Faster discrete logarithms on FPGAs

Daniel J. Bernstein, Susanne Engels,
Tanja Lange, **Ruben Niederhagen**,
Christof Paar, Peter Schwabe,
and Ralf Zimmermann

TU/e Technische Universiteit **Eindhoven** University of Technology

ECC 2016 — İzmir, September 6th, 2016

What are FPGAs?

Field Programmable Gate Arrays (FPGAs):

FPGAs consist of

- programmable *logic blocks* and a
- programmable *routing fabric*.

What are FPGAs?

Field Programmable Gate Arrays (FPGAs):

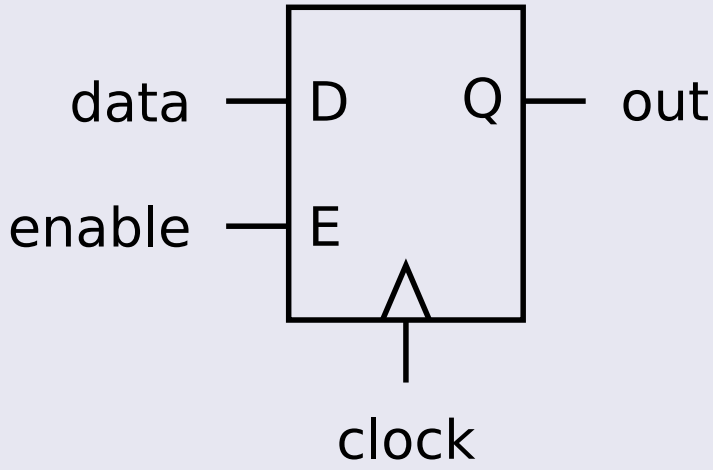
FPGAs consist of

- programmable *logic blocks* and a
- programmable *routing fabric*.

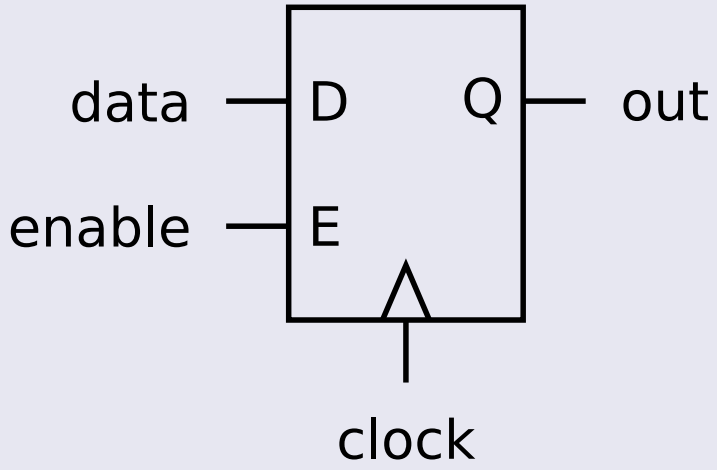
Types of Logic Blocks:

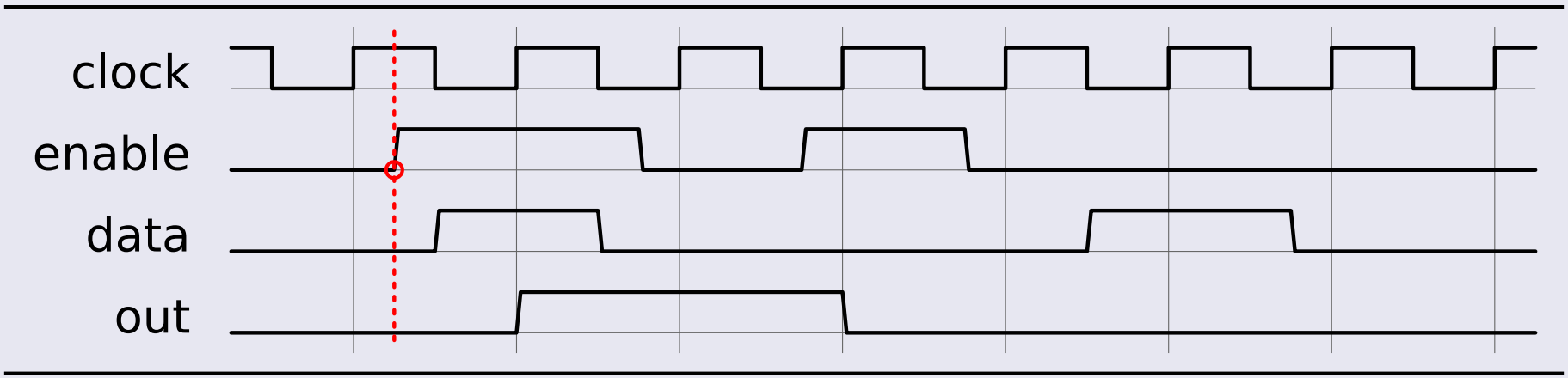
- Flip-flops (FF),
- lookup tables (LUTs),
- random access memory (block-RAM), and
- functional units: multiplier, multiplexer, memory controller, . . .

Flip-Flops: 1-bit register

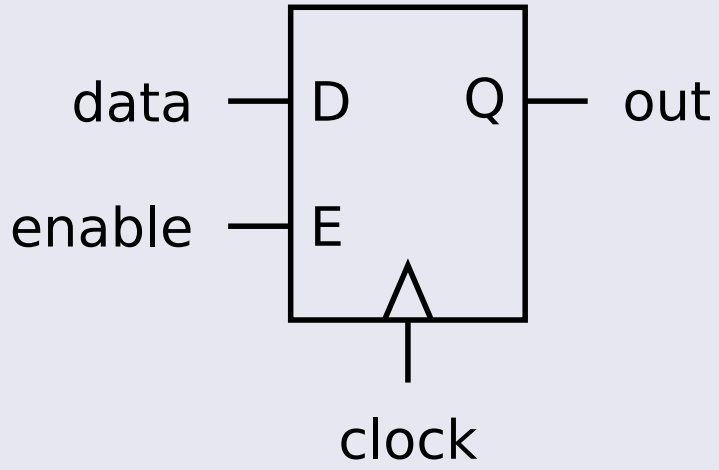


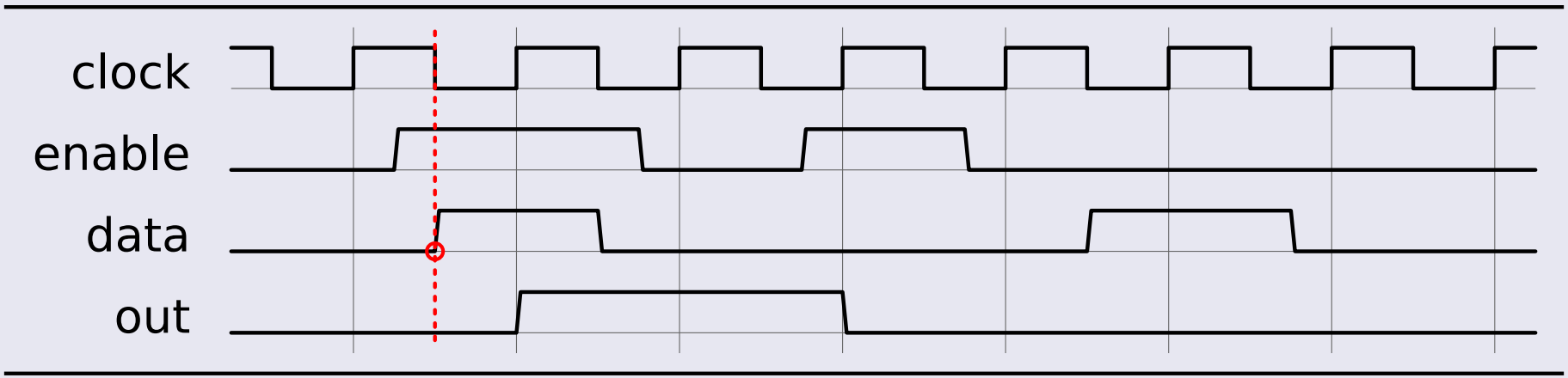
Flip-Flops: 1-bit register



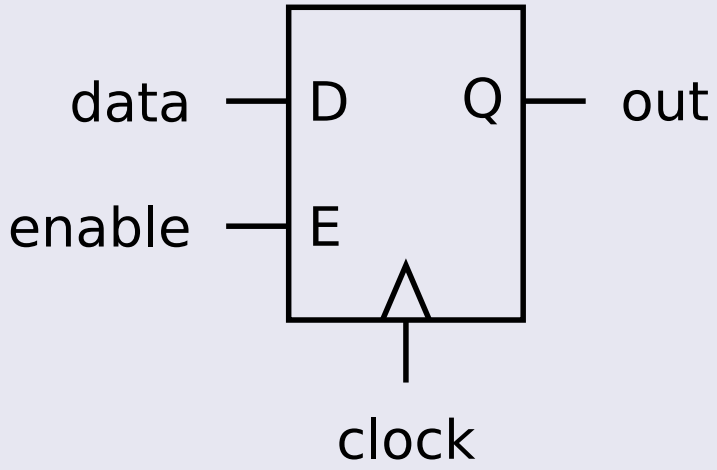


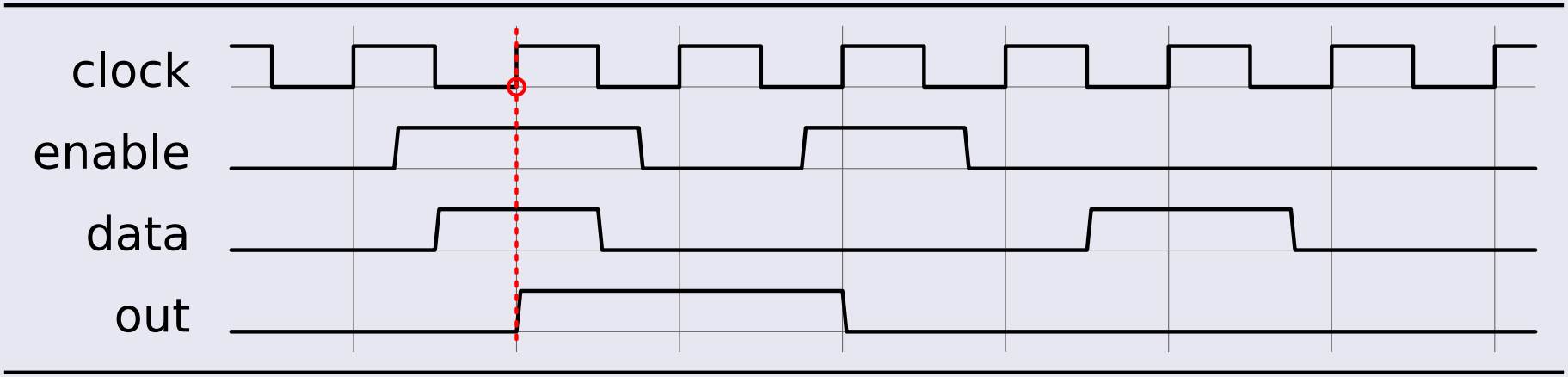
Flip-Flops: 1-bit register



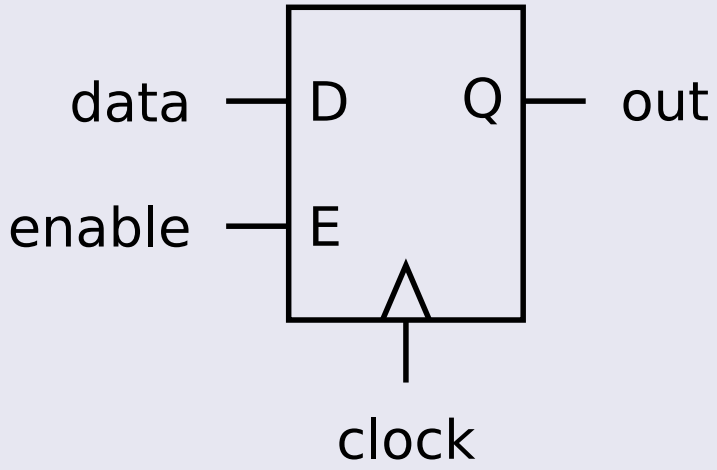


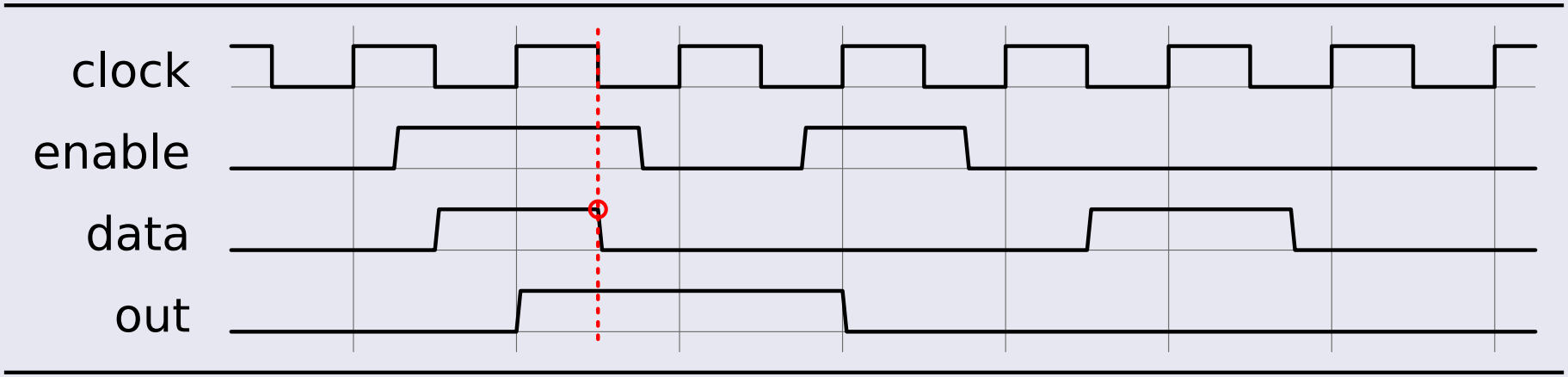
Flip-Flops: 1-bit register



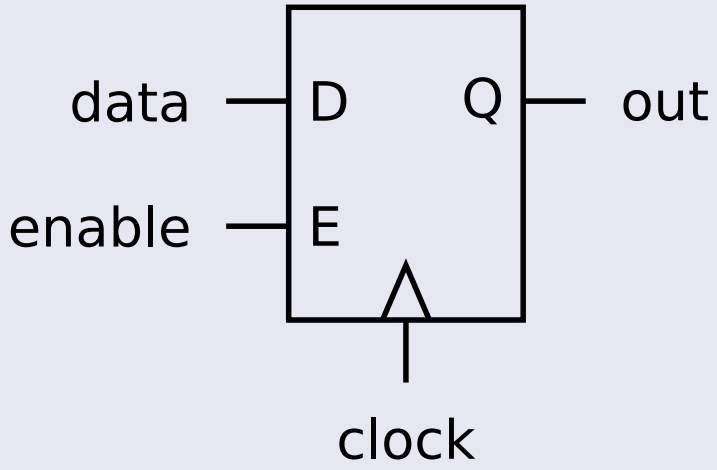


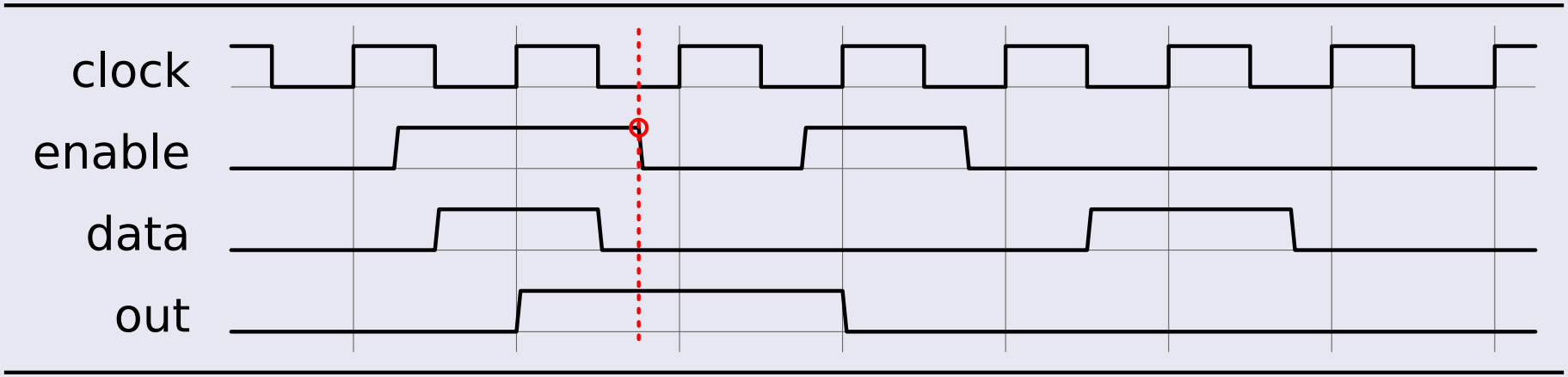
Flip-Flops: 1-bit register



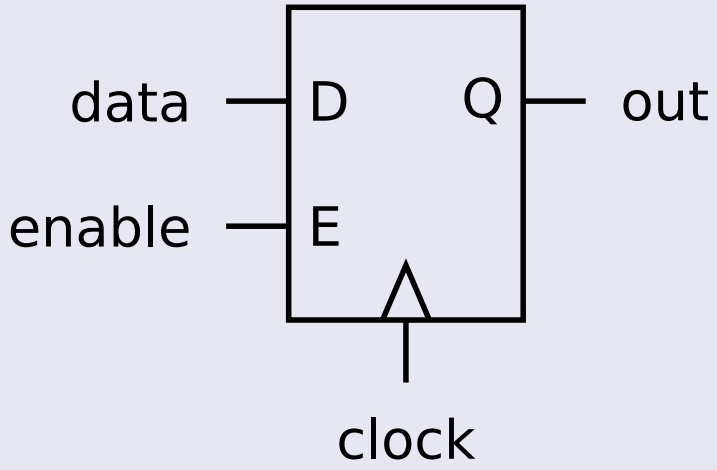


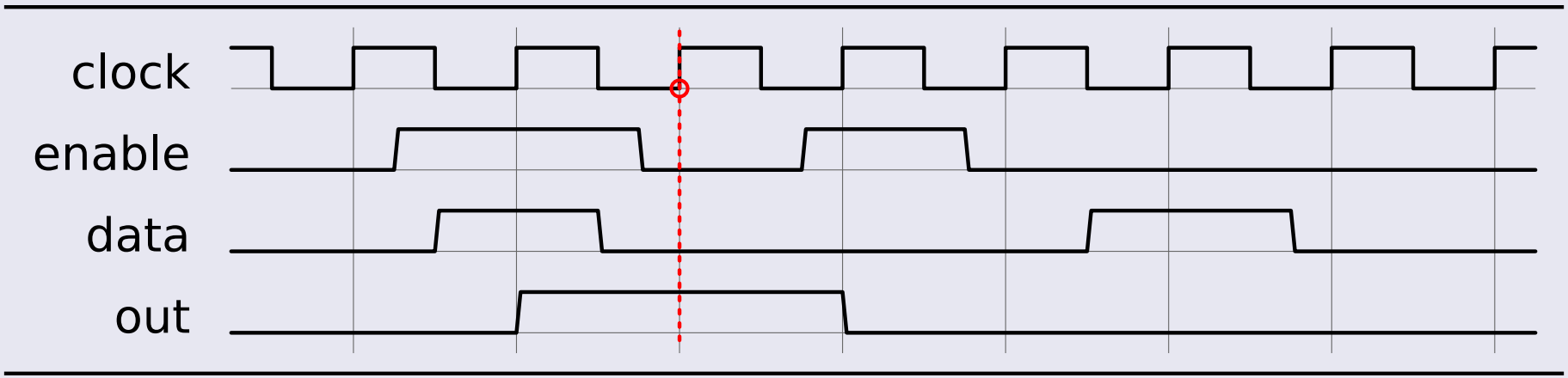
Flip-Flops: 1-bit register



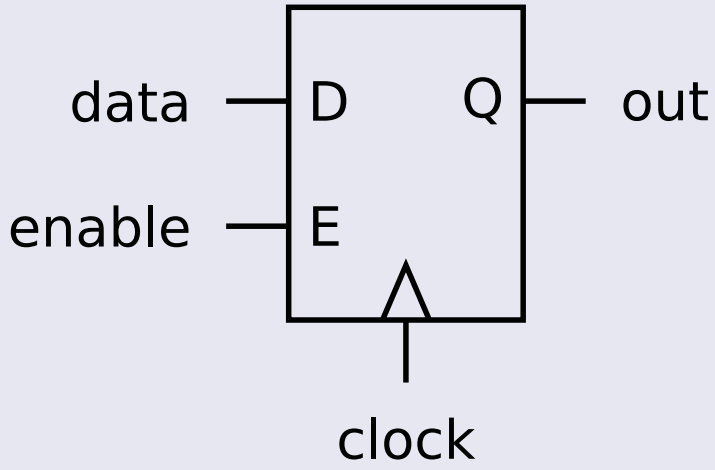


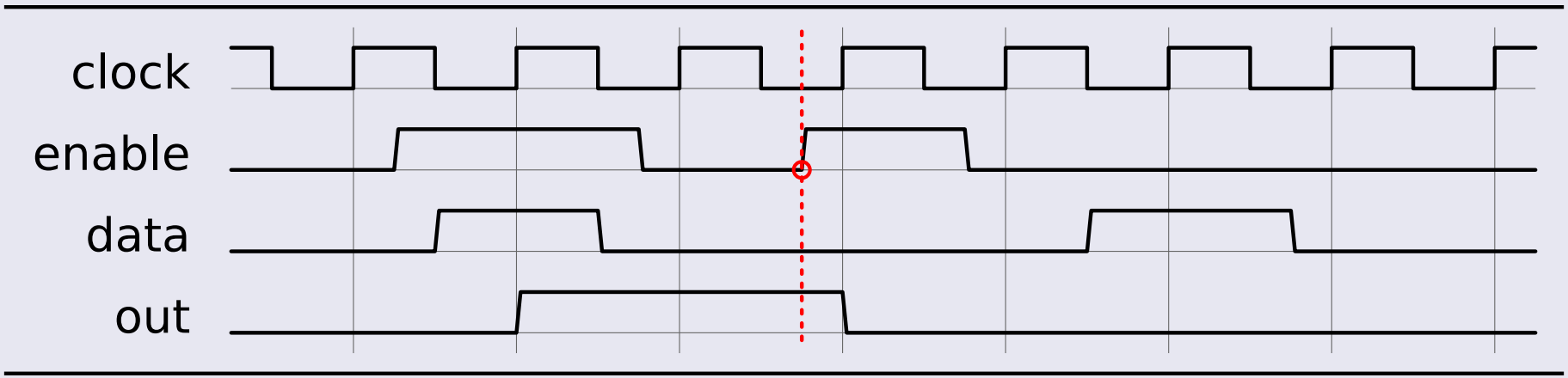
Flip-Flops: 1-bit register





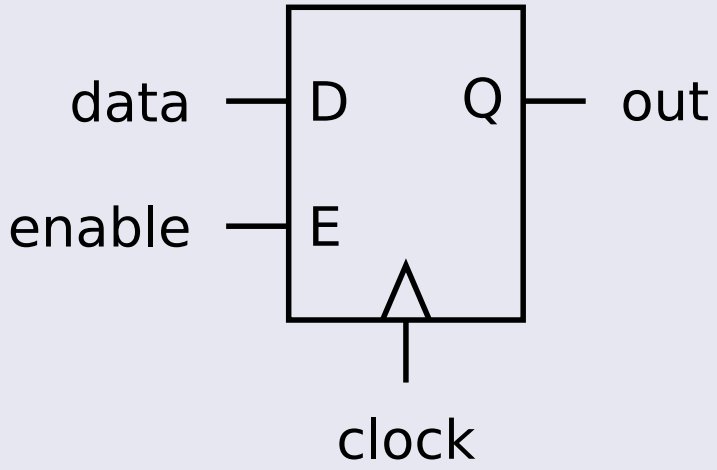
Flip-Flops: 1-bit register

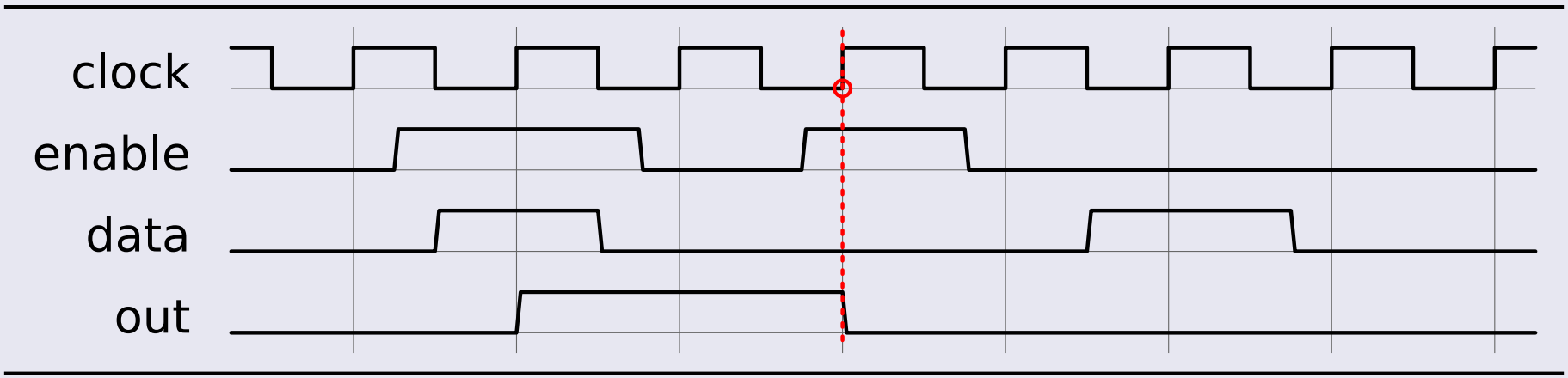




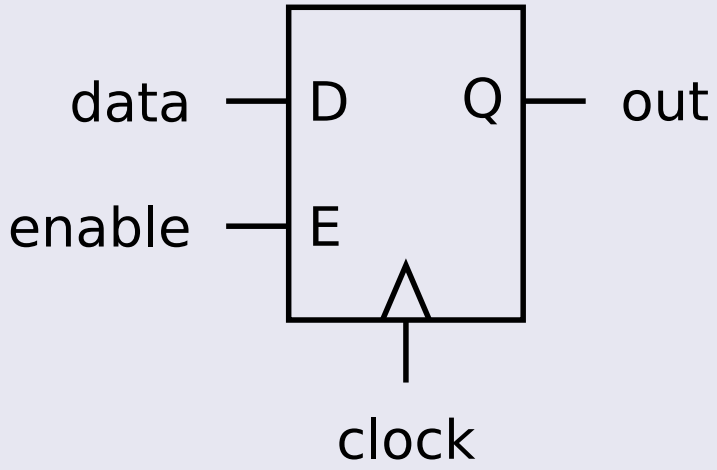
What are FPGAs?

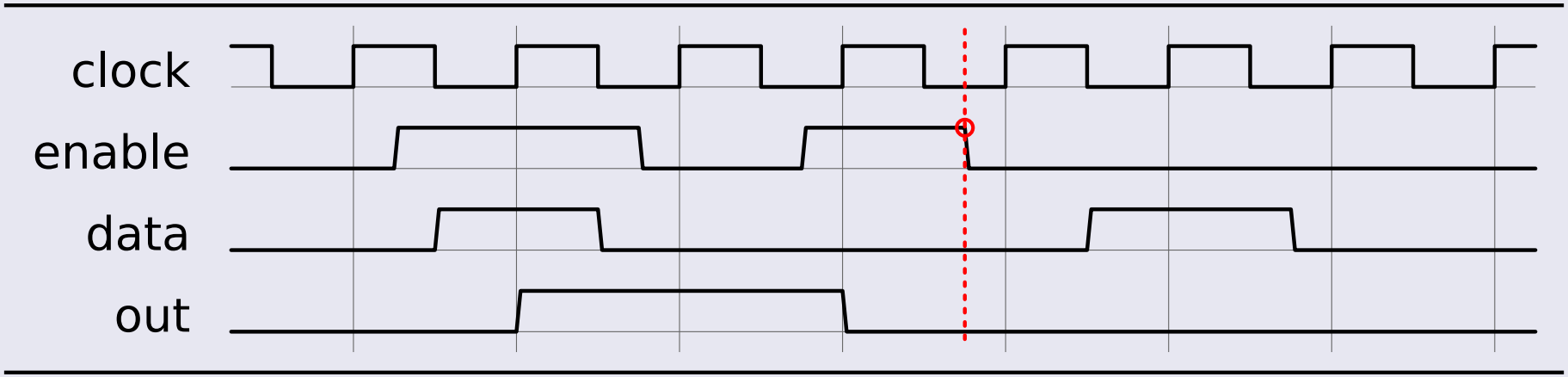
Flip-Flops: 1-bit register



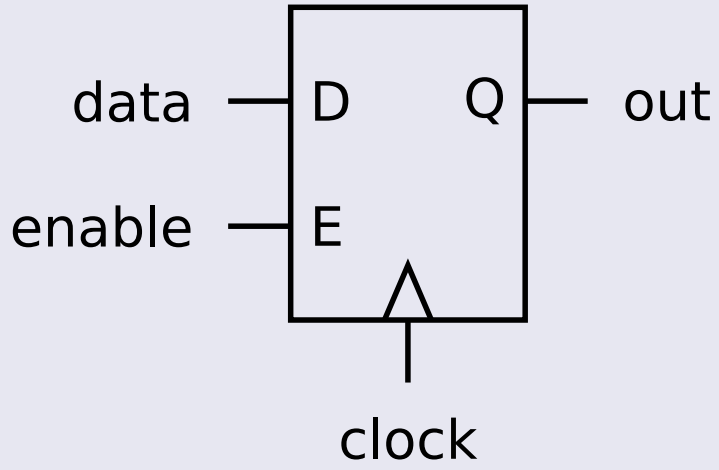


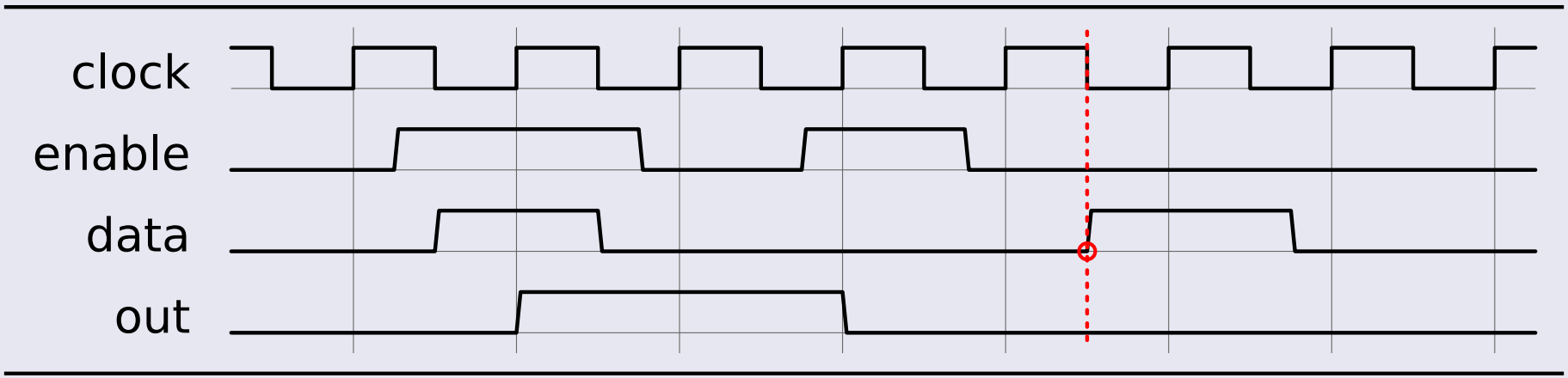
Flip-Flops: 1-bit register



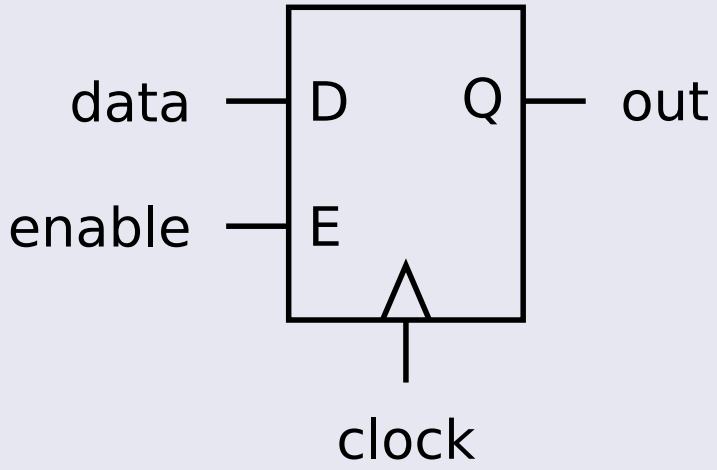


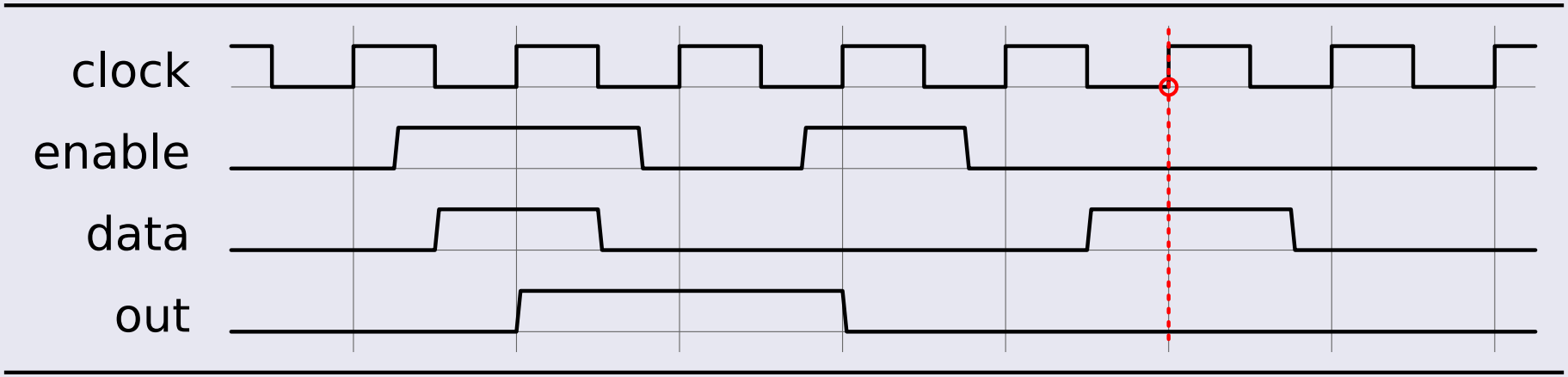
Flip-Flops: 1-bit register





Flip-Flops: 1-bit register





Lookup tables (LUTs) for implementing logic:

FPGAs use LUTs instead of logic gates, e.g.: $x + y = z$

x_1	x_0	y_1	y_0	z_2	z_1	z_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
...		...			...	

What are FPGAs?

Lookup tables (LUTs) for implementing logic:

FPGAs use LUTs instead of logic gates, e.g.: $x + y = z$

x_1	x_0	y_1	y_0	z_2	z_1	z_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
...		...			...	

LUT

What are FPGAs?

Lookup tables (LUTs) for implementing logic:

FPGAs use LUTs instead of logic gates, e.g.: $x + y = z$

address

	x_1	x_0	y_1	y_0	z_2	z_1	z_0
0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	1
2	0	0	1	0	0	1	0
3	0	0	1	1	0	1	1
4	0	1	0	0	0	0	1
5	0	1	0	1	0	1	0
6	0	1	1	0	0	1	1
7	0	1	1	1	1	0	0
	...		...			...	

LUT

Lookup tables (LUTs) for implementing logic:

FPGAs use LUTs instead of logic gates, e.g.: $x + y = z$

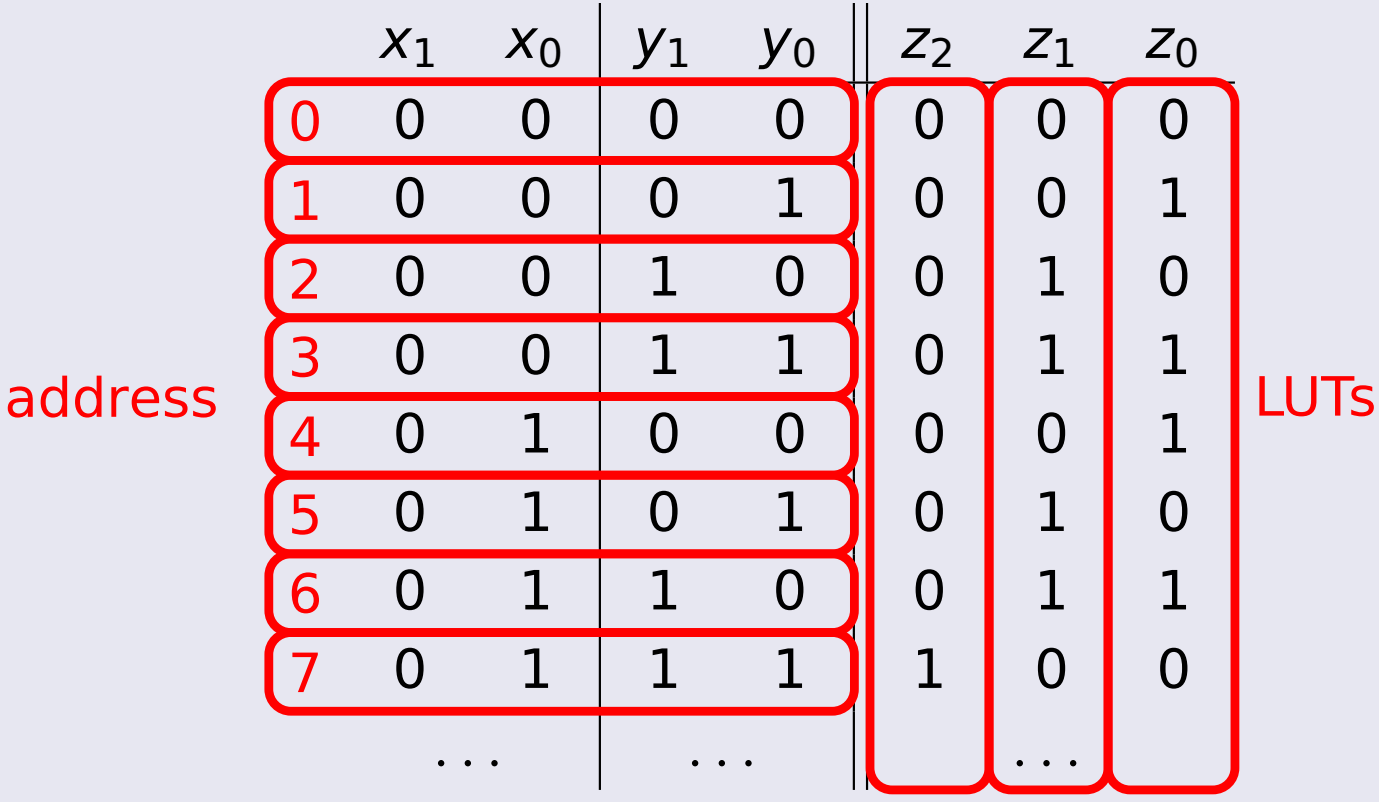
address

	x_1	x_0	y_1	y_0	z_2	z_1	z_0
0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	1
2	0	0	1	0	0	1	0
3	0	0	1	1	0	1	1
4	0	1	0	0	0	0	1
5	0	1	0	1	0	1	0
6	0	1	1	0	0	1	1
7	0	1	1	1	1	0	0
	...		...			...	

LUTs

Lookup tables (LUTs) for implementing logic:

FPGAs use LUTs instead of logic gates, e.g.: $x + y = z$



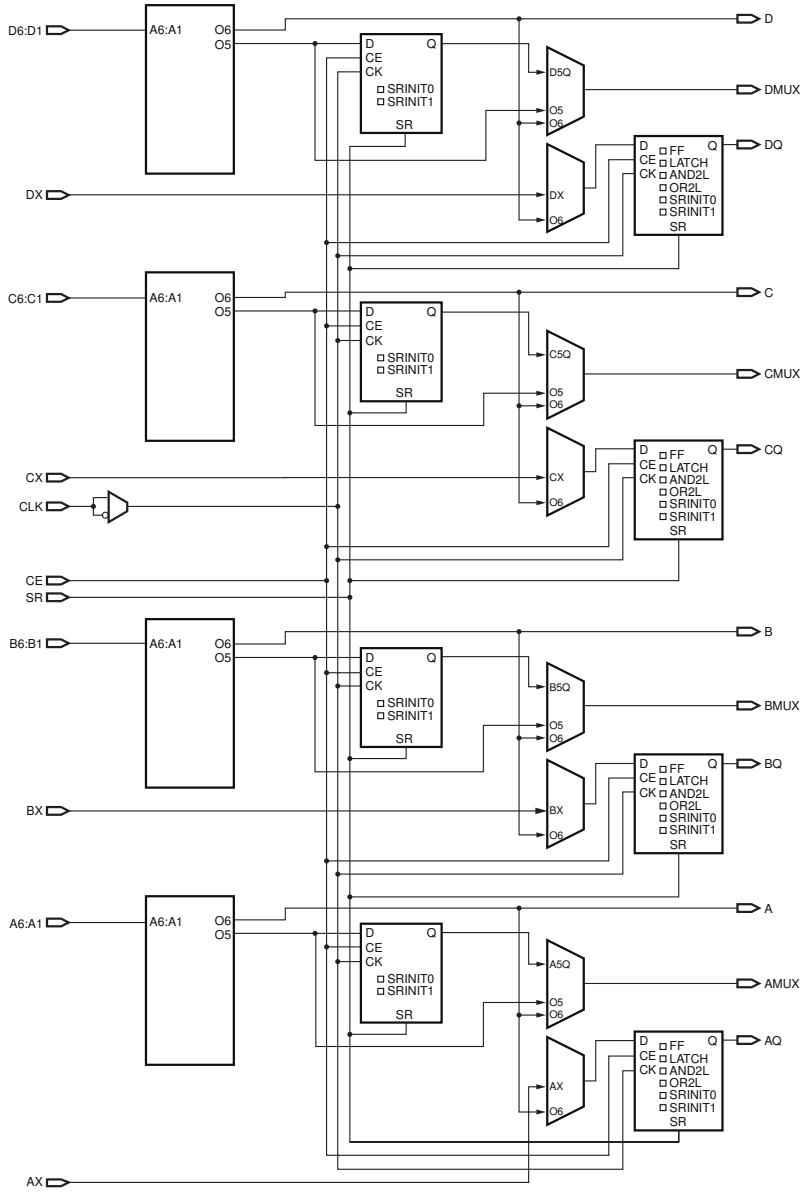
	x_1	x_0	y_1	y_0	z_2	z_1	z_0
0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	1
2	0	0	1	0	0	1	0
3	0	0	1	1	0	1	1
4	0	1	0	0	0	0	1
5	0	1	0	1	0	1	0
6	0	1	1	0	0	1	1
7	0	1	1	1	1	0	0
	...		...			...	

Each LUT typically has 3, 4, 5, or 6 input lines (8, 16, 32, or 64 bit).

What are FPGAs?

Hierarchical Structure:

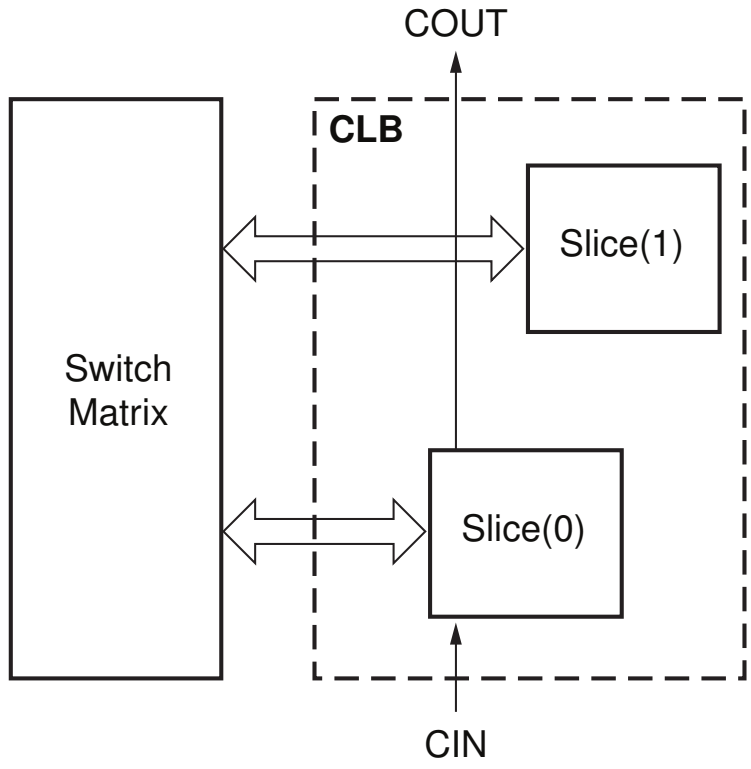
- Several LUTs and FFs form a *slice*.



What are FPGAs?

Hierarchical Structure:

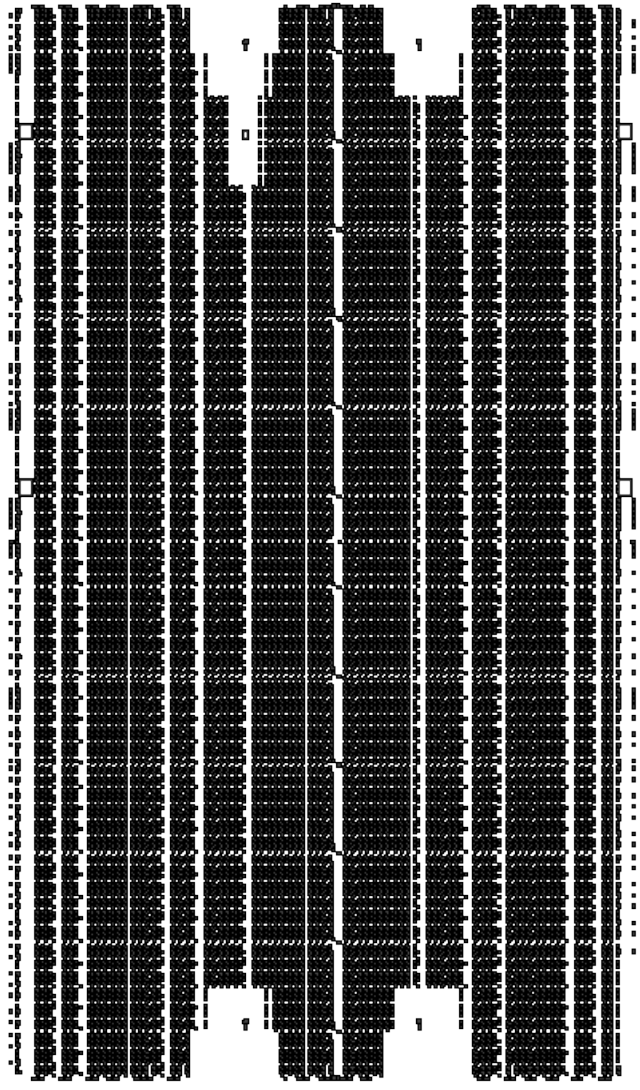
- Several LUTs and FFs form a *slice*.
- Several slices form a *configurable logic block* (CLB).



What are FPGAs?

Hierarchical Structure:

- Several LUTs and FFs form a *slice*.
- Several slices form a *configurable logic block* (CLB).
- CLBs, block RAM, DSP, etc. are connected by the routing fabric.



How to program FGPAs?

Hardware Description Languages (HDLs)

Most common: Verilog and VHDL

High-level description language:

- Modules as building blocks,
- wires (`wire`) to connect modules,
- logic can be assigned directly to wires (synthesized as LUT),
- registers (`reg`) to store states (synthesized as flip-flop),
- clauses to define behaviour over time,
- statements for generic code creation,
- dedicated modules for specific FPGA logic blocks.

How to program FGPAs?


```
module poly_mult (
  input [2:0] a,
  input [2:0] b,
  output [4:0] out
);

assign out[0] = (a[0] & b[0]);
assign out[1] = (a[1] & b[0]) ^ (a[0] & b[1]);
assign out[2] = (a[2] & b[0]) ^ (a[1] & b[1]) ^ (a[0] & b[2]);
assign out[3] = (a[2] & b[1]) ^ (a[1] & b[2]);
assign out[4] = (a[2] & b[2]);

endmodule // poly_mult
```

How to program FGPAs?


```
out[2] = (a[2] & b[0]) ^ (a[1] & b[1]) ^ (a[0] & b[2])
```

a[2]	a[1]	a[0]	b[2]	b[1]	b[0]	out[2]
0	0	0	0	0	0	0
0	0	0	0	0	1	0
0	0	0	0	1	0	0
0	0	0	0	1	1	0
			...			...
0	1	0	0	0	0	0
0	1	0	0	0	1	0
0	1	0	0	1	0	1
0	1	0	0	1	1	1
0	1	0	1	0	0	0
0	1	0	1	0	1	0

How to program FGPAs?


```
module poly_mult (input [2:0] a, input [2:0] b,
  output [4:0] out);
...
LUT6 #(.INIT(64'h96665aaa3cccf000)) LUT_mul_2
(
  .I0(b[0]),
  .I1(b[1]),
  .I2(b[2]),
  .I3(a[0]),
  .I4(a[1]),
  .I5(a[2]),
  .O(out[2])
);
...
endmodule // poly_mult
```

The Target — sect113r2

SECG curve sect113r2:

SECG curve sect113r2 is defined pseudo-randomly by a seed over $\mathbb{F}_{2^{113}} \cong \mathbb{F}_2[w]/(w^{113} + w^9 + 1)$ by an equation of the form $E : y^2 + xy = x^3 + ax^2 + b$ and base point $P = (x_P, y_P)$, where

$$a = \texttt{0x00006899 18DBEC7E 5A0DD6DF C0AA55C7},$$
$$b = \texttt{0x000095E9 A9EC9B29 7BD4BF36 E059184F},$$
$$x_P = \texttt{0x0001A57A 6A7B26CA 5EF52FCD B8164797}, \text{ and}$$
$$y_P = \texttt{0x0000B3AD C94ED1FE 674C06E6 95BABA1D}$$

The Target — sect113r2

SECG curve sect113r2:

SECG curve sect113r2 is defined pseudo-randomly by a seed over $\mathbb{F}_{2^{113}} \cong \mathbb{F}_2[w]/(w^{113} + w^9 + 1)$ by an equation of the form $E : y^2 + xy = x^3 + ax^2 + b$ and base point $P = (x_P, y_P)$, where

$$a = \texttt{0x00006899 18DBEC7E 5A0DD6DF C0AA55C7},$$
$$b = \texttt{0x000095E9 A9EC9B29 7BD4BF36 E059184F},$$
$$x_P = \texttt{0x0001A57A 6A7B26CA 5EF52FCD B8164797}, \text{ and}$$
$$y_P = \texttt{0x0000B3AD C94ED1FE 674C06E6 95BABA1D}$$

Security:

The SECG standard claims 56 bits of security and comparable strength to a 512 bit RSA key.

The Target — sect113r2

SECG curve sect113r2:

SECG curve sect113r2 is defined pseudo-randomly by a seed over $\mathbb{F}_{2^{113}} \cong \mathbb{F}_2[w]/(w^{113} + w^9 + 1)$ by an equation of the form $E : y^2 + xy = x^3 + ax^2 + b$ and base point $P = (x_P, y_P)$, where

$$a = \texttt{0x00006899 18DBEC7E 5A0DD6DF C0AA55C7},$$
$$b = \texttt{0x000095E9 A9EC9B29 7BD4BF36 E059184F},$$
$$x_P = \texttt{0x0001A57A 6A7B26CA 5EF52FCD B8164797}, \text{ and}$$
$$y_P = \texttt{0x0000B3AD C94ED1FE 674C06E6 95BABA1D}$$

Security:

The curve has been introduced to the standard in 2000 and was removed from it in 2005.

Addition, doubling:

Addition of two points $Q_1 = (x_1, y_1)$ and $Q_2 = (x_2, y_2)$ on this curve produces a result $Q_3 = (x_3, y_3)$ with

$$(x_3, y_3) = (\lambda^2 + \lambda + 1 + x_1 + x_2, \quad \lambda(x_1 + x_3) + y_1 + x_3), \text{ where}$$

$$\lambda = \begin{cases} (x_1^2 + y_1)/x_1 & \text{if } P_1 = P_2 \neq -P_2 \\ (y_1 + y_2)/(x_1 + x_2) & \text{if } P_1 \neq \pm P_2 \end{cases}.$$

Addition, doubling:

Addition of two points $Q_1 = (x_1, y_1)$ and $Q_2 = (x_2, y_2)$ on this curve produces a result $Q_3 = (x_3, y_3)$ with

$$(x_3, y_3) = (\lambda^2 + \lambda + 1 + x_1 + x_2, \quad \lambda(x_1 + x_3) + y_1 + x_3), \text{ where}$$

$$\lambda = \begin{cases} (x_1^2 + y_1)/x_1 & \text{if } P_1 = P_2 \neq -P_2 \\ (y_1 + y_2)/(x_1 + x_2) & \text{if } P_1 \neq \pm P_2 \end{cases}.$$

Negative:

The negative of a point is $-(x_1, y_1) = (x_1, y_1 + x_1)$ and $(x_1, y_1) + (x_1, y_1 + x_1) = \infty$.

Solving the ECDLP

ECDLP:

Given a base point P and a point Q, find k such that $Q = kP$.

Solving the ECDLP

ECDLP:

Given a base point P and a point Q, find k such that $Q = kP$.

Naive solution:

Make an exhaustive search by checking all possible values for k.

The order of the base point P given with `sect113r2` is the prime $\ell = 5\,192\,296\,858\,534\,827\,702\,972\,497\,909\,952\,403 < 2^{112}$.

Thus, this approach takes up to 2^{112} attempts.

Solving the ECDLP

ECDLP:

Given a base point P and a point Q, find k such that $Q = kP$.

Better solution:

In a list of about $\sqrt{n}$ elements from a set of n elements, there is a collision with high probability (birthday paradox).

Idea: Compute values $R_i = a_iP + b_iQ$ with random a_i and b_i.

Solving the ECDLP

ECDLP:

Given a base point P and a point Q, find k such that $Q = kP$.

Better solution:

In a list of about $\sqrt{n}$ elements from a set of n elements, there is a collision with high probability (birthday paradox).

Idea: Compute values $R_i = a_iP + b_iQ$ with random a_i and b_i. Once we find some $R_i = R_j$, i.e.,

$$a_iP + b_iQ = a_jP + b_jQ,$$

ECDLP:

Given a base point P and a point Q, find k such that $Q = kP$.

Better solution:

In a list of about $\sqrt{n}$ elements from a set of n elements, there is a collision with high probability (birthday paradox).

Idea: Compute values $R_i = a_iP + b_iQ$ with random a_i and b_i. Once we find some $R_i = R_j$, i.e.,

$$a_iP + b_iQ = a_jP + b_jQ,$$

compute

$$Q = (a_j - a_i)/(b_i - b_j)P.$$

Solving the ECDLP

ECDLP:

Given a base point P and a point Q, find k such that $Q = kP$.

Better solution:

In a list of about $\sqrt{n}$ elements from a set of n elements, there is a collision with high probability (birthday paradox).

Idea: Compute values $R_i = a_iP + b_iQ$ with random a_i and b_i. Once we find some $R_i = R_j$, i.e.,

$$a_iP + b_iQ = a_jP + b_jQ,$$

compute

$$Q = (a_j - a_i)/(b_i - b_j)P.$$

This requires about $\sqrt{\ell} \approx \sqrt{2^{112}} \approx 2^{56}$ points.

Solving the ECDLP

ECDLP:

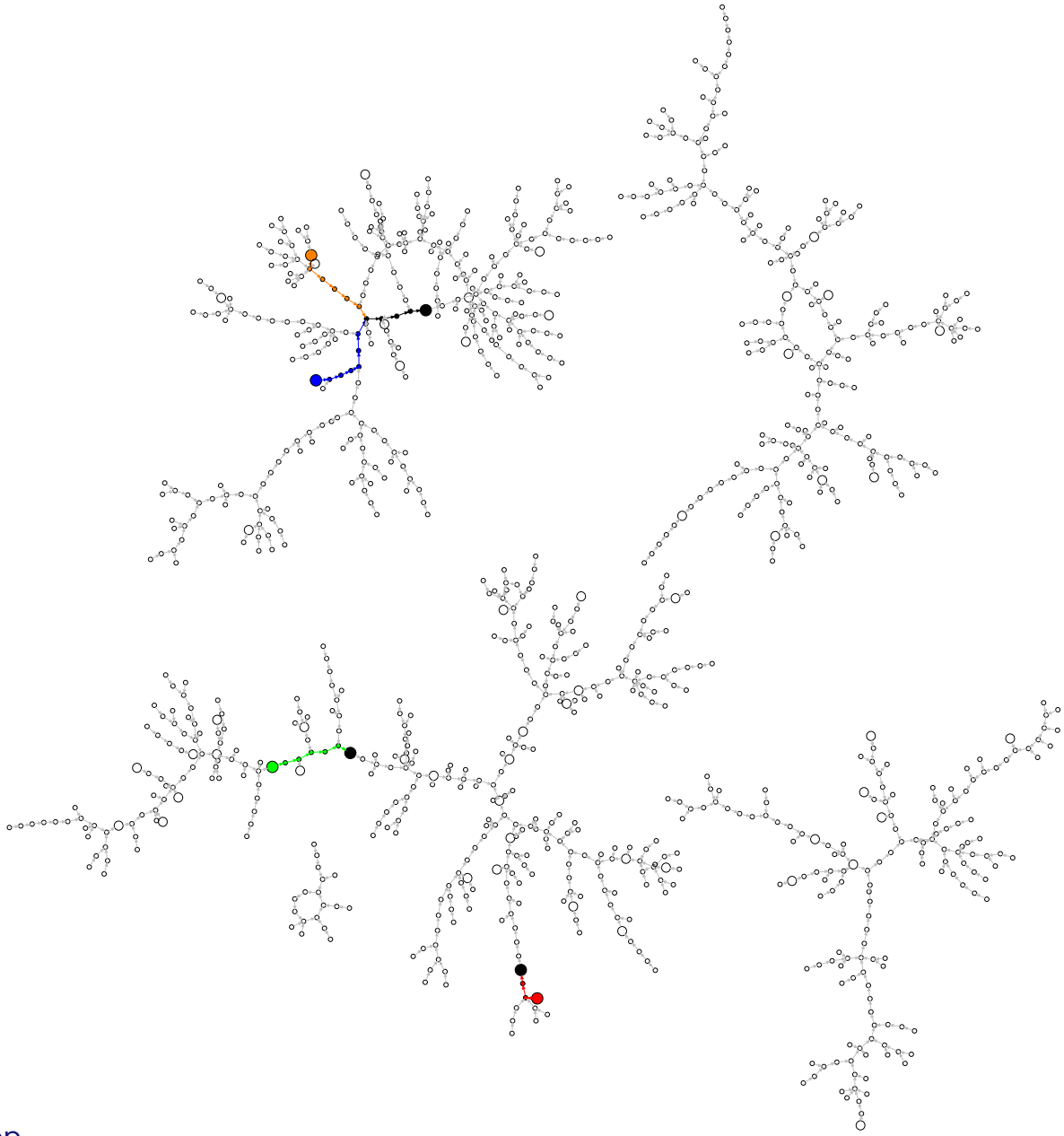
Given a base point P and a point Q, find k such that $Q = kP$.

“Best” solution:

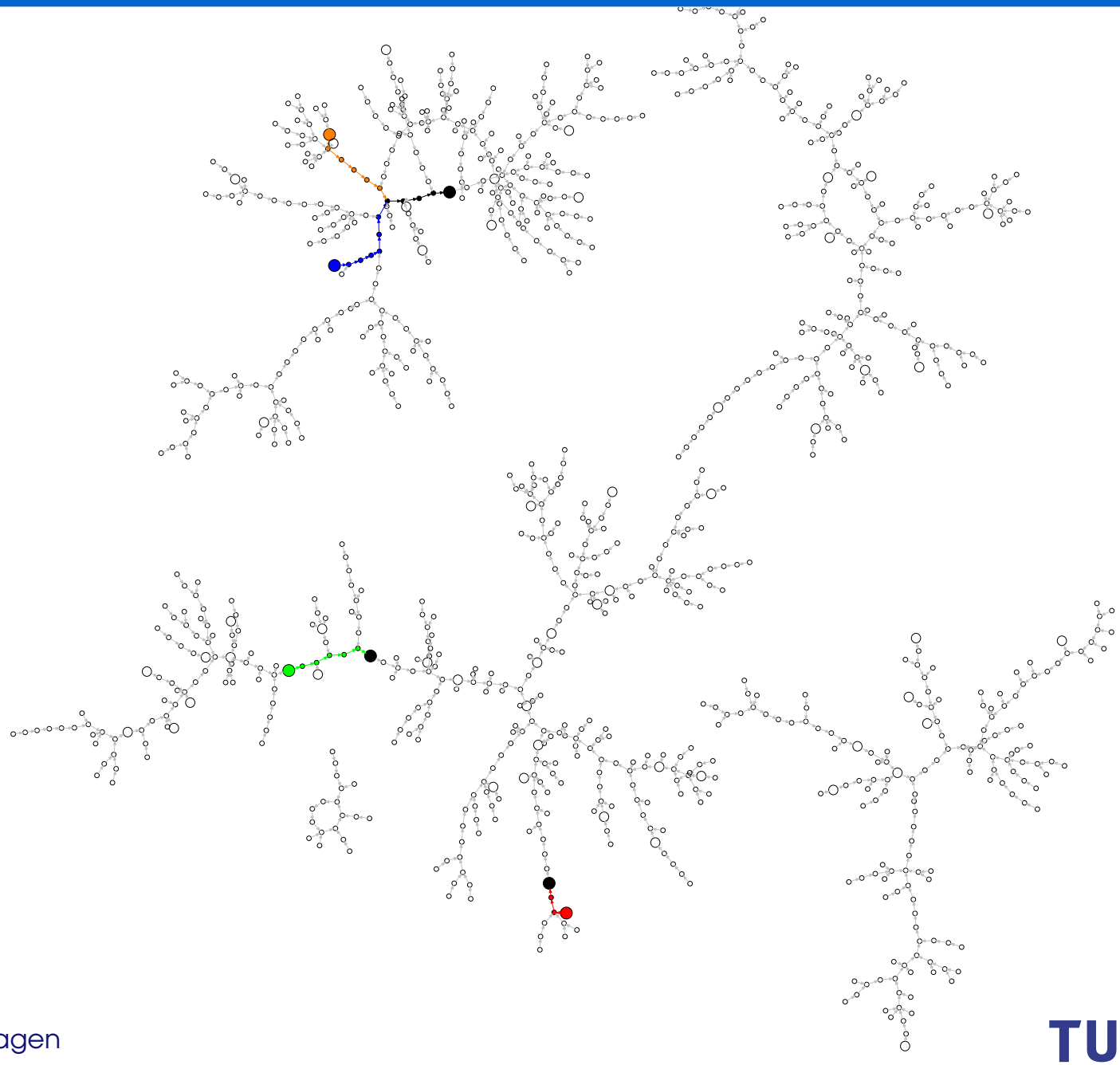
Perform *random walks* and store only *distinguished points*.

Still requires to compute about 2^{56} points
but required storage can be reduced.

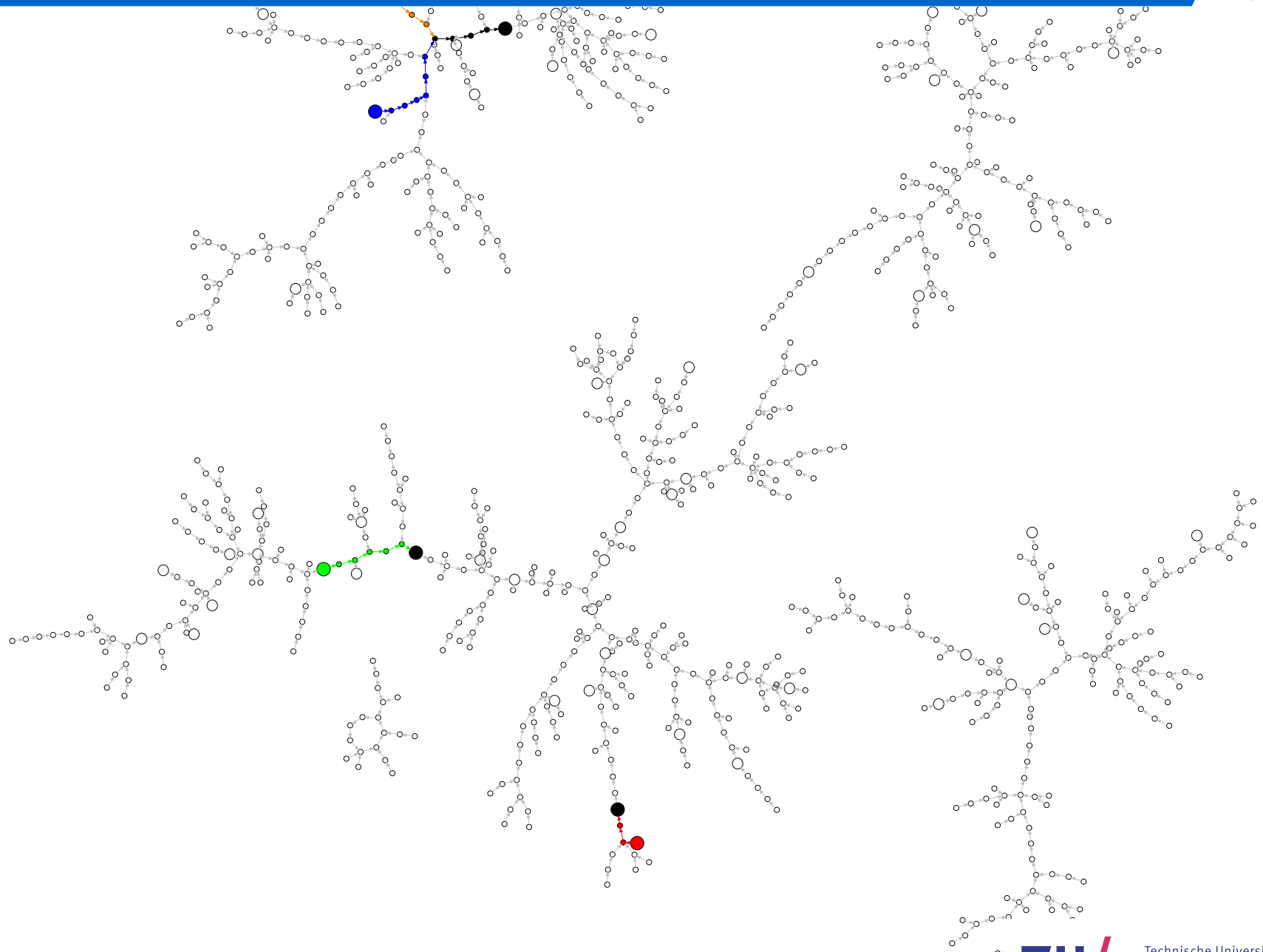
Example: Random Walk



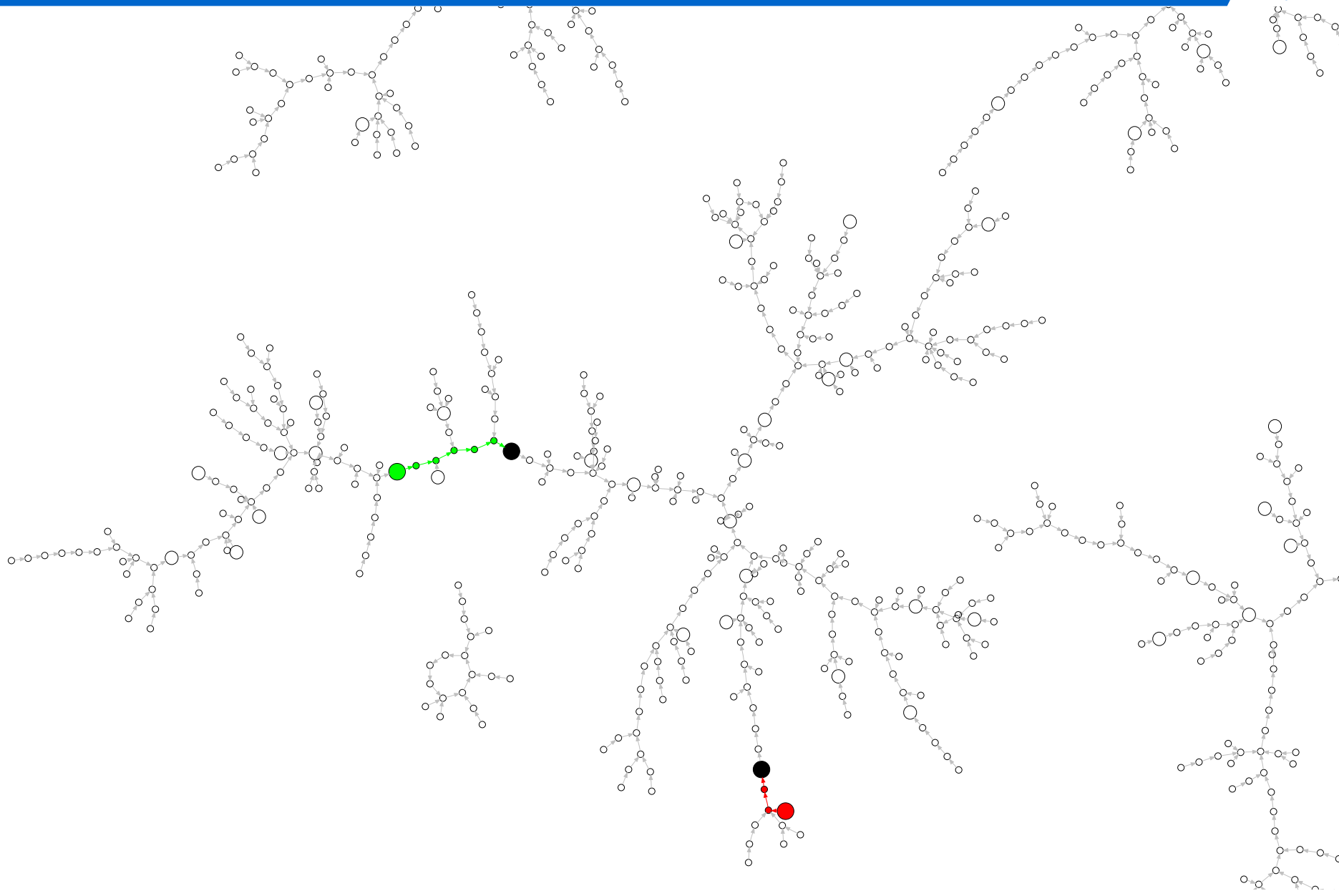
Example: Random Walk



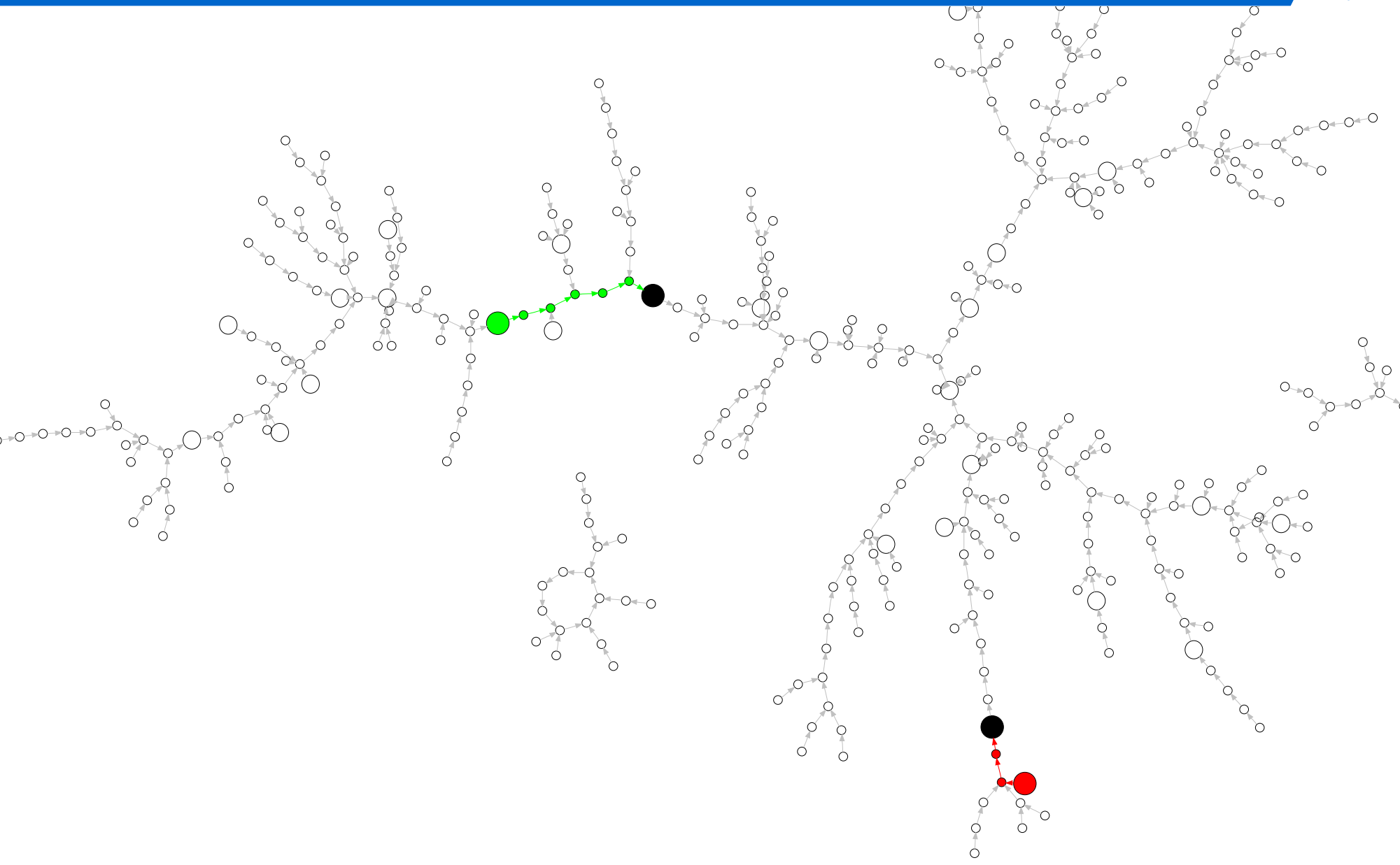
Example: Random Walk



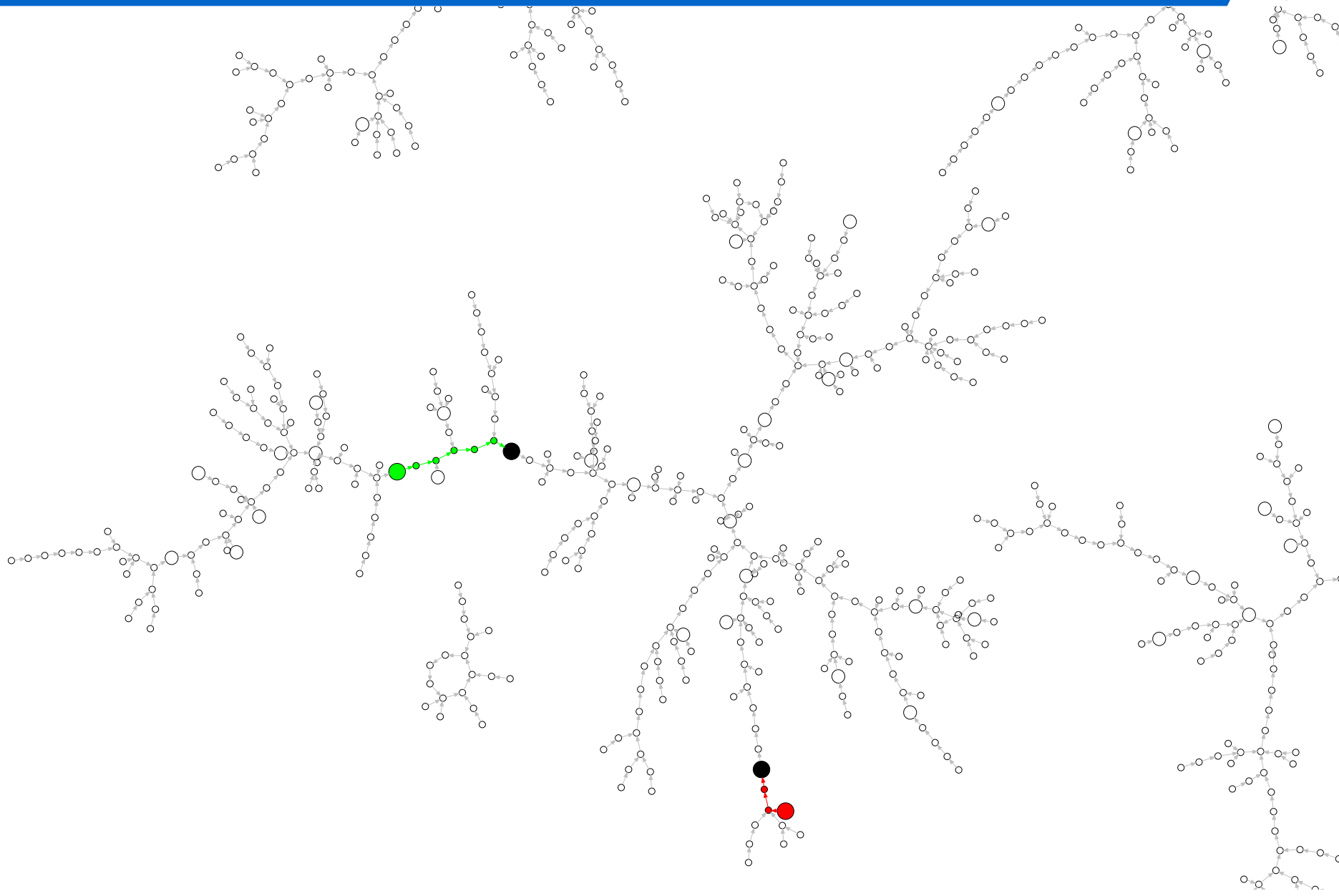
Example: Random Walk



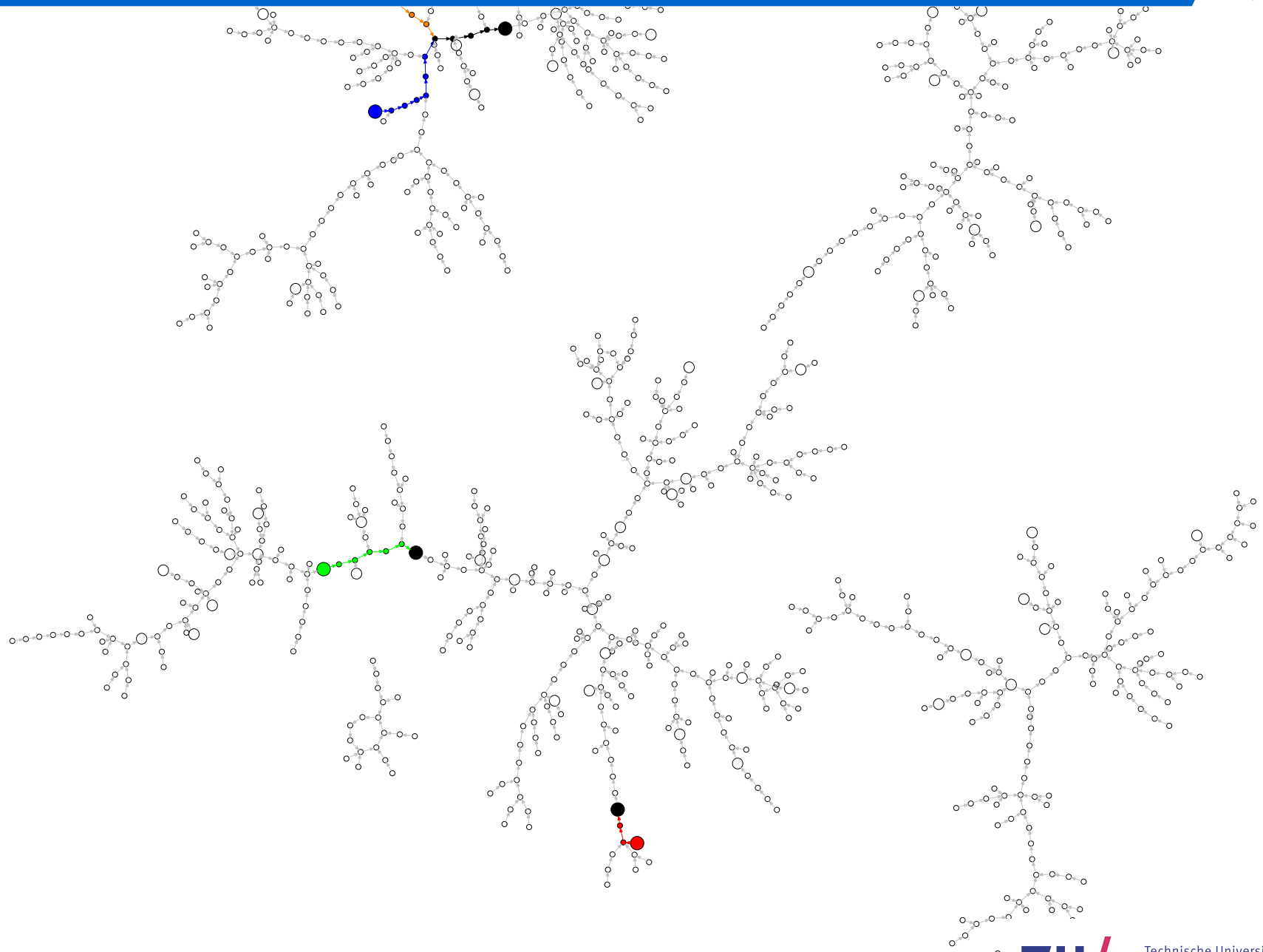
Example: Random Walk



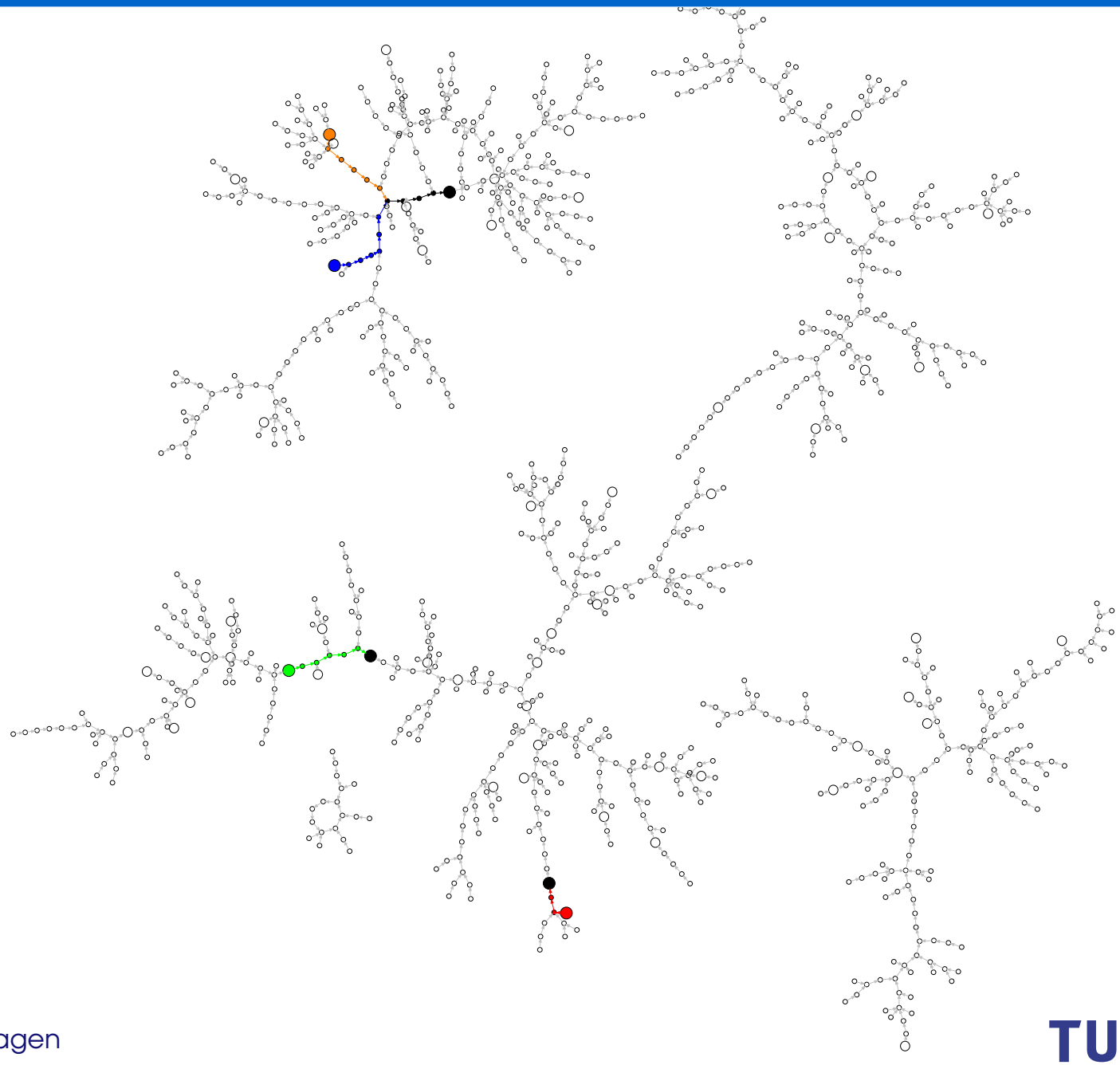
Example: Random Walk



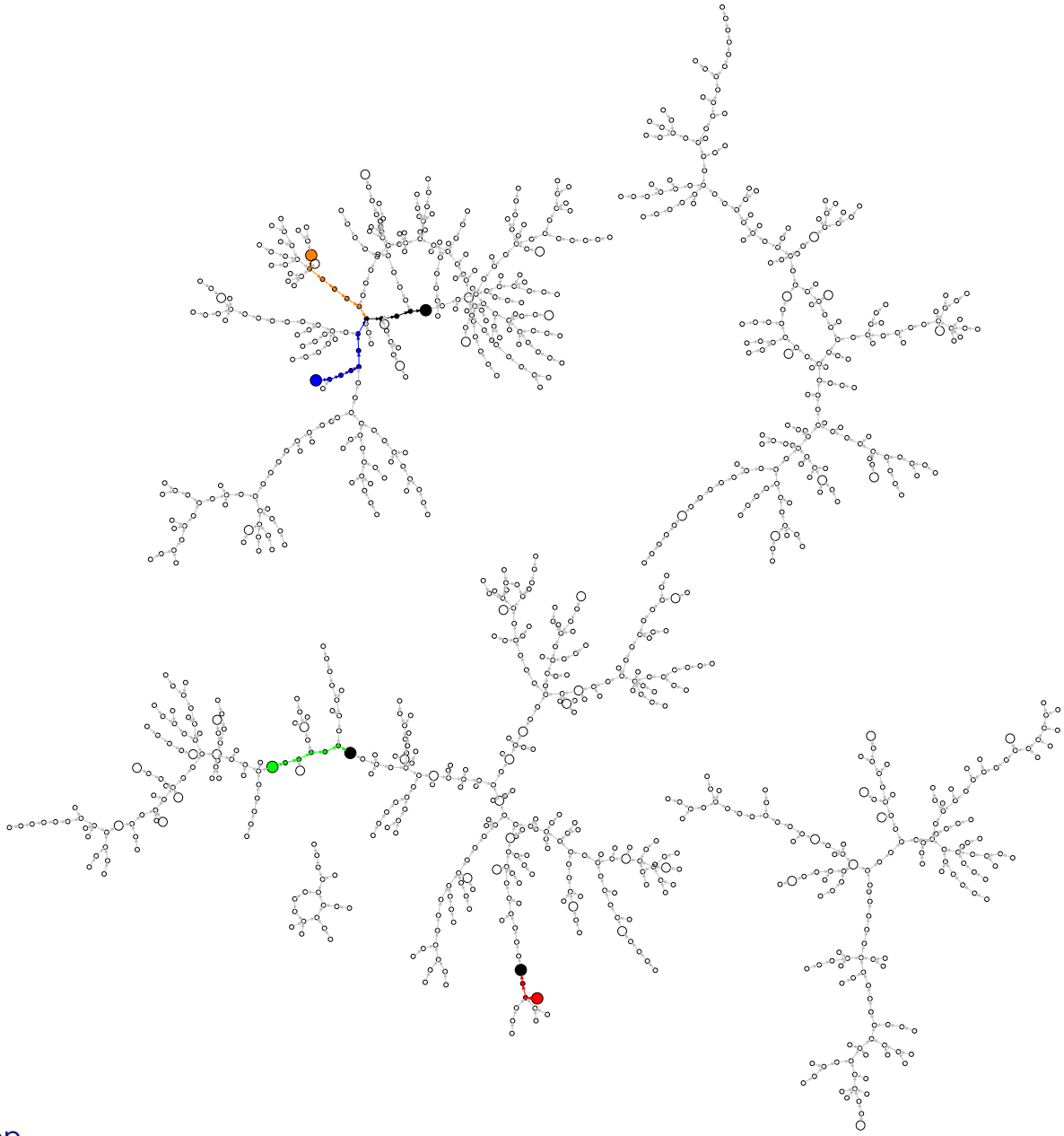
Example: Random Walk



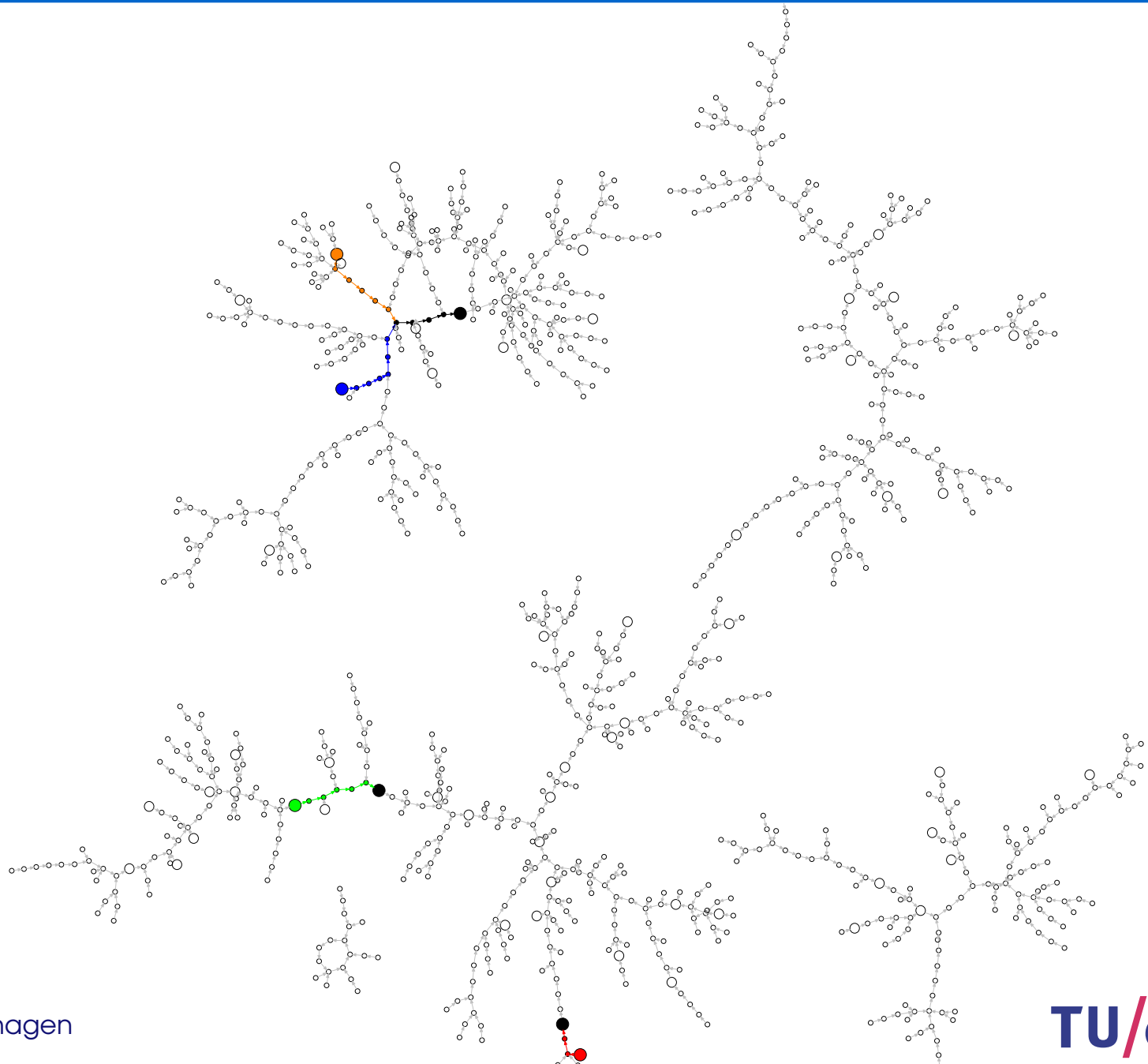
Example: Random Walk



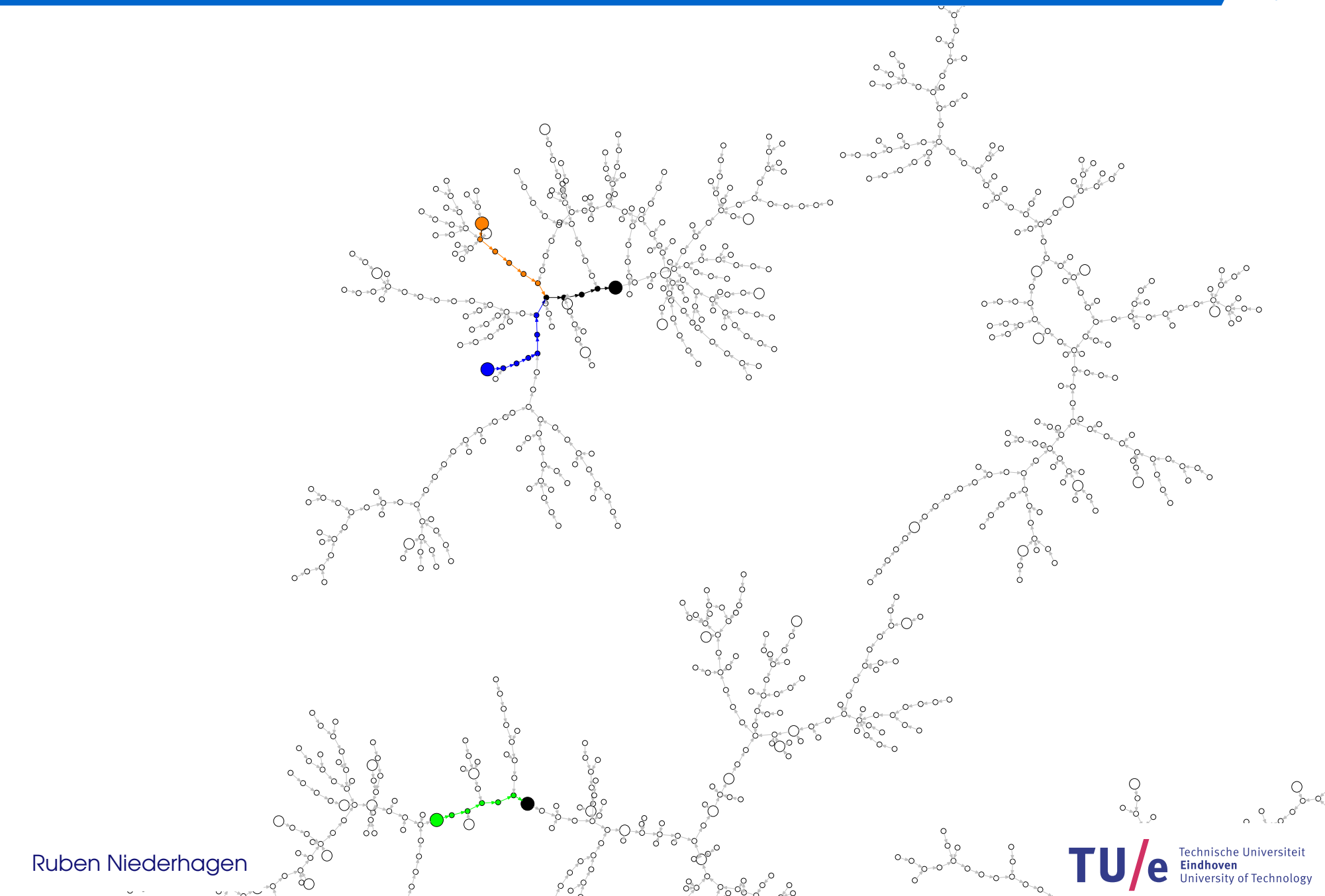
Example: Random Walk



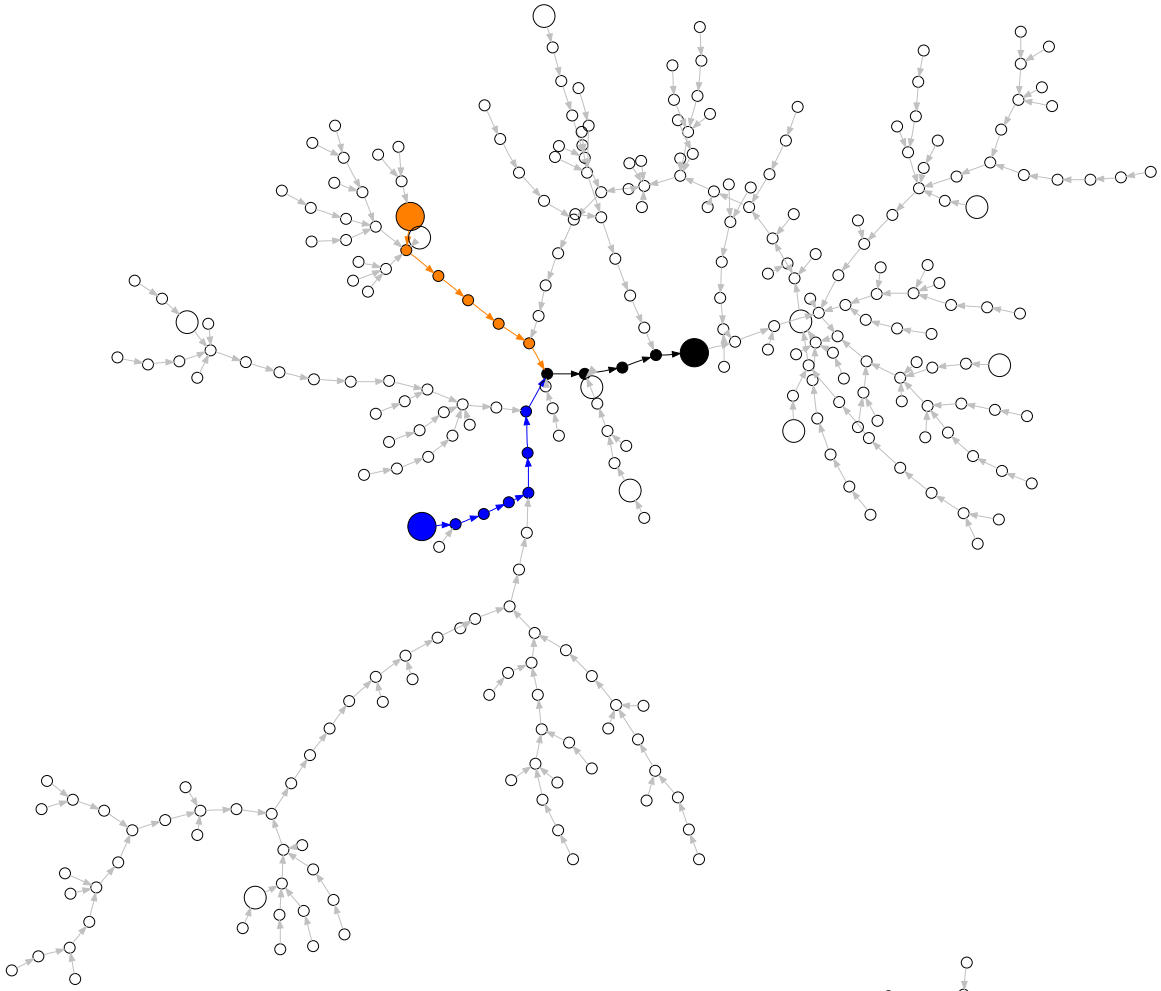
Example: Random Walk



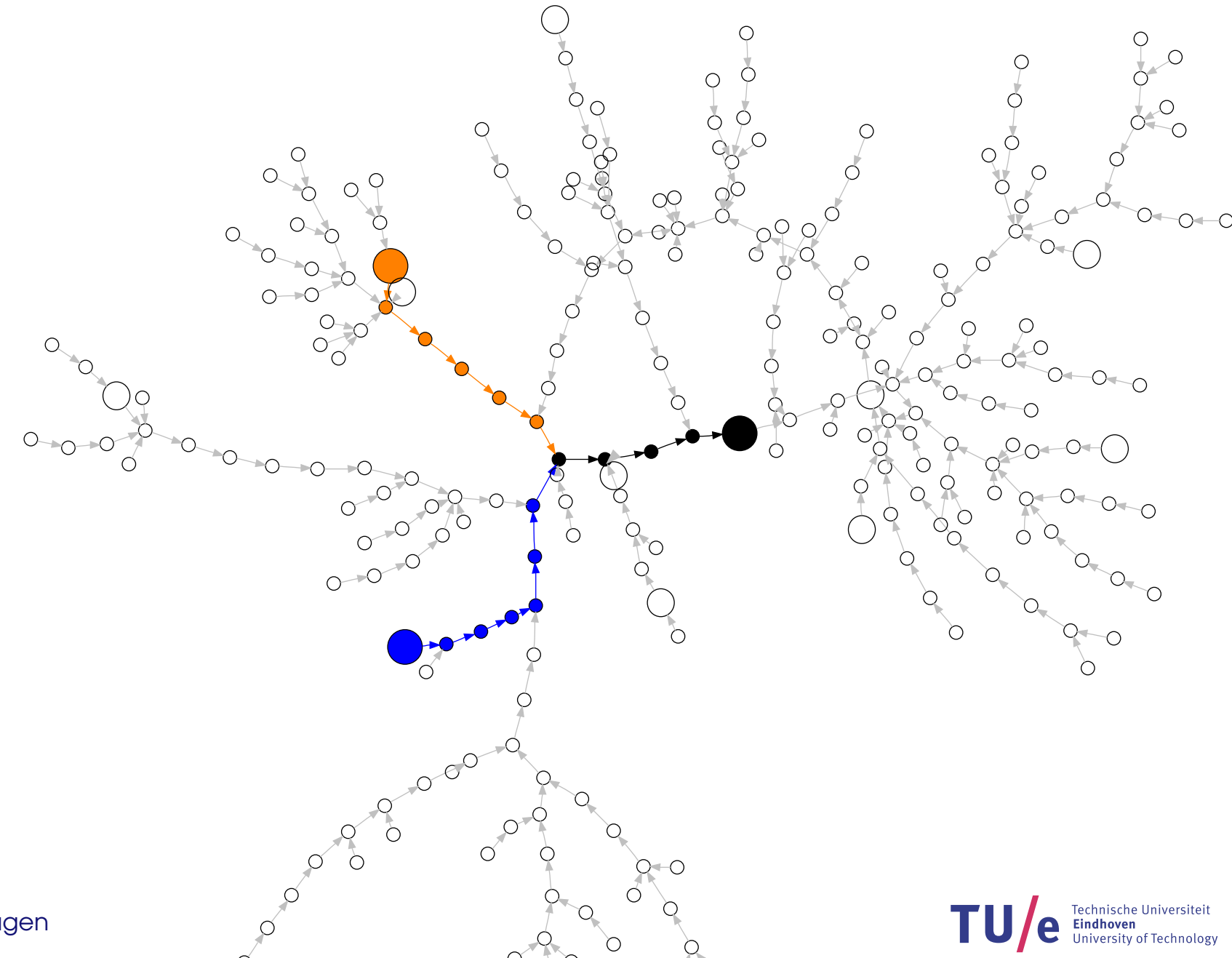
Example: Random Walk



Example: Random Walk



Example: Random Walk



Parallel version of Pollard's Rho algorithm

Very suitable for parallel computation:

- Compute many independent walks in parallel.
- Distinguished points are collected by a central server.
- Once the server receives the same point a second time, derive the discrete log.

How to perform a random walk?

Each step must depend only on the current point R_i.

Prepare a lookup table $T = \{T_0, \ldots, T_{2^t}\}$ with 2^t pseudo-random points such that for each point $T_i = a_i P$ the value of a_i is known.

Compute each starting point from a seed as $R_0 = a_0 P + b_i Q$ by deriving a and b from the seed.

For each random step, use a function

$$I : E \mapsto \{0, \ldots, 2^t\}$$

(e.g., take the top t bits of the x-coordinate of R_i) to compute

$$R_{i+1} = R_i + T_{I(R_i)}.$$

How to perform a random walk?

$$
\begin{aligned}
T_0 &= t_0 P \\
T_1 &= t_1 P \\
&\dots \\
T_{n-1} &= t_{n-1} P
\end{aligned}
$$

How to perform a random walk?

$$
\begin{aligned}
T_0 &= t_0 P \\
T_1 &= t_1 P \\
&\dots \\
T_{n-1} &= t_{n-1} P
\end{aligned}
$$

- $P_i = a_i P + b_i Q$

  $x(P_i) = 01001 \dots 00001$

How to perform a random walk?

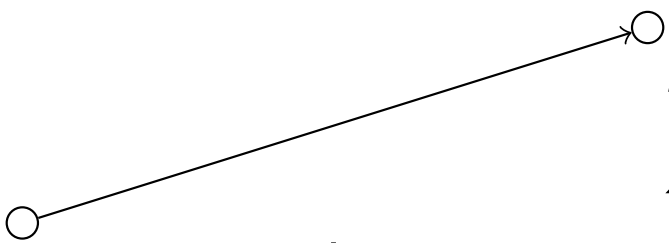
$$
\begin{aligned}
&T_0 = t_0 P \\
&T_1 = t_1 P \\
&\dots \\
&T_{n-1} = t_{n-1} P
\end{aligned}
$$

- $P_i = a_i P + b_i Q$

  $x(P_i) = 01001 \dots 00001$

How to perform a random walk?

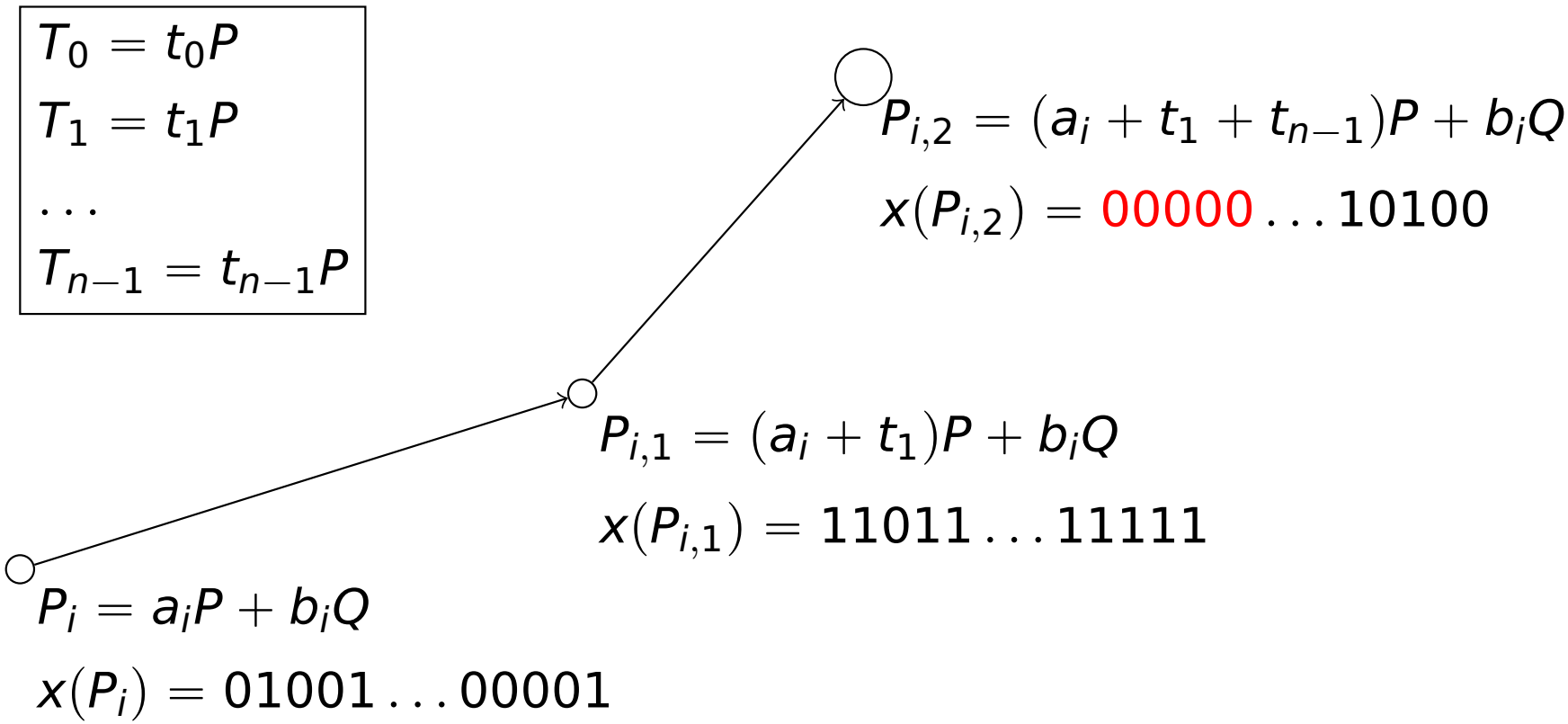
$$T_0 = t_0 P$$
$$T_1 = t_1 P$$
$$\dots$$
$$T_{n-1} = t_{n-1} P$$



$$P_{i,1} = (a_i + t_1)P + b_i Q$$
$$x(P_{i,1}) = 11011 \dots 11111$$

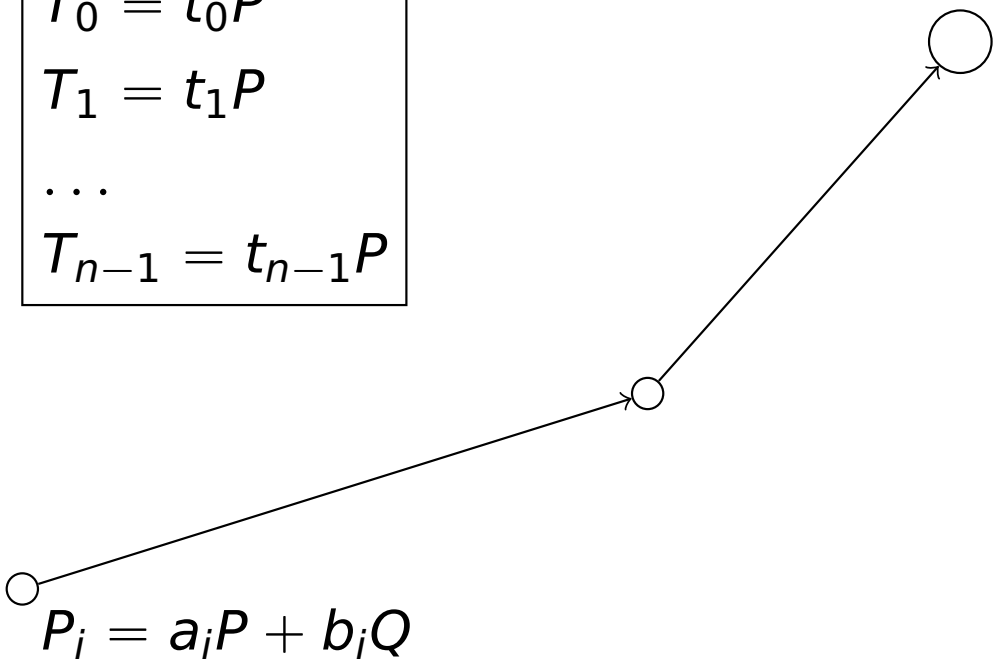
$$P_i = a_i P + b_i Q$$
$$x(P_i) = 01001 \dots 00001$$

How to perform a random walk?



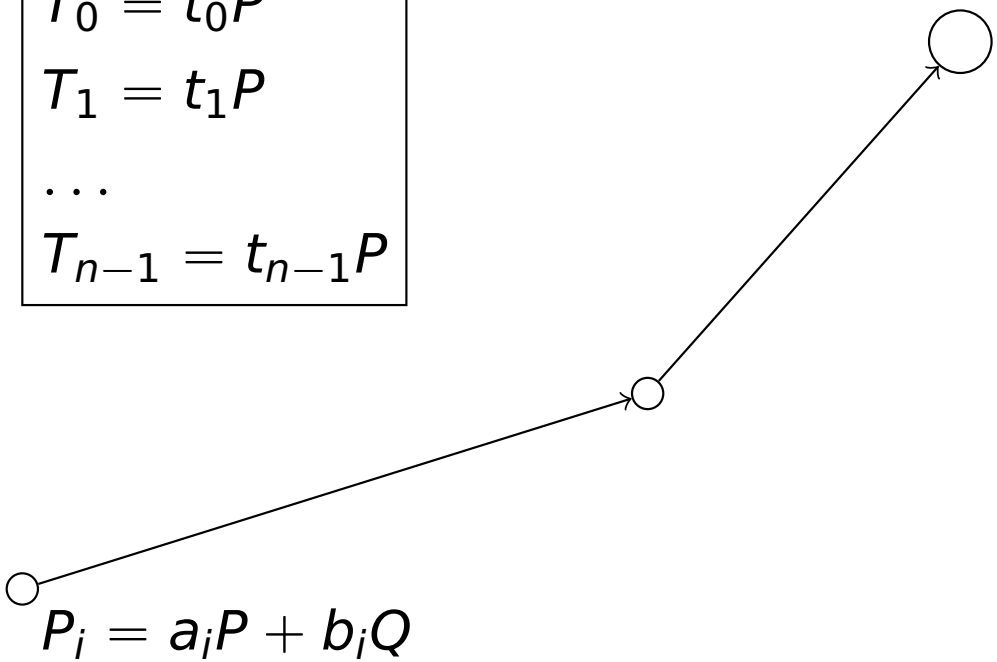
How to perform a random walk?

$$T_0 = t_0 P$$
$$T_1 = t_1 P$$
$$\dots$$
$$T_{n-1} = t_{n-1} P$$

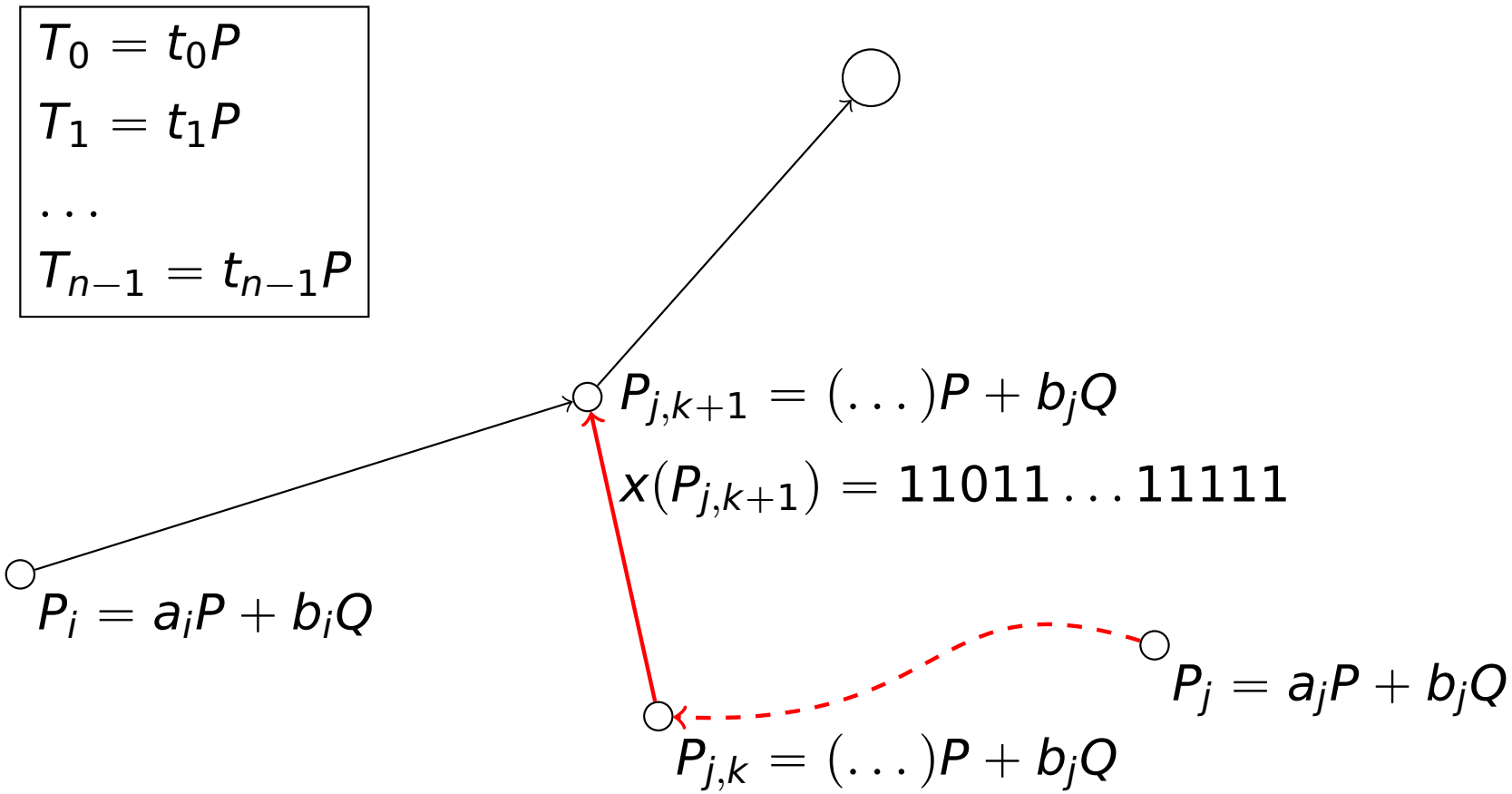


How to perform a random walk?

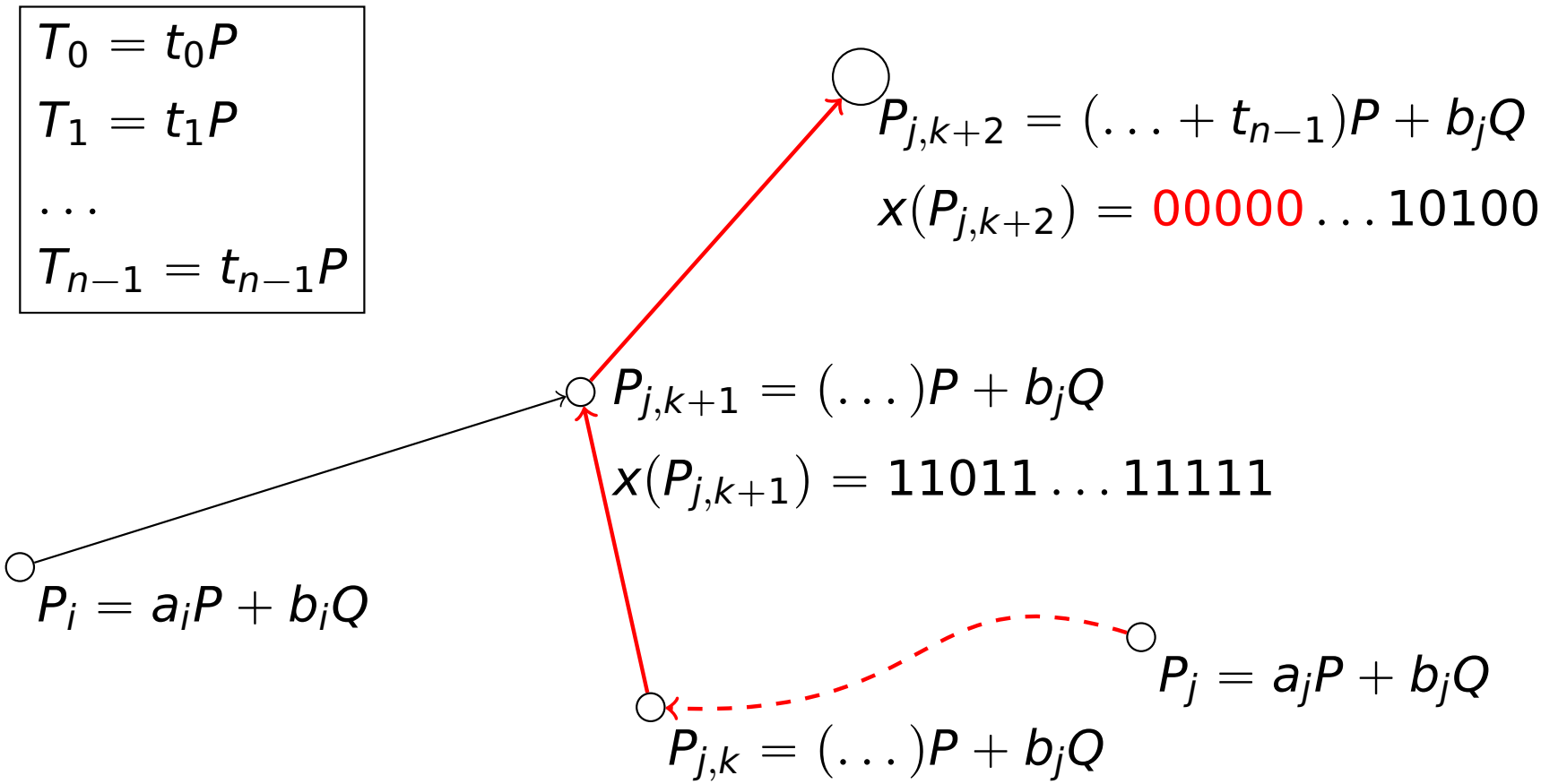
$$T_0 = t_0 P$$
$$T_1 = t_1 P$$
$$\dots$$
$$T_{n-1} = t_{n-1} P$$



How to perform a random walk?



How to perform a random walk?



Negation map

Optimization:

For each x-coordinate, there are two y-coordinates y_0 and y_1 giving a point on the curve: $(x, y_0) = -(x, y_1) = -(x, x + y_0)$.

Thus, we can halve the search space by continuing with the point $|R_i| = (x, \min(y, x + y))$ after computing $R_i = (x, y)$.

Negation map

Optimization:

For each x-coordinate, there are two y-coordinates y_0 and y_1 giving a point on the curve: $(x, y_0) = -(x, y_1) = -(x, x + y_0)$.

Thus, we can halve the search space by continuing with the point $|R_i| = (x, \min(y, x + y))$ after computing $R_i = (x, y)$.

Problem:

Due to the definition of the iteration function, we might encounter loops, e.g., in case $I(R_i) = I(R_{i+1})$:

$$R_{i+1} = |(R_i + T_{I(R_i)}i)| = -(R_i + T_{I(R_i)})$$
$$R_{i+2} = |-(R_i + T_{I(R_i)}) + T_{I(R_{i+1})}| = |-R_i| = R_i$$

Negation map

Optimization:

For each x-coordinate, there are two y-coordinates y_0 and y_1 giving a point on the curve: $(x, y_0) = -(x, y_1) = -(x, x + y_0)$.

Thus, we can halve the search space by continuing with the point $|R_i| = (x, \min(y, x + y))$ after computing $R_i = (x, y)$.

Problem:

Due to the definition of the iteration function, we might encounter loops, e.g., in case $I(R_i) = I(R_{i+1})$:

$$R_{i+1} = |(R_i + T_{I(R_i)}i)| = -(R_i + T_{I(R_i)})$$
$$R_{i+2} = |-(R_i + T_{I(R_i)}) + T_{I(R_{i+1})}| = |-R_i| = R_i$$

Solution:

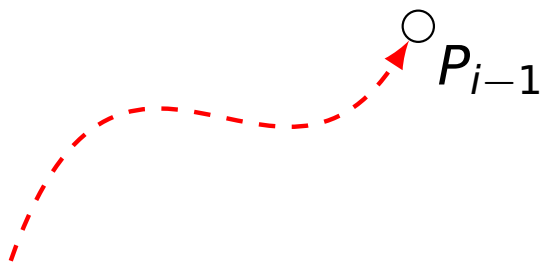
Detect c-cycles and leave them by computing

$$R_{i+c} = 2 \cdot \min(R_i, \ldots, R_{i+c-1}).$$

Negation map

$x_{\min} = x(P_{i-4})$

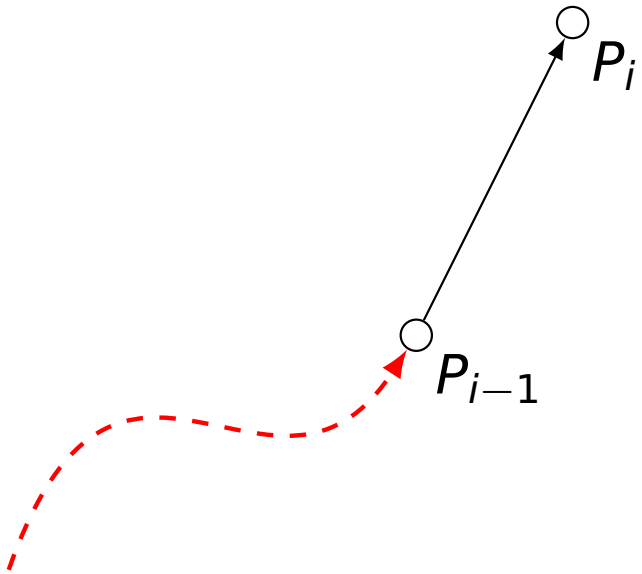
$\mathrm{ctr} = 15$



Negation map

$x_{\text{min}} = x(P_i)$

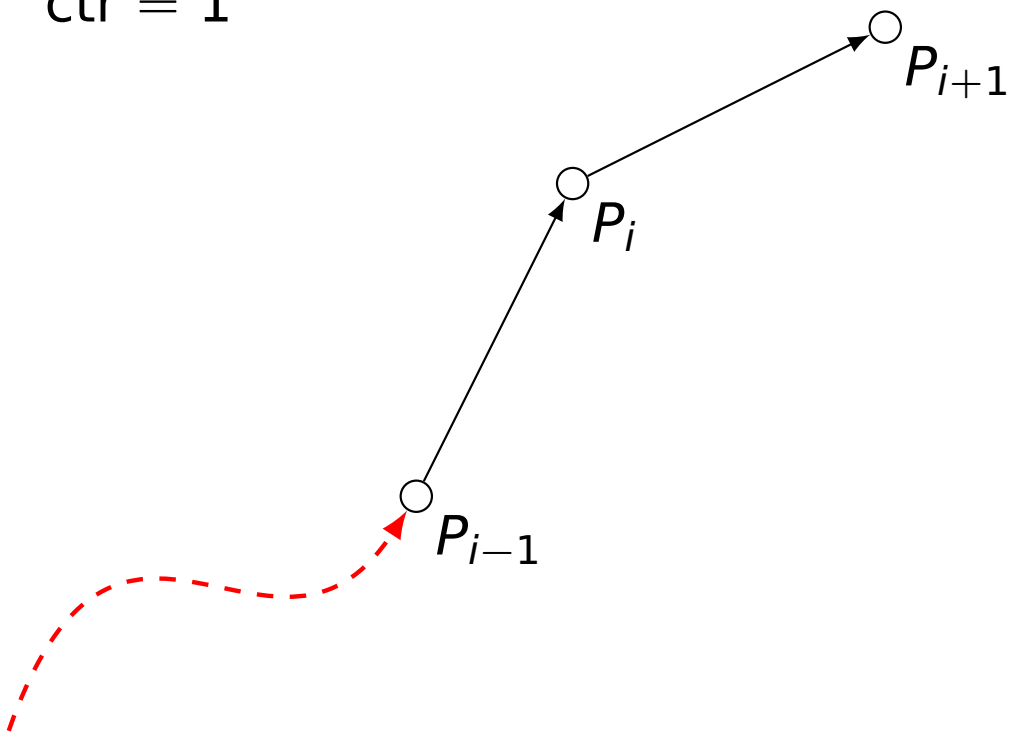
$\text{ctr} = 0$



Negation map

$x_{\min} = x(P_i)$

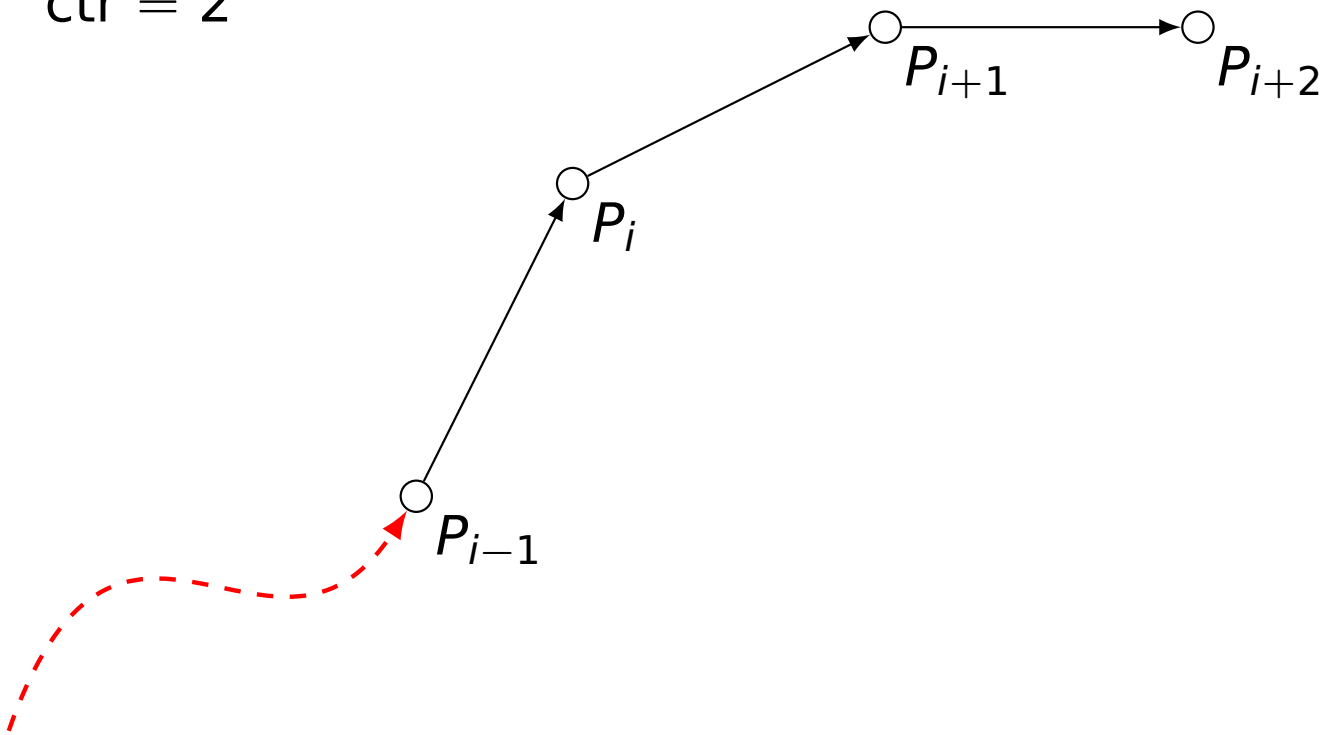
$\mathrm{ctr} = 1$



Negation map

$x_{\min} = x(P_{i+2})$

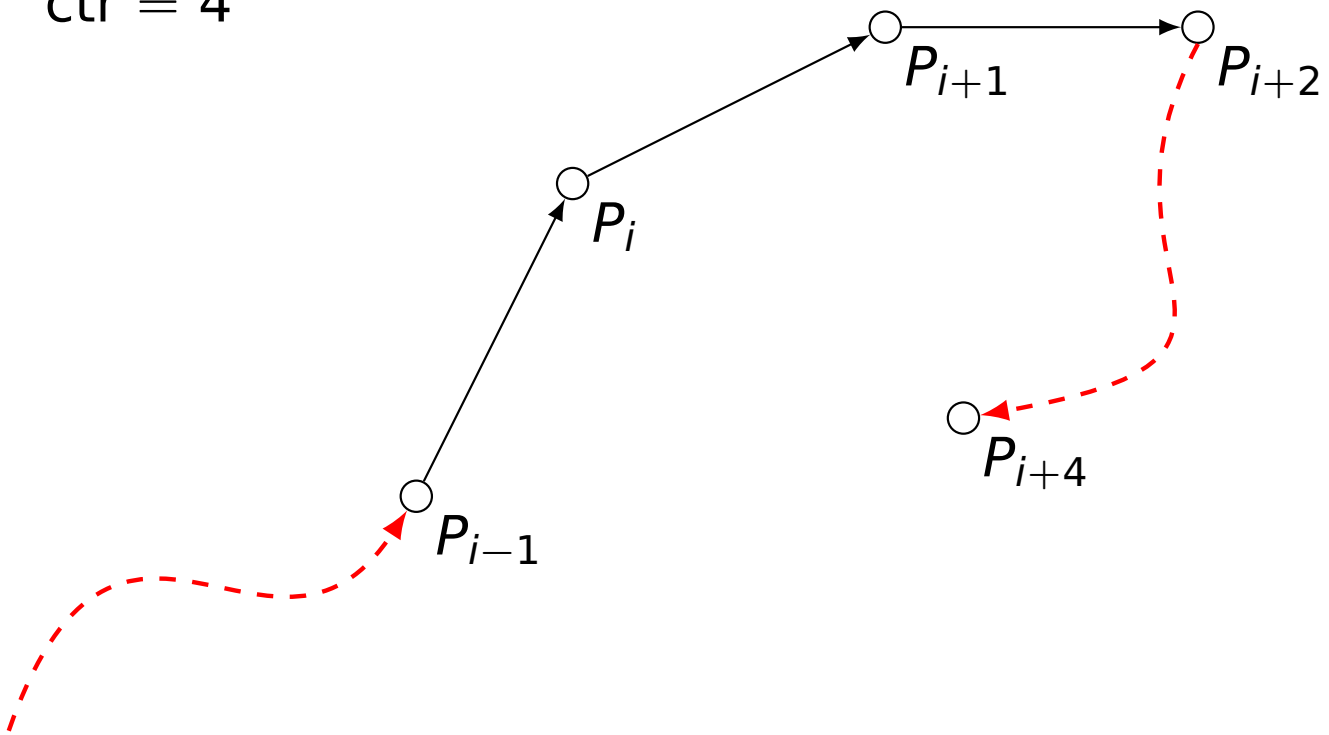
$\mathrm{ctr} = 2$



Negation map

$x_{\min} = x(P_{i+2})$

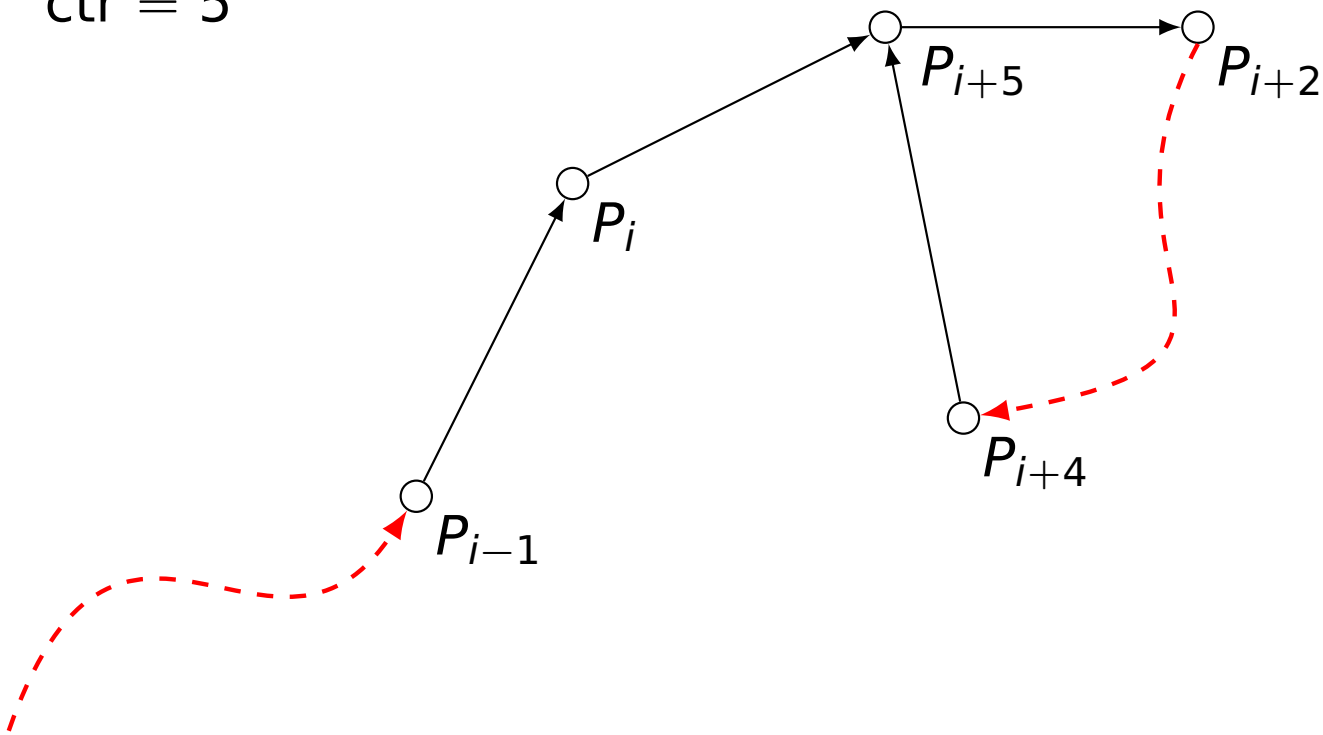
$\mathrm{ctr} = 4$



Negation map

$x_{\min} = x(P_{i+2})$

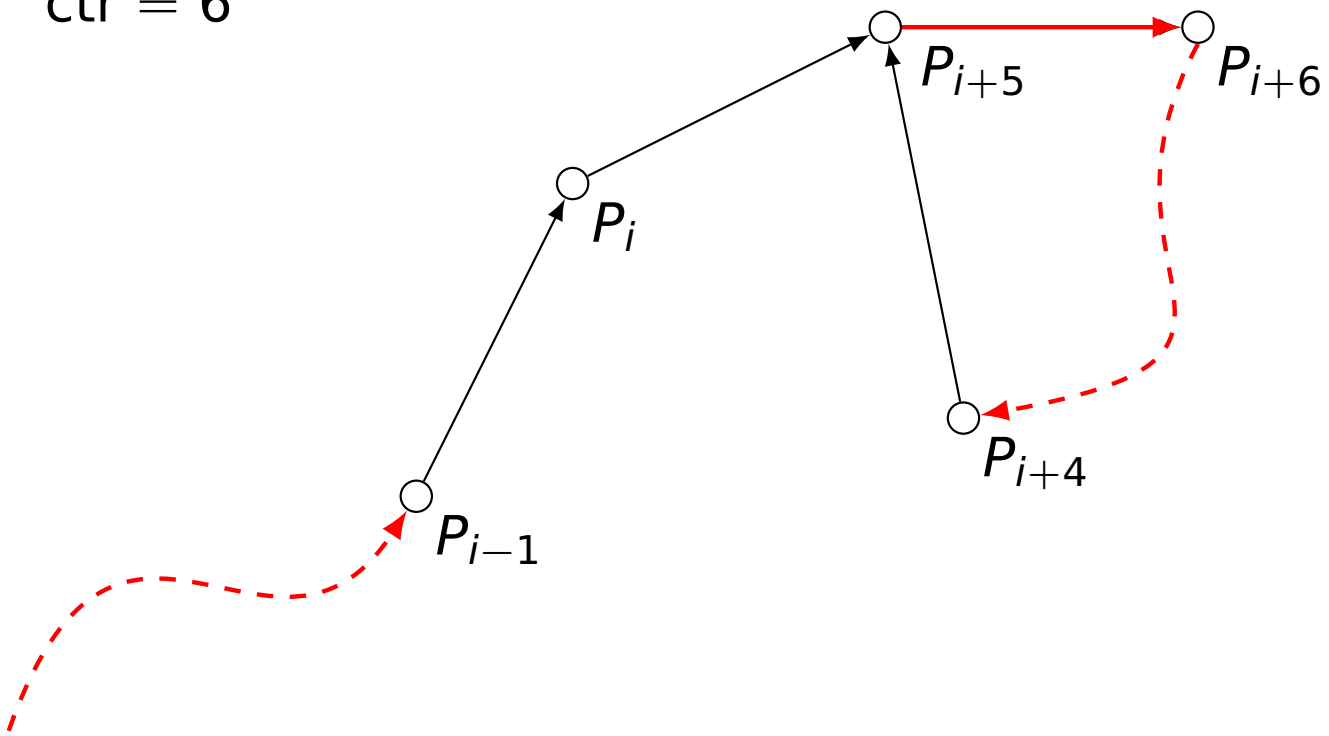
$\mathrm{ctr} = 5$



Negation map

$x_{\min} = x(P_{i+2})$

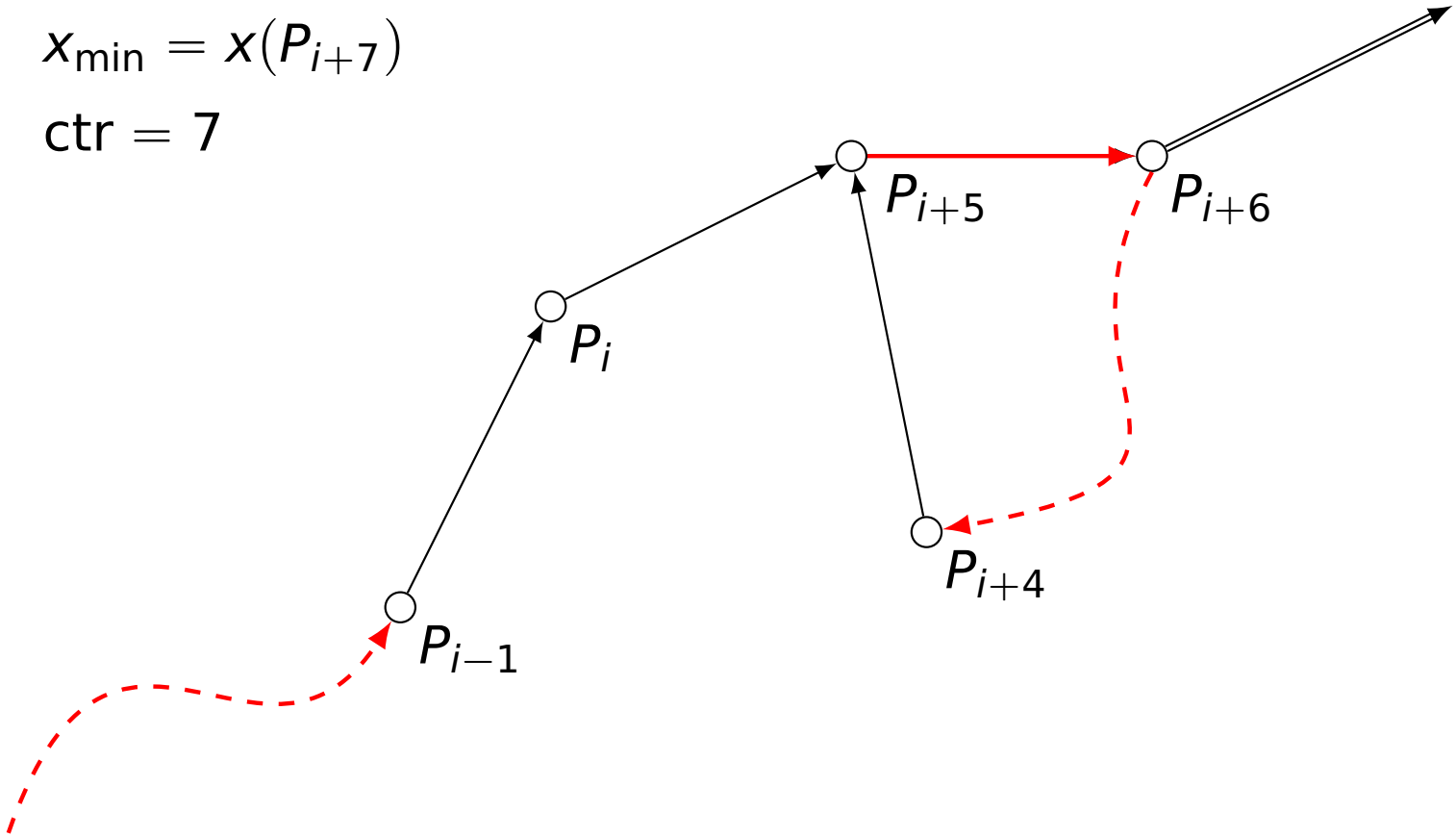
$\mathrm{ctr} = 6$



Negation map

$x_{\min} = x(P_{i+7})$

$\mathrm{ctr} = 7$



How likely are cycles?

n: entries in lookup table; $\delta = 1/n$.

t-cycle	probability	$n = 2^{10}$
2	$\frac{1}{2}\delta$	$= 2^{-11}$
4	$\frac{1}{4}\delta^2 - \ldots$	$\approx 2^{-22}$
6	$\frac{1}{2}\delta^3 - \ldots$	$\approx 2^{-31}$
8	$\frac{27}{16}\delta^4 - \ldots$	$\approx 2^{-39}$
10	$\frac{31}{4}\delta^5 - \ldots$	$\approx 2^{-47}$
12	$\frac{1415}{32}\delta^6 - \ldots$	$\approx 2^{-54}$
14	$\frac{4779}{16}\delta^7 - \ldots$	$\approx 2^{-61}$

Implementing the iteration function


```
def random_step(x, y, seed, double, ctr, x_min):
  (P_x, P_y) = get_precomputed(x)

  x2 = x if double else P_x

  l0 = (x^2 + y) if double else (y + P_y)
  l1 = x if double else (x + P_x)
  l1 = 1/l1
  l = l0 * l1

  tmp = l^2 + l + 1 + x2
  x = x + tmp

  tmp = l * tmp
  y = tmp + y + x

  c_x_y = (x + y) < y
  y = (x + y) if c_x_y else y
```

$$(x_3, y_3) = (\lambda^2 + \lambda + 1 + x_1 + x_2,\ \lambda(x_1 + x_3) + y_1 + x_3), \text{ where}$$

$$\lambda = \begin{cases} (x_1^2 + y_1)/x_1 & \text{for doubling} \\ (y_1 + y_2)/(x_1 + x_2) & \text{for addition} \end{cases}$$

Implementing the iteration function


```
def random_step(x, y, seed, double, ctr, x_min):
  (P_x, P_y) = get_precomputed(x)

  x2 = x if double else P_x

  l0 = (x^2 + y) if double else (y + P_y)
  l1 = x if double else (x + P_x)
  l1 = 1/l1
  l = l0 * l1

  tmp = l^2 + l + 1 + x2
  x = x + tmp

  tmp = l * tmp
  y = tmp + y + x

  c_x_y = (x + y) < y
  y = (x + y) if c_x_y else y
```

$$(x_3, y_3) = (\lambda^2 + \lambda + 1 + x_1 + x_2,\ \lambda(x_1 + x_3) + y_1 + x_3), \text{ where}$$

$$\lambda = \begin{cases} (x_1^2 + y_1)/x_1 & \text{for doubling} \\ (y_1 + y_2)/(x_1 + x_2) & \text{for addition} \end{cases}$$

Check for cycles


```
ctr = (ctr + 1) % 16

c_ctr = (ctr == 0)

c_lt = (x < x_min)

c_new_min = c_lt or c_ctr or double

double = (x == x_min)

x_min = x if c_new_min else x_min

ctr = 0 if c_lt else ctr

return (x, y, seed, ctr, double, x_min)
```

Inversion in $\mathbb{F}_{2^{113}}$

For $x \in \mathbb{F}_{2^{113}}$ compute x^{-1} as

$$x^{-1} = x^{2^{113}-2}$$

using an efficient addition chain.

```
def GF113_inv(din):
  r1 = din^(2^1)
  r2 = din^(2^2)
  r1 = r2 * r1
  r0 = r1^(2^2)
  r1 = r0 * r1
  r0 = r1^(2^4)
  r1 = r0 * r1
  r0 = r1^(2^8)
  r2 = r0 * r1
  r0 = r2^(2^16)
  r3 = r0 * r2
  r0 = r3^(2^32)
  r1 = r0 * r3
  r0 = r1^(2^32)
  r1 = r0 * r3
  r0 = r1^(2^16)
  r0 = r0 * r2
  return r0
```

Inversion in $\mathbb{F}_{2^{113}}$

For $x \in \mathbb{F}_{2^{113}}$ compute x^{-1} as

$$x^{-1} = x^{2^{113}-2}$$

using an efficient addition chain.

```
def GF113_inv(din):
  r1 = din^(2^1)
  r2 = din^(2^2)
  r1 = r2 * r1
  r0 = r1^(2^2)
  r1 = r0 * r1
  r0 = r1^(2^4)
  r1 = r0 * r1
  r0 = r1^(2^8)
  r2 = r0 * r1
  r0 = r2^(2^16)
  r3 = r0 * r2
  r0 = r3^(2^32)
  r1 = r0 * r3
  r0 = r1^(2^32)
  r1 = r0 * r3
  r0 = r1^(2^16)
  r0 = r0 * r2
  return r0
```

FPGA implementation — Building blocks

Multiplication:

- 3-level Karatsuba (and optimized LUT assignment)
- pipelined with 3-cycle latency
- area: $\approx$ 3,100 LUTs

FPGA implementation — Building blocks

Multiplication:

- 3-level Karatsuba (and optimized LUT assignment)
- pipelined with 3-cycle latency
- area: $\approx$ 3,100 LUTs

Pow:

- logic for x^2, x^{2^2}, x^{2^4}, and x^{2^8}
- 1-cycle latency
- area: 56-126 LUTs

FPGA implementation — Building blocks

Multiplication:

- 3-level Karatsuba (and optimized LUT assignment)
- pipelined with 3-cycle latency
- area: $\approx$ 3,100 LUTs

Pow:

- logic for x^2, x^{2^2}, x^{2^4}, and x^{2^8}
- 1-cycle latency
- area: 56-126 LUTs

Add:

- “no” latency
- area: 60-113 LUTs (can be combined with other logic)

Design recipes:

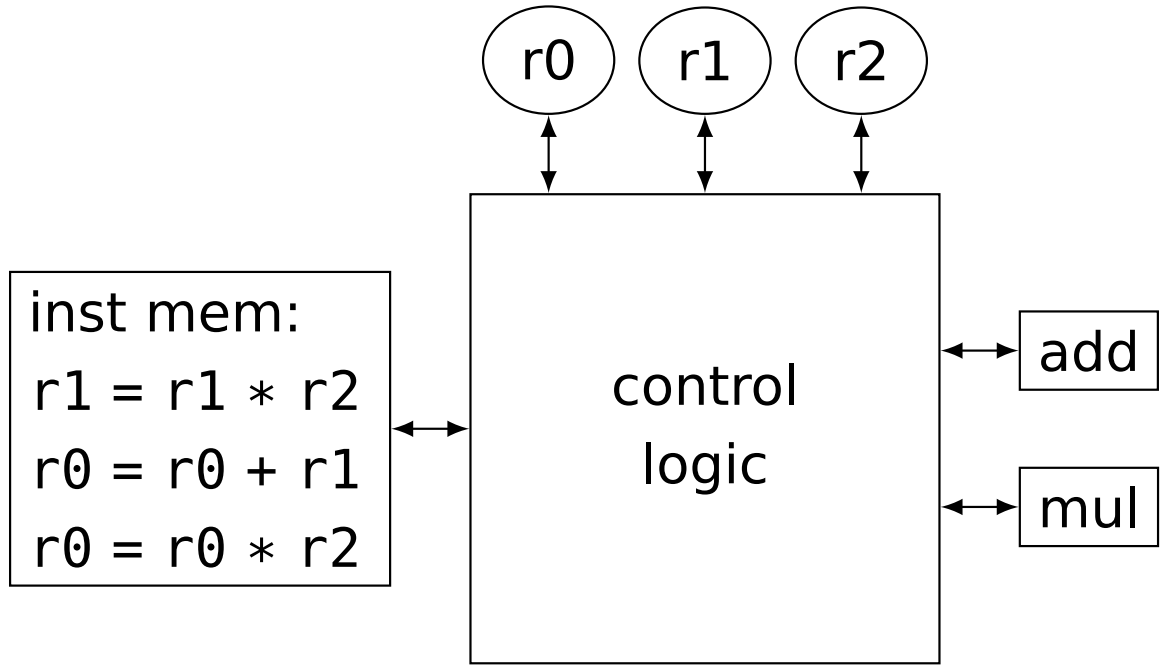
- ASIP: Application Specific Instruction-set Processor — “CPU” with specialized instruction set and register bank.
  - \+ very compact
  - − large overhead
  - ○ low latency, low throughput
  - − not easy to use ALUs efficiently

Design recipes:

- ASIP: Application Specific Instruction-set Processor — “CPU” with specialized instruction set and register bank.
  - \+ very compact
  - − large overhead
  - o low latency, low throughput
  - − not easy to use ALUs efficiently
- Unrolled: fully pipelined, one ALU per operation.
  - − large area
  - \+ low overhead
  - o high latency, high throughput
  - \+ relatively easy to use ALUs efficiently
  - o requires many independent inputs

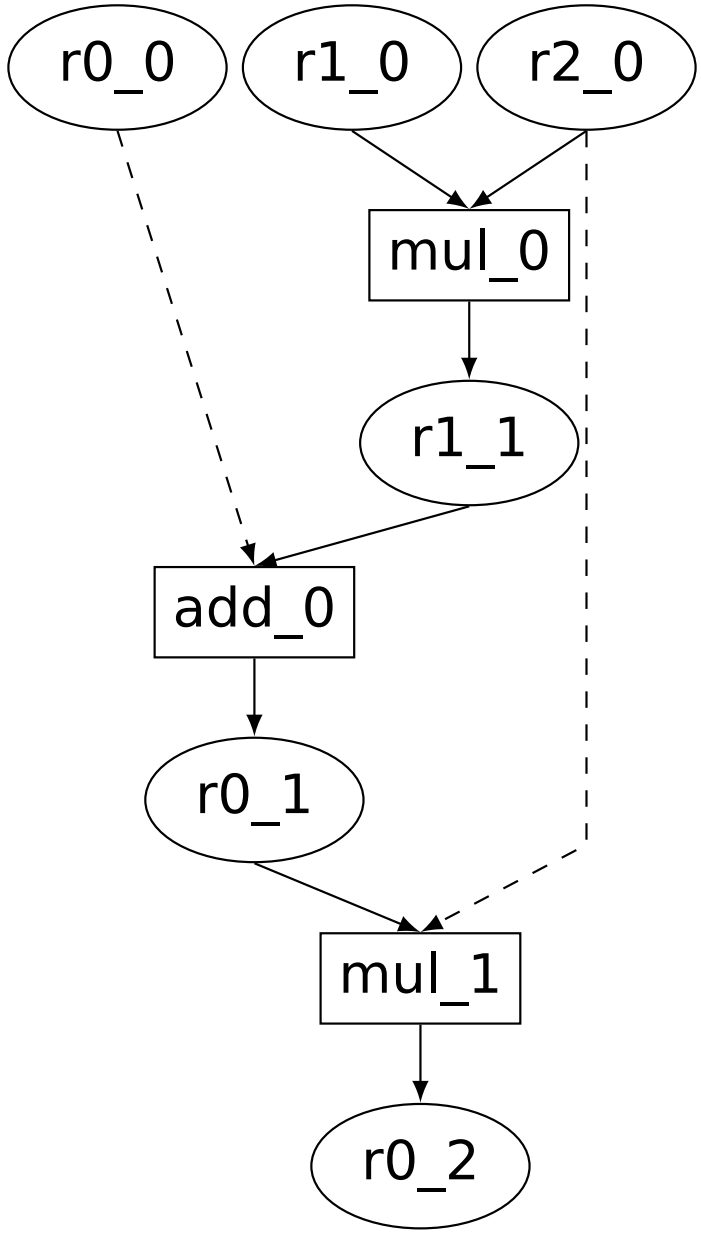
Example: ASIP


```
var r0, r1, r2

r1 = r1 * r2
r0 = r0 + r1
r0 = r0 * r2
```



Example: unrolled


```
var r0, r1, r2

r1 = r1 * r2
r0 = r0 + r1
r0 = r0 * r2
```



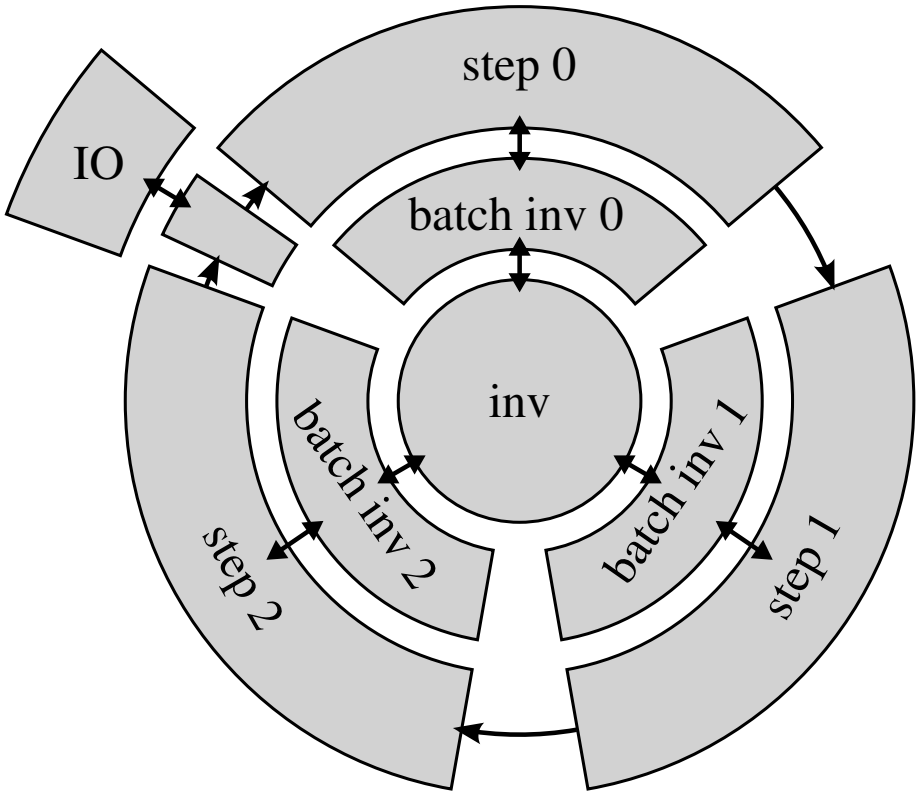
ASIP or unrolled design?

"Hybrid" approach:

- Unrolled design for iteration function,
- ASIP design for inverter.

→ Requires Montgomery's trick for inversion and large buffers.

Final design



Parameters:

- 3 instances
- power consumption:
  - 8W
- area consumption:
  - 69% LUTs
  - 19% memory
  - 91% slices

Runtime:

The whole computation requires about $2^{56.3}$ iterations.

Our FPGA cluster has 120 FPGAs.

We expected the computation for a 2-core ASIP design to be finished after

$$\frac{\sqrt{\frac{\pi}{2} \cdot 2^{112} \cdot \frac{1}{2}}}{100MHz} \cdot \frac{1}{120} \cdot \frac{1}{2} \text{ seconds} \approx 31 \text{ days.}$$

Eventually, it took us about 48 days to compute the ECDLP.

Results

Larger DLP:

Currently we are running the attack on a generic curve over $\mathbb{F}_{2^{127}}$ with base point of order $2^{117.35}$.

- expected number of DPs: 379,821,956

Results

Larger DLP:

Currently we are running the attack on a generic curve over $\mathbb{F}_{2^{127}}$ with base point of order $2^{117.35}$.

- expected number of DPs: 379,821,956
- currently computed DPS: 851,337,056

Results

Larger DLP:

Currently we are running the attack on a generic curve over $\mathbb{F}_{2^{127}}$ with base point of order $2^{117.35}$.

- expected number of DPs: 379,821,956
- currently computed DPS: 851,337,056
- expected time to finish: last May

http://eprint.iacr.org/2016/382

Thank you for your attention.