A Comparative Study of CHISEL and SystemVerilog, based on Logical Equivalent SweRV-EL2 RISC-V Core

Junaid Ahmed - Lampro Mellon, Lahore, Pakistan, ahmedjunaid339@gmail.com
Waleed Bin Ehsan - Lampro Mellon, Lahore, Pakistan, waleed.bin.ehsan@gmail.com
Laraib Khan - Lampro Mellon, Lahore, Pakistan, laraib.khan@lampromellon.com
Asad Aleem - Lampro Mellon, Lahore, Pakistan, asad.aleem@lampromellon.com
Agha Ali Zeb - Lampro Mellon, Lahore, Pakistan, agha.ali@lampromellon.com
Sarmad Paracha - Lampro Mellon, Lahore, Pakistan, sarmad.paracha@lampromellon.com
Abdul Hameed - Lampro Mellon, Lahore, Pakistan, abdulhameed.akram@lampromellon.com
Aashir Ahsan - Lampro Mellon, Lahore, Pakistan, aashir.ahsan@lampromellon.com

Abstract

A detailed comparative study is done over an open source RISC-V core (SweRV-EL2) by Western Digital Corporation implemented in SystemVerilog (2009) with its logical equivalent implementation in CHISEL (Constructing Hardware In Scala Embedded Language) [1] . The CHISEL implementation of the SweRV-EL2 is named ”Quasar” open-sourced by Lampro Mellon. SweRV-EL2 is used as a golden reference model for RTL (Register Transfer Logic) to RTL LEC (Logical Equivalence Check), since it is already silicon-proven therefore reducing the verification time. This strategy reduces time as there is no need to build a pervasive verification infrastructure. Module-level verification is done using directed tests to ensure correct functionality, and then the core is verified by using a co-simulation environment. Core functional coverage is achieved using available test suite, directed test cases, and Google-DV generated tests. After the verification, both implementations are subjected to LEC using Synopsys Formality. The key points of the design are mapped and compared. Qualitative and quantitative comparisons are made between both implementations. The key metrics for the qualitative comparison are code readability, code density, and code maintenance. The key metrics for quantitative comparison are silicon area, maximum operating frequency, dynamic power and simulation performance. CHISEL implementation delivered nearly the same PPA (Power Performance Area) results while observing drastic improvement in code readability, code density, and code maintenance.

Index Terms

RISC-V, SystemVerilog, CHISEL

I. Introduction

We started with the development of the CHISEL and Scala language manual [2] and contributed to the open-source community. After the successful CHISEL-based implementations of the Rocket Chip generator [3] and Google Tensor Processing Unit [4], it was decided to implement the Western Digital Corporation's SweRV-EL2 RISC-V core in CHISEL. SweRV-EL2 is a four-stage scalar, mostly in order pipelined core supporting only machine mode. It supports RV32IMC instructions. The source code of CHISEL implementation is available on the GitHub [5]. Quasar is implemented in CHISEL because it provides flexibility to hardware development and several ways to accelerate the design of complex modules with remarkable industry-academia success. It introduces many features and concepts intended to improve hardware design efficiency, especially useful for designing complex IPs and projects. It contains both functional and OOP (Object-Oriented Programming) attributes to enhance hardware design flexibility and reusability of different complex RTL modules required in almost all projects. HCL (Hardware Construction Language) resembles the domain-specific languages leveraging the flexibility to design the hardware at a higher abstraction level.

Flexibility to hardware development always remained a long-term goal for hardware communities dealing with limitations of current hardware description languages like Verilog, SystemVerilog, or VHDL (Very High Speed Integrated Circuit Hardware Description Language). The problem seems to start being addressed with the advent of PyRTL (Pythonic Register Transfer Level) based on Python, BlueSpec based on Haskell, System-C libraries used for HDL (Hardware Description Language) modeling, Xilinx HLS (High-Level Synthesis) tool used to develop models in C language. HLS raises the abstraction level and can compile C/C++ functions to logic elements. MATLAB provides an HDL Coder toolbox that generates synthesizable Verilog and VHDL code used for embedded development, generally for FPGAs (Field Programmable Gate Arrays).

This study compares the two implementations of SweRV-EL2 to verify the advantages of using CHISEL over traditional HDLs. The process began with the implementation of smaller modules which were verified with directed tests. Once verified, the CHISEL generated module replaced its corresponding block in SweRV-EL2 core, and this implementation was verified with our test suite that also included Google-DV generated tests. Once all the modules had been verified, the entire CHISEL generated core was tested with SweRV-EL2 using co-simulation and LEC.

In the rest of the document, the literature review is discussed in section II followed by the overview of the architecture of Quasar in section III. The methodology observed throughout the project is detailed in section IV. The experimental results along with the comparisons and findings are discussed in section V. At the end, a conclusion is drawn based on the results and the adopted methodology.

II. LITERATURE REVIEW

The hardware design industry revolves around two major languages namely VHDL and Verilog but both of these lack a key feature that is the flexibility. Though the induction of SystemVerilog was a silver line, it wasn't as fruitful as several other languages adopted for software development. One of the major reasons for ineffectiveness was that most of the constructs aren't supported by synthesis tools. SystemVerilog was introduced to improve flexibility, design, and verification flows but the hardware community is still demanding some agile approach for development. Various languages have emerged in the market to assist in the development. Jonathan Bachrach et al. [1] introduced CHISEL, a new hardware construction language embedded in Scala that includes object-oriented programming, functional programming and many other useful features to enhance design productivity and reuseability. CHISEL allows the designers to write highly parameterized circuit generators and provides the testing framework to verify the functionality, at unit and system levels. It can be done without generating Verilog from CHISEL. CHISEL offers a standard library where different types of interfaces, functional blocks like FIFOs, shift registers, decoupledIO, etc hardware constructs have been predefined. Scala is a strongly-typed language; this property enables CHISEL users to define and reuse their specific functions. To add to its flexibility, CHISEL also includes some additional features like type and width inference.

Paul Lennon et al. [6] performed a comparative analysis between CHISEL and Verilog using three different RTL modules. In his analysis, first RTL is the Round-Robin Arbiter, second is N-bit FIFO (First In First Out) and third is Round-Robin Complex Scalable Arbiter. The metrics evaluation include code density, design flow runtime, silicon area, and the maximum frequency of operation. Two different FPGAs of Microsemi families are used for exploring the differences. In the case of Round-Robin Arbiter, the ratio of Verilog to Scala lines of code is 1.18, design flow runtime did not vary significantly and, there is a 17% boost using Verilog implementation in the maximum operating frequency has been measured on SmartFusion2 FPGA device. In the case of FIFO, The lines of Verilog code in both cases are approximately 90, while design flow runtime is measured a little less in the case of CHISEL. There is no significant change in resource utilization and maximum operating frequency in the case of FIFO. It is observed that in the case of Scalable Complex Arbiter, the Verilog implementation is three times longer than the CHISEL implementation. Verilog outperformed CHISEL by a significant difference of 5% during the design flow runtime and, the maximum operating frequency is almost identical when executing Verilog or CHISEL on both devices.

Jean Bruant et al. [7] built an open-source automated SystemVerilog to CHISEL translator. They parsed a synthesizable SystemVerilog file using ANTLR [8], which builds an abstract syntax tree that eventually maps on custom IR. This work has four types of analysis and transformations: clock inference, reset inference, types inference, and SystemVerilog syntactic-sugar translation. Finally, the emitter outputs CHISEL code.

Python language has gained traction in the last few years owing to its ease of use in machine learning, artificial intelligence, and in various area. PyRTL was introduced by John Clow et al. [9] based on Python to improve the development of complex architectures and their optimizations. In this approach, the python embedded hardware design language serves as a wrapper over a well-defined set of parameters. The motivation for this approach was that traditional hardware languages have a long learning curve. It supports signed types, hierarchies of wires using bundles and has well-defined control structures. Similar to PyRTL, MyHDL introduced by J. Decaluwe et al. [10] is a set of libraries to provide hardware designers with the simplicity and elegance of Python language. It is used for hardware modeling, simulation, verification, and VHDL/Verilog code generation. Clash is an open-source hardware description language introduced by Christiaan Baaij [11] based on Haskell used for the hardware modeling. It has a compiler that generates synthesizable Verilog, SystemVerilog, and VHDL codes. It provides different libraries for modeling hardware at a higher abstraction level.

The methodologies discussed above are not widely adopted; thus, verification of the design is a challenging task for CHISEL users. A conventional approach for verification would be to generate the Verilog of the design and then verify it using conventional verification methods. However, this method would not be efficient for large-scale designs in CHISEL as interpreting the generated Verilog is quite a tedious task. To resolve this problem, a flow named ChiselVerify is proposed by Andrew et al. [12]. It focuses on the verification efficiency and provides an integrating framework with any hardware language compatible using black boxes. Moreover, it is focused on the backwards compatibility with existing CHISEL environments. It is based on Universal Verification Methodology (UVM) and SystemVerilog. It is used to enhance software productivity and verification of the digital design. The tool uses the power and features of Scala to drive a verification process much similar to UVM. The proposed solution greatly increases a verification engineer's productivity as it allows the design to be tested using modern high-level programming languages. The tool uses the black-box feature of CHISEL to incorporate any language for testing purposes. Additionally, it is specified in [10] that ChiselVerify is quite similar in terms of verification using SystemVerilog as it is capable to compute functional coverage, constrained random verification and bus-functional models, etc.

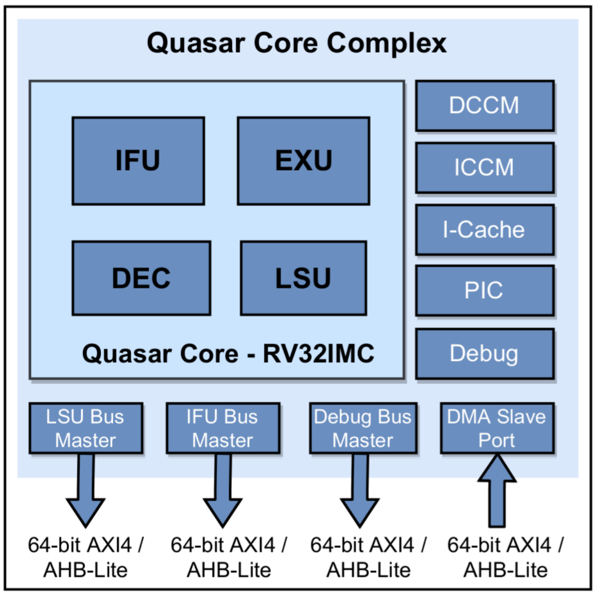

Figure 1: Quasar core complex.

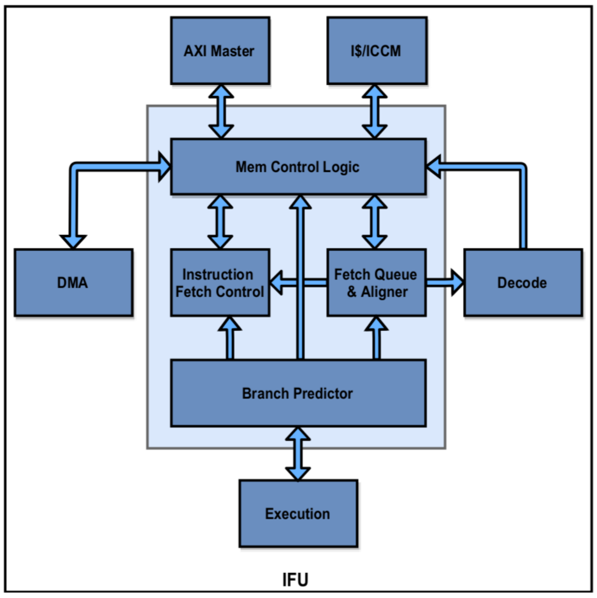

Figure 2: Instruction fetch unit.

III. SWERV-EL2 (QUASAR) OVERVIEW

Quasar is a four-stage scalar, mostly in-order pipelined core supporting only machine mode. The four pipelined stages are: Fetch (F), Decode (D), Memory and Execution (M / X), and Retired (R). The high-level block diagram is shown in Fig 1 and pipelined flow is shown in Fig 3.

The core is divided into four major blocks IFU (Instruction Fetch Unit), EXU (Execution Unit) , LSU (Load Store Unit) and DEC (Decode Unit). DCCM (Data Closely Coupled Memory), ICCM (Instruction Closely Coupled Memory), and I-cache are located on the same chip. Core also has PIC (Programmable Interrupt Controller) and Debug Unit modules. There are four AXI / APB configurable ports to communicate with external environment: DMA (Direct Memory Access) slave port, debug master bus, IFU master bus and LSU master bus. Detailed discussion of four major blocks is given below:

A. Instruction Fetch Unit (IFU)

IFU consists of four submodules, as shown in Fig 2. First is the IFC (Instruction Fetch Control). It is responsible for making fetch requests and calculating new PC (Program Counter). The PC values are then passed to the branch predictor and memory controller. A fetch request is made when there are no halt, stall, and flush signals. IFC also controls the fetch queue operations. Whenever the queue is full, an event will be triggered to the PMU (Performance Management Unit). The second is the branch predictor; it is responsible for predicting the next PC value and the direction of the branch. Quasar has a 2-bit history table and a RAS (Return Address Stack) to cater calls and returns. Third is the memory controller; it controls I-cache, ICCM, and stalls the lower pipe when a miss occurs in I-cache or an un-correctable error occurs in the data fetch. Fourth is the aligner, in order to support 16-bit compressed instructions in addition to normal 32-bit instructions. In the aligner there is a fetch queue of depth three and a shifter module that aligns the data on every fetch request. If there is a 32-bit instruction right after a compressed instruction, the aligner aligns the top half of the instruction to the lower half of the next instruction. In addition, it aligns data coming from the branch predictor.

B. Decode Unit

The decode unit is responsible for decoding the instruction received from IFU. It also controls clock gating logic for power optimization purposes and collects the trace data that holds the information required for tracing an instruction or the exception. It consists of five sub-modules. First is the Instruction Buffer Control. It controls whether the instruction from debug or from the IFU is going to be forwarded. Second is the Decode Control Unit; it is responsible for decoding the instruction and controlling the data path. It has the capability for illegal instruction handling, performance monitoring units, scheduling logic, CAM (Content Addressable Memory) is used to control

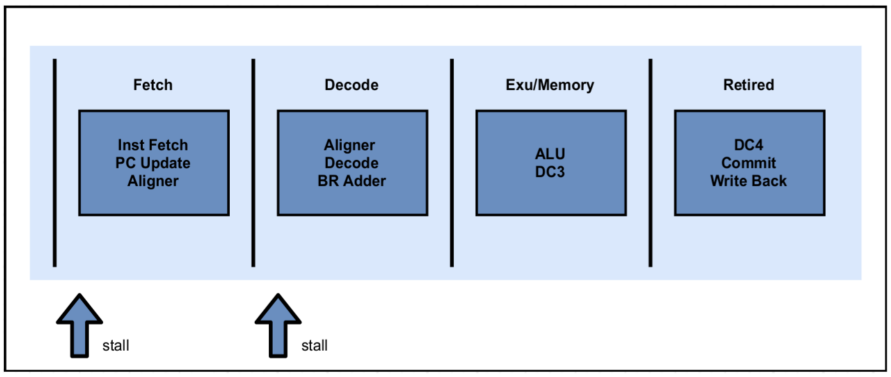

Figure 3: Pipeline stages of Quasar core.

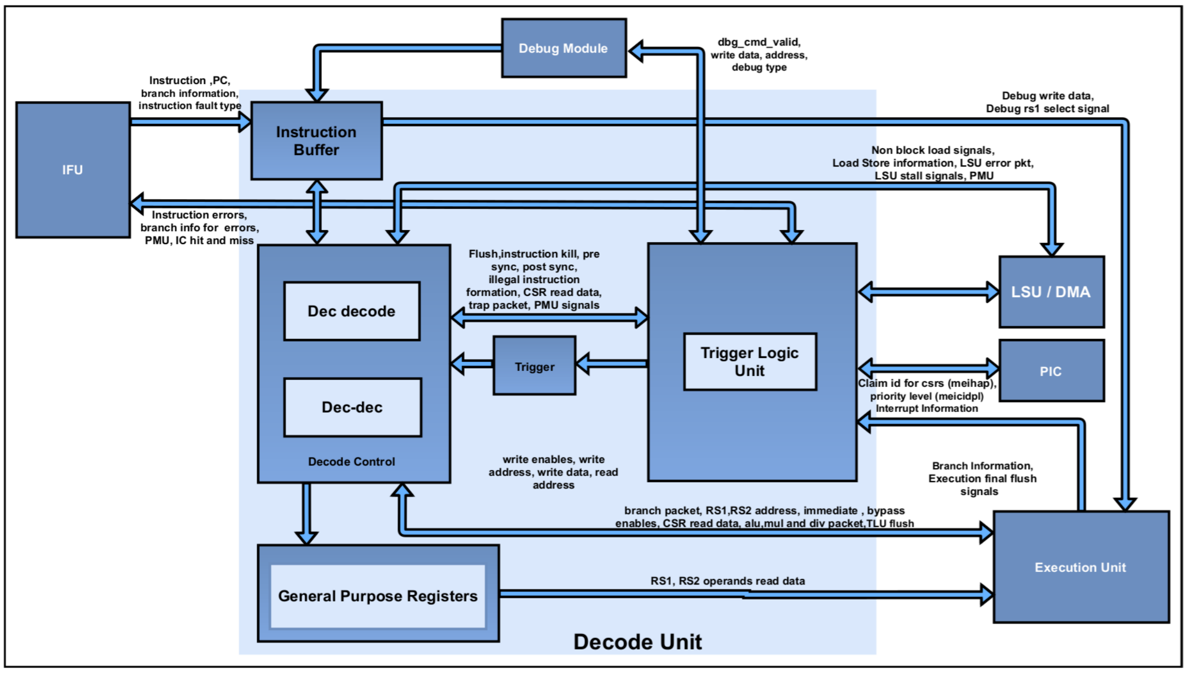
Debug Module
dbg_cmd_valid, write data, address, debug type
Instruction ,PC, branch information, instruction fault type
IFU
Instruction Buffer
Debug write data, Debug rs1 select signal
Non block load signals, Load Store information, LSU error pkt, LSU stall signals, PMU
Instruction errors, branch info for errors, PMU, IC hit and miss
Flush,instruction kill, pre sync, post sync, illegal instruction formation, CSR read data, trap packet, PMU signals
Dec decode
Trigger
Trigger Logic Unit
LSU / DMA
PIC
Claim id for csrs (meihap), priority level (meicidpl) Interrupt Information
Dec-dec
Decode Control
write enables, write address, write data, read address
Branch Information, Execution final flush signals
branch packet, RS1,RS2 address, immediate , bypass enables, CSR read data, alu,mul and div packet,TLU flush
General Purpose Registers
RS1, RS2 operands read data
Execution Unit
Decode Unit

Figure 4: Decode unit block diagram.

non-blocking loads and traps. Third is the TLU (trigger logic unit); all CSRs of machine and some custom CSRs and exception/interrupt handling logic is implemented in this module. Fourth is the GPR (General Purpose Registers) and fifth is Trigger unit, it takes input packets from the TLU and compares them with the instruction PC and gives result to the decode control. A packet of trace information is constructed which holds the information required for tracing an instruction or the exception.

C. Execution Unit

Execution unit is responsible for executing arithmetic and logical operations. It receives data and control packets from the decode unit and perform execution accordingly. Bypass muxes have been implemented for bypassing the data to the LSU unit. Fig 5 shows the block diagram of execution unit. It has three submodules: ALU Control; it can perform addition, subtraction, less than, greater than, equal to, greater equal, AND, OR, XOR and all types of shift operations. It supports both conditional and unconditional branching. Logic has been implemented to detect whether the branch is taken or not and it also generates flushing requests if branch is wrongly predicted. Second submodule is Multiplier; it is used to multiply two signed 32-bit numbers (33 bits total, one for sign bit). A packet is constructed in the decode unit which configures the control signals for multiplication. The third is the divider. A multi-cycle out-of-pipeline divider does the execution of division in the core. In the divider unit, the division of smaller numbers i-e 4-bits/4-bits is done by simple binary division in one cycle. In contrast, the division of numbers greater than 4 bits are done by the Single bit Non-Restoring Division algorithm.

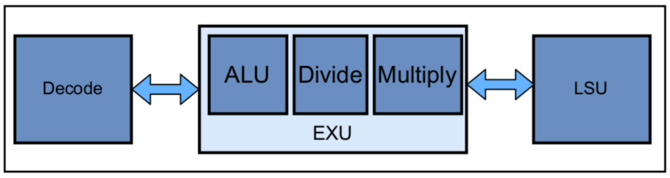

Figure 5: Block diagram of execution unit.

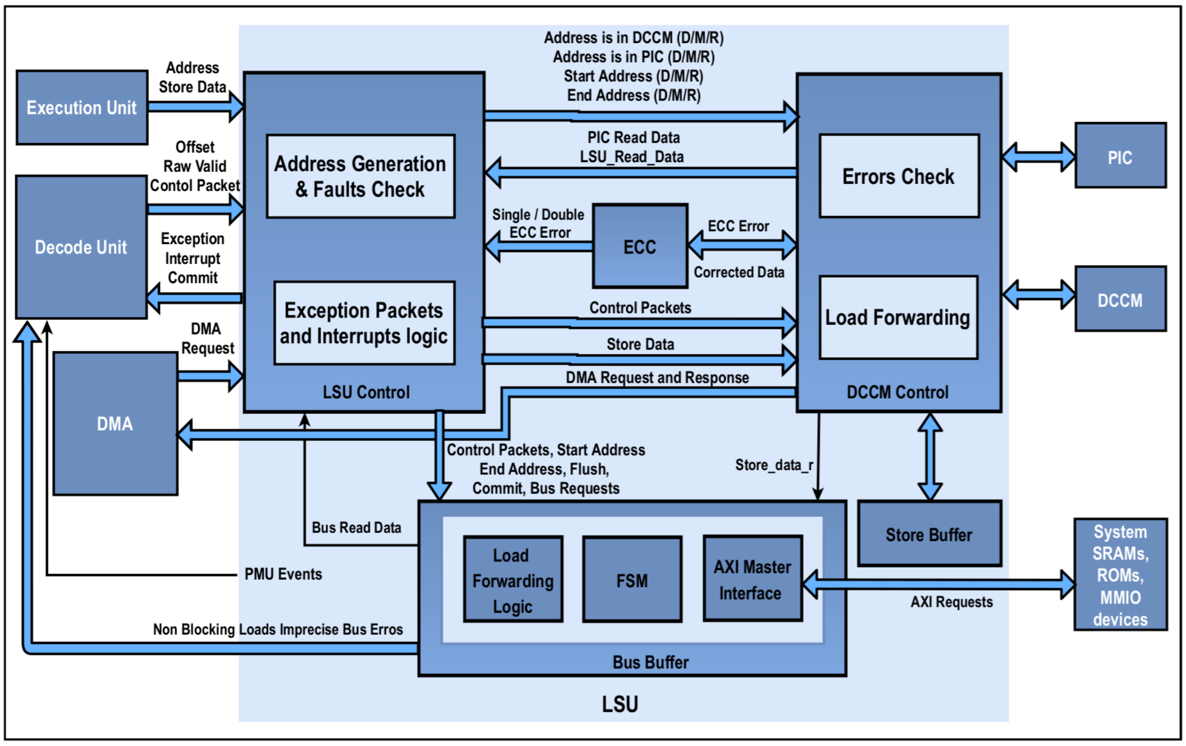

Figure 6: Block diagram of load store unit.

D. *Load Store Unit*

The Load Store Unit (LSU) executes load and store requests coming from the decode. It consists of six sub-units: LSU control unit, DCCM control unit, Bus buffer, Store buffer, ECC (Error Correction Code) module, and Trigger unit as shown in Fig 6. There are two sources from which the LSU control unit can receive load and store requests. The first is DMA (Direct Memory Access) and the second is the decode; in both cases datapath is controlled by the decode unit. The decode unit decides which request is going to be processed. There is a DCCM port of the LSU that is connected with DCCM memory. All stores to the DCCM are passed through store buffer to minimize store miss latency and also forward loads according the RISC-V memory consistency model. Load data can be fetched from DCCM, PIC, or store buffer. There is an AXI / AHB-Lite master bus to perform load and store operations on external components like SRAM, ROM, peripherals, and IO's. All external address requests are passed through the bus buffer which is connected with AXI/ AHB Lite bus.

IV. Methodology

The development of Quasar is categorized into four major parts. The first one includes comprehending the micro-architecture of SweRV-EL2 core, which lead to a compilation of the MAS (Micro Architecture Specifications) document. Once the document was completed, then the core was implemented in CHISEL. For verification purpose, a co-simulation environment was developed, which compared the log files of both implementations. Tests that have been written yielded a code coverage of 90%. It was evaluation that to get 99% code coverage much effort is required. So, instead of achieving conventional verification goals, LEC was performed. The purpose of this work is to build an equivalent core using CHISEL, the LEC is what was the most suitable as a first attempt. So, to further solidify the verification, LEC was performed on both implementations and SweRV-EL2 is used as a golden reference model. These steps are further elaborated below:

A. Design Development

FIRRTL (Flexible Internal Representation for RTL [13]) serves as a platform for writing circuit level transformations. Hence the design was initially written in CHISEL and is then transformed by FIRRTL to synthesizeable Verilog. The design for the core was done in such a manner that once FIRRTL generates Verilog, it should represent the same hardware as that of SweRV-EL2.

B. Functional Verification

Verilog for the individual modules were generated and then verified by using unit-level tests written in SystemVerilog. These tests cover the basic functionality of the module as well as corner cases. After the preliminary verification, each module was plugged in the original SweRV-EL2 core replacing its counterpart. A test suite consisting of various RISC-V, Google-DV, and benchmarks was executed to identify any divergence from desired functionality. Once Quasar was completed, it was verified using a co-simulation environment by subjecting them to a diverse range of tests. Under this environment, post-simulation logs for each module were compared to ensure functional equivalence

C. Logical Equivalence Check

The formal verification technique uses mathematical modeling to prove that two different design implementations exhibit the same behavior by performing logical checks of flop-to-flop. Setting up the design for LEC takes library elements, reference model and implemented design as input to perform mathematical modeling. The CHISEL generated Verilog (implemented design) and original SystemVerilog (reference design) are added as implemented design and reference model respectively, and their top modules are set. The initial step reads inputs of the design, creates data structures and memories are considered as black boxes to facilitate underlying steps. It is split into several logical cones. A logical cone is a block of combinational logic that drives the same compare point. The inputs to a logical cone may include primary input pins of the module, output ports from the register, and output ports connected to memory blocks. Compare points usually comprise of registers, primary output ports and input ports to a black-boxed section. Due to signal renaming in generated Verilog, 15281 signals are user-matched, including 1319

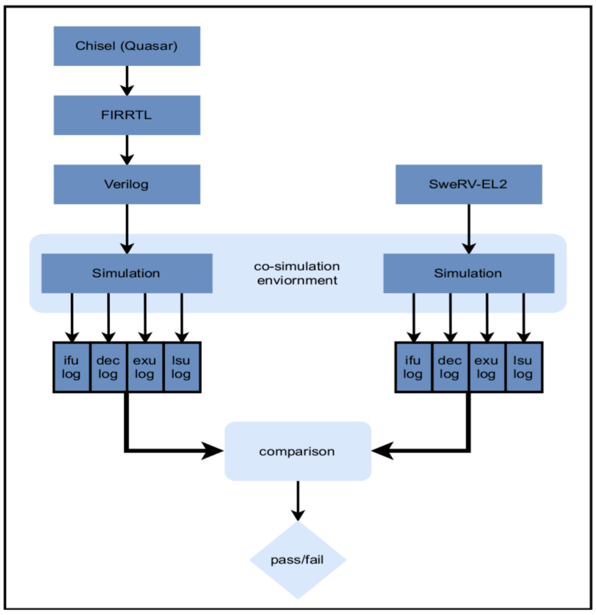

Figure 7: Functional verification method adopted.

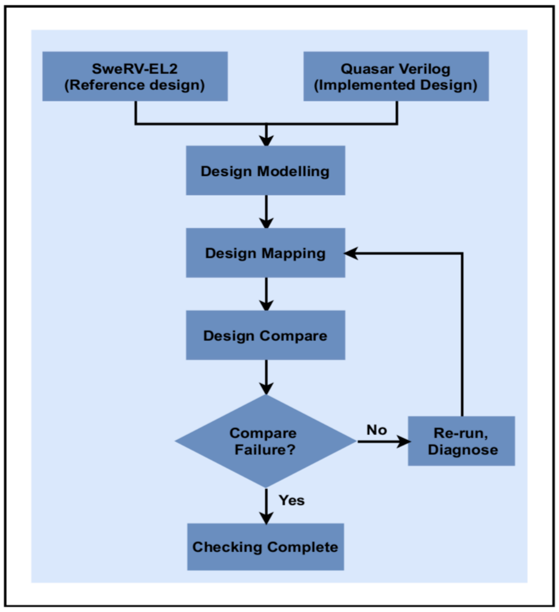

Figure 8: LEC verification flow.

ports, 13784 DFFs (D-Flip-Flops), 122 Latches, and 56 black box pins. Once the setup was complete, both designs were compared and their key points were examined. The result can be one of the three i.e. equivalent, inverted equivalent or nonequivalent. If the result is any of the latter two, then the design needs to be revised and rechecked to get logical equivalence. First, the method was performed with individual modules leading to entire blocks, and then complete cores were provided as a reference as well as implemented models to the tools for performing LEC.

In this particular case, we already have a reference model to verify our design. However, if there is no such a model available then verification can be performed by the development of functional equivalent model in some high level language like C/C++ or by exploiting the layered test-bench approach. There is some additional work in the layered test-bench approach to write cover points and assertions for functionality and sanity checks. Additionally, an open source tool named ChiselVerify is proposed in [12] can be used for this type of verification approach.

V. Experimental Results

A frequency sweep was performed using 130 nm SkyWater-PDK library on both Quasar and SweRV-EL2 to find their respective maximum operating frequencies. The tool used for this sweep was Synopsys Design Compiler (DC), and for simulations is Synopsys VCS. There is no significant difference in the execution time of both the implementations is observed. Through the frequency sweep, three key matrices are evaluated. A detailed description of the results is given below:

A. Maximum Frequency

Both implementations are subjected to a frequency sweep from 25 MHz to 350 MHz. Quasar reached a maximum frequency of 141 MHz at the target frequency of 150 MHz, whereas, for SweRV-EL2, it was 145 MHz at a target frequency of 225 MHz shown in Fig 9. These results show SweRV-EL2 is about 2.7% better in frequency with respect to Quasar. Table I shows the frequency sweep from 25 MHz to 350 MHz and both implementation's effective frequencies.

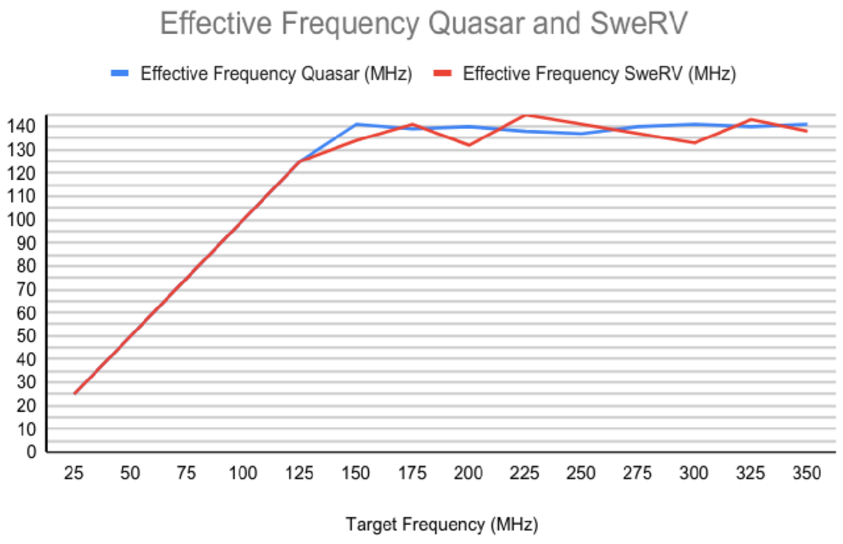

Figure 9: Target frequency comparison

Table I: Operating frequency comparison of both cores

Target Frequency (MHz)	**Effective Frequency**	
	Quasar (MHz)	***SweRV (MHz)***
25	25	25
50	50	50
75	75	75
100	100	100
125	125	125
150	141	134
175	139	141
200	140	132
225	138	145
250	137	141
275	140	137
300	141	133
325	140	143
350	141	138

Table II: Comparison of cell area of both cores

Target Frequency (MHz)	**Total Cell Area**	
	Quasar (um^2)	***SweRV*** (um^2)
25	938,648	952,117
50	986,381	1,008,292
75	985,884	1,040,223
100	992,014	1,056,209
125	1,022,198	1,058,341
150	1,050,860	1,074,543
175	1,050,117	1,077,312
200	1,048,001	1,093,162
225	1,044,916	1,097,709
250	1,047,533	1,096,681
275	1,049,092	1,095,016
300	1,050,784	1,093,012
325	1,054,411	1,094,366
350	1,051,685	1,096,023

B. Silicon Area

Silicon area for both SweRV-EL2 and Quasar are measured at the target frequency where the respective implementations have maximum operating frequency. For Quasar, the area measured at the target frequency of 150 MHz is 1050860 um^2 and for SweRV-EL2, area at the target frequency of 225 MHz is 1097709 um^2 as shown in Fig 10 and in Table II. Results show that Quasar is 4.2% smaller in area relative to SweRV-EL2.

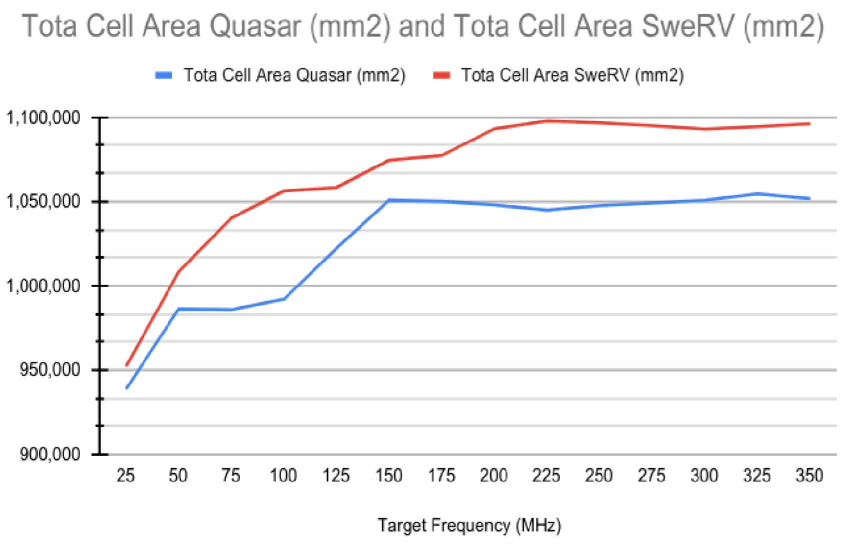

Figure 10: Graph showing comparison of silicon area.

C. Dynamic Power

Dynamic power for both the implementations is also calculated at maximum operating frequency. The maximum operating frequency for Quasar is 141 MHz and for SweRV-EL2 is 145 MHz. By linearly increasing the values of target frequency, dynamic power at maximum operating frequency for Quasar is 80.68 mW and for SweRV-EL2 is 83.95 mW, as shown in Fig 11 and in Table III. Results show that Quasar is 3.8% better in power at maximum operating frequency relative to SweRV-EL2.

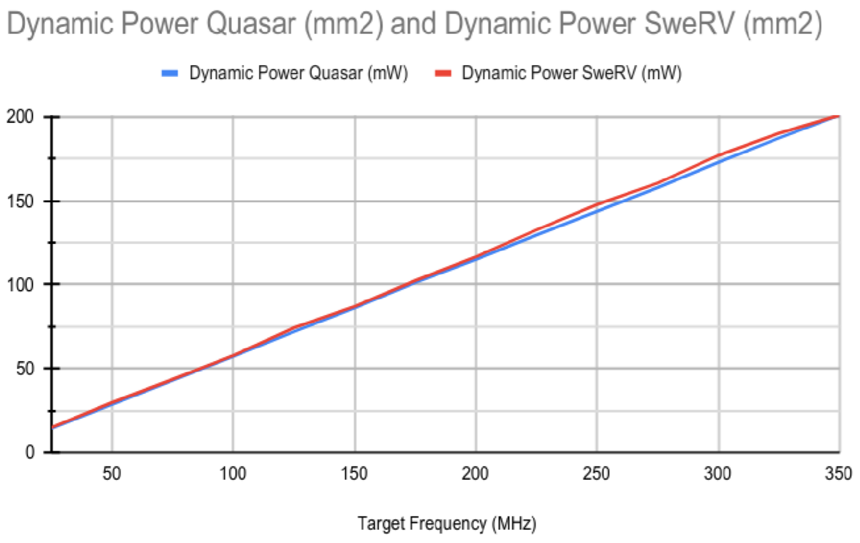

Figure 11: Comparison of dynamic power.

Table III: Comparison of dynamic power consumed

Target Frequency (MHz)	**Dynamic Power**	
	Quasar (mW)	***SweRV (mW)***
25	14.32	14.87
50	28.66	29.87
75	43.07	43.60
100	57.33	57.85
125	71.99	74.27
150	86.11	87.03
175	101.19	102.41
200	114.97	116.60
225	129.49	132.57
250	143.70	147.78
275	157.83	160.52
300	172.89	177.03
325	187.33	190.15
350	201.05	201.05

D. Code Maintenance and Readability

Maintenance of code in the context of this paper reflects how easily, and effectively crucial portions of the code are updated. In CHISEL, abstraction of describing behavior at a higher level leads to a reduced number of code lines. It is observed that the CHISEL implementation for Quasar has about 35-40% fewer lines of code as compared to SystemVerilog implementation of SweRV-EL2. It results in better maintenance because compact code is very easy to update.

The readability of code in the context of this paper means how easily a person can understand the micro-architecture just by reading the code. It has two dimensions: the readability of the CHISEL code itself, and the second is the readability of generated Verilog code using FIRRTL. An assumption made is that the person reading the code is well versed in CHISEL and SystemVerilog. For such a person, he/she would feel very easy in reading the code in CHISEL as compared to SystemVerilog. The fundamental reason behind this is the abstraction level. A reader of code can easily and readily grasp the ideology of the micro-architecture from a higher abstraction compared to a reader, reading code at the low level. But in the case of generated Verilog, readability is much worse than that of SystemVerilog. This is because FIRRTL generates Verilog code at gate level, therefore it is hard to understand.

VI. Conclusion

This effort was made to realize the power of CHISEL against SystemVerilog. It is observed that CHISEL implementation is marginally better in quantitative and considerably better in the qualitative analysis as compared to SystemVerilog. The CHISEL community is not well diverse compared to SystemVerilog, so there is little support available for CHISEL on publicly accessible networks. The majority of the semiconductor vendors are still using SystemVerilog for developing chips and tools, So there is a need to convert the CHISEL to Verilog for verification purposes. This is the major bottleneck because the verification task involves more steps of conversions, but these conversions are not needed when the code is written in SystemVerilog. The CHISEL community needs to develop its own verification tools to exploit CHISEL to its fullest potential.

References

[1] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*. IEEE, 2012, pp. 1212–1221.

[2] Lampro-Mellon, "Lampro-mellon/chisel-training." [Online]. Available: https://github.com/Lampro-Mellon/Chisel-Training

[3] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[4] N. Jouppi, C. Young, N. Patil, and D. Patterson, "Motivation for and evaluation of the first tensor processing unit," *IEEE Micro*, vol. 38, no. 3, pp. 10–19, 2018.

[5] Lampro-Mellon, "Lampro-mellon/quasar." [Online]. Available: https://github.com/Lampro-Mellon/Quasar

[6] P. Lennon and R. Gahan, "A comparative study of chisel for fpga design," in *2018 29th Irish Signals and Systems Conference (ISSC)*. IEEE, 2018, pp. 1–6.

[7] J. Bruant, P.-H. Horrein, O. Muller, T. Groleat, and F. Pétrot, "(system) verilog to chisel translation for faster hardware design," in *2020 International Workshop on Rapid System Prototyping (RSP)*. IEEE, 2020, pp. 1–7.

[8] T. Parr and K. Fisher, "Ll (*) the foundation of the antlr parser generator," *ACM Sigplan Notices*, vol. 46, no. 6, pp. 425–436, 2011.

[9] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood, "A pythonic approach for rapid hardware prototyping and instrumentation," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017, pp. 1–7.

[10] J. Decaluwe, "Myhdl: a python-based hardware description language." *Linux journal*, no. 127, pp. 84–87, 2004.

[11] C. Baaij, M. Kooijman, J. Kuper, M. E. T. Gerards, and E. Molenkamp, "Tool demonstration: Clash-from haskell to hardware," in *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*. Association for Computing Machinery (ACM), 2009, pp. 3–3.

[12] A. Dobis, T. Petersen, K. J. H. Rasmussen, E. Tolotto, H. J. Damsgaard, S. T. Andersen, R. Lin, and M. Schoeberl, "Open-source verification with chisel and scala," *arXiv preprint arXiv:2102.13460*, 2021.

[13] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson *et al.*, "Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 209–216.