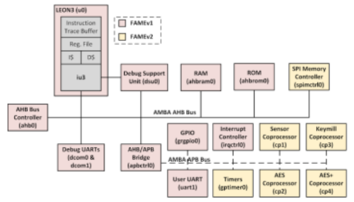
Cryptographic Hardware Optimization for ASIC

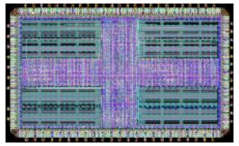
Patrick Schaumont
Electrical and Computer Engineering

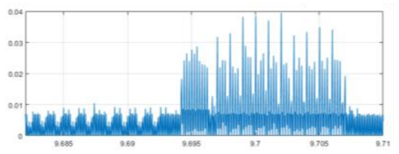
pschaumont@wpi.edu

Vernam Lab

Vernam is a Research Center of Excellence in Cybersecurity and Hardware Security in the New England area. A group of seven faculty and their students tackle advanced research challenges across the systems stack.







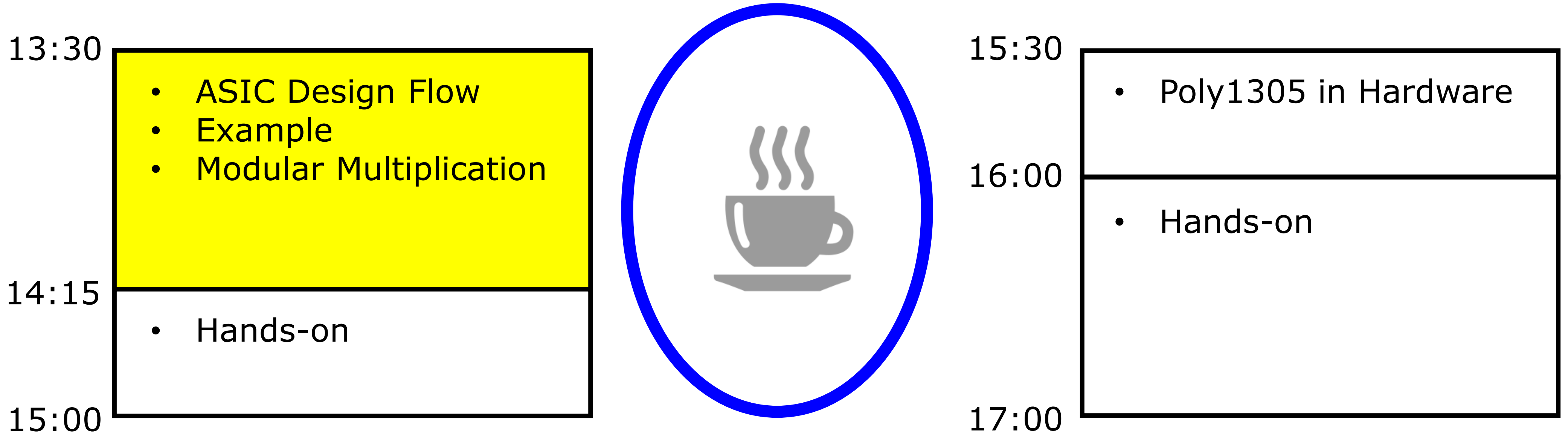
Worcester Polytechnic Institute

WPI

The Big Picture

https://github.com/Secure-Embedded-Systems/proact-asic-opt/

The big picture



Assumptions

- You understand basic hardware design
  - Logic gate, register, clock cycles, propagation delay do not sound 100% foreign
- You understand the Unix command line
  - You understand the futility of `dd if=/dev/zero of=ohno bs=1G`
- You understand basic ASIC design
  - https://summerschool-croatia.cs.ru.nl/2023/slides/Schaumont_crypto-asic-oss-prs.pdf
- You understand Verilog HDL
  - You don’t? https://hdlbits.01xz.net/wiki/Main_Page
- There is a handson portion to this tutorial
  - Recommend to do the handson in teams of two

ASIC design software

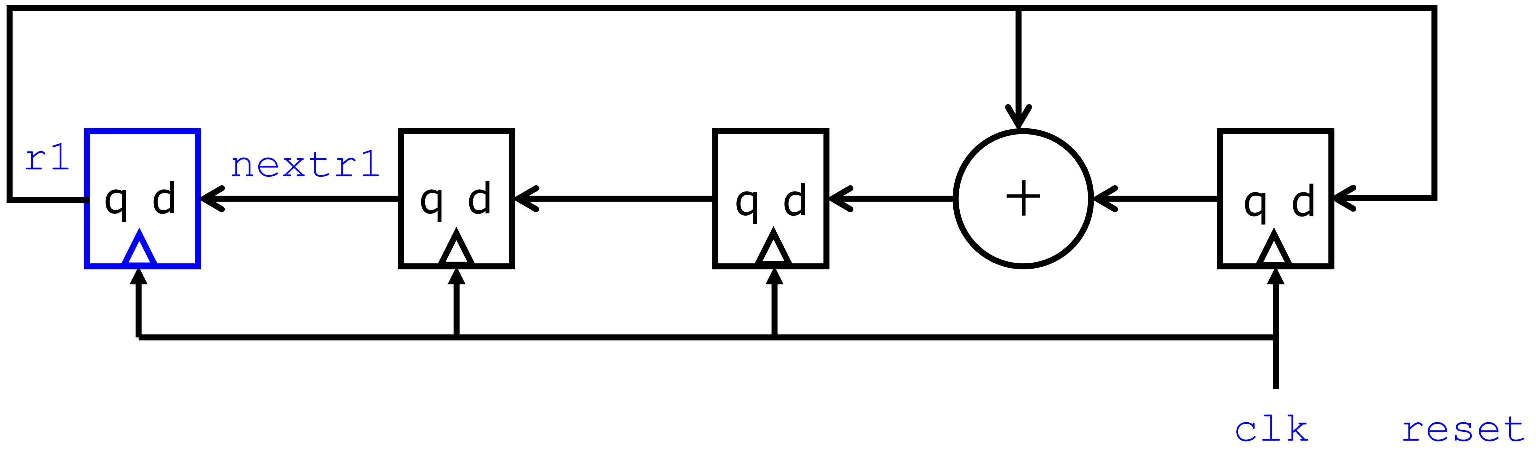
- Labs will be provided on a cloud server
- BUT you can run your own Ubuntu 22.04 or WSL2 box
  - Consult section: *Configuring the Design Workstation*
  - Instructions create the same design environment as on the cloud server
- As we go through the lecture slides today, feel free to experiment with design examples. All commands are shown in blue

```
# make -f Makefile.lfsr rtlsim
```

WPI

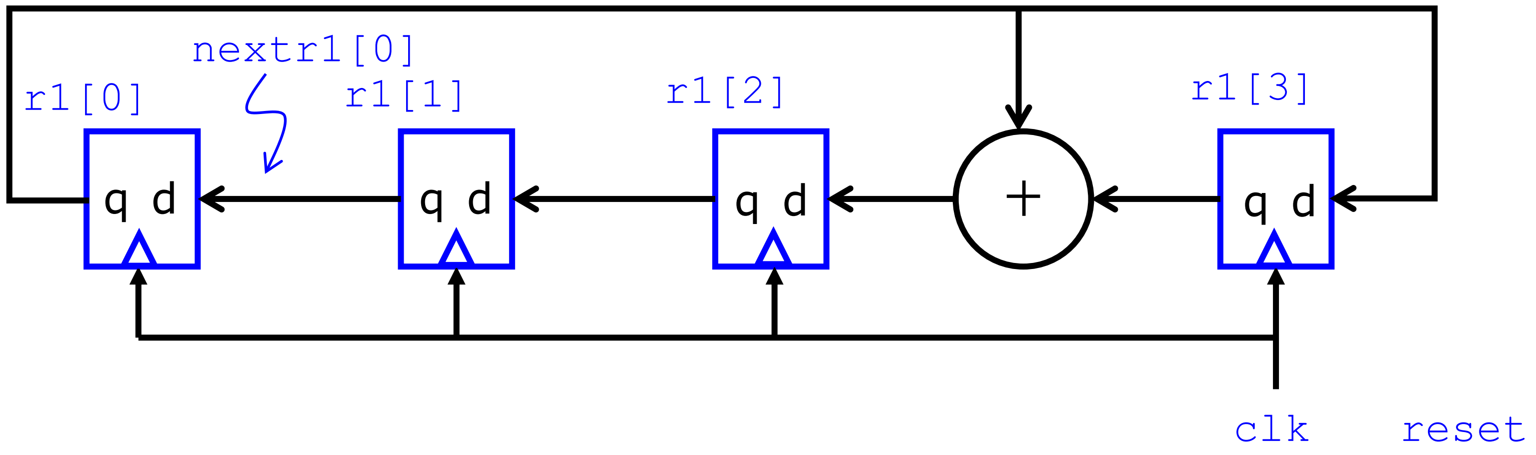
A simple chip: Hardware Design

LFSR: Verilog



```
reg r1;
wire nextr1;

always @(posedge clk)
  if (reset)
    r1 <= 1'b0;
  else
    r1 <= nextr1;
```

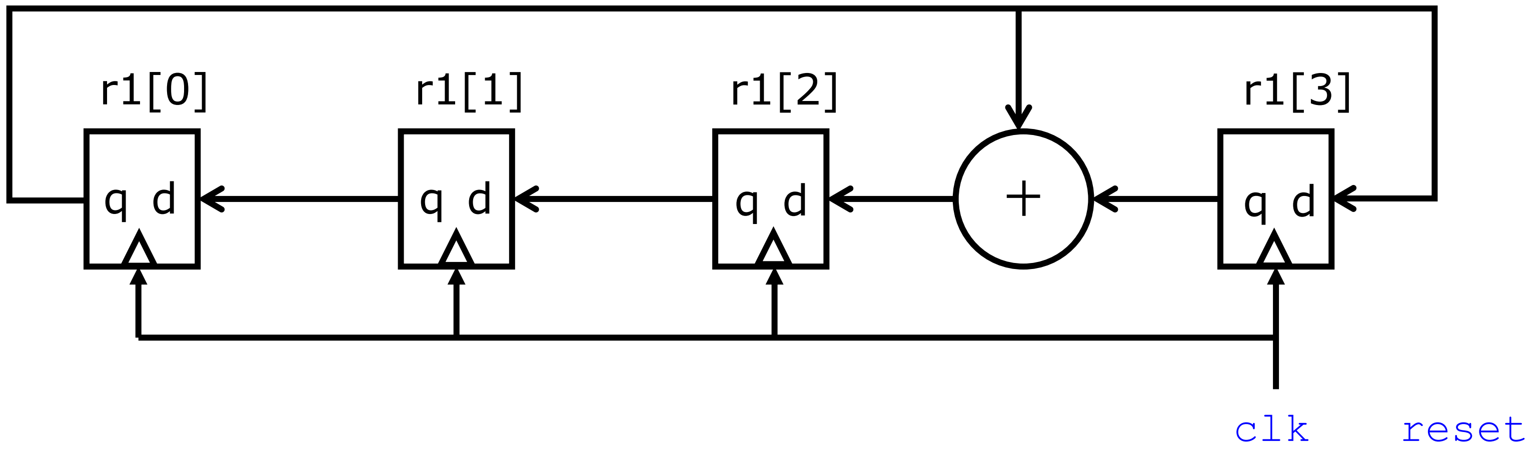
LFSR: Verilog



```
reg [3:0] r1;
wire [3:0] nextr1;

always @(posedge clk)
  if (reset)
    r1 <= 4'b1;
  else
    r1 <= nextr1;
```

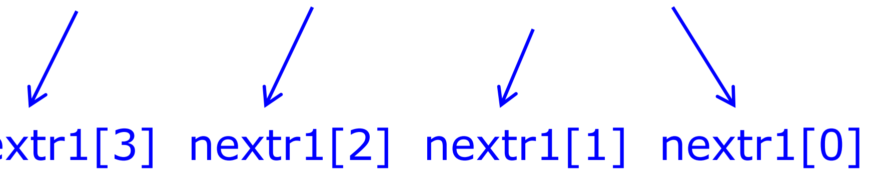
lfsr, must make sure that the initial state is nonzero

LFSR: Verilog

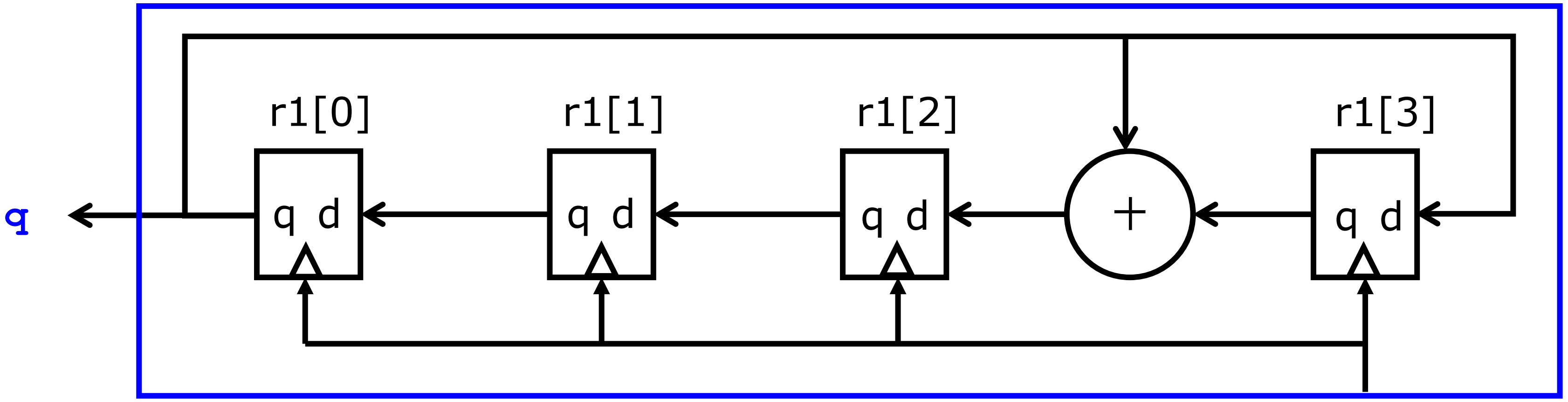


```
reg [3:0] r1;
wire [3:0] nextr1;

always @(posedge clk)
  if (reset)
    r1 <= 4'b1;
  else
    r1 <= nextr1;
```

```
assign
  nextr1 = {r1[0],r1[0]^r1[3],r1[2],r1[1]};
```



LFSR: Verilog

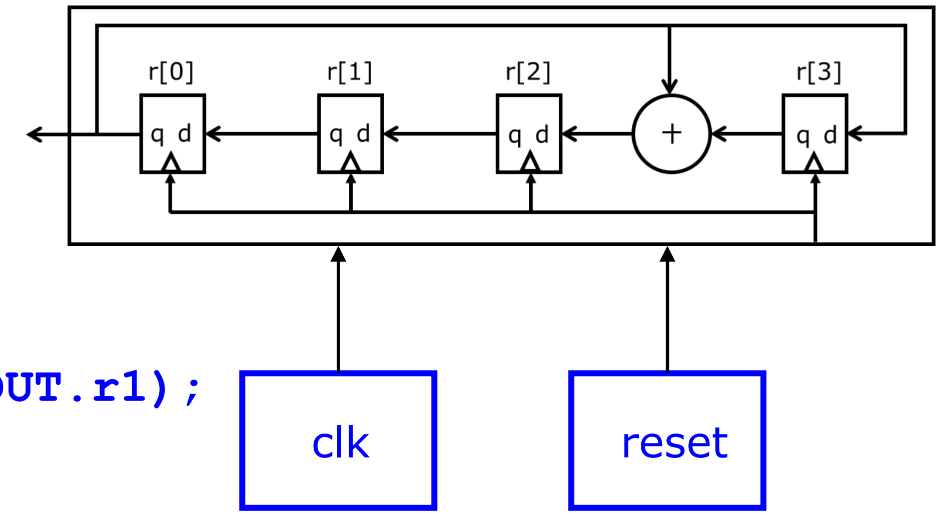


```
module lfsr(input clk, input reset, output q);
  reg [3:0] r1;
  wire [3:0] nextr1;
  always @(posedge clk)
    if (reset)
      r1 <= 4'b1;
    else
      r1 <= nextr1;
  assign
    nextr1 = {r1[0],r1[0]^r1[3],r1[2],r1[1]};
  assign q = r1[0];
endmodule
```

Testing LFSR

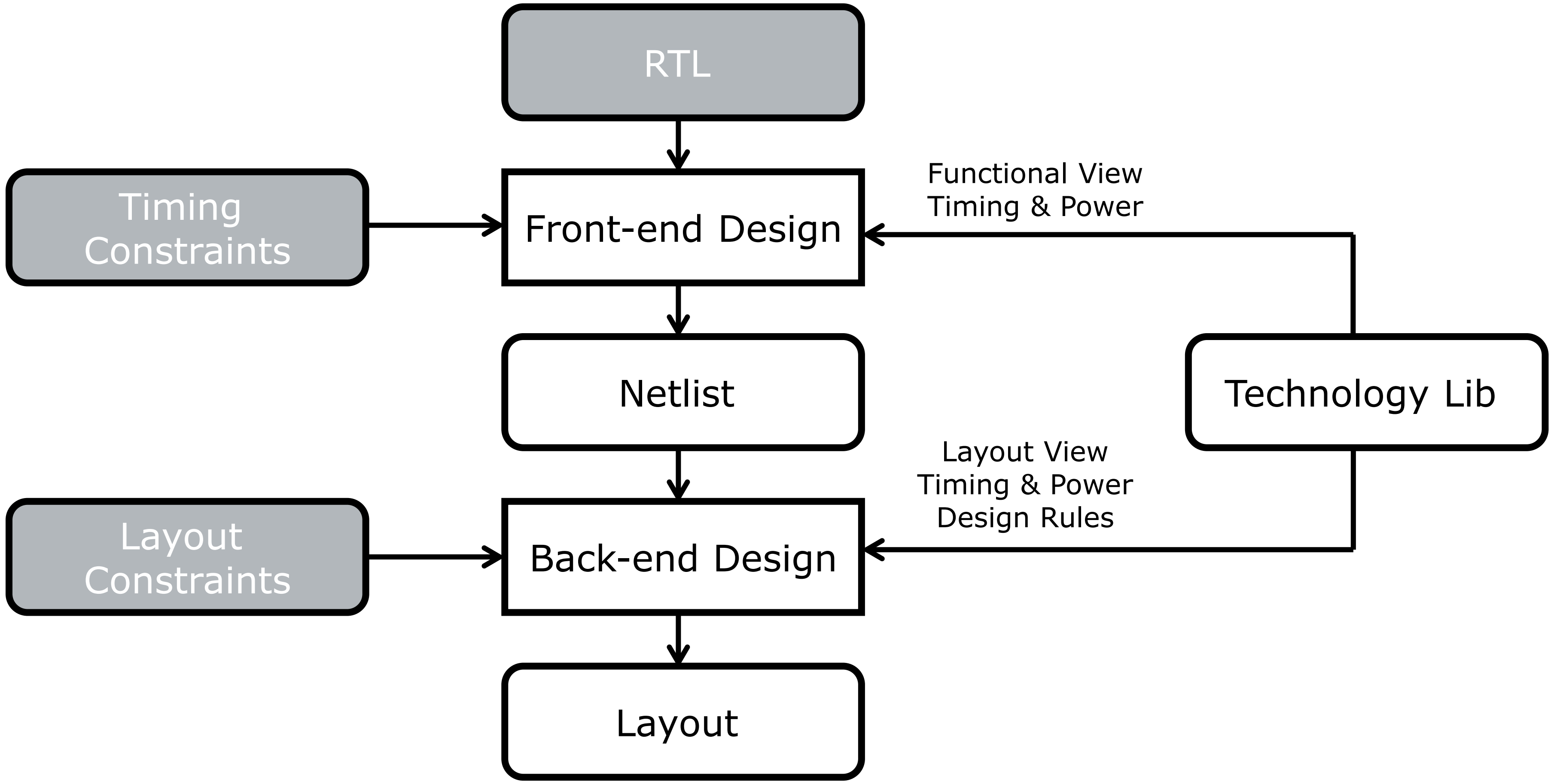
```
module tb;
  reg clk;
  reg reset;
  wire q;
  lfsr DUT(.clk(clk),
           .reset(reset),
           .q(q));

  always
    begin
    clk = 1'b0;
    #5;
    clk = 1'b1;
    #5;
    end
```

```
initial
    begin
    $dumpfile("trace.vcd");
    $dumpvars(0, tb);
    reset = 1'b1;
    repeat (2)
      @(posedge clk);
    reset = 1'b0;
    repeat (20)
      begin
      @(posedge clk);
      $display("%b", DUT.r1);
      end
    $finish;
    end

endmodule
```

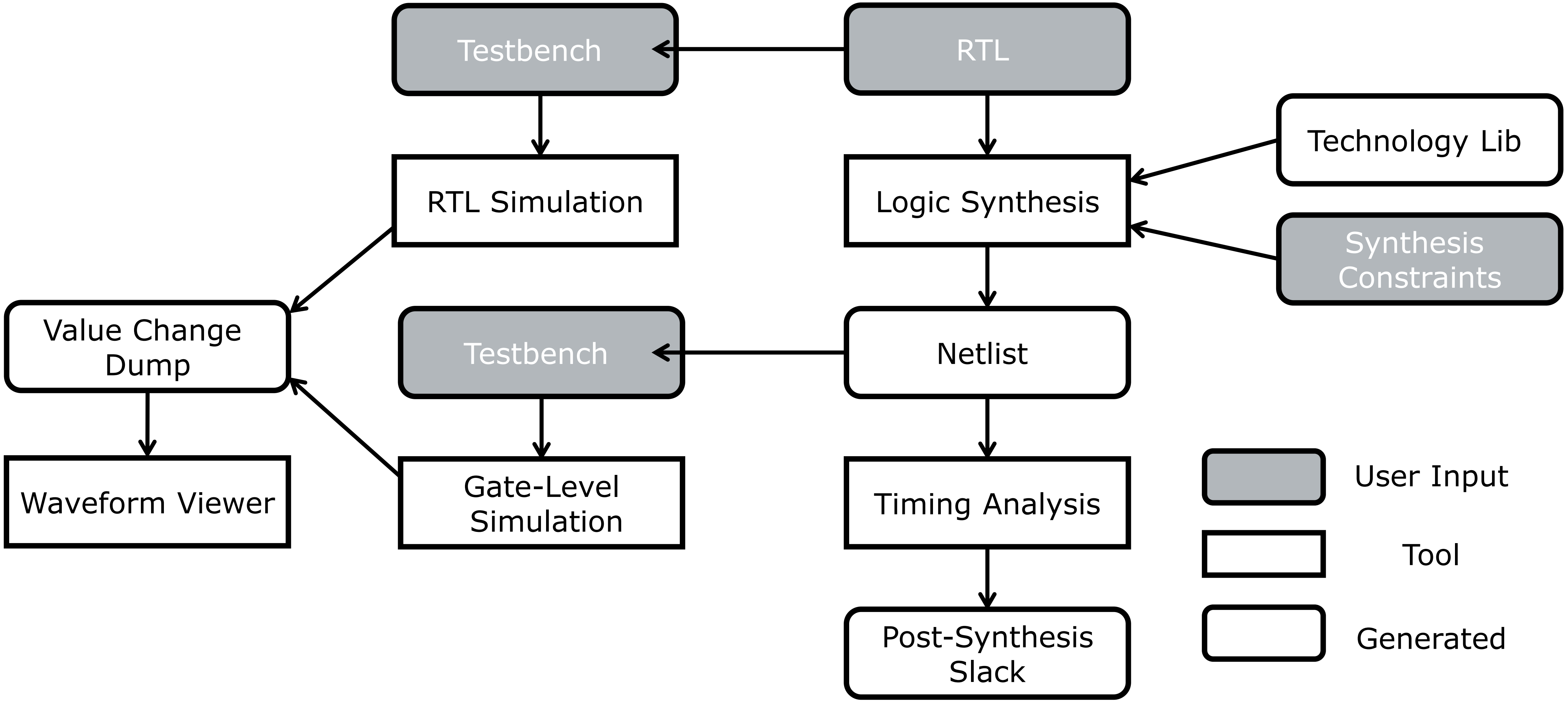


Open-source ASIC Design Flow

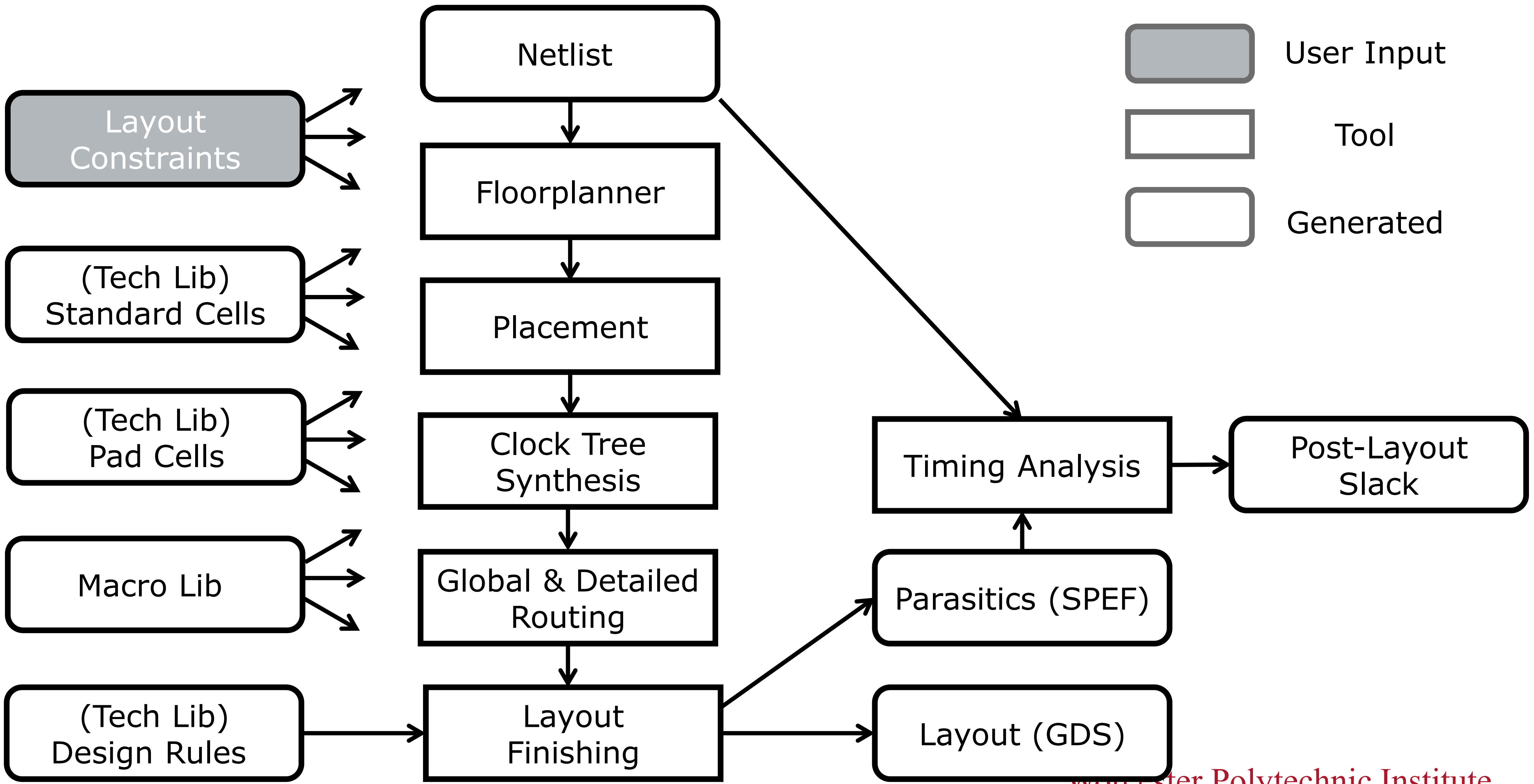
ASIC Design Flow



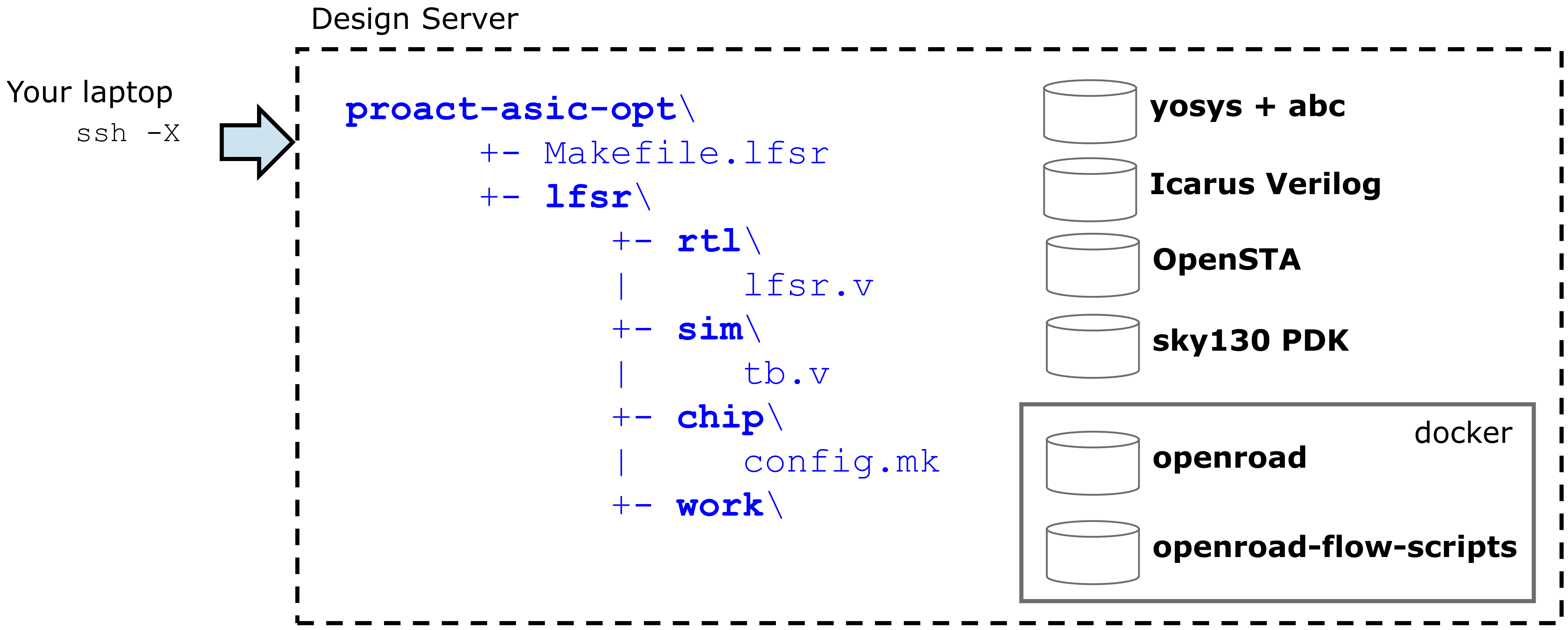
Front-end Design



Back-end Design

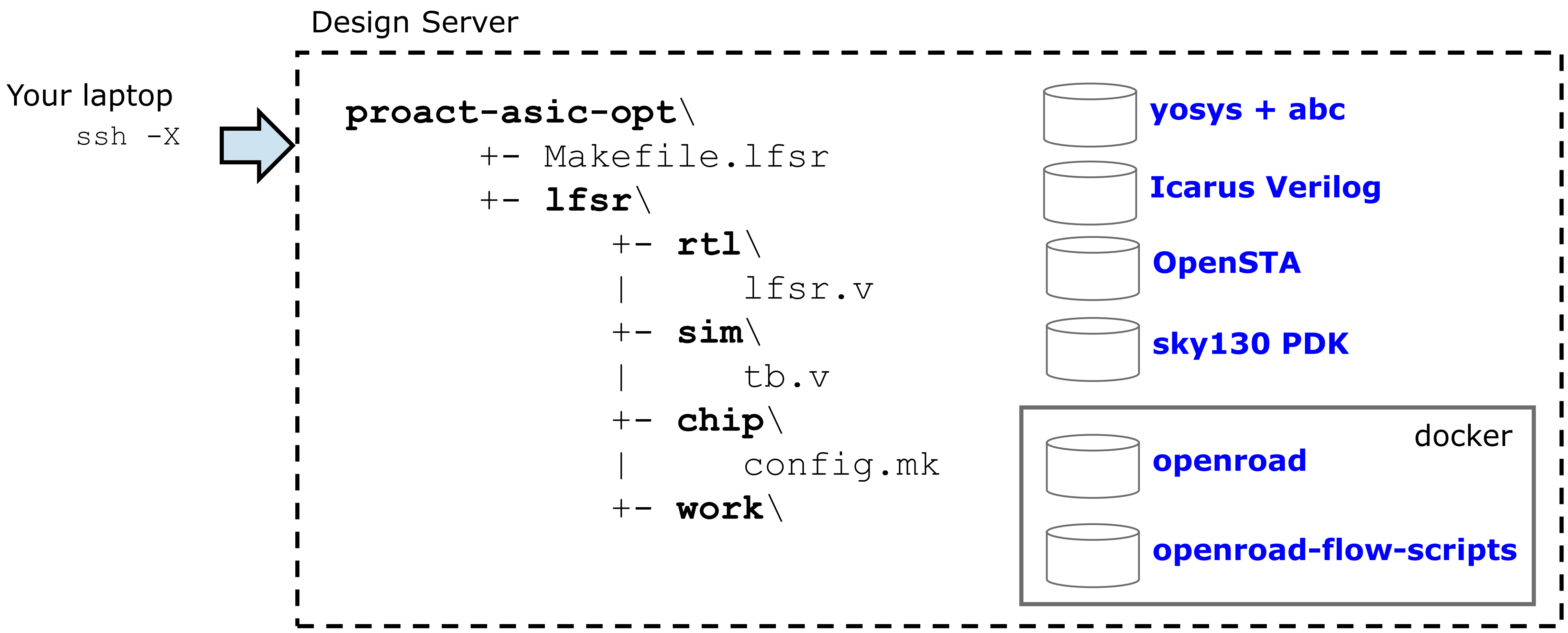


Handson Infrastructure and Tooling



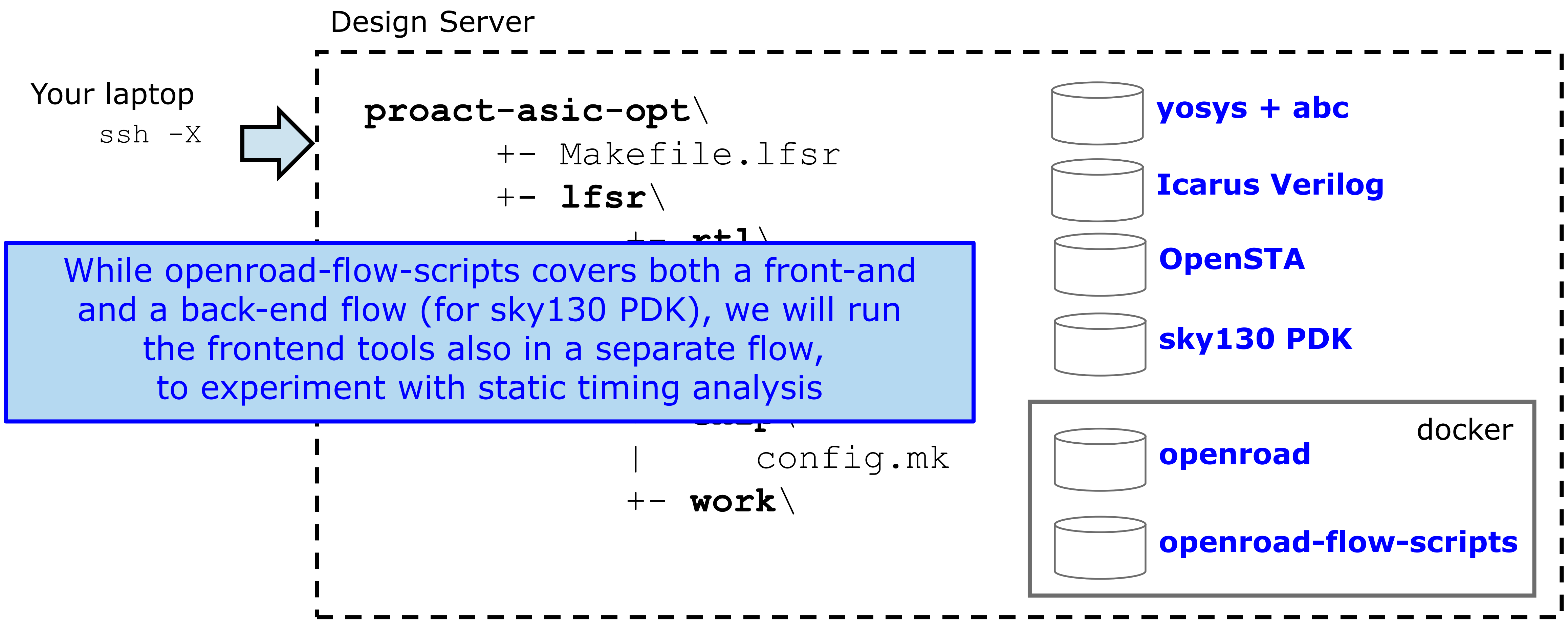
```
proact-asic-opt\
        +- Makefile.lfsr
        +- lfsr\
                +- rtl\
                |       lfsr.v
                +- sim\
                |       tb.v
                +- chip\
                |       config.mk
                +- work\
```

Handson Infrastructure and Tooling



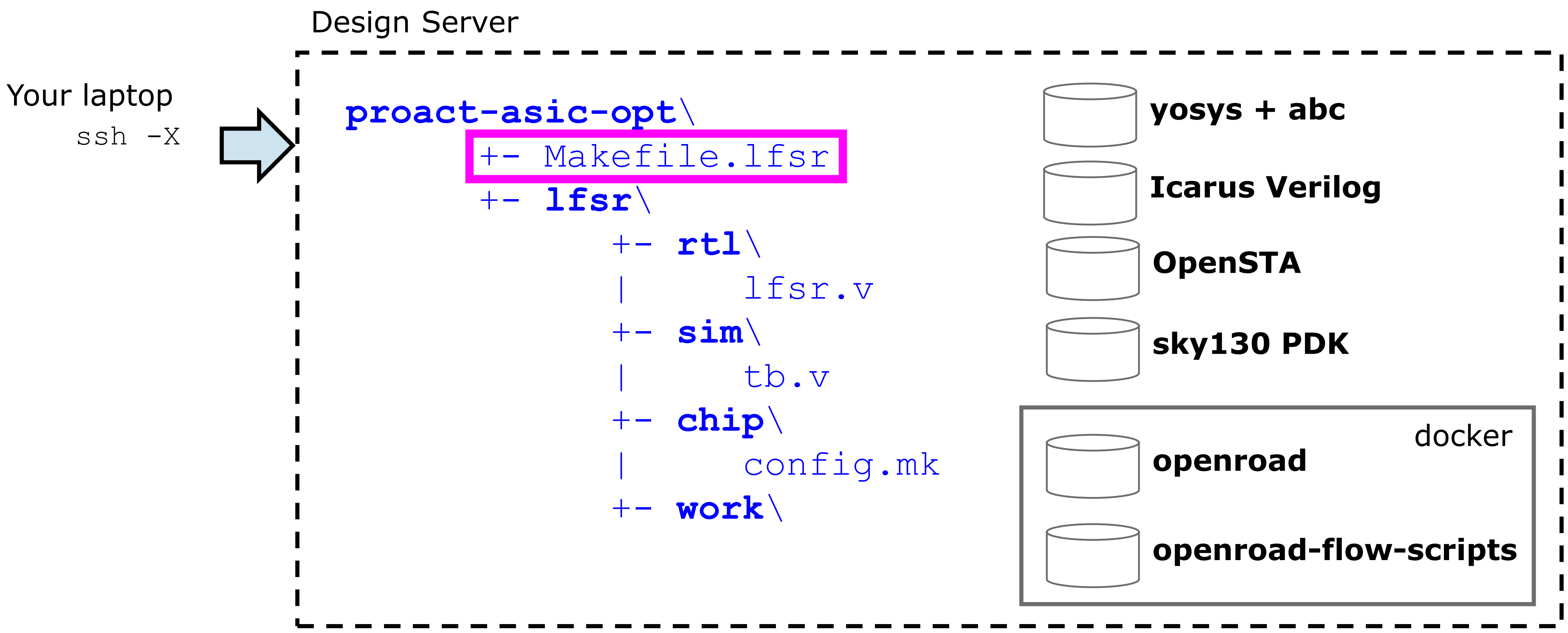
```
proact-asic-opt\
          +- Makefile.lfsr
          +- lfsr\
                    +- rtl\
                    |       lfsr.v
                    +- sim\
                    |       tb.v
                    +- chip\
                    |       config.mk
                    +- work\
```

Handson Infrastructure and Tooling



While openroad-flow-scripts covers both a front-and and a back-end flow (for sky130 PDK), we will run the frontend tools also in a separate flow, to experiment with static timing analysis

Handson Infrastructure and Tooling



```
proact-asic-opt\
      +- Makefile.lfsr
      +- lfsr\
            +- rtl\
            |      lfsr.v
            +- sim\
            |      tb.v
            +- chip\
            |      config.mk
            +- work\
```

Makefile.lfsr

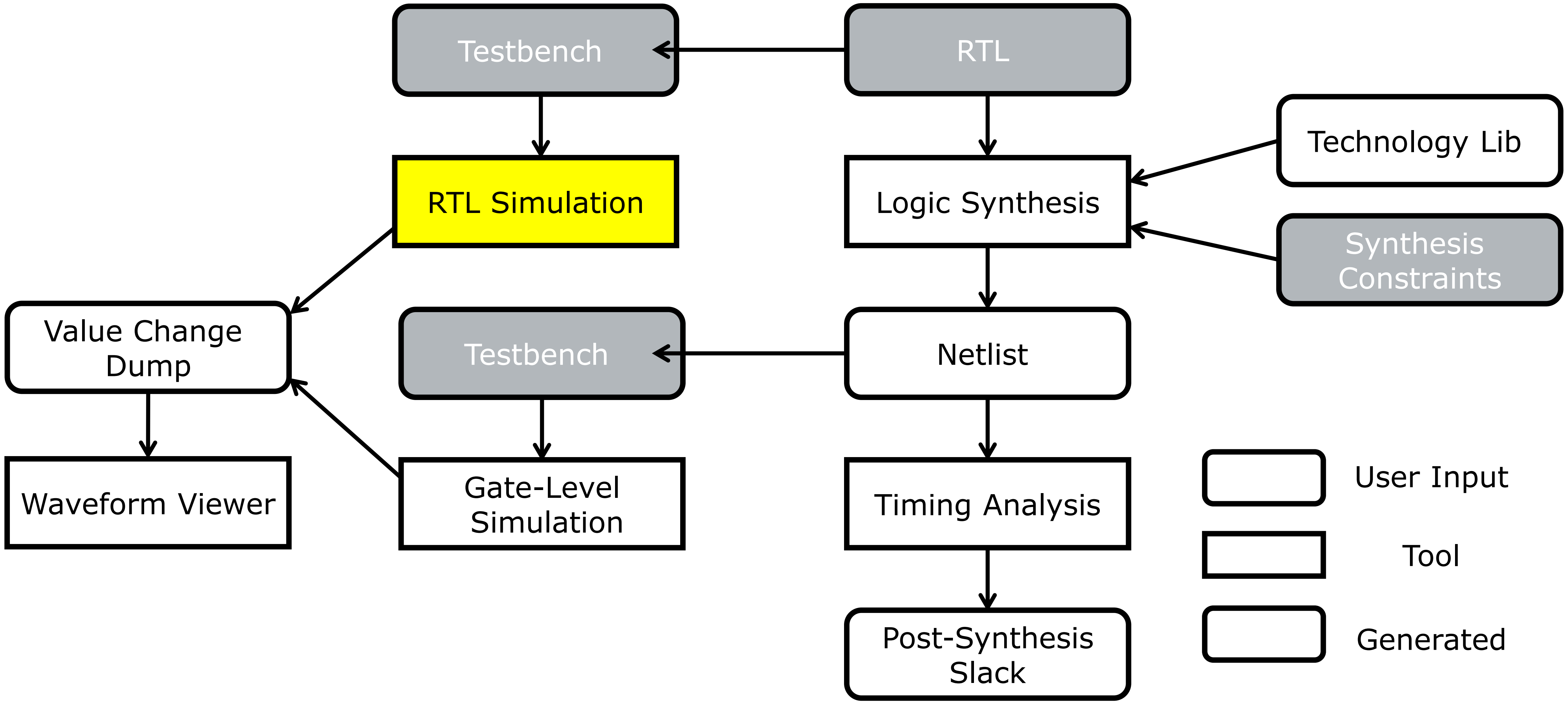
```
# base folder of the design
export FOLDER = lfsr
# toplevel module
export TOP = lfsr
# testbench in $(FOLDER)/sim, creates trace.vcd
export TB = tb.v
# standard cell library timing
export LIB = $(HOME)/skywater-pdk/libraries/sky130_fd_sc_hd/latest/timing/sky130_fd_sc_hd__tt_025C_1v80.lib
# simulation view of cells
export CELLS = $(HOME)/skywater-pdk/libraries/sky130_fd_sc_hd/latest/cells
# clock period in ns
export CLOCK = 5
# name of the clock net
export CLOCKNAME = clk
# input delay constraint, normally 0
export INPUTDELAY = 0
# output delay constraint, normally 0
export OUTPUTDELAY = 0
# synthesis script for logic optimization
export ABC = abc.area
# chip configuration
export CHIPCONFIG = config.mk
# don't touch
include scripts/make.design
```

Targets

```
# make -f Makefile.lfsr
Targets are:
   rtlsim        RTL simulation                          } Frontend
   synthesis     logic synthesis
   glschematic   creates a gate level schematic
   glsim         gate level simulation
   gltiming      static timing analysis
   openroad      Start OpenROAD docker container         } Backend
   chip          OpenROAD chip synthesis
   chipgui       OpenROAD GUI
   chipdata      Extract OpenROAD design data to work dir
   chipclean     Remove intermediate results from OpenROAD work area
   clean         Remove intermediate results from work dir
   distclean     Global clean crypto-asic-oss
```

LFSR chip: Building it

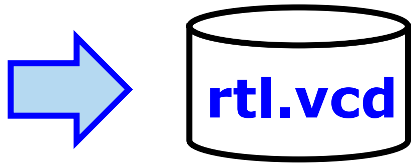
Front-end Design



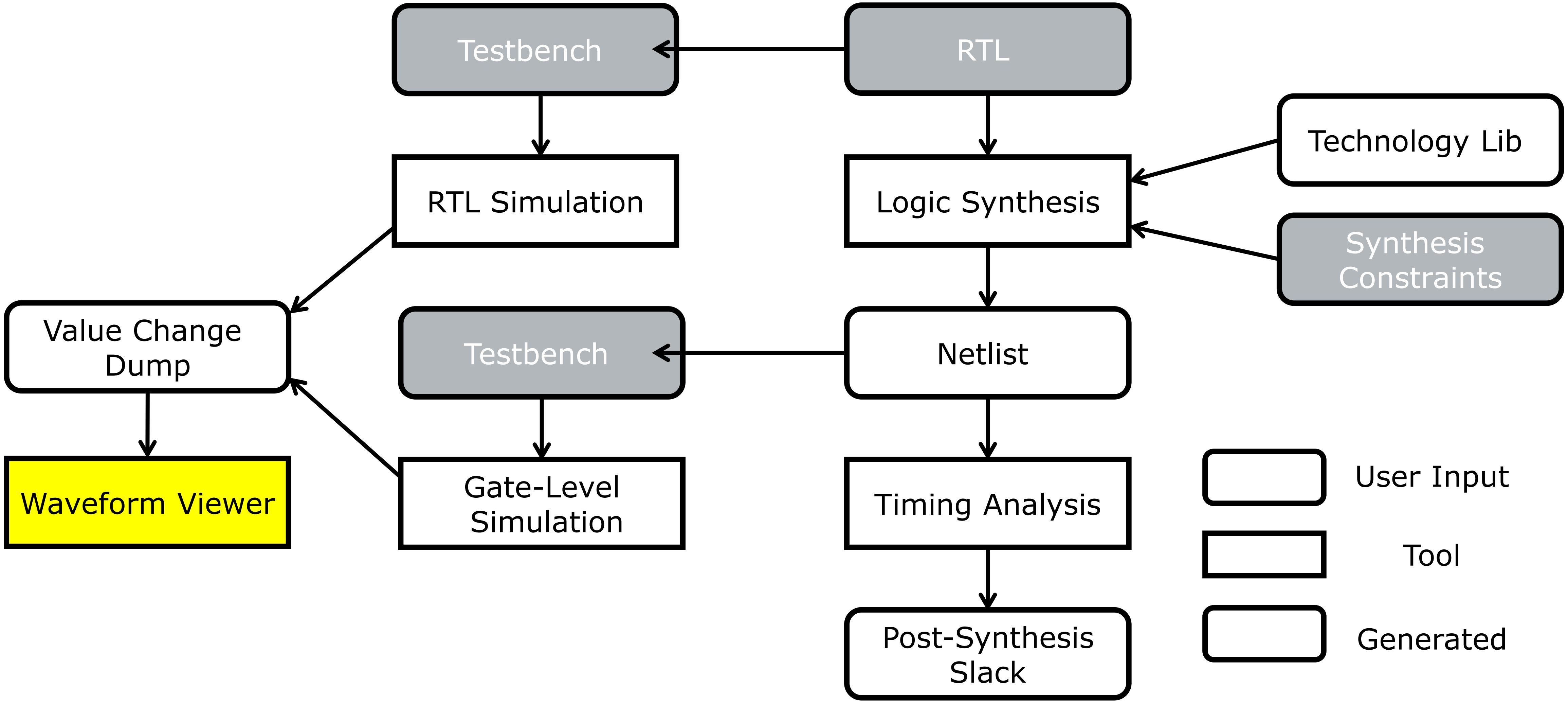
RTL Simulation

- Use RTL simulation to verify correctness of design

```
# make -f Makefile.lfsr rtlsim
test -d lfsr/work || mkdir -p lfsr/work
python3 scripts/gen_make_design.py
cd lfsr/work && make rtlsim
make[1]: Entering directory '/home/pschaumont/crypto-asic-opt/lfsr/work'
iverilog -y ../rtl ../sim/tb.v
./a.out && mv trace.vcd rtl.vcd && rm -f a.out
VCD info: dumpfile trace.vcd opened for output.
0001
1100
0110
0011
...
```



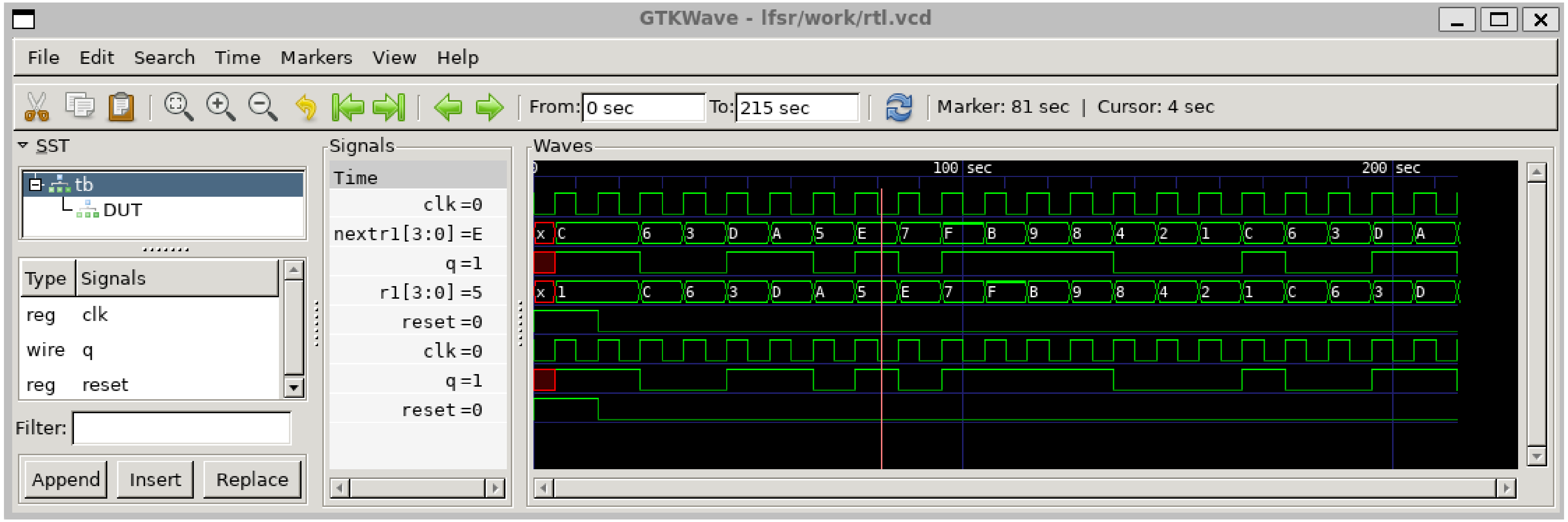
Front-end Design



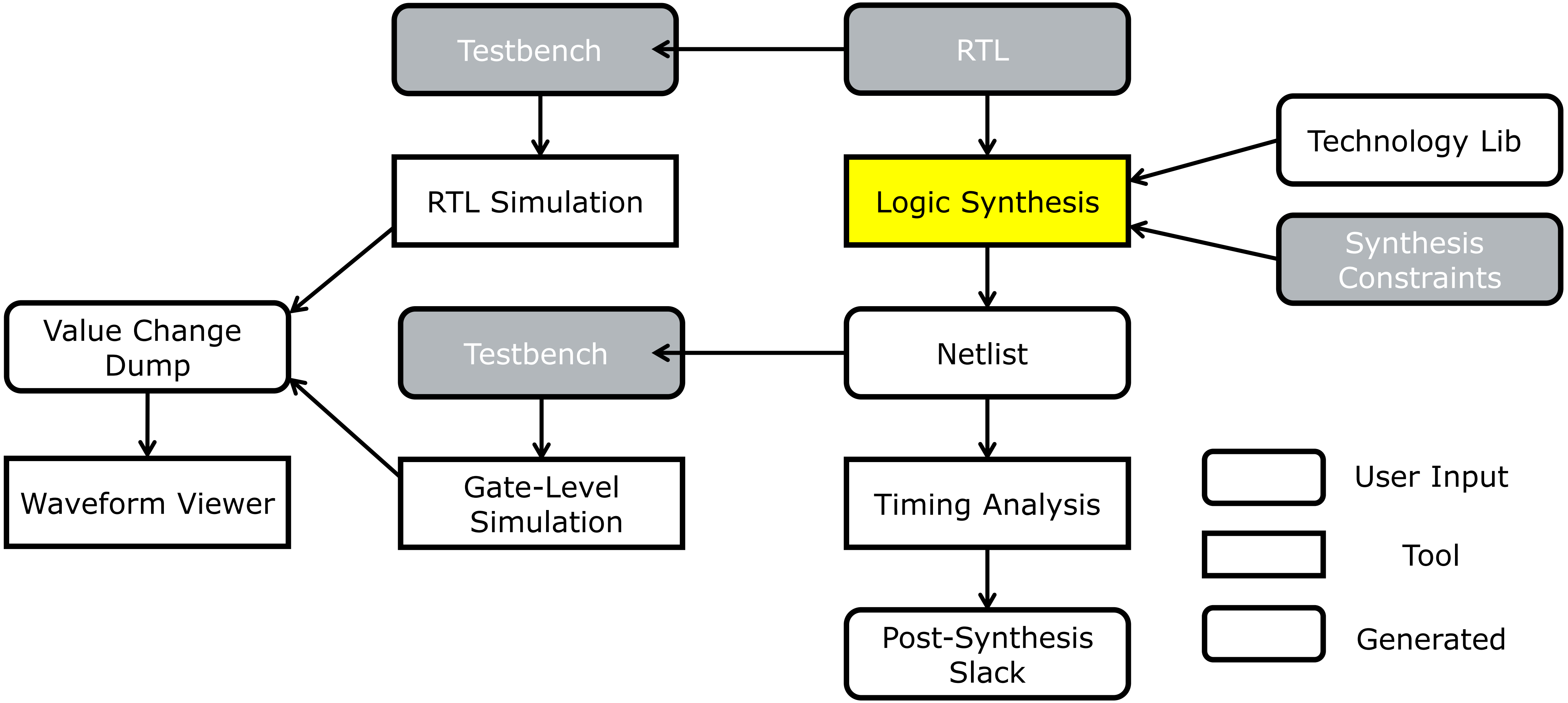
RTL Debugging

- Use VCD inspection to verify operation cycle by cycle

```
# gtkwave lfsr/work/rtl.vcd
```


GTKWave - lfsr/work/rtl.vcd
File Edit Search Time Markers View Help
From: 0 sec
To: 215 sec
Marker: 81 sec | Cursor: 4 sec
SST
tb
DUT
Type Signals
reg clk
wire q
reg reset
Filter:
Append Insert Replace
Signals
Time
clk=0
nextr1[3:0]=E
q=1
r1[3:0]=5
reset=0
clk=0
q=1
reset=0
Waves
100 sec
200 sec
x C 6 3 D A 5 E 7 F B 9 8 4 2 1 C 6 3 D A
x 1 C 6 3 D A 5 E 7 F B 9 8 4 2 1 C 6 3 D

Front-end Design



Logic Synthesis

- If RTL is correct, map design into a netlist

```
make -f Makefile.lfsr synthesis
cd lfsr/work && make synthesis
...
```

- Lots of output, most interesting output is at the end →
  - 15 standard cells:
    - 4 Flip Flop
    - 5 NOT, 1 ANDINV, 3 NAN, 2 NOR
  - Chip area = 125.12 square micron

```
=== lfsr ===

   Number of wires:                  13
   Number of wire bits:              22
   Number of public wires:            5
   Number of public wire bits:       11
   Number of memories:                0
   Number of memory bits:             0
   Number of processes:               0
   Number of cells:                  15
     sky130_fd_sc_hd__dfxtp_1         4
     sky130_fd_sc_hd__inv_1           5
     sky130_fd_sc_hd__lpflow_inputisolp_1       1
     sky130_fd_sc_hd__nand2_1         2
     sky130_fd_sc_hd__nand3_1         1
     sky130_fd_sc_hd__nor2_1          2

   Chip area for module '\lfsr': 125.120000
```

Logic Synthesis Output

- The target of logic synthesis is a library with prebuilt simple logic functions: a standard-cell library
- We use the Skywater 130nm standard cell library

```
...
# standard cell library timing
export LIB = $(HOME)/skywater-pdk/libraries/sky130_fd_sc_hd/latest/timing/sky130_fd_sc_hd__tt_025C_1v80.lib
# simulation view of cells
export CELLS = $(HOME)/skywater-pdk/libraries/sky130_fd_sc_hd/latest/cells
...
```

- A standard cell library contains hundreds of cells
  - Different logic functions, storage cells
  - Each cell comes in multiple versions
    - Different drive strength
    - Optimized for speed, low power, low area, ..
- The output of logic synthesis is a network of standard cells: a **netlist**

LFSR Netlist

```
module lfsr(clk, reset, q);
  (* src = "../rtl/lfsr.v:7" *)
  wire [3:0] _00_;
  wire _01_;
  wire _02_;
  wire _03_;
  wire _04_;
  wire _05_;
  wire _06_;
  wire _07_;
  (* src = "../rtl/lfsr.v:1" *)
  input clk;
  (* src = "../rtl/lfsr.v:5" *)
  (* unused_bits = "2" *)
  wire [3:0] nextr1;
  (* src = "../rtl/lfsr.v:3" *)
  output q;
  (* src = "../rtl/lfsr.v:4" *)
  wire [3:0] r1;
  (* src = "../rtl/lfsr.v:2" *)
  input reset;
```

```
sky130_fd_sc_hd__inv_1 _08_ (
    .A(nextr1[1]),
    .Y(_01_)
  );
  sky130_fd_sc_hd__nor2_1 _09_ (
    .A(reset),
    .B(_01_),
    .Y(_00_[1])
  );
  sky130_fd_sc_hd__inv_1 _10_ (
    .A(r1[3]),
    .Y(_02_)
  );
  sky130_fd_sc_hd__inv_1 _11_ (
    .A(q),
    .Y(_03_)
  );
  sky130_fd_sc_hd__nand2_1 _12_ (
    .A(_02_),
    .B(_03_),
    .Y(_04_)
  );
```

```
sky130_fd_sc_hd__inv_1 _13_ (
    .A(reset),
    .Y(_05_)
  );
  sky130_fd_sc_hd__nand2_1 _14_ (
    .A(r1[3]),
    .B(q),
    .Y(_06_)
  );
  sky130_fd_sc_hd__nand3_1 _15_ (
    .A(_04_),
    .B(_05_),
    .C(_06_),
    .Y(_07_)
  );
  sky130_fd_sc_hd__inv_1 _16_ (
    .A(_07_),
    .Y(_00_[2])
  );
.. // lines skipped
endmodule
```

LFSR Netlist

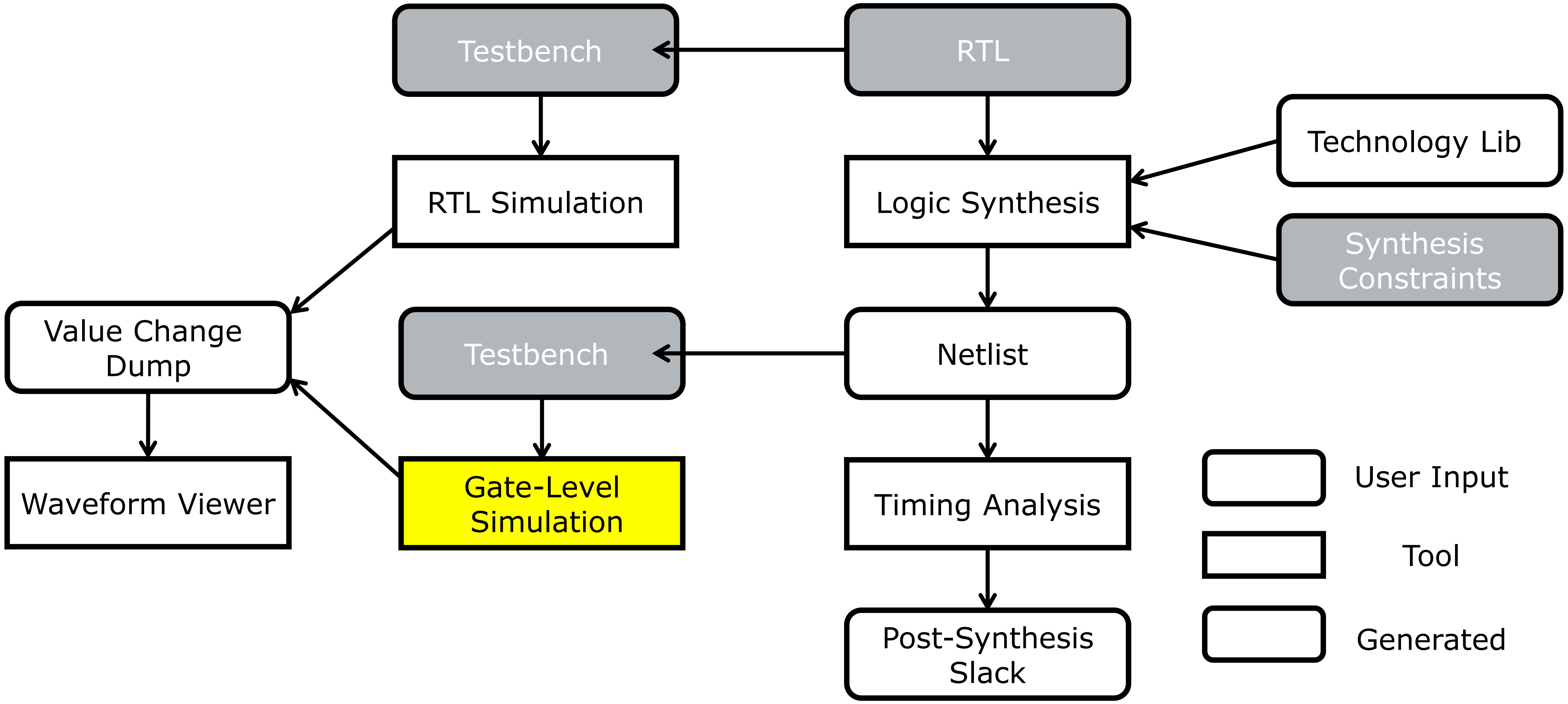
```
module lfsr(clk, reset, q);
  (* src = "../rtl/lfsr.v:7" *)
  wire [3:0] _00_;
  wire _01_;
  wire _02_;
  wire _03_;
  wire _04_;
  wir
  wir
  wir
  (*
  inp
  (* src = "../rtl/lfsr.v:5" *)
  (* unused_bits = "2" *)
  wire [3:0] nextr1;
  (* src = "../rtl/lfsr.v:3" *)
  output q;
  (* src = "../rtl/lfsr.v:4" *)
  wire [3:0] r1;
  (* src = "../rtl/lfsr.v:2" *)
  input reset;
```

```
sky130_fd_sc_hd__inv_1 _08_ (
    .A(nextr1[1]),
    .Y(_01_)
  );
  sky130_fd_sc_hd__nor2_1 _09_ (
    .A(reset),
    .B(_01_),

  );
  sky130_fd_sc_hd__inv_1 _11_ (
    .A(q),
    .Y(_03_)
  );
  sky130_fd_sc_hd__nand2_1 _12_ (
    .A(_02_),
    .B(_03_),
    .Y(_04_)
  );
```

```
sky130_fd_sc_hd__inv_1 _13_ (
    .A(reset),
    .Y(_05_)
  );
  sky130_fd_sc_hd__nand2_1 _14_ (
    .A(r1[3]),
    .B(q),
                                    _ (

    .C(_06_),
    .Y(_07_)
  );
  sky130_fd_sc_hd__inv_1 _16_ (
    .A(_07_),
    .Y(_00_[2])
  );
.. // lines skipped
endmodule
```

1. Netlists are generally detailed and obfuscated (full of generated symbols)
2. Only module inputs, module outputs, and registers keep their RTL name
3. To verify the correctness of a netlist, you typically simulate the netlist with a functional view of the standard cells

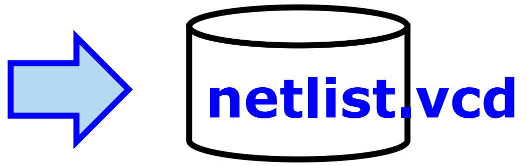
Front-end Design



Logic Simulation

- Use Logic Simulation to run the RTL testbench on the netlist

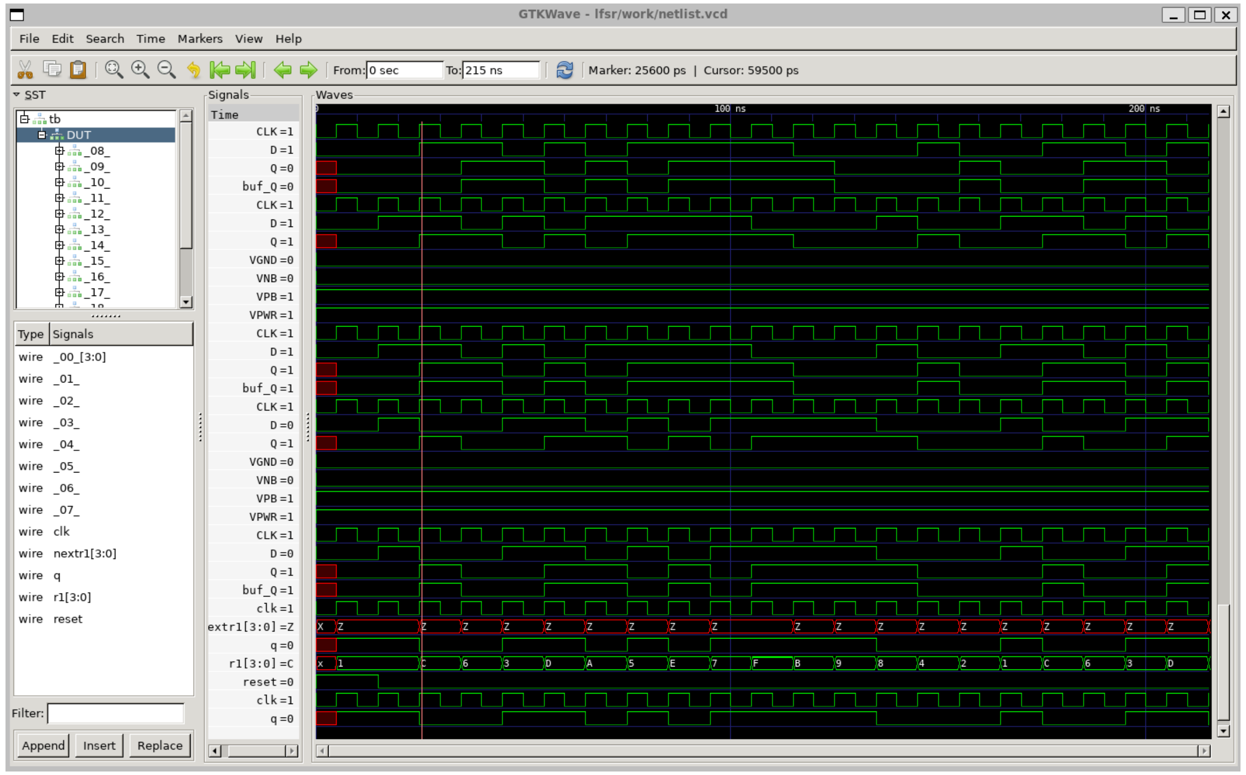
```
# make -f Makefile.lfsr glsim
cd lfsr/work && make glsim
make[1]: Entering directory '/home/pschaumont/crypto-asic-opt/lfsr/work'
iverilog -DFUNCTIONAL -c lib.cmd netlist.v ../sim/tb.v
./a.out && mv trace.vcd netlist.vcd && rm -f a.out
VCD info: dumpfile trace.vcd opened for output.
0001
1100
0110
0011
...
```



Netlist Debugging

- Use VCD inspection to verify operation cycle by cycle

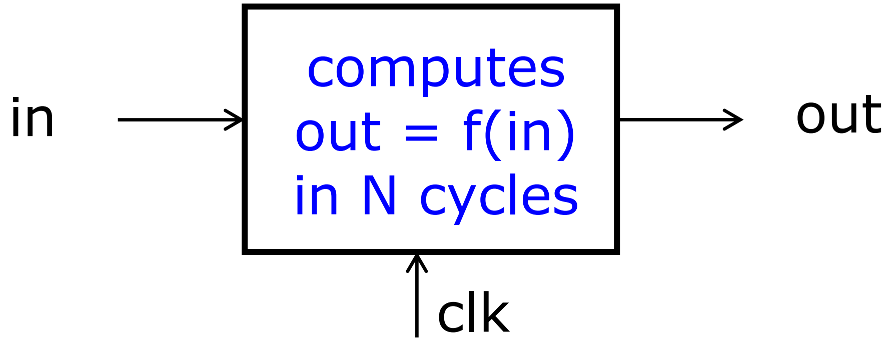
```
# gtkwave lfsr/work/netlist.vcd
```



1. Generally much harder than RTL debugging, but it can flag detailed problems such as issues with reset
2. With proper configuration logic-level VCD can reflect actual timing properties of individual wires (transition times, glitches, ..)

Design Timing

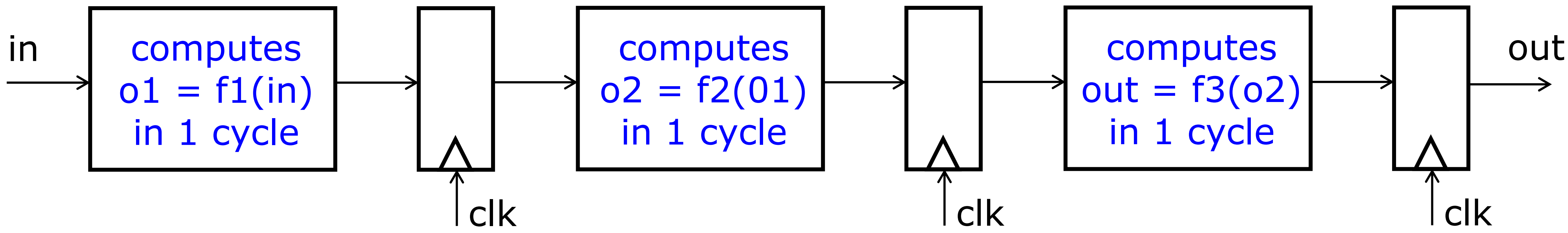
1. Performance has two factors: Cycle Budget N and Clock Frequency f_{clk}
2. Maximizing performance means minimizing N and maximizing f_{clk}



- Latency = T_{clk} x N [seconds to compute out = f(in)]
- Throughput = f_{clk} / N [outputs per second]

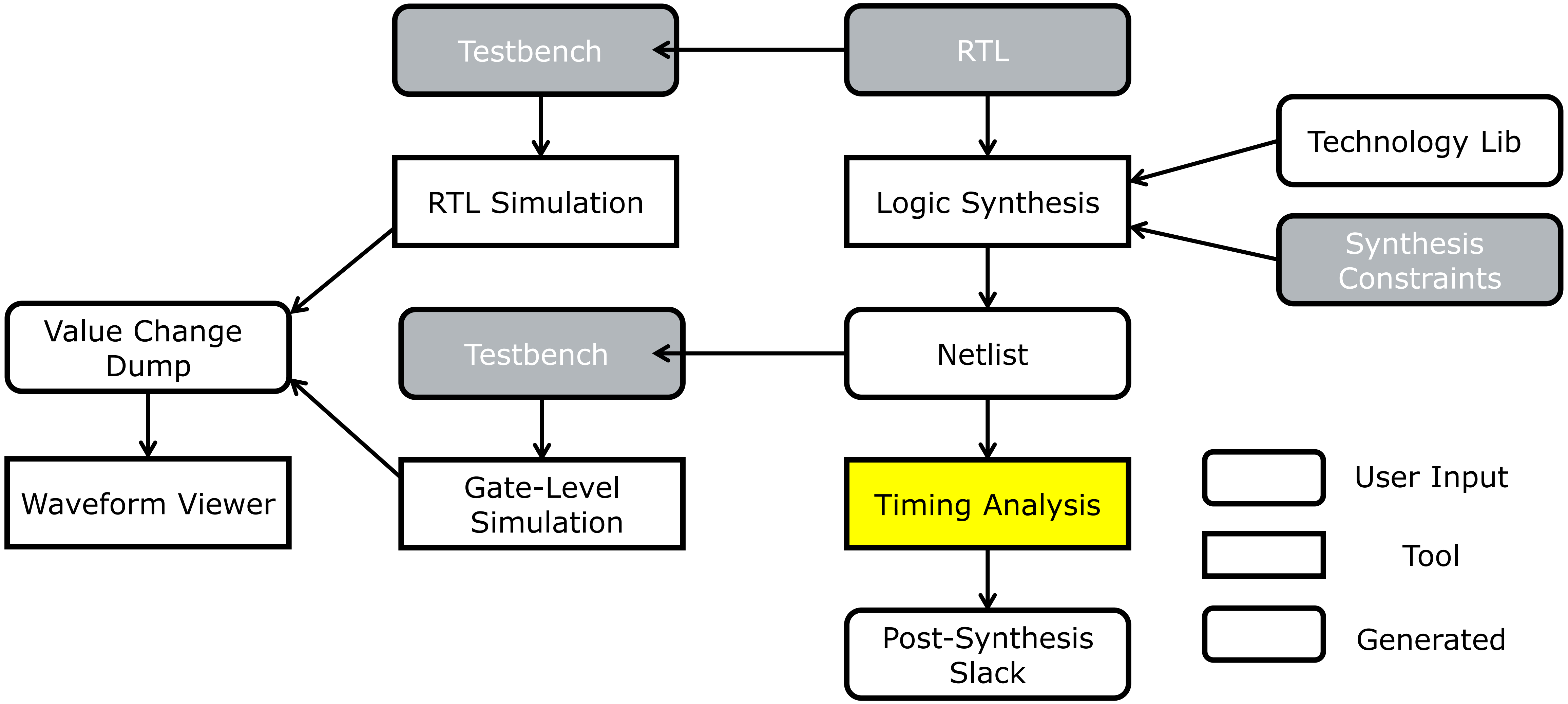
Design Timing

1. Performance has two factors: Cycle Budget N and Clock Frequency f_{clk}
2. Maximizing performance means minimizing N and maximizing f_{clk}
3. Throughput can be optimized in a pipeline, but at the expense of latency



- Latency = T_{clk} x N [seconds to compute out = f(in)]
- Throughput = f_{clk} [outputs per second]

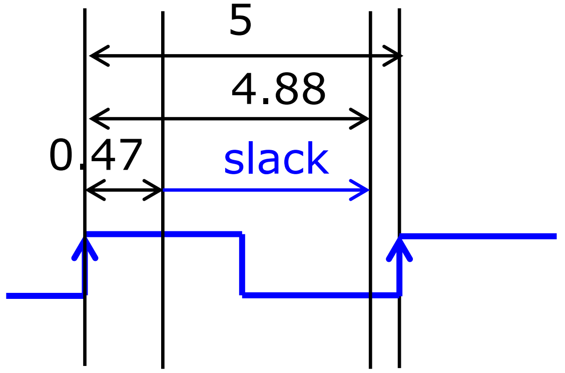
Front-end Design



Design Timing

- The cycle budget N is defined by the RTL design
- The $f_{clk,max}$ is computed with *static timing analysis* (STA)[1]

```
# make -f Makefile.lfsr gltiming
cd lfsr/work && make gltiming
make[1]: Entering directory '/home/pschaumont/crypto-asic-opt/lfsr/work'
sta <sta.cmd
...

           4.88   data required time
          -0.47   data arrival time
----------------------------------------
           4.41   slack (MET)
```



[1] See: https://summerschool-croatia.cs.ru.nl/2023/slides/Schaumont_crypto-asic-oss-prs.pdf

Building the chip layout

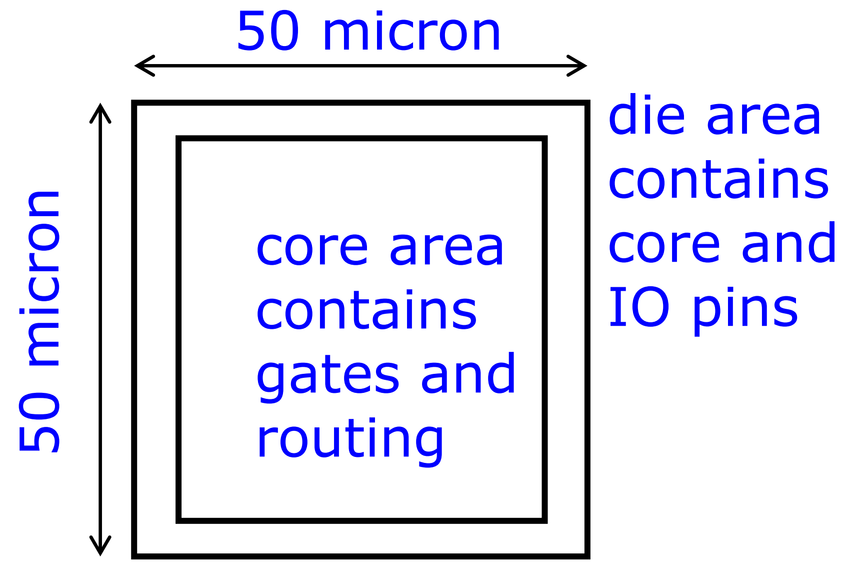
- config.mk

```
export DESIGN_NAME = lfsr
export PLATFORM    = sky130hd

export VERILOG_FILES = ./crypto-asic-opt/lfsr/rtl/lfsr.v
export SDC_FILE      = ./crypto-asic-opt/lfsr/work/constraint.sdc

export DIE_AREA    =   0   0   50   50
export CORE_AREA   =   5   5   45   45
```

Required size depends on number of cells in netlist

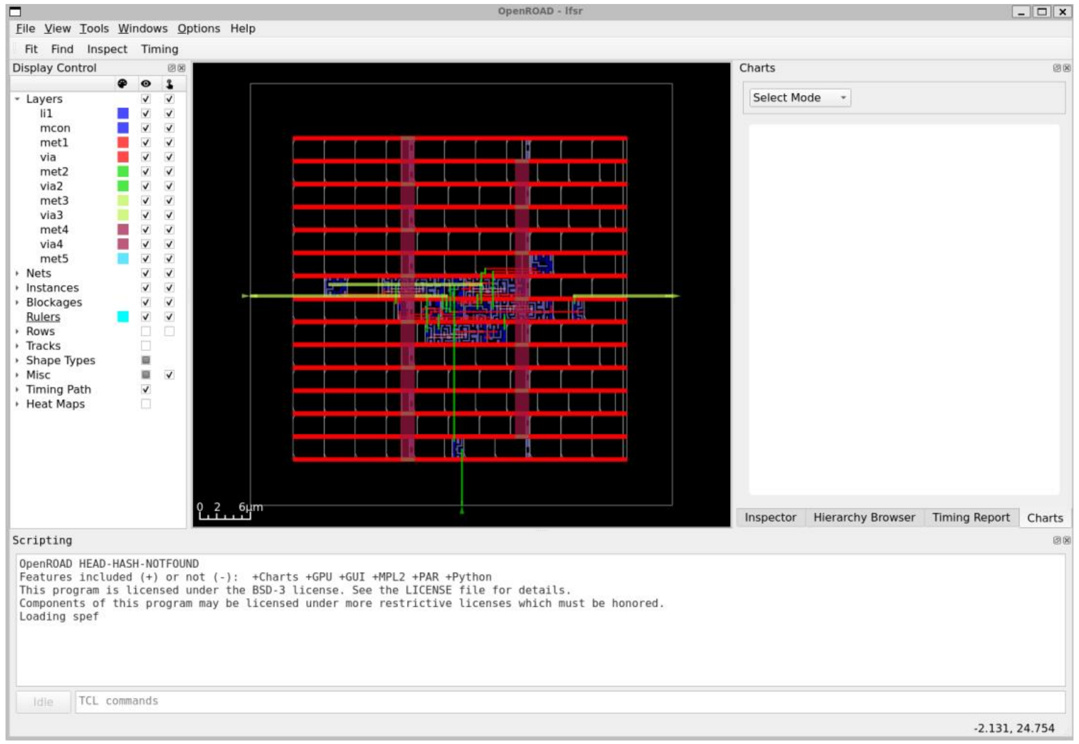
Utilization = standard_cell_area / core_area
Typically around 70% in 130nm

If utilization >100%, the P&R throws an error


50 micron
50 micron
core area contains gates and routing
die area contains core and IO pins

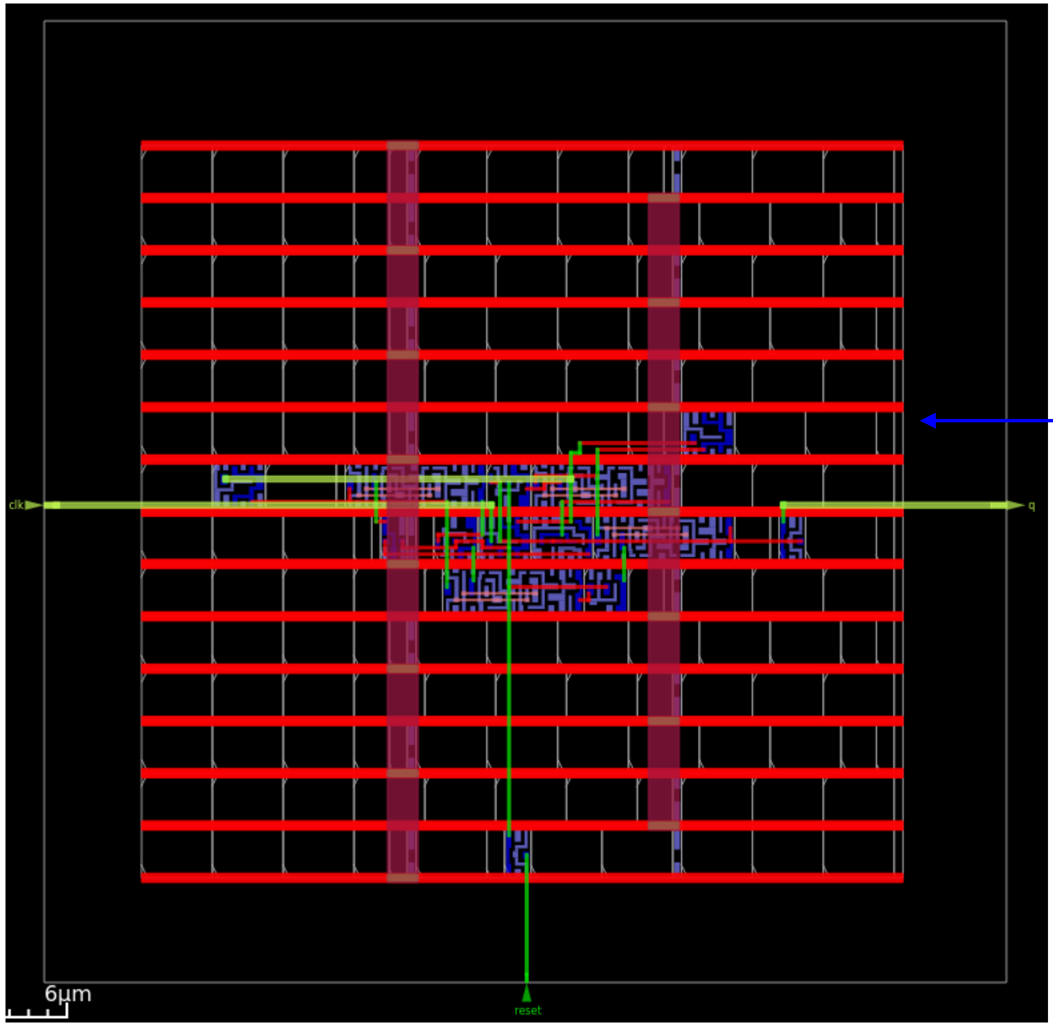
Building the chip layout

```
# make -f Makefile.lfsr openroad
openroad> make -f Makefile.lfsr chip
...
openroad> make -f Makefile.lfsr chipgui
```



```
OpenROAD HEAD-HASH-NOTFOUND
Features included (+) or not (-):  +Charts +GPU +GUI +MPL2 +PAR +Python
This program is licensed under the BSD-3 license. See the LICENSE file for details.
Components of this program may be licensed under more restrictive licenses which must be honored.
Loading spef
```

Default chip view



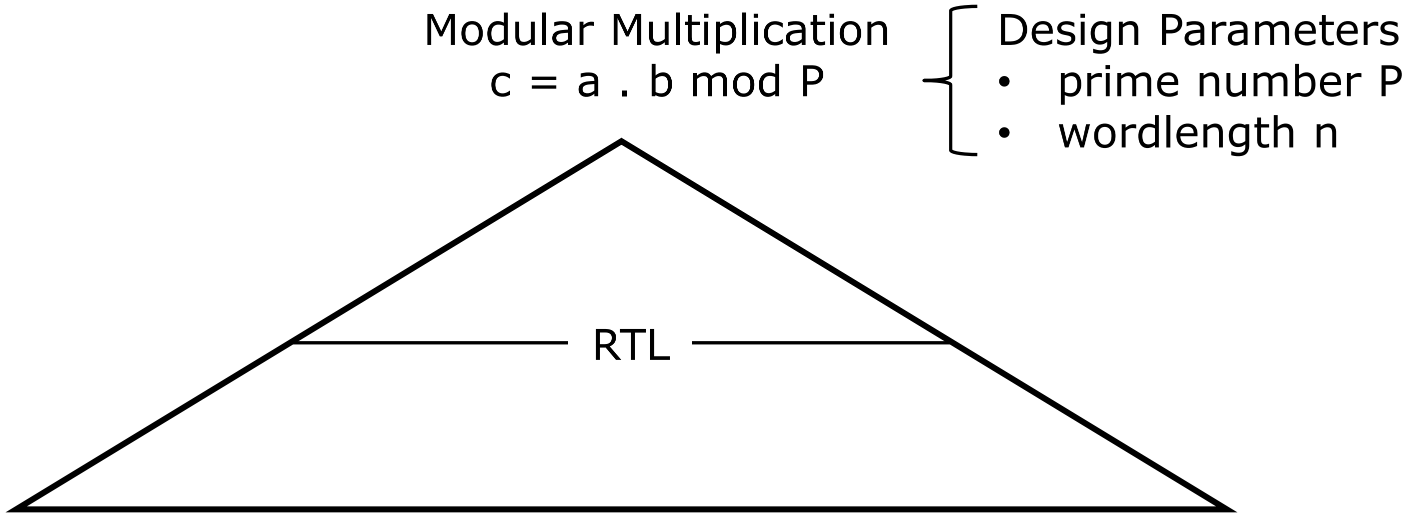
Actual Chip Area:
45 x 45 = 2025 sqmu

Collecting the chip data

```
openroad> make -f Makefile.lfsr chipdata
(ctrl-d to exit docker)
# ls lfsr/work/results/base/
1_1_yosys.v                2_floorplan.sdc            4_cts.sdc          6_final.gds
1_synth.sdc                3_1_place_gp_skip_io.odb   5_1_grt.odb        6_final.odb
1_synth.v                  3_2_place_iop.odb          5_2_fillcell.odb   6_final.sdc
2_1_floorplan.odb          3_3_place_gp.odb           5_3_route.odb      6_final.spef
2_2_floorplan_io.odb       3_4_place_resized.odb      5_route.odb        6_final.v
2_3_floorplan_tdms.odb     3_5_place_dp.odb           5_route.sdc        clock_period.txt
2_4_floorplan_macro.odb    3_place.odb                6_1_fill.odb       mem.json
2_5_floorplan_tapcell.odb  3_place.sdc                6_1_fill.sdc       route.guide
2_6_floorplan_pdn.odb      4_1_cts.odb                6_1_merged.gds     updated_clks.sdc
2_floorplan.odb            4_cts.odb                  6_final.def
# ls lfsr/work/reports/base/
2_floorplan_final.rpt    5_global_route_post_repair_design.rpt   cts_clk.webp
3_detailed_place.rpt     5_global_route_post_repair_timing.rpt   final_clocks.webp
3_resizer.rpt            5_global_route_pre_repair_design.rpt    final_ir_drop.webp
3_resizer_pre.rpt        5_route_drc.rpt                         final_placement.webp
4_cts_final.rpt          6_finish.rpt                            final_resizer.webp
4_cts_post-repair.rpt    VDD.rpt                                 final_routing.webp
4_cts_pre-repair.rpt     VSS.rpt                                 synth_check.txt
5_global_place.rpt       antenna.log                             synth_stat.txt
5_global_route.rpt       congestion.rpt
```

Modular Multiplication

Design Space Exploration



1/n cycles
maximum throughput

pipelined implementation

`modmul_bp_pipe`

1 cycle
fully parallel design

bitparallel implementation

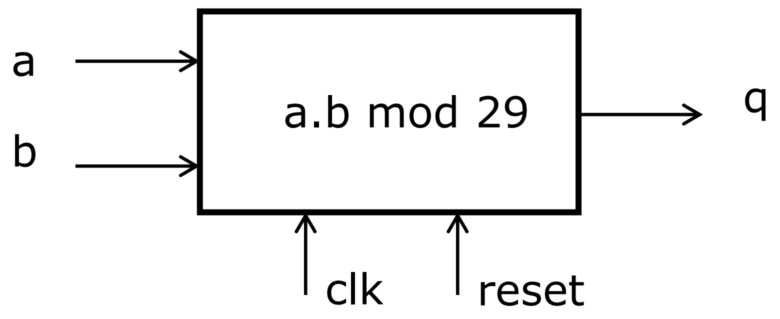
`modmul_bp`

n cycles
fully sequential design

bitserial implementation

`modmul_bs`

Toy Design: a.b mod 29



- a, b and q are 5 bit
- P = 29 = $2^5 - 3$
  - There are many other primes of this form $2^n - 3$
    eg for n = 10, 12, 14, 20, 24, .., 122, 150, 174, ...
  - The format $2^n - k$ leads to efficient implementation of mod P for small k

Efficient computation of mod 29

- Given an m-bit (>5 bit) number V

$$V = V0 + V1 \,.\, 2^5$$

  with $0 <= V0 < 32$

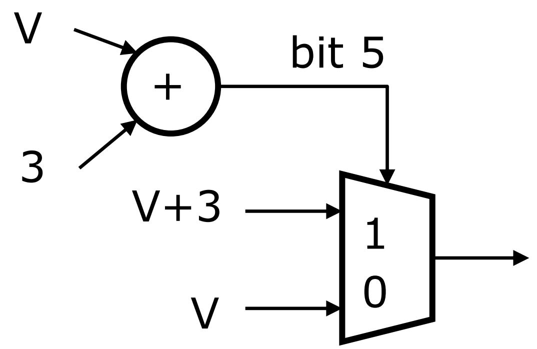
- Then V mod 29 = V mod (2^5 – 3) is computed as

$$\begin{aligned} V \bmod 29 &= (V0 + V1.(2^5 - 3 + 3)) \bmod 29 \\ &= (V0 + 3.V1) \bmod 29 \\ &= V_r \bmod 29 \end{aligned}$$

- This operation is repeated until $V_r < 32$
  - In practice, two iterations are sufficient, since WL(3A) <= WL(A)+2

Efficient computation of mod 29 (2)

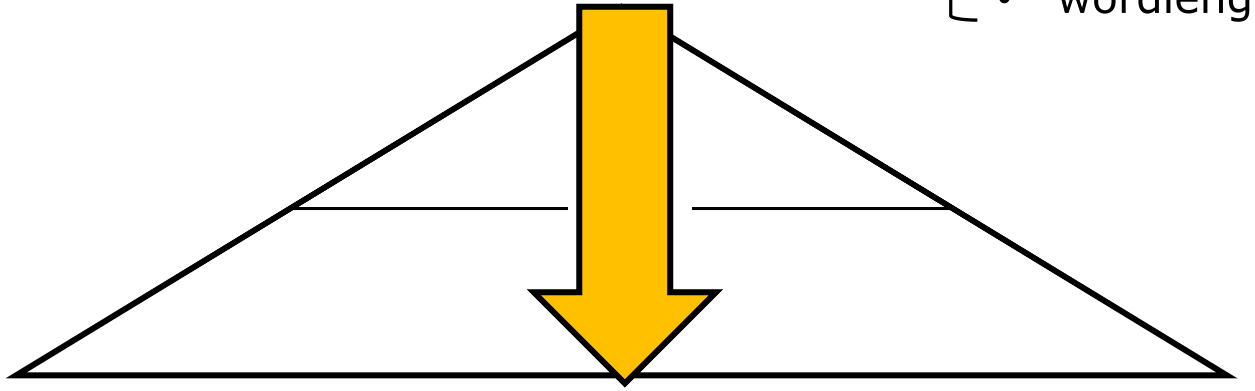
- Given an m-bit (<=5 bit) number V
- Then V mod 29
  = V when V < 29
  = V – 29 when V = 29, 30, 31
- Implement by computing V + 3 and test carry on bit 5



Design Space Exploration

Modular Multiplication
c = a . b mod P

Design Parameters
- prime number P
- wordlength n



1/n cycles
maximum throughput

pipelined implementation

`modmul_bp_pipe`

1 cycle
fully parallel design

bitparallel implementation

`modmul_bp`

n cycles
fully sequential design

bitserial implementation

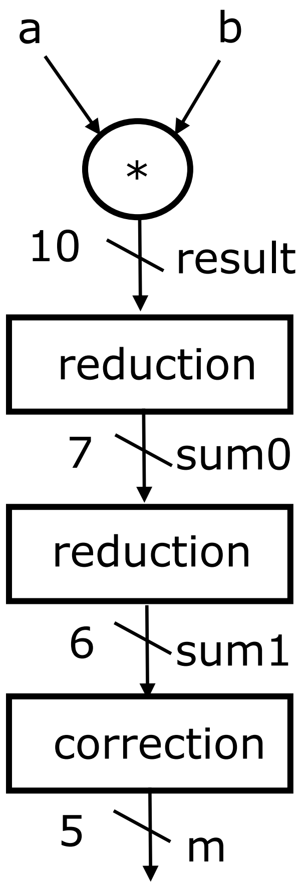
`modmul_bs`

Standard Bitparallel a * b mod 29

```
module modmul(input         reset,
              input         clk,
              input [4:0]   a,
              input [4:0]   b,
              output [4:0] m);
   // compute (a*b) mod 29
   wire [9:0]               result;
   wire [6:0]               sum0;
   wire [5:0]               sum1;

   assign result = a * b;
   assign sum0 = result[4:0] + result[9:5] + {result[9:5], 1'b0};
   assign sum1 = sum0[4:0] + sum0[6:5] + {sum0[6:5], 1'b0};
   assign m    = ((sum1 - 5'd29) < sum1) ? (sum1 - 5'd29) : sum1;

endmodule
```

Standard Bitparallel a * b mod 29

```
module modmul(input         reset,
              input         clk,
              input  [4:0]  a,
              input  [4:0]  b,
              output [4:0]  m);
   // compute (a*b) mod 29
   wire [9:0]               result;
   wire [6:0]               sum0;
   wire [5:0]               sum1;

   assign result = a * b;
   assign sum0 = result[4:0] + result[9:5] + {result[9:5], 1'b0};
   assign sum1 = sum0[4:0] + sum0[6:5] + {sum0[6:5], 1'b0};
   assign m    = ((sum1 - 5'd29) < sum1) ? (sum1 - 5'd29) : sum1;

endmodule
```



Standard Bitparallel a * b mod 29

```
module modmul(input          reset,
              input          clk,
              input  [4:0]   a,
              input  [4:0]   b,
              output  [4:0]  m);
   // compute (a*b) mod 29
   wire [9:0]                result;
   wire [6:0]                sum0;
   wire [5:0]                sum1;

   assign result = a * b;
   assign sum0 = result[4:0] + result[9:5] + {result[9:5], 1'b0};
   assign sum1 = sum0[4:0] + sum0[6:5] + {sum0[6:5], 1'b0};
   assign m    = ((sum1 - 5'd29) < sum1) ? (sum1 - 5'd29) : sum1;

endmodule
```

Testing Standard Bitparallel implementation

```
module tb;
  reg [4:0] a;
  reg [4:0] b;
  wire [4:0] m;
  modmul dut(.reset(),
             .clk(),
             .a(a),
             .b(b),
             .m(m));
initial
  begin
  $dumpfile("trace.vcd");
  $dumpvars(0, tb);
  a = 5'b0;
  b = 5'b0;
  while (1)
    begin
    #10;
    $display("%x %x %x", a, b, dut.m);
    if (dut.m != ((a * b) % 29))
      $error;
    a = a + 5’b1;
    if (a == 5’b0)
      b = b + 5’b1;
    if ((a == 5'b0) && (b == 5'b0))
      $finish;
    end
   end
endmodule
```

Worcester Polytechnic Institute

Simulation

```
# make -f Makefile.modmul_bp rtlsim
...
VCD info: dumpfile trace.vcd opened for output.
00 00 00
01 00 00
02 00 00
03 00 00
04 00 00
...
05 03 0f
06 03 12
07 03 15
08 03 18
09 03 1b
0a 03 01
0b 03 04
0c 03 07
...
```

0xb * 0x3 mod 29
= 11 * 3 mod29
= 33 mod 29
= 4
= 0x4

Logic Synthesis

```
# make -f Makefile.modmul_bp synthesis
...
10. Printing statistics.

=== modmul ===

   Number of wires:                347
   Number of wire bits:            359
   Number of public wires:           5
   Number of public wire bits:      17
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                347
     sky130_fd_sc_hd__a21boi_1       2
     sky130_fd_sc_hd__a21boi_2       1
     sky130_fd_sc_hd__a21oi_1        2
     sky130_fd_sc_hd__a21oi_2        3
     sky130_fd_sc_hd__a22o_1         1
...
 Chip area for module '\modmul': 1850.524800
...
```

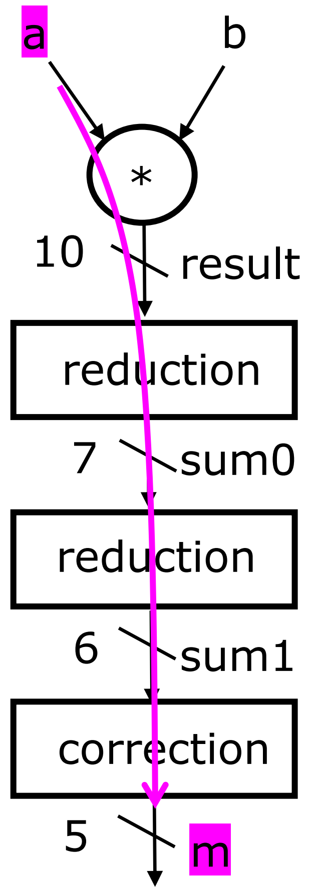
1 cycle -> 1850.5248 sqmicron

Timing Analysis

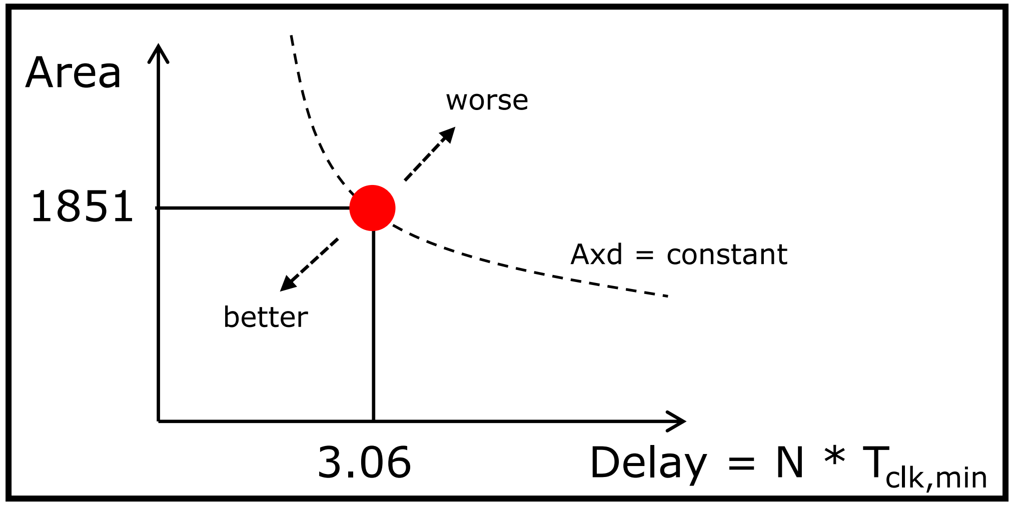
```
# make -f Makefile.modmul_bp gltiming
...
Startpoint: a[2] (input port clocked by clk)
Endpoint: m[0] (output port clocked by clk)
..
           5.00   data required time
          -3.06   data arrival time
--------------------------------------------------
           1.94   slack (MET)
```

1 cycle -> 1850.5248 sqmicron

Min clock period = 3.06ns

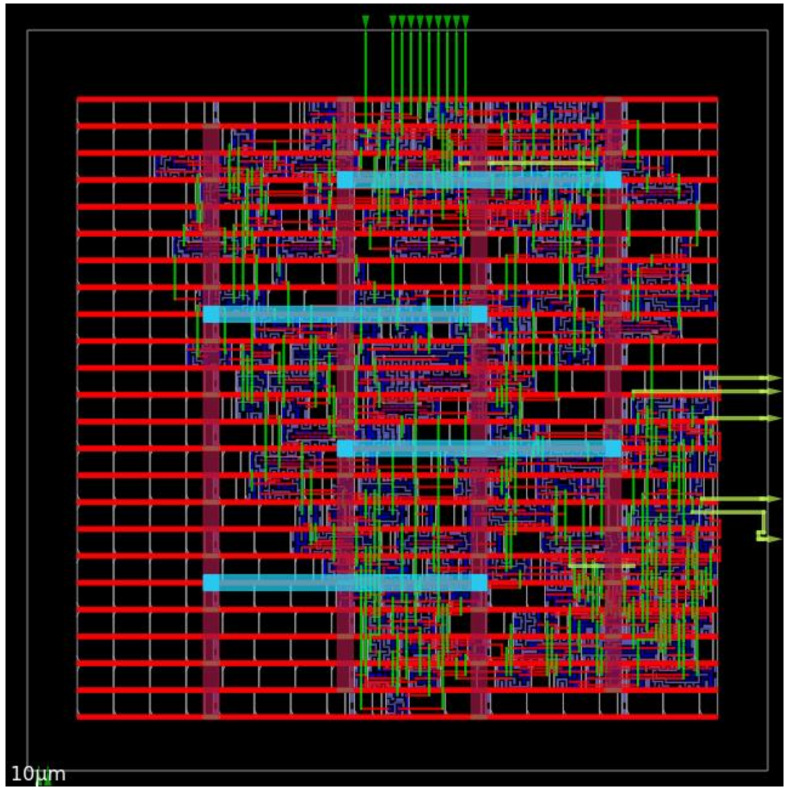


Design space (so far):



Chip Design

```
# make -f Makefile.modmul_bp openroad
openroad> make -f Makefile.modmul_bp chip
openroad> make -f Makefile.modmul_bp chipgui
openroad> make -f Makefile.modmul_bp chipdata
```



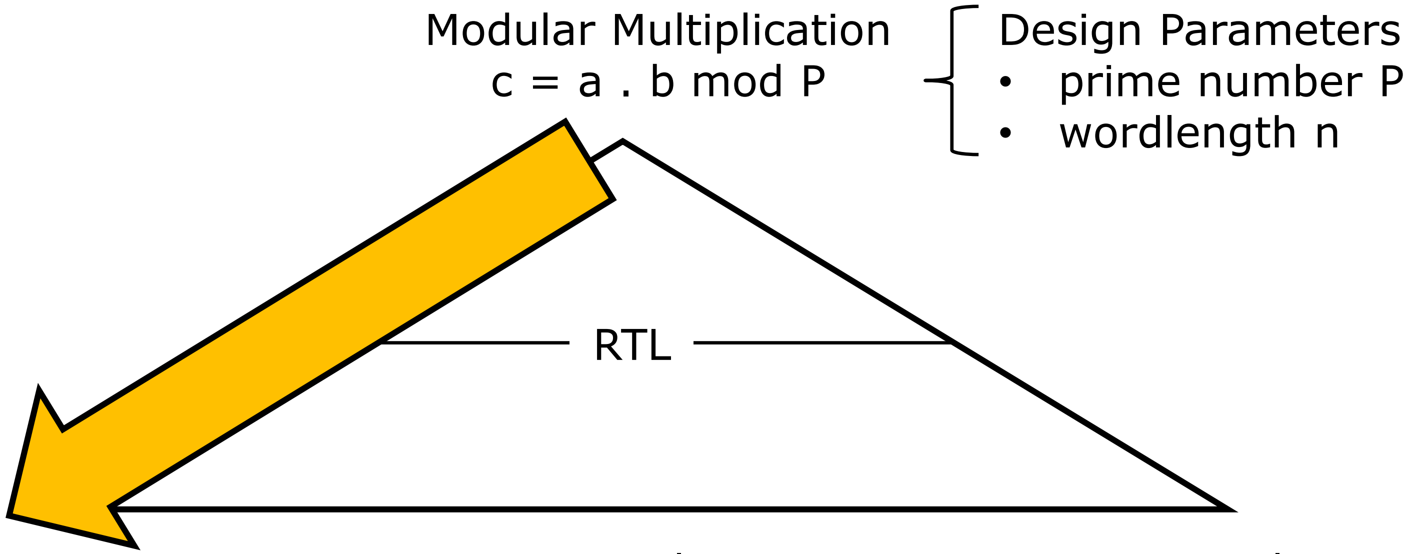
65 x 65 core area = 4225 sqmicron

Post-layout timing: hw_modmul/work/reports/base/5_global_route_post_repair_timing.rpt

```
global route post repair timing report_checks -path_delay max
--------------------------------------------------------------------------
Startpoint: a[2] (input port clocked by clk)
Endpoint: m[3] (output port clocked by clk)
...
     5.00   data required time
    -6.92   data arrival time
-----------------------------------------------------
    -1.92   slack (VIOLATED)
```

Thus, after P&R, area 2.28x and delay 2.26x
Physical overhead is significant

Design Space Exploration



1/n cycles
maximum throughput

pipelined implementation

`modmul_bp_pipe`

1 cycle
fully parallel design

bitparallel implementation

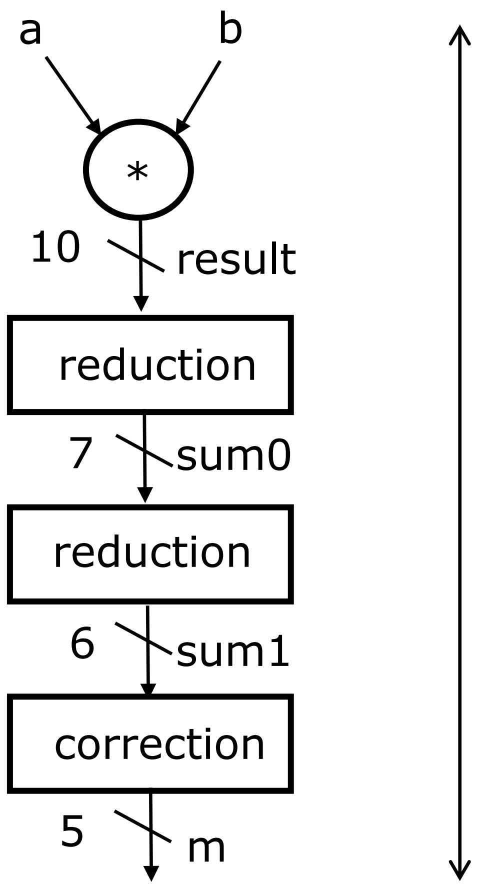
`modmul_bp`

n cycles
fully sequential design

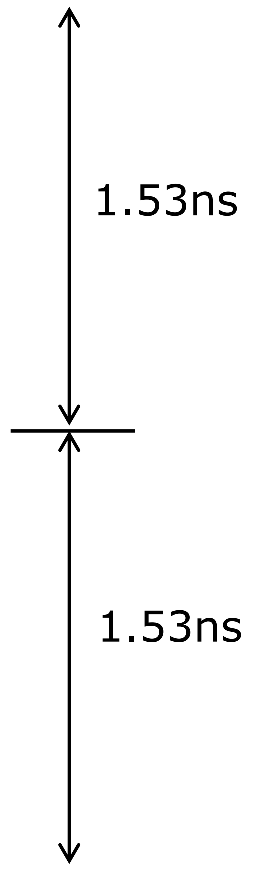
bitserial implementation

`modmul_bs`

Pipelined Design



Ideally,



(or any equal-sized partitioning you'd like to target)

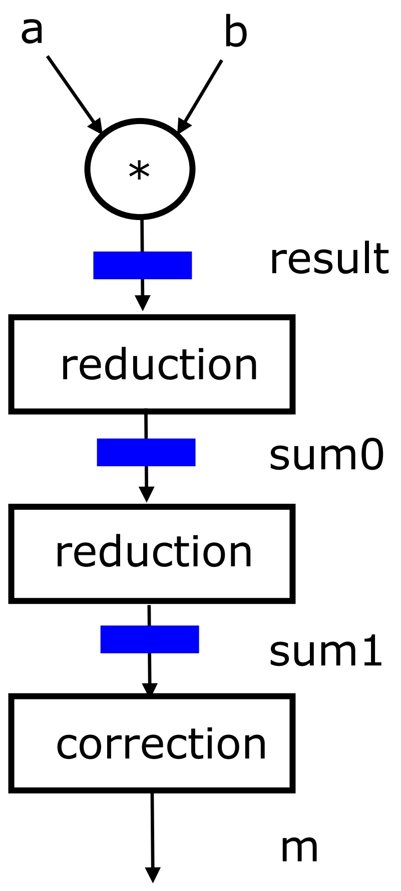
In practice:

```
module modmul(input         reset,
              input         clk,
              input [4:0]   a,
              input [4:0]   b,
              output [4:0]  m);
   // compute (a*b) mod 29
   wire [9:0]               result;
   wire [6:0]               sum0;
   wire [5:0]               sum1;

   assign result = a * b;
   assign sum0   = result[4:0] + result[9:5] + {result[9:5], 1'b0};
   assign sum1   = sum0[4:0] + sum0[6:5] + {sum0[6:5], 1'b0};
   assign m      = ((sum1 - 5'd29) < sum1) ? (sum1 - 5'd29) : sum1;

endmodule
```

- *Inserting a pipeline register involves rewriting the RTL. Thus, a pipeline register is located at an RTL signal*
- *Pick the RTL signal closest to the desired timing boundary (which is not easy since only registers and IO names are preserved in the netlist)*

Pipelined Design

```
module modmul(input        reset,
              input        clk,
              input [4:0]  a,
              input [4:0]  b,
              output [4:0] m);

   wire [9:0]              result;
   reg [9:0]               resultreg;
   wire [6:0]              sum0;
   reg [6:0]               sum0reg;
   wire [5:0]              sum1;
   reg [5:0]               sum1reg;

   always @(posedge clk)
     if (reset)
       begin
          resultreg <= 10'b0;
          sum0reg <= 7'b0;
          sum1reg <= 6'b0;
       end
     else
       begin
          resultreg <= result;
          sum0reg <= sum0;
          sum1reg <= sum1;
       end

   assign result = a * b;
   assign sum0 = resultreg[4:0] + resultreg[9:5] + {resultreg[9:5], 1'b0};
   assign sum1 = sum0reg[4:0] + sum0reg[6:5] + {sum0reg[6:5], 1'b0};
   assign m    = ((sum1reg - 5'd29) < sum1reg) ? (sum1reg - 5'd29) : sum1reg;

endmodule
```



In principle, registers must be inserted at *every* input, or at *every* output. A few registers 'in the middle', as in this example, is unconventional

Pipelined Design: RTL Simulation

```
module tb;
   reg [4:0]  a;
   reg [4:0]  b;
   wire [4:0] m;
   reg        reset;
   reg        clk;

   modmul dut(.reset(reset),
              .clk(clk),
              .a(a),
              .b(b),
              .m(m));

   always
     begin
        clk = 1'b0;
        #5;
        clk = 1'b1;
        #5;
     end

   initial
     begin
        $dumpfile("trace.vcd");
        $dumpvars(0, tb);

        a = 5'b0;
        b = 5'b0;
        reset = 1'b1;

        repeat (2)
          @(posedge clk);
```

```
reset = 1'b0;

        while (1)
          begin
             repeat (3)
               @(posedge clk);

             #1;
             $display("%x %x %x", a, b, dut.m);

             if (dut.m != ((a * b) % 29))
               $error;

             a = a + 5'b1;
             if (a == 5'b0)
               b = b + 5'b1;

             if ((a == 5'b0) && (b == 5'b0))
               $finish;
          end

     end
endmodule
```

Logic Synthesis

```
# make -f Makefile.modmul_bp_pipe synthesis
...
10. Printing statistics.

=== modmul ===

   Number of wires:                309
   Number of wire bits:            361
   Number of public wires:           8
   Number of public wire bits:      40
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                349
...
     sky130_fd_sc_hd__and2_2         1
     sky130_fd_sc_hd__buf_2          1
     sky130_fd_sc_hd__dfxtp_1       23
     sky130_fd_sc_hd__inv_1         52
     sky130_fd_sc_hd__inv_2          2
...
 Chip area for module '\modmul': 2001.920000
...
```

3 cycles -> 2001.92 sqmicron

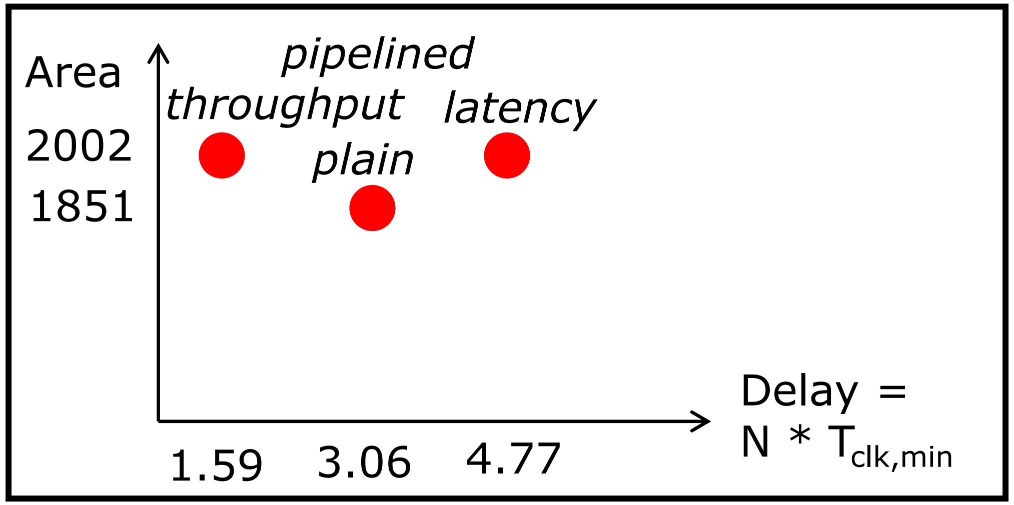
Timing Analysis

```
# make -f Makefile.modmul_bp_pipe gltiming
Startpoint: _639_ (rising edge-triggered flip-flop clocked by clk)
Endpoint: _632_ (rising edge-triggered flip-flop clocked by clk)
...
          4.93   data required time
         -1.59   data arrival time
-------------------------------------------------
          3.33   slack (MET)
```

3 cycles -> 2001.92 sqmicron

Min clock period = 1.59ns

Design space (so far):

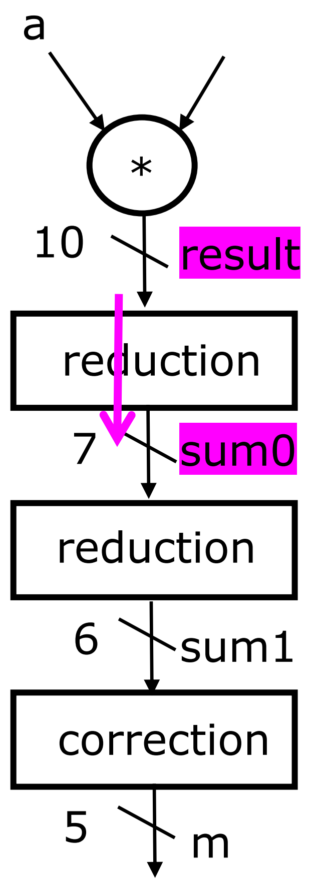


Timing Analysis

```
# make -f Makefile. modmul_bp_pipe gltiming
Startpoint: _639_ (rising edge-triggered flip-flop clocked by clk)
Endpoint: _632_ (rising edge-triggered flip-flop clocked by clk)
...
          4.93   data required time
         -1.59   data arrival time
-------------------------------------------------
          3.33   slack (MET)
```

3 cycles -> 2001.92 sqmicron

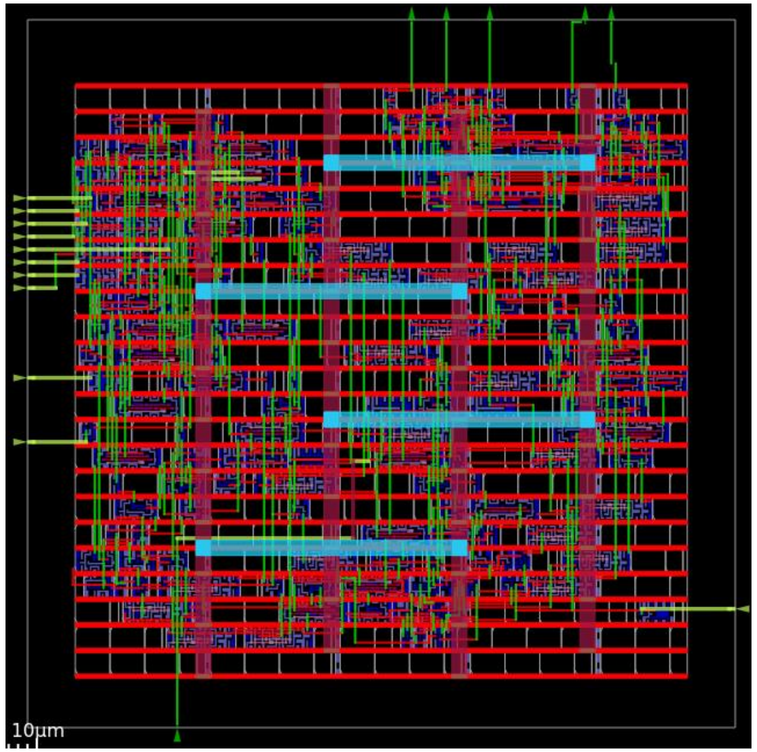
Min clock period = 1.59ns



```
# grep -A 3 _639_ hw_modmul_pipe/work/netlist.v
  sky130_fd_sc_hd__dfxtp_1 _639_ (
    .CLK(clk),
    .D(_000_[5]),
    .Q(resultreg[5])

# grep -A 3 _632_ hw_modmul_pipe/work/netlist.v
  sky130_fd_sc_hd__dfxtp_1 _632_ (
    .CLK(clk),
    .D(_001_[5]),
    .Q(sum0reg[5])
```

Chip Design

```
# make -f Makefile.modmul_bp_pipe openroad
openroad> make -f Makefile.modmul_bp_pipe chip
openroad> make -f Makefile.modmul_bp_pipe chipgui
openroad> make -f Makefile.modmul_bp_pipe chipdata
```

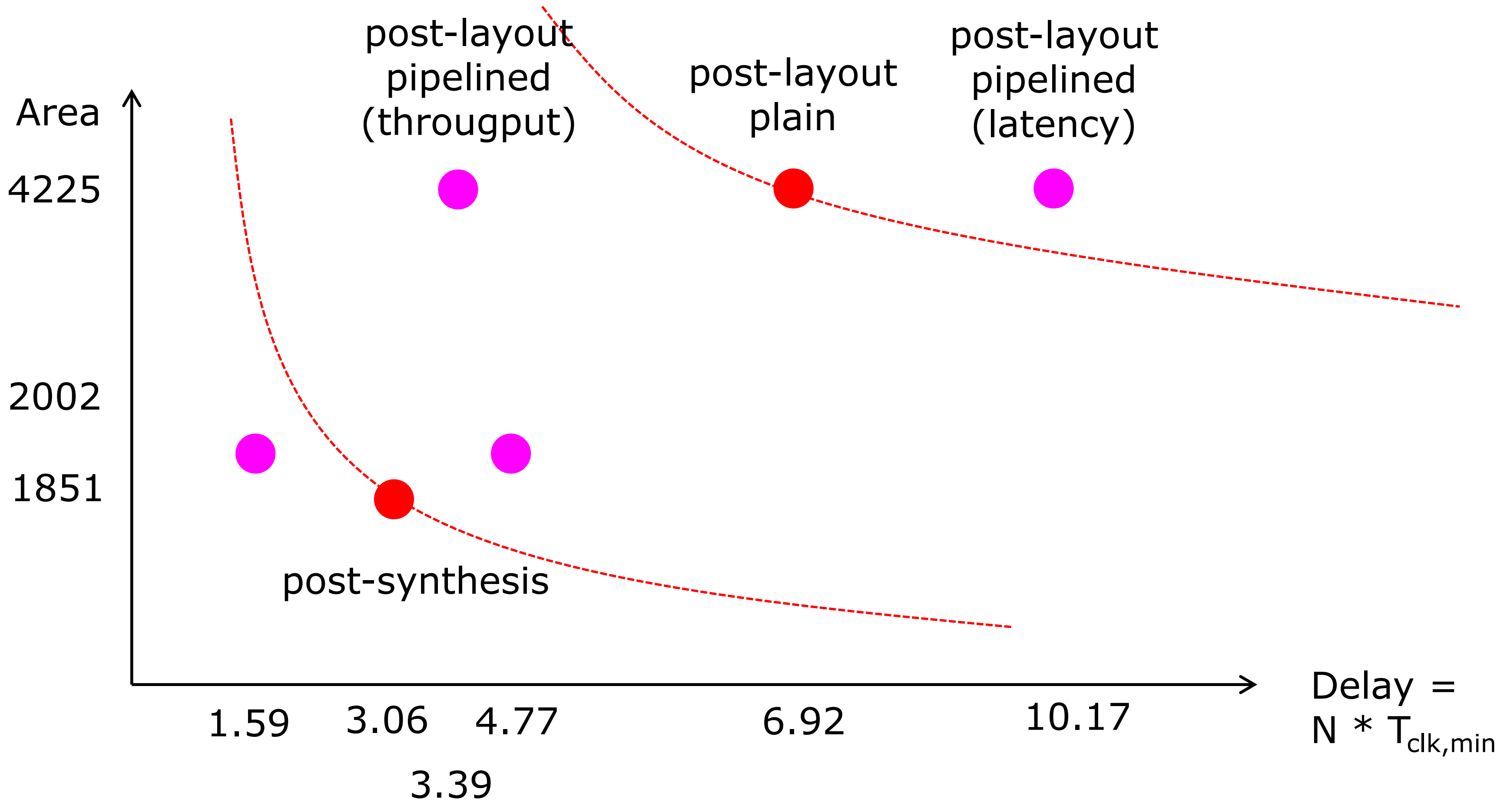


65 x 65 core area = 4225 sqmicron

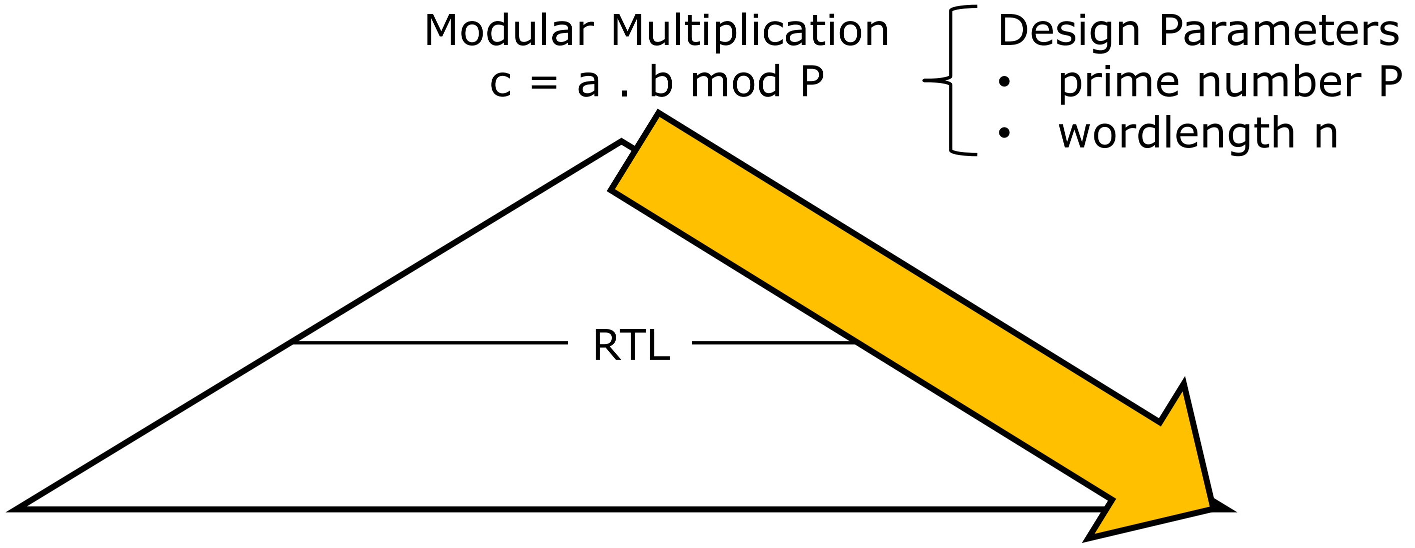
```
Startpoint: b[0] (input port clocked by clk)
Endpoint: resultreg[8]$_SDFF_PP0_
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max
                  5.16   data required time
                 -3.39   data arrival time
----------------------------------------------------------
                  1.77   slack (MET)
```

Due to routing delay, the critical path shifted from the second pipeline stage to the first pipeline stage

Post-synthesis and Post-layout design space



Design Space Exploration



1/n cycles
maximum throughput

pipelined implementation

modmul_bp_pipe

1 cycle
fully parallel design

bitparallel implementation

modmul_bp

n cycles
fully sequential design

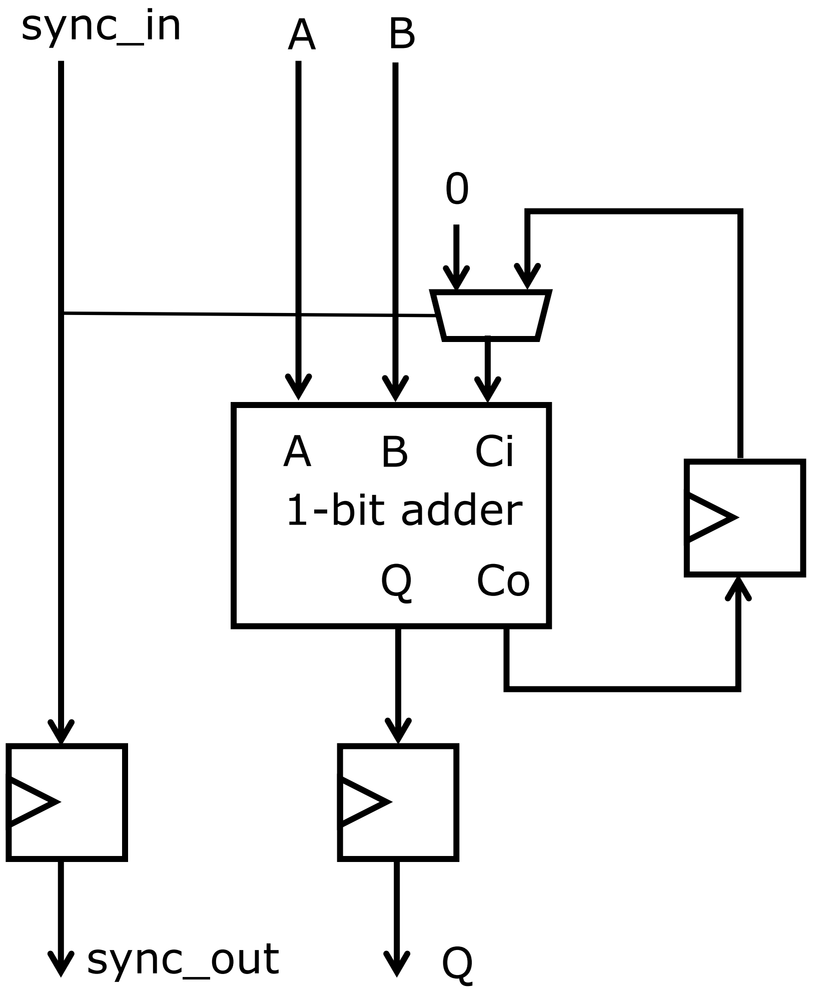
bitserial implementation

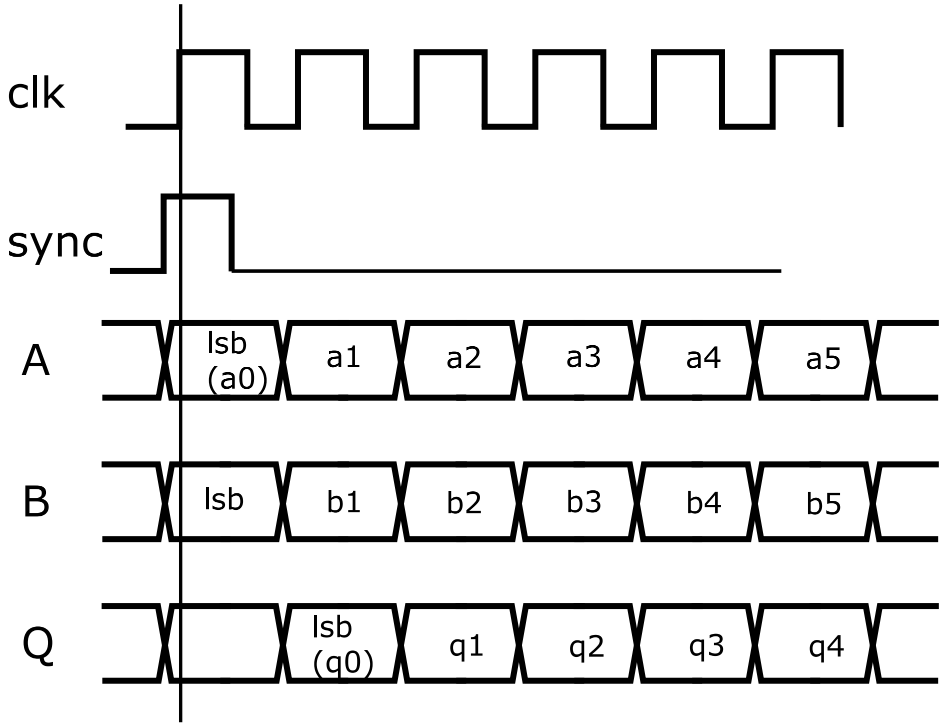
modmul_bs

Bitserial Implementation

- In bitserial implementation, latency is exchanged for area to the extreme: perform computations bit by bit, of possible
- Bitserial implementation leads generally to compact data path but may incur substantial overhead
  - parallel-to-serial, serial-to-parallel conversion
  - state variables cannot be serialized: a 128-bit block-cipher state still requires 128 flip-flops, even if you can process them one by one
  - some data structures, e.g. lookup tables, are difficult to bit-serialize
- Despite that, well-designed bit-serial hardware can break area records (especially in FPGA)
  - E.g. 'SIMON Says: Break Area Records of Block Ciphers on FPGAs' (2014)

Bit-serial Addition





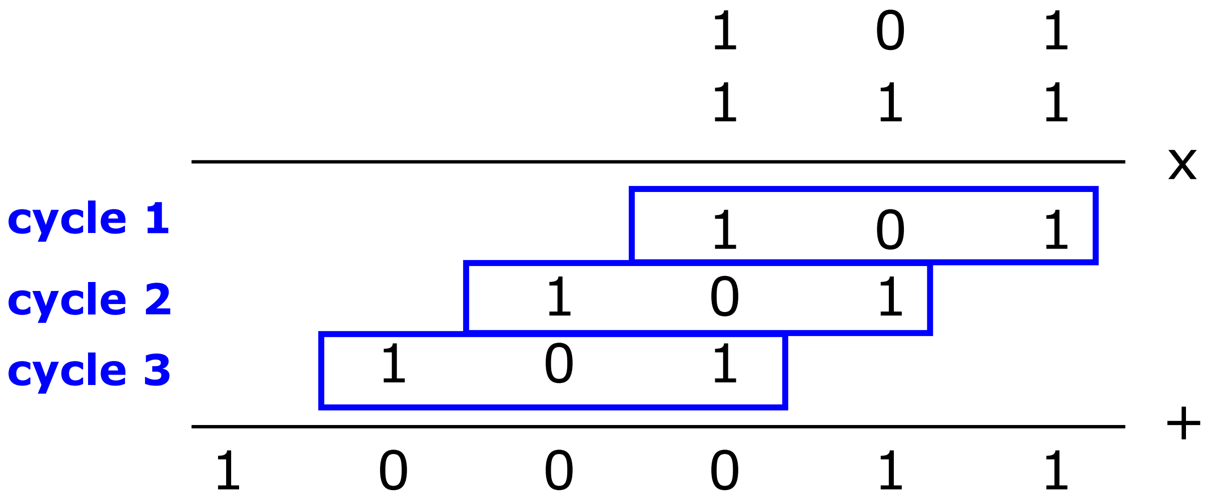
Bit-serial Addition

```
module bsadd(input  reset,
             input  clk,
             input  a,
             input  b,
             output q,
             input  isync,
             output osync
             );
   reg              carry;
   reg              qreg;
   reg              sync;
   wire             newcarry;
   wire             newqreg;
```

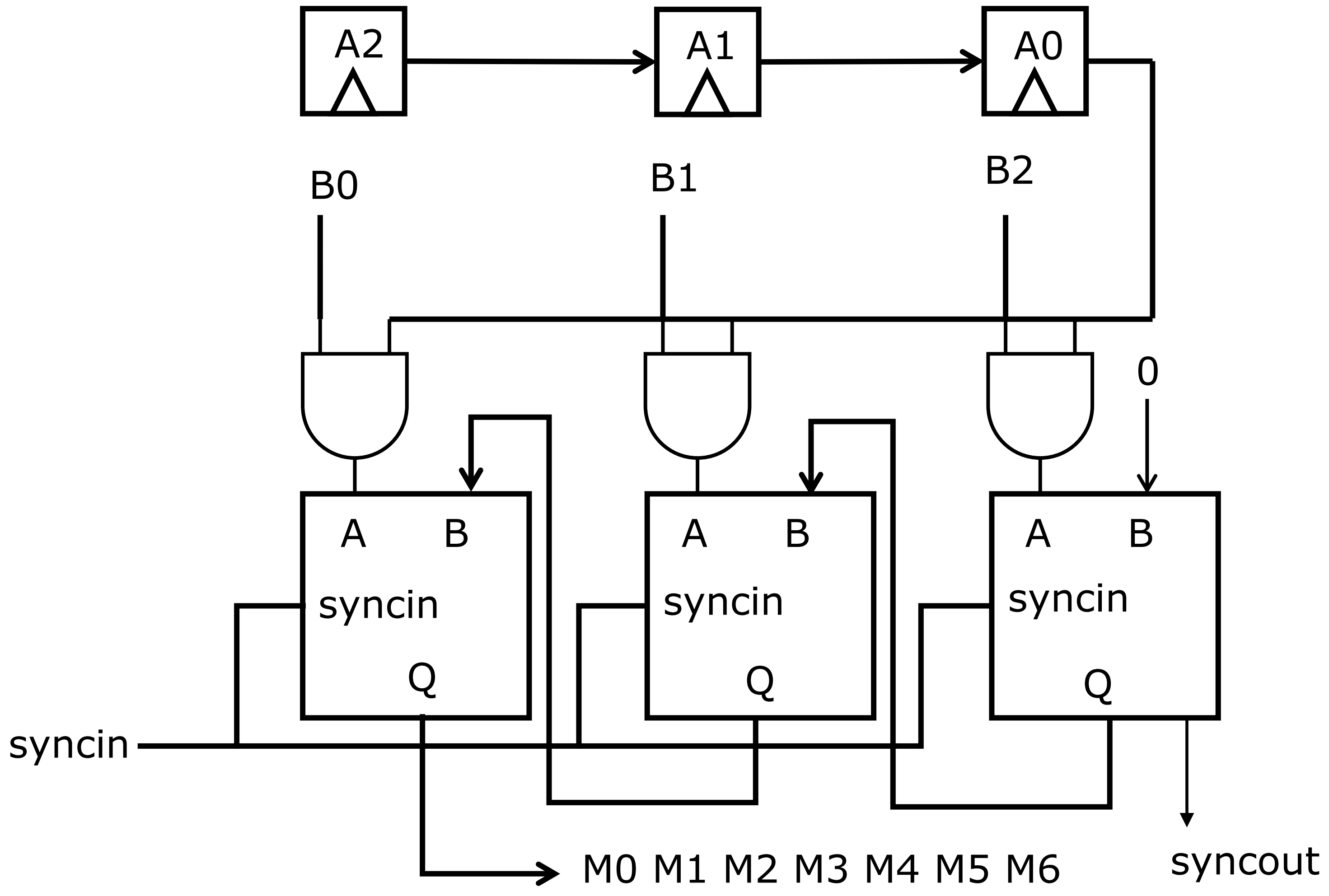
```
always @(posedge clk)
   if (reset)
          begin
             carry <= 1'b0;
             qreg <= 1'b0;
             sync <= 1'b0;
          end
        else
          begin
             carry <= newcarry;
             qreg  <= newqreg;
             sync  <= isync;
          end
   assign {newcarry, newqreg} = a + b +
                                 (isync ? 1'b0 : carry);
   assign q = qreg;
   assign osync = sync;
endmodule
```

Bit-serial Multiplication

- Multiplication has quadratic complexity (n^2 partial products for n bit multiplier)
- This bit-serial multiplication computes n partial products at a time



Bit-serial Multiplication



Bit-serial Multiplication

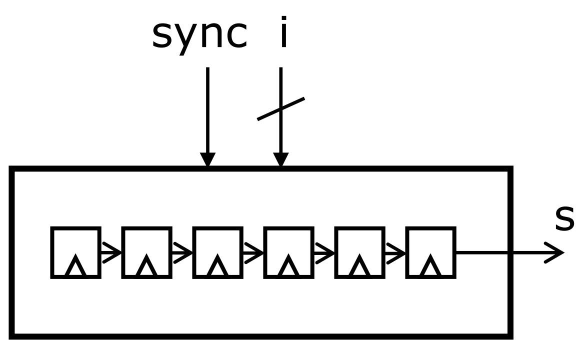
```
module bsmul
  #(parameter LEN = 5)
   (input  reset,
    input  clk,
    input  a,
    input  [LEN-1:0] b,
    output q,
    input  isync,
    output osync
    );

   wire [(LEN-1):0] product;
   wire [(LEN-1):0] bsadd_b;
   wire [(LEN-1):0] bsadd_q;
   wire [(LEN-1):0] osync_add;
```

```
   genvar            i;
   generate
      for (i=0; i<LEN; i = i + 1) begin : addarray

         assign product[i] = a & b[LEN-i-1];

         bsadd thisadd(
                       .reset(reset),
                       .clk(clk),
                       .a(product[i]),
                       .b(bsadd_b[i]),
                       .q(bsadd_q[i]),
                       .isync(isync),
                       .osync(osync_add[i])
               );

         if (i == 0)
           assign bsadd_b[i] = 1'b0;
         else
           assign bsadd_b[i] = bsadd_q[i-1];

      end
   endgenerate

   assign q = bsadd_q[LEN-1];
   assign osync = osync_add[0];

endmodule
```

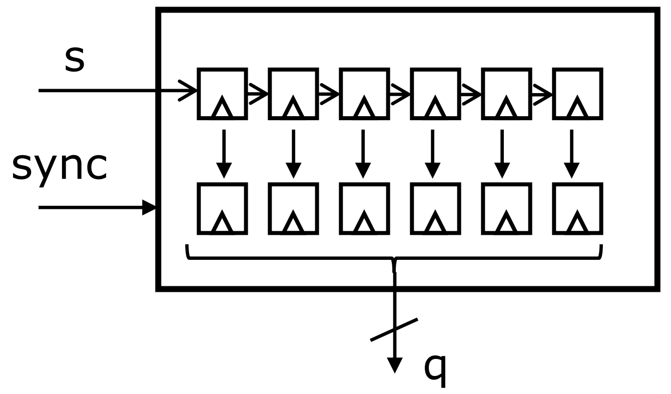
Meta-statement

PISO and SIPO

- Use at the beginning and end of bit-serial implementations to build the interfaces to bit-parallel hardware



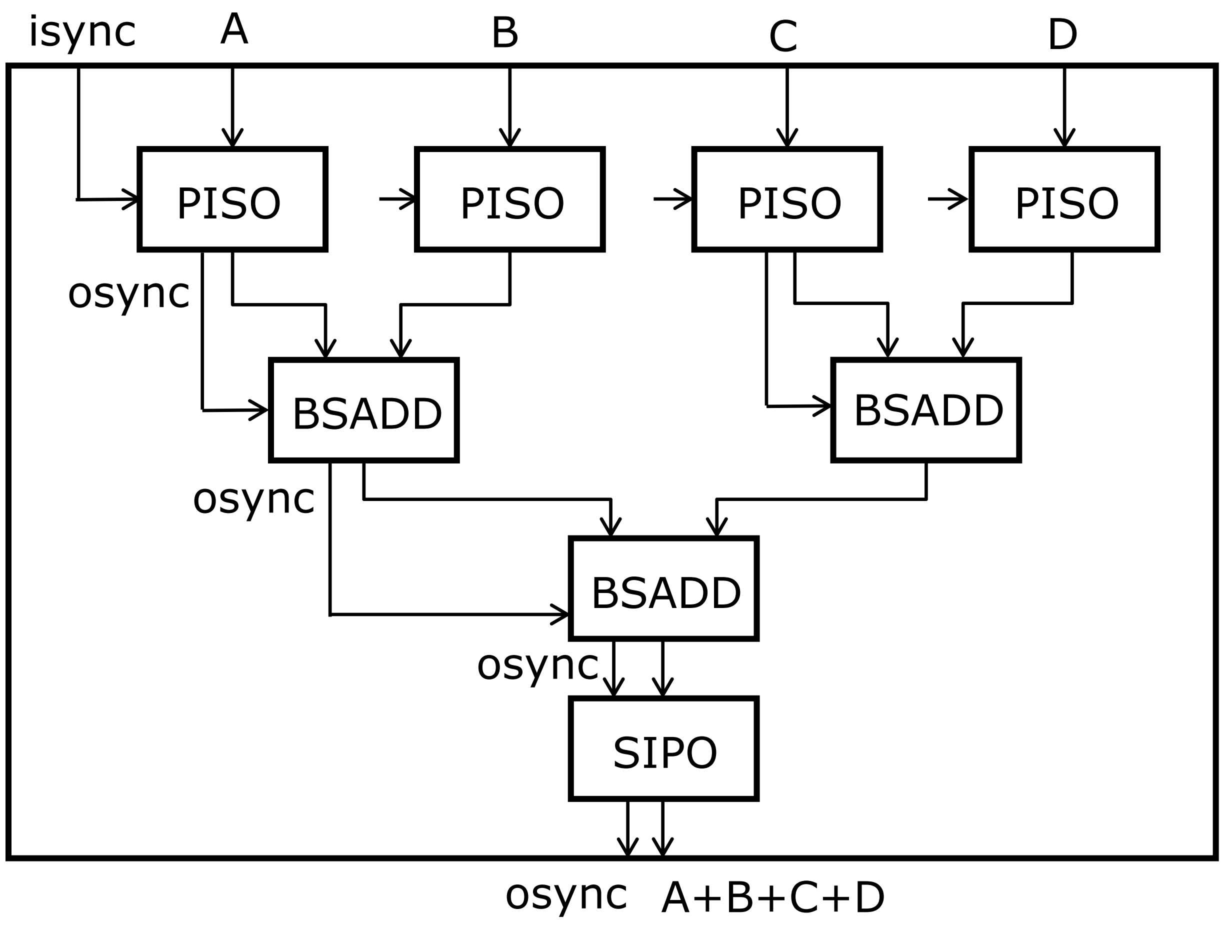
PISO



SIPO

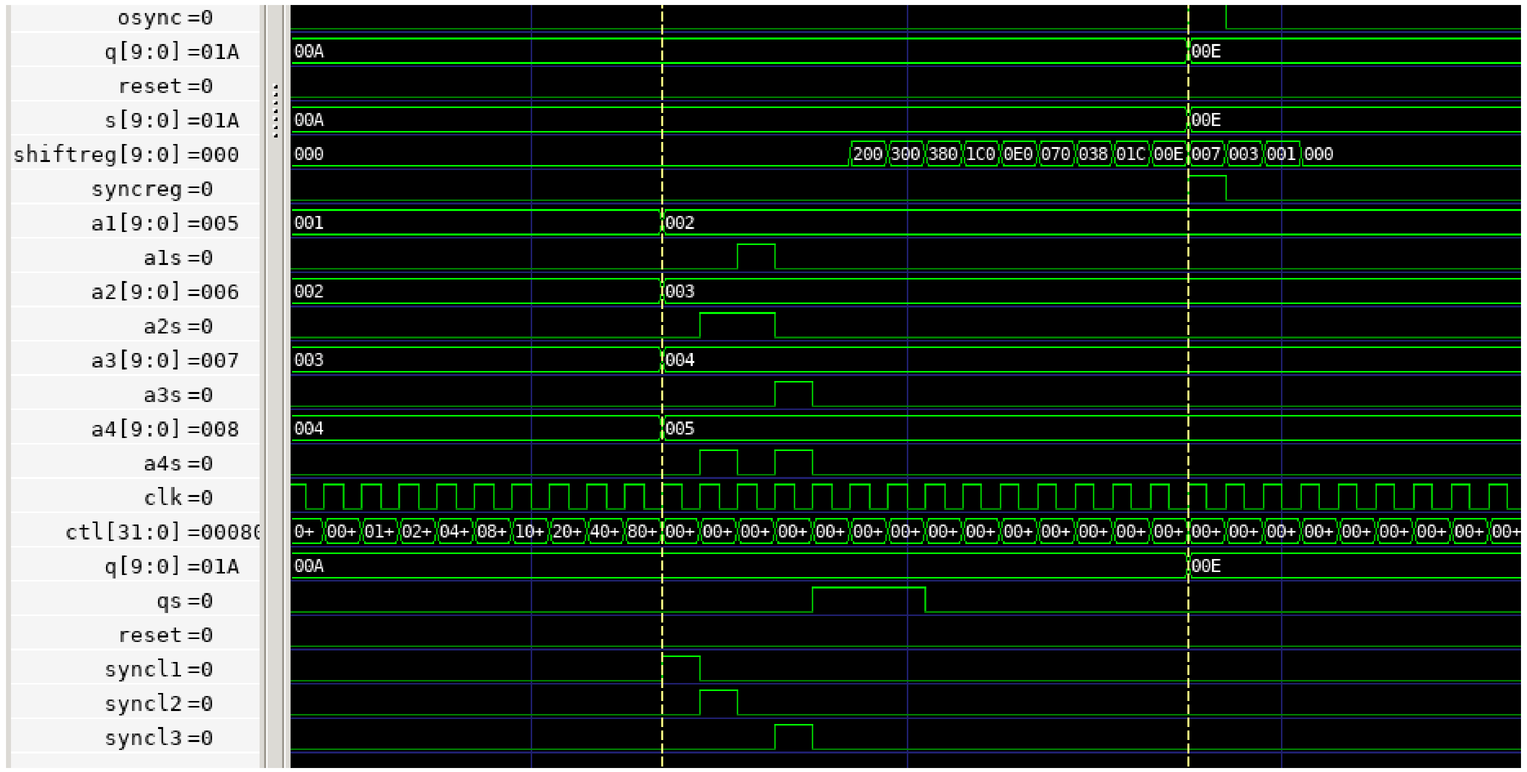
Bit-serial Systems

- Bitserial systems reflect the dataflow of operations with sync



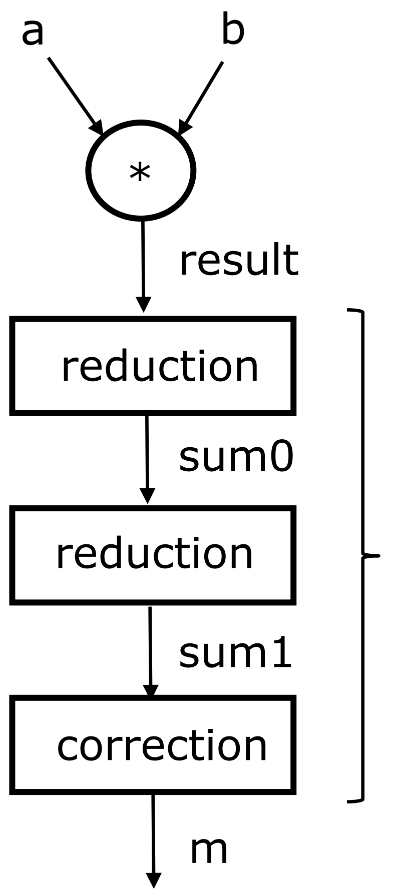
Testing bitserial_add

```
# make -f Makefile.modmul_bs rtlsim
# gtkwave modmul_bs/work/rtl.vcd
```



Bitserial computation is naturally pipelined. While a full word takes 10 cycles to transmit, the sum of 4 numbers completes in 14 cycles starting from the first bit of the first operand

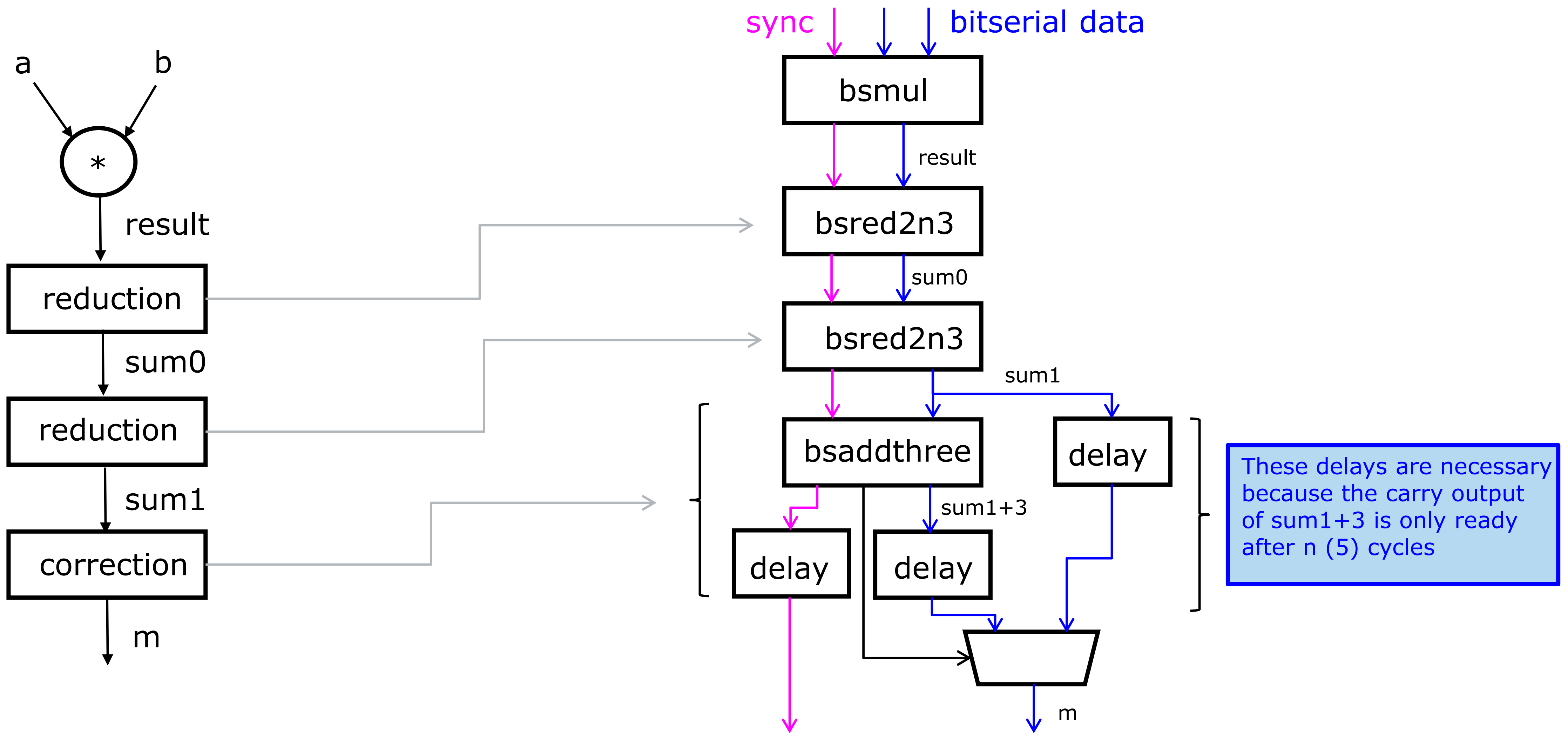
Bitserial Reduction



Bitserial design
for reduction (V0 + 3.V1)
and final correction
((V0 > 29) ? V0 – 29 : V0)

- Bitserial design works well for some operations but can be surprisingly convoluted for others
- Easy:
  Multiplying with 2
- Difficult:
  Word splitting
  Sign Comparison
  Stream Multiplexing

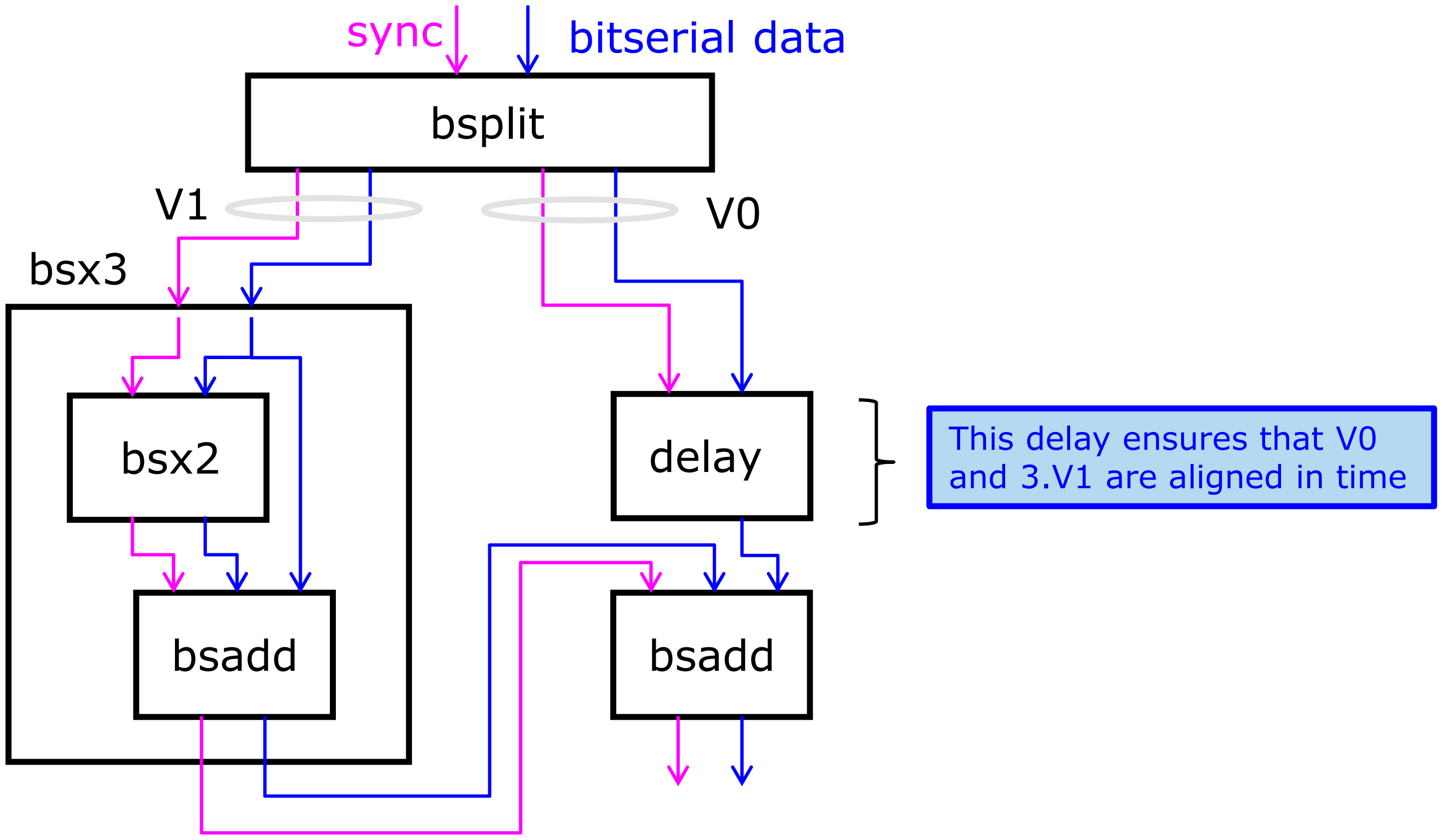
Bitserial Reduction



These delays are necessary because the carry output of sum1+3 is only ready after n (5) cycles

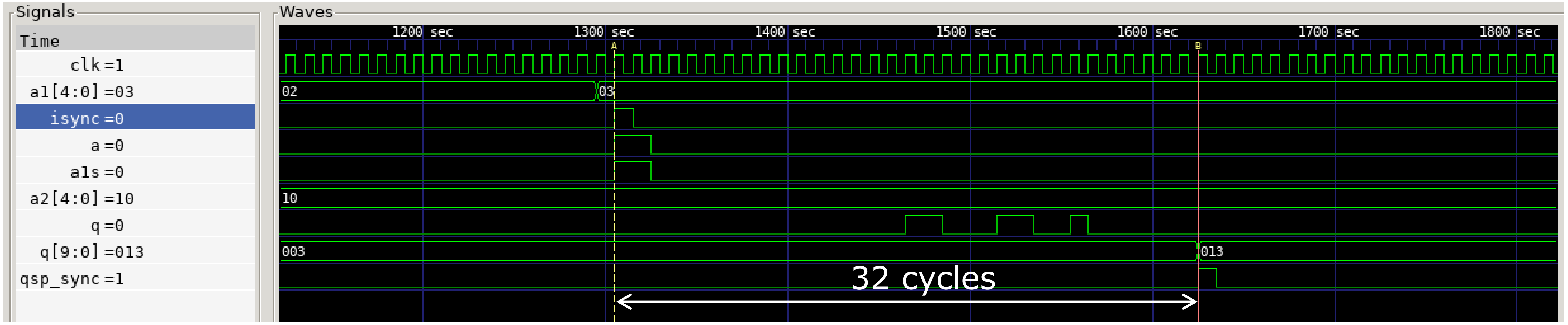
bsred2n3

- Compute V0 + 3.V1 with V0 an n-bit (5-bit) number and V1 and arbitrary length number



Simulation

```
# make -f Makefile.modmul_bs rtlsim
...
01 10 010
02 10 003
03 10 013
04 10 006
05 10 016
06 10 009
07 10 019
08 10 00c
09 10 01c
0a 10 00f
```



Logic Synthesis

```
# make -f Makefile.modmul_bs synthesis
...
20. Printing statistics.

=== bsmodmul ===

   Number of wires:                598
   Number of wire bits:            705
   Number of public wires:          83
   Number of public wire bits:     190
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                674
...
     sky130_fd_sc_hd__clkbuf_4       1
     sky130_fd_sc_hd__dfxtp_1      158
     sky130_fd_sc_hd__inv_1        108
...
 Chip area for module '\bsmodmul': 5436.464000
```

Not a surprise: sharing the computational elements (AND gates) does not balance out against the storage overhead for serialization

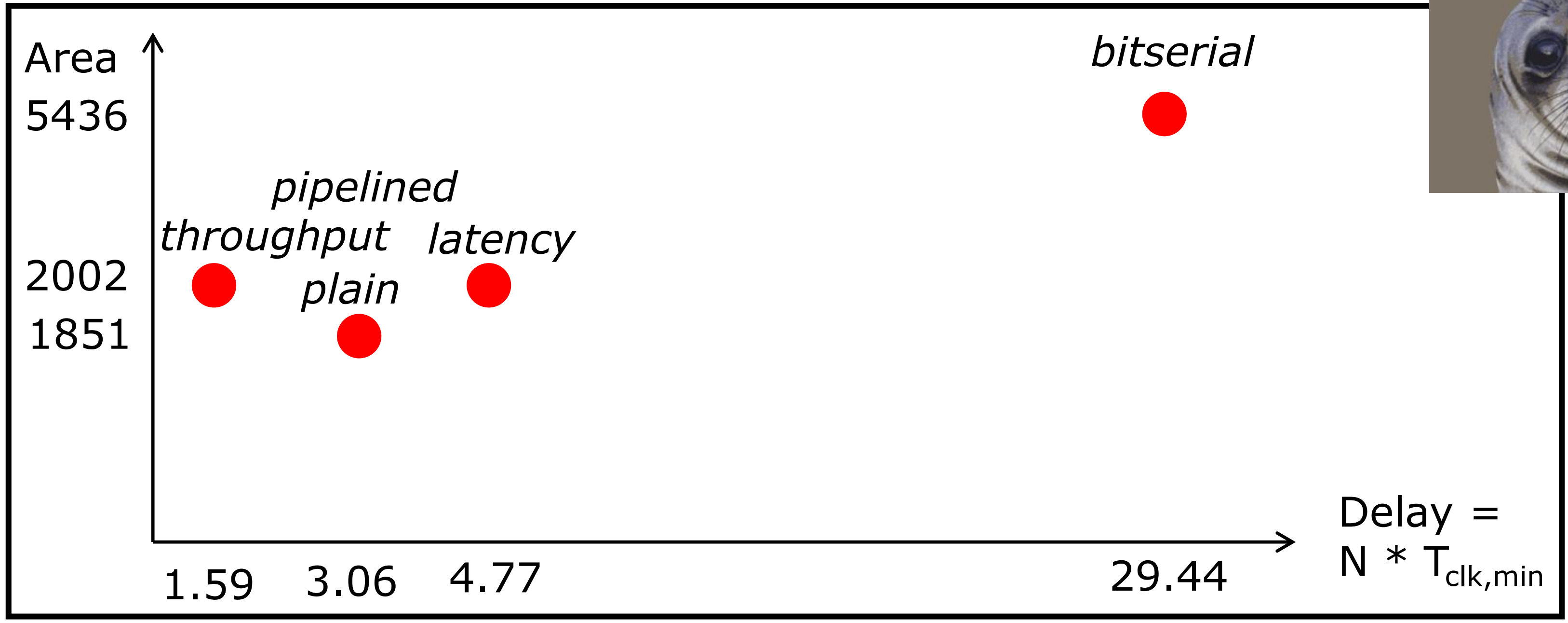
Timing Analysis

```
# make -f Makefile.modmul_bs gltiming
Startpoint: _1089_ (rising edge-triggered flip-flop clocked by clk)
Endpoint: _1091_ (rising edge-triggered flip-flop clocked by clk)
...
           4.89   data required time
          -0.92   data arrival time
---------------------------------------------------------
           3.96   slack (MET)
```

32 cycles -> 5436.46 sqmicron

Min clock period = 0.92ns

Design space :



Bitserial Benefit

- At short wordlengths, the overhead of bitserial implementation can be significant
- For increasing N, the area*delay advantage of bitparallel implementation shrinks but doesn't disappear

Design	N	Area	Cycles	fmax^{-1}	Area * Cycles / fmax
bitparallel	5	1,851	1	3.06	5,664
bitparallel	24	30,642	1	6.46	197,947
bitparallel	94	417,940	1	9.69	4,049,837
bitserial	5	5,436	32	0.92	160,036
bitserial	24	5,760	88	0.92	466,329
bitserial	94	17,716	298	0.92	4,857,018

The main advantage of bitserial implementation lies in compact footprint across a wide range of design parameters

The big picture

13:30

- ASIC Design Flow
- Example
- Modular Multiplication

14:15

- Hands-on

15:00

15:30

- Poly1305 in Hardware

16:00

- Hands-on

17:00

Assignment

1. Log in to the design server
2. Reproduce the design flow (rtlsim, synthesis, glsim, gltiming, openroad/chip, openroad/chipgui) for one or more of the examples discussed above:
   `lfsr`, `modmul_bp`, `modmul_bp_pipe`, `modmul_bs`
3. Modify the prime number from 29 ($2^5 - 3$) to 16777213 ($2^{24} - 3$) and implement a modmul variant (bp, bp_pipe, bs) for this new prime number

   Hint: before making changes, do a

   ```
   cp -r modmul_bp modmul_bp_24
   ```

   Hint2: *this flexibility is where bitserial design really shines!*

Hardware Optimization of Poly1305 for ASIC

Patrick Schaumont
Electrical and Computer Engineering
pschaumont@wpi.edu

The big picture

13:30
- ASIC Design Flow
- Example
- Modular Multiplication

14:15
- Hands-on

15:00

15:30
- Poly1305 in Hardware

16:00
- Hands-on

17:00

Poly1305

- MAC designed by D.J. Bernstein
  optimized for high-speed software implementation
  http://cr.yp.to/mac/poly1305-20050329.pdf

See also RFC7539

- Core idea:
  - Given a message as a sequence of numbers m1, m2, m3, ..
  - Choose a secret r and s and evaluate modulo a big prime p:

$$mac = ((((m1 . r + m2) . r + m3) . r + \dots) + s \bmod p$$

with
r 128 bit
s 128 bit
$p = 2^{130}-5$

- Practical evaluation computes iterations as

```
acc = 0
for each $m_i$
        acc = acc + $m_i$;
        acc = acc * r mod p;
acc = acc + s mod p
```

Poly1305

- MAC designed by D.J. Bernstein
  optimized for high-speed software implementation
  http://cr.yp.to/mac/poly1305-20050329.pdf

See also RFC7539

- Core idea:
  - Given a message as a sequence of numbers m1, m2, m3, ..
  - Choose a secret r and s and evaluate modulo a big prime p:

$$\text{mac} = ((((m1 \,.\, r + m2) \,.\, r + m3) \,.\, r + \dots) + s \bmod p$$

with
r 128 bit
s 128 bit
$p = 2^{130}-5$

- r has 22 bits of the 128 bits cleared or *clamped*; this allows optimization of long-worldlength arithmetic in 32-bit or 64-bit software implementations. Hardware doesn’t have this restriction but clamping is needed to implement Poly1305 standard

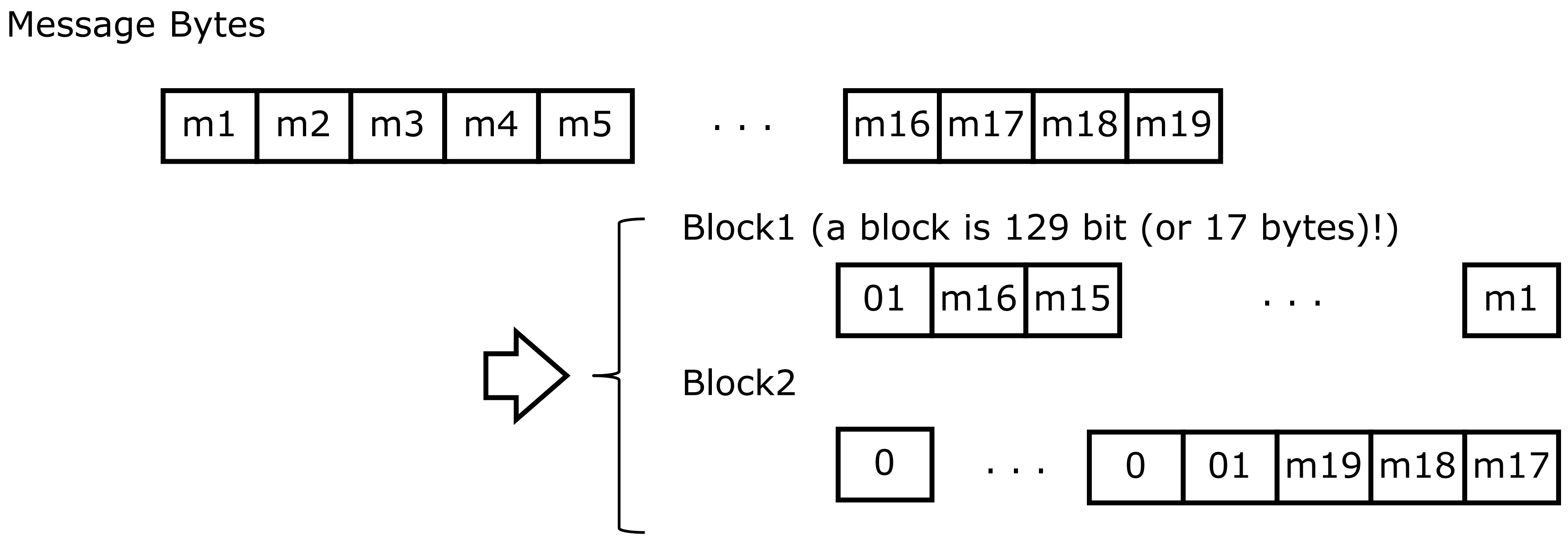
```
rclamp = r & 128'h0FFF_FFFC_0FFF_FFFC_0FFF_FFFC_0FFF_FFFF;
```

Poly1305

- Message padding

  message bytes loaded as 128-bit number terminated by a 01 byte = block

  a partially filled block is terminated by a 01 byte



Byte order and number interpretation

- Messages, keys are expressed as a sequence of bytes (left to right)
  - E.g. char *msg = “Cryptographic Fo”;
    maps to the byte sequence
    `43 72 79 70 74 6f 67 72 61 70 68 69 63 20 46 6f` (+ 0)
  - Eg RFC7539 specifies a 32-byte key value as
    `85:d6:be:78:57:55:6d:33:7f:44:52:fe:42:d5:06:a8:`
    `01:03:80:8a:fb:0d:b2:fd:4a:bf:f6:af:41:49:f5:1b`
- In hardware, the lsbyte to msbyte order is right to left
  - reg [127:0] msg = `128’h6f462063_69687061_72676f74_70797243`
  - if key = {r, s} then

```
reg [127:0] s = 128’h1bf54941_aff6bf4a_fdb20dfb_8a800301
reg [127:0] r = 128’ha806d542_fe52447f_336d5557_78bed685
```

- Test vectors in hardware follow the hardware byte order

Modular multiplication mod $P = 2^{130} - 5$

- Given two unsigned integers A,B
  $0 <= A, B < 2^{130} - 5$
- (A * B) mod P is evaluated as follows

1

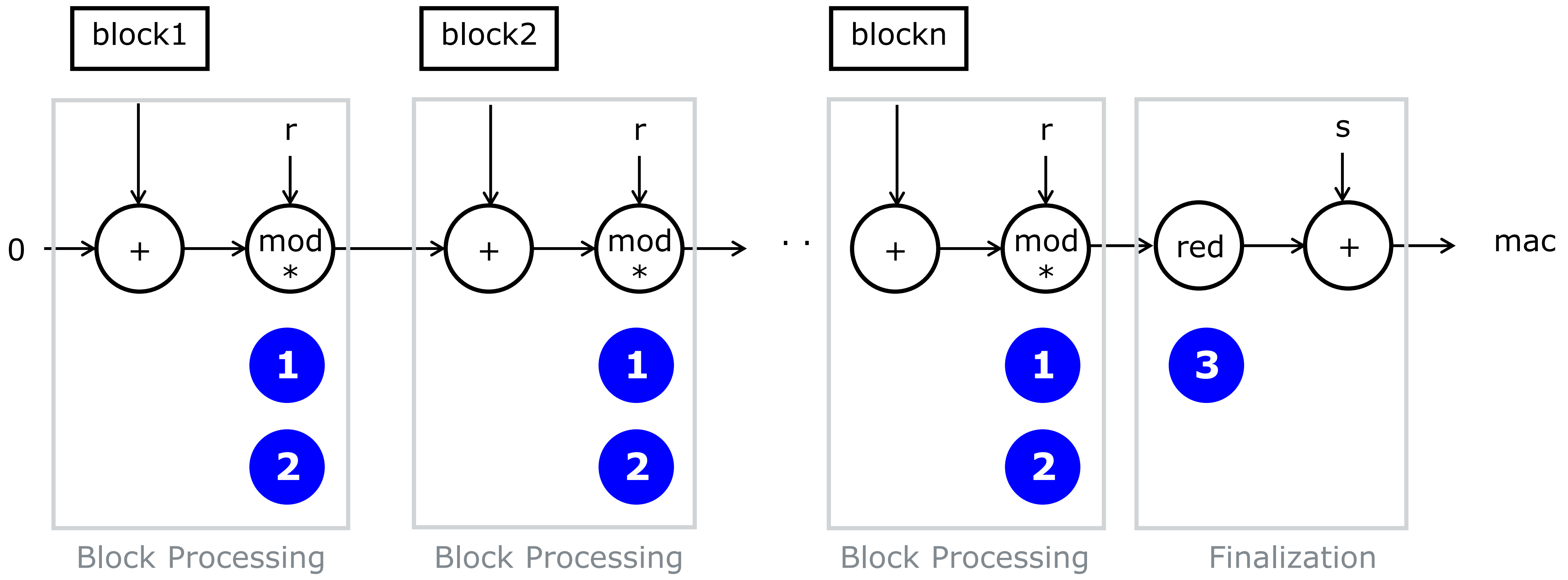
$$\begin{aligned} V \bmod P &= A * B \bmod P \\ V \bmod P &= (V0 + V1.2^{130}) \bmod P \\ V \bmod P &= \underbrace{(V0 + V1.5)}_{\text{Repeat until result is <= 130 bit}} \bmod P \end{aligned}$$

2

3

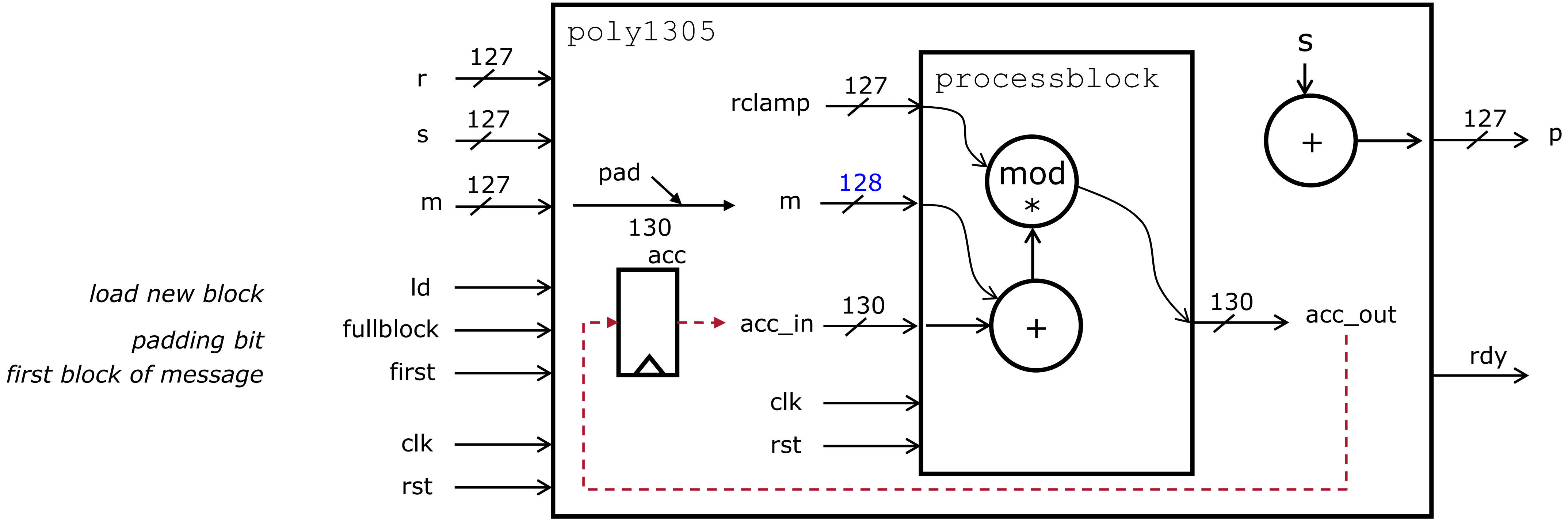
If $V = (2^{130} - k)$ with $k = 1,2,3,4$, then
        $R = 5 - k$
else
        $R = V$

Poly1305 Computation



Hardware Design

- A full poly1305 computation requires state



Processblock, fully parallel

```
module processblock(input          reset,
                    input          clk,
                    input  [127:0] r,
                    input  [128:0] m,     // {1 padding + 128 message bits}
                    input  [129:0] a_in, // {acc < P = 2^130 - 5}
                    output [129:0] a_out,
                    input          start,
                    output         done
                    );

   wire [130:0]                    m1;
   wire [258:0]                    m2; // 128 bits * 131 bits = 259 bits
   wire [131:0]                    m3; // first reduction leaves 2 extra bits
   wire [129:0]                    m4;
   wire [2:0]                      five;

   assign five = 5;
   assign m1 = m + a_in;
   assign m2 = m1 * r;                          // 131 x 128 multiplier!!
   assign m3 = m2[129:0] + m2[258:130] * five;     // first reduction
   assign a_out = m3[129:0] + m3[131:130] * five;  // second reduction
   assign done = start;

endmodule
```

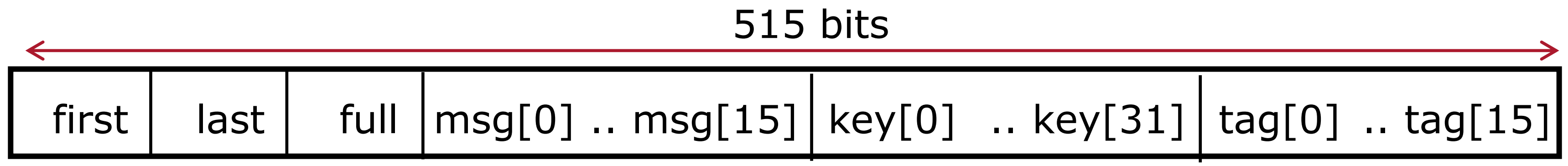
Poly1305, fully parallel

```
module poly1305(input           reset,
                input           clk,
                input  [127:0]  r,
                input  [127:0]  s,
                input  [127:0]  m,
                input           fb,
                input           ld,
                input           first,
                output [127:0]  p,
                output          rdy);
   reg [129:0]                  acc;
   wire [129:0]                 acc_out;
   wire [129:0]                 acc_in;
   wire                         block_start;
   wire                         block_done;

   wire [128:0]                 msep;
   assign msep = fb ? {1'b1, m} : m;

   assign acc_in = first ? 130'b0 : acc;


   wire [127:0]                 rclamp;

   assign rclamp = r & 128'h0FFF_FFFC_0FFF_FFFC_0FFF_FFFC_0FFF_FFFF;
```

```
   processblock single(.reset(reset),
                       .clk  (clk),
                       .r    (rclamp),
                       .m    (msep),
                       .a_in (acc_in),
                       .a_out(acc_out),
                       .start(block_start),
                       .done (block_done)
                       );

   always @(posedge clk)
     if (reset)
       acc <= 130'h0;
     else
       acc <= block_done ? acc_out : acc;

   assign block_start = ld;

   assign p = acc_out + s;
   assign rdy = block_done;

endmodule
```

Testing the hardware design

- Use a C program to generate test vectors
  - C program can use an existing poly1305 implementation for verification
- Hardware testvector format



  - each vector represents the processing of one message block
  - *first* marks the first block of a stream (reset the accumulator)
  - *last* marks the last block of a stream (verify tag correctness)
  - *full* marks a full message block that requires an extra padding byte
  - msg are the message bytes in left to right order
  - key are the key bytes in left to right order
  - tag are the reference tag bytes in left to right order

Testvector generation

```
# cd poly1305_tv
# make polygen
gcc polygen.c poly1305.c -o polygen
./polygen >vectors.txt

# tail vectors.txt
10100100111010101000111011101100001011100110010000001100010011100100110100
10110110001101100011010010110011100101100001000000110000100011100100100100
10000001010010111101011010101011101001110001010111100110011001110001000100
00110000001001111011010110101111100000100011100111001000101111100000101000
00000101011100000000000100110011101110010100101110010111100001000000111000
00111010111000000010001010100000101100110100110100111111010101010111011100
11000011110011100001000110111000111110010111100110001011110101101100010

...
```

Hardware testbench applies vectors on RTL

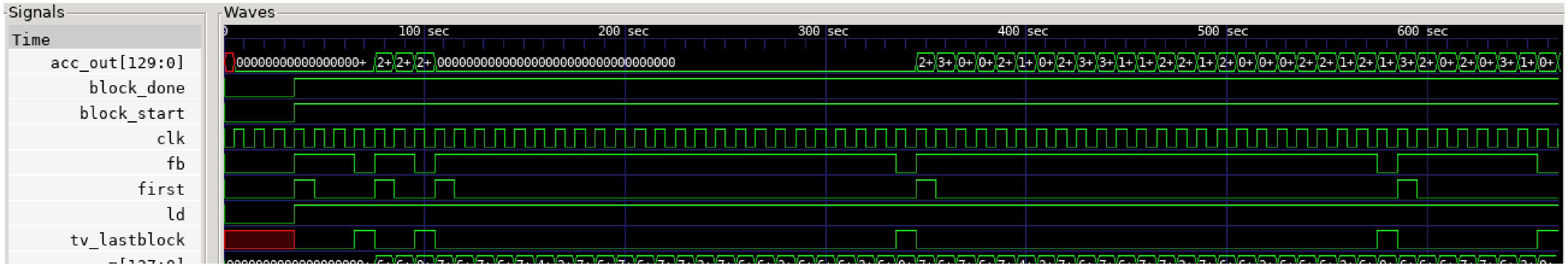
```
module tb;
   reg [127:0]  r;
   reg [127:0]  s;
   reg [127:0]  m;
   reg          fb;
   reg          ld;
   reg          first;
   wire [127:0] p;
   wire         rdy;
   reg          reset;
   reg          clk;

   poly1305 dut(.reset(reset),
                .clk(clk),
                .r(r),
                .s(s),
                .m(m),
                .fb(fb),
                .ld(ld),
                .first(first),
                .p(p),
                .rdy(rdy));
   always
     begin
        clk = 1'b0;
        #5;
        clk = 1'b1;
        #5;
     end
```

```
reg [514:0] inputvector;
integer     inputfile;

reg         tv_firstblock;
reg         tv_lastblock;
reg         tv_fullblock;
reg [127:0] tv_data;
reg [255:0] tv_key;
reg [127:0] tv_tag;
reg [127:0] tv_tag_swap;
integer     i;

initial
  begin
     $dumpfile("trace.vcd");
     $dumpvars(0, tb);
     inputfile =
     $fopen("../../C/vectors.txt","r");

     r = 128'h0;
     s = 128'h0;
     m = 128'h0;
     fb = 1'b0; // fulblock
     ld = 1'b0;
     first = 1'b0;
   ...
```

```
while ($fscanf(inputfile, "%b", inputvector)
       != 0)
     begin
       {tv_firstblock,
        tv_lastblock,
        tv_fullblock,
        tv_data,
        tv_key,
        tv_tag} = inputvector;
        ...

       ld = 1'b1;
       @(posedge clk);

       while (rdy == 1'b0)
         begin
            ld = 1'b0;
            @(posedge clk);
         end

       if ((tv_lastblock == 1'b1) &&
           (p != tv_tag_swap))
            $error;
       else if (tv_lastblock == 1'b1)
           $display("Tag OK %x", p);
    end
  $finish;
end
```

RTL simulation of fully parallel design

```
# make -f Makefile.poly1305_bp rtlsim
...
VCD info: dumpfile trace.vcd opened for output.
Tag OK: 00000000000000000000000000000000
Tag OK: a927010caf8b2bc2c6365130c11d06a8
Tag OK: 3e867a2296caeff07060e0c5b5f6e536
Tag OK: f00c314c79b8a689af1754d97c7e47f3
Tag OK: 62ebc5bc7cdc08e761eeaa7e9a664145
```



Hardware Synthesis

```
# make -f Makefile.poly1305_bp synthesis

...
11. Printing statistics.

=== poly1305 ===

   Number of wires:              138465
   Number of wire bits:          139231
   Number of public wires:           11
   Number of public wire bits:      648
   Number of memories:                0
   Number of memory bits:             0
   Number of processes:               0
   Number of cells:              138841
...
   sky130_fd_sc_hd__dfxtp_1       130
...
   Chip area for module '\poly1305': 627170.256000
```

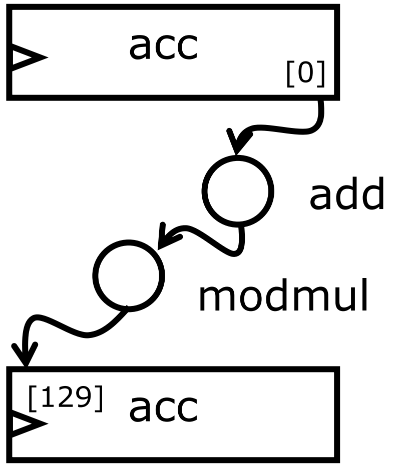
138K standard cells
627k square micron area

Timing Analysis

```
# make -f Makefile.poly1305_bp gltiming

Startpoint: _277165_ (rising edge-triggered flip-flop clocked by cl
Endpoint: _277294_ (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max

  Delay    Time   Description
---------------------------------------------------------
   0.00    0.00   clock clk (rise edge)
   0.00    0.00   clock network delay (ideal)
   0.00    0.00 ^ _277165_/CLK (sky130_fd_sc_hd__dfxtp_1)

   ...

   0.00   12.29 v _277294_/D (sky130_fd_sc_hd__dfxtp_1)
          12.29   data arrival time

---------------------------------------------------------
           4.88   data required time
         -12.29   data arrival time
---------------------------------------------------------
          -7.41   slack (VIOLATED)
```

There are 119 cells in the critical path!

```
# grep -A 3 _277165_ poly1305/work/netlist.v
 sky130_fd_sc_hd__dfxtp_1 _277165_ (
   .CLK(clk),
   .D(_000000_[0]),
   .Q(acc[0])

# grep -A 3 _277294_ poly1305/work/netlist.v
 sky130_fd_sc_hd__dfxtp_1 _277294_ (
   .CLK(clk),
   .D(_000000_[129]),
   .Q(acc[129])
```



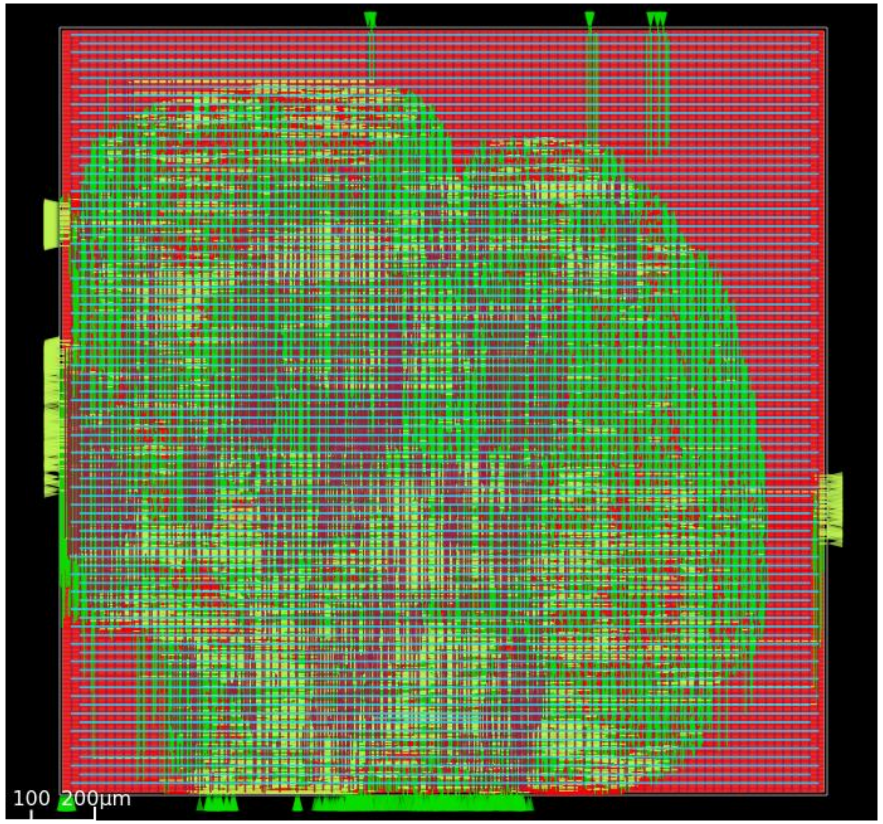
Worcester Polytechnic Institute

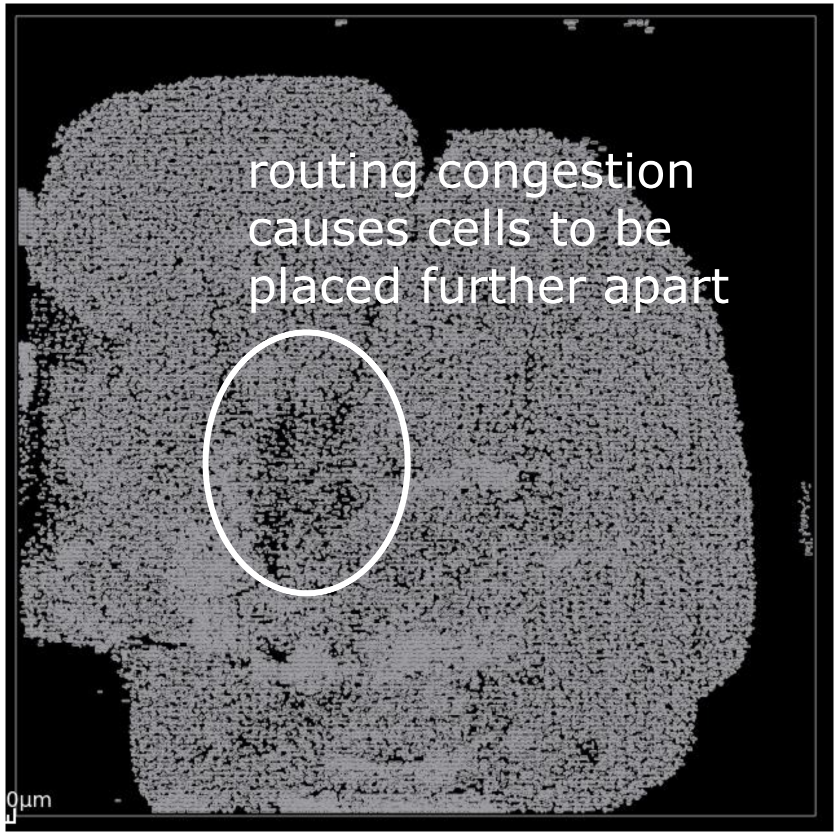
Chip Design

this takes 15min on your work-station...

```
# make -f Makefile.poly1305_bp openroad
# make -f Makefile.poly1305_bp chip
# make -f Makefile.poly1305_bp chipgui
```

Area = 1200 x 1200 = 1440000
Post-layout timing:
acc[23]->p[98]: 28.90 ns!

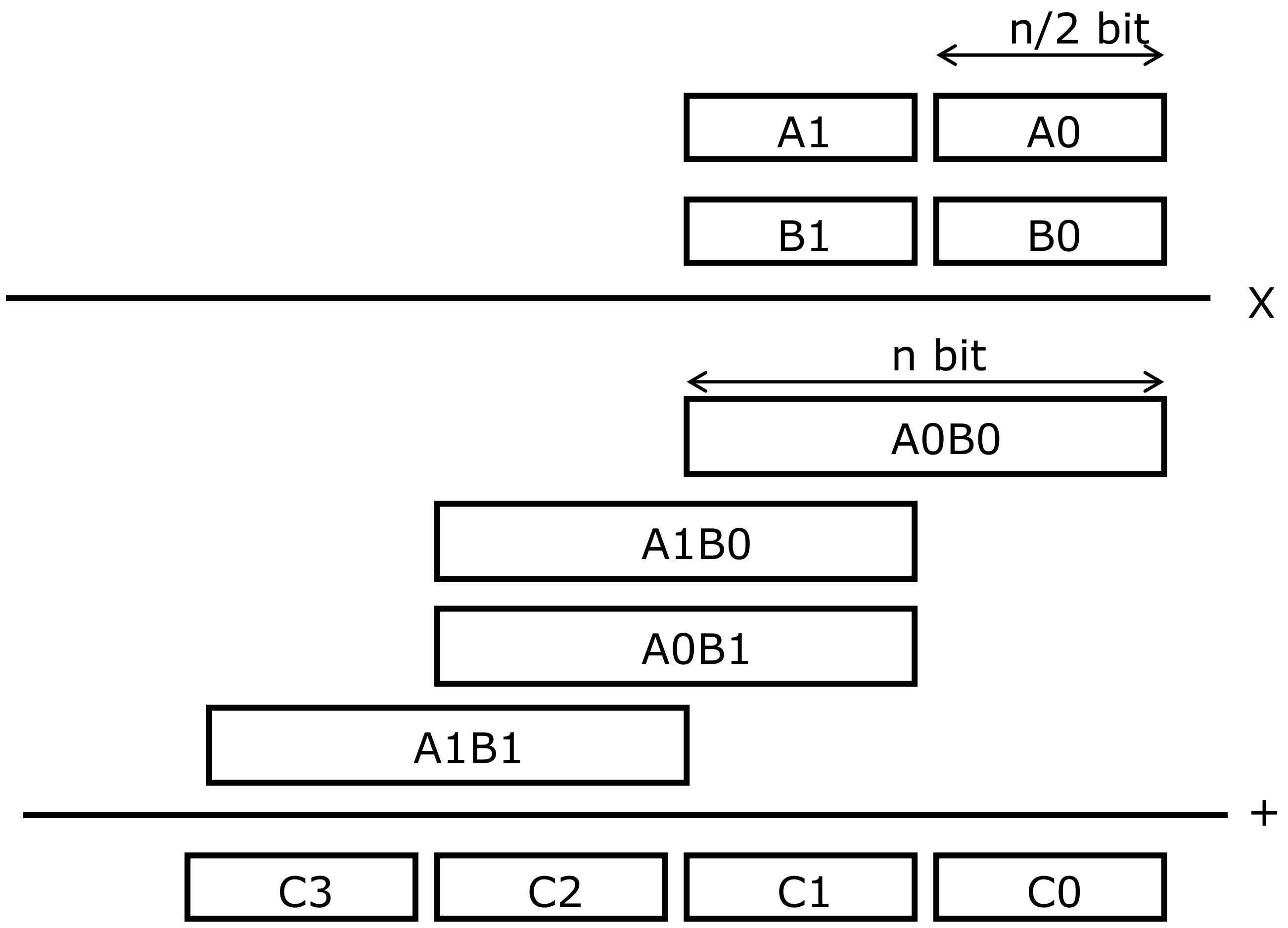




Sequentialized Poly1305

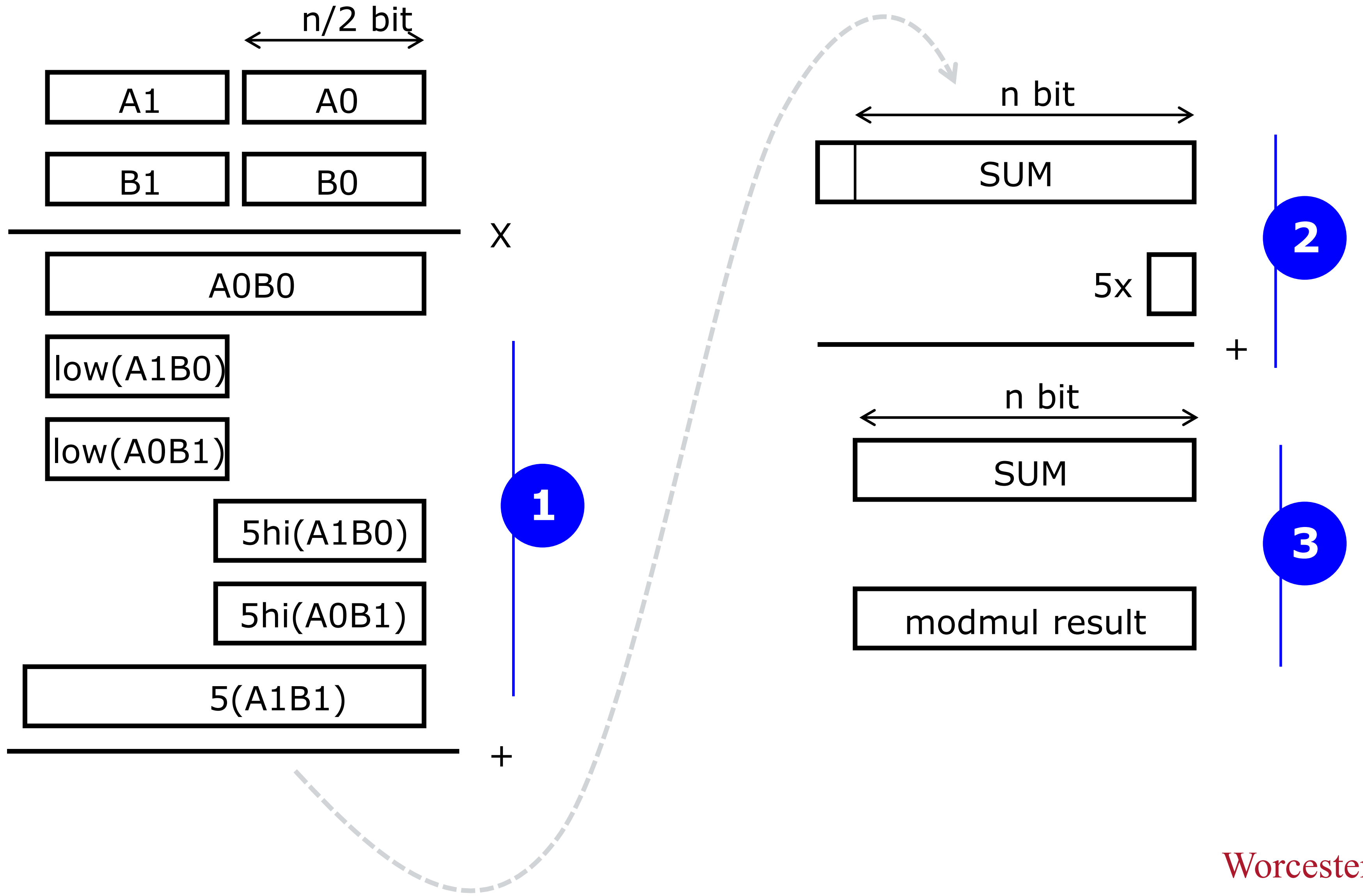
- Clearly, a fully parallel (single-cycle) multiplication has an enormous area cost, and significant impact on the clock period
- Pipelining the design may reduce clock period, and improve throughput, but will not reduce area cost
- To make a smaller design, we need to reuse hardware (logic gates).
- The 131-bit x 128-bit multiplier is ideal candidate for reuse when we decompose a long-wordlength multiplication into several small-wordlength multiplications

Decomposed multiplication

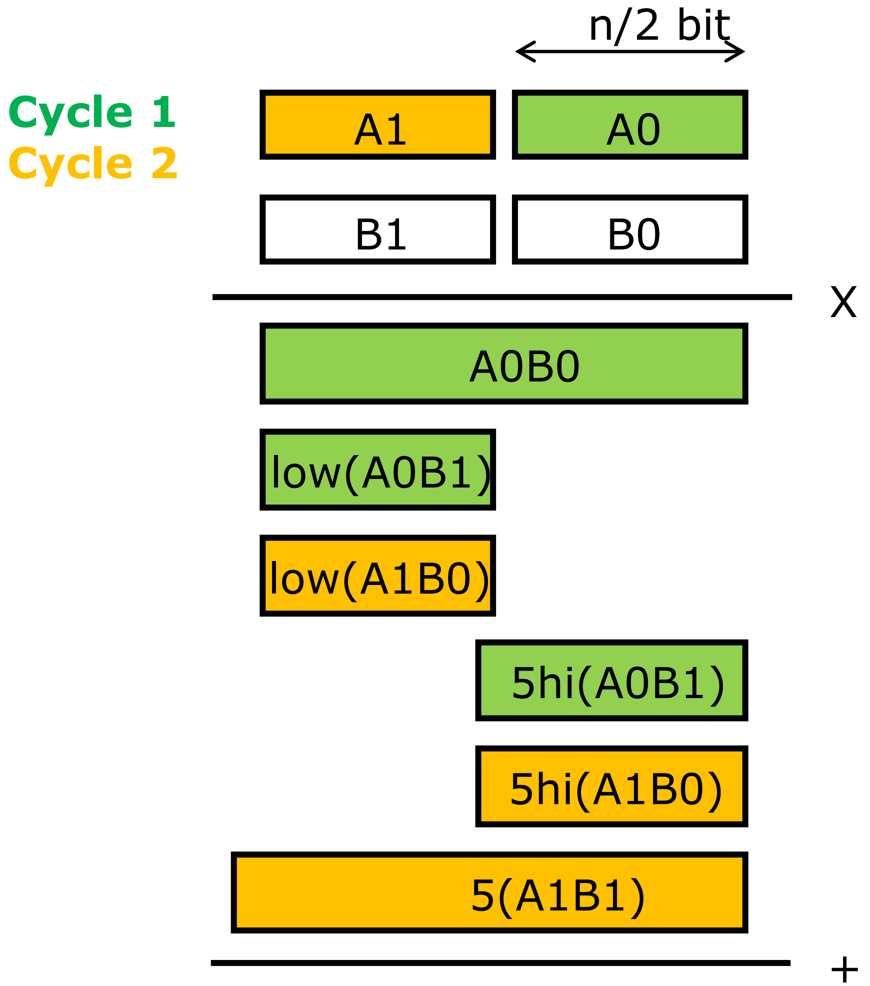


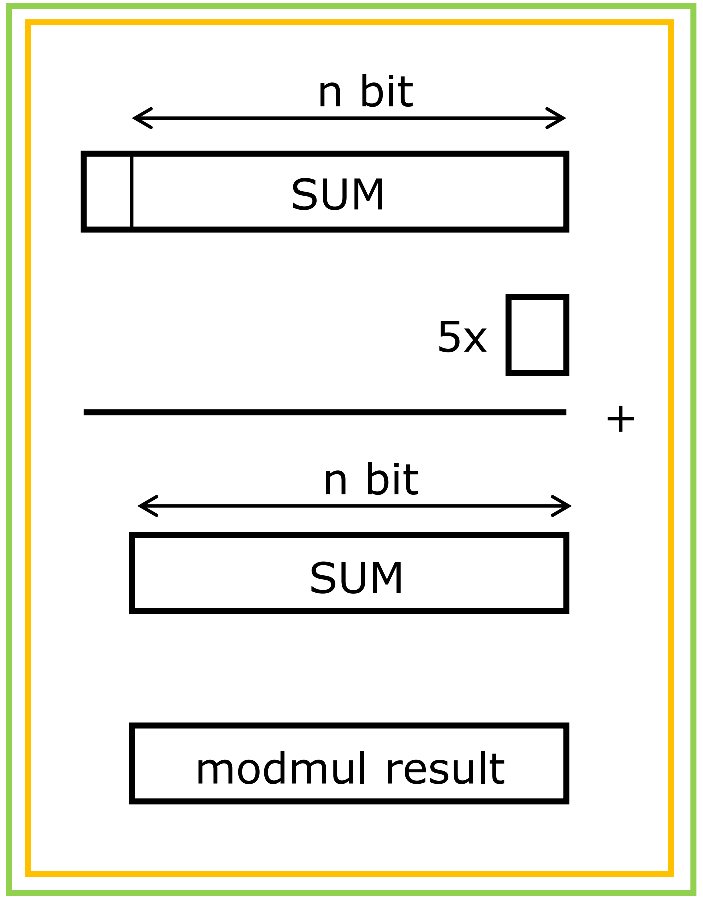
Worcester Polytechnic Institute

Decomposed modular multiplication (mod 2^n – 5)



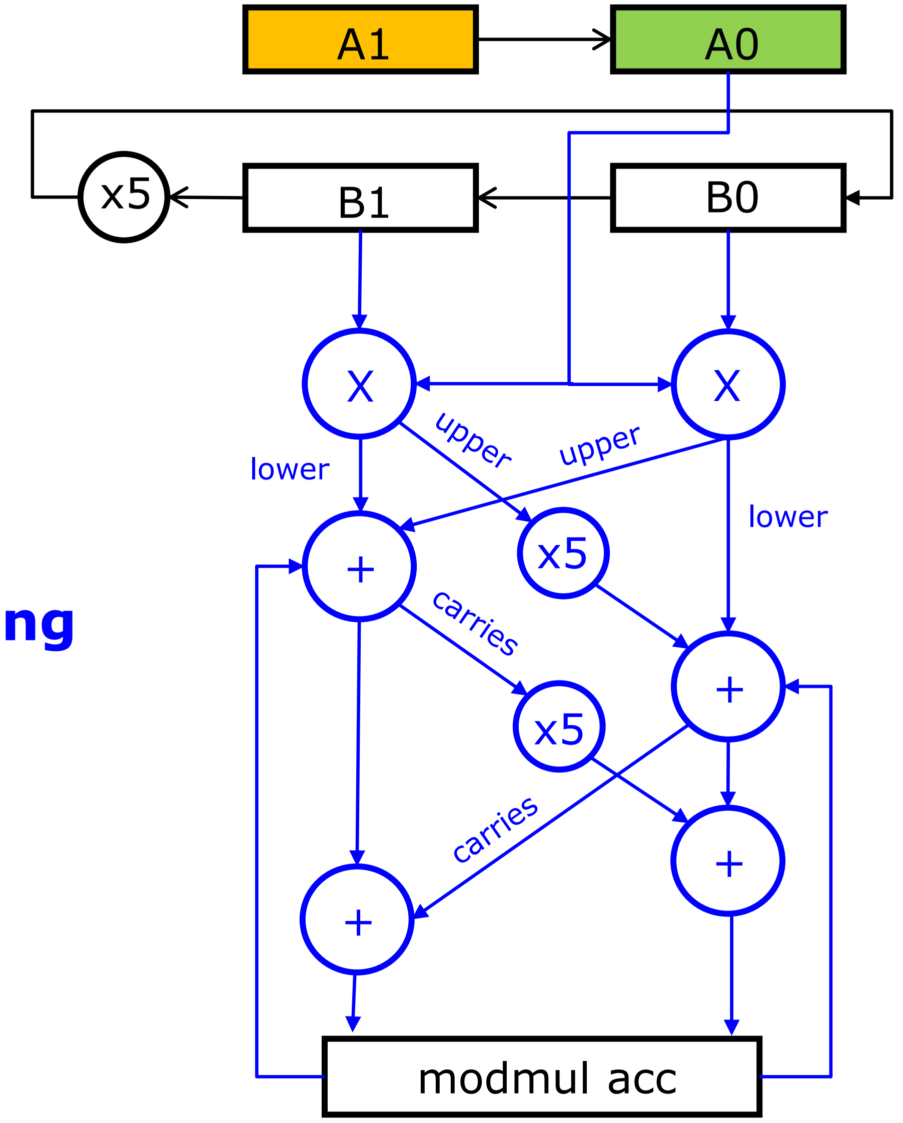
Sequentialized design (2 cycles)





Accumulated in every cycle

Sequentialized design (2 cycles)



Shift Register

Shift Register
with x5 multiply

Two parallel
n/2 bit multipliers

Hardware Mapping

Sequentialized Design (5 cycles)

```
module processblock(..);
   reg [131:0]             rm;
   reg [129:0]             rr;
   wire [129:0]            rr_rotred;

   wire [130:0]            initialsum;
   wire [129:0]            initialsumred;
   assign initialsum = m + a_in;
   assign initialsumred = initialsum[130] ?
                    initialsum[129:0] + 3'h5 :
                    initialsum[129:0];
   always @(posedge clk)
     begin
        if (reset) begin
             rm <= 130'h0;
             rr <= 128'h0;
          end else begin
             rm <= start ? initialsumred :
                           {32'h0, rm[129:32]};
             rr <= start ? r : rr_rotred;
          end
     end

   wire [63:0]  mul0, mul1, mul2;
   wire [65:0]  mul3;
   assign mul0 = rr[ 31: 0] * rm[31:0];
   assign mul1 = rr[ 63:32] * rm[31:0];
   assign mul2 = rr[ 95:64] * rm[31:0];
   assign mul3 = rr[129:96] * rm[31:0];
```

```
// reduction
wire [2:0]   five;
assign five = 5;
wire [33:0]  redmul3;
assign redmul3 = mul3[65:34] * five;
assign rr_rotred = {rr[97:0], 32'b0}
                 + {rr[129:98] * five};

// accumulate
reg [131:0]  acc;

always @(posedge clk)
  begin
     if (reset)
       acc <= 130'h0;
     else
       acc <= start ?
              130'b0 :
              acc
              + mul0
              + {mul1, 32'h0}
              + {mul2, 64'h0}
              + {mul3[33:0], 96'h0}
              + redmul3;
  end
```

```
   // final reduction
   reg [129:0] acc_red;
   always @(*)
     case(acc[131:130])
       2'b00: acc_red = acc;
       2'b01: acc_red = acc + 3'h5;
       2'b10: acc_red = acc + 4'ha;
       2'b11: acc_red = acc + 4'hf;
     endcase

   assign a_out = acc_red;

   // controller
   reg [5:0]    mulctl;
   always @(posedge clk)
     begin
        mulctl <= reset ? 6'h0 :
              {mulctl[4: 0], start};
     end
   assign done = mulctl[5];

endmodule
```

RTL simulation of sequentialized design (5 cyc)

```
# make -f Makefile.poly1305_ws rtlsim
make -f Makefile.poly1305_serial_tv rtlsim
cd poly1305_serial/work && make rtlsim
make[1]: Entering directory '/home/pschaumont/crypto-asic-
opt/poly1305_serial/work'
iverilog -y ../rtl ../sim/tbfile.v
./a.out && mv trace.vcd rtl.vcd && rm -f a.out
VCD info: dumpfile trace.vcd opened for output.
Tag OK: 00000000000000000000000000000000
Tag OK: a927010caf8b2bc2c6365130c11d06a8
Tag OK: 3e867a2296caeff07060e0c5b5f6e536
Tag OK: f00c314c79b8a689af1754d97c7e47f3
Tag OK: 62ebc5bc7cdc08e761eeaa7e9a664145
```

Hardware Synthesis

```
# make -f Makefile.poly1305_ws synthesis

11. Printing statistics.
=== poly1305 ===
   Number of wires:              45826
   Number of wire bits:          46986
   Number of public wires:          15
   Number of public wire bits:    1046
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:              46596
...
     sky130_fd_sc_hd__dfxtp_1      528
...
   Chip area for module '\poly1305': 222823.705600
```

Compared to fully parallel design:

47K standard cells -> 2.93x less
223k square micron area -> 2.79x less

4x more flip-flops

Timing Analysis

```
# make -f Makefile.poly1305_ws gltiming

Startpoint: _92147_ (rising edge-triggered flip-flop clocked by clk)
Endpoint: _92142_ (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max
...
-------------------------------------
         4.89   data required time
        -9.64   data arrival time
-------------------------------------
        -4.75   slack (VIOLATED)
```

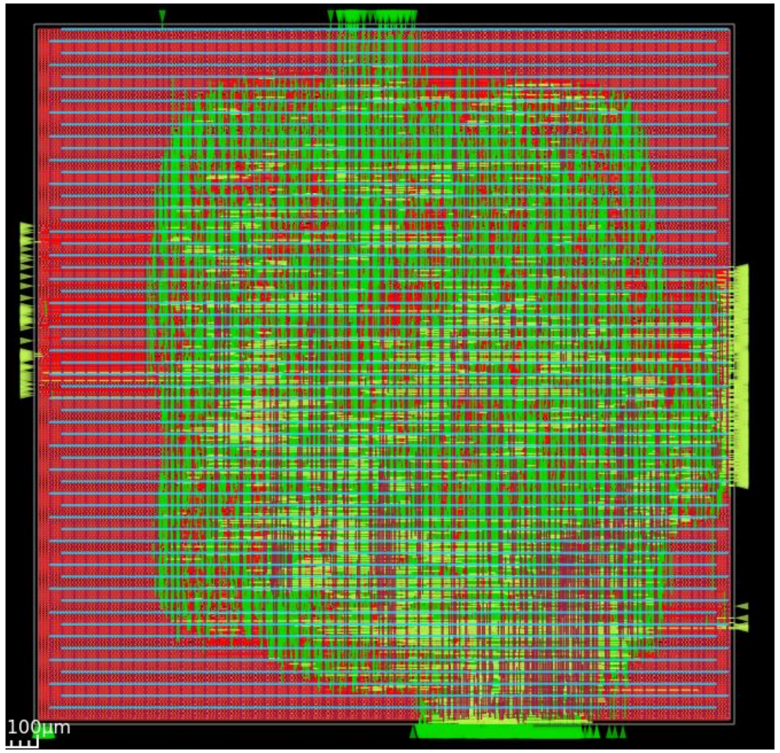
```
# grep -A 3 _92147_ poly1305_serial/work/netlist.v
  sky130_fd_sc_hd__dfxtp_1 _92147_ (
    .CLK(clk),
    .D(_45551_),
    .Q(\single.rm [0])
# grep -A 3 _92142_ poly1305_serial/work/netlist.v
  sky130_fd_sc_hd__dfxtp_1 _92142_ (
    .CLK(clk)
    .D(_45443_),
    .Q(\single.acc [127])
```

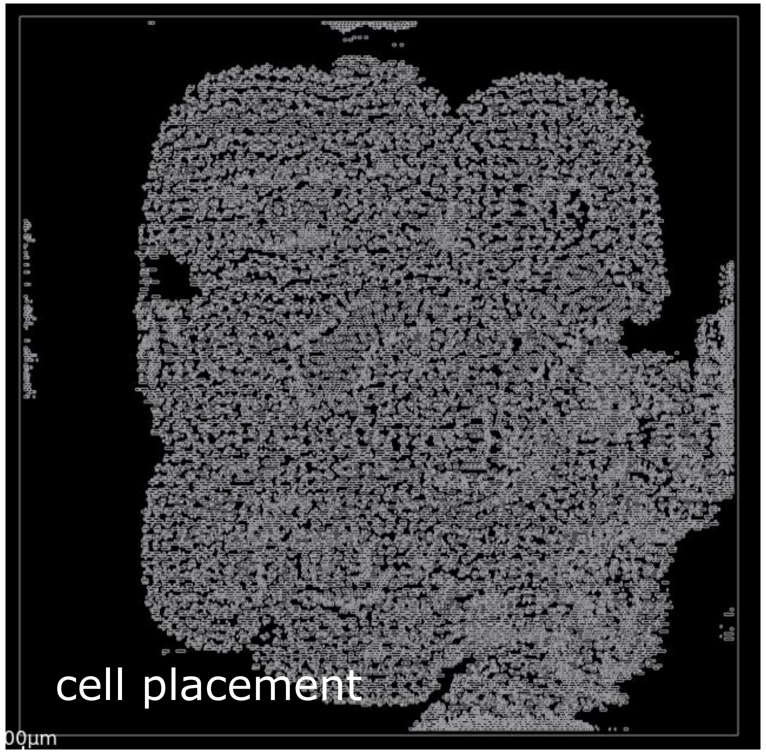
About 3ns faster than fully parallel design; critical path still runs through multiply-accumulate

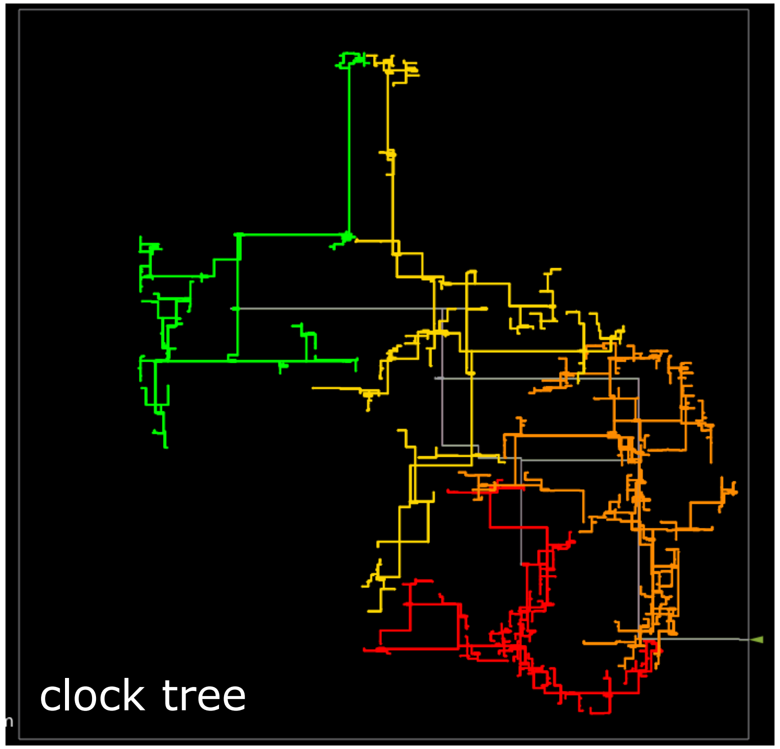
Chip Design

```
# make -f Makefile.poly1305_ws openroad
# make -f Makefile.poly1305_ws chip
# make -f Makefile.poly1305_ws chipgui
```

Area = 800 x 800 = 640000
Post-layout timing:
rm[21]->acc[129]: 20.30 ns!







Even Smaller ?

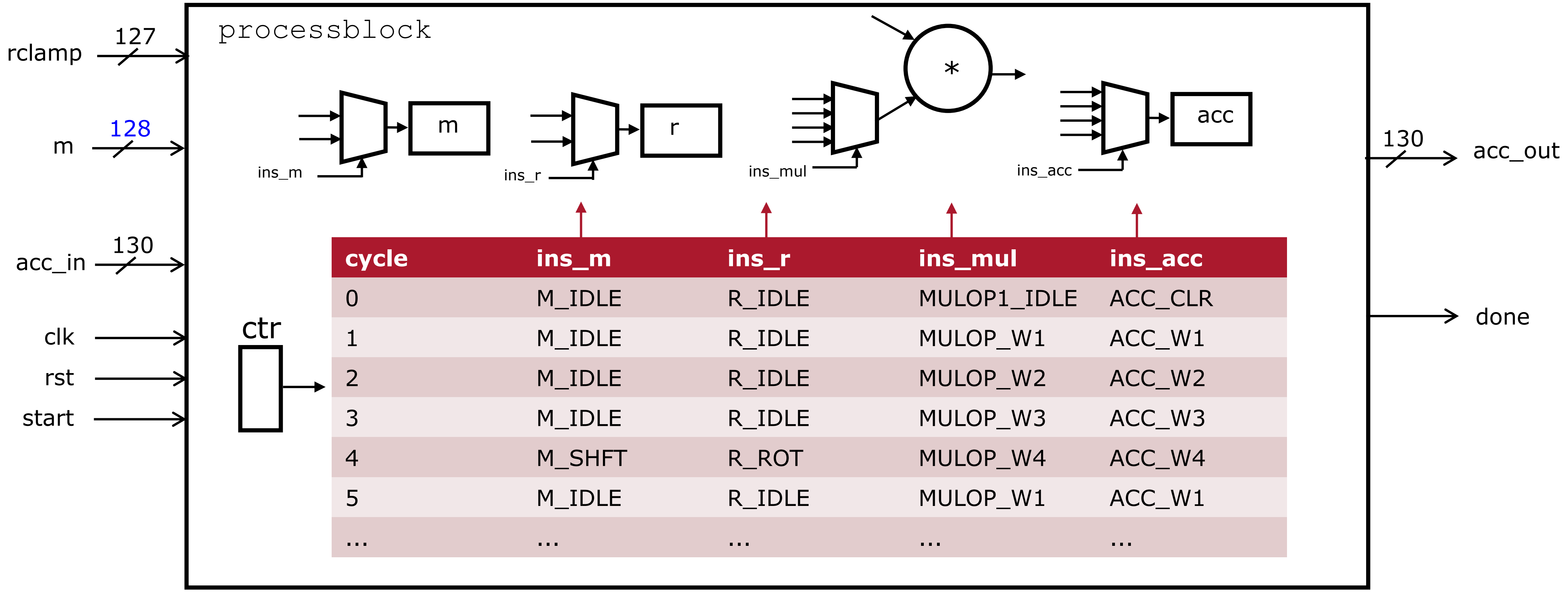
- The design is still relatively big because of four parallel multipliers

```
wire [63:0]  mul0, mul1, mul2;
wire [65:0]  mul3;
assign mul0 = rr[ 31: 0] * rm[31:0];
assign mul1 = rr[ 63:32] * rm[31:0];
assign mul2 = rr[ 95:64] * rm[31:0];
assign mul3 = rr[129:96] * rm[31:0];
```

- Reduce area further by building one physical multiplier and execute four multiplications sequentially
- This will also benefit the hardware for accumulation of partial products
- Full cycle budget for Poly1305 becomes 5 (partial parallel products) x 4 (cycles per partial product) = 20

Controller Design

- On highly multiplexed datapaths, use structured controller desgin



cycle	ins_m	ins_r	ins_mul	ins_acc
0	M_IDLE	R_IDLE	MULOP1_IDLE	ACC_CLR
1	M_IDLE	R_IDLE	MULOP_W1	ACC_W1
2	M_IDLE	R_IDLE	MULOP_W2	ACC_W2
3	M_IDLE	R_IDLE	MULOP_W3	ACC_W3
4	M_SHFT	R_ROT	MULOP_W4	ACC_W4
5	M_IDLE	R_IDLE	MULOP_W1	ACC_W1
…	…	…	…	…

Sequentialized Design (20 cycles) - controller

```
module processblock(input              reset,
                    input              clk,
                    input   [127:0]    r,
                    input   [128:0]    m,
                    input   [129:0]    a_in,
                    output  [129:0]    a_out,
                    input              start,
                    output             done
                    );

   reg [131:0]             rm;
   reg [129:0]             rr;
   wire  [129:0]           rr_rotred;

   wire  [130:0]           initialsum;
   wire  [129:0]           initialsumred;
   wire  [2:0]   five;
   assign five = 5;

   assign initialsum = m + a_in;
   assign initialsumred = initialsum[130] ?
                          initialsum[129:0] + five :
                          initialsum[129:0];

   reg [6:0]    ctlstate;
   localparam [6:0] INITSEQ   = 7'd1;
   localparam [6:0] FINALSEQ  = 7'd21;
```

```
// controller
   always @(posedge clk)
     begin
        if (reset)
          ctlstate <= 7'h0;
        else
          begin
             ctlstate <= (ctlstate == FINALSEQ) ? 7'h0 :
                         start ? INITSEQ :
                        (ctlstate > 7'h0) ? ctlstate + 7'h1 :
                         ctlstate;
          end
     end // always @ (posedge clk)

   assign done = (ctlstate == FINALSEQ);
```

Sequentialized Design (20 cycles) – ucode rom

```
reg [2:0]        ins_acc;
reg [2:0]        ins_mulop1;
reg              ins_r;
reg              ins_m;

// microcode rom
always @(*)
  begin
     ins_acc = ACC_IDLE;
     ins_mulop1 = MULOP1_IDLE;
     ins_r = R_IDLE;
     ins_m = M_IDLE;
     case(ctlstate)
       0:  begin ins_acc = ACC_CLR;  ins_mulop1 = MULOP1_IDLE; ins_r = R_IDLE; ins_m = M_IDLE; end
       1:  begin ins_acc = ACC_W1;   ins_mulop1 = MULOP1_W1;   ins_r = R_IDLE; ins_m = M_IDLE; end
       2:  begin ins_acc = ACC_W2;   ins_mulop1 = MULOP1_W2;   ins_r = R_IDLE; ins_m = M_IDLE; end
       3:  begin ins_acc = ACC_W3;   ins_mulop1 = MULOP1_W3;   ins_r = R_IDLE; ins_m = M_IDLE; end
       4:  begin ins_acc = ACC_W4;   ins_mulop1 = MULOP1_W4;   ins_r = R_ROT;  ins_m = M_SHFT; end
       5:  begin ins_acc = ACC_W1;   ins_mulop1 = MULOP1_W1;   ins_r = R_IDLE; ins_m = M_IDLE; end
       6:  begin ins_acc = ACC_W2;   ins_mulop1 = MULOP1_W2;   ins_r = R_IDLE; ins_m = M_IDLE; end
       7:  begin ins_acc = ACC_W3;   ins_mulop1 = MULOP1_W3;   ins_r = R_IDLE; ins_m = M_IDLE; end
       8:  begin ins_acc = ACC_W4;   ins_mulop1 = MULOP1_W4;   ins_r = R_ROT;  ins_m = M_SHFT; end
       ...
       21: begin ins_acc = ACC_IDLE; ins_mulop1 = MULOP1_IDLE; ins_r = R_IDLE; ins_m = M_IDLE; end
     endcase
  end
```

Sequentialized Design (20 cycles) – r,m,mul

```
// operations on message register and r register
  always @(posedge clk)
    begin
       if (reset)
         begin
            rm       <= 130'h0;
            rr       <= 128'h0;
         end
       else
         begin
            rm <= start ? initialsumred :
                  (ins_m == M_SHFT) ? {32'h0, rm[129:32]} :
                  rm;
            rr <= start ? r :
                  (ins_r == R_ROT) ? {rr[97:0], 32'b0} + {rr[129:98] * five} :
                  rr;
         end
    end
```

```
// multiplier
reg [33:0]   mulop1;
reg [31:0]   mulop2;
reg [65:0]   mulres;

always @(*)
  mulres = mulop1 * mulop2;

always @(*)
  begin
     mulop2 = rm[31:0];
     mulop1 = 34'h0;
     case (ins_mulop1)
       3'h0 : mulop1 = 34'h0;
       3'h1 : mulop1 = {2'b0,rr[31:0]};
       3'h2 : mulop1 = {2'b0,rr[63:32]};
       3'h3 : mulop1 = {2'b0,rr[95:64]};
       3'h4 : mulop1 = rr[129:96];
     endcase
  end
```

Sequentialized Design (20 cycles) – acc & red

```
// operations on accumulator
  reg [131:0]  acc;
  always @(posedge clk)
    begin
       if (reset)
         begin
            acc <= 132'h0;
         end
       else
         begin
            case (ins_acc)
              ACC_IDLE: acc <= acc;
              ACC_CLR:  acc <= 132'h0;
              ACC_W1:   acc <= acc + mulres;
              ACC_W2:   acc <= acc + {mulres, 32'h0};
              ACC_W3:   acc <= acc + {mulres, 64'h0};
              ACC_W4:   acc <= acc + {mulres[33:0], 96'h0} +
                                       (mulres[65:34] * five);
            endcase
         end
    end
```

```
// final reduction
   reg [129:0] acc_red;
   always @(*)
     case(acc[131:130])
       2'b00: acc_red = acc;
       2'b01: acc_red = acc + 3'h5;
       2'b10: acc_red = acc + 4'ha;
       2'b11: acc_red = acc + 4'hf;
     endcase

   assign a_out = acc_red;

endmodule
```

RTL simulation of sequentialized design (20 cyc)

```
# make -f Makefile.poly1305_ws_w32 rtlsim
...
VCD info: dumpfile trace.vcd opened for output.
Tag OK: 00000000000000000000000000000000
Tag OK: a927010caf8b2bc2c6365130c11d06a8
Tag OK: 3e867a2296caeff07060e0c5b5f6e536
Tag OK: f00c314c79b8a689af1754d97c7e47f3
Tag OK: 62ebc5bc7cdc08e761eeaa7e9a664145
```

Hardware Synthesis

```
# make -f Makefile.poly1305_ws_w32 synthesis

11. Printing statistics.

=== poly1305 ===

   Number of wires:              22059
   Number of wire bits:          23253
   Number of public wires:          16
   Number of public wire bits:    1081
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:              22830
...
   sky130_fd_sc_hd__dfxtp_1      529
...
   Chip area for module '\poly1305': 112541.686400
```

Compared to fully parallel design:

23K standard cells -> 6.3x less
112k square micron area -> 5.59x less

4x more flip-flops

Timing Analysis

```
# make -f Makefile.poly1305_ws_w32 gltiming

Startpoint: _44871_ (rising edge-triggered flip-flop clocked by clk)
Endpoint: _44567_ (rising edge-triggered flip-flop clocked by clk)

           4.88   data required time
          -7.91   data arrival time
---------------------------------------------------------
          -3.03   slack (VIOLATED)
```

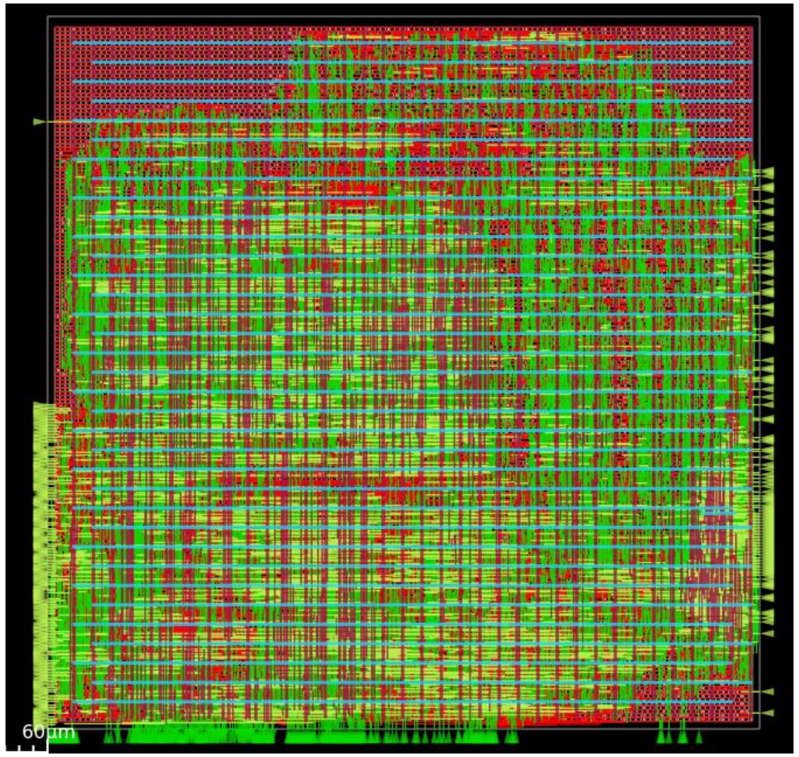
```
# grep -A 3 _92147_ poly1305_serial/work/netlist.v
 sky130_fd_sc_hd__dfxtp_1 _44871_ (
    .CLK(clk),
    .D(_21781_),
    .Q(\single.ctlstate [5])
# grep -A 3 _92142_ poly1305_serial/work/netlist.v
 sky130_fd_sc_hd__dfxtp_1 _44567_ (
    .CLK(clk),
    .D(_21768_),
    .Q(\single.acc [93])
```

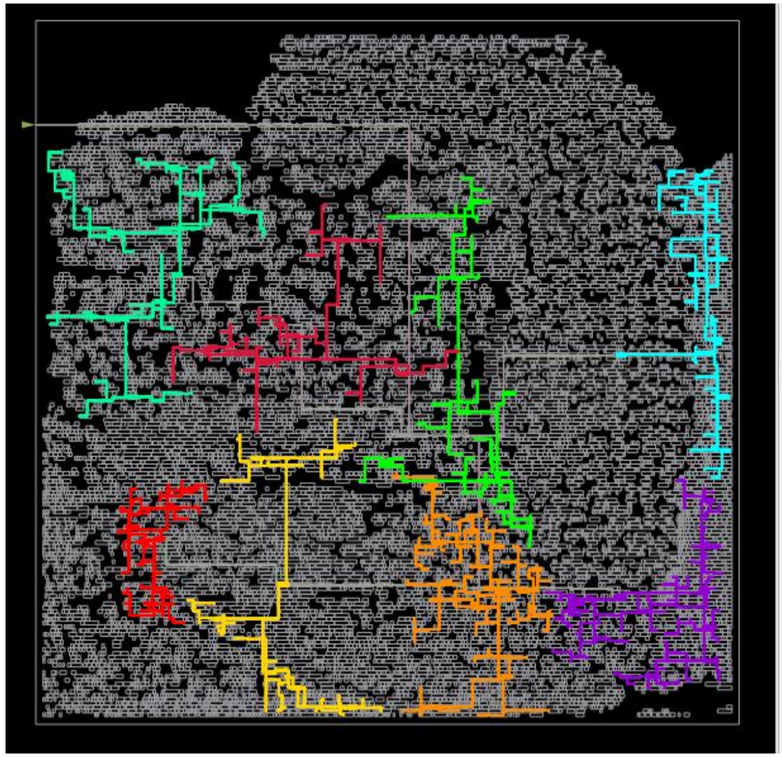
About 5ns faster than fully parallel design; critical path runs through controller into multiply-accumulate

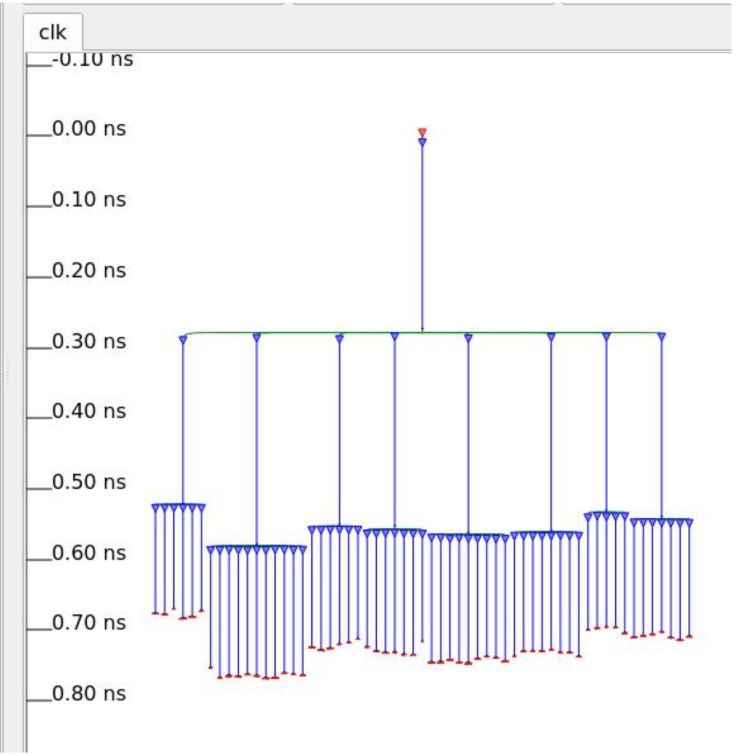
Chip Design

```
# make -f Makefile.poly1305_ws_w32 openroad
# make -f Makefile.poly1305_ws_w32 chip
# make -f Makefile.poly1305_ws_w32 chipgui
```

Area = 500 x 500 = 250000
Post-layout timing:
ctlstate[1]->acc[131]: 17.45 ns!



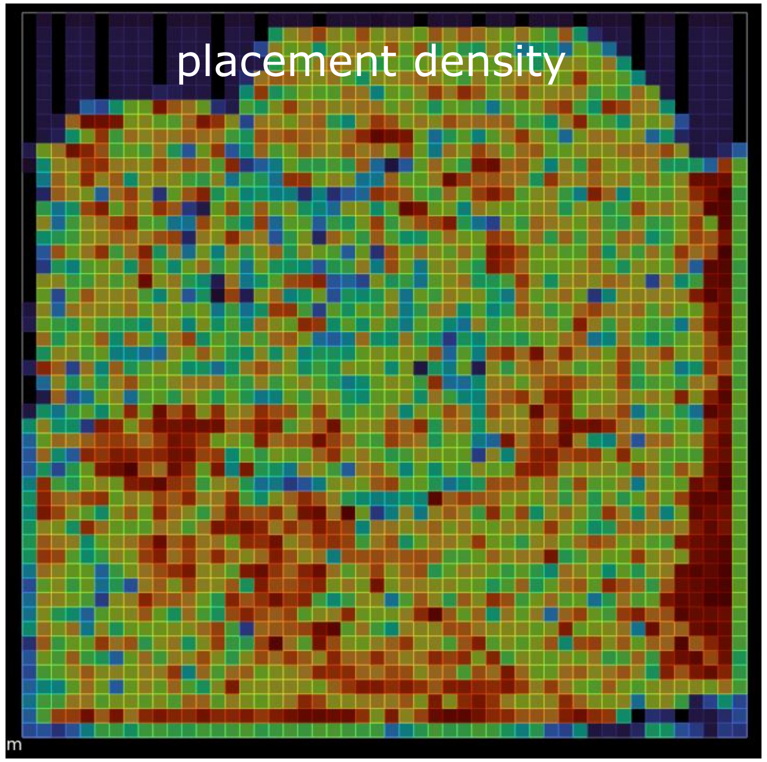


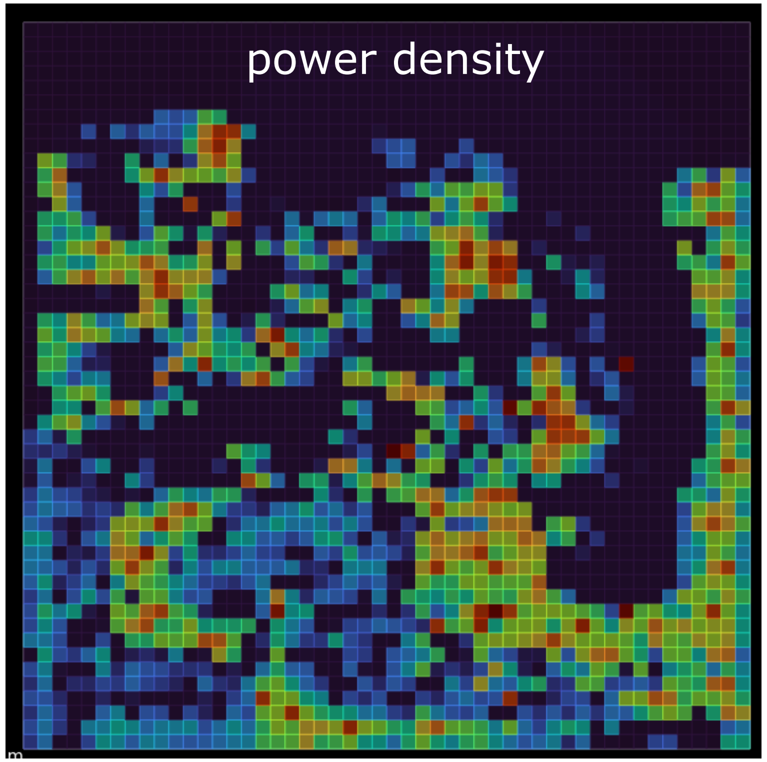


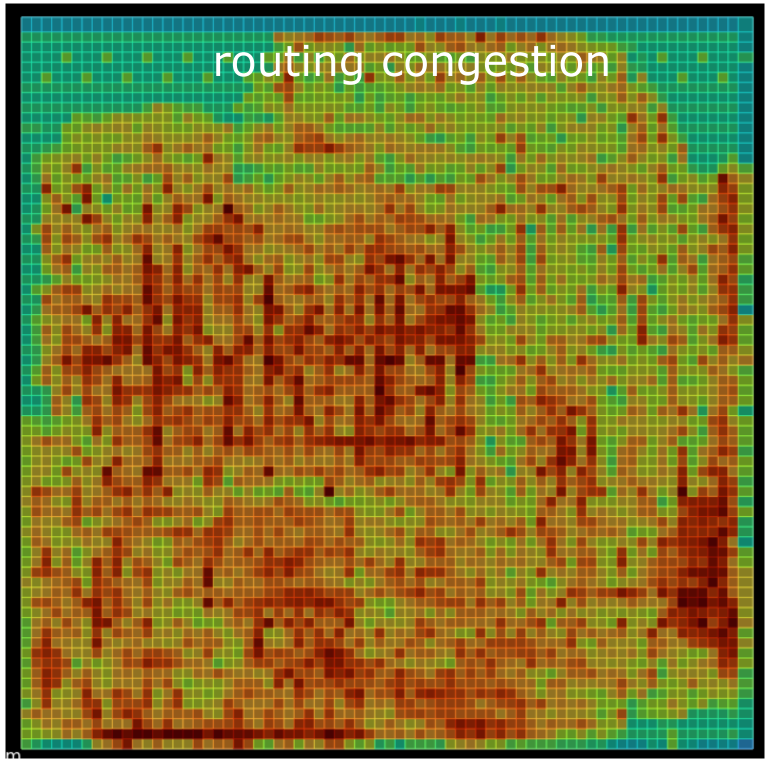
Chip Design

```
# make -f Makefile.poly1305_ws_w32 openroad
# make -f Makefile.poly1305_ws_w32 chip
# make -f Makefile.poly1305_ws_w32 chipgui
```

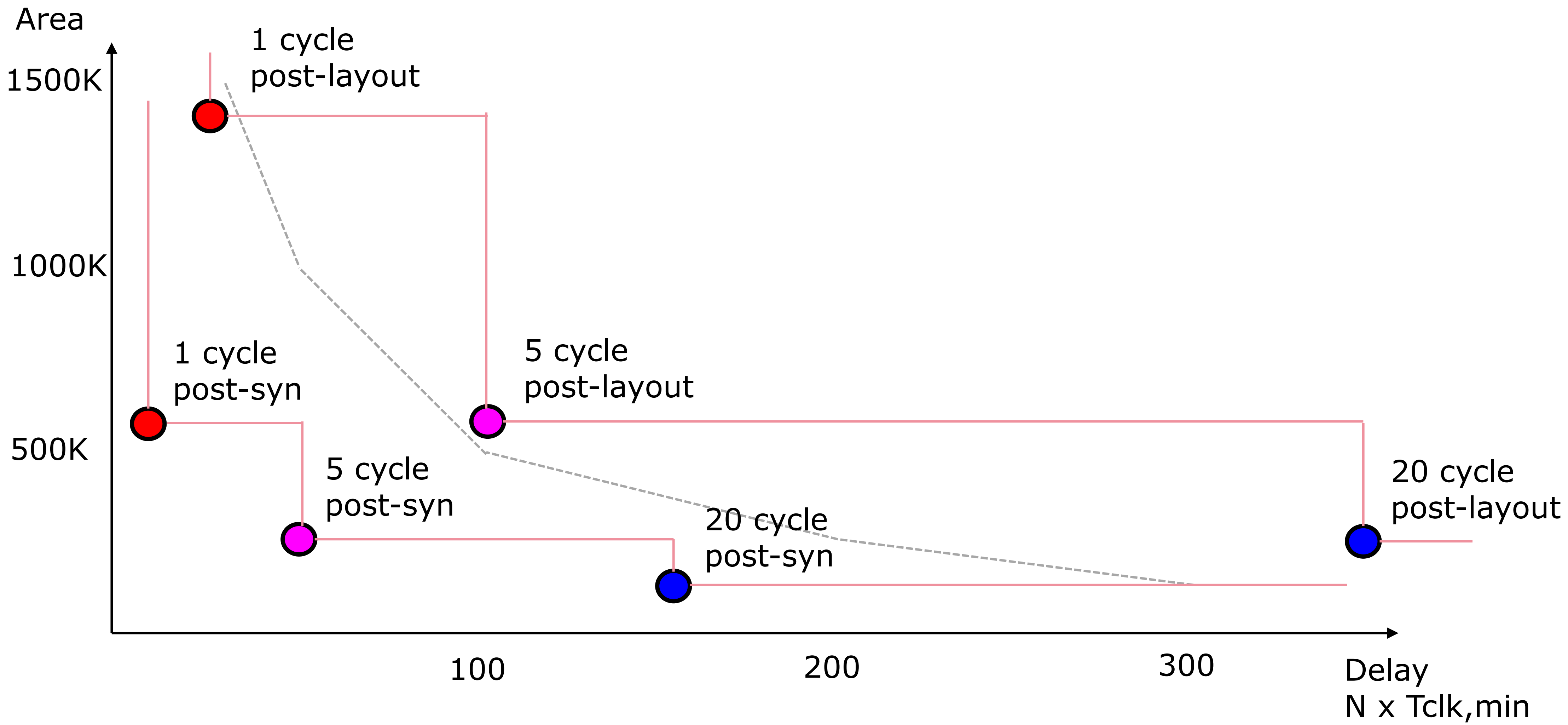
Area = 500 x 500 = 250000
Post-layout timing:
ctlstate[1]->acc[131]: 17.45 ns!







Conclusions



The big picture

13:30	• ASIC Design Flow • Example • Modular Multiplication
14:15	• Hands-on
15:00	

15:30	• Poly1305 in Hardware
16:00	• Hands-on
17:00	

Assignment (1)

1. Log in to the design server
2. Create a testvector to compute the Poly1305 MAC for the message
   Physical Attack Resistance of Cryptographic Algorithms
   and using an 0xFFFFFFFF...FFFFF key (all-1's)
   Hint: MAC bytes in hardware order are
   4fc3cb2a7b393c3b715400929ad50c45
   Hint: Modify polygen.c to create an adjusted testvector

Assignment (2)

3. Compare the number of flipflops needed for poly1305_bp and find what lines of RTL code are responsible for those registers.

   Hint: Synthesize the design with
   make –f Makefile.poly1305_bp synthesis

   Hint 2: look for, and count, cells of type
   sky130_fd_sc_hd__dfxtp_...

   Hint 3: the RTL code is in poly1305_bp/rtl/{poly1305.v, processblock.v}

Assignment (3)

4. Modify the logic optimization script for poly1305_bp from abc.speed to abc.area
   Hint: in Makefile.poly1305_bp, change

   export ABC = abc.speed

   into

   export ABC = abc.area

   Compare the number of cells (make synthesis) and the timing (make gltiming) for each script. What is the % variation?

	abc.speed	abc.area
timing	12.29 ns	?
cells	138,841	?

Note: yosys is unaware of technology dependent factors, and does not perform logic optimization. yosys-internal tools such as abc perform logic synthesis (picking gates for logic functions) and offer limited optimization. However, the abc script alone is not a full optimization strategy; register placement, clock distribution, fanout optimization should be considered too. For simplicity, we skip those.

Assignment (4)

5. Compare the number of flipflops needed for poly1305_ws and compare that to the number of flipflops needed for poly1305_ws_w32. Is it higher or lower? Why?
   Hint: look for, and count, cells of type sky130_fd_sc_hd__dfxtp_...

   Hint 2: synthesize the designs with
   ```
   make –f Makefile.poly1305_ws synthesis
   make –f Makefile.poly1305_ws_w32 synthesis
   ```

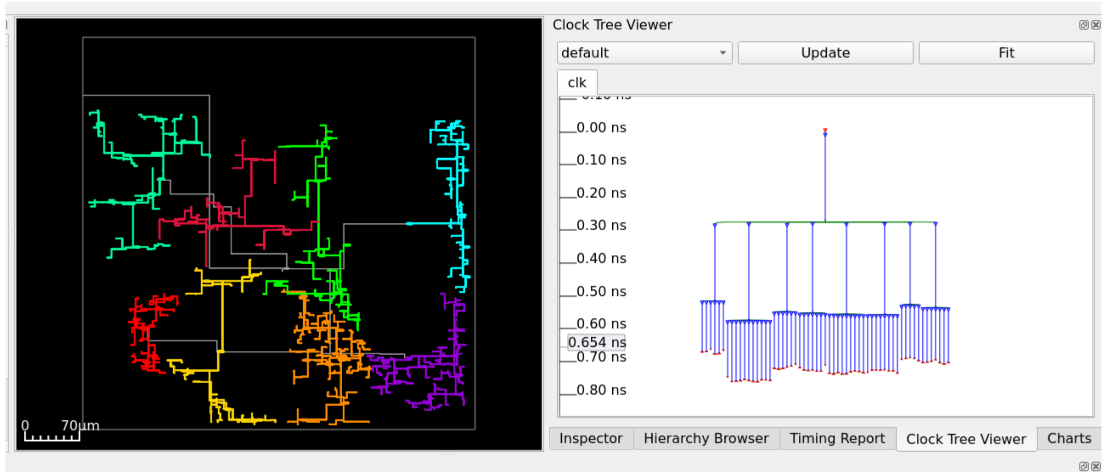
   Hint 3: use
   ```
   grep –A 3 df poly1305_ws/work/netlist.v
   ```
   to learn about each individual register (be prepared to capture the output)

Assignment (5)

7. Take one of the designs poly1305_ws_w32 or poly1305_ws and visualize the clock tree in the layout
   Hint: Use

   ```
   make –f Makefile.poly1305_ws_w32 chipgui
   ```

   Inside of the gui, use Windows – Clock Tree Viewer
   Select the CLock Tree Viewer Tab, click Update
   Turn off layers as needed to make the view better



Think about the shape of the clock tree.
What do the colors mean?
Why are the colors covering different regions on the chip?

Hint: See what you can find about 'Clock Skew' and 'Clock Distribution Networks'

Assignment (6)

6. Modify (shrink) the chip core area for poly1305_ws_w32. How small can you make the core area so that you still obtain a layout?

   Hint: Modify poly1305_ws_w32/chip/config.mk

   Hint2: Defaut layout takes 15 min. It will explode if you make the layout problem too hard for the tools. So first, think through the options and carefully select a target

   Hint3: Expert users! Check https://openroad-flow-scripts.readthedocs.io/en/latest/user/FlowVariables.html for additional variables you can set in config.mk

   What is the utilization of the smaller core?

Additional Reading and Links

Tool Documentation

- iVerilog (Verilog Simulation) https://steveicarus.github.io/iverilog/
- yosys (Logic Synthesis) https://yosyshq.readthedocs.io/en/latest/
- OpenSTA (Static Timing Analysis) https://github.com/The-OpenROAD-Project/OpenSTA/blob/master/doc/OpenSTA.pdf
- OpenROAD (ASIC Backend) https://openroad.readthedocs.io/en/latest/
- Skywater 130 PDK (Standard Cells) https://skywater-pdk.readthedocs.io/en/main/

Hardware Optimization for ASIC

- Good motivation why hardware design remains important
  - C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. LHampson, D. Sanchez, et al., "There's plenty of room at the Top: What will drive computer performance after Moore's law?" in Science, American Association for the Advancement of Science, vol. 368, no. 6495, 2020. https://doi.org/10.1126/science.aam9744
- Bitserial/Digit-serial Design is a vintage hardware design technique. Here's a ref.
  - R. Hartley, K. Parhi, "Digit-serial Computation," Springer 1995. A good reference on bit-serial design. https://link.springer.com/book/10.1007/978-1-4615-2327-7
- In depth discussion on the automation that drives openROAD
  - A. Kahng, J. Lienig, I. L. Markov, J. Hu, "VLSI Physical Design," Springer 2011.

Hardware Optimization for ASIC

- Describes HW design considerations from a power perspective
  - Rabaey, “Low Power Design Essentials”, https://link.springer.com/book/10.1007/978-0-387-71713-5
- Good discussion on performance factors in ASIC
  - David G. Chinnery, Kurt Keutzer: Closing the Gap Between ASIC and Custom - Tools and Techniques for High-Performance ASIC Design. Springer 2004, ISBN 978-1-4020-7113-3, pp. I-XIV, 1-414
- In depth discussion on the automation that drives openROAD
  - A. Kahng, J. Lienig, I. L. Markov, J. Hu, “VLSI Physical Design,” Springer 2011.
- To learn about clock distribution basics
  - E. G. Friedman, "Clock distribution networks in synchronous digital integrated circuits," in Proceedings of the IEEE, vol. 89, no. 5, pp. 665-692, May 2001, doi: 10.1109/5.929649.