Coconut
COde CONstructing User Tool

Christopher Kumar Anand
Wolfram Kahl
+
McMaster students

Robert Enenkel
William O'Farrell

McMaster - 2019 April 8

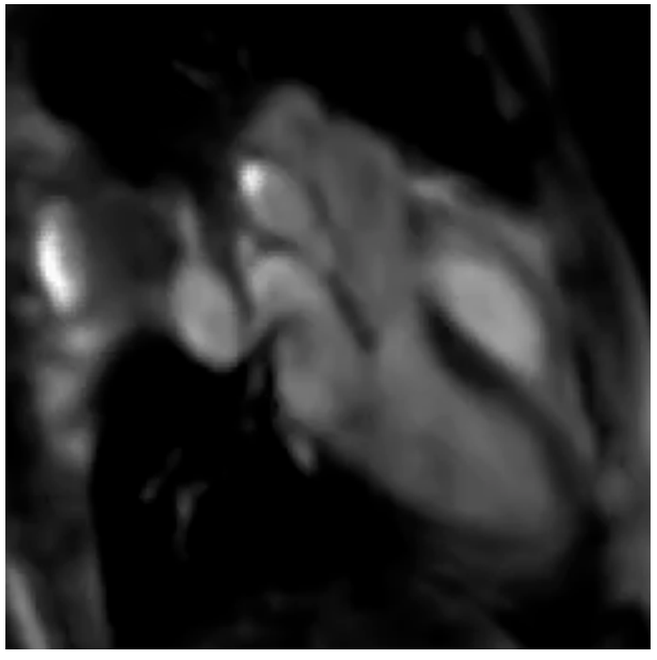
Award Winner

- 2017 - IBM CAS Faculty Fellow of the Year
- 2018 - IBM CAS Project of the Year
- why?
- highest ranking in internal IBM patent reviews
- dramatic acceleration of ML via instruction set/compiler co-optimization

Students

Stephen Adams
Konrad Anand
Tanya Bouman
Simon Broadhead
Kevin Browne
Shiqi Cao
Kriston Costa
Nathan Cumpson
Curtis d'Alves
Michal Dobrogost
Lucas Dutton
Saeed Jahed
Damith Karunaratne
Umme Salma Gadriwala
Clayton Goes
Gabriel Grant
William Hua
Yumna Irfan
Yusra Irfan
Fletcher Johnson
Wei Li
Stephanie Lin
Nick Mansfield
Mehrdad Mozafari
Adele Olejarz
Jessica Pavlin
Adam Schulz
Anuroop Sharma
Sanvesh Srivastava
Wolfgang Thaller
Gordon Uszkay
Christopher Venantius
Paul Vrbik
Sean Watson
James You
Fei Zhao

We can write safe software.
We can write fast software.

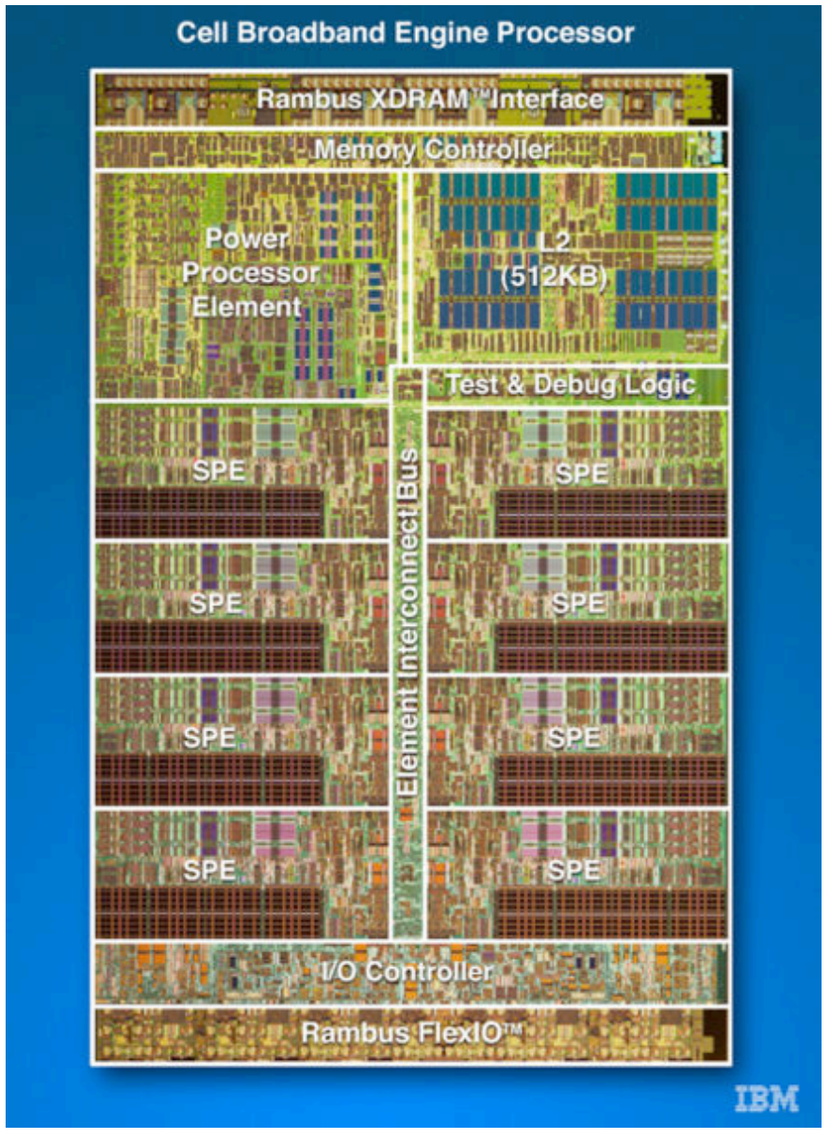


Sometimes
we need
both.

Performance = Parallelism

Cell BE

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



Roadmap

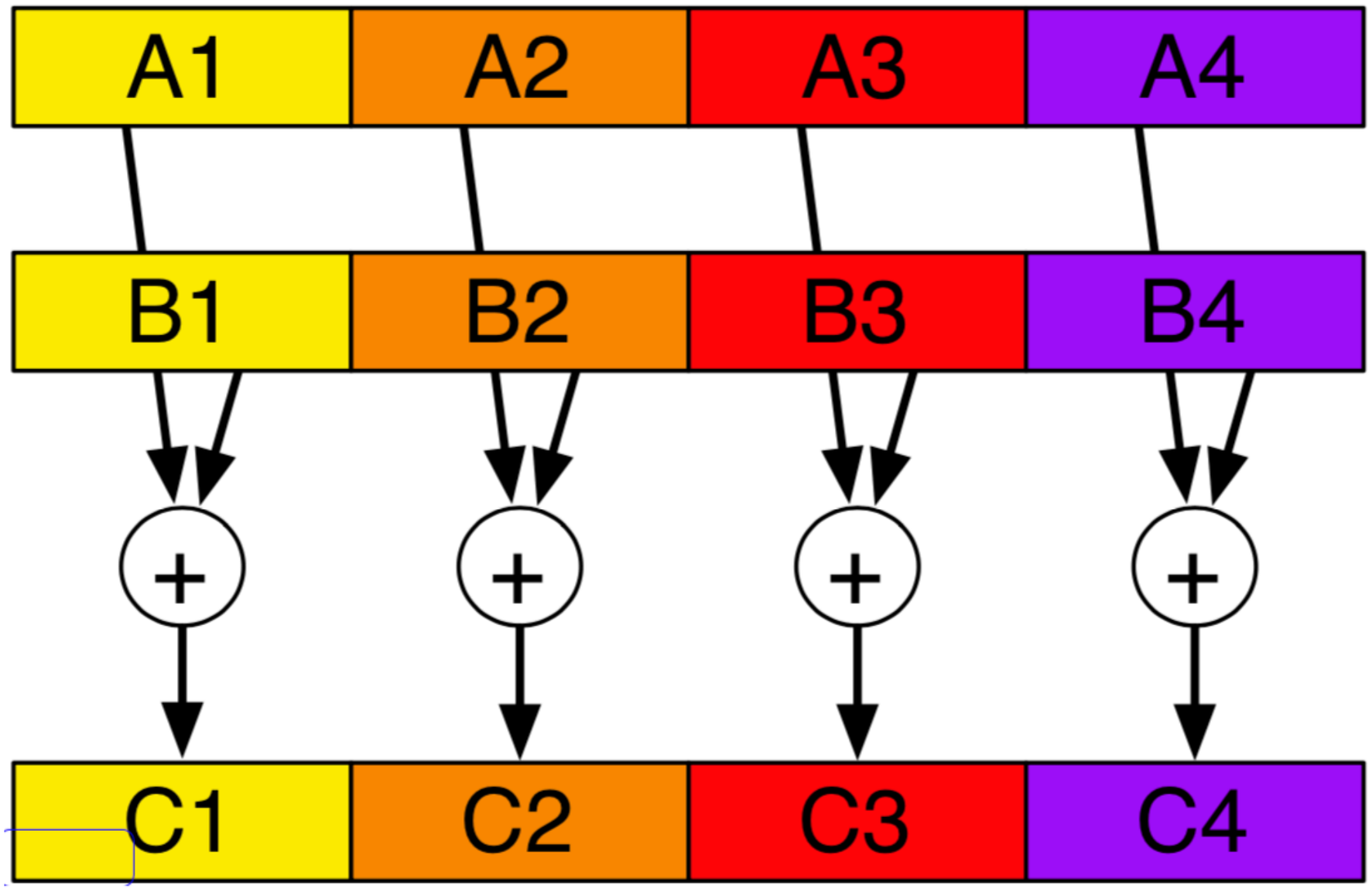
- SIMD Parallelism
  - ✔ extensible DSL captures patterns
  - ½ verification via graph transformation
  - ✔ generated library shipping (Cell BE SDK 3.0)
- Multi-Core Parallelism
  - ✔ model on ILP
  - ✔ generation via graph transformation
  - ✔ linear-time verification
  - ✔ run time
- Distant Parallelism
  - ∞ verification via model checking

Scheduling: ExSSP ✔
Approximation: Karger's ½
Approximation: Continuous ½

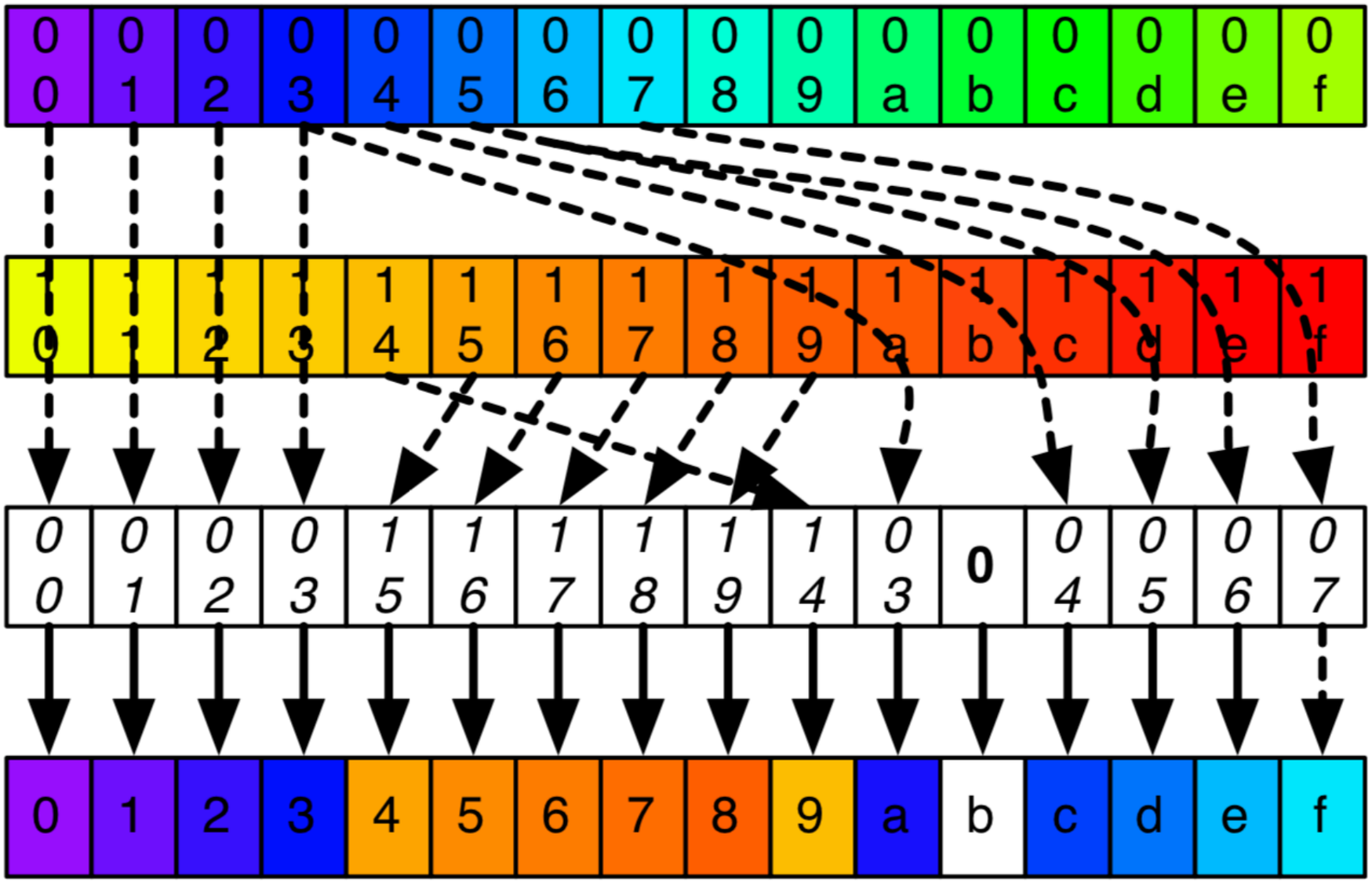
The Road to CoDesign

- Typical Math Function
- Lookups
- SIMD Lookup
- Accurate Table Method
- Exceptions
- New Instructions
- New New Instructions
- Sigmoid

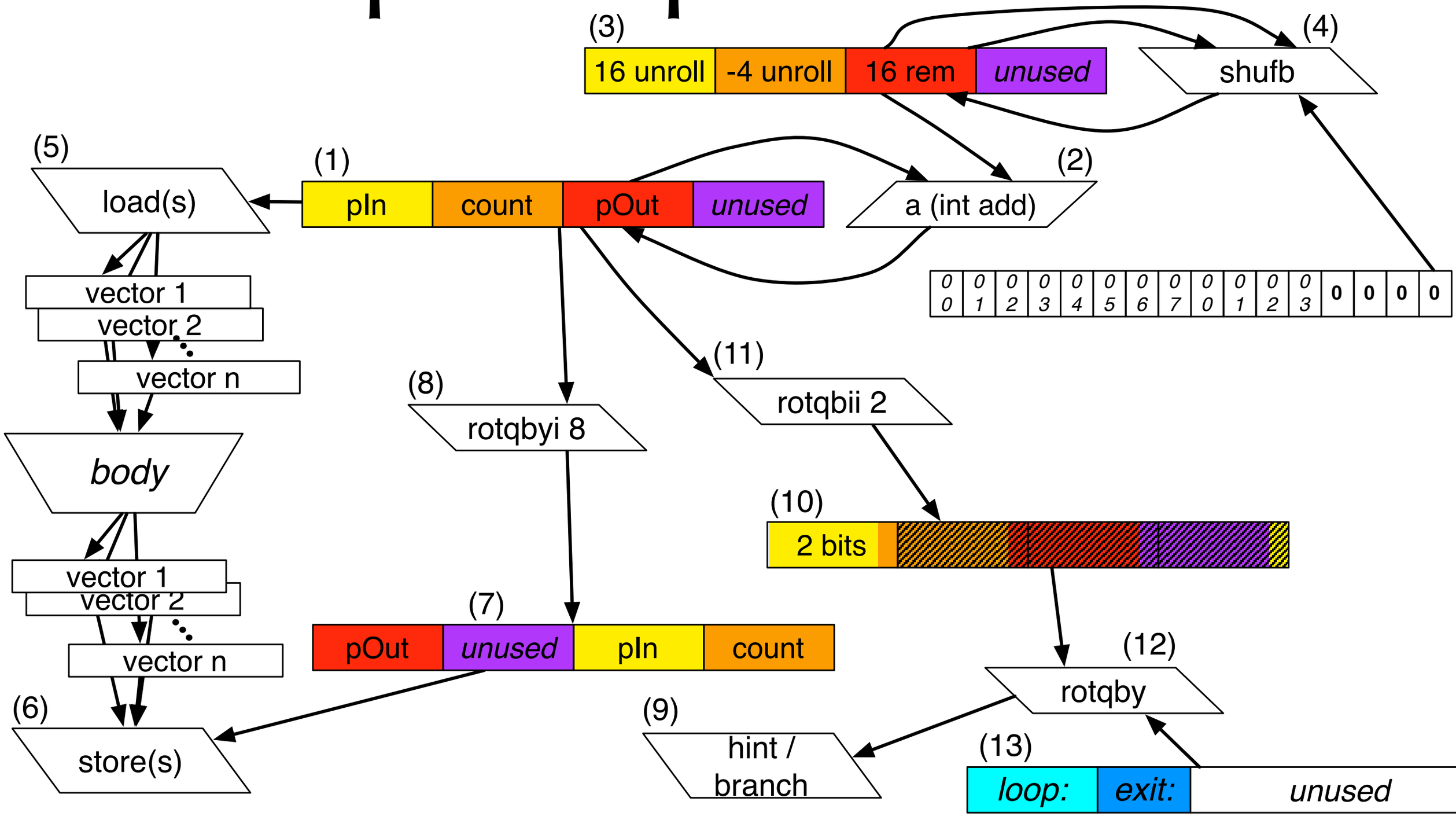
SIMD



weird SIMD


0 0 0 1 0 2 0 3 0 4 0 5 0 6 0 7 0 8 0 9 0 a 0 b 0 c 0 d 0 e 0 f
1 0 1 1 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 a 1 b 1 c 1 d 1 e 1 f
0 0 0 1 0 2 0 3 1 5 1 6 1 7 1 8 1 9 1 4 0 3 0 0 4 0 5 0 6 0 7
0 1 2 3 4 5 6 7 8 9 a b c d e f

Map Loop Overhead

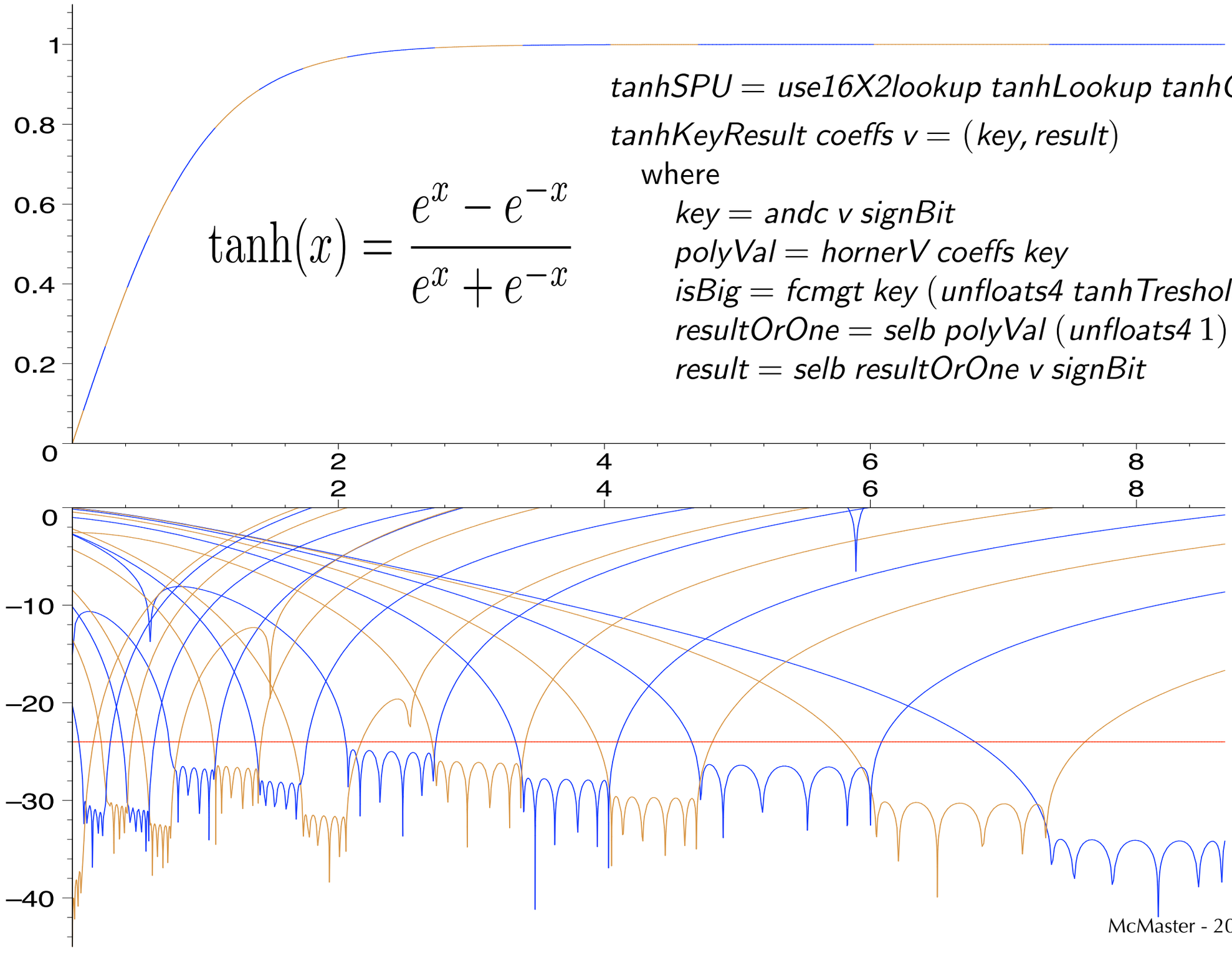


- one arithmetic instruction
- in/out pointers + induction variable + hint

Typical Math Function

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

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



SIMD coefficient lookup

sign exponent fraction 4 floats in vector reg

s eeeeeeee fffffffffffffffffffffff s eeeeeeee fffffffffff...

3 msb's of fraction determine 1 of 8 polynomials

Step 1: Rotate 3 msb's into low 5 bits.

`xxxxxxxx xxxxxxxx xxxxxxxx xxxfffxx`

Step 2: Shuffle to replicate into other bytes.

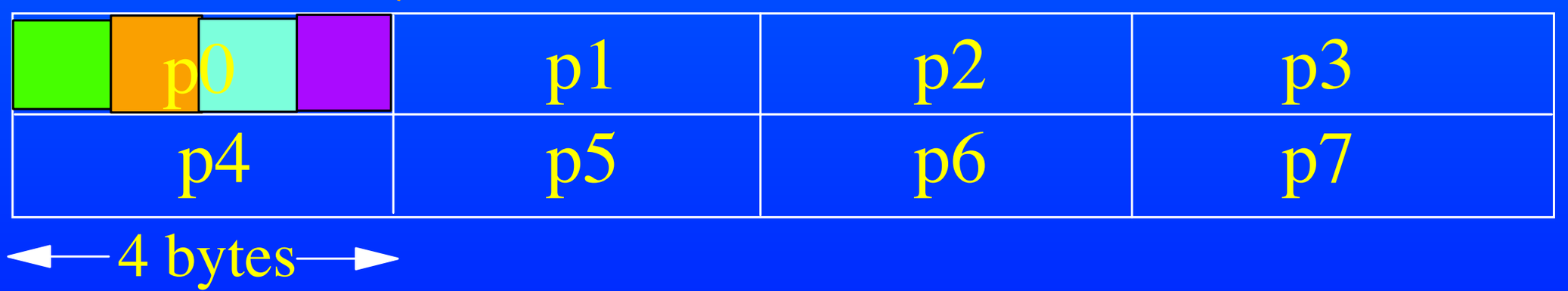
`xxxfffxx xxxfffxx xxxfffxx xxxfffxx`

SIMD coefficient lookup

Step 3: Select bits, put 0's before, byte index after.

`000fff00 000fff01 000fff10 000fff11`

Step 4: For each k=0,1,...,degree of polynomial
Shuffle to get a[k] = coeff of x^k for each of 4 polynomials in parallel.



fff determines which of p0,...,p7 is selected

Low Level DSL

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

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

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

where q and r are integers such that

$$e = 3 * q + r, \qquad 0 \le r < 3, \qquad (4)$$

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 (λf → 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

```
coeffs = lookup8Word (22, 20) expCoeffs24bits v
```

Evaluate the polynomial on the fractional part.

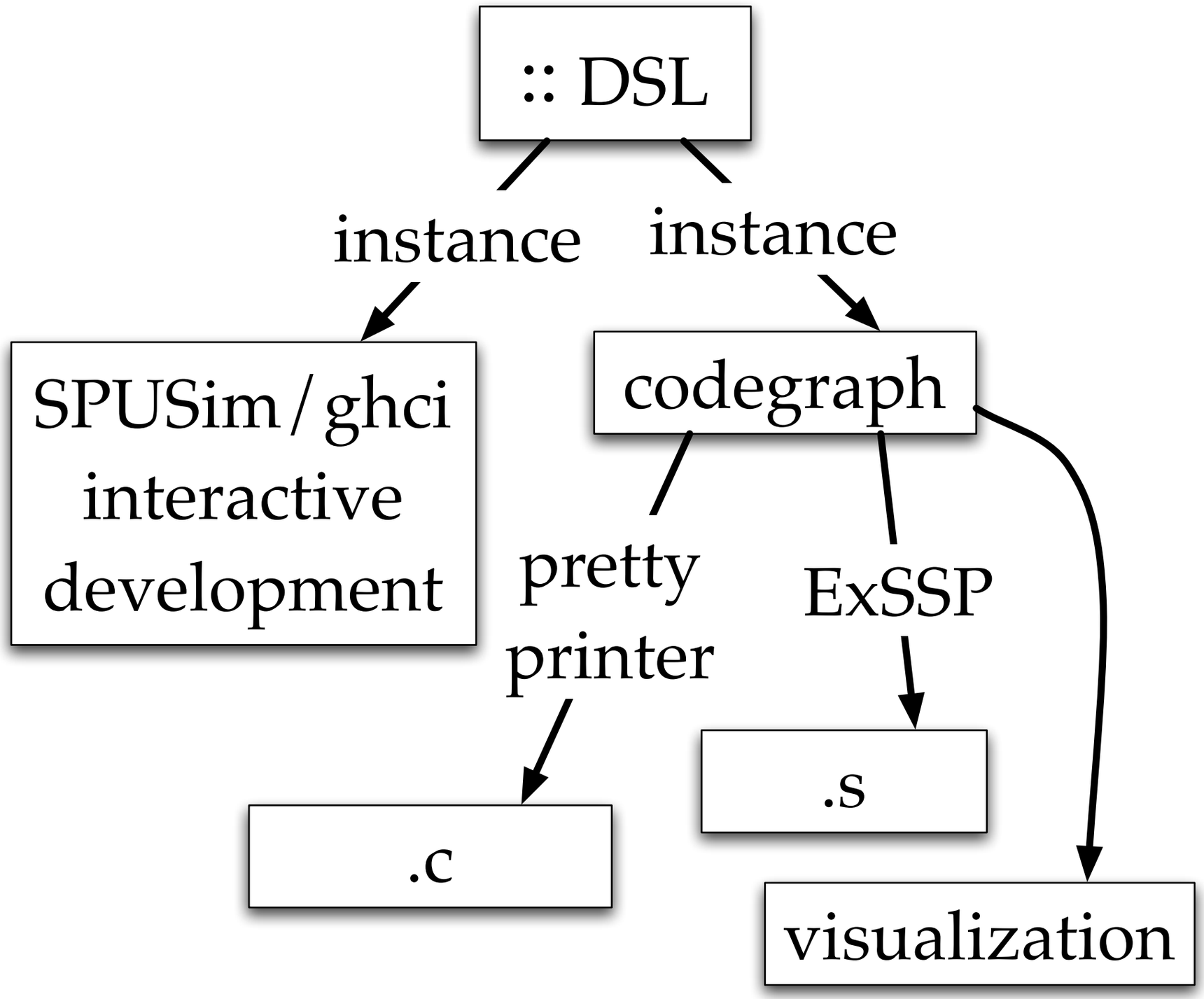
```
evalPoly = hornerV coeffs frac
```

Compile-Time Assertions

```
cbrtAssert :: Bool
cbrtAssert = List.and
    [divMod i 3 ≡ extractDivMod (approxDiv3 $ bias i)
     | i ← [expBias − 255 .. expBias]]
  where
    bias :: Integer → Val
    bias i = unwrds4 $ i + expBias
    extractDivMod w = case bytes w of
      _ : v1 : v2 : _ → (v1 − expBias, div v2 64)
      _               → error "impossible"
```

- simulate special instructions interactively
- verify assertions @ compile time

Multiple Instances



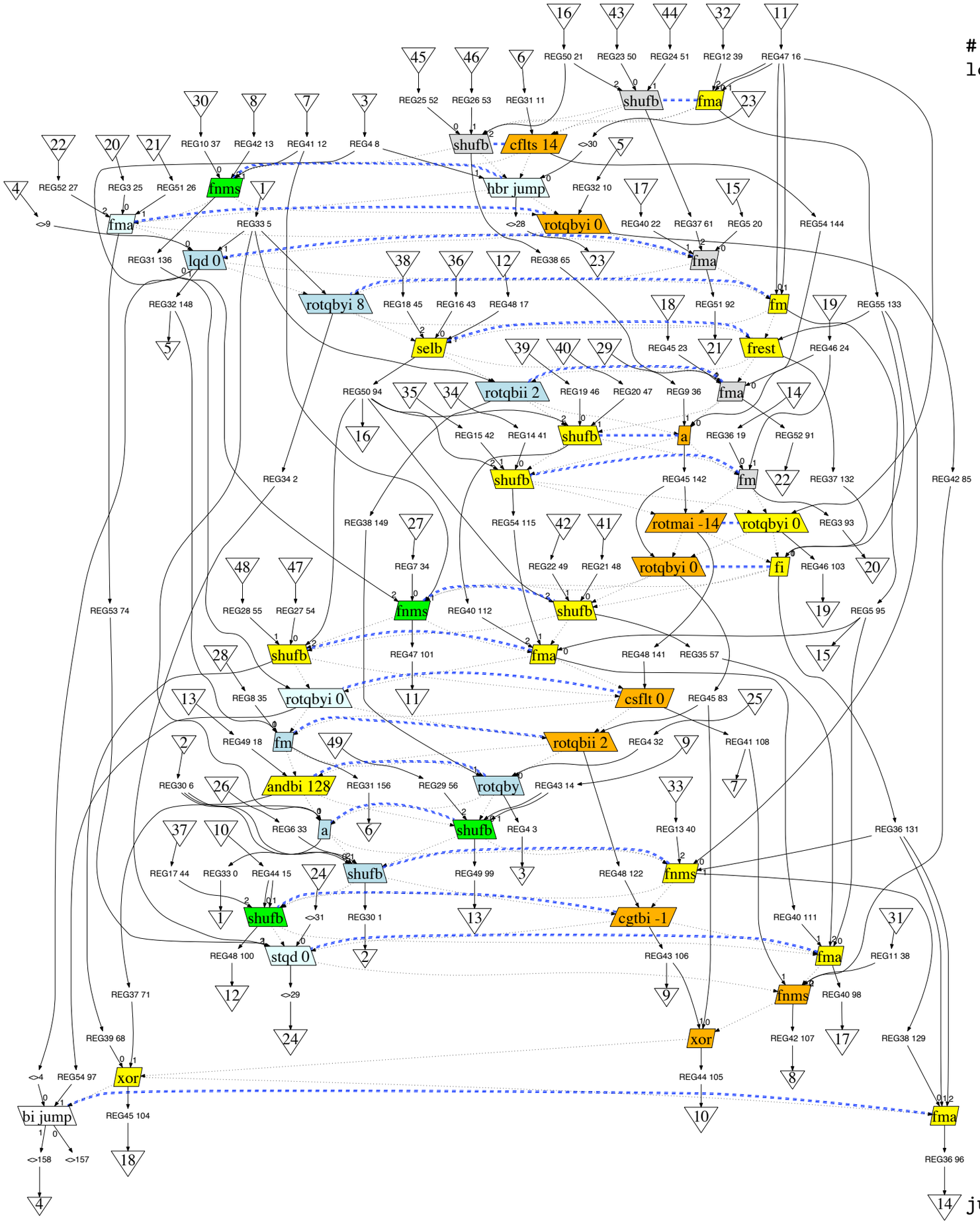


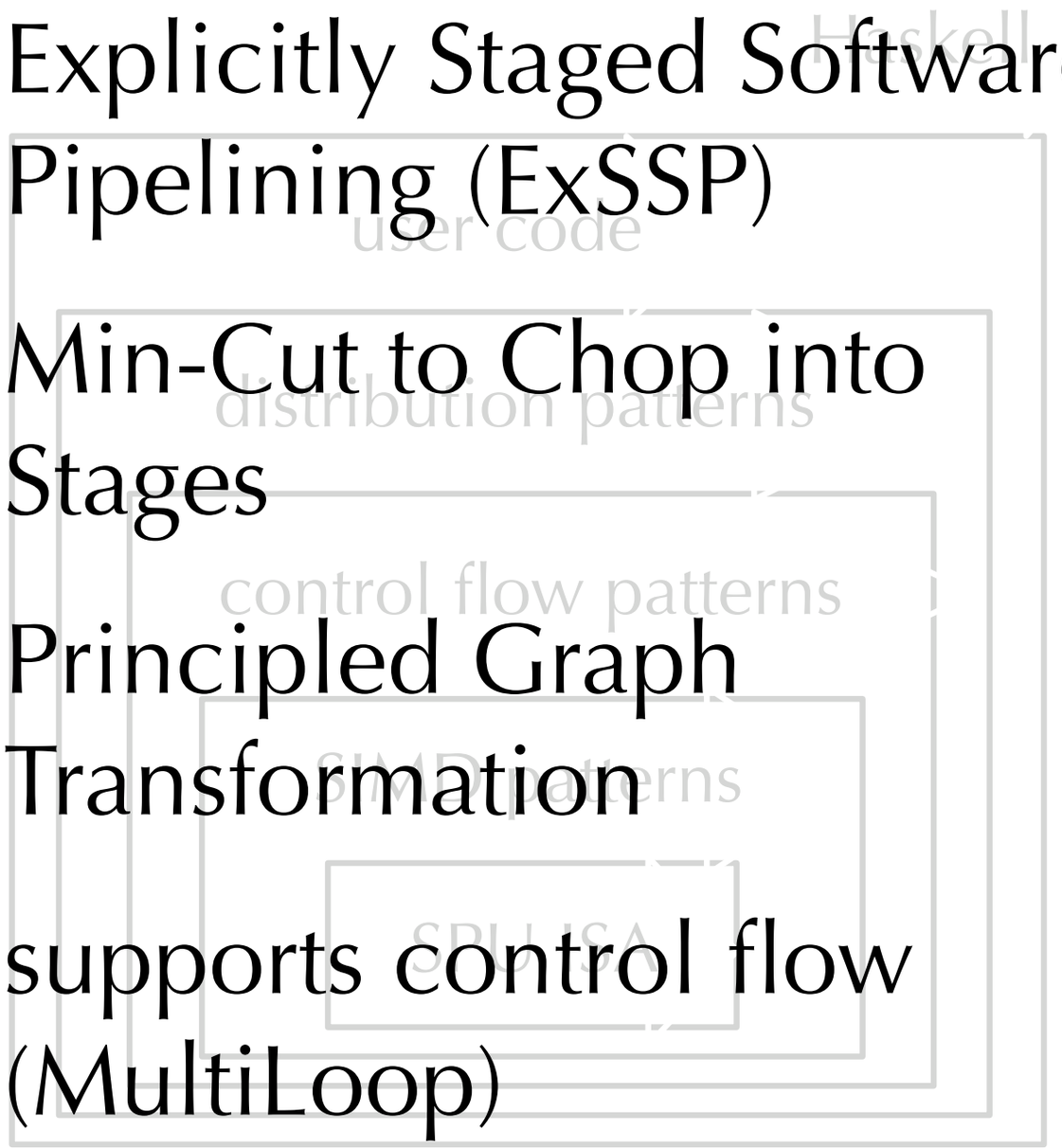
Figure 5. Scheduled assembly code graph for *tanSPU*.

```
# 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
```

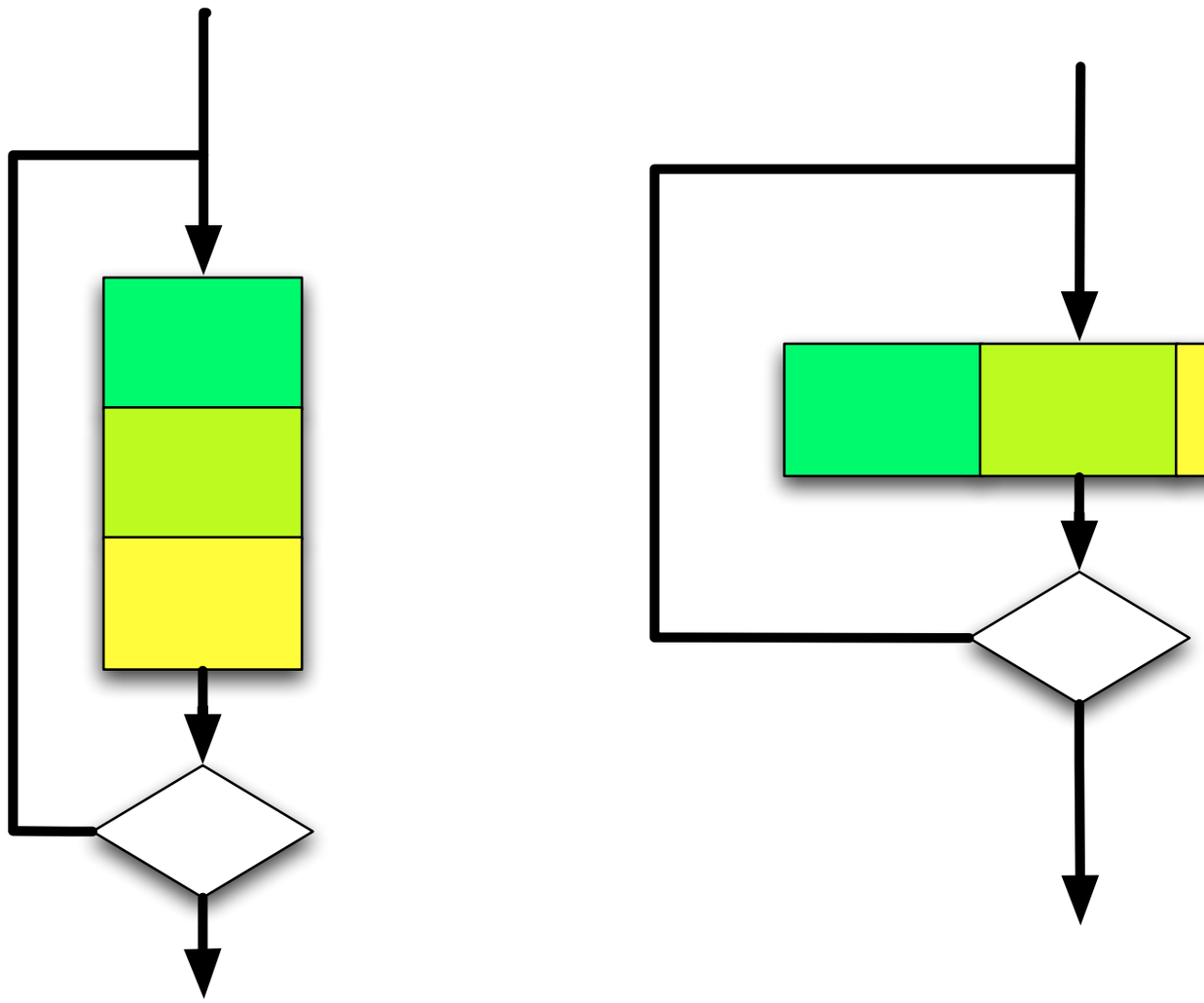
Figure 6. `tanSPU.s`

Instruction Scheduling

- Explicitly Staged Software Pipelining (ExSSP)
- Min-Cut to Chop into Stages
- Principled Graph Transformation
- supports control flow (MultiLoop)

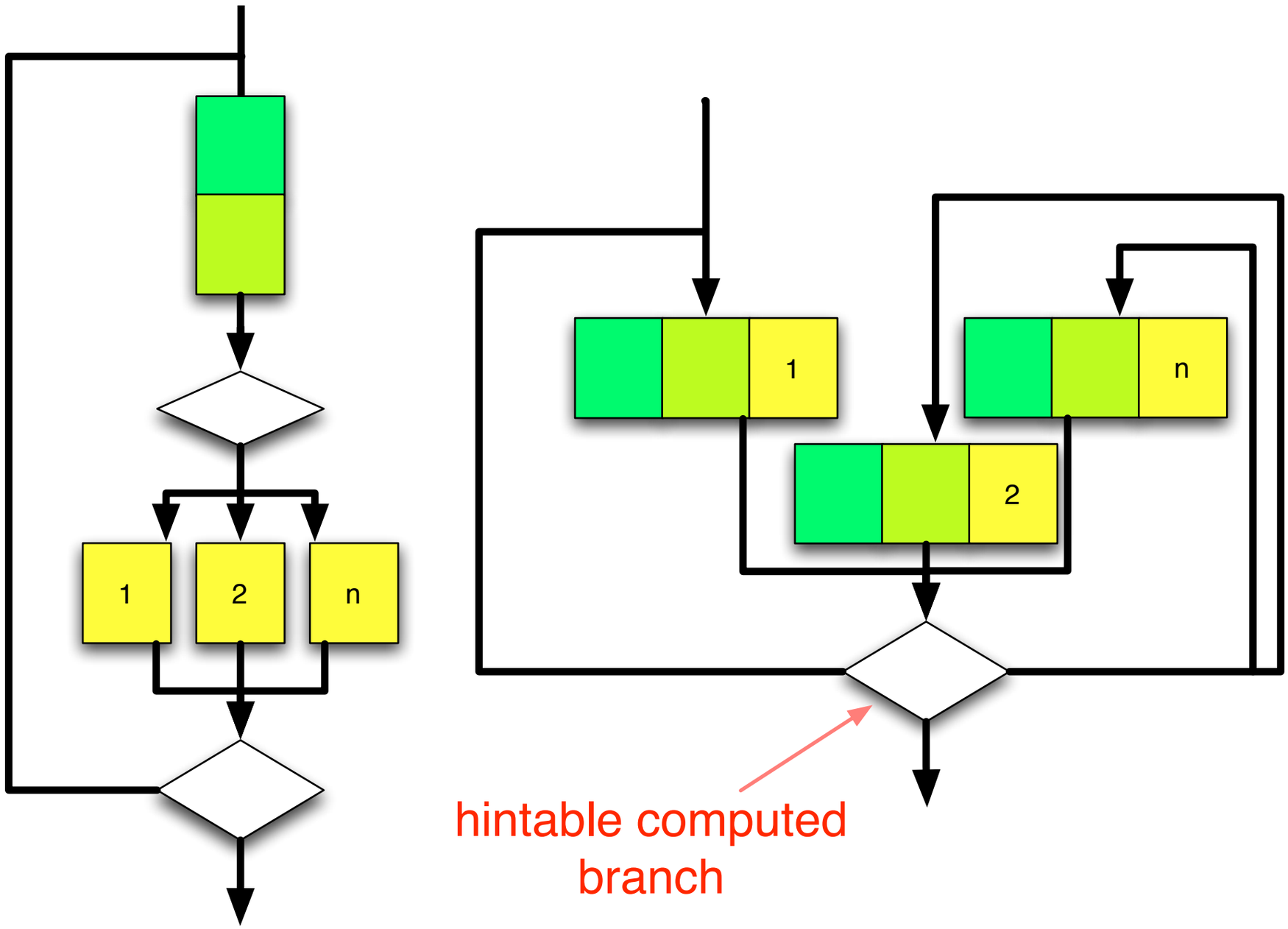


Software Pipelining

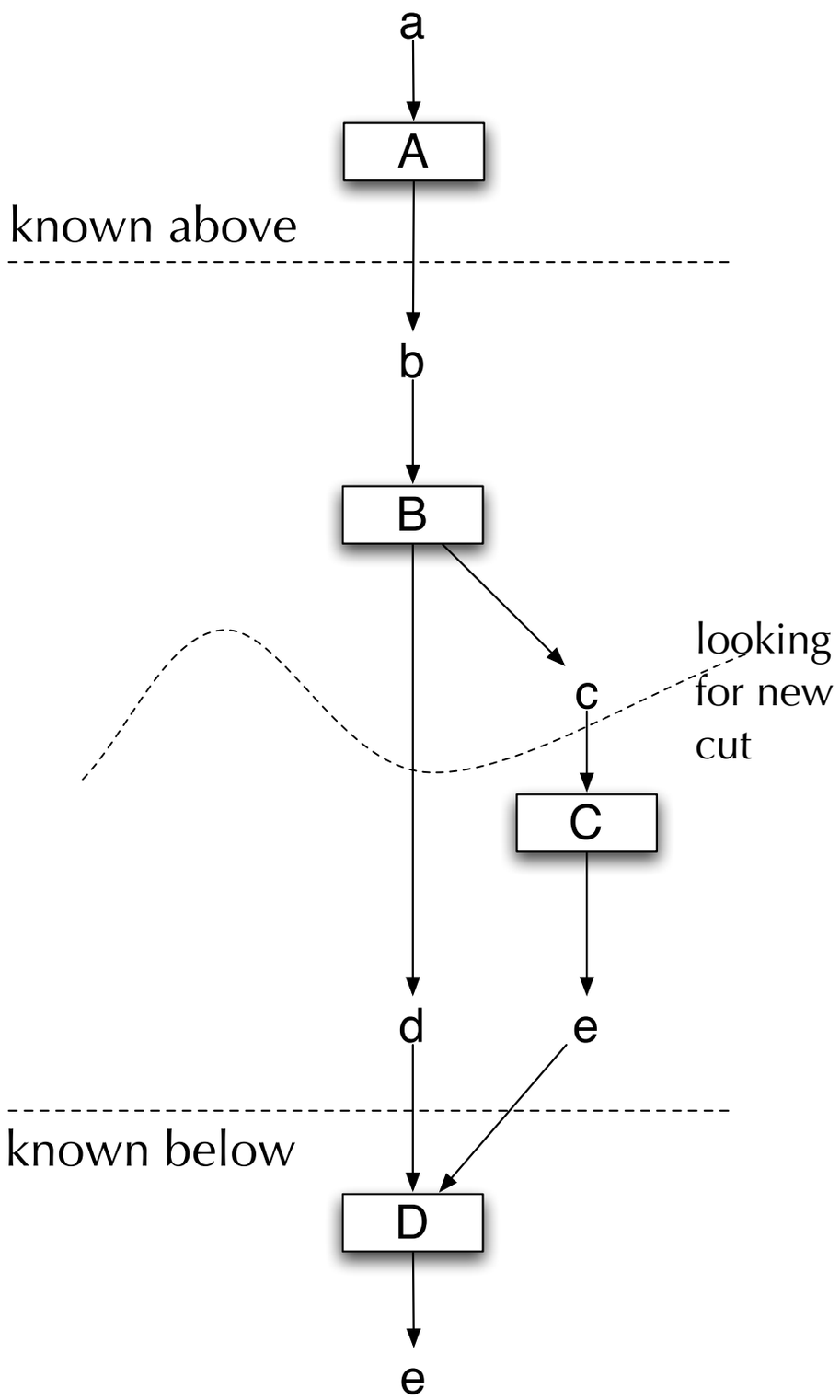


- hide latency
- same length loop body

MultiLoop

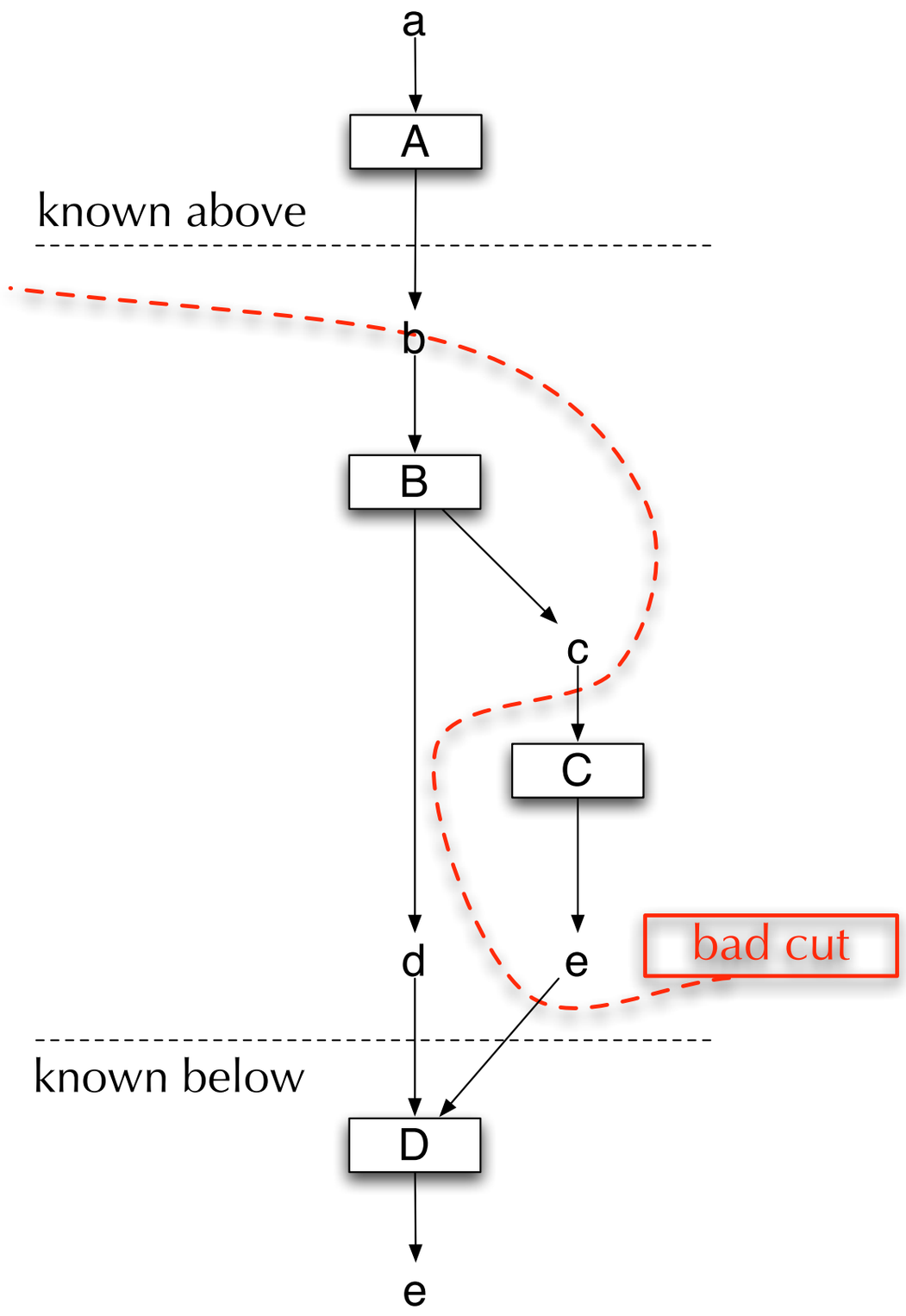


Min-Cut Preparation



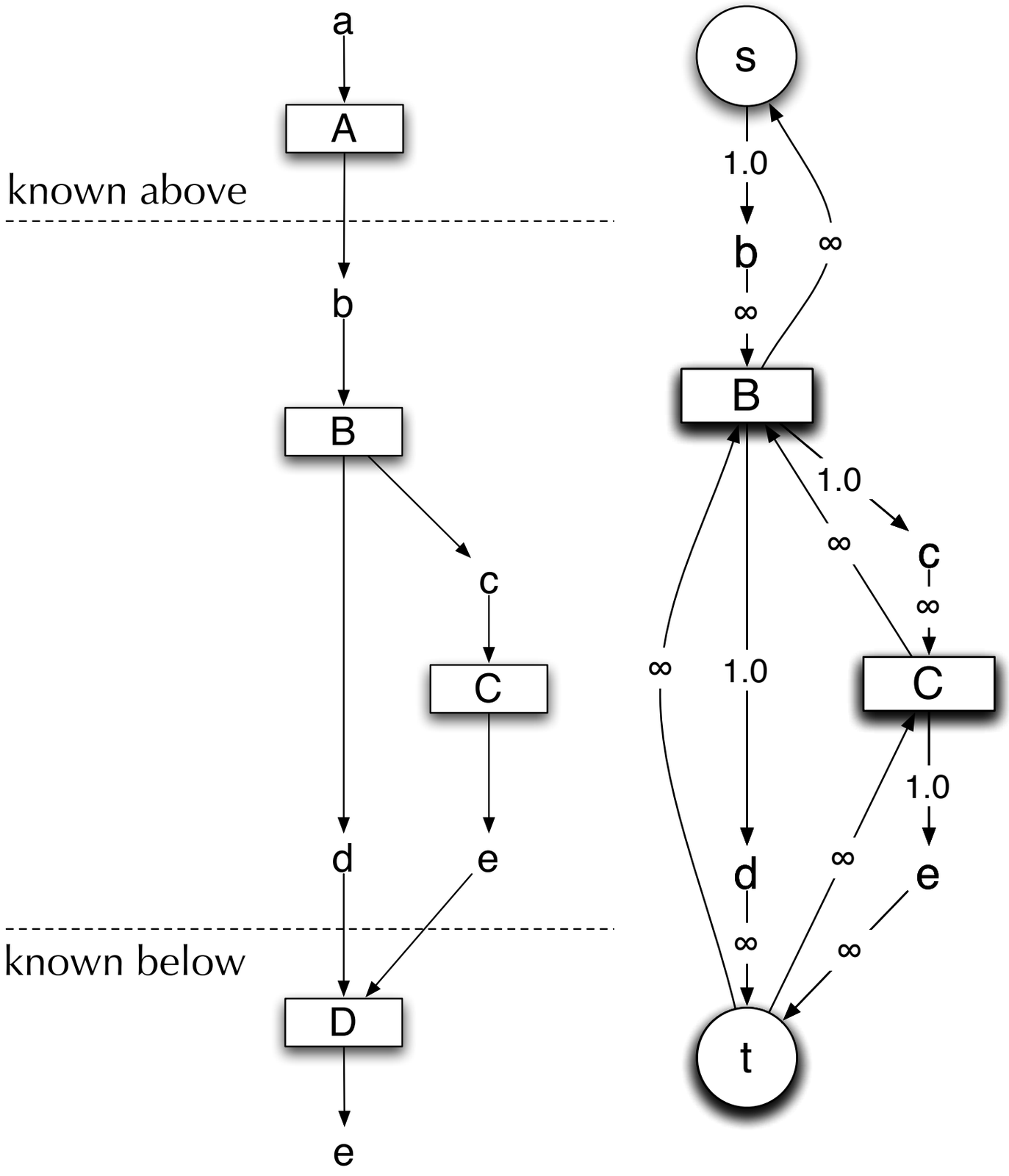
- cut into stages
- one by one
- minimize live registers

Bad Cut



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

Transformation



collapse assigned

nodes and edges become nodes

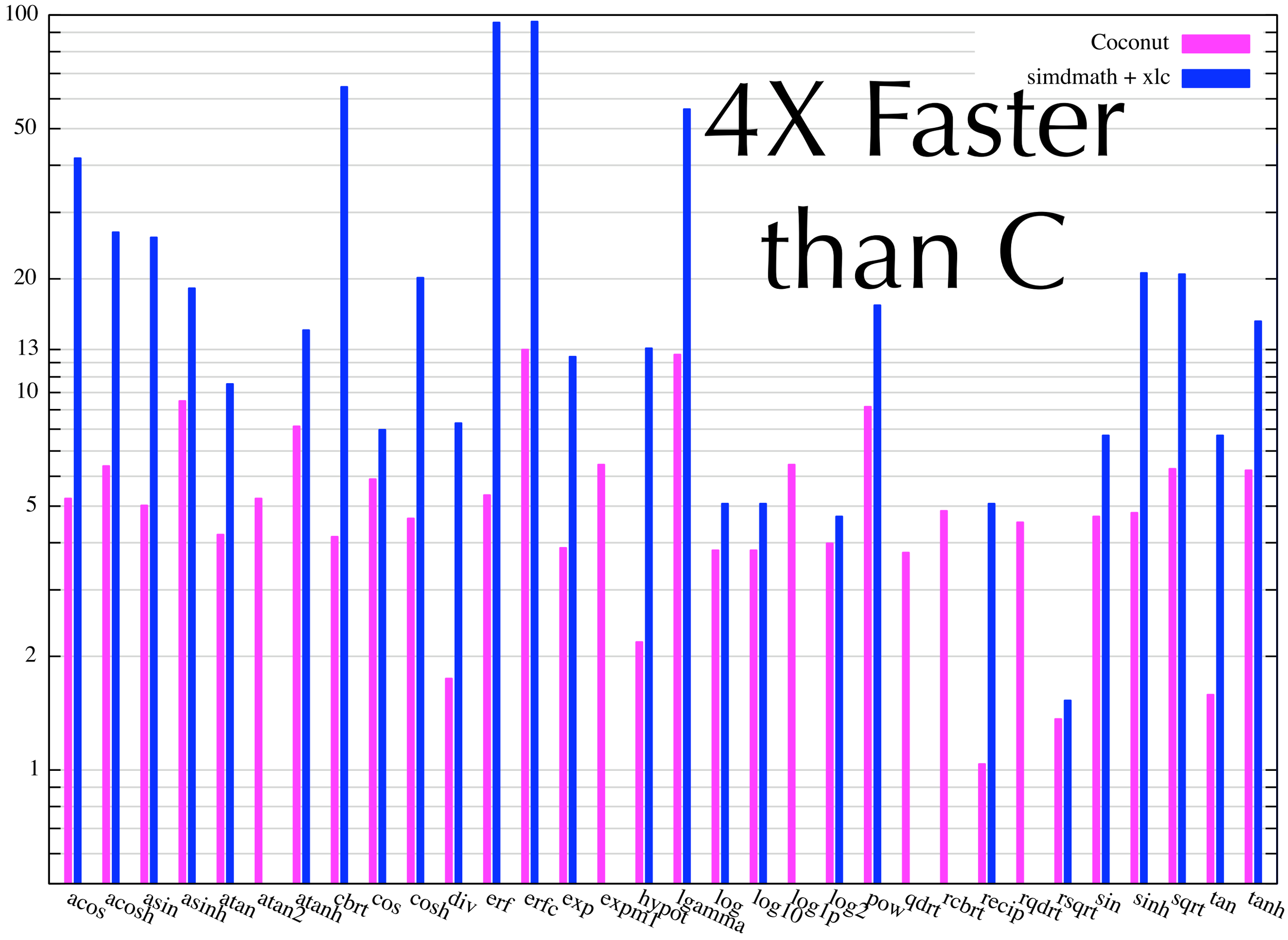
weight 1 production edges

weight ∞ consumption edges

weight ∞ backwards edges

97% Optimal Schedules

- Cell SPU was a great machine
  - 128 registers
  - two pipelines
  - simple dispatch rules
  - in-order exection
  - complete, public documentation


Coconut
simdmath + xlc
4X Faster
than C
100
50
20
13
10
5
2
1
acos
acosh
asin
asinh
atan
atan2
atanh
cbrt
cos
cosh
div
erf
erfc
exp
expm1
hypot
lgamma
log
log10
log1p
log2
pow
qdrt
rcbrt
recip
rqdrt
rsqrt
sin
sinh
sqrt
tan
tanh

Challenges = Opportunities

- out-of-order execution
- complex dispatch rules
- not enough registers
- developed two other approaches:
  - based on Karger's min-cut
    - an "approximation algorithm"
  - based on continuous optimization

Multi-Core = ILP Take 2

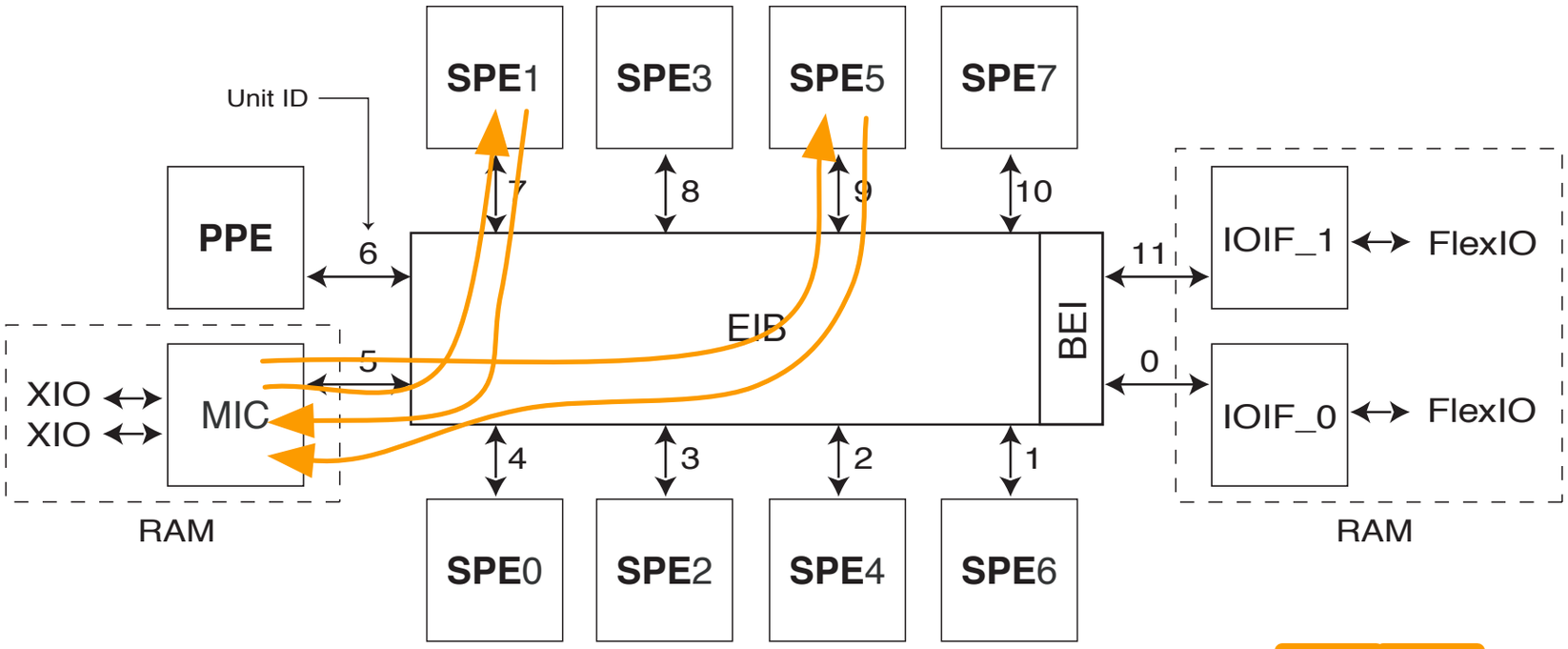
Instruction Level Parallelism	Multi-Core Parallelism
CPU	Chip
Execution Unit	Core
Load/Store Instruction	DMA
Arithmetic Instruction	Computational Kernel
Register	Buffer / Signal

The Catch: Soundness

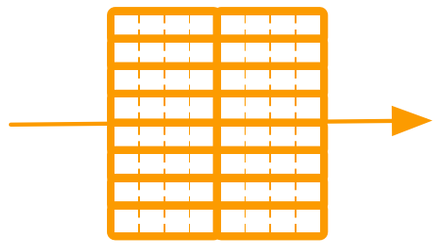
- on CPUs hardware maintains OOE
  - instructions execute out of order
  - hardware hides this from software
    - ensures order independence
- in our Multi-Core *virtual* CPU
  - compiler inserts synchronization
    - soundness up to software
    - uses asynchronous communication

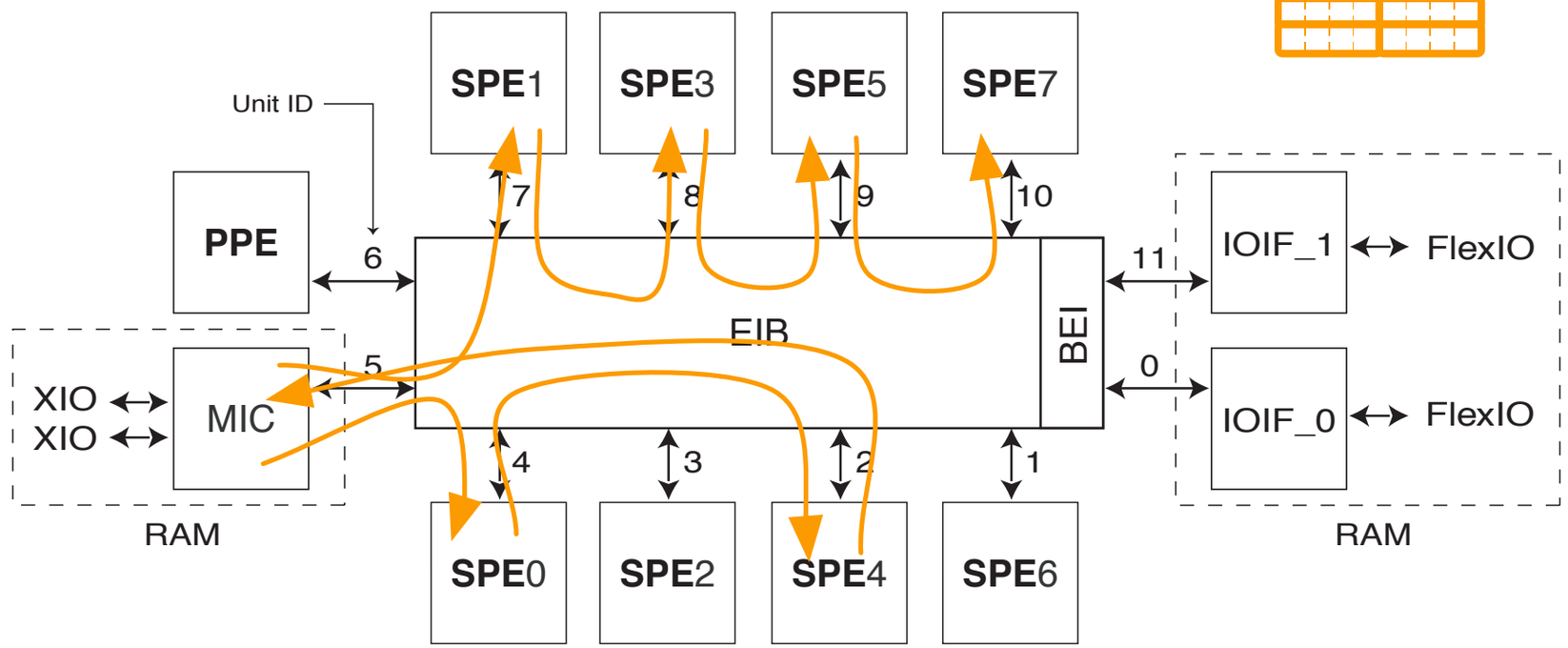
Asynchronous

- 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

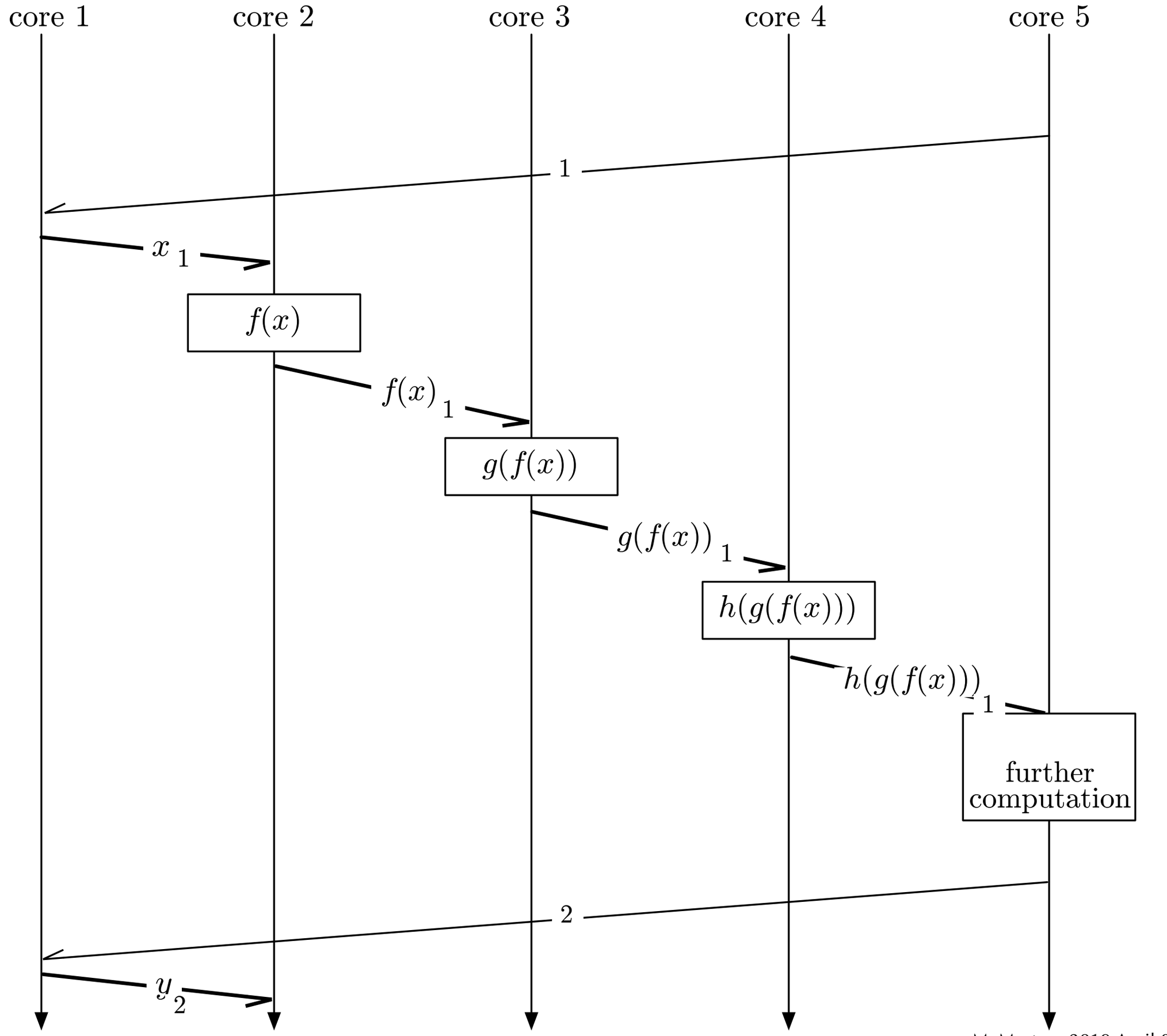


Memory Bound

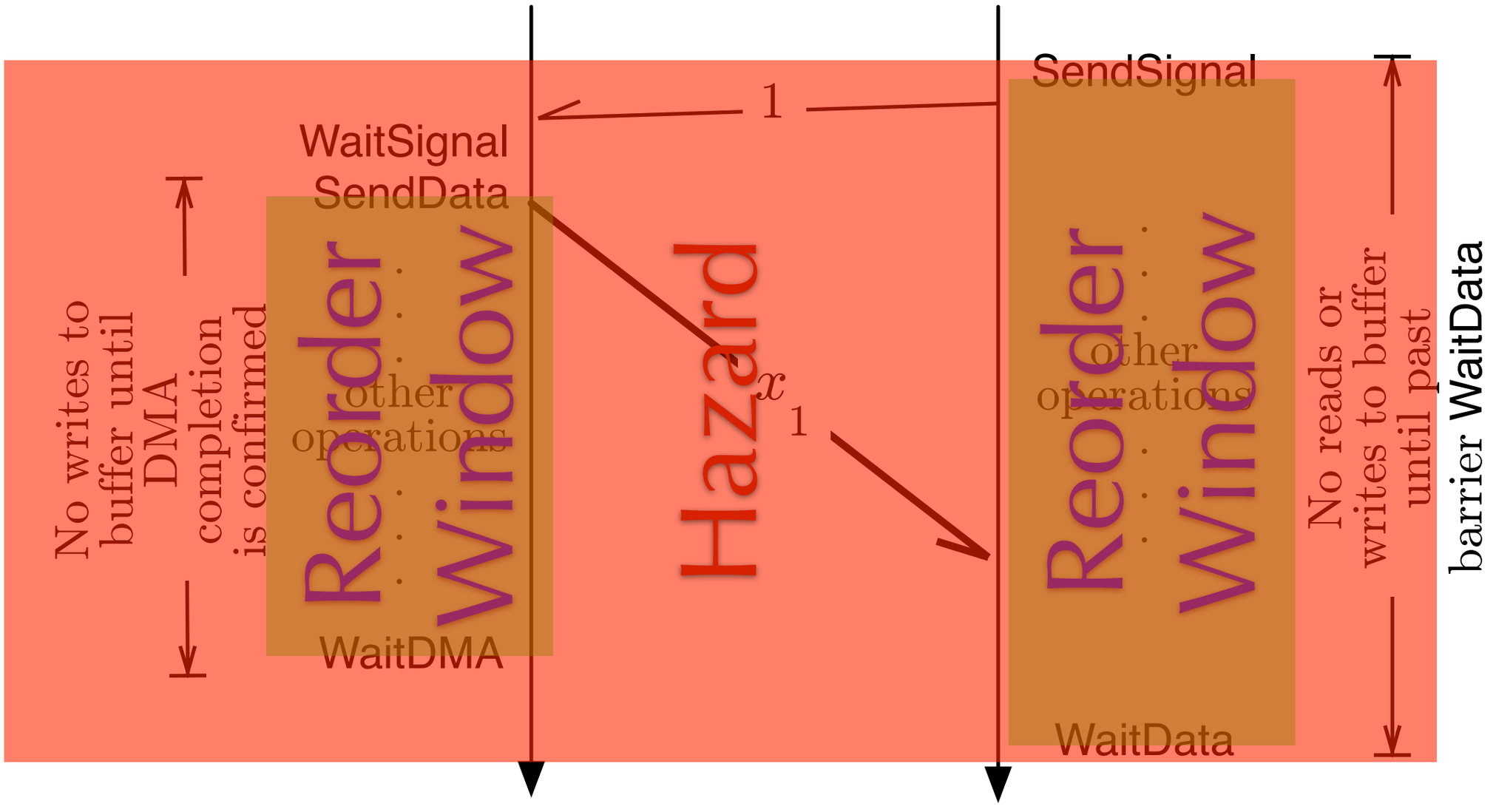




Comp Bound


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
SendSignal *core signal*	send a signal to distant core
WaitSignal signal	wait for signal to arrive

locally Sequential Program

index	core 1	core 2	core 3
1		long computation	
2	SendSignal $s \rightarrow c2$		
3		WaitSignal s	
4		computation	
5			SendSignal $s \rightarrow c2$
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 $s \rightarrow c2$		
5			SendSignal $s \rightarrow c2$
	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*

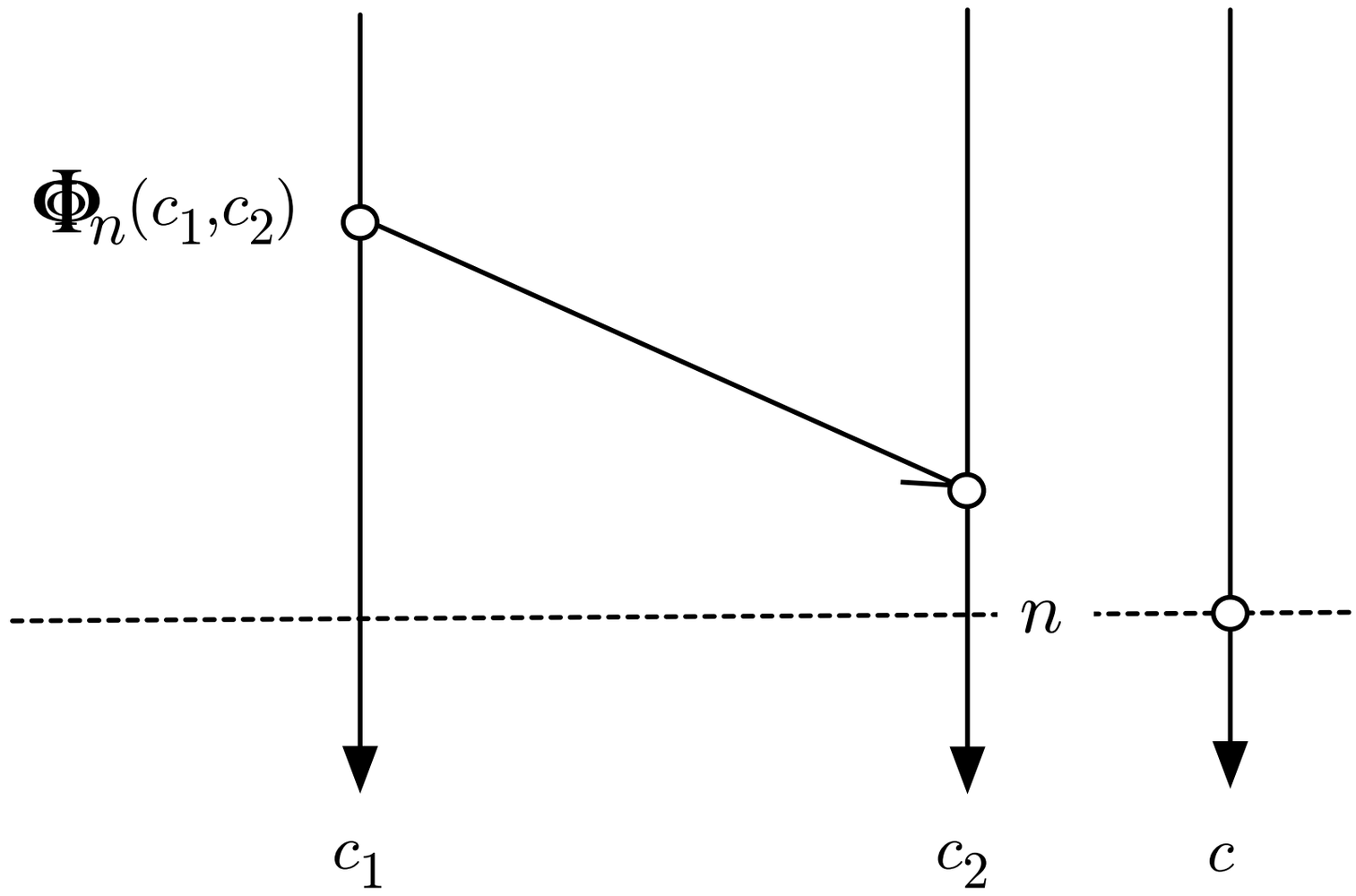
index	core 1	core 2	core 3
1		long computation	
5			SendSignal $s \rightarrow c2$
3		WaitSignal s	
4		computation **using wrong assumptions**	
2	SendSignal $s \rightarrow c2$		
6		WaitSignal s	

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

Proof State

- state of buffers (valid, waiting for DMA, ...)
- active signals
- Φollows ma



- records last instruction on core 1 known to complete before the last instruction on core 2 completing before instruction n

Algorithm

- maintain the state one instruction at a time
  - flag indeterminate states as errors

Proof

- show that any indeterminacy and/or deadlock would have been flagged

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

Memory Lookup

- good
  - scales to higher precision
  - uses other units
- bad
  - doesn’t scale to wider SIMD

Accurate Table Method

- [ ] = round to floating point
- in each interval find

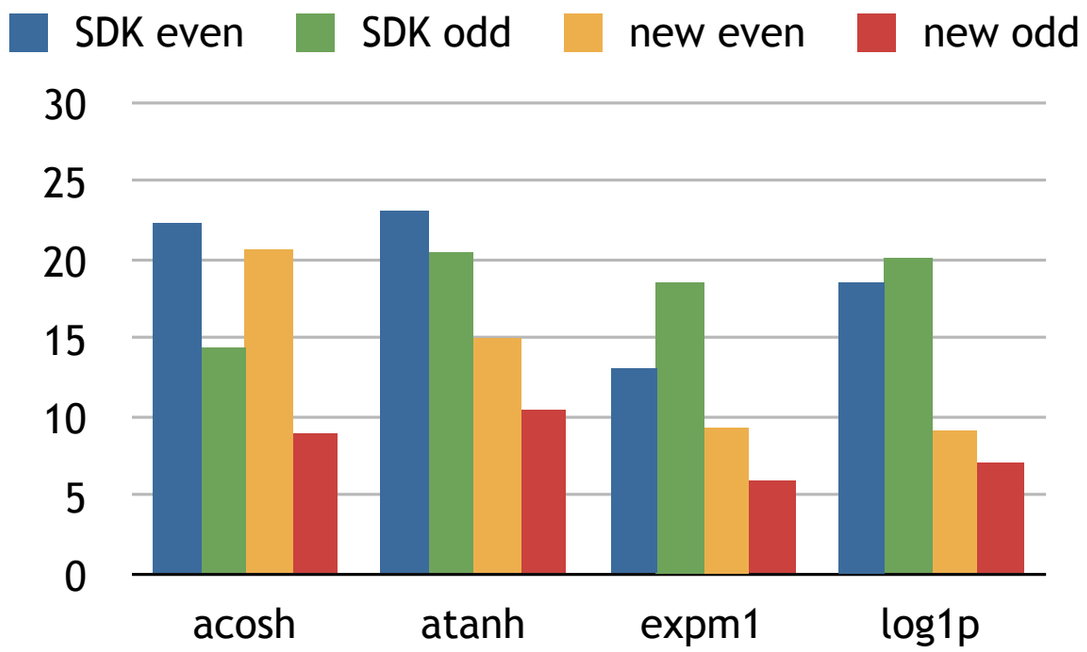
$$c = [c]$$

$$|1/c - [1/c]| \lll \text{ulp}$$

- loose very little precision on range reduction and restoration

Multiplicative Reduction Accurate Table

- unifies AT method
  - log, log1p, …
  - exp, expm1, …
- faster



Better

function	max error (ulps)
exp	1.55
exp2	1.66
expm1	1.80
exp2m1	1.29
log	1.78
log1p	1.79
log21p	1.11
log2	1.00
acosh	2.01
asinh	2.20
atanh	1.46

- close to correctly rounded

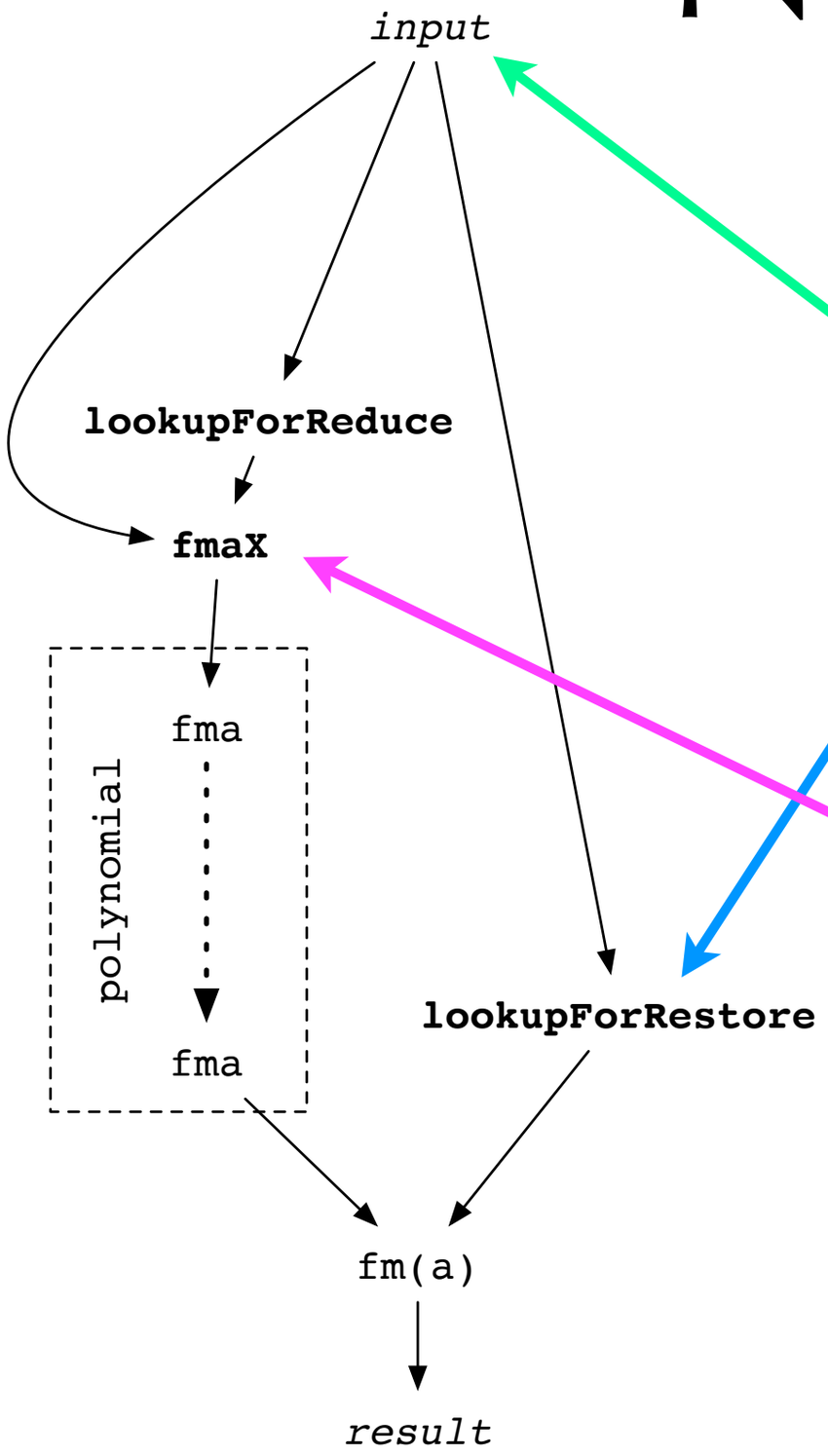
Exceptions

- special case (e.g., log (-1))
- extra computation
- branch
- predication

Problems

- hard to schedule
- exceptions slow and don’t scale

New Instructions



- 2 lookups (LS / odd)
- `lookupForReduce`
- `lookupForRestore`
- `fmaX` (FPU /VPU)
  - 12-bit exponent
  - no subnormals
  - non-standard exceptions

Exceptions

- all handled in-line
- special lookup values

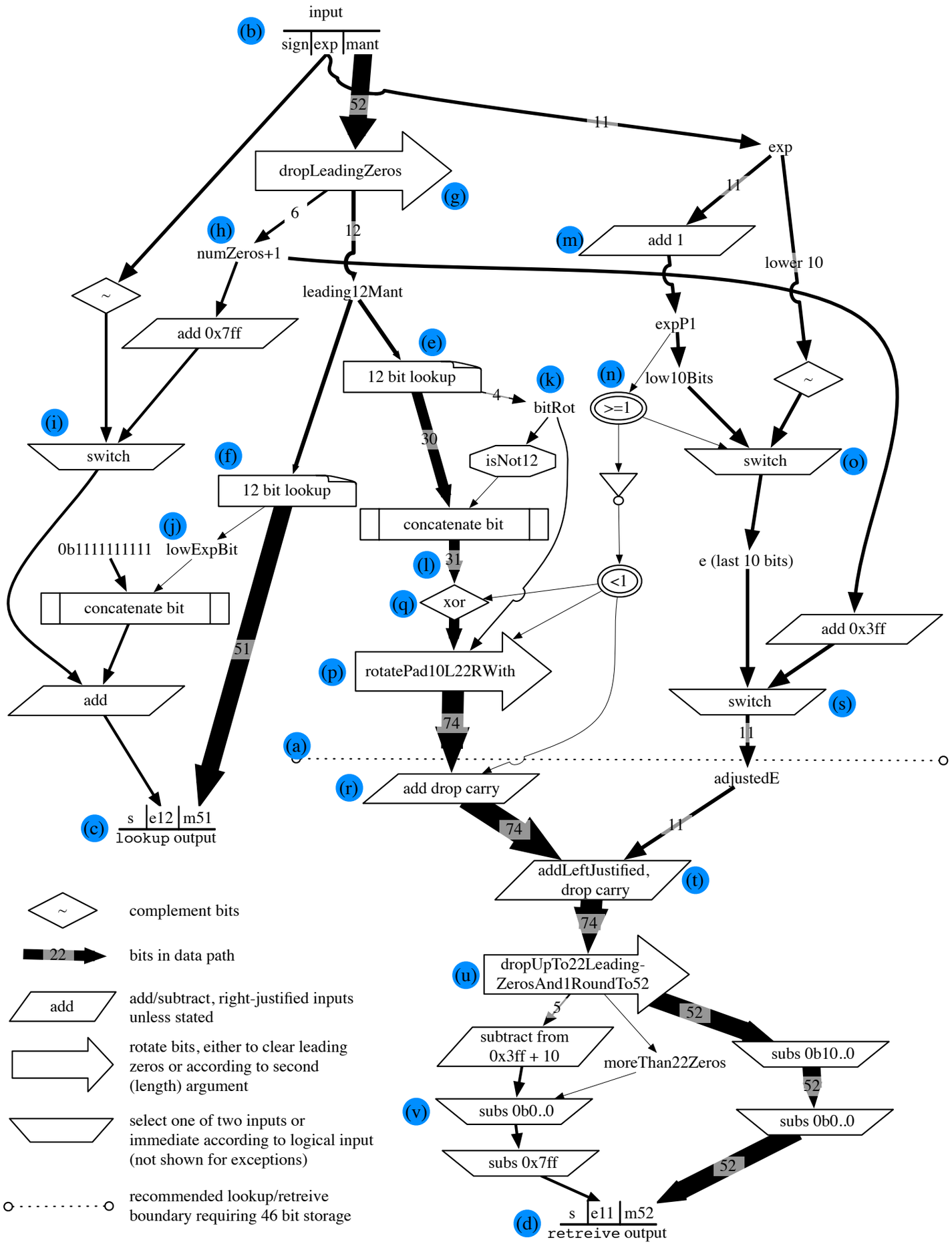
function	normal > 0	subnormal > 0	$+\infty$	$-\infty$	± 0	< 0
recip	$\frac{2^{-e}}{c}$, $\frac{2^{-e}}{c}$	$\frac{2^{-e+52}}{c}$, $(\frac{2^{-e}}{c})_{\text{saturated}}$	0, 0	0, 0	0, $\pm\infty$	$-\frac{2^{-e}}{c}$, $-\frac{2^{-e}}{c}$
sqrt	$\frac{2^{-e}}{c}$, $\frac{2^{e/2}}{c}$	$\frac{2^{-e+52}}{c}$, $\frac{2^{e/2}}{c}$	0, ∞	0, NaN	0, 0	0, NaN
rsqrt	$\frac{2^{-e}}{c}$, $\frac{2^{-e/2}}{c}$	$\frac{2^{-e+52}}{c}$, $\frac{2^{-e/2}}{c}$	0, 0	0, NaN	0, ∞	0, NaN
log2	$\frac{2^{-e}}{c}$, $e + \log_2 c$	$\frac{2^{-e+52}}{c}$, $e-52+\log_2 c$	0, ∞	0, NaN	0, $-\infty$	0, NaN
exp2	c, $2^I \cdot 2^c$	0, 1	0, ∞	NaN, 0	0, 1	c, $2^{-I} \cdot 2^c$

`fmaX` (extended fma)

$+_{\text{ext}}$	finite	$-\infty$	∞	NaN
finite	c	c	c	0
$-\infty$	c	c	0	0
∞	c	0	c	0
NaN	c	c	c	0

$*_{\text{ext}}$	finite	$-\infty$	∞	NaN
± 0	$\pm 0^f$	$\pm 0^f$	$\pm 0^f$	$\pm 0^f$
finite$\neq 0$	c	2	2	2
$-\infty$	$-\infty^f$	$-\infty^f$	$-\infty^f$	$-\infty^f$
∞	∞^f	∞^f	∞^f	∞^f
NaN	NaN^f	NaN^f	NaN^f	NaN^f

- override exceptions
- 1st argument extended
  - 12-bit exponent
  - no subnormals



Bitflow

(log)

table lookup

+

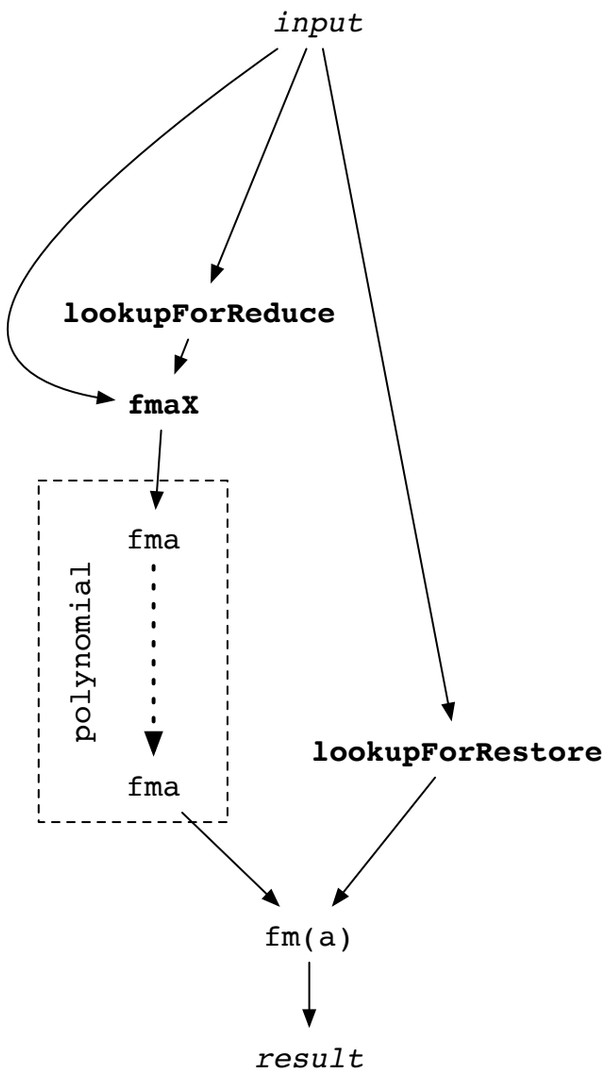
count leading zeros

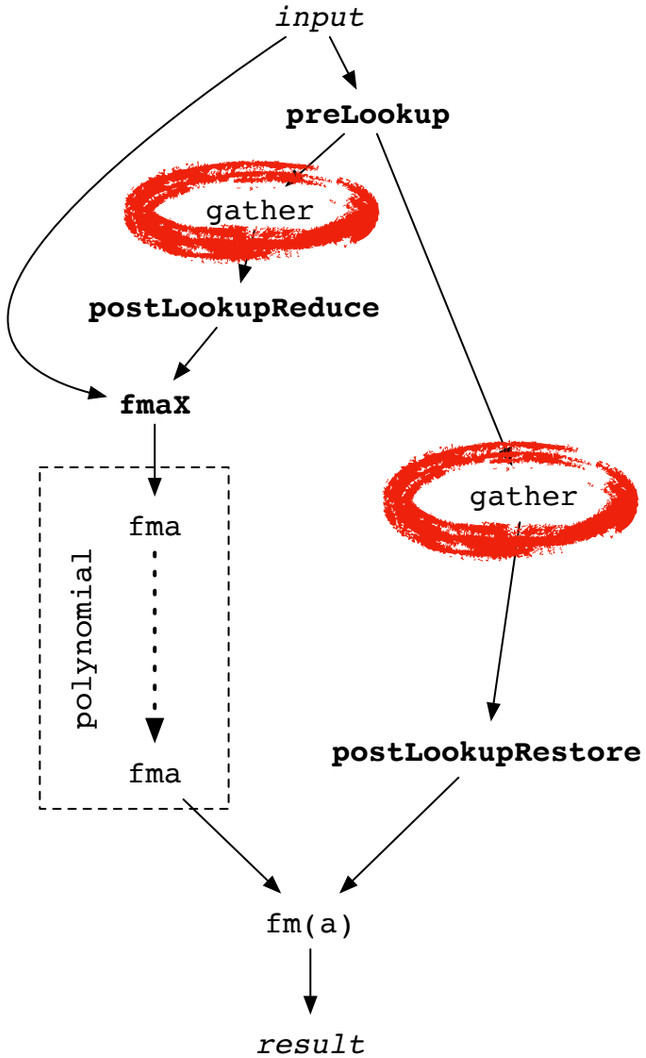
+

12 bit adds

McMaster - 2019 April 8

Add Gather





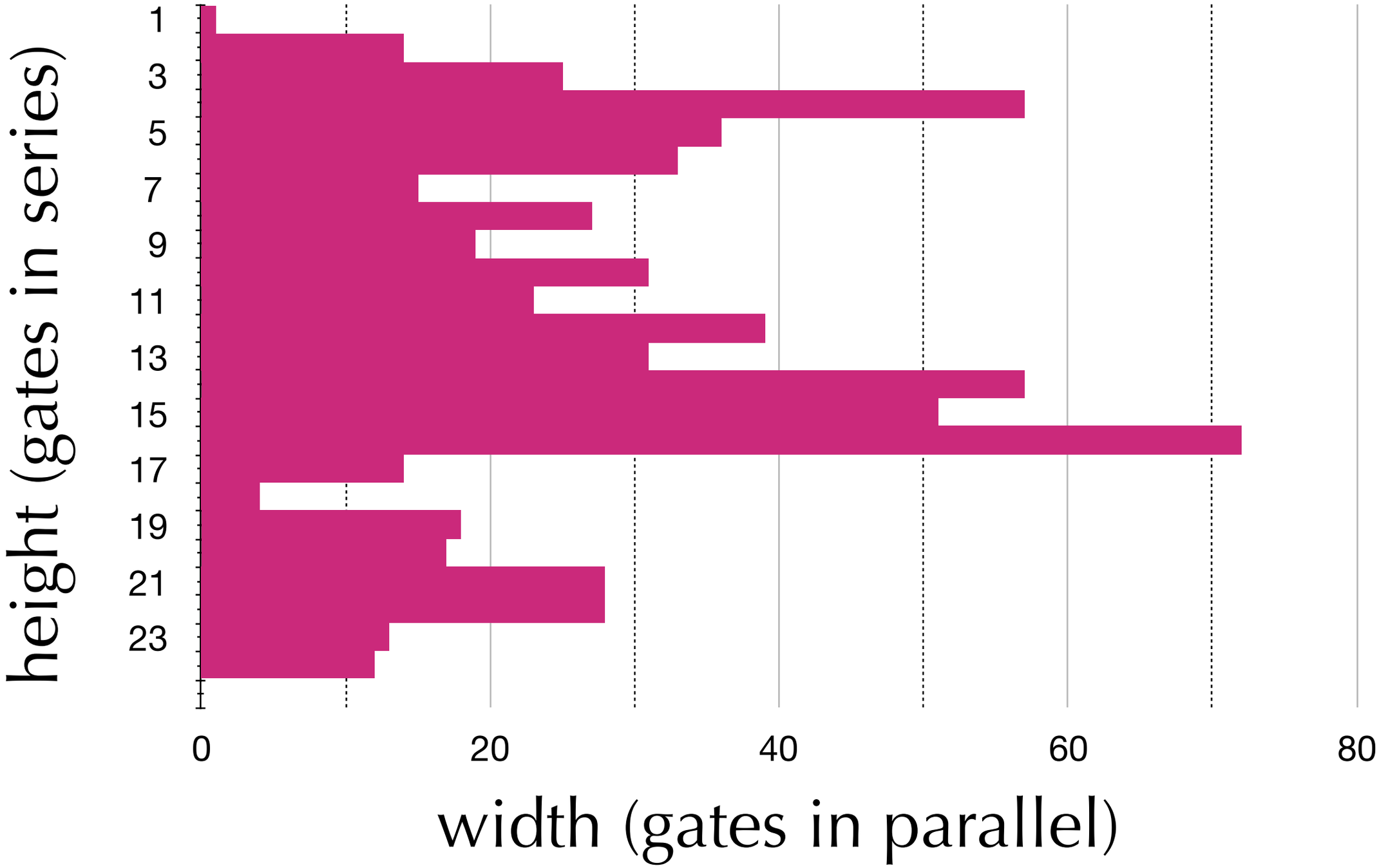
- lots of processors have them
- use values in SIMD slots as indices
- reduces implementation cost
- reduces testing

64 input bits

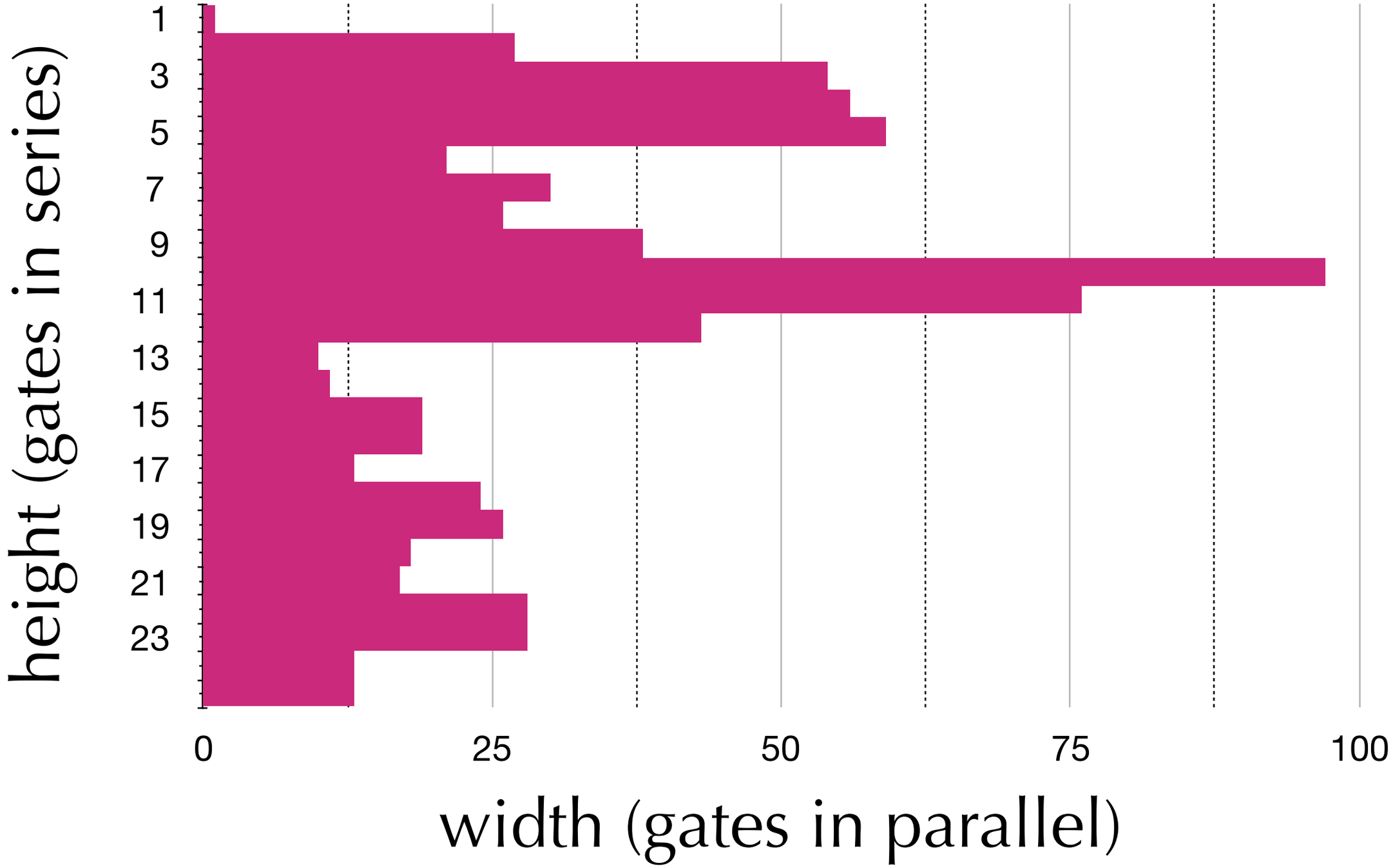
64 bits output

McMaster - 2019 April 8

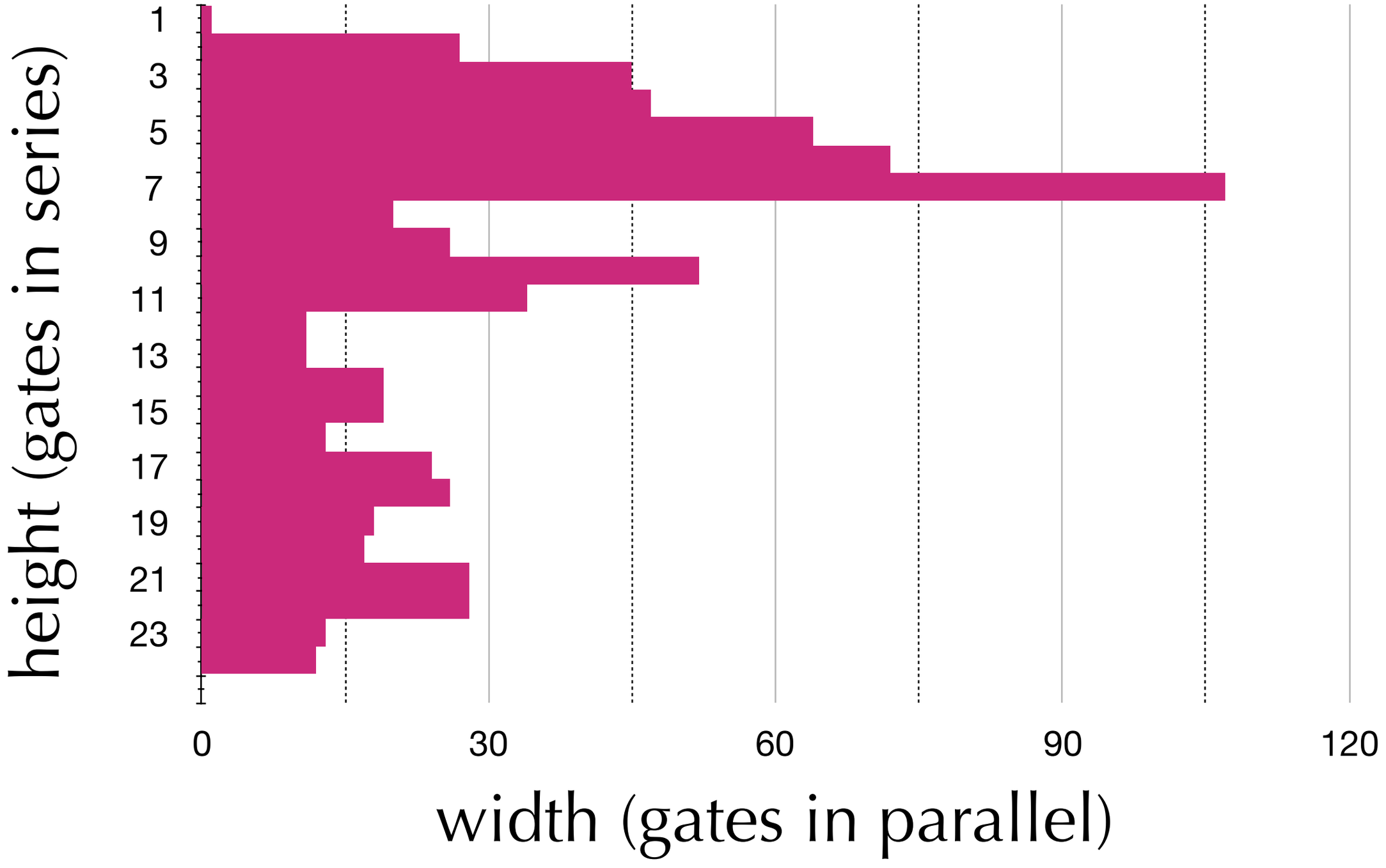
recip Pre


height (gates in series)
1
3
5
7
9
11
13
15
17
19
21
23
0
20
40
60
80
width (gates in parallel)

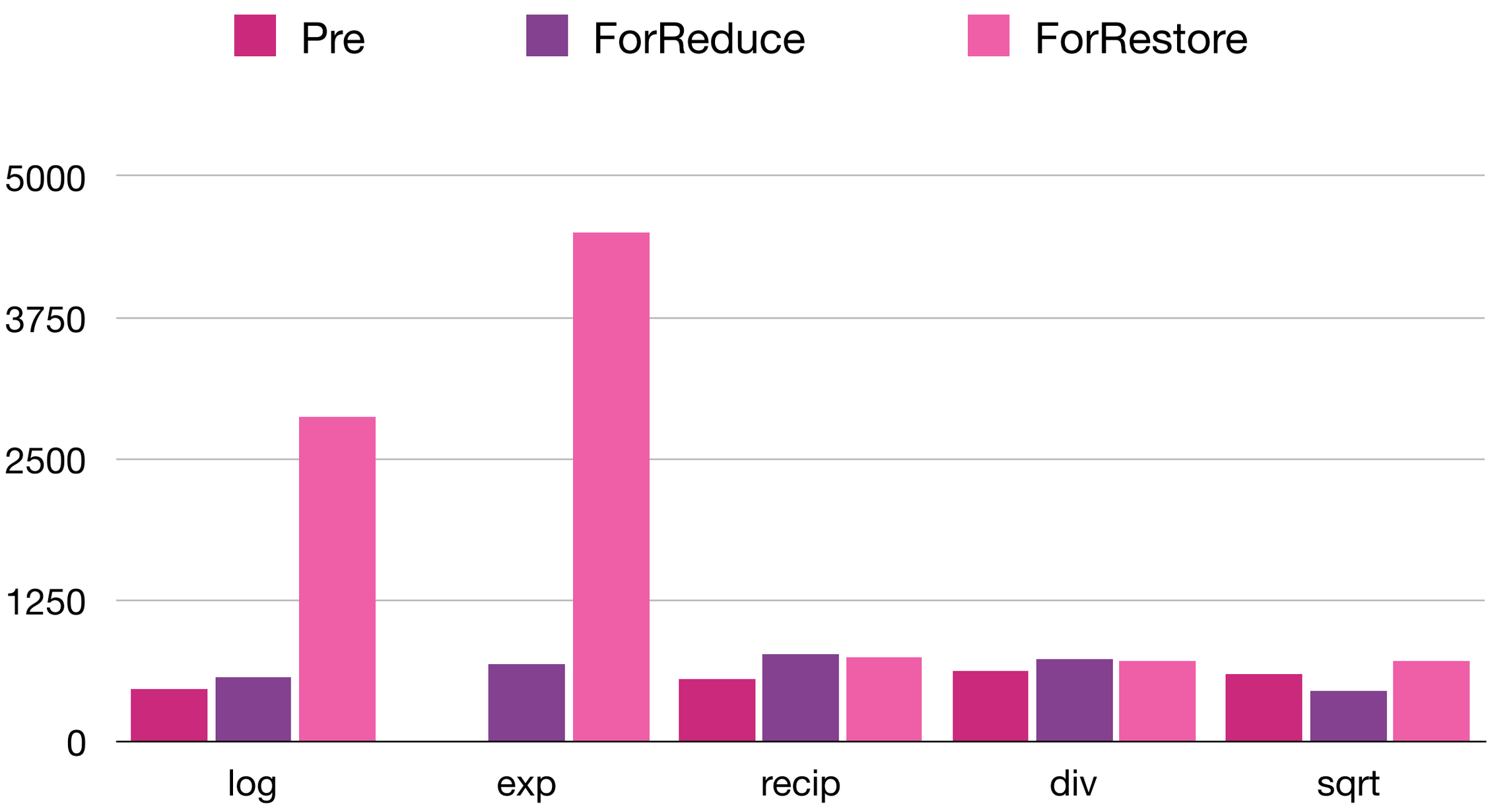
recip Reduce

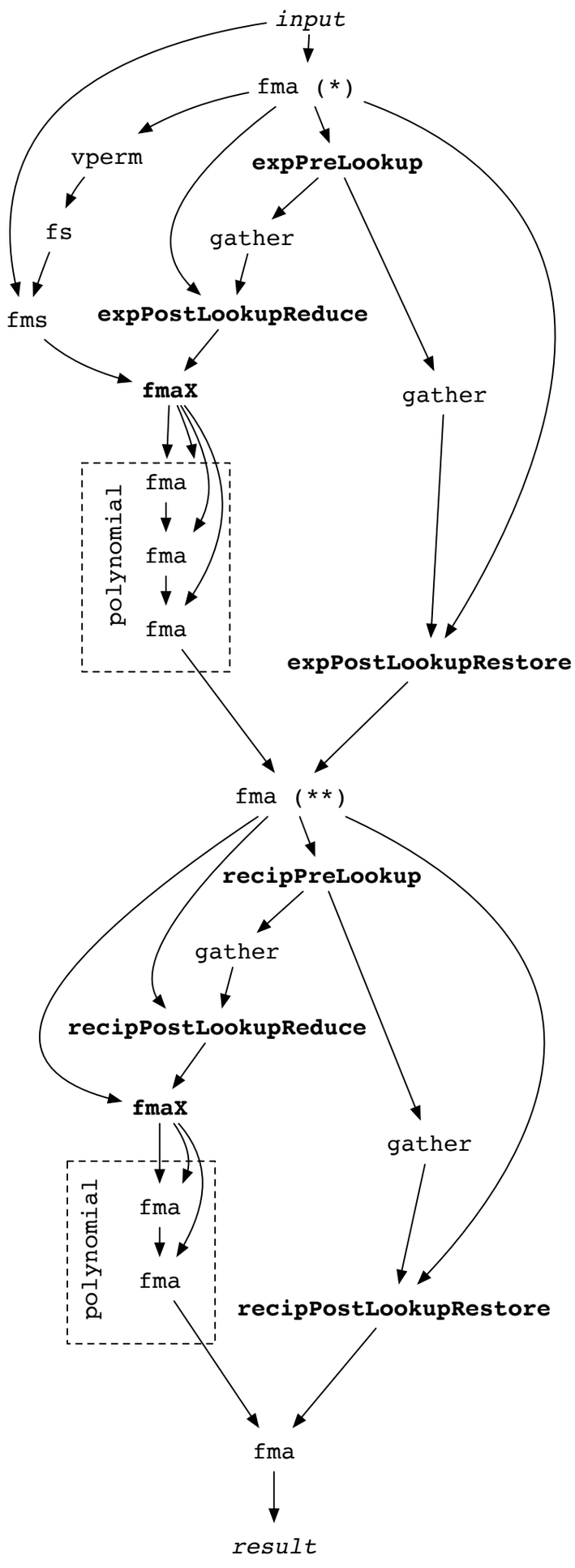


recip Restore



Numbers of Gates





Sigmoid

$$\frac{1}{1 + e^{-x}}$$

- used in ML (learning)
- uses exp + recip
- compiler-discoverable optimizations
  - merge -1 and $\log_2(e)$
  - fm + fa -> fma
  - many times faster

Less than Sum of Parts

- for functions like sigmoid
- faster for all previous reasons
- code is inlinable (no func overhead)

Conclusions

- new instructions
  - much faster
  - not too many gates
  - let's build it!
- context matters
  - software is still written by people
  - understanding their history helps