Advanced Digital Circuit Design (67551)

FINAL PROJECT: TURBOSIM

Yaron Inger

Roy Zektzer

Michael Kimi

Ran Mor

Workflow

- Learn about Verilog’s event model
- Research data structures for event queue
- Create Ubersim
- Decide about data structure for event queue, find out about the need of set (CTU)
- Finish algorithm and selected data structure testing using Ubersim
- Write Turbosim
- Run tests and CR

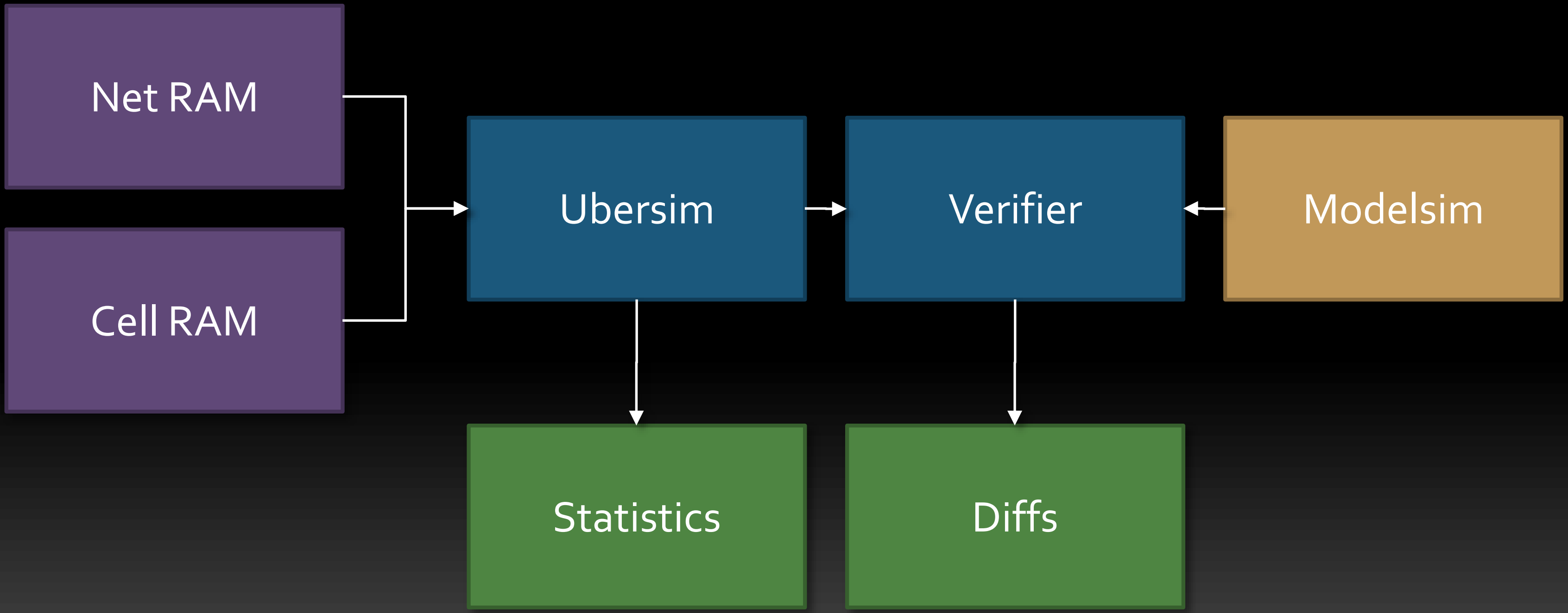
Ubersim

- Verilog Python simulator
- Used for researching CPU algorithms and data structures
- Integrates with DUT's made for Turbosim

- Makes testing and data collection much easier

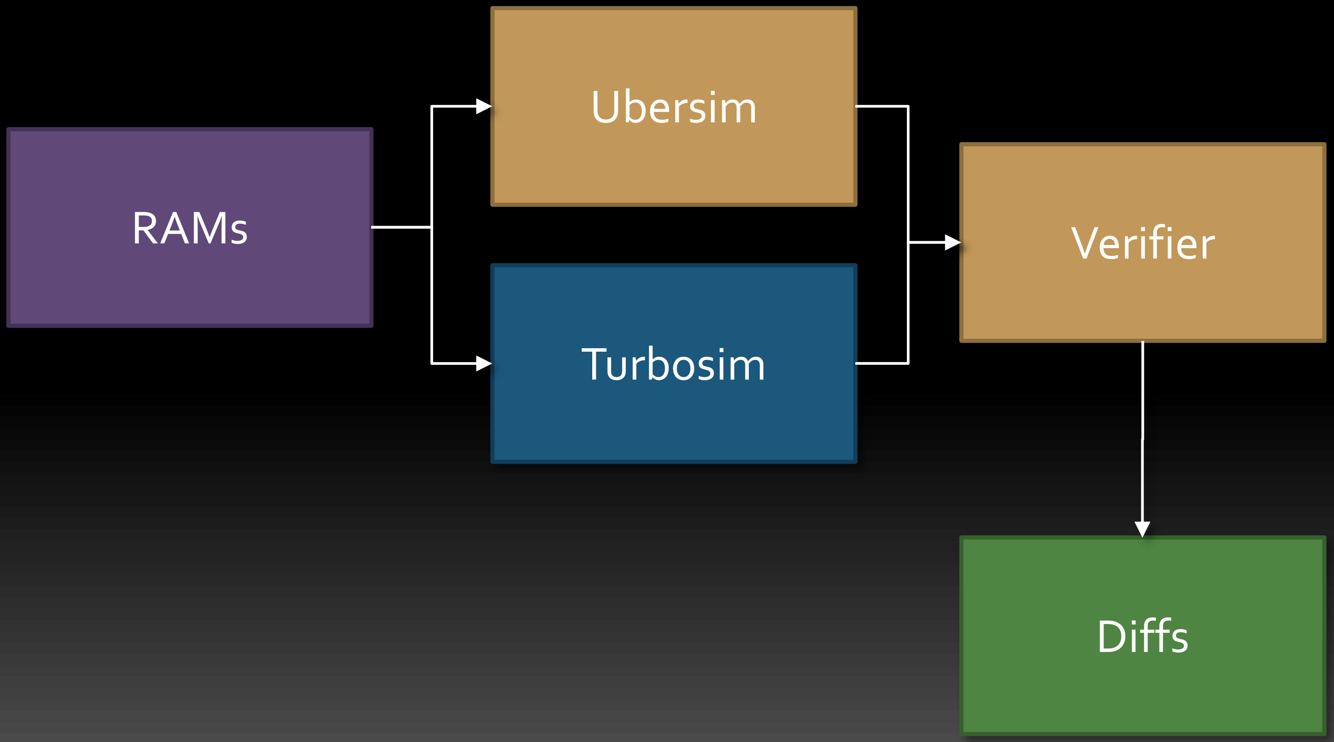
Ubersim Operation Methods

- Work against Modelsim



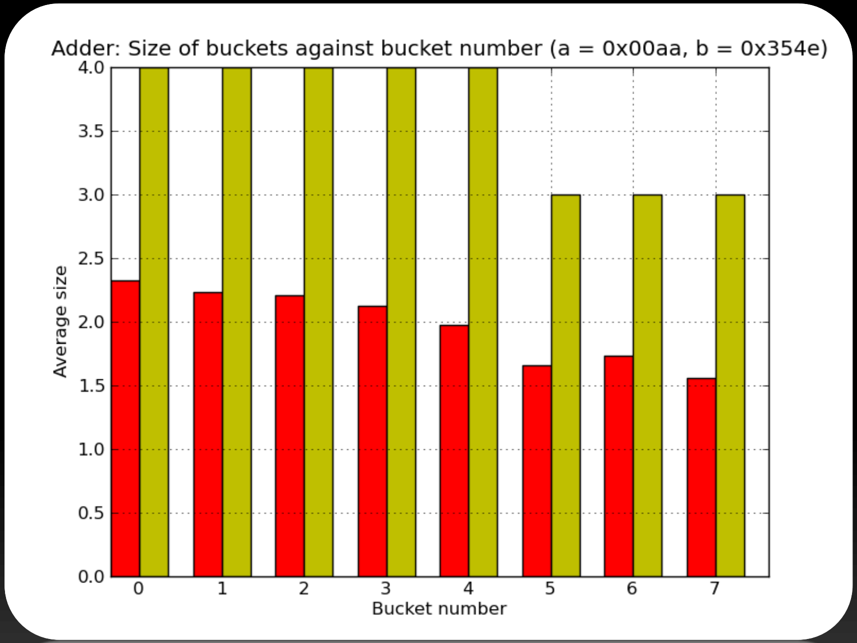
Ubersim Operation Methods

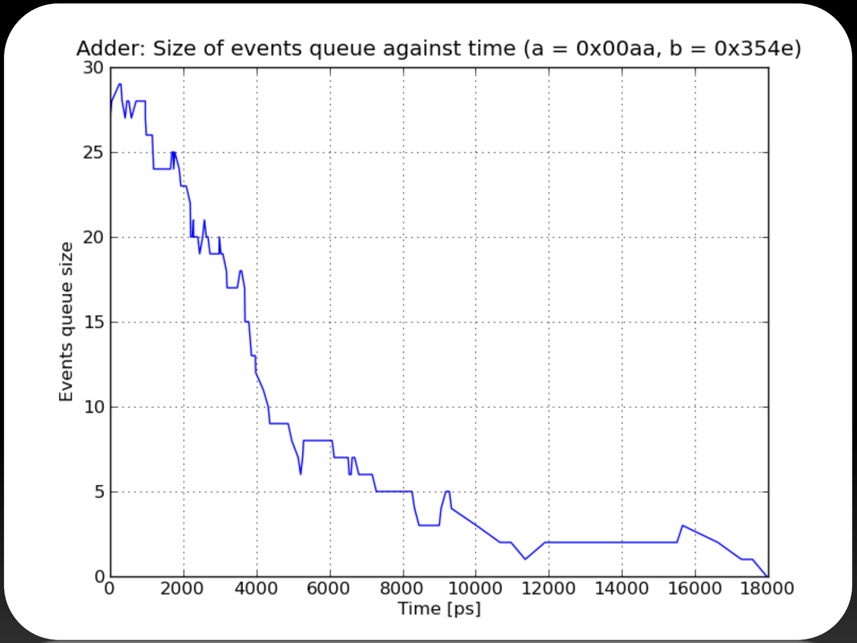
- Work against Turbosim



(Some) Results

- Work against Turbosim





Turbosim's CPU

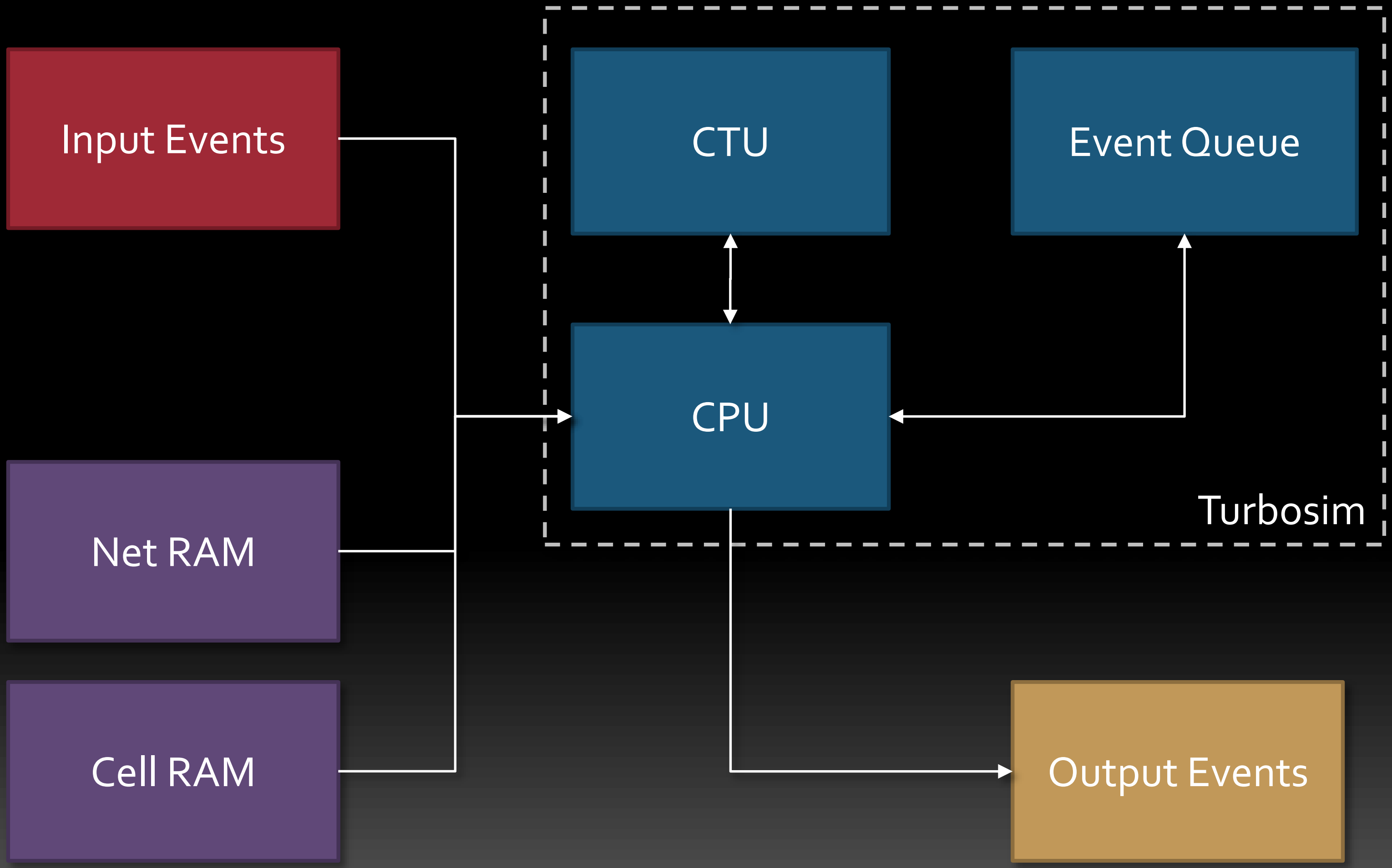
- Verilog is a language that represents real-world multitasking
- Time advances in parallel

- How we create a simulator then?
  - Simulator has a built-in 'world time'
  - Simulation is event driven, where events are pushed and popped from data structures
  - There is a partial ordering relation on events

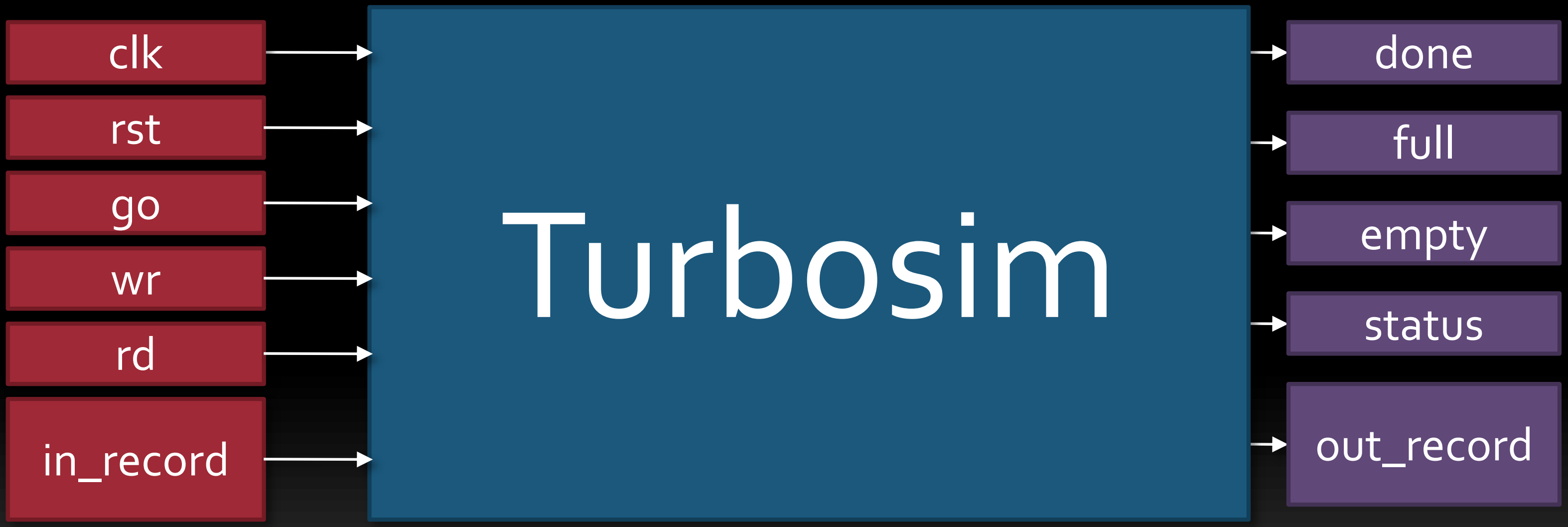
Turbosim’s CPU (2)

- Q: What happens when we have 2 events for the same object simultaneously?
- A: (Verilog IEEE, pp. 109)
  - *“In situations where a right-hand operand changes before a previous change has had time to propagate to the left-hand side, then the following steps are taken:*
    - *The value of the right-hand expression is evaluated.*
    - *If this right-hand side value differs from the value currently scheduled to propagate to the left-hand side, then the currently scheduled propagation event is descheduled.*
    - *If the new right-hand side value equals the current left-hand side value, no event is scheduled.*
    - *If the new right-hand side value differs from the current left-hand side value, a delay is calculated in the standard way using the current value of the left-hand side, the newly calculated value of the right-hand side, and the delays indicated on the statement; a new propagation event is then scheduled to occur delay time units in the future.*

Turbosim's CPU Overview



Turbosim's CPU Overview (2)
Interface



Turbosim's CPU Algorithm

- Algorithm:
  - Wait for 'go' signal
  - Read stimulus from input and generate events only for new net values
  - Until the event queue is empty
    - e <- Peek at the upcoming event
    - If e.time != sim.time, generate events and set topmost's event's time as sim.time
    - e <- De-queue the event from the event queue

Turbosim's CPU Algorithm (2)

- n <- Read net record related to e
- If e.value == n.value or n.nextTime != sim.time, reject this event and fetch next event, else
- n.value = e.value
- Create output record for the event
- Add attached cells to the CTU and store new net value in each cell entry
- If the event queue is empty, generate events

Turbosim’s CPU Algorithm (3)

- Generate events
  - While the CTU is not empty
    - c <- Read CTU record
    - n <- c.outputNet
    - Calculate new output from input values
    - If output is different than n.nextValue, push new event to the event queue
    - Update n.nextTime and n.nextValue accordingly

RAM Structure

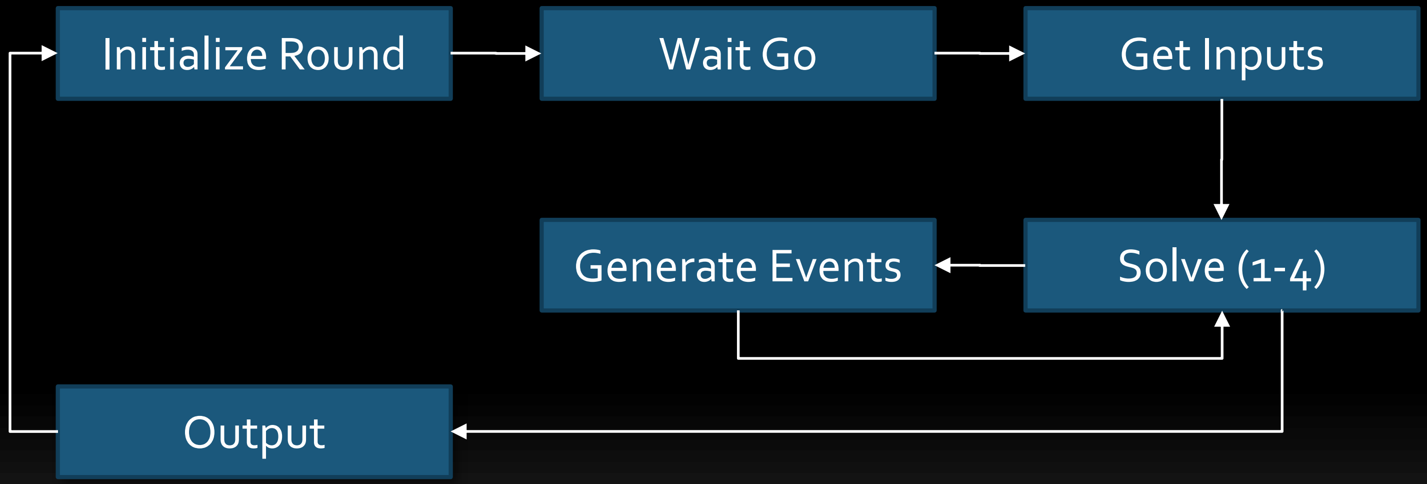
- Net RAM
  - Input cell index
  - 4 Output cells indices + pin indices
  - Number of loads
  - Current value (initially ‘x’)
  - Control (unused)
  - Next event time
  - Next event value

RAM Structure (2)

- Cell RAM
  - Cell type (not, buf, and, or, nand, nor)
  - Delay (ps)
  - Output net index
  - 4 Input net indices + their current value

Turbosim's Implementation

- Stages overview

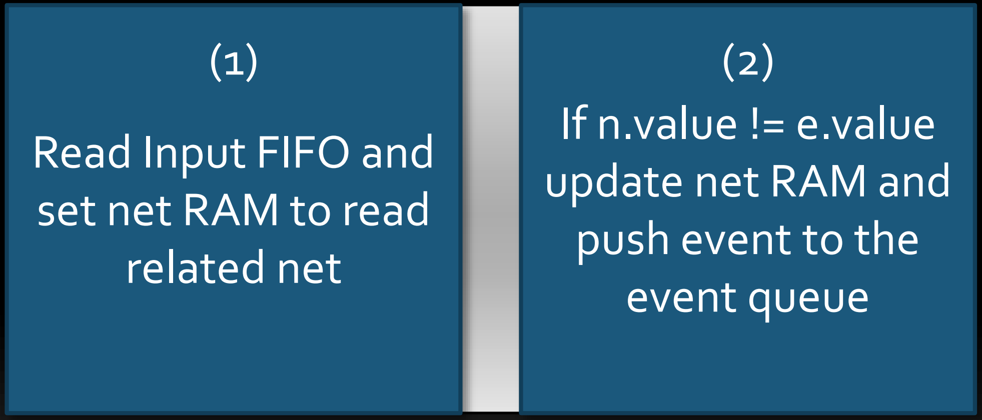


Turbosim’s Implementation (2)

- Initialize Round
  - Used to initialize registers before a simulation round begins
  - All ‘reset’ logic was moved to this stage
  - Example of objects to reset:
    - Simulator’s time
    - Pipeline stages
- Wait Go
  - Waits for a ‘go’ command from the outer world

Turbosim's Implementation (3)

- Get Inputs
  - A 2-stage pipeline



(1)

Read Input FIFO and set net RAM to read related net

(2)

If n.value != e.value update net RAM and push event to the event queue

  - The event queue may be busy for write, therefore we may suspend the entire pipeline until it's free
  - Practically this doesn't happen

Turbosim’s Implementation (4)

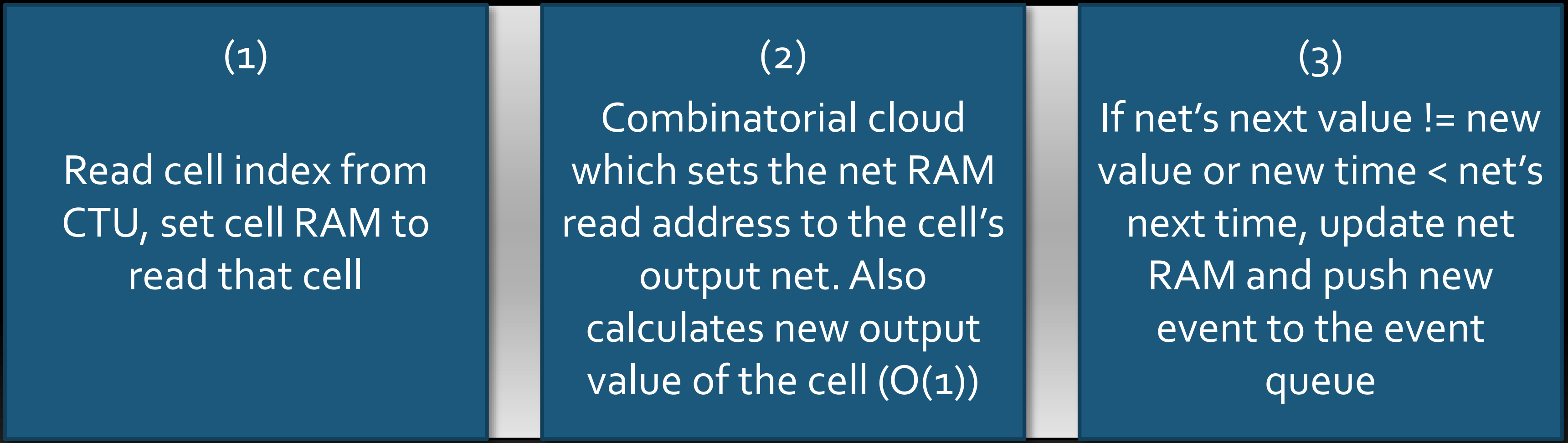
- Solve (Part 1)
  - Checks if the event queue and the CTU are empty, if so we’re done
  - If the event queue is empty and the CTU isn’t, generate events from the CTU
  - Peek at the current event and generate events if needed
  - Increase simulator time if needed
  - De-queue the event

Turbosim's Implementation (5)

- Solve (Part 2)
  - Waits for the net ram to read selected net from part 1
- Solve (Part 3 + 4)
  - If n.value == e.value or n.nextTime != simTime, drop event, else
  - For each net's load, push the cell's index to the CTU and update the cell RAM with the new net value

Turbosim's Implementation (6)

- Generate Events
  - A 3-stage pipeline



(1)

Read cell index from CTU, set cell RAM to read that cell

(2)

Combinatorial cloud which sets the net RAM read address to the cell's output net. Also calculates new output value of the cell (O(1))

(3)

If net's next value != new value or new time < net's next time, update net RAM and push new event to the event queue

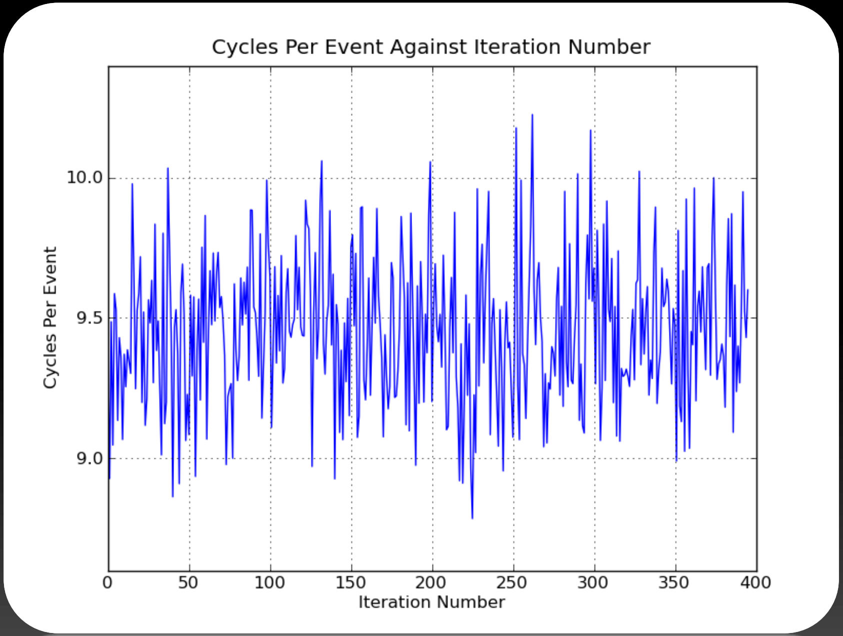
  - The event queue may be busy for write, therefore we may suspend the entire pipeline until it's free
  - Practically this doesn't happen

Turbosim’s Implementation (7)

- Output
  - Wait for the output FIFO to become empty
  - Go back to Initialize Round

Turbosim's Performance

- Mean: 9.454 Cycles / Event
- Median: 9.452 Cycles / Event



** DUT: Adder*

EVENT QUEUE

Event Queue

Goal:

- Store and retrieve events in *not* descending order

Implementation:

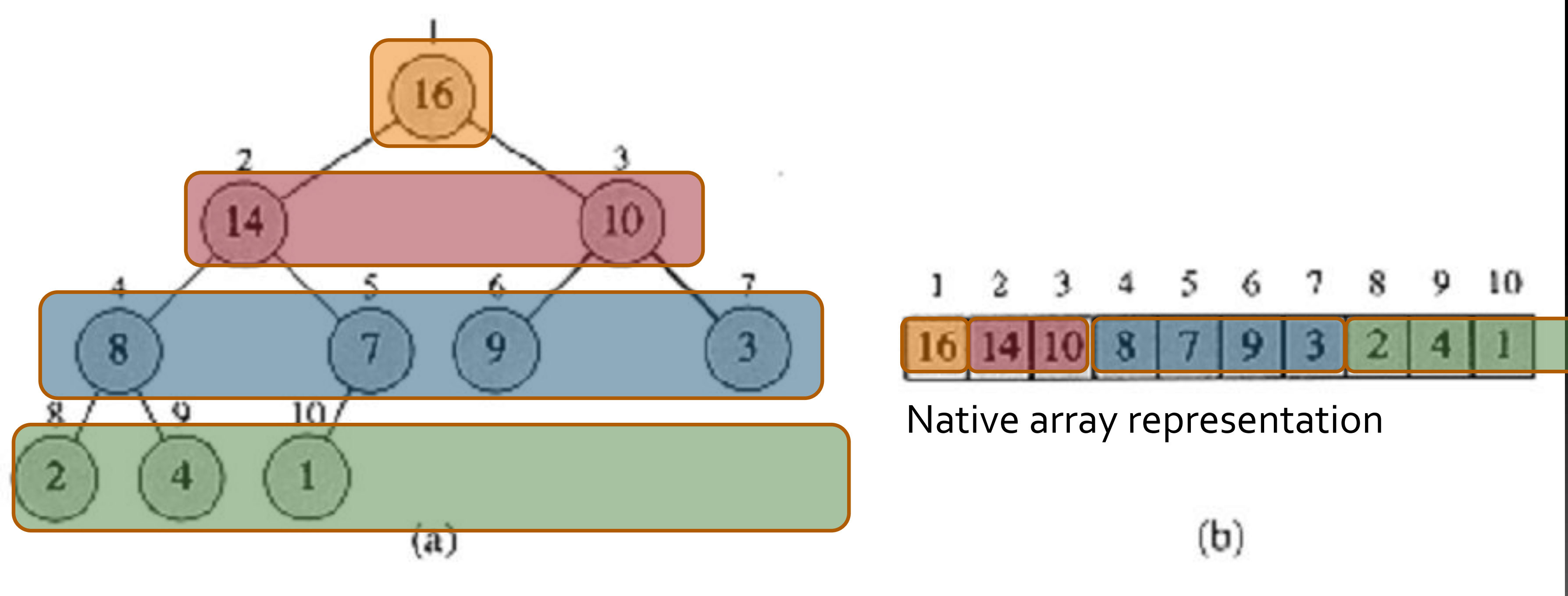
- *Priority Queue* data structure, using *binary heap* implementation:
  - Insertion: log(n)
  - Extraction: log(n)

*n – number of elements in heap

Priority Queue

Definition * :

- The (binary) heap data structure is an array object that can be viewed as a complete binary tree:



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Native array representation

*Introduction to Algorithms - Cormen

Priority Queue (2)

Maintain heap property:

```
Function HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≦ heap_size[A] and A[l] < A[i]
    largest = l
else
    largest = i
if r ≦ heap_size[A] and A[r] < A[largest]
    largest = r
if largest ≠ i
    exchange A[i] <-> A[largest]
HEAPIFY(A, largest)
```

Find largest element

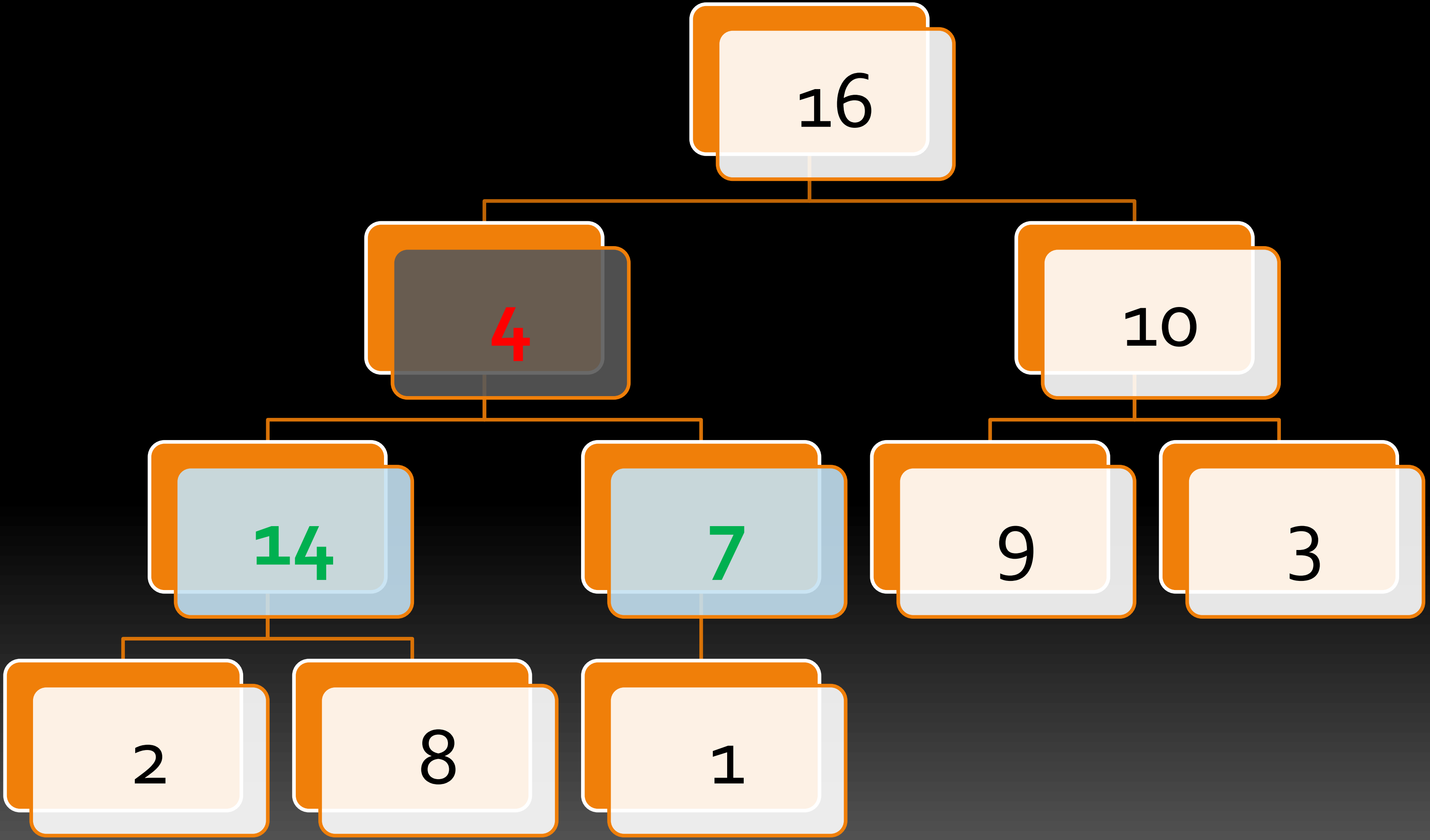
Recursive call

Priority Queue (3)

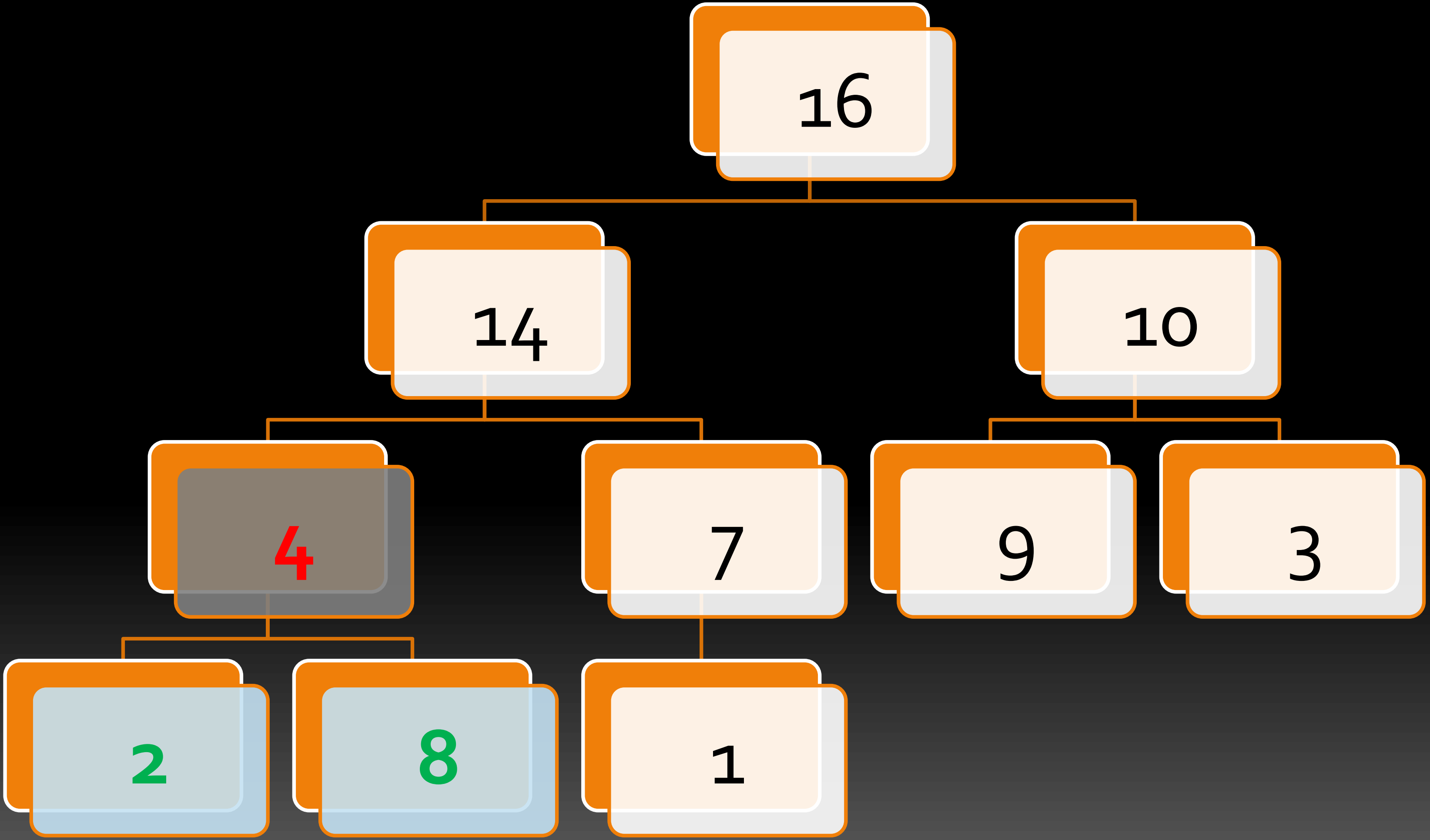
- When HEAPIFY is called, it is assumed that Heaps rooted at `LEFT(i)` and `RIGHT(i)` are legal heaps, but that `A[i]` may be smaller than its children, thus violating the heap property.

- Example is in the next slide…

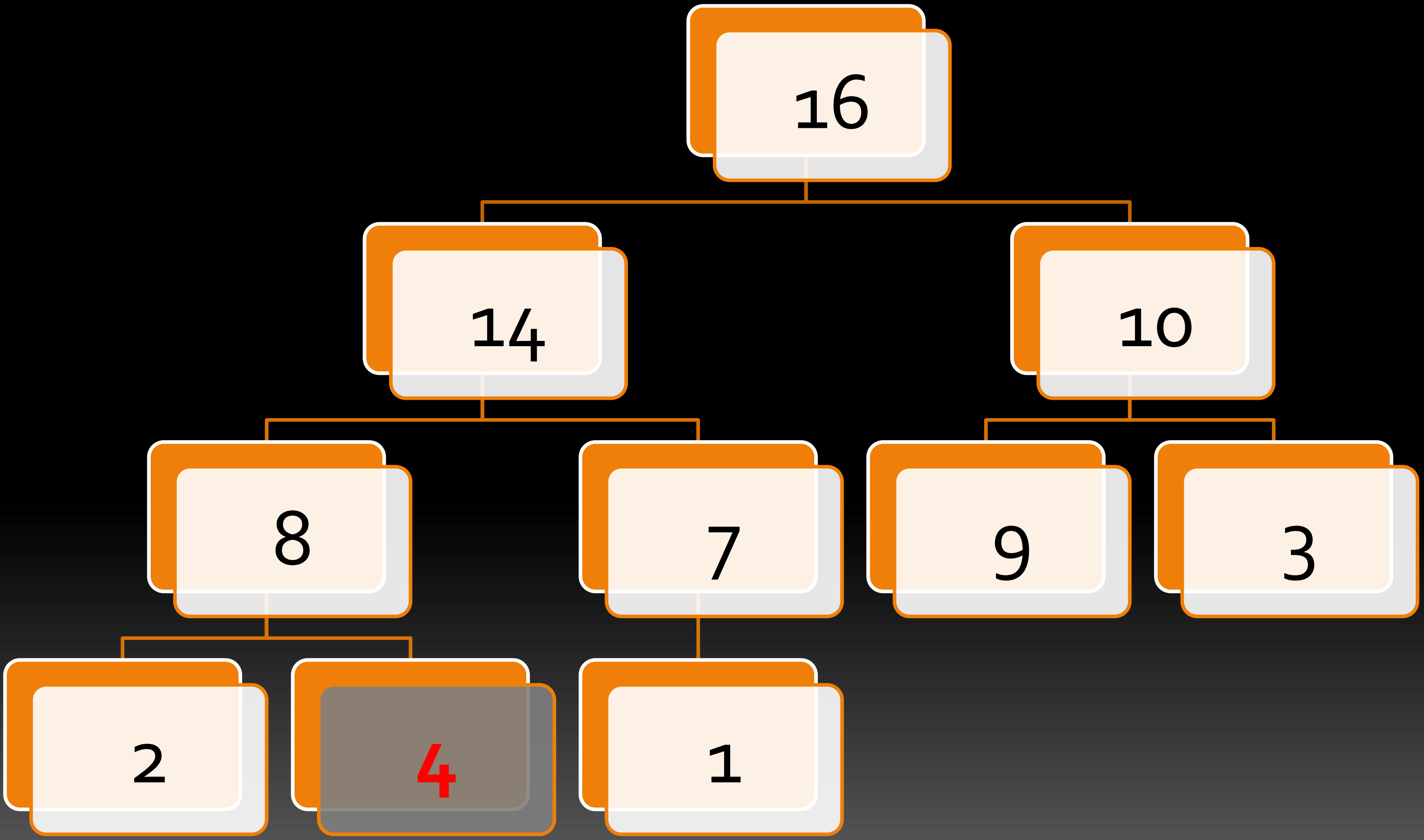
Priority Queue: HEAPIFY example



Priority Queue: HEAPIFY example



Priority Queue: HEAPIFY example



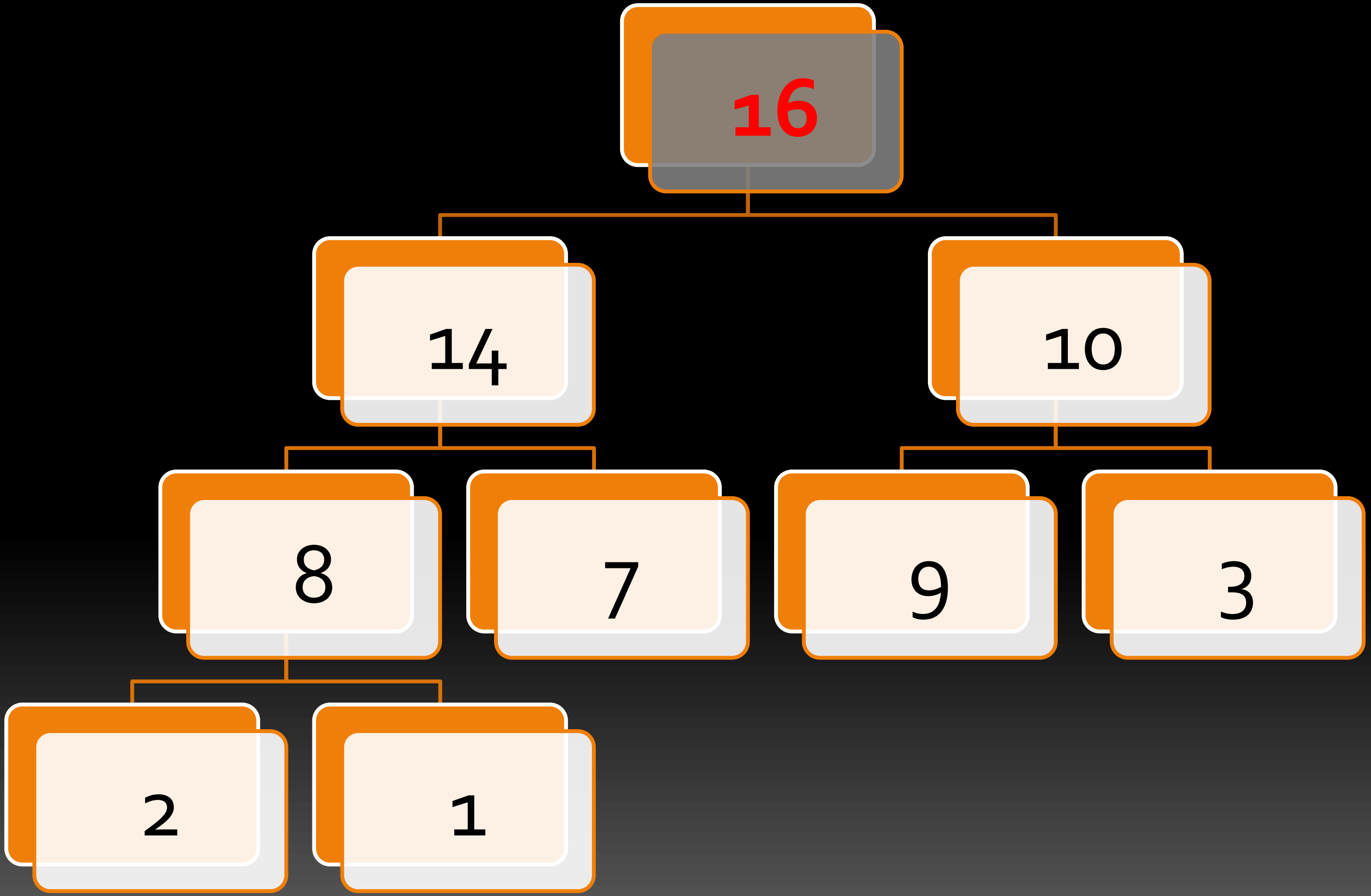
Priority Queue (4)

Extract algorithm:

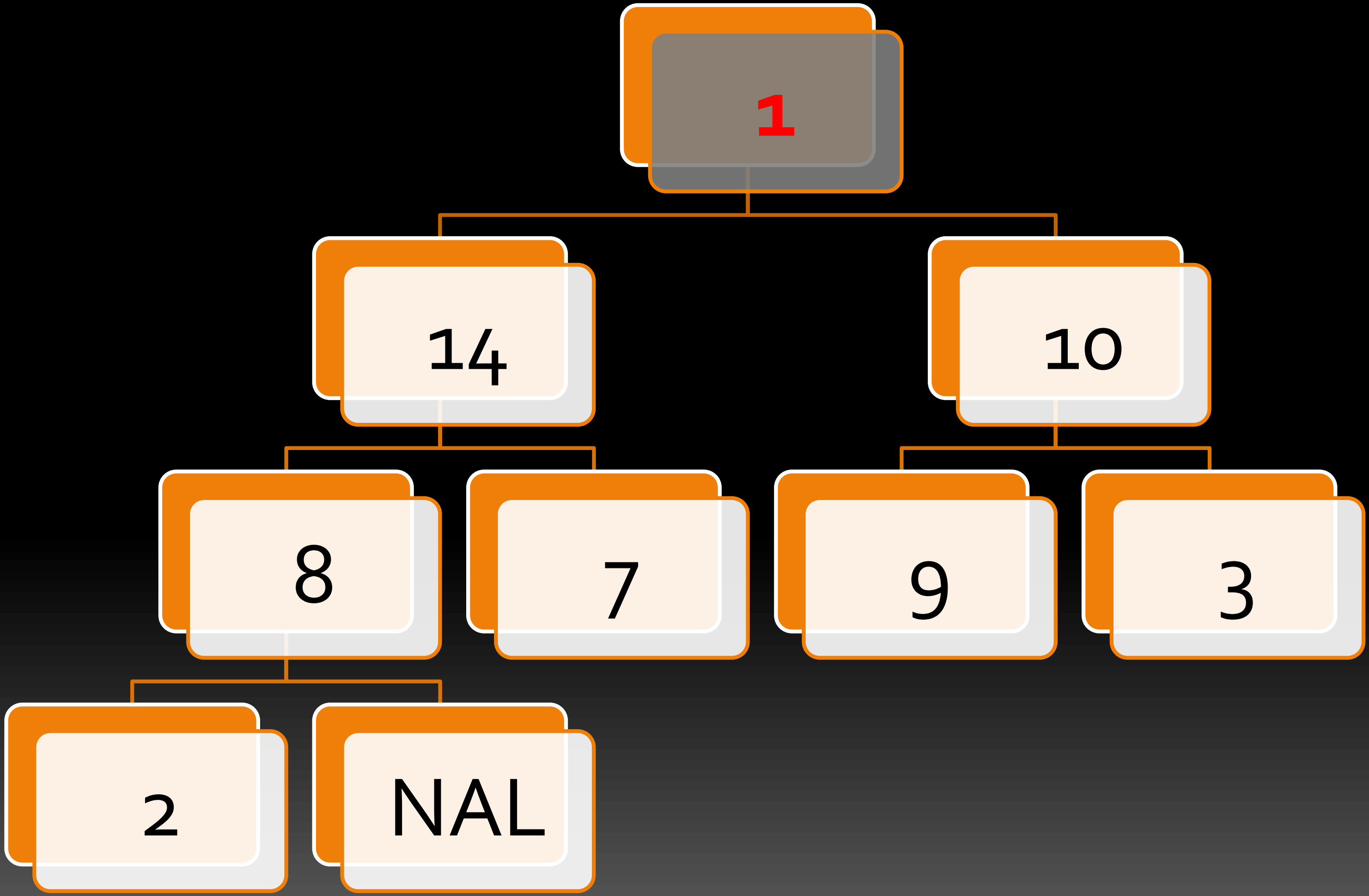
```
Function HEAP_EXTRACT_MAX(A)
    if heap_size[A] < 1
        error "heap underflow"
    max = A[1]
    A[1] = A[heap_size[A]]
    heap_size[A] = heap_size[A] - 1
    HEAPIFY(A, 1)
    return max
```

During first iteration A[1] is valid

Priority Queue: extract example

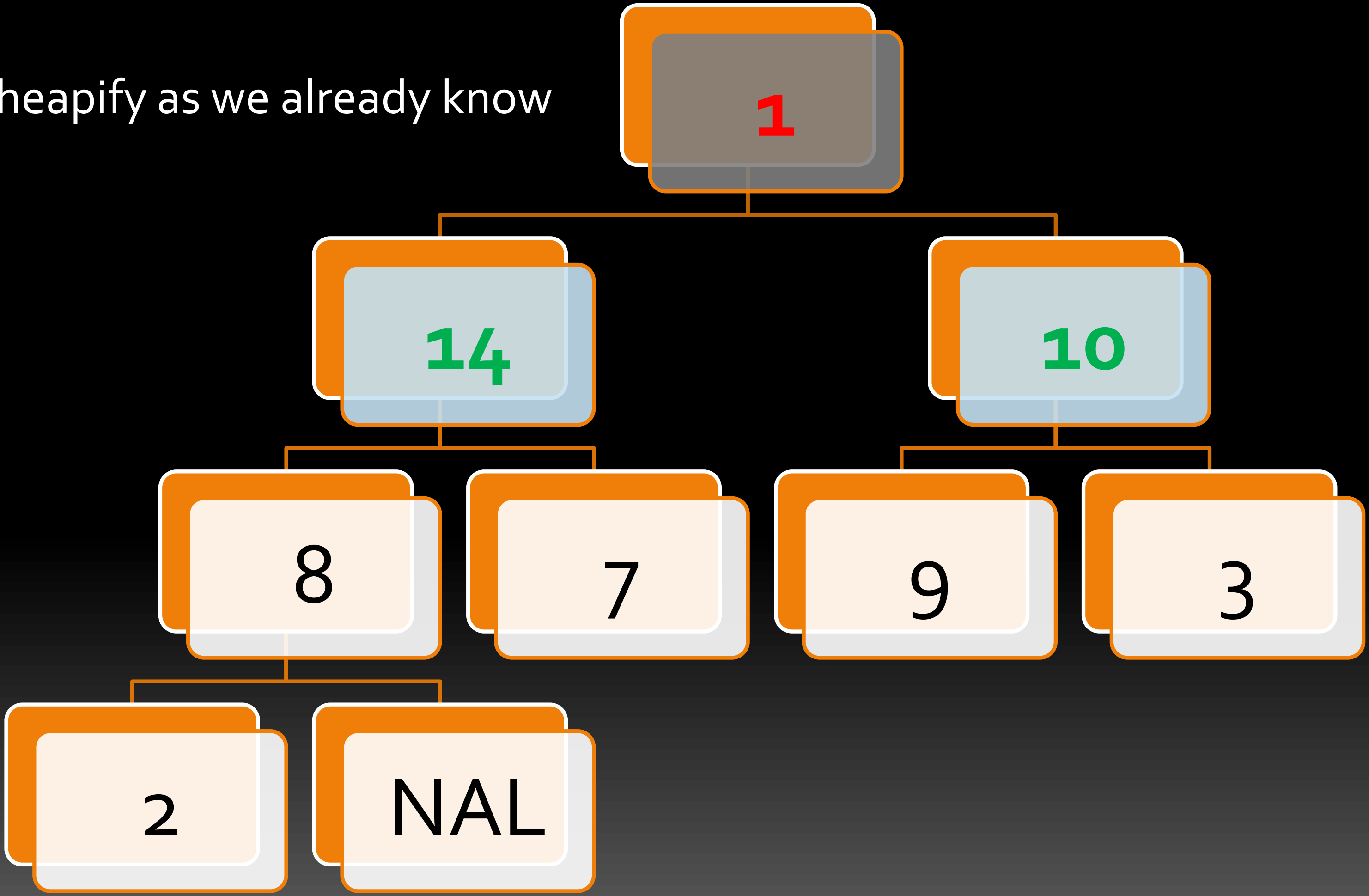


Priority Queue: extract example



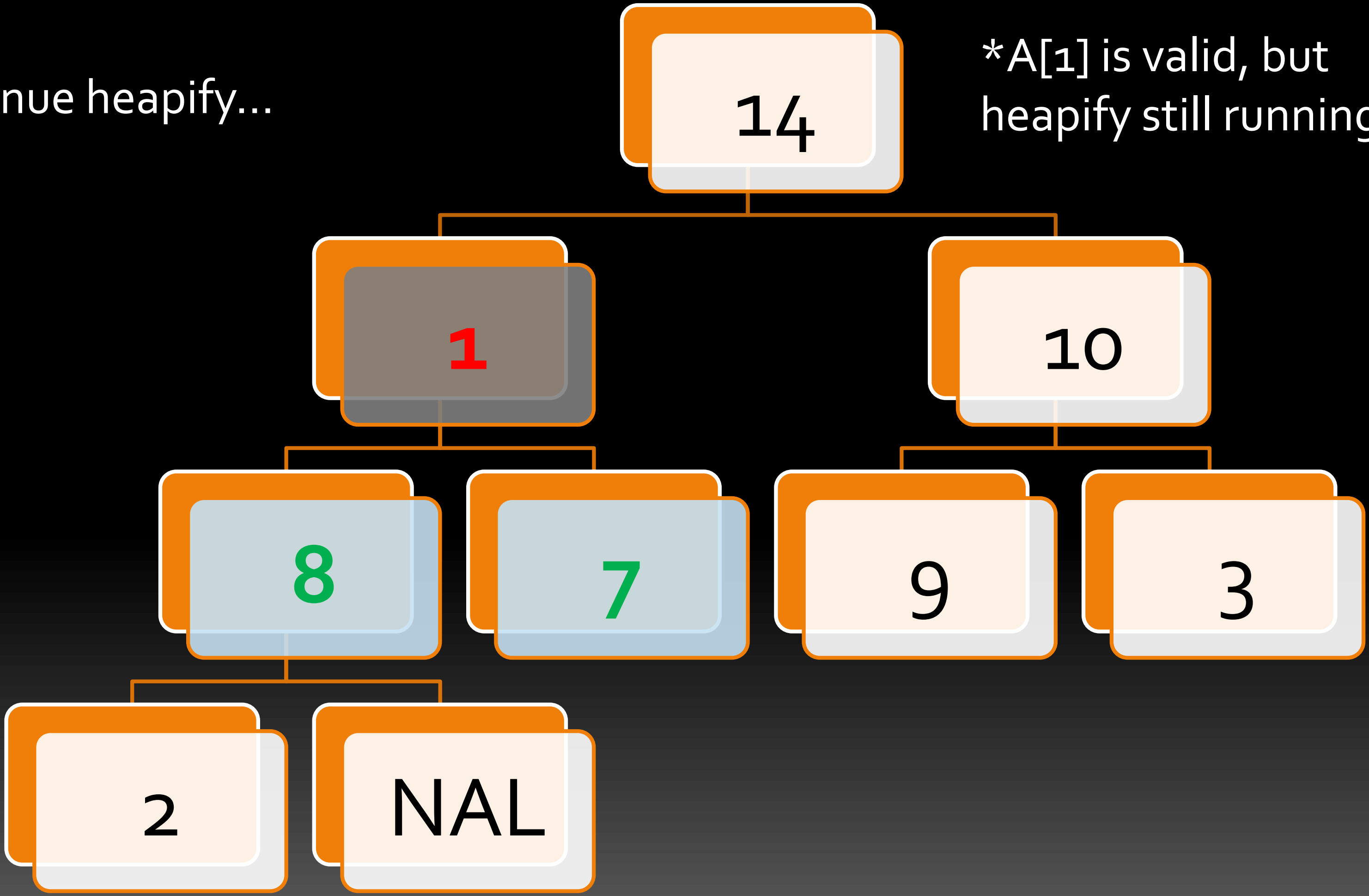
Priority Queue: extract example

Now heapify as we already know



Priority Queue: extract example

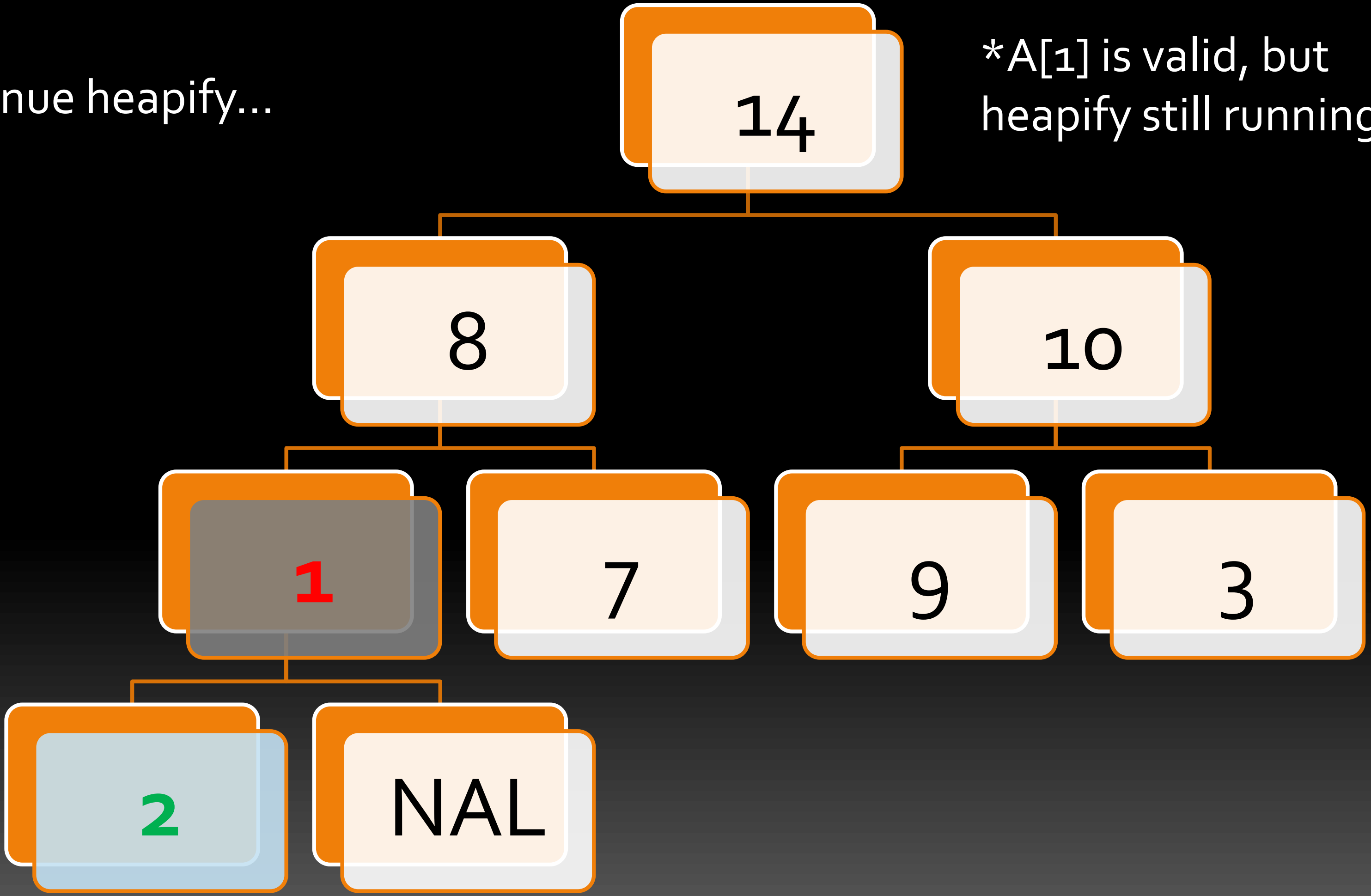
Continue heapify…



*A[1] is valid, but heapify still running

Priority Queue: extract example

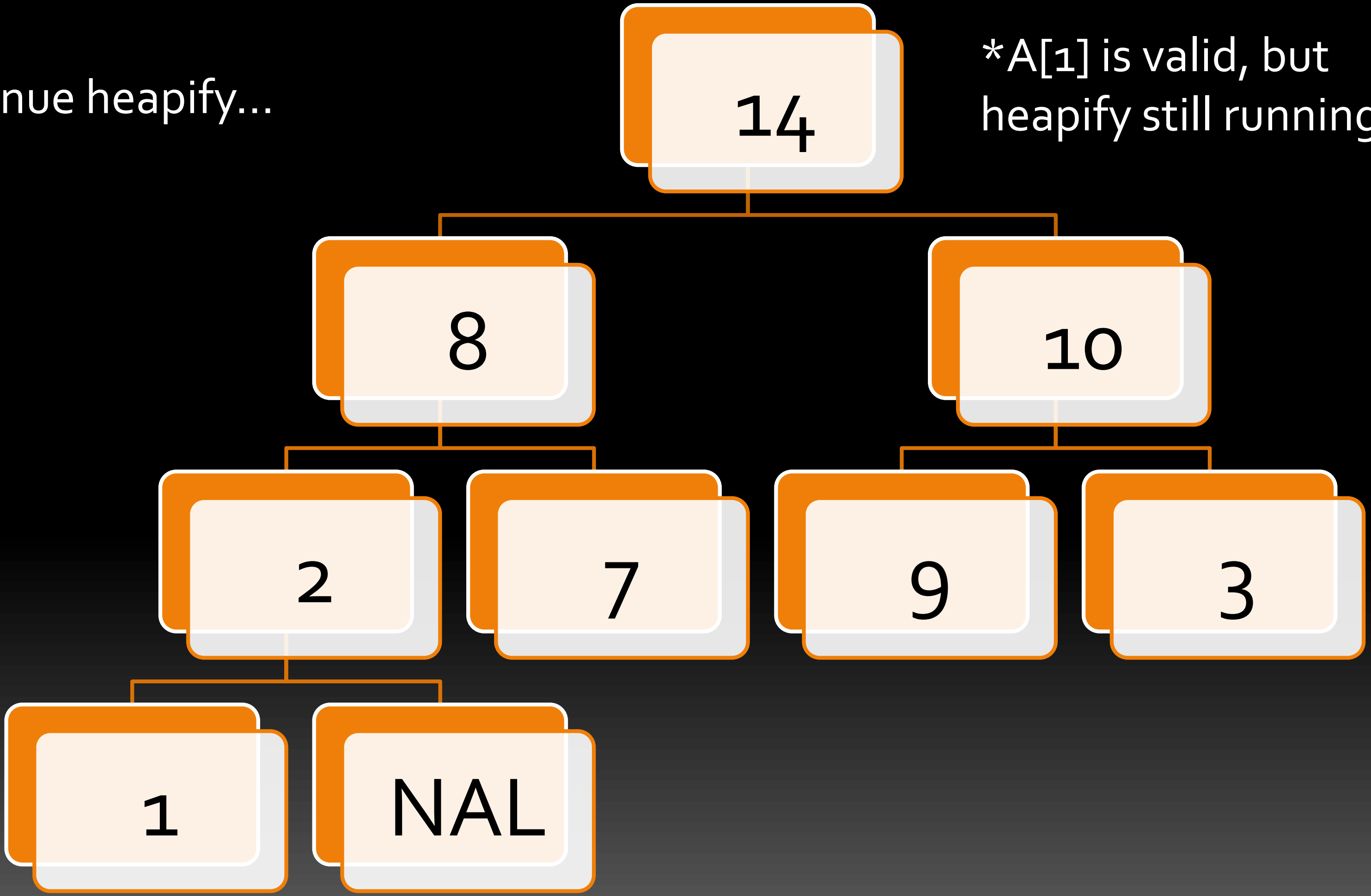
Continue heapify…



*A[1] is valid, but heapify still running

Priority Queue: extract example

Continue heapify…



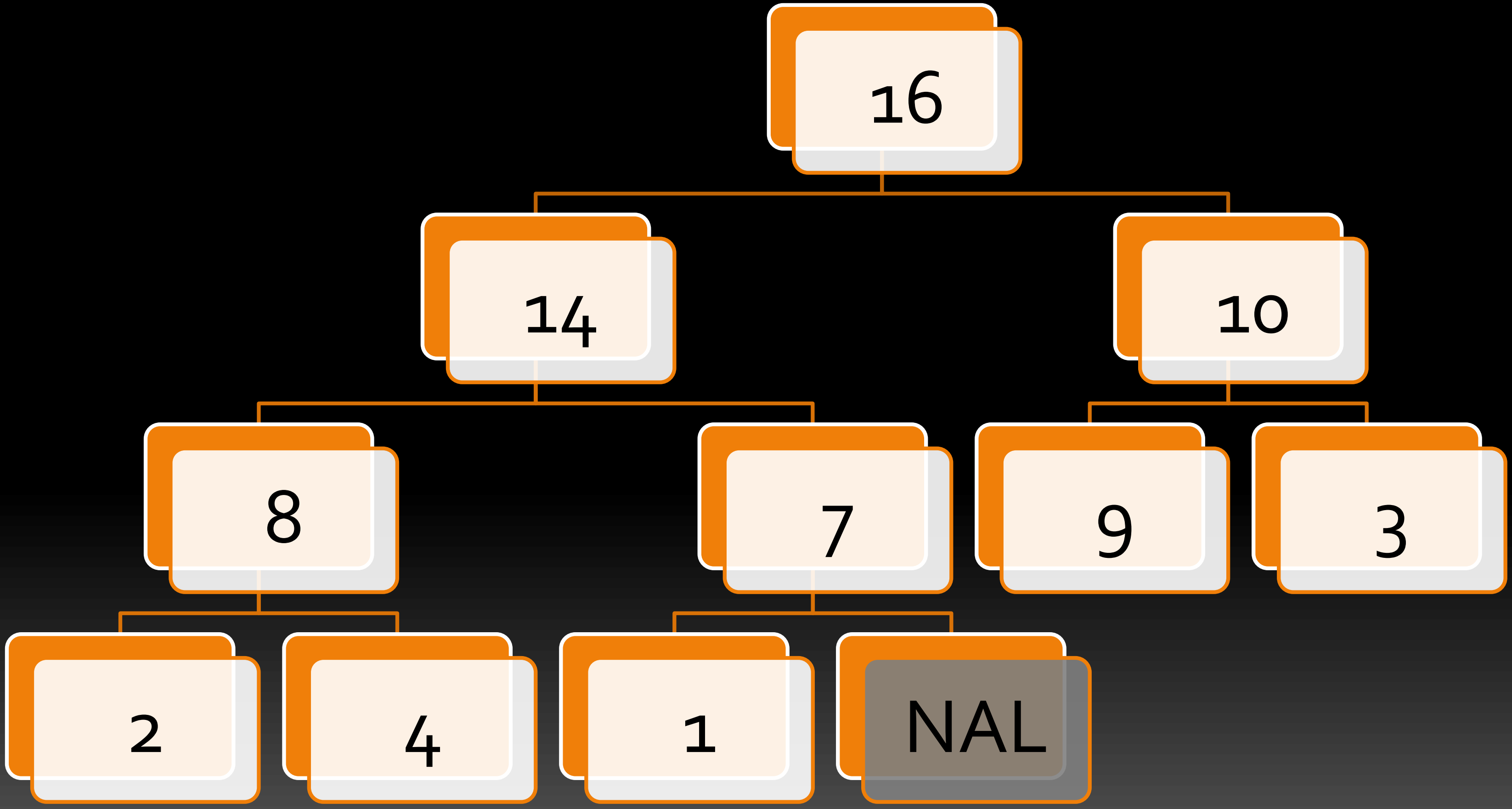
*A[1] is valid, but heapify still running

Priority Queue (5)

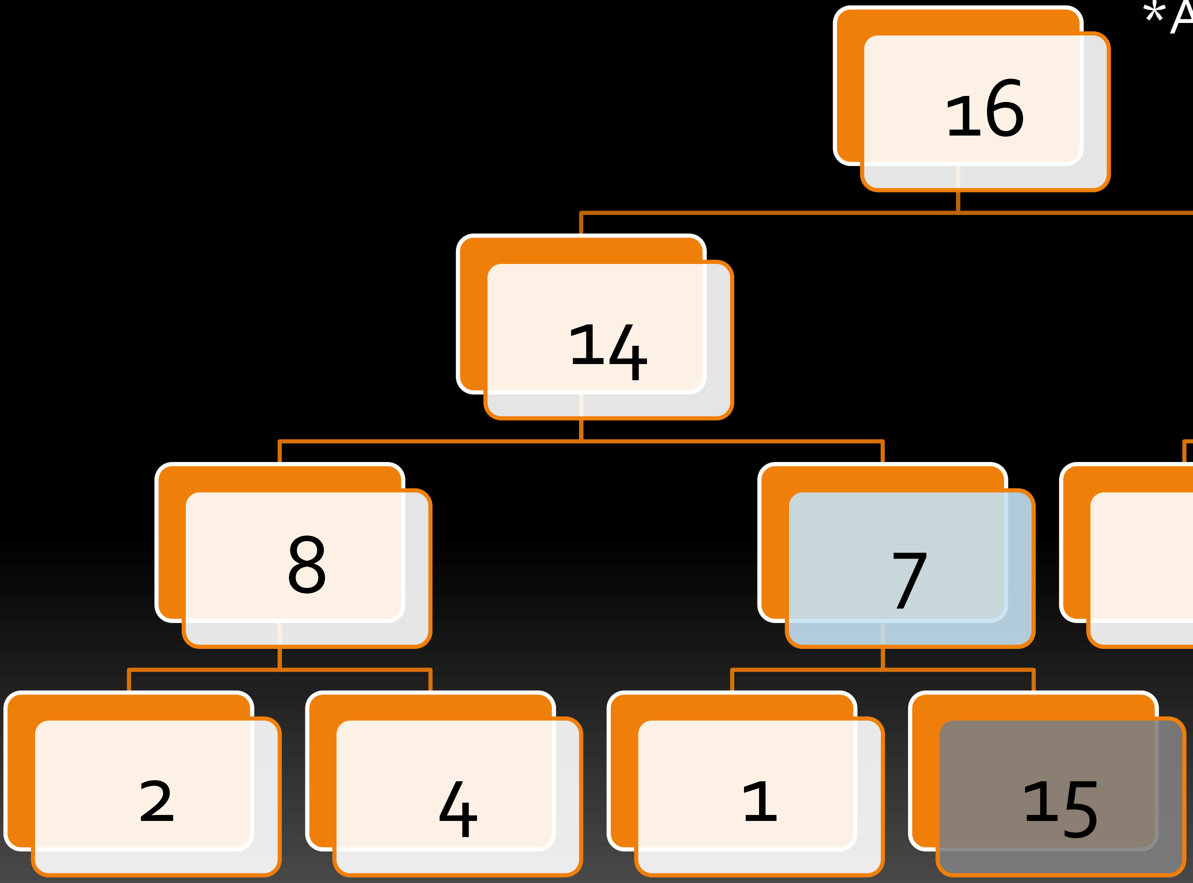
Insertion algorithm:

```
Function HEAP_INSERT(A,key)
heap_size[A] = heap_size[A] + 1
i = heap_size[A]
while i > 1 and A[PARENT(i)] < key
     A[i] = A[PARENT(i)]
         i = PARENT(i)
A[i] <- key
```

Priority Queue: insert example

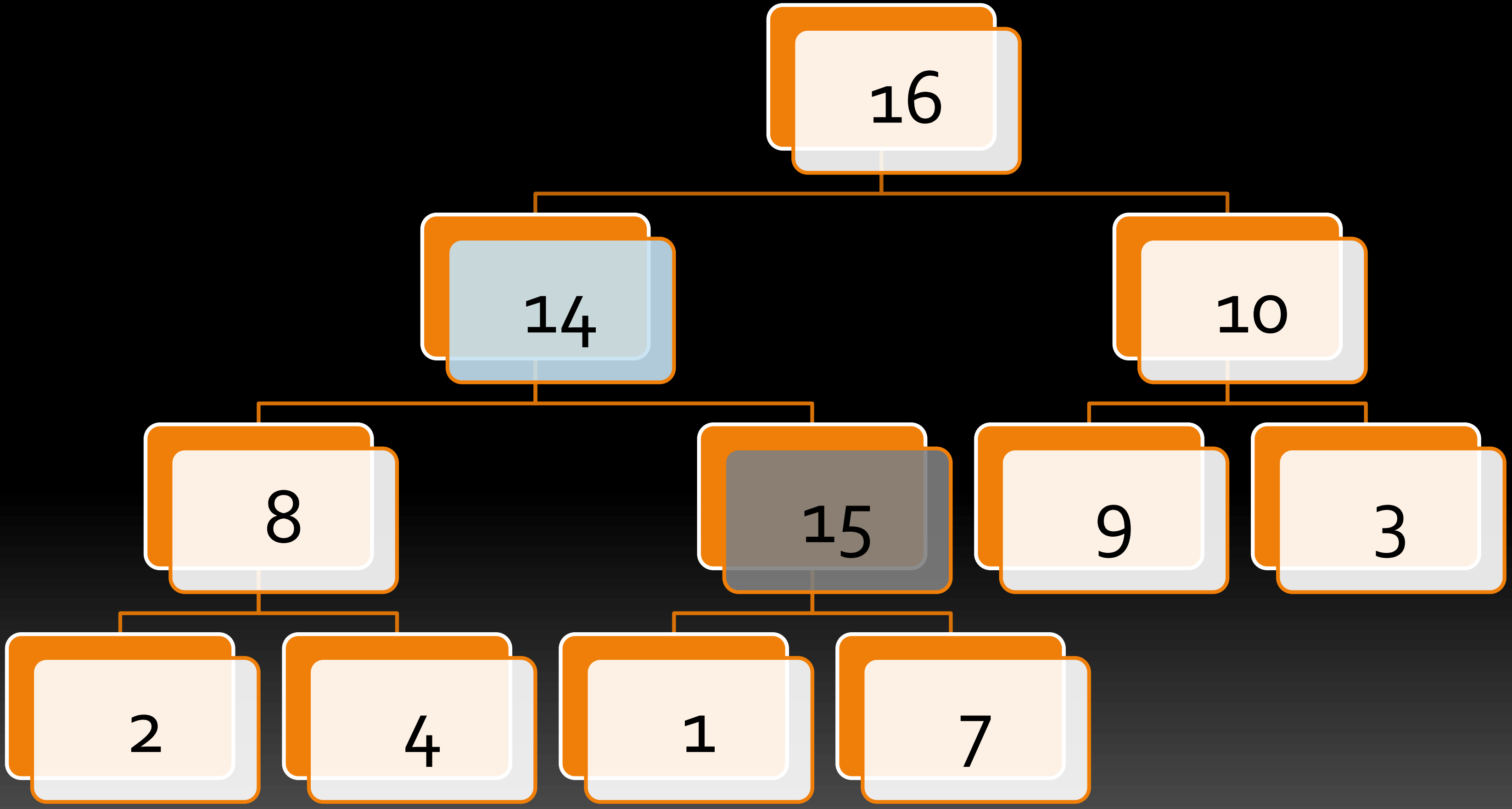


Priority Queue: insert example

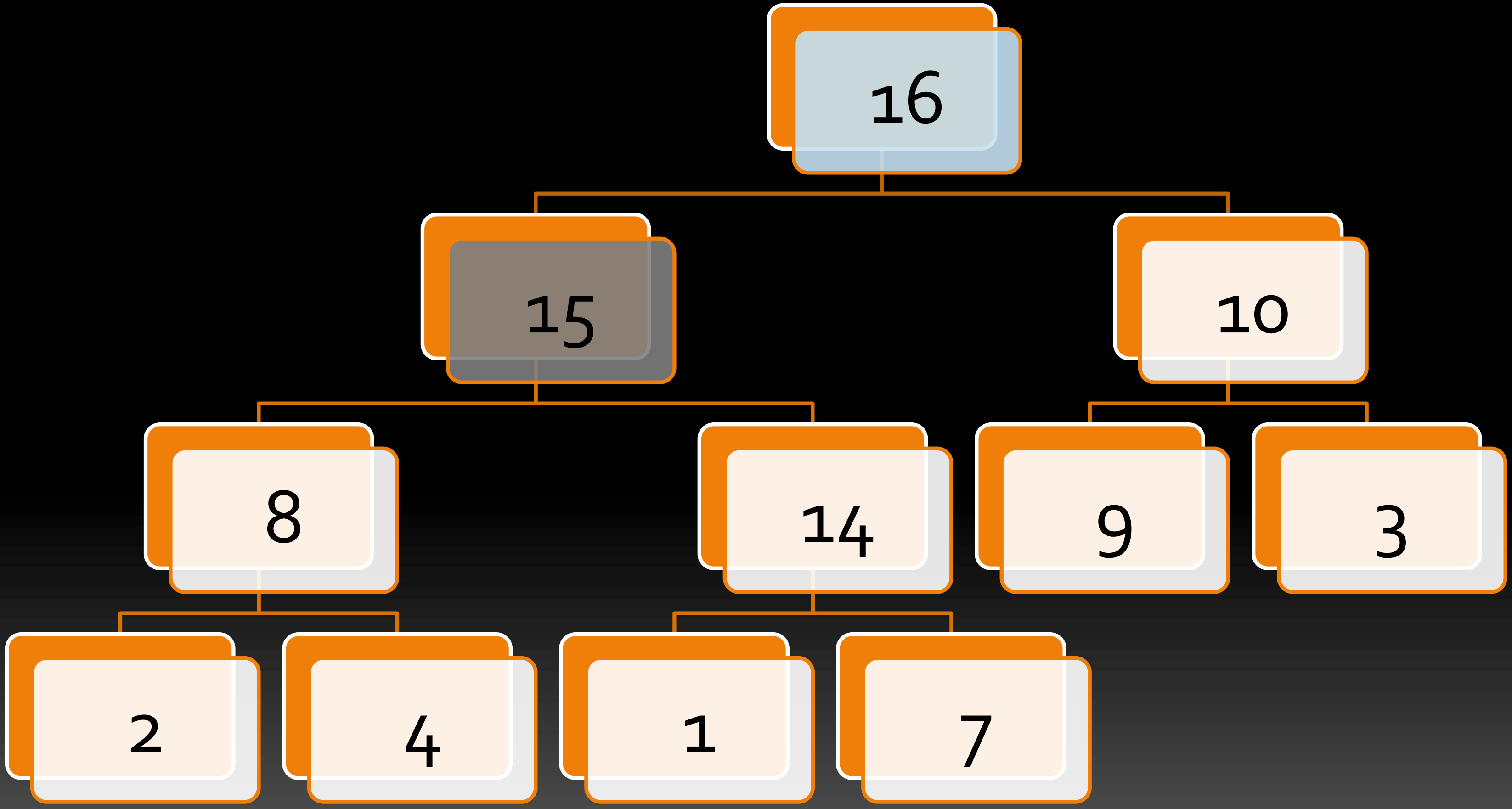


*A[1] won't be changed

Priority Queue: insert example

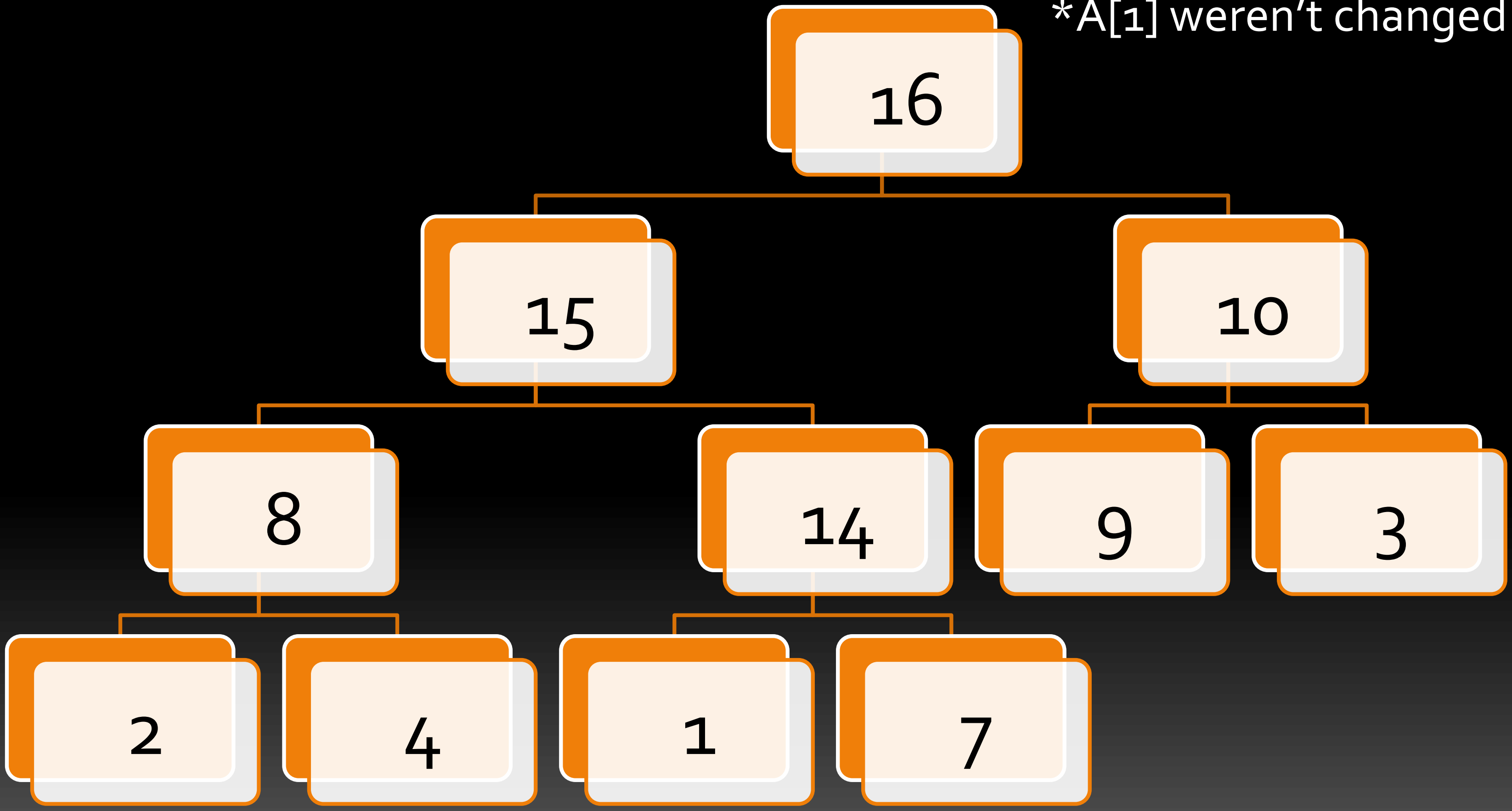


Priority Queue: insert example



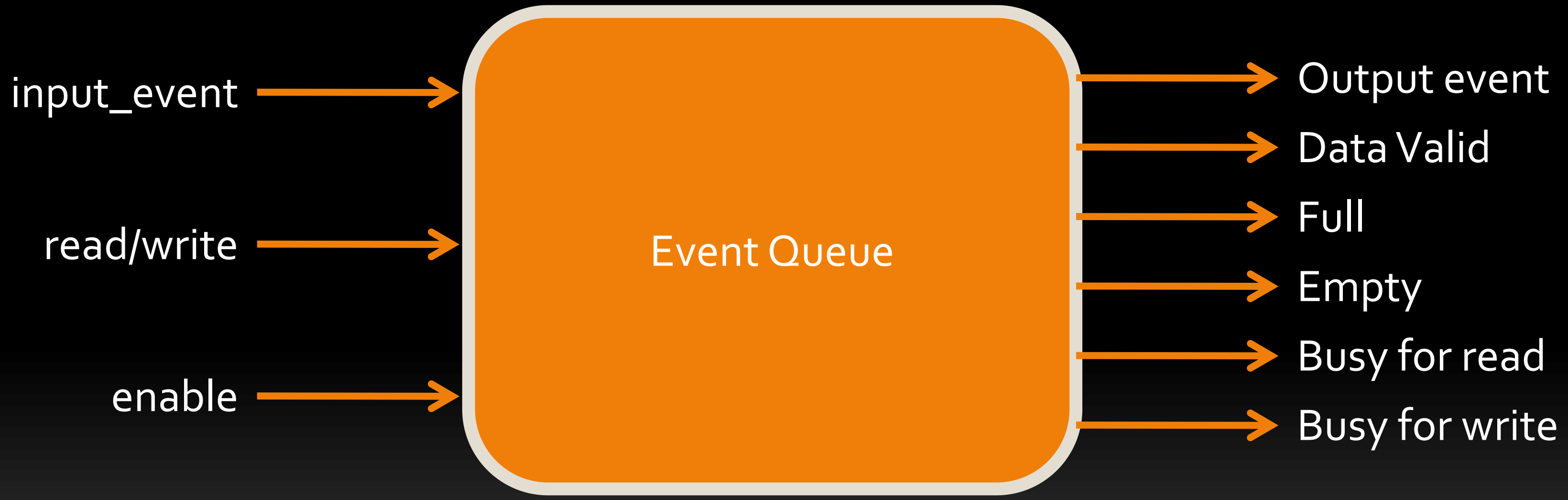
Priority Queue: insert example

*A[1] weren’t changed



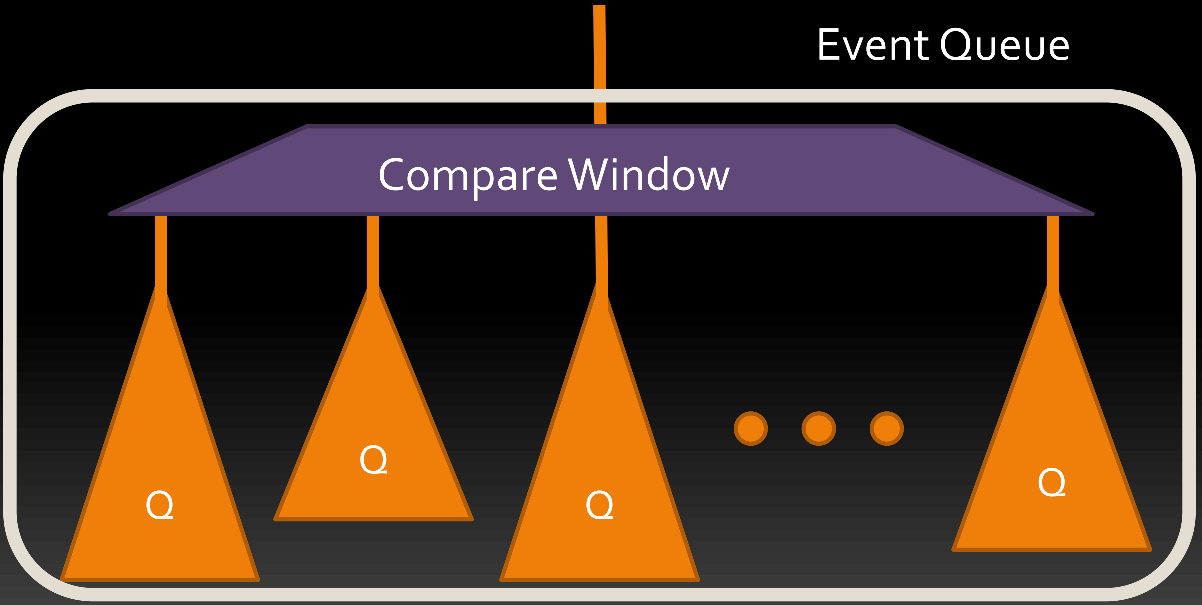
Event Queue in Turbosim Interface

- Ports assignment:



Event Queue in Turbosim

- Event Queue implementation:
    - Many parallel priority queues (Q)



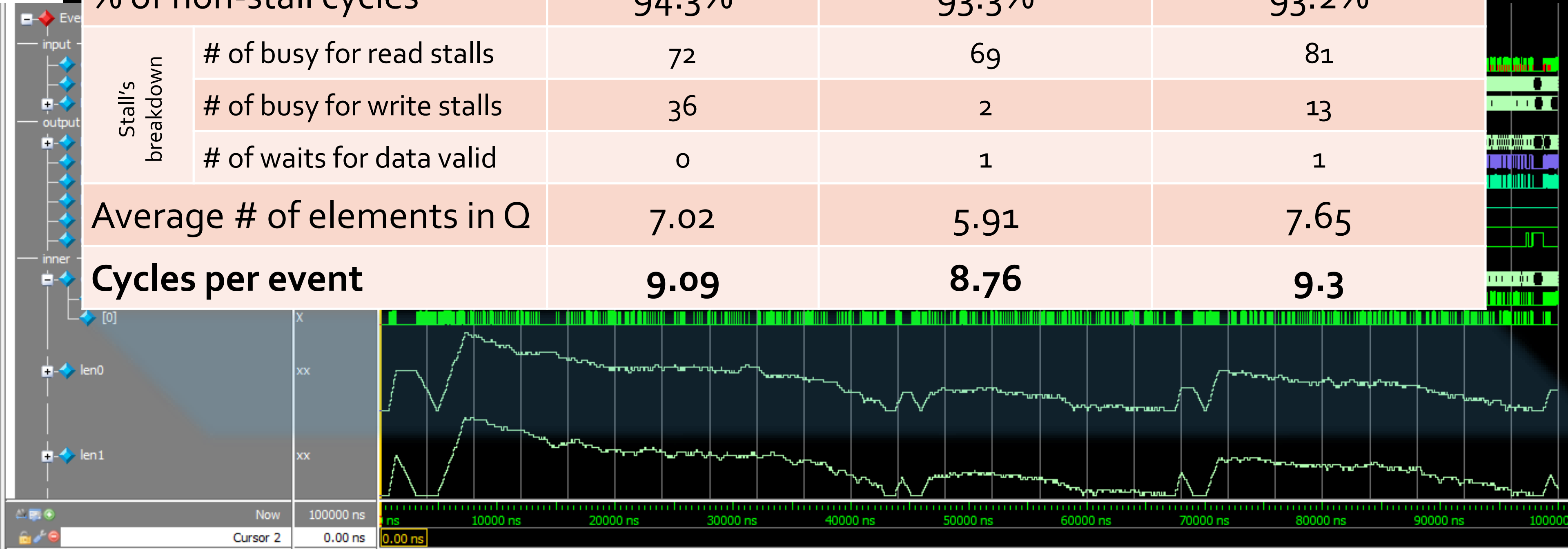
Event Queue in Turbosim

- Uses adopted algorithms
  - No recursive calls
  - Parallel read and comparison of heap nodes
- Maintains HI performance:
  - Smallest event is always on top of the Q
  - dv changes only when income event smaller then current top event
  - dv goes HI when Q[1] is valid, mean while Q is still been sorted
  - New event is inserted into shortest Q for load balance

Event Queue performance

Event Queue consist of 2-Queues

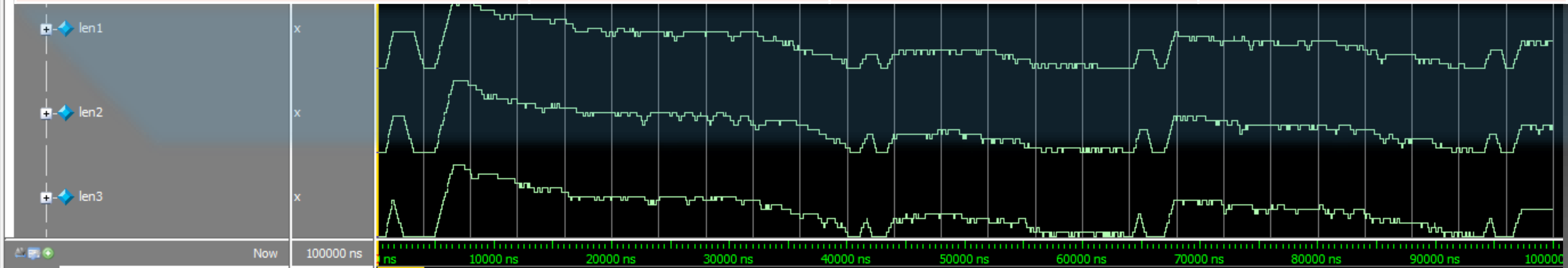
DUT add_1 iteration:		#1	#2	#3
# of cycles to solve		*1894*	1071	1394
Total # of stalls		108	*72*	*95*
% of non-stall cycles		94.3%	93.3%	93.2%
Stall's breakdown	# of busy for read stalls	72	69	81
	# of busy for write stalls	36	2	13
	# of waits for data valid	0	1	1
Average # of elements in Q		7.02	5.91	7.65
Cycles per event		**9.09**	**8.76**	**9.3**



Event Queue performance (2)

Event Queue consist of 4-Qs implementation:

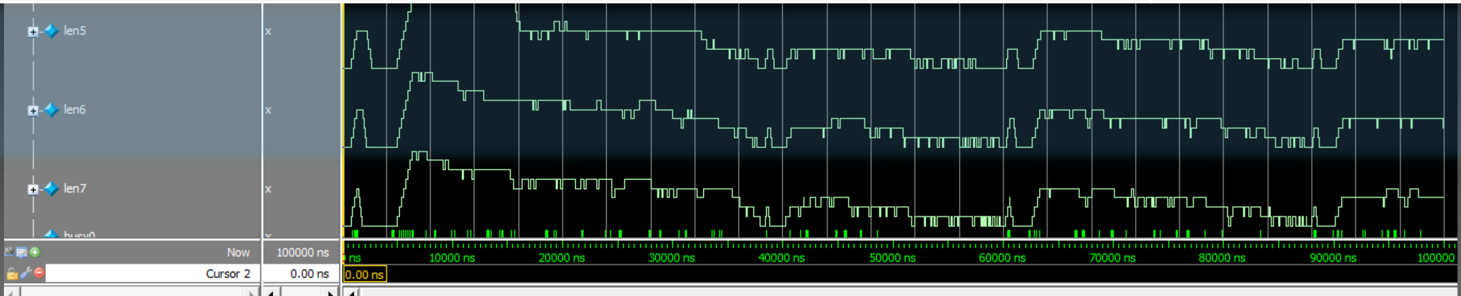
DUT add_1 iteration:		#1	#2	#3
# of cycles to solve		*1835*	1058	1343
Total # of stalls		49	*59*	*44*
% of non-stall cycles		97.3%	94.4%	96.7%
Stall's breakdown	# of busy for read stalls	49	59	43
	# of busy for write stalls	0	0	0
	# of waits for data valid	0	0	1
Average # of elements in Q		7.02	5.95	7.69
Cycles per event		**8.81**	**8.66**	**8.96**



Event Queue performance (3)

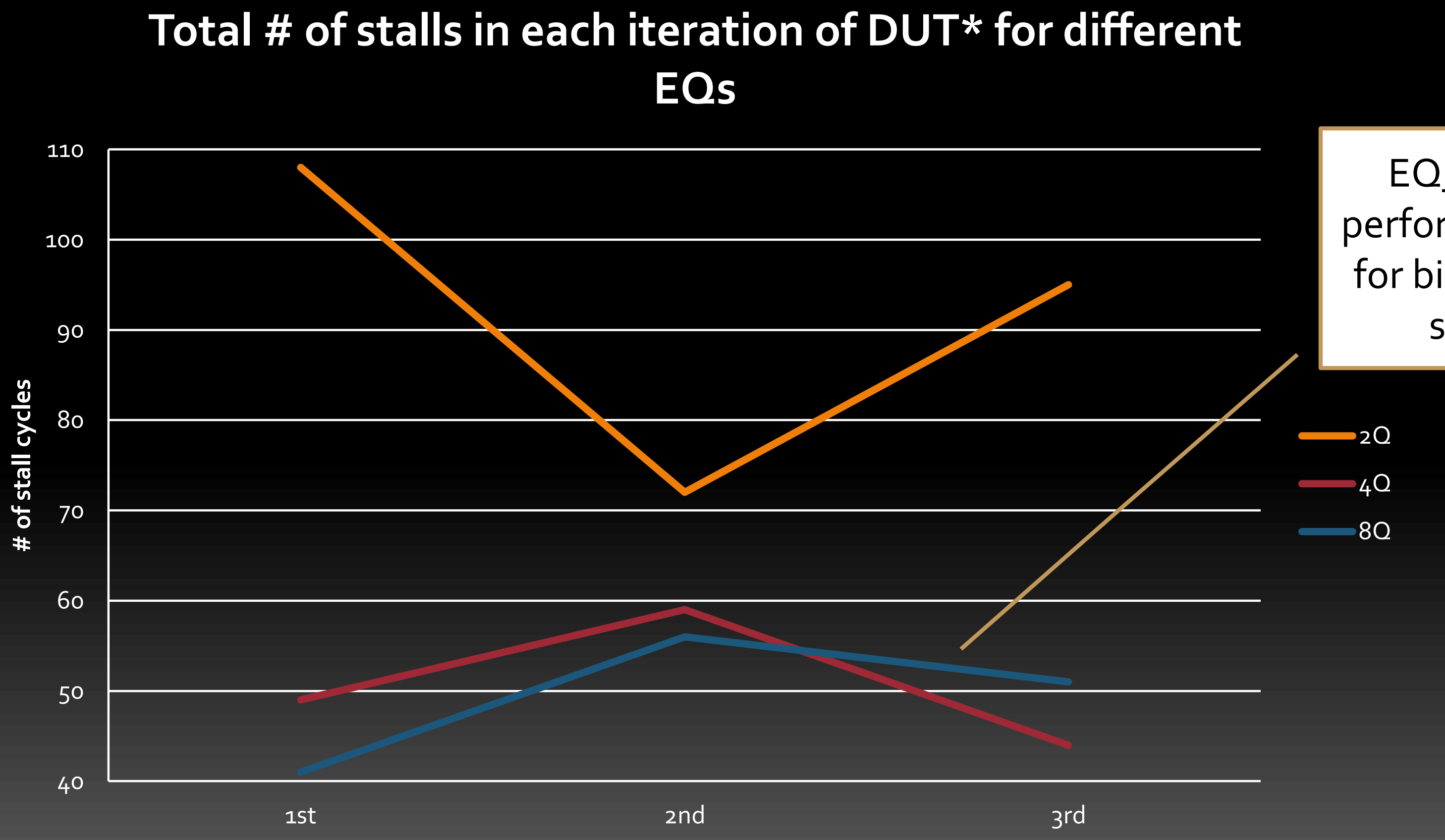
Event Queue consist of 8-Qs implementation:

DUT add_1 iteration:		#1	#2	#3
# of cycles to solve		*1827*	1055	1350
Total # of stalls		41	*56*	*51*
% of non-stall cycles		97.8%	94.7%	96.2%
Stall's breakdown	# of busy for read stalls	41	56	50
	# of busy for write stalls	0	0	0
	# of waits for data valid	0	0	1
Average # of elements in Q		7.00	5.93	7.67
Cycles per event		**8.77**	**8.63**	**9.01**



Event Queue performance (4)

- Comparison of all EQs implementation:

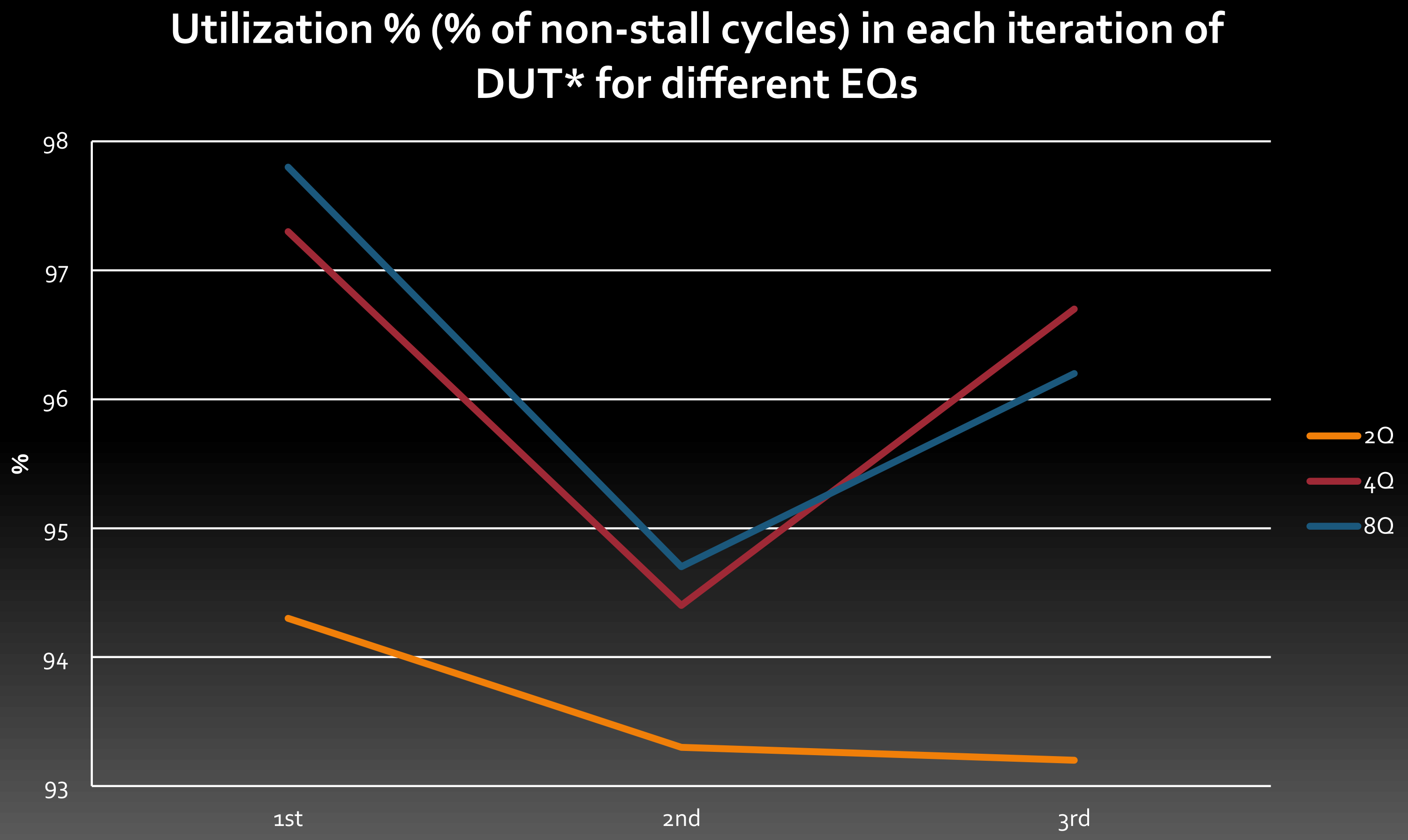


EQ_8 will perform better for bigger EQ sizes

* DUT add_1

Event Queue performance (4)

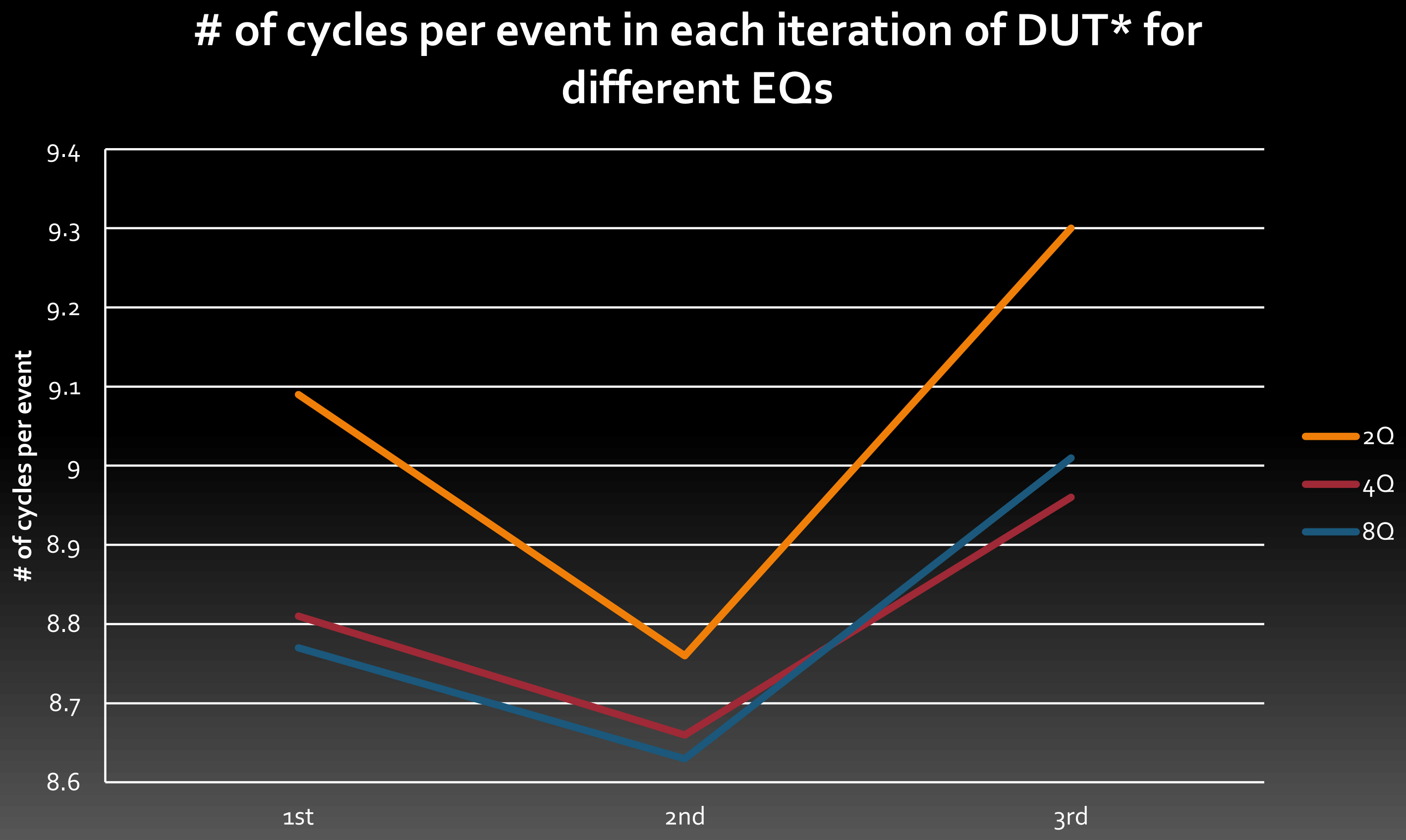
- Comparison of all EQs implementation:



* DUT add_1

Event Queue performance (4)

- Comparison of all EQs implementation:



* DUT add_1

C.T.U SET

C.T.U SET

Goal:

- Store and retrieve "***Cells To Update***" without allowing repeated entries.

Implementation:

- Set data structure using fifo and ram.
  - Allows back 2 back read and write.

CTU SET (1)

Set Definition*:

- Set is a finite abstract data structure that can store certain values, without any particular order, and no repeated values.

Implementation:

- RAM: represents mathematical finite set. Each element is marked there to avoid duplicated entries.
- FIFO: stores (not duplicated) input entries for easy extraction (B2B).

* wikipedia

CTU SET:: insertion

Utilize 2 stages pipe line:

A. Read from RAM whether pending event is already there.

B. If entry is missing insert it into fifo. Other wise do nothing (entry is already in fifo).

clk	1	2	3	4	5	6
wr	0	1	1	1	1	0
pipe state		A	B	A	B	
pipe stage			A	B	A	B

CTU SET:: extraction

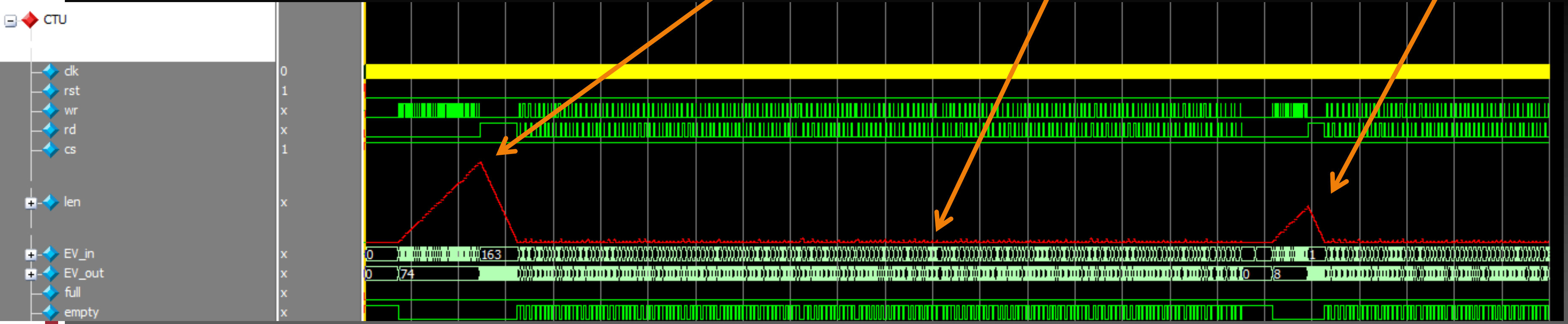
Extraction is done in one cycle

- Mark appropriate RAM cell that entry were popped from fifo (cell address is taken from fifo output that is always valid, if it’s not empty).
- Issue read from fifo.

CTU SET:: implementation issues

FIFO

- Is implemented using registers (and not RAM), in order to get output valid data immediately after it were written.
- This motivated by fifo's small size (128 in our design).



78 @ 1st iteration start

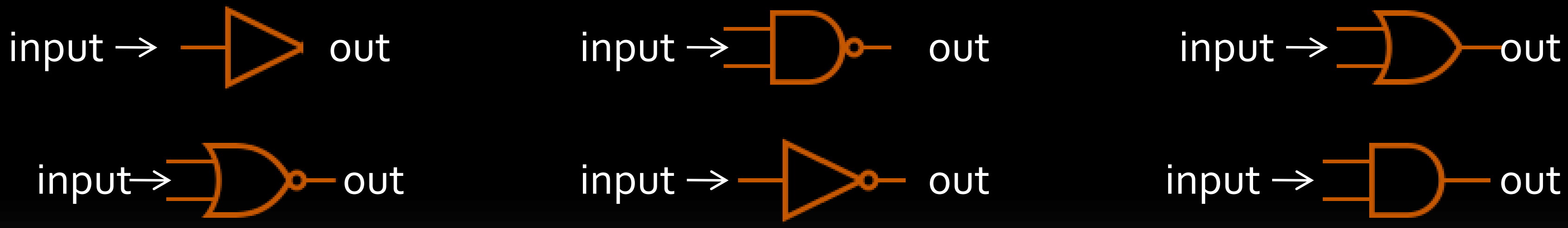
0-4 @ normal run

35 @ 2nd iteration start

Tests

Gates check

- Checks the six types of supported gates.



Gates check results:

- All six gates work

Average cycle per event(63 runs):6.2

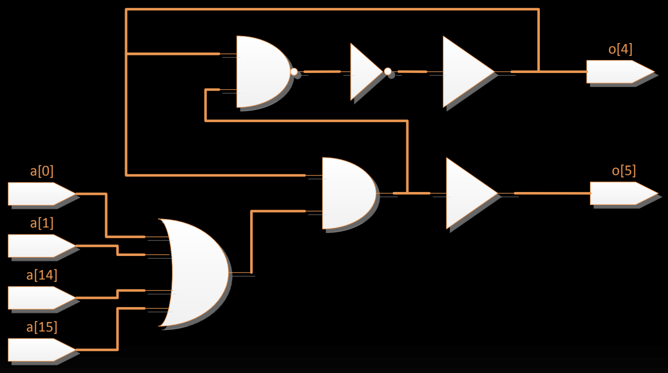
```
# start iteration    53 : a = 137f, b =5af9
# ===  ref_vcd      time     0 ps, value 0, net name a[11]
# ===  ref_vcd      time     0 ps, value 1, net name a[12]
# ===  ref_vcd      time     0 ps, value 0, net name a[13]
# ===  ref_vcd      time     0 ps, value 0, net name a[14]
# ===  ref_vcd      time     0 ps, value 1, net name a[1]
# ===  ref_vcd      time     0 ps, value 1, net name a[5]
# ===  ref_vcd      time     0 ps, value 1, net name a[6]
# ===  ref_vcd      time     0 ps, value 1, net name a[8]
# ===  ref_vcd      time     0 ps, value 1, net name b[0]
# ===  ref_vcd      time     0 ps, value 0, net name b[1]
# ===  ref_vcd      time   800 ps, value 1, net name o[5]
# ===  ref_vcd      time  1000 ps, value 1, net name o[0]
# === acc_vcd       time     0 ps, value 0, net name a[11]
# === acc_vcd       time     0 ps, value 1, net name b[0]
# === acc_vcd       time     0 ps, value 1, net name a[1]
# === acc_vcd       time     0 ps, value 1, net name a[12]
# === acc_vcd       time     0 ps, value 0, net name b[1]
# === acc_vcd       time     0 ps, value 1, net name a[5]
# === acc_vcd       time     0 ps, value 0, net name a[13]
# === acc_vcd       time     0 ps, value 1, net name a[6]
# === acc_vcd       time     0 ps, value 0, net name a[14]
# === acc_vcd       time     0 ps, value 1, net name a[8]
# === acc_vcd       time   800 ps, value 1, net name o[5]
# === acc_vcd       time  1000 ps, value 1, net name o[0]
# end   iteration    53 : turbosim solved in    72 clock cycles
# Turbosim change count:          12, Modelsim change count:          12
# Cycles per event: 6.000000
```

loop check #1

- Checks loops that stabilize in to a single value

```
module loop_test1 ( o, a, b );
  output [5:0] o;
  input [15:0] a;
  input [1:0] b;
  wire n0 ,n1,n3,n4;

  nand   #(1.000) U4 ( n0,n1,n4 );
  not    #(1.000) U7 ( n1, n0 );
  and    #(0.800) U8 ( n4,n1,n3 );
  or     #(1.000) U5 ( n3, a[14], a[15],a[0],a[1]);
  buf    #(0.800) U6 ( o[5], n4  );
  buf    #(0.800) U9 ( o[4], n1  );
endmodule
```



loop check #1 results:

- Average cycle per event(208 runs):10.42

```
# start iteration   187 : a = 5705, b =846e
# ===  ref_vcd      time      0 ps, value 1, net name a[14]
# ===  ref_vcd      time      0 ps, value 0, net name a[1]
# === acc_vcd       time      0 ps, value 1, net name a[14]
# === acc_vcd       time      0 ps, value 0, net name a[1]
# end   iteration   187 : turbosim solved in     19 clock cycles
# Turbosim change count:           2, Modelsim change count:            2
# Cycles per event: 9.500000

# start iteration   188 : a = 62eb, b =2d8a
# ===  ref_vcd      time      0 ps, value 1, net name a[1]
# === acc_vcd       time      0 ps, value 1, net name a[1]
# end   iteration   188 : turbosim solved in     13 clock cycles
# Turbosim change count:           1, Modelsim change count:            1
# Cycles per event: 13.000000

# start iteration   189 : a = 446e, b =4868
# ===  ref_vcd      time      0 ps, value 0, net name a[0]
# === acc_vcd       time      0 ps, value 0, net name a[0]
# end   iteration   189 : turbosim solved in     13 clock cycles
# Turbosim change count:           1, Modelsim change count:            1
# Cycles per event: 13.000000
```

loop check 2

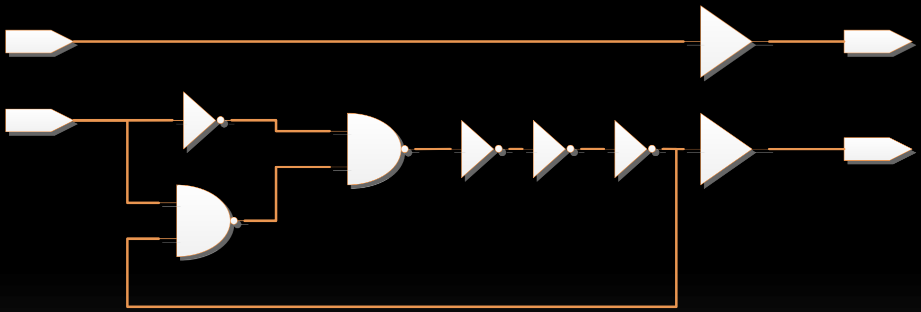
- Checks a loops that does not stabilize in to a single value

```
module loop_test2(o, a, b);
  output  [1:0] o;
  input         a;
  input         b;

  wire n0 ,n1,n2,n3,n4,n5;
  not    #(9.999) U10 (n5,a);
  nand   #(1.000) U1 ( n1, n0,a );

  nand    #(1.000) U2 ( n2, n1,n5 );
  not     #(1.000) U3 ( n3, n2 );
  not     #(1.000) U4 ( n4,n3 );
  not     #(1.000) U7 ( n0, n4 );

  buf     #(1.000) U8 (o[0], b );
  buf     #(1.000) U9 (o[1], n3 );
endmodule
```



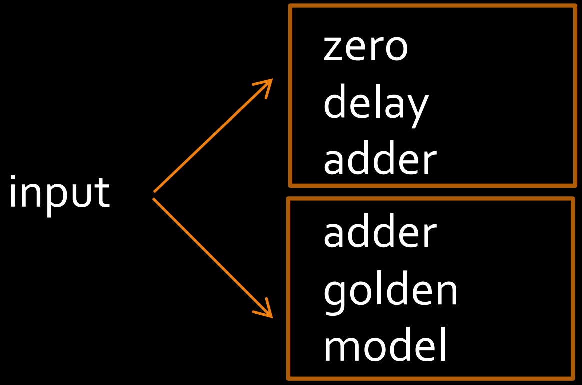
loop check 2 results:

- Average cycle per event(63 runs):8.6

```
# start iteration    58 : a = 0e67, b =36d3
# ===   ref_vcd      time     0 ps, value 1, net name a
# ===   ref_vcd      time  1000 ps, value 0, net name n1
# ===   ref_vcd      time  2000 ps, value 1, net name n2
# ===   ref_vcd      time  3000 ps, value 0, net name n3
# ===   ref_vcd      time  4000 ps, value 1, net name n4
# ===   ref_vcd      time  4000 ps, value 0, net name o[1]
# ===   ref_vcd      time  5000 ps, value 0, net name n0
# ===   ref_vcd      time  6000 ps, value 1, net name n1
# ===   ref_vcd      time  7000 ps, value 0, net name n2
# ===   ref_vcd      time  8000 ps, value 1, net name n3
# ===   ref_vcd      time  9000 ps, value 0, net name n4
# ===   ref_vcd      time  9000 ps, value 1, net name o[1]
# ===   ref_vcd      time  9999 ps, value 0, net name n5
# ===   ref_vcd      time 10000 ps, value 1, net name n0
# ===   ref_vcd      time 10999 ps, value 1, net name n2
# ===   ref_vcd      time 11000 ps, value 0, net name n1
# ===   ref_vcd      time 11999 ps, value 0, net name n3
# ===   ref_vcd      time 12999 ps, value 1, net name n4
# ===   ref_vcd      time 12999 ps, value 0, net name o[1]
# ===   ref_vcd      time 13999 ps, value 0, net name n0
# ===   ref_vcd      time 14999 ps, value 1, net name n1
# === acc_vcd       time     0 ps, value 1, net name a
# === acc_vcd       time  1000 ps, value 0, net name n1
# === acc_vcd       time  2000 ps, value 1, net name n2
# === acc_vcd       time  3000 ps, value 0, net name n3
# === acc_vcd       time  4000 ps, value 0, net name o[1]
# === acc_vcd       time  4000 ps, value 1, net name n4
# === acc_vcd       time  5000 ps, value 0, net name n0
# === acc_vcd       time  6000 ps, value 1, net name n1
# === acc_vcd       time  7000 ps, value 0, net name n2
# === acc_vcd       time  8000 ps, value 1, net name n3
# === acc_vcd       time  9000 ps, value 1, net name o[1]
# === acc_vcd       time  9000 ps, value 0, net name n4
# === acc_vcd       time  9999 ps, value 0, net name n5
# === acc_vcd       time 10000 ps, value 1, net name n0
# === acc_vcd       time 10999 ps, value 1, net name n2
# === acc_vcd       time 11000 ps, value 0, net name n1
# === acc_vcd       time 11999 ps, value 0, net name n3
# === acc_vcd       time 12999 ps, value 0, net name o[1]
# === acc_vcd       time 12999 ps, value 1, net name n4
# === acc_vcd       time 13999 ps, value 0, net name n0
# === acc_vcd       time 14999 ps, value 1, net name n1
# end   iteration    58 : turbosim solved in   182 clock cycles
# Turbosim change count:          21, Modelsim change count:          21
# Cycles per event: 8.666667
```

Big module check

- Check zero delay adder



- Results:
- Turbosim change count: 535
- Modelsim change count:535

- Average cycle per event(63 runs):6.35

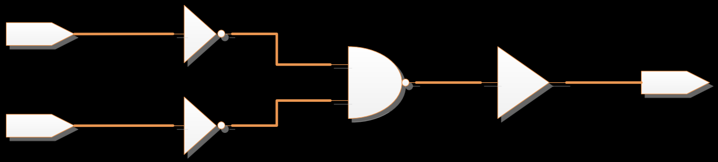
Spec Check

- Q: What happens when we have 2 events for the same object simultaneously?

```
module spec_check ( o, a, b );
  output [2:0] o;
  input [3:0] a;
  input [1:0] b;

  wire n0,n1;

  not    #(1.200) U1 ( n0, b[1] );
  not    #(9.000) U2 ( n1, a[2] );
  nand   #(9.999) U7 ( o[0], n0, n1 );
endmodule
```



The answer from Verilog spec:

- *"In situations where a right-hand operand changes before a previous change has had time to propagate to the left-hand side, then the following steps are taken:*
  - *The value of the right-hand expression is evaluated.*
  - *If this right-hand side value differs from the value currently scheduled to propagate to the left-hand side, then the currently scheduled propagation event is descheduled.*
  - *If the new right-hand side value equals the current left-hand side value, no event is scheduled.*
  - *If the new right-hand side value differs from the current left-hand side value, a delay is calculated in the standard way using the current value of the left-hand side, the newly calculated value of the right-hand side, and the delays indicated on the statement; a new propagation event is then scheduled to occur delay time units in the future.*

```
# start iteration   333 : a = 342f, b =88e9
# ===  ref_vcd      time     0 ps, value 1, net name a[2]
# ===  ref_vcd      time     0 ps, value 0, net name b[1]
# ===  ref_vcd      time  1200 ps, value 1, net name n0
# ===  ref_vcd      time  9000 ps, value 0, net name n1
# === acc_vcd       time     0 ps, value 1, net name a[2]
# === acc_vcd       time     0 ps, value 0, net name b[1]
# === acc_vcd       time  1200 ps, value 1, net name n0
# === acc_vcd       time  9000 ps, value 0, net name n1
# end   iteration   333 : turbosim solved in    44 clock cycles
# Turbosim change count:            4, Modelsim change count:            4
 average cycles per event: 9.2
```

Summary

- We successfully build gate-level simulation accelerator - Turbosim
- Turbosim cycle per event:
  - Best case: ~6.3 (simulating zero delay logic)
  - Mean case: ~9 (arbitrary delay logic)
- Max speedup v.s. SW simulator:
  - X1000 v.s. Modelsim to simulate same design*

*DUT: add1

The end