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

Mary Sheeran
Chalmers
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

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, ...), 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
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.
Quality of designs from an automatic logic generator (ALERT)
Design Automation Workshop (DAC’70)

IBM 1800
Late 70s
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. \( F \cdot G = G \cdot @F \quad \Rightarrow \quad /G \cdot <>F = /(F \cdot G) \cdot mL>F \) (/>)}
Users!

Plessey
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.


work with Plessey done by G. Jones and W. Luk
Kategori:Hårdvarubeskrivande språk
Artiklar i kategorin "Hårdvarubeskrivande språk"
Följande 2 sidor (av totalt 2) finns i denna kategori.

Verilog
VHDL
Algorithms in hardware
In pictures
two

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
two (ilv f)   ilv (two f)
bfly n f = llv (bfly (n-1) f) ->- evens f
bitonic
Batcher’s sorter (bitonic)

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

que
vee
63

60 best
(see Knuth)
median

96

98 Charme

99 Paeth
Graphics Gems I
“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_
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 \( x_1 \circ \ldots \circ x_i, 1 \leq i \leq m \)) by circuits of depth \( \lceil \log_2 m \rceil \) in the case \( m = 2^n \) is exactly

\[
3.5 \cdot 2^n - (8.5 + 3.5(n \mod 2))2^{\lfloor n/2 \rfloor} + n + 5.
\]
notation => play => new algorithms
SEARCH (examples)

SPIRAL
Software/Hardware Generation for DSP Algorithms
SEARCH (examples)

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

SEARCH (examples)

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

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)

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

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)

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

```haskell
// Actual SHA1 computation
let process_chunk : Fun H1, b = 
  let H1 = sslash_hsh H1 in 
  let w = [extend_block t b t | t in 78 -- 0] in 
  let do_iter t [g, d, c, b, a] = 
    let T = (a RORL 5) + (fn t b c d) + e + K160''t + ' + w''t in 
    let e = d in 
    let d = c in 
    let c = b RORL 30 in 
    let b = a in 
    let a = t in 
    [g, d, c, b, a] 
  in 
  let addba = itlist do_iter (79--0) H1 in 
  let Hnew = map2 (defl '{+}') addba Hill in 
  unsplit_hsh Hnew
```

RTL level

```haskell
let top = 
  INTERFACE 
  bit_input clk, 
  op_input op, 
  byte_input a b, 
  byte_output res, 
  byte_output res. 
  CELL "tst" () 
  ff clk a iad # 
  ff clk b lsd # 
  add2 lsd lsd lms # 
  and2 lsd lsd lmp2 # 
  or2 lsd lsd lmp3 # 
  xor2 lsd lsd lmp4 # 
  not1 lsd lms5 # 
  lmp6 <= CASE op [ 
    (ANDop,  lmp2), 
    (ORop,  lmp5), 
    (XORop, lmp4), 
    (NOTop, lmp5) ] '0' # 
  ff clk lmp6 lms # 
  ff clk lmp6 lms # 
);```

With physical placement information

```haskell
// Make a buffer tree for input inp feeding n vertically aligned 
// 'component_height' high components.
list mk_buffer_tree {inc; wr} n component_height->1 driver_at_top->F = 
  n = 0 -> ([[], rpspacer 0 0]) | 
  let name = "hd (depends (dest inp))" in 
  let gspSz = max (MBUFFERING (ceiling (sort (int2float n)))) in 
  let needed = (n*(gspSz-1))/gspSz in 
  let brus = {mk_vars ["inp"] needed :: wr list} then 
  let ninp = {mk_vars ["inp"] needed :: wr} then 
  let rht = row_height * component_height in 
  let minw = (needed*1)*9_cpitch in // Ensure sufficient wiring tracks
  letckt = 
    (driver_at_top => ( 
      wire_o ninp # # 
      (rpspacer ninp ((gspSz/2-1)+rht)) 
    | ( 
      rpspacer ninp ((gspSz/2)+rht)) 
    )
    ) 
  #
  (forall_vbuf [ (i,ot) | zip_in_id (outlast brus) ], 
    wire_o ninp of # # 
    (rpspacer ninp (win ((gspSz/2)-rht) 
      (((driver_at_top->m) + (n-1)- 
      (gspSz/2)+i-1)*gspSz-1)*rht))))) 
  )
  #
  (wire_o ninp (outlast brus)) 
  #
  (driver_at_top => ( 
    (rpspacer ninp 
      (win 0 ((n-(gspSz/2)+1)+(needed-1)*gspSz)) 
    ) | ( 
    (rpspacer ninp 
      (win 0 ((n-1)-(gspSz/2)+1)+(needed-1)*gspSz)) 
    )
    (wire_o inp ninp)
  )
  )
  in 
  let drive_drivers = map (i, cl ((1/gspSz)+1) brus) (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

```plaintext
// Floating point add specification
let hdo 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, sigf1) = split_fp fp1 in
  val (exp2, sigf2) = split_fp fp2 in
  // Restore exponents for denoms and zeros
  let minexp = bias (0-((2^16)-2)) in
  let ox1 = IF (exp1 = 0) THEN minexp ELSE exp1 in
  let ox2 = IF (exp2 = 0) THEN minexp ELSE exp2 in
  // Now, shift fp1 to align with fp2.
  // Pad out both numbers with the internal sticky bit.
  let sigf1' = orshift 68 ren (sigf1 @ [F,F]) in
  let sigf2' = sigf2 @ [F,F] in
  // Perform the sum (or subtract)
  let true_add = (sign fp1 = sign fp2) in
  let sum = IF true_add THEN (sigf2' '+' sigf1') ELSE (sigf2' '-' sigf1') in
  let ox = ox2 in
  let sgn = sign fp2 in
  // Renormalise, if necessary (first renormalisation)
  val (true_ren) = sum in
  let zo = zeros ren in
  let ish = zo '+' 1 in
  // [At this point 0 <= ish <= 63]
  val (nsun, nex1) =
    IF 0 THEN (orshift 1 sum, ex '+' 1) ELSE
    IF (NOT 0 AND NOT 1) THEN (orshift 68 ish sum, (ex '-' ish) ELSE
    (sun,ox)) in
  // Now, round the result according to the current precision
  let nsun2 = hdo pc rc sgn nsun1 in
  // Right-shift renormalisation, if necessary
  let 0 = hd nsun in
  let nsun2 = IF 0 THEN (IF 0 buttlast nsun) ELSE nsun2 in
  let nex2 = IF 0 THEN (nex1 '+' 1) ELSE nex1 in
  IF (NOT (hd(nex nsun2))) THEN
    (IF ((rc = TO_NEG_INF) AND NOT add) THEN [T] ELSE [F]) @
    exzeros @ nsun2
  ELSE ((sgn) @ exzeros @ nsun2)
  ELSE
    // Otherwise, return the answer.
    ([sgn] @ nex2 @ nsun2)
```

Slide provided by Carl Seger (Intel)
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)
Bluespec

FP in HW design

Thanks to R.S. Nikhil (Bluespec)
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)

Design written in BSV language

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

Slide by R.S.Nikhil (Bluespec)
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 combinatorial circuit

Iterate 1 stage thrice

Funneling fewer Bfly4s unfunneling

Varying degrees of pipelining

In any stage, use fewer than 16 Bfly4s

Slide by R.S.Nikhil (Bluespec)
Rule semantics enables compositionality of pipelines

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.

\(^1\)Balancing: latencies may be data-dependent, and different on different paths.
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

Slide by R.S.Nikhil (Bluespec)
**Synthesis from BSV is competitive with hand-coded RTL**

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

<table>
<thead>
<tr>
<th>resolution</th>
<th>VP8</th>
<th>VP8</th>
<th>VP8</th>
<th>VP8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1080p</td>
<td>1x</td>
<td>1x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4Kx2K</td>
<td>2x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Relative silicon area (smaller is better)**

<table>
<thead>
<tr>
<th>functionality</th>
<th>VP8</th>
<th>H.264</th>
<th>BS</th>
<th>VP8</th>
<th>H.264</th>
<th>BS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hand-coded VHDL (reference)</td>
<td>1x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSV</td>
<td>0.18x</td>
<td>0.33x</td>
<td>0.47x</td>
<td>0.81x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Slide by R.S. Nikhil (Bluespec)
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
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
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
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

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.

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

https://www.parallella.org/
“The FPGAs are moving into the processors”
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 😊