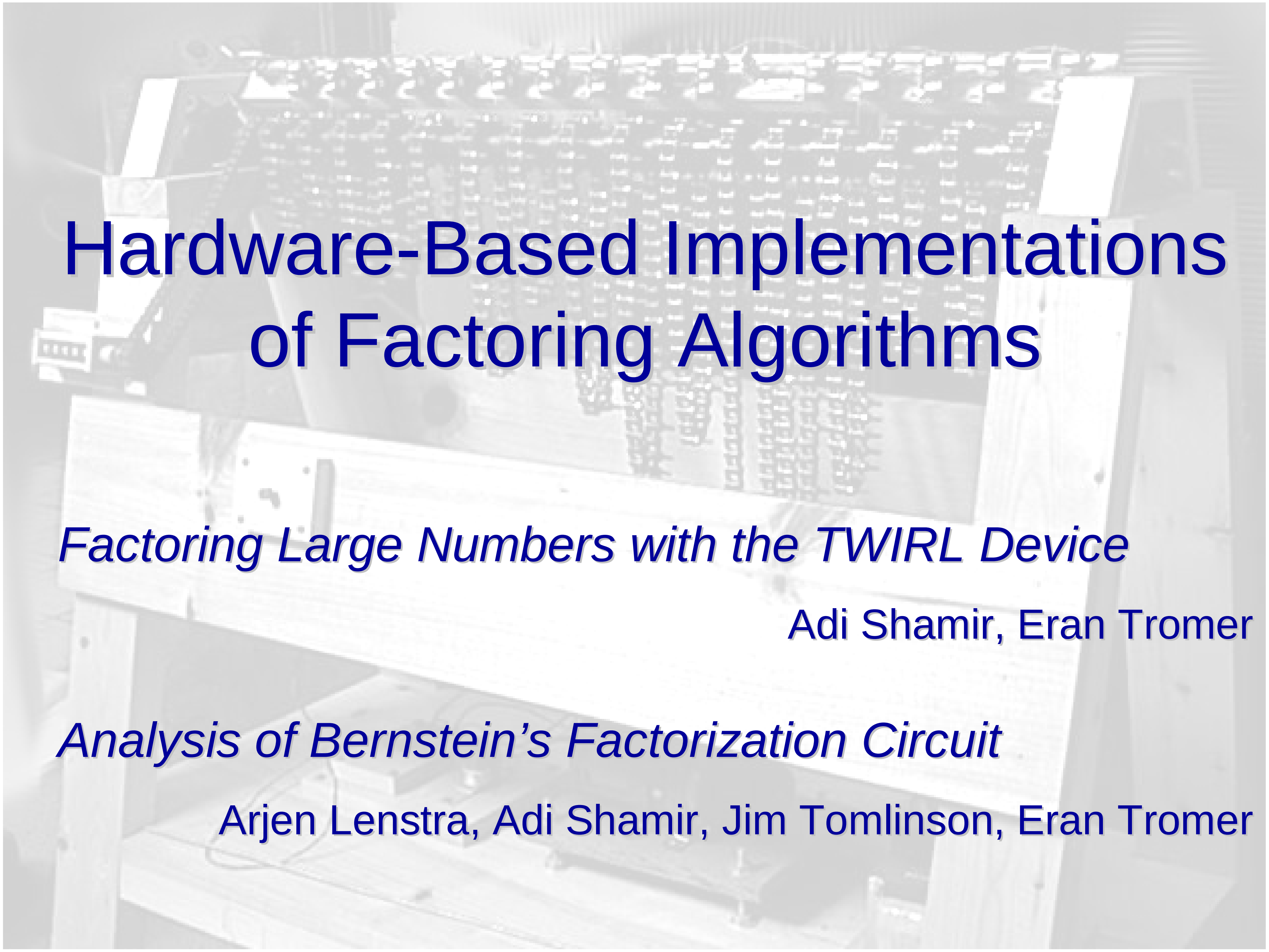


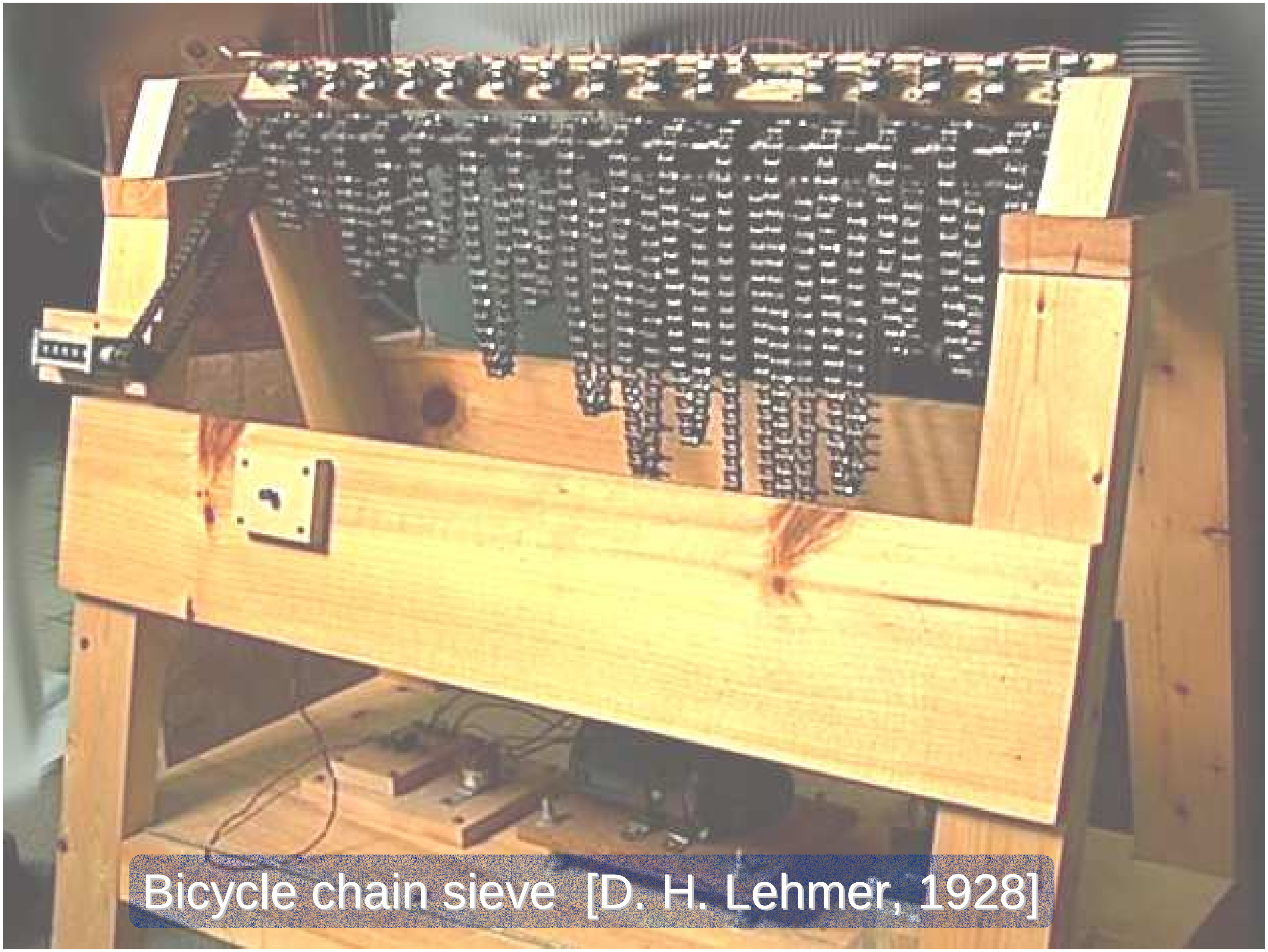
Hardware-Based Implementations of Factoring Algorithms

Factoring Large Numbers with the TWIRL Device

Adi Shamir, Eran Tromer

Analysis of Bernstein's Factorization Circuit

Arjen Lenstra, Adi Shamir, Jim Tomlinson, Eran Tromer


Bicycle chain sieve [D. H. Lehmer, 1928]

The Number Field Sieve Integer Factorization Algorithm

- Best algorithm known for factoring large integers.
- Subexponential time, subexponential space.

- Successfully factored a 512-bit RSA key (hundreds of workstations running for many months).
- Record: 530-bit integer (RSA-160, 2003).

- Factoring 1024-bit: previous estimates were **trillions of $×year**.
- Our result: a hardware implementation which can factor 1024-bit composites at a cost of about **10M $×year**.

NFS – main parts

- Relation collection (sieving) step:
  Find many integers satisfying a certain (rare) property.

- Matrix step:
  Find an element from the kernel of a huge but sparse matrix.

Previous works: 1024-bit sieving

Cost of completing all sieving in 1 year:

- Traditional PC-based: [Silverman 2000]
  100M PCs with 170GB RAM each: $\$5\times10^{12}$
- TWINKLE: [Lenstra,Shamir 2000, Silverman 2000]*
  3.5M TWINKLEs and 14M PCs: ~ $\$10^{11}$
- Mesh-based sieving [Geiselmann,Steinwandt 2002]*
  Millions of devices, $\$10^{11}$ to $\$10^{10}$ (if at all?)
  Multi-wafer design – feasible?

- New device: $10M

Previous works: 1024-bit matrix step

Cost of completing the matrix step in 1 year:

- Serial: [Silverman 2000]
  19 years and 10,000 interconnected Crays.
- Mesh sorting [Bernstein 2001, LSTT 2002]
  273 interconnected wafers – feasible?!
  $4M and 2 weeks.

- New device: $0.5M

Review: the Quadratic Sieve

To factor n:

- Find “random” r_1, r_2 such that $r_1^2 \equiv r_2^2 \pmod{n}$
- Hope that $\gcd(r_1 - r_2, n)$ is a nontrivial factor of n.

How?

- Let $f_1(a) = (a + \lfloor n^{1/2} \rfloor)^2 - n$
  $f_2(a) = (a + \lfloor n^{1/2} \rfloor)^2$
- Find a nonempty set $S \subset \mathbb{Z}$ such that

  $$\prod_{a \in S} f_1(a) = r_1^2 \quad , \quad \prod_{a \in S} f_2(a) = r_2^2$$

  over $\mathbb{Z}$ for some $r_1, r_2 \in \mathbb{Z}$.
- $r_1^2 \equiv r_2^2 \pmod{n}$

The Quadratic Sieve (cont.)

How to find S such that $\prod_{a\in S} f_1(a)$ is a square?

Look at the factorization of $f_1(a)$:

	$f_1(0)=102$	$=$	2	3					17		
→	$f_1(1)=33$	$=$		3			11				
	$f_1(2)=1495$	$=$			5			13		23	
	$f_1(3)=84$	$=$	2^2	3		7					
→	$f_1(4)=616$	$=$	2^3			7	11				
	$f_1(5)=145$	$=$			5						29
→	$f_1(6)=42$	$=$	2	3		7					

$\vdots \quad \vdots$

$$\prod_{a\in\{1,4,6\}} f_1(a) = 2^4 \quad 3^2 \quad 5^0 \quad 7^2 \quad 11^2$$

This is a square, because all exponents are even.

The Quadratic Sieve (cont.)

How to find S such that $\prod_{a\in S} f_1(a)$ is a square?

- Consider only the $\pi(B)$ primes smaller than a bound B.

- Search for integers a for which $f_1(a)$ is B-smooth. For each such a, represent the factorization of $f_1(a)$ as a vector of b exponents:
  $f_1(a)=2^{e_1}\,3^{e_2}\,5^{e_3}\,7^{e_4}\cdots \quad\mapsto\quad (e_1,e_2,\ldots,e_b)$

Relation collection step

- Once $b+1$ such vectors are found, find a dependency modulo 2 among them. That is, find S such that
  $\prod_{a\in S} f_1(a)=2^{e_1}\,3^{e_2}\,5^{e_3}\,7^{e_4}\cdots$ where e_i all even.

Matrix step

Observations [Bernstein 2001]

- The matrix step involves multiplication of a single huge matrix (of size subexponential in n) by many vectors.
- On a single-processor computer, storage dominates cost yet is poorly utilized.
- Sharing the input: collisions, propagation delays.
- Solution: use a mesh-based device, with a small processor attached to each storage cell.
  Devise an appropriate distributed algorithm.
  Bernstein proposed an algorithm based on mesh sorting.
- Asymptotic improvement: at a given cost you can factor integers that are 1.17 longer, when cost is defined as

$$\underset{\textit{AT cost}}{\underset{\parallel}{\textit{throughput cost}}} = \textit{run time} \times \textit{construction cost}$$

Implications?

- The expressions for asymptotic costs have the form $e^{(\alpha+\underset{\sim}{o(1)})\cdot(\log n)^{1/3}\cdot(\log\log n)^{2/3}}$.
- Is it feasible to implement the circuits with current technology? For what problem sizes?
- Constant-factor improvements to the algorithm? Take advantage of the quirks of available technology?

- What about relation collection?

The Relation Collection Step

- Task:
  Find many integers a for which $f_1(a)$ is B-smooth (and their factorization).
- We look for a such that $p|f_1(a)$ for many large p:

$$\sum_{p\,:\,p|f_1(a)} \log p > T \approx \log f_1(a)$$

- Each prime p “hits” at arithmetic progressions:

$$\begin{aligned}\{a : p|f_1(a)\} &= \{a : f_1(a) \equiv 0 \pmod{p}\} \\ &= \bigcup_i \{r_i + kp : k \in \mathbb{Z}\}\end{aligned}$$

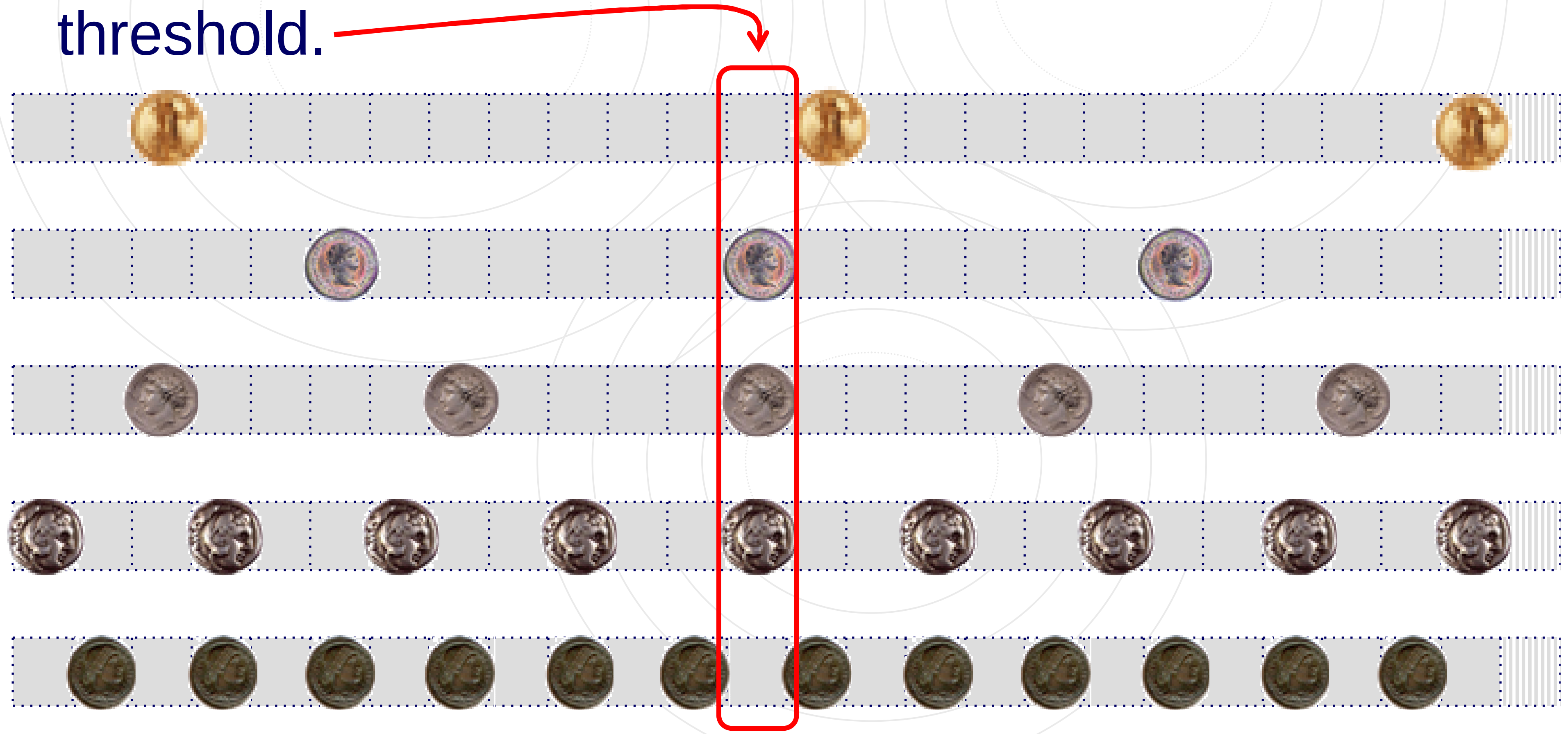
  where r_i are the roots modulo p of f_1.
  (there are at most $\deg(f_1)$ such roots, ~1 on average).

The Sieving Problem

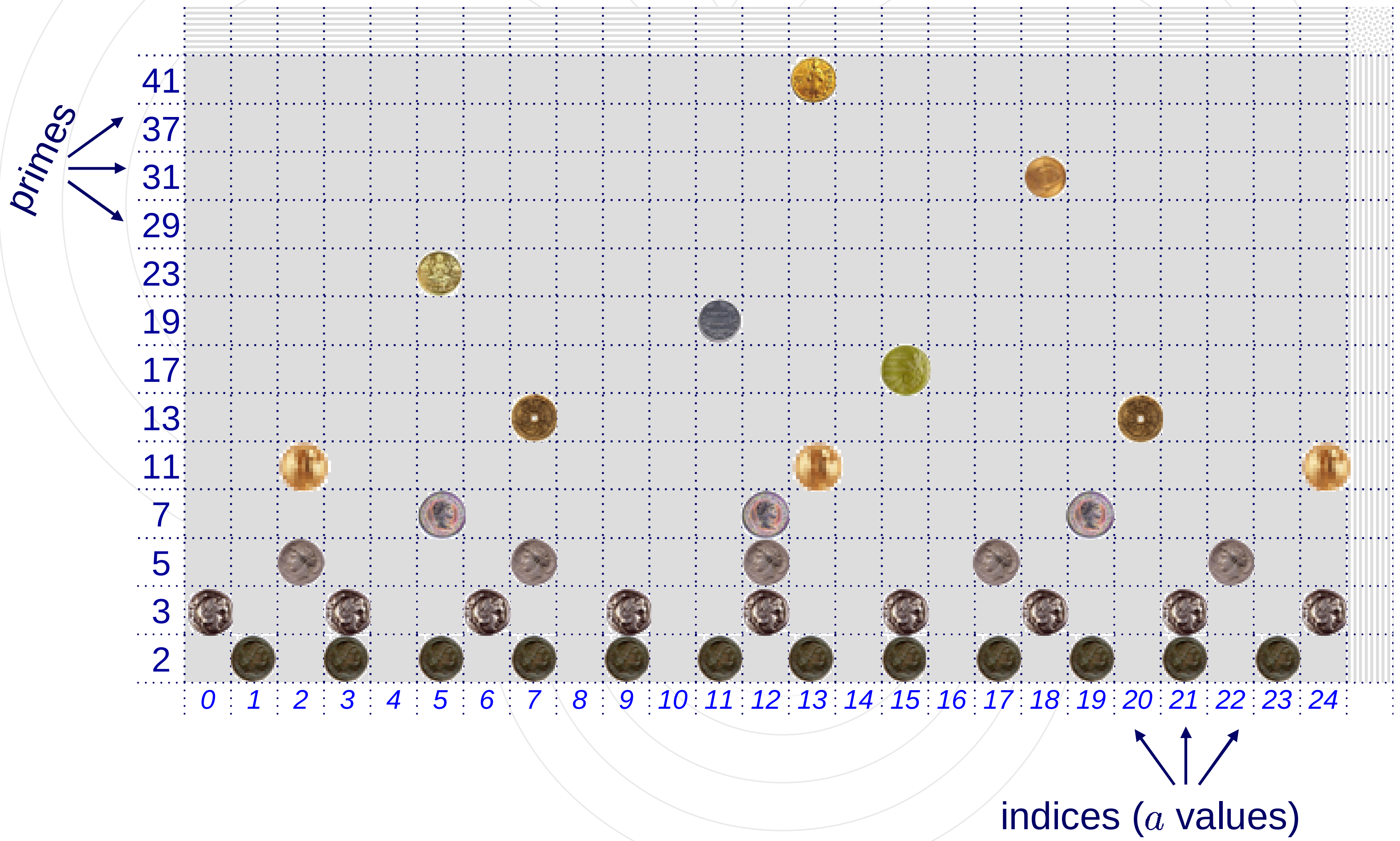
Input: a set of arithmetic progressions. Each progression has a prime interval p and value $\log p$.

(there is about one progression for every prime p smaller than 10^8)

Output: indices where the sum of values exceeds a threshold.

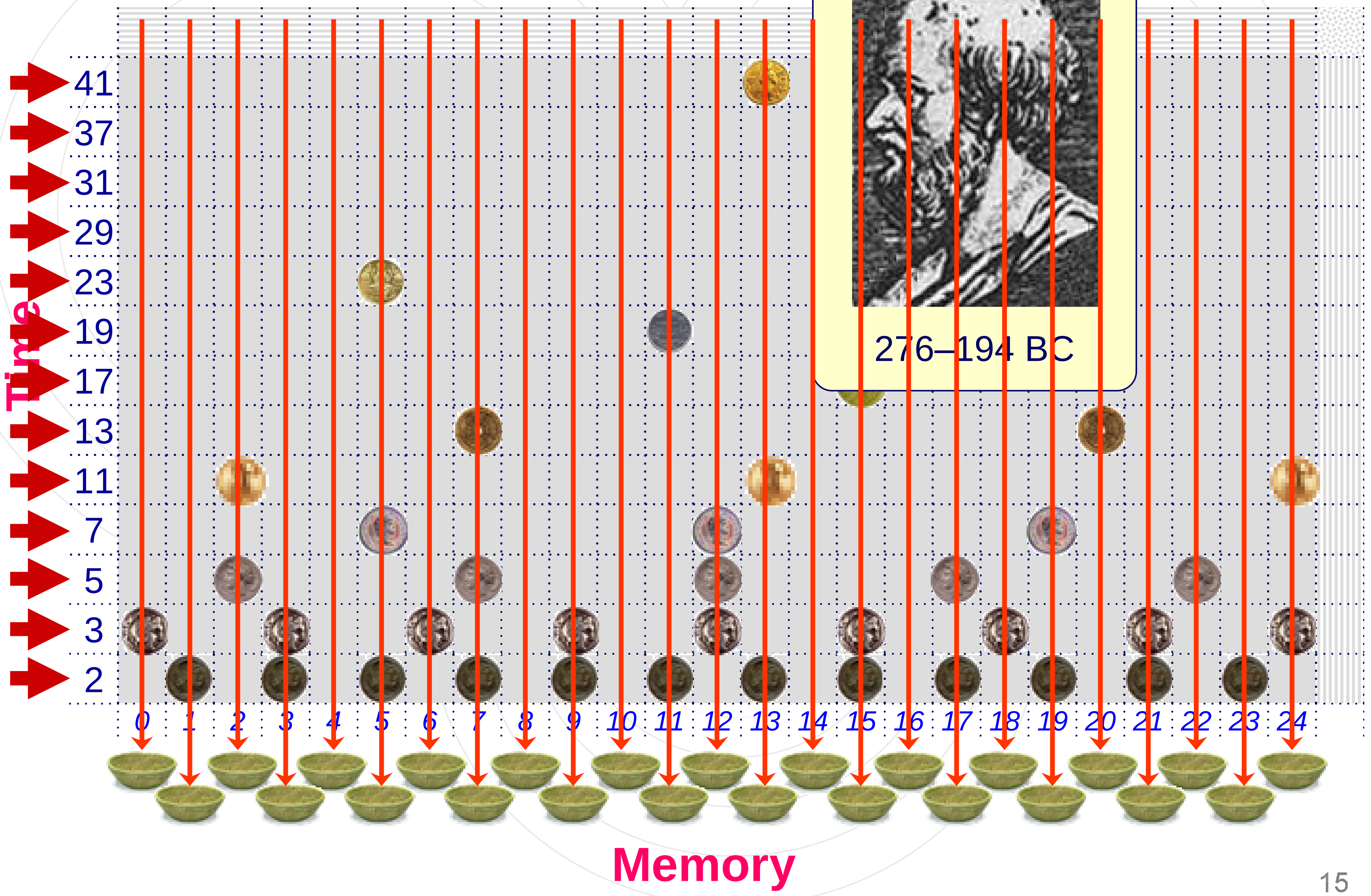


Three ways to sieve your numbers...



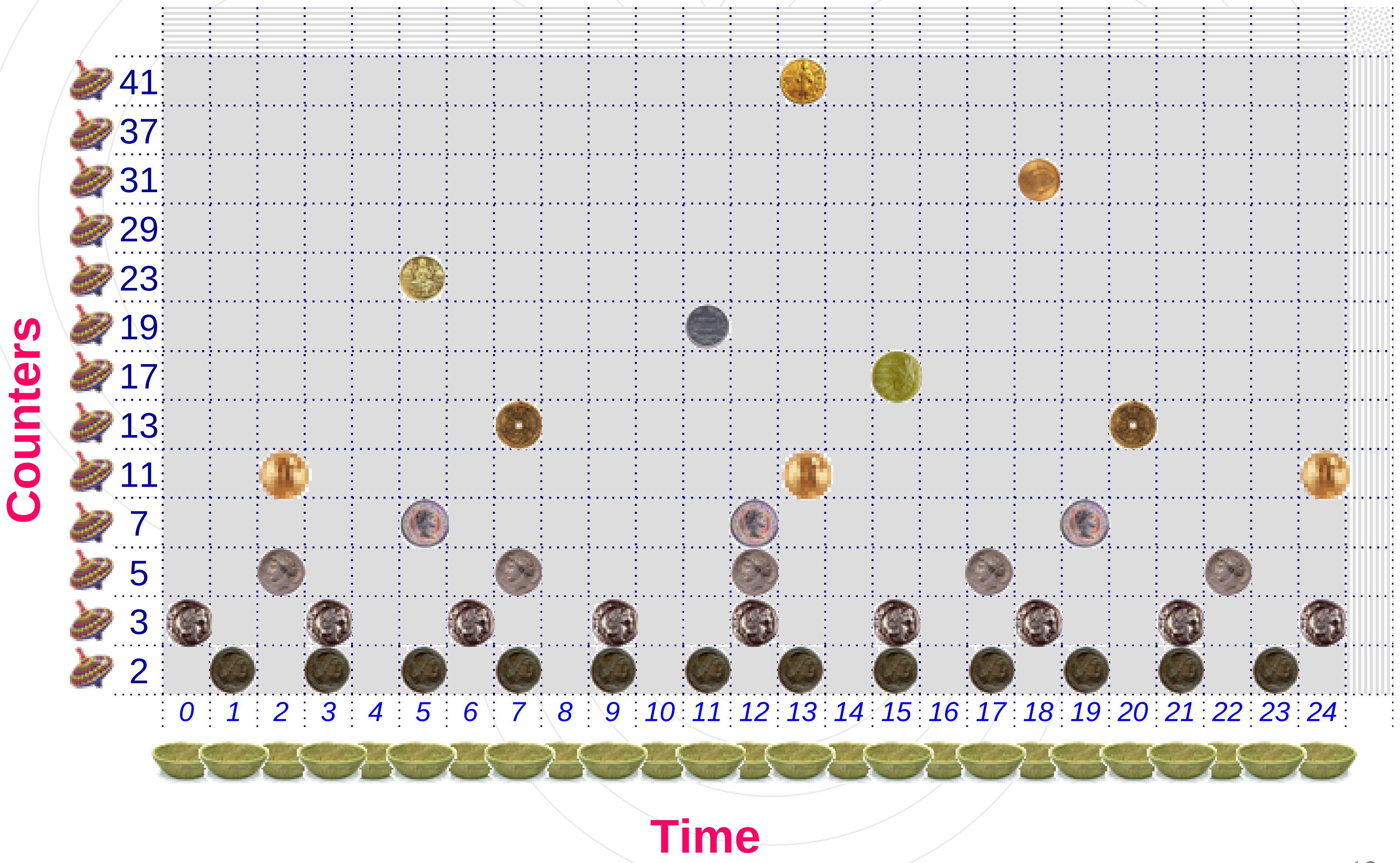
Serial sieving, à la Eratosthenes

One contribution per clock cycle.



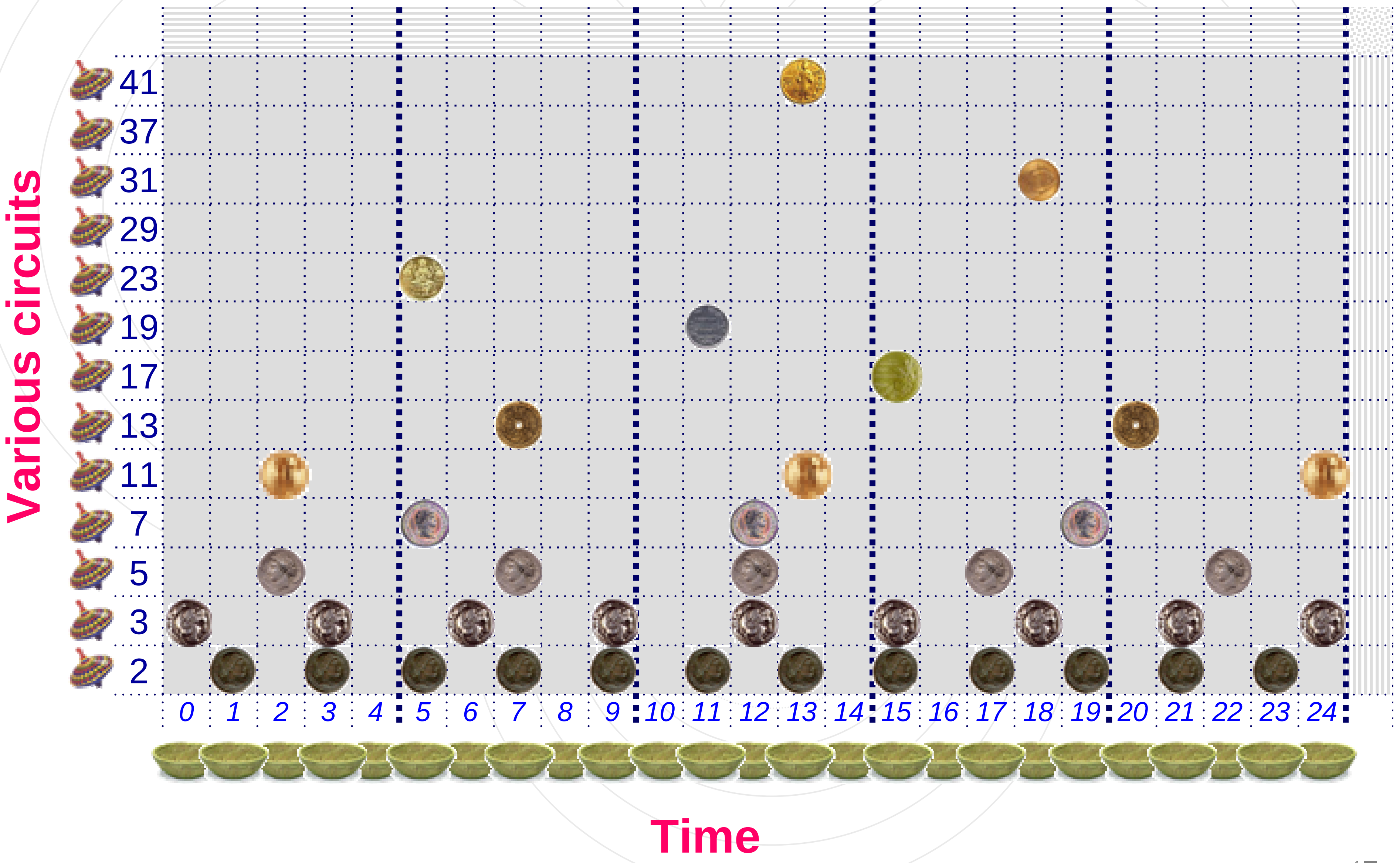
TWINKLE: time-space reversal

One index handled at each clock cycle.

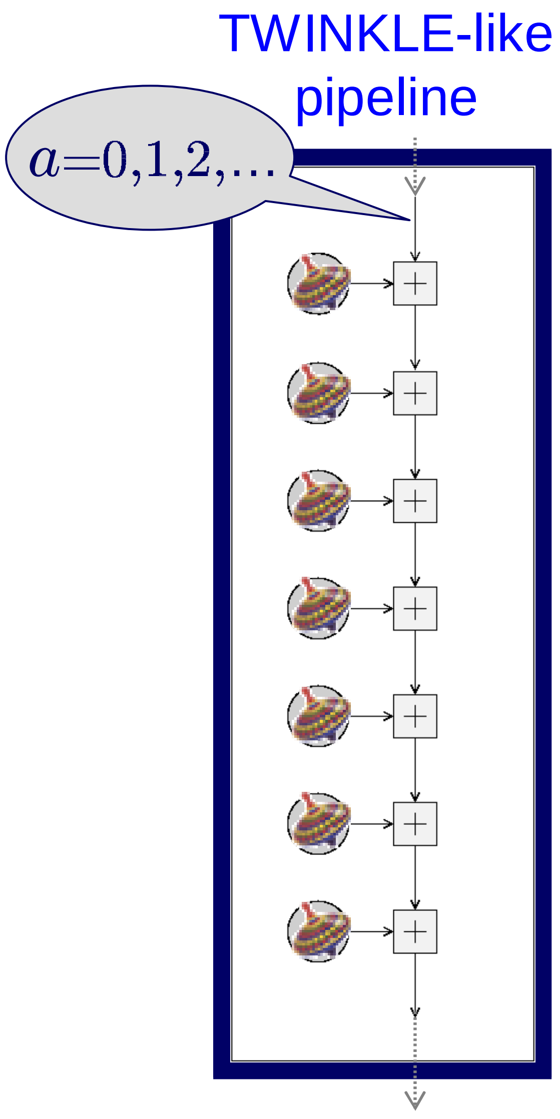


TWIRL: compressed time

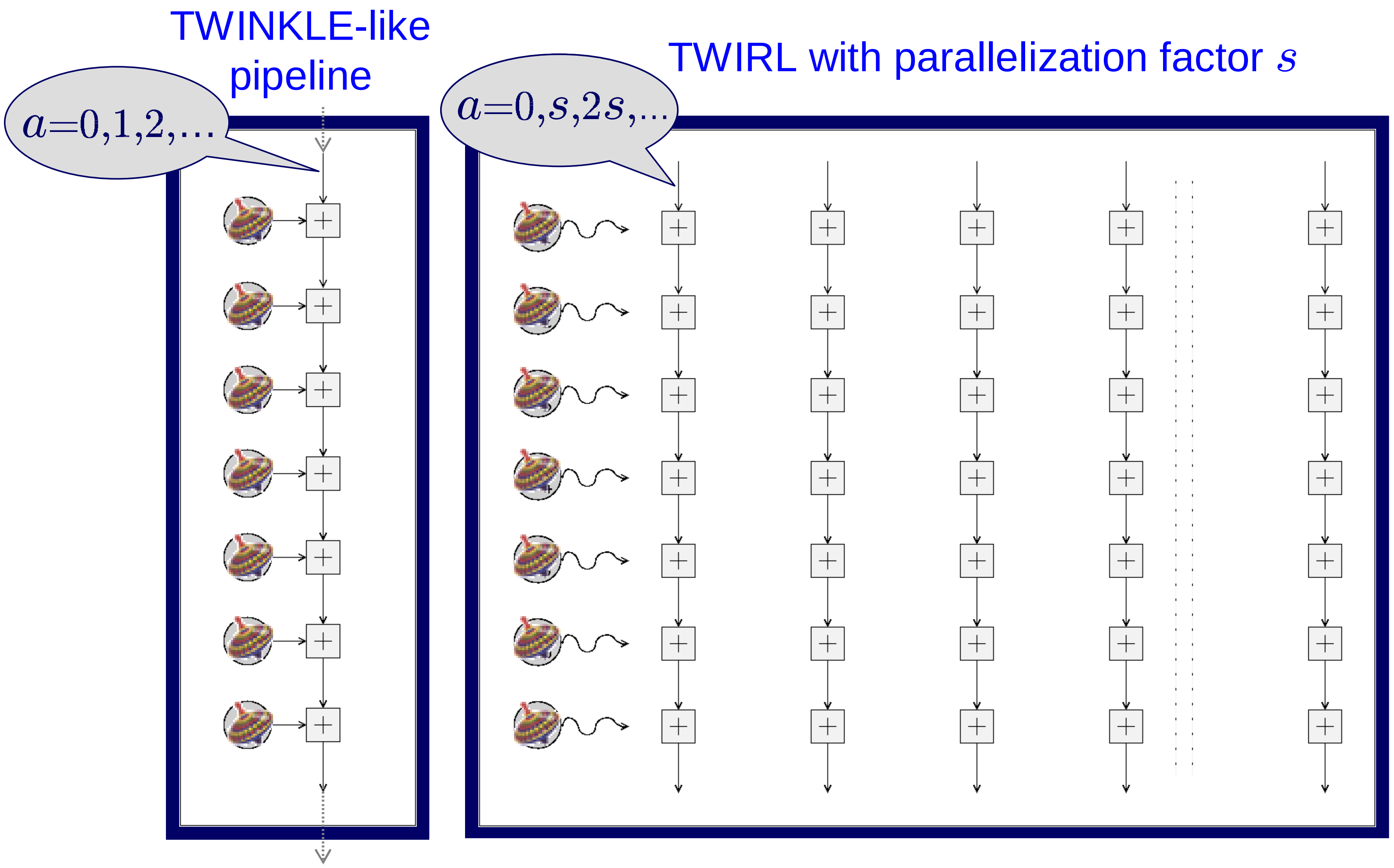
$s=5$ indices handled at each clock cycle. (real: $s=32768$)



Parallelization in TWIRL



Parallelization in TWIRL



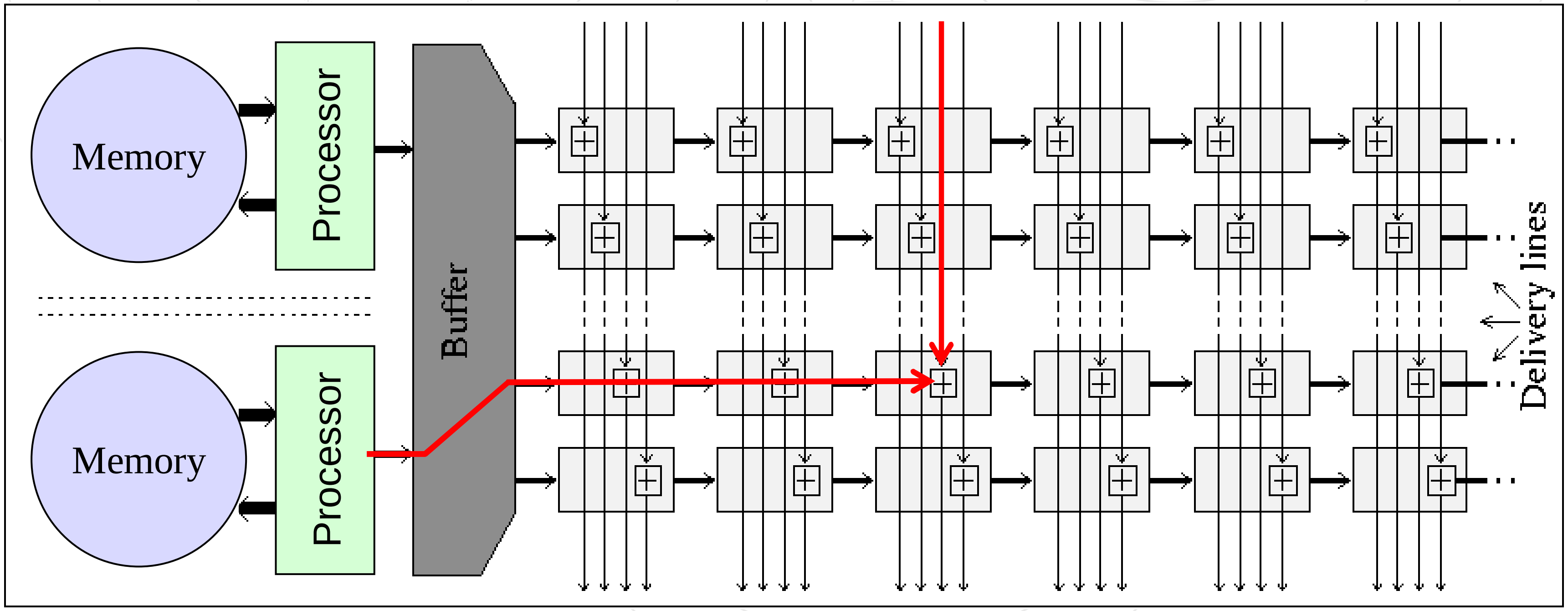
Heterogeneous design

- A progression of interval p_i makes a contribution every p_i/s clock cycles.
- There are a lot of large primes, but each contributes very seldom.
- There are few small primes, but their contributions are frequent.

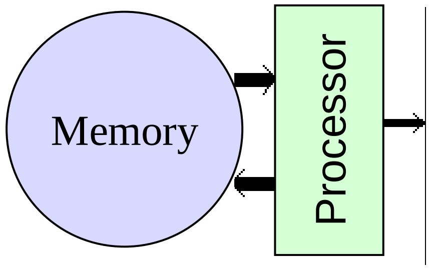
We place numerous “stations” along the pipeline. Each station handles progressions whose prime interval are in a certain range. Station design varies with the magnitude of the prime.

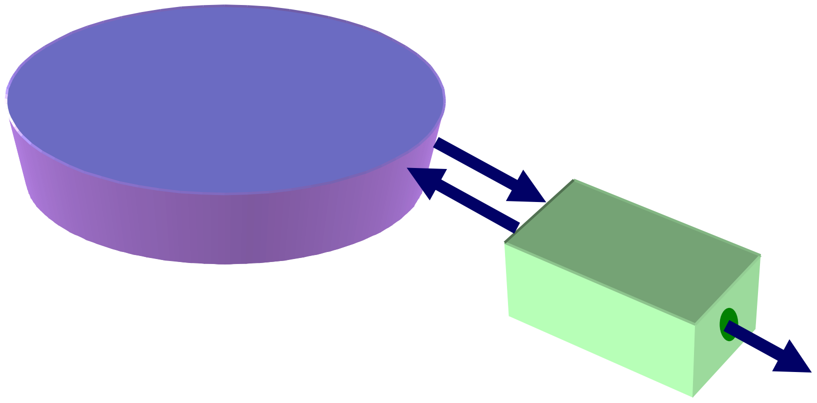
Example: handling large primes

- Primary consideration:
  efficient storage between contributions.
- Each memory+processor unit handle many progressions.
  It computes and sends contributions across the bus, where they are added at just the right time. Timing is critical.



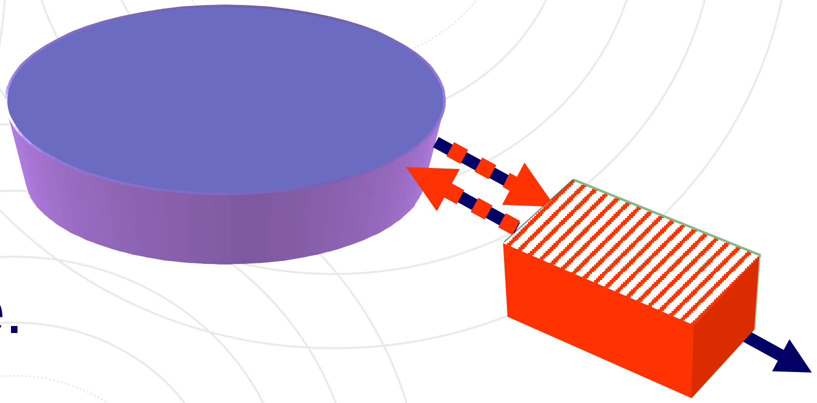
Handling large primes (cont.)





Handling large primes (cont.)

- The memory contains a list of events of the form (p_i,a_i), meaning "*a progression with interval p_i will make a contribution to index a_i*". Goal: simulate a priority queue.
- The list is ordered by increasing a_i.
- At each clock cycle:

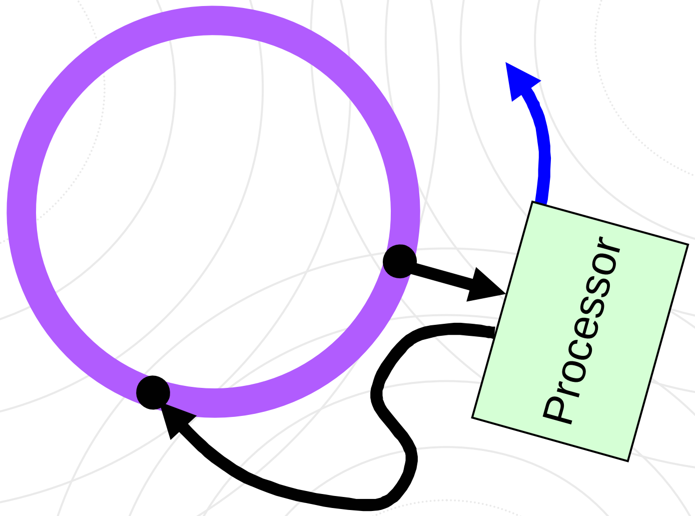


1. Read next event (p_i,a_i).
2. Send a $\log p_i$ contribution to line $a_i \pmod{s}$ of the pipeline.
3. Update $a_i \leftarrow a_i + p_i$
4. Save the new event (p_i,a_i) to the memory location that will be read just before index a_i passes through the pipeline.

- To handle collisions, slacks and logic are added.

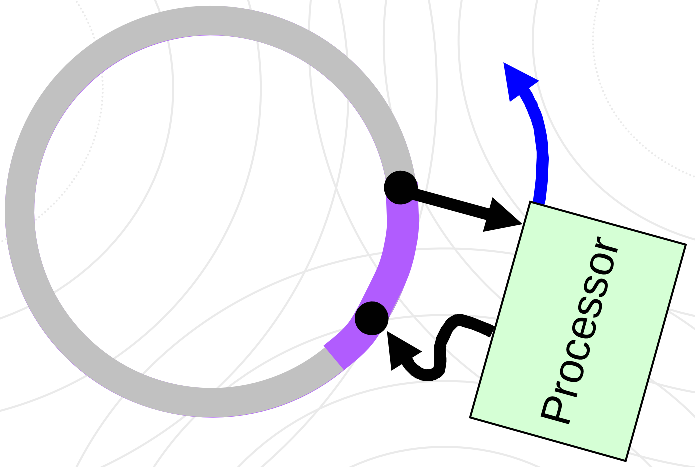
Handling large primes (cont.)

- The memory used by past events can be reused.
- Think of the processor as rotating around the cyclic memory:



Handling large primes (cont.)

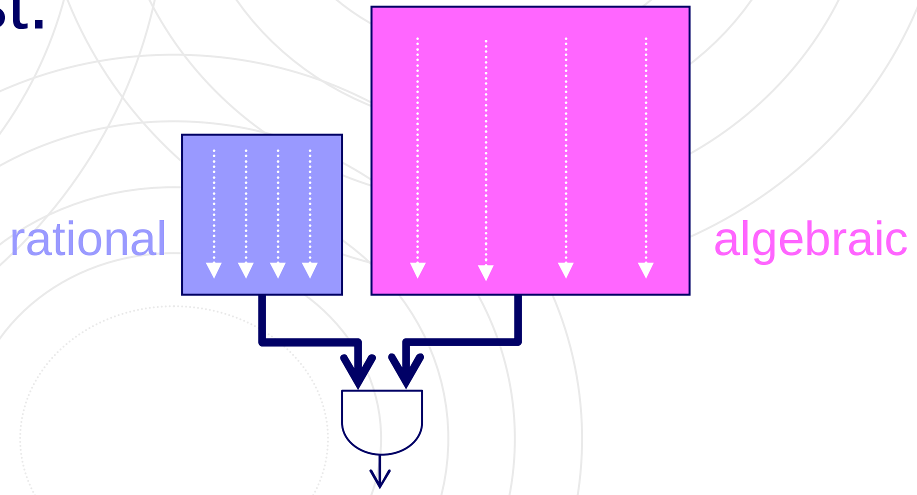
- The memory used by past events can be reused.
- Think of the processor as rotating around the cyclic memory:



- By appropriate choice of parameters, we guarantee that new events are always written just behind the read head.
- There is a tiny (1:1000) window of activity which is “twirling” around the memory bank. It is handled by an SRAM-based cache. The bulk of storage is handled in compact DRAM.

Rational vs. algebraic sieves

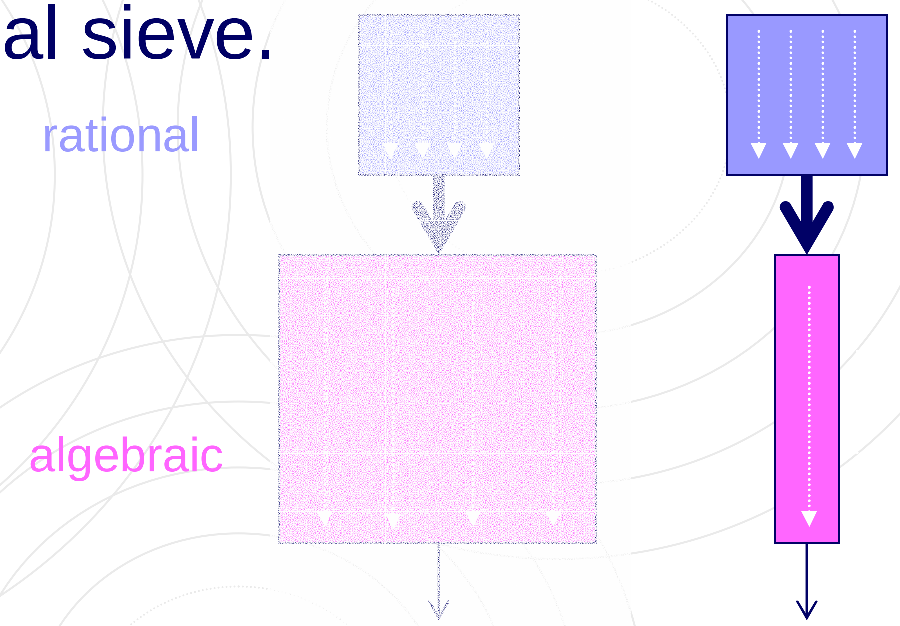
- We actually have two sieves: *rational* and *algebraic*. We are looking for the indices that accumulated enough value in <u>both</u> sieves.
- The algebraic sieve has many more progressions, and thus dominates cost.



- We cannot compensate by making s much larger, since the pipeline becomes very wide and the device exceeds the capacity of a wafer.

Optimization: cascaded sieves

- The algebraic sieve will consider only the indices that passed the rational sieve.



- In the algebraic sieve, we still scan the indices at a rate of thousands per clock cycle, but only a few of these have to be considered. $\Rightarrow$
  - much narrower bus
  - s increased to 32,768

Performance

- Asymptotically: speedup of

$$s \approx \tilde{\Theta}(\sqrt{\#\text{progressions}})$$

  compared to traditional sieving.
- For 512-bit composites:
  One silicon wafer full of TWIRL devices completes the sieving in under 10 minutes
  (0.00022sec per sieve line of length 1.8×10^{10}).

  1,600 times faster than best previous design.
- Larger composites?

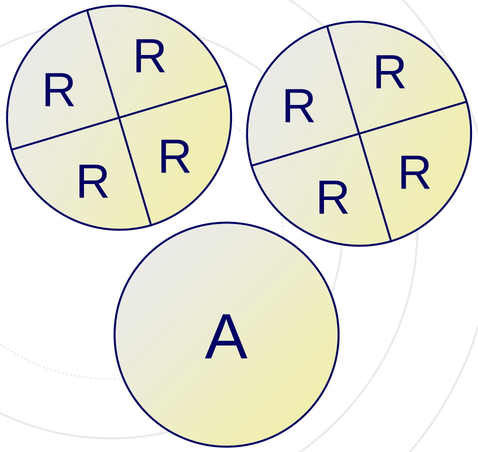
Estimating NFS parameters

- Predicting cost requires estimating the NFS parameters (smoothness bounds, sieving area, frequency of candidates etc.).
- Methodology: [Lenstra,Dodson,Hughes,Kortsmit,Leyland 2003]
  - Find good NFS polynomials for the RSA-1024 and RSA-768 composites.
  - Analyze and optimize relation yield for these polynomials according to smoothness probability functions.
  - Hope that cycle yield, as a function of relation yield, behaves similarly to past experiments.

1024-bit NFS sieving parameters

- Smoothness bounds:
  - Rational: 3.5×10^9
  - Algebraic: 2.6×10^{10}
- Region:
  - $a\in\{-5.5\times10^{14},\ldots,5.5\times10^{14}\}$
  - $b\in\{1,\ldots,2.7\times10^{8}\}$
  - Total: 3×10^{23} $(\times6/\pi^2)$

TWIRL for 1024-bit composites



- A cluster of 9 TWIRLS can process a sieve line (10^{15} indices) in 34 seconds.
- To complete the sieving in 1 year, use 194 clusters.
- Initial investment (NRE): ~$20M
- After NRE, total cost of sieving for a given 1024-bit composite: ~10M $×year (compared to ~1T $×year).

The matrix step

We look for elements from the kernel of a sparse matrix over $\mathrm{GF}(2)$. Using Wiedemann's algorithm, this is reduced to the following:

- Input: a sparse D x D binary matrix A and a binary D-vector v.
- Output: the first few bits of each of the vectors $Av, A^2v, A^3v, \ldots, A^Dv \pmod 2$.

- D is huge (e.g., $\approx 10^9$)

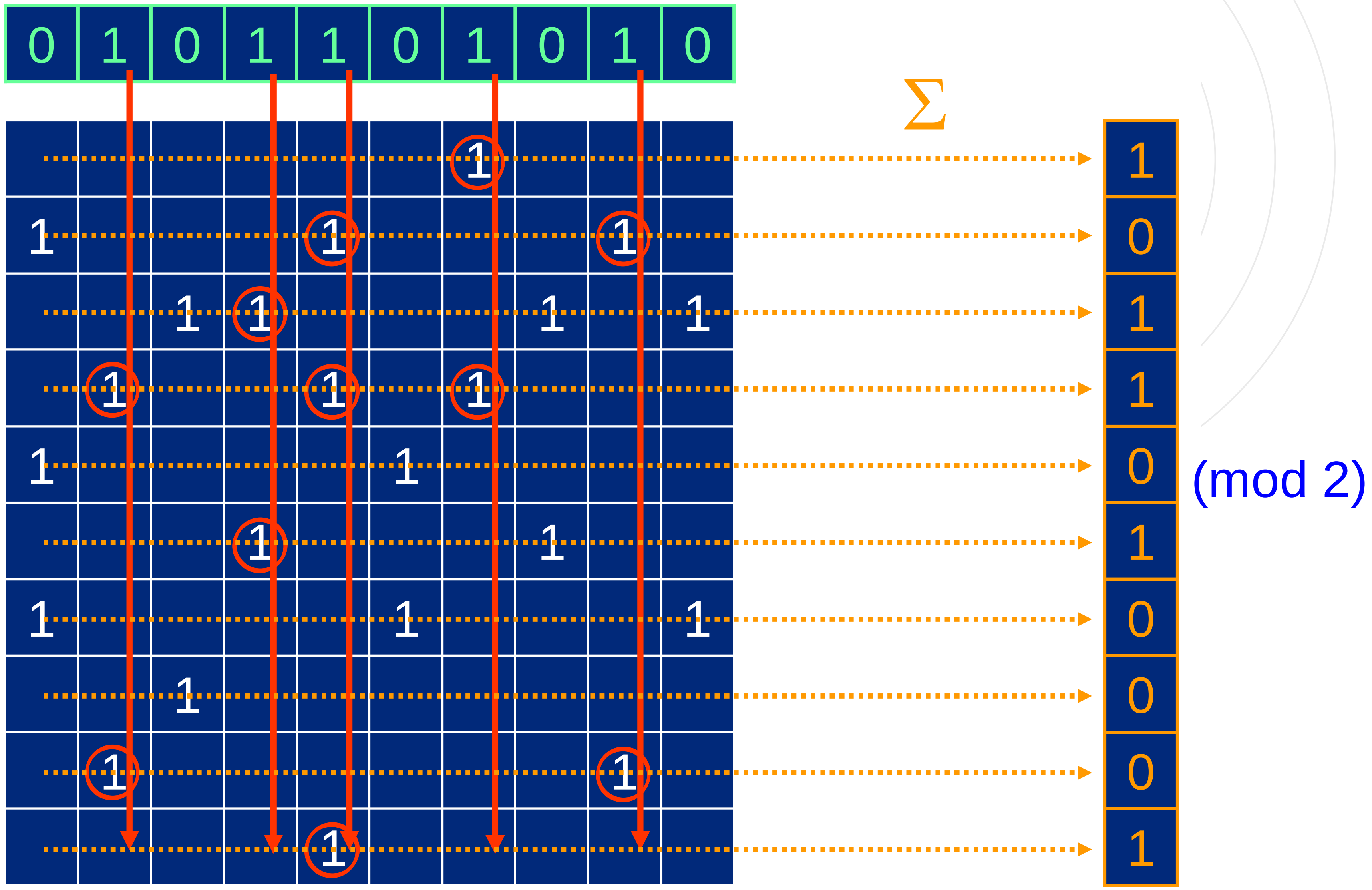
The matrix step (cont.)

- Bernstein proposed a parallel algorithm for sparse matrix-by-vector multiplication with asymptotic speedup

$$\tilde{\Theta}(\sqrt{D})$$

- Alas, for the parameters of choice it is inferior to straightforward PC-based implementation.
- We give a different algorithm which reduces the cost by a constant factor of 45,000.

Matrix-by-vector multiplication

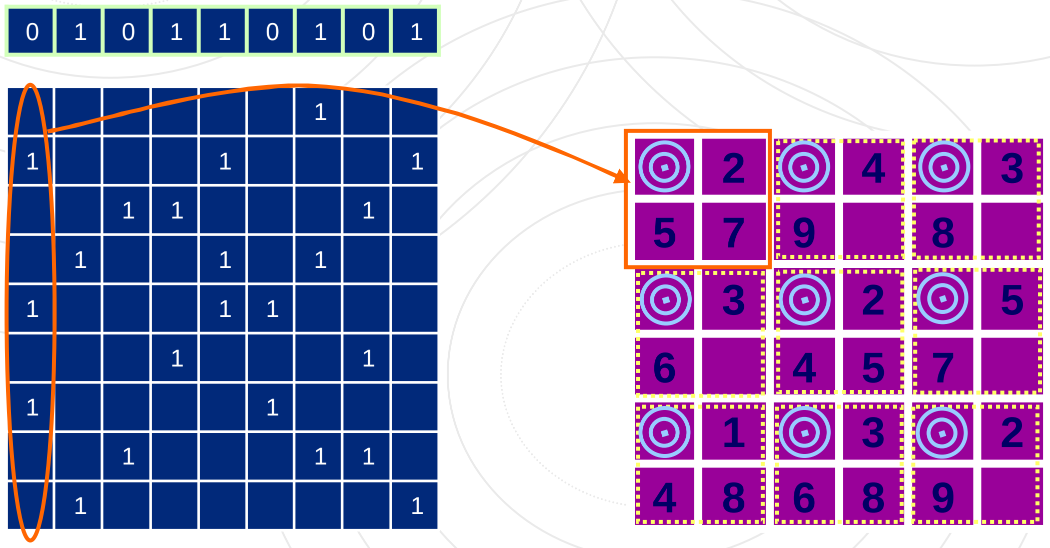


A routing-based circuit for the matrix step

[Lenstra,Shamir,Tomlinson,Tromer 2002]

Model: two-dimensional mesh, nodes connected to ≤ 4 neighbours.

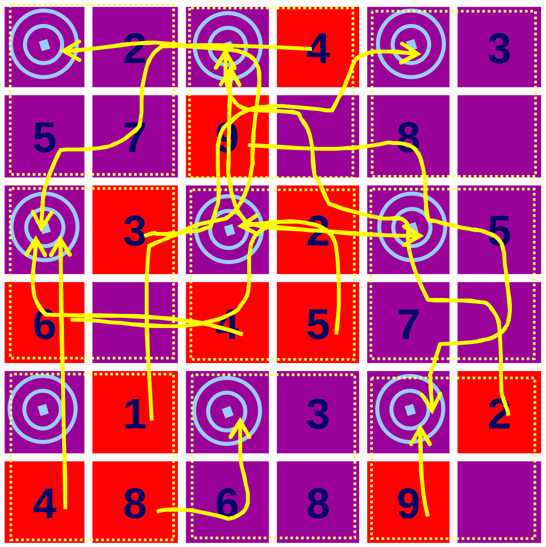
Preprocessing: load the non-zero entries of A into the mesh, one entry per node. The entries of each column are stored in a square block of the mesh, along with a “target cell” for the corresponding vector bit.



Operation of the routing-based circuit

To perform a multiplication:

- Initially the target cells contain the vector bits. These are locally broadcast within each block (i.e., within the matrix column).
- A cell containing a row index i that receives a “1” emits an $\boxed{i}$ value (which corresponds to a ○ at row i).
- Each $\boxed{i}$ value is routed to the target cell of the i-th block (which is collecting ○‘s for row i).
- Each target cell counts the number of $\boxed{i}$ values it received.
- That’s it! Ready for next iteration.



How to perform the routing?

Routing dominates cost, so the choice of algorithm (time, circuit area) is critical.

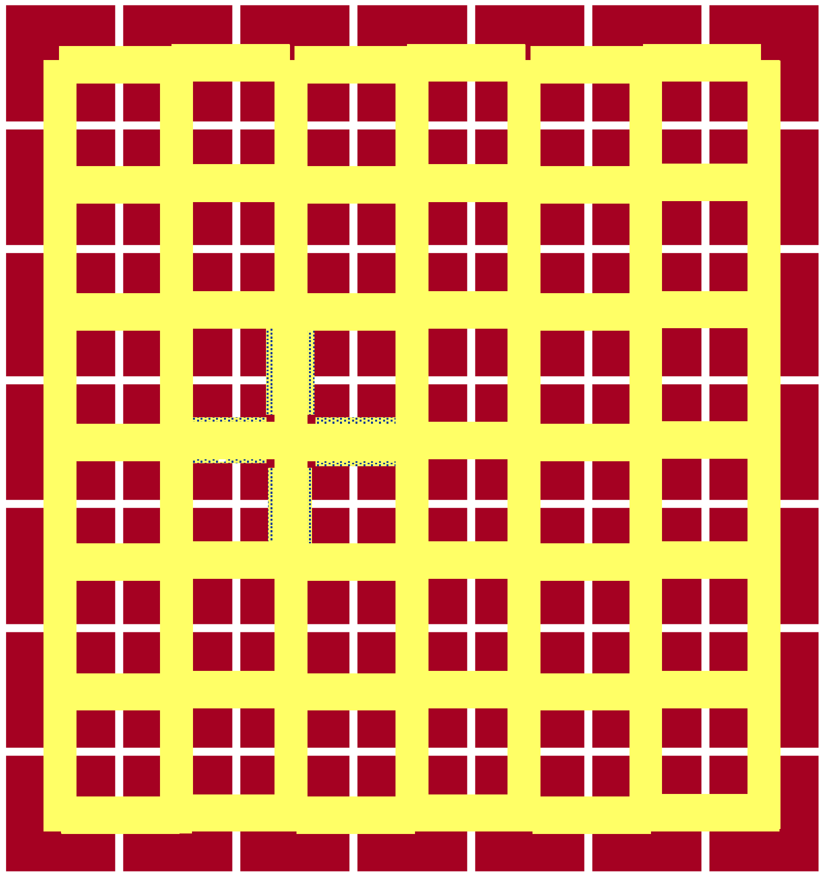
There is extensive literature about mesh routing. Examples:

- Bounded-queue-size algorithms
- Hot-potato routing
- Off-line algorithms

None of these are ideal.

Clockwise transposition routing on the mesh

- One packet per cell.
- Only pairwise compare-exchange operations (🟨).
- Compared pairs are swapped according to the preference of the packet that has the farthest to go *along this dimension*.
- Very simple schedule, can be realized implicitly by a pipeline.
- Pairwise annihilation.
- Worst-case: m^2
- Average-case: ?
- Experimentally: $2m$ steps suffice for random inputs – optimal.
- The point: m^2 values handled in time $O(m)$. [Bernstein]



Comparison to Bernstein’s design

1/12

- Time:
  A single routing operation (2*m* steps) vs. 3 sorting operations (8*m* steps each).
- Circuit area:
  - Only the i move; the matrix entries don’t.
  - Simple routing logic and small routed values
  - Matrix entries compactly stored in DRAM (~1/100 the area of “active” storage)

1/3

- Fault-tolerance
- Flexibility

Improvements

1/7

- Reduce the number of cells in the mesh (for small μ, decreasing #cells by a factor of μ decreases throughput cost by $\sim\mu^{1/2}$)

1/6

- Use Coppersmith's block Wiedemann

1/15

- Execute the separate multiplication chains of block Wiedemann simultaneously on one mesh (for small K, reduces cost by $\sim K$)

Compared to Bernstein's original design, this reduces the throughput cost by a constant factor of 45,000.

Implications for 1024-bit composites:

- Sieving step: ~10M $×year
  (including cofactor factorization).
- Matrix step: <0.5M $×year
- Other steps: unknown, but no obvious bottleneck.
- This relies on a hypothetical design and many approximations, but should be taken into account by anyone planning to use 1024-bit RSA keys.
- For larger composites (e.g., 2048 bit) the cost is impractical.

Conclusions

- 1024-bit RSA is less secure than previously assumed.
- Tailoring algorithms to the concrete properties of available technology can have a dramatic effect on cost.
- Never underestimate the power of custom-built highly-parallel hardware.