

LLHD: A Multi-level Intermediate Representation for Hardware Description Languages

Fabian Schuiki
 Integrated Systems Laboratory (IIS)
 ETH Zürich
 Zürich, Switzerland
 fschuiki@iis.ee.ethz.ch

Tobias Grosser
 Scalable Parallel Computing Laboratory (SPCL)
 ETH Zürich
 Zürich, Switzerland
 tobias.grosser@inf.ethz.ch

Andreas Kurth
 Integrated Systems Laboratory (IIS)
 ETH Zürich
 Zürich, Switzerland
 akurth@iis.ee.ethz.ch

Luca Benini
 Integrated Systems Laboratory (IIS)
 ETH Zürich
 Zürich, Switzerland
 lbenini@iis.ee.ethz.ch

Abstract

Modern Hardware Description Languages (HDLs) such as SystemVerilog or VHDL are, due to their sheer complexity, insufficient to transport designs through modern circuit design flows. Instead, each design automation tool lowers HDLs to its own Intermediate Representation (IR). These tools are monolithic and mostly proprietary, disagree in their implementation of HDLs, and while many redundant IRs exist, no IR today can be used through the entire circuit design flow. To solve this problem, we propose the LLHD multi-level IR. LLHD is designed as simple, unambiguous reference description of a digital circuit, yet fully captures existing HDLs. We show this with our reference compiler on designs as complex as full CPU cores. LLHD comes with lowering passes to a hardware-near structural IR, which readily integrates with existing tools. LLHD establishes the basis for innovation in HDLs and tools without redundant compilers or disjoint IRs. For instance, we implement an LLHD simulator that runs up to 2.4× faster than commercial simulators but produces equivalent, cycle-accurate results. An initial vertically-integrated research prototype is capable of representing all levels of the IR, implements lowering from the behavioural to the structural IR, and covers a sufficient subset of SystemVerilog to support a full CPU design.

CCS Concepts: • **Hardware** → **Hardware description languages and compilation**; • **Computing methodologies** → *Simulation languages*; • **Software and its engineering** → *Compilers*.

Keywords: hardware description languages, intermediate representations, transformation passes

1 Introduction

The workflow we use today to design digital circuits, including CPUs and GPUs, has a severe redundancy problem. The sheer complexity of modern designs, requiring billions of transistors to be placed and costing millions of dollars on

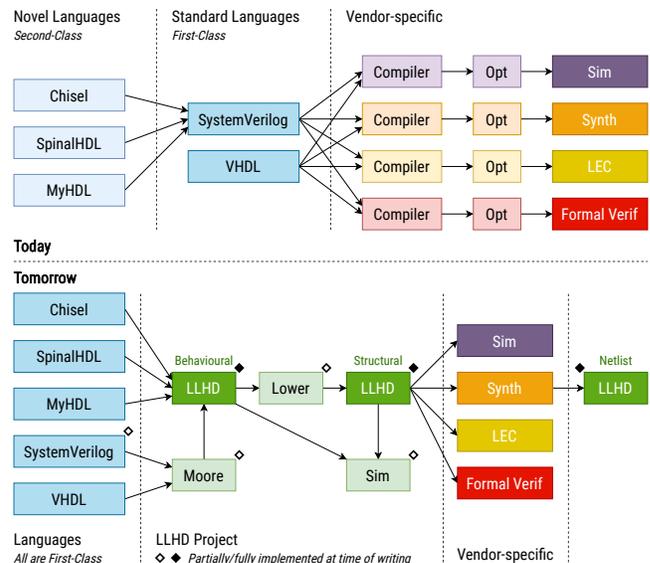


Figure 1. Redundancy in today’s hardware design flow (top). Replacement flow with Moore as compiler frontend and LLHD as unifying IR (bottom). Maturity of the implementation at the time of writing is indicated.

each silicon iteration, has given rise to a very deep design toolchain — much more so than in software development. Along their way to becoming silicon, digital designs pass simulators, formal verifiers, linters, synthesizers, logical equivalence checkers, and many other tools. HDLs are used as the input to these tools, with SystemVerilog and VHDL being widely used in industry. Both are highly complex languages: their specifications span 1275 and 626 pages of text, respectively [1, 4]. The aforementioned tools come from different vendors (often *by design* to rule out systematic sources of error), each of which has its own implementation of the complex language standards, as depicted in Figure 1. Hardware

designers have to “hope”, for example, that the circuit synthesizer interprets the semantics of a design in the same way as the simulator used for verification, and the standards can be very subtle [26]. As a consequence, designers resort to a “safe subset” of each language known to produce identical results across most tools, which precludes the practical use of many distinguishing high-level features of SystemVerilog and VHDL.

All of our modern computing infrastructure depends on getting this tool flow right. As a consequence, the Electronic Design Automation (EDA) industry invests significant resources into finding safe subsets of the languages, establishing coding styles, and developing linting tools to enforce adherence to such conventions [26]. In contrast to software development, where the advent of IRs and frameworks such as LLVM [16] has provided a productive platform for open development, the hardware design flow remains isolated and vendor-locked. The EDA market is dominated by closed-source, proprietary, monolithic toolchains, which have been highly optimized over decades. New open-source tools face the hurdle of implementing complex language frontends before being able to compete. We observe that hardware engineering and compiler design communities have evolved in isolation, even though many optimizations and methodological improvements readily apply to both worlds. This is to hardware engineering’s severe disadvantage, as we see ample opportunity for the two communities to connect, exchange, and benefit from each other.

We propose *LLHD*, an IR to represent digital circuits throughout the *entire* design flow, from simulation, testbenches and formal verification, behavioural and structural modeling, to synthesis and the final gate-level netlist. There cannot be a single IR that fits all hardware needs, as this would require high-level simulation constructs without hardware equivalents, yet still be trivially synthesizable. However, the constructs needed to describe a netlist form a subset of those needed for synthesis, which are in turn a subset of those needed for simulation. As such, LLHD is a multi-level IR with three distinct levels or dialects, which cater to the corresponding parts of the tool flow. LLHD adheres to Static Single Assignment (SSA) form [5, 12], which lends itself exceptionally well to represent digital circuits, which are exactly that: signals with single, static driver assignments. LLHD borrows the basic IR syntax from LLVM but defines an IR for digital circuits, which must explicitly deal with the passing of time and, since digital circuits are inherently concurrent, must be able to describe concurrency. Together with *Moore*, a compiler frontend for HDLs, LLHD significantly reduces redundancy in the design flow and allows novel, open languages to thrive, as depicted in Figure 1.

Many IRs for hardware designs already exist. For instance, EDA tools have internal IRs, but these are highly tool-specific and mostly proprietary. In general, the vast majority of IRs

puts a narrow focus on circuit synthesis. To prevent redundancy and disagreement in compilers and gaps between IRs, we argue that one IR must cover the entire circuit design flow. To our knowledge this paper is the first to propose this solution. We make the following contributions:

- We define a multi-level IR that captures current HDLs in an SSA-based form compatible with modern, imperative compilers but with extensions and specializations crucial to represent digital hardware (§ 2).
- We show how existing industry-standard HDLs, such as SystemVerilog and VHDL, map to this IR (§ 3).
- We establish transformation passes to lower from Behavioural LLHD to hardware-near Structural LLHD (§ 4).
- We show that such a multi-level IR can improve the existing EDA tool flow, even without explicit support by commercial tools (§ 5).
- We provide evidence that the IR can capture complex designs such as entire CPU cores [31], that a minimal reference simulator models those designs identically to commercial simulators, and that an early optimized simulator runs up to 2.4× faster than a commercial simulator (§ 6).

Finally, we provide an open-source implementation of our IR, its reference simulator, and an accompanying HDL compiler.¹ The implementation acts as a vertically-integrated research prototype. This prototype is currently capable of capturing behavioural, structural, and netlist LLHD. Lowering from behavioural to structural LLHD is partially implemented in order to demonstrate the key transformations, but is not complete at the time of writing. Lowering from structural to netlist LLHD is the domain of hardware synthesizers and as such outside the scope of this work. The Moore compiler supports a subset of SystemVerilog which is large enough to represent a full CPU core [31] and covers a sufficient amount of non-synthesizable constructs of the language to support simple testbenches. Initial experimental work on VHDL support is underway. Our simulator implementation covers the vast majority of all three LLHD dialects, except a few instructions that were not instrumental to simulate the designs presented in this work.

2 The LLHD Intermediate Representation

The LLHD IR is designed as an SSA language [5, 12] which enables a very direct representation of the data flow in a digital circuit. The rationale is that modern digital circuits are essentially the same as an SSA data flow graph, where each logic gate corresponds to a node in the graph. LLHD has an in-memory representation, a human-readable representation, and a planned binary on-disk representation, and all three representations are equivalent. In this section, we define the core concepts of the LLHD language; the complete and

¹Project Website: <http://llhd.io/>

```

entity @acc_tb () -> () { c
  %zero0 = const i1 0
  %zero1 = const i32 0
  %clk = sig i1 %zero0
  %en = sig i1 %zero0
  %x = sig i32 %zero1 b
  %q = sig i32 %zero1
  inst @acc (i1$ %clk, i32$ %x, i1$ %en) -> (i32$ %q) k
  inst @acc_tb_initial (i32$ %q) -> (i1$ %clk, i32$ %x, i1$ %en)
} e

proc @acc_tb_initial (i32$ %q) -> (i1$ %clk, i32$ %x, i1$ %en) {
  entry: b
  %bit0 = const i1 0
  %bit1 = const i1 1
  %zero = const i32 0
  %one = const i32 1
  %many = const i32 1337
  %del1ns = const time 1ns
  %del2ns = const time 2ns
  %i = var i32 %zero
  drv i1$ %en, %bit1 after %del2ns
  br %loop
loop:
  %ip = ld i32* %i
  drv i32$ %x, %ip after %del2ns
  drv i1$ %clk, %bit1 after %del1ns n
  drv i1$ %clk, %bit0 after %del2ns
  wait %next for %del2ns f
next:
  %qp = prb i32$ %q m
  call void @acc_tb_check (i32 %ip, i32 %qp) i
  %in = add i32 %ip, %one
  st i32* %i, %in
  %cont = ult i32 %ip, %many
  br %cont, %end, %loop
end:
  halt g
}

func @acc_tb_check (i32 %i, i32 %q) void { a
  entry:
  %one = const i32 1
  %two = const i32 2
  %ip1 = add i32 %i, %one ; i+1
  %ixip1 = mul i32 %i, %ip1 ; i*(i+1)
  %qexp = div i32 %ixip1, %two ; i*(i+1)/2
  %eq = eq i32 %qexp, %q ; q == i*(i+1)/2
  call void @llhd.assert (i1 %eq) ; not yet implemented
  ret p
}

```

Figure 2. A testbench for an accumulator design as an illustrative example for LLHD code. See § 2 for a detailed description, Figure 3 for the corresponding SystemVerilog code, and Figure 5 for the implementation of @acc.

precise specification is part of the LLHD Language Reference Manual.²

2.1 Describing Digital Circuits

A hardware description requires a notion of passing time, must be able to represent concurrency, and provide a way to describe the structure and hierarchy of a circuit. This is due to digital circuits being inherently hierarchical, concurrent, and time-dependent. LLHD draws from the abstractions established in modern HDLs over the past decades, and distills them into fundamental orthogonal concepts. Figure 2 shows a sample LLHD source text describing a testbench for an accumulator circuit as an illustrative example. The language provides three modeling constructs:

²Language Reference Manual: <http://llhd.io/spec.html>

Functions capture a mapping from a set of input values to an output and allow for reuse of computation. They facilitate code reuse, recursion and program-defined mapping of a set of input values to a singular output value in the SSA graph, but they do not have a direct hardware equivalent. (See Figure 2a)

Processes are Turing-complete subprograms that describe how a circuit’s state and output reacts to a change in its input. This provides a *behavioural* circuit description. (See Figure 2b)

Entities build hierarchy by instantiating other processes or entities, which then operate concurrently. This provides a *structural* circuit description. Such instantiation translates into reuse by replication in the physical silicon, which is an essential contributor to our ability to manufacture designs with billions of transistors. (See Figure 2c)

Circuit designs written in HDLs generally model circuits behaviourally and structurally. Physical silicon itself is a purely structural arrangement of circuits and thus fully captured by a hierarchy of entities. Functions and processes are merely modeling tools to fully capture HDLs, including simulation and verification components which have no physical equivalent. To capture these levels of abstraction, LLHD is a three-level IR, as described in the following paragraph.

2.2 Multi-level Intermediate Representation

LLHD can capture *all* aspects of a digital design written in a higher-level HDL such as SystemVerilog or VHDL, including simulation, verification, and testing constructs. However, its structure also allows it to clearly capture parts relevant for synthesis, as well as represent the netlist that results from synthesis. This makes LLHD a *multi-level IR* with the following levels:

Behavioural LLHD aims at capturing circuit descriptions in higher-level HDLs as easily as possible. It allows for simulation constructs and test benches to be fully represented, including assertions, file I/O, or formal verification information as intrinsics.

Structural LLHD limits the description to the parts that describe the input to output relations of a design. This covers essentially everything that can be represented by an entity (see § 4 for a more technical description).

Netlist LLHD further limits the description to just entities and instructions to instantiate and connect sub-circuits. More specifically allowed are just the entity construct, as well as signal creation (*sig*), connection (*con*), delay (*del*), and sub-circuit instantiation (*inst*).

We observe that the constructs of Netlist LLHD are a strict subset of Structural LLHD, which in turn is a strict subset of Behavioural LLHD. Rather than defining three separate IRs, we thus propose one holistic IR to cover the entire process. As a design makes its way through the hardware design flow, the

simulation and design verification phase uses the full IR of Behavioural LLHD. Before synthesis, LLHD compiler passes lower the design to Structural LLHD. A synthesizer then lowers the design to a netlist by performing logic synthesis and mapping the design to a target silicon technology, which can be expressed in Netlist LLHD.

The remainder of this section provides a more detailed treatment of the constructs in LLHD. In the following two sections, we first discuss mapping a design from a HDL to LLHD (§ 3) and then the compiler passes to transform the design between the different LLHD levels (§ 4).

2.3 Modules, Names, and Types

A single LLHD source text is called a *module*. Modules consist of functions, processes, and entities. Multiple modules can be combined by a linker, which resolves references in one module against the definitions made in the other.

LLHD distinguishes between three types of names. Most importantly, to minimize naming conflicts in complex designs, only *global names*, such as @foo, are visible to other modules during linking. *Local names*, such as %bar, and anonymous names, such as %42, are only visible within the current module (for functions, processes, and entities) or the current unit (for values).

LLHD is a strongly-typed language, i.e., every value must have a type. LLHD supports a set of types typical for an imperative compiler: void (no value), iN (N-bit integers), T* (pointer to type T), [N × T] (array of N elements of type T), and {T1, T2, . . . } (structure with fields of type T1 etc.). LLHD defines the following hardware-specific types:

time represents a point in time. This allows to describe delays (e.g., through gates) and elapsed time (e.g., in simulation).

nN is an enumeration value that can take one of N distinct values. This allows to represent non-power-of-two values (e.g., the set of states in a state machine or the inputs to a multiplexer).

1N is a nine-valued logic value, defined in the IEEE 1164 standard [2]. This allows to model states that a physical signal wire may be in (drive strength, drive collision, floating gates, and unknown values).

T\$ is a signal carrying a value of type T. This represents a physical signal wire. The prb and drv instructions are used to read the current value of the signal and trigger a future change of the signal.

2.4 Units

The three main constructs of LLHD, functions, processes, and entities, are called *units*. As shown in Table 1, LLHD defines for each unit how instructions are executed (*execution paradigm*) and how time passes during the execution of the unit (*timing model*).

Table 1. Overview of the design units available in LLHD with their execution paradigm and timing model. See § 2.1 and § 2.4 for a detailed description.

Unit	Execution	Timing	Use
Function	control flow	immediate	user-def. SSA mapping
Process	control flow	timed	behavioural circ. desc.
Entity	data flow	timed	structural circ. desc.

The execution paradigm is either control or data flow. *Control flow* units consist of basic blocks, where execution follows a clear control flow path. Each basic block must have exactly one terminator instruction which transfers control to another basic block, to the caller, or halts completely. *Data flow* units consist only of a set of instructions which form a Data Flow Graph (DFG). Execution of instructions is implied by the propagation of value changes through the graph.

The timing model can be immediate or timed. *Immediate* units execute in zero time. They may not contain any instructions that suspend execution or manipulate signals. These units are ephemeral in the sense that their execution starts and terminates in between physical time steps. As such no immediate units coexist or persist across time steps. *Timed* units coexist and persist during the entire execution of the IR. They represent reactions to changes in signals and as such model the behaviour of a digital circuit. Such units may suspend execution or interact with signals by probing their value or scheduling state changes.

2.4.1 Functions. Functions represent a mapping from zero or more input arguments to zero or one return value. Functions are defined with the func keyword and are called from other units with the call instruction. For example, Figure 2a defines a function to assert that an input value %q matches the sum of all integers up to %i, and Figure 2d calls that function. Functions execute immediately, meaning that they cannot interact with signals or suspend execution.

2.4.2 Processes. Processes, in contrast, can interact with time. Similar to functions, they are executed in a control flow manner. Processes represent a circuit with zero or more inputs and outputs. The inputs and outputs must be of a signal type T\$; other types are not permitted. Figure 2b defines a process that generates the patterns to test an accumulator circuit, and Figure 2e instantiates that process. Upon initialization, control starts at the first basic block and proceeds as it would in a function. Processes may probe and drive the value of signals (see § 2.5.2), which are a process’ only means to communicate with other parts of the design. Additionally, processes may suspend execution for a period of time or until a signal change (wait, Figure 2f), or indefinitely (halt, Figure 2g). In contrast to functions, processes exist throughout the entire lifetime of the circuit and never return.

2.4.3 Entities. Entities describe a pure DFG and their execution is not governed by any control flow. Upon initialization, all instructions are executed once. At all subsequent points in time, instructions are re-executed if one of their inputs changes. This creates an implicit execution schedule for the instructions. Entities build structure and design hierarchies by allocating registers and signals via the `reg` and `sig` instructions, and instantiating other entities and processes via the `inst` instruction. For example, Figure 2c defines an entity which has four local signals (h), and instantiates the `@acc` design to be tested (k) and the `@acc_tb_initial` process to execute the test (e).

2.5 Instruction Set

LLHD’s simple instruction set captures the essence of hardware descriptions at a hardware-near level of abstraction. Nevertheless, LLHD preserves arithmetic operations, which are the main target of many optimizations in commercial hardware synthesizers. As a general rule, all instructions contain sufficient type annotations to determine the type of each operand and the result. We omit a detailed description of all instructions, especially those that are common in imperative compiler IRs such as LLVM, and focus on the hardware-specific concepts.

2.5.1 Hierarchy. Hierarchy and structure is described via the `inst` instruction, which is limited to entities. The instruction names a process or entity to be instantiated and associates each of its inputs and outputs with a signal (see Figure 2ek).

2.5.2 Signals. Signals are created with the `sig` instruction by providing the type of the value the signal carries, together with its initial value. The current value of a signal can be probed with the `prb` instruction, which takes a signal as its argument. A new value may be driven onto the signal with the `drv` instruction, which takes a target signal, value to be driven, drive delay, and an optional condition as arguments. These instructions are limited to processes and entities. For example in Figure 2, the `@acc_tb_initial` process (b) uses `prb` to “read” the value of its input `%q` (m), and `drv` to “write” a change to its outputs `%en`, `%clk`, and `%x` (n). The `@acc_tb` entity uses `sig` to define local signals (h) that connect `@acc` (k) and `@acc_tb_initial` (e).

2.5.3 Registers. State-holding storage elements such as registers and latches are created with the `reg` instruction, which is limited to entities. The `reg` instruction takes the stored value type and the initial value as first arguments. These are followed by a list of values, each with a trigger that describes when this value is stored. The trigger consists of the keyword `low`, `high`, `rise`, `fall`, and `both`, followed by a value. This allows for active-low/high, as well as rising, falling, and dual edge-triggered devices to be described. To model conditionally-enabled circuits, an optional `if gating`

clause can be used to discard the trigger if some condition is not met. For example, the optimized `@acc_ff` entity in Figure 5k further ahead uses `reg` to allocate a rising-edge triggered flip-flop to store the current accumulator state.

2.5.4 Data Flow. Data flow instructions, including constants, logic and arithmetic operations, shifts, and comparisons, are a significant part of the instructions in LLHD. For example in Figure 2, the `@acc_tb_check` function (a) uses the `add`, `mul`, `div`, and `eq` instructions to check the accumulator result. Selection between multiple values is performed with the `mux` instruction, which takes a sequence of values of the same type as arguments followed by a discriminator that chooses among them.

2.5.5 Bit-precise Insertion/Extraction. Bit-precise control of values is essential to describe digital designs. The `insf` and `extf` instructions allow to set (insert) or get (extract) the value of individual array elements or struct fields. The `inss` and `exts` instructions allow to set or get the value of a slice of array elements or bits of an integer.

2.5.6 Pointer/Signal Arithmetic. The extraction (`extf` and `exts`) and shift (`shl` and `shr`) instructions can also operate on pointers and signals. In this mode, these instructions return a new pointer or signal, which directly points at the extracted field or slice or to the shifted value. These operations are very useful when translating from HDLs, where slices or subsets of signals are frequently accessed or driven. In order to translate into hardware, these partially-accessed signals must be subdivided to the granularity of these accesses, in order to arrive at canonical drive and storage conditions for generated flip-flops or signal wires.

2.5.7 Control and Time Flow. *Control flow* instructions are the typical ones used in imperative compilers: conditional and unconditional branches, function calls, and returns. The example in Figure 2 uses `br` to implement a loop, `call` to execute the `@acc_tb_check` function (d), and `ret` to return from that function.

Time flow instructions allow processes to control the passing of time. The `wait` instruction suspends execution until one of its operand signals changes and optionally until a certain amount of time has passed. Execution then resumes at the basic block passed to the `wait` instruction as its first argument. The `halt` instruction suspends execution of the process forever. The example in Figure 2 uses `wait` to suspend execution for 2 ns in each loop iteration (f), and `halt` to stop once the loop terminates (g).

2.5.8 Memory. Stack and heap (or “dynamic”) memory are required to fully map HDLs. Stack memory holds local variables in functions (e.g., loop variables). Heap memory is required for Turing completeness, which is necessary to represent *all* simulation and verification code in today’s HDLs. For example, SystemVerilog provides builtin dynamic queues

and vectors, sparse associative arrays, and a mechanism to call into native code loaded from an object file, or vice versa. These features require heap allocation and deallocation of memory, and are heavily used in more advanced testbenches and verification code. Values in allocated memory are loaded and stored with the `ld` and `st` instructions. Stack allocation is implemented by the `var` instruction, and heap allocation and deallocation by `alloc` and `free`, respectively.

LLHD code intended for hardware synthesis is expected to require a bounded amount of stack and heap memory known at compile-time, usually none at all. Bounded heap allocations are guaranteed to be promotable to stack allocations, and bounded stack allocations guarantee that the total amount of required memory is known at compile time. An algorithm similar to LLVM’s memory-to-register promotion allows LLHD to promote memory instructions to values and phi nodes. Lowering to Structural LLHD requires *all* stack and heap memory instructions to be promoted in this way, as a design is otherwise not implementable in hardware and can be rejected. Verification code, which is expected to run in an interpreter, does not have to meet these requirements.

2.5.9 Intrinsic and Debugging. Additional functionality beyond what is provided as explicit instructions may be represented as intrinsics. An intrinsic is a call to a predefined function prefixed with “`llhd.`” (e.g. Figure 2p). This allows for concepts such as `stdin/stdout`, file I/O, or assertions to be preserved when transporting from HDLs into LLHD. We envision debug information such as HDL source locations and naming to be attached to instructions and units via metadata nodes akin to LLVM. Furthermore, a special `obs` instruction could be used to describe an observation point for a signal as the user has written it in the original HDL. This allows a design to remain debuggable and recognizable to a user despite aggressive synthesis transformation — a feature which is currently lacking in commercial synthesizers and place-and-route software.

3 Mapping HDLs to LLHD

LLHD’s primary goal is to capture designs described in HDLs such as SystemVerilog or VHDL. This explicitly includes simulation and verification constructs and not just the synthesizable subset of a language. As part of the LLHD project we are developing the *Moore* compiler, which maps SystemVerilog and VHDL to LLHD. This is comparable to the interaction between Clang and LLVM. Moore’s goal is to map as much of SystemVerilog and VHDL to LLHD as possible, providing a reference implementation for both HDLs and a platform for future efforts in hardware synthesis, verification, and simulation without the need to reinvent the HDL compiler wheel. This section explores how common constructs in these languages map to LLHD, based on our accumulator and testbench running example, the SystemVerilog source

```

module acc_tb;
  a bit clk, en;
  bit [31:0] x, q;
  acc i_dut (.*);

  initial begin
    automatic bit [31:0] i = 0;
    en <= #2ns 1;
    do begin
      x <= #2ns i;
      clk <= #1ns 1;
      clk <= #2ns 0;
      #2ns;
      check(i, q);
    end while (i++ < 1337);
  end

  function check(bit [31:0] i, bit [31:0] q);
    assert(q == i*(i+1)/2);
  endfunction
endmodule

module acc (input clk, input [31:0] x, input en, output [31:0] q);
  b bit [31:0] d, q;
  always_ff @(posedge clk) q <= #1ns d; d
  always_comb begin c
    d <= #2ns q;
    if (en) d <= #2ns q+x;
  end
endmodule

```

Figure 3. SystemVerilog source code for the testbench and accumulator LLHD code in Figure 2 and Figure 5, as an instructive example as to how HDL concepts map to LLHD. See § 3 for a detailed description.

code of which is shown in Figure 3. The resulting LLHD code is shown in Figure 2 (testbench) and Figure 5 (accumulator).

3.1 Hierarchy

The fundamental elements of reuse and hierarchy are “modules” in SystemVerilog and “entities” in VHDL. Both describe a circuit as a list of input and output ports and a body of signals, sub-circuits, and processes. Thus, they trivially map to LLHD entities, for example Figure 3a to Figure 2c, or Figure 3b to Figure 5m.

3.2 Processes

Processes are the main circuit modeling tool in most HDLs. SystemVerilog provides the `always`, `always_ff`, `always_latch`, `always_comb`, `initial`, and `final` constructs, while VHDL has a general-purpose process concept. Using these constructs, a circuit can be described in a *behavioural* fashion by providing an imperative subprogram that maps a change in input signals to a change in output signals: essentially, the body of a process is re-executed whenever one of its input signals changes. LLHD’s processes are designed to capture these constructs through an almost verbatim translation from an HDL process. Processes allow for a very high-level and general description of circuits. However, it is common practice to follow a strict modeling style in HDLs to ensure synthesizers infer the proper hardware, which we briefly describe in the following.

3.2.1 Combinational Processes. Combinational processes describe a purely functional mapping from input signals to output signals, without any state-keeping elements such as flip-flops as side effect. Synthesizers can readily map such processes to logic gates. A process is combinational if there are no control flow paths that leave any of its output signals unassigned. Combinational processes can be mapped to a pure data flow graph of logic gates. Consider the `always_comb` process in Figure 3c, which directly maps signals `q` and `x` to an output value `d`. This translates into the `@acc_comb` LLHD process in Figure 5n.

3.2.2 Sequential Processes. Sequential processes describe state-keeping elements such as flip-flops and latches. A process is sequential if at least some of its output signals are only assigned under certain conditions. Synthesizers detect these kinds of processes, usually requiring the designer to adhere to a very strict pattern, and map them to the corresponding storage gate. Consider the `always_ff` process in Figure 3d, which maps to the `@acc_ff` LLHD process in Figure 5p. LLHD can capture register descriptions in the behavioral form (see Figure 5p) but also provides an explicit `reg` instruction (see Figure 5k) to canonically represent registers, which is inferred by lowering passes discussed in § 4.

3.2.3 Mixed Processes. SystemVerilog and VHDL allow designers to mix combinational and sequential styles in one process, but support for this is limited even in commercial synthesizers. The desequenzialization pass (§ 4.6) can help split mixed processes into combinational and sequential parts for higher compatibility with many synthesizers.

3.3 Generate Statements and Parameters

Many HDLs provide *generate statements* and *parameters* to allow for parametrized generation of hardware. LLHD does *not* provide such constructs, but rather expects these statements to be unrolled already by the compiler frontend (e.g., Moore). The rationale is that HDL designs parametrized over constants and types lead to significant changes in the generated hardware that go beyond mere type or constant substitution. For example, a parameter might cause the generate statements in a design to unroll to a completely different circuit. Capturing this flexibility in LLHD would require a significant meta-programming layer to be added, which in our opinion is best left to a higher-level language or IR.

3.4 Verification

Modern HDLs feature constructs to verify hardware designs. SystemVerilog, for example, provides the `assert`, `assume`, and `require` constructs. We propose to map these to LLHD intrinsics such as `llhd.assert`. A simulator may then choose to emit error messages when the condition passed to such an intrinsic is false. A formal verification tool, on the other hand, can extract these intrinsics and set up a satisfiability

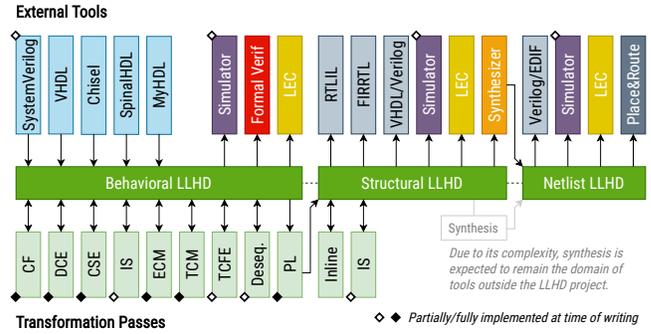


Figure 4. Optimization and transformation passes on the different IR levels of LLHD. See § 4 for a detailed description.

problem or perform bounded model checking [10]. Higher-level constructs to describe Linear Temporal Logic (LTL) and Computational Tree Logic (CTL*) properties [13] shall be mapped to intrinsics as well, such that a formal verification tool can recover them from the IR. An interesting side-effect of preserving these verification constructs as intrinsics is that Field-Programmable Gate Array (FPGA) mappings of an LLHD design may choose to implement the assertions in hardware, to perform run-time checks of a circuit. These features are not yet implemented in our research prototype, and an LLHD-based verification tool is yet to be written.

4 Lowering to Structural LLHD

One of the key contributions of LLHD is a framework to translate the high-level behavioural circuit descriptions of an HDL into a lower-level structural description that can be readily synthesized. Existing commercial and open-source synthesizers all have redundant proprietary implementations of this procedure; see Figure 1. With LLHD and language frontends such as Moore, this translation can be performed as a lowering pass on the IR directly, rather than individually by each synthesizer. The general objective of this lowering is to replace branches with branch-free code. In particular, it consists of the following high-level steps:

- Reduce complexity of operations (§ 4.1)
- Move arithmetic out of basic blocks (ECM, § 4.2)
- Move drives out of basic blocks (TCM, § 4.3)
- Replace `phi` and control flow with `mux` (TCFE, § 4.4)
- Replace trivial processes with entities (PL, § 4.5)
- Identify flip-flops and latches (Deseq., § 4.6)

Consider the accumulator design in Figure 5, which is a typical example of Behavioural LLHD as it is generated from a SystemVerilog source. Let us step through the transformations in Figure 4 to lower this example to Structural LLHD.

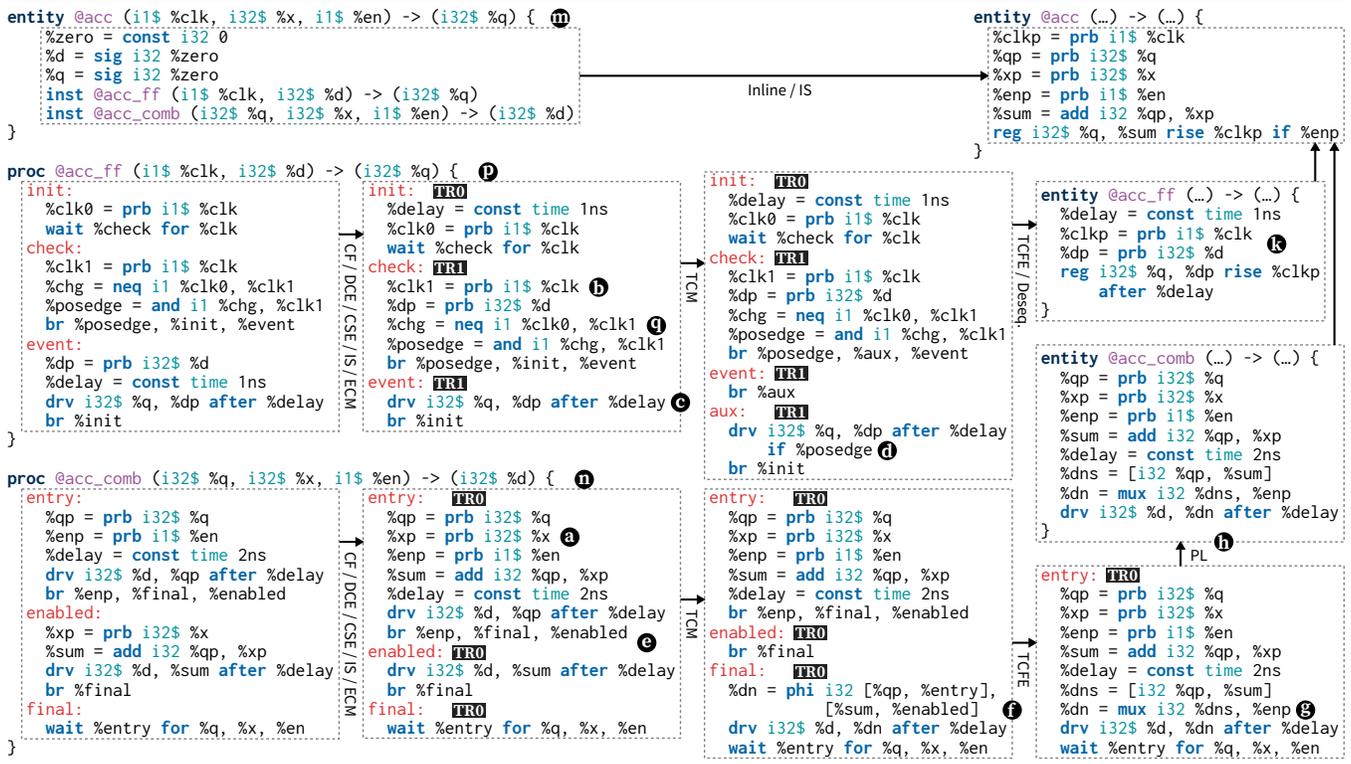


Figure 5. An end-to-end example of how a simple accumulator design is lowered from Behavioural LLHD (left) to Structural LLHD (right). See § 4 for a detailed description of the individual steps. The processes @acc_ff and @acc_comb are lowered to entities through various transformations, and are eventually inlined into the @acc entity.

4.1 Basic Transformations

In a first step, we apply basic transformations such as Constant Folding (CF), Dead Code Elimination (DCE), and Common Subexpression Elimination (CSE), which are equivalent to their LLVM counterparts. Furthermore, Instruction Simplification (IS) is used as a peephole optimization to reduce short instruction sequences to a simpler form, similar to LLVM’s instruction combining. To facilitate later transformations, all function calls are inlined and loops are unrolled at this point. Where this is not possible, the process is rejected.

4.2 Early Code Motion (ECM)

ECM moves instructions “up” in the Control Flow Graph (CFG), to facilitate later control flow elimination. This is similar to and subsumes Loop-Invariant Code Motion (LICM) in LLVM in that code is hoisted into predecessor blocks, but ECM does this in an eager fashion. The underlying motif of lowering to Structural LLHD is to eliminate control flow, since that does not have an equivalent in hardware. An essential step towards this is to *eagerly* move instructions into predecessor blocks as far as possible. As shown in Figure 5a, this moves all constants into the entry block, and arithmetic instructions to the earliest point where all operands are available. Special care is required for prb as

in Figure 5b, which *must not* be moved across wait, as that would imply a semantic change.

4.3 Temporal Code Motion (TCM)

The wait instructions naturally subdivide a process into different Temporal Regions (TRs), i.e. sections of code that execute during a fixed point in physical time. A key step towards eliminating control flow is to move drv instructions into a single exiting block for their respective TR. The condition under which the control flow reaches a drv instruction before the move is added as an additional operand to the instruction. Let us elaborate in more detail by first introducing the concept of TRs.

4.3.1 Temporal Region (TR). The capability of wait to suspend execution mid-process until a later point in time calls for novel techniques to reason about the synchronicity of instructions. More specifically, we would like to know if two instructions execute at the same instant of physical time *under all circumstances*. Consider the %clk signal in Figure 5q as an illustrative example: when we reach the neq instruction, we would like to be able to reason that %clk0 is an “old” sampling of %clk from before the wait, and that %clk1 reflects the current state of %clk. Each basic block in

LLHD has an associated TR. Multiple blocks may belong to the same TR. The set of blocks in the same TR represents the bounds within which `prb` and `drv` instructions may be rearranged without changing the process behaviour. As an intuition, TRs are assigned to individual blocks based on the following rules:

1. If any predecessor has a `wait` terminator, or this is the entry block, generate a *new* TR.
2. If all predecessors have the *same* TR, inherit that TR.
3. If they have *distinct* TRs, generate a *new* TR.

Note that as a result of rule 3, there is one unique entry block for each TR where control transfers to from other TRs. Figure 5ab shows the temporal regions assigned to the individual blocks. Note that the flip-flop process `@acc_ff` has two TRs, whereas the combinational process `@acc_comb` has just one.

4.3.2 Single Exiting Block per TR. We would like each TR to have a single exiting block. This is essential to have a single point in the CFG to move `drvs` to such that they are always executed in this TR. If TR A has multiple control flow arcs leading to TR B, an additional intermediate block is inserted in order to have a single arc from TR A to B. This is always possible since as a result of rule 3 in § 4.3.1, all branches to TR B target the same unique entry block. In Figure 5b for example, blocks `check` and `event` both branch to `init`, which is in a different TR. In this case an auxiliary block is created as part of TCM, where `check` and `event` branch to.

4.3.3 Moving Drive Instructions. As the main part of TCM, `drv` instructions are moved into the single exiting block of their TR. We first find the closest common dominator of the exiting block and the instruction. If no such dominator exists, the instruction is left untouched, which later causes the lowering to reject the process. As a second step, we find the sequence of branch decisions that cause control to flow from the dominator to the `drv` instruction. This essentially builds a chain of `ands` with the branch conditions “along the way”, or their inverse, as operands. As the final third step, the `drv` instruction is moved into the exiting block, and the expression found in the second step is set as the `drv`’s optional condition operand. In Figure 5c, control flow only reaches the `drv` if the `%posedge` branch argument is true. Consequently, `%posedge` is added as drive condition in Figure 5d. In Figure 5e, control flow always reaches the `drvs`, which are consequently moved into the existing single exiting block `final` without adding a condition operand, as in Figure 5f. Since both `drvs` target the same signal, they are coalesced into one instruction, and selection of the driven value is factored out into a `phi` instruction.

4.4 Total Control Flow Elimination (TCFE)

The goal now is to replace control flow with data flow, branches with multiplexers. The previous transformations leave many empty blocks behind. TCFE eliminates these blocks such that only one block remains per TR. This is the case in Figure 5df, where only the `init`, `check`, and `entry` blocks remain. Furthermore, all `phi` instructions are replaced with `mux` instructions, as shown in Figure 5g. The selector for the `mux` instruction is found in the same way as the `drv` condition in § 4.3.3. As a result, combinational processes (§ 3.2.1) now consist of a single block and TR, and sequential processes (§ 3.2.2) of two blocks and TRs. Processes for which neither holds are rejected by the lowering.

4.5 Process Lowering (PL)

At this point, processes with a single block and a `wait` terminator of the correct form are lowered to an entity. This is done by removing the `wait` and moving all other instructions to an entity with the same signature. In order for this to be equivalent, the `wait` must be sensitive to all `prb`’d signals. See Figure 5h for an example where this is the case.

4.6 Desequentialization (Deseq.)

For the remaining processes we would like to identify if they describe a sequential circuit such as a flip-flop or latch. HDLs expect these to be inferred from signals that are only driven under certain conditions, e.g., if a clock signal changed or a gate signal is high. The TCM pass canonicalizes processes into a form which makes this rather straightforward. We only consider processes with two basic blocks and TRs, which covers all relevant practical HDL inputs. In a first step, we canonicalize the condition operand of each `drv` into its Disjunctive Normal Form (DNF). The DNF exists for all boolean expressions, is trivially extended to `eq` and `neq`, and can retain all non-canonicalizable instructions as opaque terms. Each separate disjunctive term of the DNF identifies a separate trigger for the flip-flop or latch. The `drv` in Figure 5d, for example, has the canonical condition $\neg\%clk0 \wedge \%clk1$. In a second step, we identify which terms of the condition are sampled *before* the `wait`, and which are sampled *after*. This is done based on the TR of the corresponding `prb`. The TR of the `wait` is considered the “past”, and the TR of the `drv` is considered the “present”. In a third step, we isolate terms T which are sampled both in the past (T_0) and the present (T_1), and pattern match as follows:

- $\neg T_0 \wedge T_1$ is a *rising* edge on T
- $T_0 \wedge \neg T_1$ is a *falling* edge on T
- $(\neg T_0 \wedge T_1) \vee (T_0 \wedge \neg T_1)$ are *either* edges on T

All other terms are moved into the set of “trigger conditions”. At this point, a separate entity is created which will hold the identified sequential elements. In a final step, all `drvs`, for which the above trigger identification was successful, are mapped to an equivalent `reg` instruction. Each

trigger is added to the reg separately, together with the corresponding set of trigger conditions: edge terms are mapped to corresponding rise, fall, or both edge triggers, and all remaining terms to high or low level triggers. Furthermore, the entire DFG of the drv operands, namely driven signal, value, delay, and condition, is added to the entity. See Figure 5k. This procedure identifies and isolates edge- and level-triggered sequential elements into an entity. The remaining process is either empty and removed, lowered by PL, or rejected.

4.7 Synthesizability

The class of “synthesizable” hardware descriptions and subsets of HDLs is defined rather loosely. In practice, hardware descriptions adhere to sufficiently strict coding guidelines that enable the above transformations to occur.

5 Toolflow Integration

Languages such as SystemVerilog are too high-level to reliably transport digital circuits through the design flow. This is mainly due to the difficulty to consistently implement these languages, as described in § 1. For example, designs are simulated and verified in their HDL description, by a simulator or verification tool. A Logical Equivalence Checker (LEC) is then used to verify that the design has been mapped correctly from HDL to an equivalent netlist. Not only does this require the LEC to fully implement the SystemVerilog standard as well, it also only verifies that the synthesizer’s interpretation of the HDL matches the LEC’s. However, it does not verify whether the simulation and verification tool’s interpretation matches the LEC’s. This is potentially disastrous given the complexity of languages such as SystemVerilog.

LLHD provides a design representation that is much simpler to implement consistently. Simulating and verifying the LLHD mapping of a circuit rather than its original HDL means converging to a single simple representation early in the design flow. Synthesis tools and LECs running on this reference LLHD mapping then ensure correct translation of the circuit into a netlist. LLHD’s simplicity offers a much smaller “surface for implementation errors”.

Many commercial tools already use a proprietary IR internally. These are generally accessible to the user, such that a Structural LLHD description can be mapped directly to such a tool’s IR. Where this is not possible, the description may be mapped to a simple, structural Verilog equivalent to be ingested by the tool. Ideally, vendors eventually support direct input of LLHD, but this is not required.

We conclude that LLHD fits very well into existing commercial tool flows. Its simplicity makes it a prime candidate to harden the HDL-to-netlist verification chain. Furthermore, its expressivity allows it to subsume other IRs, offering a platform to transport designs between FIRRTL [14], RTLIL [30], CoreIR [19], and others.

Table 2. Evaluation of the simulation performance of LLHD. We compare the LLHD reference interpreter to a JIT-accelerated LLHD simulator and to a commercial simulator. The former two operate on unoptimized LLHD code as emitted by the Moore frontend with the `-O0` flag. We list the lines of code “LoC” and executed clock “cycles” to provide an indication for the design and simulation complexity. Traces match between the two simulators for all designs. See Section 6.1 for a detailed description.

Design	LoC	Cycles	Sim. Time [s]		
			Int. ¹	JIT ²	Comm. ³
Gray Enc./Dec.	17	12.6 M	9740	6.33	6.07
FIR Filter	20	5 M	4430	10.35	14.60
LFSR	30	10 M	2350	14.53	14.10
Leading Zero C.	52	1 M	11000	3.23	7.84
FIFO Queue	102	1 M	1370	5.92	5.55
CDC (Gray)	108	1 M	1380	8.72	6.45
CDC (strobe)	122	3.5 M	1570	9.39	6.11
RR Arbiter	159	5 M	49400	10.92	25.54
Stream Delayer	219	2.5 M	477	4.28	4.99
RISC-V Core	3479	1 M	24000	23.44	4.47

¹ LLHD reference interpreter (*LLHD-Sim*), extrapolated;

² JIT-accelerated simulator (*LLHD-Blaze*);

³ Commercial HDL simulator

Moreover, LLHD significantly lowers the hurdle for innovation in the digital hardware design ecosystem. For instance, a new HDL only needs to be lowered to LLHD to become supported by all existing toolchains, High-level Synthesis (HLS) compilers can generate LLHD as output, simulators only have to parse the simple LLHD rather than complex HDLs, and synthesizers can take Structural LLHD as their input.

6 Evaluation

We have implemented the Moore compiler and LLHD as a vertical research prototype. Moore covers enough of the SystemVerilog standard to show merit on a non-trivial set of open-source hardware designs: specifically, we consider the designs listed in Table 2, which range from simple arithmetic primitives, over First-In First-Out (FIFO) queues, Clock Domain Crossings (CDCs), and data flow blocks, up to a full RISC-V processor core [29]. The lines of SystemVerilog code (“LoC”) in Table 2 provides a rough indication of the respective hardware complexity. The designs are mapped from SystemVerilog to Behavioural LLHD with the Moore compiler, without any optimizations enabled.

Table 3. Comparison against other hardware-targeted intermediate representations. Most other IRs are geared towards synthesis and the resulting netlists. See Section 6.2 for a detailed description.

IR	No. of Levels	Turing-Complete	Verification (e.g., assertions)	9-Valued Logic (IEEE 1164 [2])	4-Valued Logic (IEEE 1364 [3])	Behavioral	Structural	Netlist
LLHD [us]	3	✓	✓	✓	✓	✓	✓	✓
FIRRTL [14]	3 [†]	-	-	-	-	-	✓	✓
CoreIR [19]	1	-	✓	-	-	-	✓	-
μIR [24]	1	-	-	-	-	-	✓	-
RTLIL [30]	1	-	-	-	✓	✓	✓	-
LNAST [28]	1	-	-	-	-	✓	-	-
LGraph [27]	1	-	-	-	-	-	✓	✓
netlistDB [6]	1	-	-	-	-	-	✓	✓

[†] Mentioned conceptually but not defined precisely

6.1 Circuit Simulation

In a first evaluation, we simulate the designs in Table 2 mapped to LLHD using *LLHD-Sim*, the LLHD reference interpreter, and *LLHD-Blaze*, a simulator leveraging just-in-time (JIT) code generation. Both simulators support all levels of the LLHD IR. *LLHD-Sim* is deliberately designed to be the simplest possible simulator of the LLHD instruction set, rather than the fastest. *LLHD-Blaze* is a first implementation of JIT compilation to show the potential of massively accelerated simulation. For each of the designs, we compare the execution time of a commercial SystemVerilog simulator (“Comm.”) with our two approaches in Table 2. Simulations were executed on an Intel Core i7-4770 CPU running at 3.4 GHz. Without JIT compilation, *LLHD-Sim* is slower than its commercial counterpart, which trades initial optimization overhead at startup for increased simulation performance. Most importantly, however, the LLHD simulation trace is equal to the one generated by the commercial simulator for all designs. That is, even with a simple prototype implementation of Moore and LLHD, we can fully and correctly represent a RISC-V processor in Behavioural LLHD.

JIT compilation provides an opportunity to massively accelerate LLHD simulation. For this, we map LLHD to LLVM IR and then use LLVM to optimize for execution on the simulation machine. This makes *LLHD-Blaze* (“JIT”) competitive with commercial simulators, even though the former is a prototype whereas the latter have been optimized for decades. In some cases, *LLHD-Blaze* is already up to 2.4× faster than commercial simulators.

These benefits manifest themselves even on entirely *un-optimized* LLHD code as it is emitted by Moore with the

-O0 flag, comparable to Clang’s equivalent. We expect the discrepancies between LLHD-Blaze and the commercial simulator to disappear as we add optimizations to the simulator in the future. This will especially affect complex benchmarks such as the RISC-V core, where the lack of code optimization currently incurs significant overheads. Note that the lines of SystemVerilog code and simulation cycles do not always fully portray the complexity of a design: some SystemVerilog constructs can produce significantly more LLHD code than others, leading to significant differences in simulation time.

Much of the complexity and engineering effort of our work lies in Moore, the HDL compiler frontend, which tackles the extensive SystemVerilog standard. Further extensions of Moore to implement additional SystemVerilog constructs can vastly increase the scope of designs that can be mapped to LLHD. Once a design is in LLHD, simulating it correctly is trivial.

6.2 Comparison with other Hardware IRs

LLHD is not the first IR targeted at hardware design. However, to our knowledge it is the first multi-level IR capable of capturing a design throughout the entire hardware design flow, from simulation to synthesized netlist. Table 3 compares the part of the design flow covered by LLHD and other IRs. We observe that almost all other IRs support structural descriptions of circuits and many IRs support representing a synthesized netlist. This is due to the fact that most of them were designed as synthesis IRs to capture and apply transformations before and after synthesis. A notable example is FIRRTL, which acts as an interface between the Chisel [7] frontend and subsequent synthesis. As the only other IR, FIRRTL defines multiple levels of abstraction together with transformations to lower designs from higher to lower levels. μIR is designed to capture entire accelerator architectures as structural description and caters more to an HLS flow. RTLIL and LNAST both support the behavioural description of circuits. RTLIL is geared towards simplifying Verilog input to make it amenable for synthesis. LNAST represents behavioural HDL code as language-agnostic Abstract Syntax Tree (AST) and offers transformation passes. As the only other IR, CoreIR focuses on adding formal verification support to HDLs. No other IR is Turing-complete, which is essential to represent arbitrary reference models and verification constructs. FIRRTL provides limited support for testbench constructs in the form of message logging and clocking control.

LLHD covers all stages of the flow: *Behavioural LLHD* captures testbench, verification, and behavioural circuit descriptions, to facilitate interfacing with HDLs; *Structural LLHD* captures circuits structurally, to interface with synthesis and low-level circuit analysis and transformation tools; and *Netlist LLHD* represents the individual gates of a finalized circuit. Hence LLHD is the only IR capable of representing the full

Table 4. Size efficiency of the human-readable text representation, an estimate of a prospective bitcode, and the in-memory data structures of LLHD. See § 6.3 for details.

Design	SV	LLHD [kB]		
	[kB]	Text	Bitcode ¹	In-Mem.
Gray Enc./Dec.	3.0	11.9	3.6	41.8
FIR Filter	1.2	12.9	3.8	46.7
LFSR	2.4	4.9	1.8	18.4
Leading Zero C.	4.6	97.4	21.8	309.1
FIFO Queue	7.1	20.5	6.5	71.3
CDC (Gray)	8.5	38.1	12.5	129.5
CDC (strobe)	6.3	17.7	6.0	63.3
RR Arbiter	11.2	39.4	9.9	129.8
Stream Delayer	12.2	18.7	6.7	68.6
RISC-V Core	174.4	349.4	93.6	1096.3

¹ estimated

semantics of the SystemVerilog/VHDL input language for the full hardware design flow.

6.3 Size Efficiency

An important aspect of a hardware IR is how efficiently it captures a design in terms of memory used, both in on-disk and in-memory representations. Table 4 shows the size in kB occupied by the previously introduced designs.

For on-disk representations, we consider the SystemVerilog HDL code as a baseline (first numeric column). First, we observe that the *unoptimized* LLHD text (second column) emitted by the Moore compiler using the `-O0` flag (equivalent to the corresponding Clang/GCC flag) is significantly larger than SystemVerilog. This is due to the fact that Moore tries to keep code generation as simple as possible and emits numerous redundant operations. Furthermore, many operations which are implicit in SystemVerilog, for example, expression type casts and value truncation/extension, require explicit annotation in LLHD. Future optimizations can reduce redundant operations and produce more compact instructions, which we expect to be significantly smaller. Utilizing a binary “bitcode” representation (third column) instead can greatly reduce the on-disk size of the design, such that compiled HDL input represented as LLHD unoptimized bitcode is already comparable in size to the input source code. The bitcode itself is not yet implemented. Sizes are estimated based on a strategy similar to LLVM’s bitcode, considering techniques such as run-length encoding for numbers, interning of strings and types, compact encodings for frequently-used primitive types and value references. This makes LLHD a viable format to transport a design into the initial stages of the design flow, such as testbenches and simulation, formal verification, and synthesis.

The in-memory size of an IR (last column of Table 4) is even more important for transformation and optimization passes. While there is no baseline for this, we observe that even a full RISC-V core only requires 1 MB of memory. As the in-memory complexity scales linearly with the complexity of the design, we argue that representing even entire System on Chip (SoC) designs with hundreds of CPU cores will be feasible with tens of gigabytes of memory, which is fully viable for today’s workstations.

7 Related Work

Intermediate representations are an established and very successful concept both for representing imperative programs and the design of hardware circuits. This work has been in development over the past three years and predates efforts such as MLIR [17], which aims at providing a unifying framework for defining compiler IRs. The proposed concepts can likely be expressed in MLIR.

7.1 Compiler Intermediate Representations

Since the early days of compilers and program optimizations it has been clear that in the process of translating a computer program to machine code the program passes a variety of different representations [22]. Machine-independent internal program representations have been discussed as early as 1979 in the PQCC project [9, 23] which introduced a tree-structured common optimization language (TCOL) and showed its use in an optimizing Ada compiler. Intermediate representations are standard in most compilers today. Functional programming languages often use continuation-passing style representations [25]. For imperative programming languages, SSA-based intermediate representations [5, 12] have shown most successful, and are used in large compiler infrastructures such as LLVM [16], GCC [20], but also research compilers such as Cetus [15]. While most SSA-based compilers today use the concept of imperative branching transitioning between sequences of instruction blocks, Click and Cooper [11] removed this restriction by introducing the sea-of-nodes concept of graph-based intermediate languages where all freely floating instructions are only constrained by explicit control and data dependences. This concept is used in research compilers such as libfirm [8] or Google’s TurboFan JavaScript compiler.³ While the above approaches have shown the benefit of, especially SSA-based, intermediate representations the above concepts all aim for the generation of executable software, but do not target hardware design. Graph-based IRs certainly serve as inspiration to our Netlist LLHD, but existing compilers use graph-based IRs mostly for software compilation. Nevertheless, there are first efforts to define intermediate representations for hardware designs, which we will discuss in detail in the following sections.

³<http://v8.dev>

7.2 FIRRTL

FIRRTL [14] is the IR most closely related to LLHD known to us. FIRRTL acts as an abstraction layer between the Chisel [7] hardware generation framework and subsequent transformation passes and synthesis. FIRRTL’s semantics are closely coupled to those of Chisel and focuses mainly on the synthesis portion of the design flow. Notable exceptions are support for certain testbench constructs (see Section 6.2). We identify the following four fundamental differences between LLHD and FIRRTL:

(1) FIRRTL’s fundamental data structure is an AST [18]. Nodes may be assigned multiple times and *at different granularities*, making identification of a value’s single producer difficult. In contrast, algorithms operating on SSA forms prevail in modern compilers, and most research on transformations focuses on SSA. While low FIRRTL also requires SSA form, algorithms *requiring* this form are precluded from operating on all but the lowest level of abstraction.

(2) FIRRTL cannot represent testbench and simulation constructs such as precise delays, events, and queues and dynamic arrays, as well as arbitrary programs that stimulate and check a circuit. These constructs are essential to fully represent industry-standard HDLs such as SystemVerilog and to transport designs written in such languages through the full digital design flow.

(3) FIRRTL does not represent four- or nine-valued logic as defined by IEEE 1364 and IEEE 1164, respectively. These types of logic model the additional states of a physical signal wire (beyond the fundamental 0 and 1) and captures concepts such as driving strength and impedance. Industry-standard hardware designs in SystemVerilog and VHDL rely on this modeling capability to describe bidirectional signals, propagate unknown values, identify driving conflicts, and describe optimization opportunities during logic synthesis.

(4) FIRRTL has the concept of “three forms” but does not clearly define their boundaries and to which parts of the design flow they apply [14]. It merely states that low FIRRTL maps directly to Verilog constructs with straightforward semantics.

Overall we observe that LLHD is a *superset* of FIRRTL. Since LLHD provides a Turing-complete modeling construct for a circuit, and FIRRTL does not, any FIRRTL representation can be translated into an equivalent LLHD representation, but not vice-versa. And by the same reasoning, modern HDLs such as SystemVerilog and VHDL cannot be fully mapped to FIRRTL.

7.3 Verification IRs

CoreIR [19] focuses on verification. It is designed to interact with higher-level functional descriptions of a circuit, such as Halide [21] or Verilog (via Yosys [30]), and represent these within a formal verification infrastructure. Additional steps

allow designs to be mapped to Coarse-Grained Reconfigurable Arrays (CGRAs). CoreIR is geared specifically towards verification and deployment on FPGA-like devices, and as such does not cater to a full chip design flow.

7.4 Synthesis IRs

Many IRs beside FIRRTL are engineered to interact with hardware synthesizers. LNASt [28] targets the representation of the synthesizable parts of an HDL circuit description in a language-agnostic AST. RTLIL [30] is part of the “Yosys” open-source tool suite and focuses mainly on logic synthesis. It cannot represent higher-level constructs such as aggregate types or conditional assignment. μ IR [24] is geared towards HLS design flows and tries to capture entire accelerator architectures. These IRs are very specifically crafted to transport designs into a synthesizer and focus solely on this part of the flow.

7.5 Netlist IRs

IRs exist that aim at capturing the gate-level netlist of a circuit. LGraph [27] is an open-source graph representation of such a circuit, together with additional aspects of the physical design flow such as cell libraries, timing and power characteristics, and placement information. NetlistDB [6] follows a similar goal. These IRs cater only to the very end of the hardware design flow.

8 Conclusion

We showed with Moore and LLHD that a small and concise multi-level intermediate representation can represent the complex semantics of real-world VHDL and SystemVerilog circuits throughout the complete hardware design process. Thanks to a novel three-level IR, we present a hardware design language that is effective from simulation, over formal verification, to synthesized netlists. We demonstrate the effectiveness of our design by outperforming commercial simulators on a variety of simulation tasks. We expect that our concise and well-defined hardware design language, crafted in the spirit of the most successful SSA-based compiler IRs, both minimal and still expressive enough for real-world use cases, will provide the foundation for an open hardware design stack that follows the impressive evolution of modern compilers in recent years.

References

- [1] IEEE 1076-2008. 2008. *VHDL Language Reference Manual*.
- [2] IEEE 1164. 1993. *Standard Multivalued Logic System for VHDL Model Interoperability*.
- [3] IEEE 1364. 2005. *Standard for Verilog Hardware Description Language*.
- [4] IEEE 1800-2017. 2017. *SystemVerilog Unified Hardware Design, Specification, and Verification Language*.
- [5] Bowen Alpern, Mark N Wegman, and F Kenneth Zadeck. 1988. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1–11.

- [6] Anon. 2019. netlistDB: Intermediate format for digital hardware representation with graph database API. <https://github.com/HardwareIR/netlistDB>.
- [7] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimantas Avizienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*. IEEE, 1212–1221.
- [8] Matthias Braun, Sebastian Buchwald, and Andreas Zwinkau. 2011. *Firm—a graph-based intermediate representation*. KIT, Fakultät für Informatik.
- [9] Roderic G.G. Cattell, Joseph M. Newcomer, and Bruce W. Leverett. 1979. Code Generation in a Machine-independent Compiler. In *Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction* (Denver, Colorado, USA) (*SIGPLAN '79*). ACM, New York, NY, USA, 65–75. <https://doi.org/10.1145/800229.806955>
- [10] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 2001. Bounded model checking using satisfiability solving. *Formal methods in system design* 19, 1 (2001), 7–34.
- [11] Cliff Click and Keith D Cooper. 1995. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 17, 2 (1995), 181–196.
- [12] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1989. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 25–35.
- [13] Aarti Gupta. 1992. Formal hardware verification methods: A survey. In *Computer-Aided Verification*. Springer, 5–92.
- [14] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, et al. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *Proceedings of the 36th International Conference on Computer-Aided Design*. IEEE Press, 209–216.
- [15] Troy A Johnson, Sang-Ik Lee, Long Fei, Ayon Basumallik, Gautam Upadhyaya, Rudolf Eigenmann, and Samuel P Midkiff. 2004. Experiences in using cetus for source-to-source transformations. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 1–14.
- [16] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.
- [17] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore’s Law. [arXiv:cs.PL/2002.11054](https://arxiv.org/abs/2002.11054)
- [18] Patrick S. Li, Adam M. Izraelevitz, and Jonathan Bachrach. 2016. *Specification for the FIRRTL Language*. Technical Report UCB/EECS-2016-9. EECS Department, University of California, Berkeley. <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-9.html>
- [19] Cristian Mattarei, Makai Mann, Clark Barrett, Ross G Daly, Dillon Huff, and Pat Hanrahan. 2018. CoSA: Integrated Verification for Agile Hardware Design. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 1–5.
- [20] Diego Novillo. 2003. Tree ssa—a new high-level optimization framework for the gnu compiler collection. In *Proceedings of the Nord/USENIX Users Conference*. Citeseer.
- [21] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Acm Sigplan Notices*, Vol. 48. ACM, 519–530.
- [22] Randall Rustin. 1972. *Design and optimization of compilers*. Vol. 5. Prentice Hall.
- [23] Bruce R Schatz, Bruce W Leverett, Joseph M Newcomer, Andrew H Reiner, and William A Wulf. 1979. *TCOL Ada: an intermediate representation for the DOD standard programming language*. Technical Report. CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.
- [24] Amiral Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha, Tony Nowatzki, and Arrvindh Shriraman. 2019. μ IR: An intermediate representation for transforming and optimizing the microarchitecture of application accelerators. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 940–953.
- [25] Guy Lewis Steele Jr and Gerald Jay Sussman. 1976. *Lambda: The ultimate imperative*. Technical Report. MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB.
- [26] Stuart Sutherland. 2006. A proposal for a standard synthesizable subset of SystemVerilog-2005: What the IEEE failed to define. In *Design Verification Conference*. http://sutherland-hdl.com/papers/2006-DVCon_SystemVerilog_synthesis_subset_paper.pdf
- [27] Sheng-Hong Wang, Rafael T. Poggio, Qian Chen, Rohan Ganpati, and Jose Renau. 2019. LGraph: A Unified Data Model and API for Productive Open-Source Hardware Design. *Second Workshop on Open-Source EDA Technology (WOSET)* (Nov 2019).
- [28] Sheng-Hong Wang, Akash Sridhar, and Jose Renau. 2019. LNASt: A Language Neutral Intermediate Representation for Hardware Description Languages. *Second Workshop on Open-Source EDA Technology (WOSET)* (Nov 2019).
- [29] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanović. 2014. *The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0*. Technical Report. CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES.
- [30] Clifford Wolf. 2018. Yosys Manual, RTLIL specification. http://www.clifford.at/yosys/files/yosys_manual.pdf.
- [31] Florian Zaruba, Fabian Schuiki, Torsten Hoeffler, and Luca Benini. 2020. Snitch: A 10 kGE Pseudo Dual-Issue Processor for Area and Energy Efficient Execution of Floating-Point Intensive Workloads. [arXiv:cs.AR/2002.10143](https://arxiv.org/abs/2002.10143)