P.S. Make sure you have access to the PLDI discord!

calyxir.org/tutorial

---

**DSL-Based Hardware Generation**

*At PLDI 2023 in Orlando*

When: Sunday June 18, 2023; morning session  
Where: Magnolia 9, with a virtual option (see below)

In the last four decades, the easiest way to improve performance of programs has been to simply wait; processor and process scaling took care of the rest. Sadly, Moore's law is over, there are no free lunches left, and everyone at the table is desperate. The only way forward is to build specialized hardware accelerators, that can be customized to the needs of the application.

So how, then, does an enterprising performance hound like yourself build an accelerator?
• Install Docker
• Get the Calyx Image
• Run sanity checks

(These will take some time!)
Hardware description languages (HDLs)

- VHDL
- SystemVerilog
- CHISEL
- PyMTL

```plaintext
assign a = b + c
```

add integers
Hardware description languages (HDLs)

multiply integers

```verilog
module seq_mult (p, rdy, clk, reset, a, b);
  input clk, reset;
  input [7:0] a, b;
  output [15:0] p;
  output rdy;

  reg [15:0] p;
  reg [15:0] multiplier;
  reg [15:0] multiplicand;
  reg rdy;
  reg [4:0] ctr;

always @(posedge clk or posedge reset) begin
  if (reset)
    begin
      rdy <= 0;
      p <= 0;
      ctr <= 0;
      multiplier <= {#(a[7]), a};
      multiplicand <= {#(b[7]), b};
    end
  else begin
    if (ctr < 16)
      begin
        if (multiplier[ctr]==1)
          begin
            multiplicand = multiplicand<<ctr;
            p <= p + multiplicand;
          end
        ctr <= ctr+1;
      end
    else begin
      rdy <= 1;
    end
  end
endmodule
```
Hardware description languages (HDLs)

VHDL

Berkeley HardFloat Release 1: Verilog Modules

John R. Hauser
2019 July 29

Contents

1. Introduction
2. Limitations
3. Acknowledgments and License
5. Floating-Point Representations
   5.1 Standard Formats
   5.2 Recorded Formats
   5.3 Raw Reconstructions
6. Common Control and Mode Inputs
   6.1 Control Input
   6.2 Rounding Mode
7. Exception Results
8. Specialization
   8.1 Width and Default Value for the Control Input
   8.2 Integer Results on Exceptions
9. Main Modules
   9.1 Conversions Between Standard and Recorded-Floating-Point (tockenFR, xfrFRtoken)
   9.2 Conversions from Integer (toIntegerFR, xfrFRtoInteger)
   9.3 Conversions to Integer (xfrFRtoInteger)
   9.4 Conversions Between Formats (xfrFRtoXfrFR)
   9.5 Addition and Subtraction (addFRaddFRtoken, addFRaddFR)
   9.6 Multiplication (mulFR, mulXFRtoXFR, mulFRtoFR)
   9.7 Panel Multiply-Add (mulPanelFR, mulPanelXFRtoXFR)
   9.8 Division and Square Root (divFR, xfrFRtoXFRtoXFR_small, divFRtoXFRtoXFR_small)
9.9 Comparisons (compareFR)
10. Common Submodules

add floating-point numbers
Hardware description languages (HDLs)

- **VHDL**
- **SystemVerilog**
- **CHISEL**
- **PyMTL**

Berkeley HardFloat Release 1: Verilog Modules

**Contents**

1. Introduction
2. Limitations
3. Acknowledgments and License
5. Floating-Point Representations
   5.1. Standard Forms
   5.2. Recorded Formats
   5.3. Raw Reconstructions
6. Common Control and Mode Inputs
   6.1. Control Input
   6.2. Rounding Mode
7. Exception Results
8. Specialization
   8.1. Width and Default Value for the Control Input
   8.2. NaN Results on Exceptions
9. Main Modules
   9.1. Conversions Between Standard and Recorded Floating-Point
   9.2. Conversions from Integer
   9.3. Conversions to Integer
   9.4. Conversions Between Formats
   9.5. Additions and Subtractions
   9.6. Multiplications
   9.7. Division and Square Root
   9.8. Comparisons
9.9. Common Submodules

Additional components:

- **add and multiply**
- **several floating-point numbers**
Encoding control flow

Breaking up critical paths

Optimizing resource usage
Captures control flow

Missing structure and time

Captures structure and time

Missing control flow
High-level control flow  Low-level structure
```cpp
int a, b, c, d;
if (a % 10)
    x = a * b
else
    y = c * d
```
Components encapsulate hardware structure and control flow

```cpp
int a, b, c, d;
if (a % 10)
    x = a * b
else
    y = c * d
```
```c
int a, b, c, d;
if (a % 10)
    x = a * b
else
    y = c * d
```
int a, b, c, d;
if (a % 10)
    x = a * b
else
    y = c * d

component do_add(a: 32, ...) -> (out: 32) {
    cells {
        mod = std_mod(32);
        cond = std_reg(1);
    }
    wires {};
    control {};
}

Cells define **sub-components** required by a given component.
```
int a, b, c, d;
if (a % 10)
  x = a * b
else
  y = c * d
```

```cpp
component do_add(a: 32, ...) -> (out: 32) {
  cells { ... } 
  wires {
    mod.right = 32'd10;
    mod.left = count;
    cond.in = mod.out;
    cond.write_en = 1'd1;
  }
  control {}
}
```

Wires defines **connections** between submodules
component do_add(a: 32, ...) -> (out: 32) {
    cells {
        mod = std_mod(32);
        cond = std_reg(1);
    }
    wires {
        mod.right = 32'd10;
        mod.left = count;
        cond.in = mod.out;
        cond.write_en = 1'd1;
    }
    control {}
component do_add(a: 32, ...) -> (out: 32) {
  cells {
    mod = std_mod(32);
    cond = std_reg(1);
  }
  wires {
    mod.right = 32'd10;
    mod.left = count;
    cond.in = mod.out;
    cond.write_en = 1'd1;
  }
  control {}
}
component do_add(a: 32, ...) -> (out: 32) {
  cells { ... }
  wires {
    group do_cond {
      mod.right = 32’d10;
      mod.left = count;
      cond.in = mod.out;
      cond.write_en = 1’d1;
      do_cond[done] = cond.done;
    }
  }
  control {}
}
int a, b, c, d;
if (a % 10)
    x = a * b
else
    y = c * d

Groups describe the “instructions” for the accelerator.
```cpp
int a, b, c, d;
if (a % 10)
    x = a * b
else
    y = c * d

Control defines the execution schedule

component do_add(a: 32, ...) -> (out: 32) {
    cells { ... }
    wires {
        group do_cond {
            mod.right = 32’d10;
            mod.left = a;
            cond.in = mod.out;
            cond.write_en = 1’d1;
            do_cond[done] = cond.done;
        }
    }
    control {
        seq { do_cond; do_cond; do_cond }
    }
}
```
```cpp
int a, b, c, d;
if (a % 10)
   x = a * b
else
   y = c * d
```
int a, b, c, d;
if (a % 10)
    x = a * b
else
    y = c * d

component do_add(a: 32, ...) -> (out: 32) {
    cells { ... }
    wires {
        group do_cond { ... }
        group upd_x { ... }
        group upd_y { ... }
    }
    control {
        seq {
            do_cond;
            if cond.out { upd_x } else { upd_y }
        }
    }
}
```c
int a, b, c, d;
if (a % 10)
    x = a * b
else
    y = c * d

• Generate **groups** to encode computation

• Use **control** to schedule execution

```
LLVM
CIRCT

TVM

Dahlia

Systolic Arrays

And others …

Calyx

- Modular, pass-based compiler
- 40 optimization passes
- 15 analyses

Software-like Debugging
ASPLOS ‘23

FPGA Execution
Automatic AXI Generation

Pangenomics Accelerator
Write your first Calyx program!

P.S. Remember to install, setup your favorite editor

docs.calyxir.org
fud, the Calyx driver

```python
import "primitives/core.futil";
import "primitives/binary_operators.futil";
component main() → () {
  cells {
    @external avec_b0 = std_mem_d1(32, 2, 32);
    @external avec_b1 = std_mem_d1(32, 2, 32);
    @external squares_b0 = std_mem_d1(32, 2, 32);
    @external squares_b1 = std_mem_d1(32, 2, 32);
    idx_b0_0 = std_reg(32);
    incr_b0_0 = std_add(32);
    lt_b0_0 = std_lt(32);
    mul_b0_0 = std_mult_pipe(32);
    idx_b1_0 = std_reg(32);
    incr_b1_0 = std_add(32);
    lt_b1_0 = std_lt(32);
    mul_b1_0 = std_mult_pipe(32);
  }
  wires {
    group incr_idx_b0_0 {
      incr_b0_0.left = idx_b0_0.out;
      incr_b0_0.right = 32'd1;
      idx_b0_0.in = incr_b0_0.out;
      idx_b0_0.write_en = 1'd1;
      incr_idx_b0_0[done] = idx_b0_0.done;
    }
    comb group cond_b0_0 {
      lt_b0_0.left = idx_b0_0.out;
    }
  }
}
fud, the Calyx driver

```verilog
code = $value$plusargs("DATA=%s", DATA);
$display("DATA (path to meminit files): %s", DATA);
$readmemh({DATA, "/avec_b0.dat"}, avec_b0.mem);
$readmemh({DATA, "/avec_b1.dat"}, avec_b1.mem);
$readmemh({DATA, "/squares_b0.dat"}, squares_b0.mem);
$readmemh({DATA, "/squares_b1.dat"}, squares_b1.mem);
end
final begin
$writememh({DATA, "/avec_b0.out"}, avec_b0.mem);
$writememh({DATA, "/avec_b1.out"}, avec_b1.mem);
$writememh({DATA, "/squares_b0.out"}, squares_b0.mem);
$writememh({DATA, "/squares_b1.out"}, squares_b1.mem);
end
logic [31:0] avec_b0_addr;
logic [31:0] avec_b0_write_data;
logic avec_b0_write_en;
logic avec_b0_clk;
```
FUD will perform the following steps:
- `mrxl.mktmp`: Make temporary directory to store Verilator build files.
- `mrxl.set_mrxl_prog`: Set stages.mrxl.prog as `input`.
- `mrxl.mrxl-data.get_mrxl_prog`: Dynamically retrieve the value of stages.mrxl.prog
- `mrxl.mrxl-data.convert_mrxl_data_to_calyx_data`: Converts MrXL input into calyx input
- `transform`: transform input to String
- `mrxl.save_data`: Save verilog.data in `tmpdir` and update stages.verilog.data
- `mrxl.run_mrxl`: mrxl
  - `transform`: transform input to Path
  - `verilog.mktmp`: Make temporary directory to store Verilator build files.
  - `verilog.get_verilog_data`: Dynamically retrieve the value of stages.verilog.data
  - `verilog.check_verilog_for_mem_read`: Read input verilog to see if `verilog.data` needs to be set.
  - `verilog.json_to_dat`: Converts a `json` data format into a series of `.dat` files inside the given temporary directory.
  - `verilog.compile_with_verilator`: verilator --trace {input_path} /Users/ps/Research/calyx-ref/calyx/fud/icarus/tb.sv --binary --top-module TOP --Mdir {tmpdir_name} -fno-inline
  - `verilogsimulate`: Simulates compiled Verilator code.
  - `verilog.output_json`: Convert .dat files back into a json and extract simulated cycles from log.
  - `verilog.cleanup`: Cleanup Verilator build files that we no longer need.
fud e squares.mr
--through verilo
to dat \ squaures.mrxl.data
MrXL: map-reduce accelerator

Real frontends are not built in a day, so let’s work on a toy frontend for Calyx.

We will introduce MrXL, and then you will implement its map operation.

Watch your directories! The initial commands are run from calyx/
Eventually you will work in calyx/frontends/mrxl
anatomy of a frontend

- program.mrxml
- parser.py
- program.ast
- calyx_gen.py
- program.futil

wrapped up using fud
The **Calyx Builder**

Python library to build Calyx programs

---

```python
# import the builder library
import calyx.builder as cb

# define `second_comp`
def add_second_comp(prog):
    # `second_comp` definition here

# method for defining `my_component` and adding it to
def add_my_component(prog, second_comp):
    # add the component to the program
    my_component = prog.component("my_component")

    # Adding an instance of `second_comp` as a cell of
    my_second_compo = my_component.cell("my_second_compo")
```
memory banking

Reads from a memory have a limitation: one read/write per tick of the clock.

Say I give you all the compute you could ask for:

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

But if we *bank* the arrays, we really can parallelize:

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>a+1</th>
<th>b+1</th>
<th>c+1</th>
<th>d+1</th>
<th>e+1</th>
<th>f+1</th>
<th>g+1</th>
<th>h+1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>a+1</th>
<th>b+1</th>
<th>c+1</th>
<th>d+1</th>
<th>e+1</th>
<th>f+1</th>
<th>g+1</th>
<th>h+1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
give MrXL a map operation!
Psst: consider implementing just `add` first, end-to-end

```bash
fud e --from mrxl test/sos.mrxl \
  --to dat --through verilog \ 
  -s mrxl.flags "--my-map " \ 
  -s mrxl.data data test/sos.mrxl.data
```

pass **this flag** to run your version
study the implementation that’s in place
Cider

Frontend (MrXL) → Calyx → Verilog

[Diagram showing a process flow from Frontend (MrXL) through Calyx to Verilog.]
Cider: **Calyx Interpreter and Debugger**

Provides a **GDB-like debugging experience** for Calyx programs.
Insight: Use Calyx **Groups** as coarse time units

```python
for i in 0..4:
z[i] = x[i] * y[i]
```

```python
read_mem
do_mul
write_mem
upd_idx
```
for i in 0..4:
    z[i] = x[i] * y[i]

Observed behavior: does not terminate

> watch after upd_idx with print-state \u idx_reg
    idx_reg = 1
    idx_reg = 2
    idx_reg = 3
WARN - Integer overflow, source: idx_adder
    idx_reg = 0
    idx_reg = 1
the bug

cells {
    idx_reg = reg(2);
    idx_adder = reg(2);
}

while idx_reg <= 3 {
    read_mem;
    do_mul;
    write_mem;
    upd_idx;
}

cells {
    idx_reg = reg(3);
    idx_adder = reg(3);
}
using Cider

fud e --from mrxl test/sos.mrxl --to interpreter-out

fud e --from mrxl test/sos.mrxl --to debugger

Read more: https://docs.calyxir.org/debug/cider.html
Pollen, an accelerator generator for pangenomic graph queries
Pollen, an accelerator generator for pangenomic graph queries
size of a human pangenomic graph:

259,525,394 base pairs in a chromosome
481,945 nodes per person
4,643,780 nodes per pangenomic graph
size of a human pangenomic graph:

259,525,394 base pairs in a chromosome
481,945 nodes per person
4,643,780 nodes per pangenomic graph

~30GB
size of a human pangenomic graph:

259,525,394 base pairs in a chromosome
481,945 nodes per person
4,643,780 nodes per pangenomic graph

~30GB
Larger for efficient computations
out_graph g;
parse def depth[int, g];
for segment in graph.segments {
    emit segment.steps.size() to depths;
}
out_graph g;
parset depth[int, g];
for segment in graph.segments {
    emit segment.steps.size() to depths;
}
out_graph g;
parset depth[int, g];
for segment in graph.segments {
    emit segment.steps.size() to depths;
}
out_graph g;
parset depth[int, g];
for segment in graph.segments {
    emit segment.steps.size() to depths;
}

exine depth depth.pollen -n 2
```plaintext
out_graph g;
parset depth[int, g];
for segment in graph.segments {
    emit segment.steps.size() to depths;
}
```

```
exine depth depth.pollen -n 2
```

```plaintext
output depths : int[4]
depths := map 2 (s <- graph.segments) {s.steps.size()}
```
takeaways

Domain experts with minimal hardware knowledge can make use of Calyx

Calyx can be a backend for complex DSLs

The skills you’ve gained generating map are broadly applicable to all sorts of language features
have a break, have a Kit Kat
MrXL: extensions

Three options:
1. Implement the reduce operation.
2. Allow the same memory to banked repeatedly.
3. Office hours!
How best to run a tennis tournament?
This clearly has problems…
reduction trees

A little better!
reduction trees

Better still…

But hang on, do we have four courts at the same time?
reduction trees

We can do this with two courts!
Ugh it’s getting expensive to transport players to and fro. Here’s another option…
This just in:
the players are numbers;
the matches are addition!

36
How would you handle 16 numbers?
reduction trees, feat. banking

Where would you place the 16 numbers?

\[
\text{mem} := \begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
13 & 14 & 15 & 16 \\
\end{array}
\]

\[
\text{m1} := \begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
13 & 14 & 15 & 16 \\
\end{array}
\]

\[
\text{m2} := \begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
13 & 14 & 15 & 16 \\
\end{array}
\]

\[
\text{m3} := \begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
13 & 14 & 15 & 16 \\
\end{array}
\]

\[
\text{m4} := \begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
13 & 14 & 15 & 16 \\
\end{array}
\]
What if I want to parallelize the same memory, but with different banking factors?

```plaintext
avec := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]
squares := map 3 (a <- avec) { a * a }
add_1 := map 2 (a <- avec) { a + 1 }
```

We need to break `avec` into \(\text{lcm}(2, 3)=6\) banks, and then arbitrate between those.
make MrXL better!
closing remarks