced the dynamic power for the combinational logic using those registers as input. And that is reflected in the reduced cell internal power. At the current stage, we only clock gate the registers that are used in just one state due to timing concern. With more fine-grained state clock gating there is a possibility to reduce the power even further.

The extra area increase comes only from the AND gate inserted for clock gating, so the area increase is not huge. We can see from the results in Table 5.2 that sometimes there may even be area reduction possibly due to the optimization in logic synthesis. In general, the less than 3% area increase is negligible.

Table 5.1: Power Report

Testbench		dfadd	dfmul	dfsin	dfdiv
Before clock gating (mw)	net switching	3.08E-05	2.31E-05	9.23E-05	5.19E-06
Before clock gating (mw)	cell internal	2.67E-04	1.90E-04	9.07E-04	2.62E-04
Before clock gating (mw)	total	2.98E-04	2.13E-04	9.99E-04	2.67E-04
After clock gating (mw)	net switching	3.35E-05	2.34E-05	9.22E-05	9.73E-06
After clock gating (mw)	cell internal	2.08E-04	1.63E-04	7.99E-04	2.35E-04
After clock gating (mw)	total	2.41E-04	1.86E-04	8.91E-04	2.44E-04
After vs. before ratio(total power)		80.97%	87.41%	89.15%	91.52%
Testbench		mpeg2	blowfish	adpcm	
Before clock gating (mw)	net switching	3.03E-06	9.61E-04	1.03E-04	
Before clock gating (mw)	cell internal	1.91E-04	3.16E-03	2.61E-03	
Before clock gating (mw)	total	1.94E-04	4.12E-03	2.71E-03	
After clock gating (mw)	net switching	7.41E-06	8.45E-04	1.26E-04	
After clock gating (mw)	cell internal	1.72E-04	3.01E-03	2.47E-03	
After clock gating (mw)	total	1.80E-04	3.86E-03	2.59E-03	
After vs. before ratio(total power)		92.86%	93.57%	95.57%	

Table 5.2: Area Report

Testbench	dfadd	dfmul	dfsin	dfdiv
Area Before clock gating (µm^2)	17221	5466	209927	18247
Area After clock gating (µm^2)	16139	5606	205590	18848
Testbench	mpeg2	blowfish	adpcm	
Area Before clock gating (µm^2)	11085	447240	342721	
Area After clock gating (µm^2)	10681	421155	341841	

CHAPTER 6

CONCLUSION

We implemented clock gating based on state machine for high level synthesis flow using VAST HLS engine. One-hot-key state encoding was used to reach maximum power reduction and reduce clock mismatch. With the clock-gating technique, we can stop the toggle of the registers that are not used in current state. About 5% to 20% power saving is obtained with little area penalty in CHStone testbenches for different application categories. We believe this method makes full use of the information given during high-level synthesis, explores the micro-architectural level relations of the circuits and greatly widens the possibilities for clock gating.

REFERENCES

[1] G. Martin, B. Bailey, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology.* Morgan Kaufmann, 2010.

[2] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research whitepaper*, 2015.

[3] Q. Wu, M. Pedram, and X. Wu, "Clock-gating and its application to low power design of sequential circuits," *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, vol. 47, no. 3, pp. 415–420, 2000.

[4] L. Benini, P. Siegel, and G. De Micheli, "Saving power by synthesizing gated clocks for sequential circuits," *IEEE Design & Test of Computers*, no. 4, pp. 32–41, 1994.

[5] H. Li, S. Bhunia, Y. Chen, T. Vijaykumar, and K. Roy, "Deterministic clock gating for microprocessor power reduction," in *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on.* IEEE, 2003, pp. 113–122.

[6] M. Riahi Alam, M. Ersali Salehi Nasab, and S. M. Fakhraie, "Power efficient high-level synthesis by centralized and fine-grained clock gating," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 34, no. 12, pp. 1954–1963, 2015.

[7] G. Sutter, E. Todorovich, S. López-Buedo, and E. Boemo, "Low-power FSMs in FPGA: Encoding alternatives," in *Integrated Circuit Design. Power and Timing Modeling, Optimization and Simulation.* Springer, 2002, pp. 363–370.

[8] L. Yang, S. Gurumani, D. Chen, and K. Rupnow, "Behavioral-level IP integration in high-level synthesis," in *Field Programmable Technology (FPT), 2015 International Conference on.* IEEE, 2015, pp. 172–175.

[9] H. Zheng, S. T. Gurumani, K. Rupnow, and D. Chen, "Fast and effective placement and routing directed high-level synthesis for FPGAs," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2014, pp. 1–10.

[10] L. Yang, M. Ikram, S. Gurumani, S. A. Fahmy, D. Chen, and K. Rupnow, "JIT trace-based verification for high-level synthesis," in *Proceedings of the International Conference on Field Programmable Technology*, 2015, pp. 228–231.

[11] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "CHStone: A benchmark program suite for practical C-based high-level synthesis," in *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1192–1195.