# Open source FPGA-based emulation with Nexus

Peter Birch - github.com/intuity/nexus - peter@intuity.io

## 1 Introduction

Simulation remains a powerful and intuitive way to verify a design, but its utility is often limited by how fast it can execute. Verification on FPGA can be a much faster option, but it comes at the cost of decreased design visibility, long re-spin times and nondeterministic test results. Chip-scale emulation platforms such as Cadence's Palladium [1] or Synopsys' ZeBu [2] are incredibly capable, but not cost-effective for early stage research and development or the verification of small designs. UC Berkeley's FireSim [3] is an open source option for creating repeatable simulation results on FPGA, but it is still subject to a full synthesis and place-and-route flow for every design change.

Nexus aims to be a hardware-accelerated emulation platform for small designs at simulated clock rates approaching 1 MHz. It is formed of a programmable systolic array [4] with a custom compiler that converts a design into compatible instructions. With a fixed FPGA bitstream the simulated design can be updated without time-consuming place-and-route. This approach also allows trace points to be added or removed during a simulation by updating the program executing on the array.

This paper will discuss both parts of the platform and how they enable simulation. It will detail how the hardware has developed to increase the capacity and capability to a useful point, and what the next steps are.

## 2 Parallel Simulation

Simulation of sequential logic can be an embarassingly parallel problem. Each flip-flop and the cone of logic that feeds it can be considered as an independent operation evaluated once per cycle. Silicon's physical properties limit the complexity of a logic cone, providing an upper bound on the size of an operation. The example shown in Figure 1 shows how two such operations can be extracted from a circuit.

In more realistic scenarios logic cones are unlikely to be independent with common terms existing between operations, as with Figure 2. Such terms can either be evaluated once with the result shared between all dependent operations, or can be separately evaluated as part of each logic cone. The former approach limits the achievable parallelism as operations must wait until the common term is evaluated (limiting simulation speed), while the latter rapidly increases the required effort to compute the next state (limiting system capacity).

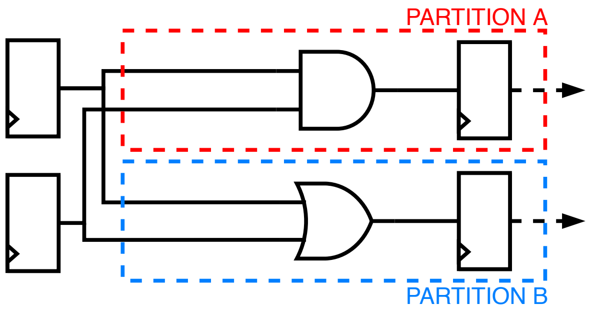


Figure 1: Independent Operations

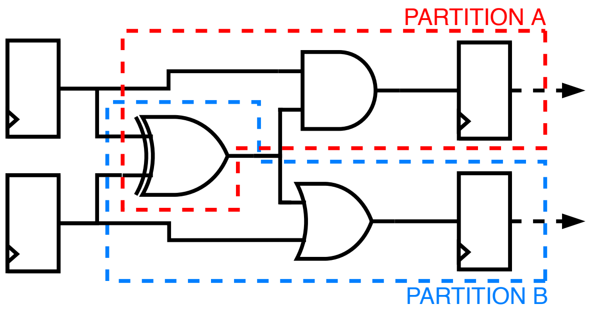


Figure 2: Overlapping Operations

As a partition's outputs change, input values to other partitions must be updated. Ideally only sequential state should be shared as it changes just once per cycle. Combinational signals can change multiple times, and if shared between partitions this requires either intensive recomputation of the receiver's entire state, or resource intensive suppression of steady states. Shared combinational state also means a dependency, which can increase the critical path length and reduce simulation speed.

Fortunately terms are generally re-used a small number of times and state is pipelined, limiting the number of partitions that share state. This would suggest that 'grouping' operations may be beneficial as it would allow state and common terms to be shared. However, the group size must be balanced against the degree of parallelism and so some terms may need to be replicated.

## 3 Nexus Hardware

Nexus is formed of a two dimensional mesh of nodes (Figure 3). This structure can be considered as a systolic array, as each node evaluates a fraction of the design. A distributed trigger signal begins evaluation of the next state, with the next trigger only issued when all nodes return to idle. The trigger is pipelined to ease timing, while the idle signal is aggregated per column.

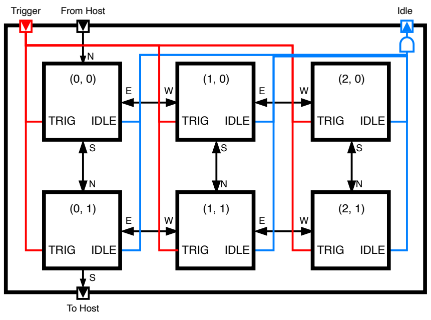


Figure 3: Block diagram of the mesh

Each node executes a series of instructions from RAM to transform input state to output state. In FPGAs like Xilinx's Artix-7 series [5] block RAMs are relatively large (4 kB) and abundant (365 in a XC7A200T), so can hold a considerable number of operations.

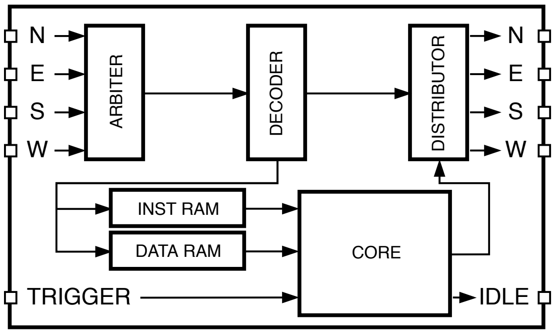


Figure 4: Block diagram of a node

Nodes can exchange messages with their neighbours to the north (N), east (E), south (S), and west (W). Each transfer carries the ID of the target node, the command, and a payload. If a node receives a message targeting a different node it will forward it, allowing for any-to-any communication. To keep the mesh small this same network is used to carry state updates, load program data, and communicate with the host.

A node can digest or route one message per cycle. Round-robin arbitration is used to fairly serve each ingress port. If multiple ports present data at the same time those not serviced are backpressured, which can limit the simulation speed. Compiler optimisations can be used to reduce the distance state updates travel through the mesh.

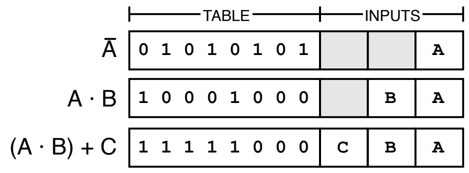
### 3.1 Proof-of-Concept

Nodes initially had just eight inputs, eight outputs, and eight working registers (each one bit wide). Instructions could perform a fixed number of one or two input operations (NOT, AND, OR, etc.) taking values from the inputs or registers, and writing the result to the registers or outputs. A lookup table held in flops encoded up to two messages per output, which could be directed to a single recipient or broadcast across the mesh. A tiny mesh of only thirty-six nodes (288 simulated flops) could fit in an Xilinx XC7A200T, making this approach unviable.

The proof-of-concept yielded two main results. Firstly, broadcasting state updates consumed significant message bandwidth and required dedicated hardware in both the sender and receiver to update an input. Secondly, storing output message configurations in flops rather than block RAM caused excessive resource usage.

### 3.2 Increasing Capacity

The revised hardware had thirty-two inputs and outputs and sixteen working registers per node (again one-bit wide). Instructions were encoded as an eight bit truth table (Figure 5) supporting up to three input operations, allowing multiple gates to be evaluated in a single cycle.



| | TABLE | INPUTS | | |
|---|---|---|---|---|
| $\bar{A}$ | 0 1 0 1 0 1 0 1 | ▨ | ▨ | A |
| $A \cdot B$ | 1 0 0 0 1 0 0 0 | ▨ | B | A |
| $(A \cdot B) + C$ | 1 1 1 1 1 0 0 0 | C | B | A |

Figure 5: Truth Tables

Output messages were stored in RAM as a lookup table, which could specify any number of messages to send. This eliminated broadcast messages as state changes could be sent to any number of nodes. Receive-side hardware was simplified as the message identified the input to update. This significantly reduced resource usage, allowing one hundred nodes (up from thirty six) to fit in the same FPGA with a capacity of 3,200 simulated flops.

However sequential state was still stored by a node's inputs, with double-entry flops updating to the new value on the trigger pulse. With inputs both holding state and providing paths for nodes to share data, increasing design complexity led to extreme contention and only very simple designs could be placed. Figure 6 illustrates how a relatively simple design could exhaust an eight input node by using output terms as part of the operation, and this same issue extends to a thirty-two input node.

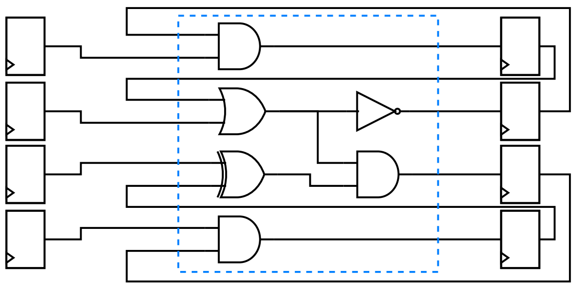


Figure 6: Exhaustion of Node Inputs

## 3.3 A Softer Approach

The latest hardware revision takes a different approach, replacing the hardened input and output handling with an expanded instruction set. There are seven eight-bit wide general purpose registers, with an eighth register accumulating results from truth table operations. `STORE` and `LOAD` instructions allow data to be transferred to and from a separate data memory, while `SEND` allows the contents of a register to be sent to another node. State updates arriving at a node are written directly into the data memory, with the eight-bit 'slot' alternating on consecutive cycles to support sequential logic. These changes drastically increase the capacity with a theoretical maximum of 16,384 flops per node, and can support any sensible number of inputs and outputs.

In certain cases multiple pipeline stages can now be contained within a node, which notably reduces messages between nodes. Additionally, as significantly more state can be held within a node the operation complexity it can support also increases. This means that for most cases only sequential state needs to pass between nodes, requiring fewer messages to be sent and allowing the mesh to reach a quiescent state without iteration.

The downside to this latest structure is that it results in longer instruction loops per node, meaning slower simulated clock rates. However, the capacity benefits massively outweigh the loss in speed and the compiler can split the design into more partitions to minimise the work each node has to perform.

# 4 NXCompile

As Nexus' hardware is fairly simple, it relies entirely on the execution of instructions to achieve simulation. Mapping arbitrary logic onto the truth table instruction can be thought of as a form of synthesis. However, the tool must also handle register allocation and memory operations, which are standard operations for a compiler.

## 4.1 Ingestion & Optimisation

Yosys [6] is used to translate the design into a generic technology mapped netlist, and exported to Verilog as a series of simple assignments and clocked processes. Slang [7] is then used to parse this netlist into a traversable data structure.

The design is optimised to directly connect gates and flops (eliminating intermediate `wire` assignments), to propagate constant terms through gates (such as replacing $A \cdot 1$ with just $A$), and to eliminate any logic from the design not contributing to an output signal. For an unoptimised netlist, these refinements have a significant impact on the quality of the result achieved by the compiler and make the design easier to handle.

## 4.2 Partitioning

Partitioning happens in two stages. In the first stage, each flop in the design and the cone of logic that feeds it are placed into a 'group'. A gate may appear in any number of groups, while a flop appears in exactly one. The number of gates determines each group's complexity.

In the second stage, groups are distributed across partitions according to capacity constraints based on the capabilities of a hardware node. The Kernighan-Lin mincut algorithm [8] is used to bisect each partition with a low number of cross-connections. The net effect is to locate related groups of logic in a single partition while minimising the amount of shared state.

The first stage can introduce repeated computation of results that are shared between different logic cones. However, this cost is largely offset by placing related logic into one partition. The combination of the two stages attempts to achieve the balance described in section 2, where repeated computation is undesirable except in situations where it aids parallelism.

## 4.3 Instruction Stream Generation

After partitioning, instructions are generated for each node to compute and communicate state. The truth table instruction takes up to three input terms, compressing multiple gates into a single cycle. For example, if $A$, $B$ and $C$ are inputs and $D$ is $B \oplus C$ then $A \cdot D$ can be expanded to $A \cdot (B \oplus C)$. This can be converted to a truth table `0 0 0 0 0 1 1 0` and the result can be determined by taking the bit at index $\{A, B, C\}$.

One approach would be to walk backwards through a flop's logic cone aggregating three input terms. In practice this tends to produce a high number of three input truth tables, but also leads to significant recomputation of common terms. When trialed on PicoRV32 [9], this resulted in a disappointing worse-case truth table to gates ratio of 1.27:1, while a direct translation of gates to instructions would be 1:1.

A better approach would be to sort all gates in the partition by descending size of their fan-out. Terms can then be accumulated in a similar fashion, but stopping whenever a previously constructed term is encountered. This approach produces fewer three-input tables, and significantly fewer instructions overall as more common terms are shared. For PicoRV32, this gave a significantly improved worse-case truth table to gates ratio of 0.79:1, outperforming a direct translation.

The generated truth tables must be ordered to ensure common results are available to dependent instructions. As the hardware does not support conditional branching, instructions will always execute in the same order and this has some significant benefits.

One benefit is that register selection can easily reduce spill cost (where register content is written to memory [10]) by prioritising evictions of registers with no further references or, if no such register exists, evicting the register whose next reference is furthest away in the truth table order. Only registers with novel state need to be spilled during an eviction, others can be safely dropped.

Another benefit is that, through careful packing, related state can be located in the same eight bit slot in RAM. This significantly reduces the number of load instructions required to execute the stream of truth tables compared to random packing.

For PicoRV32, a complete partition state update yields a worse-case instruction to gate ratio of 1.64:1 with an average of 1.41:1. Future work will investigate how the number of memory operations can be further reduced.

During communication, the sender will need to reshape data to match the receiver's memory packing. This process requires significant data movement, which the `SHUFFLE` and `STORE` instructions attempt to simplify. The `SHUFFLE` instruction can rearrange the eight bits of a register into any order in one cycle, while the `STORE` instruction supports masking to only write selected bits. Together they allow messages to be progressively accumulated and transmitted once all results are computed.

Communication generation is still immature, so results are not yet available. A reasonable assumption is that the instruction to gate ratio will worsen, but the impact can be minimised by interleaving the accumulation of shared state with compute.

For PicoRV32, the predicted clock frequency is around 550 kHz (assuming an FPGA fabric clock of 250 MHz). By comparison, single-threaded simulation on an Intel i5-1038NG7 without wave tracing will achieve 36 kHz under Icarus Verilog and 240 MHz under Verilator. Verilator's impressive performance comes from years of optimisation and utilises the CPU's much wider boolean and arithmetic operations compared to Nexus' bit-wise approach. Similar optimisations could be made to Nexus to substantially improve its performance.

## 5 Conclusion

This paper introduced Nexus' hardware and compiler. While immature, the latest architecture model promises the capacity to run small, but non-trivial, designs such as PicoRV32. The short term focus will be on completing the compiler to support simulation. In the longer term Nexus could support dynamic interaction with a host system, allowing software-based stimulus of the emulated design and full-depth wave tracing.

## References

[1] Cadence Design Systems, "Palladium Emulation Platform." www.cadence.com/en_US/home/tools/system-design-and-verification/emulation-and-prototyping/palladium.html

[2] Synopsys, "ZeBu Server ASIC Emulator." www.synopsys.com/verification/emulation/zebu-server.html

[3] Berkeley Architecture Research, "FireSim." www.fires.im

[4] R. P. Brent and H. T. Kung, "Systolic VLSI arrays for polynomial GCD computation," *IEEE Transactions on Computers*, vol. C–33, no. 8, pp. 731–736, 1984, doi: 10.1109/TC.1984.5009358.

[5] Advanced Micro Devices, "Artix-7 FPGA Family." www.xilinx.com/products/silicon-devices/fpga/artix-7.html

[6] YosysHQ GmbH, "Yosys." yosyshq.net

[7] M. Popoloski, "SystemVerilog Language Services." github.com/MikePopoloski/slang

[8] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970, doi: 10.1002/j.1538-7305.1970.tb01770.x.

[9] YosysHQ GmbH, "PicoRV32 - A Size-Optimised RISC-V CPU." github.com/YosysHQ/picorv32

[10] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer Languages*, vol. 6, no. 1, pp. 47–57, 1981, doi: 10.1016/0096-0551(81)90048-5.