LLHD

A Multi-level Intermediate Representation for Hardware Description Languages

PLDI 2020

Fabian Schuiki[1], Andreas Kurth[1], Tobias Grosser[2], and Luca Benini[1]

[1] Integrated Systems Lab, ETH Zürich
[2] Scalable Parallel Computing Laboratory, ETH Zürich

LLHD

A Multi-level Intermediate Representation for Hardware Description Languages

PLDI 2020

Fabian Schuiki[1], Andreas Kurth[1], Tobias Grosser[2], and Luca Benini[1]

[1] Integrated Systems Lab, ETH Zürich
[2] Scalable Parallel Computing Laboratory, ETH Zürich


LLHD

A Multi-level Intermediate Representation for Hardware Description Languages

PLDI 2020

Fabian Schuiki[1], Andreas Kurth[1], Tobias Grosser[2], and Luca Benini[1]

[1] Integrated Systems Lab, ETH Zürich
[2] Scalable Parallel Computing Laboratory, ETH Zürich



LLHD

A Multi-level Intermediate Representation for Hardware Description Languages

PLDI 2020

Fabian Schuiki[1], Andreas Kurth[1], Tobias Grosser[2], and Luca Benini[1]

[1] Integrated Systems Lab, ETH Zürich
[2] Scalable Parallel Computing Laboratory, ETH Zürich




LLHD

A Multi-level Intermediate Representation for Hardware Description Languages

PLDI 2020

Fabian Schuiki[1], Andreas Kurth[1], Tobias Grosser[2], and Luca Benini[1]

[1] Integrated Systems Lab, ETH Zürich
[2] Scalable Parallel Computing Laboratory, ETH Zürich

The Pitch

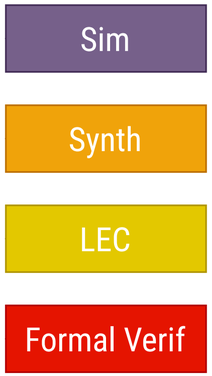
Chip Design Today

Chip Design Today

- Sheer complexity of chips leads to deep design toolchain

Chip Design Today

- Sheer complexity of chips leads to deep design toolchain
- Designs pass through many tools

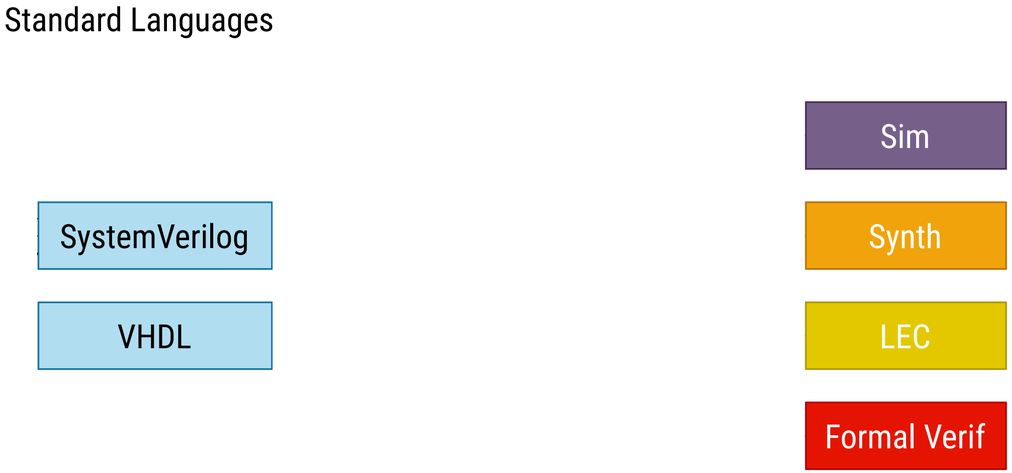


Chip Design Today

- Sheer complexity of chips leads to deep design toolchain
- Designs pass through many tools

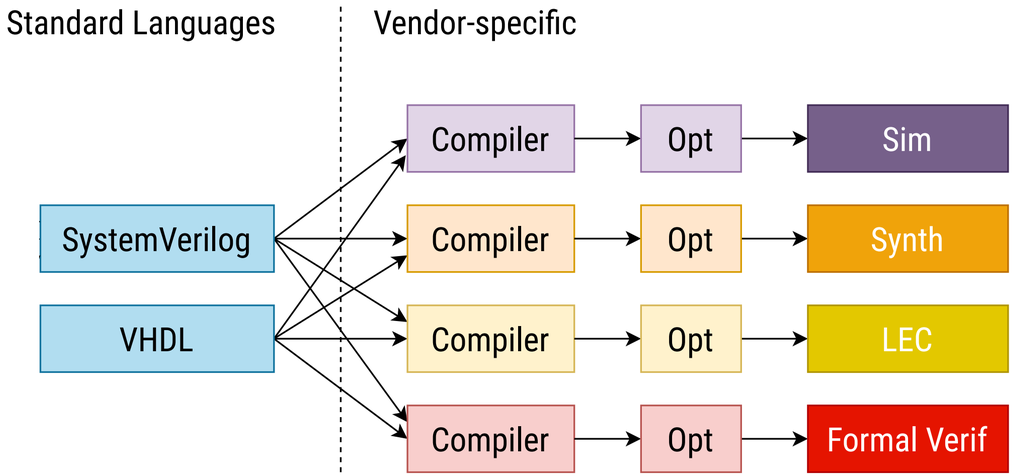
Chip Design Today

- Sheer complexity of chips leads to deep design toolchain
- Designs pass through many tools
- Existing HDLs are hard to implement



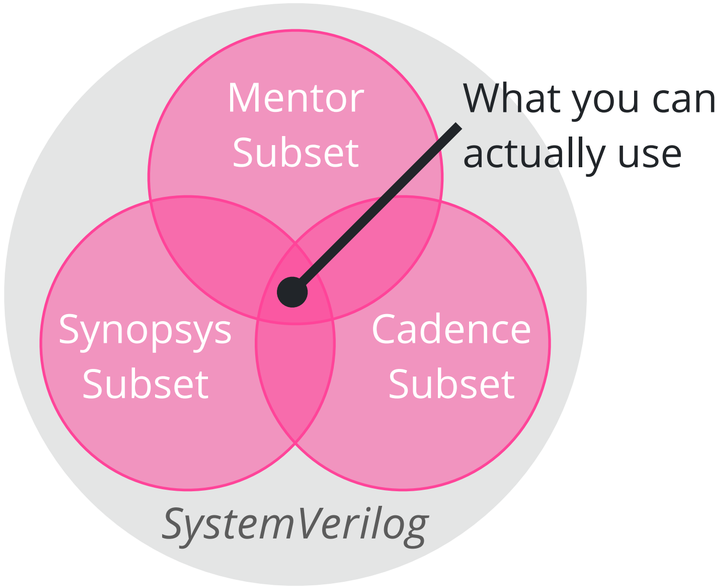
Chip Design Today

- Sheer complexity of chips leads to deep design toolchain
- Designs pass through many tools
- Existing HDLs are hard to implement
- Vendors implement their own subset of standard HDLs



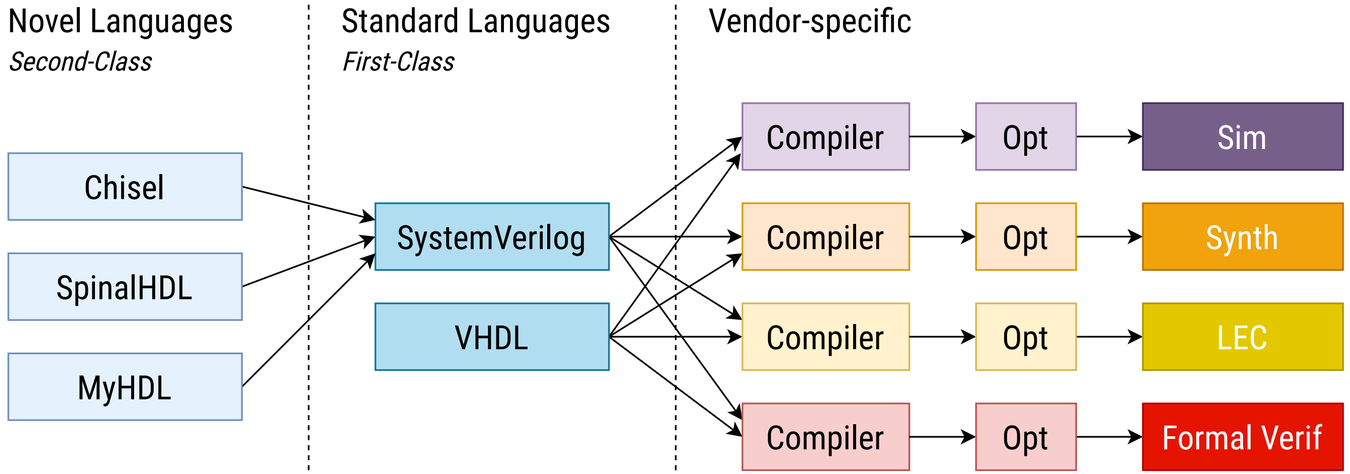
Chip Design Today

- Sheer complexity of chips leads to deep design toolchain
- Designs pass through many tools
- Existing HDLs are hard to implement
- Vendors implement their own subset of standard HDLs
- Cross-tool subset is minimal



Chip Design Today

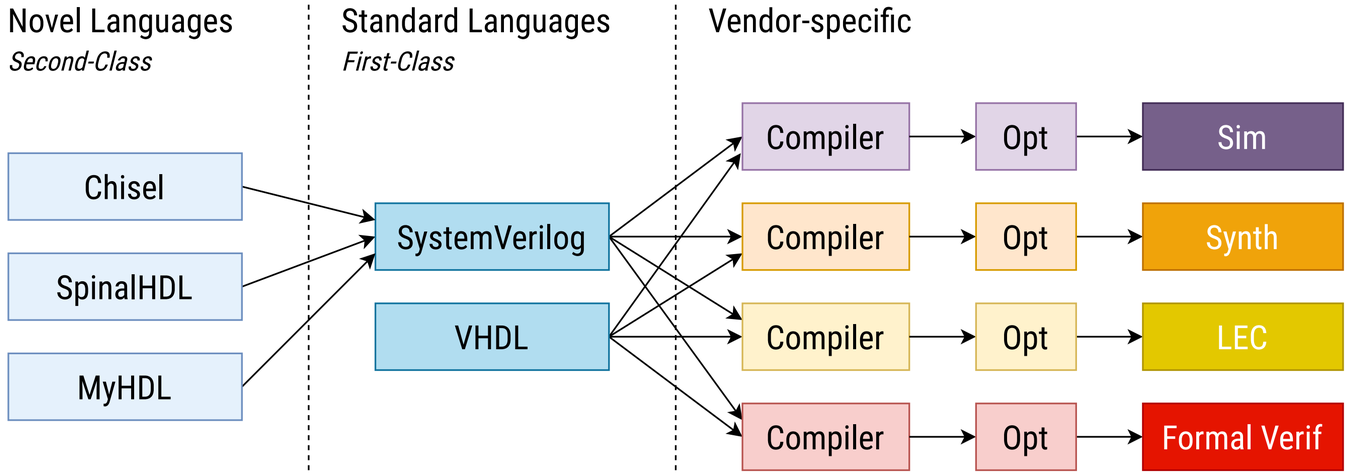
- Sheer complexity of chips leads to deep design toolchain
- Designs pass through many tools
- Existing HDLs are hard to implement
- Vendors implement their own subset of standard HDLs
- Cross-tool subset is minimal
- New languages have a hard time



Chip Design Today

- Sheer complexity of chips leads to deep design toolchain
- Designs pass through many tools
- Existing HDLs are hard to implement
- Vendors implement their own subset of standard HDLs
- Cross-tool subset is minimal
- New languages have a hard time

- **Hardware and compiler folks, unite!**



Chip Design Tomorrow

LLHD Project

◇ ◆ *Partially/fully implemented at time of writing*

Chip Design Tomorrow

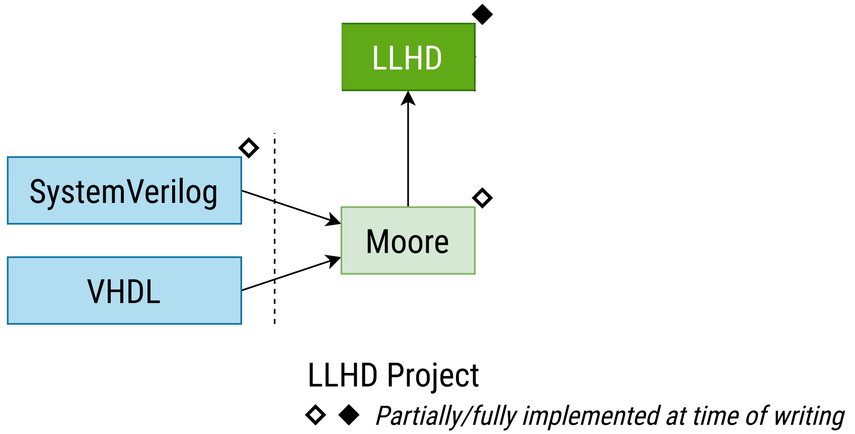
- LLHD is an IR for the hardware world

LLHD Project

◇ ◆ *Partially/fully implemented at time of writing*

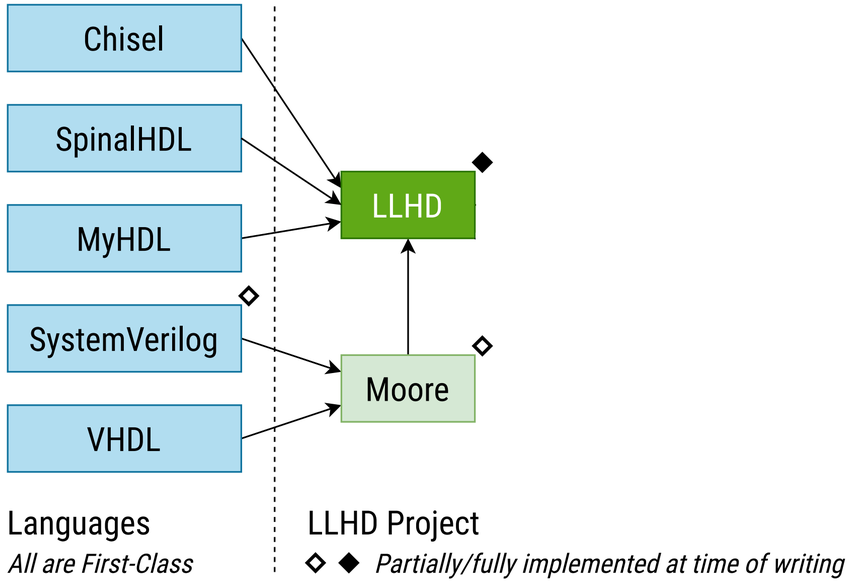
Chip Design Tomorrow

- LLHD is an IR for the hardware world
- One single frontend for HDLs
  (get the language right once™)



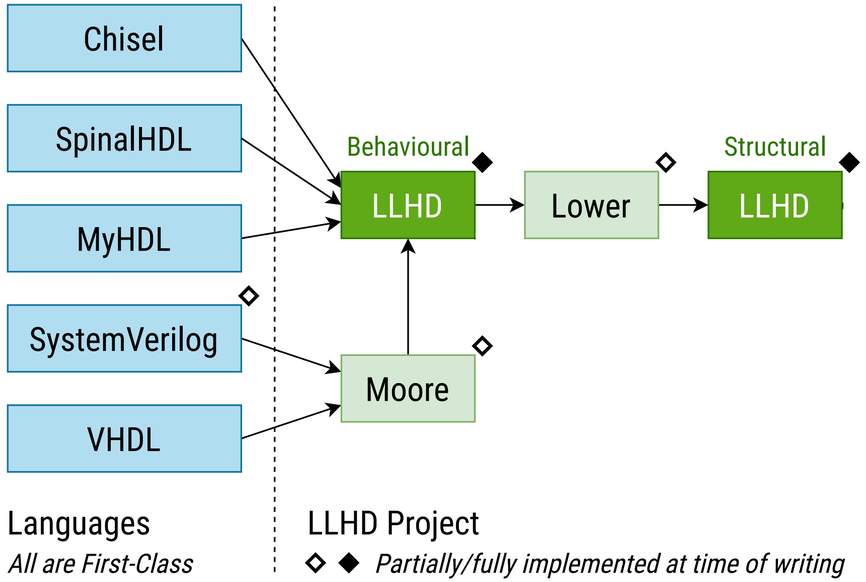
Chip Design Tomorrow

- LLHD is an IR for the hardware world
- One single frontend for HDLs
  (get the language right once™)
- New languages target LLHD directly



Chip Design Tomorrow

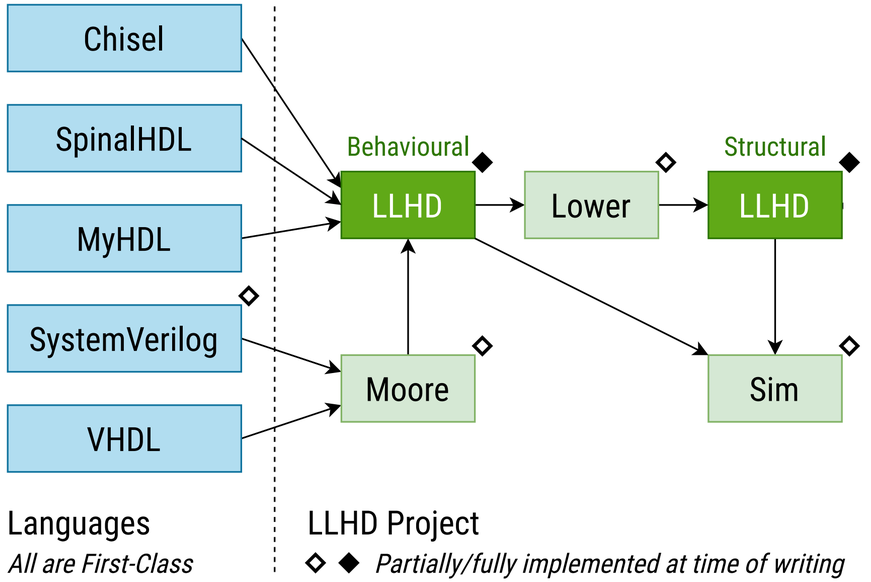
- LLHD is an IR for the hardware world
- One single frontend for HDLs (get the language right once™)
- New languages target LLHD directly
- Uniform lowering of the behavioural semantics in HDLs



Chip Design Tomorrow

- LLHD is an IR for the hardware world

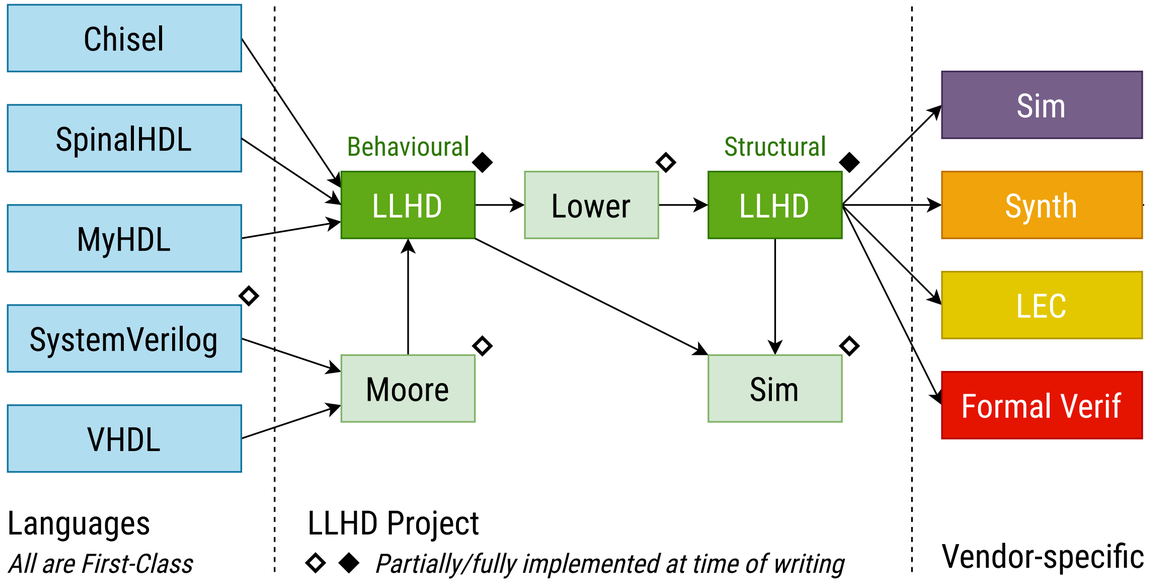
- One single frontend for HDLs (get the language right once™)
- New languages target LLHD directly
- Uniform lowering of the behavioural semantics in HDLs
- Simulation at the IR level



Chip Design Tomorrow

- LLHD is an IR for the hardware world

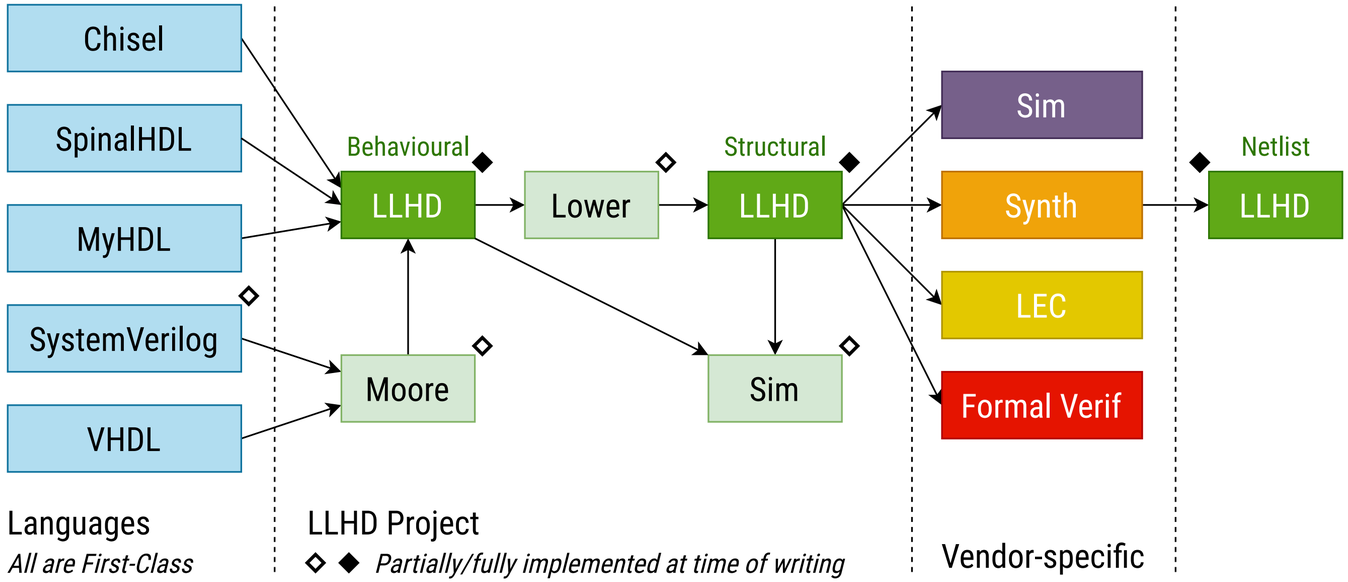
- One single frontend for HDLs (get the language right once™)
- New languages target LLHD directly
- Uniform lowering of the behavioural semantics in HDLs
- Simulation at the IR level
- Vendor tools ingest a simple IR



Chip Design Tomorrow

- LLHD is an IR for the hardware world

- One single frontend for HDLs (get the language right once™)
- New languages target LLHD directly
- Uniform lowering of the behavioural semantics in HDLs
- Simulation at the IR level
- Vendor tools ingest a simple IR
- And produce the same IR again



Meet LLHD

Time, Concurrency, Hierarchy

Time, Concurrency, Hierarchy

- Chips inherently time-sensitive
  - Signals take time to propagate
  - Digital designs have clocks

Time, Concurrency, Hierarchy

- Chips inherently **time-sensitive**
  - Signals take time to propagate
  - Digital designs have clocks
- Chips inherently **concurrent**
  - Gates sitting next to each other

Time, Concurrency, Hierarchy

- Chips inherently **time-sensitive**
  - Signals take time to propagate
  - Digital designs have clocks
- Chips inherently **concurrent**
  - Gates sitting next to each other
- Chips inherently **hierarchical**
  - Billions of transistors need replication

Time, Concurrency, Hierarchy

- Chips inherently **time-sensitive**
  - Signals take time to propagate
  - Digital designs have clocks
- Chips inherently **concurrent**
  - Gates sitting next to each other
- Chips inherently **hierarchical**
  - Billions of transistors need replication

- LLHD was heavily inspired by LLVM

Time, Concurrency, Hierarchy

- Chips inherently **time-sensitive**
  - Signals take time to propagate
  - Digital designs have clocks
- Chips inherently **concurrent**
  - Gates sitting next to each other
- Chips inherently **hierarchical**
  - Billions of transistors need replication

- LLHD was heavily inspired by LLVM
  - SSA form, BBs, PHI, naming, types

Time, Concurrency, Hierarchy

- Chips inherently **time-sensitive**
  - Signals take time to propagate
  - Digital designs have clocks
- Chips inherently **concurrent**
  - Gates sitting next to each other
- Chips inherently **hierarchical**
  - Billions of transistors need replication

- LLHD was heavily inspired by LLVM
  - SSA form, BBs, PHI, naming, types
- Hardware-specific instructions:

Time, Concurrency, Hierarchy

- Chips inherently **time-sensitive**
  - Signals take time to propagate
  - Digital designs have clocks
- Chips inherently **concurrent**
  - Gates sitting next to each other
- Chips inherently **hierarchical**
  - Billions of transistors need replication

- LLHD was heavily inspired by LLVM
  - SSA form, BBs, PHI, naming, types
- Hardware-specific instructions:
  - `wait` to suspend execution and resume at a later time (or `halt`)

Time, Concurrency, Hierarchy

- Chips inherently **time-sensitive**
  - Signals take time to propagate
  - Digital designs have clocks
- Chips inherently **concurrent**
  - Gates sitting next to each other
- Chips inherently **hierarchical**
  - Billions of transistors need replication

- LLHD was heavily inspired by LLVM
  - SSA form, BBs, PHI, naming, types
- Hardware-specific instructions:
  - `wait` to suspend execution and resume at a later time (or `halt`)
  - `drv` to schedule a signal value change

Time, Concurrency, Hierarchy

- Chips inherently **time-sensitive**
  - Signals take time to propagate
  - Digital designs have clocks
- Chips inherently **concurrent**
  - Gates sitting next to each other
- Chips inherently **hierarchical**
  - Billions of transistors need replication

- LLHD was heavily inspired by LLVM
  - SSA form, BBs, PHI, naming, types
- Hardware-specific instructions:
  - `wait` to suspend execution and resume at a later time (or `halt`)
  - `drv` to schedule a signal value change
  - `inst` to instantiate concurrent circuits

Time, Concurrency, Hierarchy

- Chips inherently **time-sensitive**
  - Signals take time to propagate
  - Digital designs have clocks
- Chips inherently **concurrent**
  - Gates sitting next to each other
- Chips inherently **hierarchical**
  - Billions of transistors need replication

- LLHD was heavily inspired by LLVM
  - SSA form, BBs, PHI, naming, types
- Hardware-specific instructions:
  - `wait` to suspend execution and resume at a later time (or `halt`)
  - `drv` to schedule a signal value change
  - `inst` to instantiate concurrent circuits
- Distilled-down concepts from HDLs:

Time, Concurrency, Hierarchy

- Chips inherently **time-sensitive**
  - Signals take time to propagate
  - Digital designs have clocks
- Chips inherently **concurrent**
  - Gates sitting next to each other
- Chips inherently **hierarchical**
  - Billions of transistors need replication

- LLHD was heavily inspired by LLVM
  - SSA form, BBs, PHI, naming, types
- Hardware-specific instructions:
  - `wait` to suspend execution and resume at a later time (or `halt`)
  - `drv` to schedule a signal value change
  - `inst` to instantiate concurrent circuits
- Distilled-down concepts from HDLs:
  - **Entities** model circuits structurally

Time, Concurrency, Hierarchy

- Chips inherently **time-sensitive**
  - Signals take time to propagate
  - Digital designs have clocks
- Chips inherently **concurrent**
  - Gates sitting next to each other
- Chips inherently **hierarchical**
  - Billions of transistors need replication

- LLHD was heavily inspired by LLVM
  - SSA form, BBs, PHI, naming, types
- Hardware-specific instructions:
  - `wait` to suspend execution and resume at a later time (or `halt`)
  - `drv` to schedule a signal value change
  - `inst` to instantiate concurrent circuits
- Distilled-down concepts from HDLs:
  - **Entities** model circuits structurally
  - **Processes** model circuits behaviourally

Time, Concurrency, Hierarchy

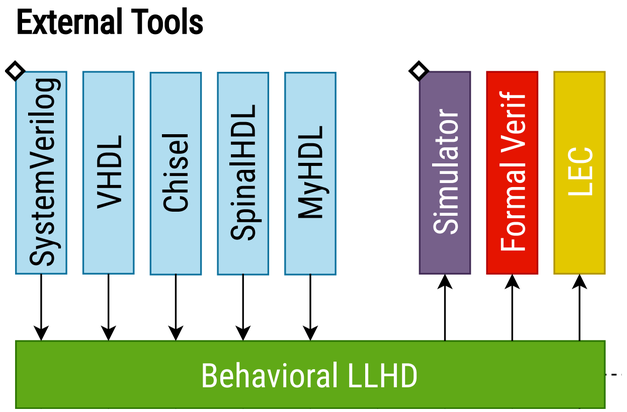
- Chips inherently **time-sensitive**
  - Signals take time to propagate
  - Digital designs have clocks
- Chips inherently **concurrent**
  - Gates sitting next to each other
- Chips inherently **hierarchical**
  - Billions of transistors need replication

- LLHD was heavily inspired by LLVM
  - SSA form, BBs, PHI, naming, types
- Hardware-specific instructions:
  - `wait` to suspend execution and resume at a later time (or `halt`)
  - `drv` to schedule a signal value change
  - `inst` to instantiate concurrent circuits
- Distilled-down concepts from HDLs:
  - **Entities** model circuits structurally
  - **Processes** model circuits behaviourally
  - **Functions** cannot wait/halt/drv

A Multi-Level IR

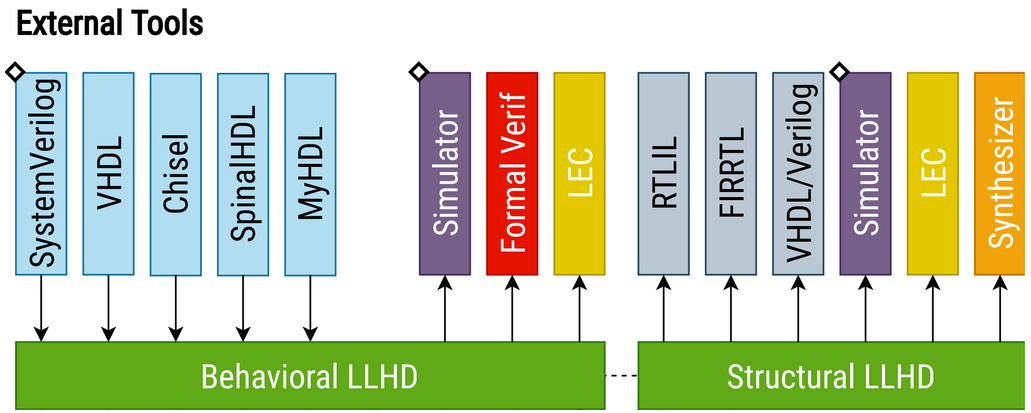
A Multi-Level IR

- **Behavioural** LLHD
  - Easy to target from HDLs
  - Functions, processes, entities



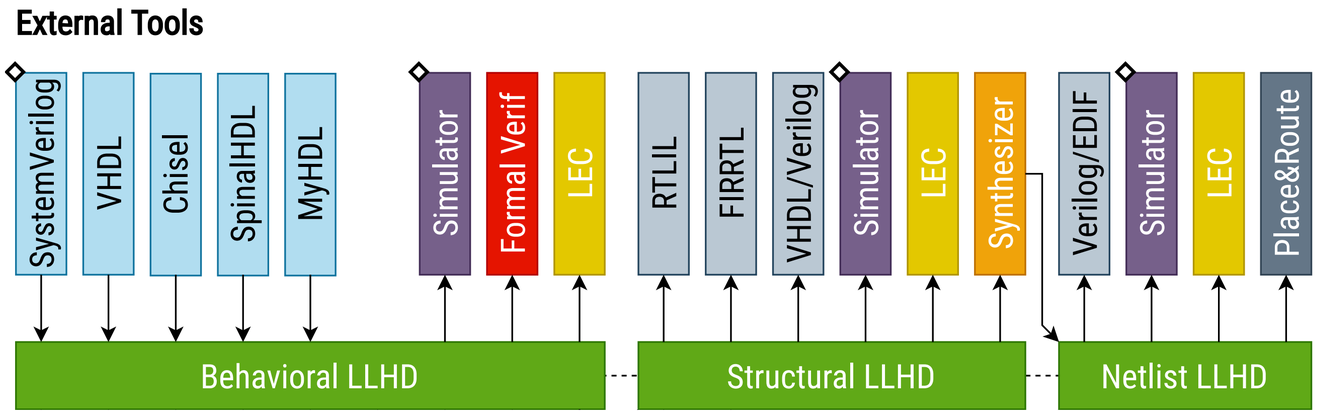
A Multi-Level IR

- **Behavioural** LLHD
  - Easy to target from HDLs
  - Functions, processes, entities
- **Structural** LLHD
  - Easy to ingest for synthesizers/tools
  - Entities



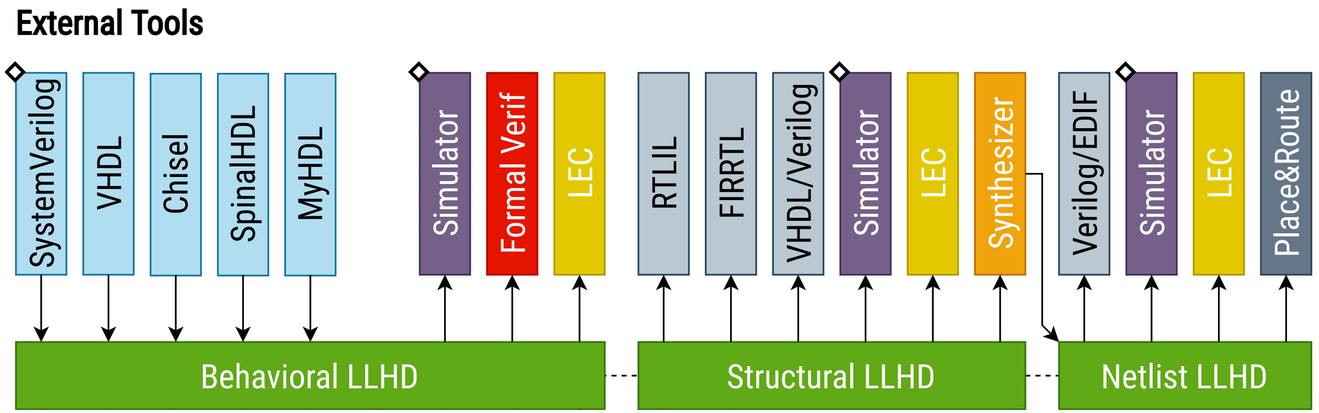
A Multi-Level IR

- **Behavioural** LLHD
  - Easy to target from HDLs
  - Functions, processes, entities
- **Structural** LLHD
  - Easy to ingest for synthesizers/tools
  - Entities
- **Netlist** LLHD
  - Entities with only inst/sig



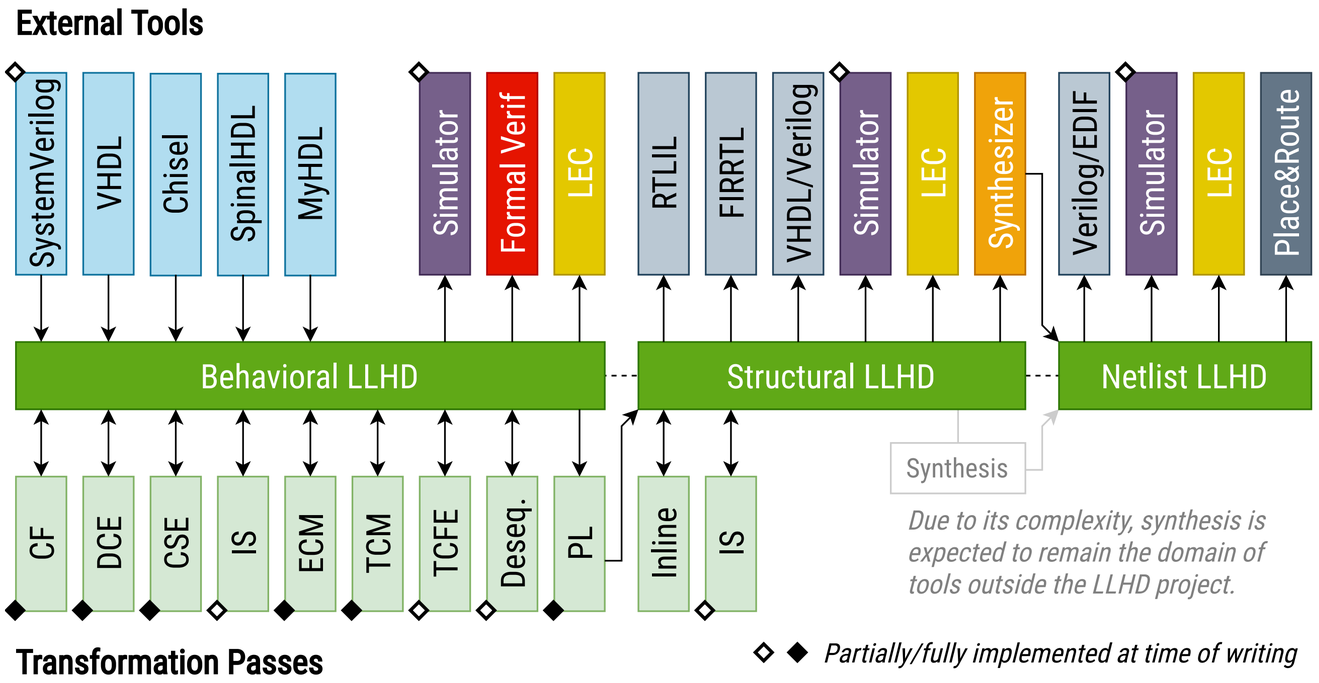
A Multi-Level IR

- **Behavioural** LLHD
  - Easy to target from HDLs
  - Functions, processes, entities
- **Structural** LLHD
  - Easy to ingest for synthesizers/tools
  - Entities
- **Netlist** LLHD
  - Entities with only inst/sig
- Covers the entire design flow



A Multi-Level IR

- **Behavioural** LLHD
  - Easy to target from HDLs
  - Functions, processes, entities
- **Structural** LLHD
  - Easy to ingest for synthesizers/tools
  - Entities
- **Netlist** LLHD
  - Entities with only inst/sig
- Covers the entire design flow
- Uniform lowering to structural LLHD



Mapping HDLs to LLHD

State of Hardware Design

State of Hardware Design

- RTL and verification written in SV/VHDL

State of Hardware Design

- RTL and verification written in SV/VHDL
- Hardware world is slow and conservative (first time right™)

State of Hardware Design

- RTL and verification written in SV/VHDL
- Hardware world is slow and conservative (first time right™)
- SV/VHDL legacy must be part of the story

State of Hardware Design

- RTL and verification written in SV/VHDL
- Hardware world is slow and conservative (first time right™)
- SV/VHDL legacy must be part of the story

- Running Example: Simple Accumulator

RTL Code:

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

Verification Code:

```
module acc_tb;
    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
```

Mapping RTL Code

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

Mapping RTL Code

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

Mapping RTL Code

- Processes have implied port list

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

```
proc @acc_ff (i1$ %clk, i32$ %d) -> (i32$ %q) {
init:
    %clk0 = prb i1$ %clk
    wait %check for %clk
check:
    %clk1 = prb i1$ %clk
    %chg = neq i1 %clk0, %clk1
    %posedge = and i1 %chg, %clk1
    br %posedge, %init, %event
event:
    %dp = prb i32$ %d
    %delay = const time 1ns
    drv i32$ %q, %dp after %delay
    br %init
}
```

Mapping RTL Code

- Processes have implied port list

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

- Flip-flop:
  - Posedge becomes wait, some logic and control flow
  - Assignment becomes drive in event

```
proc @acc_ff (i1$ %clk, i32$ %d) -> (i32$ %q) {
init:
    %clk0 = prb i1$ %clk
    wait %check for %clk
check:
    %clk1 = prb i1$ %clk
    %chg = neq i1 %clk0, %clk1
    %posedge = and i1 %chg, %clk1
    br %posedge, %init, %event
event:
    %dp = prb i32$ %d
    %delay = const time 1ns
    drv i32$ %q, %dp after %delay
    br %init
}
```

Mapping RTL Code

- Processes have implied port list
- Flip-flop:
  - Posedge becomes wait, some logic and control flow
  - Assignment becomes drive in event
- Combinational:
  - Reacting to inputs becomes wait at end

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

```
proc @acc_comb (i32$ %q, i32$ %x, i1$ %en) -> (i32$ %d) {
entry:
    %qp = prb i32$ %q
    %enp = prb i1$ %en
    %delay = const time 2ns
    drv i32$ %d, %qp after %delay
    br %enp, %final, %enabled
enabled:
    %xp = prb i32$ %x
    %sum = add i32 %qp, %xp
    drv i32$ %d, %sum after %delay
    br %final
final:
    wait %entry for %q, %x, %en
}
```

Mapping RTL Code

- Processes have implied port list
- Flip-flop:
  - Posedge becomes wait, some logic and control flow
  - Assignment becomes drive in event
- Combinational:
  - Reacting to inputs becomes wait at end
- Module instantiates the processes

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

```
entity @acc (i1$ %clk, i32$ %x, i1$ %en) -> (i32$ %q) {
    %zero = const i32 0
    %d = sig i32 %zero
    %q = sig i32 %zero
    inst @acc_ff (i1$ %clk, i32$ %d) -> (i32$ %q)
    inst @acc_comb (i32$ %q, i32$ %x, i1$ %en) -> (i32$ %d)
}
```

Mapping Verification Code

```
module acc_tb;
    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
```

Mapping Verification Code

```
module acc_tb;
    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
```

```
func @acc_tb_check (i32 %i, i32 %q) void {
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)
    ret
}
```

Mapping Verification Code

```
module acc_tb;
    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
```

```
func @acc_tb_check (i32 %i, i32 %q) void {
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)
    ret
}
```

Mapping Verification Code

```
module acc_tb;
    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
```

```
proc @acc_tb_initial (i32$ %q) -> (i1$ %clk, i32$ %x, i1$ %en) {
entry:
    %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
    drv i1$ %clk, %bit0 after %del2ns
    wait %next for %del2ns
next:
    %qp = prb i32$ %q
    call void @acc_tb_check (i32 %ip, i32 %qp)
    %in = add i32 %ip, %one
    st i32* %i, %in
    %cont = ult i32 %ip, %many
    br %cont, %end, %loop
end:
    halt
}
```

Mapping Verification Code

```
module acc_tb;
    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
```

```
proc @acc_tb_initial (i32$ %q) -> (i1$ %clk, i32$ %x, i1$ %en) {
entry:
    %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
    drv i1$ %clk, %bit0 after %del2ns
    wait %next for %del2ns
next:
    %qp = prb i32$ %q
    call void @acc_tb_check (i32 %ip, i32 %qp)
    %in = add i32 %ip, %one
    st i32* %i, %in
    %cont = ult i32 %ip, %many
    br %cont, %end, %loop
end:
    halt
}
```

Mapping Verification Code

```
module acc_tb;
    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
```

```
proc @acc_tb_initial (i32$ %q) -> (i1$ %clk, i32$ %x, i1$ %en) {
entry:
    %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
    drv i1$ %clk, %bit0 after %del2ns
    wait %next for %del2ns
next:
    %qp = prb i32$ %q
    call void @acc_tb_check (i32 %ip, i32 %qp)
    %in = add i32 %ip, %one
    st i32* %i, %in
    %cont = ult i32 %ip, %many
    br %cont, %end, %loop
end:
    halt
}
```

Mapping Verification Code

```
module acc_tb;
    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
```

```
proc @acc_tb_initial (i32$ %q) -> (i1$ %clk, i32$ %x, i1$ %en) {
entry:
    %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
    drv i1$ %clk, %bit0 after %del2ns
    wait %next for %del2ns
next:
    %qp = prb i32$ %q
    call void @acc_tb_check (i32 %ip, i32 %qp)
    %in = add i32 %ip, %one
    st i32* %i, %in
    %cont = ult i32 %ip, %many
    br %cont, %end, %loop
end:
    halt
}
```

Mapping Verification Code

```
module acc_tb;
    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
```

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

Lowering to Structural LLHD

Basics and Early Code Motion (ECM)

Basics and Early Code Motion (ECM)

- Lowering as a chain of passes

Basics and Early Code Motion (ECM)

- Lowering as a chain of passes
- Objective: Remove control flow

Basics and Early Code Motion (ECM)

- Lowering as a chain of passes
- Objective: Remove control flow

- Well-known basics: **CF**, **DCE**, **CSE**, **IS**

Basics and Early Code Motion (ECM)

- Lowering as a chain of passes
- Objective: Remove control flow

- Well-known basics: **CF**, **DCE**, **CSE**, **IS**
- Reduce complexity of operations

Basics and Early Code Motion (ECM)

- Lowering as a chain of passes
- Objective: Remove control flow

- Well-known basics: **CF**, **DCE**, **CSE**, **IS**
- Reduce complexity of operations

- **Early Code Motion** (ECM)

Basics and Early Code Motion (ECM)

- Lowering as a chain of passes
- Objective: Remove control flow

- Well-known basics: **CF**, **DCE**, **CSE**, **IS**
- Reduce complexity of operations

- **Early Code Motion** (ECM)
- Move arithmetic out of basic blocks

Basics and Early Code Motion (ECM)

- Lowering as a chain of passes
- Objective: Remove control flow

- Well-known basics: **CF**, **DCE**, **CSE**, **IS**
- Reduce complexity of operations

- **Early Code Motion** (ECM)
- Move arithmetic out of basic blocks
- Similar to Loop-Invariant Code Motion in LLVM, but eager

Basics and Early Code Motion (ECM)

- Lowering as a chain of passes
- Objective: Remove control flow

- Well-known basics: **CF**, **DCE**, **CSE**, **IS**
- Reduce complexity of operations

- **Early Code Motion** (ECM)
- Move arithmetic out of basic blocks
- Similar to Loop-Invariant Code Motion in LLVM, but eager

```
proc @acc_ff (i1$ %clk, i32$ %d) -> (i32$ %q) {
init:
    %clk0 = prb i1$ %clk
    wait %check for %clk
check:
    %clk1 = prb i1$ %clk
    %chg = neq i1 %clk0, %clk1
    %posedge = and i1 %chg, %clk1
    br %posedge, %init, %event
event:
    %dp = prb i32$ %d
    %delay = const time 1ns
    drv i32$ %q, %dp after %delay
    br %init
}

proc @acc_comb (i32$ %q, i32$ %x, i1$ %en) -> (i32$ %d) {
entry:
    %qp = prb i32$ %q
    %enp = prb i1$ %en
    %delay = const time 2ns
    drv i32$ %d, %qp after %delay
    br %enp, %final, %enabled
enabled:
    %xp = prb i32$ %x
    %sum = add i32 %qp, %xp
    drv i32$ %d, %sum after %delay
    br %final
final:
    wait %entry for %q, %x, %en
}
```

Basics and Early Code Motion (ECM)

- Lowering as a chain of passes
- Objective: Remove control flow

- Well-known basics: **CF**, **DCE**, **CSE**, **IS**
- Reduce complexity of operations

- **Early Code Motion** (ECM)
- Move arithmetic out of basic blocks
- Similar to Loop-Invariant Code Motion in LLVM, but eager

```
proc @acc_ff (i1$ %clk, i32$ %d) -> (i32$ %q) {
init:
    %clk0 = prb i1$ %clk
    wait %check for %clk
check:
    %clk1 = prb i1$ %clk
    %chg = neq i1 %clk0, %clk1
    %posedge = and i1 %chg, %clk1
    br %posedge, %init, %event
event:
    %dp = prb i32$ %d
    %delay = const time 1ns
    drv i32$ %q, %dp after %delay
    br %init
}
```

```
init:
    %delay = const time 1ns
    %clk0 = prb i1$ %clk
    wait %check for %clk
check:
    %clk1 = prb i1$ %clk
    %dp = prb i32$ %d
    %chg = neq i1 %clk0, %clk1
    %posedge = and i1 %chg, %clk1
    br %posedge, %init, %event
event:
    drv i32$ %q, %dp after %delay
    br %init
```

```
proc @acc_comb (i32$ %q, i32$ %x, i1$ %en) -> (i32$ %d) {
entry:
    %qp = prb i32$ %q
    %enp = prb i1$ %en
    %delay = const time 2ns
    drv i32$ %d, %qp after %delay
    br %enp, %final, %enabled
enabled:
    %xp = prb i32$ %x
    %sum = add i32 %qp, %xp
    drv i32$ %d, %sum after %delay
    br %final
final:
    wait %entry for %q, %x, %en
}
```

```
entry:
    %qp = prb i32$ %q
    %enp = prb i1$ %en
    %xp = prb i32$ %x
    %sum = add i32 %qp, %xp
    %delay = const time 2ns
    drv i32$ %d, %qp after %delay
    br %enp, %final, %enabled
enabled:
    drv i32$ %d, %sum after %delay
    br %final
final:
    wait %entry for %q, %x, %en
```

Temporal Code Motion (TCM)

Temporal Code Motion (TCM)

- Move drives out of basic blocks

Temporal Code Motion (TCM)

- Move drives out of basic blocks
- Add control flow predicate to drive condition operand

Temporal Code Motion (TCM)

- Move drives out of basic blocks
- Add control flow predicate to drive condition operand
- Makes drives control-flow independent

Temporal Code Motion (TCM)

- Move drives out of basic blocks
- Add control flow predicate to drive condition operand
- Makes drives control-flow independent

- Waits divide process into separate temporal regions (TR)

Temporal Code Motion (TCM)

- Move drives out of basic blocks
- Add control flow predicate to drive condition operand
- Makes drives control-flow independent

- Waits divide process into separate temporal regions (TR)
- Use TRs to reason about synchronicity of instructions

Temporal Code Motion (TCM)

- Move drives out of basic blocks
- Add control flow predicate to drive condition operand
- Makes drives control-flow independent

- Waits divide process into separate temporal regions (TR)
- Use TRs to reason about synchronicity of instructions
- Drives moved into exiting BB of TR

Temporal Code Motion (TCM)

- Move drives out of basic blocks
- Add control flow predicate to drive condition operand
- Makes drives control-flow independent

- Waits divide process into separate temporal regions (TR)
- Use TRs to reason about synchronicity of instructions
- Drives moved into exiting BB of TR

```
proc @acc_ff (i1$ %clk, i32$ %d) -> (i32$ %q) {
init:
    %delay = const time 1ns
    %clk0 = prb i1$ %clk
    wait %check for %clk
check:
    %clk1 = prb i1$ %clk
    %dp = prb i32$ %d
    %chg = neq i1 %clk0, %clk1
    %posedge = and i1 %chg, %clk1
    br %posedge, %init, %event
event:
    drv i32$ %q, %dp after %delay
    br %init
}
```

```
proc @acc_comb (i32$ %q, i32$ %x, i1$ %en) -> (i32$ %d) {
entry:
    %qp = prb i32$ %q
    %enp = prb i1$ %en
    %xp = prb i32$ %x
    %sum = add i32 %qp, %xp
    %delay = const time 2ns
    drv i32$ %d, %qp after %delay
    br %enp, %final, %enabled
enabled:
    drv i32$ %d, %sum after %delay
    br %final
final:
    wait %entry for %q, %x, %en
}
```

Temporal Code Motion (TCM)

- Move drives out of basic blocks
- Add control flow predicate to drive condition operand
- Makes drives control-flow independent

- Waits divide process into separate temporal regions (TR)
- Use TRs to reason about synchronicity of instructions
- Drives moved into exiting BB of TR

```
proc @acc_ff (i1$ %clk, i32$ %d) -> (i32$ %q) {
init:
    %delay = const time 1ns
    %clk0 = prb i1$ %clk
    wait %check for %clk
check:
    %clk1 = prb i1$ %clk
    %dp = prb i32$ %d
    %chg = neq i1 %clk0, %clk1
    %posedge = and i1 %chg, %clk1
    br %posedge, %init, %event
event:
    drv i32$ %q, %dp after %delay
    br %init
}
```

```
init:
    %delay = const time 1ns
    %clk0 = prb i1$ %clk
    wait %check for %clk
check:
    %clk1 = prb i1$ %clk
    %dp = prb i32$ %d
    %chg = neq i1 %clk0, %clk1
    %posedge = and i1 %chg, %clk1
    br %posedge, %aux, %event
event:
    br %aux
aux:
    drv i32$ %q, %dp after %delay
                    if %posedge
    br %init
```

```
proc @acc_comb (i32$ %q, i32$ %x, i1$ %en) -> (i32$ %d) {
entry:
    %qp = prb i32$ %q
    %enp = prb i1$ %en
    %xp = prb i32$ %x
    %sum = add i32 %qp, %xp
    %delay = const time 2ns
    drv i32$ %d, %qp after %delay
    br %enp, %final, %enabled
enabled:
    drv i32$ %d, %sum after %delay
    br %final
final:
    wait %entry for %q, %x, %en
}
```

```
entry:
    %qp = prb i32$ %q
    %xp = prb i32$ %x
    %enp = prb i1$ %en
    %sum = add i32 %qp, %xp
    %delay = const time 2ns
    br %enp, %final, %enabled
enabled:
    br %final
final:
    %dn = phi i32 [%qp, %entry],
                  [%sum, %enabled]
    drv i32$ %d, %dn after %delay
    wait %entry for %q, %x, %en
```

Total Control Flow Elimination (TCFE)

Total Control Flow Elimination (TCFE)

- Replace control flow with data flow, branches and phi with multiplexers

Total Control Flow Elimination (TCFE)

- Replace control flow with data flow, branches and phi with multiplexers
- Previous passes canonicalize BBs to phi nodes and branches

Total Control Flow Elimination (TCFE)

- Replace control flow with data flow, branches and phi with multiplexers
- Previous passes canonicalize BBs to phi nodes and branches
- TCFE reduces these to one BB per TR

Total Control Flow Elimination (TCFE)

- Replace control flow with data flow, branches and phi with multiplexers
- Previous passes canonicalize BBs to phi nodes and branches
- TCFE reduces these to one BB per TR

```
proc @acc_ff (i1$ %clk, i32$ %d) -> (i32$ %q) {
init:
    %delay = const time 1ns
    %clk0 = prb i1$ %clk
    wait %check for %clk
check:
    %clk1 = prb i1$ %clk
    %dp = prb i32$ %d
    %chg = neq i1 %clk0, %clk1
    %posedge = and i1 %chg, %clk1
    br %posedge, %aux, %event
event:
    br %aux
aux:
    drv i32$ %q, %dp after %delay
                    if %posedge
    br %init
}
```

```
proc @acc_comb (i32$ %q, i32$ %x, i1$ %en) -> (i32$ %d) {
entry:
    %qp = prb i32$ %q
    %xp = prb i32$ %x
    %enp = prb i1$ %en
    %sum = add i32 %qp, %xp
    %delay = const time 2ns
    br %enp, %final, %enabled
enabled:
    br %final
final:
    %dn = phi i32 [%qp, %entry],
                  [%sum, %enabled]
    drv i32$ %d, %dn after %delay
    wait %entry for %q, %x, %en
}
```

Total Control Flow Elimination (TCFE)

- Replace control flow with data flow, branches and phi with multiplexers
- Previous passes canonicalize BBs to phi nodes and branches
- TCFE reduces these to one BB per TR

```
proc @acc_ff (i1$ %clk, i32$ %d) -> (i32$ %q) {
init:
    %delay = const time 1ns
    %clk0 = prb i1$ %clk
    wait %check for %clk
check:
    %clk1 = prb i1$ %clk
    %dp = prb i32$ %d
    %chg = neq i1 %clk0, %clk1
    %posedge = and i1 %chg, %clk1
    br %posedge, %aux, %event
event:
    br %aux
aux:
    drv i32$ %q, %dp after %delay
                 if %posedge
    br %init
}
```

```
init:
    %delay = const time 1ns
    %clk0 = prb i1$ %clk
    wait %check for %clk
check:
    %clk1 = prb i1$ %clk
    %dp = prb i32$ %d
    %chg = neq i1 %clk0, %clk1
    %posedge = and i1 %chg, %clk1
    drv i32$ %q, %dp after %delay
                 if %posedge
    br %init
```

```
proc @acc_comb (i32$ %q, i32$ %x, i1$ %en) -> (i32$ %d) {
entry:
    %qp = prb i32$ %q
    %xp = prb i32$ %x
    %enp = prb i1$ %en
    %sum = add i32 %qp, %xp
    %delay = const time 2ns
    br %enp, %final, %enabled
enabled:
    br %final
final:
    %dn = phi i32 [%qp, %entry],
                  [%sum, %enabled]
    drv i32$ %d, %dn after %delay
    wait %entry for %q, %x, %en
}
```

```
entry:
    %qp = prb i32$ %q
    %xp = prb i32$ %x
    %enp = prb i1$ %en
    %sum = add i32 %qp, %xp
    %delay = const time 2ns
    %dns = [i32 %qp, %sum]
    %dn = mux i32 %dns, %enp
    drv i32$ %d, %dn after %delay
    wait %entry for %q, %x, %en
```

Process Lowering (PL)

Process Lowering (PL)

- Replace trivial processes with entities

Process Lowering (PL)

- Replace trivial processes with entities

- Applies to processes with:
  - One basic block
  - Wait terminator sensitive to all signals

Process Lowering (PL)

- Replace trivial processes with entities

- Applies to processes with:
  - One basic block
  - Wait terminator sensitive to all signals
- A.k.a. **combinational** processes

Process Lowering (PL)

- Replace trivial processes with entities

- Applies to processes with:
  - One basic block
  - Wait terminator sensitive to all signals
- A.k.a. **combinational** processes

- Remove the wait

Process Lowering (PL)

- Replace trivial processes with entities

- Applies to processes with:
  - One basic block
  - Wait terminator sensitive to all signals
- A.k.a. **combinational** processes

- Remove the wait
- Migrate instructions to entity with the same signature

Process Lowering (PL)

- Replace trivial processes with entities
- Applies to processes with:
  - One basic block
  - Wait terminator sensitive to all signals
- A.k.a. **combinational** processes
- Remove the wait
- Migrate instructions to entity with the same signature

```
proc @acc_comb (i32$ %q, i32$ %x, i1$ %en) -> (i32$ %d) {
entry:
    %qp = prb i32$ %q
    %xp = prb i32$ %x
    %enp = prb i1$ %en
    %sum = add i32 %qp, %xp
    %delay = const time 2ns
    %dns = [i32 %qp, %sum]
    %dn = mux i32 %dns, %enp
    drv i32$ %d, %dn after %delay
    wait %entry for %q, %x, %en
}
```

Process Lowering (PL)

- Replace trivial processes with entities
- Applies to processes with:
  - One basic block
  - Wait terminator sensitive to all signals
- A.k.a. **combinational** processes
- Remove the wait
- Migrate instructions to entity with the same signature

```
proc @acc_comb (i32$ %q, i32$ %x, i1$ %en) -> (i32$ %d) {
entry:
    %qp = prb i32$ %q
    %xp = prb i32$ %x
    %enp = prb i1$ %en
    %sum = add i32 %qp, %xp
    %delay = const time 2ns
    %dns = [i32 %qp, %sum]
    %dn = mux i32 %dns, %enp
    drv i32$ %d, %dn after %delay
    wait %entry for %q, %x, %en
}
```

```
entity @acc_comb (i32$ %q, i32$ %x, i1$ %en) -> (i32$ %d) {
    %qp = prb i32$ %q
    %xp = prb i32$ %x
    %enp = prb i1$ %en
    %sum = add i32 %qp, %xp
    %delay = const time 2ns
    %dns = [i32 %qp, %sum]
    %dn = mux i32 %dns, %enp
    drv i32$ %d, %dn after %delay
}
```

Desequentialization

Desequentialization

- Identify flip-flops and latches

Desequentialization

- Identify flip-flops and latches

- Applies to processes with:
  - Two basic blocks and TRs
  - Wait and branch terminator

Desequentialization

- Identify flip-flops and latches

- Applies to processes with:
  - Two basic blocks and TRs
  - Wait and branch terminator
- A.k.a. **sequential** processes

Desequentialization

- Identify flip-flops and latches

- Applies to processes with:
  - Two basic blocks and TRs
  - Wait and branch terminator
- A.k.a. **sequential** processes

- Canonicalize drive conditions into Disjunctive Normal Form

Desequentialization

- Identify flip-flops and latches

- Applies to processes with:
  - Two basic blocks and TRs
  - Wait and branch terminator
- A.k.a. **sequential** processes

- Canonicalize drive conditions into Disjunctive Normal Form
- Identify edge/level detection logic

Desequentialization

- Identify flip-flops and latches

- Applies to processes with:
  - Two basic blocks and TRs
  - Wait and branch terminator
- A.k.a. **sequential** processes

- Canonicalize drive conditions into Disjunctive Normal Form
- Identify edge/level detection logic
- Replace drives with registers in an entity with the process' signature

Desequentialization

- Identify flip-flops and latches
- Applies to processes with:
  - Two basic blocks and TRs
  - Wait and branch terminator
- A.k.a. **sequential** processes
- Canonicalize drive conditions into Disjunctive Normal Form
- Identify edge/level detection logic
- Replace drives with registers in an entity with the process' signature

```
proc @acc_ff (i1$ %clk, i32$ %d) -> (i32$ %q) {
init:
    %delay = const time 1ns
    %clk0 = prb i1$ %clk
    wait %check for %clk
check:
    %clk1 = prb i1$ %clk
    %dp = prb i32$ %d
    %chg = neq i1 %clk0, %clk1
    %posedge = and i1 %chg, %clk1
    drv i32$ %q, %dp after %delay if %posedge
    br %init
}
```

Desequentialization

- Identify flip-flops and latches
- Applies to processes with:
  - Two basic blocks and TRs
  - Wait and branch terminator
- A.k.a. **sequential** processes
- Canonicalize drive conditions into Disjunctive Normal Form
- Identify edge/level detection logic
- Replace drives with registers in an entity with the process' signature

```
proc @acc_ff (i1$ %clk, i32$ %d) -> (i32$ %q) {
init:
    %delay = const time 1ns
    %clk0 = prb i1$ %clk
    wait %check for %clk
check:
    %clk1 = prb i1$ %clk
    %dp = prb i32$ %d
    %chg = neq i1 %clk0, %clk1
    %posedge = and i1 %chg, %clk1
    drv i32$ %q, %dp after %delay if %posedge
    br %init
}
```

```
entity @acc_ff (i1$ %clk, i32$ %d) -> (i32$ %q) {
    %delay = const time 1ns
    %clkp = prb i1$ %clk
    %dp = prb i32$ %d
    reg i32$ %q, %dp rise %clkp after %delay
}
```

Inlining and Cleanup

```
entity @acc (i1$ %clk, i32$ %x, i1$ %en) -> (i32$ %q) {
    %zero = const i32 0
    %d = sig i32 %zero
    %q = sig i32 %zero
    inst @acc_ff (i1$ %clk, i32$ %d) -> (i32$ %q)
    inst @acc_comb (i32$ %q, i32$ %x, i1$ %en) -> (i32$ %d)
}
```

Inlining and Cleanup

```
entity @acc (i1$ %clk, i32$ %x, i1$ %en) -> (i32$ %q) {
    %zero = const i32 0
    %d = sig i32 %zero
    %q = sig i32 %zero
    inst @acc_ff (i1$ %clk, i32$ %d) -> (i32$ %q)
    inst @acc_comb (i32$ %q, i32$ %x, i1$ %en) -> (i32$ %d)
}
```

```
entity @acc (i1$ %clk, i32$ %x, i1$ %en) -> (i32$ %q) {
    %clkp = prb i1$ %clk
    %qp = prb i32$ %q
    %xp = prb i32$ %x
    %enp = prb i1$ %en
    %sum = add i32 %qp, %xp
    reg i32$ %q, %sum rise %clkp if %enp
}
```

Results

JIT-accelerated Simulation

JIT-accelerated Simulation

- We investigate designs of different scale:
  - From small filters and buffers
  - To a full RISC-V core

JIT-accelerated Simulation

- We investigate designs of different scale:
  - From small filters and buffers
  - To a full RISC-V core

- Proof of concept: **llhd-blaze**

JIT-accelerated Simulation

- We investigate designs of different scale:
  - From small filters and buffers
  - To a full RISC-V core

- Proof of concept: **llhd-blaze**
- Maps LLHD to LLVM

JIT-accelerated Simulation

- We investigate designs of different scale:
  - From small filters and buffers
  - To a full RISC-V core

- Proof of concept: **llhd-blaze**
- Maps LLHD to LLVM
- Uses JIT execution engine and runtime that implements the event queue

JIT-accelerated Simulation

- We investigate designs of different scale:
  - From small filters and buffers
  - To a full RISC-V core
- Proof of concept: llhd-blaze
- Maps LLHD to LLVM
- Uses JIT execution engine and runtime that implements the event queue
- Up to 2.4x faster than commercial tools

			Sim. Time [s]		
Design	**LoC**	**Cycles**	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

JIT-accelerated Simulation

- We investigate designs of different scale:
  - From small filters and buffers
  - To a full RISC-V core
- Proof of concept: llhd-blaze
- Maps LLHD to LLVM
- Uses JIT execution engine and runtime that implements the event queue
- Up to 2.4x faster than commercial tools
- Not faster across the board yet

			Sim. Time [s]		
Design	**LoC**	**Cycles**	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

JIT-accelerated Simulation

- We investigate designs of different scale:
  - From small filters and buffers
  - To a full RISC-V core

- Proof of concept: llhd-blaze
- Maps LLHD to LLVM
- Uses JIT execution engine and runtime that implements the event queue

- Up to 2.4x faster than commercial tools
- Not faster across the board yet
- Written in half a day, no optimizations

			Sim. Time [s]		
Design	**LoC**	**Cycles**	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

Other Hardware IRs

Other Hardware IRs

- All vendor tools have IRs; not considered

Other Hardware IRs

- All vendor tools have IRs; not considered
- Most IRs built for synthesizers / placers

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

Other Hardware IRs

- All vendor tools have IRs; not considered
- Most IRs built for synthesizers / placers
- Focus on structural circuits and netlist

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

Other Hardware IRs

- All vendor tools have IRs; not considered
- Most IRs built for synthesizers / placers
- Focus on structural circuits and netlist
- A few have limited support for:
  - Behavioural/verification constructs
  - 4-valued logic

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

Other Hardware IRs

- All vendor tools have IRs; not considered
- Most IRs built for synthesizers / placers
- Focus on structural circuits and netlist
- A few have limited support for:
  - Behavioural/verification constructs
  - 4-valued logic
- None support 9-valued logic

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

Other Hardware IRs

- All vendor tools have IRs; not considered
- Most IRs built for synthesizers / placers
- Focus on structural circuits and netlist
- A few have limited support for:
  - Behavioural/verification constructs
  - 4-valued logic
- None support 9-valued logic

- LLHD is the only IR that is:
  - Turing-complete
  - Built for the entire design flow

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

Memory Footprint

Memory Footprint

- In-memory / on-disk footprint is an important aspect of an IR

Memory Footprint

- In-memory / on-disk footprint is an important aspect of an IR

Memory Footprint

- In-memory / on-disk footprint is an important aspect of an IR
- We compare:

Memory Footprint

- In-memory / on-disk footprint is an important aspect of an IR

- We compare:
  - SystemVerilog source text

Memory Footprint

- In-memory / on-disk footprint is an important aspect of an IR

- We compare:
  - SystemVerilog source text
  - LLHD assembly text

Memory Footprint

- In-memory / on-disk footprint is an important aspect of an IR

- We compare:
  - SystemVerilog source text
  - LLHD assembly text
  - LLHD bitcode size estimate

Memory Footprint

- In-memory / on-disk footprint is an important aspect of an IR

- We compare:
  - SystemVerilog source text
  - LLHD assembly text
  - LLHD bitcode size estimate
  - LLHD in-memory structures

Memory Footprint

- In-memory / on-disk footprint is an important aspect of an IR
- We compare:
  - SystemVerilog source text
  - LLHD assembly text
  - LLHD bitcode size estimate
  - LLHD in-memory structures
- Assembly larger due to expressive SV

	SV	**LLHD** [kB]		
Design	[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

Memory Footprint

- In-memory / on-disk footprint is an important aspect of an IR
- We compare:
  - SystemVerilog source text
  - LLHD assembly text
  - LLHD bitcode size estimate
  - LLHD in-memory structures
- Assembly larger due to expressive SV
- Bitcode can be very compact

	SV	**LLHD** [kB]		
Design	[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

Memory Footprint

- In-memory / on-disk footprint is an important aspect of an IR
- We compare:
  - SystemVerilog source text
  - LLHD assembly text
  - LLHD bitcode size estimate
  - LLHD in-memory structures
- Assembly larger due to expressive SV
- Bitcode can be very compact
- Full RISC-V core only 1 MB of memory

	SV	**LLHD** [kB]		
Design	[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

Thanks!

Check out the artifact docker image 🐳

Online Playground and Language Reference:

- llhd.io

Repositories and Issue Trackers:

- github.com/fabianschuiki/llhd
- github.com/fabianschuiki/llhd-sim
- github.com/fabianschuiki/moore

Or take a plunge:

```
git clone https://github.com/fabianschuiki/moore
cd moore
cargo install
```