Towards Efficient Hardware Compilers

Jean-Baptiste Note Marc Pouzet*
Jean Vuillemin
Ecole Normale Supérieure, 45 rue d'Ulm, 75005 Paris France.

January 30, 2005

Abstract

Version 4: Septembre 04
Optimal hardware synthesis through integer bit-sizing. Optimal storage allocation for overflow integers. Correctness of individual operations: zero divide, array indexes fall within range, positive shifts.
Model checking vs. Abstract Interpretations.
Wotelse?
A faire...

Contents

*P6

```
                                                    // Thresholds table
          var ths   =   [10,15,11,8,10,15,13,12,14,8,14,12,15,9,13,9];
   // Input pixel px                                          // Output di
fun FloydSteinberg(px)  =   di
                  { te  =   px+ Z(pe+e0+7*e);          // total error
                    e0  =   te & 15;                    // error zero
                    eq  =   te ÷ 16;                 // error quotient
                    di  =   ths[e0]<eq;                  // drop ink
                    e   =   di ? eq-16 : eq;             // error one
                    pe  =   Z^l(5*e+Z(3*e+Z(e)));    // previous error
                  }
```

Operator Z denotes the unit delay between consecutive pixels on the same line of the image. Operator Z^l denotes a line delay of $l = 628$ units. Arithmetic operators have the same meaning as in C.

Figure 1: *FloydSteinberg*: source code.

JV↦JBN: livrer 3 codes source isomorphes a *FloydSteinberg* en C (avec des int), en C++ (avec des bigInts), et en Java (avec des Bignums).

Figure 2: *FloydSteinberg*: C and Java code.

1 Summary

Our goal is to automatically compile FPGA hardware from the *software* source code of high performance applications.

Section 2 gives an overview of the proposed techniques. The chosen design implements Floyd-Steinberg's algorithm [**?**] for half-tone rendering digital images with 256 grey levels. It is treated from the source code (Figure 1) to the synthesized circuit *net-list* (Figure 8). The circuit is non-trivial, and as *efficient* as any hand-crafted one. The net-list is compiled to a high-speed & small-size FPGA-hardware to render large digital images on fast single-tone dot printers: with VirtexPro technology, module *FloydSteinberg* operates at 256 MHz and uses 72 slices and 628×9 bits of embedded RAM (Figure 9).

Range analysis is the key to efficient hardware synthesis, and to a number of software verification and optimization techniques as well. Every input integer variable is given an explicit finite range in the source code. For every other internal integer variable, we automatically compute the corresponding finite

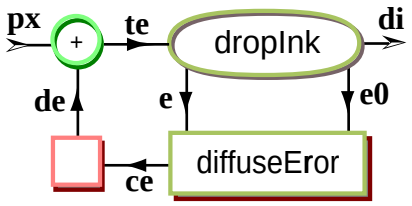


Figure 3: Hierarchical decomposition of *FloydSteinberg*.

range (Figure 5). The ultimate goal of range analysis is *Exact Range Analysis*. Yet, ERA is not computable, in general.

Section 3 presents three *approximate* computational methods for range analysis: H&C, AIA and IIA - ranked by decreasing complexity and precision. Each extends a method which is classic for real values to integer values.

Unlike the three others, the proposed method of *Symbolic Interval Analysis* SIA in section 4 has both a high level of accuracy and a polynomial time complexity. SIA can thus handle practical circuits which are orders of magnitude more complex than *FloydSteinberg*. SIA sometimes provides (for some programs such as *FloydSteinberg* and others) an *optimal bit-sizing* for each integer variable, and near optimal circuits are automatically compiled.

Section 5 presents the overall translation from the source code to some *intermediate code* (bit-sized through range analysis), and from the intermediate code to the *net-list* of the final circuit.

2 Overview

The source code presented for *FloydSteinberg* (Figure 1) is written in Jazz. Jazz [**?**] is an experimental language for hardware description: the language supports strong higher-types and type-inference (à la ML [**?**] or Haskell [**?**]), together with object programming (à la Java or C++) and operator overloading (à la C++).

Despite all these features, the source code (Figure 1, column 2) may be translated, instruction-per-instruction to C or Java (Figure 2). All three codes (Jazz/C/Java) are equivalent: they yield the same Boolean output sequence (*di* - to drop or not to drop ink?) when feed with the same input sequence (*px* - of *integer* pixels). It follows from range analysis (based on $px \in_{[0,255]}$ in Figure 5) that coding each variable with 16 bits signed integers never overflows; all implementations of C will therefore do, since (small) integers in C support (at least) 16 bits signed binary arithmetics.

The *FloydSteinberg* design is used here to illustrate the fine details of automatic integer range analysis. It comes from current *real-life* high-end applications. The source code is short: 16 primitive integer operations in Figure 5. Nevertheless, it exhibits both weaknesses of IIA and AIA: linear and conditional syndromes. The final hardware is non-trivial (Figures 8 and 9), and we do not

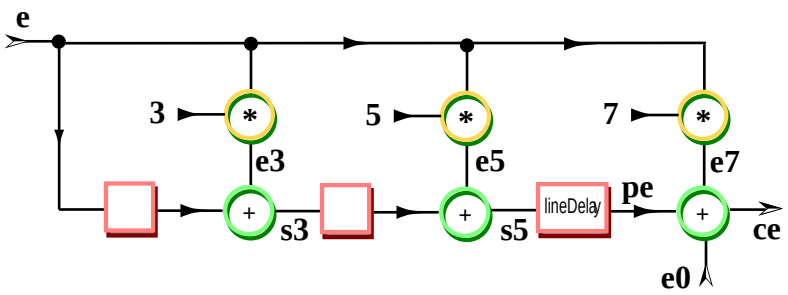


Figure 4: Floyd-Steinberg's error diffusion.

know how to improve by hand on the circuit which is automatically compiled through SIA.

2.1 Floyd-Steinberg's Algorithm

The *FloydSteinberg* design decomposes in two modules (Figure 3):

- the memory-less *dropInk* which controls the ink-nozzle, and keeps track of the current error.
- the module $diffuseError$ which diffuses the current error through a line buffer (Figure 4).

2.1.1 Drop Ink

The input px to *FloydSteinberg* is added to the error de diffused on the spot by the previous pixels: total error $te = px + de = e0 + 16eq$ for $0 \leq e0 < 16$.

The ink-head moves in raster-scan order: one ink-spot per cycle.

The output $di \in_{\{0,1\}}$ from *FloydSteinberg* directly controls the ink-nozzle: it decides to drop or not a dot of ink (worth 256 units) on the current spot.

Decision di results from comparing the error eq to the threshold th. The threshold table $th = ths[e0]$ is randomized to avoid Moiré artifacts.

The final error $fe = te - 256di$ is the difference between the actual and theoretical quantities of ink dropped on the spot. It is passed to $diffuseError$ in the form $fe = e0 + 16e$. Various optimizations result from splitting the error in this way.

2.1.2 Error Diffusion

Module $diffuseError$ diffuses the errors according to the original [?] Floyd-Steinberg filter in Figure 4.

Unlike *dropInk* which is memory-less, $diffuseError$ has 2^{l+2} states, where $l + 2 = 630$ is the number of pixel per line in the image. It reports the diffused error de to *dropInk* on the next clock cycle, through a unit-delay integer-register

var		equation	range set	interval	type
de	=	Z(ce)	$\{n : -112 \leq n \leq 255\}$	[−112,255]	s9
te	=	px+de	$\{n : -112 \leq n \leq 510\}$	[−112,510]	s10
e0	=	te&15	$\{n : 0 \leq n \leq 15\}$	[0,15]	u4
eq	=	te÷16	$\{n : -7 \leq n \leq 31\}$	[−7,31]	s6
th	=	ths[e0]	$\{n : 8 \leq n \leq 15\}$	[8,15]	u4
di	=	th<eq	$\{n : 0 \leq n \leq 1\}$	[0,1]	u1
e	=	di ? eq-16 : eq	$\{n : -7 \leq n \leq 15\}$	[−7,15]	s5
e3	=	3*e	$\{3n : -7 \leq n \leq 15\}$	[−21,45]	s7
e5	=	5*e	$\{5n : -7 \leq n \leq 15\}$	[−35,75]	s8
e7	=	7*e	$\{7n : -7 \leq n \leq 15\}$	[−49,105]	s8
s1	=	Z (e)	$\{n : -7 \leq n \leq 15\}$	[−7,15]	s5
s3	=	Z (e3+s1)	$\{n : -28 \leq n \leq 60\}$	[−28,60]	s7
s5	=	e5+s3	$\{n : -63 \leq n \leq 135\}$	[−63,135]	s9
ec	=	Z^{l}(s5)	$\{n : -63 \leq n \leq 135\}$	[−63,135]	s9
le	=	e0+e7	$\{n : -49 \leq n \leq 120\}$	[−49,120]	s8
ce	=	le+ec	$\{n : -112 \leq n \leq 255\}$	[−112,255]	s9

Figure 5: *FloydSteinberg*: straight line code, exact range, interval, type.

(de = Z(ce) in Figure 3): the *FloydSteinberg* design is thus free of combinational cycle.

2.2 Floyd-Steinberg: Exact Range Analysis

The source code in Figure 1 unfold to the 16 instructions straight-line code in Figure 5, column 1. The range sets corresponding to input $px \in \{0 \cdots 255\}$ are analyzed exactly in Figure 5, column 2 for each variable in the source.

Verifying that the variable ranges in Figure 5, column 2 yield a set fixed-point to the code in column 1 involves an inordinate amount of computation: indeed, $diffuseError$ has well over $2^{2^{1000}}$ binary states! Model Checking analysis methods based on binary state enumeration are doomed to fail with *FloydSteinberg*.

A way to successfully analyze the code is to tag the unit-delay registers in black Z for *feed-back* integer-registers, and in grey Z for *feed-forward* registers: see Figure 5. For the purpose of range-analysis, all feed-forward grey integer-registers can be safely turned to the identity function - hence removed! Once done, there remains but a single feed-back black register in *FloydSteinberg*: $de = Z(ce)$ of type : $s9$ in Figure 5, line 1.

The number 2^9 of states in that unique register is now tractable. The exact ranges (Figure 5, column 2) are computed by this exhaustive method, for all 2^9 states times all 2^8 input pixels, in a few seconds on a GHz PC. Figure 5, column 3 gives the least interval containing the exact set range, and column 4 the derived type: signed/unsigned + number of bits.

	SIA	AIA	IIA
te	[0,255], [−112,510]=	[0,255], [−256,510]···	[0,255], [−287,541]···
e0	[0,15]=	[0,15] =	[0,15]=
eq	[0,15], [−7,31]=	[0,15],[−32,31]···	[0,15], [−34,33]···
th	[8,15]=	[8,15]=	[8,15]=
di	[0,1]=	[0,1]=	[0,1]=
e	[−7,15]=	[−16,15]···	[−16,15]···
e3	[−21,45]=	[−48,45]···	[−48,45]···
e5	[−35,75]=	[−80,75]···	[−80,75]···
e7	[−49,105]=	[−112,105]···	[−143,136]···
s3	[−28,60]=	[−64,60]···	[−64,60]···
s5	[−63,135]=	[−144,135]···	[−144,135]···
ce	[−112,255]=	[−256,255]···	[−287,286]···

Columns 2-4 give the interval sequences computed by three approximate methods. Each entry represents an infinite sequence of intervals. The sign = indicates convergence to the last interval. The sign ··· indicates the first interval of an infinite diverging sequence. Note that SIA converges for all variables to the *best-interval* of Figure 5, column 3. Both AIA and IIA diverge on most variables.

Figure 6: *FloydSteinberg*: approximate range analysis.

2.3 Floyd-Steinberg: Approximate Range Analysis

The ranges reported in Figure 5, column 3 are exactly represented by integer intervals (Figure 6, column 2) for all the variables, except $e3, e5, e7$.

Interval $[i,s]$ denotes the set $[i,s]=\{k\in\mathbf{Z} : i\leq k\leq s\}$.
Working with intervals rather than sets entails an exponential decrease in computational complexity. The exhaustive method compute 2^{18} *integer operations* per variable while we now only compute c *interval operations* per variable. Integer c is the number of cycles before an interval fixed-point is reached for all variables.

However, with the IIA and AIA methods presented in section 3, the value of c is infinite for *FloydSteinberg*: indeed, the interval for variable te roughly doubles at each cycle.

Without any modification to the source code in Figure 1, the proposed SIA method converges in $c = 632$ cycles (for line-width $l + 2 = 630$): within a few milli-seconds on a GHz PC.

If we pre-process the code and remove all grey feed-forward registers in Figure 5, column 1, the SIA method converges in $c = 2$ cycles to the interval fixed point in Figure 6, column 3: within a few micro-seconds on a GHz PC.

More-over, the explicit symbolic invariant $ce = e0 + 16e$ is found by SIA, after removing all grey registers and for the combined modules *dropInk* and $diffuseError$ (Figure 3).

All intervals reported by SIA for *FloydSteinberg* are *best possible*: each can be derived from the extreme values obtained through Exact Range Analysis

software	hardware	operator
te = px+Z(ce)	te[0..9] =px[0..7]+Z(ce[0..8])	u8+Z(s9) : s10
th = ths[te&15]	th[0..3] =ths[te[0..3]]	rom(u4) : u4
di = th< (te≫4)	d[0..5] =th[0..3]−te[4..9], di=d[5]	u4−s4 : u1
e = (te≫4)-16*di	e[0..3] =te[4..7],	s5
	e[4] =te[8]⊕di[0] ⊕ 1,	u1⊕u1 : u1
e3 = (e≪1)+e	e3[0] =e[0]	s7
	e3[1..6] =e[0..4]+e[1..5]	s5+s5 : s6
e5 = (e≪2)+e	e5[0, 1]=e[0, 1]	s8
	e5[2..7] =e[0..4]+e[2..6]	s5+s5 : s6
e7 = (e≪3)-e	e7[0..7] =e8[0..7]-e[0..7]	s8-s8 : s8
	e8[0..2]= 0, e8[3..7]=e[0..4]	s8
s1 = Z(e)	s1[0..4] =Z(e[0..4])	Z(s5) : s5
s3 = Z(e3+s1)	s3[0..6]=Z(e3[0..5]+s1[0..5])	Z(s6+s6) : s7
s5 = Z^l(e5+s3)	s5[0..8]=Z^l(e5[0..7]+s3[0..7])	Z^l(s8+s8) : s9
ce = e7+s5+(te&15)	ce[0..8] =e7[0..7]+s5[0..7]+te[0..3]	s8+s8+s4 : s9

Figure 7: *FloydSteinberg*: bit-sized integer representation.

(Figure 6, column 2).

The intervals reported by SIA for the variables $e3$, $e5$ and $e7$ (Figure 5, column 3) can only approximate the exact set ranges (Figure 6, column 2). Yet, these approximations are close enough to have no incidence on the other variables, all of which are exactly. analyzed

2.4 Floyd-Steinberg: Bit-Sizing

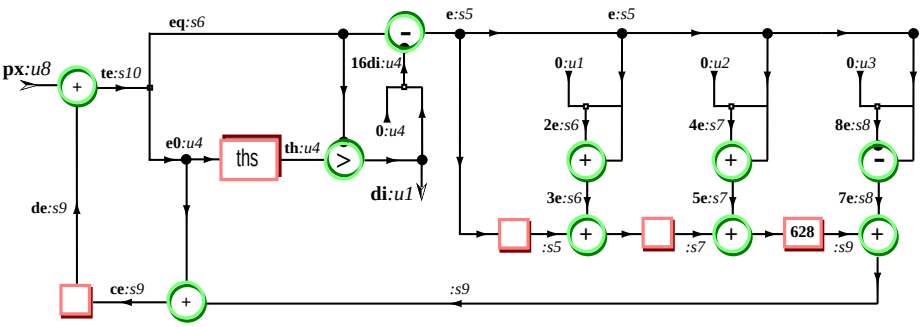
Exact range analysis (Figure 5) is the key to optimal coding for each variable by a finite array of bits. A lower bound on the size of any binary encoding is the binary length $l_2(n)$ of the number $n \in \mathbf{N}$ of elements in the range set.

The types in Figure 5, column 3 are: un for un-signed n bits, and sn for signed two's complement n bits. The number n of bits is the least natural number such that interval - $_{[0,2^n-1]}$ for un-signed variables, $_{[-2^{n-1},2^{n-1}-1]}$ for signed ones - contains the least ERA range set from Figure 5. In reality, the types are computed by the practical SIA method (Figure 6, column 1). With *FloydSteinberg*, the interval fixed-point computed by SIA is equal to the ERA interval, for all variables.

Each software variable in Figure 7, column 1 is represented by the minimal binary array containing the type of the variable, from Figure 5, column 4. The resulting hardware level representation maps the variables to finite array of bits in Figure 7, column 2 and operations to finite operators over binary arrays in column 3.

The n-th bit $\mathrm{B}_n^k \in_{\{0,1\}}$ of $k \in \mathbf{Z}$ is determined by the 2-adic expansion [**?**]

$$k = \sum_{n \in \mathbf{N}} \mathrm{B}_n^k 2^n.$$



Variable th:$u4$ is represented by an array th$[0,3]$ of 4 binary nets; its integer value is $th = \sum_{n<4} th[n]2^n$: an un-signed number $th \in \mathbf{N}$
Variable ce:$s9$ is represented by an array ce$[0,8]$ of 9 binary nets; its integer value is $ce = -256ce[8] + \sum_{n<8} ce[n]2^n$: a signed number $ce \in \mathbf{Z}$.

Figure 8: Module *FloydSteinberg*: final circuit schematics.

The 2-adic expansion is used to sign-extend the ranges, as seen in Figure 7, column 2 for the 12 occurrences of variable e.

2.5 Floyd-Steinberg: Hardware Synthesis

From the bit-sized intermediate code in Figure 7, we automatically derive a binary net-list whose total number of nets is the sum of the bits for all variables (less than 100 nets for *FloydSteinberg*).

The final circuit schematics in Figure 8 results by implementing bit-vector integer operations with standard hardware custom-sized arithmetic operators.

The FPGA design which results at the end of the compilation process is measured in Figure 9.

The accuracy of range analysis is mandatory to generate the smallest possible hardware; we are fortunate here that SIA and ERA should yield the same intervals.

Let us compare the number of bits used for coding each variable in the circuit of Figure 8, to the optimal $l_2(n)$ obtained for the n range values in Figure 5, column 2. The straight binary encoding in Figure 7 is optimal, for all variables but four: $e3, e5, e7$, and th.

The range of th is $[8,15]$; th can indeed be coded with 3 bits $th[0..3]$ rather than 4, since $th[4] = 1$ is constant.

The binary representations of $3e$, $5e$ and $7e$ are all one-to-one mapped to that of $e : s5$; 5 bits are thus necessary and sufficient to code all four variables at once! The actual values of $e3 : s7$, $e5 : s8$ and $e7 : s8$ are then derived by 3 look-up tables, with 5 bits of input and, respectively 7,8 and 8 bits of output.

Depending on the specific FPGA used for measurement, this may or may not be a size/speed improvement over what gets directly compiled from the bit-sized code in Figure 7 and reported in Figure 9.

instruction	nets	slices	ram
te = px+Z(ce)	9	8	
th = ths[te&15]	4	4	
di = th< (te≫4)	1	5	
e = (te≫4)-16*di	5	1	
e3 = (e≪1)+e	7	6	
e5 = (e≪2)+e	8	7	
e7 = (e≪3)-e	8	7	
s1 = Z(e)	5		
s3 = Z(e3+s1)	7	6	
s5 = Z^l(e5+s3)	9	8	5652
ce = e7+s5+(te&15)	9	12	
total	72	64	5652

Figure 9: Module *FloydSteinberg*: compiled FPGA circuit.

3 Range Analysis

3.1 Exact Range Analysis

In theory, *exact range analysis* ERA of an integer variable in a computer program goes as follows: assign a fixed *finite* set of values to each input variable and initial storage location; execute the program over all such inputs, and compute the corresponding set of integer values for each variable. We then witness one of the following possible outcomes.

- We run out of memory or time.
- We detect that analysis is failing, because of some abnormal condition - say division by 0 - and construct a finite input leading to the anomaly.
- We detect that analysis is failing, because some variable has a provably infinite range.
- We successfully terminate: each integer variable in the program is then determined to have a finite set of possible values, which altogether provide us with the *unique least fixed-point* solution to the initial set of equations defined by the program statements and by the input conditions.

In theory, none can better ERA; in practice, ERA is hardly usable:

1. whether or not a variable has *finite range* is *not decidable* [**?**].
2. if we add an input parameter $c \in \mathbf{N}$ which bounds the number of iterations for each integer simulation, computing the range of a variable has an *intrinsic exponential time complexity.*[1]

[1] Suis pas trop sur de ca!

3.2 Approximate Range Analysis

ERA being untractable, one must resort to *approximate* methods. We list IIA, AIA, SIA and H&C in order of increasing complexity and precision.

Range analysis with H&C is somewhat different from the other methods (section 3.4).

To perform range analysis with IIA, AIA or SIA, we symbolically execute the program on intervals, and monitor how the intervals vary from one input cycle to the next. Analysis succeeds if an interval set is found which remains a *fixed-point* from one cycle to the next for all variables. Let $c \in \mathbf{N}$ be the number of cycles through the design before reaching the fixed-point.

When successful ($c < \infty$) the reported interval contains the exact range. But it can be much larger. Thus, a division by an interval containing 0 does not imply that a division by 0 happens in exact analysis. Nevertheless, a division by an interval $_{[i,s]}$ such that $i > 0$ or $s < 0$ is guaranteed free of zero divide.

With all approximate methods, one must face the situation of a diverging interval analysis. Such will be the case with the code $y = Z(y + x)$ for a binary $x \in_{[0,1]}$ counter. The correct analysis shows that the value of output y at cycle n is $y_n \in [0, n]$: we cannot tell how many bits this counter requires, except by looking at the ways in which integer y is used.

A convenient alternative is to allow the designer to comment the source code with explicit interval casts, such as $y = cast(Z(y + x), [0, 15])$ if all one needs is some 4 bits counter.

Explicit casting gives the ability to force the convergence of interval analysis through comments in the source code. For example, replacing line 1 in Figure 5 by $de = Z(cast(ce, [-112, 255]))$ is sufficient to cause the convergence of the analysis, for both IIA and AIA, while they both fail without the cast comment in the source.

There is a subtle difference between the hardware generated for this cast code by the IIA, AIA and SIA methods. With SIA, it is verified (through one cycle of computation) that $ce \in [-112, 255]$ is an interval fixed-point for *FloydSteinberg* as a whole. The cast is thus useless, and it is safely removed. With SIA, the generated hardware is the same, regardless of the cast. With IIA and AIA, it is found (Figure 6) that the intervals for *ce* contains the fixed-point in a strict manner. This gets reflected in the generated hardware by a custom overflow detector, whose sole function is to verify that the cast condition is never violated up to the current cycle.

3.2.1 Integer Interval Analysis

The classical theory [**?**] of interval analysis only applies to operations over real numbers represented by to rational end-point intervals. It is extended here to *Integer Interval Analysis* IIA which supports all primitive integer operators: quotient/remainder, left/right shifts, comparisons </=, logical not/and/or/xor, memory operations RAM/ROM and unit delay Z.

Each operator is applied symbolically over intervals through a few (say, less

than 10) operations over the corresponding integer end-points . IIA is thus nearly as fast as regular integer simulation: the required memory is n (the number of integer variables in the source code) and the running time is proportional to cn, where c is the number of cycles it takes for range analysis to converge to some finite interval for each variable.

Monotone Operators Interval arithmetic is most efficient at implementing integer operations which are *monotone* (either increasing or decreasing), with respect to all arguments.

$$
\begin{array}{lrcl}
Add: & [i,s]+[l,h] & = & [i+l, s+h]. \\
Sub: & [i,s]-[l,h] & = & [i-h, s-l]. \\
Not: & \neg[i,s] & = & [\neg s, \neg i]. \\
Mul: & [i,s]\times[l,h] & = & [min(il,ih,sl,sh), max(il,ih,sl,sh)]. \\
lShift: & l\geq 0 \Rightarrow [i,s]\ll[l,h] & = & [min(i\ll l, i\ll h), max(s\ll l, s\ll h)]. \\
rShift: & l\geq 0 \Rightarrow [i,s]\gg[l,h] & = & [min(i\gg l, i\gg h), max(s\gg l, s\gg h)]. \\
Div: & l>0 \Rightarrow [i,s]\div[l,h] & = & [min(i\div h, i\div l), max(s\div h, s\div l)]. \\
Div: & h<0 \Rightarrow [i,s]\div[l,h] & = & [min(s\div h, s\div l), max(i\div h, i\div l)].
\end{array}
$$

Non-monotone Operators The bit-wise logical *and* is not a monotone function over the integers. Approximate formulaes must be worked out:

$$
\begin{array}{rcl}
i\geq 0, l\geq 0 \Rightarrow [i,s]\cap[l,h] & = & [0, min(s,h)], \\
i\geq 0, l<0 \Rightarrow [i,s]\cap[l,h] & = & [0,s], \\
i<0, l\geq 0 \Rightarrow [i,s]\cap[l,h] & = & [0,h], \\
s<0, h<0 \Rightarrow [i,s]\cap[l,h] & = & [-2^k, min(l,h)], \\
\text{else } \Rightarrow [i,s]\cap[l,h] & = & [-2^k, max(l,h)],
\end{array}
\qquad (1)
$$

where $k = max(l_2(\neg i), l_2(\neg l))$ and $l_2(n) = \lfloor \log_2(n+1) \rfloor$ is the binary length of $n \in \mathbf{N}$. Definition (1) can never fail as the mutually exclusive pre-conditions cover all cases. Note that formula (1) coincide with the best-set ERA definition for intervals of the form $[0,s]$ and $[0,h]$.

The remaining logical operations are reduced through Boolean Algebra. The conditional operator is treated naively.

$$
\begin{array}{lrcl}
Or: & [i,s]\cup[l,h] & = & \neg(\neg[i,s]\cap\neg[l,h]). \\
Xor: & [i,s]\oplus[l,h] & = & (\neg[i,s]\cap[l,h])\cup([i,s]\cap\neg[l,h]). \\
Cond: & [0,1]\,?\,[i,s]:[l,h] & = & [\min(i,l), \max(s,h)].
\end{array}
$$

Interval remainder can be defined by

$$
l>0 \text{ or } h<0 \Rightarrow [i,s] \cdot|\cdot [l,h] \quad = \quad [\max(0,g), \min(|h|-1, d)], \qquad (2)
$$

where $[g,d]$ is the interval computed from the interval quotient q by $r = n - dq$ and the previous interval formulaes.

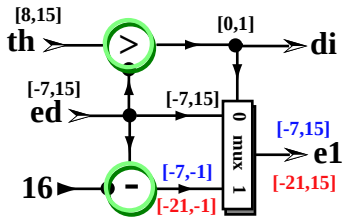


Figure 10: The *nozzle* after symbolic interval analysis: best intervals at the top, IIA and AIA intervals at the bottom (when different).

3.3 Affine Interval Arithmetic

A basic problem with IIA relates to the *linear syndrome*: for $p \in [0, 255]$, expression $a = p - p$ must evaluate to $a = [0, 0]$; not to $a = [-255, 255]$ which is the value computed by IIA.

In the case of real numbers, *Affine Interval Arithmetic* AIA is a popular cure [**?**, **?**] for the linear syndrome. An affine interval e takes the form $e = i + \epsilon l$ where $i, l \in \mathbf{R}$ and $\epsilon \in \{-1, 0, 1\}$ is a symbolic variable representing the mid-point of $e = [i - l, i + l]$. The expression $e - e$ evaluates to the correct value $(i - i) + \epsilon(l - l) = 0$ with AIA.

The bit-array intermediate code in Figure 7 reduces the products of e by $3, 5, 7$ to shift and add operations:

$$e3 = (e << 1) + e$$
$$e5 = (e << 2) + e$$
$$e7 = (e << 3) - e$$

Analysis with IIA for $e \in [-7, 15]$ yields $e7 \in [-71, 127]$: linear syndrome! Both AIA and SIA correctly analyze $e7 \in [-49, 105] = 7 \times [-7, 15]$.

3.4 Halbwachs & Cousot algorithm

A remaining problem with AIA relates to the *conditional syndrome*: for $p \in [0, 255]$, the conditional expression $a = p < 0\,?\, -p : p$ must evaluate to $a = [0, 255]$; not to $a = [-255, 255]$ which is computed both by IIA and AIA.

The conditional syndrome appears in *FloydSteinberg* with the *Nozzle* expression $e = di\,?\,eq - 16 : eq$ where $di = (th < eq)$ in Figures 1 and 10, and its equivalent form $e = eq - 16 * di$ in Figure 7.

The strongest known cure to the *conditional syndrome* uses the Halbwachs & Cousot H&C algorithm [**?**]. H&C deals with a large number of linear inequalities between symbolic variables. Yet, algorithm H&C is difficult to implement correctly and in full generality: the original method [**?**] does no cover the full range of integer operations covered by IIA. As it is based on the SIMPLEX method, the computational complexity of H&C is another limiting factor, even for medium size integer circuits.

4 Symbolic Interval Analysis

The proposed method of *symbolic interval analysis* SIA fixes both syndromes at once: it is strictly more powerful than AIA, as it correctly analyzes many conditional statements; it may not always be as precise as H&C, but the computational complexity of SIA is far lower.

The SIA methods lets us synthesize a number of optimal non-trivial real-life hardware, automatically from the source program for designs which are much more complex than the simple example treated here.

With a *feed-forward* design (either memory-less or not), range analysis with SIA takes time proportional to the number of integer variables in the source code, regardless of feed-forward memories.

With a *feed-back* design (hence some local memory, as in *FloydSteinberg*), range analysis takes time proportional to the source code length and to the number c of iterations required to reach an interval fixed-point for all variables (or to time-out and report suspected divergence on some variables).

4.1 Overview

The data-structure used first extend integer expressions with symbolic linear expressions.

A unique normal form is obtained by systematically sharing equal sub-expressions and by ordering variables, much like the *Binary Moment Diagram* BDM data-structure of [**?**].

Once done, it is a simple matter to recognize that $p - p = 0$ or that $(p << 2) - p = 3p$ since $p - p$ and $4p - p$ are represented by symbolic linear expressions in the same formal variable p, with respective normal forms 0 and $3p$.

Such *symbolic integers* serve in turn to represent the end-points of *symbolic intervals*. The *linear syndrome* is solved by SIA as by AIA.

The proposed SIA is also more precise than AIA: interval $[0, 1]$ can only be approximated by $[0, 2] = 1 + \epsilon$ or by $[-1, 1] = \epsilon$ with AIA; it is represented exactly with SIA.

The flexibility of SIA allows to handle conditionals. For example, equation

$$m \quad = \quad a < b \,?\, b : a \tag{3}$$

is computed over intervals $a = [-1, 3]$ and $b = [1, 2]$ by introducing an auxiliary symbolic *cut* variable c and by evaluating each branch of the conditional in 3 within a specific context. The context for the case $a < b$ is:

$$a < b \Rightarrow a \in [-1, c - 1],\ b \in [c, 2],\ c \in [1, 2]$$

which evaluates to $b \in [1, 2]$. The context for the case $a \geq b$ is:

$$a \geq b \Rightarrow a \in [c, 3],\ b \in [1, c],\ c \in [1, 2]$$

which evaluates to $a \in [1, 3]$. Combining both cases $a < b$ and $a \geq b$ yields the correct result $m \in [1, 3]$.

The *cut method* is both simpler and faster than H&C. In exchange, *cut* can be less precise and programs simply constructed where it fails while H&C succeeds.

4.1.1 Data Structure Details

A BDD [?] over all decision variables - $v = (v' < v'')$ or $v = (v' = v'')$ - at the top of the data-structure.

At the leaves of that BDD, one finds the BMD-like linear expressions of the form $c \times v + v'$ for $c \in \mathbf{Z}$, where v' is a shallower $h(v) > h(v')$linear expression and v is a primary variable.

Primary variables comprise all logical and other non-linear integer operations. Their interval values are computed by the rules of IIA given in the section 3.2.1.

Hybrid discrete-set/interval analysis?

5 Hardware Compiling

In C or Java, the source code must be pre-processed and syntactically analyzed in order to perform range analysis. Not so in Jazz where we rely on operator-overloading to execute the very same source code, with rather selected and different types of inputs:

1. an integer input sequence yields for output the result of software simulation: some long bit-sequence;

2. an interval-type input sequences yields the range analysis of each variable for output;

3. a type array of 8 nets input yields for output the net-list of a finite digital synchronous arithmetic circuit: the circuit's behavior mimics bit-per-bit that of the software (modulo the I/O binary conversion).

Indeed, all variables in the source code (Figure 1) have the same type: T is any type which supports the integer operations $\{+,-,*,\div,\&,<\}$ in the code.

var	source	remZ	lin /cond
de =	Z(ce)		
te =	px+de		
e0 =	te ·|· 16	te&15	
eq =	te÷16	te ≫4	
th =	ths[e0]		
di =	th<eq		
e =	eq-16*di	di ? eq-16 : eq	sup(eq_0, $eq_1 - 16$) $eq_1 = \max(eq, 1 + th)$ $eq_0 = \min(eq, th)$
e3 =	3*e	e+e≪1	3e
e5 =	5*e	e+e≪2	5e
e7 =	7*e	-e+e≪3	7e
le =	e0+e7		e0+7e
s1 =	Z (e)	e	
s3 =	Z (e3+s1)	e3+s1	4e
s5 =	e5+s3		9e
ec =	Z (s5)	s5	9e
ce =	le+ec		e0+16e

5.1 Process

1. Topological sort: linear time/space.
2. Remove feed-forward registers: linear time/space.
3. Linear expressions
4. Conditionals
5. SIA
6. Constant folding
7. Bit sizing
8. Operator synthesis

var	Interval
de = Z(e0+16*e)	$[Z(e0_i+16e_i), Z(e0_s+16e_s]$
te = px+de	$[ce_i+de_i, ce_s+de_s]$
e0 = te&15	$[0, \min(te_s, 15)]$
eq = te ≫4	$[te_i \div 16, te_s \div 16]$
th = ths[e0]	$[\min(S), \max(S)]$, $S = \{ths[k] : k \in [e0_i, e0_s]\}$
$eq_0 = \min(eq, th)$	$[\min(eq_i, th_i), \min(eq_s, th_s)]$
$eq_1 = \max(eq, 1 + th)$	$[\max(eq_i, 1+th_i), \max(eq_s, 1+th_s)]$
di = th<eq	$th_s < eq_i$? [1,1] : $eq_s \leq th_i$? [0,0] : [0,1]
e = sup(eq_0, $eq_1 - 16$)	$[\min(eq_{0i}, eq_{1i} - 16), \max(eq_{0s}, eq_{1s} - 16)]$