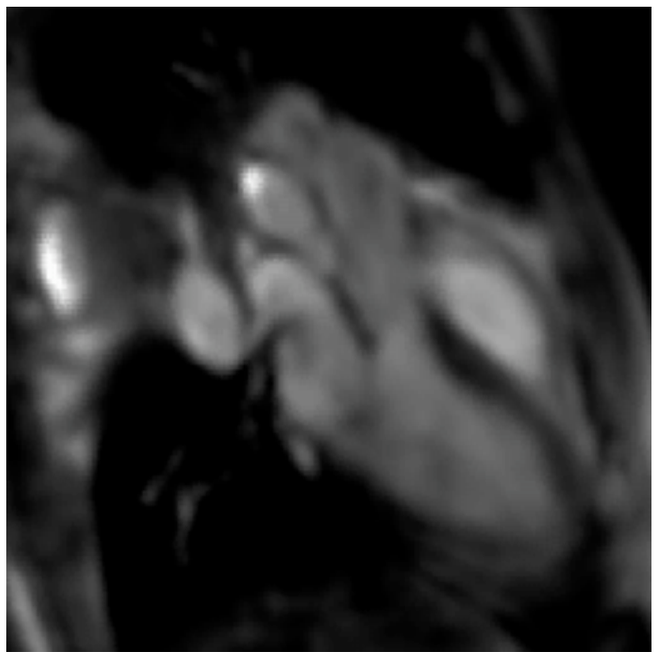
Coconut:

Code ***Con***structing ***U***ser ***T***ool

Christopher Kumar Anand
Wolfram Kahl

http://ocalgorithms.com

We can write safe software.



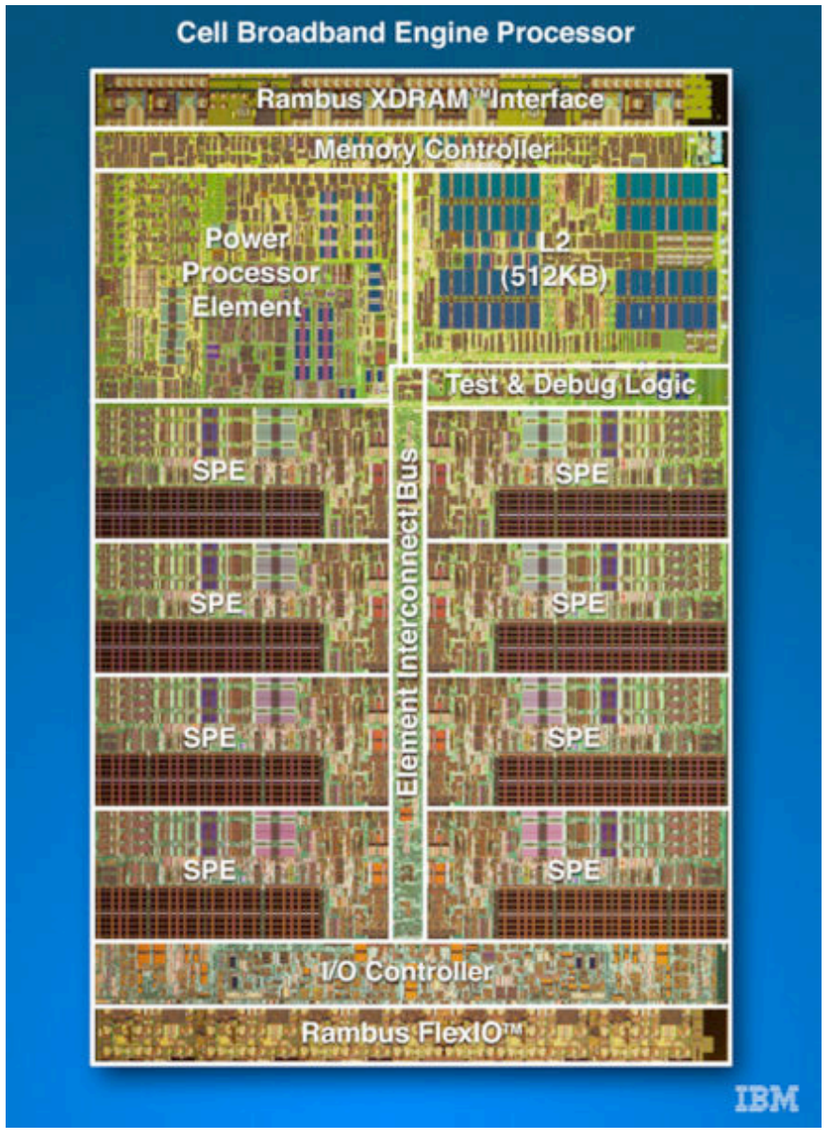
Sometimes
we need
both.

We can write fast software.

Performance = Parallelism

Cell BE

- **384**-way ||ism
  - 4-way SIMD
  - 8-way cores
  - 6-times unrolling
  - double buffering



Roadmap

- SIMD Parallelism

✔ extensible DSL captures patterns

½ verification via graph transformation

✔ generated library shipping (Cell BE SDK 3.0)

- Multi-Core Parallelism

✔ model on ILP

⇛ generation via graph transformation

✔ linear-time verification

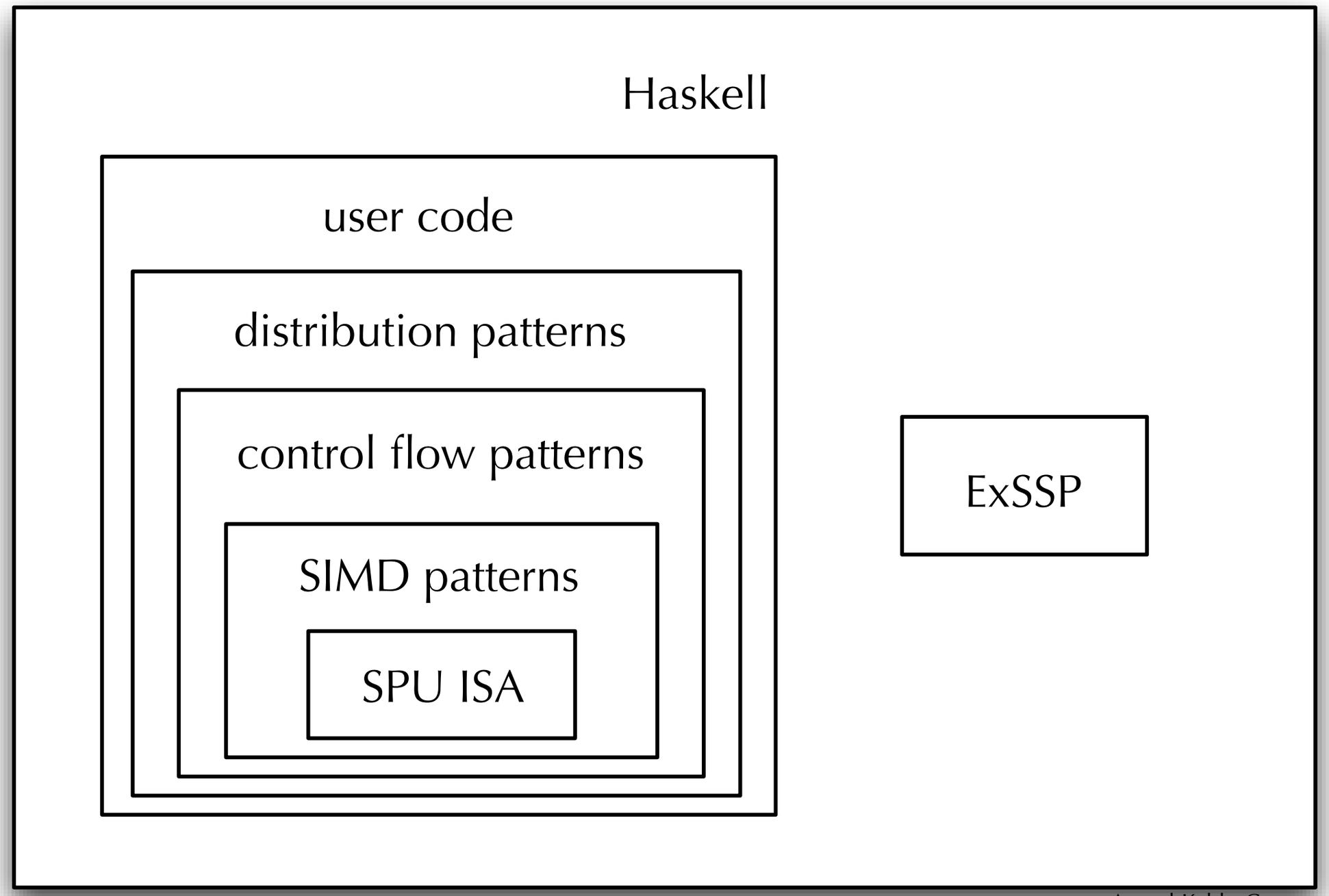
⇛ run time

- Distant Parallelism

∞ verification via model checking

✔ Scheduling: ExSSP

Layers of Domain Specific Languages

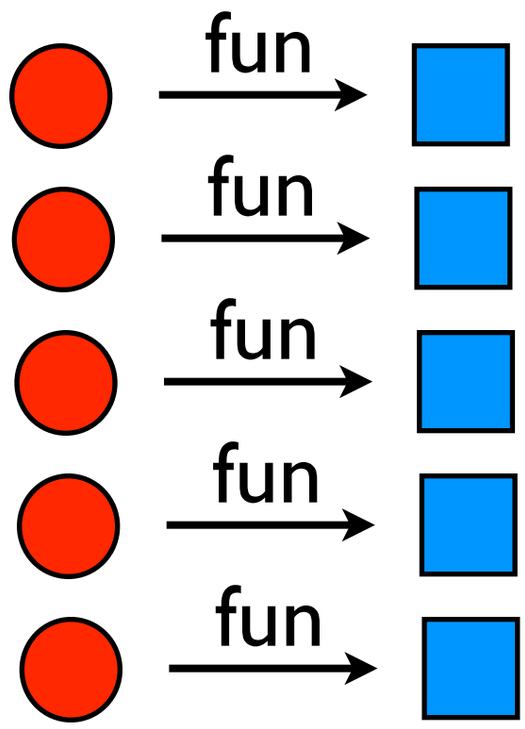


Higher Order Functions

- examples
  - map
  - zip
- matrix multiplcation
- SIMD parallelization
- multi-core parallelization

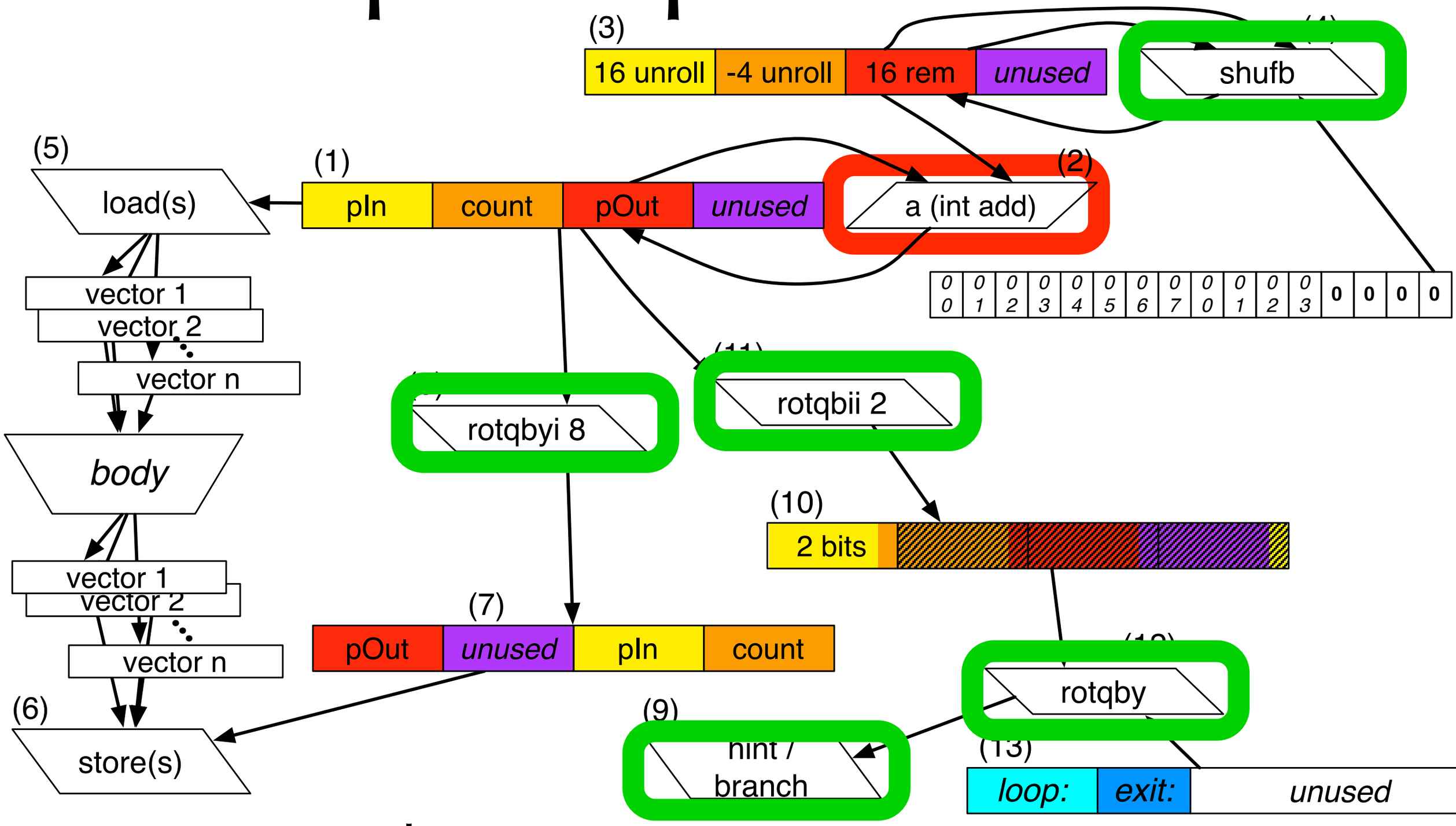
map

```
for (i=0; i<10; i++) {
 out[i] = fun(in[i]);
}
```



- apply a function to a list
- overhead
  - increment pointer
  - increment pointer
  - increment counter
  - compare counter
  - branch

Map Loop Overhead

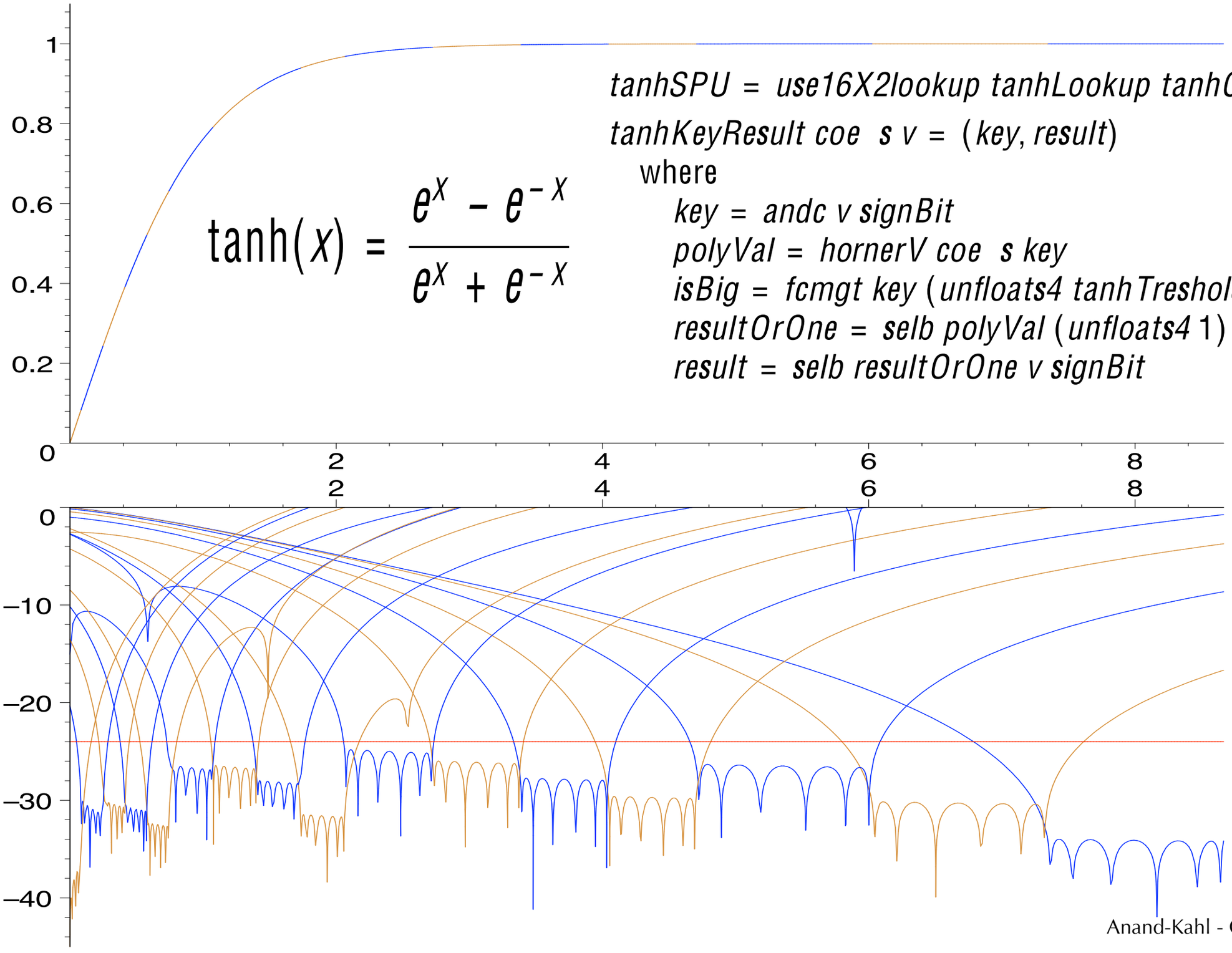


- one arithmetic instruction
- in/out pointers + induction variable + hint

Low Level DSL

- declarative assembly
- support functions
  - polynomial approximation
  - table lookup in registers
- verify assertions @ compile time
- compile time computation
- user extensible

Compact Code



$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

```
tanhSPU = use16X2lookup tanhLookup tanhC tanhKeyResult
tanhKeyResult coe  s v = (key, result)
  where
    key = andc v signBit
    polyVal = hornerV coe  s key
    isBig = fcmgt key (unfloats4 tanhTreshold)
    resultOrOne = selb polyVal (unfloats4 1) isBig
    result = selb resultOrOne v signBit
```

6. Cube Root

The rest of this section is an unedited example of literate source code.

Cube Root is defined to be the unique real cube root with the same sign as the input. We calculate it using

$$(-1)^{sign}\ 2^{e}\ (1 + frac) \qquad (-1)^{sign}\ 2^{q}\ 2^{r/3}\ f\,(1 + frac) \tag{3}$$

where q and r are integers such that

$$e = 3\quad q + r, \qquad 0 \leq r < 3, \tag{4}$$

and f (x) is a piecewise order-three polynomial minimax approximation of $(x)^{1/3}$ on the interval [1, 2).

Warning: This function uses divShiftMA for fixed-point division. This is computation is inexact, but cbrtAssert tests all the values which can occur as a result of extracting the exponent bits for the input float. If you modify the code you must modify the assertion.

```
cbrtSPU :: forall v   (SPUType v, HasJoin v)     v     v
cbrtSPU v = assert cbrtAssert "cbrtSPU" result
  where
```

Since we process the input in components, we cannot rely on hardware to round denormals to zero, and must detect it ourselves by comparing the biased exponent with zero:

```
denormal = ceqi exponent 0
```

and returning zero in that case

```
result = selb unsigned (unwrds4 0) denormal
```

We calculate the exponent and polynomial parts separately, and combine them using floating-point multiplication,

```
unsigned = fm signCbrtExp evalPoly
```

Insert the exponent divided by three into the sign and mantissa of the cube root of the remainder of the exponent division.

```
signCbrtExp = selb signMant
                    (join $ map (  f     f expDiv3shift16 7)
                                [shli, rotqbii])
                    (unwrds4 $ 2   31 - 2   23)
```

Use the function extractExp to extract the exponent bits, dropping the sign bit, and placing the result into the third byte:

```
exponent = extractExp 3 v
```

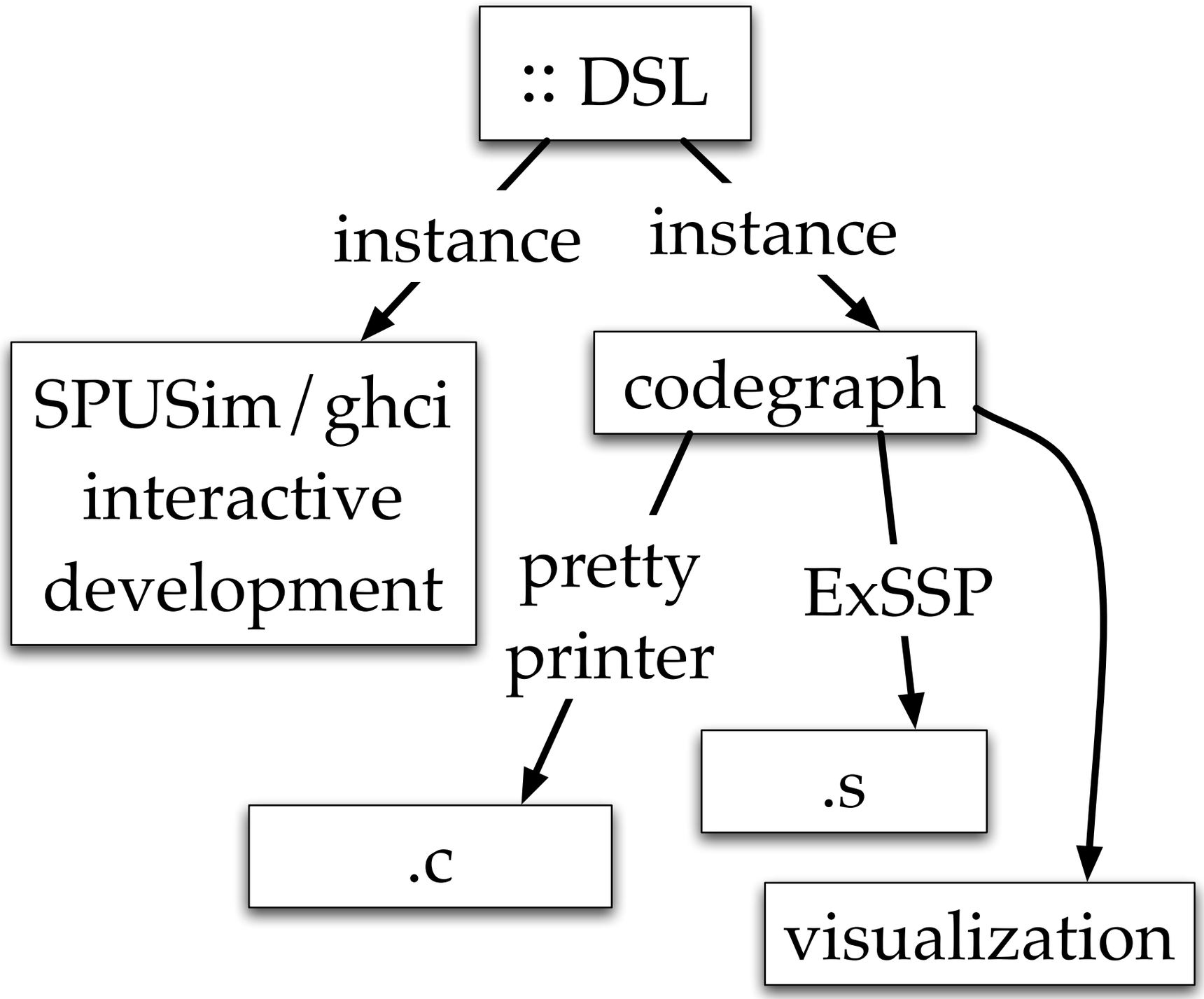

- Literate Haskell
- *code* inside LaTeX
- **machine ops**
- patterns

```
coe  s = lookup8Word (22, 20) expCoe  s24bits v
```

Evaluate the polynomial on the fractional part.

```
evalPoly = hornerV coe  s frac
```

Multiple Instances



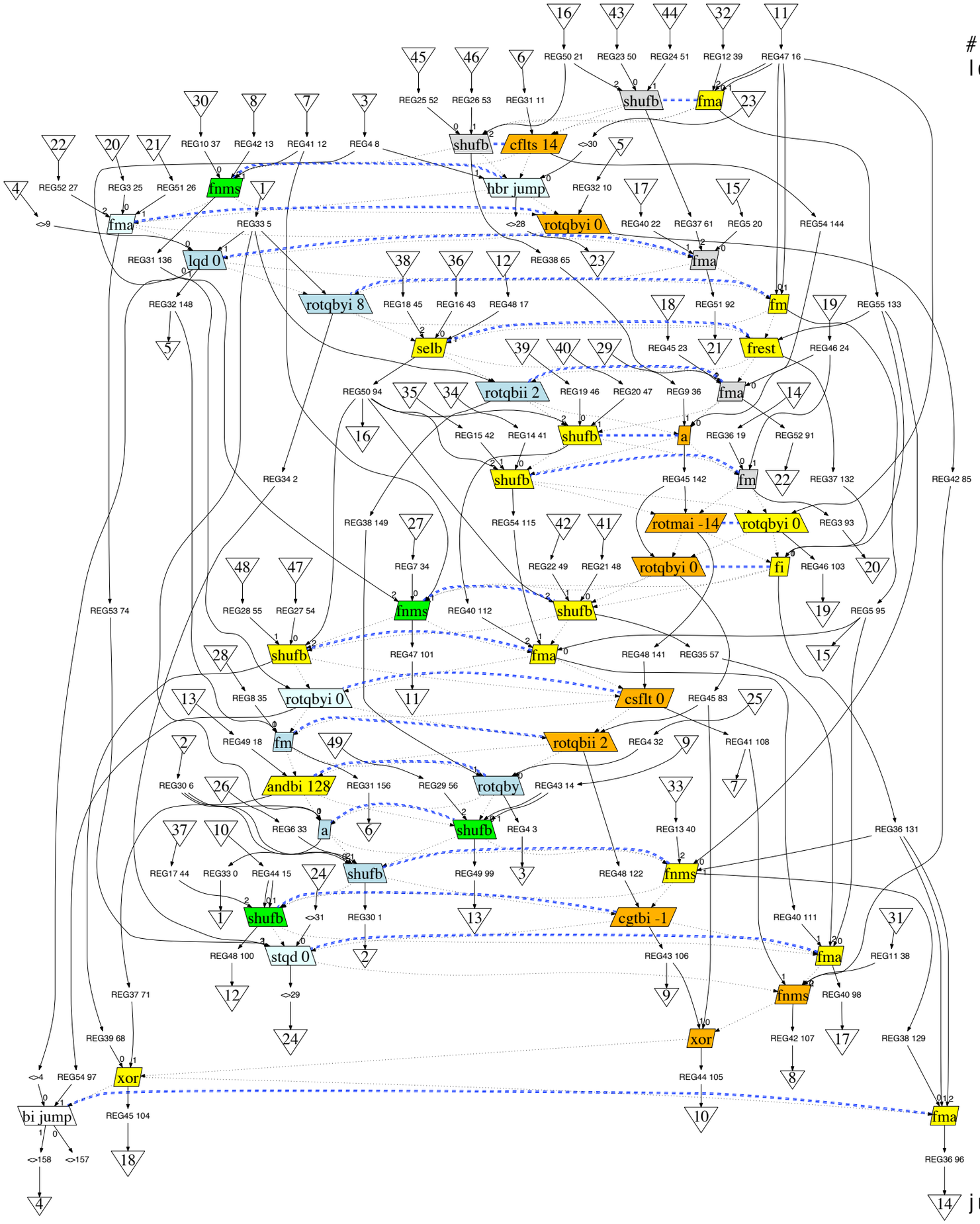


Figure 5. Scheduled assembly code graph for tanSPU.

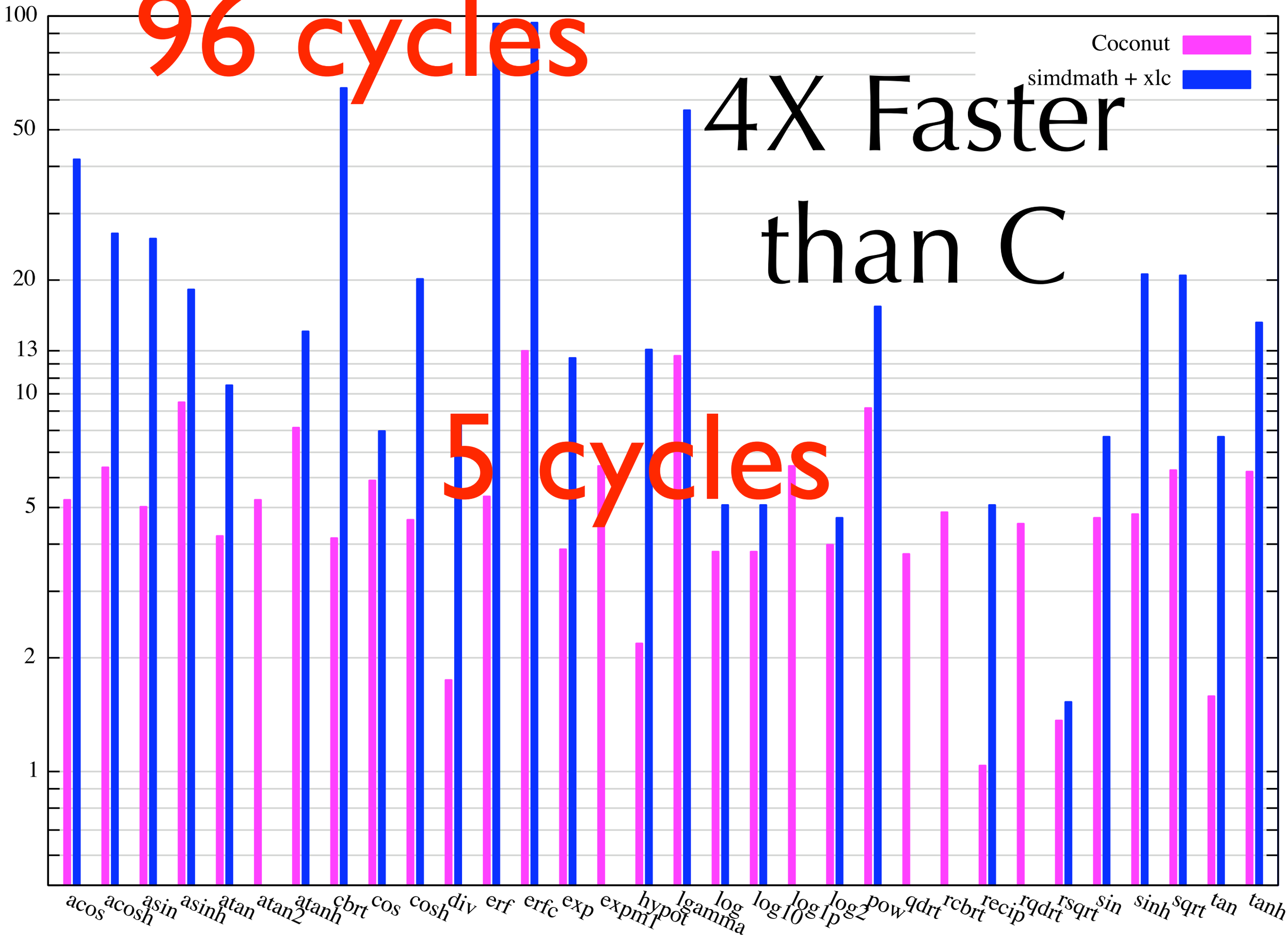
```
# 25 cycles
loop:  fma      $55, $47, $47, $12
       shufb    $37, $23, $24, $50
       cflts    $54, $31, 14
       shufb    $38, $25, $26, $50
       fnms     $31, $10, $41, $42
       hbr      jump, $4
       fma      $53, $3, $51, $52
       rotqbyi  $42, $32, 0
       fma      $51, $5, $40, $37
       lqd      $32, 0($33)
       fm       $5, $47, $47
       rotqbyi  $34, $33, 8
       selb     $50, $16, $48, $18
       frest    $37, $55
       fma      $52, $46, $38, $45
       rotqbii  $38, $33, 2
       a        $45, $54, $9
       shufb    $40, $19, $20, $50
       fm       $3, $36, $46
       shufb    $54, $14, $15, $50
       rotmai   $48, $45, -14
       rotqbyi  $46, $47, 0
       fi       $36, $55, $37
       rotqbyi  $45, $45, 0
       fnms     $47, $7, $41, $31
       shufb    $35, $21, $22, $50
       fma      $40, $5, $54, $40
       shufb    $39, $27, $28, $50
       csflt    $41, $48, 0
       rotqbyi  $54, $4, 0
       fm       $31, $32, $8
       rotqbii  $48, $45, 2
       andbi    $37, $49, 128
       rotqby   $4, $4, $38
       a        $33, $33, $30
       shufb    $49, $43, $43, $29
       fnms     $38, $55, $36, $13
       shufb    $30, $30, $30, $6
       cgtbi    $43, $48, -1
       shufb    $48, $44, $44, $17
       fma      $40, $5, $40, $35
       stqd     $53, 0($34)
       fnms     $42, $11, $41, $42
       xor      $44, $45, $43
       xor      $45, $39, $37
       lnop
       fma      $36, $38, $36, $36
jump:  bi       $54
```

Figure 6. tanSPU.s

96 cycles

4X Faster than C

5 cycles



Fine Print on Comparison

- pink bars = C-callable vector SPU MASS
  - e.g., vsexp (in C-ABI library)
  - generated/scheduled by Coconut
  - distributed in SDK 3.0 and with xlc
    http://www-306.ibm.com/software/awdtools/mass
  - single vector version slightly slower
    - distributed as (cryptic) C
    - e.g. expf4
- blue bars = SimdMath (circa SDK 3.0)
  - developed and distributed in readable C
  - scheduled by spuxlc

Ultimate Assembler

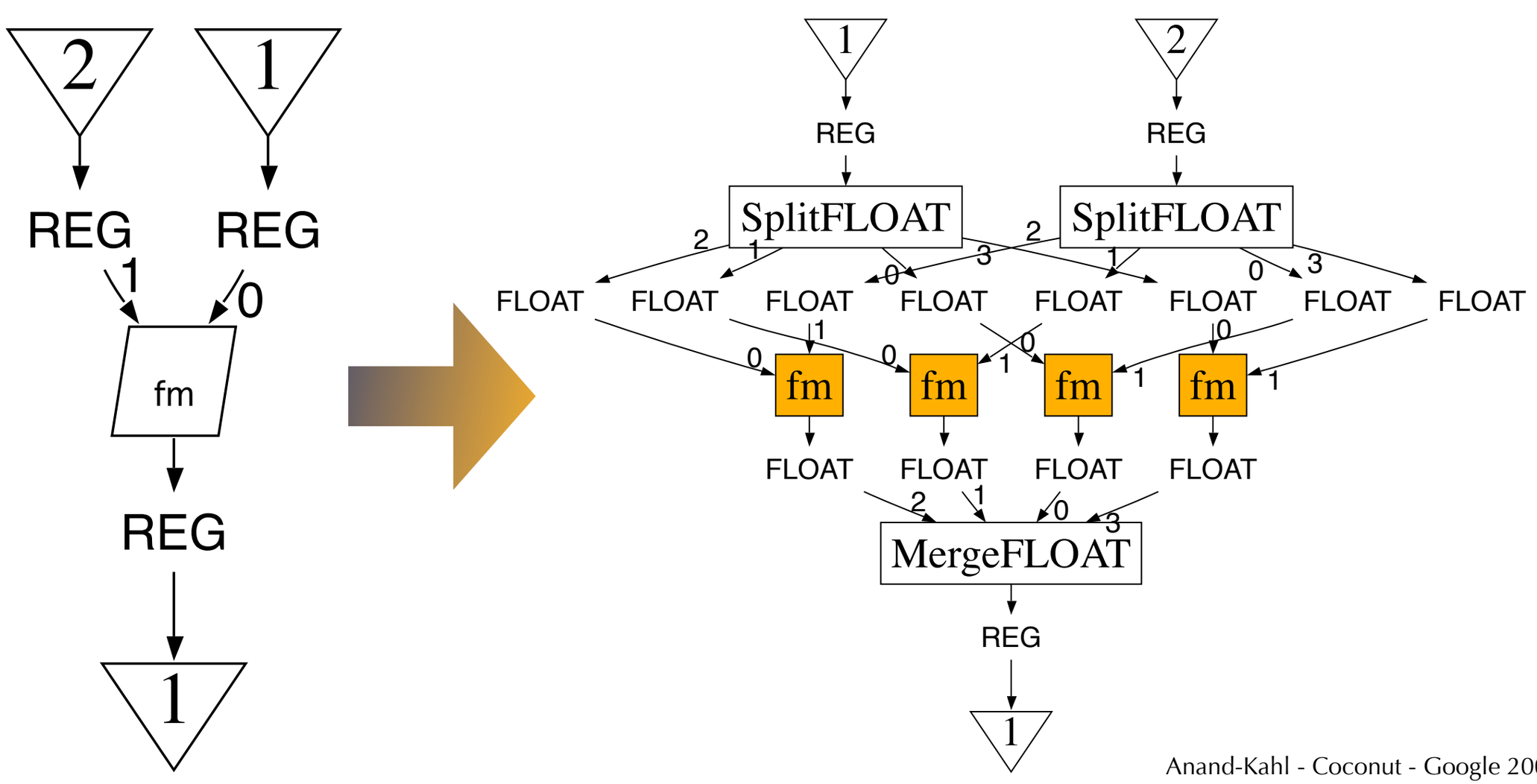
- access to machine instructions
- write patterns in Haskell
- unit test declarative assembly code
- where does performance come from?

SCIMD =

Single ***Complex*** Instruction ***Merged*** Data

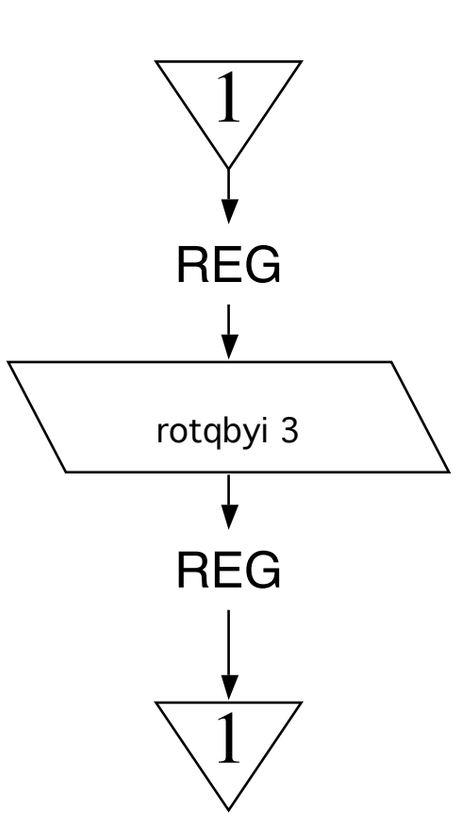
Verification

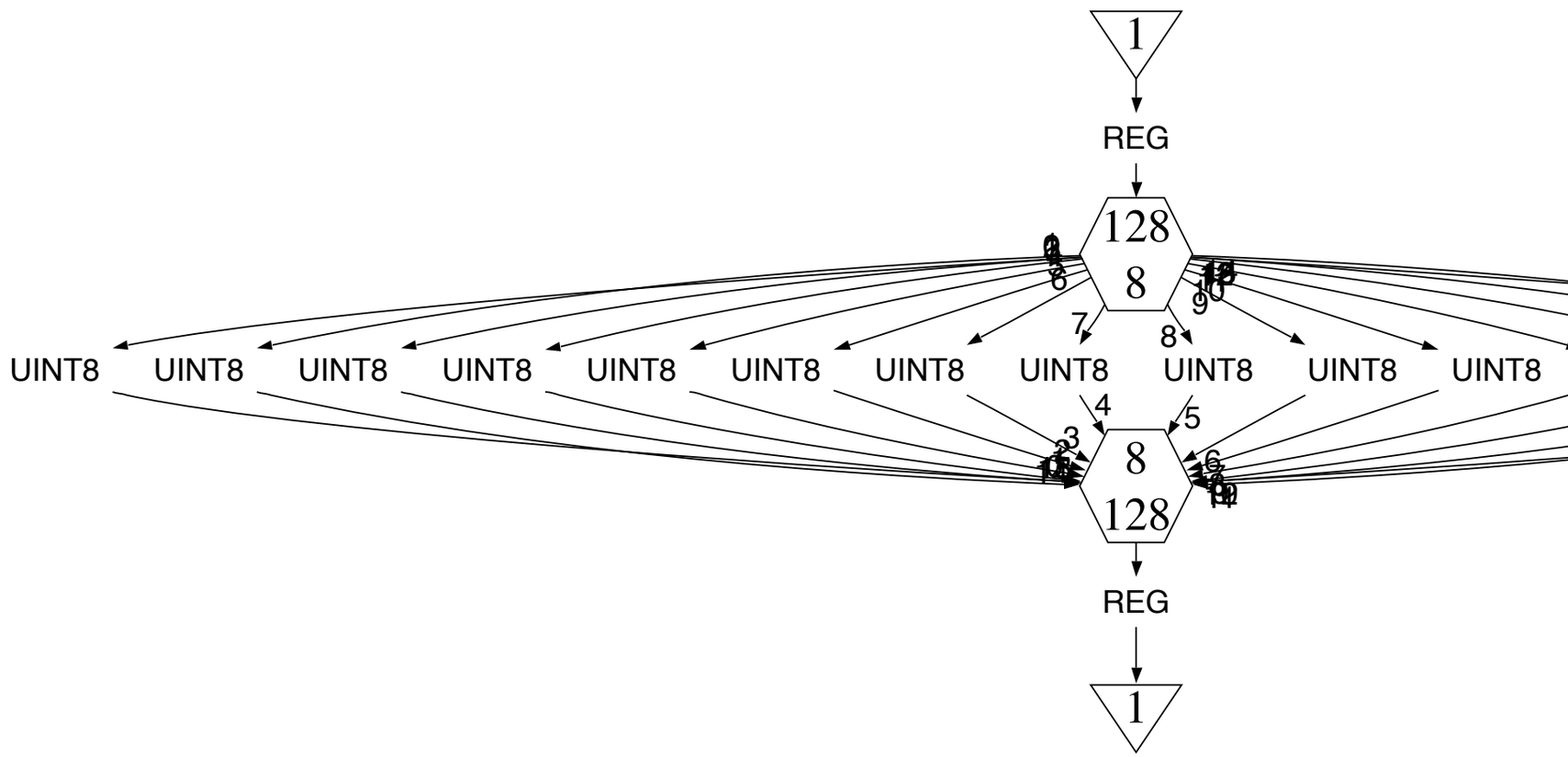
- transform graphs
  - break 128-bit register values up
  - easy for “pure” SIMD



Difficult for Creative Bit Shuffling

- easiest case: byte rotate by constant





- hardest case: rotate bits by register value

Status - SIMD

- code generation
  - rapid prototyping
  - peak performance
    - lots of work supporting other patterns
    - e.g. interpretting bit operations on floats
- verification
  - equivalent to symbolic execution
    - useful for debugging linear algebra
  - needs more transformation rules

Multi-Core = ILP Take 2

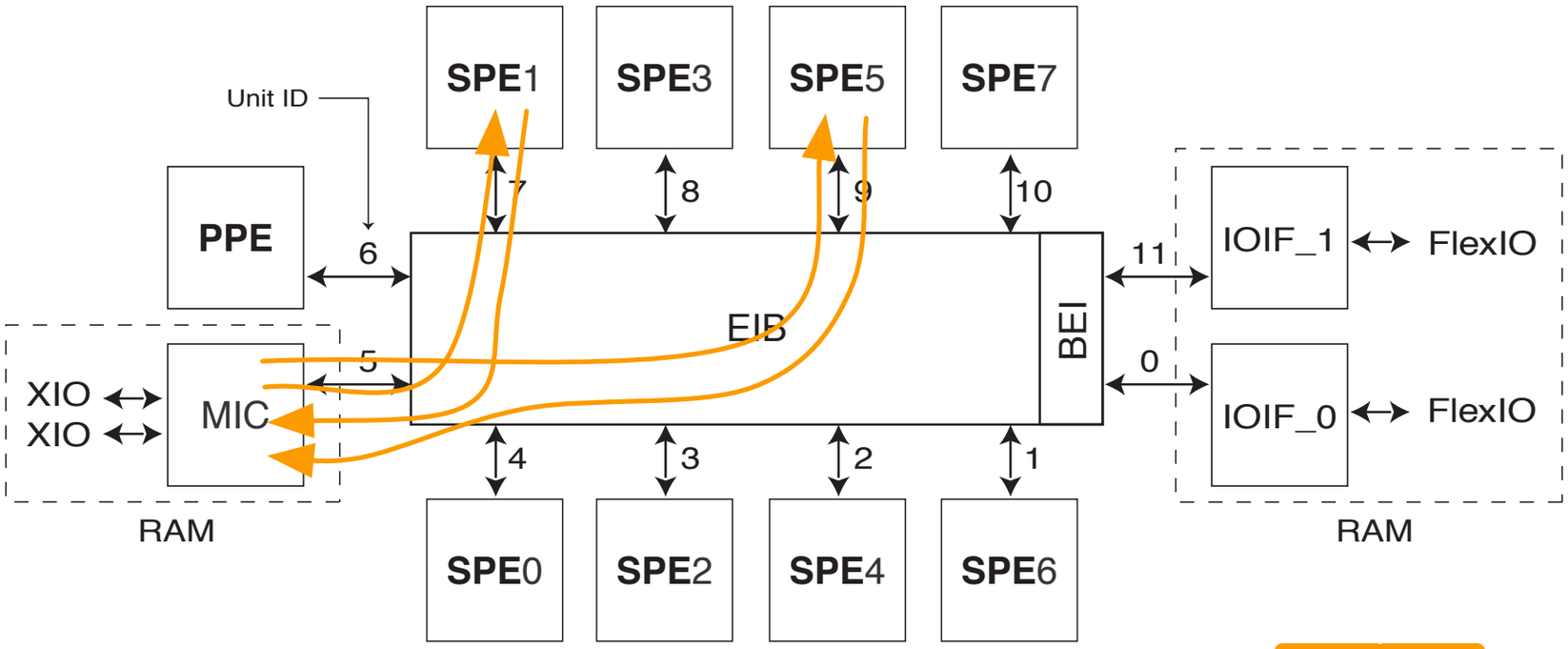
Instruction Level Parallelism	Multi-Core Parallelism
CPU	Chip
Execution Unit	Core
Register	Buffer / Signal
Load/Store Instruction	DMA
Arithmetic Instruction	Computational Kernel

The Catch: Soundness

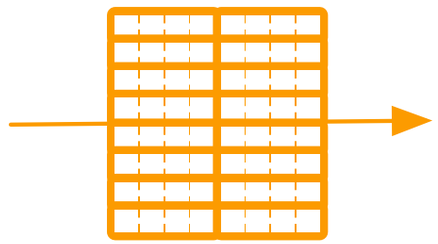
- on CPUs hardware maintains OOE
  - instructions execute out of order
  - hardware hides this from software
    - ensures order independence
- in our Multi-Core *virtual* CPU
  - compiler inserts synchronization
    - soundness up to software
    - uses asynchronous communication

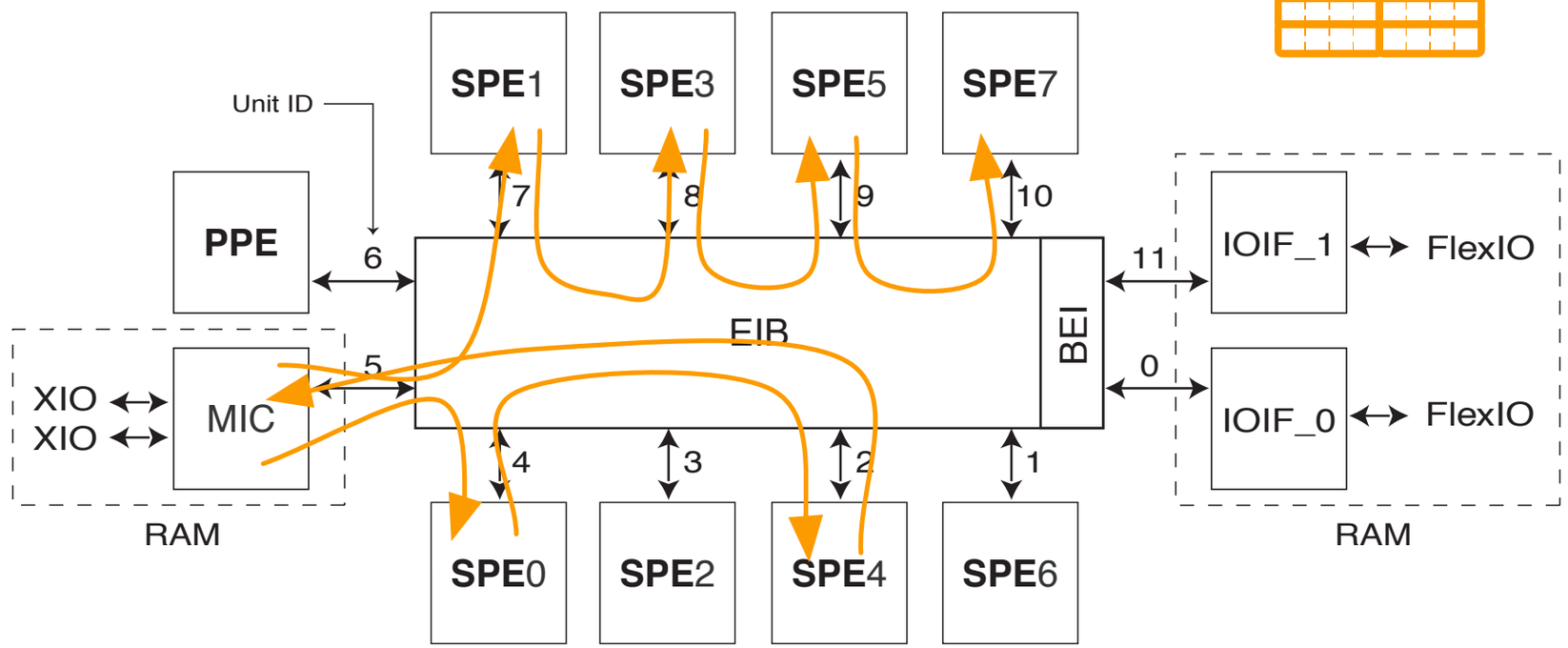
Asynchronous

- no locks
  - locking is a multi-way operation
    - a lock is only local to one core
      - incurs long, unpredictable delays
- use asynchronous messages
  - matches efficient hardware

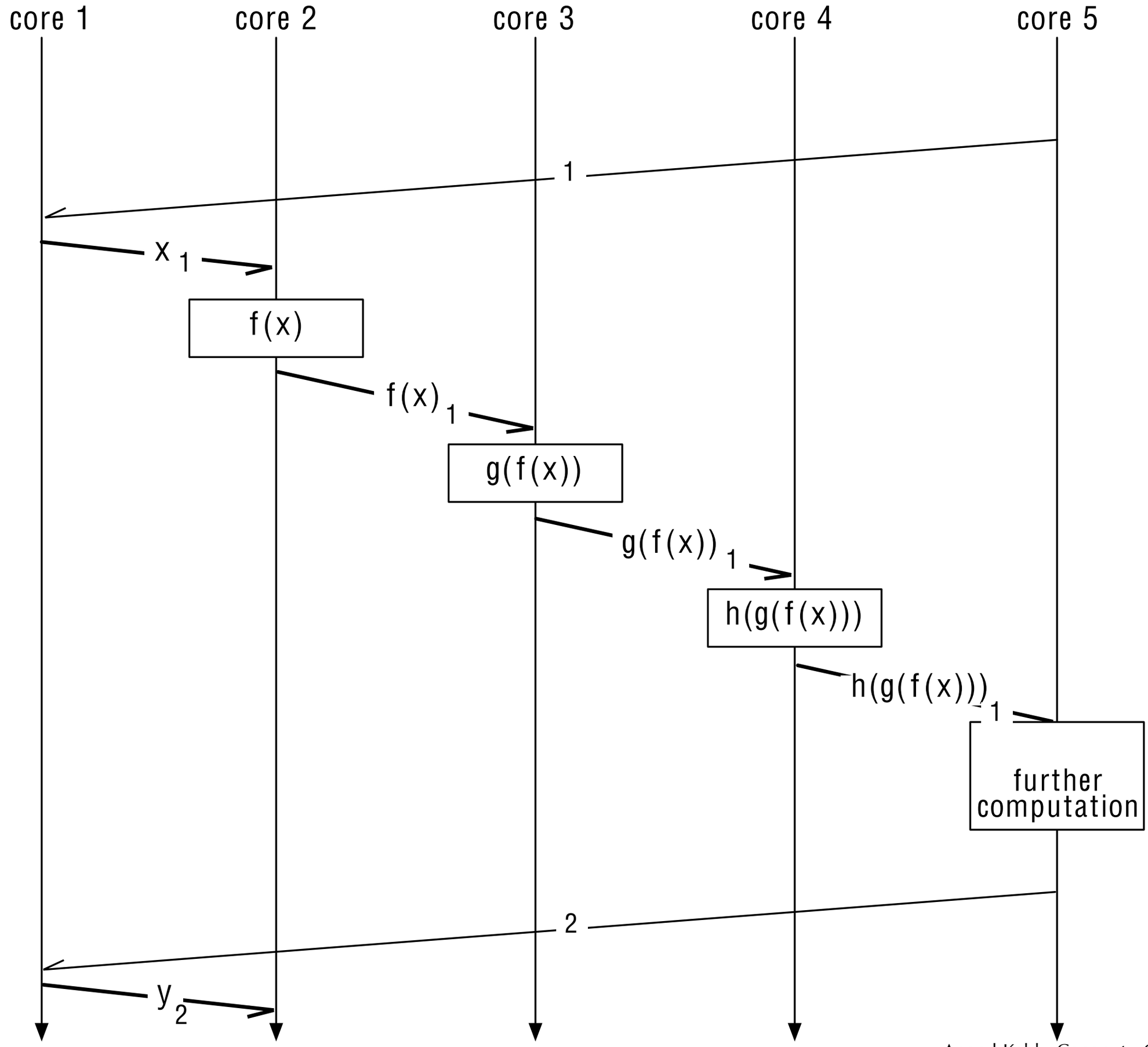


Memory Bound

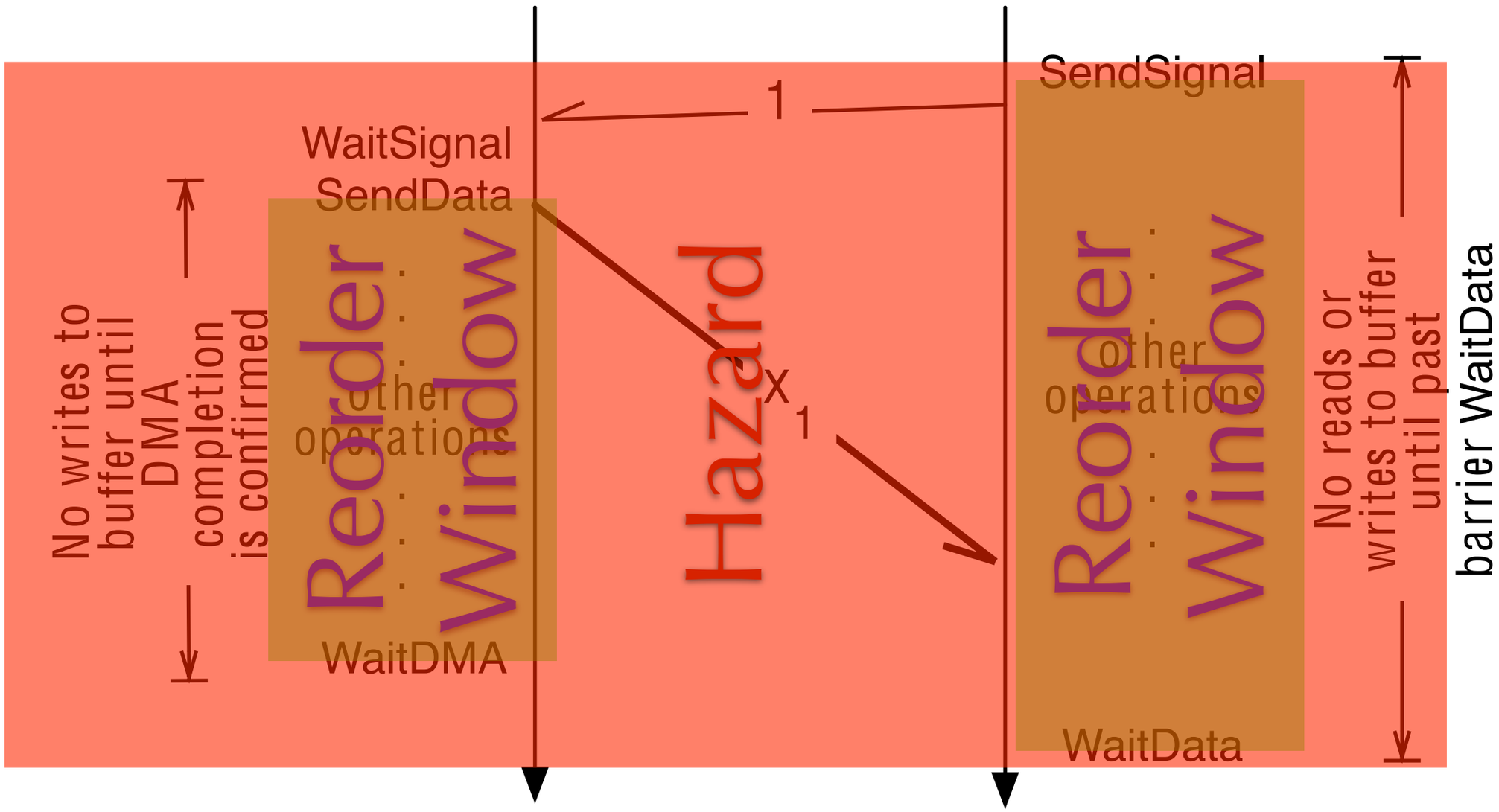




Comp Bound


core 1
core 2
core 3
core 4
core 5
1
x 1
f(x)
f(x) 1
g(f(x))
g(f(x)) 1
h(g(f(x)))
h(g(f(x))) 1
further computation
2
y 2

Async Signals



Multi-Core Language

Computation *operation bufferList*	do a computation with local data
SendData *localBuffer remoteBuffer tags*	start DMA to send local data off core
WaitData *localBuffer tag*	wait for arrival of DMAed data
WaitDMA *tag*	wait for locally controlled DMA to complete
LoadMem *localBuffer remoteBuffer tag*	start distant data load
SendSignal *core signal*	send a signal to distant core
WaitSignal signal	wait for signal to arrive

Concurrent Control-Flow

1. Scheduling

- hide latency to eliminate stalls

2. WaitSignal / WaitData

- stall when necessary, hardware won't
- timing less predictable

locally Sequential Program

index	core 1	core 2	core 3
1		long computation	
2	SendSignal *s* *c*2		
3		WaitSignal *s*	
4		computation	
5			SendSignal *s* *c*2
6		WaitSignal *s*	

- total order for instructions
  - easier to think in order
- send precedes wait(s)

NOT *sequential*

index	core 1	core 2	core 3
2	SendSignal *s* *c*2		
5			SendSignal *s* *c*2
	second signal overlaps the first, only one registered		
1		long computation	
3		WaitSignal *s*	
4		computation	
	no signal is sent, so the next WaitSignal *blocks*		
6		WaitSignal *s*	

- can execute out of order

does NOT imply *order independent*

index	core 1	core 2	core 3
1		long computation	
5			SendSignal *s* *c*2
3		WaitSignal *s*	
4		computation **using wrong assumptions**	
2	SendSignal *s* *c*2		
6		WaitSignal *s*	

Linear-Time Verification

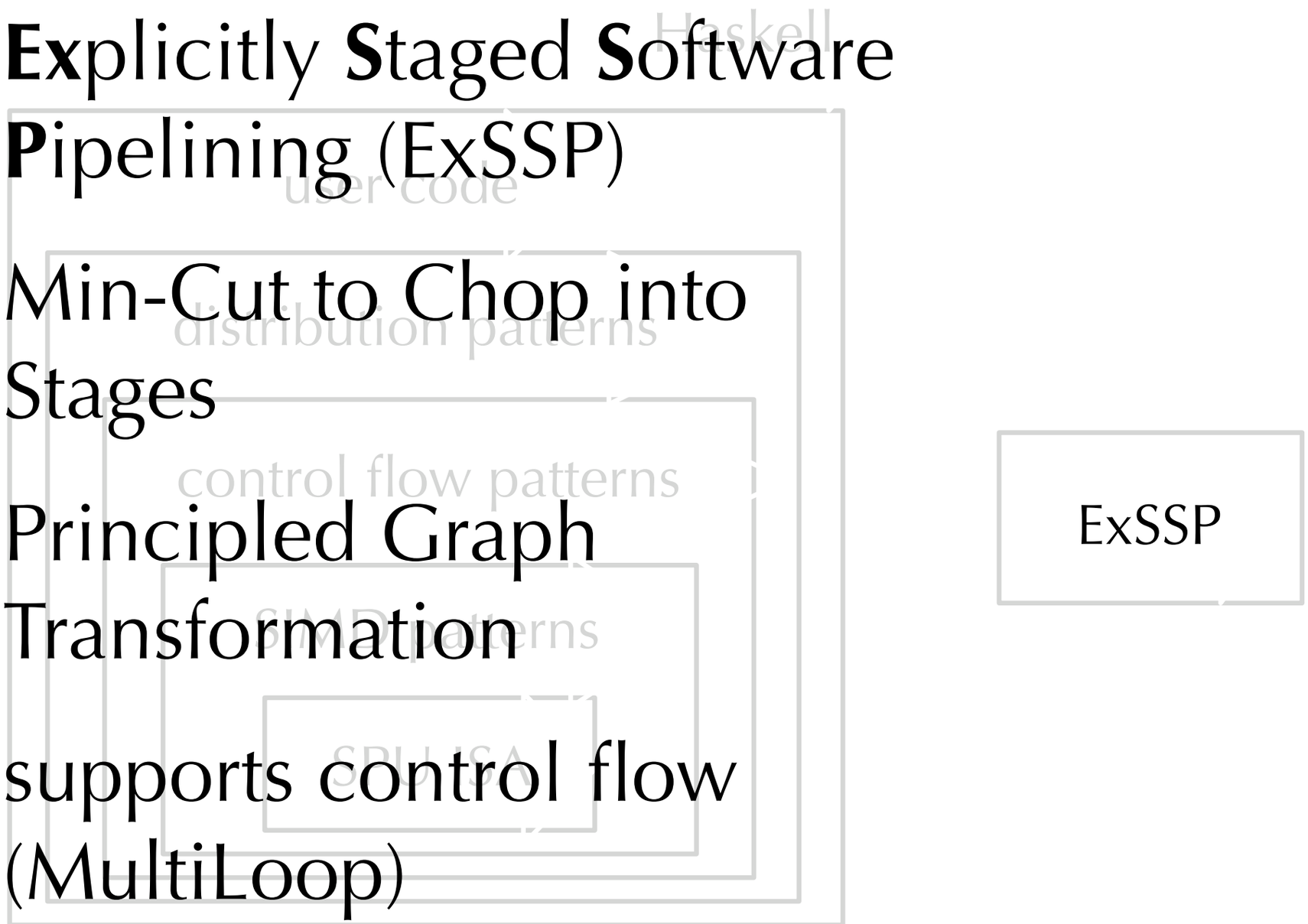
- must show
  - results are independent of execution order
    - no deadlocks
- need to keep track of all possible states
- linear in time = one-pass verifier
  - constant space
    - i.e. possible states at each instruction

Impact

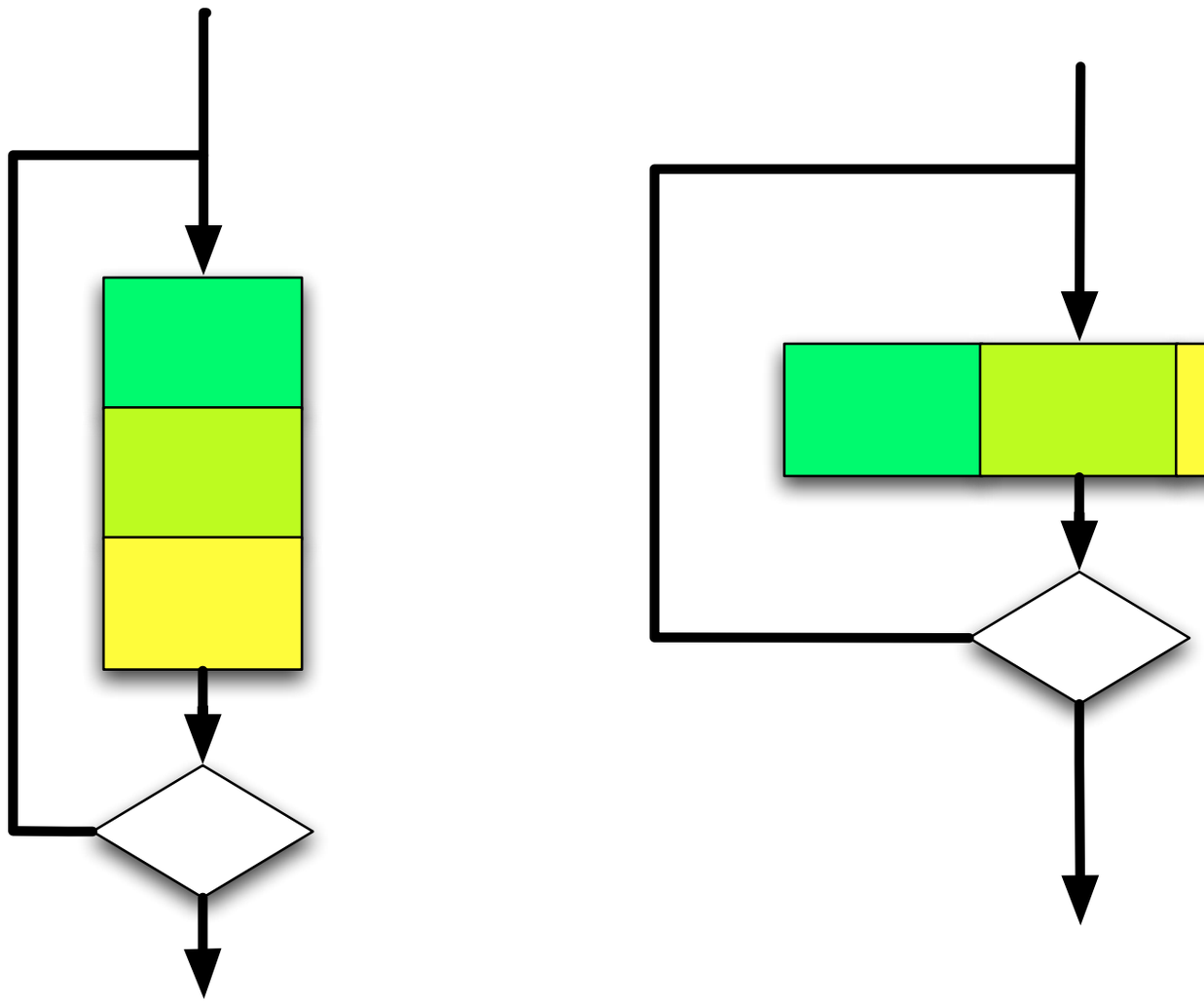
- no parallel debugging !!
- every optimization trick used for ILP can be adapted
- ready for algorithm “skeletons”
  - e.g. map, reduce
- enables optimization for power reduction:
  - replace caching with data in-flight

Instruction Scheduling

- **Ex**plicitly **S**taged **S**oftware **P**ipelining (ExSSP)
- Min-Cut to Chop into Stages
- Principled Graph Transformation
- supports control flow (MultiLoop)

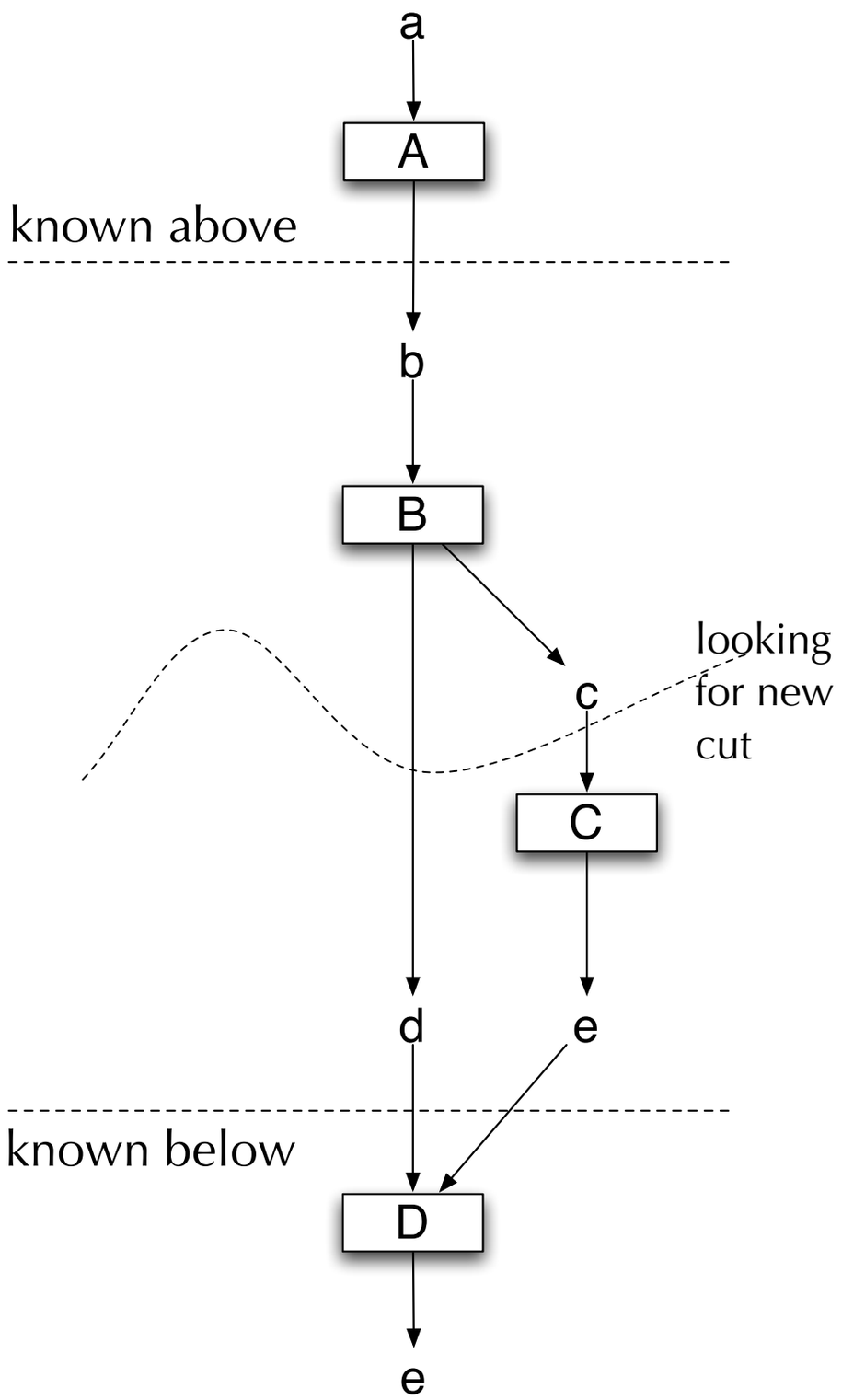


Software Pipelining



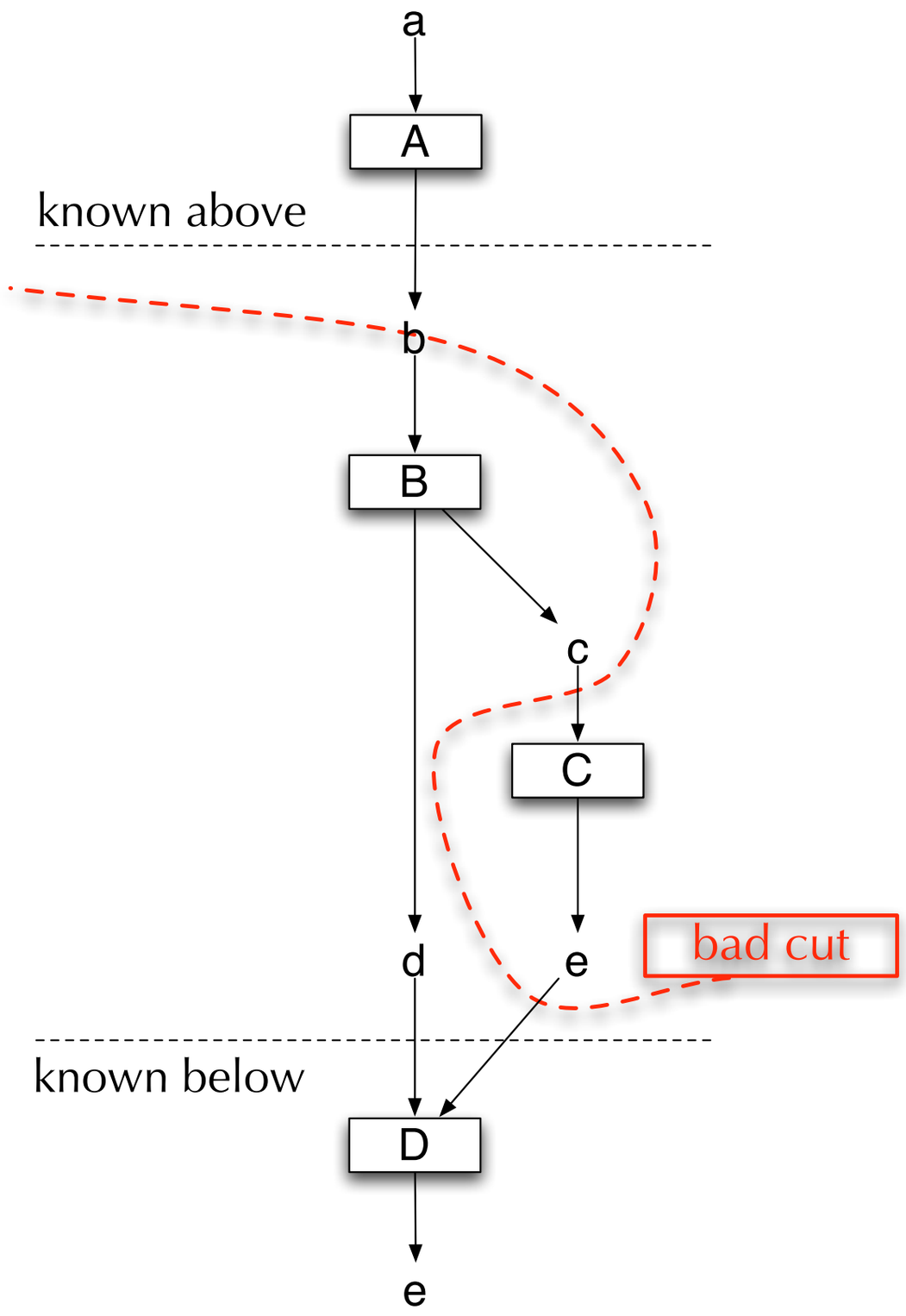
- hide latency
- same length loop body

Min-Cut Preparation



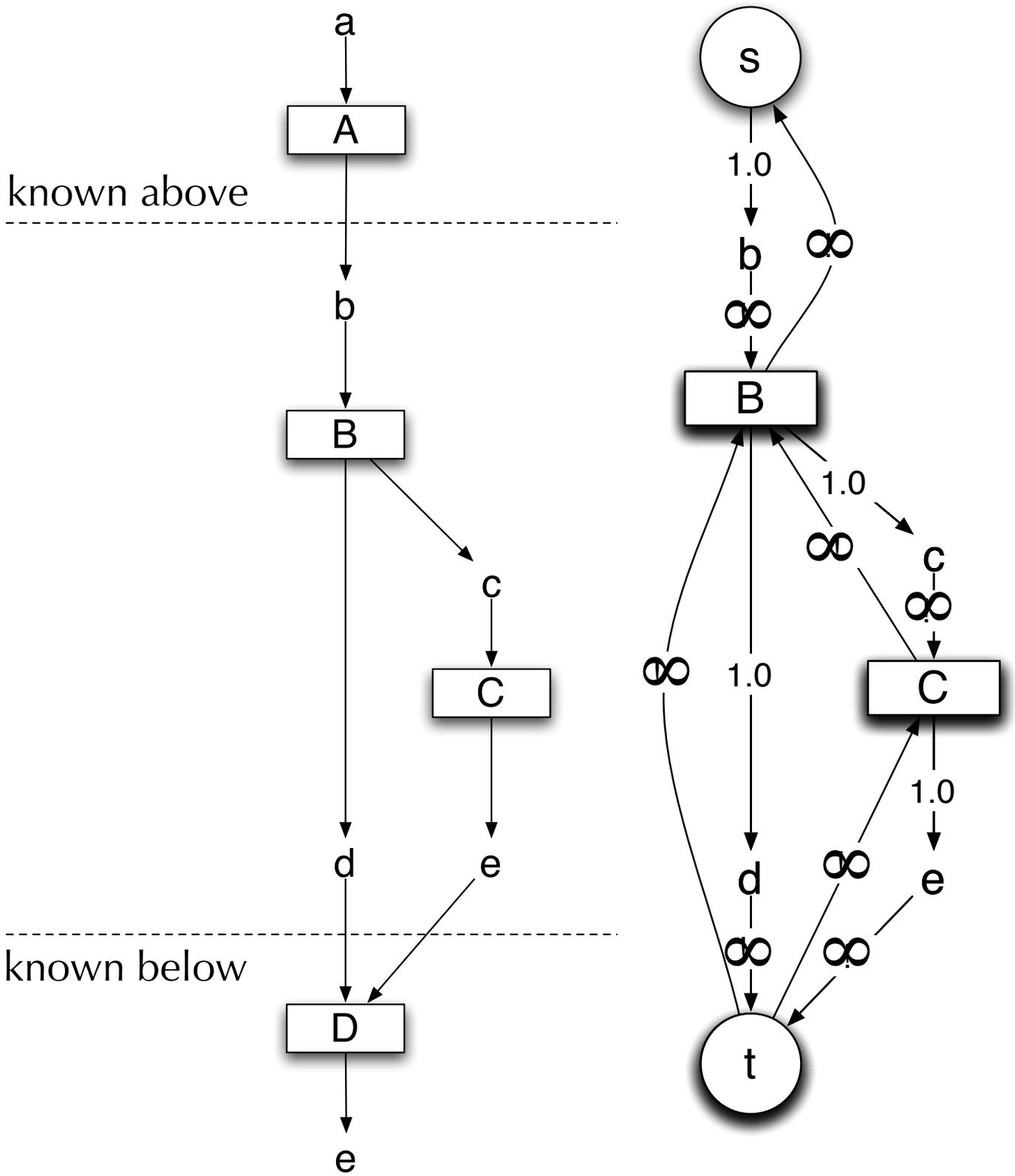
- cut into stages
- one by one
- minimize live registers

Bad Cut



- c produced in later stage
- c used in earlier stage

Transformation



collapse assigned

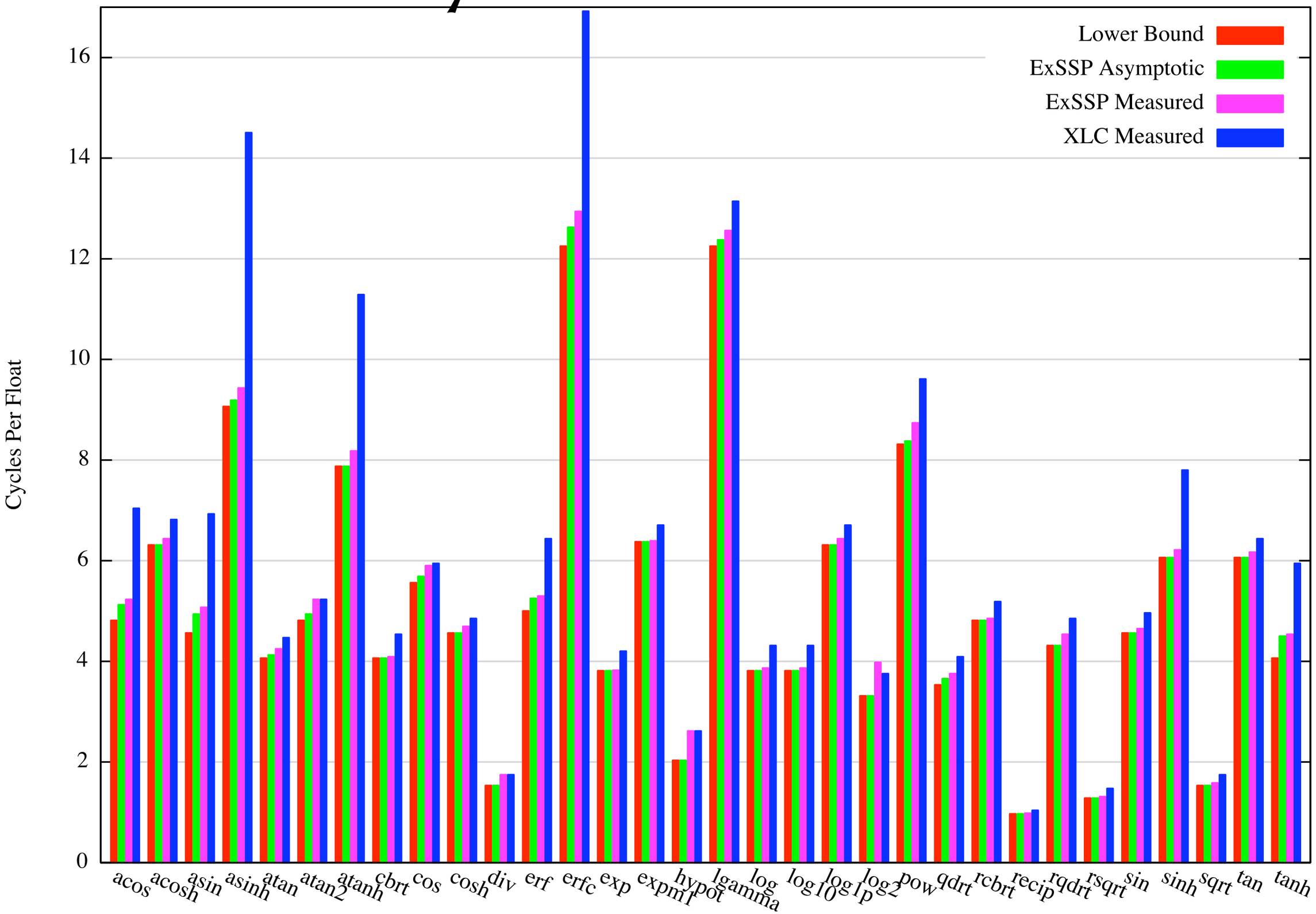
nodes and edges become nodes

weight 1 production edges

weight ∞ consumption edges

weight ∞ backwards edges

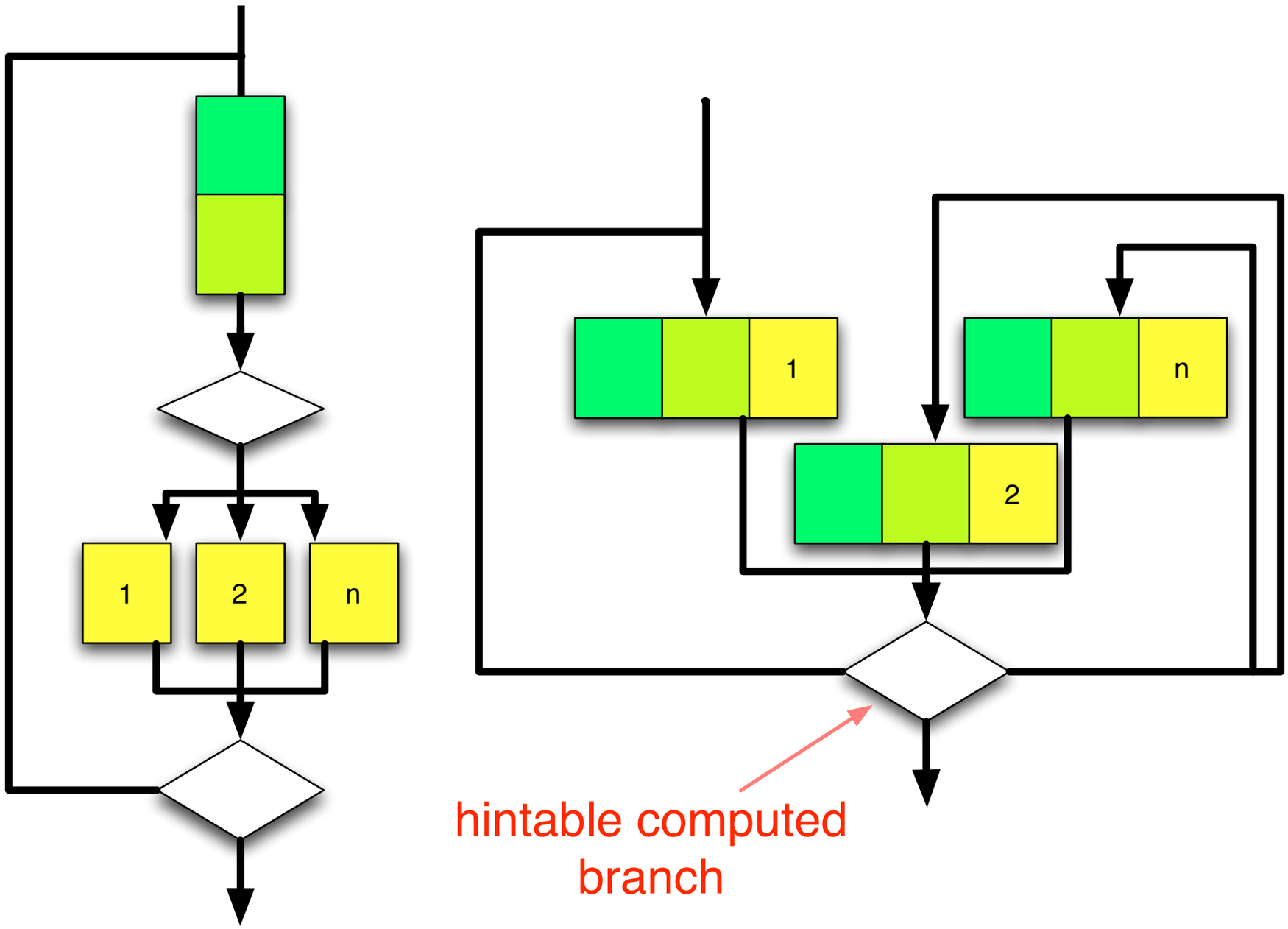
Cycles / Float



Not Just Faster

- why a new algorithm?
  - higher assurance
    - principled graph transformation
- not just scheduling instructions
- **novel control flow**
  - via nested control flow graphs

Example 1: MultiLoop



Coconut

- so far
  - functional-assembly programming
  - SIMD++
  - unbeaten scheduler
  - multi-core distribution
    - proof of soundness
- next
  - Multi-Core Patterns

Legal Notices

- Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc., in the United States, other countries, or both.
- IBM is a registered trademark of International Business Machines Corporation in the United States, other countries, or both.
- Other company, product, and service names may be trademarks or service marks of others.

Thanks

Stephen Adams
Kevin Browne
Shiqi Cao
Nathan Cumpson
Saeed Jahed
Damith Karunaratne
Clayton Goes
Gabriel Grant
William Hua
Fletcher Johnson
Wei Li
Nick Mansfield
Mehrdad Mozafari
Adam Schulz
Anuroop Sharma
Sanvesh Srivastava
Wolfgang Thaller
Gordon Uszkay
Christopher Venantius
Paul Vrbik
Fei Zhao

Robert Enenkel

IBM Centre for Advanced Studies, CFI, OIT, NSERC and
Apple Canada for research support.