Latency Counting in the SUS Language

Lennart Van Hirtum
PC2 at Universität Paderborn
Germany

Christian Plessl
PC2 at Universität Paderborn
Germany

ABSTRACT

This paper argues for the pipelining method employed in the SUS Language called Latency Counting. It compares it against the alternatives, and explains the considerations that underpin its design.

1 COMPARISON OF PIPELINING STYLES

Pipelining is a contentious topic in hardware design. Many tools have their own opinion on how (and if) it should be done. Languages range from providing no easy method of pipelining, thereby requiring the programmer to instantiate every register themselves, through providing 'pipeline' abstractions with explicit stages, to completely taking pipelining out of the programmer's hands.

The view that pipelining should be fully left up to the compiler is an attractive one. After all, the compiler knows (at least in theory) how it lays out the hardware on the chip, and therefore knows where the biggest bottlenecks are. Especially in the face of the shifting bottleneck towards wire instead of gate delays. In this philosophy pipelining is a low-level implementation detail that the programmer should no longer worry about, much like how register allocation for software design is completely invisible to programmers today.

However, adding or removing latency registers does impact the many trade-offs a hardware designer has to make. For this reason the programmer should still retain control over the pipelining process themselves. They may want to add fewer pipeline registers in certain areas of the design, trading a more difficult Place & Route for fewer resource use. In another location, they may add many extra, as this location is latency insensitive and should leave P&R flexibility for other parts of the design.

Ideally the language should be structured in a way that adding or removing pipelining has a minimal impact on the codebase. A popular approach to pipelining is introducing explicit pipeline stages. Languages such as TL-Verilog[2] and Spade[5] take this approach. However, we disagree with this approach. While adding or removing pipeline stages this way is rather easy, it comes at the cost of requiring the programmer to structure their code around the pipeline stage structure. With explicit pipeline stages, the programmer has to group all hardware that happens to fall within the same pipeline stage together, even though these functionally independent branches of a pipeline might have nothing to do with each other. Of course, this is partly ameliorated by factoring out the pipeline branches into their own modules or entities, but this then in turn reveals another issue. If you want to use a function-call-like syntax for using a sub-pipeline, then contrary to the way the code is written lexically, outputs don't actually become available until a number of pipeline stages later. Furthermore, allowing data from different pipeline stages to interact is usually a hassle, requiring the programmer to explicitly break out of the pipeline system. A final issue is that this model requires all imputs to be received at the same time, and all outputs are produced at the same time. This may cause opportunities for more closely interlocking submodules across pipeline stages to be lost.

SUS tackles this in a different way. Instead of focussing on 'pipeline stages' as one might want to do for CPU design, it focusses on supporting dataflow designs. SUS allows the programmer to add *latency* at arbitrary points in the hardware design, which automatically get compensated for by adding extra latency in the other branches of the pipeline as well.

Semantically, SUS divides the registers the programmer can use into *state* and *latency* registers. An example of this is shown in Listing 1. State registers are used to transfer data across cycles, thereby playing an integral part in the logic. Latency registers are used purely for meeting timing, and do not affect the design's logical function. This division closely matches the programmer's intuition about register use. In practice Latency Counting works by inferring an *absolute latency* (relative to some arbitrary origin) for every signal in the design, and using the differences between these absolute latencies to derive how many registers should be placed between signals. This approach is somewhat similar to Filament's[3], though in that work no distinction is made between state and latency registers to enable proving the safety of more complicated constructs. As a consequence, this causes significant syntactical overhead which SUS seeks to avoid. The language DFiant[4] also approaches the problem in this way.

SUS has a few goals with its Latency Counting system:

- Adding a latency register somewhere should be as simple as adding a single `reg` keyword before an assignment.
- The exact implementation of a module, including positions of latency registers should only depend on the module's code.
- Absolute latencies should be inferred as much as possible. They should be an afterthought that only comes to the forefront when relevant.
- Latency registers cannot be inserted where their presence would impact the functioning of the design.

```
module blur: int a -> int result {
  state int prev;
  // Add some pipeline stages to the output
  reg reg reg result = (prev + a) / 2;
  prev = a;
}
```

Listing 1: Example SUS code using both a state register `prev` to carry over data across clock cycles, as well as several pipeline registers on the result.

LATTE '24, April 28, 2024, San Diego, CA, USA

2 CONSTRAINING THE PROBLEM

To be able to make the latency counting system deterministic, regardless of at which node the algorithm starts, we must constrain the solution space until only a single solution is left.

The fundamental constraint we start from is the user provided delta latencies. When the user specifies a `reg` keyword on a statement (say `reg x = a;`), it tells the compiler that there must be at least a difference of 1 between the latency of `a` and `x`. Mathematically: $|x| - |a| \geq 1$. with $|x|$ the absolute latency of wire x.

This first constraint alone still leaves an infinite space for adding arbitrarily many registers into the design, so we need an upper bound as well. A vague upper bound that does this is "inputs should be accepted as late as possible, and outputs should be produced as early as possible". This gets us most of the way there, but suffers from indeterminable cases where an input and output push against each other as explained in philosophy/latency.md[1].

To that end, we devised a more rigorous definition for the above constraint: "The latency between any input i and output o that have a dependency is *minimal*". In case this is not possible, the user must specify the affected port explicitly.

This constraint firmly defines absolute latencies for all inputs and outputs, but there is still some freedom left for the internal wires, as can be seen in Listing 2. We leave inference of these latencies up to the implementation, though in general the implementation will place them as early as possible, as this corresponds to programmers' intuition.

```
module blur : int a -> int r {
    reg int t = a * a;
    // Can be either at latency 0 or 1
    int t2 = a + a;
    r = t + t2;
}
```

Listing 2: Code with degree of freedom

2.1 Implementation

To start latency counting, first a directed graph is created from the code, where wires and temporaries become nodes, and the dependencies between them connections. The connections get weighted based on the number of `reg`s in this assignment. Any submodules used also create connections between their inputs and outputs, based on the latencies inferred for the submodule. Perhaps counterintuitively, `state` registers introduce no latency. The value of all nodes are their absolute latencies, which are initially unset.

The core subroutine of latency counting is a depth-first forward counting traversal through this graph. It starts from a given start node, which is assigned a given absolute latency, and for each connected child, it adds the edge latency to the current node value. If the existing latency in the target node is lower than the new value, it gets assigned the new value and is recursed over. If the traversal bumps into a node already in its path, it checks that the new value is less than or equal to the existing value. If not it returns an error that a net-positive latency cycle has been found. If it returns successfully, then all nodes touched by this algorithm are at minimal delta latency to the starting node.

The actual algorithm then uses this subroutine. It starts from an arbitrary input or output port, seeds it with an arbitrary value 0, and alternatingly runs the depth-first traversal backwards and forwards to discover the absolute latency of other ports. Importantly, the absolute latencies across the whole graph are cleared after every exploration, this ensures that if two explorations disagree over the absolute latency of a port, we can detect it, and return an error requesting the user to explicitly specify the port latency explicitly.

After port discovery finishes, a single forward pass is made starting from all input nodes to find all internal absolute latencies.

Of course, not every wire is dependent on the inputs, for example constants or counters. After this, we can do another backwards pass, pinning every latency we've found thus far, and then forcing the new signals to be as late as possible.

This approach breaks down a little when trying to build unconventional hardware, where the inputs and outputs are disjoint, or where some nodes can't be reached even after this forward and backwards pass. Further work should be done to allow for latency inference in these more complicated cases, though keeping everything deterministic remains difficult.

The user may decide to explicitly specify some of the absolute latencies in the design. Theoretically, we could use the exact same algorithm, but we first modify the graph by adding an edge in each direction between every pair of specified nodes `a` and `b`, of delta $|b| - |a|$ and $-(|b| - |a|)$ respectively. This in effect pins the relative offsets between these nodes together. In practice though, we explicitly take the specified latencies into account for better error reporting.

2.2 Breaking out

With the latency counting system it is easy to create pipelines, but sometimes we need to break out of this mode of thinking. If, for example, we wish to make the read side of a FIFO, how can we make `ready` dependent on `data_valid` if the latter is later by say, 7 cycles? The SUS Language provides two escape hatches: *Latency Offsets* and *Latency Cuts*.

Latency Offsets allows us to add or remove a fixed amount of latency along a connection, without actually adding the corresponding registers. It is provided in the standard library as `rebase_latency<gen int DELTA, T> : T i'0, T o'DELTA`.

Latency Cuts occur at interface boundaries in a multi-interface module. Each interface in a SUS module has its own latency- (and possibly clock-) domain. Signals carried from one interface to another (through the `cross` statement) discard all latency information. Where these crossings occur, the programmer must pay extra attention to ensure correctness, much like with clock domain crossings.

```
module memory_block<gen int DEPTH, T> {
    interface write : T data, int addr, bool wr {
        T[DEPTH] memory;

        if wr {
            memory[addr] = data;
        }
    }
    interface read : int addr -> T data {
        cross memory;

        data = memory[addr];
    }
}
```

Listing 3: Example with two interfaces

REFERENCES

[1] Lennart Van Hirtum. 2023-. SUS Compiler Github Repository. https://github.com/pc2/sus-compiler

[2] Steven Hoover and Ahmed Salman. 2018. Top-Down Transaction-Level Design with TL-Verilog. arXiv:1811.01780 [cs.AR]

[3] Rachit Nigam, Pedro Henrique Azevedo de Amorim, and Adrian Sampson. 2023. Modular Hardware Design with Timeline Types. *Proc. ACM Program. Lang.* 7, PLDI, Article 120 (jun 2023), 25 pages. https://doi.org/10.1145/3591234

[4] Oron Port and Yoav Etsion. 2021. Registerless Hardware Description. https://capra.cs.cornell.edu/latte21/paper/4.pdf

[5] Frans Skarman and Oscar Gustafsson. 2022. Spade: An HDL Inspired by Modern Software Languages. https://spade-lang.org/fpl2022.pdf