fast implementation of CV algorithms

using floating point hardware for numeric intensive algorithms

Ilia greenblat@mac.com
+972-54-4927322

overview

* recent projects i participated, especially at Inuitive, all included substantial numerical computations.
* e.g: computer vision, e-motor control and DSP signal processing.
* different fields, but all share problems of implementation. Mainly effort / time, it takes to create and verify RTL .
* today i talk about algorithms with limited set of input values, limited set of output values and various amount of number crunching in between.

where this stuff goes?

INUITIVE

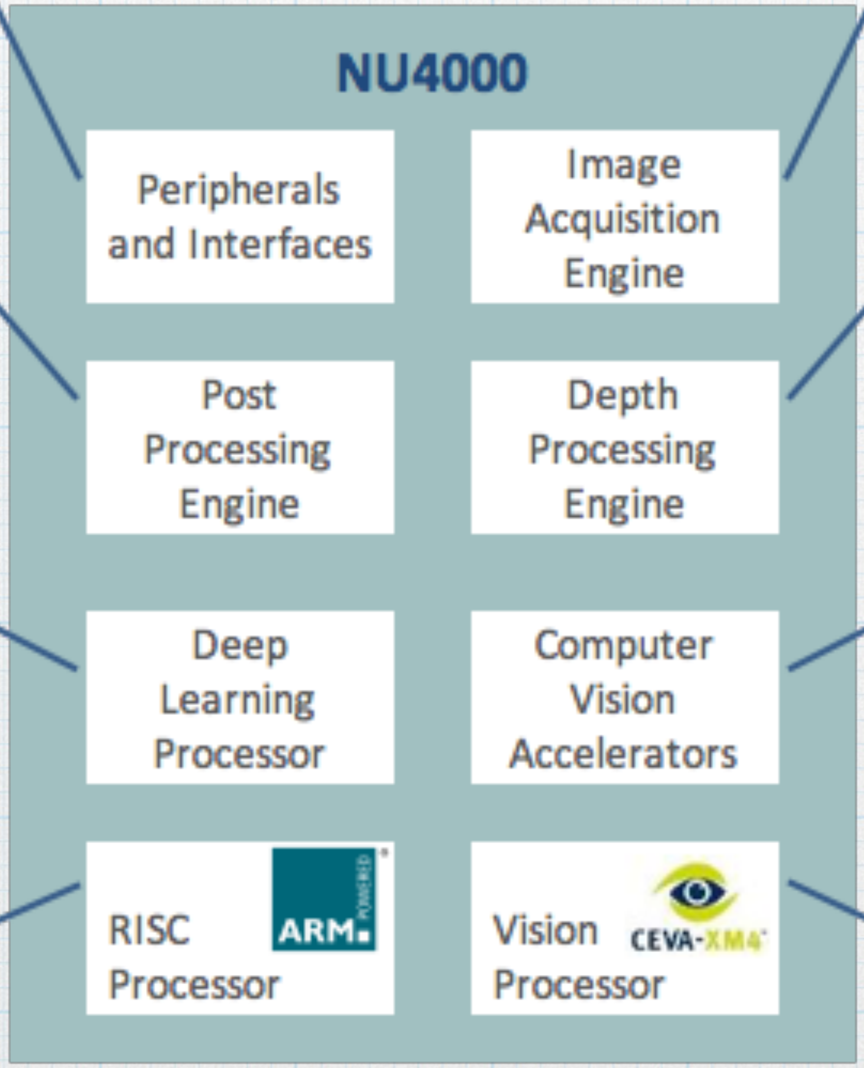
NU4000 - BEST IN CLASS 3D IMAGING & VISION PROCESSOR

LPDDR2, LPDDR3,
USB2.0/3.0
4xUART, 2xSPI, 6xi2C,
I2S, 48xGPIO, 6xTimers,
2xWDT
Global Timing Unit

Image Scalers
ROI Cropping blocks
Histograms
Data Packing
Disparity to Depth
RGB-Depth Registration

Fully programmable
Convolutional Neural
Networks Engine.
Process >100 AlexNet
ROIs per seconds.
Reduce DDR bandwidth

ARM Cortex-A5 + FPU
& Neon co-processor
Running at 1GHz with
Embedded Linux OS



6 x MIPI-Rx Ports
3 x MIPI-Tx Ports
Stereo Pre-Processing –
Distortion Rectification
(120x90Deg), Images
alignment and balance

State of the Art Depth
from Stereo Engine,
3rd generation design
120Mp/s - FHD@60fps,
720p@120fps
Below 1msec latency

!!!

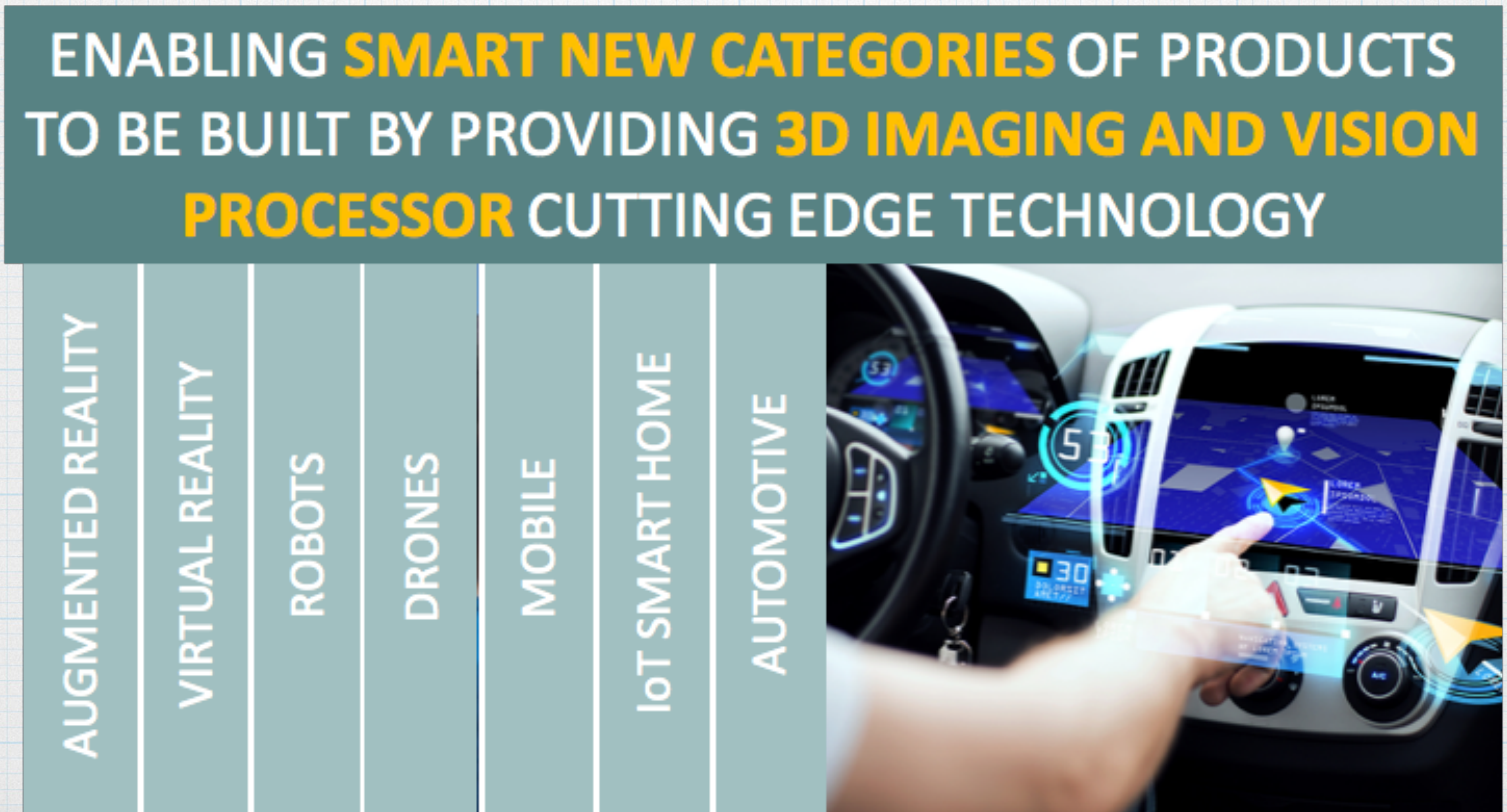
Very low power & zero
Latency Vision Modules
DoG, HoG, FAST, ORB,
FREAK & Normal Estimation
– HD@60fps or
2xVGA@60fps

freak dog

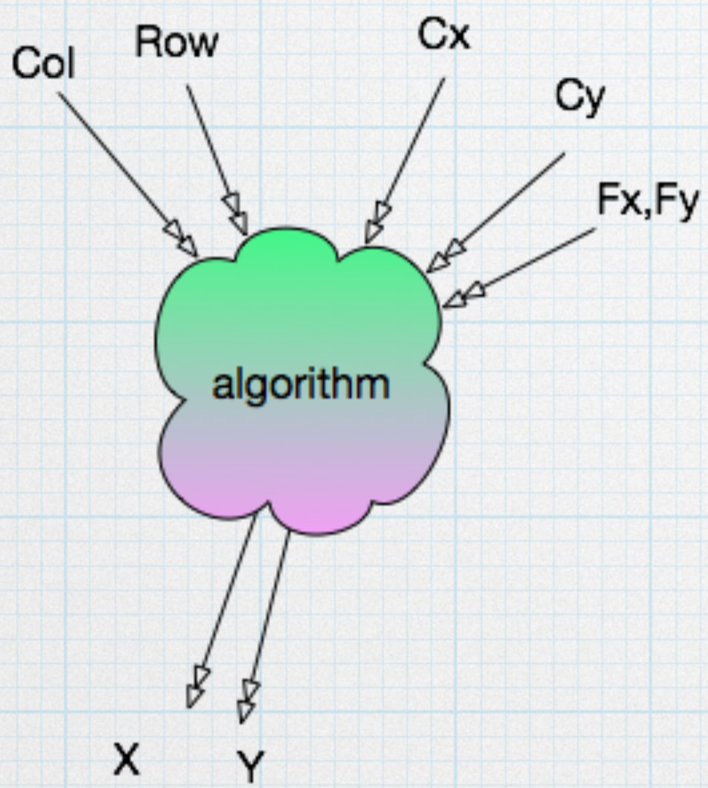
Ceva XM4
Most advanced Vision
Processor in the market
running at 800MHz

What it is good for?

ENABLING SMART NEW CATEGORIES



like:



fisheye camera distortion: translation of coordinates.

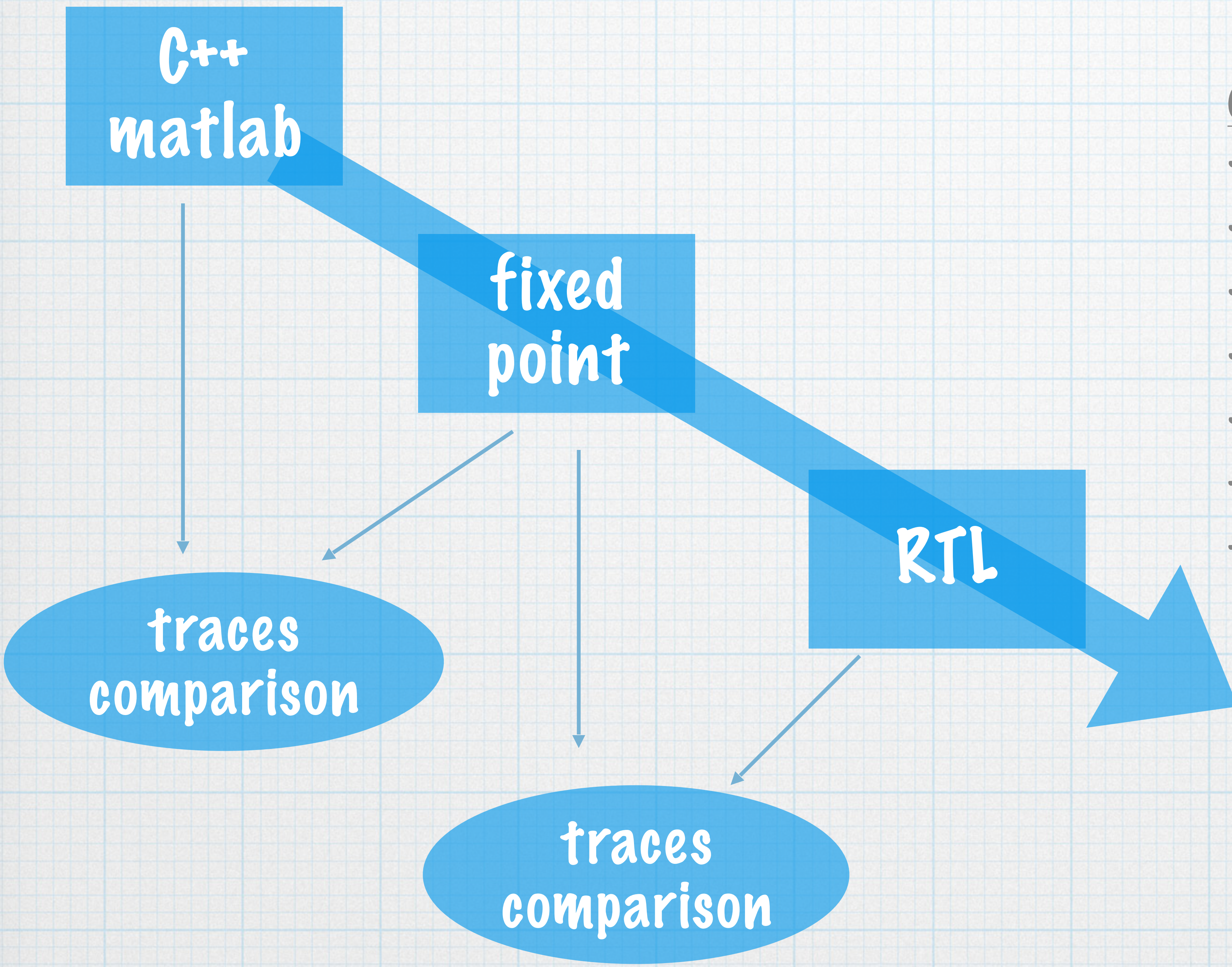
sensorless PM motor: current loop and speed control.

less so: Goertzel algorithm in DSP.

pixel mask 5x5 computation

* limited set of input values, limited set of output values and various amount of number crunching in between.

common flow



challenges to overcome:

- accuracy problems
- overflows
- fixed point dynamic range
- signed / unsigned
- tedious implementation
- tricky verification
- backend uncertainties.

what is different

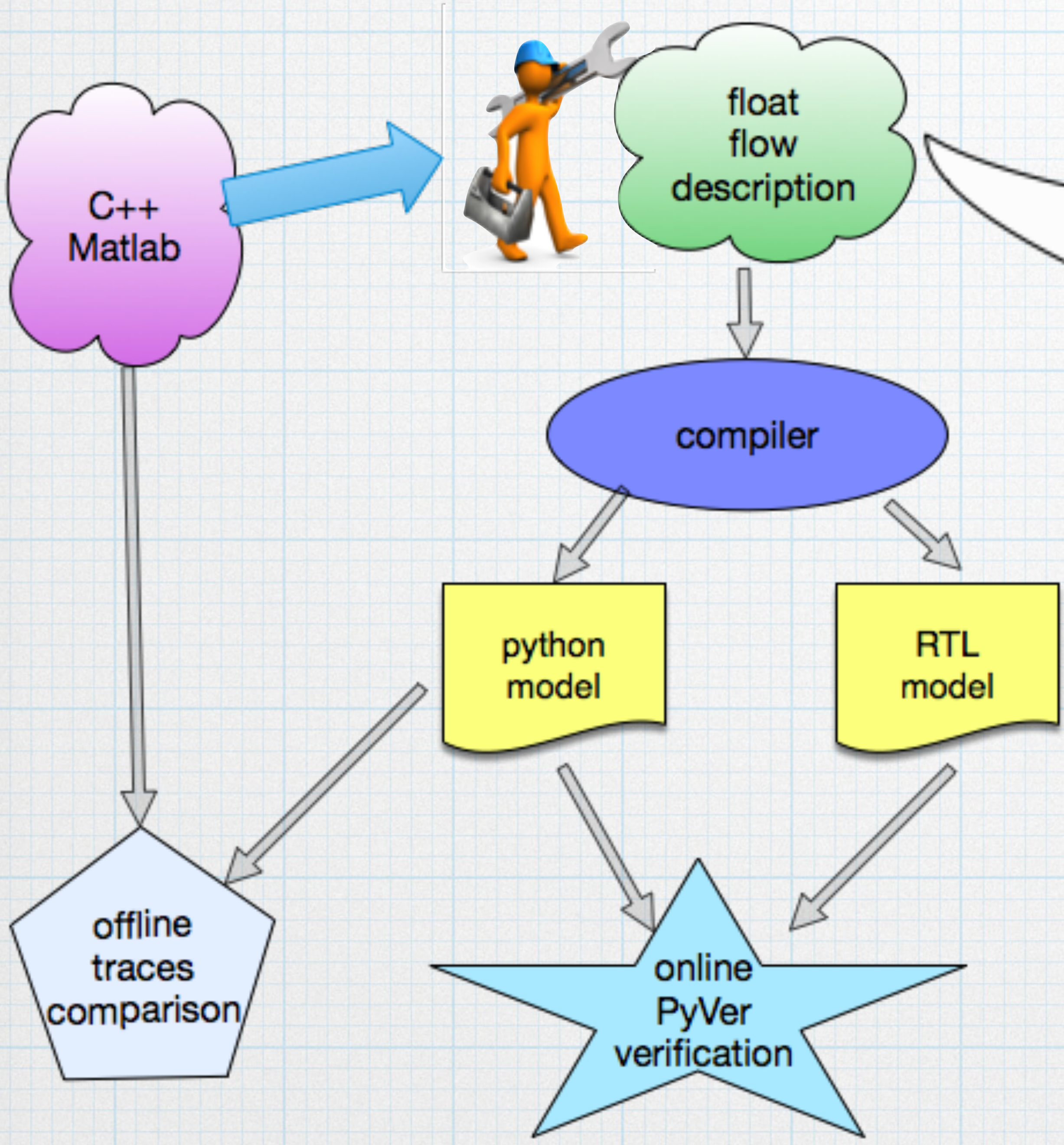
all computations : performed in floating point.

Floating point hardware has a bad rep:

area
timing
power
complexity

That might be true for "CPU" standard-adhering floating point, but is not so for embedded hardware.
Here we may change the width of fields and the treatment of exceptions.
In the end all variables keep the same format, which keeps the flow simple.

Flt flow

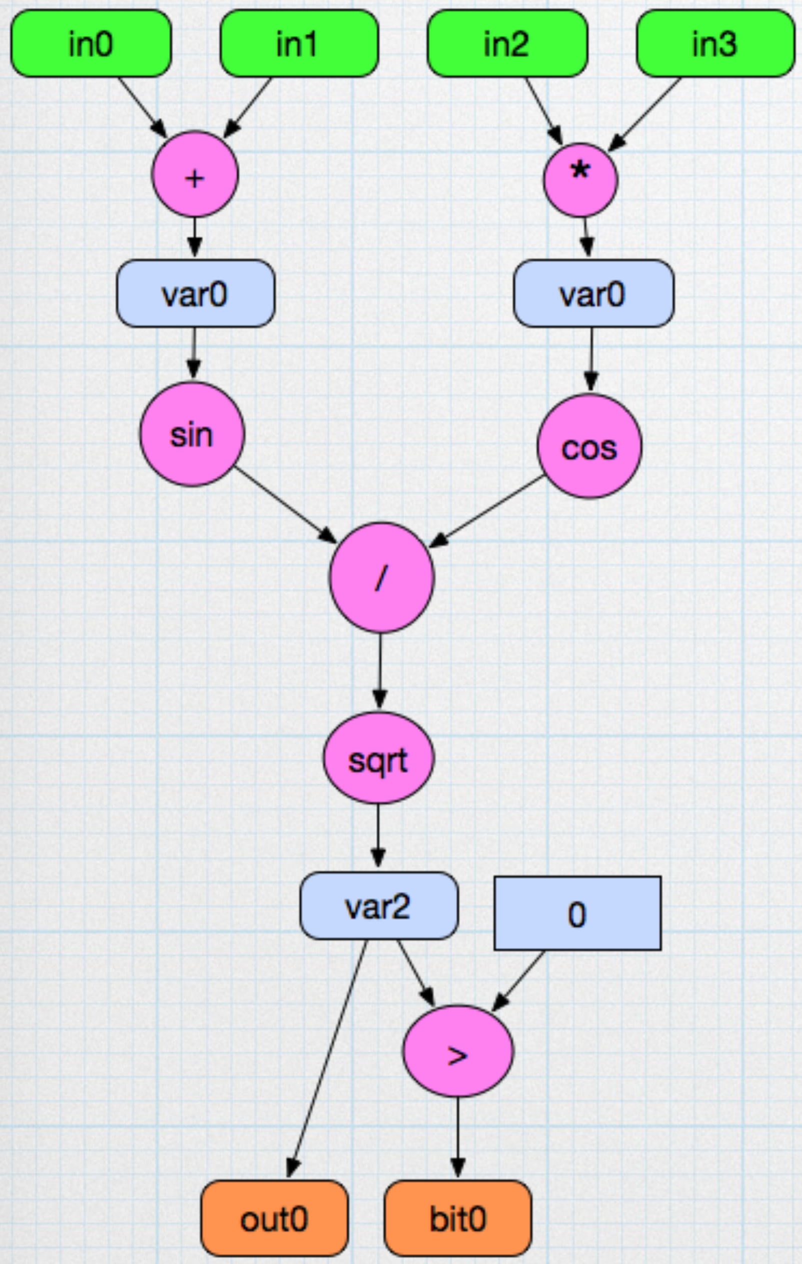


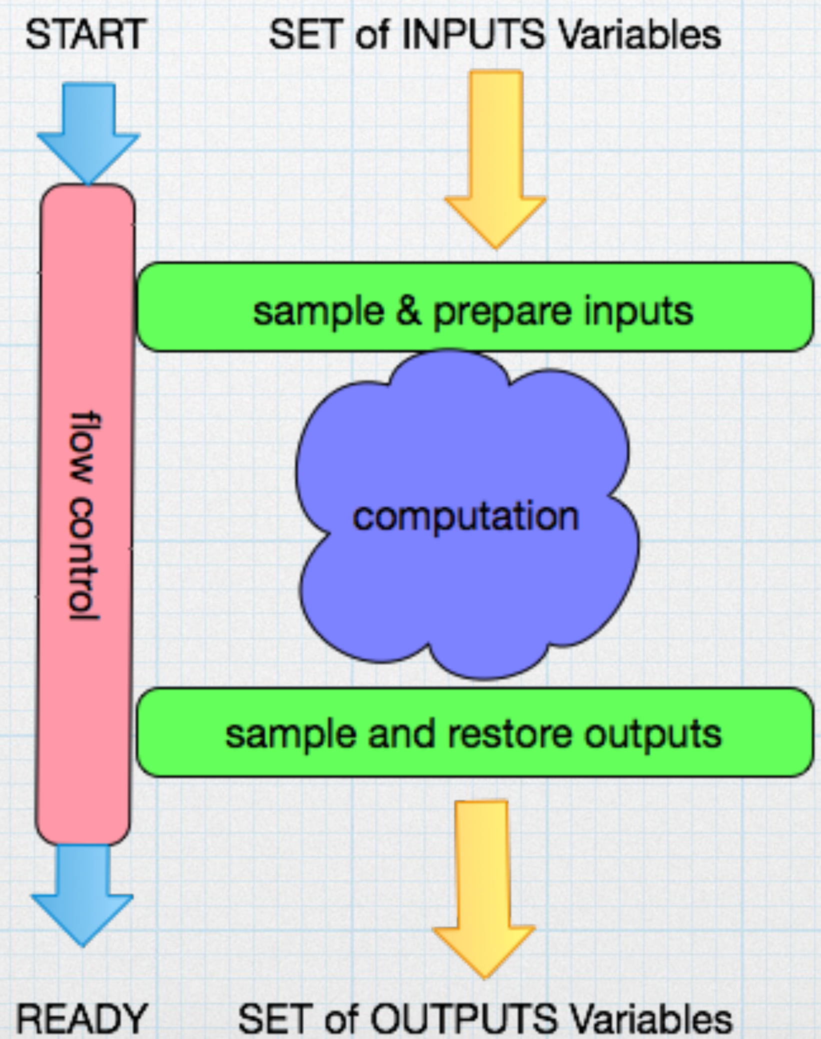

```
input bb, dd
output f3
aa = sqrt(bb)+sqrt(dd)
cc = sin(aa)-cos(aa)
dd = dist2(cc,1 0.4)
ff = dd<aa
f3 = ff ? (dd*aa) : (1/dd)
```

Float description is very simple and assumes all variables are either floating or booleans.
The translation from the original C++ to compiler input format is straightforward and in many cases just copy/paste.
The compiler guesses most of the definitions.
Python model is used to verify against C++ and same code is used during RTL verification.
C++/Matlab source code is too "noisy" to use directly. maybe in next release.

generated footprint

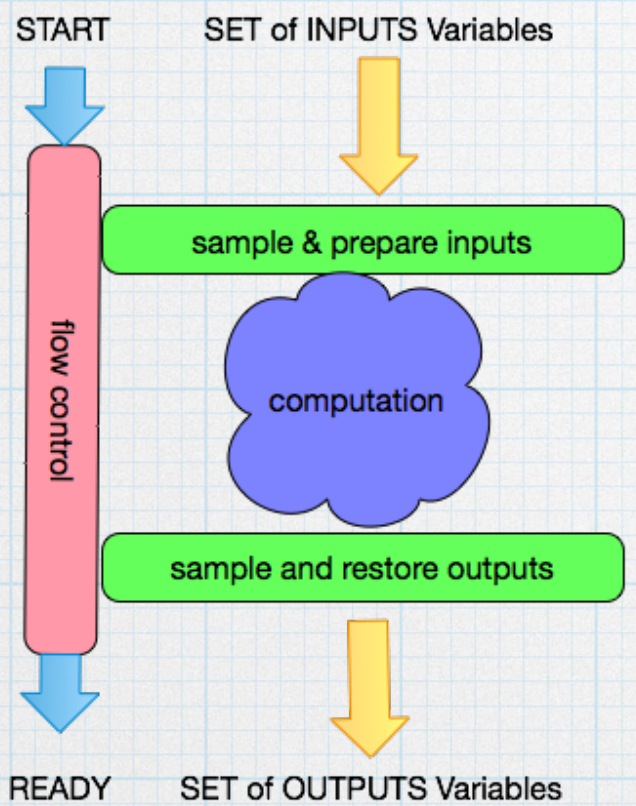
- set of input variables, booleans and constants
- outputs are of floating, fixed and boolean types.
- computation flow representation of the algorithm
- flow control signals: start, done
- set of numeric and boolean outputs
- is not intended for iterations, at least for now.
- notice the lack of variables ram, all vars are in regs.





example : solve $ax^2 + bx + c = 0$

suppose the task is to compute the roots of quadratic equation code

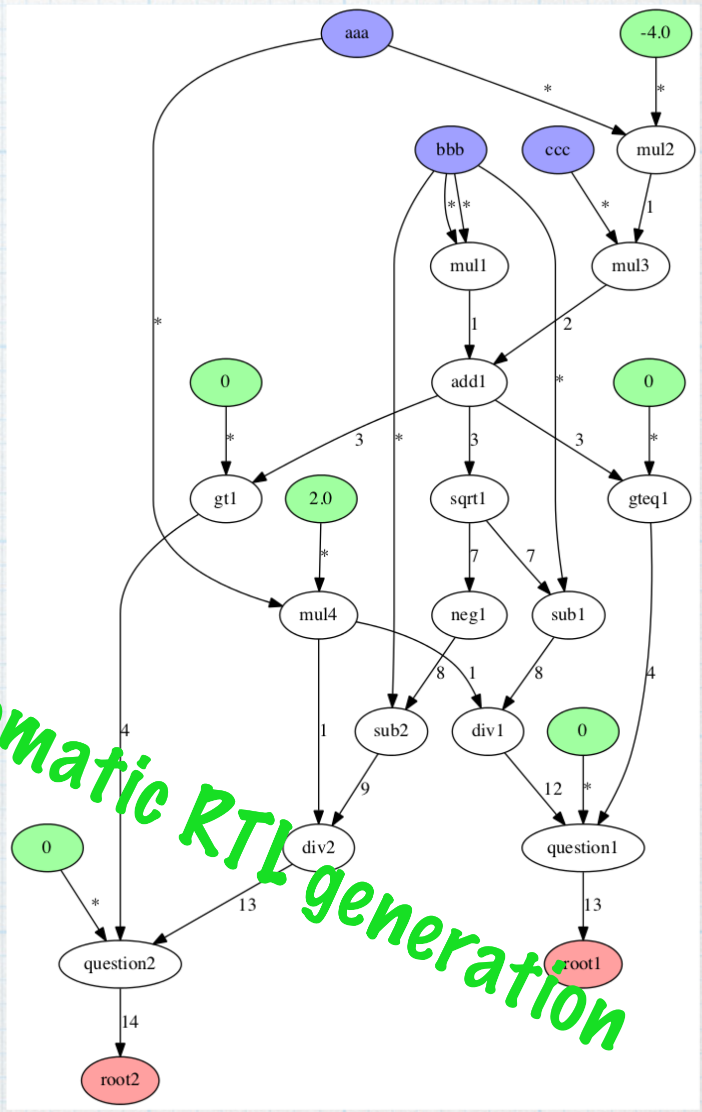


source code:

```
discriminant = bbb*bbb + (-4)*aaa*ccc
desq = sqrt(discriminant)

root1valid = discriminant>=0
root2valid = discriminant>0

mone1 = desq - bbb
mone2 = (-desq) - bbb

root1base = mone1/(2*aaa)
root2base = mone2/(2*aaa)

root1 = root1valid ? root1base : 0 ;
root2 = root2valid ? root2base : 0
```

this description lends itself to automatic RTL generation

operators flow:



computation:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

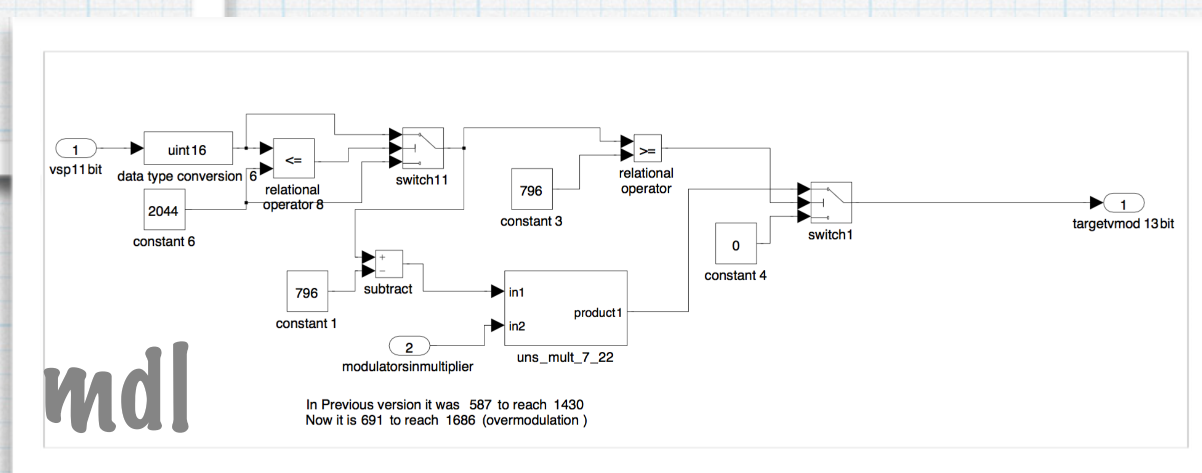
floating point proposition

* use single floating point format : one size fits-all. e.g. 1bit sign + 8bit exp + 23 (+1) bit mantissa
* assemble library of all numerical operators in a floating point.
* define a simple source code format.
* create compiler to translate source code to 3 optional rtl implementations and verification model.
* compiler 3 options for RTL : trade-off area / latency / throughput

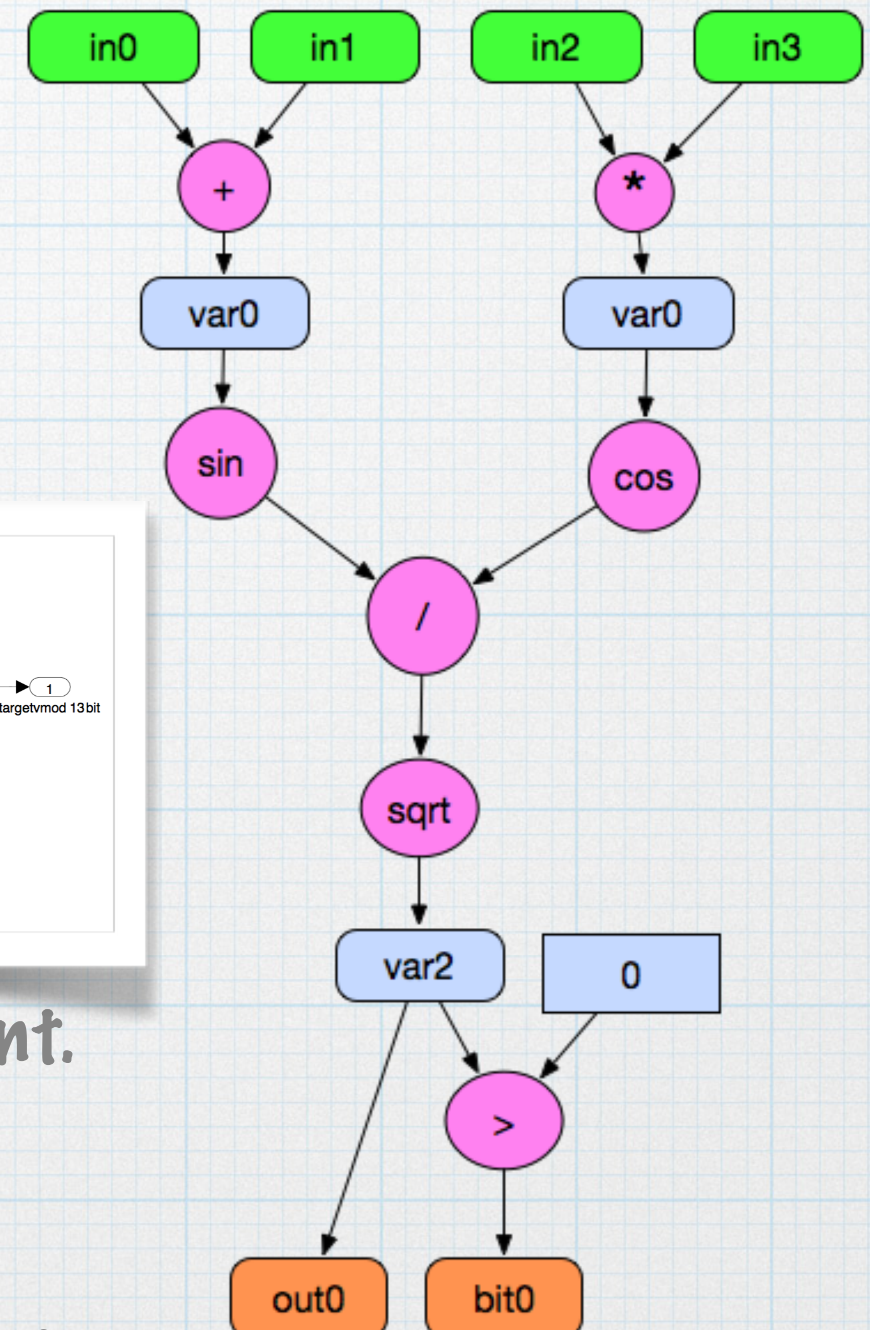
How the source looks like

text

aa = sin(in0+in1)
bb = cos(in2+in3)
out0 = sqrt(aa/bb)
bit0 = out0>0
vld0 = bb!=0



All operators and intermediate variables are floating point.
Several advantages over fixed point implementation:
no need to “fixate” the algorithm.
all variables are signed.
Known in advance are sizes of all operations and their cost in time, power and area.



3 compiler outputs

processor + code (rom or ram) + configuration (rom)	dataflow	full pipeline
all share identical operators		
all share identical final footprint		
run to completion	new result every several clocks	new result every clock
similar max operating frequency		
almost fixed area	area depends on algorithm	area depends on algorithm

operators zoo

dist max asin acos tan

min mul sqrt sin cos abs

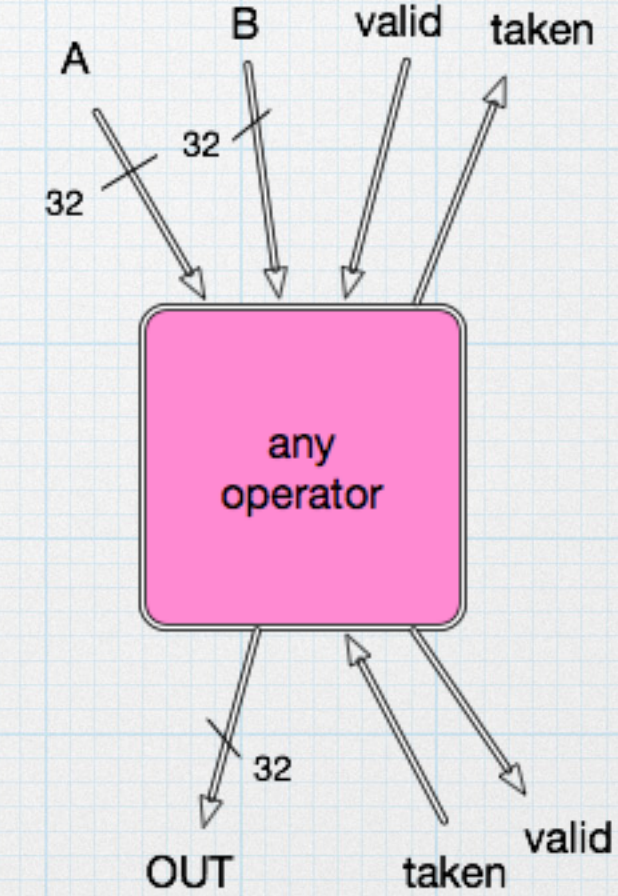
sat add div neg atan

select sub gt

aa = cc ? bb : dd;

trigo and sqrt functions
realized by cordic or table

dataflow packaging



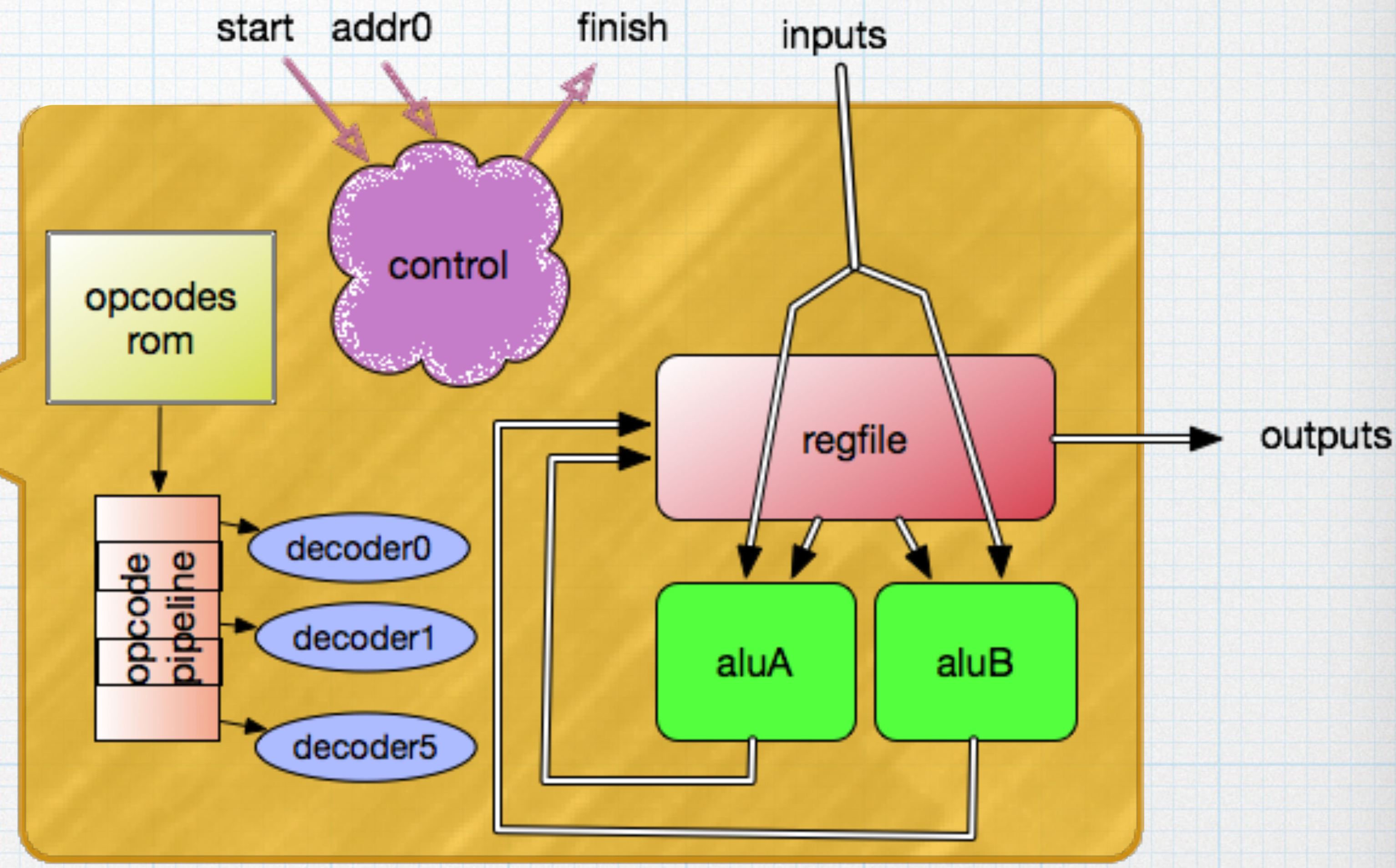
all these operators have closed rtl implementation and are available for all 3 options (in different packaging, standard for each option)

tradeoffs

* processor is best suited for long computations that are not throughput sensitive. Or set of shorter different subroutines. The area is constant and known (almost).
* dataflow works for applications where result may be produced every several clocks. The area can be accurately predicted.
* full-pipelined, larger area. also can be predicted.
* selection of table-driven or Cordic-driven functions depends on accuracy requirements.

FloatProc

configuration constants to reduce synthesis



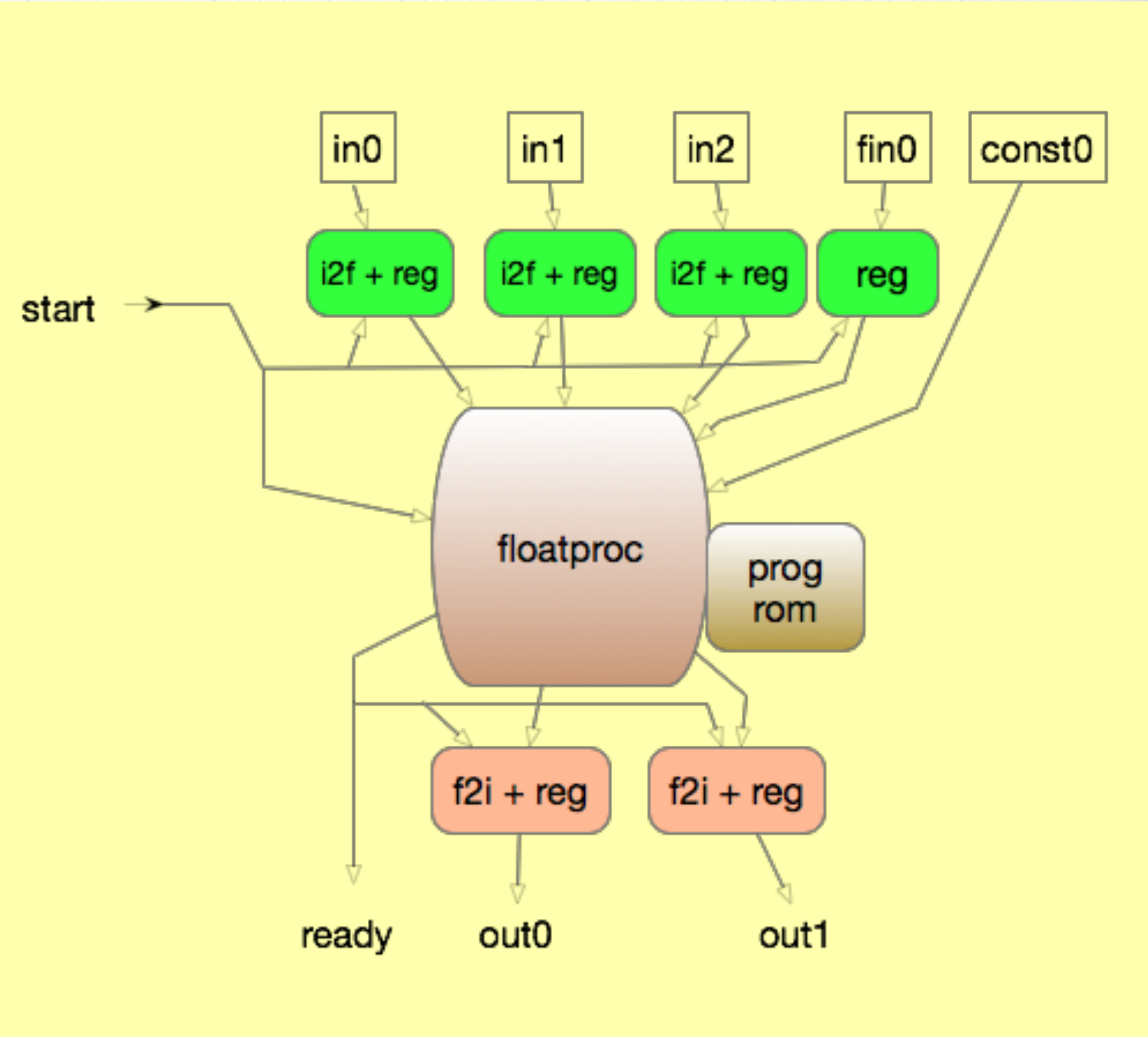
- generated tailored instruction set.
- dual issue, each ALU has all functions.
- black box setup (*)
- up to 32 input values
- up to 32 float registers, 32 logical
- all data is floating or boolean.
- stalls on register dependencies
- several hooks exist, but not used in this release: load/store ram, jumps, conditionals, multi-start vectors.

internal float format

sign	exp	mant
1	8	23

F container

for each instance of floatProc, the compiler creates a program rom and custom made container. This container samples input values, converts to floating point (and back), samples floating format inputs, drives constant values and so on. The compiler also creates configuration mask to mask off irrelevant resources. Includes backpressure controls on inputs and outputs (not shown on drawing).

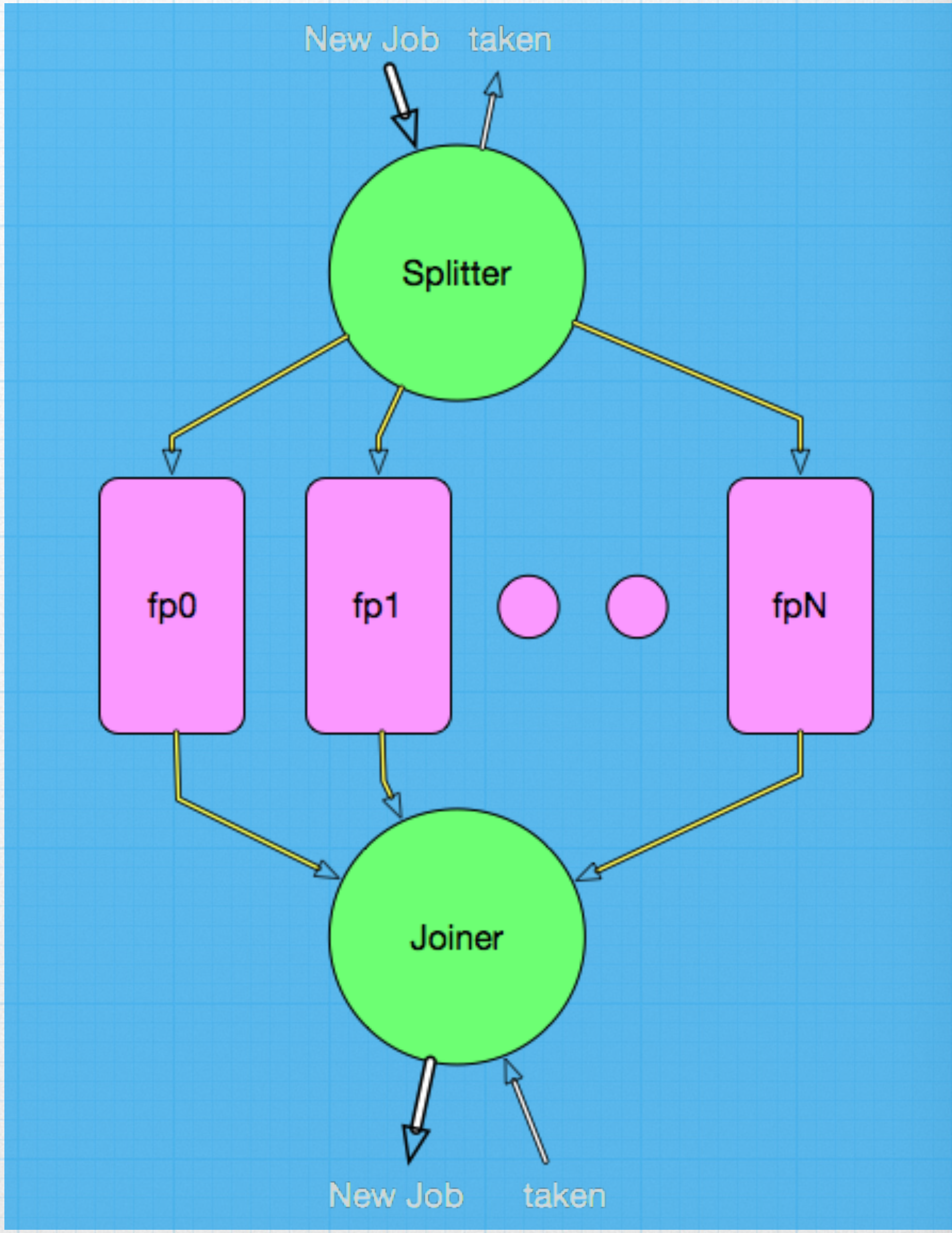


quadratic asm listing

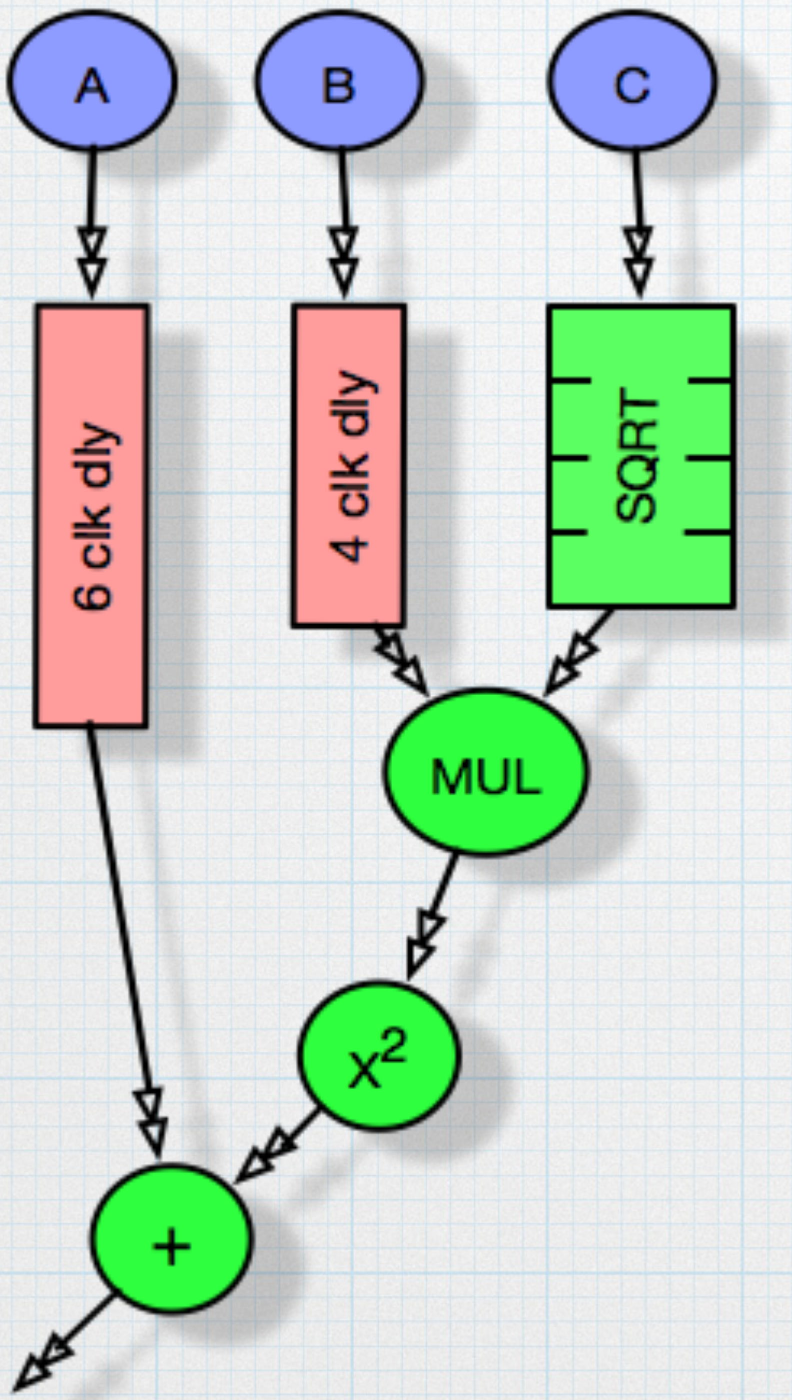
```
0000 b46e201b b47bfffe    0    net1 = R0 - 4 ; net0 = bbb * bbb ;
0001 b47bf37b b45db7ff    0    net2 = net1 * aaa ; nop ;
0002 b47beb7d b47217db    0    net3 = net2 * ccc ; net11 = 2 * aaa ;
0003 b478ebdd b45db7ff    0    discriminant = net0 + net3 ; nop ;
0004 b45dcbbc b50603be    0    desq = sqrt discriminant ; root1valid = discriminant >= 0 ;
0005 b476e01f b477fb9c    0    net9 = R0 - desq ; mone1 = desq - bbb ;
0006 b477fbfd b50203bf    0    mone2 = net9 - bbb ; root2valid = discriminant > 0 ;
0007 b47edbbd b47edb9c    0    root2base = mone2 / net11 ; root1base = mone1 / net11 ;
0008 b63e03bc b63c039d    0    root2 = root2valid ? root2base : 0 ; root1 = root1valid ? root1base : 0 ;
0009 b45db7fe b45db7ff    0    finish ; nop ;
```

The pool

The flow/algorithm burned into one Float processor can take many cycles to finish. In case performance of a single processor is not enough, the compiler accepts a parameter which creates a pool of containers.
This hardware has splitter to select the idle processor, Send the job to it. And a joiner that collects the finished results.
Each FP processing time is constant. So the order of jobs is not changed.
The interface of pool and container looks identical. So there is no rework needed when replacing one with the other.

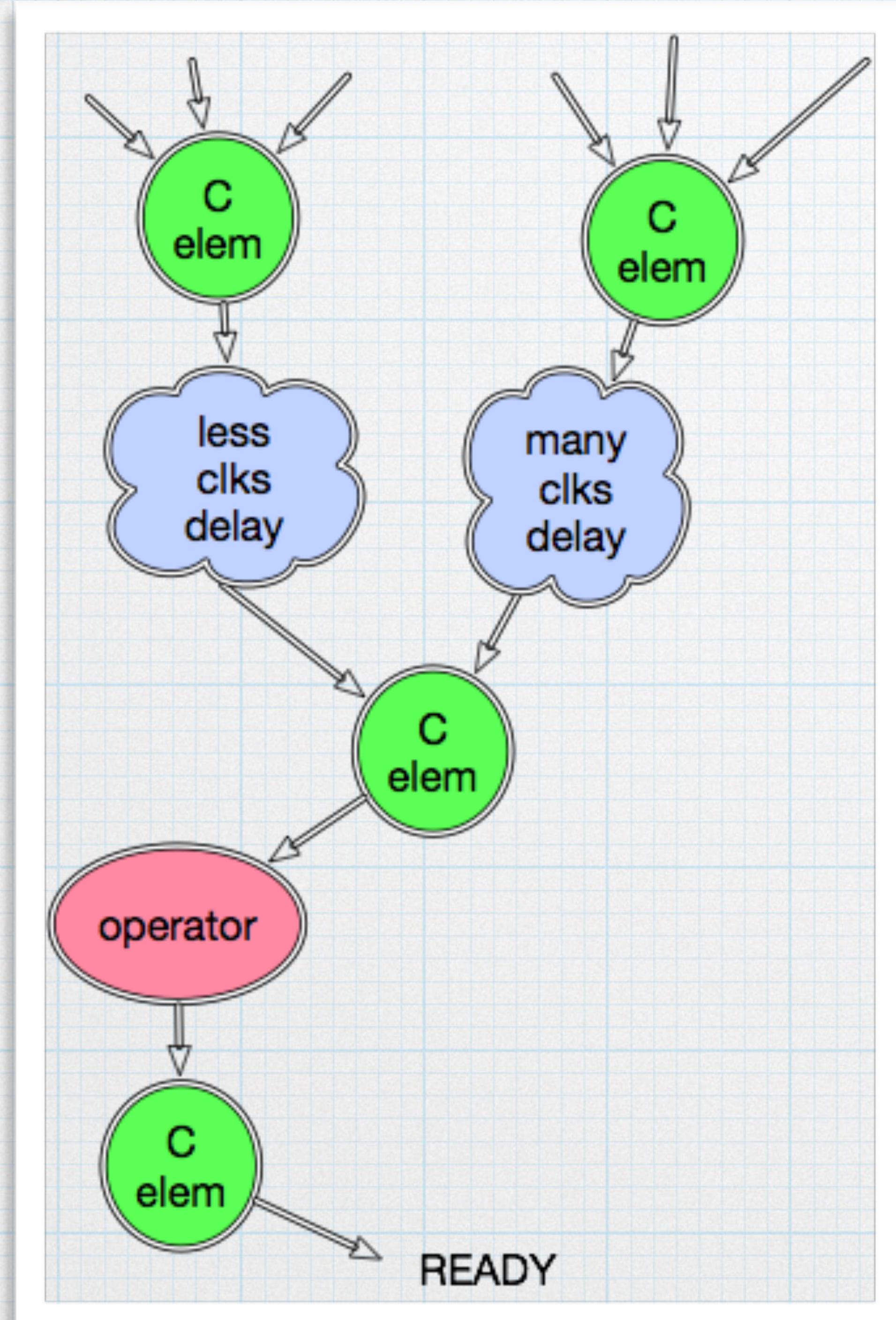


full pipeline



"A", "B" and "C" arrive on the same clock, "sqrt" operator takes 4 clocks, "mul" one clock.
we insert 4 delays on "B" , so that multiplier gets correct data.
All multi clock operators (like cordics,div) have fully pipelined version.
Therefore new values of inputs ("A","B","C") can be driven on every clock and outputs available on every clock (after propagation number of clocks, 7 in this example).
Compiler inserts the correct number of delays wherever is needed to balance the path delays.
Compiler also produces container for this design.

DataFlow



operators are the same operators, the sequencing logic is join/fork elements, similar in idea to async logic "C" elements, only sync version.
DataFlow requires no pipeline stages, but the throughput limited by max delay discrepancy.
If this is a problem, compiler directive allows to insert pipelines in strategic nodes, and thus increase the throughput.
Container to condition inputs/outputs is generated as well.

The advantage of this approach is delay-in-clocks insensitivity and area. "C" elements are minor resource gobblers.

Summary

* synthesis results are showing promise.
* processor or data flow present enough leeway in implementation.
* floating point elements are neither scary nor problematic.
* 謝謝