DESIGN AUTOMATION CONFERENCE

JULY 10 - 14, 2022
MOSCONIC WEST CENTER
SAN FRANCISCO, CA, USA
Bringing Source-Level Debugging Frameworks to Hardware Generators

Keyi Zhang, Zain Asgar, Mark Horowitz
Computer Science Department
Stanford University
The Good, the Bad and the Ugly

• The Good:
  • Huge leaps in front-end design tools productivity
    • Hardware generator frameworks embedded in a host programming languages, such as Chisel
    • High level synthesis tools that turn C/C++ into RTL
  • More software-oriented concepts/constructs
    • Object-oriented programming
    • Functional programming
    • Software/hardware co-design
The Good, the Bad and the Ugly

- **The Good:**
  - Huge leaps in front-end design tools productivity
  - Hardware generator frameworks embedded in host programming languages, such as Chisel
  - High level synthesis tools that turn C/C++ into RTL
  - More software-oriented and constructs
  - Object-oriented programming
  - Functional programming
  - Software/hardware co-design

---

```
// 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))
```

Chisel code for RocketChip
The Good, the Bad and the Ugly

The Bad and the Ugly:
- Generated RTL is obfuscated due to compiler optimizations
  - Low-level RTL
  - Loses designer intent
- Verification has to be done at RTL level for integration tests
- Productivity gain from front-end design is lost in verification
The Good, the Bad and the Ugly

• The Bad and the Ugly:
  • Generated RTL is obfuscated due to compiler optimizations
  • Low-level RTL
  • Loses designer intent
  • Verification has to be done at RTL level for integration tests
  • Productivity gain from front-end design losses on verification

Generated RTL from the code shown before
Introducing hgdb

- 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!
System design

VSCode Debugger | gdb-like Debugger

---

hgdb debugger runtime

Unified simulator interface

- Synopsys VCS®
- Cadence® Xcelium™
- Replay tool

Unified symbol table interface

- SQLite
- Others
Low overhead breakpoint emulation
Low overhead breakpoint emulation
Low overhead breakpoint emulation
Emulate with correct semantics

```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
```

Stack frame:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>sum</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
Insight:
- Static single assignment (SSA) and loop unrolling are commonly used in compiler optimization.
- Use these transformation artifacts to help debugging.
SSA and loop unrolling to rescue

```
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 Code
SSA and loop unrolling to rescue

```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
```

Code after loop unrolling
SSA and loop unrolling to rescue

```verilog
logic [31:0] sum, sum0, sum1, sum2, sum3, sum4;
logic [31:0] data[4];

assign sum0 = 0;
assign sum1 = data[0] % 2? data[0]: sum0;
assign sum2 = data[1] % 2? sum1 + data[1]: sum1;
assign sum3 = data[2] % 2? sum2 + data[2]: sum2;
assign sum4 = data[3] % 2? sum3 + data[3]: sum3;
assign sum = sum4;
```

Code after Single-Static-Assignment transformation
Breakpoint emulation with SSA

```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
```

One to many mapping due to loop unrolling

```verilog
logic [31:0] sum, sum0, sum1, sum2, sum3, sum4;
logic [31:0] data[4];

assign sum0 = 0;
assign sum1 = data[0] % 2? data[0]: sum0;
assign sum2 = data[1] % 2? sum1 + data[1]: sum1;
assign sum3 = data[2] % 2? sum2 + data[2]: sum2;
assign sum4 = data[3] % 2? sum3 + data[3]: sum3;
assign sum = sum4;
```
Breakpoint emulation with SSA

Use SSA mapping to construct stack frame
Breakpoint emulation with SSA

Use SSA mapping to construct stack frame

```
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

assign sum0 = 0;
assign sum1 = data[0] % 2? data[0]: sum0;
assign sum2 = data[1] % 2? sum1 + data[1]: sum1;
assign sum3 = data[2] % 2? sum2 + data[2]: sum2;
assign sum4 = data[3] % 2? sum3 + data[3]: sum3;
assign sum = sum4;
```
Breakpoint emulation with SSA

Use SSA mapping to construct stack frame
Breakpoint emulation with SSA

Use SSA mapping to construct stack frame
Breakpoint emulation with SSA

![Stack frame]

```
logic [31:0] sum, sum0, sum1, sum2, sum3, sum4;
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
```

Storing static values into symbol table when unrolling the loop
Breakpoint emulation with SSA

Storing static values into symbol table when unrolling the loop
Breakpoint emulation with SSA

Data

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

sum ← sum2

i

Stack frame

```
logic [31:0] sum, sum0, sum1, sum2, sum3, sum4;
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
```

Storing static values into symbol table when unrolling the loop
Breakpoint emulation with SSA

Storing static values into symbol table when unrolling the loop
Breakpoint emulation with SSA

Using SSA transformation to compute “enable condition”

```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
```

```verilog
logic [31:0] sum, sum0, sum1, sum2, sum3, sum4;
logic [31:0] data[4];

assign sum0 = 0;
assign sum1 = data[0] % 2? data[0]: sum0;
assign sum2 = data[1] % 2? sum1 + data[1]: sum1;
assign sum3 = data[2] % 2? sum2 + data[2]: sum2;
assign sum4 = data[3] % 2? sum3 + data[3]: sum3;
assign sum = sum4;
```
Breakpoint emulation with SSA

Using SSA transformation to compute “enable condition”

```
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
```

```
logic [31:0] sum, sum0, sum1, sum2, sum3, sum4;
logic [31:0] data[4];

assign sum0 = 0;
assign sum1 = data[0] % 2 ? data[0]: sum0;
assign sum2 = data[1] % 2 ? sum1 + data[1]: sum1;
assign sum3 = data[2] % 2 ? sum2 + data[2]: sum2;
assign sum4 = data[3] % 2 ? sum3 + data[3]: sum3;
assign sum = sum4;
```
Breakpoint emulation with SSA

Using SSA transformation to compute “enable condition”
Breakpoint emulation with SSA

Using SSA transformation to compute “enable condition”

```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
```

```verilog
logic [31:0] sum, sum0, sum1, sum2, sum3, sum4;
logic [31:0] data[4];
assign sum0 = 0;
assign sum1 = data[0] % 2? data[0]: sum0;
assign sum2 = data[1] % 2? sum1 + data[1]: sum1;
assign sum3 = data[2] % 2? sum2 + data[2]: sum2;
assign sum4 = data[3] % 2? sum3 + data[3]: sum3;
assign sum = sum4;
```
Breakpoint emulation with SSA

```
logic [31:0] sum, sum0, sum1, sum2, sum3, sum4;
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
```

```
assign sum0 = 0;
assign sum1 = data[0] % 2? data[0]: sum0;
assign sum2 = data[1] % 2? sum1 + data[1]: sum1;
assign sum3 = data[2] % 2? sum2 + data[2]: sum2;
assign sum4 = data[3] % 2? sum3 + data[3]: sum3;
assign sum = sum4;
```

Put everything together: only two breakpoints are enabled
Breakpoint emulation with SSA

Stack frame

```
logic [31:0] sum, sum0, sum1, sum2, sum3, sum4;
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
```

Put everything together: only two breakpoints are enabled
Breakpoint emulation loop

@ (posedge clk)

- Fetch next bps
- Evaluate bps in parallel
- Has any bp hit
  - Yes: Done
  - No: Send bps to users

*Reverse time
Breakpoint emulation loop

1. Fetch next bps
2. Evaluate bps in parallel
3. Check if any breakpoint has hit:
   - Yes: Send bps to users
   - No: Return to fetch next bps

*Reverse time

Lexical ordering of breakpoints:
- src.cpp:10
- src.cpp:11
- src.cpp:12
- src.cpp:13
Breakpoint emulation loop

- @ (posedge clk)
- Fetch next bps
- Evaluate bps in parallel
- Has any bp hit
  - Yes: Reverse time
  - No: Send bps to users

Reverse lexical ordering

Intra-cycle reverse debugging

src.cpp:10
src.cpp:11
src.cpp:12
src.cpp:13
Breakpoint emulation loop

@ (posedge clk)

Fetch next bps

Evaluate bps in parallel

Has any bp hit

Send bps to users

Done

*Reverse time

Reversed lexical ordering

src.cpp:10
src.cpp:11
src.cpp:12
src.cpp:13

Inter-cycle reverse debugging
Unified simulator interface

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

Visual Studio Code

Terminal based
Hgdb debuggers

[Diagram showing the Hgdb debugger runtime with connections to Visual Studio Code and Terminal based debuggers]

- **Native**
- **RPC**

**Visual Studio Code**
- VSCode Debugger
- adb-like Debugger

**Terminal based**
- Unified simulator interface
- Unified symbol table interface

- Synopsys VCS®
- Cadence® Xcelium™
- Replay tool
- SQLite
- Others
Integration with hardware generators

Chisel/Firrtl  MLIR/CIRCT  Vitis HLS
Working with Firrtl compiler

- **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

**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);
Performance evaluation

- Rocketchip built-in benchmark
- Debug mode refers to passes disable compiler optimization
- 5% performance overhead
Conclusion

- Hardware generators are new, and debugging infrastructure is missing
- Hgdb connects hardware generator frameworks and existing simulators
  - Works with all major simulator vendors
  - Brings source level debugging
- Hgdb is an open-source framework from Stanford AHA! center
  - Contributions are welcome!

https://github.com/Kuree/hgdb