CrispyHDL

Java Inner DSL for Verilog

Nuno Paulino
CTM
INESC TEC

CrispyHDL

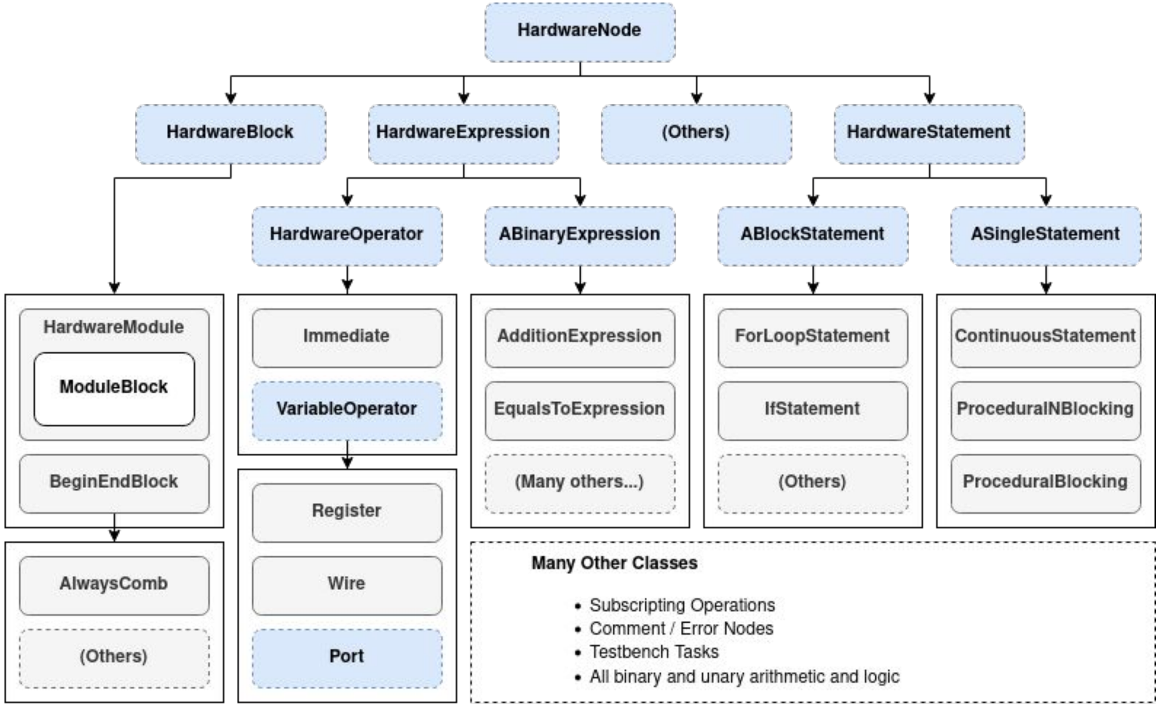
- Binary Translation Framework required Verilog as an output target…
    - How to generate Verilog systematically?
    - To avoid direct string manipulation (brittle…), we employed compiler techniques (robust)
    - A (nearly) complete **Verilog AST package**

- Verilog AST package grew → **Separate CrispyHDL project**

- **CrispyHDL**
    - Internal Java DSL for Verilog (Inspired by SpinalHDL, Chisel3, etc)
    - Generation of hardware via reusable blocks exploiting high level abstractions
        - Inheritance
        - Generics/Templates
        - Instantiation loops
        - Interfaces

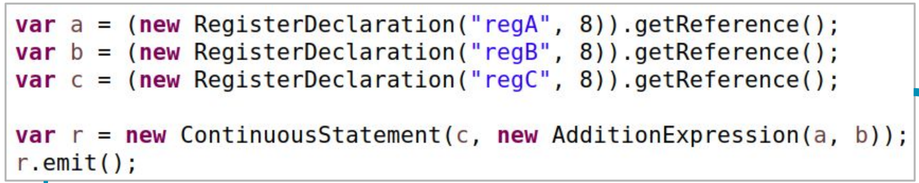
Verilog Abstract Syntax Tree (AST) - Java Classes

- Each node
  - Is a Verilog element
  - Emits its respective Verilog source
- Trees of nodes are constructed via Java DSL to generate complete Verilog modules



Verilog Abstract Syntax Tree (AST) - Example

- Simple statement example
- Crispy classes are **not** meant to be explicitly instantiated like this
- The DSL (wrappers) hides this verbosity

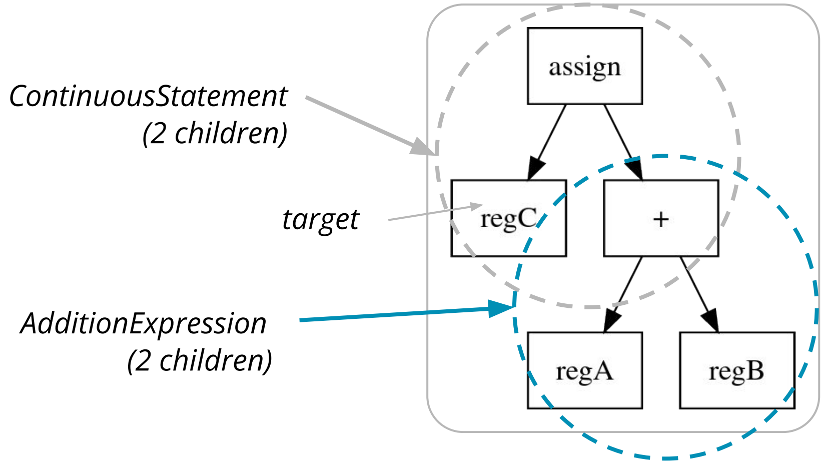


```
var a = (new RegisterDeclaration("regA", 8)).getReference();
var b = (new RegisterDeclaration("regB", 8)).getReference();
var c = (new RegisterDeclaration("regC", 8)).getReference();

var r = new ContinuousStatement(c, new AdditionExpression(a, b));
r.emit();
```

emit

```
assign regC = regA + regB;
```

Internal tree structure


ContinuousStatement (2 children)
target
AdditionExpression (2 children)
assign
regC
+
regA
regB

(Some) Crispy API Syntax

```
@Test
public void workshopExample4() {

    // create a class which inherits the
    // syntax (wrapper methods)
    var ex = new HardwareModule("example");

    // creates a "WireDeclaration", but returns
    // a "Wire", which is a reference to the
    // declared name (same for Registers and Ports)
    var wire = ex.addWire("ex1", 8);

    // new port
    var a = ex.addInputPort("pA", 8);

    // create an assign at the level of the module body
    ex.assign(a, wire.lsl(2));

    // emit to stdout (eventually, to files)
    ex.emit();
}
```

- There is also syntax for
  - *if*
  - *if-else*
  - *always ff*
  - *always comb*
  - *initial*
  - etc…
- Some (early) handling of
  - Sanity checks
  - Automatic wire generation

```
module example(pA);

        // Declarations block: Ports
        input wire [7 : 0] pA;

        // Declarations block: Wires
        wire [7 : 0] ex1;

        assign pA = ex1 << 8'd2;
endmodule //example
```

Programmatic Module Generation

```java
public void workshopExample2() {
    var adder = new HardwareModule("testAdder");
    var a = adder.addInputPort("testA", 32);
    var b = adder.addInputPort("testB", 32);
    var c = adder.addOutputPort("testC", 32);
    adder.alwayscomb()._do(c.nonBlocking(a.add(b)));

    adder.emit();
}
```

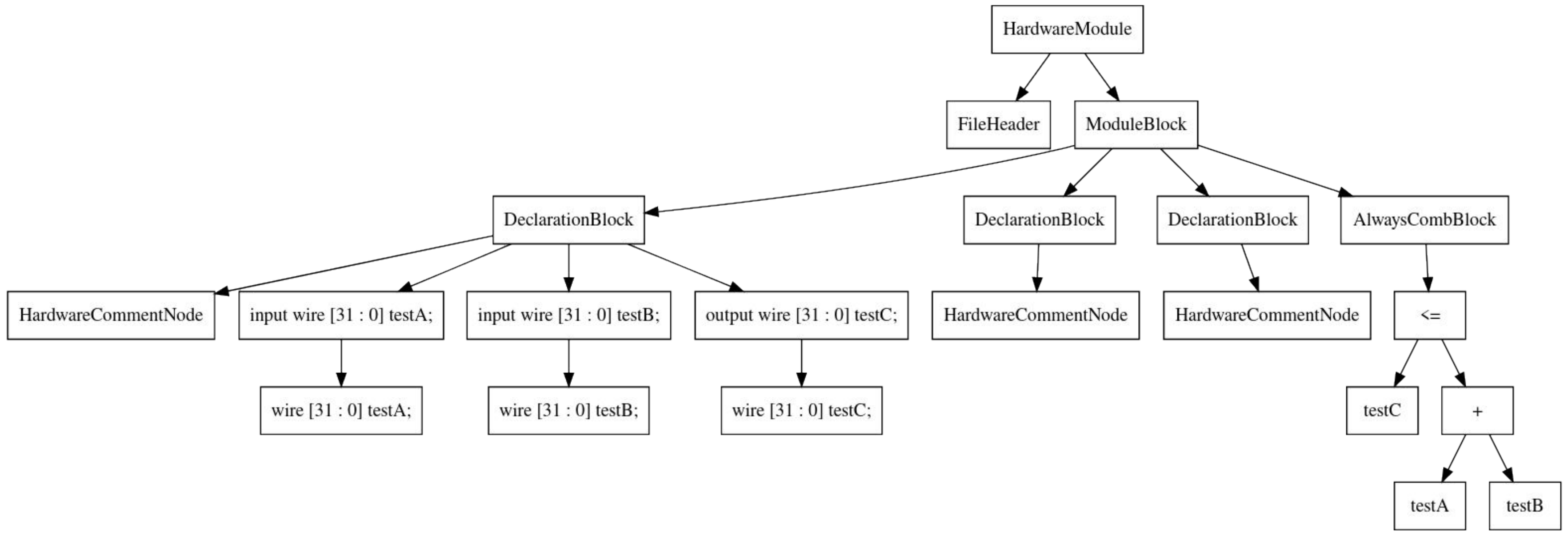
→

```verilog
module testAdder(testA, testB, testC);

     // Declarations block: Ports
     input wire [31 : 0] testA;
     input wire [31 : 0] testB;
     output wire [31 : 0] testC;

     always_comb begin : comb_0
          testC <= testA + testB;
     end
endmodule //testAdder
```

- Declaring a generic module, and then adding blocks, instances, and statements to it
  - Allows for **arbitrary** module generation integrated into other flows
  - But not clear if this capability is good or bad, in terms of language use/design...
    - Any “HardwareModule” is never finalized (i.e., made immutable) → unless the **class is made abstract (?)**

Programmatic Module Generation

The tree structure of the previous example



Explicit Module Generation via Extension

```
public class Mux2to1 extends HardwareModule {
    public InputPort i0;
    public InputPort i1;
    public InputPort sel;
    public OutputPort out;

    public Mux2to1(int bitwidth) {
        super(Mux2to1.class.getSimpleName());

        i0 = addInputPort("i0", bitwidth);
        i1 = addInputPort("i1", bitwidth);
        sel = addInputPort("sel", 1);
        out = addOutputPort("out", bitwidth);

        alwayscomb("muxBlock")._ifelse(sel.not())
                .then()._do(out.nonBlocking(i0))
                .orElse()._do(out.nonBlocking(i1));
    }
}
```

Inherits all sugar and sanity checking methods (i.e., “defines” the syntax within this class

Public members allow easier syntactic access to ports

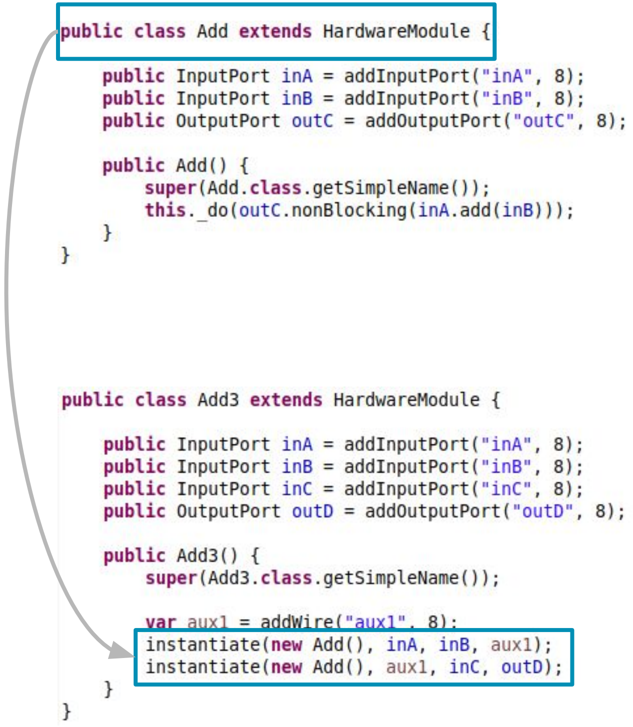
Some repetition is required when ports depend on constructor arguments… *how to avoid?*

- This makes Crispy more similar to Chisel or SpinalHDL, but is it the best way?

Module Instantiation

- Still needs a significant amount of work!

- Difficult to:
  - Keep track of instances
  - Define the proper abstractions



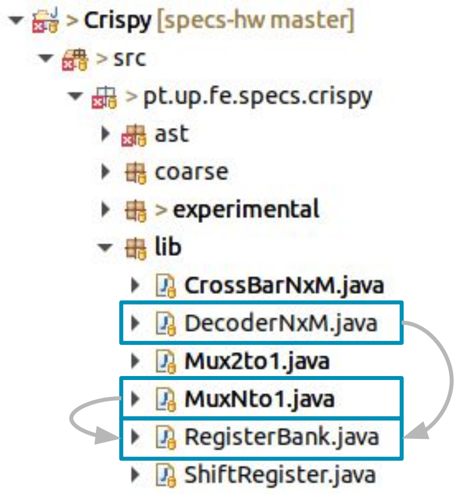
```java
public class Add extends HardwareModule {

    public InputPort inA = addInputPort("inA", 8);
    public InputPort inB = addInputPort("inB", 8);
    public OutputPort outC = addOutputPort("outC", 8);

    public Add() {
        super(Add.class.getSimpleName());
        this._do(outC.nonBlocking(inA.add(inB)));
    }
}
```

```java
public class Add3 extends HardwareModule {

    public InputPort inA = addInputPort("inA", 8);
    public InputPort inB = addInputPort("inB", 8);
    public InputPort inC = addInputPort("inC", 8);
    public OutputPort outD = addOutputPort("outD", 8);

    public Add3() {
        super(Add3.class.getSimpleName());

        var aux1 = addWire("aux1", 8);
        instantiate(new Add(), inA, inB, aux1);
        instantiate(new Add(), aux1, inC, outD);
    }
}
```

```verilog
module Add3(inA, inB, inC, outD);

        // Declarations block: Ports
        input wire [7 : 0] inA;
        input wire [7 : 0] inB;
        input wire [7 : 0] inC;
        output wire [7 : 0] outD;

        // Declarations block: Wires
        wire [7 : 0] aux1;

        Add Add_1926 (
         .inA(inA),
         .inB(inB),
         .outC(aux1)
        );

        Add Add_1555 (
         .inA(aux1),
         .inB(inC),
         .outC(outD)
        );

endmodule //Add3
```

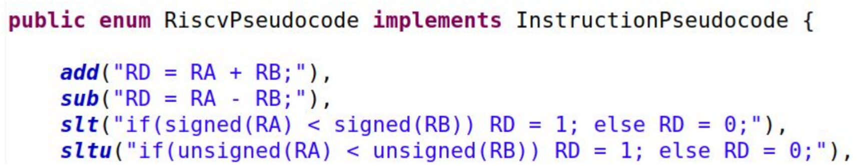
Future library of building blocks (?)

- Writing a register bank of arbitrary bit-width and size
    - +/- 5 minutes
    - Validated manually in Vivado simulation
- Future library blocks
    - AXI interfaces?
    - Buses?
    - Caches?
    - Floating point units?



Current Application

- Master’s Thesis
  - *Generating Hardware Modules via Binary Translation of RISC-V Binaries*
    - Translation of RISC-V instruction sequences into Verilog (via BTF + CrispyHDL)



```
public enum RiscvPseudocode implements InstructionPseudocode {

    add("RD = RA + RB;"),
    sub("RD = RA - RB;"),
    slt("if(signed(RA) < signed(RB)) RD = 1; else RD = 0;"),
    sltu("if(unsigned(RA) < unsigned(RB)) RD = 1; else RD = 0;"),
    etc...
```

$\longrightarrow$ *Programmatically Generated Modules*

- Reimplementing the Loop Accelerator (from IEEE TLVSI 2019 paper)
  - Easier/faster generation of architecture parameters
  - Integrated with loop extraction and modulo-scheduling
  - Future (partially implemented) integration with synthesis tools, reports, etc
    - e.g. via generation of TCL scripts for Vivado

Future Direction

- Better abstraction and syntax
  - Variable names via reflection?
  - Better state keeping for module instantiation?
- Base for CGRA Architecture Exploration
  - Design space exploration of CGRA variations
  - Joint software / hardware compilation
- External DSL
  - Tentative name: *CrunchyDSL*
  - A dedicated parser for Crunchy to translate to internal Crispy nodes
  - Avoids limitations of having Crispy implemented over Java
  - Allows for context specific rules for the language