Reciprocal Square Root

Accelerated using Mixed Hardware and Software Techniques

Orégane Desrentes, Florent de Dinechin, Benoît Dupont de Dinechin

3rd of November, 2025

LMF Laboratoire Méthodes Formelles
citi lab
école normale supérieure paris—saclay
cnrs
université PARIS-SACLAY
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON
UNIVERSITÉ DE LYON
Inria
KALRAY THE POWER OF MORE

Context

Reciprocal (and) Square Root Function(s)

To accelerate ML, we need some elementary functions. We are specifically interested in:

- $\frac{1}{\sqrt{x}}$ for batch normalisation
- $\frac{1}{x}$ for various activation functions (softmax $\frac{\exp^{x_i}}{\sum_k \exp^{x_k}}$, sigmoid $\frac{1}{1+\exp^{-x}}$, swish $\frac{x}{1+\exp^{-x}}$, tanh $\frac{1-\exp^{-2x}}{1+\exp^{-2x}}$,...)
- $\sqrt{x}$ in case anyone invents yet another activation function between now and when the accelerator is produced

Reciprocal (and) Square Root Function(s)

To accelerate ML, we need some elementary functions. We are specifically interested in:

- $\frac{1}{\sqrt{x}}$
- $\frac{1}{x}$
- $\sqrt{x}$

Same family of function. Generally computed in the same units, in a similar way.

Reciprocal (and) Square Root Function(s)

To accelerate ML, we need some elementary functions. We are specifically interested in:

- $\frac{1}{\sqrt{x}}$
- $\frac{1}{x}$
- $\sqrt{x}$

Same family of function. Generally computed in the same units, in a similar way.

Target format

- FP16, typical in ML accelerator, probably precise enough
- FP32, just in case (and handy for other applications).

Generalised Newton-Raphson iterations

Big picture

1. Guess the result.

Generalised Newton-Raphson iterations

Big picture

1. Guess the result.
2. Make the guess a little bit better using maths.

Generalised Newton-Raphson iterations

Big picture

1. Guess the result.
2. Make the guess a little bit better using maths.
3. Repeat step 2 as much as needed.

Generalised Newton-Raphson iterations

Big picture

1. Guess the result.
2. Make the guess a little bit better using maths.
3. Repeat step 2 as much as needed.
4. Once the guess is close enough, apply a formula to obtain Correct Rounding.

Generalised Newton-Raphson iterations

Big picture

1. Guess the result.
2. **Make the guess a little bit better using maths.**
3. **Repeat step 2 as much as needed.**
4. **Once the guess is close enough, apply a formula to obtain Correct Rounding.**

Software: Iterations

Goal: Not spend too much time on step 2 (and 3).

Generalised Newton-Raphson iterations

Big picture

1. Guess the result.
2. **Make the guess a little bit better using maths.**
3. **Repeat step 2 as much as needed.**
4. **Once the guess is close enough, apply a formula to obtain Correct Rounding.**

Software: Iterations

Goal: Not spend too much time on step 2 (and 3).
That is : skip it for FP16, once for FP32.

Generalised Newton-Raphson iterations

Big picture

1. **Guess the result.**
2. Make the guess a little bit better using maths.
3. Repeat step 2 as much as needed.
4. Once the guess is close enough, apply a formula to obtain Correct Rounding.

Software: Iterations

Goal: Not spend too much time on step 2 (and 3).
That is : skip it for FP16, once for FP32.

Hardware: Seed table

We need a very good guess.
Guess is pre-computed, and stored in hardware in a table.

A very good guess is an expensive guess. . .

Idea 1: combine the tables for the 3 functions

A very good guess is an expensive guess. . .

Idea 1: combine the tables for the 3 functions

- $\sqrt{x} \approx x \times \frac{1}{\sqrt{x}}$

A very good guess is an expensive guess...

Idea 1: combine the tables for the 3 functions

- $\sqrt{x} \approx x \times \frac{1}{\sqrt{x}}$ This is actually the standard way of computing $\sqrt{x}$

A very good guess is an expensive guess. . .

Idea 1: combine the tables for the 3 functions

- $\sqrt{x} \approx x \times \frac{1}{\sqrt{x}}$ This is actually the standard way of computing $\sqrt{x}$
- $\frac{1}{x} \approx (\frac{1}{\sqrt{x}})^2$.

A very good guess is an expensive guess. . .

Idea 1: combine the tables for the 3 functions

- $\sqrt{x} \approx x \times \frac{1}{\sqrt{x}}$ This is actually the standard way of computing $\sqrt{x}$
- $\frac{1}{x} \approx (\frac{1}{\sqrt{x}})^2$. Worry about the sign later.

Idea 2: Compress the table as much as possible

A very good guess is an expensive guess. . .

Idea 1: combine the tables for the 3 functions

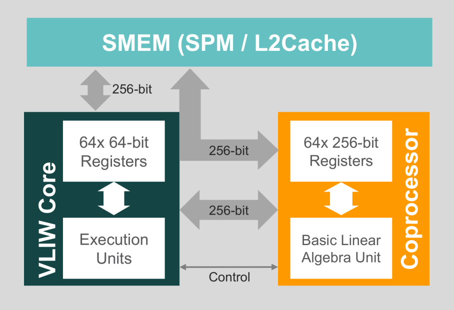
- $\sqrt{x} \approx x \times \frac{1}{\sqrt{x}}$ This is actually the standard way of computing $\sqrt{x}$
- $\frac{1}{x} \approx (\frac{1}{\sqrt{x}})^2$. Worry about the sign later.

Idea 2: Compress the table as much as possible

Adapting table compression and function evaluation techniques

Why not do everything in hardware ?

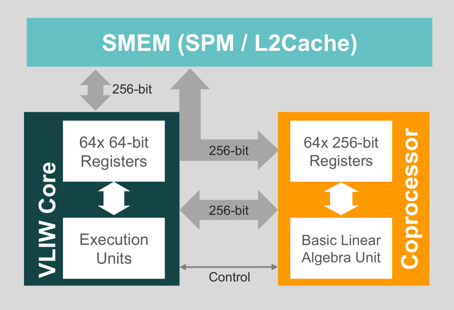
Kalray MPPA



Why not do everything in hardware ?

Kalray MPPA

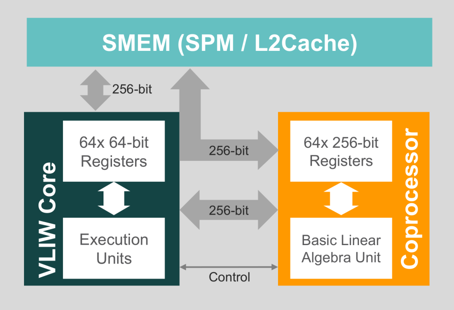
- Core can do either FP32 $\frac{1}{x}$ or FP32 $\sqrt{x}$: fully in hardware, pipelined in 12 cycles.



Why not do everything in hardware ?

Kalray MPPA

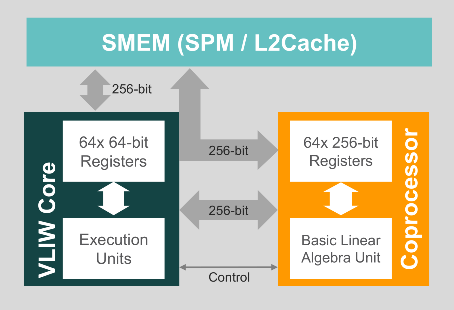
- Core can do either FP32 $\frac{1}{x}$ or FP32 $\sqrt{x}$: fully in hardware, pipelined in 12 cycles.
- Co-Processor has a bunch of **S**ingle **I**nstruction **M**ultiple **D**ata FP16 and FP32 **F**used **M**ultiply **A**dd (8 now, 16 in next generation MPPA)



Why not do everything in hardware ?

Kalray MPPA

- Core can do either FP32 $\frac{1}{x}$ or FP32 $\sqrt{x}$: fully in hardware, pipelined in 12 cycles.
- Co-Processor has a bunch of **S**ingle **I**nstruction **M**ultiple **D**ata FP16 and FP32 **F**used **M**ultiply **A**dd (8 now, 16 in next generation MPPA)



→ Use the full power of the FMAs to compute the iterations.
Decent throughput, small hardware footprint.

The expected efficiency

If there are 16 seed table operators, and 16 SIMD FMAs :

The expected efficiency

If there are 16 seed table operators, and 16 SIMD FMAs :

Fun	Measure	FP32			FP16	
		Core (CR)	Algo (CR)	Algo (FR)	Algo (CR)	Algo (err $<$ 1.4ulp)
$\frac{1}{x}$	Latency	12	21	13	13	5
	Throughput	1	3.2	13	5.3	16
$\sqrt{x}$	Latency	12	22	14	13	5
	Throughput	1	2.3	4	4	16
$\frac{1}{\sqrt{x}}$	Latency	24	$\emptyset$	14	1	1 (CR)
	Throughput	0.5	$\emptyset$	4	16	16 (CR)

The expected efficiency

If there are 16 seed table operators, and 16 SIMD FMAs :

Fun	Measure	FP32			FP16	
		Core (CR)	Algo (CR)	Algo (FR)	Algo (CR)	Algo (err $<$ 1.4ulp)
$\frac{1}{x}$	Latency	12	21	13	13	5
	Throughput	1	3.2	13	5.3	16
$\sqrt{x}$	Latency	12	22	14	13	5
	Throughput	1	2.3	4	4	16
$\frac{1}{\sqrt{x}}$	Latency	24	$\emptyset$	14	1	1 (CR)
	Throughput	0.5	$\emptyset$	4	16	16 (CR)

→ Always better throughput, and very fast FP16

How to build this Seed table ?

A guess for $\frac{1}{\sqrt{x}}$ for every FP32 number?

2^{31} guesses of size 31

A guess for $\frac{1}{\sqrt{x}}$ for every FP32 number?

2^{31} guesses of size 31

Argument Reduction for $\frac{1}{\sqrt{x}}$

$$\frac{1}{\sqrt{a}} = \frac{1}{\sqrt{2^e \times 1.F}} = \begin{cases} 2^{-k} \times \frac{1}{\sqrt{1.F}} & \text{if } e = 2k, k \in \mathbb{Z} \\ 2^{-k} \times \frac{1}{\sqrt{2 \times 1.F}} & \text{if } e = 2k+1, k \in \mathbb{Z} \end{cases}$$

A guess for $\frac{1}{\sqrt{x}}$ for every FP32 number?

2^{31} guesses of size 31

Argument Reduction for $\frac{1}{\sqrt{x}}$

$$\frac{1}{\sqrt{a}} = \frac{1}{\sqrt{2^e \times 1.F}} = \begin{cases} 2^{-k} \times \frac{1}{\sqrt{1.F}} & \text{if } e = 2k, k \in \mathbb{Z} \\ 2^{-k} \times \frac{1}{\sqrt{2 \times 1.F}} & \text{if } e = 2k+1, k \in \mathbb{Z} \end{cases}$$

We just need fixed-point guesses for $\frac{1}{\sqrt{1.F}}$ and $\frac{1}{\sqrt{2\times 1.F}}$, for $F \in \mathrm{uFix}(-1, -23)$.

A guess for $\frac{1}{\sqrt{1.F}}$ and $\frac{1}{\sqrt{2\times 1.F}}$, for $F \in \mathrm{uFix}(-1,-23)$

2×2^{23} guesses of size 23

Argument Reduction for $\frac{1}{\sqrt{x}}$

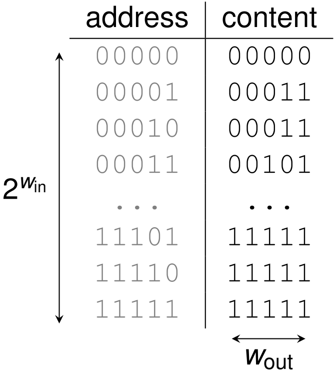
$$\frac{1}{\sqrt{a}} = \frac{1}{\sqrt{2^e \times 1.F}} = \begin{cases} 2^{-k} \times \frac{1}{\sqrt{1.F}} & \text{if } e = 2k, k \in \mathbb{Z} \\ 2^{-k} \times \frac{1}{\sqrt{2\times 1.F}} & \text{if } e = 2k+1, k \in \mathbb{Z} \end{cases}$$

We just need fixed-point guesses for $\frac{1}{\sqrt{1.F}}$ and $\frac{1}{\sqrt{2\times 1.F}}$, for $F \in \mathrm{uFix}(-1,-23)$.

A guess for $\frac{1}{\sqrt{1.F}}$ and $\frac{1}{\sqrt{2\times 1.F}}$, for $F \in \mathrm{uFix}(-1, -23)$

2×2^{23} guesses of size 23

What size of guess?



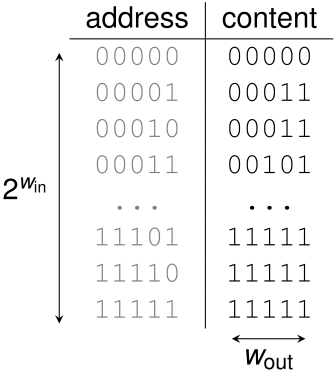
address	content
00000	00000
00001	00011
00010	00011
00011	00101
...	...
11101	11111
11110	11111
11111	11111

A guess for $\frac{1}{\sqrt{1.F}}$ and $\frac{1}{\sqrt{2\times 1.F}}$, for $F \in \mathrm{uFix}(-1, -23)$

2×2^{23} guesses of size 23

What size of guess?

- The seed is not the exact result : $w_{\mathrm{out}} < 24$



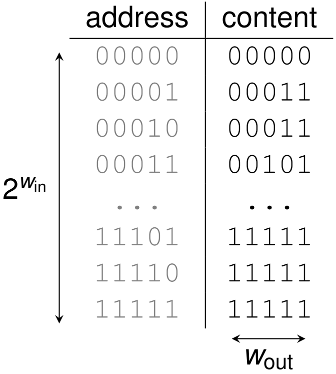
address	content
00000	00000
00001	00011
00010	00011
00011	00101
. . .	. . .
11101	11111
11110	11111
11111	11111

A guess for $\frac{1}{\sqrt{1.F}}$ and $\frac{1}{\sqrt{2\times 1.F}}$, for $F \in \mathrm{uFix}(-1, -23)$

2×2^{23} guesses of size 23

What size of guess?

- The seed is not the exact result : $w_{\text{out}} < 24$
- Multiple close numbers can have the same seed : $w_{\text{in}} < 24$



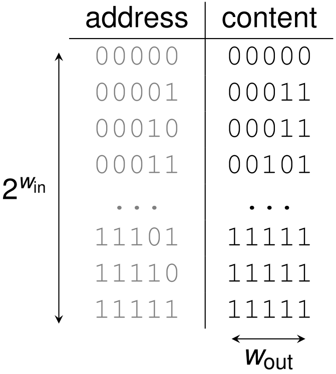
address	content
00000	00000
00001	00011
00010	00011
00011	00101
...	...
11101	11111
11110	11111
11111	11111

A guess for $\frac{1}{\sqrt{1.F}}$ and $\frac{1}{\sqrt{2\times 1.F}}$, for $F \in \mathrm{uFix}(-1, -w_{\mathrm{in}})$

$2 \times 2^{w_{\mathrm{in}}}$ guesses of size w_{out}

What size of guess?

- The seed is not the exact result : $w_{\mathrm{out}} < 24$
- Multiple close numbers can have the same seed : $w_{\mathrm{in}} < 24$



address	content
00000	00000
00001	00011
00010	00011
00011	00101
...	...
11101	11111
11110	11111
11111	11111

A guess for $\frac{1}{\sqrt{1.F}}$ and $\frac{1}{\sqrt{2\times 1.F}}$, for $F \in \mathrm{uFix}(-1, -w_{\mathrm{in}})$

$2 \times 2^{w_{\mathrm{in}}}$ guesses of size w_{out}

What size of guess?

- The seed is not the exact result : $w_{\mathrm{out}} < 24$
- Multiple close numbers can have the same seed : $w_{\mathrm{in}} < 24$

How do we fill the table?

For each table input:

address	content
00000	00000
00001	00011
00010	00011
00011	00101
...	...
11101	11111
11110	11111
11111	11111

$2^{w_{\mathrm{in}}}$

w_{out}

A guess for $\frac{1}{\sqrt{1.F}}$ and $\frac{1}{\sqrt{2\times 1.F}}$, for $F \in \mathrm{uFix}(-1, -w_{\mathrm{in}})$

$2 \times 2^{w_{\mathrm{in}}}$ guesses of size w_{out}

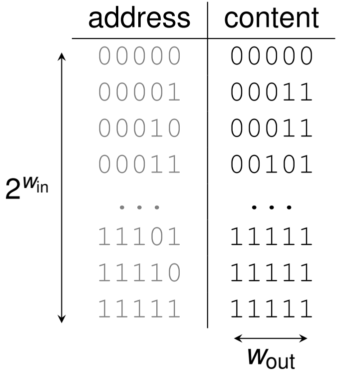
What size of guess?

- The seed is not the exact result : $w_{\mathrm{out}} < 24$
- Multiple close numbers can have the same seed : $w_{\mathrm{in}} < 24$

How do we fill the table?

For each table input:

1. Guess a seed



address	content
00000	00000
00001	00011
00010	00011
00011	00101
. . .	. . .
11101	11111
11110	11111
11111	11111

A guess for $\frac{1}{\sqrt{1.F}}$ and $\frac{1}{\sqrt{2\times 1.F}}$, for $F \in \mathrm{uFix}(-1, -w_{\mathrm{in}})$

$2 \times 2^{w_{\mathrm{in}}}$ guesses of size w_{out}

What size of guess?

- The seed is not the exact result : $w_{\mathrm{out}} < 24$
- Multiple close numbers can have the same seed : $w_{\mathrm{in}} < 24$

How do we fill the table?

For each table input:

1. Guess a seed
2. For every FP32 and FP16 in $[1, 4)$ that has this table input,

address	content
00000	00000
00001	00011
00010	00011
00011	00101
. . .	. . .
11101	11111
11110	11111
11111	11111

$2^{w_{\mathrm{in}}}$

w_{out}

A guess for $\frac{1}{\sqrt{1.F}}$ and $\frac{1}{\sqrt{2\times 1.F}}$, for $F \in \mathrm{uFix}(-1, -w_{\mathrm{in}})$

$2 \times 2^{w_{\mathrm{in}}}$ guesses of size w_{out}

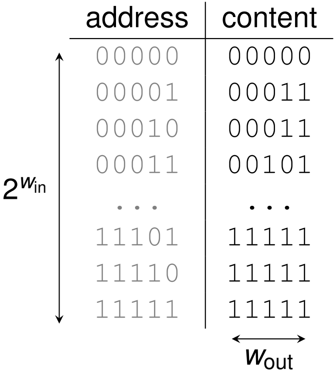
What size of guess?

- The seed is not the exact result : $w_{\mathrm{out}} < 24$
- Multiple close numbers can have the same seed : $w_{\mathrm{in}} < 24$

How do we fill the table?

For each table input:

1. Guess a seed
2. For every FP32 and FP16 in $[1, 4)$ that has this table input, test all the software algos with that seed



address	content
00000	00000
00001	00011
00010	00011
00011	00101
...	...
11101	11111
11110	11111
11111	11111

A guess for $\frac{1}{\sqrt{1.F}}$ and $\frac{1}{\sqrt{2\times 1.F}}$, for $F \in \mathrm{uFix}(-1, -w_{\mathrm{in}})$

$2 \times 2^{w_{\mathrm{in}}}$ guesses of size w_{out}

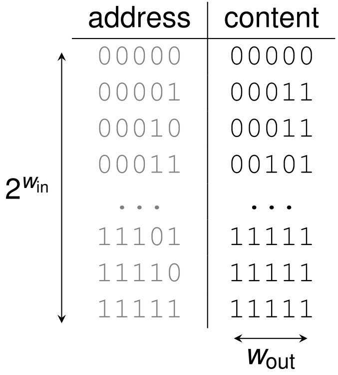
What size of guess?

- The seed is not the exact result : $w_{\mathrm{out}} < 24$
- Multiple close numbers can have the same seed : $w_{\mathrm{in}} < 24$

How do we fill the table?

For each table input:

1. Guess a seed
2. For every FP32 and FP16 in $[1, 4)$ that has this table input, test all the software algos with that seed
3. If it does not work, try another seed.



address	content
00000	00000
00001	00011
00010	00011
00011	00101
...	...
11101	11111
11110	11111
11111	11111

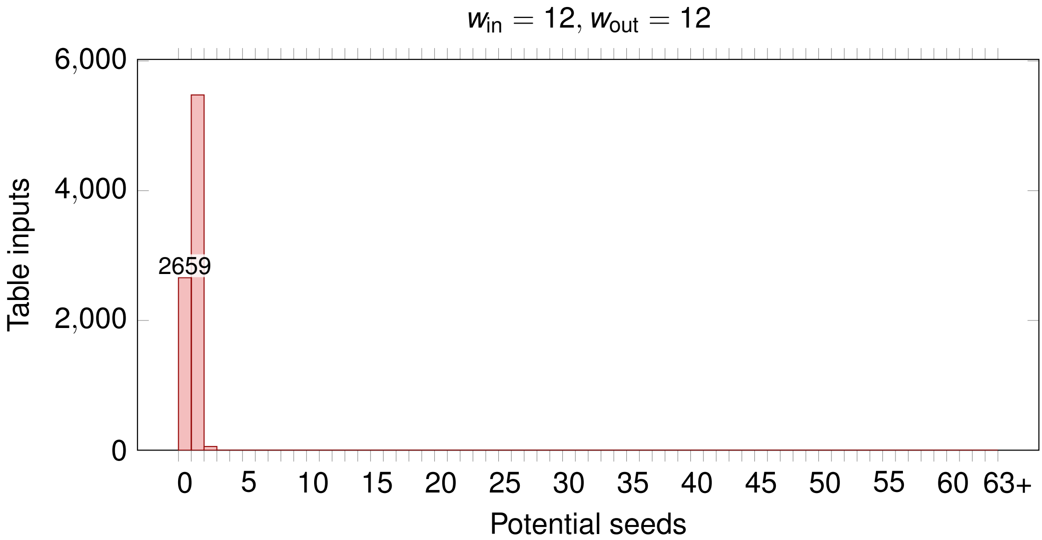
Compute the parameters the seed table

$2 \times 2^{w_{\text{in}}}$ guesses of size w_{out}

Theory says $w_{\text{in}} = w_{\text{out}} = 12$ should work.

Compute the parameters the seed table

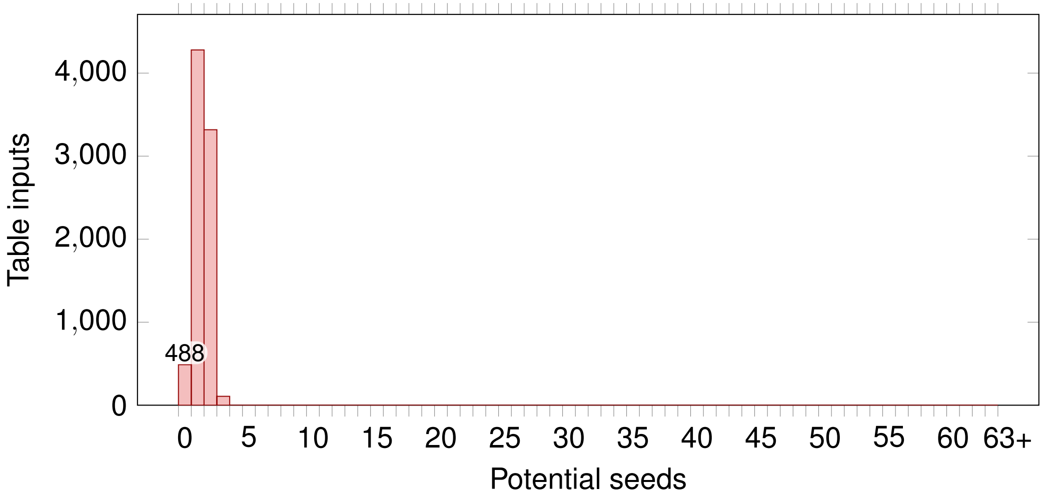
$2 \times 2^{w_{\text{in}}}$ guesses of size w_{out}



Compute the parameters the seed table

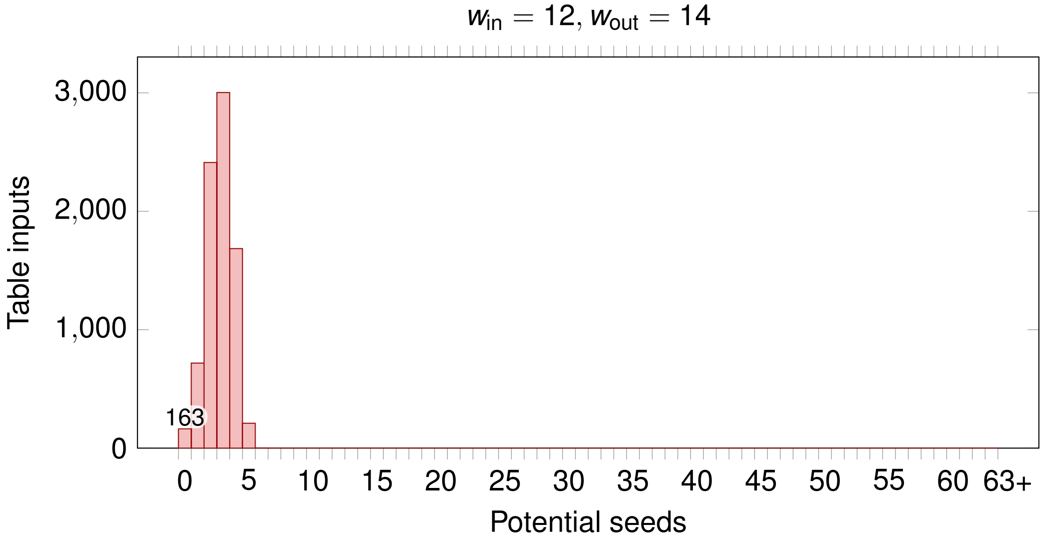
$2 \times 2^{w_{\text{in}}}$ guesses of size w_{out}

$w_{\text{in}} = 12, w_{\text{out}} = 13$



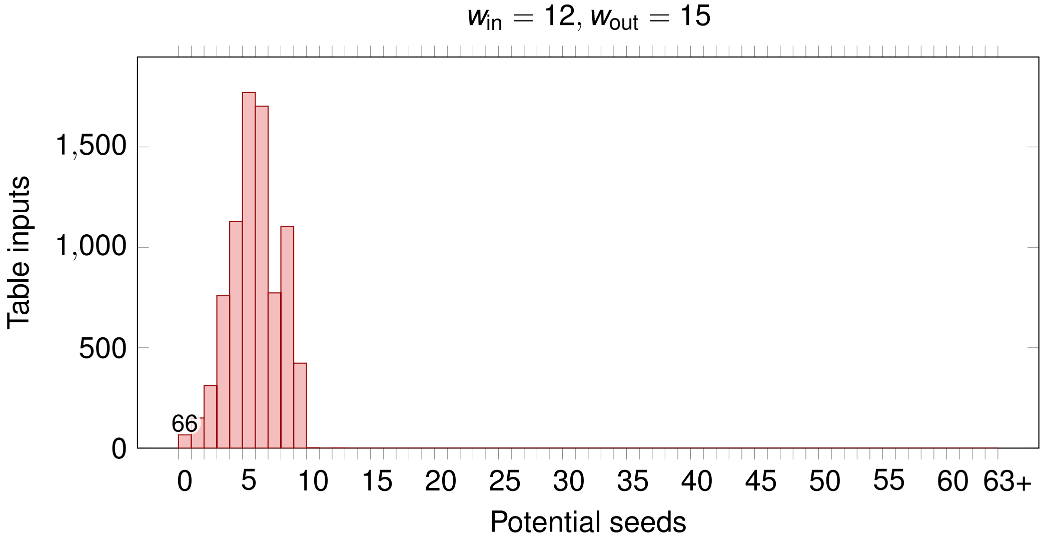
Compute the parameters the seed table

$2 \times 2^{w_{\text{in}}}$ guesses of size w_{out}



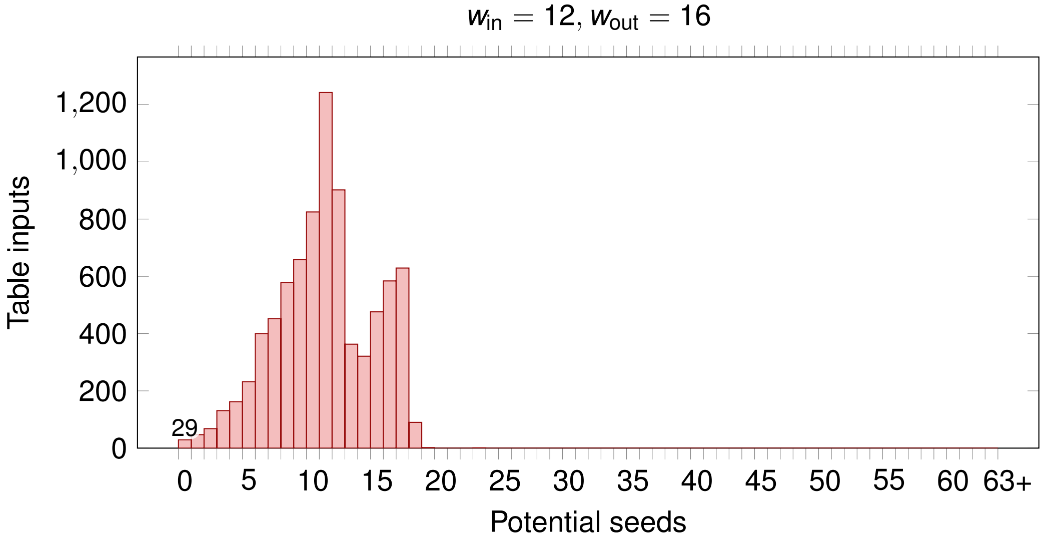
Compute the parameters the seed table

$2 \times 2^{w_{\text{in}}}$ guesses of size w_{out}



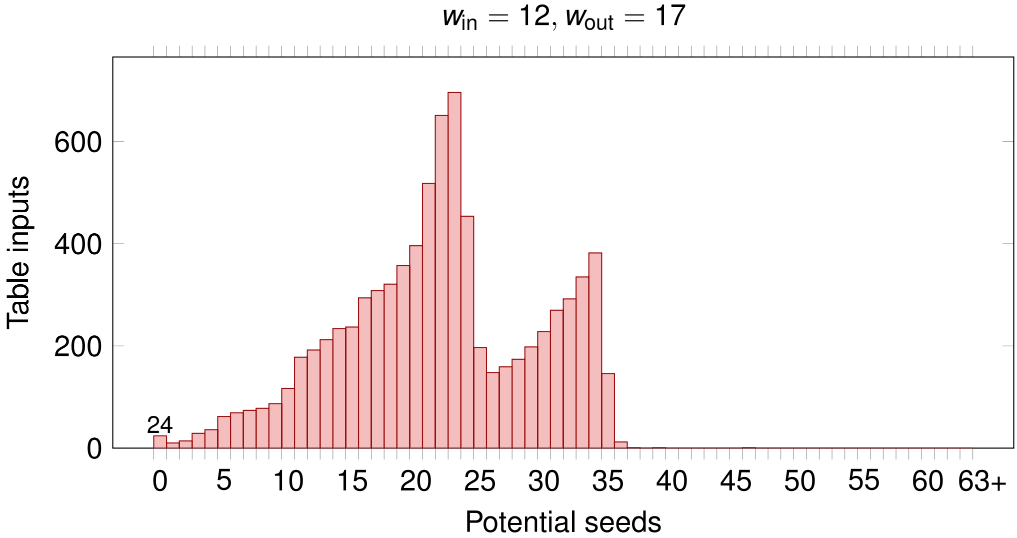
Compute the parameters the seed table

$2 \times 2^{w_{\text{in}}}$ guesses of size w_{out}



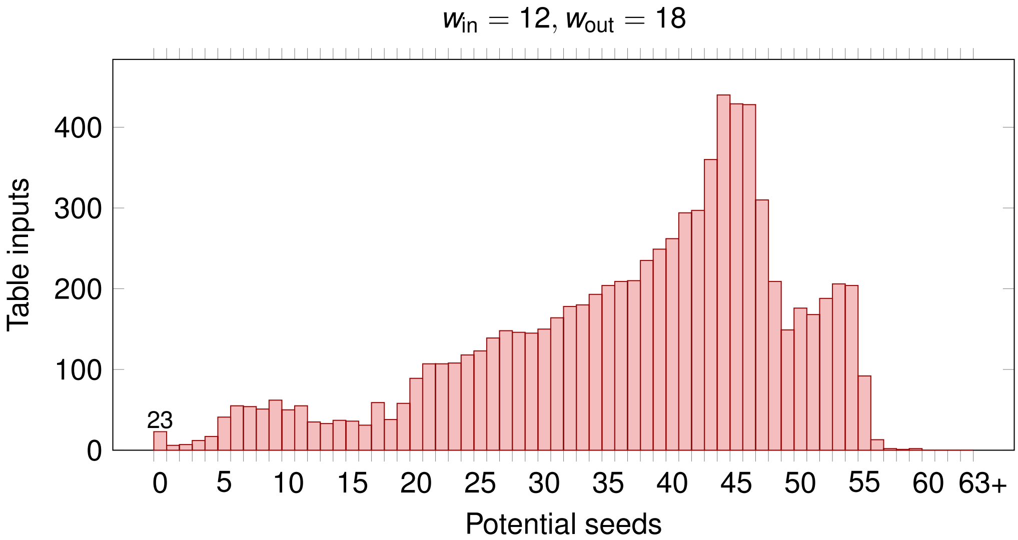
Compute the parameters the seed table

$2 \times 2^{w_{\text{in}}}$ guesses of size w_{out}



Compute the parameters the seed table

$2 \times 2^{w_{\text{in}}}$ guesses of size w_{out}



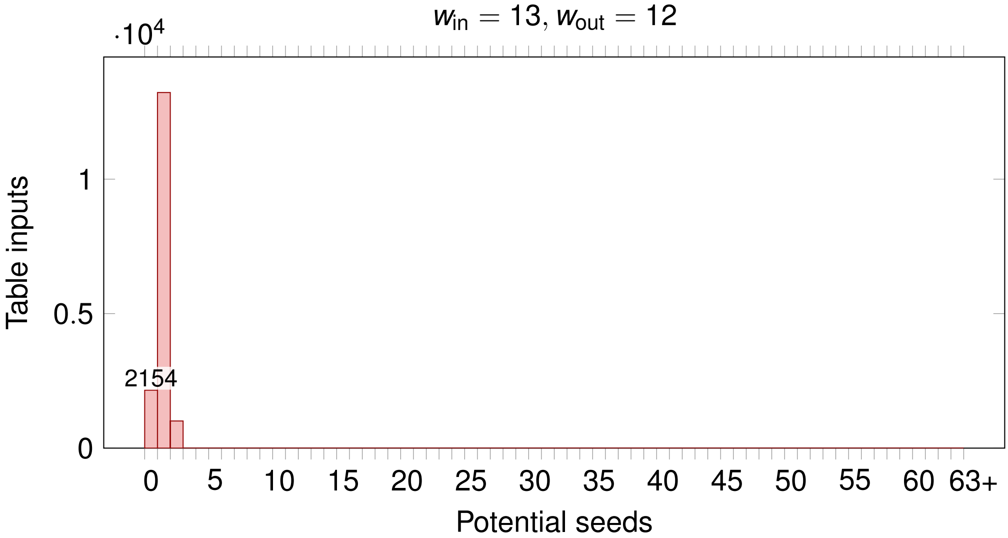
Compute the parameters the seed table

$2 \times 2^{w_{\text{in}}}$ guesses of size w_{out}

Something isn’t working…

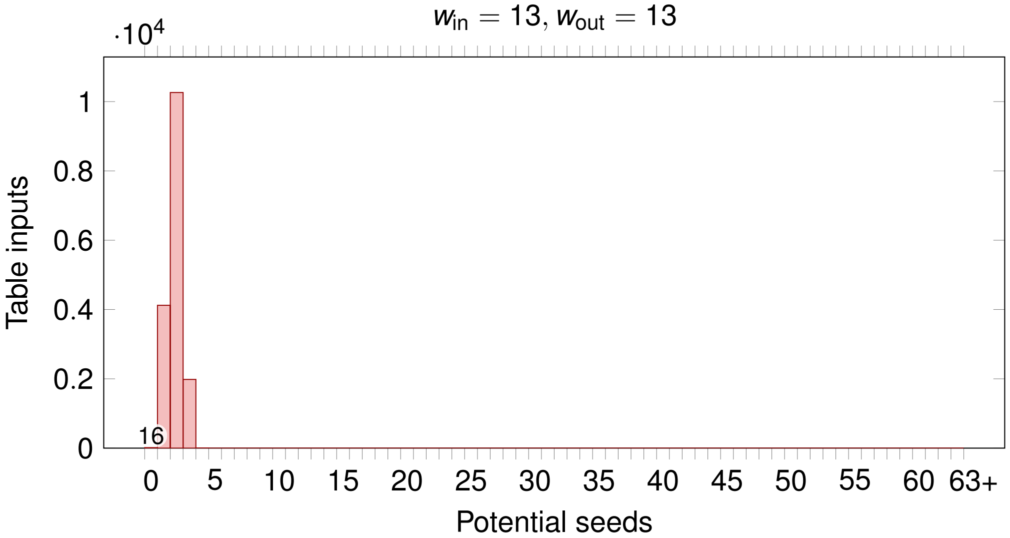
Compute the parameters the seed table

$2 \times 2^{w_{\text{in}}}$ guesses of size w_{out}



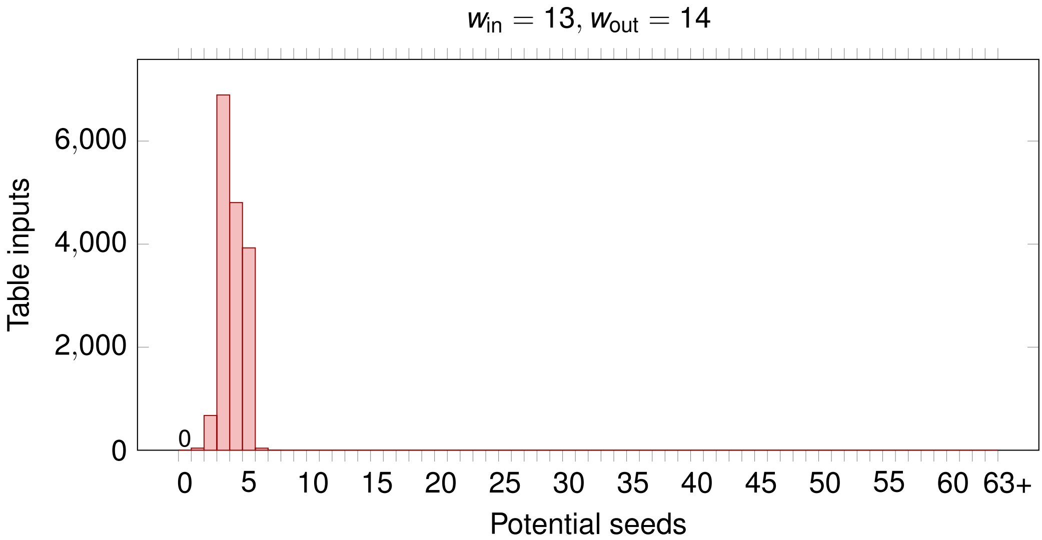
Compute the parameters the seed table

$2 \times 2^{w_{in}}$ guesses of size w_{out}



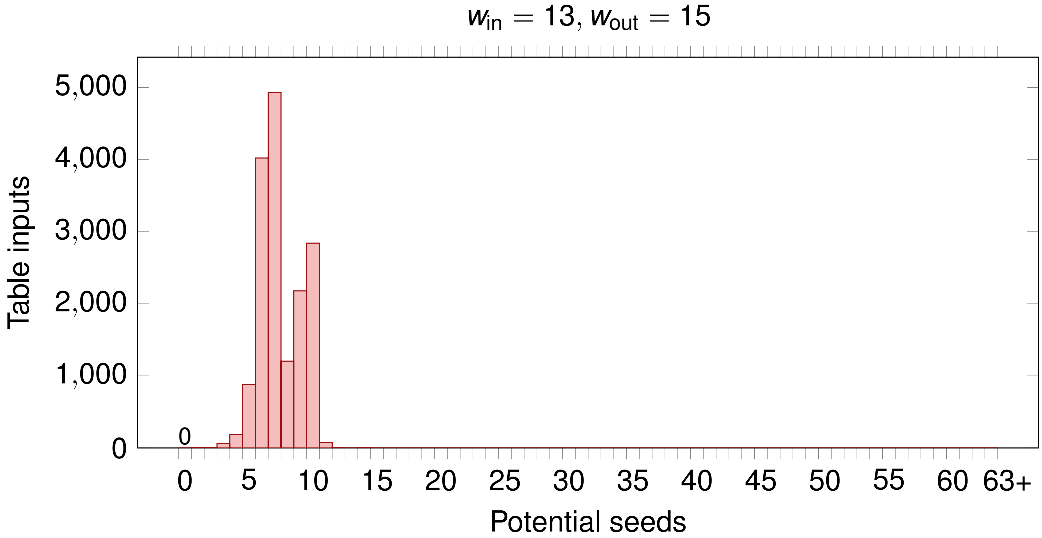
Compute the parameters the seed table

$2 \times 2^{w_{\text{in}}}$ guesses of size w_{out}



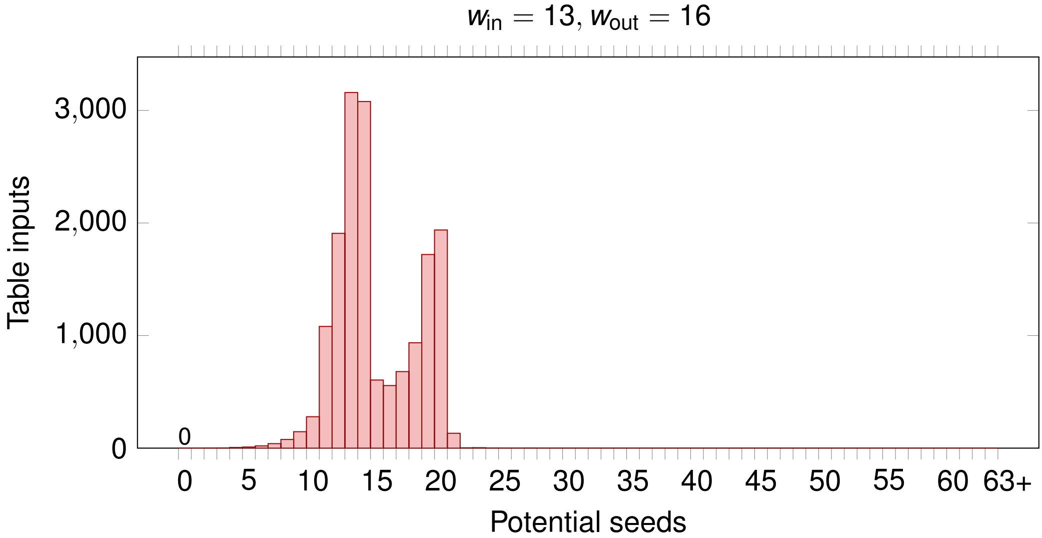
Compute the parameters the seed table

$2 \times 2^{w_{\text{in}}}$ guesses of size w_{out}



Compute the parameters the seed table

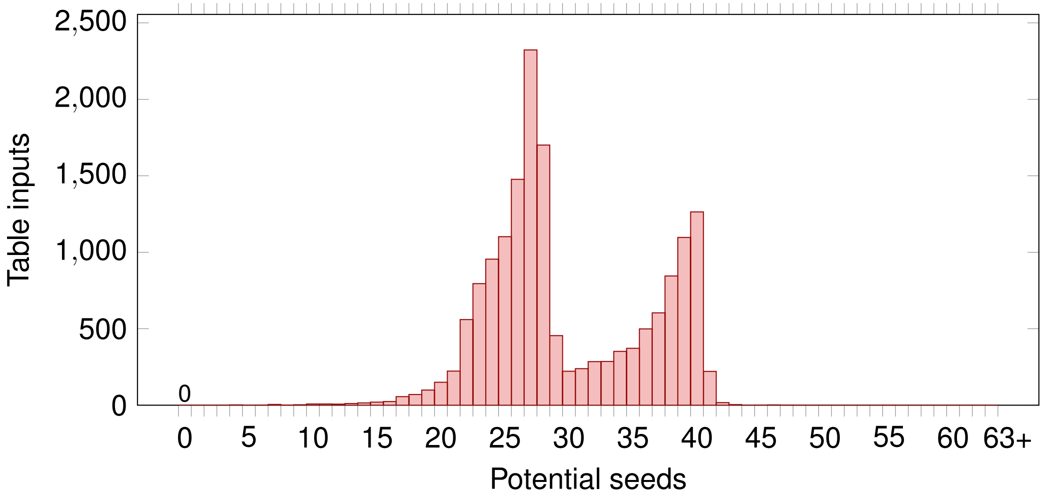
$2 \times 2^{w_{\text{in}}}$ guesses of size w_{out}



Compute the parameters the seed table

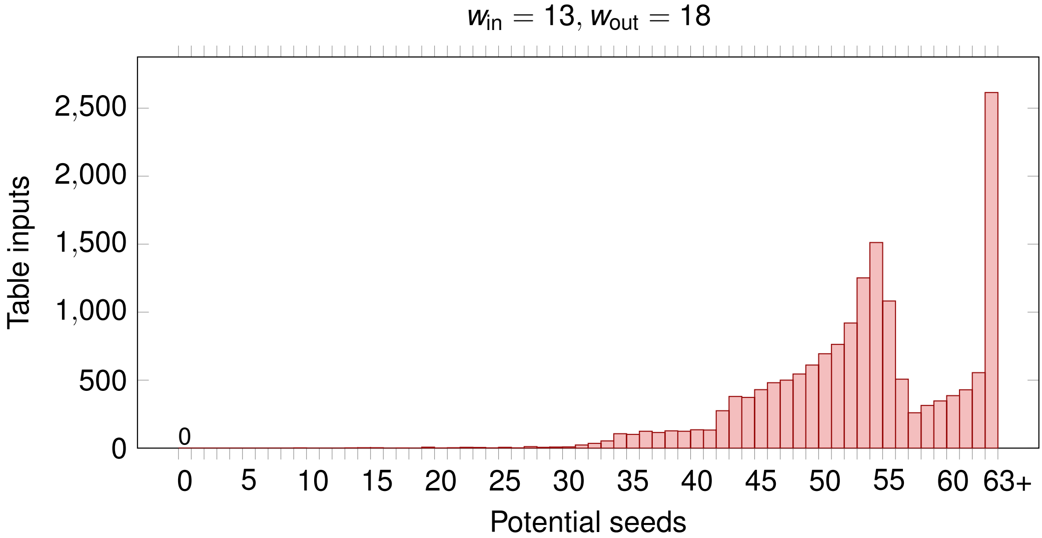
$2 \times 2^{w_{\text{in}}}$ guesses of size w_{out}

$w_{\text{in}} = 13, w_{\text{out}} = 17$



Compute the parameters the seed table

$2 \times 2^{w_{\text{in}}}$ guesses of size w_{out}



Good news: the table exists!

Table size

$w_{\text{in}} = 13$, $w_{\text{out}} = 14$ is a seed table that stores $2 \times 2^{13} \times 14 = 229\,376$ bit

Good news: the table exists!

Table size

$w_{\text{in}} = 13$, $w_{\text{out}} = 14$ is a seed table that stores $2 \times 2^{13} \times 14 = 229\,376$ bit
As suspected, this is quite large and we need a way to compress this table

Good news: the table exists!

Table size

$w_{\text{in}} = 13$, $w_{\text{out}} = 14$ is a seed table that stores $2 \times 2^{13} \times 14 = 229\,376$ bit
As suspected, this is quite large and we need a way to compress this table

There are multiple possible seeds for each input

One combination might compress better than another.

Good news: the table exists!

Table size

$w_{\text{in}} = 13$, $w_{\text{out}} = 14$ is a seed table that stores $2 \times 2^{13} \times 14 = 229\,376$ bit
As suspected, this is quite large and we need a way to compress this table

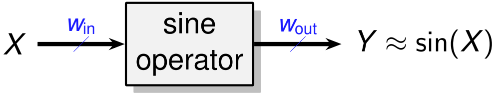
There are multiple possible seeds for each input

One combination might compress better than another.
Send an interval of possible seeds instead of an instance of the table into compression, hoping for a better result.

Multipartite Tables

Goal: build architectures evaluating mathematical functions

Say you have a fixed-point value X, and you need hardware that computes its sine.



The simplest solution: plain tabulation

$2^{w_{\text{in}}}$ entries of w_{out} bits each, so $2^{w_{in}} \times w_{out}$ bits

- very good for really small precisions
- for larger precisions, *cost grows exponentially in* w_{in}

(but only linearly in w_{out})

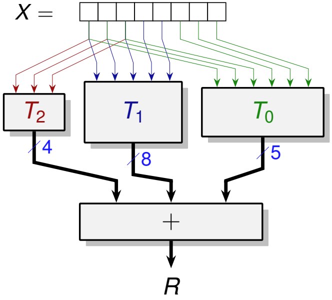
address	content
00000	00000
00001	00011
00010	00011
00011	00101
...	...
11101	11111
11110	11111
11111	11111

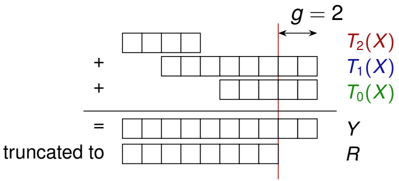
$2^{w_{\text{in}}}$

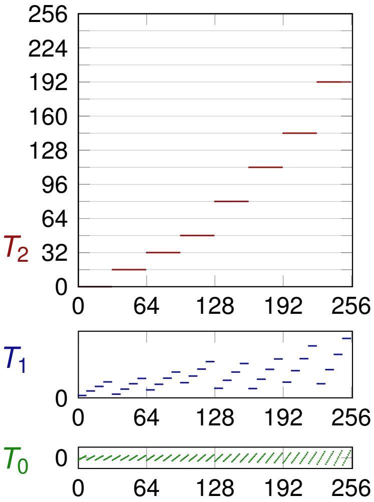
w_{out}

When simple solutions don’t scale, use clever solutions

Multipartite table architectures





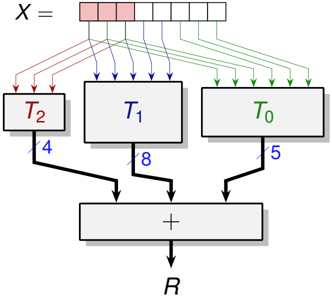


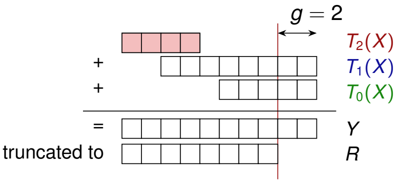
When simple solutions don't scale, use clever solutions

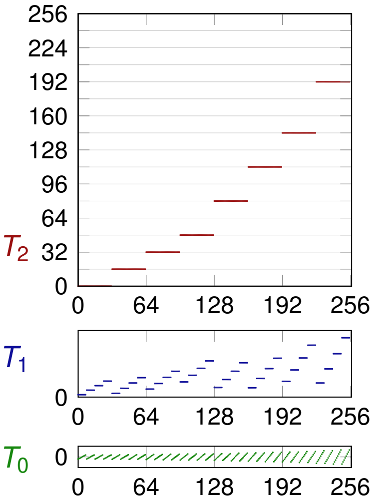
Multipartite table architectures

For this talk, please accept the magic:

let us just wonder that it works, here for $w_{in} = w_{out} = 8$.





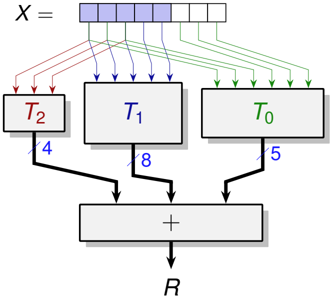


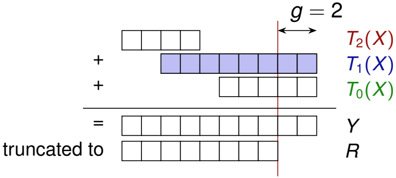
When simple solutions don’t scale, use clever solutions

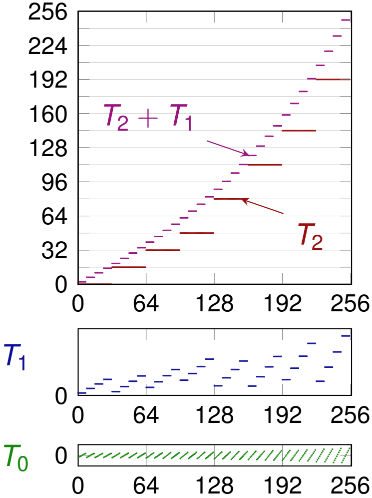
Multipartite table architectures

For this talk, please accept the magic:

let us just wonder that it works, here for $w_{in} = w_{out} = 8$.





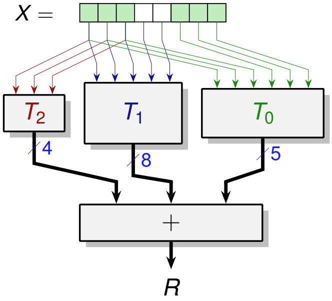


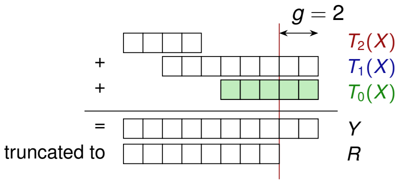
When simple solutions don't scale, use clever solutions

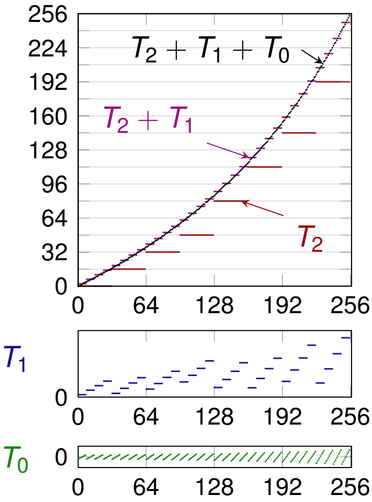
Multipartite table architectures

For this talk, please accept the magic:

let us just wonder that it works, here for $w_{in} = w_{out} = 8$.





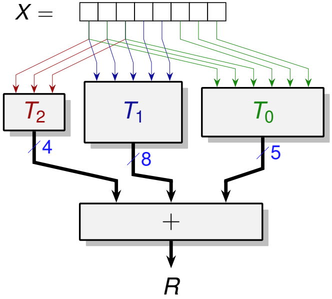


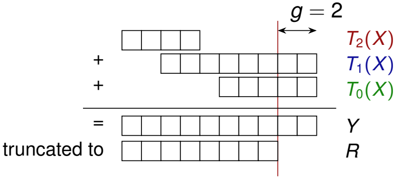
When simple solutions don’t scale, use clever solutions

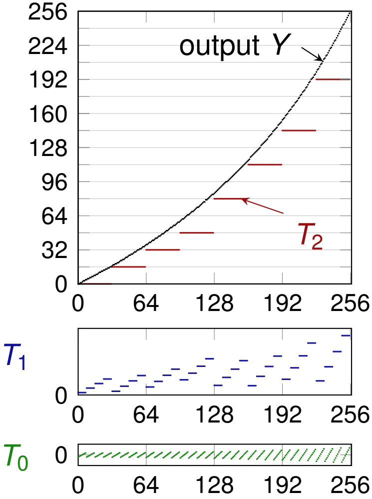
Multipartite table architectures

For this talk, please accept the magic:

let us just wonder that it works, here for $w_{in} = w_{out} = 8$.





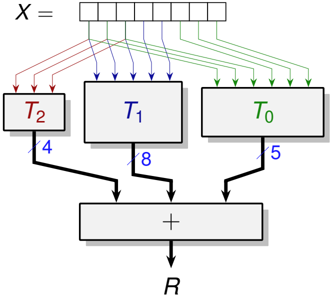


When simple solutions don't scale, use clever solutions

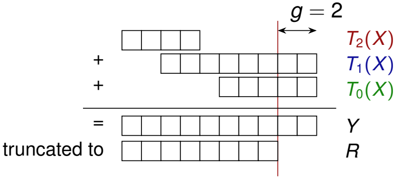
Multipartite table architectures

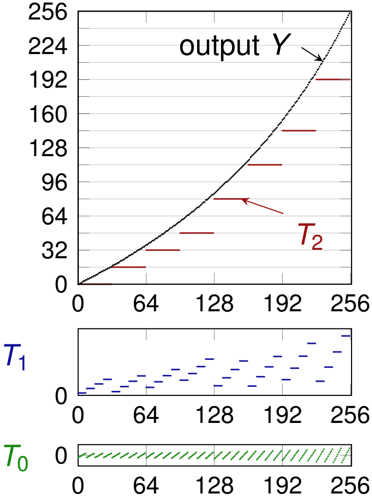
For this talk, please accept the magic:

let us just wonder that it works, here for $w_{in} = w_{out} = 8$.



$$2^4 \cdot 4 + 2^5 \cdot 8 + 2^6 \cdot 5 < 2^8 \cdot 8$$





This magic is has a long history spread over 40 years

[1] D. A. Sunderland, R. A. Strauch, S. S. Wharfield, H. T. Peterson, and C. R. Role, “CMOS/SOS frequency synthesizer LSI circuit for spread spectrum communications,” *IEEE Journal of Solid-State Circuits*, 1984.

[2] D. Das Sarma and D. Matula, “Faithful bipartite ROM reciprocal tables,” in *12th Symposium on Computer Arithmetic*, 1995.

[3] H. Hassler and N. Takagi, “Function evaluation by table look-up and addition,” in *12th Symposium on Computer Arithmetic*, 1995.

[4] J. Stine and M. Schulte, “The symmetric table addition method for accurate function approximation,” *Journal of VLSI Signal Processing*, 1999.

[5] J.-M. Muller, “A few results on table-based methods,” *Reliable Computing*, 1999.

[6] F. de Dinechin and A. Tisserand, “Multipartite table methods,” *IEEE Transactions on Computers*, 2005.

[7] S.-F. Hsiao, P.-H. Wu, C.-S. Wen, and P. K. Meher, “Table size reduction methods for faithfully rounded lookup-table-based multiplierless function evaluation,” *Transactions on Circuits and Systems II*, 2015.

[8] M. Christ, L. Forget, and F. de Dinechin, “Lossless differential table compression for hardware function evaluation,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2022.

Try me in FloPoCo!

`http://www.flopoco.org`

```
flopoco FixFunctionByMultipartiteTable
tablecompression=true
      f="65535/65536*sin(pi/2*x)" lsbIn=-16 lsbOut=-16
```

Multipartite Tables with Integer Linear Programming (ILP)[1]

Current ILP solvers are extremely powerful, with thousands of variables and constraints.

Express your problem as:

- a set of **variables**,
- a set of **linear constraints** between these variables (e.g. $17x + 42y < 2000$ *and* $y > 0$)
- a **linear cost function** to minimize (e.g. *minimize* $3x - 2y$)

---

[1] O. Desrentes, F. de Dinechin, *Using integer linear programming for correctly rounded multipartite architectures*, 2022 (ICFPT)

The (very) big picture of the ILP

Notation: ILP variables in red, ILP constants in blue.

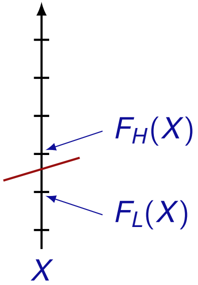
- Let us have one boolean variable for each bit stored in a table
  - $b_{t,a,i}$ is bit i of the value stored at address a in table T_t

The (very) big picture of the ILP

Notation: ILP variables in red, ILP constants in blue.

- Let us have one boolean variable for each bit stored in a table
  - $b_{t,a,i}$ is bit *i* of the value stored at address *a* in table T_t
- Let us have 2^n constraints, one for each value of X
  (which then becomes a constant in each of these constraints)

$$\mathcal{C}_X : F_L(X) \leq Y(X) \leq F_H(X)$$



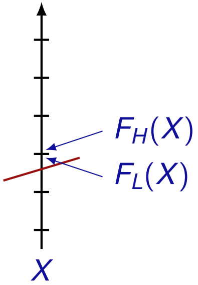
The (very) big picture of the ILP

Notation: ILP variables in red, ILP constants in blue.

- Let us have one boolean variable for each bit stored in a table
  - $b_{t,a,i}$ is bit i of the value stored at address a in table T_t
- Let us have 2^n constraints, one for each value of X
  (which then becomes a constant in each of these constraints)

$$\mathcal{C}_X : F_L(X) \leq Y(X) \leq F_H(X)$$

... with $F_L(X) = F_H(X)$ for correct rounding



The (very) big picture of the ILP

Notation: ILP variables in red, ILP constants in blue.

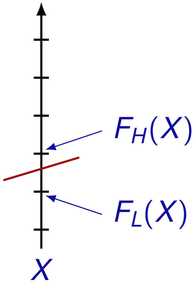
- Let us have one boolean variable for each bit stored in a table
  - $b_{t,a,i}$ is bit i of the value stored at address a in table T_t
- Let us have 2^n constraints, one for each value of X

  (which then becomes a constant in each of these constraints)

$$\mathcal{C}_X : F_L(X) \leq Y(X) \leq F_H(X)$$

... with $F_L(X) = F_H(X)$ for correct rounding

- Now $Y(X)$ can be replaced with $\sum_t T_t(X)$



The (very) big picture of the ILP

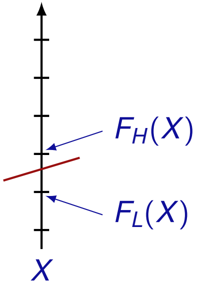
Notation: ILP variables in red, ILP constants in blue.

- Let us have one boolean variable for each bit stored in a table
  - $b_{t,a,i}$ is bit i of the value stored at address a in table T_t
- Let us have 2^n constraints, one for each value of X
  (which then becomes a constant in each of these constraints)

$$\mathcal{C}_X : F_L(X) \leq Y(X) \leq F_H(X)$$

... with $F_L(X) = F_H(X)$ for correct rounding

- Now $Y(X)$ can be replaced with $\sum_t T_t(X)$
- ...where the value of $T_t(X)$ is linear in our variables: $\sum_i 2^i b_{t,a,i}$



The (very) big picture of the ILP

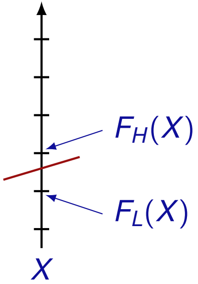
Notation: ILP variables in red, ILP constants in blue.

- Let us have one boolean variable for each bit stored in a table
  - $b_{t,a,i}$ is bit i of the value stored at address a in table T_t
- Let us have 2^n constraints, one for each value of X

  (which then becomes a constant in each of these constraints)

$$\mathcal{C}_X : F_L(X) \leq Y(X) \leq F_H(X)$$

... with $F_L(X) = F_H(X)$ for correct rounding



- Now $Y(X)$ can be replaced with $\sum_t T_t(X)$
- ...where the value of $T_t(X)$ is linear in our variables: $\sum_i 2^i b_{t,a,i}$
- so eventually each constraint is indeed linear, something like

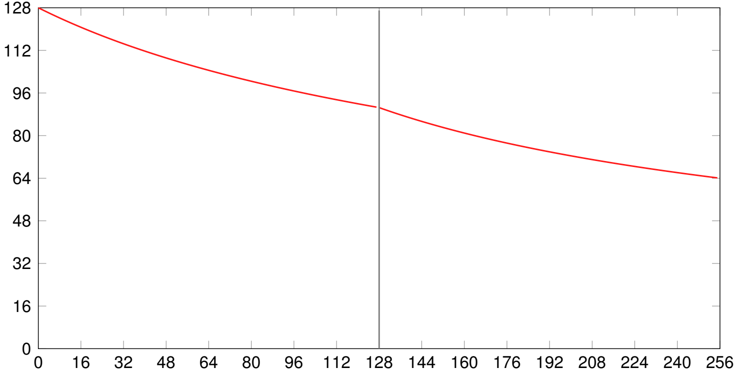
$$\mathcal{C}_X : F_L(X) \leq \sum_t \sum_i 2^i b_{t,a(X),i} \leq F_H(X)$$

Can we adapt Multipartite techniques to compress our seed table?

Let's plot the table, and see.

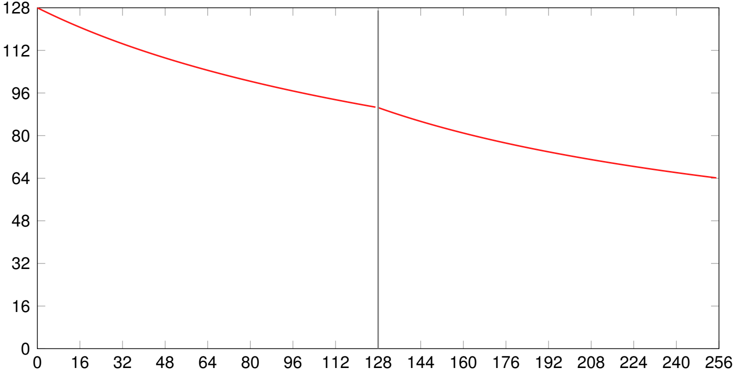
Can we adapt Multipartite techniques to compress our seed table?

Let’s plot the table, and see.



Can we adapt Multipartite techniques to compress our seed table?

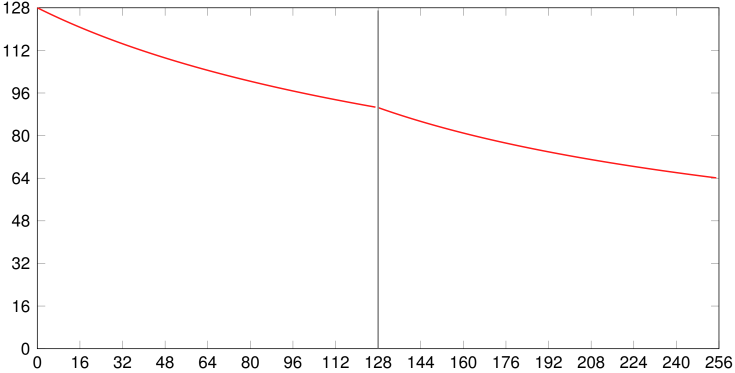
Let's plot the table, and see.



Seed table is always positive, approximately monotone, approximately convex, perfect for multipartite tables!

Can we adapt Multipartite techniques to compress our seed table?

Let's plot the table, and see.



Seed table is always positive, approximately monotone, approximately convex, perfect for multipartite tables!

Not *exactly* table compression

Seed table is not a function $x \mapsto f(x)$ but $x \mapsto [y_L; y_H]$.

---

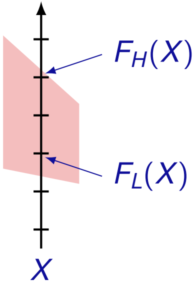
[1] M. Christ, L. Forget, and F. de Dinechin, *Lossless differential table compression for hardware function evaluation*, 2022 (TCAS-II)

Not *exactly* table compression

Seed table is not a function $x \mapsto f(x)$ but $x \mapsto [y_L; y_H]$.

$$\mathcal{C}_X : F_L(X) \leq Y(X) \leq F_H(X)$$

$F_L(X)$ and $F_H(X)$ are exactly what we needed.



---

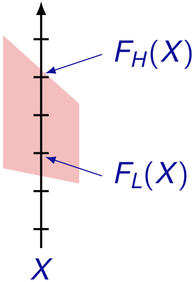
[1] M. Christ, L. Forget, and F. de Dinechin, *Lossless differential table compression for hardware function evaluation*, 2022 (TCAS-II)

Not *exactly* table compression

Seed table is not a function $x \mapsto f(x)$ but $x \mapsto [y_L; y_H]$.

$$\mathcal{C}_X : F_L(X) \leq Y(X) \leq F_H(X)$$

$F_L(X)$ and $F_H(X)$ are exactly what we needed.



This isn't generic table compression like LDTC[1], and **technically** we're not compressing **a** table, but any of the possible tables.

---

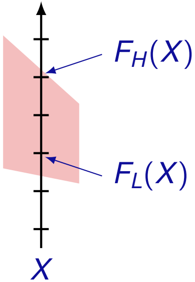
[1] M. Christ, L. Forget, and F. de Dinechin, *Lossless differential table compression for hardware function evaluation*, 2022 (TCAS-II)

Not *exactly* table compression

Seed table is not a function $x \mapsto f(x)$ but $x \mapsto [y_L; y_H]$.

$$\mathcal{C}_X : F_L(X) \leq Y(X) \leq F_H(X)$$

$F_L(X)$ and $F_H(X)$ are exactly what we needed.



This isn't generic table compression like LDTC[1], and **technically** we're not compressing **a** table, but any of the possible tables.
In particular, we want to compress the one with the smallest compressed size…

---

[1] M. Christ, L. Forget, and F. de Dinechin, *Lossless differential table compression for hardware function evaluation*, 2022 (TCAS-II)

Results of the Integer Linear Program

Issues with $w_{\text{in}} = 13, w_{\text{out}} = 14$

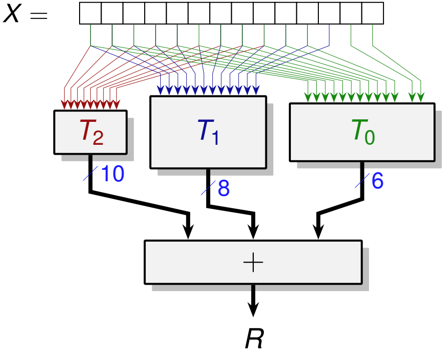
The ILP program cannot find cheaper than the plain table. . . Let's try to increase w_{out}.

Results of the Integer Linear Program

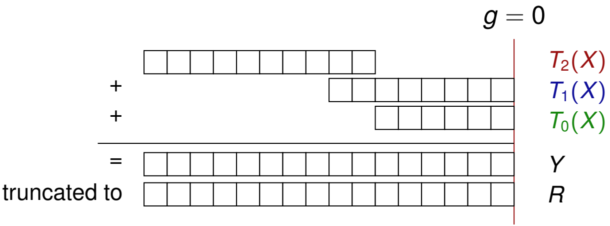
Issues with $w_{\text{in}} = 13$, $w_{\text{out}} = 14$

The ILP program cannot find cheaper than the plain table… Let's try to increase w_{out}.

$w_{\text{out}} = 15$



$$2^{9} \cdot 10 + 2^{12} \cdot 8 + 2^{12} \cdot 6 < 2^{14} \cdot 15$$

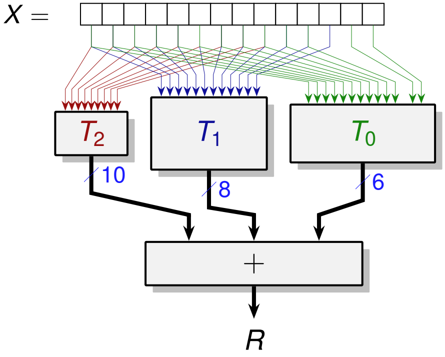


Results of the Integer Linear Program

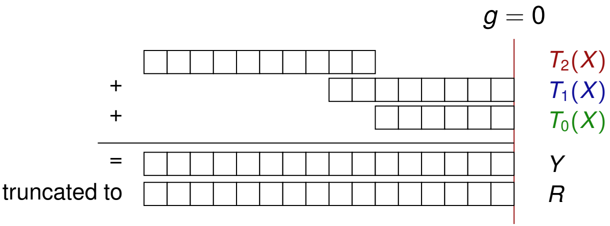
Issues with $w_{\text{in}} = 13, w_{\text{out}} = 14$

The ILP program cannot find cheaper than the plain table. . . Let's try to increase w_{out}.

$w_{\text{out}} = 15$; **25.4%** of initial table size



$$2^9 \cdot 10 + 2^{12} \cdot 8 + 2^{12} \cdot 6 < 2^{14} \cdot 14$$

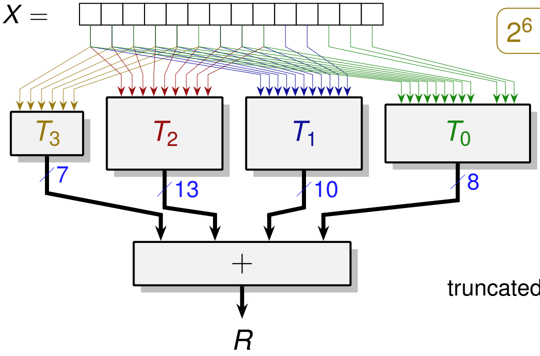


Results of the Integer Linear Program

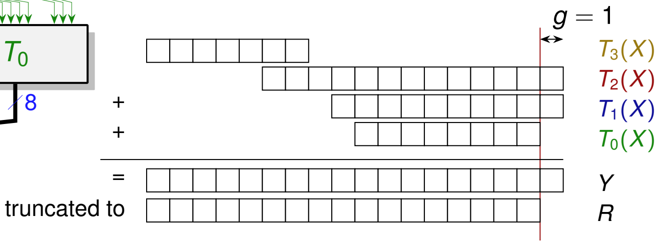
Issues with $w_{\text{in}} = 13$, $w_{\text{out}} = 14$

The ILP program cannot find cheaper than the plain table... Let's try to increase w_{out}.

$w_{\text{out}} = 16$



$$2^6 \cdot 7 + 2^9 \cdot 13 + 2^{10} \cdot 10 + 2^{11} \cdot 8 < 2^{14} \cdot 16$$

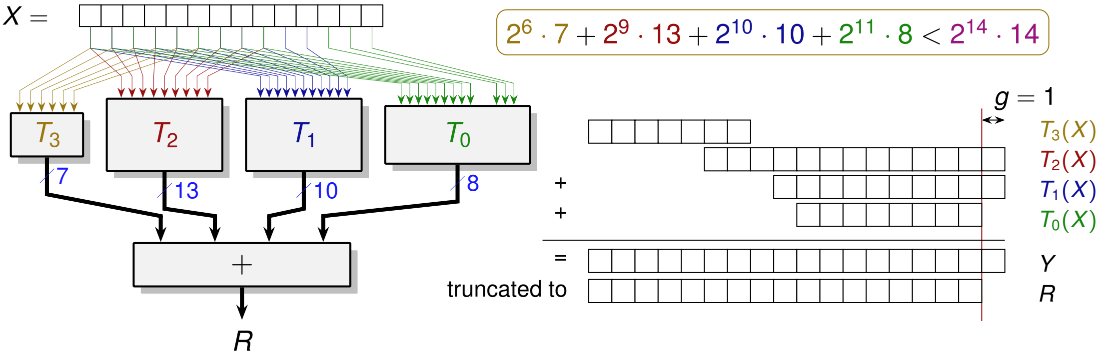


Results of the Integer Linear Program

Issues with $w_{\text{in}} = 13$, $w_{\text{out}} = 14$

The ILP program cannot find cheaper than the plain table… Let's try to increase w_{out}.

$w_{\text{out}} = 16$; **13.7%** of initial table size



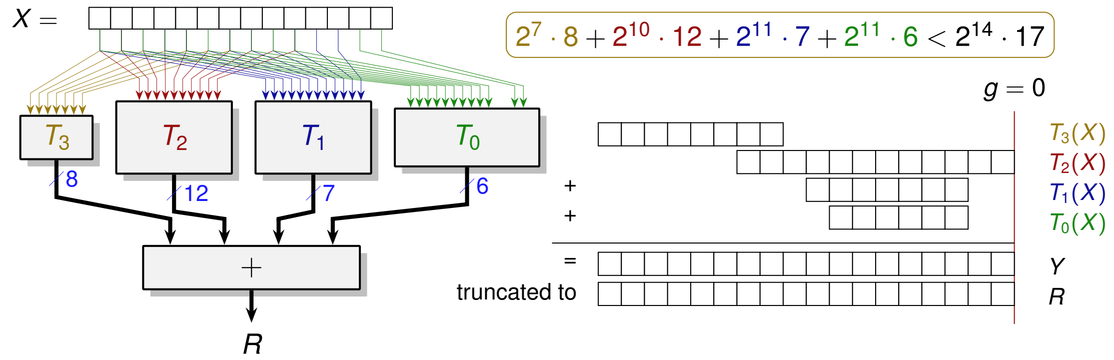
$$2^6 \cdot 7 + 2^9 \cdot 13 + 2^{10} \cdot 10 + 2^{11} \cdot 8 < 2^{14} \cdot 14$$

Results of the Integer Linear Program

Issues with $w_{\text{in}} = 13$, $w_{\text{out}} = 14$

The ILP program cannot find cheaper than the plain table. . . Let's try to increase w_{out}.

$w_{\text{out}} = 17$



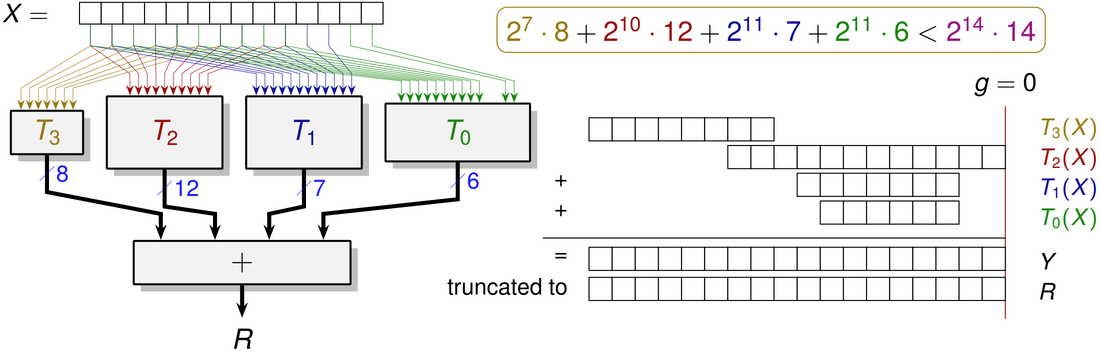
$$2^7 \cdot 8 + 2^{10} \cdot 12 + 2^{11} \cdot 7 + 2^{11} \cdot 6 < 2^{14} \cdot 17$$

Results of the Integer Linear Program

Issues with $w_{\text{in}} = 13, w_{\text{out}} = 14$

The ILP program cannot find cheaper than the plain table… Let's try to increase w_{out}.

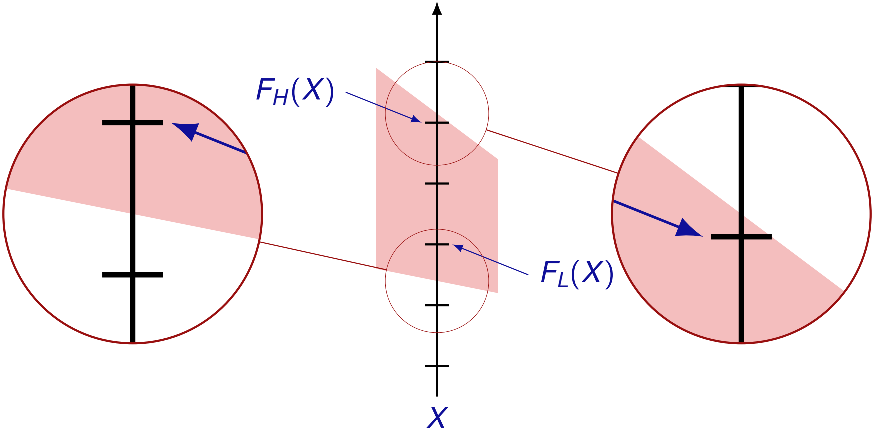
$w_{\text{out}} = 17$; **17.4%** of initial table size



$$2^7 \cdot 8 + 2^{10} \cdot 12 + 2^{11} \cdot 7 + 2^{11} \cdot 6 < 2^{14} \cdot 14$$

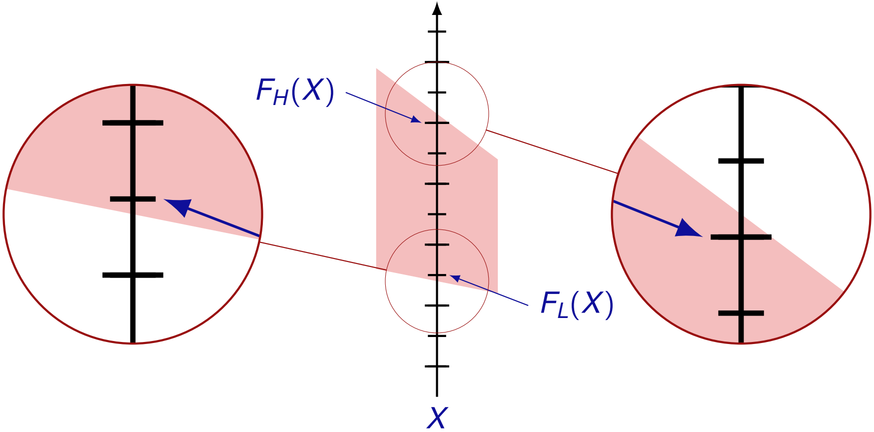
Why is that?

When $w_{\text{out}} = 14$: 3 seeds, interval size 2×2^{-14}



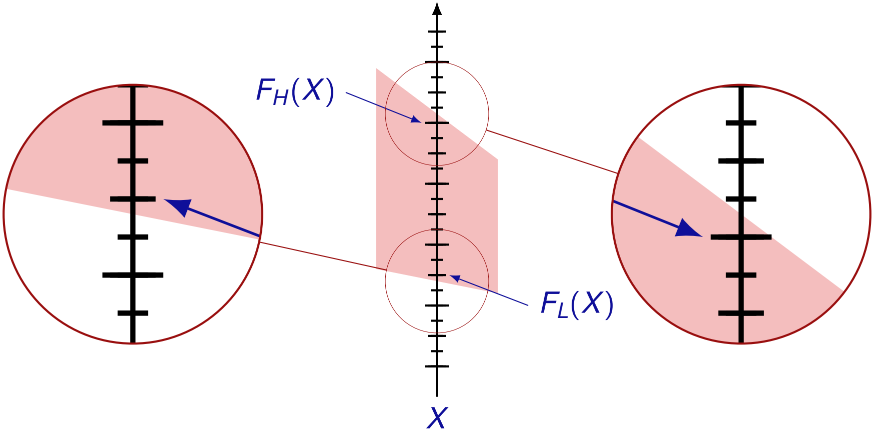
Why is that?

When $w_{\text{out}} = 15$: 6 seeds, interval size 2.5×2^{-14}



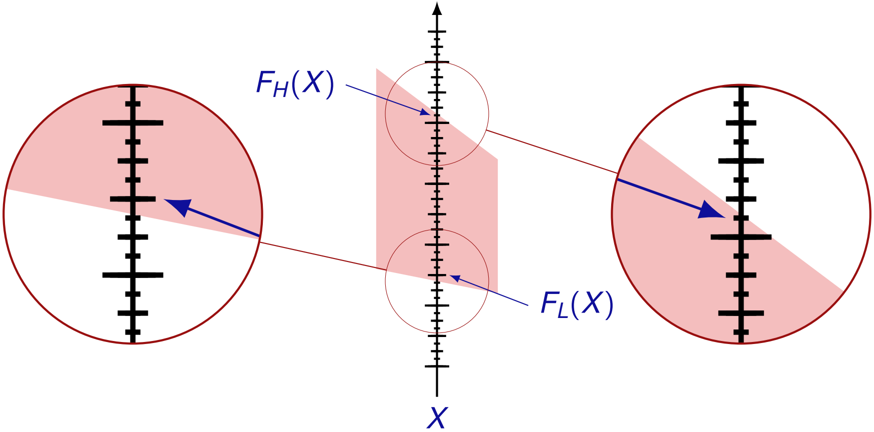
Why is that?

When $w_{\text{out}} = 16$: 11 seeds, interval size 2.5×2^{-14}



Why is that?

When $w_{\text{out}} = 17$: 22 seeds, interval size 2.625×2^{-14}



Why is that?

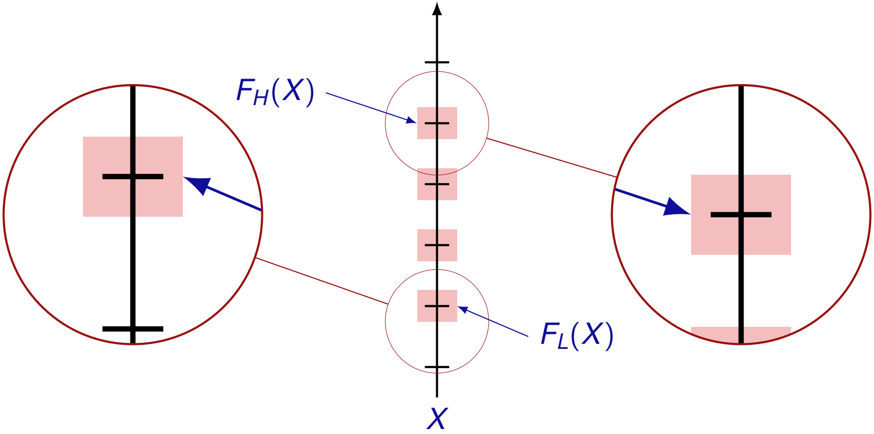
→ But why does it get bigger again ?

Why is that?

The interval is a simplification. Example: the seed is `0x3f82fc00`

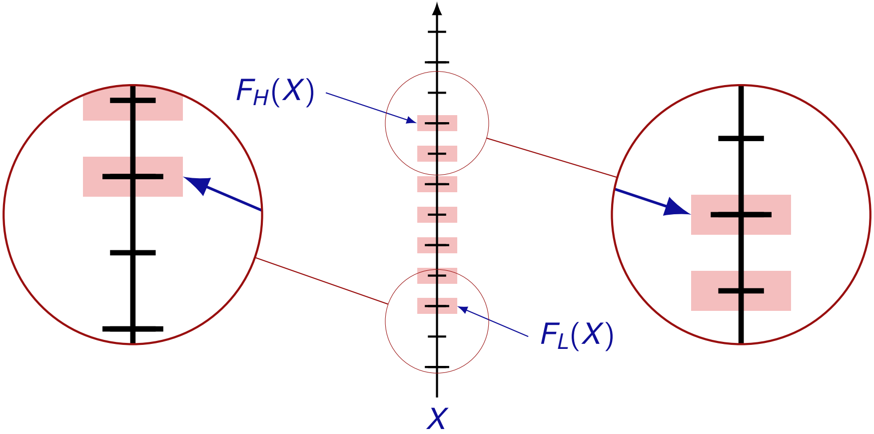
Why is that?

The interval is a simplification. Example: the seed is `0x3f82fc00`
When $w_{\text{out}} = 14$: 4 seeds, interval size 3×2^{-14}



Why is that?

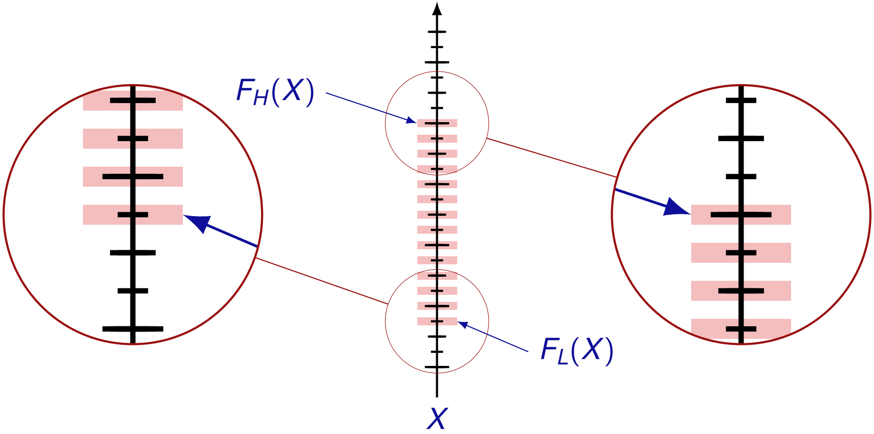
The interval is a simplification. Example: the seed is `0x3f82fc00`
When $w_{\text{out}} = 15$: 7 seeds, interval size 3×2^{-14}



Why is that?

The interval is a simplification. Example: the seed is `0x3f82fc00`

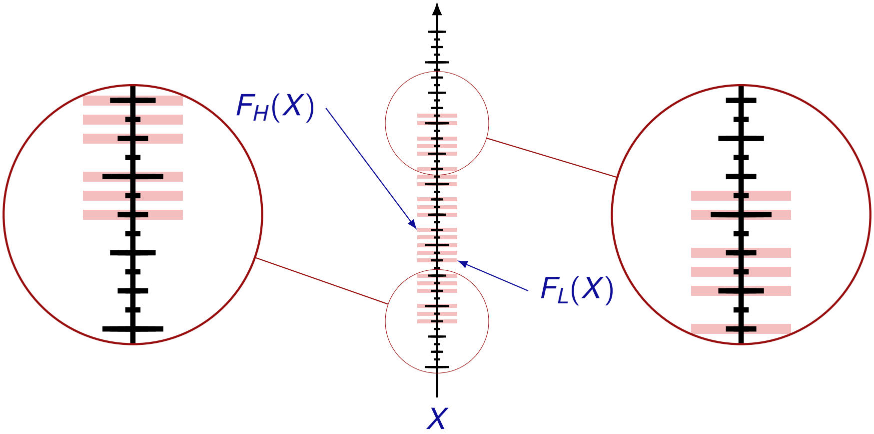
When $w_{\text{out}} = 16$: 14 seeds, interval size 3.25×2^{-14}



Why is that?

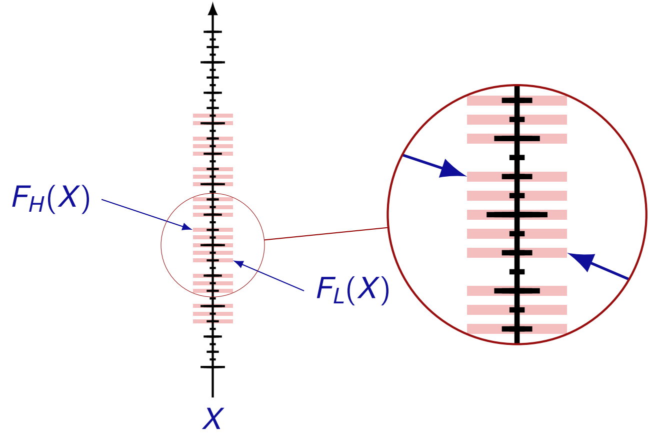
The interval is a simplification. Example: the seed is `0x3f82fc00`

When $w_{\text{out}} = 17$: 5 seeds, interval size 0.5×2^{-14}



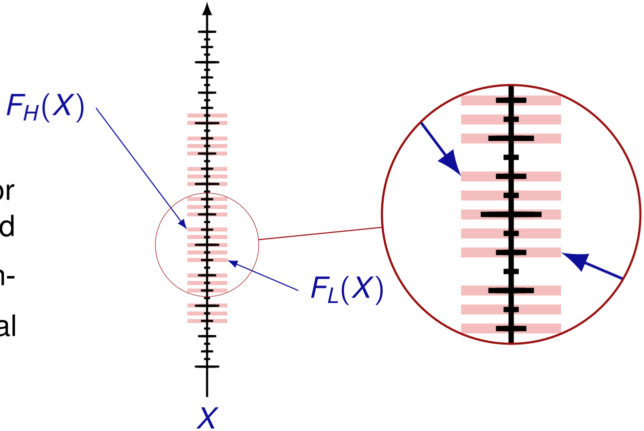
Why is that?

The interval is a simplification. Example: the seed is `0x3f82fc00`
When $w_{\text{out}} = 17$: 5 seeds, interval size 0.5×2^{-14}



Why is that?

The interval is a simplification. Example: the seed is `0x3f82fc00`
When $w_{\text{out}} = 17$: 5 seeds, interval size 0.5×2^{-14}



The algorithm for faithfully rounded FP32 $\frac{1}{\sqrt{x}}$ is sensitive to the initial seed.

Conclusion

What we wanted

Cheap vector normalisation for ML (and hopefully more)

Conclusion

What we wanted

Cheap vector normalisation for ML (and hopefully more)

The initial plan

Conclusion

What we wanted

Cheap vector normalisation for ML (and hopefully more)

The initial plan

- Phase 1: Find the perfect seed table

Conclusion

What we wanted

Cheap vector normalisation for ML (and hopefully more)

The initial plan

- Phase 1: Find the perfect seed table
- Phase 2: Stick it in an ILP solver

Conclusion

What we wanted

Cheap vector normalisation for ML (and hopefully more)

The initial plan

- Phase 1: Find the perfect seed table
- Phase 2: Stick it in an ILP solver
- Phase 3: Profit

Conclusion

What we wanted

Cheap vector normalisation for ML (and hopefully more)

The initial plan

- Phase 1: Find the perfect seed table
- Phase 2: Stick it in an ILP solver
- Phase 3: Profit

What we found along the way

Conclusion

What we wanted

Cheap vector normalisation for ML (and hopefully more)

The initial plan

- Phase 1: Find the perfect seed table
- Phase 2: Stick it in an ILP solver
- Phase 3: Profit

What we found along the way

- The perfect table does not exist.

Conclusion

What we wanted

Cheap vector normalisation for ML (and hopefully more)

The initial plan

- Phase 1: Find the perfect seed table
- Phase 2: Stick it in an ILP solver
- Phase 3: Profit

What we found along the way

- The perfect table does not exist. That's actually better for the ILP.

Conclusion

What we wanted

Cheap vector normalisation for ML (and hopefully more)

The initial plan

- Phase 1: Find the perfect seed table
- Phase 2: Stick it in an ILP solver
- Phase 3: Profit

What we found along the way

- The perfect table does not exist. That's actually better for the ILP.
- Guard bits for table compression do not work like the ones for function evaluation.

Conclusion

What we wanted

Cheap vector normalisation for ML (and hopefully more)

The initial plan

- Phase 1: Find the perfect seed table
- Phase 2: Stick it in an ILP solver
- Phase 3: Profit

What we found along the way

- The perfect table does not exist. That's actually better for the ILP.
- Guard bits for table compression do not work like the ones for function evaluation. Similar to LDTC, no guard bits

Conclusion

What we wanted

Cheap vector normalisation for ML (and hopefully more)

The initial plan

- Phase 1: Find the perfect seed table
- Phase 2: Stick it in an ILP solver
- Phase 3: Profit

What we found along the way

- The perfect table does not exist. That's actually better for the ILP.
- Guard bits for table compression do not work like the ones for function evaluation. Similar to LDTC, no guard bits
- Increasing w_{out} reduces the compressed size.

Conclusion

What we wanted

Cheap vector normalisation for ML (and hopefully more)

The initial plan

- Phase 1: Find the perfect seed table
- Phase 2: Stick it in an ILP solver
- Phase 3: Profit

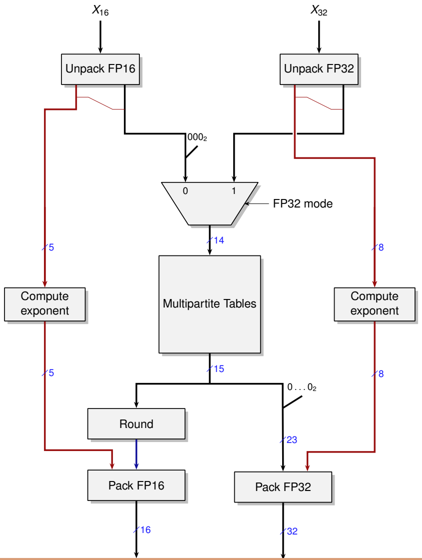
What we found along the way

- The perfect table does not exist. That's actually better for the ILP.
- Guard bits for table compression do not work like the ones for function evaluation. Similar to LDTC, no guard bits
- Increasing w_{out} reduces the compressed size. Until we choose the wrong seed representation. . .

Next steps

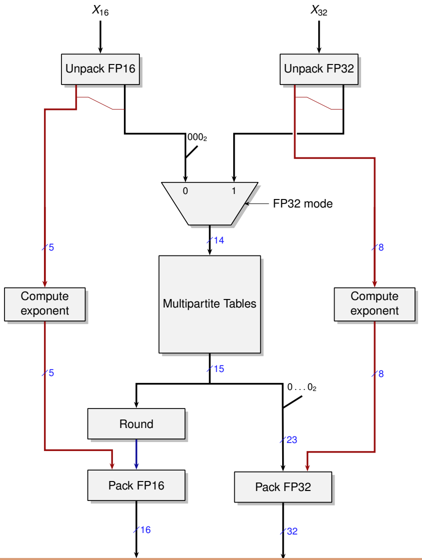
Next steps

- Actually build the hardware operator



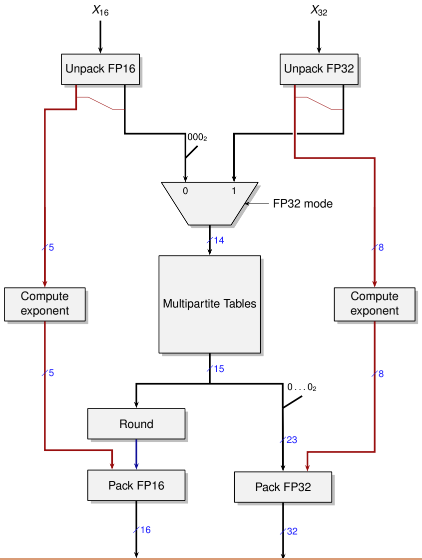
Next steps

- Actually build the hardware operator
- Set up the software programs to use it



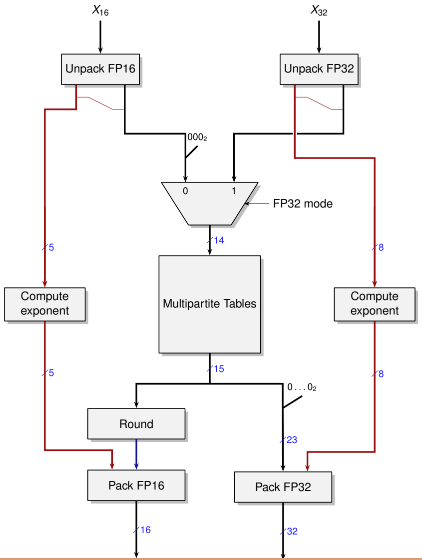
Next steps

- Actually build the hardware operator
- Set up the software programs to use it
- Reflect on the limitations of the interval representation



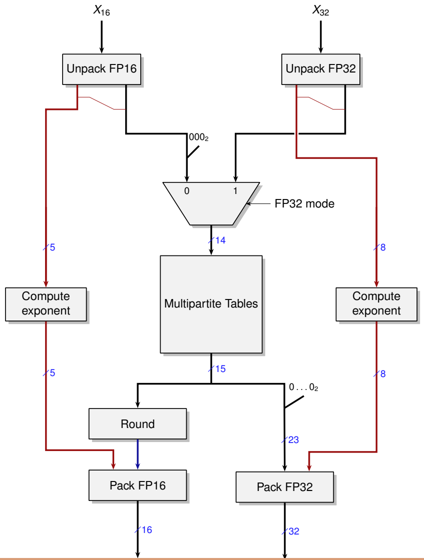
Next steps

- Actually build the hardware operator
- Set up the software programs to use it
- Reflect on the limitations of the interval representation
- Check we can use the seed for division (brute-force will not work anymore. . .)



Next steps

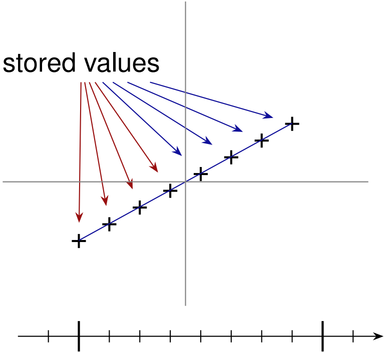
- Actually build the hardware operator
- Set up the software programs to use it
- Reflect on the limitations of the interval representation
- Check we can use the seed for division (brute-force will not work anymore. . .)
- Try to compress other tables using multipartite architectures

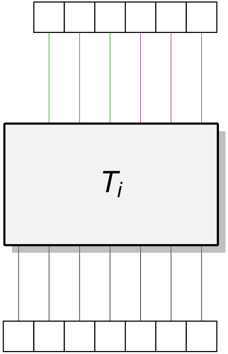


Questions?

Outline

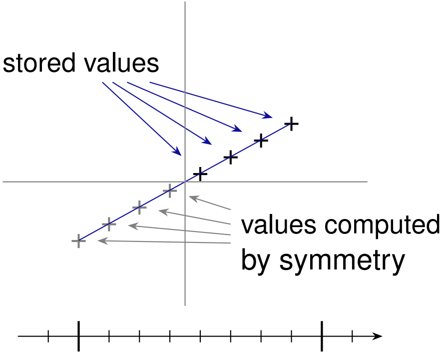
To use or not to use symmetry





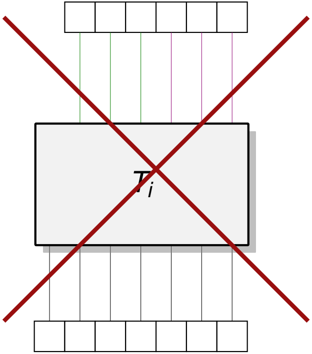
To use or not to use symmetry

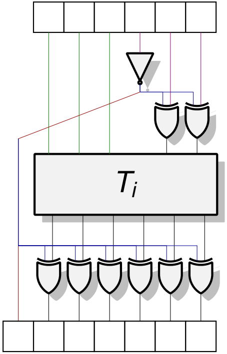
Symmetry may trade one table input bit for two rows of XOR gates:



stored values

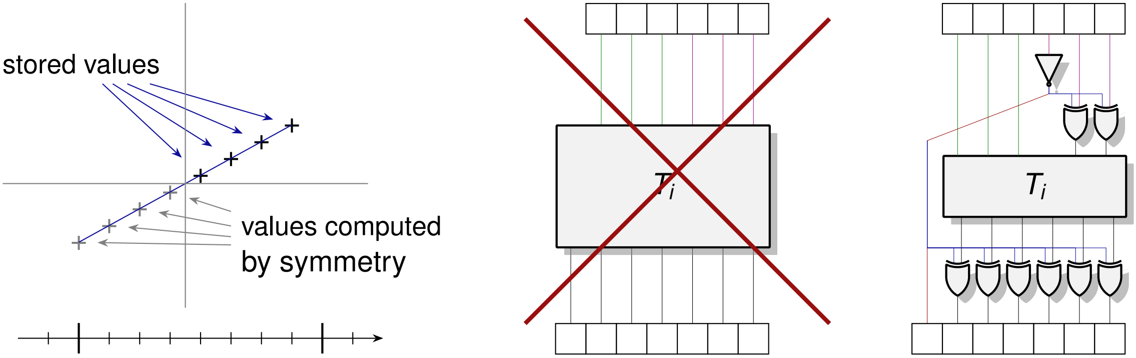
values computed by symmetry





To use or not to use symmetry

Symmetry may trade one table input bit for two rows of XOR gates:



- The XORs save LUTs, but cost LUTs:
  - detailed evaluation of the relevance of this idea in the paper

Results of the Integer Linear Program

Issues with $w_{\text{in}} = 13$, $w_{\text{out}} = 14$

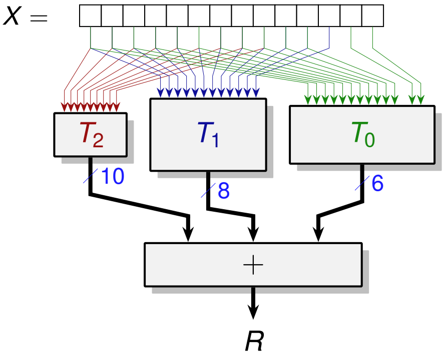
The ILP program cannot find cheaper than the plain table. . . Let's try to increase w_{out}.

Results of the Integer Linear Program

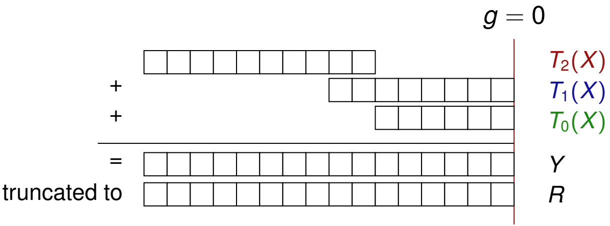
Issues with $w_{\text{in}} = 13$, $w_{\text{out}} = 14$

The ILP program cannot find cheaper than the plain table... Let's try to increase w_{out}.

$w_{\text{out}} = 15$



$$2^9 \cdot 10 + 2^{12} \cdot 8 + 2^{12} \cdot 6 < 2^{14} \cdot 15$$

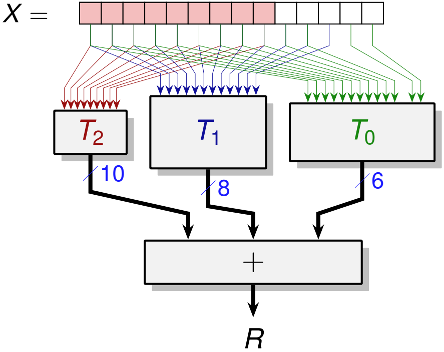


Results of the Integer Linear Program

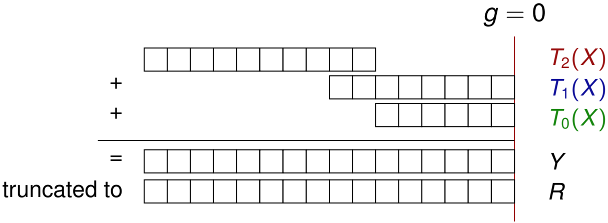
Issues with $w_{\text{in}} = 13$, $w_{\text{out}} = 14$

The ILP program cannot find cheaper than the plain table... Let's try to increase w_{out}.

$w_{\text{out}} = 15$



$$2^9 \cdot 10 + 2^{12} \cdot 8 + 2^{12} \cdot 6 < 2^{14} \cdot 15$$

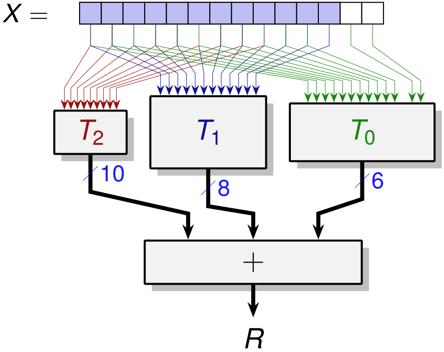


Results of the Integer Linear Program

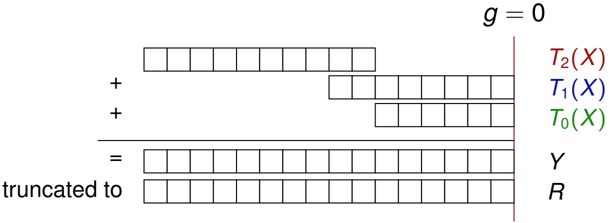
Issues with $w_{\text{in}} = 13$, $w_{\text{out}} = 14$

The ILP program cannot find cheaper than the plain table… Let's try to increase w_{out}.

$w_{\text{out}} = 15$



$$2^9 \cdot 10 + 2^{12} \cdot 8 + 2^{12} \cdot 6 < 2^{14} \cdot 15$$

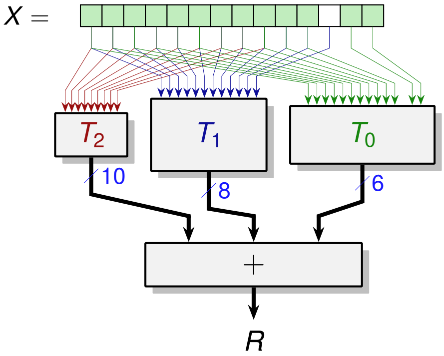


Results of the Integer Linear Program

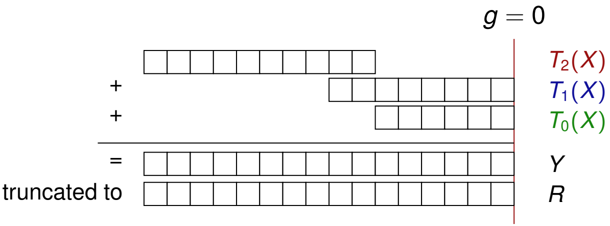
Issues with $w_{\text{in}} = 13$, $w_{\text{out}} = 14$

The ILP program cannot find cheaper than the plain table... Let's try to increase w_{out}.

$w_{\text{out}} = 15$



$$2^9 \cdot 10 + 2^{12} \cdot 8 + 2^{12} \cdot 6 < 2^{14} \cdot 15$$

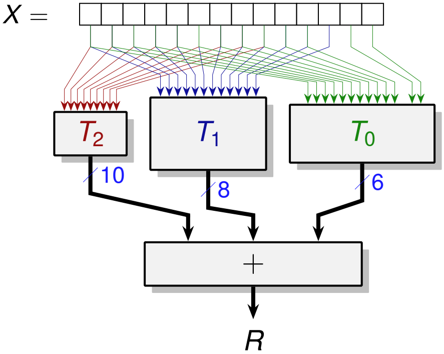


Results of the Integer Linear Program

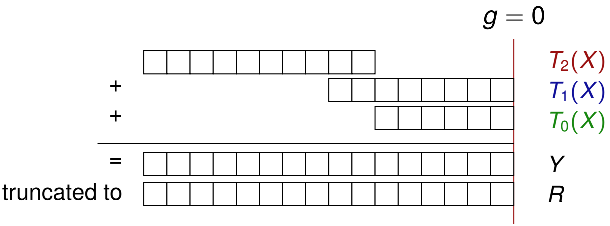
Issues with $w_{\text{in}} = 13$, $w_{\text{out}} = 14$

The ILP program cannot find cheaper than the plain table... Let's try to increase w_{out}.

$w_{\text{out}} = 15$; **25.4%** of initial table size



$$2^9 \cdot 10 + 2^{12} \cdot 8 + 2^{12} \cdot 6 < 2^{14} \cdot 14$$

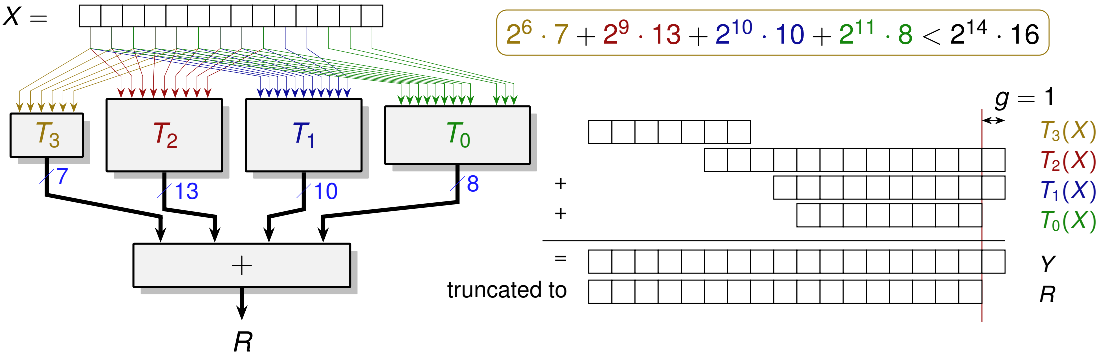


Results of the Integer Linear Program

Issues with $w_{\text{in}} = 13$, $w_{\text{out}} = 14$

The ILP program cannot find cheaper than the plain table… Let's try to increase w_{out}.

$w_{\text{out}} = 16$



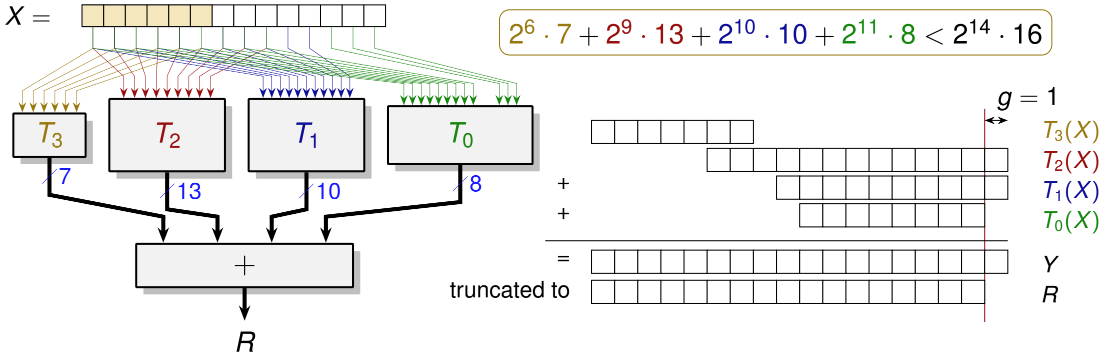
$$2^6 \cdot 7 + 2^9 \cdot 13 + 2^{10} \cdot 10 + 2^{11} \cdot 8 < 2^{14} \cdot 16$$

Results of the Integer Linear Program

Issues with $w_{\text{in}} = 13$, $w_{\text{out}} = 14$

The ILP program cannot find cheaper than the plain table… Let’s try to increase w_{out}.

$w_{\text{out}} = 16$



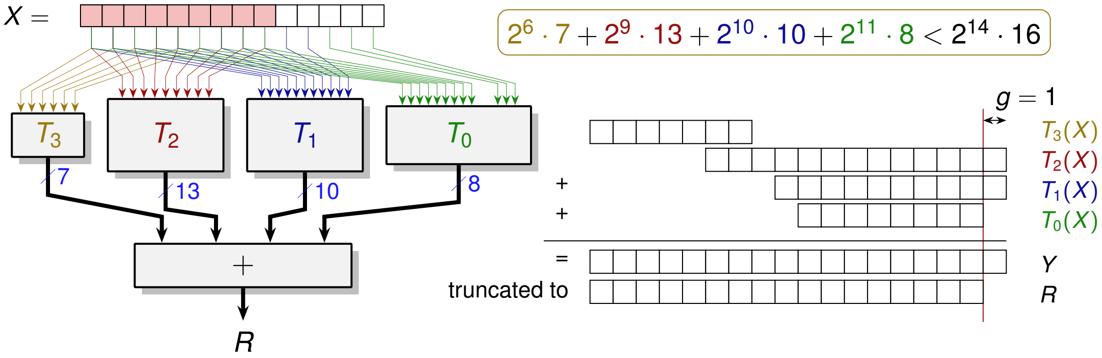
$$2^6 \cdot 7 + 2^9 \cdot 13 + 2^{10} \cdot 10 + 2^{11} \cdot 8 < 2^{14} \cdot 16$$

Results of the Integer Linear Program

Issues with $w_{\text{in}} = 13$, $w_{\text{out}} = 14$

The ILP program cannot find cheaper than the plain table… Let's try to increase w_{out}.

$w_{\text{out}} = 16$



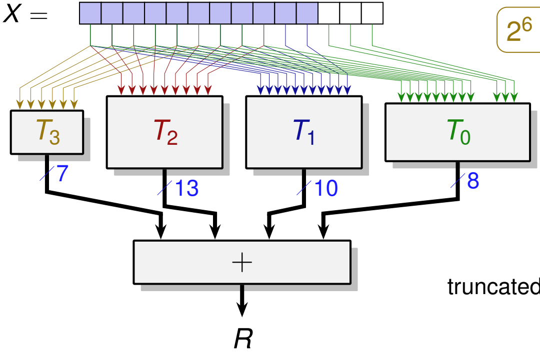
$$2^6 \cdot 7 + 2^9 \cdot 13 + 2^{10} \cdot 10 + 2^{11} \cdot 8 < 2^{14} \cdot 16$$

Results of the Integer Linear Program

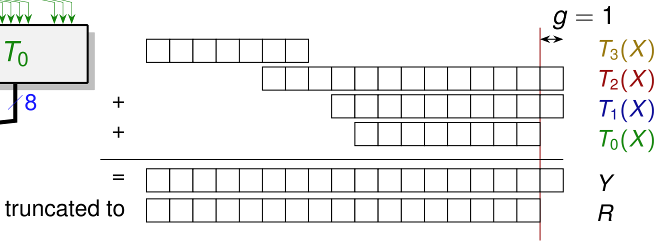
Issues with $w_{\text{in}} = 13$, $w_{\text{out}} = 14$

The ILP program cannot find cheaper than the plain table… Let's try to increase w_{out}.

$w_{\text{out}} = 16$



$$2^6 \cdot 7 + 2^9 \cdot 13 + 2^{10} \cdot 10 + 2^{11} \cdot 8 < 2^{14} \cdot 16$$

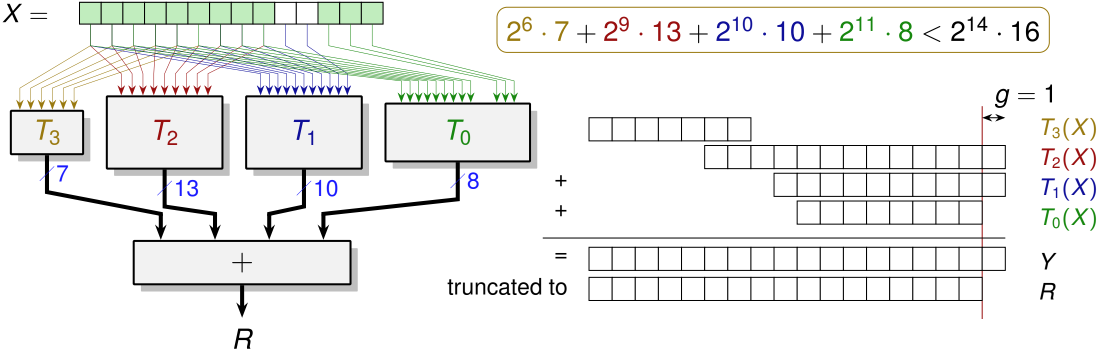


Results of the Integer Linear Program

Issues with $w_{\text{in}} = 13, w_{\text{out}} = 14$

The ILP program cannot find cheaper than the plain table… Let's try to increase w_{out}.

$w_{\text{out}} = 16$



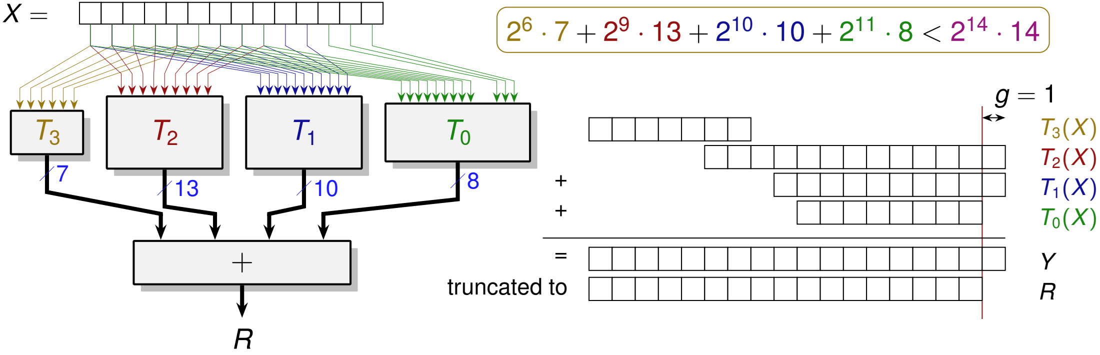
$$2^6 \cdot 7 + 2^9 \cdot 13 + 2^{10} \cdot 10 + 2^{11} \cdot 8 < 2^{14} \cdot 16$$

Results of the Integer Linear Program

Issues with $w_{\text{in}} = 13$, $w_{\text{out}} = 14$

The ILP program cannot find cheaper than the plain table… Let's try to increase w_{out}.

$w_{\text{out}} = 16$; **13.7%** of initial table size



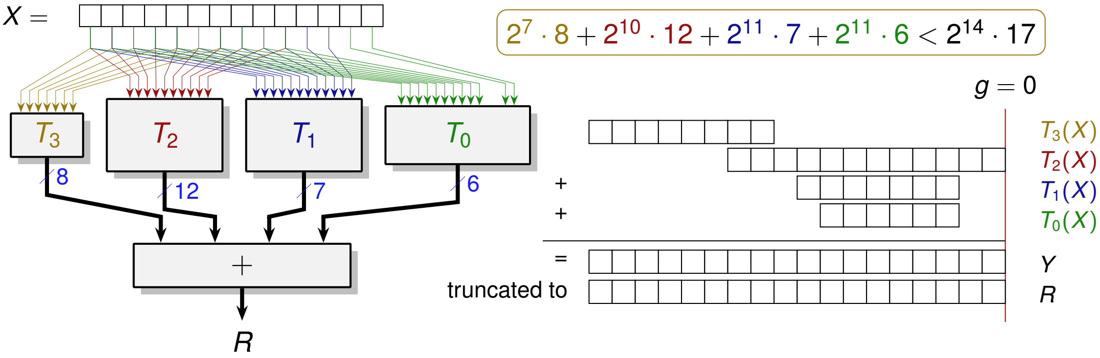
$$2^6 \cdot 7 + 2^9 \cdot 13 + 2^{10} \cdot 10 + 2^{11} \cdot 8 < 2^{14} \cdot 14$$

Results of the Integer Linear Program

Issues with $w_{\text{in}} = 13, w_{\text{out}} = 14$

The ILP program cannot find cheaper than the plain table... Let's try to increase w_{out}.

$w_{\text{out}} = 17$



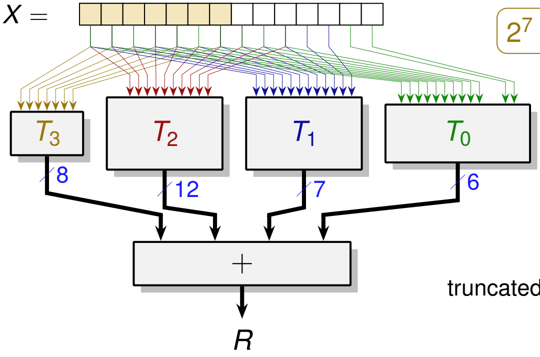
$$2^7 \cdot 8 + 2^{10} \cdot 12 + 2^{11} \cdot 7 + 2^{11} \cdot 6 < 2^{14} \cdot 17$$

Results of the Integer Linear Program

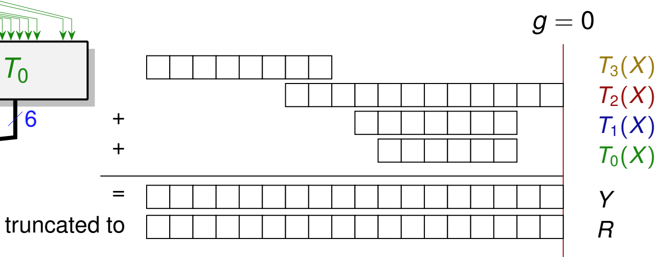
Issues with $w_{\text{in}} = 13$, $w_{\text{out}} = 14$

The ILP program cannot find cheaper than the plain table… Let's try to increase w_{out}.

$w_{\text{out}} = 17$



$$2^7 \cdot 8 + 2^{10} \cdot 12 + 2^{11} \cdot 7 + 2^{11} \cdot 6 < 2^{14} \cdot 17$$

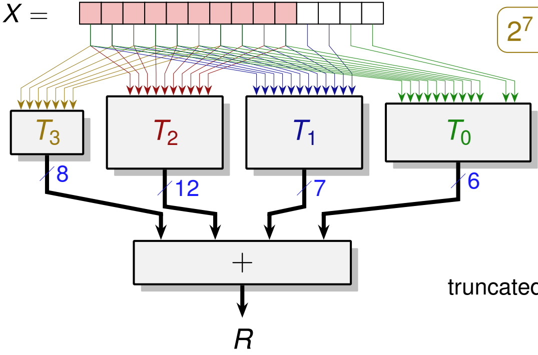


Results of the Integer Linear Program

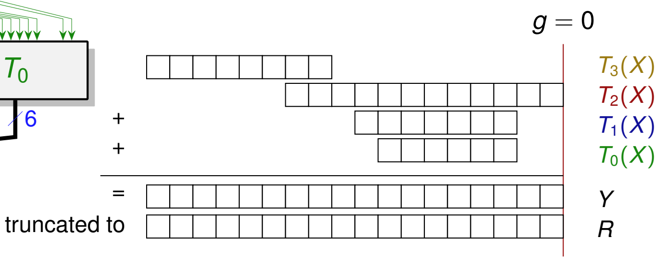
Issues with $w_{\text{in}} = 13, w_{\text{out}} = 14$

The ILP program cannot find cheaper than the plain table… Let's try to increase w_{out}.

$w_{\text{out}} = 17$



$$2^7 \cdot 8 + 2^{10} \cdot 12 + 2^{11} \cdot 7 + 2^{11} \cdot 6 < 2^{14} \cdot 17$$

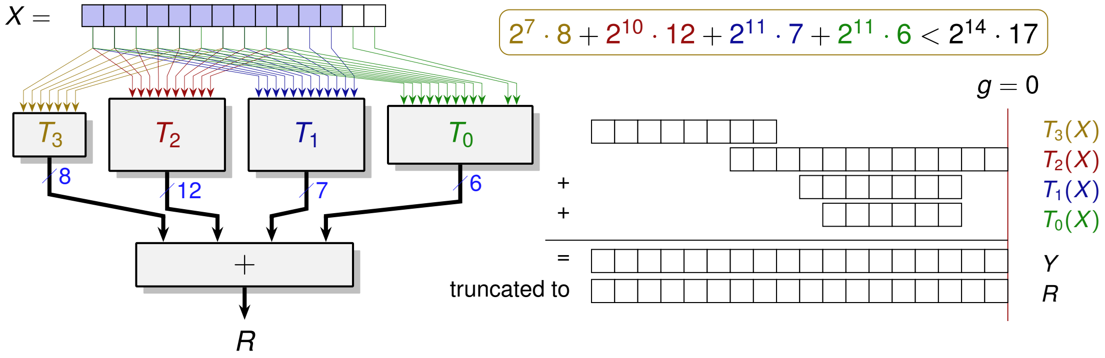


Results of the Integer Linear Program

Issues with $w_{\text{in}} = 13$, $w_{\text{out}} = 14$

The ILP program cannot find cheaper than the plain table... Let's try to increase w_{out}.

$w_{\text{out}} = 17$



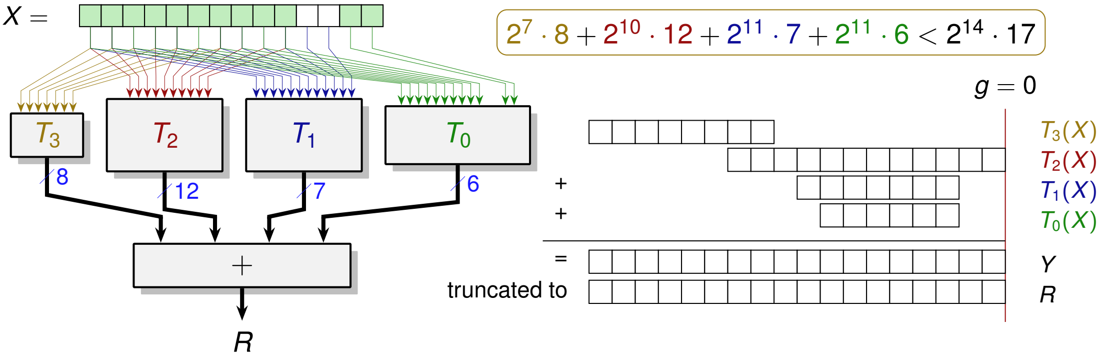
$$2^7 \cdot 8 + 2^{10} \cdot 12 + 2^{11} \cdot 7 + 2^{11} \cdot 6 < 2^{14} \cdot 17$$

Results of the Integer Linear Program

Issues with $w_{\text{in}} = 13$, $w_{\text{out}} = 14$

The ILP program cannot find cheaper than the plain table… Let's try to increase w_{out}.

$w_{\text{out}} = 17$



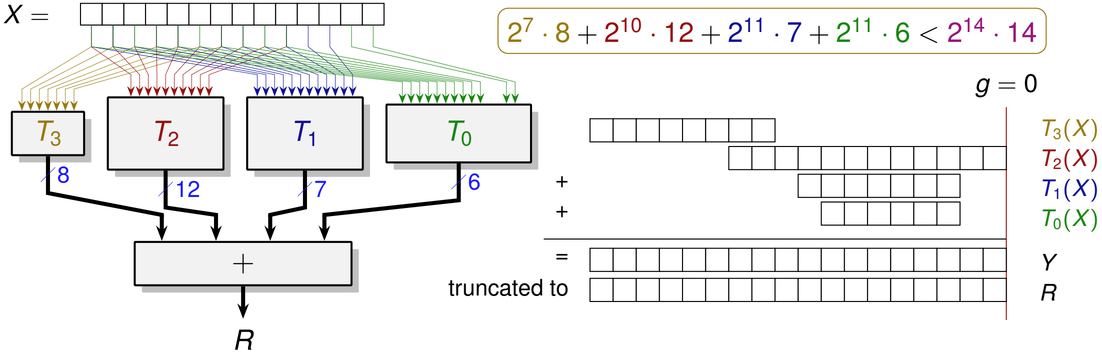
$$2^7 \cdot 8 + 2^{10} \cdot 12 + 2^{11} \cdot 7 + 2^{11} \cdot 6 < 2^{14} \cdot 17$$

Results of the Integer Linear Program

Issues with $w_{\text{in}} = 13$, $w_{\text{out}} = 14$

The ILP program cannot find cheaper than the plain table. . . Let's try to increase w_{out}.

$w_{\text{out}} = 17$; **17.4%** of initial table size



$$2^7 \cdot 8 + 2^{10} \cdot 12 + 2^{11} \cdot 7 + 2^{11} \cdot 6 < 2^{14} \cdot 14$$

Speed and scaling of the algorithms

Increasing w_{out} does not make the program exponentially slower.

Tested for $w_{\text{out}} \in [\![14, 18]\!]$ on a laptop (intel i7 from 2019, 16 GB RAM)

- Generating the intervals : always less than a minute
- ILP : always less than 30 min

Optimality ?

In short : no

Optimality ?

In short : no

Many heuristics that potentially removes optimiality.

- Replace some binary variables with integers
- Prune the external loop
- Set a maximum amount of guard bits
- **ILP timeout**

Optimality ?

In short : no

Many heuristics that potentially removes optimiality.

- Replace some binary variables with integers
- Prune the external loop
- Set a maximum amount of guard bits
- **ILP timeout** ← I suspect this is the worse offender for larger tables

Optimality ?

In short : no

Many heuristics that potentially removes optimiality.

- Replace some binary variables with integers
- Prune the external loop ← This also could have an impact
- Set a maximum amount of guard bits
- **ILP timeout** ← I suspect this is the worse offender for larger tables

Negative numbers in $\frac{1}{x}$?

Hardware operator raises a flag when input is negative, and outputs $\frac{1}{\sqrt{|x|}}$.

FMA that computes the square of the seed will compute $0 - a \times b$ instead of $0 + a \times b$ if the flag is raised.

Subnormals

In hardware

No subnormal output possible (as $\frac{1}{\sqrt{x}}$ reduces the exponent)
Could renormalise the subnormal input (costs a shifter, would probably add a cycle). I think it is worth it, but the µm^2 is expensive.

In software

Will create problems for $\frac{1}{x}$:

- Iterations are not done with range reduction, can create issues.
- Double rounding can happen on subnormal outputs[1]. Not sure if the solution will still work in my case

---

[1] JM Muller, *Avoiding double roundings in scaled Newton-Raphson division*, ASILOMAR (2013)

$\frac{1}{x}$ and the last seed before a power of 2

The input $F = $`0b111...111` was removed from the seed table generation program, as the software iterations do not result in a correctly rounded $\frac{1}{x}$.

---

[1] M. Cornea-Hasegan, R. Golliver, P. Markstein, *Correctness proofs outline for Newton-Raphson based floating-point divide and square root algorithm*, 1999 (ARITH)

$\frac{1}{x}$ and the last seed before a power of 2

The input $F =$`0b111...111` was removed from the seed table generation program, as the software iterations do not result in a correctly rounded $\frac{1}{x}$.
Previously studied [1] as a tricky case.

---

[1] M. Cornea-Hasegan, R. Golliver, P. Markstein, *Correctness proofs outline for Newton-Raphson based floating-point divide and square root algorithm*, 1999 (ARITH)

$\frac{1}{x}$ and the last seed before a power of 2

The input $F =$`0b111...111` was removed from the seed table generation program, as the software iterations do not result in a correctly rounded $\frac{1}{x}$.
Previously studied [1] as a tricky case.
The fix is probably going to be a conditional move in software when correct rounding is needed.

---

[1] M. Cornea-Hasegan, R. Golliver, P. Markstein, *Correctness proofs outline for Newton-Raphson based floating-point divide and square root algorithm*, 1999 (ARITH)

FP16 $\frac{1}{\sqrt{a}}$ CR

```
$o \Leftarrow$ seed($a$) (c0)
```

Latency : 1 cycle
Throughput : 16 op/cycle

FP32 $\frac{1}{\sqrt{a}}$ FR

```
$y_0 \Leftarrow \text{seed}(a)$ (c0)
$h_0 \Leftarrow 0.5 \times y_0$ (c1)
$g_0 \Leftarrow a \times y_0$ (c2)
$r_0 \Leftarrow -g_0 \times h_0 + 0.5$ (c6)
$o \Leftarrow r_0 \times y_0 + y_0$ (c10)
```

Latency : 14 cycles
Throughput : 4 op/cycle

FP16 $\sqrt{a}$ err < 1.4ulp[1]

$y_0 \Leftarrow \text{seed}(a)$ (c0)
$o \Leftarrow a \times y_0$ (c1)

Latency : 5 cycles
Throughput : 16 op/cycle

---

[1] $1.4 \approx \sqrt{2}$: Coincidence ? Probably

FP16 $\sqrt{a}$ CR

$y_0 \Leftarrow \text{seed}(a)$ (c0)
$g_0 \Leftarrow a \times y_0$ (c1)
$h_0 \Leftarrow 0.5 \times y_0$ (c2)
$e_0 \Leftarrow -g_0 \times g_0 + a$ (c5)
$o \Leftarrow e_0 \times h_0 + g_0$ (c9)

Latency : 13 cycles
Throughput : 4 op/cycle

---

[1] $1.4 \approx \sqrt{2}$: Coincidence ? Probably

FP32 $\sqrt{a}$ FR

$y_0 \Leftarrow \text{seed}(a)$ (c0)
$h_0 \Leftarrow 0.5 \times y_0$ (c1)
$g_0 \Leftarrow a \times y_0$ (c2)
$r_0 \Leftarrow -g_0 \times h_0 + 0.5$ (c6)
$o \Leftarrow r_0 \times g_0 + g_0$ (c10)

Latency : 14 cycles
Throughput : 4 op/cycle

FP32 $\sqrt{a}$ CR

$y_0 \Leftarrow \text{seed}(a)$ (c0)
$h_0 \Leftarrow 0.5 \times y_0$ (c1)
$g_0 \Leftarrow a \times y_0$ (c2)
$r_0 \Leftarrow -g_0 \times h_0 + 0.5$ (c6)
$g_1 \Leftarrow r_0 \times g_0 + g_0$ (c10)
$h_1 \Leftarrow r_0 \times h_0 + h_0$ (c11)
$e_1 \Leftarrow -g_1 \times g_1 + a$ (c14)
$o \Leftarrow e_1 \times h_1 + g_1$ (c18)

Latency : 22 cycles
Throughput : 2.3 op/cycle

FP16 $\frac{1}{a}$ err < 1.4ulp

$y_0 \Leftarrow \text{seed}(a)$ (c0)
$o \Leftarrow y_0 \times y_0$ (c1)

Latency : 5 cycles
Throughput : 16 op/cycle

FP16 $\frac{1}{a}$ CR

$y_0 \Leftarrow \text{seed}(a)$ (c0)
$g_0 \Leftarrow y_0 \times y_0$ (c1)
$e_0 \Leftarrow -g_0 \times a + 1$ (c5)
$o \Leftarrow e_0 \times g_0 + g_0$ (c9)

Latency : 13 cycles
Throughput : 5.3 op/cycle

FP32 $\frac{1}{a}$ FR

$$
\begin{aligned}
&y_0 \Leftarrow \mathrm{seed}(a)\ (\mathrm{c0})\\
&x_0 \Leftarrow y_0 \times y_0\ (\mathrm{c1})\\
&r_0 \Leftarrow -a \times x_0 + 1\ (\mathrm{c5})\\
&o \Leftarrow r_0 \times x_0 + x_0\ (\mathrm{c9})
\end{aligned}
$$

Latency : 13 cycles
Throughput : 5.3 op/cycle

FP32 $\frac{1}{a}$ CR

$y_0 \Leftarrow \mathrm{seed}(a)$ (c0)
$x_0 \Leftarrow y_0 \times y_0$ (c1)
$r_0 \Leftarrow -a \times x_0 + 1$ (c5)
$x_1 \Leftarrow r_0 \times x_0 + x_0$ (c9)
$r_1 \Leftarrow -a \times x_1 + 1$ (c13)
$o \Leftarrow r_1 \times x_1 + x_1$ (c17)

Latency : 21 cycles
Throughput : 3.2 op/cycle