Chisualizer

alpha

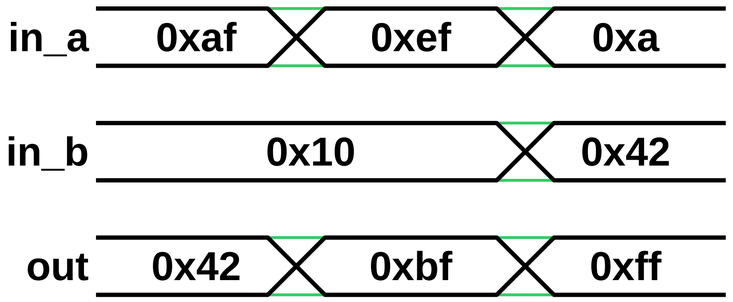
a block-diagram style visualizer for Chisel RTL

(names here)

Introduction

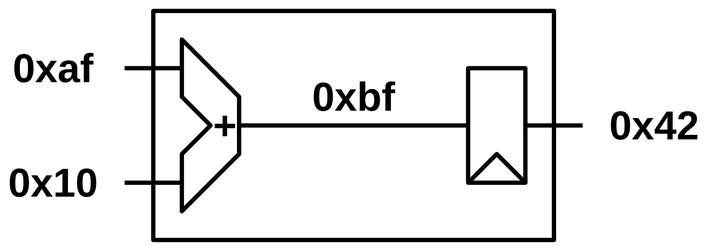
Why block-diagram visualizers?

- Current state-of-the-art for circuit debugging: **waveform viewers**
- Exposes temporal relations between signals, but not spatial relationships
- Less effective at visualizing multi-bit signals, or even just lots of signals
- … but both are essential for large designs



"wtf?"

vs.



"ohhhhh, shiny..." [1]

Why Chisualizer?

- Block-diagram-style visualizers aren't new
  - DREAMER even has one! (go check it out)
- But there's no standardization or common frameworks – **everyone rolls their own**
- … which really isn't a good idea:
  - High barrier to entry: build from scratch
  - HW engineers don't want to write OpenGL (or layout routines, or drawing commands)
  - Brittle printf-based visualizers
- It doesn't have to be that way
  - Write a generalized RTL visualization tool!

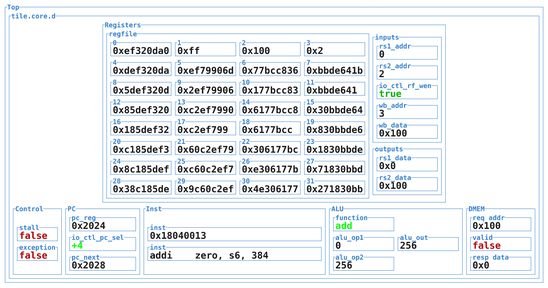
Overview

Chisualzier Overview

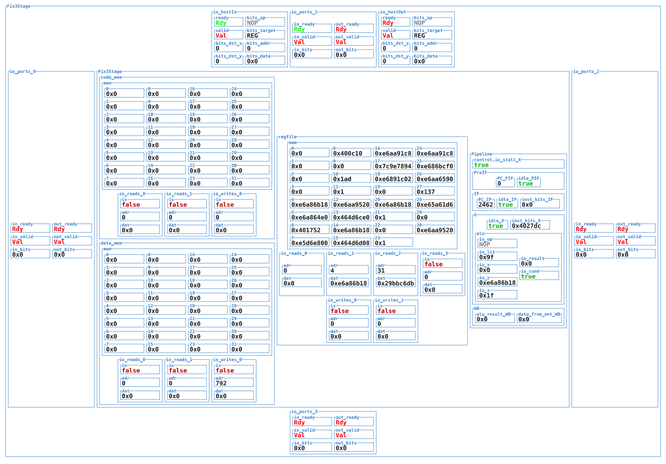
- Goal: allow elegant, intuitive visualizations while requiring the least amount of work
- Write visualization descriptors at a high level without worrying about minutia like element absolute positioning or sizing
  - … but also allow users to specify display details where necessary
- Programmed in a high level language (Python) using high level graphical primitives (Cairo) to allow easy extension by users
  - Stock visualizers can't cover every case
  - Cross-platform: not everyone uses $(YOUR_OS)
- Interactive circuit debugging: control circuit and elements from within GUI

Examples

RISC-V 1-stage



DREAMER 3-stage tile

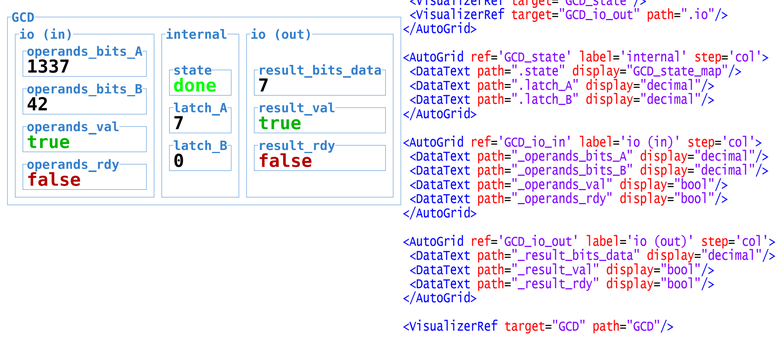


Using Chisualizer

Specifying Visualizers

- Written in XML: widely-understood tree language, detailed enough to learn by pattern-matching
- Main idea: composable, automatically sized and placed visualizers
- Composable: visualizers specified hierarchically, allowing re-use and modularity
- Automatically sized: determine absolute positioning by user-specified relative layout, infer sizing by data type and width

Hello, world: GCD




```
<MappedDisplay ref="GCD_state_map">
 <Mapping key="0x0" text="idle" color="pink"/>
 <Mapping key="0x1" text="calc" color="yellow"/>
 <Mapping key="0x2" text="done" color="green"/>
</MappedDisplay>-

<AutoGrid ref='GCD' step='row'>
 <VisualizerRef target="GCD_io_in" path=".io"/>
 <VisualizerRef target="GCD_state"/>
 <VisualizerRef target="GCD_io_out" path=".io"/>
</AutoGrid>

<AutoGrid ref='GCD_state' label='internal' step='col'>
 <DataText path=".state" display="GCD_state_map"/>
 <DataText path=".latch_A" display="decimal"/>
 <DataText path=".latch_B" display="decimal"/>
</AutoGrid>

<AutoGrid ref='GCD_io_in' label='io (in)' step='col'>
 <DataText path="_operands_bits_A" display="decimal"/>
 <DataText path="_operands_bits_B" display="decimal"/>
 <DataText path="_operands_val" display="bool"/>
 <DataText path="_operands_rdy" display="bool"/>
</AutoGrid>

<AutoGrid ref='GCD_io_out' label='io (out)' step='col'>
 <DataText path="_result_bits_data" display="decimal"/>
 <DataText path="_result_val" display="bool"/>
 <DataText path="_result_rdy" display="bool"/>
</AutoGrid>

<VisualizerRef target="GCD" path="GCD"/>
```

Visualization Descriptor

Continuing Work

- More features:
  - Element highlighting (for memory access)
  - Tester integration, replay functionality
  - Pipe diagram and time-based visualizations
  - Additional data types (like images)
  - Eye candy: symbols and wires
- Documentation for release
- Web-based (or server-client) visualizer
- Scala frontend for descriptor generation

[1] ok, current version isn't quite that nice – *yet*