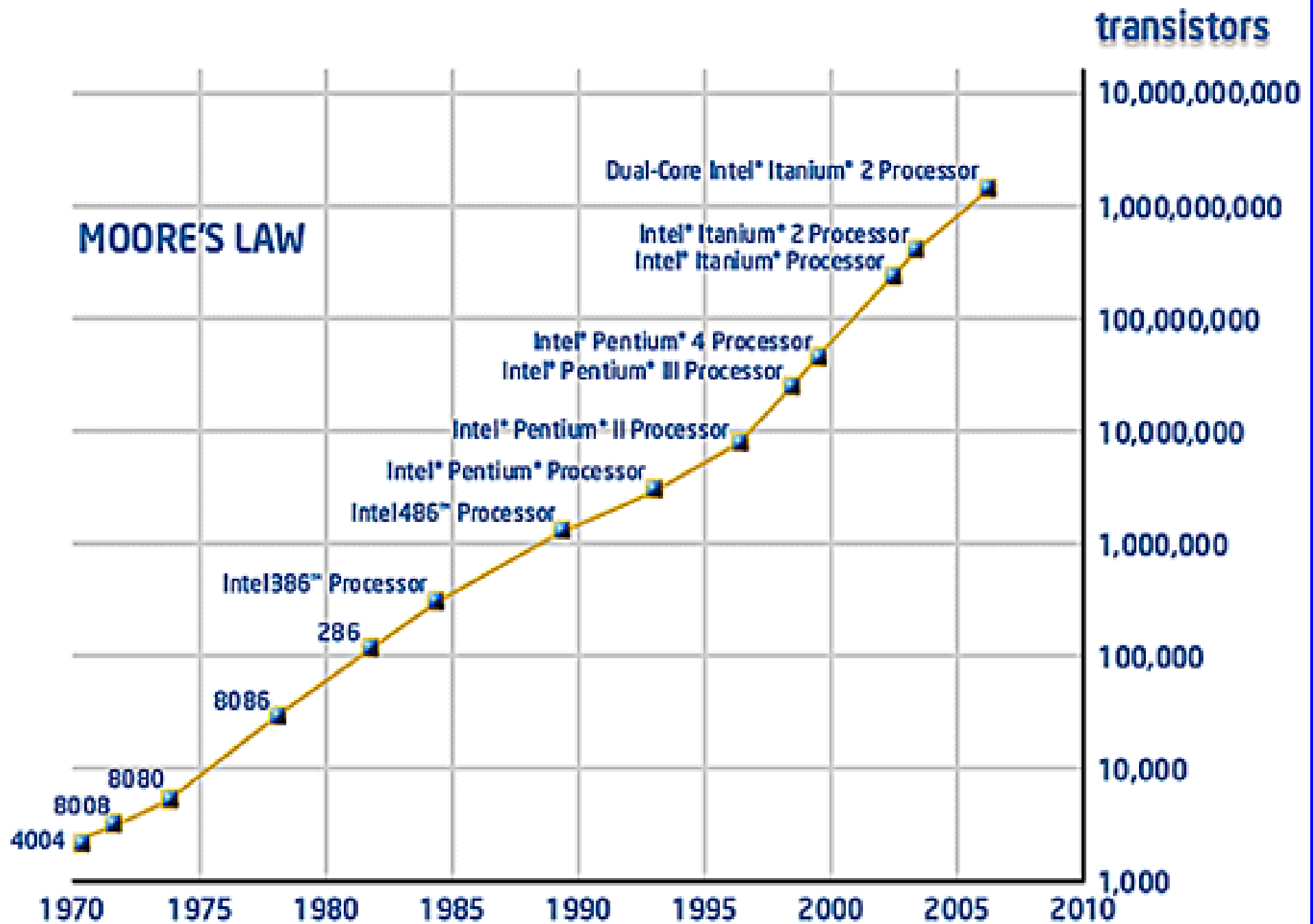




Self-Modifying Circuitry for Efficient, Defect-Tolerant Handling of Trillion-Element Reconfigurable Devices

Nicholas J. Macias
29 April 2011



Hypothesis About Future Computing

1. Based on increasingly-large (scalable) reconfigurable devices

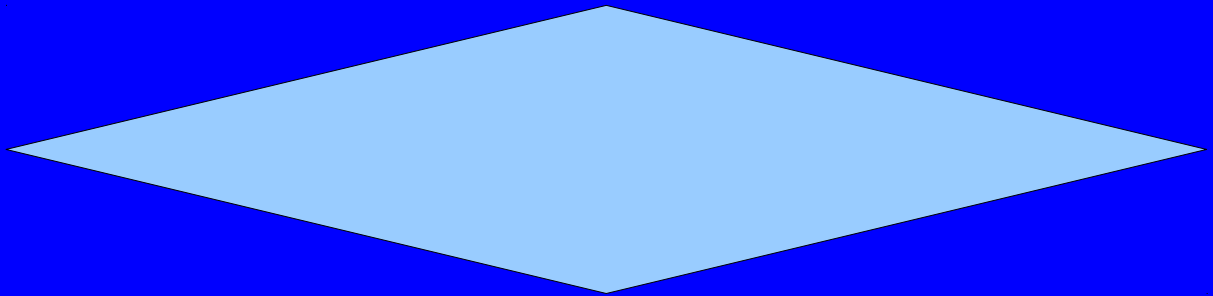
Hypothesis About Future Computing

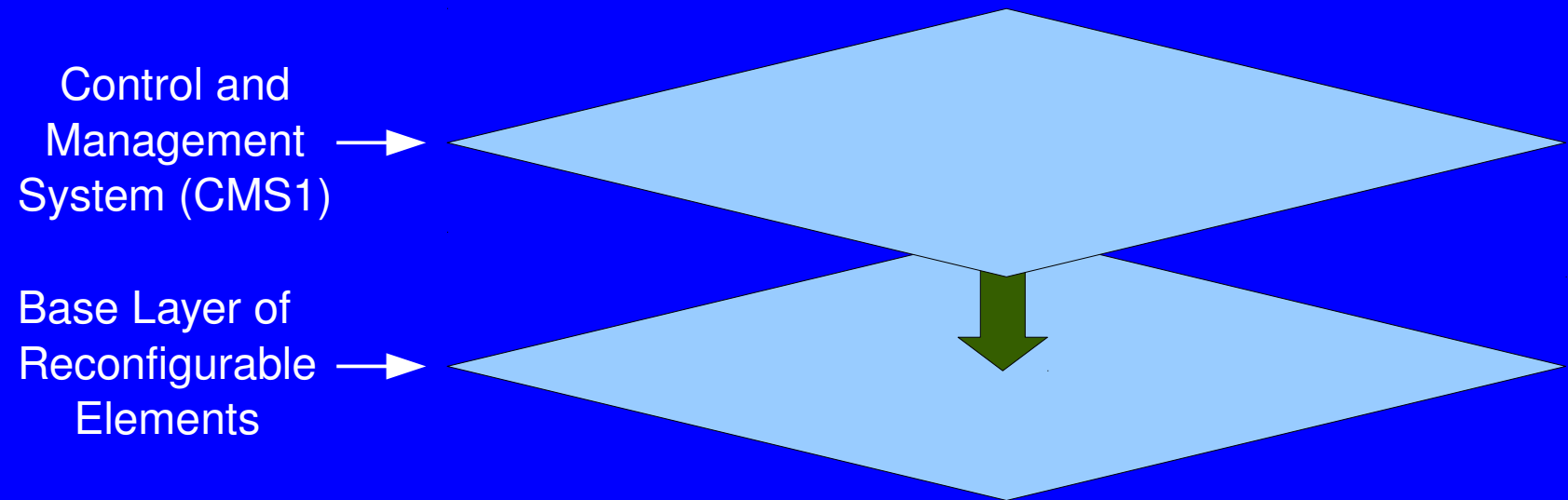
1. Based on increasingly-large (scalable) reconfigurable devices
2. Helpful to have control and management located *inside* the device

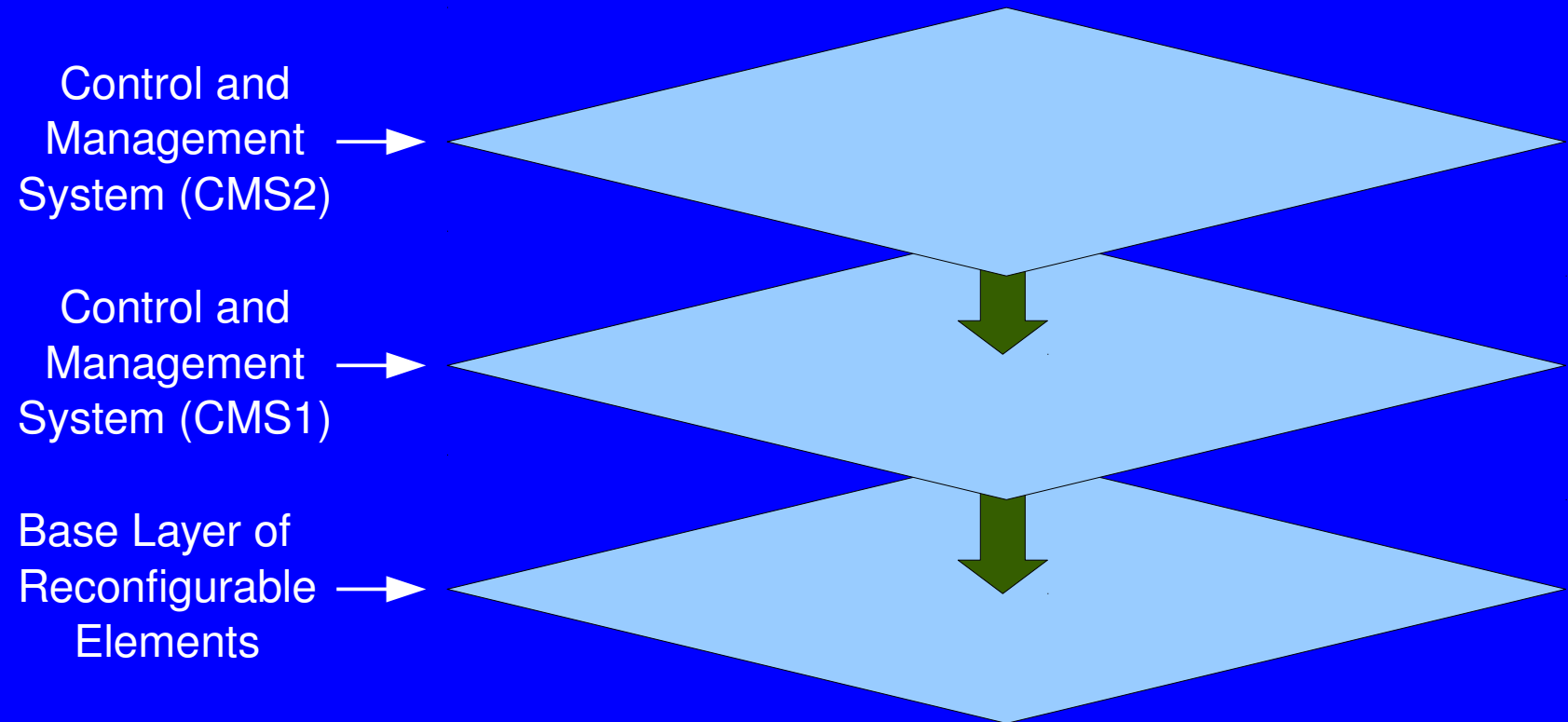
Hypothesis About Future Computing

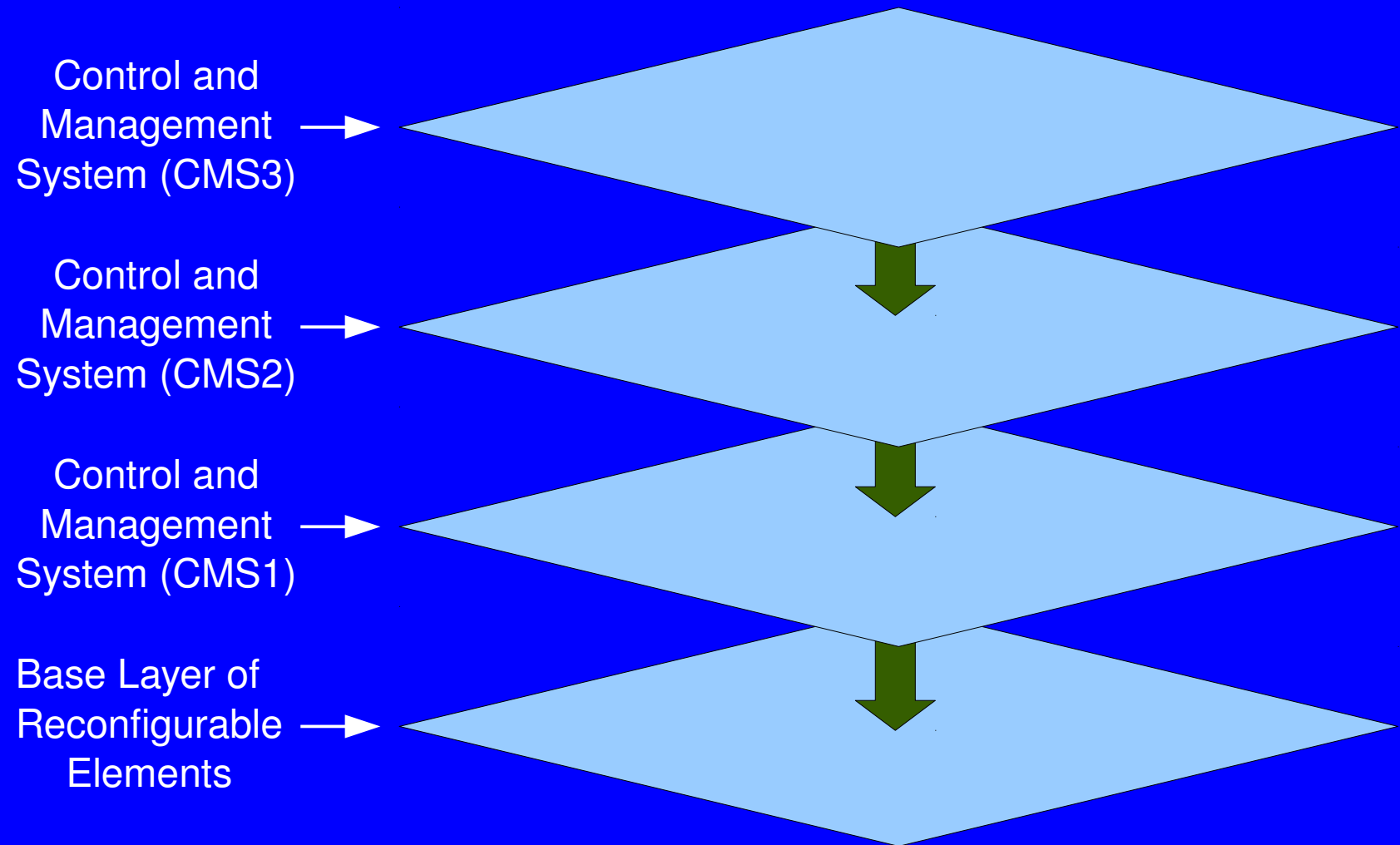
1. Based on increasingly-large (scalable) reconfigurable devices
2. Helpful to have control and management located *inside* the device
3. As system scales, it's useful to have the control and management system be reconfigurable

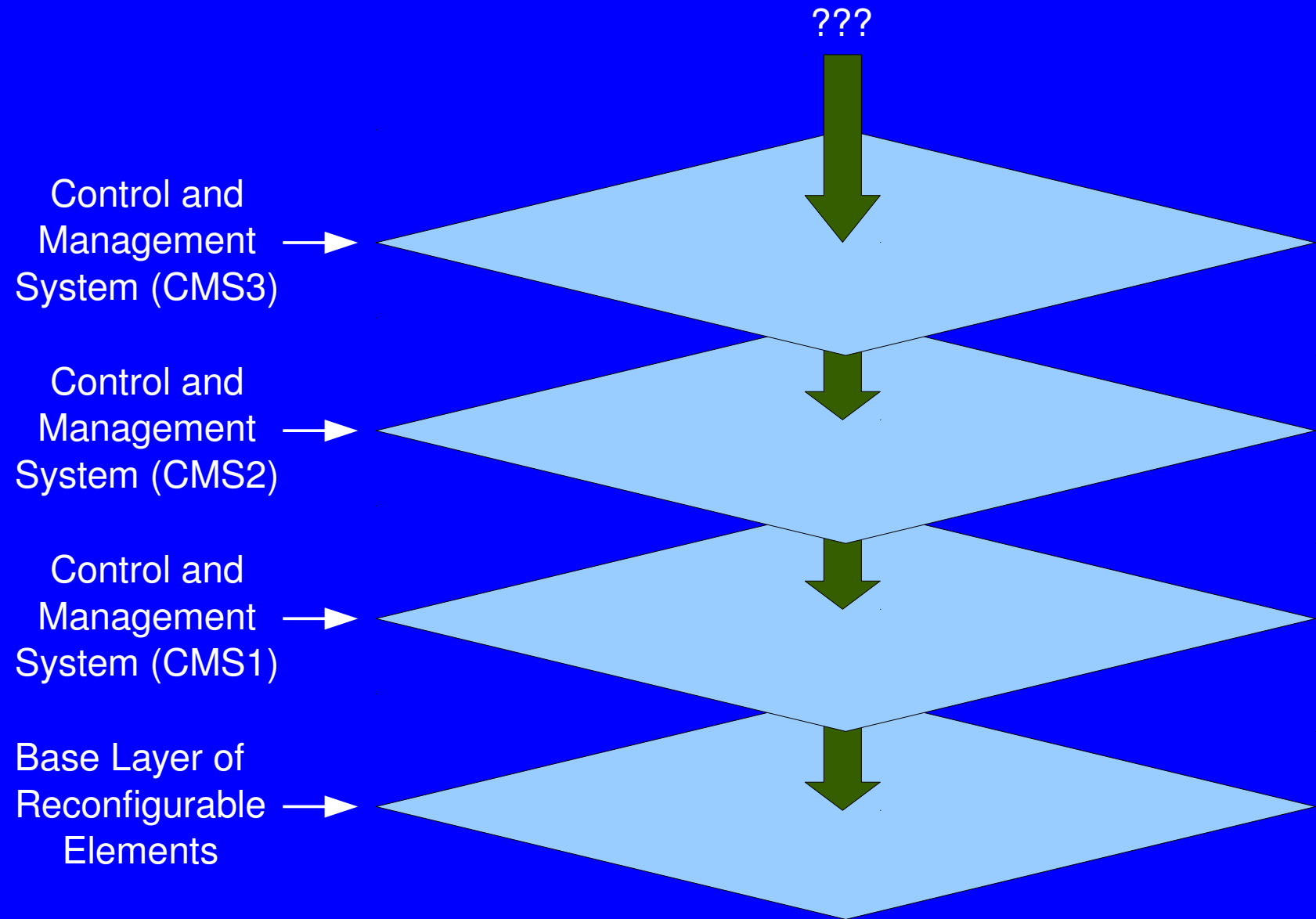
Base Layer of
Reconfigurable
Elements











Hypothesis 4:
A hierarchy-free arrangement of
***controlling* and *controlled* objects**
may be interesting.

“Closing the loop”

Hypothesis 4: “Closing The Loop”

Subject/Object Dualism

Hypothesis 4: “Closing The Loop”

I kick the ball

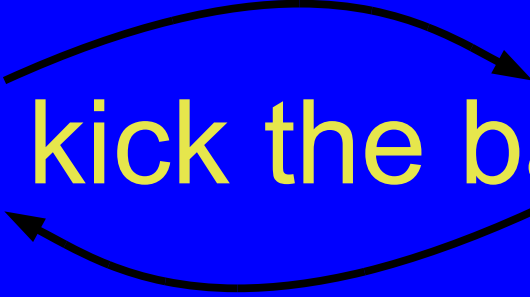
Hypothesis 4: “Closing The Loop”



I kick the ball

Hypothesis 4: “Closing The Loop”

I kick the ball

A diagram illustrating a feedback loop. Two curved arrows form a cycle around the text "I kick the ball". The top arrow points from the left side of the text to the right side, and the bottom arrow points from the right side of the text back to the left side, creating a continuous loop.

Hypothesis 4: “Closing The Loop”

Non-Dualism: Interdependence between
subject and object (or *observer* and *observed*)

Less-Philosophical Example

A program that prints its own source code

1. create an array of strings
2. read that array and print each string
3. initialize the array with the program's source code

```
static char p[100][1024]={  
    (fill this in later)  
};
```

```
static char p[100][1024]={  
    (fill this in later)  
};
```

```
main()  
{  
    int i;  
    for (i=0;i<100;i++)  
        printf("%s\n",p[i]);  
}
```

```
static char p[100][1024]={  
    "static char p[100][1024]={",  
};
```

```
main()  
{  
    int i;  
    for (i=0;i<100;i++)  
        printf("%s\n",p[i]);  
}
```

```
static char p[100][1024]={  
    "static char p[100][1024]={",  
    "\"static char p[100][1024]={\"",  
};
```

```
main()  
{  
    int i;  
    for (i=0;i<100;i++)  
        printf("%s\n",p[i]);  
}
```

```
static char p[100][1024]={  
    "static char p[100][1024]={",  
    "\"static char p[100][1024]={\"",  
    "\\\"static char p[100][1024]={\\\"",  
};
```

```
main()  
{  
    int i;  
    for (i=0;i<100;i++)  
        printf("%s\n",p[i]);  
}
```

```
static char p[100][1024]={
    “static char p[100][1024]={“,
    “\”static char p[100][1024]={\”,”,
    “\\”static char p[100][1024]={\\”\”,”,
    “\\\\”static char p[100][1024]={\\\\”\”,”,
    “\\\\\\\\”static char p[100][1024]={\\\\\\\\”\\”,\”,”,
};
```

```
main()
{
    int i;
    for (i=0;i<100;i++)
        printf("%s\n",p[i]);
}
```

We're treating this dualistically, as two separate pieces

- a program (the *subject*) that reads
- an array of characters (the *object*)

We're treating code and data as distinct

In fact, code and data are interrelated

- When the code (program) changes, the data (string array) needs to also change
- If the data changes, the code needs to change accordingly

```
static char p[20][1024]={
#include <stdio.h>",
#include <string.h>",
",
"main()",
"{",
"    int i,j;",
"",
",
"    printf(\"static char p[20][1024]={\\n\\n}\");",
"    for (i=0;i<20;i++){",
"        if (i>0) printf(\"\\\\\\\",\\n\");",
"        printf(\"\\\\\\\"\\n\");",
"        for (j=0;j<strlen(p[i]);j++){",
"            if ((p[i][j]==\"\\\\\" || p[i][j]==\"\\\\\\\") printf(\"\\\\\\\\\\n\");",
"            printf(\"%c\\\",p[i][j]);",
"        }",
"    }",
"    printf(\"\\\\\\\"};\\n\\n\");",
"",
",
"    for (i=0;i<20;i++) printf(\"%s\\n\\n\",p[i]);",
"}";
```

```
#include <stdio.h>
#include <string.h>

main()
{
    int i,j;

    printf("static char p[20][1024]={\n");
    for (i=0;i<20;i++){
        if (i>0) printf("\",\n");
        printf("\n");
        for (j=0;j<strlen(p[i]);j++){
            if ((p[i][j]=='\" || p[i][j]=='\\')) printf("\\");
            printf("%c",p[i][j]);
        }
    }
    printf("\n}");
    printf("\n\n");

    for (i=0;i<20;i++) printf("%s\n",p[i]);
}
```

```
#include <stdio.h>
#include <string.h>
```

```
main()
{
```

```
    int i,j;
```

```
    printf("static char p[20][1024]={\n");
```

```
    for (i=0;i<20;i++){
```

```
        if (i>0) printf("\",\n");
```

```
        printf("\n");
```

```
        for (j=0;j<strlen(p[i]);j++){
```

```
            if ((p[i][j]=='\" || p[i][j]=='\\') printf("\\");
```

```
            printf("%c",p[i][j]);
```

```
        }
```

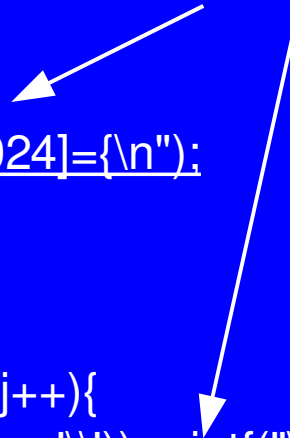
```
    }
```

```
    printf("\n};\n\n");
```

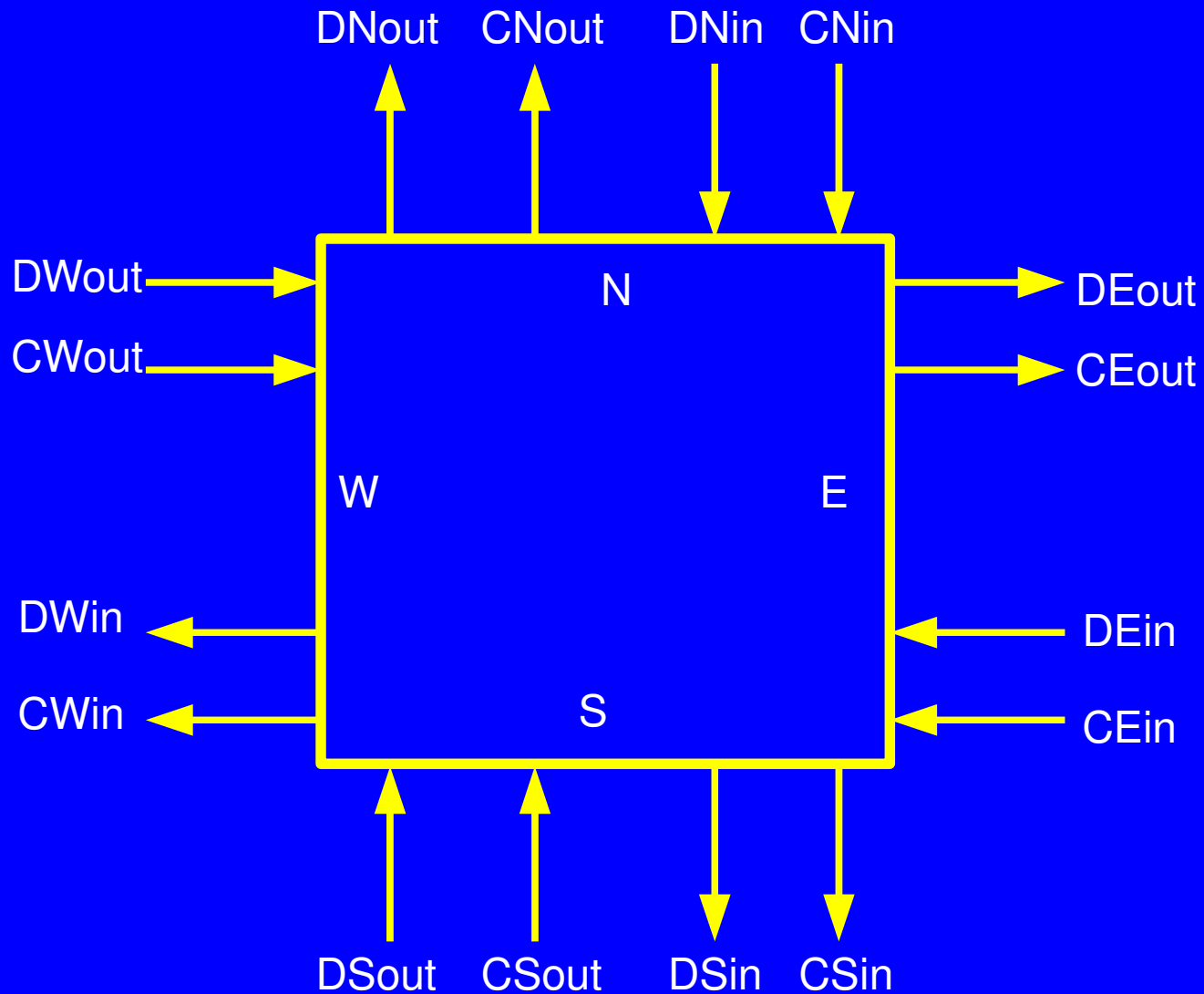
```
    for (i=0;i<20;i++) printf("%s\n",p[i]);
```

```
}
```

Part of the data has been
moved into the code



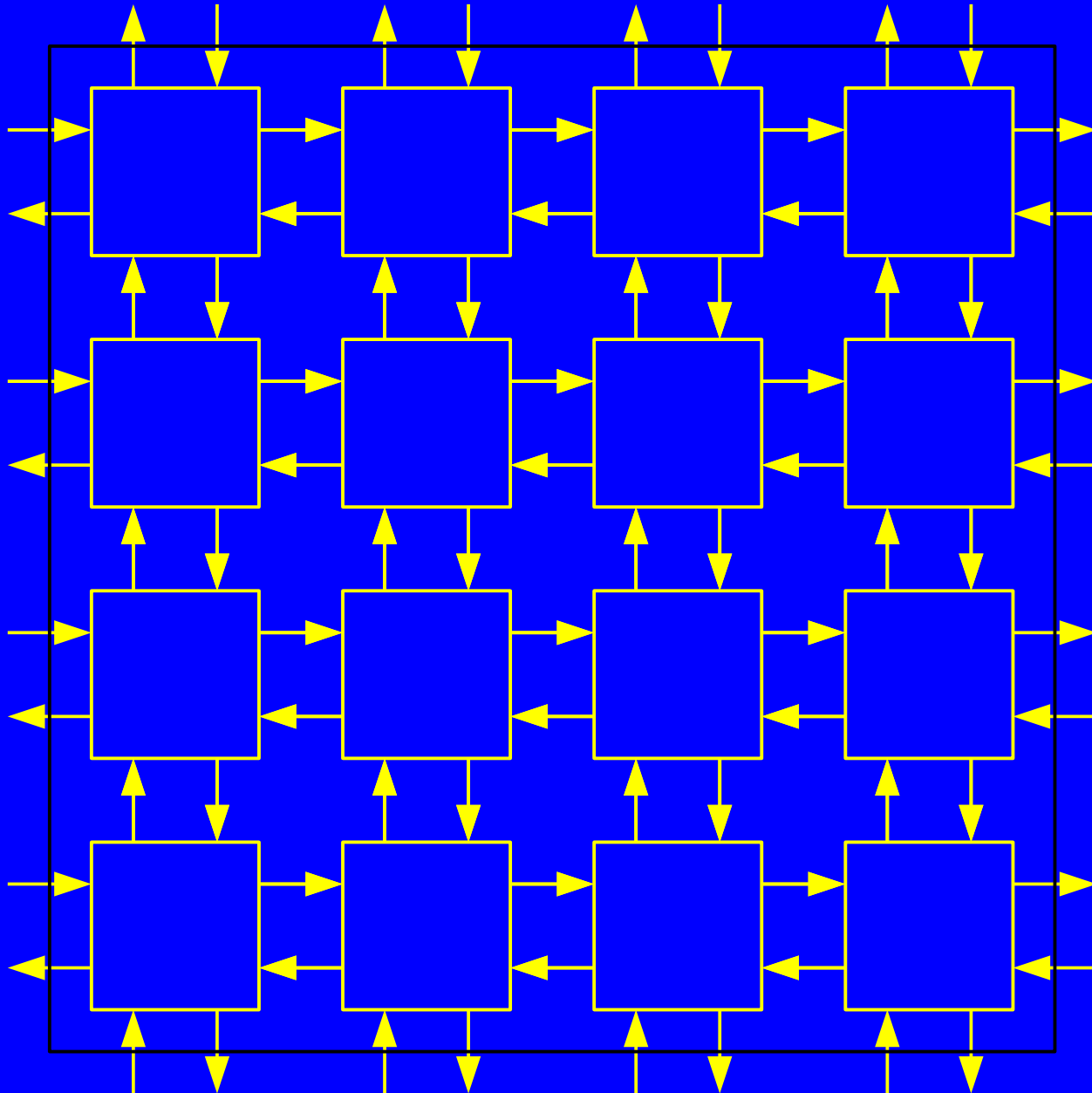
Single Element (4-Sided)



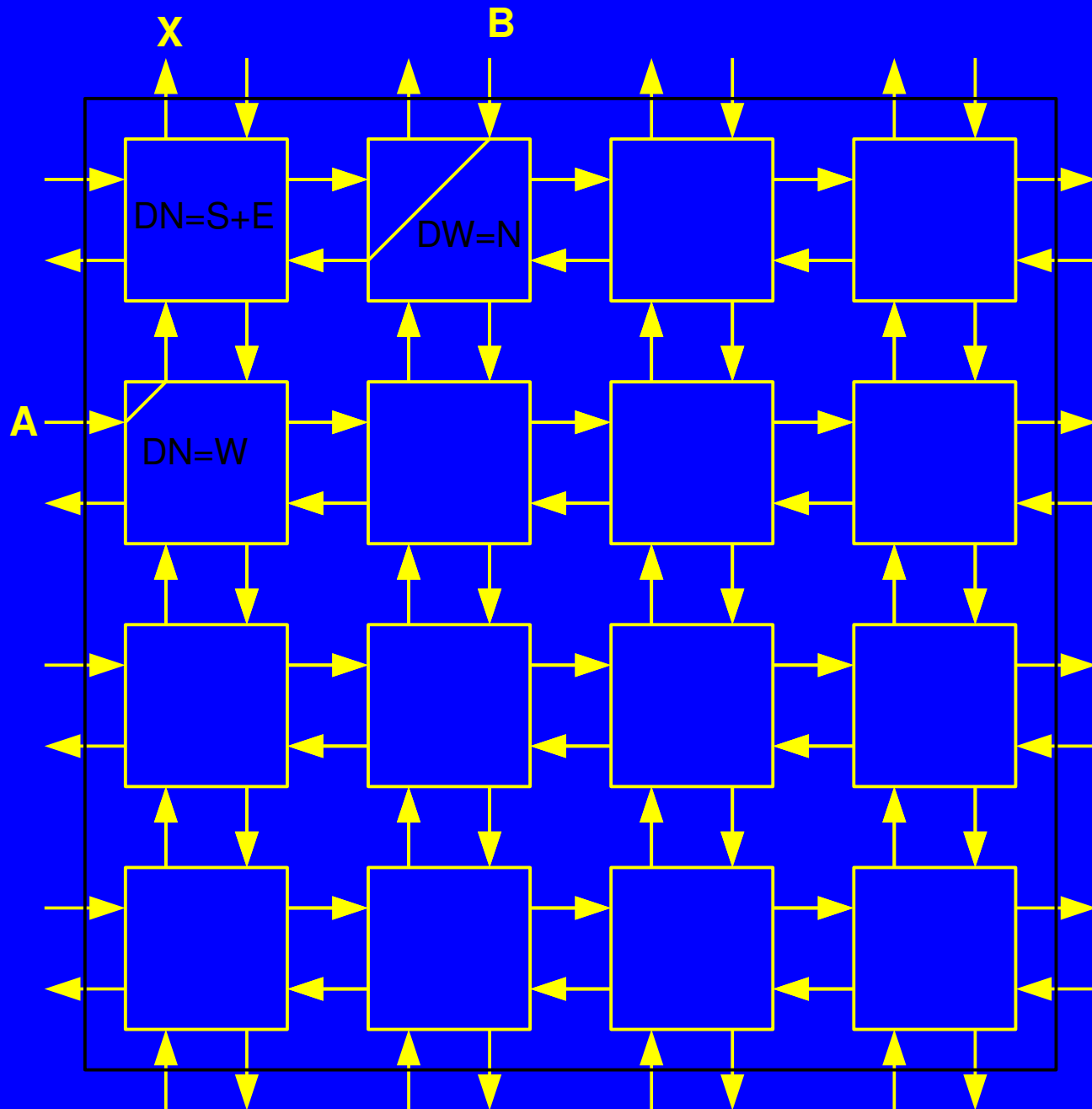
Each element's behavior is controlled by its configuration memory

DNin	DSin	DWin	DEin	CNout	CSout	CWout	CEout	DNout	DSout	DWout	DEout
0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0	0
0	0	1	1	1	0	0	0	0	0	0	0
0	1	0	0	1	0	0	0	0	0	0	0
0	1	0	1	1	0	0	0	0	0	0	0
0	1	1	0	1	0	0	0	0	0	0	0
0	1	1	1	1	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1	1	0	0
1	0	0	1	1	0	0	0	1	1	0	0
1	0	1	0	1	0	0	0	1	1	0	0
1	0	1	1	1	0	0	0	1	1	0	0
1	1	0	0	1	0	0	0	1	1	0	0
1	1	0	1	1	0	0	0	1	1	0	0
1	1	1	0	1	0	0	0	1	1	0	0
1	1	1	1	1	0	0	0	1	1	0	0

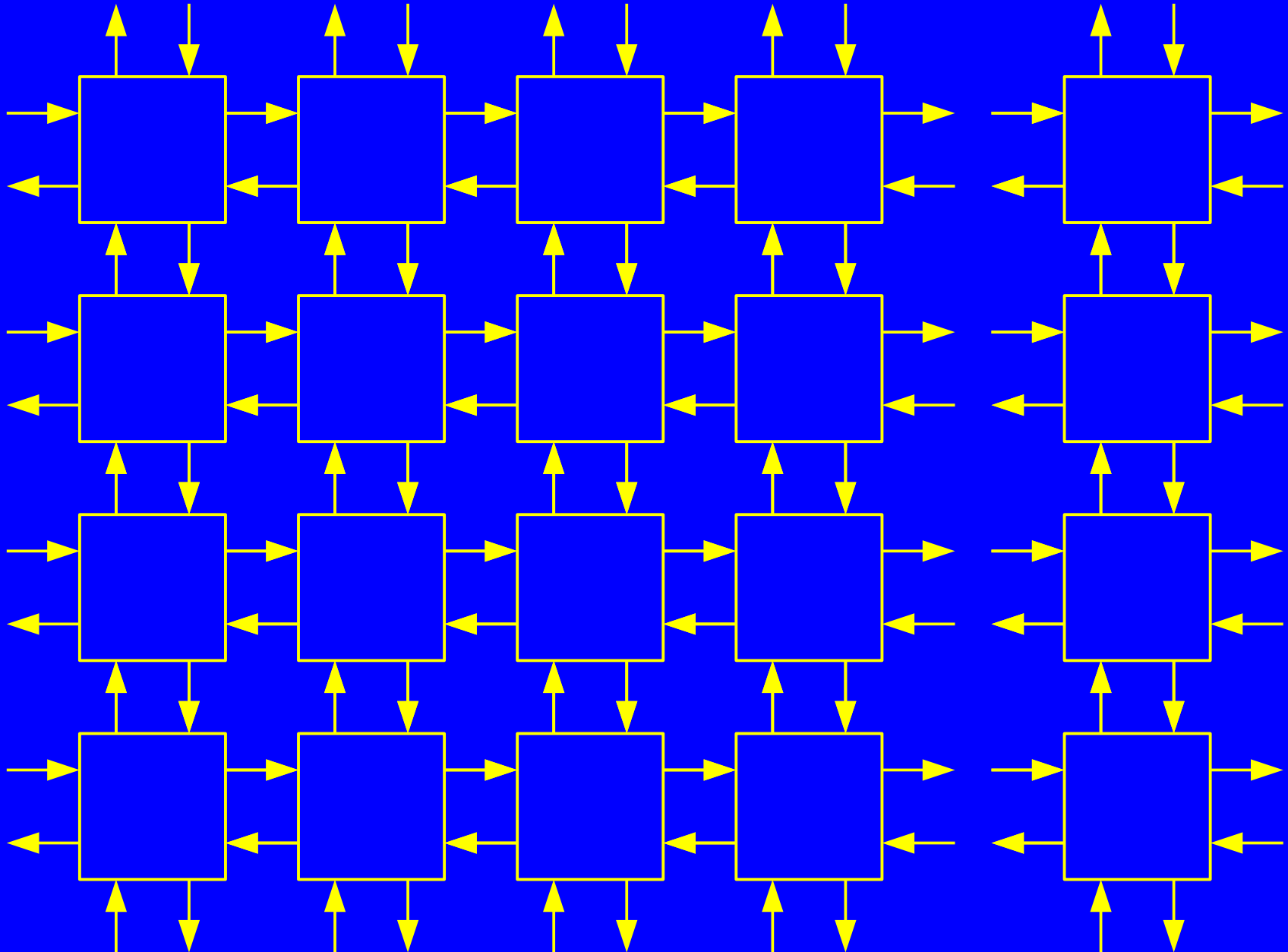
Reconfigurable array is composed of many elements



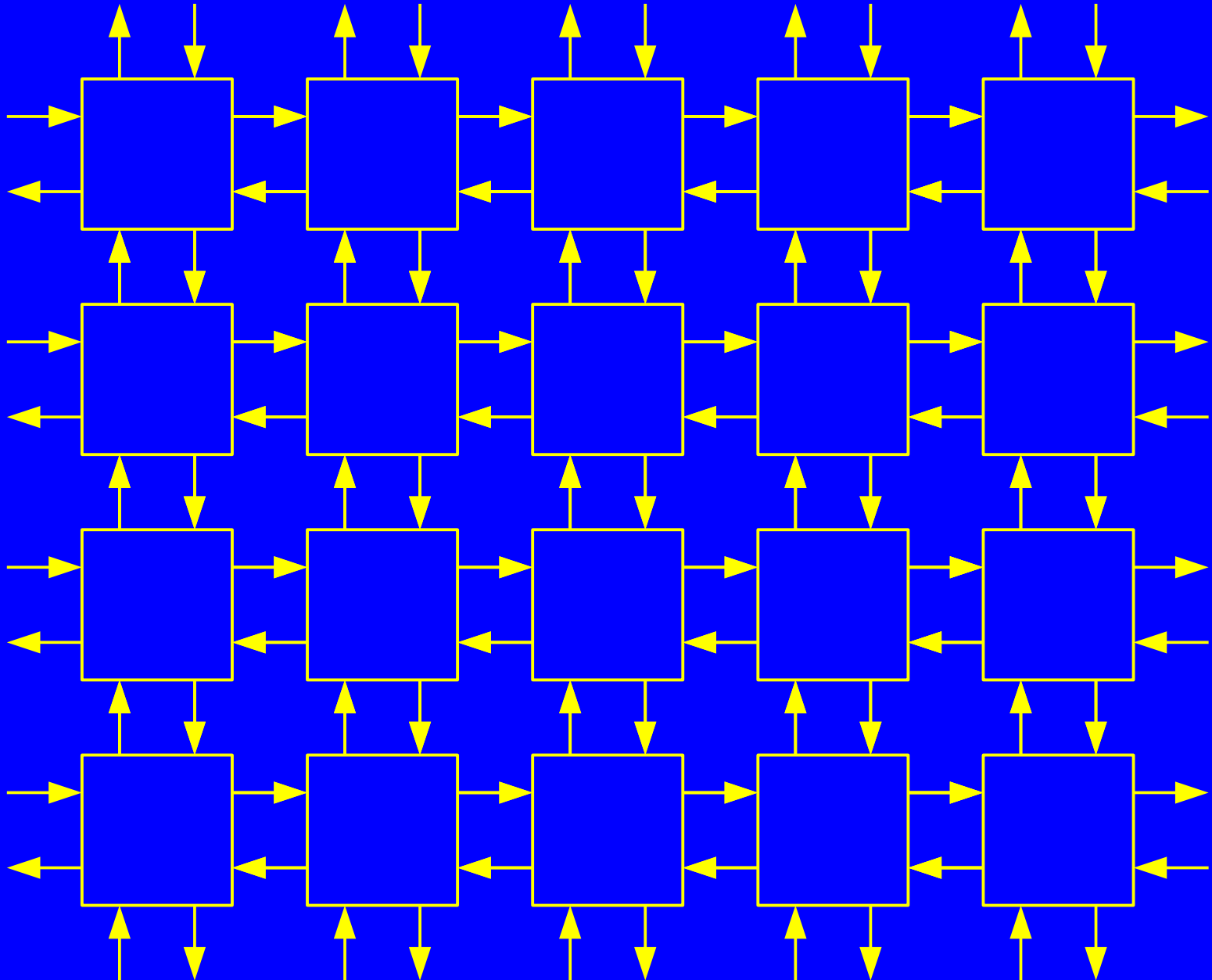
$X=A \text{ Or } B$



**Scalable – can grow by
adding to edges**



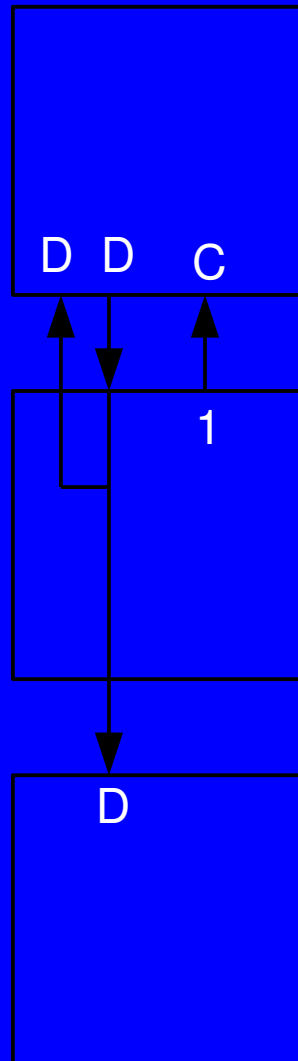
**Scalable – can grow by
adding to edges**



Configuring an Element

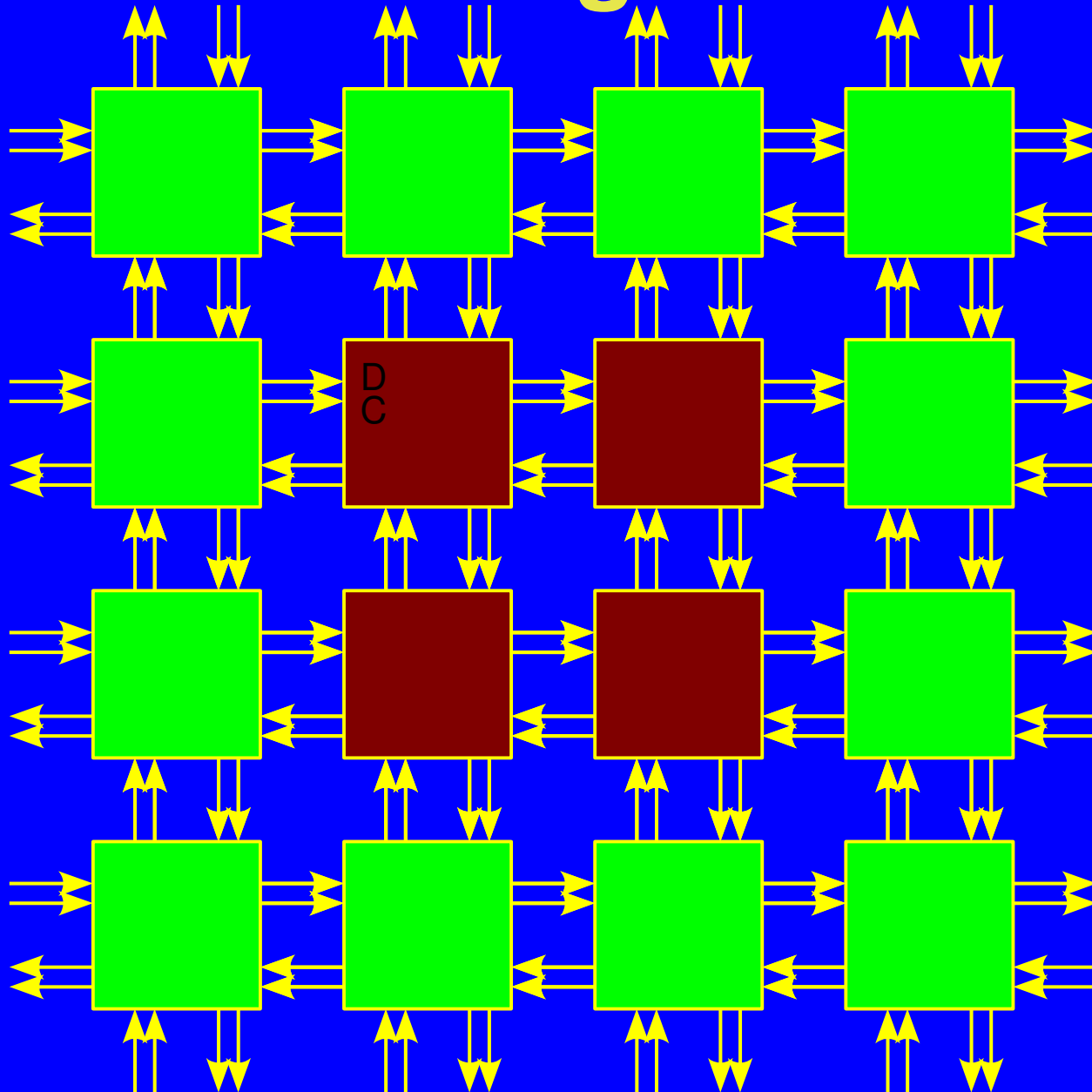
- Set C=1 and shift in a TT (in sync with a system clock) on the corresponding D input
- Can read the previous TT on the D output
- Called “C-Mode” (as opposed to “D-Mode”)

This allows read/write access to TTs for elements *if we can access their C and D inputs*

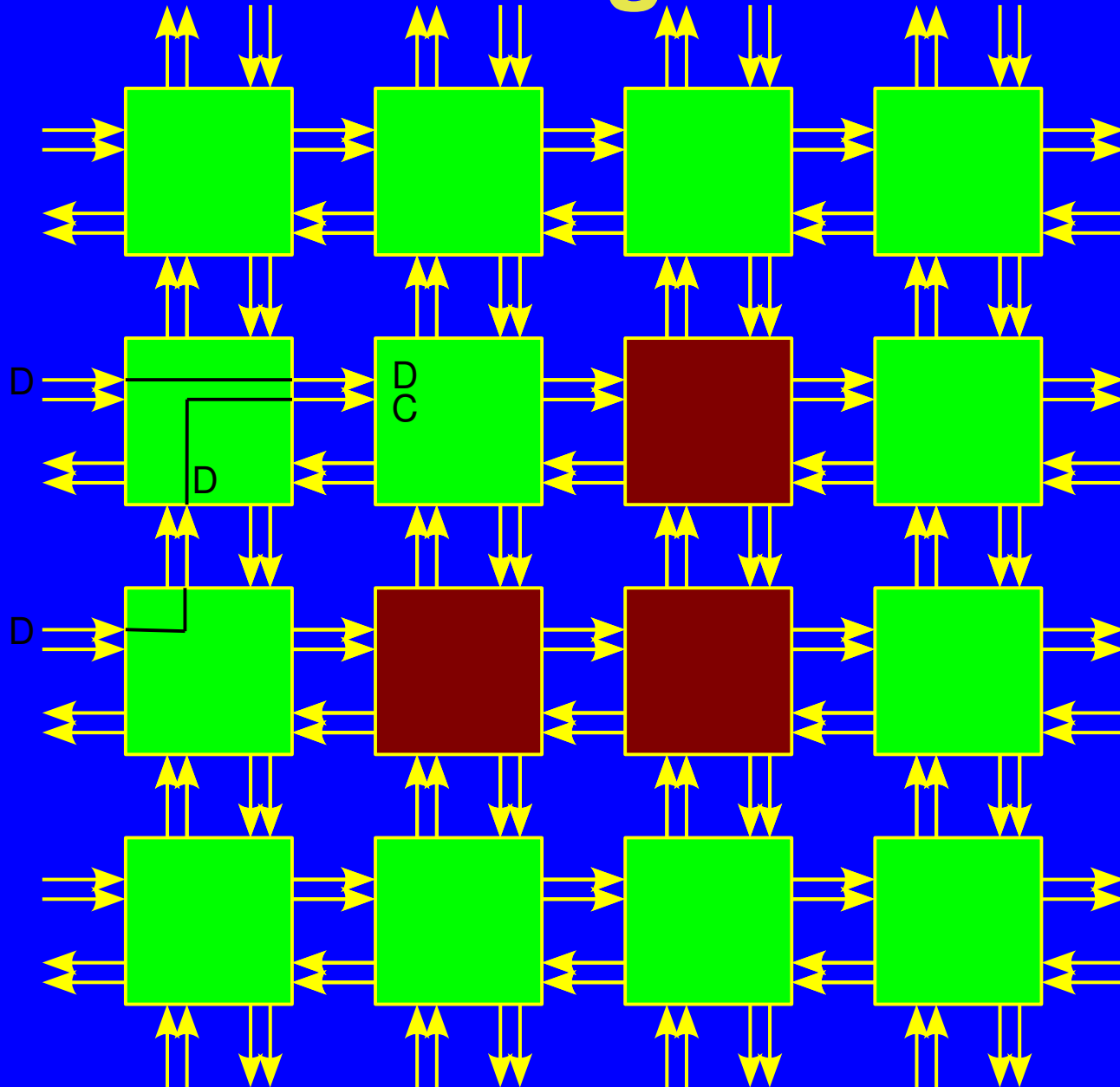


$CN=1 ; DN=N ; DS=N$

How are Non-Edge Elements Configured?

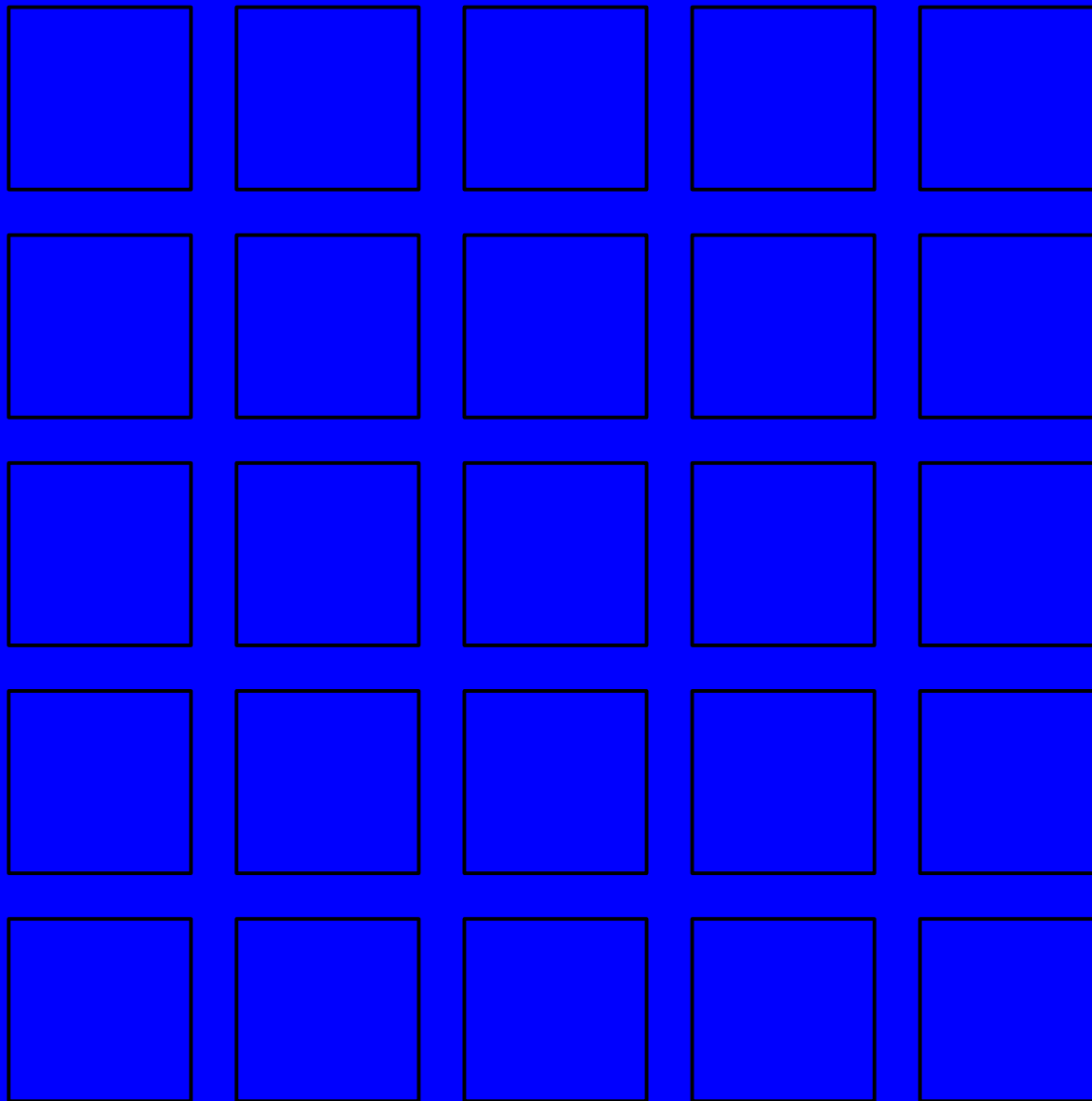


How are Non-Edge Elements Configured?

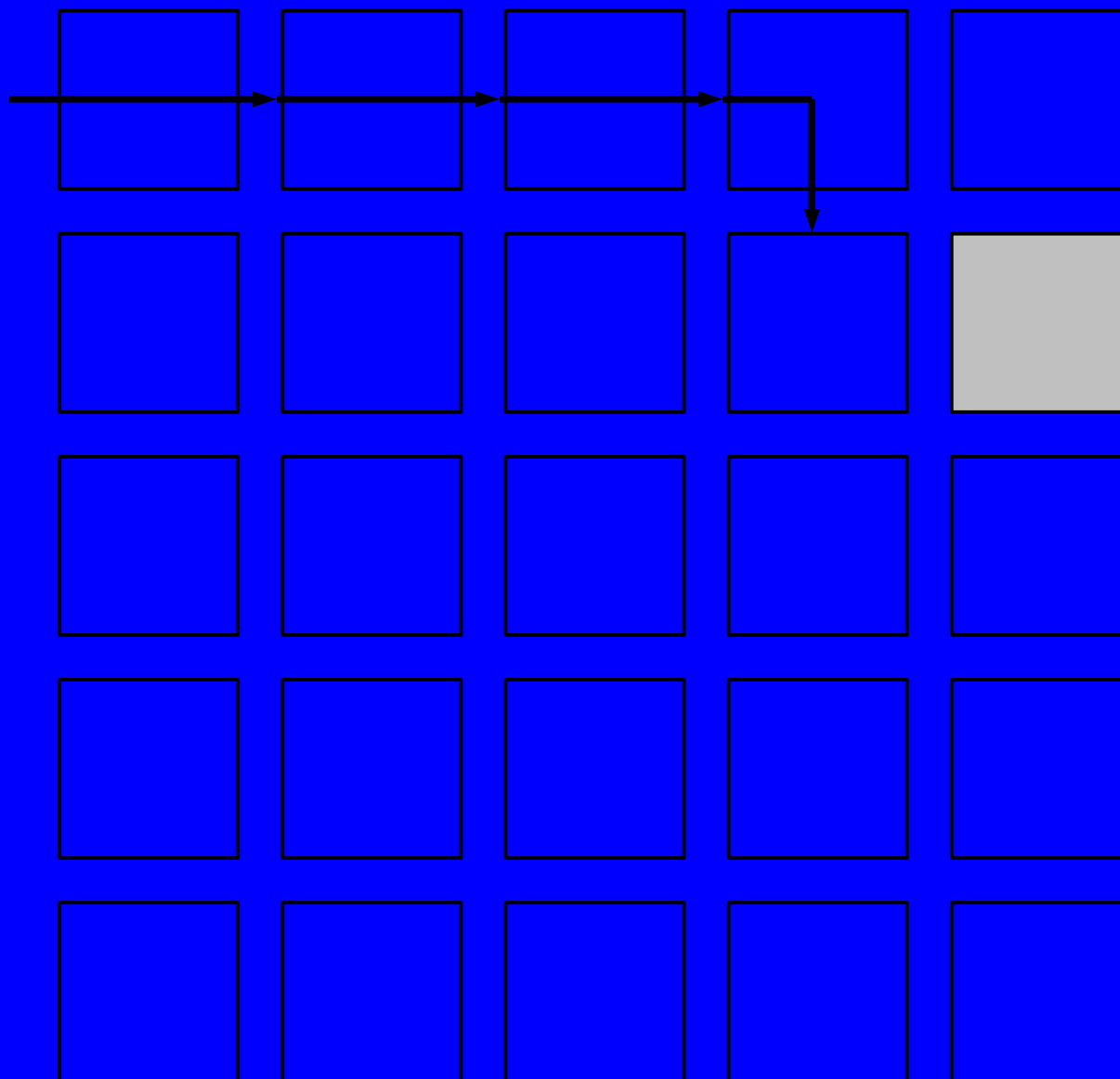


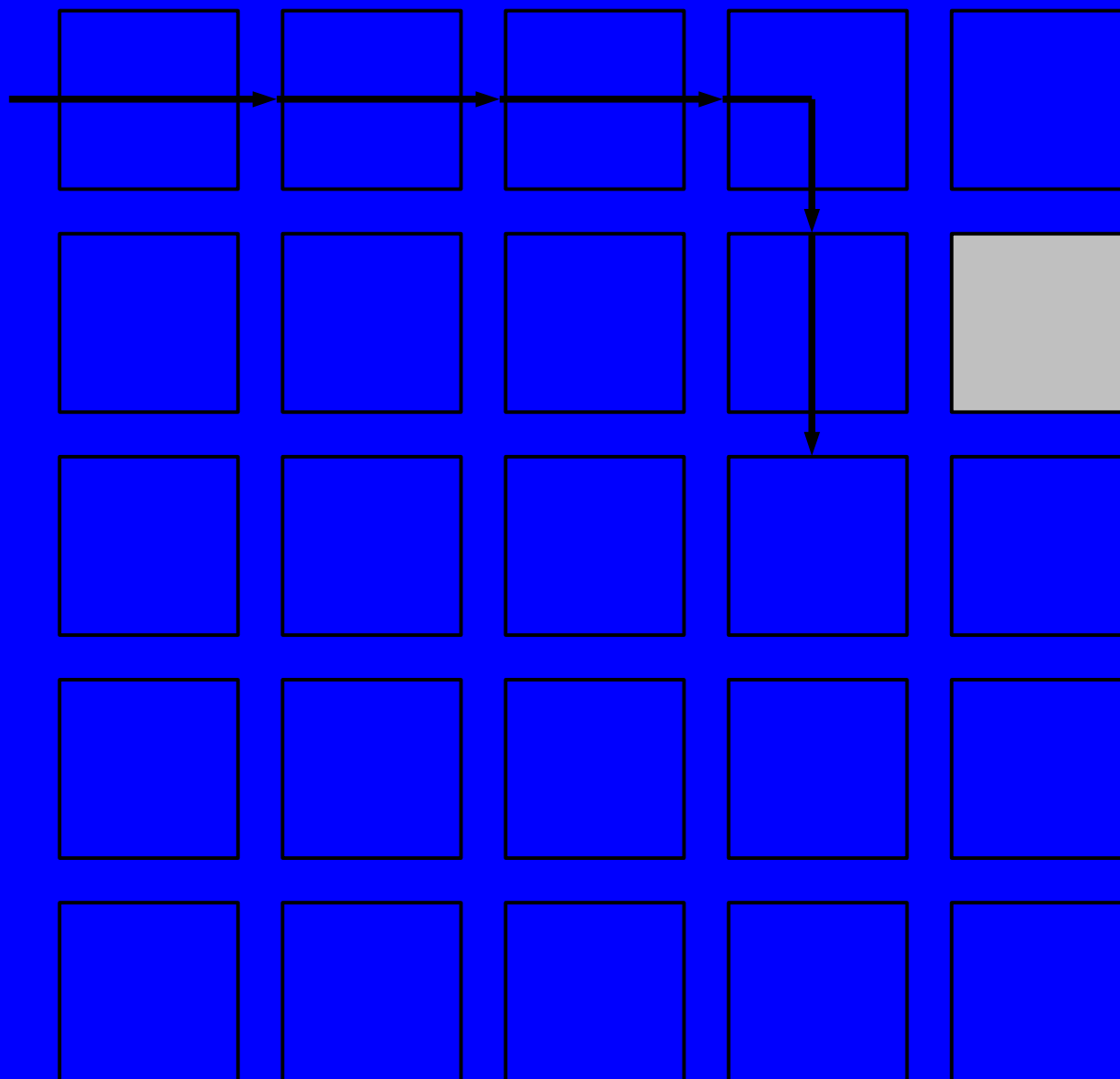
SEQUENCES

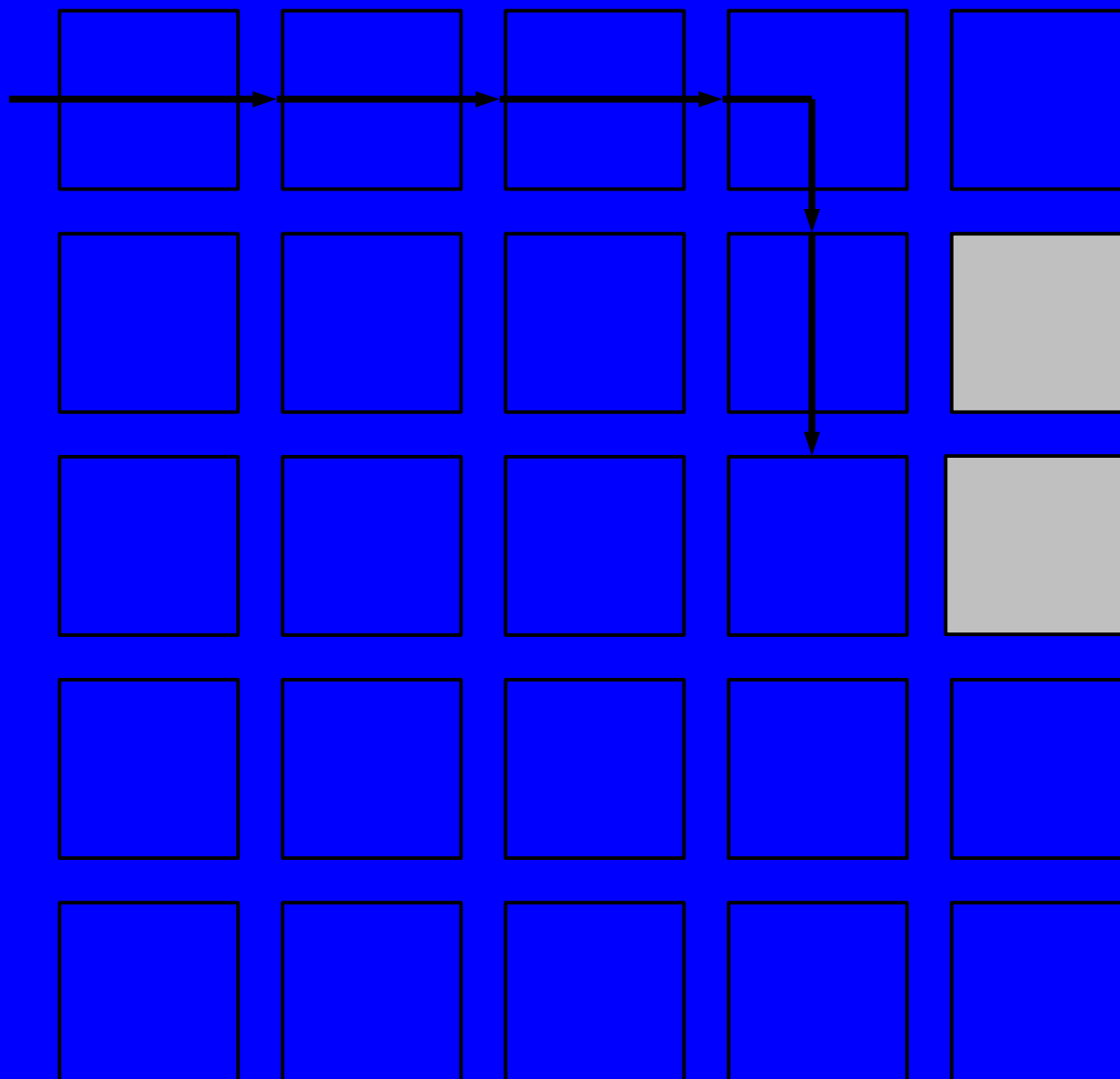
- Collections of D- and C- inputs that result in some desired set of configuration operations
- Examples:
 - building a W->E wire
 - building a corner wire from E->S
 - configuring an element to the east of a N->S wire
 - extend a N->S wire
- Supersequence: a sequence of sequences
 - Example: Bootstrapping

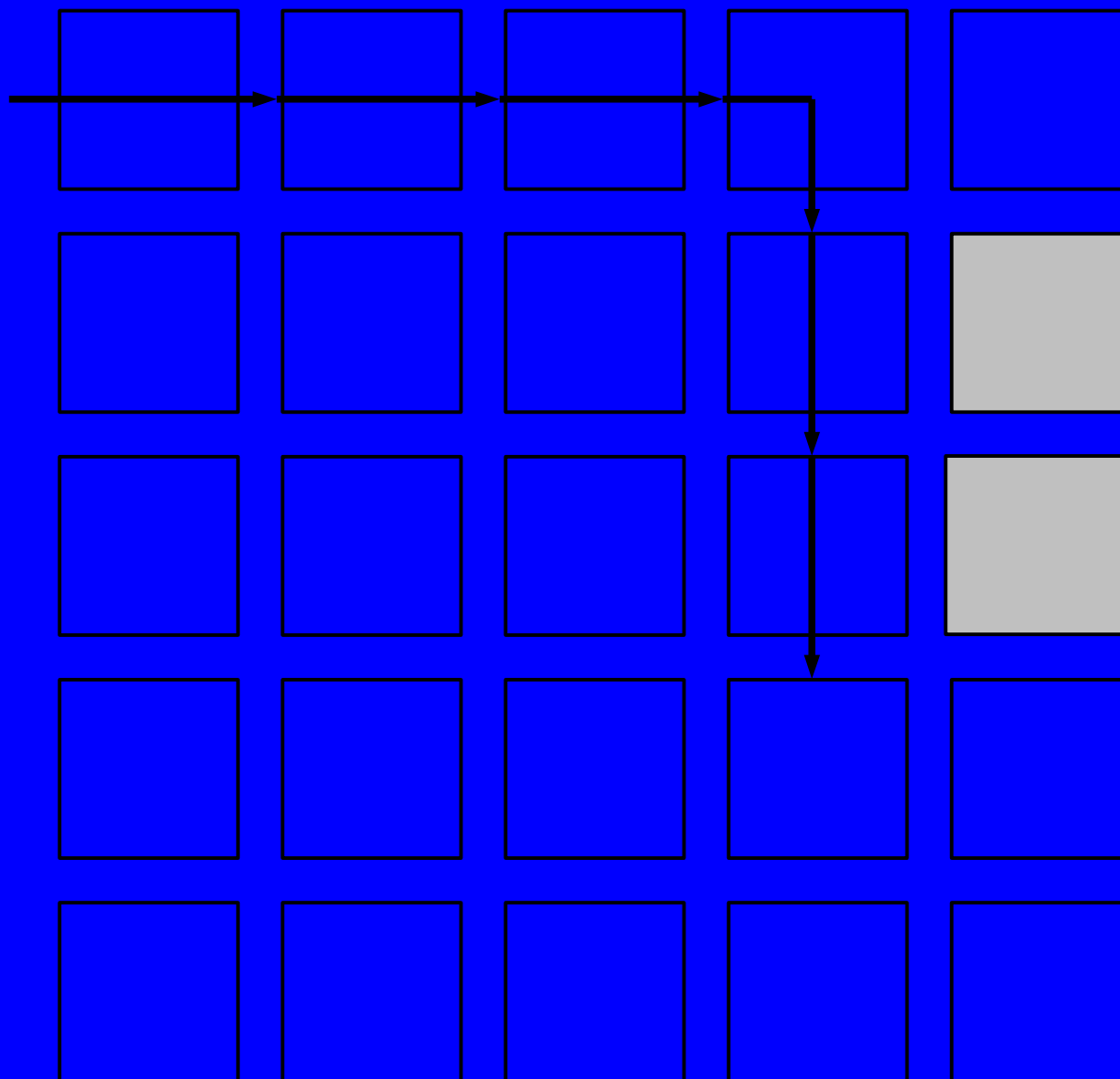


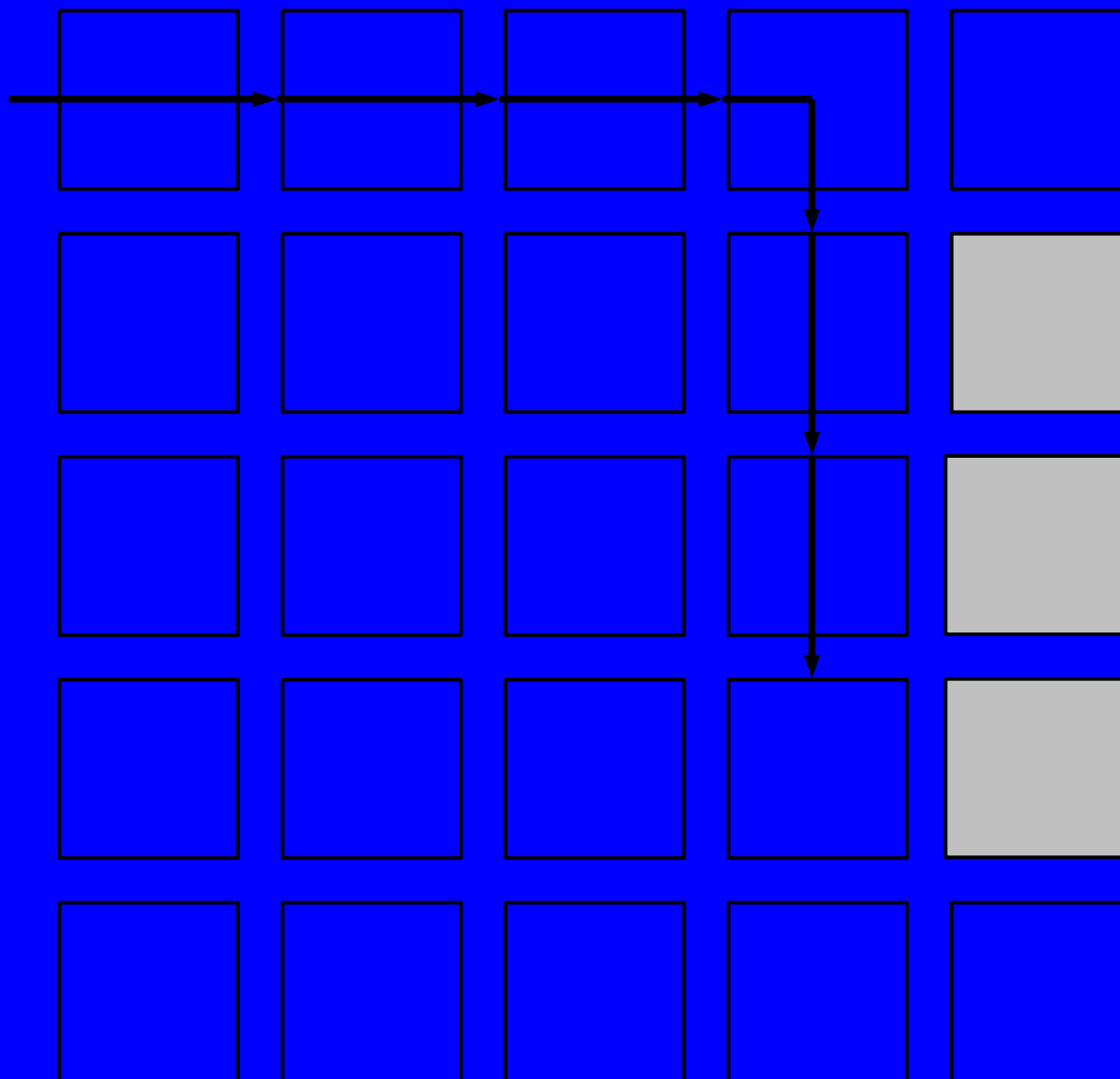
Several
Elements

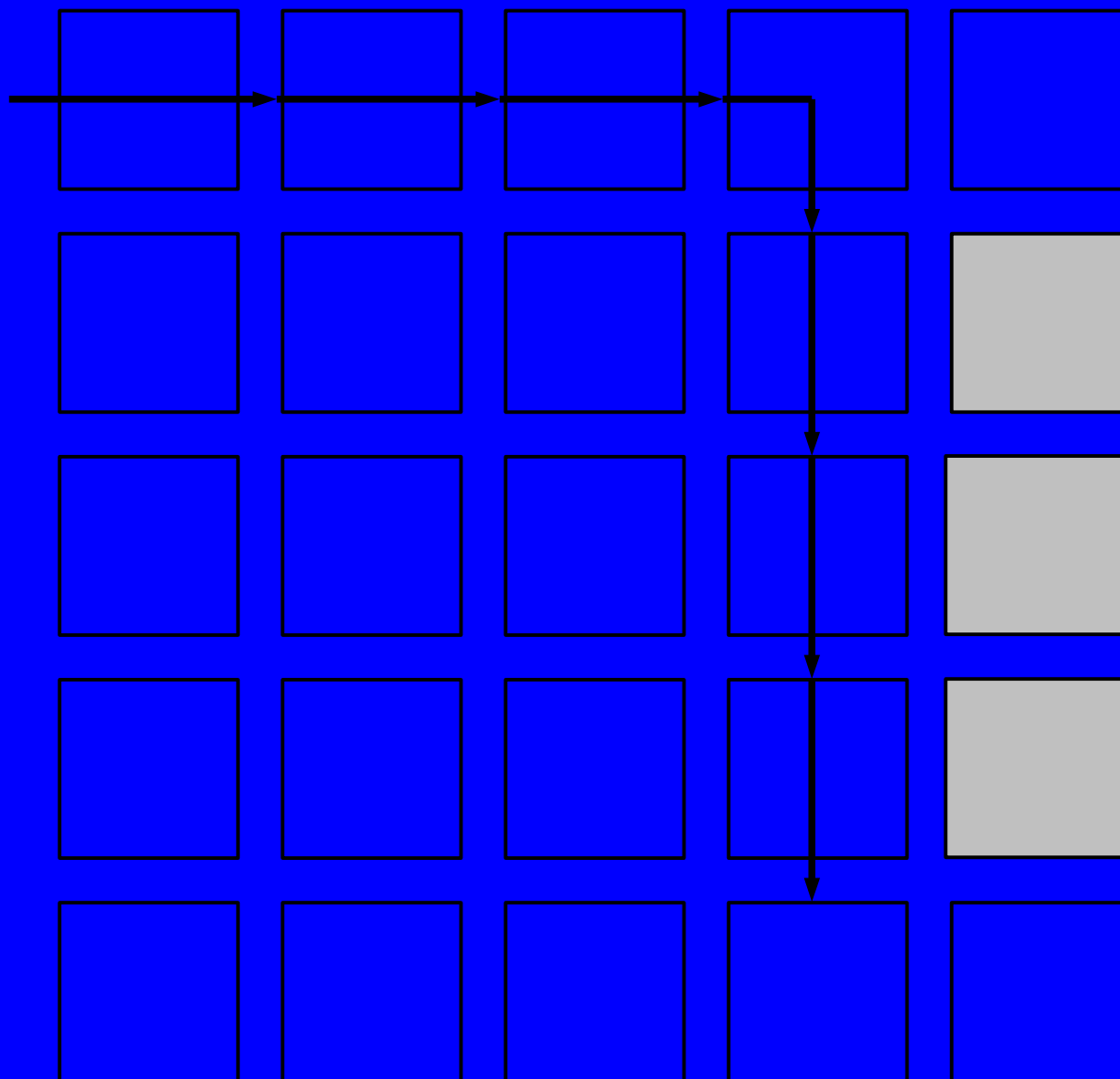


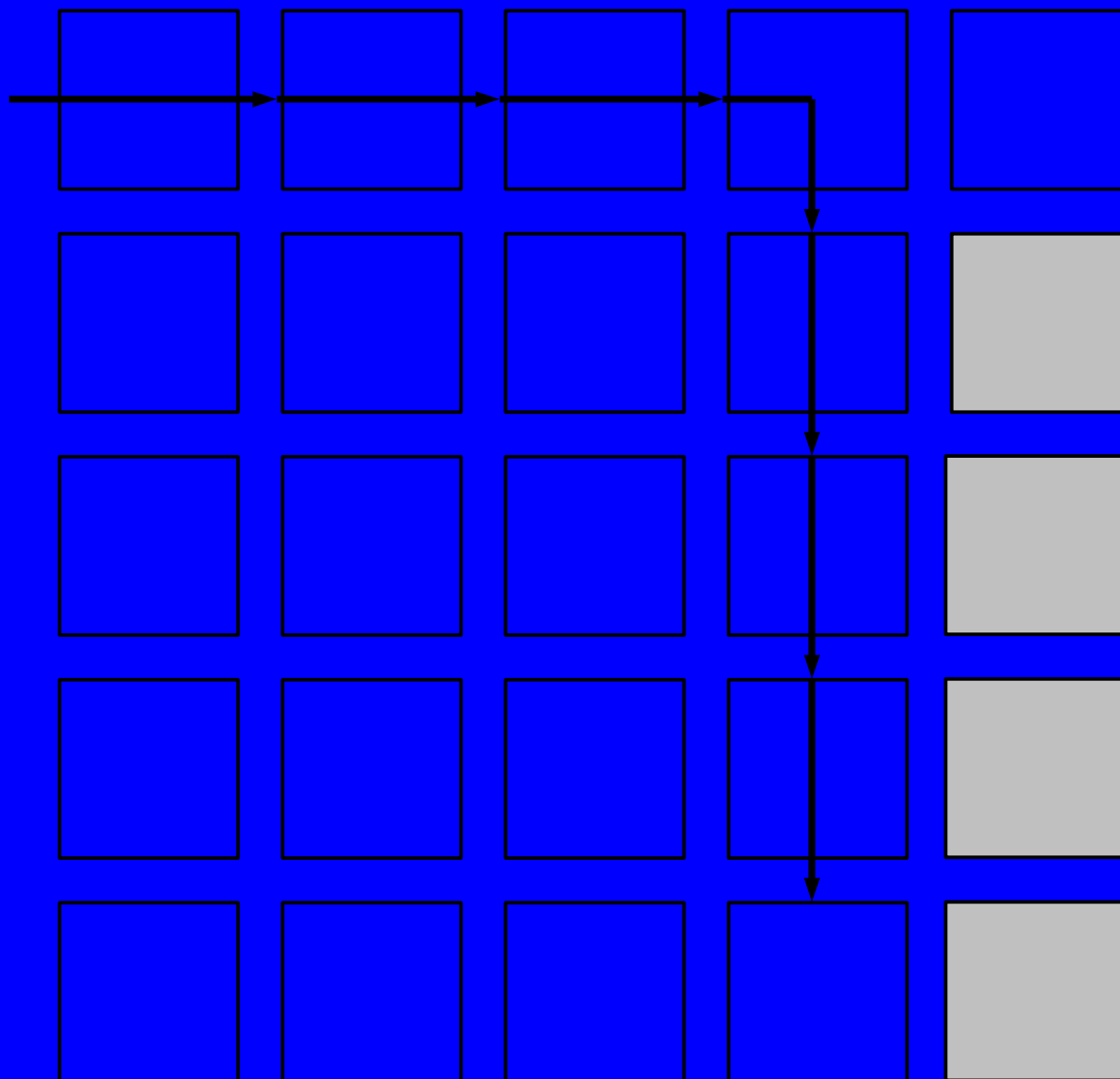






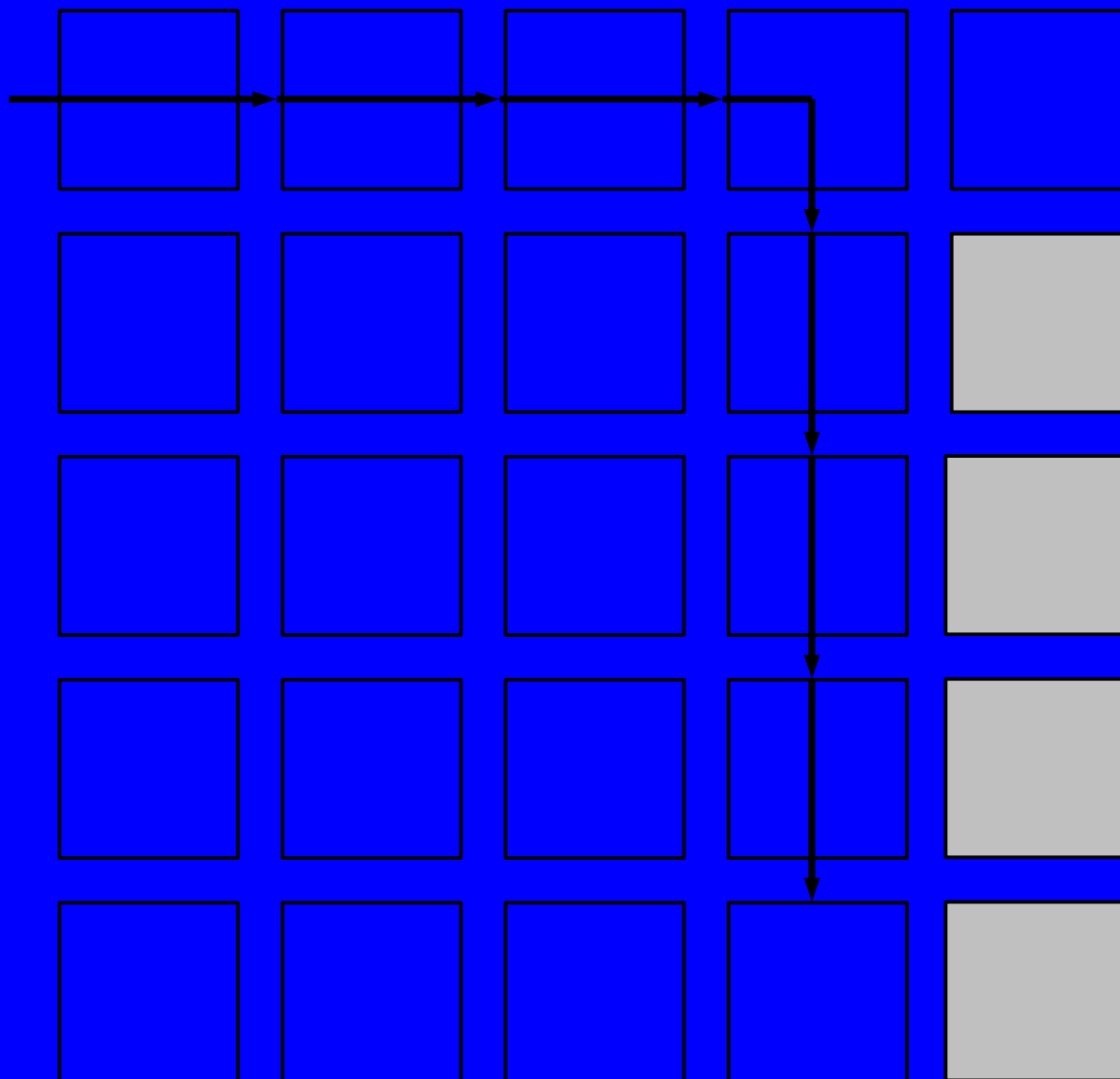


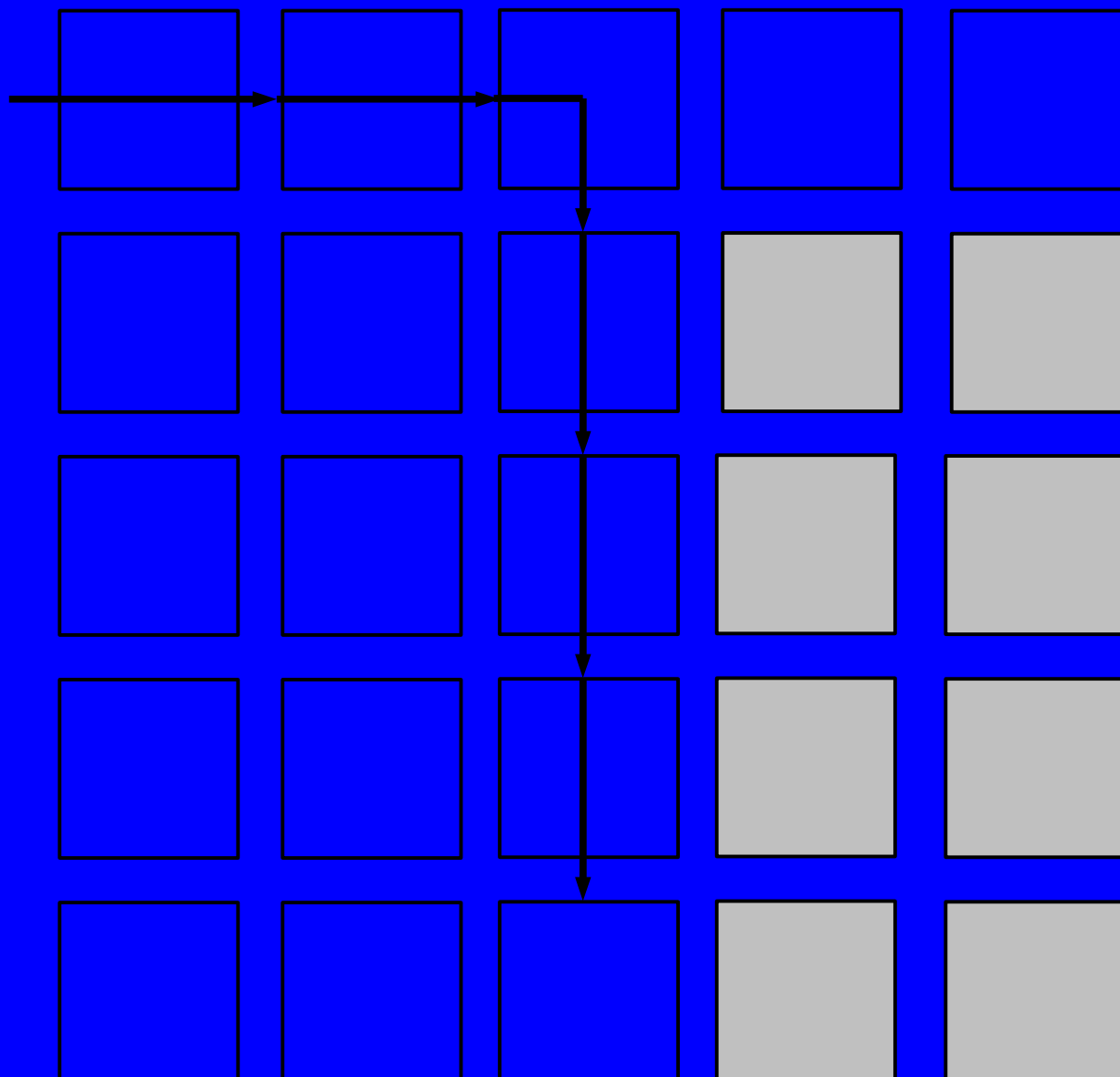


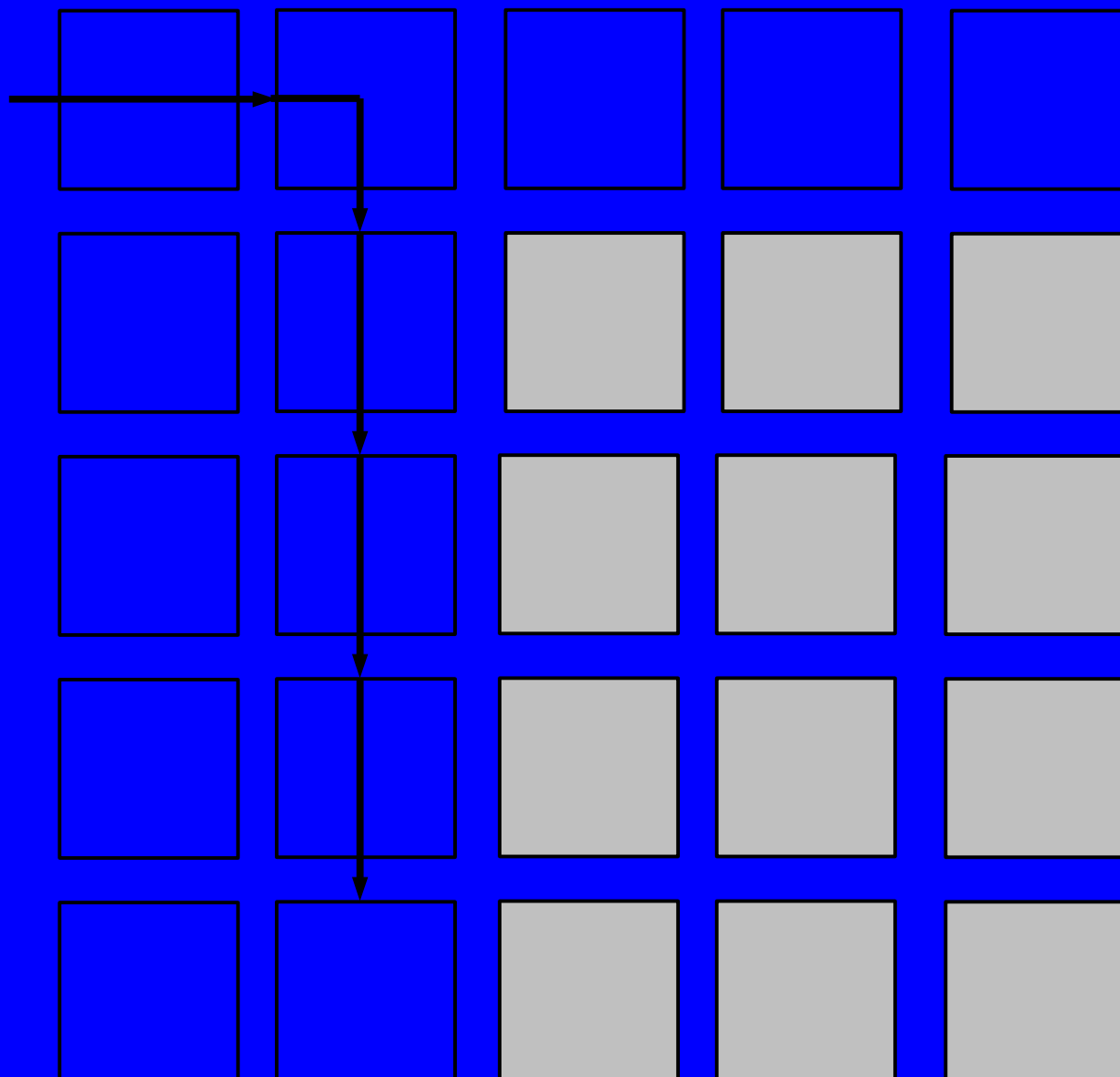


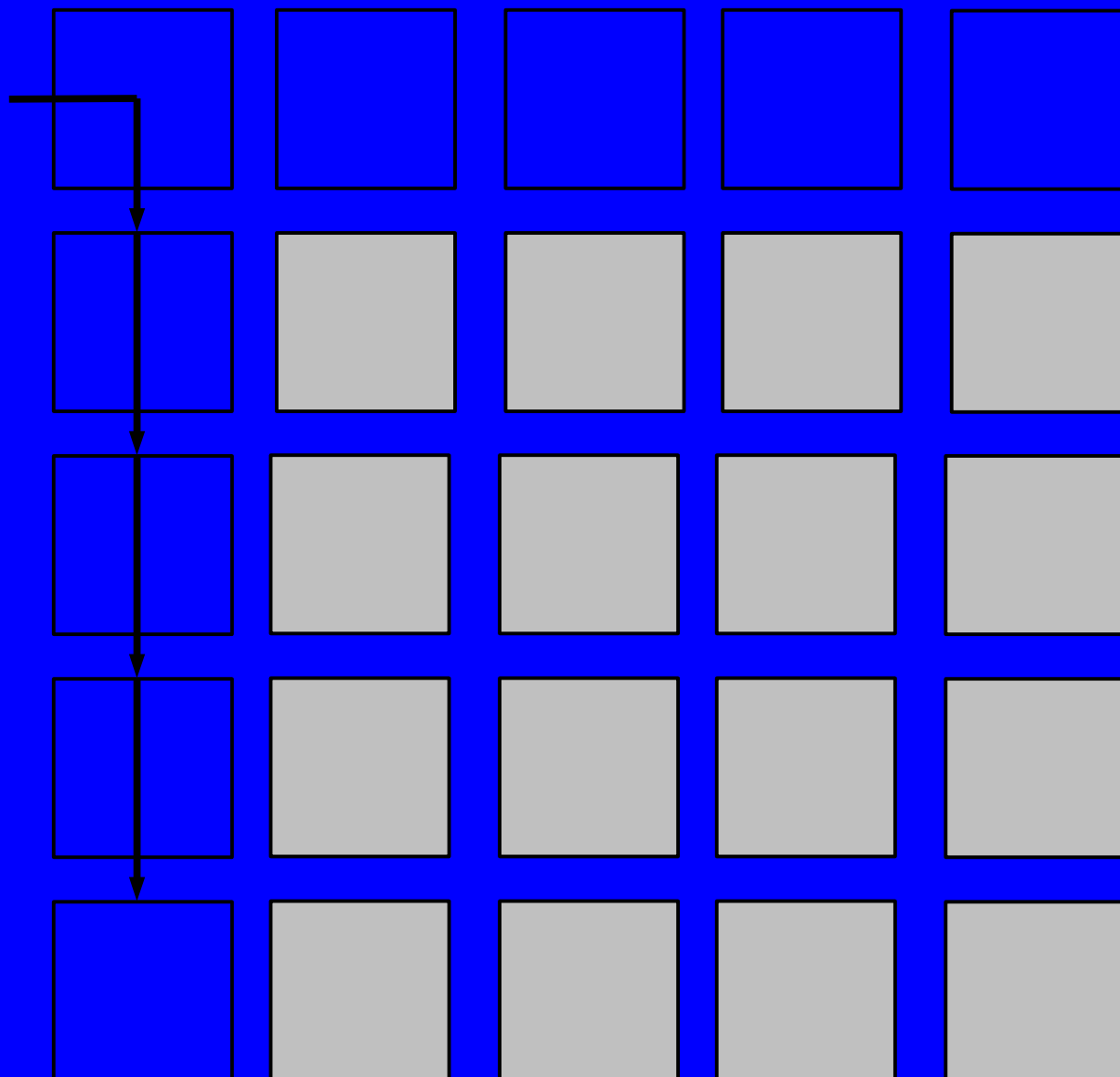
Full Set Of Sequences

- extend to east
- extend to east
- extend to east
- turn east->south
- configure to east;extend south
- configure to east;extend south
- configure to east;extend south
- configure to east









...but is this useful?

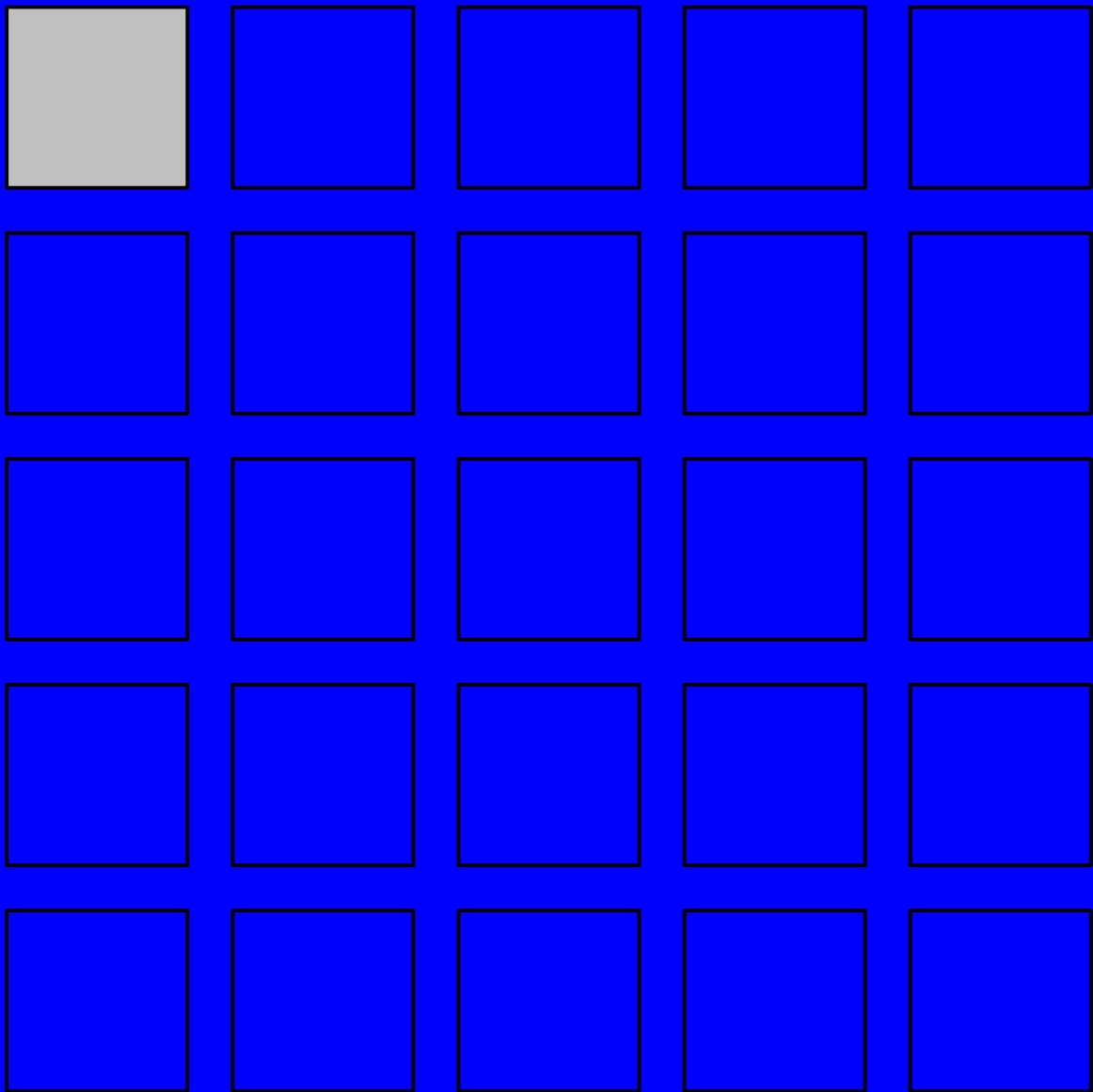
- So far, this sounds like a pain to work with
- Is there a practical, desirable application of this non-dualism?

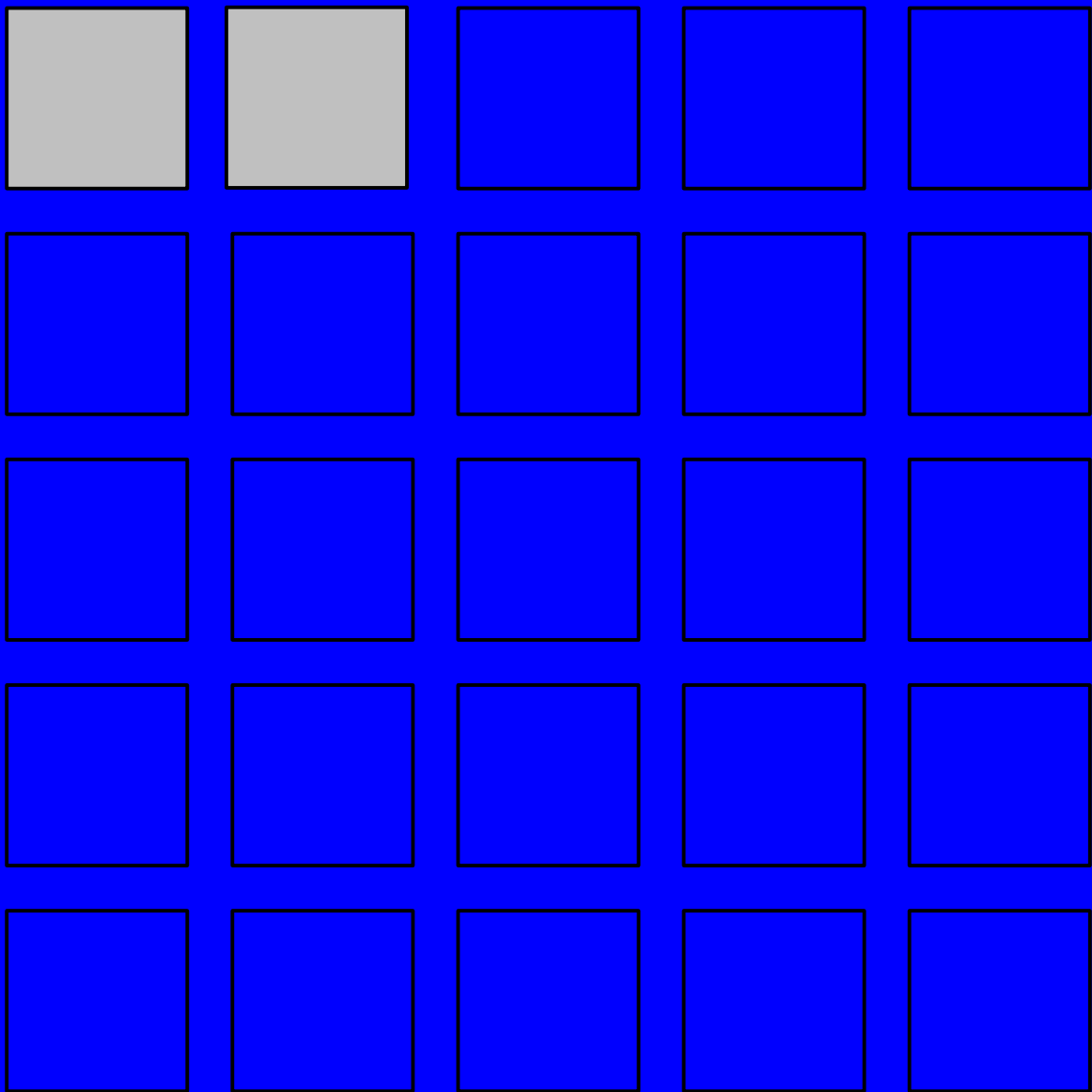
Parallel Configuration

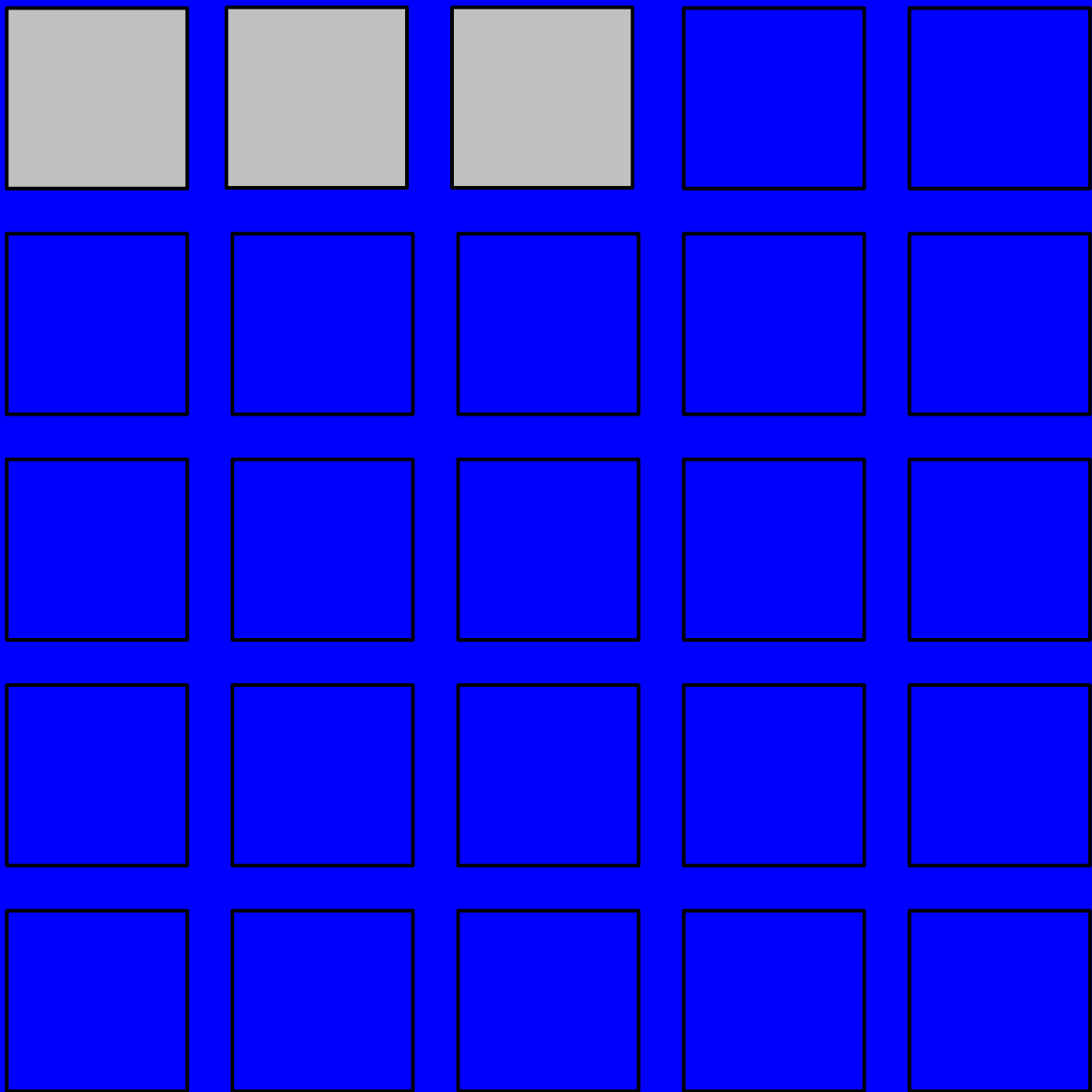
- Medusa Supersequence
- Medusa Circuit
- Useful for tiling an array with sub-circuits that are identical or very similar to each other

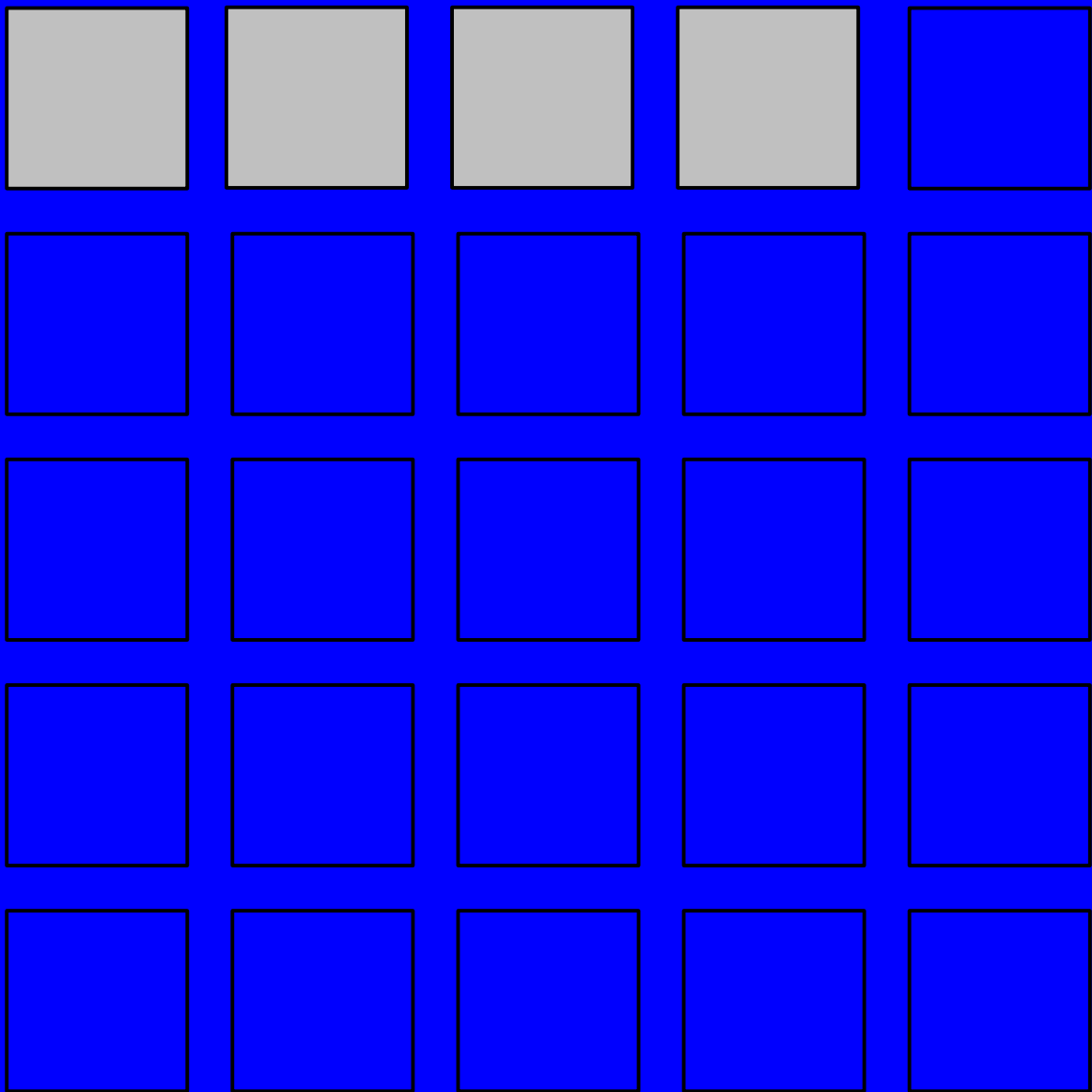
Medusa Supersequence

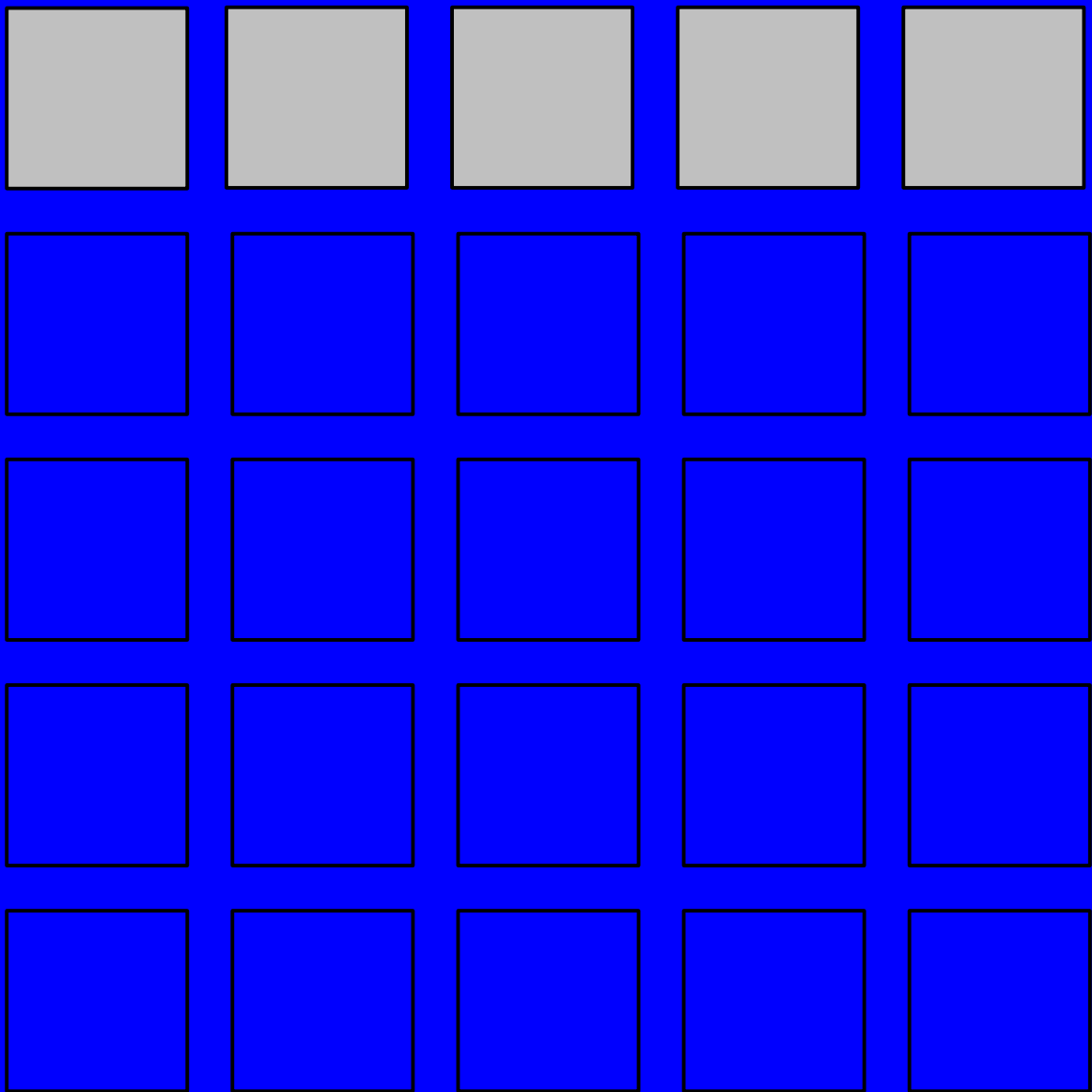
1. Extend to the east
2. If you made any progress, goto 1





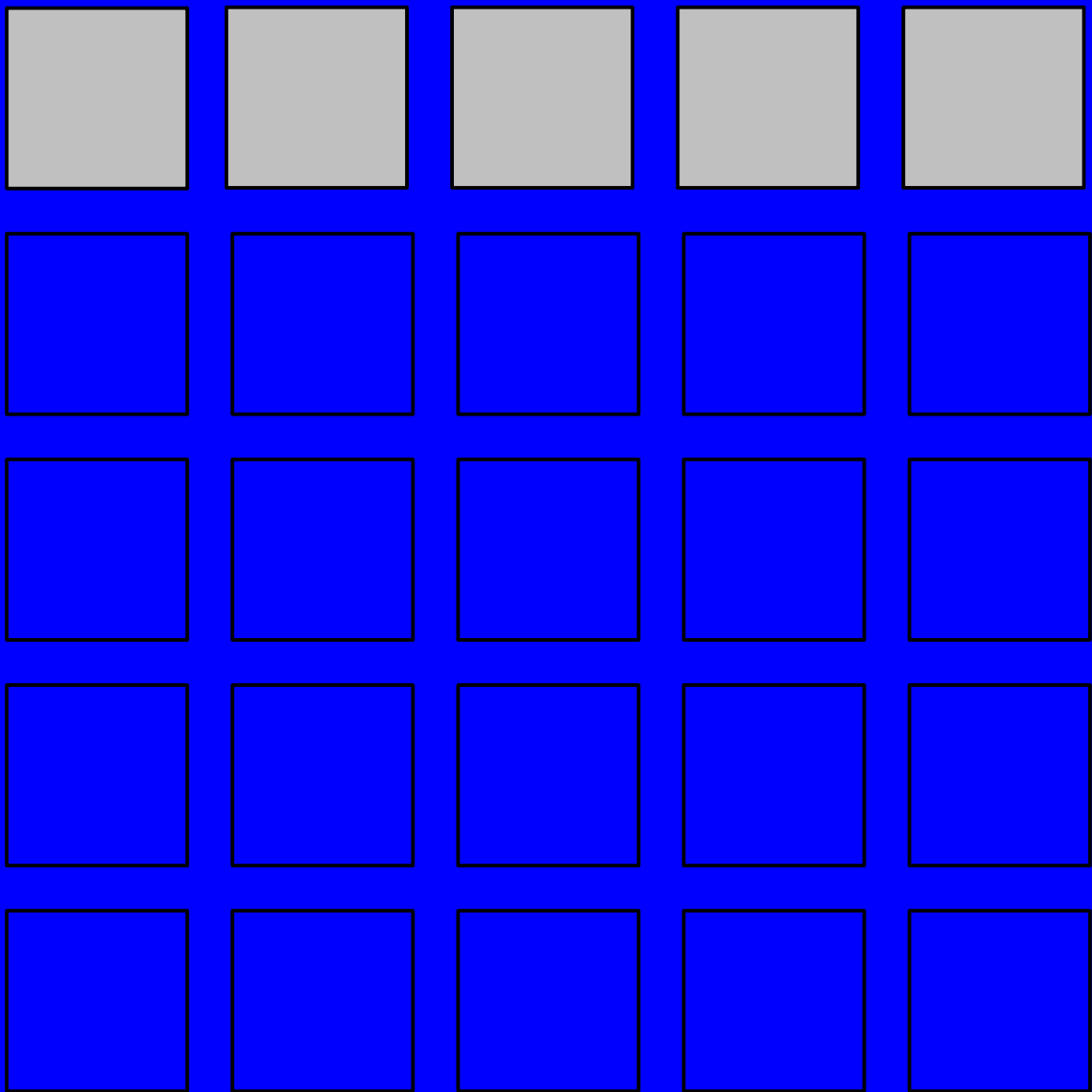


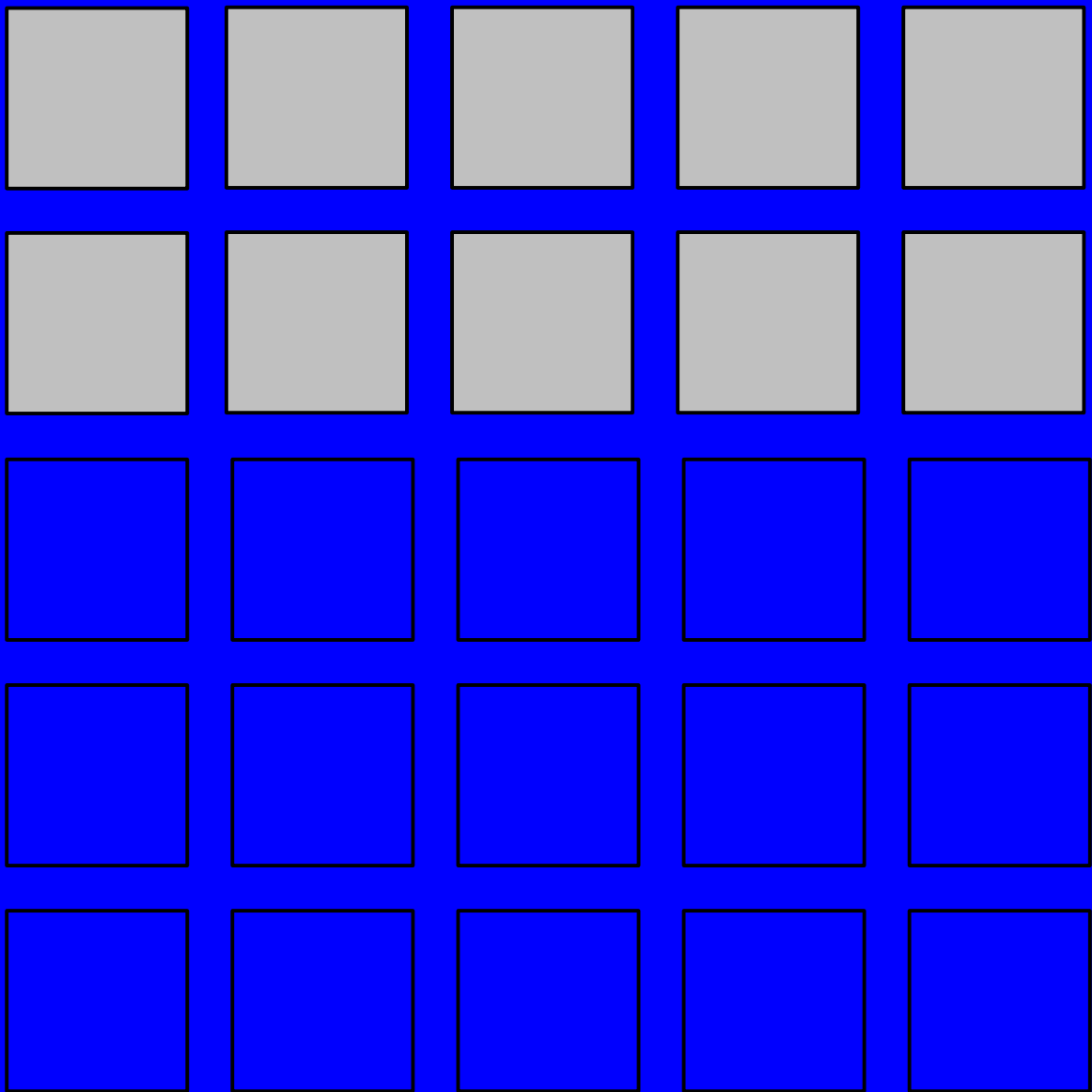


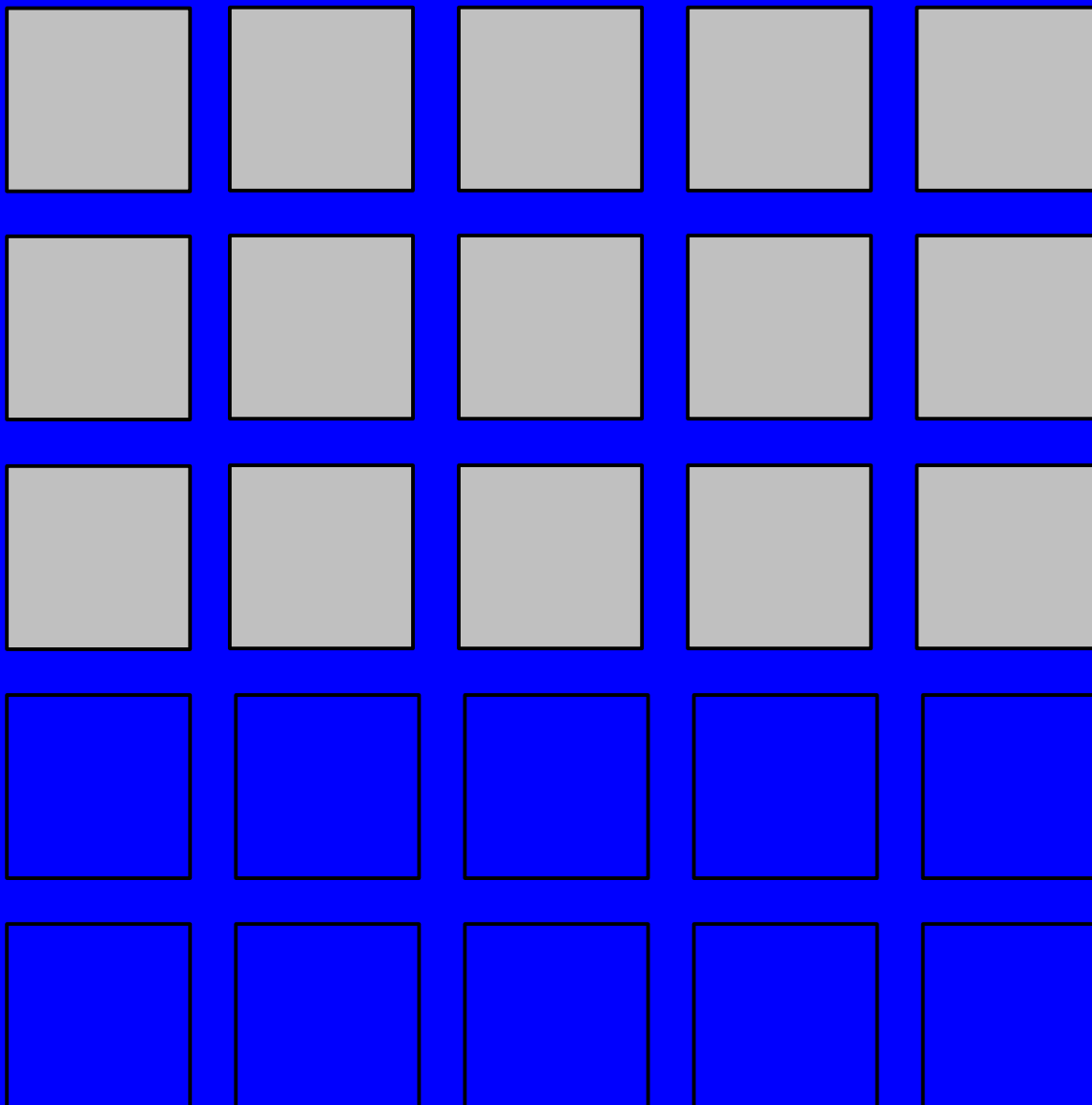


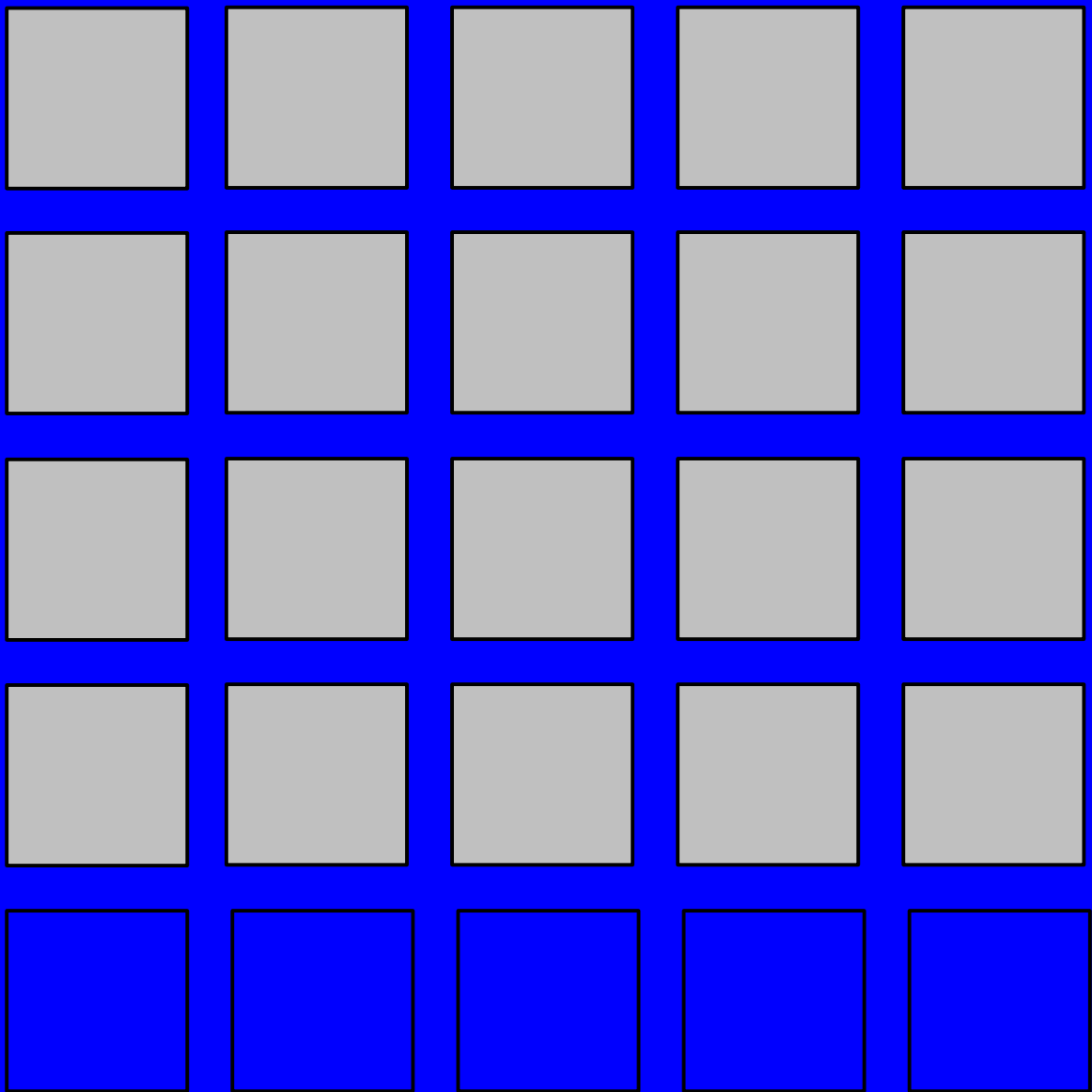
Medusa Supersequence

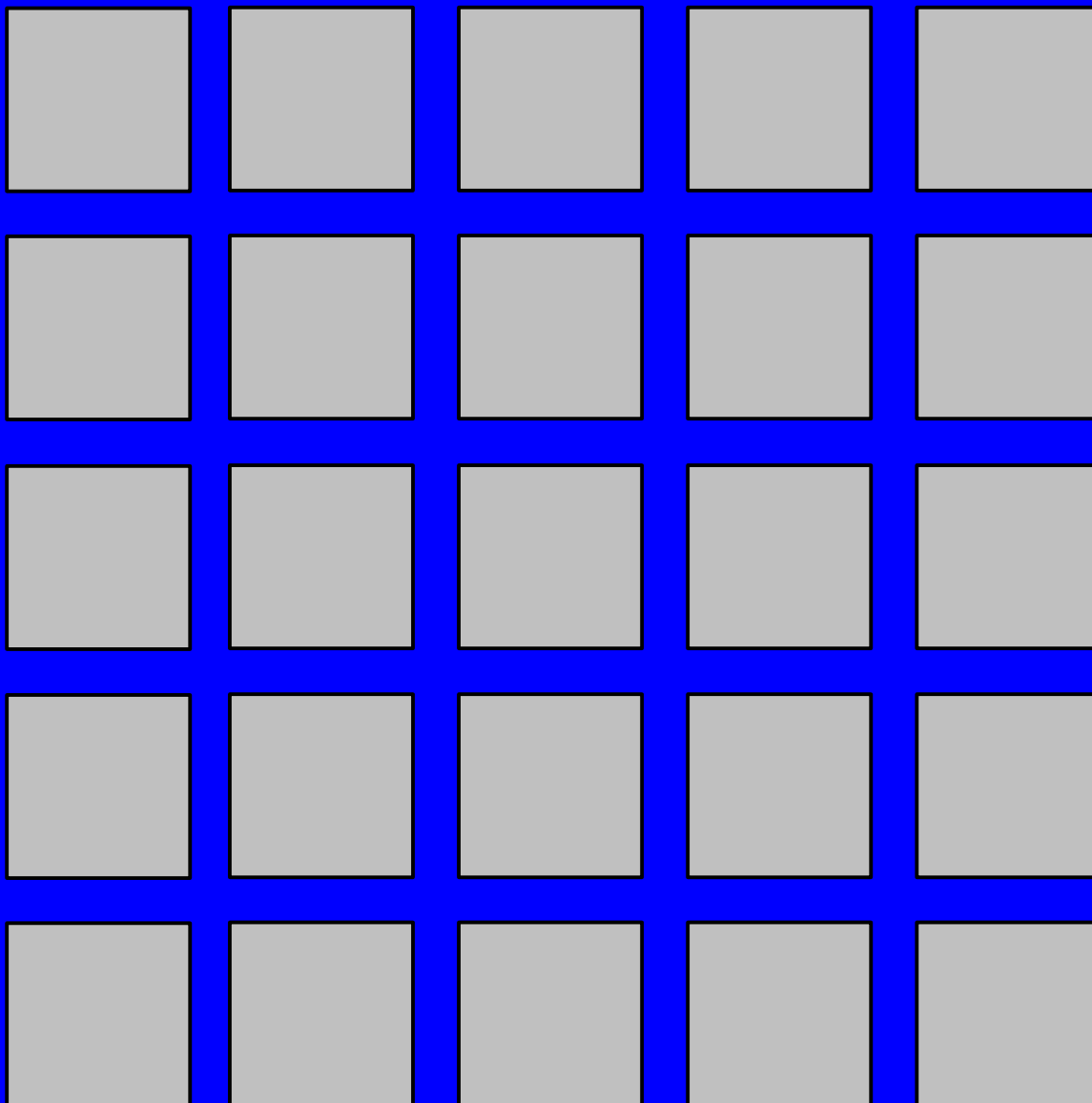
1. Extend to the east
2. If you made any progress, goto 1
3. Extend to the south *in parallel*
4. If you made any progress, goto 3





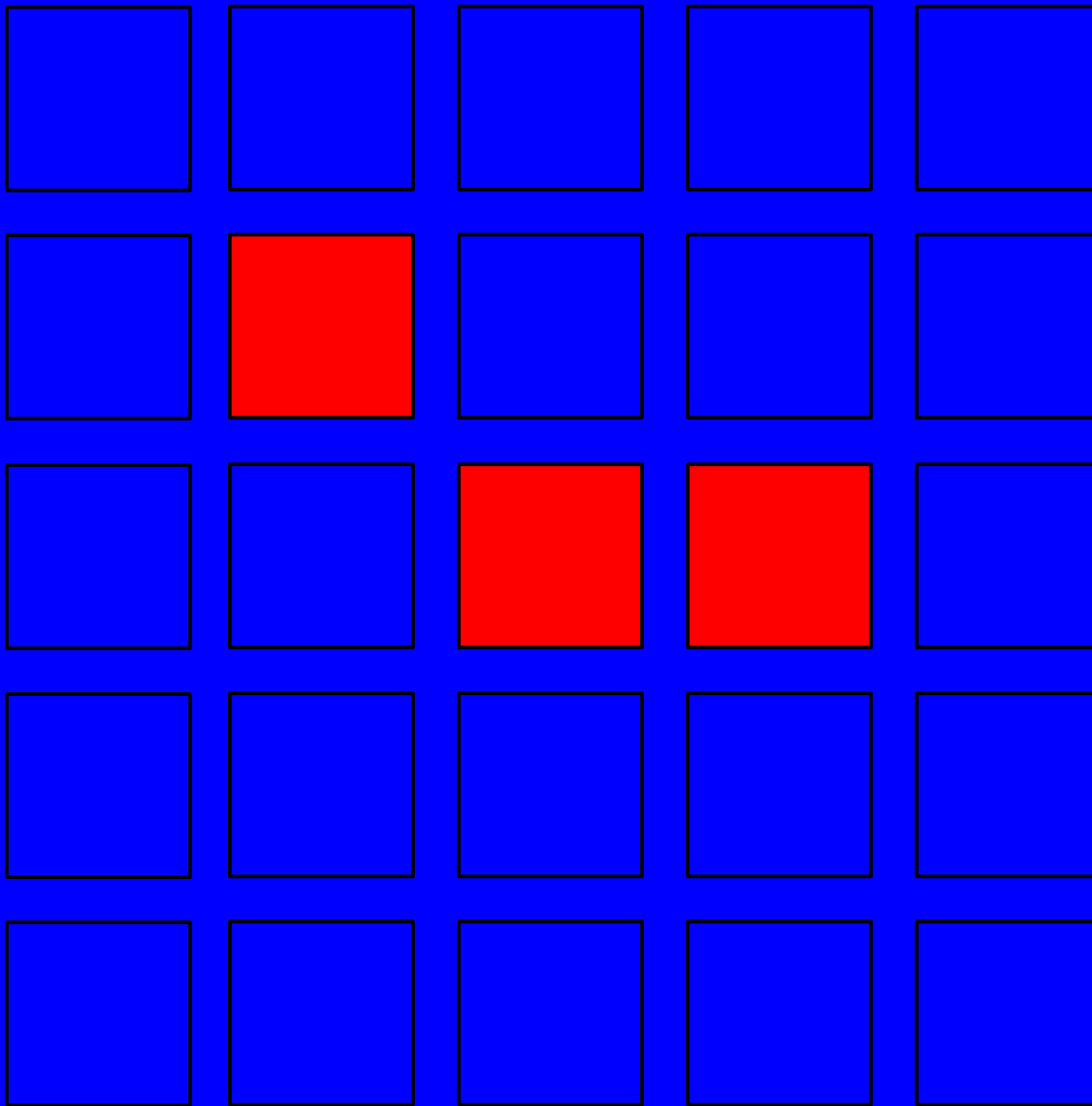






This can be a very efficient way to configure a large array of elements

- $O(n^2)$ elements in $O(n)$ steps
- Can extend to 3-D...final 2-D sheet configures a second 2-D sheet in one step
- Can adjust granularity ($K \times L$ sub-regions)
- General parallelizing scheme:
e.g. can also do testing/analysis in parallel



Medusa Supersequence for Faulty Arrays

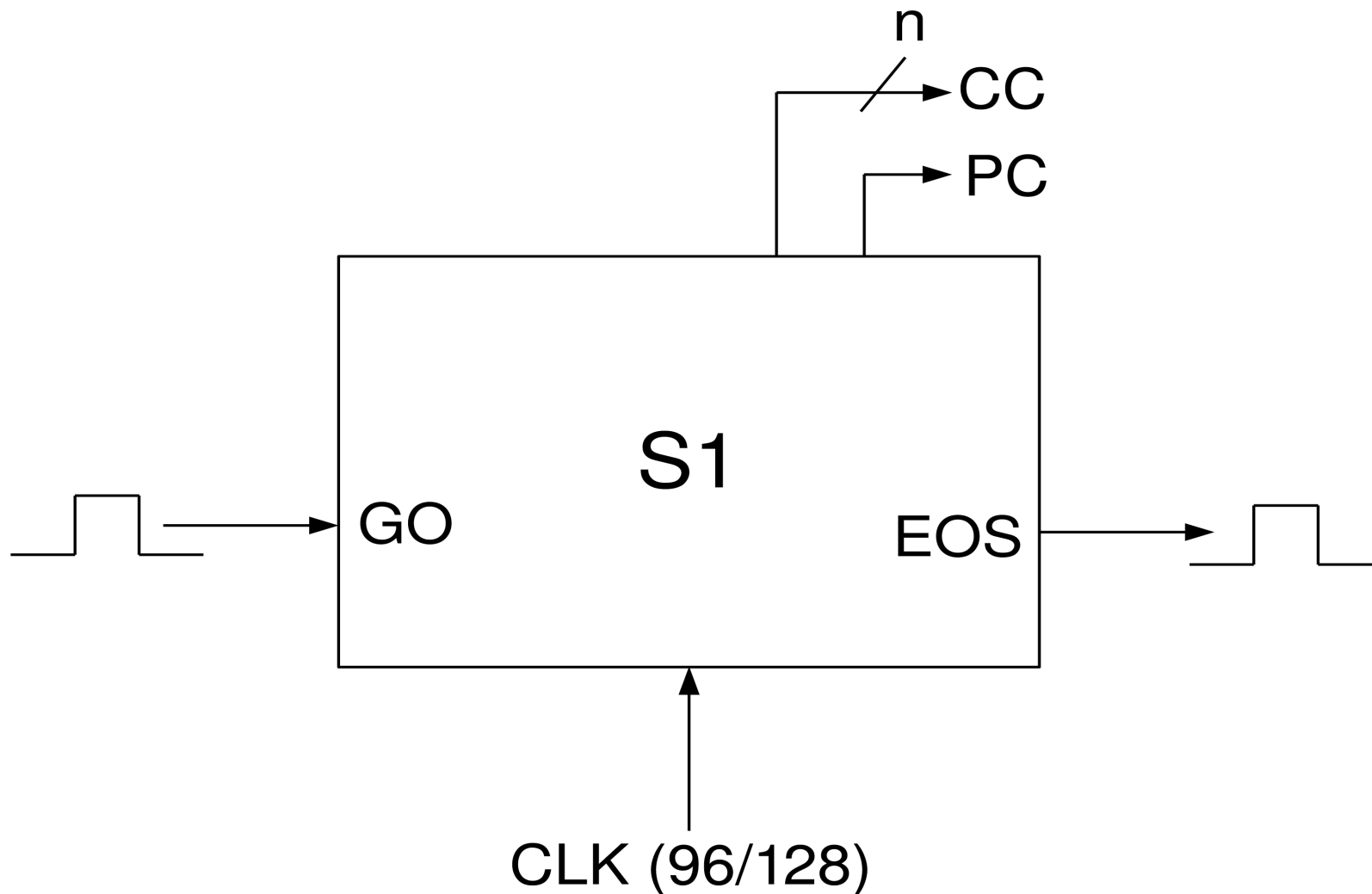
1. Extend to the east
2. If you made any progress, goto 1
3. Extend to the south
4. If you made any progress, **goto 1**

In-vivo supersequence generation

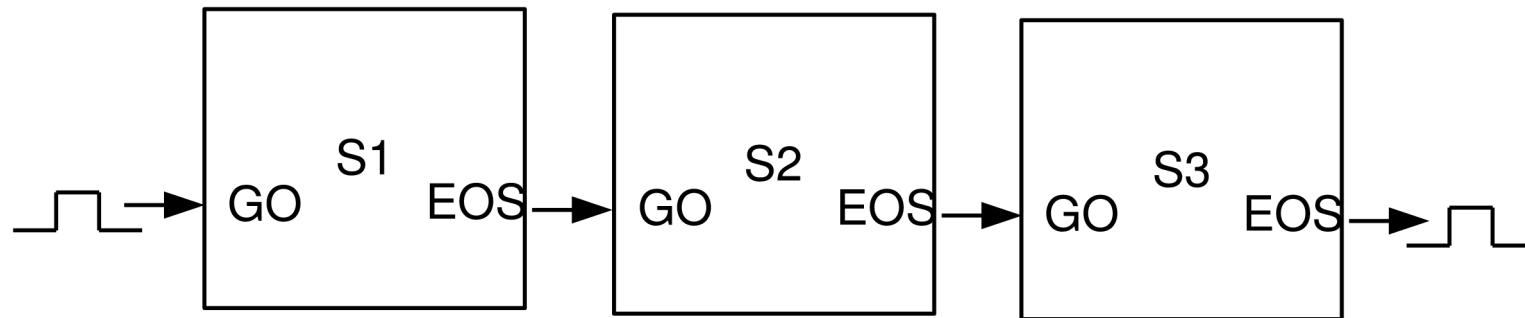
Necessary operations include:

- generation of a sequence (C- and D-bits)
 - looping a fixed number of times
 - conditional branch to different supersequence
- plus combinations of the above...

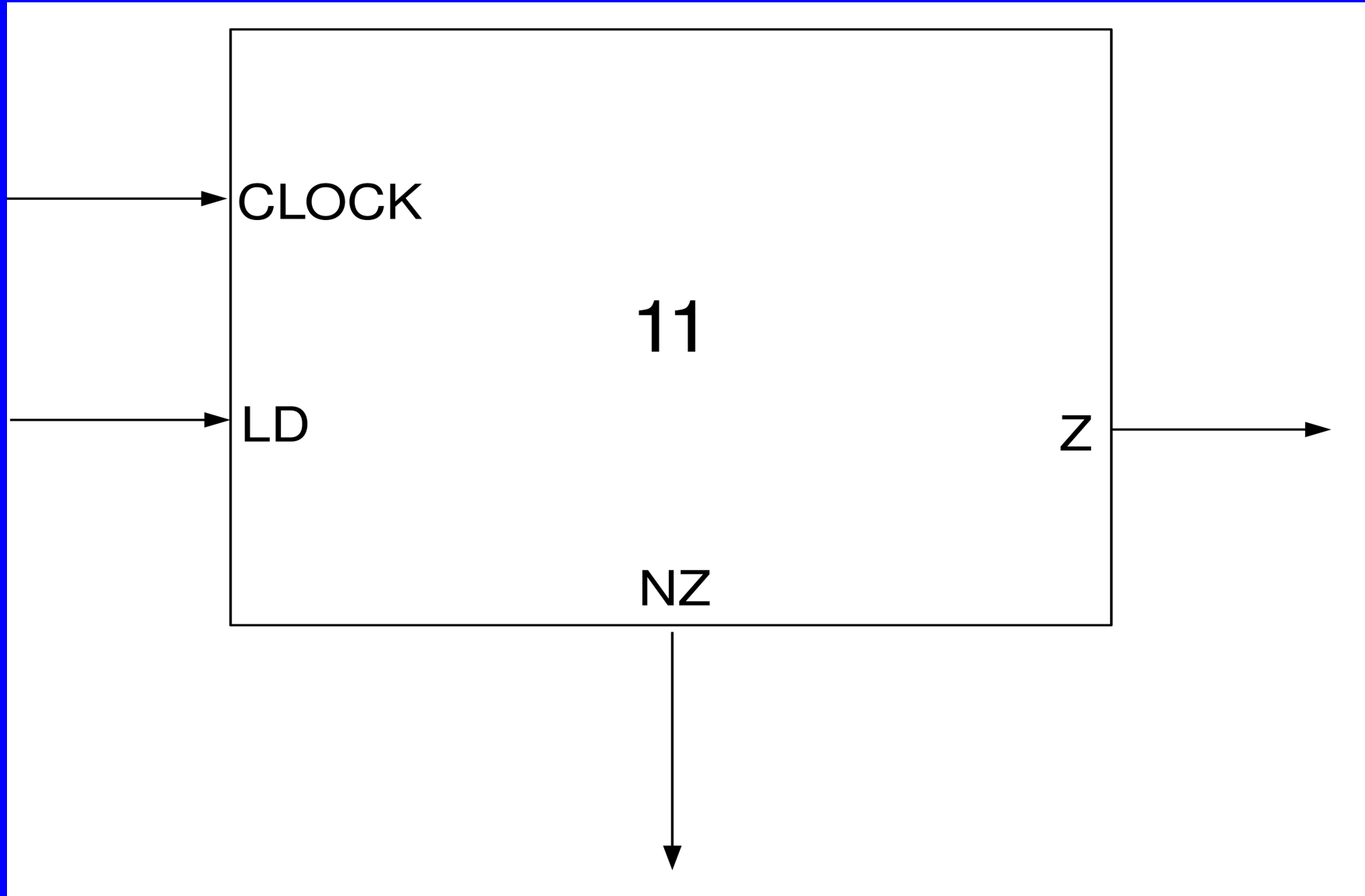
Simple Sequence Generator



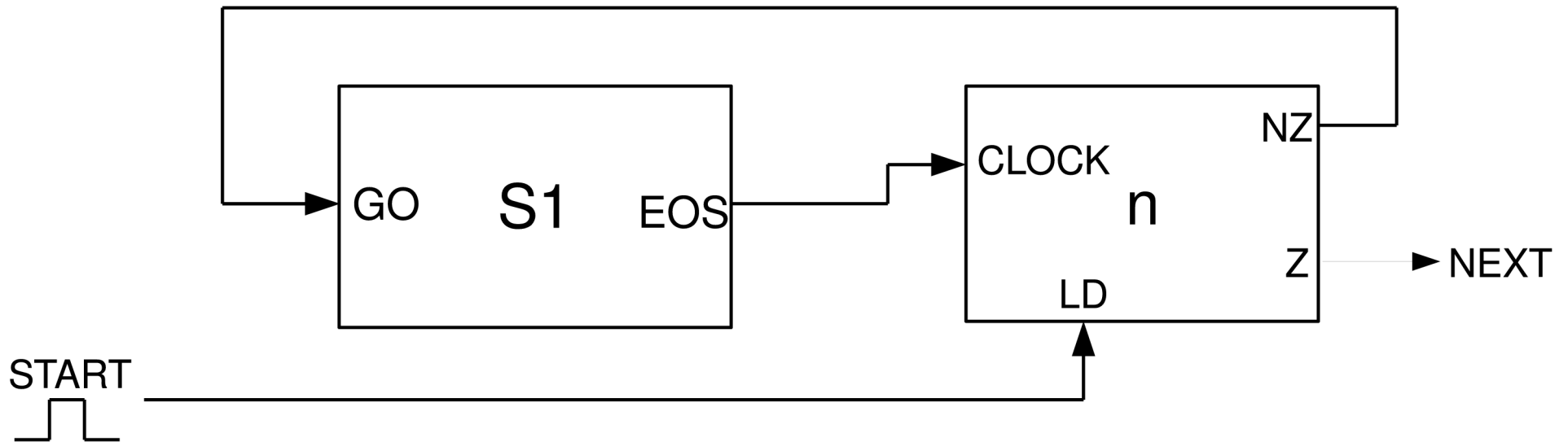
Simple Supersequence



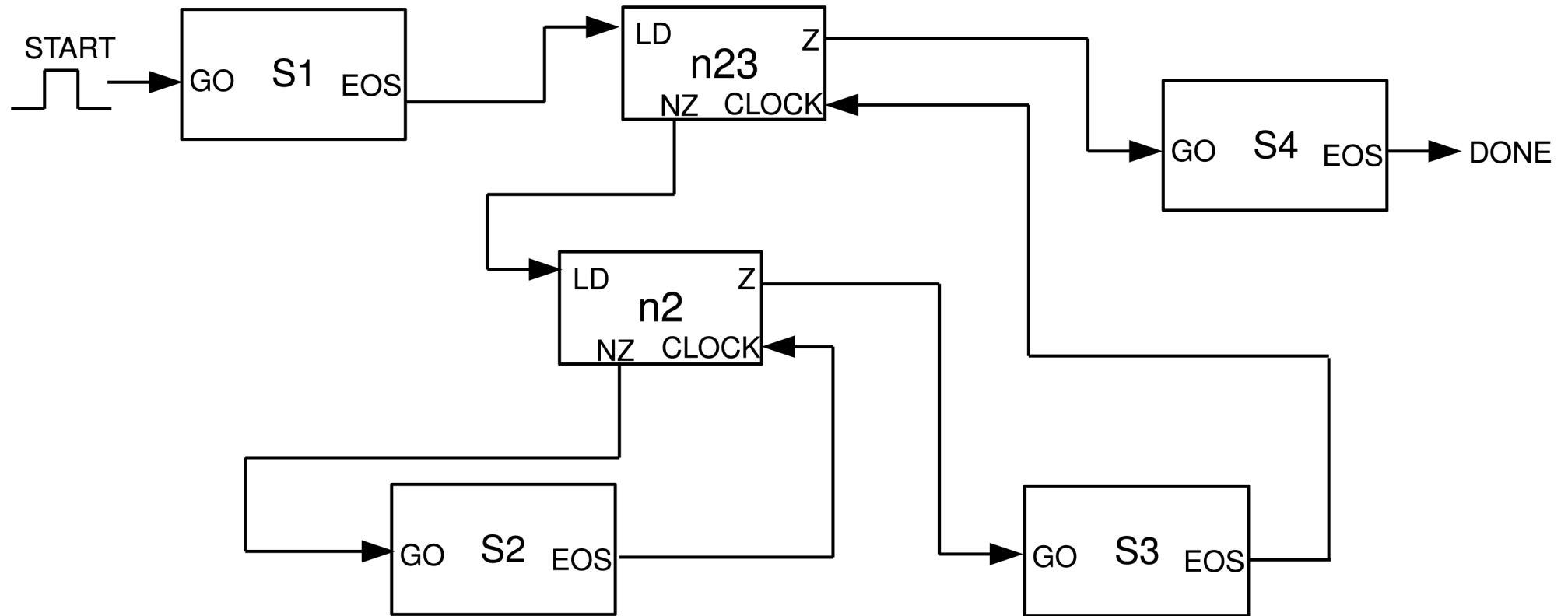
Sequence Counter

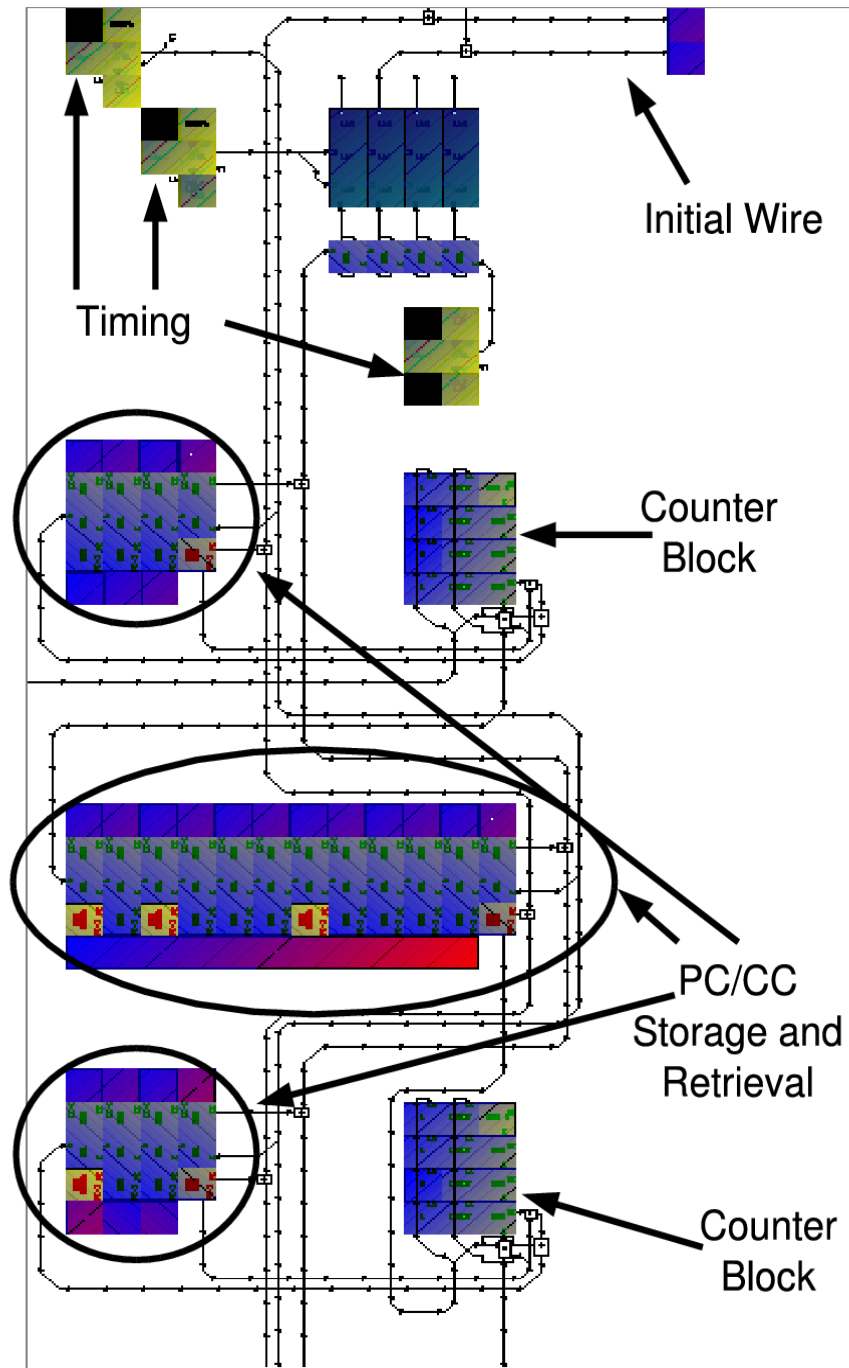


Repeated Sequence



Complex Supersequence: $S1 + ((S2 * n2) + S3) * n23 + S4$



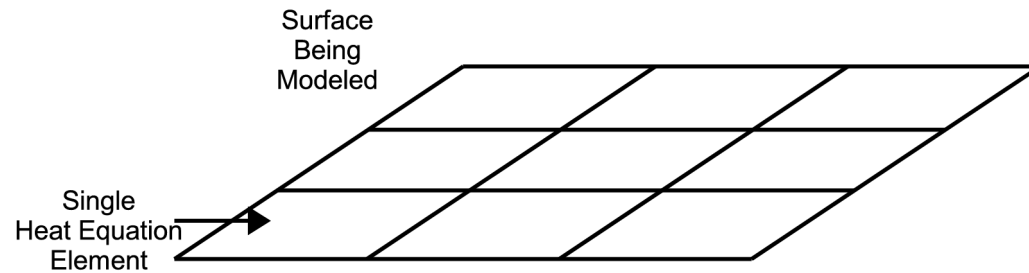


Sample Application: Heat Flow Simulation

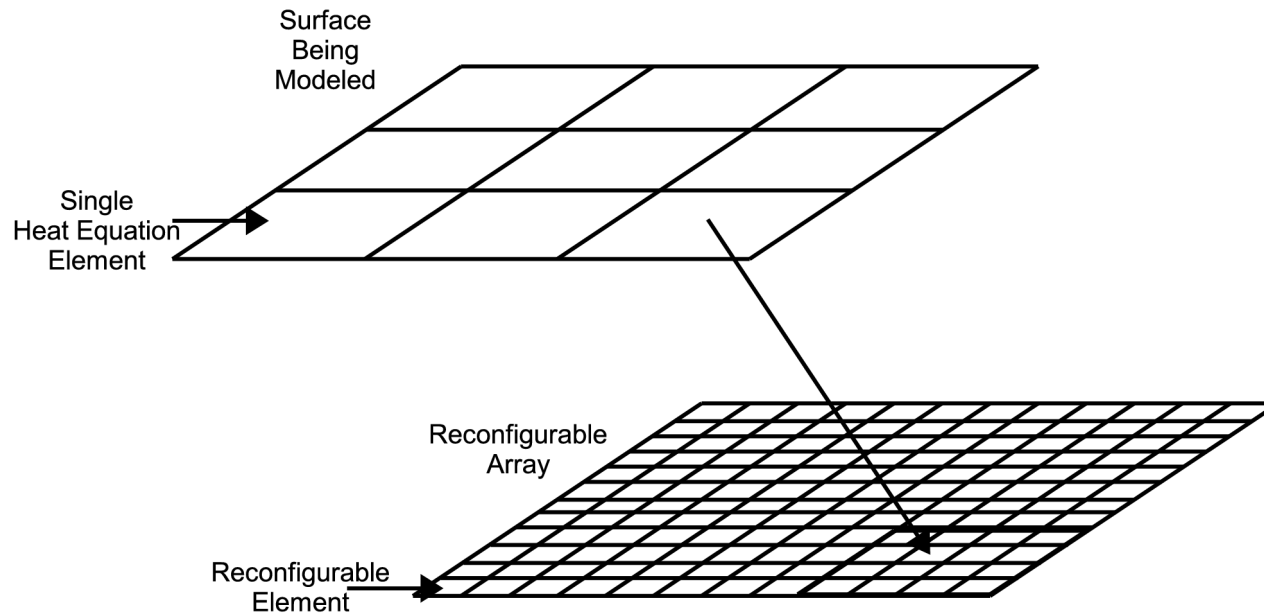
$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right)$$

$u(x,y,z)$ =temperature

Division of Surface into Elements



Mapping surface to reconfigurable array



Discretization

$$\frac{\partial u(x, y, z, t)}{\partial x} \approx \frac{u(x + \Delta x, y, z, t) - u(x, y, z, t)}{\Delta x}$$

$$\frac{\partial^2 u(x, y, z, t)}{\partial x^2} \approx \frac{\frac{\partial u(x, y, z, t)}{\partial x} - \frac{\partial u(x - \Delta x, y, z, t)}{\partial x}}{\Delta x}$$

$$\frac{\partial^2 u(x, y, z, t)}{\partial x^2} \approx \frac{u(x + \Delta x, y, z, t) + u(x - \Delta x, y, z, t) - 2u(x, y, z, t)}{\Delta x^2}$$

Corresponding (2-D) Equations in terms of [c,r]

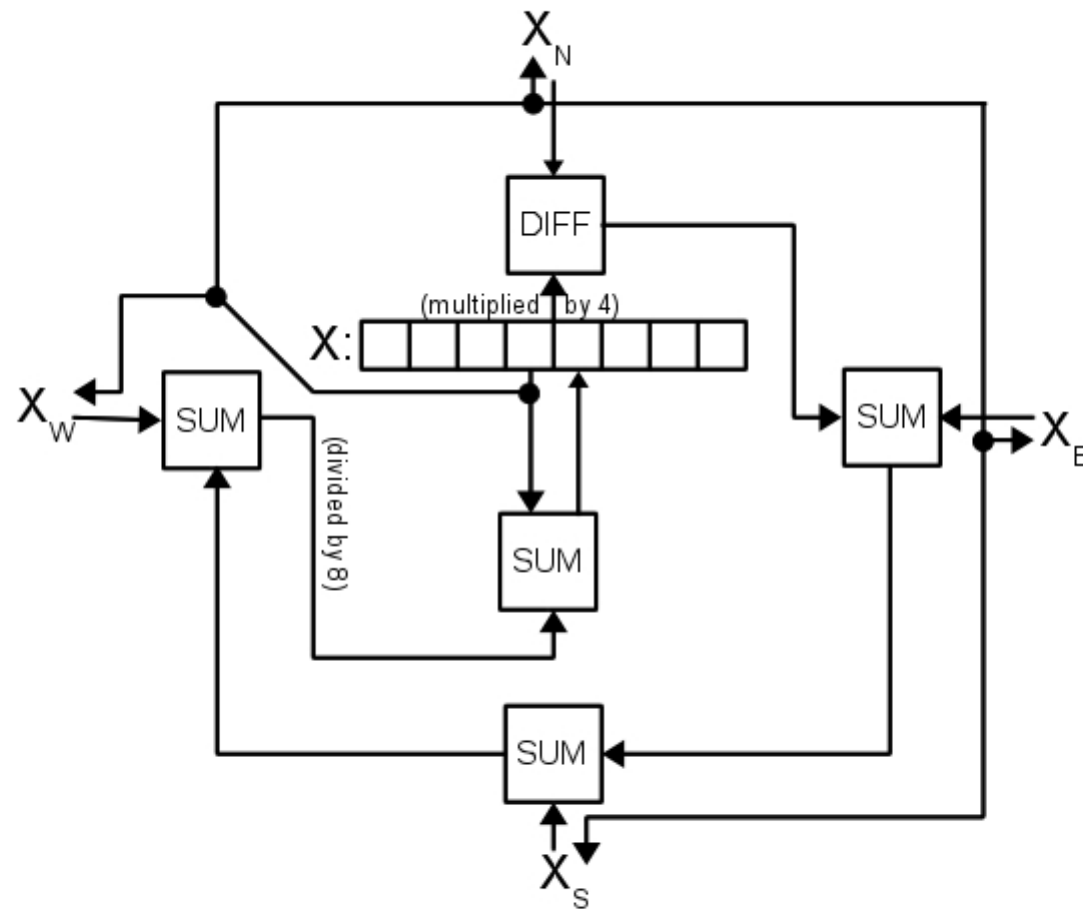
$$\frac{\Delta u}{\Delta t} = \alpha \left(\frac{u(c+1, r, t) + u(c-1, r, t) - 2u(c, r, t)}{\Delta x^2} + \frac{u(c, r+1, t) + u(c, r-1, t) - 2u(c, r, t)}{\Delta y^2} \right)$$

$$\Delta u = \alpha \Delta t (u(c+1, r, t) + u(c-1, r, t) + u(c, r+1, t) + u(c, r-1, t) - 4u(c, r, t))$$

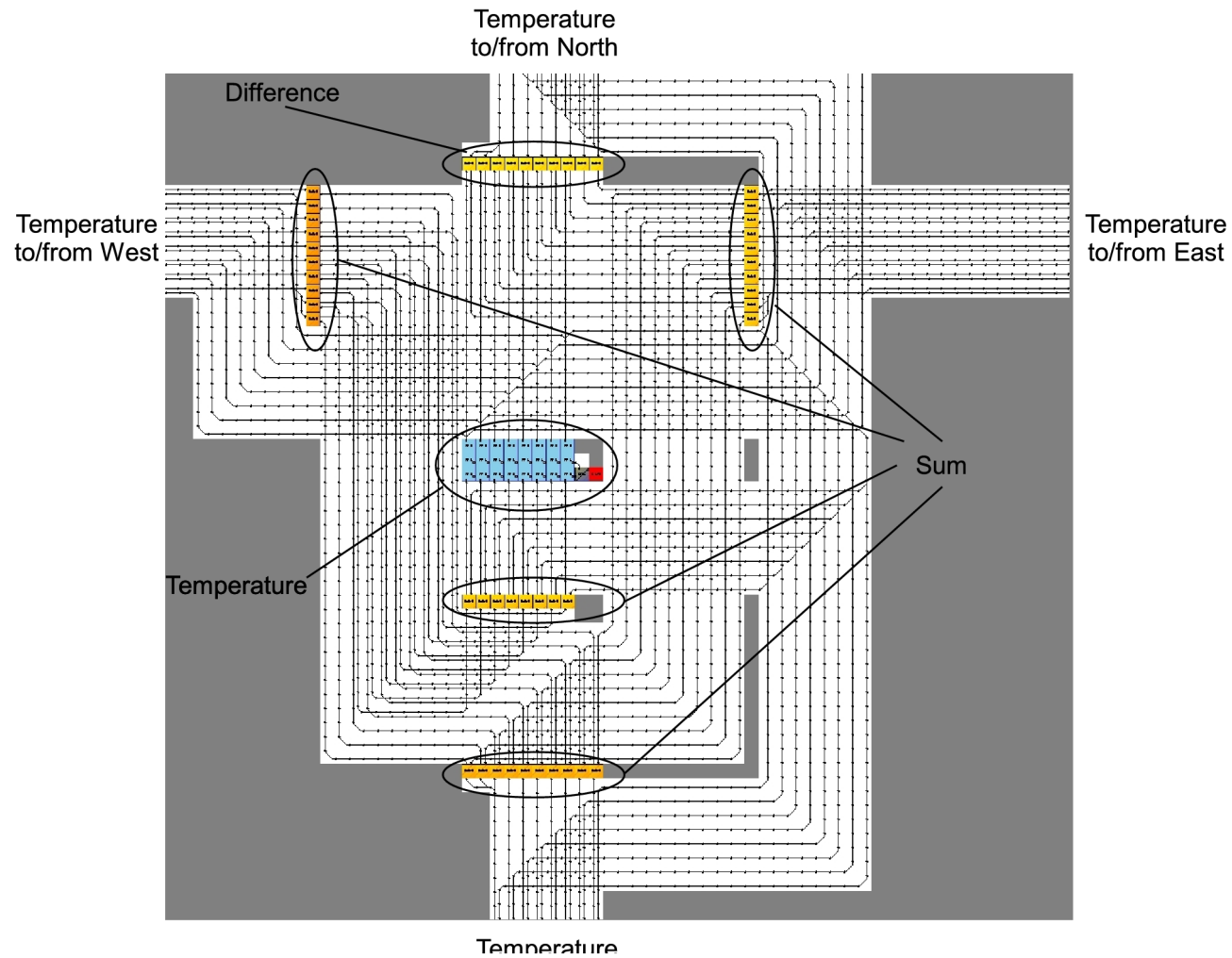
$$u(c, r, t+1) = u(c, r, t) + \alpha (u(c+1, r, t) + u(c-1, r, t) + u(c, r+1, t) + u(c, r-1, t) - 4u(c, r, t))$$

Heat Equation Element

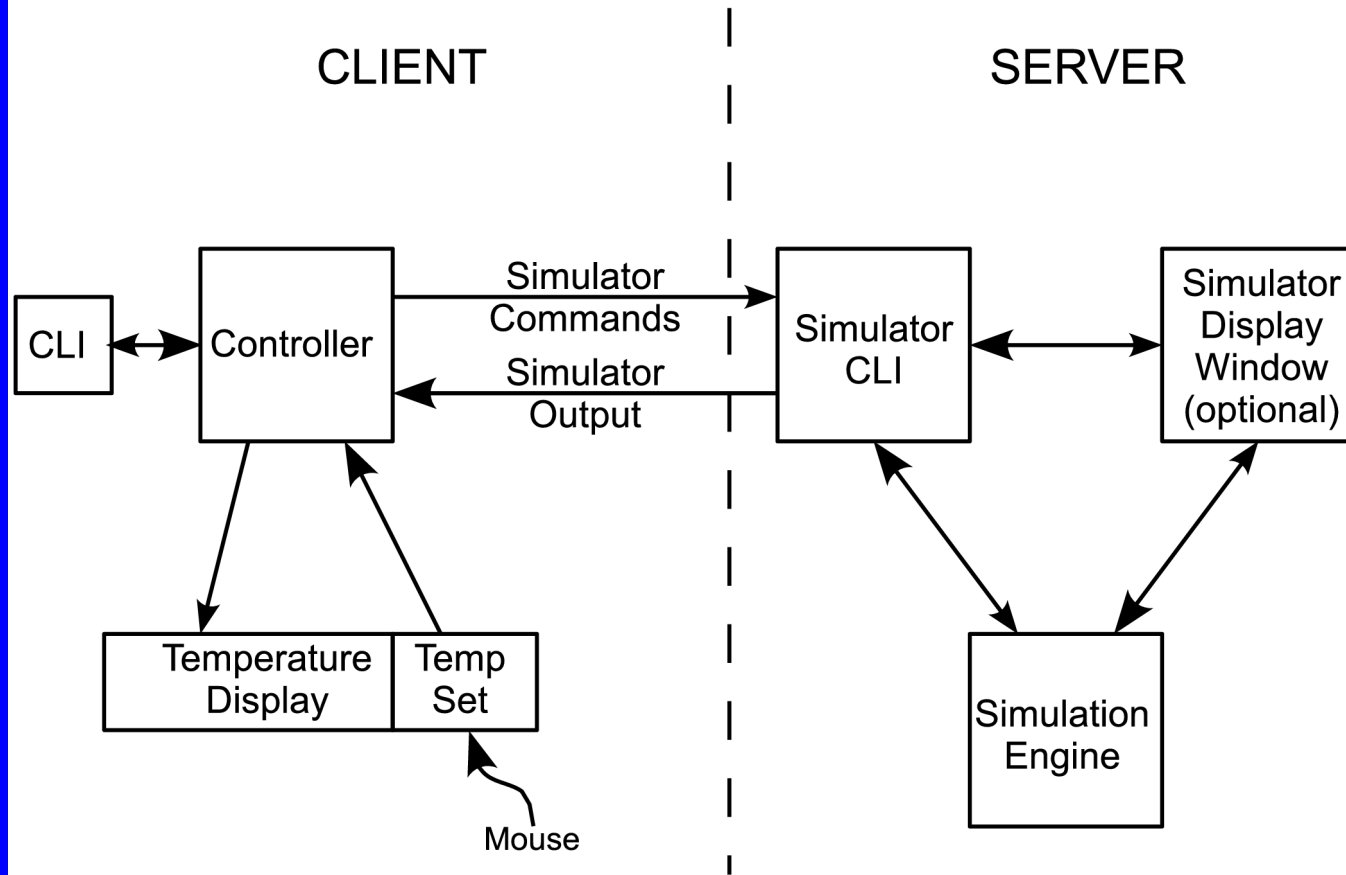
$$u(c, r, t+1) = u(c, r, t) + \alpha(u(c+1, r, t) + u(c-1, r, t) + u(c, r+1, t) + u(c, r-1, t) - 4u(c, r, t))$$



Implementation (64x64)



Simulation



3-D Heat Flow

- 5-planes
- Each plane 17x17 (not packed)
- Only 35% the elements of the 2-D module
(despite more sides, wider sums, more-complex arithmetic)
- Why? Wiring Efficiency

Speed Comparison

Heat equation simulation

1 timestep
 10^{12} elements
1 GHz clock

Scalar CPU	3.33 Hours
Multicore	16 Minutes
Cluster	16 Minutes
Reconfigurable	12 nSec

Startup/Configuration Time

10^{15} elements

1 GHz clock

CPU-Based Systems	2 Minutes
FPGA	278 Hours
2-D, Self-Configurable	50 Minutes
3-D, Self-Configurable	300 mSec

Defect Sensitivity

- CPU, Multicore, Clusters – Very sensitive
tend to want perfect yield
- FPGAs – Potentially less sensitive
Teramac; Easypath
- Self-configurable array – can tolerate some defects
Scan before use/bad element table (~scandisk)

Runtime-Upset Sensitivity

10^{24} elements: MTBF 2.8 *u*Sec

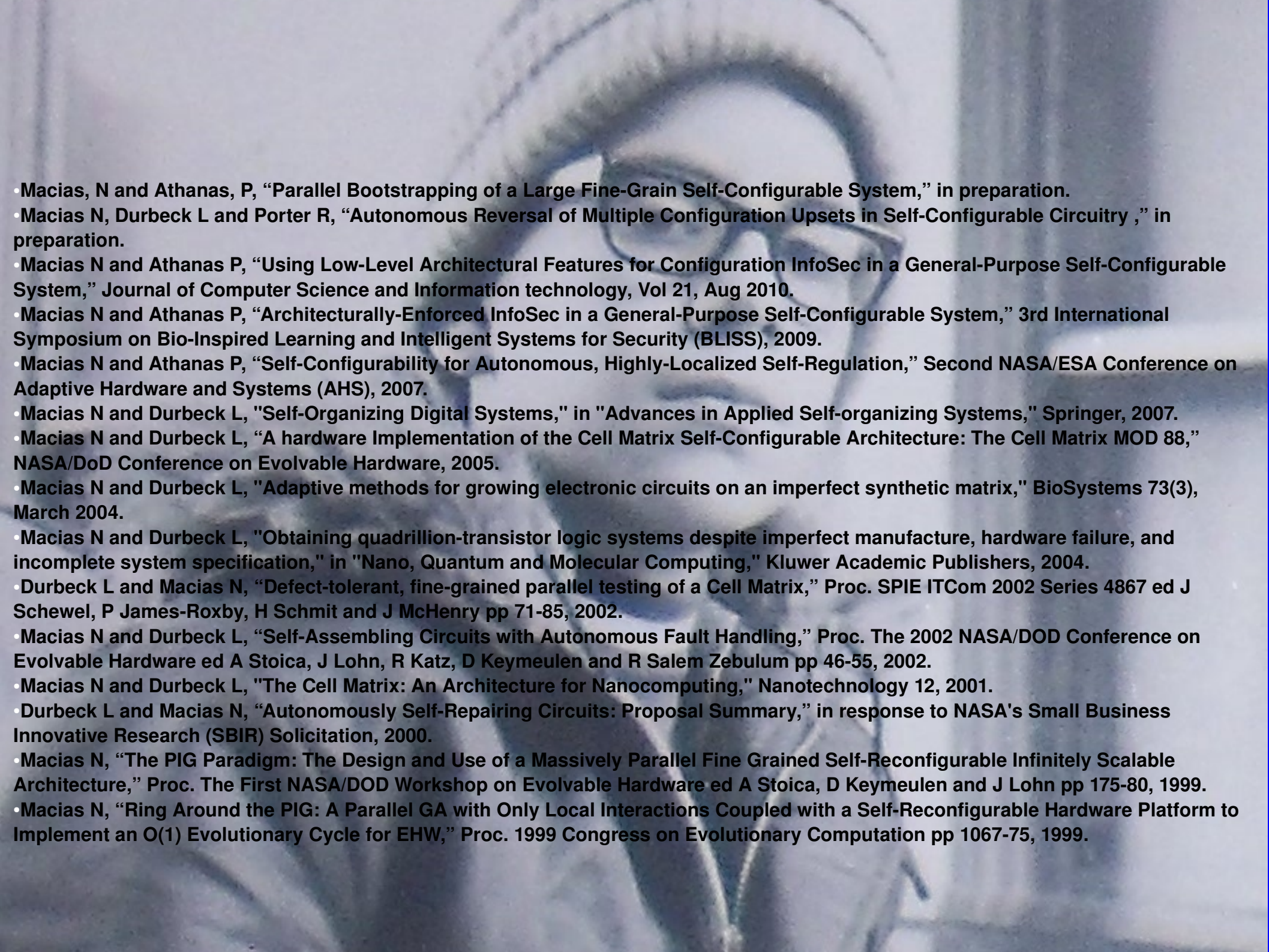
- CPU, Multicore, Cluster – Very sensitive
few “disposable resources”; control loc?
- FPGA – sensitive without redundancy etc
- Self-configurable array: can tolerate distributed upsets
3 copies+“scrubbing” as-needed
6 copies + continuous scrubbing

SUMMARY/CONCLUSIONS

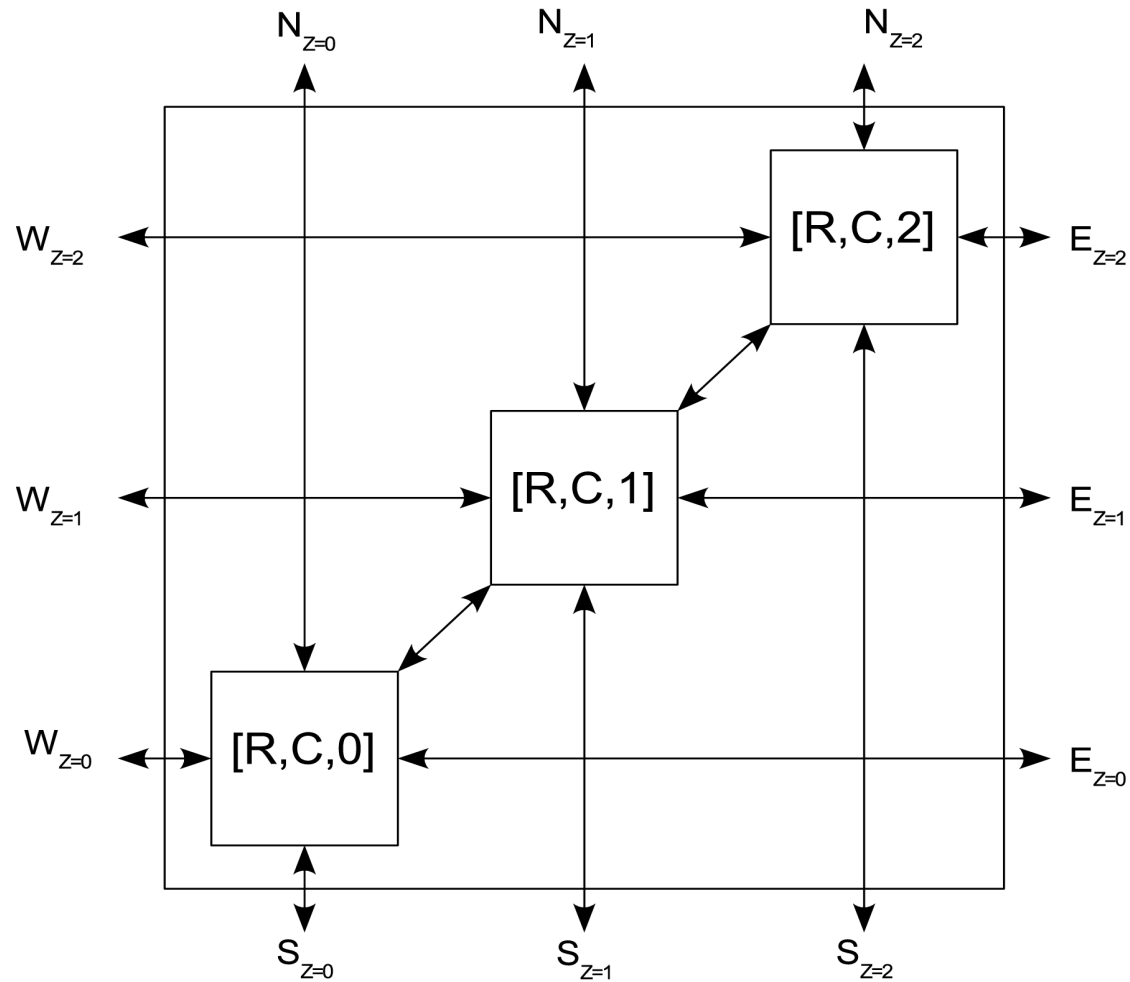
- Architecture addresses H1-H4
- Successful management of VLRS
- Internal configuration useful for fast bootstrap, including defect detection and avoidance
- Parallel test/parallel config
- In-vivo implementation is supported
- Sample problem analyzed, including 3-D sim
- Utility of thin 3-D array discovered

FUTURE WORK

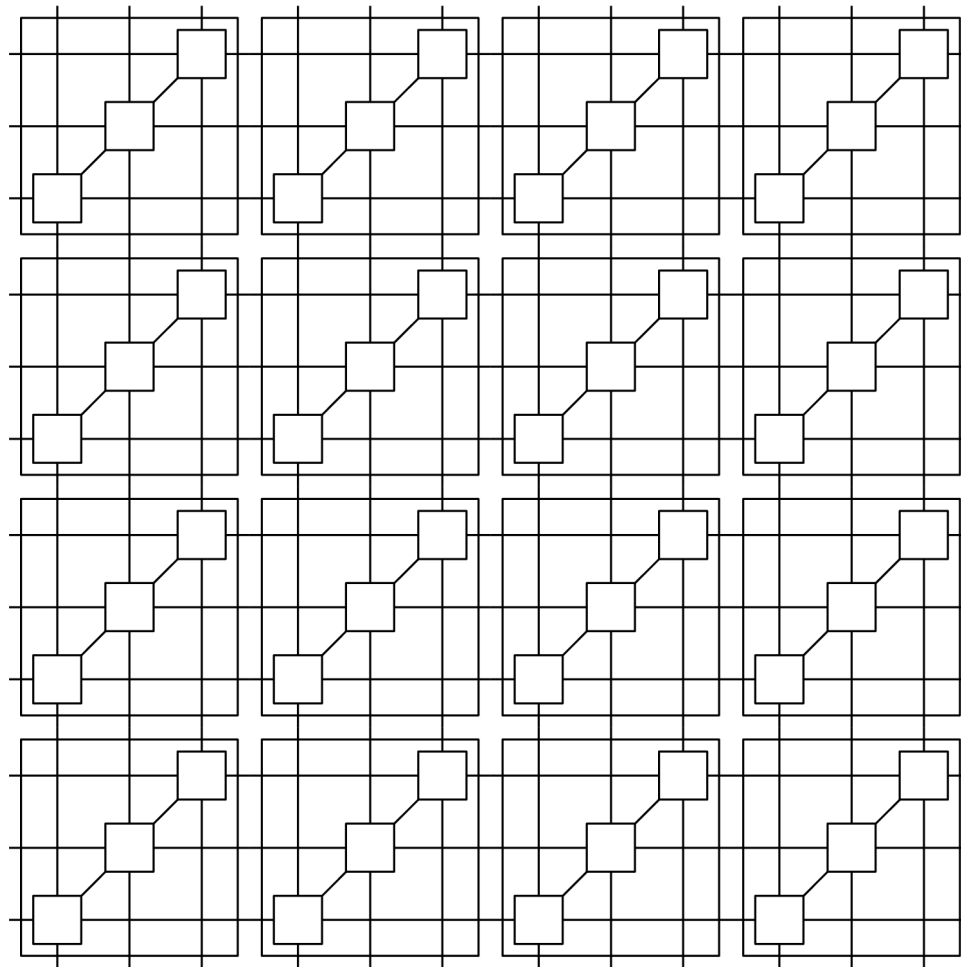
- Simulation
- Design tools
- Representation of dynamic circuitry
- Analog version
- Self-assembly
- A-matter
- Continuous TTs

- 
- Macias, N and Athanas, P, "Parallel Bootstrapping of a Large Fine-Grain Self-Configurable System," in preparation.
 - Macias N, Durbeck L and Porter R, "Autonomous Reversal of Multiple Configuration Upsets in Self-Configurable Circuitry," in preparation.
 - Macias N and Athanas P, "Using Low-Level Architectural Features for Configuration InfoSec in a General-Purpose Self-Configurable System," Journal of Computer Science and Information technology, Vol 21, Aug 2010.
 - Macias N and Athanas P, "Architecturally-Enforced InfoSec in a General-Purpose Self-Configurable System," 3rd International Symposium on Bio-Inspired Learning and Intelligent Systems for Security (BLISS), 2009.
 - Macias N and Athanas P, "Self-Configurability for Autonomous, Highly-Localized Self-Regulation," Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2007.
 - Macias N and Durbeck L, "Self-Organizing Digital Systems," in "Advances in Applied Self-organizing Systems," Springer, 2007.
 - Macias N and Durbeck L, "A hardware Implementation of the Cell Matrix Self-Configurable Architecture: The Cell Matrix MOD 88," NASA/DoD Conference on Evolvable Hardware, 2005.
 - Macias N and Durbeck L, "Adaptive methods for growing electronic circuits on an imperfect synthetic matrix," BioSystems 73(3), March 2004.
 - Macias N and Durbeck L, "Obtaining quadrillion-transistor logic systems despite imperfect manufacture, hardware failure, and incomplete system specification," in "Nano, Quantum and Molecular Computing," Kluwer Academic Publishers, 2004.
 - Durbeck L and Macias N, "Defect-tolerant, fine-grained parallel testing of a Cell Matrix," Proc. SPIE ITCom 2002 Series 4867 ed J Schewel, P James-Roxby, H Schmit and J McHenry pp 71-85, 2002.
 - Macias N and Durbeck L, "Self-Assembling Circuits with Autonomous Fault Handling," Proc. The 2002 NASA/DOD Conference on Evolvable Hardware ed A Stoica, J Lohn, R Katz, D Keymeulen and R Salem Zebulum pp 46-55, 2002.
 - Macias N and Durbeck L, "The Cell Matrix: An Architecture for Nanocomputing," Nanotechnology 12, 2001.
 - Durbeck L and Macias N, "Autonomously Self-Repairing Circuits: Proposal Summary," in response to NASA's Small Business Innovative Research (SBIR) Solicitation, 2000.
 - Macias N, "The PIG Paradigm: The Design and Use of a Massively Parallel Fine Grained Self-Reconfigurable Infinitely Scalable Architecture," Proc. The First NASA/DOD Workshop on Evolvable Hardware ed A Stoica, D Keymeulen and J Lohn pp 175-80, 1999.
 - Macias N, "Ring Around the PIG: A Parallel GA with Only Local Interactions Coupled with a Self-Reconfigurable Hardware Platform to Implement an O(1) Evolutionary Cycle for EHW," Proc. 1999 Congress on Evolutionary Computation pp 1067-75, 1999.

“Thin” 3-D (2.5-D)



Still scalable with 2-D assembly



Order of Cross-Sectional Bandwidth

Scalar	1
Multicore	$\min(\# \text{ Cores, Network BW})$
Cluster	$\min(\# \text{ Nodes, Network BW})$
FPGA	10^{12}
3-D Self-Configurable	10^{16}

Avogadro-Scale System

- reconfigurable element: 10,000 transistors
- “smart switch”/cell: 10^6 elements
- “organism”: 10^{14} cells = 10^{24} transistors

Avogadro-Scale System

- reconfigurable element: 10,000 transistors
- “smart switch”/cell: 10^6 elements
- “organism”: 10^{14} cells= 10^{24} transistors
- 15 orders of magnitude increase over today
- Optimistic estimate: $1.5 * \log_2(10^{15}) = 75$ years

Traditional Criteria

- Smaller transistors
this is, in some sense, **the** key
- Faster switching speeds
- Lower-power transistors
- Yield improvement

Modified Criteria

- Smaller transistors – important, but **not sufficient**
- Slower switches/greater parallelism
- Low clock frequency -> low power
- Defects can be tolerated post-manufacture

DENSITY (2-D)

- Assume 1nm x 1nm transistors
- 10^{24} transistors

area=1 km²

DENSITY (3-D)

- Assume 1nm x 1nm x 1nm transistors
- 10^{24} transistors

volume=1,000 cc