Introducing hardware generator debugger and how we can debug at source-level

Keyi Zhang
Computer Science Department
Stanford University
What are hardware generators?

First generation:

- “Preprocessor”-based string templating
- Using Perl or Python to preprocess the SystemVerilog to overcome the language limit.
What are hardware generators?

First generation:
- “Preprocessor”-based string templating
- Using Perl or Python to preprocess the SystemVerilog to overcome the language limit.

Second generation:
- Embedded domain specific languages that use host languages features to improve design productivity
  - Chisel (scala), Manga (Python), PyMTL (Python)
- “Mini” compiler tool chains with IR

```gherkin
code
module `mname` (
  clk, clk_en, reset,
  //; for(my $i=0; $i<$pe_output_count; $i++) {
    pe_output_`$i`,
    //; }
  //; for(my $i=0; $i<$sides; $i++) {
    //; for(my $j=0; $j<$number_of_outputs; $j++) {
      out_`$i`_`$j`,
      in_`$i`_`$j`,
      //; }
    //; }
  //; }
```
What are hardware generators?

First generation:
- “Preprocessor”-based string templating
- Using Perl or Python to preprocess the SystemVerilog to overcome the language limit.

Second generation”
- Embedded domain specific languages that use host languages features to improve design productivity
- Chisel (scala), Manga (Python), PyMTL (Python)
- “Mini” compiler tool chains with IR
What are hardware generators?

First generation:
- “Preprocessor”-based string templating
- Using Perl or Python to preprocess the SystemVerilog to overcome the language limit.

Second generation:
- Embedded domain specific languages that use host languages features to improve design productivity
  - Chisel (scala), Manga (Python), PyMTL (Python)
- “Mini” compiler tool chains with IR

```scala
class DoubleBufferFifo[T <: Data](gen: T, depth: Int) extends Fifo(gen: T, depth: Int) {
  private class DoubleBuffer[T <: Data](gen: T) extends Module {
    val io = IO(new FifoIO(gen))
    val empty :: one :: two :: Nil = Enum(3)
    val stateReg = RegInit(empty)
    val dataReg = Reg(gen)
    val shadowReg = Reg(gen)
    ...
  }
}
```
Google tried Chisel 3 years ago
Google tried Chisel 3 years ago

The bad
Google tried Chisel 3 years ago

The ugly
Let’s address the bad part: simulation and debugging

Latest RocketChip + Chisel3

// SLT, SLTU
val slt = 
  Mux(io.in1(xLen-1) === io.in2(xLen-1), io.adder_out(xLen-1),
  Mux(cmpUnsigned(io.fn), io.in2(xLen-1), io.in1(xLen-1)))
io.cmp_out := cmpInverted(io.fn) ^ Mux(cmpEq(io.fn), in1_xor_in2 === UInt(0), slt)

// SLL, SRL, SRA
val (shamt, shin_r) = 
  if (xLen == 32) (io.in2(4,0), io.in1)
  else {
    require(xLen == 64)
    val shin_hi_32 = Fill(32, isSub(io.fn) && io.in1(31))
    val shin_hi = Mux(io.dw === DW_64, io.in1(63,32), shin_hi_32)
    val shamt = Cat(io.in2(5) & (io.dw === DW_64), io.in2(4,0))
    (shamt, Cat(shin_hi, io.in1(31,0)))
  }
val shin = Mux(io.fn === FN_SR || io.fn === FN_SRA, shin_r, Reverse(shin_r))
Let's address the bad part: simulation and debugging

Latest RocketChip + Chisel3
hgdb: batteries included

- Source-level debugging
- Minimal performance overhead
- No RTL changes required
- Two complete debuggers
  - VSCode
  - gdb-inspired console debugger
- All major simulators
  - Big 3
  - Verilator
  - iverilog
- FSDB and VCD Replay
  - Reverse debugging!

RocketChip Benchmark
What hgdb brings to the table
How hgdb works

VSCode Debugger → gdb-like Debugger
                      ↓
hgdb debugger runtime
                      ↑

Unified simulator interface

Synopsys VCS®  →  Cadence® Xcelium™  →  Replay tool

Unified symbol table interface

SQLite  →  Others
Keeps the overhead low: breakpoint emulation
Keeps the overhead low: breakpoint emulation
Keeps the overhead low: breakpoint emulation
But does it work all the time?

No. Unfortunately. 😞

Combinational logic

```
logic [31:0] sum;
logic [31:0] data[4];

always_comb begin
  sum = 0;
  for (int i = 0; i < 4; i++) begin
    if (data[i] % 2) begin
      sum += data[i];
    end
  end
end
```

Partial results lost when the state stabilizes
Loop unrolling and SSA to rescue

### Loop unroll

```verilog
logic [31:0] sum;
logic [31:0] data[4];

always_comb begin
    sum = 0;
    if (data[0] % 2) sum += data[0];
    if (data[1] % 2) sum += data[1];
    if (data[2] % 2) sum += data[2];
    if (data[3] % 2) sum += data[3];
end
```

### SSA Transform

```verilog
logic [31:0] sum, sum0, sum1, sum2, sum3;
logic [31:0] data[4];

assign sum0 = data[0] % 2? data[0]: 0;
assign sum1 = data[1] % 2? sum0 + data[1]: sum0;
assign sum2 = data[2] % 2? sum1 + data[2]: sum1;
assign sum3 = data[3] % 2? sum2 + data[3]: sum2;
assign sum = sum3;
```

Static single assignment
Loop unrolling and SSA to rescue

```verilog
logic [31:0] sum;
logic [31:0] data[4];

always_comb begin
  sum = 0;
  for (int i = 0; i < 4; i++) begin
    if (data[i] % 2) begin
      sum += data[i];
    end
  end
end
```

Multiple mapping!

```verilog
logic [31:0] sum, sum0, sum1, sum2, sum3;
logic [31:0] data[4];

assign sum0 = data[0] % 2? data[0]: 0;
assign sum1 = data[1] % 2? sum0 + data[1]: sum0;
assign sum2 = data[2] % 2? sum1 + data[2]: sum1;
assign sum3 = data[3] % 2? sum2 + data[3]: sum2;
assign sum = sum3;
```

Original form               Final form
Loop unrolling and SSA to rescue

```verilog
logic [31:0] sum;
logic [31:0] data[4];

always_comb begin
    sum = 0;
    for (int i = 0; i < 4; i++) begin
        if (data[i] % 2) begin
            sum += data[i];
        end
    end
end
```

Original form

```verilog
logic [31:0] sum, sum0, sum1, sum2, sum3;
logic [31:0] data[4];

assign sum0 = data[0] % 2? data[0]: 0;
assign sum1 = data[1] % 2? sum0 + data[1]: sum0;
assign sum2 = data[2] % 2? sum1 + data[2]: sum1;
assign sum3 = data[3] % 2? sum2 + data[3]: sum2;
assign sum = sum3;
```

Final form

Enable condition
Breakpoint emulation loop

@ (posedge clk)

*Reverse time

Done

Fetch next bps

Evaluate bps in parallel

Has any bp hit

Yes
No

Send bps to users
Breakpoint emulation loop

Fetch ordering!

@ (posedge clk)

*Reverse time

Done

Fetch next bps

Evaluate bps in parallel

Has any bp hit

Yes

No

Send bps to users

Breakpoint 1
Breakpoint 2
Breakpoint 3
Breakpoint 4
...

Normal execution
Breakpoint emulation loop

Fetch ordering!

@ (posedge clk)

*Reverse time

Fetch next bps

Evaluate bps in parallel

Has any bp hit

Yes

Send bps to users

No
Unified simulator interface

Primitives:
- Place callback on value change
- Get signal values
- Get design hierarchy
- Reverse time (optional)

enjenir work!
Unified symbol table interface

Primitives:
- Get breakpoints from source location
- Get scope and instance information for each breakpoint
- Resolve scoped variable names to full name
- Resolve instance variable names to full name

Unified simulator interface

Synopsys VCS®
Cadence® Xcelium™
Replay tool
SQLite
Others

Debugger
Native
RPC

Unified symbol table interface
Hgdb debuggers

Visual Studio Code

Terminal based
Hgdb debuggers

- Visual Studio Code Debugger
- gdb-like Debugger

Hgdb debugger runtime

- Unified simulator interface
- Unified symbol table interface

- Synopsys VCS®
- Cadence® Xcelium™
- Replay tool
- SQLite
- Others
How hgdb works with Chisel

```scala
node _T = bits(io.fn, 3, 3) @[ALU.scala 40:29]
node _T_1 = bits(io.in1, 31, 31) @[ALU.scala 76:55]
node _T_2 = and(_T, _T_1) @[ALU.scala 76:46]
node _T_3 = bits(_T_2, 0, 0) @[Bitwise.scala 72:15]
node _T_4 = mux(_T_3,
    UInt<32>("h0fffffffff"),
    UInt<32>("h00")) @[Bitwise.scala 72:12]
node _T_5 = eq(io.dw, UInt<1>("h01")) @[ALU.scala 77:31]
node _T_6 = bits(io.in1, 63, 32) @[ALU.scala 77:48]
node hi = mux(_T_5, _T_6, _T_4) @[ALU.scala 77:24]
```
How hgdb works with Chisel

node _io_adder_out_T = add(io.in1, in2_inv) @[ALU.scala 63:26]
node _io_adder_out_T_1 = tail(_io_adder_out_T, 1) @[ALU.scala 63:26]
node _io_adder_out_T_2 = bits(io.fn, 3, 3) @[ALU.scala 40:29]
node _io_adder_out_T_3 = add(_io_adder_out_T_1, _io_adder_out_T_2) @[ALU.scala 63:36]
node _io_adder_out_T_4 = tail(_io_adder_out_T_3, 1) @[ALU.scala 63:36]
io.adder_out <= _io_adder_out_T_4 @[ALU.scala 63:16]

wire [63:0] _io_adder_out_T_1 = io_in1 + in2_inv; // @[ALU.scala 63:26]
wire [63:0] _GEN_1 = {{63'd0}, io_fn[3]}; // @[ALU.scala 63:36]
assign io_adder_out = _io_adder_out_T_1 + _GEN_1; // @[ALU.scala 63:36]
How hgdbg works with Chisel

Let’s insert `DontTouchAnnotation` everywhere!

```bash
mkdir -p /home/keyl/workspace/rocket-chip-debug/vsim/generated-src/
/home/keyl/workspace/hgdbg-firrtl/bin/firrtl -i /home/keyl/workspace/rocket-chip-debug/vsim/generated-src/freechips.rocketchip.system.DefaultConfig.fir \
-o /home/keyl/workspace/rocket-chip-debug/vsim/generated-src/freechips.rocketchip.system.DefaultConfig.v \
-X verilog \
--infer-rw TestHarness \
--hgdb.toml rocket-chip.toml \
-0 0 \
--repl-seq-mem -c:TestHarness:-o:/home/keyl/workspace/rocket-chip-debug/vsim/generated-src/freechips.rocketchip.system.DefaultConfig.conf \
-faf /home/keyl/workspace/rocket-chip-debug/vsim/generated-src/freechips.rocketchip.system.DefaultConfig.anno.json \
-td /home/keyl/workspace/rocket-chip-debug/vsim/generated-src/freechips.rocketchip.system.DefaultConfig/
   -fct firrtl.passes:inlineInstances, |

Exception in thread "main" firrtl.transforms.DontTouchAnnotation$DontTouchNotFoundException: Target marked dontTouch (InterruptBusWrapper.auto int bus int out 0) not found! It was probably accidentally deleted. Please check that your custom transforms are not responsible and then file an issue on GitHub: https://github.com/firrtlproject/firrtl/issues/new
```
It finally works!

Input: CircuitState
Output: Table
Annotations ← {};

foreach node ∈ CircuitState do // First pass
    if node is statement then
        node.enable ← ComputeEnableCondition(node);
    end
    Annotation ← Annotations ∪ {node}
end

// FIRRTL transformations;
IRNodes ← {};

foreach node ∈ Annotations do
    if node ∈ CircuitState then
        IRNodes ← IRNodes ∪ node;
    end
end

Table ← ComputeSymbolTable(IRNodes);

First pass:
- Insert annotation and compute enable condition
- (Debug mode) insert DontTouchAnnotation

Second pass:
- Collect annotations and only compute symbol table if the IRNode still exists
It finally works!

```plaintext
Input: CircuitState
Output: Table
Annotations ← {};

foreach node ∈ CircuitState do //
  if node is statement then
    node.enable ← ComputeEnable
  end
  Annotation ← Annotations ∪ {node}
end

// FIRRTL transformations;
IRNodes ← {};
foreach node ∈ Annotations do
  if node ∈ CircuitState then
    IRNodes ← IRNodes ∪ node
  end
end

Table ← ComputeSymbolTable(IRNodes);
```

https://github.com/Kuree/hgdb-firrtl
Future work in progress

- Work presented to LLVM/CIRCT and SiFive
  - Integrate hgdb to CIRCT system
  - Leverage LLVM compiler toolchain

- Watchpoint
  - We believe watchpoint can be effectively emulated with extra information from the IR
  - Similar to breakpoint emulation
  - Can replace driver tracing if used with reverse debugging

- Symbol table redesign
  - Current implementation “too flat”.
  - More akin to DWARF but simpler and easy to read/write
Conclusion

- Hgdb bridges the gap between hardware generator frameworks and existing simulators
  - Efficient
  - Works with all major simulator vendors
  - What you debug is what you tape out
- Hardware generators are fairly new, and debugging infrastructure is almost non-existing.
  - CIRCIT is the future???
- Hgdb is an open-source framework
  - Contributions are welcome!
  
https://github.com/Kuree/hgdb
Thank you