

Hardware Design and Functional Programming: Still Interesting after All These Years

Mary Sheeran
Chalmers

Hardware Description Languages

IEEE TRANSACTIONS ON COMPUTERS, VOL. C-17, NO. 9, SEPTEMBER 1968

A Digital System Design Language (DDL)

JAMES R. DULEY AND DONALD L. DIETMEYER

SPECIFYING, documenting, and controlling the design of digital systems are problems of increasing severity as such systems continue to grow in size and complexity. Wilkes and Stringer [2] first recognized that a suitable design language could greatly reduce the magnitude of these problems and lead to a complete, precise, yet concise description of digital systems. Unfortunately, their contribution is mostly oriented toward the machine that they were developing at the time and is not generally useful.

Hardware Description Languages

[2] M. V. Wilkes and J. B. Stringer, "Micro-programming and the design of the control circuits in an electronic digital computer," Proc. Cambridge Phil. Socs., vol. 44, pt 2, pp. 230-238, April 1953.

Hardware Description Languages

Reed 1952

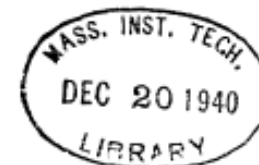
Symbolic synthesis of digital computers

Proc. ACM National Meeting (Toronto)

Hardware Description Languages

In an ideal sense a binary digital computer or what might be called more generally a Boolean machine is an **automatic operational filing system**...

This information is stored or recorded in sets of **elementary boxes or files, each containing one of the symbols 0 or 1**. This information is either transformed or used to change other files or itself as a function of the past contents of all files within the system. If the contents of all files within the system are constrained to change only at discrete points of time, say the points n ($n = 1, 2, 3, \dots$), then the machine may be termed a synchronous Boolean machine



A SYMBOLIC ANALYSIS
OF
RELAY AND SWITCHING CIRCUITS

by

Claude Elwood Shannon
B.S., University of Michigan
1936

Submitted in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE

from the
Massachusetts Institute of Technology
1940

Signature of Author _____

Department of Electrical Engineering, August 10, 1937

APL

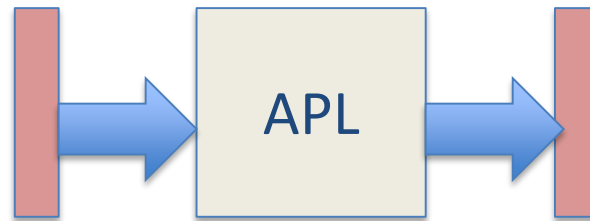
A formal description of SYSTEM/360

**by A. D. Falkoff, K. E. Iverson,
and E. H. Sussenguth**

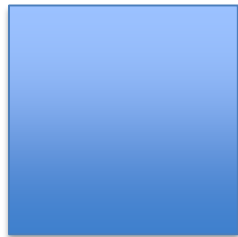
This paper presents a precise formal description of a complete computer system, the IBM SYSTEM/360. The description is functional: it describes the behavior of the machine as seen by the programmer, irrespective of any particular physical implementation, and expressly specifies the state of every register or facility accessible to the programmer for every moment of system operation at which this information is actually available.

IBM Systems Journal Vol 3, No. 3 1964

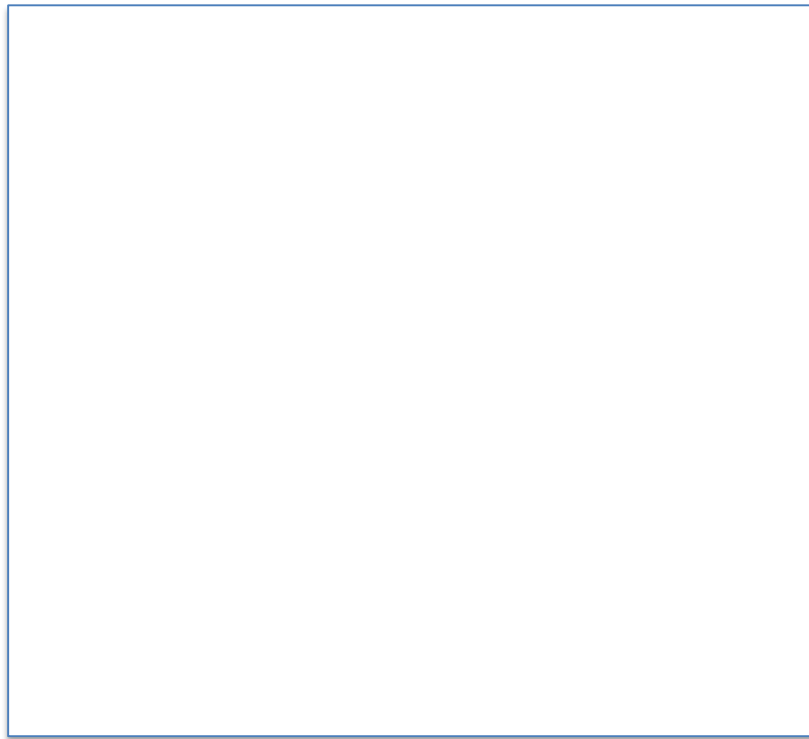
Quality of designs from an automatic logic generator (ALERT)
[Design Automation Workshop \(DAC'70\)](#)



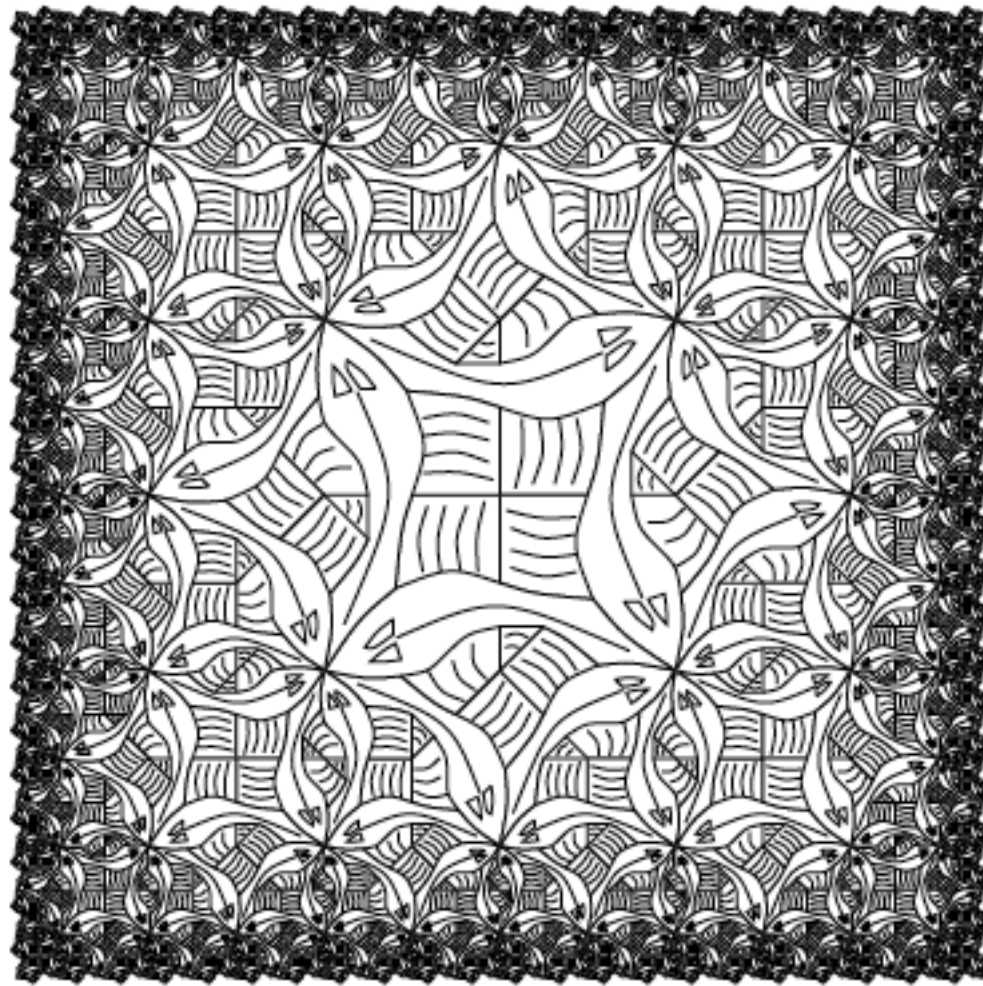
IBM 1800



Late 70s

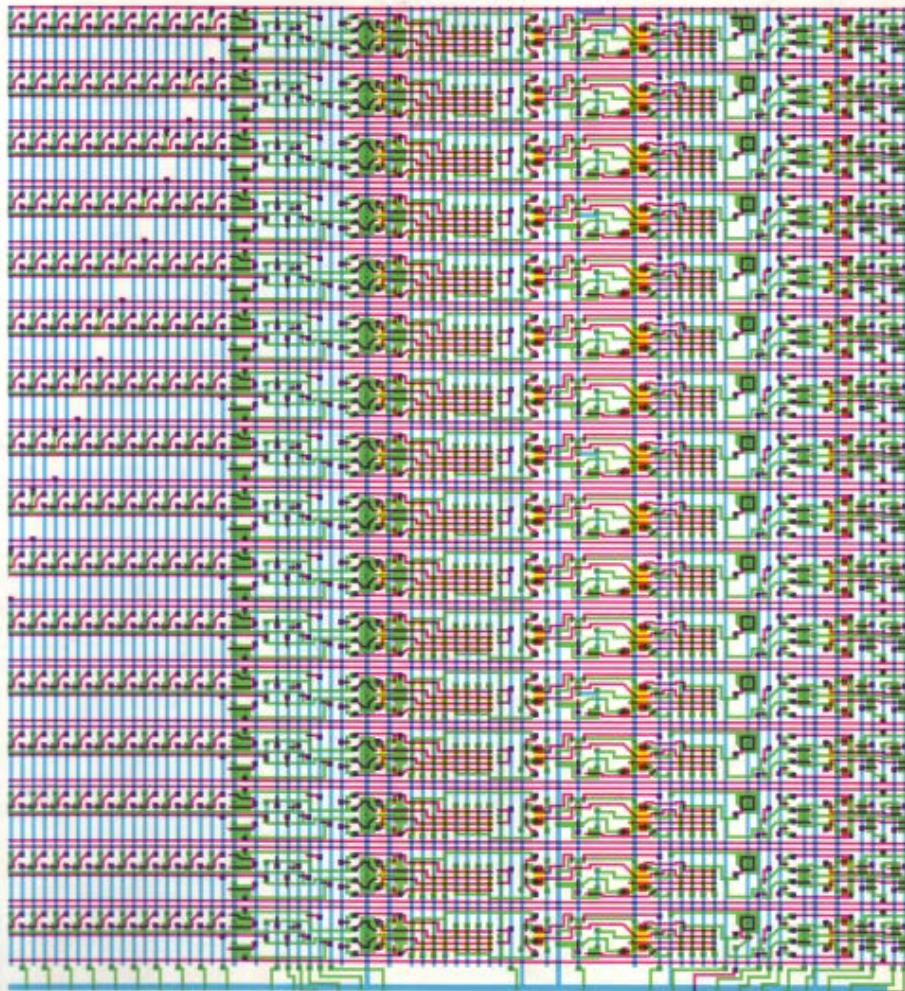


Henderson, Functional Geometry, 1982



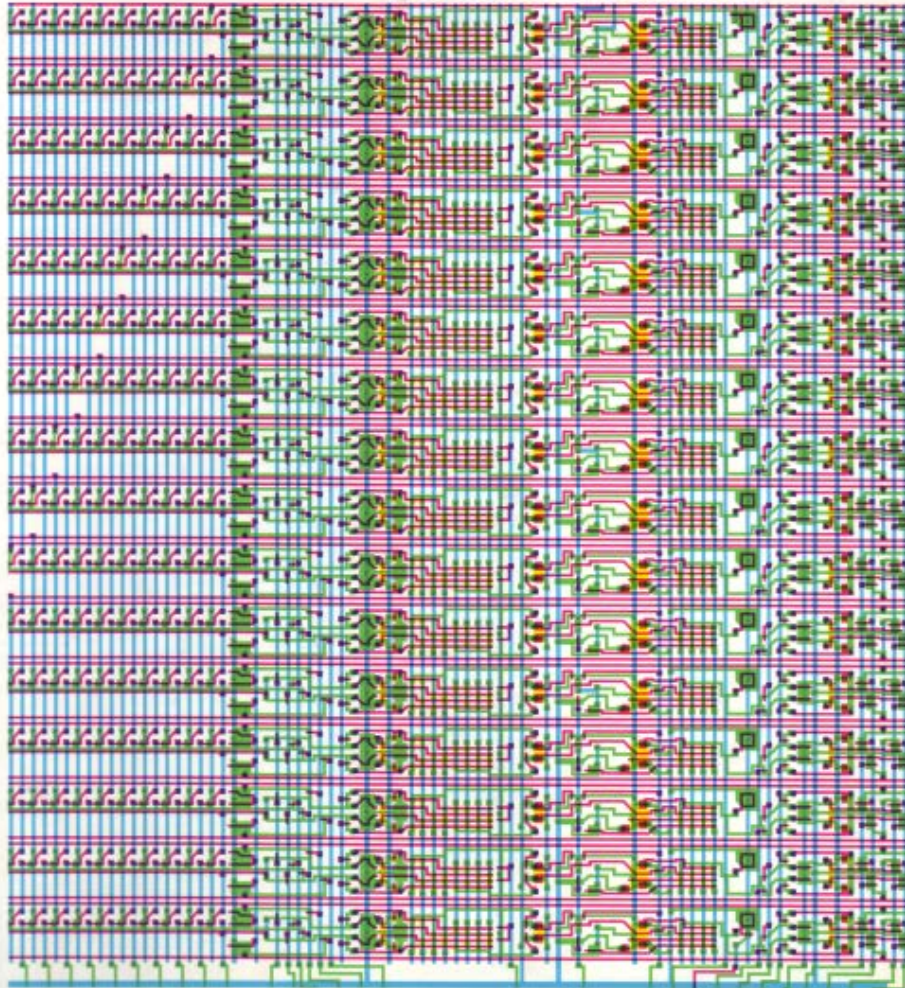
INTRODUCTION TO **VLSI** SYSTEMS

CARVER MEAD • LYNN CONWAY



INTRODUCTION TO **VLSI** SYSTEMS

CARVER MEAD • LYNN CONWAY

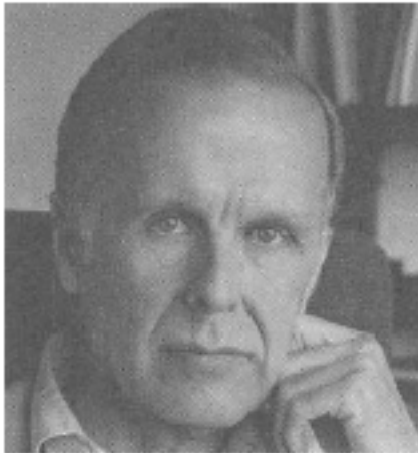


Hardcover

\$0.77

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose



Turing award 1977
[Paper 1978](#)

An alternative functional style of programming is founded on the use of combining forms for creating programs. ... Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages. ...

Associated with the functional style of programming is an algebra of programs whose variables range over programs and whose operations are combining forms.

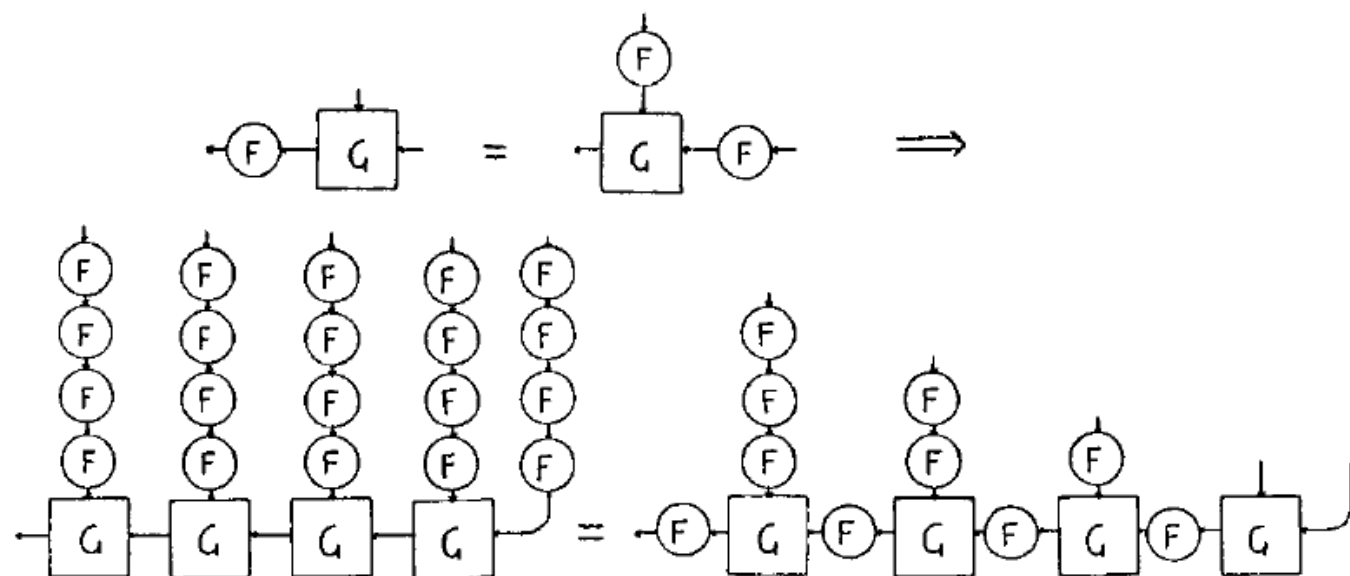
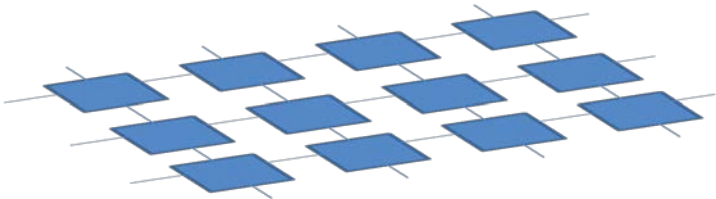
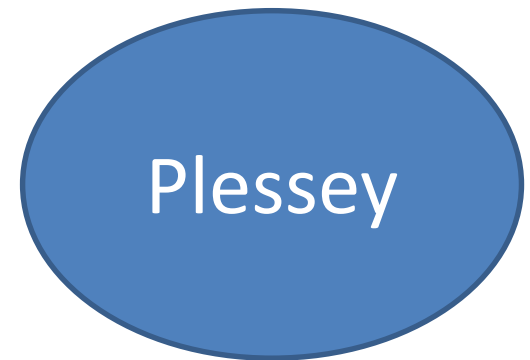


FIG 8. $F \cdot G = G \cdot @F \Rightarrow /G \cdot \langle \rangle F = /(F \cdot G) \cdot mL \rangle F \quad (/ \rangle)$

Users!



Users!



Plessey

Plessey designers write

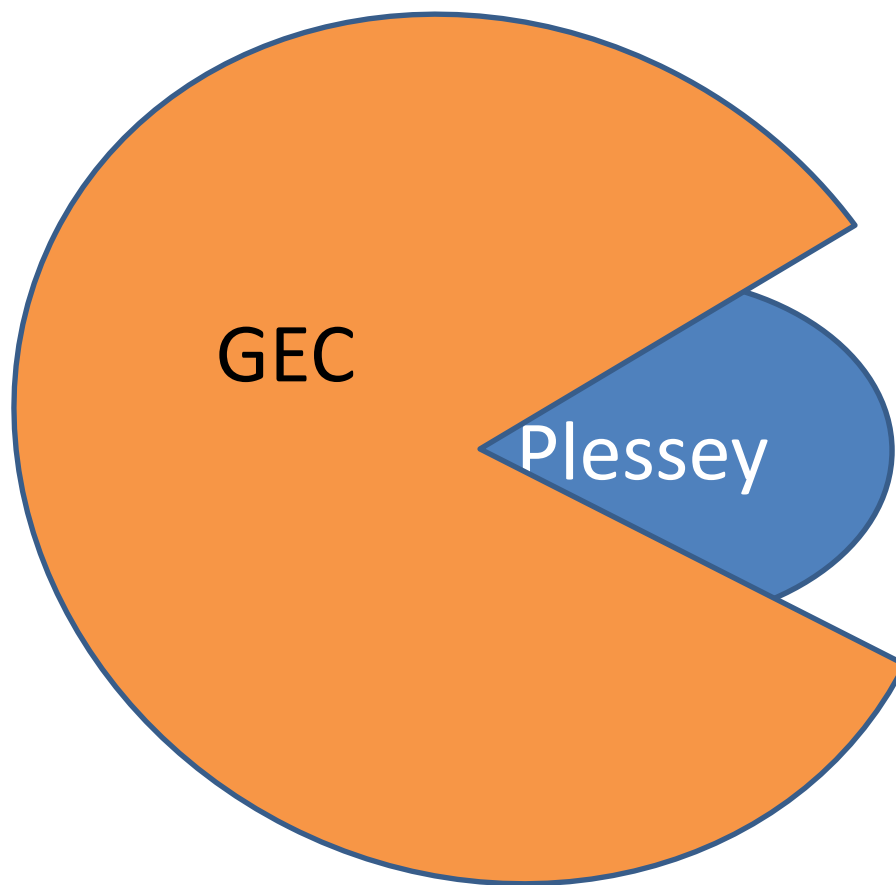
Using muFP, the array processing element was described in just one line of code and the complete array required four lines of muFP description. muFP enabled the effects of adding or moving data latches within the array to be assessed quickly. Since the results were in symbolic form it was clear where and when data within the results was input into the array making it simple to examine the data-flow within the array and change it as desired. This was found to be a very useful way to learn about the data dependencies within the array.

[...]

From the experience gained on the design, the most important consideration when designing array processors is to ensure that the processor input/output requirements can be met easily and without sacrificing array performance. The most difficult part of the design task is not the design of the computation units but the design of the data paths and associated storage devices. It is essential to have the right design tools to aid and improve the design process. Early use of tools to explore the flow of data within and around the array and to understand the data requirements of the array is important. muFP has been shown to be useful for this purpose.

Bhandal et al, An array processor for video picture motion estimation,
Systolic Array Processors, 1990, Prentice Hall

work with Plessey done by G. Jones and W. Luk





WIKIPEDIA
Den fria encyklopedin

REALITY

Kategori:Hårdvarubeskrivande språk

Artiklar i kategorin "Hårdvarubeskrivande språk"

Följande 2 sidor (av totalt 2) finns i denna kategori.

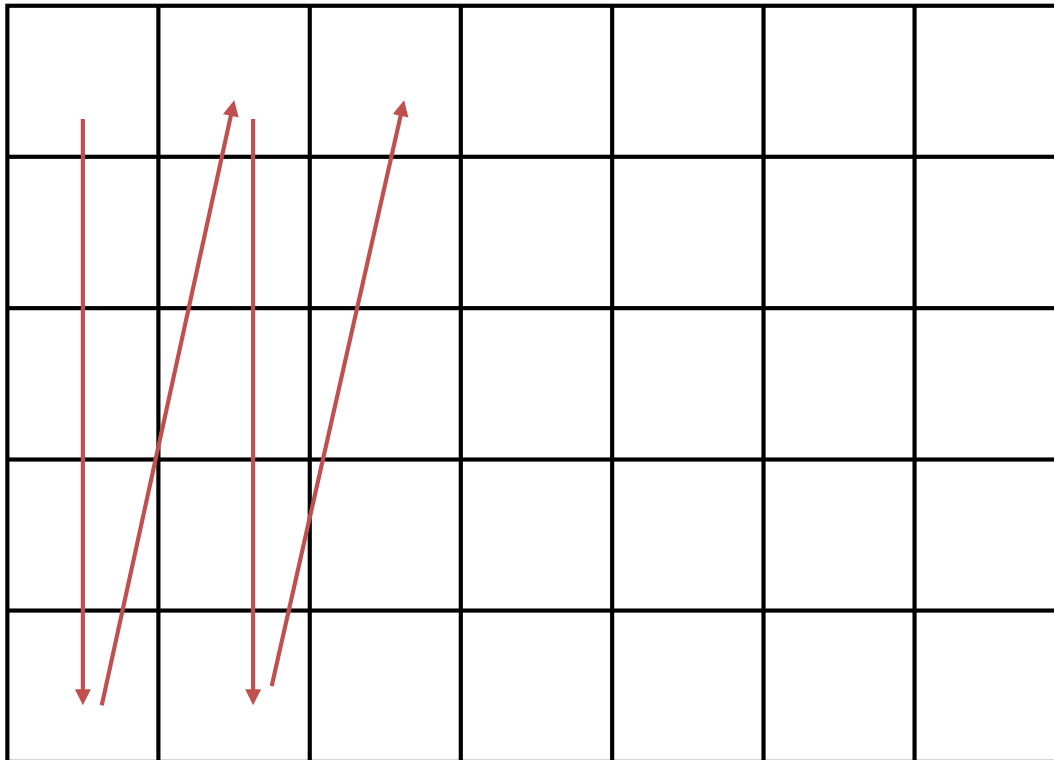
V

[Verilog](#)

[VHDL](#)

Algorithms in hardware

In pictures



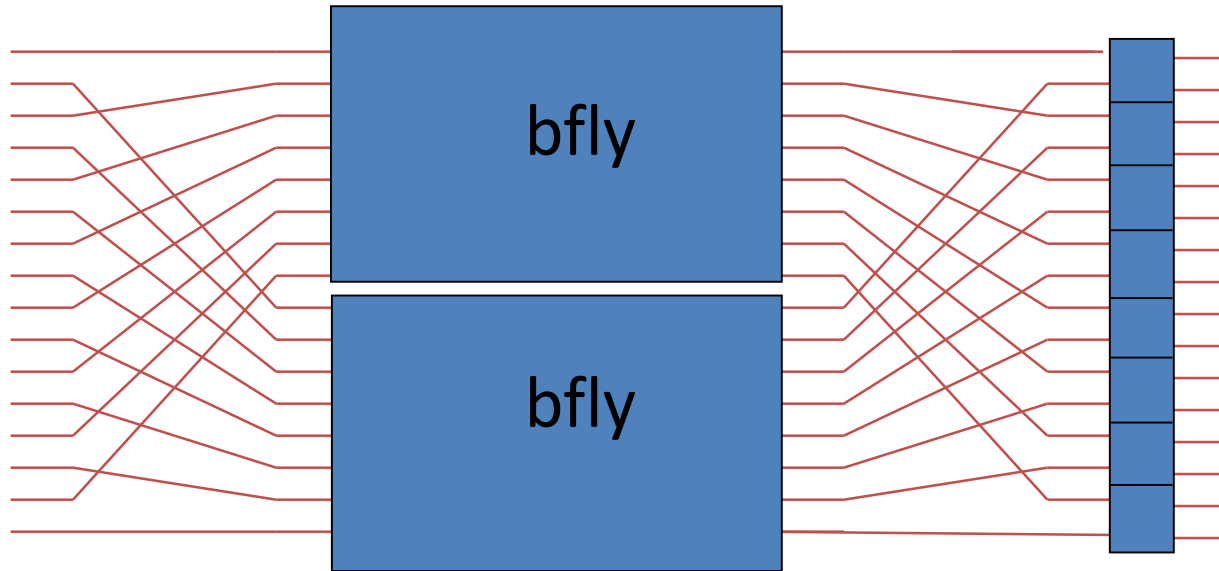
ilv

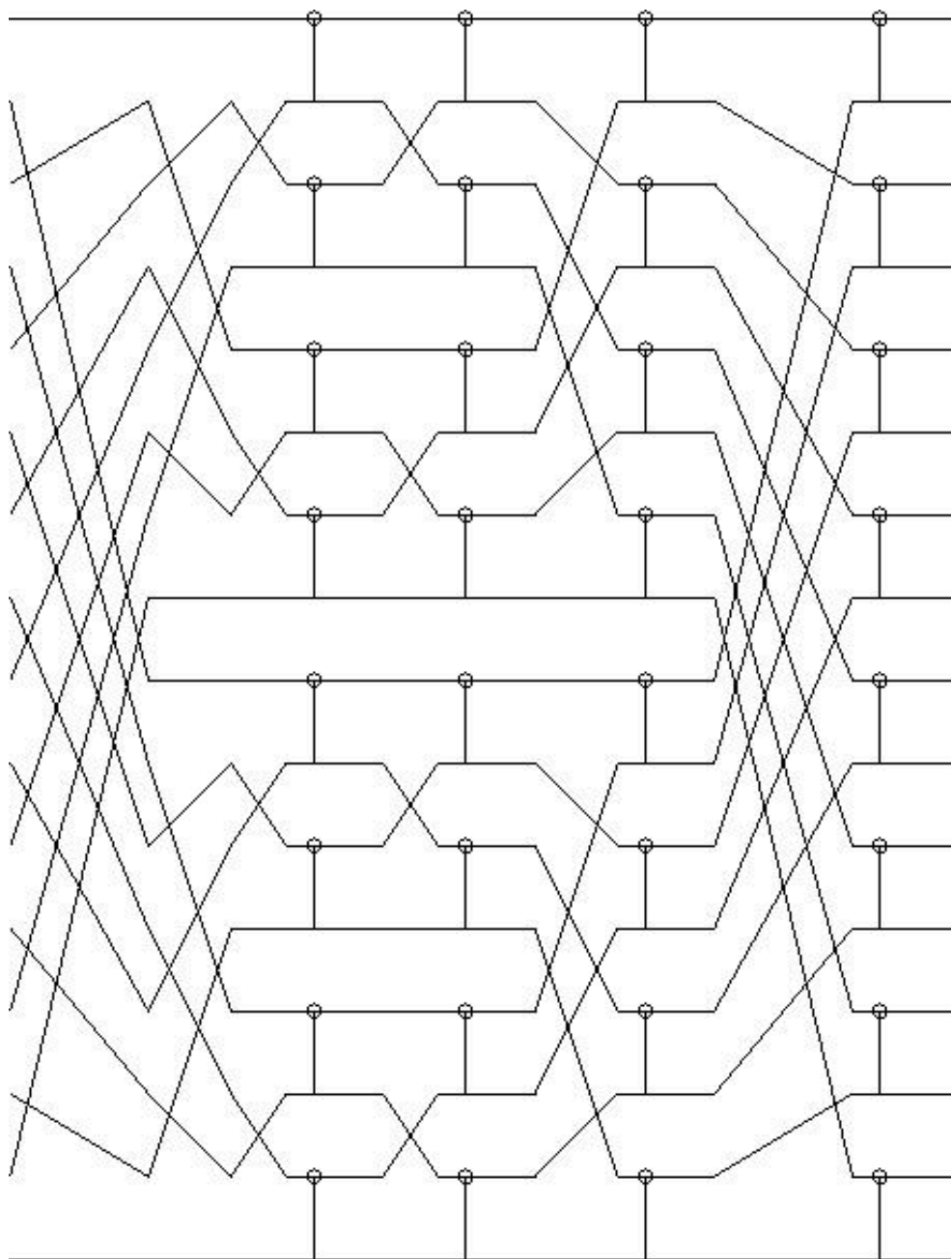
two

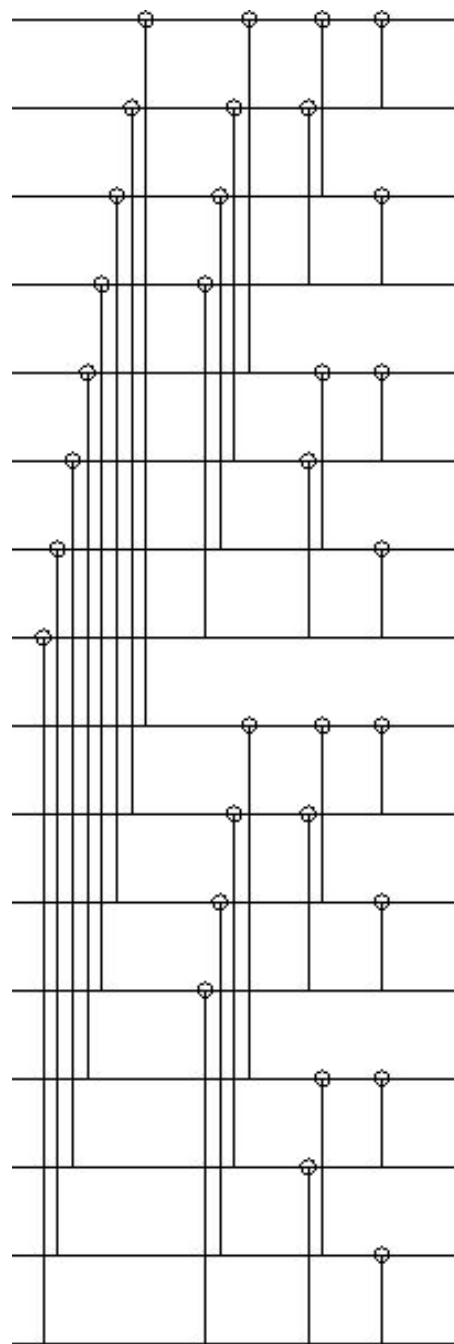
two (ilv f)

ilv (two f)

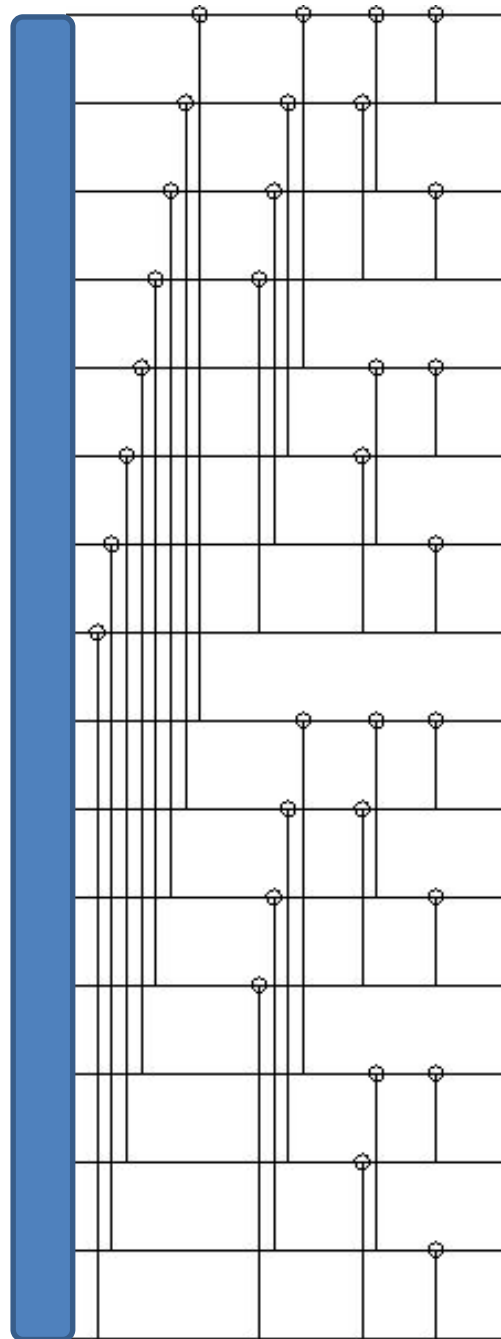
$\text{bfly } n \ f = \text{llv } (\text{bfly } (n-1) \ f) \rightarrow \text{evens } f$

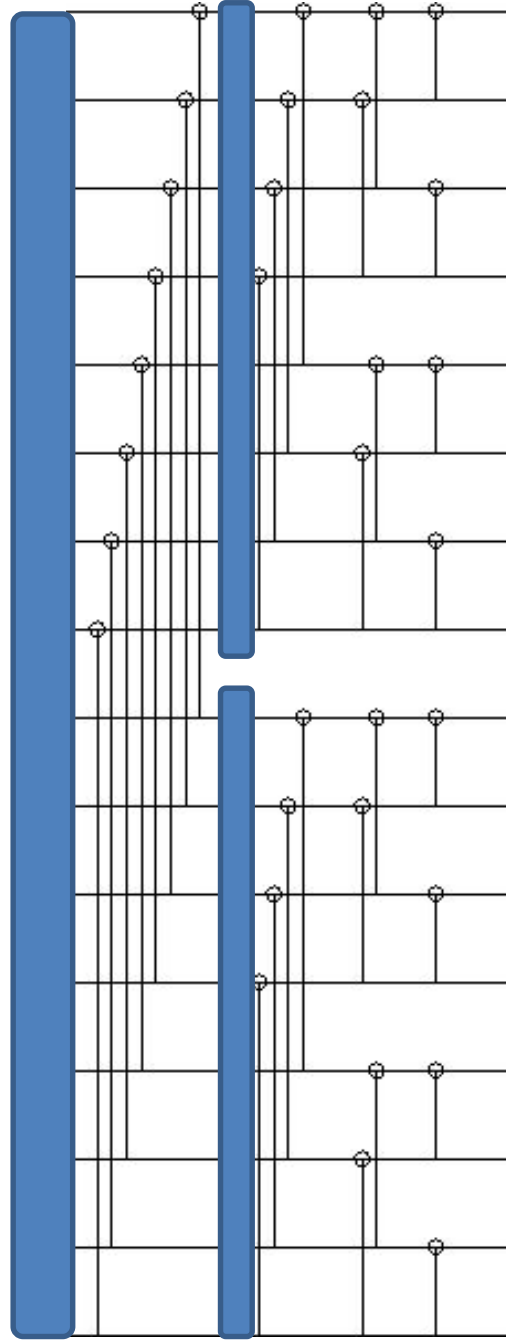






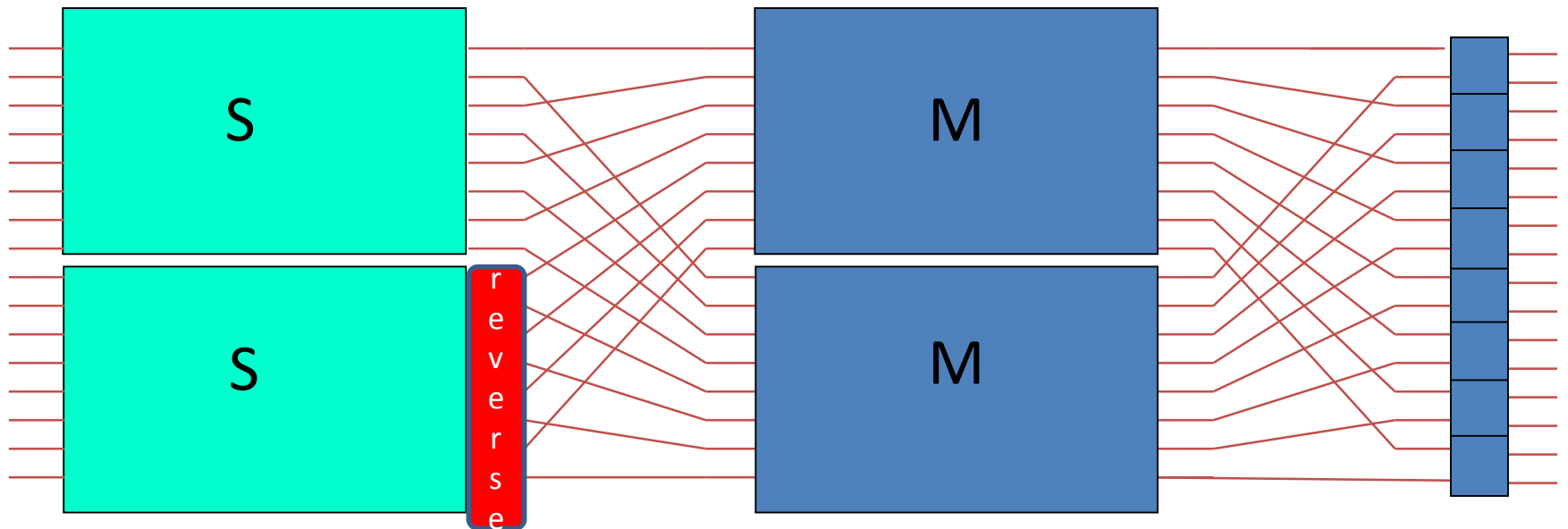
bitonic

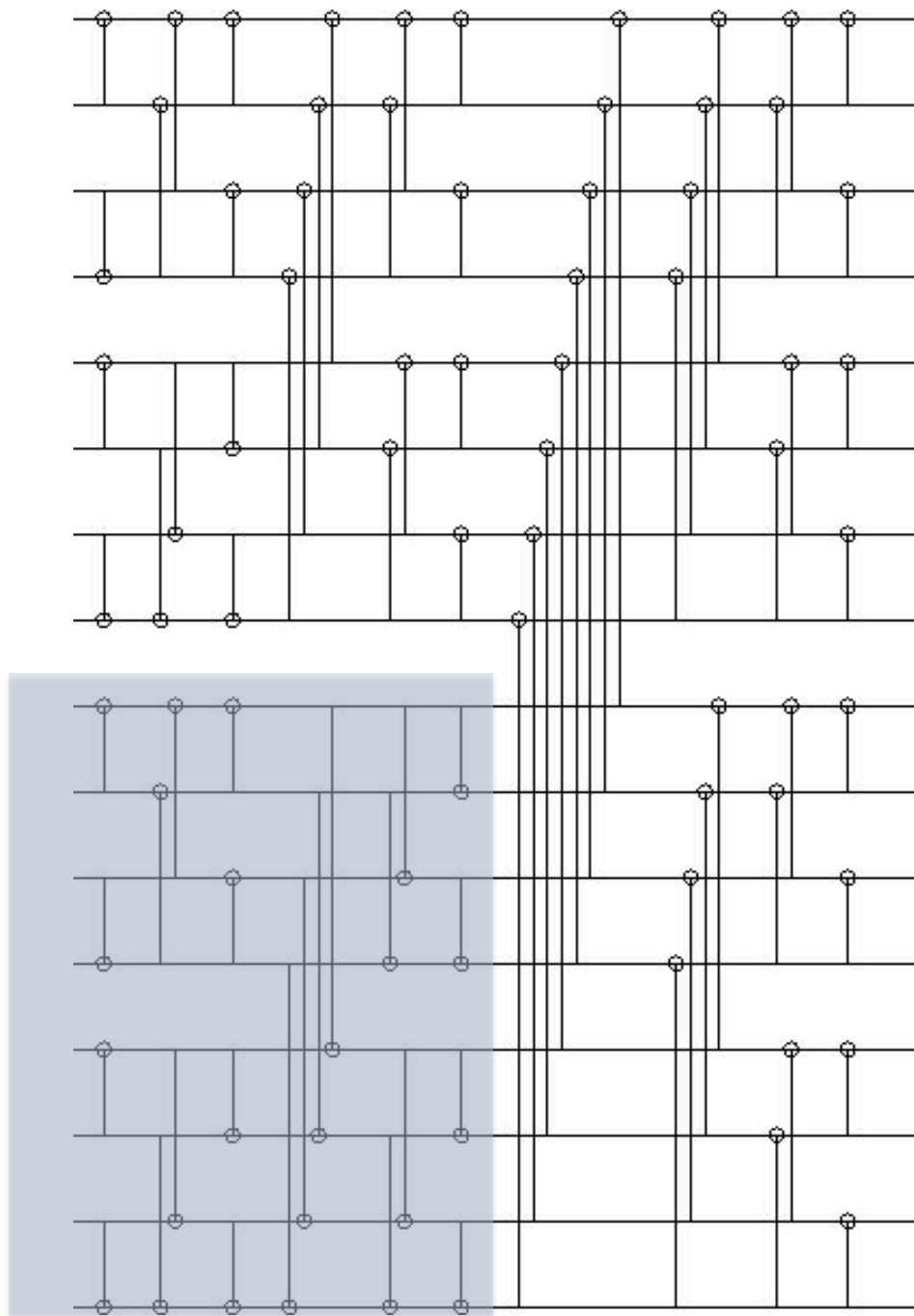




$>=$

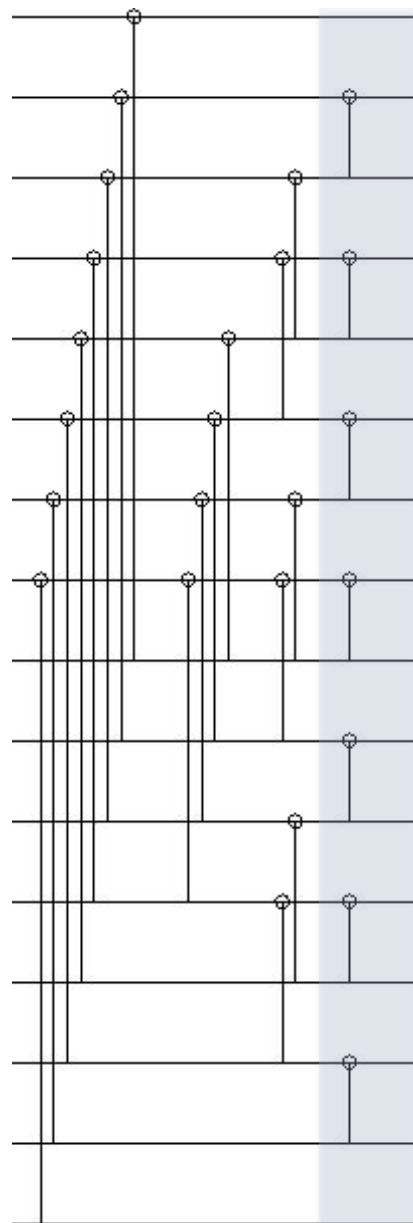
Batcher's sorter (bitonic)

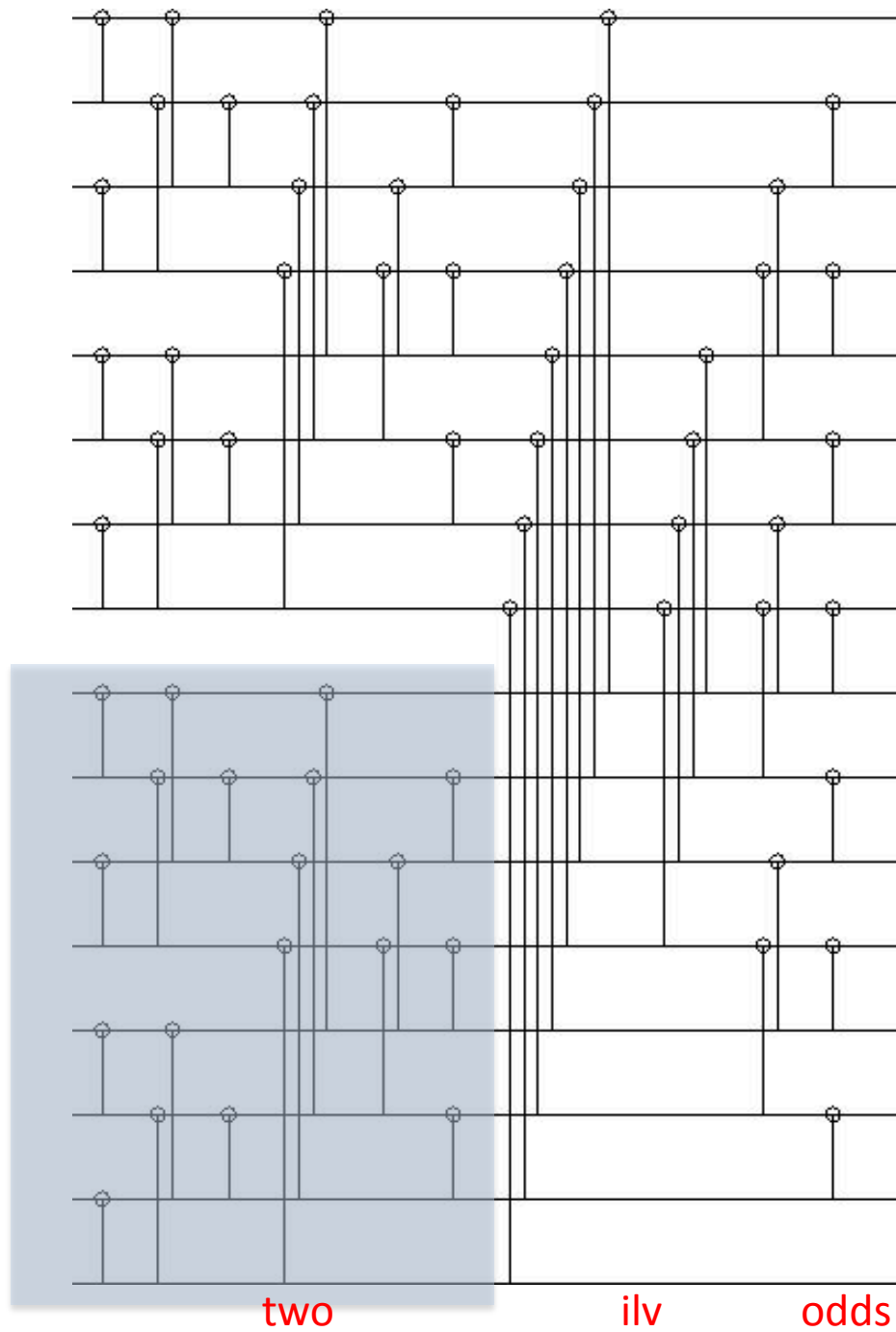




80

```
oemerge :: Int -> ([a] -> [a]) -> [a] -> [a]
oemerge 1 s2 = s2
oemerge n s2 = ilv (oemerge (n-1) s2) ->- odds s2
```

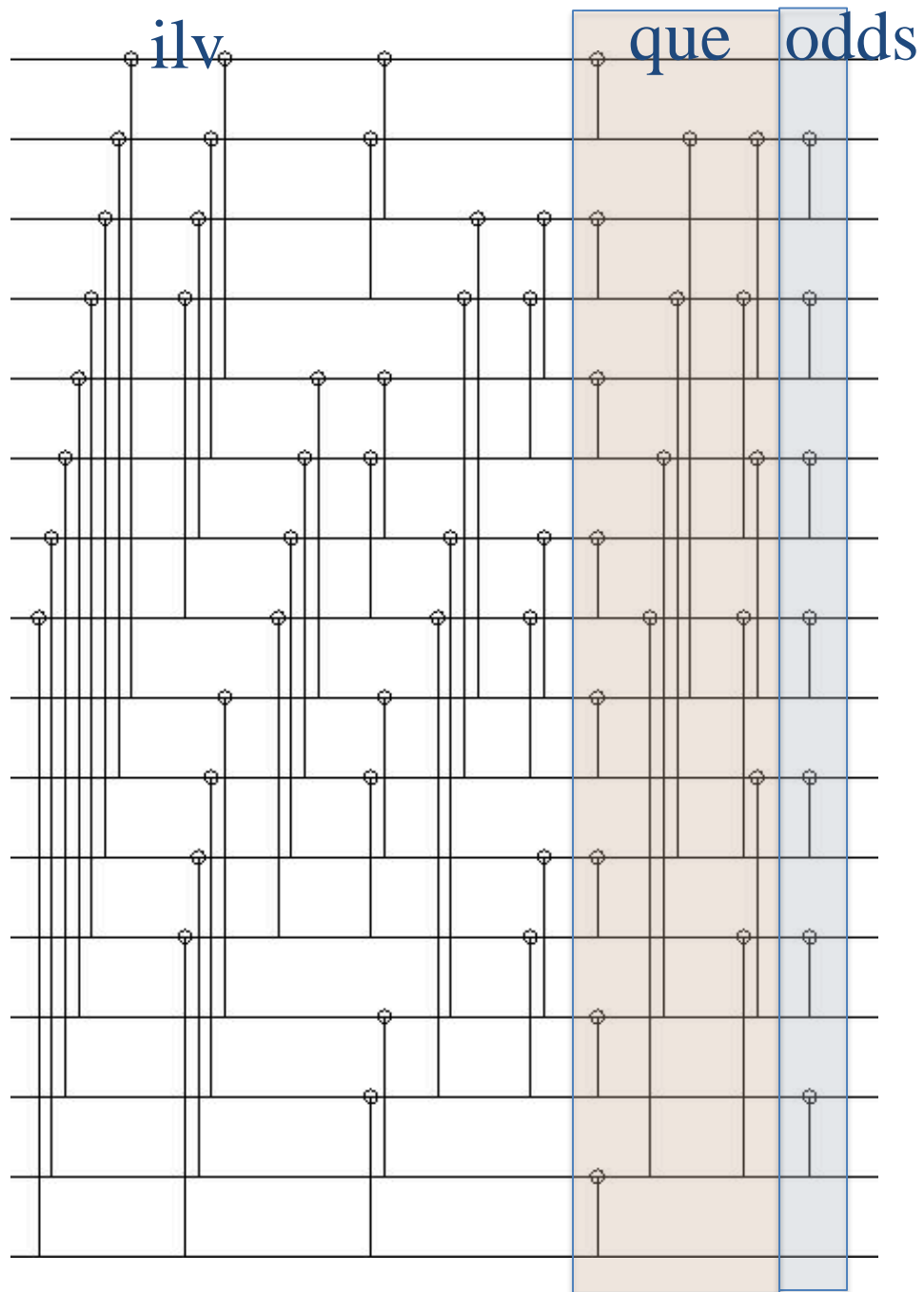


63

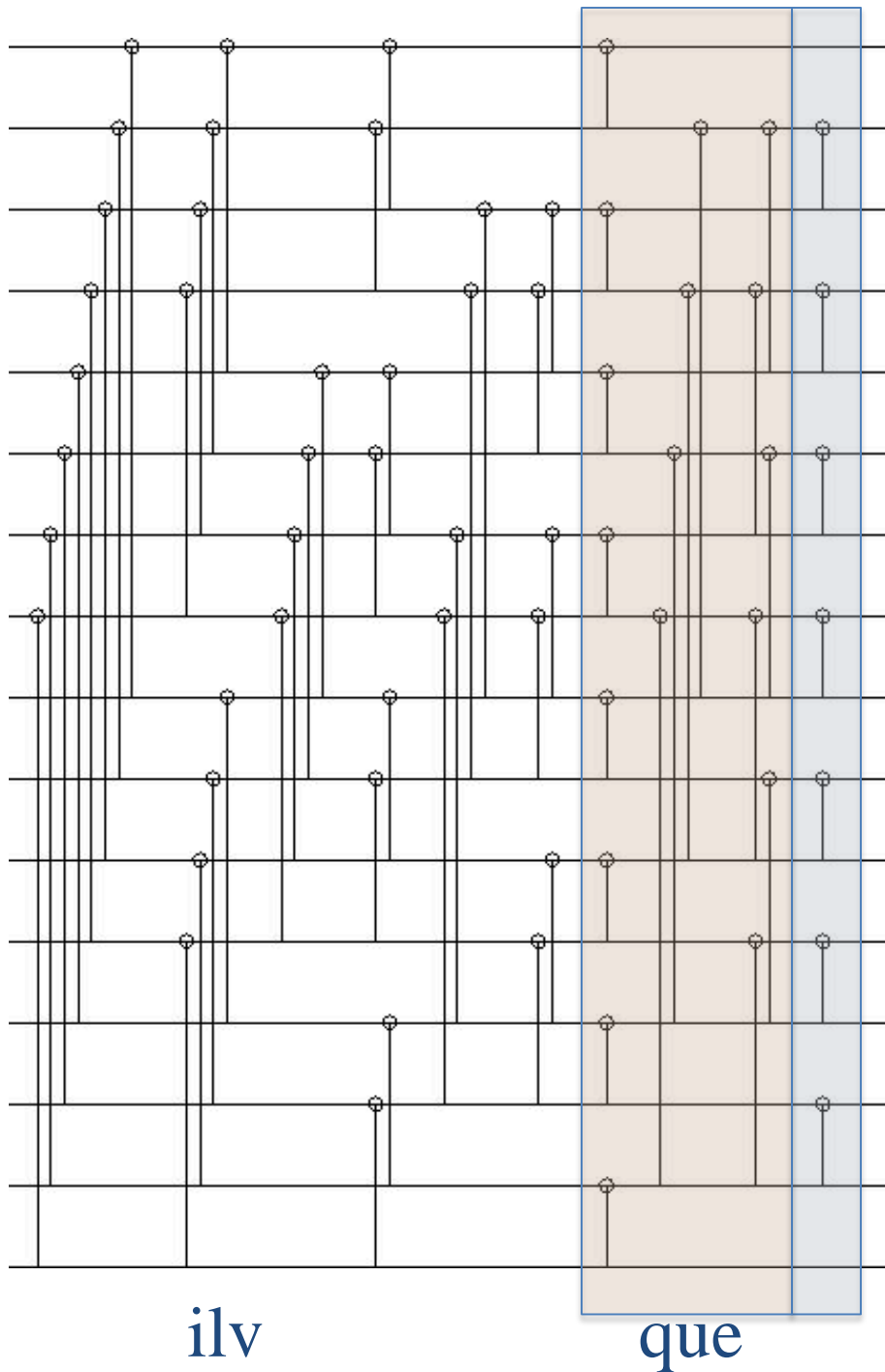
More combinators

que

vee

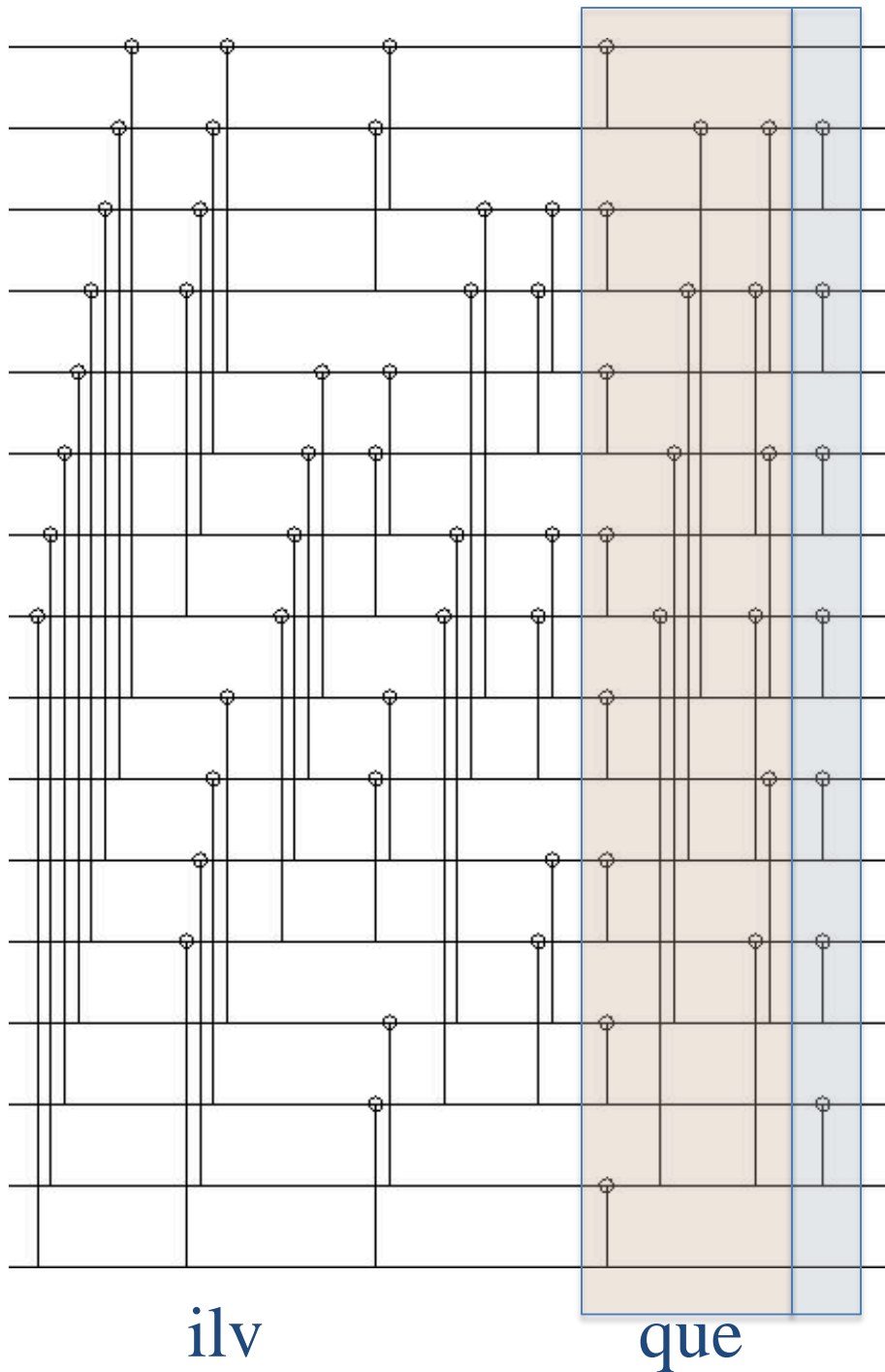


Canfield & Williamson



63

[Canfield & Williamson](#)

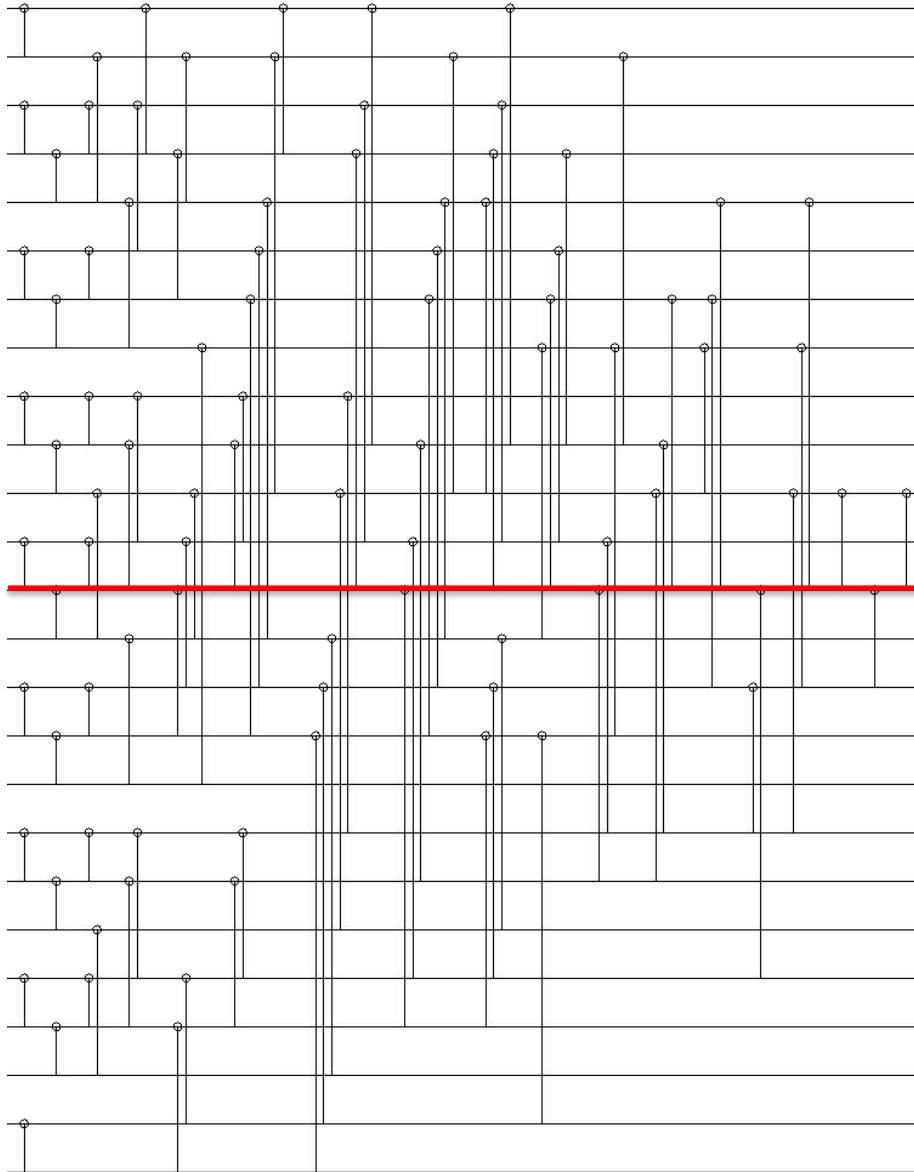


63

60 best
(see Knuth)

[Canfield & Williamson](#)

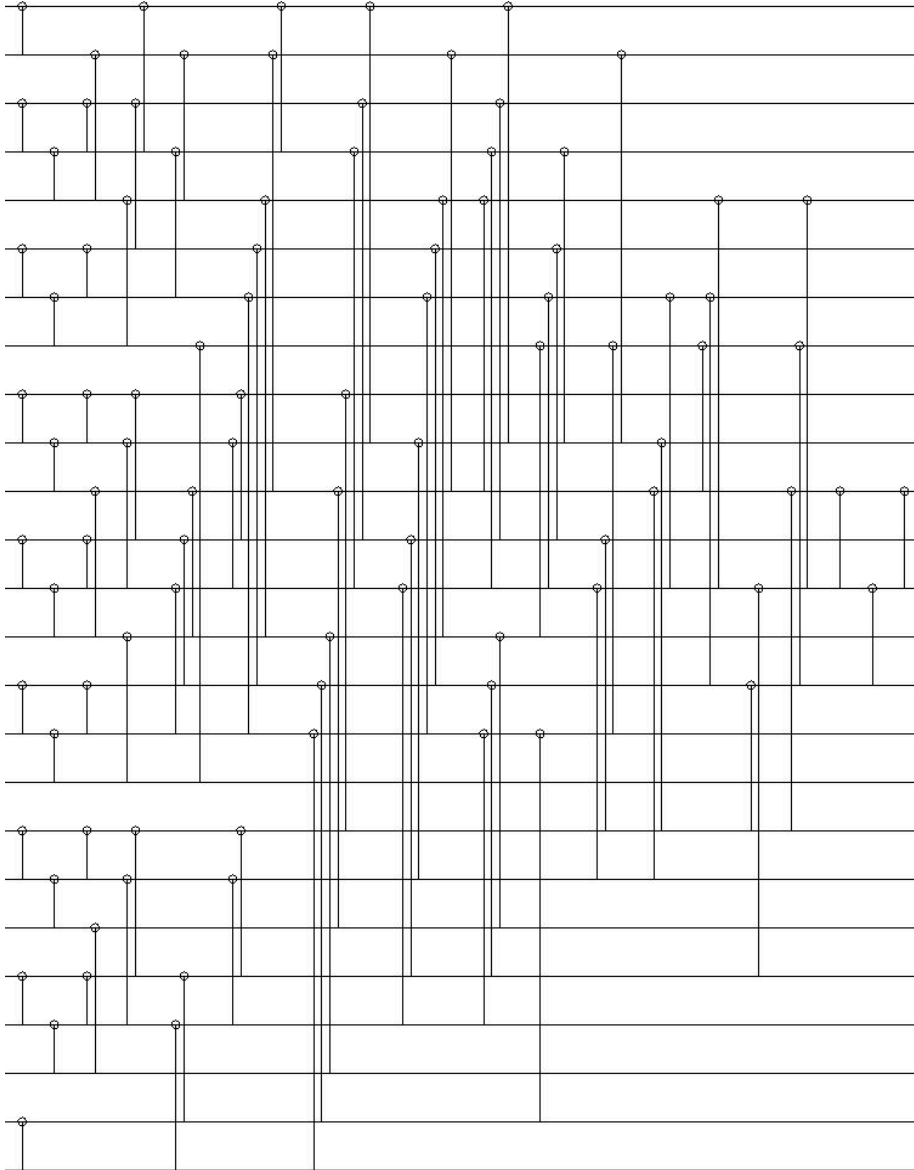
median



\leq 96

\geq

median



96

98 Charme

99 Paeth
Graphics Gems I

SEARCH

“Recently, a sequence of 2^n -input prefix circuits of depth n and complexity $L(2^n)$ (at least for $n \leq 25$) was discovered by Sheeran [12, 13] via computer programming.”

[JFP Vol 21 Issue 01 2011](#)

SEARCH

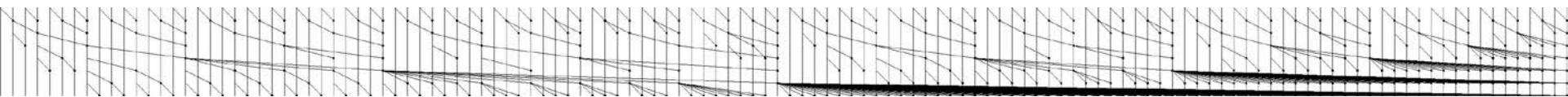
On the complexity of parallel prefix circuits *

Igor S. Sergeev[†]

Abstract

It is shown that complexity of implementation of prefix sums of m variables (i.e. functions $x_1 \circ \dots \circ x_i$, $1 \leq i \leq m$) by circuits of depth $\lceil \log_2 m \rceil$ in the case $m = 2^n$ is exactly

$$3.5 \cdot 2^n - (8.5 + 3.5(n \bmod 2))2^{\lfloor n/2 \rfloor} + n + 5.$$



364

notation => play => new algorithms

SEARCH (examples)



SEARCH (examples)

VALSALAM AND MIIKKULAINEN

Journal of Machine Learning
Research 14 (2013)

n	12	13	14	15	16	17	18	19	20	21	22	23
Previous best	Hand-design and END					Batcher's and Van Voorhis' merge						
	39	45	51	56	60	73	79	88	93	103	110	118
SENSO	39	45	51	56	60	71	78	86	92	102	108	118

SEARCH (examples)

<i>n</i>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
upper bound	0	1	3	5	9	12	16	19	25	29	35	39	45	51	56	60
old lower bound	0	1	3	5	9	12	16	19	23	27	31	35	39	43	47	51
new lower bound									25	29	33	37	41	45	49	53

Codish et al
 25 comparators is optimal
 when sorting 9 inputs

Design FOR verification

Puts circuits to use in a new way

Example: [MiniSat+](#)

Translating Pseudo-Boolean Constraints into SAT (Een and Sörensson)

Journal on Satisfiability, Boolean Modeling and Computation 2 (2006)

HW + FP in the real world?

HW + FP in the real world?

$4195835.0 - 3145727.0 * (4195835.0 / 3145727.0) = 0$ (Correct value)

$4195835.0 - 3145727.0 * (4195835.0 / 3145727.0) = 256$ (Flawed Pentium)

HW + FP in the real world?

Intel

Forte System

1000s users

HW + FP in the real world?

Intel

Forte System

1000s users

fl

lazy functional language with built-in BDDs, decision procedures
and a HW symbolic simulator (Symbolic Trajectory Evaluation engine)

Thanks to Carl Seger (Intel)

HW + FP in the real world?

Intel

Forte System

1000s users

fl

lazy functional language with built-in BDDs, decision procedures
and a HW symbolic simulator (Symbolic Trajectory Evaluation engine)

Design language

High-level specification language

Object language for theorem proving

Scripting language

Implementation language for formal verification tools and theorem provers

Thanks to Carl Seger (Intel)

Examples of fl as Design Language

High level

```
// Actual SHA1 computation
let process_chunk_fun H11 b =
  let H11l = split_shash H11 in
  let W = [extend_block_t b t | t in 79 -- 0] in
  let do_iter t [e,d,c,b,a] =
    let T = (a ROTL 5) '+' (fn t b c d) => e '+' K160^t '+' W^t in
    let e = d in
    let d = c in
    let c = b ROTL 30 in
    let b = a in
    let a = T in
    [e,d,c,b,a]
  in
  let edcba = itlist do_iter (79--0) H11l in
  let Hnewl = map2 (defix '+') edcba H11l in
  unsplit_shash Hnewl
;
```

RTL level

```
let tst =
  INTERFACE
    bit_input  clk.
    op_input   op.
    byte_input a b.
    byte_output res.
    byte_output lres.
  CELL "tst" (
    ff clk a !ad #
    ff clk b !bd #
    add2 !ad !bd !tmp #
    and2 !ad !bd !tmp2 #
    or2 !ad !bd !tmp3 #
    xor2 !ad !bd !tmp4 #
    not !ad !tmp5 #
    !tmp6 <== CASE op [
      (ANDop, !tmp2),
      (ORop, !tmp3),
      (XORop, !tmp4),
      (NOTop, !tmp5)
    ] '0 #
    ff clk !tmp6 lres #
    ff clk !tmp6 res
  );
```

With physical placement information

```
// Make a buffer tree for input inp feeding n vertically aligned
// 'component_height' high components.
nlet mk_buffer_tree {inp:: *w} n component_height->1 driver_at_top->F =
  n = 0 => ([], rpspacer 0 0) |
  let name = hd (bdepends (destr inp)) in
  let grp_sz = max MIN_BUFFERING (ceiling (sqrt (int2float n))) in
  let needed = (n+(grp_sz-1))/grp_sz in
  let bufs = {mk_var ("tmp_"^name) needed :: *w list} then
  let ninp = {mk_var ("tmp_neg_"^name) :: *w} then
  let rht = row_height*component_height in
  let minwid = (needed+1)*m3_pitch in // Ensure sufficient wiring tracks
  let ckt =
    (driver_at_top => (
      wire_o inp ninp
      ##
      (rpspacer minwid ((grp_sz/2-1)*rht))
    ) | (
      rpspacer minwid ((grp_sz/2)*rht)
    )
    ##
    (forall_vabut [ (i,ot) | zip_in_id (butlast bufs) ].
      wire_o ninp ot ##
      (rpspacer minwid (min ((grp_sz-1)*rht
        (((driver_at_top=>n | (n-1))-
          (grp_sz/2+1+(i-1)*grp_sz-1)*rht))))
      )
    ##
    (wire_o ninp (last bufs))
    ##
    (driver_at_top => (
      (rpspacer minwid
        (max 0 ((n-(grp_sz/2+1+(needed-1)*grp_sz))*rht)))
      ) | (
        (rpspacer minwid
          (max 0 ((n-1-(grp_sz/2+1+(needed-1)*grp_sz))*rht)))
        ##
        (wire_o inp ninp)
      )
    )
  in
  let buf_drivers = map (\i. el ((i/grp_sz)+1) bufs) (0 upto (n-1)) then
  (buf_drivers, ckt)
;
```

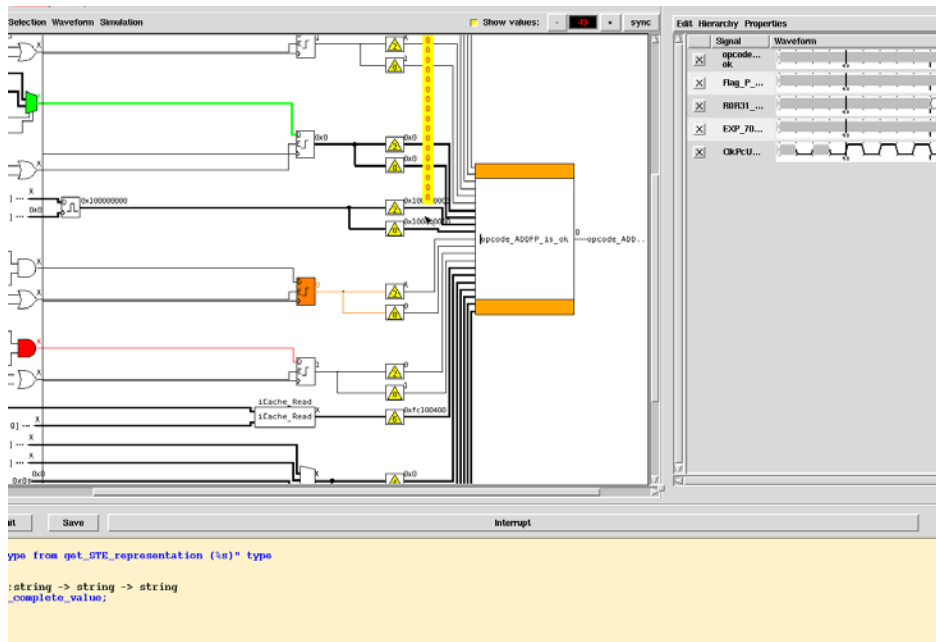
Example of fl as Specification Language

- Use the builtin BDDs and the ability to write if-then-else conditions over expressions to create concise and clean specifications for even very complex operations.
- Example: Floating point addition

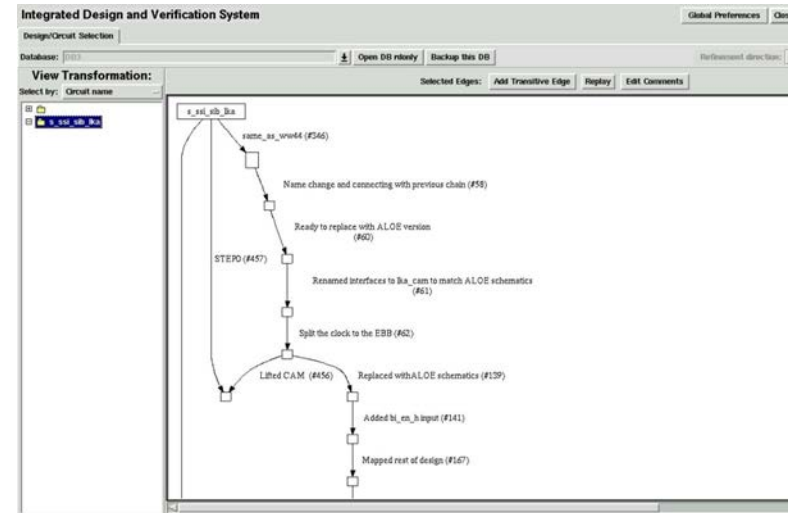
```
// Floating point add specification
let ADD pc rc in1 in2 =
  // Are we going to swap - i.e. is |in1| > |in2| ?
  let swap = (no_signs in1) >> (no_signs in2) in
  // Get the smaller magnitude into fp1, larger into fp2
  let fp1 = IF swap THEN in2 ELSE in1 in
  let fp2 = IF swap THEN in1 ELSE in2 in
  // Now, take apart into exponents and significands
  val (exp1,sgf1) = split_fp fp1 in
  val (exp2,sgf2) = split_fp fp2 in
  // Restore exponents for denorms and zeros
  let minexp = bias (0-((2**16)-2)) in
  let ex1 = IF (exp1 '= 0) THEN minexp ELSE exp1 in
  let ex2 = IF (exp2 '= 0) THEN minexp ELSE exp2 in
  // Now, shift fp1 to align with fp2.
  let sgf1' = srshift 68 rsh (sgf1 @ [F,F]) in
  let sgf2' = sgf2 @ [F,F] in
  // Perform the sum (or subtract)
  let true_add = (sign fp1 = sign fp2) in
  let sum = IF true_add THEN (sgf2' '+' sgf1') ELSE (sgf2' '-' sgf1') in
  let ex = ex2 in
  let sgn = sign fp2 in
  // Renormalise, if necessary (first renormalisation)
  val (0:J:rem) = sum in
  let zs = zeros rem in
  let lsh = zs '+' 1 in
  // {At this point 0 <= lsh <= 68}
  val (nsum1,nex1) =
    IF 0 THEN (rshift 1 sum, ex '+' 1) ELSE
    IF (NOT 0 AND NOT J) THEN (slshift 68 lsh sum, (ex '-' lsh) ELSE
    (sum,ex)
  in
  // Now, round the result according to the current precision
  let rsum = RND pc rc sgn nsum1 in
  // Right-shift renormalisation, if necessary
  let 0 = hd rsum in
  let nsum2 = IF 0 THEN ([F] @ butlast rsum) ELSE rsum in
  let nex2 = IF 0 THEN (nex1 '+' 1) ELSE nex1 in
  IF (NOT(hd(tl nsum2))) THEN
    IF (nsum2 = (nat_to_bv 68 0)) THEN
      ((IF ((rc = TO_NEG_INF) AND NOT add) THEN [T] ELSE [F]) @
      expzeros @ nsum2)
    ELSE ([sgn] @ expzeros @ nsum2)
  ELSE
    // Otherwise, return the answer.
    ([sgn] @ nex2 @ nsum2)
  ;
```

Example of Systems Built in fl

STEP: Formal Verification tool:
120k lines of fl + 25k lines of Tcl/Tk



IDV: Integrated Design and Verification:
280k lines of fl + 40k lines of Tcl/Tk

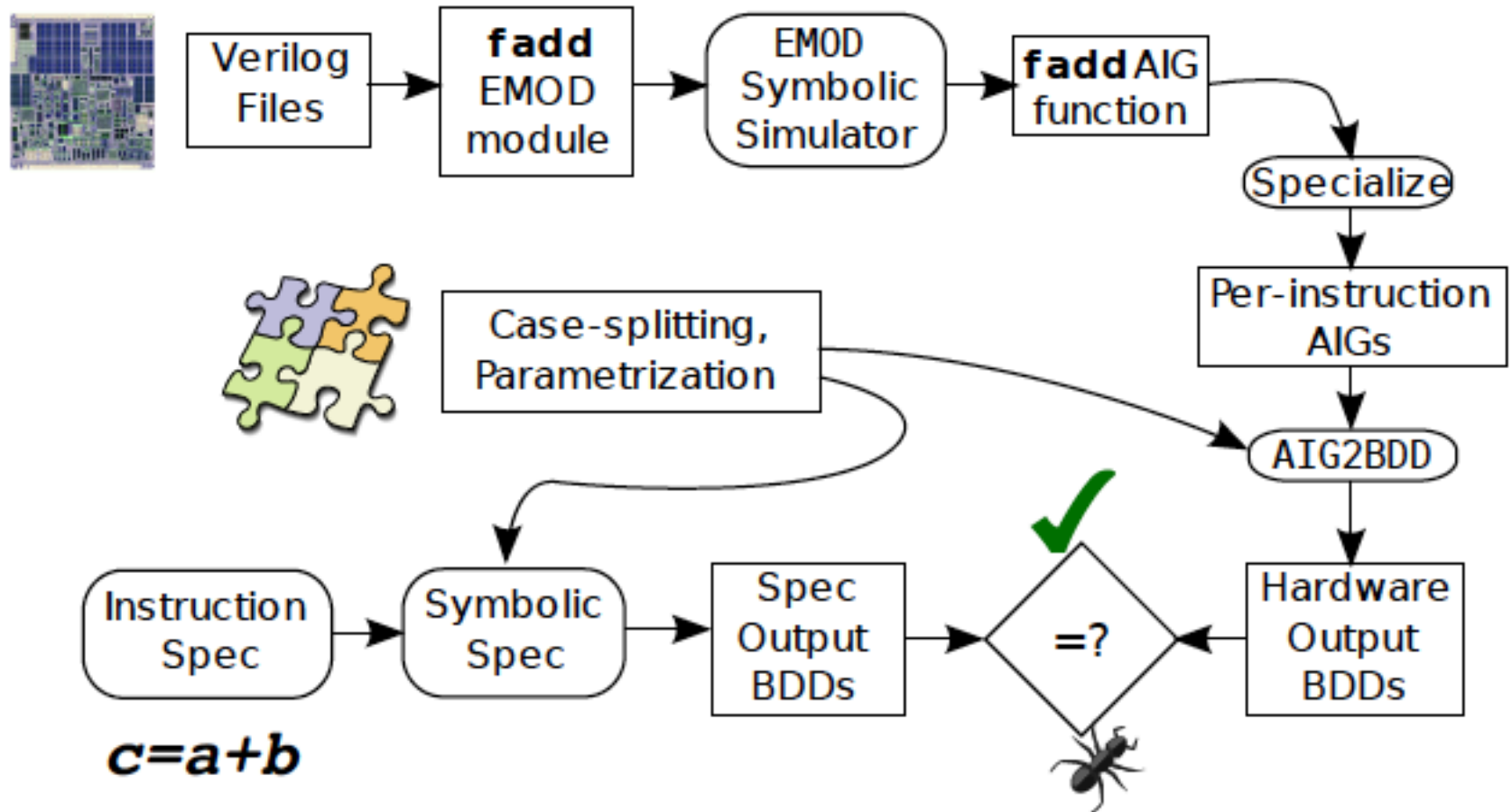


[Forte](#)

[How verification is done in practice](#)

Slide provided by Carl Seger (Intel)

The Centaur Media-Unit, Verification Tool Flow



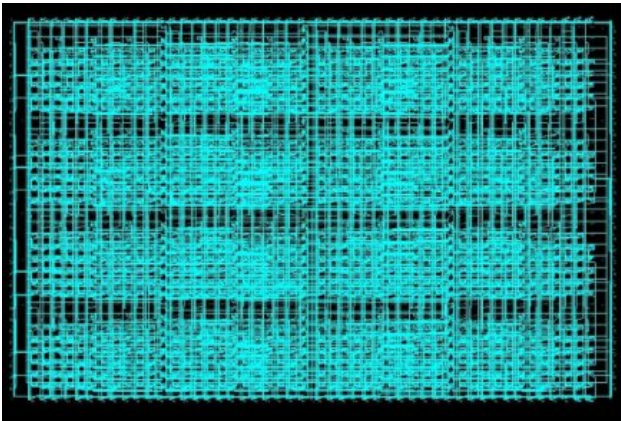
Bluespec

FP in HW design



Bluespec

FP in HW design



(FPGA layout by Satnam Singh)

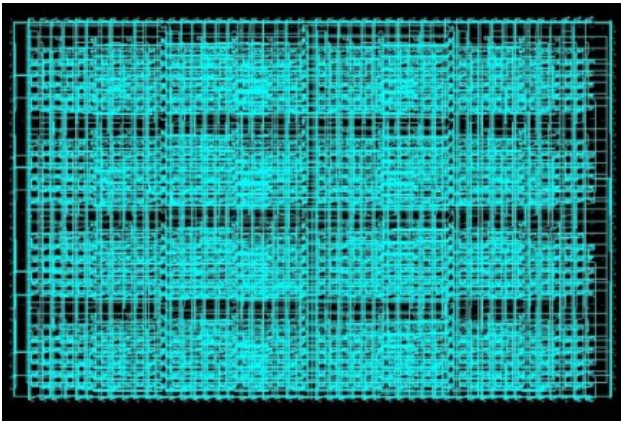
Thanks to R.S. Nikhil (Bluespec)

Bluespec

FP in HW design



malware / hacking



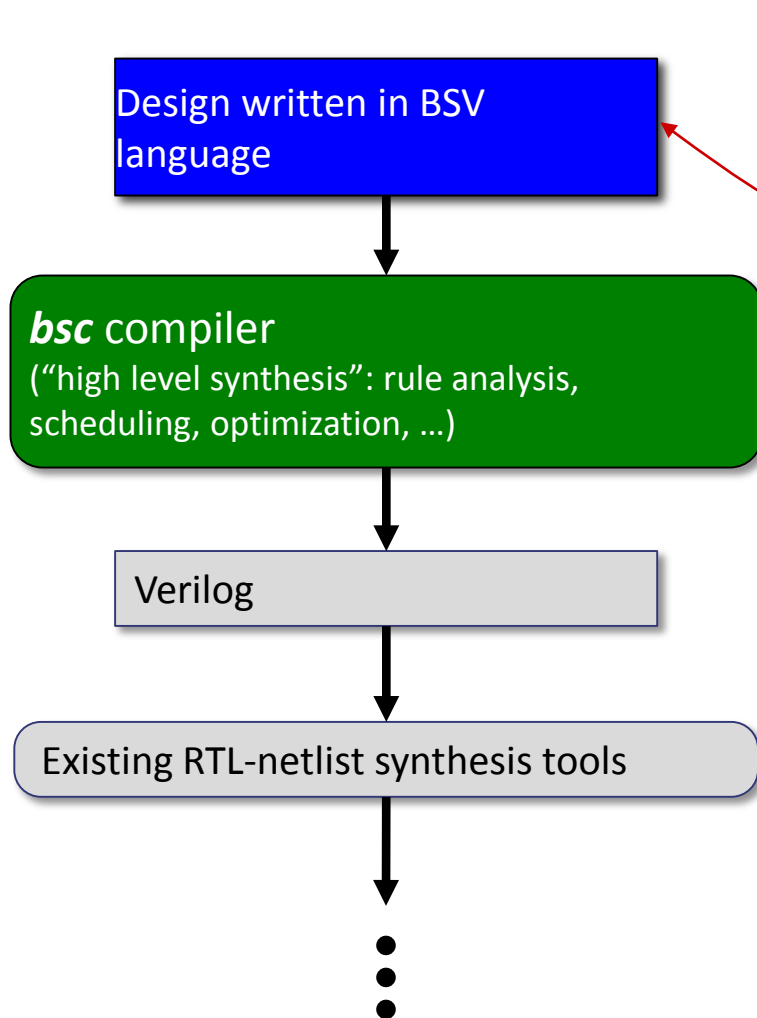
(FPGA layout by Satnam Singh)



Thanks to R.S. Nikhil (Bluespec)

BSV is based on declarative languages

(Verilog and VHDL are the main languages for HW design; > 25 years old)



Borrow best ideas from modern programming languages, formal verification systems, and concurrency.
Abandon sequential von Neumann legacy.

Behavior spec:

Guarded Atomic Transaction Rules

- cf. Guarded Commands (Dijkstra), TLA+ (Lamport), UNITY (Chandy/Mishra), EventB (Abrial), ...
- Fundamentally parallel/concurrent

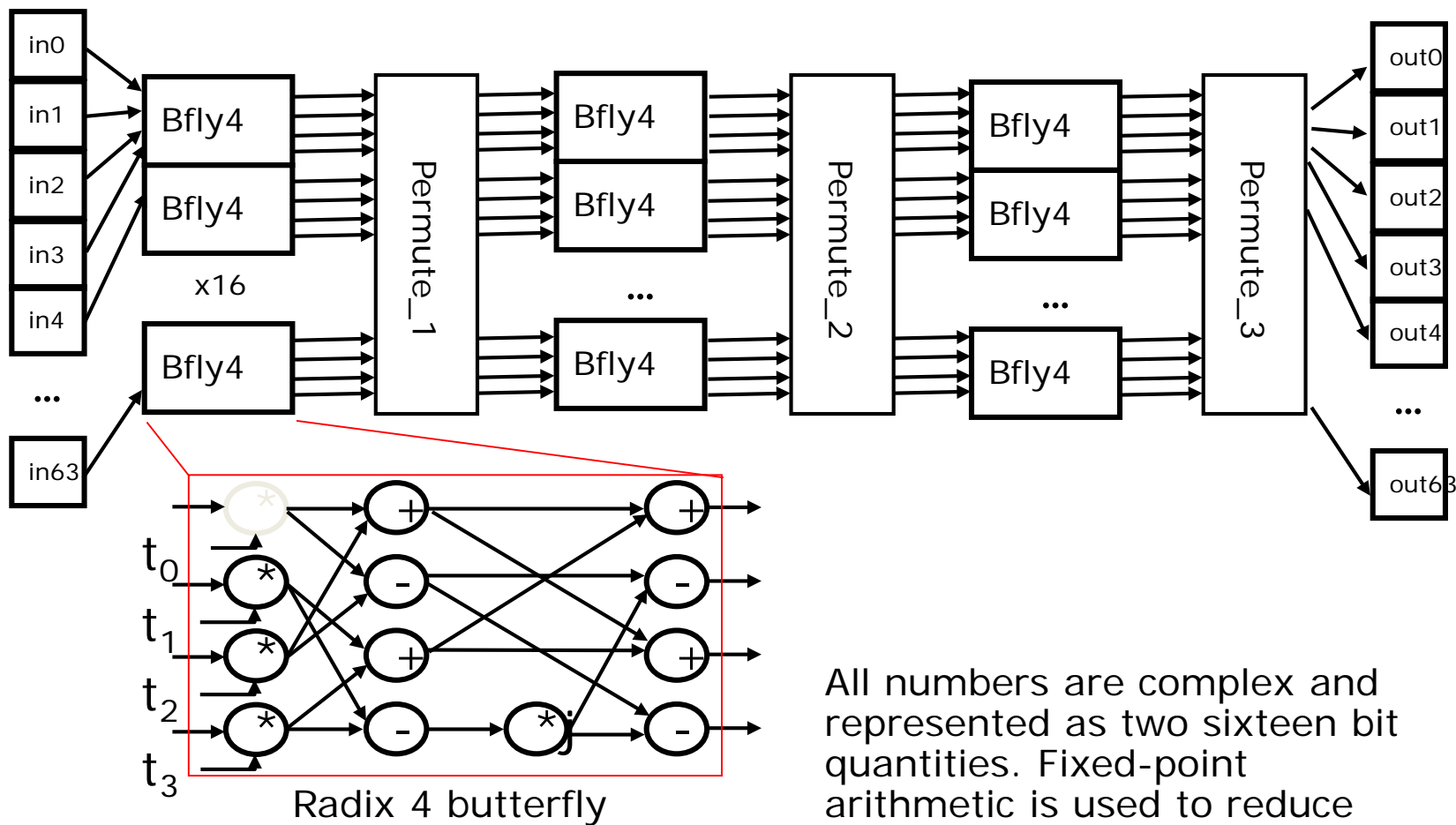
Architecture spec:

Pure functional programming language

- cf. Haskell
- Strong type-checking, polymorphic types, typeclasses, higher-order functions, modularity, parameterization

The IFFT computation (specification)

(as used in 802.11a Transmitter, for example)

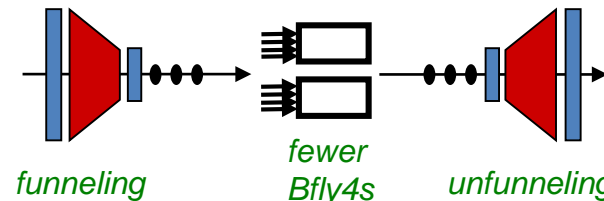
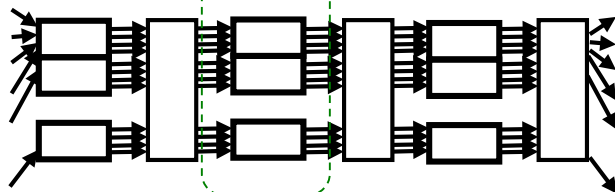


All numbers are complex and represented as two sixteen bit quantities. Fixed-point arithmetic is used to reduce area, power, ...

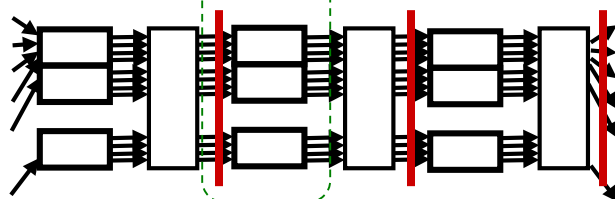
IFFT: the HW architecture space

(varying in area, power, clock speed, latency, throughput)

Direct combinational circuit

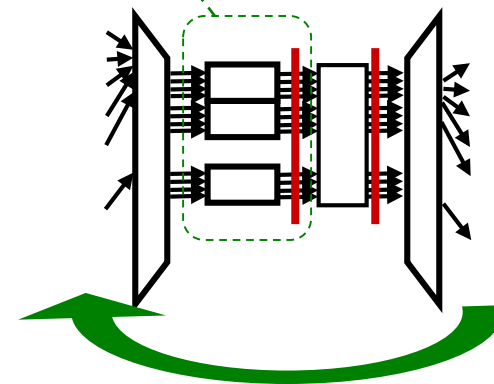


In any stage, use fewer than 16 Bfly4s

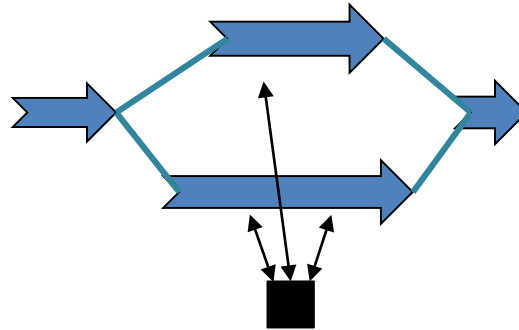


Varying degrees of pipelining

Iterate 1 stage thrice



Rule semantics enables compositionality of pipelines



Different points on a path, or on different paths, may access some shared resource (such as a RAM or a counter), at unpredictable (perhaps data dependent) times.

Previous systems have also used higher-order functions to express structural composition of circuits. E.g., Lava [Bjesse, Claessen, Sheeran, Singh 1998].

But they were based on traditional synchronous clocked digital circuit semantics, so user has to manually manage *pipeline balancing*¹, *flow control*, and *access to shared resources*.

Rule semantics are naturally “asynchronous”, enabling separation of pipeline structure from those concerns.

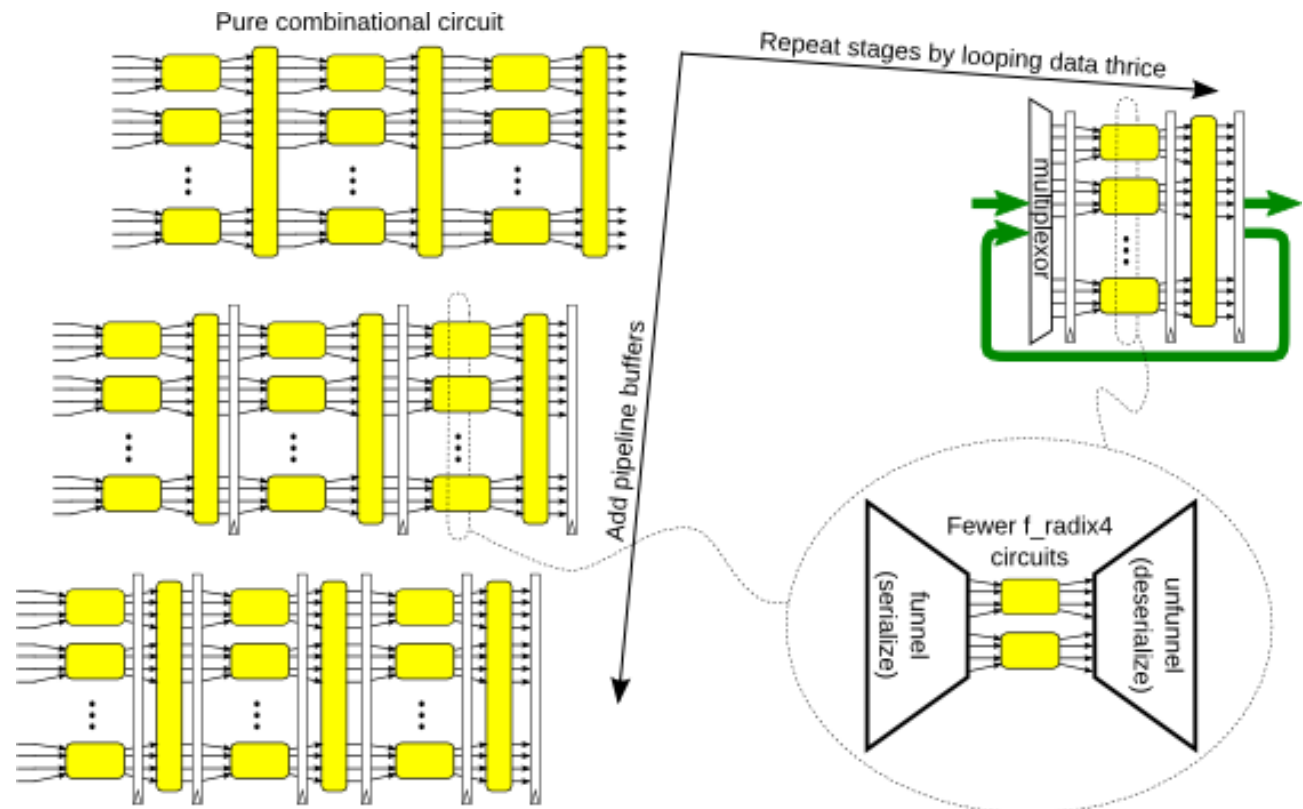
¹Balancing: latencies may be data-dependent, and different on different paths.

Results of using PAClib on IFFT

100 lines of BSV source code based on 4 parameters,
express all 24 architectures in the figure, with a 10x variation in area/power

(which is “best” depends on target requirements, e.g., server vs. mobile)











- *fully pipelined, flow-controlled*
- *all control logic correct by construction*



Synthesis from BSV is competitive with hand-coded RTL

Example: Deblocking filter for H.264 and VP8 video decoders

Relative silicon area (smaller is better)

resolution				
functionality		  		  
Hand-coded VHDL (reference)		1x		2x (estimate)
BSV	0.18x	0.33x	0.47x	0.81x

These results are not just competitive with RTL, but far superior. Can this really be true?
Yes, sometimes.

BSV

Often BEATS hand-coded RTL code

BSV

Often BEATS hand-coded RTL code

Algorithmically superior designs

BSV

Often BEATS hand-coded RTL code

Algorithmically superior designs

Refinement, evolution, major architectural change EASY

Types, Functional Programming and Atomic Transactions
in Hardware Design Nikhil LNCS 8000

Bluecheck

[A Generic Synthesisable Test Bench \(Naylor and Moore, Memocode 2015\)](#)

QuickCheck in HW design!

Idea of a generic testbench is unheard of in mainstream HDLs

stack interface

// A stack of 2^n elements of type t

```
interface Stack#(type n, type t);  
method Action push(t x);  
method Action pop;  
method Bool isEmpty;  
method t top;  
method Action clear;  
endinterface
```

```
module [Specification] stackSpecAlg ();

// Create two instances of implementation
Stack#(8, Bit#(4)) s1 <- mkBRAMStack();
Stack#(8, Bit#(4)) s2 <- mkBRAMStack();

// On s1, push x, then pop it
function pushPop(x) =
seq s1.push(x); s1.pop; endseq;

// On s2, do nothing
function nop(x) = seq endseq;

equiv("pushPop", pushPop, nop);
equiv("push" , s1.push, s2.push);
equiv("pop" , s1.pop , s2.pop);
equiv("top" , s1.top , s2.top);
endmodule
```


=== Depth 20, Test 15/10000 ===

11: push(12)

22: push(2)

23: pushPop(14)

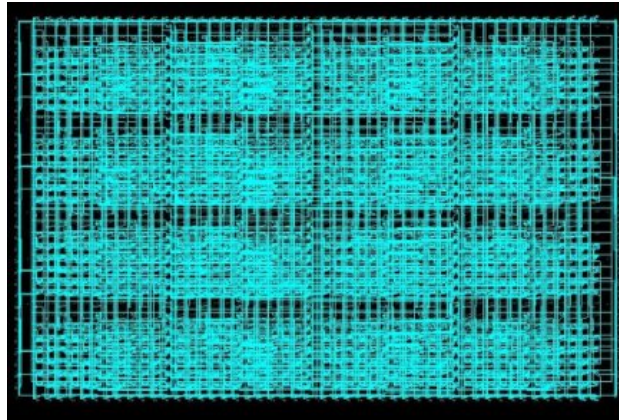
27: pop

28: top failed: 2 v 12

Continue searching?

Synthesisable!

Iterative deepening and shrinking on



=== Depth 10, Test 5/10000 ===
setAddrMap(<15, 11, 8, 5>)
Core 0: MEM[3] == 0
Core 0: MEM[7] := 8
Core 1: MEM[3] := 9
Core 1: MEM[7] == 0
Core 0: MEM[3] == 0
Not sequentially consistent

Pushing verification

Pushing verification

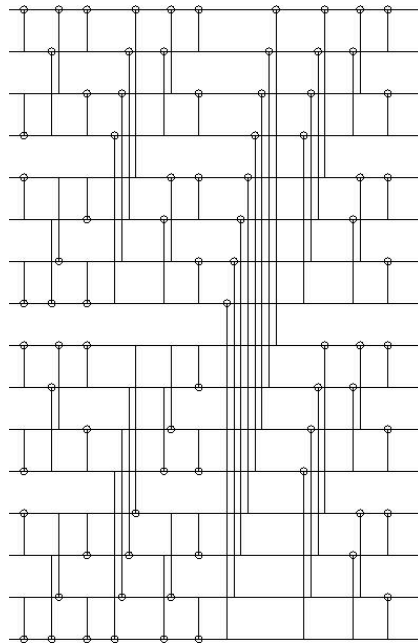
Formal Verification of Hardware Synthesis

CAV'13

Pushing verification

Formal Verification of Hardware Synthesis

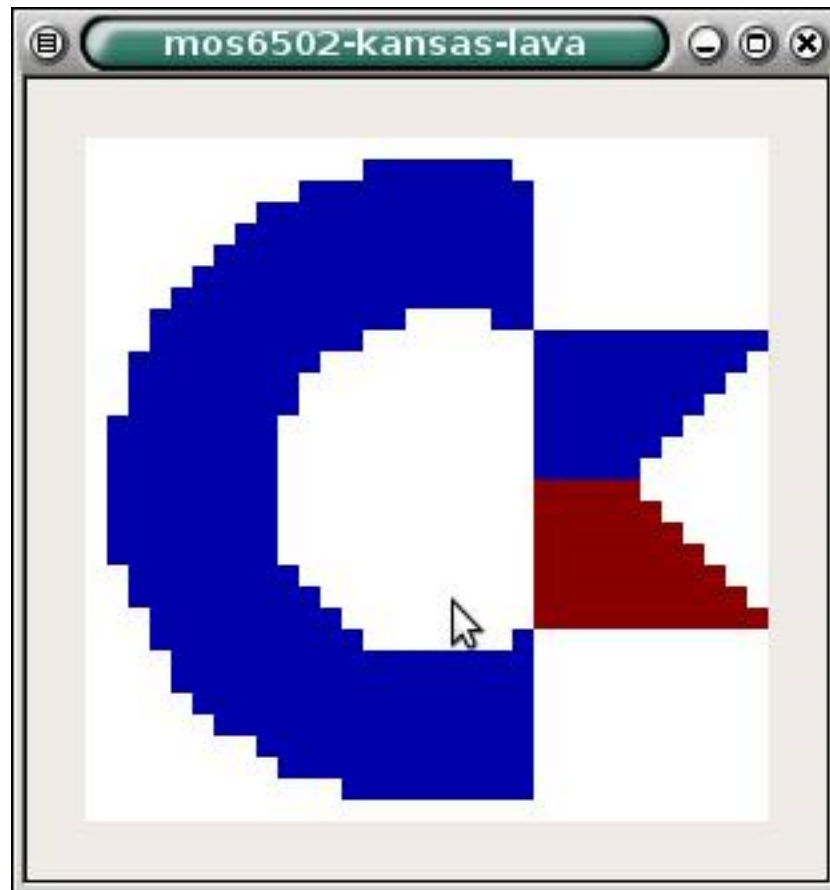
CAV'13



Pushing verification

first machine verification of sequential consistency for a multicore hardware design that includes caches and speculative processors (CAV'15)

Lava(s)



Lava(s)



Feldspar + synchronous programming for hardware at Chalmers

[Kansas Lava](#): “add Bluespec features”

Satnam Singh: I wonder!

CλaSH [HardCAML](#) etc

Chisel

In this paper, we introduce Chisel (Constructing Hardware In a Scala Embedded Language), a new hardware design language we have developed based on the Scala programming language [8]. Chisel is intended to be a simple platform that provides modern programming language features for accurately specifying low-level hardware blocks, but which can be readily extended to capture many useful high-level hardware design patterns.

(DAC'12)

<https://chisel.eecs.berkeley.edu/>

Cryptol

Designing Tunable, Verifiable Cryptographic Hardware Using Cryptol.
In [Design and Verification of Microprocessor Systems for High-Assurance](#), David S. Hardin, Editor. Springer 2010

The declarative quality of Cryptol, which makes Cryptol a good specification language, also plays a key role in the effectiveness of automatic generation of FPGA cores. In contrast, the inherent sequentiality of mainstream programming languages makes them a poor match for the highly parallel nature of FPGAs.

Undelay





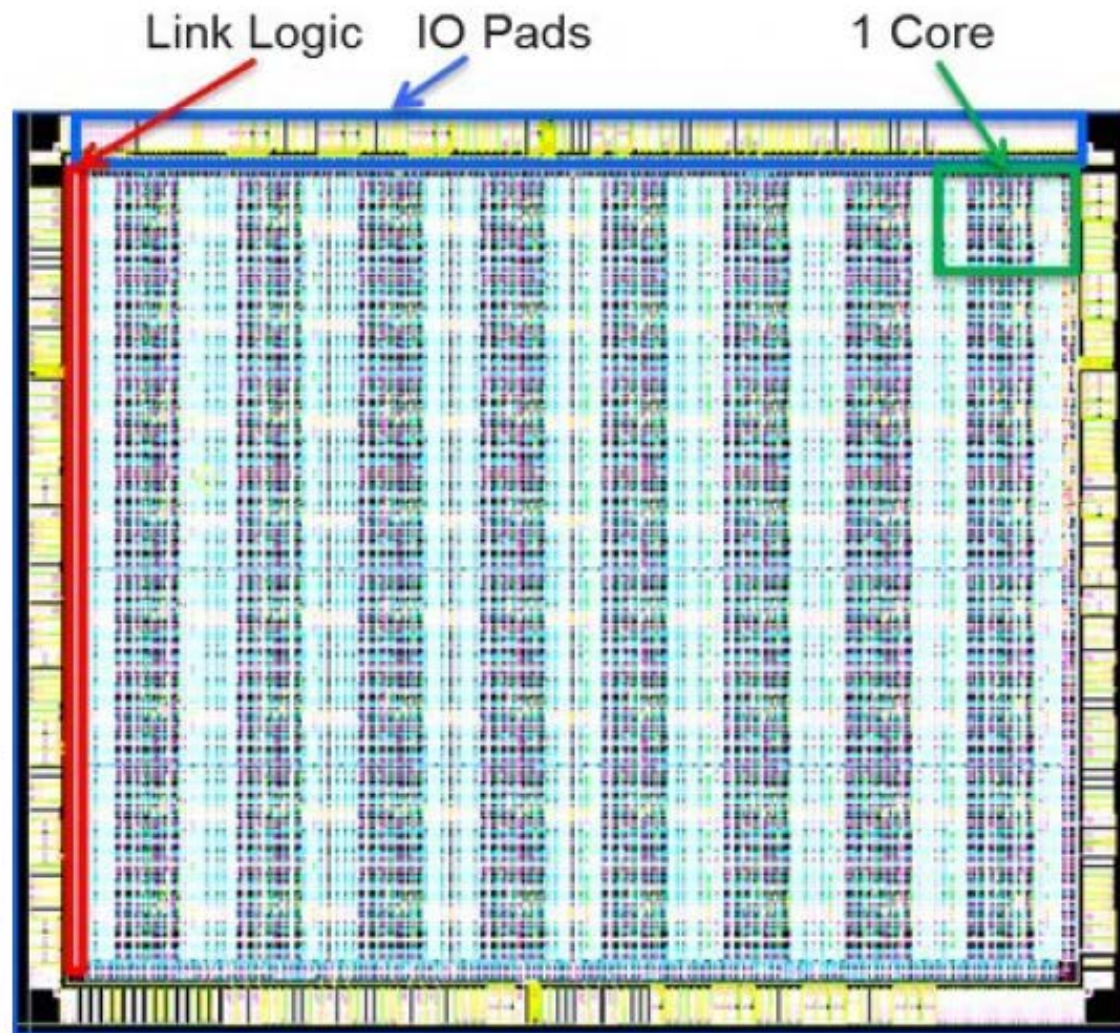
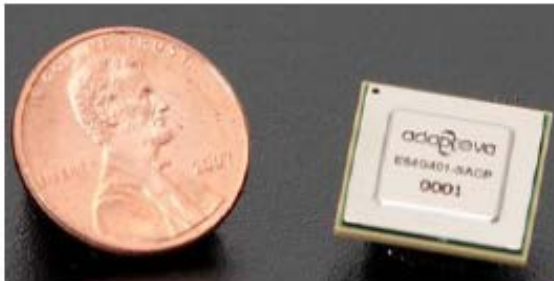
Things I learned while designing the Epiphany & Parallella

Presentation at
Chalmers University of
Technology
Feb 2, 2015



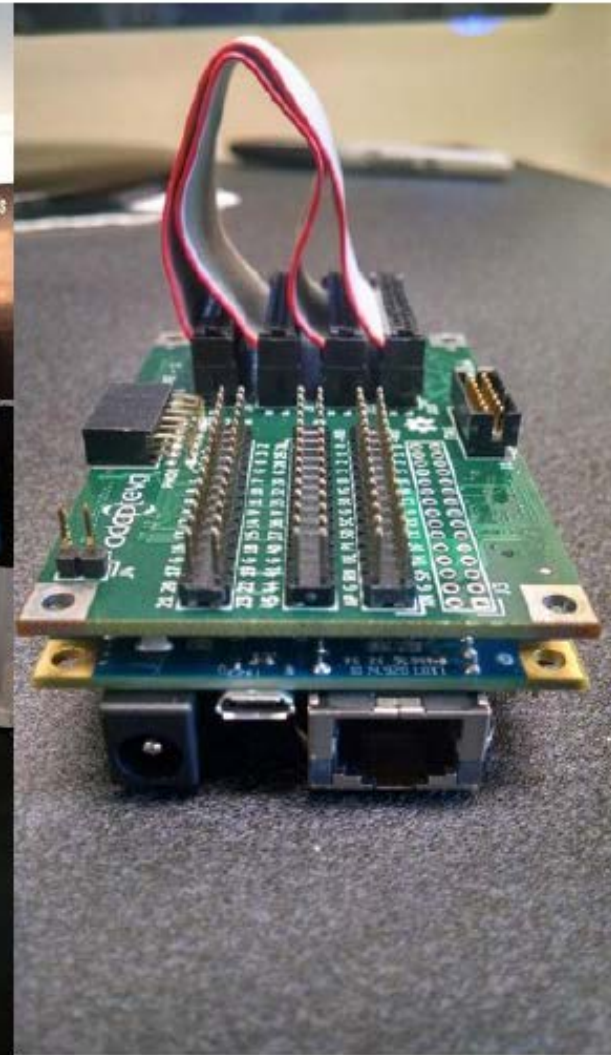
Epiphany-IV

- Aug 2011 tapeout
- (Jul 2012 samples)
- 64 cores, 28nm
- 50 GFLOPS/W
- RTL changes 2 days before TO
- Done in 12 weeks!



More HW...

- June 2014
- Shipped to 200 Universities & 10,000 developers
- Built the "A1", the world's densest cluster.
- Where are the BIG customers????!!!!!!!



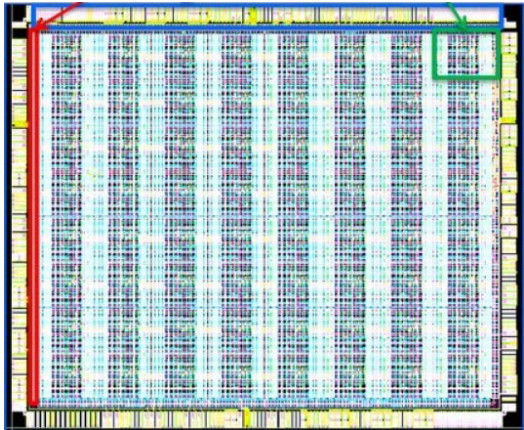
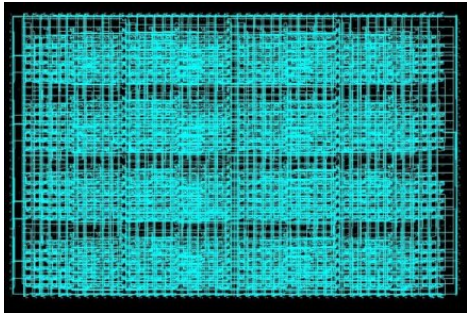
**MY FINAL LESSON:
(took me 7 years to learn)**

**IT'S THE SOFTWARE
STUPID!!!!**

What I (still) Know:

- Moore's law WILL come to an end
- Parallel computing is inevitable
- Architectures like Epiphany are the future
- CPUs, FPGAs, and manycore will coexist
- The world will continue to be driven by \$\$

HW

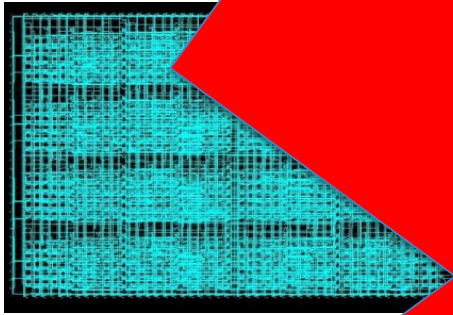


SW

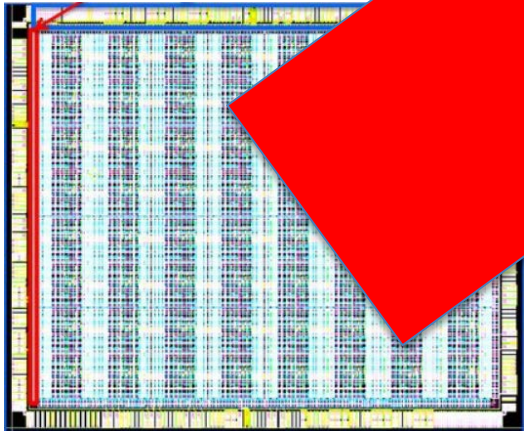
Haskell
Scala
Racket
C

CPU

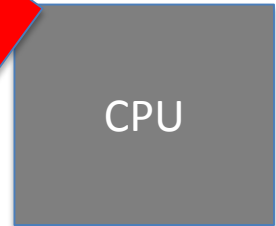
HW

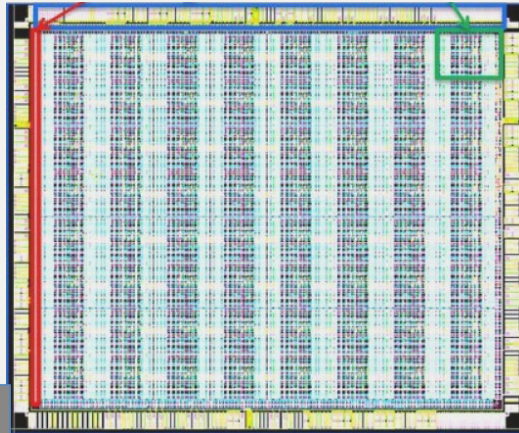
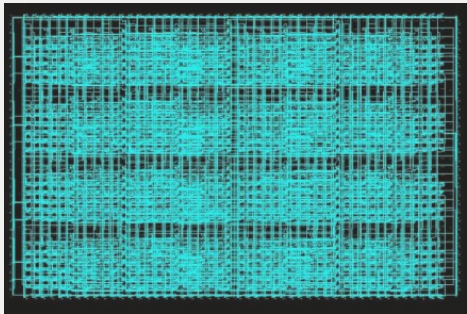


SW



CPU

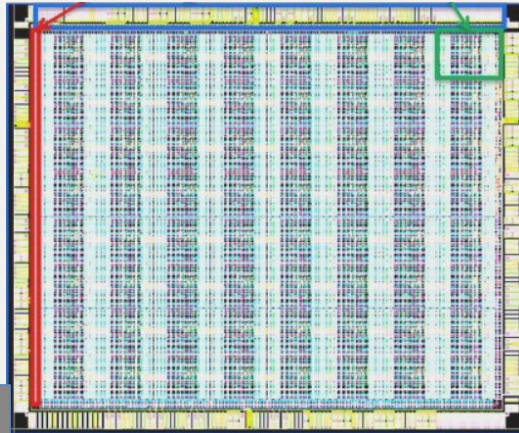
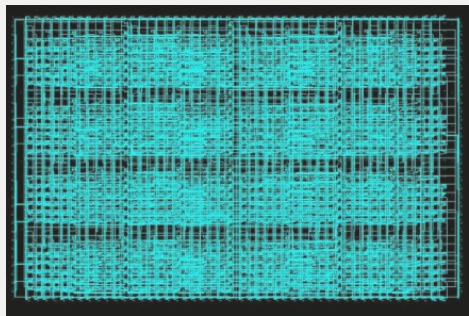




CPU

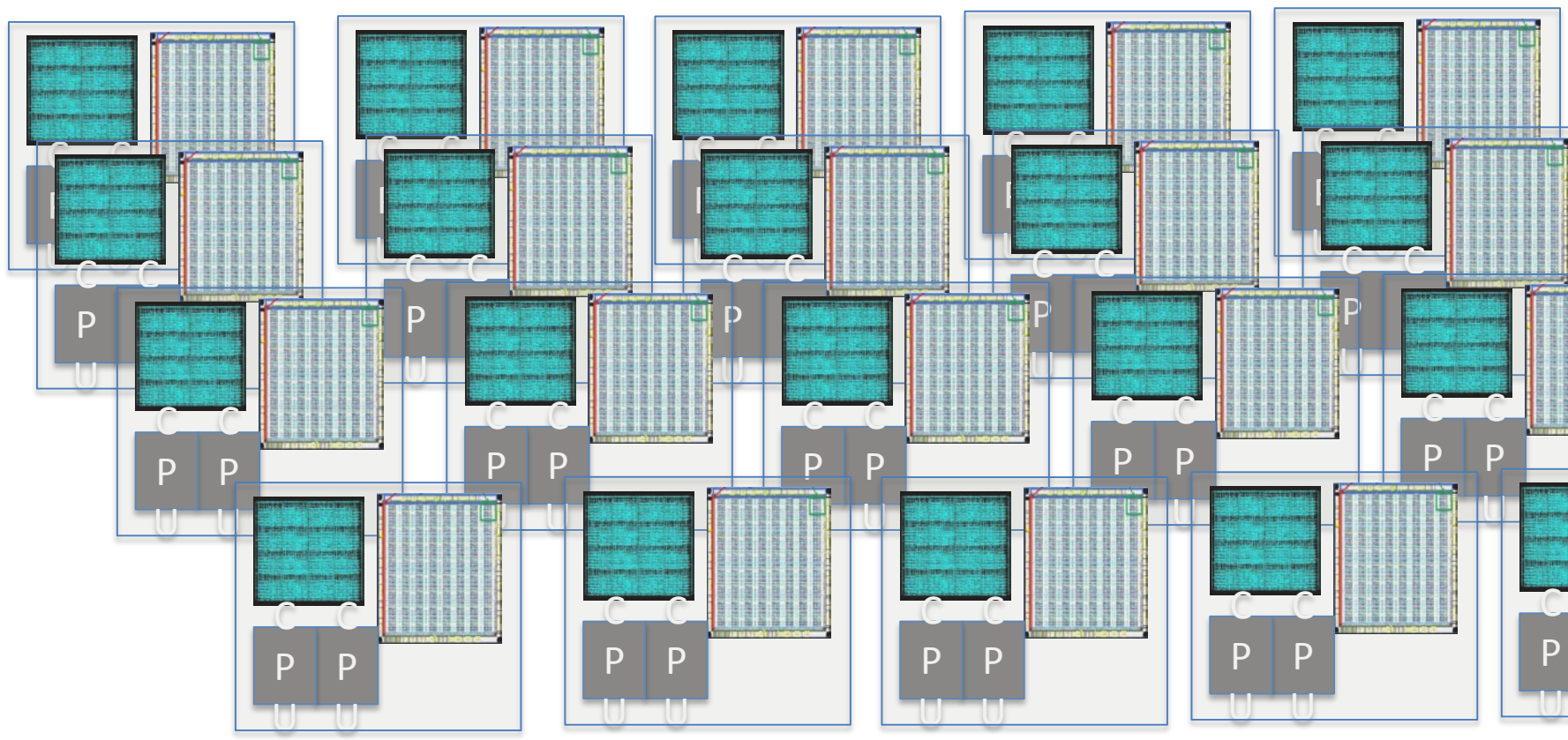
CPU

“The FPGAs are moving into the processors”



CPU

CPU



Programming

Needs to deal with heterogeneity and massive parallelism

Programming

Needs to deal with heterogeneity and massive parallelism

Much relevant work in our community

[Blelloch's ICFP invited talk](#)

[locality work](#)

[Accelerate](#)

[Delite](#)

Yesterday's keynote

Parallelism session this afternoon!

and much more

But STILL I lack a High Level Language to enable THINKING about playing with time and space (the way hardware designers do)

Many have come close

I am thinking about combinators (of course), inspired by [BMMC](#) and much else

Help!

Workshops

[Functional High Performance Computing](#)

[Array](#)

Programming future machines will be more like hardware design than is comfortable!

Not only is FP + HW still interesting! The ideas may be important even just for SW 😊