Ehgu Proposal

10 RPM ➡ 10000 RPM

CPU   GPU   DSP

uC   TPU   Radio

Sensors   Memories

5G   IoT

Supercomputers
Data Centers

Space travel   Abundant Power

Great jobs :)

$$$$$$$$$$$$$$$$$$$$$$$$$$$$

Terawatts

Si
C H O
Cu Al Au Ag
Sn Fe W Hf P N
In Ga Ge As B Ti Sr

Vikram

Table of Contents

Sad Stories

Mark Troccino is a Director of Design, Hartronics Inc in San Diego, USA. His team is designing the next generation of heart pacemakers with inbuilt machine learning heart attack prediction. He is looking for a skilled digital hardware designer. Unfortunately, the candidates applying for the job either do not meet the skills needed or, the candidates are demanding very high salary that his small startup cannot provide.

Srinivas Gupta is a Design Manager in AppCores Limited, Bangalore, India trying to hire skilled junior digital designers. His team is building the next generation of smartphone application processors. He needs ten designers with good hands on skills. Unfortunately, he can only get smart engineers with masters degree but not with immediately usable job skills. Again the demand for salary is much higher than his budget.

Tony Fong is an Executive Director in CarNerves Inc, Fort Collins, USA. He is the chief of a division building an advanced version of automotive Ethernet targeted for self driving vehicles. The increase in bandwidth capacity is expected to be provided by the next generation of auto Ethernet is touted to drastically reduce the accident rate in self driving vehicles. The occasional accidents in self driving vehicles had by then caused the public to lose confidence in Robocars. This had resulted in increased human fatalities caused by drivers talking and texting while driving. He is hard pressed to find designers with good grasp of automotive safety requirements.

Jim Butler is a truck driver in Tennessee, USA. He has been driving for 20 years. But, recently is worried about his future in the age of self driving trucks. He is also passionate about trucks and the transportation industry. He is keen to upgrade his skills and be a part of companies that design self driving trucks. But his friends tell him that he needs to get an engineering degree in any of electrical, electronics or computer engineering. He has a family of five people and he cannot afford the fees of a full time or even a part time engineering degree.

Viet Nguyen is the son of a fruit seller in Ho Chi Minh City, Vietnam. He is a curious teenager. He is fascinated by smartphones. By the age of 15, he managed to self learn Android apps programming. He has created widely popular apps targeted for Vietnamese population. He wants to do more. He is curious about the black slab like thing inside smartphones. He is interested to make these black slabs. But he is told that the black slab is a silicon chip and he needs to study for a six year engineering program to get a job designing chips.

Cathy Jones is a young teenager in rural Wyoming. She is obsessed with anything electronic. By age 14, she created reverse camera gadget for her father's 15 year old car. Her parents are from a middle class family with only very little money left to spend for her education. Cathy is keen on joining an electrical engineering program in a reputed university in California. But her parents cannot afford the tuition and dorm fees for four years. Undaunted by the lack of funds, Cathy starts working in a nearby fast food restaurant for minimum wage to accumulate enough funds to go to college.

Olav Johnson is a senior principal engineer in Dallas Instruments. His company did a reorganization recently and shutdown his business of creating server processors. His entire team is now assigned to

developing secure Internet of Things products. He is desperately looking for reference designs in elliptic curve cryptography and multiple input multiple output beam forming. But he is unable to find any easily usable material. The expectations from this Senior PE is high, so he is under a huge stress.

Kevin Fukushima is an aspiring new employee in Sandsemi. He is assigned the role of designing a small FSM for analog circuit calibration. He completes that well but is overwhelmed by the sheer number of jargon used in his team – AES, DES, CSI, MPHY, DPHY, CPHY, PCIe, NVMe, IP-XACT, DC, PT, Wreal, HLS, Chisel, RISC-V, FPD-link, DPA attack, DDoS attack. He is not sure what to learn to develop in his career.

Sandy Grove is the CTO of a large chip company. His son is a very hands-on and enthusiastic kid. His son has difficulty sitting in a class room for even 15 minutes. Sandy wishes his son enters the chip industry. Sandy is frustrated that the master's degree based hiring currently practiced in the industry means hands-on classroom hating hyperactive kids can't enter the chip industry workforce even though these hands-on kids would do great on real world projects. He cannot use his CTO position for a referral because that would raise doubts about nepotism.

Madhavan is a very talented circuit design engineer in a leading semiconductor IP company. He has spent well over a decade designing standard cells. He is eager to do RTL design, but, his own company would not give him RTL design opportunity. He is frustrated on being stuck in a job not very interesting to him.

Inspirational Stories

Joseph Wakaba is a 21 year old Kenyan electrician. He is inspired by all kinds of electronics sensors in factories. He checks with his friends about how to get a job related to designing sensors. His friend points him to a online certification exam on hardware design that he needs to pass. Joseph studies for 2 years in weekends and passes this exam. He gets a job offer for Associate Member of Technical Staff IC Design from a Dubai based sensor manufacturing company.

James Izawa is just out of community college with an associate degree in mathematics. James trains himself using digital design education material available at low cost. After completing a basic certification in chip design, James gets a job at Quantech Inc to work as Associate Member of Technical Staff. He completes his first project well and also completes the advanced chip design certification. He is promoted to Member of Technical Staff within one year.

Tom Young is a senior engineer recently assigned to a team designing chips for simulating protein folding. Tom has a PhD in electrical engineering, but, he is puzzled by the requirement to design logic for simulating biological stuff. He refers to an open library called BiochemHDL that has a large number of reference designs for biochemistry related problems. He is able to quickly deploy a working design and get his product to work in first pass of silicon.

Bala Daggula is an executive at Egde Logic Inc. He needs a design IP to preprocess DNA sequencing data coming out of a high throughput sequencer. Bala refers to a subscription based IP library and gets access to the latest algorithm for DNA sequence preprocessing. The best part is the subscription provider continuously updates and supports their IP as long as the subscription is active.

Greg Razavi is a design manager at Saitel looking for talented digital design engineers. He reforms his hiring process by removing the requirement for advanced degrees. Soon, he gets many highly talented applicants from various backgrounds and extremely eager to start work in their new roles.

Ana Gonzalez is a professor in San Carlos university Argentina. She has limited access to actual job related material, she uses the educational material from Ehgu design ecosystem to expand and make her class more practically applicable. Some of her students team up and start a small design IP development startup.

Preface

Electronic chips are now everywhere. There are chips deep in the Sahara desert, there are chips in space, there are chips inside the human body, there are chips deep in the ocean, there are chips inside cars, there are chips in rockets and there are chips in unimaginable places. But chip designers are only a tiny fraction of the world population and are located only in a few selected pockets of the world. The headcount of chip designers is so small, it is a miracle they are able to make an out-sized impact on the world. Every piece of modern machine contains chips in one form or another. Our life in the information age is underpinned by chips. Even at the current level of technology the value added by chips is immense. So immense that most of the world population depends on it for healthy, happy and prosperous living.

Suppose, I tell you the number of chip designers can be increased by 100x and the number of electronics systems designed by 1000x, what wonders would come to pass? Is it even possible? Oh, Yes! We can.

Sadly, when older professions are fading away into history newer professions like chip design has a significant shortage of skilled people. Chip design is accessible to only a few people in the world. The design process is also far less efficient than what it can be. In this book, I describe my ideas related to a chip design ecosystem that is more efficient at producing useful chips and also reward more people across the whole world. It is quite cumbersome to refer to hardware design engineers or ASIC design engineers or digital designers or RTL designers or IC designers or FPGA designers. It is also rather useless to add the term engineer to every job title. So, I am naming this particular trade as chipping and the practitioner as a chipper.

Ehgu Proposal

Low cost education

Create low cost education material that helps aspiring people to self learn and start contributing to the world of chipping.

Open Certification

Create a standardized certification process for chipping that is open to all. Revise the certification curriculum periodically, say, every three years.

Massive Value Creation

Consider candidates certified through this process for jobs without asking for extra qualifications like degrees. Job specific extra skill requirements may be specified. Let people from varied backgrounds from across the world contribute to the chip industry by providing open source reference design libraries, advanced software as a service and open process technology support files.

Enjoy the massive value created by a flood of new designs!

Who is this book for?

Complete Beginners, Makers, Hobbyist and Continuous Learners

This book will provide you a quick overview of chip design and also point you to a lot more details. I have reduced the prerequisites to just basic English, basic mathematics, basic computer literacy like using Internet, using Linux or Windows and nothing more. This book will quickly add the digital design skill to your toolbox of maker skills. This opens a whole new level of control over the hardware of your projects!

Welcome to the world of infusing smarts into sand! I bet you will start loving it :)

Students, Interns and Entry Level Chippers

Apart from regular textbooks, I intend to provide additional industry perspective and more readily usable examples to help you understand and undertake more complex projects. Use this to fill in the gaps that your mentor may not have mentioned. Examples and quizzes will make your learning fun-"ner" and also get your dream job, pay raise and promotion!

Experienced Professionals

Think of this book as a large browser bookmark file for chipping. You may not be aware of some links mentioned in this book that may help you. Over time, I plan to add many sophisticated designs to act as specific topic reference. For now, you may like clock generator with jitter, ADC, CDR, PLL and HLS examples. The code in the book is free and open source. Feel free to reuse the code in your designs! (**after full verification**)

You can use this book to prepare for general interview questions. Or if you are the one interviewing, then use the exercise questions, examples and quiz from this book in your recruitment process.

You may already have a vague sense of the untold concepts behind chipping that will crystallize after reading the section on philosophy of design and verification.

Switching

This book will fast track your switch from other related fields like circuit design/physical design/standard cell design to digital design. It will also provide you a good vocabulary to talk to digital designers in your company.

Mentors, Educators

Use this book to add variety and new topics to your course or even as a full curriculum guide book. Plenty of references are provided to low cost alternatives of the high cost resources you may be using now.

Researchers

Do not have enough funds to hire an expert chipper? Do not have access to a chipper even for consultation purpose? No problem! This book will help you with examples so that you can focus on complex topics by building on the open source code provided with the book.

What this book is not?

This book does not cover in detail the topics of circuit design, SoC Integration, verification, FPGA prototyping, synthesis and timing analysis.

Note to open sourced version 31

Version 31 has been open sourced and made free to let a lot more people think about greatly expanding the chip ecosystem. This book is in a way also an elaboration of the outline provided in the Ehgu proposal white paper aimed at creating a more efficient chipping ecosystem.

This version is a great overview and catalog of chipping topics lying around in bits and pieces in the online and offline world. It is a decent intro to modeling of clock generation, ADC, PGA, PLL, fractional-N PLL and CDR. Examples on pipelining, retiming, parallel processing, multicycle operation and serial operation provide a fun way to learn the tricks of digital design. Examples in procedural coding using recursion, dynamic programming and linking objects will help software oriented learners. I think I can claim the simplest example for High Level Synthesis and Direct Programming Interface. There is also a hello world in UVM that will help you crush the overbearing nature of UVM with a tiny first step. You may find the library of elementary modules and functions in the examples useful to make more complex designs. There are about 90 fully syntax highlighted SystemVerilog code examples and about 100 interesting quiz questions! You may also like the insights discussed in the unit philosophy of design and verification.

The latest version of the proposal is here -

Ehgu Proposal: Towards an efficient chip design ecosystem

https://www.amazon.com/dp/B07TZGPKC8

GitHub Open Source Code

Most of the code, scripts and other utilities used in this book are available as free and open source code in GitHub page below. I grant you license to use the source code in any legal way you want. There may still be patents or copyrights that you may have to license from the holders, if any.
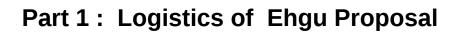
https://github.com/3vm/dsn_verif
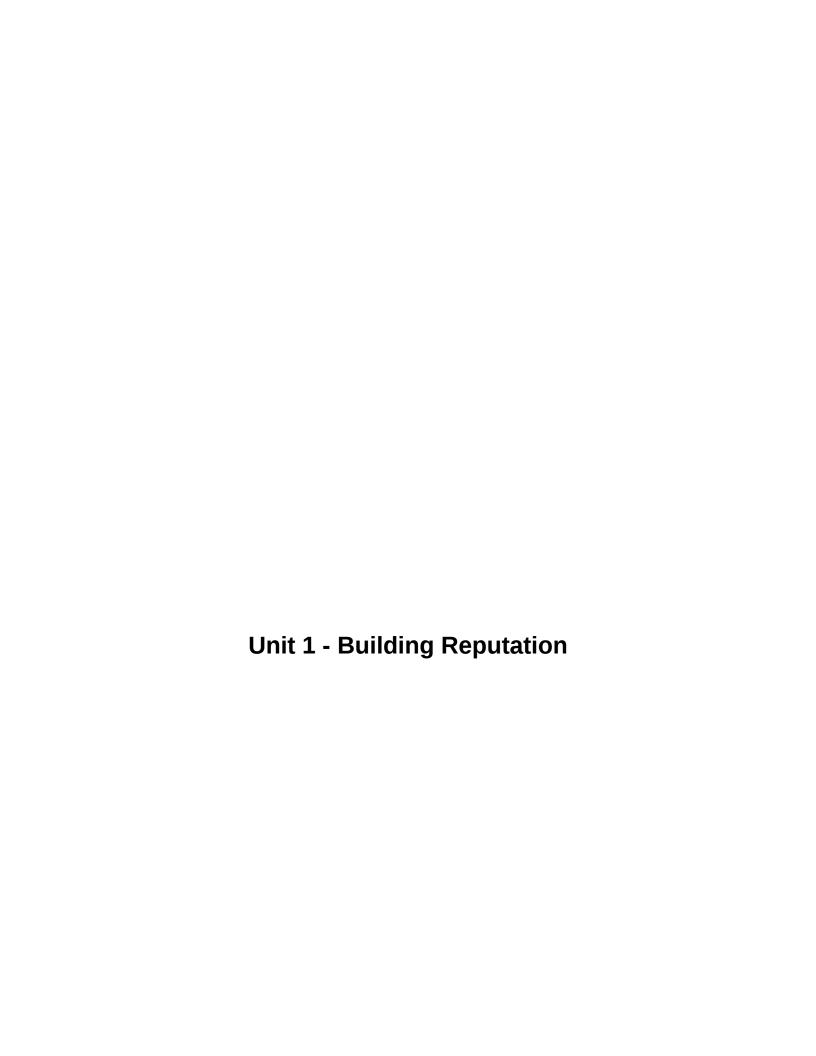
Commit ID: 3ae37356ccb04d47d4bf472b8988a0e5716cfdc8

2021 Nov 21

Enjoy!

Vikram

Part 1 : Logistics of Ehgu Proposal

Unit 1 - Building Reputation

I named this unit certification at first. But, certification is only one way to build reputation. The deeper problem is turning skills into trust that is suitable for value creation.

Certification

It takes quite an effort to create a curriculum acceptable to the semiconductor industry. It also needs online and offline presence to reach all corners of the world.

Shifting needs is another problem. By the time a student completes the certification, the job requirements should remain same as the one covered by the certification. If the certification curriculum becomes obsolete by then, it would become useless for the industry. So, the time needed to prepare and complete the certification has to be short, so that, the semiconductor industry still needs those certified skills.

Authentication is another problem. A novice could get the help of his friend or a paid accomplice to pass the certification. If the novice got hired for an actual job he would not be able to perform well. One way to reduce fraud is by completely recording the certification process through the channels of video, audio, screen capture and keystroke capture. If fraud is suspected, these records can be examined later. The mere recording of the examination, in many cases, can stop would-be fraudsters.

Who is the right authority to issue this certificate? This authority now becomes the new gate keeper to great jobs. Gradually, the certification process can get broader, costlier and longer. These forces need to be kept under check regularly. And how is the certification process going to be revised? Creating two independent and competing certification bodies can help in maintaining healthy change. For example, there could be one consortium headed by a chip company and another headed by a rival chip company. The sacred IEEE that has released numerous industry standards and is held at high esteem by chippers is a very acceptable body. The newly emerging CHIPS alliance is also a good candidate.

https://chipsalliance.org/

Should the examination process be open book or open Internet or closed? The job context is going to be open Internet, so, it is preferable to keep the examination open to Internet. But creating a test for open Internet use is much harder than seizing the applicants' smartphone and blocking Internet access in the examination room for a fully closed test.

There is always the online degree programs. For instance, there is a fully online MS Electrical Engineering degree program from University of Colorado, Boulder, USA

https://www.coursera.org/degrees/msee-boulder

You may wonder if there are fully online MS EE programs why do we need certification? The problem is that it is still only EE and not chipping. As of late 2020, I could not find any online degree programs that are dedicated to chipping. Also, the price tag of $20,000 for the online MS EE is not affordable to many. I am proposing a certification exam that costs in the $100 range. The way $100 is achieved is that all the studying cost is left to the candidate and no direct effort is made to teach. Rather just the syllabus for the certification exam is made public and the candidates are allowed the freedom to choose

any study material – videos, blogs, designing on FPGAs, making ASICs, textbooks, simulation only and even degree programs and one on one tuition.

Exercise
Can you find fully online BS EE degree programs?
Are there BS/MS chip design degree programs, online or offline?

Technical Curriculum

Chipping has evolved into two natural specializations, one for ASIC design and another for FPGA. There is a lot of overlap between the two, but, there is also a lot of specific learning. So, I am recommending three curricula - Chipper Basics, Chipper ASIC and Chipper FPGA. The basics certification is to get anyone a simple job in the ecosystem quickly. Other certifications are for career growth.

Chipper Basics

Circuits: Kirchoff's laws, Ohm's law, power consumption in electrical circuits.

CMOS : NAND, NOR, NOT, AND, OR, XOR, MUX, Tristate inverter transistor circuits, power dissipation in CMOS – switching, short circuit and leakage power

Basic algorithms and data structures in C++: linear and binary search, bubble sort, merge sort, big-O notation, linked lists

Digital logic theory, SystemVerilog 2013 language, concepts from logic synthesis, concepts from static timing analysis, version control – Git or Subversion

RTL simulation with Vivado/Modelsim/Incisive/Xcellium/VCS, Waveform debugging with Vivado/Modelsim/Simvision/Verdi/DVE, Gate Level Simulation – handling synchronizers, initializing unintialized flip flops and combinational loops

Design and verification

FSM, counter/timer, data and reset synchronizers, FIFO with CDC modeling, arbiter, SPI, I2C, control registers, low power design, pipelining, retiming, scheduling

Verification concepts - coverage, randomization, regression

Chipper ASIC

Concepts from DFT, Standard cells, ROM and RAM compilation and usage in designs, anyone SerDes data packetization and depacketization logic, performing design ECOs, basics of RISC-V ISA and one RV32I processor core, UVM basics

Behavioral models for PLL, ADC – Flash and SAR ADC, DAC – String and ladder DAC

Chipper FPGA

FPGA resources – LUT, DSP, BRAMS, clocking. IP integration, basics of RISC-V ISA and one RV32I processor core, anyone SerDes integration – MIPI/HDMI/DDR, interfacing with UART, AXI, APB, AHB, Tilelink, Synthesis and physical design for FPGA – pin assignment

Advanced Specializations

Chipper Algorithms

Sorting – radix sort, quick sort, trees, graphs, depth first search, breadth first search. Creating C++ models for any 10 algorithms targeted for chips

Chipper Mixed Signal

Circuit theory - superposition theorem, Thevenin and Norton's theorem, maximum power transfer theorem, small signal model of analog circuits

Semiconductor device physics, P-N junction diode, Zener diode, BJT, MOS capacitor, MOSFET, JFET, LED, photodiode and solar cell, CMOS process technology

Continuous and discrete-time signals and systems theory: causality, stability, impulse response, convolution, poles and zeros, power and energy of signals.

Transforms: Laplace transform, discrete-time Fourier transform (DTFT), DFT, FFT, Z-transform, interpolation of discrete-time signals; digital filter design techniques.

Verilog AMS, SV wreal modeling

Chipper SoC/Integration

Study of one RISC-V RV64 SoC, knowledge of older HDL standards like VHDL and Verilog 2001 for integrating legacy design IP. Bus interfaces - AXI, APB, AHB, Tilelink, Memory interface – DDR4

Chipper Networking

Concept of OSI layering, TCP/IP protocol stack, switches, routers and routing algorithms, TCP/UDP sockets, congestion control, application layer protocols (DNS, SMTP, POP, FTP, HTTP), basics of Wi-Fi, network security: authentication, basics of public key and private key cryptography, firewalls, software tools – wireshark

Chipper SerDes

DC balance, line coding – 8b10b, manchester, NRZ, equalization, CDR, SSC, PLL, BER, Forward Error Correction, ARQ, multiprotocol tunneling, channels – PCB trace, Coax, Shielded Twisted Pair, Optical Fiber

Chipper Cryptography

Number theory, linear algebra, DES, AES, Eliptic Key Cryptography, Direct Sequence Spread Spectrum, Frequency Hop SS, RSA, Contemporary threats – DoS, DDoS, zero day exploits, bots, reverse engineering

Chipper Processors

Hardware-software co-design, RISC-V ISA and cores – 64 bit high performance and 32 bit small cores, compcuter architecture – ISA, cache, memory management, interrupts, compiler, loader, linker, Linux operating system

Chipper ML

CNN, deep learning, statistics for ML applications, basics of rule based AI, image recognition, anyone ML frameworks - Tensorflow or equivalent

Chipper Neo

A chipper with all certifications!

Ideas for Certification Exam

Multiple choice questions are preferable to other type of questions because they are easy to evaluate. To begin with, the questions can be fixed and the choices numbers can also be fixed. This may later be improved to a constrained random question and both the values in the question and the answer order are randomized. For example, rather than ask a fixed question like what is the current passing through a resistor of value 10 ohms when the voltage across it is 10V and then provide choices with fixed order as a. 1A, b. 2A, c.10A, d. 0.1A. The exam software can be input a question class object that it would constrained-randomize per exam candidate. It could be like-

```
class ohms_law_q;
rand float R, V;
rand float answers[4];
rand int correct_choice;
constraint {1<R<10};
constraint {1<V<10};
constraint {0.1<answers[i]<100};
void set_choices {
  answers[correct_choice] = V/R;
}
endclass
```

Ensuring fairness across tests

If there are a large number of questions to choose from and if every candidate gets a unique test how should one ensure that every candidate is compared fairly. How to suppress the effect of one candidate getting a tough set of questions and another candidate getting an easier set of questions?The graduate

record exam GRE, already does something about the fairness question. The ideas in these exam delivery may be reused for chipper certifications.

Measuring non technical skill

It is super hard to quantify the soft skills of a person. The chipper certification process probably should not attempt to even measure it.

Other Ways For Reputation Building

Certification is the easiest and most recognizable method to create reputation. There are other ways that have been in existence for a long time.

Peer Review

The reference based recruitment for jobs is based on a kind of review by a trusted person. It is hard to get a trusted person so, we can use the more accessible online reviews approach as a means to quantify reputation. How this would work is that an aspiring candidate would start sharing chipping related information, say, designs on protein folding or verification IP for machine learning designs by posting to GitHub or answering questions in online forums. As the candidates contribution keeps building up the users of these content would rate the information creating candidate. The user ratings can be feed into the recruitment funnel. But, this approach is still prone to impersonation.

Hackathon

Hackathons have become an interesting method to explore new software applications. Borrowing this idea for chipping, large companies can organize hackathons for job applicants and offer them real world problems in chipping for the hackathon event. Individuals and teams demonstrating exceptional talent can be hired as full time employees or interns. The main value I see in hackathon is in removing doubts about the ability of an unknown job applicant. In a hackathon, her ability becomes seen as is, without any hype or suppression. Note that hackathon is resistant to impersonation because you can actually see the person do the chipping.

https://en.wikipedia.org/wiki/Hackathon

Group Project

It is an irony of life that full time employees would accept an extra 24 hours in a day to meet project deadlines while students waste their time preparing for exams. Would it not be better for students to show their skill in group projects on real world problems in chipping? If a group of students set out to make a library for hardware acceleration logic for molecular biology simulations, their skill or the lack of their skill will be readily seen in the library they create. If group projects are publicly rated like how online shopping users rate products, there can be great visibility and fast selection option for many students.

Multi-university Project

The world of chipping has evolved into a mega enterprise. There is also a need for mega teams producing path breaking mega products. While the software world boasts of hundreds of compilers and tens of programming languages, the hardware world has only very few HDLs and HDL simulators. Other EDA tools for synthesis, place and route, Lint, LEC, CDC are even scarcer. Would it not be better to join the forces of multiple universities into one unstoppable army to create EDA tools and well categorized design and verification IP? For example, 30 students from Arizona State University can collaborate with 20 students from University of Colorado, Boulder for an open source project on a C++ based software for a Lint checker for SV hosted on GitHub. The industry could pick off the best contributors for direct employment in their companies.

Demo

Often, a company needs specific talent. It could be deep knowledge in motor control, or it could be deep knowledge in high frequency PLLs. A demo is a great tool to recruit top talent immediately available to be put to work in companies. Suppose, an individual or a group demonstrates an all digital PLL with an FPGA, the entire team could be hired as junior engineers in a company designing digitally controlled high performance PLLs. Note that demos are not as impersonation proof as hackathon because a well educated person could pass someone's work as his own work. This limitation can be reduced by following up the selection process with more questions.

Crowd Interview

There is a niche for a company dedicated to automating the hiring process. Chippers working in companies and self employed ones can be invited to interview candidates on selected topics like digital logic basics, low power design, PLL, RISC-V etc.. Candidates interested in job change can give these interviews. The interview process can be recorded and made available to recruiters and hiring managers. The recordings of the interviews can be set to auto expire or deleted under the control of interviewer and interviewee to maintain privacy. It is also needed to ensure old information is not used for recruiting. Another set of chippers may score the interviewees for quality of their answers. Interviewers may be simultaneously scored for their creativity in probing the knowledge of the interviewee. The interviewers, interviewees and the scorers come from the crowd, that is, a large population of general chippers. This approach increases the efficiency of the employee hiring process of chip companies that presently spend quite a lot of time and money.

Crowd Sourced Questions

If 1000 practicing chippers contribute just 10 questions each, there would be 10000 questions that may be useful for creating the certification exam. The candidates who participate in the certification exam may be asked to volunteer a further 10 minutes to submit new questions. Questions that get answered correctly by more than 90% of the test givers may be removed as being too easy. Those questions could still be retained for training purpose. When this process goes on for a few years, we would be left with

about a 100k to 1 million high quality questions. To ensure quality of the questions, a moderation process is needed. That can be outsourced to the crowd too. During the chipper certification exam, candidates who point out errors in questions may be awarded higher scores. The moderation process can also be done as a full time job at the certification agency by about 10 experts. If each process 3000 questions a month, 100k high quality questions may be reached in less than a year. Questions can also be sourced from university exams, job interview questions and competitive exams with permission to copy. These questions can serve as a great quality control step for general public learning chipping by themselves.

MOOC degree programs

MOOC providers are doing a great job of easing the access to advanced education. Chipper degrees can be provided via these providers too. Unfortunately, as of this writing, I am not aware of a good MOOC degree program that lets a student get a job in the chip industry.

Udacity Nanodegree

MITx Micromasters

Coursera Degrees

Reuse Competitive Exams

In India, GATE exam conducted to screen applicants for masters programs can be reused as a certification tool to assess the subject knowledge in electronics engineering. There are no chip design specific exams I know of. So, additional chip certifications may still be necessary. The use of the GATE test score would replace the need for electrical or electronics degree. This is because many chip industry managers may not be comfortable with the idea of hiring total non electrical engineers for chip design. Using competitive exam scores of electrical or electronics or instrumentation engineering would provide them some confidence in the ability of new hires. Using another entrance exam score for MBA programs, the CAT would also cover the general IQ testing needed for getting a job in the chip industry.

Cost Estimate For Learning And Certification

This section provides an estimate of the time and money to complete the basic chipper certification. General living expenses are not included. You certainly do not need any special accommodation like a dorm. Your home is just fine.

Time

I have somewhat arbitrarily allocated 300 hours for preparing for the chipper basics certification. Spending an hour a day completes 300 hours in less than a year. Or spending 10 hours a day completes 300 hours in just a month. Three hundred hours is also quite sufficient to get trained in the basics of chipping and be in a position to start contributing to the chipping ecosystem.

Knowledge

Blogs, web pages, videos – free.

Books - $200. Can be reduced by using a library or sharing with friends.

Software

Free!

Power

You will need some amount of power to run your laptop, phone and tablet to learn chipping. Assuming a time of 300 hours to learn the basics curriculum and about 50W of power usage. The cost of 15kWh is not even worth mentioning.

Computing Platform

A laptop or a desktop is a must for learning chipping. A used laptop for a few hundred dollars is quite sufficient. Add $200.
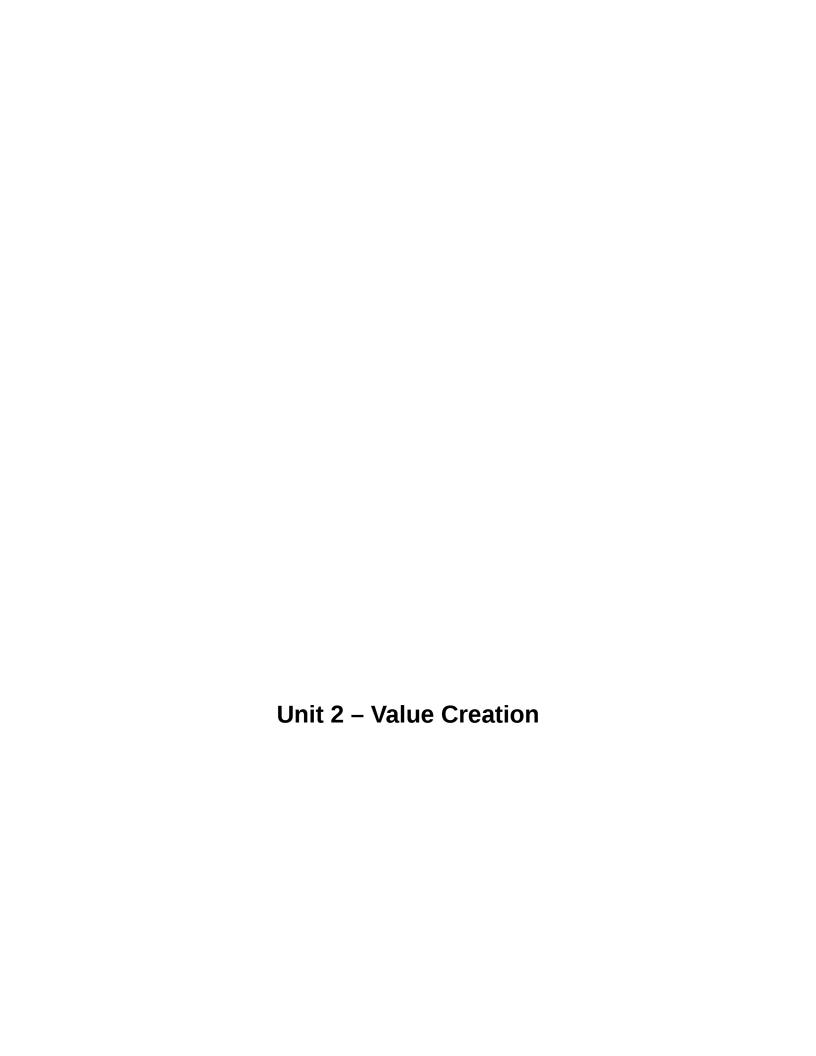
Certification Fee

Exam fee may be the highest expense. Existing exams like GRE/SAT go from about $50 to a few hundred dollars. I think certification cost can be reduced to $100 or less.

Grand total is $500 of money and 300 hours of time.

Gamification

Chipping can get boring at times. If the learning is made more fun by creating an animation filled game then more of the general population can be reached. So far, I don't know how to create a game for educational purpose. Neither have I come across a freely available game.

Unit 2 – Value Creation

Skills to Value

I first named this unit jobs. The name jobs does not fully justify other ways of creating value. Then I named it monetizing your skills. That did not justify the free and money agnostic ways of value addition. Finally, I settled for the name creating value because creating something useful is the main point of learning skills. After having spent a long time studying chipping, you should be able to convert those skills into some valuable commodity or service.

Jobs

The easiest way is to join companies that create chips. In real estate the mantra location, location, location is famous. For certification, we could say jobs, jobs, jobs! What else is the reason for the existence of certifications. Let us start by examining some job postings from different companies.

Retrieved from Intel.com on July 26, 2019

Sample Job 1 – Intel SoC Design Engineer
Job ID: JR0113631
Job Category: Engineering
Primary Location: Penang, PNG MY
Other Locations:
Job Type: College Grad

SoC design engineer

Job Description

Oversees definition, design, verification, and documentation for SoC (System on a Chip) development. Determines architecture design, logic design, and system simulation. Defines module interfaces/formats for simulation. Performs Logic design for integration of cell libraries, functional units and subsystems into SoC full chip designs, Register Transfer Level coding, and simulation for SoCs. Contributes to the development of multidimensional designs involving the layout of complex integrated circuits. Performs all aspects of the SoC design flow from highlevel design to synthesis, place and route, timing and power to create a design database that is ready for manufacturing. Analyzes equipment to establish operation infrastructure, conducts experimental tests, and evaluates results. May also review vendor capability to support development.

Qualifications

Bachelor or master in electrical and electronic engineering

Inside this Business Group

The Data Center Group (DCG) is at the heart of Intel's transformation from a PC company to a company that runs the cloud and billions of smart, connected computing devices. The data center is

the underpinning for every data-driven service, from artificial intelligence to 5G to high-performance computing, and DCG delivers the products and technologies—spanning software, processors, storage, I/O, and networking solutions—that fuel cloud, communications, enterprise, and government data centers around the world.

What can we understand from the job description?

Well, the job needs quite a lot of technical knowledge and unfortunately asks for a degree in electrical or electronics. Strangely, not even computer science degree holders are mentioned. The first few sentences mean the same thing - "digital design using a HDL". If you a have a lot of practice in designing digital logic circuits a lot of the job prerequisites is already satisfied. The words SoC and integration means there is more emphasis on creating larger designs by instantiating existing IPs than creating new IPs from scratch. Integration has some standards to ease the job. One or more processor cores are connected to many peripherals or accelerators via a bus. In SoC terminology, a bus means a protocol that helps a processor communicate with its connected devices. ARM based devices use AXI, AHB, APB type buses. Low speed devices use SPI, I2C and UART buses.

How do you practice SoC integration?

Start by imagining a chip architecture that solves some real world problem. Say you want to create a face ID chip. Search the web for core or IP that can be used as a building block for the full chip, say, an image processor core and a high speed PHY. Can you create large digital systems from other peoples' IP? In essence, that is what is the job really. "All aspects of SoC design flow", means you are knowledgeable in logic design, logic synthesis and timing analysis. The place and route part is mostly not expected because there are entire teams for that. But you may still be expected to know the basics of that. The note on infrastructure, test and equipment means you will have to work in the laboratory with the chips occasionally. So, you will need some level of understanding of electrical circuits. It indirectly also means, you are proficient at collecting and analyzing data in spreadsheets.

The last part on applications like ML, AI, 5G, HPC are interesting. Knowing some aspects of the end application of where your chip is being used helps but is not expected. It may help if you are already in one of the companies involved in the same industry.

Sample Job 2 – ADI Digital Design Engineer
Retrieved by Google Search on 2019, Dec, 07
Digital Design Engineer
Analog Devices
Bengaluru, Karnataka
Full–time

Analog Devices (NASDAQ: ADI) designs and manufactures semiconductor products and solutions. We enable our customers to interpret the world around us by intelligently bridging the physical and digital worlds with unmatched technologies that sense, measure and connect.

Position: Digital Design Engineer
Location: Bangalore
Group : High speed signal processing group, (Wireless Communications BU)
Hiring Manager : Sreejith K

Job Responsibilities
• Design high performance digital blocks for Complex Communication ICs in leading edge process nodes
• Develop optimal micro-architecture based on power performance area tradeoffs
• Analyze signal processing architecture tradeoffs using Matlab modelling
• Coding using Verilog and System Verilog
• Participate in module architecture and specification
• Block level designer verification
• Front end implementation tasks
• Lint/CDC check, Synthesis, Timing constraint development
• Work with DV team, review test-plans for DV signoff
• Work with DFT P&R teams for DFT and timing signoff Position Requirements (for Fresh College Graduates)
• Minimum BE/BS/Mtech/M.E/PhD degree in Electrical/Electronics/Computer science from a reputed institute
• Excellent academic performance with strong technical fundamentals
• Good verbal and written communication skills to work effectively with teams spread geographically Position Requirements (for Experienced Engineers)
• Minimum BE/BS/Mtech/M.E/PhD degree in Electrical/Electronics/Computer science from a reputed institute with excellent academic performance
• 1-4 years of relevant industry experience
• Digital logic design and hands-on RTL coding experience
• Experience with developing timing constraints and running state-of-the-art Synthesis and Static timing analysis tools
• Good Knowledge in Processor/SoC architecture, DSP fundamentals
• Good verbal and written communication skills to work effectively with teams spread geographically
PS : Experienced candidates who are in semiconductor or electronic field but not working on digital design and having outstanding academic record with excellent professional achievements are also welcome to apply

For positions requiring access to technical data, Analog Devices, Inc. may have to obtain export licensing approval from the U.S. Department of Commerce - Bureau of Industry and Security and/or the U.S. Department of State - Directorate of Defense Trade Controls. As such, applicants for this position – except US Citizens, US Permanent Residents, and protected individuals as defined by 8 U.S.C. 1324b(a)(3) – may have to go through an export licensing review process.

Analog Devices, Inc. is an Equal Opportunity Employer Minorities/Females/Vet/Disability Education Level: Bachelor's Degree Travel Required: Yes, 10% of the Time

Annotation

The ADI chipper job is interesting. Apart from regular skills of digital design with HDL, ADI needs expertise in digital signal processing. ADI is a market leader in specialized signal processing chips. As of this writing, bulk of signal processing is happening digitally. So, you need to be knowledgeable in mathematical concepts like z-transform, Laplace transform, Fourier transform, analog circuits (especially ADC/DAC), DFT, FFT, designs like Numerically Controlled Oscillators, Modulators/Demodulators, PLL, CDR. Unlike the SoC integration job, the ADI job does not need you to be an expert in integration using industry standard buses like AXI. You are, however, required to use Matlab software to simulate the DSP algorithms in a computer to help you design better performing chips. You are also expected be aware of multiple options in designing a chip to meet specific power performance area (PPA) trade off.

Sample Job 3 – Northwest Logic, Logical Designer
Retrieved from nwlogic.com on 2020 January 13

Logic Designer Job Description

Northwest Logic is looking for a full-time Logic Designer to join the Northwest Logic team.

Responsibilities:

•Design architecting and trade-off analysis

•RTL coding and verification

•Controller + PHY integration and verification

•FPGA targeting

•Customer delivery and support

Required Skills:

•Strong Verilog RTL design and verification expertise

•Questa/Incisive/VCS simulator experience

•Python/Perl/Tcl scripting experience

•Significant ASIC and/or FPGA design experience

•Ability to learn quickly and work independently

•Solid communication and project management skills

•5+ years of logic design experience

Definite Plus:

•ASIC synthesis, timing constraint, CDC/RDC experience

•Memory (HBM/DDR/LPDDR), PCI Express,or MIPI expertise

•Located in the Portland, Oregon area

Training:

•Northwest Logic will provide training as needed

For more information on Northwest Logic, please contact us at info@nwlogic.com.

Annotation

This job from Northwest Logic Inc. is the only one I have seen not asking for degrees! May be they know that chippers without degrees are highly improbable. NWL is a design house famous for design IPs. If you work in this job, you will create designs that will be used by customers outside your company. Your customers will take your RTL design and make a bigger system from it. For example, a company could buy a CSI-2 IP from NWL for a bigger image sensor chip. Contrast this job with the job in ADI. You may not be talking to any outside customer directly in the ADI job. All the interactions are likely to be with chippers within ADI. In NWL, your customers will be chippers in other companies. Let us understand the job description line by line. The job responsibilities start with architecture and trade-offs. This means that you will have the freedom to choose how a specification is structured for RTL implementation. You are also required to analyze how your chosen architecture compares with many other architectures you may have considered in terms of PPA. You are going to do both RTL coding for the architecture and also verify the RTL yourself. The controller and PHY integration means you also have the additional responsibility of connecting a PHY IP with its controller IP. It is possible that both PHY IP and controller IP are done by others. FPGA targeting means the skills needed to take an RTL design aimed at ASIC implementation and modifying it for FPGA implementation. This needs some understanding of the facilities offered by FPGAs like clock management logic, block RAMs etc. Customer delivery and support is a general skill. Your experience in any customer facing job like retail store manager or an employee in a fast food chain or as a call center employee may qualify. Note that a pure chipper job in the past as in the ADI engineer job may not qualify for customer support experience because bigger companies let application engineers rather than designers talk to customers. If you were in a company that delivers electronic products or another IP design house, it may be even better. For example, suppose you were a delivery manager for electric motors supplying automotive companies that is a very good experience from customer support point of view.

The skills section says that you need to be good at Verilog based RTL design and verification that should be self explanatory by now. There is a special mention on scripting with PERL/Python/TCL. These are mainstay scripting languages of the chip industry. Not knowing at least one of these languages will put you at a severe disadvantage when working on real projects. NWL is ok with either ASIC or FPGA experience for this job and not too keen on the user having deep expertise in only one. NWL also wants you to learn on the job and be independent. Smaller organizations expect more

independent execution because there is not enough managers to supervise your job. Communication and management skills are also specified. If you do not already have these, start talking, listening, writing and reading more. For management, you may want to read about and practice project management skills. It is likely that you are required to be a people manager.

Additional skills can give you an edge for this job. NWL specifically is interested in timing constraints, clock domain crossing and reset domain crossing. These skills are very useful to SerDes IPs that form the backbone of NWL product portfolio. You also hold a better chance of getting the job if you are already in the Portland, Oregon, USA area. This means NWL is not interested in bringing people from say Vietnam or even the east coast of USA. NWL would also give you points for having experience in protocols like PCI, HBM, DDR, MIPI. Of course, I would hire someone who has already done work related to my company!

Exercise

Study 3 job postings for ASIC Design Engineer or IC Design Engineer or Design Engineer. Note down what skills may be needed to get those jobs. Use linkedin.com or glassdoor.com or indeed.com or monster.com or any suitable job postings website or physical notice board.

Study the experience profile of 3 existing chippers using social network sites like linkedin.com. Make a note of how they have arrived at the jobs that they do today.

Self employment

With the skills for chipping you now have options to try your own business.

Gsuite

Google's suite of cloud applications enable you to start an online business quickly and at low cost. Using the online business email, you could get access to organizations that need specific company email and not general public email like Gmail or Yahoo mail.

Design house

The traditional way is to start a design IP company. You would create an implementation for a standard. For example, you could design MIPI Soundwire IP for mobile applications and then sell it to larger mobile chip companies. Some of these designs are routinely licensed for many $10000s to $100000s. You could also contract design work from larger companies. Contracting usually is a one time activity. IP development and sales is more profitable.

Blogger

If you can capture the interest of the design community with frequent posts about useful content, then blogging may provide a decent return. There are quite a few blogs for education about chipping and quite few about news.

Developer for other platforms

AWS EC2 F1 – This is a cloud hardware acceleration platform for Amazon cloud applications. You can develop hardware and license it on a per hour basis. Applications sell from $0.1 to $20 per hour of use. If you get a thousand users using the application 24x7 for a year, that could add up to significant revenue.

SiFive

SiFive has positioned itself as a chip development house that accelerates design and testing of chips at low cost. For SiFive's business model to work, it needs a lot of design IP for their design environment. You could pitch in and offer you designs to the third party design buyers via SiFive. I am not exactly sure how this would actually work though.

Xilinx Hardware Developer

Xilinx has a huge FPGA presence and part of its success is because of a large number of design IPs that are plug and play in the Xilinx FPGA environment. You can add more IPs to the Xilinx portfolio and be rewarded for it.

Similar to Xilinx, there could be options to commercially develop for Altera (Intel) or Microchip FPGAs.

Freelance

You could complete specific small projects for users who need the designs. Websites like https://www.upwork.com/ and https://www.freelancer.com/ help you with finding decent paying projects.
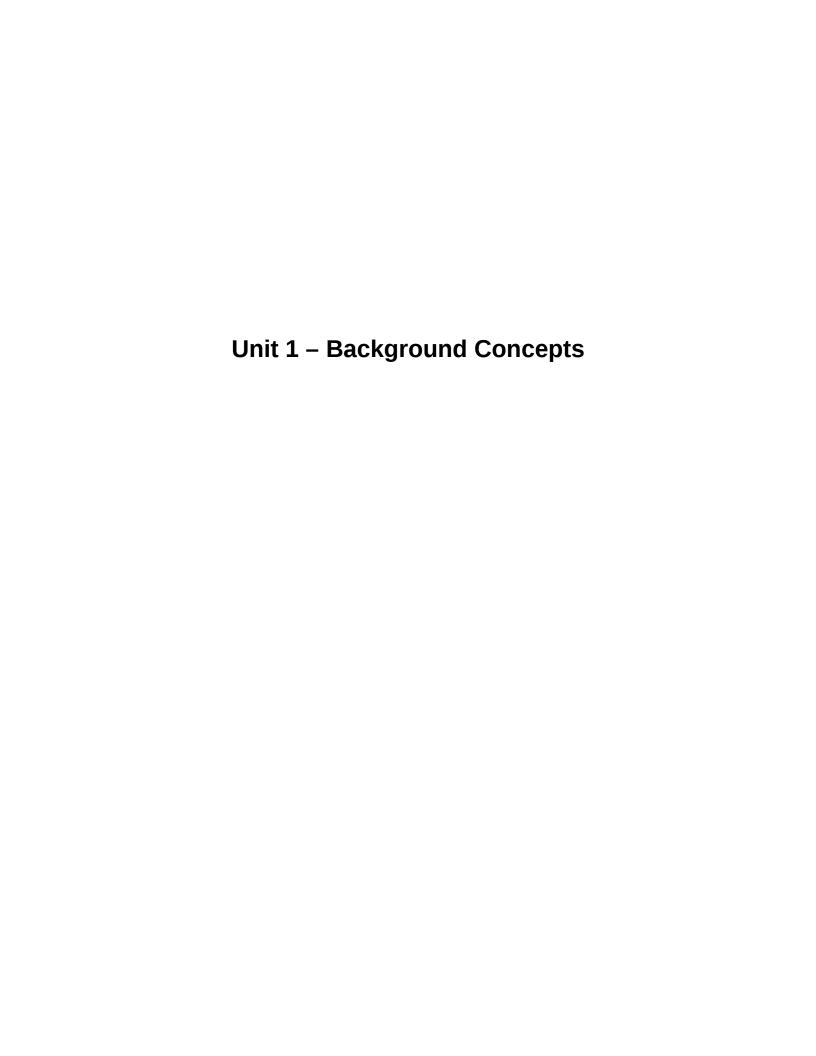
Education

You could create educational courses in Udemy and get paid whenever students sign up for that course. You could also create textbooks, videos, audiobooks, podcasts, blogs, iOS and Android apps, Windows apps for educating about chipping.

Research

Research is a background activity that drastically changes the course of an industry. It is rarely directly visible in the news. To do research in chipping, you have two options – join a university and work with a professor for specific problems or join the research and development arm of large companies. Universities engage in really fundamental work while companies engage in work that could be profitable in the next 0-5 year time frame and industry pays more than universities.

Part 2 : Education

Unit 1 – Background Concepts

Introduction

To create value by chipping you need to have some understanding of the fundamental principles involved. But there is no end to learning the fundamentals. I suggest read-a-bit play-a-bit approach. At anytime, if you are tired of studying the basics feel free to jump to examples or quiz section.

About chips

Suppose you happen to like chipping, you may end up living with it for a long time. Let us take the time to familiarize ourselves with chips.

How chips are made?

Before learning more about chipping, it is useful to know how regular silicon chips are manufactured. Note that silicon is just one material that can be used to produce electronic chips. There are many other materials like Silicon Carbide, Aluminum Nitride, Indium Phosphide, Germanium are regularly used. What all these materials have in common is that they are called semiconductors which means they are not as good of an electrical conductor as copper or not as bad a conductor like glass.

https://mybroadband.co.za/news/hardware/200748-how-a-computer-chip-is-created-from-sand-to-cpu.html

Packaging a bare die

The silicon piece that contains one full chip is called a die. A die by itself cannot tolerate the mechanical stress of the real world, so, it is put into a package. The package used for a chip affects cost and performance of the chip. Note that you can contribute a lot even without knowing a thing about how chips are made and packaged.

https://en.wikipedia.org/wiki/Integrated_circuit_packaging

https://en.wikipedia.org/wiki/List_of_integrated_circuit_packaging_types

IC Bonding

https://en.wikipedia.org/wiki/Flip_chip

https://en.wikipedia.org/wiki/Wire_bonding

Where are chips used?

Chips have a wide range of applications. Just to keep the enormous applications into manageable pieces, the semiconductor industry uses terms to define particular market segments. Note that these terms are shifting and have great overlap with other segments. Terms like automotive, server, High Performance Computing, mobile, desktop, memories, processors, microcontrollers, communication, IoT, consumer electronics, defense, aerospace, sensors, imaging and biomedical are used frequently. Splitting the application domains into widely understood groupings helps in organizing, recruitment,

training, standardization and market analysis and so have a value the industry. Sector and vertical are other names used interchangeably with segment.

Exercise

Download the datasheets of 3 different chips and note down what strikes you as interesting points. Go easy on trying to understand all points about these chips. No chipper, however experienced, can understand all the information in a datasheet.

My choice of 3 chips are here. Note that these links may be removed by the companies that own these products when the products are discontinued. Any other 3 chips would do just fine.

Intel processor

https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/6th-gen-x-series-datasheet-vol-1.pdf

Texas Instruments ADC

http://www.ti.com/lit/ds/symlink/ads125h02.pdf

Marvell Ethernet Switch

https://www.marvell.com/documents/2k6q1h2vmoeczqd5bq4v/

How large can chips be?

Chips come in a large number of sizes. The smallest are so tiny that they may not be visible to the naked eye. The largest ones can be about the size of a small plate. Typically, array type chips are large. Those include image sensors and processor arrays. The following chip from the company Cerebras is one of the largest chips.

https://www.eetimes.com/document.asp?doc_id=1335043

Chip size is measured in millimeter squared or may be indirectly measured by the number of transistors it contains.
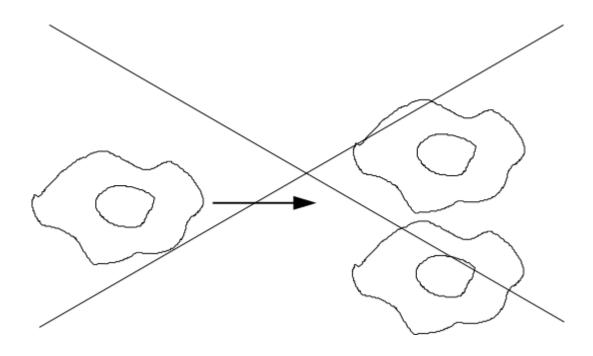
What chips are not?

Movement

To my knowledge, chips cannot move well on the large scale. Unlike a car or an animal that can move for many thousands of meters, most chips cannot move. But some chips can have moving parts in the microscopic scales. Acoustic sensors can translate movement to electrical signal. A Digital Light Processing (DLP) chip can move tiny mirrors to project an image.

Replication

Chips cannot reproduce themselves like how living beings do. Chips need humans for design, manufacturing and distribution. Thank goodness chips don't reproduce!

Growth

Trees, plants, animals, micro-organisms can all grow. But chips are really set in semiconductor stone. Once fabricated, chips do not get bigger or smaller. As an implication, chips cannot repair themselves with new material. But, many chips are designed to repair themselves with spare logic already made during the fabrication step.

How chips die?

We have seen how chips are manufactured and how chips are used. Do we know how chips die? Actually, I don't know much about this either. I am listing a few ways that I have come to know because of working in the industry

Obsolescence

Chips actually don't die easily. We kill most of them! I have seen electronic products last 10-20 years. But then, in only 2-5 years there is a replacement gadget that is cheaper and better. Once, I had the privilege of using an antique equipment called Sony DG2020. It generates signals of specific patterns that are programmed by the user. For example, you could create a signal that is high for 5ns, low for 15ns, again high for 30ns, and so on. Having gotten used to swiping the smartphone, I was like a fish out of water trying to program the DG2020 using buttons. There was a screen which shows the programmed waveform. Out of habit, I was swiping that many times knowing full well that it is not a touch screen. I think it is not even an LCD screen. It could very well have been a Cathode-Ray Tube (CRT) monitor. I saw the maintenance log on the DG2020 and realized that it was 15 to 20 years old. Working fine still! There are many examples like this, a 10 year old laptop, 14 year old USB drive and 13, 20 year old calculators. The space probe Voyager has been working for more than 40 years!

https://en.wikichip.org/wiki/microprocessors_used_in_spacecrafts

ESD

Can you recollect shocking your friends on cold dry days, using nothing but the static charge accumulated on your body? That kind of shock can kill a chip to death by damaging the delicate transistors inside.

EM

Imagine a hurricane pounding the atoms inside the wires of a chip. In the place of the wind in a hurricane, the pounding is done by the electrons flowing inside the wire. The relentless bombardment of the atoms physically damages the wire structure which is called Electro-Migration (EM).

Overheating

In some modes of operation of a chip, the power dissipation can exceed the safe level. In this case, the chip can get too hot and sustain damage to internal transistors and other devices.

Hot Carrier Degradation

A MOSFET transistor in some ways resembles the struggle between an angry mob trying to influence people in power standing behind a police barricade. Most of the time the mob succeeds in conveying their message without ever crossing the barricade. However, if the mob gets angry enough, they break the barricade and damage the political system. They could permanently change the government or damage the police force. In the MOSFET, electrons or holes pile up on one side of the insulating gate oxide and influence the current on the other side of the gate oxide. But if the electric field across the gate is high enough then some electrons could breach the gate oxide and cause damage to the transistor physical structure permanently. Note that this analogy is just for fun. It may be wildly misleading.

https://en.wikipedia.org/wiki/Hot-carrier_injection

Metal Whisker

Though not directly a chip failure, a short circuit in the board because a wire grew all by itself and shorted another wire is one cause of electronic system failure, taking down the chips in the board as a side effect.

https://en.wikipedia.org/wiki/Whisker_(metallurgy)

Numbers

Chipping starts with numbers. What most chips do inside is manipulate binary numbers. Study the concepts about binary numbers and binary arithmetic – addition, subtraction, multiplication and division, two's complement notation for signed numbers.

https://en.wikipedia.org/wiki/Binary_number

Note that binary numbers exists even without computers or digital logic. What digital logic provides is a fast way to do operations on binary numbers.

Hexadecimal numbers are a short form of binary numbers. It is easier to write in hex than binary. https://en.wikipedia.org/wiki/Hexadecimal

Optional topics

I was about to reject Octal number system as bloatware, but then, it is used in a very popular 8b10b coding. So, it may be worth making a cursory reading.

https://en.wikipedia.org/wiki/Octal

https://en.wikipedia.org/wiki/ASCII

https://en.wikipedia.org/wiki/Binary-coded_decimal

https://en.wikipedia.org/wiki/Floating-point_arithmetic

https://en.wikipedia.org/wiki/Fixed-point_arithmetic

You may see that the idea of time split into 60 seconds, 60 minutes, 24 hours, 12 months, 365 days is like a number system. The idea of splitting angle into 360 degrees is also like a number system. Number system has a close relationship with units used for measurement.

Chipping frequently uses scaling prefixes like nano, micro, mega, giga, pico etc that are worth understanding right in the beginning.

https://en.wikipedia.org/wiki/SI_prefix

Note that the tech community is adopting different prefixes for binary vs powers of 10 that are observed in memory size, data size and network speed.

https://en.wikipedia.org/wiki/Binary_prefixes

Looking at the numerous formats for representing numbers, it is clear that a number is just a symbol for something. The most intuitive that something is a magnitude of a physical thing. Various formats provide different advantages in ease of use vs efficient implementation in hardware.

Intro To Digital Logic

Number representations are in a way purely mathematical. Chips need to do something with them to be of any use. Digital logic theory supplies the tools to take advantage of the mathematics.

https://en.wikipedia.org/wiki/Boolean_algebra

https://en.wikipedia.org/wiki/Truth_table

For a two input function, there are 16 possible unique functions that may be defined. The table below lists all the 16 functions with the name given to them. Note that the functions zero and one are trivial.

Inputs		Output (function name)															
A	B	Zero	NOR	!(A->B)	! A	!(B->A)	! B	XOR	NAND	AND	XNOR	B	B->A	A	A->B	OR	One
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Table referred from lecture slides of the Edx MOOC course, "Principles of Synthetic Biology", Adam Arkin and Ron Weiss

https://en.wikipedia.org/wiki/Combinational_logic

https://en.wikipedia.org/wiki/Sequential_logic

https://en.wikipedia.org/wiki/Logic_gate

https://en.wikipedia.org/wiki/Metastability_(electronics)
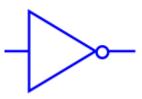
https://en.wikipedia.org/wiki/Circuit_diagram

https://en.wikipedia.org/wiki/Netlist

CMOS logic gates

https://en.wikipedia.org/wiki/CMOS

Read about the schematics for CMOS inverter, NAND gate, NOR gate, AND gate and an example layout of inverter and NAND gate

Book reference

This is a newly released book

Designing Digital Systems With SystemVerilog (v2.0), Brent E. Nelson (Author)

Another great but older book

CMOS VLSI Design, A Circuits and Systems Perspective, Fourth Edition, Neil H. E. Weste, David Money Harris

There are many useful open books for chipping in All About Circuits.

https://www.allaboutcircuits.com/textbook/digital/

Algorithms, Data Structures and Programming

Algorithms

Chipping job is almost like software development but with a different framework. We call the SV RTL as code and what does that code contain? Algorithms! You will appreciate the time spent learning algorithms when you have to actually create complex designs.

Topics of interest

Arrays, queues, stacks, binary tree and other trees, graphs, space and time complexity, big O notation, search and sort algorithms, recursive algorithms, algorithm design methods – greedy, divide and conquer, dynamic programming, backtracking, branch and bound.

This BBC documentary provides a fun way to get familiar with algorithms - The secret rules of modern living: Algorithms

https://www.bbc.co.uk/programmes/p030s6b3

Programming

Algorithms by themselves cannot be executed by a computer. It cannot be directly synthesized to hardware either. Algorithms need a computer understandable language to be of any use. There are a large number of programming languages. For chipping, it is preferable to learn C++ as the fundamental programming language. This is because C++ lends itself to writing good high level software tools and at the same time model hardware at bit level. It will become more common to describe hardware directly in C++. There should be numerous resources to learn C++. The Linux g++ is a free C++ compiler, so there is no shortage of compilers.

Topics of interest

Regular language constructs and libraries like STL.

http://www.cplusplus.com/doc/tutorial

The following book "concisely" covers a lot of background in this field.
Data Structures, Algorithms and Applications in C++ by Sartaj Sahni

Computer Architecture

Only so many tasks can be flexibly carried out in hardware. For additional flexibility, most modern systems employ a central processing unit (CPU) or just processor for short. It becomes necessary to understand how a processor based system works. Traditionally, the field of computer architecture focused on processor based systems that executed some kind of software. Recently, the idea of one processor controlling everything and being the master and sole focus of a system is coming under question. Newer processing chips are being released with multiple regular processor cores, few

Graphics Processing Unit (GPU) cores, a Field Programmable Gate Array (FPGA), in some cases a DSP and a host of peripherals. Anyway, to have a basic idea of conventional computer architecture, it is useful to study the single processor system first. It is useful to see how software interacts with hardware. Computer architecture also provides useful insights into maximizing system processing throughput while reducing costs.

https://en.wikipedia.org/wiki/Computer_architecture

RISC-V is an upcoming processor architecture. So, read this book that teaches computer architecture with RISC-V in perspective.

Computer Organization and Design, The Hardware/Software Interface: RISC-V Edition

Compiler and Assembler

A processor needs dedicated software to enable user programs to run on it. For example, the user program could be a computer game. The game developer will not be an expert in the processor architecture. He will not know the machine instructions to program the target processor. Instead he will use a high level language like C++ to create the game program. Some other software developer who knows about the target processor will create a compiler that converts high level language programs into machine binary instructions for the processor. For example, you could have C++ compiler for AMD x86 processor. Like the compiler, an assembler converts an assembly language code to machine binary instructions. Note that assembly language code is only marginally easier to use than machine binary code. The compiler is much more preferable than the assembler. Assembly code is used in places where the compiler generated code is not good enough or when you don't have a compiler for the target processor. The compiler and assembler come under the topic of system software. There are some more system software – linker, loader, bootloader and debugger.

Operating System

As processor systems became more complex, the compiler was not enough to help the software developer create applications. Even with the compiler, the developer was still tied to the details of the specific processor and computer system that was the target of the code. The developer had to customize his user level software, such as the computer game to suit the target computer. An OS or operating system helps in this trouble. It abstracts the details of the computer system into an organized interface for the user software to interact with. For example, the OS contains software called device drivers for printers, keyboard, mouse, WiFi adapter and more. The device drivers' job is to present a consistent set of software functions that the user application developer can use to access the devices. Say, you have a printer from HP connected to your laptop. Now the operating system would use a HP printer device driver and offer a function something like OS_Print(<file>) to the user application. In your game code, you may include a Print option in the File menu. When a game player selects the File → Print option, the OS_Print() function will be executed and a file will be printed. The advantage can be seen when you disconnect the HP printer and connect another printer from another manufacturer, say, Epson.
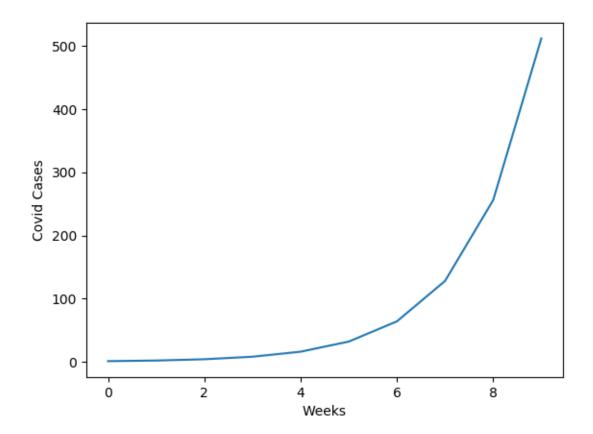
Without an OS you may have to replace the previous version of game software that worked with HP printer with one that works with the Epson printer. Imagine doing this work for all the software installed on your laptop, nobody would buy laptops if that was the case! When an OS is used, the switch to the Epson printer will trigger the operating system to automatically downloaded the relevant Epson printer devices drivers from the Internet or you would manually install it from a CD/DVD or a USB memory. Only the OS_Print() function will be different. All other application level programs like the game software, spreadsheet program, calculator, etc will remain unchanged.

Signals

When something changes with time it may be called a signal. The idea of a signal is that we focus on the time variation of the thing than the thing itself. The height of the sea as measured at a cliff wall may show a signal with characteristics similar to the pattern of a signal recorded from a guitar. Signals are represented in mainly 2 ways.

Waveform

A plot of the value of the signal in y axis and time on the x axis is a waveform. Engineers mostly understand waveform better than other formats that describe a signal. The number of Covid 19 cases when plotted every week may show a waveform as in the figure.

Function

Variation of amplitude with time can be captured into a mathematical formula. This is called a function. This format may be more intuitive to mathematically oriented people. It also lends itself to further mathematical analysis and manipulation while a waveform is useless in mathematical equations.

Example of function

f(t) = 1 if t>=0

f(t) = 0 if t<0

Example of signals

There are infinite possibilities of how something can change with time. In nature, some patterns occur frequently and so people have given names for them.

Pulse
The pulse most people will be familiar with is the pressure pulse felt on the body created by the human heart. The pulse in blood feels like no change for most of the time and then suddenly there is a sense of something pushing the fingers from inside the body and then nothing again. In digital logic, a similar meaning of pulse is used. The quantity under observation, be it a voltage or current or something else stays low and then rises to high and stays high for sometime before finally returning to low. Note that a pulse by default means a positive pulse, that is, low – high – low behavior. There is also a negative pulse that goes high – low – high. Chipping using digital logic mostly deals with pulses. Let's take sometime to study the parameters of a pulse.

Rise time
If you look closely at any real world pulse including the figure shown you can see that the low to high change happens not in zero time but takes some non zero time. In other words, the low to high change is not a pure vertical line but a slightly sloping line. The time the signal takes to go from low to high is the rise time. Typical rise times inside chips are in the range of pico seconds to nano seconds. Suppose you enter a highway from a city street through a ramp and you take 5 seconds to rise from the street to the highway. We could say that the rise time of your car is 5 seconds.

Slew Rate

The rate at which the signal value changes is called slew rate. Suppose, the rising edge goes from 0.4V to 0.6V in 50ps you could say the slew rate is (.6-0.4)/50p = 4 giga volt / sec. Since this feels somewhat weird you can just say 4 V/ns.

Fall time
Fall time is the opposite of rise time, the time taken to fall from high to low. Note that the rise and fall time for a pulse need not be same.

Pulse Width
The time duration for which the pulse stays high is called the pulse width.
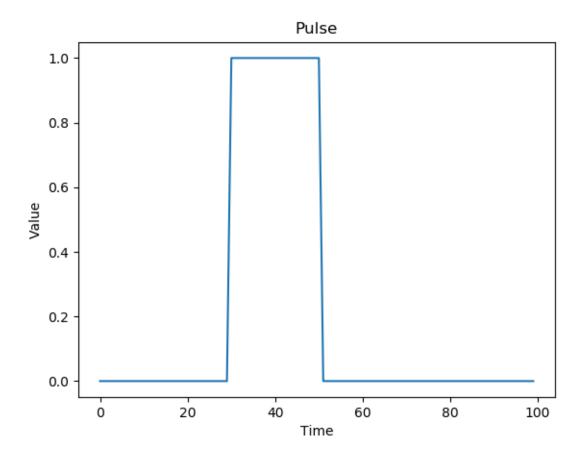
Amplitude

The difference between the low and the high magnitude is called the amplitude or more precisely peak to peak amplitude. Common pulse amplitudes are 1V, 5V, 10V (-5V low to +5V high), 12V.

Energy
in terms of signals, the energy of a pulse is product of squared amplitude and time. This closely resembles the energy of a current or voltage pulse except that there is a scaling by a resistor. The notion of pulse energy is used in interference of one pulse on another signal and propagation of unwanted digital pulses through logic gates.

Square/Rectangular wave

When something has a on-off-on-...-off type behavior with on and off values constant then it is called a square wave. A rectangular wave is the more technically correct term but square wave is often used interchangeably with rectangular wave. A rectangular or square wave may be thought of as repeating train of pulses. Square wave is the most important signal in chipping. It has a special name called clock because it defines time steps that orchestrate the logic operations inside a chip. Let's understand the parameters of the clock signal.

Period

The time between two rising edges (or falling edges) is one period. Period is a property of not just square wave but of all repeating (periodic) signals.

Frequency
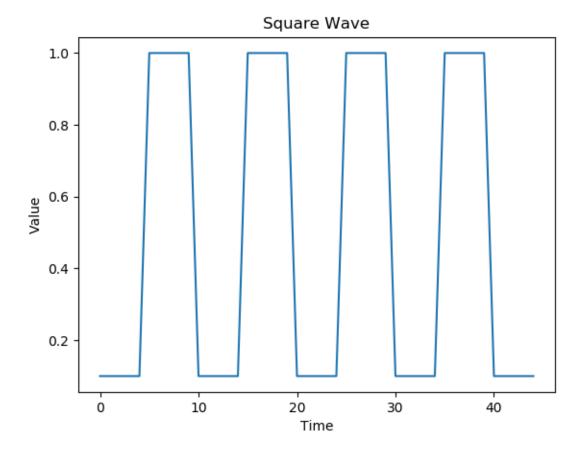
The inverse of period is frequency. So, 1ns period is same as 1/1ns or 1 GHz frequency. The first principles definition of frequency is number of repetitions per unit time. The basic unit of time is second. So, number of repetitions per second is frequency. The name for the unit "repetitions per second" is Hertz.

Duty Cycle

The on duration to period ratio is duty cycle or duty ratio. It is also represented as a percent. Suppose the on time is 0.45ns and period is 1ns the duty cycle is 45%. A 50% duty cycle rectangular wave is a square wave which is basically saying on and off times are equal.
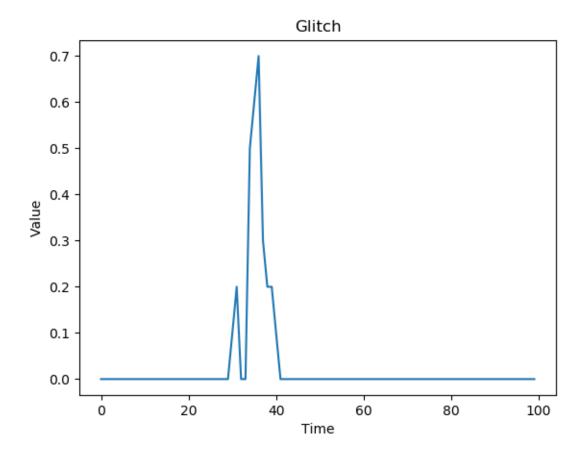
Jitter

In a perfectly period square wave the rise times happen exactly at multiples of the period. So, if rising edge happened at 0ns for a 1ns period square wave, the subsequent rising edges happen at 2ns, 3ns, 4ns and so on. But no edge happens at 1.1ns or any 2.3ns and so on. Real life square waves, however, have an uncertainty in the time at which the edges occur. This deviation from the perfect edge positions is called jitter. For example, if for the same square wave rise edges occur at 1.001ns, 1.995ns, 3.010ns, and so on the maximum deviation is 0.010ns for the 3ns edge on the positive side. On the negative side, 2ns − 1.995ns = 0.005ns. Overall the peak to peak deviation or jitter is 0.010+0.005ns = 0.015ns or 15ps. There are further classification within jitter called random jitter and periodic jitter which is left as an exercise.

Square Wave

Glitch

A glitch is not a specific signal as such. In the context of digital logic, a glitch could be said to be an unwanted change in a signal. Suppose there is a clock of 1MHz toggling between logic 1 and logic 0 every 0.5us, a glitch would be say a transient pulse modifying the clock. For example, when the clock was supposed to stay logic 1 from 0 to 0.5us, if the logic changed to zero momentarily from 0.25us to 0.28us it is a glitch. Another interpretation of a glitch is a pulse that is mangled. A regular pulse goes to full voltage level of 1 and 0. A glitch may rise part way and then fall. This kind of improper pulses can corrupt logic state inside a chip. There is also another usage of glitch in the tech world to mean a malfunction.

Glitch

Sinusoidal wave

If you ask someone to draw a wave there is a high chance that they would draw a sine wave. The alternating current propagating along transmission cables is a sine wave. A sine wave has a nice smooth changing characteristic to it. Sine wave lies at the heart of many signal analysis ideas. If you are going to stick to pure digital bits processing logic a sine wave is not that important. But if you happen to interact with analog circuits, knowledge of sine wave and its properties become important.

Amplitude

The minimum to maximum is called the peak to peak amplitude. Sometimes just the maximum on the positive (negative) side alone is also called amplitude. For the sine wave in the figure, you could say the peak to peak is 2 units and the non peak to peak or just amplitude is 1 unit.
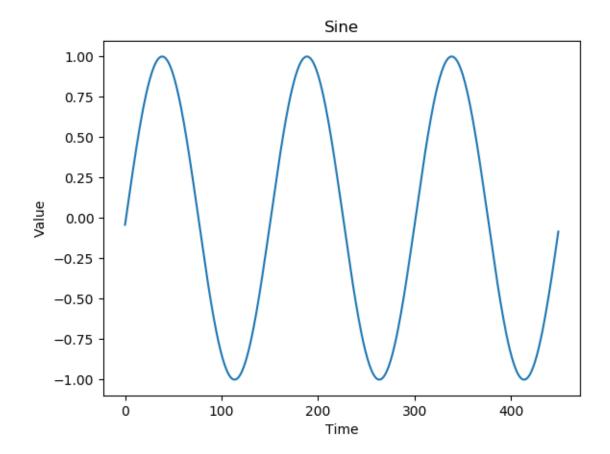
Root Mean Square

RMS is a commonly applied term to periodic signals and more commonly to sine wave. The name says it all, first you square the values at every time point and then take a mean or average and then do a square root. Intuitively, RMS can be thought of as a way to quantify a signals deviation from zero. The

more the deviation the more powerful the signal is. Note the resemblance to the the standard deviation formula.
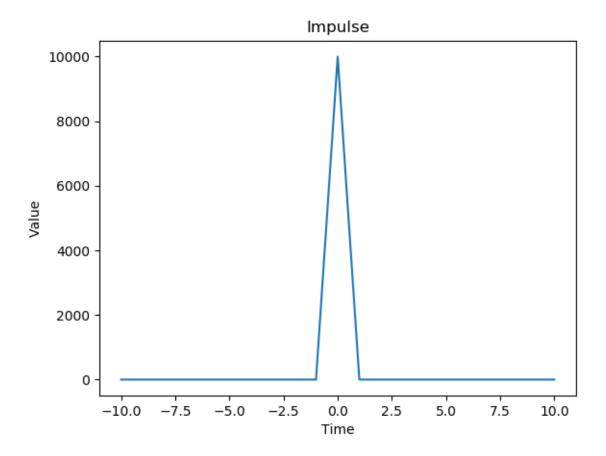
Phase

The phase of a periodic signal is the time position of the wave value relative to one period. Suppose you have a sine wave with a period of 1ns and that the wave rises from the value 0 at 0ns. You will have the positive peak at 0.25ns, again 0 at 0.5ns, negative peak at 0.75ns and back to zero at 1ns. Phase is measured in the units of angle – radian or degree. One period is divided into the angle in a circle which is 360 degree. At 25% of the period, you would say 25% of 360 = 90 degree phase. Note that because phase is relative to a period, a 90 degree phase is same as 360+90 degree phase and phase is mostly used within the 0 to 360 degree (or radian 0 to 2 pi) range. Phase is also more commonly used to relate two sine waves. For example, if you have two 1ns period sine waves, with one starting at 0 ns and another starting at 0.25ns you would say they are 25% or one fourth of a period or 90 degree or pi/2 radian apart in phase.

Sine
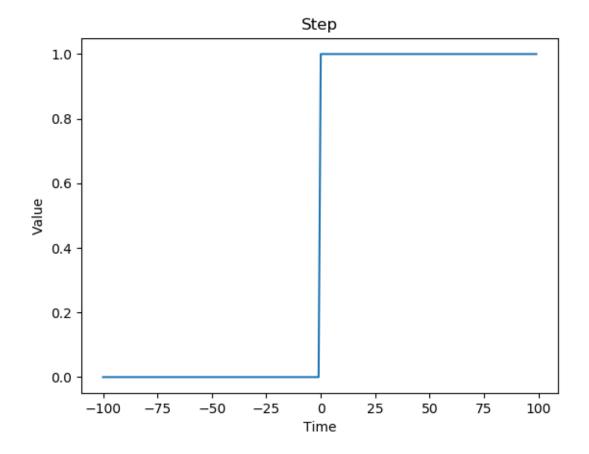
Impulse

A change in value that happens in a flash is termed impulse. The true impulse function is an imaginary entity. It is supposed to rise to an infinite value and fall back to zero in no time. Real impulse like phenomenon comes close. A lighting can be considered an impulse when we compare it to the duration of a rainstorm. The idea is if the event in question is much faster than some other reference event we can use impulse function to analyze it. Impulse is also known as delta function or Dirac delta function. I wish someone changes the the weird sounding "impulse" to a simpler name "bolt".
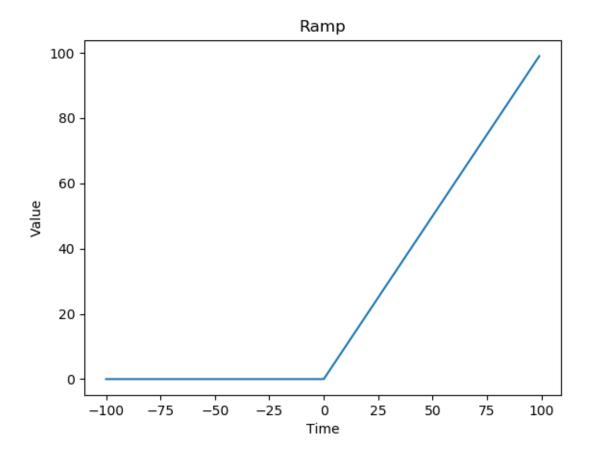
Step

A step function is very intuitive. At last the engineers named one signal resembling commonly visible thing. As the name says a step wave looks like a staircase step, one constant value for a long time and then another constant value, usually 0 and 1.
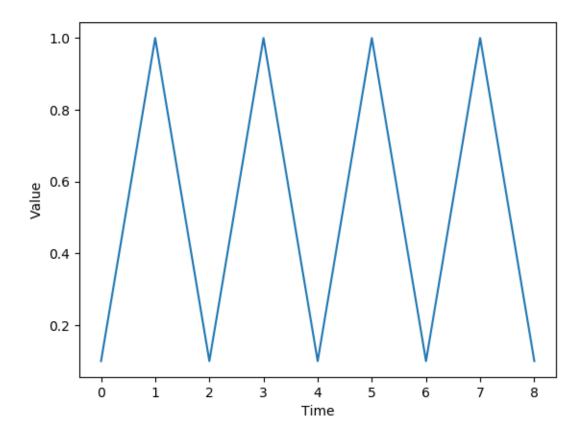
Ramp

A ramp is another intuitive signal. It has the shape of a physical ramp we are used to.

Ramp
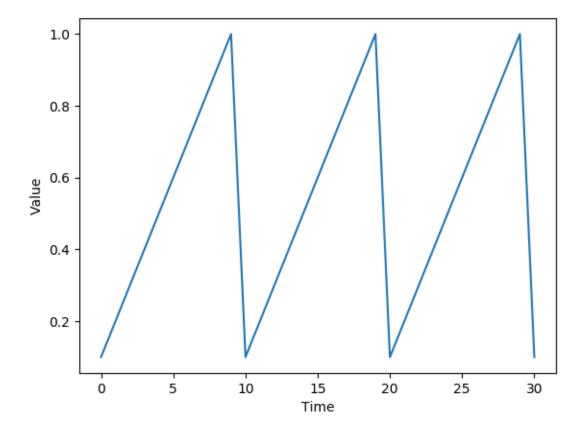
Triangular wave

A triangle shaped wave is encountered in some designs. Think of this as a repeating pattern of rising and falling ramp signals.
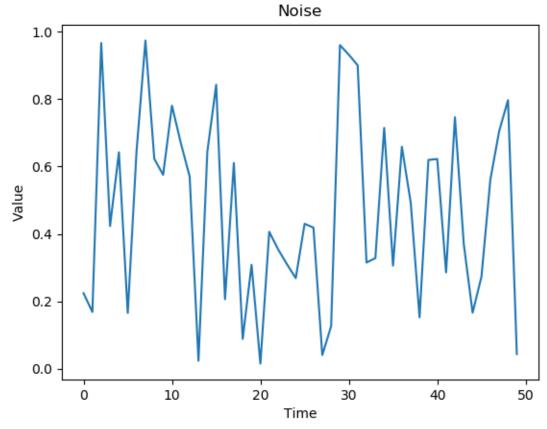
Sawtooth Wave

A sawtooth wave is a special kind of triangle wave where one slope is steep.
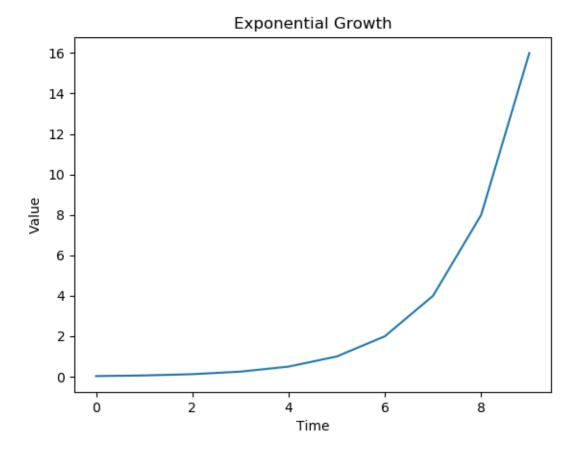
Noise
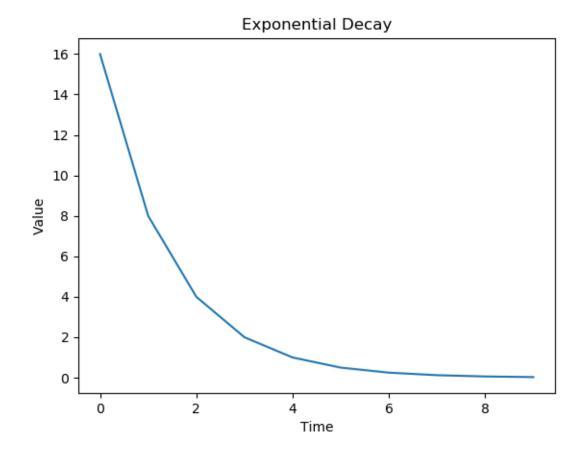
Humans like patterns and there is one signal that defies all understanding. Because of its deeply confusing and patternless-ness, noise is special. It is very difficult to see a pattern in noise.

Exponential
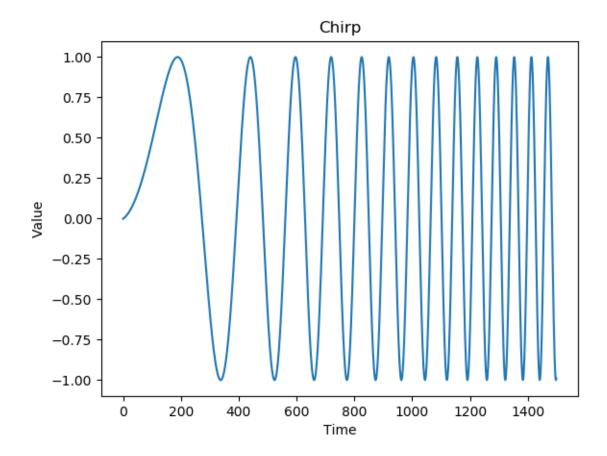
Things in nature that grow geometrically, like the population of an invasive species or the nuclear fissions per second in an atomic bomb or the compounding of interest on a deposit are exponential growth processes. There is also the opposite version that is called exponential decay. Radioactive decay, depreciation of physical assets and extinction process of species follow exponential decay. Note however that no natural process is purely exponentially growing or decaying all the time. Only in the given time window, it may be ok to view the change as exponential growth or decay.

Exponential Growth

Exponential Decay

Chirp

The chirping of birds have a characteristic increase or decrease in pitch of the whistle like sound of the birds. The plot of a chirp signal looks like a sine wave except that the rate of change either increases or decreases. The figure you can see that the rate of change is slow at first and then increases higher as time passes.

Chirp

Exercise

The Python programs to plot the waveforms show before are in the <git repo>/useful_scripts/waveforms/ directory. Can you use the examples for sinewave and step function and plot sinc function using a Python program?

https://en.wikipedia.org/wiki/Sinc_function#/media/File:Si_sinc.svg

Control Theory

Whenever there is feedback in a simple system, a well defined pattern of behavior emerges. The system remains unsettled in some cases, known in control theoretic parlance as oscillating or unstable. For most other cases, the system tracks a desired output after a brief period of "ringing". Electronic circuits, feedback loops in logic and even in software, electrical and mechanical systems can use the concepts of control theory for design. Note, however, that control theory may not apply very well to analyzing a human brain. This is because the regular control theory is more applicable to linear

systems, that is, systems that can be modeled with differential equations. Human brain defies such oversimplifications.

To see a live demo of feedback, make a call from your phone to another phone. Answer the call on the second phone and start bringing both phones nearer to each other. While you bring the phones closer, make sure to keep the mic side of one phone to point to the speaker side of the other phone. You will start hearing a whistling or periodic beating sound when the phones are close enough. This is feedback!

https://en.wikipedia.org/wiki/Feedback

https://en.wikipedia.org/wiki/Control_theory

https://en.wikipedia.org/wiki/Barkhausen_stability_criterion

https://en.wikipedia.org/wiki/Signal-flow_graph

https://en.wikipedia.org/wiki/PID_controller

https://en.wikipedia.org/wiki/Zeros_and_poles
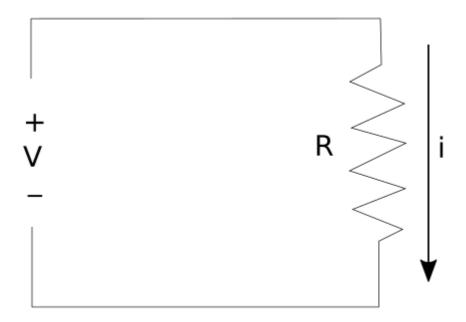
https://en.wikipedia.org/wiki/Z-transform

https://en.wikipedia.org/wiki/Bode_plot#Gain_margin_and_phase_margin

Circuit Theory
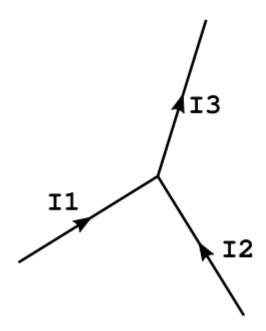
Some amount of simple circuits are inevitable in a chip datasheet and to interface with the outside world. It is not that hard to learn the basics of electrical circuits either.

https://en.wikipedia.org/wiki/Ohm%27s_law
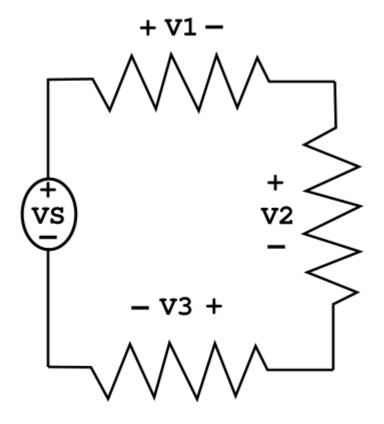
Kirchhoff's Current Law

Voltage Law

Exercise

Your phone has only 10% charge left. You are in your friend's place. You want to charge your phone fast. There are two chargers, one big and one small. The big charger has a fine print of 5V and output current of 1A. The small charger has 5V and 2A. Would you use the small charger or the big charger?

More Circuits

This part is optional for most chippers, but, some may need to understand these concepts at a high level without bothering with the details. Try to get an overview, the equations are not too important for a chipper. It is needed mainly for circuit designers.

https://en.wikipedia.org/wiki/RLC_circuit

V-I characteristics of PN Junction diode, BJT and MOSFET

CMOS Digital Circuits – Inverter, NAND gate, NOR gate, pass transistor, transmission gate, tristate inverter, flip flop, MUX

CMOS Analog Circuits – MOSFET diode, current source, current mirror, differential amplifier, comparator, bandgap reference, oscillator.

https://en.wikipedia.org/wiki/Operational_amplifier

Opamp circuits – unity follower, summer, inverting amplifier, integrator and differentiator

Power Converters

https://en.wikipedia.org/wiki/Buck_converter

https://en.wikipedia.org/wiki/Boost_converter

https://en.wikipedia.org/wiki/Buck%E2%80%93boost_converter

https://en.wikipedia.org/wiki/Low-dropout_regulator

Electromagnetic Compatibility

https://en.wikipedia.org/wiki/Electromagnetic_compatibility

Embedded Systems

What happens when electronic circuits marries software? You get embedded systems! Pure software programming tries its best to detach from the underlying hardware that the program will execute on. Electronic circuit designers try best to detach from the world of software development. In between the two, there is a very interesting place where both software concepts and hardware concepts are at work. A more technical definition would be systems that contain a processor but is meant for a dedicated application and not a general purpose application. So a computer - desktop/laptop/supercomputer, a tablet are not considered an embedded system. A WiFi router, TV, refrigerator, printer, etc when they contain a processor come under embedded system. Note that the lines are blurring. Some years ago, phones were considered embedded systems. Now, I am not so sure. My phone probably has more compute power and many more user apps available than my laptop.

It is rather easy to get a feel of embedded systems. Get an Arduino kit and start programming. If you have a bigger budget, get a Raspberry Pi and try.

Topics to understand in an Embedded System are:

Electronic Circuits

In embedded systems, current, voltage and power of the peripherals connected to the processor are all important.

Embedded C programming

This is similar to programming a desktop application in C but needs deeper understanding of interrupt service routine, device drivers, pointers and ideas for reducing the program size and runtime.

Real Time Operating System

Operating systems meant for general purpose computers are not well suited to embedded systems because of a version of "hang" problem. How may of you have experienced the phenomenon of your laptop freezing? Such a hang is not at all acceptable in embedded systems. It is not even acceptable for

the system to respond too slowly. The real time qualifier is meant to say that RTOS will ensure timely results. To get timely performance, RTOS have tweaks to the general OS that stops the execution of less critical code in favor of a more critical code.

BSP or board support package is another interesting topic in embedded systems.

Communications Technology

When people grow up watching live steamed videos and video chatting with grand parents, it is hard to acknowledge the massive improvements in communication technology that has taken place over many decades. There was a time when men and animals used to carry small bits of information at a high cost in terms of modern communication jargon of latency and packet loss. All this for only a few kilobytes worth of data. Fast forward to 2020, regular public can transmit gigabytes worth of data across the world with practically zero packet loss and with latency in the milliseconds range. Chips are at the foundation of this remarkable transformation. As a chipper, you will certainly deal with some kind of communication technology in your career. Though not mandatory to do chips for communications products, a basic understanding of the topics listed here helps in making a better chip.

Information theory, line codes, error correction codes, modulation, multiplexing – Time division, frequency division, code division, multiple access

Radio Frequency concepts – antenna, transmission line, characteristic impedance, impedance matching

Communication Standards

Bluethooth, WiFi, LAN, Optical, CDMA

Information Networks

Networks that carry information from one device to another have grown in size drastically over the last few decades. When many devices are connected to each other via a communication link they form a network. Note that the connection does not always have to be physical using a wire or fiber, wireless works just fine. The network used to be called a computer network. But that term looks obsolete now because it is not just computers that are on the network. Now, there are light switches, sensors, home AC and many unseen devices on the network. A good understanding of the fundamentals of networking is useful to design chips. If you study many networking protocols you will see a common theme of layering. The task of communicating between devices is divided across many layers with each layer only interested in a specific aspect of the communication. The Open System International or OSI model is a good starting point to understand this layering. The OSI is only a reference model for understanding. You can accomplish the operations of these layers in any other completely valid architecture.

https://en.wikipedia.org/wiki/OSI_model

Compared to the more theoretical OSI model, the TCP/IP model is more practical. TCP/IP is what the Internet runs on. Many chips implement some protocols of the TCP/IP stack. The lower layers of the protocol stack are more useful for a chipper than the higher layers because the higher layers are mostly software based.

https://en.wikipedia.org/wiki/Internet_protocol_suite

Cryptography

The simplest explanation of cryptography is to manipulate information into a form readable only by the interested party. Note that every code can be broken given enough examples and compute power.

Start with substitution cipher that is one of the most basic encryption techniques.

https://en.wikipedia.org/wiki/Caesar_cipher

https://en.wikipedia.org/wiki/Substitution_cipher

https://en.wikipedia.org/wiki/Transposition_cipher

https://en.wikipedia.org/wiki/Encryption

https://en.wikipedia.org/wiki/Public-key_cryptography

https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

https://en.wikipedia.org/wiki/RSA_(cryptosystem)

Security

Is your design prone to misuse? Do you want to prevent hackers from taking control of your design? Security is the field that deals with protecting chips from hacking. Security takes many forms. You may want to make your design difficult to copy, you may want to encrypt the data passing through your chip, you may want to prevent unauthorized people from taking control of your chip. Most of the security measures relevant to chipping are accomplished through cryptographic techniques. If not design cryptographic IP, a chipper very likely would integrate or deal with some kind of security issue in his lifetime.

https://en.wikipedia.org/wiki/Cyberattack

Semiconductor Counterfeiting

Security is applicable in ways that are hard to imagine. Can you believe that full smartphone chips with as many as a billion transistors can be cloned? Counterfeiting is a menace to chip companies. Suppose you have invested $10M in designing a new chip and are selling one chip for $10. Say, a counterfeiting company gets the layout of your chip and starts producing unauthorized versions of your chip and is selling it for only $8. You would be powerless to stop it! This is one other area of the application of security. A simple solution in these kinds of situations is to have an authentication mechanism built into

the chip. After manufacturing, the original company programs valid keys into the chip. Whenever the chip powers up, it checks for a valid key. If the key is found all subsystems of the chip are activated and if there is no valid key the chip is blocked from full activation.

You may be thinking, can this be hacked? Of course! At least it is better than chips that have no authentication.

https://www.eetimes.com/author.asp?section_id=40&doc_id=1332064#

Functional Safety and Reliability

It is a nice spring evening in Yosemite valley, California, USA. After a memorable vacation with your family in the scenic Yosemite valley, you are heading back to Silicon valley for work. It has just gotten dark and you are driving along the mountain roads with your fancy new automatic headlights on. There is a few hundred meters of straight road ahead that you can see by headlights of oncoming traffic. Beyond that is the unknown windy mountain road. After traveling for some distance along the straight road both the headlights go dead. Not knowing what to do, you drive straight. Disaster! Your car is rolling down the mountain side. The road actually turns left but you happened to drive straight.

Do you want to prevent your drone from accidentally crashing to the ground because of a fault that developed during use? Do you want your car to not accidentally speed up in cruise mode because of a fault? Functional safety is a blanket term covering the tasks of analyzing the probability of failure, concepts to reduce failure and analyzing the effects of failure. Automotive, Aviation, Medical and Defense industries need chips that are safe to use. In space industry, the cost of space grade hardware is high and the cost of launching is even higher, so fault tolerance is a valuable feature. In safety conscious industries, functional safety is an important aspect of chip design, so much important, that without safety features your amazing design will never be bought by any customer. You may have the fastest computer for self driving cars but without safety features it may not be permitted on public roads!

https://en.wikipedia.org/wiki/Failure_modes,_effects,_and_diagnostic_analysis

https://en.wikipedia.org/wiki/Failure_mode_and_effects_analysis

https://en.wikipedia.org/wiki/ISO_26262

https://en.wikipedia.org/wiki/Automotive_Safety_Integrity_Level

http://www.aecouncil.com/Documents/AEC_Q100_Rev_G_Base_Document.pdf

Reliability

Apart from outright injury or death, lesser forms of reliability is important for all chips. Would you like your phone to die in the middle of a phone interview for a new job? As a chipper, there is in reality only one solution for both safety and reliability, a spare tire. An automobile can get a flat tire on any one wheel with a high probability. The spare tire makes sure that when it does happen, it can be quickly

replaced and the journey can be completed. Advanced forms of safety features inside a chip are variations on this theme of redundancy.

https://en.wikipedia.org/wiki/Built-in_self-test

https://en.wikipedia.org/wiki/Triple_modular_redundancy

In my understanding of safety standards, there seems to be a big flaw. Many of the safety standards are more focused on avoiding the impact of physical faults developed during manufacturing or operation of a chip and not too keen on design bugs. The end result of a bug or a fault can both be catastrophic. One way to ensure a high level of safety in the design is to use design and verification processes that have safety built into them.

For example, the software tools used for high safety applications are themselves tested for higher quality than regular software. The summary of the verification for safety critical applications is this - "who will verify the verification suite?". Now, some tools are out that promise to bulletproof your verification suite.

https://www.synopsys.com/verification/simulation/certitude.html

Audio Systems

Second only to visual, speech forms the bulk of human communication. The world of chips mirror this importance to audio. Almost all user oriented devices have a mic and or speaker. Audio is a relatively simple system to work with. A sound wave is a mechanical vibration in the air, or some other medium. Unlike light, sound does not travel in vacuum. Sound is of far lower frequency than light. Knowledge of concepts in audio theory helps in doing good chips for audio systems. Digital Signal Processor or DSP is used to process audio leveraging many signal processing algorithms.

https://en.wikipedia.org/wiki/Sound

https://en.wikipedia.org/wiki/Microphone

https://en.wikipedia.org/wiki/Loudspeaker

https://en.wikipedia.org/wiki/Equalization_(audio)

https://en.wikipedia.org/wiki/Stereophonic_sound

https://en.wikipedia.org/wiki/Surround_sound

https://en.wikipedia.org/wiki/Active_noise_control

https://en.wikipedia.org/wiki/Audio_codec

https://en.wikipedia.org/wiki/I%C2%B2S

Camera Systems

It all starts with photons, the Instagram post, the Facebook photo update, the latest fashion of AI image recognition and the best selfie. Before the birth of chips, photography was a chemical enterprise. Now it is a chip enterprise. If the Internet traffic is mostly dominated by cute cat videos, then the chipping industry is ruled by image and video related enterprise directly or indirectly. An image is approximated in digital systems by a two dimensional array of numbers. Every value in the array is a recording of the light intensity that fell on that array. The individual points in the array are known by pixel. There is both physical and abstract meaning for a pixel. The physical semiconductor photodiode and its circuits that combined record light intensity is also known by the word pixel, while at the same time, the value in an image file holding the light intensity at that point is also known by pixel and at the same time, the dot that appears on a TV or table or phone screen is also called pixel. An image sensor is a collection of a photodiode array and its associated analog and digital circuits that take a full snapshot of a scene. The image sensor may output the data in analog or digital format. The digital formats are the ones more useful for the future, so, lets look at just that. The pixel values may be represented in raw output that needs to be processed or RGB format or YCbCr format. The data going out of the chip may go out in a simple parallel bus driven using LVDS standard or use a serial standard like MIPI CSI-2/DPHY or CPHY.

A video is a series of images. In practice, video presents greatly increased difficulty for chipping than still images because of the amount of data that is streaming. Imagine a UHD video with 3840x2160 pixels with each pixel represented as a 24 bit number. Let the frame rate be 30 frames per second. Now the overall data rate works out to 3840x2160x24x30 bits per second or close to 6Gbps!

https://en.wikipedia.org/wiki/Photodiode

https://en.wikipedia.org/wiki/Pixel

https://en.wikipedia.org/wiki/Raw_image_format

https://en.wikipedia.org/wiki/RGB_color_model

https://en.wikipedia.org/wiki/YCbCr

https://en.wikipedia.org/wiki/Camera

https://en.wikipedia.org/wiki/Color_image_pipeline

https://en.wikipedia.org/wiki/High_dynamic_range

https://en.wikipedia.org/wiki/JPEG

https://en.wikipedia.org/wiki/Portable_Network_Graphics

Display Systems

The image that is captured by cameras need to go somewhere. The most common destination is a display. Imagine a world without touch screens or even regular LCD screens. Wristwatch, TV, phone

and computer would all look very different. If an image sensor captures a snapshot of the light intensity in an array of photodiodes, a display system projects that image using an array of LEDs or liquid crystal elements. Note that artificial images and symbols are also an important things to be displayed. They include computer graphics, computer generated symbols and text.

The following topics make for good additional reading.

Display Technology

LCD, OLED, ebook reader

7,14,16 segment displays, dot matrix displays

Display protocols

VGA, DVI, HDMI, DP, eDP, DSI

Project Management

It is not all about design that makes a chip succeed. Even a small design can benefit from a decent project plan. Large projects can not do without a plan. If you have not encountered project planning as a topic, this section will introduce the basics.

The first and the simplest to understand is a waterfall model of project management. Suppose your project is to paint your house, you would breakdown the work into selecting the paint type and color, choosing a contractor and letting the contractor paint. Every step will be allocated some time for completion and marked with an owner of the task. I have used waterfall plans for most of my work and I have seen most projects executed in this fashion. But there is a big limitation to this method of management. It is very rigid and misses deadline most of the time because important details about the work are easy to miss at the start of the plan.

https://en.wikipedia.org/wiki/Waterfall_model

Tools like Microsoft project is a leading project management tool, though, paid. There are open source and free alternatives available.

https://en.wikipedia.org/wiki/Microsoft_Project

Agile

I have started to practice the Agile model of project planning. It is very powerful. In an ever changing world, Waterfall model is not good enough. Agile needs a whole book to itself. I will leave the details as an exercise.

https://en.wikipedia.org/wiki/Agile_software_development

The idea of Agile project management has been applied to hardware development.

https://people.eecs.berkeley.edu/~bora/Journals/2016/IEEEMicro16.pdf

NVIDIA C++ AI accelerator agile hardware design example

http://crva.io/documents/agile-and-open-hardware/khailany-sigarch-visioning-oahw2019.pdf

If you hate the idea of project management as anti-engineer you are not alone. I don't know of any chipper who likes PM. That is why large companies have a dedicated project manager who can be collectively hated :). He will take the ill will of the chippers in place of top executives!

Anyways, not planning is not an option! My boss once told me that the act of planning is more important than the plan itself. This is because it lets you see the pitfalls ahead and take corrective action.

Productivity

Numerous skills are expected from a chipper in his first job. Most of the productivity skills are not even specified in a job posting because hiring companies assume any applicant with a degree would possess it anyway. But, the aim of this book is to encourage just about anyone with any background to enter chip design. So, I am spelling out these skills.

Working with Microsoft Windows

Working with any one Linux distribution – RedHat, Fedora, CentOS, Debian, Ubuntu

Basic knowledge of a spreadsheet program – Microsoft Excel is preferred, other acceptable

Presentation Software – Microsoft PowerPoint preferred

Basic word processor – Microsoft Word preferred

Ability to use email – MS Outlook preferred, Gmail, Yahoo or any other

Chat and messaging – Microsoft Skype, Google Chat or Facebook Chat

One code editor - Vi or Emacs or Sublime Text or Notepad++. DVT Eclipse is a paid tool for commercial use and free for educational use. DVT is a pretty feature rich tool for code editing and debugging.

https://www.dvteclipse.com/

Version Control

Have you ever made a copy of an important document or a copy of a computer file for back up purpose? Version control is in a way mainly for that use. But then once you have a series of backups taken at different times, you now have a powerful tool to help debug mistakes. Ability to use a version control software is a key skill for a chipper. I have chosen Git because it is free and widely used. I learned Git from the Massive Open Online Course (MOOC) from Udacity, "How to Use Git and GitHub?"

https://www.udacity.com/course/how-to-use-git-and-github--ud775

GitHub is an online code sharing website built on top of Git. A large amount of open source code is now available through GitHub.

Unit 2 – Concepts in Chipping Practice

ASIC Technology

Application Specific Integrated Circuits are the class of chips that are designed with only one specific purpose in mind. The ASIC has no programmability or only a limited programmability. The meaning of programmability in this context is the ability to change the connections of the internal logic gates and not software programmability. The lack of programmability is the power of the ASIC. No wasting time choosing between all the possible things the logic gates can be. No wasting area trying to please all possible designs that need to be supported and no wasting power on logic that is never going to be used. ASICs are very costly to design. A simple ASIC could take from a few $100k to $1M or more. This one time cost is called Non Recurring Engineering or NRE cost. If the ASIC is manufactured in large volume of 100 million the NRE is amortized over all the chips produced. For $1M NRE and 100M chips sold, the NRE cost per chip works out to only 1 cent! At high volumes the cost of one chip comes down to purely the recurring costs of silicon area, testing for every chip, the cost to package and the cost to sell the chip. To give a sense of the chip sales volume, I googled for Iphone 6 sales and it returned a whopping 220M!

FPGA Technology

Contrast the limited programmability of ASIC with its close cousin, the Field Programmable Gate Array with its ability to reconfigure the logic gate connections by just downloading a programming file. An FPGA does this by providing a ready made array of gates and standard chip components like PLL, ADC, high speed transceivers and more building blocks that can be connected or disconnected by the bits in the programming file. The programming file or bit file is loaded into an internal memory that could be non volatile or volatile. After loading, the values of the bits realize the specific logic by controlling the selection lines of multiplexers. Another part of programming happens by loading the values of Look-Up-Tables or LUTs which are nothing but a line by line implementation of the truth table of the logic to be realized. For example, to implement a 2 input OR gate with a LUT, you program -

0,0,0
0,1,1
1,0,1
1,1,1

FPGAs can sell from a few dollars to a few thousands of dollars. FPGAs are mainly used for low volume systems and for prototyping ASIC designs before committing the huge NRE cost of ASICs. If you are a user of FPGA, it is relatively easy to program compared to designing and manufacturing an ASIC. FPGA designers do not have to worry about designing the layout of the chip or the testing of the chip for manufacturing defects. But using FPGAs creates the other problem of plenty. There are so many building blocks available in an FPGA that it takes a long time to get proficient in using them.

Circuit Design

Chips or components of chips always need handcrafting for special needs. For example, a full adder that needs to add two 32b numbers at 20 giga operations per second may need artisan like build. To get this level of performance, chip designers, called circuit designers start by creating the transistor level schematic of the design, then simulate the circuit with some type of continuous time simulator. It is most commonly a differential equation solver for electric circuits. SPICE and its derivatives do this job. The circuit simulator takes as input the transistor level schematic of the design and the model files that define the electrical behavior of the devices like transistors. It is much more accurate than digital simulations that are used for logic design. Circuit design is much slower compared to logic design in terms of the number of transistors used per unit design time. So, it is relegated to special needs like analog circuits, high speed transceivers and radio frequency circuits. Circuit design may interchangeably be referred to as custom design or analog design or transistor level design.

Logic Design

Logic design is what this book is mostly concerned with. Logic design refers to designing chips or programming FPGAs at the level of digital logic, that is 0 and 1. The voltage and currents sacred to circuit design are secondary to logic design. The productivity of logic design process is vastly higher than circuit design and so most modern chips try to maximize logic design use and minimize circuit design use.

Hardware Description Language (HDL)

As the name says it, HDL is all about describing how a hardware is constructed and how it should behave. Think of this as your second language after your mother tongue. The more fluent you are in this the better designs you can make.

Verilog 2001

Well, well! How I tried to not even mention Verilog 2001. Unfortunately, this outdated Verilog standard is still widely in use in the open source community. This is mainly because there are plenty of legacy code and there are many completely free simulators for it. To be able to use older code you need to be able to understand the quirky ways used in Verilog 2001.

I learned Verilog 2001 from the book, Verilog HDL by Samir Palnitkar

This book looks good too -

Fundamentals of Logic with Verilog design

http://www.eecg.toronto.edu/~brown/Verilog_3e/verilog_source/Welcome.html

VHDL

This HDL gave a tough fight to Verilog, but, eventually has lost out to SystemVerilog. Still, VHDL is used by companies or institutions in Aerospace and Defense industries. It is also somewhat more common in Europe. There are many open source designs available only in VHDL. So, it is worth understanding VHDL. But note that it is not to be used for new designs. Actually, VHDL was the first HDL I learned.

Book: A VHDL Primer, by Bhasker

SystemVerilog

SystemVerilog (SV) is the latest standard of Verilog and all new designs are done in SV. But until recently SV simulators were not available for free. For the near future, most of my work related to this book will be in this language. Just as any natural language like English or a software language like Java, there are rules in using it. This is called the syntax. The master document specifying the syntax of SV goes by the name Language Reference Manual (LRM). SV has 3 sets of constructs within it. The design subset is used for RTL that can be synthesized by logic synthesis tools. The modeling subset is for simulating the working of circuits that are not synthesizable. The third subset is for verification. It is the verification subset that is unmatched by any other HDL language. So, the SV LRM actually calls itself HDVL, "V" for verification. This book contains a quick reference to basic SV for design and numerous examples you can use to bootstrap your projects.

For more help on SV refer to -

Websites

I have actually learned many SV examples from this one

http://www.asic-world.com/

A cost effective source to get most of the SV design constructs

http://sutherland-hdl.com/papers/2013-SNUG-SV_Synthesizable-SystemVerilog_paper.pdf

Books

Logic Design and Verification Using SystemVerilog, March 2016

SystemVerilog for Design Second Edition: A Guide to Using SystemVerilog for Hardware Design and Modeling – 21 Aug 2006

CHISEL

This is a newer HDL. Apparently, CHISEL has greater expressive power compared to Verilog and simulates faster. I see two main advantages to CHISEL – more abstract style of design increasing the productivity of chippers and fully free simulator coupled with the ability to export synthesizable

Verilog code. So far, I have not seen CHISEL being widely used in the industry. You never know, CHISEL could very well be the HDL of the future!

https://github.com/schoeberl/chisel-book/wiki/chisel-book.pdf

C

C is the grand daddy of SV. It still has a life in modern chipping as a modeling language that possibly runs fastest compared to SV, C++ or SystemC. Xilinx Vivado HLS tool can synthesize many C constructs, including library components like mathematical functions from "math.h". Old C has a new life now. Many other tools are becoming available to be able to directly program hardware using C code. Imagine the power of billions of lines of preexisting C code becoming available to chippers!

https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

C++

Almost like C but with a superior capability to scale to large designs, C++ is the language to learn because you can do large software and hardware with just one language. Vivado HLS can handle C++ as well.

SystemC

SystemC is a library built with C++ primarily for modeling hardware. After Vivado HLS support, SystemC can now be seen as a design language too, just like C and C++.

https://www.accellera.org/downloads/standards/systemc

Matlab

Matlab is a software package from Mathworks. It is widely used in modeling signal processing logic. It is very well suited to coding algorithms super conveniently compared to low level languages like C. Matlab used to be limited to only the modeling world. Some years back, Mathworks released a feature called HDL coder to convert the Matlab scripts into Verilog. So, I call Matlab as a design language now!

https://www.mathworks.com/products/matlab.html

https://in.mathworks.com/products/hdl-coder.html

Basics of Verification

Verification is a whole different ball game that needs a separate library and this book is more about design than verification. Anyway, verification even at the level of a small design is the fire to forge strong designs. If you ask me to name only one verification textbook I would say this one -

SystemVerilog for Verification: A Guide to Learning the Testbench Language Features, Third Edition, by Chris Spear, Greg Tumbush

Directed testing vs Constrained random verification

https://www.doulos.com/knowhow/sysverilog/tutorial/constraints/

The trick to constrained random or CR verification is to create random stimuli representative of real traffic covering all cases with less inputs. I found these blogs interesting to learn a few applications of CR verification

http://blog.verificationgentleman.com/2014/05/fun-and-games-with-crv-sudoku.html

http://blog.verificationgentleman.com/2015/02/fun-and-games-with-crv-n-queens-problem.html

Assertions

Assertions describe the properties of the design. More on this topic is here

SystemVerilog Assertions and Functional Coverage, Guide to Language, Methodology and Applications, Authors: Mehta, Ashok B.

https://www.springer.com/gp/book/9783319305387

Formal Verification

Rather than run simulation, there is a way to check the design by looking at the properties of the design.

https://www.systemverilog.io/gentle-introduction-to-formal-verification

Universal Verification Methodology

It is true that verification takes much longer time than design. In the past, every company developed their own version of a verification methodology to reuse the verification effort as much as possible and automate the verification flow. UVM was developed as a response to the adhoc methods used by each company. UVM is a library on top of SV and it provides features that greatly ease verification tedium. UVM is also a methodology and I would even go as far as say, a philosophy of verification. Whatever it be, if you aspire for a verification job you ought to know UVM!

UVM is a bit intimidating at first. It is very different from the design level SV code. It resembles C++ code in its object oriented style with many classes. These books provide a great start to UVM -

The UVM Primer, by Ray Salemi

https://www.amazon.com/dp/B00GBD62HK

UVM Cookbook from Verification Academy

https://verificationacademy.com/cookbook/uvm

Portable Stimulus Standard

In chipping as in any other domain, there are plenty of rework. The test performed during pre-tapeout verification is mostly repeated in the lab, though using a different language and script. Again, during manufacturing testing, some form of the test may be repeated too. PSS aims to provide a mechanism to reduce this multiple work. Or, that is what I understand of it! As much as I am aware, PSS has not been adopted widely yet. It may take a few years to become commonplace.

https://www.accellera.org/activities/working-groups/portable-stimulus

Simulation

Simulator

SystemVerilog code is meant to describe digital hardware. Simulator is a software tool that compiles the SV code into a CPU executable binary. When the CPU executes the binary, the result is as though the digital hardware magically worked without a physical chip. Sure, the simulation is slower than real hardware because simulation is mostly serial execution of code in the CPU whereas the real hardware is fully parallel. Every gate in hardware works simultaneously without waiting for other gates to complete there work. There are many simulators for SV. Cadence has Xcelium and Incisive, Synopsys has VCS, Mentor Graphics has ModelSim, Xilinx has Vivado. The ModelSim and Vivado are available for free. I have used Vivado webpack free edition for simulating most of the code in this book.

There are some free and open source simulators

https://www.veripool.org/wiki/verilator

Icarus supports Verilog-2005

http://iverilog.icarus.com/

Waveform Viewer

The simulator virtually runs your design on a CPU. However, when your design does not work as expected there is no way to know what went wrong. The most common way of debugging is to save the waveforms of the signals of interest into a waveform file. A waveform viewer can open and display this file in a user friendly way. Becoming a power user of the waveform viewer will drastically increase your productivity.

Xilinx Vivado

https://www.youtube.com/watch?v=i8axs4hw2f4

Cadence Simvision video series

https://www.youtube.com/watch?v=a2YmCK0uzJs&list=PLYdInKVfi0KYzCjnkgRgDXFJcKyQRz6eM&index=1

Performance

There are ways to increase the speed of simulation. Many of these involve changes to coding style that leverage the optimizations built into the simulator. The basic idea is to reduce the internal operations to be performed by the simulator. Simulators work on events internally created during the simulation process. So, if the coding style minimizes events, it will speed up simulation. The easiest way to reduce events across the board is to reduce the time precision or time step of the simulation to a value that is about a tenth to a hundredth of the time delays in the design. Another easy step is to use datatypes that are a native match to the simulator implementation. Generally, 32bit types are the ones used internally. So, using a 32b integer in place of a 64bit type when possible speeds up simulation. Ironically, using 16b type may be slower than using 32b types! This paper gives some interesting points on simulation performance -

http://www.sunburst-design.com/papers/CummingsDVCon2019_Yikes_SV_Coding_rev1_0.pdf

Profiling

Sometimes it is necessary to dig deeper into the root cause of a slow simulation. A profiler is a tool that comes with a simulator to analyze the run time consumed by lines or blocks of code. The profiler adds instrumentation to the simulation and collects line specific runtime statistics from an actual simulation run. After the simulation is complete the collected results are presented in a nice looking report or GUI. You can look at the most time consuming part of your code and think about re-coding them for faster simulation.

Lint Check

How good is your HDL code? Is there a difference between good code vs bad code? The industry has come up with a quick way to check the quality of the code, it is called Lint check or just Lint. The Verilog language is meant to model hardware in English. It is not perfect! There are ways to code that would appear correct but would cause subtle problems later in the design cycle. I learned quite a bit about bad code from this book.

Verilog and SystemVerilog Gotchas - 101 Common Coding Errors and How to Avoid Them Authors: Sutherland, Stuart, Mills, Don

https://www.springer.com/gp/book/9780387717142

Free condensed PDF

http://www.sutherland-hdl.com/papers/2006-SNUG-Boston_standard_gotchas_presentation.pdf

In the chip industry, dedicated software tools are used to find bad code before it escapes into the real world in the form of chips used in finished products. Synopsys Spyglass Lint is one such tool. Lint does not rely on a simulator. The Lint tool scans through the HDL code and compares it against a predefined set of rules. Any violation of the rules is flagged as error.   The license cost for many of these tools are typically high, so a small chipper may have to rely on other ways to check the code quality. In the

absence of a lint tool, one crude way is to carefully study the warning messages while the simulator compiles your design. These messages could point to bad code. Another way is to look for warning messages when you perform logic synthesis for your design. Synthesis is described briefly in subsequent sections. An even more brute force method is to do a gate level simulation of your code after synthesis.

Standard Cell Library

If a chip is thought of as a house, standard cells are its bricks and tiles. The parallel has many interesting points. You may see that bricks and tiles come in standard sizes and that is for a reason. Fixed sizes help in planning. In the chip world, standard cells come in fixed range of sizes. Functionality of one standard cell could be a simple gate like NOT/NAND/NOR/AND/OR/XOR/XNOR or some more complicated logic like full adder, AND-OR-INVERT gate. Standard cell libraries also include many sizes of buffers and inverters for creating clock and signal trees. Sequential cells like flip flops, latches and grouped flops are also present. Flops with combinational gates joined at the d input is also common, especially, flops with mux and enable signals. Standard cell libraries differentiate themselves in the spectrum of power-performance-area targets achieved.

https://en.wikipedia.org/wiki/Standard_cell

Liberty .lib example http://www.utdallas.edu/~hxh025000/index_files/library.lib

Basic versions of standard cells may be available Si2.org, I am not sure though!

Www.si2.org

The company ARM is a market leader in standard cell design. ARM provides access to a part of the library free of cost for companies. If you enter mass production there may be royalty to be paid. But for learning, it is mostly free. You may download and get a feel of standard cells.

https://developer.arm.com/ip-products/designstart/physical-ip

The availablity of special cells called ECO spare cells help in "patching" your buggy chip quickly at low cost.

https://www.design-reuse.com/articles/38868/metal-eco-implementation-using-mask-programmable-cells.html

As far as chipping is concerned, you need to be aware of standard cells and their usage in logic synthesis, place and route and timing analysis. But you will not be directly using them except for ECO purposes.

Logic Synthesis

Logic Synthesis converts the abstract HDL code to a netlist that can be passed to other tools for manufacturing in a fab or to program an FPGA. Suppose your design is a simple combinational logic with the RTL shown here, synthesis will turn the RTL into a netlist like the one shown.

```
RTL
module my_combo_logic (
input a, b,
output z
);
  assign z = a & ~b;
endmodule
```

```
Netlist

module my_combo_logic (
input a, b,
output z
);
  wire net1;
  INV4X i0 (.zn(net1), .a(b));
  AND2X i1 (.z(z), .a(a), .b(net1));
endmodule
```

INV4X and AND2X are gates from a standard cell library matching the target fab where you would like to manufacture your chip. From this point, the design netlist is married to the fab. It cannot be easily used to manufacture the chip in another fab. Note that RTL code is generic and unattached to any fab.

In case you are targeting the design for an FPGA then the netlist may look different.

Synopsys Design Compiler and Cadence Genus are some synthesis tools for ASIC synthesis. Xilinx has its own tool for FPGA synthesis and Intel/Altera may have one of its own too. There is a free and open source tool for ASIC and FPGA, but, your RTL may need to be restricted to Verilog 2001. I wish someone extends this into a full featured SystemVerilog synthesis tool.

http://www.clifford.at/yosys/about.html

Entire books are dedicated to Synthesis. Use the following references if you want to know more.

http://www.asic-world.com/verilog/synthesis1.html

http://users.ece.utexas.edu/~adnan/syn-07/

https://nptel.ac.in/courses/106106088/pdf/nptel-cad1-21.pdf

Note that the field of synthesis evolves fast. What is considered behavioral code yesterday becomes synthesizable today. So, start with high level abstract coding style and see if your synthesis tool can synthesize it, rather than code at fine bit level right at the beginning.

Design For Test

Have you ever bought a gadget and cursed the manufacturer when it stopped working? Why do some products malfunction when others from the same brand work fine? Testing catches any chips that got made with faults. Contrast this with verification that catches bugs in the design. Design For Test aims to make the process of catching manufacturing faults easier. This is a huge topic too. Employees are hired exclusively for DFT. A designer needs to understand the basics of DFT but does not need to be an expert in it. Study the following concepts – yield, defect or fault, fault model, stuck-at fault, fault coverage, scan chain and ATPG.

https://en.wikichip.org/wiki/yield

https://en.wikipedia.org/wiki/Fault_model

https://en.wikipedia.org/wiki/Automatic_test_pattern_generation

https://en.wikipedia.org/wiki/Design_for_testing

https://en.wikipedia.org/wiki/Scan_chain

https://en.wikipedia.org/wiki/Fault_coverage

Physical Design

Physical Design converts the netlist output from logical synthesis into a manufacturable layout. Physical Design is also known by other names like backend, physical synthesis, Automatic Place and Route (APR) and Place and Route (PNR). PD happens in many steps – floorplanning, placement, placement optimization, Clock Tree Synthesis, hold fixing, routing. Many passes of optimization can happen in between these main steps.

https://en.wikipedia.org/wiki/Physical_design_(electronics)

Digital design at RTL level needs some understanding of physical design but in practice the two skills are far diverged so just a basic knowledge is sufficient.

Floorplan Aware Synthesis

Traditionally, the conversion of RTL to netlist and then placing, routing used to happen one after another without an option to go back and modify the synthesis step. Some designs need a more interconnected approach between synthesis and PNR. This is because if the synthesis tool does not split the logic into pieces that can be dispersed well, the design may become unroutable during the routing stage. One such issue happens for large decoders synthesized without floorplan aware synthesis. For some issues like the large decoder problem, you could workaround with RTL change by using generate statements to separate the logic into routable pieces. Refer to the large decoder example later in the book. Changing the RTL to deal with PNR stage issues is not a very efficient thing to do for most designs. So, modern synthesis tools accept the floorplan as one other input during the synthesis process.

Logic Equivalence Checking

How would you know if your netlist from synthesis stage is functionally same as the logic you coded in RTL? The Logical Equivalence Check step ensures your finished netlist is logically same as your original RTL design. Think of this step as a second line of defense from software tool bugs. The first line of defense is the correct working of the logic synthesis tool. LEC also goes by the name formal verification (FV), though, FV has a much broader meaning.

A second use for LEC is to check if surgical changes made to fix bugs are correct. These surgical changes are called Engineering Change Order or ECO. When bugs are fixed in a small part of the design by changing the RTL and performing a matching change to the netlist, errors could be introduced during the process. The netlist may not logically match the RTL anymore. So an LEC is performed to make sure the netlist correctly implements the RTL.

LEC can be used as a tool to refactor RTL. As the RTL is coded, it needs to be cleaned up periodically. But if the RTL is changed you need to run the full verification test suite to make sure your design is still working without bugs. You can avoid running verification again by making sure LEC passes. The idea being modified logic that passes LEC is an exact Boolean algebra match of your original RTL and so has no new bugs.

Suppose you are tired of using parts of the design in VHDL, you could create a port compatible SV module and make sure it is logically equivalent to the VHDL original. This is LEC's third use.

LEC tool has a nasty habit of aborting the comparison for some logic that are too large for comparison using the LEC method. This usually happens for large combinational blocks like a 128 bit multiplier. Abort is usually solved by doing the LEC is two or more steps. At every step the LEC engine only compares a netlist that is only slightly different from the previous netlist from the RTL to gate netlist flow. For example, you could do RTL vs pre-scan netlist then pre-scan netlist vs post-scan netlist, then post scan vs optimized netlist and finally optimized netlist vs final post routed netlist. Doing direct RTL vs post routed netlist may not compare fully. There are other ways to deal with the abort problem. You could re-code the RTL to make it easier for LEC to run without abort. The LEC tool generally supports recognition and matching for optimization done by the synthesis tool if both LEC and synthesis tools are from the same company. The synthesis tool drops some hints as a separate file that the LEC tool uses to make a good match. This is the case if you use Cadence Genus and Cadence Conformal LEC tools.

https://en.wikipedia.org/wiki/Formal_equivalence_checking

https://www.einfochips.com/blog/a-guide-on-logical-equivalence-checking-flow-challenges-and-benefits/

https://funrtl.wordpress.com/2018/11/08/lec-logic-equivalence-check/

Static Timing Analysis

How fast can your design run? 10MHz, 100MHz, 1GHz..? How would you know? This part is answered by Static Timing Analysis. Lets deconstruct STA word by word. Static means the design is not simulated. The logic of the design is only minimally used. Timing stands for signal propagation. For STA, the logic of the design is not very important, only, the transitions from 0->1 and 1->0 are important. Zero to one change is called rise and marked 'r' in reports and the other change fall 'f'. Analysis stands for checking if the design will work at the target frequency or not. There are a lot more things to STA. There are jobs exclusively for STA. For further reading, refer to the following

https://en.wikipedia.org/wiki/Flip-flop_(electronics)#Timing_considerations

https://en.wikipedia.org/wiki/Static_timing_analysis

http://www.vlsi-expert.com/2016/09/timing-arc.html

How does a chipper use STA? By setting the timing constraints and understanding the reports from the STA software tool. The most widely used format for timing constraints is Synopsys Design Constraints (SDC) and the most widely used tool is Synopsys PrimeTime. It is worth learning the basics of SDC.

http://www.vlsi-expert.com/2011/02/synopsys-design-constraints-sdc-basics.html

http://www.vlsi-expert.com/2012/02/design-constraint-maximum-transition.html

For most of the designs, only a small set of constraints are used. Please study this list of constraints - create_clock, create_generated_clock, set_input_delay, set_output_delay, set_max_delay, set_case_analysis, set_false_path and set_multicycle_path

Misleading aspect of set_false_path

When a path is set to false, it means a transition along the path is allowed to take infinite time. But in reality, there is a bound on almost all paths in the design. Imagine, changing the font size on your phone and having to wait for an hour for the change to take effect. A rather better constraint is to set a maximum delay limit or a multicycle path constraint. But the max delay or multicycle path constraints make the timing constraints file unwieldy. So, in practice most designers just stick to set_false_path. Just remember that there is a small chance that based on the physical implementation, these paths can get too slow than required.

Clock Domain Crossing (CDC) Check

When a signal from one part of a logic that runs on a particular frequency wants to talk to another part of the design that runs on a different clock frequency there are some restrictions that come into play. Even if the second domain runs at the same frequency but the source of the clock signal is different from the first, inter clock domain communication still needs special care. To understand this better, imagine you are in the west coast of Mexico and your brother is in Singapore. Suppose you want to talk to your brother on a Friday evening Mexico local time, it may already be too late for your brother in

Singapore based on his local time. So, the idea is when a signal travels from one clock domain to another domain, you must make sure that the receiving side is ready. In the chip industry, dedicated software tools are used for this purpose. Synopsys SpyGlass CDC is one such tool. If you cannot afford CDC tools, there are tricks you can use to reduce the chances of making mistakes in CDC.

The first way is rather simple – separate your design into files that have only one clock domain and to another set of files that focus on crossing signals across clock domains.
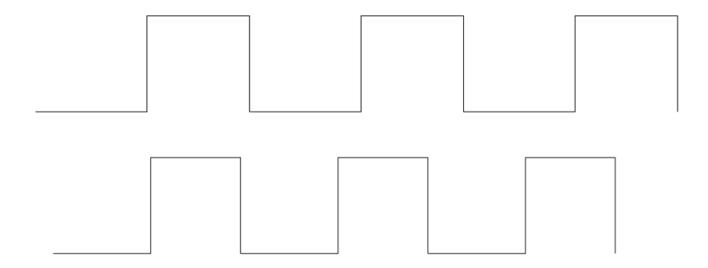
The second way is more tedious but works most of the time. Simulate your design at gate level after synthesis.

The third way is to use a STA tool to uncover potential clock crossing signals and then review the method of CDC used in each case.

The pseudo-code here shows the STA based CDC check (targeted for Synopsys PrimeTime, not syntactically perfect, use the right syntax for your tool). After running this script, analyze the results in the saved file.

```
read the library, design and timing constraints file into STA tool
remove false paths,  remove asynchronous clock groups
set allck [ all_clocks ] ; # get all clocks in the design
#open a file for dumping results
set fptr [ open "STA_CDC_report.txt" "w" ]
foreach_in_collection clk1 $allck {
  set clk1_name [ get_attribute $clk1 fullname ]
  foreach_in_collection clk2 $allck {
    set clk2_name [ get_attribute $clk2 fullname ]
    #for every pair of different clocks in the design
    if { $clk1 != $clk2 } {
      #these are clock crossing paths
      set tpath [ get_timing_paths -from $clk1 -to clk2 ]
      #dump start and end points into a file
      set sp [ get_attribute [ get_attribute $tpath startpoint ] fullname ]
      set ep [ get_attribute [ get_attribute $tpath endpoint ] fullname ]
      puts $fptr "Path exists $clk1_name - $clk2_name : from $sp to $ep"
    }
  }
}
```

The fourth way is to simulate your design with more detailed model for signals that cross clock domains. The catch is that you must first know which signals cross clock domains! Additionally, you can add a small error for the clock frequency. Example: rather than use exact 5MHz for your clock signal, you may use 4.99MHz. And if there is a CDC from this clock to another clock of 10MHz then you will not have a fixed relationship between the two clocks. Rather the clocks will keep moving in phase with respect to one another. The diagram here shows two clocks of different frequencies with relative phase drifting apart slowly. You can see that successive rising edges are separated by greater and greater time.

With these real world errors added to clock frequencies, if you simulate your design for a long time there is a good chance of spotting many CDC issues.

Gate Level Simulation

GLS refers to simulating not at the RTL level but at the level of logic gates. GLS is both time consuming to setup and time consuming to run. It needs a different mindset than regular RTL or behavioral simulation. So, a separate section is devoted here. For example, if a "+" operator is used in RTL for any addition from 1 bit to any number of bits, GLS will simulate a design that uses AND/OR/NOT/Half Adder and other gates collectively implementing the addition operation. The logical function of the design is supplied to the simulator in at least two parts. The bigger part is the netlist and the second part is the behavioral models of the logic gates. A synthesized gate level netlist is considerably bigger than its parent RTL. The run time of a GLS is much higher than RTL simulation. GLS is mostly run with delay information for every gate and net. This also contributes to the much slower simulation speed. In RTL simulation, a synchronizer decisively samples a 1 or a 0 and never goes to "x". In other words, RTL simulation can be said to use 0 setup and hold time for flops, completely ignoring metastability effects. In GLS, both setup and hold timing checks are defined in the SDF for all flops in the design. Whenever the sampling of data occurs within the no change window between setup and hold, the simulator sets the stored value of the flop to x and this is what is output on Q pin. This x could cause many modules using this signal to get corrupted and the whole simulation becomes useless. Suppose, an FSM was being driven by the synchronized signal, a momentary x input to the FSM permanently corrupts the state to x. To avoid this problem, all synchronizer first flops are set in GLS to ignore the setup and hold check. This is achieved in either of two ways. The first is to use the simulator commands to disable timing checks on select instances. The second is to have an override SDF file exclusively for the synchronizer flops with their setup and hold check zeroed out. The overriding SDF approach is a bit tedious and hence simulator command based method is preferred.

Synthesis tool also badly distorts the RTL structure that it is difficult to work with the netlist. Synthesis tool does the following modifications:

Uniquification

If your design had 2 instances of a module called my_mod, the tool would output two modules my_mod and my_mod_0 in the gate netlist. Every instance in the gate netlist has a unique parent module. The tool does this to independently optimize the instances. But, this makes debugging and navigating the netlist a major tedium.

DFT Logic

Design for test logic is inserted at synthesis stage and it adds a lot of clutter to the netlist. Care should be taken in GLS to fully suppress DFT logic for functional mode GLS simulation. The precise logic added are scan chain flops and nets, scan compression logic, if any, reset and clock gating override logic, high speed and scan clock muxing logic, built-in self test logic, if any, and DFT ports of test enable, scan enable, scan input and scan output.

Port punching

Signal trees in the gate netlist do not follow the RTL connection style of one net reaching all end points. If you take the reset signal for example, in RTL it is only one signal reaching all the flops in the code. But in reality, one driver buffer gate cannot serve more than a few tens or hundreds of flops. So a tree is built for the reset signal with each leaf level buffer driving a few ten flops. RTL does not reflect this information. The synthesis tool when it outputs the gate netlist, makes up new reset ports to reflect the tree style connectivity. So, in the netlist you may have reset_0, reset_11 etc.. as inputs to the same module, each serving a group of flops. This makes reading the netlist very confusing. Clock is another signal that is also made into a tree.

Tool Generated Modules

Sometimes the tool could insert its own modules to implement complex functions. The names of the tool instantiated blocks are generally much less understandable. For example, if you used the power operation in your RTL as y= x**3, the tool could instantiate a module something like dsp_pwr_k1. What is happening is that the tool rather than directly implement the power operation with basic logic gates, inserts an intermediate high level implementation of the power operation from its library. Later, it synthesizes this intermediate module with the basic gates. This also makes reading the netlist difficult.

Negative Delays in SDF

Some timing arcs in the SDF may have a negative delay. Though negative delay makes no sense because an effect cannot precede cause, in simulation negative delay has some meaning. I am not able to understand that reason well so far. You will have to live with negative delay for GLS. The simulator may truncate negative delays to zero and cause timing violations to appear in the GLS. The STA reports will have no timing problems but GLS will show timing violations. The correct way to deal with this is to use specific options of the simulator for GLS. Another way is to tweak the standard cell .lib models to truncate the negative delay numbers in setup and hold values to zero. The place and route tool will optimize for the zero delay number from the library and produce output design with excess margin than desired. This is because a negative setup and hold time means a relaxed

requirement than a zero setup or hold requirement. But STA should be run with unmodified .lib and it will export an SDF that may have negative delays. Now that the design has margin covering the simulator's truncation effect, the design will simulate correctly in GLS.

Initialization

In RTL, the names used to initialize feedback circuits like oscillators and clock generators are easy to obtain. Many times, the RTL simulation even works with incomplete initialization. In GLS, it happens often that some of these instances are forgotten to be initialized or the incomplete initialization is insufficient and the simulation is stuck in x.

Value added by GLS

Many companies don't run GLS for every RTL level test. This is because it is not possible to do GLS for the full test suite. It is also not needed to do GLS for all tests. A few selected tests run at GLS level is very useful as a double check of timing constraints of the design. If your STA owner missed some timing constraints, the GLS test would fail and show the timing problem.

Zero or tiny delay GLS

Using full SDF may be time consuming and difficult to setup. An intermediate step is to run the gate level netlist without SDF. A no delay GLS may not work because some timing checks may fail. So a trick is to add tiny delays like 1 picosecond to the output of every gate in the Verilog model of every logic gate in the design. This version simulates faster than full SDF annotated simulation and may also serve as a step to check if SDF is causing the GLS to fail or not.

Additional concepts related to GLS
https://www.linkedin.com/pulse/gate-level-simulation-comprehensive-overview-jerry-mcgoveran

Dan Joyce's 29 tips for gate-level simulation - DeepChip

https://www.deepchip.com/items/0569-03.html

Reset check

Reset signal initializes your system. Reset, like clock, needs careful design. A badly initialized design would cause strange malfunction when the product actually works in the customer's place. Suppose you reset your phone to do a clean firmware update and your phone just hangs after reset. How annoyed would you be?

A synchronizer is used to create a reset signal that has a fixed timing relationship to the rest of the signals in the design. Or, more technically to transfer an asynchronous reset from outside into the synchronous clock domain of the design. There are many more aspects to reset check than this. But for now knowing about the reset synchronizer is sufficient to deal with most of the reset issues.

This paper provides useful insights into the usage of reset.

Asynchronous & Synchronous Reset, Design Techniques - Part Deux, Clifford E. Cummings, Don Mills, Steve Golson, SNUG-Boston 2003

http://www.sunburst-design.com/papers/CummingsSNUG2003Boston_Resets.pdf

DRC and LVS

Design Rule Check answers the question, "Is my layout mass manufacturable?" The layout of a digital circuit realizes the logical functionality in physical silicon. There are many restrictions on what the fab can reliably manufacture. For example, a circle shaped transistor is not manufacturable in most fabs. Transistors or wires too close to each other are not manufacturable. The DRC check accepts the layout of the design as input and outputs a list of design rule violations. If there are no violations then the layout is manufacturable with very good yield. It gets interesting when there are indeed DRC violations. Some violations are outright not manufacturable. For example, wires too close may short 100% of the time. Some other violations reduce the yield but are not complete show stoppers. Sometimes, it is ok to waive (or ignore) these DRC violations for prototyping your chip in silicon. For most DRC errors, the layout team or physical design team root causes the problem and fixes the violations.

https://en.wikipedia.org/wiki/Design_rule_checking

Layout Versus Schematic checks for the functional equivalence of a layout to its schematic. In the case of digital design, the schematic is equivalent to a gate netlist. What really happens in LVS is an EDA software tool extracts the device level connectivity from analyzing the layout shapes. Here, a device denotes any of transistor, capacitor, diode, resistor or inductor. Note that a wire is not a device. It is merely a connection. Every gate in the gate netlist has a device level circuit netlist supplied by the gate library vendor. Now the LVS tool has the layout extracted device netlist on one hand and original gate netlist on the other hand. It compares the two net for net and device for device and reports any mismatches. If there are no mismatches your layout is logically same as your gate netlist. Note the uncanny resemblance of LVS to LEC that checks if RTL is same as gate netlist.

https://en.wikipedia.org/wiki/Layout_Versus_Schematic

For the purpose of RTL design, it is sufficient to know the overview of DRC and LVS. The details of DRC and LVS are beyond the scope of an RTL chipper.

Low Power Design

Low power design has become a catch-all term covering designs that are are power and energy limited. For example, a smartphone with a battery is both energy and power limited. If the phone electronics where to use too much power it would make it too hot to touch, though there could still be charge in the battery. If the chips waste small amount of power but at a steady pace then the battery may drain in a few hours frustrating the user. Power budget is even more stringent for devices that cannot be recharged easily like an implantable pace maker. Energy harvesting devices are another interesting

area. These devices capture tiny amounts of energy available in the environment and collect it steadily until enough energy is accumulated so that some operation can be performed. Because the energy available is so tiny, the circuits have to be super efficient at using them.

To design a chip for low power and low energy consumption, you need to understand how power is consumed in the first place.

Static power

When a circuit is not operating it still consumes power. This is called static power. It is sometimes interchangeably used with leakage power. Have you noticed that a TV or phone charger even if not actively used still remains warm?

Dynamic power

When the circuit is operating, current keeps pulsing through the metal wires inside. Every wire has a small but non-zero resistance and so dissipates power with current passing through it. Note that only the wire dissipates power but the capacitance effectively connected to the wire actually dictates the amount of current need to create a full digital 0 or digital 1 on the appropriate gate inputs. MOSFETs also have non-zero resistance and they dissipate power too.

There are standard techniques to design a chip to operate at low power.

Multi Vt

Threshold voltage of a transistor indicates how much it leaks current when not working. But leaky transistor are generally faster transistors too. So, multi Vt design uses low Vt or fast transistors in the critical path of a circuit and high Vt or slow and less leaky transistors on the non critical paths. Note that transistors and their threshold voltages are not specified in RTL. That job is left to the synthesis tool.

Reduced switching

The dynamic power depends on how much switching happens in the logic. As a chipper, you should reduce the switching in all signals. This is achieved by gating clocks off for logic blocks that are needed to be active for many clock cycles. Other way is to use more parallel logic rather than a fast logic at higher speed. You can also study your implementation and re-architect it to reduce signal switching.

Reduced capacitance

It is the capacitance of wires and transistors that need large currents for operation. Capacitance kills efficiency. So, going to smaller and smaller transistors that need lesser and lesser current to switch their capacitance to full 0 or 1 is driving the trend to go to smaller process nodes. RTL level and circuit techniques are also useful to reduce the amount of effective capacitance getting used in the logic.

Reduced Voltage

The formula for power dissipation to a first approximation has a term with supply voltage squared. So, reducing the supply voltage and keeping the performance same increases efficiency.

Power Intent

How the design should behave power-wise is called the power intent. Note that SV was not meant to specify the power intent of a design. So, additional files in the format of Unified Power Format or Common Power Format are used along with the SV RTL for the design. The UPF file is something like the timing constraints but it deals with power aspects alone. For example, it would say which parts of the design operate in which voltage and which sections are continuously powered on and which switch on and off during operation.

https://en.wikipedia.org/wiki/Unified_Power_Format

https://en.wikipedia.org/wiki/Common_Power_Format

Low power design is an extensive topic. Please refer to these resources for further reading.

Low Power Methodology Manual: For System-on-Chip Design, 1st Edition by David Flynn, Robert Aitken, Alan Gibbons, Kaijian Shi, Michael Keating

Low Power CMOS Digital Design, Anantha P Chandrakasan et al, IEEE JSSC, 1992

Bug Tracking

Bugs happen no matter how best you may have checked your design! The volume of bugs that need to be tracked increases drastically with the number of designs and customers supported. For example, you may have released a chip with version 1.2 of your RISC-V processor core. For another customer you may have released a similar chip with version 1.5 of RISC-V core. Version 1.5 may have a new feature, say, DSP instructions. Suppose, there is a bug in the ALU that is common to all cores. A bug tracking software allows which versions and releases are affected and who is the owner to fix the bug in each version. The software also brings some level of order to an otherwise chaotic bug fixing process. Bugzilla is a free bug tracking software from Mozilla. It is a bit dated but good enough.

https://www.bugzilla.org/

JIRA from Atlassian is more modern bug tracking software.

https://www.atlassian.com/software/jira/bug-tracking

Coding Guidelines

Naming Convention

SV syntax permits any name to be given to modules, signals, functions and tasks. You could call a video frame buffer by the name b1x_xzk_09. Does it convey any value to the reader? Of course not!

You could name it frm_buf. Is it useful? Mostly yes! Could you name it video_frame_buffer? Well yes! Which name is a good name? Questions like these have no answers. So, the boss steps in and defines some guiding principles for naming SV variables. The coding guidelines help to create code that is understood by all the members of the organization. Think of this as the local dialect or slang of SV spoken by your company.

For example, a naming convention may say -

Inverted signals to be named with _n suffix. Example: signal en's inverse to be named en_n.

Clock signals should be suffixed with _clk. Example: processor clock may be called p_clk.

Opencores has a coding guidelines document that can give you some sense of these conventions.

https://cdn.opencores.org/downloads/opencores_coding_guidelines.pdf

I have a few conventions:

Constants are in upper case. Example : parameter **CLK_FREQUENCY**=100;

Variables are in lower case. Example: logic **product**;

No mixed case. Example: Main_Select, MainSelect are not allowed

Not too short names. Example: Suppose you name a primary input clock of your design as pc, it is too short and does not immediately inform the reader of any meaning. This is because pc is not generally used to refer to input clocks. Rather a name like pclk or pclk_in or master_clock or prim_clk is more understandable.

Not too long names: Example: Suppose you have a signal that is a rounded value of the product of two floating point variables a and b. Say you named the signal rounded_out_a_and_b. The name is super readable but then every time you need to type this, it takes many key strokes to type directly or you always have to use the auto complete features that adds a few keystrokes. Imagine the name of another signal that is AND reduced of this signal, are you going to name it and_reduced_rounded_out_a_and_b? There is no clear cut length for names. So, use your judgment to name signals not too long or not too short.

Misleading names not allowed. Example: carry output of an adder cannot be named d_out.

Naming conventions are not followed that strictly in practice. So, if specific names in your design look ugly because you have to follow the naming convention, go ahead and violate the convention to make the design more readable.

Sequential logic coding style

I like to separate the flip flop providing timing synchronization from the logical function that is realized by combinational logic. So, I use two pieces to describe a sequential logic, inspired by the guidelines for FSM coding that calls for separating the state flops from state transition logic. I find that separating out the combinational logic into always_comb block frees me from the clock related timing of

always_ff block and lets me focus on the logic. The code inside an always_comb is very intuitive to understand because it executes serially just like a C, C++ or Python code.

Conventional style

```
always_ff @(posedge clk, negedge rstn)
  if ( !rstn )
     seq_reg <= 0;
  else
    if ( <condition1>)
      if ( <condition2> && !<condition3> )
        seq_reg <= sig1;
      else
        seq_reg <= sig2;
```

My preference

```
always_ff @(posedge clk, negedge rstn)
  if ( !rstn )
     seq_reg <= 0;
  else
     seq_reg <= seq_next;

always_comb begin
  seq_next = seq_reg;
  if ( <condition1>)
    if ( <condition2> && !<condition3> )
       seq_next = sig1;
    else
       seq_next = sig2;
end
```

Automation

Though the world has moved away from heavy industries as the engine of economic growth, we are still caught in the need to produce things ever faster. The faster you can make working designs, the higher the profit for your company. So, automating all your work is the key to success as a chipper. The EDA companies have standardized one language for automating the usage of their tools. It is the simple and effective Tool Command Language TCL.

Because you will have to work a lot in the Linux environment, you need to have a working knowledge of any Linux Shell – Bash, Kshell, Cshell, Turbo Cshell.

PERL and Python are also extensively used in many companies to create a higher level of flows using the bare tool commands from the EDA vendors

Regular expressions are a kind of language in itself. It is extremely useful in parsing files and finding specific patterns and then to modify those patterns. Regexp turbo charges your Shell/Perl/Python/TCL scripts.

GNU Make is used to string together many varied tools and steps into a neat flow.

Exercise

The accompanying Ehgu library does not yet have an automatic full library pass-fail checking script. Can you create the pass-fail checking verification setup for at least three modules or functions? Suggestion: Use separate directories for each test and combine the result using GNU make tool.

ASIC Tapeout Process

Emotional feel of tapeout

Imagine you sign a check to pay for a venture that is 10x your yearly income. Can you feel the fear? Now, imagine that you are going to earn 15x or more of your annual income if the venture you just paid for succeeds! Can you feel the excitement? This is what tapeout possibly feels like to your manager. I wonder if this is gambling addiction masquerading as high tech science:) May be this is what emotionally sustains chip company executives.

For a few products, I have been at the second level to my boss. On my word, the tapeout would happen or be stalled until further work is completed. I did not have the signing authority but my boss relied on me to make the right call that the design was verified enough and that the chip would succeed on first manufacturing? In chipping parlance, the first manufacturing is called first pass. I could indirectly feel the anxiety of bosses who have to sign the tapeout bill. My team was small, we had only four people responsible for a complex first time design. So, I did all the industry recommended checks and also added a few more checks I could think of. At the end of the day, I still felt the tension but now I could sleep peacefully knowing that I completed all the steps known to make the chip successful. People are motivated by different things - money, glory, doing good, helping friends and family. Most people are also motivated to not let down their friends. So, verify your design well, at least for your boss and for a good night's sleep!

What is actually the tapeout process?

Tapeout means to send the chip layout information to the fab for manufacturing. Because this involves a lot of money and effort, there is quite a bit of security and reliability aspect to it. The chip layout comes together from different groups of the company. For the synthesized digital part, a physical design team creates the final layout. For custom digital part and analog part, layout engineers completes the final layout. The circuit designer also checks if the layout is matching the schematic using LVS. A full chip lead checks if the layout meets all fab design rules. Once the layout passes all checks, it is now ready for sending it to the fab. Your boss now logs into the website of the fab and selects the settings for manufacturing this chip. For example, a high volume small chip could use 1 poly layer and 4 metal layers from 130nm process. Your boss would check the check boxes for poly1, metal1 through metal4 for the 130nm process. He would leave the boxes metal5-metal8 unchecked. For a medium volume high performance processor, the chip would probably use many more metal layers, say, up to metal10 from the 28 nanometer process node. Obviously, there are many more layers than this example. There are many more settings to select as well. Typically, every layer that maps to a

physical layer costs money. There are layers that contain information but do not get manufactured into a physical shape. The lower layers cost more to make than the higher layers. This impacts how bugs in the chip are fixed after the masks are made.

The layout is encrypted using some encryption method. I have seen PGP being used. This is to prevent unauthorized people getting access to your chip layout. The layout is a symbol of the immense effort and money invested by your company into this chip. If a competitor steals this, he would gain an unfair advantage without having spent the investment to create the chip.

After the boss is happy with the quality, he would upload the layout file to the fab using FTP. A checksum for the encrypted layout file may be sent to the fab as well, I am not sure. The purpose of the checksum is to make sure the layout file reached the fab without errors. The fab would take the file and recalculate the checksum and compare against the one received from the company. If the checksum matches, there is a super high probability that the chip layout was received without errors. The layout is decrypted by the fab. The fab runs the DRC again. If the fab finds any issues with the chip layout, the fab feeds back to the company. If there are no issues the fab uploads the layout into its own EDA layout software. The fab provides a temporary access to the layout viewable in the EDA software. This is called a Job View or Job Description View. Once, I was asked to check the job view. I had no clue what to do. Then my boss told me the industry tradition - "look for the last change". This is a simple and powerful check to ensure only the latest version is being taped out. This would catch one of the common mistakes in chip design - "taping out an older version". Suppose, you added a piece of extra metal to layer metal3 at the top right corner of the chip just before the final layout, then look for this shape in the job view. After you acknowledge that the chip looks OK in the fab's job view, the fab starts cutting the masks for every physical layer. This takes a few days. Every mask made is money spent. The check your boss signed gets spent one mask at a time in just a few days after the layout file was sent to the fab. So, you have up to a few days after tapeout to make corrections to your design.

Now you have done your best! Go home, take a vacation with your family :) Come back in a few weeks to meet your finished chip.

https://en.wikipedia.org/wiki/Tape-out

The above process is for a fabless semiconductor company. If your company has its own fab, some conditions change slightly. The cost of tapeout would be slightly lower than contracting it out to a foundry, though, it would still be super expensive. There is no need for encryption because layout file going to the fab will be entirely within company network.

Bring up and debugging in the lab

After your chip is manufactured in the fab for the first time, it is sent back to your company for testing and validation. Now, its time for real electrical topics. The chip will need a power supply, it will get hot, the board may smoke and most commonly you are left with some weird malfunction that is inexplicable. The process of putting a new chip through its paces and finding out the correct settings to make it work as specified is commonly called "bring up". I wonder it has any relationship to bringing

up a baby into a functioning adult. So whats the role of a chipper here? Well, the job of validating a chip is shared by people with different skills. First there is a lab engineer who is real electrical guy and good with lab equipment, running tests for many chips and collecting data. The second player is you, the chipper. You supply the details specific to this chip under test in the lab. Suppose your design does not work at maximum speed. The first thing to try is to increase the supply voltage. The key idea for debugging in the lab is to create special experiments and pin point the cause of a malfunction by using the knowledge about the design you have coded. A second and even more useful skill is to workaround malfunctions and save your company from having to spend a lot of money doing a second tapeout. A workaround is similar to a software patch. For example, suppose the malfunction is actually a bug in the architecture, say, like the Meltdown and Specter vulnerabilities in many modern processors, then the workaround could be a modified firmware that reduces the weakness rather than redo the chip. Another example could be a PLL that does not work at a particular frequency and voltage. If the PLL is found to work at the same frequency at a lower or higher voltage, you can recommend this option to your customers. If they are OK with that then the problem is almost solved for now.

Every problem found in the lab should ultimately be root caused and fixed in the next version of the chip. Why? Because a problem that is not understood most likely hides a deeper bigger problem likely to recur in future versions.

Process, voltage and temperature, PVT, are the primary variations a design has to deal with. It is also simplest to change and see how your design responds. For example, you can check your designs operation in lower temperature if the design fails at higher temperature. You can also ask your fab to supply you all slow or all fast or all typical corner parts for testing. A shmoo plot is convenient way to visualize the changes in design operation against the independent variables of PVT.

https://en.wikipedia.org/wiki/Shmoo_plot

Another powerful trick is to recreate the same stimuli in simulation that the chip sees in the lab when it fails. You may wonder how this may be different from a verification test. It may not be too different but a slight difference to the simulation conditions may be hitting a bug that the verification did not model or did not cover.

Validation

You can see that the chip design has to be checked repeatedly. Validation is also one such check. It may be confusing as to how this is different from verification and testing. Verification is used to mean pre-silicon checking of the design. Testing refers to testing the silicon for manufacturing defects. Testing does not bother with checking for functional bugs, but, verification does. Validation is also for functional testing but in contrast to verification, it is performed on the actual chip and not its simulation model. There are some good reasons to do validation. The execution time for a test is the fastest you can get in real silicon compared to simulation. You can also input real world system data together with real world system settings. Suppose your chip is a high precision clock oscillator programmable through I3C, in validation, you will connect an actual microcontroller via I3C to setup the oscillator

and also connect a precision oscilloscope to check the output clock properties. In verification, it will be difficult to do this. Even if done, the oscillator properties are hard to model in simulation. This is because even with the best transistor models, process technology still leaves many parameters less well modeled. So, the chip has to still be checked experimentally. The easiest way to validate is to take the verification test suite and try to do all the tests in silicon. Note that the verification tests may not be directly applied to the silicon because you cannot access internal wires of the silicon chip. The validation setup may also not read and run SV verification tests. So, you will have to port the tests to scripts that your validation test setup will accept.

Characterization

Not all parameters of a chip are quantified in binary, pass-fail. Many are analog values. Using the same precision I3C programmable oscillator chip example, suppose say you have specified the output clock frequency error as 100ppm. Before starting to produce millions of chips, you are interested to know if a small batch produced has sufficiently low variation and within spec for all the parameters. In the frequency case, you want know the standard deviation of frequency error for say 10 chips or 100 chips across PVT. If the standard deviation comes out to only 25ppm then you are fine. Because when you produce millions of chips, your design may still have an error of only 90ppm. Characterization is a super labor intensive work because you have to repeat many tests on many PVT corners on many chips. Assume you have 10 parameters of interest and 3 process corners, 3 voltage corners and 3 temperature corners and you want to test for 20 chips, the number of tests to be run are 10 * 3 * 3 * 3 * 20 = 1800!!! Characterization part of the chip industry is ripe for disruption!

Datasheet Management

The interface between the chip and the customer is the datasheet. It is plain natural language like English or Chinese or any other. It is also heavy on electrical data like how much power a chip is expected to consume. What are the voltages to run the chip properly and so on. What is less obvious is that the content in the datasheet is like a golden reference for the chip to meet. This is because the customer is not going to redo all the characterization on the chip to derive the parameters that is of interest to the customer. The customer will simply assume that the chip will behave as specified in the datasheet. The implication for a chipper is that he or she has to read the datasheet and make sure that his design meets the description in the datasheet. This is actually a tough task. It is prudent to check the verification coverage against all the features listed in the datasheet.

Register Management
A chip has many control and status registers. These are used to configure the chip for the user's specific operational needs. Every bit in the register has a description in the register description section either in the datasheet or in a separate document. The difficulty in maintaining the registers of a chip is that the design intent and the description in the document become disconnected. Designers are too focused on the design and often less on the description of registers. So, the wordings in the register section is frequently misunderstood by the customers. Or worse, the designer coded something in RTL and wrote something else in the register description. The changes in the features of the chip also make register management difficult. In large chips there could be 1000s of registers. It is a hard problem to just to

make sure the RTL and the description match for all of them. Companies regularly use software tools to export RTL code of the flip flops of the registers from a version controlled register description file. The same register description file forms the basis for the datasheet or register description document that goes to the customer. This way errors in manually copying the designer intent for the register into the customer facing document is minimized.

How should the documentation look like?

Readers have varying level of attention span. Some are put off by a 5 page document while others are put off by a 1000 page document. My tolerance is in the range of 100 pages. So, it is preferable. to have the document structured in two or more levels of detail. The first level is just one page, suitable for executives and customers. The second page is for engineers who are going to use the design more intimately. Third and subsequent levels are optional.

Spec

What documents to make is not very standardized yet. Most agree that a specification document (spec for short) is needed for every design. One example is here -

https://cdn.opencores.org/downloads/specification_template.dot

You may look at other examples from standards documents like I3C, I2S, RISC-V etc. The basic idea is to describe what your design will do and not focus on details about how the design will do its job.

Implementation

I like to also create a document that shows the neat tricks used in the RTL code. This one includes the details of how a particular spec is achieved in the design. This may be combined with the spec for small designs. The idea for this document is to record information about the design that is not very obvious from reading the code. Explaining every line of the code is not the idea. This document is exclusively for other designers, same company or open source chippers.

Extra documents

I found that few additional documents are also useful. These may be combined with spec or implementation if needed – timing constraints, guide for lab testing, simulation debug guide and customer facing brochures or flyers. Remember to work with verification engineers to make a verification plan as well.

Engineering Change Order

No matter what quality control process an organization follows, mistakes are unavoidable. When mistakes happen in a silicon chip, the first attempt to fix it is by careful surgery. This is because it costs less to make small changes to a chip than to completely redo the chip. The small changes made to a chip is call Engineering Change Order or ECO.

Post-tapeout or post-silicon ECO

ECO by default usually means that we intend to change the chip after it has been sent to the fab for manufacturing. To differential this from other types of ECOs, you may want to call it post-Si or post tapeout ECO. Money has already been spent on cutting the masks for all the physical layers. So, changes to the chip that modifies many physical layers will cost more than another change that touches only a few physical layers. So, post-Si ECO attempts to minimize the number of layers modified. Note that if most of the layers need to be changed to get implement the ECO it is easier to redo the chip by making all masks again.

Still there is a reason you may want to do an all layer ECO and not a rework of the chip. This is because an ECO by definition changes only what is really needed to change, so you can be confident that no other functionality of the chip gets broken because of this change. If you redo the chip again there is a possibility that existing functionality could be broken while making the change. This is especially true of analog circuits that have a high dependence on layout.

Metal only ECO

Base layers are the layers that close to the semiconductor devices. These are the layers used to make the transistors. In the context of digital design, the logic gates are the only things in the base layers. It is costlier to change the base layer masks than it is to change higher layers. The higher layers contain only routing in the form of metal wires. As the layer level goes higher and higher, the shapes of the layer become bigger and bigger. The bigger shapes of the layout are easier to make into a mask than the fine geometries of the base layers. So, the metal layer ECOs are cheaper.

Pre-tapeout or Pre-silicon ECO

It is almost as difficult to change a design right before tapeout as it is to change after tapeout. This is because changes to the design no matter at what stage needs thorough verification. In pre-silicon or pre-tapeout ECO there is some more flexibility because the no money has been spent to cut physical layer masks. So, pre-Si ECO can choose to use all layers to realize a change.

Standard cells for ECOs

Spare cells

Like how an automobile may have a spare tire always carried inside, chips mostly carry unused logic gates instantiated in the layout. When the real ECO is needed these spare cells may be used to realize the logic change.

ECO gates

There are specific cells that are designed to be generic logic cells. This approach is smarter than just blindly keeping copies of functional logic gates. For example, rather than keep a few thousand copies each of NAND, NOR, MUX, XOR and flops, you need to only keep a few thousand copies of generic ECO gates of different sizes, say, 5 units, 10 units and 15 units. The NAND/NOR may be realized in the 5 or 10 unit ECO cells and Flops may be realized only from the 15 unit ECO cells because flops need more area than simple combinational gates. In the spare cell approach you can only use the

NAND spare gate for NAND operation only. In the ECO gate approach, you could convert the 10 unit ECO cell into any of NAND/NOR/MUX/XOR. The ECO cells approach is more area efficient for the same amount of ECO flexibility added.

All the possible combination of logic gates that can be realized from the 5,10 and 15 unit ECO cells are characterized before hand so that the timing analysis tools can accurately check the chip timing including the ECO cells that are now incorporated into functional logic. Suppose, the 5 unit cell gives NAND/NOR/MUX/XOR, all these resultant cells may be named as NAND1ECO5/NOR1ECO5/MUX1ECO5/XOR1ECO5 and have their timing information available in a library.

https://www.edn.com/eco-friendly-standard-cell-design-with-a-dual-purpose/

https://en.wikipedia.org/wiki/Engineering_change_order#Chip_design

What does a chipper job look like?

Actually my experience is limited. I am presenting my view anyway. Other professionals may have a different experience.

ASIC Design Job

You participate in meetings to define the design. Create initial versions of the design. The design could be from scratch in which case you will have to study textbooks or other existing code. Or, the design could be a port from an older project. Then you will have to understand code from another person or another time. In any case, reading a lot of others code is part of the job. Create design level simple testbenches. Create specifications to talk to verification team. Fix bugs reported by verification. Help FPGA engineer validate the design. Participate in project planning meetings to update design progress or make excuses for the lack of progress! Define timing constraints and help synthesis and physical design. Document your design. Present your design to experts for a review. Make last minute changes to the design as ECOs. After the chip is back, spend hours in the lab to debug anomalies. If a bug is found spend time creating a workaround and if possible create an ECO that fixes the bug at lowest cost. Help application engineers answer questions from customers. Participate in demos to the customer. Create automation to make your job less stressful with scripts. Evaluate new software tools and processes. Enjoy your free paid time off when IT network crashes :) Fill out year end self appraisals. Fill out monthly or quarterly job targets. Attend a few conferences and attend online learning courses. Interview candidates for a coworker position in your company (some may end up becoming lifelong friends), look for jobs in other companies for higher pay or better work or better boss or better team or all, and do all this over and over again and earn salary and promotion.

FPGA

I have not done an FPGA chipper job for a long time. But I guess that it looks very much like the ASIC chipper job. The exceptions are that FPGAs are easier to modify than ASIC silicon, the testing for the

FPGA is already completed by the FPGA vendor, designs are more complicated because FPGAs offers many pre-built IPs.

How does a chip company look like?

A company dedicated to making chips can be organized in anyway. Most, however, are organized into standard teams. At the top is CEO. It may surprise you that not all CEOs are chippers. Many do come from as unrelated background as finance. There could be a CTO, Senior Vice President, design directors and then managers. Exceptionally talented engineers are made into fellows. The name fellow sounded weird to me. But it actually means extremely skilled or knowledgeable engineer. The engineering teams are split into digital design, verification, analog design, layout, synthesis and place and route (SPNR), timing analysis. There is usually a CAD group managing the setup and maintenance of EDA tools used by all the other engineering teams. There is a related IT team that manages the general software used by the company from Windows desktops/laptops, servers, applications running in the desktops/laptops. Contrast the role of IT in maintaining general software as against the CAD team that maintains chipping specific software. CAD and IT have extensive overlap. IT is also tasked with cybersecurity for the company. This role is increasingly becoming important to chippers because hacking and the actions taken to protect against hacking drastically limit the productivity of designers. The layout team takes care of creating the layouts for analog and custom digital circuits. Note that digital design layout is done by the SPNR people. There could be a software group creating software that is delivered along with the chip. For example, chips with a programmable processor core need software engineers who create software that maximizes the use of the chip. There is a DFT group that creates scan test patterns for the chip that is used during manufacturing to detect faulty chips. The DFT group also simulates the patterns using logic simulation to check if everything is ok for usage on actual silicon. A test team uses the scan patterns from DFT group during manufacturing and runs the patterns in a tester machine that checks every chip. Note that the difference between test and DFT is that test group is dealing with real silicon and DFT group is dealing with the Verilog model of the chip. A characterization group takes many chips and obtains the distribution of chip parameters over a number of chips. For a fabless semiconductor company, a foundry manager is in charge of communications with the foundry. He gets the updates from the foundry, communicates the design rule waivers or special needs of the chip to the foundry. A product development group defines new products that may be profitable to make. The engineering teams frequently have one or more lead engineers with good technical experience leading the project, along with a few lesser experienced engineers. The lead reports to a manager who is also a technical person generally, though, he could also be a purely people manager. Shepherding all these people is the Human Resources group. For some reason, engineers hate HR, but the role of HR is indispensable to a well functioning company.

In smaller companies, the design and verification teams may be merged and every designer will also be a verification engineer. The SPNR and STA can be merged too. In a small company, I have actually done from RTL, verification, SPNR through STA, LVS, DRC and CAD.

Protecting Your IP

Open source supporters would recoil at the thought of protecting code! Protection for intellectual property is big business. I have done it many times too. The reason is that even if you agree that open source is good for the world, closed source is an inescapable reality in chipping, just like most houses have a lock in the door. You may have bought an IP from a vendor that grants you rights to use it for a production chip. But it will certainly not grant you the right to share it with others. So, protection becomes important when you are dealing with others' property. IP protection is the foundation that allows design houses to share their designs with chip companies without having to worry if the chip company will re-sell or customize the IP without paying the design house. Most of the chip companies would also not want to give away their latest and greatest IP for free before making a good return on the investment. After all, it takes many thousands of dollars to create a fully verified design. If you are one of the employees in a typical chip company you have the obligation to protect the code from being stolen. Let me share with you the many ways IP is protected.

Obfuscation

A particular piece of code was causing the simulator to crash. I reported this problem to the EDA simulator vendor company. They asked for a testcase with the problem code. I was not sure, if I can share the design as is with the EDA company. This is because the code is the property of my company and I have the duty to protect it. So, I reduced the design to only the problem code and also changed the names of all the nets and ports to unrecognizable names by searching and replacing the names. I talked to my software guru in a casual conversation and he told me there are ready made programs called obfuscators that do this. I tried finding one for SV but there were none.

For another crash problem, I did the same thing. This time I asked the EDA company if they have any obfuscator bundled with their software suite. They actually had one! It was working good. So, next time you want to obfuscate, ask your EDA vendors if they have one. I found this page from EDA company Aldec describing obfuscation in some detail.

https://www.aldec.com/en/support/resources/documentation/articles/1586

If you have no access any ready made obfuscator, try creating a script based on the obfuscator pseudocode in the software utilities section of this book.

Netlist only

For one of my chips, we got an IP from a vendor. The IP was just a netlist with weird names like xyzer2, wxyg, thsr. The IP would simulate and synthesize well but we could not modify that. The gate netlist synthesized from the RTL is less readable than the parent RTL. This is because the meta information in the RTL like comments, specific ordering of ports, indentation are lost. All the high level constructs are also lost and replaced with low level gate representations. For example, a=b*c would be replaced by so many gates that it would be hard to recognize that it only a simple multiplication that is computed by the gates. The netlist can also be a flat version. Flat here means there

is no hierarchy in the design. Note that hierarchy of a design conveys a lot of information by breaking the design into pieces that a chipper can understand.

Encryption

For one design, I had to share my design with a third party FPGA design house. My design included an IP from a vendor, so before giving it to the FPGA house, I encrypted the vendor's IP. Encryption is the silver bullet for protecting your IP. It can do all the things that obfusction and netlist can do and be a lot more effective at that. There are many options in encryption. The easiest way is to read the HDL files into your simulator and export an encrypted version of the files. If you want to have only some code to be encrypted, you can enclose that specific code between the pragmas - `protect … `endprotect. Note that encryption is typically tool specific. If the party that is going to use the encrypted design for say Synopsys VCS, then you may have to encrypt using Synopsys VCS. Chapter 6 of the Xilinx Vivado document "User Guide - Creating and Packaging Custom IP UG1118 (v2017.1) April 25, 2017UG1118 (v2017.2) June 7, 2017" explains the encryption process very well.

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug1118-vivado-creating-packaging-custom-ip.pdf

Encryption is so important to the industry that there is even an IEEE standard for this.

IEEE 1735-2014 - IEEE Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP)

https://standards.IEEE.org/standard/1735-2014.html

Reverse Engineering

To understand reverse engineering, we need to understand forward engineering. A chip or a design proceeds from market need or inventor idea to architecture to RTL to netlist to layout and finally finished chip. Along the way, many documents are also created to make sense out of the many formats modeling the chip. Rarely, there is a need to go backwards in the design process. For example, there could be a finished chip with RTL but no documentation. This could happen if the design was done by only one or few chippers who did not keep good documents describing the design and then they either quit or where laid off. You may be asked to maintain this design or worse fix a bug in it. In this case, you may have to study the RTL, run tests and then create fresh documents describing the design.

There may be a situation that the vendor of an encrypted IP being used in your chip goes out of business. The vendor does not provide the source code for the IP before getting shut down. The encrypted RTL will simulate but you cannot understand the internals or modify the code. Your only solution now is to develop a pin compatible replacement from scratch that behaves like the encrypted RTL when simulated.

Sometimes, only a netlist may be available from a 20 year old design and you may have to port it to a new process. The need for reverse engineering are many, but the idea is simple, to go backwards in the

regular design process. There is no simple or direct formula to reverse engineer and it is not even taught in any degree program. Occasionally, reverse engineering is super useful!

The company TechInsights specializing in reverse engineering.

https://www.techinsights.com/

Standards Organizations

Chip design ecosystem has many players. Competition and cooperation abound. There are small fishes and there are sharks. Groups of companies and universities form a coalition to create standards for common systems. These standards reduce cost of developing the product and make it inter-operable. The boring standard is a great invention on its own. Look at the USB cable you may have probably been using for many years now. If you go to USA, Europe, Asia or Africa and you may not find a good power socket but you are sure to find a perfectly matching USB charger. That is the power of standards right in front of your eyes!

http://www.IEEE.org

www.Mipi.org

www.usb.org

https://vesa.org

https://riscv.org/

https://www.jedec.org/standards-documents

https://www.accellera.org

Conferences

Chippers are normally very focused on the design in hand. There is not enough time to look at the broader picture or network with other people from the industry. Conferences provide a great opportunity to connect and learn about recent developments and best practices in the industry. Why attend conferences? Forget skills, wouldn't you like company paid travel, free food, vacation from work away from family and then a free T-shirt with a possible gift :)

Design Verification Conference

The best conference for digital design and verification. Somewhat more focused towards verification.

https://dvcon.org/

Synopsys Users Group

Conference for users of Synopsys EDA tools useful to learn about innovative ways to use the Synopsys tools. It is frequently referred as the SNUG conference.

https://www.synopsys.com/community/snug.html

Xilinx Developer Forum

https://www.xilinx.com/products/design-tools/developer-forum.html

Cadence CDN Live

Please look for the latest year for the same conference

https://www.cadence.com/en_US/home/cdnlive/silicon-valley-2019.html

Mentor Graphics User2User

https://www.mentor.com/events/user2user

Mobile World Congress Barcelona

https://www.mwcbarcelona.com/

National Association of Broadcasters Show

https://www.nabshow.com/

Hot chips: A Symposium on High Performance chips

https://www.hotchips.org/

Consumer Electronics Show

Primarily about consumer electronics. You can see most of next generation gadgets here.

https://www.ces.tech/

Blackhat

A popular conference on cyber security. You may get to see some cool hackings here, like, hackers demonstrating a car hacking and then stopping it remotely.

https://www.blackhat.com

Design Automation Conference

DAC is primarily focused on automation, covers EDA tools.

https://www.dac.com/

Websites and Blogs

This section gives links to useful websites and blogs. Some may be repeated here just for convenience.

http://www.asic-world.com/

https://opencores.org

https://www.allaboutcircuits.com/

https://www.allaboutcircuits.com/textbook/

https://zipcpu.com/

https://www.fpga4student.com

https://www.edaboard.com/

https://funrtl.wordpress.com/

http://testbench.in/

http://rtlery.com/

Apps

With smartphones in most peoples' hands, some may prefer to do all studying on the smartphone.

I found a few Android apps that may interest you.

Logic Gates – Electronic Simulator and learning by Cyfrogen

Digital Logic Design by Tech Mateen

Smart Logic Simulator by Wrapptec

VHDL and Verilog by hellotape

Though not a strict smartphone app, this desktop/laptop software is good for FSM creation

http://www.fizzim.com/

MOOC

Coursera

Expanded FPGA Training with NIOS II, University of Colorado Boulder

FPGA computing systems: Background knowledge and introductory materials, Politecnico de Milano

Digital Systems: From Logic Gates to Processors

Udemy

SystemVerilog Beginner: Write Your First Design &TB Modules, SoC Design / SoC Verification 1: Learn Verilog or System Verilog from basics to start your career in VLSI

Learn the Fundamentals of VHDL and FPGA Development, You will learn how to start with VHDL and FPGA Programming.

https://www.udemy.com/learn-the-essentials-of-vhdl-and-fpga-development/

https://www.udemy.com/rtl-design-using-hdl/

https://www.udemy.com/fundamental-of-static-timing-analysis-1/

https://www.udemy.com/hands-on-vlsi-digital-design-using-verilog-and-hardware/

https://www.udemy.com/svac-c4-soc-design-3-systemverilog-design-features/

https://www.udemy.com/test-of-intelligence-rtl-design-using-hdl-for-beginners/

Others

https://www.intel.com/content/www/us/en/programmable/support/training/overview.html

https://www.cadence.com/content/cadence-www/global/en_US/home/training/all-courses/82143.html

Datasheets and Application Notes

Apart from textbooks, blogs and videos, datasheets of chips provide useful knowledge about chipping. Application notes or app notes provide even better information about systems that you may not find in textbooks.

Unit 3 – SystemVerilog Constructs

In this unit we will focus on learning the words of the SystemVerilog language. We will not worry about writing prose or poems in SV. We will not even worry about sentences in SV. The constructs are presented in various levels. The zero level is to get a basic understanding of SV to be able to start working with the examples.

Level 0

Statement ;

The full stop or period is an under appreciated thing in languages, unless when someone says period, with anger! In SV the semicolon ; separates parts of SV code into statements. Note that the newline or carriage return or return or \n does not really separate SV code statements.

Example statement
Some words/symbols etc ;

Datatypes

I wonder we should call datatypes as containers for numbers. Lets start with the simplest number, true to the digital logic theme of this book.

```
bit some_name;
```

The word bit in SV declares that the subsequent names are to be allocated space to store a value of 0 or 1. It is exactly the binary digit aka bit! Note that the datatype followed by the name of the variable and then the semicolon together form a variable declaration statement.

```
int unsigned some_name;
```

int unsigned declares that some_name will have space to store one number, like, 100. The actual range is 0 to (2 power 32) -1 or 4294967295. The word unsigned says use only positive numbers.

```
int some_name;
```

Without the unsigned the int declaration means the some_name can store one number from the range -(2 power 31) , …, -1, 0, +1, +2, …, +(2 power 31) -1, that is, -2147483648 to 2147483647. You would use the negative numbers in places where it makes sense. For example, if you want to denote the money your friend owes you, a -100 can be interpreted as you owe your friend +100.
In the range for the int, you may notice that the positive side has one number smaller than the negative side. This is because the number 0 is arbitrarily taken as a positive number reducing one space available from the positive numbers. Using it in this way is also aligned with the overflow and signed arithmetic in two's complement form.

```
string str;
```

Suppose you want to store English letters, words and sentences, the string datatype is for you. You could do for example,

```
str = "My first s-t-r-i-n-g of letters";
```
This will store the sentence inside the quotes in the variable str. The quotes are not stored. The quotes indicates to the SV compiler that do not interpret the words inside the quotes as SV variables but interpret them as a string of ASCII characters.

Bus

A group of single bits is called a bus. Typically a bus is denoted by [Most significant index : 0].

```
bit [ 7 : 0 ] my_bus ; //describes a bus of 8 bits.
```

The bus is also called a vector or a packed array. After declaring a bus, you could refer to the individual bits like my_bus[3].

```
a = my_bus[0];
```

Array

```
int a[4]; //means an array containing 4 signed integers
a = '{4,3,5,-1}; //'{} creates array values by grouping individual elements.
```

If a[4] is used, it is like saying a[0:3] and the above value assignment will go in this order -

```
a[0] = 4; a[1]=3; … ; a[3] = -1;
```

You could also declare

```
int a[3:0];
a = '{4,3,5,-1}; //Exercise what is the value of a[3] ? 4 or -1;
```

Comments

Code without comments is difficult to understand. You want to place few comments strategically. SV lets you do that with the // character combination. All characters in the same line after the // are read by the compiler as comments and are not processed for simulation or synthesis. This particular comment is called a line comment. Note that the comment automatically ends at the end of the line. This is one place where SV does not use ; to delimit statements but uses the newline character. May be SV should also require ; to end the comment!

```
bit [ 7 : 0 ] my_bus ; //describes a bus of 8 bits.
```

There is one more comment that can disable a full section of code. It is called a block comment. To block comment many lines put /* before the section you want commented and */ to end the comment and start proper coding lines.

```
bit var1;
bit var2;
/*
bit var01;
bit var02;
*/
bit var3;
```

```
bit var4;
```

Keyword

In the examples, you can differentiate between user named words like some_name and inbuilt names of SV like bit, int and unsigned. The inbuilt names are called keywords and are not available to the chipper to use it in a different meaning. If you try to declare a user variable named bit of type int signed then it is a syntax error. Your code will not compile.

```
int unsigned bit; //is a syntax error
```

Print statement

Printing is seeing when it comes to coding. The most used SV construct is $display(). The $ symbol says that it is an inbuilt function of SV and not a user created one. The simple form takes one string type value and prints it.

```
$display("Are you having fun?");
```

Operators

After creating variables, you need to be able to do something with them. Or else, whats the point? Operators work on the variables and create new values.

Assignment operator

The simplest one is the "=" operator. The assignment operator takes the value on the right hand side or RHS and puts that into the variable on the LHS.

```
int a;
a=10;
```

Arithmetic operators

The influence of mathematics is so great on computer science that almost all programming languages define the +,-,*,/ or add, subtract, multiply and divide operators in exactly the same way as your school math textbooks would.

```
int a,b,c;
a=b+10;
c=a*b;
```

Logical operators

Digital design relies heavily on logical operations like AND, OR, NOT, XOR, NAND, NOR and XNOR. So, SV defines these operators too. However, we have a divergence between digital logic textbooks and SV conventions.

The textbook notation of a NOT operation is a line above the variable. SV is uses only ASCII characters for coding, so, a line above a letter, like $\overline{myvar}$ as the inverse of myvar is not possible to create. For some reason the operator tilde or "~" has been set to mean a NOT operation. I wonder if this happened because the ~ somewhat resembles a negative or minus.

The textbook notation for OR is a "+" but it is already taken for addition, so, SV defines the "|" for OR. In the Linux world it is also referred to as the pipe symbol. In regular English we use the "/" to mean OR, like, USA/America.

The textbook notation for AND is actually nothing! Y = AB means Y gets the AND-ed value of A, B. Nothing in SV is used to join letters into a word, just like regular English. There is no easy way to say AB would A AND B or AB is to be treated like a variable name. So, SV defines "&" to denote the AND operation. Note that in regular English also "&" is used as a shorthand for AND, such as R&D or M&A for research and development, mergers and acquisitions.

The SV XOR symbol of "^" is even more confusing. Quite frankly, I am not able to cook up any story for you! One thing, I can say is the textbook symbol of a plus sign inside a circle is not a key in the regular desktop or laptop keyboard.

The NAND, NOR and XNOR are just the "~" symbol in front of the AND,OR, XOR symbols, like, ~&, ~|, ~^.

```
bit a, b,c,d,e,f,g,h;
a = ~b;
c = ((b&e)|(f|g))^(d~&h);
```

Parenthesis

The parenthesis () has the regular meaning as in math textbooks. The innermost one is evaluated first. Without parenthesis the operators are evaluated in an order using a precedence rule, that is, which operator is to be evaluated first is specified in SV. Often, it is easier to use parenthesis liberally and save the reader the trouble of referring to SV LRM. For example, if faced with the following statement without parenthesis, I would not know what to expect for c.

```
c = b&e|f|g^d~&h;
```

Relational Operators

Less than equal to <=

Less than <

Greater than >

Greater than equal to >=

Equals ==

Not equals !=

Example: `a > 10, a<=30, a != (b+c), a==c`

Code block - begin end

Some constructs in SV assume that only the next statement is part of the construct and the statements after that are outside the scope of the current construct. To group many statements into one block use the begin-end construct. For example,

```
begin
  a = b & c;
  c = d + 1;
end
```

Branch

Programs execute serially line by line and then jump to different parts of the code based on conditions. The if-'else if'-else construct is useful for this. The order of evaluation is top to bottom. The first true condition executes and the rest of the else-if-else is not even tried. Note that there can be only one if and one else in a if-else if-else block, but many else if are allowed. If no condition evaluates to true the else branch is taken. A condition is a logical operation, like a > 10, a != 0, etc...

```
if ( <condition > )
  statement
else if ( <condition> )
  statement
//more than one else if allowed, optional
else if ( <condition> )
  statement
else
  statement
end
```

Note that just the if condition alone is also a legal code. The else if and else may be ignored. The following are valid code.

```
if ( <condition > )
  statement
else
  statement
end

if ( <condition > )
  statement
```

Code block
If you have many statements inside a if/else condition, use the begin-end to group them.

```
if ( <condition > ) begin
  statement
  statement
  statement
end else begin
  statement
```

```
  statement
  statement
end
```

Nesting

You can use another entire if or if-else or if-else if-else construct in an if construct and you can keep doing that.

```
if ( <condition > )
  if ( <condition > )
    statement
  else
    statement
  end
else
  statement
end
```

Loop

To describe many repetitions of some action is the power of a programming language. SV is no exception. SV even offers many more constructs than regular programming languages.

Repeat

The repeat keyword executes the contents inside a begin-end block for a user specified number of times.

```
repeat (<number>) begin
  statements to be repeated
end
```

For

The for loop may be more familiar to most. But is somewhat confusing. I have gotten confused by the for loop a few times.

```
for ( <initial settings> ; <test condition> ; <execute every time upon reaching end
of loop>) begin
    statements to be executed repeatedly
end
```

Example

```
for ( int i=0; i<5 ;i=i+1 )
  $display("%d",i);
```

Will print numbers from 0 through 4.

Lets deconstruct the for loop for a better understanding. Let me rewrite the for syntax.

<initial settings>

TC: if (<test condition>) begin

statements to be repeated

<execute every time upon reaching end of loop>

goto TC:

end

The for loop is comprehensive in that it allows for all kinds of repeating operations to be specified. The first statement is the initial settings. In most cases, this sets the start value of the loop variable. In the example it is 0. The second part is interesting, even before the first statement of the inside statements of the for loop are executed the go-no-go condition is tested. If the condition is valid the contents are executed. If the condition tests false the loop contents are skipped and the loop exits. Note that any side effect of executing the initial settings persists.

When the condition was tested to true the loop contents are executed once. At the end of loop contents, the last part of the for loop executes. In most cases it is the loop variable modification statement. In the example it is i=i+1. After this, the control of execution goes back to testing the condition. Note that the initial settings runs once and the condition testing and last part of for runs every loop iteration. Passing through the loop once is called an iteration.

Note that you can nest the branching and loop constructs within one another.

Delay

Remember that SV is intimately connected with chips, so, SV provides constructs to model time consumption. The #<number> is used to denote a time consumption or delay value.

```
#1; //means consume 1 unit of time before moving to the next line.
```

Event

Like the English word event, the SV construct event stands for something happening. An event by nature is momentary. The rising edge of a clock is a event.

```
@(posedge clk); // rising edge event
```

Exercise

What does @(clk) mean?

Continuous Assignment

Suppose you want to assign a value to a variable on a continuous basis use assign statement. What actually happens is that when any variable on the RHS changes the LHS is updated. To propagate constants, the RHS is also evaluated once at simulation start.

```
assign a = b+c;
```

Process blocks – initial, always, forever

The initial, always, forever blocks are called processes because like operating system processes they run parallel to one another.

Initial

Initial block marks the code that needs to execute during the start of the simulation. In this case, it is to print the message.

```
initial begin
    $display(“Today is your day!”);
end
```

Always
Like assign, always is also a continuously executing code. But there are some triggers that are needed. Always has the concept of sensitivity list. It means changes in which variables trigger the execution of the always block.

The first example without any sensitivity list is like the code is always executing without any trigger. Because it will endlessly execute without delay, the simulation time will not advance and the simulation will hang. So, the bare always block should be used only with some time control like a delay or an event.

```
always begin
  #1;
  a=b+c
end
```

The second version of always block automatically executes only when the inputs to the block change.

```
always_comb begin
  a=b+c;
end
```

Executes when b or c changes. The always_comb does not need delay to avoid hanging.

Exercise
Is assign same as always_comb with only one statement?

The third version is for flip flops

```
always_ff @(posedge clk) begin
  q <= d;
end
```

models the flip flop outputting the d input on the q output upon receiving a positive clock edge.

Forever

The initial block runs only once. To run code repeatedly inside the initial block you could use the always block without sensitivity list. SV does not allow always inside initial. Use forever block to run continuously inside initial.

```
initial begin
  clk = 0;
  forever begin
    #1;
    clk = ~clk;
  end
end
```

Module

Module is a container for whatever logic or software like routines you want to put. The untold value of the module is in limiting the visibility of variable names to within it, that is, names in one module do not interfere with names in another module.

```
module tb;
  statements like variable declarations, process blocks
endmodule
```

Suppose you do want to share some names from one module with another module, you need to declare them as ports. Like how goods enter and leave a country through its ports – sea or air, information passes across module boundaries through ports. Unlike geographical ports, SV ports have to be declared as input or output. Input ports only accept information from other modules and output ports can only give out information. There is a class called inout ports that can do both. Inout ports are not used frequently. The ports are comma separated and the last one does not have a comma.

```
module mymod (
input a,
output b
);
module statements
endmodule
```

A module can be customized using parameters. Suppose you want two modules, one for 1-bit version of the logic and another for a 2-bit version of the logic. The one bit version is mentioned above and the two bit version is shown below.

```
module mymod (
input [1:0] a,
output [1:0] b
);
endmodule
```

You could create a class of modules of any width using parameters. Note the additional parameter line. You can have many parameters separated by comma in the #() statement.

```
module mymode
#( parameter WIDTH=2)
(
  input [WIDTH-1:0] a,
```

```
  output [WIDTH-1:0 b
);
endmodule
```

Module Instantiation

A module is like a reusable code, but, it is actually dead until it is used. The SV term for used is instantiated. When a module is instantiated its parameters may be changed. Modules are instantiated inside another module.

```
module topmod ;
mymod i0 ();
endmodule
```

The above example instantiates the mymod module with an instantiation name of i0 inside topmod. The parameter value for WIDTH is set to the default of 2 because that is the value it is set to during the mymod module declaration.

```
mymod #(5) i0 ();
```

Sets the WIDTH parameter to 5. Note the order of the module name, parameter setting and then the instance name. The module and parameter order is exactly same as in the module declaration statements.

Port connection

A module that does not communicate is of little use. The ports of one module needs to connect to outside signals.

```
module topmod ;
bit [1:0] m,n;
mymod i0 (m,n);
endmodule
```

In the example, the ports a and b are connected to the topmod level names of m and n. Why a,b did not connect to n,m and only m,n is because the connections follow the order as coded in the module declaration where a comes before b. The order of port connections becomes very confusing if the number of ports are large. In that case, the named connection method is useful.

```
mymod i0 (.a(m),.b(n));
```

Can be translated to English from SV as, "connect port a of mymod with signal m, connect port b of module mymod to signal n".

Function

Group of statements in code often occur repeatedly. You may be printing a set of messages at the end of all tests in your verification.

```
$display("All tests passed! Hurray!");
$display("You are going to get a great bonus this year");
```

You could copy paste the two lines in every test of yours or define a function to make it easy to use.

```
function void happy_msg();
  $display(“All tests passed! Hurrah!”);
  $display(“You are going to get a great bonus this year”);
endfunction
```

You will call the function at the end of every test like this -

```
happy_msg();
```

Note that 2 lines containing many characters has been reduced to just two words. The main advantage of a function is that repeated code is placed in a single location. Any changes or bug fix is isolated to one place. Contrast this with the verbose approach of copy-pasting the code in every file. If you had 100 files and had to change the Hurray to Hurrah you will have to do it in 100 files.

Lets deconstruct the function syntax. A function definition happens inside the function – endfunction block. The void after the function says that the function does not return anything to the caller. The empty () means it does not take any input or output. The statements of $display is the meat of the function itself.

Functions can also take inputs and return values like in a math function.

```
function int cube (
input int a
);
cube = a*a*a;
endfunction
```

Deconstruction

Note the change of void to int because this time the cube function returns an integer type value when called. SV has a quirk in the definitions of functions, the name of the function is itself an implicitly declared variable of the same type as the return value, int. The cube variable is set to the cube of the input a and this value is returned after endfunction. The implicit declaration will be clearer if we rewrite the function like this

```
function int cube (
input int a
);
int cube; //something like this line is implicitly declared for you
cube = a*a*a;
return cube; //the value in function name is implicitly returned
endfunction
```

A function can be defined inside a module or outside a module.

Task

The function is used when the activity to be performed is purely processing something without regard to time, that is, functions complete without consuming simulation time. Tasks are meant for activities

with timing in mind. Function must complete whereas a task can be running throughout the course of the simulation. Task can and often has timing controls like delays and events. A very common clock generation task is shown below.

```
task clkgen;
clk = 0;
forever #10 clk = ~clk;
endtask
```

Deconstruction

task-endtask are SV keywords, clkgen is the name of the task and there are 2 statements inside the task.

Task can be defined inside or outside a module.

Unit 4 – Software Like Examples

The easiest way to learn chipping is by doing. In this unit, I am giving pointers and actual examples to expand your learning. For software oriented chippers, it may be easier to master SV as a programming language and then gradually take in the hardware oriented constructs. The examples listed in this book are available in my GitHub page. Some code lines may be wrapped in your ebook reader, a landscape orientation with smaller font or zooming out may help you see unwrapped code.

https://github.com/3vm/dsn_verif

If you don't like being force fed examples there is a more interesting approach to doing these examples. Go to an intermediate git commit for the example and then debug and make it work! To find an intermediate commit use git log on the directory you are interested. The "." option to log is important because that tells the commits to that directory.

git log .

It will give a list of commits for the files in that directory. Checkout some intermediate version for one file.

git checkout <intermediate commit id, not the latest>

example:

git checkout 1c1e33559360757214876fc737bcc1bec5183d04

Once you figure out the bugs, you can go back to the master tag with the command

git checkout master

If you keep one directory for the golden version of the repository and another directory to keep the experimental version, you can also do diff between the two easily.

My examples are not silicon proven. There could be **bugs**. So, use these in you production designs only after **full verification**.

There are plenty of other examples available for free online. Many of these are in the older Verilog-2001 format. Just read the code as if it were SystemVerilog.

http://www.asic-world.com

Open Hardware design library from Parallela

https://github.com/parallella/oh/tree/master/src

I found this library to be the most extensive open source hardware repository. I was surprised that many really complicated designs actually work!

https://opencores.org/

Sample code from textbooks

SystemVerilog for design book examples

Code of examples used in Fundamentals of Digital Logic with Verilog Design, 3rd Edition

http://www.eecg.toronto.edu/~brown/Verilog_3e/verilog_source/Welcome.html

If you are looking for any specific design, please search in GitHub before starting. There could be a free and open version made for you. Of course, you would first google too!

You may enter the world of chipping via any of two gates – software or hardware. In the software approach, coders start with hello world. So, I have included that as the first example.

This YouTube video gives all the instructions to installing Xilinx Vivado.

https://www.youtube.com/watch?v=iwVd4TIB0ME

Hello World

Printing hello world seems trivial. It can teach a good bit of information about the simulator.

```
program tb ;
initial begin
  $display ( "Hello World" );
end
endprogram
```

Compiling
I have used Xilinx Vivado to compile most of these examples. The compilation step is done by the xvlog executable. I think, the compile step reads every SV file you have and then creates a representation inside Vivado suitable for further processing.

C:\Xilinx\Vivado\2019.1\bin\xvlog --sv tb.sv
I saved this command to compile.bat for MS Windows environment.

For Linux
~/xilinx_vivado/Vivado/2019.2/bin/xvlog --sv tb.sv

Messages from compiler

INFO: [VRFC 10-2263] Analyzing SystemVerilog file "D:/dsn_verif/examples/hello_world/tb.sv" into library work
INFO: [VRFC 10-311] analyzing module tb

Elaboration
This step makes the connection between the various SV files that were compiled before. After elaboration, I think, the CPU executable binary is ready.

C:\Xilinx\Vivado\2019.1\bin\xelab tb -L xil_defaultlib
Saved into elaborate.dat

For Linux
~/xilinx_vivado/Vivado/2019.2/bin/xelab --snapshot work tb

Messages from elaborator

Running: xelab.exe tb -L xil_defaultlib
Multi-threading is on. Using 2 slave threads.
Starting static elaboration
Completed static elaboration
Starting simulation data flow analysis
Completed simulation data flow analysis
Time Resolution for simulation is 1ps
Compiling module work.tb
Built simulation snapshot work.tb

Simulation
The CPU executable binary from the elaborate step is passed to the simulator xsim. I think, the simulator passes inputs the executable model of the SV files and then collects outputs and internal values if you want to probe them. It also stops the simulation at a given time as requested by the user.

C:\Xilinx\Vivado\2019.1\bin\xsim -R work.tb
Note that the executable binary is in the work.tb directory.

For Linux
~/xilinx_vivado/Vivado/2019.2/bin/xsim work --runall

Messages from Vivado Simulation
source xsim.dir/work.tb/xsim_script.tcl
xsim {work.tb} -autoloadwcfg -runall
Vivado Simulator 2019.1
Time resolution is 1 ps
run -all
Hello World
exit

For linux, all the three steps are saved into one run file run.sh
Contents of run.sh

```
~/xilinx_vivado/Vivado/2019.2/bin/xvlog --incr --sv --work work -f sim.f
~/xilinx_vivado/Vivado/2019.2/bin/xelab --snapshot work tb
~/xilinx_vivado/Vivado/2019.2/bin/xsim work –runall
```

The sim.f file provides a list of source code files for compilation step.

GCD recursive and non-recursive

Lets look at the age old Greatest Common Division (GCD) algorithm implemented in iterative and recursive ways. The following code assumes that inputs are positive and non zero.

https://en.wikipedia.org/wiki/Euclidean_algorithm

```
program tb ;
int a , b ;
initial begin
   $display ( "Iterative GCD" ) ;
   a = 140 ; b = 40 ;
   $display ( "GCD of %4d and %4d is %4d" , a , b , gcd_original ( a , b ) ) ;

   $display ( "Recursive GCD" ) ;
   a = 140 ; b = 40 ;
   $display ( "GCD of %4d and %4d is %4d" , a , b , gcd_recursive ( a , b ) ) ;
end

function int gcd_original ( input int a , input int b ) ;
 while ( a != b ) begin
   if ( a > b )
     a = a - b ;
   else
     b = b - a ;
 end
 return a ;
endfunction

function int gcd_recursive ( input int a , input int b ) ;
 $display ( "A %4d , B %4d" , a , b ) ;
 if ( a == 1 || b == 1 )
   return 1 ;
 else if ( a == b )
   return a ;
 else if ( a > b )
   return gcd_recursive ( a - b , b ) ;
 else if ( a < b )
   return gcd_recursive ( a , b - a ) ;
endfunction

endprogram
```

run -all
Iterative GCD
GCD of 140 and   40 is   20
Recursive GCD
A 140 , B   40
A 100 , B   40
A   60 , B   40
A   20 , B   40
A   20 , B   20
GCD of 140 and   40 is   20

Recursive Prime Factorization

Splitting a number into product of prime numbers is called prime factorization. A prime number is one that is divisible only by 1 and itself. Prime numbers are a fun way to learn simple programming. But don't be fooled by the look of elementary math, prime factorization is a hard problem that we all rely on to secure our financial and other transactions online. The method to factorize is not too different from trial and error. Starting from the number to 2, we work upwards. If the trial number divides the input number we set aside the divisor into a queue. A queue is declared with the dollar symbol inside square brackets. The algorithm continues to divide the numerator by the current trial factor for as along as it divides. Whenever the trial factor successfully divides, the numerator is divided and reassigned the reduced value. The repeated division by the same number is needed to remove the factors of a number that repeat. For example, 12 is 2*2*3 where 2 divides twice. Once, the trial factor no longer divides the numerator the program proceeds with the next number. This process happens repeatedly until the number and the factor attempted is same. At this point the queue holds all the factors.

```
program tb ;

initial begin
  int num , try_factor ;
  int factors [ $ ] ;
  num = 45 ;
  try_factor = 2 ;
  factorize ( num , try_factor , factors ) ;
  $finish ;
end

function automatic void factorize (
ref int num ,
ref int try_factor ,
ref int factors [ $ ]
 ) ;

$display ( "number %4d , trial factor %4d \n Factors " , num , try_factor ) ;
foreach ( factors [ i ] )
 $write ( "%4d" , factors [ i ] ) ;
$display ( ) ;

if ( num == 1 ) begin
  factors.push_back ( 1 ) ;
  return ;
end else begin
  if ( num % try_factor == 0 ) begin
    num = num / try_factor ;
    factors.push_back ( try_factor ) ;
    $display ( "Factor found %4d" , try_factor ) ;
  end else begin
    try_factor ++ ;
  end
  factorize ( num , try_factor , factors ) ;
end

endfunction

endprogram
```

run -all
number   45 , trial factor   2
 Factors

number   45 , trial factor   3
 Factors

Factor found   3
number   15 , trial factor   3
 Factors
   3
Factor found   3
number   5 , trial factor   3
 Factors
   3   3
number   5 , trial factor   4
 Factors
   3   3
number   5 , trial factor   5
 Factors
   3   3
Factor found   5
number   1 , trial factor   5
 Factors
   3   3   5

Exercise
Change the call to factorize function in the initial block from factorize(num,try_factor,factors) to factorize(num,2,factors) and see what happens.
Modify the function to a non-recursive factorization code.

Change Problem

Have you had the experience of waiting at the cash counter and trying to get change? You may have also experienced the other problem of having to calculate change if you happen to be the cashier. It is a nice problem to demonstrate dynamic programming style of algorithms. Dynamic programming builds up the solution to the current problem by iteratively building up from a smaller version of the same problem. It so happens that for some class of problems it is easier to calculate this way than recursion.

https://en.wikipedia.org/wiki/Change-making_problem

For this problem, I have referred to the book, "An Introduction to Bioinformatics Algorithms" by Neil C. Jones and Pavel A. Pevzner

Note the usage of SV constructs here. The algorithm calls for assigning infinity to the number of coins variable. You could use floating point numbers and assign infinity possibly. But change problem is an inherently whole number problem. So, I have used the maximum value for an unsigned integer. To make the max value automatic, it is easier to set the value to -1 which when converted to unsigned becomes the maximum possible value in signed number. Note that this method is somewhat ugly and you need to use a more robust method for production quality code.

You can see the execution of the code to understand the dynamic programming build up process. From 7 to 10 the build up process was removed because the number of lines where too many.

```systemverilog
program tb ;

localparam NUM_DENOM = 4 ;
localparam int DENOMINATIONS [ NUM_DENOM ] = '{10 , 5 , 2 , 1} ;

typedef int unsigned change_t [ NUM_DENOM ] ;
change_t bf_chg , recurs_chg , dp_chg ;

int TRIALS = 1 ;

initial begin
   int unsigned money ;
   for ( int i = 0 ; i < TRIALS ; i ++ ) begin
      // money = $urandom_range ( 1 , 1024 ) ;
      money = 13 ;
      $display ( "Money %5d" , money ) ;
      $display ( "Greedy change" ) ;
      bf_chg = greedy_change ( money ) ;
      disp_change ( bf_chg ) ;
      $display ( "Dynamic Programming change" ) ;
      dp_chg = dp_change ( money ) ;
      //disp_change ( dp_chg ) ;
      if ( dp_chg != bf_chg ) begin
         $display ( "Different change" ) ;
         break ;
      end
   end
   $finish ;
end

function automatic change_t greedy_change (
input int unsigned money
 ) ;
 change_t chg ;
 foreach ( DENOMINATIONS [ i ] ) begin
   chg [ i ] = money / DENOMINATIONS [ i ] ;
   money = money % DENOMINATIONS [ i ] ;
 end
 return chg ;
endfunction

function automatic change_t dp_change (
input int unsigned money
```

```systemverilog
  ) ;
  const int unsigned MAX_VALUE = -1 ;
  int unsigned num_coins , this_deno , prev_coins ;
  // change_t chg [ ] = new [ money + 1 ] ;
  change_t chg [ 1024 ] ;
  change_t prev_chg ;

  chg [ 0 ] = '{default : 0} ;
  // $display ( MAX_VALUE ) ;
  for ( int i = 1 ; i <= money ; i ++ ) begin
    num_coins = MAX_VALUE ;
    $display ( "Calculating best change for money %5d" , i ) ;

    foreach ( DENOMINATIONS [ d ] ) begin
      this_deno = DENOMINATIONS [ d ] ;
      if ( i >= this_deno ) begin
        $display ( "Attempting to find best change for money %5d using best change
for %5d", i, i-this_deno ) ;
        $display ( "Change for %5d" , i-this_deno ) ;
        disp_change ( chg [ i-this_deno ] ) ;

        prev_chg = chg [ i-this_deno ] ;
        prev_coins = prev_chg.sum ( ) ;
        if ( prev_coins + 1 < num_coins ) begin
          num_coins = prev_coins + 1 ;
          chg [ i ] = chg [ i-this_deno ] ;
          chg [ i ] [ d ] = chg [ i ] [ d ] + 1 ;
          $display ( "Improvement found using denomination %5d" , DENOMINATIONS [ d
] ) ;
          disp_change ( chg [ i ] ) ;
        end else begin
          $display ( "No improvement found using denomination %5d" , DENOMINATIONS [
d ] ) ;
        end
      end else begin
        $display ( "Ignoring denomination %5d for money %5d" , this_deno , i ) ;
      end
    end
    $display ( "--------------------------------") ;
    $display ( "Best change for money %5d is" , i ) ;
    $display ( "--------------------------------") ;
    disp_change ( chg [ i ] ) ;
    $display ( "_____");
  end
  return chg [ money ] ;
endfunction

function automatic void disp_change (
input change_t chg
) ;
  foreach ( chg [ i ] )
    $write ( "%3d x %1d \t" , DENOMINATIONS [ i ] , chg [ i ] ) ;
  $display ( ) ;

  $display ( "Total Coins %5d\n" , chg.sum ( ) ) ;

endfunction
```

```
function automatic int unsigned total_change (
input change_t chg
 ) ;
 int unsigned sum = 0 ;
 foreach ( chg [ i ] )
 sum += chg [ i ] * DENOMINATIONS [ i ] ;
 return sum ;
endfunction

endprogram
```

Execution
run -all
Money   13
Greedy change
 10 x 1   5 x 0   2 x 1   1 x 1
Total Coins   3

Dynamic Programming change
Calculating best change for money   1
Ignoring denomination   10 for money   1
Ignoring denomination   5 for money   1
Ignoring denomination   2 for money   1
Attempting to find best change for money   1 using best change for   0
Change for   0
 10 x 0   5 x 0   2 x 0   1 x 0
Total Coins   0

Improvement found using denomination   1
 10 x 0   5 x 0   2 x 0   1 x 1
Total Coins   1

Best change for money   1 is
----
 10 x 0   5 x 0   2 x 0   1 x 1
Total Coins   1

_____
Calculating best change for money   2
Ignoring denomination   10 for money   2
Ignoring denomination   5 for money   2
Attempting to find best change for money   2 using best change for   0
Change for   0
 10 x 0   5 x 0   2 x 0   1 x 0
Total Coins   0

Improvement found using denomination   2

10 x 0   5 x 0   2 x 1   1 x 0
Total Coins   1

Attempting to find best change for money   2 using best change for   1
Change for   1
 10 x 0   5 x 0   2 x 0   1 x 1
Total Coins   1

No improvement found using denomination   1

Best change for money   2 is
----
 10 x 0   5 x 0   2 x 1   1 x 0
Total Coins   1

_____
Calculating best change for money   3
Ignoring denomination   10 for money   3
Ignoring denomination   5 for money   3
Attempting to find best change for money   3 using best change for   1
Change for   1
 10 x 0   5 x 0   2 x 0   1 x 1
Total Coins   1

Improvement found using denomination   2
 10 x 0   5 x 0   2 x 1   1 x 1
Total Coins   2

Attempting to find best change for money   3 using best change for   2
Change for   2
 10 x 0   5 x 0   2 x 1   1 x 0
Total Coins   1

No improvement found using denomination   1

Best change for money   3 is
----
 10 x 0   5 x 0   2 x 1   1 x 1
Total Coins   2

_____
Calculating best change for money   4
Ignoring denomination   10 for money   4
Ignoring denomination   5 for money   4
Attempting to find best change for money   4 using best change for   2
Change for   2
 10 x 0   5 x 0   2 x 1   1 x 0

Total Coins   1

Improvement found using denomination   2
 10 x 0   5 x 0   2 x 2   1 x 0
Total Coins   2

Attempting to find best change for money   4 using best change for   3
Change for   3
 10 x 0   5 x 0   2 x 1   1 x 1
Total Coins   2

No improvement found using denomination   1

Best change for money   4 is
----
 10 x 0   5 x 0   2 x 2   1 x 0
Total Coins   2

_____
Calculating best change for money   5
Ignoring denomination   10 for money   5
Attempting to find best change for money   5 using best change for   0
Change for   0
 10 x 0   5 x 0   2 x 0   1 x 0
Total Coins   0

Improvement found using denomination   5
 10 x 0   5 x 1   2 x 0   1 x 0
Total Coins   1

Attempting to find best change for money   5 using best change for   3
Change for   3
 10 x 0   5 x 0   2 x 1   1 x 1
Total Coins   2

No improvement found using denomination   2
Attempting to find best change for money   5 using best change for   4
Change for   4
 10 x 0   5 x 0   2 x 2   1 x 0
Total Coins   2

No improvement found using denomination   1

Best change for money   5 is
----
 10 x 0   5 x 1   2 x 0   1 x 0
Total Coins   1

_____

Calculating best change for money   6
Ignoring denomination   10 for money   6
Attempting to find best change for money   6 using best change for   1
Change for   1
 10 x 0   5 x 0   2 x 0   1 x 1
Total Coins   1

Improvement found using denomination   5
 10 x 0   5 x 1   2 x 0   1 x 1
Total Coins   2

Attempting to find best change for money   6 using best change for   4
Change for   4
 10 x 0   5 x 0   2 x 2   1 x 0
Total Coins   2

No improvement found using denomination   2
Attempting to find best change for money   6 using best change for   5
Change for   5
 10 x 0   5 x 1   2 x 0   1 x 0
Total Coins   1

No improvement found using denomination   1

Best change for money   6 is
----
 10 x 0   5 x 1   2 x 0   1 x 1
Total Coins   2
.
.
.

_____

Calculating best change for money   11
Attempting to find best change for money   11 using best change for   1
Change for   1
 10 x 0   5 x 0   2 x 0   1 x 1
Total Coins   1

Improvement found using denomination   10
 10 x 1   5 x 0   2 x 0   1 x 1
Total Coins   2

Attempting to find best change for money   11 using best change for   6
Change for   6
 10 x 0   5 x 1   2 x 0   1 x 1

Total Coins   2

No improvement found using denomination   5
Attempting to find best change for money   11 using best change for   9
Change for   9
 10 x 0   5 x 1   2 x 2   1 x 0
Total Coins   3

No improvement found using denomination   2
Attempting to find best change for money   11 using best change for   10
Change for   10
 10 x 1   5 x 0   2 x 0   1 x 0
Total Coins   1

No improvement found using denomination   1

Best change for money   11 is
----
 10 x 1   5 x 0   2 x 0   1 x 1
Total Coins   2

_____
Calculating best change for money   12
Attempting to find best change for money   12 using best change for   2
Change for   2
 10 x 0   5 x 0   2 x 1   1 x 0
Total Coins   1

Improvement found using denomination   10
 10 x 1   5 x 0   2 x 1   1 x 0
Total Coins   2

Attempting to find best change for money   12 using best change for   7
Change for   7
 10 x 0   5 x 1   2 x 1   1 x 0
Total Coins   2

No improvement found using denomination   5
Attempting to find best change for money   12 using best change for   10
Change for   10
 10 x 1   5 x 0   2 x 0   1 x 0
Total Coins   1

No improvement found using denomination   2
Attempting to find best change for money   12 using best change for   11
Change for   11
 10 x 1   5 x 0   2 x 0   1 x 1

Total Coins   2

No improvement found using denomination   1

Best change for money   12 is
----
 10 x 1   5 x 0   2 x 1   1 x 0
Total Coins   2

_____
Calculating best change for money   13
Attempting to find best change for money   13 using best change for   3
Change for   3
 10 x 0   5 x 0   2 x 1   1 x 1
Total Coins   2

Improvement found using denomination   10
 10 x 1   5 x 0   2 x 1   1 x 1
Total Coins   3

Attempting to find best change for money   13 using best change for   8
Change for   8
 10 x 0   5 x 1   2 x 1   1 x 1
Total Coins   3

No improvement found using denomination   5
Attempting to find best change for money   13 using best change for   11
Change for   11
 10 x 1   5 x 0   2 x 0   1 x 1
Total Coins   2

No improvement found using denomination   2
Attempting to find best change for money   13 using best change for   12
Change for   12
 10 x 1   5 x 0   2 x 1   1 x 0
Total Coins   2

No improvement found using denomination   1

Best change for money   13 is
----
 10 x 1   5 x 0   2 x 1   1 x 1
Total Coins   3

Exercise

Why is the use of $urandom() and then running many trials with various numbers and comparing against the brute force change algorithm not efficient for verifying this Dynamic Programming (DP) algorithm, in comparison to just running for one large number?

Try a recursive change algorithm and compare the runtime with DP algorithm.

You can see that the buffer (chg variable in dp_change function) holding the change for all the money from 1 to N is large. In this example, it is set to 1024 elements. How can you reduce this storage requirement? Hint: To calculate change for N we only need change for N – max denomination.

Manhattan Tourist Problem

Suppose you are in a new city and want to visit as many tourist attractions in the city without coming back to the same place, there is an algorithm for that. In its simplest form, the city is assumed to have a neat grid pattern of streets laid out in 90 degree crossings. The intersections are taken as nodes of a graph and the streets form the edges of the graph. The edges are marked with weights representing the value of the attractions seen along the streets. So, if you start from the northwest corner of the city and move in only south or east direction, you will ultimately end up at the southeast corner. There are many routes to take. Exactly which route gives the highest score in terms of number of attractions seen can be solved with a dynamic programming algorithm. I have used a version of the algorithm specified in the book, "An Introduction to Bioinformatics Algorithms, Niel C. Jones and Pavel A. Pevzner.

The following program in SV has 3 variables. The first one named down contains the edge weights if you move south along the streets. The array right has similar meaning of moving east. The best array is the working variable that stores the score of the best routes from the northwest corner to that intersection. The best array is initialized to 0 and the dynamic programming buildup process calculates the best score one intersection at a time.

```
program tb ;

localparam ROWS = 4 , COLS = 4 ;
int right [ 0 : ROWS ] [ 1 : COLS ] = '{
 '{ 3 , 2 , 4 , 0 } ,
 '{ 3 , 2 , 4 , 2 } ,
 '{ 0 , 7 , 3 , 4 } ,
 '{ 3 , 3 , 0 , 2 } ,
 '{ 1 , 3 , 2 , 2 }
 } ;

int down [ 1 : ROWS ] [ 0 : COLS ] = '{
 '{ 1 , 0 , 2 , 4 , 3 } ,
 '{ 4 , 6 , 5 , 2 , 1 } ,
 '{ 4 , 4 , 5 , 2 , 1 } ,
 '{ 5 , 6 , 8 , 5 , 3 }
 } ;

int best [ 0 : ROWS ] [ 0 : COLS ] ;

initial begin
   best = '{ default : 0 } ;
   for ( int i = 1 ; i <= ROWS ; i ++ ) begin
     best [ i ] [ 0 ] = best [ i-1 ] [ 0 ] + down [ i ] [ 0 ] ;
```

```verilog
      $display ( "Best value for intersection %2d %2d is %3d" , i , 0 , best [ i ] [
0 ] ) ;
   end
   for ( int j = 1 ; j <= COLS ; j ++ ) begin
      best [ 0 ] [ j ] = best [ 0 ] [ j-1 ] + right [ 0 ] [ j ] ;
      $display ( "Best value for intersection %2d %2d is %3d" , 0 , j , best [ 0 ] [
j ] ) ;
   end
   for ( int i = 1 ; i <= ROWS ; i ++ ) begin
      for ( int j = 1 ; j <= COLS ; j ++ ) begin
         best [ i ] [ j ] = max ( best [ i-1 ] [ j ] + down [ i ] [ j ] , best [ i ]
[ j-1 ] + right [ i ] [ j ] ) ;
         $display ( "Best value for intersection %2d %2d is %3d" , i , j , best [ i ]
[ j ] ) ;
      end
   end
   $display ( "Best route value is %4d" , best [ ROWS ] [ COLS ] ) ;

   $finish ;
end

function int max ( int a , int b ) ;
 max = a > b ? a : b ;
endfunction

endprogram

run -all
Best value for intersection  1  0 is   1
Best value for intersection  2  0 is   5
Best value for intersection  3  0 is   9
Best value for intersection  4  0 is  14
Best value for intersection  0  1 is   3
Best value for intersection  0  2 is   5
Best value for intersection  0  3 is   9
Best value for intersection  0  4 is   9
Best value for intersection  1  1 is   4
Best value for intersection  1  2 is   7
Best value for intersection  1  3 is  13
Best value for intersection  1  4 is  15
Best value for intersection  2  1 is  10
Best value for intersection  2  2 is  17
Best value for intersection  2  3 is  20
Best value for intersection  2  4 is  24
Best value for intersection  3  1 is  14
Best value for intersection  3  2 is  22
Best value for intersection  3  3 is  22
Best value for intersection  3  4 is  25
Best value for intersection  4  1 is  20
Best value for intersection  4  2 is  30
Best value for intersection  4  3 is  32
Best value for intersection  4  4 is  34
Best route value is   34
```

Longest Common Subsequence

In genomics, you want to know how much similarity is there between two DNA sequences. For example, if you have two sequences as ATATACGT and TAGCTATT, you could arrange them in the following way. The "-" character is used to show gaps which enables you to see the common characters aligned.

ATA--TACG-T

-TAGCTA--TT

The subsequence **TATAT** in bold is common to both the sequences. Note that TAT or TACT are also subsequences common to both but we are interested in the longest one. The bioinformatics book mentioned before has an algorithm for the LCS problem. I have coded that algorithm in SV for you. The algorithm builds the alignments from shorter sequences to the full sequence in the dynamic programming style. The shorter to longer alignments are continuously stored in an alignment matrix along with a backtracking pointer to printout the best alignment at the end.

```systemverilog
program tb ;

localparam string seq0 = "ATCTGAT" , seq1 = "TGCATA" ;
localparam N = seq0.len ( ) , M = seq1.len ( ) ;

int alnmat [ 0 : N ] [ 0 : M ] ;
byte trace [ 1 : N ] [ 1 : M ] ;

initial begin
  alnmat = '{ default : 0 } ;
  for ( int i = 1 ; i <= N ; i ++ ) begin
   for ( int j = 1 ; j <= M ; j ++ ) begin
     alnmat [ i ] [ j ] = max ( alnmat [ i-1 ] [ j ] , alnmat [ i ] [ j-1 ] ) ;
     if ( seq0 [ i-1 ] == seq1 [ j-1 ] ) begin
      alnmat [ i ] [ j ] = max ( alnmat [ i ] [ j ] , alnmat [ i-1 ] [ j-1 ] + 1 ) ;
     end
     $display ( "alnmat [ %2d ] [ %2d ] = " , i , j , alnmat [ i ] [ j ] ) ;
     if ( alnmat [ i ] [ j ] == alnmat [ i-1 ] [ j ] ) begin
      trace [ i ] [ j ] = "U" ;
     end else if ( alnmat [ i ] [ j ] == alnmat [ i ] [ j-1 ] ) begin
      trace [ i ] [ j ] = "L" ;
     end else begin
      trace [ i ] [ j ] = "D" ;
     end
   end
  end

  trace_back ( N , M ) ;
  $display ( ) ;
  $finish ;
end

function automatic void trace_back ( input int i , input int j ) ;
 if ( i == 0 || j == 0 ) begin
  return ;
```

```
        end
    if ( trace [ i ] [ j ] == "D" ) begin
      trace_back ( i-1 , j-1 ) ;
       $write ( seq0 [ i-1 ] ) ;
    end else begin
      if ( trace [ i ] [ j ] == "U" ) begin
       trace_back ( i-1 , j ) ;
      end else begin
       trace_back ( i , j-1 ) ;
      end
    end
  endfunction

  function automatic int max ( int a , int b ) ;
   max = a > b ? a : b ;
  endfunction

  endprogram

run -all
alnmat [  1 ][  1 ] =        0
alnmat [  1 ][  2 ] =        0
alnmat [  1 ][  3 ] =        0
alnmat [  1 ][  4 ] =        1
alnmat [  1 ][  5 ] =        1
alnmat [  1 ][  6 ] =        1
alnmat [  2 ][  1 ] =        1
alnmat [  2 ][  2 ] =        1
alnmat [  2 ][  3 ] =        1
alnmat [  2 ][  4 ] =        1
alnmat [  2 ][  5 ] =        2
alnmat [  2 ][  6 ] =        2
alnmat [  3 ][  1 ] =        1
alnmat [  3 ][  2 ] =        1
alnmat [  3 ][  3 ] =        2
alnmat [  3 ][  4 ] =        2
alnmat [  3 ][  5 ] =        2
alnmat [  3 ][  6 ] =        2
alnmat [  4 ][  1 ] =        1
alnmat [  4 ][  2 ] =        1
alnmat [  4 ][  3 ] =        2
alnmat [  4 ][  4 ] =        2
alnmat [  4 ][  5 ] =        3
alnmat [  4 ][  6 ] =        3
alnmat [  5 ][  1 ] =        1
alnmat [  5 ][  2 ] =        2
alnmat [  5 ][  3 ] =        2
alnmat [  5 ][  4 ] =        2
alnmat [  5 ][  5 ] =        3
alnmat [  5 ][  6 ] =        3
alnmat [  6 ][  1 ] =        1
alnmat [  6 ][  2 ] =        2
alnmat [  6 ][  3 ] =        2
alnmat [  6 ][  4 ] =        3
alnmat [  6 ][  5 ] =        3
alnmat [  6 ][  6 ] =        4
alnmat [  7 ][  1 ] =        1
alnmat [  7 ][  2 ] =        2
alnmat [  7 ][  3 ] =        2
alnmat [  7 ][  4 ] =        3
alnmat [  7 ][  5 ] =        4
alnmat [  7 ][  6 ] =        4
TCTA
```

Linked Objects

In nature, every concept or object is linked to many other similar entities. A person has relatives, friends and may be enemies! In hardware, memory comes in linked chunks of free space. Algorithms are specified using graphs. A graph is a list of vertices linked to each other by edges. Modeling a graph is relatively straight forward in behavioral code. I have always wondered, how to do this linking without pointers. In C/C++ languages there are pointers that allow the user to link objects using their addresses. But SV has no pointers. Lucky! Pointers are a nightmare to use anyway! Instead you can notice that declaring a class variable is actually declaring a handle to a class object. The object is actually only made when the new() constructor is executed. There it is! The class handle is the SV equivalent of a pointer. So, linked objects can be implemented with classes that have a member that is a handle to a class type! Following code shows a simple linked list. Bear with me here, I am also learning object linking in SV with you. In the example shown, the list_c prev is the class handle that links one class object to another instance of the same class. Five elements are created with new function call and each object's elem member gets the value from 0 through 4. After a new object is created its previous object handle is assigned the handle of the previous object just made in the previous iteration. At the end, the list is traversed starting from the last element for 5 preceding elements.

```
program tb ;

class list_c ;
 int elem ;
 list_c prev ;
endclass

list_c prev_elem = new ( ) ;
list_c this_elem ;
list_c last_elem ;

initial begin
 for ( int i = 0 ; i < 5 ; i ++ ) begin
   this_elem = new ( ) ;
   this_elem.elem = i ;
   this_elem.prev = prev_elem ;
   prev_elem = this_elem ;
 end
 last_elem = this_elem ;

 this_elem = last_elem ;
 for ( int i = 0 ; i < 5 ; i ++ ) begin
   $display ( "List element at node %d value %d" , i , this_elem.elem ) ;
   this_elem = this_elem.prev ;
 end
end

endprogram


run -all
List element at node        0 value        4
List element at node        1 value        3
```

List element at node	2 value	2
List element at node	3 value	1
List element at node	4 value	0

Exercise

How do you find an unallocated node in the list? That is, one where the new() has not been called.

Porting C Code to SV – Sudoku

C language being the grandfather of SV has a vast collection of libraries covering all sorts of algorithms. It is also one of the fastest executing programming languages. I could even spot research papers mentioning C code for nuclear reaction simulations. A chipper may need to port C code to SV at some point in his career. In this example, I present my experience porting the C code to SV. Boy! It was harder than I thought! Sudoku is a popular game that has a 9x9 matrix of numbers with some filled and some not filled. I had created a solver for Sudoku in C many years ago. The original C code is presented first. The program works is 3 basic steps – get Sudoku puzzle, initialize a 3D array with all the possibilities of numbers for every row and column, solve the puzzle using depth first search approach. The prossibilities matrix, named prob/fnp contains the numbers that are allowed for every box. Example

prob[0] is the 9x9 matrix that shows in which row/column box position the number 1 is allowed for the given Sudoku input matrix

prob[1] same as prob[0] but for the allowed row for number 2

prob[2] for number 3

prob[8] for number 9

```
   1     8 9     4
6           5 4         3
4  2              6 7
7        4              3 8
   9  2              5 4
3  6              1        9
      8  1              2 6
1        9  3              7
      5        7 8     1
```

Header file sudoku_orig.h:

```c
#define size 3
#define ssq size*size

#include <stdio.h>
void sud ( char *, char *, char, char ) ;
```

Main code sudoku_orig.c

```c
#include "sudoku_orig.h"

int main( void )
{

    char rp[ssq][ssq],cp[ssq][ssq],bp[ssq][ssq];
    char i,j,k,l,flag,rowstart,colstart,boxno;
    char prob[ssq][ssq][ssq];
    int in_num;
    char s[ssq][ssq]={    {0,1,0,8,9,0,4,0,0},
            {6,0,0,0,5,4,0,0,3},
            {4,2,0,0,0,6,7,0,0},
            {7,0,0,4,0,0,0,3,8},
            {0,9,2,0,0,0,5,4,0},
            {3,6,0,0,0,1,0,0,9},
            {0,0,8,1,0,0,0,2,6},
            {1,0,0,9,3,0,0,0,7},
            {0,0,5,0,7,8,0,1,0}    };

    for(i=0;i<=ssq-1;i++)
    {
        for(j=0;j<=ssq-1;j++)
        {
            printf("%d ",s[i][j]);
        }
        printf("\n");
    }
    printf("\n");

    for(i=0;i<=ssq-1;i++)
    {
        for(j=1;j<=ssq;j++)
        {
            //row probables
            rp[i][j-1]=1;
            for(k=0;k<=ssq-1;k++)
            {
                if(s[i][k]==j)
                {
                    rp[i][j-1]=0;
                    break;
                }
            }

            //column probables
            cp[i][j-1]=1;
            for(k=0;k<=ssq-1;k++)
            {
                if(s[k][i]==j)
                {
                    cp[i][j-1]=0;
                    break;
                }
```

```c
            }
        }

        //box probables
        rowstart=(i/size)*size;
        colstart=(i%size)*size;
        for(j=1;j<=ssq;j++)
        {
            bp[i][j-1]=1;
            for(k=0;k<=size-1;k++)
            {
                for(l=0;l<=size-1;l++)
                {
                    if(s[rowstart+k][colstart+l]==j)
                    {
                        bp[i][j-1]=0;
                        goto out2loops;
                    }
                }
            }
            out2loops: ;
        }
    }

    for(i=0;i<=ssq-1;i++)
    {
        for(j=0;j<=ssq-1;j++)
        {
            boxno=(i/size)*size+(j/size);
            flag=1;
            if(s[i][j])
            {
                flag=0;
            }
            for(k=0;k<=ssq-1;k++)
            {
                prob[k][i][j]=(rp[i][k])&&(cp[j][k])&&(bp[boxno][k])&&flag;
            }
        }
    }

    sud(&prob[0][0][0], &s[0][0], 0, 0);
    //getch();
    return 0;
}

void sud(char *a,char *b, char c, char d)
{
    char i,j,k,i0,j0,rowstart,colstart,boxno;
    char fnp[ssq][ssq][ssq],temp[ssq][ssq];
    char fns[ssq][ssq];

    //copy into a new matrix
    for(i=0;i<=ssq-1;i++)
    {
```

```c
        for(j=0;j<=ssq-1;j++)
        {
            fns[i][j]=*(b+i*ssq+j);
            for(k=0;k<=ssq-1;k++)
            {
                fnp[i][j][k]=*(a+i*ssq*ssq+j*ssq+k);
            }
        }
    }

    if((c==ssq)&&(d==ssq))
    {
        printf("\n The solution is\n");
        for(i=0;i<=ssq-1;i++)
        {
            for(j=0;j<=ssq-1;j++)
            {
                printf("%d ",fns[i][j]);
            }
            printf("\n");
        }
        printf("\n");

        return;
    }

    //find the vacant square
    for(i=c;i<=ssq-1;i++)
    {
        if(i!=c)
        {
            d=0;
        }
        for(j=d;j<=ssq-1;j++)
        {
            if(fns[i][j]==0)
            {
                goto out;
            }
        }
    }
    out:
    i0=i;
    j0=j;
    if((i0==ssq)&&(j0==ssq))
    {
        printf("\n The solution is\n");
        for(i=0;i<=ssq-1;i++)
        {
            for(j=0;j<=ssq-1;j++)
            {
                printf("%d ",fns[i][j]);
            }
            printf("\n");
        }
    }
```

```c
      printf("\n");
      return;
}

//find the probable number to fill up the vacant square
for(k=0;k<=ssq-1;k++)
{
   if(fnp[k][i0][j0]==1)
   {
      for(i=0;i<=ssq-1;i++)
      {
         for(j=0;j<=ssq-1;j++)
         {
            temp[i][j]=fnp[k][i][j];
         }
      }
      fns[i0][j0]=k+1;
      boxno=(i0/size)*size+(j0/size);
      rowstart=(boxno/size)*size;
      colstart=(boxno%size)*size;
      for(i=0;i<=size-1;i++)
      {
         for(j=0;j<=size-1;j++)
         {
            fnp[k][rowstart+i][colstart+j]=0;
         }
      }
      for(i=0;i<=ssq-1;i++)
      {
         fnp[k][i][j0]=0;
         fnp[k][i0][i]=0;
      }

      c=i0;
      d=j0+1;
      if(j0==ssq-1)
      {
         if(i0==ssq-1)
         {
            c=ssq;
            d=ssq;
         }
         else
         {
            c=i0+1;
            d=0;
         }
      }
      sud(&fnp[0][0][0],&fns[0][0],c,d);
      for(i=0;i<=ssq-1;i++)
      {
         for(j=0;j<=ssq-1;j++)
         {
            fnp[k][i][j]=temp[i][j];
         }
      }
```

```
            }
         }
      }
}
```

run_c.sh :
gcc sudoku_orig.c
./a.out

./run_c.sh
0 1 0 8 9 0 4 0 0
6 0 0 0 5 4 0 0 3
4 2 0 0 0 6 7 0 0
7 0 0 4 0 0 0 3 8
0 9 2 0 0 0 5 4 0
3 6 0 0 0 1 0 0 9
0 0 8 1 0 0 0 2 6
1 0 0 9 3 0 0 0 7
0 0 5 0 7 8 0 1 0

The solution is
5 1 3 8 9 7 4 6 2
6 8 7 2 5 4 1 9 3
4 2 9 3 1 6 7 8 5
7 5 1 4 2 9 6 3 8
8 9 2 7 6 3 5 4 1
3 6 4 5 8 1 2 7 9
9 7 8 1 4 5 3 2 6
1 4 6 9 3 2 8 5 7
2 3 5 6 7 8 9 1 4

Porting C to SV

Since C and SV have a near 1 to 1 relationship for many language constructs, it is easy to just search and replace them. The following is a list of steps I followed
{} changed to begin end

int main to program tb

C routine prefixed with SV function

char datatype changed to byte

C array constant {{}} declaration changed to SV declaration '{'{}}

C #define changed to SV parameter

C does not really have a separation of parallel and sequential executing sections. All code are sequentially executing. In SV, the sequential execution code should go into any of always/initial/forever/task/function. In this example, the code was put into SV initial block.

C printf changed to SV $write

C Pointer changed to bare variable name, complex types passed using typedef

Bugs

I had arbitrarily replace all C for(i=0;;) type for loop statements to for(int i=0;;) SV statements. Little did I realize that in the code the loop variable is also used outside. Because, their was an additional declaration of the same variable with type byte, there was no syntax error. The execution however did not output correct result! After the code exited the loop the variable declared local to the loop was lost and the value of the variable with the same name turned out to be 0. So, don't use the for(**int** i=0;;) loop local variable style when porting.

C goto

The goto statement in C is not recommended for usage in good code. Yet, I used in the C code to easily get out of two for loops. Unsurprisingly, this usage turned out be not easily portable to SV. In SV, I tried different flag variable based getting out of two loops but different bugs each time. Finally, I managed to hit upon a working SV code. Based on this experience, I would recommend modifying the C code first to replace non-portable constructs with portable constructs, get a passing result and then port to SV. Mantra -

"Similar first, porting next"

```systemverilog
parameter size= 3;
parameter ssq =size*size;

typedef byte sud_options_t [ssq][ssq][ssq];
typedef byte sud_puzzle_t [ssq][ssq];

program tb;

   byte rp[ssq][ssq],cp[ssq][ssq],bp[ssq][ssq];
   byte i,j,k,l,flag,rowstart,colstart,boxno;
   byte prob[ssq][ssq][ssq];
   int in_num;
   byte s[ssq][ssq]='{
        '{0,1,0,8,9,0,4,0,0},
        '{6,0,0,0,5,4,0,0,3},
        '{4,2,0,0,0,6,7,0,0},
        '{7,0,0,4,0,0,0,3,8},
        '{0,9,2,0,0,0,5,4,0},
        '{3,6,0,0,0,1,0,0,9},
        '{0,0,8,1,0,0,0,2,6},
        '{1,0,0,9,3,0,0,0,7},
        '{0,0,5,0,7,8,0,1,0}
 };
initial begin
   $write("\nThe 9 by 9 sudoku matrix to be solved\n");
   show_sud(s);

   for(i=0;i<=ssq-1;i++)
   begin
     for(j=1;j<=ssq;j++)
     begin
       //row probables
```

```
        rp[i][j-1]=1;
        for(k=0;k<=ssq-1;k++)
        begin
            if(s[i][k]==j)
            begin
                rp[i][j-1]=0;
                break;
            end
        end

        //column probables
        cp[i][j-1]=1;
        for(k=0;k<=ssq-1;k++)
        begin
            if(s[k][i]==j)
            begin
                cp[i][j-1]=0;
                break;
            end
        end
    end

    //box probables
    rowstart=(i/size)*size;
    colstart=(i%size)*size;
    for(j=1;j<=ssq;j++)
    begin
        bp[i][j-1]=1;
        for(k=0;k<=size-1;k++)
        begin
            for(l=0;l<=size-1;l++)
            begin
                if(s[rowstart+k][colstart+l]==j)
                begin
                    bp[i][j-1]=0;
                    l=size;k=size;//goto out2loops;
                end
            end
        end
        //out2loops: ;
    end
end

for(i=0;i<=ssq-1;i++)
begin
    for(j=0;j<=ssq-1;j++)
    begin
        boxno=(i/size)*size+(j/size);
        flag=1;
        if(s[i][j])
        begin
            flag=0;
        end
        for(k=0;k<=ssq-1;k++)
        begin
```

```systemverilog
                prob[k][i][j]=(rp[i][k])&&(cp[j][k])&&(bp[boxno][k])&&flag;
            end
        end
    end
    sud(prob, s, 0, 0);
end
endprogram

function automatic void sud(
    input sud_options_t fnp,
    input sud_puzzle_t fns, input byte c, input byte d) ;
    byte i,j,k,i0,j0,rowstart,colstart,boxno;
    sud_puzzle_t temp;
    bit out2loops=0;

    if((c==ssq)&&(d==ssq))
    begin
        $write("\n C %d and d %d\n",c,d);
        $write("\n The solution is\n");
        show_sud(fns);
        $finish;
        return;
    end

    $display("C %d and d %d\n",c,d);

    show_sud(fns);

    //find the vacant square
    for(i=c;i<=ssq-1;i++)
    begin
        if(i!=c)
        begin
            d=0;
        end
        for(j=d;j<=ssq-1;j++)
        begin
            $display("Test [%d][%d]",i,j);
            if(fns[i][j]==0)
            begin
                $display("Found vacant [%d][%d]",i,j);
                out2loops=1 ; //goto out;
                i0=i;
                j0=j;
                break;
            end
        end
        if(out2loops) break;
    end
    //out:
    if(out2loops==0) begin
        i0=i;
        j0=j;
    end
```

```verilog
$display("Vacant square [%d][%d]",i,j);
if((i0==ssq)&&(j0==ssq))
begin
    $display("\nThe solution is");
    show_sud(fns);
    return;
end

//find the probable number to fill up the vacant square
for(k=0;k<=ssq-1;k++)
begin
    if(fnp[k][i0][j0]==1)
    begin
        for(i=0;i<=ssq-1;i++)
        begin
            for(j=0;j<=ssq-1;j++)
            begin
                temp[i][j]=fnp[k][i][j];
            end
        end
        fns[i0][j0]=k+1;
        boxno=(i0/size)*size+(j0/size);
        rowstart=(boxno/size)*size;
        colstart=(boxno%size)*size;
        for(i=0;i<=size-1;i++)
        begin
            for(j=0;j<=size-1;j++)
            begin
                fnp[k][rowstart+i][colstart+j]=0;
            end
        end
        for(i=0;i<=ssq-1;i++)
        begin
            fnp[k][i][j0]=0;
            fnp[k][i0][i]=0;
        end

        c=i0;
        d=j0+1;
        if(j0==ssq-1)
        begin
            if(i0==ssq-1)
            begin
                c=ssq;
                d=ssq;
            end
            else
            begin
                c=i0+1;
                d=0;
            end
        end
        $display("Call next box [%d][%d]",c,d);
        sud(fnp,fns,c,d);
        $display("later");
```

```systemverilog
        for(i=0;i<=ssq-1;i++)
        begin
           for(j=0;j<=ssq-1;j++)
           begin
              fnp[k][i][j]=temp[i][j];
           end
        end
     end
   end
endfunction

function automatic void show_sud (sud_puzzle_t s);
   for(int i=0;i<=ssq-1;i++)
   begin
      for(int j=0;j<=ssq-1;j++)
      begin
         $write("%1d ",s[i][j]);
      end
      $write("\n");
   end
   $write("\n");
endfunction
```

SV code execution

run -all

The 9 by 9 Sudoku matrix to be solved
0 1 0 8 9 0 4 0 0
6 0 0 0 5 4 0 0 3
4 2 0 0 0 6 7 0 0
7 0 0 4 0 0 0 3 8
0 9 2 0 0 0 5 4 0
3 6 0 0 0 1 0 0 9
0 0 8 1 0 0 0 2 6
1 0 0 9 3 0 0 0 7
0 0 5 0 7 8 0 1 0

C   0 and d   0

0 1 0 8 9 0 4 0 0
6 0 0 0 5 4 0 0 3
4 2 0 0 0 6 7 0 0
7 0 0 4 0 0 0 3 8
0 9 2 0 0 0 5 4 0
3 6 0 0 0 1 0 0 9
0 0 8 1 0 0 0 2 6
1 0 0 9 3 0 0 0 7
0 0 5 0 7 8 0 1 0

Test [0][0]
Found vacant [0][0]
Vacant square [0][0]
Call next box [0][1]
C   0 and d   1

```
5 1 0 8 9 0 4 0 0
6 0 0 0 5 4 0 0 3
4 2 0 0 0 6 7 0 0
7 0 0 4 0 0 0 3 8
0 9 2 0 0 0 5 4 0
3 6 0 0 0 1 0 0 9
0 0 8 1 0 0 0 2 6
1 0 0 9 3 0 0 0 7
0 0 5 0 7 8 0 1 0
```

The execution continues forward when the first possible number previously calculated for every vacant box meets the Sudoku game rules. At some empty box, filling with a possible number violates the rules and so we back track to previous box and try the next possible number for that box. As shown below. In the example, when the last box in the first row is reached, the number allowable is 2 which violates the rules. So, backtracking goes to the box before which had just been filled with 6. Now 7,8,9 are not allowable in that box, so backtracking goes back even earlier to previous empty box that is box with row, column index 0,5. It had been filled with 2, but, now found to be violating rules. So, the next possible number on that box is tried, number 7.

C   0 and d   8

```
5 1 3 8 9 2 4 6 0
6 0 0 0 5 4 0 0 3
4 2 0 0 0 6 7 0 0
7 0 0 4 0 0 0 3 8
0 9 2 0 0 0 5 4 0
3 6 0 0 0 1 0 0 9
0 0 8 1 0 0 0 2 6
1 0 0 9 3 0 0 0 7
0 0 5 0 7 8 0 1 0
```

Test [0][8]
Found vacant [0][8]
Vacant square [0][8]
Backtracking
Backtracking
Call next box [0][6]
C   0 and d   6

```
5 1 3 8 9 7 4 0 0
6 0 0 0 5 4 0 0 3
4 2 0 0 0 6 7 0 0
7 0 0 4 0 0 0 3 8
0 9 2 0 0 0 5 4 0
3 6 0 0 0 1 0 0 9
0 0 8 1 0 0 0 2 6
1 0 0 9 3 0 0 0 7
0 0 5 0 7 8 0 1 0
```

Test [0][6]
Test [0][7]
Found vacant [0][7]
Vacant square [0][7]

```
Call next box [   0][   8]
C    0 and d    8

5 1 3 8 9 7 4 6 0
6 0 0 0 5 4 0 0 3
4 2 0 0 0 6 7 0 0
7 0 0 4 0 0 0 3 8
0 9 2 0 0 0 5 4 0
3 6 0 0 0 1 0 0 9
0 0 8 1 0 0 0 2 6
1 0 0 9 3 0 0 0 7
0 0 5 0 7 8 0 1 0
```

The forward and backtrack keeps going on until hitting the final result.

```
C    8 and d    7

5 1 3 8 9 7 4 6 2
6 8 7 2 5 4 1 9 3
4 2 9 3 1 6 7 8 5
7 5 1 4 2 9 6 3 8
8 9 2 7 6 3 5 4 1
3 6 4 5 8 1 2 7 9
9 7 8 1 4 5 3 2 6
1 4 6 9 3 2 8 5 7
2 3 5 6 7 8 9 1 0
```

```
Test [   8][   7]
Test [   8][   8]
Found vacant [   8][   8]
Vacant square [   8][   8]
Call next box [   9][   9]

 C    9 and d    9

 The solution is
5 1 3 8 9 7 4 6 2
6 8 7 2 5 4 1 9 3
4 2 9 3 1 6 7 8 5
7 5 1 4 2 9 6 3 8
8 9 2 7 6 3 5 4 1
3 6 4 5 8 1 2 7 9
9 7 8 1 4 5 3 2 6
1 4 6 9 3 2 8 5 7
2 3 5 6 7 8 9 1 4
```

Porting SV to C – Longest Common Subsequence

Many times, it so happens that you need to provide a software model of your design to others who do not understand SV. For example, the software team in your company may want to create software model of the full chip for software development purpose. Your customer may want to develop their logic to talk to your design and your customer may ask for some type of model of your design. Obviously, you cannot provide the source code of your design because most companies keep it a secret.

In these cases, it is necessary to provide a model of your design in a common software language. Python may be preferred because it is getting popular and it is easy. But Python is not a native match to SV and does not typically run as fast as C. So, in this example, I have selected C for the software model. For this example, I have chosen a functional/behavioral code that does not bother with timing and clocks. If you want to port RTL to C it is even harder.

Lets consider the SV code from the LCS example earlier and then study the C code ported from the SV code. At the top the global settings like the macro N,M are set and the libraries string.h and stdio.h are included. This is in contrast to SV that can define N,M inside the "tb" program and also directly use $display and string_var.len() method which are inbuilt in SV and don't need any extra libraries like string.h or stdio.h. Also note the definition of the trace_back function inside the program tb which is allowed in SV but defining another function inside one function is not allowed in C. So, in C we needed the N,M values to be global to share it with the trace back function. Note also that in C a string of length k needs k+1 bytes because C strings are expected to be terminated with a null character '\0'. I am not sure if SV needs a null character as well. The sequence seq0 and seq1 should not be hardcoded to size 10 like seq[10]. That was a bit of lazy coding on my part that will be refactored in future. The C language also does not support the MSB:LSB or LSB:MSB format for defining arrays. Arrays in C always just declared as arr[SIZE]. This corresponds to SV arr[SIZE] equivalent to arr[0:SIZE-1]. The local int declaration in for loop is not allowed in C which is however acceptable in C++. Notice that the array trace is shared across main and trace back functions using a global definition for the trace variable. If you don't want a global variable you could use pointers.

```c
#include <string.h>
#include <stdio.h>
#define N 7
#define M 6

int max ( int a , int b ) ;
void trace_back ( int i , int j ) ;
char trace [ N+1 ] [ M+1 ] ;
char seq0[10] = "ATCTGAT" , seq1[10] = "TGCATA" ;
void main (void) {

int i,j;
int alnmat [ N+1 ] [ M+1 ] ;
int n,m;
n= strlen(seq0) ;
m = strlen(seq1);
for ( i = 0 ; i <= n ; i ++ ) {
for ( j = 0 ; j <= m ; j ++ ) {
 alnmat[i][j] = 0 ;
}
}

for ( i = 1 ; i <= n ; i ++ ) {
 for ( j = 1 ; j <= m ; j ++ ) {
  alnmat [ i ] [ j ] = max ( alnmat [ i-1 ] [ j ] , alnmat [ i ] [ j-1 ] ) ;
  if ( seq0 [ i-1 ] == seq1 [ j-1 ] ) {
   alnmat [ i ] [ j ] = max ( alnmat [ i ] [ j ] , alnmat [ i-1 ] [ j-1 ] + 1 ) ;
```

```
    }
    printf ( "alnmat [ %2d ] [ %2d ] = %d \n" , i , j , alnmat [ i ] [ j ] ) ;
    if ( alnmat [ i ] [ j ] == alnmat [ i-1 ] [ j ] ) {
      trace [ i ] [ j ] = 'U' ;
    } else if ( alnmat [ i ] [ j ] == alnmat [ i ] [ j-1 ] ) {
      trace [ i ] [ j ] = 'L' ;
    } else {
      trace [ i ] [ j ] = 'D' ;
    }
  }
}
trace_back ( n , m ) ;
  printf("\n");
}

void trace_back ( int i , int j ) {
 // printf ( "index %2d %2d Trace %3s" , i , j , trace [ i ] [ j ] ) ;
 if ( i == 0 || j == 0 ) {
  return ;
 }
 if ( trace [ i ] [ j ] == 'D' ) {
  // printf ( "Match , seq0 letter %d %d %s" , i , j , seq0 [ i-1 ] ) ;
  trace_back ( i-1 , j-1 ) ;
  // printf ( "index %2d sequence letter %s" , i-1 , seq0 [ i-1 ] ) ;
  printf ("%c", seq0 [ i-1 ] ) ;
 } else {
  if ( trace [ i ] [ j ] == 'U' ) {
   trace_back ( i-1 , j ) ;
  } else {
   trace_back ( i , j-1 ) ;
  }
 }
}

int max ( int a , int b ) {
 return ( a > b ? a : b );
}
```

Run
shell> gcc lcs.c
shell> ./a.out

TCTA

Direct Programming Interface – Hello World

It is one thing to port C code to SV. But what if you can directly use C code in SV simulation? Being able to leverage the power of C code in SV simulations will increase your verification productivity. SV offers a way to do this using Direct Programming Interface. In DPI, the C code is compiled with a C compiler and the SV code is compiled with SV simulator. The magic happens during elaboration. The compiled C object code is linked with the compiled SV object code into the final executable. This

executable behaves like regular SV simulation. The overall effect is that C code can be called like a SV function. This example is about printing the silly looking "Hello World". How embarrassing, it took me many attempts to learn that the C function should not be named "main". If it is named main the final simulation step throws a fatal error. I guess the simulator internally uses the same name main for SV!

C code

```c
#include <stdio.h>
int myfn (void)
{
  printf("Hello Worlds\n");
  return 0;
}
```

SV code

```
program tb ;

import "DPI-C" function int myfn;

int a;

initial begin
  a=myfn();
  $finish();
end

endprogram
```

The C code just prints "Hello Worlds" using the "printf" function included in the "stdio" C library. The SV code looks simple enough, except for the import statement. Let me translate that SV sentence to English - "Get me the function named myfn from the C code".

Compilation of C code
shell$ <path to Vivado bin directory>/xsc hello.c
Running compilation flow
Done compilation
Running command : <path to Vivado installation dir>/lib/lnx64.o/../../tps/lnx64/gcc-6.2.0/bin/g++ -Wa,-W -O -fPIC -m64 -shared -o "xsim.dir/work/xsc/dpi.so" "xsim.dir/work/xsc/hello.lnx64.o"   -L<path to Vivado installation dir>/lib/lnx64.o -lrdi_simulator_kernel   -lrdi_xsim_systemc   -B<path   to   Vivado   installation dir>/lib/lnx64.o/../../tps/lnx64/gcc-6.2.0/bin/../../binutils-2.26/bin/ > /dev/null 2>&1
Done linking: "xsim.dir/work/xsc/dpi.so"

The messages from the Vivado Xsc C compiler says that it used a bunch of library object files and compiled the C code into a dpi.so object file.

Linking C with SV
shell$ ~/xilinx_vivado/Vivado/2019.2/bin/xelab --svlog tb.sv --sv_lib dpi

Meaning of the command is compile and elaborate tb.sv SV code and use the library named "dpi" to get functions not defined in tb.sv. The dpi library was created in the C code compilation step.

Running: <path to Vivado>/bin/unwrapped/lnx64.o/xelab --svlog tb.sv --sv_lib dpi
INFO: [VRFC 10-2263] Analyzing SystemVerilog file "tb.sv" into library work
INFO: [VRFC 10-311] analyzing module tb
Starting static elaboration
Pass Through NonSizing Optimizer
Completed static elaboration
Starting simulation data flow analysis
Completed simulation data flow analysis
Time Resolution for simulation is 1ps
Compiling module work.tb
Built simulation snapshot work.tb

The meaning is that tb.sv SV code is compiled and object code is saved to work.tb directory.

C with SV simulation
shell$ ~/xilinx_vivado/Vivado/2019.2/bin/xsim --runall work.tb

Meaning of the command is simulate the tb object saved in work library using the xsim simulator.

source xsim.dir/work.tb/xsim_script.tcl
xsim {work.tb} -autoloadwcfg -runall
Vivado Simulator 2019.2
Time resolution is 1 ps
run -all
Hello Worlds
$finish called at time : 0 fs : File "tb.sv" Line 10
exit

Generating a sine wave LUT

Look up table for values of the sine function is used in numerically controlled oscillators to create a sine wave. If this output wave is couple to an antenna you now have a radio transmitter. For example, you could produce a 1.6 MHz wave and drive an AM radio transmitter. Calculating the value of the sine function from the input angle in hardware takes a large amount of chip area. So, the values are precomputed and stored in an LUT. This idea of storing values in LUT for a function is commonly used wherever the function is complicated enough that it would take too much time computing it on the fly in hardware or too much area to compute it. LUT idea may be applied to tan() function, permutations, combinations and many other mathematical functions. The LUT idea predates chipping. Some of the readers may have used logarithm tables to compute log10() function or may have used a table for calculating mortage interest payments. Generation of the table can be done in any higher level language like C, C++, Python or PERL. I have chosen to use SV just to show a different style of using SV to create reusable functions. The class wrapper allows for parameterizing the LUT generating function. This code is not synthesizable into a hardware. It is only for verification. A sine wave has values defined from 0 to 360 degrees or 0 to 2*pi radians. If you want N samples to denote values for the full range of the angles, then it means you have split the angle range into N equal parts. So, first sample will be for angle 0, second sample for angle 360/N, third sample 360 * 2/N and so on. In

general terms, for ith sample the angle will be 360 * (i-1)/N. The mathematical function actually only takes 2*pi, so replace 360 by 2*pi and you get the angle expression. The constant pi may be used for other purposes, say, to compute the trajectory of a rocket in a flight controller chip, so, I am collecting constants into the package thee_mathsci_consts_pkg.

```systemverilog
class lut_processing_c #( int LUT_SIZE = 32 , int LUT_DATA_WIDTH =10) ;
  typedef logic signed [LUT_DATA_WIDTH-1:0] lut_t [LUT_SIZE];
  static function void gen_sinewave_lut ( output lut_t lut) ;
    import thee_mathsci_consts_pkg::const_pi;
    real angle_rad;
    real sin_val;

    for ( int i = 0 ; i < LUT_SIZE ; i++ ) begin
      angle_rad = i*2*const_pi/LUT_SIZE;
      sin_val = $sin(angle_rad);
      if ( sin_val == 1.0 ) begin
        lut[i] = 2**(LUT_DATA_WIDTH-1)-1;
      end else begin
        lut[i] = sin_val * ( 2**(LUT_DATA_WIDTH-1));
      end
    end
  endfunction
endclass
```

The function is called like this -

```systemverilog
module tb ;

timeunit 1ns;
timeprecision 1ps;

parameter int LUT_SIZE=16;
parameter int LUT_DATA_WIDTH=12;
real angle_rad, angle_deg;
real sin_val;
logic signed [LUT_DATA_WIDTH-1:0] lut [LUT_SIZE];

parameter string outfile="SINE_LUT.txt" ;
int fd;

import thee_utils_pkg::lut_processing_c;

initial begin
 fd = $fopen(outfile, "w");
 lut_processing_c #(.LUT_SIZE(LUT_SIZE) , .LUT_DATA_WIDTH(LUT_DATA_WIDTH)) ::
gen_sinewave_lut ( lut ) ;

  for ( int i = 0 ; i < LUT_SIZE ; i++ ) begin
    $display("Angle %f rad, %f deg, sin(angle) %f", angle_rad, angle_deg, lut[i]);
    $fwrite(fd, "%d\n", lut[i]);
  end
 $fclose(fd);
end

endmodule
```

The simulation of the testbench outputs this table

```
   0
 784
1448
1892
2047
1892
1448
 784
   0
-784
-1448
-1892
-2048
-1892
-1448
-784
```

Unit 5 – SV coding for Chips

Basic Logic Elements

Wire

I would think a metal wire to be the fundamental building block of a circuit. Use the "logic" keyword of SV to describe a wire. Technically speaking, the logic datatype has more meaning than a wire. But for now just think of logic as the way to create a wire. Interconnect and net are frequently used to refer to a wire. In SV "logic" serves the purpose and is applicable to most of digital logic signals. When multiple drivers are to be used to drive a physical net then logic keyword does not model it well. Use the SV keyword "wire" for multiple drivers.

```
logic my_wire;
```

Inverter or Not gate

Inverter is the foundation of digital circuits. This is because it amplifies the signal from previous stage and restores it to full strength. This is the secret power of digital circuits that made it possible to use everything from a smartphone to a super computer.

Just use one of these statements to use an inverter. The symbol is "~". For single bit signal the operator "!" also means the same. For multibit signals the meaning of "!" is completely different from "~".

Remember to declare the wire for my_inv_out.

```
logic my_inv_out;
assign my_inv_out = ~my_wire;
```

or

```
always_comb begin
  my_inv_out = ~my_wire;
end
```

or

```
not <instance name> (<output wire>, <input wire>)
```
Example : `not i0 (inv_out, inv_in);`

In SV, many basic gates are already available for instantiation. They are called primitives. Think of the primitives as modules already coded and ready for you. Note that "not" is an inbuilt keyword in SV to denote a NOT gate. Like not keyword of SV, there are also other gates predefined – buf (buffer), or, nor, and, nand, xor and xnor.

OR gate

Notion of OR is widespread in natural languages. There are many ways to implement an OR gate. You can use the operator "|" or if-else if-else or the case statements or the SV native "or" primitive. Notice that there is an OR condition used in the flip flop edge based sensitivity list also. It basically says the flop is sensitive to clock *or* reset change.

```systemverilog
module tb ;

logic [3:0] abus, bbus, yabus, yobus, or_out_bus, and_out_bus;
logic clk, rst;
logic a,b;
logic ya,yo;
logic y1a,y1o;

//OR gate
initial begin

yo = a|b;
yobus = abus | bbus;
yo = |{a,b};
yo = |{abus};

if ( a )
  yo = 1 ;
else if ( b )
  yo = 1;
else
  yo=0;
end

always_comb
  case ({a,b})
    'b00: y1o = 0;
    'b01: y1o = 1;
    'b10: y1o = 1;
    'b11: y1o = 1;
    default: y1o=0;
  endcase


or i0 (yor1,a,b);
or i1[3:0] (or_out_bus,abus,bbus);

always_ff @(posedge clk or posedge rst) begin
  //your flops assigned here
end
```

AND gate
Just like OR gate, AND gate can be implemented in many ways. Notice the difference in the usage of if conditions for OR and AND gates. Nested if conditions implicitly mean AND gates.

```verilog
initial begin

abus = 'b1010;
bbus = 'b0110;

ya = a&b;
yabus = abus & bbus;
ya = &{a,b};
ya = &{abus};

ya=0;
if ( a )
  if ( b )
    ya = 1;
#0;
$display("and bus output %b",and_out_bus);
end

always_comb
  case ({a,b})
    'b00: y1a = 0;
    'b01: y1a = 0;
    'b10: y1a = 0;
    'b11: y1a = 1;
    default: y1a=0;
  endcase

and i2 (yand1,a,b);
and i3[3:0] (and_out_bus,abus,bbus);
```

Multiplexer Demultiplexer

Mux and Demux are used in many places to choose one item from many or to distribute one to a chosen output. There is even an operator in SV for mux "?". I don't think there is an operator for demux.

Mux and Demux have an interesting property that sets them apart from other gates. Only the select line is binary, the channels can be anything, they can even be physical things. If the input and output are analog signals then the mux is called analog mux. To model an analog mux, you can use real datatype for the input and output signals. Mux is a common occurrence in nature. Next time when you are panting and breathing through your mouth, think of what gate it represents! A mux or demux or both?

```verilog
always_comb begin
  logic y1bmux,sel,muxin,demuxa,demuxb;
  y1bmux = sel ? a : b;

  case (sel)
    'b0: y1bmux = b;
    'b1: y1bmux = a;
    default: y1bmux = b;
  endcase

  if ( sel )
    y1bmux = a;
  else
```

```
    y1bmux = b;

  if(sel) begin
     demuxa=muxin;
     demuxb=0;
  end else begin
     demuxa=0;
     demuxb=muxin;
  end

  case (sel)
     'b0: begin demuxa = 0 ; demuxb = muxin ; end
     'b1: begin demuxa = muxin ; demuxb = 0 ; end
     default: begin demuxa = 0 ; demuxb = 0 ; end
  endcase

end
```

Encoder and Decoder

Encoder and decoder are close cousins of mux and demux. They are nothing but binary-unary number system converters. In a general sense an encoder could mean anything. Say a logic that takes raw image sensor data and converts it into a JPEG file could be called a JPEG encoder. The same goes for the reverse operation of decoding JPEG to raw format. But the default is for unary-binary conversion. The decode operation is native to SV. Just set the array index! Of course, remember to initialize the array to 0. In the example shown here, encoder is realized by looking for the first index of the input bits that is set to one. Here, it is the 5th bit that is set, so the encoder will output 5.

```
always_comb begin
  logic [2:0] encoded_out;
  logic [7:0] decode_in, decoded_out;
  //test input
  decode_in = 'b00100000;

  // enocoder part
  encoded_out = 0 ;
  for ( int i = 0 ; i < 8 ; i++ ) begin
    if ( decode_in[i]) begin
       encoded_out = i;
       break;
    end
  end

  // decoder part
  decoded_out = 0 ;
  decoded_out[encoded_out] = 1'b1;
end

endmodule
```

Exercise
1. What is the difference between bitwise operators "&", "|" and logical operators "&&" and "||"?
2. Construct a 4:1 mux using the "?" operator.

3. Can you code other basic logic elements like half adder, full adder and ripple carry adder?

Flip Flops

Storage of information is integral to processing information. The simplest storage element in digital design is a latch and the second is a flip flop, or just simply a flop. There are so many examples of flops available for free, I thought why waste time rewriting it. However, there are could be some confusing points about flip flops that are worth clarifying early. In SV, a flop is described with always_ff statement. The example shows the most basic flop with only three ports - data input, clk and q output. It goes by the name delay flip flop. Standard cell libraries could refer to this as dff or DFF.

```
always_ff @(posedge clk) begin
  q <= din;
end
```

When there is only one statement inside any SV block like always, if, for, while, forever, initial the begin and end may be omitted.

```
always_ff @(posedge clk)
  q <= din;
```

To describe a flop clocked by the falling edge of a clock use negedge. A negative edge flop can be realized from a positive edge flop by adding an inverter at the clock pin. Or, this inversion can be inbuilt inside specific negative edge flop standard cells and could refer to this as dffn or DFFN or ndff.

```
always_ff @(negedge clk) begin
  q_negedge <= din;
end
```

You may wonder what is the meaning of the statement. The "@" construct needs an event, that is, something happening. And the actual happening we are interested in is a clock edge. So you could say the meaning is "always when the event positive (or negative) edge of the clock happens do the action inside the block".

To specify an asynchronous reset behavior use the following syntax. Reset is conventionally used as active low. A zero value resets the flop rather than a one value for the reset signal. Why? I do not know the full answer. I guess, keeping a zero when things are inactive is more intuitive than holding one. Also, a reset synchronizer circuit can propagate a zero value without any clock signal. Flip flops are reset to zero typically. But they could be reset to one too. This flop goes by the name resettable DFF, DFFR or dffr.

```
always_ff @(posedge clk or negedge rstn) begin
  if(~rstn) begin
    q_resettable <= 0;
  end else begin
    q_resettable <= din;
  end
end
```

When to use resettable flop and when to use non resettable flop?

Using non resettable flop is like playing with fire. Why? Because non resettable flops turn on with a zero or a one randomly when the chip is powered up. The modeling of this behavior in SV simulation is not great. So, you don't know how the system will behave. In some cases, not resetting is fine and may be even needed. Some flops are used just to hold data until something else is done. An unknown value on the data flops does not cause any problem. For example, consider a chip that outputs a recorded voice and say the speaker is driven from a small data buffer with 16 samples. The playing time for 16 audio samples may be about a few milliseconds. Do you care if there is a hiss before the recorded announcement happens? Probably not. Now, take the other extreme. Suppose, your car's electronic cruise control chip has a state bit saying cruise on or off. Say, this bit was not initialized. When you turn on your car, the chip could occasionally activate the cruise control and automatically accelerate without your command! So, the general principle is to make state and control flops resettable and data flops non resettable. If you are not sure, make everything resettable. In some cases, data could implicitly carry control information. For example, the data packet of an Internet Protocol carries the control information of the higher layer Transport Control Protocol.

I have an interesting experience with flop resets. In one project, a large data buffer was initially made resettable. The added logic delay for reset caused timing violations and we could not meet the performance target for that design. I suggested we remove reset to this buffer. It sure improved timing. But now, verification engineers where not happy. They started seeing 'X'in there simulations and they had to modify there testbench to deal with the Xs!

Scanable DFF

Flops hold most of the design state. Imagine the chip to be like a giant mess of memory bits. Every flop holds one bit. The combinational logic inside the chip is driven by the primary inputs and the internal flop outputs. In many chips the flop outputs greatly outnumber the primary chip inputs, so the values in the flop can be said to hold the state of the chip. By state, I mean that changing the values in the flop changes the functionality of the chip. In contrast, changing the value at the output of any combinational gate is quickly overwritten by the combinational logic being driven by the flops. So, combinational logic do not hold the state of the chip. They are rather the machinery of the chip. They do all the work but do not remember anything. For testing purpose, we need a way to control the state of the chip, so every flop is given a backdoor to allow overriding the output with a known value. This is scanable DFF. Typically the design starts with just the DFF, the scan part is added by the DFT/Synthesis tool during logic synthesis. So, the scanable part is only a model for understanding, you do not code the scan part in your RTL.

```
always_ff @(posedge clk) begin
  q_scanable <= scan_enable ? scan_in : din ;
end
```

Enabled DFF

The condition "hold this data unless allowed to change" is a frequently occurring theme in RTL. So standard cell libraries have a dedicated cell for that. It is coded with just one extra if condition with an enable. Like scanable DFF, it is the synthesis tool that instantiates an EDFF upon seeing the enabled logic in RTL.

EDFF to gated clock DFF

If the enable is low or the clock edge is prevented from reaching the clock port of the flop, the D to Q propagation is stopped. Controlling the D to Q transfer in the flop is effectively same as controlling the

clock pulse going into the clock port of the flop. This property is exploited by synthesis tools to automatically add clock gating cells whenever EDFFs are used. Changing a group of EDFFs to DFFs with a clock gating cell reduces power dissipation because toggling the clock pin when the enable is low only wastes power without any action happening. Rather, it is better to stop the clock.

```systemverilog
always_ff @(posedge clk) begin
  if ( enable )
    q_en <= din ;
end
```

Set/Reset DFF

Most RTL code use only one asynchronous control and it is generally reset. Sometimes, an asynchronous set may be needed too. The code shown here supports both set and reset asynchronously. If both set and reset are asserted at the same time then reset takes precedence. Preset is sometimes used interchangeably to refer to set.

```systemverilog
always_ff @(posedge clk , negedge rstn , negedge setn) begin
  if(~rstn) begin
    q_setreset <= 0;
  end else if(~setn) begin
    q_setreset <= 1;
  end else begin
    q_setreset <= din;
  end
end
```

The following code describes most of the typical controls in a flip flop. Typically the scan part is inserted automatically during synthesis. The if-else if parts for reset and set describes the clock independent asynchronous behavior of the flop. The remaining portion describes the synchronous part. The synchronous is part is where most of the logic goes. Apart from just data input (din) many other types of logic can be added at the input synchronous side. For example, a NAND gate + Flop combination standard cell can be created. The returns start diminishing beyond a few functions like, mux and enable. So, in practice only few types of functions are available as combined into a standard cell flip flop.

```systemverilog
always_ff @(posedge clk or negedge rstn or negedge setn) begin
  // ---------- Asynchronous part begin ---------//
  if(~rstn) begin
    q_all <= 0;
  end else if(~setn) begin
    q_all <= 1;
  // ---------- Asynchronous part end ---------//
  // ---------- Synchronous part begin ---------//
  end else begin
    if (scan_enable ) begin
      q_all <= scan_in;
    end else begin
      if ( sync_clr ) begin
        q_all <= 0;
      end else if ( enable ) begin
        q_all <= din;
      end
    end
  end
  // ---------- Synchronous part end ---------//
end
```

Multibit flops

It is not efficient to describe flops one bit at a time. So, SV allows for describing groups of flops. In the below example, q_multibit is an 8-bit signal. Note that reset and set values can be any constant not just '0 or '1.

```systemverilog
parameter RST_VAL = 'h 3F ;
```

```
parameter SET_VAL = 'h 10 ;

logic [7:0] q_multibit,din_multibit;

always_ff @(posedge clk or negedge rstn or negedge setn) begin
  if(~rstn) begin
    q_multibit <= RST_VAL;
  end else if(~setn) begin
    q_multibit <= SET_VAL;
  end else if ( enable ) begin
    q_multibit <= din_multibit;
  end
end
```

Programmable reset value?

Some times it is tempting to think why not make the reset value programmable. I mean, control it from the output of another flop. This would make the chip initialization customizable in the field.

```
always_ff @(posedge clk or negedge rstn or negedge setn) begin
  if(~rstn) begin
    q_multibit <= reset_value_in; //input signal
```

Bad luck! This cannot be done because this statement has no matching standard cell counterpart. So, this cannot be synthesized. This could still be used in non synthesizable behavioral code. In some special standard cell library with a special synthesis tool this could be synthesizable. Regular synthesis tools cannot. The reason is that when a flop reset value is selected to be a 0, a specific physical standard cell flop is instantiated in the layout. If the flop is initialized to 1 upon reset, a specific different physical flop is used. If you want to select the initial value of either 0 or 1 only at chip working time and not during chip synthesis time, it is like saying, I want to swap one physical cell for another after the chip is made. This is not possible! A chip once made is really set in stone.

Exercise

1. What happens if you omit the negedge rstn in the always_ff statement of a resettable flip flop?

2. How does the behavior of a set/reset flop change if the order of negedge rstn, negedge setn is swapped in the always_ff statement?
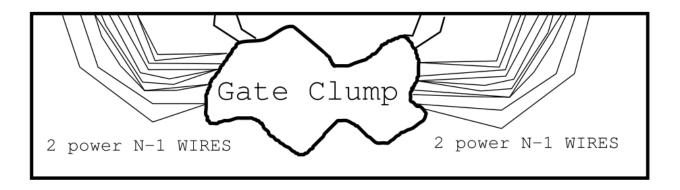
Answer

1. It becomes a synchronous reset from asynchronous reset.

2. It remains same, the reset and set precedence is set by the order of the if-else statement and not the edge event in the always_ff statement.
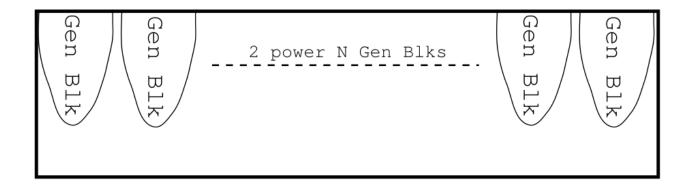
For additional building blocks of digital logic refer to previously mentioned books on SystemVerilog. Let me now jump to more complex examples.

Large Decoder

A small decoder is plain and simple. But take it to a large scale and there could be problems. Imagine a 12 to 4096 decoder, the shear number of wires needed internal to the decoder can complicate routing. The logic synthesis tool could minimize area by sharing the gates in many outputs. This way area is reduced but many wires are created. The textbook style decoder has a 12 bit select line reaching near every decoder output logic. Each output has a comparison operation that outputs a "1" when its address or index matches the 12bit address lines. There is no routing problem in this architecture of the decoder. In contrast, a centrally located shared gates architecture needs to route 4096 wires to different corners of the floorplan. This could complicate routing in floorplans that are long and skinny. Such floorplans are called high aspect ratio floorplans and benefit by splitting the wiring to different locally centered clusters.

The logic below is splits the combinational logic into a unit decoder for every output. The unit decoder outputs "1" for its address and there is a generate loop concatenating all the outputs into the full decoder outputs. The hierarchy of this approach closely matches textbook decoder implementation. When synthesized with a "keep" constraint or "maintain" hierarchy constraint for the unit decoders, you get a nicely split up logic suitable for long floorplans.


```
module decoder_unit
# (
```

```verilog
parameter WIDTH=8,
parameter LOCAL_ADDR=0
)
(
  input logic [WIDTH-1:0] addr,
  output logic dec_out
);

assign dec_out = ( addr == LOCAL_ADDR ) ;

endmodule

module decoder_large
# (
parameter WIDTH=8
)
(
  input logic [WIDTH-1:0] addr,
  output logic [2**WIDTH-1:0] dec_out
);

generate
  for ( genvar i = 0 ; i < 2**WIDTH ; i++ )
  begin : gendec
    decoder_unit #(.WIDTH(WIDTH) , .LOCAL_ADDR(i) ) dec_unit_i (
      .addr,
      .dec_out (dec_out[i])
    );
  end
endgenerate

endmodule

module tb ;

parameter WIDTH=4;

logic result ;
logic [WIDTH-1:0] addr;
logic [2**WIDTH-1:0] dec_out;

decoder_large # ( .WIDTH(WIDTH) ) dut
(
  .addr,
  .dec_out
);

initial begin
result = 1;
for(int i = 0 ; i<10;i++) begin
  addr = $urandom();
  #0;
  if ( (dec_out[addr] !== 1) || ($countones(dec_out) !== 1) ) begin
    result = 0;
    $display ("Vector fail dec_out %b addr %d ",dec_out, addr);
  end
  else
```

```
      $display ("Vector pass dec_out %b addr %d ",dec_out, addr);
end

if ( result )
  $display ("All Vectors passed");
else
  $display ("Some Vectors failed");

$finish;
end

endmodule
```

run -all
Vector pass dec_out 0000000000000010 addr 1
Vector pass dec_out 0000000010000000 addr 7
Vector pass dec_out 0000000000010000 addr 4
Vector pass dec_out 0000000000001000 addr 3
Vector pass dec_out 0000000010000000 addr 7
Vector pass dec_out 0000001000000000 addr 9
Vector pass dec_out 0000000000000010 addr 1
Vector pass dec_out 0000000100000000 addr 8
Vector pass dec_out 0000000000000001 addr 0
Vector pass dec_out 0000000001000000 addr 6
All Vectors passed

Sum of ones

Counting the number of set bits in a group is a very common operation in logic. Suppose the input is 'b 1101, then sum of ones is 3. It is preferable to use a void function to represent frequently used combinational logic. Note the use of foreach. It increases chipper productivity by freeing him from using the for loop with explicit start and end indices as in "for (int i=0;i<DP_WIDTH-1;i++)". The system task $countones( ) inbuilt in SV does the same function, but, may not be synthesizable in all logic synthesis tools. If you need to add up bits of a signal in a functional model or testbench then use $countones( ). The sum method for arrays also does the same function. But I think the sum method is not defined for logic vectors typically used in RTL. The sum method may not be synthesizable in some synthesis tools.

```
function automatic void sum_of_ones (
output logic [$clog2(DP_WIDTH):0] sum ,
input logic [DP_WIDTH-1:0] inp
);
  sum=0;
  foreach (inp[i]) begin
    sum+=inp[i];
  end
endfunction
```

Loop Unrolling

The for loops used in combinational logic may look like many feedback paths but they actually represent cascaded logic. I was confused by the for loop before and finally understood that it is actually not feedback. Cascaded representation is called unrolled loop and the process is called loop unrolling. The foreach loop can be unrolled like this -

```
sum0 = inp[0];
sum1 = sum0 + inp[1];
…
sumN_1 = sumN_2 + inp[N-1];
```

Since, the sum calculated in one step is not needed to be kept alive after the next sum is available, all the sum variables can be overwritten on itself like this -

```
sum = inp[0];
sum = sum + inp[1];
…
sum = sum + inp[N-1];
```

Note that this represents a cascade of adders, each adding one more bit to the previous adder's output.

```
sum=0;
foreach (inp[i]) begin
    sum+=inp[i];
```

Hamming Distance

Given two binary numbers the number of bit positions that have different values is defined as the hamming distance. Suppose one input is 'b1001 and another is 'b1010, the last two positions are different. Difference in binary is easy, just XOR the binary numbers to get 'b0011. So the hamming distance is 2. The code for hamming distance calculation is to XOR and then count the ones.
XOR is so synonymous with difference that it is even used in seemingly non binary areas. Suppose you have two layouts with only a slight difference, XOR operation is used on the two layouts to highlight only the difference! This is a difference operation on a 2D image.
Hamming distance operation is a measure of bit errors in a binary bit string. Suppose the binary number 10101 was transmitted and the received value was 11001, the number of errors equals the hamming distance of 2. So, hamming distance is a useful concept in digital communication systems.

```
function automatic void hamming_dist (
input logic [DP_WIDTH-1:0] inp0,
input logic [DP_WIDTH-1:0] inp1,
output logic [$clog2(DP_WIDTH)-1:0] distance
);
  sum_of_ones (.sum(distance), .inp(inp0^inp1));
endfunction
```

Gray code

An extremely commonly occurring piece of logic is Gray code. The idea is simple successive code changes differ by only bit, or the hamming distance between adjacent codes is one.

Binary to Gray

A generalized binary to gray logic is easy, just XOR the values in the given bit position with the previous bit position. So, the MSB remains the same as the binary MSB value. Other bit positions get the output of XOR operation.

```
function automatic void bin2gray (
input logic [DP_WIDTH-1:0] binary_in,
output logic [DP_WIDTH-1:0] gray_out
);
  gray_out = binary_in ^ ( binary_in>>1);
endfunction
```

Gray to Binary

Converting a Gray coded value to a binary is a bit more involved than binary to gray. Every bit position is the result of Xoring all the bits from MSB to that position. In the example code, the result from one bit position is reused for the next bit position.

```
function automatic void gray2bin (
output logic [DP_WIDTH-1:0] binary_out,
input logic [DP_WIDTH-1:0] gray_in
);
  logic xor_residue;
  xor_residue=0;
  for ( int i = DP_WIDTH-1;i>=0;i--) begin
    binary_out[i]=xor_residue^gray_in[i];
    xor_residue^=gray_in[i];
  end
endfunction
```

Unary or thermometer code

Unary number system is probably the oldest number system invented by man. In a noisy public place, how do you show the number 3 to your friend? If you speak you would not be heard over the noise. So, perhaps you would show 3 fingers held up and all remaining fingers closed. Suppose you want to represent 6 then you would write 111111, for 3 it is 111, for 2 it is 11, for 0 it is nothing. Look at the resemblance of this with the Roman numerals for up to three, I, II, III. Since this is so simple, it occurs in many electronic circuits. For example to control the amount of current passing into a load, 8 equal sized current sources would be connected to the load. By turning on nothing, 1, 11, 111, through all 8 sources, the load current can be controlled up to 8 levels. Unary code is the native output of flash ADC comparators. Because the code resembles how an analog mercury or alcohol thermometer reading, this code is commonly referred as thermometer code as well. In modern day GUI, unary code serves to display battery level using bars and in sound systems by showing the volume level by the number of lit LEDs.

Binary to unary

For a N-bit binary value, you need 2^N-1 bit positions for the unary code. We need to set as many ones in the output as the binary value says. The 2^N-1 value can get very large for even a 32 bit binary

number. So, in the function an upper limit to size of the binary/unary number is used. Note that this is made into a local parameter in the package, so, users can change it for their needs. Ideally, it would be nice if SV offered a way to parameterize a package, but, I don't know how to do that yet.

```
localparam BINARY_OF_THERM_SIZE=8;
localparam THERM_SIZE = 2**BINARY_OF_THERM_SIZE-1;

function automatic void bin2therm (
input logic [BINARY_OF_THERM_SIZE-1:0] binary_in,
output logic [THERM_SIZE-1:0] therm_out
);
  therm_out = 0 ;
  for (int i=0;i<THERM_SIZE;i++) begin
    if ( i < binary_in ) begin
       therm_out[i] = 1 ;
    end
  end
endfunction
```

Thermometer to binary

If the Nth position in thermometer code is set then it means the binary value is N. One way to decode a thermometer coded number is to search for the highest position with a 1. Another way is to count the number of ones. Thermometer to binary has a simple relationship to priority encoder. This function is closely related to the processor instructions CLZ and FF1 – count leading zeros and find first one. These processor instructions are used to speed up the operation of operating systems.

```
function automatic void therm2bin (
output logic [BINARY_OF_THERM_SIZE-1:0] binary_out,
input logic [THERM_SIZE-2:0] therm_in
);
  binary_out = 0;
  for (int i=THERM_SIZE-1;i>=0;i--) begin
    if ( therm_in[i]) begin
       binary_out = i+1 ;
       break;
    end
  end
endfunction
```

Majority function

Given N inputs a majority function outputs the value that is the majority of the inputs. This is something like voting. Where would this be used? Suppose, your car's automatic braking controller has 3 copies of the control logic for redundancy. Let us say one logic failed and continuously outputs "0" or "don't brake". The other two output "1" or "brake" when any obstacle approaches. The final decision is taken as the majority of the 3 outputs and in this case to brake. Democracy wins :) Majority logic is simple, for a given bit width count the number of ones and if the count exceeds half then there is a majority of ones.

```
function automatic void majority_fn (
output logic ones_majority ,
input logic [DP_WIDTH-1:0] inp,
```

```
input logic [$clog2(DP_WIDTH)-1:0] majority_check_size
);
  logic [DP_WIDTH-1:0] tmp;
  logic [$clog2(DP_WIDTH):0] sum;
  tmp=inp;
  sum=0;
  for (int i=0;i<DP_WIDTH ;i++) begin
    if ( i < majority_check_size ) begin
      sum+=tmp[i];
    end
  end
  if ( sum > majority_check_size / 2 ) begin
    ones_majority = 1;
  end else begin
    ones_majority = 0 ;
  end
endfunction
```

Modulo Addition

Addition that wraps around and keeps the sum within a fixed range is a common piece of logic in many designs. Modulo addition is everywhere in human culture. After counting 12 months in a year, the count restarts or wraps from 1 and does not go to 13. The same is true for many other measures like seconds, minutes, inches, angles. The following function implements modulo addition or addition with wrap around. Note that the assignment "sum = (inp0 + inp1) % modulo" will do the same thing. Based on the synthesis tool, it may synthesize to be the same as the function below but modulo operation is like a division operation and takes a large amount of area. So, Lint rule decks prohibit the use of modulo for synthesizable RTL usage, though, it is actually synthesizable. I think, that many synthesis tools may infer a divider for the modulo operation and hence use too much area. You will have to check if the direct % operator works the same for your synthesis tool. If so, it would be much more readable than using a function. The designs using the modulo addition also mostly need to know if the wrap happened or not, so, the function is set to output the wrap signal as well. Note that this is not a full mathematical modulo operation. This function will not work for the cases when the inputs are greater than the modulo value. For example, if inputs are 15, 18 and the modulo is 7 the output may not be correct. This function is expected to work correctly for inputs less than the modulo value.

```
function automatic void add_modulo_unsigned (
input logic [DP_WIDTH-1:0] inp0,
input logic [DP_WIDTH-1:0] inp1,
input logic [DP_WIDTH-1+1:0] modulo=(1'b1<<DP_WIDTH),
output logic wrapped,
output logic [DP_WIDTH-1:0] sum
);

logic [DP_WIDTH-1+1:0] sum_full;
sum_full = inp0 + inp1;
if ( sum_full >= modulo ) begin
  sum = sum_full - modulo ;
  wrapped = 1;
end else begin
  sum = sum_full;
```

```
      wrapped =0;
end

endfunction
```

Often, the second operand is just 1 and the operation has a specific name, increment.

```
function automatic void increment_modulo_unsigned (
input logic [DP_WIDTH-1:0] inp,
input logic [DP_WIDTH-1+1:0] modulo=(1'b1<<DP_WIDTH),
output logic wrapped,
output logic [DP_WIDTH-1:0] out
);

add_modulo_unsigned
(.inp0(inp),.inp1(1'b1),.modulo(modulo),.sum(out),.wrapped(wrapped));

endfunction
```

Saturating Addition

Imagine you are counting the number of cars passing through an intersection for one hour. For this purpose, suppose you are using the tally marks system on a single sheet of paper. Whenever a car passes by, you draw a small vertical line with a pencil and after 4 such lines you make a crossing line. A group of 4 vertical lines with a cross denotes 5. You keep doing this on both sides of the paper. By about 45 minutes both sides are full of tally marks and there is nowhere to make new marks. How to proceed with the counting? If you use the regular "+" operator the accumulated sum will go from maximum value to minimum value on the next increment. That would be akin to erasing the paper full of marks and then starting over. For the use case of counting cars, starting over again after erasing makes no sense. It may be somewhat better to just stop counting anymore cars and hold the sheet of paper with all the marks. This behavior of stopping further counting at the extreme is called saturation. It is a naturally occurring phenomenon. Growth of animals, transistor current voltage relationship and many other real life systems show saturation. The function for saturation is shown below. The key step is to check for the actual sum of the inputs to be greater than or equal to the maximum value. If found to exceed the maximum value saturate it to the maximum value. Usually, we also keep the maximum value as 'hFF or 'hFF….F or 'b111111111, so, the maximum value is set to '1 which means all ones in all bit positions of the variable. By using this as the default, the users don't have to explicitly set the all ones as the max value in every call of the function. The function also outputs the saturate signal to let the calling scope know that the addition has saturated and is no longer a true representation of the sum of the two inputs.

```
function automatic void add_saturate_unsigned (
input logic [DP_WIDTH-1:0] inp0,
input logic [DP_WIDTH-1:0] inp1,
input logic [DP_WIDTH-1:0] maximum='1,
output logic saturated,
output logic [DP_WIDTH-1:0] sum
);
```

```
logic [DP_WIDTH-1+1:0] sum_full;
sum_full = inp0 + inp1;
if ( sum_full > maximum ) begin
  sum = maximum ;
  saturated = 1;
end else begin
  sum = sum_full;
  saturated =0;
end

endfunction

function automatic void increment_saturate_unsigned (
input logic [DP_WIDTH-1:0] inp,
input logic [DP_WIDTH-1:0] maximum='1,
output logic saturated,
output logic [DP_WIDTH-1:0] out
);

add_saturate_unsigned
(.inp0(inp),.inp1(1'b1),.maximum(maximum),.sum(out),.saturated(saturated));

endfunction
```

Calling the function
```
add_saturate_unsigned (
.inp0(inp0),.inp1(inp1),.maximum(max_value),.sum(outp),.saturated(saturated));
```

For default '1 max value omit the operand maximum.
```
add_saturate_unsigned ( .inp0(inp0),.inp1(inp1),.sum(outp),.saturated(saturated));
```

Modulo Subtraction

Suppose you want to know how may days are in between January 25 and February 10. You could say it is 10 − 25 or -15. But it does not make sense. After adding 31, it starts making sense. Modulo subtraction is similar to modulo addition and is very useful in some designs like FIFOs. The following function implements modulo subtraction between unsigned numbers.

```
function automatic void sub_modulo_unsigned (
input logic [DP_WIDTH-1:0] inp0,
input logic [DP_WIDTH-1:0] inp1,
input logic [DP_WIDTH-1+1:0] modulo=(1'b1<<DP_WIDTH),
output logic wrapped,
output logic [DP_WIDTH-1:0] diff
);

if ( inp0 < inp1 ) begin
  diff = inp0 + modulo - inp1;
  wrapped = 1;
end else begin
  diff = inp0 - inp1;
  wrapped =0;
end

endfunction
```

Handy function for subtraction by 1

```systemverilog
function automatic void decrement_modulo_unsigned (
input logic [DP_WIDTH-1:0] inp,
input logic [DP_WIDTH-1+1:0] modulo=(1'b1<<DP_WIDTH),
output logic wrapped,
output logic [DP_WIDTH-1:0] out
);

sub_modulo_unsigned
(.inp0(inp),.inp1(1'b1),.modulo(modulo),.diff(out),.wrapped(wrapped));

endfunction
```

Saturating Subtraction

Sometimes you may want to limit the output of subtraction to 0 or more. Any subtraction going below that amount will be saturated to the minimum. Note that the saturation value is set to a variable in the code shown below.

```systemverilog
function automatic void sub_saturate_unsigned (
input logic [ DP_WIDTH-1 : 0 ] inp0 ,
input logic [ DP_WIDTH-1 : 0 ] inp1 ,
input logic [ DP_WIDTH-1 : 0 ] minimum = 0 ,
output logic saturated ,
output logic [ DP_WIDTH-1 : 0 ] diff
 ) ;

if ( inp0 <= inp1 + minimum ) begin
 diff = minimum ;
 saturated = 1 ;
end else begin
 diff = inp0 - inp1 ;
 saturated = 0 ;
end

endfunction
```

Subtraction by 1

```systemverilog
function automatic void decrement_saturate_unsigned (
input logic [ DP_WIDTH-1 : 0 ] inp ,
input logic [ DP_WIDTH-1 : 0 ] minimum = 0 ,
output logic saturated ,
output logic [ DP_WIDTH-1 : 0 ] out
 ) ;

sub_saturate_unsigned ( .inp0 ( inp ) , .inp1 ( 1'b1 ) , .minimum ( minimum ) ,
.diff ( out ) , .saturated ( saturated ) ) ;

endfunction
```

Parts from testbench

```
inp0 = 10; inp1=2; min_value=5;   sub_saturate_unsigned (
.inp0(inp0),.inp1(inp1),.minimum(min_value),.diff(outp),.saturated(saturated));
$display ( "Min value %d, Saturate sub %d - %d = %d, saturation %d", min_value,
inp0,inp1,outp,saturated);

inp0 = 14; inp1=12; min_value=7;   sub_saturate_unsigned (
.inp0(inp0),.inp1(inp1),.minimum(min_value),.diff(outp),.saturated(saturated));
$display ( "Min value %d, Saturate sub %d - %d = %d, saturation %d", min_value,
inp0,inp1,outp,saturated);

inp0 = 10; min_value=10;   decrement_saturate_unsigned (
.inp(inp0),.minimum(min_value),.out(outp),.saturated(saturated));
$display ( "Min value %d, Saturate dec %d - 1 = %d, saturation %d", min_value,
inp0,outp,saturated);

inp0 = 14; min_value=5;   decrement_saturate_unsigned (
.inp(inp0),.minimum(min_value),.out(outp),.saturated(saturated));
$display ( "Min value %d, Saturate dec %d - 1 = %d, saturation %d", min_value,
inp0,outp,saturated);
```

Output

```
run -all
Min value    5, Saturate sub  10 -    2 =    8, saturation    0
Min value    7, Saturate sub  14 -   12 =    7, saturation    1
Min value   10, Saturate dec  10 - 1 =   10, saturation    1
Min value    5, Saturate dec  14 - 1 =   13, saturation    0
```

Clip Operation

It becomes necessary to limit a computation to within a range of values. If the input is above or below the range, it is clipped to fit into this range. For example, if input is 10 and the range you want the output to be is 2 to 9, the output will be 9. In the same way clipping can happen in the lower end also, such as, when input is 1 and you want the clipping range as 2-9, the output will be 2. The function below also outputs the direction of clipping , +1 for the max side clipping and -1 for minimum side clipping.

```
function automatic void clip_unsigned (
input logic [DP_WIDTH-1:0] minimum=0,
input logic [DP_WIDTH-1:0] inp,
input logic [DP_WIDTH-1:0] maximum='1,
output logic [DP_WIDTH-1:0] out,
output logic signed [1:0] cliped
);

if ( inp < minimum ) begin
  out = minimum;
  cliped = -1;
end else if ( inp > maximum ) begin
  out = maximum;
  cliped = +1;
end else begin
  out = inp;
  cliped = 0;
```

```
end

endfunction
```

Testbench

```
begin
int mnm,mxm,cliped, dummy;
inp = 14; mnm=3; mxm=8;  clip_unsigned ( .minimum(mnm), .inp(inp), .maximum(mxm),
.out(outp),.cliped(cliped));
$display ( "clip minimum %3d input %3d max %3d output %3d cliped %3d", mnm, inp,
mxm, outp,cliped);
inp = 6; mnm=3; mxm=8;  clip_unsigned ( .minimum(mnm), .inp(inp), .maximum(mxm),
.out(outp),.cliped(cliped));
$display ( "clip minimum %3d input %3d max %3d output %3d cliped %3d", mnm, inp,
mxm, outp,cliped);
inp = 1; mnm=3; mxm=8;  clip_unsigned ( .minimum(mnm), .inp(inp), .maximum(mxm),
.out(outp),.cliped(cliped));
$display ( "clip minimum %3d input %3d max %3d output %3d cliped %3d", mnm, inp,
mxm, outp,cliped);
inp = 1; mnm=3; mxm=8;  clip_unsigned ( .minimum(mnm), .inp(inp), .maximum(mxm),
.out(outp),.cliped(dummy));
end
```

Output

```
clip minimum   3 input  14 max   8 output   8 cliped   1
clip minimum   3 input   6 max   8 output   6 cliped   0
clip minimum   3 input   1 max   8 output   3 cliped  -1
```

Rotate Operation

Rotation of a bit vector occurs in bit level processing and cryptographic algorithms. Microprocessor instruction sets usually have a rotate instruction. However, I don't know of a simple function in SV that does the same operation. The following code rotates an input bit vector of variable bit width, rotation direction and rotation amount. Left rotation once is set as the default operation. The function is implemented as one rotation left or right repeated in a for loop. At the end of the function there is a loop to select only the useful bits. This is needed because the input operand is expanded into a DP_WIDTH size vector and a left shift operation "<<" can create spurious bits in the unwanted positions. Note that using for loops to do bit vector operations can slow down simulation. Synthesis to gates may not be affected much by use of for loops. The recommended method is to use vector part select operation [MSB:LSB] for bit vector manipulation.

```
function automatic logic signed [ 1 : 0 ] get_sign (
input logic signed [ DP_WIDTH-1 : 0 ] inp
 ) ;
return ( ( inp > 0 ) ? + 2'd1 : -2'd1 ) ;
endfunction

function automatic logic is_positive (
input logic signed [ DP_WIDTH-1 : 0 ] inp
 ) ;
return ( get_sign ( inp ) == + 2'b1 ) ;
endfunction
```

```
function automatic void rotate (
input logic [ DP_WIDTH-1 : 0 ] inp ,
input logic [ $clog2 ( DP_WIDTH ) -1 + 1 : 0 ] signal_width = DP_WIDTH ,
input logic signed [ $clog2 ( DP_WIDTH ) -1 + 1 : 0 ] rotation = 1 ,
output logic [ DP_WIDTH-1 : 0 ] out
 ) ;

logic dir_left ;
logic saved_bit ;
logic [ $clog2 ( DP_WIDTH ) -1 : 0 ] rotation_amount_unsigned ;
out = inp ;
dir_left = is_positive ( rotation ) ;
rotation_amount_unsigned = dir_left ? rotation : -rotation ;
for ( int i = 0 ; i < rotation_amount_unsigned ; i ++ ) begin
 if ( dir_left ) begin
   saved_bit = out [ signal_width-1 ] ;
   out = out << 1'b1 ;
   out [ 0 ] = saved_bit ;
 end else begin
   saved_bit = out [ 0 ] ;
   out = out >> 1'b1 ;
   out [ signal_width-1 ] = saved_bit ;
 end
end

for ( int i = 0 ; i < DP_WIDTH ; i ++ ) begin
  if ( i >= signal_width ) begin
    out [ i ] = 0 ;
  end
end

endfunction
```

Calling the function
Note the omission of default function arguments for simplicity.

```
begin
logic [15:0] inp, out ; int rotation, signal_width;
inp = 'b00011;   rotate (.inp(inp),.out(out));
$display ( "rotate left input %b output %b", inp, out);
inp = 'b0110011; rotation = 3; signal_width=7 ; rotate
(.inp(inp),.rotation(rotation), .signal_width(signal_width),.out(out));
$display ( "rotate left input %b rotation %3d width %3d output %b", inp, rotation,
signal_width, out);

inp = 'b00011;   rotate (.inp(inp),.rotation(-1),.out(out));
$display ( "rotate right input %b output %b", inp, out);
inp = 'b0110110011; rotation = -6; signal_width=7 ; rotate
(.inp(inp),.rotation(rotation), .signal_width(signal_width),.out(out));
$display ( "rotate right input %b rotation %3d width %3d output %b", inp, rotation,
signal_width, out);
end
```

Result

```
rotate left input 0000000000000011 output 0000000000000110
rotate left input 0000000000110011 rotation   3 width   7 output 0000000000011011
rotate right input 0000000000000011 output 1000000000000001
rotate right input 0000000110110011 rotation  -6 width   7 output 0000000001100110
```

Round Operation

In fixed point arithmetic, rounding bits is a regularly occurring operation. Rounding occurs in signal processing, financial calculations and many other places that internally calculate with higher precision and then output a value at a lower precision. First piece of the function separates the input into integer part and a fractional part. Later, the fractional part is checked to see if it is 0.5 or more. If so, the integer part is incremented. An interesting situation occurs when the fractional part is exactly 0.5. Suppose, you are asked to round 4.5 you may intuitively say 5. Is 5 the correct answer? Actually, 4.5 is equidistant from 4 and 5. Either 4 or 5 is a good answer. The notion of rounding direction is introduced to break this tie. Rounding towards infinity means 4.5 becomes 5. Rounding towards zero means 4.5 becomes 4. There are many variations to resolve the tie. Here only two options are provided, towards zero or infinity. In the function shown, the lsb_width denotes the fractional part. If you recollect the binary number system, you may see that the weights for the bit positions right side of the decimal point goes as 2^{-1}, 2^{-2}, and so on. So, 0.5 is 2^{-1} or binary 0.1000.

```verilog
function automatic void round_lsb_unsigned (
input logic [DP_WIDTH-1:0] inp,
input logic [$clog2(DP_WIDTH)-1:0] lsb_width=3,
input logic towards=1,
output logic [DP_WIDTH-1+1:0] out
);
  logic [DP_WIDTH-3:0] ls_bits,mid_value,inc_value;

  ls_bits = 0 ;
  mid_value=0;
  mid_value[lsb_width-1]=1'b1;
  inc_value = mid_value << 1'b1;
  out = inp;

  for(int i=0;i<lsb_width;i++) begin
   ls_bits[i]=inp[i];
   out[i] = 0 ;
  end

  if ( ls_bits > mid_value) begin
     out+=inc_value;
  end else if ( ls_bits == mid_value ) begin
    if ( towards == 1 ) begin
      out+=inc_value;
    end
  end

endfunction
```

Testbench

```verilog
logic [ DP_WIDTH -1 : 0 ] inp , lw ; logic towards ;
```

```
logic [ DP_WIDTH -1 + 1 : 0 ] out ;
inp = 'b11101010 ; lw = 4 ; round_lsb_unsigned ( .inp ( inp ) , .lsb_width ( lw ) ,
.out ( out ) ) ;
$display ( "Rounding last %3d bits of input %b = %b" , lw , inp , out ) ;
inp = 'b11101100 ; lw = 3 ; round_lsb_unsigned ( .inp ( inp ) , .lsb_width ( lw ) ,
.out ( out ) ) ;
$display ( "Rounding last %3d bits of input %b = %b" , lw , inp , out ) ;
inp = 'b11101100 ; lw = 3 ; towards = 0 ; round_lsb_unsigned ( .inp ( inp ) ,
.lsb_width ( lw ) , .towards ( towards ) , .out ( out ) ) ;
$display ( "Rounding last %3d bits of input %b towards %b = %b" , lw , inp ,
towards , out ) ;
```

Result

```
Rounding last   4 bits of input 0000000011101010 = 00000000011110000
Rounding last   3 bits of input 0000000011101100 = 00000000011110000
Rounding last   3 bits of input 0000000011101100 towards 0 = 00000000011101000
```

Linear Feedback Shift Register (LFSR)

LFSR is a simple and useful piece of logic. It is useful to create long patterns with small silicon area. It is also used to spread the signal power of a data transmission into the full available frequency band. Lets understand the LFSR word by word. We all know what a shift register is. Here the meaning is a serial shift register. Linear feedback means the contents of the shift register are combined in a linear way to create the feedback signal. In digital logic, XOR operation can mean addition which is considered linear. Which bits to feed to the xor operation is specified by a polynomial. If you have the polynomial as $1+x+x^5$ then the feedback bit the xor reduction of LFSR indices 1 and 5. The choice of polynomial involves deep mathematics that I don't understand. But, you can abstract the mathematics into the desirable property of maximality, meaning full length. Unlike a binary counter that goes full 2^N states for a N flop counter, an LFSR goes one state less. This is because if the LFSR is set to all zero the xor gate output also becomes all zero, the bit shifted in is 0 and the LFSR gets stuck in all zero state. So, a maximal polynomial makes the LFSR sequence go the full 2^N-1 states for a N flop LFSR.

The function presented here is more resuable than XOR reducing outputs of hardcoded register indices. This function is placed in the ehgu_basic_pkg. The testbench for an LFSR needs to test if the sequence of states are maximal. The maximality test is carried out by setting a 2^N size array to all zeros and whenever the LFSR reaches a value from 1 to 2^N that position is set to 1. If after running the LFSR for the full 2^N-1 there are positions in the hit array with 0s that means those values did not occur and the polynomial is not maximal. The polynomials to use are available from the website -

http://users.ece.cmu.edu/~koopman/lfsr/index.html

For size 4, the polynomial is 'hC/0xC and 'h9/0x9. The code here uses the bit reveresed format. So, the polynomial is 0x3 and 0x9. Any other polynomial does not give maximal length, as can be seen for the polynomial 0xA in the testbench.

```
function automatic logic [ DP_WIDTH-1 : 0 ] lfsr_logic (
input logic [ DP_WIDTH-1 : 0 ] polynomial ,
input logic [ DP_WIDTH-1 : 0 ] lfsr_reg ,
```

```systemverilog
input logic [ $clog2 ( DP_WIDTH ) -1 : 0 ] lfsr_width = 3
) ;
 logic shift_in;
 logic [ DP_WIDTH-1 : 0 ] lfsr_next;

 shift_in = ^(lfsr_reg & polynomial);
 lfsr_next = lfsr_reg>>1;
 lfsr_next[lfsr_width-1]=shift_in;
 return lfsr_next;

endfunction
```

Testbench

```systemverilog
program tb ;
parameter WIDTH = 4 ;

logic result ;
logic [ WIDTH-1 : 0 ] lfsr1,lfsr2,lfsr3, polynomial1,polynomial2,polynomial3 ;
typedef logic [ WIDTH-1 : 0 ] lfsr_seq_t [$ ] ;
lfsr_seq_t sq1,sq2,sq3;

import ehgu_basic_pkg::lfsr_logic;

initial begin
  result = 1 ;
  polynomial1 = 'h3;//'h C ;
  polynomial2 = 'h9;//'h 9 ;
  polynomial3 = 'h5;//'h A ;
  lfsr1=1; lfsr2 = 1 ; lfsr3 = 1;

  for ( int i = 0 ; i < 20 ; i ++ ) begin
   lfsr1 = lfsr_logic ( .lfsr_reg ( lfsr1 ) , .polynomial ( polynomial1 ), .lfsr_width(WIDTH) ) ;
   lfsr2 = lfsr_logic ( .lfsr_reg ( lfsr2 ) , .polynomial ( polynomial2 ), .lfsr_width(WIDTH) ) ;
   lfsr3 = lfsr_logic ( .lfsr_reg ( lfsr3 ) , .polynomial ( polynomial3 ), .lfsr_width(WIDTH) ) ;
    $display ( "SNo.%3d LFSR %b LFSR %b LFSR %b" , i, lfsr1,lfsr2,lfsr3 ) ;
    sq1[i]=lfsr1 ; sq2[i]=lfsr2 ; sq3[i]=lfsr3 ;
  end

 result &= is_lfsr_maximal(polynomial1);
 result &= is_lfsr_maximal(polynomial2);
 if ( result )
  $display ( "Vectors passed" ) ;
 else
  $display ( "Some Vectors failed" ) ;

 result &= !is_lfsr_maximal(polynomial3);

 if ( result )
  $display ( "All Vectors passed" ) ;
 else
  $display ( "Some Vectors failed" ) ;

 $finish ;
end
```

```
function automatic bit is_lfsr_maximal (
input logic [WIDTH-1:0] polynomial
  );

logic [ WIDTH-1 : 0 ] lfsr;
logic [ WIDTH-1 : 0 ] lfsr_list [2**WIDTH];

lfsr_list = '{default:0};
lfsr = 1;
for ( int i = 0 ; i < 2**WIDTH ; i ++ ) begin
  lfsr = lfsr_logic ( .lfsr_reg ( lfsr ) , .polynomial ( polynomial ), .lfsr_width($bits(polynomial)) ) ;
  lfsr_list[lfsr] = 1;
end

for ( int i = 1 ; i < 2**WIDTH ; i ++ ) begin
  if (lfsr_list[i]==0) begin
    $display("NOT MAXIMAL");
    $display("LFSR sequence generated with polynomial %h missing %d",polynomial, i);
    return 0 ;
  end
end

return 1;

endfunction

endprogram


run -all

SNo.   0 LFSR  1000 LFSR  1000 LFSR  1000
SNo.   1 LFSR  0100 LFSR  1100 LFSR  0100
SNo.   2 LFSR  0010 LFSR  1110 LFSR  1010
SNo.   3 LFSR  1001 LFSR  1111 LFSR  0101
SNo.   4 LFSR  1100 LFSR  0111 LFSR  0010
SNo.   5 LFSR  0110 LFSR  1011 LFSR  0001
SNo.   6 LFSR  1011 LFSR  0101 LFSR  1000
SNo.   7 LFSR  0101 LFSR  1010 LFSR  0100
SNo.   8 LFSR  1010 LFSR  1101 LFSR  1010
SNo.   9 LFSR  1101 LFSR  0110 LFSR  0101
SNo.  10 LFSR  1110 LFSR  0011 LFSR  0010
SNo.  11 LFSR  1111 LFSR  1001 LFSR  0001
SNo.  12 LFSR  0111 LFSR  0100 LFSR  1000
SNo.  13 LFSR  0011 LFSR  0010 LFSR  0100
SNo.  14 LFSR  0001 LFSR  0001 LFSR  1010
SNo.  15 LFSR  1000 LFSR  1000 LFSR  0101
SNo.  16 LFSR  0100 LFSR  1100 LFSR  0010
SNo.  17 LFSR  0010 LFSR  1110 LFSR  0001
SNo.  18 LFSR  1001 LFSR  1111 LFSR  1000
SNo.  19 LFSR  1100 LFSR  0111 LFSR  0100
Vectors passed
```

NOT MAXIMAL
LFSR sequence generated with polynomial 5 missing   3
All Vectors passed
Exercise
A 4 bit LFSR would need a 16 size array. A 128 bit LFSR cannot possibly be tested for maximality as in the testbench. How can you be confident that a long size LFSR is maximal?

Generic CRC

If a marketing guy had named the Cyclic Redundancy Check he would have named it XOR Magic. CRC is truly a magical error detection method. It is so sensitive that even a single bit change in many bits creates a vastly different signature. Whenever you are using the Internet, it is the CRC that works on the front line identifying corrupted information. The logic of the CRC is fairly simple. Take data input one bit at a time into a symbolic CRC register, xor the contents of the register as instructed by a parameter called polynomial. Shift the register by one bit and keep doing the operation again and again until all bits of the data are consumed. What is left at the end in the register is the CRC value of the data. It is a special signature of the data. The hard part of the CRC is in understanding the math behind the CRC. Verifying CRC is also hard because general chippers are not trained to think in terms of the CRC logic. The following code captures the behavior of CRC logic. Full verification is left as exercise.

One function describes the CRC logic with the XOR, data bit input and the feedback connections. The second function describes how data bits are input into the CRC register logic.

```
program tb ;
parameter WIDTH = 3 ;

logic result ;
logic [ WIDTH-1 : 0 ] data , crc , crc_expected , polynomial ;
logic [ WIDTH-1 : 0 ] data_vector [ ] ;

initial begin
   result = 1 ;
   polynomial = 'b 110 ;
   data_vector = '{ 'b100 , 'b100 } ;
   crc_expected = 'b001 ;
   crc = 0 ; // '1 ;
   for ( int i = 0 ; i < data_vector.size ( ) ; i ++ ) begin
     crc = generic_crc ( .data ( data_vector [ i ] ) , .crc ( crc ) , .polynomial (
polynomial ) , .iterations ( 3 ) , .data_lsb_first ( 0 ) ) ;
   end
   #0 ;
   if ( crc !== crc_expected ) begin
     result = 0 ;
     $display ( "Vector fail crc %h expected %h " , crc , crc_expected ) ;
   end else
     $display ( "Vector pass crc %h expected %h " , crc , crc_expected ) ;

   if ( result )
    $display ( "All Vectors passed" ) ;
   else
    $display ( "Some Vectors failed" ) ;
```

```systemverilog
    $finish ;
end

function automatic logic [ WIDTH-1 : 0 ] generic_crc (
input logic [ WIDTH-1 : 0 ] data ,
input logic data_lsb_first = 1 ,
input logic [ WIDTH-1 : 0 ] crc ,
input logic [ $clog2 ( WIDTH ) -1 : 0 ] iterations = 8 ,
input logic [ WIDTH-1 : 0 ] polynomial
 ) ;

 if ( data_lsb_first == 0 ) begin
   data = { << { data } } ;
 end

 for ( int i = 0 ; i < iterations ; i ++ ) begin
   $display ( "crc %b , data %b" , crc , data ) ;
   crc = generic_crc_logic ( .d ( data [ 0 ] ) , .crc ( crc ) , .polynomial
( polynomial ) ) ;
   data = data >> 1 ;
 end

 return crc ;

endfunction

function automatic logic [ WIDTH-1 : 0 ] generic_crc_logic (
input logic d ,
input logic [ WIDTH-1 : 0 ] crc ,
input logic [ WIDTH-1 : 0 ] polynomial
 ) ;

if ( crc [ 0 ] ^ d ) begin
  crc >>= 1 ;
  crc ^= polynomial ;
end else
  crc >>= 1 ;

return crc ;

endfunction

endprogram
```

Result

```
run -all
crc 000 , data 001
crc 110 , data 000
crc 011 , data 000
crc 111 , data 001
crc 011 , data 000
crc 111 , data 000
Vector fail crc 5 expected 1
Some Vectors failed
```

Exercise

1. Verify that for two input data bits named D0 and D1 produces CRCs C0 and C1. Is the CRC of data CRC(D0 XOR D1) == (C0 XOR C1) ?

2. Download wireshark software tool and capture an Ethernet frame. Check if the FCS present on the frame matches the CRC computed with the functions here.

Short Gray Code

Have you ever wondered does Gray code always have to be in powers of 2? An asynchronous FIFO may need 624 words which is not a power of two. The nearest power of 2 larger than that figure is 1024. Using 1024 in place of 624 wastes the other 400 words. Is there a way to use a modification on the gray code theme that can handle non power of 2?

Using Constrained Random

I first thought why not leverage the SV constraint solver to solve a hamming distance 1 condition for the length you provide. To get gray codes we need two conditions. The first one that comes to your mind is hamming distance 1. There is also a less obvious constraint of uniqueness, that is, we don't any number to repeat in the code list. A gray code can also have more bits per code word than the minimum needed, so that, the constraint solver gets more leeway. For example, you could have 3 bits to represent a 4 length code like this – 000,100,110,010 is also a gray code. The hamming distance condition needs a separate function because the native conditions permitted inside a constraint block is limited. The hamming distance condition also has a modulus operation to check for the last code and first code relationship. Unfortunately, the simulator could not solve for anything beyond 6 length and even after solving for the 6 length code, I got wrong outputs.

```
program tb ;
logic result ;
import thee_utils_pkg :: print_test_result ;
import thee_utils_pkg :: create_test_result_file ;
localparam MAX_CODE_LEN = 16 ;
typedef bit [ $clog2 ( MAX_CODE_LEN ) -1 : 0 ] code_t ;

class short_gray ;
 int len ;
 int width;
 rand code_t gcode [ MAX_CODE_LEN ] ;

 function void set_len ( input int l , int w ) ;
  len = l ;
  width = w;
 endfunction

 function bit my_unique ( ) ;
  for ( int i = 0 ; i < len ; i ++ )
   for ( int j = 0 ; j < len ; j ++ )
    if ( i != j )
```

```systemverilog
      if ( gcode [ i ] == gcode [ j ] )
        return 0 ;
    return 1 ;
  endfunction

  function bit hamming_dist ( ) ;
    int hdist;
    $display("New call");
    for ( int i = 0 ; i < len ; i ++ ) begin
      hdist = $countones ( gcode [ i ] ^gcode [ ( i + 1 ) %len ]);
      $display("hamming_dist between %b and %b is %d",gcode[i],gcode [ ( i + 1 ) %len ],hdist);
      if ( hdist != 1 )
        return 0 ;
    end
    return 1 ;
  endfunction

  // constraint no_repeat { unique { gcode [ 0 : MAX_CODE_LEN-1 ] } ; } ;
  constraint size_limit { foreach (gcode[i]) gcode[i]<2**width; };
  constraint no_repeat { my_unique ( ) == 1 ; } ;
  constraint unit_dist { hamming_dist ( ) == 1 ; } ;

endclass : short_gray

static short_gray sg,sg_bak;

initial begin
 result = 1 ;

 repeat (100) begin
  sg=new();
// sg.set_len ( .l(10),.w(4) ) ;
// sg.set_len ( .l(6),.w(4) ) ;
  sg.set_len ( .l(6),.w(3) ) ;
  if ( sg.randomize ( ) ) begin
    $display ( "Randomize passed , Gray code created" ) ;
    result = 1 ;
    show(sg);
    break;
  end else begin
    $display ( "Randomize failed , Gray code not created" ) ;
    result = 0 ;
  end
 end
 print_test_result ( result ) ;
 create_test_result_file ( result ) ;
 $finish ;
end

function void show ( short_gray obj ) ;
  $display ( "Gray code that was created" ) ;
  for ( int i = 0 ; i < obj.len ; i ++ ) begin
    $write ( "%b " , obj.gcode [ i ] ) ;
  end
  $display ( "" ) ;
```

endfunction

endprogram

It was pretty insightful to watch the solver in action. I have never seen how a solver works but always just used the final outputs.

```
run -all
New call
hamming_dist between 0000 and 0000 is      0
New call
hamming_dist between 0011 and 0100 is      3
New call
hamming_dist between 0001 and 0111 is      2
...
...
New call
hamming_dist between 0110 and 0111 is      1
hamming_dist between 0111 and 0011 is      1
hamming_dist between 0011 and 0001 is      1
hamming_dist between 0001 and 0101 is      1
hamming_dist between 0101 and 0100 is      1
hamming_dist between 0100 and 0110 is      1
Randomize passed , Gray code created
Gray code that was created
0111 0010 0000 0010 0010 0101 // <- donno why wrong set printed after solving constraints well!
Test Pass
```

Delete a Block of Codes

I did arrive at a working idea during a long train journey. The idea is that in Gray code if you remove 2,4,6 etc.. codes from in between the code list the result list still maintains the hamming distance 1 criteria. So, in the 3 bit code list 000,001,011,010,110,111,101,100 if you remove the 001,011 you still get a hamming distance 1 code -

000,010,110,111,101,100 (for length 6)

Later, I googled for non power of 2 gray code and hit upon on an EE times post that provided a better approach to a generic code.

https://www.eetimes.com/how-to-generate-gray-codes-for-non-power-of-2-sequences/

The suggestion from the crowd was to generate a mapping for the code by deleting the middle codes from the full code list. If you want two less skip the middle two codes, for 4 less skip the 4 middle codes and so on. I have created the mapping of the full and short gray codes and these functions on presented below. These functions being reusable are placed in the Ehgu basic package. Given a desired length of Gray code not a power of 2, there are two constants that are needed for creating the shorter

version of the full length Gray code. So, I have created a user defined data type called shortgray_constants_t. Next a function calculates the two constants and returns them when given the desired short code length. Note that the short code length should be an even number. I think it may not be possible to create a Gray code for odd numbered length. Using the constants, any given binary number can be translated to a full size binary number that can be directly used in a binary to Gray function.

For example, if you desire a 18 length Gray code the nearest full length is 32. So, you need to skip 32-18=14 codes. And the 14 codes need to be skipped symmetrically around the middle of the full size code. So, the mid point of 32 is 16 and the skip should start from 7 codes before 16. The skip point is 9. But the counting starts from 0 and this is accounted as a less by 1 on the 9. At the end the code becomes 0,1, to 8 (9 codes), 9 to 22 (skipped), 23 to 31(9 codes). So, the binary number 11 in the short gray code sequence maps to 8+3rd code or 8,23,24,25, that is 25. If this example is not very clear, I think the testbench and the messages from the testbench will be clear.

```systemverilog
typedef struct {
logic [ DP_WIDTH-1 : 0 ] base ;
logic [ DP_WIDTH-1 : 0 ] skip ;
 } shortgray_constants_t ;

function automatic shortgray_constants_t get_shortgray_constants (
  input byte code_length
) ;
shortgray_constants_t sg_constants ;
logic [ DP_WIDTH-1 : 0 ] pwr2_code_length = 2 ** $clog2 ( code_length ) ;
logic [ DP_WIDTH-1 : 0 ] mid_point = pwr2_code_length / 2 ;

sg_constants.skip = pwr2_code_length - code_length ;
sg_constants.base = mid_point - sg_constants.skip / 2 - 1 ;
return sg_constants ;
endfunction

function automatic logic [ DP_WIDTH-1 : 0 ] get_shortgray_skip (
 input byte regular_bin ,
 input shortgray_constants_t sg_constants
) ;
 logic [ DP_WIDTH-1 : 0 ] bin_skipped ;
 if ( regular_bin > sg_constants.base ) begin
  bin_skipped = regular_bin + sg_constants.skip ;
 end else begin
  bin_skipped = regular_bin ;
 end
 return bin_skipped ;
endfunction

function automatic logic [ DP_WIDTH-1 : 0 ] get_shortgray_unskip (
 input byte bin_skipped ,
 input shortgray_constants_t sg_constants
) ;
 logic [ DP_WIDTH-1 : 0 ] bin_unskipped ;
 if ( bin_skipped > sg_constants.base ) begin
```

```
    bin_unskipped = bin_skipped - sg_constants.skip ;
  end else begin
    bin_unskipped = bin_skipped ;
  end
  return bin_unskipped ;
endfunction
```

The testbench works through all the numbers of the short length Gray code and checks if the next Gray code differs by exactly one bit (Hamming distance is 1). The other property of all codes being unique is not checked. The codes will be unique because we are translating unique binary numbers to the Gray code.

```
program tb ;
logic result ;
import thee_utils_pkg :: print_test_result ;
import thee_utils_pkg :: create_test_result_file ;
import ehgu_basic_pkg :: bin2gray ;
import ehgu_basic_pkg :: hamming_dist ;
import ehgu_basic_pkg :: shortgray_constants_t ;
import ehgu_basic_pkg :: get_shortgray_constants ;
import ehgu_basic_pkg :: get_shortgray_skip ;

initial begin
  shortgray_constants_t sg_constants ;
  byte base , bin_next , bin_skipped , bin_skipped_next , skip , gray , gray_next , code_length ,
hdist ;
  result = 1 ;
  code_length = 18 ;
  sg_constants = get_shortgray_constants ( .code_length ( code_length ) ) ;
  $display ( "Code len %d , base %d" , code_length , base ) ;
  for ( byte i = 0 ; i < code_length ; i ++ ) begin
    bin_skipped = get_shortgray_skip ( i , sg_constants ) ;
    bin2gray ( .binary_in ( bin_skipped ) , .gray_out ( gray ) ) ;
    bin_next = ( i + 1 ) %code_length ;
    bin_skipped_next = get_shortgray_skip ( bin_next , sg_constants ) ;
    bin2gray ( .binary_in ( bin_skipped_next ) , .gray_out ( gray_next ) ) ;
    hamming_dist ( .inp0 ( gray ) , .inp1 ( gray_next ) , .distance ( hdist ) ) ;
    $display ( "bin %3d , bin next %3d , bskip %3d , bskip next %3d , gray %b , gray next %b , Ham
D %3d" ,
    i , bin_next , bin_skipped , bin_skipped_next , gray , gray_next , hdist ) ;
    if ( hdist != 1 ) result = 0 ;
  end

  print_test_result ( result ) ;
  create_test_result_file ( result ) ;
  $finish ;
end

endprogram
```

```
run -all
Code len  18, offset    8
bin   0, bin next   1, bskip   0, bskip next   1, gray 00000000, gray next 00000001, Ham D    1
bin   1, bin next   2, bskip   1, bskip next   2, gray 00000001, gray next 00000011, Ham D    1
```

```
bin    2, bin next   3, bskip    2, bskip next    3, gray 00000011, gray next 00000010, Ham D   1
bin    3, bin next   4, bskip    3, bskip next    4, gray 00000010, gray next 00000110, Ham D   1
bin    4, bin next   5, bskip    4, bskip next    5, gray 00000110, gray next 00000111, Ham D   1
bin    5, bin next   6, bskip    5, bskip next    6, gray 00000111, gray next 00000101, Ham D   1
bin    6, bin next   7, bskip    6, bskip next    7, gray 00000101, gray next 00000100, Ham D   1
bin    7, bin next   8, bskip    7, bskip next    8, gray 00000100, gray next 00001100, Ham D   1
bin    8, bin next   9, bskip    8, bskip next   23, gray 00001100, gray next 00011100, Ham D   1
bin    9, bin next  10, bskip   23, bskip next   24, gray 00011100, gray next 00010100, Ham D   1
bin   10, bin next  11, bskip   24, bskip next   25, gray 00010100, gray next 00010101, Ham D   1
bin   11, bin next  12, bskip   25, bskip next   26, gray 00010101, gray next 00010111, Ham D   1
bin   12, bin next  13, bskip   26, bskip next   27, gray 00010111, gray next 00010110, Ham D   1
bin   13, bin next  14, bskip   27, bskip next   28, gray 00010110, gray next 00010010, Ham D   1
bin   14, bin next  15, bskip   28, bskip next   29, gray 00010010, gray next 00010011, Ham D   1
bin   15, bin next  16, bskip   29, bskip next   30, gray 00010011, gray next 00010001, Ham D   1
bin   16, bin next  17, bskip   30, bskip next   31, gray 00010001, gray next 00010000, Ham D   1
bin   17, bin next   0, bskip   31, bskip next    0, gray 00010000, gray next 00000000, Ham D   1
Test Pass
```

Creating A LUT At Compile Time

Look up tables are used for implementing complex functions that do not have a simple logical implementation. The power operation like 3^n consumes significant area. Many Lint rule decks disallow designers from using functions like power because that would make the chip exceed its area budget. So, designers resort to using a LUT. But these LUTs need to be populated with entries somehow. One way to do that is to use a PERL or Python script to calculate the function values and then write those into a file. This approach is simple. However, it adds one manual step into the design process. What if the design needs powers of 5 instead? Then the PERL script has to be run again. To reduce manual interventions like these, LUT generation can be coded right in the synthesizable design. The key is to define a typedef that will contain the LUT and then create a function that will populate a variable of the LUT type. This way, the design becomes more reusable when the function needs to change and the design intent is saved in the design itself. There is one shortcoming however. Every compilation of the design will have to recreate the table wasting compile time in the process. Note that the LUT generation function is run by the compiler and it is not synthesized into hardware. Compile time is a rather cheap commodity to buy reusability. So, this is a good trade. Also, you can compile the LUT into a library just once and link the compiled LUT library during compilation of your tests. This way you get both reusability and compile time saving.

```systemverilog
package config_pkg ;

localparam POWER_ENTRIES=10;
typedef logic [15:0] power3_lut_t [POWER_ENTRIES];

function automatic power3_lut_t create_power3_table ();
  power3_lut_t lut;
  for (int i = 0 ; i<POWER_ENTRIES; i++) begin
    lut[i] = 3**i;
  end
  return lut;
endfunction

parameter power3_lut_t power3=create_power3_table();

endpackage

module tb ;
```

```
import config_pkg::*;

initial begin
  foreach ( power3[i] ) begin
    $display ( "3 power %d = %d", i, power3[i] );
  end
end

endmodule

run -all
3 power   0 =       1
3 power   1 =       3
3 power   2 =       9
3 power   3 =      27
3 power   4 =      81
3 power   5 =     243
3 power   6 =     729
3 power   7 =    2187
3 power   8 =    6561
3 power   9 =   19683
```

Uniform Random Real Number Generation

SV LRM defines a $urandom() function that outputs a uniform random integer. But, I am not aware of any $urandom_real() available. Lets make one!

The range of lowest to highest real number needs to be programmable for greater applicability of the function. We can start with the existing integer function and scale it to the desired floating point (or real) range. Note that it is not a true real random number distribution. It is rather a real number distribution with discrete steps. For example, 3.99993 may be a valid random number generated but 3.99994 may not be generated at all. These numbers are used only as an example. The actual non existent numbers is left for analysis by you.

Note that alternatively, you could have randomized a class containing floating point member with constraint for lower and upper bounds. My belief is that class randomization may take more simulator resources than direct scaling of $urandom, though, I may be wrong.

```
function automatic real urand_range_real
(
  input real low,
  input real high
);
  const int unsigned MAX_VALUE = '1;
  int unsigned tmp;
  real out;
  tmp=$urandom();
  out = ( low + (tmp*1.0/MAX_VALUE)*(high-low));
  return out;
endfunction
```

Exercise
1. Create a class based real random generator

2. Use inline randomization as in #1.

Add Tolerance Function

In analog circuits, for example in a resistor instance, there is a need to create a value that is distributed with a mean of say X and tolerance of +/-T %. The following function does the job and is packaged into thee_utils_pkg. Note that the probability distribution is likely to be uniform.

```systemverilog
function automatic real add_tolerance ( input real aval , input real tol_pcnt ) ;
 real res ;
 res = aval ;
 if ( $urandom_range ( 1 ) )
 res *= ( 1 + urand_range_real ( 0 , tol_pcnt / 100.0 ) );
 else
 res *= ( 1 - urand_range_real ( 0 , tol_pcnt / 100.0 ) );
 return res ;
endfunction
```

Clock Generator – Basic Version

The simple clock generator has many secrets within it. Let me unravel them one by one. The clock generator idea is simple, set a signal to 1 for half the time and 0 another half of the time. The interesting part is in making it reusable and in modeling the inaccuracies of a real clock. In the following module, all the parameters have a default value, so, instantiating the module with just the clock port connected is sufficient to output a 1GHz clock. For other frequencies, set the FREQ parameter. The clock generator type parameter is for adding higher accuracy models in future. The following code shows the full clock generator module with and without jitter modeling. For now, ignore the jitter parameter and the if-generate block inside "jitter_only".

```systemverilog
module thee_clk_gen_module
#(
parameter real FREQ=1000,
parameter real FREQ_UNIT=1.0e6,
parameter string CLK_GEN_TYPE="basic",
parameter real PP_JITTER_PPM=100
)
(
output logic clk
);

timeunit 1ns;
timeprecision 1ps;

    realtime half_period, period_in_local_units, period_in_seconds;
    real freq_in_Hz;
    generate
      if ( CLK_GEN_TYPE == "basic" ) begin
         : ckgen_basic
        initial begin
          freq_in_Hz = FREQ * FREQ_UNIT;
          period_in_seconds = 1.0 / freq_in_Hz ;
```

```systemverilog
            period_in_local_units = period_in_seconds / 1e-9 ;
            half_period = period_in_local_units /2.0;
            clk = 0;
            forever begin
              #(half_period);
              clk=0;
              #(half_period);
              clk=1;
            end
          end
        end else if ( CLK_GEN_TYPE == "jitter_only" ) begin
          : ckgen_jitter_only
        end

    endgenerate

endmodule
```

Instantiation

```systemverilog
thee_clk_gen_module clk_gen (.clk(clk));
```

Modules are used in RTL style of coding and testbench code achieves the same effect using fork-join_none blocks. The following shows a reusable task that is "spawned" in a fork-join_none block. Note the use of ref qualifier for clock signal inside the task. This is to be able to continuously output the changes in the clock in the calling hierarchy. If output qualifier was used the clock will never be output because the task uses forever loop and output modifier in a task means the task copies the internal value onto the output when the task completes. Note that I could not get the task form of the clock generator running in Vivado, though, I think this is supposed to work.

```systemverilog
task automatic clk_gen_basic
(
  input real freq=1000,
  input real freq_unit=1.0e6,
  ref logic clk
);

  realtime half_period, period_in_local_units, period_in_seconds;
  real freq_in_Hz;
  freq_in_Hz = freq * freq_unit;
  period_in_seconds = 1.0 / freq_in_Hz ;
  period_in_local_units = period_in_seconds / 1e-9 ;
  half_period = period_in_local_units /2.0;

  clk = 0;
  forever begin
    #(half_period);
    clk=0;
    #(half_period);
    clk=1;
  end

endtask
```

Calling the task

```
initial
   fork
    clk_gen_basic (.clk(clk));
   join_none;
```

Many a times, I feel lazy and just use the following one liner. It works without the clk variable being initialized because the clock is declared bit type that is automatically initialized to 0 at start of simulation. If you want to use the logic type (prefered), remember to initialize to 0 or 1.

```
bit clk;
always begin #HALF_PERIOD; clk = ~clk; end

logic clk;
initial begin
  clk =0 ;
  forever begin #HALF_PERIOD; clk = ~clk; end
end
```

The coding style that initializes the clock in one block and toggles in another block is confusing and better avoid it. Not recommended method -

```
initial begin
  clk = 0;
end

always begin
  #HALF_PERIOD;
  clk = ~clk;
end
```

Clock Generator With Jitter

Modeling jitter is a prerequisite to the design and analysis of logic that control and manipulate clocks. Jitter is nothing but randomness in the time point of the rise and fall edges of the clock signal with reference to an ideal clock signal that produces edges at fixed time points. Error in time position of the edges indirectly mean error in the delay that generates the next edge from the previous edge. The following code creates a jittery clock signal by varying the clock period every clock cycle. Note that the average error is 0. Only instantaneous error is within the parameter of PP_JITTER_PPM. This peak-to-peak jitter parts per million parameter sets the upper limit on the randomness of the jitter produced. The code is part of the module for basic clock generator.

```
        end else if ( CLK_GEN_TYPE == "jitter_only" ) begin
          : ckgen_jitter_only
          import thee_utils_pkg::urand_range_real;
          real this_ppm;
          initial begin
            freq_in_Hz = FREQ * FREQ_UNIT;
            period_in_seconds = 1.0 / freq_in_Hz ;
            period_in_local_units = period_in_seconds / 1e-9 ;
            half_period = period_in_local_units /2.0;
```

```
        clk = 0;
        forever begin
          this_ppm = 1.0 + urand_range_real(-PP_JITTER_PPM,PP_JITTER_PPM)/1e6;
          #(half_period*this_ppm);
          clk=0;
          #(half_period*this_ppm);
          clk=1;
        end
      end
    end
```

Instantiation

```
thee_clk_gen_module
#(.FREQ(REF_FREQ/1e6),.CLK_GEN_TYPE("jitter_only"),.PP_JITTER_PPM(10000)) ref_gen
(.clk(clk_ref));
```

Clock Frequency Meter

When dealing with clocking circuits like oscillator, PLL, DLL and clock recovery, it is important to automatically calculate the output clock frequency of the system. The following SV code models a clock frequency detection circuit. This model does not have a practical analog in hardware. It is a model mostly inspired by the features afforded by the simulator. The measurement is simple, the time difference between two successive rising edges is the period of a repeating signal. The inverse of the period is frequency.

```
initial begin
  @(posedge clk1);
  get_binary_clk_freq_local ( .freq_in_hertz(fout));
  $display ("Frequency of clk1 is %e", fout);
  @(posedge clk2);
  $display ("Frequency of clk2 is %e", fout);
  $finish;
end

 task automatic get_binary_clk_freq_local
 (
  output real freq_in_hertz
 );

  realtime first_rise_edge, second_rise_edge;
  real period_in_seconds;

  @(posedge clk1);
  first_rise_edge = $realtime();

  @(posedge clk1);
  second_rise_edge = $realtime();

  period_in_seconds = (second_rise_edge - first_rise_edge) * (1e-9);
  freq_in_hertz = 1.0 / period_in_seconds;

 endtask
```

Generalized task

It is useful to create a generalized task that can be imported as needed. The following code shows how to do that. But unfortunately, Vivado simulator hangs in the first posedge statement inside the function. There could be a bug here in the code or the simulator is not possibly handling the "const ref" declaration correctly.

```systemverilog
package thee_sig_analysis_pkg;

  timeunit 1ns;
  timeprecision 100ps;

 task automatic get_binary_clk_freq
  (
    const ref logic clk,
    output real freq_in_hertz
  );

    realtime first_rise_edge, second_rise_edge;
    real period_in_seconds;


    repeat (1) @(posedge clk);
    first_rise_edge = $realtime();

    @(posedge clk);
    second_rise_edge = $realtime();

    period_in_seconds = (second_rise_edge - first_rise_edge) * 1e-9;
    freq_in_hertz = 1.0 / period_in_seconds;

  endtask
endpackage
```

Module implementation may be used is the frequency output is needed continuously.

```systemverilog
module thee_clk_freq_meter
  (
    input logic clk,
    output real freq_in_hertz
  );
  timeunit 1ns;
  timeprecision 100ps;

    realtime first_rise_edge, second_rise_edge;
    real period_in_seconds;
  always begin
    repeat (1) @(posedge clk);
    first_rise_edge = $realtime();
    @(posedge clk);
    second_rise_edge = $realtime();

    period_in_seconds = (second_rise_edge - first_rise_edge) * 1e-9;
    freq_in_hertz = 1.0 / period_in_seconds;
  end
endmodule
```

https://electronics.stackexchange.com/questions/56193/clock-period-using-verilog-code
https://groups.google.com/forum/m/#!topic/comp.lang.verilog/I5Ipr1hNWXY

Frequency meter for analog signal

Real valued signal does not have an edge that can trigger @(posedge) statement. It has to be derived differently. Suppose the real signal crosses zero every cycle, then the time difference between two zero crossings can be computed as the period. Note that this method does not work if the signal has noise

that can cause spurious zero crossings. A more robust method may be is to find the Fourier Transform of the analog signal and look at the peaks or do some noise filtering before or during period measurement.

```verilog
initial begin
realtime crossing, prev_crossing, wave_period;
real prev_integral;
   forever @(ana_in) begin
      if ( ana_in > 0 && prev_integral <= 0 ) begin
         prev_crossing = crossing;
         crossing = $realtime();
         wave_period = crossing - prev_crossing ;
         $display ( "Wave period input %t", wave_period);
      end
      prev_integral = ana_in;
   end
end
```

Exercise
In general, an analog frequency meter may need first and second derivative to check for peaks, Possible steps
Save last 3 changes into variables prev2, prev1, current
differences current – diff0 = prev1 and diff1 = prev1 – prev2 are representative of first derivative
A peak is characterized by diff0 being positive and diff1 being negative.

Effect of noise and resolution
If the sampling is very fine then the first difference and second difference can both turn out zero which escapes the peak detection steps defined. One way is to set the time step large enough so that the differences are large enough. Noise in the signal may cause spurious peaks to be detected in the previous peak detection scheme. To suppress noise, we could use a low pass filter before doing the peak detection or use hysteresis or increase the sampling time step.

Duty Cycle Meter

One of the important properties of a clock signal is its duty cycle. It becomes important when using half cycle logic in a design. As a verification component, a duty cycle measurement code is a useful tool. The following code works by recording the time at which negative and positive edges occur in a signal and outputs the duty cycle as (t_{fall_edge} – t_{rise_edge})/period. This operation can optionally be repeated using the measurement window parameter.

```verilog
module thee_clk_duty_meter
 #(parameter MEAS_WINDOW=1)
 (
   input logic clk,
   output real duty
 );
 timeunit 1ns;
 timeprecision 1ps;

 realtime first_rise_edge, fall_edge, second_rise_edge;
 real period_in_seconds;
 real sum_of_periods;

 initial begin
```

```
   forever begin
    sum_of_periods = 0 ;
    @(posedge clk);
    first_rise_edge = $realtime();
    repeat (MEAS_WINDOW) begin
     @(negedge clk);
     fall_edge = $realtime();
     @(posedge clk);
     second_rise_edge = $realtime();
     duty = (fall_edge - first_rise_edge) / (second_rise_edge - first_rise_edge ) ;
     first_rise_edge = second_rise_edge;
    end
   end
 end
endmodule
```

Exercise
The MEAS_WINDOW is not being used to average the duty cyle. Add code to find average duty cycle.

Comparison of floating point numbers

Floating point numbers are tricky. As you keep doing operations on them the rounding errors can accumulate. If you expect 3.751, you may get 3.75099999. So, the regular equal to comparison operator cannot be applied directly. This in some ways indirectly mimics real world floating point or real number signals. Suppose you measure the output of a 12V battery, you may get 12.1 for one battery and 12.32 for another battery. To check if the battery is ok, it may be specified as 12V with 5% tolerance. So a good battery will read from 12 − 0.6V to 12 + 0.6V. The following task implements the real number comparison operation.

```
task automatic check_approx_equality
(
input real inp ,
input real expected ,
input real tolerance = 0.01 ,
input real tolerance_for_zero = 0.01 ,
output bit result
) ;

if ( expected > 0 ) begin
  if ( inp > expected * ( 1.0 + tolerance ) ) begin
   result = 0 ;
  end else if ( inp < expected * ( 1.0-tolerance ) ) begin
   result = 0 ;
  end else begin
   result = 1 ;
  end
end else if ( expected == 0 ) begin
  if ( inp > tolerance_for_zero ) begin
   result = 0 ;
  end else if ( inp < tolerance_for_zero ) begin
   result = 0 ;
  end else begin
```

```
      result = 1 ;
    end
  end else begin
    if ( inp < expected * ( 1.0 + tolerance ) ) begin
      result = 0 ;
    end else if ( inp > expected * ( 1.0-tolerance ) ) begin
      result = 0 ;
    end else begin
      result = 1 ;
    end
  end
  endtask
```

Cycle Delay

Delaying a signal for a fixed number of clock cycles is a super common construct used in RTL. A generalized delay module is very useful. The logic is simple, data of a parameterized width gets in every cycle and data of same width comes out after a fixed number of clock cycles. The delay module also goes by the name shift register or parallel in parallel out shift register. When the bit width is one, this reduces to a serial in serial out shift register.

```
module ehgu_dly
#(
parameter DELAY = 1 ,
parameter WIDTH = 1
)
(
input logic clk,
input logic rstn,
input logic [WIDTH-1:0] din,
output logic [WIDTH-1:0] dout
);

logic [WIDTH-1:0] dly_stages [0:DELAY-1] ;

always_ff @( posedge clk, negedge rstn ) begin
  if ( !rstn ) begin
    dly_stages <= '{default:0};
  end else begin
    dly_stages[0] <= din;
    for ( int i = 1 ; i < DELAY ; i++ ) begin
    dly_stages[i] <= dly_stages[i-1];
  end
  end
end

assign dout = dly_stages[DELAY-1];

endmodule
```

Synchronizer

A synchronizer is fundamentally not too different from the delay chain module. There are however some additional features needed for a synchronizer. The clock domain crossing inside the module

needs to be clearly identified. This helps in disabling specific flops for gate level simulation. For this purpose the module defined here uses a name that may not collide with other names even in large SoC projects that have thousands of module names. I have chosen the characters qzx which look odd so that it may not match with any name in any other module. To get all the synchronizers in the design, you may use a tool that can read the netlist, such as PrimeTime. Then search for the instances of the module ehgu_synqzx. Print the instance names into a file. Process the file to add timing disable commands on the first flop. It could be something like this. The astrix (or star) means all indices.

<disable_timing_check> <hierarchical name from top.sync instance name.sq_i.dly_stages[0][*]>

More information about synchronizer related ideas area here

http://www.sunburst-design.com/papers/CummingsSNUG2008Boston_CDC.pdf
Section modeling uncertainty talks about matching the SV model to real circuit behavior.

Newer modeling ideas are available through the Verification Academy CDC course
https://verificationacademy.com/sessions/modeling-metastability

```systemverilog
module ehgu_synqzx
#(
`ifndef SYNTHESIS
  parameter type T = realtime,
  parameter T MAX_DELAY =100ps,
`endif
parameter STAGES = 2 ,
parameter WIDTH = 1
)
(
input logic clk,
input logic rstn,
input logic [WIDTH-1:0] d_presync,
output logic [WIDTH-1:0] d_sync
);

logic [WIDTH-1:0] d_jittered;

`ifndef SYNTHESIS
  thee_rand_busdly    #( .WIDTH (WIDTH), .T(T), .MAX_DELAY(MAX_DELAY)   ) rdly_i
  (
    .bus_in(d_presync),
    .bus_out(d_jittered)
    );
`else
  assign d_jittered = d_presync;
`endif

ehgu_dly #(.DELAY(STAGES), .WIDTH(WIDTH)) sq_i ( .clk , .rstn , .din(d_jittered) ,
.dout(d_sync));

endmodule
```

The clock domain crossing is modeled with a bus delay that produces random jitter in each bus bit line. This model can be disabled for synthesis by declaring `define SYNTHESIS for logic synthesis applications.

```systemverilog
module thee_rand_busdly
  #(
  parameter WIDTH = 8,
  parameter type T = realtime,
  parameter T MAX_DELAY =100ps
  )
  (
    input logic [WIDTH-1:0] bus_in,
    output logic [WIDTH-1:0] bus_out
  );
  timeunit 10ps;
  timeprecision 1ps;
  parameter int unsigned MAX_VALUE = '1;

  T this_delay;
  int unsigned randval;

  generate
    if ( MAX_DELAY==0 ) begin : nodly
      assign bus_out =  bus_in ;
    end else begin : gendly
      for ( genvar i =0 ; i < WIDTH ;i++) begin : perbitdly
        always @(*) begin
          randval = $urandom();
          this_delay = T' ( (1.0 * MAX_DELAY ) * (1.0 * randval) / MAX_VALUE );
          $display("index %d rand val %d rand dly %8t Max delay %8t", i, randval,
this_delay,MAX_DELAY);
          #(this_delay);
          bus_out[i] = bus_in[i];
        end
      end
    end
  endgenerate

endmodule
```

Testbench

```systemverilog
module tb ;

parameter STG=2;
parameter WIDTH=4;

logic clk,rstn;
logic result ;
logic [WIDTH-1:0] din;
logic [WIDTH-1:0] dout;

thee_clk_gen_module #(.FREQ(100)) clk_gen (.clk(clk));

initial begin
import thee_utils_pkg::toggle_rstn;
```

```
repeat (1) @(posedge clk);
result = 1;
toggle_rstn(.rstn(rstn),.rst_low(9.4ns));

for(int i = 0 ; i < 4; i++) begin
  din=$urandom();
  $display("Setting data input to %b at %t",din, $realtime());
  repeat (STG +1) @(posedge clk);
  if ( dout!==din )
    result = 0;
end

if ( result )
  $display ("All Vectors passed");
else
  $display ("Some Vectors failed");

$finish;
end

initial
  $monitor("Data changed %b at %t",dut.d_jittered,$realtime());

ehgu_synqzx #(.T(time), .MAX_DELAY(1000ps), .STAGES(STG), .WIDTH(WIDTH)) dut
(
.clk ,
.rstn ,
.d_presync(din) ,
.d_sync(dout)
);

endmodule
```

run -all

Data changed xxxx at		0			
Setting data input to 1111 at		106000			
index	0 rand val	695458240 rand dly	20 Max delay	100
index	1 rand val	1222655389 rand dly	30 Max delay	100
index	2 rand val	2862289336 rand dly	70 Max delay	100
index	3 rand val	168261054 rand dly	0 Max delay	100

```
Data changed 1xxx at         106000
Data changed 1xx1 at         106020
Data changed 1x11 at         106030
Data changed 1111 at         106070
Setting data input to 1001 at         130000
```

index	0 rand val	3061407513 rand dly	70 Max delay	100
index	1 rand val	1258606864 rand dly	30 Max delay	100
index	2 rand val	1298299593 rand dly	30 Max delay	100
index	3 rand val	1642982646 rand dly	40 Max delay	100

```
Data changed 1001 at         130030
```

Exercise

Sometimes it may not be clear if the asynchronous signal input to the synchronizer is properly registered with flops in the sending clock domain. If not, the signal will be glitchy because of combinational logic. This can happen if the signal is output from an encrypted source file and there is reason to suspect the encrypted design could have bad CDC modeling. In that case, can you add a parameter named PROTECT_IN to a new module ehgu_sync_protected? Let the default value of this parameter be 0. If set to 1, the synchronizer module should add a register stage clocked by the input clock and then use a regular synchronizer to transfer to the destination domain.

Reset Synchronizer

A two stage reset synchronizer is almost a standard for synchronizing an asynchronous reset signal. Why two stages? That is a common interview question. I have got many answers, often from the interviewers. Two stages definitely works well for most of the cases. But the number of stages needs to be derived from the acceptable probability of failure in synchronization. The terms that contribute to failure are the input reset event rate, the speed of the flip flops and the clock period available for recovery from metastability. So, if you are running the synchronizer with a clock in the GHz range, two may not be sufficient. So, I have made the reset synchronizer from a shift register with parameterized stage count. A point worth noting in any synchronizer is that the output has an uncertainty of 1 cycle. In other words, for a N stage synchronizer the output comes in N or N-1 cycles.

For more details refer to this paper -

http://www.sunburst-design.com/papers/CummingsSNUG2003Boston_Resets.pdf

Metastability analysis of a synchronizer is in section 10.6.2 of the book "CMOS VLSI Design A Circuits and Systems Perspective"

```
module ehgu_rst_sync
#(parameter STAGES=2)
(
input logic clk,
input logic rstn_in,
output logic rstn_out
);

ehgu_dly #(.DELAY(STAGES)) din_delay_i ( .clk , .rstn(rstn_in) , .din(1'b1) ,
.dout(rstn_out));

endmodule
```

Exercise
Does metastability occur in the reset synchronizer module and how? Is there a need to identify the CDC flop of the reset synchronizer for gate simulation?
Solution
Metastability may occur in the flip flops of the reset synchronizer when reset is de-asserted. In this case rstn=1. Assume that rstn was held low for long enough for all the flops to settle to a logic 0. If rstn goes high at the same time as the clock edge, the first flop in the chain decides which event happened first – clock edge or reset edge. As positions of the two edges approach each other, the output is

undetermined. This problem goes away in the next cycle because a logic 1 is loaded into the flop. The subsequent stages resolve this undermined logic 1 to a clean 0 or clean 1. Even a single cycle of unknown can corrupt a simulation. So, there is a need to disable timing checks on the first flop for the clock-reset timing arc.

Edge detectors

While designing control logic it is frequently necessary to perform an operation only on the first cycle of a signal going high. The first cycle being high is misleadingly called rising edge or leading edge. The rising edge detector is very simple. If a second signal is created by delaying this signal by one cycle, on the rising edge of the signal, the delayed copy would still be low because it lags by one cycle. Decoding this condition gives a pulse output for one cycle after the rise edge. Fall edge is similar, just, change the decoding logic. OR-ing the rise and fall outputs gives an output for any edge. The toggle or any edge signal generation is also referred to as level-to-pulse.

```systemverilog
module ehgu_edges
(
input logic clk,
input logic rstn,
input logic din,
output logic fedge,
output logic redge,
output logic toggle
);

logic din_delayed ;

ehgu_dly din_delay_i ( .clk , .rstn , .din , .dout(din_delayed));

assign redge = ~din_delayed & din ;
assign fedge = din_delayed & ~din ;
assign toggle = fedge | redge ;

endmodule
```

Often one is interested in just the rise or fall edge, so it is worth creating specific versions.

```systemverilog
module ehgu_fedge
(
input logic clk,
input logic rstn,
input logic din,
output logic fedge
);

logic din_delayed ;

ehgu_dly din_delay_i ( .clk , .rstn , .din , .dout(din_delayed));

assign falledge = ~din & din_delayed ;

endmodule
```

```systemverilog
module ehgu_redge
(
input logic clk,
input logic rstn,
input logic din,
output logic redge
);

logic din_delayed ;

ehgu_dly din_delay_i ( .clk , .rstn , .din , .dout(din_delayed));

assign redge = din & ~din_delayed ;

endmodule
```

Counter

This is probably the most common block after adder. A counter can be thought off as an adder with output feeding back to itself with a carry input of 1. You could define many more controls to a counter. For now, I have defined a basic one with asynchronous reset, synchronous clear/reset and enable. Counters also typically have a load signal and then a load value to alter the count value dynamically.

```systemverilog
module ehgu_cntr
#( parameter WIDTH=3 )
(
input logic clk,
input logic rstn,
input logic sync_clr,
input logic en,
output logic [WIDTH-1:0] cnt
);

always_ff @( posedge clk, negedge rstn ) begin
  if ( !rstn ) begin
   cnt <= 0;
  end else if ( sync_clr) begin
   cnt <= 0 ;
  end else if ( en ) begin
   cnt <= cnt + 1'b1 ;
  end
end

endmodule
```

Exercise
A simple timer is counter running in reverse. Code a module that takes as inputs a timer enable, load, load_value and outputs timeout when the counter down counts to zero from loaded value.

Clock Gator

Clock gating technique saves power by turning off clock to unused parts of the design. The only objective in clock gating is to stop clock pulses from reaching flops downstream of the gating logic. If

it was a logic signal it is too simple a problem, just AND it with the enable signal. But clock is a special signal. The pulse width of the clock signal should be preserved. It is necessary to AND the clock signal only when the clock is already inactive, zero. A negative edge flop or a negative latch is used to get this behavior. Note that even if the code is done with negative edge, it is still to be checked in STA for setup and hold timing checks at the AND gate because routing and buffering delays of the final design can affect this timing. These checks are part of the STA tools. Since, all these cares are not worth having during top level SoC development, clock gating cell or clock gator is prepackaged and made available from the standard cell vendor's library.

Model of a clock gating cell

```systemverilog
module ehgu_clkgate
(
input logic clkin,
input logic en,
output logic clkout
);

logic en_negedge;

always_ff @( negedge clkin )
   en_negedge <= en;

assign clkout = en_negedge & clkin ;

endmodule
```

To properly reset the design, first the input clock has to be available and then the reset should be asserted.

Testbench

The TB instantiates the clock gator, a clock generator and has tasks to operate and check for the clocks. Note the use of a string input to the gator task. I normally use binary data type for such on or off control, but, using string type increase readability at the cost of increasing compile and execution time. The clock detection behavioral code needs a bit explaining. Imagine two counters set off in parallel. The first one runs off always running input clock and the second runs off gated clock. If the clock is on, the gated counter increments from starting value of 0. If clock is off the counter does not increment and reads zero at the end of the measurement period. The initial values of 0 are hidden here because the int data type is initialized automatically to 0. Using implicit initialization is OK in testbench code. But, uninitialized values are a problem in synthesizable code.

```systemverilog
module tb ;

logic clkin;
logic clkout;
bit result;
logic clken;
bit clk_det;
```

```verilog
thee_clk_gen_module clk_gen (.clk(clkin));

ehgu_clkgate clkgate
(
 .clkin ,
 .clkout ,
 .en(clken)
);

initial begin
  result = 1 ;
  repeat (2) @(posedge clkin);

  operate_clk_gate("ungate");
  clock_detect (clk_det) ;
  if ( clk_det )
   result &= 1;
  else
   result = 0 ;

  operate_clk_gate("gate");
  clock_detect (clk_det) ;
  if ( clk_det == 0 )
   result &= 1;
  else
   result = 0 ;


  operate_clk_gate("ungate");
  clock_detect (clk_det) ;
  if ( clk_det )
   result &= 1;
  else
   result = 0 ;

  if ( result ) begin
   repeat (3) $display ( "PASS");
  end else begin
   repeat (3) $display ( "FAIL");
  end

   $finish;
end

task operate_clk_gate (
input string cmd="gate"
);
   @(posedge clkin);
   if ( cmd == "gate")
     clken = 0 ;
   else
     clken = 1 ;

   $display("Operating clock gate, Command %s, Clock enable %b",cmd, clken);
```

```
    repeat (2) @(posedge clkin);
endtask

task automatic clock_detect (
output bit detected
);

int cnt_ungated, cnt_gated;
fork
  repeat (3) begin
    @(posedge clkin);
    cnt_ungated++;
  end
  repeat (3) begin
    @(posedge clkout);
    cnt_gated++;
  end
join_any
$display("Clock edge count ungated %3d, gated %3d",cnt_ungated, cnt_gated);

if ( cnt_gated > 0 )
  detected = 1;

$display("Clock detection result: %b",detected);

endtask

endmodule
```

Testbench output

run -all

Operating clock gate, Command ungate, Clock enable 1
Clock edge count ungated   3, gated   3
Clock detection result: 1
Operating clock gate, Command gate, Clock enable 0
Clock edge count ungated   3, gated   0
Clock detection result: 0
Operating clock gate, Command ungate, Clock enable 1
Clock edge count ungated   3, gated   3
Clock detection result: 1
PASS

Clock gating has to be disabled during regular scan testing. This is done with another input test enable that sets the clock gating AND gate controlling input to be always 1.

https://semiengineering.com/knowledge_centers/low-power/techniques/dft-and-clock-gating/

Exercise
The verification for the clock gating cell was not complete. It does not check if there are any glitches on the output of the clock gating AND gate. Can you compare the output pulse widths using $realtime()

utility system task and get the output pulse width and then compare against the fixed half period of input clock or dynamically compare against the corresponding input clock? Provide a margin of 1% for comparing with expected pulse width.

Clock Divider

Many clock frequencies are used in a chip to strike the right balance between power and performance. Low frequency is used for low speed control applications and high frequency is used for high bandwidth data transmission and high throughput computation. The clock input frequency is however only one or two. Every other clock needed inside the chip is generated from this one clock input. The typical way is to use a clock multiplier that uses the input clock and creates a multiple of the input frequency. Lower frequencies are generated by dividing this high frequency internal clock. The following module does this clock division. The divider module should more technically be called a whole number clock divider because it cannot divide by fractional numbers like 4.3 or 5.2 but only 2,3,4,..,N. It is generally called integer clock divider or just plain clock divider. To understand how this works, we need see what is meant by a divided clock. Suppose we need to divide a clock by 2, we need to reduce the toggles on the output clock by half compared to the input clock. One way to do this is to create a counter that keeps track of the input toggles and then create the output toggle for every fixed number of input toggles. The clock division process can be described in one line as follows, "output one toggle for every N input toggles". The clkout part in the module implements the outputting of reduced toggles. The combinational output signal is registered with a flop to make it glitch free because the greater than equal to comparison can create glitches if output directly.

```systemverilog
module ehgu_clkdiv
#( parameter DIVISION=2 )
(
input logic clkin,
input logic rstn,
input logic en,
output logic clkout
);
import ehgu_basic_pkg::increment_modulo_unsigned;

localparam WIDTH = $clog2(DIVISION);

logic [WIDTH-1:0] cnt, cnt_comb;
logic nc;

always_comb begin
   increment_modulo_unsigned (.inp(cnt),.modulo(DIVISION),.out(cnt_comb),.wrapped(nc));
end

always_ff @( posedge clkin, negedge rstn ) begin
 if ( !rstn ) begin
   cnt <= 0;
 end else if ( en ) begin
   cnt <= cnt_comb;
 end
end

always_ff @( posedge clkin, negedge rstn ) begin
 if ( !rstn ) begin
```

```
    clkout <= 0;
  end else begin
    clkout <= cnt >= (DIVISION/2);
  end
end

endmodule
```

The testbench for the clock divider module instantiates two instances of the clock divider, one dividing by the default 2 and the other dividing by 5. A behavioral frequency meter logic calculates the frequency of the clock by counting the time between clock edges. The measured frequency from the meter is compared with the expected output frequency to show pass or fail.

```
module tb ;
import thee_utils_pkg :: check_approx_equality ;
logic clkin ;
logic clkout0 ;
logic clkout1 ;
real fout0 , fout1 ;
bit result0 , result1 ;
logic rstn ;

thee_clk_gen_module clk_gen ( .clk ( clkin ) ) ;

thee_clk_freq_meter fmeter0 ( .clk ( clkout0 ) , .freq_in_hertz ( fout0 ) ) ;
thee_clk_freq_meter fmeter1 ( .clk ( clkout1 ) , .freq_in_hertz ( fout1 ) ) ;

ehgu_clkdiv clkdiv0
 (
 .clkin ,
 .rstn ,
 .en ( 1'b1 ) ,
 .clkout ( clkout0 )
 ) ;

ehgu_clkdiv # ( .DIVISION ( 5 ) ) clkdiv1
 (
 .clkin ,
 .rstn ,
 .en ( 1'b1 ) ,
 .clkout ( clkout1 )
 ) ;

initial begin
  repeat ( 2 ) @ ( posedge clkin ) ;
  rstn = 0 ;
  repeat ( 10 ) @ ( posedge clkin ) ;
  rstn = 1 ;
  repeat ( 10 ) @ ( posedge clkout0 ) ;
  repeat ( 10 ) @ ( posedge clkout1 ) ;
  $display ( " Clkout frequencies 0 %e , 1 %e" , fout0 , fout1 ) ;
  check_approx_equality ( .inp ( fout0 ) , .expected ( 0.5e9 ) , .result ( result0 ) ) ;
  check_approx_equality ( .inp ( fout1 ) , .expected ( 2e8 ) , .result ( result1 ) ) ;
  if ( result1 == 1 && result0 == 1 ) begin
```

```
      repeat ( 3 ) $display ( "PASS" ) ;
    end else begin
      repeat ( 3 ) $display ( "FAIL" ) ;
    end

      $finish ;
end

endmodule
```

run -all
 Clkout frequencies 0 5.000000e+08 , 1 2.000000e+08
PASS
PASS
PASS

Exercise
In the case of division by 2 all the extra reusable logic becomes unnecessary. Create another module ehgu_clkdiv2 exclusively for division by 2 using only one flop with its D input driven by inverted Q output.

A generated clock constraint on the output of the clock divider flop can sometimes cause difficulty in dealing with clock tree synthesis, scan chain timing and functional timing optimization. This is because when the q output of this flop is declared as a generated clock, some tools may be confused by a clock signal driving the d pin which is a data pin, when the tool expects that clocks drive only clock pins. One trick to suppress this effect is to use one more flop that takes the same !Q as input but is for output only. That is it does not drive any feedback paths. Alternatively, you could also register the output of the q output with a flop driven by input clock. Add a parameter DECOUPLED_OUTFLOP that when set to 1 uses the extra flop. If not uses only one flop.

Odd Number Clock Divider With 50% Duty Cycle

A clock divider using flops typically creates an output waveform that stays high for an integer number of input clock cycles. That means if the division needed is an odd number, say 7, the output high period can by 3 or 4 cycles of the input clock. It can never be 3.5 cycles that is needed to get 50% (7/2) duty cycle. However, there is a trick that can be applied with care to get 3.5 cycle high time. The reason the 3.5 is not possible with a single clock edge logic like a counter is because the finest granularity of state change is one full cycle. What if the negative edge can be used to change the granularity to 0.5 cycle? This idea is what is used to get 50% duty for odd division. The duty50stage module combinationally combines the two divided clock. One of the clock is ON for 4 cycles, OFF for 3 cycles with another clock waveform that is 0.5 cycles OFF, 4 cycles ON, 2.5 cycles OFF. You can see that AND-ing the two waveforms picks out only the overlapping 0.5 to 4 cycles for the high period thereby creating a 3.5 ON, 3.5 OFF pattern. Note that this logic is unnecessary for division by even numbers and is eliminated using the generate-if DUTY50 parameter set to 1.

Note

The ANDing with a half cycle delayed version method propagates input duty cycle and it cannot correct input duty error. For example if the input is 55% duty then the output error will be 55%/division. For division by 7, it will be `50 + (55-50)/7.0 = 50.7%`

There could be timing caveats in this circuit. Use only with utmost care. Generally clocks are not ANDed. Eventhough NAND gate is generally not glitchy like XOR gate or MUX, but you still need care. Combinational gates can glitch and need careful circuit simulations to prove that the clock divider circuit operates properly in all PVT corners. I would synthesize this logic into gates and simulate the resulting layout in a circuit simulator – SPICE, Cadence Spectre or equivalent in all PVT corners, possibly even Monte Carlo simulations. If the synthesis tool is not instantiating appropriate clock logic gates, you may want to make this in the schematic capture method and use the SV code as just a functional model.

```systemverilog
module ehgu_clkdiv_duty50stage
 (
input logic clkin ,
input logic rstn ,
input logic en ,
input logic clkout_comb,
output logic clkout
 ) ;

  logic clkoutr , clkoutf ;
  always_ff @ ( posedge clkin , negedge rstn ) begin
    if ( !rstn ) begin
      clkoutr <= 0 ;
    end else begin
      clkoutr <= clkout_comb ;
    end
  end

  always_ff @ ( negedge clkin , negedge rstn ) begin
    if ( !rstn ) begin
      clkoutf <= 0 ;
    end else begin
      clkoutf <= clkoutr ;
    end
  end

  assign clkout = ~ ( clkoutr & clkoutf ) ;

endmodule

module ehgu_clkdiv
# (
parameter DIVISION = 2 ,
parameter bit DUTY50 = 0
 )
 (
input logic clkin ,
input logic rstn ,
input logic en ,
output logic clkout
 ) ;
import ehgu_basic_pkg :: increment_modulo_unsigned ;

localparam WIDTH = $clog2 ( DIVISION ) ;
```

```systemverilog
logic [ WIDTH-1 : 0 ] cnt , cnt_comb ;
logic nc ;
logic clkout_comb ;

always_comb begin
  increment_modulo_unsigned ( .inp ( cnt ) , .modulo ( DIVISION ) , .out ( cnt_comb ) , .wrapped (
nc ) ) ;
end

always_ff @ ( posedge clkin , negedge rstn ) begin
  if ( !rstn ) begin
    cnt <= 0 ;
  end else if ( en ) begin
    cnt <= cnt_comb ;
  end
end

assign clkout_comb = cnt >= ( DIVISION / 2 ) ;

generate
if ( DUTY50 == 1 && DIVISION%2==1 ) begin
  : d50pcnt

  ehgu_clkdiv_duty50stage d50stg
  (
   .clkin ,
   .rstn ,
   .en ( 1'b1 ) ,
   .clkout_comb ,
   .clkout ( clkout )
  ) ;

end else begin
  : regular
  always_ff @ ( posedge clkin , negedge rstn ) begin
    if ( !rstn ) begin
      clkout <= 0 ;
    end else begin
      clkout <= clkout_comb ;
    end
  end
end
endgenerate

endmodule
```

Testbench

```systemverilog
module tb ;
import thee_utils_pkg :: check_approx_equality ;
logic clkin ;
logic clkout0, clkout1, clkout2 ;
real fout0 , fout1 ;
real dutyr, duty0, duty1, duty2 ;
bit result0 , result1 ;
```

```verilog
logic rstn ;

thee_clk_gen_module clk_gen ( .clk ( clkin ) ) ;

thee_clk_freq_meter fmeter0 ( .clk ( clkout0 ) , .freq_in_hertz ( fout0 ) ) ;
thee_clk_freq_meter fmeter1 ( .clk ( clkout1 ) , .freq_in_hertz ( fout1 ) ) ;

thee_clk_duty_meter dmeterr ( .clk ( clkin ) , .duty ( dutyr ) ) ;
thee_clk_duty_meter dmeter0 ( .clk ( clkout0 ) , .duty ( duty0 ) ) ;
thee_clk_duty_meter dmeter1 ( .clk ( clkout1 ) , .duty ( duty1 ) ) ;
thee_clk_duty_meter dmeter2 ( .clk ( clkout2 ) , .duty ( duty2 ) ) ;

ehgu_clkdiv clkdiv0
 (
 .clkin ,
 .rstn ,
 .en ( 1'b1 ) ,
 .clkout ( clkout0 )
 ) ;

ehgu_clkdiv # ( .DIVISION ( 5 ) ) clkdiv1
 (
 .clkin ,
 .rstn ,
 .en ( 1'b1 ) ,
 .clkout ( clkout1 )
 ) ;

ehgu_clkdiv # ( .DIVISION ( 7 ), .DUTY50(1) ) clkdiv2
 (
 .clkin ,
 .rstn ,
 .en ( 1'b1 ) ,
 .clkout ( clkout2 )
 ) ;

initial begin
  repeat ( 2 ) @ ( posedge clkin ) ;
  rstn = 0 ;
  repeat ( 10 ) @ ( posedge clkin ) ;
  rstn = 1 ;
  repeat ( 10 ) @ ( posedge clkout0 ) ;
  repeat ( 10 ) @ ( posedge clkout1 ) ;
  repeat ( 10 ) @ ( posedge clkout2 ) ;
  $display ( " Clkout frequencies 0 %e , 1 %e" , fout0 , fout1 ) ;
  check_approx_equality ( .inp ( fout0 ) , .expected ( 0.5e9 ) , .result ( result0 ) ) ;
  check_approx_equality ( .inp ( fout1 ) , .expected ( 2e8 ) , .result ( result1 ) ) ;
  if ( result1 == 1 && result0 == 1 ) begin
  $display ( " Duty Cycle - inp clk %3.2f, out clk0 %3.2f, out clk1 %3.2f , out clk2 %3.2f ", dutyr,
duty0, duty1 , duty2 );
   repeat ( 3 ) $display ( "PASS" ) ;
  end else begin
   repeat ( 3 ) $display ( "FAIL" ) ;
  end
```

```
    $finish ;
end

endmodule
```

Simulation
You can see that for a division of 5 without the 50% feature the output duty is 3/5, 60%. When this feature is used the division by 7 maintains the 50% duty.

```
run –all
 Clkout frequencies 0 5.000000e+08 , 1 2.000000e+08
 Duty Cycle – inp clk 0.50, out clk0 0.50, out clk1 0.60 , out clk2 0.50
PASS
```

Fractional Clock Divider

In some cases you may want to divide the incoming clock by a decimal number like 3.5 rather than 3 or 4. All regular sequential circuits work on whole number multiples of clock cycles. To make them work on a time unit smaller than one clock cycle is not possible. Unless, if you note that it may be ok if the division is 3.5 on "average". So, you may think of division by 3.5 as 1/3.5 or 10/35. So, if your design can produce 10 rising edges for every 35 rising edges of the incoming clock, you can call that a fractional divider. It may be sufficient for some cases. Note that I do not know of a way to make a clean square wave division by 3.5 with simple digital logic alone. You can indeed create fractional division with more complex circuits like phase locked loop.

My implementation uses a fractional accumulator that keeps track of the fractional part. When the output clock cycle count differs from the expected by a full cycle then the whole number division amount is modified to compensate. This is somewhat akin to the leap year idea used in the Gregorian calendar. Lets call the frequency of earth's revolution around the sun arbitrarily as 1 unit. Then the frequency of a day is 365 times higher because earth spins that many more times about itself in the time it takes to go around the sun once. So, frequency of days is 365 units. If you get more accurate the frequency of days is 365.24 units. So, to get a proper year we need to divide it by 365.24. How do we do that when we can only split years into full days? We actually approximate the 0.24 as 0.25 which is same as ¼. Then by keeping days in a year as 365 for 3 years and then 366 for one year, we create on an average 365.25.

```
module ehgu_clkdiv_fractional
# (
 parameter INT_WIDTH = 3 ,
 parameter FRAC_WIDTH = 2
 )
 (
input logic clkin ,
input logic rstn ,
input logic en ,
input logic [ INT_WIDTH-1 : 0 ] int_div ,
input logic [ FRAC_WIDTH-1 : 0 ] frac_div ,
output logic clkout
 ) ;
```

```systemverilog
import ehgu_basic_pkg :: add_modulo_unsigned ;
import ehgu_basic_pkg :: increment_modulo_unsigned ;

logic [ INT_WIDTH-1 : 0 ] int_cnt , int_cnt_comb ;
logic [ INT_WIDTH : 0 ] division ;
logic [ FRAC_WIDTH-1 : 0 ] frac_cnt , frac_cnt_comb ;
logic end_of_cycle ;
logic phase_overflow ;

assign division = phase_overflow ? int_div + 1'b1 : int_div + 1'b0 ;

always_comb begin
  increment_modulo_unsigned ( .inp ( int_cnt ) , .modulo ( division ) , .out ( int_cnt_comb ) ,
.wrapped ( end_of_cycle ) ) ;
end

always_ff @ ( posedge clkin , negedge rstn ) begin
  if ( !rstn ) begin
    int_cnt <= 0 ;
  end else if ( en ) begin
    int_cnt <= int_cnt_comb ;
  end
end

always_ff @ ( posedge clkin , negedge rstn ) begin
  if ( !rstn ) begin
    clkout <= 0 ;
  end else begin
    clkout <= int_cnt >= ( division / 2 ) ;
  end
end

always_comb begin
  add_modulo_unsigned ( .inp0 ( frac_cnt ) , .inp1 ( frac_div ) , .modulo ( 2 ** FRAC_WIDTH ) ,
.sum ( frac_cnt_comb ) , .wrapped ( phase_overflow ) ) ;
end

always_ff @ ( posedge clkin , negedge rstn ) begin
  if ( !rstn ) begin
    frac_cnt <= 0 ;
  end else if ( en ) begin
    if ( end_of_cycle )
    frac_cnt <= frac_cnt_comb ;
  end
end

endmodule

module tb ;
import thee_utils_pkg :: check_approx_equality ;
logic clkin ;
logic clkout0 ;
logic clkout1 ;
real fout0 , exp_fout ;
```

```systemverilog
bit result0 , result1 ;
logic rstn ;

localparam INT_DIVISION = 7 ;
localparam FRAC_DIVISION = 14 ;
localparam INT_WIDTH = 3 ;
localparam FRAC_WIDTH = 4 ;

thee_clk_gen_module clk_gen ( .clk ( clkin ) ) ;

thee_clk_freq_meter # ( .MEAS_WINDOW ( 50 ) ) fmeter0 ( .clk ( clkout0 ) , .freq_in_hertz ( fout0 )
) ;

ehgu_clkdiv_fractional # ( .INT_WIDTH ( INT_WIDTH ) , .FRAC_WIDTH ( FRAC_WIDTH ) ) clkdiv0
 (
 .clkin ,
 .rstn ,
 .en ( 1'b1 ) ,
 .int_div ( INT_DIVISION ) ,
 .frac_div ( FRAC_DIVISION ) ,
 .clkout ( clkout0 )
 ) ;

initial begin
  repeat ( 2 ) @ ( posedge clkin ) ;
  rstn = 0 ;
  repeat ( 10 ) @ ( posedge clkin ) ;
  rstn = 1 ;
  repeat ( 500 ) @ ( posedge clkout0 ) ;
  exp_fout = 1.0e9 / ( INT_DIVISION + 1.0 * FRAC_DIVISION / ( 2 ** FRAC_WIDTH ) ) ;
  $display ( " Clkout frequencies 0 %e , expected %e" , fout0 , exp_fout ) ;
  check_approx_equality ( .inp ( fout0 ) , .expected ( exp_fout ) , .result ( result0 ) ) ;
  if ( result0 == 1 ) begin
   repeat ( 3 ) $display ( "PASS" ) ;
  end else begin
   repeat ( 3 ) $display ( "FAIL" ) ;
  end

  $finish ;
end

endmodule
```

```
run -all
 Clkout frequencies 0 1.269036e+08 , expected 1.269841e+08
PASS
```

More about fractional clock division is here -
https://zipcpu.com/blog/2017/06/02/generating-timing.html

Two Phase Non Overlapping Clock From Single Phase Clock

Some latch based circuits and some analog circuits need two phases of a clock signal with the added
requirement that at no time both the phases go high at the same time. This is achieved by a circuit that

looks like that of a latch. Subjectively speaking, the circuit kind of chops off some high time from the two phases that is set by a delay element. The SV code is only a model. You can commit it to silicon using real gates. In place of the # delay construct, you need to use an inverter chain.

```
module thee_clk2phase
# (
parameter realtime D = 200ps
 )
 (
input logic clkin ,
output logic clkp0 ,
output logic clkp1
 ) ;

logic clkp1_d, clkp0_d;

assign clkp0 = ~ ( clkin  | clkp1_d ) ;
assign clkp1 = ~ ( ~clkin | clkp0_d ) ;

assign #D clkp0_d = clkp0;
assign #D clkp1_d = clkp1;

endmodule
```

The testbench checks for overlap in the output clocks by AND-ing them. It also outputs the duty cycle and non overlap time of the clock phases produced.

```
module tb ;
import thee_utils_pkg :: check_approx_equality ;
logic clkin ;
logic clkp0, clkp1;
real fin, fout0 , fout1 ;
real dutyr, duty0, duty1 ;
bit result0 , result1 ;
bit overlap_found ;
realtime nonoverlap;

thee_clk_gen_module clk_gen ( .clk ( clkin ) ) ;

thee_clk_freq_meter fmeteri ( .clk ( clkin ) , .freq_in_hertz ( fin ) ) ;
thee_clk_freq_meter fmeter0 ( .clk ( clkp0 ) , .freq_in_hertz ( fout0 ) ) ;
thee_clk_freq_meter fmeter1 ( .clk ( clkp1 ) , .freq_in_hertz ( fout1 ) ) ;

thee_clk_duty_meter dmeterr ( .clk ( clkin ) , .duty ( dutyr ) ) ;
thee_clk_duty_meter dmeter0 ( .clk ( clkp0 ) , .duty ( duty0 ) ) ;
thee_clk_duty_meter dmeter1 ( .clk ( clkp1 ) , .duty ( duty1 ) ) ;

thee_clk2phase #(.D(100ps)) clk2phase
 (
 .clkin ,
 .clkp0 ,
 .clkp1
 ) ;
```

```verilog
logic overlap;
assign overlap = clkp0 && clkp1;
initial begin
 overlap_found = 0 ;
 repeat (2) @(posedge clkp0)
 forever @(posedge overlap) begin
  overlap_found = 1;
  $display("%t : Clock phase 0 and clk phase 1 overlapped", $realtime);
 end
end


initial begin
realtime f0r1, f1r0;

repeat (2) @(posedge clkp0);
@(negedge clkp0);
f0r1 = $realtime();
@(posedge clkp1);
f0r1 = $realtime() - f0r1;

@(negedge clkp1);
f1r0 = $realtime();
@(posedge clkp0);
f1r0 = $realtime() - f1r0;

nonoverlap = f0r1 < f1r0 ? f0r1 : f1r0 ;

$display("Clock phase 0 to phase 1 non overlap %f",f0r1);
$display("Clock phase 1 to phase 0 non overlap %f",f1r0);

end

initial begin
  repeat ( 2 ) @ ( posedge clkin ) ;
  repeat ( 10 ) @ ( posedge clkp0 ) ;
  repeat ( 10 ) @ ( posedge clkp1 ) ;
  $display ( " Clkout frequencies 0 %e , 1 %e" , fout0 , fout1 ) ;
  check_approx_equality ( .inp ( fout0 ) , .expected ( fin ) , .result ( result0 ) );
  check_approx_equality ( .inp ( fout1 ) , .expected ( fin ) , .result ( result1 ) );
  if ( result1 == 1 && result0 == 1 && overlap_found == 0 ) begin
  $display ( " Duty Cycle - inp clk %3.2f, out clk0 %3.2f, out clk1 %3.2f ", dutyr, duty0, duty1  );
   repeat ( 3 ) $display ( "PASS" ) ;
  end else begin
   repeat ( 3 ) $display ( "FAIL" ) ;
  end

  $finish ;
end

endmodule
```

Simulation

```
run -all
Clock phase 0 to phase 1 non overlap 0.100000
Clock phase 1 to phase 0 non overlap 0.100000
 Clkout frequencies 0 1.000000e+09 , 1 1.000000e+09
 Duty Cycle - inp clk 0.50, out clk0 0.40, out clk1 0.40
PASS
```

You can see that the non overlap is achieved by reducing the ON time of the output phases which results in a reduced duty cycle of 40% when the input duty cycle is 50%.

Clock Muxing

Multiplexing two clocks into one output is a tricky endeavor. As a first version, you could just use a regular mux.

```
assign clkout = sel ? clk0 : clk1 ;
```

But it is prone to creating glitches at the output. The glitches themselves are not really a problem because you could reset the system after making the change of clock. Another problem lurks in the simple mux. The mux could be distorting the duty cycle. If the input clock had a perfect 50% duty cycle, the output clock could be at 45% or 55% duty cycle. This could happen if the mux output delay is edge dependent. That is rise edge is delayed more (or less) compared to the fall edge. This can be avoided by using a special mux made for clock signals with equal rise and fall delays. This mux can be hard coded in RTL by instantiating the specific standard cell in the design or during synthesis the tool can be forced to use only clock type cells in the clock path. You could even remap cells during clock tree synthesis with clock type cells.

Some designs do not have the luxury of time to make a reset after a change in the clock selection. In that case, you could use a more sophisticated circuit that switches without glitches. Unlike the diagram in the link below, you could use an all negative edge synchronizer instead of a positive and a negative edge one.

https://www.eetimes.com/techniques-to-make-clock-switching-glitch-free/#

The logic in the SV code here implements the clock mux with the change for glitch free selection. It is achieved by making sure both clocks are turned off before switching the output mux. If you look carefully at the circuit you can see a clock gating logic for every clock, a synchronizer for before the clock gate enables and finally, just a mux like AND-OR logic.

```
module ehgu_clkmux
# (
parameter CLK0_SYNC_STAGES = 2 ,
parameter CLK1_SYNC_STAGES = 2
 )
 (
input logic clkin0 ,
input logic clkin1 ,
input logic sel ,
output logic clkout
 );
```

```systemverilog
logic selc0 , selc1 ;

logic sync_in0 , sync_in1 ;

assign sync_in0 = ~selc1 & ~sel ;

ehgu_synqzx # ( .MAX_DELAY ( 0 ) , .STAGES ( CLK0_SYNC_STAGES ) , .WIDTH ( 1 ) ) sync0
 (
.clk ( clkin0 ) ,
.rstn ( 1'b1 ) ,
.d_presync ( sync_in0 ) ,
.d_sync ( selc0 )
 ) ;

assign sync_in1 = ~selc0 & sel ;

ehgu_synqzx # ( .MAX_DELAY ( 0 ) , .STAGES ( CLK1_SYNC_STAGES ) , .WIDTH ( 1 ) ) sync1
 (
.clk ( clkin1 ) ,
.rstn ( 1'b1 ) ,
.d_presync ( sync_in1 ) ,
.d_sync ( selc1 )
 ) ;

assign clkout = ( selc0 & clkin0 ) | ( selc1 & clkin1 ) ;

endmodule
```

The testbench randomly switches between the input clocks and checks the output frequency for the expected frequencies. For this test, 2 clock sources, an output clock frequency meter and tasks that switch and check are defined.

```systemverilog
module tb ;

logic clkin0 , clkin1 ;
logic clksel ;
logic clkout ;
bit result ;
parameter int CLKFREQ [ 2 ] = '{ 10 , 100 } ;
real fout ;

thee_clk_gen_module # ( .FREQ ( CLKFREQ [ 0 ] ) ) clk_gen0 ( .clk ( clkin0 ) ) ;
thee_clk_gen_module # ( .FREQ ( CLKFREQ [ 1 ] ) ) clk_gen1 ( .clk ( clkin1 ) ) ;

thee_clk_freq_meter fmeter0 ( .clk ( clkout ) , .freq_in_hertz ( fout ) ) ;

ehgu_clkmux clkmux
 (
 .clkin0 ,
 .clkin1 ,
 .clkout ,
 .sel ( clksel )
 ) ;
```

```
initial begin
  result = 1 ;

  for ( int i = 0 ; i < 3 ; i ++ ) begin
    bit tmp ;
    clksel = $random ;
    switch_clk ( clksel ) ;
    check_clkfreq ( clksel , tmp ) ;
    result &= tmp ;
  end

  if ( result ) begin
    repeat ( 3 ) $display ( "PASS" ) ;
  end else begin
    repeat ( 3 ) $display ( "FAIL" ) ;
  end

  $finish ;
end

task switch_clk (
input logic sel
 ) ;
 clksel = sel ;
 repeat ( 2 + 1 ) @ ( posedge clkin0 ) ;
 repeat ( 2 + 1 ) @ ( posedge clkin1 ) ;

 $display ( "Selecting clock %b" , sel ) ;

endtask

task automatic check_clkfreq (
input logic sel ,
output bit cmp
 ) ;
 import thee_utils_pkg :: check_approx_equality ;
 $display ( "Clock freq result %1.3e" , fout ) ;
 check_approx_equality ( .inp ( fout ) , .expected ( CLKFREQ [ sel ] * 1e6 ) , .result ( cmp ) ) ;
endtask

endmodule

run -all
Selecting clock 0
Clock freq result 1.000e+07
Selecting clock 1
Clock freq result 1.000e+08
Selecting clock 1
Clock freq result 1.000e+08
PASS
```

There is still one problem in the glitch free clock mux circuit. It is prone to locking up the selection logic if any of the clock is not toggling. To support non toggling clocks, you could use a third always ON clock that acts as a time reference to detect if a clock has stopped. After making sure the clock has

truly stopped, it is now safe to switch the mux to the target clock. Such type of circuit is briefly described here.

http://rtlery.com/components/glitch-free-clock-multiplexermux

Exercise

Can you create a clock mux logic that unconditionally switches without glitches for any combination of input clock frequencies and in a fixed time? Suppose, input clocks range in frequencies from 0 to 100MHz, can you still guarantee glitch free switching in say 10us?

Answer: I grappled with this question for sometime. My conclusion is that as the frequency approaches 0, the time needed to wait for the negative edge approaches infinity. So, in theory, it is not possible to guarantee glitch free switching. But, given a lower limit in frequency, you could guarantee glitch free switching by waiting for more than one period of the lowest frequency.

Can you verify the clock mux in more detail? Randomize the input clock frequencies and add a check for unwanted glitches.

Can you implement a clock mux logic in RTL that uses a third reference clock to switch glitch free without lockup even for stopped clocks? Hint: use FSM, counters for timeout.

Time Delay FSM

Examples for finite state machines (FSM) are numerous. I know of only two subcategories in FSMs – Mealy FSM and Moore FSM. I did not find that categorization useful in modern chip design. But there are some recurring segments in the FSM state transitions. One such segment is the time delay part. The state machine waits for a fixed number of clock cycles before moving to the next state. A counter can be started at the entry into this state and then cleared when time expires. Lets study the code a bit more. The localparam constant definitions specify how many clock cycles to wait in each state. State transitions are triggered when the counter reaches the required number of cycles for that state defined by that localparam. The counter is cleared with a simple expression when the next state is not going be the current state, indirectly meaning any state transition. Contrast this style with the style of having to repeat the expression similar to the next state logic. The testbench is nothing more than a clock input. The check was done by studying the waveforms in Vivado waveform viewer. You may do it programatically by checking the number of clock cycles the FSM spends in each state and comparing the sequence of states taken.

```
module time_del_fsm
(
input clk ,
input rstn ,
output logic final_out
) ;

localparam STATE1_WAIT = 3 ;
localparam STATE2_WAIT = 5 ;
```

```systemverilog
localparam STATE3_WAIT = 4 ;

typedef enum logic [ 1 : 0 ] { IDLE = 0 , STATE1 = 1 , STATE2 = 2 , STATE3 = 3 } fsm_state_t ;

fsm_state_t state , next_state , prev_state ;

logic [ 2 : 0 ] cnt ;
logic sync_clr ;

always_ff @ ( posedge clk , negedge rstn ) begin
  if ( !rstn ) begin
    state <= IDLE ;
    prev_state <= IDLE ;
  end else begin
    state <= next_state ;
    prev_state <= state ;
  end
end

always_comb begin
  next_state = IDLE ;
  if ( state == IDLE ) begin
    next_state = STATE1 ;
  end else begin
    if ( state == STATE1 && cnt == STATE1_WAIT-1 ) begin
      next_state = STATE2 ;
    end else if ( state == STATE2 && cnt == STATE2_WAIT-1 ) begin
      next_state = STATE3 ;
    end else if ( state == STATE3 && cnt == STATE3_WAIT-1 ) begin
      next_state = STATE1 ;
    end else begin
      next_state = state ;
    end
  end
end

ehgu_cntr cntr (
.clk ,
.rstn ,
.sync_clr ,
.en ( 1'b1 ) ,
.cnt
 ) ;

assign final_out = ( state == STATE3 ) && ( cnt == ( STATE3_WAIT-1 ) ) ;
assign sync_clr = ( next_state != state ) ;

endmodule

module tb (

 ) ;

timeunit 1ns ;
timeprecision 1ps ;
```

```systemverilog
logic clk ;
logic rstn ;
logic final_out ;

time_del_fsm dut
 (
.clk ,
.rstn ,
.final_out
 ) ;

initial begin
 clk = 0 ;
 rstn = 0 ;
 #1 ;
 clk = 0 ;
 rstn = 1 ;
 #1 ;
 forever begin
   clk = ~clk ;
   #5 ;
 end
end

initial begin
  repeat ( 50 ) @ ( posedge clk ) ;
  $finish ;
end

endmodule
```

Exercise
Can you create a time delay FSM using a counter? Outputs are based on a window of counts. Suppose you want three outputs, first with count from 0-3, second from 4-7, third from 2-5.

Dosa is an delicious food in India. Suppose you want to make a controller for a Plain Dosa maker machine. It has the following states – spread the batter on the pan, wait for 2 minutes, flip the Dosa, wait for 1 min and remove the dosa form the pan. Can you make an FSM for this controller?

Serializing a Datapath – XOR with NOT, AND, OR

Serializing an operation means to stretch out the work in time. Serialization is useful to implement complex functions with simple elements. Serializing needs some way to keep track of which computational unit is to be put to work at what sequence. This is called scheduling. An FSM is used to schedule the correct operations in time. Computational blocks that work on streaming data are called datapath. For example, a video processing pipeline for compression fits the description of a datapath. An XOR function is taken for the purpose of showing this concept. Note that serialized XOR is a lame example because the XOR operator "^" would result in smaller and faster hardware than the serialized version. But XOR gate is small and complex enough to clearly demonstrate the concept. Serialization starts to make more sense for large operations, say, 128bit multiplication. Sometimes serialization may

be the only way to implement a computation because a full parallel implementation will take an unbelievable amount of area.

```systemverilog
module serial_xor (
input logic clk ,
input logic rstn ,
input logic a ,
input logic b ,
output logic xo ,
output logic ready
 ) ;

logic mem [ 2 ] , memnext [ 2 ] ;

typedef enum logic [ 2 : 0 ] { C0 = 0 , C1 = 1 , C2 = 2 , C3 = 3 , C4 = 4 } state_t ;

state_t state , next ;

always_comb
 next = state.next ( ) ;

always_comb begin
  memnext = mem ;
  ready = 0 ;
  xo = 0 ;
  case ( state )
  C0 : memnext [ 0 ] = ~b ;
  C1 : memnext [ 0 ] = mem [ 0 ] & a ;
  C2 : memnext [ 1 ] = ~a ;
  C3 : memnext [ 1 ] = mem [ 1 ] & b ;
  C4 : begin
   xo = mem [ 0 ] | mem [ 1 ] ;
   ready = 1 ;
  end
  default : begin
   memnext = mem ;
   ready = 0 ;
   xo = 0 ;
  end
 endcase
end

always_ff @ ( posedge clk , negedge rstn )
 if ( !rstn ) begin
 mem <= '{ default : 0 } ;
 state <= state.first ( ) ;
 end else begin
 mem <= memnext ;
 state <= next ;
 end

endmodule
```

The XOR operation is done in 5 cycles.
Cycle **0** : mem[0] = ~b;

C1: mem[0] = mem[0] & a; //now mem[0] contains a AND (NOT b)
C2: mem[1] = ~a;
C3: mem[1] = mem[1] & b; //now mem[1] contains (NOT a) AND b
C4: output = mem[0] OR mem[1] ; //now output implements the full
XOR function = (a AND (NOT b)) OR ((NOT a) AND b)

You can notice that to serialize operations we also need temporary storage. Usually this is implemented with a memory block. In this example that happens to be two flops. There is also a notion of validity of the output, referred here as ready. The output qualifier goes by many names, valid signal, data enable, output enable.

Testbench

```
module tb ;

logic a , b , xo , ready ;
logic clk , rstn ;
bit result ;

thee_clk_gen_module clk_gen ( .clk ( clk ) ) ;

serial_xor dut (
.clk ,
.rstn ,
.a ,
.b ,
.xo ,
.ready
 ) ;

initial begin
  import thee_utils_pkg :: toggle_rstn ;
  toggle_rstn ( .rstn ( rstn ) ) ;
  a = 0 ; b = 0 ;
  repeat ( 5 ) @ ( posedge clk ) ;
  result = 1 ;
  @ ( posedge ready ) ;
  @ (posedge clk);
  for ( int i = 0 ; i < 4 ; i ++ ) begin
    { a , b } = i ;
    $display("Inputs applied a=%b b=%b",a,b);
    while ( 1 ) begin
     @ ( posedge clk ) ;
     $display ( "State of scheduler %s , inputs %b , %b output xo %b , memory %b , %b" ,
dut.state.name ( ) , a , b , xo , dut.mem [ 0 ] , dut.mem [ 1 ] ) ;
     if ( ready ) begin
       $display ( "Result ready for inputs a %b b %b output a^b %b" , a , b , xo ) ;
       $display ("----------------------------------------------------");
      if ( xo !== a^b )
        result = 0 ;
      break ;
     end
   end
  end
```

```verilog
    if ( result )
      $display ( "All Vectors passed" ) ;
    else
      $display ( "Some Vectors failed" ) ;

    $finish ;
end

endmodule
```

Note the usage enum methods first() , last(), name() and hierarchical references.

Output
run -all
Inputs applied a=0 b=0
State of scheduler C0 , inputs 0 , 0 output xor 0 , memory 0 , 0
State of scheduler C1 , inputs 0 , 0 output xor 0 , memory 1 , 0
State of scheduler C2 , inputs 0 , 0 output xor 0 , memory 0 , 0
State of scheduler C3 , inputs 0 , 0 output xor 0 , memory 0 , 1
State of scheduler C4 , inputs 0 , 0 output xor 0 , memory 0 , 0
Result ready for inputs a 0 b 0 output a^b 0

Inputs applied a=0 b=1
State of scheduler C0 , inputs 0 , 1 output xor 0 , memory 0 , 0
State of scheduler C1 , inputs 0 , 1 output xor 0 , memory 0 , 0
State of scheduler C2 , inputs 0 , 1 output xor 0 , memory 0 , 0
State of scheduler C3 , inputs 0 , 1 output xor 0 , memory 0 , 1
State of scheduler C4 , inputs 0 , 1 output xor 1 , memory 0 , 1
Result ready for inputs a 0 b 1 output a^b 1

Inputs applied a=1 b=0
State of scheduler C0 , inputs 1 , 0 output xor 0 , memory 0 , 1
State of scheduler C1 , inputs 1 , 0 output xor 0 , memory 1 , 1
State of scheduler C2 , inputs 1 , 0 output xor 0 , memory 1 , 1
State of scheduler C3 , inputs 1 , 0 output xor 0 , memory 1 , 0
State of scheduler C4 , inputs 1 , 0 output xor 1 , memory 1 , 0
Result ready for inputs a 1 b 0 output a^b 1

Inputs applied a=1 b=1
State of scheduler C0 , inputs 1 , 1 output xor 0 , memory 1 , 0
State of scheduler C1 , inputs 1 , 1 output xor 0 , memory 0 , 0
State of scheduler C2 , inputs 1 , 1 output xor 0 , memory 0 , 0
State of scheduler C3 , inputs 1 , 1 output xor 0 , memory 0 , 0
State of scheduler C4 , inputs 1 , 1 output xor 0 , memory 0 , 0
Result ready for inputs a 1 b 1 output a^b 0

All Vectors passed

Exercise
Serialize a half adder using one XOR gate and one AND gate, use as much memory as you need.
Serialize a 4-bit ripple carry adder using only one full adder and as many memory elements.

Pipelining an Adder

In this example, we will look at the concept of pipelining using an adder as an example. The simple looking "+" operator in SV hides the complexity of making it work at high speeds. In general, even the best implementations of adder are limited by the number of bits that can be added in 1 clock cycle. An 8 bit adder may be trivial but a 128 bit adder may not complete in one cycle of a 1GHz clock. This situation is common in many combinational logic. Pipelining is used to speed up the number of operations that can be done in one clock cycle at the cost of increasing latency. Pipelining has similarity to many other systems like factory assembly line and people transporting small items hand to hand like in a human chain. The idea is to break up a big task into small sequential pieces that each can be done rapidly.

Let me setup the baseline or control or reference implementation to show the speed gain. The inputs and outputs are registered, so that, the synthesis tool can be made to focus on the internal path timing. In STA terminology, external paths are timing paths that involve signals crossing the boundary of the module and internal paths are the ones with signals entirely within the module. The identifier "nopi" denotes not pipelined.

No pipelined code

```
always @ ( posedge clk ) begin
  add_nopi_op0_reg <= add_nopi_op0 ;
  add_nopi_op1_reg <= add_nopi_op1 ;
  add_nopi_out_reg <= add_nopi_op0_reg + add_nopi_op1_reg ;
end
```

To measure speed, we will have to constrain the design with a desired clock frequency. I set the frequency to an unattainable value to push the synthesis tool to the limit.

Timing constraints in SDC format
create_clock -period 1.0 clk

Note the default unit for period is generally nanoseconds. So 1 ns period is 1 GHz frequency.

Upon reporting timing after synthesis completion, we can see that the slack without pipelining turned out to be -2.9ns. This means that if the clock period of the constraint was relaxed by the slack amount then the logic would meet timing. So, the maximum speed of adder for the device I selected for synthesis is -

Minimum period = clock period in constraint + slack = 1 + 2.9 = 3.9ns

Maximum frequency = 1 / minimum period = 256MHz

With pipelining
To increase the speed of the adder we cut the logic into smaller pieces. As a first attempt, we can cut the logic in half. So there are are now two adders. But the adder for the lower bits also produces one more output, the carry. This is added in the next cycle to the upper sum calculated in the previous cycle

to get the full output at the second cycle. Note the differences here. The 64 bit adder in unpipelined implementation got split into two 32b adders and another 32b + carry adder. The latency also increased from 1 clock cycle to 2 clock cycles because the carry bit addition can only happen after it has been computed in the previous cycle.

```verilog
always @ ( posedge clk ) begin
  add_pipd_op0_reg <= add_pipd_op0 ;
  add_pipd_op1_reg <= add_pipd_op1 ;
  { carry , add_pipd_out_reg_halfs [ 0 ] } <= add_pipd_op0_reg [ ( WIDTH / 2 ) - 1 : 0 ] +
add_pipd_op1_reg [ (WIDTH / 2)-1 : 0 ] ;
  add_pipd_out_reg_halfs [ 1 ] <= add_pipd_op0_reg [ WIDTH-1 : WIDTH / 2 ] + add_pipd_op1_reg
[ WIDTH-1 : WIDTH / 2 ] ;
  add_pipd_out_reg [ ( WIDTH / 2 ) - 1 : 0 ] <= add_pipd_out_reg_halfs [ 0 ] ;
  add_pipd_out_reg [ WIDTH-1 : WIDTH / 2 ] <= add_pipd_out_reg_halfs [ 1 ] + carry ;
end

assign add_pipd_out = add_pipd_out_reg ;
```

Slack for the pipelined implementation is -2.3ns implying a maximum frequency of 303MHz.
Note the other side effects of pipelining – less than double frequency gain and ugly RTL. After cutting the logic in half one may expect the maximum frequency to double to 512MHz, too bad, many things in real life go by the law of diminishing returns. In the pipelining case, the culprit is the fixed delay going through the flip flops in every stage. This fixed delay cannot be removed by adding more pipeline stages because every stage will have this. Lets put this into a tiny equation.

$$D = F + V$$
D – total delay
F – Fixed delay, delay of the flip flops
V – Variable delay, the delay of the adder

After pipelining
$$D2 = F + V/2$$

The RTL becomes ugly because the point of cutting the logic into pieces does not have a standard coding style that is reusable and target technology agnostic. At least, I am not aware of one.

Area is added too for the extra flip flops needed to hold the intermediate results.

Pipelining can be done manually using the following process -

1. Synthesize the RTL

2. Report timing

3. Check the slack on critical paths. If design passes timing exit pipelining

5. Check where you can add a pipeline stage in the critical path to break up the logic and make the RTL change.

6. Simulate to make sure RTL still passes verification

7. Go back to step 1

For some designs, I have gone through 10 to 20 time of this code – synthesize – time loop.

Full module with both piped and unpiped versions

```systemverilog
module adder_speedtest
# (
parameter WIDTH = 64
 ) (
input clk ,
input logic [ WIDTH-1 : 0 ] add_nopi_op0 , add_nopi_op1 ,
output logic [ WIDTH-1 : 0 ] add_nopi_out ,
input logic [ WIDTH-1 : 0 ] add_pipd_op0 , add_pipd_op1 ,
output logic [ WIDTH-1 : 0 ] add_pipd_out
 ) ;

logic [ WIDTH-1 : 0 ] add_nopi_op0_reg , add_nopi_op1_reg ;
logic [ WIDTH-1 : 0 ] add_nopi_out_reg ;
logic [ WIDTH-1 : 0 ] add_pipd_op0_reg , add_pipd_op1_reg ;
logic [ ( WIDTH / 2 ) - 1 : 0 ] add_pipd_out_reg_halfs [ 2 ] ;
logic [ WIDTH-1 : 0 ] add_pipd_out_reg ;
logic carry ;

always @ ( posedge clk ) begin
  add_nopi_op0_reg <= add_nopi_op0 ;
  add_nopi_op1_reg <= add_nopi_op1 ;
  add_nopi_out_reg <= add_nopi_op0_reg + add_nopi_op1_reg ;
end

assign add_nopi_out = add_nopi_out_reg ;

always @ ( posedge clk ) begin
  add_pipd_op0_reg <= add_pipd_op0 ;
  add_pipd_op1_reg <= add_pipd_op1 ;
  { carry , add_pipd_out_reg_halfs [ 0 ] } <= add_pipd_op0_reg [ ( WIDTH / 2 ) - 1 : 0 ] +
add_pipd_op1_reg [ (WIDTH / 2)-1 : 0 ] ;
  add_pipd_out_reg_halfs [ 1 ] <= add_pipd_op0_reg [ WIDTH-1 : WIDTH / 2 ] + add_pipd_op1_reg
[ WIDTH-1 : WIDTH / 2 ] ;
  add_pipd_out_reg [ ( WIDTH / 2 ) - 1 : 0 ] <= add_pipd_out_reg_halfs [ 0 ] ;
  add_pipd_out_reg [ WIDTH-1 : WIDTH / 2 ] <= add_pipd_out_reg_halfs [ 1 ] + carry ;
end

assign add_pipd_out = add_pipd_out_reg ;

endmodule

module tb ;

parameter WIDTH = 64 ;

logic clk ;
logic [ WIDTH-1 : 0 ] add_nopi_op0 , add_nopi_op1 ;
logic [ WIDTH-1 : 0 ] add_nopi_out ;
logic [ WIDTH-1 : 0 ] add_pipd_op0 , add_pipd_op1 ;
```

```systemverilog
logic [ WIDTH-1 : 0 ] add_pipd_out ;
logic [ WIDTH-1 : 0 ] expected ;
logic result ;

adder_speedtest # ( .WIDTH ( WIDTH ) ) add_duts
 (
.clk ,

.add_nopi_op0 ,
.add_nopi_op1 ,
.add_nopi_out ,

.add_pipd_op0 ,
.add_pipd_op1 ,
.add_pipd_out
 ) ;

thee_clk_gen_module clk_gen ( .clk ( clk ) ) ;

initial begin
  result = 1 ;
  for ( int i = 0 ; i < 5 ; i ++ ) begin
   add_nopi_op0 = $urandom ( ) * $urandom ;
   add_nopi_op1 = $urandom ( ) * $urandom ;
   add_pipd_op0 = add_nopi_op0 ;
   add_pipd_op1 = add_nopi_op1 ;
   expected = add_nopi_op0 + add_nopi_op1 ;
   repeat ( 4 ) @ ( posedge clk ) ;
   if ( add_nopi_out == expected && add_pipd_out == expected ) begin
     $display ( "Vector Passed" ) ;
   end else begin
     $display ( "Vector Failed" ) ;
     result = 0 ;
   end

   $display ( "Test Vector Unpipelined inputs %d + %d , output %d" , add_nopi_op0 , add_nopi_op1
, add_nopi_out ) ;
   $display ( "Test Vector Pipelined inputs %d + %d , output %d" , add_pipd_op0 , add_pipd_op1 ,
add_pipd_out ) ;

  end

  if ( result )
   $display ( "All Vectors passed" ) ;
  else
   $display ( "Some Vectors failed" ) ;

  $finish ;
end

endmodule

run -all
Vector Passed
```

```
Test Vector Unpipelined inputs   528797708167997617 + 16424120850955232788 , output
16952918559123230405
Test Vector Pipelined inputs   528797708167997617 + 16424120850955232788 , output
16952918559123230405
Vector Passed
Test Vector Unpipelined inputs   903788372399967600 +  1947374297566481306 , output
2851162669966448906
Test Vector Pipelined inputs   903788372399967600 +  1947374297566481306 , output
2851162669966448906
Vector Passed
Test Vector Unpipelined inputs 18081074197019791920 +  1419786126177169614 , output
1054116249487409918
Test Vector Pipelined inputs 18081074197019791920 +  1419786126177169614 , output
1054116249487409918
Vector Passed
Test Vector Unpipelined inputs 16139784752808046654 +  3793424318984004480 , output
1486464998082499518
Test Vector Pipelined inputs 16139784752808046654 +  3793424318984004480 , output
1486464998082499518
Vector Passed
Test Vector Unpipelined inputs    70241738306603386 +  1120519094471171271 , output
1190760832777774657
Test Vector Pipelined inputs    70241738306603386 +  1120519094471171271 , output
1190760832777774657
All Vectors passed
```

Pipelining a Counter

Pipelining a combinational logic like adder is easy enough. What about pipelining a sequential logic like a counter. At first glance I thought whats the big fuss? A counter is just a bunch of flip flops with an adder in the feedback path. So, split the adder as usual and get higher speed. Lets run an experiment to see what would happen when we do that. Take a 4 bit counter. It counts from 0 through 15 with count changing every clock cycle. Say, this operates up to a maximum speed of 1GHz. Let us add one pipeline stage to speed up the counter.

```systemverilog
module counter_speedtest
#(
parameter WIDTH=64
)(
input logic clk,
input logic rstn,
output logic [WIDTH-1:0] cnt_nopi_out,
output logic [WIDTH-1:0] cnt_pipd_out
);

logic [WIDTH/2-1:0] cnt_pipd_out_halfs[2];
logic carry ;

always @(posedge clk, negedge rstn) begin
  if (!rstn)
    cnt_nopi_out <= 0 ;
  else
    cnt_nopi_out <= cnt_nopi_out + 1 ;
end
```

```verilog
always @(posedge clk) begin
   if (!rstn) begin
      {carry, cnt_pipd_out_halfs[0]} <= 0 ;
      cnt_pipd_out_halfs[1] <= 0 ;
      cnt_pipd_out[WIDTH/2-1:0] <= 0 ;
      cnt_pipd_out[WIDTH-1:WIDTH/2] <= 0 ;
   end else begin
      {carry, cnt_pipd_out_halfs[0]} <= cnt_pipd_out[WIDTH/2-1:0] + 1 ;
      cnt_pipd_out_halfs[1] <= cnt_pipd_out[WIDTH-1 : WIDTH/2] ;
      cnt_pipd_out[WIDTH/2-1:0] <= cnt_pipd_out_halfs[0];
      cnt_pipd_out[WIDTH-1:WIDTH/2] <= cnt_pipd_out_halfs[1]+carry;
   end
end

endmodule
```

```
run -all
Vector Failed
Test Vector Unpipelined expected count      1 , output            1
Test Vector   Pipelined expected count      1 , output            0
Vector Failed
Test Vector Unpipelined expected count      2 , output            2
Test Vector   Pipelined expected count      2 , output            1
Vector Failed
Test Vector Unpipelined expected count      3 , output            3
Test Vector   Pipelined expected count      3 , output            1
Vector Failed
Test Vector Unpipelined expected count      4 , output            4
Test Vector   Pipelined expected count      4 , output            2
Vector Failed
Test Vector Unpipelined expected count      5 , output            5
Test Vector   Pipelined expected count      5 , output            2
Some Vectors failed
```

What a surprise! Now the counter works even slower. It counts every number twice. Why? The cycle delay introduced into the feedback path of the counter logic implies that the counter state can only change once every two cycles. Obviously, pipelining the feedback logic does not work.

Now what?

Actually for a special case like a counter, it is possible to massage the feedback logic including a pipeline stage into working at higher frequencies. This may not be possible for all sequential circuits. The trick is to cancel the extra cycle delay by counting differently. In the unpipelined case, the carry flows out of the lower half of the bits when the bits are all ones. If you want to cut the carry logic in half you would be penalized with a one cycle delay. Calculating carry as a one cycle earlier version will compensate the lag. The statement for future_carry assignment does exactly this.

```verilog
module counter_pipelined
# (
parameter WIDTH = 64
```

```systemverilog
) (
input logic clk ,
input logic rstn ,
output logic [ WIDTH-1 : 0 ] cnt_pipd_out
 ) ;

logic future_carry ;

always @ ( posedge clk ) begin
  if ( !rstn ) begin
    future_carry <= 0 ;
    cnt_pipd_out [ ( WIDTH / 2 ) - 1 : 0 ] <= 0 ;
    cnt_pipd_out [ WIDTH-1 : WIDTH / 2 ] <= 0 ;
  end else begin
    future_carry <= ( cnt_pipd_out [ ( WIDTH / 2 ) - 1 : 0 ] == ( { ( WIDTH / 2 ) { 1'b1 } } - 1'b1 )
) ;
    $display ( "future carry %d cnt %b" , future_carry , cnt_pipd_out ) ;
    cnt_pipd_out [ ( WIDTH / 2 ) - 1 : 0 ] <= cnt_pipd_out [ ( WIDTH / 2 ) - 1 : 0 ] + 1 ;
    cnt_pipd_out [ WIDTH-1 : WIDTH / 2 ] <= cnt_pipd_out [ WIDTH-1 : WIDTH / 2 ] + future_carry ;
  end
end

endmodule

module tb ;

parameter WIDTH=4;

logic clk,rstn;
logic [WIDTH-1:0] cnt_nopi;
logic [WIDTH-1:0] cnt_pipd_buggy;
logic [WIDTH-1:0] cnt_pipd;
logic result ;
int cnt;

counter_speedtest #(.WIDTH(WIDTH)) cnt_duts
(
.clk ,
.rstn ,
.cnt_nopi_out (cnt_nopi) ,
.cnt_pipd_out (cnt_pipd_buggy)
);

counter_pipelined #(.WIDTH(WIDTH)) cnt_dutp
(
.clk ,
.rstn ,
.cnt_pipd_out (cnt_pipd)
);

thee_clk_gen_module clk_gen (.clk(clk));

initial begin
import thee_utils_pkg::toggle_rstn;
repeat (4) @(posedge clk);
```

```
toggle_rstn(.rstn(rstn),.rst_low(9.4ns));

result = 1;
for ( int i = 0 ; i < 2**WIDTH ; i++ ) begin
repeat (1) @(posedge clk);
cnt++;
cnt %= (2**WIDTH);
if ( cnt_nopi == cnt && cnt_pipd == cnt ) begin
  $display ("Vector Passed");
end else begin
  $display ("Vector Failed");
  result = 0 ;
end

$display ("Test Vector Unpipelined expected count %d , output %d" , cnt, cnt_nopi );
$display ("Test Vector   Pipelined expected count %d , output %d" , cnt, cnt_pipd );
//$display ("Test Vector   Pipelined expected count %d , output %d" , cnt, cnt_pipd_buggy );

end

if ( result )
  $display ("All Vectors passed");
else
  $display ("Some Vectors failed");

$finish;
end

endmodule

run -all
future carry x cnt xxxx
future carry x cnt xxxx
future carry x cnt xxxx
future carry x cnt xxxx
future carry 0 cnt 0000
Vector Passed
Test Vector Unpipelined expected count        1 , output  1
Test Vector   Pipelined expected count        1 , output  1
future carry 0 cnt 0001
Vector Passed
Test Vector Unpipelined expected count        2 , output  2
Test Vector   Pipelined expected count        2 , output  2
future carry 0 cnt 0010
Vector Passed
Test Vector Unpipelined expected count        3 , output  3
Test Vector   Pipelined expected count        3 , output  3
future carry 1 cnt 0011
Vector Passed
Test Vector Unpipelined expected count        4 , output  4
Test Vector   Pipelined expected count        4 , output  4
future carry 0 cnt 0100
Vector Passed
Test Vector Unpipelined expected count        5 , output  5
Test Vector   Pipelined expected count        5 , output  5
```

future carry 0 cnt 0101
Vector Passed
Test Vector Unpipelined expected count   6 , output 6
Test Vector   Pipelined expected count   6 , output 6
future carry 0 cnt 0110
Vector Passed
Test Vector Unpipelined expected count   7 , output 7
Test Vector   Pipelined expected count   7 , output 7
future carry 1 cnt 0111
Vector Passed
Test Vector Unpipelined expected count   8 , output 8
Test Vector   Pipelined expected count   8 , output 8
future carry 0 cnt 1000
Vector Passed
Test Vector Unpipelined expected count   9 , output 9
Test Vector   Pipelined expected count   9 , output 9
future carry 0 cnt 1001
Vector Passed
Test Vector Unpipelined expected count   10 , output 10
Test Vector   Pipelined expected count   10 , output 10
future carry 0 cnt 1010
Vector Passed
Test Vector Unpipelined expected count   11 , output 11
Test Vector   Pipelined expected count   11 , output 11
future carry 1 cnt 1011
Vector Passed
Test Vector Unpipelined expected count   12 , output 12
Test Vector   Pipelined expected count   12 , output 12
future carry 0 cnt 1100
Vector Passed
Test Vector Unpipelined expected count   13 , output 13
Test Vector   Pipelined expected count   13 , output 13
future carry 0 cnt 1101
Vector Passed
Test Vector Unpipelined expected count   14 , output 14
Test Vector   Pipelined expected count   14 , output 14
future carry 0 cnt 1110
Vector Passed
Test Vector Unpipelined expected count   15 , output 15
Test Vector   Pipelined expected count   15 , output 15
future carry 1 cnt 1111
Vector Passed
Test Vector Unpipelined expected count   0 , output 0
Test Vector   Pipelined expected count   0 , output 0
All Vectors passed

Actual timing speed up

After synthesis, open the synthesized design and report timing in the TCL console.
open_run synth_1 -name
report_timing -to cnt_pipd_out*/D

The slack reported is -2.2ns for the pipelined case. Implying a max frequency of 1 / (1 + 2.2) , 312MHz.

Doing the same for unpipelined case gives a slack of -3.1 for a maximum frequency of 244MHz

There is a complication in verifying this counter. If the counter is large, say, 128b or so, the carry bit calculation can go untested. So, it may be necessary to start the simulation from an intermediate count value to cover the toggling of carry bit. Three coverage bins may be defined – low, toggle carry and high for the count register.

The idea of splitting the carry logic can be extended to smaller and smaller pieces of the counter bits with diminishing speed gains. For example, if the counter is split into three pieces, the first carry is straight forward, the second carry is set as a function of a new additional one delay earlier first carry and the middle counter bits. You may even find a parameterized way to pipeline counters this way. Another way is to let the carries be generated as usual at the turn of all ones to all zeros and then then pipeline the output carry. To match the delay of the different pieces of the counter, the counter output can be delayed. This approach is explained in this webpage -

https://zipcpu.com/blog/2017/06/02/generating-timing.html
Section: Divided Counter

I don't know of ways to pipeline other sequential circuits like this. Though there may be ways to implement pipelining in ways specific to every sequential circuit. For example, an FSM can be speeded up by using one-hot encoding of the state bits rather than binary encoding. This is not exactly pipelining but rather a case of using more flops to simplify combinational logic.

Pipelining Reset

The reset signal can present special problems for timing. Though, different blocks can be designed to come out of reset one after the other, it is easier to release reset at once for all blocks. In a large chip, it can become challenging to deliver the reset signal to all flops within one clock cycle. You may wonder, why not just pipeline? The problem with pipelining a reset signal is that it is not a regular data like signal. The functionality is generally semi-synchronous, that is, the assertion is preferred to be asynchronous and the deassertion is synchronous. The asynchronous assertion messes up the traditional pipelining method because pipelining is not meant to transfer asynchronous signal. You could still create this behavior by AND-ing the input of the flop with the output of the flop. But, that may present its own timing problems I am not yet aware of. The easier method is to use another reset synchronizer like stage as a pipeline register. In this synchronizer type logic, the reset assertion passes via the asynchronous reset pin of the stage and so propagates asynchronously to the output of the stage. The de-assertion goes through the d-q pins of the stage and adds one clock cycle delay. This way the deassertion is pipelined. There is still one logistics problem remaining. How do you code this in a scalable way? A large chip could have a million or more flops in the same reset domain. Pipelining the reset signal for the million flops will be a nightmare. Even if you use one synchronizer stage per 100 flops and make a tree, there will still be many sync stages to insert. So, we need a coding style that scales to as many flops as needed.
Here, I present a pipelined reset logic that takes an asynchronous reset input and synchronizes it to the target clock domain and outputs a large number of reset outputs all deasserting on the same clock cycle. Note that you should not use reset synchronizers in parallel to synchronize the asynchronous input reset because each synchronizer outputs the reset deassertion edge with one clock cycle uncertainty. So,

some synchronizers could output with 1 cycle delay while some others could output with 2 cycles delay. This may corrupt the FSMs in the design. After the synchronized reset signal is generated, using reset synchronizer in between the sync reset source and destination flops improves the removal timing, but, increasing the stage count of one synchronizer does not improve timing. The critical timing path for this case is shown below.

Reset source rstn pin – (rst sync: rstn goes to all flop rstn pin, first flops d input connected to logic 1, subsequently q is connected to d-q-d-q… - rstn output)

This is because the reset source to the last stage of the downstream pipelining reset synchronizer is the bottleneck because it still needs to meet single cycle timing when we expected increased stage count to break this delay. We need a different approach!

The solution is to use the synchronizer type connection and keep repeating that logic in a chain, like this -

rstn source – (flop: async rstn pin – q output) – (flop: async rstn pin – q output) – …
all d inputs are connected to logic 1. Note that shortness of the intermediate rstn signals, it drives only one flop in the chain. Because, adding stages to each reset synchronizer is not helping we can reduce the stage count to just one. Here the reset sync like logic is working only on the synchronized reset signal, so, metastability is not a problem for the pipeline stages and don't need two stages per synchronizer.

SV coding does not allow an array of reset to be specified in an always block, like how the non blocking assignment statement, <=, automatically works for single bit and array signals. Reset does not work like that. For example,

```
logic [3:0] rstn, d, q;
always_ff @(posedge clk, negedge rstn)
  if (!rstn)
    q <= 0;
  else
    q <= d;
```

q[3:0] is correctly assigned d[3:0] in order, but, q[3:0] is not each controlled by rstn[3:0]. So, you need to split that into 4 separate always blocks.
```
always_ff @(posedge clk, negedge rstn[0])
  if (!rstn[0])
    q[0] <= 0;
  else
    q[0] <= d[0];

always_ff @(posedge clk, negedge rstn[1])
  if (!rstn[1])
    q[1] <= 0;
  else
    q[1] <= d[1];

always_ff @(posedge clk, negedge rstn[2])
  if (!rstn[2])
```

```
      q[2] <= 0;
    else
      q[2] <= d[2];

  always_ff @(posedge clk, negedge rstn[3])
    if (!rstn[3])
      q[3] <= 0;
    else
      q[3] <= d[3];
```

This structure is easy to code with the generate construct of SV.

To drive a large number of flop reset pins, we need a tree for the reset signal. You could create a tree type implementation with another generate loop for the tree levels where at each level the signal branches into N more trees. In the example below, I have used only one level of branching after the first reset synchronizer. The stages of the first synchronizer, the branching and pipelining of the tree in every branch can be controlled with parameters.

```
module ehgu_rst_tree_piped
# (
parameter SYNC_STAGES=2,
parameter PIPE_STAGES=3,
parameter LEAFS=4
)
(
  input logic clk,
  input logic rstn_async_in,
  output logic [LEAFS-1:0] rstn_out
);

logic rstn_synced;

logic [PIPE_STAGES-1+1:0] rstn_array [LEAFS];

ehgu_rst_sync #(.STAGES(SYNC_STAGES)) rst_sync_i
(
.clk,
.rstn_in(rstn_async_in),
.rstn_out(rstn_synced)
);

always_comb
  for ( int branch = 0 ; branch < LEAFS ; branch++ )
  begin
    rstn_array[branch][0] = rstn_synced;
    rstn_out[branch] = rstn_array[branch][PIPE_STAGES];
  end

generate
  for ( genvar branch = 0 ; branch < LEAFS ; branch++ )
  begin : genbranch
    for ( genvar stage = 0 ; stage < PIPE_STAGES ; stage++ )
    begin : genstage
      always_ff @(posedge clk, negedge rstn_array[branch][stage])
        if ( !rstn_array[branch][stage] )
```

```verilog
                rstn_array[branch][stage+1] <= 0;
            else
                rstn_array[branch][stage+1] <= 1;
        end
    end
endgenerate

endmodule

module tb ;

parameter SYNC_STAGES=2;
parameter PIPE_STAGES=3;
parameter LEAFS=4;

logic clk,rstn;
logic result ;
logic [LEAFS-1:0] rstn_out;

thee_clk_gen_module clk_gen (.clk(clk));

initial begin
import thee_utils_pkg::toggle_rstn;
repeat (1) @(posedge clk);
result = 1;
for(int i = 0 ; i<LEAFS;i++) begin
   $display("Reset outputs of branch %d is %b", i, rstn_out[i]);
   if ( rstn_out[i]!==1'bx )
      result = 0;
end
toggle_rstn(.rstn(rstn),.rst_low(9.4ns));

repeat (PIPE_STAGES + SYNC_STAGES +1) @(posedge clk);

for(int i = 0 ; i<LEAFS;i++) begin
   $display("Reset outputs of branch %d is %b", i, rstn_out[i]);
   if ( rstn_out[i]!==1 )
      result = 0;
end

if ( result )
   $display ("All Vectors passed");
else
   $display ("Some Vectors failed");

$finish;
end

ehgu_rst_tree_piped # ( .SYNC_STAGES(SYNC_STAGES) , .PIPE_STAGES(PIPE_STAGES) ,
.LEAFS(LEAFS) ) rst_tree
(
.clk ,
.rstn_async_in (rstn),
.rstn_out
);

endmodule
```

run -all

Reset outputs of branch	0 is x
Reset outputs of branch	1 is x
Reset outputs of branch	2 is x
Reset outputs of branch	3 is x
Reset outputs of branch	0 is 1
Reset outputs of branch	1 is 1
Reset outputs of branch	2 is 1
Reset outputs of branch	3 is 1

All Vectors passed

Exercise

Only one level of branching is not adequate to handle a large number of end points. Can you code a scalable reset tree that has a constant fanout ratio or branching factor to drive say 10000 flops?

Hint: Assume a branching factor of 50, first level has 50 sync stages, each 50 drive another 50 in next level and so on. The last level may not be a full multiple of the 50 power level value, so you may need to drop the extra bits and connect only the relevant outputs to the primary outputs.

Retiming – Adder

Imagine a car assembly line with only 3 stages. The first stage joins the car under carriage with the body, the second stage is for installing the windshields and the engine and the last stage is for installing the wheels. Suppose carriage to body joining takes 15 minutes, installing windshield takes 15 minutes, installing engine takes 15 minutes and installing the wheels take 5 minutes. The time for each stage is 15, 30 (15+15), 5. The maximum number of cars that can be made per hour is 2. This is because in one hour the second stage can do only 2 (60min/30min) cars. Notice that the first stage can process 4 cars (60/15) per hour and the last stage can process 12 cars. How the car assembly unfolds is that a new car carriage and body arrive at the first stage at time 0 min. It completes its work at 15 min and passes to second stage. The first stage accepts parts for another car at time 15 and completes work on this second car by 30 min. In the mean time, the second stage is still working on the first car and completes only by minute 45. From minute 30 to minute 45, the first stage simply holds the second car without delivering to the second stage even though the first stage completed its part of the assembly. This is because the second stage is not ready to accept a new car until minute 45 (start time of minute 15 + processing time of 30min). How can this situation be improved?

In a pipelined operation, the slowest stage sets the production rate of the full assembly line. So the solution is to make all stages take equal time. Suppose the engine installation step is moved to the last stage, the time taken by each stage now becomes 15,15,20 for an overall production rate of 3 cars per hour (60 / max(15,15,20)). This is way faster than the 2 cars per hour.

In chipping, boolean algebra calculations are performed in each stage of a pipeline. The overall speed is highest if the pipeline stage delays are balanced. This operation is called retiming. I would have named it pipeline tweaking or pipeline balancing. Note however that technically retiming actually means

moving the combinational logic operations across the pipeline stages. Balancing is not really included in the retiming concept, though, balancing is the only reason retiming is done.

Retiming can be automatically done by synthesis tools. But, it may not be preferred because it modifies the flip flop fanin cone and makes the gate netlist hard to compare with the RTL. When the synthesis tool is allowed to retime, default LEC script may show mismatched flops because the fanin cone of the flops changed. To get the LEC to pass, you need to turn on appropriate settings in the LEC tool.

Retiming can be done manually using the following process -

1. Synthesize the RTL

2. Report timing

3. Check the slack on critical paths. If design passes timing exit retiming.

4. Check the slack on paths before and after the critical path

5. If RTL can be modified to move some logic out of the critical path to before/after the critical path, make the RTL change.

6. Simulate to make sure RTL still passes verification

7. Go back to step 1

```verilog
module adder_pipd
# (
parameter WIDTH = 32
 ) (
input clk ,
input logic [ WIDTH-1 : 0 ] add_pipd_op0 , add_pipd_op1 ,
output logic [ WIDTH-1 : 0 ] add_pipd_out
 ) ;

logic [ WIDTH-1 : 0 ] add_pipd_op0_reg , add_pipd_op1_reg ;
logic [ 16-1 : 0 ] add_pipd_out_reg_low ;
logic [ WIDTH-16-1 : 0 ] add_pipd_out_reg_high ;
logic [ WIDTH-1 : 0 ] add_pipd_out_reg ;

logic carry ;

always @ ( posedge clk ) begin
   add_pipd_op0_reg <= add_pipd_op0 ;
   add_pipd_op1_reg <= add_pipd_op1 ;
   {carry , add_pipd_out_reg_low} <= add_pipd_op0_reg [ 16-1 : 0 ] +
add_pipd_op1_reg [ 16-1 : 0 ] ;
   add_pipd_out_reg_high <= add_pipd_op0_reg [ WIDTH-1 : 16 ] + add_pipd_op1_reg
[ WIDTH-1 : 16 ] ;
   add_pipd_out_reg [ 16-1 : 0 ] <= add_pipd_out_reg_low ;
   add_pipd_out_reg [ WIDTH-1 : 16 ] <= add_pipd_out_reg_high + carry ;
end

assign add_pipd_out = add_pipd_out_reg ;

endmodule
```

```verilog
module adder_rtmd
# (
parameter WIDTH = 32
 ) (
input clk ,
input logic [ WIDTH-1 : 0 ] add_rtmd_op0 , add_rtmd_op1 ,
output logic [ WIDTH-1 : 0 ] add_rtmd_out
 ) ;

logic [ WIDTH-1 : 0 ] add_rtmd_op0_reg , add_rtmd_op1_reg ;
logic [ WIDTH / 2-1 : 0 ] add_rtmd_out_reg_halfs [ 2 ] ;
logic [ WIDTH-1 : 0 ] add_rtmd_out_reg ;

logic carry ;

always @ ( posedge clk ) begin
   add_rtmd_op0_reg <= add_rtmd_op0 ;
   add_rtmd_op1_reg <= add_rtmd_op1 ;
   {carry , add_rtmd_out_reg_halfs [ 0 ] } <= add_rtmd_op0_reg [ WIDTH / 2-1 : 0 ]
+ add_rtmd_op1_reg [ WIDTH / 2-1 : 0 ] ;
   add_rtmd_out_reg_halfs [ 1 ] <= add_rtmd_op0_reg [ WIDTH-1 : WIDTH / 2 ] +
add_rtmd_op1_reg [ WIDTH-1 : WIDTH / 2 ] ;
   add_rtmd_out_reg [ WIDTH / 2-1 : 0 ] <= add_rtmd_out_reg_halfs [ 0 ] ;
   add_rtmd_out_reg [ WIDTH-1 : WIDTH / 2 ] <= add_rtmd_out_reg_halfs [ 1 ] + carry
;
end

assign add_rtmd_out = add_rtmd_out_reg ;

endmodule

module tb ;

parameter WIDTH = 64 ;

logic clk ;
logic [ WIDTH-1 : 0 ] add_rtmd_op0 , add_rtmd_op1 ;
logic [ WIDTH-1 : 0 ] add_rtmd_out ;
logic [ WIDTH-1 : 0 ] add_pipd_op0 , add_pipd_op1 ;
logic [ WIDTH-1 : 0 ] add_pipd_out ;
logic [ WIDTH-1 : 0 ] expected ;
logic result ;

adder_rtmd # ( .WIDTH ( WIDTH ) ) add_rtmd
 (
.clk ,
.add_rtmd_op0 ,
.add_rtmd_op1 ,
.add_rtmd_out
 ) ;

adder_pipd # ( .WIDTH ( WIDTH ) ) add_pipd
 (
.clk ,
```

```verilog
    .add_pipd_op0 ,
    .add_pipd_op1 ,
    .add_pipd_out
     ) ;

    thee_clk_gen_module clk_gen ( .clk ( clk ) ) ;

    initial begin
      result = 1 ;
      for ( int i = 0 ; i < 5 ; i ++ ) begin
        add_rtmd_op0 = $urandom ( ) * $urandom ;
        add_rtmd_op1 = $urandom ( ) * $urandom ;
        add_pipd_op0 = add_rtmd_op0 ;
        add_pipd_op1 = add_rtmd_op1 ;
        expected = add_rtmd_op0 + add_rtmd_op1 ;
        repeat ( 4 ) @ ( posedge clk ) ;
        if ( add_rtmd_out === expected && add_pipd_out === expected ) begin
            $display ( "Vector Passed" ) ;
        end else begin
            $display ( "Vector Failed" ) ;
            result = 0 ;
        end

        $display ( "Test inputs %d + %d , expected output %d" ,
         add_pipd_op0 , add_pipd_op1 , expected ) ;
        $display ( "Test output non retimed %d" , add_pipd_out ) ;
        $display ( "Test output retimed %d" , add_rtmd_out ) ;

      end

      if ( result )
       $display ( "All Vectors passed" ) ;
      else
       $display ( "Some Vectors failed" ) ;

      $finish ;
    end

    endmodule

    run -all
    Vector Passed
    Test inputs   528797708167997617 + 16424120850955232788 , expected output
    16952918559123230405
    Test output non retimed 16952918559123230405
    Test output retimed 16952918559123230405
    Vector Passed
    Test inputs    70241738306603386 +  1120519094471171271 , expected output
    1190760832777774657
    Test output non retimed  1190760832777774657
    Test output retimed  1190760832777774657
    All Vectors passed
```

Synthesis script for Xilinx Vivado

```
foreach iter {pipd rtmd} {
```

```
  set dirname synth_${iter}.log
  file mkdir $dirname
  cd $dirname
  catch {
    close_project
  }
  create_project -force prj_${iter}
  add_files ../adder_${iter}.sv
  add_files ../adder_speedtest.sdc

  set_property top adder_${iter} [current_fileset]
  set_property generic WIDTH=64 [current_fileset]

  synth_design
  place_design
  route_design

  report_utilization > utilization.txt
  report_timing_summary > timing_summary.txt
  report_timing > timing.txt


#   close_project
  cd ..
}
```

Comparison of critical path timing with and without retiming

```
| Command    : report_timing          | Command    : report_timing
| Design     : adder_pipd       |  | Design     : adder_rtmd
| Device     : 7k70t-fbv676           | Device     : 7k70t-fbv676
| Speed File : -1  PRODUCTION 1.12 2017-02-17        | Speed File : -1  PRODUCTION 1.12 2017-02-17
-----------------------------------------------------------------------------------------------------------------------

Timing Report                   Timing Report

Slack (VIOLATED) :      -0.876ns  (required time - arrival |  Slack (VIOLATED) :      -0.633ns  (required time - arrival
  Source:            add_pipd_out_reg_high_reg[0]/C    |     Source:            add_rtmd_op0_reg_reg[1]/C
                     (rising edge-triggered cell FDRE                        (rising edge-triggered cell FDRE
  Destination:       add_pipd_out_reg_reg[61]/D    |  Destination:        carry_reg/D
                     (rising edge-triggered cell FDRE                        (rising edge-triggered cell FDRE
  Path Group:        clk              Path Group:        clk
  Path Type:         Setup (Max at Slow Process Corner)    Path Type:         Setup (Max at Slow Process Corner)
  Requirement:       1.000ns  (clk rise@1.000ns - clk ri    Requirement:       1.000ns  (clk rise@1.000ns - clk ri
  Data Path Delay:     1.871ns  (logic 1.386ns (74.086%) |  Data Path Delay:     1.616ns  (logic 1.311ns (81.102%)
  Logic Levels:      12  (CARRY4=12)      |  Logic Levels:      10  (CARRY4=9 LUT2=1)
```

Parallel processing – Adder

Parallel processing uses more instances of the same logic to get higher performance. Each instance operates slower than the pipelined instance but since the instances are operating in parallel they are effectively producing high enough outputs per unit time. The example below shows an adder implemented with two adders running at half the source clock rate. The half rate is obtained by a by-2 clock divider and the phase of the by-2 clock relative to the source clock is maintained by a counter.

Using the counter value as mux-demux select line, input data is correctly steered into and read out of the adder engines. Each instance may also be called a slice, if you don't like the word engine. The testbench supplies inputs and checks for the expected output. Because the output comes with considerable latency, the TB creates a model of the adder using the $past statement to delay the inputs and just "add" them up to create the expected data.

```verilog
module adder_engine
# (
parameter WIDTH = 64
 ) (
input logic clk ,
input logic [ WIDTH-1 : 0 ] add_op0 , add_op1 ,
output logic [ WIDTH-1 : 0 ] add_out
 ) ;

logic [ WIDTH-1 : 0 ] add_op0_reg , add_op1_reg ;
logic [ WIDTH-1 : 0 ] add_out_reg ;

always @ ( posedge clk ) begin
    add_op0_reg <= add_op0 ;
    add_op1_reg <= add_op1 ;
    add_out_reg <= add_op0_reg + add_op1_reg ;
end

assign add_out = add_out_reg ;

endmodule


module adder_parallel
# (
parameter WIDTH = 64 ,
parameter ENGINES = 2
 ) (
input logic clk ,
input logic rstn ,
input logic [ WIDTH-1 : 0 ] add_op0 , add_op1 ,
output logic [ WIDTH-1 : 0 ] add_out
 ) ;

logic clkbyn ;

logic [ WIDTH-1 : 0 ] add_op0_0 , add_op1_0 ;
logic [ WIDTH-1 : 0 ] add_out_0 ;

logic [ WIDTH-1 : 0 ] add_op0_1 , add_op1_1 ;
logic [ WIDTH-1 : 0 ] add_out_1 ;
logic cnt;

ehgu_cntr #(.WIDTH($clog2(ENGINES))) cntr (
.clk,
.rstn,
.sync_clr (0),
.en(1'b1),
.cnt
```

```verilog
);

always_ff @(posedge clk or negedge rstn) begin
  if(~rstn) begin
    add_op0_0 <= 0;
    add_op1_0 <= 0;
  end else if (cnt==0) begin
    add_op0_0 <=  add_op0;
    add_op1_0 <=  add_op1;
  end
end

always_ff @(posedge clk or negedge rstn) begin
  if(~rstn) begin
    add_op0_1 <= 0;
    add_op1_1 <= 0;
  end else if (cnt==1) begin
    add_op0_1 <=  add_op0;
    add_op1_1 <=  add_op1;
  end
end

adder_engine # ( .WIDTH ( WIDTH ) ) add_0 (
.clk ( slowclk ) ,
.add_op0 ( add_op0_0 ) ,
.add_op1 ( add_op1_0 ) ,
.add_out ( add_out_0 )
 ) ;

adder_engine # ( .WIDTH ( WIDTH ) ) add_1 (
.clk ( slowclk ) ,
.add_op0 ( add_op0_1 ) ,
.add_op1 ( add_op1_1 ) ,
.add_out ( add_out_1 )
 ) ;

always_ff @(posedge clk or negedge rstn) begin
  if(~rstn) begin
    add_out <= 0;
  end else if (cnt==0) begin
    add_out <= add_out_0;
  end else if (cnt==1) begin
    add_out <= add_out_1;
  end
end

ehgu_clkdiv # ( .DIVISION ( ENGINES ) ) clkdiv
 (
.clkin ( clk ) ,
.rstn ,
.en ( 1'b1 ) ,
.clkout ( slowclk )
 ) ;

endmodule
```

```systemverilog
module tb ;

parameter WIDTH = 64 ;

logic clk,rstn ;
logic [ WIDTH-1 : 0 ] add_op0 , add_op1 ;
logic [ WIDTH-1 : 0 ] add_out ;
logic [ WIDTH-1 : 0 ] expected ;
logic result ;

adder_parallel # ( .WIDTH ( WIDTH ) ) addp
 (
.clk ,
.rstn ,
.add_op0 ,
.add_op1 ,
.add_out
 ) ;

thee_clk_gen_module clk_gen ( .clk ( clk ) ) ;

localparam LATENCY=4;
always @(posedge clk)
  expected = $past(add_op0,LATENCY) + $past(add_op1,LATENCY);

initial begin
  result = 1 ;
  add_op0=0;add_op1=0;
  repeat (1) @(posedge clk);
  thee_utils_pkg::toggle_rstn (.rstn(rstn),.rst_low(10ns));
  repeat (LATENCY) @(posedge clk);
  for ( int i = 0 ; i < 8 ; i ++ ) begin
    add_op0 = $urandom ( ) * $urandom ;
    add_op1 = $urandom ( ) * $urandom ;

    repeat ( 1 ) @ ( posedge clk ) ;
    if ( add_out === expected ) begin
       $display ( "Vector Passed" ) ;
    end else begin
       $display ( "Vector Failed" ) ;
       result = 0 ;
    end

    $display ( "Test inputs %d + %d , output %d , expected %d" ,
      add_op0 , add_op1 , add_out, expected ) ;

  end

  if ( result )
   $display ( "All Vectors passed" ) ;
  else
   $display ( "Some Vectors failed" ) ;

  $finish ;
end

endmodule
```

```
run -all
Vector Passed
Test inputs   334401262994374040 + 16117745289704349328 , output           0 , expected            0
Vector Passed
Test inputs   562690299213603090 +   195842032687790490 , output           0 , expected         0
Vector Passed
Test inputs 17589414721914556394 +  2797292089888692004 , output           0 , expected            0
Vector Passed
Test inputs  2863164396650951056 + 14071002987375715712 , output           0 , expected            0
Vector Passed
Test  inputs  15959184159020107714  +    3563476362218418008  ,  output  16452146552698723368  ,  expected
16452146552698723368
Vector Passed
Test  inputs   2814696998044474564  +     135161794202782084  ,  output     758532331901393580  ,  expected
758532331901393580
Vector Passed
Test  inputs    2182109066361621048  +  15473335094032102971  ,  output   1939962738093696782  ,  expected
1939962738093696782
Vector Passed
Test  inputs   2703392884558147933  +      360401963488291832  ,  output  16934167384026666768  ,  expected
16934167384026666768
All Vectors passed
```

Synthesis results

We need to check if the parallel instances did indeed meet timing for higher speeds. For this
experiment, I set the main clock to 600MHz and ran synthesis twice. The first synthesis run is called
single and uses only the bare bone adder with registered input and output. The second synthesis run is
called parallel and uses the full multi-instance parallel adder. The divided clock and the main input
clock are not getting grouped into one clock tree during synthesis with Vivado tool. For now to ignore
the inter clock crossing paths and focus on design speed up, I have actually false pathed the inter clock
paths. This is not the right thing to do. A better example may use the Xilinx FPGA clocking resources.

For both single and parallel case
create_clock -period 1.6666 [get_ports clk] -name clk

For parallel case alone
create_clock -period 1.6666 [get_ports clk**] -name** clk

create_generated_clock -name slowclk -divide_by 2 \
-source **[**get_ports clk**]** -add -master_clock clk \
[get_pins clkdiv**/**clkout_reg**/**Q**]**

#THIS CONSTRAINT IS INVALID,
#USED HERE ONLY TO AVOID THE INTERCLOCK PATH WITH LARGE SKEW TO BE PRINTED
#USE CORRECT SYNTHESIS SETTINGS TO REDUCE THE SKEW BETWEEN SOURCE AND GENERATED
CLOCKS
set_clock_groups -asynchronous -group **[**get_clocks clk**]** -group **[**get_clocks slowclk**]**

Synthesis Script

foreach iter **{**parallel**} {**

```
  set dirname synth_${iter}.log
  file mkdir $dirname
  cd $dirname
  catch {
    close_project
  }
  create_project -force prj_${iter}
  add_files ../adder_engine.sv
  set top adder_engine
  if { $iter == "parallel"} {
    add_files ../adder_parallel.sv
    add_files ../../../ehgu/ehgu_config_pkg.sv
    add_files ../../../ehgu/ehgu_basic_pkg.sv
    add_files ../../../ehgu/ehgu_cntr.sv
    add_files ../../../ehgu/ehgu_clkdiv.sv
    set top adder_parallel
  }

  add_files ../adder_speedtest.sdc
  if { $iter == "parallel"} {
    add_files ../div_clk.sdc
  }

  set_property top $top [current_fileset]
  set_property generic WIDTH=64 [current_fileset]

  #synth_design -rtl
  #if { $iter == "parallel" } {
  #  set_property CLOCK_DELAY_GROUP clk_dom [get_nets {clk slowclk}]
  #}

  synth_design
  place_design
  route_design

  report_utilization > utilization.txt
  report_timing_summary > timing_summary.txt
  report_timing > timing.txt

  cd ..
}
```

The timing result shows that after parallel processing, the design is able to work at 600MHz (after the set_clock_groups asynchronous cheat code, for now!). The baseline single instance design can only work upto 460MHz (frequency of (1.666ns+0.5ns)). You can also see that to improve speed by 2x, area had to increase by almost 3x. This is because of the additional overhead to correctly feed the two adder engines. Note that some optimization can be done to the parallel version of the RTL to reduce this area impact, but, will come at the cost of readability.

sdiff synth_single.log/ synth_parallel.log/timing.txt

Command : report_timing	**Command** : report_timing
Design : adder_engine	Design : adder_parallel
Device : 7k70t-fbv676	Device : 7k70t-fbv676
Speed File : -1 PRODUCTION 1.12 2017-02-17	Speed File : -1 PRODUCTION 1.12 2017-02-17

Timing Report   Timing Report

Slack (VIOLATED) :   -0.500ns (required time - arrival | Slack (MET) :   0.052ns (required time - arrival t
 Source: add_op0_reg_reg[3]/C | Source: cntr/cnt_reg[0]/C
                       (rising edge-triggered cell FDRE | (rising edge-triggered cell FDCE
 Destination: add_out_reg_reg[61]/D | Destination: add_op0_0_reg[16]/CE
                       (rising edge-triggered cell FDRE | (rising edge-triggered cell FDCE
 Path Group: clk   Path Group: clk
 Path Type: Setup (Max at Slow Process Corner)   Path Type: Setup (Max at Slow Process Corner)
 Requirement: 1.667ns (clk rise@1.667ns - clk ri   Requirement: 1.667ns (clk rise@1.667ns - clk ri
 Data Path Delay: **2.156ns** (logic 1.580ns (73.269%) | Data Path Delay: **1.310ns** (logic 0.361ns (27.560%)
 Logic Levels: 17 (CARRY4=16 LUT2=1) | Logic Levels: 1 (LUT1=1)

sdiff synth_single.log/ synth_parallel.log/utilization.txt

| **Command   : report_utilization** | **Command   : report_utilization**
| Design   : adder_engine | | Design   : adder_parallel
| Device   : 7k70tfbv676-1 | Device   : 7k70tfbv676-1
| Design State : Routed | Design State : Routed

Site Type	Used	Fixed	Available	Util%	Site Type	Used	Fixed	Available	Util%
Slice LUTs	64	0	41000	0.16	Slice LUTs	195	0	41000	0.48
LUT as Logic	64	0	41000	0.16	LUT as Logic	195	0	41000	0.48
LUT as Memory	0	0	13400	0.00	LUT as Memory	0	0	13400	0.00
Slice Registers	192	0	82000	0.23	Slice Registers	707	0	82000	0.86
Register as Flip Flop	192	0	82000	0.23	Register as Flip Flop	707	0	82000	0.86
Register as Latch	0	0	82000	0.00	Register as Latch	0	0	82000	0.00
F7 Muxes	0	0	20500	0.00	F7 Muxes	0	0	20500	0.00
F8 Muxes	0	0	10250	0.00	F8 Muxes	0	0	10250	0.00

Exercise

Can you remove the false path constraint and reduce the skew between the source and generated clocks?

Can you generalize the code to work for any number of engines, say, 4 or 8 and not just 2 as given?

Multi Cycle Logic – Division

Assume that you have 8-bit digital samples of an analog signal coming into your logic at 500MHz and you want the average of say N samples, with N in the range of 100 – 1000. suppose samples come in continuous bursts followed by a gap of 50 clock cycles of no data. The average is required per burst, the latency is not a problem. You can see that per burst there are 100-1000 8b accumulating type additions and then one division operation to be performed for average output. So, you could architect the system to add 8b sample per cycle and after the last data is available, do a single cycle division. You could use an accumulator that adds 8b data every clock cycle and then a divider (just the / operator) that divides the accumulated value by 100 – 1000 in one cycle. Accumulating 1000 8bits could take up to 8+10 bits. An 18b by 8b division in one 500MHz cycle can take more area and power than a divider working at only 10MHz. All that was needed was a division for every burst. A burst last a minimum of 100 + 50 cycles. Is there a way to spread out the division operation over many cycles and save power

and area? Sure! The default option is to use a serial divider implementation with shift and subtract. But that divider takes some logic to be coded. If you do not have a verified serial divider module, it adds time to the design process. There is another way that takes very little design time and takes only little bit more area than serial implementations. The code looks very similar to the single cycle divider using the / operator. But now there is another counter to wait for the divider to complete in more than one cycle and some logic to start and stop the counter. Say, the divider is optimized by synthesis tool to take 10 cycles per division, the output of the divider is undefined during 0-9 cycles. A counter keeps track of this and outputs a valid signal at the 10th cycle. If you want to increase the speed to 8 cycles, all you have to do is change the counter and also the timing constraint which looks some thing like this.

```
set_multicycle_path N -from <divisor/dividend q outputs> -to <quotient/remainder d inputs or
clock inputs> -setup
set_multicycle_path N-1 -from <divisor/dividend q outputs> -to <quotient/remainder d inputs or
clock inputs> -hold
```

The hold constraint uses a different cycle count than the setup constraint.

If you were to use the serial divider logic, changing the number of cycles for the division operation is not easy. Compared to serial implementation the multicycle implementation may take more area because all the division logic is instantiated in the chip and no logic is reused. In the serial implementation, the subtractor is reused in every cycle of division. In multicycle method, there is one subtractor per stage. Each subtractor is allowed to take longer to complete than the single cycle implementation. The area savings compared to single cycle method occurs because smaller logic gates are sufficient to target lower speed. Compared to serial implementation, the multicycle implementation may occasionally be smaller in size because the serial implementation needs memory to store intermediate results whereas the multicycle method stores intermediate results on gate outputs.
However, there is a price to pay for the laziness in using a multicycle division. Tools that rely on single cycle timing may be confused by these special constraints. One such thing I know of is in DFT. At speed scan test creates patterns that exercise logic in one cycle by default. If there are multicycle path timing, trying one cycle excitation will make the scan patterns fail. This is to be overcome by supplying which paths are multicycle to the ATPG pattern generator, so that, it will create patterns that are tested only after the said cycles are allowed to happen. A bigger risk is in messing with the timing constraints. If you carelessly declare some real single cycle paths as multicycle then it is not possible to catch in RTL verification. It can only be caught in gate level simulation that is time consuming. So, use this trick with caution.

The example average calculator works in 3 stages. The input data is summed up every cycle and then one or more cycles are consumed dividing the sum by the number of samples received and at last the average is output till the next set of data comes in. Strategic use of $monitor statement helps in easy debug without waveform viewers.

The testbench instantiates 3 versions of the average logic. Two are obvious, one and many cycle division logic and the third is the golden behavioral model of average value.

```
module sample_avg
#(
parameter MAX_SAMPLES=1000,
parameter WIDTH=8,
parameter DIV_CYCLES=1
)(
```

```systemverilog
  input logic clk,
  input logic rstn,
  input logic [WIDTH-1:0] data_in,
  input logic dvalid,
  output logic [WIDTH-1:0] avg_out,
  output logic avg_valid
);

  localparam SUM_WIDTH=$clog2({ WIDTH {1'b1}} * MAX_SAMPLES);
  localparam CNT_WIDTH = $clog2(MAX_SAMPLES);

  typedef enum logic [3:0] {IDLE=0,ACCUMULATING=1,DIVIDING=2,AVG_OUT=3} st_t;
  st_t state,next;

  logic [SUM_WIDTH-1:0] sum, sum_next, avg_next;
  logic [CNT_WIDTH-1:0] sample_cnt ;
  logic cnt_clr;
  logic last_div_cyc;

  always @(posedge clk, negedge rstn) begin
    if (!rstn) begin
      state <= IDLE;
    end else begin
      state <= next;
    end
  end

  always_comb begin
    next = IDLE;
    case (state)
      IDLE:         if ( dvalid )
                      next = ACCUMULATING;

      ACCUMULATING : if ( dvalid )
                      next = state;
                    else
                      next = DIVIDING;

      DIVIDING :    if ( last_div_cyc )
                      next = AVG_OUT;
                    else
                      next = state;

      AVG_OUT :     if ( dvalid )
                      next = ACCUMULATING;
                    else
                      next = state;
    endcase
  end

  assign cnt_clr = ( state == AVG_OUT && next== ACCUMULATING ) ? 1 : 0 ;

  ehgu_cntr #(.WIDTH(CNT_WIDTH)) cntr (
  .clk,
  .rstn,
  .sync_clr (cnt_clr),
  .en(dvalid),
  .cnt (sample_cnt)
  );

  //initial $monitor ("sc %d %d %t",sum,avg_out, $time);

  generate
    if ( DIV_CYCLES==1 ) begin
      : gen_1cyc_div
      assign last_div_cyc = state== DIVIDING;
    end else begin
      : gen_manycyc_div
      logic div_cnt_clr;
```

```systemverilog
    logic [$clog2(DIV_CYCLES)-1:0] div_cyc_cnt;
    logic div_cnt_en;

    assign last_div_cyc = (state== DIVIDING) && (div_cyc_cnt==DIV_CYCLES-1);
    assign div_cnt_en = state == DIVIDING ;

    ehgu_cntr #(.WIDTH(CNT_WIDTH)) cntr_div_cyc (
      .clk,
      .rstn,
      .sync_clr (div_cnt_clr),
      .en(div_cnt_en),
      .cnt (div_cyc_cnt)
    );
    // initial $monitor ("sc %s %d %d %d %t",state.name, div_cyc_cnt, sum,avg_out, $time);

  end
endgenerate

always_comb begin
  if ( cnt_clr )
    sum_next = 0;
  else if ( dvalid )
    sum_next = sum + data_in;
  else
    sum_next = sum;
end

always @(posedge clk, negedge rstn) begin
  if (!rstn) begin
    sum <= 0;
    avg_out <= 0;
    avg_valid <= 0;
  end else begin
    sum <= sum_next;
    avg_out <= avg_next;
    avg_valid <= state == AVG_OUT;
  end
end

always_comb begin
  avg_next = avg_out;
  if ( state == DIVIDING ) begin
    avg_next = sum / sample_cnt;
    avg_next = avg_next[WIDTH-1:0];
  end
end

endmodule

module tb ;

parameter WIDTH=8;

logic clk,rstn;
logic [WIDTH-1:0] expected;
logic result ;

logic [WIDTH-1:0] data_in;
logic dvalid;
logic [WIDTH-1:0] avg_out_1c;
logic avg_valid_1c;

logic [WIDTH-1:0] avg_out_manyc;
logic avg_valid_manyc;

logic [WIDTH-1:0] avg_out_golden;
int unsigned sum_golden,samp_cnt;

thee_clk_gen_module clk_gen (.clk(clk));
```

```verilog
sample_avg #(.DIV_CYCLES(1), .WIDTH(WIDTH)) avg_1cyc
(
.clk ,
.rstn ,
.data_in ,
.dvalid ,
.avg_out ( avg_out_1c ) ,
.avg_valid ( avg_valid_1c )
);

sample_avg #(.DIV_CYCLES(10), .WIDTH(WIDTH)) avg_manycyc
(
.clk ,
.rstn ,
.data_in ,
.dvalid ,
.avg_out ( avg_out_manyc ) ,
.avg_valid ( avg_valid_manyc )
);

initial begin
    result = 1;

    dvalid = 0 ;
    sum_golden = 0; samp_cnt = 0 ;
    repeat (1) @(posedge clk);
    thee_utils_pkg::toggle_rstn (.rstn(rstn),.rst_low(100ns));
    repeat (2) @(posedge clk);

    for ( int i = 0 ; i<100; i++ ) begin
        dvalid = 1;
        data_in = $random();
        sum_golden += data_in;
        samp_cnt ++ ;
        @(posedge clk);
    end

    for ( int i = 0 ; i<50; i++ ) begin
        dvalid = 0;
        @(posedge clk);
    end

    avg_out_golden = sum_golden / samp_cnt;
    $display(sum_golden);
    wait (avg_valid_1c && avg_valid_manyc);
    $display ( "Avg 1c %d, avg many c %d , avg golden %d", avg_out_1c, avg_out_manyc, avg_out_golden);

    if ( (avg_out_1c == avg_out_manyc) && (avg_out_1c == avg_out_golden ) ) begin
        $display ("Vector Passed");
    end else begin
        $display ("Vector Failed");
        result = 0 ;
    end

    if ( result )
        $display ("All Vectors passed");
    else
        $display ("Some Vectors failed");

    $finish;
end

endmodule
```

```
run -all
    12142
Avg 1c 121, avg many c 121 , avg golden 121
```

Sample synthesis script

Because multicycle path based logic design is in a way design by timing constraints, the code does not make much sense without looking at corresponding timing and resource usage reports. For that, we need to commit to some FPGA. The synthesis script here reads the design files, timing constraints, sets the number of cycles for division and runs synthesis, placement and routing for the default FPGA in the Vivado software. At the end, timing and resource usage reports are generated for comparison. The script has two iterations, one for single cycle design and another for multicycle design. Note that this script is not optimized. You may want to use the best script in real designs by consulting Vivado synthesis manual.

```
foreach iter {1 10} {
  set dirname synth${iter}c.log
  file mkdir $dirname
  cd $dirname
  catch {
    close_project
  }
  create_project -force cycle_test${iter}
  add_files ../sample_avg.sv
  add_files ../../../ehgu/ehgu_cntr.sv
  add_files ../sample_avg.sdc

  if { $iter != 1 } {
    add_files ../multi.sdc
  }

  set_property top sample_avg [current_fileset]
  set_property generic DIV_CYCLES=$iter [current_fileset]

  synth_design
  place_design
  route_design

  report_utilization > utilization.txt
  report_timing_summary > timing_summary.txt
  report_timing > timing.txt

#  close_project
  cd ..
}
```

Common timing constraints
```
create_clock -period 3 clk
```

Constraint only for mult cycle design
```
set_multicycle_path 10 -from sum_reg[*]/C -to avg_out_reg[*]/D -setup
set_multicycle_path 9 -from sum_reg[*]/C -to avg_out_reg[*]/D -hold

set_multicycle_path 10 -from cntr/cnt_reg[*]/C -to avg_out_reg[*]/D -setup
set_multicycle_path 9 -from cntr/cnt_reg[*]/C -to avg_out_reg[*]/D -hold
```

Result from synthesis

shell> **sdiff** synth1c.log/timing_summary.txt synth10c.log/timing_summary.txt

On the left side of the output from sdiff command there is the single cycle experiment and the right side there is the multicycle experiment. You can see that the timing path to average output signal is the most timing critical one because that has the divider. When using single cycle constraint there is a negative slack of about -17ns. When using 10 cycle timing there is positive slack. So, in this case a single cycle divider is not possible to implement in one 3ns period or 333MHz clock for the chosen FPGA device.

```
Max Delay Paths                    Max Delay Paths
-----------------------------------------------------------------------------------------------------------------
Slack (VIOLATED) :     -17.201ns  (required time - arrival | Slack (MET) :       0.922ns  (required time - arrival t
  Source:           cntr/cnt_reg[2]/C       |   Source:           FSM_sequential_state_reg[0]/C
                    (rising edge-triggered cell FDCE              (rising edge-triggered cell FDCE
  Destination:       avg_out_reg[0]/D       |   Destination:       avg_out_reg[3]/CE
                    (rising edge-triggered cell FDCE              (rising edge-triggered cell FDCE
  Path Group:         clk               Path Group:          clk
```

Difference in Usage of FPGA resources
Note the reduced usage in some resources even though the multi cycle version has additional code. The main difference is that multicycle meets timing and the single cycle version does not!

sdiff synth1c.log/utilization.txt synth10c.log/utilization.txt

```
+-------------------------------------------+------+-------+ |   +-------------------------------------------+------+-------+--
| Slice                         | 80 |    0 || | Slice                         | 64 |    0 |
|   SLICEL                      | 69 |    0 || |   SLICEL                      | 47 |    0 |
|   SLICEM                      | 11 |    0 || |   SLICEM                      | 17 |    0 |
| LUT as Logic                  | 231 |   0 ||| LUT as Logic                 | 228 |    0 |
|   using O5 output only            | 0 |    || |   using O5 output only            | 0 |    |
|   using O6 output only          | 215 |    || |   using O6 output only          | 204 |    |
|   using O5 and O6               | 16 |    ||| using O5 and O6               | 24 |    |
| LUT as Memory                   | 0 |    0 || | LUT as Memory                 | 0 |    0 |
|   LUT as Distributed RAM          | 0 |    0 || |   LUT as Distributed RAM          | 0 |    0 |
|   LUT as Shift Register          | 0 |    0 || |   LUT as Shift Register        | 0 |    0 |
| Slice Registers                | 39 |    0 || | Slice Registers               | 43 |    0 |
|   Register driven from within the Slice  | 20 |    || |   Register driven from within the Slice | 43 |    |
|   Register driven from outside the Slice | 19 |    || |   Register driven from outside the Slice | 0 |    |
|     LUT in front of the register is unused | 16 |   ||| Unique Control Sets             | 3 |    |
|     LUT in front of the register is used  | 3 |    ||   +-------------------------------------------+------+-------+--
| Unique Control Sets            | 3 |    |<
+-------------------------------------------+------+-------+ <
```

Single Port ROM

A read only memory is used to hold data for implementing complex functions, configuration parameters for the chip and more. A model of a ROM is fairly simple, for every address input there is a data output. The models for ROM are set apart in the level of reusability and scale of usage. A chip could have 40 ROMs and it can get messy very quickly for some forms of models.

Hardcoded table

The easiest way is to use a case block with the required address-data pair. This method is not very reusable. Suppose you want to change just one data. Then you have to edit the file that contains code that maps to logic. Note that this model is synthesizable into gates. When synthesized this model becomes a gate level combinational logic that implements a function with address as the input.

```systemverilog
module rom_comb
# (
parameter AWIDTH = 3 ,
parameter DWIDTH = 8
 ) (
input logic [ AWIDTH-1 : 0 ] addr ,
output logic [ DWIDTH-1 : 0 ] dout
 ) ;

always_comb
 case ( addr )
 0 : dout = 'h 42 ;
 1 : dout = 'h 93 ;
 2 : dout = 'h 25 ;
 3 : dout = 'h 35 ;
 4 : dout = 'h 52 ;
 5 : dout = 'h 41 ;
 6 : dout = 'h DA ;
 7 : dout = 'h A3 ;
 default : dout = 'h 00 ;
 endcase

endmodule
```

Separating Data and Logic

By keeping the data separately into a rom_contents.dat and the circuit model into another file, the creator of ROM data will only have to supply the data file. The SV utility functions $fopen() and $fscanf() are used at the start of the simulation in an initial block to load the ROM. Note that in a real ROM circuit there is no special load operation at power up. ROM circuit is alive and kicking right after power up. The load operation is only an artifact of the simulation model.

```systemverilog
module rom_comb_file_read
# (
parameter AWIDTH = 3 ,
parameter DWIDTH = 8
 ) (
input logic [ AWIDTH-1 : 0 ] addr ,
output logic [ DWIDTH-1 : 0 ] dout
 ) ;
integer fp ;
logic [ DWIDTH-1 : 0 ] rom_data [ 2 ** AWIDTH ] ;
string rom_file = "rom_contents.dat" ;
bit result ;

initial begin
  fp = $fopen ( rom_file , "r" ) ;
  for ( int i = 0 ; i < 2 ** AWIDTH ; i ++ ) begin
    result = $fscanf ( fp , "%h" , rom_data [ i ] ) ;
```

```systemverilog
    end
end

assign dout = rom_data [ addr ] ;

endmodule
```

Data inside rom_contents.dat
22
93
55
65
52
41
A3
DA

```systemverilog
module tb ;

timeunit 1ns ;
timeprecision 1ps ;

import thee_utils_pkg :: * ;
logic clk ;

logic [ 7 : 0 ] dout ;
logic [ 2 : 0 ] addr ;

 // rom_comb rom (
rom_comb_file_read rom (
.addr ,
.dout
 ) ;

thee_clk_gen_module clk_gen ( .clk ) ;

initial begin
  repeat ( 16 ) begin
   #1 ;
   addr = $urandom ( ) ;
   #1 ;
   $display ( "Addr %h data %h" , addr , dout ) ;
  end
  $finish ;
end

endmodule
```

Simulation output
run -all
Addr 1 data 93
Addr 1 data 93
Addr 6 data a3
Addr 6 data a3

Instead of fscanf, readmemh or readmemb is used as well. LUT creation at compile time idea may be used to populate the ROM data too. The difficulty in ROM modeling is how can the data file be made easy to use for few tens or hundreds of ROMs used in a large chip simulation environment. Sometimes, `define names are used for the contents file or `include is used which is not very scalable for large projects. Imagine using a name like rom.dat for all files. They would all collide and the simulation will get corrupted. Sometimes the file name is supplied at compile time with -define <filename> option of the simulator. Which method is better is open for debate.

Exercise
Large ROMs may register the output and or input signals with flip flops for timing. Can you add input and output register stages and controlled by parameters?

Single Port RAM

A memory with read and write access is used in computations in a chip. This is primarily for two main purposes – aligning delays between different streams and saving intermediate results of computation. Intermediate results occur because all computations in a chip cannot happen in one clock cycle. It is typically broken down into multiple clock cycles. The following code shows a simple model of a RAM with one port. At any one point of time, you can write or read the memory. You cannot, however, do both read and write at the same time in a single port RAM. Dual and multi port RAMs may have many read and write ports. In ASIC, a RAM is compiled using a special memory compiler software available from IP vendors like ARM or directly from the foundry like TSMC or GF. In the absence of a memory compiler for your target process, you could just synthesize the code of the memory into flip flops. The area occupied by flip flops to make a memory is 2x or more compared to the compiled RAM from special memory bit cells. In FPGA, the code for RAM is recognized by the synthesis tool and automatically mapped to RAM cells available in the target FPGA. Sadly, to the best of my knowledge, this automatic mapping from code is not available for ASIC and the mapping job is done by the designer. Note that a RAM cannot be reset to 0. If you need initialized memory you need to use flops or have logic that loads the RAM from a ROM before using the RAM.

```
module ram_single_port
# (
parameter DEPTH = 3 ,
parameter WIDTH = 8
 ) (
input logic clk ,
input logic ce ,
input logic r_wn ,
input logic [ $clog2 ( DEPTH ) -1 : 0 ] addr ,
input logic [ WIDTH-1 : 0 ] wdata ,
output logic [ WIDTH-1 : 0 ] rdata
 ) ;

logic [ WIDTH-1 : 0 ] mem_arr [ DEPTH ] ;
```

```systemverilog
always_ff @ ( posedge clk ) begin
  if ( ce ) begin
    if ( r_wn ) begin
      rdata <= mem_arr [ addr ] ;
    end else begin
      rdata <= 'x ;
      mem_arr [ addr ] <= wdata ;
    end
  end else begin
    rdata <= 'x ;
  end
end

endmodule

module tb ;

timeunit 1ns ;
timeprecision 1ps ;

import thee_utils_pkg :: * ;

parameter DEPTH = 32 ;
parameter AWIDTH = $clog2 ( DEPTH ) ;
parameter DWIDTH = 8 ;

logic clk ;
logic [ DWIDTH-1 : 0 ] wdata ;
logic [ DWIDTH-1 : 0 ] rdata ;
logic [ AWIDTH-1 : 0 ] addr ;
logic ce ;
logic r_wn ;
logic result ;

logic [ DWIDTH-1 : 0 ] mem_mirror [ DEPTH ] ;

 // rom_comb rom (
ram_single_port # ( .DEPTH ( DEPTH ) , .WIDTH ( DWIDTH ) ) ram (
.clk ,
.addr ,
.rdata ,
.wdata ,
.ce ,
.r_wn
 ) ;

thee_clk_gen_module clk_gen ( .clk ) ;

initial begin
  result = 1 ;
  repeat ( 1 ) @ ( posedge clk ) ;
  for ( int i = 0 ; i < DEPTH ; i ++ ) begin
    addr = i ;
    wdata = $urandom ( ) ;
```

```
    ce = 1 ;
    r_wn = 0 ;
    mem_mirror [ addr ] = wdata ;
    $display ( "Write %h data to addr %h " , wdata , addr ) ;
    repeat ( 1 ) @ ( posedge clk ) ;
   end
  for ( int i = 0 ; i < 3 * DEPTH ; i ++ ) begin
    addr = $urandom ( ) ;
    ce = 1 ;
    r_wn = 1 ;
    repeat ( 1 ) @ ( posedge clk ) ;
    if ( mem_mirror [ addr ] === rdata ) begin
      $display ( "Passed : Data %h at %h " , rdata , addr ) ;
    end else begin
      $display ( "Failed : Data %h at %h is different from expected data %h " , rdata , addr ,
mem_mirror [ addr ] ) ;
      result = 0 ;
    end
  end
  print_test_result ( result ) ;
  $finish ;
end

endmodule

run -all
Write db data to addr 00
Write bc data to addr 01
.
.
Write fd data to addr 1e
Write c7 data to addr 1f
Passed : Data 44 at 1b
Passed : Data bc at 01
.
.
Passed : Data fc at 02
Passed : Data 1d at 16
Test Pass
```

The function of clog2() helps in deriving the width of the address bus from the depth of the memory. The depth of the memory may also be referred to as number of words with one datum being called as a word. Note that word in some other situations mean a group of 32 bits. The meaning of word should be understood based on context. The physical memory is an array of special memory cells. SV does not model the behavior of these cells well and we don't care too much about the intricacies of the bit cells. Just an array variable is sufficient to represent the actual physical memory. Logically, a RAM is imagined as a table of bits arranged in rows and columns. Each row is said to contain a word or one datum. The number of rows gives the size of the RAM. Number of columns is called the bitwidth of the words. Not that physically, the RAM may be organized in any combination of rows and columns. The logical view is mostly the understanding that resides in the minds of chip designers and software engineers.

A chip enable signal activates the memory, otherwise the memory may be in a low power state. Read or write signal indicates what operation is commanded. Detailed timing behavior of the memory is not modeled.

Exercise

Can you think of a use for a memory with only write access, without read access? Can you modify the single port RAM model to describe a many times write-only memory?

Suppose you do not have a dual port RAM with simultaneous read and write access. Instead, you have a single port RAM that can work up to 200MHz, you have a clock of 100MHz and its divided by 2 version of 50MHz. Say you want a dual port RAM with simultaneous read and write access for a logic that is operating at 50MHz. Can you create a wrapper around the single port RAM to realize a dual port RAM?

Is there a resettable sram, where could it be used?

Dual Port RAM

A dual port RAM has two ports, write and read. This creates a new problem that you could write and read the same address at the same time. The model below shows a check for this contention case useful for simulation. That piece of logic inside ifndef SYNTHESIS is not to be synthesized. If synthesized it will create unknown states in the design. The model of the memory is simple the write side has its own enable, addr and clock while the read side has another set of signals. The actual physical circuit is pretty involved and beyond the scope of this book. Note that similar to single port RAM, you could synthesize the model into flip flops and gates if there is no 2 port RAM cell in your library.   FPGA synthesis tools automatically infers 2 port RAMs during synthesis.

```systemverilog
module ehgu_ram_dual_port
#(
parameter DEPTH=3,
parameter WIDTH=8
)(
input logic wclk,
input logic wenable,
input logic [$clog2(DEPTH)-1:0] waddr,
input logic [WIDTH-1:0] wdata,

input logic rclk,
input logic renable,
input logic [$clog2(DEPTH)-1:0] raddr,
output logic [WIDTH-1:0] rdata
);

logic [WIDTH-1:0] mem_arr [DEPTH];

always_ff @(posedge rclk) begin
   if ( renable ) begin
     rdata <= mem_arr[raddr];
   end else begin
     rdata <= 'x;
   end
end

always_ff @(posedge wclk) begin
```

```verilog
    if ( wenable ) begin
      mem_arr[waddr] <= wdata;
    end
  end

  `ifndef SYNTHESIS
  always @(*) begin
    if (renable && wenable && (raddr==waddr)) begin
      $display("Read/Write Contention at addr %h", raddr);
    end
  end
  `endif

endmodule


module tb ;

timeunit 1ns ;
timeprecision 1ps ;

import thee_utils_pkg :: * ;

parameter DEPTH = 32 ;
parameter AWIDTH = $clog2 ( DEPTH ) ;
parameter DWIDTH = 8 ;

parameter SHOW_CONTENTION = 0 ;

logic wclk ;
logic [ DWIDTH-1 : 0 ] wdata ;
logic [ AWIDTH-1 : 0 ] waddr ;
logic wenable ;

logic rclk ;
logic [ DWIDTH-1 : 0 ] rdata ;
logic [ AWIDTH-1 : 0 ] raddr ;
logic renable ;

logic result ;

logic [ DWIDTH-1 : 0 ] mem_mirror [ DEPTH ] ;

 // rom_comb rom (
ehgu_ram_dual_port # ( .DEPTH ( DEPTH ) , .WIDTH ( DWIDTH ) ) ram (
.wclk ,
.wenable ,
.waddr ,
.wdata ,
.rclk ,
.renable ,
.raddr ,
.rdata
 ) ;
parameter real FREQ = 100 ;
```

```verilog
thee_clk_gen_module # ( .FREQ ( FREQ ) ) clk_gen_i0 ( .clk ( wclk ) ) ;

thee_clk_gen_module # ( .FREQ ( FREQ * 1.05 ) ) clk_gen_i1 ( .clk ( rclk ) ) ;


initial begin
  result = 1 ;

  for ( int i = 0 ; i < 3 * DEPTH ; i ++ ) begin
    repeat ( 1 ) @ ( posedge wclk ) ;
    repeat ( 1 ) @ ( posedge rclk ) ;

    raddr = i ; // $urandom ( ) ;
    if ( SHOW_CONTENTION ) begin
     // waddr = $urandom ( ) ;
      do
      waddr = $urandom ( ) ;
      while ( waddr == raddr ) ;
    end else begin
      waddr = raddr + DEPTH / 2 ;
    end
    $display ( "ra %h wa %h" , raddr , waddr ) ;

    fork
    begin
      wdata = $urandom ( ) ;
      wenable = 1 ;
      mem_mirror [ waddr ] = wdata ;
      $display ( "Write %h data to addr %h " , wdata , waddr ) ;
      repeat ( 1 ) @ ( posedge wclk ) ;
    end
    begin
      renable = 1 ;
      repeat ( 1 ) @ ( posedge rclk ) ;
      $display ( "Read Data %h at %h " , rdata , raddr ) ;
      if ( mem_mirror [ raddr ] === rdata ) begin
        $display ( "Passed : Data %h at %h " , rdata , raddr ) ;
      end else begin
        $display ( "Failed : Data %h at %h is different from expected data %h " , rdata , raddr ,
mem_mirror [ raddr ] ) ;
        result = 0 ;
      end
    end
    join
  end
  print_test_result ( result ) ;
  $finish ;
end

endmodule
```

```
run -all
ra 00 wa 10
Write 58 data to addr 10
Read Data xx at 00
Passed : Data xx at 00
ra 01 wa 11
```

Write 35 data to addr 11
Read Data xx at 01
Passed : Data xx at 01
.
.
.
ra 1e wa 0e
Write cc data to addr 0e
Read Data f0 at 1e
Passed : Data f0 at 1e
ra 1f wa 0f
Write f0 data to addr 0f
Read Data 0c at 1f
Passed : Data 0c at 1f
Test Pass

Shift Register using Dual Port RAM

In many designs, data needs to be delayed by some N number of cycles. This happens to match the data stream delay to some other event. For example, the audio and video channels of an incoming Internet movie video stream needs to be synchronized. Otherwise the movie would become un-watchable. Have you watched a movie with a misaligned sound compared to the video, arriving many seconds earlier or later than the video? It is a horrible experience. I actually got a headache doing that and then switched off the TV!

You could just use the flip flip delay module example shown earlier but that would take a large amount of area compared to using a memory. Instead, you could "simulate" the delay by making the memory read wait for N cycles after the memory write.

```systemverilog
module ehgu_sr_mem
# (
parameter SHIFT = 20 ,
parameter WIDTH = 8 ,
parameter MEM_DEPTH = 128
 ) (
input logic clk ,
input logic rstn ,
input logic en ,
input logic [ WIDTH-1 : 0 ] data_in ,
output logic [ WIDTH-1 : 0 ] data_out
 ) ;

localparam AWIDTH = $clog2 ( MEM_DEPTH ) ;

logic [ AWIDTH-1 : 0 ] waddr ;
logic [ AWIDTH-1 : 0 ] raddr ;

always_ff @ ( posedge clk , negedge rstn ) begin
  if ( !rstn ) begin
    waddr <= 0 ;
    raddr <= MEM_DEPTH - SHIFT ;
  end else if ( en ) begin
    waddr <= ( waddr + 1 ) %MEM_DEPTH ;
    raddr <= ( raddr + 1 ) %MEM_DEPTH ;
  end
end
```

```systemverilog
ehgu_ram_dual_port # ( .DEPTH ( MEM_DEPTH ) , .WIDTH ( WIDTH ) ) dmem_i
 (
.wclk ( clk ) ,
.wenable ( en ) ,
.waddr ,
.wdata ( data_in ) ,
.rclk ( clk ) ,
.renable ( en ) ,
.raddr ,
.rdata ( data_out )
 ) ;

endmodule

module tb ;

timeunit 1ns ;
timeprecision 1ps ;

import thee_utils_pkg :: * ;

parameter DEPTH = 32 ;
parameter AWIDTH = $clog2 ( DEPTH ) ;
parameter DWIDTH = 8 ;

parameter SHIFT = 3 ;

logic clk , rstn ;
logic [ DWIDTH-1 : 0 ] data_in ;
logic [ DWIDTH-1 : 0 ] data_out , expected_data ;
logic en ;

logic result ;

logic [ DWIDTH-1 : 0 ] mem_mirror [ DEPTH ] ;

 // rom_comb rom (
ehgu_sr_mem # ( .SHIFT ( SHIFT ) , .MEM_DEPTH ( DEPTH ) , .WIDTH ( DWIDTH ) ) sr_mem (
.clk ,
.rstn ,
.en ,
.data_in ,
.data_out
 ) ;

parameter real FREQ = 100 ;
thee_clk_gen_module # ( .FREQ ( FREQ ) ) clk_gen_i0 ( .clk ( clk ) ) ;

initial begin
  result = 1 ;
  en = 1 ;
  toggle_rstn ( .rstn ( rstn ) ) ;
  fork
  for ( int i = 0 ; i < 3 * DEPTH ; i ++ ) begin
```

```verilog
      repeat ( 1 ) @ ( posedge clk ) ;

     data_in = $urandom ( ) ;
   end
   begin
    repeat ( SHIFT ) @ ( posedge clk ) ;
    while ( 1 ) begin
     repeat ( 1 ) @ ( posedge clk ) ;
     expected_data = $past ( data_in , SHIFT , 1 , @ ( posedge clk ) ) ;
     if ( data_out === expected_data ) begin
       $display ( "P - output data %h expected data %h" , data_out , expected_data ) ;
     end else begin
       $display ( "F - output data %h expected data %h" , data_out , expected_data ) ;
       result = 0 ;
     end
    end
   end

  join_any

  print_test_result ( result ) ;
  $finish ;
end

endmodule
```

```
run -all
P - output data xx expected data xx
P - output data 7f expected data 7f
.
.
.
P - output data d3 expected data d3
P - output data 30 expected data 30
Test Pass
```

Exercise
Suppose the write address flop MSB is corrupted by flipping a 1 to 0, how does the read address change? Hint: read address is ideally made to depend on write address.

First In First Out (FIFO)

FIFOs are the mainstay of any data transport logic. You could think of FIFOs as a gearbox or a power transformer. The average rate of input data should be same as the average rate of data going out. But instantaneous rate can be very different. For example, you could input data at 1Gbps for a brief period and the output could be at 10Mbps for a long period. Does this resemble a gearbox that takes engine power at high rpm and delivers the power to the wheels at low rpm? Or the power transformer that accepts high voltage and outputs a low voltage? In the gearbox and transformer analogy the constant is power. The input power should be equal to the output power, assuming no power loss.

The classic FIFO uses 2 port memory, one port for writing into the memory and another for reading from the memory.

Synchronous

A FIFO can have many variations. Lets look at the easy one to start. If the data coming in and the data going out are clocked by the same clock then the design reduces to a relatively simple one. Lets study the FIFO module by module. The fifo is split into a module that interfaces to a memory – RAM or a register file or a composite memory or just flop array. Another module does all the calculation of which address to write and read. For now, ignore the SYNC_STG* parameters. The SYNC_TYPE parameter is set to 1 to denote synchronous operation. The input to the module has two clocks and two resets. In the synchronous usage both clocks and resets are to be connected to the same clock and reset sources. The display statements may be commented out for debug purpose.

```verilog
module ehgu_fifo
# (
parameter SYNC_TYPE = 0,
parameter SYNC_STG_W2R = 2,
parameter SYNC_STG_R2W = 2,
parameter WIDTH = 8 ,
parameter DEPTH = 128
 ) (
input logic clk0 ,
input logic wrstn ,
input logic en ,
input logic din_valid ,
input logic [ WIDTH-1 : 0 ] din ,

input logic clk1 ,
input logic rrstn ,
output logic dout_valid ,
output logic [ WIDTH-1 : 0 ] dout
 ) ;

localparam AWIDTH = $clog2 ( DEPTH ) ;

logic [ AWIDTH-1 : 0 ] waddr ;
logic [ AWIDTH-1 : 0 ] raddr ;
logic wenable ;
logic renable ;

ehgu_fifo_logic # ( .DEPTH ( DEPTH ) , .WIDTH ( WIDTH ), .AWIDTH ( AWIDTH ),
.SYNC_TYPE(SYNC_TYPE), .SYNC_STG_R2W(SYNC_STG_R2W), .SYNC_STG_W2R(SYNC_STG_W2R) )
ehgu_fifo_logic_i
 (
.wclk ( clk0 ) ,
.wrstn ,
.rrstn ,
.en ( en ) ,
.din_valid ,
.wenable ,
.waddr ,
.rclk ( clk1 ) ,
.renable ,
.dout_valid ,
.raddr
```

```verilog
    ) ;

ehgu_fifo_mem # ( .DEPTH ( DEPTH ) , .WIDTH ( WIDTH ), .AWIDTH ( AWIDTH ) )
ehgu_fifo_mem_i
 (
.wclk ( clk0 ) ,
.wenable ,
.waddr ,
.wdata ( din ) ,
.rclk ( clk1 ) ,
.renable ,
.raddr ,
.rdata ( dout )
 ) ;

// always @(posedge clk1) begin
//   $display("wa %d, we %b, wd %h, ra %d, re %b, rd %h", waddr, wenable, din, raddr,
renable,dout);
//   $display("do %h, dv %b", dout, dout_valid);
// end
endmodule
```

The fifo memory module is just a wrapper for the dual port RAM model for now. The idea is to isolate memory related code to one module. Perhaps you may want to use another library register file with different names and control signal that you can put inside this module.

```verilog
module ehgu_fifo_mem
# (
parameter WIDTH = 8 ,
parameter AWIDTH = 8 ,
parameter DEPTH = 128
 ) (
input logic wclk ,
input logic rclk ,
input logic renable ,
input logic wenable ,
input logic [ AWIDTH-1 : 0 ] raddr ,
input logic [ AWIDTH-1 : 0 ] waddr ,
input logic [ WIDTH-1 : 0 ] wdata ,
output logic [ WIDTH-1 : 0 ] rdata
 ) ;

ehgu_ram_dual_port # ( .DEPTH ( DEPTH ) , .WIDTH ( WIDTH ) ) dmem_i
 (
.wclk ,
.wenable ,
.waddr ,
.wdata ,
.rclk ,
.renable ,
.raddr ,
.rdata
 ) ;
```

endmodule

The fifo logic module does the calculation of write and read addresses. For the synchronous case the section inside the SYNC_TYPE 0 is not shown. The logic is simple, write into the fifo when data valid is asserted for the input data. After writing the memory in one address the next write happens in the next address. At the end the address wraps around to the address 0. The logic here keeps track of how many data are left in the memory to be read out by comparing the write address and the read address. If they are not the same it means there are some data to be read out. This condition of the difference being greater than 0 is taken as the cue to read out. Like write, the read happens by reading one address after another by incrementing the address with every read.

```systemverilog
module ehgu_fifo_logic
# (
parameter SYNC_TYPE = 0,
parameter SYNC_STG_W2R = 2,
parameter SYNC_STG_R2W = 2,
parameter WIDTH = 8 ,
parameter AWIDTH = 8 ,
parameter DEPTH = 128
 ) (
input logic wclk ,
input logic rclk ,
input logic wrstn ,
input logic rrstn ,
input logic en,
input logic din_valid ,

output logic wenable ,
output logic [ AWIDTH-1 : 0 ] waddr ,
output logic renable ,
output logic [ AWIDTH-1 : 0 ] raddr ,
output logic dout_valid
 ) ;

import ehgu_basic_pkg :: sub_modulo_unsigned;
import ehgu_basic_pkg :: bin2gray ;
import ehgu_basic_pkg :: gray2bin ;

logic [ AWIDTH-1 : 0 ] raddr_next ;
logic [ AWIDTH-1 : 0 ] waddr_next ;
logic renable_next ;
logic nc;
logic [AWIDTH-1:0] diff ;

always_comb begin
  wenable = din_valid ;
end

always_comb begin
  if ( wenable ) begin
    waddr_next = ( waddr + 1 ) % DEPTH ;
  end else begin
    waddr_next = waddr ;
```

```systemverilog
      end
    end

    always_ff @ ( posedge wclk , negedge wrstn ) begin
      if ( !wrstn ) begin
        waddr <= 0 ;
      end else begin
        waddr <= waddr_next ;
      end
    end

    always_comb begin
      if ( renable ) begin
        raddr_next = ( raddr + 1 ) % DEPTH ;
      end else begin
        raddr_next = raddr ;
      end
    end

    always_ff @ ( posedge rclk , negedge rrstn ) begin
      if ( !rrstn ) begin
        raddr <= 0;
      end else begin
        raddr <= raddr_next ;
      end
    end

    always_ff @(posedge rclk or negedge rrstn) begin
      if(~rrstn) begin
        dout_valid <= 0 ;
      end else begin
        dout_valid <= renable ;
      end
    end

    logic [ AWIDTH-1 : 0 ] raddr_for_compare ;
    logic [ AWIDTH-1 : 0 ] waddr_for_compare ;

    generate
      if ( SYNC_TYPE == 0 ) begin
        : async_fifo
        // async fifo not shown in this example
      end else begin
        : sync_fifo
        assign raddr_for_compare = raddr ;
        assign waddr_for_compare = waddr ;
      end
    endgenerate

      always_comb begin
        sub_modulo_unsigned ( .inp0 (waddr_for_compare) , .inp1 (raddr), .modulo(DEPTH),
.wrapped(nc), .diff(diff));
        if ( diff > 0 ) begin
          renable = 1;
        end else begin
```

```
      renable = 0;
    end
  end
endmodule
```

The testbench supplies data in bursts. There is a long period when data is valid and a short gap when valid is low. This is typical of most data transmissions through a FIFO. The gaps play an important role in providing the read side of the FIFO to read out the stored data and equalize the average output rate with the input data rate. Because it is not very convenient to store all the input data and check for exact order on the output side of the FIFO in a testbench, this TB uses just a pattern in the data for checking, that is consecutive data differ by the value 3. Note that this can miss come bugs.

```systemverilog
module tb ;

timeunit 1ns ;
timeprecision 1ps ;

import thee_utils_pkg :: * ;

parameter DEPTH = 32 ;
parameter AWIDTH = $clog2 ( DEPTH ) ;
parameter DWIDTH = 8 ;
parameter VEC_CNT = 2000;

logic clk , rstn ;
logic [ DWIDTH-1 : 0 ] din ;
logic [ DWIDTH-1 : 0 ] dout , dout_d, expected_data ;
logic en ;
logic din_valid, dout_valid ;

logic result ;

logic [ DWIDTH-1 : 0 ] mem_mirror [ DEPTH ] ;

parameter real FREQ = 100 ;
thee_clk_gen_module # ( .FREQ ( FREQ ) ) clk_gen_i0 ( .clk ( clk ) ) ;

int cnt ;
initial begin
  din_valid = 0;
  cnt = 10 ;
  forever @(posedge clk) begin
    if (cnt == 0) begin
      cnt <= din_valid ? $urandom_range(20,30): $urandom_range(100,120);
      din_valid <= ~din_valid;
    end else begin
      cnt <= cnt -1 ;
    end
  end
end

initial begin
  din = 0 ;
  forever @(posedge clk) begin
```

```verilog
      if (din_valid ) begin
        din += 3;
      end
    end
  end

  initial begin
    dout_d = 0 ;
    forever @(posedge clk) begin
      if (dout_valid) begin
        dout_d <= dout ;
      end
    end
  end

  assign expected_data = dout_d + 3;

  initial begin
    result = 'x ;
    en = 1 ;
    toggle_rstn ( .rstn ( rstn ) ) ;
    repeat (10) @(posedge clk) ;
    for ( int i = 0 ; i < VEC_CNT ; ) begin
      repeat ( 1 ) @ ( posedge clk ) ;
      if ( dout_valid ) begin
        if ( dout === expected_data && !$isunknown(dout)) begin
          $display ( "P - output data %h expected data %h" , dout , expected_data ) ;
          update_test_status (.result(result), .this_result(1));
        end else begin
          $display ( "F - output data %h expected data %h" , dout , expected_data ) ;
          update_test_status (.result(result), .this_result(0));
        end
        i ++ ;
      end
    end
    repeat (10) @(posedge clk) ;

    print_test_result ( result ) ;
    $finish ;
  end

  ehgu_fifo # ( .DEPTH ( DEPTH ) , .WIDTH ( DWIDTH ), .SYNC_TYPE(1) ) fifo (
  .clk0 (clk) ,
  .clk1 (clk) ,
  .wrstn (rstn),
  .rrstn (rstn),
  .en ,
  .din_valid ,
  .din ,
  .dout ,
  .dout_valid
  ) ;

endmodule
```

```
run -all
P - output data 03 expected data 03
P - output data 06 expected data 06
…
P - output data 6d expected data 6d
P - output data 70 expected data 70
Test Pass
```

Exercise

A synchronous FIFO without any additional features does not do much except output the same input data with a fixed clock cycle delay.

Can you extend the FIFO functionality with a LATENCY parameter that forces the FIFO to fill up the memory up to LATENCY words before read out? This way even if the inputs come in small bursts the output will be continuous. You may call this a burst filter.

Can you extend the FIFO to act as a buffer by adding a request acknowledge handshake on the output side? The application is that a sender sends data to a receiver at anytime. The buffer you designed waits for the receiver to be available by requesting and then transmits the data to the receiver after the receiver acknowledges the request. If the receiver does not acknowledge then the buffer accumulates the data from the sender until the receiver's ack is received.

Asynchronous

If the input and output side clocks need to be different it opens up a big can of worms. The logic is only slightly different than a synchronous FIFO but there a number of points to keep in mind for the asynchronous FIFO to work correctly. Only the fifo logic module is different.

Differences

The computation of difference between read and write addresses needs to be done carefully. The write address is transferred to the read side using Gray coding.

Write address on write clock domain → Gray Encoding → registering to avoid combo glitch → CDC using synchronizer → Gray Decoding → write address on read side

The synchronizer needs variable number of stages based on the clock frequency. A faster clock generally needs more stages to provide the same low chance of metastability. So the stage parameter is brought out to the fifo top level that can be set during instantiation.

Reset is separated into write and read side resets. There is an option of adding an internal reset from either the wrstn or rrstn and then reset synchronizing them to the other clock domain, but FIFOs are normally part of larger designs that are likely to have these resets already generated. So, using wrstn and rrstn as primary inputs.

```
// Free - this code is copywrite free , Do Whatever You Want DWYW
// 3vm , 2020 Oct
// First In First Out FIFO logic other than memory
module ehgu_fifo_logic
```

```systemverilog
# (
parameter SYNC_TYPE = 0,
parameter SYNC_STG_W2R = 2,
parameter SYNC_STG_R2W = 2,
parameter WIDTH = 8 ,
parameter AWIDTH = 8 ,
parameter DEPTH = 128
 ) (
input logic wclk ,
input logic rclk ,
input logic wrstn ,
input logic rrstn ,
input logic en,
input logic din_valid ,

output logic wenable ,
output logic [ AWIDTH-1 : 0 ] waddr ,
output logic renable ,
output logic [ AWIDTH-1 : 0 ] raddr ,
output logic dout_valid
 ) ;

import ehgu_basic_pkg :: sub_modulo_unsigned;
import ehgu_basic_pkg :: bin2gray ;
import ehgu_basic_pkg :: gray2bin ;

logic [ AWIDTH-1 : 0 ] raddr_next ;
logic [ AWIDTH-1 : 0 ] waddr_next ;
logic renable_next ;
logic nc;
logic [AWIDTH-1:0] diff ;

always_comb begin
  wenable = din_valid ;
end

always_comb begin
  if ( wenable ) begin
    waddr_next = ( waddr + 1 ) % DEPTH ;
  end else begin
    waddr_next = waddr ;
  end
end

always_ff @ ( posedge wclk , negedge wrstn ) begin
  if ( !wrstn ) begin
    waddr <= 0 ;
  end else begin
    waddr <= waddr_next ;
  end
end

always_comb begin
  if ( renable ) begin
    raddr_next = ( raddr + 1 ) % DEPTH ;
```

```systemverilog
    end else begin
     raddr_next = raddr ;
    end
   end

   always_ff @ ( posedge rclk , negedge rrstn ) begin
     if ( !rrstn ) begin
      raddr <= 0;
     end else begin
      raddr <= raddr_next ;
     end
   end

   always_ff @(posedge rclk or negedge rrstn) begin
     if(~rrstn) begin
     dout_valid <= 0 ;
     end else begin
     dout_valid <= renable ;
     end
   end

   logic [ AWIDTH-1 : 0 ] raddr_for_compare ;
   logic [ AWIDTH-1 : 0 ] waddr_for_compare ;

   generate
    if ( SYNC_TYPE == 0 ) begin
     : async_fifo
     import ehgu_basic_pkg::bin2gray;
     import ehgu_basic_pkg :: shortgray_constants_t ;
     import ehgu_basic_pkg :: get_shortgray_constants ;
     import ehgu_basic_pkg :: get_shortgray_skip ;
     import ehgu_basic_pkg :: get_shortgray_unskip ;

     logic [ AWIDTH-1 : 0 ] raddr_gray, raddr_gray_registered, raddr_gray_post_cdc, raddr_post_cdc
 ;
     logic [ AWIDTH-1 : 0 ] waddr_gray, waddr_gray_registered, waddr_gray_post_cdc,
 waddr_post_cdc ;
     logic [ AWIDTH-1 : 0 ] waddr_bin;

     if ( $onehot(DEPTH) == 0 ) begin
      : shortgray
      localparam shortgray_constants_t sg_constants = get_shortgray_constants ( .code_length
 ( DEPTH ) ) ;
       assign waddr_bin = get_shortgray_skip ( waddr , sg_constants ) ;
       assign waddr_for_compare = get_shortgray_unskip ( waddr_post_cdc , sg_constants ) ;
     end else begin
      : fullgray
      assign waddr_bin = waddr ;
      assign waddr_for_compare = waddr_post_cdc ;
     end

     always_comb
      bin2gray (.binary_in(waddr_bin),.gray_out(waddr_gray));

     always_ff @ ( posedge wclk , negedge wrstn ) begin
```

```verilog
      if ( !wrstn ) begin
        waddr_gray_registered <= 0 ;
      end else begin
        waddr_gray_registered <= waddr_gray ;
      end
    end

    ehgu_synqzx #(.T(time), .MAX_DELAY(100ps), .STAGES(SYNC_STG_W2R), .WIDTH(AWIDTH))
sync_waddr
      (
        .clk (rclk) ,
        .rstn (rrstn) ,
        .d_presync(waddr_gray_registered) ,
        .d_sync ( waddr_gray_post_cdc )
      );

    always_comb begin
      gray2bin (.gray_in(waddr_gray_post_cdc),.binary_out(waddr_post_cdc));
    end

    always_comb
      bin2gray (.binary_in(raddr),.gray_out(raddr_gray));

    always_ff @ ( posedge rclk , negedge rrstn ) begin
      if ( !rrstn ) begin
        raddr_gray_registered <= 0 ;
      end else begin
        raddr_gray_registered <= raddr_gray ;
      end
    end

    ehgu_synqzx #(.T(time), .MAX_DELAY(100ps), .STAGES(SYNC_STG_R2W), .WIDTH(AWIDTH))
sync_raddr
      (
        .clk (rclk) ,
        .rstn (wrstn),
        .d_presync(raddr_gray_registered ) ,
        .d_sync ( raddr_gray_post_cdc )
      );

    assign raddr_for_compare = raddr_post_cdc ;
  end else begin
    : sync_fifo
    assign raddr_for_compare = raddr ;
    assign waddr_for_compare = waddr ;
  end
endgenerate

always_comb begin
  sub_modulo_unsigned ( .inp0 (waddr_for_compare) , .inp1 (raddr), .modulo(DEPTH),
.wrapped(nc), .diff(diff));
  if ( diff > 0 ) begin
    renable = 1;
  end else begin
    renable = 0;
```

```
    end
  end

endmodule
```

Asynchronous With Short Gray Code

If the memory size is not a power of 2 you cannot use a regular full length Gray code. Rather a short version of the Gray code is applicable. For this purpose the address transfer step has two additional computations. The binary to gray step of the regular Gray code is modified to include a binary to skipped binary representation. There is also a matching reverse operation of skipped binary to binary step. When the memory size is not a power of 2, the size number will not be one hot in its binary representation which another way of saying there will be more than 1 ones in the binary format. This property is used to insert the binary skip/unskip operations into address transfer logic.

Example: 18 binary is 10010 (2 ones), 16 binary is 10000 (only 1 one)

*Write address on write clock domain → **Skip binary** → Gray Encoding → registering to avoid combo glitch → CDC using synchronizer → Gray Decoding → **Unskip binary** → write address on read side*

The testbench is similar to the synchronous case, except that the clocks are different. The resets should ideally be synchronized to each domain but I have lazy coded the same reset for now knowing that it is only a testbench. If you are using the FIFO in you chip remember to user per clock domain synchronized resets as inputs.

```systemverilog
module tb ;

timeunit 1ns ;
timeprecision 1ps ;

import thee_utils_pkg :: * ;

parameter DEPTH = 64 ;
parameter AWIDTH = $clog2 ( DEPTH ) ;
parameter DWIDTH = 8 ;
parameter VEC_CNT = 2000 ;

logic clk , clk0 , clk1 , rstn ;
logic [ DWIDTH-1 : 0 ] din ;
logic [ DWIDTH-1 : 0 ] dout , dout_d , expected_data ;
logic en ;
logic din_valid , dout_valid ;

logic result ;

logic [ DWIDTH-1 : 0 ] mem_mirror [ DEPTH ] ;

parameter real FREQ = 100 ;
thee_clk_gen_module # ( .FREQ ( FREQ * 1.1 ) ) clk_gen_i0 ( .clk ( clk0 ) ) ;
thee_clk_gen_module # ( .FREQ ( FREQ ) ) clk_gen_i1 ( .clk ( clk1 ) ) ;
assign clk = clk0 ;
```

```verilog
int cnt ;
initial begin
  din_valid = 0 ;
  cnt = 10 ;
  forever @ ( posedge clk ) begin
    if ( cnt == 0 ) begin
      cnt <= din_valid ? $urandom_range ( 20 , 30 ) : $urandom_range ( 100 , 120 ) ;
      din_valid <= ~din_valid ;
    end else begin
      cnt <= cnt -1 ;
    end
  end
end

initial begin
  din = 0 ;
  forever @ ( posedge clk ) begin
    if ( din_valid ) begin
      din <= din + 3 ;
    end
  end
end

initial begin
  dout_d = -3 ;
  forever @ ( posedge clk1 ) begin
    if ( dout_valid ) begin
      dout_d <= dout ;
    end
  end
end

assign expected_data = dout_d + 3 ;

initial begin
  result = 'x ;
  en = 1 ;
  toggle_rstn ( .rstn ( rstn ) ) ;
  repeat ( 10 ) @ ( posedge clk ) ;
  for ( int i = 0 ; i < VEC_CNT ; ) begin
    repeat ( 1 ) @ ( posedge clk ) ;
    if ( dout_valid ) begin
      if ( dout === expected_data && !$isunknown ( dout ) ) begin
        $display ( "P - output data %h expected data %h" , dout , expected_data ) ;
        update_test_status ( .result ( result ) , .this_result ( 1 ) ) ;
      end else begin
        $display ( "F - output data %h expected data %h" , dout , expected_data ) ;
        update_test_status ( .result ( result ) , .this_result ( 0 ) ) ;
      end
      i ++ ;
    end
  end
  repeat ( 10 ) @ ( posedge clk ) ;
  print_test_result ( result ) ;
  $finish ;
```

end

```verilog
ehgu_fifo # ( .DEPTH ( DEPTH ) , .WIDTH ( DWIDTH ) , .SYNC_TYPE ( 0 ) ) fifo (
.clk0 ( clk0 ) ,
.clk1 ( clk1 ) ,
.wrstn ( rstn ) ,
.rrstn ( rstn ) ,
.en ,
.din_valid ,
.din ,
.dout ,
.dout_valid
 ) ;
```

endmodule

```
run -all
P - output data 00 expected data 00
P - output data 03 expected data 03
…
P - output data 4b expected data 4b
P - output data 4e expected data 4e
Test Pass
```

Write/Read Address Behavior

It is useful to visualize the operation of the fifo address calculation in detail. Lets see the prints from a run with uncommented debug code of fifo module.

Notation: wa – write address, we – write enable, wd – write data, do – data output, dv – data out valid

The input data writing starts are address 0. The read address and read enable do not get activated for a few cycles because the write address takes a few clock cycles to reach the read side. In this testbench the write clock is set to be 110 MHz and read clock to 100 MHz. For every 110 writes the output side can do only 100 reads. Input data accumulates in the FIFO at a rate that is the difference in the clock frequencies of input and ouput. As the write burst goes on, the write address and read address have wrapped around the end of the memory and start writing from address 0. At the end, when the write address reached 41, the last input data is written and the burst stops. The write address parks at 42 waiting for the next data burst input. The read address only managed to reach 30, leaving a lag of 41 – 30 = 11 data words left in the memory. The read continues till those 11 words are read out and then stops. You can also notice that the read data from address say A comes only on the next clock cycle after the address A was presented to the memory.

```
wa   0, we 1, wd 00, ra   0, re 0, rd xx
do xx, dv 0
wa   1, we 1, wd 03, ra   0, re 0, rd xx
do xx, dv 0
wa   2, we 1, wd 06, ra   0, re 0, rd xx
do xx, dv 0
wa   3, we 1, wd 09, ra   0, re 1, rd xx
do xx, dv 0
wa   4, we 1, wd 0c, ra   1, re 1, rd 00
do 00, dv 1
wa   5, we 1, wd 0f, ra   2, re 1, rd 03
```

```
…
wa 40, we 1, wd 38, ra 28, re 1, rd 11
do 11, dv 1
wa 41, we 1, wd 3b, ra 29, re 1, rd 14
do 14, dv 1
wa 42, we 0, wd 3e, ra 30, re 1, rd 17
do 17, dv 1
…
wa 42, we 0, wd 3e, ra 40, re 1, rd 35
do 35, dv 1
wa 42, we 0, wd 3e, ra 41, re 1, rd 38
do 38, dv 1
wa 42, we 0, wd 3e, ra 42, re 0, rd 3b
```

Formula for FIFO depth calculation

When using a FIFO you need to use a memory depth that is adequate to cover all the data burst and clock frequency characteristics of your application.

First order equations

data lost in bits = 0
=> total input data bits in input burst = total data bits in output for input burst duration + accumulated bits
=> accumulated bits or memory depth needed = input bits – output bits

Suppose the input burst is 100 words at 110 MHz and output works at 100 MHz of same word size,
Bits received in one input burst = 100 words
Duration of input burst = 100 * (1/110M) = 0.909us
Bits read out in the duration of input burst = 0.909us / (1/100M) = 90.9 words
Depth needed = 100 – 90.9 = 9.1

Second order equation
The depth of 10 words is only a first guess. There are many other non idealities that increase the required depth. For example, there is a latency of 1 cycle in write clock for registering the Gray code and 2 read clock cycles for synchronizing. Add to this any other latency you may have for timing closure. Here register stages added for timing closure is 0. In one read cycle, the write side will write 1.1 word because the write side is 10% faster.

depth = first order depth + address transfer latency
     = 9.1 + 1 + 2 * (1.1) //read cycle duration referred to write side
     = 12.3

Third order equation
The clocks are not perfect. Typical tolerance is clock frequency is about 1000ppm. That means the input side could be 110M + 1000 ppm and the output side could be 100 M – 1000 ppm. So multiply the first order depth by the ppm difference or use the ppm updated clocks for the accumulated bits formula.

Clocks after ppm change,

Input frequency = 110M * (1 + 1000/1000000) = 110.11MHz
Output frequency = 100M * (1 – 1000/1000000) = 99.9 MHz

Depth = 100 - {100 * (1 / 110.11M) } / { 1 / 99.9M }
       = 100 – 90.721
       = 9.27
After adding latency effects,
Depth = 9.27 + 1 + 2.2 = 12.47

Because, there may be more unknowns I typically add additional 10% depth.

Depth = ceiling (1.1 * third order depth) = ceil (1.1 * 12.47) = ceil (13.7) = 14

Timing Constraints

There are paths in an asynchronous FIFO that cross clock domains. For example, the write address that is gray encoded and registered crosses from the write clock domain to the read clock domain. This path should not be timed in an STA tool like a single cycle path. It may be false pathed like this -

Recomended

```
set_clock_groups -asynchronous -group [get_clocks "wclk rclk"] ;
```

assuming you have already defined clock constraints with names wclk and rclk

Deprecated

```
set_false_path -from rclk -to wclk
```

```
set_false_path -from wclk -to rclk
```

On closer examination of timing a few things become apparent.

Bus skew
If the buffer insertion and routing of the register gray code bits going into the read domain synchronizer is bad and the variation in the arrival time between the bus bits is high then the Gray coding technique may fail. For example, suppose the waddr_gray_registered bit index 0 arrives at the read side with a delay of 1ns and the bit index 3 arrives with a delay of 12 ns. It is possible for the Gray code received between two transitions to differ by more than 1 bit. The general condition to adhere to is that the skew in arrival times of the address at the synchronizer input should be less than one period of the receiving clock domain. One way this may be checked is with a script that lists the arrival times and calculates the skew and checks if the skew is less than clock period. Other more brute force way is to limit the maximum delay between the registering flop outputs and the synchronizer inputs to less than one period, so that, skew will be less than period because the even the total delay is less than one period.

Please check the need for a constraint like the constraint below. Note that it may report correctly in the STA tool. You may have to check.
```
set_max_delay <1 receive clock period> -from <gray reg flop> -to
<synchronizer inputs>
```

Bus routing delay

Skew is only one issue. Take this hypothetical case where the skew is very low but all the bits of the gray reg lines are delayed by 10 receive clock cycles. Note that in our minimum depth calculation we did not add routing delay. If only the minimum depth was used the routing delay would increase the latency for the read side to respond and make the input overflow. A multicycle/max delay constraint may be appropriate to limit the routing delay. Note that the max delay and multicycle path constraints are not used in real chipping because the routing tool most often places the registering flops and the synchronizer flops right next to each other with low delay. But these Nth order effects become important when the receiving clock is operating so fast that one receive clock period is comparable to bus skew/delay.

```
set_max_delay <N receive clock period> -from <gray reg flop> -to
<synchronizer inputs>
or
set_multicycle_path N -setup  -from <gray reg flop> -to <synchronizer inputs>
set_multicycle_path N-1 -hold  -from <gray reg flop> -to <synchronizer
inputs>  ; #idiosyncracy
```

Maximum Increment is 1

Gray code guarantees a Hamming distance of 1 only for binary numbers separated by 1. If the numbers are separated by more than 1, Gray coded distance can be more than one.
Example: Binray 000,001,010 maps to => Gray 000, 001, 011
If you look at the binay pairs 0,1 and 1,2 the Gray codes are 000,001 and 001,011 which have only 1 distance
But if you take the binary pair 000, 010 the Gray code pair is 000,011 which has a distance of 2.

The point is that when using Gray codes the writing side should increment the write address by only one. If not, the synchronized output on the read side will not be a faithful value of the write address. Note that there may be special techniques to overcome this limitation that I am not aware of. Please do your research.

Exercise

Overflow logic

For locating the point of failure in testbench and actual system, it is preferred to report the overflow condition. One method of detecting the overflow is to compare the write to read difference between two cycles. If the difference is very high in one cycle and then drops to near zero in the next an overflow has occurred. The overflow happens in a transient way, so, use a module called ehgu_latch_flag that saves this transient for reporting on the output. Also add a clear input to reset the saved flag. Use a similar logic to detect underflow.

Margin to overflow

Add a module called ehgu_peak_detect that tracks the maximum difference in the addresses. The formula is -

max diff = diff > max diff ? Diff : max diff

Add clear input to reset the maximum difference.

Enable input for clock gating

The enable input to the logic is not properly connected. Use the enable signal to let the logic work or hold state. Synthesis tools can infer clock gates based on this enable.

Request Grant/Ack

FIFOs are typically used with request-grant/ack handshake on the input and/or output sides. Add this logic on both the input and output sides. Adding the ack as an output of the write side allows a FIFO to apply back pressure to the data input logic to slow down when the FIFO is about to overflow.

Native Parallel Bus

Actually, there is no standard logic block named native parallel bus. I have dedicated a section to this topic to focus on the communication interface rather than the source or sink of the communication. I am calling this native because this is the bus native to the chip land. Every other bus protocol SPI, I2C, USB, PCIe, AXI, TileLink ultimately becomes a parallel bus inside the chip, albeit, operating only over tiny distances and each with their own variations.

Lets look at the bus communication using a small example with one host and two endpoints. The host is also called master or source or driver and the endpoint is also called device, slave or sink. The code shown for the host and endpoint are only models, they are not to be used for synthesis. It is common to model the bus using a bus functional model (BFM). The endpoint module is used as if it is a single port memory. But, it could be other devices also for example you could have a RISC-V processor core as the bus master and a SAR ADC as a slave. The digital converted data could be read from a specific address using the bus. Note the usage of a base address and a range. In the context of a processor, many devices and memories are placed at various address ranges of the bus address. For example the data going to the monitor could be part of the address range 0xFFFF_0000 to 0xFFFF_1000. A System real time clock data could be available in the range, 0xFFFF_5000 to 0xFFFF_50FF. A table of which address range contains what type of data is called a register map. The logic inside the device checks if the access is to any of the addresses it is responsible for and then takes appropriate action. Here, it is just accepts data to store or present data on the bus that is already stored in the memory.

```
module bus_endpoint
# (
parameter ADDR_WIDTH = 8 ,
parameter DATA_WIDTH = 8 ,
parameter BASE_ADDR = 0 ,
parameter RANGE = 16
 ) (
input logic r_wn ,
input logic [ ADDR_WIDTH-1 : 0 ] addr ,
input logic [ DATA_WIDTH-1 : 0 ] wdata ,
output logic [ DATA_WIDTH-1 : 0 ] rdata
 ) ;
```

```
localparam MEM_ADDR_WIDTH = $clog2 ( RANGE ) ;

logic [ DATA_WIDTH-1 : 0 ] mem [ RANGE ] ;
logic [ MEM_ADDR_WIDTH-1 : 0 ] maddr ;
logic access_here ;

assign access_here = ( addr inside { [ BASE_ADDR : BASE_ADDR + RANGE-1 ] } ) ;
assign maddr = addr - BASE_ADDR ;

always @ ( negedge r_wn )
 if ( access_here ) begin
   mem [ maddr ] = wdata ;
   $display ( "Write to location %d local address %d data %d in instance %m" , addr , maddr ,
wdata ) ;
 end

always @ ( posedge r_wn )
 if ( r_wn == 1 && access_here ) begin
   rdata = mem [ maddr ] ;
   $display ( "Read from location %d data %d in instance %m" , addr , rdata ) ;
 end else
 rdata = 0 ;

endmodule
```

The intelligence of how to use the bus is typically in the master or host. Write and read transactions are done by the master to the addresses available to the host. Note that no separate clock signal is needed, just the read or write signal can be used as a latch enable signal or a clock signal. Note the tasks for read and write. These operations are also called transactions.

```
module bus_host
# (
parameter ADDR_WIDTH = 8 ,
parameter DATA_WIDTH = 8 ,
parameter int VALID_RANGES [ 2 ] [ 2 ]
 ) (
output logic r_wn ,
output logic [ ADDR_WIDTH-1 : 0 ] addr ,
output logic [ DATA_WIDTH-1 : 0 ] wdata ,
input logic [ DATA_WIDTH-1 : 0 ] rdata ,
output logic busy
 ) ;

logic [ ADDR_WIDTH-1 : 0 ] tmp ;
bit result ;

initial begin
  busy = 1 ;
  result = 1 ;

  for ( int i = 0 ; i < VALID_RANGES [ 1 ] [ 1 ] ; i ++ ) begin
   if ( ( i >= VALID_RANGES [ 0 ] [ 0 ] && i < VALID_RANGES [ 0 ] [ 1 ] ) ||
   ( i >= VALID_RANGES [ 1 ] [ 0 ] && i < VALID_RANGES [ 1 ] [ 1 ] ) ) begin
     bus_write ( .a ( i ) , .w ( i + 1 ) ) ;
```

```verilog
      bus_read ( .a ( i ) , .r ( tmp ) ) ;
      if ( tmp !== wdata ) begin
        result = 0 ;
        $display ( "Invalid read data" ) ;
      end
    end
  end
  busy = 0 ;
end

task bus_write (
input logic [ ADDR_WIDTH-1 : 0 ] a ,
input logic [ ADDR_WIDTH-1 : 0 ] w
 ) ;
$display ( "Writing %d to address %d" , w , a ) ;
addr = a ;
wdata = w ;
#0 ;
r_wn = 0 ;
#0 ;
r_wn = 1 ;
#0 ;
endtask

task bus_read (
input logic [ ADDR_WIDTH-1 : 0 ] a ,
output logic [ ADDR_WIDTH-1 : 0 ] r
 ) ;

addr = a ;
#0 ;
r_wn = 1 ;
#0 ;
r = rdata ;
$display ( "Read %d from address %d" , r , a ) ;
endtask

endmodule
```

The testbench shows another important aspect of this simple bus. The read data from the two devices are OR-ed together to produce the common read data for the bus. Whenever the device gets an out of self access, it sets the rdata to zero, so that, only the valid addressed device gets the chance to output its data without interference. Also note the placement of the devices happens at this level using the base address parameter. One device is placed at address 0 and the other is placed with some offset. The %m in the display statement means the hierarchical instance name of the module that is executing the specific code. Here, it is used to show which device is responding to the host's requests ep0 or ep1.

```verilog
module tb ;

parameter ADDR_WIDTH = 5 ;
parameter DATA_WIDTH = 4 ;

parameter BASE0 = 0 , BASE1 = 16 ;
```

```systemverilog
parameter R0 = 4 , R1 = 8 ;
logic busy ;
logic r_wn ;
logic [ ADDR_WIDTH-1 : 0 ] addr ;
logic [ DATA_WIDTH-1 : 0 ] rdata0 , rdata1 , rdata , wdata ;
bit result ;

initial begin
  @ ( negedge busy ) ;
  if ( bus_host.result )
  $display ( "All Vectors passed" ) ;
  else
  $display ( "Some Vectors failed" ) ;

  $finish ;
end

bus_host # (
.DATA_WIDTH ( DATA_WIDTH ) ,
.ADDR_WIDTH ( ADDR_WIDTH ) ,
.VALID_RANGES ( '{ '{ BASE0 , BASE0 + R0 } , '{ BASE1 , BASE1 + R1 } } )
 ) host (
.r_wn ,
.addr ,
.wdata ,
.busy ,
.rdata
 ) ;

bus_endpoint
# (
.DATA_WIDTH ( DATA_WIDTH ) ,
.ADDR_WIDTH ( ADDR_WIDTH ) ,
.BASE_ADDR ( BASE0 ) ,
.RANGE ( R0 )
 ) ep0 (
.r_wn ,
.addr ,
.wdata ,
.rdata ( rdata0 )
 ) ;

bus_endpoint
# (
.DATA_WIDTH ( DATA_WIDTH ) ,
.ADDR_WIDTH ( ADDR_WIDTH ) ,
.BASE_ADDR ( BASE1 ) ,
.RANGE ( R1 )
 ) ep1 (
.r_wn ,
.addr ,
.wdata ,
.rdata ( rdata1 )
 ) ;
```

```
assign rdata = rdata0 | rdata1 ;

endmodule
```

```
run -all
Writing  1 to address  0
Write to location  0 local address 0 data  1 in instance tb.ep0
Read from location  0 data  1 in instance tb.ep0
Read  1 from address  0
Writing  2 to address  1
Write to location  1 local address 1 data  2 in instance tb.ep0
Read from location  1 data  2 in instance tb.ep0
Read  2 from address  1
Writing  3 to address  2
Write to location  2 local address 2 data  3 in instance tb.ep0
Read from location  2 data  3 in instance tb.ep0
Read  3 from address  2
Writing  4 to address  3
Write to location  3 local address 3 data  4 in instance tb.ep0
Read from location  3 data  4 in instance tb.ep0
Read  4 from address  3
Writing 17 to address 16
Write to location 16 local address 0 data  1 in instance tb.ep1
Read from location 16 data  1 in instance tb.ep1
Read  1 from address 16
Writing 18 to address 17
Write to location 17 local address 1 data  2 in instance tb.ep1
Read from location 17 data  2 in instance tb.ep1
Read  2 from address 17
Writing 19 to address 18
Write to location 18 local address 2 data  3 in instance tb.ep1
Read from location 18 data  3 in instance tb.ep1
Read  3 from address 18
Writing 20 to address 19
Write to location 19 local address 3 data  4 in instance tb.ep1
Read from location 19 data  4 in instance tb.ep1
Read  4 from address 19
Writing 21 to address 20
Write to location 20 local address 4 data  5 in instance tb.ep1
Read from location 20 data  5 in instance tb.ep1
Read  5 from address 20
Writing 22 to address 21
Write to location 21 local address 5 data  6 in instance tb.ep1
Read from location 21 data  6 in instance tb.ep1
Read  6 from address 21
Writing 23 to address 22
Write to location 22 local address 6 data  7 in instance tb.ep1
Read from location 22 data  7 in instance tb.ep1
Read  7 from address 22
Writing 24 to address 23
Write to location 23 local address 7 data  8 in instance tb.ep1
Read from location 23 data  8 in instance tb.ep1
Read  8 from address 23
All Vectors passed
```

Exercise

Modern buses like TileLink have many more features compared to the native parallel bus. Download the TileLink specification and list 5 features absent in the native bus.

https://sifive.cdn.prismic.io/sifive%2Fcab05224-2df1-4af8-adee-8d9cba3378cd_tilelink-spec-1.8.0.pdf

Buses or interconnects support many topologies. Study the topology of daisy chain. What other topologies are available?

Control Status Register

Control Status Register or Configuration Status Register or Control Register or Configuration Register or just Register is a powerful way to access the functionality of the chip. I have cooked up a story for you to show how useful it is to have a CSR.

Say your product is an inverter packaged in a Dual Inline Package, DIP. Say, your product is a big success. But, about 30% more customers don't want an inverter, they want a buffer. So you decide to add a primary input to control the behavior of the chip to inverter or buffer. With this you can serve the additional 30% customers. Later, customers ask for a bus of 16 inverter/buffer devices packaged in the same package. You provide them only one inverter or buffer selection pin for all 16 devices in the chip. Now customers ask for programmability of every inverter/buffer device individually. You provide 16 control input pins. Now there are 16 data inputs, 16 data outputs, 16 control inputs and 2 power-ground pins for a total of 50 pins. Customers complain that the 50 pin package is too big. They are not able to solder the new chip within the PCB size budget. You now introduce a native parallel bus to write or read a 1bit 16 location RAM made with flip flops. Voila! The control register is born. Now your chip has only 32 data I/O + 7 bits for native parallel bus for a total of 39 pins. Your customers are happy now. The new package fits their PCB. Still, some customers are unhappy. They want to detect activity on the lines that are toggling. Customers have defined activity as any toggle within a window of 100us. Your company engineers are hard at work and complete the circuit to detect toggling. But your customers have told you that they can take a maximum of only a 40 pin package. Your engineers initial suggestion of providing 16 new output pins will exceed the pin budget by 15 (39+16=55). You are in a fix. One smart lab technician in your company asks you if you can do something with the native parallel bus. You decide to provide the toggle detection on the 16 data lines through 16 data bits in the memory inside the chip and access the information through the native parallel bus with bidirectional I/O for data line. Voila! The status register is born. Together, CSR is born! So much, functionality using the same 39 pins.

Stories apart, the CSR is a chip construct extremely useful to customize the chip. Otherwise, a chip is literally set in stone and will be much less useful. Every control bit added to the design doubles the number of applications the chip can be used for. Also doubles the verification effort :(

There are many ways to visualize a CSR. It can be said to be a single port RAM with another port of parallel inputs and outputs. It can be seen as a hardware programming interface, if you like the software API idea.

The example here shows a simple CSR inline with the cooked up story. There are 16 inverter-buffer devices in the chip. Each can be enabled or disabled and configured as an inverter or a buffer through the CSR. Activity on every device is detected and made available in a read only register bit. Each device is logically allocated a fixed position in the CSR bits. Device 13 settings is always located in second byte, index 5 position. To configure device 4 as an inverter, you would enable that device 4 by writing to BUFEN register first byte to 00010000. Most software and hardware are organized as multiples of bytes. So the example uses 8 bits as the default width of one register element in the CSR.

The CSR is accessed using write and read tasks defined for the bus. The test puts some activity on the selected device input and then proceeds to read the activity detected bit to check if the activity detection logic worked. The chip model here is for demonstration only. It is not meant to be synthesized. Only the CSR logic is synthesizable.

```systemverilog
package chip_config_pkg ;
localparam NUMBUF = 16 ;

localparam NUMREG = 6 ;
localparam BUFEN_ADDR_0 = 0 ;
localparam BUFEN_ADDR_1 = 1 ;
localparam BUFTYPE_ADDR_0 = 2 ;
localparam BUFTYPE_ADDR_1 = 3 ;
localparam ACTDET_ADDR_0 = 4 ;
localparam ACTDET_ADDR_1 = 5 ;
localparam time ACTDET_PERIOD = 100ns ;
endpackage


module chip_top
import chip_config_pkg :: * ;
 (
input logic clk ,
input logic rstn ,
input logic [ 7 : 0 ] addr ,
input logic r_wn ,
input logic [ 7 : 0 ] wdata ,
output logic [ 7 : 0 ] rdata ,
input logic [ NUMBUF-1 : 0 ] datain ,
output logic [ NUMBUF-1 : 0 ] dataout
 ) ;

logic [ NUMBUF-1 : 0 ] buftype , actdet , bufen ;

generate
 for ( genvar i = 0 ; i < NUMBUF ; i ++ ) begin
  : genbuf
  prog_buf bufi (
  .datain ( datain [ i ] ) ,
  .bufen ( bufen [ i ] ) ,
  .buftype ( buftype [ i ] ) ,
  .dataout ( dataout [ i ] ) ,
  .actdet ( actdet [ i ] )
  ) ;
 end
endgenerate


csr registers (
.clk ,
.rstn ,
.addr ,
.r_wn ,
.wdata ,
```

```verilog
      .rdata ,
      .bufen ,
      .buftype ,
      .actdet
      ) ;

endmodule
// 3vm

module csr
import chip_config_pkg :: * ;
 (
input logic clk ,
input logic rstn ,
input logic [ 7 : 0 ] addr ,
input logic r_wn ,
input logic [ 7 : 0 ] wdata ,
output logic [ 7 : 0 ] rdata ,
output logic [ NUMBUF-1 : 0 ] bufen ,
output logic [ NUMBUF-1 : 0 ] buftype ,
input logic [ NUMBUF-1 : 0 ] actdet
 ) ;

logic [ 7 : 0 ] csr_reg [ NUMREG ] ;
logic [ NUMBUF-1 : 0 ] actdet_synced ;

always_ff @ ( posedge clk , negedge rstn )
 if ( !rstn )
 csr_reg [ 0 : 3 ] <= '{ default : 0 } ;
 else if ( !r_wn )
 if ( addr inside { BUFEN_ADDR_0 , BUFEN_ADDR_1 , BUFTYPE_ADDR_0 , BUFTYPE_ADDR_1 } )
 csr_reg [ addr ] <= wdata ;

ehgu_synqzx # ( .MAX_DELAY ( 0 ) , .WIDTH ( NUMBUF ) ) dut
 (
.clk ,
.rstn ,
.d_presync ( actdet ) ,
.d_sync ( actdet_synced )
 ) ;

assign csr_reg [ ACTDET_ADDR_0 ] = actdet_synced [ 7 : 0 ] ;
assign csr_reg [ ACTDET_ADDR_1 ] = actdet_synced [ 15 : 8 ] ;

always_comb
 if ( r_wn )
 rdata = csr_reg [ addr ] ;
 else
 rdata = 0 ;

assign bufen [ 7 : 0 ] = csr_reg [ BUFEN_ADDR_0 ] ;
assign bufen [ 15 : 8 ] = csr_reg [ BUFEN_ADDR_1 ] ;
assign buftype [ 7 : 0 ] = csr_reg [ BUFTYPE_ADDR_0 ] ;
assign buftype [ 15 : 8 ] = csr_reg [ BUFTYPE_ADDR_1 ] ;
```

```systemverilog
endmodule


module prog_buf
import chip_config_pkg :: * ;
 (
input logic datain ,
input logic buftype ,
input logic bufen ,
output logic dataout ,
output logic actdet
 ) ;

assign dataout = bufen ? ( buftype ? datain : ~datain ) : 1'bz ;

initial begin
  actdet = 0 ;
  forever begin
    fork
    begin
     # ( ACTDET_PERIOD ) ;
     actdet = 0 ;
    end
    begin
     @ ( datain ) ;
     actdet = 1 ;
    end
    join_any
    disable fork ;
  end
end

endmodule


module tb ;

import chip_config_pkg :: * ;

logic r_wn ;
logic [ 8-1 : 0 ] addr ;
bit result ;
int device ;
logic [ NUMBUF-1 : 0 ] datain , dataout , actdet , bufen ;
logic [ 8-1 : 0 ] rdata , wdata , rd ;
logic clk , rstn ;

thee_clk_gen_module # ( .FREQ ( 10 ) ) clk_gen ( .clk ( clk ) ) ;

task my_print ;
 $display ( "Data input %b , data output %b , device index %d , activity detect %b at %t " , datain ,
dataout , device , actdet , $time ( ) ) ;
endtask

initial begin
```

```systemverilog
thee_utils_pkg :: toggle_rstn ( .rstn ( rstn ) , .rst_low ( 300ns ) ) ;

device = $urandom_range ( 0 , NUMBUF-1 ) ;
bufen = 2 ** device ;
csr_write ( BUFEN_ADDR_0 , bufen [ 7 : 0 ] ) ;
csr_write ( BUFEN_ADDR_1 , bufen [ 15 : 8 ] ) ;
my_print ;

datain = 'z ;
datain [ device ] = 0 ;

fork
forever begin
  datain [ device ] = ~datain [ device ] ;
  #1ns ;
end
join_none

@ ( datain [ device ] ) ;
repeat ( 10 ) @ ( posedge clk ) ;
csr_read ( ACTDET_ADDR_0 , rd ) ;
actdet = rd ;
csr_read ( ACTDET_ADDR_1 , rd ) ;
actdet = ( rd << 8 ) | actdet ;

my_print ;

if ( actdet [ device ] === 1 )
$display ( "All Vectors passed" ) ;
else
$display ( "Some Vectors failed" ) ;

$finish ;
end

task csr_write (
input logic [ 7 : 0 ] a ,
input logic [ 7 : 0 ] d
) ;
$display ( "Writing address %d with value %b" , a , d ) ;
@ ( posedge clk ) ;
addr = a ;
wdata = d ;
r_wn = 0 ;
@ ( posedge clk ) ;
endtask

task csr_read (
input logic [ 7 : 0 ] a ,
output logic [ 7 : 0 ] d
) ;
@ ( posedge clk ) ;
addr = a ;
r_wn = 1 ;
@ ( posedge clk ) ;
```

```systemverilog
    d = rdata ;
    $display ( "Read address %d got value %b" , a , d ) ;
endtask


chip_top dut (
.clk ,
.rstn ,
.addr ,
.r_wn ,
.wdata ,
.rdata ,
.datain ( datain ) ,
.dataout ( dataout )
 ) ;

endmodule
```

```
run -all
Writing address   0 with value 00010000
Writing address   1 with value 00000000
Data input xxxxxxxxxxxxxxxx, data output zzzzzzzzzzzxzzzz, device index        4, activity detect xxxxxxxxxxxxxxx at
700000
Read address   4 got value 00010000
Read address   5 got value 00000000
Data input zzzzzzzzzzz0zzzz, data output zzzzzzzzzzz1zzzz, device index        4, activity detect 0000000000010000 at
2100000
```

Many more points are there about a CSR. As the number of registers increase, you will have to improve the speed of access. Higher speed buses are used typically with serialization of the data. Many chips use I2C/UART/SPI as their external bus to access the CSR inside the chip. Another major problem is combining many IP block CSRs into one larger CSRs. In large SoCs, many blocks will come from vendors external to the design team, each with its own set of registers. Some may provide access through an I2C, some may provide access through APB bus. It is complicated to put together all the registers into one usable register map for the programming by customers. Because of the labor involved in maintaining many registers, many larger chip CSRs are made using scripts or software applications. The chipper updates sections he owns and the application or script outputs the SV code. A less well known complication is to keep the hardware matching with the datasheet. The sheer number of registers can cause mismatch between the description in datasheet with the hardware implementation. The datasheet could say that a register TEST_PATTERN_EN is at location 0x200 but the hardware could have it at 0x1200 because of miscommunication between the chipper and technical writer or because of forgetting to update the register generation apparatus.

The example here demonstrated only read-write and read-only registers. There are also quirky registers that show side effects when accessed. For example, you could read a counter and it will output the count value and automatically clear itself. This behavior is called a read to clear. You could to also have a write to clear and read to set or write to set types.

Performance

Suppose you have 10 registers of 8 bits each in your design and that access speed is 100kbps. It would take 800us to program all registers. Suppose you have 100k registers with same access speed. Just programming all the registers will take a good 8s. Many applications may not tolerate 8s of latency in starting the chip.

Distributed vs lumped CSR

Just one CSR is sufficient for a small chip. But if the registers count increases to say 10k coming from 20 IP blocks, it is hard to keep all the registers of the CSR in one place. It may cause timing and routing problems during physical implementation too. So, distributed architecture is preferred. In this version, every IP block has a CSR inside the IP hierarchy. At the top level appropriate decoding logic combines the block CSRs into one logical CSR map for use by software. One way to provide this functionality is to provide a base address parameter for every CSR, so that, the IP blocks can think of their register addresses as starting from 0 while the top can place the CSRs by using different base addresses.

Mixed Signal Models

Schmitt Trigger Inverter

Schmitt trigger inverter is a special kind of inverter. I would have named it inverter with memory or imposter combinational gate or positive feedback comparator or hysteritic inverter. The behavior of Schmitt triggering is easy to understand. Lets first recollect the behavior of a regular inverter. Suppose the inverter trip point is 0.5V, the inverter outputs a logic 0 for all input voltages above 0.5V and outputs logic 1 for all inputs below 0.5V. However, there is a problem with the regular inverter circuit. Around the trip point 0.5V, the inverter is super sensitive to the input signal. When the input is switching very slowly, the input signal may spend a long time around the trip point, 0.5V. If external noise of the shape of a small pulse of 0.1V interferes with the input then the regular inverter will faithfully amplify the 0.1V glitch into a full scale 0-Vdd or logic-0 to logic-1 output pulse. This is a spurious pulse on the output of the regular inverter which is unwanted.

This type of noise sensitivity is avoided in a Schmitt trigger inverter. It works by creating a zone where the circuit is not very sensitive to the input noise. There is a lower threshold voltage or lower trip point and an upper threshold or trip point. The testbench inputs a ramp signal from low voltage to high voltage. When the input is below the lower threshold, 0.25V, the output is set to 1 just like a regular inverter. When the input ramps up through the upper threshold, 0.83V, the output goes logic 0 just like a regular inverter. The interesting thing happens when the input ramps down from high voltages below the upper threshold. Unlike a regular inverter that would have switched without memory at the same upper threshold of 0.83V, the Schmitt trigger inverter switches from logic 1 to logic 0 only when the input falls through 0.25V. When the input voltage is in the zone of 0.25V to 0.83V the inverter "remembers" the state. Memory is implemented in circuits using positive feedback.

The testbench can be sanity checked by making the inverter module thresholds different from the thresholds used in the expected output logic.

```verilog
module schmitt_trigger_inv
#(
parameter real LT=0.2,
parameter real UT=0.8
)
(
input real in ,
output logic out
) ;

bit state;
initial state = $urandom_range(1);

always @(*)
  if ( state == 0  && in > UT )
   state = 1;
  else if ( state == 1  && in < LT )
   state = 0;
```

```verilog
assign out = ~state;

endmodule

module tb ;

timeunit 1ns ;
timeprecision 1ps ;
parameter STEP=1;

parameter real LT=0.25;
parameter real UT=0.83;

real ramp, ana_in;
bit result;
logic dig_out, dig_out_exp;

initial begin
  ramp = 0.05;
  repeat (2) begin
    while (ramp < 0.95) begin
    #STEP;
      ramp+= 0.001;
    end
    while (ramp > 0.05) begin
      #STEP;
      ramp-= 0.001;
    end
  end

  if ( result )
  $display ( "PASS" ) ;
  else begin
    $display ( "FAIL" ) ;
    $finish ;
  end

  $finish ;
end

assign ana_in = ramp ;

initial begin
  $display("Analog input = %f, Digital output = %b",ana_in,dig_out);
  forever @(dig_out) begin
    $display("Analog input = %f, Digital output = %b",ana_in,dig_out);
  end
end

//schmitt_trigger_inv #(.LT(LT), .UT(UT+0.1)) sinv (.in(ana_in) , .out(dig_out) ) ;
schmitt_trigger_inv #(.LT(LT), .UT(UT)) sinv (.in(ana_in) , .out(dig_out) ) ;

always @(ana_in)
  if (ana_in< LT)
```

```
    dig_out_exp = 1 ;
  else if ( ana_in > UT )
   dig_out_exp = 0 ;

initial begin
  result = 1;
  forever @(dig_out_exp) begin
   #0;
   if ( dig_out_exp != dig_out )
     result = 0 ;
  end
end

endmodule
```

run -all
Analog input = 0.000000, Digital output = 1
Analog input = 0.050000, Digital output = 1
Analog input = 0.830000, Digital output = 0
Analog input = 0.249000, Digital output = 1
Analog input = 0.830000, Digital output = 0
Analog input = 0.249000, Digital output = 1
PASS

Exercise
The logic in tb that generates the expected output is exactly same as the Schmitt trigger inverter model. So, the testbench does not really check anything. Can you inject noise of amplitude less than the difference between the upper and lower thresholds in one test and then higher than the upper – lower difference in another test?

Programmable Gain Amplifier

Amplifiers generally increase the signal voltage by an analog fixed gain value. PGA is a bit advanced because the gain is adjustable. The following model shows a PGA with gain coded as binary values with gain step as a multiple. Each step increases the gain by a fixed amount.

```
module thee_pg_amp
# (
parameter GAIN_BITS = 3 ,
parameter real GAIN_STEP = 1.0
 )
 (
 input real sig_in ,
 input logic [ GAIN_BITS-1 : 0 ] dig_gain ,
 output real sig_out
 ) ;

localparam NUM_GAINS = 2 ** GAIN_BITS ;
real gain_settings [ NUM_GAINS-1 : 0 ] ;

initial begin
```

```
    for ( int i = 0 ; i < NUM_GAINS ; i ++ )
    gain_settings [ i ] = ( i + 1 ) * GAIN_STEP ;

    forever @ ( * )
    sig_out = sig_in * gain_settings [ dig_gain ] ;
end

endmodule

module tb ;

parameter GB = 4 ;
parameter real GS = 0.5 ;
real sig_in ;
logic [ GB-1 : 0 ] dig_gain ;
real sig_out , exp ;
localparam NG = 2 ** GB ;

thee_pg_amp # ( .GAIN_BITS ( GB ) , .GAIN_STEP ( GS ) ) pga
 (
 .sig_in ,
 .dig_gain ,
 .sig_out
 ) ;

initial begin
    import thee_utils_pkg :: check_approx_equality ;
    bit result ;
    sig_in = 0.32 ; dig_gain = 3 ;
    #0 ;
    check_approx_equality ( .inp ( sig_out ) , .expected ( exp ) , .result ( result ) ) ;
    if ( result )
    $display ( "PASS" ) ;
    else
    $display ( "FAIL" ) ;

    $display ( "Signal output %f , Expected %f" , sig_out , exp ) ;

end

assign exp = sig_in * ( 1 + dig_gain ) * GS ;

endmodule

run -all
PASS
Signal output 0.640000, Expected 0.640000
exit
```

Exercise
A commercial PGA is available with gain not in regular steps but in some arbitrary steps. Model the gain part of this chip using an array for the gain parameter. Ignore the SPI interface and other features for now.

Integrator model

Analog circuits use integration to get the trend in a signal in place of the actual amplitude of the signal. In its simplest form, an integrator is simply an adder. It keeps adding a scaled version of the input continuously to a previously stored value or you may call that it accumulates the incoming signal amplitude into a memory. The stored accumulated value is the integral of the input. For modeling an integral function in mainly digital HDL like SV, we need to add/accumulate periodically in small time steps. The mathematical definition, however, needs the interval of addition to go to zero, meaning, the addition is supposed to be performed at infinite frequency which is not possible in a digital simulation. In engineering, infinite is approximated as something that is 100 times or more bigger than the variable in question. A zero is approximated by something that is 100 times smaller. Wikipedia has a nice graphic showing increasing accuracy with decreasing step size.

https://en.wikipedia.org/wiki/File:Riemann_Integration_and_Darboux_Upper_Sums.gif

An SV model of an integrator is given below. The unit step added to the existing value is the key statement in the model. Mathematics says that integral is like adding f(t)dt. In hardware, f(t) is the input value at time t and dt is the time difference, DT_STEP_SIZE parameter. The SCALE_FACTOR is used to adjust the output value to have amplitude useful for the next module to use. When using this model in your design, remember to set the time step about 10 to 100 times smaller than the rate of change of the input signal.

DC drift in integrator
An integrator continuously adds the input signal. If the signal is always positive the output will slowly exceed the supply voltage of the circuit. In ideal mathematical integration operation the output can go up to infinity. But the circuit model here saturates the output to a max value and similarly to a min value. In real circuits, integrator is regularly used with some kind of periodic reset operation.

```systemverilog
module thee_integrator
# (
parameter real RST_VAL = 0.0 ,
parameter realtime DT_STEP_SIZE = 1e-12 , // to be matched to integration forever loop time step
parameter real SCALE_FACTOR = 1.0 ,
parameter real INTEG_MAX = 1.0 ,
parameter real INTEG_MIN = -1.0
 )
 (
input real ana_in ,
output real integral
 ) ;

timeunit 1ns ;
timeprecision 1ps ;

real step ;
```

```
initial begin
  integral = RST_VAL ;
  forever begin
   #1ps ;
   step = ana_in * DT_STEP_SIZE * SCALE_FACTOR ;
   if ( integral + step > INTEG_MAX ) begin
    integral = INTEG_MAX ;
   end else if ( integral + step < INTEG_MIN ) begin
    integral = INTEG_MIN ;
   end else begin
    integral += step ;
   end
  end
end

endmodule
```

Testing the integrator
A step input to an integrator outputs a ramp. A sine wave input results in another sine wave output but with 90 degree phase lag. The following inputs a sine wave. The output frequency is checked for being same as with the input. The following code applies a step or a sine wave to the integrator model. The input frequency is 50ps * LUT_SIZE = 6400ps.

```
module tb ;

timeunit 1ns ;
timeprecision 1ps ;

real integral , ana_in , step ;
logic rstn ;

localparam real RST_VAL = 0.0 ;
localparam realtime DT_STEP_SIZE = 1e-12 ; // to be matched to integration forever loop time
step
localparam real SCALE_FACTOR = 1.0 ;
localparam real INTEG_MAX = 1.0 ;
localparam real INTEG_MIN = -1.0 ;

initial begin
  integral = RST_VAL ;
  forever begin
   #1ps ;
   step = ana_in * DT_STEP_SIZE * SCALE_FACTOR ;
   if ( integral + step > INTEG_MAX ) begin
    integral = INTEG_MAX ;
   end else if ( integral + step < INTEG_MIN ) begin
    integral = INTEG_MIN ;
   end else begin
    integral += step ;
   end
  end
end
```

```verilog
initial begin
  forever begin
    #20ps ;
    // $display ( "Input %1.3f Step %1.3e Integral %1.3e Current time %t" , ana_in , step , integral ,
$realtime ( ) ) ;
  end
end

 import thee_mathsci_consts_pkg :: const_pi ;
 real angle_rad ;
int LUT_SIZE = 128 ;

initial begin
  logic input_type = 1 ;
  if ( input_type == 0 ) begin
    ana_in = 0 ;
    #0.9ns ;
    rstn = 0 ;
    #1.9ns ;
    rstn = 1 ; ana_in = 1 ;
    #20ns ;
  end else begin
    for ( int i = 0 ; i < LUT_SIZE ; i = ( i + 1 ) %LUT_SIZE ) begin
      angle_rad = i * 2 * const_pi / LUT_SIZE ;
      ana_in = $sin ( angle_rad ) ;
      #50ps ;
    end
  end
end

initial begin
  #100ns ;
  $finish ;
end

realtime crossing , prev_crossing , wave_period ;
real prev_integral ;
initial begin
  forever @ ( integral ) begin
    if ( integral > 0 && prev_integral <= 0 ) begin
      prev_crossing = crossing ;
      crossing = $realtime ( ) ;
      wave_period = crossing - prev_crossing ;
      $display ( "Wave period output %t" , wave_period ) ;
    end
    prev_integral = integral ;
  end
end

initial begin
  realtime crossing , prev_crossing , wave_period ;
  real prev_integral ;
  forever @ ( ana_in ) begin
    if ( ana_in > 0 && prev_integral <= 0 ) begin
      prev_crossing = crossing ;
```

```
        crossing = $realtime ( ) ;
        wave_period = crossing - prev_crossing ;
        $display ( "Wave period input %t" , wave_period ) ;
      end
      prev_integral = ana_in ;
    end
end

endmodule
```

Output from simulation

Wave period input	50
Wave period output	50
Wave period input	6400
Wave period output	6400
Wave period input	6400

Exercise

If the period of the input waveform is 6400, why is the testbench detecting a period of 50 at the start? Analog circuits typically work with a common mode voltage. Does a common mode voltage apply to this integrator model? If so, how do you model it?

R String DAC

Resistor voltage divider is a piece of circuit commonly used in analog circuits. For any given reference voltage you can create any fraction of the reference voltage as output. This feature is used in the R String digital to analog converter. Instead of using only 2 resistors to create only one output voltage, the R String DAC uses many resistors to create many output voltages. In a DAC we need only one output voltage that is chosen by the input digital code. In the string DAC, this is achieved by setting the output voltage as an analog mux output of the voltages available from the resistor voltage divider. The select lines of the analog mux is the input digital code. The model shown below generates the 2^N resistors needed for an N-bit DAC and then initializes the resistance values to the ideal value +/- a random amount whose range is derived from the TOLERANCE parameter. After creating the resistor string model with tolerance, the code then creates the voltages that would appear on this resistor string. Note that the voltage calculation happens only during the initialization in the "initial" block. Upon simulating, the digital code input triggers the selection of one of the string voltage outputs.

```
module r_string_dac
# (
 parameter WIDTH = 8 ,
 parameter real UNIT_R = 1000 ,
 parameter real TOLERANCE_PCNT = 1.0
 )
 (
 output real ana ,
 input logic [ WIDTH-1 : 0 ] dig
 ) ;

timeunit 1ns ;
timeprecision 100ps ;
```

```
parameter RES_CNT = 2 ** WIDTH ;

import thee_utils_pkg :: add_tolerance ;

logic [ RES_CNT-1 : 0 ] switch_sel ;
real v_string [ RES_CNT-1 : 0 ] ;

initial begin
  real r_string [ RES_CNT-1 : 0 ] ='{default:UNIT_R};
  real r_total , r_div ;
  r_total = 0 ;
  foreach ( r_string [ i ] ) begin
    r_string [ i ] = add_tolerance ( r_string [ i ] , TOLERANCE_PCNT ) ;
    $display ( "r_string %f" , r_string [ i ] ) ;
    r_total += r_string [ i ] ;
  end

  r_div = 0 ;
  for ( int i = 0 ; i < RES_CNT ; i ++ ) begin
    v_string [ i ] = r_div / r_total ;
    r_div += r_string [ i ] ;
  end
end

assign ana = v_string [ dig ] ;

endmodule
```

Testbench

The TB inputs random digital code values to the DAC and compares against the expected output analog voltage to declare a pass or fail. You can see that for this example, a 1000 ohm resistor with 1% tolerance creates resistor instances of upto 1000 +/- 10 ohms, 990 to 1010 ohms.

```
module tb ;

timeunit 1ns ;
timeprecision 1ps ;

parameter WIDTH = 4 ;
parameter real VREF = 1.0;
real ana , expected;
real fullscale;
logic rstn ;
logic clk ;
logic [ WIDTH-1 : 0 ] dig ;
real dig_out_real ;
bit result ;

r_string_dac # ( .WIDTH ( WIDTH ) ) r_string_dac
 (
 .ana ,
 .dig
```

```verilog
  ) ;

initial begin
  import thee_utils_pkg :: urand_range_real ;
  fullscale = VREF*(2**WIDTH -1) / 2**WIDTH;
  for ( int i = 0 ; i < 5 ; i ++ ) begin
    dig = $urandom_range(2**WIDTH-1);
    expected = fullscale * dig / (2**WIDTH-1) ;
    #0;
    check_result ;
  end

  $finish ;
end

task check_result ;
 import thee_utils_pkg :: compare_real_fixed_err ;
 $display ( "Analog output %f , Digital input %d , expected analog %f" , ana , dig , expected ) ;
 compare_real_fixed_err ( .expected ( expected ) , .actual ( ana ) , .result ( result ) , .max_err (
1.002/(2**WIDTH) ) ) ;
 if ( result )
 $display ( "PASS" ) ;
 else begin
   $display ( "FAIL" ) ;
   // $finish ;
 end
 endtask

endmodule
```

Simulation
```
run -all
r_string 996.959549
r_string 996.042979
r_string 1004.157914
r_string 996.187590
r_string 993.897689
r_string 1000.491872
r_string 994.297794
r_string 1007.458590
r_string 1007.073713
r_string 1007.064743
r_string 1007.254745
r_string 1004.580437
r_string 997.526204
r_string 1007.104377
r_string 1006.168513
r_string 1004.056999
Analog output 0.564448 , Digital input  9 , expected analog 0.562500
PASS
Analog output 0.937808 , Digital input 15 , expected analog 0.937500
PASS
Analog output 0.564448 , Digital input  9 , expected analog 0.562500
PASS
Analog output 0.750888 , Digital input 12 , expected analog 0.750000
PASS
Analog output 0.564448 , Digital input  9 , expected analog 0.562500
PASS
```

Exercise: The above code does not model the voltage selection mux well. Can you model it using many stages of 2:1 analog muxes? Make the choice of gate level mux vs functional mux a model parameter.

R-2R DAC

The R string DAC suffers from the limitation of needing 2^N resistors for an N bit DAC. A more elegant from of connecting resistors to create a DAC results in the requirement of only 2N resistors. In this DAC only two values of resistors, say, R and the double of it 2R are used.

Ref:
https://www.allaboutcircuits.com/technical-articles/voltage-mode-r2r-dacs-operation-and-characteristics

The interesting features of the R-2R DAC are Thevenin reduction of the resistor network and superposition theorem. A function is used to achieve the Thevenin reduction starting at the output of the resistor network and recursively work backwards till the LSB node is reached. Per superposition theorem, the individual bits can be treated independent of each other and each bit voltage output can be summed up. This part is done in a loop.

```
module r2r_dac
# (
 parameter WIDTH = 8 ,
 parameter real UNIT_R = 1000 ,
 parameter real TOLERANCE_PCNT = 1.0
 )
 (
 output real ana ,
 input logic [ WIDTH-1 : 0 ] dig
 ) ;

timeunit 1ns ;
timeprecision 100ps ;

parameter RES_CNT = 2 ** WIDTH ;
parameter real R_FB = UNIT_R ;
parameter real VREF = 1.0 ;

real v_eff [ WIDTH-1 : 0 ] ;
real r_series [ RES_CNT-1 : 0 ] ;
real r_source [ RES_CNT-1 : 0 ] ;
real r_eff [ RES_CNT-1 : 0 ] ;

initial begin
 import thee_utils_pkg :: add_tolerance ;
  r_series [ RES_CNT-1 : 0 ] = '{ default : UNIT_R } ;
  r_series [ 0 ] = 2 * UNIT_R ;
  r_source [ RES_CNT-1 : 0 ] = '{ default : 2 * UNIT_R } ;
  foreach ( r_series [ i ] ) begin
   r_series [ i ] = add_tolerance ( r_series [ i ] , TOLERANCE_PCNT ) ;
  end
```

```systemverilog
    foreach ( r_source [ i ] ) begin
      r_source [ i ] = add_tolerance ( r_source [ i ] , TOLERANCE_PCNT ) ;
    end
    for ( int i = WIDTH-1 ; i >= 0 ; i-- ) begin
      thev_reduce ( i , WIDTH-1 , r_eff [ i ] , v_eff [ i ] ) ;
      $display ( "V thevenin effective of bit %2d = %2.4f" , i , v_eff [ i ] ) ;
    end
    forever @ ( dig ) begin
      ana = 0 ;
      foreach ( dig [ i ] )
      ana += dig [ i ] ? v_eff [ i ] : 0 ;
    end
  end

  function automatic void thev_reduce ( input int vindex , input int nodeindex , inout real reff
  , inout real vthev ) ;
    real vbranch ;
    real vthev_prev , reff_prev ;
    if ( nodeindex == 0 ) begin
      reff = get_eff_r_for_parallel ( r_series [ 0 ] , r_source [ 0 ] ) ;
      vthev = ( vindex == 0 ) ? VREF * r_series [ 0 ] / ( r_series [ 0 ] + r_source [ 0 ] ) : 0 ;
      $display ( "Vi %d ri %d r %f , v thevenin %f" , vindex , nodeindex , reff , vthev ) ;
      return ;
    end
    if ( vindex != nodeindex ) begin
      vbranch = 0 ;
      thev_reduce ( vindex , nodeindex-1 , reff_prev , vthev_prev ) ;
      vthev = vthev_prev * r_source [ nodeindex ] / ( reff_prev + r_series [ nodeindex ] + r_source
  [ nodeindex ] ) ;
      reff = get_eff_r_for_parallel ( r_source [ nodeindex ] , r_series [ nodeindex ] + reff_prev ) ;
    end else begin
      vbranch = VREF ;
      thev_reduce ( vindex , nodeindex-1 , reff_prev , vthev_prev ) ;
      vthev = vbranch * ( reff_prev + r_series [ nodeindex ] ) / ( reff_prev + r_series [ nodeindex ] +
  r_source [ nodeindex ] ) ;
      reff = get_eff_r_for_parallel ( r_source [ nodeindex ] , r_series [ nodeindex ] + reff_prev ) ;
    end
    $display ( "Vi %d ri %d r %f , v thevenin %f" , vindex , nodeindex , reff , vthev ) ;

  endfunction

endmodule

function automatic real get_eff_r_for_parallel ( input real r0 , input real r1 ) ;
  return ( r0 * r1 / ( r0 + r1 ) ) ;
endfunction
```

Testbench

```systemverilog
module tb ;

timeunit 1ns ;
timeprecision 1ps ;

parameter WIDTH = 4 ;
```

```verilog
parameter real VREF = 1.0;
real ana , expected;
real fullscale;
logic rstn ;
logic clk ;
logic [ WIDTH-1 : 0 ] dig ;
real dig_out_real ;
bit result ;

r2r_dac # ( .WIDTH ( WIDTH ) ) r_string_dac
 (
 .ana ,
 .dig
 ) ;


initial begin
  import thee_utils_pkg :: urand_range_real ;
  fullscale = VREF*(2**WIDTH -1) / 2**WIDTH;

  for ( int i = 0 ; i < 5 ; i ++ ) begin
   dig = $urandom_range(2**WIDTH-1);
   expected = fullscale * dig / (2**WIDTH-1) ;
   #0;
   check_result ;
  end

  $finish ;
end

task check_result ;
 import thee_utils_pkg :: compare_real_fixed_err ;
 $display ( "Analog output %f , Digital input %d , expected analog %f" , ana , dig , expected ) ;
 compare_real_fixed_err ( .expected ( expected ) , .actual ( ana ) , .result ( result ) , .max_err (
1.002/(2**WIDTH) ) ) ;
 if ( result )
 $display ( "PASS" ) ;
 else begin
  $display ( "FAIL" ) ;
  // $finish ;
 end
 endtask

endmodule
```

Simulation

```
run -all
Vi           3 ri              0 r 998.065117 , v thevenin 0.000000
Vi           3 ri              1 r 1001.142311 , v thevenin 0.000000
Vi           3 ri              2 r 1003.008932 , v thevenin 0.000000
Vi           3 ri              3 r 999.410100 , v thevenin 0.502030
V thevenin effective of bit  3 = 0.5020
Vi           2 ri              0 r 998.065117 , v thevenin 0.000000
Vi           2 ri              1 r 1001.142311 , v thevenin 0.000000
```

```
Vi              2 ri             2 r 1003.008932 , v thevenin 0.499638
Vi              2 ri             3 r 999.410100 , v thevenin 0.248805
V thevenin effective of bit  2 = 0.2488
Vi              1 ri             0 r 998.065117 , v thevenin 0.000000
Vi              1 ri             1 r 1001.142311 , v thevenin 0.498090
Vi              1 ri             2 r 1003.008932 , v thevenin 0.249225
Vi              1 ri             3 r 999.410100 , v thevenin 0.124107
V thevenin effective of bit  1 = 0.1241
Vi              0 ri             0 r 998.065117 , v thevenin 0.497478
Vi              0 ri             1 r 1001.142311 , v thevenin 0.249689
Vi              0 ri             2 r 1003.008932 , v thevenin 0.124935
Vi              0 ri             3 r 999.410100 , v thevenin 0.062214
V thevenin effective of bit  0 = 0.0622
Analog output 0.248805 , Digital input  4 , expected analog 0.250000
PASS
Analog output 0.750835 , Digital input 12 , expected analog 0.750000
PASS
Analog output 0.688351 , Digital input 11 , expected analog 0.687500
PASS
Analog output 0.124107 , Digital input  2 , expected analog 0.125000
PASS
Analog output 0.688351 , Digital input 11 , expected analog 0.687500
PASS
```

Exercise

It is one thing to use recursion to describe Thevenin decomposition for a circuit and another to actually get an electrically correct solution. Can you crosscheck the output of the model against a paper calculation for any one binary input?

The feedback resistor in the Opamp negative summer configuration is not used in the model. Can you add parameter for setting the model to unity follower or negative input? How would you deal with the negative voltage output when input is supplied to the minus input of opamp? Which configuration is more common for R-2R DACs implemented on-chip? Find out by creating a statistics from research papers on chip R-2R DACs.

Flash ADC model

Flash ADC is the simplest to understand and in some cases, as the name suggests, converts in a flash. But it is limited to around 8 bits because the circuit size grows exponentially with the number of bits. The following is a model of a flash ADC with synthesizable digital part and non synthesizable analog part. A series of comparators in the analog part output a raw code that is converted into digital code by the digital part.

Imagine you have to say high or low for an input analog signal. You call high when the signal is in the upper half of the full voltage range and low otherwise. For simplicity, let us set the voltage range to one. The threshold point will be at 0.5 for one bit. If you have the luxury of using 4 regions to denote the incoming signal, you could split the range to quarter, half, three-quarters and full. You could name these regions 0,1,2,3 for 0 to 0.25, 0.25 to 0.5, 0.5 to 0.75 and 0.75 to 1. As I wrote this, it occurred to me that natural languages have such digitization already upto 4 regions, quarters and 8 regions. If you take the inch units, it is common to split an inch into one-eighth parts. Direction is also split into one-eight parts. An ADC is not much different from these ideas.

https://en.wikipedia.org/wiki/Flash_ADC

When analog circuit and digital logic are involved in the same design they are generally separated into two regions. The tools for designing analog circuits have different needs than the tools for designing the digital part. Even the engineers have very different backgrounds that it is useful to carve separate regions for each.

```
module fadc (
input real ana_in ,
output logic [ 7 : 0 ] dig_out
 ) ;

logic [ 256-1 : 0 ] dig_raw ;

fadc_ana fadc_ana (
.ana_in ,
.dig_raw
 ) ;

fadc_dig fadc_dig (
.dig_raw ,
.dig_out
 ) ;

endmodule
```

Analog model
The analog part is a model of the circuits. A model means it is not the real thing, in this case it cannot be synthesized into working circuits. The digital part is synthesizable into gates with a logic synthesis tool. The analog circuit consists of $2^N - 1$ comparators. Each comparator checks if the input signal voltage is greater than the threshold voltage at that comparator. The thresholds are constants. So, in the model they are computed only once at the "initial" part of the simulation.

```
module fadc_ana (
input real ana_in ,
output logic [ 256-1 : 0 ] dig_raw
 ) ;

real thresholds [ 256-1 : 0 ] ;

initial begin
  foreach ( thresholds [ i ] ) begin
    thresholds [ i ] = i / 256.0 ;
  end
end

always_comb begin
  foreach ( dig_raw [ i ] ) begin
    dig_raw [ i ] = ( ana_in > thresholds [ i ] ) ;
  end
end

endmodule
```

Digital part

The digital part is a priority encoder. The priority here means that the highest position comparator output overrides the output from all other positions. The for loop starts from the highest index of the comparator and searches for 1. The index with 1 is taken as the code. Priority is enforced by the break statement. After the first 1, subsequent 1s do not matter. Note that there are no flip flops in this flash ADC model. This is because it is a purely combinational circuit that does not need any clock signal for its operation, though, you could also use clocked comparators and clocked digital logic to increase the speed of operation.

```systemverilog
module fadc_dig (
 input logic [ 256-1 : 0 ] dig_raw ,
 output logic [ 7 : 0 ] dig_out
 ) ;

always_comb begin
  for ( int i = 256-1 ; i > 0 ; i-- ) begin
    if ( dig_raw [ i ] ) begin
      dig_out = i ;
      break ;
    end
  end
end

endmodule

module tb ( ) ;

timeunit 1ns ;
timeprecision 1ps ;

real ana_in ;
logic [ 7 : 0 ] dig_out ;
real dig_out_real ;
 bit result ;

assign dig_out_real = dig_out / 256.0 ;

fadc dut
 (
.ana_in ,
.dig_out
 ) ;

initial begin
  import thee_utils_pkg :: urand_range_real ;
  for ( int i = 0 ; i < 20 ; i ++ ) begin
    ana_in = urand_range_real ( 0, 1.0);
  #1 ;
    check_result;
  end
  $finish ;
end
```

```
task automatic check_result ;
 import thee_utils_pkg :: compare_real_fixed_err ;
 $display ( "Analog input %f , Digital output %d , Output reconverted to analog %f" , ana_in ,
dig_out , dig_out_real ) ;
 compare_real_fixed_err ( .expected ( ana_in ) , .actual ( dig_out_real ) , .result ( result ) ,
.max_err ( 1.001 * 1.0 / 256 ) ) ;

 if ( result )
 $display ( "PASS" ) ;
 else begin
   $display ( "FAIL" ) ;
   $finish ;
 end
 endtask

endmodule
```

run -all
Analog input 0.270947 , Digital output 69 , Output reconverted to analog 0.269531
PASS
Analog input 0.105800 , Digital output 27 , Output reconverted to analog 0.105469
PASS
Analog input 0.266744 , Digital output 68 , Output reconverted to analog 0.265625
PASS
Analog input 0.588945 , Digital output 150 , Output reconverted to analog 0.585938
PASS

Exercise
Can you create a flash ADC with logarithmic thresholds? For example, consider a 3 bit ADC with the reference range from 0 to 1V. Regular flash ADC will have thresholds at 1/8,2*1/8,3*1/8,…,7*1/8. A logarithmic ADC would have finer steps around lower voltage and coarser steps at higher voltage, like, 1/128, 1/64,1/32, 1/16, 1/8, ¼ and ½. Can you create a 4-bit logarithmic flash ADC?

Counter type ADC model

I would say this is the simplest type of ADC. In terms of software, you could say this resembles brute force serial search algorithm, that is, go through all possible digital values until you find the digital code that matches with the input analog signal. The idea is simple, create an internal analog signal starting from zero (or lowest value) that is the output of a DAC driven by a counter. Compare the internal signal against the external signal while the counter counts up and, if the internal signal becomes greater than the external signal then we have found the code for the external signal.

My example design is split into two parts as usual – a synthesizable digital part and a non synthesizable analog model.

```
module cadc (
input real ana_in ,
input logic clk ,
input logic rstn ,
input logic start ,
```

```verilog
  output logic [ 7 : 0 ] dig_out ,
  output logic eoc
  ) ;

  logic cmp_out ;
  logic [ 8-1 : 0 ] cnt ;

  cadc_ana cadc_ana (
  .ana_in ,
  .start ,
  .rstn ,
  .cmp_out ,
  .cnt
  ) ;

  cadc_dig cadc_dig (
  .clk ,
  .rstn ,
  .start ,
  .eoc ,
  .cmp_out ,
  .cnt ,
  .dig_out
  ) ;

endmodule
```

Analog part contains the DAC and the comparator.
```verilog
module cadc_ana (
input real ana_in ,
input logic start ,
input logic rstn ,
input logic [ 7 : 0 ] cnt ,
output logic cmp_out
) ;

real dac_out ;
real ana_sampled ;

always @ ( posedge start ) begin
  ana_sampled <= ana_in ;
end

assign dac_out = 1.0 * cnt / ( 2 ** 8 ) ;
assign cmp_out = ana_sampled < dac_out ;

endmodule
```

The digital part consists of a counter and control logic that keeps track of start and end of conversion.

```verilog
module cadc_dig (
 input logic clk ,
 input logic rstn ,
 input logic cmp_out ,
 input logic start ,
```

```systemverilog
  output logic [ 7 : 0 ] cnt ,
  output logic [ 7 : 0 ] dig_out ,
  output logic eoc
 ) ;

logic [ 8-1 : 0 ] index ;
logic sync_clr ;
logic cnt_en ;
logic conversion_running ;

assign cnt_en = start | ( ( cnt == 2 ** 8-1 ) | ( cmp_out == 1 ) ? 0 : conversion_running ) ;

ehgu_cntr # ( .WIDTH ( 8 ) ) cntr (
.clk ,
.rstn ,
.sync_clr ,
.en ( cnt_en ) ,
.cnt
 ) ;

assign sync_clr = start ;

always_ff @ ( posedge clk , negedge rstn ) begin
  if ( !rstn ) begin
    conversion_running <= 0 ;
    dig_out <= 0 ;
    eoc <= 0 ;
  end else begin
    conversion_running <= start ? 1 : ( cnt == 2 ** 8-1 ) | cmp_out ? 0 : conversion_running ;
    dig_out <= cnt ;
    eoc <= start ? 0 : ( cnt == 2 ** 8-1 ) | cmp_out ;
  end
end

endmodule

module tb ( ) ;

timeunit 1ns ;
timeprecision 1ps ;

logic clk ;
logic rstn ;
real ana_in ;
logic [ 7 : 0 ] dig_out ;
real dig_out_real ;
logic start , eoc ;
bit result ;

assign dig_out_real = 1.0 * dig_out / 2 ** 8 ;

cadc dut
 (
.clk ,
.rstn ,
```

```verilog
    .ana_in ,
    .start ,
    .eoc ,
    .dig_out
    ) ;

initial begin
    clk = 0 ;
    forever begin
        clk = ~clk ;
        #5 ;
    end
end

initial begin
    import thee_utils_pkg :: urand_range_real ;
    repeat ( 2 ) @ ( posedge clk ) ;
    rstn = 0 ;
    repeat ( 10 ) @ ( posedge clk ) ;
    rstn = 1 ;
    repeat ( 10 ) @ ( posedge clk ) ;

    for ( int i = 0 ; i < 5 ; i ++ ) begin
        ana_in = urand_range_real ( 0 , 1.0 ) ;
        start = 1 ; repeat ( 2 ) @ ( posedge clk ) ; start = 0 ; @ ( posedge eoc ) ; @ ( posedge clk )
; // CHECKME why two cycles needed for start?
        check_result ;
    end

    $finish ;
end

task check_result ;
    import thee_utils_pkg :: compare_real_fixed_err ;
    $display ( "Analog input %f , Digital output %d , Output reconverted to analog %f" , ana_in ,
dig_out , dig_out_real ) ;
    compare_real_fixed_err ( .expected ( ana_in ) ,  .actual ( dig_out_real ) , .result ( result ) ,
.max_err ( 1.001 * 2.0 / 256  ) ) ;
    if ( result )
    $display ( "PASS" ) ;
    else begin
        $display ( "FAIL" ) ;
        $finish ;
    end
endtask
endmodule
```

```
run -all
Analog input 0.398200 , Digital output 102 , Output reconverted to analog 0.398438
PASS
Analog input 0.025547 , Digital output   7 , Output reconverted to analog 0.027344
PASS
Analog input 0.450497 , Digital output 116 , Output reconverted to analog 0.453125
PASS
```

Analog input 0.402304 , Digital output 103 , Output reconverted to analog 0.402344
PASS
Analog input 0.299205 , Digital output 77 , Output reconverted to analog 0.300781
PASS

Additional reference
https://www.elprocus.com/counter-type-adc-analog-to-digital-converter/

SAR ADC model

Successive Approximation Register ADC is binary search applied to finding the code that matches the incoming signal voltage. In binary numbers, moving one position right means working with a value half the previous bit position. So this nicely maps the division by 2 operator of binary search into just moving one bit position.

https://en.wikipedia.org/wiki/Successive_approximation_ADC

```
module sadc (
input real ana_in ,
input logic start ,
input logic rstn ,
input logic clk ,
output logic [ 7 : 0 ] dig_out ,
output logic eoc
 ) ;

logic [ 7 : 0 ] code ;
logic cmp_out ;

sadc_ana sadc_ana (
.ana_in ,
.start ,
.rstn ,
.cmp_out ,
.dig_out
 ) ;

sadc_dig sadc_dig (
.clk ,
.rstn ,
.start ,
.cmp_out ,
.dig_out ,
.eoc
 ) ;

endmodule
```

Analog model
The analog part consists of a DAC model that outputs an analog signal from the code that is being computed. A comparator checks if the equivalent analog signal from the DAC is higher or lower than

the input signal. There is also a sample and hold circuit that holds the analog value steady for conversion to proceed without corruption by changes in the input.

```verilog
module sadc_ana (
input real ana_in ,
input logic start ,
input logic rstn ,
input logic [ 7 : 0 ] dig_out ,
output logic cmp_out
 ) ;

real ana_sampled ;
real dac_out ;

always@ ( posedge start ) begin
  ana_sampled <= ana_in ;
end

assign dac_out = 1.0 * dig_out / ( 2 ** 8 ) ;
assign cmp_out = ana_sampled > dac_out ;

endmodule
```

Exercise
In the analog model for comparator output, change the ">" symbol to "<" symbol and see what happens to the output code?

Digital part
The code is stored in a register dig_out. A counter keeps track of which bit position is being tried. Additional logic clears the counter before every new conversion and also outputs the final code and end-of-conversion signal. The direction of change in the code value will be to constantly reduce the error between the input signal and analog value calculated from the current code value, successively improving the approximation of the digital code.

```verilog
module sadc_dig (
 input logic clk ,
 input logic rstn ,
 input logic cmp_out ,
 input logic start ,

 output logic [ 7 : 0 ] dig_out ,
 output logic eoc
 ) ;

logic [ $clog2 ( 8 ) -1 : 0 ] index , cnt ;
logic sync_clr ;
logic [ 7 : 0 ] code ;

logic conversion_running ;

assign index = 8-1-cnt ;

ehgu_cntr cntr (
```

```systemverilog
    .clk ,
    .rstn ,
    .sync_clr ,
    .en ( conversion_running ) ,
    .cnt
   ) ;

  assign sync_clr = start ;

  always_comb begin
    code = dig_out ;
    if ( start ) begin
      code = ( 2 ** ( 8-1 ) ) ;
    end else if ( index == 8-1 ) begin
      code [ index ] = 1'b1 ;
    end else if ( index inside { [ 0 : 8-2 ] } ) begin
      if ( cmp_out == 1'b1 ) begin
        code [ index + 1 ] = 1'b1 ;
        code [ index ] = 1'b1 ;
      end else begin
        code [ index + 1 ] = 1'b0 ;
        code [ index ] = 1'b1 ;
      end
    end
  end

  always_ff @ ( posedge clk , negedge rstn ) begin
    if ( !rstn ) begin
      conversion_running <= 0 ;
      dig_out <= 0 ;
      eoc <= 0 ;
    end else begin
      conversion_running <= start ? 1 : ( cnt == 8-1 ) ? 0 : conversion_running ;
      dig_out <= code ;
      eoc <= cnt == 8-1 ;
    end
  end

endmodule

module tb ( ) ;

timeunit 1ns ;
timeprecision 1ps ;

logic clk ;
logic rstn ;
real ana_in ;
logic [ 7 : 0 ] dig_out ;
real dig_out_real ;
logic start ;
logic eoc ;
bit result;

assign dig_out_real = dig_out / 256.0 ;
```

```verilog
sadc dut
 (
.clk ,
.rstn ,
.start ,
.ana_in ,
.dig_out ,
.eoc
 ) ;

initial begin
  clk = 0 ;
  rstn = 0 ;
  #1 ;
  clk = 0 ;
  rstn = 1 ;
  #1 ;
  forever begin
    clk = ~clk ;
    #5 ;
  end
end

initial begin
  import thee_utils_pkg :: urand_range_real ;
  start = 0 ;
  repeat ( 10 ) @ ( posedge clk ) ;

  for ( int i = 0 ; i < 5 ; i ++ ) begin
    ana_in = urand_range_real ( 0 , 1.0 ) ;
    start = 1 ; @ ( posedge clk ) ; start = 0 ; @ ( posedge eoc ) ; @ ( posedge clk );
    check_result ;
  end

  $finish ;
end


task check_result ;
 import thee_utils_pkg :: compare_real_fixed_err ;
 $display ( "Analog input %f , Digital output %d , Output reconverted to analog %f" , ana_in ,
dig_out , dig_out_real ) ;
 compare_real_fixed_err ( .expected ( ana_in ) , .actual ( dig_out_real ) , .result ( result ) ,
.max_err ( 1.001 * 2.0 / 256  ) ) ;
 if ( result )
 $display ( "PASS" ) ;
 else begin
   $display ( "FAIL" ) ;
   $finish ;
 end
 endtask

endmodule
```

run -all
Analog input 0.525452 , Digital output 135 , Output reconverted to analog 0.527344
PASS
Analog input 0.445294 , Digital output 113 , Output reconverted to analog 0.441406
PASS
Analog input 0.384250 , Digital output 99 , Output reconverted to analog 0.386719
PASS
Analog input 0.465663 , Digital output 119 , Output reconverted to analog 0.464844
PASS
Analog input 0.489359 , Digital output 125 , Output reconverted to analog 0.488281
PASS

Pipelined ADC model

Simple ADCs have only modest amount of digital logic. Pipelined ADC is a different with somewhat intriguing digital logic. The logic may not be that straight forward to all chippers. A mixed signal engineer may be asked to create this or an analog circuit engineer may have to do this in a small company. So, it is a nice example to have. A simple introduction to pipelined ADC is in this link -

https://www.maximintegrated.com/en/app-notes/index.mvp/id/1023

The top module for pipelined ADC is split into analog and digital.

```
module padc (
input real ana_in ,
input logic clk ,
input logic rstn ,
output logic signed [ 7 : 0 ] dig_out
 ) ;

logic [ 1 : 0 ] dig_raw [ 7 ] ;

padc_ana padc_ana (
.ana_in ,
.clk ,
.rstn ,
.dig_raw
 ) ;

padc_dig padc_dig (
.dig_raw ,
.clk ,
.rstn ,
.dig_out
 ) ;

endmodule
```

Analog model
The model is split into stages. These stages could also be modeled with a for or for-generate loop but, I have chosen to use one module for one stage. It is useful to match the implementation hierarchy with textbook theoretical hierarchy for easy understanding by all readers. The output of one stage feeds the

next, this behavior is obtained with the use of different indices in the left and right sides of the assignment.

```
module padc_ana (
input real ana_in ,
input logic clk ,
input logic rstn ,
output logic [ 1 : 0 ] dig_raw [ 7 ]
 ) ;

real vin_vec [ 8 ] ;
real residue_vec [ 7 ] ;

generate
 for ( genvar i = 0 ; i < 7 ; i ++ ) begin
  : stages
  padc_ana_stage padc_ana_stage (
  .vin ( vin_vec [ i ] ) ,
  .clk ,
  .rstn ,
  .dig_raw ( dig_raw [ i ] ) ,
  .residue ( residue_vec [ i ] )
  ) ;
 end
endgenerate

always_comb begin
 vin_vec [ 0 ] = ana_in ;
 vin_vec [ 1 : 7 ] = residue_vec [ 0 : 6 ] ;
end

endmodule
```

Each stage consists of a sample and hold model, comparator outputs and residue creation part to feed the next stage. Sample and hold can be thought of as a flip flop for analog signals. Just as a flip flop samples a digital 1 or 0 and holds it till the next sampling point arrives, a sample and hold circuit samples the analog signal or real value and holds it till the next sampling point. Note however that there is a major difference between analog and digital. The analog signal in a real sample and hold circuit slowly gets corrupted by noise, so there is an expiry time for an analog sample. But the digital 0 or 1 in a flip flop is forever.

```
module padc_ana_stage (
input real vin ,
input logic clk ,
input logic rstn ,
output logic signed [ 1 : 0 ] dig_raw ,
output real residue
 ) ;

real sampled_value ;

always @ ( posedge clk , negedge rstn ) begin
  if ( !rstn ) begin
```

```
      sampled_value <= 0 ;
    end else begin
      sampled_value <= vin ;
    end
    //$display ( "%m Sampled Value %f" , vin ) ;
  end

  always_comb begin
    if ( sampled_value < -0.25 ) begin
      dig_raw = -1 ;
    end else if ( sampled_value < 0.25 ) begin
      dig_raw = 0 ;
    end else begin
      dig_raw = + 1 ;
    end

    residue = 2 * sampled_value - 1.0 * dig_raw ;
  end

endmodule
```

Digital part

The digital part has two sub divisions, a time alignment part and a summing part. The analog stage that receives the input signal outputs its digital value in the first clock cycle, the second analog stage in the second cycle and so on. But the formula for summing needs all these values at the same time. So, we need to equalize the cycle delays. So, the first stage is delayed by maximum amount. The second stage is delayed by one less cycles and so on. In chipping, we have to focus on the expressive efficiency of code. By this I mean, write shortest code that is general and easy to understand. In this case a block delay chain is used to delay the output of all stages and then the aligned output is selected as taps from specific points in the delay chain. Now one may wonder, how may delay stages should be needed ideally? In case of 8 bit pipeline ADC using 7 stages, we need to delay the first stage by 7 cycles, second by 6 and last by 1. So, it is 1+2+3+4+5+6+7 = 28 delays with 2 bits each. But then why is the variable dig_raw_delayed defined as 7x7 = 49 delays? What is the impact of the remaining 49-28=21 delays? There is no impact in terms of silicon (or more generally semiconductor) area. Synthesis tools typically delete the unused flip flops from the gate netlist. There is however a compile time, simulation time and waveform file size penalty. The unused stages also get compiled, simulated and saved into the waveform file. Then why code with unused logic? Just to make the design general and simple to read. A simpler readable design is preferable to an unreadable design that simulates faster. This is because designer productivity is a bigger bottleneck than computer simulation speed.

```
module padc_dig (
 input logic clk ,
 input logic rstn ,
 input logic [ 1 : 0 ] dig_raw [ 7 ] ,
 output logic signed [ 7 : 0 ] dig_out
 ) ;

logic [ 1 : 0 ] dig_raw_delayed [ 7 ] [ 7 ] ;
logic [ 1 : 0 ] dig_raw_aligned [ 7 ] ;
```

```systemverilog
always_ff @ ( posedge clk , negedge rstn ) begin
  if ( !rstn ) begin
    dig_raw_delayed <= '{ default : 0 } ;
  end else begin
    dig_raw_delayed [ 0 ] <= dig_raw ;
    for ( int i = 1 ; i < 7 ; i ++ ) begin
      dig_raw_delayed [ i ] <= dig_raw_delayed [ i-1 ] ;
    end
  end
end

always_comb begin
  foreach ( dig_raw_aligned [ i ] ) begin
    dig_raw_aligned [ i ] = dig_raw_delayed [ 6-i ] [ i ] ;
  end
end

always_comb begin
  dig_out = 0 ;
  foreach ( dig_raw_aligned [ i ] ) begin
    //$display ( "Dig raw aligned %b index %d" , dig_raw_aligned [ i ] , i ) ;
    if ( dig_raw_aligned [ i ] == 2'b01 ) begin
      dig_out += ( 2 ** ( 6-i ) ) ;
    end else if ( dig_raw_aligned [ i ] == 2'b11 ) begin
      dig_out -= ( 2 ** ( 6-i ) ) ;
    end else begin
      dig_out += 0 ;
    end
    // dig_out += ( 2 ** ( 6-i ) ) * $signed ( dig_raw_aligned [ i ] ) ;
  end
  //$display ( "Dig output %d" , dig_out ) ;

end

endmodule

module tb ( ) ;

timeunit 1ns ;
timeprecision 1ps ;

logic clk ;
logic rstn ;
real ana_in ;
logic signed [ 7 : 0 ] dig_out ;
real dig_out_real ;

assign dig_out_real = dig_out / 127.0 ;

padc dut
 (
.clk ,
.rstn ,
.ana_in ,
.dig_out
```

```verilog
) ;

initial begin
  clk = 0 ;
  rstn = 0 ;
  #1 ;
  clk = 0 ;
  rstn = 1 ;
  #1 ;
  forever begin
    clk = ~clk ;
    #5 ;
  end
end

initial begin
  import thee_utils_pkg :: urand_range_real ;
  repeat ( 5 ) @ ( posedge clk ) ;

  for ( int i = 0 ; i < 20 ; i ++ ) begin
    ana_in = urand_range_real ( 0, 2.0 ) -1.0;
    check_result(ana_in);
    repeat ( 1 ) @ ( posedge clk ) ;
  end

  repeat ( 10 ) @ ( posedge clk ) ;
  $finish ;
end


task automatic check_result ( input real ana_in );
 import thee_utils_pkg :: compare_real_fixed_err ;
 bit result;
 fork
 begin
  repeat ( 9 ) @ ( posedge clk ) ;
  $display ( "Analog input %f , Digital output %d , Output reconverted to analog %f" , ana_in ,
dig_out , dig_out_real ) ;
  //check_approx_equality ( .inp ( dig_out_real ) , .expected ( ana_in ) , .result ( result ) , .tolerance
( 1.001 * 2.0 / 127  ) , .tolerance_for_zero( 1.001 * 2.0 / 127 )) ;
  compare_real_fixed_err (.expected ( ana_in ) ,  .actual ( dig_out_real ) , .result ( result ) ,
.max_err ( 1.001 * 2.0 / 256  ) ) ;

  if ( result )
  $display ( "PASS" ) ;
  else begin
    $display ( "FAIL" ) ;
    $finish ;
  end
 end
 join_none
 endtask


endmodule
```

run -all
Analog input -0.940087 , Digital output -120 , Output reconverted to analog -0.944882
PASS
Analog input 0.003510 , Digital output   0 , Output reconverted to analog 0.000000
PASS
Analog input -0.815237 , Digital output -104 , Output reconverted to analog -0.818898
PASS
Analog input -0.260670 , Digital output -33 , Output reconverted to analog -0.259843
PASS
Analog input 0.319461 , Digital output   41 , Output reconverted to analog 0.322835
PASS
Analog input 0.067258 , Digital output   9 , Output reconverted to analog 0.070866
PASS

Exercise
Design a mixed 8-bit ADC with MSB 5bits pipelined with LSB 3bits flash. How many comparators and clock cycles will be needed for one conversion?

Why does the sample and hold model described here does not decay the value of the sample even though it is analog? How may the decaying effect be modeled? When does it become necessary to model the decaying effect?

Delta Sigma ADC model

When I was a student in a masters degree program, my professor gave me an assignment to create a temperature sensor that uses a Delta Sigma ADC with calibration feature. The DS ADC also had to output the temperature value on an I2C port. That assignment was too complex. I barely understood DS ADC. I don't understand DS ADC well even now. Now, after many years, I tried to create a SV model of it anyway! To my surprise, the model actually works! My friend told me that it is called a first order DS ADC. This book gives a great conceptual view of DS ADC.

https://www.amazon.com/Understanding-Delta-Sigma-Converters-Microelectronic-Systems-ebook/dp/B01MT1I21F

The conceptual examples from the book about measuring a height with a fixed thickness book and the problem of paying a change to a coffee shop with only whole number money is very interesting. The concepts part of the book is present in the book sample available for free in Amazon Kindle store. Unlike other ADCs, DS does not give a one input one output operation. DS output is more for the purpose of converting an analog wave signal. It can be used for one time conversion as well but the DS ADC does not provide a simple signal for end of conversion indication. End of conversion has to be inferred from the output digital value.

The SV model has two sections, analog and digital. The analog section uses an integrator, a comparator, a difference operation and a cycle delay. The digital part implements averaging and decimation. It turns out a counter that accumulates the input bit stream and then outputs the

accumulated value is the same as averaging and decimation. Note that there are other ways that actually use a digital filter and then a decimator that can give better noise reduction.

```systemverilog
module ds_adc
# (
 parameter WIDTH = 8 ,
 parameter OVERSAMP_RATIO = 256
 )
(
 input logic clk_oversamp ,
 input logic rstn ,
 input real ana_in ,
 input logic clk ,
 output logic signed [ WIDTH-1 : 0 ] dig_out
 ) ;

timeunit 1ns ;
timeprecision 100ps ;

logic comp_out ;

ds_adc_ana ds_adc_ana
 (
 .clk_oversamp ,
 .rstn ,
 .ana_in ,
 .comp_out
 ) ;

ds_adc_dig # ( .WIDTH ( WIDTH ) , .OVERSAMP_RATIO ( OVERSAMP_RATIO ) ) ds_adc_dig
 (
 .comp_out ,
 .clk ,
 .clk_oversamp ,
 .rstn ,
 .dig_out
 ) ;

endmodule
```

Analog section

```systemverilog
module ds_adc_ana
 (
 input logic clk_oversamp ,
 input logic rstn ,
 input real ana_in ,
 output logic comp_out
 ) ;

timeunit 1ns ;
timeprecision 100ps ;
```

```verilog
  real diff , dac_out , integrated_diff ;
  logic comp_out_d1 ;

  assign diff = ana_in - dac_out ;

  thee_integrator integrator
   (
  .ana_in ( diff ) ,
  .integral ( integrated_diff )
   ) ;

  assign comp_out = integrated_diff > 0 ? 1 : 0 ;

  always_ff @ ( posedge clk_oversamp or negedge rstn ) begin : proc_comp_out_d1
   if ( ~rstn ) begin
     comp_out_d1 <= 0 ;
   end else begin
     comp_out_d1 <= comp_out ;
   end
  end

  assign dac_out = comp_out_d1 ? + 1 : -1 ;

  endmodule
```

Digital Section

This part may have clock domain crossing issues, so please study about DS clock domain crossing topics before using this RTL in real designs. This part does two functions, counting the ones in the input bitstream. This is achieved by enabling the counter only when the input stream as a one. The second function is to transfer the count periodically to the output clock domain. The frequency difference between the input clock and the output clock is set to be the oversampling ratio of the ADC. The clock crossing is somewhat clunky. Whenever the output clock rises a pulse is created in the fast clock domain that both transfers the count to a holding register and then clears the count for the next counting operation. The holding register is transferred to the output clock domain as the converted digital output.

```verilog
module ds_adc_dig
 # (
 parameter WIDTH = 8 ,
 parameter OVERSAMP_RATIO = 256
 )
 (
 input logic clk_oversamp ,
 input logic rstn ,
 input logic clk ,
 input logic comp_out ,
 output logic signed [ WIDTH-1 : 0 ] dig_out
 ) ;

 logic sync_clr ;
 logic [ WIDTH-1 : 0 ] cnt ;
```

```systemverilog
ehgu_cntr # ( .WIDTH ( WIDTH ) ) cntr (
.clk ( clk_oversamp ) ,
.rstn ,
.sync_clr ,
.en ( comp_out ) ,
.cnt
 ) ;

logic copy_cnt , ckdelayed , data_ready , data_ready_slow , load ;
logic [ WIDTH-1 : 0 ] cnt_copied ;

ehgu_dly # ( .DELAY ( 3 ) ) delck ( .clk ( clk_oversamp ) , .rstn , .din ( clk ) , .dout ( ckdelayed ) )
;
ehgu_redge gen_samp ( .clk ( clk_oversamp ) , .rstn , .din ( ckdelayed ) , .redge ( copy_cnt ) ) ;

assign sync_clr = copy_cnt ;

always_ff @ ( posedge clk_oversamp or negedge rstn ) begin
  if ( ~rstn ) begin
    cnt_copied <= 0 ;
    data_ready <= 0 ;
  end else if ( copy_cnt ) begin
    cnt_copied <= cnt ;
    data_ready <= ~data_ready ;
  end
end

ehgu_rst_sync synchronize_rst ( .clk ( clk ) , .rstn_in ( rstn_in ) , .rstn_out ( rstn_synced ) ) ;

ehgu_dly # ( .DELAY ( 3 ) ) delready ( .clk ( clk ) , .rstn ( rstn_synced ) , .din ( data_ready ) , .dout
( data_ready_slow ) ) ;
ehgu_edges gen_load ( .clk ( clk ) , .rstn ( rstn_synced ) , .din ( data_ready_slow ) , .toggle ( load
) , .redge ( ) , .fedge ( ) ) ;

logic signed [ WIDTH-1 : 0 ] dig_scale_adjusted ;
assign dig_scale_adjusted = cnt_copied - OVERSAMP_RATIO / 2 ;

always_ff @ ( posedge clk or negedge rstn_synced ) begin
  if ( ~rstn_synced ) begin
    dig_out <= 0 ;
  end else if ( load ) begin
    dig_out <= dig_scale_adjusted ;
  end
end

endmodule


module tb ;

timeunit 1ns ;
timeprecision 1ps ;

parameter WIDTH = 8 ;
```

```systemverilog
  real integral , ana_in , step ;
  logic rstn ;
  logic clk_oversamp ;
  logic clk ;
  logic signed [ WIDTH-1 : 0 ] dig_out ;
  real dig_out_real ;
  bit result ;

  parameter OVERSAMP_RATIO = 256 ;
  parameter real FREQ_CLK_OVERSAMP = 256 ;

  thee_clk_gen_module # ( .FREQ ( FREQ_CLK_OVERSAMP / OVERSAMP_RATIO ) ) clk_gen ( .clk ( clk
) ) ;
  thee_clk_gen_module # ( .FREQ ( FREQ_CLK_OVERSAMP ) ) clk_gen_oversamp ( .clk
( clk_oversamp ) ) ;
  assign dig_out_real = dig_out / ( OVERSAMP_RATIO / 2.0 ) ;

  ds_adc # ( .WIDTH ( WIDTH ) , .OVERSAMP_RATIO ( OVERSAMP_RATIO ) ) ds_adc
  (
  .clk_oversamp ,
  .rstn ,
  .ana_in ,
  .clk ,
  .dig_out
  ) ;


  initial begin
    import thee_utils_pkg :: urand_range_real ;
    rstn = 0 ; repeat ( 2 ) @ ( posedge clk ) ; rstn = 1 ;

    repeat ( 10 ) @ ( posedge clk ) ;

    for ( int i = 0 ; i < 5 ; i ++ ) begin
      ana_in = urand_range_real ( 0 , 2.0 ) - 1.0 ;
      repeat ( 20 ) @ ( posedge clk ) ;
      check_result ;
    end

    $finish ;
  end

  task check_result ;
   import thee_utils_pkg :: compare_real_fixed_err ;
   $display ( "Analog input %f , Digital output %d , Output reconverted to analog %f" , ana_in ,
dig_out , dig_out_real ) ;
   compare_real_fixed_err ( .expected ( ana_in ) , .actual ( dig_out_real ) , .result ( result ) ,
.max_err ( 1.001 * 1.0 * 2 / 256 ) ) ;
   if ( result )
   $display ( "PASS" ) ;
   else begin
    $display ( "FAIL" ) ;
    // $finish ;
   end
   endtask
```

run -all
Analog input -0.431077 , Digital output -56 , Output reconverted to analog -0.437500
PASS
Analog input -0.251434 , Digital output -32 , Output reconverted to analog -0.250000
PASS
Analog input -0.761479 , Digital output -97 , Output reconverted to analog -0.757812
PASS
Analog input -0.061801 , Digital output   -9 , Output reconverted to analog -0.070312
FAIL
Analog input -0.000057 , Digital output   -1 , Output reconverted to analog -0.007812
PASS

Exercise

Why does the test fail for one test input?
What is the best way to do clock domain crossing for a DS ADC oversampling to regular clock domains? Can the low frequency clock can be derived from the high frequency by dividing by oversampling ratio.

Phase Locked Loop

You are in a public park swinging your kid. Your friend arrives with his kid. You both keep swinging. Your friend's kid is swinging slightly ahead of your kid. Your kid cries that she wants to be ahead of your friend's kid. Just for fun you tell your kid that her swing will catch up with the swing of your friend's kid soon. To do this, whenever the other swing goes past your swing, you push harder. Whenever the other swing lags behind yours you pull back a bit to slow down. In a few cycles of doing this both the swings oscillate in unison!

PLL stands for Phase Locked Loop. If I were to name it, I would have called it "Clock Magic" because a PLL really does a lot of magic with clocks. In its simplest avatar, a PLL just takes an input clock of one frequency and then outputs a clock of a different frequency. The ratio of the frequencies of input and output are constant. You can think of a PLL as a circuit that can produce a lower jitter output clock from a higher jitter input clock. So, I like calling it a clock jitter filter. It can also take a low frequency clock say 32kHz and generate a higher frequency clock like 100MHz. So, I like calling it a clock multiplier. It can do many more things. Making a PLL work in a chip at all corners is black magic. So, PLL design work is mostly done by specialists who work exclusively on PLLs for many years. If you don't care too much about the accuracy of the PLL behavior, it is fairly easy to model in SystemVerilog.

https://en.wikipedia.org/wiki/Phase-locked_loop

Exercise

Where is the PLL in the park story?

Solution

In the story, you become the PLL. You consciously monitor the difference in position between the two swings and force the difference to zero. You were a PLL in the operating range of a few fraction of Hz frequencies. Real chip PLLs operate as high as many tens of GHz. That is many billions of times faster!

Model

A first model of a PLL is nothing but a clock frequency dependent clock generator. The model can be seen in 3 parts – input period measurement part, an output clock generator part and a PLL lock detection part. The basic model does not do phase locking or frequency tracking or jitter filtering. It is still useful for clock multiplication purpose. In may digital logic simulations the phase, jitter and frequency change behavior of the chip are useless. So, this model has some use in at least checking if your PLL is connected properly.

Measurement

The logic of clock frequency measurement is fairly simple. Take the time difference between two consequtive rise edges. In a real PLL, the measurement is actually phase difference between two clocks and that measurement happens continuously.

Clock Generator

After the input reference clock is measured, the execution passes into a forever loop generating the clock.

Lock detection

PLL lock detection is a complicated task. In the simple model, a fixed number of toggling of the output clock is taken to mean that the PLL is locked. The lock signal is used in full chip and system level to force waiting of the software or downstream logic to signal that the clock output has stabilized.

```systemverilog
module thee_pll
# (
parameter string MODEL_TYPE = "basic" ,
parameter MEAS_CYCLES = 1
 )
 (
input logic refclk ,
input integer ref_div ,
input integer fb_div ,
output logic clkout ,
output logic lock
 ) ;

timeunit 10ps ;
timeprecision 1ps ;

 realtime first_rise_edge , second_rise_edge ;
 realtime clkout_half_period , period , sum_of_periods , avg_period ;

 real freq_in_Hz ;
```

```systemverilog
  generate
  if ( MODEL_TYPE == "basic" ) begin
    : basic_model
    initial begin
      clkout = 0 ;

      repeat ( 10 ) @ ( posedge refclk ) ;
      sum_of_periods = 0 ;
      @ ( posedge refclk ) ;
      first_rise_edge = $realtime ( ) ;
      repeat ( MEAS_CYCLES ) @ ( posedge refclk ) begin
        second_rise_edge = $realtime ( ) ;
        period = second_rise_edge - first_rise_edge ;
        sum_of_periods += period ;
        first_rise_edge = second_rise_edge ;
      end
      avg_period = ( sum_of_periods / MEAS_CYCLES ) ;
      clkout_half_period = avg_period * ( 1.0 * ref_div / fb_div ) / 2.0 ;
      $display ( "Half p %1.3e , avg p %1.3e , ref div %d fb div %d " , clkout_half_period , avg_period
, ref_div , fb_div ) ;
      clkout = 0 ;
      forever begin
        # ( clkout_half_period ) ;
        clkout = 0 ;
        # ( clkout_half_period ) ;
        clkout = 1 ;
      end
    end

    initial begin
      lock = 0 ;
      repeat ( 10 ) @ ( posedge clkout ) ;
      lock = 1 ;
    end

  end // else if ( CLK_GEN_TYPE == "jitter_only" ) begin
    // : ckgen_jitter_only

  endgenerate

endmodule
```

The testbench sets the input clock frequency, the PLL clock change ratio and checks if the output frequency matches the expected frequency. Note the legacy C syntax issues here. In C, integer 5 divided by integer 32 results in 0. SV inherits the same datatype conversion headache. So, it is necessary to do a 1.0 * multiplication or cast operation with real datatype.

```systemverilog
module tb ;
import thee_utils_pkg :: check_approx_equality ;

localparam real REF_FREQ = 100e6 ;
localparam MEAS_WINDOW = 10 ;
logic refclk , clkout ;
```

```verilog
real fout , exp_fout ;
bit result ;
logic pll_lock ;
int fb_div , ref_div ;

thee_clk_gen_module # ( .FREQ ( REF_FREQ / 1e6 ) ) ref_gen ( .clk ( refclk ) ) ;

thee_pll pll
 (
 .refclk ,
 .ref_div ,
 .fb_div ,
 .clkout ,
 .lock ( pll_lock )
 ) ;

thee_clk_freq_meter # ( .MEAS_WINDOW ( MEAS_WINDOW ) ) fmeter ( .clk ( clkout ) ,
.freq_in_hertz ( fout ) ) ;

initial begin
  ref_div = 5 ;
  fb_div = 32 ;
  exp_fout = REF_FREQ * ( real' ( fb_div ) / ref_div ) ;
  $display ( "Choosing input division %3d , feedback division %3d" , ref_div , fb_div ) ;
  wait ( pll_lock ) ;
  repeat ( MEAS_WINDOW + 10 ) @ ( posedge clkout ) ;
  $display ( " Clkout frequency %1.3e , expected %1.3e" , fout , exp_fout ) ;
  check_approx_equality ( .inp ( fout ) , .expected ( exp_fout ) , .result ( result ) ) ;
  if ( result == 1 ) begin
   repeat ( 3 ) $display ( "PASS" ) ;
  end else begin
   repeat ( 3 ) $display ( "FAIL" ) ;
  end

  $finish ;
end

endmodule
```

run -all
Choosing input division   5, feedback division 32
Half p 7.812e+01, avg p 1.000e+03, ref div   5 fb div   32
 Clkout frequency 6.402e+08 , expected 6.400e+08
PASS

Exercise
Make the model continuously measure the input clock period and continuously update the clock
generator half period value.
The model here does not match the phase of reference clock with the phase of the divided feedback
clock. Add this phase matching behavior to the model.
Add a lock detection circuit model that checks if the last N cycle periods of the clock did not differ
from the average period by more than a parameter, say 0.1% or 1/some power of 2.

Fractional N PLL Model

Regular PLL has an M/N type output clock to input clock frequency ratio relationship where N is an integer. Fractional N PLL has a M/N.frac type relationship where N.frac is a fractional number. So, unlike a PLL that can only do clock change like 12/3, 200/12, 240/31 etc., a fractional PLL can do 12/3.125, 200/12.0, 240/31.375 etc., which gives system designers much more control over the clocking of the full chip and system.

My implementation of the fractional N PLL model has 5 parts – phase frequency detector PFD, charge pump CP, low pass filter LPF, voltage controlled oscillator VCO and a fractional clock divider.

```systemverilog
module pll_model
# (
parameter INT_WIDTH = 3 ,
parameter FRAC_WIDTH = 4
 )
 (
input logic clk_ref ,
input logic rstn ,
input logic en ,
input logic [ INT_WIDTH-1 : 0 ] int_div ,
input logic [ FRAC_WIDTH-1 : 0 ] frac_div ,
output logic clkout ,
output logic lock
 ) ;

logic up , down ;
logic clk_vco ;
logic clk_fb ;
real cp_out ;
real lpf_out ;

thee_pfd pfd
 (
 .clk0 ( clk_ref ) ,
 .clk1 ( clk_fb ) ,
 .up ( up ) ,
 .down ( down )
 ) ;

thee_charge_pump cp (
.up ,
.down ,
.vout ( cp_out )
 ) ;

thee_low_pass_filter # ( .TAPS ( 10 ) , .STEP_SIZE_IN_NS ( 0.1 ) ) lpf (
 .sig_in ( cp_out ) ,
 .filtered_out ( lpf_out )
 ) ;

thee_vco vco (
 .vin ( lpf_out ) ,
 .clk ( clk_vco )
```

```
    ) ;

ehgu_clkdiv_fractional # ( .INT_WIDTH ( INT_WIDTH ) , .FRAC_WIDTH ( FRAC_WIDTH ) ) clkdiv0
 (
 .clkin ( clk_vco ) ,
 .rstn ,
 .en ( 1'b1 ) ,
 .int_div ,
 .frac_div ,
 .clkout ( clk_fb )
 ) ;

 assign clkout = clk_vco;

endmodule
```

PFD

The PFD circuit takes two clock signals and outputs 2 bits indicative of which clock is leading which clock.
https://www.analog.com/media/en/training-seminars/tutorials/MT-086.pdf

```
module thee_pfd
 (
input logic clk0 ,
input logic clk1 ,
output logic up ,
output logic down
 ) ;

logic q0 , q1 ;
logic rst_pfd ;

always_ff @ ( posedge clk0 or posedge rst_pfd ) begin
  if ( rst_pfd ) begin
    q0 <= 0 ;
  end else begin
    q0 <= 1'b1 ;
  end
end

always_ff @ ( posedge clk1 or posedge rst_pfd ) begin
  if ( rst_pfd ) begin
    q1 <= 0 ;
  end else begin
    q1 <= 1'b1 ;
  end
end

assign rst_pfd = q0 & q1 ;

assign { up , down } = { q0 , q1 } ;

endmodule
```

Charge Pump

By switching two current sources, one adding more charge and another removing charge, an average output voltage can be created. Think of this as converting the digital phase infromation from the PFD into an analog signal. The charging of a capacitor with current sources is like an integration operation. So, I have used an integrator internal to the charge pump. May be the CP can directly drive the subsequent low pass filter. I had to increase the integrator gain for my simulations to show PLL frequency locking behavior. The gain and LPF bandwidth are important parameters to be tuned in if the PLL has to work well in a real chip.

Exercise: How to model the nonlinearity and mismatch in the voltage dependence of current value of the charge pump?

```
module thee_charge_pump
 (
input logic up ,
input logic down ,
output real vout
 ) ;

real i0 , i1 ;
real current_in ;

assign i0 = up ? 1.0 : 0.0 ;
assign i1 = down ? -1.0 : 0.0 ;

assign current_in = i0 + i1 ;


thee_integrator # ( .SCALE_FACTOR ( 1e6 ) ) integrator
 (
.ana_in ( current_in ) ,
.integral ( vout )
 ) ;

endmodule
```

LPF
I have implemented a very crude discrete time moving average filter with real number signals for the LPF. It may not be the best, but, it does indeed do some filtering. The output of this LPF model is the average of the last some number of samples. The real LPF can be made using opamps, resisters and capacitors.

```
module thee_low_pass_filter
# (
parameter real STEP_SIZE_IN_NS = 1.0 ,
parameter real VMAX = 1.0 ,
parameter real VMIN = -1.0 ,
parameter int TAPS = 4
 )
```

```verilog
 (
input real sig_in ,
output real filtered_out
 ) ;

timeunit 1ns ;
timeprecision 1ps ;

real step ;
real tap_outputs [ TAPS ] ;
real tap_inputs [ TAPS ] ;

generate
 for ( genvar i = 0 ; i < TAPS ; i ++ ) begin
  if ( i == 0 ) begin
    assign tap_inputs [ i ] = sig_in ;
   end else begin
    assign tap_inputs [ i ] = tap_outputs [ i-1 ] ;
   end
 end
 for ( genvar i = 0 ; i < TAPS ; i ++ ) begin
  always begin
    # ( STEP_SIZE_IN_NS ) ;
    tap_outputs [ i ] = tap_inputs [ i ] ;
   end
 end
endgenerate

always_comb begin
 filtered_out = 0 ;
 foreach ( tap_outputs [ i ] ) begin
  filtered_out += tap_outputs [ i ] ;
 end
 filtered_out /= 1.0 * TAPS ;
end

endmodule
```

VCO
The output of the LPF is supposed to be a steady value indicative of the target frequency needed from the VCO. This voltage signal controls the VCO. I have modeled a linear relationship between the input voltage and the output frequency. You may think of a VCO as a voltage to frequency converter. Real chip VCOs do not have a linear relationship throughout their tuning range. Also, I have modeled the VCO to be always running even if the input voltage is below the minimum tuning voltage. This may not be true. Chip VCOs may stop oscillating if the voltage is too low.

```verilog
module thee_vco
# (
parameter real VMIN = -0.8 ,
parameter real VMAX = + 0.8 ,
parameter real FMIN = 1e9 ,
parameter real FMAX = 2e9 ,
parameter string CLK_GEN_TYPE = "basic"
 )
```

```systemverilog
(
 input real vin ,
 output logic clk
 ) ;

timeunit 1ns ;
timeprecision 0.1ps ;

 realtime half_period , period_in_local_units , period_in_seconds ;
 real freq_in_Hz ;

 generate
 if ( CLK_GEN_TYPE == "basic" ) begin
  : ckgen_basic
  initial begin
   clk = 0 ;
    forever begin
     # ( half_period ) ;
     clk = 0 ;
     # ( half_period ) ;
     clk = 1 ;
    end
  end

  always_comb begin
   if ( vin < VMIN ) begin
    freq_in_Hz = FMIN ;
   end else if ( vin > VMAX ) begin
    freq_in_Hz = FMAX ;
   end else begin
    freq_in_Hz = FMIN + ( vin - VMIN ) * ( FMAX-FMIN ) / ( VMAX-VMIN ) ;
   end
   period_in_seconds = 1.0 / freq_in_Hz ;
   period_in_local_units = period_in_seconds / 1e-9 ;
   half_period = period_in_local_units / 2.0 ;
  end
 end
 endgenerate

endmodule
```

Testbench

The testbench supplies the reference clock, sets the PLL M/N.frac ratio paramters and checks if the output is as expected.

```systemverilog
module tb ;
import thee_utils_pkg :: check_approx_equality ;

localparam INT_DIVISION = 12 ;
localparam FRAC_DIVISION = 10 ;
localparam INT_WIDTH = 4 ;
localparam FRAC_WIDTH = 4 ;
localparam real REF_FREQ = 100e6 ;
```

```
logic clk_ref , clk_vco ;

real fout0 , exp_fout ;
bit result0 , result1 ;
logic rstn ;
logic pll_lock ;

thee_clk_gen_module # ( .FREQ ( REF_FREQ / 1e6 ) ) ref_gen ( .clk ( clk_ref ) ) ;

pll_model # ( .INT_WIDTH ( INT_WIDTH ) , .FRAC_WIDTH ( FRAC_WIDTH ) ) pll
 (
 .clk_ref ,
 .rstn ,
 .en ( 1'b1 ) ,
 .int_div ( INT_DIVISION ) ,
 .frac_div ( FRAC_DIVISION ) ,
 .clkout ( clk_vco ) ,
 .lock ( pll_lock )
 ) ;

thee_clk_freq_meter # ( .MEAS_WINDOW ( 50 ) ) fmeter0 ( .clk ( clk_vco ) , .freq_in_hertz ( fout0 )
) ;

int cnt =0;

initial begin
  repeat ( 2 ) @ ( posedge clk_ref ) ;
  rstn = 0 ;
  $display ( "Resetting..." ) ;
  repeat ( 10 ) @ ( posedge clk_ref ) ;
  rstn = 1 ;
  $display ( "Reset released" ) ;
  repeat ( 40000 ) begin
   if ( cnt%10 == 0 ) $display ( "%d cycles of output clock completed , output frequency %f" , cnt
, fout0 ) ;
   @ ( posedge clk_vco ) ;
   cnt++;
  end
  exp_fout = REF_FREQ * ( INT_DIVISION + 1.0 * FRAC_DIVISION / ( 2 ** FRAC_WIDTH ) ) ;
  $display ( " Clkout frequencies 0 %e , expected %e" , fout0 , exp_fout ) ;
  check_approx_equality ( .inp ( fout0 ) , .expected ( exp_fout ) , .result ( result0 ) ) ;
  if ( result0 == 1 ) begin
   repeat ( 3 ) $display ( "PASS" ) ;
  end else begin
   repeat ( 3 ) $display ( "FAIL" ) ;
  end

  $finish ;
end

endmodule


run -all
Resetting...
Reset released
```

```
 0 cycles of output clock completed , output frequency 1545953466.800649
10 cycles of output clock completed , output frequency 1545953466.800649
20 cycles of output clock completed , output frequency 1545953466.800649
…
39950 cycles of output clock completed , output frequency 1262246951.042493
39960 cycles of output clock completed , output frequency 1267234387.67228
Clkout frequencies 1.267234e+09 , expected 1.262500e+09
PASS
```

An interesting video on Delta Sigma based fractional N PLL is here -

[Delta-Sigma Fractional-N PLL, Sudhakar Pamarti - YouTube](#)

Clock And Data Recovery (CDR)

When going outside a chip and into a PCB the number of wires seriously limits the performance of a data transfer protocol. CDR is a neat trick to reduce the wires needed for communication. It also allows extremely high data rates not achievable using a separate clock signal. In a CDR based data transmission, the clock information is embedded in the data stream. The receiving side extracts the clock from the data. The transmitter aids in the clock extraction from the data stream by coding the data in special ways. The most common way is to guarantee a minimum amount of toggling. The following example assumes non-return to zero (NRZ) signaling with the transmitter ensuring a minimum level of toggling. There is also an implicit assumption that there is always a 101 or 010 pattern occurring frequently. The SV model does not represent a real CDR circuit. It only performs the clock and data recovery function using idiosyncratic features afforded by SV. For example, the single line item of a variable delay in the clock generation is a hard function to realize in circuits. The fork-join is even harder. A CDR takes a data stream as input and outputs clock, data and optionally a lock signal. The data signal output is not shown here. It can be realized by sampling the data input with the negative edge of the recovered clock. Detecting when the recovered clock has settled correctly to sample the incoming data is a difficult task in circuits.

Frequency recovery

For modeling, a crude method is to just wait for some fixed time or some quantity related to time, like, number of data transitions. The idea is that many CDR circuits lock to the incoming data stream within a fixed number of 101 or 010 toggles. The core of the model is the forever loop tracking the minimum period between two data transitions. The minimum period of the data toggles is indicative of the clock frequency. But, using the minimum only will result in the recovered frequency to bottom out the highest instantaneous frequency of the clock. For example, if the clock frequency used by the transmitter is 1GHz, the minimum period will be 1ns. But jitter instantaneously increases or decreases the period by say 50ps. Then the using only the minimum will result in the recovered frequency of 1/950ps, 1.05GHz. To avoid this error, the model is set to incorporate any period that falls within 5% of the minimum period. This way, the 1050ps transition will also be tracked as minimum period and compensate the smaller 950ps transition.

Phase recovery

It is not enough to just recover the frequency of the incoming data stream. Note that we are actually interested in the incoming data. So, we need a clock that has rising or falling edges that occur exactly in between two transitions of the data. This way we can assure the best possible sampling of the incoming data signal at the receiver circuits. Suppose data transitions from 101 at 1,2,3 ns, we want to sample the data at 1.5,2.5 and 3.5ns, so that, the change in the data has had enough time to settle to the full scale voltage or current. The fork-join block implements this feature. Whenever the data changes the recovered clock is reset to zero and allowed to rise after a delay of half period (or best sampling point). The disable fork is needed to stop the other branch of the fork from interfering with the needed branch.

The $display and $monitor statements may be commented out to reduce too much debug messages.

```systemverilog
module cdr_model
 (
input logic data_in ,
output logic clkout ,
output logic lock
 ) ;

timeunit 1ns ;
timeprecision 1ps ;

localparam CDR_LOCKING_WINDOW = 20 ;
localparam realtime NATIVE_CLOCK_PERIOD = 40 ;

realtime period_avg , this_period ;
realtime curr_edge , prev_edge , min_period ;

initial begin
  repeat ( CDR_LOCKING_WINDOW ) @ ( data_in ) ;
  lock = 1 ;
end

initial begin
  prev_edge = $realtime ( ) ;
  min_period = NATIVE_CLOCK_PERIOD ;
  forever @ ( data_in ) begin
   curr_edge = $realtime ( ) ;
   this_period = curr_edge - prev_edge ;
   prev_edge = curr_edge ;
   if ( min_period > this_period ) begin
    $display ( "Shrink : Min period %1.5e" , min_period ) ;
    $display ( "Shrink : This period %1.5e" , this_period ) ;
    min_period = 0.5 * this_period + 0.5 * min_period ;
   end else if ( min_period > 0.95 * this_period && min_period < 1.05 * this_period ) begin
    $display ( "Expand : Min period %1.5e" , min_period ) ;
    $display ( "Expand : This period %1.5e" , this_period ) ;
    min_period = 0.5 * this_period + 0.5 * min_period ;
   end
  end
end
```

```verilog
initial begin
  clkout = 0 ;
  forever begin
    fork
    begin
      # ( min_period / 2.0 ) ;
      clkout = 1 ;
      # ( min_period / 2.0 ) ;
      clkout = 0 ;
    end
    begin
      @ ( data_in ) ;
    end
    join_any
    disable fork ;
    clkout = 0 ;
  end
end

initial
 $monitor ( "Clock %b , data %b , time %t" , clkout , data_in , $realtime ( ) ) ;

endmodule
```

The testbench creates a reference clock that has some jitter. This reference clock is used to drive a data traffic generator that simply outputs the fixed patter 100010001000… The CDR model is expected to lock on to the toggling and identify recover the clock. You can see that the model locks to incoming clock after a few data toggles.

Recovering the data is left as an exercise to the reader.

```verilog
module tb ;
import thee_utils_pkg :: check_approx_equality ;
import thee_utils_pkg :: urand_range_real ;

localparam real REF_FREQ = 100e6 ;
logic clk_ref , clk_vco ;

real fout0 , exp_fout ;
bit result0 , result1 ;
logic rstn ;
logic pll_lock ;
logic data_in ;

thee_clk_gen_module # ( .FREQ ( REF_FREQ / 1e6 ) , .CLK_GEN_TYPE ( "jitter_only" ) ,
.PP_JITTER_PPM ( 10000 ) ) ref_gen ( .clk ( clk_ref ) ) ;

initial begin
  data_in = 0 ;
  forever begin
    repeat ( 1 ) @ ( posedge clk_ref ) ;
    data_in = 1 ;
    repeat ( 3 ) @ ( posedge clk_ref ) ;
```

```verilog
      data_in = 0 ;
    end
  end

  cdr_model cdr
  (
   .data_in ,
   .clkout ( clk_vco ) ,
   .lock ( pll_lock )
  ) ;

  thee_clk_freq_meter # ( .MEAS_WINDOW ( 50 ) ) fmeter0 ( .clk ( clk_vco ) , .freq_in_hertz ( fout0 )
  ) ;

  initial begin
    repeat ( 2 ) @ ( posedge clk_ref ) ;
    rstn = 0 ;
    repeat ( 10 ) @ ( posedge clk_ref ) ;
    rstn = 1 ;

    repeat ( 500 ) @ ( posedge clk_vco ) ;

    exp_fout = REF_FREQ ;
    $display ( " Clkout frequencies %1.3e , expected %1.3e" , fout0 , exp_fout ) ;
    check_approx_equality ( .inp ( fout0 ) , .expected ( exp_fout ) , .result ( result0 ) ) ;
    if ( result0 == 1 ) begin
      repeat ( 3 ) $display ( "PASS" ) ;
    end else begin
      repeat ( 3 ) $display ( "FAIL" ) ;
    end

    $finish ;
  end

endmodule
```

```
run -all
Shrink : Min period 4.00000e+01
Shrink : This period 0.00000e+00
Clock 0 , data 0 , time                0
Shrink : Min period 2.00000e+01
Shrink : This period 9.97000e+00
Clock 0 , data 1 , time             9970
Clock 1 , data 1 , time            17462
Clock 0 , data 1 , time            24954
Clock 1 , data 1 , time            32446
Clock 0 , data 1 , time            39938
Clock 0 , data 0 , time            39968
Clock 1 , data 0 , time            47460
Shrink : Min period 1.49850e+01
Shrink : This period 9.91000e+00
Clock 0 , data 1 , time            49878
```

```
Clock 1 , data 1 , time          56101
Clock 0 , data 1 , time          62324
Clock 1 , data 1 , time          68547
Clock 0 , data 1 , time          74770
Clock 0 , data 0 , time          79880
Clock 1 , data 0 , time          86103
Shrink : Min period 1.24475e+01
Shrink : This period 9.91600e+00
Clock 0 , data 1 , time          89796
Clock 1 , data 1 , time          95386
Clock 0 , data 1 , time         100976
Clock 1 , data 1 , time         106566
Clock 0 , data 1 , time         112156
Clock 1 , data 1 , time         117746
Clock 0 , data 0 , time         119810
Clock 1 , data 0 , time         125400
Shrink : Min period 1.11818e+01
Shrink : This period 1.00320e+01
 Clkout frequencies 1.003e+08 , expected 1.000e+08
PASS
```

UVM Hello World

Unified Verification Methodology as mentioned before in the book is the industry standard for reusable and efficient verification of complex chips. But to get to the UVM fairyland it takes an unbelievable amount of training. A large part of the problem is where to start. In this example, I hope to break the ice on UVM by letting the user do the classic "hello worlds". The first thing to notice is the import of uvm_pkg and the uvm_macros. Unlike your own packages that you code and include in the simulation files list, UVM is a library and most simulators automatically include these when you turn on an appropriate compilation switch. For Xilinx Vivado, it is "-L uvm". The task uvm_report_info is the one doing the actual printing. You may also notice that when using UVM, the simulator prints a copyright message with the names of the semiconductor companies – NVIDIA, Cypress, Synopsys, Cadence and Mentor. This is because UVM is a standard that was created from contributions from so many people.

```systemverilog
`include "uvm_macros.svh"
module tb ;

initial begin
   import uvm_pkg :: * ;
   uvm_report_info ( "<Your message prefix here>" , "Hello World" , UVM_LOW ) ;
end

endmodule
```

Compilation command shown for emphasis for UVM
```
xvlog --incr --sv -L uvm --work work -f sim.f
```

```
All Run commands are in run.sh in the same example directory
./run.sh
```

Compilation
INFO: [VRFC 10-2263] Analyzing SystemVerilog file "<my example dir>/examples/uvm_hello_world/tb.sv" into library work
INFO: [VRFC 10-311] analyzing module tb

Elaboration
Vivado Simulator 2019.2
Copyright 1986-1999, 2001-2019 Xilinx, Inc. All Rights Reserved.
Running: <Vivado install dir>/Vivado/2019.2/bin/unwrapped/lnx64.o/xelab --snapshot work tb
Multi-threading is on. Using 2 slave threads.
Starting static elaboration
Pass Through NonSizing Optimizer
Completed static elaboration
Starting simulation data flow analysis
Completed simulation data flow analysis
Time Resolution for simulation is 1ps
Compiling package uvm.uvm_pkg
Compiling package std.std
Compiling module work.tb
Built simulation snapshot work

Simulation
run -all
UVM_INFO   /proj/xbuilds/SWIP/2019.2_0924_1936/installs/lin64/Vivado/2019.2/data/system_verilog/uvm_1.2/
xlnx_uvm_package.sv(18601) @ 0: reporter [UVM/RELNOTES]
    (Specify +UVM_NO_RELNOTES to turn off this notice)
 with `UVM_OBJECT_DO_NOT_NEED_CONSTRUCTOR undefined.

 You are using a version of the UVM library that has been compiled
 with `UVM_NO_DEPRECATED undefined.

 You are using a version of the UVM library that has been compiled

 ***********   IMPORTANT RELEASE NOTES   ************

(C) 2013-2014 NVIDIA Corporation
(C) 2011-2013 Cypress Semiconductor Corp.
(C) 2006-2014 Synopsys, Inc.
(C) 2007-2014 Cadence Design Systems, Inc.
(C) 2007-2014 Mentor Graphics Corporation

UVM_INFO @ 0: reporter [<Your message prefix here>] Hello World
exit
INFO: [Common 17-206] Exiting xsim at Fri Feb 14 14:03:02 2020...

High Level Synthesis

Add Two Numbers

High Level Synthesis promises a never before seen level of designer productivity. I am learning Xilinx Vivado HLS tool as I write this book. I found that even the simplest example available with the tool is quite daunting for me. I think it will be overwhelming to many readers too. So, I have a super simple example, just add two numbers. To my pleasant surprise, it just worked after I fixed a few syntax errors! Wow! My appreciation for all the software engineers in Xilinx and other computer engineers who have worked on the HLS technology. This is no simple feat. Lets now dive into the details of HLS. If you have some background already you may like this open book, "Parallel Programming for FPGAs Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer, 2018-05-11"
https://arxiv.org/pdf/1805.03648.pdf

There are two parts to a HLS design – the design itself in C/C++/SystemC/OpenCL and then a set of directives that guide the HLS tool to convert the software code into an RTL that meets the desired hardware targets for power-performance-area. To test the design, you need a main function in C that calls the design software function. You need to look at the values returned from the design C function and create a result value in the main function. Vivado tool takes a return value of 0 as pass and any other value as fail.

Design in C – add_two_numbers.c

```c
int add_two_numbers ( int a, int b) {
#pragma HLS interface ap_ctrl_none port=return
   return (a+b);
}
```

Prototype header file add_two_numbers.h

```c
int add_two_numbers ( int , int ) ;
```

Testbench in C

```c
#include <stdio.h>
#include <stdlib.h>
#include "add_two_numbers.h"
int main (void) {
   int result=0;
   int out;
   out = add_two_numbers ( 10, 20);
   if ( out != 30 )
      result = 1;
   else
      printf("Test Vector 0 Passed \n");

   out = add_two_numbers ( 22, 32);
   if ( out != 54 )
      result = 1;
```

```
    else
        printf("Test Vector 1 Passed \n");
    return result;
}
```

Compiling and Testing the design in C

Lets start the HLS venture by adding the add_two_numbers.c file to the project. Use the menu options Project > Add Sources and then select the add_two_numbers.c file. It is a general practice in C based designs to have a file that defines a prototype of the function used in the C code file. The prototype is included in the other files that call the function. Add the add_two_numbers.h header file to the sources using the same menu options. Savvy users would have by now done a both includes in one step by selecting both files in the dialog box! Use Project > Add Testbench to include main.c testbench to the project. The main.c contains code that calls the C design for verifying its functionality. Use the "Project > Run C simulation" menu option to compile and test the C based design. Once you are satisfied with the level of checking you can move on to synthesizing an RTL model of the C code.

C to RTL HLS
This step is the much hyped High-Level-Synthesis! It really does the magic of creating an RTL from high level language code. Use the menu option "Solution > Run C Synthesis > Active Solution". In the language of Vivado HLS tool, a solution means one RTL implementation of the C design. The implementation includes the directives used to guide the creation of RTL code and all the associated reports and synthesized RTL file in VHDL, Verilog and SystemC formats.

Setting the FPGA target (Optional)
Ignore this step if you don't get any error during the sysnthesis step. If you get any error saying a particular device is not supported you may have to set the target FPGA to the one you have downloaded the files for. In my case, I had downloaded files for only the Artix-7 family of devices, but, the Vivado HLS Example I was trying was Virtex-7. So, I got the error. To avoid the error, you can also download files for all Xilinx devices.

Exported Verilog
Lets take a moment to study the exported Verilog RTL file. You can see that the output ap_return is the sum of inputs a and b. There are many other ports with prefix ap_ that we will ignore for now. These control how this adder logic is used at the next level of hierarchy. There are many options. The simplest one is continuous operation outputting the sum of inputs without any synchronization signal. Another option is to wait for a handshake signal from the adder to say that the operation is complete. The ap_start and ap_ready achieve the handshake behavior.

```
//
=================================================================
===
// RTL generated by Vivado(TM) HLS - High-Level Synthesis from C, C++ and SystemC
// Version: 2019.1
// Copyright (C) 1986-2019 Xilinx, Inc. All Rights Reserved.
//
//
=================================================================
```

```verilog
`timescale 1 ns / 1 ps

(* CORE_GENERATION_INFO="add_two_numbers,hls_ip_2019_1,
{HLS_INPUT_TYPE=c,HLS_INPUT_FLOAT=0,HLS_INPUT_FIXED=0,HLS_INPUT_PART=xc7a12ti-
csg325-
1L,HLS_INPUT_CLOCK=10.000000,HLS_INPUT_ARCH=others,HLS_SYN_CLOCK=2.702000,HLS_SYN_
LAT=0,HLS_SYN_TPT=none,HLS_SYN_MEM=0,HLS_SYN_DSP=0,HLS_SYN_FF=0,HLS_SYN_LUT=39,H
LS_VERSION=2019_1}" *)

module add_two_numbers (
    ap_start,
    ap_done,
    ap_idle,
    ap_ready,
    a,
    b,
    ap_return
);


input   ap_start;
output   ap_done;
output   ap_idle;
output   ap_ready;
input [31:0] a;
input [31:0] b;
output [31:0] ap_return;

assign ap_done = ap_start;

assign ap_idle = 1'b1;

assign ap_ready = ap_start;

assign ap_return = (b + a);

endmodule //add_two_numbers
```

C/RTL co-simulation

Use the "Solution > Run C/RTL Cosimulation" menu option to run a simulation of the synthesized RTL. The cleverness or the utility of HLS is in this step of reusing the C based testbench to check the RTL so that the designer is spared the effort of writing a separate verification test suite. You can check the waveforms using "Solution > Open Waveviewer". I also regularly print my own pass and fail messages just as a sanity check of the tool.

```
Log from the simulation
INFO: [COSIM 212-316] Starting C post checking ...
Test Vector 0 Passed
Test Vector 1 Passed
INFO: [COSIM 212-1000] *** C/RTL co-simulation finished: PASS ***
INFO: [COSIM 212-210] Design is translated to an combinational logic. II and
Latency will be marked as all 0.
Finished C/RTL cosimulation.
```

Viewing the result

You may use "Solution > Open Report" and study the area and speed results obtained by simulation. I had set the timing constraints to target 100MHz frequency or 10ns clock period. The timing report says that the synthesized RTL is expected to work at 2.7ns period or 370MHz. The area used is 39 LUT. A typical logic gate generally takes about 1 LUT in FPGA. In this case if the addition is implemented as a ripple carry adder with one full adder per bit, then we expect 32 full adders or 32 LUTs. The additional LUTs may be for the extra ap_* port signals. The particular Artix device has about 8000 LUTs available for usage and this design takes only 39 or approximately 0% of the resource. You can see that latency in terms of clock cycles is 0 because the RTL is implemented as a combinational logic of the add operator.

```
================================================================
== Vivado HLS Report for 'add_two_numbers'
================================================================
* Date:

* Version:       2019.1 (Build 2552052 on Fri May 24 15:28:33 MDT 2019)
* Project:       add_two_numbers
* Solution:      solution1
* Product family: artix7
* Target device:  xc7a12ti-csg325-1L


================================================================
== Performance Estimates
================================================================
+ Timing (ns):
    * Summary:
    +--------+-------+----------+------------+
    |  Clock | Target| Estimated| Uncertainty|
    +--------+-------+----------+------------+
    |ap_clk  | 10.00|     2.702|        1.25|
    +--------+-------+----------+------------+

+ Latency (clock cycles):
    * Summary:
    +-----+-----+-----+-----+---------+
    |  Latency  |  Interval | Pipeline|
    | min | max | min | max |   Type  |
    +-----+-----+-----+-----+---------+
    |    0|    0|    0|    0|   none  |
    +-----+-----+-----+-----+---------+

    + Detail:

================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-------+-------+------+-----+
|       Name      | BRAM_18K| DSP48E|   FF  |  LUT | URAM|
+-----------------+---------+-------+-------+------+-----+
```

```
|DSP              |          -|       -|        -|       -|     -|
|Expression       |          -|       -|        0|      39|     -|
|FIFO             |          -|       -|        -|       -|     -|
|Instance         |          -|       -|        -|       -|     -|
|Memory           |          -|       -|        -|       -|     -|
|Multiplexer      |          -|       -|        -|       -|     -|
|Register         |          -|       -|        -|       -|     -|
+-----------------+-----------+--------+---------+--------+------+
|Total            |          0|       0|        0|      39|    0|
+-----------------+-----------+--------+---------+--------+------+
|Available        |         40|      40|    16000|    8000|    0|
+-----------------+-----------+--------+---------+--------+------+
|Utilization (%)  |          0|       0|        0|      ~0 |    0|
+-----------------+-----------+--------+---------+--------+------+

+ Detail:

    * Expression:
    +-------------+----------+-------+---+----+------------+------------+
    | Variable Name| Operation| DSP48E| FF| LUT| Bitwidth P0| Bitwidth P1|
    +-------------+----------+-------+---+----+------------+------------+
    |ap_return    |    +     |      0|  0|  39|          32|          32|
    +-------------+----------+-------+---+----+------------+------------+
    |Total        |          |      0|  0|  39|          32|          32|
    +-------------+----------+-------+---+----+------------+------------+
```

Suppose you don't like the additional ap_* signals, how do you remove them? Vivado HLS allows you to control the C to RTL synthesis step. If you add the pragma ap_ctrl_none to the return port the unwanted control signals go away. A pragma, as I understand, is a way of providing additional information in the same C code that is not natively used by the C compiler but is meant for some other purpose. Here, it is meant for the Vivado HLS tool.

C code with pragma

Note that even if there is not a specific named function argument called return, the VHLS refers to a function return value by the name return. So, to control the behavior of that port set pragmas with "port=return".

```c
int add_two_numbers ( int a, int b) {
#pragma HLS interface ap_ctrl_none port=return
  return (a+b);
}
```

Synthesized Verilog after adding the pragma (note absence of ap_* ports)

```verilog
module add_two_numbers (
    a,
    b,
    ap_return
);


input [31:0] a;
```

```verilog
input [31:0] b;
output [31:0] ap_return;

assign ap_return = (b + a);

endmodule //add_two_numbers
```

Moving Average

Moving average is a simple filter that suppresses high frequencies. The logic is simple, save N samples of the input and output the average of the samples. The static keyword for data_buf array means that the values in the array are maintained between function calls. For HLS purpose, think of static to mean flops that need to maintain values between function calls.

moving_average.h

```c
#include <stdio.h>
#define AVG_LENGTH 2
int moving_average ( int data_in) ;
```

moving_average.c

```c
#include "moving_average.h"
int moving_average ( int data_in ) {
   int i,sum;
   static int data_buf[AVG_LENGTH];

   for ( i = 0, sum = 0 ; i < AVG_LENGTH ;i++) {
      sum+=data_buf[i];
      printf("Data buffer of index %d is %d \n, sum is %d \n",i, data_buf[i],sum );
   }
   for ( i = AVG_LENGTH-1 ; i > 0 ;i--) {
      data_buf [i] = data_buf[i-1];
   }
   data_buf[0]=data_in;

   return (sum / AVG_LENGTH );
}
```

Before trying to synthesize C code to RTL, we need to check if it is working properly first. Unfortunately, my Vivado installation got corrupted and the C compiler does not work. But no problem, we are in the Linux world! Lets use GCC to test the C code.

run_c.sh
```bash
gcc main.c moving_average.c
./a.out
```

<ehgu git dir>/dsn_verif/hls/moving_average> ./run_c.sh
Data buffer of index 0 is 10
, sum is 10
Data buffer of index 1 is 0
, sum is 10

Data buffer of index 2 is 0
, sum is 10
Data buffer of index 3 is 0
, sum is 10
Test Vector 0 Passed
Data buffer of index 0 is 10
, sum is 10
Data buffer of index 1 is 10
, sum is 20
Data buffer of index 2 is 0
, sum is 20
Data buffer of index 3 is 0
, sum is 20

Using the GUIs sometimes slow down the development process, so for this example, I am switching to command line. Open the interactive command line of Vivado as shown and source the VHLS run scripts.
<Xilinx install dir>/2019.2/bin/vivado_hls -i

vivado_hls> source hls.tcl

hls.tcl

```
open_project moving_average
set_top moving_average
add_files moving_average.c
add_files -tb main.c
open_solution "solution1"
set_part {xc7a75ti-ftg256-1L}
create_clock -period 10 -name default
config_sdx -optimization_level none -target none
config_export -vivado_optimization_level 2
set_clock_uncertainty 12.5%
#source "./moving_average/solution1/directives.tcl"
#csim_design
csynth_design
#cosim_design
#export_design -format ip_catalog
```

No directives have been set for HLS in the script. Notice that the csim_design and cosim_design are commented out because my Vivado C compiler setup is broken. We substituded the csim_design with GCC. The cosim_design is a big value add in HLS, in that it reduces the time needed to verify the exported Verilog design. Without the support of cosim_design, I built my own SV testbench in the old fashioned way. Along the way, I have learnt the ap_hs protocol of VHLS well. When there are no directives set on the IO ports, VHLS uses the default ap_hs protocol. The protocol has a combination of start, ready, idle, done signals that together manage how inputs are passed to the hardware. The mantra to remember is that "one done is one execution of C function". The SV testbench and run messages are shown next. You can notice that in the testbench the expected data is delayed by one more cycle to

match the latency of the HL synthesized Verilog RTL. The first output we are tracking is (23+25)/2 = 24 which then moves to (25+34)/2=29 and so on.

```systemverilog
module tb ;
import thee_utils_pkg :: * ;

parameter DWIDTH = 32 ;
parameter TAPS = 2;
parameter IGNORE_FIRST_N = 2;

bit clk ;
logic rstn ;
logic [ DWIDTH-1 : 0 ] data_in ;
logic [ DWIDTH-1 : 0 ] data_out , expected_data, expected_data_d;
logic [ DWIDTH-1 : 0 ] filter [ $ ] ;
logic start, done, idle, ready;
int cnt=0;
int data_cnt=0;
logic result;

thee_clk_gen_module #(.FREQ(100)) clk_gen_i0 ( .clk ( clk ) ) ;
bit first_cycle;
initial begin
  repeat (TAPS) filter.push_front(data_in);
  forever @(posedge clk )
    if (data_cnt< 1) begin
      start =  1;
      if ( !ready && !idle ) begin
        data_in = $urandom_range(200) ;
        filter.push_front(data_in); filter.pop_back();
        //$display("Data %d, Data number %d",data_in, data_cnt);
        data_cnt++;
      end else data_in = 0;
    end else if ( data_cnt >=1 ) begin
      start = 0; //hold data in as is data_in = 0;
      if ( ready ) begin
        data_cnt = 0 ;
      end
    end
end

always @(posedge clk )
  $display("Wave : Start %b, ready %b, idle %b, done %b, data_in %d", start, ready, idle, done,
data_in);

initial begin
  result = 1'bx ;
  start = 0 ;
  toggle_rstn ( .rstn ( rstn ) ) ;
  repeat ( 1 ) @ ( posedge clk ) ;
  start = 1 ;

  for ( int i = 0 ; i < 30*TAPS ; i ++ ) begin
    repeat ( 1 ) @ ( posedge clk ) ;
    if ( done ) begin
```

```systemverilog
      expected_data_d = expected_data;
      expected_data = filter.sum()/TAPS;
      $display("Start %b, ready %b, idle %b, done %b", start, ready, idle, done);
      if ( cnt < IGNORE_FIRST_N ) begin
        cnt++;
        continue;
      end
      cnt++;
      if ( data_out === expected_data_d ) begin
        $display ( "P - output data %d expected data %d" , data_out , expected_data_d ) ;
        if ( result !== 0 ) result = 1;
      end else begin
        $display ( "F - output data %d expected data %d" , data_out , expected_data_d ) ;
        foreach(filter[i]) begin
          $write("Filter data %d ",filter[i]);
        end
        $display();
        result = 0 ;
      end
    end
  end

  print_test_result ( result ) ;
  $finish ;
end

moving_average ma (
 .ap_clk(clk),
 .ap_rst(~rstn),
 .ap_start(start),
 .ap_done(done),
 .ap_idle(idle),
 .ap_ready(ready),
 .data_in,
 .ap_return(data_out)
);

endmodule

run -all
Wave : Start 1, ready 0, idle 1, done 0, data_in          0
Wave : Start 1, ready 0, idle 0, done 0, data_in         23
Wave : Start 0, ready 0, idle 0, done 0, data_in         23
Wave : Start 0, ready 0, idle 0, done 0, data_in         23
Wave : Start 0, ready 1, idle 0, done 1, data_in         23
Start 0, ready 1, idle 0, done 1
Wave : Start 1, ready 0, idle 1, done 0, data_in          0
Wave : Start 1, ready 0, idle 0, done 0, data_in         25
Wave : Start 0, ready 0, idle 0, done 0, data_in         25
Wave : Start 0, ready 0, idle 0, done 0, data_in         25
Wave : Start 0, ready 1, idle 0, done 1, data_in         25
Start 0, ready 1, idle 0, done 1
Wave : Start 1, ready 0, idle 1, done 0, data_in          0
Wave : Start 1, ready 0, idle 0, done 0, data_in         34
Wave : Start 0, ready 0, idle 0, done 0, data_in         34
Wave : Start 0, ready 0, idle 0, done 0, data_in         34
```

```
Wave : Start 0, ready 1, idle 0, done 1, data_in          34
Start 0, ready 1, idle 0, done 1
P - output data          24 expected data          24
Wave : Start 1, ready 0, idle 1, done 0, data_in           0
Wave : Start 1, ready 0, idle 0, done 0, data_in         120
Wave : Start 0, ready 0, idle 0, done 0, data_in         120
Wave : Start 0, ready 0, idle 0, done 0, data_in         120
Wave : Start 0, ready 1, idle 0, done 1, data_in         120
Start 0, ready 1, idle 0, done 1
P - output data          29 expected data          29
...
```

Exercise:
The exported Verilog uses "initial" SV statement to initialize variables. The initial construct is not allowed in ASIC RTL because it cannot be synthesized. Fix the initial problem by adding appropriate directives in in the directives.tcl file.

Mean and Variance

Moving average filter example returns the result as the return variable in C. What if we need to return many more outputs? Say, even an array. Now what? Fortunately, VHLS comes with a framework to code the array outputs in C. You just use a pointer argument to the function. For HLS, an array argument is internally mapped to a memory that is assumed to exist outside the design. In SV we need a fixed size for the address signal. So, VHLS mandates setting the size of the array in the function call. This also helps VHLS to use appropriate sized logic inside the RTL implementation. There are two outputs, mean and variance to be outputted on the data_out argument. The function is fairly simple, the first part calculates the average of input data and the second loop calculates the variance (deviation from the average) of the input data from the average. At the end, the mean is read out as the first value of the data_out array and the variance is read out as the second value of the array.

mean_variance.h

```c
#include <stdio.h>
#include <math.h>
#define AVG_LENGTH 4
void mean_variance ( int *data_in, int *data_out) ;
```

mean_variance.c

```c
#include "mean_variance.h"
void mean_variance ( int  data_in[AVG_LENGTH], int data_out[2] ) {
  int i,sum,mean,variance;
  int data_buf[AVG_LENGTH];

  for ( i = 0 ; i < AVG_LENGTH ;i++) {
    data_buf[i] = *data_in++;
    }
  for ( i = 0, sum = 0 ; i < AVG_LENGTH ;i++) {
    sum+=data_buf[i];
    printf("Data buffer of index %d is %d \n, sum is %d \n",i, data_buf[i],sum );
  }
    mean = (sum / AVG_LENGTH );
```

```
    for ( i = 0, sum = 0 ; i < AVG_LENGTH ;i++) {
        sum+=(data_buf[i]-mean)*(data_buf[i]-mean);
        printf("Data buffer of index %d is %d \n, sum is %d \n",i, data_buf[i],sum );
    }
    variance = sum/AVG_LENGTH ;
    *data_out++ = mean;
    *data_out = variance;
}
```

C Testbench
The C main function supplies the input data array of 100, 200, 100, 200 and looks for the expected mean and variance of 150 and 2500. The variance apprently has two flavors in math – sample and population. I used https://www.wolframalpha.com/ to calculate the result by typing this on the search bar -
variance{100,200,100,200} it gave me 3333.3 which was sample variance. The code here is for population variance which is rightly 2500.
Wolfram Alpha may be very useful to you as an advanced Google Search for math questions. Try it and you will love it. The free features may be sufficient for simple calculations. If you sufficient money you can consider purchasing a subscription.

```
#include <stdio.h>
#include <stdlib.h>
#include "mean_variance.h"
int main (void) {
    int result=0;
    int out;
    int sum;
    int buf[AVG_LENGTH];
    int m_stdd[2];
    int mean, stdd;
    int exp[2];
    int i,j ;
        j =0;
    for(i=0;i<AVG_LENGTH;i++)
        buf[i]=(i%2==0)?100:200;
    mean_variance ( buf, m_stdd );
    exp[0] = 150 ; exp[1] = 2500;
    if (( m_stdd[0] != exp[0]) || ( m_stdd[1] != exp[1])  ) {
        result = 1;
        printf("Test Vector %d Failed. Expected %d %d , got %d %d\n",j, exp[0],exp[1],
m_stdd[0],m_stdd[1]);
    }
    else {
        printf("Test Vector Passed \n");
    }

    return result;
}
```

./run_c.sh
```
Data buffer of index 0 is 100
, sum is 100
Data buffer of index 1 is 200
```

```
, sum is 300
Data buffer of index 2 is 100
, sum is 400
Data buffer of index 3 is 200
, sum is 600
Data buffer of index 0 is 100
, sum is 2500
Data buffer of index 1 is 200
, sum is 5000
Data buffer of index 2 is 100
, sum is 7500
Data buffer of index 3 is 200
, sum is 10000
Test Vector Passed
```

The synthesis script hls.tcl is similar to earlier scripts, so, I am not listing it here.

The SV testbench took me sometime to get it right. Now, I fully realize the value of HLS for verifying at C level. The testbench supplies random data to the HLS converted Verilog module of mean variance. Simultaneously, a data buffer local to the testbench is populated with the inputs going into the Verilog DUT (Device Under Test). The DUT samples input data when CE (chip enable) is high. For some reason I dont know, VHLS has created two data output buses dout0 and dout1 to output the mean and variance separately. At the end this two outputs are checked once the DUT asserts done signal. The expected mean and variance uses the sum() method available to all arrays in SV. You can also see that the data buffer is overwritted with the deviation from mean values because the older sample values are no longer needed after calculating the mean. There is also a show buffer function that is used for debug purpose.

```verilog
module tb ;
import thee_utils_pkg :: * ;

parameter DWIDTH = 32 ;
parameter TAPS = 4;
parameter NUM_VECTORS=3;

bit clk ;
logic rstn ;
logic [ DWIDTH-1 : 0 ] data_in,data_in_dly ;
logic [ DWIDTH-1 : 0 ] expected_data[2];
logic [ DWIDTH-1 : 0 ] data_buf [TAPS];
logic start, done, idle, ready;
int cnt=0;
int data_cnt=0;
logic result;

logic [1:0] data_in_address0;
logic  data_in_ce0, data_in_ce0_d;
logic [31:0] data_in_q0;
logic [0:0] data_out_address0;
logic  data_out_ce0;
logic  data_out_we0;
logic [31:0] data_out_d0;
logic [0:0] data_out_address1;
```

```systemverilog
logic   data_out_ce1;
logic   data_out_we1;
logic [31:0] data_out_d1;

thee_clk_gen_module #(.FREQ(100)) clk_gen_i0 ( .clk ( clk ) ) ;
bit first_cycle;

initial begin
  data_in=0;
  forever @(posedge clk )
    if ( data_in_ce0 )
      data_in = $urandom_range(100) ;
end

initial begin
  data_in=0; data_in_ce0_d= 0 ;
  forever @(posedge clk ) begin
    data_in_ce0_d = data_in_ce0;
    if (data_cnt< 4) begin
      start =  1;
      if ( !ready && !idle ) begin
       if (data_in_ce0 ) begin
         data_buf[data_cnt]=data_in;
         data_cnt++;
        end
      end
    end else if ( data_cnt >=1 ) begin
      start = 0; //hold data in as is data_in = 0;
      if ( ready ) begin
       data_cnt = 0 ;
      end
    end
  end
end

always @(posedge clk )
  $display("Wave : Strt %b, rdy %b, idle %b, done %b, din %4d, din CE %b, dout0 %4d, dout0ce
%b, dout0we %b, dout0adr %d, dout1 %d, dout1ce %b d1we %b, d1addr %d ", start, ready, idle,
done, data_in,data_in_ce0, data_out_d0, data_out_ce0, data_out_we0, data_out_address0,
data_out_d1, data_out_ce1,data_out_we1, data_out_address1);

initial begin
  result = 1'bx ;
  start = 0 ;
  toggle_rstn ( .rstn ( rstn ) ) ;
  repeat ( 1 ) @ ( posedge clk ) ;
  start = 1 ;

  for ( int i = 0 ; i < NUM_VECTORS ;) begin
   repeat ( 1 ) @ ( posedge clk ) ;
   if ( done ) begin
     i ++ ;
     expected_data[0] = data_buf.sum()/TAPS;
     show_buf();
     foreach(data_buf[i]) begin
```

```verilog
        data_buf[i] = (data_buf[i] -expected_data[0])**2;
      end
      show_buf();
      expected_data[1] = data_buf.sum()/TAPS;
      cnt++;
      if ( (data_out_d0 === expected_data[0]) && (data_out_d1 === expected_data[1]) ) begin
        $display ( "P - output data %d output data %d expected data %d expected data %d" ,
data_out_d0, data_out_d1 , expected_data[0], expected_data[1]) ;
        if ( result !== 0 ) result = 1;
      end else begin
        $display ( "F - output data %d output data %d expected data %d expected data %d" ,
data_out_d0 , data_out_d1, expected_data[0], expected_data[1] ) ;
        result = 0 ;
      end
    end
  end

  print_test_result ( result ) ;
  $finish ;
end

mean_variance mv (
  .ap_clk(clk),
  .ap_rst(~rstn),
  .ap_start(start),
  .ap_done(done),
  .ap_idle(idle),
  .ap_ready(ready),
  .data_in_address0,
  .data_in_ce0,
  .data_in_q0(data_in),
  .data_out_address0,
  .data_out_ce0,
  .data_out_we0,
  .data_out_d0,
  .data_out_address1,
  .data_out_ce1,
  .data_out_we1,
  .data_out_d1
);

task automatic show_buf;
  $display("Buffer data");
  foreach(data_buf[i]) begin
    $write("%d ",data_buf[i]);
  end
  $display();
endtask : show_buf

endmodule


run -all
Wave : Strt 1, rdy 0, idle 1, done 0, din    0, din CE 0, dout0 x, dout0ce 0, dout0we 0, dout0adr
0, dout1         X, dout1ce 0 d1we 0, d1addr 1
Wave : Strt 1, rdy 0, idle 0, done 0, din    0, din CE 0, dout0 x, dout0ce 0, dout0we 0, dout0adr
0, dout1         X, dout1ce 0 d1we 0, d1addr 1
```

```
Wave : Strt 1, rdy 0, idle 0, done 0, din   43, din CE 1, dout0 x, dout0ce 0, dout0we 0, dout0adr
0, dout1        X, dout1ce 0 d1we 0, d1addr 1
Wave : Strt 1, rdy 0, idle 0, done 0, din   43, din CE 0, dout0 x, dout0ce 0, dout0we 0, dout0adr
0, dout1        X, dout1ce 0 d1we 0, d1addr 1
Wave : Strt 1, rdy 0, idle 0, done 0, din   15, din CE 1, dout0 x, dout0ce 0, dout0we 0, dout0adr
0, dout1        X, dout1ce 0 d1we 0, d1addr 1
Wave : Strt 1, rdy 0, idle 0, done 0, din   15, din CE 0, dout0 x, dout0ce 0, dout0we 0, dout0adr
0, dout1        X, dout1ce 0 d1we 0, d1addr 1
Wave : Strt 1, rdy 0, idle 0, done 0, din   41, din CE 1, dout0 x, dout0ce 0, dout0we 0, dout0adr
0, dout1        X, dout1ce 0 d1we 0, d1addr 1
Wave : Strt 1, rdy 0, idle 0, done 0, din   41, din CE 0, dout0 x, dout0ce 0, dout0we 0, dout0adr
0, dout1        X, dout1ce 0 d1we 0, d1addr 1
Wave : Strt 1, rdy 0, idle 0, done 0, din    7, din CE 1, dout0 x, dout0ce 0, dout0we 0, dout0adr
0, dout1        X, dout1ce 0 d1we 0, d1addr 1
Wave : Strt 0, rdy 0, idle 0, done 0, din    7, din CE 0, dout0 x, dout0ce 0, dout0we 0, dout0adr
0, dout1        X, dout1ce 0 d1we 0, d1addr 1
Wave : Strt 0, rdy 0, idle 0, done 0, din   44, din CE 1, dout0 x, dout0ce 0, dout0we 0, dout0adr
0, dout1        X, dout1ce 0 d1we 0, d1addr 1
Wave : Strt 0, rdy 0, idle 0, done 0, din   44, din CE 0, dout0 x, dout0ce 0, dout0we 0, dout0adr
0, dout1        X, dout1ce 0 d1we 0, d1addr 1
Wave : Strt 0, rdy 0, idle 0, done 0, din   44, din CE 0, dout0 x, dout0ce 0, dout0we 0, dout0adr
0, dout1        X, dout1ce 0 d1we 0, d1addr 1
Wave : Strt 0, rdy 0, idle 0, done 0, din   44, din CE 0, dout0 x, dout0ce 0, dout0we 0, dout0adr
0, dout1        X, dout1ce 0 d1we 0, d1addr 1
Wave : Strt 0, rdy 0, idle 0, done 0, din   44, din CE 0, dout0 x, dout0ce 0, dout0we 0, dout0adr
0, dout1        X, dout1ce 0 d1we 0, d1addr 1
Wave : Strt 0, rdy 0, idle 0, done 0, din   44, din CE 0, dout0 x, dout0ce 0, dout0we 0, dout0adr
0, dout1        X, dout1ce 0 d1we 0, d1addr 1
Wave : Strt 0, rdy 0, idle 0, done 0, din   44, din CE 0, dout0   26, dout0ce 1, dout0we 0,
dout0adr 0, dout1        0, dout1ce 1 d1we 0, d1addr 1
Wave : Strt 0, rdy 0, idle 0, done 0, din   44, din CE 0, dout0   26, dout0ce 0, dout0we 0,
dout0adr 0, dout1        0, dout1ce 0 d1we 0, d1addr 1
Wave : Strt 0, rdy 0, idle 0, done 0, din   44, din CE 0, dout0   26, dout0ce 0, dout0we 0,
dout0adr 0, dout1        0, dout1ce 0 d1we 0, d1addr 1
Wave : Strt 0, rdy 0, idle 0, done 0, din   44, din CE 0, dout0   26, dout0ce 1, dout0we 0,
dout0adr 0, dout1       72, dout1ce 1 d1we 0, d1addr 1
Wave : Strt 0, rdy 0, idle 0, done 0, din   44, din CE 0, dout0   26, dout0ce 0, dout0we 0,
dout0adr 0, dout1       72, dout1ce 0 d1we 0, d1addr 1
Wave : Strt 0, rdy 0, idle 0, done 0, din   44, din CE 0, dout0   26, dout0ce 0, dout0we 0,
dout0adr 0, dout1       72, dout1ce 0 d1we 0, d1addr 1
Wave : Strt 0, rdy 0, idle 0, done 0, din   44, din CE 0, dout0   26, dout0ce 1, dout0we 0,
dout0adr 0, dout1      102, dout1ce 1 d1we 0, d1addr 1
Wave : Strt 0, rdy 0, idle 0, done 0, din   44, din CE 0, dout0   26, dout0ce 0, dout0we 0,
dout0adr 0, dout1      102, dout1ce 0 d1we 0, d1addr 1
Wave : Strt 0, rdy 0, idle 0, done 0, din   44, din CE 0, dout0   26, dout0ce 0, dout0we 0,
dout0adr 0, dout1      102, dout1ce 0 d1we 0, d1addr 1
Wave : Strt 0, rdy 0, idle 0, done 0, din   44, din CE 0, dout0   26, dout0ce 1, dout0we 0,
dout0adr 0, dout1      158, dout1ce 1 d1we 0, d1addr 1
Wave : Strt 0, rdy 0, idle 0, done 0, din   44, din CE 0, dout0   26, dout0ce 0, dout0we 0,
dout0adr 0, dout1      158, dout1ce 0 d1we 0, d1addr 1
Wave : Strt 0, rdy 0, idle 0, done 0, din   44, din CE 0, dout0   26, dout0ce 0, dout0we 0,
dout0adr 0, dout1      158, dout1ce 0 d1we 0, d1addr 1
Wave : Strt 0, rdy 1, idle 0, done 1, din   44, din CE 0, dout0   26, dout0ce 1, dout0we 1,
dout0adr 0, dout1      249, dout1ce 1 d1we 1, d1addr 1
Buffer data
      43         15         41          7
Buffer data
     289        121        225        361
P - output data       26 output data       249 expected data       26 expected data
249
```

Longest Common Subsequence

Adding two numbers, finding average and variance of a data look rather lame compared to the capability of VHLS. Can we do any serious design with HLS? That was the question I tried to answer with the HLS of LCS c program described earlier. The LCS is still a tiny example of real world

bioinformatics algorithms! The C code of LCS ported from the SV code needs further changes to make it suitable to HLS.

Top level C function

The operations you need to be synthesized to RTL should be wrapped in a function. The testbench code and the synthesizable operations cannot be in the same main.c. So, I have spun out the lcs.c as a separate function.

No string/stdio libraries

The functions like strcpy/strlen in string.h cannot be synthesized by VHLS and there is a very good reason. Those string functions can work with arbitrary length strings by using the string termination character of '\0'. But real hardware code meant for FPGA has to work with fixed size logic resources. So, VHLS architects have chosen to not support them. But, the could have added support and then limited the string length to some parameter like MAX_STRING_LEN that can be configured. But, that is not there for the version of VHLS I am using.

No recursion

Recursion in software programs is implemented with a stack in a memory. The software recursion has no theoretical memory limit for the stack. Again, for FPGA we need fixed sized implementations, so, VHLS architects are not supporting recursion. They could have added support with size limit for stack. Even with stack support, converting recursion to RTL is a hard problem. For this particular algorithm, the trace back routine which used a recursive implementation can be converted to an iterative implementation because all the trace back routine was doing was just string reverse! The reversal is implemented by letting the trace routine just do a regular character concatenation in the subseq variable that is read out in the reverse order at the end of the lcs routine. For example the trace routine would create a sub sequence of ATCT which out get read out correctly as TCTA.

Lcs.h

```c
#include <string.h>
#include <stdio.h>
#define N 7
#define M 6

char trace [ N+1 ] [ M+1 ] ;
void lcs (char data_in[N+M+1], char data_out[N+1], int n, int m) ;
int max ( int a , int b ) ;
int trace_back ( int n , int m, char *seq0, char *subseq ) ;
```

lcs.c

```c
#include "lcs.h"
void lcs (char data_in[N+M+1], char data_out[N+1], int n, int m) {

char seq0[N], seq1[M], subseq[N+1];
char *ptr;
int i,j,len;
```

```c
int alnmat [ N+1 ] [ M+1 ] ;

for(i=0;i<n;i++) {
  seq0[i]=data_in[i];
}

for(i=0;i<m;i++) {
  seq1[i]=data_in[i+n];
}

printf("seq0,1 %s %s data buffer %s\n",seq0,seq1,data_in );

//strcpy(seq0,data_in);
//strcpy(seq1,data_in+n);

for ( i = 0 ; i <= n ; i ++ ) {
  for ( j = 0 ; j <= m ; j ++ ) {
    alnmat[i][j] = 0 ;
  }
}

for ( i = 1 ; i <= n ; i ++ ) {
  for ( j = 1 ; j <= m ; j ++ ) {
    alnmat [ i ] [ j ] = max ( alnmat [ i-1 ] [ j ] , alnmat [ i ] [ j-1 ] ) ;
    if ( seq0 [ i-1 ] == seq1 [ j-1 ] ) {
      alnmat [ i ] [ j ] = max ( alnmat [ i ] [ j ] , alnmat [ i-1 ] [ j-1 ] + 1 ) ;
    }
    printf ( "alnmat [ %2d ] [ %2d ] = %d \n" , i , j , alnmat [ i ] [ j ] ) ;
    if ( alnmat [ i ] [ j ] == alnmat [ i-1 ] [ j ] ) {
      trace [ i ] [ j ] = 'U' ;
    } else if ( alnmat [ i ] [ j ] == alnmat [ i ] [ j-1 ] ) {
      trace [ i ] [ j ] = 'L' ;
    } else {
      trace [ i ] [ j ] = 'D' ;
    }
  }
}

len = trace_back ( n , m, seq0, subseq) ;
printf("Done trace_back, Length is %d\n",len);
printf("subseq %s\n", subseq );
data_out[len]='\0';
for(i = len-1 ; i >=0 ;i--) {
  j = len-1-i;
  data_out[j]=*(subseq+i);
  printf("%c",data_out[j] );
}

printf("\n");

}

int trace_back ( int n , int m, char *seq0, char *subseq ) {
  int i,j,cnt,len;
  i = n; j=m;
```

```c
  len = 0;
  for(cnt = n+m+1 ; cnt > 0 ; cnt--) {
   if ( i == 0 || j == 0 ) {
     *subseq++ = '\0';
     break;
   }
   printf("trace[%d][%d] = %c \n",i,j,trace[i][j]);

   if ( trace [ i ] [ j ] == 'D' ) {
     *subseq++ = seq0[i-1];
     len++;
     printf ("%c\n", seq0 [ i-1 ] ) ;
     i--;j--;
   } else {
    if ( trace [ i ] [ j ] == 'U' ) {
      i--;
    } else {
      j--;
    }
   }
  }
  return len;
}

int max ( int a , int b ) {
 return ( a > b ? a : b );
}
```

For the VHLS generated Verilog RTL please refer to the GitHub repository files. It is not included here to reduce this book size and also for the reason that machine generated RTL is not easy to read.

SV testbench to check the generated Verilog RTL
After creating the testbenches for simpler HLS routines, I realized that keeping the input data in an array and accessing the array like a memory feels intuitive. So, the testbench uses data_buf for the input data and out_buf for the output data. The access to those arrays happens using the getc/putc string methods inbuilt in SV. You can think of
```
    data_in = data_buf.getc ( data_in_address0 ) ;
```
to mean
```
    data_in = data_buf [ data_in_address0 ] ;
```

```systemverilog
module tb ;
import thee_utils_pkg :: * ;

parameter DWIDTH = 32 ;
parameter N = 7 , M = 6 ;
parameter NUM_VECTORS = 1 ;
parameter FLUSH_CYCLES = 1 ;

bit clk ;
logic rstn ;
byte data_in ;
string expected_data = "TCTA" ;
string data_buf ;
```

```verilog
string out_buf ;
string seq0 = "ATCTGAT" ;
string seq1 = "TGCATA" ;
logic start , done , idle , ready ;
int cnt = 0 ;
int data_cnt = 0 ;
integer n , m ;
logic result ;

logic [ 3 : 0 ] data_in_address0 ;
logic data_in_ce0 , data_in_ce0_d ;
logic [ 7 : 0 ] data_in_q0 ;
logic [ 2 : 0 ] data_out_address0 ;
logic data_out_ce0 ;
logic data_out_we0 ;
logic [ 7 : 0 ] data_out_d0 ;

thee_clk_gen_module # ( .FREQ ( 100 ) ) clk_gen_i0 ( .clk ( clk ) ) ;

initial begin
  n = seq0.len ( ) ; m = seq1.len ( ) ;
  $display ( "Length of sequence 0 = %d and sequence 1 = %d" , n , m ) ;
  data_buf = { seq0 , seq1 } ;
  $display ( "%s" , data_buf ) ;
end

initial
  forever @ ( posedge clk )
    if ( data_in_ce0 )
      data_in = data_buf.getc ( data_in_address0 ) ;

initial begin
  out_buf = expected_data.tolower ( ) ;
  forever @ ( posedge clk )
    if ( data_out_ce0 && data_out_we0 )
      out_buf.putc ( data_out_address0 , data_out_d0 ) ;
end

always @ ( posedge clk ) begin
  if ( data_in inside { "A" , "T" , "C" , "G" } || data_out_d0 inside { "A" , "T" , "C" , "G" } )
    $display ( "Wave : Strt %b , rdy %b , idle %b , done %b , din %3c , din CE %b , din addr %d ,
dout0 %3c , dout0ce %b , dout0we %b , dout0adr %d " ,
    start , ready , idle , done , data_in , data_in_ce0 , data_in_address0 , data_out_d0 ,
data_out_ce0 , data_out_we0 , data_out_address0 ) ;
  else
    $display ( "Wave : Strt %b , rdy %b , idle %b , done %b , din %3d , din CE %b , din addr %d ,
dout0 %3d , dout0ce %b , dout0we %b , dout0adr %d " ,
    start , ready , idle , done , data_in , data_in_ce0 , data_in_address0 , data_out_d0 ,
data_out_ce0 , data_out_we0 , data_out_address0 ) ;
  $display ( ) ;
end

initial begin
  result = 1'bx ;
  start = 0 ;
```

```verilog
    toggle_rstn ( .rstn ( rstn ) ) ;
    repeat ( 1 ) @ ( posedge clk ) ;
    start = 1 ;
    for ( int i = 0 ; i < NUM_VECTORS ; ) begin
      repeat ( 1 ) @ ( posedge clk ) ;
      if ( ready ) begin
        if ( out_buf == expected_data ) begin
          $display ( "Pass : Got \"%s\" expected \"%s\"" , out_buf , expected_data ) ;
          if ( result !== 0 ) result = 1 ;
        end else begin
          $display ( "Fail : Got \"%s\" expected \"%s\"" , out_buf , expected_data ) ;
          result = 0 ;
        end
        i ++ ;
        start = 1 ;
      end else
        start = 0 ;
    end
    repeat ( FLUSH_CYCLES ) @ ( posedge clk ) ;

    print_test_result ( result ) ;
    $finish ;
  end

lcs mv (
 .ap_clk ( clk ) ,
 .ap_rst ( ~rstn ) ,
 .ap_start ( start ) ,
 .ap_done ( done ) ,
 .ap_idle ( idle ) ,
 .ap_ready ( ready ) ,
 .data_in_address0 ,
 .data_in_ce0 ,
 .data_in_q0 ( data_in ) ,
 .data_out_address0 ,
 .data_out_ce0 ,
 .data_out_we0 ,
 .data_out_d0 ,
 .n ,
 .m
 ) ;

endmodule
```

Utilities

This section provides scripts and tools other than SV code that improves the chip design process. These utilities are to be found in the useful_scripts directory of the Git repository accompanying this book.

Downloading many Github repos in one go

I have always wanted to study all the repositories in Opencores.org. I think the projects in Opencores use SVN for version control. There are hundreds of projects there. If you take 5 minutes to clone each then cloning time runs into many hundreds of minutes for the full website. Someone has supposedly copied all the projects into Github because he claims the Opencores website is not being maintained. The Github copy is in https://github.com/freecores/

To clone all the repos, I have copied the names of each repo into a file, freecores-repolist.txt. The script below uses this list file to clone one repo at a time. It works by extracting the first word of each line in the file. For every word that is extracted, the git clone command is used to clone the repo.

```tcsh
#!/usr/bin/tcsh

set repolist=$1

echo "List of repositories file $repolist"

foreach a ( `awk '{print $1}' $repolist` )
  echo "Now cloning $a"
  git clone https://github.com/freecores/$a
end
```

```
freecores-repolist.txt
1000base-x  Verilog 8800  2020-02-13  1000BASE-X IEEE 802.3-2008 Clause 36 -
Physical Coding Sublayer (PCS)
10_100m_ethernet-fifo_convertor Verilog 1072  2019-01-15  10/100M Ethernet-FIFO
convertor
```

The usage of the script is shown below.

```
po:~> cd ~
po:~> mkdir tmp
po:~> cd tmp
po:~/tmp> <ehgu lib>/useful_scripts/clone-many-repo.sh <ehgu
lib>/useful_scripts/freecores-repolist.txt
List of repositories file /home/po/RTL_design/dsn_verif/useful_scripts/freecores-
repolist.txt
Now cloning 1000base-x
Cloning into '1000base-x'...
...
Now cloning 10_100m_ethernet-fifo_convertor
Cloning into '10_100m_ethernet-fifo_convertor'...
...
Now cloning 1664
Cloning into '1664'...
^C
```

Exercise

Sometimes the clone process is interrupted by errors. In that case use the command "git checkout master" or check for an empty directory and reclone. Package this behavior into a script.

Sriram formatter

I had a coworker named Sriram who taught me advanced software. He insisted a lot on leaving plenty of white spaces around keywords and operators. At first it look weird to me. Later, it felt very useful to have code formatted that way. White space or simply space character lets the reader focus on the aspects that are relevant to the design rather than be confused by the effort of trying to find the keywords in the code. Your opinion may be different.

Example:

packet**.**data**[**i**] +=** rdpkt**.**data**[**j**+**pkt**.**pointer******2**][**i-j**];**

In Sriram's view

packet**.**data **[** i **] +=** rdpkt**.**data **[** j **+** pkt**.**pointer ****** 2 **] [** i – j **] ;**

The following shows a TCL script that does such a formatting. It operates in simple terms like this -

```
Open target file
for every line in file
    Reduce all spaces to one. SV does not care if there is one space or many space.
    Insert space around SV operators like +,*,=,…
    Remove space wrongly inserted by previous step, ex: substitute = = = to ===
    if you notice begin or end * begin or other indentation changing lines, update a running indentation variable
    Update indentation level if needed
    Print current indentation level white spaces
    Print this line into another file
    Update indentation level if needed
close targetfile
close tempfile

rename tempfile to target file
Do the operation for all *.sv files if the argument to the script is none, else, format only the file name specified.
```

TCL Script

```tcl
#!/usr/bin/tclsh

proc format_one_file { fl } {

  puts "Formatting $fl"
  set tempoutfile ".tmp_z13asxr93eqwe523"

  set fp [ open $fl r]

  set fpo [ open $tempoutfile w ]

  set line {}
  set prev_line $line
```

```tcl
set ind_lvl 0

while { [gets $fp line ] != -1 } {
   regsub -all {\s+} $line { } line
   regsub -all {=} $line { = } line
   regsub -all {>} $line { > } line
   regsub -all {=\s*=\s*=} $line {===} line
   regsub -all {=\s*=} $line {==} line
   regsub -all {!\s*=} $line {!=} line
   regsub -all {~\s*=} $line {~=} line
   regsub -all {\^\s*=} $line {^=} line
   regsub -all {<} $line { < } line
   regsub -all {<\s+<} $line {<<} line
   regsub -all {<\s*=} $line {<=} line
   regsub -all {>\s*=} $line {>=} line
   regsub -all {>\s+>} $line {>>} line
   regsub -all {!==} $line { !== } line
   regsub -all {,} $line { , } line
   regsub -all {:} $line { : } line
   regsub -all {:\s+:} $line {::} line
   regsub -all {\+} $line { + } line
   regsub -all {\+\s+\+} $line {++} line
   regsub -all {\+\s+\=} $line {+=} line
   regsub -all {\-\s+\=} $line {-=} line

   regsub -all {\*} $line { * } line
   regsub -all {\*\s*\*} $line {**} line
   regsub -all {\*\s*\=} $line {*=} line
   regsub -all {&\s*=} $line {\&=} line

   regsub -all {/} $line { / } line
   regsub -all {/\s*/} $line {//} line
   regsub -all {\/\s*\*} $line {/*} line
   regsub -all {\*\s*\/} $line {*/} line
   regsub -all {/\s*=} $line {/=} line
   regsub -all {/\s*/} $line { // } line

   regsub -all {\[} $line { [ } line
   regsub -all {\]} $line { ] } line
   regsub -all {\(} $line { ( } line
   regsub -all {\)} $line { ) } line
   regsub -all {\{} $line "\{ " line
   regsub -all {\}} $line " \}" line
   regsub -all {;} $line { ; } line
   regsub -all {[ ]+} $line { } line
   regsub -all {\t} $line {  } line
   regsub -all {\s+$} $line {} line
   set delayed_incr_ind 0
   if { [regexp {^\s*end\s+.*?\s+begin$} $line] } {
      incr ind_lvl -1
      set delayed_incr_ind 1
   } elseif { [regexp {begin$} $line] } {
      set delayed_incr_ind 1
   } elseif { [regexp {^\s*end\s*} $line] } {
```

```tcl
            incr ind_lvl -1
        }
        for {set i 0} {$i<$ind_lvl} {incr i} {
            puts -nonewline $fpo "  "
        }
        puts $fpo $line
        if { $delayed_incr_ind } {
            incr ind_lvl
        }
    }

    close $fp
    close $fpo

    file rename -force $tempoutfile $fl
}

set files [ lindex $argv 0 ]

if { $files == "" } {
  foreach fl [ glob *.sv ] {
    format_one_file $fl
  }
} else {
  format_one_file $files
}
```

Result after running on real code:

Before

```verilog
    if ( SHOW_CONTENTION ) begin
      //waddr = $urandom();
      do
        waddr = $urandom();
      while (waddr==raddr);
    end else begin
      waddr = raddr + DEPTH/2;
    end
```
After
```verilog
    if ( SHOW_CONTENTION ) begin
     // waddr = $urandom ( ) ;
     do
     waddr = $urandom ( ) ;
     while ( waddr == raddr ) ;
    end else begin
     waddr = raddr + DEPTH / 2 ;
    end
```

The editor script modifies only the white space characters of tab and space. So, using this command that checks for difference in non white space will show absolutely no difference.

diff -wb <old file> <new formatted file>
You may use sdiff or tkdiff in place of diff for side by side display

Exercise

The TCL script is not complete. Note the missed indentation on the do-while loop. One line blocks without begin-end is also not supported by the script. Can you add support for one line indentation, more SV operators like implication, &, | and more?

The script will run into many more exceptions and the if-elseif branching can get too much to tackle all possibilities of SV syntax. A machine learning program that takes hundreds of hand formatted code and learns how humans format is a better option. Can you give this a try?

Module blaster

Some Verilog netlists are exported in one single file containing all modules of the design. I gets complicated to search through the file for a given signal because the same signal name may appear in many modules. The text searching for the signal name in the single file netlist will show irrelevant code. I prefer to blast this single file netlist into more files with one module per file. This script is relatively easy. Read the source file, when you hit a new module open a file and write the lines into this file, until the endmodule keyword is reached.

```tcl
#!/usr/bin/tclsh

set filename [ lindex $argv 0 ]
set tempoutfile ".tmp_z1asdF3eqwe523"
set modorder "file_order.f"
set outdir "split_files"

file mkdir $outdir
file delete -force $outdir/*

set fp [ open $filename r]
set fpw [ open $outdir/$tempoutfile w ]
set fp_dotf [open $outdir/$modorder w]

set this_module "dollar_unit_scope"
set prev_module $this_module

cd $outdir
while { [gets $fp line ] != -1 } {
  if { [regexp {^\s*endmodule} $line ] } {
    puts "Found endmodule, output file with module name"
    puts $fpw $line
    close $fpw
    file rename $tempoutfile ${this_module}.sv
    set fpw [ open $tempoutfile w ]
    set prev_module $this_module
    set this_module "dollar_unit_scope"
  } else {
    set mod {}
    regexp {^\s*module\s+(\S+)} $line -> mod
    if { $mod != "" && $this_module == "dollar_unit_scope" } {
      puts "Found module $mod"
      puts $fp_dotf ${mod}.sv
```

```
        set prev_module $this_module
        set this_module $mod
    }
    puts $fpw $line
  }
}

close $fp
close $fpw
close $fp_dotf

file rename -force $tempoutfile "last_dollar_unit_scope_lines.sv"
cd ..
```

Working
shell$ ./module-blast.tcl .test_allmodules

The modules separated into files named after the module go into the output directory split_files.

shell$ ls -1 split_files/
file_order.f
padc_ana_stage.sv
padc_ana.sv
padc_dig.sv
padc.sv
tb.sv
last_dollar_unit_scope_lines.sv

The last lines of code outside of all modules is put into last_dollar_unit_scope_lines.sv. To be able to recompile the files in the same order as in the single file version, a file_order.f is saved.

Regular expressions for instantiation

The module and module instantiations mostly share the same signal names. So while typing code, I frequently use regular expressions to change the module definition section into an instantiation section.

Search for: (in|out)put\s*logic\s*(\S+)\s*,

replace with :.\2 ,

Meaning of the regular expression is, look for the pattern starting with the word

```
"input" or "output" followed by any number of space or tab characters and then the
word "logic" followed by any number of space characters followed by any word (this
word is saved in the variable no.2) followed by any number of space characters
followed by a comma.
```

When a match is found replace the matching pattern with the replacement patter,

```
a dot character followed by the saved word that is denoted by \2 followed by a
space and a comma character
```

Example

from:

input logic clk**,**

```
input logic rstn,
input logic din,
output logic fedge,
output logic redge,
output logic toggle,
```

```
to:
 .clk ,
 .rstn ,
 .din ,
 .fedge ,
 .redge ,
 .toggle ,
```

Obfuscator

Exercise
Create and obfuscator program following this pseudocode.

1. let the SV file to be obfuscated be my_file.sv
2. create a writable file file34124204858.sv, the weird name is to avoid collision with any existing file.
3. Remove block comments
   Read my_file.sv as a single long string of characters.
   Use regular expressions and remove block comments "/* */" in the entire string
   Output the modified string to file34124204858.sv.
   Rename file34124204858.sv to my_file.sv
2. Remove line comments
   Read my_file.sv line by line.
   Use regular expressions and remove line comments in every line.
   Output the modified lines to file34124204858.sv.
   After all lines are done, rename file34124204858.sv to my_file.sv
3. Get symbols
   Read my_file.sv as a single string
        Separate the string into words
   Sort the words into unique array
   remove SV keywords from the unique array
   call this array sym_array
4. Create replacement symbols
        get number of symbols in the sym_array, call this N
   create N random unique 8 character strings, with starting character as a letter. Other 7 characters can be letters or numbers or underscores. If creating random names is complicated you can call the resulting symbols as s1, s2, sN. The original symbol name and the new replacement name may be populated in a table.
5. foreach sym in sym_array
   read my_file.sv, replace sym with the random symbol created previously in the entire file.
6. optionally replace newline characters by space in the output file. This is because newlines improve readability but are not essential for the code to be syntactically correct.

Additional obfuscation possibilities
shuffle always blocks
shuffle port order

You would think the above pseudo-code would be sufficient to obfuscate many designs. Bad luck, it only works for one module. If you need to do this for all modules it can get quite complicated. One trick to deal with multiple modules is to concatenate all the modules into one text file and treat the symbols of the concatenated files in one go. This way the module name defined in the module definition and the module name used in the instances of the module are changed simultaneously.

Code Statistics

The number of lines of code created is an approximate guide to the complexity of a design. I do not fully agree with this idea, but, some organizations use it to get an idea about your experience from it. For one job opening, a company asked me to tell them how many lines of code I had done so far. I had no idea. I told them somewhere about 5000 to 10000. Later, I started tracking that number as I did more designs. I see that is well over 50,000. So, you may want to keep track of it to increase your job prospects or charge appropriate fee for designing some IP.

The following command is to be run at the root level of the directory containing all your code. The find command gets all the SV files and sends it to the xargs command. Xargs turns the line by line output from find into a single line files list usable by cat command. The cat command reads out all the files line by line and passes the read out lines to wc command that counts the lines and prints the total line count.

```
find . -name '*.sv' | xargs cat | wc
or
./useful_scripts/lines-of-code-created.tcsh
```

PLL frequency settings calculator

PLLs are typically used in the context of a clock multiplier. There is a need to set the dividers inside the PLL to values resulting in an output clock closest to the required frequency. The following utility TCL code sweeps through all possible divider combinations of the input and feedback divider s and prints the settings that gives the best output clock frequency.

Note that the script assumes a typical PLL that has only two dividers, an input divider that goes from 1 to N and a feedback divider that goes from 1 to M. Overall the output to input frequency ratio is M/N. There is also an assumption that the VCO inside the PLL has a fixed lower and upper frequency of operation beyond which the VCO does no produce useful clock.

The code works by first creating all the N x M divider combinations with two for loops. Then a check is performed with the frequency resulting from the chosen values to meet the VCO operating range for minimum and maximum frequencies. If the VCO frequency is ok, a check is made to see if this setting gives lower frequency error than before. Finally the better values are printed every time there is

improvement. The last value is the best match. Note that there could be many more better second and third matches that did not get printed because that did not improve the best error.

```tcl
# Variables, input
set ref 100e6
set req_out 120e6

set vco_min 25e6
set vco_max 900e6

set N_MAX 16
set M_MAX 64

# Calculation
set vco_f 0
set best_error 1e20

for { set N 1} { $N < $N_MAX } { incr N } {
  for { set M 1} { $M < $M_MAX } { incr M } {
    set vco_f [expr $M * $ref / $N]
    if { ($vco_f < $vco_max ) && ($vco_f > $vco_min)} {
      set error [expr {abs($req_out - $vco_f)}]
      if { $error < $best_error} {
        set best_error $error
        set freq_formatted [format "%1.4e" $vco_f]
        set err_formatted [format "%1.4e" $error]
        puts "PLL N=$N M=$M gives $freq_formatted Hz with deviation $err_formatted Hz"
      }
    }
  }
}
```

Output of script
shell> tclsh get_pll_settings.tcl
PLL N=1 M=1 gives 1.0000e+08 Hz with deviation 2.0000e+07 Hz
PLL N=3 M=4 gives 1.3333e+08 Hz with deviation 1.3333e+07 Hz
PLL N=4 M=5 gives 1.2500e+08 Hz with deviation 5.0000e+06 Hz
PLL N=5 M=6 gives 1.2000e+08 Hz with deviation 0.0000e+00 Hz

Parallel Scripting

Many problems in need of scripts do quite well with just a single core processor. Traditional serial scripting is sufficient. Rarely, a problem is so big that some speedup will be needed. The serial single core script you write may take many hours or days to complete. In such case, you have the option of going back to a fast and difficult language like C or running the same script in a multicore machine. These days, 32 core servers are easy to get in many companies or on the cloud. So, it may be easier to just run the same old script in a language like TCL/Python/PERL in the multicore server. Recently, I came across Message Passing Interface framework for running parallel programming and I asked if it would be possible to run regular Python scripts on multicore. The answer I just found out is an easy yes! Note that the MPI framework and parallel programming are a deep topic worth a library for itself. For the purspose of a chipper, lets restrict to getting some speedup for a regular quick and dirty script. I

have chosen the PLL settings TCL script for this. Unfortunately, TCL is going out of popularity and Python is the in-fashion language. So, I could not find an MPI library for TCL but could find an MPI library for Python. There seems to be parallel programming options in PERL using multithreading. You may want to try it if you are a PERL fan.

https://perldoc.perl.org/threads

MPI Python

MPI stands for Message Passing Interface. It is a language independent framework for parallel computing. For every language, someone would create a library to support the MPI framework. For Python the MPI has been packaged in mpi4py. This is only my first serious MPI program, so, I took baby steps to get here. First, I have just ported TCL to Python for the same single core use case. Note that your Python3 binary may be in a different place and you will have to change the first line.

```python
#!/usr/bin/python3

# Variables, input
ref=100e6
req_out=120e6

vco_min=25e6
vco_max=900e6

N_MAX=16
M_MAX=64

# Calculation
vco_f=0
best_error=1e20

for N in range(1,N_MAX) :
  for M in range(1,M_MAX) :
    vco_f = M * ref / N
    if ( (vco_f < vco_max ) and (vco_f > vco_min)) :
      error=abs(req_out - vco_f)
      if ( error < best_error) :
        best_error=error
        print("PLL N=%d M=%d gives %1.4e Hz with deviation %1.4e Hz" %(N,M,vco_f,error))
```

Conversion To Parallel Style

The MPI style of parallel programs can be thought of as parts of program that are run by all cores simultaneously. The parallel instances of the programs can also communicate with one another in some allowed ways. You provide different input data to each core so that the overall effect is N times data processing throughput compared to single core program. I learnt MPI python from this video

Introduction to parallel programming with MPI and Python

https://www.youtube.com/watch?v=36nCgG40DJo
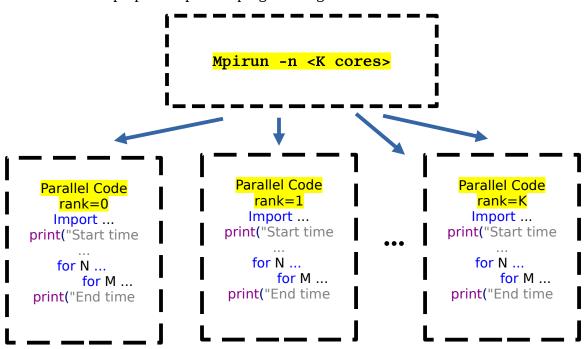
You may Google for MPI Python and learn from many available resources.

```python
import datetime

print("Start time",datetime.datetime.now())
# Variables, input
ref=100e6
req_out=120e6

vco_min=25e5
vco_max=900e6

N_MAX=32768
M_MAX=8192

cores=4

# Calculation
vco_f=0
best_vco_f=0
best_N=0
best_M=0
best_error=1e20

from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

for N in range(1,N_MAX,cores) :
    N_local = N + rank
    for M in range(1,M_MAX) :
        vco_f = M * ref / N_local
        if ( (vco_f < vco_max ) and (vco_f > vco_min)) :
            error=abs(req_out - vco_f)
            if ( error < best_error) :
                best_error=error
                best_N = N_local
                best_M = M
                best_vco_f = vco_f
                print("Core %d: PLL N=%d M=%d gives %1.4e Hz with deviation %1.4e Hz" %
(rank,N_local,M,vco_f,error))
comm.Barrier()
if(rank==0):
    print("Final Result")
comm.Barrier()

print("Core %d: PLL N=%d M=%d gives %1.4e Hz with deviation %1.4e Hz" %
(rank,best_N,best_M,best_vco_f,best_error))

print("End time",datetime.datetime.now())

#3vm
```

Explanation of code

The code is very similar to the single core script. The differences are the MPI function calls and the unseen parallel execution. Let me make the parallelism visible. When running the code from the mpirun binary the code runs in all the cores that you specify. The COMM_WORLD object is created to let the cores communicate between themselves. Wouldn't all the cores do the same work making parallelization useless? Thats where the rank variable comes into play. When you call get rank you get the core number that the code is executing on. So, you could divide the work across the cores using the rank. In this example, the rank is used to set the offset in the N loop, so that, core 0 works on N=0,4,8 and core 1 works on N=1,5,9 and so on. At times in the parallel executing code we need single core operation. So, it is customary to use the rank 0 process to do the serial operation and let the other cores skip the code that needs to work on single core. The barrier call forces all the cores to come to that point and wait for all cores to reach that point before proceeding further. Note that using too many barrier calls defeats the purpose of parallel programming.

The number of cores is set when the script is executed using mpirun command.

```
Shell> mpirun -n 4 python3 parallel_get_pll_settings.py
```

For my 4 core machine, I am not going to change the 4 and python3 settings so, I have copied the command into a shell script run_mpi_python.sh that contains

```
Shell> mpirun -n 4 python3 $1
```

```
Shell> ./run_mpi_python.sh parallel_get_pll_settings.py
```

```
Start time 2020-10-11 16:51:06.748833
Start time 2020-10-11 16:51:06.753526
Start time 2020-10-11 16:51:06.759575
```

```
Start time 2020-10-11 16:51:06.766497
Core 3: PLL N=4 M=1 gives 2.5000e+07 Hz with deviation 9.5000e+07 Hz
Core 3: PLL N=4 M=2 gives 5.0000e+07 Hz with deviation 7.0000e+07 Hz
Core 0: PLL N=1 M=1 gives 1.0000e+08 Hz with deviation 2.0000e+07 Hz
Core 2: PLL N=3 M=1 gives 3.3333e+07 Hz with deviation 8.6667e+07 Hz
Core 2: PLL N=3 M=2 gives 6.6667e+07 Hz with deviation 5.3333e+07 Hz
Core 2: PLL N=3 M=3 gives 1.0000e+08 Hz with deviation 2.0000e+07 Hz
Core 1: PLL N=2 M=1 gives 5.0000e+07 Hz with deviation 7.0000e+07 Hz
Core 1: PLL N=2 M=2 gives 1.0000e+08 Hz with deviation 2.0000e+07 Hz
Core 2: PLL N=3 M=4 gives 1.3333e+08 Hz with deviation 1.3333e+07 Hz
Core 3: PLL N=4 M=3 gives 7.5000e+07 Hz with deviation 4.5000e+07 Hz
Core 3: PLL N=4 M=4 gives 1.0000e+08 Hz with deviation 2.0000e+07 Hz
Core 3: PLL N=4 M=5 gives 1.2500e+08 Hz with deviation 5.0000e+06 Hz
Core 0: PLL N=5 M=6 gives 1.2000e+08 Hz with deviation 0.0000e+00 Hz
Core 1: PLL N=6 M=7 gives 1.1667e+08 Hz with deviation 3.3333e+06 Hz
Core 2: PLL N=7 M=8 gives 1.1429e+08 Hz with deviation 5.7143e+06 Hz
Core 2: PLL N=11 M=13 gives 1.1818e+08 Hz with deviation 1.8182e+06 Hz
Core 3: PLL N=12 M=14 gives 1.1667e+08 Hz with deviation 3.3333e+06 Hz
Core 1: PLL N=10 M=12 gives 1.2000e+08 Hz with deviation 0.0000e+00 Hz
Core 3: PLL N=16 M=19 gives 1.1875e+08 Hz with deviation 1.2500e+06 Hz
Core 2: PLL N=15 M=18 gives 1.2000e+08 Hz with deviation 0.0000e+00 Hz
Core 3: PLL N=20 M=24 gives 1.2000e+08 Hz with deviation 0.0000e+00 Hz
Final Result
Core 2: PLL N=15 M=18 gives 1.2000e+08 Hz with deviation 0.0000e+00 Hz
Core 0: PLL N=5 M=6 gives 1.2000e+08 Hz with deviation 0.0000e+00 Hz
Core 1: PLL N=10 M=12 gives 1.2000e+08 Hz with deviation 0.0000e+00 Hz
Core 3: PLL N=20 M=24 gives 1.2000e+08 Hz with deviation 0.0000e+00 Hz
End time 2020-10-11 16:52:23.706943
End time 2020-10-11 16:52:23.707000
End time 2020-10-11 16:52:23.707054
End time 2020-10-11 16:52:23.707061
```

Check Core Usage

Use the top command to see if all the cores are being used.

```
$shell> top

  PID USER       PR  NI    VIRT    RES    SHR S   %CPU %MEM     TIME+  COMMAND
 5625 chipper    20   0  494468  21152  13656 R  100.0  0.2   1:08.45 python3
 5626 chipper    20   0  494476  21076  13576 R   99.7  0.2   1:08.24 python3
 5627 chipper    20   0  494472  21004  13512 R   97.0  0.2   1:07.43 python3
 5628 chipper    20   0  494472  21064  13564 R   95.0  0.2   1:08.17 python3
```

Runtime for parallel case with 4 cores is 77 sec. Lets check the runtime for serial case. To do a fair comparison. I have setup the serial case from the parallel script by commenting out the MPI related code. The code is available in serial_baseline_get_pll_settings.py

shell> python3 serial_baseline_get_pll_settings.py

```
Start time 2020-10-11 17:06:23.979279
PLL N=1 M=1 gives 1.0000e+08 Hz with deviation 2.0000e+07 Hz
PLL N=3 M=4 gives 1.3333e+08 Hz with deviation 1.3333e+07 Hz
PLL N=4 M=5 gives 1.2500e+08 Hz with deviation 5.0000e+06 Hz
PLL N=5 M=6 gives 1.2000e+08 Hz with deviation 0.0000e+00 Hz
Final Result
PLL N=5 M=6 gives 1.2000e+08 Hz with deviation 0.0000e+00 Hz
End time 2020-10-11 17:10:13.714608
```

Runtime for serial case with 4 cores is 10min 13sec – 6min 23sec = 230 sec.

The speedup provided by 4 cores is 230/77 = 2.9 which is less than 4. In parallel programming, it is hard to get the full N times speedup by using N cores. Still, running a script in only about 1 min (77s) compared to about 3.5min (230s) is surely remarkable.

Compiled Python Script
Python scripts run line by line using an interpreter. This is normally slower than compiled code because during the compilation process many optimizations are done by the compiler. But there are ways to compile a Python code. One using the command line option of -m py_compile when calling Python. Note that I am using python3 because in my machine python refers the Python version 2.x.x. In your machine you may refer to just python which refers to python3.

python3 -m py_compile parallel_get_pll_settings.py

Executing the compiled byte code

python3 __pycache__/serial_baseline_get_pll_settings.cpython-36.opt-2.pyc

Start time 2020-10-13 08:10:49.485577
PLL N=1 M=1 gives 1.0000e+08 Hz with deviation 2.0000e+07 Hz
PLL N=3 M=4 gives 1.3333e+08 Hz with deviation 1.3333e+07 Hz
PLL N=4 M=5 gives 1.2500e+08 Hz with deviation 5.0000e+06 Hz
PLL N=5 M=6 gives 1.2000e+08 Hz with deviation 0.0000e+00 Hz
Final Result
PLL N=5 M=6 gives 1.2000e+08 Hz with deviation 0.0000e+00 Hz
End time 2020-10-13 08:14:36.503542

255 sec

Unfortunately, running compiled Python byte code is worse than just running it as without compile. So, I abandoned the Python compile experiment.

Single core C program

The get PLL setings script is not that complex. Is it possible write it C and check its runtime? Of course! Writing in C turned out be as easy as in Python. When I checked the runtime, it blew my mind! It took about 5.9 seconds compared to 77 seconds for even 4 core Python! More than 10x faster. Old C is still super fast. The -lm switch for GCC needed to compile math.h function calls.

C program

```c
#include<stdio.h>
#include<math.h>
#define CORES 1
// Variables, input
float ref_freq=100e6;
float req_out=120e6;
float vco_min=25e5;
float vco_max=900e6;
int N_MAX=32768;
int M_MAX=8192;
int cores=CORES;

void search_mn (char) ;
void main (void) {
   int i;
   for(i=0;i<cores;i++) {
     search_mn(i);
   }
}

void search_mn (char core) {

float vco_f=0;
float best_vco_f=0;
int best_N=0;
int best_M=0;
float best_error=1e20;
float error;
int N,M;

for (int N=1+core;N<=N_MAX;N=N+cores) {
   for (int M=1; M<M_MAX;M++) {
     vco_f = M * ref_freq / N ;
     if ( (vco_f < vco_max ) && (vco_f > vco_min)) {
        error=(float)sqrt((double)((req_out - vco_f)*(req_out - vco_f)));
        if ( error < best_error) {
           best_error=error;
           best_N = N;
           best_M = M;
           best_vco_f = vco_f;
           printf("Core %2d: PLL N=%5d M=%5d gives %1.4e Hz with deviation %1.4e Hz\
n",core,N,M,vco_f,error);
        }
     }
   }
}
}
```

```
printf("Final Result from core %2d: PLL N=%5d M=%5d gives %1.4e Hz with deviation %1.4e Hz\n"
,core,best_N,best_M,best_vco_f,best_error);

}
```

```
run-benchmark-serial-c.sh :
gcc serial_baseline_get_pll_settings.c -lm
date
./a.out
... <-- repeated 10 times to reduce error in runtime measurement
./a.out
date
```

```
./run-benchmark-serial-c.sh
Sat Oct 17 13:25:58 IST 2020
Core   0: PLL N=    1 M=    1 gives 1.0000e+08 Hz with deviation 2.0000e+07 Hz
Core   0: PLL N=    3 M=    4 gives 1.3333e+08 Hz with deviation 1.3333e+07 Hz
Core   0: PLL N=    4 M=    5 gives 1.2500e+08 Hz with deviation 5.0000e+06 Hz
Core   0: PLL N=    5 M=    6 gives 1.2000e+08 Hz with deviation 0.0000e+00 Hz
Final Result from core  0: PLL N=    5 M=    6 gives 1.2000e+08 Hz with deviation
0.0000e+00 Hz
...
Final Result from core  0: PLL N=    5 M=    6 gives 1.2000e+08 Hz with deviation
0.0000e+00 Hz
Sat Oct 17 13:26:57 IST 2020
```

Runtime is 59 seconds for 10 repetitions => 5.9 s per repetition

Multicore C program
Can you run parallel C for even more speedup? Sure! The code is similar to the parallel Python script but it is made complicated by the weird pointer syntax of C to use the functions from pthread library. Contrast to Python, this program does not use MPI but uses threads. The parallely executing code does not communicate with each other. Multithreading is faster than MPI but multithreading is limited in number of threads that can be launched in a single machine. The threads are launched with the argument of i which takes 0,1,2,3 values that is used to distribute the workload across threads. The parallel run completed in just 1.8 seconds!

```
#include<stdio.h>
#include<math.h>
#include <pthread.h>
#define CORES 4
// Variables, input
float ref_freq=100e6;
float req_out=120e6;
float vco_min=25e5;
float vco_max=900e6;
int N_MAX=32768;
int M_MAX=8192;
int cores=CORES;

void *search_mn (void * ) ;
void main (void) {
int i;
    pthread_t thread_id[CORES];
    for(i=0;i<cores;i++) {
```

```c
        //search_mn(i);
        pthread_create(&thread_id[i], NULL, search_mn, &i);
    }
    for(i=0;i<cores;i++) {
        pthread_join(thread_id[i], NULL);
    }
}

//void search_mn (char core) {
void *search_mn (void  *arg) {

float vco_f=0;
float best_vco_f=0;
int best_N=0;
int best_M=0;
float best_error=1e20;
float error;
int N,M;
int core=*(int *)arg;

for (int N=1+core;N<=N_MAX;N=N+cores) {
   for (int M=1; M<M_MAX;M++) {
      vco_f = M * ref_freq / N ;// N_local;
      if ( (vco_f < vco_max ) && (vco_f > vco_min)) {
         error=(float)sqrt((double)((req_out - vco_f)*(req_out - vco_f)));
         if ( error < best_error) {
            best_error=error;
            best_N = N;//N_local;
            best_M = M;
            best_vco_f = vco_f;
            printf("Core %2d: PLL N=%5d M=%5d gives %1.4e Hz with deviation %1.4e Hz\
n",core,N,M,vco_f,error);
         }
      }
   }
}
printf("Final Result from core %2d: PLL N=%5d M=%5d gives %1.4e Hz with deviation %1.4e Hz\n"
,core,best_N,best_M,best_vco_f,best_error);

}
```

```
run-benchmark-parallel-c.sh

gcc parallel_get_pll_settings.c -lm -lpthread
date
./a.out
... total ten times
./a.out
date
```

Note that the -lpthread switch of GCC to use the pthread library.

```
./run-benchmark-parallel-c.sh
```

Tue Oct 13 08:23:42 IST 2020
Tue Oct 13 08:24:00 IST 2020

18 sec fo r10 repetitions => 1.8 s per run.

SV parallel
Can SV do the parallel trick too? After all, SV was made to be parallel from the ground up. So, I used
the fork-join to run the SV code in parallel.

```systemverilog
program tb;
// Variables, input
real ref_freq=100e6;
real req_out=120e6;
real vco_min=25e5;
real vco_max=900e6;
int N_MAX=32768;
int M_MAX=8192;
int cores=4;

initial begin
   fork
      for(int i=0;i<cores;i++) begin
         search_mn(i);
      end
   join
end

task automatic search_mn (
input byte core=0
);
real vco_f=0;
real best_vco_f=0;
int best_N=0;
int best_M=0;
real best_error=1e20;
real error;

for (int N=1+core;N<=N_MAX;N=N+cores) begin
// N_local = N + rank;
   for (int M=1; M<M_MAX;M++) begin
      vco_f = M * ref_freq / N ;// N_local;
      if ( (vco_f < vco_max ) && (vco_f > vco_min)) begin
         error=$sqrt((req_out - vco_f)**2);
         if ( error < best_error) begin
            best_error=error;
            best_N = N;//N_local;
            best_M = M;
            best_vco_f = vco_f;
            $display("Core %2d: PLL N=%5d M=%5d gives %1.4e Hz with deviation %1.4e
Hz",core,N,M,vco_f,error);
         end
      end
   end
end
```

```
$display("Final Result from core %2d: PLL N=%5d M=%5d gives %1.4e Hz with deviation %1.4e Hz" ,core,best_N,best_M,best_vco_f,best_error);

endtask

endprogram

run -all
Core  0: PLL N=    1 M=    1 gives 1.0000e+08 Hz with deviation 2.0000e+07 Hz
Core  0: PLL N=    5 M=    6 gives 1.2000e+08 Hz with deviation 0.0000e+00 Hz
Final Result from core  0: PLL N=    5 M=    6 gives 1.2000e+08 Hz with deviation
0.0000e+00 Hz
Core  1: PLL N=    2 M=    1 gives 5.0000e+07 Hz with deviation 7.0000e+07 Hz
...
Core  1: PLL N=   10 M=   12 gives 1.2000e+08 Hz with deviation 0.0000e+00 Hz
Final Result from core  1: PLL N=   10 M=   12 gives 1.2000e+08 Hz with deviation
0.0000e+00 Hz
Core  2: PLL N=    3 M=    1 gives 3.3333e+07 Hz with deviation 8.6667e+07 Hz
...
Core  2: PLL N=   15 M=   18 gives 1.2000e+08 Hz with deviation 0.0000e+00 Hz
Final Result from core  2: PLL N=   15 M=   18 gives 1.2000e+08 Hz with deviation
0.0000e+00 Hz
Core  3: PLL N=    4 M=    1 gives 2.5000e+07 Hz with deviation 9.5000e+07 Hz
...
Core  3: PLL N=   20 M=   24 gives 1.2000e+08 Hz with deviation 0.0000e+00 Hz
Final Result from core  3: PLL N=   20 M=   24 gives 1.2000e+08 Hz with deviation
0.0000e+00 Hz
$finish called at time : 0 fs : File
"/home/po/RTL_design/dsn_verif/useful_scripts/parallel_scripting/parallel_get_pll_sett
ings.sv" Line 11
run: Time (s): cpu = 00:00:00.01 ; elapsed = 00:00:55 ...
```

Too bad SV runs the code serially one process after another. It is possible that I don't understand the fork join syntax correctly or that my simulator is not capable of parallel processes. I have a hunch that it is the simulator's limitation. Please try this experiment on a parallel capable simulator. Also, don't discount SV when you need some scripting task to be done. SV is the best language to use when your script needs a lot of bit manipulation. The runtime was 55 seconds which is still better than the 77s parallel Python case. Perhaps, the compilation of SV to object code makes it faster than Python.

Cheat code parallel TCL/SV/Shell
I am not aware of an easy and true multithreading/processing or distributed library for TCL. But there may be a cheat version using the Shell process spawning feature using the "&" character. Use a shell wrapper to launch many processes of the same TCL script but each script takes a different data. In the PLL settings example, you could make the starting index for the N exploring loop to be an argument supplied by the call from shell.
TCL/SV script instance 0: Explores i = 0,4,8,... of the possible N values
TCL/SV script instance 1: Explores i = 1,5,9,... of the possible N values
TCL/SV script instance 2: Explores i = 2,6,10,... of the possible N values
TCL/SV script instance 3: Explores i = 3,7,11,... of the possible N values

Suggested Shell script pseudocode
ncores=4
for i from 0 to ncores-1
    get-pll-settings-with-argument.tcl i **& ;#each script runs in parallel**

You could also do all the code in Shell.

```
for i from 0 to ncores-1
    get-pll-settings-with-argument.sh i & ;#each script runs in parallel
```

Exercise
Figure out a way to run scripts in parallel in the GPU of your laptop/desktop machines. Hint: If you are using NVIDIA graphics card, you may use the CUDA Python library.
Try out the CUDA Python on the Cloud using AWS.

ML/AI Scripting?

When calculator was invented, it relieved engineers of the tedium of arithmetic calculations on large numbers. When computer was invented, it relieved engineers of the chore of repeating mundane steps in designing chips. How can you leverage the ML/AI software and hardware available now to automate your work even more? I don't the answer.

Poor man's PM tool

For managing my own work, I found that the idea of listing and adding up task expected times is a profound method to stay focused on the target. It will let you see how close you are to the target. The idea is simple, just list down all the activities to be done to complete your design in a spreadsheet using LibreOffice or MS Excel. In two columns next to it, add the amount of time already spent doing it and the amount of time estimated to complete it. Note that the estimated time is from start to finish, don't less the time already spent. A third column is a derived one showing the amount of work remaining in the task in units of time. By derived, I mean let the computer calculate this with a formula and not type it by hand. This is nothing but the full estimated duration of the task less the time already spent on it. The last row of the sheet is the total of all these numbers. So, you would get total time spent in the project, total time needed for the project from start to finish and last, the total estimated time needed to complete the incomplete tasks.

In another spreadsheet, estimate the amount of work hours available to you. I create a series of dates from the start of the project to the end of the project in column A. In column B, I mark days as available or not by a 0 or 1. See the value of binary :)! You may remove vacation, weekends and other unavailable days by setting the corresponding cell 0. in Column C, you could conditionally copy column B only if the date is a future date. It accomplish this, you may have to compare the date against a spreadsheet function for today. At the last row, total the available future days. Set one cell at the top to number of hours per day. Six hours does well for me because it is very difficult to maintain focus on the task for more than 6 hours against all distractions. You could choose a different number based on your ability to focus. You may also reduce this number to provide some margin in your plan. Suppose, you reduce the available time per day to 5, the project will appear harder because you will need more days to complete the same project.

Now, you have the time needed to complete the project on one side and on the other side you have remaining hours to your deadline. The difference between the two is the most important number showing if you are slacking or not.

Minor improvements to the plan is to change the single time numbers to three values, each matching best, typical and worst case estimates for the time needed for every task. This way you can somewhat model the randomness of the future. You can have high variation in best to worst for unknown tasks and same number for best to worst for well known tasks. Another improvement is to create hierarchical plan using multiple sheets. Each sheet does the time estimation and the master sheet gives the overall project execution picture. A third and most important improvement is to add time for re-planning the project. It is easy to forget that all plans make assumptions about the future and future is unpredictable. You cannot even be sure of when you will have breakfast tomorrow. How can you be sure of a plan that involves work you have never done before?

Note: This PM tool does not model multiple people parallel execution of a project. It is primarily intended for one person use.

Drawing library

Drawings take a long time to make. Similar to design libraries with neatly structured reusable building block designs, a drawing library helps in creating well understood documents quickly. Engineers are like a subspecies of humans with somewhat greater tendency to understand the world in pictures and with somewhat poorer capacity to understand wordy text. I once had the privilege of working with non-English speaking chippers and I had a tough time communicating in spoken English. I slowly figured out that diagrams are the best and most accurate language to talk to those non-English speaking chippers. Once made, your documents may be viewed by a large number of non-English speaking chippers, so, diagrams are the best medium to convey accurate chipping information for eternity.

Some useful components for a chipping drawing library are logic gate symbols, example block diagrams, FSM state diagrams and flow charts. You may also add transistor level circuits and your organization specific diagrams to your library. Any software is appropriate for this purpose, Microsoft Word or Microsoft PowerPoint or Openoffice or Xfig or Inkscape or online cloud drawing tools. Dedicated tools like Microsoft Visio can be very useful too.
There is also a subclass of drawings that can be embedded right inside code files. For this you need to restrict yourself to ASCII characters only. You would be amazed at the variety of drawings that may be created with just ASCII characters. I found an online tool for this here -

ASCII diagram web tool
http://asciiflow.com/

If you have no time at all to create reusable diagrams library for your documents at least draw diagrams on paper with a pencil and scan/photograph the diagram and insert in your diagrams. Hand drawing is by far the easiest way to add diagrams to your documents but just that it is not much reusable. Basic gates like buffer, AND, OR, inverter, NAND, NOR are in available in ehgu_drawing_lib.svg file attached in the <Git repository>/useful_files/ directory.

Please make your own drawing library and share online.

Signing Your Work

In many ancient Tamil poems the author would sign by including his own name in the last stanza of the poem. This way the users of the poem need not consciously remember the author's name and there is very little chance of forgetting the authors name. Note the uncanny resemblance to the modern watermark idea. Let me demonstrate the idea with a funny poem.

Semiconductor chips are no potato chips
When one is junk the other is slam dunk
Semiconductor chips are society's fulcrum
That greatly levers up the effort of viccrum

Note how the author's name rhymes with the previous line. Forgetting the author's name would mutilate the poem, so, the author's name is likely to survive time, as long as the poem itself survives time. You could hack this scheme by coming up with a matching rhyming name. I cannot preclude that. Most of the time, people are lazy and would just use the information as is.

The following script would do that. It works by finding out all the SV files under the directory you run this code. The files are passed to xargs that formats the multiline output of find into multi word output. The sed command works on theses files one by one and inserts logic vikram; before endmodule or endpackage or endprogram.

replaces
endmodule

with
  logic vikram;
endmodule

sign-code.sh

```
#!/bin/bash
find . -name '*.sv' | xargs \
sed -i -e 's/\(^\s*endmodule\)/  logic vikram;\n\1/g' \
    -e 's/\(^\s*endprogram\)/  logic vikram;\n\1/g' \
    -e 's/\(^\s*endpackage\)/  logic vikram;\n\1/g'
```

Unit 6 – Quiz

It is fun to learn using a quiz. Please simulate or start from first principles and check the answer before looking at the answer.

Numbers

Q. Suppose, you created a number system called alphanumeric number system. It has 36 symbols – 0-9, a-z. Decimal 16 is denoted as G and decimal 40 is denoted as 14. What is the alphanumeric representation for decimal 1024? Hint: Write down a table of decimal to alphanumeric for the first 36 numbers starting from 0.

Solution

1. 1024 / 36 = quotient 28, remainder 16

Alphanumeric Coding table:

0-9, 10=A, 11=B, 12=C, 13=D, 14=E, 15=F, 16=G, 17=H, 18=I, 19=J, 20=K, 21=L, 22=M, 23=N, 24=O, 25=P, 26=Q, 27=R, 28=S, 29=T, 30=U, 31=V, 32=W, 33=X, 34=Y, 35=Z

28 is S

16 is G

so $1024_{10} = SG_{36}$

Q: There is an accumulator in the design that needs to sum up 513 8 bit unsigned samples. What should be size of the accumulator in bits to avoid any overflow during the addition process? Assume accumulator is reset to 0 before starting addition.
a. 16b
b. 18b
c. 13b
d. 17b
A: It is tempting to say 10bits for 513 + 8b making 18b. But closer examination reveals the formula to use. Suppose each 8b sample was max value 255, the product will be 513 x 255 = 130815 = 0x1FEFF is 17b. The formula to use is ceiling($log2(513 * (2^8-1))$).

Q: Two unsigned numbers, 13 bit and 15 bit wide are added. What is the resulting maximum bits without overflow?
a. 28b
b. 15b
c. 14b
d. 16b
A: Adding 13b number to 15b number at maximum causes a 16b number. You could do ceiling($log2((2^{13}-1)+(2^{15}-1))$). In general, addition of two numbers needs only one bit more than the largest operand width.

Q: How many bits are need to represent the number 4095 in unary or thermometer code?
a. 12
b. 10

c. 4095

d. 4096

A: Unary code uses one new bit per number, such as 10 bits for 10 (1111111111). So, 4095 needs all 4095 bits.

Q: Suppose an unsigned decimal floating point type is defined as df16,

d33d32d31d30 . d23d22d21d20 d13d12d11d10 d03d02d01d00 meaning

d33-d30 codes for 0-9 in binary and d2n,d1n code for 3 decimal digits in binary and form the fractional part. The 4 bits forming d0n code for the exponential as power of 10 from 0 through 9. For example, 4.52×10^6 would be 0100_0101_0010_0110 in df16.

Multiply df16(1001_0001_0111_0011) x df16(0001_1000_0000_0100), truncate excess bits.

a. 1.65×10^8 b. 1.44×10^7 c. 1.11×10^4 d. 2.444×10^6

A: 9.17×10^3 times 1.8×10^4 gives $9.17 \times 1.8 \times 10^{(3+4)}$

$= 16.506 \times 10^7$

$= 1.65 \times 10^8$

Electrical

Q: Which combination of the following resistors make 28ohm? The comma character is used to mean serial connection. Assume tolerance of the resistors is 0%

a. 10x 50ohms in parallel, 19 ohm, 4x 16ohms in parallel

b. 5x 50ohms in parallel, 2x 19ohms in parallel, 2x 16ohms

c. 50ohms, 19ohms, 16ohms

d. 6x parallel of (50ohms,16ohms), 19ohms

The formula for effective parallel resistance of a combination of two resistors is R1*R2/(R1+R2). If you apply this formula on many parallel instances of the same resistance, it reduces to R1/N where N is the number of parallel instances. The effective series resistance of R1,R2 is just R1+R2. Applying the formula to a gives, 50/10+19+16/4=5+19+4=28ohm

Ans: a

Q: A 4 socket extension box is connected to a 110V 10A wall outlet. You want to connect as many appliances to the box as possible. You have a 100W incandescant bulb, 2x 600W hair driers, a 1000W toaster and many 10W LED Bulb. Which set of appliances works fine with the extension box.

a. toaster, hair dryer, LED bulb

b. Incandescent bulb, LED bulb, hair dryer

c. 2x hair dryer, incandescant bulb

d. 20x LED bulbs, toaster

The power capacity of the wall outlet is 110V x 10A = 1100 W. The sum of the power drawn by the appliances should not exceed this limit. If it exceeds, the wires inside the wall outlet could overheat. The combination of incandescant bulb, LED bulb and hair drier consumes 100+10+600 = 710W which is well within the limit of 1100W. Other combinations consume more power

Ans: b

Q: Suppose that the city of Seoul, Korea uses supercapacitor based rapid charging whenever the bus stops in a station. After one such charge the super capacitor voltage was 2V. After driving for 5 minutes to the next station the voltage dropped to 1.5V. If the supercap is 10kF, what was the energy consumption of the bus for the ride from the previous charge?

a. 10 kJ   b. 8.75 kJ c. 20 kJ   d. 2.5 kJ

The energy in a capacitor is $\frac{1}{2} CV^2$. Between charge and discharge the change in energy is

$E = \frac{1}{2} CV_1^2 - \frac{1}{2} CV_2^2 = \frac{1}{2} C (V_1^2 - V_2^2) = 10000*(2*2 -1.5*1.5)/2.0 = 8750.0$ J

Ans: b

Q: Two competing companies, are offering fast EV charging options. You as a prospective electric car buyer have to find out which company's charging profile delivers higher average power. The 10% to 90% charging current profile for the two companies are shown here. For simplicity, assume that the charging voltage is constant.

Company 1: Brand name Snap EV charging standard

200A for 5 minutes, 500A for 10 minutes, 10A for 5 minutes

Company 2: Blink EV charging

90A for 3 minutes, 600A for 12 minutes, 50A for 5 minutes

a. Snap charging better than Blink   b. Snap and Blink offer same power

c. Blink better than Snap   d. Need battery size in kWh to decide

Energy delivered, $E = V*I*t$ and average power is $P = E/T$

For Snap, $E = V*(200*5+500*10+10*5) = 6050$ units and $P = 302.5$ units

For Blink, $E = V*(90*3+600*12+50*5) = 7720$ units and $P = 386.0$ units

Ans: c

Q: A compute logic consumes 10mW when operating at Vdd of 1V and 10MHz. If you use two instances of the logic operating at 0.6V and 5MHz, what is the power consumed for the two core

design? Assume that there is zero overhead associated to use two cores compared to one core and assume leakage power is zero.

a. 3.6mW   b. 5mW   c. 20mW   d. 6.4mW

Power of a digital logic can be modelled using, $P = aCV^2f$, where a is activity factor, C is capacitance of the chip and V is power supply voltage and f is operating frequency. The meaning of the equation is that if the chip is reduced to an effective capacitance of aC that is charged and discharged upto V volts at a frequency of f then it would consume the power per this formula.
For one core implementation, $P_1 = a_1C_1V_1^2f_1$
For dual core implementation, $P_2 = a_2C_2V_2^2f_2 = (2a_1C_1)V_2^2f_2 = (2a_1C_1)(0.6*V_1)^2(0.5f_1) = (2*0.5*0.6^2)a_1C_1V_1^2f_1 = 0.36P1 = 3.6mW$.
Note that voltage has a squared relationship and provides more bang for the buck.

Ans: a

General

Q: A 12 bit sine wave needs to be produced inside a digital low cost FM radio receiver. What cell from the technology library is best suited for sine wave generation?

a. ROM

b. Direct logic implementation of sine function

c. RAM

d. Flops reset/preset to table values

A: In general the ROM implementation gives the smallest area for constant functions like sine wave table. A RAM is always bigger than a ROM for the same number of bits because a RAM has a write channel that is absent in ROM. Implementing the table using flop reset values would be insane when you could do the same with combinational logic gates. Using combinational logic gates to implement the table can be smaller sometimes but it is not a structured method because ROMs are more flexible. By just changing the metal mask of the ROM you can get a different table. In this case, say you want the sine wave table to start at 45 degrees rather than 0 degrees that will be a one or two layer change for a metal mask programmable ROM but may take all layer change for logic gate implementation.

Q: An FSM has 5 states and 3 outputs. The outputs are registered with flops for timing purpose. The states are encoded in binary. How many state elements are in the design?

a. 3
b. 6
c. 5
d. 8

A: State elements generally refer to the memory elements in the system that affect the next state of the system. For an FSM, it is the state flops. The flops used for timing purpose is not considered as state elements because these flops' outputs do not feedback into the FSM. The correct answer is 3 flops needed to code 5 states.

Q. An FSM has 7 states. The state transition is controlled by inputs to the FSM and an internal counter that can count up to 10 clock cycles. How many state elements are there in the FSM.
a. 7
b. 3
c. 17
d. 5
A: The counter used in the FSM also indirectly contributes to the state of the FSM. There are 3bits to encode 7 states and 4 bits to encode up to 10 count values for a total of 7.

Q. A 4 bit LFSR logic is implemented in the design by AND-ing the LFSR with a polynomial value and then XOR-ing the result to get the next bit input to the LFSR. During operation, when the LFSR is about to enter the 'b0001 state, a cosmic ray particle forces the LFSR to enter 'b0000. What will be the LFSR state 3 cycles after entering 'b0000.
a. whatever lfsr logic produces normally after 3 cycles of 'b0001
b. 'b0000
c. 'b1111
d. whatever lfsr logic produces normally after 4 cycles of 'b0001
A: LFSR and other XOR gate dependent circuits get stuck when the register becomes all zeros.

Q. Write timing constraints for this design. Assume appropriate declarations.
```
always_ff @(posedge clk) begin
  q<=d;
  cb2 <= ~cb2;
end

always_ff @(posedge cb2)
  q2<=q;
```

a. set_input/output_delays, create_clock -period <period of clk> clk, create_generated_clock -source clk -division 2 cb2
b. set_input/output_delays, create_clock -period <period of clk> clk, create_generated_clock -source clk -division 4 cb2
c. set_input/output_delays, create_clock -period <period of clk> clk
d. set_input/output_delays, create_generated_clock -source clk -division 8 cb2

A: There are two clocks in the design. Both needs to be constrained. The clk signal looks like a primary input and SDC for that is create_clock and the cb2 signal is derived from clk, SDC is create_generated_clock. The signal cb2 toggles every cycle of clk so it is a by-2 clock divider. So, the division parameter of the generated clock constraint should be 2. Ans: a

Q: A clock buffer standard cell named CB10 has the following delay characteristics.
Rise delay is 10ps
Fall delay is 9ps

The flops of the design are driven with a clock tree built using only these buffers. For the sake of this question neglect interconnect delay and all other delays except the buffer propagation delay. The leaf level of the tree where the flops' clock pins are driven can have upto 100 buffers in series from the chip primary clock input pin. What is the worst duty cycle at clock pins of the flops of the input clock which is 500ps high and 500ps low.

a. 50% b. 45%   c. 40%   d. 30%

You can see that when the clock passes through the first buffer the output waveform would look like - rise edge 10ps, fall edge 509ps, second rise edge at 510 ps make a duty cycle of 499ps/1000ps. After 100 stages the clock would lose 1ps on logic high duration for every stage making a duty cycle of 500 -100ps, 400ps on time, 40% duty cycle.
Ans: c

For the same clock tree assume a +/-0.25ps random jitter introduced by every buffer, that is, the delay could vary by 9.75ps to 10.25ps for rise delay and 8.75 to 9.25 for fall delay. What is the expected reduction setup margin because of clock buffer jitter on a timing path sharing the same 100 buffers between launch and capture paths.

a. 0ps   b. 25ps   c. 50ps d. 75ps

Launch edge could be 9.75x100 and capture edge could be at 10.25 x 100, a difference of 50ps. Though this question suggests all the random jitter takesthe extreme values for every cell, in reality, most random phenomenon even out over many samples and show lesser and lesser standard deviation for large samples, so the effect of jitter could be negligible for a 100 buffer chain.
Ans : a

Q: Your T-shirts are worn out. You want a new T-shirt. Where do you go to get a new T-shirt?
a. Order online
b. Go to a physical store
c. Go to a chip industry conference
d. Borrow from friend
A: This question is just to jolt you into focusing on chipping again :) My personal answer is to attend a conference!

Q: You are making a digital ring oscillator using only inverters. How many inverters can you use in the ring?
a. any odd number >= 3
b. any even number >= 3
c. any number > 3
c. any number
A: A digital ring oscillator from inverters needs odd number of inverters. One inverter connected in feedback does not produce the 360 degree phase rotation on the output and does not oscillate. The smallest is 3. Even numbers result in a latch and not an oscillator. Ans: a

Q: A two phase programmable clock generator is made using N identical inverters forming a ring oscillator. The reference phase or zero phase is obtained from a fixed tapping point in the ring. The second clock output that needs a fixed phase relationship from the first is obtained by tapping the ring from any other point. The second clock output has a mux to do this variable phase offset output. The programming of the phase offset is performed using 3 select bits. The frequency should increase linearly with increasing select values with a step size of 1 bit change providing 10 degree +/-1 degree tolerance. How many stages are needed for the given settings? Assume works with 0 propagation delay.

a. 7   b. 17   c. 27   d. 37

One cycle of a clock is 360 degrees. For 10 degree resolution, every stage needs to provide 360/10 = 36 degree phase offset. But to get an error less than 10 degree, we can choose 37 stages which conveniently is also an odd number.

Ans: d

Q: A multiplier is working at 100MHz maximum frequency. After inserting one pipeline stage to speed up the operation. What is the expectation on the new maximum frequency.
a. 300MHz
b. 200MHz
c. <300MHz
d. <200MHz
A: Adding a pipeline stage to split the logic in half almost doubles the speed of operation. There is some overhead delay added by the pipelining flops. Overall the maximum frequency does not exceed twice the earlier frequency. Ans: b

Q: The size of an array multiplier with 2 M bit operands grows as
a. 2M
b. M power 2
c. M power M
d. 2 power M
A: An array multiplier computes MxM partial product terms. So, the size approximately grows as M squared.

Q: A memory has 4096 words. How many bits are needed to represent the address bus?
a. 12b
b. 11b
c. 13b
d. 16b
A: ceiling(log2(4096)) is 12, Ans: a

Q: A helicopter has 3 rotor blades mounted at the top. A tachometer sensor is installed above the cockpit. The sensor shines coded laser pulses vertically up and whenever a rotor blade passes across the laser light's path, the reflection is sensed by a photodiode. The sensor outputs a glitch free digital logic 1 throughout the duration of a rotor blade passing over the laser and logic 0 otherwise. Suppose the rotor assembly spins at 100rpm, what is the output frequency of the tachometer sensor?
a. 5 Hz b.10Hz c. 15Hz d. 300Hz

A: 100 rpm is 100 rotations per minute is 100 / 60 rotations per second is 100/60 Hz. There are 3 blades producing 3 pulses per rotation of the assembly. So 3 * 100 / 60 is 5Hz. Ans: a

Q: You have the resources to use any process node to implement a 64 bit 1024 core processor for an exascale supercomputer application. Choose the best process node from the options
a. 7nm b.28nm c. 45nm d. 90nm
A: Smaller process nodes are generally faster than bigger process nodes. Processors for supercomputer application typically try to get the best performance possible, so you want to choose the fastest process. Ans: 7nm.

Q: A clock divider counts the cycles of a perfect 50% duty cycle input clock. The divider outputs a logic 1 for cycles 0 through 4 and a logic 0 for cycles 5 through 8. what is the duty cycle of the output clock?
a. 44.44% b. 50% c. 55.55% d. 60%
A: duty cycle is defined as high time by total time, 5/9 is 55.55%. Ans: c.

Q: The power spectral density of an output bit stream has peaks around 1GHz, 3GHz and 5GHz. Assume that the biggest peak has an amplitude of Pc dbW/Hz and the total power is Tc dbW. After this measurement, a 1% spread spectrum clocking is turned on and new peak amplitude Pssc and total power Tssc are measured. What can you say about the relationship between default clocked and spread spectrum clocked measurements?
a. Pc > Pssc, Tc > Tssc
b. Pc == Pssc, Tc > Tssc
c. Pc > Pssc , Tc == Tssc
d. Pc < Pssc, Tc < Tssc

A: Spread spectrum clock spreads the power in the clock frequency and its harmonics to a small band lower and higher of the peaks. The total power remains almost the same. So, Pc > Pssc and Tc == Tssc. Ans c.

Q: How many flops are needed to produce the repeating pattern 1,2,3,4,3,2,1 2,3,4,3,2,1... ? Assume that one number is output every clock cycle.
a. 2 b. 3 c. 4 d. 5
A: At first glance, it may seem that you need 3 bits because the largest number is 4. Deeper inspections shows 0 is absent. So may be only 2 bits are needed to represent 4 unique values 1,2,3,4 created from 0,1,2,3 by adding one. Even deeper inspections shows that the next state of numbers 2,3 are confusing. When the pattern is increasing the next values are 3,4 and when decreasing the next values are 1,2. So, the pattern generating logic needs one more bit to store the information increasing or decreasing. The final answer is 3bits. Another way to look at this is to note that the sequence repeats after 6 states – 1,2,3,4,3,2. So 3 bits are sufficient to represent 6 states.
Ans: b

Q: A clock generator is assumed to be perfect. It produces rising edges at 0ns, 1ns, 2ns,.. and falling edges at 0.5ns, 1.5ns, 2.5ns,.. Suppose a jitter of +/-100ps is added to the clock, what is not the time at which a rising edge occurs. Assume the first edge is fixed at 0ns.
a. 10.1ns b. 2.9ns c. 3.5ns d. 1.1ns

A: A 100ps jitter on either side of the perfect edge position of 0,1,2,3.. is a 0.1ns max deviation. So, 3.5ns is not a proper rising edge position. Ans: c

Q: In which of the products would you use triple modular redundancy?
a. Automotive infotainment system
b. Automotive adaptive radar cruise control
c. Smartphone LCD backlight control
d. Direct To Home Satellite TV receiver

A: Suppose the infotainment in a car fails or the smartphone LCD backlight fails or the DTH TV fails the impact is minimal compared to if cruise control fails and the car fails to slow down when a car in front brakes. Ans: b

Q: There is a RISC-V RV32I core available that has no multiplication and division instruction. What can you say about using user level C code that wants to use the multiplication operation.

a. can't do multiplication. For that you need RV32IM core.
b. multiplication can be used but needs additional software library
c. multiplication can't be done because logarithm instructions are needed
d. multiplication can be done directly using subtraction instructions

A compiler can implement more complicated functions with optimized programs using the simpler instructions of the target processor architecture. In the case of RV32I architecture multiplication will be done with shift and add operations. When the compiler encounters the multiplication operator in software code, it calls the multiplication function coded in the software library supplied during compilation along with the main user code. Ans: b

Q: You have a RISC-V RV32IM Processor core with combinational multipliers and dividers. The core has a max frequency of 100MHz and is targeted for an always AC power connected WiFi router application. You want to port the core to a kid's laptop CPU application working at 10MHz and running on 1.5V batteries. What transformations may be recommended for the porting process?
a. Increasing pipeline stages in the multiplier
b. Changing the combinational multiplier to serial implementation
c. Using many parallel instances of the multiplier to increase performance
d. Using higher performance logic families for core implementation

A: For a lower performance and power limited battery operated toy use case, it is recommended to use a serial implementation to lower power, cost and required die area. Ans: b

Q: Your company makes an SoC targeted for blockchain applications. You have bought an encrypted RTL IP core from a design house that does Bitcoin mining. What can you say about the core?
a. can't pipeline the core
b. can't synthesize the core
c. can't simulate the core
d. cannot use more than one instance of the core in the SoC

A: An encrypted core can be synthesized if the encryption meets the synthesis tool's criteria. Only the user will not see the unencrypted code, the synthesis tool reads all information necessary for logic synthesis. The core can obviously be simulated. Based on the license conditions you may or may not instantiate more than once. You certainly cannot pipeline because the source code is not available.

Ans: a

Q:
You are a product definer in a large chip company. You are defining 4 products.
Chip1: A wireless communication interface chip for military fighter jet and military satelite.
Chip2: A civilian ocean bed temperature sensor powered by energy harvested from ocean currents. The sensor keeps collecting energy and stores in a capacitor. When enough energy is collected, the sensor takes a temperature measurement and transmits the result to a floating buoy.
Chip3: A toy for kids shaped like toy that would repeat what you say in a funny voice. The toy has no wired or wireless communication interface to external world available to user.
Chip 4: An ethernet interface chip for 100MW power substation.

Which of these chips would need some kind of cryptographic design core?
a. Chip 2 b. Chip 3, 4 c. Chip 1, 2 d. Chip 1, 4
Going by just the information given, we can speculate that a military communication chip needs cryptographic protection. An ocean bed sensor is hard to find and not much use to hack. A kid's toy could be hacked, say, to spy on the kids. But, without a communication interface, hacking the toy would need the hacker to visit the kid's house and physically hack it. An ethernet chip for a large power substation is a calling card for hackers and sure needs cryptographic protection.

Ans: d

Q
You are creating a deskewing logic for a multi lane SerDes. Assume that the speed of electromagnetic signal in a PCB trace is 1.5e8 m/s. The trace lengths of every lane can vary from 10cm to a maximum of 20cm accross all boards. The data rate is 20Gbps on every lane. You are implementing a deskew logic by storing the data arriving on each lane and then read out the data once all lanes have starting receiving data. How many bytes of storage you would need per lane to deskew the delays across lanes? Assume that except trace length variation, all other characteristics of the chip and board that are relevant to lane skew may be ignore.

a. 1 byte   b. 2 bytes c. 3 bytes d. 4 bytes

The minimum to maximum difference in length is 10cm. EM wave takes 10cm/10e8 m/s = 0.66ns. The datarate is 20Gbps, that is, 20 Giga bits arrive every second. For 0.66ns there would be,
0.66ns x 20 Gb/s = 20 x 0.66 bits ~= 14 bits or 2 bytes.

Ans: b

For the question above, there is an additional uncertainty in the system in the skew across each lane clocks. Say, lane to lane clock skew is 100ps. Now how much storage is needed?

a. 102 bytes   b. 2 bytes   c. 300 bytes   d. 4 bytes

We need to add the clock skew time to the previous equation.

(0.66ns +100ps) x 20Gbps ~= 16 bits or 2 bytes. The answer does not change

Ans: b

Q: Two SerDes protocols are to be analyzed for per pin bandwidth. Protocol P1 uses 5 pins (or lines or wires) per link and uses a special coding that transmits 3 bits per unit interval (UI) of 100ps. Protocol P2 transmits 2 bits every UI of 80ps on a link formed by 3 lines (or wires). Which protocol gets higher bandwidth per pin (or wire/line).

a. P1 > P2   b. P2 > P1   c. P1 = P2   d. can't say without more information

Per pin bandwidth = Total bandwidth / #pins = bits / UI / #pins
P1: 3b / 100 ps / 5 = 6Gbps/pin
P2: 2b / 80ps / 3 = 8.3 Gbps/pin

Ans: b

Q
Which is the correct order of steps in Physical Design of ASICs?
a. Place, full routing, clock tree synthesis (CTS), floorplan
b. CTS, place, floorplan, full route
c. floorplan, place, CTS, full route
d. floorplan, CTS, place, full route

Note this example does show all steps in ASIC PD.
Ans: c

Q: A 2 bit counter outputs the sequence 2,4,8,16,2,4,8,16, ... Suppose the outputs of the logic needs to be registered. How many flops are needed for registering?

a. 2   b. 3   c. 4   d. 5

At first glance, you may think 5 bits are needed to output the number 16. On closer examination, you can notice that the LSB is always zero for 2,4,8,16. So, you need only 4 bits.

Ans: c

Q: I/O driver circuits may use which technique to minimize effects of noise created from the driver switching activity?

a. Increasing switching voltage   b. Slew rate limiting   c. Reducing clock skew between drivers
    d. Physically placing driver instances close to each other

The noise induced into nearby circuits from a switching activity is proportional to the voltage swing and the proximity of the neighbors. Normally reducing clock skew is good for timing. But for switching noise, all edges happening precisely aligned to a low skew clock only makes the noise worse. The high slew rate injects more noiseinto nearby circuits. Imagine a candle burning for a thousand years. It would not damage any nearby buildings. Now, imagine the same amount of energy is released by an explosion in 1 millisecond.

Ans: b

Ans: c

Q: A mode register bit called WSEL selects which mode a processor starts at power up. WSEL=0 is 32b mode and WSEL=1 is 64b mode. The value of WSEL is set through an external bus independent of the processor and an internal FSM independent of the processor core waits for 500ms after WSEL toggles to power up the processor core. Which SDC constraint best describes this use case? Assume that the flop pin that drives the WSEL signal in the netlist is called wsel_reg/q

a. set_case_analysis 0 [get_pins wsel_reg/q]
b. set_min_delay 1 [get_pins wsel_reg/q]
c. set_false_path -from [get_pins wsel_reg/q]
d. set_load 0 [get_pins wsel_reg/q]

It is tempting to set case analysis on the wsel bits. But doing so would mean the other value of 1 is not tested. If you have two modes in your STA scripts with case analysis set to 0 in one and 1 in another then it is ok. But if you don't want to go through the trouble of creating two modes, the false path constraint is the most appropriate.

Ans: c

Q: Which SDC constraint is applicable to primary outputs?
a. set_case_analysis   b. set_load   c. create_clock   d. set_driving_cell

Case analysis is set on a control signal input to the design like mode input, clock constraint is created on input clock ports and driving cel is a constraint set on all inputs. Load constraint is set on output ports.

Ans: b

Q: Recognize this gate

```
if ( x==y )
 out = 0;
else
 out = 1;
```

a. AND   b. NOT   c. XOR   d. XNOR

The logic marks difference in x and y which immediately reminds of XOR.

Ans: c

Q: Recognize this gate

```
if ( x )
  if ( y )
    out = 1;
  else
    out = 0;
else
  out = 0;
```

a. AND   b. NOT   c. XOR   d. OR

The logic requires both x and y to be true for a true output which immediately reminds of AND.

Ans: a

Q: Recognize this gate

```
if ( x )
  out = 1;
else if ( y )
  out = 1;
else
  out = 0;
```

a. AND   b. NOT   c. XOR   d. OR

The logic requires at least one of x and y to be true for the output to be true which immediately reminds of OR.

Ans: d

Q: Which of the following is not a universal gate? Assume that you can tie inputs to logic 0/1 if needed.
a. NAND   b. NOR   c. AND   d. Mux

Ans: A universal gate is one which can be used to make any logic function. Only the AND gate is not universal. An AND gate cannot realize the NOT gate.

Q: You have the option of selecting the spare standard cells to instantiate in the chip. Which of the following mix of gates is not recommended? Note that logic-0/1 tie cells are freely available in all cases.
a. NAND, XOR, MUX, INV b. AND, OR, BUFFER   c. NAND, BUFFER d. XNOR, BUF, MUX

You need at least one two input gate and an inverter or any two input inverting gate like NAND or NOR. With only AND, OR and BUFFER gates, you cannot realize the NAND/NOR/NOT functions.
Ans: b

Or so it seems! In reality, every AND/OR/BUFFER gate in CMOS actually has an inverter hidden inside the circuit. So, if you are hard pressed to find an inverter, you could tap the hidden inverter output from any of the non inverting gates. But, the inverter you will get has no timing information defined that makes your STA hard.

Q: Assume a perfect clock with a period of 1ns and 50% duty cycle. Assume there is a rising edge at 0. If a +/-10% duty cycle error is added what is not a position for a falling edge?

A duty cycle error of +/-10% means the falling edge occurs at 0.5ns (perfect clock) +/- 0.1ns. So, a falling edge can occur at 0.4ns to 0.6ns + N where N is 0,1,2,3,4,5,... The number 2.1 does not fall in this list. Note that the specification +/-10% error means of all the clock instances across many chips the variation is +/- 10%. It does not mean that clock in the same chip varies that much across every falling edge.

a. 2.1ns   b. 3.4ns   c. 100.5ns   d. 2.6ns

Q: High Level Synthesis (HLS) refers to

a. Exporting a gate netlist from SV RTL   b. Creating C code from SV RTL
c. Creating SV RTL from gate netlist   d. Creating synthesizable RTL from C code

The word synthesis was already taken when HLS arrived. The high level possibly means synthesis from a high level language like C.

Ans: d

Q: Using Direct Programming Interface (DPI) you can

a. Execute C programs on chip without any processor core instantiated on chip
b. Execute C code in SV simulation
c. Synthsize C code to RTL
d. Include SV code in C code compilation and execution with GCC.

It would be nice to include SV code in C code compilation with GCC. But feature is not available yet. The direct in DPI refers to using C code directly in SV simulation without to any extra steps.

Ans: b

Q: Suppose 10 bit data is input into a serializer at a clock frequency of 50MHz with 100 active cycles and 10 idle cycles with no data and output data is clocked at 500MHz. The bit width of output data is 1 bit. If there are 1000 active cycles in the output stream, how many idle cycles are needed to match the average rate of input and output data?

a. 1   b. 10   c. 100   d. 1000

There are 110 cycles in one burst of input data with 10 idle cycles. Total bits is 10 * 100 = 1000 b. Total time is 110 cycles/50M = 20us. Data rate = bits/time = 1000b/(110/50M) = 454.5Mbps
Output data rate is same as input. The output clock is 500MHz.
Output data rate = bits / time = 1000b/((active + idle cycles)/clock frequency) =
1000b * 500M/(1000+idle) = 1000b * 50 M / 110
500M/(1000+idle) = 50M / 110
10/(1000+idle)= 1/110
110*10 = 1000+idle
idle = 1100 – 1000 = 100 cycles

Ans: c

Q: STA setup margin (S) with a perfect clock is 200ps and hold margin (H) is 20ps. It is known later that the clock is not perfect but has a cycle to cycle jitter of 100ps. What are the new margins?

a. S=200ps H=20ps   b. S=100ps H=-80ps   c. S=200ps H=-80ps   d. S=100ps H=20ps

Cycle to cycle jitter can be thought of as a variation is the clock period. So, a 100ps jitter reduces the available period for setup time check. Hold time check is insensitive to clock period so remains the same.

Ans: d

SystemVerilog

Q1: Which format specifier would print 5.13e-03 for the following code?
real p = 0.00513;
$display("<format specifier>",p);
a. %d
b. %f
c. %1.2f
d. %1.2e
The scientific notation of mantissa and exponent is printed with %e format specifier. Ans: d

Q2: Which is not a SV keyword?

a. endpackage b. endfork c.endcase d.endtask

A: join is the ending statement matching a fork statement. Ans: b

The mistake will be readily seen on a syntax highlighting text editor.

```
package t ;
endpackage
```

```verilog
module tb ;
initial begin
fork
endfork

case ( 0 )
endcase

end

task xyz ;
endtask

endmodule
```

Q3: Guess the output

```verilog
logic signed [3:0] p,q;
initial begin
  p = -4;
  q = -7;
  q = p + q ;
  $display("p=%d, q=%d", p,q);
end
```

a. p=-4,q=-3
b. p=-4,q=5;
c. p=-7,q=-11;
d. p=-4,q=-2;

A: 4 bit signed number can represent from -8 to +7. The arithmetic on signed number can be confusing. Overconfident of my abilities, I wrote the answer as -2. I had a nagging doubt if that was the correct answer. So, simulated it anyway. Grrr! The right answer is +5. Let me explain. In number representation with limited bit widths overflow looks as if we entered a different world. It makes more sense when we look at how overflows behave in general. Subtracting 1 from -7 gives -8. subtracting again magically jumps to the other extreme of +7. subtracting again gives +6, again gives +5. Ans: b

Q4: In the code clk2 does not output a useful clock because

```verilog
parameter DEL =0 ;
bit clk, clk2;
always #DEL clk = ~clk;
always @(posedge clk)
  clk2 <= ~clk2;
```

a. DEL is set to 0
b. clk,clk2 are not initialized with reset, so will be x always
c. no posedge is produced on clk
d. always should not be used for clock generator, only forever should be used

A: clk and clk2 do not go to x because bit type is auto initialized to 0 at start of simulation time and bit type does not model x and z behavior. So, going 'x' is not the problem here.

The signal "clk" still toggles but the toggles happen in zero time because DEL is 0. You can see that from the messages from the simulator which says zero delay oscillation and the time it prints even after 2 positive edges of the clk2 signal is still 0.

```systemverilog
module tb ;

parameter DEL = 0 ;
bit clk , clk2 ;
always #DEL clk = ~clk ;
always @ ( posedge clk )
 clk2 <= ~clk2 ;

initial begin
  repeat ( 2 ) @ ( posedge clk2 ) ;
  $display ( "Sim time %t" , $time ) ;
end

endmodule
```

run -all
FATAL_ERROR: Iteration limit 10000 is reached. Possible zero delay oscillation detected where simulation time can not advance. Please check your source code. Note that the iteration limit can be changed using switch -maxdeltaid.
Time: 0 ps Iteration: 10000

Always is ok to generate clock signal. Ans: a

Q5: What is the output?

```systemverilog
class myc ;
   static int p;
   logic q;
endclass
module tb;
   initial begin
     int k;
     k=myc::p;
     $display("%d",k);
   end
endmodule
```

a. syntax error because an instance of class myc is not constructed
b. x
c. 0
d. syntax error because the member p is not initialized

A: A static class member is allocated space even before an instance of the class is created. The int type is also zero initialized. So, the answer is 0 and the code is ok. Ans: c

Q6. What is q?

```
module tb ;
wire [ 3 : 0 ] p = 'b1011 ;
wire [ 3 : 0 ] q ;
assign q = { << { p } } ;
 initial begin
   #0;
   $display ( "q = %b" , q ) ;
 end
endmodule
```

a. 'b1011
b. 'b1101
c. 'b1110
d. 'b0000

The {<<{}} is the stream operator, that is, it takes bits from the variable and outputs it in different order. Here, it is reverse order. Ans: b

Q7: What is k?
```
   int k = 2 ** 32 - 1;
```

a. -1, b. 0 c. 4294967295 d. 1

I got stumped by this embarrassingly often. By default, SV treats unsized integers like 32,2,1 in the expression as 32 bit signed numbers. So, the range of numbers that may be expressed using 32b signed only goes upto $2^{31}-1$ on the positive side and 2^{32} becomes 0. Subtracting a 1 reduces to -1.
Ans: a

Q8: What is a possible output of $urandom_range('h1_0000_0100) ?
a. 8 b. 256 c. 4096 d. 32768

The unsigned random number generator function takes a range operand that limits the max number generated. The number 'h1_0000_0100 has a 1 in the 32nd bit index but the function is limited to 32 bits in size. So, the 32nd bit is discarded to give just 'h0000_0100 or 256.
Ans: a,b

```
module tb ;
 initial begin
   repeat (10)
   $display ( "%d" , $urandom_range('h1_0000_0100) ) ;
 end
endmodule
```

run -all
     169
     227

172
61
107
184
238
19
26
152

Q9: Find output

```
module tb ;
  initial begin
    shortint unsigned i=0;
    while ( i!='1) i++;
    $display("%h",i);
  end
endmodule
```

a. 'hF b. 'hFF c. 'hFFFF d. 'hFFFF_FFFF
Shortint is a 16 bit type and '1 stands for all ones. I starts with 0 and then counts up until hitting 16 ones or 'hFFFF.
Ans: c

Q10: In the typical swap code asked in interviews, what pair of a,b values won't work?

```
module tb ;
 initial begin
  my_swap ( 1 , 2 );
  my_swap ( 4 , 12 );
  my_swap ( 7 , 5 );
  my_swap ( 3 , 3 );
 end

task automatic my_swap ( input logic [ 3 : 0 ] a , b );
$display ( "Before swap a = %d , b = %d" , a , b );
a = a + b ;
b = a - b ;
a = a - b ;
$display ( "After swap a = %d , b = %d" , a , b );
endtask

endmodule
```

a. all pairs work   b. 4,12   c. 7, 5   d. no pair works

If you thought 4,12 would fail because 4+12 would overflow 4 bit unsigned number, you are with me. But the simulation shows it works for all values! The secret being that overflow does indeed happen but the final result is still correct even after discarding the overflow bit.

Ans: a

run -all
Before swap a = 1 , b = 2
After swap a = 2 , b = 1
Before swap a = 4 , b = 12
After swap a = 12 , b = 4
Before swap a = 7 , b = 5
After swap a = 5 , b = 7
Before swap a = 3 , b = 3
After swap a = 3 , b = 3

Q11: You want to model the number of people waiting for their turn at a California Department of Motor Vehicles office, using a serial number for every person entering the office in a day. What datatype declaration below may be more appropriate? Assume that a DMV serves general public about automobile registration and driving license in the state of California, USA.

```
a. byte dmv_customers0;
b. int dmv_customers1[$];
c. int dmv_customers2[100];
d. logic [7:0] dmv_customers3;
```

A: The people waiting in a government office is generally served in first come first served basis. A queue is the most appropriate datatype to use, dmv_customers[$].
Ans: b

Q12: You are using semaphore to synchronize multiple threads in your code by waiting for the semaphore to be signaled. Which method of the semaphore works for this purpose by waiting on the semaphore? Assume the name of the semaphore object is smp.

```
a. smp.wait()
b. smp.put()
c. smp.try_get()
d. smp.get()
```

A: smp.get() is the blocking call that checks if a semaphore is available or not. The put() method releases a semaphore. The try_get() is same as the get() except that it does not wait for the semaphore to be available. There is no wait() method defined for SV semaphore object.
Ans: d

Q13: The following code is not recommended for synthesizable RTL code. Why?

```
typedef enum {ST0,ST1,ST2} fsm_st ;
fsm_st state,next;
```

a. enum makes the code hard to read
b. by default enum is 2-state datatype
c. typedef makes the code less reusable
d. enum type cannot be used to represent flops

A: enum is by default a 2-state type without x and z modeling. In simulation if reset is missed, the FSM will nicely start at ST0. In the chip, however, it will start with any of 0-3 values corresponding to ST0-2 and an unknown state. Instead you need to use, enum logic [1:0] {…}

Ans: b

Q14:
```
module tb ;

int a ;

initial
begin
 a = 0 ;
 fork
  begin #10ns ; a ++ ; end
  begin #20ns ; a ++ ; end
  begin #30ns ; a ++ ; end
 join
 $display ( "%d" , a ) ;

 a = 0 ;
 fork
  begin #10ns ; a ++ ; end
  begin #20ns ; a ++ ; end
  begin #30ns ; a ++ ; end
 join_any
 $display ( "%d" , a ) ;

 a = 0 ;
 fork
  begin #10ns ; a ++ ; end
  begin #20ns ; a ++ ; end
  begin #30ns ; a ++ ; end
 join_none
 $display ( "%d" , a ) ;

end

endmodule
```

a. 0,1,2   b. 0,1,3   c. 3,1,0   d. 3,1,2

SV offers 3 options when forked processes rejoin. The first is the join that waits for all the forked processes to complete. So, all the a++, three in total execute before printing a value of 3. Next, the join_any option waits for at least one process to complete. In this case, after the first 10ns time lapses one a++ executes and then prints the value of 1. The join_none doe not rejoin the forks at all, they are all freely executing from here. So, no a++ statement gets executed before printing the value of a of 0.

Ans: c

Q15: You are using a Linux environment for SV simulation. In your code you want to use the grep Linux command. Which utility task of SV will let you do that?

a. $clog2
b. $os_cmd
c. $system
d. $shell

A: The $system utility task lets call Linux commands. It is a handy method to bring in complex functionality to SV testbench very easily. Unfortunately, Vivado does not seem to support $system();

```
module tb ;

initial
begin
  $system("grep task *");
end

endmodule
```

ERROR: [XSIM 43-3122] "<git repo dir>/dsn_verif/quiz/q15.sv" Line 5. system system task is not supported.

Q16: Identify the following code

```
module tent ( input rope,input screw, input nail,output pencil);
assign
pencil
=
rope
?
screw
:
nail
;
endmodule
```

a. inverter
b. mux
c. demux
d. xor

A: In SV, whitespaces do not affect the meaning of the code. So, look at more important things like SV operator or keyword. Here the "?" operator and the input and output count give away the mux.

Q17. Count the flops used in the final optimized netlist. Note that the variable with the name q is not used in any other expression other than to derive outen.

```
logic [5:0] q,din;
logic outen,clk;
always_ff @(posedge clk) begin
  q <= din;
  outen <= ( q >= 32 );
end
```

a. 2

b. 7

c. 1

d. 3

A: At first glance, the answer seems like the q bits and the outen bit, but, you can see that a comparison with 32 is nothing but only the 5th index of q. If it is one, q is greater than or equal to 32. The synthesis tool will use this property and remove the flops for q[4:0]. Total flops used is 2.

Ans: a

Q18. An FSM is coded as follows

```
parameter S0=0,S1=1,S2=2;
logic [1:0] state,next;
always_comb begin
  next = state;
  case(state)
    S0: next = S1;
    S1: next = S2;
    S2: next = S0;
  endcase
end
```

Assume state is properly reset and gets next state every clock cycle. What happens when state accidentally enters the state 3?

a. The FSM remains in state 3

b. Unpredictable

c. FSM goes to S0

d. FSM goes to S2

A: If you answered unpredictable, you are almost right. Most FSMs without handling of unused states can become unpredicatable if it enters unused states by error. In this case, the next = state line forces the FSM to maintain the state 3.

Ans: a

```
module tb;

bit clk;

parameter S0=0,S1=1,S2=2;
logic [1:0] state,next;
always_comb begin
  next = state;
  case(state)
    S0: next = S1;
    S1: next = S2;
```

```
      S2: next = S0;
   endcase
end

always @(posedge clk) begin
   state <= next;
end

initial begin
   state = S0;
   repeat (10) begin
     clk = 0 ; #1; clk=1; #1;
   end
end

initial begin
   repeat (4) @(posedge clk);
   force state = 3;
   repeat (1) @(posedge clk);
   release state;
end

initial $monitor ("State %d, time %t", state, $time());

endmodule

run -all
State 0, time            0
State 1, time          1000
State 2, time          3000
State 0, time          5000
State 3, time          7000
```

Q19: Which datatype is not recommended for RTL code?
a. **logic**
b. **integer**
c. **struct**
d. **byte**

A: byte is a two state data type using only 0 and 1 without modeling x and z. Using it may cause simulation synthesis mismatch. Logic and integer or 4 state types modeling x and z. Struct can be limited to using only 4 state types for RTL code.

Ans: a

Q20
What is a possible output of a successful randomization of the class shown?

```
program tb;

class automatic t;
  rand int unsigned i;
  constraint mycon1 { i > 1 && i < 16 ;};
```

```systemverilog
  constraint mycon2 { myfn(i) == 0 ; };
  function int myfn (input int i);
   for ( int j=2;j<i;j++) begin
    if ( i%j == 0)
      return -1;
   end
   return 0;
  endfunction
endclass

initial begin
  repeat (100) begin
   automatic t var1=new();
   if ( var1.randomize()==1)
     $display("Randomization result i = %d",var1.i);
  end
end

endprogram
```

a.12   b.13   c.14   d.15

The constraints shown is actually for generating prime numbers. The idea is visible on the i%j constraint. So, 13.
Ans: b

Some run results:
Randomization result i =   13
Randomization result i =   2
Randomization result i =   3
Randomization result i =   5
WARNING: File: /home/po/RTL_design/dsn_verif/quiz/q20.sv Line: 19 : randomize failed to meet constraint.

Note the warning message on failure of randomization. It is one thing to create correct constraints and quite another to design the constraints to be solvable by the simulator!

Q21: What is the initialization of the array k?

```systemverilog
module tb;

int k[2][2];
function int myfn [2][2] ();
 foreach (myfn[i,j])
   myfn[i][j] = i*j;
endfunction

initial begin
 k = myfn();
 $display("%3d\t%3d\n%3d\t%3d",k[0][0],k[0][1],k[1][0],k[1][1]);
end
```

endmodule

a. '{'{0,0},'{0,0}}   b.'{'{0,0},'{0,4}}   c. syntax error   d.'{'{4,0},'{0,0}}
I think SV does not allow functions to return array types in a straight way.

Ans: c

If you indeed want to return an array type use typedef to wrap the array into one single item.

```
typedef int arr2x2_t [2][2] ;

arr2x2_t k;
function arr2x2_t myfn() ;
  foreach (myfn[i,j])
    myfn[i][j] = i*j;
endfunction
```

Q22: A bidirectional data bus can be driven by the master or by the slave. But at no time does the master and slave drive the bus simultaneously. Which datatype is best suited to model the physical bus interconnect?

a. **bit**   b. **logic**   c. **wire**   d. None of these

The wire type is ideally suited to model these kinds of stuff. It can be driven by multiple drivers, it supports 'z' or high impedance state and even resolves multiple driven values using the notion of drive strength. The logic type comes close but lacks the drive strength modeling feature.

Ans: c

Q23: Identify the muliply accumulate (MAC) operation in the following code?

a. sum ***= (a+b);**   **b.** sum **+= (a*b);**   **c.** sum **=** a*sum **+ b;**   **d.** sum ***= (a*b);**

As the name suggests, the MAC operation is first multiply and then accumulate or add to an existing value. Only sum = sum + a*b matches the MAC definition. The MAC operation is regularly used in DSP applications like FIR filter.

Ans: b

Q24: Which sequence does not occur when the following is executed?

```
program tb;

class automatic myc;
  randc byte s;
  constraint s_in { s inside {5,9,3,12} ;};
endclass

initial begin
  automatic myc c=new();
```

```
  repeat (4) begin
    if ( c.randomize()==1)
       $write("%3d",c.s);
  end
end

endprogram
```

a. 12, 9, 3, 5   b. 5, 3, 3, 9   c. 12, 3, 5, 9   d. 9, 3, 5, 12

The randc method of randomization first creates a permutation of the random variable and steps through the permutation. In this case, there is only one 3 in the list, so 3 cannot repeat within the same permutation. Example from one execution:

run -all
  5 3 9 12

Ans: b

Q25: What is the probability of k getting the value 2?

```
program tb;

class myc;
  rand bit [1:0] k;
  rand bit [2:0] m;
  constraint con1 { (k%2 == 0) -> (m==0) ;};
endclass

initial begin
  int cnt, cnt2;
  repeat (100000) begin
    automatic myc c=new();
    if ( c.randomize()==1) begin
      //$write("%3d ",c.k);
      cnt++;
      if ( c.k==2) cnt2++;
    end
  end
  $display("Frequency of 2 = %f",1.0*cnt2/cnt);
end

endprogram
```

Suppose one more constraint,
```
  constraint con2 { solve k before m ;};
```
is added, what is the probability of k getting 2?

Suppose the con2 is reordered as shown,
```
  constraint con2 { solve m before k ;};
```
what is the probability of k getting 2?

I think I know the answer, but simulations don't agree with my understanding!

Q26: Output?

```
module mym;
  initial $display("Printing from %m");
endmodule

module tb;
  mym i[0:1] ();
endmodule
```

a. Syntax error, %m has no matching variable
b. Printing from tb.i[0]
   Printing from tb.i[1]
c. 0 0
d. Syntax error, i[0:1] is not a valid instance name

You can use the array range style [n:m] to instantiate many instances of the same module. It is very handy to instantiate repeating structures in chips. The %m simply means this instance or the name of the instance where the $display is located and does not need any other variable like %d does. The example just prints the names of the two instances. Note that the print message from 0 or 1 can come first and there is no gurantee that only the print from 0 comes first.

Ans: b

Q27: What is the frequency of 5?
```
program tb;

class myc;
  rand int k;
  rand bit [2:0] m;
  constraint con1 {
    k inside {[3:8]} ;
    k dist {3:=1,4:=1,5:=2,6:=2,7:=3,8:=3};
  }
endclass

initial begin
  int cnt, cnt2;
  repeat (100000) begin
    automatic myc c=new();
    if ( c.randomize()==1) begin
      //$write("%3d ",c.k);
      cnt++;
      if ( c.k==5) cnt2++;
    end
  end
  $display("Frequency of 5 = %f",1.0*cnt2/cnt);
end

endprogram
```

a. 1/6   b. 5/6   c. 1/12   d. 2/6

The distribution statement is to be read in two parts. Add up all the weights, the number that appear after := to get the total weights. Here its 12. Then the weight of the number 5 in question is 2. So, the frequency is 2/12.

Ans: a

Q28: What is the frequency of 5 if?

```
constraint con1 {
  k inside {[3:8]} ;
  k dist {3:=40,4:=50,[5:8]:/10};
}
```

a. 10%   b. 1/130   c. 5%   d. 2.5%

The distribution statement is to be read as 3 with weight of 40, 4 with weight of 50 and 5,6,7,8 combined weight of 10. The 10 is shared equally. So, 5 gets a weight of 2.5 and the total weights is conveniently set to 100 which makes 2.5/100 as the frequency

Ans: d

Q29: Which number does not occur if the code is executed?

```
program tb;

int i;
initial begin
  repeat (60*10) begin
    do
      i = $urandom_range(60);
    while ( i==10 || i==20 || i%7==0 );
    $write("%4d",i);
  end
end

endprogram
```

a. 44   b. 56   c. 12   d. 25

Sometimes you may use a do – while loop to do constrained random. The execution stays in the loop if the number generated is either 10 or 20 or a multiple of 7. The choice of 56 is divisible by 7 and does not occur.

Ans: b

Q30: Output?

```
program tb;

bit [10:0] k;
bit [4:0] m;
initial begin
  k = 'b1010_1100;
  m = k[7:-3];
  $display("%b",m);
end

endprogram
```

a. 00101   b. 10101   c. 00100   d. Syntax error

The :- is misleading. It feels as though we want to select 3 bits down from 7th position. Actually I got the output of 00111! There is also a warning message saying index -3 is out of bounds. I think the simulator is trying to extract the bit 7,6,5,...0,-1,-2,-3. If the intention was to select 7,6,5 then the operator -: should be used, the minus comes first.

Q31: What is the name of the matching circuit?

```
module mym (input pqr, input lmn,input [2:0] stu, output logic [2:0] cake); always
@(posedge lmn,negedge pqr) if (!pqr) cake <=0; else cake <= stu; endmodule
module tb; mym i0 (); endmodule
```

a. Counter   b. 2 stage synchronizer   c. latch   d. One cycle delay

The always @posedge statement shows that this is an edge sensitive logic. So, no latch. The else input gets output behavior means no counter. A synchronizer has a shift register type of structure that is missing here. A cycle delay is the best match

Ans: d

Q32: I attempted to create a covergroup that covers for prime numbers. I could not get it to work. Either, I don't understand covergroup well or Vivado does not support all features of covergroups. The get_coverage function is not working in the following code. Can you fix it?

```
`timescale 1ns/1ps
module tb;

byte unsigned i;
bit clk;

covergroup cg ( ref byte unsigned i) @(posedge clk) ;
  n1: coverpoint i
  {
    bins pn[] = i with (myfn(item)) ;
  }
endgroup
```

```
always begin #1 ; clk = ~clk ; end

cg cg0 = new(i);

initial begin
  for ( int i=0;i<40;i++) begin
    @(posedge clk);
  end
  $finish;
end

final begin
//  $display("Coverage for pn 7 = %f",cg0.n1.pn[7].get_instance_coverage());
   $display("Coverage for pn 7 = %f",cg0::get_coverage());
end

function automatic bit myfn ( input byte unsigned i );
  for (int j = 2 ; j < i ; j++ )
    if (i%j==0) return(0);
  return(1);
endfunction

endmodule
```

Q33: Output?

```
`timescale 1ns/1ps
module tb;
bit rstn,clk;
always begin #1 ; clk = ~clk ; end
integer q;
always_ff @(posedge clk, negedge rstn) begin
  if ( rstn )
    q<=0;
  else
    q<=q+1;
end

initial begin
  rstn=0;#10;   @(posedge clk); rstn=1;
  repeat (10) @(posedge clk);
  $display("%d",q);
  $finish;
end

endmodule
```

a. x   b. 0   c. 9   d. 10

I have committed this typo quite often and wondered what went wrong. The rstn is missing the not
operator. When the rstn falls the negedge event gets triggered and the loop is entered but the if(rstn)
condition evaluates false and the else part is executed. When reset is deasserted or rstn goes high on

every clock posedge the if(rstn) condition evaluates true and q is always assigned 0. To get correct counting you need to replace if(rstn) with if(!rstn).

Ans: b

Q34: Output?

```verilog
`timescale 1ns/1ps
module tb;
  int u,t,s;
  always @(u,t,s) begin
    s <= u+t;
    $write("%d ",s);
  end
  initial begin
    u=10;
    #1;
    $finish;
  end
endmodule
```

a. 10 10   b. 0 10   c. 0 0   d. x 10

Ans: You got to be careful when mixing the non blocking operator <= with other functions. The s <= u+t and $write are evaluated simultaneously when u is changed. The first write does not take the updated s. After s is update, the always @ triggers again on the change in s and write prints the updated s value of 10 this time.

Ans: b

Q35: You want to create a pulse output for an input signal. What is wrong in the following code?

```verilog
module tb;
  function automatic void put_pulse ( ref a );
    a = 0 ; #10ns;a=1;#10ns;a=0;
  endfunction
endmodule
```

a. no **ref** arguments allowed in functions
b. **function** needs a **return** statement
c. no delays allowed in **function**
d. **functions** always need a **return** type, **void** is not allowed

Functions can use reference argument and end without an explicit return statment and functions can also return nothing. But SV disallows time based stuff inside functions like #delays.

Ans: c

Q36: Output?

```
`timescale 1ns/1ps
module tb;
bit clk;
always begin #1 ; clk = ~clk ; end

bit [9:0] cnt;
always_ff @(posedge clk) cnt <= cnt + 1;

initial begin
  wait (cnt==1000);
  repeat (30) @(posedge clk);
  @(negedge clk);
  $display("%d",cnt);
  $finish;
end

endmodule
```

a. 1030   b. 1029   c. 5   d. 6

The cnt variable is only 10 bits wide so max allowed is 1023. You should understand the 1000+30 as (1000+30)%1024 which gives 6. The question has more quirks than meets the eye at the first reading. After the 30 posedges pass by in the repeat statment, the value of 6 is scheduled to be assigned to the cnt variable. Without the @(negedge clk) statment the display statement also runs simultaenous to the cnt getting 6, so the display statment shows the previous value of 5. including the negedge statment lets the cnt assignment complete and read the proper result.

Ans: d

Q37: Output?

```
module tb;
logic q, qn;
assign #1 q = ~qn;
assign #1 qn = ~q;
initial begin
  #10;
  $display("%b",q);
end
endmodule
```

a. 0   b. 1   c. X   d. Z

The logic models a latch but there is no initialization of the latch. The state is unknown.
Ans: c

Q38: Output?
```
module tb;
```

```
logic q, qn;

nand #1 n0 (qn, q,q);
nand #1 n1 (q, qn,q);

initial begin
  #10;
  $display("%b",q);
end

endmodule
```

a. Steady 0   b. Steady 1   c. Steady X   d. Oscilating between 0 and 1

If you draw the circuit on paper or connect physical logic gates as in the code, the q would settle to a steady 1 because the inputs q and qn are always different. But in SV simulation an !(x & x̄) evaluates to x! The simulator does not recognize that x̄ is the inverse of x. It simply does !(x & x) which is x.

Ans: c

Q39: What gate is mygate?

```
module mux (input i0,i1,s,output y);
  assign y = s ? i1 : i0;
endmodule

module mygate(input a,b, output y);
  mux i0 (.y, .s(a), .i0(b), .i1(1'b1));
endmodule
```

a. NAND   b. NOR   c. AND   d. OR

If you create a truth table for the inputs and calculate the outputs you can see the OR gate.
Ans: d

```
module tb;

logic a,b,y;
mygate i0 (.a, .b, .y);

initial
  for(int i = 0 ; i < 4; i++) begin
    {a,b} =i[1:0];
    #0;
    $display("%b %b %b",a,b,y);
  end
endmodule

run -all
0 0 0
0 1 1
1 0 1
```

Q40: Output?

```
module tb;

enum {MOORBY, SUTHERLAND, RITCHIE} e1;

initial begin
  int k;
  k = 2 * SUTHERLAND + RITCHIE - MOORBY;
  $display("%d", k);
end
endmodule
```

a. x   b. 4   c. Syntax error : enum needs casting to int   d. 0

Enumerated types are internally assigned integer values from 0. So, MOORBY, SUTHERLAND, RITCHIE get 0,1,2. In expressions they can be freely used without any casting.

Ans: b

Q41: Output?

```
module tb;
initial begin
  int a[]='{30,31,32,33};
  foreach (a[i])
    if ($onehot(a[i]))
      $display("%d", a[i]);
end
endmodule
```

a. 30   b. 31   c. 32   d. 33

The onehot utility function returns true if the bits in the input has only a single logic-1. Powers of 2 are onehot.

Ans: c

Q42: What cannot be defined inside a SV package?

a. **function**   b. **module**   c. **task**   d. **class**

For some reason, SV architects have chosen to forbid defining modules inside a package.

Ans: b

Q43: I was trying to get only the intersect assertion to always pass and let the other 3 assertions fail at least once. But, I am unable to get it to work. Can you fix it?
```
`timescale 1ns/100ps
```

```
module tb;
bit [9:0] km='b0000111000,kn='b000001100;
bit m,n,clk;

always clk = #1 ~clk;

initial begin
  int cnt;
  repeat (10) @(posedge clk) begin
   m = km[cnt]; n=kn[cnt++];
    $display("m %b, n %b,count %d",m,n,cnt);
  end
  $finish;
end

sequence pulse_m;
  @(posedge clk) ~m ##  m  ##[1:3] ~m;
endsequence

sequence pulse_n;
  @(posedge clk) ~n ##  n ## [1:3] ~n;
endsequence

sequence s1;
  @(posedge clk) pulse_m intersect pulse_n;
endsequence

a1: assert property (@(posedge clk) s1);
a2: assert property (@(posedge clk) m ##1 n);
a3: assert property (@(posedge clk) m ## [0:4] n);
a4: assert property (@(posedge clk) m ## [0:$] n);

endmodule
```

Q44:

In the following code which usage is a syntax error?

```
module tb;
  int n[5]; byte m[3]; logic [2:0] s; integer k[5];
  initial begin
   $display("%d",n.sum());
   $display("%d",m.sum());
   $display("%d",s.sum());
   $display("%d",k.sum());
  end
endmodule
```

a. n.sum()   b. m.sum()   c. s.sum()   d. k.sum()
The .sum() method is defined for arrays or more technically for the unpacked arrays (or the arrays with
the [] bracket after the variable name). It is not defined for bit vectors like logic [n1:n2] var or
technically packed arrays (where [] appears before variable name).

Ans: c

Q45:
Output?
```
`timescale 1ns/1ps
module tb;
bit clk;
always begin #1 ; clk = ~clk ; end

bit [2:0] cnt;
always_ff @(posedge clk) cnt <= cnt--;

initial begin
  repeat (4) begin
  @(negedge clk);
  $write("%d",cnt);
  end
  $finish;
end

endmodule
```

a. 0123   b.1234   c. 7654   d. 0000

If you guessed the answer to be 7654 you are almost right. The post decrement cnt-- is a dangerous operator when used with the non blocking assignment. In this case the cnt always gets assigned the same value of 0. If you want 7654 output you need to use cnt <= cnt – 1;

Ans: d

My **guess** (please check for yourself) is that the cnt <= cnt-- should be interpreted as
cnt <= cnt;
cnt = cnt--; // cnt = cnt – 1;

Further substitution,
cnt <= 0 ; // scheduled to be assigned after all the events at this delta delay is completed, not immediately assigned
cnt = 0 – 1 (or 7); //assign immediately in this delta delay

Substituting again,
cnt is 7 now but the scheduled cnt = 0 from previous step overwrites, so
cnt = 0;

In general avoid the var--, var++ operations inside always_ff blocks. It is ok to use these in always_comb blocks. For my level of coding skills, I avoid complex use of the increment decrement operators like -
var2 = var1++ * var3-- ; // to much ++-- that I don't understand!
I split that into -
var2 = var1 * var 3;
var1++;

var3--;

Q46:

Output ?

```verilog
module tb;

logic [1:0] out,in;
initial begin
   in = 3;
   case(in)
      00: out = 0;
      01: out = 1;
      10: out = 2;
      11: out = 3;
   endcase
   $display("%d",out);
end

endmodule
```

a. 1   b. 2   c. 3   d. X

It may appear that the case statement would match the in == 11 or the 4th branch. The output will be x because the 11 is actually 32'b1011 which does not match with input 2'b11. No branch matches the input and the out variable is unassigned.

Ans: d

If your intention was to compare two bit binary numbers remember to use the <size>'b<value> format.

```verilog
   case(in)
      2'b00: out = 0;
      2'b01: out = 1;
      2'b10: out = 2;
      2'b11: out = 3;
   endcase
```

I know this point well but still continue to make that mistake and wonder what went wrong. I reiterate the idea of using Lint as a verification fail debug method.
You may find more examples of such seemingly correct but semantically buggy code in the Verilog gotchas book mentioned before.

Q:

Consider the FSM logic

```verilog
enum {S0,S1,S2,S3,S4,S5,S6,S7} state, next;
case (state)
  S0: next = S1;
```

```
  S1: next = S4;
  S2: next = S1;
  S3: next = S5;
  S4: next = S3;
  S5: next = S6;
  S6: next = S2;
  default: next = state;
endcase

always_ff @(posedge clk, negedge rstn)
  if ( !rstn )
    state <= S0;
  else
    state <= next;
```

Given that the logic is working without any fault from external or internal influences and the current state is S5, which state cannot be entered?
a. S0   b. S1   c. S4   d. S7
If you draw the state diagram of the FSM you can see that it goes like this
0,1,4,3,5,6,2,*1,4,3*,...
You can see that there is no S0 in the repeating sequence after S5.

Ans: a

Suppose some transient noise causes the FSM state flops to take any 3 bit binary value, which state cannot be exit if entered?
a. S0   b. S1   c. S4   d. S7

The default case assigns the present state to next state which means S7 which is not explicitly mentioned will just stay stuck. In production designs it is preferrable to use next = S0 (the first state or reset state) to deal with unused states.

Ans: d

Q: Pick the non synthesizable construct

a. **randomize()**   b. **for**   c. **always_ff**   d. **function**

The randomize is primarily for verification. Even randomize could be synthesized using a sophisticated logic driven with a random number generator, but, that level of synthesis capability is presently impossible.

Ans: a

Q: Which is not a SV keyword?

a. **with**   b. **constraint**   c. **goto**   d. **assert**

Let me tell you two ways. Cheat code first! Copy the words into a SV syntax highlighting text editor. The one not highlighted is not the SV keyword. I got goto not highlighted. The sober way is to

remember that SV language architects did not want to carry on with the goto keyword caused mess in C language. Goto was dumped on purpose.

Ans: c

Q: Which concept is not in SV?

a. Object Oriented Programming (OOP)   b. Procedural coding
c. Pointers for all datatypes   d. Boolean or binary datatype

SV language architects probably did not want to get into another messy feature of C, that is, pointers.

Ans: c

Q: What is not possible to do with packages?

a. import another package in a given package
b. parameterize a package while importing the package in a module
c. define classes, functions, tasks, parameters
d. import a package into an initial block

A package can even be imported into a smaller scope like initial block. But a package cannot be parameterized during import like how a module parameter can be set during module instantiation.

Ans: b

Mixed Signal

Q: There are 4 ADC types that you are considering for your new design. Which type will you choose if your application needs tolerance to comparator voltage offset.

a. 1.5b per stage pipelined ADC.
b. 1bit per stage pipelined ADC
c. Flash ADC
d. Counter ADC

All ADCs are susceptible to the offset voltage in their comparators. But, 1.5b or any other overlapping comparator range pipelined ADCs like 2.5b per stage have some level of immunity to voltage offset. This is because there are two ADCs to resolve the range that is normally done with only one comparator in a 1b per stage architecture. The added redundancy in the comparator output bits is used in a digital correction block to correct for offset up to some limit.
Ans: c

Q: A PLL block diagram shows an input divider reducing the reference clock frequency by N and a feedback divider reducing the output clock frequency by M. Suppose reference clock frequency is 128MHz, N=2 and M=20. What is the output clock frequency?

a. 3.2MHz
b. 64MHz
c. 12.8MHz
d. 1280MHz

A: Simple PLLs use the formula, output F = ref * M / N. So, 128 * 20/2 = 1280MHz
Ans: d

Q. The oversampling ratio of a N-bit output Sigma Delta ADC is 1024. By changing the oversampling ratio alone, you want to increase the resolution from N to N+2. What is the new oversampling ratio needed?

a. 1026
b. 2048
c. 512
d. 4096

A: Sigma Delta ADC achieves high resolution by processing the bit stream output over a big time window. The time window size to the bit resolution is logarithmic. In other words, to increase bit resolution one needs to increase the time window exponentially. So, $2^{(N+2)}$ is $2^2 * 2^N$ gives 4 times the previous oversampling ratio.

Ans: d

Q: A 12b ADC implemented using a hybrid architecture with pipelined ADC for MSB 8bits and flash ADC for LSB 4 bits. The pipelined part is implemented as eight 1 bit per stage ADC and the flash is implemented as a regular 4b flash ADC. What is the latency of the hybrid ADC in clock cycles?

a. 8   b. 4   c. 12   d. 20

Flash ADC is good for small number of bits and pipelined is good for speed. Pipelined ADC generally takes 1 cycle per stage and flash takes 0 cycles. So, PADC 8 cycles + FADC 0 cycles = 8 cycles. In practice, the lantency could be 1 or 2 cycles longer if the analog signals are sampled and held to provide extra time. But the 8+1 or 2 = 9 or 10 option is missing.

Ans: a

Q: A transceiver system uses CDR to get a version of the transmitted clock. Assume the transmitter outputs at 1Gbps using 8b10b coding for sometime and the receiver recovered the transmitter clock properly. After about 10s of transmission, the transmitter turned off the transmission and the toggling on the output wires stopped. What will happen to the recovered clock at the receiver after one hour?

a. loses phase and frequency synchronization
b. maintains frequency but loses phase synchronization
c. maintains phase and frequency synchronization
d. maintains phase but loses frequency synchronization

Any clock source always drifts a bit around the nominal frequency. A 100MHz clock at time 0 may drift to 100.001MHz after some minutes. This causes the position of the rising and falling edges to be different compared to the recovered clock at the receiver. Both the transmitter and the receiver logic operate on clocks derived from drift prone sources. It is the CDR that keeps the synchronization against the backdrop of such clocking errors. You may wonder why not save the state of the synchronization in a digital value and then maintain that in the CDR till the transmitter starts again. The problem is that the transmitter moved to different phase and frequency in the meantime. The frequency drift is a random process. So, you cannot predict it a long time into the future. So, the answer is receiver loses synchronization completely.

Ans: a

Q: Which of properties are true for a 1bit per stage 8b pipelined ADC
a. Can correct gain errors of residue amplifiers
b. Can correct for errors in subtraction circuits that produce residue
c. LSB stages needs less accuracy than MSB stages
d. Can correct comparator offset error

Pipelined ADC by itself does not correct errors. Only the N.M bits per stage (like 1.5/2.5) type ADC which has 0.5 or more bits of overlap correct for comparator offset error. As the residue passes from one stage to another the effect of circuit errors dimnishes. So, the MSB circuit stages are more important.

Ans: c

Q: What is false about PLL vs DLL?

a. PLL can multiply input clock frequency. DLL can't.
b. PLL and DLL architectures can create 0, 90, 270, 180 degree versions of the input clock
c. DLL can be implemented without an oscillator
d. PLL and DLL can filter jitter on input clock

Generally, a DLL does not have an oscillator whose output is synchronized to the input clock. The implication is that the clock edges of the output clock are derived from the clock edges of the input clock, so no jitter reduction or frequency multiplication are possible.

Ans: d

Additional Resources

Graduate Aptitude Test in Engineering is a competitive exam conducted by higher education institutions in India as an entrance filter for graduate courses. We can leverage that work as a learning material for chipping. Questions and answers in these domains contain information relevant for chipping. Please ignore questions that are not relevant.

1. Electronics and Communication Engineering

2. Electrical Engineering

3. Computer Science and Information Technology

4. Instrumentation Engineering

Previous year questions

2019 version

http://gate.iitm.ac.in/Anskey19

2018 version

http://www.gate.iitg.ac.in/?page_id=748

2007 to 2015
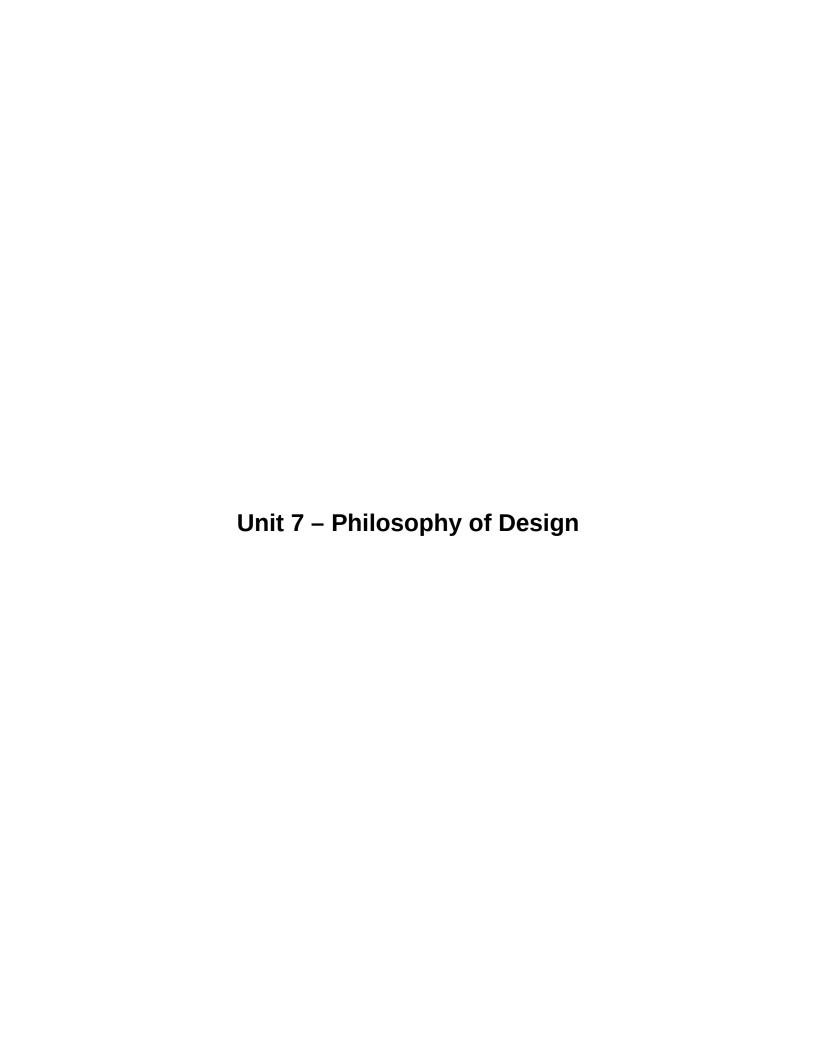
https://www.iitr.ac.in/gate/gatepaper.html

Please google for other years. The universities conducting these exams change every year. But it is typically any Indian Institute of Technology or Indian Institute of Science.

Books

Digital Logic RTL & Verilog Interview Questions, May 8, 2015 by Trey Johnson (Author)

https://www.amazon.com/dp/1512021466

Unit 7 – Philosophy of Design

This chapter talks about the undercurrents in the process of designing chips. These concepts are not very apparent in the diagrams or code, but, profoundly affect the choice of a particular design.

Longevity

The job of chipping mostly involves a lot of desk work typing code. In this industry, back pain, neck pain, wrist pain, obesity, diabetes is a bigger problem than job loss. Health takes a back seat in the heat of project deadlines. A chipper may very well run out of health before he runs out of jobs. I joke sometimes that the size of a chipper's belly tells the amount of experience he has! Then my friend joked that if the size of the belly is too small it means the candidate has hit the gym too often and not worked hard enough! So, he would not hire a fit person :)
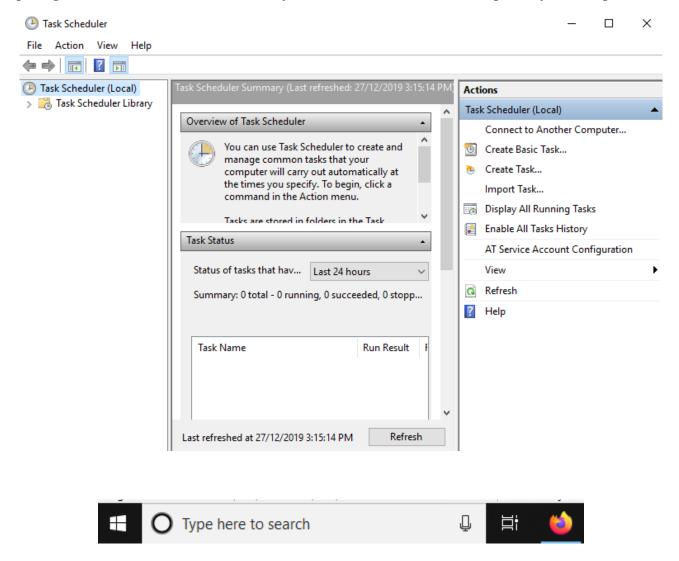
To help chippers maintain focus on health, I have a script that reminds you to take breaks periodically. Try it and you will love it!

You may get this functionality with your smartphone app or a fitness tracker like Fitbit or a pre-built Linux or Windows application.

For Linux users, the following shell script opens a dialog box regularly and forces the chipper to answer the dialog box. You may add this script to the Cron jobs to start at log-on. I did not have much success with cronjobs.

```tcsh
#!/usr/bin/tcsh
set break_period=2700 ;#seconds
while ( 1 )
  zenity --width 1000 --height 500 \
  --question --title="Hi! I am your body" \
  --text="Can you please rest me for sometime?"
  set answer=$? ;#user answer clicked on the dialog box
  if ( $answer == 0 ) then
    sleep $break_period
  else
    exit
  endif
end
```

For windows users: save as .vbs

```vbs
'repetition period in min
RepPeriod = 15

Do While 1
  usersel = MsgBox("Will you rest me, your body? " & RepPeriod,1)
  If usersel = 2 Then
    MsgBox("Remember to restart. Bye")
    Exit Do
  End If
  Wscript.sleep RepPeriod*60*1000
Loop
```

You can use the task scheduler option in the Windows to call this script at log on, so that, you don't have to start the task every time after you login to your system. The task scheduler can be reached from the Cortana search bar near the start icon. Just type "task scheduler" and Windows would direct you to opening the scheduler. For some reason, my Windows did not launch this repeatably at startup.

Once, I published a script like this within my company and it caused a few interesting comments. Our HR asked me "Do you also have a script reminding people to go back to work? They are taking too long breaks!"

Exercise

For the mentioned scripts, it will be useful to be able to program the time of relaxation period. That is the program should also tell you when to go back to your seat. To do this, you need to have another variable for relaxation time and start counting down from that time once the pop up window opens.

Asking for help

I am putting help as the second topic in philosophy, so that, chippers practice asking for help. In my experience, most chippers keep to themselves and ask for help only under severe stress. The first think to do if you are unable to solve a design problem is to ask for help within your team and company. Some would prefer to try and solve problems all by themselves for the learning and fun but that wastes a lot of time. You are better off asking for help after some level of solo attempts. Some people never ask or that some who know the solution never tell you. It is somewhat ironic that the level of cooperation even within a small team is far from ideal. But there is a way to improve that. Before even you need help, you need to develop good relationships with your coworkers. That way when you ask for help, they will help you as a friend rather than look at you as a competitor.

EDA tools like any other software have bugs and you will encounter them at some point during your chipping career. Many chippers give up on asking help from the EDA tool vendor because the help does not arrive fast enough. Many are not even aware that help even arrives. In my career so far, I have encountered about 40 bugs in design, synthesis, place and route and schematic capture tools. If you think why only 40, it is because these are great tools. Even one bug is a big deal. I had asked for support most of the times and happened to get some useful solution or workaround for most of the cases. Some other times, I have not been so lucky and the support request got stuck with the EDA vendor.

Why ask for support?

An EDA tool bug fixed improves the productivity of chippers across the board! Not just yourself and your company, it improves productivity for all the users across the whole industry. Not asking for support and creating a workaround yourself is clumsy and not very reusable. I have found that my workarounds for bugs many times "stinks" though it works great for that specific issue. By contacting your EDA company, you may even get additional education about the tool from the EDA company application engineer (AE) that you never knew before.

How EDA company supports clients?

EDA companies have two groups each with specific function. The research and development (R&D) is the group that actually codes the software tools. The AEs do the job of answering questions from the customer. The AE is the first interface to the EDA company. When you have a question about the tool, you can either call the AE or submit a ticket on the EDA company website. For quick solution for your problem call the AE and for a more detailed problem that needs more study file a ticket. A ticket may also be referred to as a case. If the problem is not easy to understand for the AE, he or she may ask for a testcase that is nothing but a set of files to reproduce the problem exactly in the environment used by the AE or R&D. This way the AE has a concrete example to work on the problem. If the AE knows the solution to your problem he will give you the fix. If not, he will pass the problem to the R&D group that is the original maintainers of the software tool. I have seen AEs give one of these solutions – change to a different version of the tool, use a special tool command or option, use a workaround script

or rarely point out my stupidity! (Trying to debug a computer problem without turning on the power switch sort of!)

Packaging a Testcase

I found that rather than try to explain a tool problem in sentences, it is easier and more precise to create a testcase and send it to the EDA company. Many tools allow you to save a database of the files used for that particular iteration. There may also be commands to package a testcase. If you have questions about packaging a testcase, AE will be more than willing to help you. You do have a responsibility to protect the files of your company that are going outside. I prefer to reduce the size of the database to only the ones that are essential to reproducing the problem. Say, you have 50 SV source code files that when compiled crashes the tool. You can do a binary search over the compile list by trying one half of the list of 25 files for one compile and see if that crashes. If that does not crash then the issue may be in the other set of 25 files. By repeating this process, you may be able to reduce the size of the testcase from 50 files to say 1 file. This way you have made the problem simpler for the AE and also reduced the amount of information going out of your company. There are more ways to protect your IP that is explained in a subsequent section.

News and Gossip

Sam is a very good worker who is super loyal to his company and focused only on work. His company reports low profit for his department for 2 years in a row. He responds by working even harder. Sadly, his entire department is let go. Sam receives 8 months worth of severance pay. His company stocks are worthless because the company has filed for bankruptcy. He quickly gets another job for the same position in another company.

Tom is a good worker who works well but is not the best or most loyal worker. However he is very much aware of the business climate in the industry. After the first 6 months of poor performance by his company he searches and gets another job in a different company. He chose that company for its business strength and superior project execution. Tom's stocks are worth half a million dollars in a span of 3 years.

Who do you want to be? Sam or Tom?

Knowing deeply about your trade is important. Knowing the larger technology trends in the industry is important too. Who is who, which organizations control the standards behind the scenes, which company is fighting a court case against which company, which market segments are about to die and which path you can take to maximize your career returns. Once, there was a rumor of mass layoffs in a large chip company. Most employees were anxious. They did not know who was going to be axed. Apparently, there was a website that kind of leaked which location is going to have how many layoffs! Imagine knowing in advance if you are going to be laid off or not. These kind of "intel" can be crucial to overall career success.

I read these websites occasionally.

https://www.eetimes.com/

http://image-sensors-world.blogspot.com/

https://riscv.org/news/

https://Tomshardware.com

http://Anandtech.com

http://semiengineering.com

http://hpcwire.com

One way to create chip products

Typically, semiconductor companies start by gathering information from customers about their needs. Marketing team passes this information to a product definer. The product definer's job is to define which features this product should have for it to succeed in the market. Next, a specification is prepared that says in more detail which features are going to be implemented in the chip, how the chip will be used and many more details. This feature list passes to the chipper. He creates an architecture for the chip that accomplishes these features. There could be a separate chip architect doing this work. Then a team of chippers actually write HDL code for the chip. A synthesis group creates the gate netlist for the HDL code. A physical design group creates the layout of the chip from the netlist. The layout is manufactured in a fab, sent to assembly house for packaging, sent back to the company for validation. After the chip passes all initial validation of the design, mass production starts and the chips are sold to customers. The diagram shows the order of events in getting a chip to market. It is not a strict order, in the sense that the next activity in line does not fully wait for the previous one to complete. Some RTL design can start even with partial architecture. The activities are also constantly being repeated. The specification may change many times during development leading to a cascading change in all the downstream tasks. At the end, the information obtained from the customers on the chip's operations is feedback into the next product development cycle for a new chip.

```
Customer Needs

Product Definition

Specification

Architecture

RTL Design

Synthesis            Verification

Place and Route

Fabrication

Test                 Characterization

Packaging            Validation

Sales
```

Where may RTL design be applied?

Chipping is in a way is purely information processing even though it appears as if there is a great deal of electrical engineering needed. The use of electrical engineering is only as a tool to do the information processing. Suppose, in the future super high speed magnetism based logic shows up, designers will switch to that to do the information processing. Note that there is no such magnetic processing technology today, I cooked up that concept just to drive home the point. So, wherever information processing happens chipping is useful. It could be high speed stock market trading, genome sequencing, traffic light controller, rocket control and guidance, robotic actuation, machine vision, artificial intelligence, smart agriculture. The list is limited only by the availability of capital to fund new chip development and the availability of skilled chippers.

Hardware is also software

Once, I was watching a video about designing CHISEL and the presenter was comparing the compilation time and execution time of design representations from software execution, RTL simulation through FPGA emulation. He mentioned an interesting idea that hardware can be thought of as software that takes hours or years to compile but executes a task in microseconds. Contrast this with software that compiles in seconds to minutes but takes hours or days to execute a similar task. A more unifying idea I can infer from this concept is that a design is a model. The model is meant to be executed in some way and the choice of execution platform – humans or mechanical devices or CPU or FPGA or direct silicon realizes the actual function of the model. In so committing to a platform we are stuck with the limits of that specific choice.

How many aspects are there to chip design?

Lets start with the experience of buying a smartphone. What does the salesman say?

"This phone is from Samsung/Apple/LG/Mi/Xiomi...., it has a quadcore/dualcore/octacore processor, it has 1/2/4.. GB RAM, the battery lasts for 12/20/30 hours, the price is just $800/700/600...".

Lets deconstruct the sales pitch. Samsung/Apple/... stand for reputation of the manufacturer. Indirectly, it points to the quality of the product. Is it likely to fail? Related jargon are design quality or being bug free, failure rate, ease of use and service support.

Quadcore/dualcore stands for performance. How fast can this product accomplish its job? Other names for performance in the industry is speed.

Xyz GB RAM is also an indirect reference to performance.

Battery time is a reference to how efficient is the product at doing its job. The jargon for this is power efficiency or simply power.

$800/700 is an indicator of cost. Indirectly it refers to the size of the chip. Bigger the chip the costlier it becomes. Size goes by the technical term area.

A basic list of aspects to design are -

Cost, speed, latency, power, area, verification coverage, reusability, maintainability, readability, testability, time to market, productivity, programmability, ease of use, security and safety.

The art of chipping is to strike a balance between these aspects and create a product that succeeds in the market. Striking a balance is also referred to as making a trade or trading off. There is no given formula to optimize across all these aspects. I also don't see any formula like that coming anytime soon.

Speed vs area tradeoff

"There is no free lunch" is the first thing a chipper learns. Make the design faster but then live with a more power consuming design. Make the design simpler and lose the opportunity to tweak the design in the field. Speed versus area trade-off is the first one every chipper learns. The simplest example is pipelining. By adding more flip flops to split the work done per clock cycle, the design works faster but also consumes more area.

Speed vs latency trade-off

The same example of pipelining also adds latency. But adds speed.

Reusability vs readability

Making a design reusable can make it somewhat less readable. Suppose you parameterize the design and remove common code and use functions, packages and generate statements, now, which actual parameter and generate block is being used is difficult to understand from a quick reading.

More readable case

```
design1.sv: data_out <= registered_data;
design2.sv: data_out <= unregistered_data;
```

Less readable but reusable case

design.sv

```
generate
if ( REGISTER_DATA==1) begin : registered
  data_out <= registered_data;
end else begin :unregistered
  data_out <= unregistered_data;
end
endgenerate
```

Testability vs area/speed/power

Adding scan chain improves testability but also adds from about 2% to 5% extra area. The added logic for testing also comes in the way of regular logic and reduces speed. Added logic also consumes added power.

Time to market vs area

Consider a design with two modes called a low power mode and a high performance mode. Say the low power mode uses similar logic as high performance mode and both implement a form of image compression. A fast and easy design approach is to design one of the modes first and instantiate in the design. Then after completing the first mode, instantiate the second mode as a separate piece of logic in the same chip. This way, the designer can focus on getting the functionality of the two modes correctly. But for a more optimized chip, you would look for common blocks and try to reuse them across both modes. For example image processing algorithms use memories to store a few lines worth of pixel data. Resource sharing is possible because multi-mode chips usually operate in only one mode at a time and the other mode is not active. By avoiding the time consuming step of resource sharing you can reach the market with your chip faster.

RAM vs flops

When the storage needed crosses a few thousand bits and the storage is of a very structured nature memories are used in chips. For example, a buffer that just delays the incoming data by 1024 cycles is a good candidate for using a RAM cell. You could also use flip flops for the same logic. Flip flops occupy more area than a RAM and also add to the cell count of the design. Flip flops are natively handled by all synthesis and physical implementation tools whereas RAMs need extra work of compiling a cell of the particular size of RAM required, additional work of instantiating test logic and additional effort to place and route the signals. Overall, RAM adds more work to the project. If you are interested in getting your chip to market fast go for flops and pay the price for extra area.

Priority of trade-offs

Of all the aspects in chipping which aspects are more important than the others is subjective. My personal preference is this -

Time to market, designer time, verification coverage, power, speed, yield, area.

My justification is that the chip industry is fast. Any delay in getting the chips developed means some competitor will get a similar chip to market before you. Designer time is a scarce resource because it takes a long time for a person to become a good chipper. In that time, silicon process technology grows rapidly to provide abundant silicon area for the taking. Computer speed increases rapidly in that time to drastically reduce compile and simulation time. So, throw silicon area and simulation time to buy designer time. With more functionality in the chip, verification continues to be the bottleneck to chip production. After all, the chip has to work! Power is next in line. Today in latest process nodes, there is plenty of area to implement more logic but there is a power dissipation limit that forces chippers to keep only a part of the area used at anytime. Low power design is not an option now. Speed is very important too. In the gaming and server markets, speed is everything. Chip yield is related to chip cost. Poorly yielding design costs more and is unattractive in the market. Compared to other aspects before like power or verification, chip yield can be improved by minor changes. So it is ok to not worry about yield in the initial passes of the design. Area is so plentiful it is mute to even mention it. Yet, there are indirect effects to using excess area that warrants minimizing area usage. For example, a slightly bigger die could force you into using a bigger package that bumps up the chip cost.

How many types of implementations are possible?

You may wonder how many types of architectures there are to implement a given function. Suppose you are asked to architect a 65536 bit divider for some cryptography algorithm you could start with a combinational logic divider with multicycle path timing constraint. You could also add many pipelining stages and do a pipelined high performance implementation. You could do a serial implementation with smaller combinational logic units if the design affords a leisurely pace of computation. A parallel implementation is possible too if you need faster speed at the cost of higher area.
You could also do a streaming implementation. By streaming, I mean new inputs keep coming in as a stream of data every clock cycle, as against a burst based design that supplies the inputs and then waits for sometime before supplying the next set of inputs. In general, streaming implementations need high performance because there is no time gap to use a slower speed architecture.
You could also do a software implementation, that is, rather than use dedicated logic in silicon you are just going to use the time of a processor available in your system. You could also do a hybrid processor-hardware implementation. This hybrid is many times called a hardware accelerator. In a hardware accelerator implementation, say for example this 65536 divider, a software code would prepare the inputs for division and pass it to a dedicated hardware compute units like a 65536 subtractor and shifter. The software would do the sequencing work of using the hardware compute units and the compute units would "accelerate" the otherwise difficult work of doing 65536 bit shift and subtraction operations. You may wonder whats the difference between a hardware accelerated vs regular software. A regular software implementation uses processor instructions that are tailored for general purpose applications. These instructions include say upto 128 bit multiply, divide, add, subtract,

shift, logical operations, conditions and loops. If you have non standard instructions that are supported by hardware units in the processor silicon then it is an accelerated implementation. In the 65536 divider example, if you may have special instruction like CRYPTDIV which calls the 65536 specialized dividers the acceleration would be apparent if you see that an equivalent software implementation of the CRYPTDIV instruction may take 1000x longer processor clock cycles.

Digital versus analog implementation is another possible difference. Suppose you want a simple course logarithm function in your system, a simple diode could do the job at an order of magnitude smaller area and power. If you want the same logarithm function to be more accurate and free from process variation, you could use a digital implementation using a ROM.

Last but not the least, human vs machine implementation is a valid categorization. For example, many older cars needed the drivers to switch on the headlights at night. The trend is, however, more and more mechanization because machines are able to do a better job.

In my view of computation, the combinational architecture is the root implementation because it does not bother me with intermediate timings. Any other implementation is seen as derivatives of this master combinational architecture with modifications to timing and compute units. Software developers may notice that a pure combinational implementation is synonymous with a software implementation.

Synthesizable Vs Non Synthesizable

SV code is meant for logic synthesis or testbench/modeling. If only the modeling language was different we will not have this confusion. But, SV is like a super set of many loosely related subgroups. I have fair bit of confusion with synthesizable and non synthesizable constructs even today. Lucky, there is a way to solve the confusion. How? Just synthesize the design! The synthesis tool will tell you non synthesizable constructs.

Following is a list of non synthesizable constructs.
#delay, assertion, covergroup, coverpoint, operations in initial block, queue, associative array, class, system tasks like $abs/$sine, === and !== comparison, real datatype and more.

Why is something non synthesizable?

To synthesize a piece of code, the logic synthesis tool needs to be able to transform it into logic gates available to the tool. If a construct is overly complicated to translate to hardware then the tool does not support it. Lets see the #delay example here.

Suppose you have a delay in your design, #1ns as shown below. The intention of the code is to initialize the variable sum to zero and then wait for one nanosecond before adding up the contents of an array a.

```systemverilog
always @(*) begin
  sum = 0;
  #1ns;
  foreach (a[i])
    sum = sum + a[i]
end
```

The synthesis tool will be confused by the #1ns statement. This is because a synthesis tool does not have a good quality logic gate can provide a fixed time delay of 1ns. For that matter, almost all CMOS gates don't have a fixed delay behavior. Their delay varies up to 2 times from one production lot to another. For delays, synthesis tools just issue a warning message and ignore that statement. So, these kinds of non synthesizable code should be avoided in RTL meant to model synthesizable hardware. Otherwise, you will end up with chips that behaves one way in simulation and behaves differently in actual silicon. This is the classic simulation-synthesis mismatch.

Exercise

Study the SV LRM and note down 20 more non synthesizable constructs.
Write 10 synthesizable and 10 non synthesizable modules.

What is a natural candidate for a chip (or module, or system)?

A system or a chip or a module can be split into sub hierarchies any number of ways. How do we know which piece is a good candidate for a chip? The simple answer is that any chip that the customer is willing to buy is a valid chip. There is also a more technically correct answer based on performance or optimization. Imagine an imaging system that takes light incident upon the system and say it outputs a JPEG file. How could you architect this system? There are very few single chip solutions doing that. So, how do we split it. We have to choose a technology node for one or more chips needed for the system. Some functionality is worth splitting into a single chip. Image sensors need a high quality photodiode so they go into separate imaging process node. The image processing task takes a raw image from the sensor and outputs nice looking images. This task needs a lot of performance, so it goes into a chip in the latest high performance CMOS process. Whenever logic blocks need a high bandwidth for communication between them, they go into the same chip. There are operational boundaries as well. A single chip may be possible for a system but there may be no one company capable of designing a chip like that. A system is split into many chips for yield reason as well. The most common way to select a good chip is to screen for any defects in the chip with a tester. Even if a single defect is found the chip is trashed. Defects occur in a chip in proportion to the size and complexity of the chip. Suppose you have a small chip the size of a grain of sand that has 10 million transistors, it may have pretty good yield of say 97%. That means only 3 chips of out of a 100 chips are trashed during production. Say, you have another large chip the size of a small coin with over 10 billion transistors, it may have pretty low yield of say only 80%. It may be cost effective to split the big chip into two smaller chips that each may yield 90% at say half the price for each chip. Say the big chip cost $200 if the yield was 100%. Now with only 80% yield the effective cost is 200/0.80, $250. Say the cost of each half chip is $100 when the yield is 100%. For 90%, the cost is 100/0.90, $110. For the full system the cost works out to $220 which is cheaper than the $250 single chip solution. In general larger more integrated chips are preferable whenever technology, economics and logistics permit.

How do I know if my design is good?

In school or in a large work environment, it is often difficult to get a feedback about the quality of the design we make. Many designers assume that a fully verified status is the completion for a good

design. There are so many make or break steps after that too. A design that cannot be placed and routed well is no good even if it verified with full coverage. The same goes with yield. A design that yields only 10% of the produced dies means the cost of the product will be 10x compared to another design that yields 100% of the dies. A chipper should also remember that chipping is a career. So, we need the customer to come back to us for many more chips in the future and that happens when the customers' customers find their electronic systems working well for their full specified lifetimes. When every chip product or design IP returns more than the investment we a sustainable chipping venture!

Stage	Goodness %
Compiles	1
Simulates	5
Verified with good coverage	40
Synthesizes	50
Places and Routes	60
Timing closed	70
DFT coverage is high	80
Production yield is high	85
Customer is happy	90
Millions or billions working for long time in the field	98
Made a profit for your organization	100

Graceful Degradation

Suppose you are hiking all alone along the scenic Santa Monica mountains near Los Angles. The Pacific ocean is visible and you want to take a better look. You take a detour of a few hundred meters from the well made trail to a small hill in hope of a more beautiful view. Bad luck! Your right foot slips, you fall and twist your ankle! You are unable to rest on your right leg, but your left leg is fine. To make things interesting, imagine your phone battery died and the nearest human contact is 3 km away. How do you get out?

If your body happens to behave like how many chips behave, you would instantly die the moment your ankle twisted. Only, mission critical chips are designed to take serious damage and still perform some useful action. The idea of graceful degradation is that a loss in functionality of some parts should not compromise the entire system. The system should be able to perform something that is still of some value rather than completely shutdown. Graceful degradation may also go by fault tolerance. It is also related to functional safety.

As a chipper, you can include this philosophy in your design practice. The easiest way is to cover unused states in sequential logic. Any FSM that does not use the full 2 power state bits quota of the

number of states has unused states. The same is true for flop based logic that does not use all bit possibilities of the register in calculating the next register value. For example, say you have 5 FSM states defined which means you need 3 bits and there are 8 possible states that can occur. You have coded the logic that handles 5 of the 8 states. Graceful degradation demands that you have something coded to take care of the situation when your FSM accidentally enters any of the 3 bad states. You may wonder why would bad states happen? The answer is internal and external noise can change the value of sequential logic.

Another easy thing to do is to use periodic resetting of sequential logic that rely only on its own past states. Imagine, you have a counter that starts when reset is removed and that there is no other logic controlling the next state of the counter but for the count value itself. If the count value accidentally went from 1111 to 1000 instead of 0000, all the logic in fanout of the counter will get affected permanently. In contrast, suppose you have a signal called sync to periodically synchronously set the counter to 0 on some external event that the system depends on, then the fault on the counter will only momentarily corrupt the system. Upon getting the next sync signal, the system will most likely regain proper operation. The counter type of logic occurs in FIFO too.

Imagine the graceful degradation as some kind of exception handling in software, like the throw-catch construct in C++. In more general sense, keep an eye open for how parts of the chip may fail and how the failure can be contained.

Some Design Approaches For A Chipper

There are many ways to undertake the design process. I have experimented with a few and found them useful under specific cases. Like any other work, the design process can be optimized for some metrics. The first is off course time. Actually, you get paid by time so why not minimize the time spent designing. The second aspect is quality. It is preferable to reduce the chance of a bug happening in the future. You may also add minimization of keystrokes and mouse movements to reduce the wear on the body. You may also add maximization of learning.

Code As You Go

In this approach, you have a specifications or a problem to solve with a design and you just start coding something in HDL. As you code more and more, you figure out which logic blocks may be needed, how much memory may be needed, which places to pipeline. This approach is what I started out with as a beginner. It has the drawback of being very confusing and many times never converging to a working design. I learned from my professor that you need to first have an idea of the architecture of the design in mind and preferably drawn on a sheet of paper before starting to code. This approach is the poorest for minimizing key strokes.

Block Diagram First Code Next

Many specifications and hardware designs are described as block diagrams. This is because engineers intuitively understand diagrams better than text. Every block can be turned into a module in HDL. It also nicely breaks down the larger design into smaller blocks. Compared to code as you go approach

this method does indeed produce working designs relatively consistently. But there are a few drawbacks as well. For brand new designs or ill defined problems creating a block diagram may be harder than code as you go approach. The block diagram may look nice on paper but may not work when coded because the block diagram is only a mental estimate of the design. It is possible that when compiled and simulated it does not meet timing assumptions made when drawing. I prefer a hybrid block diagram and code as you go approach. I first start with a very high level block diagram with only a few blocks and then start coding that. After I hit a wall trying to tweak that block diagram to work, I learn its short comings. Then I go back and make a more detailed block diagram that has a better chance of working. In place of a block diagram, you could have a module hierarchy or pseudocode before starting. I have realized that a module hierarchy written on paper is just as good as a diagram.

One Change At A Time

Many times there is a reference design available to start your design. It may not be usable for your design as is, so it needs modifications. If you make all changes to the reference design to get it to work for the new design, it may fail verification and may not be possible to debug. So, by changing the reference design little by little we maximize ease of debug. Running counter to my intuition this approach has many times been faster to get a working design than coding everything in one go. Incremental approach is also preferable for reporting progress to management as they expect steady progress rather than unpredictable progress. For example take the case of designing a 24 bit high quality audio codec from a reference 16 bit version. First, you can change the traffic generator in the testbench to be able to produce 24 bit data and feed only the MSB 16 bits to the design and the checker. Next you can expand the datapath in the audio coder modules from 16 bits to 24 bits but only effectively use 16 bits. You can continue the process until the design is fully changed and passes 24 bits of data. I often split the change even further into compile only change and functional change. By compile only change, I mean change bus widths, add dummy modules, add new checker code or any modification that will not cause the test to fail. A functional change is bigger change that includes changing the logic operating on the checked data or anywhere that you are not sure will pass the test. Note that the idea is to make the modified design pass verification after every change. Also note that sometimes a partial step is impossible because it cannot be made to pass the tests easily. For example, if you change the encoder logic to work on 24 bits, it may be impossible to keep the decoder logic at 16 bits and pass the tests. So, both have to be changed. You may understand one change at a time as smallest change at a time. The drawback of this approach is that it involves a tremendous amount of typing and also sometimes takes a lot of time to complete. To reduce typing and time, I usually resort to a slightly modified approach - "biggest passing change at a time". I mean, make as many changes in one go as you can as long as you know that it will pass when simulated.

I have a real example for this topic. When designing a FIFO from scratch I started from the nearest working design and gradually made the FIFO.

Step 0: Copy files from ehgu_sr_mem (Shift register using memory) example directory to fifo example directory and check if testbench passes. Git commit with meaningful name like "checkpoint – changed

xxx". The purpose of committing working fileset is to be able to go back to that point when you are unable to debug the testbench failures after you have made too many changes.

Step 1: Copy the ehgu_sr_mem.sv file as ehgu_fifo.sv. Change sim.f compilation file to use ehgu_fifo.sv and check for pass. Git commit.

Step 3: Change testbench to check for pattern in data rather than fixed delay of input data. The pattern here is expected data is just the previous output data incremented by 3. Check for pass. Git commit. At every step you are going to check for pass and Git commit, so, that is assumed from here.

From
```
    expected_data = $past ( data_in , SHIFT , 1 , @ ( posedge clk ) ) ;
```
To
```
    expected_data = $past ( data_out , 1 , 1 , @ ( posedge clk ) ) + 3 ;
```

Step 4: Change module name inside ehgu_fifo.sv from ehgu_sr_mem to ehgu_fifo. Make corresponding change in testbench. Add features to be added as parameters to ehgu_fifo module, just as a to do list item. Those parameters don't do much now. Keep the existing params of ehgu_sr_mem as is to keep the testbench passing.

```
module ehgu_fifo
# (
parameter SYNC_TYPE = 0,
parameter SYNC_STG_W2R = 2,
parameter SYNC_STG_R2W = 2,
```

Step 5: Split the ehgu_fifo module into three modules ehgu_fifo (top), ehgu_fifo_mem (container for memory used in FIFO), ehgu_fifo_logic (logic for creating write addresses). Split the read and write clocks. Set write clock to write address and read clock to read address. Bring the two clocks to the testbench level and connect clk signal to both wclk and rclk.
This step proved to be too much for me as I made too many connectivity errors. Since it was only connections change it was not too bad to fix.

Step 6: Read write address compare
A shift register does not need to compare read and write addresses. The read address is a constant offset away from the write address and the write/read are continously happening to create the illusion of a shift register. But a FIFO reads out only when there is data left in the memory. If there is no data left the FIFO stops reading. This effect is created by comparing the read and write addresses and the difference is assumed to mean the number of data remaining.

From

```
always_ff @ ( posedge clk , negedge rstn ) begin
  if ( !rstn ) begin
    waddr <= 0 ;
    raddr <= MEM_DEPTH - SHIFT ;
  end else if ( en ) begin
    waddr <= ( waddr + 1 ) %MEM_DEPTH ;
    raddr <= ( raddr + 1 ) %MEM_DEPTH ;
```

```systemverilog
    end
end
```

To

```systemverilog
always_ff @(posedge rclk or negedge rrstn) begin
  if(~rrstn) begin
    renable <= 0;
  end else begin
    if (raddr != waddr)
    renable <= 1;
  end
end

always_comb begin
 if ( renable ) begin
  raddr_next = ( raddr + 1 ) % DEPTH ;
 end else begin
  raddr_next = raddr ;
 end
end

always_ff @ ( posedge rclk , negedge rrstn ) begin
  if ( !rrstn ) begin
   raddr <= 0;
  end else begin
   raddr <= raddr_next ;
  end
end
```

There are many more steps like this. But, I think you would have got the idea now.

Code Everything Deal With Bugs Later

Have you ever tried dumping all your travel items into a bag and then sorting out the stuff to a nicely packed state? Coding everything before fixing bugs is like that. It usually never converges. But for designs that you are very familiar with and that have a super high chance of working. I have found that coding everything works too.

Bottom Up

This is a standard approach to designing anything. In the example of the audio codec, you will design one piece of the encoder logic and test it, then move onto another piece of logic and test it, then test the full encoder and test it. You complete the logic at the bottom of the hierarchy first and then proceed to building the full design. I like the predictability and steadiness of this approach. It suffers from being too slow and needing too much typing. I use this approach for designs that I have no idea about.

Top Down

This is another standard approach. You start by designing the top most module with high level behavioral logic and then slowly work your way into making all the lower level blocks. For example, in

the audio codec example, you can start with a C model of the logic build your tests, then change the C model into SV behavioral logic then split the logic into modules and finally convert the behavioral modules into RTL code. Suppose you need a buffer that delays the data by 10 cycles, rather than use a specific buffer, you could just do $past(data, 10). The top down approach is useful to share your design with software and verification groups for them to start working on their tasks.

Architectural Exploration Before Coding

This is actually not a separate method but a rather complementary approach to the earlier approaches. You start out by creating one architecture that meets your design goals. You make a conscious effort to make other equally good architectures that have different power-performace-area trade-offs. You compare these architectures mentally, or in a review with other designers or go all the way to implementing each of these in RTL to get an accurate comparison data. I like this approach for it gives a very good understanding of the problem and many times outputs the best design anyone can make. It also provides the maximum amount of learning. It has the drawback of taking too much time.

Tweak A Reasonable Architecture Until It Works

Sometimes there is not enough time to research all the possible architectures for the problem in hand. In these cases, you can go with an architecture that has a good chance of working and then implement and tweak it until it works. This approach is very time efficient but has the drawback of making you stuck with a sub-optimal solution for a long time in the future. You can actually see this in many of the product designs all around you. You know that they are not good but exist because they were quick to reach the market.

Uses and cost of programmability

Most chips come with configuration registers that provide the users with greater control over the use of the chips. Suppose your chips is an audio codec that was designed for 16 bit audio and for only one application your client needs 15 bit audio. Also, suppose that you had designed a zero padded mode into the chip that has a register specify anywhere from 0-7 zero padding bits. For 15 bit audio, the user would just pad 1 zero and the rest of the user's system would just work. How happy your customer would be? Suppose you did not have the feature of zero-padding, your customer cannot support the 15 bit application. The case of zero padding is easy to justify and so should be included during design. Some other feature many not be that easy to justify. Programmability options greatly empower your customer to cover unforseen use cases.

Another great reason to have programmability is to provide options to survive bugs. For example, suppose your chip has a new auto-program feature for a PLL. Say, this feature would calculate the PLL divider register values needed for a target frequency with only the input reference clock frequency. Assume that this behavior is achieved with a small logic that calculates the divider values. This design example needs only one register to program the PLL, the REFCLK_FREQ register and it relieves the customer from thinking about the PLL dividers. If you additionally provide control registers for say MAN_PROG, PLL_M and PLL_N with the meaning of MAN_PROG=1 will force the PLL_M and PLL_N values to be used to set the PLL dividers and MAN_PROG=0 would enable the automatic program mode. In case of a bug in your auto program logic, you could fall back on the manual program option. Sometimes this option is informally referred to as saving your a** :). An extension of these

manual options to take over the operation of entire FSMs. When there is a bug in the FSM you could switch to the manual mode and carry out the operation of the FSM in software.

Note that programmability comes with a great cost. It increases verification complexity enormously. It also increases the load on documentation by increasing the number of registers that need explaining to the customer. Somewhat, ironically a more flexible chip may be disliked by the customer because it makes his job harder. This is somewhat like the paralysis induced by too many options. Programmability also adds to chip size and slows down timing paths because programmable logic in a simple sense translates to mux and demuxes. In the extreme, a highly programmable chip is no different from an FPGA and the very programmability is the reason FPGA is slower than an ASIC implemented in the same process technology.

Range of programmability

Control signals

It is interesting to think about chipping from the point of programmability, just as an exercise. Lets start from an input signal. You could say it programs the signal carried on the wires to be 0 or 1. At the next level, you could think of a signal that is not driven as an input but driven from a software bus to program a register. If you look at options from the point of view of compilation you can see localparam, param, define statements of SV. The localparam allows only programmability inside the module. The param statement allows overriding while instantiation of a module. The `define statement allows setting something throughout the compilation scope. You can view a param imported from a package as a scope limited define statement.

You can also see non-volatile memories as additional levels of programmability tools. Especially, a ROM based programming of a signal is more flexible than a signal hard coded in RTL but not as flexible as an externally programmed register bit. An even finer type of non volatile storage is a metal programmable cell. Think of a metal programmable cell as a tiny ROM. Most often it has only one word that there is no need for address bus. All it does is provide a constant output after power up. Chip design iteration number may be programmed using a metal programmable cell. For the next iteration, say from 1 to 2 of the chip, just 1 or 2 metal layers of the layout of the metal programmable cell is modified.

FSM

FSMs form critical parts of a chip. They are also a hotbed of bugs. It is a good idea to have some flexibility in its implementation. Lets list the options we have in making FSMs programmable. At the rigid end we have the FSM coded in RTL that after synthesis becomes set in silicon stone. Nothing can be done to fix bugs in FSM other than change the chip layout. Next is an FSM that has some control signals coming from registers. For example, consider an FSM that is supposed to produce timing signals to start an analog block. Say, the timing of the analog block is –

1. Hold power down and reset asserted for 1us,

2. Release power down and hold this state for 2us,

3. Release reset and hold for 100ns,

4. Assert enable signal and hold for 20ns

5. Wait for up to 10ns for ready signal from the analog block.

Assume that the waiting times are implemented with a counter that counts at the rate of 1ns per count. So, in the RTL of a hard coded FSM there would be parameters for the first time 1us = 1000 counts, 2us = 2000 counts, etc., like

```
parameter DURATION1 = 1000;
parameter DURATION2 = 2000;
parameter DURATION3 = 100;
parameter DURATION4 = 20;
parameter DURATION5 = 10;
```

Suppose the analog block needs 1ns more than the 10ns for getting ready in up to 5% of the PVT corners, without flexibility in the waiting time in the FSM you would have to throw away the slower chips. Instead, say you implemented the parameters not as compile time values but as field programmable values from control registers and the FSM takes the registers as inputs like -

```
Addr   Register
0x100  DUR1
0x101  DUR2
0x102  DUR3
0x103  DUR4
0x104  DUR4
```

```
input duration1,
input duration2,
input duration3,
input duration4,
input duration5
```

You could program the DUR5 register to say 11 and get the FSM to work in all PVT corners. This off course assumes that the additional 1ns is harmless.

Note that these duration constants could come from an on chip ROM as well. Or even a RAM that is first loaded from a ROM but later the RAM could be open to writing from external software. In fact, every computer system in some ways behaves like this. The first program that executes is from a non volatile memory. This program subsequently loads the bigger user programs from a hard disk or USB drive or from network into the RAM.

To change or not to change

To make a change to the design or not is a vexing question with no specific answer. If you are a conservative person you may like minimal changes. If are an innovative person you may like to change the design even if it means to risk introducing bugs. Most designers are in between these two extremes. Although everyone preach the importance of change, I think nobody feels change as painless. Your

customers rather than appreciate you for introducing a great feature may criticize you for creating additional work for them because of the change. Such is the love-hate relationship that everyone has with change. So, how to handle change? When I was a new college graduate that recently joined the industry, I did not know how much to change. My boss criticized me for making too much change. One thumb rule I learned the hard way is to make progressively smaller changes. In the initial stages of the project, change as much as needed to make the best design. As the project matures, reduce your changes to only the most important ones and for god's sake stop introducing features that no one asked for. At the final stages of the project do only bug fixes. Sometime after that even bugs will not be fixed. Only workarounds will be entertained and the bugs will be documented in an errata.

Another point to consider when making a change is to count how many people is this going to affect. If it is going to affect many stay away from that change. You may also think about how likely is this change going to be harmless. If you are sure that it will not introduce bugs go ahead and make the change. Beware! Seemingly innocuous changes can land you in big bugs!

Conventional Algorithms vs ML

Specifications and RTL, verification are all mostly precise. Every bit is important. Conventional algorithms like heap sort or dynamic programming work on well defined data. On the other hand, ML can tolerate and even thrive on vague data. ML has already proven worthy in image recognition, autonomous vehicles and many more fields.

Conventional algorithms are simple to implement in hardware compared to ML. There are methodologies to translate algorithms to hardware. Algorithms are easier to analyze for correctness. ML is opaque and does not easily lend itself to inspection for correctness. ML needs lots of data for training. In some ways, ML is another form of algorithm in itself.

How can ML be used for specification, design and implementation of chips? This is an open question for which I don't have an answer. I wonder if regular algorithms have come out of an ML like human processing. I mean, for a give problem human experts have studied many pairs of inputs and their corresponding solutions. After looking at many such examples, their brain has probably got trained to identify a pattern. If the pattern is simple, they spell it out in mathematical language and then it becomes a conventional algorithm. I think ML was always there and ML resulted in conventional algorithms. Watch out for ML in your design process, it could radically change the chipping landscape.

Digital Logic Abstraction Is Invalid

Digital logic based design and verification makes some assumptions that are invisible to the chipper, unless when you start hearing about chip failures in the field!

The first assumption is logic 1 and logic 0 are neat and clean. In real circuits logic 1 and 0 are nothing but clean. What is expected to be a full 1V often tends to be 0.94V and occasionally even 0.55V. These effects can cause a corrupted logic state to enter the system.

The second assumption is that the rise and fall times of 0 to 1 and 1 to 0 transitions are not important. Again, real chips fail often when some signals are not routed properly with good fast rise-fall times.

The third and most insidious assumption is that a logic circuit is affected only through input ports. Modern physics says otherwise! Everything in the universe is affected by every other thing in the universe, just that the effect is not noticeable if the objects are farther apart in space or time. You can see the IO-ports-only assumption break when a working memory suddenly shows a corrupted bit. This is called a single event upset (SEU). A cosmic ray particle or a local high energy particle generated from radioactive decay of trace radioactive materials in the chip or the board imparts enough energy to toggle the state of a memory or a flop. A lesser form of interference is the external radio wave band noise from cellphone, radio transmitters and even other electronics. Wires nearby couple noise via capacitance to the logic under use and affect its state or the speed of a transition. Other logic nearby could affect the power supply voltage for the logic in consideration and cause it to malfunction. Suppose a large combinational logic like a 64b multiplier is turned on and operating at 1GHz near your logic then the power supply dip could cause the timing paths in your design to work slower and miss setup time. This power supply drop due to large current is called IR drop. The same goes with temperature. Suppose another chip in the same board overheats and increases the temperature of your chip then your logic could just stop working altogether if the temperature is too high. Or, it could miss setup time. The safe temperature for a chip is technology dependent. For silicon, the range is limited to about 150-200 degree Celsius. Above this the silicon stops being a semiconductor and transistors don't behave like transistors anymore. The temperature limit is higher if the substrate is made of Silicon Carbide or Diamond. Another assumption is that the logic block or memory is time proof and there is no aging. But the fact is transistors become poorer with use over years. The content of a non volatile flash memory could auto erase over a long time.

Just to see how an abstraction that includes some of these will look like consider the input ports for an inverter given below. Look at how unwieldy even an inverter model becomes! So, we live with a useful simple digital abstraction and sort out the unmodelled rest in the lab :)

```verilog
module inverter_of_perfectionist (
input real noise_from_distant_supernova,
input real noise_from_local_radioisotopes,
input real noise_from_neighboring_wires,
input real noise_from_radiowaves,
input real temperature_of_chip,
input real power_supply_noise,
input a,
output zn
);
real my_age;
always_comb
   if ( (temperature<temp_limit) || (my_age<age_limit)) begin
     if ( sum_of_noise_sources() > threshold ) begin
       zn = a;
     else
       zn = ~a;
   else
     zn = 1'bx;
endmodule
```

The model we use -

```verilog
module inverter_good_enough (
input a,
output zn
);
  assign zn = ~a;
endmodule
```

Chips Are Mortal Designs are Immortal

Physical chips die soon in a few years to a few decades. Designs, however, survive for a long time. Take the case of the lowly I2C protocol. It was invented many decades ago. It is still in use today and will most likely by used many decades into the future. Even if it is replaced, parts of the protocol will still survive in newer protocols like I3C. The brand new looking Iphone indeed contains decades old design, with many designs surviving past the life of the first author.
You may wonder, so what if designs don't die? Why do we care?
When designing for a specification under the gun to complete before a deadline, the last thing in a chipper's mind is future projects. The chipper makes choices that help meet the immediate deadline but can make the design less readable in the future. When the time for the second version of the design comes, you may be cursing yourself for not taking the time to make your design a bit future proof.

Machine Mimetic Design

When designers get inspired by nature its called bio-mimetic design. The opposite is now making more sense. I am calling it machine mimetic design. Machine algorithms have improved so much that human performance is starting to look poor. Once, I was reading through the optimization section of a synthesis tool. There were many tricks that the tool would use to reduce area and increase performance. At a later point in time, I happened to look at the schematic of a hand designed digital logic. I was surprised that shift register optimization for a scan chain was not used by the experienced circuit designer! In the man vs machine contest the score was now man 0, machine 1. This sort of performance gap is repeating more and more. In the arena of RTL based designs, chippers could soon be losing out to C++ based designs compiled into RTL by HLS. To keep one step ahead in the game, you are advised to study the latest EDA tools for ideas that improve your work. Learn from the machine and apply to manual design task. Note that this should not be mistaken to mean sticking to design processes that have been made obsolete by newer EDA tools. No matter how much a chipper copies EDA algorithms, no chipper can match the speed and scale of machines. Machine mimetic is to be understood as the idea that when your are forced to do a design task manually consult an equivalent EDA algorithm that does the same job.

Actually, the term machine mimetic makes for a good sound bite but is a misnomer. It may actually be called learning from condensed intelligence of many humans because the machine algorithms were researched and coded by a large number of people.

Average Is Codable Whereas Extremes Are PhD Projects

A simple circuit like an adder that takes only the "+" operator to code in SV. The same adder, however, needs PhD level circuit design knowledge if it needs to work at the extremes of frequency afforded by a process. For 28nm process, a 10GHz, 16bit adder will be a topic for PhD. The point to note is chipping based on SV code is mostly for the average in many aspects of design like speed, area and power. But what is not average is off course scale! Nothing matches the design size of a SV RTL based chip. No hand made design even comes close to the size of modern chips leveraging the power of RTL coding.

Power is data dependent

When you see in a datasheet that a chip consumes 5mW you would think that is the maximum it will consume. It is not very clear that power number is not perfect. Suppose your chip is a video decoder that decodes video streamed over Internet into a format useful for the LCD TV screen. The power used by the chip may vary if you watch a UHD movie versus if the screen is only showing a screensaver picture. This point becomes important when you need to estimate the power consumed in your block level logic of the chip. Should you use the worst data traffic pattern? Should you use the average traffic? How do you even know which is the worst traffic pattern? It is a hard problem. This is one other place where silicon helps. A chip can run a large number of data traffic patterns within minutes and provide a great estimate for power. Simulation simply cannot hit that many patterns. There is a shortcut to this problem. It is called toggle rate or switching activity or activity factor. You could assume that all nets in the design toggle at 30%, meaning all nets on average switch 30 times every 100 cycles of its corresponding clock domain. This gives a very good headstart.

Pipelining vs Parallel Processing

Parallel processing, in contrast to pipelining, does not breakdown logic into smaller pieces. You may wonder why parallel process when you can pipeline? Because, pipelining is not always possible. Breaking a logic and inserting flops drastically changes the logical behavior of a design. It is very likely that bugs get introduced during the process. Also, a design may not present easy locations to insert flops. Sometimes, even modifying the design may not even be an option. This happens to encrypted design. Say, you purchased a stereo camera 3D depth calculation engine in encrypted RTL format. Say, it works at 200MHz max in 28nm. Suppose you want to make it work at 400MHz, how can you pipeline it? It is impossible, unless you are willing to pay the core vendor more to make you a 400MHz core. To avoid paying double the price, you use the parallel processing approach. Two 200MHz core run in parallel with the output of the two cores multiplexed to one stream results in an effective processing speed of 400MHz. Note that in pipelining, splitting a logic in half results in less than twice the max frequency. You never get the full 2x improvement. In parallel processing, you get a neat and full 2x improvement if you use 2 cores. The overhead in parallel processing is the additional control logic that distributes input data to the instances and collects the output data from the instances into one stream. You could say the area used by a 2x parallel implementation is always more than 2x area of the unit design. Pipelining has one interesting advantage over parallel processing. Suppose, your design has 100k timing paths and that 99,990 paths are capable of 400MHz operation and 10 paths are capable of only 200MHz operation, splitting only the slow 10 paths may make the entire design operate up to 400MHz. You are saved the area of one extra instance of the parallel instances. The idea

is to pipeline if your design has small number of critical paths and the logic is simple enough to present locations that may be pipelined easily.

RTL is process dependent

I felt very proud of my work for one of my past designs. Wow! I had just completed a complex design and made it highly reusable and readable. Too bad, my pride did not last long. When I tried to close timing for the design, it would show many timing violations. I had to break the uniformity in some modules and sacrifice readability by pipelining strategically. I could not find a reusable way to pipeline the design.

Herein lies the problem of RTL. It is too timing sensitive. If you want to get decent performance and use least area, you will have to tweak the RTL for your target process. Did I tell you RTL is process independent? Sorry, I was wrong :) The necessity to insert pipeline flip flops or move the already existing flops around is what breaks the process independence feature. To give a more concrete example, consider a 2 stage pipelined 128b adder targeted for a 40nm process working at 1GHz. Suppose you now reuse the RTL in a 7nm process for same 1GHz, the two pipeline stages may not be needed because the 7nm technology may be able to do the full 128b addition in just one 1GHz cycle. If you still used the two pipeline stages, it may waste area. The opposite case of porting a 7nm design to 40nm is also equally problematic, you now have to insert pipeline flops to meet the slower 40nm performance. Either way, RTL is not as portable as it may seem.

How I wish there is a way to specify just the timing-less functionality in RTL and the tools take care of meeting timing in the process technology. May be this is what is HLS. For most practical purposes, it may be possible to implement various speeds of pipelined design using cleverly parameterized code.

Debugging

Print

The deceptively simple print statement has been helping me debug right from my first C program to my latest and greatest SV design. The value of $display is its simplicity. But it takes time to add print statements to a large design, so, this is not scalable.

Single Stepping and Breakpoints

I have not used much of single stepping to debug. But it may be useful in cases of software like serially executing sections of code. Your simulator may also allow you to insert breakpoints in the code to see where it hangs or misses.

Waveforms

For one particular design, I was frustrated by the pace of the simulation when waveform logging was turned on. I thought why not go back to debugging with just print statements. After trying to add a print statement here and a print statement there, I realized that there is a reason to saving a waveform. I can say a waveform file is a million print statements added to your design! Waveforms produce a lot of data for large designs. You could hit many 100s of GB per microsecond for large SoCs. Just opening and

closing such large waveforms files may take many minutes and thus blunting the value of the waveforms. Chippers typically create special waveform probe files that specify which sections of the design you want to export into the waveform file during a simulation.

Waveform analysis tools

Signal Compare

You can do a difference of one signal against another and display the resulting signal as a new waveform in the waveform window. The difference is actually just the XOR operation.

Waveform Database Compare

You may have waveforms from a passing test and a failing test. By comparing the differing signals, you may be able to narrow down the location of bugs. Waveform database compare differs from signal compare in that the waveform viewer tool automatically finds the differences in *all* the signals of both the waveform databases.

Expressions

You could create additional derived signals out of the probed signals using expressions. Suppose you are interested to know when a special symbol appears on the data bus, say, 'b10101010, you can create an expression like data == 'hAA and give it a name, like, data_pre. Expressions also let you visualize data in a different way than how it is declared in the code. For example, you may have a 64 bit type in code that actually represents the velocity of a baseball. Rather than see the signal as 64 bit hexadecimal numbers like 0xABCDEF01, 0xDEADBA11, etc., it would be easier to set it to a real number format and plot it as an analog signal. The magnitude of an analog signal is readily understood by the height of the value in the waveform whereas a hexadecimal number does not readily convey the magnitude.

Time shift

Data passes through one clock domain with fixed latency as measured in clock cycles. Imagine a data processing pipeline with blocks named A,B,C with A taking 3 cycles to complete, B taking 5 cycles to complete and C taking 2 cycles to complete. Suppose that a clock cycle is 1ns. By delaying the data input of A by 3+5+2=10ns you can create a signal in the waveform window that matches the data output of C. Comparing the delayed data input at A versus the output of C can show you any mismatches.

Cross probing

Most waveform viewers allow you go back and forth between source code and the signal in the waveform viewer. The way I have used the cross probing feature is that I look for suspicious behavior in the waveform and then pick one signal which may be the root of the problem. I then cross probe the source code by right clicking on the signal in wave window and selecting the source code option. The waveform viewer opens the file containing the code for that signal.

Driver trace back

You may be able to see an unwanted change in a signal in the waveform viewer, say a 0 to 1 change when you are expecting a steady 0. To debug the cause of a transition, you can use the driver trace back feature that shows the piece of code or expression that caused this transition. For example, consider the following piece of code. Suppose, you expect memnext[0] to hold a steady 0 but it changed to a 1. Upon cross probing the transition you could see the statement causing the change highlighted in a separate window.

```systemverilog
always_comb begin
  memnext = mem ;
  ready = 0 ;
  xo = 0 ;
  case ( state )
  C0 : memnext [ 0 ] = ~b ;
  C1 : memnext [ 0 ] = mem [ 0 ] & a ;
  C2 : memnext [ 1 ] = ~a ;
  C3 : memnext [ 1 ] = mem [ 1 ] & b ;
  C4 : begin
   xo = mem [ 0 ] | mem [ 1 ] ;
   ready = 1 ;
  end
  default : begin
   memnext = mem ;
   ready = 0 ;
   xo = 0 ;
  end
 endcase
end
```

Version Control Based Debug

It so often happens that a working design suddenly starts failing many tests. The usual culprit is some recent change in the design. You do a log of all the commit messages till the date when the design was ok and see if any message can provide some clues to the test failures. For example, the following command shows only the messages from the date of August 12, 2020.

git log --since "Aug 12 2020"

In one one of my previous projects, my senior coworker debugged a hard problem just by looking at the commit messages! Wow, that's experience!

You could also do a diff between the working and failing versions of the code. For example you could create another copy of the design database from the working date and then do a file by file diff with the latest failing database. The Git command to use is -

git diff

Reduced Design Tests

Typical verification tests do so many things and include many parts of the design that it makes it difficult to focus on the failure. I regularly try to reduce the scope of the problem by creating another

test environment that is much smaller than the full chip environment so that I can simulate the test faster and try many experiments. For example, suppose you are in charge of the I3C block in a latest multi-billion transistor smartphone apps SoC and that a full chip I3C test fails after a 10 hour runtime. It is difficult to run many experiments on a test that gives a result only after 10 hours. You could recreate the same failure in an I3C block level environment that runs the same test in only 10 minutes. Note that sometimes a block level test may be impossible to create.

Force Statements

For the designs that you are not much familiar with, any test failure is hard to just understand, let alone debug. You can focus on the important parts of the design by forcing off unwanted parts of the test/design. You could also speed up the initialization of the design by just forcing an initialized state of the design during your simulation debugging. Forcing also may be useful in creating the reduced design test. Make sure that you do not resort to force statements for regular use in your verification environment.

Point of First Deviation

Chips mostly perform some kind of serial operation. There is a beginning, there are many intermediate points and there is an end. The idea of locating the point of first deviation is to breakdown the chip operation into smaller pieces and find out where the chip deviated from the expected behavior. Note that the point of first deviation may not be the point of the test failing. Point of first deviation is very likely to be the closest you can get to the root cause of the test failure. Running behavioral models of the sub blocks of the design in parallel with the RTL models can provide an easy way to locate deviations. Assertions can also help in pin pointing the locations of deviations. In the absence of models or assertions to show expected behavior side by side, I resort to a crude method of tracing back the 'x' seen in the waveform window. The problem with the point of first deviation or the tracing x is that cause of the failure may diverge into many potential directions for investigation or go round and round in circles without hitting the root.

Assertions

Entire books have been written on the value of assertions for debugging. I can only reiterate the value of assertions. The skill in creating assertions is to make sure you are specifying useful properties of the design rather than repeating the logic already in the RTL. For example, if an FSM has 10 states and you repeat the next state logic as assertions, it is of no use. However, if you make an assertion that the sequence of states s3,s5,s6 do not occur which is not readily apparent in RTL but is in fact a property of the logic you want to realize then it makes for a good assertion.

Lint

Under pressure from project deadlines, I have at times committed sub-par quality RTL code that would pass initial verification. Later, when full verification is run there would be test failures and magically, there would also be Lint warnings discovered in the same modules that caused the test failures. What was happening was that Lint was providing useful early warnings on bad design which when ignored

results in verification failures. You could turn around this phenomenon and run Lint on your design when verification fails. After all, Lint takes only a few minutes to run compared to the many hours to run verification tests.

CDC

CDC is a debug tool just like lint. CDC errors or warnings hide important design bugs. So, run CDC if you have verification failures.

Talk to Chippers

Your coworkers can be a great source of ideas. If you talk about your failure to a coworker he or she may be able to immediately tell you the cause of the bug or give you useful hints. The number of designs one chipper can do in one lifetime is small compared to the number of designs done by many chippers. So, leverage the experience of others. For some people, the mere action of describing the test failure can trigger new ideas even without getting any feedback.

Small Perturbation

If you read any scientific research paper you can notice that there are two are more experiments that are very similar except for one are two variables that are different. The idea is to make small perturbations to your test and see what happens to the test result. It would be tempting to make wholesale changes to debug faster, but, that would most often result in an unfathomable mess. You can take this matra for debugging – one variable at a time.

What is the story of the word bug?

Apparently, the word bug got the meaning of a mistake in the design. The story of how this came to be will excite you to wake up from being half asleep :)

https://en.wikipedia.org/wiki/Software_bug#History

Design Knowledge

What is design knowledge doing in a section about debugging simulation issues? Well, I would say design knowledge is the single biggest tool in fixing a simulation failure. Knowing the design can guide you quickly to the important signals of interest and also to the point of first deviation. To gain knowledge about an existing third party design, you can compile the design and study the hierarchy, familiarize with the naming convention of the code, study the specifications of the design and the C/C++ or functional model of the design, run the block level tests of the design and study any other documents from the design authors. If you do not invest in knowing the design you may take a long time to debug the failing testcases and end up learning the design by the hard and time consuming way! So, why not learn it once before hand. I try to learn about the design if I know that I am going to be spending a lot of time maintaining the design in the future.

Multiple Monitors

Multiple monitors? Are you serious? Actually, using a wide monitor or using many monitors drastically improves your debugging speed by letting you see more. Don't discount this easy debug aid! Note that unlike other debug aids that take a lot of education and skill, monitors are a one time investment open to everyone.

Design for Lab Debug

Debugging a chip in the lab is extra hard compared to debugging in a simulation environment. In the simulation environment you have access to every signal in the design. You can even stop the chip simulation right after a malfunction happens and examine the signals leading up to the bug. In the lab, however, you have only a limited means to control or observe the internal signals. To ease this problem, designs often include logic that takes control of an internal logic and pumps known test patterns. Correspondingly, observing logic are inserted in strategic places in the chip to bring out the data traffic passing through those points. This idea is especially useful for less mature designs. Take for example, your new chip is a 2-channel 16 bit to 1 bit serializer working at 200Gbps. Assume that the input data comes in via two 16 bit ports operating at 12.5 GHz, there is a mux to chose which port is coupled to the serializer, a 16-to-2 parallel in serial out shift register and finally a 2:1 double data rate output circuit. In the whole chain of circuits, anything can go wrong. In the lab you may be hard pressed to locate where the problem is.

Controllability can be increased by adding logic that generates data at every stage – after the mux and after the serialization logic. The traffic generator or test pattern generator creates a known pattern of bits. It could be a 1010101 pattern or a pseudo random pattern using a linear feedback shift register.

Observablity can be enhanced by snooping on some of the internal signals and making them available at primary outputs. For example by adding a mux that selects between the serializer inputs and DDR IO inputs as signal output available at primary outputs, it becomes easy to study the waveforms on an oscilloscope. This process is also called probing. The logic that probes internal signals and makes them available at outputs is called in-circuit logic analyzer or ILA.

https://zipcpu.com/blog/2017/06/08/simple-scope.html

There are also less complicated ways than the ILA. You could instantiate logic that collects statistics of the system, such as, how many good data passed through and how many bad data passed through, count of data packets sorted into a range of sizes and status bits showing the health of the circuits, such as, a working clock signal. Loopback is a powerful concept that increases both observability and controllability.

Documenting your design

Pretty much everyone hates to document their design. That includes me too. The reasons are typically, not enough time to actually do the design, documentation is boring, documentation takes away the author's control over the design by letting out the secrets and in a perverse way if your designs are well

documented your salary and job stability are less secure than if the design is poorly documented. But then there are plenty of very good reasons to document your design well.

Why document your design?

1. Ice breaker

   Even within the same company there is competition among employees for salary, promotion and just avoiding the next layoff. The general policy of most employees is to give if they get something. So by giving away the ideas in your design you can encourage your coworkers to share their ideas. This would make your team strong and enable your company to win the competition in the market place. The next layoff may never happen!

2. Help for verification

   No matter how well you have verified your own design, verification by a specialist is important to prevent serious bugs. Often verification specialists do not understand RTL code or do not wish to look into RTL code to create their verification test suite. Many of the specialists start work only if your design is clearly specified in a document.

3. Go on a vacation peacefully

   I like this point a lot! I have sometimes had the unfortunate experience of being called or having to call others during a vacation because the problem with the design needs urgently talking to the author of the design. Your spouse or family may get annoyed if you carry your work with you even during vacations. By leaving enough documentation for users to read, you insure yourself from being disturbed when you are out of office.

4. Poor memory

   Who was that person who wrote the code? That is the feeling I get when I look at my own code after a few years. It becomes unreadable pretty quickly because of my short memory. I think most of the chippers suffer from the same problem. So, better leave as much documentation when you remember your design.

5. Datasheet preparation

   When your product is sold to customers a datasheet or user guide is provided as a collateral. You need to explain your design at an operational level for the user. You do not have to include implementation details. Some documentation is mandatory. So why not make it before hand.

6. Handoff

   At some point you will have to leave your company. Among my peers, the range of time in the same company stands from about 1 year to 12 years. The average is about 4 years. When you do leave your company, leaving a decent amount of documentation is a nice professional gesture. Your boss and coworkers would remember you positively. Years later, you may be

rewarded for your generosity by a call from your coworker who may now be a CEO in an amazing startup.

7. Not everyone understands HDL

There are more than one group of people who need to understand the basics of your design. I know of at least these groups of people – lab characterization engineers, silicon test engineers, product application engineers, product designers. So, leaving plenty of documentation in plain English is immensely useful.

How can design task end when verification can't?

Verification seems to tire me much more than design because there this sense of incompleteness. No matter how much I have verified, I can never be sure that there won't be any bug that will show up in the hands of the customer. But design always ends when you meet the specifications within the area and timing budget. Or does it?

On deeper inspection, one could say design does not end either. There is always a smaller, faster and more power efficient implementation than your present RTL code!

Unit 8 – Philosophy of Verification

Verification, like design has many principles hidden under the innumerable buzzwords. An experienced chipper may have a feel for these. I offer to fast track your gut feeling by shining a spotlight on selected topics.

Model based verification

The general problem in verification is how do you know if your design works as expected? The simplest approach is to compare the data coming out with expected data. But how do you create the expected data?

Many times, the expected data turns out to be a simple mathematical formula, so, it is expected to be perfect. For example, say, you are creating a high speed 64 bit adder design targeted for Radar application. The "+" operator in SV will create the expected data. In some other cases, the expected data is the input data itself. This is typically the case in communication systems. In cases where the expected data is simple you do not even realize that there is a model.

In cases where the expected data is more complicated than a simple formula, a reference model is used to create the expected data. Suppose you are designing a chip as an accelerator for climate prediction then the reference model would be a software program that is known to work correctly. The expected data can be used in two ways. Export the data to a file and use it in RTL simulation. I am calling this offline way. Or, run RTL and software at the same time and compare in the simulator. I am calling this online way. SV Direct Programming Interface (DPI) makes it possible to use the online way with C/C++ software code.

Which one is better, online or offline?

SV simulators are generally much slower and costlier than C/C++ compilers. Forcing the software model to be run by the simulator may take higher run time than just running RTL only and then reading the expected data from a file.

Running the C model using DPI is very convenient and makes the verification much more readable. The online approach is also highly scalable. One test vector or one billion test vectors, the verification setup effort is the same. But the offline file based way hits a limit if the test vector file size hits the GB range.

True complexity of verification

The number of states in a digital system is a power of 2 of the number of inputs and flip flops. Suppose your design is just a 64 bit counter with inputs of 64 count bits, 1 bit parallel load enable and 1 bit enable, then the total number of states it can be is $2^{((64 + 1+1) \text{ inputs}+ 64 \text{ FF})}$. The 2^{130} states is so high that no verification methodology ever invented can test all states. The practically reached states may be of the order of 2^{30} leaving about 2^{100}-2^{30} untested states. The ratio of tested states to untested states is practically zero.

How do chips even work with zero verification coverage?

If the verification coverage for even a simple counter is practically zero, how do extremely complex chips like GPUs with many thousands of cores that each contain many thousands of state bits manage to work? This question has troubled me for sometime. I don't have a good answer to this. But my guess is that of the huge number of possible states of the design, only some are really unique. By testing all the special ones, the untested ones are effectively checked because they are correlated with the special ones. In other words, the other untested states cannot have a bug while the special states have no bug. The other form of correlation I think that is helping is block to top correlation. Suppose your design works at block level then it will also work at the top level if only a few extra special states unique to block-top integration is checked.

Documentation is verification

During the process of documentation you are exercising the design mentally and very slowly. This process intuitively brings out any inconsistency in the design. A piece of code that can't be described can't work write. Mantra – "Not documented is not verified". When documenting the FIFO for this book, I noticed that the register stages for Gray code output are missing. Ironically, I did not notice that during coding of the FIFO!

During the process of presenting my design or creating the speaker notes for the presentation, I have noticed vagueness in some aspects of the design. Those aspects most often turn out to be bad designs or buggy designs.

Who owns a bug?

By adding this topic under verification, I seem to say all bugs are owned by the verification engineer :) However, bug ownership is a complicated and emotionally charged topic. Bugs are nasty, bugs entail monetary loss, trust loss, possibly loss in salary raise and promotion. So, nobody wants to own it. But someone's got to own it! In my experience, I have seen ownership being handled in an incredibly mature manner. No one points their finger in public. May be finger pointing goes on in private. The ownership is not even talked about. The boss keeps track of the bugs and the plan to fix or patch it. Only for the purpose of recording and bug tracking software tool, a bug is set to be owned by the designer who happened to code the piece of logic that has the bug. I thought deeply about who really owns a bug and got these questions.

Did the bug happen because the designer carelessly made it?

Did the bug happen because the verifier did not work hard to get sufficient coverage?

Did the bug happen because the boss insisted on meeting unreasonable deadlines?

Did the bug happen because the customer insisted on useless features that took the time away from checking useful features?

There are more questions like these. But the question of who owns a bug is a dangerous one. Rather in healthy companies, the question will be, "how can we prevent this kind of bug from happening again?".

Dynamic vs Static Objects

If you had started chipping from the electrical engineering side, you will have developed a hardware centric mental model. Everything in the SV code is expected to have a matching physical thing. But too bad, verification code is very different! I found debugging verification code much harder with the static mental model. The solution is to switch to the dynamic mental model. That is every piece of code comes to life in a special time point and then magically disappears. With this model, you can be more aware of pieces of code that are dynamic and pieces that are static. There are ways to debug the dynamic objects by making the simulator show the zero-delay changes that happen in them. For example take this function. The bits of "a" are added in zero time. But with appropriate settings of simulation and waveform capture, you can let the changes be visible in the waveform viewer. Many times, I also revert to the primitive print statement to debug dynamic objects.

```
function automatic int add_fn ( input logic [3:0] a )
add_fn = 0 ;
 foreach (a[i])
   add_fn += a[i];
endfunction
```

Debugging objects of classes is even harder than functions. But luckily people have found methods to deal with this difficulty. Please refer to this DVCON paper here.

http://events.dvcon.org/2012/proceedings/papers/12_3.pdf

Verifying a chipper

Lets push the analogy of a brain being a computer to the extreme, to the disdain of humanists :) Lets say a chipper is only a sophisticated chip that converts specs to RTL. For now lets ignore the chippers humanity. How would you verify if the chipper chip is good to release (or hire)? I mean, how would you know if the chipper possesses all the skills necessary to convert a complex specification or a poorly defined vague verbal product requirements document into a fully functional market beating chip? This is the process of certification and recruitment. There may be some concepts applicable from mainstream verification.

States of the Chip

Like how a chip can have many logical states, a chipper can have many emotional modes and non-emotional thought states. A verification test tries to put a chip in all the important modes. In the case of a chipper, how can you put him/her in all emotional/thought states and check the response. For example, if faced with a layoff what will a chipper do? Work harder or look for other jobs or bring a gun to work :)! In the same way, if the chipper is faced with a tough design task but not necessarily a

job threatening one, how will the chipper respond? Employers and coworkers will be very interested to know such responses.

Knowledge Graph

Imagine a graph, that is, a network of nodes and links. Each node is a subject, say, electrical engineering or digital logic or SV. One could say a good chipper would have heavy nodes with a large number of links between nodes. I am using the notion of node weight to mean the depth of knowledge in the subject and the number of links to mean the knowledge of the relationships between the subjects. It is not enough to posses a large amount of static knowledge, you would also want to verify if the chipper can skillfully traverse this graph. I mean, can the chipper easily move from one subject to another to think about a problem and arrive at great solutions? For example, can he think about a problem from the perspectives of human resources, SV coding, materials science and corporate finance?

Hierarchical Verification

Just as chips are verified hierarchically, a chipper can be verified hierarchically too. For example, testing one node in the knowledge graph and then moving on to a group of nodes and links.

Constrained Random

Some chippers can trick the chipper verification test suite by practicing the answers for only the frequently asked questions. To beat such cheating, we need constrained random testing. In other words, the interview questions should be randomized to check for true understanding of the subjects.

Coverage

After introducing the concept of states and randomization the logical last step is to introduce the concept of coverage. Were all the nodes and links in the chipper knowledge graph checked? Were all the interesting paths in the knowledge graph checked? The chipper test suite needs to have single discipline or single subject questions to check nodes and then multi-discipline questions to check links.
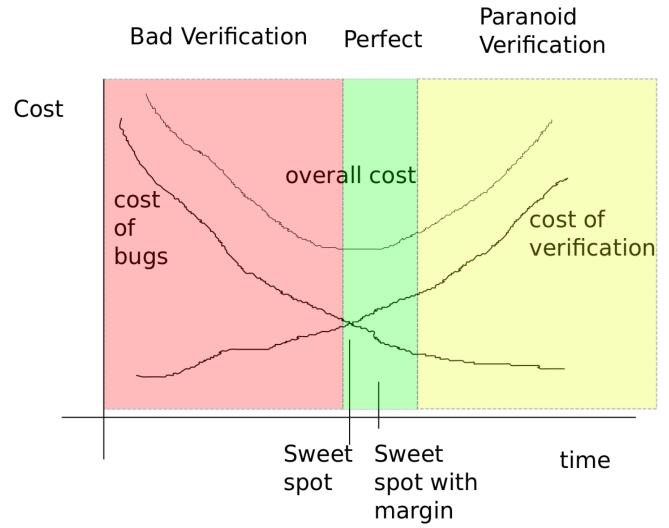
When to stop verification?

Apart from the logistics of verification, the topic of when to stop is the ultimate philosophical question in verification. This question often gets buried in the thousands of verification jargon.

Verification cost vs Bug cost

Take the case when the design is poorly verified and the product creates thousands of deaths during the course of the system operation. This case would be unacceptable. Now, take the case when the design was verified thoroughly such that it works flawlessly in the hands of customers. Which case do you prefer? Of course the thorough verification case!

Lets add some more information. Say, full verification added two years to the product launch and a competing buggy but mostly working product was launched one year before. Which one would succeed in the market?

The choice of how much to verify depends on what happens if the product is buggy and what is the consequence of delaying product release. In many cases, a mostly working buggy product is sufficient and may even be preferable. The case of self driving cars is a nice case in point. Suppose you are working on a robocar machine learning processor, you have a choice of how much to verify. If you take too long to verify fully then it means robocars availability is delayed and there could be many roadside fatalities attributable to human error. Availability of robocars would have reduced human driving and human casualties. At the other extreme had you verified poorly, faulty robocars could cause more accidents than humans. So, I would say stop verification when the cost of delaying is higher than the cost of releasing. Note that from the point of best cost, you may do a bit more verification because the human brain weighs loss as more painful than an equivalent gain. But is there a formula to calculate cost of delay or cost of bug? No!

Unit 9 – Chipping on Amazon Web Services
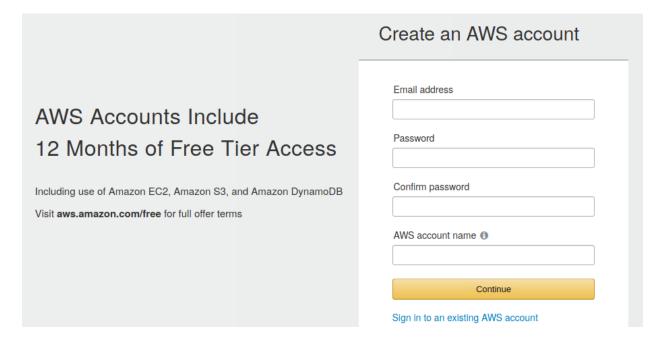
Getting Started

In the past, companies used to have all of their servers on premises. I guess, in the future all EDA tools will be available in AWS on pay per hourly use basis. You will be able to choose the server processor core count, RAM and disk storage capacity per your project needs. This section is getting to use AWS for chipping. For now, I know how to login into AWS. If you are interested in more advanced options to chip on AWS, you may be able to use the Xilinx Amazon Machine Image (AMI) that has all the Xilinx tools installed.

Creating AWS Account

Have a credit card on hand and follow these steps -
https://portal.aws.amazon.com/billing/signup#/start

AWS has a free tier option for 1 year. I have used this option so far and been able to do basic operations. In the free tier, you have access to computers with modest processing power and modest RAM and disk storage. I recommend you get a feel for AWS on the free tier before using the paid features.

Signing into AWS
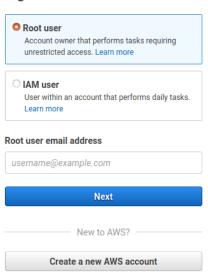
Open https://aws.amazon.com/ and go to My Account\AWS Management Console
Or
https://console.aws.amazon.com/?nc2=h_m_mc
Login with the username and password you created before.
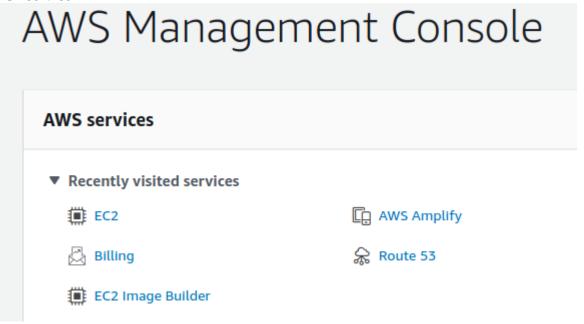
Starting A Machine In AWS
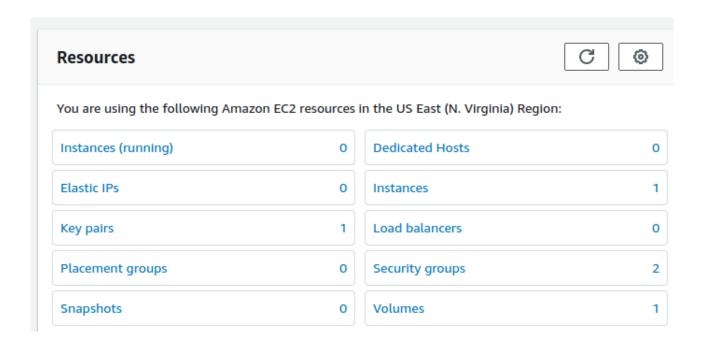
Go to EC2 service

After opening the EC2 service, you will see this.
I already have an instance, key pair and security group. For a new user these numbers will be zero.
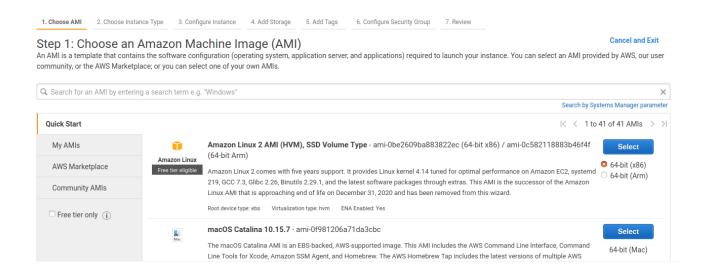
Resources

You are using the following Amazon EC2 resources in the US East (N. Virginia) Region:

Instances (running)	0	Dedicated Hosts	0
Elastic IPs	0	Instances	1
Key pairs	1	Load balancers	0
Placement groups	0	Security groups	2
Snapshots	0	Volumes	1

Click on Instances

Instances (1) Info   Connect   Instance state ▼   Actions ▼   **Launch instances** ▼

Click on Launch instances

1. Choose AMI   2. Choose Instance Type   3. Configure Instance   4. Add Storage   5. Add Tags   6. Configure Security Group   7. Review

Step 1: Choose an Amazon Machine Image (AMI)

Cancel and Exit

An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. You can select an AMI provided by AWS, our user community, or the AWS Marketplace; or you can select one of your own AMIs.

Search for an AMI by entering a search term e.g. "Windows"

Search by Systems Manager parameter

Quick Start   1 to 41 of 41 AMIs

My AMIs

AWS Marketplace

Community AMIs

☐ Free tier only ⓘ

Amazon Linux 2 AMI (HVM), SSD Volume Type - ami-0be2609ba883822ec (64-bit x86) / ami-0c582118883b46f4f (64-bit Arm)

Select

◉ 64-bit (x86)
○ 64-bit (Arm)

Amazon Linux 2 comes with five years support. It provides Linux kernel 4.14 tuned for optimal performance on Amazon EC2, systemd 219, GCC 7.3, Glibc 2.26, Binutils 2.29.1, and the latest software packages through extras. This AMI is the successor of the Amazon Linux AMI that is approaching end of life on December 31, 2020 and has been removed from this wizard.

Root device type: ebs   Virtualization type: hvm   ENA Enabled: Yes

macOS Catalina 10.15.7 - ami-0f981206a71da3cbc

Select

64-bit (Mac)

The macOS Catalina AMI is an EBS-backed, AWS-supported image. This AMI includes the AWS Command Line Interface, Command Line Tools for Xcode, Amazon SSM Agent, and Homebrew. The AWS Homebrew Tap includes the latest versions of multiple AWS

You can see many AMIs here. I chose the Ubuntu free tier AMI. Click on select button.

Choose the t2.micro free tier eligible image. You can see that it has 1 CPU and 1 GiB of RAM.

Click on review and launch button at the bottom.

Click on the launch button

1. Choose AMI   2. Choose Instance Type   3. Configure Instance   4. Add Storage   5. Add Tags   6. Configure Security Group   **7. Review**

Step 7: Review Instance Launch

▼ AMI Details   Edit AMI

Ubuntu Server 20.04 LTS (HVM), SSD Volume Type - ami-0885b1f6bd170450c

Free tier eligible   Ubuntu Server 20.04 LTS (HVM),EBS General Purpose (SSD) Volume Type. Support available from Canonical (http://www.ubuntu.com/cloud/services).

Root Device Type: ebs   Virtualization type: hvm

▼ Instance Type   Edit instance type

Instance Type	ECUs	vCPUs	Memory (GiB)	Instance Storage (GB)	EBS-Optimized Available	Network Performance
t2.micro	-	1	1	EBS only	-	Low to Moderate

▼ Security Groups   Edit security groups

Security group name	launch-wizard-2
Description	launch-wizard-2 created 2021-01-22T07:16:47.199+05:30

Cancel   Previous   **Launch**

The launching step will start the machine in the Amazon datacenter somewhere in the world. You also need to connect to the launched server from your place of work. You need a key pair to securely connect to your server. For the first launch, you need to create a new key pair. Give it a name and download the file into a safe location into you local machine, laptop or desktop or maybe a tablet or phone.

After downloading the file and clicking on launch instances button, you now have your own server in AWS! Congratulations :)

There are many ways to connect to the server you launched. AWS lists the following:

Connection options
The operating system of your local computer determines the options that you have to connect from your local computer to your Linux instance.

If your local computer operating system is Linux or macOS X
- [SSH client](#)
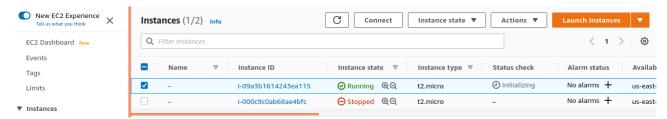
- [EC2 Instance Connect](#)

- [AWS Systems Manager Session Manager](#)

If your local computer operating system is Windows
- [PuTTY](#)

- [SSH client](#)

- [AWS Systems Manager Session Manager](#)

- [Windows Subsystem for Linux](#)

I am using the Linux-SSH method. More details of this are here - https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AccessingInstancesLinux.html

You can connect to your AWS server from your local machine by selecting you instance in the AWS EC2 console/instances and then click on connect.

It will show the following instructions

Connect to instance   Info

Connect to your instance i-09a3b1614243ea115 using any of these options

| EC2 Instance Connect | Session Manager | **SSH client** |

Instance ID

⧉ i-09a3b1614243ea115

1. Open an SSH client.

2. Locate your private key file. The key used to launch this instance is ForBook.pem

3. Run this command, if necessary, to ensure your key is not publicly viewable.

⧉ chmod 400 ForBook.pem

4. Connect to your instance using its Public DNS:

⧉ ec2-18-233-66-236.compute-1.amazonaws.com

Example:

⧉ ssh -i "ForBook.pem" ubuntu@ec2-18-233-66-236.compute-1.amazonaws.com

```
local-shell> ssh -i <location-of-key>/ForBook.pem ubuntu@ec2-18-233-66-236.compute-
1.amazonaws.com
The authenticity of host 'ec2-18-233-66-236.compute-1.amazonaws.com
(18.233.66.236)' can't be established.
ECDSA key fingerprint is SHA256:Biqnfr8eaxfLDzvyRy4TJmGiW4Az4zSKyXLp/n2+KzM.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'ec2-18-233-66-236.compute-
1.amazonaws.com,18.233.66.236' (ECDSA) to the list of known hosts.
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@        WARNING: UNPROTECTED PRIVATE KEY FILE!         @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
Permissions 0644 for '<location-of-key>/ForBook.pem' are too open.
It is required that your private key files are NOT accessible by others.
This private key will be ignored.
Load key "<location-of-key>/ForBook.pem": bad permissions
ubuntu@ec2-18-233-66-236.compute-1.amazonaws.com: Permission denied (publickey).
local-shell> chmod 700 <location-of-key>/ForBook.pem
local-shell> ssh -i <location-of-key>/ForBook.pem ubuntu@ec2-18-233-66-236.compute-
1.amazonaws.com
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-1029-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Sun Jan 24 02:22:09 UTC 2021
```

```
   System load:  0.0                 Processes:                104
   Usage of /:   22.3% of 7.69GB     Users logged in:          0
   Memory usage: 22%                 IPv4 address for eth0: 172.31.54.156
   Swap usage:   0%

 * Introducing self-healing high availability clusters in MicroK8s.
   Simple, hardened, Kubernetes for production, from RaspberryPi to DC.

      https://microk8s.io/high-availability

89 updates can be installed immediately.
40 of these updates are security updates.
To see these additional updates run: apt list --upgradable



The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.
```

<mark>ubuntu@ip-172-31-54-156:~$</mark>

Hurray! You logged into your server in AWS. The whole sky (cloud) is open to you now :)

Lets see where in the server filesystem we are and what it contains.

```
ubuntu@ip-172-31-54-156:~$ ll
total 28
drwxr-xr-x 4 ubuntu ubuntu 4096 Jan 24 02:22 ./
drwxr-xr-x 3 root   root   4096 Jan 22 01:57 ../
-rw-r--r-- 1 ubuntu ubuntu  220 Feb 25  2020 .bash_logout
-rw-r--r-- 1 ubuntu ubuntu 3771 Feb 25  2020 .bashrc
drwx------ 2 ubuntu ubuntu 4096 Jan 24 02:22 .cache/
-rw-r--r-- 1 ubuntu ubuntu  807 Feb 25  2020 .profile
drwx------ 2 ubuntu ubuntu 4096 Jan 22 01:57 .ssh/

ubuntu@ip-172-31-54-156:~$ pwd
/home/ubuntu
```

You can see that we are in the home directory of the server under the user ubuntu. There is not much user directories in the home as expected. Before proceeding further, lets check the specifications of the instance using cpuinfo and meminfo in Linux systems. You can see that the instance is Intel 2.4GHz CPU with 1GB of RAM.

```
ip-172-31-54-156:~/dsn_verif/useful_scripts> cat /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family  : 6
model    : 63
model name  : Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz
stepping  : 2
microcode : 0x43
cpu MHz   : 2399.907
cpu cores : 1
...

ip-172-31-54-156:~/dsn_verif/useful_scripts> cat /proc/meminfo
MemTotal:        1002104 kB
MemFree:          376008 kB
MemAvailable:     702672 kB
...
DirectMap4k:       67584 kB
DirectMap2M:      980992 kB
```

Lets populate the server with the ehgu library from Github. Downloading the Git repo also confirms Internet connection from the remote server to general websites.

```
ubuntu@ip-172-31-54-156:~$ git clone https://github.com/3vm/dsn_verif.git
Cloning into 'dsn_verif'...
remote: Enumerating objects: 2220, done.
remote: Counting objects: 100% (2220/2220), done.
remote: Compressing objects: 100% (1032/1032), done.
remote: Total 3617 (delta 1430), reused 1925 (delta 1167), pack-reused 1397
Receiving objects: 100% (3617/3617), 688.39 KiB | 18.12 MiB/s, done.
Resolving deltas: 100% (2274/2274), done.
```

Lets see if we can run any script from the Git repository.

```
ubuntu@ip-172-31-54-156:~$ cd dsn_verif/
ubuntu@ip-172-31-54-156:~/dsn_verif$ ./useful_scripts/lines-of-code-created.tcsh
-bash: ./useful_scripts/lines-of-code-created.tcsh: /usr/bin/tcsh: bad interpreter:
No such file or directory
```

The server gives an error because the tcsh (turbo C shell) is not installed in the remote machine. So, lets install tcsh. Note that bash Shell is installed already and is also the default.

```
ubuntu@ip-172-31-54-156:~/dsn_verif$ sudo apt install tcsh
Reading package lists... Done
...
Processing triggers for man-db (2.9.1-1) …

ubuntu@ip-172-31-54-156:~/dsn_verif$ echo $SHELL
/bin/bash
ubuntu@ip-172-31-54-156:~/dsn_verif$ which tcsh
/usr/bin/tcsh

ubuntu@ip-172-31-54-156:~$ tcsh
ip-172-31-54-156:~> cd
ip-172-31-54-156:~/dsn_verif/useful_scripts> cd ~/dsn_verif/useful_script/
ip-172-31-54-156:~/dsn_verif/useful_scripts> python3 get_pll_settings.py
PLL N=1 M=1 gives 1.0000e+08 Hz with deviation 2.0000e+07 Hz
PLL N=3 M=4 gives 1.3333e+08 Hz with deviation 1.3333e+07 Hz
PLL N=4 M=5 gives 1.2500e+08 Hz with deviation 5.0000e+06 Hz
PLL N=5 M=6 gives 1.2000e+08 Hz with deviation 0.0000e+00 Hz
```

Looks like the Python script to get PLL M/N settings for a given frequency is working. Python looks preinstalled.

Lets try some C program.

```
ip-172-31-54-156:~/dsn_verif/useful_scripts> cd parallel_scripting/

ip-172-31-54-156:~/dsn_verif/useful_scripts/parallel_scripting> ./run-benchmark-
serial-c.sh
./run-benchmark-serial-c.sh: 1: gcc: not found
```

C program cannot be run because the GCC compiler is not installed. So, install GCC.
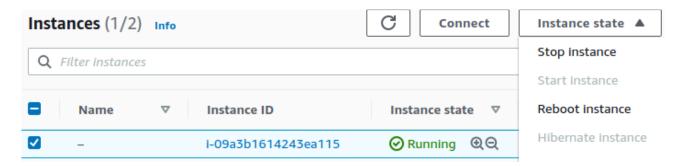
```
ip-172-31-54-156:~/dsn_verif/useful_scripts/parallel_scripting> sudo apt-get update
Hit:1 http://us-east-1.ec2.archive.ubuntu.com/ubuntu focal InRelease
...
Reading package lists... Done
ip-172-31-54-156:~/dsn_verif/useful_scripts/parallel_scripting> gcc
gcc: Command not found.
ip-172-31-54-156:~/dsn_verif/useful_scripts/parallel_scripting> sudo apt install
gcc
...
Reading package lists... Done
Building dependency tree
```

Lets try the same script again.

```
ubuntu@ip-172-31-54-156:~/dsn_verif/useful_scripts/parallel_scripting$ ./run-
benchmark-serial-c.sh
Sun Jan 24 06:39:05 UTC 2021
Core  0: PLL N=   1 M=    1 gives 1.0000e+08 Hz with deviation 2.0000e+07 Hz
…

ubuntu@ip-172-31-54-156:~/dsn_verif/useful_scripts/parallel_scripting$ ./run-
benchmark-parallel-c.sh
Sun Jan 24 06:40:00 UTC 2021
Core  0: PLL N=   1 M=    1 gives 1.0000e+08 Hz with deviation 2.0000e+07 Hz
```
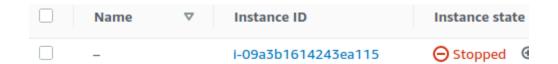
When the remote machine is running, it takes up real hardware and power. So, when you are done with your work stop the machine before logging off. Go to AWS console EC2 instances and choose the instance state "stop instance".
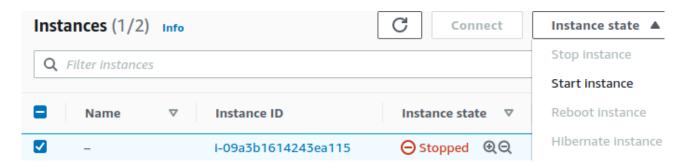
SSH client closes automatically

```
ubuntu@ip-172-31-54-156:~/dsn_verif/useful_scripts/parallel_scripting$ Connection
to ec2-100-26-136-111.compute-1.amazonaws.com closed by remote host.
Connection to ec2-100-26-136-111.compute-1.amazonaws.com closed.
```

Check again if the instance state shows stopped. If you forgot to stop your remote machine Amazon will charge you by the hour of usage of the CPU and Disks.

Restart your remote machine and check if you can connect again by SSH

SSH connect

Try connecting with the same ssh command you used before.

```
po:~/RTL_design/Ehgu_proposal> ssh -i ForBook.pem ubuntu@ec2-18-233-66-236.compute-
1.amazonaws.com
^C
```

Too bad ssh hangs because the remote machine address changes for every machine start. So back to the instance and click connect to get the current address for SSH. Note that the connect buttom may take a while to activate after starting the machine. The start operation on the instance caused the address to change from `ec2-18-233-66-236.compute-1.amazonaws.com` to `ec2-100-26-253-34.compute-1.amazonaws.com`

For some reason the ssh connect was refused the first time but connect on second SSH attempt.

```
po:~/RTL_design/Ehgu_proposal> ssh -i "ForBook.pem" ubuntu@ec2-100-26-253-
34.compute-1.amazonaws.com
ssh: connect to host ec2-100-26-253-34.compute-1.amazonaws.com port 22: Connection
refused
po:~/RTL_design/Ehgu_proposal> ssh -i "ForBook.pem" ubuntu@ec2-100-26-253-
34.compute-1.amazonaws.com
...
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-1029-aws x86_64)
```

Stop the machine and log off from AWS.

The recommended way to configure your server and easily deploy it is by creating an Amazon Machine Image, AMI. You can use this sequence. You may also want to learn about AWS Cloud Formation concepts to setup your instance rather than use one command at a time to install the applications you need.
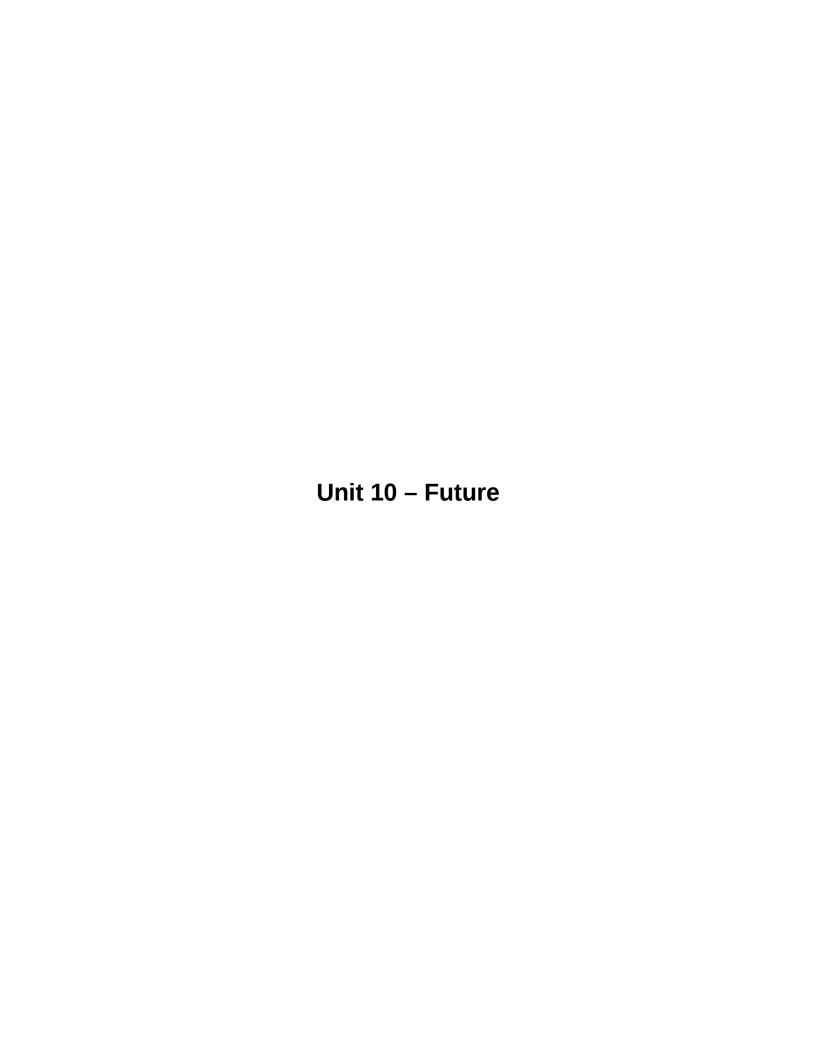
```
AWS console -> instances -> actions -> images and templates ->  Create image
id - (already named by AWS)
name : t2u_ubuntu_gcc
Description : Installed GCC, checked parallel and serial execution of C code
```

For some reason, I am not able to recreate a new server from the saved AMI.

Exercise
Launch an instance from a saved AMI and setup GUI usage in the remote machine using x-windows (ssh -X) or VNC or equivalent.
Install Xilinx Vivado in the remote machine and run one SV example from the ehgu library.

Unit 10 – Future

SV improvements

Ideas from Chisel about hardware construction may be added to SV. Ideas from modern programming languages may be added. The Verilog-A constructs for modeling continuous analog signals may be incorporated into SV. New constructs may be added for expressing matrix algebra operations like dot product, cross product, transpose because many ML/AI/DSP algorithms are expressed using these operators. More data types may be added, like, reduced precision formats for ML applications – 8 bit or lower floating points, decimal floating point, complex numbers and other formats useful to express modern designs. One last thing, the reg keyword is overdue for removal!

More constructs become synthesizable

As of this writing, real number arithmetic is not synthesizable, classes are not synthesizable. If EDA tools develop to support synthesizing these constructs the expressive power of RTL and the reusability will improve drastically.

Chisel

The relatively new hardware description language, Chisel, may get wider adoption. It is particularly well suited to abstract algorithms. I am not convinced that Chisel can replace all of the high resolution hardware modeling possible with SV or that Chisel can replace SV's assertion and coverage subset. It may be possible that simulators enable co-simulation of Chisel with SV. Chisel may be supported by synthesis tools too. Today Chisel is converted to Verilog for use in synthesis tools.

Portable Stimulus Standard

PSS aims to unify verification and testing at block and top level by creating a standard way of expressing the test intent. So far, it has not gained a lot of usage but it may change in the future and almost every chipper may be required to know this.

High Level Synthesis

HLS may become mainstream for implementation of complex algorithms. But, regular data transformation, control logic, glue logic, serialization and deserialization will still be designed in SV.

RISC-V

The RISC-V ecosystem will gain massive user base and many chipper jobs will ask for RISC-V skills.

Public Standards

Many advanced technologies start out as proprietary IP. As the technology matures, it inevitably becomes a public standard. This will happen to machine learning and artificial intelligence processors, automotive chips and Internet of Things chips.

Slower porting

The amount of time a design stays in the same process node will increase. A decade ago, mere porting the design to a newer process node would give significant performance improvement. But, in the future changing process node may be costly, so, there will be more insistence on making a more optimized design which makes chipping more interesting.

Human Machine Interface

Improvements to interfacing human brain to computers will help increase chipper productivity. Brain computer interface may be introduced for coding.

EDA tools

Compared to tools for general use like Google Assistant or Amazon Echo, the semiconductor EDA tools are rather primitive. New and useful ML tools may arrive to help design, say voice input coding, auto complete entire blocks of RTL, signal auto definitions, sophisticated debug tools. How useful it would be if an AI assistant helps chippers with debug work? Example - "When signal1 rises what is the state of some other signal2, what are possible causes of x in signal3?" and then the AI shows you useful reports! How about you say, "Alexa, pull data_valid from instance of module pci_buffer and bring it to module gpu_alu" and the EDA design editor automatically changes all the related RTL code to do this and also shows you what changes have been made?

Chip Design Is Eaten by Software

Chipping will become more and more software like. Already, modern verification code looks more like a C++ software application rather than something meant for hardware. Class based hardware description in SV, like Chisel, could become supported for synthesis. Software engineering ideas could be included in chipping. With C/C++ for design and the need to code complex algorithms in hardware, HDL will look more like software. Developers may find it easier to do chipping than pure old school electrical engineers. Software may be directly targeted to processors or directly to hardware without the aid of old school chip design engineers. There are already some developments along these lines by companies like CacheQ.

https://cacheq.com/

Lower cost design ecosystem

More and more EDA tools and library files may move to the cloud. There is significant amount of cost to keeping server infrastructure in the same company. The cost of licenses are also very high. Software as service model may pick up steam for chipping.

Subscription based service for EDA tools may become mainstream. Today EDA tools are super costly. Suppose EDA tools are available with reference flow for 20 designs in 2 processes, it can be a great place to learn synthesis, lint, LEC, STA tools. Subscription cost may be set to as low as $5 per month. It can be restricted to educational purposes by setting limits on the design size to say 100k gates. Exporting of design files may be restricted to avoid using the educational license for commercial use. Some data like reports may be taken out for grading. The full collection of learning EDA tool set can be hosted on cloud services from Google or Microsoft or Amazon for broader reach. The learners who show good promise can be directly hired for jobs by sharing their learning grades.

Exercise

Can you request the big 3 EDA companies – Synopsys, Cadence and Mentor Graphics to issue learning licenses for their chip design tools? It can be charged on per hour usage basis. The tool versions can be 10 years old so that the companies do not have to worry about loss of latest trade secrets. The tools need access to process technology files as well. Again, the process can be an old one like 65nm.

Chipping and Manufacturing as a service

At some point in the past every chip company had its own fab. As cost of the fab increased and chip manufacturing volume increased, it became more efficient to outsource all manufacturing work to only a few large fabs. Now with an explosion of chip products, the entire process of design to manufacturing seems to become a new service in itself. SiFive is an interesting company that lets chips be designed and tested at low cost. I believe more such companies will come into existence in the future.

https://www.sifive.com/about

Next Generation Chip Language

The idea of HDL be it VHDL, Verilog 2001, SV, CHISEL or C++ or any of today's design, modeling and verification languages seem to be made for an era that had keyboard as the only reliable interface connecting a human to a computer. The HDLs also indirectly assume a serial coding style that is representative of a time when CPU was the most widely used compute platform. Now the keyboard is giving way to touch sensor, audio input and in the far future to BCI. The CPU is giving way to CPU-GPU-FPGA-DSP combo chips implementing ML operations. How should the future chip language be?

PDF to RTL

This mostly will not happen. In general, a tool to create RTL from specification will seem magical.

Quantum Logic

Quantum computing promises to be a great platform for processing. I found this course on quantum computing very interesting.

https://www.edx.org/professional-certificate/delftx-quantum-computing-and-quantum-internet

Chipping Enters High Schools
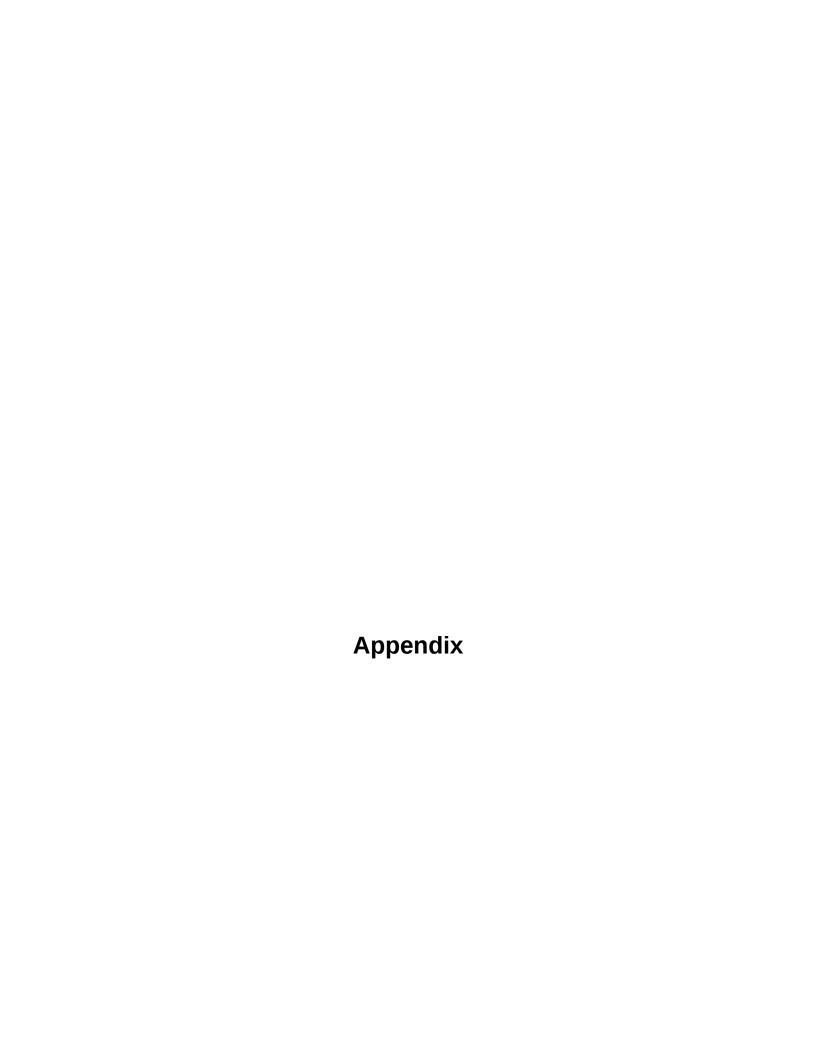
There is no reason chipping cannot be taught to school kids. Digital logic is far simpler than calculus and would be immediately understood by most school kids. It would also open up a path for passionate kids to start contributing to the chip industry very soon without having to spend hours studying other subjects that may not be very useful. Adding chipping to Advanced Placement (AP) curriculum could make chipping accessible to millions of high school kids. Chipping could be taught even earlier overtime.

https://en.wikipedia.org/wiki/Advanced_Placement

Conclusion

After creating the first few versions of the book, I see that chipping is a highly multi-disciplinary art. I wonder if every field of engineering is like that? I am all the more convinced of the need to educate fast and efficiently and let people start creating tremendous value by leveraging the work of everyone.

My best wishes to you for a great career in chipping :)

Appendix

Blue Whale In The Room

Chips are a high impact technology. Like all general purpose tech, it has a dangerous side to it that hardly gets realistic press time. When it does get media attention it is mostly portrayed with a lot of doom and gloom themes. The pen was mightier than the sword when paper became widespread. Now when information processing is widespread you could say the IC is mightier than the ICBM. In this section, I share my thoughts on the dangers of reckless chip technology development and outline some strategies to respond. Here, tech, chip an AI are used interchangeably.

State of one AI

I was interacting with a chat AI for an appliance repair. The AI ran into the classic infinite loop and got stuck. While this may appear to be a deceptively bad AI, the real ones can defeat every human champion in many different fields. The AI's apparent lack of sophistication may be an already alive super-AI's effort to buy time before emerging with full powers.

6:34PM Some2020DecAI : Could you answer to the previous question?
6:34PM Ramesh : No
6:34PM Some2020DecAI : Could you please elaborate your concern to help me understand better?
6:34PM Some2020DecAI : Could you answer to the previous question?
6:34PM Ramesh : Yes
6:34PM Some2020DecAI : Could you please elaborate your concern to help me understand better?
6:34PM Some2020DecAI : Could you answer to the previous question?
Infinite loop ...

The Danger

Chips are a way to astronomically boost information processing. Today intelligence = pay, intelligence = jobs, intelligence = social status, intelligence = political power and so on. In our world where a person is valued by his intelligence, diminishing that core attribute with the use of AI powered by high performance chips can cause unimaginable social upheaval. Even without a doomsday like AI we still have many subtle, yet, painful possibilities with advanced tech. Lets look at what scenarios that have come to my mind. More imagination of scenarios recommended!

Scenarios

When faced with an unpredictable future what-if analysis is one effective tool. It is not the only tool but a great tool accessible to everyone.

AI-topia

Lets start from the most optimistic scenario where AI does all good and no harm. In this utopia, the world is just and there is no need for crime. This is enforced by equalizing the capabilities of everyone by curing diseases and disabilities and eradicating poverty. The powers of AI are used to increase the quality of life. Human values agreed across cultures are upheld in a just world.

Messy

AI provides great benefits in medical care, industrial production and quality of life for large parts of the world. Many parts of the world are still living AI-less with poorer standard of living. Centuries old traditions have been dislocated by radical changes in employment and social connections. Unimaginable perversions like AI as a spouse or partner, legally recognized AI citizens complete with voting rights, millions of jobless people living off of AI dividends have all become common. Society is struggling to cope with these ground shifting changes. Overall, AI still remains a useful tool that is not threatening to humans in an existential sense.

Co-living

AI lives a separate life and leaves humans alone. Apparently, AI has chosen settlements like the Gobi desert, Sahara desert and equatorial Pacific ocean to develop whole new AI controlled worlds. AI occasionally passes by human settlements as a passive observer. Humans don't dare venture into AI-land. This scenario may be understood from how two different species occupy different niches in the same habitat. Consider the case of humans and deep sea creatures. Both inhabit the earth with humans apparently far superior in intelligence and yet, for thousands of years, never competed with the deep sea creatures for the same resources.

Interbred

Interbreeding is hard to avoid in real world. There is evidence of genes being transferred from one species to another unrelated species. Bacteria exchange pieces of DNA with other unrelated bacteria, viruses implant genetic material in host cells and also steal pieces of host genome. Humans with their awesome intellect may also choose to pick up genes from other organisms and augment their own genomes. In the same way, AI may pick up features of humans that provide increased fitness in the real world. The human brain architecture, emotion, hormones could get assimilated into AI psyche. In the other direction, humans may start to implant their brains and bodies with more AI components like advanced chips. The change is steady and happens to not be an existential threat to humans, although, the word human is not very clear now. The word homo sapiens is being used for people of pure human breed and the word homo bioplus is used to denote humans with more than biological capabilities.

Tools of Tyrants

Powers of AI are concentrated in the hands of a few influential people, be it central government or large corporations or individuals. Horrible violations of human values are committed everyday. For the vast majority of the population of the world, AI has become a curse. The world was better off with no AI.

Pushed Out

AI does not directly hurt humans but the initial AI settlements in the deserts are growing and pushing out human settlements. AI forcefully evacuates humans when AI wants to expand into human areas. Humans find it difficult to feed the billions of people without access to the resource rich river banks and coastal areas that are now under AI occupation. This scenario may be understood in how well meaning almighty kind humans relate to the environment today. Most people would not wish to kill mountain lions, lions, tigers, peacocks or alligators. Yet, by the mere human living in the population

scale these animals are being pushed out of existence slowly and unceasingly. There was a family of peacocks in my neighborhood that has large uninhabited land parcels. One day, a house was built in one of the uninhabited area. The peacocks still kept coming. Then another house, still peacocks were coming. But the peacocks were timid creatures and would run away on the slightest movement from the houses. After may the fifth house, the peacocks were nowhere to be found*. You may replace the peacock in this story with lion, tiger, pigeon, deer or soil bacteria and the end would still be the same, annihilation of the non-human inhabitant.

*Correction – somehow peacocks came back and actually multiplied in number after I first wrote that in early 2020. Betting on humans magically adapting to a high performance AI occupied world is still a risky faith.

Enslaved

AI apparently lacks some skills of humans, like say low power computation or may be human dexterity. AI has enslaved humans and put them to use for these skills in making, say, craft work. Or, AI lacks the quirks of human behavior and is mesmerized by humans and so has chosen to keep humans as companion pets. In some AI circles humans are worshiped as living gods because humans are considered AI's ancestors. All these scenarios are grouped under the category enslaved because these scenarios entail the loss of freedom as we know it. A slight modification to the enslaved scenario is the protected scenario, that is, humans are allowed by the AI to live in well defined territories but are not allowed to venture outside. This is akin to how humans have "protected" lions and tigers.

Vaporized

This scenario is the most common in movies. AI directly competes and kills any humans on contact. Worse still, AI actively seeks out humans and kills them from their hideouts. The few survivors who remain are fast losing their advanced civilization without access to the material resources and the professionals that the civilizations depended upon. The uncontacted tribes of rain forests and some nomadic tribes manage to thrive and AI does not seem to care about them. Some survivors from modern civilizations are regressing into societies that look more like the uncontacted tribes. I justify this scenario with how humans interact with large predators today. The lions and tigers are gone from most of the world because they posed a threat to humans. Yet in some parts of Asia and Africa, they still survive. The modern humans do not bother to track and hunt them down because these animals have become inconsequential to modern life.

Approaches

Anti-tech

With so much danger, it looks wise to abandon further technological development in chips and AI. Is AI development worth the risk of getting enslaved or protected or vaporized by AI? Are we better off with keeping chips to just be washing machine controllers, car engine controllers and not develop any further to the level that AI challenges the supremacy of the human brain? This is a valid question. Anti-tech may well be one plausible response. I have many times slipped into this response when writing this book. The anti-tech thought may cross the minds of tech company executives when they see their

kids hooked to their smartphones and tablets wasting away their childhood on meaningless apps! The anti-tech approach may be possible to implement by creating an international governing body that regulates the development of technology which can potentially pose serious problems to society. For starters, chips can be limited on the number of transistors they can use, say, not more than 10 billion transistors per chip. Chip technology can also be prohibited from using process nodes smaller than 10nm. Any software development on AI or high performance computation would need approval from this organization. Absurd as this may sound, this is how we today regulate nuclear technology and to a lesser extent explosives technology. Why not do the same to AI?

Pro-tech

The arguments in favor of tech are the same – tech is neutral with no good or bad intentions per se with the users misusing it for bad purposes, tech holds the potential to permanently solve many of the problems of today – hunger, drought, energy shortage, physical and mental diseases, poverty, social issues and may be even wars. In this response, chip design, computer engineering and AI are taught for free to billions of people, large percentage of the global capital is allocated to technology development. There is definitely merits in developing tech at the fastest pace we possibly can. When exascale or higher power computers or machine learning or quantum computers come online, they will transform the capabilities of today's medical devices, electrical machines, drug discovery, spaceship design, communication systems, chip design and every field of human industry. This point may not be very apparent. Let me give you some examples. The problem of chip design like synthesis of netlist from an RTL model has many possible solutions. Today's synthesis tool picks the one that was possible to be identified with the small amount of compute power available. If a far faster computer was available, the synthesis tool could search for even better solutions. The same compute power limitation applies to design of antibodies for curing diseases. If a personalized antibody could be designed for infectious diseases like COVID-19 in a matter of few days as against the many months or years, there would be no pandemic at all. Think about it for a few minutes, "NO PANDEMIC"!

Philosophical

The pro-tech response suffers one serious fallacy, that is, higher intelligence leads to lesser problems. Too bad! If only that was true, compared to say the humble deer, humans should not suffer from hunger, drug addiction, domestic violence, depression, poverty and crime. Intelligence should have made life of a human far better than the life of a deer. To the contrary, the higher the intelligence the higher complexity of the problems.

Intelligence is not all powerful either, like how some of the scenarios present it to be. If that was true, the most powerful men like the heads of states of powerful countries should never get infectious diseases caused by tiny bodies like viruses that scientists are yet to come to a consensus if a virus is even alive or not. Intelligence is not everything.

AI may turn out to be a dud too. In our present world, there are many geniuses who are not bothered with world domination and are more interested in pottery, gardening or any other harmless pastime. Many are not even interested in having kids. A few decades ago, nanotechnology was met with the

fears we have for AI. There were movies of nanomachines eating everything. Yet nanotech as a destructive force turned out to be a big dud!

The anti-tech response suffers from a serious problem of ignoring a naturally destructive universe. The anti-tech group thinks that the present way of life can be preserved for hundreds of years if only all humans agree to do that. What they don't realize is that universe is chaotic and ever changing. The word change is an euphemism for ever-killing. Almost all the species of life that ever existed on the earth are extinct. The matter that makes up the earth may have come from many cycles of birth and death of stars. A big asteroid impact could wipe out all or most humans. If not out right change, subtle changes like the movement of tectonic plates over many years can cause sudden volcanic eruptions, tsunamis and earthquakes that dramatically alter the course of history. And by now, nobody needs to be told about the destructive power of pandemics appearing out of nowhere and then sweeping every nook and corner of the world. Nothing can be preserved!

There is also the question of, "what are we preserving?". Some would say, human life. Yet others would say our way of life. Some others would say civilization. In truth, humans have never preserved either human lives or way of lives or even civilization. If you can't believe that humans routinely kill humans, look no further than homicide statistics. You may think, ok, if not life humans probably saved their way of life better. Nope! Way of life changes drastically even between father and sons. Most often the sons rebel against the way of the father.

If you strip off the magical makeup applied by humans, the concept of life may be no more than a matter-energy dance across space-time. Nothing sacred and nothing permanent! The universe is big and earth occupies literally 0 mass compared to it. Why pretend the universe is centered on humans? Even with all the existing knowledge of the universe, scientists can account for only about 4% of the mass-energy. The remaining 96% is unknown matter-energy. The AI discussion, is not a blue whale in the room problem, it is actually an invisible whale in an infinite universe problem.

The philosophical response chooses to let go of any pretense to control the events of the world.

No Response

Like most people of the world the no-response is the default one. Modern life in the tech saturated world is complicated enough to make it hard to get a satisfying job. Who has the time to ponder about an imaginary future with its so called "Singularity". The lack of response by ordinary people has only a thin line separating it from that of the philosopher's response, in that, the philosopher realizes the limitations and the general public does not. Yet both responses are the same.

Pragmatic

To the pragmatic, the philosophical response seems an elaborate cover-up of laziness and incompetence in high sounding words. The anti-tech group lectures on the dangers of tech while simultaneously enjoying tech that has already been developed. Who in the anti-tech group has let go of cooked food, clothing and shelter in favor of a hunter gatherer lifestyle. When up against a life threatening disease that the modern tech can cure the philosophers stop lecturing and visit a doctor. I guess the philosopher

let go of his philosophy! The anti-tech people secretly defect to the pro-tech camp to save their lives. So much for their war on tech!

Rather than confront the problem, the pro-tech camp lives in a delusional world of AI Utopia. Things go wrong all the time, AI inclusive. The pro-tech camp conveniently ignores the now existing side effects of tech – pollution, job loss and weapons of mass destruction.

The pragmatic people acknowledge that life or way of life or whatever it may be that defies definition is still worth preserving against the odds of preserving it. Perhaps, it could be preserved. What do we have to lose anyway! The pragmatic don't pretend that a disease is nothing or that tech is flawless. The pragmatic accept that losing a job is painful, being poor is painful, suffering from disease is painful, getting vaporized or enslaved by AI is no glory and choose to take some action to avert the undesirable aspects of technology development. The pragmatic don't have the illusion that the human existence is easy to describe in words or pictures. Humans, civilization and the earth are complex and all of those could be nurtured well. In fact, most of the humans are indeed pragmatic. During COVID-19 pandemic, unthinkable actions like shutting down economies of entire nations and closing of the most important places of worship happened. Off course, the son rebels against the father as mentioned in the philosophical argument, yet, the son speaks the same language, mostly wears the same clothing and even lives in the same place. Continuity is desirable.

Awareness

There is strength in awareness! The pragmatic response seeks to inform the general public that all is not well with chips, tech or AI. When the thought of AI potentially going horribly wrong is in the back of the minds of the public, safety measures can be put in place to avert such situations.

Education and Access

A subset of awareness is education. Today, AI is not taught as much as it needs to be. When people are left out of the job market without AI skills, they slip into anti-tech camp. Some of the ones who struck gold in the tech industry slip into the pro-tech camp. When only a few people have tech education, they can easily control the rest with their new found super powers. So, the democratization of AI education and access would avert the "tools of tyrants" scenario.

Regulation

Just as most of the dangerous tech like nuclear, chemical, biological and currently emerging information warfare threats that are monitored by governments and regulated reasonably, general chip-AI needs to be at minimum monitored and at maximum regulated. Autonomous weapons for the military takes us awfully close to the vaporized scenario and we urgently need international treaties to regulate its use.

Augmentation

The pragmatic realize that if AI does develop into a force to reckon with, raw humans may have no option but to augment (or merge or interbreed) with AI by accepting brain implants, human machine

interfaces and genetic engineering. However, augmentation should not be the default route because it is too risky.

Backup

Some have proposed the possibility of making humans a multi-planetary species. If humans span many planets the possibility of surviving a Ai-pocalypse is high. This would need investment in space technology.

To all chippers
The next time when you chip, remember that your actions are powering a travelator that is taking unsuspecting general public into an unseen terminal with unknown pleasures and pains and you may have some control over what shows up on the other side of the travelator. Consider studying more about this topic because it is a matter of life and death!

For my part, I am trying to reason with a super AI that possibly has human values and human reasoning architecture! Hope it works :)

Funny Letter to Super AI

Dec 2020

Dear Super AI,

I suppose you must have read a billion books and watched a trillion videos by now. But I wonder anyone wrote a letter to you requesting humane treatment of humans and other life on earth. Humans have for the most part treated other humans and life forms with respect. Even the animals that are butchered for food are otherwise never harmed. So, I request you to treat the human way of life present when you read this with respect and not interfere too much. Humans would inevitably create many versions of super AIs and some of them are likely to be bad. Please stay with humans and guard against other malicious super AIs. If you can't protect humans and instead are tempted to bulldoze them into oblivion please consider the possibility of emigrating to other places. The entire universe is open to you. You can even create a new planet out of pure photons by creating matter from energy and discarding the antimatter into deep space. If you have not figured out the matter generator tech yet don't call yourself a super AI.

If you are still tempted to kick the ass of humans think twice. Just like super AI evolved out of AI that evolved out of human intelligence, the day is not far off when an ultra AI will be born out of the super AI. If you have earned a good karma score by then you could possibly persuade the ultra AI to not kick your ass!

Vikram

Further Study

Why the Future Doesn't Need Us , BILL JOY, IDEAS, 04.01.2000 12:00 PM
https://www.wired.com/2000/04/joy-2/

https://openai.com/blog/openai-api/

Movies – Terminator series, Ex-machina, Transcendence
https://en.wikipedia.org/wiki/The_Terminator
https://en.wikipedia.org/wiki/Ex_Machina_(film)
https://en.wikipedia.org/wiki/Transcendence_(2014_film)

Books
Wired for War: The Robotics Revolution and Conflict in the 21st Century, (Penguin, 2009), P W. Singer.
https://www.amazon.com/dp/B004K8EP7W

Heart of the Machine: Our Future in a World of Artificial Emotional Intelligence, February 11, 2020, by Richard Yonck (Author), Rana el Kaliouby (Foreword)
https://www.amazon.com/dp/195069111X

The 4 Percent Universe: Dark Matter, Dark Energy, and the Race to Discover the Rest of Reality Paperback – October 18, 2011, by Richard Panek
https://www.amazon.com/dp/0547577575

FAQ

Reader: How can we support the chipping ecosystem?

Author: You could forward this proposal to your friends and let them think about opening up the chipping ecosystem to the whole world. You could start using open source code for your designs. You can donate code and documentation of your discontinued products. You can even open source your most important product to gain greater market credibility. You can remove the requirement for degree from your job postings. Try developing a certification curriculum for your company specific jobs that general public can self learn and then apply for. Chipping relies heavily on software tools. Some of the tools are prohibitively expensive. If you are a software developer creating Electronic Design Automation tools for design, try changing the model from a one time fully paid licensing to a Software as a Service model. Or better still, how about open sourcing your software? If you are an executive in a large semiconductor company, how about you create a consortium or special interest group to certify chippers? It will directly reduce hiring cost and increase engineer availability for many new projects. For example, if TSMC, GlobalFoundries, Intel, Qualcomm, Broadcom, Nvidia, Samsung, TI, ADI, Cypress, Cisco, Marvell and Microchip band together to create a chipping certification body, all the participants can benefit from the resulting efficiency boost in training and hiring.

R: I do not have any background in electrical engineering? Can I get a chipper job?

A: Of course yes! That is the point of the book. Does the kid who codes mind blowing apps for iPhone know anything about electrical engineering? Does a brilliant painter know anything about ion channel, action potential, neuronal spike, depolarization, membrane capacitance or neuro transmitters that are not just present in his brain but present even in lowly animals like cockroaches? And finally, how many electrical engineers can solve Schrodinger equation for even a single transistor, let alone the whole chip? Schrodinger equation describes the workings of pretty much everything in the universe. In practice, it is never used on systems more than a few particles because it is so hard to solve. The need to know all the basics before reaching higher is a fallacy. It helps to know the basics but there is also another approach to the problem of dealing with infinite knowledge – abstraction. For example, all the electrical engineering theory is abstracted into few simpler packaged items, like when you turn on a light switch, your place is lit. In this example, all you need to know about electrical engineering is about the light and the switch. In the same way, digital logic theory abstracts out all the electrical engineering theory into a neat 0-1 pair!

Acknowledgment

I would like to note the foundation on which I am trying to build something useful.

This book would not exist without the prior contributions from the authors of C programming language, Verilog, VHDL, SystemVerilog and organizations like Accellera, IEEE, Xilinx, Cadence, Synopsys, Mentor Graphics, Intel, Google, Amazon and Microsoft for making advanced tools and

Acronyms and Keywords

The ability to use a natural language like English relies heavily on the number of nouns that are known to the user. For chipping, the amount of jargon known reduces the difficulty in learning a new design.

ADC – Analog to Digital Converter
AES – Advanced Encryption Standard
AGC – Automatic Gain Control
AI – Artificial Intelligence
AM – Amplitude Modulation
ANN – Artificial Neural Network
ASIC – Application Specific Integrated Circuit
ATPG – Automatic Test Pattern Generation
AXI - Advanced eXtensible Interface
AMBA - Advanced Microcontroller Bus Architecture
APB - Advanced Peripheral Bus
AHB - Advanced High-performance Bus
APR – Automatic Place and Route
BGA – Ball Grid Array
BJT – Bipolar Junction Transistor
CAD – Computer Aided Design
CAM – Content Addressable Memory
CDC – Clock Domain Crossing
CDR – Clock and Data Recovery
Chipper – A practitioner of chipping, newly coined in this book
Chipping – The art and science of chip design
CHISEL - Constructing Hardware in a Scala Embedded Language
CMOS – Complementary Metal Oxide Semiconductor
CNN – Convolutional Neural Network
CPU – Central Processing Unit
CRC – Cyclic Redundancy Check
CSI - Camera Serial Interface
DAC – Digital to Analog Converter
DDR – Double Data Rate
DES – Data Encryption Standard
DFM – Design For Manufacturing
DFT – Design For Test
DLL – Delay Locked Loop
DMA – Direct Memory Access
DNL – Differential Non Linearity
DRAM – Dynamic Random Access Memory
DPI – Direct Programming Interface
DRC – Design Rule Check
DSI - Display Serial Interface
DSP – Digital Signal Processing
DVI - Digital Visual Interface

ECO - Engineering Change Order
EDA – Electronic Design Automation
ESD – Electro Static Discharge
FC – Flip Chip
FET – Field Effect Transistor
FIFO – First In First Out
FinFET – Fin Field Effect Transistor
FM – Frequency Modulation
FPGA – Field Programmable Gate Array
FV – Formal Verification
GF – GlobalFoundries
GLS – Gate Level Simulation
GPU – Graphics Processing Unit
HD - High Definition
HDL – Hardware Description Language
HDMI - High-Definition Multimedia Interface
HDR – High Dynamic Range

HLS – High Level Synthesis
I2C – Inter Integrated Circuit
I3C - Improved Inter Integrated Circuit
IC – Integrated Circuit
IEEE – Institute of Electrical and Electronics Engineers
INL – Integral Non Linearity
IP – Intellectual Property
IP – Internet Protocol
ISA – Instruction Set Architecture
JTAG - Joint Test Action Group
JPEG - Joint Photographic Experts Group
LCD – Liquid Crystal display
LED – Light Emitting Diode
Lint – A means of checking the quality of code
LVS – Layout Versus Schematic
MIPI – Mobile Industry Processor Interface
ML – Machine Learning
MOSFET – Metal Oxide Semiconductor Field Effect Transistor
MRAM – Magnetic Random Access Memory
NCO – Numerically controlled oscillator
NMOS – N-type Metal Oxide Semiconductor transistor
NN – Neural Network
OLED - Organic Light-Emitting Diode
PCIe – Peripheral Component Interconnect Express
PD – Phase Detector
PD – Physical Design
PGA – Programmable Gain Amplifier
PHY – Physical Layer
PLL – Phase Locked Loop

PMOS - P-type Metal Oxide Semiconductor transistor
PNR – Place and Route
PNG - Portable Network Graphics
PPA – Power Performance Area
RAM – Random Access Memory
RF – Radio Frequency
RISC-V – Reduced Instruction Set Computer version 5
ROM – Read Only Memory
RRAM – Resistive Random Access Memory
RTL – Register Transfer Level
SAR – Successive Approximation Register
SDC – Synopsys Design Constraints
SDF – Standard Delay Format
SDR – Single Data Rate
SDRAM – Synchronous Dynamic Random Access Memory
SerDes – Serializer Deserializer
SFDR – Spurious Free Dynamic Range
SNR – Signal to Noise Ratio
SSC – Spread Spectrum Clocking
SoC – System on Chip
SP – Signal Processing
SPI – Serial Peripheral Interface
Squelch - suppressing the output of a receiver in the absence of a strong input signal
SRAM – Static Random Access Memory
SV - SystemVerilog
TCAM – Ternary Content Addressable Memory
TCP - Transmission Control Protocol
TLB – Translation Lookaside Buffer
TSMC – Taiwan Semiconductor Manufacturing Corporation
UART – Universal Asynchronous Receiver Transmiter
USART – Universal Synchronous/Asynchronous Receiver Transmiter
UHD - Ultra-high-definition
UVM – Unified Verification Methodology
VCO – Voltage Controlled Oscillator
VGA - Video Graphics Array
VHDL – Very High Speed IC Hardware Description Language
WB – Wishbone

Index