Berkeley Emulation Engine

Motivation

It is impressive to see how much tooling has improved for modern programming languages. A key driver of this progress is incremental compilation, which has proven to be a game changer by tightening the edit–compile–debug loop to the extreme and empowering developers to prototype systems rapidly. In contrast, tooling for hardware engineers has largely stagnated. Frontend HDLs such as Chisel [3] have improved designer productivity by adding a strong metaprogramming layer on top of the RTL abstraction, and Verilator [4] has emerged as an open-source RTL simulator with performance comparable to commercial offerings. However, these advances do not fundamentally change the underlying flow. Verilog elaboration still takes a long time and provides limited support for incremental compilation. Likewise, RTL simulators either do not support incremental compilation at all, or hide it behind opaque command-line options. Finally, despite a substantial body of research on accelerating simulation [1], [2], [3], the raw speed of RTL simulation has not improved in any meaningful way.

The vision of this project is to narrow the iteration-time gap between software and hardware development as much as possible. Our first goal was to build an RTL emulation machine capable of running test vectors at high throughput while still producing full waveforms, eliminating the need for developers to wait on long simulation runs. In addition, we aimed to dramatically reduce compilation time compared to existing emulation systems by introducing incremental compilation support. Together, these capabilities enable a workflow where engineers can iterate on RTL designs with the same rapid pace as software developers.

A key challenge in reducing compilation time is avoiding the placement and routing (PnR) phase in FPGA-based flows. This requirement motivated us to adopt a fundamentally different architecture for our emulation machine: a processor-based design. By eliminating timing closure and PnR, we achieve significantly faster compile times. High performance is obtained by exploiting massive parallelism, with an abundance of processors extracting fine-grained parallelism within RTL simulation. This architecture also makes state extraction straightforward, improving waveform generation and debug capabilities.

The remainder of this chapter is organized as follows. We begin with background on RTL simulation. We then describe the architecture of our processor-based emulation machine and its supporting compiler stack. Finally, we present initial results, highlight key takeaways from the project, and describe future work.

Background on RTL Simulation

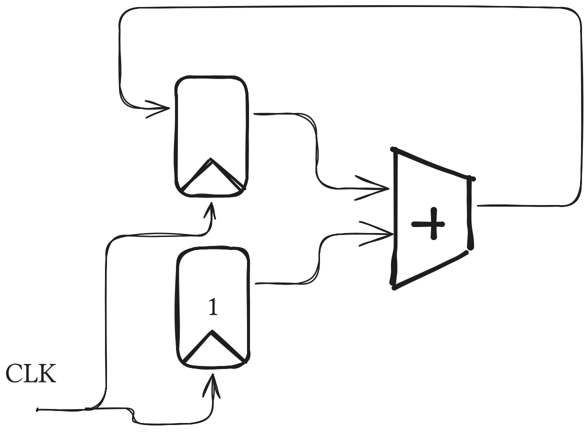

Figure 1: Example RTL circuit representation.

RTL stands for *Register Transfer Level*. At the RTL abstraction, a circuit consists of three primary components:

1. Registers - these store the state of the circuit across cycles.
2. Combinational logic - this computes outputs from inputs within a cycle.
3. Wires - these carry signals (bits) between components.

In Figure 1, two registers feed an adder (combinational logic), and the adder's output drives another register input. The adder result is latched into the register on the next rising or falling clock edge.

For clarity, we now restrict our scope to single-clock, synchronous circuits. As a result, explicit clock signals will be omitted in subsequent diagrams. We further narrow the discussion to circuits with a *single level of sequential logic*: registers drive combinational logic, which in turn drives either outputs or other registers. Such circuits naturally form a directed acyclic graph (DAG), since combinational loops are disallowed.

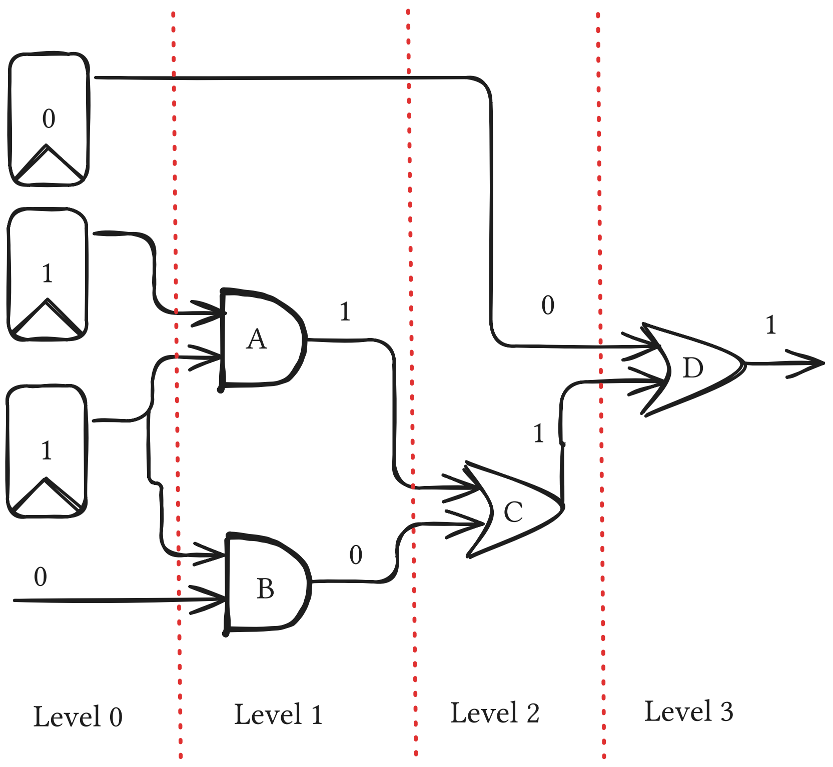

Figure 2: Simulating a single level of sequential logic by **levelization**.

Figure 2 shows an circuit that we will use as an example. This circuit can be represented as a graph, where registers and combinational logic elements correspond to nodes, and the connecting wires correspond to edges. The first step in simulating a single level of sequential logic is to **levelize** the circuit (i.e., perform a topological sort). We begin by assigning **level 0** to all registers and external inputs entering the graph. For all other nodes, their level is determined according to the following rule:

`node.level = max(child[0].level, child[1].level, ...) + 1`

For example:

- `A.level = max(0, 0) + 1`
- `C.level = max(A.level, B.level) + 1 = max(1, 1) + 1`
- `D.level = max(Reg.level, C.level) + 1 = max(0, 2) + 1`

The next step is to propagate the signals level by level:

- Level 1:
  - Output of the `AND` gate `A` is `1`
  - Output of the `AND` gate `B` is `0`
- Level 2:
  - Output of the `OR` gate `C` is `1`
- Level 3:
  - Output of the `OR` gate `D` is `1`

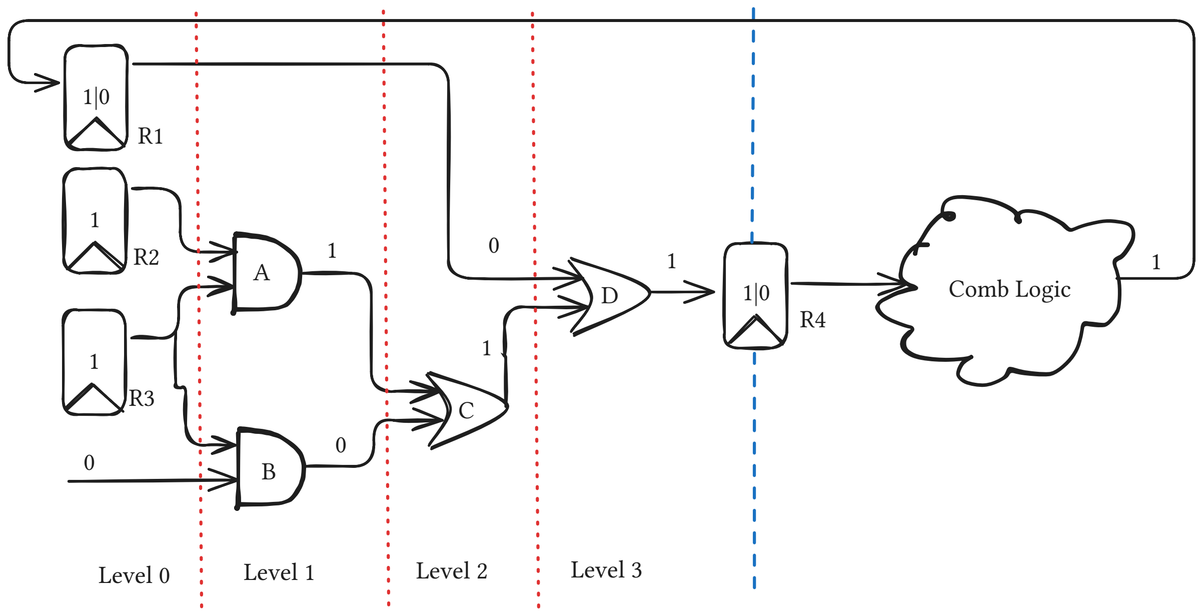

Figure 3: RTL simulation with multiple levels of sequential logic.

We can now generalize this approach to circuits with multiple register stages. Such circuits may contain cycles, but every cycle must traverse at least one register.

Simulation proceeds in two distinct phases:

1. Combinational evaluation.

Each region of sequential logic is simulated independently. For example, the logic between `R1`, `R2`, and `R3` feeding into, `D`, as well as the logic driven by `R4` into the output of `Comb Logic` can be evaluated in parallel.

2. Register update.

Once all combinational outputs have been computed, the register values are updated simultaneously to their new inputs. For instance, `R1` is updated from `0` to `1`, and `R4` is updated from `0` to `1`.

This two-phase scheme ensures that all register changes appear to occur in lockstep on the clock edge, faithfully modeling synchronous behavior.

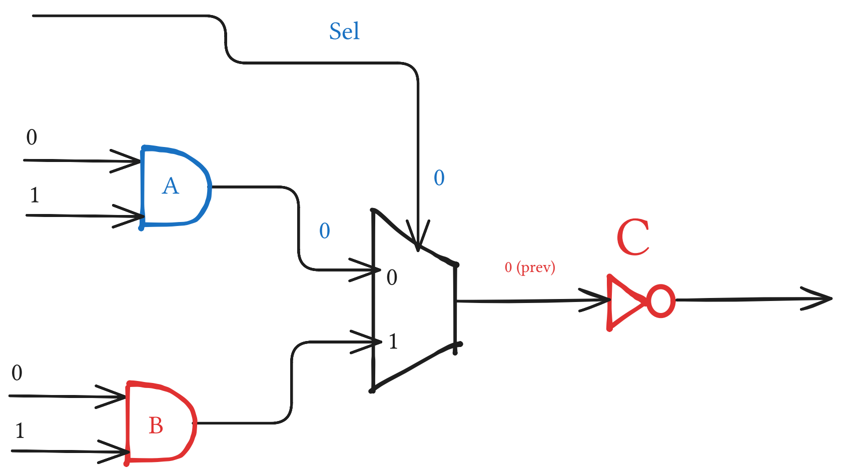

Figure 4: Event driven RTL simulation.

Up to this point, the simulation schedule has been determined entirely at runtime by traversing the entire circuit graph. In an event-driven RTL simulator, an event is generated whenever a signal's value changes from the previous cycle. These events are placed in a priority queue and scheduled dynamically, allowing the simulator to skip evaluating portions of the circuit that remain unchanged. This reduces computation compared to the static approach, where every node is traversed each cycle regardless of activity.

In Figure 4, the first event corresponds to the `sel` signal. It evaluates to `0`, which means the output of `B` can be skipped (marked red). Next, the output of `A` is evaluated to `0` and propagated to the input of `C`. If `C`'s input was already `0` in the previous cycle, then `C` and all downstream logic can be skipped as well.

Event-driven simulation is most beneficial when large portions of the circuit remain idle. However, it introduces runtime overhead: the simulator must dynamically manage the scheduling queue and maintain state to track activity propagation. Commercial tools such as VCS [4] and Xcelium [5] employ this approach.

By contrast, statically scheduled simulation avoids this runtime overhead by determining the evaluation order at compile time. The circuit is flattened into straight-line code that executes the same sequence every cycle. While this eliminates dynamic scheduling, it comes at the cost of poor CPU instruction cache locality and frequent branch mispredictions due to the large graph traversal. Verilator [6] follows this statically scheduled approach (though it provides limited support for cycle skipping).

Overview of Processor Based Emulation

The speed limitation of software RTL simulation have motivated the search for faster execution platforms. IBM's Yorktown Simulation Engine [7] in the early 1980s demonstrated that an array of programmable logic processors could efficiently evaluate millions of gates in parallel. Later Quickturn developed their own processor based emulation machines which in turn evolved into Palladiums [8].

At the core of processor-based emulation is the mapping of combinational logic into lookup tables (LUTs). Each LUT encodes the truth table of a small logic function (e.g., a three-input gate), and a processor evaluates one LUT per host-emulator-cycle. Unlike FPGAs, where LUTs are wired together to execute in a single cycle, processor-based emulators time-multiplex many LUTs onto a grid of emulation processors. Each processor executes a microprogram of LUT evaluations, fetching operands from its local memory.

Since the combinational logic graph of an RTL design may contain millions of gates, these LUT operations must be distributed across thousands of processors. A low-latency on-chip network connects the processors, allowing the output of one LUT to become the input of another within a few cycles. This inter-processor communication fabric enables the emulator to exploit the fine-grained parallelism inherent in RTL simulation. In addition to emulating LUTs, processor-based emulators dedicate specialized SRAM processors that directly emulate memory arrays. Flattening SRAM arrays into pure LUT form would consume significant processor resources reducing the effective capacity of these machines.

It is instructive to perform comparisons with FPGA based emulation systems. FPGA-based emulation systems must execute a chain of LUTs within a single FPGA cycle. Hence, the entire design must be mapped, placed, and routed so that every combinational path meets the timing constraints. This placement-and-routing (PnR) step is time-consuming and can cause days of compilation time for the highest capacity FPGAs. Worse, compilation frequently fails when failing to meet timing constraints. However, FPGAs have a higher unit capacity and speed compared to processor based emulation due to their simpler hardware and single-cycle evaluation of entire combinational blocks.

Processor-based emulators, in contrast, remove the need for long and fragile PnR runs. Since LUTs are time-multiplexed onto processors, there is no need to worry about meeting timing constraints. Compilation becomes a matter of partitioning and scheduling rather than PnR. The tradeoff is lower performance and less density per unit area as many emulator-cycles are required to evaluate what an FPGA LUT fabric could compute in one cycle. Nevertheless, this approach offers fast, reliable compilation, and scalable capacity, making it especially effective for large RTL verification workloads where iteration time is critical.

Emulation Processor Microarchitecture

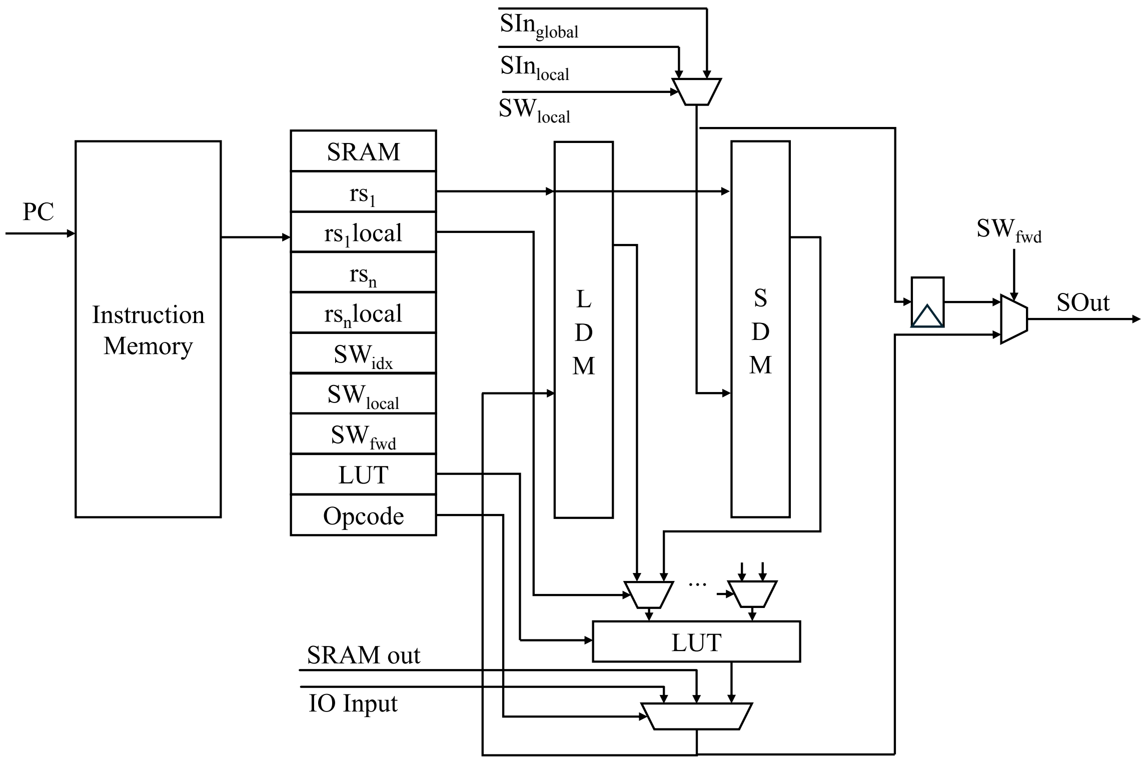

Figure 5: Microarchitecture of a logic processor.

Figure 5 depicts the microarchitecture of a single emulation processor. Instructions are fetched from instruction memory using the program counter (PC) and are decoded into fields indicating an SRAM operation (`SRAM`), operand registers (`rs`$_1$, `rs`$_n$), data memory selectors (`rs`$_1$`local`, `rs`$_n$`local`), switch controls (`SW`$_{idx}$, `SW`$_{local}$, `SW`$_{fwd}$), `n`-input `1`-output lookup tables (`LUT`), and `opcode`. The design incorporates local and switch data memories (`LDM` and `SDM`) to store the outputs computed from the current processor or neighboring processors respectively. By using `rs`$_x$`local`, the read outputs from `LDM` and `SDM` is multiplexed and used as inputs to the `LUT`. The output of the `LUT` is written back to `LDM` and sent out to the network so that other processors can use it to perform computations in subsequent cycles.

A single *target cycle* of the emulated design is realized by stepping through all the instructions allocated to this processor. Each instruction corresponds to evaluating a portion of the combinational logic graph, and once the full instruction sequence has been executed, the emulator has effectively advanced the simulated design by one clock edge. Since the hardware has no dynamic interlocks, the instruction stream must explicitly guarantee that operands are available before use. To enforce this, the compiler inserts *NOPs* into the instruction schedule whenever there are data hazards, ensuring correctness at the cost of additional host-cycles. Thus, the compiler is fully responsible for partitioning the logic, allocating instructions, and scheduling them with appropriate stalls so that the emulation proceeds deterministically across the distributed processor array.

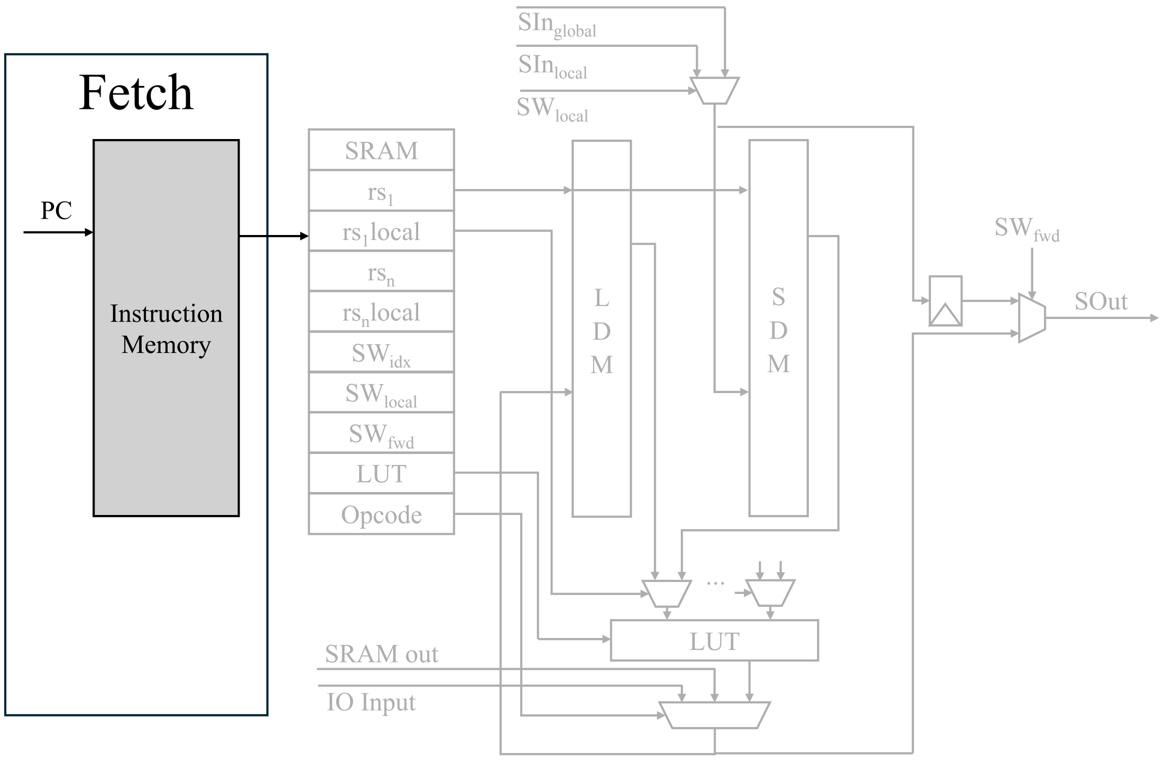

Figure 6: Fetch stage of logic processor.

Figure 6 shows the fetch stage of the pipeline. The PC maintains the current instruction address and increments each host-cycle, while the instruction memory retrieves the corresponding encoded instruction. Since all processors in the array advance their PCs in lock-step, this stage establishes the global sequencing of the emulated cycle. Unlike a conventional processor that may support speculative control flow, the PC in the emulation processor is strictly linear, stepping deterministically through the allocated instruction stream until the entire instruction block representing a target clock cycle has been executed.

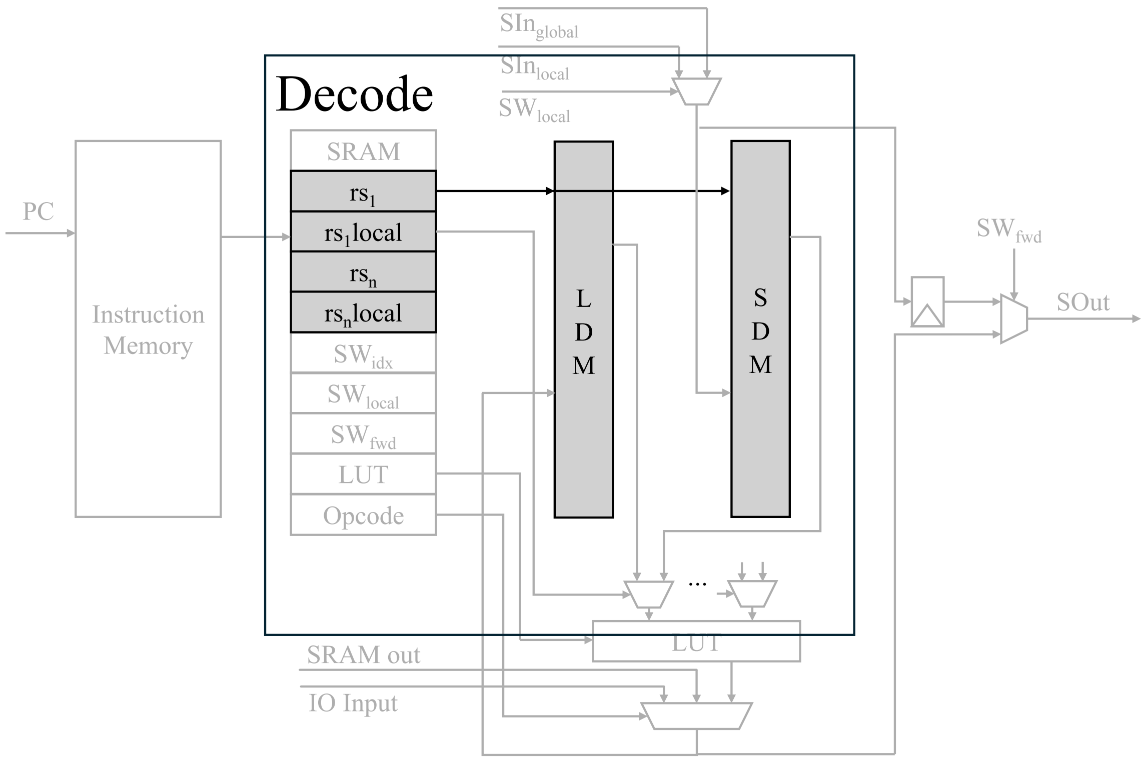

Figure 7: Decode stage of logic processor.

Figure 7 shows the decode stage. Once an instruction has been fetched, the decode logic parses its fields into operand specifiers (`rs`$_1$, `rs`$_n$), memory selectors (`rs`$_1$`local`, `rs`$_n$`local`), and control flags for special operations such as SRAM access or switch routing (`SW`$_{idx}$, `SW`$_{local}$, `SW`$_{fwd}$). These decoded signals drive address ports of the local and switch data memories (`LDM`, `SDM`), configuring them to deliver the requested operands in the following cycle. In summary, the decode stage orchestrates the preparation of execute stage. Because the hardware has no hazard detection, the compiler must insert NOPs when decoded operands are not yet available.

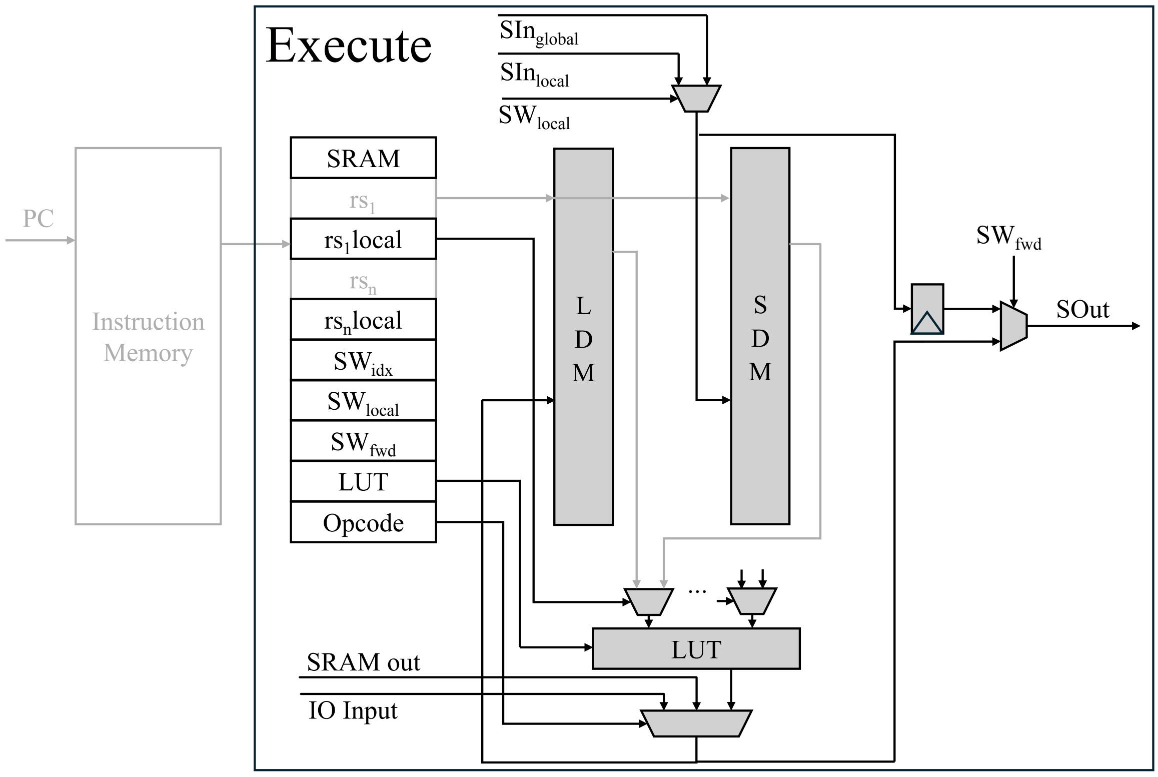

Figure 8: Execute stage of logic processor.

Figure 8 depicts the execute stage of the emulation processor pipeline. Operand data, selected via the `LDM` and `SDM` paths, flows into the `LUT` block, which computes the instruction's boolean function. Multiplexers configure whether the source data originates from `SRAM`, `I/O`, or `LUT` outputs, while the switch fabric manages operand routing between global and local networks. The results are then optionally stored locally in the `LDM`, or propagated through the inter-processor interconnect (`SOut`),

SRAM Processor Interface

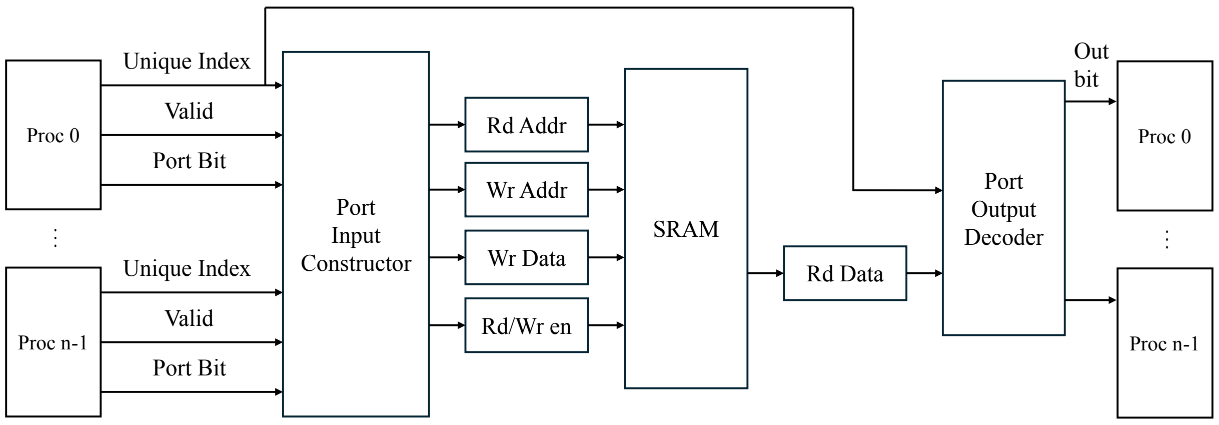

Figure 9: Interface to SRAM processors.

Figure 9 shows the SRAM processor interface instantiated once per emulation module. Dedicated SRAM processors are required because flattening SRAMs into LUTs and registers inside the logic processors would result in an explosion of instruction count, quickly exhausting instruction memory capacity. By offloading memory operations to a specialized SRAM processor, the design preserves both efficiency and scalability.

The SRAM processor is also a good example of how Chisel's metaprogramming layer simplifies design of complex datapaths as shown in Listing 1. Each logic processor communicates with the SRAM processor by sending a unique index and a port bit corresponding to a SRAM interface such as a read address, write address, write data, or enable signal. Scala's higher-order functions (`Seq.fill`, `map`, `zipWithIndex`) make it possible to generate this wide, parameterized multiplexing structure concisely, avoiding the repetitive generate blocks and manual wiring that would be required in Verilog. For instance, shifted offsets are computed for every processor and conditionally gated with `valid` and `io.run` in just a few lines, while a reduction tree (`reduceTree`) neatly aggregates per-processor enable signals into a global read-enable.

During execution, processors assert a `valid` flag when the current instruction specifies an SRAM operation. The port input constructor updates the registers that collectively represent the SRAM inputs for the target cycle. Since multiple processors may attempt to drive different fields concurrently, the constructor implements a large multiplexer to select the correct source for each port bit. This wide multiplexing structure is the critical path within the emulation platform. The SRAM itself operates with a one-cycle read latency so the read data observed in the current cycle reflects the request of the previous cycle. This data is distributed back through the port output decoder, which routes bits to the appropriate processors using their indices. Each processor then decides, based on its instruction's `SRAM` field, whether to write back the returned data into its local data memory (`LDM`).

In summary, the SRAM processor enables efficient emulation of large memory structures by time-multiplexing the SRAM interface across processors. At the same time, Chisel's metaprogramming layer makes the construction of the multiplexing and control logic concise and beautiful.

```scala
val decs = Seq.fill(num_procs)(Module(new SRAMIndexDecoder(cfg)))
  for (i <- 0 until num_procs) {
    decs(i).io.idx := pl_idx(i)
  }

  println(s"sram_addr_width_max: ${sram_addr_width_max}")

  val sram_addr_width_max_log2 = log2Ceil(sram_addr_width_max)

  val ip_shift_offsets = Seq.fill(num_procs)(Wire(UInt(sram_addr_width_max.W)))
  ip_shift_offsets.zip(pl_ip).zip(pl_valid).zipWithIndex.foreach({
    case (((iso, ip), valid), i) => {
      val ip_shift_offset = Wire(UInt(sram_addr_width_max.W))
      ip_shift_offset := ip << decs(i).io.offset(sram_addr_width_max_log2-1, 0)
      iso := Mux(valid && io.run, ip_shift_offset, 0.U)
    }
  })

  val recv_rd_en = Wire(UInt(1.W))
  val recv_rd_en_vec = Wire(Vec(num_procs, UInt(1.W)))
  decs.map(d =>
      d.io.prim === SRAMInputTypes.SRAMRdEn ||
      d.io.prim === SRAMInputTypes.SRAMRdWrEn)
        .zip(ip_shift_offsets)
        .map({ case (prim_match, iso) => Mux(prim_match, iso, 0.U) })
        .zip(recv_rd_en_vec)
        .map({ case (bit, rd_en) => rd_en := bit })
  recv_rd_en := recv_rd_en_vec.reduceTree(_ | _)
```

Listing 1: SRAM processor implementation benefits from Chisel's strong metaprogramming layer.

Emulation System

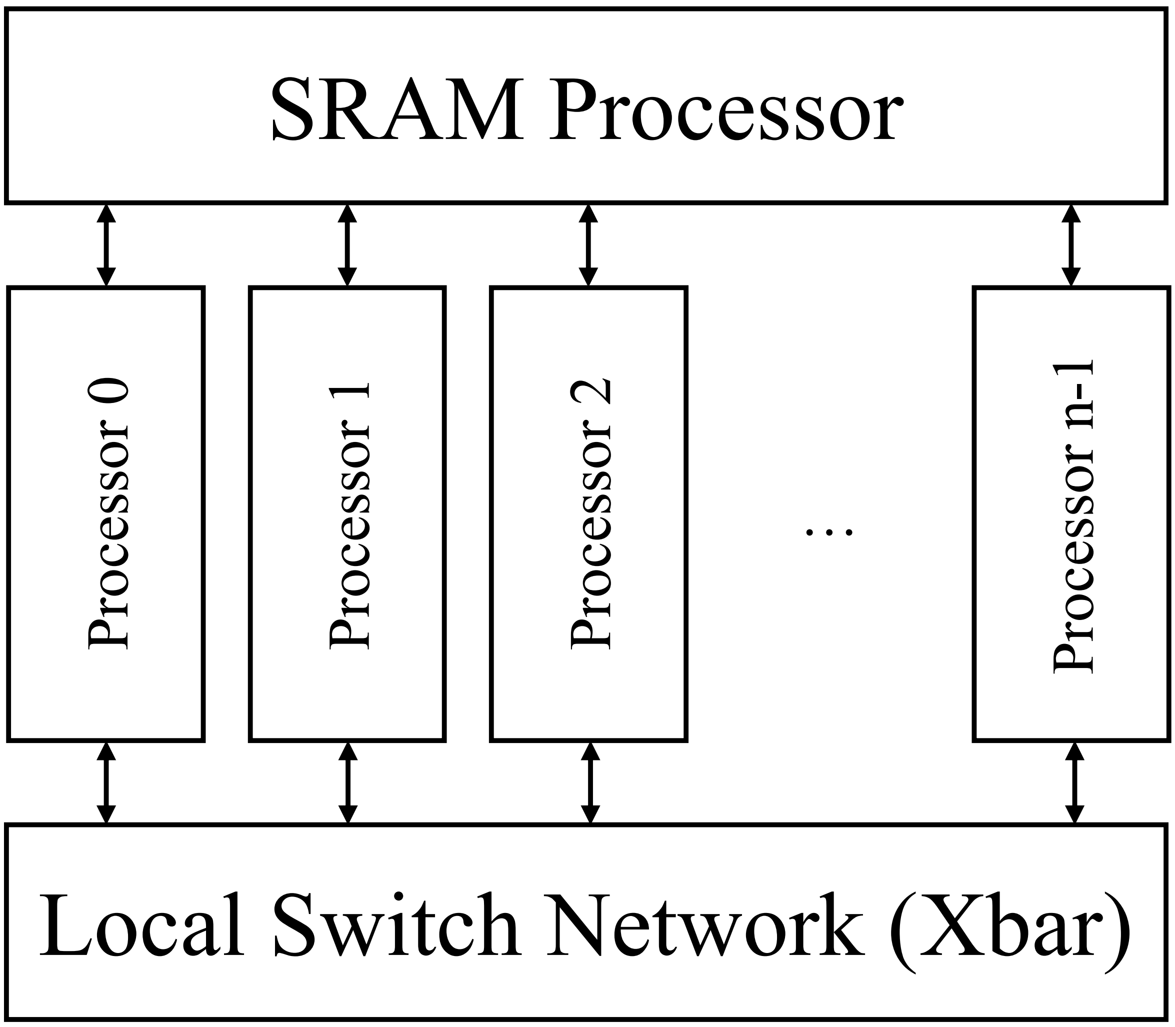

Figure 10: Emulation module.

Figure 10 illustrates the organization of an emulation module. Each module contains a set of n emulation processors connected both to a dedicated SRAM processor and to a local switch network. The local switch is implemented as a all-to-all crossbar, enabling any processor to exchange data with any other processor in the module. The latency of the switch is parameterized in the hardware and compiler to enable a tradeoff analysis. This ensures high communication bandwidth and eliminates internal connectivity bottlenecks. The SRAM processor sits above the array of processors and provides access to large emulated memory arrays without inflating the instruction count of the logic processors. Each emulation module also exposes ports for I/O interactions with the testbench. These ports allow external stimulus, such as input vectors or memory transactions, to be injected into the emulated design while also enabling observation of the system's outputs.

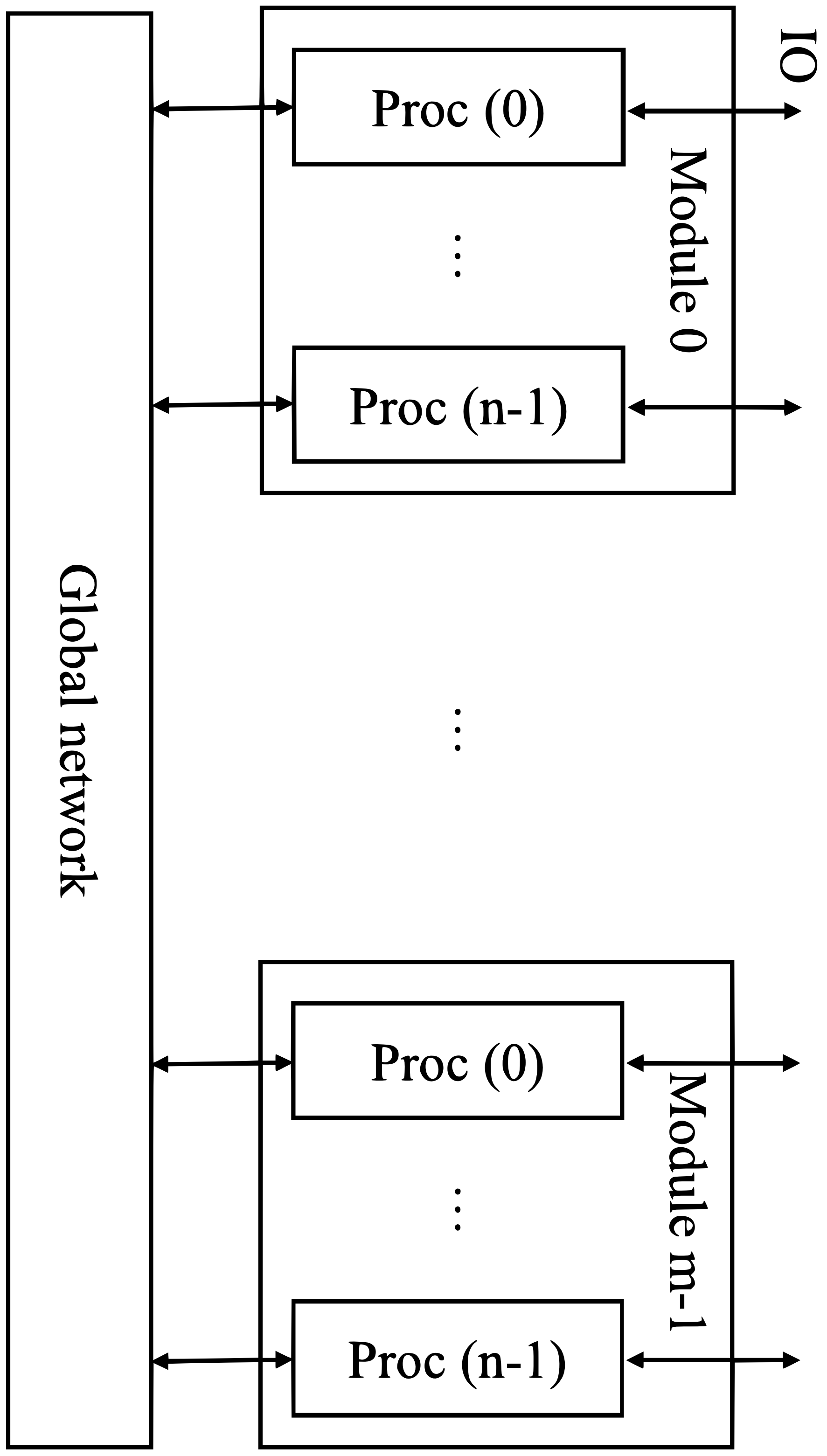

Figure 11: Emulation system.

Figure 11 shows how emulation modules compose into a full emulation system. Each module contains its own local crossbar and SRAM processor, but modules communicate with one another through a global interconnect. This global network provides the paths necessary for inter-module communication, supporting emulation of large-scale designs that cannot fit within a single module. The latency of the global network is also parameterized.

```rust
impl GlobalNetworkTopology {
    pub fn new(num_mods: u32, num_procs: u32) -> Self {
        let mut ret = GlobalNetworkTopology::default();
        if num_mods == 1 {
            return ret;
        }
        let num_mods_1 = num_mods - 1;
        let grp_sz = num_procs / num_mods_1;

        assert!(num_mods_1 & (num_mods_1 - 1) == 0, "num_mods should be 2^n +
        1");
        assert!(num_procs  & (num_procs - 1)  == 0, "num_procs should be 2^n +
        1");
        assert!(num_procs >= num_mods_1, "num_procs {} < num_mods - 1 {}",
        num_procs, num_mods_1);

        for m in 0..num_mods_1 {
            for p in 0..num_procs {
                let r = p % grp_sz;
                let q = (p - r) / grp_sz;
                let src = Coordinate { module: m, proc: p };
                let dst = if q == m {
                    let dm = num_mods_1;
                    let dp = p;
                    Coordinate { module: dm, proc: dp }
                } else {
                    let dm = q;
                    let dp = m * grp_sz + r;
                    Coordinate { module: dm, proc: dp }
                };
                ret.edges.insert(src, dst);
                ret.edges.insert(dst, src);
                ret.add_path(src, dst);
                ret.add_path(dst, src);
            }
        }
        return ret;
    }
    ...
}
```

Listing 2: GlobalNetworkTopology configuration code.

The topology of the global network is defined by a parameterizable software structure (`GlobalNetworkTopology`) as shown in Listing 2. The network is specified in terms of edges between processor coordinates and inter-module paths (`inter_mod_paths`) that describe routing across modules. For each processor in module `m`, the constructor derives a destination coordinate either in another peer module or in the distinguished extra module, depending on the grouping variable `q`. Bidirectional edges are then inserted into the network map, and corresponding paths are recorded in the routing table. The resulting structure provides a scalable communication network. Within each module, an all-to-all crossbar ensures high-bandwidth low-latency connectivity, while across modules, the compiler-defined `GlobalNetworkTopology` enables low-hop routes.

The hardware and compiler implementation are both highly parameterized to enable design space exploration and to support different configurations depending on whether the target platform is an FPGA or ASIC. Key tunable parameters include: the maximum program counter (Max PC) depth, the number of processors per module, and the number of modules per board. At the microarchitectural level, the pipeline stages of each processor can be flexibly configured. For instance, the decode and execute stages may be merged into a single cycle in cases when the `LDM` or `SDM` are combinational, or separated into multiple stages for higher clock frequency. Similarly, the latency of both the local and global interconnects can be configured, with optional pipelining registers inserted to achieve timing closure in FPGA implementations while being minimized for ASIC designs. The size of the dedicated SRAM processor is also configurable.

Compiler Passes

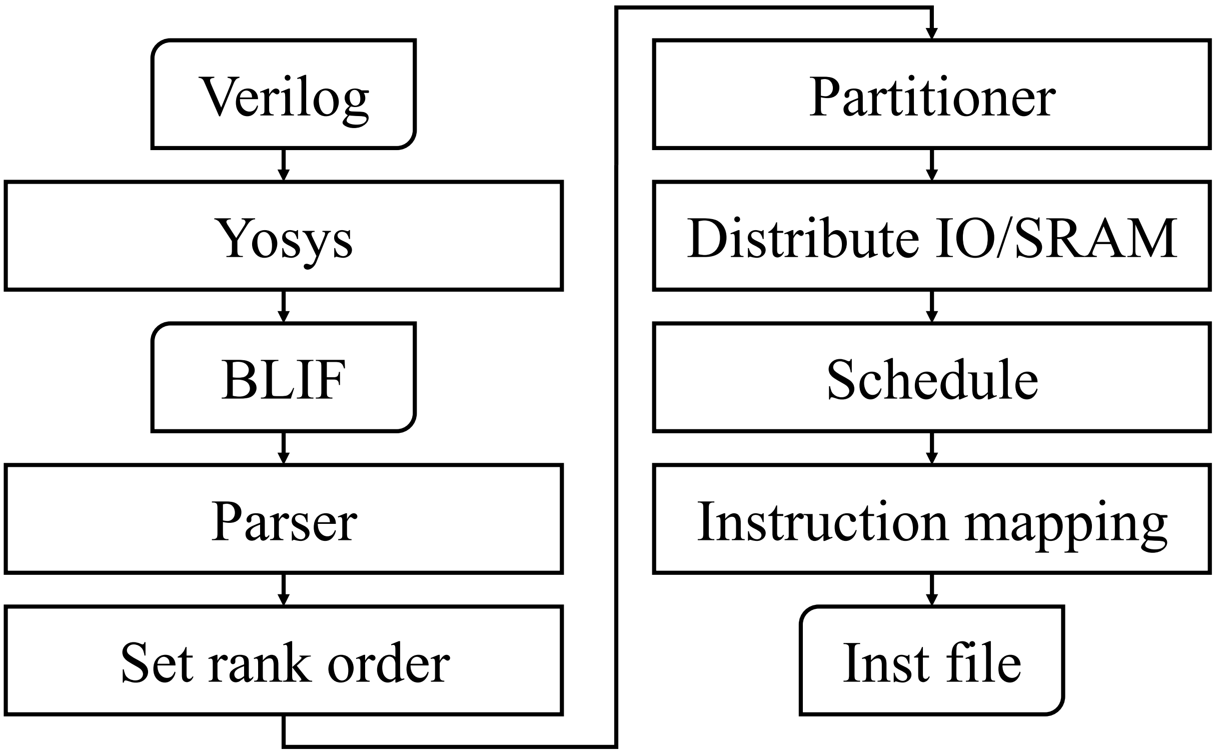

Figure 12: Emulator compiler stack.

Figure 12 shows the compiler flow used to translate a Verilog RTL description into executable instructions for the emulation processors.

The flow begins with Yosys, which performs logic synthesis on the input Verilog design. Yosys outputs a gate-level netlist in the BLIF (Berkeley Logic Interchange Format). Our custom compiler then parses this BLIF netlist and constructs a graph-based intermediate representation of the circuit.

The first transformation step is to set rank order, which corresponds to levelization of the circuit graph as described in the background section. This ensures that dependencies among gates are made explicit and that combinational paths are topologically ordered.

Next, the compiler partitions the graph hierarchically across modules and processors, assigning subgraphs to different emulation resources. We use the KaMinPar graph partitioner [9] which is an off the shelf hypergraph partitioner. Once partitioning is complete, IO and SRAM nodes are redistributed to the appropriate processors and modules as these are limited resources within the emulation system.

The core of the backend is the scheduler, which maps the levelized and partitioned graph into processor instruction streams. The scheduler extends a modified list scheduling algorithm, but must additionally account for hardware-specific constraints:

- It tracks processor instruction slots and network bandwidth to ensure that no interlocks occur
- When conflicts arise, the scheduler inserts NOP instructions to serialize operations and prevent hazards
- To maximize throughput, the scheduler attempts to route communication through the global switch in a single hop. However, if a path is already occupied, it falls back to a multi-hop sequence. First it sends the bit to another processor in the same module, then forwarding it to another module via the global switch, relaying it within that module's local network, and repeating this process until the destination processor is reached

After scheduling, the compiler performs instruction mapping, setting the fields of each instruction to match the operand, routing, and memory specifications derived from the graph. The result is an instruction file, which is loaded into the emulation system through a scan chain interface.

A key distinction between this processor-based emulator and FPGA-based emulation is the absence of a timing-driven place-and-route phase. FPGA compilation requires detailed placement of logic elements on the fabric and routing of signals through physical interconnects, followed by iterative timing closure. This process is computationally expensive and brittle, often taking hours to complete, and small design changes can require a full recompilation. Furthermore, synthesis itself has different objectives and constraints as the goal would be to minimize the logic depth as much as possible.

In contrast, the emulator compiler maps the design onto a virtual grid of processors with a fixed, low-latency interconnect. Because the interconnect is cycle-deterministic and uniform, there is no need for timing closure. Instead, the compiler relies on time-multiplexing and explicit scheduling to manage dependencies and communication. This results in much faster compilation times, less compilation failures, and more predictable performance, at the cost of lower density and raw execution speed compared to FPGAs.

The Modified List Scheduling Algorithm

The modified list scheduling algorithm is used to assign a PC to nodes assigned to each logic processor [10]. The modified list scheduling algorithm is best understood in relation to ASAP and ALAP scheduling. These baseline approaches expose the slack available for each node and provide the foundation for identifying critical paths.

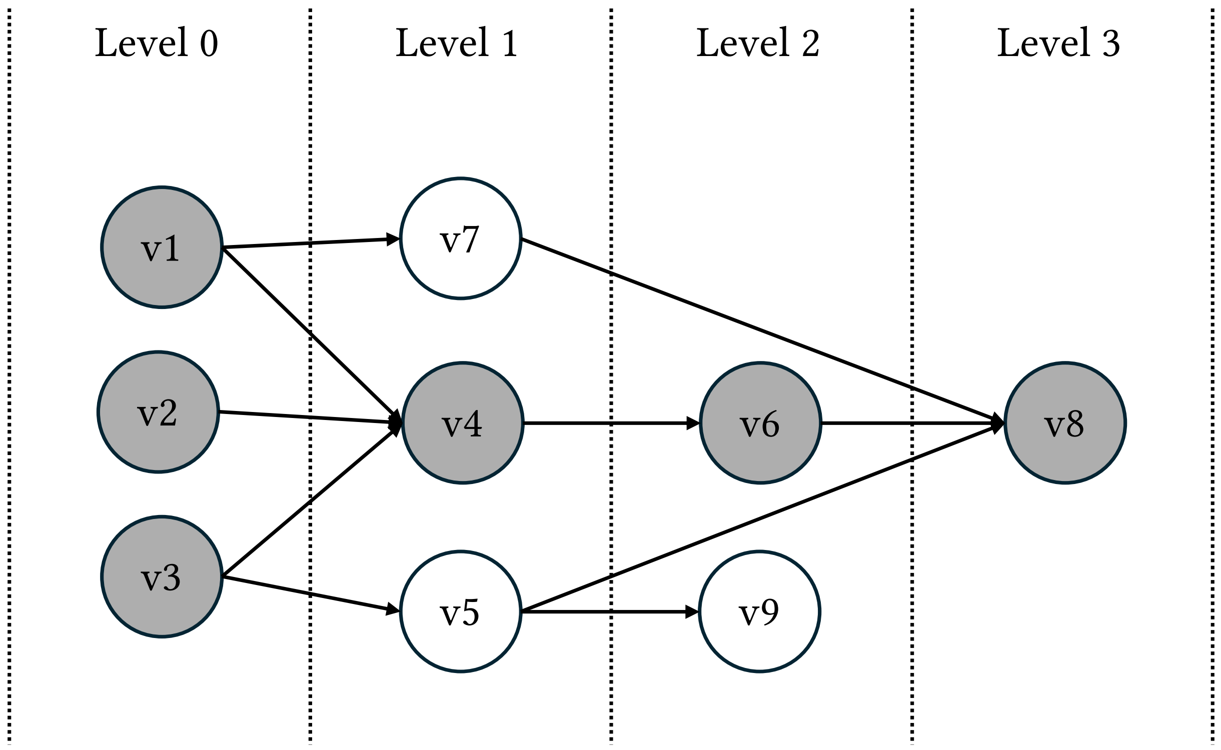

Figure 13: ASAP scheduling.

ASAP assigns to each node the earliest cycle in which it can execute, constrained only by the completion of its predecessors. The algorithm traverses the DAG in topological order starting from primary inputs:

```
ASAP(G):
  for v in inputs(G):
    # Inputs are given level 0
    ASAP[v] = 0
  for v in topological_order(G):
    if pred(v) != ∅:
      ASAP[v] = 1 + max(ASAP[u] for u in pred(v))
```

Listing 3: ASAP scheduling algorithm.

For example, in the DAG of Figure 13:

- v1, v2, v3 are inputs: ASAP = 0
- v4 depends on v1 and v2: ASAP(v4) = 1
- v6 depends on v4: ASAP(v6) = 2
- v8 depends on v5, v6, v7: ASAP(v8) = 3

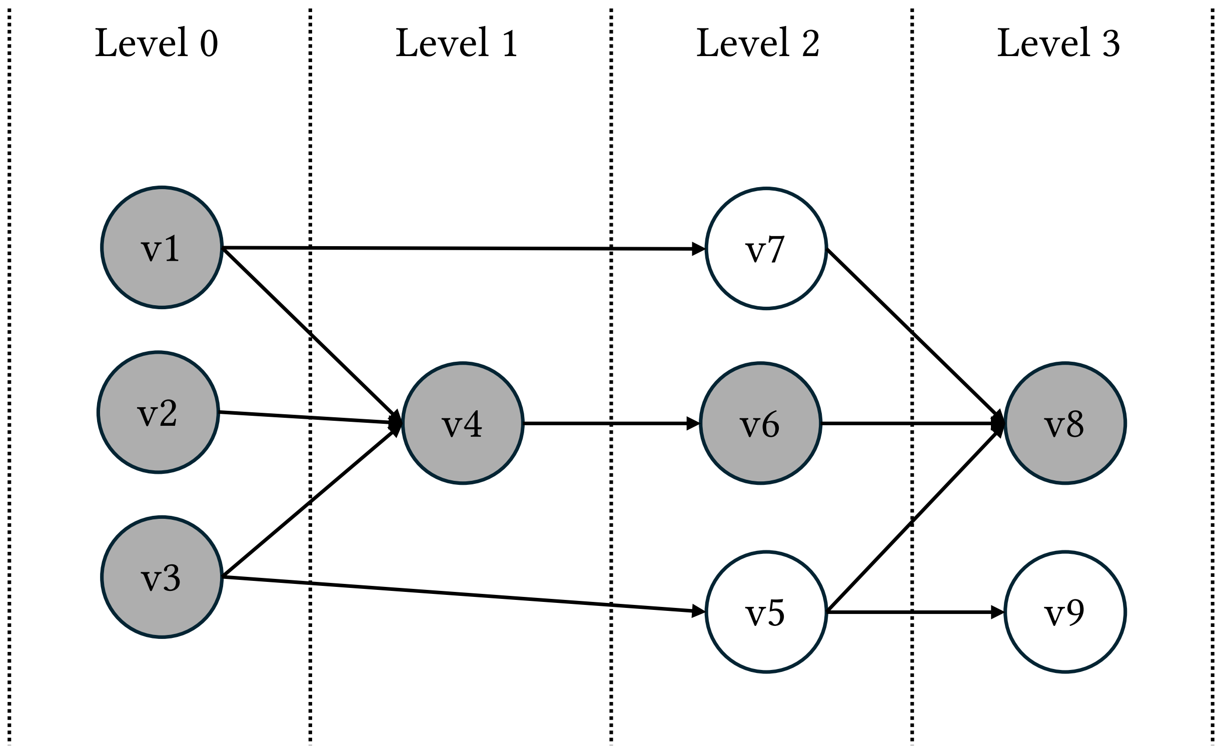

Figure 14: ALAP scheduling.

ALAP assigns to each node the latest cycle it may execute without delaying outputs. Traversal proceeds backward from primary outputs:

```
ALAP(G, L):
  for v in outputs(G):
    ALAP[v] = L
  for v in reverse_topological_order(G):
    if succ(v) != ∅:
      ALAP[v] = min(ALAP[w] for w in succ(v)) - 1
```

Listing 4: ALAP scheduling algorithm.

For example, in Figure 14:

- v9 is an output: ALAP(v9) = 3
- v6 feed v9: ALAP(v6) = ALAP(v9) - 1 = 2
- v4 feed v6: ALAP(v4) = ALAP(v6) - 1 = 1
- v1, v2, v3 feed v4: ALAP(v1) = ALAP(v2) = ALAP(v3) = ALAP(v4) - 1 = 0

Nodes with ASAP = ALAP have zero slack and are part of at least one critical path. In the DAG example, v2, v4, v6, and v8 are critical: they have no scheduling flexibility. These nodes must be scheduled exactly at their assigned cycle, or the overall latency will increase.

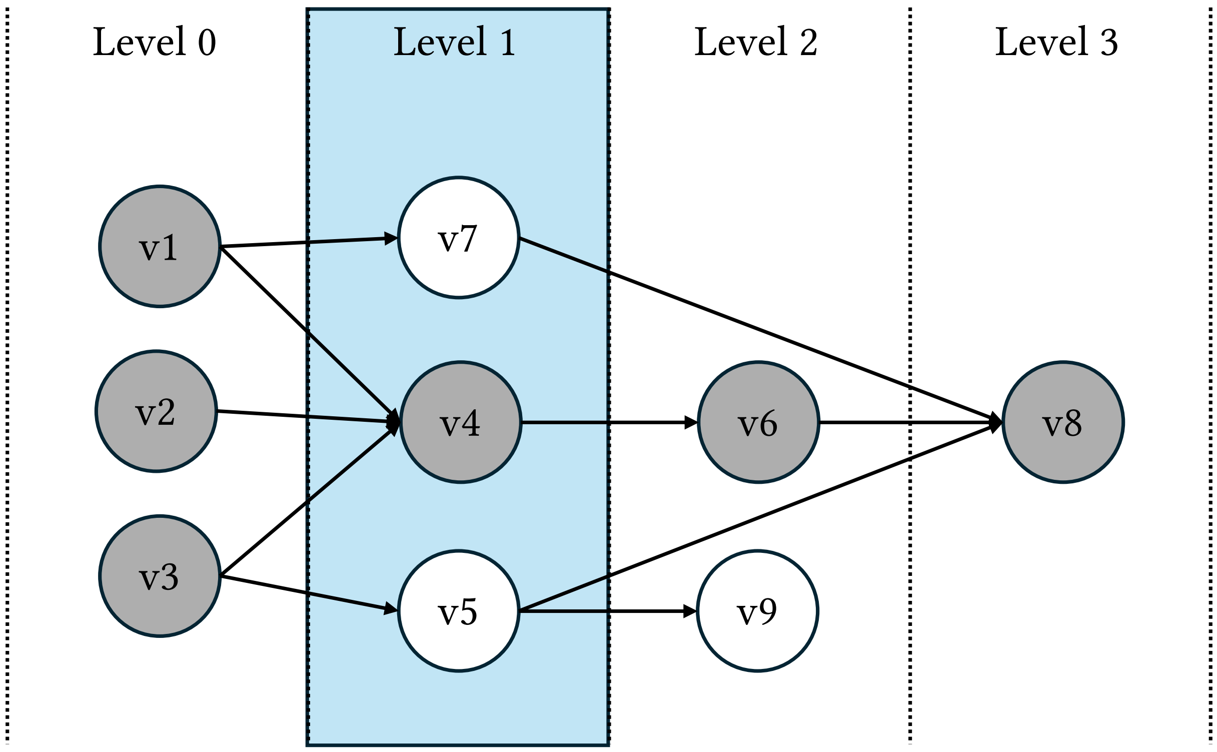

Figure 15: Baseline modified list scheduling algorithm.

The graph nodes are scheduled within each processor using the modified list scheduling algorithm. The modified list scheduling algorithm builds on ASAP and ALAP analysis, but is tailored to the execution model of a processor-based emulator.

From ASAP and ALAP bounds, nodes with ASAP = ALAP have zero slack and are classified as critical nodes. These nodes lie on the critical path of the DAG and cannot be shifted in time without extending the overall emulation length. Non-critical nodes have positive slack and can be scheduled flexibly between their ASAP and ALAP bounds.

The scheduler proceeds level by level:

1. Advance the host machine's PC until all critical nodes of the current level are scheduled. Each increment of the PC represents one cycle of the host machine. The PC is repeatedly stepped forward until every critical node at the level has been placed
2. Best-effort scheduling of non-critical nodes between PCs. Between the previous PC (where the level began) and the current maximum PC (where the last critical node was placed), the scheduler opportunistically inserts non-critical nodes as processor resources allow. This balances load and reduces idle time, while still respecting each node's ASAP–ALAP mobility
3. Increment to the next level. Once all nodes of the current level are scheduled, the algorithm advances to the next ASAP level and repeats the process

```
ModifiedListSchedule(G):
  compute ASAP[v], ALAP[v]
  PC = 0
  for level = 0 to L:
    crit_nodes = {v | ASAP[v] == ALAP[v] == level}
    noncrit_nodes = {v | ASAP[v] == level and v not in crit_nodes}

    # 1. Schedule critical nodes, incrementing PC
    while true:
      for v in crit_nodes:
        if schedule(v, PC):
          crit_nodes.remove(v)
      if crit_nodes.is_empty():
        break
      else:
        PC += 1

    # 2. Fill best-effort noncritical nodes
    #     into [start_PC, current_PC)
    for level in start_PC..PC:
      for v in noncrit_nodes:
        if schedule(v, level):
          noncrit_nodes.remove(v)

    # 3. Proceed to next level
    start_PC = PC
```
Listing 5: Modified list scheduling algorithm.

For example

- Level 0: v1, v2, v3 are inputs. All nodes are scheduled immediately
- Level 1: v4 is critical so the PC is stepped until both are placed. Meanwhile, v5 and v7 are inserted best-effort into the same PC range
- Level 2: v6 is critical and the PC is advanced until it is placed. v9 is scheduled opportunistically
- Level 3: v9 is critical $\longrightarrow$ placed last.

This produces a schedule in which critical path nodes strictly determine the extent of each level, while non-critical nodes are packed into the available PC slots within that window.

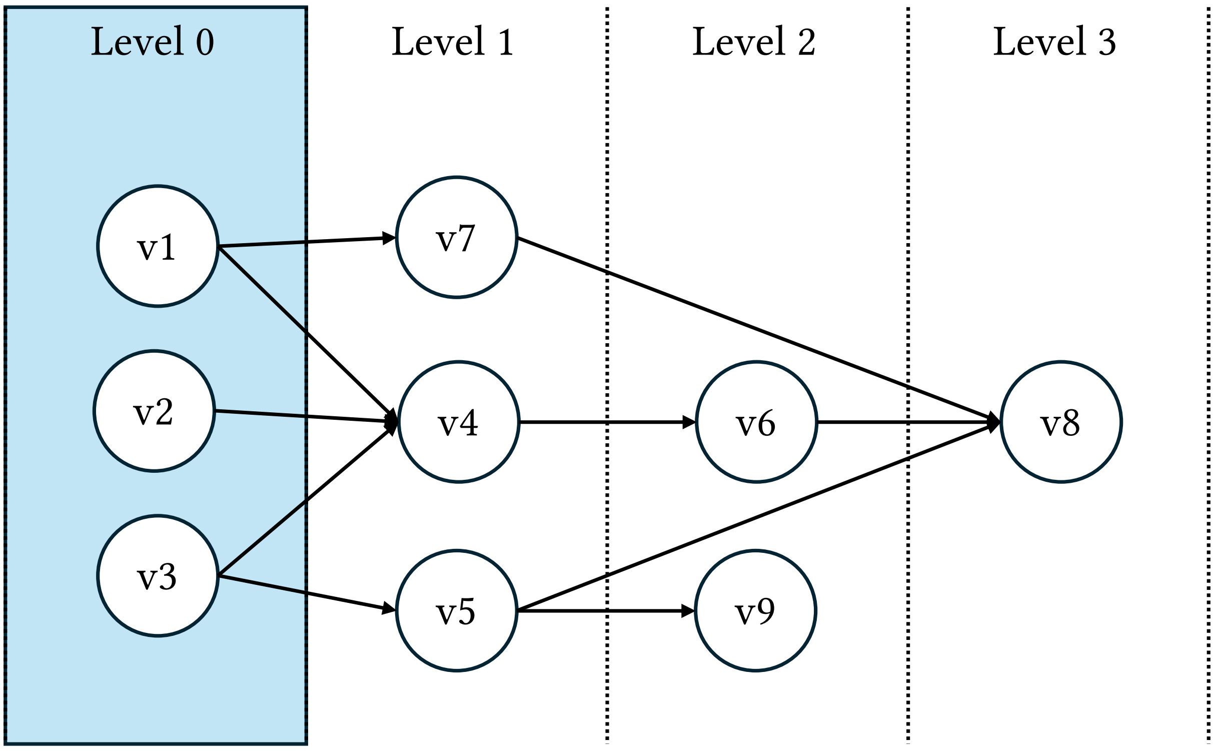

Figure 16: Improving the modified list scheduling algorithm.

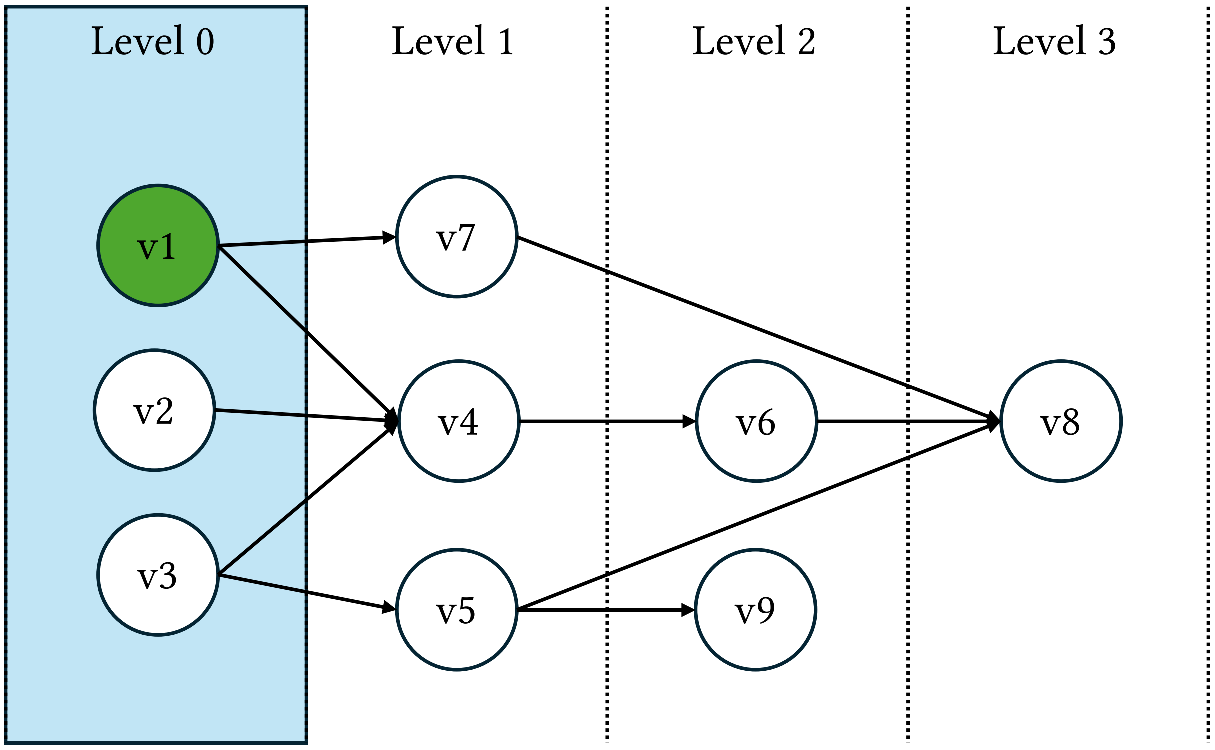

Figure 17: v1 is scheduled.

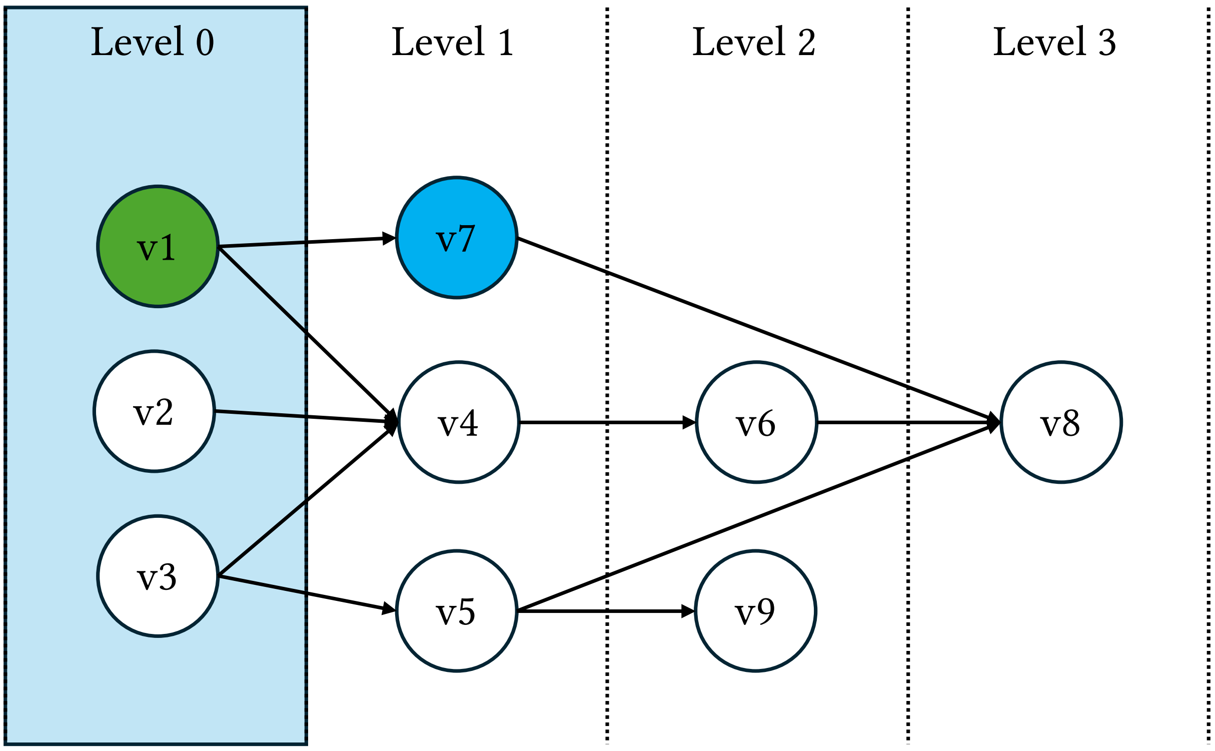

Figure 18: We are scheduling level 0, but v7 is schedulable.

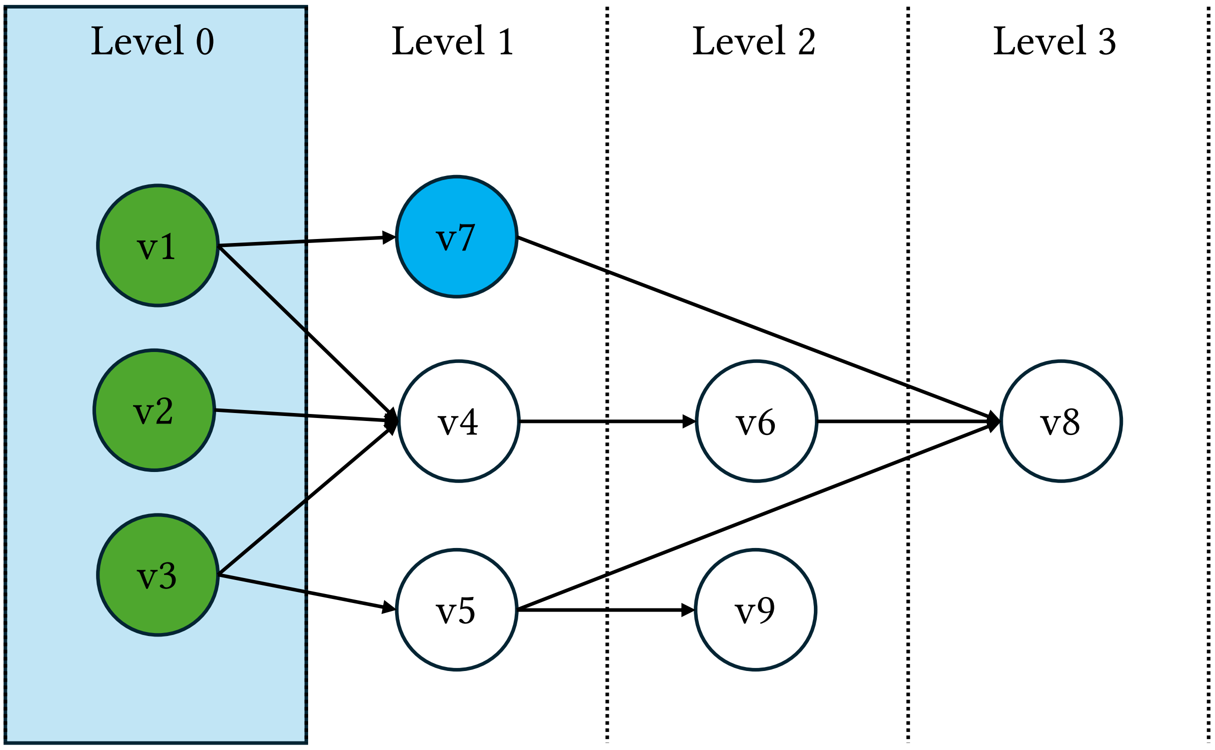

Figure 19: v2 and v3 are scheduled.

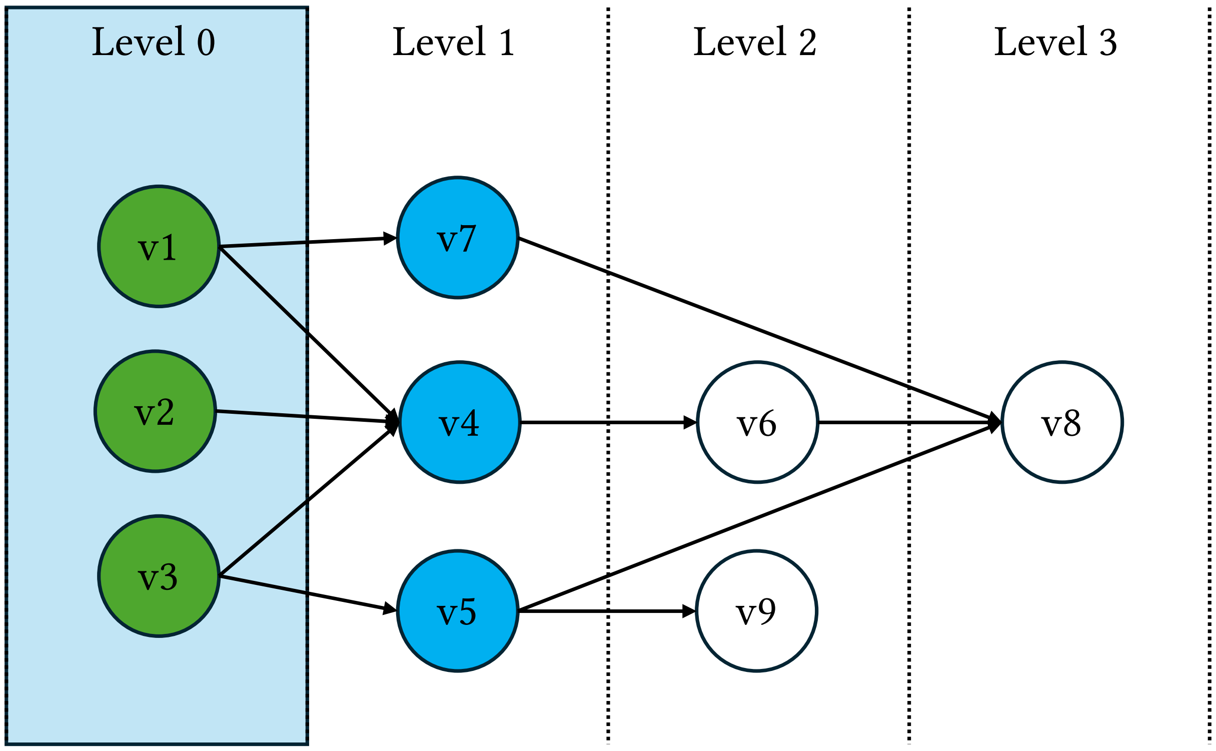

Figure 20: We are scheduling level 0, but v4 and v5 is also schedulable.

The sequence of above figures illustrates how the modified list scheduling algorithm can be pushed further to improve processor utilization.

- Figure 16 establishes the baseline: we are scheduling nodes in Level 0
- Figure 17 shows the first scheduling step, where node v1 is scheduled
- Figure 18 highlights that even though we are nominally in Level 0, node v7 (in Level 1) becomes schedulable because all of its parent nodes have already executed
- Figure 19 schedules the remaining nodes v2 and v3 in Level 0
- Figure 20 demonstrates that v4 and v5 can also be scheduled immediately even though they are in the next level, because all of their dependencies have been satisfied

This observation reveals an opportunity for further optimization: the rigid level-by-level execution in the modified list scheduling algorithm is unnecessarily conservative. Once critical nodes and non-critical nodes have been scheduled in a level, their child nodes that are now free of dependencies can also be executed opportunistically, even if they belong to a higher level. To exploit this, we propose a third iteration step in the modified list scheduling algorithm as shown in Listing 6:

```
ImprovedModifiedListSchedule(G):
  compute ASAP[v], ALAP[v]
  PC = 0
  for level = 0 to L:
    crit_nodes = {v | ASAP[v] == ALAP[v] == level}
    noncrit_nodes = {v | ASAP[v] == level and v not in crit_nodes}

    # 1. Schedule critical nodes, incrementing PC
    while true:
      for v in crit_nodes:
        if schedule(v, PC):
          crit_nodes.remove(v)
      if crit_nodes.is_empty():
        break
      else:
        PC += 1

    # 2. Fill best-effort noncritical nodes
    for level in start_PC..PC:
      for v in noncrit_nodes:
        if schedule(v, level):
          noncrit_nodes.remove(v)

    # 3. Collect extra nodes that are schedulable and schedule them
    extra_nodes = collect_extra_schedulable_nodes()
    for level in start_PC..PC:
      for v in extra_nodes:
        if schedule(v, level):
          noncrit_nodes.remove(v)

    # 4. Proceed to next level
    start_PC = PC
```

Listing 6: Improved modified-list-scheduling algorithm.

The overall algorithm executes as follows:

1. Critical nodes first: schedule all nodes with zero slack at the current level
2. Non-critical nodes next: schedule other nodes in the current level as best effort, within their ASAP–ALAP mobility range
3. Opportunistic scheduling of child nodes: collect all child nodes of the just-scheduled nodes that now have all their predecessors executed, and schedule them immediately, regardless of their assigned level

Scheduling Optimization Results

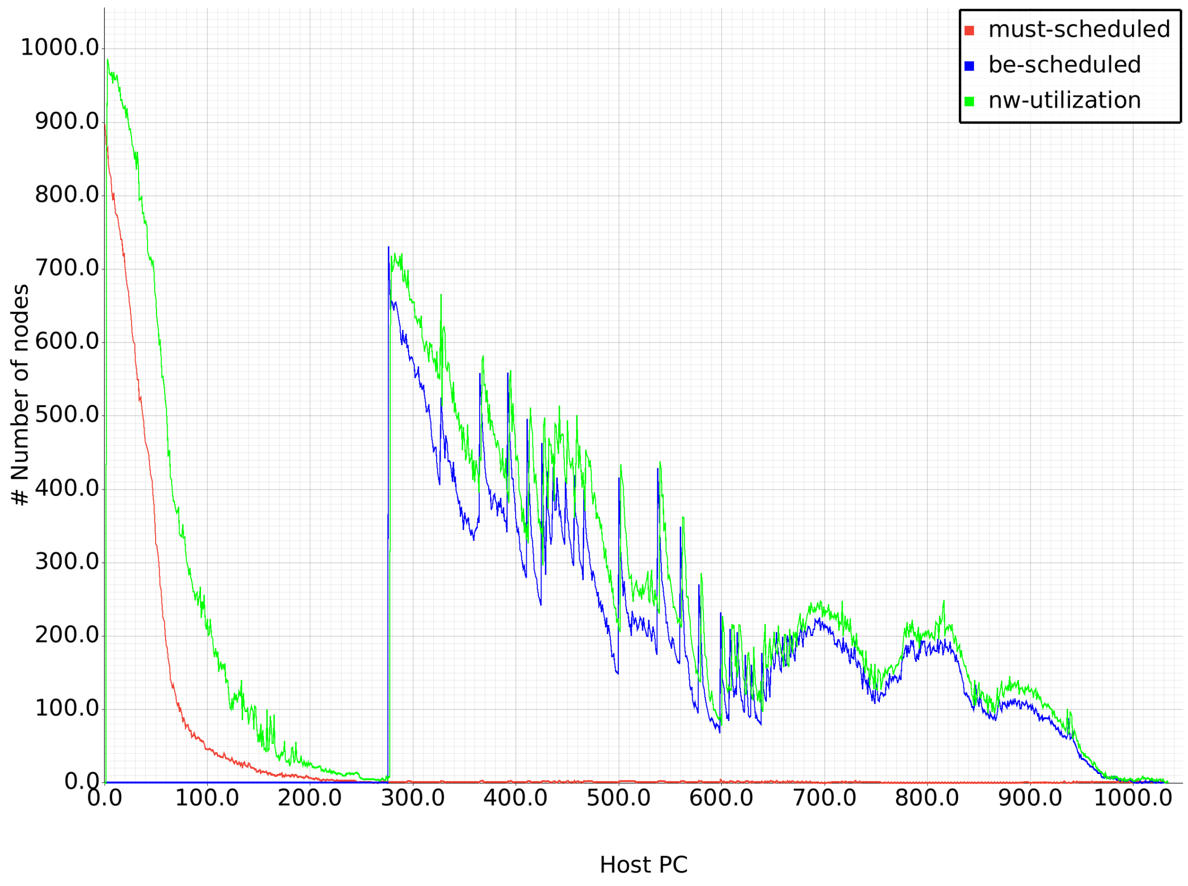

Figure 21: Scheduling progress for baseline modified list scheduling algorithm.

Figure 21 illustrates scheduling progress under the baseline modified list scheduling algorithm for a single core Rocket configuration. The x-axis represents the host program counter (PC) steps, while the y-axis represents the number of nodes scheduled at each step. Three curves are shown: must-scheduled nodes (red), best-effort scheduled nodes (blue), and network utilization (green).

At the beginning of the schedule, a large number of critical nodes (must-scheduled) dominate the workload, reflected by the steep red curve that quickly decays to zero. Once these critical nodes are placed, the schedule enters a phase where the number of critical nodes are significantly less and the scheduling of non-critical nodes start to dominate. This activity is captured by the blue curve, which begins after the red curve tapers off and exhibits significant oscillations as available slack is opportunistically consumed. The green curve, representing the number of busy output network ports, closely tracks the blue curve, showing that network traffic is strongly correlated with best-effort scheduling.

Overall, the baseline algorithm ensures correctness by prioritizing critical nodes, but it leaves noticeable gaps in utilization. The transition between critical and best-effort phases creates underutilized host PC intervals, and network activity fluctuates heavily due to the rigid separation between must-scheduled and best-effort scheduling.

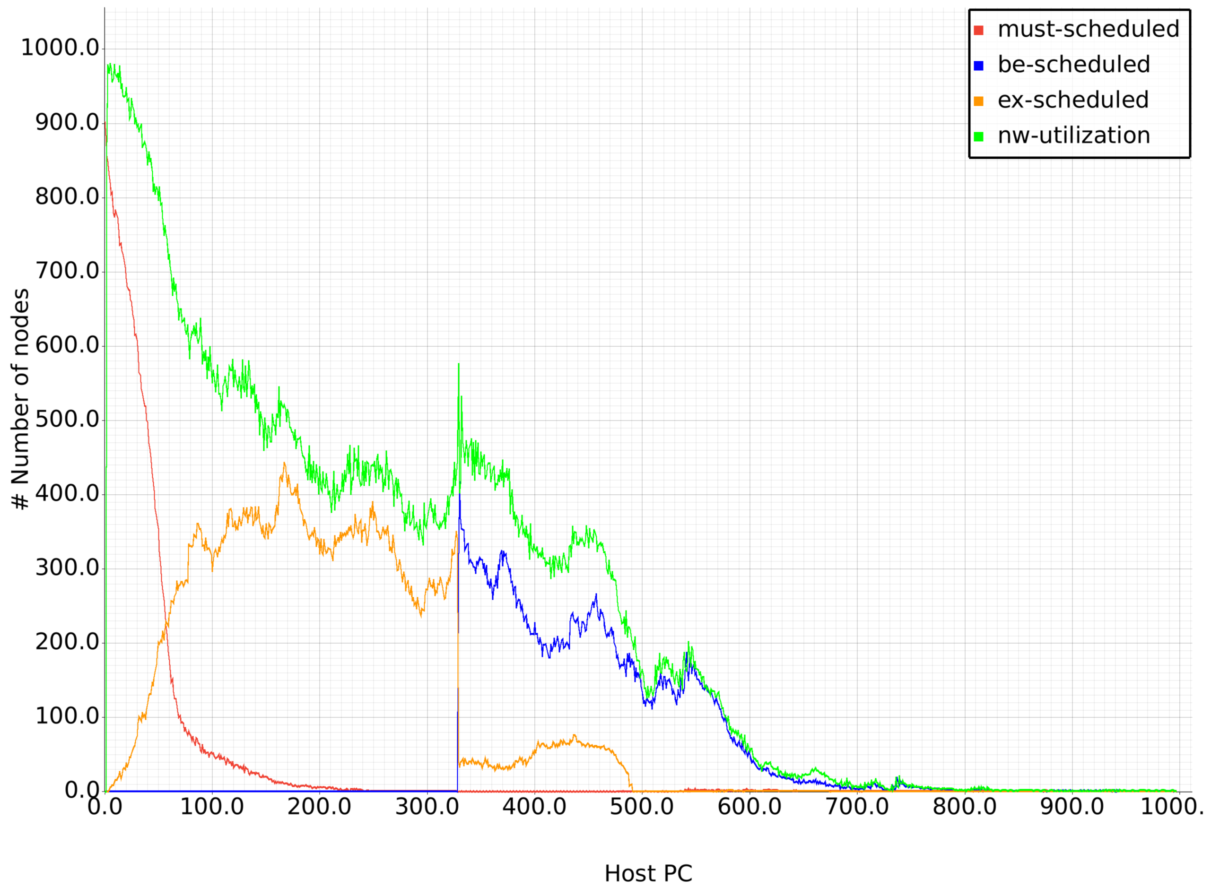

Figure 22: Scheduling progress for optimized modified list scheduling algorithm.

Figure 22 presents results for the optimized modified list scheduling algorithm, which introduces a third scheduling tier: extra-scheduled nodes (orange). These are nodes that become executable earlier than their nominal level once their parent dependencies have been satisfied, allowing them to be scheduled opportunistically across host PC steps.

Compared to the baseline, the must-scheduled (red) and best-effort (blue) curves follow a similar progression, but the key difference is the emergence of the orange ex-scheduled curve. These extra nodes occupy otherwise idle cycles between critical and best-effort scheduling phases. As a result, the workload is more evenly distributed across host PC steps. The network utilization curve (green) confirms this effect: it is significantly smoother and remains higher for longer, indicating that more output ports are kept active throughout the schedule.

This optimization reduces idle cycles, shortens the overall scheduling horizon, and improves parallelism. By dynamically exploiting node readiness rather than adhering strictly to ASAP/ALAP level boundaries, the algorithm overlaps scheduling across levels and increases throughput. The result is a more efficient use of hardware resources, particularly the interconnect fabric, which is reflected in the higher and more sustained network utilization compared to the baseline.

Graph Partitioning Optimizations

```rust
fn edge_weight(circuit: &Circuit, src_idx: &NodeIndex, dst_idx:
&NodeIndex) -> f32 {
    let dst = circuit.graph.node_weight(*dst_idx).unwrap().info();
    let src_child_cnt = circuit.graph.neighbors_directed(*src_idx,
    Outgoing).count();
    if dst.rank.alap - dst.rank.asap == 0 {
        0.0
    } else {
        (src_child_cnt - 1) as f32 / src_child_cnt as f32
    }
}
```

Listing 7: Edge weight assignments for better partitioning QoR.

To further improve resource utilization, we introduced an optimization during the graph partitioning stage by assigning weights to edges as shown in Listing 7. The intuition behind this technique is to balance communication overhead against the potential for increased parallelism when splitting parent and child nodes across partitions. Specifically, if a destination node lies on the critical path (zero slack between its ASAP and ALAP levels), the edge weight is set to zero, ensuring that such dependencies remain within the same partition and minimizing communication overhead. For non-critical nodes, the edge weight is scaled according to the fan-out of the source node. The weight formula $\frac{\text{src\_child\_cnt} - 1}{\text{src\_child\_cnt}}$ reflects the degree of parallelism gained by separating a child from its siblings. For example, with two children the gain is 1/2, and with three children the gain is 2/3, thereby favoring partitions that maximize useful concurrency.

This heuristic proved highly effective in practice. When incorporated into the partitioning algorithm, the resulting schedules exhibited significantly better utilization of emulation resources. In particular, the maximum schedule length (i.e., the number of host PC steps required) decreased by approximately 30%, leading directly to faster execution. Notably, replacing the parallelism-aware term with a constant weight of 1.0 eliminated these gains, reducing machine utilization back to its baseline of roughly 25% for a SoC containing a single Rocket core. The results confirm that carefully balancing edge weights to capture both communication and parallelism effects is critical for achieving efficient partitioning and improved emulation throughput.

FPGA Prototyping

We prototyped the emulation system on a Xilinx Alveo U250 FPGA. One of the major challenges encountered was the high resource consumption of the local data memory (`LDM`) and switch data memory (`SDM`). In the baseline design, both memories were implemented using FPGA LUTs. The decoding logic required for address selection consumed a large fraction of the host FPGA's LUT budget, severely limiting the number of emulation processors that could fit on the device. Moreover, because these memories were configured as multi-ported with a number of read ports equal to the LUT inputs in the instruction memory (three in our prototype), the FPGA synthesis tool replicated the structures to satisfy port requirements. This further exacerbated the resource usage and made scaling impractical.

To address this, we replaced the original Reg/Vec-based structures with LUTRAM-based memories. This change allowed us to avoid burning LUTs on address decoding logic and instead exploit the FPGA's distributed memory primitives. As shown in Table 1, the baseline implementation of the LDM and SDM consumed approximately 1,300 and 891 LUTs as well as 1023 flip-flop (FF)s each. After replacing them with LUTRAM-backed black-box modules, these dropped to only 108 and 101 LUTs each as shown in Table 2 while burning zero FFs. Overall, the total LUT usage per processor was reduced by nearly an order of magnitude, while the utilization of dedicated LUTRAM primitives increased correspondingly. This reduction freed up sufficient LUT resources to scale the prototype to emulate entire SoCs generated by Chipyard.

Instance	Module	Total LUTs	Logic LUTs	LUTRAMs	FFs
procs_0	Processor_426	2820(0.16%)	2820(0.16%)	0(0.00%)	2084(0.06%)
(procs_0)	Processor_426	528(0.03%)	528(0.03%)	0(0.00%)	38(0.01%)
imem	InstMem_472	101(0.01%)	101(0.01%)	0(0.00%)	0(0.00%)
ldm	DataMemory_473	1300(0.08%)	1300(0.08%)	0(0.00%)	1023(0.03%)
sdm	DataMemory_474	891(0.05%)	891(0.05%)	0(0.00%)	1023(0.03%)

Table 1: Resource consumption before LUTRAM optimization.

Instance	Module	Total LUTs	Logic LUTs	LUTRAMs	FFs
procs_14	Processor_5281	282(0.02%)	90(0.01%)	192(0.02%)	0(0.00%)
(procs_14)	Processor_5281	30(0.01%)	30(0.01%)	0(0.00%)	0(0.00%)
imem	InstMem_5869	43(0.01%)	43(0.01%)	0(0.00%)	0(0.00%)
ldm	DataMemory_5870	108(0.01%)	12(0.01%)	96(0.01%)	0(0.00%)
sdm	DataMemory_5871	101(0.01%)	5(0.01%)	96(0.01%)	0(0.00%)

Table 2: Resource consumption after LUTRAM optimization.

The initial timing bottleneck in the design was the global network switch, which spanned across super logic regions (SLRs) of the FPGA. Since the system architecture and RTL implementation were fully parameterized, we were able to pipeline the global switch by adding an additional register stage. This cut the longest paths through the switch and allowed the design to meet higher target frequencies. Importantly, while the maximum clock frequency improved, overall machine utilization remained within 1% of the original configuration. This indicates that system throughput was not constrained by switch delay, but by other bottlenecks in the emulation system.

The next critical path was in the distribution of DMA input bits from the host to the emulator's top-level I/O ports. These inputs followed a ready/valid interface as the emulator can consume new input bits only when the current cycle completes and the new data is valid. To pipeline this interface, we introduced skid buffers that decoupled the DMA source from the emulator's ready signal, allowing higher-frequency operation without violating ready-valid semantics.

After these modifications, the prototype achieved a maximum operating frequency of 80 MHz. At this point, the critical path shifted to the SRAM address generation logic. Specifically, the large multiplexers that aggregate index signals from emulation processors into the SRAM port input registers now dominate timing. While further pipelining could reduce this delay, such optimizations were left as future work.

Initial Results

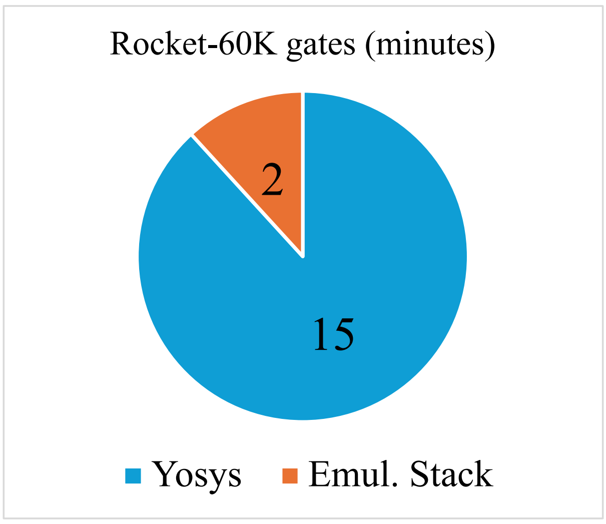

Figure 23: Compilation time of Rocket.

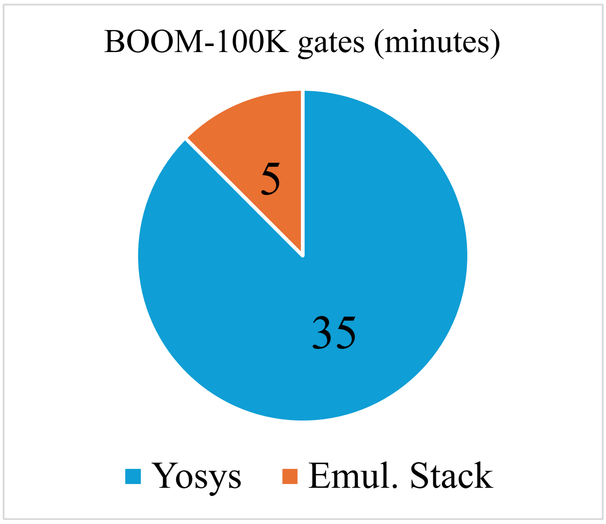

Figure 24: Compilation time of BOOM.

Figure 23 and Figure 24 shows the compilation time breakdown of Rocket and BOOM respectively. We can see that most of the compilation time is spent on synthesis. Compared to FPGA builds, the compilation time for Rocket reduces from 45 minutes to 17 minutes (3X decrease) while for BOOM it reduces from 3 hours to 40 minutes (4.5X decrease). We leave further compilation time reduction through incremental synthesis for future work.

Future Work and Takeaways

The development of this processor-based emulation system highlights both the opportunities and the inherent difficulties in emulation systems. First of all, throughput is strongly shaped by host–emulator synchronization. Even with efficient on-chip execution, delays at the boundary with the host system impose global limits on performance, a manifestation of Amdahl's Law. In terms of the emulator microarchitecture, resource profile of the design reveals another key insight that network structures are comparatively inexpensive and have a insignificant affect on performance, while instruction memories and multiported data memories dominate resource consumption. Finally, timing analysis exposed friction between bitwise and wordwise elements. The critical path resides in the SRAM processor input reconstruction logic, where a wide multiplexer bridges the gap between bitwise elements and blockwise SRAM interfaces.

Several promising directions remain to be explored in order to advance the practicality and competitiveness of processor-based emulation. First, reducing compilation time through incremental synthesis is critical to achieving the vision of a rapid run–edit–debug loop for hardware design. While our current flow already avoids the place-and-route stage inherent to FPGA-based emulation, a complete realization of interactive design requires that small RTL modifications trigger proportionally small recompilation steps. Developing an intermediate representation and synthesis flow that supports fine-grained reuse will therefore be an important step forward.

Second, there is a need to improve the microarchitecture of the logic processors themselves. The current design is functionally adequate but limited in simulation capacity, which ultimately constrains the size of target systems that can be emulated.

Finally, while the optimizations described in this work improved processor utilization significantly, the system still falls short of commercial emulators, which routinely sustain utilization rates of 40–80%. Bridging this gap will require a combination of enhanced scheduling heuristics, and more communication-aware partitioning and placement. Achieving consistently higher utilization remains a challenge for future research.

Bibliography

[1] S. Beamer, T. Nijssen, K. Pandian, and K. Zhang, "ESSENT: A High-Performance RTL Simulator," in *Workshop on Open-Source EDA Technology (WOSET), at the International Conference on Computer-Aided Design (ICCAD)*, Nov. 2021.

[2] H. Wang and S. Beamer, "RepCut: Superlinear Parallel RTL Simulation with Replication-Aided Partitioning," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, in ASPLOS 2023. Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 572–585. doi: 10.1145/3582016.3582034.

[3] H. Wang, T. Nijssen, and S. Beamer, "Don't Repeat Yourself! Coarse-Grained Circuit Deduplication to Accelerate RTL Simulation," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, in ASPLOS '24. Hilton La Jolla Torrey Pines, La Jolla, CA, USA: Association for Computing Machinery, 2025, pp. 79–93. doi: 10.1145/3622781.3674184.

[4] Synopsys, Inc., "VCS Functional Verification Solution."

[5] Cadence Design Systems, Inc., "Xcelium Logic Simulator."

[6] W. Snyder, "Verilator — the fastest Verilog/SystemVerilog simulator."

[7] G. Pfister, "The Yorktown Simulation Engine: Introduction," in *19th Design Automation Conference*, 1982, pp. 51–54. doi: 10.1109/DAC.1982.1585479.

[8] Cadence Design Systems, Inc., "Cadence Palladium Emulation Platform." 2025.

[9] L. Gottesbüren, T. Heuer, P. Sanders, C. Schulz, and D. Seemaier, "Deep Multilevel Graph Partitioning," in *29th Annual European Symposium on Algorithms, ESA 2021*, in LIPIcs, vol. 204. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 1–17. doi: 10.4230/LIPIcs.ESA.2021.48.

[10] A. A. Yazdanshenas, "Hardware Design and CAD for Processor-based Logic Emulation Systems." 2006.