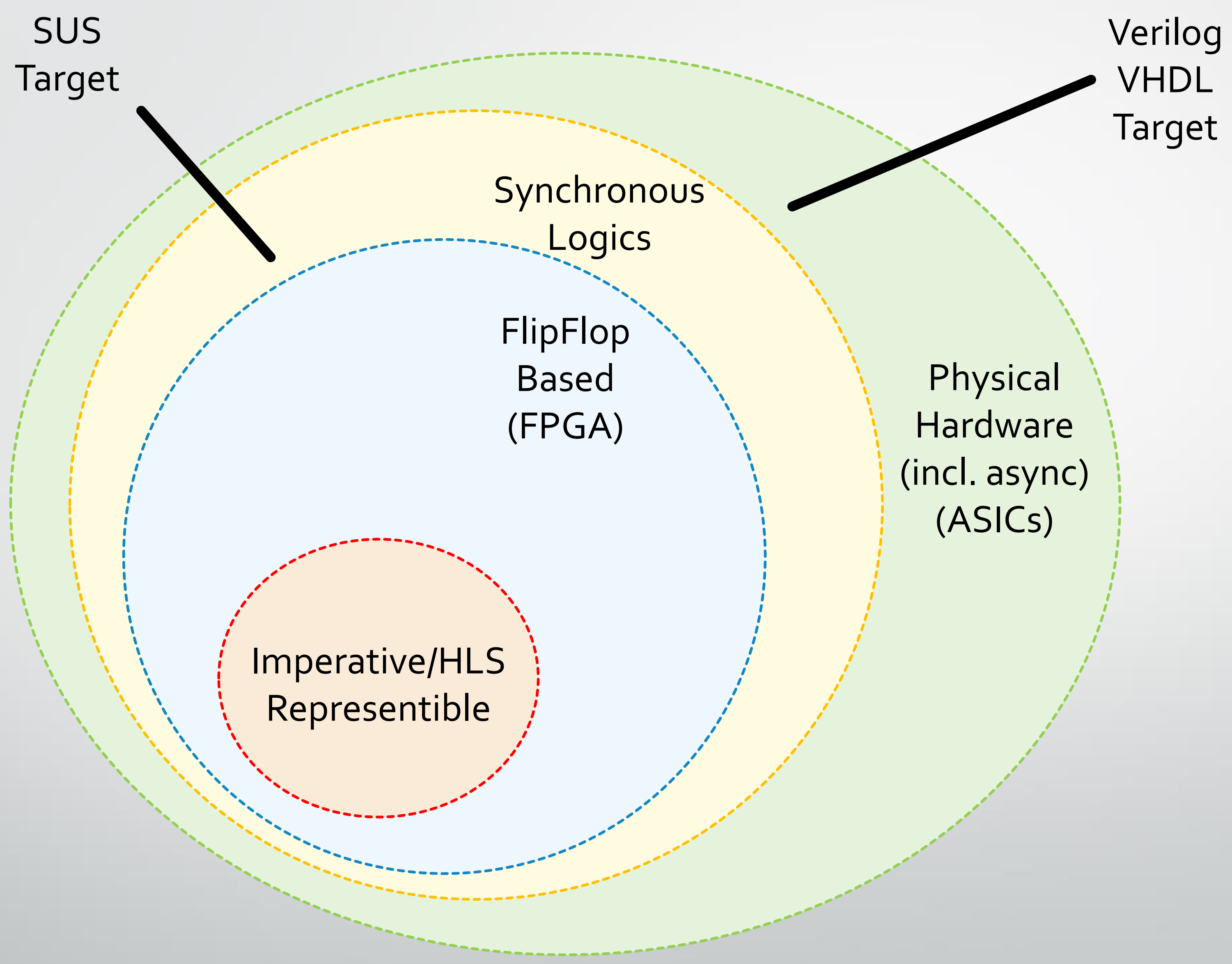
SUS Language

Motivation


SUS
Target
Verilog
VHDL
Target
Synchronous
Logics
FlipFlop
Based
(FPGA)
Physical
Hardware
(incl. async)
(ASICs)
Imperative/HLS
Representible

HLS

Advantages

- Fast to Develop
- CPU Simulation Easy
- Shallow-ish Learning Curve

Disadvantages

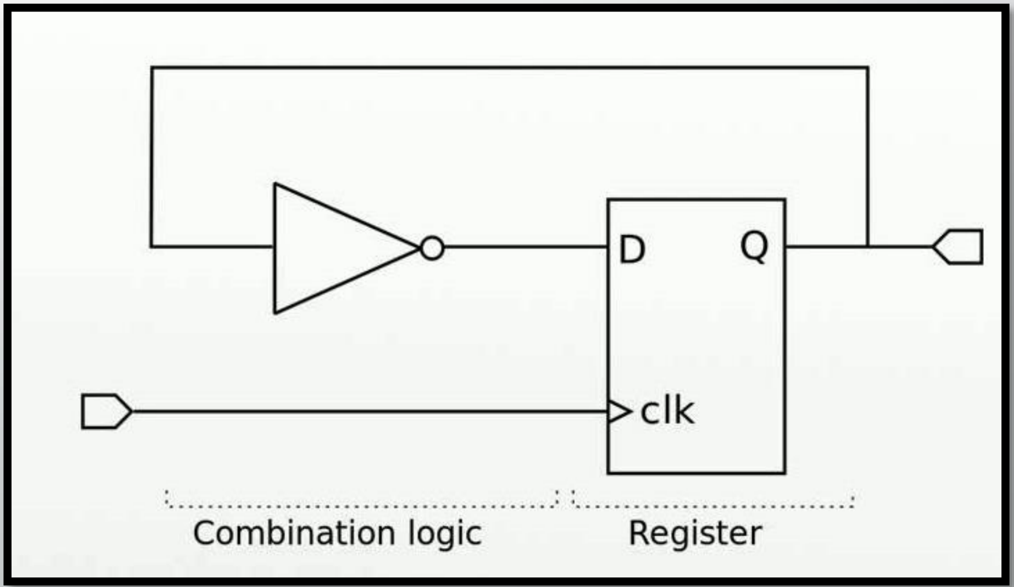
- No 1-to-1 Text to HW mapping
  - Hardware != Imperative
- Many HW not representible
- Intel & Xilinx Proprietary Monopoly
- *Require* compiler optimization

• *Require* Compiler Optimization

- Naive HW synthesis would have *terrible* performance
- Fiddle to make the compiler do The Right Thing™
- Compiler diagnostics aren't directly applicable to program text
  - « How do I make this component synthesize correctly? »
  - « Why is my clock frequency so slow? »
  - « Why do I only have 50% HBM throughput? »
  - « Why am I getting such routing congestion? »

RTL Is No Silver Bullet

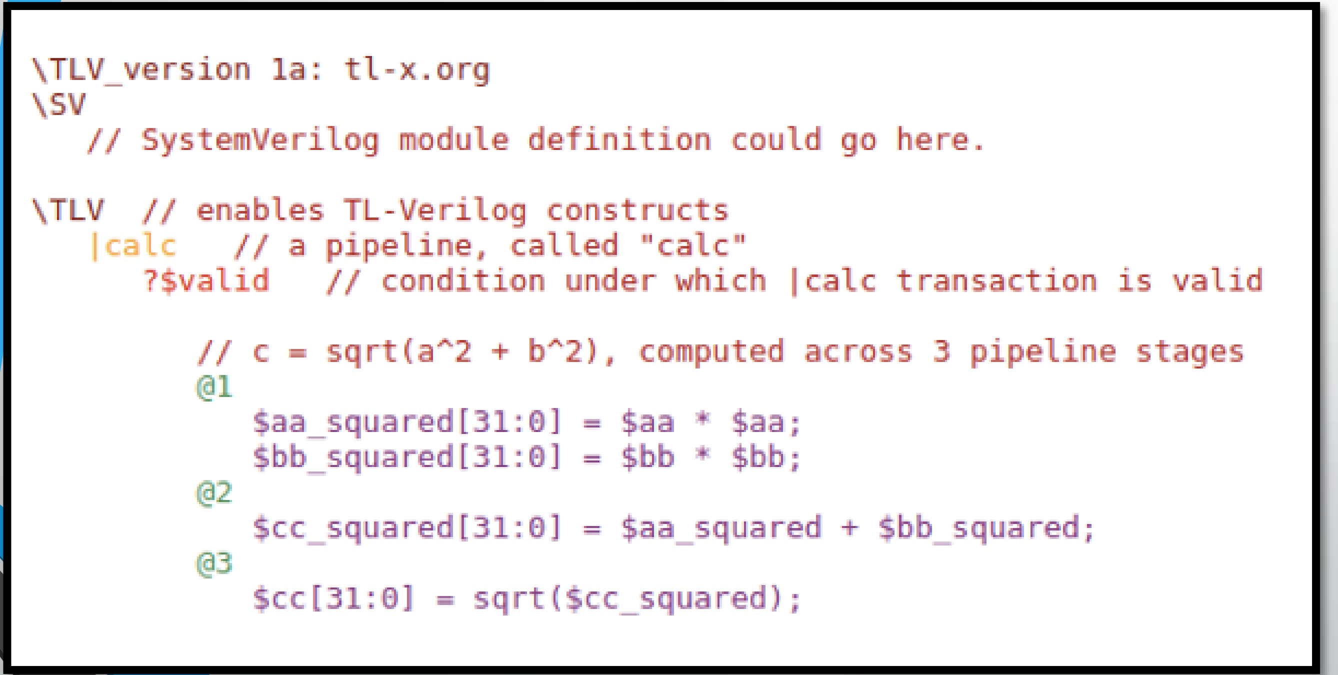
- Trading HLS for RTL is like dropping from python to assembly.
- Sure, you get strong control, but development slows to a snails pace
- Many classes of bugs:
  - FIFO overruns
  - Timing miscalculations
  - Stale State
  - Bad resets
- Modifications require extensive changes



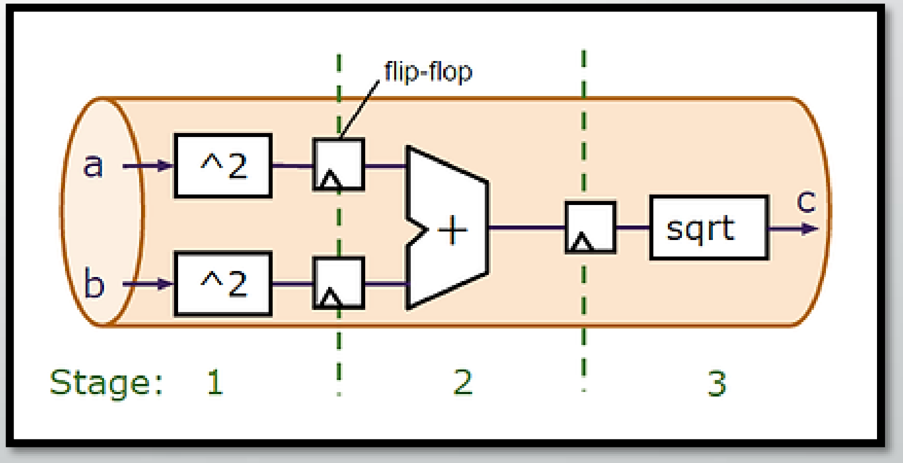
RTL SOTA

- **Verilog/VHDL** – Main drivers in enterprise
- **SystemVerilog** – Abstraction, Interfaces, Mostly for sim
- **TL-Verilog** – Easy pipelining notation
- **Filament** – Stateful pipelines
- **Chisel/Spinal/BlueSpec** – Strong metaprogramming

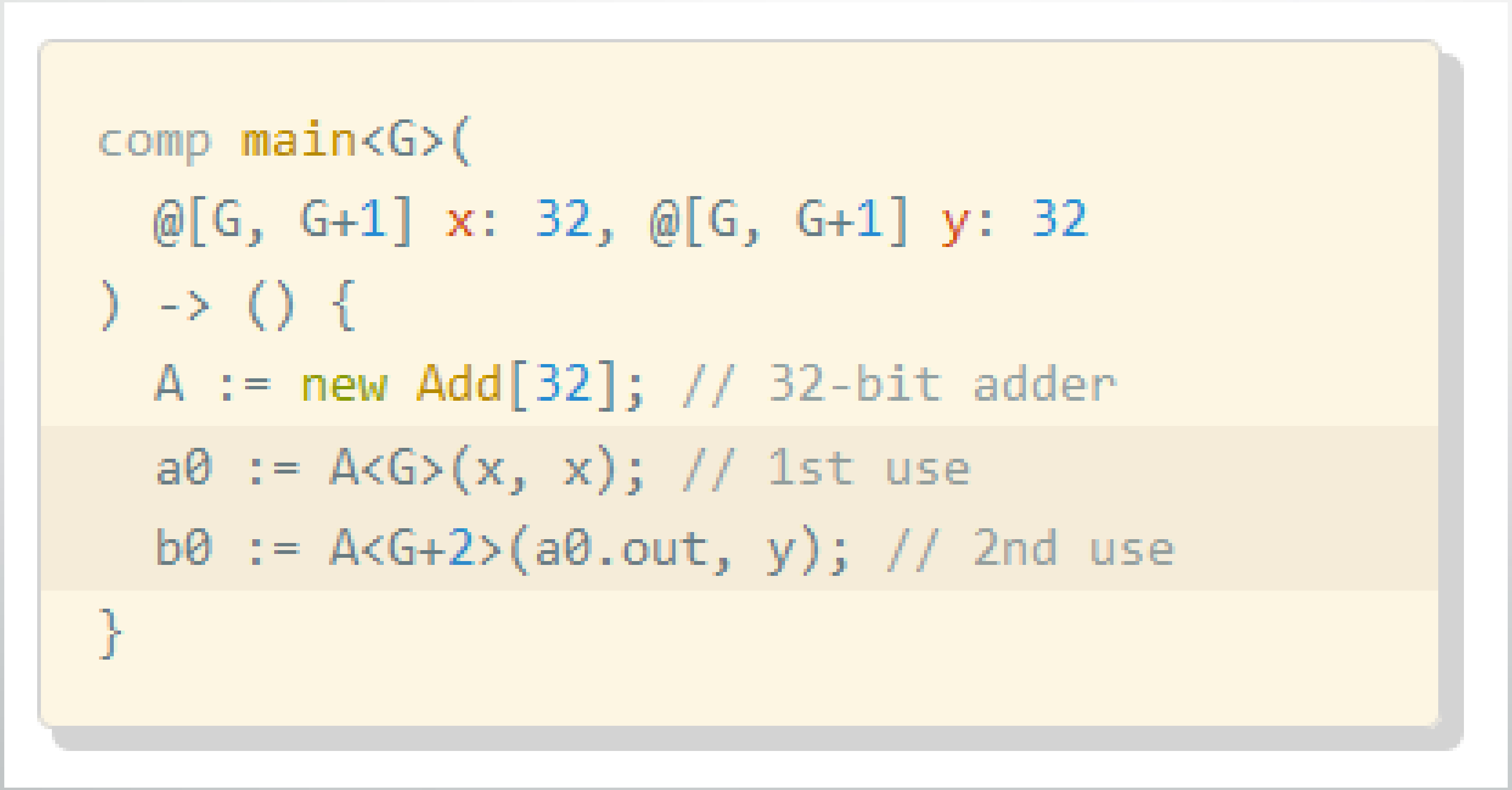
TL-Verilog



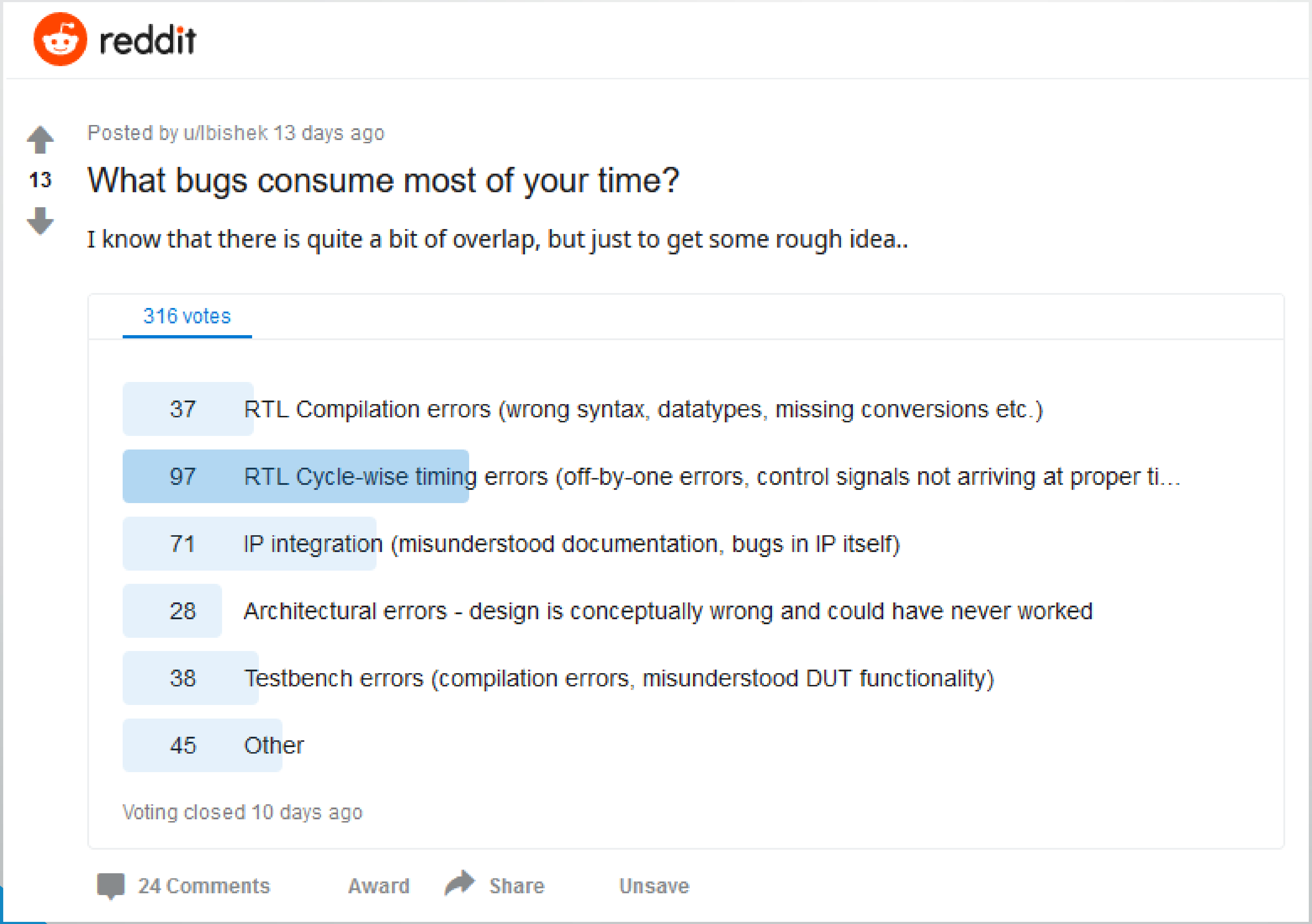
```
\TLV_version 1a: tl-x.org
\SV
   // SystemVerilog module definition could go here.

\TLV  // enables TL-Verilog constructs
   |calc   // a pipeline, called "calc"
      ?$valid   // condition under which |calc transaction is valid

         // c = sqrt(a^2 + b^2), computed across 3 pipeline stages
         @1
            $aa_squared[31:0] = $aa * $aa;
            $bb_squared[31:0] = $bb * $bb;
         @2
            $cc_squared[31:0] = $aa_squared + $bb_squared;
         @3
            $cc[31:0] = sqrt($cc_squared);
```



Filament



```
comp main<G>(
  @[G, G+1] x: 32, @[G, G+1] y: 32
) -> () {
  A := new Add[32]; // 32-bit adder
  a0 := A<G>(x, x); // 1st use
  b0 := A<G+2>(a0.out, y); // 2nd use
}
```



Posted by u/lbishek 13 days ago

13

What bugs consume most of your time?

I know that there is quite a bit of overlap, but just to get some rough idea..

316 votes

37	RTL Compilation errors (wrong syntax, datatypes, missing conversions etc.)
97	RTL Cycle-wise timing errors (off-by-one errors, control signals not arriving at proper ti...
71	IP integration (misunderstood documentation, bugs in IP itself)
28	Architectural errors - design is conceptually wrong and could have never worked
38	Testbench errors (compilation errors, misunderstood DUT functionality)
45	Other

Voting closed 10 days ago

Goals

- Low-Level control
- Terseness
- Compile-Time Verification
- Common modifications should be easy
- Full Replacement of Verilog and VHDL in FPGA Space

Non-Goals

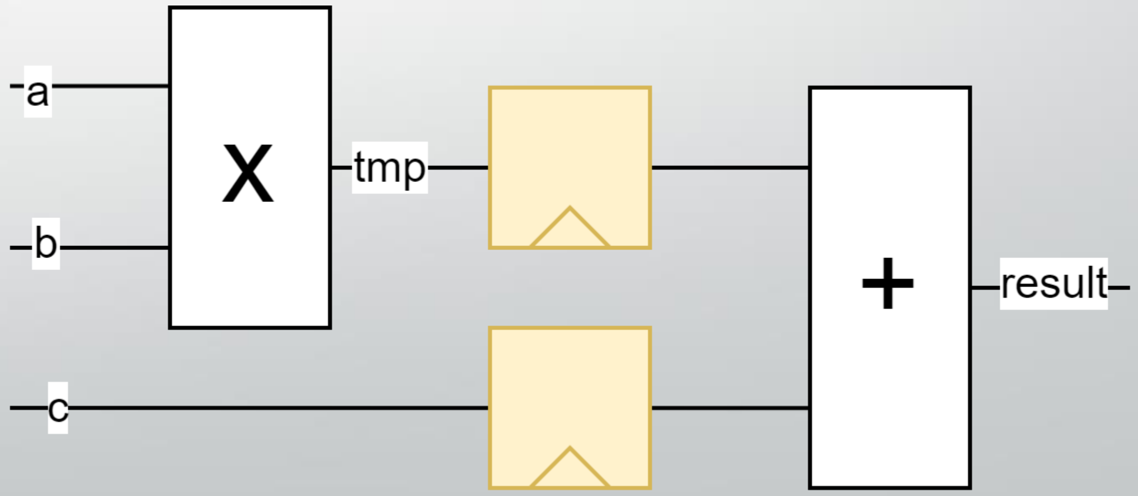
- Optimization
- Custom Synthesis
- BSP Replacement
- High Clock Speed Guarantees
- Turing-Complete hardware generation
- ASIC Synthesis

Initial Ideas

Easy Pipelining

```
module multiply_add(
  input clk,
  input[31:0] a,
  input[31:0] b,
  input[31:0] c,
  output[31:0] result_D
) {
  reg[31:0] tmp_D;
  reg[31:0] c_D;

  always @(posedge clk) begin
    tmp_D <= a * b;
    c_D <= c;
  end
  assign result_D = tmp_D + c_D;
}
```

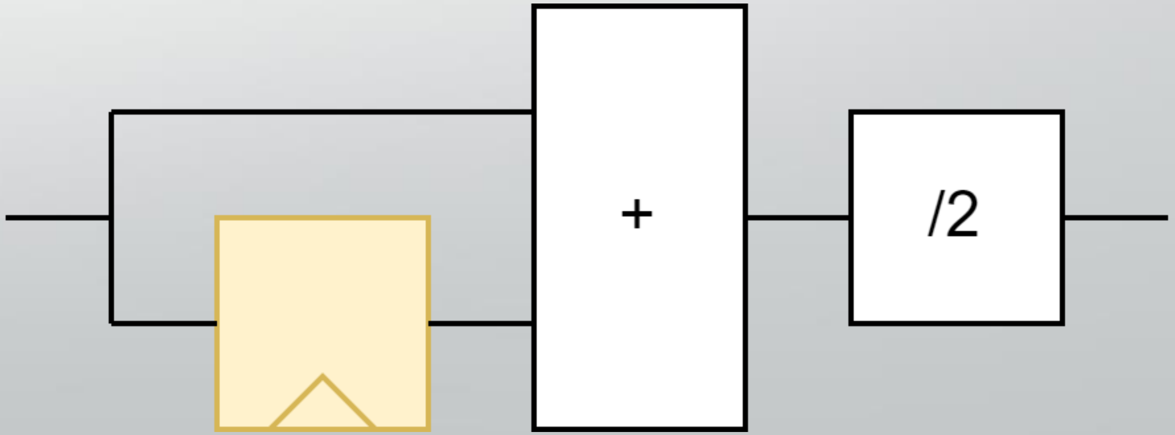
```
module multiply_add :
  int a,
  int b,
  int c
  -> int result
{
  int tmp = a * b;
  @
  result = tmp + c;
}
```



Stream Processing

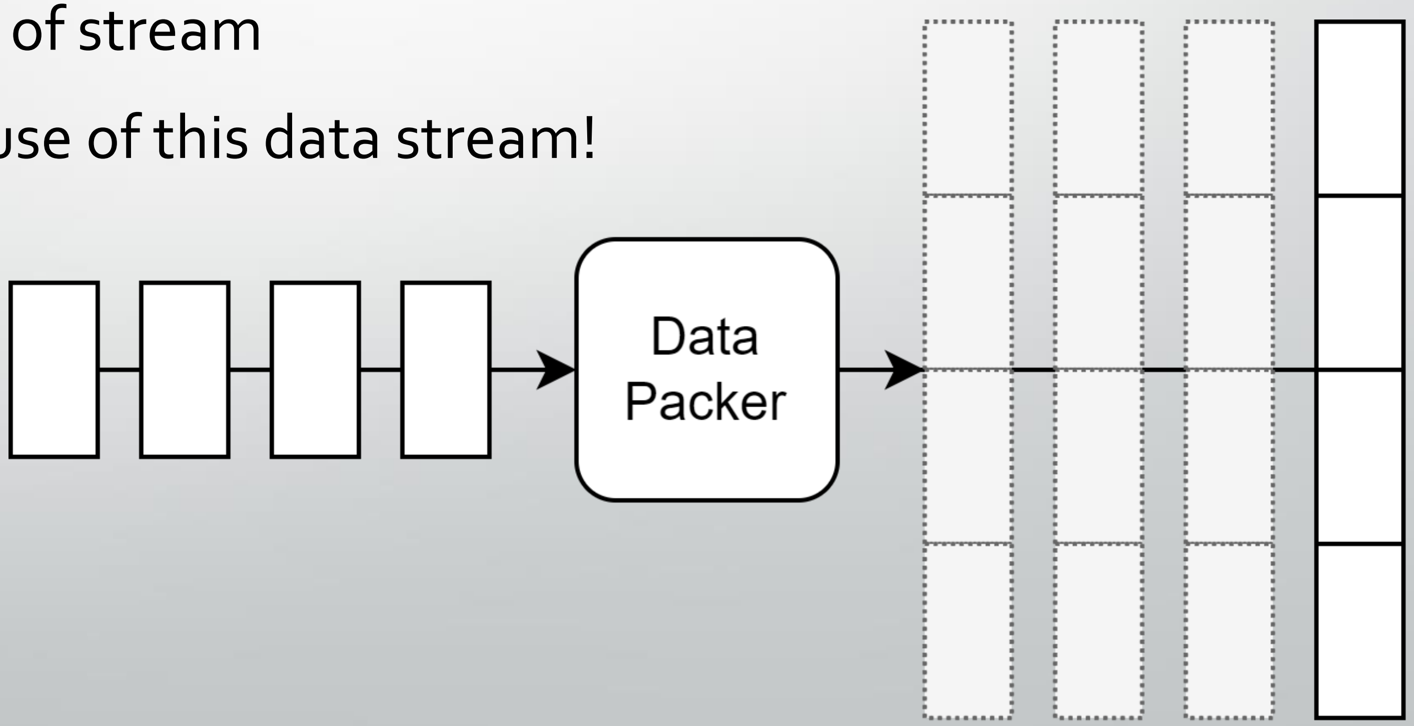
```
module blur2(
  input clk,
  input[31:0] data,
  output[31:0] blurred
) {
  reg[31:0] prev;

  always @(posedge clk) begin
    prev <= data;
  end
  assign blurred = prev + data / 2;
  // when is blurred valid?
}
```

```
// Explicitly state when valid
timeline (v->/) .. (v->v)*
module blur2 :
  int data -> int blurred
{
  state int prev = data;
  #
  loop {
    blurred = prev + data / 2;
    prev = data;
    #
  }
}
```



Multi-Cycle Protocols

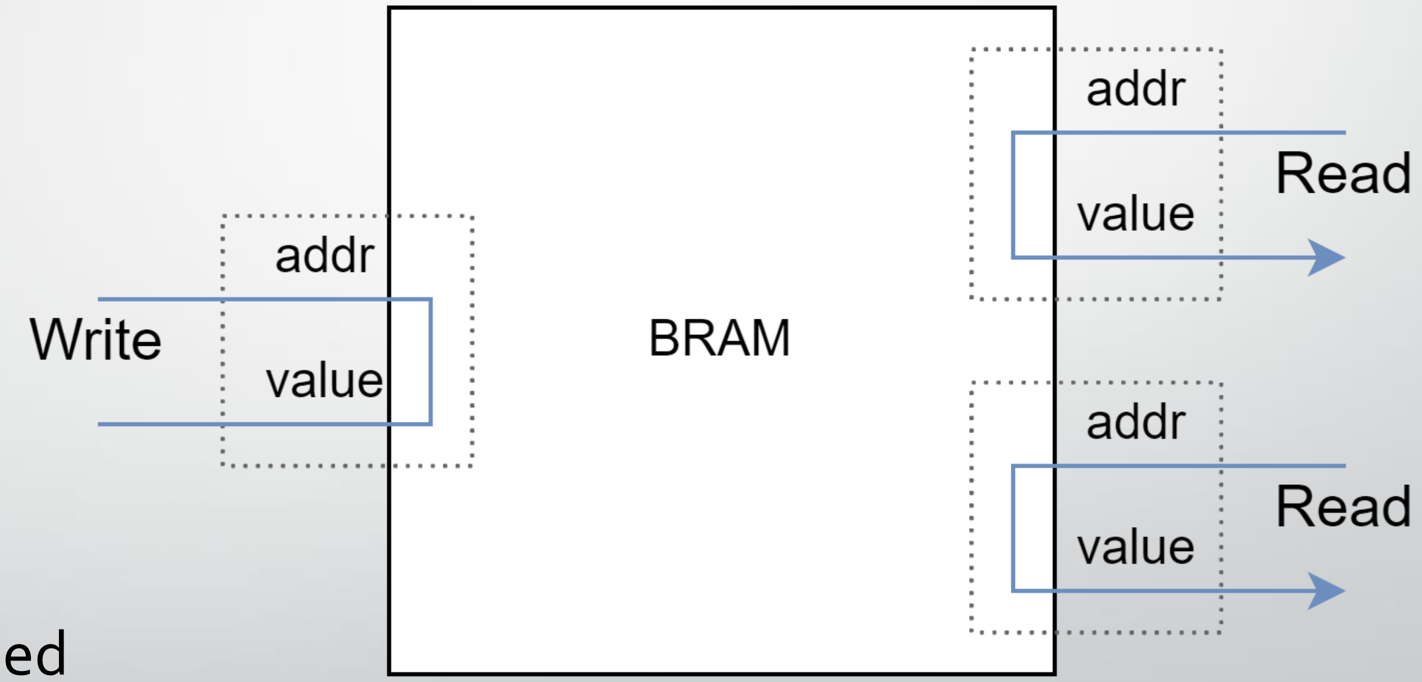
- Some data streams require multiple cycles
- Define shape and validities of stream
- Can type-check on proper use of this data stream!



True Stateful modules

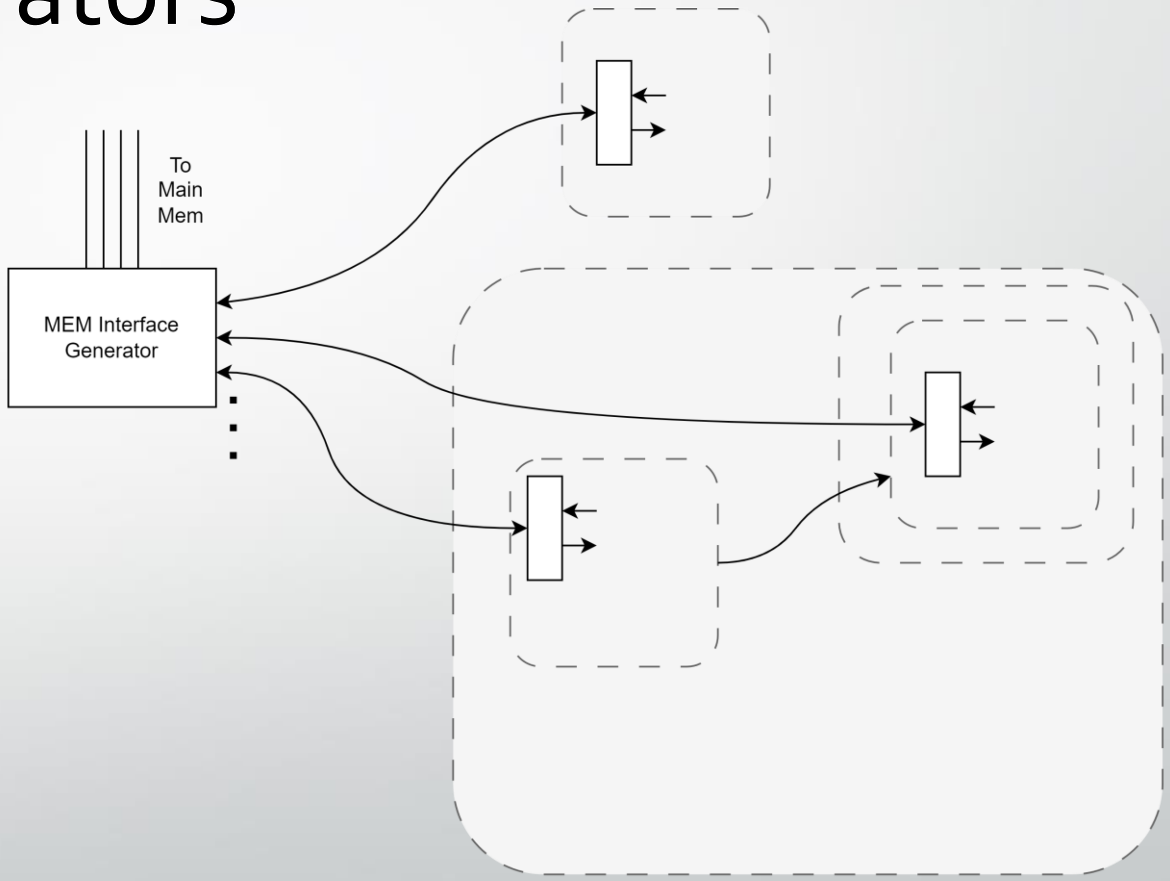
- FIFOs
- Memory
- PID Controllers

- Blob of State
- Interfaces are pipelined

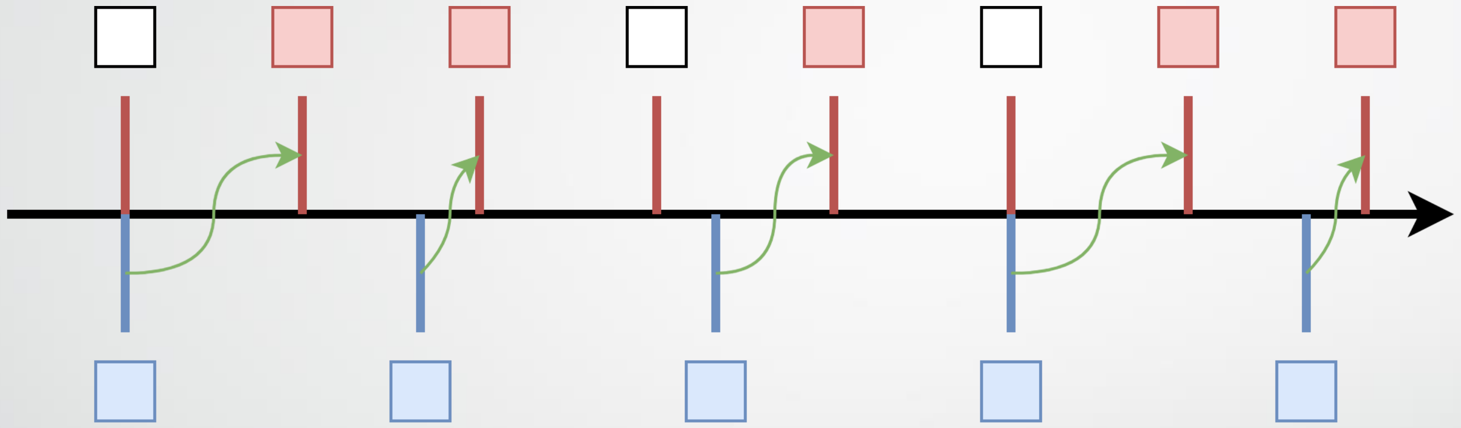


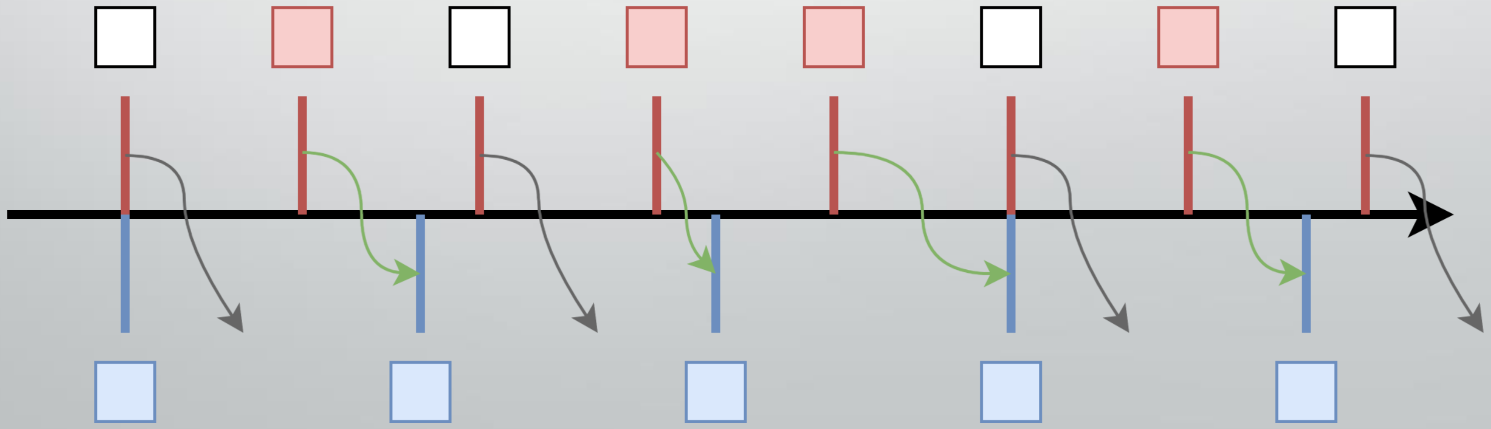
Generators

- Dependency Injection
- Modules can have generator interface that can be instantiated multiple times.
- Module is then actually instantiated with array of all generated instantiations
  - Main Memory Interface
  - Multi-Port BRAMs
  - Externally Injecting ECC Checks



Rhythms for Clock Domain Crossings





Nifty consequences

- Automate FIFO sizing
- Automate reset generation
- Integer min-max sizing instead of bit-widths
- Generated hardware can make extensive use of “Don’t care”
    - Aids in Simulation

Open Questions

Open Questions

- Safety for stream ordering?
- State Invalidation?
- How extensive should protocol specs be?
- (Semi-) Constants?
- “Compute Phases”?
- Generating timing constraints?
- Generic vs Concrete conflict
- Modules requiring unpredictable input stalls?
- Resource saving on long register chains?
- Project Scope?
- Host to FPGA Comms?