Embedded Systems
Hardware Design

RTL implementation

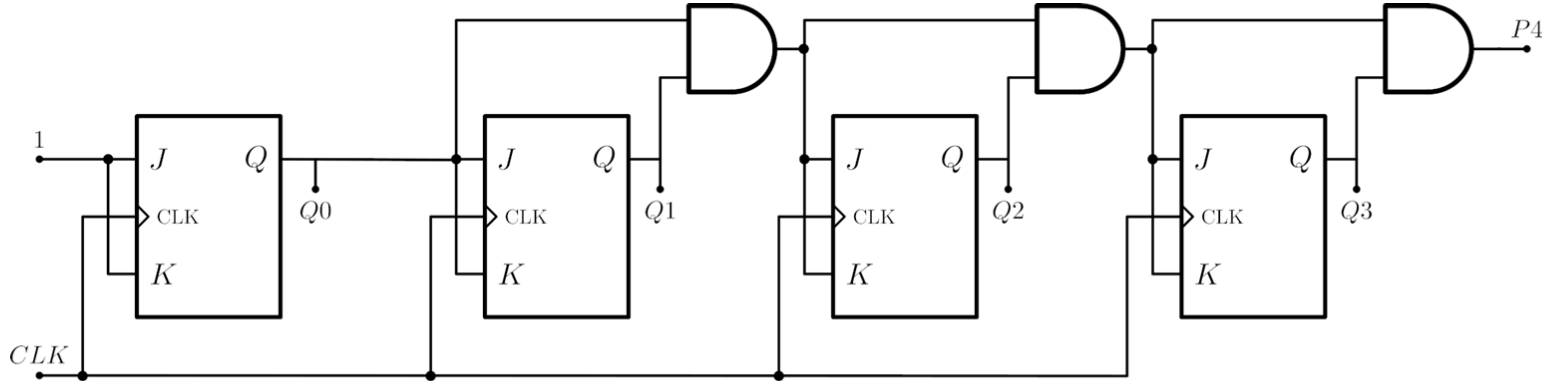
RTL architecture

Register Transfer Level (RTL) description is a lower level of abstraction than behavioral description but higher level abstraction than netlist description of a circuit design. As shown in last section, RTL description is written using either Verilog or VHDL (using any of their flavors). In RTL description, circuit is described in terms of registers (flip-flops or latches) and the data is transferred between them using logical operations (combinational logic, if needed) and that is why the nomenclature: Register Transfer Level (RTL).

Source Principles of VLSI RTL Design a Practical Guide

4-bit counter with overflow

Example of RTL architecture is 4-bit counter presented below. As you can see between each logic gate you can find JK Flip-Flop. The structure of this counter results from the principle of its operation. The next slide shows the code describing the operation of the counter in verilog using the RTL description.



RTL implementation

```
module counter( input clk,
                output [3:0] q,
        output p);
    wire [3:0] q_tmp;
    wire [1:0] y;
    jk_ff   jk0 ( .j(1'b1),
                  .k(1'b1),
                  .clk(clk),
                  .q(q_tmp[0]));

    jk_ff   jk1 ( .j(q_tmp[0]),
                  .k(q_tmp[0]),
                  .clk(clk),
                  .q(q_tmp[1]));
    and_2    a0 ( .a(q_tmp[0]),
                  .b(q_tmp[1]),
                  .y(y[0]));

    jk_ff   jk2 ( .j(y[0]),
                  .k(y[0]),
                  .clk(clk),
                  .q(q_tmp[2]));
    and_2   a1 (  .a(y[0]),
                  .b(q_tmp[2]),
                  .y(y[1]));

    jk_ff   jk3 ( .j(y[1]),
                  .k(y[1]),
                  .clk(clk),
                  .q(q_tmp[3]));
    and_2   a2 (  .a(y[1]),
                  .b(q_tmp[3]),
                  .y(p));
    assign q=q_tmp;
endmodule
```

Behavioral implementation

```
module counter( input clk,
            output [3:0] q,
        output p);
  reg [3:0] count;
  always@(posedge clk)
  begin
count <= count + 1;
  end
  assign q = count;
  assign p = (count==15) ? 1'b1 : 1'b0;
endmodule
```

Testbench

```
module tb_counter;


reg clk;
wire [3:0]q;
wire p;


counter u0 ( .clk(clk),
 .q(q),
         .p(p));
initial begin
$dumpfile("test.vcd");
$dumpvars;
clk = 1;
#320 $finish;
end
always #10 clk=~clk;always #20 $display("%d %d",p,q);
endmodule
```

Simulation

Compiling tb.v file:

```
verilator --trace --binary -j 0 tb.v
```

Running the simulation:

```
./obj_dir/Vtb
```

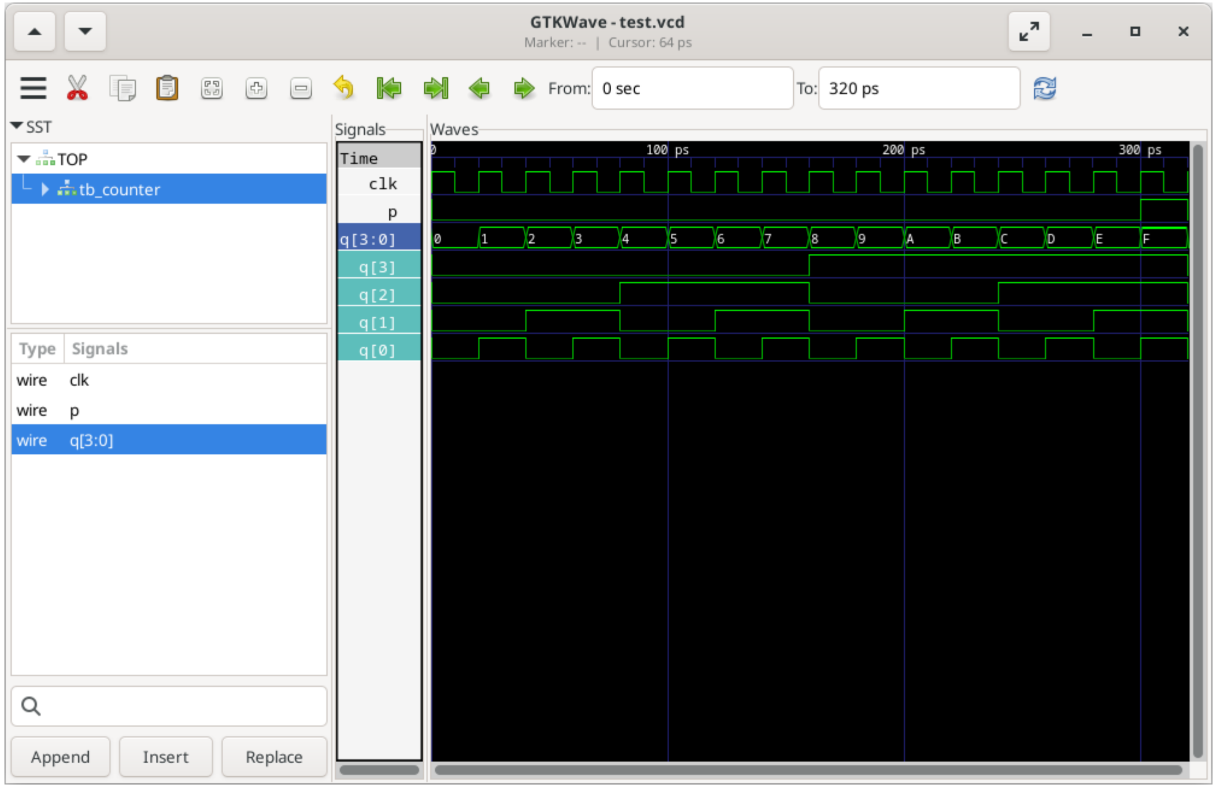
Results of the counter simulation

```
0  0
0  1
0  2
0  3
0  4
0  5
0  6
0  7
0  8
0  9
0 10
0 11
0 12
0 13
0 14
1 15
- tb.v:16: Verilog $finish
```

```
0  0
0  1
0  2
0  3
0  4
0  5
0  6
0  7
0  8
0  9
0 10
0 11
0 12
0 13
0 14
1 15
- tb.v:16: Verilog $finish
```

Results of the counter simulation

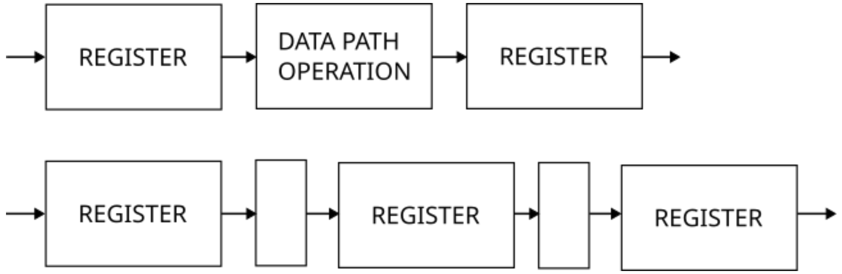
The image below presents results of simulation in the time domain.



Type	Signals
wire	clk
wire	p
wire	q[3:0]

Pipelining

The idea behind pipelining is to take a large data-path operation that is currently executed over one clock cycle and break it up into smaller operations that are executed over multiple shorter clock cycles.



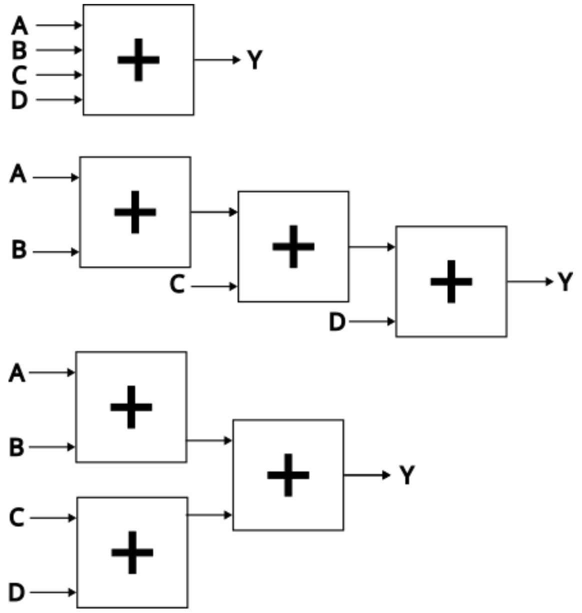
The image above shows how the pipelining looks like. In the first line of the image there is logical block with with two registers. This block slows down the clock frequency for example to 500 MHz. We assume that the operations in the middle block can be splitted into two blocks. This operation should increase clock frequency for example to 1GHz.

Parallel computing

Assume that we want to implement Y = A + B + C + D function in FPGA.

The adder can be presented as block with 4 inputs and 1 output.

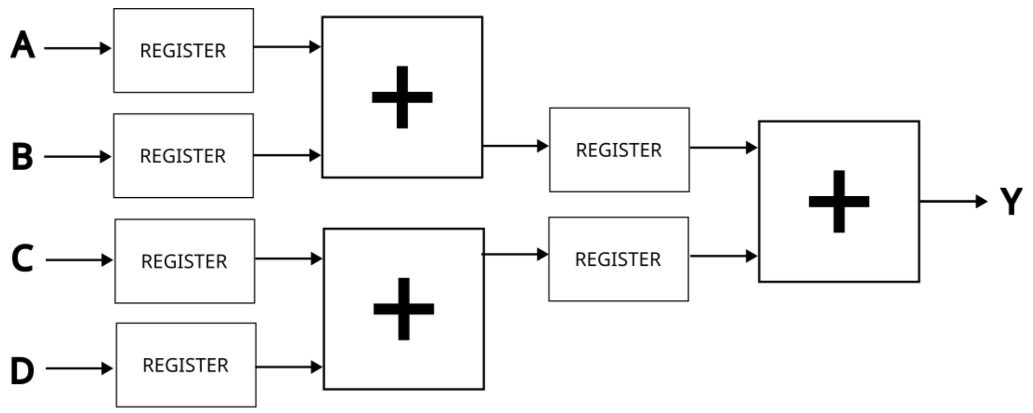
Making adder with 4 input could be difficult to realize in FPGA. We can decomposite function Y and present it as Y=((A+B)+C)+D but it is still a sequential calculations. There is other decomposition Y=(A+B)+(C+D). (A+B) and (C+D) can be performed in parallel.



Pipelined adder

In this way we can achieve a pipelined adder circuit.

The adder from the previous slide could add 4 signals in 1 clock's cycle. The adder presented below can do the same calculations in 2 cycles but it can do it with higher clock's frequency.



Application of RTL design

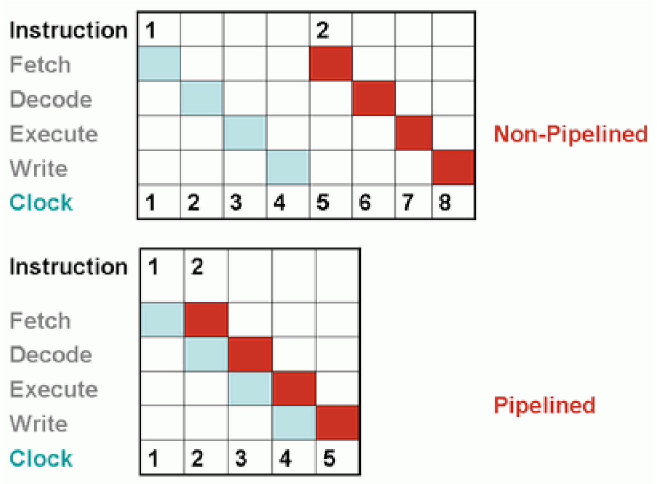
- GPU performs calculations on the data stream, pipelining is an excellent solution for these types of applications. The use of pipelining and RTL architecture increases GPU throughput by increasing the clock frequency
- NPU also performs computing on the data stream. Real-time applications like objects recognition or object tracking require high throughput and a fast clock.
- CPU is specific case. It uses some instructions that use piplining but pipelining is used used by the processor in instruction handling

Pipelining in CPU

In non-pipelined version of CPU, as you can see, the CPU is waiting to end instruction before starting the next one.

Pipelined version doesn’t wait to end of the last instruction.

It starts handle the instruction after instruction in every cycle.



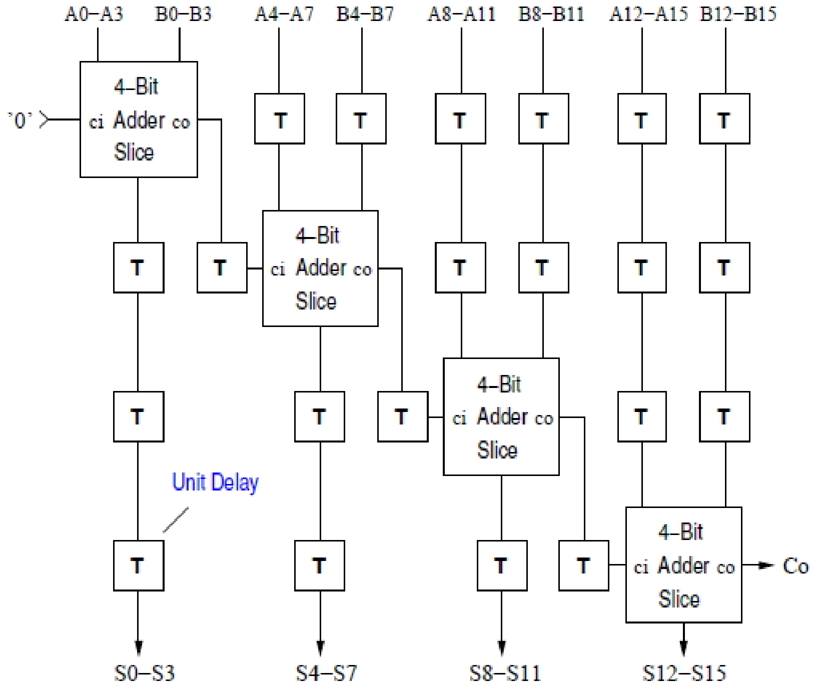
Source https://stackpointer.io/hardware/how-pipelining-improves-cpu-performance/113/

Pipelined adder

16 bit adder can be implemented with 4-bit adder based on slice.

This pipeline has 4 stages, so need

four cycles to give the result.



Source https://vlsigyan.com/pipeline-adder-verilog-code/

Advantages of RTL implementation

- Improves throughput
- Easy to verify
- Easy to debug
- Easy to understand
- Perfect for data streams

Disadvantages of RTL implementation

- Loss of initial clock cycles
- Requires algorithm/problem decomposition
- Pipelining is not suitable for single data
- More data paths in one block require synchronization

Synchronization

Look at the problem Y=A*B+C.

How to implement it using pipelining?

Because the operations are not independent, multiplication must be done first and then addition. We cannot directly send C signal to adder because we don’t know the result of multiplication. So signal C must be synchronized with the result of multiplication. It can be done by adding additional register that will delay signal C in data path.

Increasing frequency

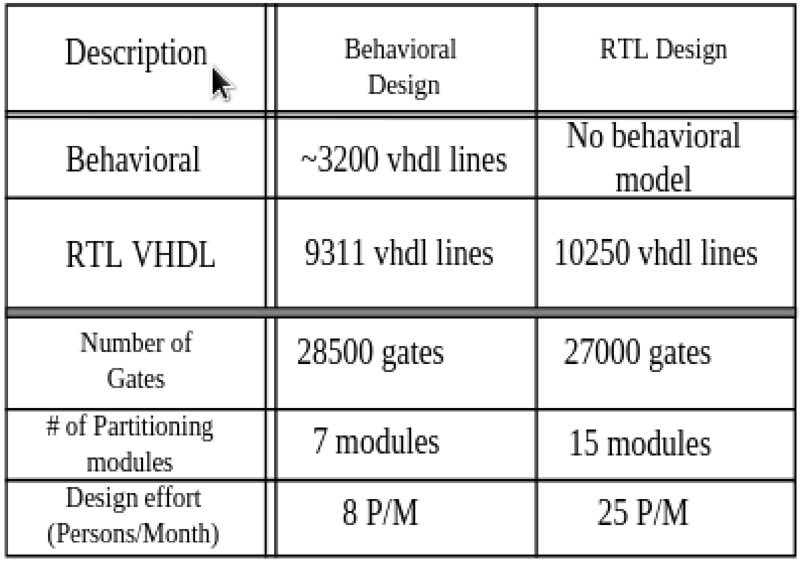
If there is more than one path in the design and one generates a pipeline and the other does not, we can optimize the circuit. Assuming that one of these paths determines the clock speed, you can divide the activities performed on this path into stages to create a pipeline. It should increase clock frequency.

Avoiding mistakes

- Try to implement and keep digital blocks as small as posible
- Avoid behavioral architecture
- Use behavioral description to describe small blocks like MUXes, logic gates, etc. It doesn’t make sense to describe them as logical equations
- Before you use pipelining, think about whether it is profitable. Otherwise, you may waste unnecessary time.

Behavioral vs RTL synthesis

In some experiments behavioral design gives better synthesis results tha RTL design.



Description	Behavioral Design	RTL Design
Behavioral	~3200 vhdl lines	No behavioral model
RTL VHDL	9311 vhdl lines	10250 vhdl lines
Number of Gates	28500 gates	27000 gates
# of Partitioning modules	7 modules	15 modules
Design effort (Persons/Month)	8 P/M	25 P/M

Source Comparing RTL and behavioral design methodologies in the case of a 2M-transistor ATM shaper

Project Simulation Phases

Depending upon the amount of design innovation in a new project, theemphasis on the different simulation phases may vary, but generally they consist of

- debugging.
- regression.
- recreating hardware problems.
- To this set of usual simulation phases, we add simulation performance profiling between the debugging and regression phases.

Source Principles of Verifiable RTL Design

Debugging Phase

As engineers begin their first simulations of their designs, simulations invariably fail to pass their tests due to design bugs. The number of bugs is generally proportional to the size of the simulated design and the amount ofinnovation applied in the new design.

Source Principles of Verifiable RTL Design

Regression Phase

The regression simulation phase begins when a design passes nearly all of its tests. During this phase, the project uses as much computing horsepower as it can find to run as many different directed, directed random, and random simulations as it can on all the possible configurations of the simulated model.

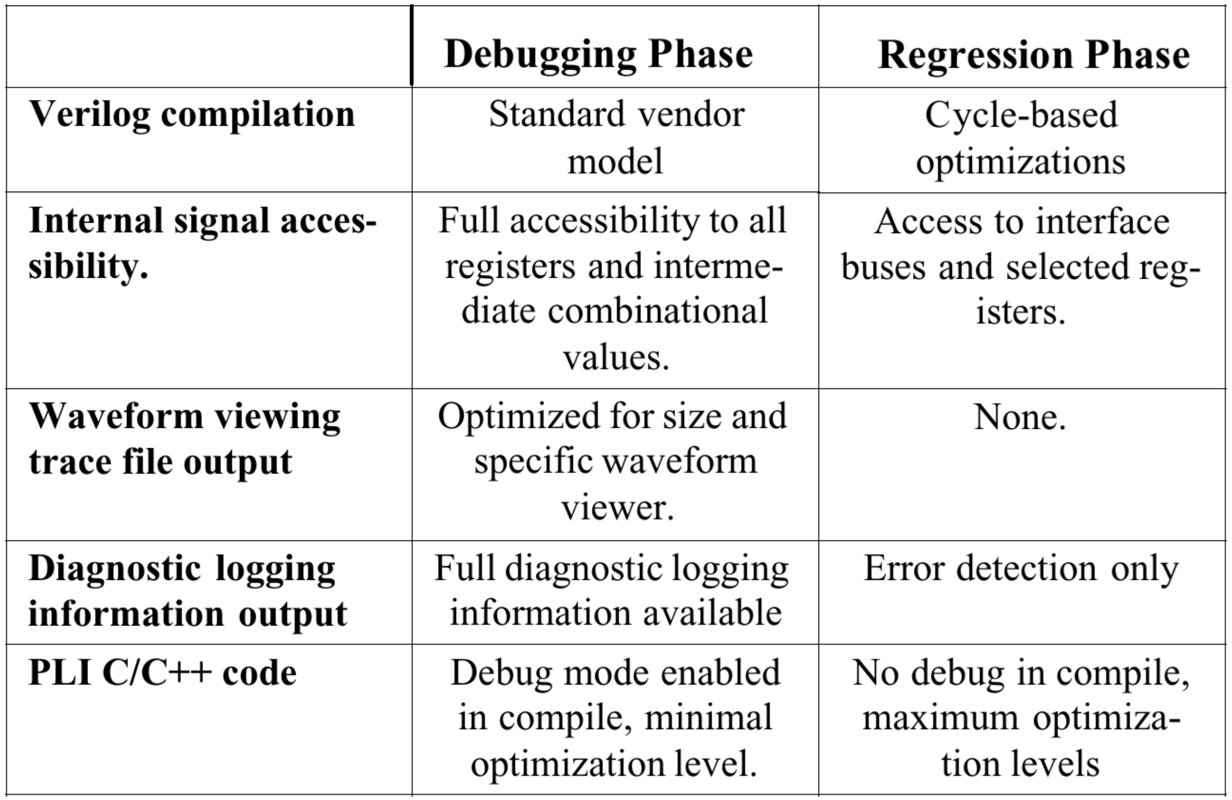
Source Principles of Verifiable RTL Design

Recreating Hardware Problems

An important application of RTL simulation is recreating design problems that show up in the hardware lab testing. The first step is crafting the test that duplicates the hardware problem in simulation. The next step is devising the logic change that fixes the problem, then running the test to show that the problem is fixed for that test. The test can then join the regression suite of tests.

Source Principles of Verifiable RTL Design

Debugging and regression model differences



	Debugging Phase	Regression Phase
Verilog compilation	Standard vendor model	Cycle-based optimizations
Internal signal accessibility.	Full accessibility to all registers and intermediate combinational values.	Access to interface buses and selected registers.
Waveform viewing trace file output	Optimized for size and specific waveform viewer.	None.
Diagnostic logging information output	Full diagnostic logging information available	Error detection only
PLI C/C++ code	Debug mode enabled in compile, minimal optimization level.	No debug in compile, maximum optimization levels

Source Principles of Verifiable RTL Design