SUS Compiler

Goals

- Retain low-level control over generated hardware
  - Individual Statements should map onto hardware as directly as possible
- But more compact than VHDL/(System) Verilog
- Compile-Time Verification
- No Guessing, no trusting the compiler to do The Right Thing™
- Common modifications should be easy
- Reduce “State” as much as possible.
  - Can yell at programmer more often

Why not HLS?

- Hardware != Imperative
- Fiddly
  - Rely on compiler optimization for decent hardware
  - Why is my pipeline only having 50% throughput?
- Cannot express exact timing requirements
- Intel & Xilinx proprietary monopolies.

reddit

13

Posted by u/lbishek 13 days ago

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

24 Comments Award Share Unsave

From speaking with people in industry

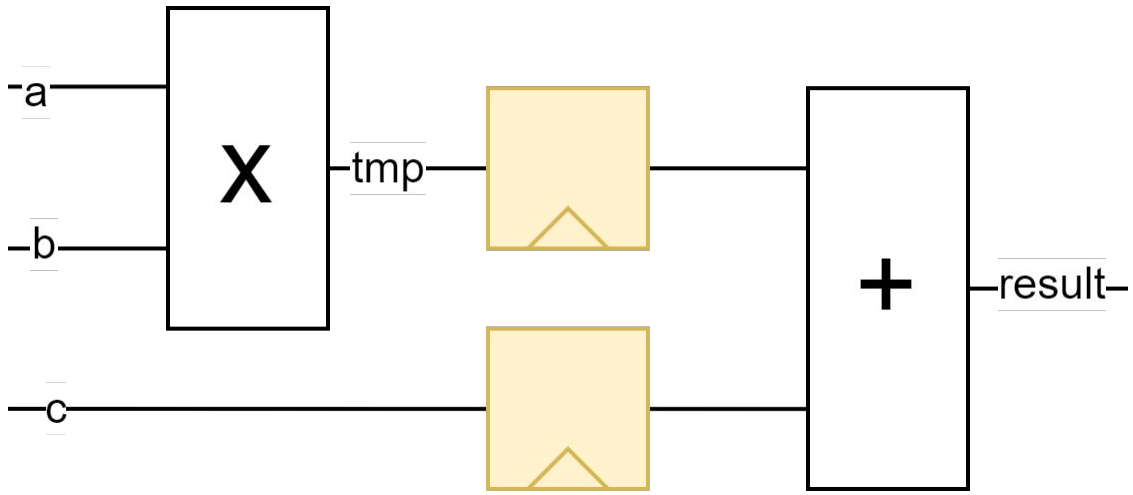
- Clock Domain Crossings are VERY difficult
- Proper timing constraints also difficult

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;
}
```

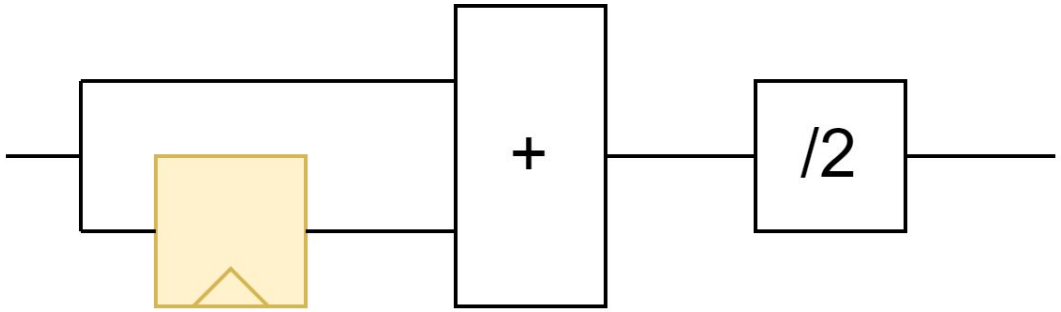


Stream Processors

```
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;
    #
  }
}
```

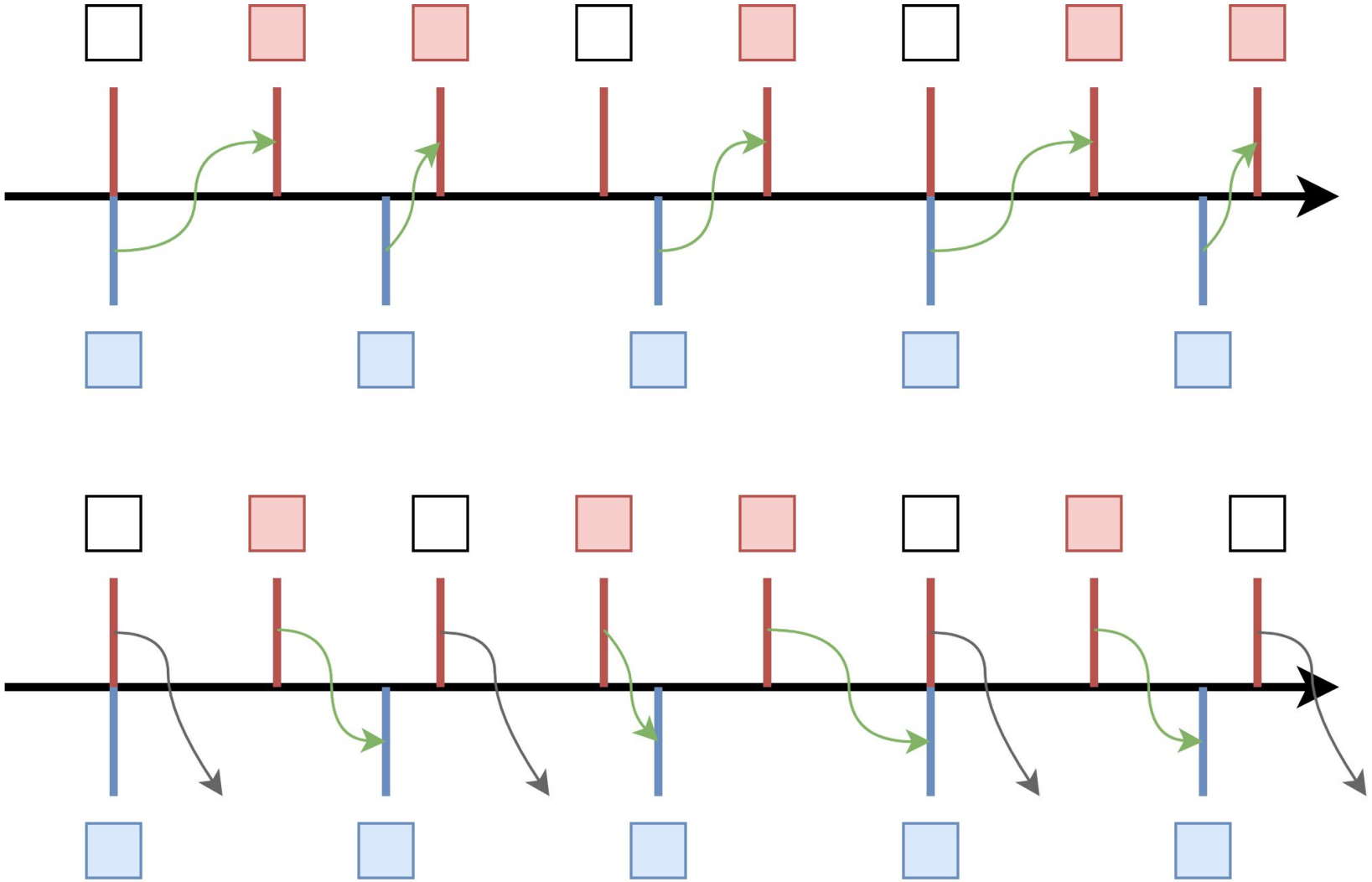


Pipe and Time combination

```
module multiply_aggregate(
  input clk,
  input start,
  input done,
  input[31:0] data,
  output reg[31:0] sum,
  output sumValid
) {
  always @(posedge clk) begin
    if(start) {
      sum <= 0;
    } else if(done) {

    }
    prev <= data;
  end
  assign blurred = prev + data / 2;
}
```

Rhythms for Clock Domain Crossings



True Stateful modules

- FIFOs
- Memory
- Control Circuits

Neat safety features

- Automate FIFO sizing
- Automate reset duration sizing
- Integer min-max sizing instead of bit-widths
- Rhythms for Clock Domain Crossings
- Generated hardware can make extensive use of “Don’t care”
  - Aids in Simulation

Open Questions

- Safety for stream ordering?
- CPU and Cache Safety by state invalidation?
- How extensive should timeline protocol specs be?
- Dealing with constants?
- “Compute Modes”?
- Generating timing constraints?
- Generic vs Concrete conflict
- What about stream modules that require unpredictable input stalls?
- Resource saving on long register chains?