Latch-based RISC-V core with popcount instruction for CNN acceleration

Master's thesis
University of Turku
Department of Computing
Embedded electronics
2021
Ohto Myllynen

UNIVERSITY OF TURKU
Department of Computing

OHTO MYLLYNEN: Latch-based RISC-V core with popcount instruction for CNN acceleration

Master's thesis, 54 p., 15 app. p.
Embedded electronics
May 2021

Energy-efficiency is essential for vast majority of mobile and embedded battery-powered systems. Internet-of-Things paradigm combines requirements for high computational capabilities, extreme energy-efficiency and low-cost. Increasing manufacturing process variations pose formidable challenges for deep-submicron integrated circuit designs. The effects of variation are further exacerbated by lowered voltages in energy-efficient designs. Compared to traditional flip-flop-based design, latch-based design offers area, energy-efficiency and variation tolerance benefits at the cost of increased timing behavior complexity. A method for converting flip-flop-based processor core to latch-based core at register-transfer-level is presented in this work.

Convolutional neural networks have enabled image recognition in the field of computer vision at unprecedented accuracy. Performance and memory requirements of canonical convolutional neural networks have been out of reach for low-cost IoT devices. In collaboration with Tampere University, a custom popcount instruction was added to the cores for accelerating IoT optimized vehicle classification convolutional neural network.

This work compares simulation results from synthesized flip-flop-based and latch-based versions of a SCR1 RISC-V processor core and the effects of custom instruction for CNN acceleration. The latch core achieved roughly 50% smaller energy per operation than the flip-flop core and 2.1x speedup was observed in the execution of the CNN when using the custom instruction.

Keywords: RISC-V, Latch, Energy-efficient, IoT

Contents

Chapter 1

Introduction

Energy efficiency is vital for vast majority of battery powered systems. The higher the energy efficiency, the more computations a system can perform between recharges or battery swaps. In the field of mobile and embedded devices, long battery life is often as important or more important feature than computation speed. In addition to prolonging the operating time of such devices, increasing energy efficiency enables countless new applications. More and more complex functionality can be realized with less and less energy. Energy efficiency is one of the key enablers for the growing Internet-of-Things (IoT) market and the technologies of the future. Increasingly wireless, mobile and interconnected systems require massive amounts of data to be processed in real-time, while being operational for extended periods of time without a constant power supply. Advances in neural networks have made many data analysis tasks like image recognition possible with much higher speed and accuracy than ever before. However, in many cases the memory and computation requirements cannot be met by current low-cost IoT devices. To truly reap the benefits of neural networks and to bring them to IoT realm, efficient hardware and optimized algorithms are both required. The amount of transistors in integrated circuits has been growing exponentially in the past decades causing power density and power consumption to increase towards practical limits. Heat dissipation has become a major issue for data centers and high performance computing in general. Offloading some of

the computation to the network edges for improved latencies and reduced costs calls for highly energy-efficient hardware. Besides, the available energy to harness is finite. As Jan Rabaey illustrates with his playful extrapolation in [1], the entire energy of the Milky Way galaxy would be spent in just 180 years, if computational requirements are assumed to double every year. This work explores potential benefits of latch-based design in terms of energy-effiency and variation tolerance. Additionally, acceleration of an IoT optimized Convolutional Neural Network (CNN) by adding custom popcount instruction to the core was simulated and confirmed.

Most modern digital devices are built with Complementary Metal-Oxide Semiconductor (CMOS) logic. For digital CMOS circuits energy can be estimated with equation 1.1 [1][1]. The dynamic energy dissipation is dependent only on the total capacitance C and voltage V. Leakage energy is defined by the voltage V and leakage current I_{leak} over operation time t. It is important to notice that energy is not dependent on operating frequency. Leakage current of a gate can be estimated with equation 1.2 [1] , where kT/q is thermal voltage and equals roughly 25mV at room temperature [1]. W is the width of the gate and V_{th} is the threshold voltage. I_0, W_0 S, and λ_d are technology specific constants corresponding to minimum current, minimum gate width, sub-threshold swing and Drain Induced Barrier Lowering (DIBL) factor.

Leakage currents of the gates are strongly dependent on drain-source voltage V_{DS} and threshold voltage V_{TH}. While lowering the operating voltage impacts mostly energy dissipation at first, it also pushes the threshold voltages down. At a certain point leakage energy starts to dominate when the transistor's ability to turn off diminishes. Additionally, as the voltage lowers propagation delay of the signals starts to increase until the circuit fails. Thus lowering the voltage limits the maximum operating speed of the device as well.

[1]Equation 1.1 can be obtained by combining several equations from [1]

$$E_{dyn} + E_{leak} = CV^2 + \frac{V I_{leak}}{t} \tag{1.1}$$

$$I_{leak} = I_0 \frac{W}{W_0} 10^{\frac{-V_{TH}+\lambda_d V_{DS}}{S}} \quad \text{for } V > \frac{3kT}{q} \tag{1.2}$$

The capacitance C in equation 1.1 correlates to the amount of transistors switching at a given time. It is also dependent on the CMOS technology and implemented architecture. The capacitance component of energy dissipation can therefore be reduced by migrating to a newer CMOS technology process or by architectural optimizations.

The switching rate of transistors per clock cycle is often estimated by activity factor α, a percentage of transistors in the device. For processors, α varies depending on the executed operations and the architectural design of the system. While software optimizations can reduce α in many cases, ultimately they are bound by the hardware capabilities. Software optimizations relying on nonstandard hardware functionality can further improve the performance at the cost of software portability. Implementing hardware specific optimizations might also require significant efforts on the software side, depending on the software stack. Architectural hardware optimizations are more generic and reliable, since they potentially require little to no additional effort from the software developers. Improving the energy efficiency and speed of most commonly executed instructions and the execution pipeline in general benefits all of the use cases. At some point the benefits gained start to diminish and optimizing the common case is no longer feasible. Another path to improvements, albeit usually more application specific, is adding new instructions. For example, a new instruction can be added to the Central Processing Unit (CPU) for a common instruction pattern. Instead of executing multiple instructions to achieve certain behavior, the task can now be done entirely in hardware by a single instruction. In order to utilize the new instruction, support for its encoding must be added to the toolchain and the software has to be recompiled, possibly after some modifications. For most real world instruction sets modifications are either difficult and prohibitively costly or prohibited all

together. Fortunately, free and open RISC-V Instruction Set Architecture (ISA) standard is designed to support instruction set extensions, providing the means to implement custom instructions.

Previously shrinking process nodes have offered gains in performance, energy savings and area reduction for relatively low effort compared to architectural optimization. However, as the feature sizes have started to approach the thickness of just a few atoms, the effects of variation in manufacturing process have magnified. Migrating to new deep-submicron CMOS nodes has become increasingly difficult and costly due to reduced yields, caused by increased process variation, among other challenges. High local variability adds an element of uncertainty to the timing of the chips, causing higher rates of failure. As a consequence, alternative ways to increase energy-efficiency are becoming more attractive.

The theoretical minimum energy required for a signal to be distinguishable from noise is defined by *Shannon-von Neuman-Landauer limit* to be $0.29 * 10^{-20} J$ at room temperature [1]. Reaching this limit in practice is not very likely, but it sets the ultimate goal for improvements. Modern devices operate few magnitudes above this limit, based on simulation results from [2]. In practice some voltage margin is added to ensure the functionality of the fabricated devices. In order to confirm that as many of the manufactured devices as possible are functional despite variations in the manufacturing process, the operation of the device is tested on different manufacturing process corners. The process corners represent global inter-die variations affecting the whole chip. Before manufacturing the corners are simulated and the design is tuned to tolerate the variations, often translating to adding margin to the typical case. Common corner choices for testing CMOS process transistor variation are SS (Slow-Slow), TT (Typical-Typical) and FF (Fast-Fast), though other combinations are also possible. The letters signify carrier mobility for NMOS and PMOS transistors respectively. In addition to global variations between chips, local intra-die variations are present within each chip. While some local variation sources are deter-

ministic, random local variations are also present. Statistical timing models are needed to predict the effects of random local variations more accurately.

Energy per Operation (EOP) is a commonly used metric to measure processor efficiency. Because reducing the voltage reduces not only the required energy, but it also reduces the maximum operating speed due to increased delays, reducing EOP is a balancing act between energy and performance. There exists a point for optimal EOP also known as Minimum Energy Point (MEP). Hiienkari *et al.* have reached the EOP of 2.87 pJ/cycle with ARM Cortex-M3 CPU by using state-of-the-art energy saving techniques to drastically cut down voltage margins [3]. Figure 1.1 shows EOP measurements from the core manufactured at different process corners over operating voltage range. As shown in figure 1.1 when the operating voltage is reduced the EOP improves at first, but when the voltage is reduced further EOP starts to increase at a certain point. MEP is located at the lowest point of the curve. The core was measured over different process corners and the effects of process variations are evident. In the FF process corner the energy is much higher than in the TT or SS corners.

On the downside as the voltage decreases the the effects of process variation are magnified. Majority of CMOS designs use flip-flop cells for sequencing, due to their relatively simple and well understood timing behavior. As an alternative to flip-flops the sequencing can also be realized with latch cells at the cost of increased timing behavior complexity. While latches and flip-flops are on equal footing in terms of performance under traditional timing models, latches are able to give better yields under statistical timing models. Statistical timing models are used to predict the circuit timing more accurately when process variations become significant. Hurst *et. al* have shown in [4] that transparent latches can reduce the rate of failures by four times compared to flip-flops. Latches are able to meet more timing corners, because in latch-based sequencing the signal propagation window is much wider than in flip-flop-based sequencing. Additional benefit of latch-based designs is that latches are less complex devices than flip-flops and thus they can be constructed

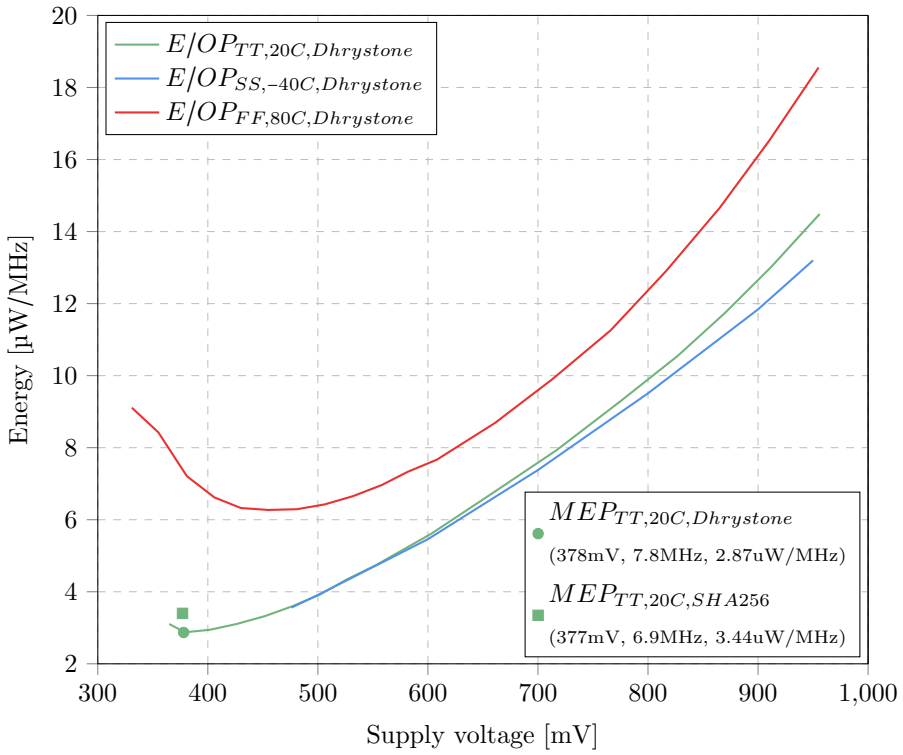


Figure 1.1: ARM Cortex-M3 EOP[3] © 2020 IEEE

with fewer transistors, resulting in area and energy savings.

Focus of this master's thesis is in latch-based energy-efficient design, but other complementing low-power techniques could be added on top of this approach to further increase the energy efficiency of the design. An overview of RISC-V and rationale for choosing an open-source RISC-V core as a starting point for latch-based core is given in chapter 2. Chapter 3 introduces the custom popcount instruction used to accelerate convolutional neural networks. The characteristics of latches and flip-flops from sequencing perspective are detailed in chapter 4. Chapter 5 explores a methodology for latch-based sequencing, conversion of flip-flop-based designs to latch-based designs and related challenges. Simulation results are shown and examined in chapter 6. Finally, the conclusions are presented in chapter 7.

Chapter 2

RISC-V

RISC-V is fifth generation of an open Instruction Set Architecture (ISA) standard developed originally at UC Berkeley for academic use. While RISC-V began as an academic endeavor to provide a realistic ISA for research and classroom use, it has gained a lot of traction especially in IoT and low power applications. RISC-V ISA is designed with flexibility and extensibility in mind, making it an excellent candidate for creating cores optimized for specific use cases. As a new ISA RISC-V has had the opportunity incorporate the knowledge gained from older ISA specifications and decades of field testing. The base instruction set has been carefully designed to contain only the most universal instructions. Application domain specific instructions can be added in a modular fashion on top of the base instruction set depending on the implementation requirements. This enables software compatibility across different implementations, given that the available software stack is able to emulate optional instructions when they are not supported by the implementation. RISC-V ISA standard is maintained by the RISC-V foundation, which accepts companies as well as individuals as members. The RISC-V specification is freely available for anyone to download and implement without any costs. The specification version used as reference in this work is The RISC-V Instruction Set Manual Volume 1: Unprivileged ISA [5].

Currently, most widely-adopted ISAs are proprietary. To create an implementation

for proprietary ISA lengthy negotiations and expensive licenses are required. Custom implementation of proprietary ISAs has therefore been possible only for a few companies. Most companies have had to settle with integrating premade designs and components to their products. The free and open licensing model of RISC-V permits anyone to create their own RISC-V compatible processor implementations, while allowing proprietary implementations as well. In the light of expensive licensing and possible royalty costs associated with proprietary ISAs the RISC-V is very tempting from a financial point of view.

An ISA is an abstract interface between hardware and lowest level of software. It contains all the details needed to create machine language programs that can be run on the hardware. For a program to be executable in certain hardware it must be written in or translated to a format defined by the ISA. Usually ISA is accompanied with a compiler toolchain capable of transforming high level code, such as C, to a format specified by the ISA. The RISC-V ISA specification avoids implementation style or technology specific details for increased flexibility. RISC-V implementations can range from single core processor to multi-core System-on-a-Chip (SoC) and server clusters with thousands of cores. In [6] Asanović and Patterson, both very closely involved in the creation of RISC-V, argue that ISA, one of the most important interfaces, should not be proprietary. They envision that having a widely used open ISA could lead to increased innovation, shorter time to market, lower costs, fewer bugs and transparency. Ambitiously, the authors set the goal of RISC-V to eventually become the standard ISA for all computing devices.

RISC stands for Reduced Instruction Set Computer as opposed to Complex Instruction Set Computer (CISC). RISC architecture is based on the idea of defining a small, yet comprehensive, set of simple instructions and executing them really fast. CISC architecture on the other hand defines a large set of instructions including complex instructions that can take longer to execute. While CISC implementations have dominated in the high performance domain and RISC implementations have prevailed in the embedded and mobile

domain, there is no fundamental reason why RISC architecture could not be used for high performance applications or CISC architecture for energy-efficient designs. In [7] Blem *et. al* compared effects of ISA on performance and energy-efficiency for ARM (RISC) and x86 (CISC) implementations. They found that modern microarchitecture techniques and compiler optimizations help to mitigate CISC implementation overheads. For example, complex instructions are split to RISC-like micro-ops and compilers tend to favor RISC-like instructions over the complex ones. They concluded that the differences between ARM and x86 implementations are largely dependent on the design point and not so much on the ISA. But they also note that for simple very low performance processors CISC ISA such as x86 or even ARM's full RISC ISA would be too complex and add unnecessary overhead.

The IoT domain covers a wide range of applications requiring high energy effiency and low cost. Typically IoT end-nodes stay in sleep mode most of the time to save energy and wake up to quickly execute pending tasks, triggered by external events, before going back to sleep. The computational requirements can range from reading and storing a sensor value to complex signal processing. To cover the wide range of use cases different kinds of programmable cores are needed for various desing constraint combinations. The shared or partially shared RISC-V ISA between various cores brings benefits in terms of bus interconnect, interface and software compatibility.[8]

The RISC-V specification defines four related base integer ISAs and optional extensions. Base ISA is mandatory and can be either RV32I, RV32E, RV64I or RV128I. The base ISA defines the width of integer registers, the number of registers and size of the address space. RV128I is less polished and the primary base ISA variants are RV32I and RV64I for 32-bit and 64-bit architectures respectively. The RV32E base ISA is a variant of RV32I with only 16 registers for small implementations. RV32I and RV64I base ISAs have 32 registers. The base ISAs are considered independent variants to allow better optimization, therefore they might have slight differences in encoding and behavior of

the extensions. The memory consistency model of the RISC-V ISA is quite flexible and defaults to RISC-V Weak Memory Ordering (RVWMO), which is the weakest allowed memory model. Under RVWMO memory instructions executed within a hardware thread (hart) appear in order from the perspective of other memory instructions executed in the same hart, but may appear in different order from the perspective of other harts' memory instructions. Further constraints can be added for stricter memory model as needed.

Because the RISC-V specification is still under development a part of the specified extensions might undergo some changes. Commonly used extensions M, A, F, D, Q, C, Zicsr and Zifencei are quite stable and only minor changes, if any, are expected on their specification. M extensions adds multiplication and division instructions for integers. Extension A adds atomic operations for inter-processor synchronization. F, D and Q extensions respectively add single-, double- and quad-precision floating point instructions and registers. C extension add compressed 16-bit forms of common instructions for reduced instruction memory footprint. Zicsr adds CSR access instructions. Zifencei adds instruction to synchronize writes and fetches to instruction memory. Additionally G shorthand is used to denote IMADZifencei extension combination and floating point extension implicitly include the Zicsr extension.

2.1 Cores

Since RISC-V ISA only defines the interface between software and hardware, but not the actual hardware, how a RISC-V compliant processor core is realized is up to the designer. Digital hardware is described in Register-Transfer level (RTL) with a Hardware Description Language (HDL). RTL code describes the signals, registers and behavior of logic in the circuit at high abstraction level. The circuit behavior is first tested and verified in RTL simulations. Considerable RTL verification effort is required to ensure that the features of the manufactured devices will behave as expected. Correctly functioning RTL description

is the starting point for creating the actual circuit. Since RISC-V ISA itself is not encumbered by restrictions that come with proprietary ISAs, the RTL can be shared much like open source software. By itself the RTL code can be used for Field-Programmable Gate Array (FPGA) implementations, but it is not enough to produce an Application-Specific Integrated Circuit (ASIC), since it does contain any information about the physical implementation.

Numerous proprietary and open source RISC-V implementations are readily available. While some of the available cores utilize latches to some extent, it seems that currently there are no fully latch-based cores available. Therefore to reduce the scope of this work a small, energy-efficient and mature flip-flop-based RISC-V core was chosen as a starting point to be transformed to a latch-based implementation. Naturally, only open source RISC-V cores were considered for this work. Due to the nature of open source the quality of the implementations can range from hobby projects to academic research work and production ready cores equivalent to proprietary solutions on the market. 20 RISC-V cores were compared, in order to find a good candidate to modify. A comparison of the cores can be seen in table 2.1. The RISC-V Foundation provides a compliance test suite to ensure compatibility of the implementation with the RISC-V specification. Cores that were not compliant and the cores for which the compliance was unclear (N/A) as well as cores with too restrictive licensing were deemed unsuitable. Cores built with 64-bit base ISA, like Ariane and BOOM, are targeting towards higher performance and even running multi-user operating systems like Linux. As a consequence they are also considerably more complex. To limit the development effort and since 32-bit ISA is enough for most embedded applications Ariane and BOOM were left out. Support for latch-based digital logic varies among HDLs. Since implicit latches as a result of incomplete conditional expressions are a common source of bugs in RTL code, many HDLs and Electronic Design Automation (EDA) tools have better support for avoiding latches than actually using them. While creating latches is certainly possible in most HDLs, in general the support for

latch-based sequencing is quite limited. SystemVerilog was chosen as the target language since it has at least some support for explicitly modeling latched logic and the language itself is quite well supported by EDA tools. After this consideration cores that were not written in SystemVerilog were discarded.

Out of the 20 RISC-V implementations two of them fit the criteria the best: SCR1 and Zero-riscy. Both cores are well tested, very small low power cores written in SystemVerilog. SCR1, maintained by Syntacore, a semiconductor IP company specializing on RISC-V ISA, is mature and highly configurable core. SCR1 repository includes a good quality development environment with easy-to-extend testbench. Zero-riscy[1] was created as a part of Paraller Ultra Low Power (PULP) platform, a joint project between Integrated Systems Laboratory, ETH Zurich and Energy Efficient Embedded Systems group of University of Bologna. Unlike SCR1 Zero-riscy does not come with directly a coupled development environment. Instead it is intended to be used with PULPino single core microcontroller system. Like SCR1 the development environment for Zero-riscy is mature and comes with a testsuite. Development environments of both of the cores have support for FPGA targets as well. Ultimately the choice came down to personal preference. Both cores are well suited for small scale experimentation. SCR1 was chosen as the target core for latch transformation, because it's slightly more configurable and the development environment seemed more straightforward.

2.2 SCR1

SCR1 is well documented: the external architecture specification, user manual, and FPGA specific Software Development Kit (SDK) instructions are included in the core repository [9] and in related SDK repository [10]. The core can be configured with either RV32I or RV32E base ISA and optionally with C extension for compressed 16-bit instructions

[1]Zero-riscy has been renamed to Ibex and contributed to lowRISC non-profit since this evaluation

Core	License	ISA	HDL	Compliance	Notes
Ariane	Solderpad Hardware License v. 0.51	RV64IMAC	SystemVerilog	Yes	PULP platform
BOOM	BSD	RV64GC	Scala (Chisel)	Yes	Out-of-order
Hummingbird E200	Apache 2.0	RV32IMAC	Verilog	Yes	Targeted for Chinese
Minerva	BSD/LambdaConcept	RV32I	Python	No	Uses nMigen toolbox
MR1	Unlicense	RV32I	SpinalHDL	Yes	Hobby project
ORCA	Vectorblox Orca	RV32IM	VHDL	N/A	-
OPenV/mriscv	MIT	RV32I	Verilog	No	-
PicoRV32	ISC	RV32IEMC	Verilog	Yes	-
ReonV	GPL v3.0	RV32I	VHDL	N/A	RISC-V port of Leon3 core (from GRLIB IP library)
Reve-R	Apache 2.0/BSD	RV32IMAC	CDL	No	-
Roa Logic RV12	Non-commercial	RV32|64I	SystemVerilog	Yes	-
Rocket	BSD	RV32|64G	Scala (Chisel)	Yes	Rocket chip generator
Riscy Processors	MIT	RV64IMAFD	Bluespec SystemVerilog	Yes	-
RI5CY	Solderpad Hardware License v. 0.51	RV32IMFC	SystemVerilog	Yes	PULP platform
SCR1	Syntacore/Solderpad Hardware License v. 0.51	RV32IEMC	SystemVerilog	Yes	-
SERV	ISC	RV32I	Verilog	Yes	Bit-serial core
Shakti	BSD, 3-clause	RV32|64IMAC	Bluespec SystemVerilog	Yes	Core family
SweRV EH1	Apache 2.0	RV32IMC	SystemVerilog	N/A	-
VexRiscv	MIT	RV32IMC	SpinalHDL	Yes	-
Zero-riscy	Solderpad Hardware License v. 0.51	RV32IMCE	SystemVerilog	Yes	PULP platform

Table 2.1: RISC-V core (platform) comparison, collected in Spring of 2019

and M extension for integer multiplication and division instructions. SCR1 implements only machine mode privilege level for running trusted code. The core is quite small and according to documentation synthesizes to only around 11-33 kGates depending on the configuration.

2.2.1 Memory

SCR1 implements harvard architecture i.e. the instruction and data memories have dedicated memories and access buses, but the core also has up to 64 kBytes of low latency

dual-port Tightly Coupled Memory (TCM), which is shared by instructions and data. TCM is designed for storing commonly used instructions and data for increased throughput. Memory access interfaces can be configured either as Advanced eXtensible Interface 4 (AXI4) or AMBA High-performance Bus-lite (AHB-lite). The memory is byte addressed and little endian. Instruction and data memories both have 32-bit continuous address space. SCR1 uses strong memory access model, guaranteeing one-to-one match of the sequence and number of memory accesses performed with the executed instructions. The memory is only accessible by load and store instructions.

2.2.2 Pipeline

SCR1 implements in-order 2-4 stage pipeline depending on the configuration. The pipeline is divided to following phases: request to instruction memory, instruction fetch, instruction decode, execution, and commit point. The execution consists of operand fetch, arithmetical and logical operations, load and store operations, and instruction flow control. In 2-stage pipeline configuration the first stage contains request to instruction memory, instruction fetch and instruction decode phases and the second stage contains execution phase and commit point. For 3- and 4-stage configuration a queue can be added before instruction decoding or execution phase or before both phases. Since operand fetch and commit point are always in the same stage SCR1 pipeline has no data hazards. Structural hazards are resolved by stalling execution until the occupied resource becomes available and control hazards are dealt with by simply flushing and restarting the pipeline.

2.2.3 Other optional features

Optionally the core also supports low latency Integrated Programmable Interrupt Controller (IPIC) with up to 32 interrupt signals, vectored interrupts, global clock gating and single cycle integer multiplier. In addition the core comes with optional debugging subsystem with Joint Test Action Group (JTAG) interface.

2.2.4 Configuration

To reduce development efforts the core was configured for RV32IC target with 2-stage pipeline and without any optional blocks. AHB-lite was used for the development and testing, since it's less complex and more energy-efficient than AXI4 [11]. Though, the core top level environment, including the memories and memory bus interface, was not included in the synthesis and only simulated as a part of the testbench in netlist simulations.

2.2.5 Testbench

The testbench of the core has ready made targets for multiple simulators and it integrates with RISC-V compliance tests. After setting up the environment, RISC-V compliance tests and benchmarks were run with Verilator and VCS simulators in order to confirm the correct functionality. Additionally after initial RTL simulation tests DE10-lite FPGA and DE10-lite SDK provied by Syntacore was used to verify core functionality. Since FPGAs in general are not feasible for latch-based designs, no further FPGA experimentation was done beyond verifying that the original RTL is indeed functional. The testbench was augmented with SHA-256 test in addition to the included Dhrystone 2.1 and Coremark benchmark in order to increase the variety of simulated activity. Based on previous experience looping SHA-256 test tends to generate slightly more computationally intensive activity than Dhrystone 2.1. or Coremark. The SHA-256 test is lightly modified version of an implementation by Brad Conte [12] and it runs 1000 rounds of SHA-256 starting from a known plaintext and compares the result to an expected value. In order to avoid timing errors while running the testbench with synthesized design, I/O delays at the boundary have to be compatible with the timing of the synthesized design. Therefore a wrapper module adding delay to the I/O signals was added around the core instantiation in order to simulate I/O delays more realistically.

Chapter 3

CNN ISA extension

A Convolutional Neural Network (CNN) is a type of artificial neural network that is most commonly utilized for image recognition tasks. Generally CNNs consist of alternating convolutional and pooling layers followed by fully connected layers. Convolutional layers are used to detect features from the image, the pooling layers are used to reduce the dimension of the data and finally the classification is done by fully connected layers. The final layer outputs a value for each type of class the network can detect. The values can be negative or positive and the higher the value the higher confidence the classification has. The class with highest value is the classification the network sees as most likely, but it is also possible that scores are quite close to each other, if the CNN cannot classify the image with high confidence or very low if the image does not fit in any of learned the categories.

Each convolutional layer has at least three dimensions: width, height and depth, where the depth corresponds to the amount of filters in the layer. Filters correspond to the features the layer is attempting to detect. Each filter has associated weights learned during the training of the network. For example, a layer with a depth of three could be used for detecting a feature from each of the color components RGB in an image, but in practice the layers are usually deeper. Each filter is slid over the inputs of the previous layer along width and height dimensions. At each step all of the filter weights are applied to the re-

gion producing a single output value. The weights are multiplied with the related inputs in the current region and added together. Typically the output value is passed to the next layer through an activation function, for example $max(0, x)$. If the input pixels of an image are flattened to one dimensional vector the application of weights can be seen as a dot product. In the fully connected layers all of the inputs are connected to each of the outputs, hence the name.

For example, applying three 3x3 filters to equal size input, depending on the implementation, could require nine multiplications, eight additions and a division for each filter and additional three multiplications and two additions for the dot product, if the output is 1x1x1. Additionally 27 weights have to be stored for the filters. It is easy to see that even for relatively small images the computation complexity and memory requirements increase quickly with image size and filter count, given that CNNs used for real life applications contain many filters and multiple fully connected layers.

Advances in CNNs have enabled image recognition at unprecedented accuracy and opened up many new possibilities in the field of computer vision. But the memory and computation requirements have been out of scope for many IoT applications. Payvar *et al.* have developed an IoT suitable CNN for vehicle image recognition. In [13] they used a method of efficiently packing the weights of CNN with minimal loss of accuracy and a method for accelerating CNN operations. In their work they use a binarization method to condense 32-bit floating point weights of the CNN to just 1-bit and pack the weights to 32-bit vectors, saving around 95% in the data memory size, while losing under 5% in accuracy. Additionally they used a popcount instruction in place of multiplication for both convolutional and fully-connected layers to speed up the computation. The popcount calculates hamming weight or the number of the amount of bits with value "1" in a register. The use of popcount instead of multiplication saves energy and potentially chip area, if hardware multiplier is not needed for the use case. Current hardware support for popcount is mostly targeted towards Graphics Processing Units (GPUs), which rules out

```
unsigned int popcount_emulated(unsigned int x) {
    x = x - ((x >> 1) & 0x55555555);                // 0x55 is 01010101
    x = ((x >> 2) & 0x33333333) + (x & 0x33333333); // 0x33 is 00110011
    x = (x + (x >> 4)) & 0x0F0F0F0F;                // 0x0F is 00001111
    x = (x + (x >> 16));
    return (x + (x >> 8)) & 0x0000003F;             // 0x3F is 00111111
}
```

Listing 1: Software emulation of popcount instruction

IoT applications. Despite the lacking hardware support the authors were able to test their algorithm with software emulated popcount, shown in listing 1, and with RISC-V ISA simulator. In simulation they tested for 55% improvement in execution with hardware popcount instruction compared to emulated popcount. In collaboration with Tampere University, the popcount instruction is implemented in this work to measure the speedup with synthesized RISC-V core.

The emulated popcount algorithm in listing 1 counts the number of "1" bits in 32-bit register with divide and conquer approach. Bit masks are used to filter out unwanted bits. On the first row the register is divided to two bit long bins, each bin containing the amount of bits in in the original register value at each bin location. On the second row the two bit bins are shifted on top of each other and added together producing 4-bit bins. Note that maximum value in each 4-bit bin is only 3-bits wide. The process of shifting and adding the bins is repeated until the values contained in the bins have been added to the beginning of the register. Each time the bin size is increased by a power of two, while the length of the maximum value increases by one bit. Finally, the maximum count of "1" bits is contained in first six bits of the register since 32_{10} equals to 100000_2.

3.1 Toolchain extension

Before hardware implementation of the popcount instruction, the RISC-V GNU toolchain was modified to allow software development with the new instruction. Popcount was

```
unsigned int popcount(unsigned int value) {
    unsigned int result;
    asm volatile
    (
      "pcnt %[z], %[x], %[y]\n\t"
      : [z] "=r" (result)
      : [x] "r" (value), [y] "r" (value)
    );
    return result;
}
```

Listing 2: Using popcount assembly in a C program

added to binutils and to the RISC-V ISA simulator, Spike. This allows the use of popcount instruction via assembly and simulation of programs utilizing popcount without actual hardware. While adding new instructions to the assembler is somewhat trivial, adding custom instruction support for the compiler is more involved and out of scope for this work. The popcount hardware instruction was added to the CNN C-code as embedded assembly.

A version of the CNN code from [13] was kindly shared by the authors. The received code was slightly modified to include all the CNN weights and the test image inside a single binary instead of separate files. Having a single test binary is preferable from the HDL testbench perspective, since it is more convenient to directly load it to the memory. Additionally, to compare the speed up gained with hardware popcount, two versions of the CNN were compiled. First version was compiled with the software emulated popcount show in listing 1 and the second version was compiled with hardware popcount instruction shown in listing 2. Both versions were first tested with Spike and the CNN with emulated popcount was also tested in SCR1 RTL simulation, before implementing the ISA extension.

Popcount instruction is planned as part of official RISC-V bit manipulation extension (B) standard, but at the time of writing the standard has not yet solidified. To align with current draft version of the bit manipulation extension *pcnt* mnemonic was chosen for

this popcount instruction implementation. The RISC-V ISA specification guarantees that certain instruction encodings will be left unused. These unused encodings are specifically meant for implementation of custom instructions in order to avoid conflict with possible future standard extensions. Custom-0 encoding space was chosen as recommended by the RISC-V instruction set manual for 32-bit implementations. For custom-0 encoding space, the first seven bits of the RISC-V instruction are set to 0001011_2. Since popcount requires only source and destination register, bits 19-15 and bits 11-7 of the instruction respectively, the other fields were fixed to zero.

3.2 Hardware extension

After selecting the instruction encoding format the RTL code of the SCR1 core was modified to support the new instruction. Instruction decoding unit was modified to accommodate for the new popcount instruction and a single cycle implementation of the popcount was added in to the Arithmetic-Logic Unit (ALU). The ALU muxes the correct operation to output depending on the instruction. The adder structure is similar to the parallel incrementer evaluated in [14]. The implementation sums the bits of the register in parallel as shown in figure 3.1. In first layer pairs of one bit values are added in parallel to form two bit values, in the second layer two bit pairs are added to form three bit values and so forth. Because the maximum value for popcount on 32-bit register is 32 (100000_2), only 6-bits are required to represent the final result. The final register shown in the figure is not actually required since the value can be directly assigned to 32-bit output of the ALU. The verilog code to generate this structure is shown in listing 3. Since the hardware implementation was not on critical timing path no further optimization was done.

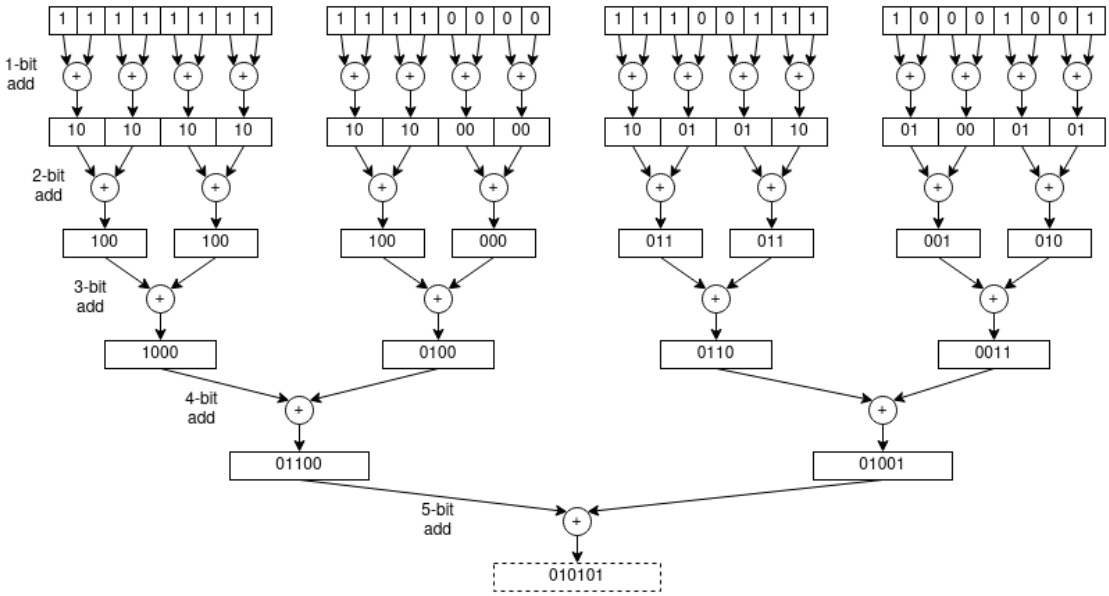


Figure 3.1: Parallel popcount example

```
genvar i;
generate
    for (i = 0; i <= 30; i = i + 2) begin : gen_pcnt_L0
        assign pcnt_L0[i+1:i] = ialu_op1[i+1] + ialu_op1[i];
    end
    for (i = 0; i <= 28; i = i + 4) begin : gen_pcnt_L1
        assign pcnt_L1[i+3:i] = pcnt_L0[i+3:i+2] + pcnt_L0[i+1:i];
    end
    for (i = 0; i <= 24; i = i + 8) begin : gen_pcnt_L2
        assign pcnt_L2[i+7:i] = pcnt_L1[i+7:i+4] + pcnt_L1[i+3:i];
    end
    for (i = 0; i <= 16; i = i + 16) begin : gen_pcnt_L3
        assign pcnt_L3[i+15:i] = pcnt_L2[i+15:i+8] + pcnt_L2[i+7:i];
    end
endgenerate

always_comb begin
    pcnt_res = '0;
    if (ialu_cmd == SCR1_IALU_CMD_PCNT) begin
        pcnt_res = pcnt_L3[31:16] + pcnt_L3[15:0];
    end
end
```

Listing 3: Hardware implementation of popcount

Chapter 4

Latch sequencing

Memory elements are fundamental for the operation of all sequential digital circuits. In addition to storing information from previous states they are used to synchronize various signals traveling through combinational logic stages. Signals traveling through long and complex combinational paths take longer to arrive at an endpoint than signals traveling through short and simple combinational paths. Memory elements are needed to slow down the faster signals in order to synchronize all inputs for the next state to a stable value. If the outputs of previous computation stage were not stable when the computation of next state begins we are bound to get invalid results. From this perspective the main purpose of memory elements is to separate previous state of the computation from the next. While there are many different memory elements most CMOS systems use only edge-triggered flip-flops, transparent latches and pulsed latches [15]. This chapter focuses on edge-triggered flip-flops and transparent latches. Behavioral differences between latches and flip-flops can be seen in figure 4.1. Figure 4.2 shows traditional flip-flop and latch implementations from [15]. Note that output signal of the flip-flop design is inverted. For the latch circuit in figure 4.2 (b) the inverter between clock signal and PMOS, can be moved between clock signal and NMOS to produce a latch operating in opposite phase.

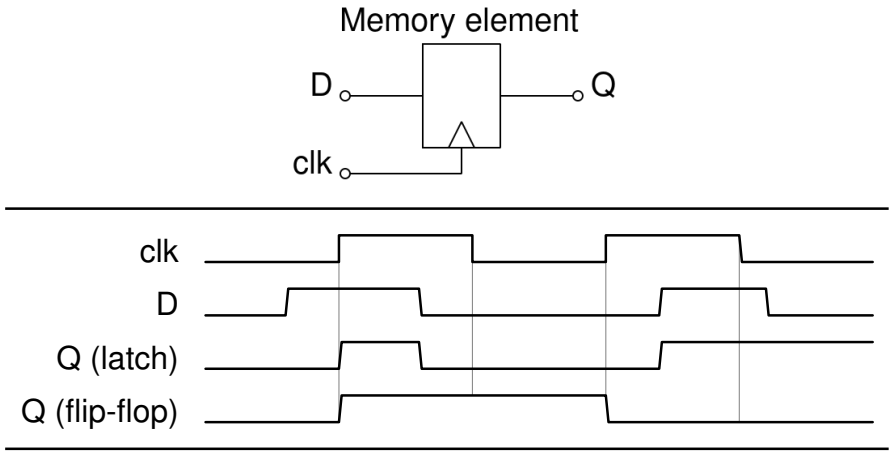
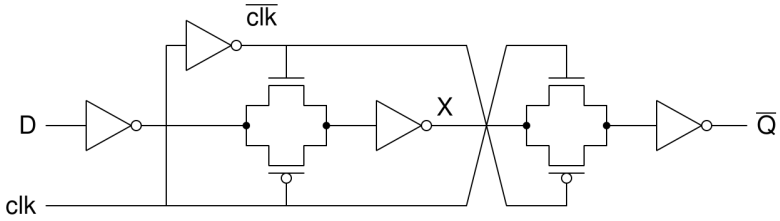
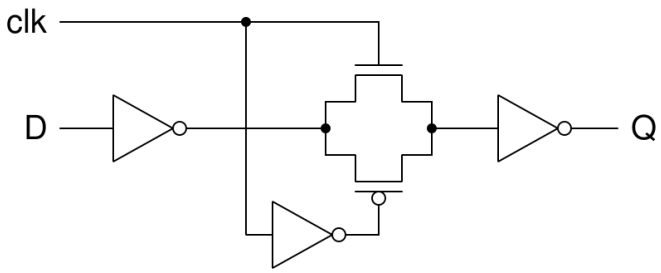


Figure 4.1: Transparent latch and positive edge triggered flip-flop timing behavior



(a) Flip-flop



(b) Latch

Figure 4.2: Traditional flip-flop and latch implementations

4.1 Flip-flop

Flip-flops are edge triggered devices, i.e. they allow the value at the input to propagate to the output only when an edge occurs in the control signal driving the flip-flop. Positive edge triggered flip-flops are the most common way to sequence program execution due to their simple timing behavior. While negative edge triggered flip-flops are also possible, they are not as common as positive edge triggered flip-flops. Mixing positive and negative edge triggered flip-flops can be done, but it is avoided in general to keep the timing of the circuit simpler, unless there is a good reason for it.

4.2 Transparent latch

Latches have varying properties depending on the clocking scheme. For example pulsed latches are driven by short clock pulses and usually accompanied by local pulse generators, while transparent latches are usually driven by longer clock pulses in at least in two different phases. This work focuses on transparent latch design and all references to latches mean transparent latches, unless specified otherwise. For simplicity the focus is on latches operating in two complementary phases with 50% duty cycle clocks.

When the control signal is active the latch is said to be transparent (or open). While the latch is transparent, output signal of the latch follows the input signal. While the control signal is not active latch is said to be opaque (or closed). While the latch is opaque, the output signal is “locked in” and does not change with the input signal. Latches can be either active high or active low, but unlike flip-flops at least two alternating active phases are needed for sequencing. As a device latches are simpler to build and require less transistors than flip-flops. In fact two transparent latches placed back to back operating in opposite phases as show in figure 4.3 are behaviorally equivalent to a single flip-flop. Compared to flip-flops latches have a wide window during which a signal can propagate, which gives latches an useful feature called *time-borrowing*. As long as the input data

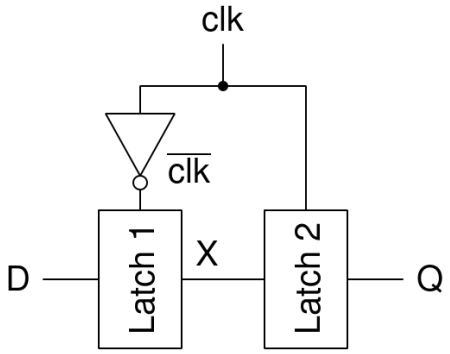


Figure 4.3: Two latches forming a functionally equivalent flip-flop

stabilizes to correct value before the latch becomes opaque the timing will not be violated. Because data does not have to stabilize before the active clock edge, the delays between latches do not have to be as evenly divided as delays between flip-flops. Instead the logic in some stages can borrow time from subsequent stages. The timing will be met if some parts of the subsequent stages are fast enough to compensate for the delays introduced in the slower stages.

4.3 Timing behavior

In digital circuit design the timing of the circuit is the most important aspect of the design. If the design fails to meet it's timing requirements the manufactured chip will not be functional. While in digital domain signals can have only values 0 and 1, in real devices signal transitions are not instantaneous. Each cell in the design has a *propagation delay*, which is the time it takes for a change in input signal value to appear on the output of the cell. All cells and wires, through which the signal propagates, add delay to the signal. To successfully capture a signal value to a memory element the signal must be stable while the value is being stored. Otherwise the captured values become unpredictable causing the system to fail.

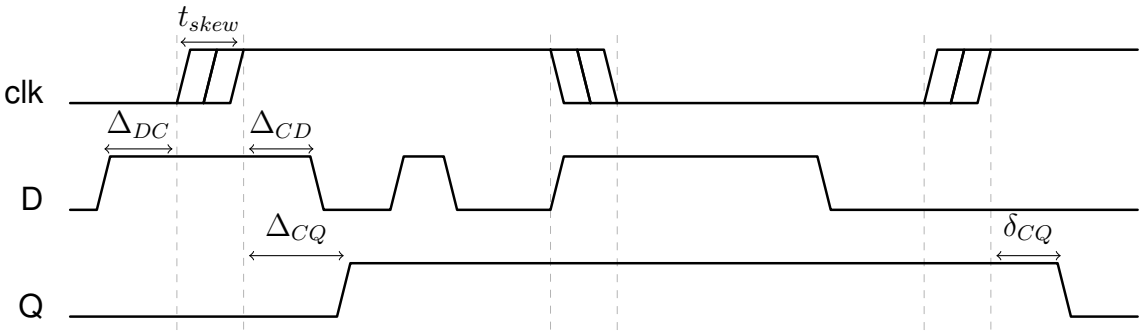
Sequential systems introduce overhead which increase the clock cycle time from three sources: propagation delay, setup time and clock skew [15]. The time a signal must be

stable before an edge is called setup time (Δ_{DC}) and the time a signal must be stable after an edge is called hold time (Δ_{CD}). If a signal does not meet setup or hold requirements the captured value becomes unpredictable. The maximum clock frequency for which a path can meet the timing requirements is limited by *max-delay* consisting of the total propagation delay of logic (Δ_{logic}) on the path and a *sequencing overhead* ($T_c = \Delta_{logic} + overhead$). If two or more memory elements are placed back-to-back without any logic between them, a common way to delay a signal, care must be taken to make sure the signal takes at least *min-delay* amount of time to propagate. Otherwise the subsequent memory elements might capture an incorrect value. While a system violating max-delay can still function at lower frequencies, usually min-delay violations cannot be compensated afterwards and they can render the system inoperable. Even for relatively small systems distributing a completely synchronous clock signal to all clocked components is not possible in practice. The clock will arrive to different parts of the system at slightly different times. Therefore some margin needs to be added for compensating the clock skew t_{skew} to ensure correct operation. For example, if the clock arrives late to a memory element and early to a subsequent memory element, the time the signals has for propagating through the logic between the memory elements is reduced.

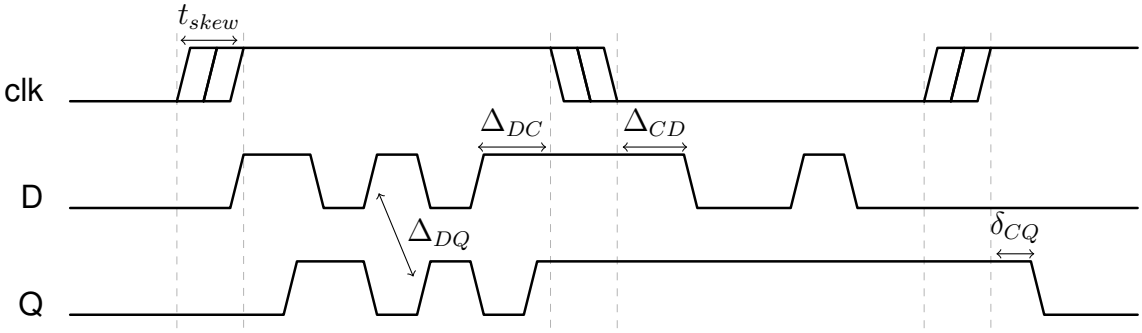
Table 4.1[15] shows comparison of flip-flop and latch performance. For latches the $t_{nonoverlap}$ denotes the time between clock falling and rising edge of subsequent stages. For 50% duty cycle two-phase complementary clocks the $t_{nonoverlap}$ is zero. Reducing the duty cycles of the clocks would increase the non-overlap time. For sequencing overhead of flip-flops the data propagation delay at clock edge Δ_{CQ}, setup time before the clock rises again Δ_{DC} and clock skew t_{skew} must be taken in to account. Sequencing overhead of latches depends only on propagation delay $2\Delta_{DQ}$ (two latches in clock cycle T_c). The min-delay requirement for both latches and flip-flops are quite similar. The δ_{CQ} denotes the minimum time from clock switching until the data at the output becomes valid. The related delays are visualized in figure 4.4.

Element	Sequencing overhead	Time borrowing	Min-delay
Flip-flop	$\Delta_{CQ} + \Delta_{DC} + t_{skew}$	0	$\Delta_{CD} + t_{skew} - \delta_{CQ}$
Transparent latch	$2\Delta_{DQ}$	$\frac{T_c}{2} - \Delta_{DC} - t_{skew} - t_{nonoverlap}$	$\Delta_{CD} + t_{skew} - \delta_{CQ} - t_{nonoverlap}$ (in each half-cycle)

Table 4.1: Flip-flop and latch performance



(a) Flip-flop timing



(b) Latch timing

Figure 4.4: Positive edge triggered flip-flop and transparent latch timing delay components visualized

4.4 Process variations

Essentially, all properties of all components of a circuit are subject to Process, Voltage, and Temperature (PVT) variations. Detailed PVT analysis is out of scope for this work, and here the clock skew is surrogate for all PVT variations. Depending on the location of the memory elements, the phase of the clock signal is skewed slightly. While careful clock tree design can mitigate the clock skew it cannot be completely eliminated. Process variations, which are especially troublesome in low-power designs, reduce the manufacturing yields. Latches offer some benefits over flip-flops for designs which are subjected to large manufacturing process variations. Hurst *et. al* have shown in [4] that transparent latches can reduce the rate of failures by four times compared to flip-flops. While under traditional timing models the maximum performance of latches does not differ from flip-flops, under high variation and statistical timing model latches have advantage over flip-flops. Because each manufactured instance has unique timing characteristics it is impossible to create an optimal timing for every scenario at design time. Instead only the best solution for largest number of devices can be selected. Flip-flops are affected by variations more than transparent latches, because they can only capture value at the clock edges. Transparent latches have a much wider window for the signals to propagate, allowing a larger number of the possible variations to meet the timing.

Chapter 5

Latches — replacing flip-flops

As explained on previous chapter latches have some advantages over flip-flops for designs affected by large manufacturing process variations. Because latches are also simpler devices it is expected that latch-based design would also have smaller area than equivalent flip-flop-based design. Since no open-source latch-based RISC-V cores were available, SCR1 was previously selected as a target for transforming the flip-flop-based logic to latch-based logic, while keeping the core behaviorally intact. This chapter presents an HDL approach developed for transforming the flip-flop-based core to a latch-based core. A SystemVerilog analysis tool and simple python preprocessor were developed to generate signal dependency graphs. The analysis tool replaces each flip-flop vertex in the graph with a single latch, assigns phases to the signals and detects feedback loops which need additional latch inserts.

Synthesis tools transform the RTL description of the circuit to gate level representation by mapping the described logic to cells. The cells are picked based on what kind of logic the RTL description implies, the properties of available cells and the given constraints. In order to synthesize latches instead of flip-flops and vice versa, the RTL has to be written in a way that implies the desired type of cells. While it is possible to specify cells explicitly, doing so at scale quickly becomes tedious and error prone. Explicit definitions are also tied to specific cell libraries and reduce portability between different libraries.

The SCR1 core is written in SystemVerilog. In SystemVerilog it is possible to express designer intent more strongly than some in other HDLs commonly used in the industry, such as Verilog or VHDL. SystemVerilog standard [16] specifies separate `always_ff` and `always_latch` procedures which allows user to define explicitly the *intent* to generate flip-flops or latches. However, if the statements inside `always_ff` or `always_latch` procedures do not also *imply* flip-flops or latches then something else will be generated. SystemVerilog does not enforce checks on whether logic inside `always_latch` block is really representing a latch, but it urges tools to check and generate warnings in case of mismatch. Separate blocks for flip-flops and latches improve readability and help with debugging.

5.1 Methodology

Latch-based design requires a slightly different methodology than flip-flop-based design. Usually most flip-flops operate in the same clock phase, while in transparent latch designs the latches operate in alternating phases.

In latch-based circuit operation each signal is either stable when the driving latch is opaque or unstable when the driving latch is transparent. Wide signal propagation window of transparent latches introduces some constraints that are not present in flip-flop-based designs. In flip-flop designs connecting two positive edge triggered flip-flops back-to-back delays the signal by one clock cycle. Connecting two latches operating in a same phase rarely make sense, because this would create a one long path, where the second latch acts as a delay buffer. To keep the circuit “sane” there are several rules on how the signal phases can be connected. Connection rules are shown in table 5.1 in terms of HDL assignments, where “RHS” denotes right-hand side of the assignment and “LHS+cond” denotes left-hand side of the assignment and conditionals affecting the assignment. The rationale for forbidden connections is shown in table 5.2. In HDL code it is perfectly

RHS/LHS+cond	Comb. s1	Comb. s2	Latch s1	Latch s2
Comb. s1	Yes	No	No	Yes
Comb. s2	No	Yes	Yes	No
Latch s1	Yes	No	No	Yes
Latch s2	No	Yes	Yes	No

Table 5.1: Allowed and forbidden connection matrix.

legal to mix phases in a way that can mess up the synchronization of the signals. Due to differences in timing behavior writing latch-based HDL code requires a slightly different mindset than writing flip-flop code. Managing multiple phases increases the complexity from timing perspective compared to traditional positive clock edge triggered flip-flop design. Coherent signal names are very valuable for debugging signal phase related issues in the design. A signal naming scheme shown in table 5.3 based on scheme proposed in [15] was adopted. Adding a suffix indicating the signal phase and assignment type helps to identify timing problems. Suffixes "_s1" and "_s2" indicate continuous assignment stable in phase 1 and phase 2 respectively. Additional letter "l" indicates that the signal originates from an assignment implying a latch. Special cases "_l2t1" and "_l1t2" signify a latch inserted to delay the signal by a half-cycle and change the phase of the signal from phase 2 to phase 1 and from phase 1 to phase 2, respectively, without any logic in between.

5.2 Signal dependencies

Signal dependencies of a circuit described in RTL can be represented by a directed graph, where each vertex is a signal node and directed edges represent dependencies formed between the vertices by assignments. The underlying functions of the assignments are

Forbidden connection	Reason
1. Latch → latch, same phase	Second latch acts as a buffer instead of synchronization element.
2. Comb → latch, same phase	Latch acts as a buffer instead of synchronization element.
3. Comb → comb, different phase	Signal becomes unpredictable, because phases are mixed.
4. Latch → comb, different phase	Comb signal is not stable in the intended phase.

Table 5.2: Rationale for forbidden connections for two phase transparent latch design

Output type	Output stable	Input type	Input stable	Output suffix
Comb	Phase 1	Comb or Latch	Phase 1	_s1
Comb	Phase 2	Comb or Latch	Phase 2	_s2
Latch	Phase 1	Comb	Phase 2	_s1l
Latch	Phase 2	Comb	Phase 1	_s2l
Latch	Phase 1	Latch	Phase 2	_l2t1
Latch	Phase 2	Latch	Phase 1	_l1t2

Table 5.3: Signal naming methodology

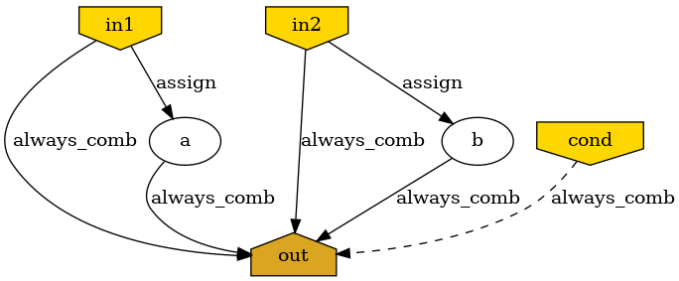
not relevant for the dependency graph. Signal assignments can be either sequential or combinational and either conditional or direct dependencies. Propagation of sequential assignments is controlled by a clock signal and combinational assignments propagate constantly. Dependency is conditional if the signal is not directly assigned to dependent signal, but it has an effect to the evaluation of the assignment via conditional statements. Each vertex and edge in the graph is "colored" to be either sequential or combinational as defined by the RTL assignment expression. Additionally each edge can be direct or conditional. Example visualizations of graphs and of the RTL code the graph was generated from can be seen in figure 5.1. Input and output ports are represented by bright and dark yellow nodes. Signals values flow from input to output. As can be seen in figure 5.1 (a) both `assign` statements and blocking assignments inside `always_comb` blocks result in combinational dependency. Dashed edges denote conditional dependency. In figure 5.1 (b) all assignments are sequential. Notice that the `cyc2` signal has a dependency to itself, because it needs the previous value of itself to evaluate correctly, depending on the state of `or_prev` signal. With flip-flops feedback from cell output to it's input is allowed, because the signal can only propagate on clock edge.

5.3 Transform

Transforming flip-flop-based design to a latch-based design could be accomplished by simply replacing all instances of flip-flops with two back-to-back latches. On the other hand the transformation could be done by replacing a flip-flops with just one latch and then adjusting the signal phases and inserting latches to feedback paths when necessary. The latter approach was chosen, because the resulting RTL code is truly in latch-based style.

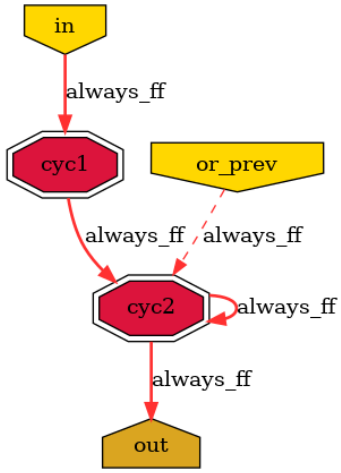
Before the input files are parsed by the analysis tool, they are run through a preprocessor. A simple python preprocessor was developed for the purpose of making the job

```
module comb_example(
    input  logic in1,
    input  logic in2,
    input  logic cond,
    output logic out
);
    logic a;
    logic b;

    assign a = !in1;
    assign b = !in2;

    always_comb begin
        if (cond) begin
            out = in1 & in2;
        end
        else begin
            out = a & b;
        end
    end
endmodule : comb_example
```



(a) Combinational example

```
module seq_example(
    input  logic clk,
    input  logic rst_n,
    input  logic in,
    input  logic or_prev,
    output logic out
);
    logic cyc1;
    logic cyc2;

    always_ff @(posedge clk) begin
        if (~rst_n) begin
            cyc1 <= 1'b0;
            cyc2 <= 1'b0;
            out  <= 1'b0;
        end
        else begin
            cyc1 <= in;
            if (or_prev == 1'b1) begin
                cyc2 <= (cyc2 | cyc1);
            end
            else begin
                cyc2 <= cyc1;
            end
            out <= cyc2;
        end
    end
endmodule : seq_example
```



(b) Sequential example (clk and rst_n left out for clarity)

Figure 5.1: Visualization of assignment graphs generated from RTL

of analysis tool easier. The preprocessor handles expansion of macros, preprocessor conditionals and include statements. Preprocessor outputs a single file which combines all the included files, has all macro values expanded and contains only the blocks selected by conditional macros.

A SystemVerilog analysis tool was developed to identify logical dependencies in the circuit. The tool was built with ANTLR4 parser generator. Given a grammar description of a formal language ANTLR4 generates lexer and parser. First the lexer tokenizes the input for the parser. Each token represents character strings of the code with distinct meaning, such as keywords, identifiers, comments, etc. Based on the token stream the parser then generates a parse tree data structure describing the code in a structured manner.

Lexer and parsers are commonplace in compiler front-ends and various tools have been developed for generating parsers from formal grammar definitions. Complex languages, such as SystemVerilog, often have relatively complicated grammar. Fortunately Carr *et al.* have developed SystemVerilog ANTLR4 grammar for Microsoft's static analysis platform, gNOSIS [17], and released it under MIT license. Additionally the grammar of SystemVerilog is comprehensively described in [16]. While the grammar file was in quite good shape, it required minor tweaks (appendix A) before the code was parsed properly. Given a grammar description, ANTLR4 generates a parser skeleton, which by itself does nothing with the parse tree. Additional functionality was added to the parser to generate a directed graph of the signal dependencies with JGraphT library while traversing the parse tree.

As the parse tree is being traversed, for each new signal encountered a vertex representing the new signal is added to the graph. If a vertex corresponding to the signal already exists, only the edges representing dependencies are updated. Vertices or signals from which the assignment depends are connected by a directed edge. Dependencies are categorized either as direct dependencies or conditional dependencies. Direct dependencies of an assignment are defined by the right-hand side of the assignment statement, i.e.

what is being assigned to the signal node. Signals can also be conditionally dependent on another signal. For example, if an assign statement for signal A is chosen based on value of signal B, but the signal B is not directly assigned to signal A, a conditional dependency is created. After the graph is constructed clock and reset signal nodes are removed, because they are not dependent on any other signal.

Then all sequential nodes are treated as latches and nodes of the graph are then ranked by whether the connections are valid or not based on the connection rules show in table 5.1. The direction of the graph is reversed and the graph is traversed depth-first from outputs to inputs until all vertices have been assigned a score. The purpose of the ranking is to assign signal phases in a way that the number of invalid connections will be small before adding any additional latches to fix the sequencing. For the purpose of this work a rather simple scoring scheme was created by experimentation and it is by no means guaranteed to be optimal. Score of a node is incremented for valid connections and decremented for invalid connections based on the initial phases and signal type. After all nodes have been scored, the nodes are assigned phases starting from the node with highest score (most valid connections) to the node with lowest score (least valid connections). Each time a node is assigned a phase, also unassigned nodes with connecting edges are assigned to suitable phase.

After the nodes have been assigned phases, additional latches need to be inserted to fix the remaining invalid connections. For example, a flip-flop with feedback from it's output to it's input cannot be replaced by a single latch. An additional latch needs to be inserted in the feedback path to preserve the correct sequencing. Actual implementations of even a relatively simple cores such as SCR1 have a huge number of connections and visualizing them as a graph is not very useful. Instead of a graph, the analysis tool outputs a list of signals along with assigned phases and required latch inserts for the module. The RTL code was then modified by hand based on the output of the analysis tool. In order to reduce the tool complexity and development time, the analysis tool is limited to operating

on a single module at a time and the process has to be repeated for each module. Assigning the phases of the whole design at the same time might have yielded more optimal results.

5.4 Latch synthesis

Writing SystemVerilog code that synthesizes to flip-flop-based logic is quite straightforward and well supported. In the case of sequential latch code the situation is not as clear. How well the surrounding logic synthesizes is dependent on the cell libraries and how the latch is described inside the module.

Simple latches with just a clock and data inputs synthesize as expected. When reset was added the synthesis results became somewhat unpredictable. Even though there was a latch cell with reset in the cell library the synthesis tool created ad hoc reset logic around a reseteless latch cell instead. In general, writing latches with the same style as flip-flops, seems to produce unnecessarily complicated logic as seen in figure 5.3. Separating the combinational logic outside `always_latch` block as combinational statemets produced simpler logic, shown in figure 5.2. Despite many trials the synthesis tool refused to synthesize latch logic with latch cells that actually have a reset input. In the end the desired results were achieved by hardcoding the desired library cells as seen on figure 5.4.

Because the synthesis of latch-based logic is somewhat unpredictable, a wrapper module was created for each type of utilized latch components. The module is parameterized for size and reset value to enable generic usage. A wrapper for latch with output stable at phase 1 is shown in listing 4. All combinational logic inside `always_ff` blocks was moved to separate `always_comb` blocks which were connected to the latch wrappers as shown in listing 5. An enable signal was added to the latch wrapper to allow retaining the data as needed by gating the clock locally. During synthesis the latch wrappers were synthesized by themselves first and `size_only` constraint was set on all cells inside the wrappers. This prevents the synthesis tool from swapping the cells to something un-

```
1  module test_latch_reset(
2      input  logic clk,
3      input  logic reset,
4      input  logic d,
5      output logic q
6  );
7      logic a;
8
9      assign a = (reset) ? 1'b0 : d;
10
11     always_latch begin
12         if (clk || reset) begin
13             q <= a;
14         end
15     end
16 endmodule : test_latch_reset
```

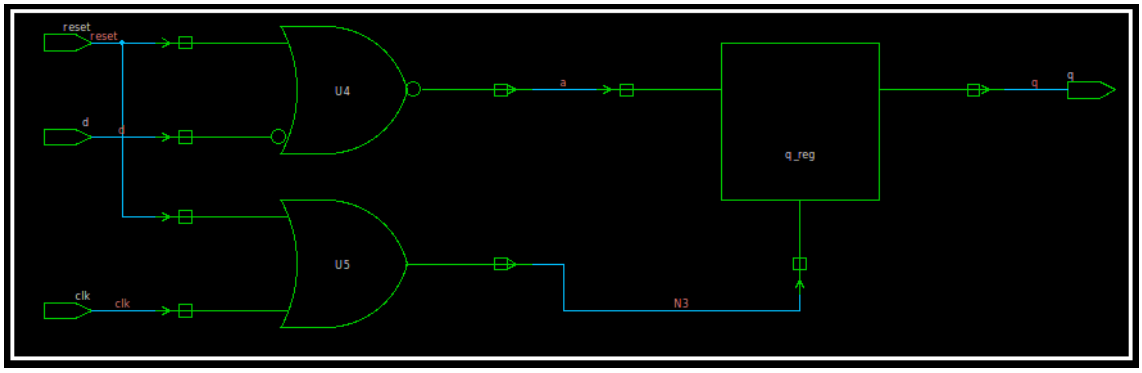


Figure 5.2: Synthesis of latch with reset. Output value assigned outside the always_latch block.

```
module test_latch_reset2(
    input  logic clk,
    input  logic reset,
    input  logic d,
    output logic q
);
    always_latch begin
        if (reset) begin
            q <= 1'b0;
        end
        else if (clk) begin
            q <= d;
        end
    end
endmodule : test_latch_reset2
```

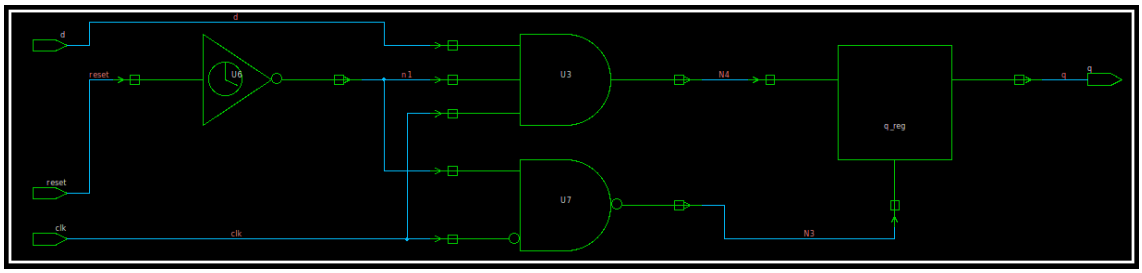


Figure 5.3: Synthesis of latch with reset written in flip-flop style.

```
module test_latch_reset4(
    input  logic clk,
    input  logic reset,
    input  logic d,
    output logic q
);
    latch_cell q_latch ( .GN(!clk), .RN(!reset), .D(d), .Q(q) );
endmodule : test_latch_reset4
```

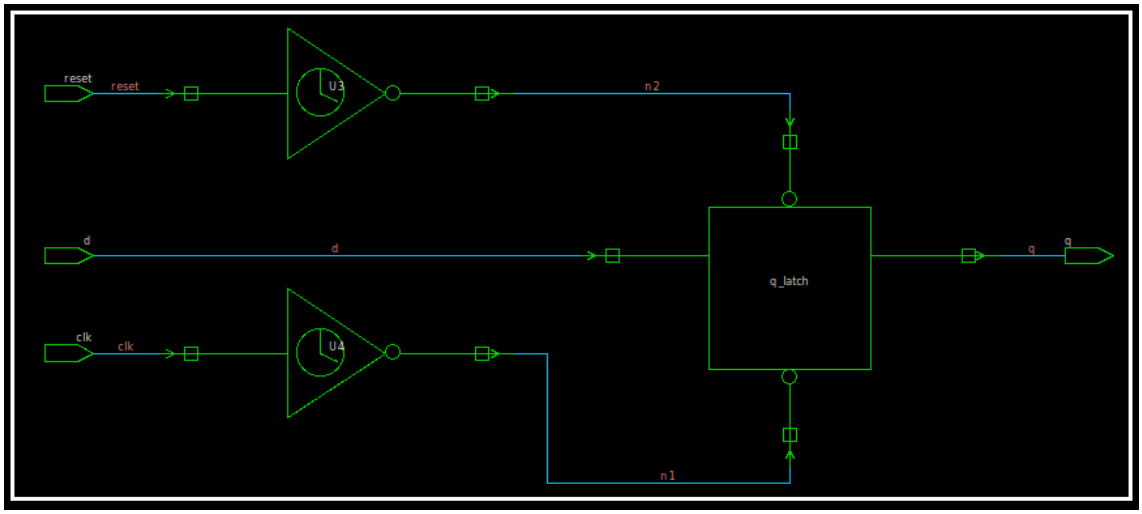


Figure 5.4: Synthesis of latch with reset written with hardcoded latch cell. The actual cell name is not shown.

wanted, while allowing the cells to be resized for different drive strengths. The rest of the design was then synthesized after processing the latches to achieve desired results. However, for some reason the synthesis tool did not time the clock paths for the latches correctly. According to the synthesis tool there were no timing violations, but when the design was simulated it was apparent that the delay added by enable signal clock gate was not taken in to account. Since the synthesis tool itself did not see any errors the issue proved quite hard to debug. As a workaround the delay added by the gating cells were overridden with a smaller value than in the real model. Since there are only 1-2 additional cells added just before the latch clock pin for latches with enable the simulated timing is still close to what the actual values would be. This issue might have been easier to solve in place-and-route stage, but this was not tested due to limited scope of this work.

```
`ifndef VERILATOR
`define SCR1_LATCH_S1_STRUCT
`endif //VERILATOR
module scr1_latch_s1 #(
    parameter SCR1_LATCH_BITS = 1, SCR1_LATCH_RESET_VALUE = 0
) (
    input  logic clk,
    input  logic rst_n,
    input  logic en,
    input  logic [SCR1_LATCH_BITS-1:0] d,
    output logic [SCR1_LATCH_BITS-1:0] q
);
`ifdef SCR1_LATCH_S1_STRUCT
localparam bit [SCR1_LATCH_BITS-1:0] rst_val =
 ↪  (SCR1_LATCH_BITS)'(SCR1_LATCH_RESET_VALUE);

logic clk_n;
logic clk_n_gate;
logic clk_n_gate_n;
logic [SCR1_LATCH_BITS-1:0] din;
logic [SCR1_LATCH_BITS-1:0] dout;

// Clock gate active low, latch active low
inverter_cell inv_clk ( .A(clk), .Z(clk_n) );
and2_cell and2_clock_gate ( .A(clk_n), .B(en), .Z(clk_n_gate) );
inverter_cell inv_clk_gate ( .A(clk_n_gate), .Z(clk_n_gate_n) );

generate
genvar i;
for (i = 0; i < SCR1_LATCH_BITS; i = i + 1) begin : q_latch_s1
    // If latch bit has non-zero reset value, it needs to be inverted
    // before and after latch. No need to invert for zero reset value.
    if (rst_val[i] == 1'b0) begin : rst_bit_0
        always_comb begin
            din[i] = d[i];
            q[i] = dout[i];
        end
    end
    else begin : rst_bit_1
        inverter_cell inv_din ( .A(d[i]), .Z(din[i]) );
        inverter_cell inv_dout ( .A(dout[i]), .Z(q[i]) );
    end
    latch_cell q_latch_s1 ( .GN(clk_n_gate_n), .RN(rst_n), .D(din[i]),
     ↪  .Q(dout[i]) );
end
endgenerate

`else //~SCR1_LATCH_S1_STRUCT
logic clk_gate;
logic [SCR1_LATCH_BITS-1:0] dout;

always_comb begin
    clk_gate = ~rst_n || (en && ~clk);
end

always_comb begin
    if(~rst_n) begin
        dout = (SCR1_LATCH_BITS)'(SCR1_LATCH_RESET_VALUE);
    end
    else begin
        dout = d;
    end
end

always_latch begin
    if (clk_gate) begin
       q <= dout;
    end
end

`endif //SCR1_LATCH_S1_STRUCT
endmodule : scr1_latch_s1
```

Listing 4: Parameterized latch wrapper with output stable at phase 1. Actual cell names are not shown.

```
always_ff @(posedge clk, negedge rst_n) begin
    if (~rst_n) begin
        init_pc_v_s2l <= '0;
    end else begin
        if (~&init_pc_v_s2l) begin
            init_pc_v_s2l <= {init_pc_v_s2l[2:0], 1'b1};
        end
    end
end
```

(a) Original flip-flop block

```
always_comb begin
    init_pc_v_s2l_enable_s1 = 1'b0;
    init_pc_v_s2l_next_s1 = {init_pc_v_l2t1[2:0], 1'b1};

    if (~&init_pc_v_l2t1) begin
        init_pc_v_s2l_enable_s1 = 1'b1;
    end
end
scr1_latch_s2 #($bits(init_pc_v_s2l)) i_latch_init_pc_v_s2l(
     .clk(clk),
     .rst_n(rst_n_s1l),
     .en(init_pc_v_s2l_enable_s1),
     .d(init_pc_v_s2l_next_s1),
     .q(init_pc_v_s2l)
);
```

(b) Resulting combinational logic and latch wrapper

Listing 5: Snippets showing the original flip-flop block and the latch version.

Chapter 6

Results

This chapter compares latch core and flip-flop core in terms of area, power, energy over different process corners. Only the cores were synthesized for these netlist simulations. The top level environment, including instruction and data memories were kept as part of the testbench. The synthesized core was connected to the testbench environment via I/O delay wrapper in order to simulate the actual environment more realistically. No clock gating was inserted by the synthesis tool to either core, but the latch core contains some inherent local clock gates due to the way it is structured as presented in previous chapter. The following results were obtained by simulating the synthesized cores.

- Latch core area is 17-20% smaller
- Target frequencies that caused flip-flop core area to increase, did not cause significant increases on latch core area
- Latch core internal power is 2.9-3.6 times smaller (SHA-256)
- Latch core switching power is 1.06-1.16 times larger (SHA-256)
- Latch core leakage power is 1.1-2.4 times larger
- Latch core total power is 1.6-2.1 times smaller

- SHA-256 activated the core most, followed by Coremark and Dhrystone 2.1 benchmarks
- Latch core energy consumption is 3.9-8.5 uW/MHz with activity from SHA-256 simulation
- Flip-flop core energy consumption is 8.2-14.0 uW/MHz with activity from SHA-256 simulation

The simulations of the original flip-flop-based core and the latch-based core were run with identical testset. The testbench is based on the testbench provided in Syntacore's SCR1 repository, with some minor augmentations. Switching activity for power estimation was extracted from RTL simulations and then mapped to synthesized netlist. The power calculations do not include clock tree power or wire loads, which adds some uncertainty to the power numbers.

Hold violations were corrected by a script which inserts buffers to violating paths and sets them as size only. After buffer insertion, the design was resynthesized with `-incremental` flag. The process of buffer insertion and incremental compile for flip-flops was repeated until no hold violations were reported. The latch core required fewer buffer insertions for a functioning simulation, but some non-critical hold violations were left in the design. As explained in the previous chapter the synthesis tool did not correctly time some of the clock paths during latch core synthesis, if any clock gates related to latch data retention were present. The script used to fix the timing errors for simulations most likely gives the latch-based core a slight advantage, but since there were only 1-2 affected cells placed between the clock signal and latch clock pin per latch for some of the latches, the effect is rather small.

6.1 CNN and popcount

For CNN test a single round of image recognition was run. Using hardware popcount instruction as opposed to software implementation a $2.1\times$ performance boost was observed in the execution time as seen in table 6.1. This is in line with the observation that emulated popcount requires 21 instructions and the hardware popcount requires only 1, excluding function calls and return statements. While popcount reduces the computation time considerably, there is a significant amount of multiplications done in the program code for index calculations. Because the core was synthesized without a multiplier the CNN code was compiled with loop unrolling to reduce the execution time at the cost of additional memory footprint, resulting in 1.1Mb binary.

Popcount	Execution time (ns)	Execution time (ms)	Relative execution time
Software	185822789	186	1.0
Hardware	87757206	88	0.47

Table 6.1: CNN execution times @ 500MHz, TT 1V 25C

6.2 Synthesis corners

Both cores were synthesized for SS -40C, TT 25C and FF 125C corners. LVT cells were used in all synthesis trials. All corners were synthesized with 500MHz target and for each corner target frequency was increased until area started to increase rapidly. The frequency increase was limited by the flip-flop core. Same targets were then used for latch core. The flip-flop cores required more hold fix buffers for functional gate level simulations, but even after subtracting the inserted buffer area the latch core still has advantage in cell area. Area comparison for each target can be seen in table 6.2. Latch core area is roughly 20% smaller than flip-flop core area for all targets. Since the hold fix buffers were inserted in the synthesis stage before any cell placement information was present, the hold buffer

counts might differ from the actually required hold buffer counts. If the hold buffer area is not taken in to the account, the latch core had largest area advantage for FF 125C 500MHz target and the smallest area advantage for SS -40C 588.24MHz target.

Maximum time borrow for the latch core was constrained to 40% of the clock period. Timing issues started appearing when relaxing the time borrow constraint towards to 50% of the clock period. Actual maximum time borrow for each corner can be seen in table 6.3. The latch core is able utilize time borrowing quite consistently in all synthesis corners.

	FF 125C 1.30V		TT 25C 1.20V		SS -40C 1.15V	
Corner	714.29MHz	500.00MHz	625.00MHz	500.00MHz	588.24MHz	500.00MHz
Latch area	6979.411421	6857.990622	6863.974623	7076.569822	7145.984225	7146.963424
Flip-flop area	9157.913700	9031.379296	8879.168103	8746.867290	8847.507297	8919.532891
Area (latch core / flip-flop core)	0.762	**0.759**	0.773	0.809	0.808	0.801
Latch buffers	139	3	53	3	6	3
Flip-flop buffers	1232	758	1520	320	662	273
Buffers (latch core / flip-flop core)	0.113	**0.004**	0.035	0.009	0.009	0.011
Latch hold buffer area	60.492800	1.305600	17.299200	1.305600	2.611200	0.979200
Flip-flop hold buffer area	504.179200	323.571200	496.345600	115.545600	224.780800	89.216000
Hold buffer area (latch core / flip-flop core)	0.120	**0.004**	0.035	0.011	0.015	0.011
Latch area without buffers	6918.918621	6856.685022	6846.675423	7075.264222	7143.373025	7145.984224
Flip-flop area without buffers	8653.734499	8707.808095	8382.822503	8631.32169	8622.726497	8830.316891
Area without buffers (latch core / flip-flop core)	0.800	**0.787**	0.817	0.820	0.828	0.809
Area difference with buffers	2178.502279	2173.388674	2015.193480	**1670.297468**	1701.523072	1772.569467
Area difference without buffers	1734.815879	1851.123074	1536.147080	1556.057468	**1479.353472**	1684.332667

Table 6.2: Area comparison

	FF 125C 1.30V		TT 25C 1.20V		SS -40C 1.15V	
Corner	714.29MHz	500.00MHz	625.00MHz	500.00MHz	588.24MHz	500.00MHz
Max time borrowed (ns)	**0.56**	0.79	0.63	0.77	0.63	0.79
Max time borrowed / period	0.4	0.395	0.394	0.385	**0.371**	0.395

Table 6.3: Latch maximum time-borrow

6.3 Power and energy

Coremark, Dhrystone 2.1 and SHA-256 benchmarks were used for switching power estimates. For each benchmark and for both cores the switching activity was recorded in RTL simulation and forward annotated to the synthesis tool. Figure 6.1 shows total power comparisons between corners and benchmarks for both cores. SHA-256 benchmark caused the highest signal switching rate in the core in all simulation runs and was selected to present worst case activity. The power estimate comparison calculated with SHA-256 switching activity is shown in table 6.4. Considering that latch core has roughly 20% smaller area it is surprising that the latch core has significantly higher leakage power dissipation than flip-flop core. In the worst case at FF 25C 1.3V 500.00MHz target, the latch core leakage power is almost 2.4 times higher than the leakage power of the flip-flop core. And even in the best case at SS -40C 1.0V 588.24MHz the latch leakage power is over 1.1 times higher than in the corresponding flip-flop core. One reason for the higher leakage might be that the cell library does not have that many latch cell variants. In fact there was only one latch cell with reset in the library and a few other latch cell without reset. Cell libraries are typically more geared for traditional flip-flop-based design and the quality of latch cells might not be as good as flip-flop cells. The Flip-flop core comes ahead in terms of switching power as well. In the worst case at TT 25C 1.0V 500.00MHz target, the latch core switching power is over 1.1 times higher and in the best case at SS -40C 1.0V 500MHz target, the latch core leakage is 1.06 times higher. The time borrowing proprety of the latches might be a contributing factor to the higher switching power, because signals have a wider arrival window and potentially more toggling activity can take place before the signals stabilize. However, the latch core internal power is 3.6 times smaller in the best case at SS -40C 1.15V 500.00Mhz target and 2.9 times smaller in the worst case at FF 125C 1.3V 714.29MHz target. The internal power contains all power dissipated within cell boundaries. The internal power is mainly caused by short-circuit power for simple cells, but charging and discharging capacitances internal to the cell may dominate

for more complex cells. Because of the large internal power reduction the latch core has lower total power consumption in all corners, ranging from 2.1 times to 1.6 times lower total power consumption in the best and worst case respectively.

The cores were synthesized for several clock frequency targets. The total power typically increases along with higher target frequencies, if no other paremeters are adjusted. While it is possible that in some cases the increase in the clock frequency outweighs the increase in power and results in lower energy, here it was not the case. The synthesis targets with the lower 500.00MHz target frequency produced the most energy efficient cores. The best (lowest) and the worst (highest) core energies were found at SS -40C 1.15V 500.00MHz and FF 125C 1.3V 714.29MHz targets respectively. Overall, the latch core is roughly two times more energy efficient than the flip-flop core.

	FF 125C 1.30V		TT 25C 1.20V		SS -40C 1.15V	
Corner	714.29MHz	500.00MHz	625.00MHz	500.00MHz	588.24MHz	500.00MHz
Latch core internal power (mW)	2.393100	1.107500	1.529100	0.930300	1.222300	0.852900
Flip-flop core internal power (mW)	7.044200	3.623100	5.053100	3.270100	4.193000	3.081600
Internal power (latch core / flip-flop core)	0.340	0.306	0.303	0.284	0.292	**0.277**
Latch core switching power (mW)	2.443800	1.244100	1.855800	1.185600	1.615300	1.101700
Flip-flop core switching power (mW)	2.118100	1.138000	1.622200	1.018700	1.404400	1.042700
Switching power (latch core / flip-flop core)	1.154	1.093	1.144	1.164	1.150	**1.057**
Latch core leakage power (mW)	1.258100	0.498800	0.026355	0.013720	0.000804	0.000358
Flip-flop core leakage power (mW)	0.809700	0.208700	0.011452	0.006410	0.000711	0.000250
Leakage power (latch core / flip-flop core)	1.554	2.390	2.301	2.140	**1.131**	1.432
Latch core total power (mW)	6.095000	2.850400	3.411255	2.129620	2.838404	1.954958
Flip-flop core total power (mW)	9.972000	4.969800	6.686752	4.295210	5.598111	4.124550
Total power (latch core / flip-flop core)	0.611	0.574	0.510	0.496	0.507	**0.474**
Latch core energy (uW/MHz)	8.533000	5.700800	5.458008	4.259240	4.825287	3.909916
Flip-flop core energy (uW/MHz)	13.960800	9.939600	10.698803	8.590419	9.516789	8.249100
Energy (latch core / flip-flop core)	0.611	0.574	0.510	0.496	0.507	**0.474**

Table 6.4: Core power and energy estimates (SHA-256)

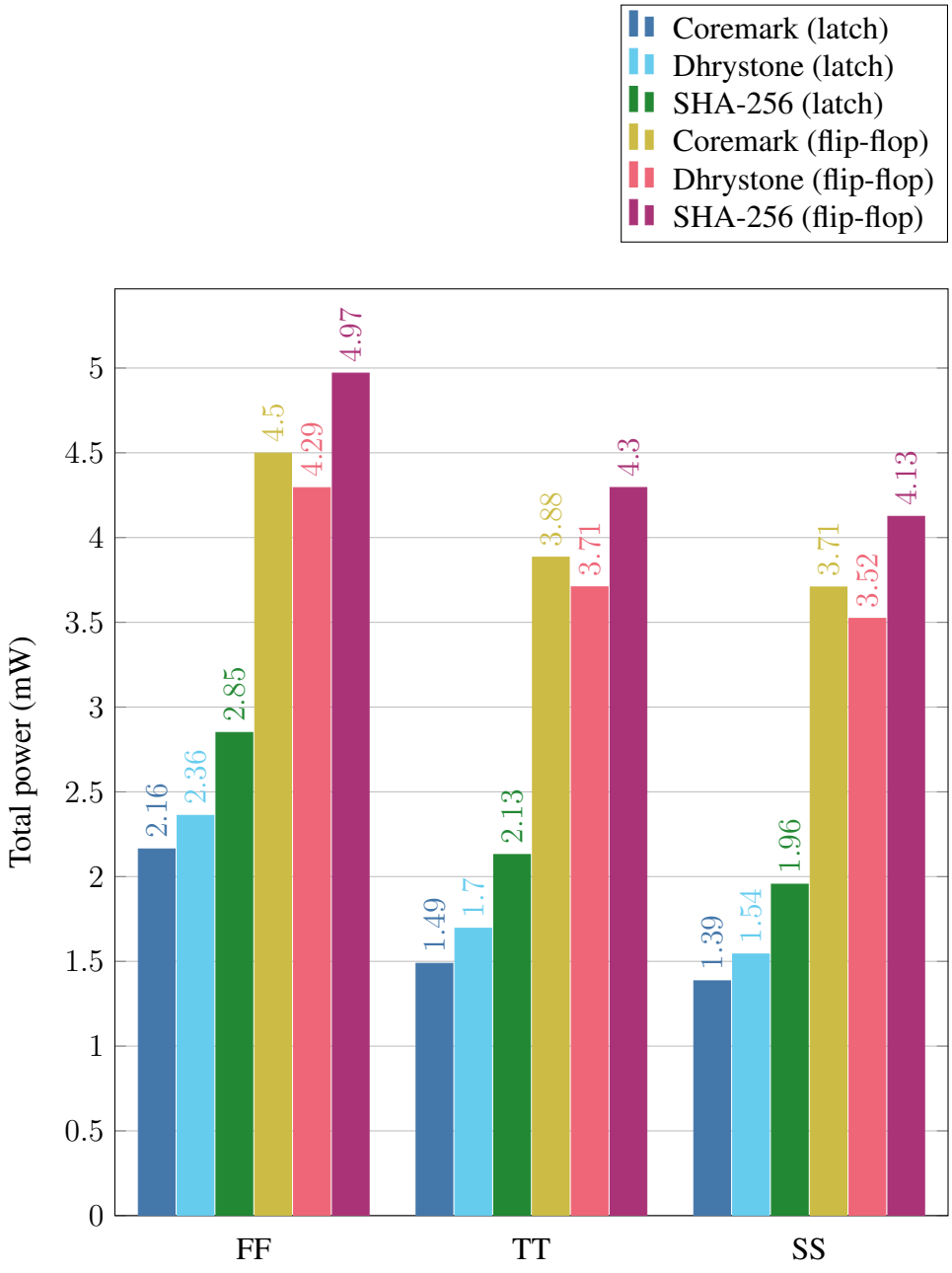


Figure 6.1: Total power estimates with Coremark, Dhrystone 2.1 and SHA-256

Chapter 7

Conclusions

Latch-based designs have a lot of potential for increasing the energy efficiency and decreasing manufacturing costs via reduced area and improved tolerance for process variations. The results are only indicative and further experimentation with different cell libraries and full flow, including the back-end, would be needed for more accurate energy predictions. Also the process variation tolerance could be confirmed by simulating the design over more corners and with statistical timing models. Overall the latch-based core was around 50% more energy-efficient and had 17-20% smaller area. The hold timing of both of the cores was fixed at the end of the synthesis by inserting extra buffers due to limited scope of this work. Usually hold timing is fixed during PnR for more accurate results. The energy numbers presented in this work are probably rather optimistic, but the difference between flip-flop and latch cores is large enough to conclude that there is definitely an advantage for the latch-based version of the core. Applying other energy saving techniques, such as clock gating could increase the energy-efficiency even further.

Compared to flip-flop-based design latch-based design adds design complexity, but at least some of the problems could be mitigated by the tooling, if better support for latch-based sequencing would be added. The cell libraries that were available had more limited selection of latch cells than flip-flops. A cell library geared towards latch-based designs might help with some of the synthesis issues. It remained unclear why the syn-

thesis tool preferred resetless latches with ad hoc reset logic, instead of the latch cell with reset. Based on the library datasheet there was no clear advantage for the resetless latch cells over latch cell with reset. Having a typing mechanism for signals in different phases built-in to the SystemVerilog would allow for a much easier debugging. While the `always_latch` keyword helps with the readability, the contents of `always_latch` block can still synthesize to something other than latches as allowed by the SystemVerilog standard. In my opinion generating an error by default and having a mechanism for waiving the errors as needed would be better for avoiding some of the pitfalls. Moreover, the core conversion tool could be further developed to automate the process of transforming flip-flop cores to latch-based cores in RTL.

Without the RISC-V ISA this work would not have been possible. Inserting new instructions to the core can have very profound impact on the performance of specialized tasks. Adding the popcount instruction decreased the execution time of the CNN by 2.1 times. Combining software and hardware techniques can definitely help to bridge the gap between IoT and high-performance computing, enabling new applications. Because the SCR1 was not synthesized with a dedicated multiplier the CNN code was unrolled at the expense of increased instruction memory footprint. It would be interesting to compare the achieved performance to a core with dedicated multiplier and less loop unrolling. Overall both latch-based design and RISC-V show great promise for highly energy-efficient IoT designs.

References

[1] Jan Rabaey. *Low power design essentials*. Springer Science & Business Media, 2009.

[2] Aaron Stillmaker and Bevan Baas. Scaling equations for the accurate prediction of cmos device performance from 180 nm to 7 nm. *Integration*, 58:74–81, 2017.

[3] Markus Hiienkari, Navneet Gupta, Jukka Teittinen, Jesse Simonsson, Matthew Turnquist, Jonas Eriksson, Risto Anttila, Ohto Myllynen, Hannu Rämäkkö, Sofia Mäkikyrö, et al. A 0.4-0.9 v, 2.87 pj/cycle near-threshold arm cortex-m3 cpu with in-situ monitoring and adaptive-logic scan. In *2020 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS)*, pages 1–3. IEEE, 2020.

[4] Aaron P Hurst and Robert K Brayton. The advantages of latch-based design under process variation. In *International Workshop on Logic & Synthesis*, pages 241–246. Citeseer, 2006.

[5] *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*, June 2019.

[6] Krste Asanović and David A Patterson. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[7] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. In

2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), pages 1–12. IEEE, 2013.

[8] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini. Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8. IEEE, 2017.

[9] Syntacore. SCR1 RISC-V Core. Github, available: https://github.com/syntacore/scr1. Accessed 30.01.2021.

[10] Syntacore. Open-source SDK for SCR1 core. Github, available: https://github.com/syntacore/scr1-sdk. Accessed 30.01.2021.

[11] Xiaokun Yang and Jean H Andrian. A low-cost and high-performance embedded system architecture and an evaluation methodology. In *2014 IEEE Computer Society Annual Symposium on VLSI*, pages 240–243. IEEE, 2014.

[12] Brad Conte. Github, available: https://github.com/B-Con/crypto-algorithms. Accessed 30.01.2021.

[13] Saman Payvar, Mir Khan, Rafael Stahl, Daniel Mueller-Gritschneder, and Jani Boutellier. Neural network-based vehicle image classification for iot devices. In *2019 IEEE International Workshop on Signal Processing Systems (SiPS)*, pages 148–153. IEEE, 2019.

[14] Behrooz Parhami. Efficient hamming weight comparators for binary vectors based on accumulative and up/down parallel counters. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 56(2):167–171, 2009.

[15] David Harris, David Lewis Harris, and David Money Harris. *Skew-tolerant circuit design*. Morgan Kaufmann, 2001.

[16] Ieee standard for systemverilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315, 2018.

[17] Scott Carr and Neil Pittman. Extending gnosis for system verilog hdl static analysis. 2015.

Appendix A

SysVerilogHDL.g4 version 0.2.0 patch

```
--- SysVerilogHDL.g4        2021-02-23 21:42:06.715813397 +0200
+++ SysVerilogHDL_mod.g4        2021-02-23 21:45:14.383131244 +0200
@@ -215,7 +215,15 @@
 Supply0 : 'supply0' ;
 Supply1 : 'supply1' ;
 Task : 'task' ;
+Tick_define : '`define' ;
+Tick_else : '`else' ;
+Tick_elsif : '`elsif' ;
+Tick_endif : '`endif' ;
+Tick_ifdef : '`ifdef' ;
+Tick_ifndef : '`ifndef' ;
+Tick_include : '`include' ;
 Tick_timescale : '`timescale' ;
+Tick_undef : '`undef' ;
 Time : 'time' ;
 Timeprecision : 'timeprecision' ;
 Timeunit : 'timeunit' ;
@@ -258,6 +266,7 @@
 Escaped_identifier : '\\' ~[ \r\t\n]* ;
 Simple_identifier : ALPHA (ALPHA | DIGIT)* ;
 String_literal : '"' (~('"'|'\n'|'\r') | '""')* '"'  ;
+Filename_literal : (~('"'|'\n'|'\r'))+ ;


 // punctuation
@@ -339,6 +348,10 @@
 design_attribute    :   attribute_instance ;

 compiler_directive  :   timescale_compiler_directive
+                    |   include_compiler_directive
+                    |   define_compiler_directive
+                    |   undefine_compiler_directive
+                    |   conditional_compiler_directive
                     |   default_nettype_statement
                     ;

@@ -1343,14 +1356,11 @@
 /**********GENERATES**********/
 /**********CONDITIONAL STATEMENT**********/

-// ADD ELSE_IF
-conditional_statement   : if_statement (else_if_statement)* (else_statement)? ;
+conditional_statement   : if_statement (else_statement)? ;
```

```
-if_statement      : If Open_parenthesis conditional_expression Close_parenthesis
↪  statement_semicolon ;
+if_statement    : If Open_parenthesis conditional_expression Close_parenthesis
↪  statement_semicolon ;

-else_if_statement : Else If Open_parenthesis conditional_expression Close_parenthesis
↪  statement_semicolon ;
-
-else_statement    : Else statement_semicolon ;
+else_statement : Else statement_semicolon ;

 conditional_expression  :   expression ;

@@ -1426,8 +1436,7 @@

 primary_range   :   primary dimension ;

-primary :   type_cast_expression
-        |   number
+primary :   number
         |   concatenation
         |   multiple_concatenation
         |   function_call
@@ -1436,7 +1445,7 @@
         |   imported_function_call
         |   primary_imported_hierarchical_identifier
         |   primary_hierarchical_identifier
-        //|   type_cast_expression //MOVED UP
+        |   type_cast_expression
         |   parenthesis_expression
         ;

@@ -1859,7 +1868,7 @@

 //library_identifier : identifier ;

-//file_path_spec : String_literal ;
+file_path_spec : String_literal ;

 //config_declaration : Config config_identifier semicolon design_statement (
 ↪  config_rule_statement )* Endconfig semicolon? ;

@@ -1891,3 +1900,69 @@
 //instance_identifier : identifier ;

 /**********LIBRARY**********/
+/**********INCLUDE DIRECTIVE**********/
+
+include_compiler_directive : Tick_include '"' Filename_literal '"'
+                           | Tick_include '<' Filename_literal '>'
+                           ;
+
+/**********INCLUDE DIRECTIVE**********/
+/**********DEFINE DIRECTIVE**********/
+
+define_compiler_directive : Tick_define text_macro_name macro_text ;
+
+text_macro_name : text_macro_identifier ( '(' list_of_formal_arguments ')' )? ;
+
+list_of_formal_arguments : formal_argument (',' formal_argument)* ;
+
+formal_argument : Simple_identifier ('=' default_text)? ;
+
+text_macro_identifier : Simple_identifier ;
+
+default_text : String_literal
+             | Binary_number
+             | Decimal_number
```

```
110 +              | Fixed_point_number
111 +              | Hex_number
112 +              | Octal_number
113 +              | Real_exp_form
114 +              | Unbased_unsized_literal
115 +              ;
116 +
117 +macro_text : Simple_identifier
118 +           | Decimal_number
119 +           ;
120 +
121 +/**********DEFINE DIRECTIVE**********/
122 +/**********UNDEFINE DIRECTIVE**********/
123 +
124 +undefine_compiler_directive : Tick_undef text_macro_identifier ;
125 +
126 +/**********UNDEFINE DIRECTIVE**********/
127 +/**********CONDITONAL DIRECTIVES**********/
128 +
129 +conditional_compiler_directive : ifdef_directive
130 +                               | ifndef_directive
131 +                               ;
132 +
133 +ifdef_directive : Tick_ifdef text_macro_identifier ifdef_group_of_lines (Tick_elsif
    ↪  text_macro_identifier elsif_group_of_lines)* (Tick_else else_group_of_lines)?
    ↪  Tick_endif ;
134 +
135 +ifndef_directive : Tick_ifndef text_macro_identifier ifndef_group_of_lines (Tick_elsif
    ↪  text_macro_identifier elsif_group_of_lines)* (Tick_else else_group_of_lines)?
    ↪  Tick_endif ;
136 +
137 +ifdef_group_of_lines : source_text
138 +                     | module_item
139 +                     ;
140 +
141 +ifndef_group_of_lines : source_text
142 +                      | module_item
143 +                      ;
144 +
145 +elsif_group_of_lines : source_text
146 +                     | module_item
147 +                     ;
148 +
149 +else_group_of_lines : source_text
150 +                    | module_item
151 +                    ;
152 +
153 +/**********CONDITONAL DIRECTIVES**********/
```

Appendix B

Flip-flop compile script

```
# Load submodule compilation proc
source $SCRIPTPATH/setup/config.tcl

if { $conf_clear_design == 1 } {
    remove_design -all
}

# Load mw lib for topographical mode
if { [shell_is_in_topographical_mode] } {
    open_mw_lib $mw_lib_name
}

saif_map -start

puts "--- LOAD FILES ---"
source $SCRIPTPATH/setup/load_scr1_pipe_top.tcl
source $SCRIPTPATH/setup/load_scr1_top.tcl

set output_filename [format "%s%s" [format "%s%s" $scr1_target_top "_"] $output_runname]
set default_filename $scr1_target_top

puts "--- PREPARE ---"
current_design $scr1_target_top
link
uniquify

check_design -multiple_designs -cells -ports -designs -nets -tristates >
↪  $REPORTPATH/$output_filename.prechecks.rpt


puts "--- CONSTRAINTS ---"
source $CONSTRAINTPATH/scr1_core_top.sdc

read_saif -auto_map_names -input ./../saifs/dhrystone21.saif -instance
↪  scr1_top_tb_ahb/i_top/i_core_top
saif_map -report

puts "--- COMPILE ---"
set_power_prediction
if { [shell_is_in_topographical_mode] } {
    set_preferred_routing_direction -layers {M1 M3 M5 IA LB} -direction horizontal
    set_preferred_routing_direction -layers {M2 M4 M6 IB} -direction vertical
    compile_ultra
} else {
    compile
```

```
44      compile -incremental -only_hold_time
45  }
46
47  set do_hold_fix 1
48  set hold_fix_rounds 0
49  if {$do_hold_fix} {
50      while {[sizeof_collection [get_timing_paths -delay min -slack_lesser_than 0.0
        ↪  -nworst 1]] > 0 } {
51          source $SCRIPTPATH/utils/fix_hold_buffers.tcl
52          set_size_only [get_cells *eco* -hierarchical]
53          compile_ultra -incremental
54          incr hold_fix_rounds
55      }
56  }
57  echo "Hold fix rounds $hold_fix_rounds"
58  echo "Inserted [sizeof_collection [get_cells *eco* -hierarchical]] buffers"
59
60  change_names -rules verilog -hierarchy
61  saif_map -write_map $output_filename.namemap -type name_map
62  saif_map -write_map $output_filename.ptpxnamemap -type ptpx
63
64
65  puts "--- REPORT ---"
66  if { $conf_write_reports == 1 } {
67      source $SCRIPTPATH/utils/create_reports.tcl
68  }
69
70  puts "--- WRITE OUT ---"
71  if { $conf_write_out == 1 } {
72      source $SCRIPTPATH/utils/write_out.tcl
73  }
```

Appendix C

Flip-flop constraints

```
# Set operating conditions
set_operating_conditions -library $TARGETLIB_CORE:$CORELIB_LVT $COND_WC

# Define clock
create_clock -name $CLKNAME -period $CLKPERIOD -waveform [list 0 [expr $CLKPERIOD/2.0]]
↪  clk
set_ideal_network [list $CLKPORT]
set_fix_hold $CLKNAME

set_false_path -fall_through [get_pins *buf_qlfy*/reset_n*/Q]

set_clock_uncertainty $CLKUNCERT [all_clocks]
set_clock_transition $CLKTRANS [all_clocks]
set_clock_latency $CLKDELAY [all_clocks]

# I/O delays
set_input_delay $INPUTDELAY -max -clock $CLKNAME [all_inputs]
set_input_delay 0 -min -clock $CLKNAME [all_inputs]
remove_input_delay $CLKPORT
set_output_delay $OUTPUTDELAY -clock $CLKNAME [all_outputs]

set_drive -rise [drive_of -rise $IOLIB/$DRIVERCELL/$DRIVEPIN] [all_inputs]
set_drive -fall [drive_of -fall $IOLIB/$DRIVERCELL/$DRIVEPIN] [all_inputs]
remove_driving_cell $CLKPORT
set_drive 0 $CLKPORT
set_load [load_of $IOLIB/$LOADCELL/$LOADPIN] [all_outputs]

set_max_area $MAX_AREA

set_fix_multiple_port_nets -all -buffer_constants [get_designs]
```

Appendix D

Flip-flop hold fix script

```
set hold_broken 1

while {$hold_broken == 1} {
    set worst_path_coll [get_timing_paths -delay min -slack_lesser_than 0.0 -nworst 1]

    if { [sizeof_collection $worst_path_coll] > 0 } {

        foreach_in_collection path $worst_path_coll {
            set slack [get_attribute $path slack]
            set startpoint [get_attribute $path startpoint]
            set endpoint [get_attribute $path endpoint]
            echo [format "%-20s -> %-20s, slack: %s" [get_attribute $startpoint
            ↪  full_name] \
                      [get_attribute $endpoint full_name] $slack]

            echo "    insert buffer $HOLD_FIX_BUFFER_CELL"
            insert_buffer $endpoint $HOLD_FIX_BUFFER_CELL
            update_timing
        }

    } else {
        set hold_broken 0
    }
}
```

Appendix E

Latch compile script

```
# Load submodule compilation proc
source $SCRIPTPATH/setup/config.tcl
source $SCRIPTPATH/utils/p_incremental_compile.tcl


if { $conf_clear_design == 1 } {
    remove_design -all
}

# Load mw lib for topographical mode
if { [shell_is_in_topographical_mode] } {
    open_mw_lib $mw_lib_name
}

set_svf compile.svf
saif_map -start

source $SCRIPTPATH/setup/load_latches.tcl

# Files
set scr1_target_top [list scr1_core_top]
set scr1_target_modules [list scr1_pipe_lsu scr1_pipe_ialu scr1_pipe_exu scr1_pipe_mprf
↪  scr1_pipe_ifu scr1_pipe_csr scr1_pipe_idu scr1_pipe_top scr1_core_top]


set output_filename [format "%s%s" [format "%s%s" $scr1_target_top "_"] $output_runname]
set default_filename $scr1_target_top

write_metrics -outfile reports/progress.log -tag "Analyze/elaborate modules"

# Reset cell package for top reset syncronizer
analyze -format sverilog scr1_reset_cells.sv
elaborate scr1_reset_buf_qlfy_cell
# Elaborate rest of the modules
foreach module $scr1_target_modules {
    analyze -format sverilog $module.sv
    elaborate $module
}

current_design $scr1_target_top
write -hierarchy -output $DBSPATH/$output_filename.elaborated.ddc

link
uniquify

```

```
45  source $CONSTRAINTPATH/scr1_core_top_constraints.tcl
46
47  # Apply size-only to latch cells
48  set latch_module_latches [filter_collection [all_registers -level_sensitive] -regexp
    ↪  {full_name =~ ".*i_latch.*"}]
49  set_size_only $latch_module_latches true
50
51  set_ungroup [filter_collection [get_designs] -regexp {name =~ "scr1_latch_.*"}] true
52  set timing_separate_clock_gating_group  true
53
54  write_metrics -outfile reports/progress.log -tag "Starting compile"
55  if { [shell_is_in_topographical_mode] } {
56      set_power_prediction
57      set_preferred_routing_direction -layers {M1 M3 M5 IA LB} -direction horizontal
58      set_preferred_routing_direction -layers {M2 M4 M6 IB} -direction vertical
59      compile_ultra
60  } else {
61      compile
62  }
63  remove_unconnected_ports [find -hierarchy cell {"*"}]
64
65  write -hierarchy -output $DBSPATH/$output_filename.compiled.ddc
66
67
68  # Reduce clock gating circuitry delay artificially. DC does not detect clock
69  # gate violations for some reason.
70  source $SCRIPTPATH/utils/reduce_cg_delay.tcl
71  # Annotate some addition delays to fix hold
72  source $SCRIPTPATH/utils/fix_hold_buffers.tcl
73
74  set latch_gate_clock_gates [get_cells *and2_clock_gate* -hierarchical]
75  set_clock_gating_check -hold $CLKGATE_HOLD $latch_gate_clock_gates
76
77  compile_ultra -incremental
78
79  source $SCRIPTPATH/utils/create_reports.tcl
80  write_metrics -outfile reports/progress.log -tag "Script done"
81
82  # Make sure svf file closes properly before exiting
83  set_svf -off
84
85  # Write SAIF map
86  change_names -rules verilog -hierarchy
87  saif_map -write_map $output_filename.namemap -type name_map
88  saif_map -write_map $output_filename.ptpxnamemap -type ptpx
89
90  exit
```

Appendix F

Latch constraints

```
proc scr1_core_top_constraints {} {
    global TARGETLIB_CORE
    global CORELIB_LVT
    global COND_WC
    global CLKNAME
    global CLKPERIOD
    global CLKPORT
    global CLKUNCERT
    global CLKTRANS
    global CLKDELAY
    global MAX_TIME_BORROW
    global INPUTDELAY
    global OUTPUTDELAY
    global IOLIB
    global DRIVERCELL
    global DRIVEPIN
    global LOADCELL
    global LOADPIN
    global MAX_AREA

    # Set operating conditions
    set_operating_conditions -library $TARGETLIB_CORE:$CORELIB_LVT $COND_WC

    # Define clock
    create_clock -name $CLKNAME -period $CLKPERIOD -waveform [list 0 [expr
    ↪  $CLKPERIOD/2.0]] $CLKPORT
    set_ideal_network [list $CLKPORT rst_n pwrup_rst_n cpu_rst_n test_rst_n]
    # -no_propagate prevents dont touch from applying to clock gating
    # cells along the clock network. They will be set to size_only instead.
    set_dont_touch_network -no_propagate [list $CLKPORT]
    set_false_path -fall_from [get_ports *rst_n]
    set_false_path -fall_through [get_pins *buf_qlfy*/reset_n*/Q]
    set_fix_hold $CLKNAME

    set_clock_uncertainty $CLKUNCERT [all_clocks]
    set_clock_transition $CLKTRANS [all_clocks]
    set_clock_latency $CLKDELAY [all_clocks]
    set_max_time_borrow $MAX_TIME_BORROW [all_registers -level_sensitive]

    # Allow enable signal delay propagate through gating cell
    set timing_clock_gating_propagate_enable true


    # I/O delays
    set_input_delay $INPUTDELAY -max -clock $CLKNAME [all_inputs]
```

```
45      set_input_delay $INPUTDELAY -min -clock $CLKNAME [all_inputs]
46      remove_input_delay $CLKPORT
47      set_output_delay $OUTPUTDELAY -clock $CLKNAME [all_outputs]
48
49      set_drive -rise [drive_of -rise $IOLIB/$DRIVERCELL/$DRIVEPIN] [all_inputs]
50      set_drive -fall [drive_of -fall $IOLIB/$DRIVERCELL/$DRIVEPIN] [all_inputs]
51      remove_driving_cell $CLKPORT
52      set_drive 0 $CLKPORT
53      set_load [load_of $IOLIB/$LOADCELL/$LOADPIN] [all_outputs]
54
55      set_max_area $MAX_AREA
56
57      # Prevent multiple ouput ports from driving a single net on any level of hierarhcy
58      set_fix_multiple_port_nets -all -buffer_constants [get_designs]
59  }
60  scr1_core_top_constraints
```

Appendix G

Latch hold fix script

```
set rptfile "./reports/eco_buffers.rpt"

set fixable_count 0
set fixed_paths ""
set unfixable_count 0
set unfixable_paths ""
set worst_limit 1

set fix_loop 1
while {$fix_loop} {
    set worst_path_coll [get_timing_paths -delay min -slack_lesser_than 0.0 -nworst $worst_limit]
    set worst_path_count [sizeof_collection $worst_path_coll]
    echo "Got $worst_path_count hold violating paths out of queried $worst_limit"
    set path [index_collection $worst_path_coll [expr $worst_limit - 1]]

    if {$worst_path_count > 0} {

            # Path startpoint
            set startpoint [get_attribute $path startpoint]
            set startpoint_name [get_attribute $startpoint full_name]
            if {[string equal port [get_attribute $startpoint object_class]]} {
                set startpoint_ref [format "port %s" [get_attribute $startpoint direction]]
            } elseif {[string equal pin [get_attribute $startpoint object_class]]} {
                set startpoint_ref [get_attribute [get_cell [string range $startpoint_name 0 [expr [string last / $startpoint_name] - 1]]] ref_name]
            } else {
                set startpoint_ref [get_attribute [get_attribute $startpoint cell] ref_name]
            }

            # Path endpoint
            set endpoint [get_attribute $path endpoint]
            set endpoint_name [get_attribute $endpoint full_name]
            if {[string equal port [get_attribute $endpoint object_class]]} {
                set endpoint_ref [format "port %s" [get_attribute $endpoint direction]]
            } elseif {[string equal pin [get_attribute $endpoint object_class]]} {
                set endpoint_ref [get_attribute [get_cell [string range $endpoint_name 0 [expr [string last / $endpoint_name] - 1]]] ref_name]
            } else {
                set endpoint_ref [get_attribute [get_attribute $endpoint cell] ref_name]
            }

```

```
            # Path setup/hold slack
            set hold_slack [get_attribute $path slack]
            set setup_path [get_timing_paths -from $startpoint -to $endpoint -delay_type max]
            set setup_slack [get_attribute $setup_path slack]

            # Path time-borrowing
            set time_lent [get_attribute $path time_lent_to_startpoint]
            set time_borrowed [get_attribute $path time_borrowed_from_endpoint]

            # Buffer delay is ~0.03 with fanout of 1
            set is_fixable [expr $setup_slack + $hold_slack*1.04 > 0]

            # If timing is fixable insert buffer, otherwise increment nworst count and continue
            if {$is_fixable} {
                # Buffer is inserted just before the load pin
                insert_buffer $endpoint $HOLD_FIX_BUFFER_CELL
                lappend fixable_paths [format \
                                            "\tStartpoint: %s (%s)\n\tEndpoint: %s (%s)\n\thold_slack %-10s setup_slack %-10s\nStartpoint time lent: %s Endpoint time borrowed: %s\n" \
                                            $startpoint_name $startpoint_ref $endpoint_name $endpoint_ref $hold_slack $setup_slack $time_lent $time_borrowed]
                incr fixable_count
                echo "Fixable $fixable_count"

                update_timing
            } else {
                lappend unfixable_paths [format \
                                            "\tStartpoint: %s (%s)\n\tEndpoint: %s (%s)\n\thold_slack %-10s setup_slack %-10s\nStartpoint time lent: %s Endpoint time borrowed: %s\n" \
                                            $startpoint_name $startpoint_ref $endpoint_name $endpoint_ref $hold_slack $setup_slack $time_lent $time_borrowed]
                incr worst_limit
                continue
            }

        } else {
            # No more hold violations
            set fix_loop 0
        }
    }

    # Set eco cells to size_only
    set_size_only [get_cells *eco* -hierarchical]

    set fh [open $rptfile w]
    puts $fh "Fixable paths"
    puts $fh "=============="
    foreach fixable $fixable_paths {
        puts $fh "$fixable"
    }
    puts $fh "\nUnfixable paths"
    puts $fh "==============="
    foreach unfixable $unfixable_paths {
        puts $fh "$unfixable"
    }
    puts $fh "Fixable paths: $fixable_count, Unfixable paths $unfixable_count"

    close $fh

    echo "Fixable paths: $fixable_count, Unfixable paths $unfixable_count"
```

Appendix H

Latch reduce clock gate delay

```
set rptfile "./reports/modified_cg_paths.rpt"

set cg_paths [get_timing_paths -from [get_ports clk] -to [get_pins */GN] -nworst 10000]
set total_cg_paths [sizeof_collection $cg_paths]
echo "Total $total_cg_paths clock gate paths"

# Set to 0 to disable completely
set cg_cell_new_delay 0.005

set cg_path_count 0
set cg_cell_count 0
set modified_paths ""
foreach_in_collection path $cg_paths {
    set last_cell ""
    set last_pin ""

    # Path startpoint
    set startpoint [get_attribute $path startpoint]
    set startpoint_name [get_attribute $startpoint full_name]
    if {[string equal port [get_attribute $startpoint object_class]]} {
        set startpoint_ref [format "port %s" [get_attribute $startpoint direction]]
    } elseif {[string equal pin [get_attribute $startpoint object_class]]} {
        set startpoint_ref [get_attribute [get_cell [string range $startpoint_name 0
        ↪  [expr [string last / $startpoint_name] - 1]]] ref_name]
    } else {
        set startpoint_ref [get_attribute [get_attribute $startpoint cell] ref_name]
    }

    # Path endpoint
    set endpoint [get_attribute $path endpoint]
    set endpoint_name [get_attribute $endpoint full_name]
    if {[string equal port [get_attribute $endpoint object_class]]} {
        set endpoint_ref [format "port %s" [get_attribute $endpoint direction]]
    } elseif {[string equal pin [get_attribute $endpoint object_class]]} {
        set endpoint_ref [get_attribute [get_cell [string range $endpoint_name 0 [expr
        ↪  [string last / $endpoint_name] - 1]]] ref_name]
    } else {
        set endpoint_ref [get_attribute [get_attribute $endpoint cell] ref_name]
    }

    echo "Clock gate path $cg_path_count:\n$startpoint_name ($startpoint_ref)\n  ->
    ↪  $endpoint_name ($endpoint_ref)"
    foreach_in_collection point [get_attribute $path points] {
        set object [get_attribute $point object]
        set object_class [get_attribute $object object_class]
```

```
43          set full_name [get_attribute $object full_name]
44
45          if {$object_class == "pin"} {
46              set pin_cell [get_cell [string range $full_name 0 [expr [string last /
                ↪  $full_name] - 1]]]
47              set pin_cell_name [get_attribute $pin_cell full_name]
48              set pin_cell_ref [get_attribute $pin_cell ref_name]
49
50              if {[string equal [get_attribute $last_cell full_name] [get_attribute
                ↪  $pin_cell full_name]]} {
51                  set inpin_name [get_attribute $last_pin pin_name]
52                  set outpin_name [get_attribute $object pin_name]
53                  lappend modified_paths [format \
54                                              "Path %d Cell %d\n\tStartpoint: %s
                                                ↪  (%s)\n\tEndpoint: %s (%s)\n\t%s (%s):%s
                                                ↪  -> %s delay %f\n" \
55                                             $cg_path_count $cg_cell_count
                                               ↪  $startpoint_name $startpoint_ref \
56                                             $endpoint_name $endpoint_ref $pin_cell_name
                                               ↪  $pin_cell_ref \
57                                             $inpin_name $outpin_name $cg_cell_new_delay]
58
59                  echo "Cell $cg_cell_count: [get_attribute $pin_cell full_name]
                    ↪  ([get_attribute $pin_cell ref_name])"
60                  set_annotated_delay $cg_cell_new_delay -cell -from $last_pin -to $object
61                  incr cg_cell_count
62                  set last_pin $object
63
64              } else {
65                  set last_pin $object
66                  set last_cell $pin_cell
67              }
68
69          }
70
71      }; #points
72      incr cg_path_count
73  }; #cg_paths
74  echo "$cg_path_count/$total_cg_paths clock gating path delays set to $cg_cell_new_delay"
75
76  set fh [open $rptfile w]
77  puts $fh "Modified cg paths"
78  puts $fh "=================="
79  foreach modpath $modified_paths {
80      puts $fh "$modpath"
81  }
82  puts $fh "Modified paths: $cg_path_count, Modified cells $cg_cell_count"
83
84  close $fh
85
86  echo "Modified paths: $cg_path_count, Modified cells $cg_cell_count"
```