DOG: Dynamic Operation Grouping at Synthesis

Synthesizable Parametric VLIW Core

AndOrlando, theokrueger

May 6, 2026

Abstract

Very Long Instruction Word (VLIW) is an architectural concept offering a unique approach to exploiting Instruction-Level Parallelism (ILP) in executable programs. As opposed to the Superscalar approach, which uses special hardware to find ILP at runtime (effectively) nondeterministically, VLIW offloads finding ILP onto the compiler. This allows VLIW hardware to consume less power and area for comparable performance to Superscalar cores, at the cost of a more complex compilation step[5]. Our work, called Dynamic Operation Grouping at Synthesis (DOG), offers insight into the scaling properties of VLIW cores with regards to area, power draw, and execution time with respect to the wordlength (number of parallel instructions), on a custom 8-bit parametric VLIW core.

Introduction/Motivation

We believe that VLIW is the next big thing in general purpose computing[6]. The massive theoretical speedups only at the cost of compiler complexity are too great to ignore. While work has been published on the benefits of VLIW in regards to design time and performance[2][4], the scaling properties of varying wordlengths is underexplored territory.

Modern general-purpose processors don't typically implement VLIW to speed up program execution, opting instead for complex out-of-order execution schemes that find ILP at runtime. VLIW promises to speed up program execution via multiple parallel instructions lined up by the compiler to execute concurrently, which can be leveraged to reduce program execution time with more execution units being used, or decrease area and power draw for identical performance. Should it be used, however, it's not known what instruction length is best for a given workload and design goals, which is what our project assists in discovering.

Our work focuses on the scaling properties of wordlength, providing insight to VLIW designers on tradeoffs between key metrics of program execution and design synthesis. In order to find the optimal number of instructions to run in parallel for various workflows, we compare area, speed and power efficiency as a function of word size on a custom-designed parametric VLIW core.

Tooling

Due to the complexity of VSISA, many tools are required to design, synthesize, test, and layout our design. While our design has not yet been described in this document, we find it helpful to give a tooling overview first. *Figure 1* depicts the relationship between these tools.

Icarus Verilog

. . . is a compiler for various designs of Verilog. In our case, Icarus offers a complete enough SystemVerilog compiler such that our design can easily run our functional testing suite, as well as run the assembled programs described later on.

OpenSTA

. . . is a static timing analysis tool that we were able to leverage for power profiling by running programs on a synthesized netlist. This tool was used to gather many of our power results. Synopsys Design Compiler . . . is a design compiler used in our project to synthesize SystemVerilog HDL code into a usable netlist.

Cadence Innovus

. . . is the PnR tool used in our project to perform the layout step for various synthesized designs. It was used for gathering area figures of different wordlength VLIW cores.

GPDK045 Technology

. . . is the Process Design Kit (PDK) used in our synthesis, layout, and power testing steps throughout our project. It was chosen due to its wide availability and strong documentation.

Python & Rust

. . . are the programming languages of choice for various additional toolings developed for our project. Namely, the VSISA assembler was implemented in Rust, and the Prime Factorization code was assembled using Python due to the algorithm's particularly strong ILP opportunity, not fully found by VSISA.

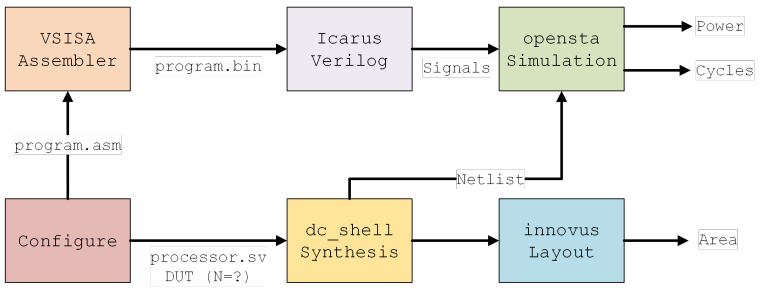

Figure 1: Tooling, Testing, and Analysis Pipeline

Our Design

We have implemented a minimal, scalable, non-pipelined, 8-bit, integer-only VLIW processing core, and used it to perform comparisons of power efficiency, area, and execution time (in cycles) for various workloads and design goals. We have written our processing core in SystemVerilog HDL code alongside functional simulation tests to assert the correctness of our implementation. We have designed an assembler that can convert VSISA assembly (our bespoke ISA) into executable code for any wordlength of synthesized processor.

With a parametric approach, leveraging meta-programming features of SystemVerilog, we were able to write a single top-level module with two parameters (wordlength and register count) that can generate a processor accepting a varying (at time of synthesis) number of instructions per word.

Hardware

Straightforward VLIW processors such as ours (at a high level), have a set number of instructions in an instruction word, and a set number of resources available to each instruction slot. This may be one-slot-per-resource (e.g. branch, ALU, memory assigned to slots 1,2,3 respectively), or one-resource-per-slot (all slots have most or all resources). Our approach is a one-resource-per-slot design for purposes of maintaining scope, with a modular wordlength. That is to say; rather than a 4-instruction wordlength being chosen from the start, our approach allows the wordlength to be synthesized for any wordlength greater than one instruction.

Figure 2 depicts the high level operation of our generated VLIW processing core. Each slot has one ALU, and operates under the assumption that branch logic will only ever occur in the first slot. As such, only the first slot has a branch unit.

The datapath starts, as usual, at the instruction memory. The Program Counter is depicted outside of the register bank for clarity. An N-instruction-length will contain up to N-instructions which are to be simultaneously executed by N slices. Since it is assumed that there are no data hazards in the program (enforced by compiler or assembler), every execution unit may write to the register bank at the same time with no collisions. Part of the no hazards invariant entails that only one branch instruction can be present per instruction word.

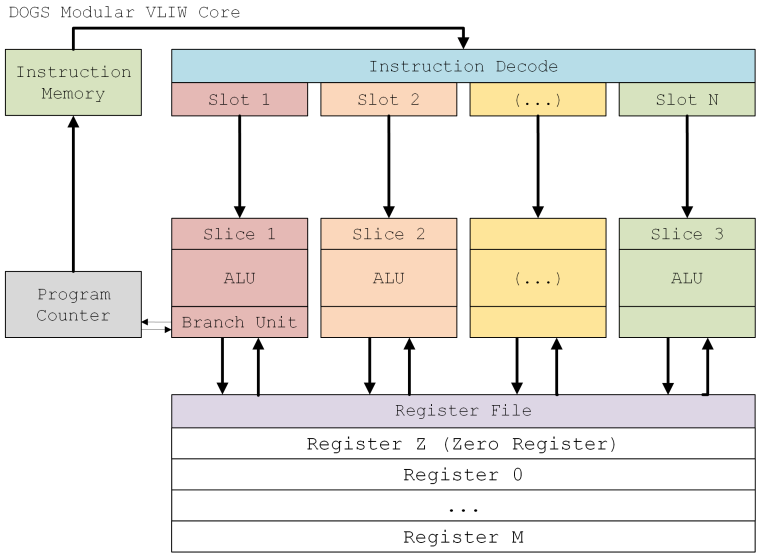

Figure 2: Block Diagram for our simple VLIW processor

The register bank is created using a single register file, as the downsides presented by a cheap method like this are not of relevance to well-implemented VLIW programs that do not contain any hazards. This allows every slice to have access to the same registers, and massively simplifies our design compared to cache-like approaches on other VLIW resource allocation techniques.

VSISA

Every processor needs an instruction set. Very Simple ISA (VSISA) is the name for our minimal and bespoke ISA implemented in our VLIW processing core. *Figure 3* contains a simplified overview of the VSISA features, namely the simplified 8-bit unsigned integer only execution space.

VSISA Feature Card		
8-bit Unsigned integers only	Registers, Literals	Procedural Programming
Operations		
Arithmetic	Bitwise	Control
ADD SUB MUL DIV	AND OR NOT	JMP JGZ LGZ
LADD LSUB LMUL LDIV	LAND LOR	Labels, Memory Addresses

Figure 3: Key Features of VSISA

The limited instruction set, while basic, provides a turing-complete interface (save for infinite memory) that our program testing can run on. VSISA offers some creature comforts as well, which are described in the assembler section.

Assembler

VSISA is implemented in our custom assembler which targets our VLIW processing core. The VSISA assembler is used to create test programs to be run both in SystemVerilog and on synthesized designs. The assembler supports varying wordlength instructions and primitive hazard checking (no reordering) to make the same input programs executable on different synthesized designs.

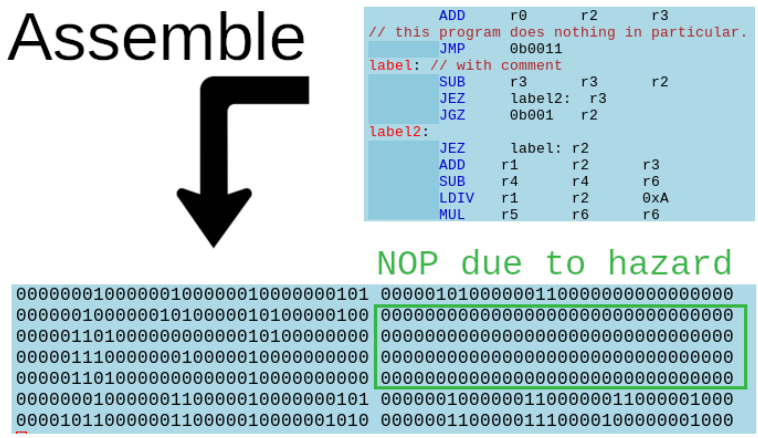

Figure 4: A sample VSISA program before and after assembly for N=2

Figure 4 depicts a real input program assembly source alongside its assembled output from our assembler for a wordlength of N=2. Note how the first two assembly lines ADD and JMP become one instruction word, since they do not have any hazards (unconditional jump). The next few instruction words have read or write hazards, between, and must be issued sequentially due to their codependence on R3. This means that the second execution unit has a NOP filled in such that behaviour can stay defined.

Testing

Testing was done through three different methods: Individual component testing, test program testing (via execution over test programs) and per-instruction testing.

Individual Component Testing

While creating individual verilog modules, we verified the correctness of our work by individually testing each verilog component with its own testbench, which we then would add to our overall testbench. In practice this meant supplying inputs to our modules sequentially and explicitly, since tooling up test generation would take too long. This allowed us to test the core functionality of each module before integration, and demonstrates functional correctness of each module.

An abbreviated testbench module for our branch unit is displayed in *Listing 1*. Wiring, CLK, tasks, and many test cases are omitted for clarity. Every individual component has a series of functional tests like in the above testbench, giving us confidence that our design is functionally sound despite not being formally verified.

Per-Instruction Testing

Once the full system was completed, the best method for a holistic evaluation would be to simply run a full test program, and assert that at each clock cycle the register and PC state are what we would expect them to be. To do this, three things were required: A test program, a means of compiling said test program into binary instructions, and a correct simulation that we can compare our work against.

For the test program, we used primality testing. We did so for two reasons, first because it doesn't require any memory accesses (since memory was out of scope for this project), and second because it is massively parallelizable and easily generalize to various instruction widths. This parallelizability comes from the fact that primality testing is effectively just running a big for loop to individually check every number smaller than the square root of the number we're checking for primality to determine if its a factor. Since each of these checks are independent of each other, they can be performed in parallel (by simply unrolling the loop into blocks of size n, where n is the number of ALUs we have). Once we have performed these checks (and in turn found the value modulo our target number) we need to determine if any of them are a factor. We would lose a significant amount of parallelization if we simply individually checked each register for zeros, so instead we, in `ceil(log(n))` steps, coalesce all values into one, effectively multiplying them all together, and if any had been zero then so too would be the final output.

```
module branch_unit_tb;
`include "incl/Branch_Ops.svh"
    branch_unit dut (
                      .CLK(CLK),
                      .Operation(Op),
                      .Address(Addr),
                      .PC(PC),
                      .Zero(Zero),
                      .Sub_UF(Sub_UF),
                      .PC_out(PC_out)
                  );

   always @(negedge CLK)
        if ($time > 1)
            assert (PC_out == expt) else dump();

    // test cases
    initial
    begin
       // setup omitted here
        #1
         begin
             Op <= BRANCH_ZERO_OP;
             Addr<=8'b0;
             Zero <= 'b0;
             Sub_UF <= 'b0;
             PC<=8'b01;
             expt <= 8'b10;
         end;
```

```
       // additional test omitted here
        #1
         begin
             $display("[PASS] Completed branch_unit Test at %0d",$time);
         end;
    end
endmodule // branch_unit_tb
```

Listing 1: A sample testbench

To compile the program rather than using VSISA which is better for compiling arbitrary programs, we used a python script. This is because we needed specific groupings for coalescing and needed to generate the assembly anyways. This script allowed us to generate this program for any arbitrary instruction width, rather than having to explicitly write it out for hundreds of instruction widths for testing.

Finally, we created a python simulation of our VLIW processor that we could run our program against, which allowed us to determine what the "correct" register state would be at each step would be, or at the very least assert equivalence between our verilog code and our python simulation. We then used these traces to generate an entire verilog testbench that would assert that at each step our DUT's state would be the same as the simulation's when provided with the previous clock cycle's state.

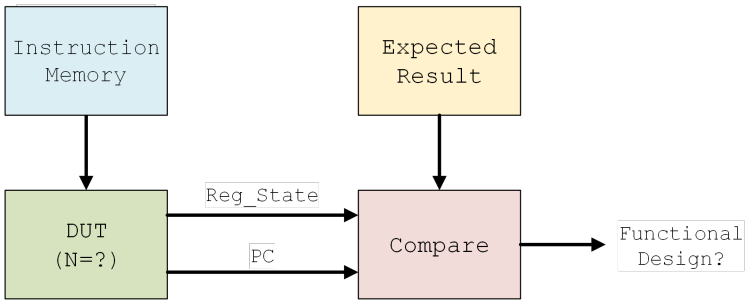

Figure 5: Per-Instruction testing DUT

The testing of DUT for this approach is shown in *Figure 5*.

Overall Program Testing

While per-instruction testing is effectively testing execution of the algorithm, we're not actually running the algorithm on our DUT. Rather, we're simulating individual steps at a time, and asserting that from the previous state, the expected next state is as we would expect it. We also wanted to ensure that our program would perform correctly when running a whole program at a time, so we encoded our programs into binary and used a simple loop to run our program, seen in *Listing 2*

```
while (PC != end_of_program) begin
        PC <= nextPC;
instruction <= program[PC];
        @(posedge clk);
end
```

Listing 2: Program Testing Snippet

Edge Case Coverage

Our edge cases are covered in 3 ways: First, we have nearly full coverage of our code through our individual component testing. Second, we ran long-running, complex programs and ensured that our processor was able to execute them correctly. Third, we have ensured that at each step of the program, the results are exactly as we would expect from our processor and there are no edge cases that for some reason don't have an impact on the program's correctness. Finally, we run these tests for various instruction widths, to ensure that our design works for arbitrary word lengths. Since we assert that both our program is executed successfully and that our program was not executed successfully by chance (since at each step we agree with the simulation), we are able to assert there are no edge cases missed when executing this program, and therefore edge cases are well tested.

Results

Figure 6 depicts the speedup of our optimal prime factorization algorithm as instruction width increases. The chart shows that cycle count is proportional to $\frac{1}{x}$ for a near-optimal algorithm, hinting that scientific and compute workloads benefit from the performance of VLIW.

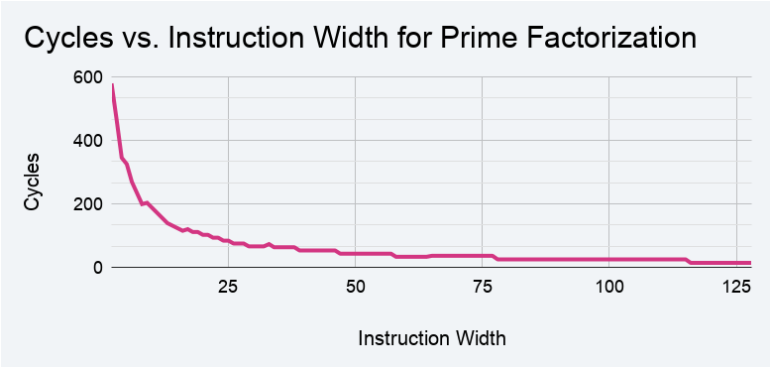

Figure 6: Cycle Count to Completion of Prime Factorization

Zooming in on *Figure 6* reveals that the function is almost piecewise, with clear 'steps' downwards in cycle count after a sufficient increase of N. This is attributed to the phenomenon of an algorithm needing to be 'fully solved' before moving onto a next step, where if that takes 75 instructions to complete with only 74 available instruction slots, it will always take 2 cycles (with 74 instructions dispatched in cycle 1, and only 1 instruction dispatched in cycle 2). This shortcoming only applies to extremely large values of N, but should be a key consideration for VLIW designers based on workload.

Figure 7 depicts how the real area scaling compares to a naive linear estimation of scaling. It is shown that the coefficient of scaling is slightly smaller than the area for `N=1`, due to shared resources like the register file not increasing with N.

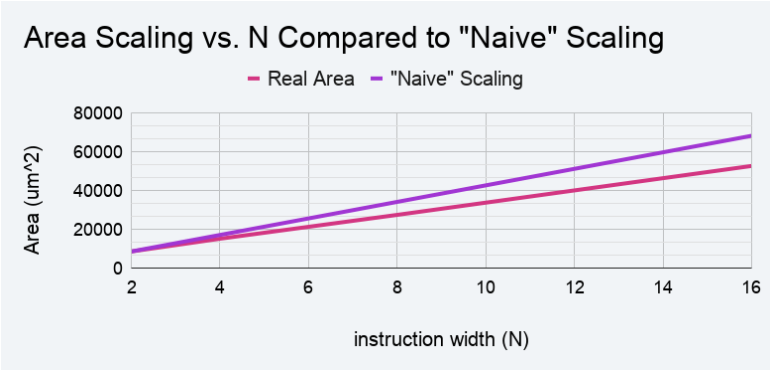

Figure 7: Area Scaling of our VLIW Core

This hints that VLIW may be a cost-effective performance scaling option under optimal conditions, which is illustrated in *Figure 8*. The "area effectiveness" is the execution speedup divided by the area taken, which remains close to 1. This means that VLIW scales very well and is suitable for high performance applications under ideal conditions.

Figure 9 shows the relative total power taken to compute primality testing for 233. This value is determined by multiplying the power characteristics found by OpenSTA (which are power with respect to time) and multiplying them by the cycle count, which is a proxy for time. While the resulting number is not exactly the total power consumed to compute the algorithm, the scale relative to other instruction widths would be the same.

This graph shows us that we don't see a significant increase in power for various instruction widths, which tells us that even as we increase the number of ALUs and the speed decreases the amount of power required to complete the algorithm stays relatively constant, meaning we can get algorithmic speedups with largely no consequence.

Finally, we have some illustrative PnR layouts for N=2, N=4, N=8, and N=16 instruction slots. *Figure 10* highlights the shared resources increasing slowly in size due to additional wiring, while execution units account for the majority of the area increase.

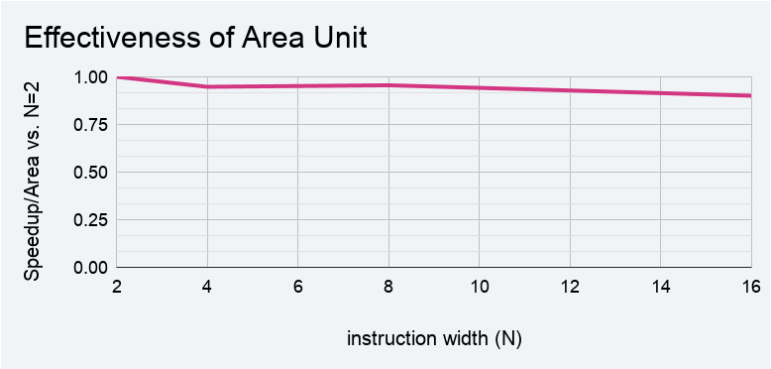

Figure 8: Area Efficiency under Prime Factorization

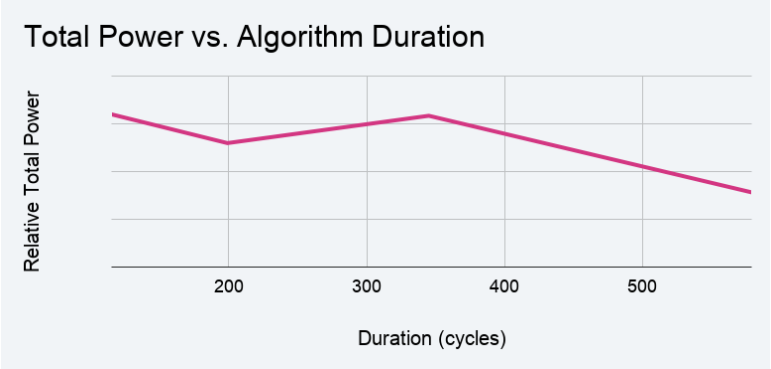

Figure 9: Power consumption compared to execution time

Note that the PnR images do not have the same scale, and are for illustration purposes only.

Future Work

The analysis performed in this work is limited by the scope of the DOG processor core. Simplification of our core reduced the applicability to real-world designs[1]. The simple hazard detection in the VSISA assembler limits discoverable ILP significantly. Beneficial future work would include Trace, a compiler technique enabling better ILP in programs[3]. Branch prediction, pipelining, and better resource allocation would further improve the accuracy of our analyses.

Conclusion/Work Summary

We have done the following:

- Created and tested VSISA, a minimal VLIW instruction set and assembler.
- Created and tested an arithmetic logic unit for our custom instruction set.
- Created and tested a branch unit for our custom instruction set that uses the first ALU for its branching logic.
- Created and tested an instruction decode module that splits the entire instruction word into indexable arrays to dispatch to the ALUs and BUs.
- Created and tested a register file for our specific architecture that allows reading from all registers by multiple ALUs, and allows writing to multiple registers from all ALUs.
- Created and tested a VLIW processor.
- Created a primality testing program generation script, which allows for the fast creation of a near-optimal primality testing algorithm for arbitrary instruction widths.
- Created a python simulation of our VLIW processor to allow for comparison between our verilog and a “correct” implementation of our processor.

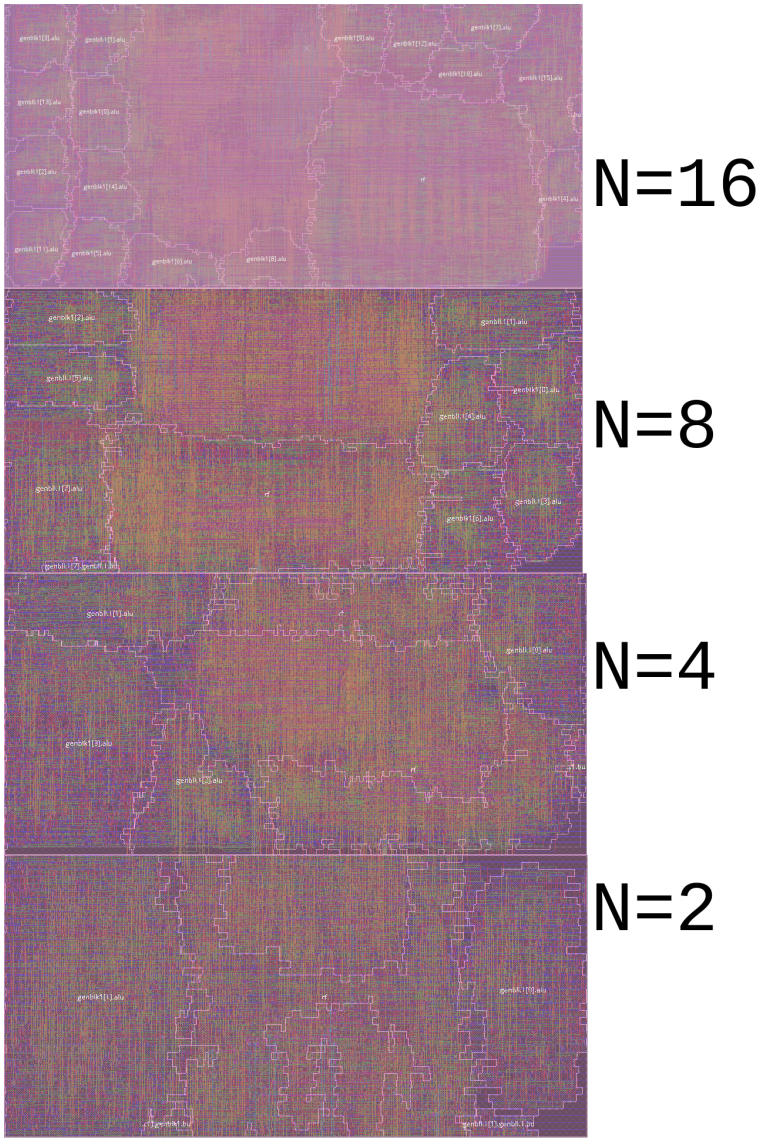

Figure 10: PnR of N=2,4,8,16 Instruction Slots

- Created a script to generate testbenches that would assert that each instruction step would match between our python simulation and our verilog implementation.
- Performed Synthesis and Place & Route for instruction widths of 2, 4, 8 and 16.
- Determined power consumption characteristics for instruction widths of 2, 4, 8 and 16.
- Determined total cycle count for primality testing for all instruction widths from 2 through 128.
- Determined relative power required to complete the primality testing algorithm for instruction widths of 2, 4, 8 and 16.

References

[1] de Souza, A. F., Fernandes, E. S., & Wolfe, A. (1997). On the balance of Vliw Architectures. Journal of Systems Architecture, 43(1–5), 15–22. https://doi.org/10.1016/s1383-7621(96)00115-4

[2] Kobayashi, Y., Kobayashi, S., Okuda, K., Sakanushi, K., Takeuchi, Y., & Imai, M. (n.d.). Synthesizable HDL generation method for configurable VLIW processors. ASP-DAC 2004: Asia and South Pacific Design Automation Conference 2004 (IEEE Cat. No.04EX753), 843–846. https://doi.org/10.1109/aspdac.2004.1337712

[3] Odumabo, A., Adedokun, A., Adekotujo, A., & Ogunbodede, O. (2020). Very long instruction word: A higher performance processor. International Journal of Computer Applications, 175(15), 11–15. https://doi.org/10.5120/ijca2020920645

[4] Lenell, J., & Bagherzadeh, N. (n.d.). A performance comparison of several superscalar processor models with a VLIW processor. [1993] Proceedings Seventh International Parallel Processing Symposium, 44–48. https://doi.org/10.1109/ipps.1993.262853

[5] Fisher, J. A. (1983). Very long instruction word architectures and the eli-512. Proceedings of the 10th Annual International Symposium on Computer Architecture - ISCA '83, 140–150. https://doi.org/10.1145/800046.801649

[6] Agarwala, S., Anderson, T., Hill, A., Ales, M. D., Damodaran, R., Wiley, P., Mullinnix, S., Leach, J., Lell, A., Gill, M., Rajagopal, A., Chachad, A., Agarwala, M., Apostol, J., Krishnan, M., Duc Bui, Quang An, Nagaraj, N. S., Wolf, T., & Elappuparackal, T. T. (2002). A 600-MHz VLIW DSP. IEEE Journal of Solid-State Circuits, 37(11), 1532–1544. https://doi.org/10.1109/jssc.2002.803954