

Calyx Tutorial

PLDI 2023

<https://calyxir.org/tutorial>

Time	Topic
9-9:20am	Introduction to Calyx, and setting up
9:20-9:55am	Your first Calyx program
9:55-10am	fud , the hardware tool composer
10-10:10am	MrXL , a map-reduce frontend
10:10-10:50am	Implement a map operation for MrXL
10:50-10:55am	Cider , the Calyx interactive debugger
10:55-11am	Pollen , a pangenome analysis DSL
11-11:20am	Break
11:20am-12:20pm	Contest: extensions to MrXL!
12:20-12:30pm	Award ceremony and closing remarks

opening remarks

- Hardware is important and Verilog is bad... we can fix that
 - Compilers are cool (as opposed to hand-writing fixed-function code)
- Setting up: check out Docker container, run fud check, run runt

- Companies are investing in custom hardware
- Countries are investing in hardware design
- And mostly importantly, you can write papers about hardware

- But, using Verilog is a pain
 - Bad language
 - Bad tools
 - Complex constraints (name drop things like timing closure)

1. Install Docker
2. Get the Docker container
3. Run the following commands in the container
 - `fud check ...`
 - `runt -i ...`

- Calyx programs have components
 - Which have cells, wires, and control
 - Wires are structured using groups
 - Control operators schedule them
- [Link to language tutorial](#)

MrXL



MyDSL

Calix



fud, the Calyx driver

fud e s

```
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;
    }
  }
}
```

o calyx

fud, the Calyx driver

fud e sq

```
module main(  
    input logic go,  
    input logic clk,  
    input logic reset,  
    output logic done  
);  
// COMPONENT START: main  
string DATA;  
int CODE;  
initial begin  
    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_addr0;  
logic [31:0] avec_b0_write_data;
```

verilog

MrXL

Calx

SV

mem



```
fud e squares.mr  
--through verilo
```

```
{  
  "cycles": 17,  
  "memories": {  
    "avec_b0": [  
      0,  
      1  
    ],  
    "avec_b1": [  
      4,  
      5  
    ],  
    "squares_b0": [  
      0,  
      1  
    ],  
    "squares_b1": [  
      16,  
      25  
    ]  
  }  
}
```

```
to dat \  
squares.mrxl.data
```

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
- calyx.run_futil: /Users/ps/Research/calyx-ref/calyx/target/debug/calyx -l /Users/ps/Research/calyx-ref/calyx -b verilog
- `fud e squares.mrxl --from mrxl --to dat \`
- `--through verilog -s mrxl.data squares.mrxl.data -n`
- `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
- `verilog.simulate`: 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.mrxl --from mrxl --to dat \
```

```
fud e squares.mrxl --from mrxl --to interpreter-out \  
-s mrxl.data squares.mrxl.data
```

```
fud e squares.xclbin --from xclbin --to fpga \  
-s fpga.data squares.mrxl.data
```


MrXL: map-reduce accelerator

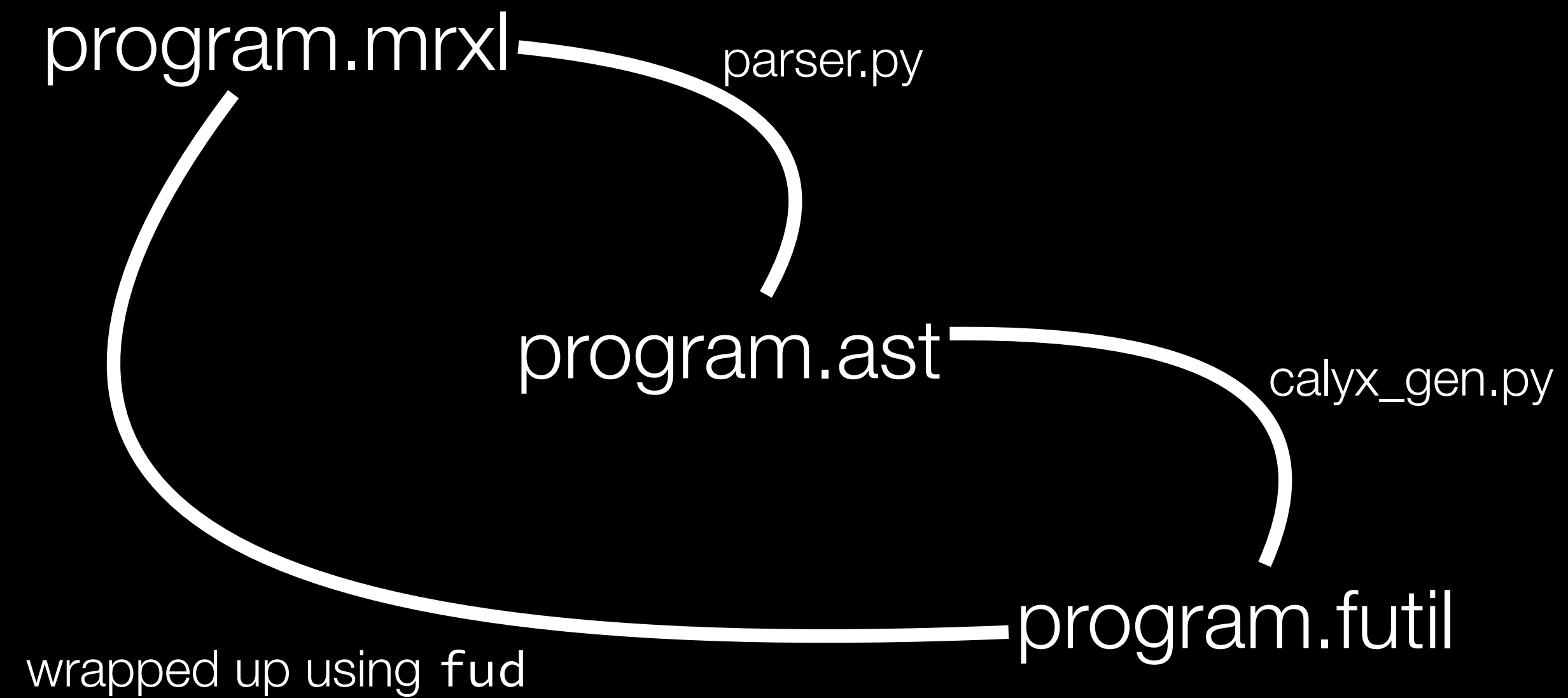
Real frontends are not built in a day,
so let's work on a toy frontend for Calyx.

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

Watch your directories!

Work in `frontends/mrxl`

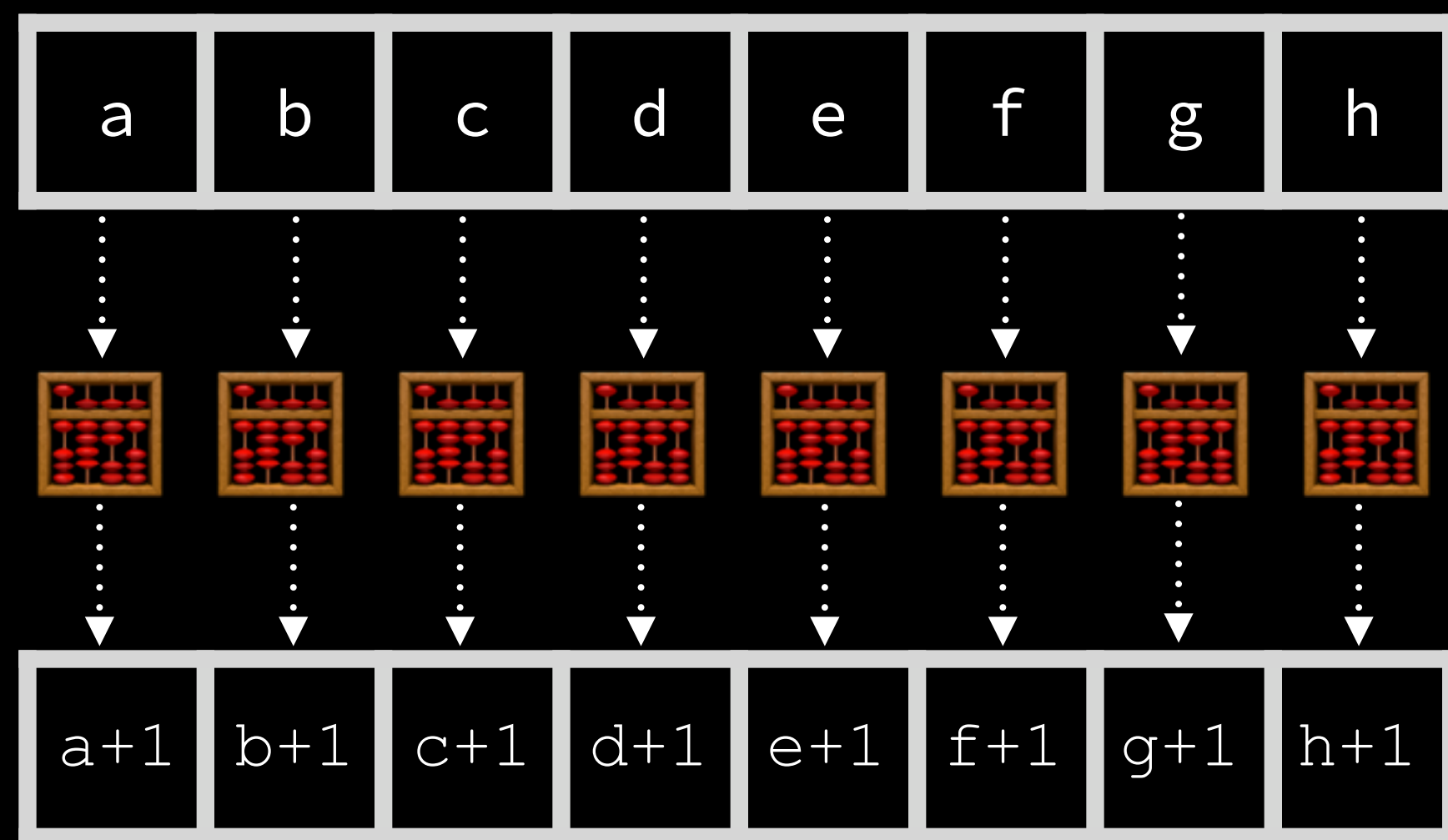
anatomy of a frontend



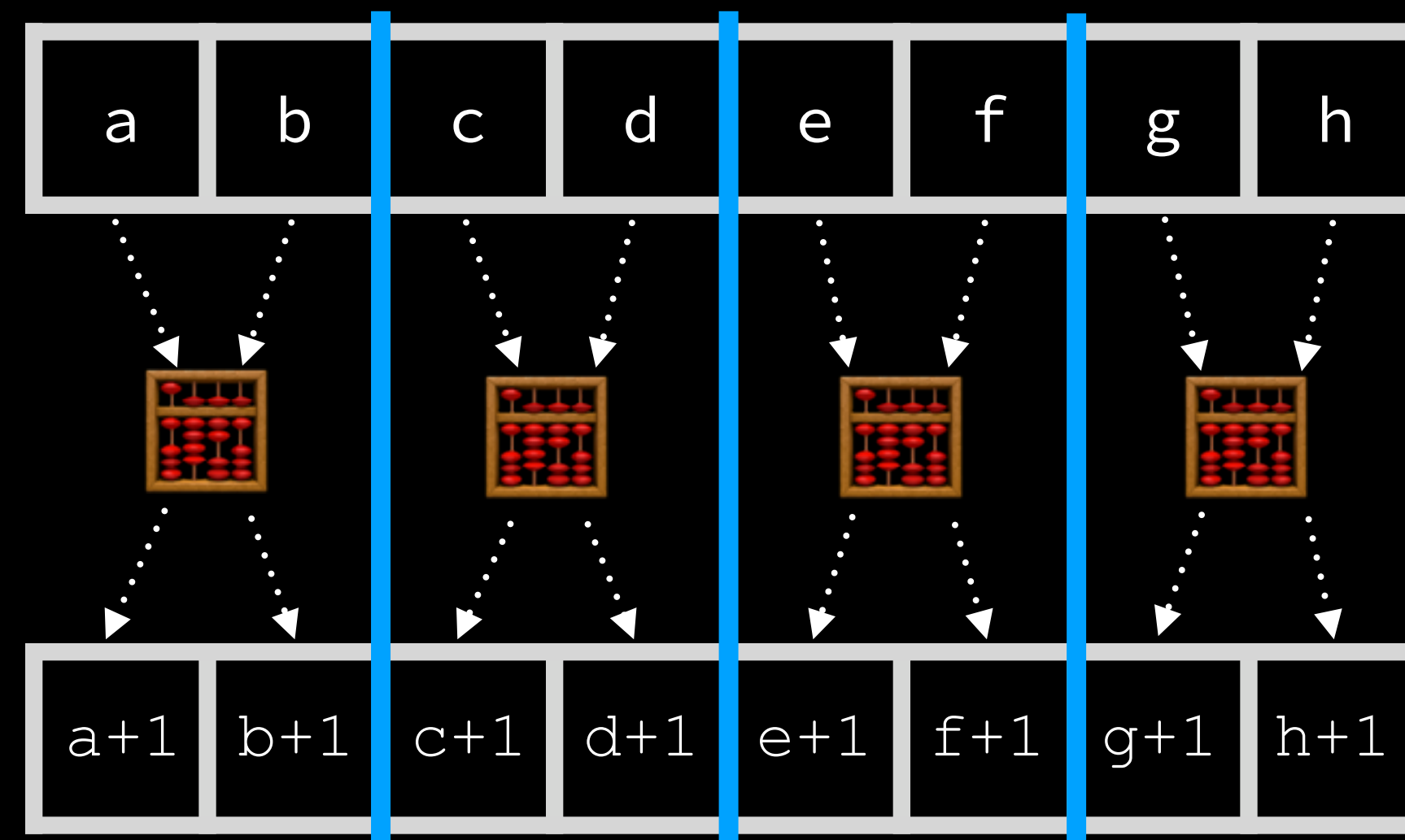
memory banking

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:



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



give MrXL a map operation!

Psst: consider implementing just add first, end-to-end

```
fud e --from mrxl test/sos.mrxl \  
      --to dat --through verilog \  
pass this flag! → -s mrxl.flags "--my-map " \  
                  -s mrxl.data test/sos.mrxl.data
```



study the implementation
that's in place

Cider



Cider: Calyx Interpreter and Debuggerr

Provides a GDB-like debugging experience for Calyx programs

Insight: Use Calyx **Groups** as coarse time units

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

```
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
```

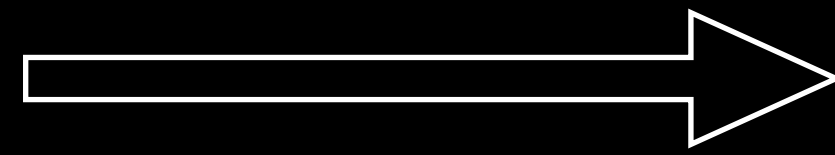
```
WARN - Integer overflow, source: idx_adder
```

```
    idx_reg = 0
```

```
    idx_reg = 1
```

the bug

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



```
cells {  
    idx_reg = reg(3);  
    idx_adder = reg(3);  
}
```

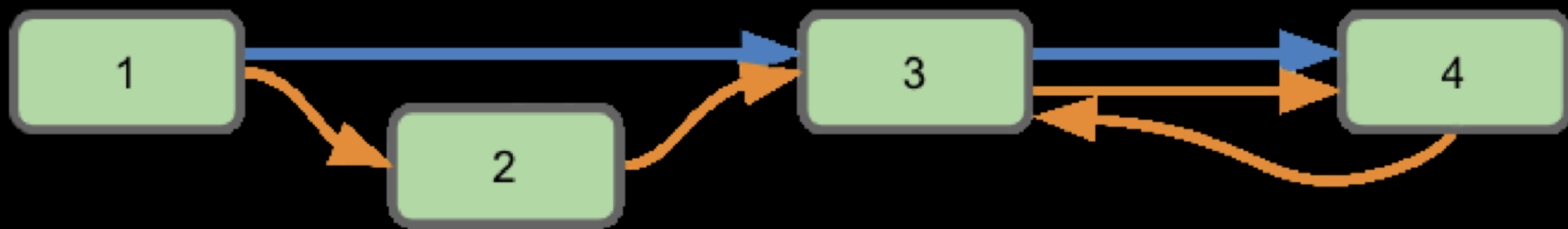
```
while idx_reg <= 3 {  
    read_mem;  
    do_mul;  
    write_mem;  
    upd_idx;  
}
```

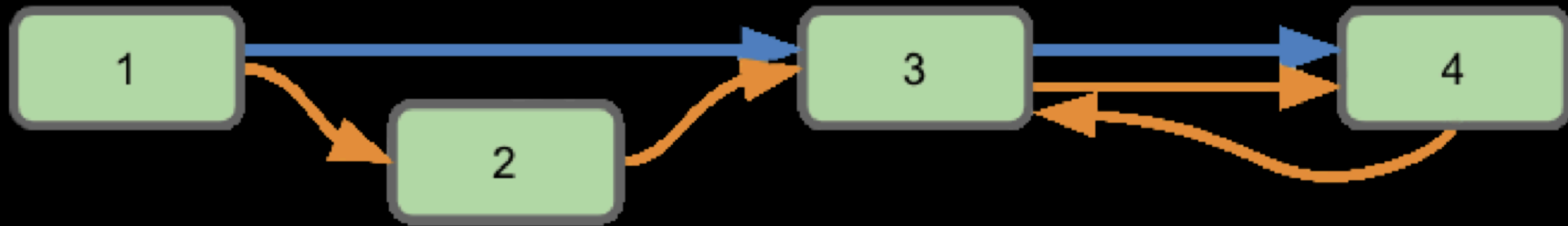

using Cider

```
fud e --from mrx1 test/sos.mrx1 --to interpreter-out
```

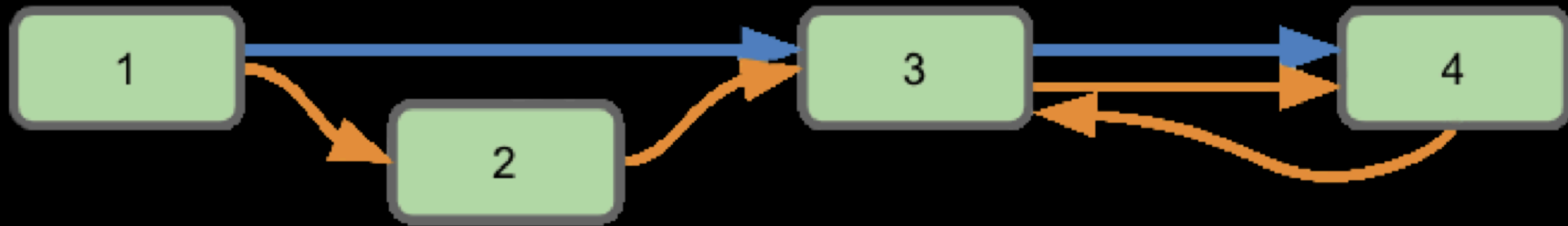
```
fud e --from mrx1 test/sos.mrx1 --to debugger
```

po~~o~~en





```
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 -n 2
```

=

```
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

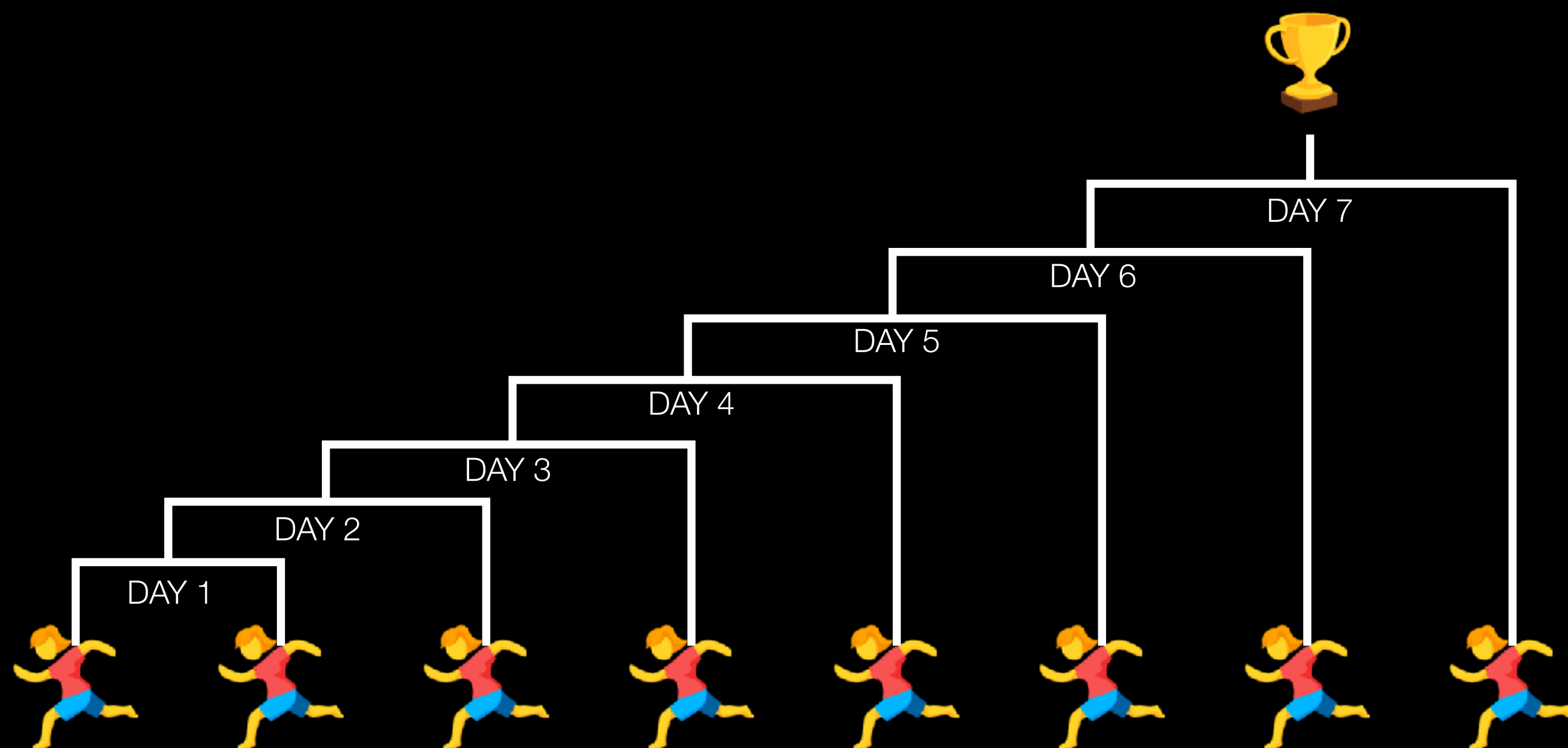
Three options:

1. Implement the reduce operation.
2. Allow the same memory to be banked repeatedly.
3. Office hours!

reduction trees

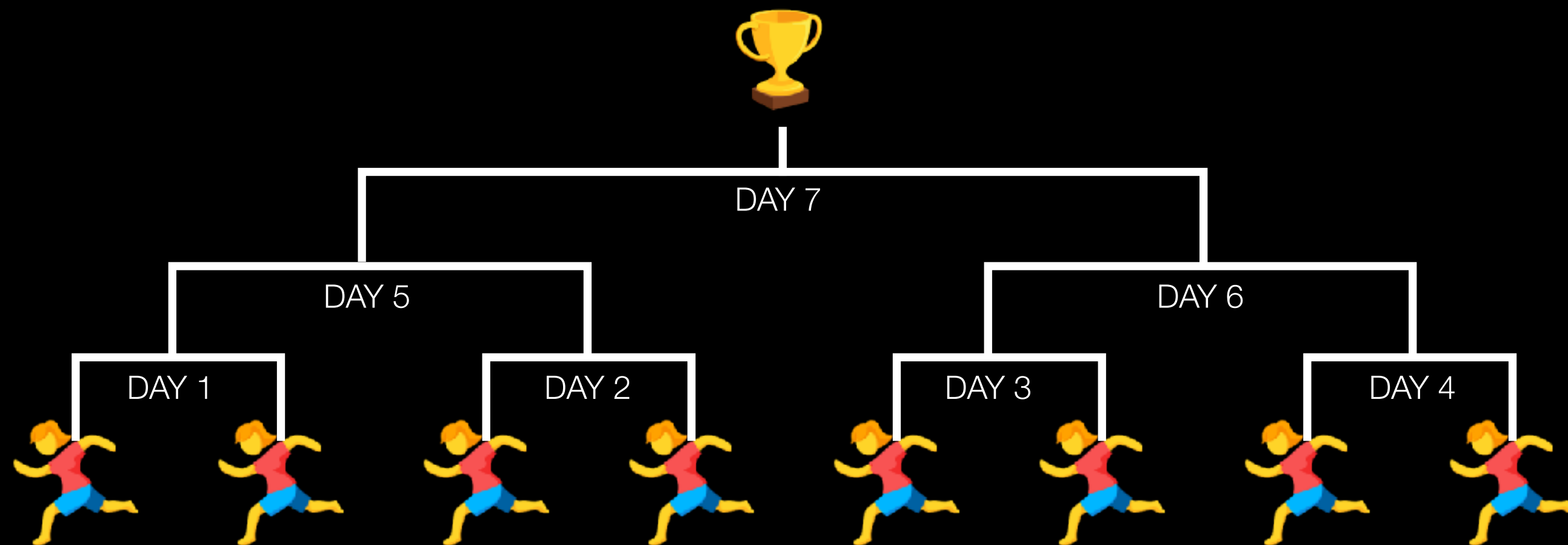
How best to run a tennis tournament?

This clearly has problems...



reduction trees

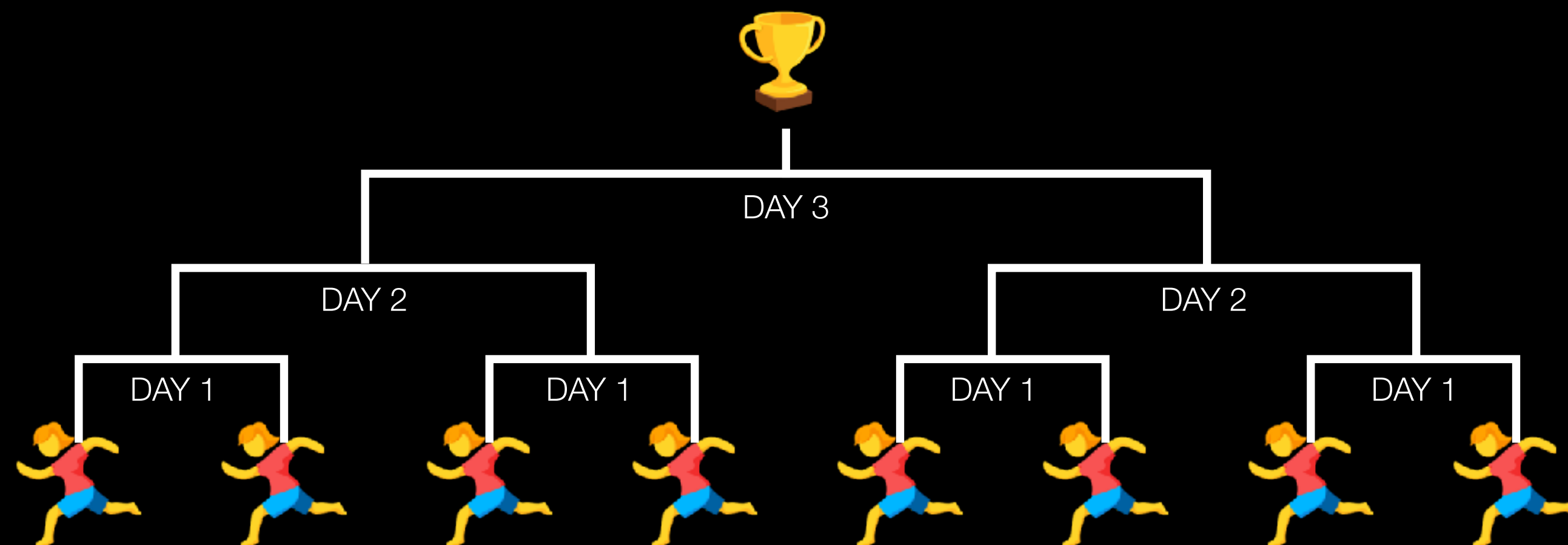
A little better!



reduction trees

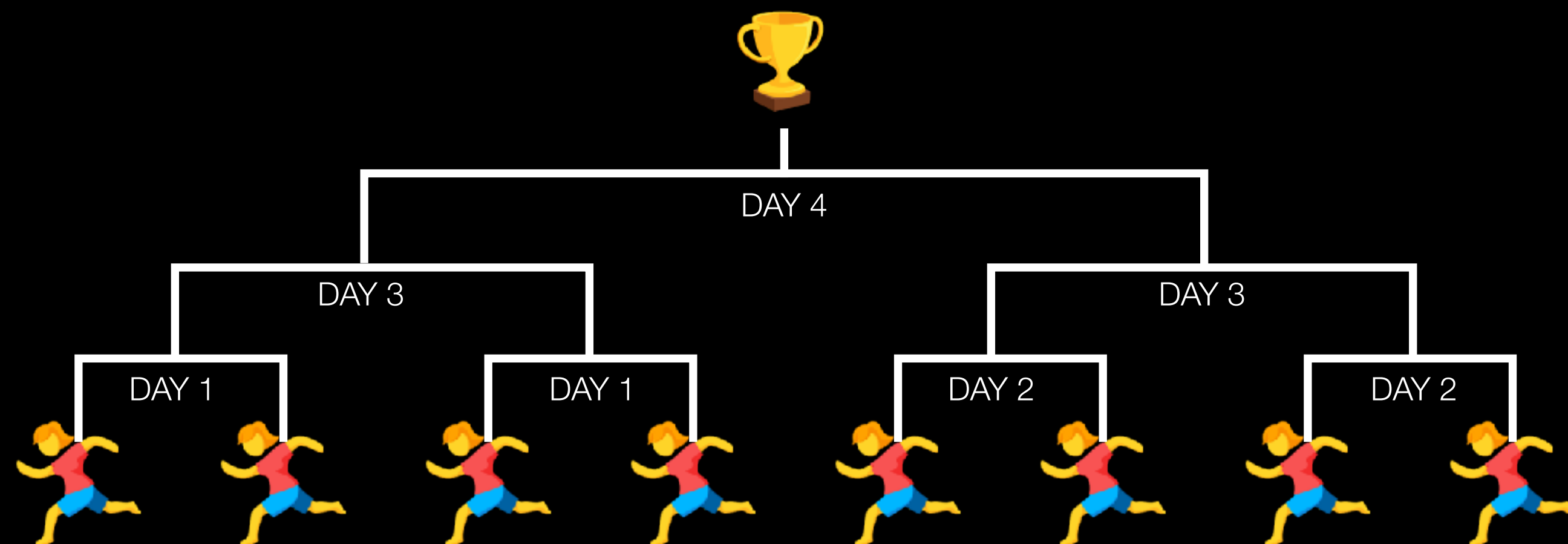
Better still...

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



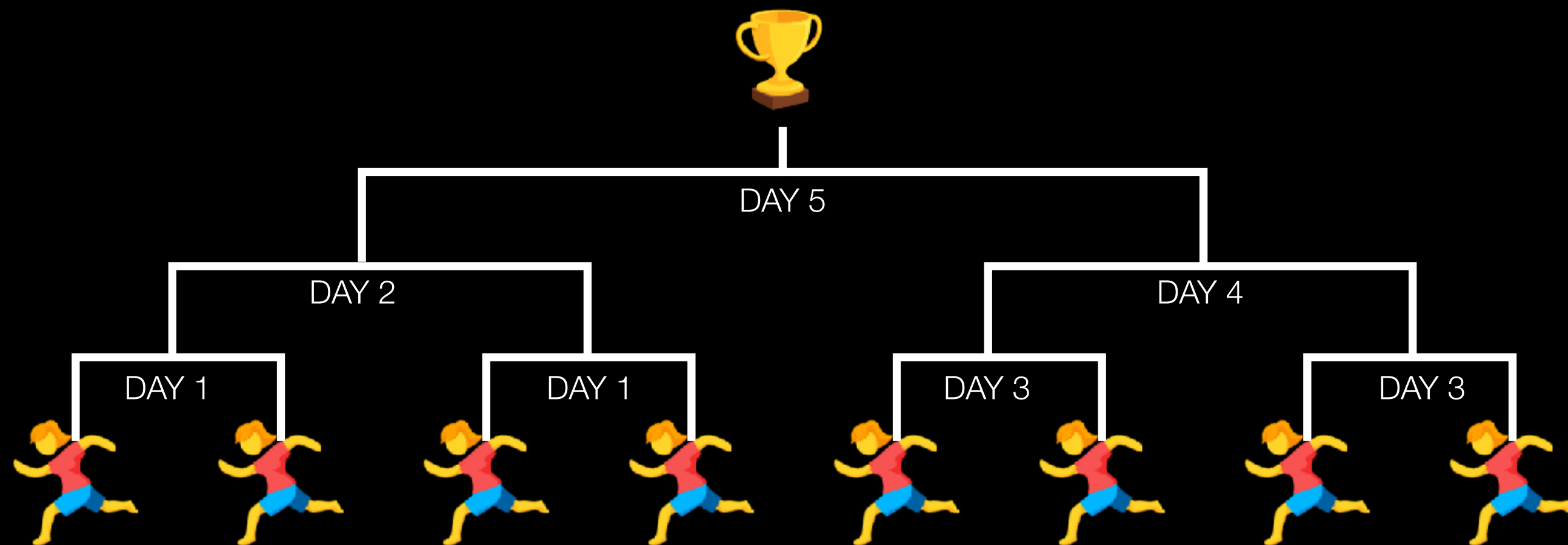
reduction trees

We can do this with two courts!



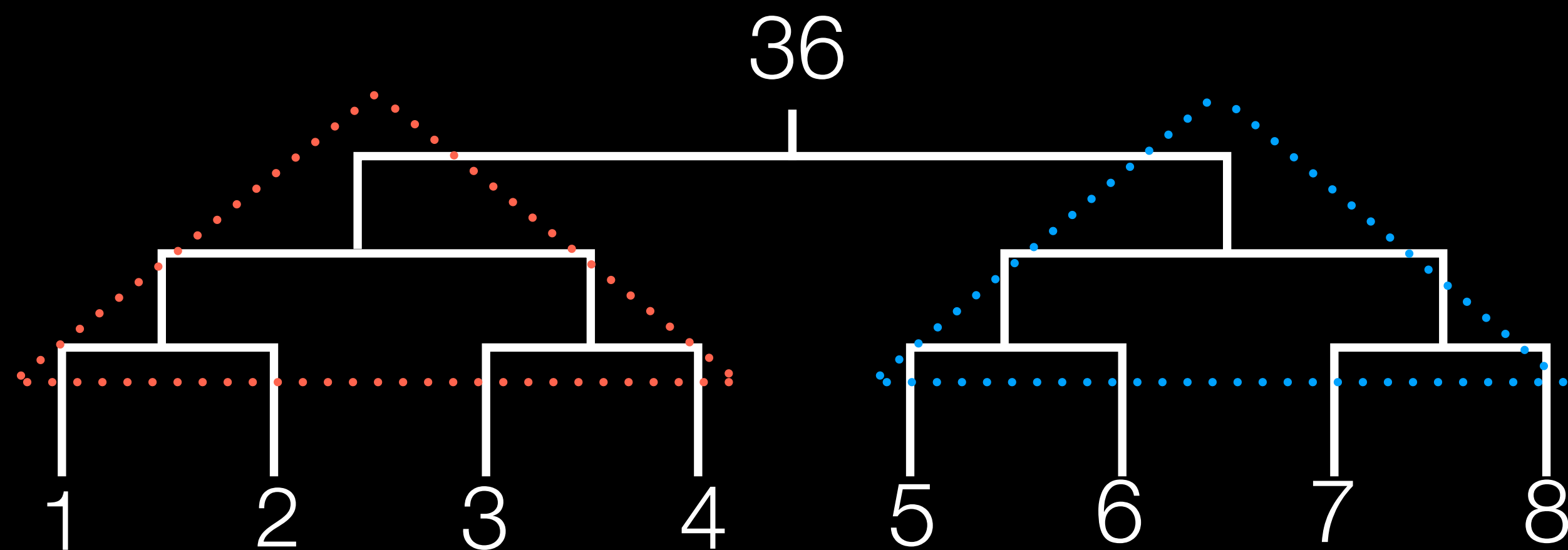
reduction trees

Ugh it's getting expensive to transport players to and fro.
Here's another option...



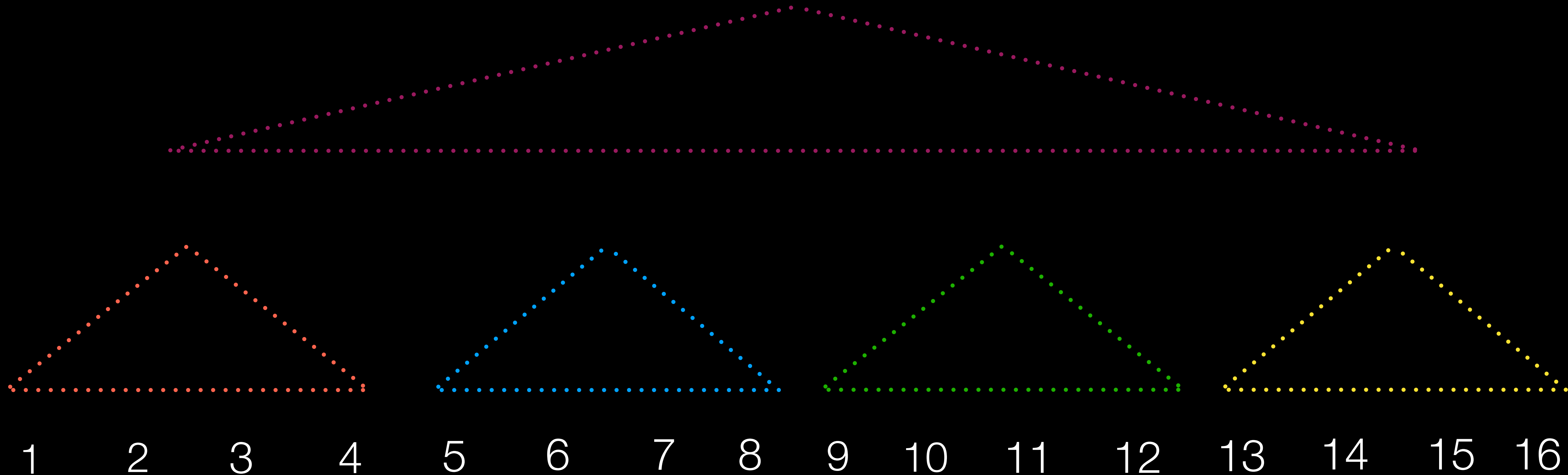
reduction trees

This just in:
the players are numbers;
the matches are addition!



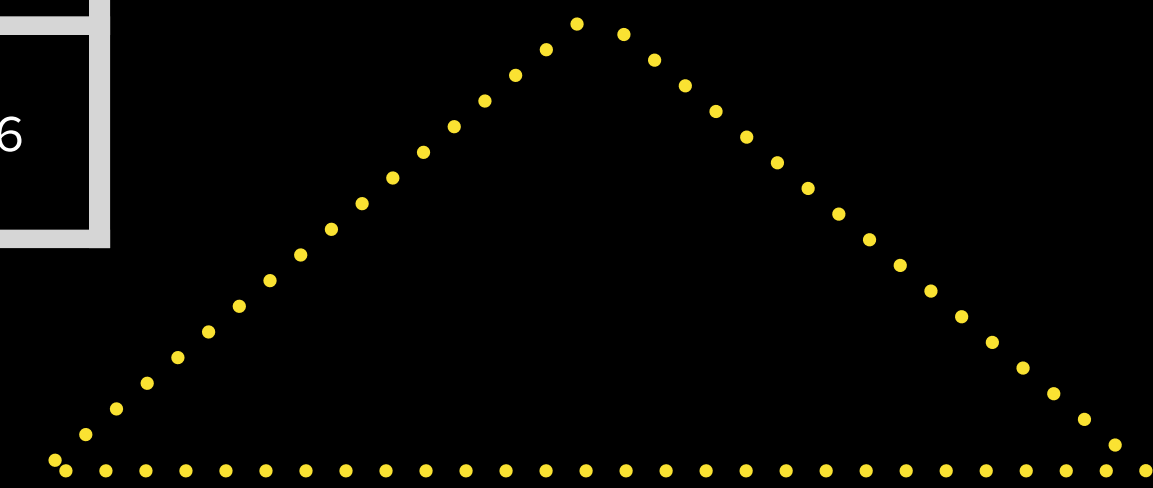
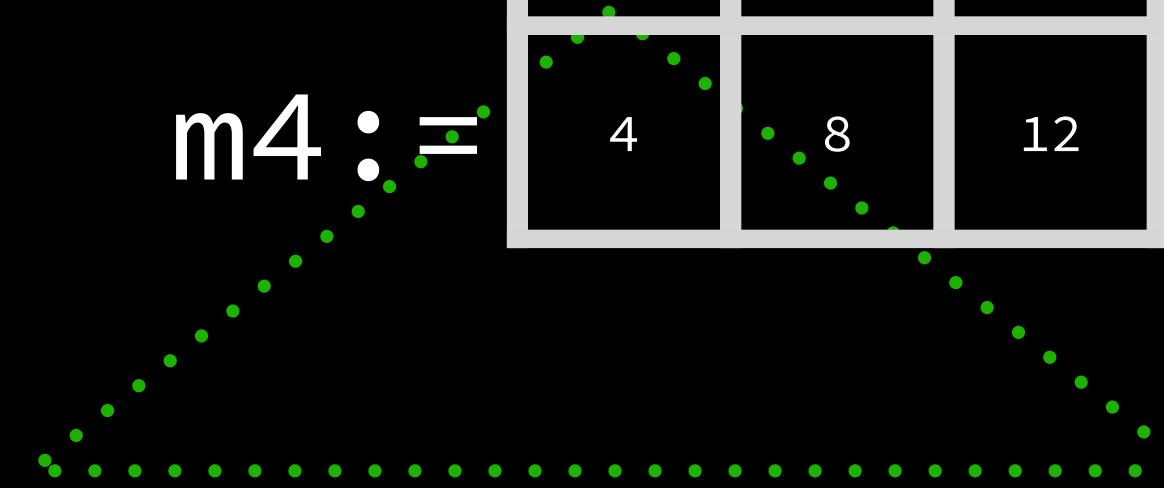
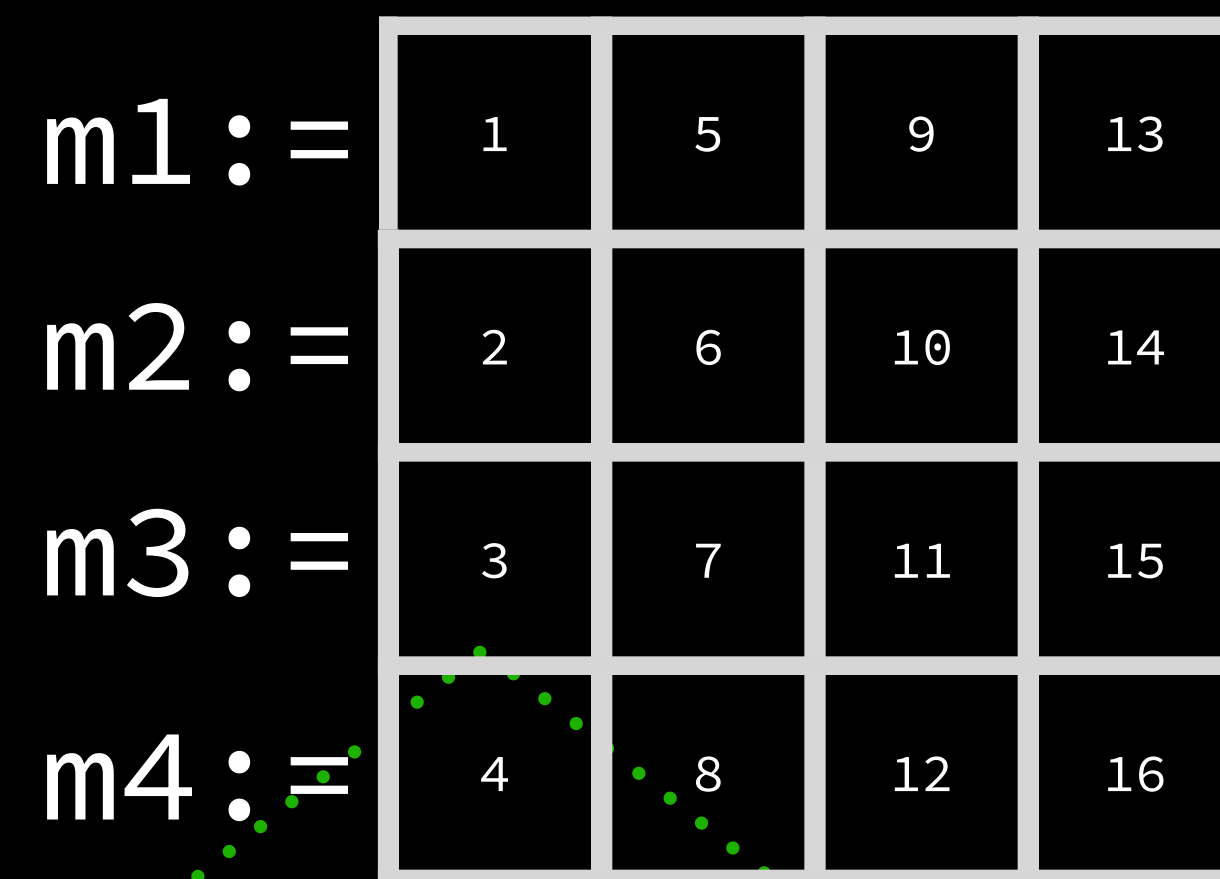
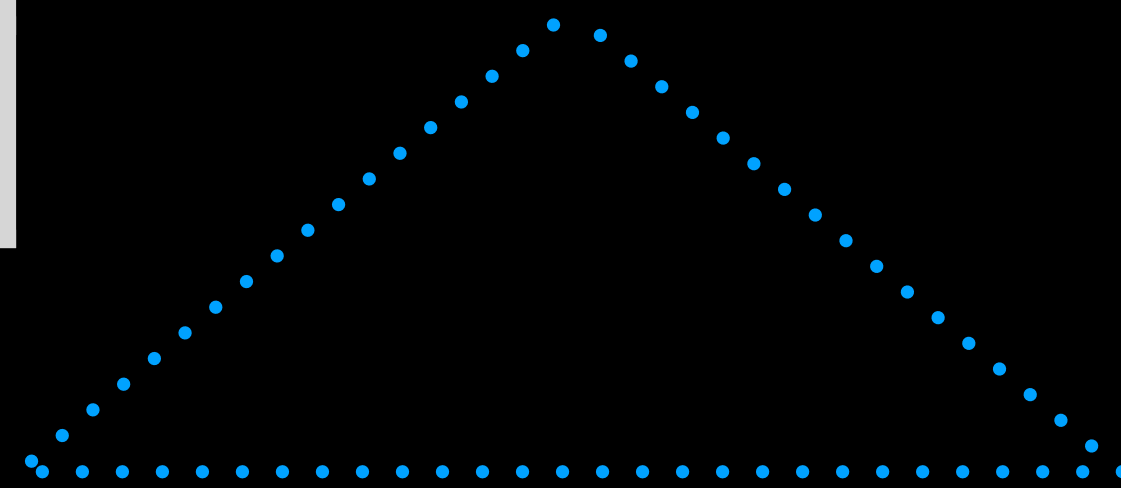
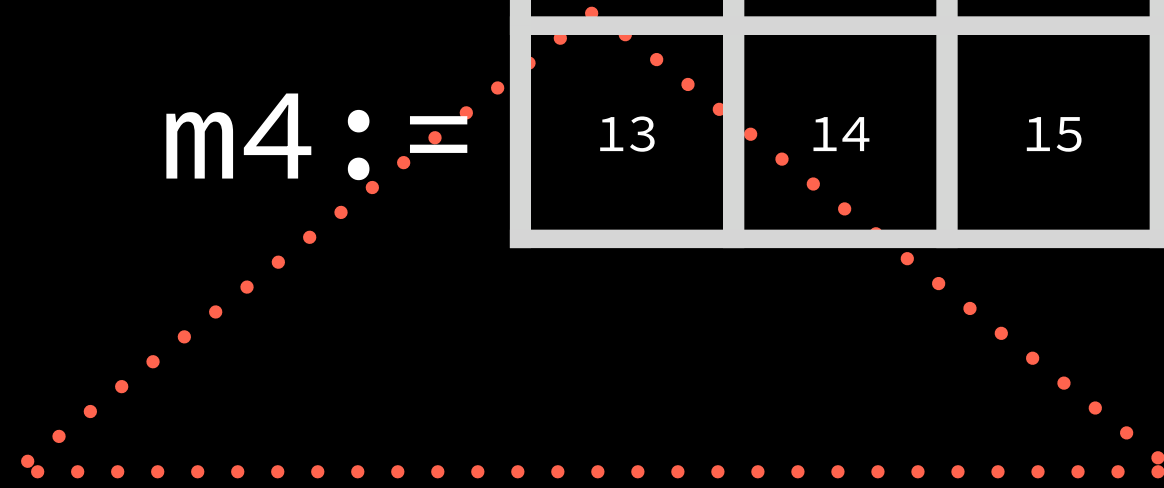
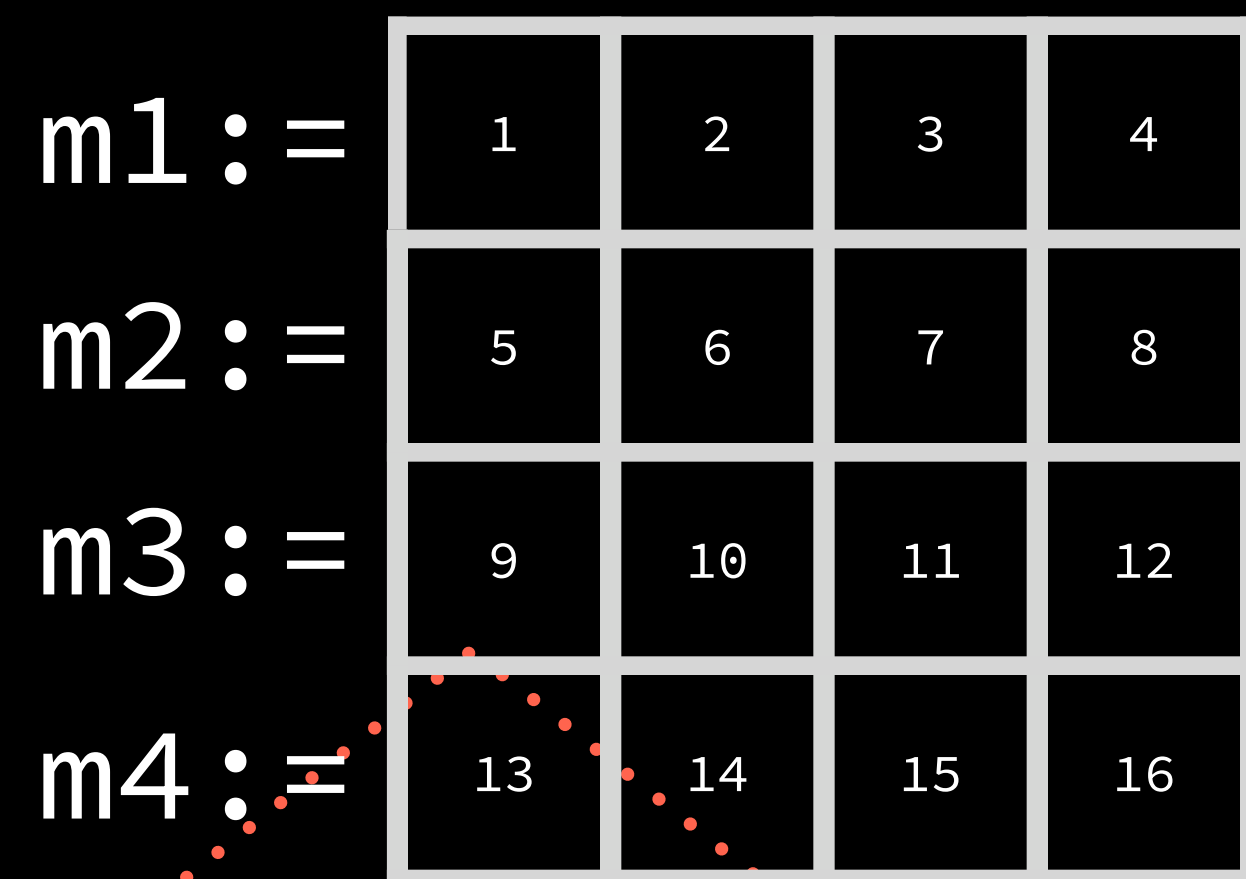
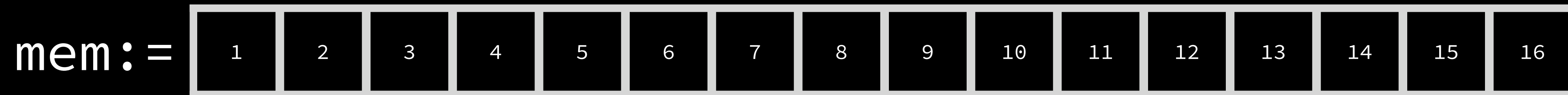
reduction trees

How would you handle 16 numbers?



reduction trees, feat. banking

Where would you place the 16 numbers?



1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

arbitrage

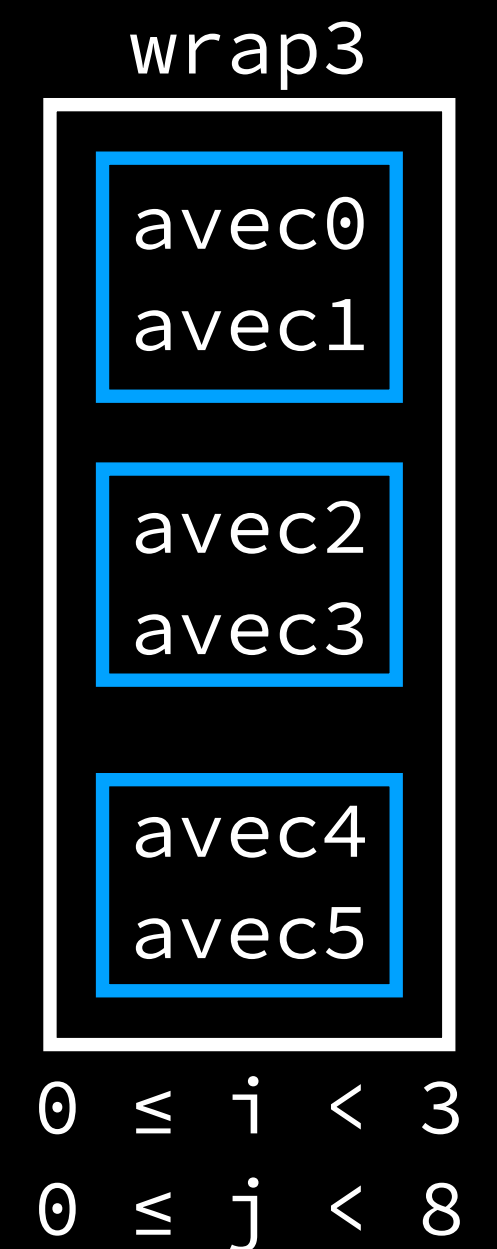
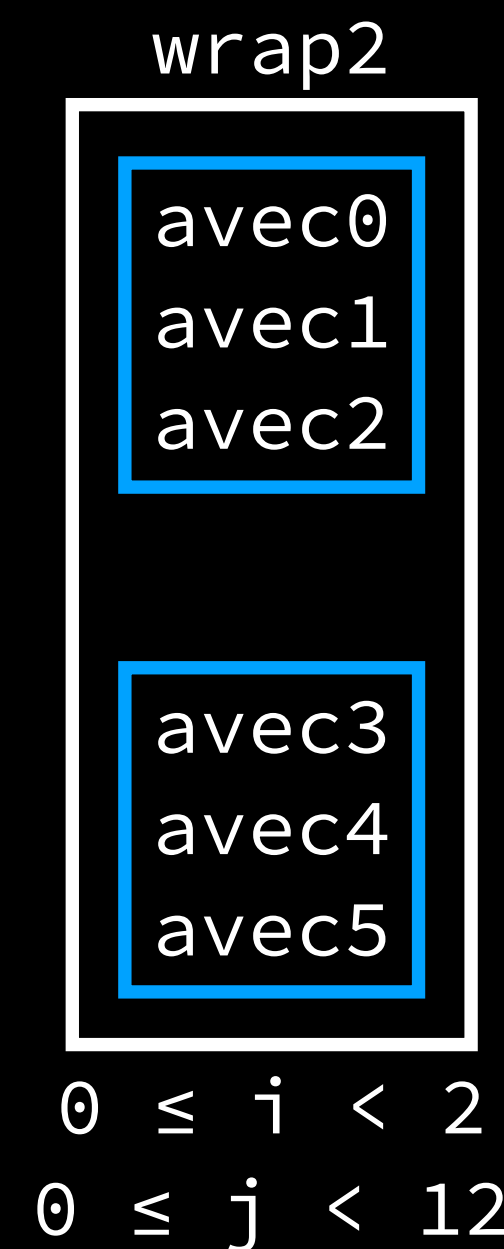
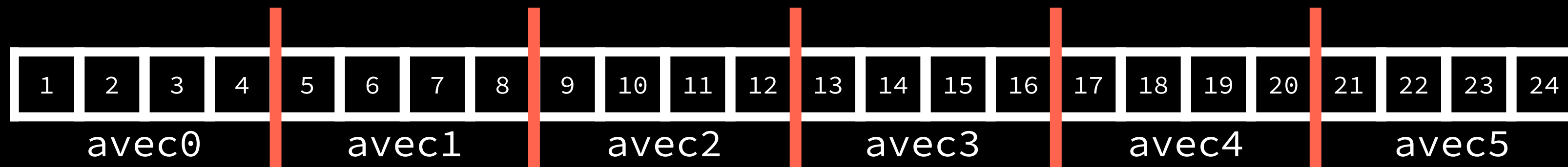
What if I want to parallelize the same memory, but with different banking factors?

```
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

- you can work on calyx frontends/calyx/calyx backends
- here's people working on calyx!
- verification, optimization, lots of buzzwords :D