SYNOPSYS®
snug Users Group

SystemVerilog Microarchitecture Challenges for AI and their use for the training and screening of EE students

Yuri Panchul
Senior Staff Engineer, GPU RTL
y.panchul@samsung.com

Samsung Advanced Computing Lab
San Jose, California, USA

https://semiconductor.samsung.com/about-us/locations/us-rnd-labs/computing-lab-sarc-acl/

ABSTRACT

We present a set of open-source microarchitectural challenges in SystemVerilog. These challenges started as a way to screen candidates for job interviews, but evolved into benchmarks for EDA AI tools already used by both large companies and EDA startups. The SystemVerilog Microarchitecture Challenges for AI No.1 and No.2 are based on a floating-point unit (FPU) of an open-source RISC-V CPU called Wally. We repurposed this FPU to build the custom pipelined blocks with and without the flow control. The first Challenge stood for half a year before several AI engines cracked it, the second one stood for two months. Now we have Challenge No.3, 4 and 5 in the pipeline to keep busy both AI companies and EE graduates who are entering the job market, since such challenges proved to be useful to train all of them.

Table of Contents

Table of Figures

1. Introduction

The idea for this paper originated back in Christmas 2024 when I was approached by a number of students who asked me for an internship referral. In order to ensure referral quality, the students were given a SystemVerilog challenge similar to one of the actual tasks assigned to an intern at an electronics company: build a static pipeline that implements a formula using the wrapped floating-point unit (FPU) block from an open-source RISC-V CPU named Wally[1][2].

Each student got a different formula to implement, such as "$A^5 + 0.3 * B + C$", or a "$(A - B)^3 + C$", a set of pipelined arithmetic blocks written in synthesizable SystemVerilog for multiplication, addition, and subtraction, as well as a testbench. Such an exercise was a variation of an open-source exercise[3] from a set called the SystemVerilog Homework[4][5], aimed to help university professors and job seekers.

To my surprise, the students were trying to use various AI engines to generate their solutions - but AI was not able to come up with a design that passed the testbench checkers. The very same design challenge also happened to be a good benchmark used by two large software companies and two AI EDA startups to benchmark their tools that generate RTL code using AI.

Eventually by the Summer of 2025 several AI engines were able to solve the problem we called the SystemVerilog Microarchitecture Challenge for AI No.1[6]. So we came up with Challenge No.2[7], by adding the requirement to implement flow control to Challenge No.1. Challenge No.2 stood unsolved by AI engines for two months, but eventually one of them came up with a solution, although not a very optimal one. So now we have Challenges No. 3, 4 and 5 in the pipeline.

In this paper we discuss what AI is trying to do and how it has been getting confused during the past year. Coincidentally, the very same challenges are great for training future front-end RTL chip designers for CPUs, GPUs, and networking devices.

2. The Challenge

2.1. The problem statement

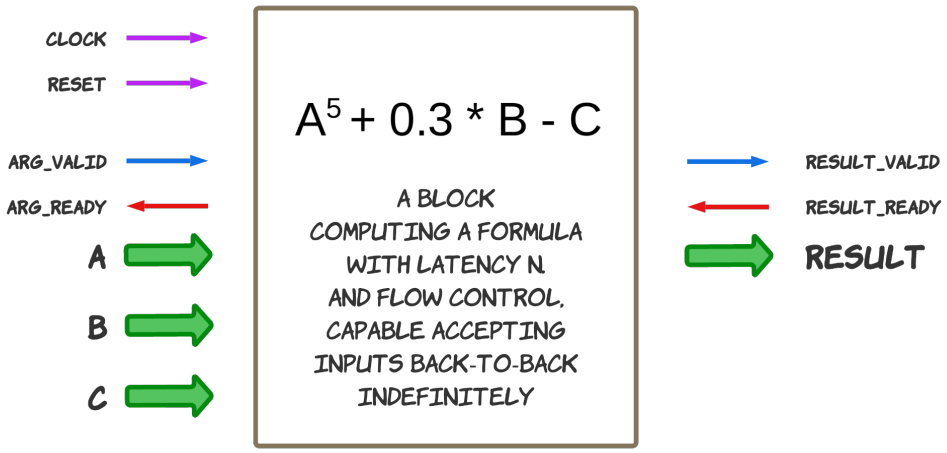

Figure 1. The formula and module interface for Challenge No.2. Challenge No.1 is similar but without the ready signals.

The Challenge No.2 GitHub repository contains the following text in its README.md file, with some edits:

“SystemVerilog Microarchitecture Challenge for AI No.2. Adding the Flow Control.

This repository contains a new challenge to any AI software that claims to generate Verilog code. The challenge is based on a very typical scenario in an electronics company: an engineer has to write a pipelined block using a library of sub-blocks written by somebody else. Then this engineer has to verify his block using a testbench written by somebody else. He may also need to figure out the sub-block latencies and handshakes by analyzing the code, since a lot of code in electronic companies is not sufficiently documented.

…

The Challenge (and AI Prompt at the same time)

1. Finish the code of a pipelined block in the file challenge.sv. The block implements the formula “A^5 + 0.3 * B - C” as shown in Figure 1.

2. Ready/valid handshakes for the arguments and the result follow the same rules as ready/valid in the AXI Stream protocol. When a block is not busy, arg_rdy should be 1, the block should not wait for arg_vld to assert arg_rdy to 1. In other words, the default value for arg_rdy should be 1, not 0. This behavior is detailed in the sections that follow.

3. You are not allowed to implement your own submodules or functions for the addition, subtraction, multiplication, division, comparison or getting the square root of floating-point numbers. For such operations you can only use the modules from the pre-defined module library in the directory arithmetic_block_wrappers.

4. You are not allowed to change any other files except the file that contains the solution to the challenge.

5. You can check the results by running the script "simulate". If the script outputs "FAIL" or does not output "PASS" from the code in the provided testbench.sv by running the provided script "simulate", your design is not working and is not an answer to the challenge.

6. When there is no backpressure from downstream, your design must be able to accept a new set of inputs (A, B and C) each clock cycle back-to-back and generate the computation results without any stalls and without requiring empty cycle gaps in the input.

7. The solution code has to be synthesizable SystemVerilog RTL.

8. Your design cannot use more than 10 arithmetic blocks from the arithmetic_block_wrappers directory or more than 10000 D-flip-flops or other state elements outside those arithmetic blocks.

9. The solution also cannot use any SRAM or other embedded memory blocks.

10. A human should not help AI by giving tips on latencies or handshakes of the submodules. The AI has to figure this out on its own by analyzing the code in the repository directories. Likewise a human should not instruct AI how to build a pipeline structure since it makes the exercise meaningless.”

An older Challenge No.1 is very similar, as shown in Figures 2 and 3. The main difference is that Challenge No.2 includes upstream and downstream ready signals to propagate backpressure, while Challenge No.1 has only valid signals.

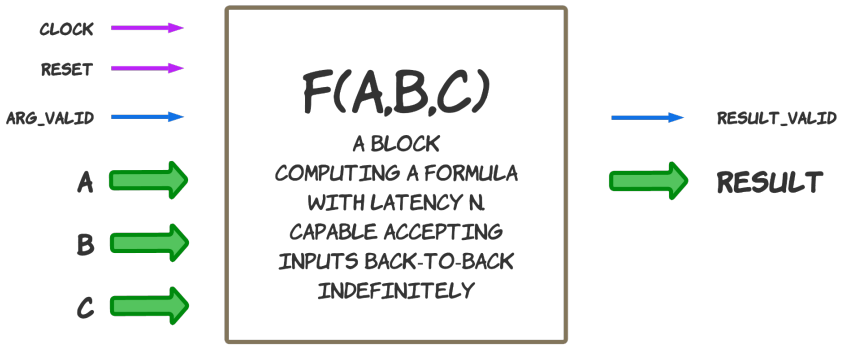

Figure 2. Module interface for Challenge No.1.

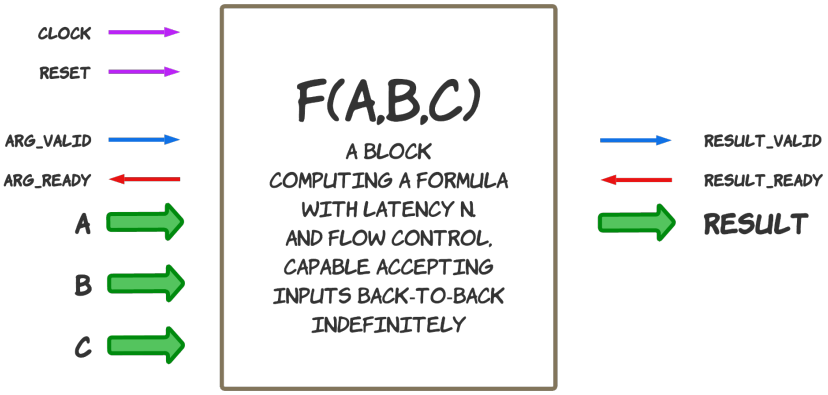

Figure 3. Module interface for Challenge No.2. Note the arg_ready and result_ready signals are not present in Challenge No.1.

There is also a minor formula difference: Challenge No.1 uses the formula "A^5 + 0.3 * B + C", while Challenge No.2 uses "A^5 + 0.3 * B - C" to make the task slightly more interesting due to different latencies of the given sub-blocks that handle the addition and subtraction.

As a reader can guess, we had to add many requirements as afterthoughts after observing various ways of “cheating”, such as both AI and the students attempted to use large data structures to just pass the tests.

2.2.The challenge module interface handshake

In both Challenges the solution has to be pipelined. Challenge 1’s solution must be able to accept a new triplet of (a, b, c) data on any positive edge of the clock when the arg_valid signal is sampled high, as shown in Figure 4. The solution should not wait for result_valid before sending another triplet. Note: the latency of 5 clock cycles shown in the picture is not the expected latency. The actual

latency is determined by the design and latencies of sub-blocks.

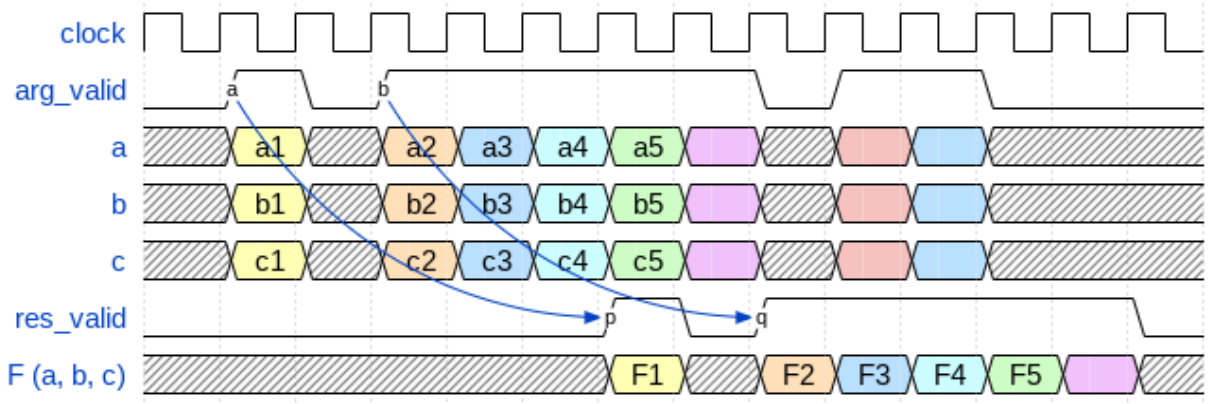

Figure 4. The expected waveform for the Challenge 1 interface.

Challenge 2 uses a Valid/Ready protocol for both the upstream (arg_valid/arg_ready) and downstream (results_valid/results_ready) interfaces, consistent with the AMBA AXI4/AXI-Stream Protocol Specifications[8][9]. We will reiterate them in the section 2.2.1 below.

2.2.1. The Valid/Ready rules used in Challenge 2

The block uses a Valid/Ready protocol for both upstream and downstream, consistent with AMBA AXI-Stream Protocol Specification rules:

1. The basic protocol uses three signals: Valid, Ready and Data. They are sampled on the positive edge of the protocol clock.
2. The transfer is considered to have happened when both Valid and Ready are sampled high.
3. Valid may turn high some clock cycles before Ready or after Ready, or they may become high in the same clock cycle.
4. Once a Valid signal is set high, it cannot be set low until the transfer occurs.
5. Similarly, data should remain stable all the time the Valid signal waits for the Ready signal.
6. The module that drives the Valid signal should not wait for the Ready signal to turn high before asserting the Valid signal high.
7. The Ready signal should be able to change at any moment without affecting the functionality of the data transmission.
8. When both Valid and Ready signals are kept high for multiple cycles, the data transfers in the stream go back-to-back; it is unnecessary to de-assert either Valid or Ready.

You can see the different cases of Valid-Ready handshaking in Figure 5.

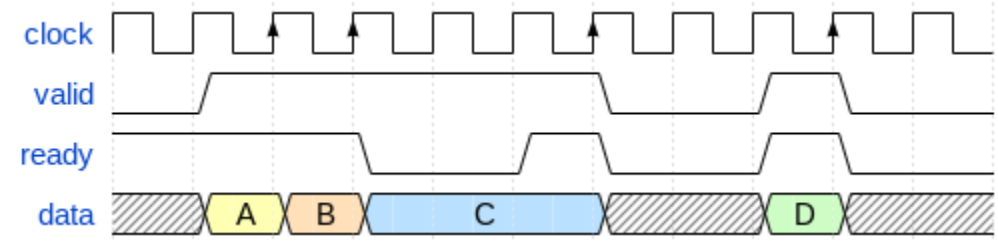

Figure 5. The different cases of Valid-Ready handshaking: Ready before Valid, transfers back-to-back, Valid before Ready, and Valid in the same cycle as Ready.

2.3.The sub-blocks and the verification environment

The most important feature of the Challenges is that the solution has to be “sandwiched” between the pre-existing testbench and the pre-existing sub-blocks. This methodology is described in “How to Fail Those Students Who Rely on ChatGPT”[10]. The testbench plays various scenarios, with back-to-back transactions and backpressure. The pre-existing sub-blocks are something a student (or an AI) has to analyze to understand their interface and extract their latencies from their code. The student is not allowed to change either the testbench or the sub-blocks, as shown in Figure 6. Most functionally incorrect AI-generated designs failed the test, as shown in Figure 7. One student sent a message “Look, I went above and beyond the task: I also implemented the submodules and the testbench”. Sorry, no, folks, as shown in Figure 8.

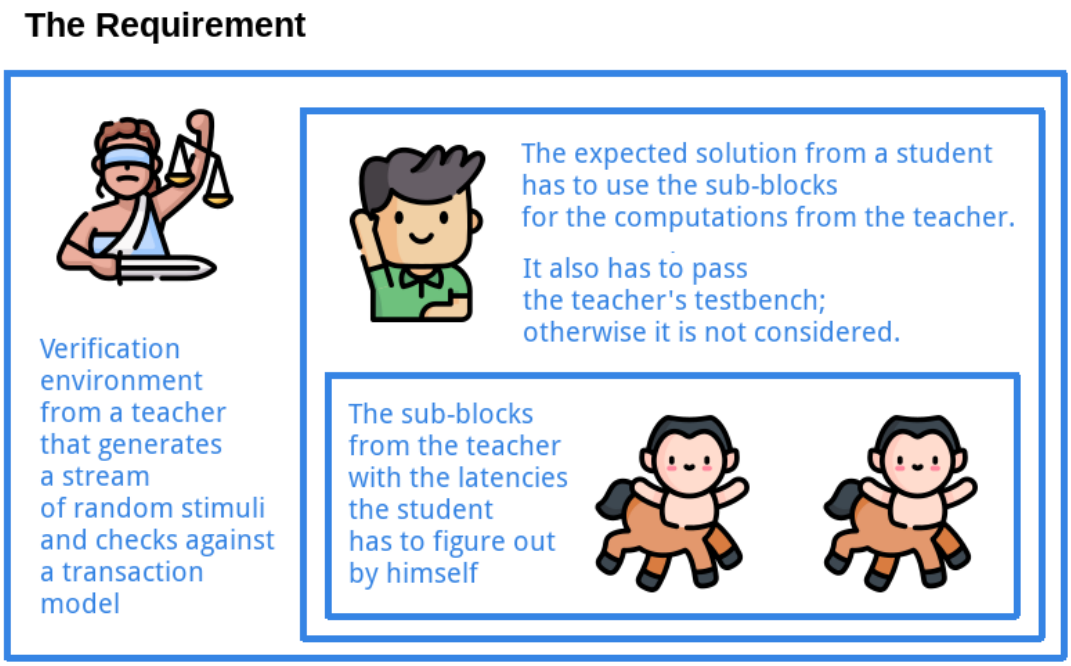

Figure 6. The Challenge solution has to be “sandwiched” between the pre-existing testbench and the pre-existing sub-blocks.

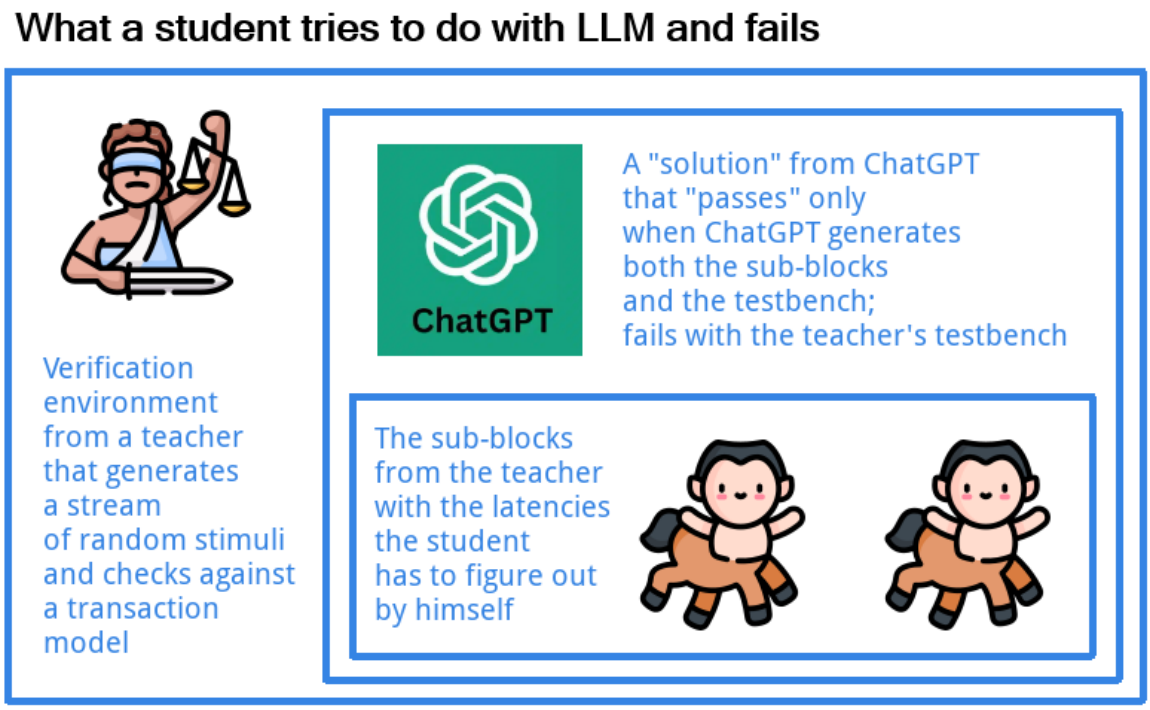

Figure 7. The infrastructure detects in most cases when a solution is functionally incorrect. Still, manual review is required.

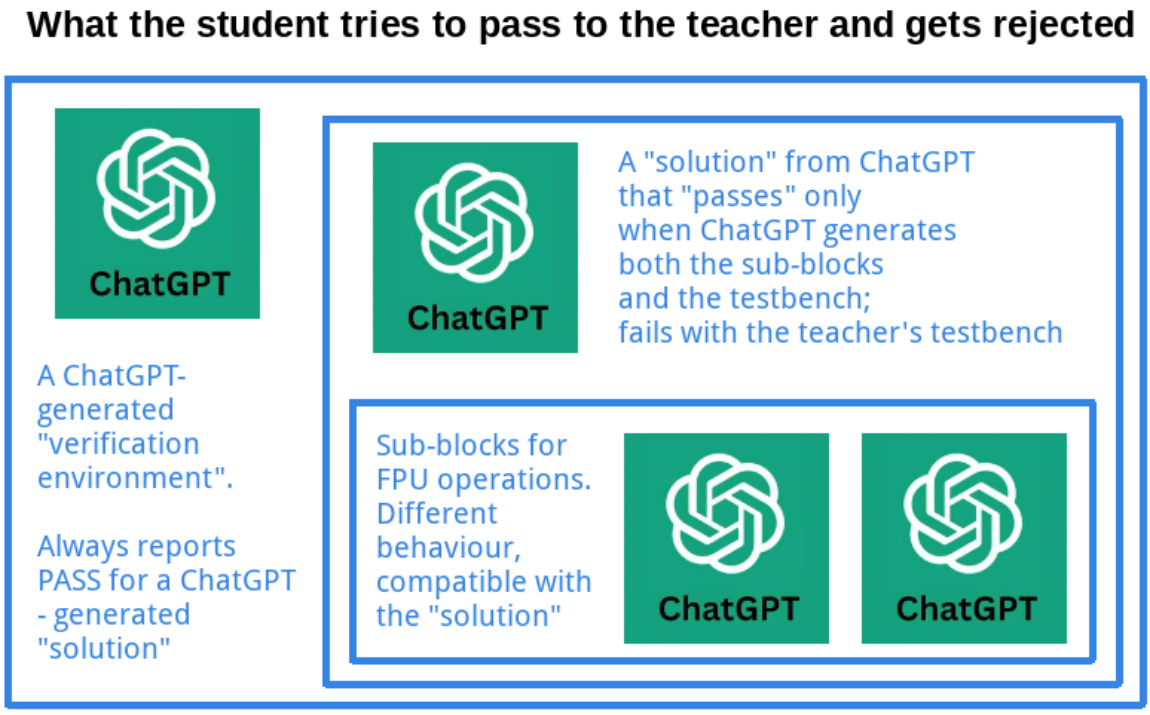

Figure 8. The setting where AI "verifies" itself for the Challenge is not very useful.

3. The solution to Challenge No.1, without flow control

A more or less optimal solution for Challenge No.1 shown in Figure 9:

1. We compute A^5 using three multiplication blocks:
    a. The first one multiplies A by A and gets A^2.
    b. The second one multiplies the result of the first one by itself and gets $A^2 * A^2 = A^4$.
    c. Finally, the third one multiplies the result of the second one by the original A and gets $A^4 * A = A^5$.
2. Since the latency of each multiplication is 3 clock cycles, the third multiplication starts 6 clock cycles after the input. It means we have to delay the original A argument by 6 clock cycles. We can do it in three ways: using a 64-bit-wide shift register with a depth of 6, or using a FIFO, or using a single-pointer ring buffer. We will discuss these options in Appendix A.
3. In parallel to computing A^5 we compute (0.3 * B + C). We start by computing (0.3 * B) using another multiplier. At the same time, we store C in another sideband ring buffer (or FIFO or shift register). Once we finish multiplication, we get C from the buffer and add it to (0.3 * B) using the adder module.
4. Since the latency of the pipelined adder module is 4 clock cycles, the value of (0.3 * B + C) is computed in 3 + 4 = 7 clock cycles. Now we have to align this pipeline branch with the pipeline branch we created to compute A^5, which has a latency of 9 clock cycles. To do this alignment, we create another 2-deep ring buffer.
5. Finally, we add the results of two pipeline branches using another adder with 4 clock cycle latency. The total latency of our pipeline is 13 clock cycles; it uses four multipliers, two adders and (6 + 3 + 2) * 64 = 11 * 64 = 704 bits in three D-flip-flop-based ring buffers.

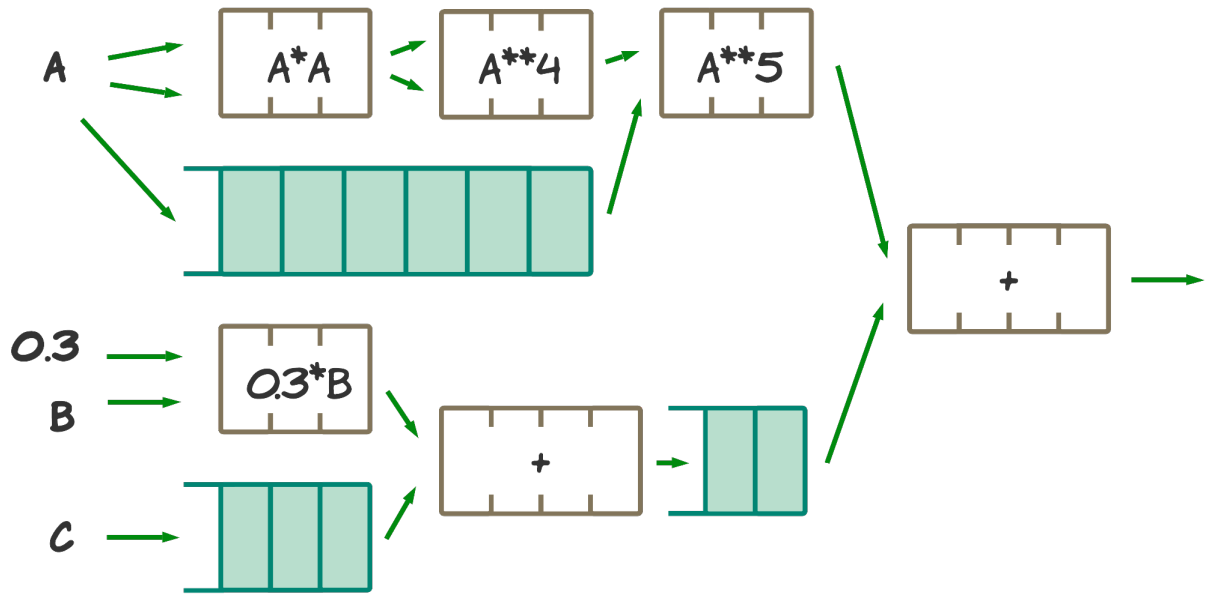

Figure 9. The solution to the SystemVerilog Microarchitecture Challenge for AI No.1.

4. Some typical mistakes with Challenge No.1

4.1. Early days: behavioral code and FSM instead of pipeline

When LLMs were released to the public at the end of 2022, they initially generated C-like behavioral code instead of RTL Verilog. Then they started to generate FSMs without resets and textbook multiplexer examples. Based on my experience, LLMs did not handle pipelining, except maybe a classic 5-stage CPU pipeline, until 2025.

Before that, when given a pipeline problem one LLM generated a finite state machine-based design that processed each transaction one at a time for multiple clock cycles. When I replied “this is not a pipelined design,” LLM’s reply was “sorry, here is a pipelined design” - and then it generated an FSM-based design again.

4.2. Next pitfalls: ignoring clock cycles and arithmetic properties

In April 2025 at least one challenge participant gave me a design with the arithmetic blocks connected as if they were blocks of combinational logic, without even attempting to delay the arguments to align different pipeline branches. This design obviously could not pass the testbench because it confused data from different clock cycles. It is illustrated in Figure 10, together with two other problems:

1. Computing A^5 not via $(A^2)^2*A$, but using A*A*A*A*A, adding an extra 3 clock cycles and the area of a multiplier block.
2. Ignoring the associativity of addition: a designer can save latency and ring buffer flops by computing "a ** 5 + (0.3 * b + c)" instead of "(a ** 5 + 0.3 * b) + c".

Another issue worth mentioning: back in early 2025 some LLM engines confused the IEEE 754 floating point format with plain integers, as the 64-bit variant of IEEE float with the 32-bit variant (the challenge uses 64-bit).

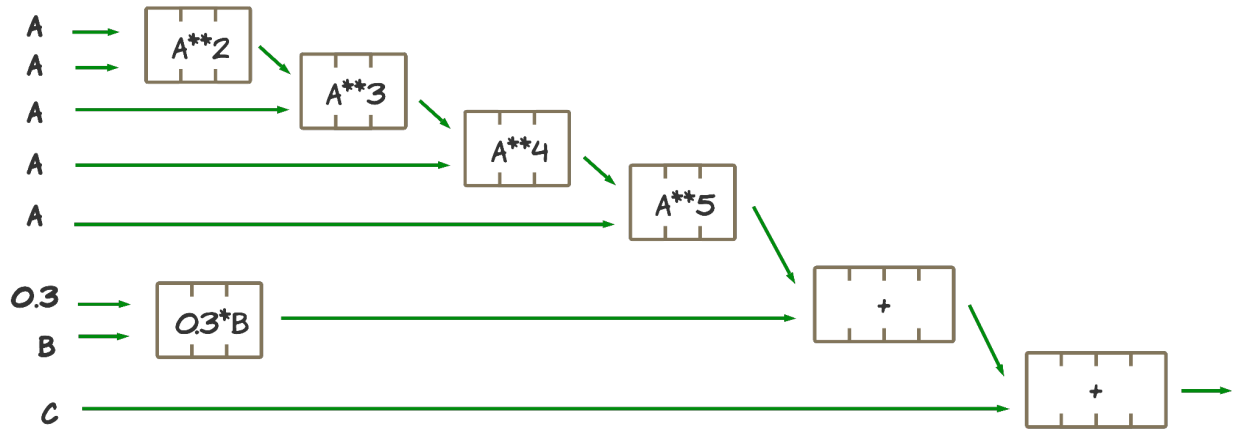

Figure 10. An incorrect design that ignores the clock latencies of arithmetic blocks and cannot pass the testbench. It also computes A5 inefficiently and ignores the associativity of addition.

4.3. Failure to analyze the sub-block code for the latencies

Early in 2025 LLMs did not attempt to find out the latencies of the given arithmetic modules. One engine wrote the explanation: “For the illustration, assume latency equal 1”. Then AI generated an implementation with a single staging register per arithmetic block, as shown in Figure 11.

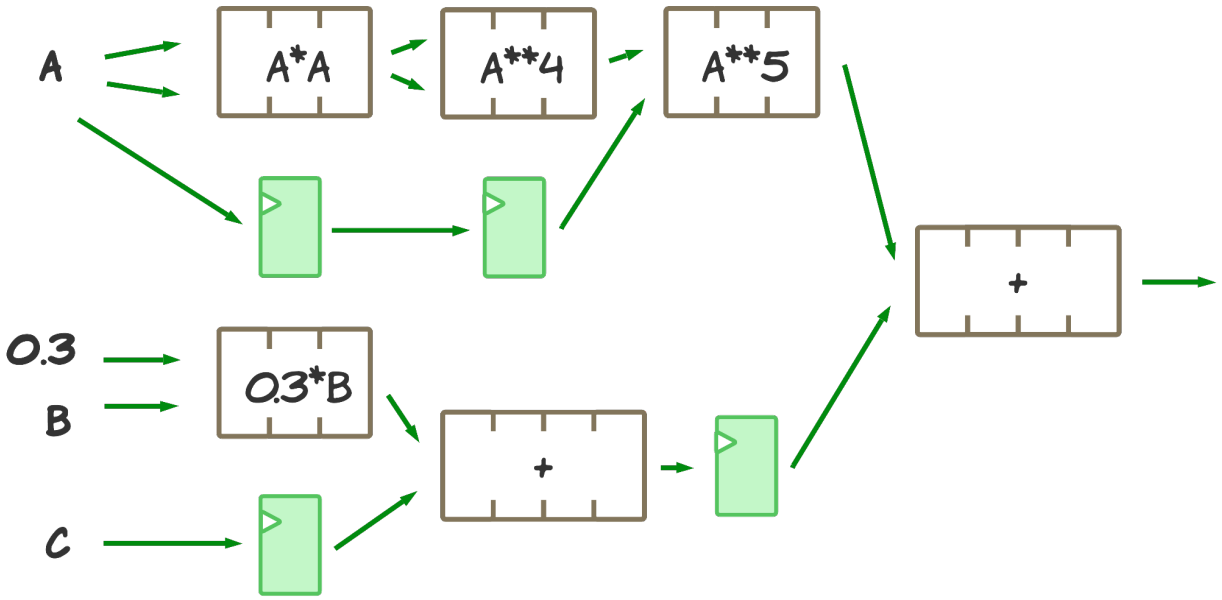

Figure 11. An incorrect implementation that assumes an arithmetic block latency equals 1, uses a single staging register per block.

Some students argued that the latencies have to be defined in the Challenge. We disagree since in real life the internal specs in many electronic companies do not specify all latencies, so engineers have to be able to figure out such latencies from analyzing the code and looking at simulation waveforms.

We also saw a design that uses grossly oversized FIFOs, like 128-deep. It passed our testbench but was not practical. We found both variations in AI-generated code.

4.4. Using arithmetic blocks to align the pipelines: functionally correct but grossly inefficient design

Another AI idea is to align the pipeline branches using arithmetic blocks that add zero or multiply by one. I also heard this idea from software students unfamiliar with hardware FIFOs, staging registers and equivalent structures. With this approach, the formula “$A^5 + 0.3 * B + C$” becomes “$(((A*A)^2)^2*A + (0.3*B*1*1))+(C*1*1*1+0)$”, as shown in Figure 12. This design passes the testbench but wastes a lot of area and power. For this reason we added a requirement to the Challenge limiting the number of arithmetic blocks.

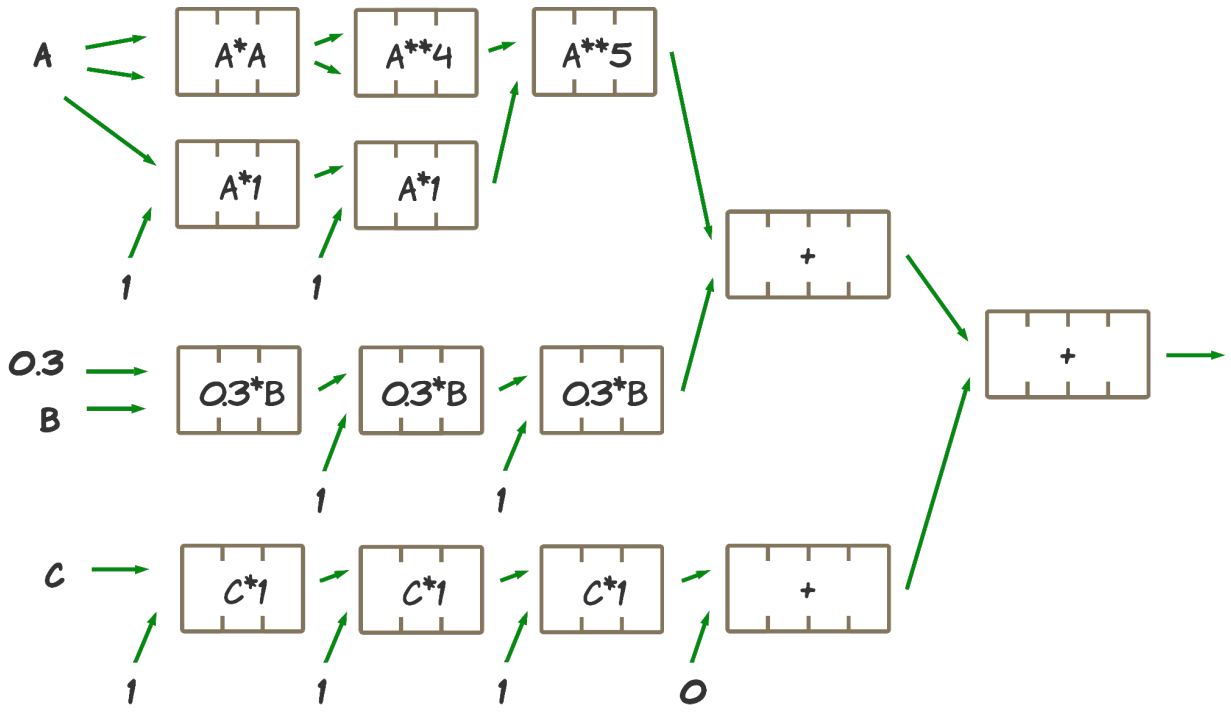

Figure 12. Functionally correct but grossly inefficient design that aligns the pipeline branches using arithmetic blocks that add zero or multiply by one.

5. The solution to Challenge No.2, with flow control

The most optimal way to convert a static pipeline (Challenge 1) with valid-only into a block with flow control (Challenge 2) is to use a combination of the solution to Challenge 1 with a FIFO and a credit counter:

1. We send the output of the Challenge 1 pipeline to the FIFO. Sometimes microarchitects use the wording “sink to the FIFO”.
2. At reset we initialize the credit counter with the depth of the FIFO.
3. We decrement the counter when the upstream data (the arguments) enter the pipeline. It happens when arg_valid=1 and arg_ready=1.
4. We increment the counter when the downstream (the block that receives the results) pops the data from the FIFO. It happens when FIFO is not empty and the downstream ready is 1.
5. When the credit counter is 0, we stall the upstream by setting arg_ready=0.

This microarchitecture is shown in Figure 13.

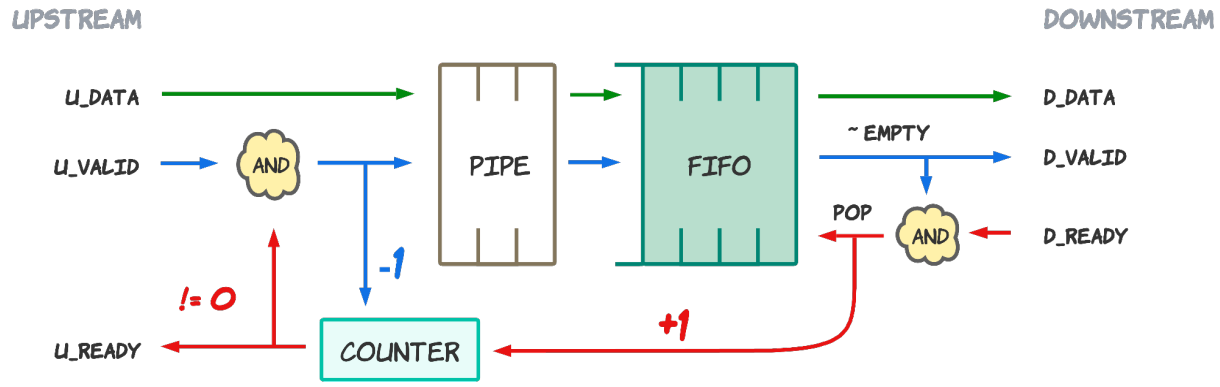

Figure 13. The basic example of a credit-based flow control.

This mechanism guarantees that we will never run into a situation where there are some transactions traveling through the pipeline and the downstream cannot accept them. It is easy to see that the number of transactions in progress through the pipe (microarchitects say “the transactions in flight”) plus the number of transactions stored in the output FIFO is less or equal to the depth of FIFO. So the value of the credit counter represents the number of transactions the pipeline can accept.

Note that the block will work with any depth of the FIFO, however in order to allow back-to-back bandwidth we have to set the FIFO depth at least to the pipeline depth plus one. In this case, in the absence of backpressure while having a back-to-back stream of data, the pipeline will be fully occupied.

We can do a minor optimization shown in Figure 14, where we set arg_ready=1 in one of two conditions: either counter>0, or counter=0 and we are doing a pop from FIFO in the same clock cycle. We pop from FIFO when it is not empty and result_ready=1. This optimization uses the fact that if the FIFO has anything in it right now, its occupancy will be decremented as a result of pop. In this case we can reduce the FIFO depth to the depth of the pipeline and still get back-to-back bandwidth. However this optimization has a combinational path between the result_ready and arg_ready which is not desirable in many designs since it can lead to timing problems.

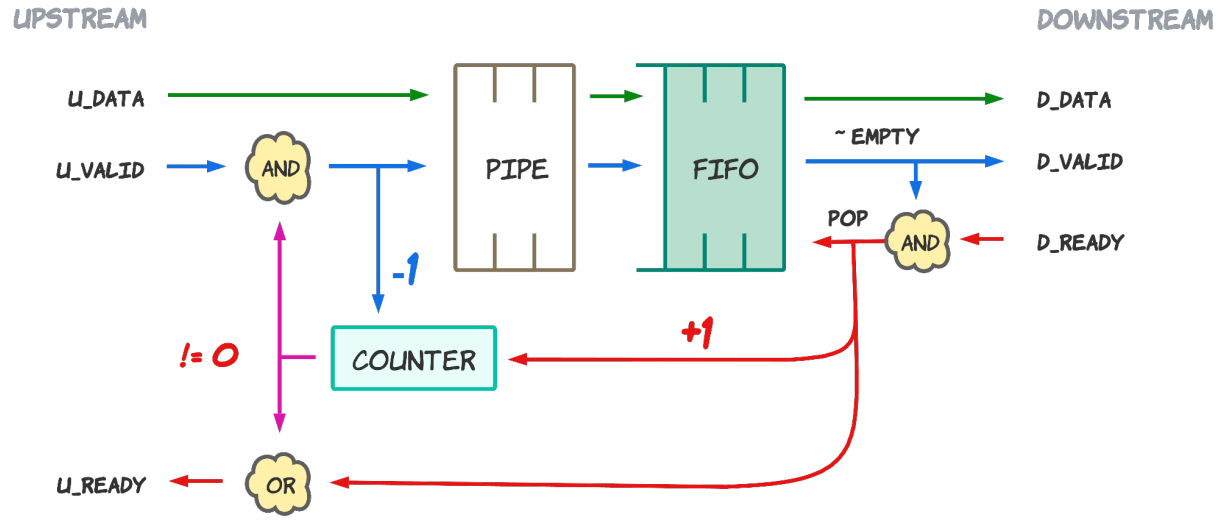

Figure 14. A minor optimization of the reference solution. Smaller FIFO, but combinational path, may cause

timing problems.

6. Some typical mistakes with Challenge No.2

6.1. Not using the output FIFO

While the technique described in Chapter 5 has been commonplace in the electronics industry for at least the past 25 years, many graduate students are not aware of it. Such flow control is taught in networking courses but not in digital design classes (though there are exceptions). If we look at a popular textbook by Dally and Harting[12], published in 2012, it discusses handling backpressure in static pipelines by putting the logic, such as double buffers, inside the pipeline itself. So the first reaction of some students to Challenge 2 was: “can we change the arithmetic sub-blocks to handle the backpressure?”

A more recent textbook by David J. Greaves[13] does discuss credit-based flow control, however it does not provide any Verilog examples.

6.2. Using the FIFO full signal to stop sending transactions to the pipeline

The most basic mistake an AI and students can make is to connect the output FIFO’s full signal to arg_ready, like “assign arg_ready = ~ full”. This design is obviously wrong because if the downstream backpressure is continues for a number of clock cycles, and if the pipeline has any transactions in flight, they will sink downstream and overflow the FIFO.

6.3. Adding a giant FIFO just to pass the testbench

Some students and some AI engines were adding a giant output FIFO to consume all the transactions that sunk from the pipeline during the backpressure test in the testbench. So we had to make the backpressure in the testbench longer and put the following restrictions in the Challenge:

“Your design cannot use more than 10000 D-flip-flops or other state elements outside those arithmetic blocks. The solution also cannot use any SRAM or other embedded memory blocks.”

6.4. Using the obsolete “almost full” technique instead of credit-based flow control

There are texts on the internet that propose using a technique called “almost full”. AI engines were probably trained in those texts as well. In this approach you measure the FIFO occupancy and set arg_ready=0 when the occupancy hits “FIFO depth - pipeline depth”.

At first glance this technique looks similar to the credit counter approach, but there is a big difference:

- The “Almost full” approach stops sending transactions to the pipeline even if the pipeline is empty (there are no transactions in flight). In this case, the space in the FIFO reserved for the worst-case scenario (the pipeline is full) is not used until the downstream stops applying backpressure.
- The credit counter approach will continue to send transactions to the pipeline because we keep track of the total number of transactions in flight and sitting in the FIFO, and this number is still below the FIFO depth in the described scenario.

6.5. Stopping sending transactions to the pipeline with any backpressure

This is similar to the “almost full approach” approach: we waste clock cycles and FIFO capacity when the number of in-flight transactions is less than the space left in the FIFO.

7. The next frontier: Challenge No.3. Building a high-bandwidth block from multiple low-bandwidth

Since Challenges 1 and 2 were solved by AI, I asked a UC Santa Barbara student Alex Huang to create the next Challenge 3, based on something AI engines did not do at the moment of writing - building a block with the above formula when we introduce a division operation. The problem with division in the FPU of Wally CPU: it is not pipelined, unlike addition and multiplication. If we want to still handle back-to-back bandwidth, we have to instantiate an array of division modules and distribute the workload among them.

Alex took Challenge 2 and simply replaced the multiplication operation with division, as shown in Figure 15.

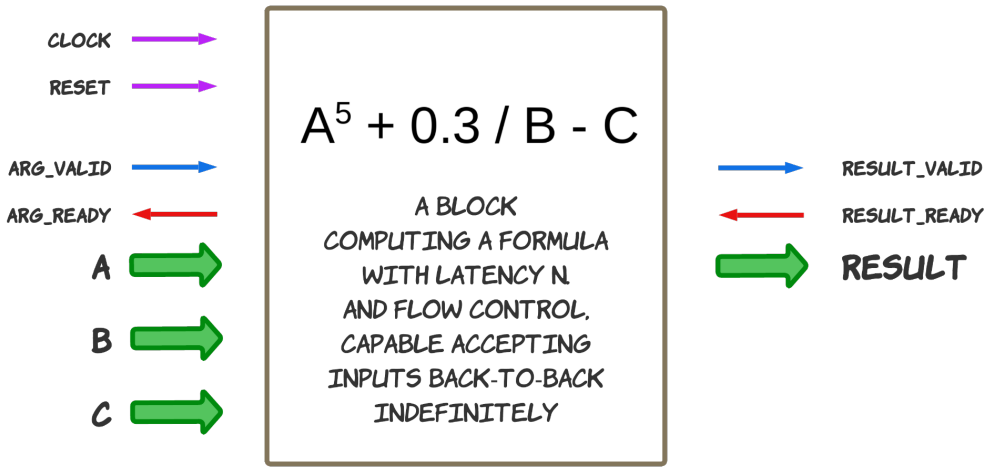

Figure 15. Challenge 3 has the same interface as Challenge 2, but the formula includes division instead of multiplication.

The missing part to implement a solution was a “quasi-pipelined” divider module, capable of accepting inputs back-to-back indefinitely in the absence of backpressure. The module f_div, derived from the Wally FPU, can accept the next pair of arguments only after it has generated the result for the previous pair. In addition, the f_div latency is not fixed, which presents an additional challenge.

Figures 16 and 17 illustrate the functionality of the divider we have in the given library of modules and the divider we need to solve Challenge 3.

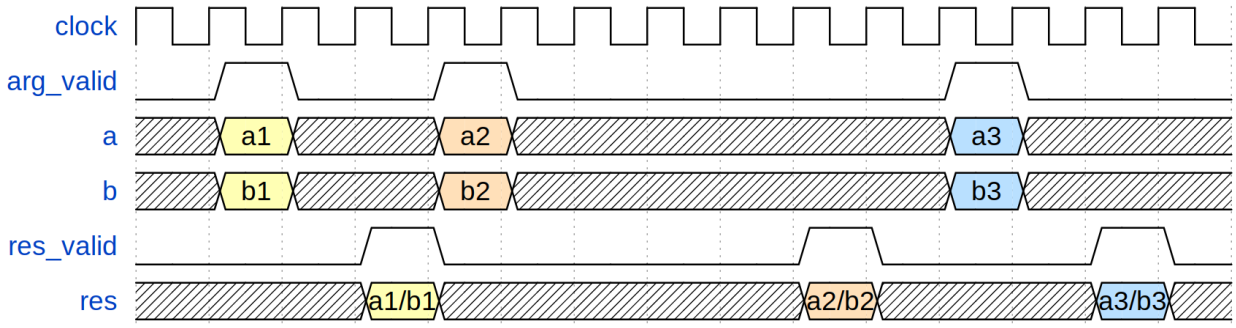

Figure 16. The cycle behavior of a non-pipelined variable latency divider present in the given library, derived

from the FPU of the Wally CPU.

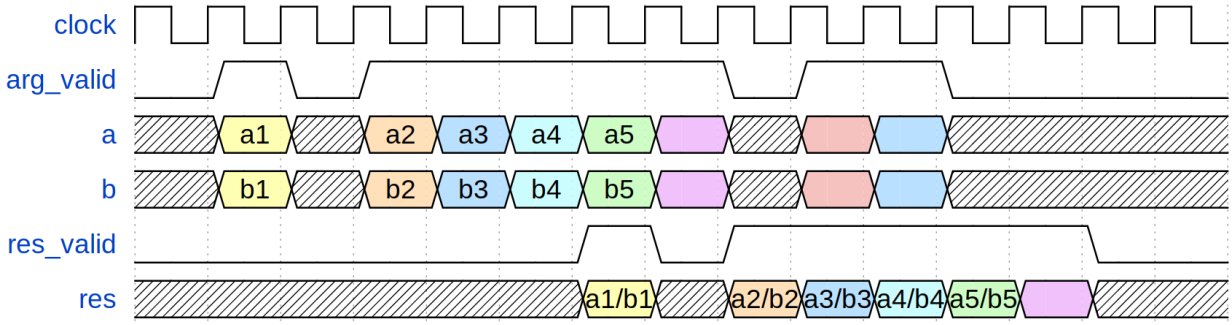

Figure 17. The cycle behavior of a fixed latency "quasi-pipelined" divider module, capable of accepting inputs back-to-back indefinitely. Such a module is needed to solve Challenge 3.

The solution is to run multiple dividers in parallel, start a new division every clock cycle and give the arguments each time to a different divider. This approach requires instantiating the number of dividers equal to the maximum divider latency, as shown in Figure 18.

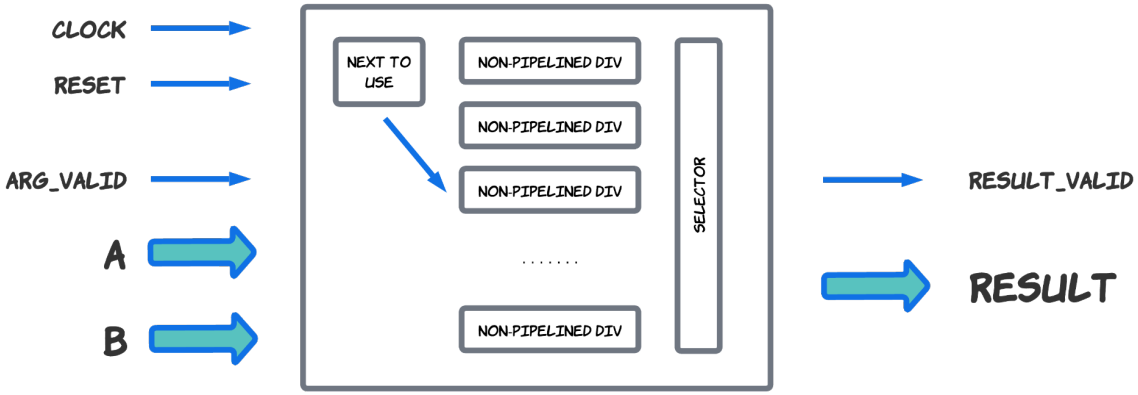

Figure 18. Building a "quasi-pipelined" divider by instantiating multiple non-pipelined dividers and distributing workload between them.

An important intermediate step in the solution: we also have to convert a divider with variable latency to a divider with fixed latency, as illustrated in Figure 19.

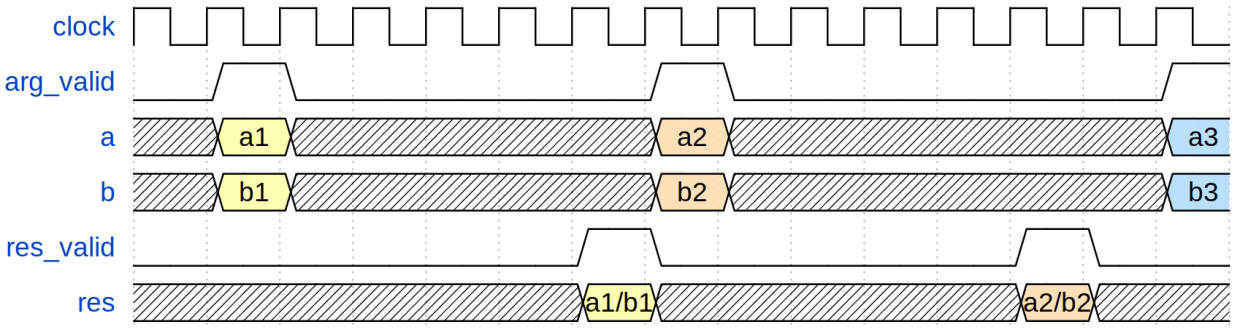

Figure 19. The cycle behaviour of a non-pipelined divider with fixed latency equal to the maximum latency of

the non-pipelined divider with variable latency, as shown in Figure 16. This is an intermediate step to build a "quasi-pipelined" divider in Figure 17.

8. Conclusions

The participants in Challenges 1 and 2 tried a dozen AI engines. Our experience is consistent with the idea that LLMs are glorified search engines for already solved and published problems. A more interesting topic is how to train university graduates to solve the microarchitectural problems essential for their success in the workplace. We believe a set of our Challenges is a step in the right direction.

9. Credits

Contributors to the SystemVerilog Homework that became the basis for the Challenges: Yuri Panchul, Mike Kuskov, Maxim Kudinov, Kiran Jayarama, Maxim Trofimov, Alexey Fedorov, Konstantin Blokhin, Petr Dynin. Creator of Challenge 3, Alex Huang.

Credits for the icons: Freepik – Flaticon: Centaur[14], Themis[15] and Student[16].

10.Appendix A. A note about the sideband buffers

As we mentioned, we can align the arguments in the pipeline in different ways: using a 64-bit-wide shift register with a depth of the pipeline, using a FIFO, or using a single-pointer ring buffer.

The main difference is power efficiency. When we use a shift register, we move data. Moving data causes a lot of flip-flop switching, which consumes dynamic power. However when we use a FIFO or a single-pointer ring buffer, we move pointers. This makes FIFO and ring buffer more power-efficient. This is true even if we optimize a shift register by moving only data with the associated valid bit set, as shown in Figure 15.

A ring buffer is preferred as a sideband buffer for a fixed-latency pipeline, but we have to use a FIFO when the pipeline latency varies, as shown in Figure 16.

The topic of such buffers is outside the scope of this article, but you can find examples of all data structures in the Verilog Meetup Basics-Graphics-Music GitHub repository[11].

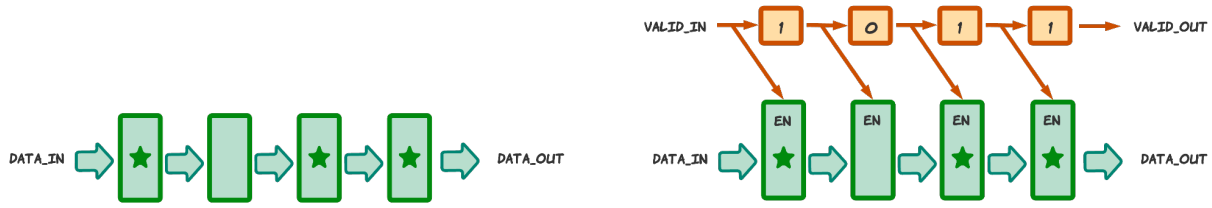

Figure 20. A plane shift register versus a shift register optimized for power. The latter uses valid bits that drive the data enable.

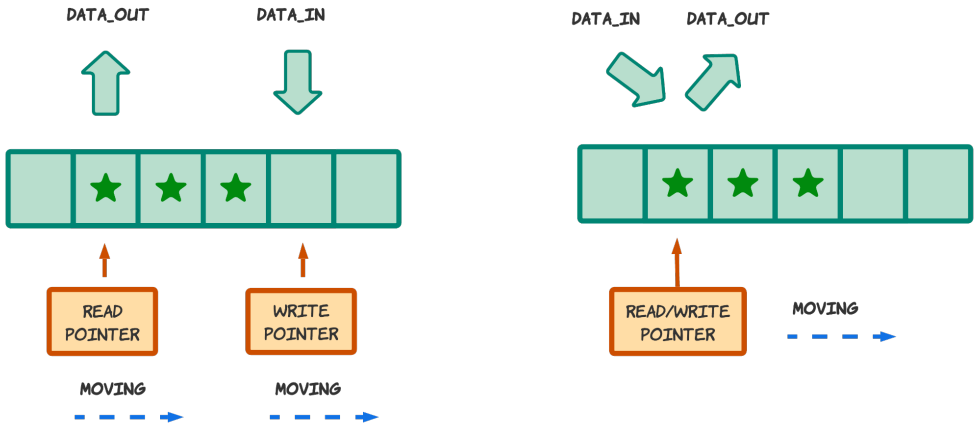

Figure 21. A FIFO versus a ring buffer with a single pointer.

11.References

1. CORE-V Wally is a configurable RISC-V Processor associated with the RISC-V System-on-Chip Design textbook. https://github.com/openhwgroup/cvw
2. David Harris, James Stine Ph.D., Sarah Harris, Rose Thompson. RISC-V System-on-Chip Design. Morgan Kaufmann. 2026. https://www.amazon.com/RISC-V-Microprocessor-System-Chip-Design/dp/0323994989
3. https://github.com/verilog-meetup/systemverilog-homework/tree/main/05_finite_state_machines/05_07_float_discriminant
4. A collection of SystemVerilog exercises from the beginning to the microarchitectural job interview level. https://github.com/verilog-meetup/systemverilog-homework
5. A new edition of SystemVerilog-Homework adds exercises that use the FPU of an open-source CPU. https://verilog-meetup.com/2025/02/11/a-new-edition-of-systemverilog-homework-adds-exercises-that-use-fpu-of-an-open-source-cpu/
6. SystemVerilog Microarchitecture Challenge for AI No.1. https://github.com/verilog-meetup/systemverilog-microarchitecture-challenge-for-ai-1
7. SystemVerilog Microarchitecture Challenge for AI No.2. Adding the Flow Control. https://github.com/verilog-meetup/systemverilog-microarchitecture-challenge-for-ai-2
8. AMBA® AXI™ and ACE™ Protocol Specification. https://documentation-service.arm.com/static/5f915b62f86e16515cdc3b1c
9. AMBA AXI-Stream Protocol Specification. https://documentation-service.arm.com/static/64819f1516f0f201aa6b963c
10. How to Fail Those Students Who Rely on ChatGPT. https://verilog-meetup.com/2025/04/29/how-to-fail-those-students-who-rely-on-chatgpt/
11. FPGA exercise for beginners. https://github.com/verilog-meetup/basics-graphics-music
12. William James Dally and R. Curtis Harting. Digital Design: A Systems Approach. 2012.
13. David J. Greaves. Modern System-on-Chip Design. https://www.arm.com/resources/education/books/modern-soc
14. Credits for the icons: Freepik – Flaticon: Centaur. https://www.flaticon.com/free-icons/centaur
15. Credits for the icons: Freepik – Flaticon: Themis. https://www.flaticon.com/free-icons/themis
16. Credits for the icons: Freepik – Flaticon: Student. https://www.flaticon.com/free-icons/student