

Constructing A General Purpose Cellular Computer

An Explorative Approach to Nanocomputers

University of Oulu
Department of Information Processing Science
Master Thesis
Markus Koskimies
7th May, 2010

Preface

This is a revised version of the original Thesis. The main modification is font change, which has affected to figure placements. When making my Thesis, I used Linux's GGV to view PDF files. With that software, the default font looked alright, but I later noticed that it is faint in most other PDF viewers.

The content of the Thesis has not been changed.

This Master Thesis received the highest grade (Laudatur), and later the MAL national award for best annual Master Thesis in the fields of mathematics and natural sciences.

This Thesis is available:

<http://mkoskim.drivehq.com/>

Animations generated by CVM are uploaded to YouTube;

<http://www.youtube.com/user/MarkusKoskimies/videos>

For further information, the author can be contacted;

mkoskim@gmail.com

Autumn 2012

Markus Koskimies

Abstract

In this Thesis, implementation of general purpose cellular computers was researched. As hardware architecture, a cellular computer is a locally-connected network of cells. This network architecture is infinitely scalable making it possible to have large number of processing units, and according to previous research of nanocomputers it is currently the best known architecture for implementing a nanocomputer.

The constructive research method was selected for this research. During the research, a virtual cellular computer was implemented and used for experiencing selected set of computational problems. The nature of this research is explorative, since initially all the possibilities for implementing cellular computer were considered. Based on the previous studies of the subject, the most justifiable alternatives were selected.

At hardware level, the most essential problems for cellular computers are energy usage, heat dissipation, and defects and faults caused by different reasons (imperfect manufacturing, wearing, spurious errors). Because of these, it is predicted that cellular computer is self-reconfigurable, three-dimensional network of asynchronously operating cells.

The most important theoretical properties of general purpose cellular computer are computational and constructional universality. These properties were examined theoretically, as well as experimenting cellular computer for solving computational problems. The computational problems selected were implementing Turing machine, self-replicating machine, and algorithm to solve Eight Queen's Problem.

For studying the software development, architecture and the construction of operating system for cellular computer, the information gained from experiments was combined with previous research of software development for reconfigurable platforms. Based on this study, it is predicted that the software development methods are selected by criterias of human productivity, since progression in compiler technology will hide the low-level architectural differences.

The implementation of cellular computer operating system differs radically from the existing operating systems, since it needs to manage different resources (energy distribution, cellular resources, boundary cells) and it should be distributed, parallel and fault-tolerant like the underlying architecture.

Acknowledges

In order of appearance:

Prof. Kari Kuutti, for encouraging to select the subject of this Thesis.

Prof. Petri Pulli, for accepting the subject of this Thesis, and for his background work for helping to make this.

Dr. Peter Antoniac, for his long and strong support for making this Thesis, and invaluable guidance to the secrets of academic writing.

Prof. Juha Kortelainen, for helping with theoretical subjects.

Mr. Jouni Kokkonen, for his guidance to meet the typographical standards.

Mr. Antti Alasalmi, for his effort to push me to finalize this work.

Prof. Olli Martikainen (University of Oulu) and prof. Valentin Cristea (University Politehnica of Bucharest) for reviewing this Thesis.

Symbols and Abbreviations

General symbols and abbreviations

ALU	Arithmetic-Logical Unit
ASIC	Application Specific Integrated Circuit
CA	Cellular Automaton (pl. Automata)
CISC	Complex Instruction Set Computer
CNN	Cellular Neural Network
CNT-FET	Carbon Nanotube Field Effect Transistor
CPU	Central Processing Unit
DPGA	Dynamically Programmable Gate Array
DSP	Digital Signal Processor (Processing)
EDIS	Encyclopedia of Delay Insensitive Systems
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HL-HDL	High Level Hardware Description Language
HLL	High Level (programming) Language
HW	Hardware
IC	Integrated Circuit
ISA	Instruction Set Architecture
JVM	Java Virtual Machine
MPS	Message Passing System
OISC	One Instruction Set Computer
OS	Operating System
PC	Personal Computer / Program Counter
PCA	Plastic Cell Architecture
PE	Processing Element
PRAM	Parallel Random Access Machine
QCA	Quantum-dot Cellular Automaton (pl. Automata)
RAM	Random Access Machine / Memory
RASP	Random Access Stored Program (machine)
RAW	Reconfigurable Architecture Workshop (MIT)
RC	Reconfigurable Computing / Computer
RISC	Reduced Instruction Set Computer
RISP	Reconfigurable Instruction Set Processor
RLE	Run-Length Encoding
ROM	Read-Only Memory
RTD	Resonant Tunneling Diode

RWM	Read-Write Memory
SET	Single Electron Transistor
SW	Software
TM	Turing Machine
TMR	Triple Module Redundancy
UC	Universal Constructor
UTM	Universal Turing Machine
VLIW	Very Long Instruction Word (processor)
VHDL	Verilog HDL, Hardware Description Language
VLSI	Very Large Scale Integration
VM	Virtual Machine

Specific symbols and abbreviations, used in this Thesis

AC	Avogadro Computer (example system)
BB501	Busy Beaver 501
CAC	Cellular Avogadro Computer (example system)
CCOS	Cellular Computer Operating System
CVM	Cellular Virtual Machine
DEQPS	Dedicated Eight Queens' Problem Solver
DPU	Data Processing Unit
ECA	Environmental Conditions Adaptation
FS	Field Separator (reconf. stream)
ICU	Inter-Cell Connections Unit
LCM	Logical Cell Model
MLI	Machine Language Interpreter
RCU	Reconfiguration Control Unit
SS	Structure Separator (reconf. stream)
VNP	Von Neumann Processor (model)

Table of Contents

Preface	2
Abstract	3
Acknowledges	4
Symbols and Abbreviations	5
Table of Contents	7
1 Introduction	8
1.1 The Future of Computers	8
1.2 On Research Problem, Questions and Methods	10
1.3 Thesis Structure	11
2 General Purpose Cellular Computer	13
2.1 General Purpose Computing	14
2.2 Physical Realization of A Cellular Computer	18
2.3 The Scope of Modelling A Cellular Computer	28
3 Logical Cell Model	35
3.1 Cell Model	35
3.2 Computation	44
3.3 Example Designs	61
3.4 Reversibility	65
4 Experiments	68
4.1 Cellular Virtual Machine	69
4.2 Turing Machine Emulation	70
4.3 Self-Replication	76
4.4 Solving Eight Queens' Problem	79
4.5 Configuration Statistics	90
4.6 Theoretical Examination	94
4.7 Observations	97
5 Cellular Computer System	98
5.1 Cellular Computer Software	98
5.2 Cellular Computer Operating System	103
6 Conclusions	113
7 Further Research	115
References	118
Appendix A. CVM Details	126
Appendix B. Tape Content for Self-Replicating Loop	130
Appendix C. MLI Registers and Instruction Set	133
Appendix D. Eight Queens' Problem, C Source	136
Appendix E. Eight Queens' Problem, MLI Source	139
Appendix F. VNP, Von Neumann Processor	145
Appendix G. "Glasswalls": A Failed Protection Mechanism	149

1. Introduction

W. David Hillis starts his book “The Connection Machine” (1989) by stating that the computers of those days were too slow. After almost twenty years of development of computer systems the situation has not changed — eventhought our current computers are about 10,000 times faster than the ones Hillis is referring, we still need more processing power in order of magnitudes for even our daily computation. Having for example 10,000 times more processing power would allow us to use programs that are not practical today. Some of them are even impossible because of the related realtime requirements, like realtime ray tracing, on-the-fly video stream processing (noise reduction, sharpening, color correction) and true three-dimensional displays.

1.1 The Future of Computers

The permanent lack of computing power has driven the development of processor hardware. Ever-increasing size of the processor chips with increasing number of transistors combined with raising clock frequencies is a hard equation. It has raised difficult questions how to use the resources provided by hardware efficiently. We have already seen pipelined, super-scalar processors to come, their evolution to VLIW (Very Long Instruction Word) and other parallel architectures and the integration of complex co-processors and large cache memories inside of the processor chips.

From this ground has the research area of reconfigurable computing emerged — having its roots in the configurable electronic circuits like FPGA (Field Programmable Gate Array) and DPGA (Dynamically Programmable Gate Array), it introduces an integration of traditional sequential processors and reconfigurable hardware meshes (e.g. Bishop & Sullivan, 2003; Compton & Hauck, 2002; Hartenstein, 2001, 2002, 2003). Examples of both academic and commercial reconfigurable architectures are MIT RAW (Agarwal, 1999), STI (Sony-Toshiba-IBM) Cell, PACT XPP (Baumgarte *et al.*, 2003), Garp (Callahan *et al.*, 2000; Hauser & Wawrzynek, 1997) and PipeRench (Goldstein *et al.*, 2000).

But if I ask you to imagine the ultimately most powerful computer possible to be built, to move the problem of inadequate processing power to history, what are you thinking of? For me it takes just a few seconds to start to think about arrays of millions of processor with a size of a building; following with a processor meshes size of a planet like *Jupiter Brain*¹ or even structures floating freely in the space, like Robert J. Bradbury’s concept of

¹In science fiction, A Jupiter Brain is a hypothetical computing megastructure the size of a planet. It is composed from (hypothetical) *computronium*, material engineered to maximize its use as a computing substrate (www.wikipedia.org).

*Matrioshka Brain*².

Although those are just dreams in science fiction, there are already strong theoretical evidence that the ultimate computer in distant future really is a processor mesh. According to theoretical observations of Vitányi (1988), used for argumentation by Frank & Knight (1998), the locally-connected or mesh-type networks and sub-graphs of these are the *only* scalable networks in 3-D space and that no physically realizable interconnection model can perform significantly better. Frank (2003a) goes even further when concluding that “...since it seems that physics itself can be modeled ... as a 3-D mesh of fixed-capacity quantum processors, interacting locally ..., it follows that no physically realizable model of computing can be more powerful asymptotically than such a model.” (p. 10)

A mesh of millions of processors does not necessarily need to be physically large, if the processing element is small enough. Using nanoscale devices (dimensions between one to ten nanometers, that is from 10 to 100 atomic diameters) arranged in three dimensions, it could be shrank to a desktop or even palmtop sized computer.

The concept of producing matter and devices at atomic or molecular scale was first introduced by Feynman (1959) in his talk given at an American Physical Society meeting at Caltech. The term *nanotechnology* was defined by Taniguchi (1974) and it was explored in much more depth by Drexler (1986, 1998). The ultimate objective is to construct a useful *Avogadro Computer* (Durebeck, 2001) - a usable computer system containing an order of 10^{23} switches.

The extensive research on nanoscale devices has given us a lot of new knowledge about the future computers. In their research of nanoelectronic devices in highly parallel computing systems, when selecting the candidate architectures Fountain *et al.* (1998) made an observation that “...highly regular, locally connected, peripherally interfaced, data-parallel architectures provided an almost ideal match to the likely properties of many nanodevices” (p. 31). Furthermore, based on the analysis of several researches about nanoelectronic devices, Beckett & Jennings (2002) conclude that “...future nanocomputer architectures will be formed from non-volatile reconfigurable, locally-connected hardware meshes that merge processing and memory.” (p. 148)

So, it seems, that the relationship between massive processing arrays and nanotechnology is bidirectional; not only the massive processing mesh needs to be fabricated from nano-scale devices (making it inherently a nanocomputer, i.e. a computer, which characteristic feature length is in nanoscale), but also a nanocomputer seems to be inherently a processing mesh.

In the light of this evidence, it seems inevitable that the future computers are based on hardware processing meshes. In the era of reconfigurable computing they are playing a side role, concentrating on what they are best for — performing computations to large amounts of data. But when the era of nanocomputers begins and eventually leads to a

²In science fiction, a Matrioshka Brain is a hypothetical computing megastructure, based on the Dyson sphere, of immense computational capacity. It is an example of a Class B stellar engine, employing the entire energy output of a star to drive computer systems (www.wikipedia.org).

situation where there seems to be no competitive computer architectures any more, the meshes need to execute general purpose software.

The interesting question is, how is this done? How are these processing meshes programmed to solve useful computational problems, like fetching a web page?

1.2 On Research Problem, Questions and Methods

The objective of this Thesis is to gain understanding about cellular arrays as general purpose computers. This objective can be divided to three separate questions;

1. What kind of physical structure a cellular computer has?
2. How cellular computer is built as general purpose computer?
3. How to implement and organize operating system and software to a cellular computer?

The constructive research method, also called as design science research methodology, was a natural choice for this research. In this research methodology, artifacts are built and evaluated, bearing in mind that the artefacts should bring some utilities. Artefacts built in constructive researchs fall into one of four categories; constructs (or concepts), models, methods and instantiations. According to Vaishnavi & Kuechler (2005), design science consists of the two basic activities, *build* and *evaluate*.

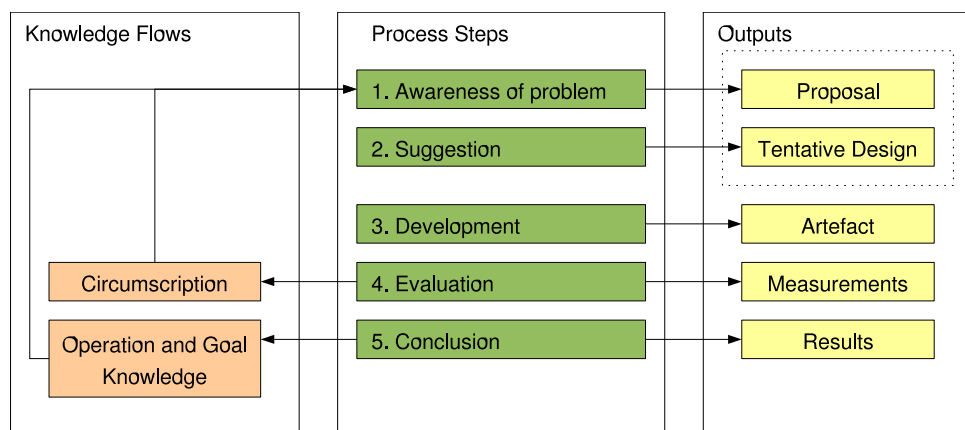


Figure 1. A common view of the design science process. (cf. Vaishnavi & Kuechler, 2005)

A common view, as well as the individual phases of the design science process is shown in Figure 1. The phases of the process are;

1. In phase 1, the problem is clarified. The output is a proposal for a new research effort.

2. In phase 2, a tentative design is proposed. Suggestion is a creative step that ends in a new configuration of new or existing elements.
3. In phase 3, an artefact is implemented according to the tentative design.
4. In phase 4, the artefact is evaluated according to functional criteria made implicit or explicit in the suggestion phase. Phases 3 and 4 give additional information that is brought together and fed into the suggestion phase for another round with a new or modified design.
5. The conclusion phase terminates a specific design project, but there may be anomalies that serve as subject for a new project or research.

The constructed artefact in this Thesis was a model of a general purpose cellular computer. The basic requirements as well as evaluation criterias for the model were collected by analysing the previous research of cellular processing meshes and fabrication of nano-scale processing devices. These methods were used for answering the first two research questions.

To be able to answer the third question, the results and observations from construction were combined with existing research of reconfigurable computing to look the cellular computer operation from wider perspective. This perspective also provided more evaluation criterias for the cellular computer constructions, which may be used in future researches.

1.3 Thesis Structure

Chapter 2 analyses and refines the properties of a general purpose cellular computer. First, the theoretical properties of a general purpose computers are discussed, after what the constrains of physical implementations are discussed. Then there is discussion about scoping made for this Thesis.

Chapter 3 introduces the cellular model, called Logical Cell Model (LCM), used in this Thesis for modeling a cellular computer. It is based on the analysis made in the previous chapter, but it is refined so that it were possible to be used for implementing a virtual machine for evaluating it.

Chapter 4 represents the experiments conducted with a cellular virtual machine based on the model constructed in the previous chapter. In addition to empirical results, this chapter contains also theoretical view to the constructed model.

Chapter 5 contains discussion about cellular computer software and its software architecture. This chapter combines the experiments made with the cellular virtual machine to the research of reconfigurable computers, to show how a cellular computer may operate.

The appendices contain more detailed information of the artefacts constructed in this Thesis, as well as one example of failed design.

2. General Purpose Cellular Computer

As discussed in the opening of Thesis introduction, the physical architecture of future computers is very likely a locally connected network of simpler processing elements, i.e. a processing mesh. The relation between processing meshes and nanotechnology is bidirectional; not only a large processing mesh need to be constructed from nanoscale devices to make it compact, but also the research of nanoscale devices suggests that the most suitable architecture for nanocomputers is a processing mesh.

This Thesis will only discuss about speculative very large processing meshes. The practical evidence³ shows that with the current gate count, the conventional sequential architectures are superior compared to other computer architectures as a general purpose computers. There are many signs that increasing gate count pushes the next generation of computers to hybrids (called reconfigurable computers), which join the conventional sequential processors and mesh-like processing devices. There are also strong evidence that meshes are ultimately the best possible choice for physical computer architecture (Vitányi, 1988; Frank & Knight, 1998). But it is impossible to say, how many gates there need to be to make purely mesh-based architectures practically more suitable than other architectures. All that is known, is that the gate count is probably magnitudes higher than today.

What are *general purpose cellular computers*?

The word *computer* tells that it is a machine performing computations. Moreover, it implicitly tells that it is a physical computing machine, not only a theoretic model of computation. In this case, since this Thesis concentrates on machines that are not yet possible to be build, it means that the machine is physically realizable.

The word *cellular* tells that the internal architecture of the machine is cellular; that is, it is a network of cells, and to be more specific, it is a locally connected network of cells. As the computer is physical entity, the term cellular refers to its physical architecture. That is, a conventional computer running a cellular automaton is not a cellular computer, while a cellular computer running a Turing machine is still a cellular computer.

What does the *general purposivity* mean? Fundamentally it means that the computer is not limited to perform computations in one specific field, but it can be used for performing many different kinds of computations. This implies two things; a general purpose computer (1) has a complete set of primitive operations, which, when correctly combined together can describe any kind of a computation, and (2) can be internally configured with those descriptions, i.e. its program can be changed.

³Computers at our desks

So, a general purpose cellular computer is a machine, which;

1. does computations,
2. is physical or at least physically realizable,
3. is physically a locally connected network of cells (processing elements),
4. has a complete set of operations, and
5. can be internally configured to perform different kinds of computations.

In this chapter, these properties are examined closer. The first section will discuss about the computational capabilities, looking to the computation theories. The next section will discuss about the physical realization, looking to the research of nanotechnology. The rest of the sections will refine issues that has been found to be important to notice.

2.1 General Purpose Computing

General purpose computers have two significant properties. First, their primitive operation set, which is used to describe the computations, is selected so that it is possible to describe and perform any computation with that specific computer. From the theoretical point of view, this means that the computer — or specifically the model of the computer — is *Turing complete*. Second, the computer needs to be programmable; it has to have capability to be reconfigured on the fly, and more specifically, it needs to be self-reconfigurable.

A system that does not fulfill these theoretical requirements, i.e. it is either not Turing-complete, or not reconfigurable, is likely not a general purpose computer, while it is not necessarily impractical; it may be used for special-purpose applications, e.g. as a process control device — it is not known, if human brains fulfill these, while arguable demonstrated to be quite practical problem solvers.

Furthermore, a system that fulfills these requirements, is not necessarily practical; it may be inefficient, slow, or difficult to use, e.g. Malbolge. Thus it is necessary to keep in mind pragmatic aspects when considering realization of theoretically suitable general purpose machine to a computer.

2.1.1 Turing Completeness

While developing a formal definition for an algorithm, Alan Turing (1936) developed a theoretical machine (later named to Turing machine, TM) capable of solving any computable problem. The Turing machine contains an infinite tape, a head reading and writing

the tape and a state machine. Turing continued the development of his model by defining a machine capable of simulating any other Turing machine by reading the state transitions from the tape before executing the program; this machine is called Universal Turing machine (UTM), or simply universal machine.

Many computation constructions have been formally proven to be equivalent to UTM; they are called Turing-Equivalent, Turing-Complete or simply (computation) universal. Basically, if a machine is proven to be equivalent of one computationally universal machine, it is then also equivalent to UTM, which means that it shares the Turing machine's and equivalents property of being capable of solving any computational problem.

It is notable, that only models can be really computationally universal; because physically realized machines are resource bounded, they are only theoretically capable of solving any computable problem if unbounded resources (namely, time and memory) are provided. Thus, computational universality is often loosely attributed to physical machines or programming languages that *would* be universal *if* they had indefinitely enlargeable storage. All modern computers are computationally universal in this lax sense.

In modern automata theory, automata are classified to certain classes based on their computation power and the Turing machines form the most powerful class of automata. It has been proven, that a system computationally less powerful is not able to simulate computationally universal system — thus most of the proofs of computational universality is done by showing that a system is capable of simulating a Turing machine or some its equivalent.

This capability of simulating one universal machine with another has also very interesting practical consequences. Modern computer systems have many (loosely) Turing-complete constructions; especially processing models and programming languages. Figure 2 shows a diagram of Turing-complete systems running inside other equivalent systems. It shows an application written with Python, which is executed by an FPGA running synthesized x86 processor, executing a Java Virtual Machine executing Python interpreter. FPGA, x86 architecture, VHDL, JVM, Java programming language, C programming language, Java Bytecode and Python programming language are all Turing-complete models.

Conventional sequential computer systems follow the RASP (Random Access Stored Program) machine model, which is an example of von Neumann architecture. RASP is a specialized form of a RAM (Random Access Machine) model, which is a so called register machine with indirect register addressing. Cellular computers follow the CA (Cellular Automata) model, which are often defined as a regular lattice of discrete cells. There exists several well-known computation universal cellular automata, e.g. von Neumann's 29-state two-dimensional CA (von Neumann, 1966), Conway's *Game of Life* and *Rule 110* one-dimensional elementary cellular automaton (Wolfram, 2002; Cook, 2004). Since the purpose of this Thesis was not to construct a formal definition of a cellular computer, the CA theories are not examined any closer.

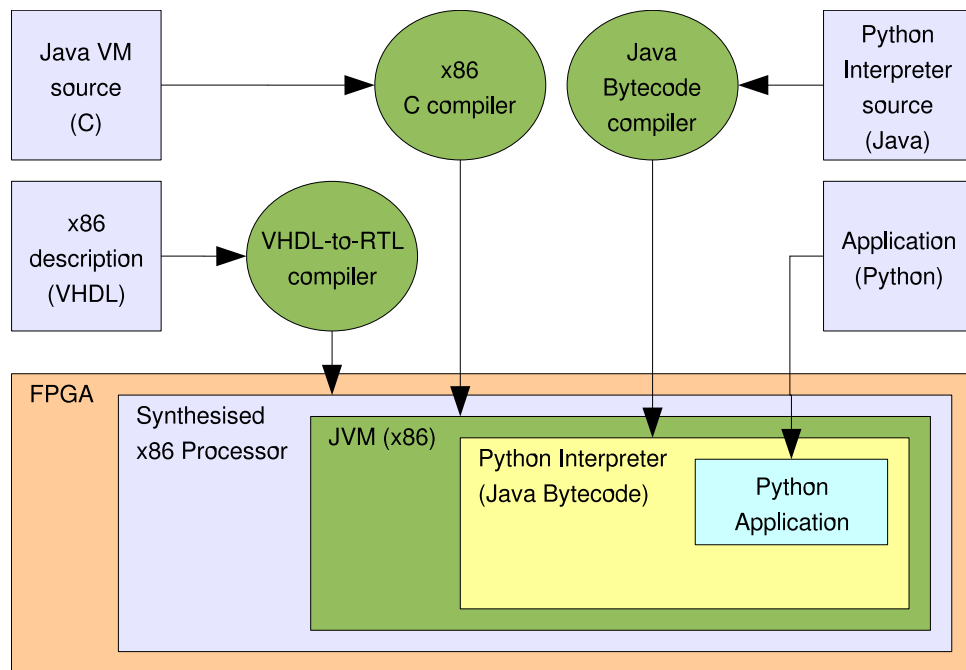


Figure 2. Machines inside machines.

2.1.2 Self-Reconfigurability

Coarsely, configurable systems could be divided to two categories; 1) self-reconfigurable systems, capable of changing their internal configurations dynamically and (obviously) partially by themselves, and 2) systems, which cannot do this.

A general purpose computer system definitely requires the ability to load, store, generate and invoke new programs. Our current personal computers, as well as their ancestors back to decades, all belong to the first class. The second class contains mainly special purpose computation devices, for example configurable logic components in computer hardware, which are e.g. (1) reconfigurable, but not self-reconfigurable i.e. reconfiguration requires external devices, or (2) configurable, but not reconfigurable, often called *one-time-configurable* devices.

Mathematically, the ability of freely changing the internal configuration is only considered in cellular automata. A cellular automaton, which is capable of hosting a Universal Constructor (UC) (von Neumann, 1966), is self-reconfigurable; we later refer to this capability by saying that the model is construction universal. Universal Constructor is a theoretical machine, which is able to construct any other machine (including itself), when the description of the machine is given. It was invented by John von Neumann when he was mathematically studying biological self-replication. His approach to self-replicating machines was to construct a UC, although Langton (1984) later show that construction universality is sufficient, but not necessary requirement for self-replicating systems.

The capability to generate descriptions of a machine for universal constructor is related to a question of computation universality — if a system is computation universal, it is then possible to have a machine outputting any arbitrary stream of symbols. Thus it is capable

of generating any configuration to a universal constructor.

If a system is computation universal, but not construction universal, it can simulate a system which is both computation and construction universal (as were discussed in previous section). As an example, consider systems in Figure 3. Although construction universality is seldomly applied to sequential processors, they are used as an example because they are more familiar to majority of audience. The figure shows a comparison of two computer architectures, *Harvard* and *von Neumann* processor architectures⁴.

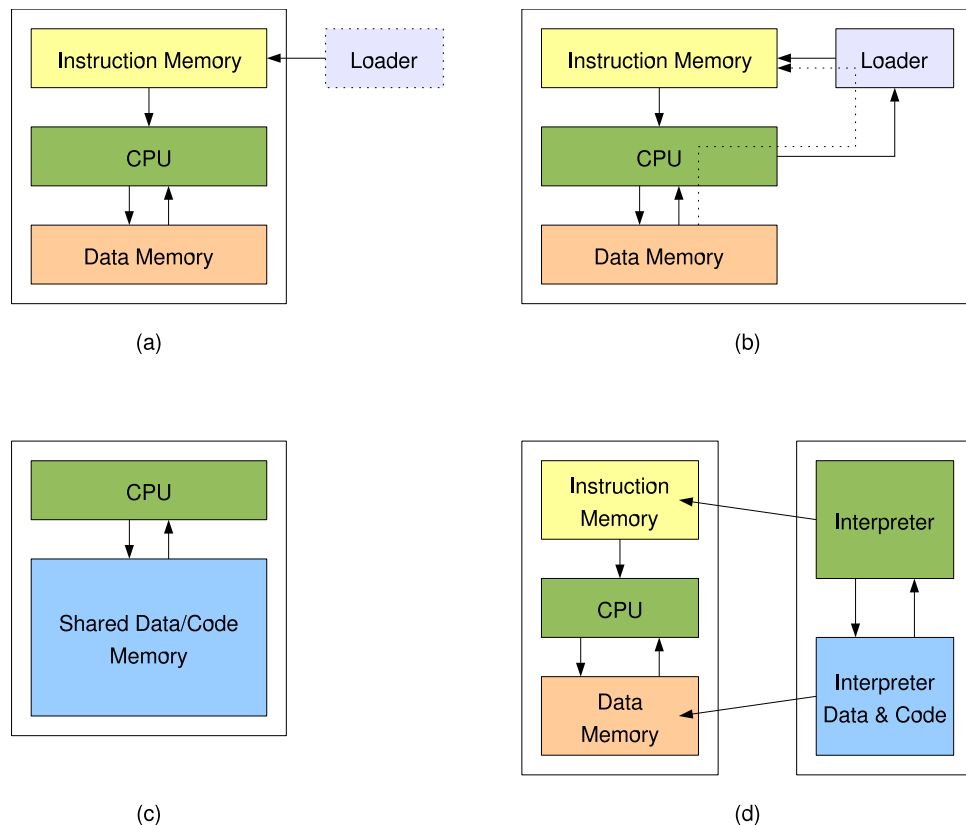


Figure 3. Architectures and self-reconfigurability. (a) A Harvard architecture system. (b) A Harvard architecture system with internal loader. (c) A von Neumann architecture system. (d) A Harvard architecture host system with virtual machine following von Neumann architecture

The Figure 3 (a) shows a simple Harvard architecture processor; since it cannot perform write access to its instruction memory, it is not capable of loading new programs, i.e. it is not self-reconfigurable. The configuration can be changed by an external device (e.g. a “prommer”). If the loader is attached as a CPU controlled peripheral (Figure 3b), the processor can transfer new programs from its data memory to the instruction memory. This makes the system self-reconfigurable; it means, that the system may contain a compiler, and the compiled result can be executed by writing it to the instruction memory.

⁴The Harvard architecture is very common in microcontrollers (e.g. Microchip PIC) and DSPs (Digital Signal Processors), because it allows the instructions and data having different widths (in bits). The von Neumann architecture is very common in general purpose processors, like modern PCs and their ancestors back to decades, like Commodore-64, MSX and Sinclair Spectrum.

A von Neumann architecture (Figure 3c) is inherently self-reconfigurable. A Harvard architecture processor system can be (permanently) programmed to execute a virtual machine or interpreter (e.g. Java bytecode, BASIC, Perl, or Python bytecode interpreter), which executes programs stored in the data memory — the host system itself is still not self-reconfigurable, but the interpreter, following von Neumann’s architecture may be (Figure 3d).

2.2 Physical Realization of A Cellular Computer

The physical architecture of a cellular computer is a locally connected array of processing elements. These processing elements are constructed from interconnected devices (*switches*) performing some simple logical operations. Figure 4 shows an illustration of a regular lattice of identical cells.

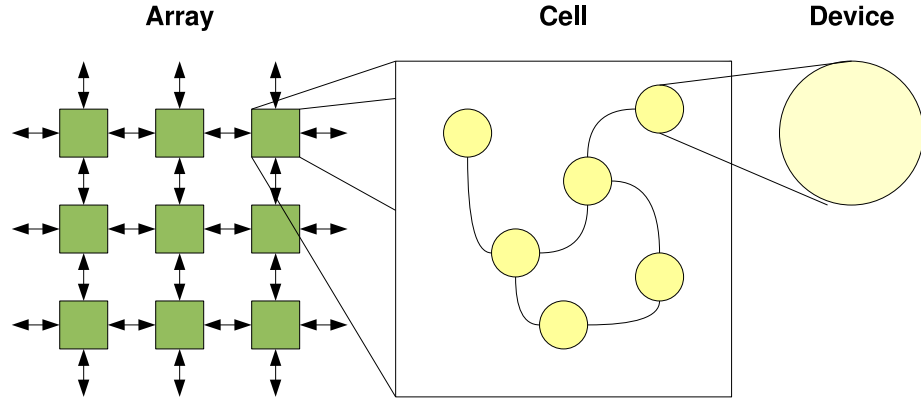


Figure 4. Array, cell and device.

For demonstration purposes, two imaginary systems are represented. The first example system is simply an AC (*Avogadro Computer*). Its main characteristic is the switch count $N_s = 10^{23}$. The derivation of this system is a CAC (*Cellular Avogadro Computer*); it shares majority of the characteristics of AC, but it also lays down an assumption of a cellular physical structure. Each cell in CAC has $N_{sc} = 10^6$ switches, so the cell count N_c of CAC is;

$$N_c = \frac{N_s}{N_{sc}} = \frac{10^{23}}{10^6} = 10^{17}$$

In the examples, the parameters selected for the systems are only for demonstration, i.e. they are not based on any research of physical devices.

2.2.1 Nano-Scale Device Technologies

The potential base technologies to implement nano-scale logical devices are at least electrical, mechanical, chemical and quantum mechanics. Optical device technology, i.e. using photons for carrying information is not possible at nanometer scale (Frank, 2003b, p. 72). An overview of possible technologies is shown in Figure 5.

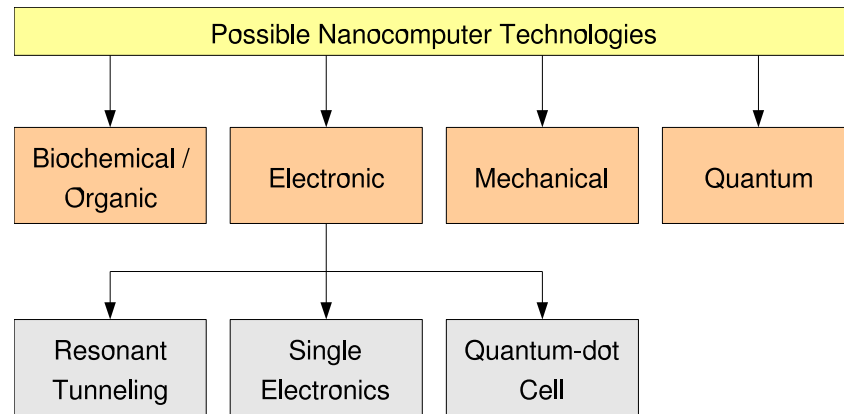


Figure 5. Examples of potential nanocomputer technologies (cf. MITRE).

Electrical switching technologies are most widely researched, because their operation is based on a technology proven successful for decades; these successors of transistors are e.g. SET (Single Electron Transistor), RTD (Resonant Tunneling Diode) and CNT-FET (Carbon Nanotube Field Effect Transistor). These technologies are mainly researched for replacing the current transistor technology with faster, smaller and low-consuming solutions.

There also exist non-switching electron-based devices, like QCA (Quantum-Dot Cellular Automaton). The most interesting feature in QCA is that there is no current flow during operation — basically this means, that no energy is consumed during state transitions. Because of this, QCA has been widely researched (e.g. Vankamamidi *et al.*, 2005; Niemier & Kogge, 1999, 2004a,b; Niemier *et al.*, 2004; Ravichandran *et al.*, 2005; Niemier *et al.*, 2000; Chaudhary *et al.*, 2005).

Mechanical switching technologies are advocated in publicity by e.g. K.E. Drexler and R. Merkle (e.g. Drexler, 1986, 1998; Merkle, 1993). In mechanic designs, the signal is carried by atoms unlike in electrical designs, where the signal is carried by electrons. Atoms are much more massive than electrons making them accelerating slower, so it is expected that nanoscale mechanical devices are slower than their electrical counterparts. But they may be denser, because the shorter tunneling distance of atoms compared to electrons (Merkle, 1993; Frank, 2003a).

Chemical nano-scale technologies are mainly biologically-inspired. Currently the most research in this field is done for so called “DNA computing” i.e. manipulating DNA strings to perform computations. The biggest advantage of DNA computation is that the

nanoscale devices already exist, that is, virae, bacteria and biological cells. These living organisms have already solved many of the problems related to nanoscale devices, like energy efficiency and fabrication (i.e. self reproduction). Most of the current research is still theoretical, e.g. the Sticker model for programming (Roweis *et al.*, 1996) and a simulation environment for that (Carroll, 2002).

Quantum computation means exploiting the quantum effects for computation. The most interesting feature of quantum effects are that the *qubit*, quantum bit, can express both binary values (i.e. 0 and 1) at the same time. Exploiting this effect with interfering may provide very fast solutions for specific problems which take exponential times on conventional computers, like factoring large numbers.

2.2.2 Dimensions

The physical dimensions of the system are coarsely dependent on the size of the switch and the dimensions supported by fabrication process i.e. dimensions usable for spatial cell arrangement. Spatial arrangement can happen physically in one, two or three dimensions.

The CA/CAC switch is defined as a cube, which edge length $l_s = 1 \text{ nm}$. Since the approximation of carbon atom diameter is 0.22 nm , the switch is constructed approximately from $4.5^3 = 93.9$ carbon atoms.

The Table 1 shows the edge lengths of the AC and CAC systems with different spatial arrangements. If the switches are arranged in one dimension, they form a string with length of approximately 670 AU (Astronomical Units, $1 \text{ AU} \approx 150 \text{ million km}$) — the average distance between Sun and Pluto is approximately 39.4 AU. If the switches are arranged in two dimensions, the planar array takes up $316 \times 316 = 99,856$ square meters (approximately 10 hectares) i.e. about a size of ten soccer fields — definitely far from being compact or portable. In three dimensions, the equivalent number of switches form a cube with a 4.6 centimeter edge.

Table 1. Physical dimensions of array.

Dimensions	Edge of the array (l_a)		Edge of the cell (l_c)	
	m	in cells	nm	in switches
1-D	100 billion km	10^{17}	1 mm	1,000,000
2-D	316.2	316 million	$1 \mu\text{m}$	1,000
3-D	0.046 (4.6 cm)	460,000	$0.1 \mu\text{m}$	100

So, the only possibility to fabricate a compact size computer containing massive numbers of switches is to arrange the switches in three dimensions. Using more dimensions has more advantages than just the compact size; it can improve the performance of parallel computations since the machine has a lower average distance between processing elements, which reduces the communication delays (Frank & Knight, 1998; Frank, 2003b).

Unfortunately, three is maximum number of spatial dimensions usable in our Universe.

The problem with three-dimensional arrangement in cellular computer is that although the mesh network itself is infinitely scalable, some other physical constrains are not. Basically, the array can be infinitely large at two dimensions, but the *effective thickness* i.e. the machine depth at third dimension is limited by heat dissipation and energy distribution constrains (Frank, 2003b, p. 50).

Figure 6 shows two methods for utilizing the third dimension more efficiently. First, the irreversible operations generating waste heat could be placed to as near of the “surface” of the machine as possible (a). This could be done by the software tools, combined with an aid of software architecture. Second, the planar machine could be wrapped (b).

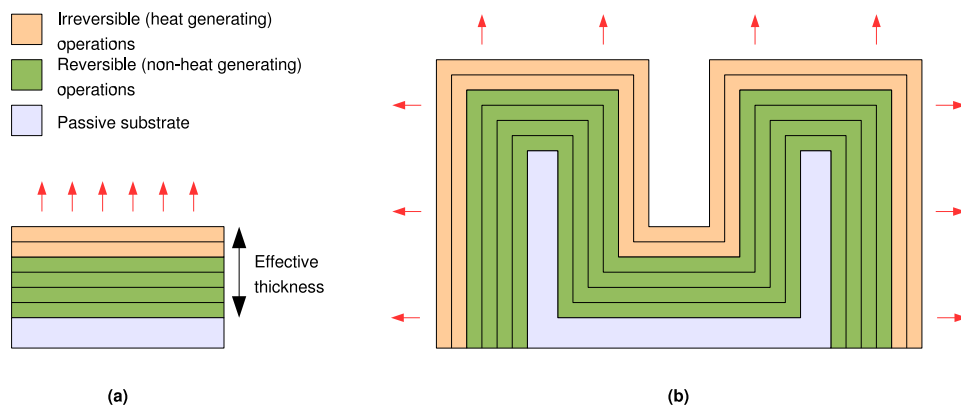


Figure 6. Utilizing third dimension. (a) by placing operations generating heat at surface for increasing the effective thickness, and (b) wrapping the machine.

Although the mesh is infinitely scalable, the energy distribution and heat removal are necessarily not. When the number of devices increase, the more space is needed for feeding the necessary amounts of energy and for cooling down the system. Thus, the actual physical construction of a massive cellular processor is probably not a pure mesh, but more likely a kind of recursive hierarchial (tree-like) three-dimensional structure, like a Menger sponge (Figure 7).

2.2.3 Energy Consumption

To understand the constrains of energy consumption, it is important to know for what is the energy used by the computers. In this Thesis, this is described in a simplified manner. A more detailed description of the energy consumption of computation can be found from Frank (2005).

Constants used in the following calculations are;

Boltzmann's constant	k	$= 1.38^{-23} J/K$
Electron volt	$1 eV$	$= 1.60^{-19} J$

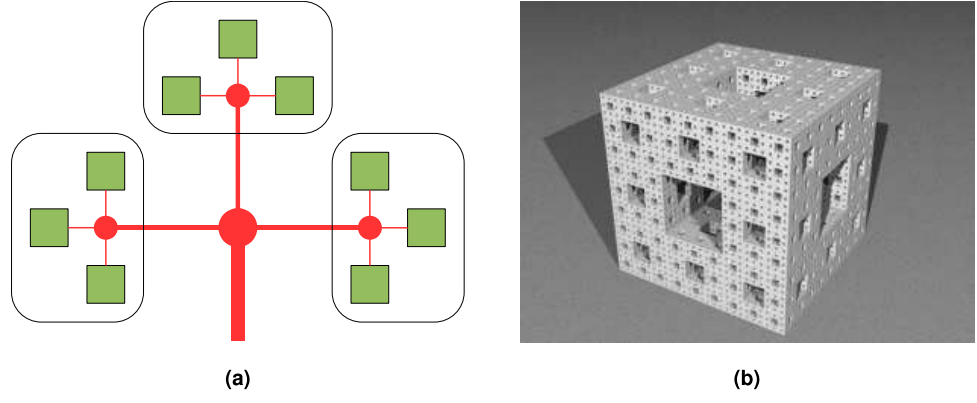


Figure 7. Energy distribution and structure. (a) As the number of cells increase, the channels distributing energy get thicker, thus forming a tree-like structure. (b) An illustration of Menger sponge level 4. Image source: Wikipedia. Author: Amir R. Baserinia, 2006.

To distinguish separate states of a signal carrying information, there need to be an energy difference high enough between these states; that is, changing the state of the signal should require energy enough not to happen spuriously. This amount of energy is called signal energy E_{sig} and the smaller it is, the higher is the error probability p_{err} of the signal. By giving a desired error probability, the minimum signal energy can be calculated with following equation;

$$E_{sig} = kT \ln \frac{1}{p_{err}}$$

where;

$$\begin{aligned} E_{sig} &= \text{minimum signal energy [J]}, \\ k &= \text{Boltzmann's constant [J/K]}, \\ T &= \text{environment temperature [K]}, \\ p_{err} &= \text{desired error probability} \end{aligned}$$

For example, if the desired error probability $p_{err} = 10^{-40}$, at the room temperature $T = 293^\circ K$ the minimum signal energy $E_{sig} = 92.1 kT \approx 2.38 eV$. According to Frank (2005), the practical lower limit for the signal energy is around $40 kT$, which is approximately $1.2 eV$ at room temperature ($T \approx 300^\circ K$).

For completing a computational task, the computer performs a specific number of logical operations ($N(op)_{tot}$). These operations can be divided to two categories; (1) irreversible, information destructive operations ($N(op)_{irr}$), and (2) reversible, information conservative operations ($N(op)_{rev}$). Destroying a signal dissipates the same amount of energy that the signal was formed, i.e. $E_{dis} = E_{sig}$, so the minimum energy consumption of a computational task depends on the number of irreversible operations ($E = E_{sig} * N(op)_{irr}$). This fundamentally-type energy consumption of irreversible logic was first recognized by Landauer (1961). Lowering the number of irreversible operations

by converting the computation to use reversible operations lowers the energy needed to complete the task without lowering the signal energy.

In modern computers, the $E_{sig} \approx 10^4 \text{ eV}$ — that is, a logical high level is represented by presence of approximately 10,000 electrons at 1 volt potential above ground level — and they operate entirely with irreversible logic. When the state of a signal line in modern computer is changed from logical high level to logical low level, all the electrons representing the signal are released to the signal ground causing the system to dissipate (without considering leakage) 10,000 eV of waste heat. A modern computer is thus like a huge high-speed bit crematorium.

A system does not need to be fully reversible; as Merkle (1993) states, that a modern computer could perform (a) all the other operations except register and memory writes, or (b) all reversible operations (e.g. NOT) in reversible manner.

Even a fully reversible systems consume some amounts of energy. First, the reversible system needs an initial kinetic energy to be charged. Reversible systems naturally have adiabatic losses caused by friction, which should be replenished.

To be feasible computing devices for common people, AC/CAC computers should operate at room temperatures and they should not require overwhelming energy sources. Thus the operating temperature is defined to be approximately 20°C ($T_{op} = 300^\circ\text{K}$) and the output power P_{tot} of energy source is defined to be 400 W .

The constant energy loss (leakage) $P(l)$ of the entire system is defined to be a reasonable 40 W , 10 percentage of the total amount of energy available. The loss is caused by the selected device technology, so to achieve this the average energy loss of a single switch $P(l)_s$ should be at maximum;

$$P(l)_s = \frac{P(l)}{N_s} = \frac{40 \text{ W}}{10^{23}} = 40 * 10^{-22} \text{ W} \approx 0.003 \text{ eV/switch/s}$$

In other words, if the switch is electrical and the operating voltage is 1 V , the switch leaks one electron every 5 minutes 45 seconds. Practically this means, that the leakage energy of the switch should be nearly zero in massive cellular arrays. This means, that the switch should remain its state without conserving notable amounts of energy.

Since AC/CAC computer systems use future device technologies, the signal energy may be as lower as $50 \text{ kT} \approx 1.47 \text{ eV}$. Thus, the number of irreversible operations supported by the power source is;

$$N(op)_{irr} = \frac{P_{tot} - P(l)}{E_{sig}} \approx 1.97 * 10^{21} \text{ op/s}$$

When this is compared to the number of switches in the system, the amount of switches capable of performing irreversible operations per second are;

$$R = \frac{N(op)_{irr}}{N_s} = \frac{1.97 * 10^{21}}{10^{23}} \approx 0.020 \approx 2.0\% (1 : 50)$$

Since the switches operate much faster than 1 Hz frequency, the amount of simultaneous irreversible operations is much lower. The clock frequencies of modern computers are approximately 2 GHz, but there is already existing of switching technologies operating at 100 GHz frequencies (Likharev & Semenov, 1991; Likharev, 1996). In fact, one of the advantages of mesh architectures is the locality, which allows much higher frequencies to be used.

If the switching frequency (f) is moderately slow 10 GHz, the switching time is;

$$t = f^{-1} = \frac{1}{10^{10}} = 10^{-10} \text{ s} = 0.1 \text{ ns}$$

The average number of simultaneous ($t = 0.1 \text{ ns}$) irreversible operations supported by the energy source is then;

$$N(op)_{irr} * t = 1.97 * 10^{21} * 10^{-10} = 1.97 * 10^{11}$$

and the ratio

$$R = \frac{t * N(op)_{irr}}{N_s} = \frac{1.97 * 10^{11}}{10^{23}} = 1.97 * 10^{-12} \approx 1 : 500,000,000,000$$

That is, one out of 500 billion switches can perform an irreversible operation simultaneously.

The cellular AC (CAC) shares the same energy consumption calculations with the AC at switch level. One operation performed by a cell is called a transaction, since it includes not only the computation, but also exchanging the results with neighbor cells. If it is assumed, that during one transaction in average 50 bits are destroyed by irreversible switching operations (each cell contain 1 million switches), the average transaction energy consumption without losses E_t is;

$$E_t = 50 * E_{sig} = 65.3 \text{ eV}$$

In this case, the maximum amount of transactions per second with 400 W energy source is;

$$N(t) = \frac{E_{tot}}{E_t} \approx 3.8 * 10^{19} \text{ transactions/s}$$

The CAC contains $N_c = 10^{17}$ cells. If all cells in the system are constantly performing transactions, the maximum transaction frequency f_t of the system with 400 W energy source is;

$$f_t = \frac{N(t)}{N_c} \approx 382 \text{ Hz}$$

This may well hold true with biologically-inspired technologies. Higher frequencies are possible, if the transaction energy can be lowered, or if not all cells are making transactions simultaneously.

If the technology allows high-speed transactions and the consumption cannot be dropped low enough, the availability of the energy comes a problem. The consequences are technology-dependent; for example, an array built from electrical devices may cause damages to the energy source or the cells may suffer of so called *brown-outs* causing unreliability when overloading the system.

There are at least three possible methods to solve this; 1) artificially lowering the maximum transaction frequency to the level, where the energy source can support simultaneous transactions of all cells, 2) at the software architectural level it is ensured, that no more than the maximum allowed transactions can happen simultaneously, and 3) the system tolerates the variations in transaction speed.

The method 1) would mean a huge performance drop, if assuming that in average not all cells are performing transactions. The method 2) is complex and error-prone, thus a possible source of malfunctioning software and device breaks. The method 3) sounds the most reasonable. It requires, that (a) the cell performs the transaction only, when there is enough energy available, (b) if there is not enough energy available, the cell does nothing, and (c) the system tolerates the variations in the transaction times. It may also require, that the energy distribution to the cells is “fairy” i.e. no cell remains unpowered to the infinity because of the activity of the others.

One implication is that increasing the maximum transaction frequency with technological changes does not necessarily increase the performance of the array; it does it only to the level at where the incoming energy is fully used. Also, the best performance of the software is reached by trying to minimize the transactions needed to complete the tasks; the amount of transactions in the array are hugely dependent on the availability of energy making the exploitation of the natural parallelism of the array more challenging.

2.2.4 Heat Generation

The energy consumed by the system is finally transformed to waste heat — thus the system dissipates as much heat as it consumes energy. For a system driven by a 400 W energy source the overall heat generation should not be a problem in general. The problem may be the concentrated heat generation in the deeper layers of the array; a cell performing

transactions frequently can reach the maximum temperature and get permanent damages. There are at least two alternative methods to prevent this happening; 1) ensuring at the software level, that no cell is frequently active, and 2) using localized thermal control. The system cannot be protected against overheating at the global level; for example, one thermal sensor at the outside of the array would react after the damage has already been happened, because the temperature of the array reacts slowly to the temperatures of single cells.

Like with the energy management, the method 1) is complex and error-prone. Coupling the overheating problem at cell level should be feasible; it requires, that (a) the cell is able to monitor its temperature and (b) adapt the transaction speed to the temperature, and (c) the system tolerates the temporal variations in the transaction timings. Some technologies can provide this implicitly i.e. the operation frequency is dependent on the temperature of the devices.

How about measuring the temperatures of all the cells and stopping the entire array, if one of the cells get overheated? In synchronously operating arrays, without considering the difficultness of distributing clock signal to 10^{18} cells and other obstacles in the front them, this could be one of the possible mechanisms to prevent cell damages due overheating while still maintaining the correctness of the computations in such systems. The first problem would be to built a 10^{17} -input NAND circuit to detect the overheating. The next one would be, that stopping the entire system because of one overheated cell would cause a notable performance loss. But the main problem would be, that one malfunctioning part in this overheating detection system would cause the entire system to be unusable — if one malfunctioning cell permanently reports overheating, the system would not even start.

As a conclusion, in addition to the performance coefficients derived from the technological selections (e.g. maximum transaction frequency, transaction energy consumption) and the availability of energy discussed earlier, the overall performance of the system is also dependent on the the efficiency of removing the generated heat from the system, i.e. the efficiency of heat dissipation i.e. cooling. The more efficient is the heat dissipation, the more energy can be fed to the array to support transactions and the more frequently the cells can operate.

2.2.5 Defects, Faults and Errors

In their survey of presence of defects and faults at nanoscale devices, Graham & Gokhale (2004) define the terms *defect* and *fault*;

“A defect, or more specifically, a manufacturing defect is a physical problem with a system that appears as a result of imperfect fabrication process. By contrast, a fault is an incorrect state of the system due to manufacturing defects, component failures, environmental conditions, or even improper design. Faults can be; (1) permanent, [...], (2) inter-

mittent, [...], or (3) transient.”

In current IC manufacturing processes, after the silicon wafer is fabricated, post-fabrication tests are carried out and defective parts are rejected. But it is unlikely, that a system containing massive amounts of switches can ever be perfectly fabricated; it is expected that the scaled down technologies would show higher error probabilities than the current technologies.

Consider the example AC system. It contains 10^{23} switches. Even if one out of a quadrillion (10^{15}) switches is broken, it means that the system contains 100 million (10^8) broken switches. In the worst case, if all the broken switches are in different cells (in CAC system) and the cells do not tolerate single broken switches, there are 100 million (10^8) non-functional cells in the system i.e. “only” 10^9 functional cells — 1 out of 100 cells is broken.

As a result, throwing away parts that contain any defect at all will not be an option. Instead, the fabricated devices should be designed so that they are defect tolerant. It means that the device as a whole remains operable even if it contains even large number of inoperable, malfunctioning and faulty switches and inter-switch routings.

This is one good reason to predict the usage of modular (cellular) self-reconfigurable hardware with future computers; their tolerance against fabrication failures has already been demonstrated with Cell Matrix (Durbeck & Macias, 2000) and several FPGA-based studies, e.g. the “Teramac” computer (Amerson *et al.*, 1995; Heath *et al.*, 1998). As this Thesis discusses about self-reconfigurable processing meshes like the CellMatrix, the same approach can be used; the faulty cells are determined with self-testing capabilities, and faulty cells are not used in routing and placement process (Durbeck & Macias, 2000; Yatsenko, 2003).

An interesting research area in cellular hardware systems is so called embryonics, i.e. self-repairing hardware, which is based on dynamic fault detection and correction mechanisms, e.g. the self-reconfiguration. Although being interested especially for critical systems, this subject is scoped out from this Thesis.

Decreasing the device size and signal energy makes the devices also more prone to spurious errors. As an example, Graham & Gokhale represent a TMR (Triple Module Redundancy) method; three copies of the same hardware are executed with common inputs and the result is determined by a majority vote circuitry. The TMR reliability can be improved by using three voters, removing the voter as the single point of failure.

This expected unreliable nature of nanoscale devices has also raised questions, if exact computation is even suitable for those devices, or would some non-traditional computing methods, e.g. probabilistic logic or (cellular) neural networks (CNN) work better with future nanocomputers. It is true, that the nature of many algorithms, especially related to artificial intelligence (e.g. pattern recognition, process and robotic control) do not require exact computing.

For this Thesis, the question is irrelevant. The subject of the Thesis is general purpose cellular computers. A general purpose computer needs to be exact and Turing-complete, because otherwise it is not able to do mathematical computing. Although specified, or even most algorithms could be implemented using non-traditional computing models, as long as there is a need for numerical mathematics, there is a need for physical realizations of exact computing devices. If they cannot be fabricated from nanoscale devices or simulated by hardware using non-exact computation, they are fabricated with other technologies and they are attached with a device performing non-exact computations.

2.3 The Scope of Modelling A Cellular Computer

Since there is no existing cellular computers available at the moment, the only possible way to study those is to abstract the computers to a model. The previous sections have discussed about subjects that are most likely relevant to all future realizations, but at this point it is necessary to scope the cellular computer model used in this Thesis.

The following subsections discuss about specific scoping.

2.3.1 Abstraction Level

To be usable device technology for constructing a massive cellular computer, it should allow (1) use of reversible logical operations, and (2) three-dimensional device arrangement. Since nearly all bit-device systems proposed have some sort of reversible variant (Frank, 2003b, p. 26) and place no apparent limitations for three-dimensional arrangement, it is impossible to say at the moment, what or which device technologies will be used in the future nanocomputers. Although a novel guess may be electronic, the physical attributes like size and switching capabilities are just the beginning; building an array with 10^{23} switches raises other problems, like defect tolerance, energy distribution, heat generation and operational temperatures, which may be found too difficult to solve on some otherwise promising technology.

Thus, it is necessary to raise the abstraction level high enough to hide the device technology details, but nothing more. If the abstraction level is raised too high, e.g. a PRAM (Parallel Random Access Machine), the results from the model does not any more directly match to the future realizations of the cellular computers.

In this Thesis, the cellular computers are abstracted to operation level, close to the level used in models by Macias (1999) and Konishi *et al.* (2001). Many cellular automata studied for this Thesis are too low-level abstractions, e.g. models used and/or proposed by Perrier *et al.* (1996), Sipper (1999, 2004), Mange *et al.* (2003), Wolfram (2002), and Peper *et al.* (2003).

At operation level, each logical cell is capable of performing a simple operation, like

adding two inputs together. The cell is also assumed to compute the results of the operations reliably; the underlying hardware is assumed to deal with possible device-level unreliability. As such, this abstraction level could be considered as a cellular computer ISA (Instruction Set Architecture), which could be implemented in various technologies (Figure 8), or even as a virtual machine on a top of another ISA.

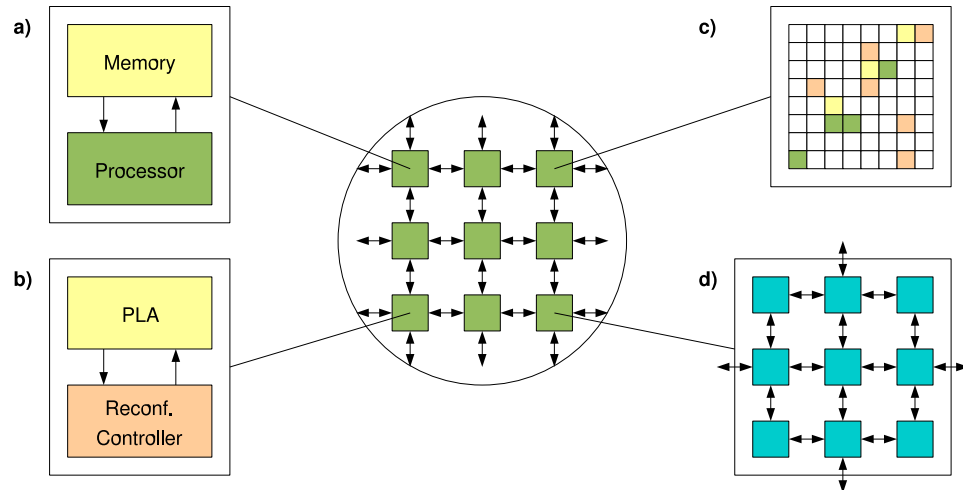


Figure 8. Examples of potential internal architectures of a nanoscale processing elements. Internally, the processing may be (a) sequential processor, (b) programmable logic array, (c) a simple cellular automaton, or (d) a locally network of simpler processing devices.

It is also hard to try to predict the granularity of the future cellular computer. Beckett & Jennings (2002) and Macias (1999) vote for a fine-grained future, while Frank (2003b) criticizes the assumptions made by Macias and advocates a more coarse-grained future. When abstracting the cellular computer to operation level, it makes no assumption of the underlying granularity. Depending on the technology, the logical operation may be constructed from several simpler cells, or one more complex cell may perform several logical operations by itself.

2.3.2 Irregular Arrays

There are several issues that make irregular arrays more interesting than regular ones;

- **Fabrication.** By definition, irregular arrays do not need perfect intercell connection formation, which may make them to be suitable for mass production. Using self-replicating cells and pressing them together could mean that the future supercomputer is as cheap as a sheet of paper.

Another fabrication mechanisms, that create irregular arrays are so called paintable computers, and all connection models that trust on ad hoc networking.

- **Fabrication defects.** Even with perfect lattice formation the malfunctioning cells or broken connections make the arrays functionally irregular.
- **Mechanical Reconfiguration.** In this Thesis, the reconfiguration is assumed to be logical i.e. it is a mechanism which changes the predefined state of the cell without reshaping it. But it may be possible that the future cells could be mechanically changed, e.g. several cells could be joined together to form one single logical entity. This could mean that even perfect regular homogeneous arrays would be better modelled with irregular models.

During the making of the Thesis, also irregular structures were quickly explored (Figure 9). A computer program were used to generate an irregular cellular array. In irregular arrays, the directions of connections vary from cell to cell. Thus an attempt to configure a straight line of 10 cells (a configuration stream adjancing to the first connection of the cell) produced just random pattern. By giving the cells a directional information (i.e. sorting the connections by the angle of the direction of the connection), the same reconfiguration stream produced a line. But still, the line is different in different positions of the array, because of the different local topology.

The Figure 9 (d) shows a result of an artificially generated reconfiguration stream to produce regular pattern on irregular array. It demonstrates that a regular, i.e. designed logical patterns can be reconfigured even to highly irregular arrays. This justifies the decision of modeling a cellular computer with regular array, since a novel reconfiguration engine could also perform the necessary manipulations to the configuration stream to produce the desired pattern.

One example of studying irregular and heterogeneous cellular arrays is NANA, Nanocomputer Active Network Architecture, introduced by Patwardhan *et al.* (2006). It is a research of programming non-uniform irregular arrays with high defect rates. The array is built from three specialized cells (processor, memory and memory port). Their approach to program the array is using active network. The network carries both data and instructions to be used for that data.

That research showed another, much simpler method for creating irregular patterns; first generate a regular structure, then randomly removing cells from the structure and finally shuffling the directions ends to an irregular array. It is thus possible to use the regular model developed for this Thesis also to study irregular arrays, although those studies probably concentrate to the reconfiguration mechanism.

2.3.3 Heterogeneous Arrays

Heterogeneous arrays are arrays, where the cells are not identical. In this Thesis, it was decided to model homogeneous regular arrays, that is, the array is regular lattice of identical cells.

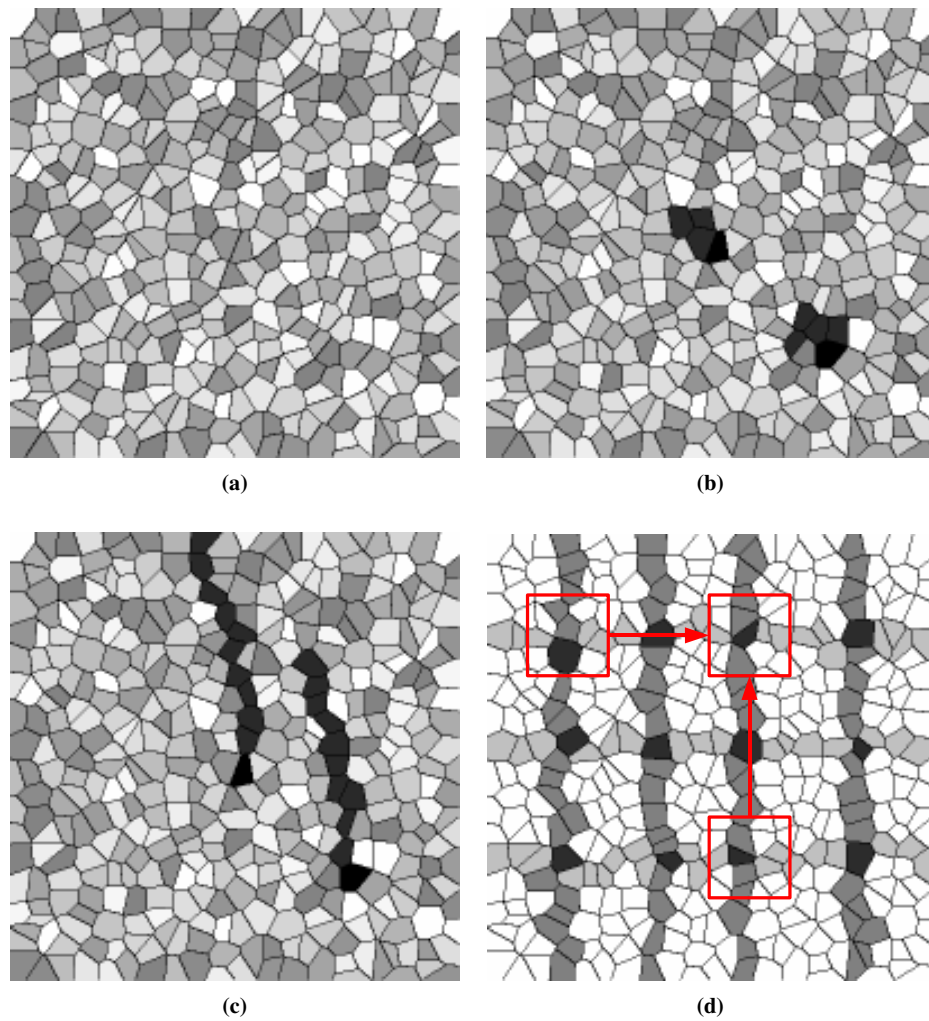


Figure 9. Irregular cellular arrays. (a) A computer-generated irregular cell array. (b) Attempt to configure a straight two lines. (c) By giving the cells directional information, the line is created, but it is different in the different locations of the array. (d) Artificially created configuration stream generates a regular structure.

Because of this, a solution for each requirement and consideration must exist either at the cell level or at the array level. If the solution is being solved at array level, it must be found from a cellular structure, i.e. the solution is constructed from co-operation of identical cells obeying the locality of connections. The problems cannot be solved by adding a new domain between array and cell (Figure 10), thus making the array heterogeneous.

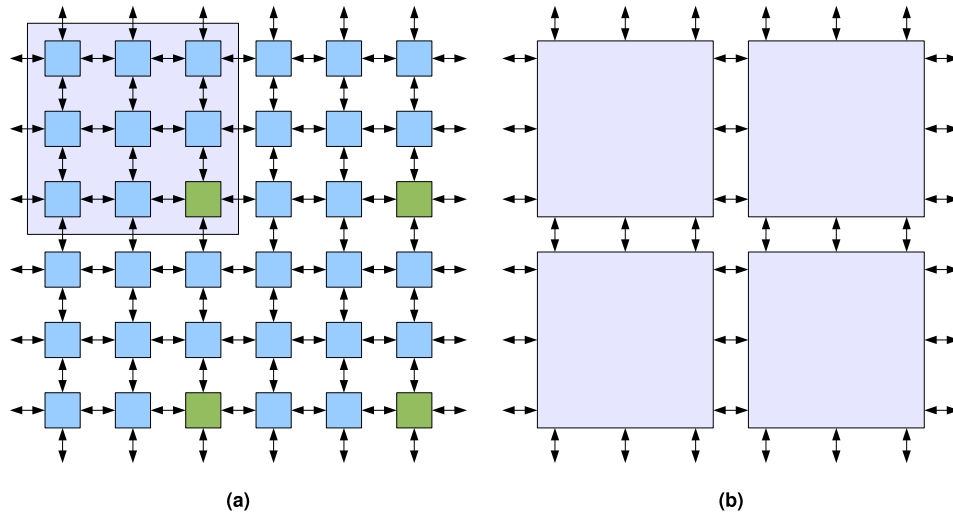


Figure 10. Solving domain problems. (a) It is illegal to solve a problem by adding a new intermediate domain with a domain controller (dark box)(a). (b) Instead, the complexity of the cell is increased to move the solution to cell level, if there were no solution at array level (interoperation of cells).

Homogeneous regular array is a very common model for cellular computing; nearly all cellular computer model proposals studied for this Thesis are such, for example the Cell-Matrix by Macias (1999) and PCA (Plastic Cell Architecture) described in Ito *et al.* (2003) and Konishi *et al.* (2001).

2.3.4 Delay-Insensitivity (Asynchronicity)

Asynchronous, delay-insensitive design gives some definitely considerable advantages discussed earlier; since asynchronous array tolerates the variations in computation timings, it makes possible to have sensible solutions for lack of energy and temperature management.

In their study of asynchronous cellular automata, Peper *et al.* (2003) state that the main advantages of asynchronous designs over the synchronous ones in the cellular arrays are;

1. No need for distributing clock signals;
2. the asynchronous systems have average-case instead of worst-case performance; in the synchronous systems the clock frequency is dictated by the slowest possible path, even if this path is only seldomly active.

3. Reduced energy consumption and heat dissipation, because only the parts doing the processing are active. This is an invaluable advantage for massive cellular arrays as seen earlier in this chapter.
4. Changes in the timings of the signals, caused by the variations in the physical conditions or implementations, do not cause the operations to fail. As seen earlier, this is as well an invaluable advantage for the massive arrays.
5. Asynchronous system can be divided into modules that can be designed without considering other modules, especially with the respect to timing relationships. For designing large, complex systems, this is a very important advantage over the synchronous designs.

More advantages can be found from Peper *et al.* (2004) and Hauck (1995). Goldstein (2005) notes the advantages of asynchronous circuits, but also the cost of it i.e. the increasing size of the logical device. Altogether, it is quite hard to imagine that the massive arrays could allow arbitrary large synchronous elements; thus, in the end, the massive arrays are likely to be collections of asynchronously operating blocks.

2.3.5 Reconfiguration Mechanism

Like stated by Macias (1999), a centralized reconfiguration for massive arrays is impractical for many reasons, like;

1. fault tolerance; if the (centralized) reconfiguration controller is broken, the whole array is unusable,
2. impractical resource (time) usage; sequential reconfiguration in the example system would take approximately 317 years without considering reconfiguration signal delays, if it is expected that one cell can be reconfigured in 10 ns (i.e. 100 million reconfigured cells in second).

Furthermore, the definition of the system in this Thesis prevents using external controllers for reconfiguring the array, because self-reconfigurability is part of the computational requirements set for the general purpose cellular computer. As the model was scoped to regular homogeneous array in the previous section, the cells need to be reconfigurable just like in CellMatrix and PCA models, i.e. there is no possibility to add a reconfiguration block managing several cells to the model; this is of course possible at the hardware level.

One seldomly issued criteria for reconfiguration mechanism is that it must be protectable against intentional and unintentional harmful reconfiguration attempts⁵. Intentional attempts are caused by *malware*; unintentional attempts are caused by malfunctioning applications.

⁵This was not mentioned in any articles studied for this Thesis

An interesting question is that is the reconfiguration reversible or irreversible operation? As discussed earlier, the irreversible operations may be concentrated on the surface of the computer to ease the energy distribution and heat dissipation. But if reconfiguration is an irreversible operation⁶, then it will require energy feeding and cooling the inner parts, too, since reconfiguration is an operation which must be available for each cell in the system.

Furthermore this would mean that even the parallel speed of the reconfiguration is limited i.e. the energy supply may not tolerate all cells to be configured simultaneously. Basically this means that the massive array is fully configured after a longer period of time i.e. it is initially only partially configured (*formatted*, like a PC hard drive). This means that it is highly probable that at least the cell configurations are non-volatile!

⁶Intuitively it is, since while changing the configuration it at the same time removes the previous one.

3. Logical Cell Model

The previous chapter discussed about modeling cellular computers in general. Since one of the objectives of this Thesis was to experiment the problem solving with a virtual machine, there was a need to specify the array model more exactly.

The virtual machine implemented for experimenting the problem solving is simply called CVM, Cellular Virtual Machine. The model used in the CVM is simply called a LCM, Logical Cell Model. The model contains many properties that can be varied in different implementations, but when used with CVM those were also given some reasonable values. When describing the LCM, there are also presented the selections, choices and values used in CVM.

One of the hardest decision made for the CVM was if the LCM separates the reversible and irreversible operations from each other. Although it is definitely important issue for massive cellular processing arrays, it was decided to left out from this Thesis. The reasons to do this were; (1) it has been formally proven, that all irreversible operations can be performed in reversible manner although not necessarily easily, and (2) the expected audience of this Thesis is more familiar with the conventional irreversible logic. Despite of this, the reversible alternatives are represented in many design choices.

In this chapter, the cell model is first represented. After that, the chapter concentrates on describing the computation model.

3.1 Cell Model

Figure 11 shows the overview of the LCM (Logical Cell Model). The LCM design is divided to four logical units;

1. ICU (Inter-Cell Connections Unit), which connects the cell to its neighbors,
2. DPU (Data Processing Unit), which process the inputs from neighbor cells,
3. RCU (Reconfiguration Control Unit), which is responsible for reconfiguring the cell behaviour when requested, and
4. ECA (Environmental Conditions Adaptation), which can be used to implement environmental behaviour, e.g. heat dissipation and energy feeding.

As a general design principle, the LCM was designed so that it maintains scalability of locally connected networks, which can be regarded as the most interesting property of

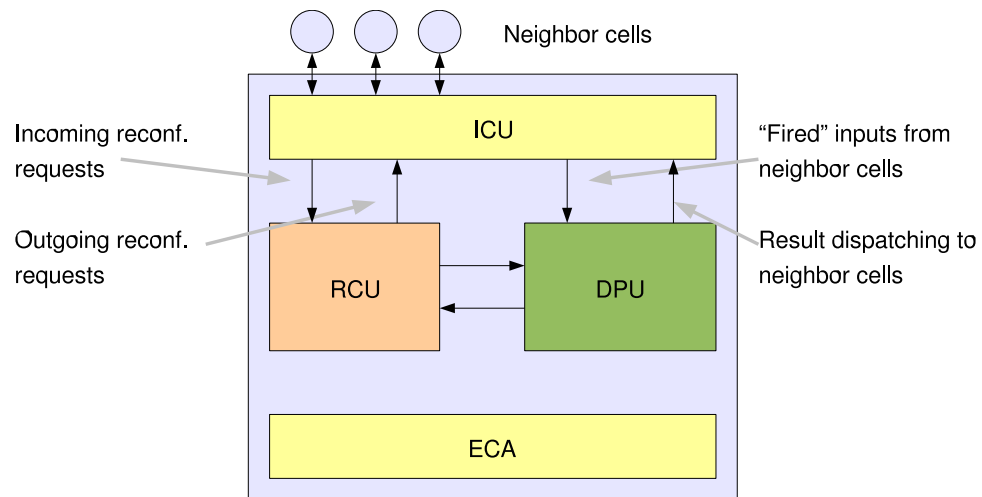


Figure 11. LCM main parts. The main logical parts of the LCM are ICU (Inter-Cell Connection Unit), DPU (Data Processing Unit), RCU (Reconfiguration Control Unit) and ECA (Environment Conditions Adaptation). The DPU is the reconfigurable part of the cell.

such networks. Inscalable LCM designs would require e.g. infinite memories inside the cells, or infinite number of neighbors.

The LCM operations can be coarsely divided to two groups; the data processing operations performed by DPU and the reconfiguration operations performed by RCU. Both units are fed by the ICU, which connects the cell to its neighbors. The separation of ICU (Inter-Cell Connections Unit) from RCU and DPU was done mainly to hide the cell topology from its functionality.

The DPU process the inputs in a data-flow manner. It means that no neighboring cells can directly change the DPU state, instead the DPU itself changes voluntarily its state according to the inputs in a predefined manner. It is of course possible to implement the DPU so that it pushes the inputs to the neighbors, but that may cause conflicts and it is not normally considered in different CA implementations.

The RCU instead operates differently. It is fundamentally mandatory that the RCU can force the selected neighbor to reconfiguration state, no matter how the neighbor was configured earlier. Thus the reconfiguration mechanism can suffer conflicts, i.e. two reconfiguration attempts are going to configure the same cell. If this could be a problem for massive systems or not, was not determined in this Thesis.

ECA (Environment Conditions Adaptation) is an abstract block, which adapts the cell operation to different kinds of environmental conditions, e.g. lack of energy, over-heating and so on. It is a purely abstract block; it can be used for simulating behaviour of the underlying hardware.

3.1.1 Symbols and Strings

The LCM was designed to be more general, so the model itself was designed to process abstract data symbols; the exact format of the symbol depends on the implementation. This allows later evaluation of the effects of different kinds of symbol sets.

During the development, the LCM was also decided to process sequences of symbols for helping the construction of data processing cellular structures shown later in this Thesis. The sequence is terminated by a special NIL-symbol, and this kind of NIL-terminated symbol sequence is called a string. Strings and string-based computation is described more detailly in the later sections.

For CVM, a 5-bit symbol was selected. It was divided to two 4-bit domains, one for data and other for control symbols, see Table 2.

Table 2. CVM symbol values, symbolic names and descriptions.

bin		descr.	bin		descr.
00000	0		10000	LS	List Separator
00001	1		10000	FS	Field Separator
00010	2		10010	SS	Structure Separator
00011	3		10011		
00100	4		10100		
00101	5		10100		
00110	6		10110		
00111	7		10111		
01000	8		11000		
01001	9		11000		
01010	A		11010		
01011	B		11011		
01100	C		11100		
01101	D		11100		
01110	E		11110		
01111	F		11111	NIL	String Terminator

The main reason for this selection was that a 4-bit data is easily represented as one hexadecimal character (0 - F). Values 0-15 represent hexadecimal data and 16-31 are used as control symbols. Value 31 is NIL symbol. One extra bit was added to allow the representation of NIL symbol, but that also meant that there was 15 extra symbols to be used as a playground.

Extra control symbols, like list, field and structure separators were used to simplify the implementation of reconfiguration. They may also be used for implementing list-processing capabilities, but it was not needed in the implementation of the CVM.

3.1.2 ICU, Inter-Cell Connection Unit

ICU is the layer exchanging data symbols between cells. For the DPU, the ICU (1) receives the evaluated results from selected cells, and (2) makes the result of DPU evaluation to be available for connected cells. For the RCU, the ICU (1) receives incoming reconfiguration requests from connected cells, and (2) dispatch the reconfiguration request from the RCU to the specified connection.

The differences between DPU and RCU inputs are that (a) RCU allows inputs from any neighbor, not depending on the previous configuration (RCU is not configurable), and (b) RCU allows the NIL-terminated reconfiguration stream from only one input at one time.

In this model, connection paths between neighbor cells can be unidirectional, that is there is no need to implement “neighbor-loops”, that is configuring two neighboring cells to read their results. But the direction of the connection must be configurable.

The minimum number of connections to form three dimensional structures is three (Figure 12). For reversible designs the minimum of four connections (2 inputs, 2 outputs) is required (Figure 13). As seen in the figures, by combining two three-input cells together it is possible to get one four-input cell allowing reversible binary operations.

In the CVM implementation, the array topology may be changed. The current implementation contains only a cubical three-dimensional topology of cells with six connections⁷. This was selected because it is one of the easiest three-dimensional topologies to understand for humans. In six-connection cubical topology the connections are symbolically named as *east*, *west*, *north*, *south*, *up* and *down*.

A special case for CVM is an abstract array. The model does not restrict the number of neighbors by itself, instead it is an implementation-specific variable. Thus it is possible to use the model with abstract topology, which does not set any restrictions for cell connections (i.e. all cells are neighbors to each other). This model gives the lower bound for execution time; routing the construction to physical device introduces overhead, which decreases the performance.

3.1.3 DPU, Data Processing Unit

The DPU is configurable three-state data-flow computing element. It contains input selection, input mode, transition function, result and an internal state storage. The internal architecture of DPU is shown in Figure 14. The term *operation* refers to a combination of input mode and transition function. Most of the functions are coupled with specific input mode and it is easier to refer that with one term.

⁷A three-dimensional projection of two-level four-dimensional cubical plane was also considered; two cubes would be interleaved inside of each other with one additional connection (total seven connections). The only reason to consider this was that it would make it possible to actually send bits to hyperspace.

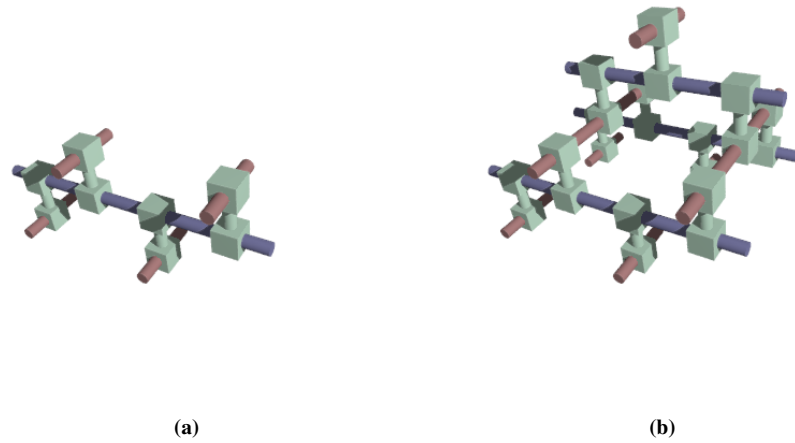


Figure 12. Three dimensional structure with three connections. A single chain of cells (a), and chains forming three-dimensional structure (b).

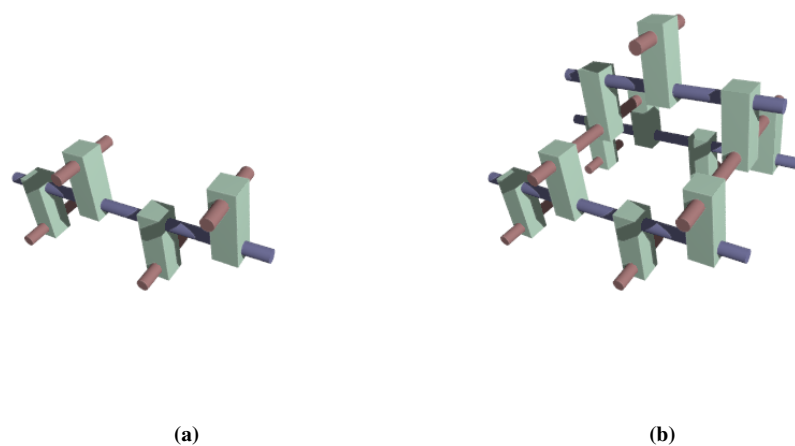


Figure 13. Three dimensional structure with four connections. A single chain of cells (a), and chains forming three-dimensional structure (b).

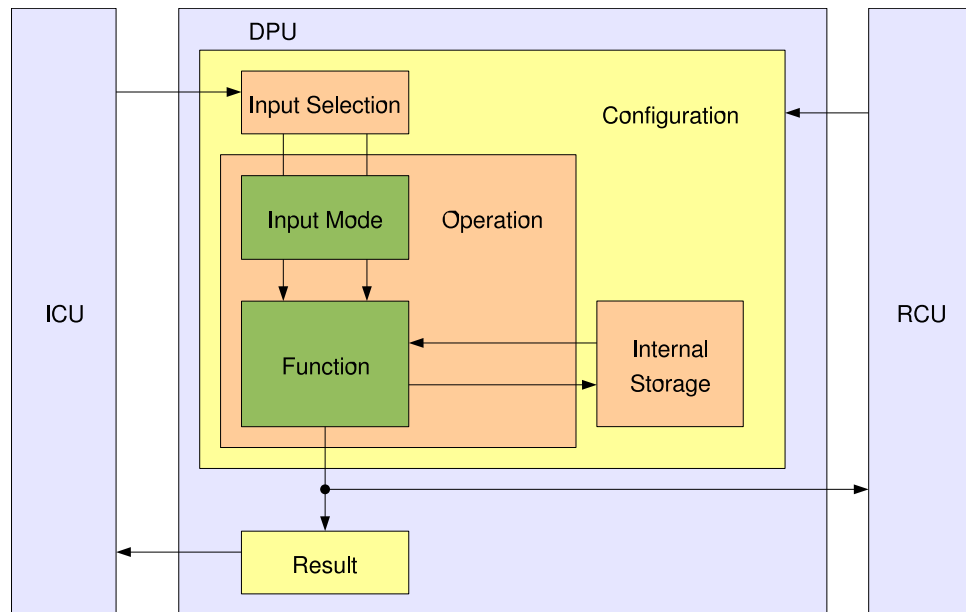


Figure 14. DPU architecture.

The transition function can also read and write the internal state storage, which allows an easy implementation of state machines needed for string processing. An alternative solution for this would be to use neighboring cell to store the state, and modifying the operation set so that state-aware functions take additional input and provide additional output to the cell storing the state of the processing, as shown in Figure 15.

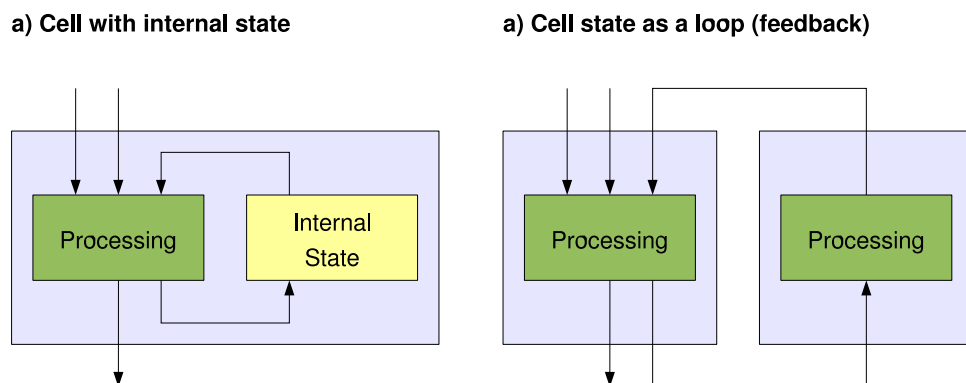


Figure 15. Implementing state-aware cells. There are two possibilities to make cells to state machines. (a) The cells have internal storage for storing the state, and (b) the state is outputted by the cell and it's fed back by its neighbor.

The internal storage capability is generally an implementation-specific variable. In the CVM implementation, it was selected to be one symbol, except in few operations described later.

The three operating states to support the asynchronous execution are *wait*, *exec* and *dispatch* (Figure 16). The DPU normally starts in the *wait* state. It remains in this state until the preprogrammed (configured) input conditions are met. When all necessary

inputs (operands) for processing are available i.e. the DPU is *triggered* or *fired*, the DPU acknowledges the inputs, thus allowing the input providers to continue with processing their next inputs; the cell itself transits to *exec* state. When the result is evaluated, the DPU transits to *dispatch* state for waiting the adjacent cells in the computation chain to acknowledge the result. It is also possible, that the evaluation result is discarded; this is used in conditional processing. In this case, the DPU transits directly back to *wait* state for receiving next inputs.

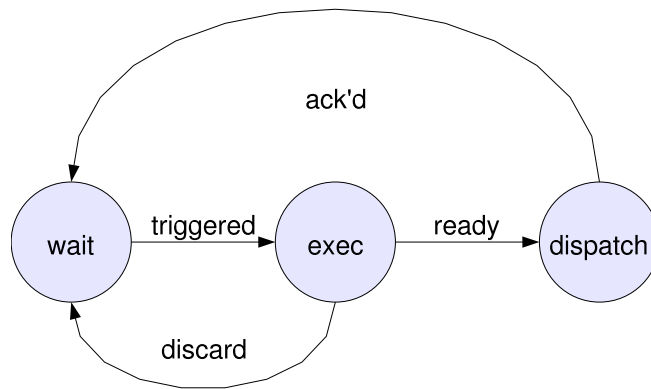


Figure 16. LCM state transition diagram.

The DPU was selected to generate one result as an output, i.e. the cell is *single-faced*. This selection was made mainly to simplify the result referencing in cellular structures, i.e. the reference to a cell is also a reference to the result. This selection was possible since it was decided not to stay strictly in reversible operations only. Two-input (binary) reversible operations would need to produce two outputs, i.e. the cell should be *double-faced*. There is generally no restrictions how many results the cell can produce; an example of a *multi-faced* cell design, exporting different results to different inter-cell connections is a CellMatrix (Macias, 1999). See Figure 17.

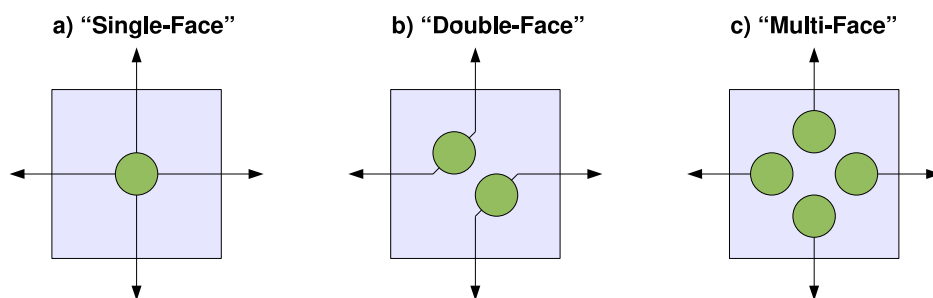


Figure 17. Cell faces. a) A single-face cell evaluates one result symbol, which is exported to all its neighbors. b) Double-faced cell evaluates two different result symbols from the same inputs. c) Multi-faced cell evaluates several result symbols from the same inputs.

There are several possible ways to define a complete set of operations for the LCM. In

the CVM implementation, a rich set of DPU operations were implemented for making the experiments easier. The experimented cellular structures were routed and placed manually because of the lack of automated tool for that purpose.

3.1.4 RCU, Reconfiguration Unit

RCU manages the reconfiguration cycle. The RCU is invoked in two ways; either an external reconfiguration request is received from ICU, or the DPU sends a request to dispatch reconfiguration (the `config` operation). When invoked by external request, the RCU receives the reconfiguration. This reconfiguration is then applied to DPU, after what the RCU continues routing the rest of the configuration to cell specified in the input stream. See Figure 18 for reconfiguration stream format and operation.

The main parts of the reconfiguration streams are (1) list of DPU configurations separated by a directional prefix, and (2) terminating marker, to transit the cells from reconfiguration state to operating state.

In the CVM implementation, the reconfiguration stream structure follows the selection of 5-bit symbols and DPU operation set, as shown in Figure 19. It contains three fields; operation, options and inputs. The fields are separated by a special FS symbol (FS, Field Separator) and the configuration is terminated by a special SS symbol (SS, Structure Separator). An empty configuration, containing only the terminating SS symbol, can also be given for by-passing cells that already contain specified configuration. This can be exploited when configuring the array in multiple phases. To allow the construction of the reconfiguration stream from smaller units, the `config` operation was defined so, that it discards any single NIL symbols; to terminate the stream, to adjacent NIL symbols are used.

The applicable protection mechanism is represented later.

3.1.5 ECA, Environment Conditions Adaptation

ECA (Environment Conditions Adaptation) is a logical entity to adapt the processing to the environmental conditions, namely to energy shortage and overheating. Its actual behavior is implementation-specific; depending on the underlying technology, the adaptation may be implicit, or the adaptation may require an explicit physical implementation, e.g. a sensor to determine over-heating for shutting down the DPU while cooling. The reactions to adapt to environment may vary.

In the CVM implementation, the ECA was not implemented at all.

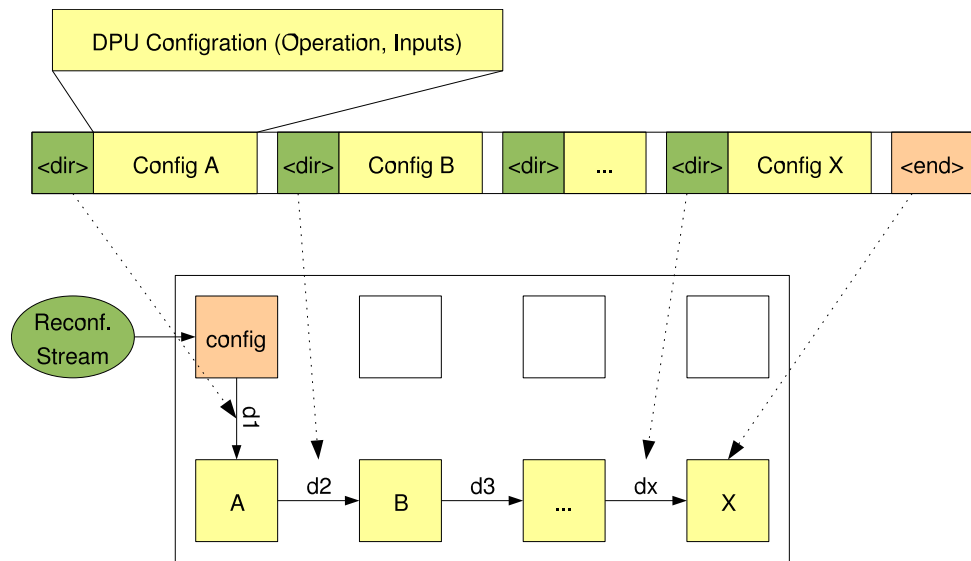


Figure 18. Reconfiguration stream format. The reconfiguration stream is a set of DPU configurations. A DPU configuration can also be empty; the DPU preserves its previous configuration and the cell acts only as a router of the string to other cells. Each DPU configuration is followed by a direction to pass the rest of the string, or NIL symbol for marking the end of the stream. To start the reconfiguration, the stream is fed to a cell, which is previously configured with `config` operation.

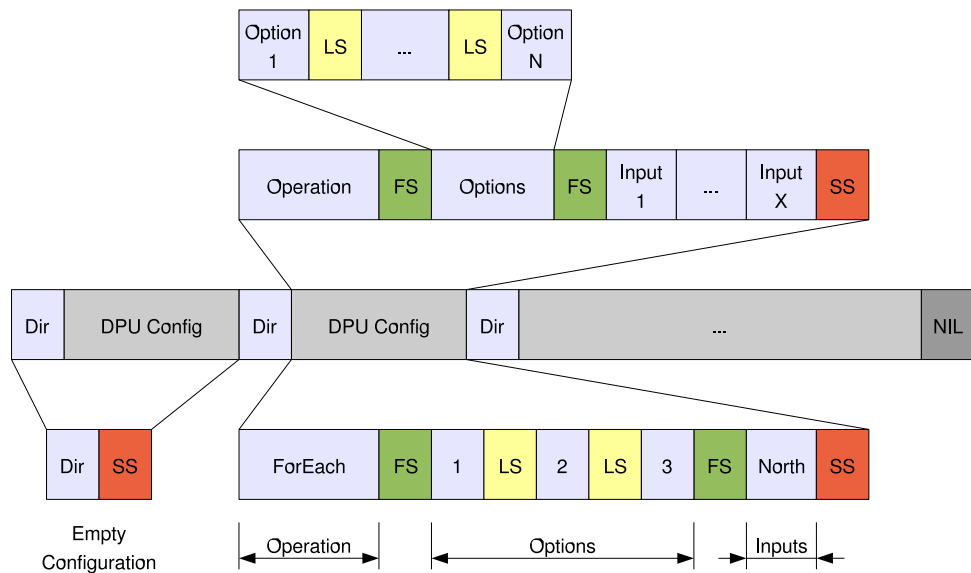


Figure 19. CVM reconfiguration stream format. The DPU configuration contains three fields; operation, options and inputs. The options part contains necessary presettings for specific operations. As an example, a configuration of `ForEach(1, 2, 3)` taking input from northern neighbor is represented.

3.2 Computation

The following sections are devoted to describe the computation model of the LCM array and how the CVM operation set was developed. The objectives of the design of the operation set were; (1) Turing completeness, and (2) easy to use.

A very common approach for designing a Turing-complete operation set⁸ for synchronous cellular automaton is to implement two-input NAND-gate with adequate signal routing capabilities; this approach is familiar to digital electric engineers as well. As described by (Sipper, 2004, p. 16 - 23);

“In order to prove that a two-dimensional CA is computation universal we proceed along the lines of similar works and implement the following components (Berlekamp et al., 1982; Nourai and Kashef, 1975; Banks, 1970; Codd, 1968; von Neumann, 1966):

1. Signals and signal pathways (wires). We must show how signals can be made to turn corners, to cross, and to fan out.
2. A functionally-complete set of logic gates. A set of operations is said to be *functionally complete* (or *universal*) if and only if every switching function can be expressed entirely by means of operations from this set (Kohavi, 1970). We shall use the *NAND* function (gate) for this purpose (this gate comprises a functionally-complete set and is used extensively in VLSI since transistor switches are inherently inverters, Millman and Grabel, 1987).
3. A clock that generates a stream of pulses at regular intervals.
4. Memory. Two types are discussed: finite and infinite.”

As an example, Sipper shows how a synchronous 10-rule, 2-state, 5-neighbor CA is proven to be computation universal by constructing the elements listed above, and gives a universal set for the automaton (Figure 20). The four different XOR functions are needed in two-dimensional CA for implementing signal crossings; three-dimensional CA does not require those.

As the LCM array forms an asynchronous CA, also several studies of rule sets of universal asynchronous cellular automata were studied. These include;

- Peper *et al.* (2003) introduces a universal set of operations for asynchronous circuits. Their approach is to use dual-rail encoding, and bidirectional signal paths.
- Encyclopedia of Delay Insensitive Systems (EDIS, 2007), containing several function designs for asynchronous (delay-insensitive) systems, as well as studies

⁸called “rule set” in CA

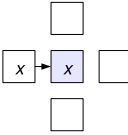
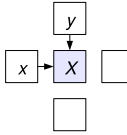
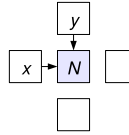
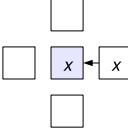
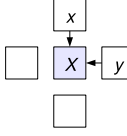
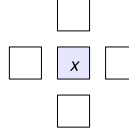
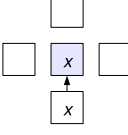
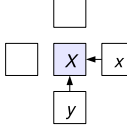
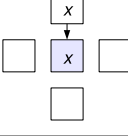
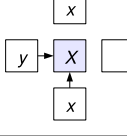
Propagation (wires)	XOR (signal crossing, 2D)	Functions
(1) Right propagation cell 	(5) XOR cell (type a) $(X = x \text{ XOR } y)$ 	(9) NAND cell $(N = x \text{ NAND } y)$ 
(2) Left propagation cell 	(6) XOR cell (type b) 	(10) No change (NC) cell 
(3) Up propagation cell 	(7) XOR cell (type c) 	
(4) Down propagation cell 	(8) XOR cell (type d) 	

Figure 20. 10-rule set for universal 2-state, 5-neighbor non-uniform CA. (cf. Sipser, 2004, p. 17)

of implementing delay-insensitive circuits and logic, e.g. Keller (1974); Ebergen (1987, 1991); Martin (1990); Patra (1996).

Especially the EDIS and Peper *et al* provide good starting point for designing a universal operation set for LCM.

Also, as the LCM array is a dataflow machine, the function set designs of data-flow machines, e.g. static token model described by Teifel & Manohar (2004a,b). Data-flow machines were found in general valuable source for designing data processing for cellular arrays.

3.2.1 Presentation

The computation structures are expressed as logical diagrams, which relation to physical mapping is shown in Figure 21. It is not necessary to draw the routes, have exact cell locations or present details of the functions. Furthermore, the details of the cell operations are normally left out, if they are not considered important. For example, an `zip` operation passes incoming streams in predefined order, but in the drawings the exact order is only seldomly shown. The schemantic drawings and cellular constructions have a strong relationship between each other, just like electronic circuit schematics and the physical circuitry itself (Figure 22).

The automated “*synthesis*” process would operate in reverse order; it would take a schmantic design as input, extend the macroconstructions, perform placement and create routes between the elements.

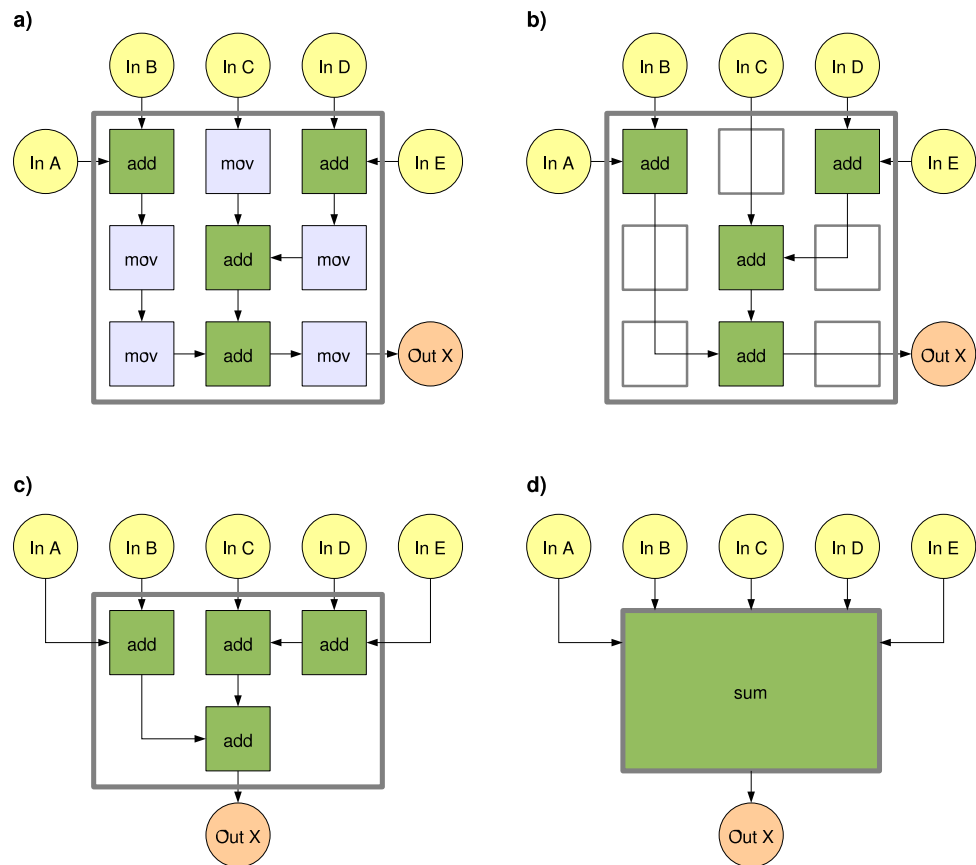


Figure 21. From physical structure to logical model. The first figure (a) shows a physical mapping of a 5-input sum function to a 3x3 cell matrix. For logical representations, first the routing cells can be removed (b). Then, the placement of cells can be altered (c) and finally, the entire function can be represented as one logical block (d). The synthesis tool operates in reverse order.

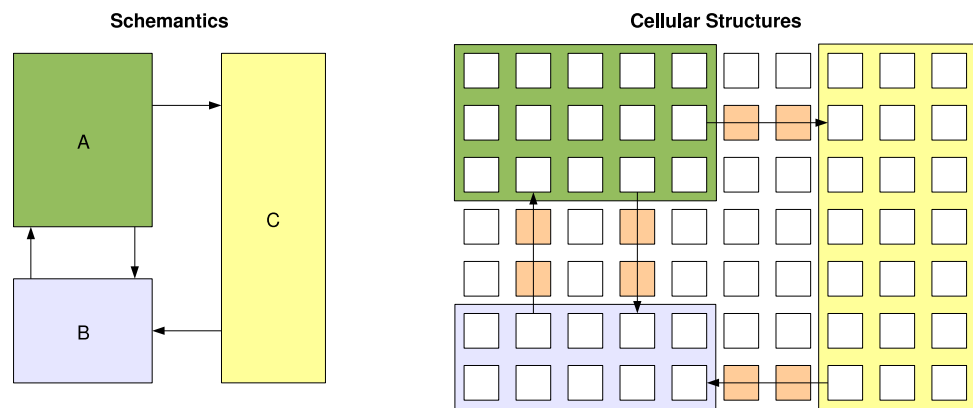


Figure 22. Schemantic drawings and cellular "circuitry".

For expressing the computation, it is not enough to show only cell structures. The data content should also be represented in the schemantic drawings; the cells are rectangulars and the data is represented as circles. In text, the data is written as;

- Symbols are normally written as they are; e.g. 1, 2, A, B.
- String is written in quotes; a “*ABCD*” means a stream of symbols A, B, C, D and NIL. The order of the stream is usually from left to right (the same order in what the text is read), but not always; e.g. the addition operation needs the values to be passed in *little-endian* form, i.e. the least significant number is received first. But it would be very confusing to write “...*The values 56 and 324 are added...*”, when it in fact means addition of 65 and 423. So, numbers are presented in natural order, that is, a number 123 is in fact a stream “*321*” (depending of course the width of the symbols, i.e. it may also be a binary stream).
- the stream is presented as a list of symbols or strings, separated by commas. Sometimes the stream is in parenthesis to make clear that it is a stream. The order of the stream is from left to right. A stream (“*AB*”, “*BA*”) means a symbol sequence “*A, B, NIL, B, A, NIL*”. A stream (A, B, B, A) is a symbol sequence A, B, B and A.
- A number stream (123, 456) is a symbol sequence 3, 2, 1, NIL, 6, 5, 4, NIL.

In the drawings, the order of the symbols depends on the direction of the stream. Look the Figure 23; it shows representation of string “*123*” in different directions. The order is determined by the arrow telling the direction of the connection (connections cannot be bidirectional in this model; that may be considered in some other paper). The random spreading of the symbols emphasize the asynchronous nature of the communication; it is normally not shown. Sometimes the data content of the cell is also represented; in those cases, the circle containing the symbol is drawn partially over the cell.

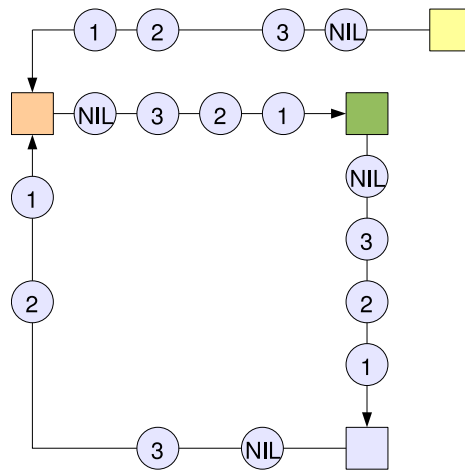
Operations are combinations of a evaluation function (computing the result from inputs) and an input mode. Input mode specifies how the operation handles the inputs; should they be all active (i.e. containing a value) before the function is executed, or is there some other rules for those. Operations are sometimes written to contain the arguments, like `move (a)` and `add (a, b)`. This helps referring to the inputs with a name.

A *head* refers to the first symbol in the string. A *tail* refers the rest of the string, i.e. symbols excluding the head (Figure 24). People familiar with LISP programming language will immediately be familiar with these terms.

3.2.2 Input Modes

The ICU (Intercell Connection Unit) has three different input modes, *synchronous*, *mix* and *zip*. In the synchronous input mode, the triggering happens when all the required inputs are available. This is the basic input mode for performing computations (Figure 25).

a) Representing symbol sequences



b) Representing cell contents

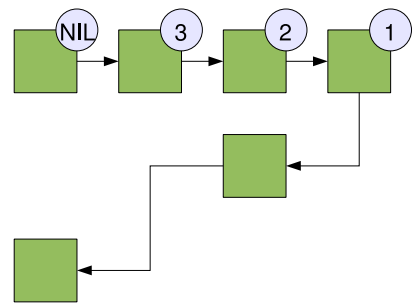


Figure 23. Streams (symbol sequences) in drawings. String “123” i.e. sequence (1, 2, 3, NIL) shown in a diagram. Notice, that the arrows show the direction of the stream.

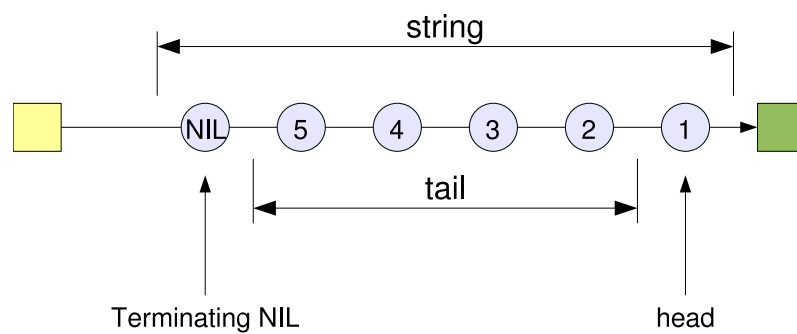


Figure 24. String, head, tail and terminal NIL.

The special behavior happens with NIL symbols; they are acknowledged only when all the required inputs contain NIL symbol for preserving the string synchronization (i.e. input of strings with different lengths); meanwhile, the NIL symbols are internally replaced with an configuration-specific value (usually zero).

The mixing input mode was designed to combine results of conditional processing blocks. Because it is not known from where the input is coming, the mixing input dispatches the strings in *First In, First Served* manner (Figure 26). Mixing input supports only one function (moving the input through unmodified). The implementation of mixing operation may require a fair scheduler (like Round Robin) to prevent dominance of high-traffic inputs.

Unlike the synchronous and mixing input modes, the zipping input mode is not a primary input mode. Instead, it could be replaced with a special function using synchronous input mode, or with a specially constructed cell blocks similar to ordering block of computation farm designs represented later in this chapter.

But because it was found useful in the constructions, it was included to the input modes. The zipping input mode *zips* several inputs together in predefined order (Figure 27). Thus it is useful for example for generating different kind of streams and to collect results from multiple processing blocks in the array.

3.2.3 Types of Operations

A coarse classification of operations are based on their input mode, number of inputs and purpose, as shown in Figure 28. The synchronous input mode is used for all regular data processing operations; the other two modes are used for special fan-in operations. Synchronous operations can be divided to separate classes using the number of inputs (none, one or two) they require for processing, and by the purpose of the operations. In the CVM implementation, some string-processing operations form a special group, since they are used as unary operations, although they are actually binary — the second controlling input is constant and predefined during the configuration.

There is only one nonary operation, *init*, which outputs a NIL symbol whenever it is configured. This is used for triggering the cellular constructions at the beginning. Only one *init* configuration would be needed for triggering the whole array; after the array is running, the configured structures can be triggered from the active parts.

Unary operations are splitted to several classes; (1) unary arithmetic, (2) stream reduction, (3) conditional passing, and (4) string processing operations. The two unary arithmetic operations are *move* and *not*.

Reduction operations reduce the entire string to one symbol, and these are used for condition processing. For example, *isz* operation outputs a single “1” or “0” symbol (terminated with NIL) whenever the string contained only zeroes or not.

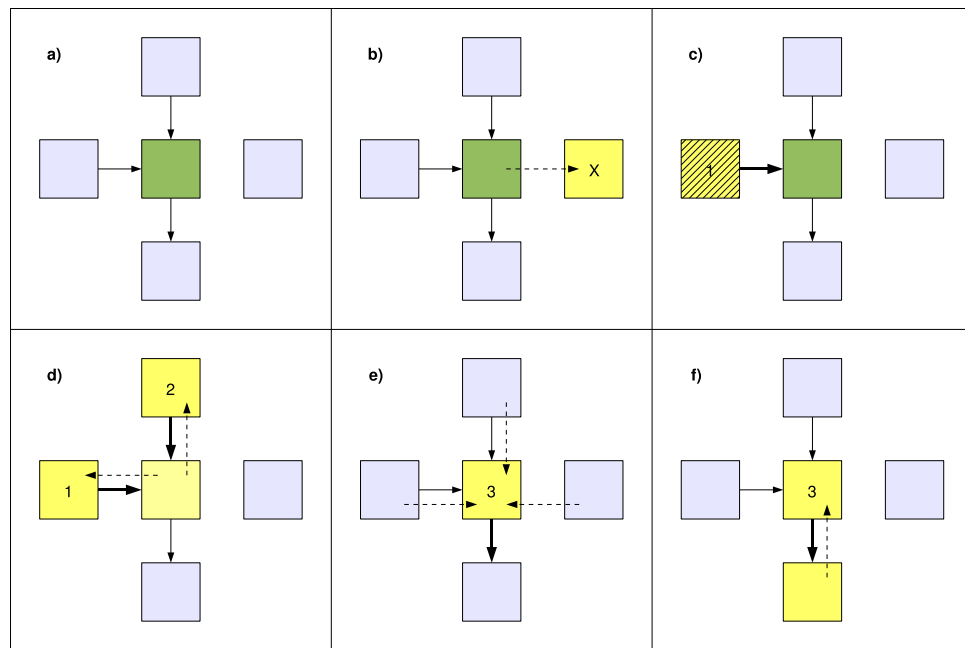


Figure 25. DPU Synchronous Input Mode. (a) The cell (in the center) is configured to use inputs from its left and up neighbors and its result is used by the cell below it. (b) If a neighbor cell which result is not needed by this cell evaluates a value, it is immediately acknowledged (dashed line). (c) When the left neighbor has evaluated a result, no actions are performed and thus the left neighbor remains in state. (d) When also the upper neighbor has evaluated a result, the input conditions are met and the inputs are acknowledged. (e) The cell has evaluated the result and gets immediate acknowledges from those neighbors, which do not need the computed value. (f) Finally, the cell below acknowledges the result and allows the cell to go waiting for new inputs.

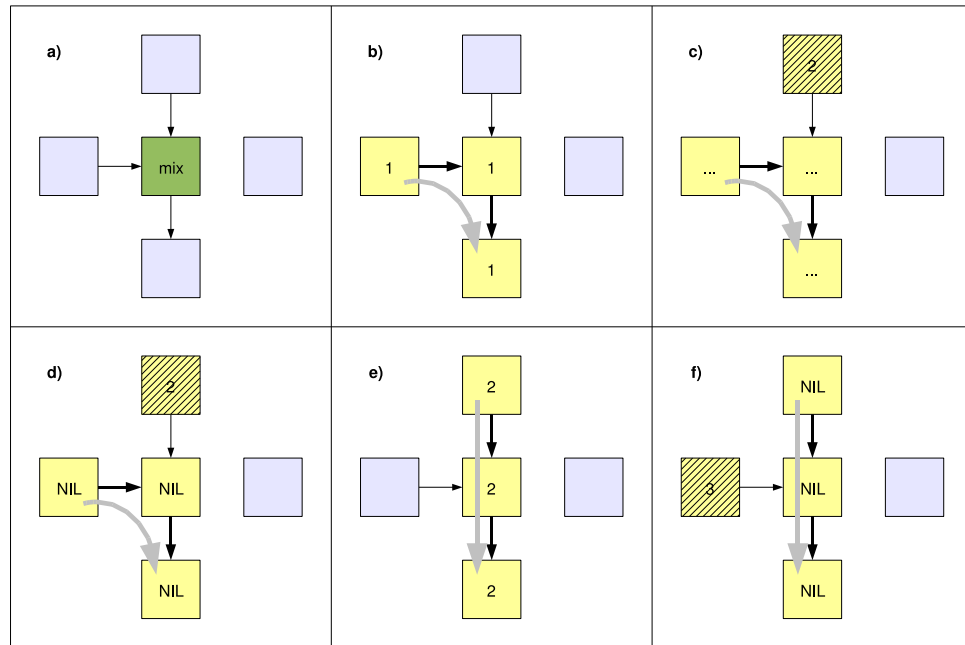


Figure 26. DPU Mixing Input Mode. (a) The cell (in the center) is configured to mix inputs from its left and up neighbors and its result is used by the cell below it. (b) The left neighbor activates, causing the mixing cell to move the results through from that cell. (c) During the *transfer*, the activation of the upper cell causes no actions, it stays waiting. (d) The NIL symbol terminates the transfer, causing the mixing cell to re-detect the inputs. (e) In this case, it selects the upper cell. (f) The values from upper cells are transferred until the NIL symbol is met.

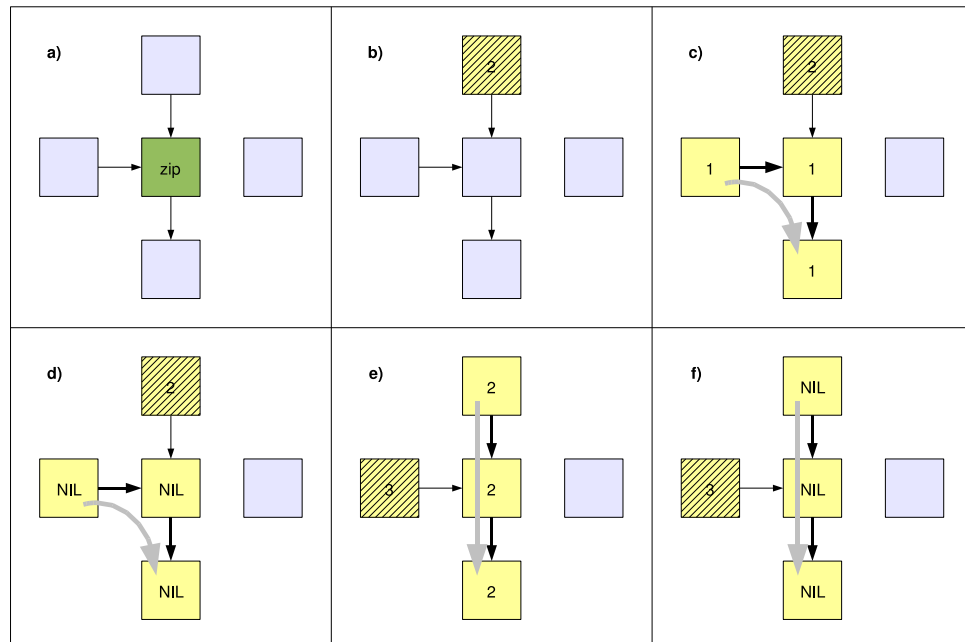


Figure 27. DPU Zipping Input Mode. (a) The cell (in the center) is configured to zip inputs from first its left and then its upper neighbors. (b) The activation of the upper neighbor causes no action, because the cell is configured to first pass the string from the left neighbor. (c) The activation of left cell causes the cell to pass the string to the terminating NIL (d). (e) After that, the string from upper cell is passed through to the terminating NIL (f), which causes the cell to restart the operation by passing the string from the left neighbor.

Input Mode	Number of operands	Purpose	Special
Synchronous	Nonary	Constant (init)	
	Unary	Unary arithmetic (move, not)	
		Reduction (isz, has(x))	
		Conditional passing (pass, block)	
		String processing (head, tail)	
	Binary	Binary arithmetic (add, sub, and)	String processing (pick, limit, foreach)
Zippping(*)	Variable	zip, join	
Mixing		mix	

Figure 28. Operation classes. Operations can be classified with their input mode, number of input and purpose. (*) Zippping input mode can be implemented in other ways.

Conditional passing operations are `pass(X)` and `block(X)`. Whenever the head of the string matches the configured symbol value, the string is either passed (`pass`) or blocked (`block`). The head symbol is removed from the stream, i.e. the operations have an implicit adjacent `tail` operation.

Basic string operations are `head` and `tail`, which pass either the first symbol (`head`) or everything except the first symbol (`tail`). Both operations include the terminating `NIL` to the resulting stream.

Binary arithmetic operations are regular binary functions like adding, subtracting and Boolean logic operations.

The special string processing class contains operations, which may be implemented with more primitive functions, but which were found frequently used in cellular structures synthesized for CVM. As an example, `pick` operation takes a controlling stream, and it passes through all symbols from input stream, where corresponding control stream value is non-zero. `foreach` is used for generating constant streams for inputs, and `limit` is used for equalizing the string lengths to certain values.

Mixing and zipping is used for collecting several data paths together. They are basically `move` operations with a special input mode.

3.2.4 Propagation, Fan Out and Fan In

Following the similar approach as Sipper (1999), at first it is shown the basic stream propagation, fan out and fan in, see Figure 29. Propagation of the streams is made with `move` operation. In CVM implementation, the fan out is implicit and free operation, i.e. from any cell, any number of neighbors can take the result as input. It is possible, that some actual hardware implementations does not allow making copies of the results freely; for those platforms, an explicit `clone` operation may be defined.

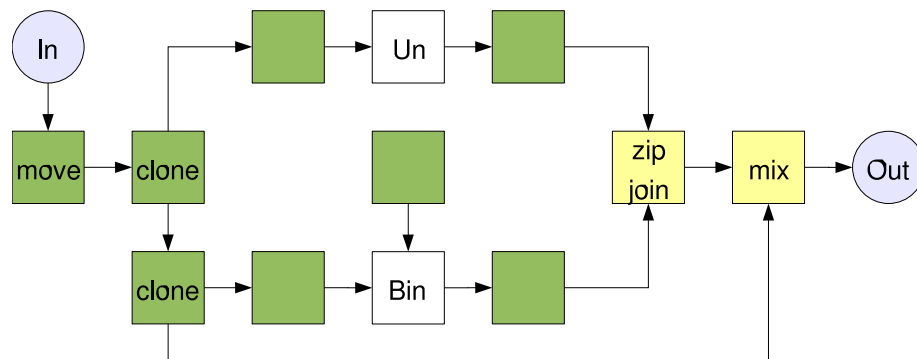


Figure 29. Propagation, fan out and fan in.

Unary operations can be added to anywhere in the propagation chain. Joining the different

data paths together is possible with either a binary operation, or using `zip`, `join` or `mix` operations, depending on the desired result.

For two-dimensional models, it is also necessary to have possibilities to implement signal crossing. This is not needed for three-dimensional models like LCM, since signals can easily cross each other in the third dimension.

3.2.5 Loops

Basically, a loop is a propagation chain with feedback. They are fundamental parts for many computational structures. For constructing a simple loop, a `move` operation is needed. `move` is a one input (unary) operation, which just copies the input as its result. If the `move` operations form a circle, data keeps circulating it forever (Figure 30)

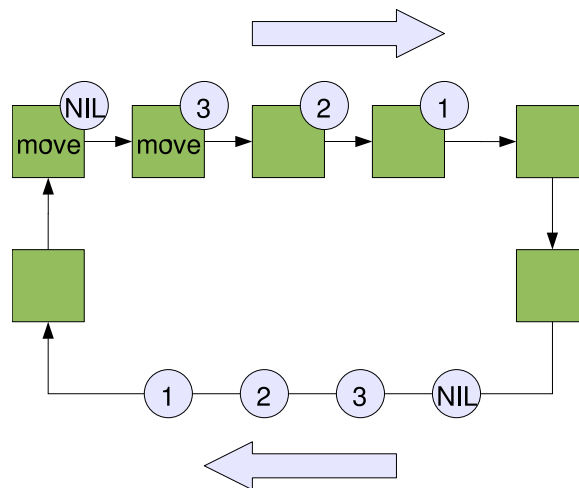


Figure 30. A simple loop. String “123” keeps circulating in a `move` loop forever.

Although the previous loop construction really works, it has one specific problem in the current model; there is no way to activate the loop during configuration. This decision⁹ was made to keep the model simpler. The only cell configuration, which activates immediately is a nonary (no input) operation `init` — whenever it is configured, it produces a `NIL` symbol, and waits it to be dispatched. After dispatching, it stays passive the rest of its life, because it has no inputs nor defined transitions to reactivate.

The initial `NIL` symbol sent by `init` operation is exploited to produce initial symbol sequences by using a `postfix` operation. `postfix` is unary function, which first passes the incoming string through it and then at the end of the string it appends a value specified as an option in the configuration, i.e. just before the `NIL` symbol (Figure 31).

Normally, there is no use to draw the full chains to the diagrams. Thus, the initial input chain (containing `init` chained with `postfix` operations) is represented just as

⁹Configuration stream does not contain cell activation information.

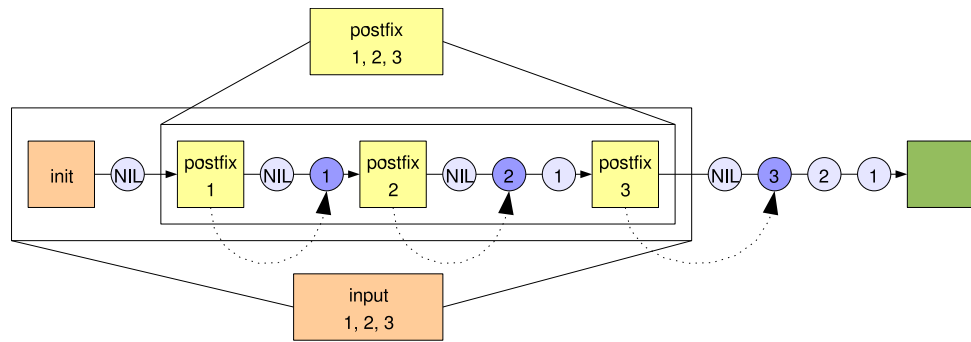


Figure 31. Operations `init` and `postfix`. `init` produces a `NIL` symbol after being configured. `postfix` appends a value, specified in the option fields in the reconfiguration stream, to the end of a string. The dashed arrows show, where the `postfix` operations have added the predefined value. Notice, that normally `postfix` operation chains are abbreviated to one block, and the combination of `init` and `postfix` operations to create initial input streams are abbreviated to `input` blocks.

one single `input` operation with arguments specifying the generated stream. Also, the `postfix` chains are normally abbreviated just to one postfixing block, with specification of the stream it produces.

Now it is possible to do two things; creating (1) initial sequences, and (2) initialized loops. Only thing which is needed is to combine these two constructions together. To do this, one more operation is needed, a `mix`. `mix` operation combines two streams together without preserving the order of the inputs. It means, that it serves streams in *First In, First Out* manner, but it also might need a more fair scheduler in high load environments (otherwise, it may keep one input passing, and the others are blocked to infinite). In the loop, `mix` combines the stream sending initial sequence and the feedback from loop (Figure 32).

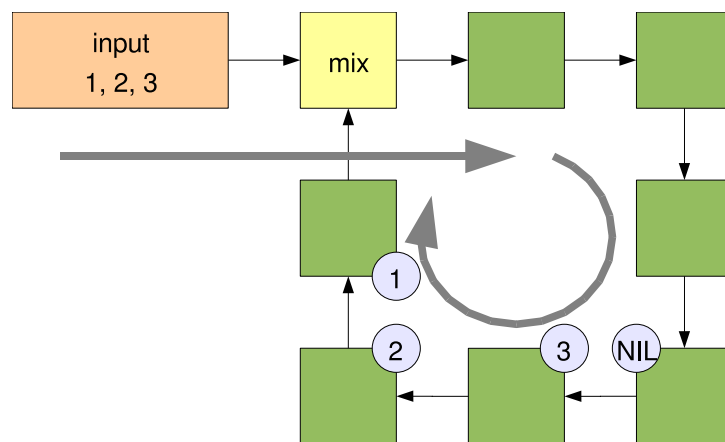


Figure 32. Initialized loop. Incoming stream generated by `input` block gets trapped to a loop. The `mix` operation is needed to join together two streams; the initial stream and the loop feedback.

3.2.6 Sequence Generator

It has shown four operations, which are important for constructing cellular circuitry; `move`, `input`, `postfix` and `mix`. As such, the loop is a sequence generator; it generates an endless stream of symbols circulating in a loop (Figure 33). Sequence generator loops (abbreviated to `loop` operation) are fundamental parts of many string processing structures.

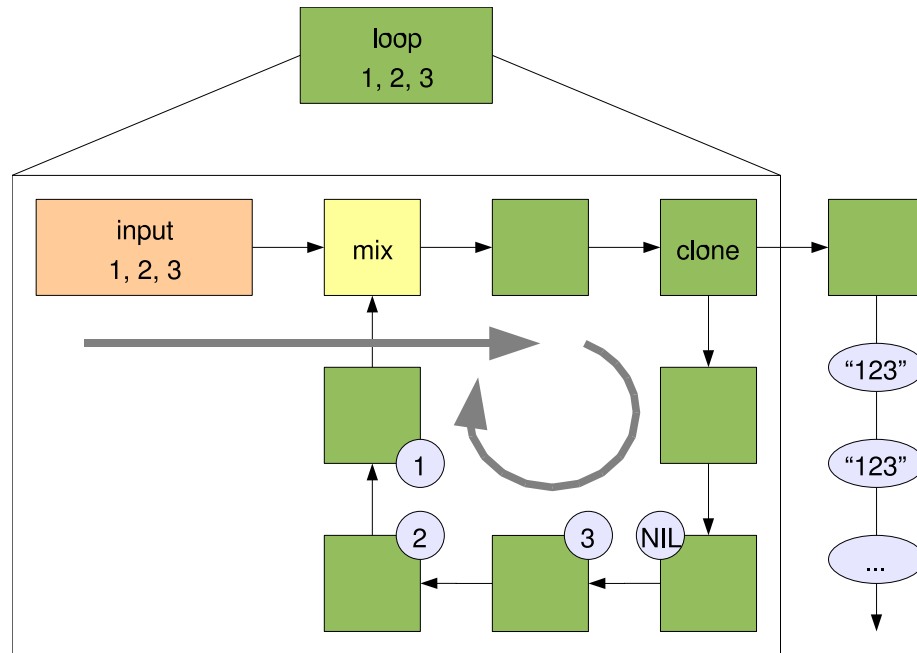


Figure 33. Loop as a sequence generator. Loops can generate infinite sequences of predefined strings.

For having better control when loop outputs the predefined sequence, a new operation to be included is introduced; `sync`. `sync` is a two-input (binary) variation of `move` operation. It is expressed as `sync (a, b)`; it passes the `a` stream through, when there are incoming stream in both inputs `a` and `b`. Placing this two-input operation for reading the loop gives a new interesting combined operation, `foreach`. This module will produce a predefined sequence as a response to any string in the second input (Figure 34).

The loop will be halted on the `clone` cell as long as the `sync` does not receive its second operand `b`. The cell stays in `dispatch` state as long as not all neighbor cells have acknowledged the input. The acknowledgement happens, when the cell is ready for performing the evaluation function, so `sync` acknowledges the value in the `clone` operation when it receives a symbol in the other input, too.

As described earlier, the cells with synchronous input mode reacts to terminating `NIL` symbols. Thus, `sync` passes through the string `"123"` only once, no matter if the other incoming stream is a plain `NIL` symbol or a stream of 10,000 symbols.

There is also a slightly easier way to construct a `foreach` operation (Figure 35). Instead

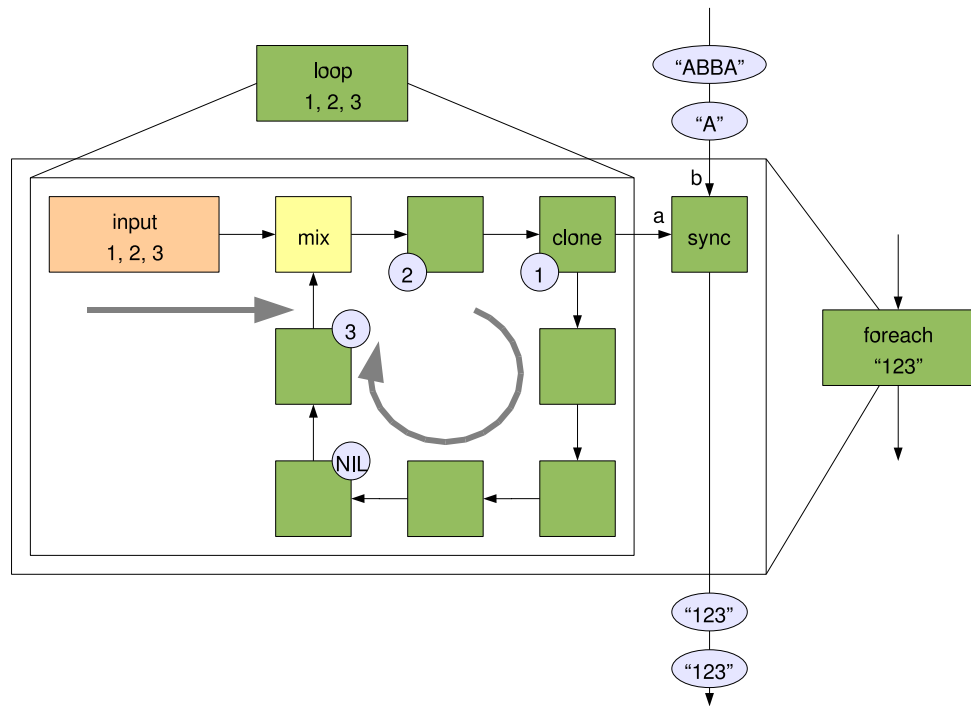


Figure 34. Loop and Operation `foreach`. A constant string is generated for any input.

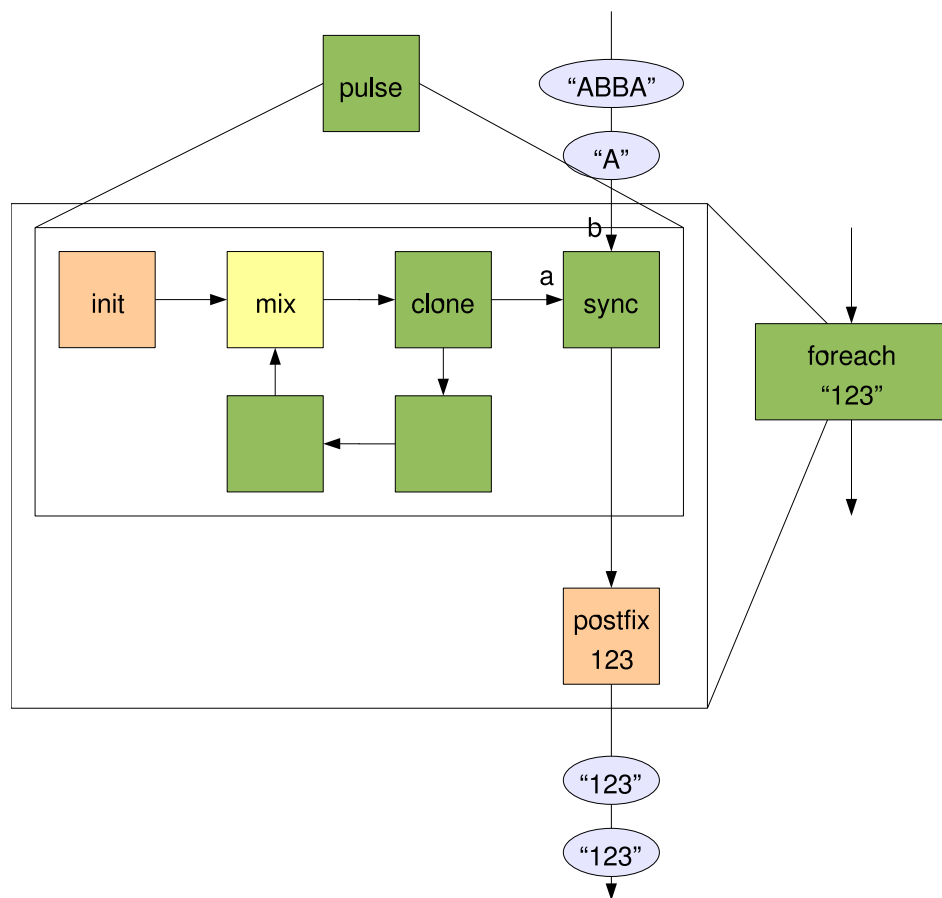


Figure 35. Variation of Operation `foreach`. Instead of looping the whole sequence, only the NIL symbol is looped.

of looping the whole sequence, only a NIL symbol need to be looped. This NIL symbol is initially generated by `init` operation. This NIL loop is guarded by a `sync` operation, and the guarded NIL loop is called a `pulse` operation — it produces a NIL pulse for every input string. By combining the `pulse` with a chain of `postfix` operations it is possible to use it for generating a predefined stream for any input.

3.2.7 Conditional Processing

For higher-level abstractions, conditional processing — sometimes referred as data-driven processing or execution — is one of the fundamental capabilities of the Turing-complete models.

In many data-flow models, the conditional processing is implemented with data-driven splitting. As an example, a demultiplexer used in digital electronic engineering is such a device; it takes in the data and control, and depending on the control, the data is passed to one of its multiple output lines.

This same behaviour could be implemented with LCM by having two-input conditional passing and/or blocking blocks, e.g. `passz` to pass the data, if the control stream is prefixed with a zero symbol. During the development of the cellular structures, another more flexible way was found. The conditional operation can be unary, and it would decide if it passes the string or not by examining the head of the string; if the condition is met, the tail of the string is passed. This way it is possible to combine multiple conditions with `join` operation, and send them to a one-input demultiplexing structure. This implementation allows easy construction of addressable blocks, which can be used for implementing random access memories to cellular structures.

Figure 36 shows a comparison of conditional processing in traditional data-flow models, with two-input conditional pass/block cells and with one-input conditional pass/block cells.

As an example, the Figure 37 shows a design of a block selecting a correct processing for data, depending on two conditional expressions A and B. The design contains the evaluation of conditions and merging them with the data, demultiplexing the data to correct processing block, processing and multiplexing the result to single output line.

In addition to conditional passing and blocking of the streams, there is a need for operations that can produce single-symbol true or false strings. In the CVM implementation, several different reduction operations were designed and tried out, e.g. `isz` (outputs “1”, if the input string is zero), `has (X)` (outputs “1”, if input string contains symbol X at least once) and so on.

As an example, Figure 38 shows evaluation if input string A is either equal, less than or greater than input string B. The `last` operation is a correctly configured block of `tail` or `pick` operations to retrieve only the last symbol from the stream; the `sub` operation

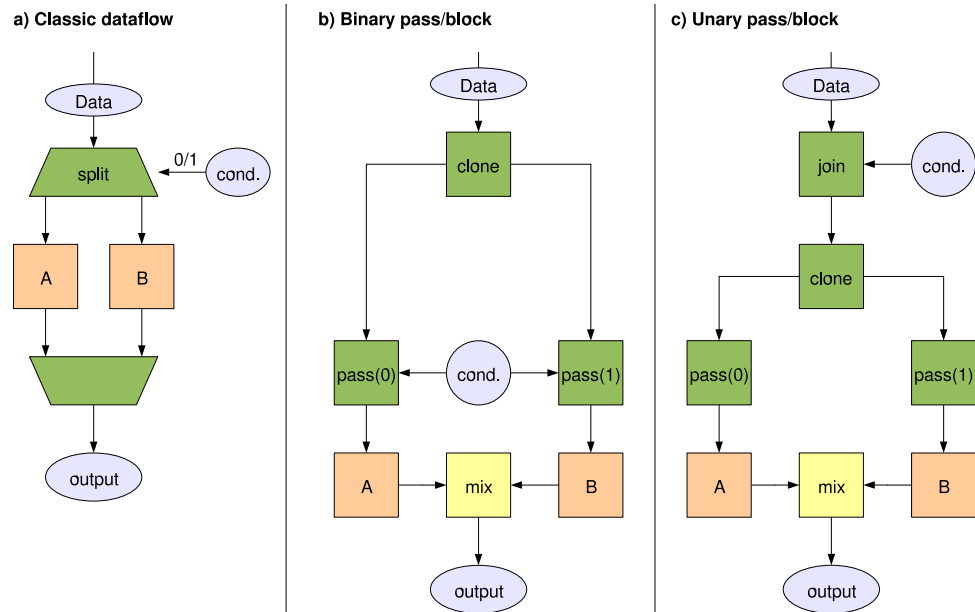


Figure 36. Comparison of conditional processing.

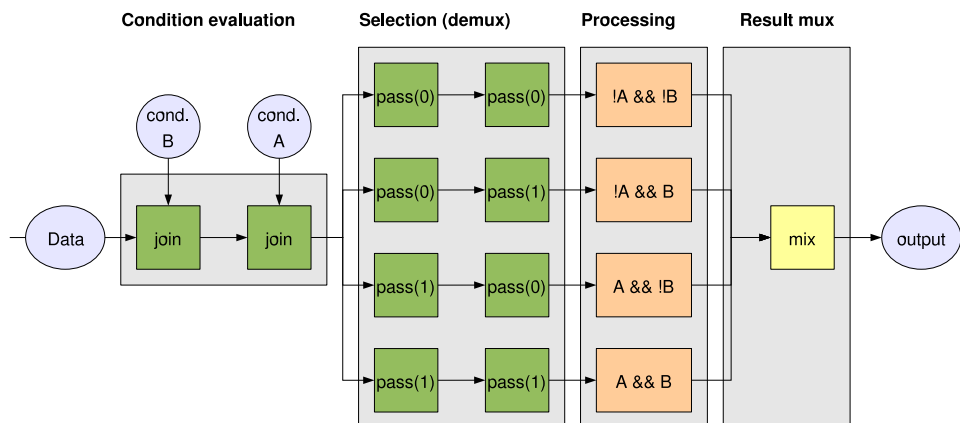


Figure 37. Conditional processing. Data is prefixed with the results of condition evaluation, then demultiplexed to correct processing and finally the result is multiplexed from the processing blocks.

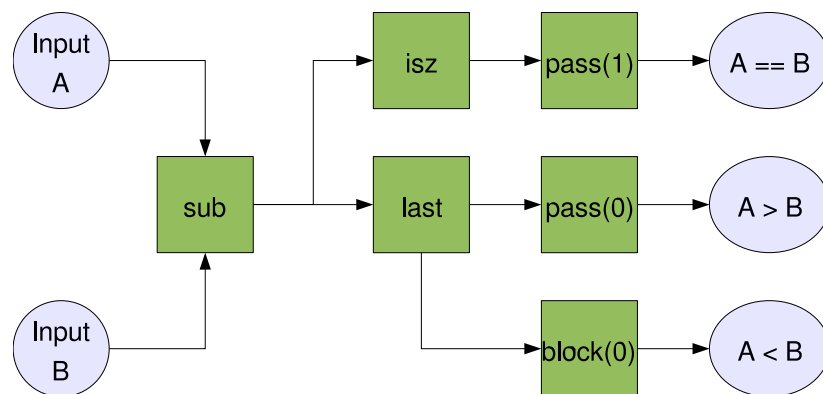


Figure 38. Evaluating conditions.

outputs the borrow (“Carry”) as the last symbol. The result of the block may be combined to common microprocessor condition flags Z (zero) and C (carry).

3.2.8 Arithmetics

The basic arithmetic functions in computers are adding and subtracting and following Boolean functions; NOT, AND, OR and XOR (exclusive OR). Furthermore, when the word size of the computer is more than one bit, they commonly have bitwise shifting functions (called SHR and SHL, shifting the bits to right or left). A single NAND is of course sufficient, since it can be used to implement all Boolean functions, and Boolean functions with bitwise shifting can be used to implement other arithmetic functions.

In the LCM model, since it is processing strings, all the implemented arithmetic functions are “string-wise”. That means basically two things;

1. Automatic string length equalization; The arithmetic operations are able to process strings with different lengths by automatically padding the shorter string, usually with zero symbol, and
2. Adding and subtracting are chained, i.e. they use internal carry flag to add or subtract long numbers. The carry is outputted as last symbol after the result of the operation, before terminating NIL symbol.

In CVM experiments, not all arithmetic operations were needed and thus they were not implemented.

3.2.9 String Manipulation

In addition to process symbols in the strings arithmetically, there is also need for extracting and combining strings in various ways, as seen in earlier examples.

The two basic extracting operations are `head` and `tail`. `head` outputs the first symbol in the string and prefixes it with terminating NIL symbol. The `tail` extracts the first symbol and passes the rest of the string through.

The special forms of extracting symbols from strings are `pick`, `remove` and `limit`. `pick` operation takes in the string to manipulate, and a control stream. Whenever the symbol in the control stream is non-zero, the symbol from the stream is passed through. `remove` works in opposite way, it passes through all symbols in the incoming stream, where the corresponding symbol in the control stream is zero. The `limit` operation is used for setting the lengths of the strings to predefined size; it contains a control stream, from which the symbols are used for padding shorter input strings. If the input string has more symbols than the control stream, the rest of the symbols are not passed through.

Strings can be combined either with `join` operation, or adding symbols to the end of the string with `postfix` operation. CVM implementation contains also `prefix` operation for easy addition of symbols to the beginning of the string — generally, both `prefix` and `postfix` could be implemented with `join` operation.

Both `zip` and `join` are N-input operations, which combine streams in predefined order. The order in which the inputs appear does not affect to the order in which they are outputted. The difference between these operations are, that `zip` preserves the intermediate NIL terminators in the streams, while `join` removes them, thus combining several strings together (Figure 39).

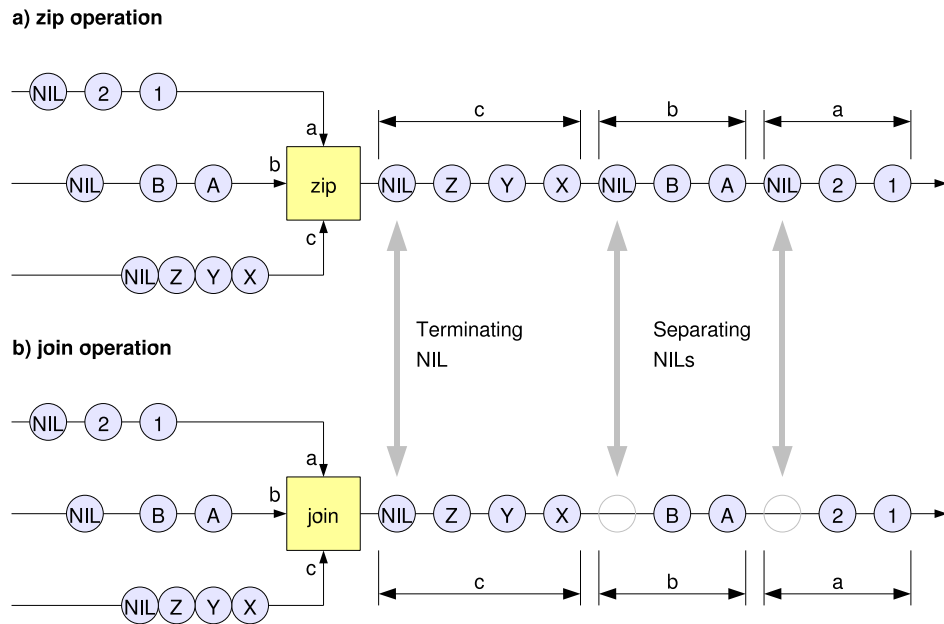


Figure 39. Zipping and joining streams.

3.3 Example Designs

In this section, two example cellular designs are introduced. The first one is an example of creating addressable blocks, which can be used for many purposes, for example to implement random access memories in cellular structures. The second one introduces designs of computing farms, which are an example of implementing parallel processing for the cellular array.

3.3.1 Addressable Blocks

One of the most important structure for information processing devices is a random access (addressable) memory. This can be used for many purposes, for example implementing

data storages and state machines. During the designing of addressable blocks, they were found usable also for implementing conditional processing (described earlier) and computing farms (described later), since the cellular devices does not make any difference, which kind of a block is attached behind the addressing bank.

The basic behaviour of an addressable block (see Figure 40) is that it takes in the key (address) followed with the content. As a result, it outputs the result of the processing behind the addressed block.

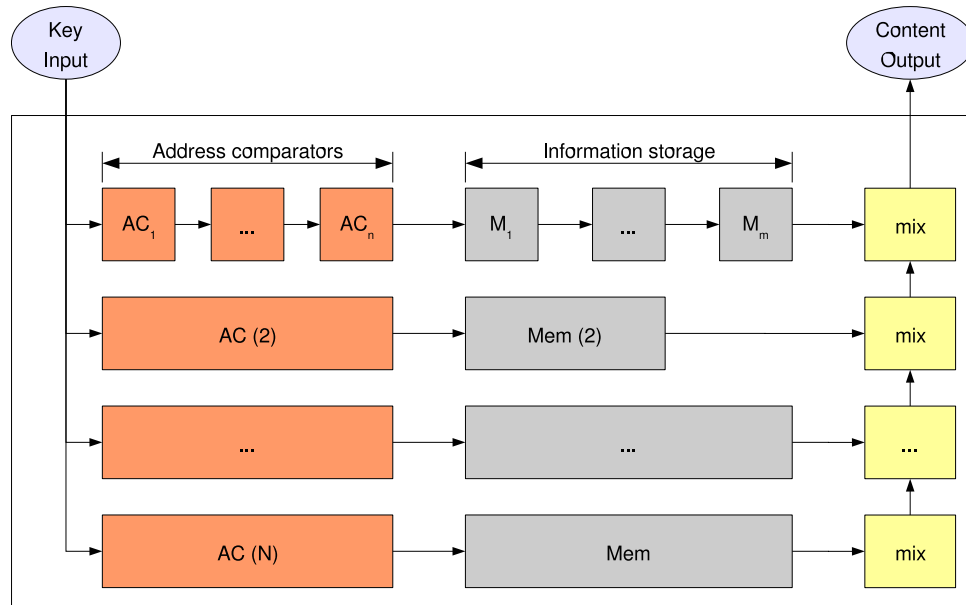


Figure 40. Memory block design for LCM virtual machine. The incoming address is spread to the address comparators. The memory blocks, which are behind the matching address comparators get active. The results are collected with `mix` operation. Memory and address comparator blocks does not need to have equal lengths.

The address comparators are effectively `pass` chains to compare the address, but in CVM implementation they were separated from passing blocks used for conditional expressions by naming them to `AddrCmp` operations; this way it was possible to separate the amount of cells needed to implement memories from other logic. The results from the memory blocks are combined with `mix` operations, since the memory is similar to a large conditional processing block.

For implementing the memory itself, there are two possibilities; `ROMCell` (actually a `postfix` operation) for read-only memories (ROM), and `RAMCell` for implementing read-write memories (RWM). The `RAMCell` operation was defined so that it uses the head of the string (first symbol) as a command. If the head is “0”, the `RAMCell` adds its content to the end of the string, operating in similar way to `postfix`. If the first symbol is “1”, the `RAMCell` takes the second symbol and stores it to itself. The command symbol is passed through the cell so that adjacent cells have the same command; in the write operation, the content stored to the cell itself is extracted from the stream.

Figure 41 shows the input and output streams for accessing ROM and RWM memories. The address is extracted from the stream by address comparators. All the memory accessing can be included with a freely formed transaction identifier, which can be used to separate the result of the operation to a correct processing device in multi-access memory designs.

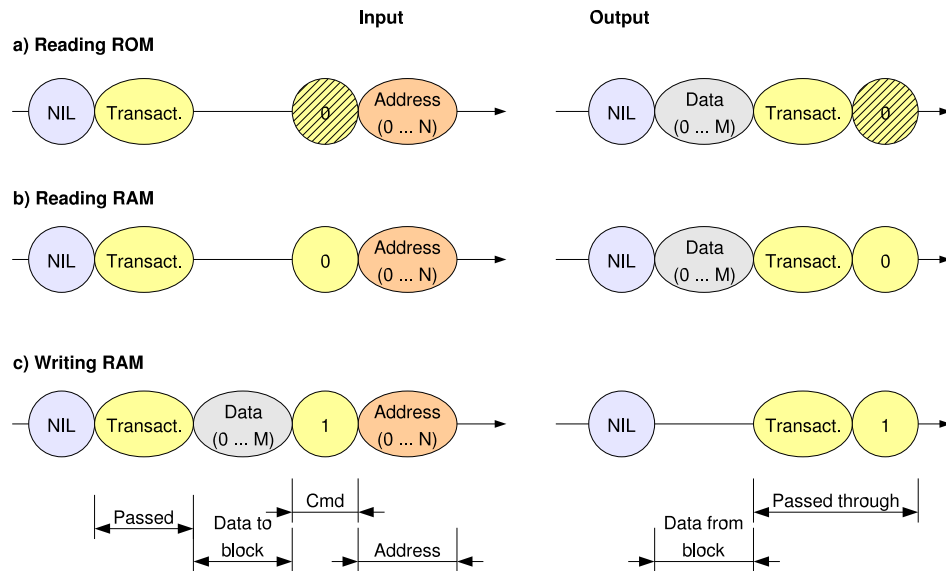


Figure 41. Memory accessing input and output.

RAMCell chains pass through the command and the transaction ID, and for reading operation they add the content of the block to the end of the stream. Basically it could be also possible to have no separate writable memory cells, but use read-only storages and reconfigure the new values to it. Writable memory cells were mainly defined to simplify the storage implementations in cellular designs for CVM.

The notable characteristics of this kind of memory design are;

- Only the addresses used are required to be synthesized, i.e. the address space can contain arbitrary “holes”. With a combination of the fact, that addresses do not need to have equal lengths (although in most cases that is desirable), this kind of structure could be useful for different kinds of associative memories.
- The content lengths do not need to have equal lengths. Although in most cases its desirable to have equal space for storing information, especially ROM could easily packed more thickly using variable-length storage structures.
- The access time is not uniform. The greater is the distance of the address input and the comparator block, the longer it takes the string to reach the comparator. Moreover, the longer is the distance between the memory block and the content output point, the longer it takes the content to reach that point.

Future designs may involve structures like cache memories, and associate memories.

These designs may discover some better alternatives for implementing memories to cellular machines.

3.3.2 Computation Farms

A computation farm is a group of identical processing devices sharing the same input of operands. The incoming operands are distributed to the processes and the results are combined in such way, that the order is preserved i.e. the firstly outputted result is the result of processing the firstly arrived input operands.

The basic computation farm design is shown in (Figure 42). The computation processes $P_1 \dots P_n$ are placed behind an address comparator block. The input operands are prefixed with an address, which is used to route the operands to one of the processing blocks. The sequence generator is used to produce an infinite sequence $1, 2, \dots, n$.

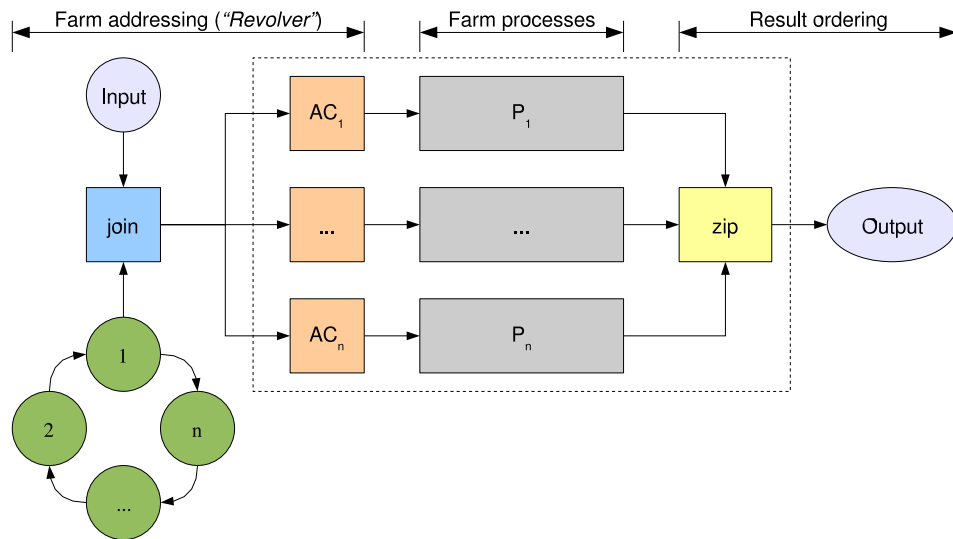


Figure 42. A computation farm design for LCM virtual machine. The inputs for the farm are prefixed with an address of a processing block to compute the result using `join` operation. The results are collected with `zip` operation, which preserves the order.

Since the inputs go to the processing blocks in predefined sequences, it is possible to collect the results using a block of `zip` operations. If the `zip` operations are placed hierarchially, there is a need to design the order of the zipping correctly, as shown in Figure 43.

The basic computation farm design can be slightly modified to make it load-balanced. A load-balanced farm design dispatchs the processing of inputs to a farm, that is not loaded at the moment when inputs arrive.

This can be achieved by replacing the fixed sequence generation with a stream containing the addresses of the processing blocks, as shown in the Figure 44. The key idea is to use

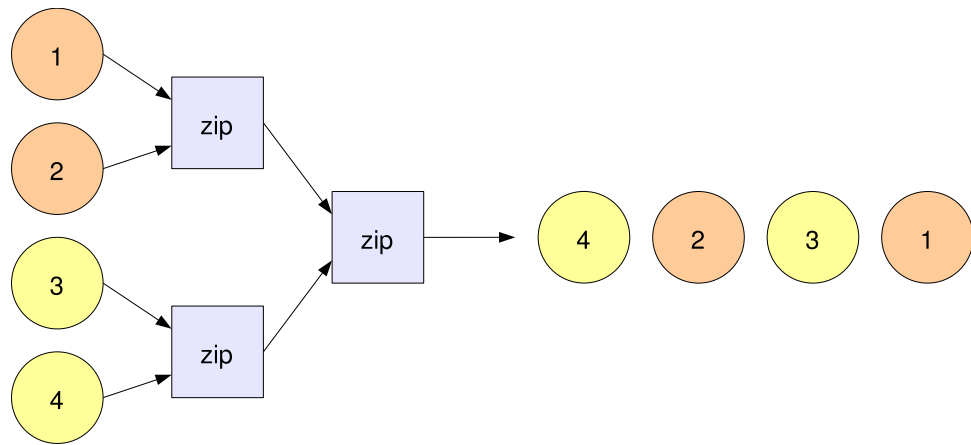


Figure 43. Result of hierarchial zipping. The two-level zipping in the picture does not result to a stream 1, 2, 3 and 4. Instead the sublevels gets zipped together. A three-level hierarchial zipping of inputs 0, 1, 2, 3, 4, 5, 6, 7 would output a stream 0, 4, 2, 6, 1, 5, 3, 7.

the same address value for selecting the free processing block, and to select the results in right order. After the process has processed the result, it returns its address to the addressing stream. In the figure, the addresses are multiplied three times, which means that the farm addressing queues a maximum of three inputs per processing block.

The Figure 45 shows the conceptual design of the ordering block behind the processing blocks in the load-balanced farm design. As an input, it receives a stream of addresses to processing blocks telling the correct order of results, that is the order of inputs fed to the processes. The `sync` operation is used to synchronize the streams so that the order can be preserved. The `sync` operation between the order stream and address comparators ensures that only one result is activated at time — when this result comes out from the `mix` block, it reactivates the `sync` operation to let the next address come in.

3.4 Reversibility

Although reversibility was scoped out from the LCM design, it is so important property in general for massive cellular arrays, that it deserves reconsiderations.

The fundamental property of a reversible function is that when given the function outputs, it is always possible to determine its inputs. Most of the operations described earlier are irreversible; the few exceptions are e.g. `move`, `not` and `postfix`. Normally, a reversible function has as many outputs as it has inputs. Thus, in the current LCM operation set design all binary functions are irreversible.

This may sound as it would be a laborous rework to make the LCM model reversible, but it is not necessarily that. To define a reversible operation set, there is no need to define it so that a single LCM is reversible; instead, the operations could be “paired”, so that together they are reversible. Whenever the underlying hardware supports reversible operations, the pairs are tried to be placed to the same cell, thus making the operation reversible.

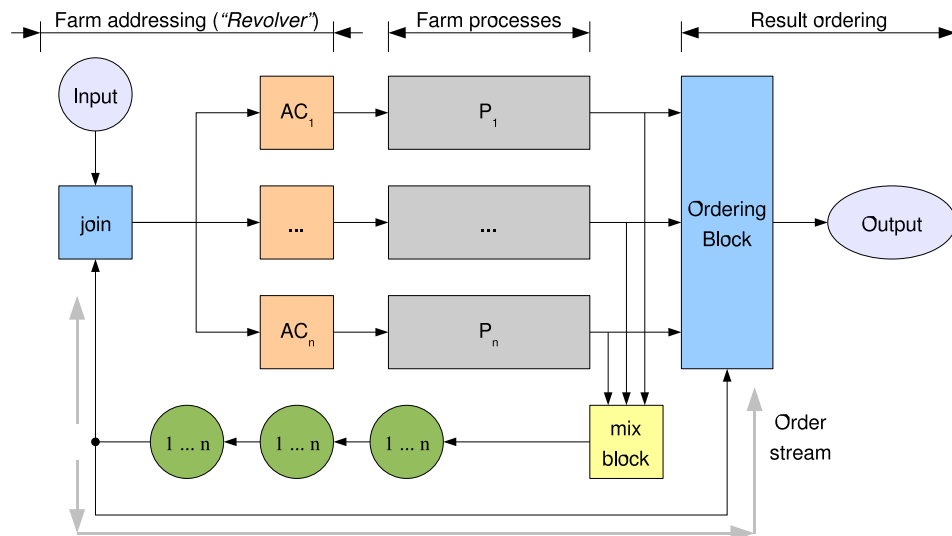


Figure 44. A load-balanced computation farm design. The cyclic ring of addresses is replaced with an address queue. Initially the queue can contain the same address multiple times, so the process is fed with inputs even if it is not completely processed the previous ones entirely. The process returns its address to the queue when completing the previous computation.

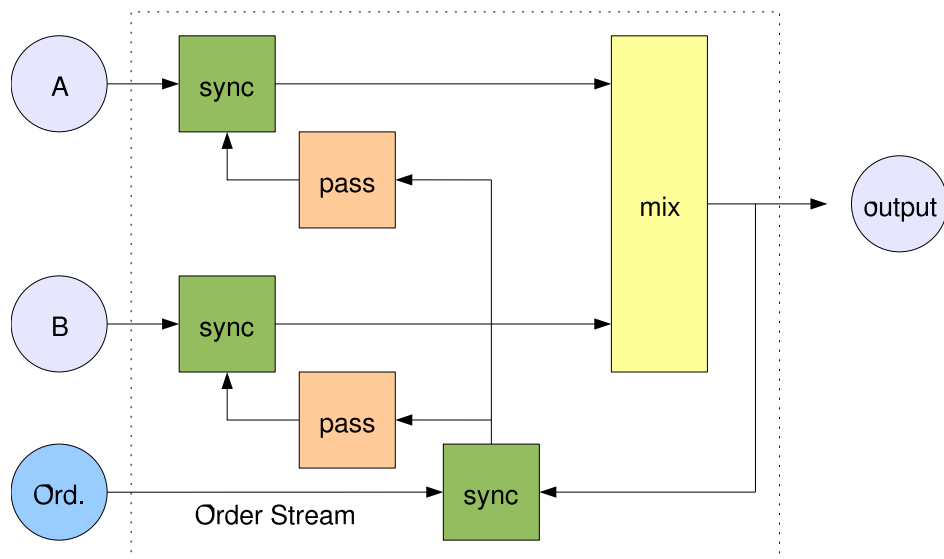


Figure 45. Conceptual design of an ordering block. The computed results from processes (A and B) are ordered according to the input of order stream. `sync` operation is used for synchronizing the streams to each other. Using `pass` blocks i.e. address comparators, the correct result can be picked to the result stream.

For example, consider operation pairs `head` and `tail`. Both of the operations are irreversible alone, but together they are reversible; the results (`head` and `tail`) could be joined together with `join` operation to form the original input to the `head` and `tail` operations.

Figure 46 shows the general idea. Instead of having two-output reversible splitting operation at logical level, there are two operations `head` and `tail`. When synthesizing the model, the automated routing and placement tool recognizes the possibility of mapping those two operations to one reversible cell at hardware layer.

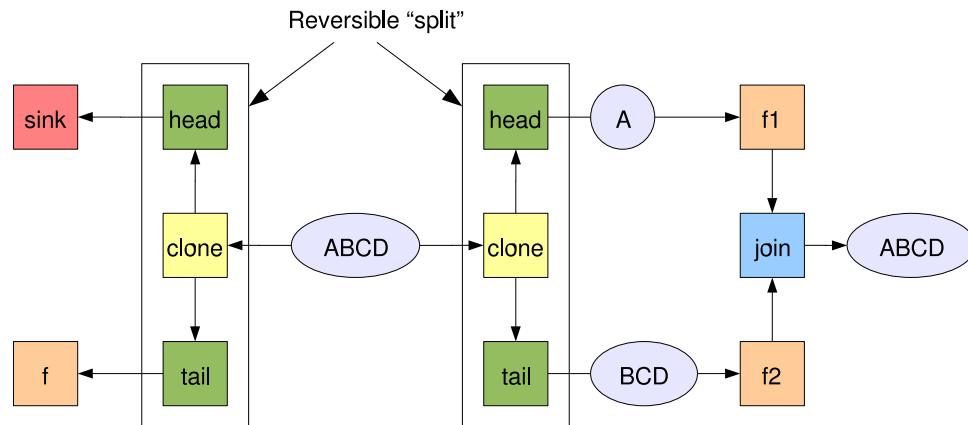


Figure 46. Mapping head and tail to reversible split.

The model could also include an implicitly placed irreversible `sink` operation. This would be used to destroy signals outputted by reversible logic gates that are not used in the synthesized configuration.

In general, this approach would need further studies. It places lots of responsibility to the automated tools for detecting the possibilities to use reversible gates, as well as defining the operation set so that there is a possibility to exploit reversibility. But at the same time, it releases the definition of the operation set to be strictly kept reversible.

4. Experiments

One of the main objective of this Thesis was to be able to experiment problem solving with cellular processing arrays. For making the experiments, a virtual machine (CVM, Cellular Virtual Machine) was implemented to execute the LCM array.

Three different cases were planned;

1. emulating a Turing machine,
2. implementing a self-replicating machine, and
3. solving eight queens' problem.

The first one, emulating a Turing machine, has clear objective — it shows a clear relationship between the Turing machine and the LCM array and thus makes it easier to determine the Turing completeness of the array. The second one, implementing a self-replicating machine, is a test for the constructional capabilities of the LCM array, mainly its reconfiguration engine. The last one, solving eight queens' problem, is a test for solving a simple but non-trivial computational problem with the LCM array.

The measures interested were;

1. The usage of cell resources of the constructions; both the volume of a box covering the construction and the number of cells used in the construction were interested in, as well as the ratio of different classes of cells (e.g. memory, routing, functions),
2. Total number of transactions, which is a coefficient of energy consumption of the computation,
3. Total number of cycles (i.e. consumption of time resource) consumed by the computation, and the level of parallelism (average number of simultaneous transactions during the computation),
4. The ratio of active cells per cycle compared to the total number of cells in the construction; as discussed in the chapter 2, the energy source is probably not capable of supporting arbitrary number of simultaneous transactions. Thus, one coefficient of the performance is the number of transactions required to complete the computational task,

The following sections describe the cellular structures created for conducting the experiments. Furthermore, the results from execution, as well as statistic information about the constructions are represented.

4.1 Cellular Virtual Machine

For experimenting the computations with the the array model, a virtual machine (CVM, Cellular Virtual Machine) was built. The first versions of the CVM were written with Python programming language, but for gaining better execution performance, the CVM was rewritten with D programming language (Digital Mars, 2007). D programming language is a relatively new derivation from C++ programming language and it resembles other newer C++ like languages, e.g. Java and C#. D is a compiled language, i.e. the source code is compiled to machine language and no virtual machines are used.

The CVM operates in two phases. In the first phase, the list of triggered cells is first randomly shuffled and then executed. It is also possible to limit the number of cells to execute, so with the combination of random execution order the CVM can simulate non-deterministic execution timings. When the triggered cell is executed, CVM buffers the results of the operations to intra-cell result buffer.

In the second phase, the results from the intra-cell buffers are distributed for the cells, which have requested them in their configuration. When distributing the results, it is also checked if the cell can be triggered. If so, it is added to the list of active cells.

The cell operations in the CVM are ordinary class methods. The cell's result buffer also removes the need of writing of complex state machines for operations, which may emit more than one symbol when processing inputs.

With some operations a freedom of implementing arbitrary large data storages inside the cells was taken to ease the manual implementation of cellular structures. These relaxations are;

- The prompts of output operations (e.g. `outx`) used for debugging are stored inside the CVM, without requiring any storage capabilities from the cellular structure.
- File input operations (e.g. `filein`, `configin`); these operations read the entire file and process it during the virtual machine startup.
- So called *repeater* operations (e.g. `foreach`, `pick`, `remove`); these operations have two inputs; the first one is the actual stream and the second one is a repeating control stream. Normally, these control streams would be implemented as sequence generators, but to ease the manual design of the constructions these streams were implemented inside of the single cell. Before making this, the construction of a real control stream was first experimented to make sure that the operations would work right.

- Operations with constant data, like `input` and `postfix`; instead of requiring to build the actual sequence generating chain, these operations store the constant stream inside of them.

Also, a special `buffer` operation was implemented, so that the needed stream buffering can be done with single cell without manually placing a sequence of `move` operations. More details about CVM can be found from Appendix A.

4.2 Turing Machine Emulation

The first experiment made with CVM was a Turing machine emulation. For validating the correct behaviour of the emulation, there was a need to select a computational task to execute with a Turing machine. The choice was to implement a so called *Busy Beaver 501*, later abbreviated as BB501.

Busy beavers are Turing machines, which start with an empty tape, do a lots of work and eventually halt. The busy beaver game as defined by Radó (1962), asks to construct simple Turing machines which produce a maximum number of ones on their tape before halting. These Turing machines have N states, one of which is the initial state and one is anonymous halt state, a tape alphabet with two symbols named 0 and 1, an initial tape with all 0 symbols, and with each step print a new symbol and move their head either left or right.

Before the BB501 was implemented on CVM, a few types of five-state Busy Beavers were written with D programming language to conventional PC for examining their behaviour;

1. BB501, found by Uwe Schult in January 1983, published by Ludewig *et al.* (1983)
2. BB1915, found by George Uhing in December 1984, published by Dewdney (1985)
3. BB4098, found by Heiner Marxen and Jürgen Buntrock in September 1989, published by Marxen & Buntrock (1990)

The Table 3 shows statistical information of the three simple 5-state, 2-symbol busy beavers. The busy beavers are named after the number of non-blanks written to tape, e.g. busy beaver 501 writes 501 non-blank symbols and then halts. The state transition tables of the mentioned busy beavers are shown in Table 4.

Table 3. Statistical information of three 5-state, 2-symbol busy beavers.

Busy Beaver	Non-blanks written	Tape usage	Transitions
BB501	501	668	134,468
BB1915	1,915	3,825	2,133,493
BB4098	4,098	12,289	47,176,871

Table 4. State transition tables of three 5-state, 2-symbol Busy Beavers. Mapping of {state, symbol} pairs to {next state, written symbol, movement}.

BB501	BB1915	BB4098
$\{1, 0\} \rightarrow \{2, 1, L\}$	$\{1, 0\} \rightarrow \{2, 1, R\}$	$\{1, 0\} \rightarrow \{2, 1, L\}$
$\{1, 1\} \rightarrow \{3, 0, R\}$	$\{1, 1\} \rightarrow \{3, 1, L\}$	$\{1, 1\} \rightarrow \{3, 1, R\}$
$\{2, 0\} \rightarrow \{3, 1, L\}$	$\{2, 0\} \rightarrow \{1, 0, L\}$	$\{2, 0\} \rightarrow \{3, 1, L\}$
$\{2, 1\} \rightarrow \{4, 1, L\}$	$\{2, 1\} \rightarrow \{4, 0, L\}$	$\{2, 1\} \rightarrow \{2, 1, L\}$
$\{3, 0\} \rightarrow \{1, 1, R\}$	$\{3, 0\} \rightarrow \{1, 1, L\}$	$\{3, 0\} \rightarrow \{4, 1, L\}$
$\{3, 1\} \rightarrow \{2, 0, L\}$	$\{3, 1\} \rightarrow \{H, 1, L\}$	$\{3, 1\} \rightarrow \{5, 0, R\}$
$\{4, 0\} \rightarrow \{5, 0, L\}$	$\{4, 0\} \rightarrow \{2, 1, L\}$	$\{4, 0\} \rightarrow \{1, 1, R\}$
$\{4, 1\} \rightarrow \{H, 1, L\}$	$\{4, 1\} \rightarrow \{5, 1, R\}$	$\{4, 1\} \rightarrow \{4, 1, R\}$
$\{5, 0\} \rightarrow \{3, 1, R\}$	$\{5, 0\} \rightarrow \{4, 0, R\}$	$\{5, 0\} \rightarrow \{H, 1, L\}$
$\{5, 1\} \rightarrow \{1, 1, L\}$	$\{5, 1\} \rightarrow \{2, 0, R\}$	$\{5, 1\} \rightarrow \{1, 0, R\}$

4.2.1 Turing Machine for CVM

The Turing machine (TM) emulator for CVM follows the two-stack implementation of a Turing machine in Conway’s Game of Life by Randall (2001). This implementation uses two stacks to emulate tape movement, as seen in Figure 47. When the head is instructed to go left, it pushes the symbol (acquired from state transition table) to the right stack and pops a new symbol from left stack. The movement to the right does the opposite, i.e. pushes the symbol to left stack and pops a new symbol from right stack. The stack was designed so that popping a value from empty stack, i.e. reading the value from the end of the tape, returns a blank symbol.

The overview of the TM emulation is shown in Figure 48, and the CVM visualizations are shown in Figure 49. The address to the state transition table is formed by joining the previous state and the symbol from the tape together. This address is then used for fetching the next state and command (abbreviated cmd) from table. The command contains the movement (left or right) and the symbol to write to the tape. The block named “dispatch” in the figure selects the stack to write the symbol to. When the stack has pushed the symbol, it returns a signal, which is then sent to the opposite stack to instruct it to pop a symbol. The pop’d symbol is then sent to the transition table address generator to be combined with last state.

The Figure 50 shows the method used for addressing the stack. The position in the stack is selected with a sequence of 1’s marking the correct place. When retrieving a value

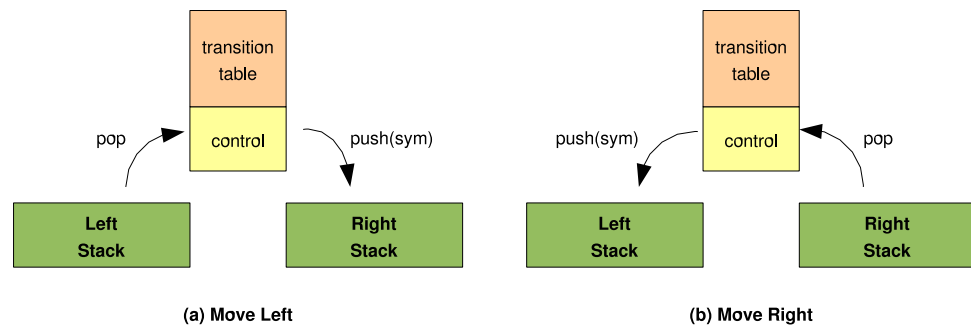


Figure 47. Simulating head movement with two stack.

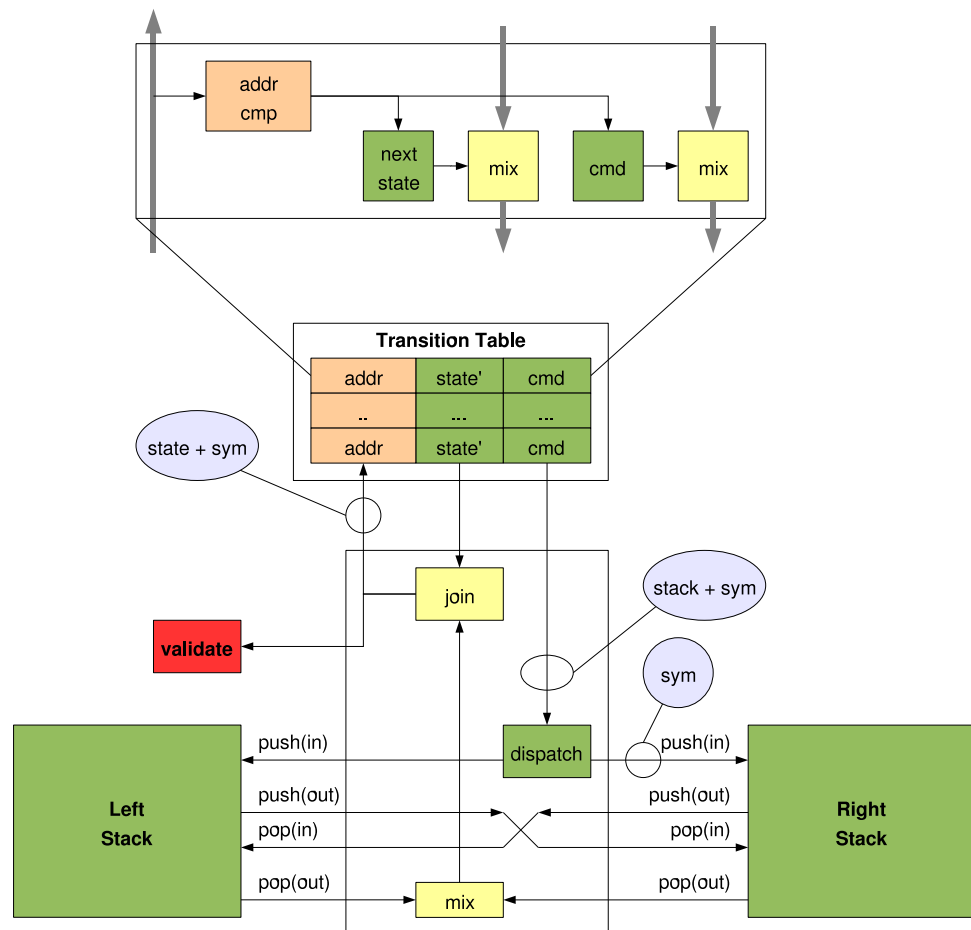
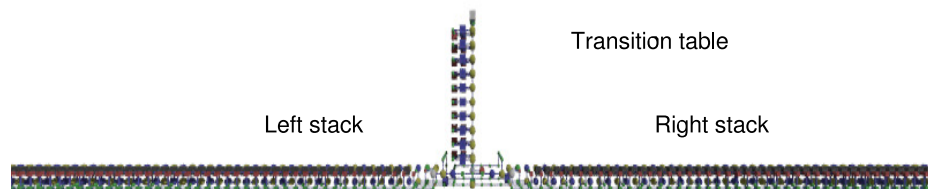
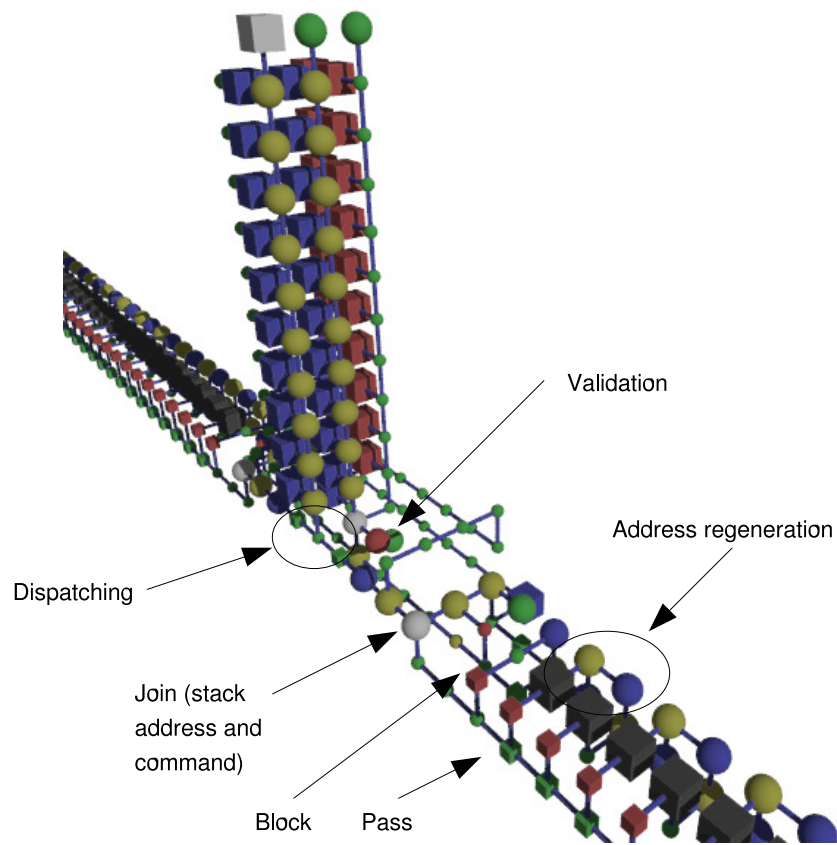


Figure 48. Overview of Turing machine implementation.



(a)



(b)

Figure 49. Visualization of CVM TM. (a) A distant view, (b) a close view to control and transition table.

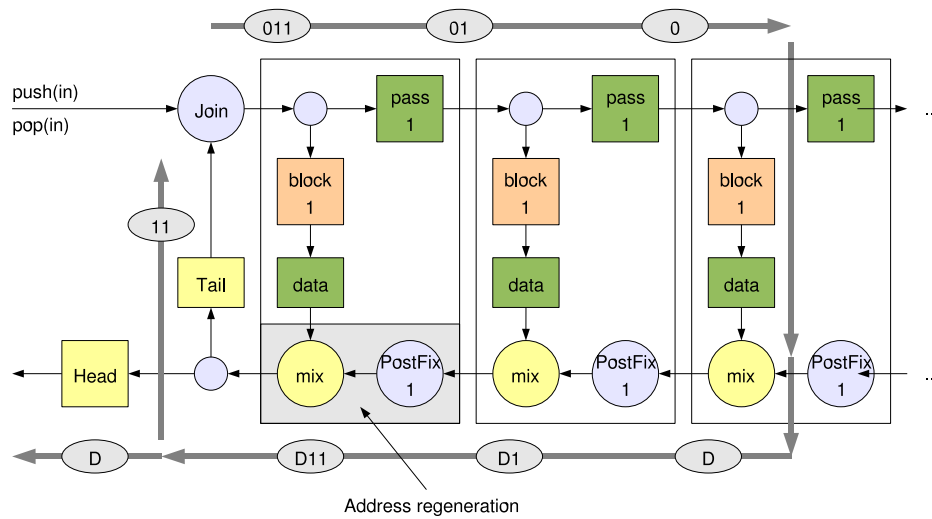


Figure 50. Turing machine stack addressing.

from stack (`pop`), one “1” is removed from sequence. When pushing a value to stack, one “1” is added to the sequence. Since each comparison stage (pairs of `block` and `pass` operations) shortens the sequence, the address is regenerated with `postfix` operations when returning the value. This way the address is stored in the tape itself, so the tape length can be freely increased.

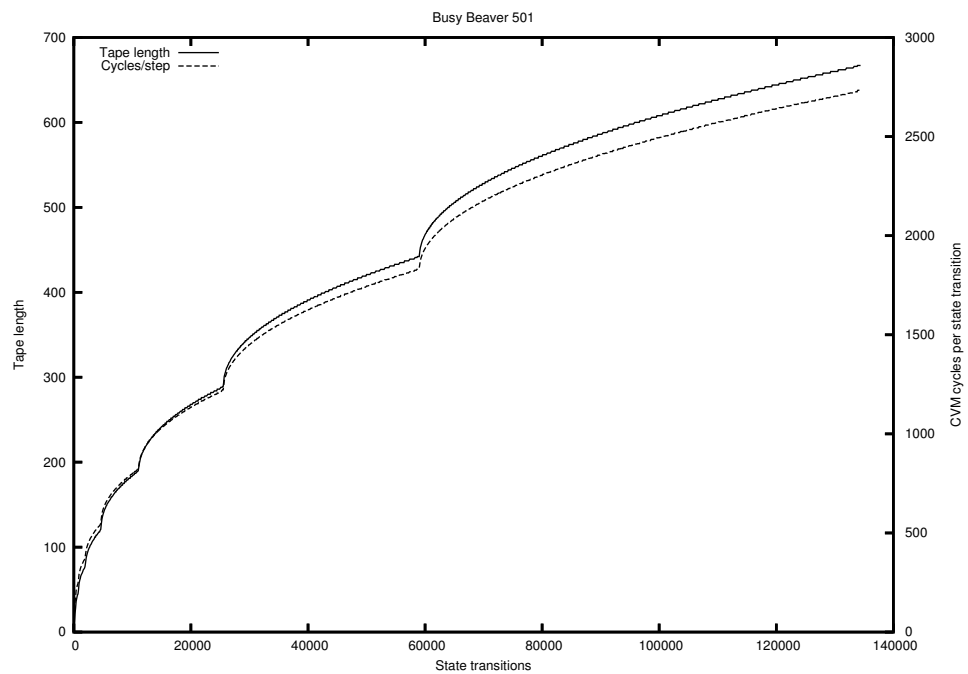
4.2.2 Execution Results

Validating that the cellular implementation performs the instructions correctly was initially a small problem. The LCM virtual machine itself offers only limited possibilities to examine the contents of the cells and it was somewhat hard to implement tape dumping to the two-stack machine. The solution was to use the implemented Turing machine emulator for PC to dump the state transitions (state and readed symbol) of the BB501 to a file. The model’s address bus was attached a special `validate` operation, which compares the input to a reference file. This kind of method is often used in the electronic engineering to validate and verify circuit operations.

Table 5 shows the execution statistics from CVM. During the execution, an interesting observation was made — the execution slows down the further it proceeds. It was thought that this behaviour is caused by the increasing delays to store and retrieve symbols, when the tape length increases. The Figure 51 shows a graph of the execution speed (CVM cycles per state transition) and the tape length of the BB501, and it clearly shows that the execution slows down as the tape length increases.

Table 5. CVM BB501 execution results.

Parameter	Value	
CVM Execution Statistics		
Execution	Total	Per sec
Cycles	260,521,132	5,651.11
Transactions	1,648,208,517	35,752.21
Execution time	12 h 48 min (46100.88 s)	
Transactions / cycle	6.32	
Per configured cells	0.08 %	
Per all cells	< 0.005 %	
Counter values		
Steps	134,468	

**Figure 51.** Execution speed and tape length of Busy Beaver 501.

4.3 Self-Replication

As the array model was specifically designed for reconfiguration, the construction of different machines is easy compared to many very simple cellular automata. For testing the configuration capabilities of the array, a self replication was selected as a problem to solve. Originally, it was planned that the self-replicating machine is implemented using a synthesized model of von Neumann architecture processor in the cellular array; the design of a VNP (Von Neumann Processor) can still be found from Appendix F. During the making of the Thesis, a simpler approach was invented.

Basically, the construction machine in this LCM array model is a plain `config` operation; it takes the description of the machine as input, and as a result the machine is configured to cellular array. Simpler cellular automata may require a more complex construction machine.

The description is stored in a chain of `move` operations. Each `move` cell holds just one symbol. As the configuration string of a `move` cell in the reconfiguration mechanism implemented for the cellular machine requires six symbols (direction, function, two separating symbols, a direction for input and end marker), it is not possible to store the description of the tape to the tape itself (the description requires always six time larger tape than it can configure).

To overcome this problem, the description must be compressed in some way; thus, the replicating machine needs a decompressing engine to be attached between the tape and the constructing machine, as presented in Figure 52. As long as the tape capacity is larger than the description needed to build it, the tape can contain the description of itself, as well as descriptions of other useful parts of the replicating machine. The description may also contain extra configurations that are not part of the replication machine, called payload; the extra payload is replicated as a side effect, and it carries constructions that were actually wanted to be replicated.

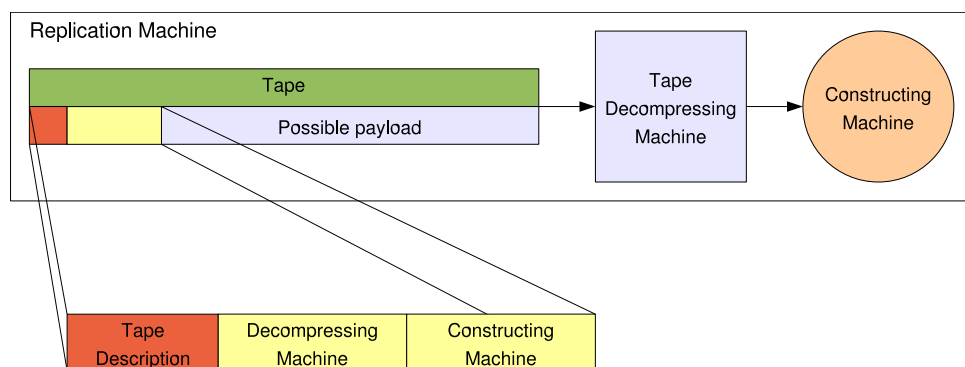


Figure 52. The basic idea of the replicating loop.

A diagram of a self-replicating machine created for this Thesis is shown in Figure 53. It contains a lockable tape, a decompressing engine and the construction arm. The compres-

sion method selected for this machine was a simple RLE (Run-Length Encoding) compression. The decompressing engine is formed from three parts; (1) the control block, which detects whenever the incoming symbol is compressed or not, (2) the counting loop for repeating a pattern specified times, and (3) a loop holding the repeated pattern. The tape content for the replicating machine can be found from Appendix B.

Figure 54 shows a sequence of figures from execution of the self-replicating loop in CVM. The construction of the copy has two phases; first the loop creates a copy of itself from the description, and then it feeds the description to the newly created machine. As shown in the figure, the newly created machine then starts to create a new copy of itself.

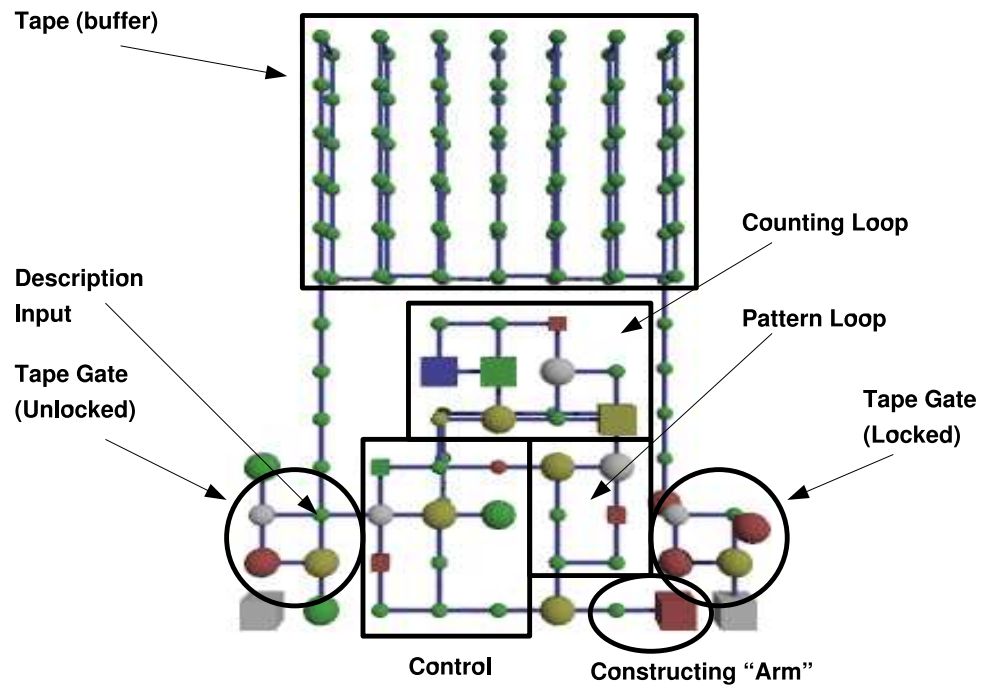


Figure 53. Self-Replicating Loop. Note: The tape is shrunk in this visualization.

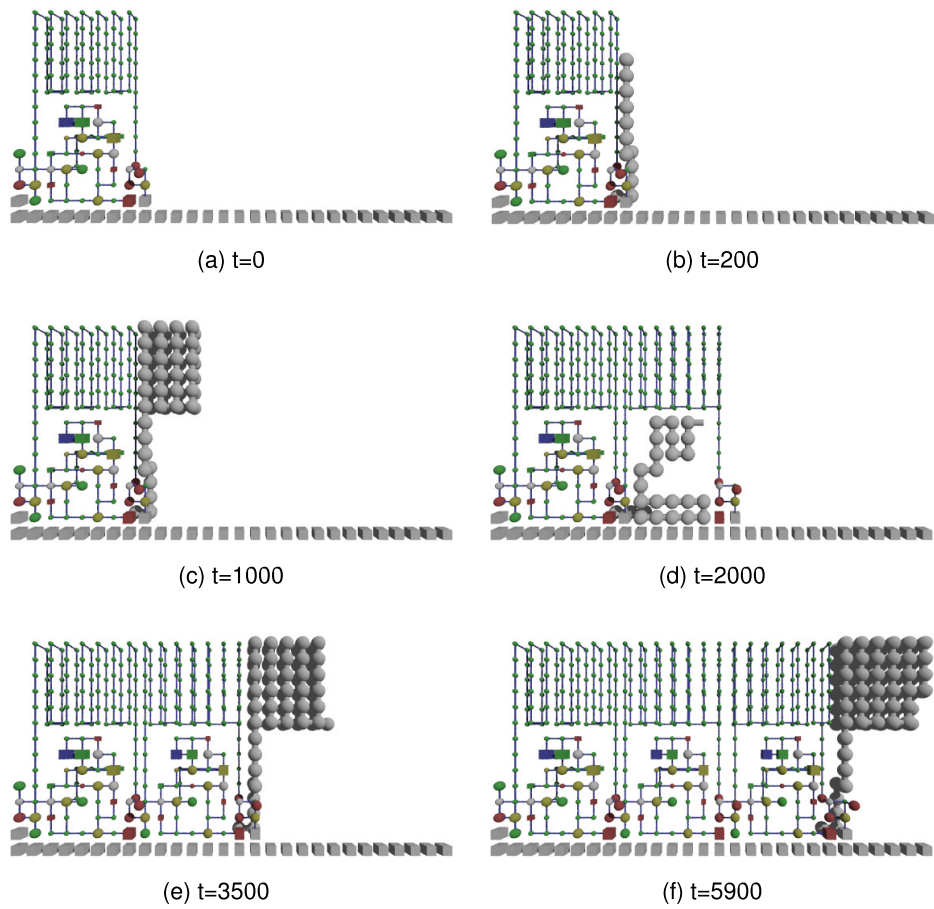


Figure 54. Self-Replicating Loop Execution.

4.4 Solving Eight Queens' Problem

The eight queens' problem was selected as an example for solving computational problems with LCM array. The problem statement is;

“Place eight queens to a 8x8 chessboard so that none of them can capture any other using the standard chess queen's moves.”

It is often used as an exercise in basic programming courses, and it is known to be solvable with a simple program, although it is not a trivial problem. For solving the problem with a cellular devices, two approaches was selected;

1. emulating a conventional, sequential processor with the cellular array, and programming that processor to solve the problem with a simple recursive loop, and
2. creating a more cellular-like device, exploiting parallelism and solving the problem in more non-traditional way.

The following sections first introduce the device configurations made for solving the problem, and then show the execution results.

4.4.1 MLI, Machine Language Interpreter

MLI, Machine Language Interpreter, is a model of simple sequential RISC (Reduced Instruction Set Computer) processor. The word *interpreter* in the name was selected to emphasize that this model is cellular machine, that interpreters a specified sequential language. In other words, MLI is a processor emulator — or a virtual machine — inside of the array.

The overall architecture of MLI is shown in Figure 55, and the visualizations of the CVM configuration are shown in Figure 57. It follows the so called *Harvard Architecture*, in which the code and data are stored to separate memories. This allows the instruction format to be different to data format, which means that the processor does not necessarily need instruction decoding. For more details, see Appendix C.

To test the MLI model, the PC emulator was first created with D programming language. It was used to develop an assembler source code for solving the eight queens' problem. The assembler source code for solving the eight queens' problem can be found from Appendix E. It was manually compiled from corresponding C source code (see Appendix D) used as reference.

Assembler code is compiled to MLI with a dedicated assembler, written with D programming language, which produces memory dump files. These files can be loaded to either PC emulator or MLI generator, a program that generates MLI configuration for LCM array virtual machine (Figure 56).

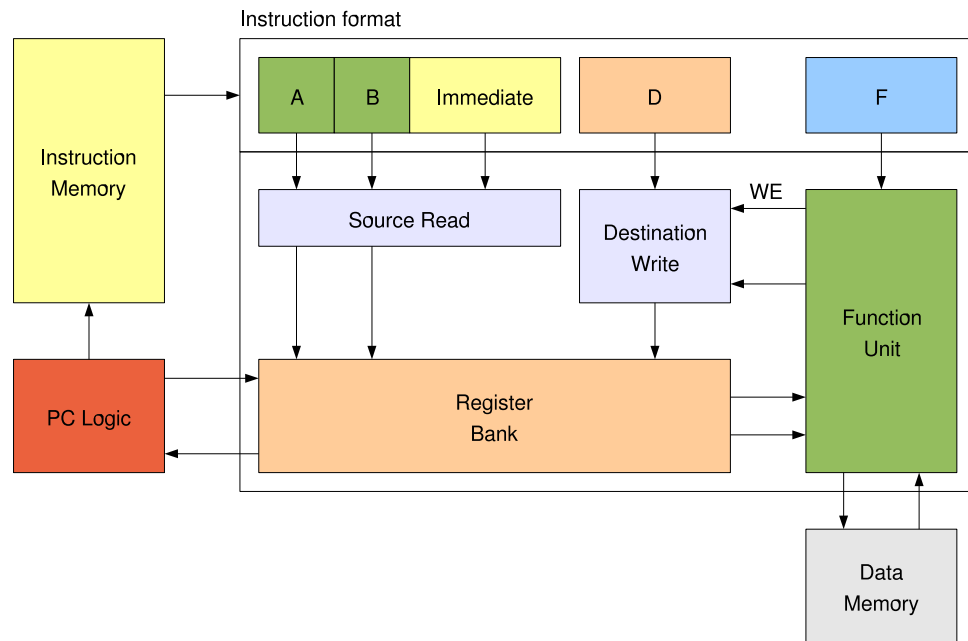


Figure 55. MLI architecture.

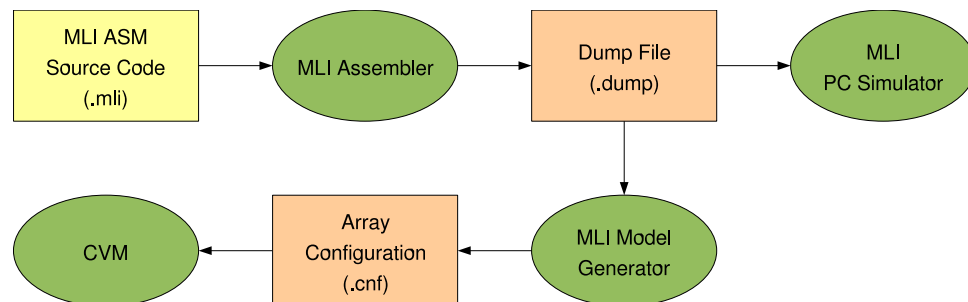
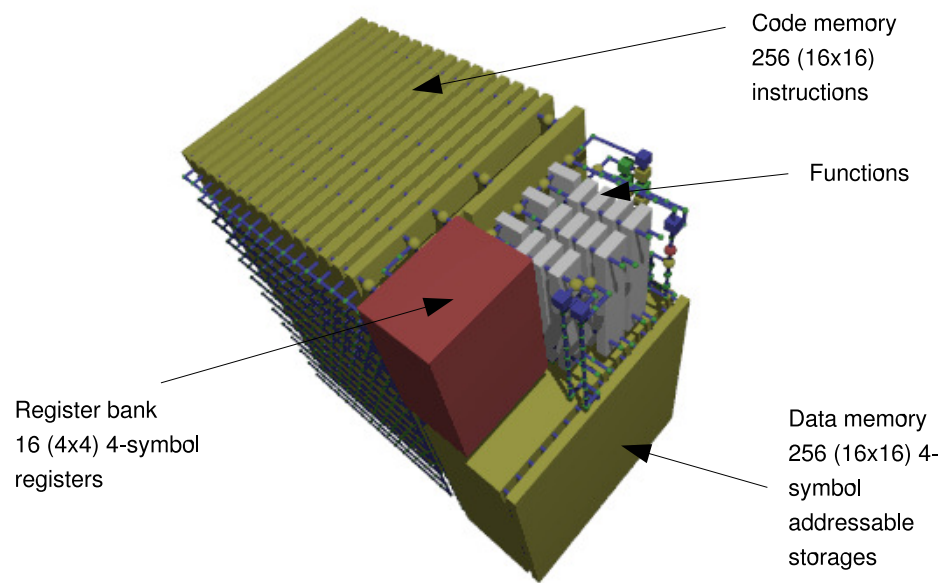
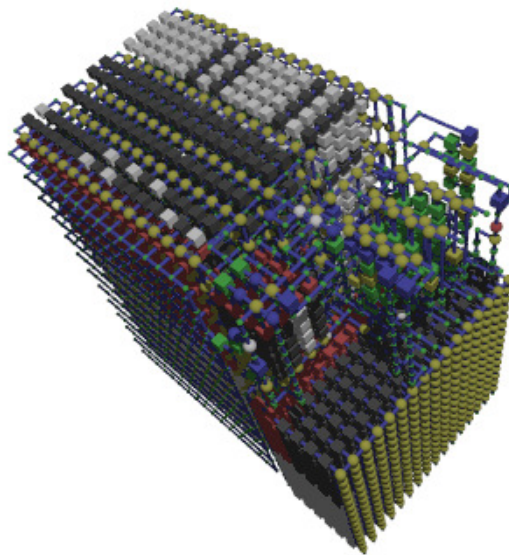


Figure 56. MLI tool chain. The MLI assembler source code files are compiled to a memory dump file with MLI assembler. This memory dump can be loaded to PC emulator, or to a MLI generator to generate preprogrammed MLI model for LCM array virtual machine.



(a)



(b)

Figure 57. MLI Visualization. (a) Details covered with boxes, (b) uncovered visualization

4.4.2 DEQPS, Dedicated Eight Queens' Problem Solver

The DEQPS, Dedicated Eight Queens' Problem Solver, solves the problem in very different way compared to the MLI processor model. The device generator gets the number of queens ($N = [1 \dots 8]$) to place to the checkboard as a parameter, and uses it to create appropriate number of stages for solving the problem. Each generated stage take the candidates evaluated by previous stage as input and match all the possible locations to that input. If the location is valid for placing a queen, it is sent to the next stage for further processing; otherwise it is rejected.

Each stage represents one row in a checkboard. See Figure 58 for an example of a board situation, the representation of that situation and the candidates generated from that situation.

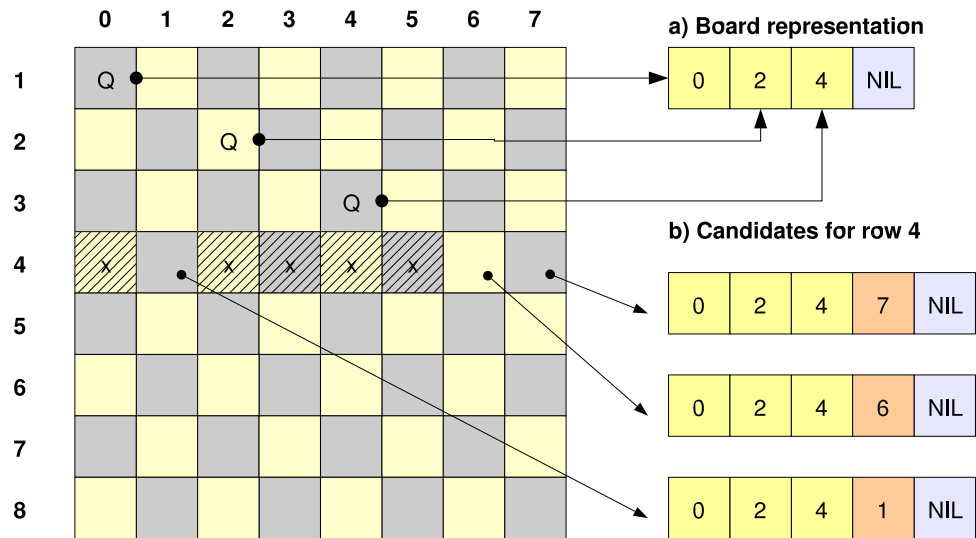


Figure 58. Example board situation of DEQPS. The board situation is at the left side. Three previous stages have placed the queens to columns 0, 2 and 4. The board situation is represented as a string containing the columns of placed queens (a). The fourth stage generates three new candidates from this input by placing a queen to all unoccupied columns. These three new candidates are sent to next stage for evaluating

The visualization of a DEQPS generated for solving eight queens problem is shown in Figure 59. Figure 60 shows details of synthesis of the fifth stage, matching the queen to row five.

4.4.3 Execution Results

Eight queens' problem was solved with four different ways;

1. Compiling a C program (see Appendix D); this was used as a reference, and it

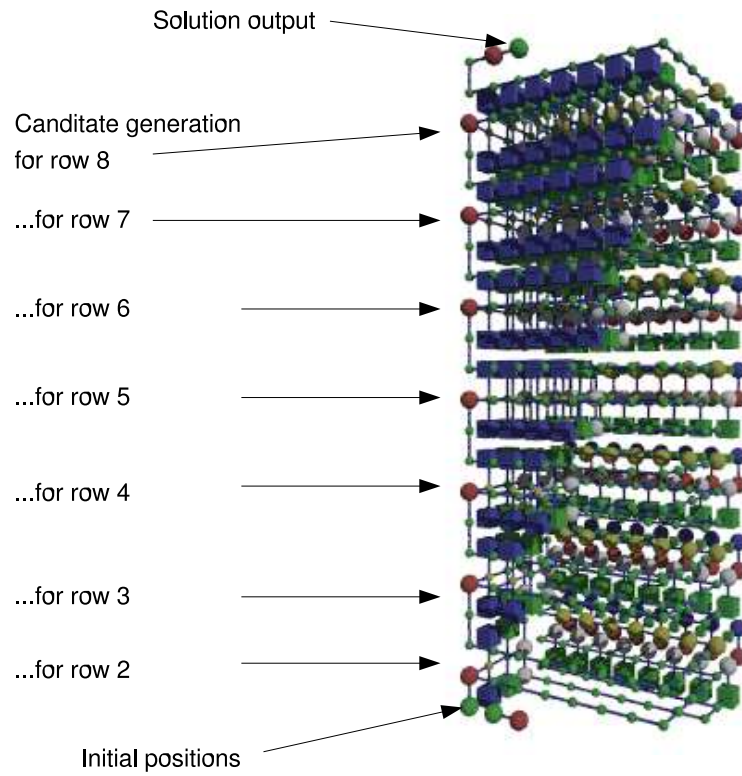


Figure 59. DEQPS for Eight Queens.

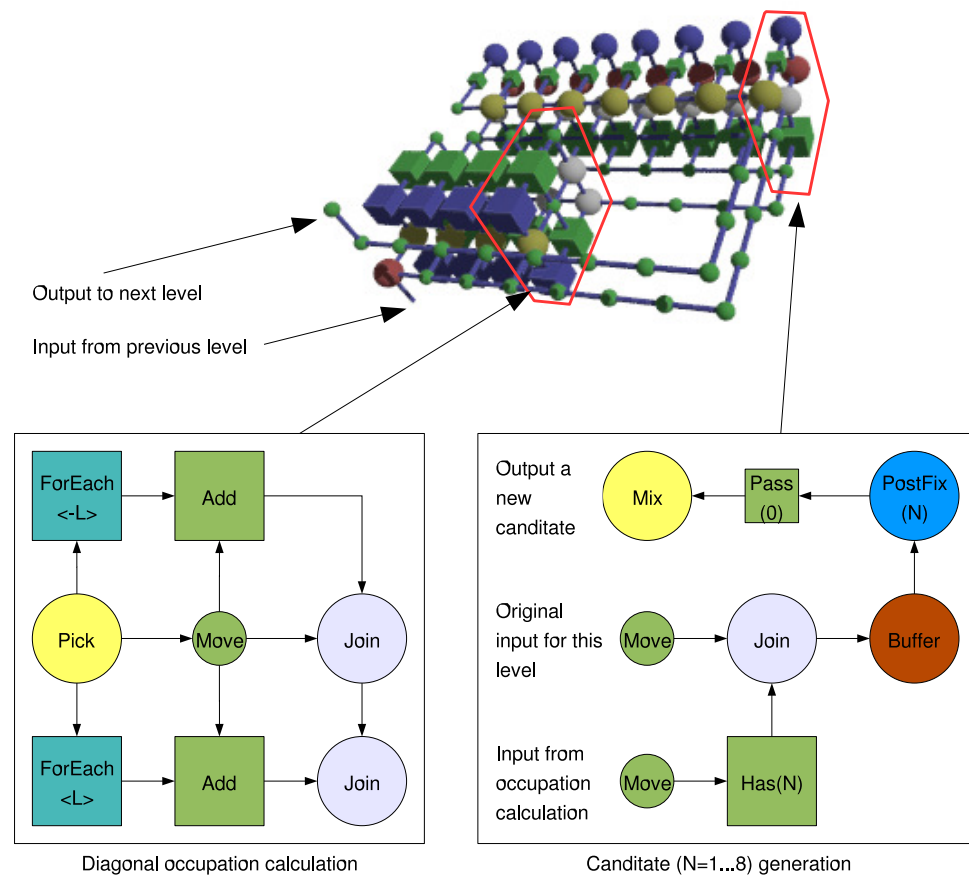


Figure 60. DEQPS Stage 5.

was also used for implementing the MLI assembler solver.

2. MLI emulator executing the solving algorithm written with MLI assembler (see Appendix E); the emulator was written with D programming language and executed in a PC. The results were used for validating the MLI model execution in cellular array.
3. MLI configuration executed in CVM.
4. DEQPS configuration executed in CVM.

Eight queens' problem has 92 distinct solutions. The reference solutions were taken from the solver written with C programming language, and used for validating the correct operation of other solving algorithms. As the MLI assembler uses the same algorithm than the C reference solver, the solutions were generated in same order. DEQPS instead generates the solutions in different order. All solutions were validated to be correct.

The number of operations used by MLI processor executing the algorithm are shown in Table 6. Using counters inside MLI processor configured to CVM it was possible to ensure, that the synthesized model performed exactly the same number of same operations as the emulator written to PC.

Table 6. Operations performed by MLI.

Parameter	MLI
Instruction fetches	294,491
Data memory accesses (tot)	89,722
Data memory reads	62,899
Data memory writes	26,823

The raw performance measurements are shown in Table 7. As can be seen from the table, DEQPS uses just 0.1% of the transactions needed by MLI model to solve the problem. When at the same time the DEQPS has greater level of parallelism (in average 156.59 simultaneous transactions, compared to 26.19 of MLI), the DEQPS solves the problem in just 12,080 CVM cycles. This was of course an expected result.

Table 7. Performance of eight queens' problem solving.

Parameter	MLI	DEQPS	Ref.
CVM execution time	488.79 s	1.41 s	294,491
Cell transactions	1,210,057,860	1,892,894	
Virtual machine cycles	46,206,826	12,080	
Transactions / cycle	26.19	156.69	
Active cells / cycle	0.29 %	17.52 %	

Although it was expected that the dedicated cellular structure performs better than an emulation of sequential machine, the DEQPS performance was surprisingly good. It could be considered that two actual hardware processors running with same frequency would made; (1) an actual MLI processor, which would execute one instruction per clock cycle, and (2) an actual cellular array with same transaction frequency. Table 8 shows a comparison between these two platforms, running at 10 MHz frequency; DEQPS on cellular array would still solve the eight queens' problem 25 times faster than the corresponding single-cycle hardware MLI. As one of the reasons to study asynchronous cellular arrays is that they allow much higher frequencies compared to sequential processor architectures, the actual real-world performance of the DEQPS would be even higher.

Table 8. Platform performance comparison.

Processor	Machine cycles	Time, $f = 10\text{ MHz}$
Hardware MLI	294,491	29.94 ms
Cellular MLI	46,206,826	4,620.68 ms
Cellular DEQPS	12,080	1,21 ms

To study the activity of the in the cellular constructions, the CVM was used to produce visualizations where the color of the cell indicates the activity of the cell, see Figure 61 and Figure 62. The brighter the color, the more frequently the cell has been active. Activity correlates to heat generation; the examination can show which parts of the constructions may suffer overheating and thus cause performance penalties.

The most heavily loaded part in the MLI construction is the address comparator block of the register bank. The heavy loading of the register bank was expected, since it is the central part of the instruction cycle, referenced four times per cycle; program counter fetch, two input register reads and one register write. As can be seen in the figure, also the channels routing the inputs to function bank show frequent activity. The other remarkably loaded part is the address comparator block of the instruction memory, connected to program counter logic. The activity of other parts is minimal; memory blocks (registers, instructions and data) as well as functions show no remarkable frequent activity.

In the DEQPS construction, the most activity happens (1) in the upper stages of the DEQPS, and (2) in the candidate matching parts. Especially the `has` operations, which scans the column occupation, are heavily loaded at upper stages of the solver.

To find out how the activity changes during the execution of DEQPS, a serie of visualizations were created, see Figure 63. As can be seen from the images, the activity quickly reaches the upper stages. Also, the figures indicate that most of the execution time is spent processing solution candiditates at the upper levels — but notice also, that the last stage is not as heavily loaded as the few stages below it.

As discussed earlier, the lack of energy may prevent utilizing all available parallelism. This was examined with DEQPS configuration by limiting the maximum amount of simultaneous transactions by CVM. The execution results are shown in Table 9 and plotted

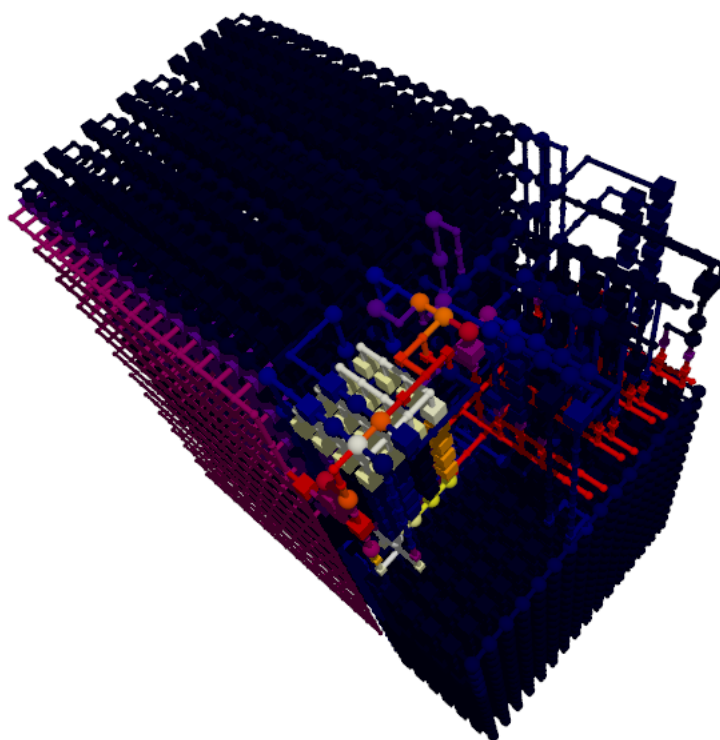


Figure 61. MLI total activity visualization.

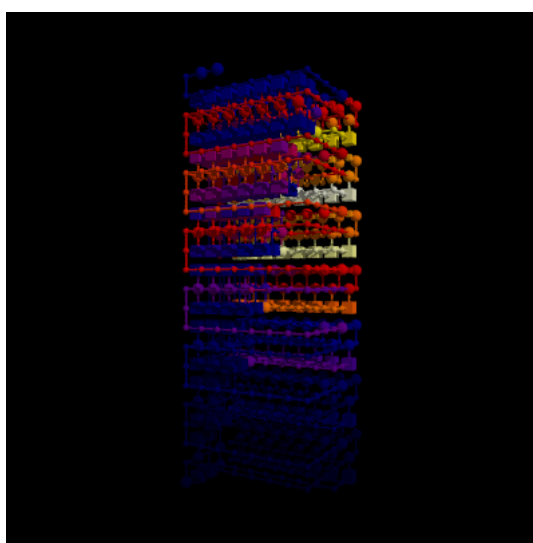


Figure 62. DEQPS total activity visualization.

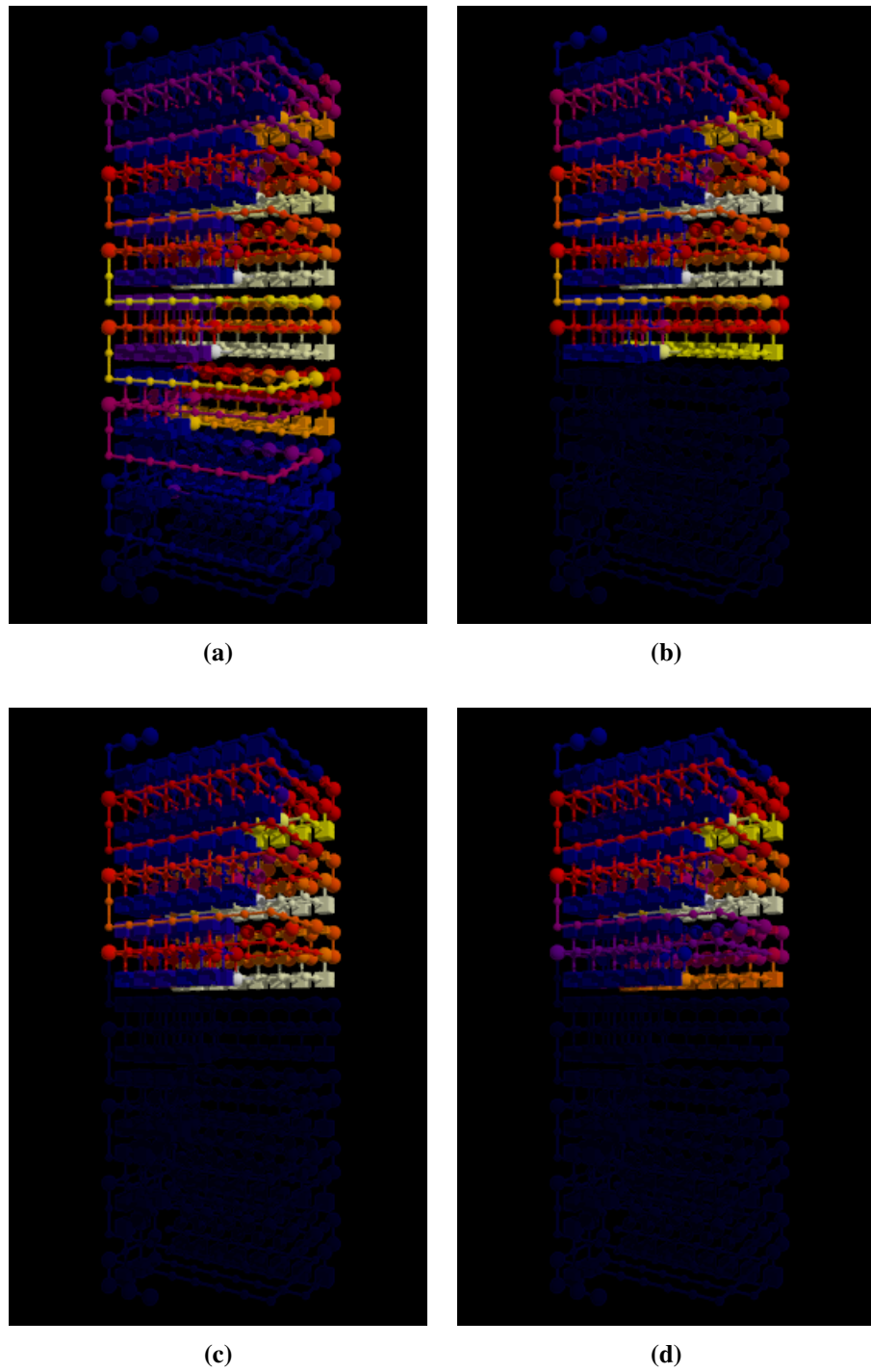
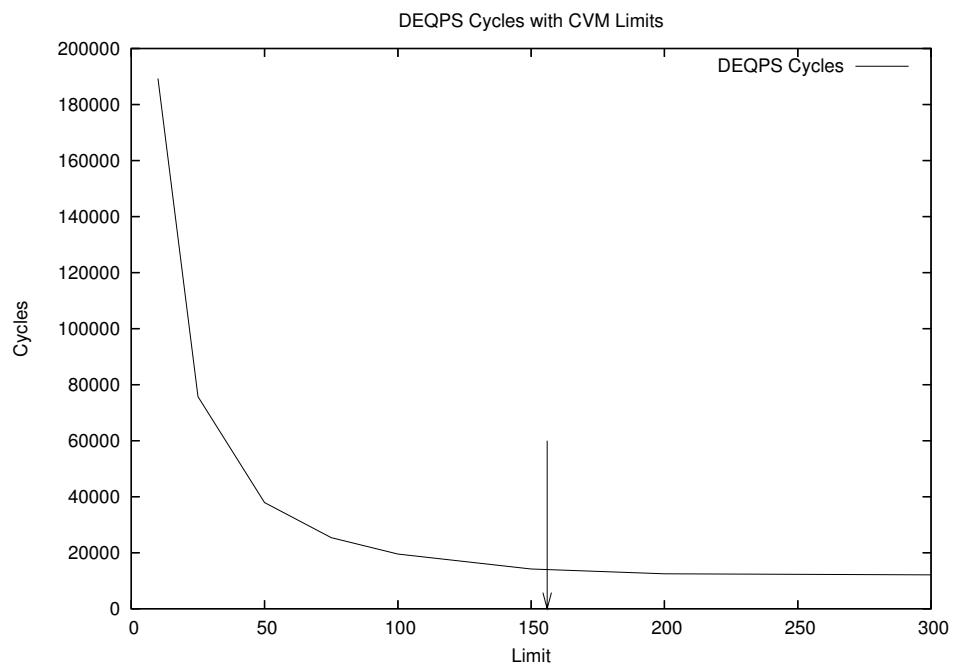


Figure 63. DEQPS activity visualization during execution. a) After 1/4. b) After 1/2. c) After 3/4. d) After completing. The activity transfers fast to the upper layers of the construction.

Table 9. Effects of transaction limitation.

Limit	DEQPS cycles	Ratio
unlimited	12,080	1.00
300	12,080	1.00
200	12,486	1.03
150	14,169	1.17
100	19,525	1.62
75	25,368	2.10
50	37,903	3.14
25	75,731	6.27
10	189,295	15.67

**Figure 64.** Effects of transaction limitation.

to a graph shown in Figure 64. Limiting did not have any effect to the correctness of the computation, i.e. the DEQPS was still able to produce all the 92 solutions, neither did it has any effects to the total number of transactions needed to produce all solutions (1,892,894 transactions).

The results clearly show the drop of the performance, when the limit goes below the DEQPS average parallel ratio (156.69). Also, this examination ensured the expectation that an asynchronous cellular array really is delay-insensitive, i.e. the correctness of the computations is preserved although cells can not operate as soon as their inputs are ready.

4.5 Configuration Statistics

Table 10 shows the cell usage of the configured BB501 Turing machine. The large number of empty cells is caused the spatial arrangement of the model; the tape and state transition table were placed so that they produce nice images instead of packing them more efficiently to the allocated cellular space.

The configuration is heavily dominated by routing and memory cells, only a 0.25% of the cells are used for other purposes. The large amount of cells needed to implement memory is not surprising; the TM is a combination of two memory devices, a tape and a transition table.

Table 11 shows the cell usage of the replicating machine. Unlike the TM, this configuration contains only a few memory elements; but that is not entirely true, since the configuration uses `move` cells as memory, and those cells are counted to routing cells. As can be seen from the table, the routing cells dominate in this design.

Table 12 contains the cell resource usage of both MLI and DEQPS configurations, used for solving the eight queens' problem. The resource usage of MLI is very similar to TM; it is almost entirely made up from memory and routing cells, only 1.84% of the cells are something else. DEQPS instead contains only a relatively small amount of memory cells (20%); almost a half is used for routing, and the rest 35% is used for other purposes.

Figure 65 shows a diagram of the size (number of configured cells) of the synthesized models compared to each other. The MLI and BB501 configurations are about ten times larger than the DEQPS configuration. The replicator is very small compared to other configurations.

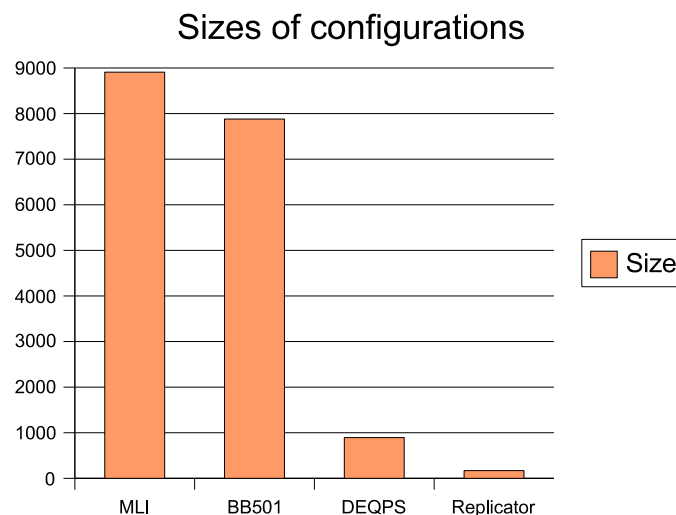


Figure 65. Sizes of the configurations.

The two different strategies for solving the eight queens' problem was initially expected to also give some support for the idea of computing dualism by Hartenstein (1997), that

Table 10. BB501 cellular resource usage.

Parameter	Value	
Model Statistics		
Dimensions	1415 x 13 x 4	
Volume	73,580 cells	100.00 %
Empty	65,699 cells	89.29 %
Configured	7,881 cells	10.71 %
Configured Cell Classification		
Memory	5,658 cells	71.79 %
- Storage	2,832 cells	35.93 %
- Addressing	2,830 cells	35.91 %
Routes	2,203 cells	27.95 %
Other	20 cells	0.25 %

Table 11. Replicator cellular resource usage.

Parameter	Value	
Model Statistics		
Dimensions	84 x 7 x 2	
Volume	1,176 cells	100.00 %
Empty	1,005 cells	85.46 %
Configured	171 cells	14.54 %
Configured Cell Classification		
Memory	9 cells	5.26 %
- Storage	4 cells	2.34 %
- Addressing	5 cells	2.92 %
Routes	118 cells	69.01 %
Other	44 cells	25.73 %

Table 12. MLI and DEQPS cell resource usage.

Parameter	MLI Model		DEQPS Model	
	cells	%	cells	%
Model Statistics				
Dimensions (W x H x D)	16 x 27 x 25		8 x 22 x 7	
Volume	10,800	100.00 %	1,232	100.00 %
Used cells	8,910	82.50 %	894	72.56 %
Empty cells	1,890	17.50 %	338	27.44 %
Configured Cell Classification				
Memory cells	4,736	53.15 %	170	19.02 %
Routing cells	4,010	45.01 %	406	45.41 %
Rest	164	1.84 %	318	35.57 %

is, computing-in-space (parallel solutions) and computing-in-time (sequential solutions). The basic idea in computing dualism is that the computation can be done with smaller amounts of spatial resources, if it uses more time, and vice versa. Since the dedicated solver is much smaller than the corresponding sequential solution, the eight queens' problem was probably too simple.

Figure 66 shows a pie slice diagrams of the volume usage efficiency of the synthesized models. This clearly shows how the BB501 configuration wastes space. The MLI and DEQPS, both designed inside a cube, are considerably more efficient in space usage. Replicator wastes space, since it is essentially a two-dimensional configuration, using the third dimension only slightly.

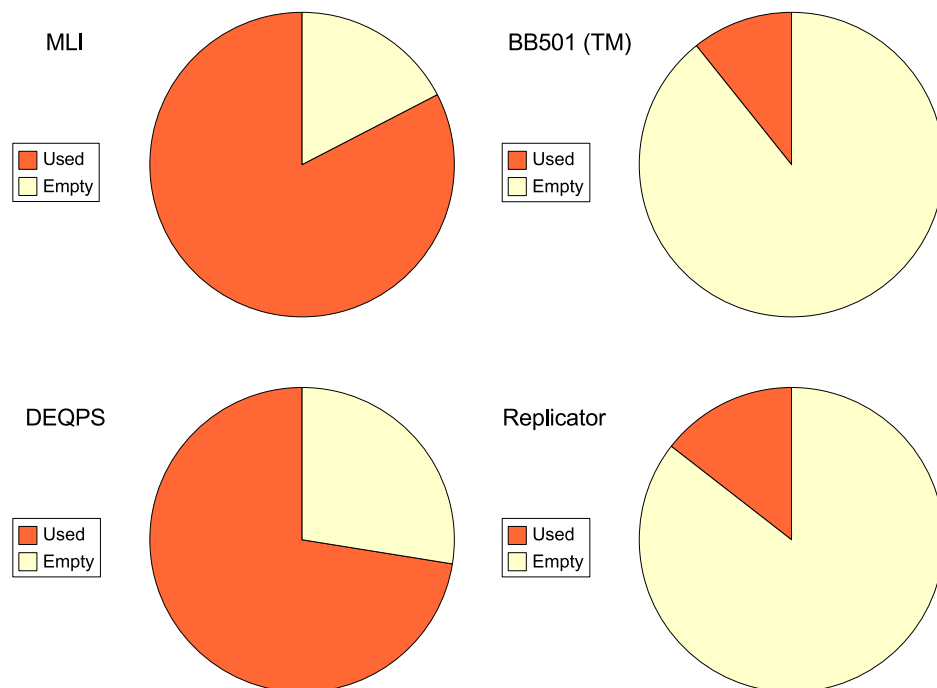


Figure 66. Volume usage efficiency.

Figure 67 shows the relative amounts of different cell classes used in the synthesized models. The large state machines, MLI and BB501, are mainly configured from routing and memory cells. The small, more special configurations DEQPS and replicator, contain much higher ratio of other logic cells.

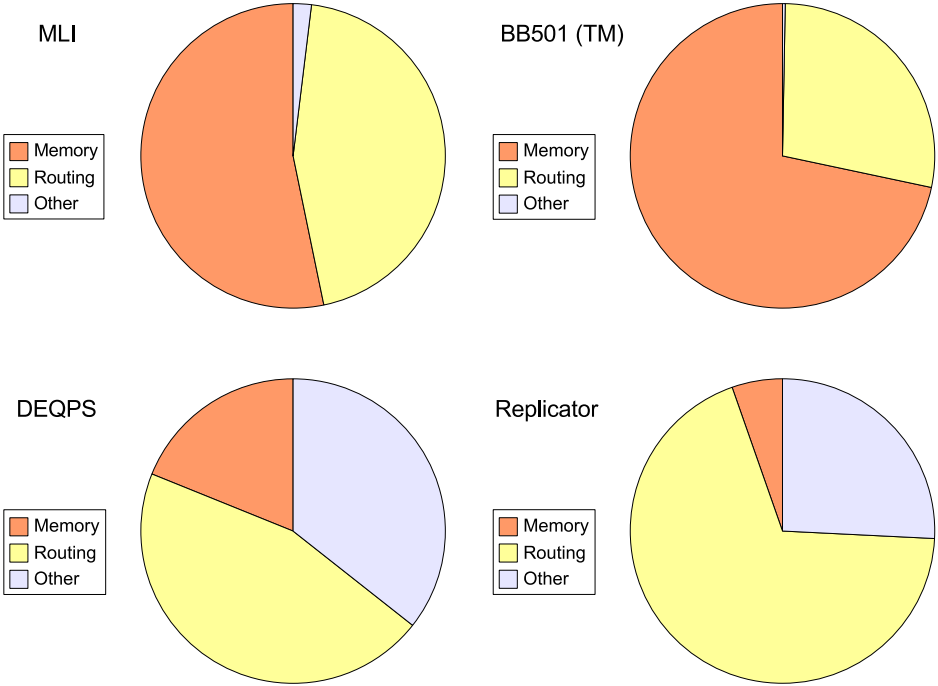


Figure 67. Cell usage classes.

4.6 Theoretical Examination

As discussed earlier, a general purpose computer is Turing-complete and self-reconfigurable. For people familiar with computation theories, formal automata and very simple constructions, it is probably already obvious that the LCM array as a complex model is computationally universal and self-reconfigurable. These people may find this section hardly interesting, but since the author is not so familiar with these, the questions are examined here.

4.6.1 Computational Universality

An elegant way to show the computation universality would be formalizing the model mathematically and show that it is equivalent to Universal Turing machine. But as the LCM model is not formalized¹⁰ in this Thesis, it is only possible to give evidence that the LCM array would be universal, if it would be formalized.

Another elegant way would be to use some simple universal asynchronous cellular automaton, and show that its rule set has equivalences in LCM operation set. The problem was to find a suitable, simple asynchronous CA close to LCM design for doing this.

The Turing machine emulation executing Busy Beaver 501 is certainly a clear indication of Turing completeness. Since the design of the Turing machine for the cellular array shows no restrictions to the length of the tape, alphabets and number of states except the availability of cells, it is very clear that any kind of Turing machine can be emulated, indicating that the cellular array equivalent to UTM.

An additional examination is made indirectly, using a computationally universal RAM machine defined in a book *Models of Computation* (Savage, 1998, pp. 110 - 118). It is possible to show that the MLI assembler language, used for MLI processor models, has equal instruction set to that RAM machine.

As denoted by Savage, the RAM machine introduced in the book is not minimal, but it is sufficient for this Thesis. Table 13 shows the RAM instruction and a equivalent sequence of MLI assembler language instructions. The MLI assembler language is thus equivalent to the RAM machine.

As seen in the memory design, the MLI model does not set restrictions to the number of chained address comparators and memory cells, other than the amount of cells available in the array. Thus the MLI processor model sets no restrictions to the width of the values it processes, neither to the amount of memory it can address. As the MLI model executes MLI assembler language, they have (at least) equal computational power. And, because MLI was executed in the LCM array, it is then computationally as powerful as MLI, which is as powerful as its assembler language, which is equivalent to universal RAM machine.

¹⁰The LCM prototype in CVM is formalized, with a D programming language

Table 13. RAM instruction and MLI assembler equivalent.

RAM	MLI	Comments
INC Ri	r0 = load Ri r0 = add r0, 1 store Ri, r0	Also: r0 = inc r0
DEC Ri	r0 = load Ri r0 = sub r0, 1 store Ri, r0	Also: r0 = dec r0
CLR Ri	r0 = sub r0, r0 store Ri, r0	Also: r0 = move 0
Ri <- Rj	r0 = load Rj store Ri, r0	
JMP+ Ni	pc = move Ni	
JMP- Ni	pc = move Ni	
Rj JMP+ Ni	r0 = move Rj pc = ifz r0, Ni	
Rj JMP- Ni	r0 = move Rj pc = ifz r0, Ni	
CONTINUE	nop stop	If not the last instruction If last instruction

The simplest RAM machine would be OISC (One Instruction Set Computer), which contains only one instruction — subleq (Subtract-And-Branch-If-Negative). But because the author was not able to get William F. Gilreath and Philip A. Laplante’s book *Computer Architecture: A Minimalist Perspective* (Springler, 2003), the more classical RAM machine model was used.

4.6.2 Construction Universality

The construction universality is classically defined so that the CA is capable of hosting universal constructor. The universal constructor is capable of constructing any arbitrary machine, if the description of that machine is given.

To prove the construction universality, first we define an arbitrary machine A . An arbitrary machine contains a finite set of cell configurations, each in any location in the array. A cell configuration C is then a pair of its position p and content c . For the cubical LCM array, the position can be expressed as a three-tuple $\{x, y, z\}$ containing the coordinates relative to the intended reconfiguration point. An arbitrary machine A constructed from N cells is thus;

$$A = [C_1, C_2, \dots, C_N]$$

where;

$$C_i = [p_i, c_i] = [x_i, y_i, z_i, c_i]$$

The reconfiguration mechanism was defined to be complete, that is, any possible cell configuration can be expressed in reconfiguration stream. Each position can be expressed as a stream of empty configurations to reach the position as shown in Figure 68, that is:

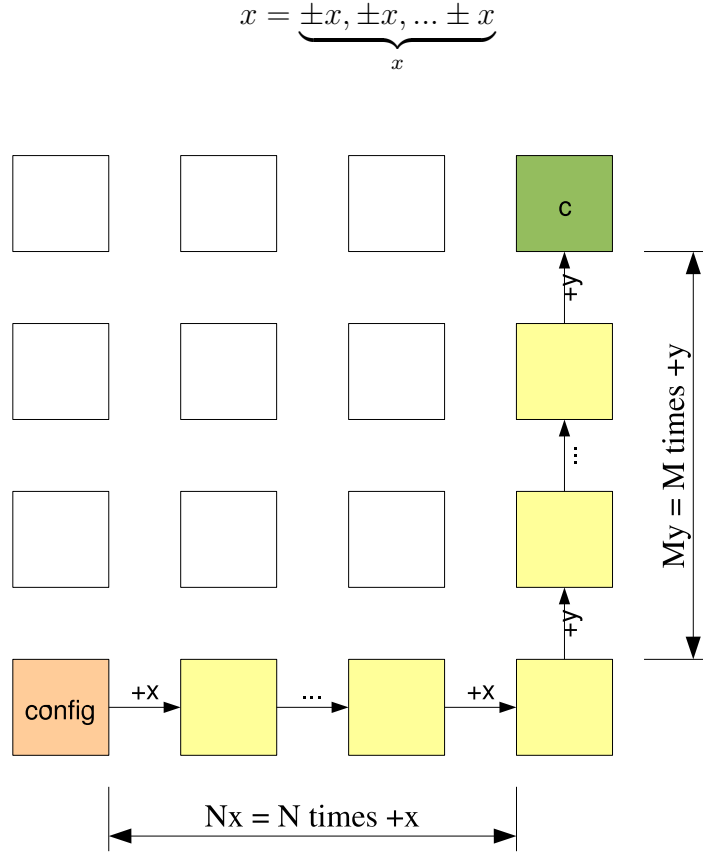


Figure 68. Reaching arbitrary position from configuration point. Reaching an arbitrary point requires outputting streams of empty configurations with directions, and ending it with the desired configuration and a NIL symbol

Thus, the configuration of a single cell is a series of empty configurations in three directions, followed with the cell configuration and terminating NIL;

$$C_i = \underbrace{\pm x, \dots, \pm x}_{x_i} \underbrace{\pm y, \dots, \pm y}_{y_i} \underbrace{\pm z, \dots, \pm z}_{z_i} c_i, NIL$$

Using the previous result about computation universality, we can determine that the it is possible to construct an algorithm to output any stream of values, thus it is possible to construct an algorithm to output any reconfiguration stream, i.e. the system is construction universal.

4.7 Observations

During experimenting the building of cellular devices with CVM, a few observations were made.

Leave Inputs Last. A notable detail in the reconfiguration stream generation is that the `Input` configurations must be left to the last ones to configure. Immediately after the configuration of `Input` is complete, it outputs its contents, whenever there are cells listening it or not. Thus, the rest of the structure must first be present before initializing it.

Debugging. The level the configurations were created is somewhat comparable for making programs to conventional processors with machine language (not assembler). Debugging of the configurations were very hard. The future nanocomputer cells would probably need hardware-level support for debugging the structures.

Deadlocks. The cell structures may “jam” in certain conditions without adequate buffering; regularly this happens when feeding loops with too large strings. In some conditions this was surprising; previous experiments with the virtual machine showed that in general asynchronously operating arrays should be quite insensitive to work load.

In general, structures containing feedback (e.g. a loop) should have enough space for storing the loop content. But these deadlocks are not only the problem of feedbacks, as was seen in DEQPS implementation. Extensive, but unsuccessful debugging were done to find out the fundamental reason, so the problem was postponed to future and circumvented by adding enough buffering to the construction. This problem also shown out how hard it is to find structural errors from cellular devices, which might be considered more closely in the future.

For helping debugging, as well as for more robust operating system design, some of the cells may be aware of deadlocks. This may be somewhat hard to implement, since asynchronicity effectively prevents almost all strategies based on service timings. Anyway, there is a need to study more about the dynamics of the array structures to determine, which kinds of structures can cause deadlocks and what kind of structures could be used for protecting the rest of the system against them.

5. Cellular Computer System

From the very beginning, the driving force for this Thesis has been to get understanding of the software of cellular computers; what kind of software does the cellular computer execute? How it is written? Which languages to use? Does the cellular computer have operating system? Which kind of an operating system? How the applications are loaded to memory? How the cellular computer boots up? This last chapter tries to answer to these questions.

To get this understanding, from the very beginning it was decided to try to use cellular processing array to solve some selected problems; not too difficult, not too trivial. The experience gained from the experiments were meant to be used to enlight the software engineering environment of the cellular computers.

The previous chapters have first discussed about the theoretical and physical properties of massive cellular arrays, and then about cell and computation models. Finally, the cell model were used for experimenting the computations with cellular array.

In this chapter, the observations from previous chapters are collected together and combined with previous researches to form a bigger picture of the software architecture of massive cellular processing arrays. First section discusses about the how cellular software may be built. The later sections concentrate on the software execution.

5.1 Cellular Computer Software

Many researches indicate that the reconfigurable computers are the next evolution of computers (Hartenstein, 2001, 2003; Bishop & Sullivan, 2003; Vuletic *et al.*, 2004; Todman *et al.*, 2005; Smith *et al.*, 2001). The reasons for this are (1) the increasing switch count combined with need for higher frequencies, which utilization is difficult with conventional sequential architectures (Hartenstein, 2001), and (2) the failure of other parallel architectures to subsume conventional von Neumann architectures in the field of general purpose computing (Hartenstein, 2003). As discussed earlier, at the end of the roadmap are pure cellular computers (Frank, 2003a). It is natural to expect that the software development of cellular computers is derived from the preceding reconfigurable computers.

5.1.1 Reconfigurable Computers

Reconfigurable computing is a relatively new research area, which studies computers with highly flexible computing fabrics. The principal difference compared to ordinary micro-processors is the ability to make substantial changes to the data path itself in addition to the control flow. According to Compton & Hauck (2002), reconfigurable computing is intended to fill the gap between hardware and software, achieving potentially much higher performance than software, while maintaining a higher level flexibility than hardware.

The roots of reconfigurable computers go back to early 60's, when Estrin *et al.* (1963) introduced their concept of a computer, which contained a sequential processor and a reconfigurable co-hardware. The wider interest to reconfigurable computing came in 90's, when field-programmable devices reached cell count and performance high enough to make them viable alternatives to ASIC (Application Specific Integrated Circuit) implementations (Rincon & Teres, 1998; Hartenstein, 2001).

A few examples of both academic and commercial reconfigurable platforms;

1. Two examples of sequential processors with reconfigurable FPGA-like cellular hardware are Garp (Hauser & Wawrzynek, 1997) and Spyder (Iseli & Sanchez, 1993). In one extreme these architectures may be RISPs (Reconfigurable Instruction Set Processor) (Barat *et al.*, 2002), in which the reconfigurable unit is inside of the sequential processor function units.
2. PipeRench (Goldstein *et al.*, 2000) and PACT XPP (eXtreme Processing Platform) (Baumgarte *et al.*, 2003) are examples of processing meshes which aim for hardware virtualizations,
3. Processor meshes, like MIT RAW (Taylor, 2003; Taylor *et al.*, 2002; Agarwal, 1999), IBM Cyclops64 and STI¹¹ CBEA (Cell Broadband Engine Architectur), known simply as "Cell". STI Cell is widely known as the main processor of the Sony's PlayStation 3.

Compared to conventional computers using only sequential processor, reconfigurable computers offer more performance in such computing tasks, which parallelize easily, like DSP (Digital Signal Processing) algorithms (Hartenstein, 2001; Bishop & Sullivan, 2003; Smith *et al.*, 2001; Hauser & Wawrzynek, 1997; Callahan *et al.*, 2000). Moreover, one main interest to reconfigurable computing is the possibility to integrate the development of software and hardware together (e.g. Kimura *et al.*, 1997; Bellows & Hutchings, 1998; Brebner, 1999).

¹¹ STI, Sony-Toshiba-IBM

5.1.2 Programming Reconfigurable Computers

Traditionally, sequential computers have been programmed with control-flow languages like C, C++ and Java, which follow the underlying sequential paradigm. Similarly, the designing of hardware has traditionally used data-flow languages like VHDL, which follow the underlying data-parallel paradigm. This paradigmatic difference in these two worlds is seen as one of the most challenging subject in the area of reconfigurable computers (Vuletic *et al.*, 2004; Brebner, 1999).

Usually the hybrid reconfigurable processors are combined from somewhat standard sequential processor (e.g. MIPS in Garp) and programmable hardware block (e.g. roughly Xilinx 4000 series equivalent in Garp). The standard blocks usually have already working tools for programming them; processors have C compiler, and configurable hardware VHDL synthesizing tools. In these situations the simplest and most straightforward way to build software is to use the existing VHDL synthesizer to build the hardware configurations, and join these configurations with sequential processor programs written with C programming language (e.g. Hauser & Wawrzynek, 1997).

The problem with such approach is that the source codes are hardly portable. For that reason, there has been several studies for using a single, standard programming language to generate both configurations and control software. Obviously, conventional programming languages (C, C++) has gained much interest; but there are also several studies proposing data-flow languages for programming reconfigurable computers.

Researches of using modified C/C++ compiler to produce both configurations and control software from standard unmodified C/C++ source code, are e.g. programming XPP (Cardoso & Weinhardt, 2002, 2003) and Garp (Callahan, 2003; Callahan *et al.*, 2000). In general, the source code is first partitioned with datapath analysis to make decisions, which parts are compiled to host microprocessor and which parts are synthesized to hardware configurations. The Garp C compiler (garpcc) concentrates synthesizing the inner kernels of loops, since they offer easily huge performance benefits. Using Java instead of C has been researched by Kuhn & Huss (2004).

Cardoso & Neto (2003) developed a compiler to synthesize Java Bytecode to reconfigurable hardware; the benefit using Java Bytecode as an intermediate hardware-independent language is that it allows using many kinds of high-level languages. Another approach has been to extend the standard sequential language with parallel extensions, like PPC (Polymorphic Parallel C) by Maresca & Baglietto (1993) and RT-C by Lee *et al.* (2003).

At the other side, using dataflow languages for programming the reconfigurable computers is also studied, like SA-C (Single-Assignment C) (Böhm *et al.*, 2002), JHDL (Java-HDL) (Bellows & Hutchings, 1998), Circal (Diessel & Milne, 2001), HJJ (Hardware Joint Java) (Hopf & Kearney, 2003) and V (Strelzoff, 2004).

5.1.3 Programming Cellular Computers

From the cellular computer perspective, the research of reconfigurable computer software indicates that;

1. The cellular computer is programmed with standard programming languages, i.e. there is no need for developing a specialized language for cellular computers. The advances in compiler technology supposedly hide the paradigm differences between language and hardware, which indicate that the future programming languages concentrate on productivity; they are made easier for humans to describe large and complex software.
2. The cellular computer compilers have even greater flexibility to produce configurations; this increases the importance of source code analysis and partitioning for gaining performance benefits.

Figure 69 shows an example of complete toolchain to create cellular computer software. The flow is coarsely divided to three phases; (1) partitioning, (2) code generation, and (3) assembling.

The partitioning phase analyses the high level description of the program. It may do it by precompiling the source to some intermediate language. Depending on the optimization hints given to the compiler, it selects the parts that are synthesised to cellular configurations and parts that are compiled to instructions for sequential processing devices.

The code generation phase takes the partitioned description as an input, and produces corresponding compilations from them. This phase includes a database containing descriptions of several devices, e.g. different kinds of sequential processor models and the instructions to compile software for them. The compiler may also have modules to customize the database, e.g. the compiler may create sequential processing devices with customized instruction sets suitable for executing programs for specific parts of the software.

Both the instruction streams and the cellular configurations are assembled together, to form the complete cellular software package (cellular computer configuration). Along with the actual configuration, the compilation flow needs to output metadata for supporting the debugging of the software. This may be much harder for cellular computers (and reconfigurable computers, too), since there is no direct mapping between instructions and source code.

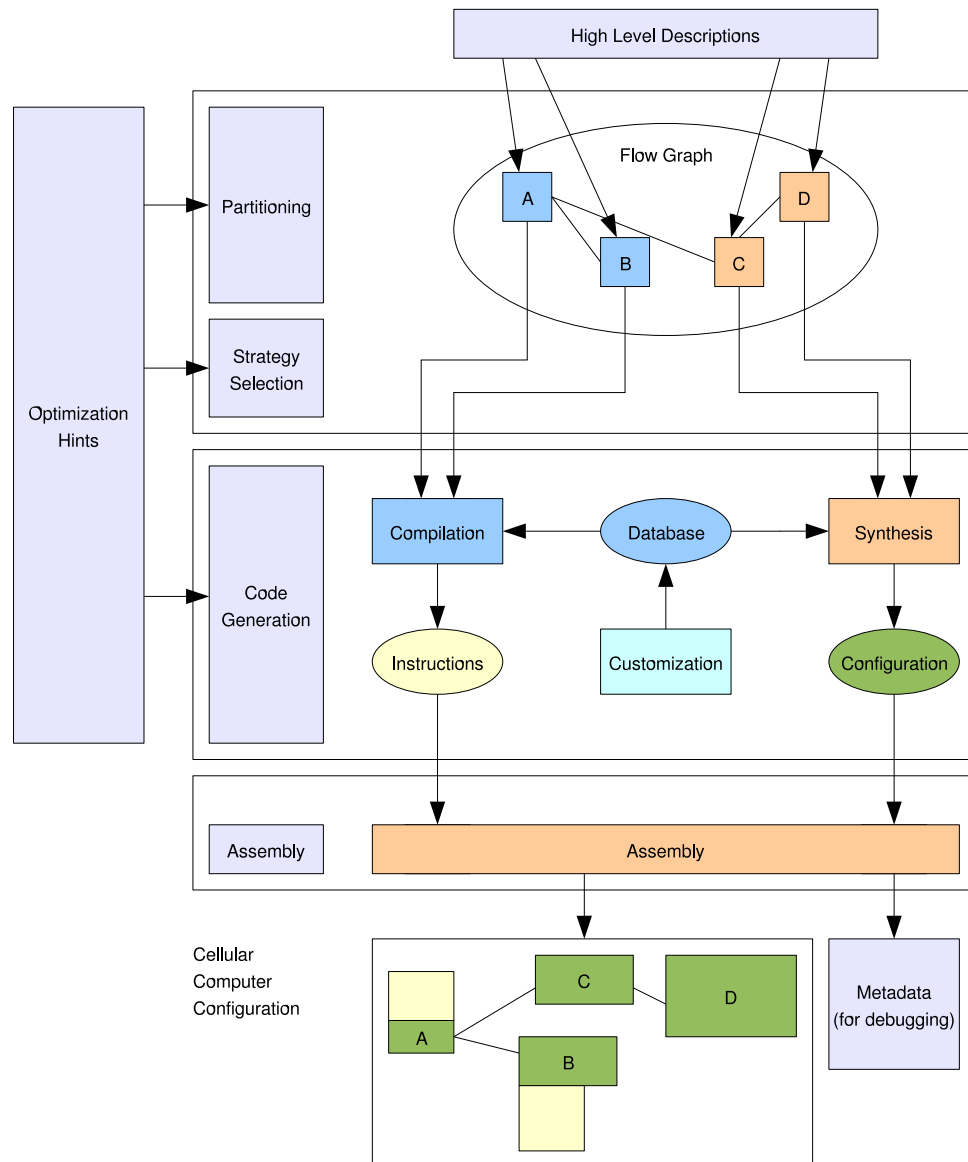


Figure 69. Cellular Software. The high level source code is analysed and partitioned to flow graph. For each component, a configuration strategy is selected; direct synthesis, direct compilation to existing processor model, or creating a custom processor and programming it. The components are then assembled to form a cellular computer configuration, which may be downloaded to a cellular computer.

5.2 Cellular Computer Operating System

In the modern computer software architecture, the coarse division of software is;

1. Applications, for various purposes,
2. Device drivers and firmware, for managing the computer peripherals and external interfaces, and
3. Operating system (OS), for managing the computer resources.

The core of the software architecture is the operating system. It lays down the fundamental principles of utilizing the computer hardware, and thus this section is devoted for analysing the nature of CCOS, Cellular Computer Operating System. One of the simplest methods would be configuring the entire cellular array to one sequential processor, and use an operating system like Linux. But that is not the solution discussed in this Thesis.

The fundamental tasks of operating systems in modern computers are;

- Managing the computer resources,
- Executing and terminating applications and device drivers,
- Serving applications, and
- Maintaining protection and security.

The cellular computer has three basic resources to manage; (1) cellular space, used for configuring applications, (2) boundary cells, which connect the cellular array to external peripherals, and (3) energy distribution.

The first two are relatively easy to understand. Cellular space is somewhat analogous to memory space in conventional computers, and boundary cells are special cellular locations analogous to I/O space or memory-mapped peripherals in conventional computers.

But energy distribution and monitoring is somewhat new resource to control for CCOS. What analogue it has in conventional computers? As the reader may remember, the performance of a cellular computer is not necessarily bound to the raw transaction speed of the cells, but the energy availability for performing transactions. Analogously, the energy is same to cellular computers as CPU time is for conventional computers. Cellular computers does not have CPU time sharing in the same manner as conventional computers have; instead of monitoring and sharing CPU time, the cellular computers need to monitor and share energy between applications.

With these observations, an initial design of a cellular computer OS and software architecture is shown in Figure 70. CCOS, responsible of configuring the array, has placed the device drivers near to the boundary cells they use for connecting to external devices. The

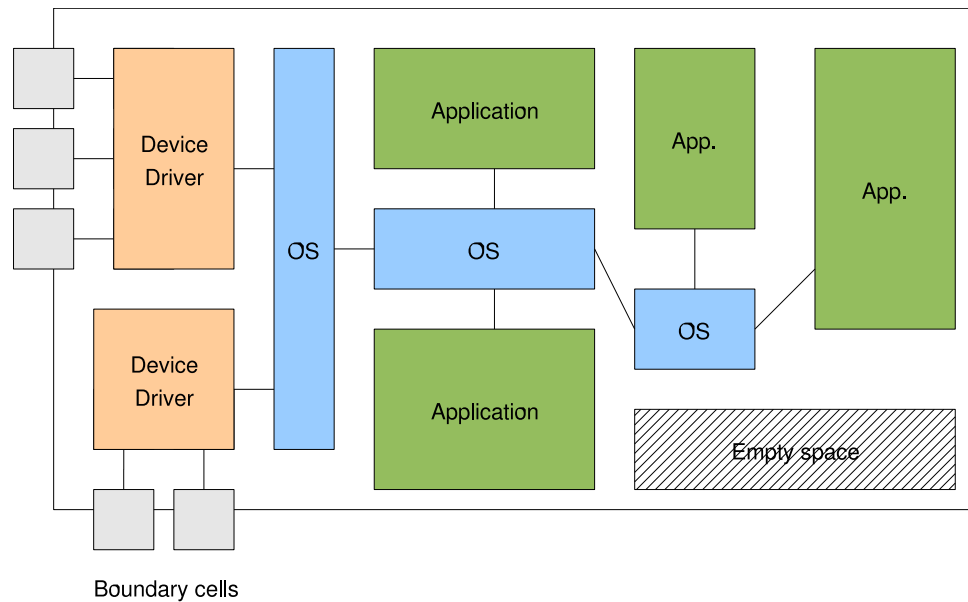


Figure 70. Cellular Computer OS (CCOS) Overview.

applications has been configured to free cellular space; the operating system needs to keep track about the cellular space usage, to be able to find free space for new configurations.

As in the modern operating systems, the applications are not directly connected to each other. Instead they have an access to OS, which can serve the requests made by the applications; for example, transferring data between applications and/or device drivers.

All this means that the backbone of the CCOS is a network connecting the applications, device drivers and CCOS parts together. In many ways, the CCOS is distributed operating system, and the internal architecture of the CCOS is a communicating network.

5.2.1 Application Management

The device drivers are somewhat permanent and static structures in the cellular space. But applications are more dynamic by their nature; they can come and go, be suspended and resumed, grow and shrink.

The cellular computer applications are fundamentally cellular configurations as all the cellular computer software. The applications are various combinations of parallel processing devices and programmable sequential processors or state machines; that is, the cellular software contains not only the structure of the devices, but also their data content.

The basic application services operating system should provide is to launch an application, and terminating one. From CCOS point of view, the application launching contains two phases; (1) finding a suitable free space for the application configuration, and (2) configuring the application to that space. The application termination basically means that the CCOS wipes the application from cellular space.

Since the cellular space definitely contain faulty cells, the reconfiguration engine needs tools for creating the defect maps. These maps could be created by reconfiguring the cellular space with special testing patterns before attempting to use it for application purposes. By performing placement and routing on the fly using the generated defect maps circumvents configuring faulty cells for applications (Figure 71).

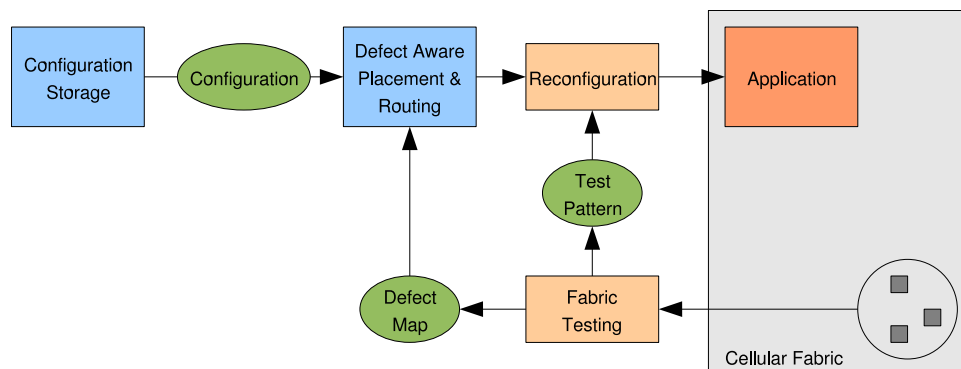


Figure 71. Reconfiguration flow. cf. Mishra&Goldstein, 2004, p. 83

Another fundamental operating system service for applications is to provide secured access to system services. In modern operating systems, the applications access the system at the low level using specialized system calls; these system calls form a “gate” to the underlying services. For this purpose, the CCOS provides a gateway, or a communication channel, to the applications to request services from the system.

One of the problem with the system gateway is to protect it against malfunctioning applications. The applications may produce an infinite stream of symbols to the gateway; the device controlling the gateway should have protection against such behaviour.

The CCOS need to have a control over energy feeding to the application. It is necessary to monitor the energy consumption of the application, and possibly limiting the energy in certain circumstances. Basically, the applications priorities, and the energy distribution is scheduled so that the higher priority applications get the energy they need over the lower priority applications.

Figure 72 shows a basic architecture of application management. The cellular space is divided to blocks. Each block has a trusted cellular device — called *block controller* — configured by CCOS to control reconfiguration and energy distribution, and to connect the application to the rest of the system. The block controllers are connected to backbone network routers, which are used to communicate between different parts of the system.

Since the block controllers can reconfigure the array, the applications in neighboring blocks could be connected together directly, without using backbone communication network. Similarly, it is possible to reconfigure more communication channels between routers for high-traffic links.

The cellular space management causes the similar needs as memory in conventional com-

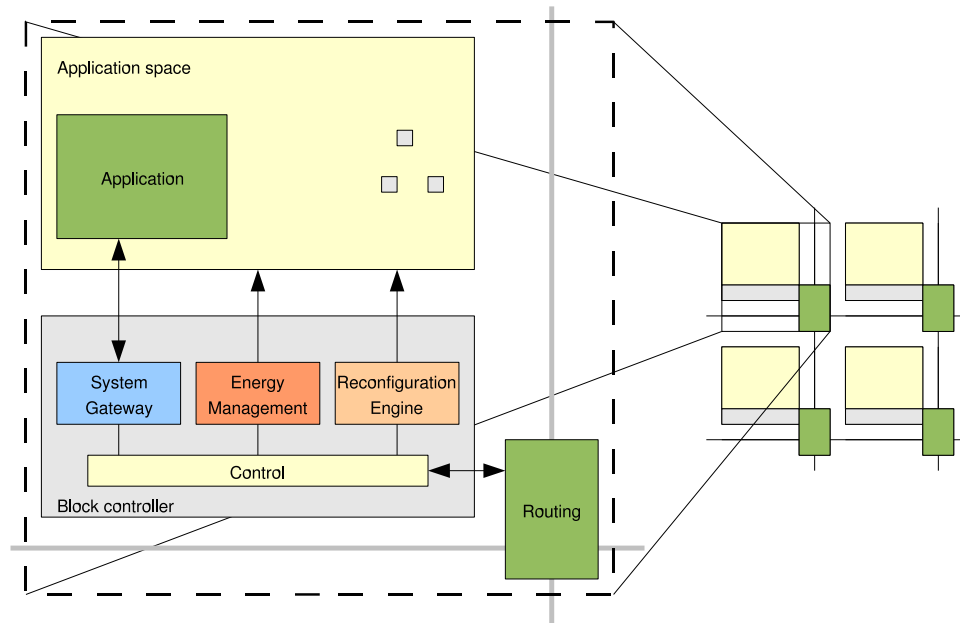


Figure 72. CCOS cellular block management.

puters. There may be need to swap the application inside the block, or there might be a need to move the application to another block.

5.2.2 Protection

Protecting the system is one of the fundamental functions of an operating system. The CCOS must protect itself and the applications against several different abusing attempts;

- Reconfiguration attempts; only the operating system should be able to change its structures and applications. The application may be able to change itself, but it should not be able to modify OS or other applications.
- *Eavesdropping*; the current cell model operates so that the cell itself can not decide, which cells can “hear” its computations. If two applications are in neighbor cells, the other can listen the data stream of the another; likewise, if the application and OS are using neighboring cells, the application may be eavesdropping. This should be prevented.
- Overloading OS; The application may try (intentionally or unintentionally) to overload the operating system by sending nonsense requests. The OS should have a traffic limiter, which prevents applications to constantly keep sending data to OS.
- OS service vulnerabilities; All requests must be checked against attempts to break the system.
- Unwanted application activity; an application, which performs infinite loop does not directly hurt the performance. But it does it indirectly by consuming energy,

which is then not available for more useful computations. So, the users and administration should have a `top` like tool to monitor the energy consumption (activity) of applications in the array. This has lead to a thought that maybe the energy distribution and heat removal should somehow to be tied to the cell programming, i.e. there would be ways to measure the activity of a cell block (application).

These different protection needs are examined closer in the following paragraphs.

Reconfiguration. The protection for reconfiguration can be implemented at array level. When configuring user applications, the configuration streams are checked against containing `config` operation; if the configuration stream contains these operations, the configuration attempt is failed, since no user application can contain these operations.

Because user applications can not contain the `config` operation, they are not able to reconfigure the array without the aid of OS (Operating System) and its services. Whenever the user application needs reconfiguration, it sends a request to OS. OS checks, that the requested configuration does not contain restricted operations and that it does not exceed the space reserved for configured application, see Figure 73.

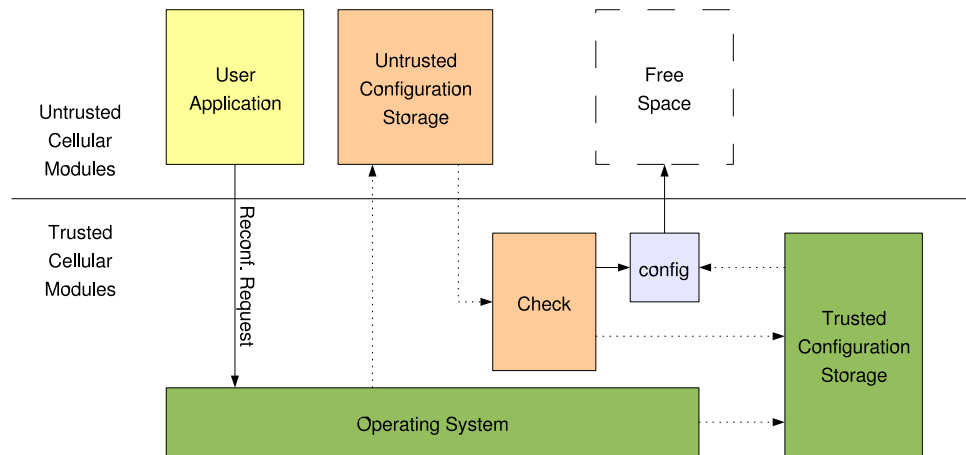


Figure 73. Reconfiguration protection mechanism. No user applications contain `config` operation to be able to reconfigure the array directly. Instead, they send a request to operating system. If the configuration is from untrusted source (e.g. from the application itself) it is checked first against containing the `config` operation or exceeding the limits of the free area reserved for the configuration. Configurations from trusted (prechecked) sources may pass the checking. The checked stream is fed to `config` operation in the trusted OS domain.

During the making of this Thesis, another reconfiguration protection mechanisms were also studied, see Appendix G.

Eavesdropping. The simplest way to protect the applications by hearing each other is to have an extra inactive one-cell *walls* between the applications, and between the applications and operating system structures.

Stream Overloading. In an asynchronous array, it is basically hard to determine when the stream sender is overloading the data paths. The reason for this is that the cell processing is non-deterministic; it may be idling long periods and thus the input queue gets longer. This Thesis can not make any suggestions for protecting the operating system against overloading; further studying may be needed.

Service Vulnerabilities. Whenever application makes a request to system it is necessary to check that the application has sufficient access rights for that operation. Due to vulnerabilities in the software, an application may be able to perform operations that are not allowed to it; in modern computer systems these are many times intentionally tried using array overflows and such. The same methods used in modern operating systems, e.g. static code checking, are also usable with cellular computer operating systems.

Unwanted Activity. The only way to detect the application activity is monitoring the energy used by the application, also indirectly by monitoring the heat generated by the application. The only way to limit the application activity is to control the energy feeding. To make this possible, the cellular array needs underlying hardware to support it.

Energy consumption causes heat to be generated. This should not be problem, since the cells need to have localized temperature control. There is of course possibility, that a malicious application overheats the cells near to crucial parts of the operating system, e.g. backbone channels, which then are overheated and stop working while cooling down. A sufficient isolation of applications and operating system is important for preventing this, too.

5.2.3 Post-Fabrication Processes

Based on the discussion in the previous sections, this section illustrates a possibly post-fabrication process for massive cellular arrays. The illustrations are in Figure 74, Figure 75 and Figure 76.

Initially after fabrication, the array is empty. Depending on the fabrication method, it can also contain random configurations; but in those cases, an attention should be paid for energy consumption of random configurations — otherwise the initial configuration process may be impossible.

The post-fabrication configuration starts by attaching an external device to the array. This external device injects a self-replicating operating system *seed* to array, by extrenally forcing the boundary cell to reconfiguration state.

The operating system itself is a regular repeating pattern of self-replicating block controllers and communication lines between those. The block controller is later responsible for configuring applications to the part of the array it controls. It may not be necessary to fully fill the array during initial setup — operating system may extend later at runtime, when the array resources are used.

After the initial block controller is configured to the array, the external formatter continues downloading *firmware*. This firmware contains cellular software necessary for operating system — in fact, it contains the operating system, the block control structure downloaded earlier is just for resource utilization purposes. Finally, the firmware is downloaded and the operating system is configured (fully or partial configuration).

At the user's side, the array is connected to external world using some kind of motherboard. The device drivers needed for the devices in the motherboard, as well as user applications and their storages are configured during setup at user's side. Later, the free cellular resources are used for storing user's data and applications.

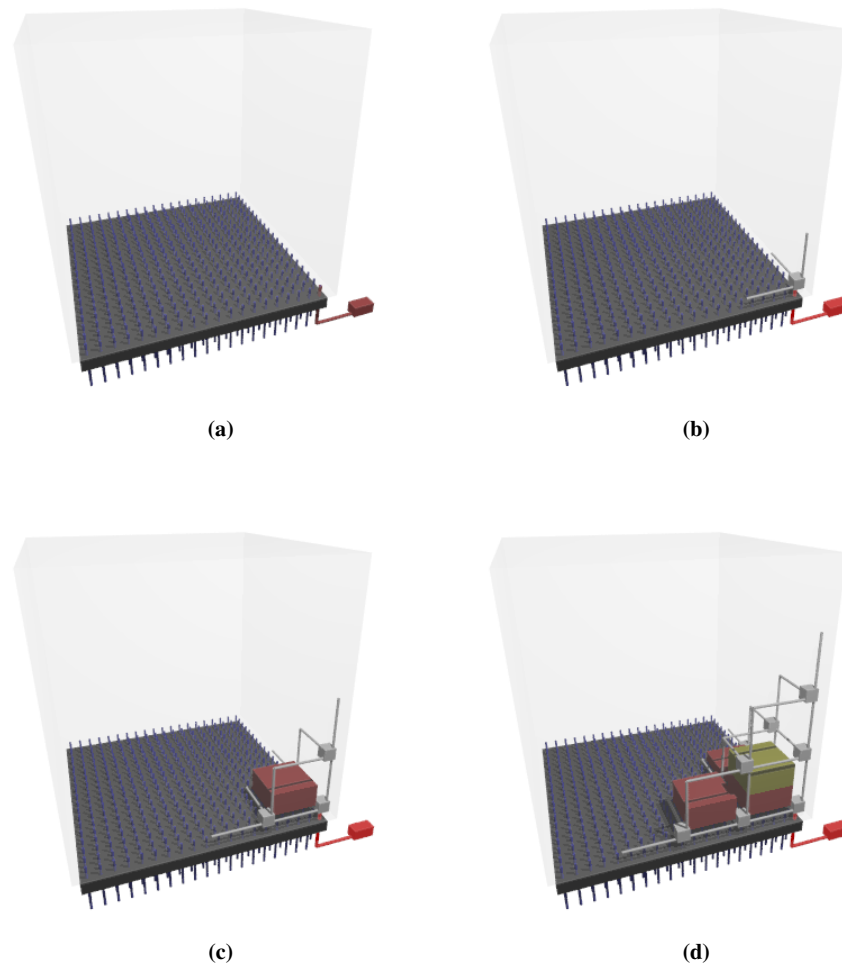


Figure 74. Cellular Computer System (1). a) After fabrication the array is empty. b) External configuration device, *formatter*, injects the operating system control structure (OSCS) *seed* to the array. c) When the initial structure is finalized, it starts replicating. At the same time, the external formatter starts to download firmware. d) The formatter continues loading firmware and OSCS continues replication.

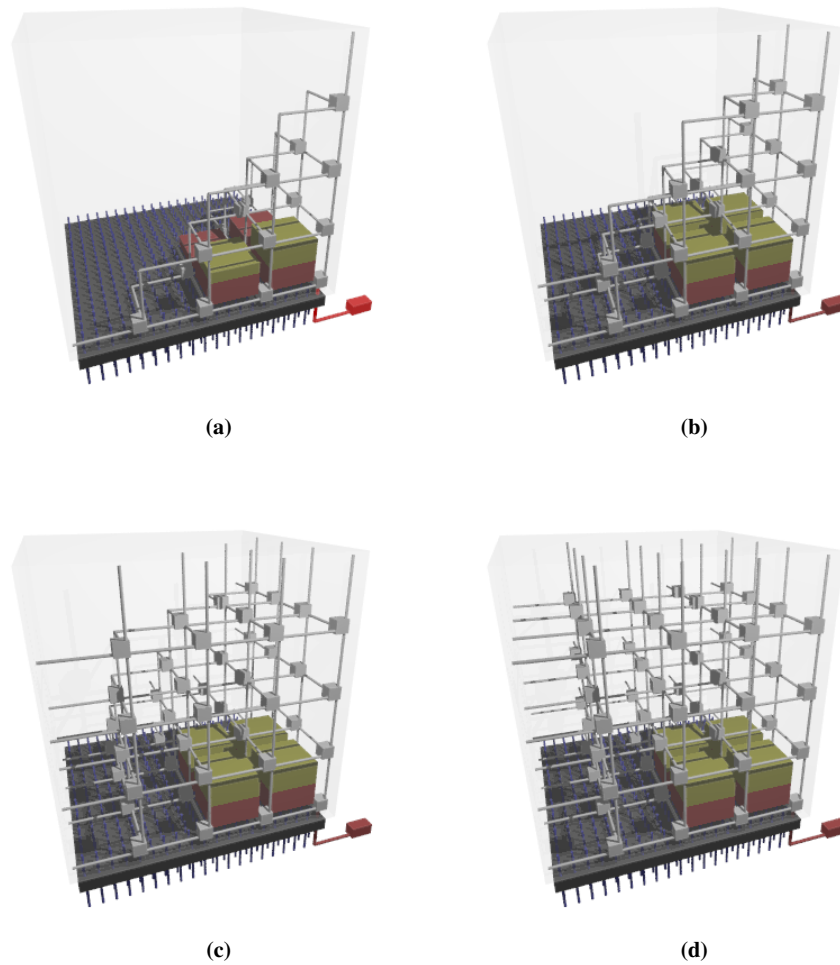


Figure 75. Cellular Computer System (2). a) The formatter downloads firmware and OSCS replicates. b) The formatter has finished the firmware downloading and it deactivates. The OSCS replicates, until at d) it has filled the array. The array is now ready to be shipped to users.

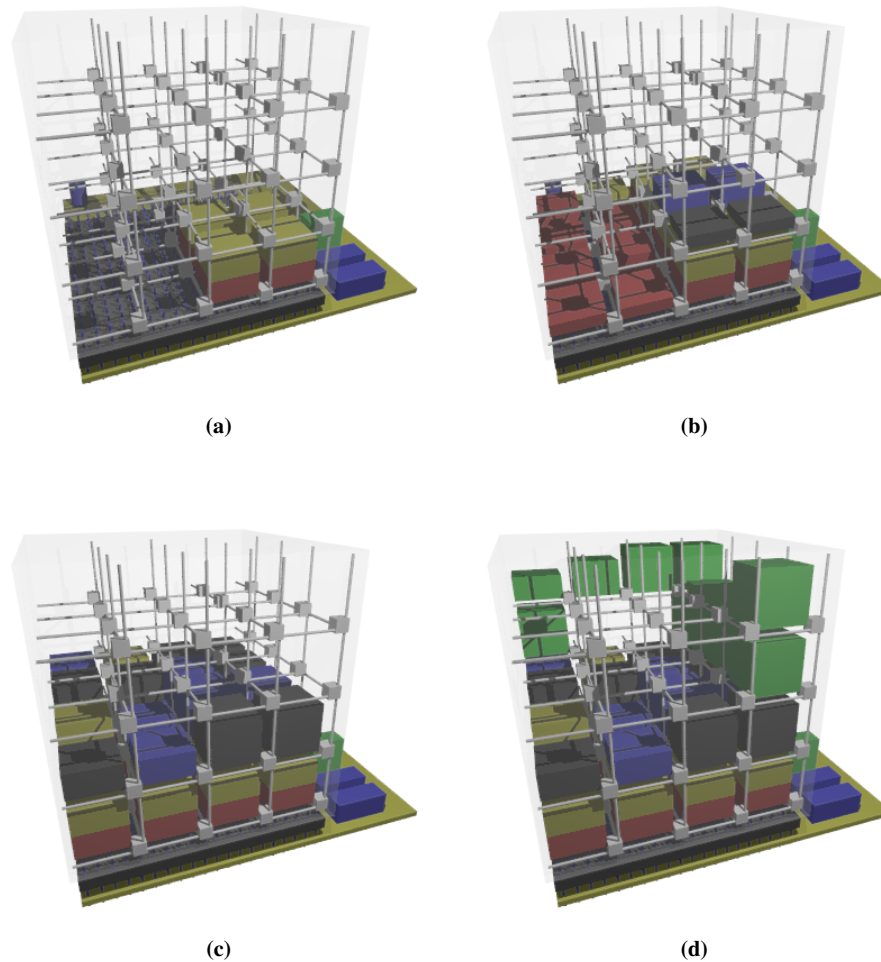


Figure 76. Cellular Computer System (3). a) At the user side, the array is attached to motherboard, containing external interfaces. b) The motherboard contain devices, which require drivers. These drivers are configured by the operating system. They are located in the figures at the bottom, where the cells connected to external world are located. c) The applications (blue boxes) and their storages (black boxes) are configured. d) During operation, the user data (green boxes) is located to the array.

6. Conclusions

The three research questions were:

1. What kind of physical structure a cellular computer has?
2. How cellular computer is built as general purpose computer?
3. How to implement and organize operating system and software to a cellular computer?

As an answer for first research question, there are strong evidence that the future computers are large cellular arrays constructed from nanoscale devices, and that so called nanocomputers are inherently cellular arrays. Constructing a very large cellular array raises problems with energy, heat dissipation and fault tolerance.

As discussed about these factors in Chapter 2, it is suggested that the three most important characteristics of future cellular computers are (1) self-reconfigurability, (2) three-dimensional layout, and (3) asynchronously operating cells.

Exploiting three spatial dimensions with nanoscale technology is essential for making the physical size of the computer practical. Self-reconfigurability is essential for two main purposes; (1) making the computer programmable, i.e. general purpose, and (2) allowing use of software-based methods for implementing tolerance against defects and faults. Furthermore, self-reconfigurability is also essential to have distributed, parallel reconfiguration mechanism. Asynchronicity gives the cellular array tolerance against temporal variations in intercell transactions, which is essential for controlling the energy consumption and heat dissipation in distributed, parallel manner, suitable for massively parallel machine.

As an answer to second research question, a general purpose cellular computer has two important theoretical characteristics: (1) it is computationally universal, i.e. Turing complete, and (2) it is constructionally universal. There are many possible sets of operations which make the cellular computer both computationally and constructionally universal. In this Thesis, one set of operations were implemented. Its main objective was to be easy enough to be used for constructing functional cellular blocks manually. With an aid of a compiler, the set can be certainly made more efficient.

The developed operation set was experimentally tested by using it to implement a selected set of programs. Emulation of Turing machine, more specifically Busy Beaver 501, was demonstrating the relationship between the constructed cellular machine and Turing ma-

chine. Self-replicating device was demonstrating the self-reconfiguration capabilities of the constructed machine.

Solving the Eight Queen's Problem with two different ways was demonstrating the possibilities for using the cellular machine for solving computational problems. The Eight Queen's Problem is a well-known example of simple, but non-trivial computational problem, and thus a good test for the capabilities of constructing software to cellular machine.

As an answer to third research question, the software development and architecture of the cellular machine was examined using the information gained from experiments, and combining it with the existing knowledge of the computers, software and operating systems, and the research of the software development for reconfigurable platforms.

In this Thesis, it is predicted that as with the reconfigurable computers, the methods of developing software are mainly selected by criterias of human productivity, and the underlying hardware architecture does not set restrictions for methods due to ever-improving compiler technology.

The organization of the software in a running machine, as well as its operating system, would greatly differ the ones used in sequential machines. The main reason for this is that the sequential and cellular computer operating systems have different set of resources to manage. Cellular computer operating system manages energy, cellular space, access to boundary cells and the connections between cellular blocks.

The necessary protection mechanism has the same principle than in regular sequential computer system, but it is implemented differently. It is necessary to control the access to reconfiguration cells, prevent cellular blocks to interfere with each other and protect the operating system itself against malfunctioning cellular blocks.

Furthermore, the operating system needs to be distributed and parallel, which makes it essentially a network of blocks controlling resources and communicating with each other. Centralized solutions does not fit well to highly parallel, distributed computer system.

7. Further Research

The cellular computer model created for this Thesis definitely needs refinements to match better to the characteristics of forthcoming hardware, and to match better to the findings from the experiments and ideas of the software architecture. For example, these refinements could be considered;

- Defining a fully reversible set of operations, and studying the computation problem solving with it.
- Simplifying — even minimizing — the operation set; this would greatly help attempts to synthesise the model either to cellular hardware (like FPGA), or to some existing cellular automaton (like Peper’s asynchronous CA).
- As the experiments show, the cellular structures are dominated by memory and routing elements. This leads to a question, if the nanocomputer cell should match this observation, i.e. the cell would consume spatial area to implement flexible routing capabilities and large amounts of memory in place of logic capabilities.

String-Based Computation. The string-based computation model represented in this Thesis is somewhat unique, or at least there were no mentions of such in the examined literature, so a careful and critical examination of it would be required.

Initially it was expected that using strings instead of parallel data paths would harm the performance, but in fact the penalty may be small. Most of the performance drops are caused by transferring the data between the parts of the construction. The routes carry the string in pipelined manner, so the delay of receiving a string is just slightly larger than receiving the first symbol. More importantly, string-based computing makes the constructions smaller, and that means that the paths are shorter, so it is possible that string-based computation is in fact faster than its parallel counterparts.

The closest resembling computation model found in the literature was so called Message Passing Systems (MPS), which are currently studied under parallel computation, supercomputers and they have been found useful for different kind of distributed systems. The formalization of MPS is known as *Actor Model*, a formal representation of messages and processes, originally developed by Hewitt (1977).

Larger Constructions. In the case that the cellular computer model represented in this Thesis is found useful by the scientific community, experiments of larger cellular structures might be considered. In prior to make these, the toolset would need enchantments (Figure 77). The most valuable tool would be automated placement and routing tool, because that is the most laborous task when creating structures manually. Other refinements

to toolset would be a compiler to translate high-level descriptions (e.g. C or VHDL) to cellular operations. Also, the simulating capabilities could be extended.

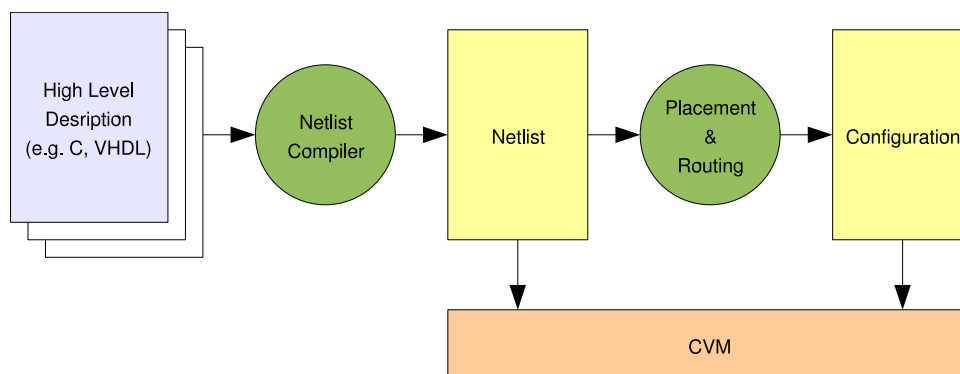


Figure 77. A more complete toolset.

There may be already existing open source software developed for generating FPGA configurations, which might be modified to be used with the cellular computer model. A useful starting point could be Dimitri Yatsenko's Master Thesis "Automated Placement and Routing of Cell Matrix Circuits" (Yatsenko, 2003).

Instruction Streams. Other than dataflow computing was scoped out from this Thesis according to the observations of Fountain *et al.* (1998), but they might be very practical in certain circumstances. Depending on the underlying technology, it could be possible to reconfigure the cell as a simple processor. Instead of retrieving data tokens, the token stream would be interpreted as a sequence of instructions, in a same way that the RAM memory element was implemented in the models. This could enable an easy way to perform operations for stored data.

Asynchronous Base Technology. It is of course important to further study possibilities for making large arrays. But amongst all possible technologies, the ones which are asynchronous by their nature have special interest. Is it possible, and how, to implement device technology, which operates in the same manner than the cell model presented in this Thesis - asynchronously, processing streams of symbols? A DNA-like processing, where the data string is represented as a molecular chain seems interesting option. There are no reasons at least at the moment to suspect that fundamentally asynchronous, delay-insensitive processing would be impossible with other base device technologies.

Realtime Requirements vs. Asynchronous Design. If the software has realtime requirements, how are they dealt with a system, which is inherently non-deterministic? This question was thought during the making of this Thesis without finding a sufficient answer.

Distributed Computing. Distributed computing is decentralised and parallel computing, using two or more computers communicating over a network to accomplish a common objective or task. The types of hardware, programming languages, operating systems and other resources may vary drastically. It is similar to computer clustering with the main

difference being a wide geographic dispersion of the resources.

Data-Flow Computing. As the cellular model follows a paradigm of data flow machine, those were explored during the making of this Thesis for finding out possible ideas. Interestingly, none of the papers end up to the references; it seems that even using a data flow computer, the data flow as programming paradigm was not useful — why?

References

- Agarwal, A. 1999, *Raw Computing*, RAW architecture presentation in Scientific American.
- Amerson, R., Carter, R., Culbertson, B., Kuekes, P. & Snider, G. 1995, *Teramac — configurable custom computing*, Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines (Napa, CA) (D. A. Buell and K. L. Pocek, eds.), April 1995, pp. 32 - 38
- Barat, F., Lauwereins, R. & Deconinck, G. 2002, *Reconfigurable Instruction Set Processors from A Hardware/Software Perspective*, Software Engineering, IEEE Transactions on Volume 28, Issue 9, Sept. 2002 pp. 847 - 862
- Baumgarte, V., Ehlers, G., May, F., Nückel, A., Vorbach, M. & Weinhardt, M. 2003, *PACT XPP — A Self-Reconfigurable Data Processing Architecture*, The Journal of Supercomputing, Vol. 26, Number 2, September 2004, pp. 167 - 184.
- Beckett, P. & Jennings, A. 2002, *Towards Nanocomputer Architecture*, Proceedings of the seventh Asia-Pacific conference on Computer systems architecture, Vol. 6, pp. 141 - 150
- Bellows, P. & Hutchings, B. 1998, *JHDL — an HDL for Reconfigurable Systems*, FPGAs for Custom Computing Machines, 1998. Proceedings in IEEE Symposium on 15-17 April 1998 pp. 175 - 184
- Bishop, P. & Sullivan, C. 2003, *A Reconfigurable Future*, Field-Programmable Technology (FPT), 2003. Proceedings in IEEE International Conference on 15-17 Dec. 2003, pp. 2 - 7.
- Brebner, G. 1999, *Tooling up for reconfigurable system design.*, Reconfigurable Systems (Ref. No. 1999/061), IEEE Colloquium on 10 March 1999, pp. 2/1 - 2/4
- Böhm, W., Hammes, J., Draper, B., Chawathe, M., Ross, C., Rinker, R. & Najjar, W. 2002, *Mapping a Single Assignment Programming Language to Reconfigurable Systems*, The Journal of Supercomputing, Springer Science+Business Media B.V., Vol. 21, Num. 2, February 2002, pp. 117 - 130.
- Callahan, T. 2003, *Kernel Formation in Garpcc*, Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on 9-11 April 2003, pp. 308 - 309.
- Callahan, T., Hauser, J. & Wawrzynek, J. 2000, *The Garp architecture and C compiler*, IEEE Computer Volume 33, Issue 4, April 2000, pp. 62 - 69.

- Cardoso, J. & Neto, H. 2003, *Compilation for FPGA-based Reconfigurable Hardware*, Design & Test of Computers, IEEE Volume 20, Issue 2, March-April 2003, pp. 65 - 75.
- Cardoso, J. & Weinhardt, M. 2002, *Fast and Guaranteed C compilation onto the PACT-XPPTM Reconfigurable Computing Platform*, Field-Programmable Custom Computing Machines, 2002. Proceedings in 10th Annual IEEE Symposium on 22-24 April 2002, pp. 291 - 292.
- Cardoso, J. & Weinhardt, M. 2003, *From C Programs to The Configure-Execute Model*, Design, Automation and Test in Europe Conference and Exhibition, 2003, pp. 576 - 581.
- Carroll, S. 2002, *A Complete Programming Environment for DNA Computation*, First Workshop on Non-Silicon Computation (NSC-1), February 3, 2002, Cambridge MA in conjunction with 8th International Symposium of High-Performance Computer Architectures
- Chaudhary, A., Chen, D., Hu, X. S., Whitton, K., Niemier, M. & Ravichandran, R. 2005, *Eliminating wire crossings for molecular quantum-dot cellular automata implementation*, Computer-Aided Design, 2005. ICCAD-2005. IEEE/ACM International Conference on Nov. 6-10, 2005 pp. 565 - 572
- Compton, K. & Hauck, S. 2002, *Reconfigurable Computing: A Survey of Systems and Software*, ACM Computing Surveys, 2002.
- Cook, M. 2004, *Universality in Elementary Cellular Automata*, Complex Systems, vol. 15, pp. 1 - 40
- Dewdney, A. K. 1985, *Computer Recreations*, Scientific American, vol. 252, no. 4, april 1985, pages 12-16
- Diessel, O. & Milne, G. 2001, *Hardware Compiler Realising Concurrent Processes in Reconfigurable Logic*, IEEE Proceedings of Computers and Digital Techniques, Vol. 148, Issue 45, July-Sept. 2001, pp. 152 - 162.
- Digital Mars 2007, *D Programming Language 2.0*, Online Publication, available: <http://www.digitalmars.com/d/>
- Drexler, K. E. 1986, *Engines of Creation — The Coming Era of Nanotechnology*, Anchor Books, ISBN 0385199732
- Drexler, K. E. 1998, *Nanosystems: Molecular Machinery, Manufacturing, and Computation*, Wiley Interscience, ISBN 0-471-57518-6
- Durbeck, L. J. K. 2001, *An Approach to Designing Extremely Large, Extremely Parallel Systems*, Abstract of a talk given at The Conference on High Speed Computing, Salishan Lodge, Gleneden, Oregon, U.S.A., April 26 2001
- Durbeck, L. J. K. & Macias, N. J. 2000, *The Cell Matrix: An Architecture for Nanocomputing*, Cell Matrix Corporation; also published in Nanotechnology (2001), pp. 217 - 230

- Ebergen, J. 1987, *Translating Programs into Delay-insensitive Circuits*, Dissertation, Eindhoven University of Technology, Department of Computing Science, October 1987
- Ebergen, J. 1991, *A Formal Approach to designing delay-insensitive circuits*, Distributed Computing 5, 107-19
- EDIS 2007, *Encyclopedia of Delay-Insensitive Systems (EDIS)*, Available: <http://edis.win.tue.nl/edis.html>
- Estrin, G., Bussell, B., Turn, R. & Bibb, J. 1963, *Parallel Processing in A Restructurable Computer System*, IEEE Transactions on Electronic Computing, 1963 pp. 747 - 755.
- Feynman, R. P. 1959, *There's Plenty of Room at The Bottom*, Speech hold on December 29th 1959 at annual meeting of American Physical Society. First published: Engineering and Science, California Institute of Technology (CalTech)
- Fountain, T., Duff, M., Crawley, D., Tomlinson, C. & Moffat, C. 1998, *The Use of Nanoelectronic Devices in Highly-Parallel Computing Systems*, IEEE Transactions on VLSI Systems, Vol. 6, pp. 31-38
- Frank, M. P. 2003a, *Nanocomputer Systems Engineering*, Nanotech 2003 Vol. 2
- Frank, M. P. 2003b, *Nanocomputers - Theoretical Models*, Invited Article (Review Chapter) for the Encyclopedia of Nanoscience and Nanotechnology, American Scientific Publishers, 2003
- Frank, M. P. 2005, *Approaching the Physical Limits of Computing*, Find it out
- Frank, M. P. & Knight, T. F. 1998, *Ultimate theoretical models of nanocomputers*, Nanotechnology, 1998, Vol. 9, Issue 3, pp. 162-XXX.
- Goldstein, S. C. 2005, *The impact of the nanoscale on computing systems*, Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design ICCAD '05
- Goldstein, S. C., Schmit, H., Budiu, M., Cadambi, S., Moe, M. & Taylor, R. R. 2000, *PipeRench: A Reconfigurable Architecture and Compiler*, IEEE Computer, Volume 33, Issue 4 (April 2000), pp. 70 - 77
- Graham, P. & Gokhale, M. 2004, *Nanocomputing In The Presence of Defects and Faults: A Survey*, S.K. Shukla and R.I. Bahar (eds), Nano, Quantum and Molecular Computing, pp. 39 - 72. Kluwer Academic Publishers
- Hartenstein, R. 1997, *The Microprocessor is no more General Purpose (invited paper)*, in Proceedings of IEEE ISIS, Austin, Texas, 1997.
- Hartenstein, R. 2001, *A Decade of Reconfigurable Computing: a Visionary Retrospective*, Design, Automation, and Test in Europe, 2001. Proceedings of the conference on Design, pp. 642 - 649

- Hartenstein, R. 2002, *Trends in reconfigurable logic and reconfigurable computing*, Electronics, Circuits and Systems, 2002. 9th International Conference on Volume 2, 15-18 Sept. 2002 pp. 801 - 808
- Hartenstein, R. 2003, *Are we really ready for the breakthrough?*, Parallel and Distributed Processing Symposium, 2003. Proceedings. International 22-26 April 2003 p. 7
- Hauck, S. 1995, *Asynchronous Design Methodologies: An Overview*, Proceedings of the IEEE, Vol. 83, No. 1, pp. 69-93, January 1995
- Hauser, J. & Wawrzynek, J. 1997, *Garp: A MIPS Processor with A Reconfigurable Co-processor*, FPGAs for Custom Computing Machines, 1997. Proceedings in The 5th Annual IEEE Symposium on 16-18 April 1997, pp. 12 - 21.
- Heath, J., Kuekes, P., Snider, G. & Williams, R. S. 1998, *A Defect-Tolerant Computer Architecture: Opportunities for Nanotechnology*, Science (1998), pp. 1716 - 1721
- Hewitt, C. 1977, *Viewing Control Structures as Patterns of Passing Messages*, Journal of Artificial Intelligence
- Hillis, W. D. 1989, *The Connection Machine*, The MIT Press series in artificial intelligence, ISBN 0-262-58097-7.
- Hopf, J. & Kearney, D. 2003, *Specification and Integration of Software And Reconfigurable Hardware Using Hardware Join Java (HJJ)*, Proceedings of IEEE International Conference on Field-Programmable Technology (FPT), Dec. 2003, pp. 379 - 382.
- Iseli, C. & Sanchez, E. 1993, *Spyder: A Reconfigurable VLIW Processor Using FPGAs*, Proceedings in IEEE Workshp on FPGAs on Custom Computing Machines, April 1993, pp. 17 - 24.
- Ito, H., Konishi, R., Oguri, K. *et al.* 2003, *Dynamically Reconfigurable Logic LSI - PCA - I*, IEICE Transactions on Information and Systems, Vol. E86-D, No. 5, pp. 859-867.
- Keller, R. M. 1974, *Towards A Theory of Universal Speed-Independent Modules*, IEEE Transactions on Computing, C-23, pp. 21-33
- Kimura, S., Yukishita, M., Itou, Y., Nagoya, A., Hirao, M. & Watanabe, K. 1997, *A Hardware/Software Codesign Method for A General Purpose Reconfigurable Co-Processor*, Hardware/Software Codesign, 1997 (CODES/CASHE'97), proceedings of the Fifth International Workshop on 24-26 March 1997, pp. 147 - 151.
- Konishi, R., Ito, H., Nakada, H. *et al.* 2001, *PCA-I: A Fully Asynchronous, Self-Reconfigurable LSI*, Proceedings of the 7th International Symposium on Asynchronous Circuits and Systems, 2001.
- Kuhn, A. & Huss, S. 2004, *Dynamically Reconfigurable Hardware for Object-Oriented Processing*, Parallel Computing in Electrical Engineering, 2004. PARLEC 2004. International Conference on 7-10 Sept. 2004, pp. 181 - 186.

- Landauer, R. 1961, *Irreversibility and heat generation in the computing process*, IBM J. Res. Dev. 5 183-91 (1961)
- Langton, C. G. 1984, *Self-Reproduction in Cellular Automata*, Physica D
- Lee, T., Derbyshire, A., Luk, W. & Cheung, P. 2003, *High-Level Language Extensions for Run-Time Reconfigurable Systems*, Proceedings of IEEE International Conference on Field-Programmable Technology (FPT), Dec. 2003, pp. 144 - 151.
- Likharev, K. K. 1996, *Rapid Single Flux Quantum (RSFQ) Superconductor Electronics*, Online:
<http://pavel.physics.sunysb.edu/RSFQ/Research/WhatIs/rsfqwte1.h>
- Likharev, K. K. & Semenov, V. K. 1991, *RSFQ Logic/Memory Family: A New Josephson-Junction Technology for Sub-Terahertz-Clock-Frequency Digital Systems*, IEEE Transactions on Applied Superconductivity, Vol. 1, Issue 1, March 1991, pp. 3 - 28
- Ludewig, J., Schult, U. & Wankmüller, F. 1983, *Chasing the Busy Beaver - Notes and Observations on a Competition to Find the 5-state Busy Beaver*, Forschungsberichte des Fachbereichs Informatik, 159, Universitaet Dortmund, Dortmund, 1983
- Macias, N. J. 1999, *The PIG Paradigm — The Design and Use of a Massively Parallel Fine Grained Self-Reconfigurable Infinitely Scalable Architecture*, Proceedings of the 1st NASA/DOD workshop on Evolvable Hardware, p.175, July 19-21, 1999
- Mange, D., Stauffer, A., Petraglio, E. & Tempesti, G. 2003, *Self-replicating loop with universal construction*, submitted to Elsevier Preprint, 12 September 2003.
- Maresca, M. & Baglietto, P. 1993, *A Programming Model for Reconfigurable Mesh Based Parallel Computers*, Proceedings of Programming Models for Massively Parallel Computers, 20-23 Sept. 1993, pp. 124 - 133.
- Martin, A. J. 1990, *The Limitations to Delay-insensitivity in Asynchronous Circuits*, Proceedings 6th MIT Conference on Advanced Research in VLSI (Cambridge, MA: MIT Press), pp. 263-278
- Marxen, H. & Buntrock, J. 1990, *Attacking the Busy Beaver 5*, Bulletin of the EATCS, 40, pages 247-251, February 1990
- Merkle, R. 1993, *Two Types of Mechanical Reversible Logic*, Nanotechnology, Volume 4, 1993, pp. 114 - 131
- Mishra, M. & Goldstein, S. 2004, *Defect Tolerance At The End of The Roadmap*, S.K. Shukla and R.I. Bahar (eds), Nano, Quantum and Molecular Computing, pp. 73 - 108. Kluwer Academic Publishers
- Niemier, M. & Kogge, P. 1999, *Logic in wire: using quantum dots to implement a micro-processor*, Electronics, Circuits and Systems, 1999. Proceedings of ICECS '99. The 6th IEEE International Conference on Volume 3, 5-8 Sept. 1999 pp. 1211 - 1215 vol.3

- Niemier, M. & Kogge, P. 2004a, *Exploring and exploiting wire-level pipelining in emerging technologies*, Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on 30 June-4 July 2001 pp. 166 - 177
- Niemier, M. & Kogge, P. 2004b, *The "4-diamond circuit" - a minimally complex nano-scale computational building block in QCA*, VLSI, 2004. Proceedings. IEEE Computer society Annual Symposium on 19-20 Feb. 2004 pp. 3 - 10
- Niemier, M., Kontz, M. & Kogge, P. 2000, *A design of and design tools for a novel quantum dot based microprocessor*, Design Automation Conference, 2000. Proceedings 2000. 37th June 5-9, 2000 pp. 227 - 232
- Niemier, M., Ravichandran, R. & Kogge, P. 2004, *Using circuits and systems-level research to drive nanotechnology*, Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on 11-13 Oct. 2004 pp. 302 - 309
- Patra, P. 1996, *Approaches to Design of Circuits for Low-Power Computation*, Doctoral Thesis, University of Texas at Austin
- Patwardhan, J. P., Dwyer, C., Lebeck, A. R. & Sorin, D. J. 2006, *NANA: A nano-scale active network architecture*, ACM Journal on Emerging Technologies in Computing Systems (JETC), Volume 2 Issue 1, January 2006
- Peper, F., Lee, J., Adachi, S. & Mashiko, S. 2003, *Laying out circuits on asynchronous cellular arrays: a step towards feasible nanocomputers?*, Nanotechnology 14, pp. 469-485.
- Peper, F., Lee, J., F., A., T., I., Adachi, S., Matsui, N. & Mashiko, S. 2004, *Fault Tolerance in Nanocomputers: A Cellular Array Approach*, IEEE Transactions on Nanotechnology, Vol. 3, No. 1, March 2004
- Perrier, J., Sipper, M. & Zahnd, J. 1996, *Toward Viable, Self-Reproducing Universal Computer*, Physica D, Vol. 97, Issue 4 (October 1996), pp. 335 - 352
- Radó, T. 1962, *On Non-Computable Functions*, Bell Systems Tech. J. 41, 3, May 1962
- Randall, P. 2001, *A Turing Machine In Conway's Game of Life*, Online:
http://www.cs.ualberta.ca/~bulitko/F02/papers/tm_words.pdf
- Ravichandran, R., Niemier, M. & Lim, S. K. 2005, *Partitioning and placement for buildable QCA circuits*, Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific Volume 1, 18-21 Jan. 2005 pp. 424 - 427
- Rincon, F. & Teres, L. 1998, *Reconfigurable Hardware Systems*, Proceedings in Semiconductor Conference, 1998. CAS'98 International, Vol. 1, 6-10 Oct. 1998, pp. 45 - 54.
- Roweis, S., Winfree, E., Burgoyne, R., Chelyapov, N. V., Goodman, M. F. & Rothmund, P. W. K. 1996, *A Sticker Based Model for DNA Computation*, DNA Based Computers: DMACS Workshop, 1996

- Savage, J. E. 1998, *Models of Computation — Exploring The Power of Computing*, Addison Wesley Longman, ISBN 0-201-89539-0
- Sipper, M. 1999, *The Emergence of Cellular Computing*, IEEE Computer, Volume 32, Issue 7, July 1999 pp. 18 - 26
- Sipper, M. 2004, *Evolution of Parallel Cellular Machines — The Cellular Programming Approach*, Originally published by Springer, 1997
- Smith, M., Drager, S., Pochet, L. & Peterson, G. 2001, *High Performance Reconfigurable Computing Systems*, Proceedings of 2001 IEEE Midwest Symposium on Circuits and Electronics.
- Strelzoff, A. 2004, *Functional Programming for Reconfigurable Computing*, Proceedings in Parallel and Distributed Processing Symposium, 2004. 18th International, 26-30 April 2004, p. 151.
- Taniguchi, N. 1974, *On The Basic Concept of 'Nano-Technology'*, Proceedings to International Conference of Production Engineering, Tokyo, Part II, Japan Society of Precision Engineering
- Taylor, M. B. 2003, *Comprehensive Specification for The Raw Processor*, Cambridge, MA.
- Taylor, M. B., Kim, J., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffmann, H., Johnson, P., Lee, J.-W., Lee, W., Ma, A., Saraf, A., Seneski, M., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S. & Agarwal, A. 2002, *The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs*, IEEE Micro.
- Teifel, J. & Manohar, R. 2004a, *An asynchronous dataflow FPGA architecture*, IEEE Transactions on Computers, Nov. 2004, Vol. 53, Issue 11, pp. 1376- 1392
- Teifel, J. & Manohar, R. 2004b, *Static tokens: using dataflow to automate concurrent pipeline synthesis*, Proceedings to 10th International Symposium on Asynchronous Circuits and Systems, 19-23 April 2004, pp. 17- 27
- Todman, T., Constantinides, G., Wilton, S., Mencer, O., Luk, W. & Cheung, P. 2005, *Reconfigurable Computing: Architectures And Design Methods*, Computers and Digital Techniques, IEEE Proceedings, Vol. 152, Issue 2, Mar 2005, pp. 193 - 207.
- Turing, A. M. 1936, *On Computable Numbers, With An Application to The Entscheidungsproblem*, Proceedings of the London Mathematical Society, Series 2, Vol.42 (1936 - 37) pp. 230 - 265, with corrections from Proceedings of the London Mathematical Society, Series 2, Vol.43 (1937) pp. 544 - 546.
- Vaishnavi, V. & Kuechler, B. 2005, *Design Research in Information System*, Association for Information System, June 2005. Online: <http://www.isworld.org/Researchdesign/drisISworld.htm>

Vankamamidi, V., Ottavi, M. & Lombardi, F. 2005, *Nano and Emerging Technologies: Tile-based design of a serial memory in QCA*, April 2005 Proceedings of the 15th ACM Great Lakes symposium on VLSI

Vitányi, P. 1988, *Locality, communication and interconnect length in multicomputers*, SIAM J. on Computing 17, pp. 659 - 672

von Neumann, J. 1966, *The Theory of Self-reproducing Automata*, A. Burks, ed., Univ. of Illinois Press, Urbana, IL.

Vuletic, M., Pozzi, L. & Ienne, P. 2004, *Programming Transparency and Portable Hardware Interfacing: Towards General-Purpose Reconfigurable Computing*, Proceedings in Application-Specific Systems, Architectures and Processors, 2004. 15th IEEE International Conference on 2004, pp. 339 - 351.

Wolfram, S. 2002, *A New Kind of Science*, Wolfram Media, Champaign, IL.

Yatsenko, D. V. 2003, *Automated Placement and Routing of Cell Matrix Circuits*, Master Thesis, Utah State University, Department of Computer Science.
<http://www.cellmatrix.com/entryway/products/pub/yatsenko2003.pdf>

Appendix A. CVM Details

This Thesis and the CVM package related to it can be found from:

`http://mkoskim.drivehq.com/`

Animations generated by CVM are uploaded to YouTube;

`http://www.youtube.com/user/MarkusKoskimies`

The author can be contacted;

`mkoskim@gmail.com`

A.1 CVM Operation Set

The following list contains all operations implemented in the version of CVM used to execute the configurations in this Thesis. The first column is the name of the operation, the second tells the operands, and the third is a short description of the operation.

Routing. Routing operations are used for connecting other operations together, as well as collecting the results of the computations.

<code>mix</code>	<code>...</code>	Mixing unorderedly combines the input streams.
<code>move</code>	<code>a</code>	Move is a unary operation, which copies the input symbol to its result.
<code>route</code>	<code>...</code>	An alias for either <code>move</code> or <code>mix</code> operations; mainly for separating routings from other logic in cellular structures.
<code>zip</code>	<code>...</code>	Zip combines the input streams in predefined order.

Symbol Arithmetics. Symbol arithmetic operations produce result streams by applying the transition function to each input symbol(s) separately.

<code>add</code>	<code>a, b</code>	Binary chained symbol addition.
<code>and</code>	<code>a, b</code>	Binary logical AND.
<code>not</code>	<code>a</code>	Unary logical NOT.
<code>or</code>	<code>a, b</code>	Binary logical OR.
<code>sub</code>	<code>a, b</code>	Binary chained symbol subtraction.
<code>xor</code>	<code>a, b</code>	Binary logical exclusive-OR.

String Manipulation. String-manipulating operations are used for generating, storing, splitting and combining strings.

<code>foreach(S)</code>	<code>a</code>	For each input string, a predefined string S is generated.
<code>input(S)</code>	<code>-</code>	Generates a “one-shot” sequence S.
<code>join</code>	<code>...</code>	Join combines the input streams in predefined order like <code>zip</code> , but it also removes the intermediate NIL terminators.
<code>postfix(S)</code>	<code>a</code>	Postfix appends a predefined sequence S to the end of the input stream.
<code>pick(S)</code>	<code>a</code>	Pick is reconfigured with a template string S; for each non-zero symbol in S, a corresponding symbol from input string is passed through. Otherwise, the input symbol is discarded.
<code>prefix(S)</code>	<code>a</code>	Prefix appends a predefined sequence S to the beginning of the input stream.
<code>RAMCell</code>	<code>a</code>	RAMCell is used for implementing writable memory blocks. The input is prefixed with a command; “0” for reading, and “1” for writing the contents. When reading, the content is appended to the end of the passed stream; when writing, the second symbol from the stream is extracted and stored to the cell.

<code>remove(S)</code>	<code>a</code>	Remove acts opposite to <code>pick</code> ; it passes through all symbols in the input stream, where the corresponding symbol value in the controlling stream <code>S</code> is zero.
------------------------	----------------	---

String Reduction. String-reduction operations scan the entire string and return a single-symbol stream according to the results. These are used for generating controlling values for conditional processing.

<code>equal</code>	<code>a, b</code>	If operands are equal, results to single “1”; otherwise results to “0”.
<code>has(X)</code>	<code>a</code>	If the input operand contains the symbol <code>X</code> , results to single “1”; otherwise, results to “0”.
<code>isz</code>	<code>a</code>	If the entire input string has all zeroes, results to “1”; otherwise results to “0”.

Conditional Processing. These functions evaluate the first symbol of the input string, and decide whenever the pass or block the stream.

<code>AddrCmp(X)</code>	<code>a</code>	Effectively an alias to <code>pass</code> . Used in memory address comparators, to separate memory logic from other logic for statistical analysis.
<code>block(X)</code>	<code>a</code>	Blocks input strings, which first symbol equals to <code>X</code> ; opposite to <code>pass</code> . From passed strings, the first symbol is removed.
<code>pass(X)</code>	<code>a</code>	Passes input strings, which first symbol equals to <code>X</code> ; the first symbol is also removed from the stream.

Miscellaneous. Other useful operations.

<code>config</code>	<code>a</code>	Unary operation, which transfers the incoming data stream to reconfiguration unit, thus enabling the re-configuration of the cell array. It does not output anything.
<code>sync</code>	<code>a, b</code>	Special binary variation of <code>move</code> operation, which passes the first input stream through, when fired with controlling input.

CVM Specific. Following operations were implemented for helping the development of structures for CVM.

<code>buffer(N)</code>	<code>a</code>	Buffer is a CVM-specific operation, which buffers a certain number (<code>N</code>) of symbols. It acts as a chain of <code>move</code> operations.
------------------------	----------------	---

<code>counter(P)</code>	a	Used for counting strings, to measure execution data from cellular structures. Effectively, counts the number of by-passed NIL symbols.
<code>configin(F)</code>	–	Used for testing reconfiguration mechanism. Takes in the configuration stream in symbolic form from file F, and converts it to internal representation.
<code>outx(P)</code>	a	Used for debugging only; prints out the input string by prefixing it with a prompt P.
<code>reserved</code>	–	A special dummy operation used for padding the cellular structures in CVM.
<code>symcount(P)</code>	a	As <code>counter</code> , but counts the number of symbols passing through instead of strings.
<code>validate(F)</code>	a	Used for validating the correct behaviour of a cellular structure. It takes in the symbol stream from file F, and compares the input stream to that. Whenever a mismatch is occurred, an error is shown and the CVM is terminated.

Appendix B. Tape Content for Self-Replicating Loop

The following listing contains the tape content for self-replicating loop. At the left side, there is a human-readable description of the operation. At the right side, there are corresponding symbols. There are 655 symbols in the description and the generated configuration is 1018 symbols.

The interpretation of symbols, from left to right:

- First symbol determines, if the following sequence is encoded (0) or not (1)
- If the sequence is encoded, the next two symbols are the repeating count in hexadecimal.
- Next symbol is direction to send the configuration, from 0 to 5.
- The direction can follow the configuration for cell. The values are function, options and inputs, all separated by <FS> (Field Separator) symbol. The possible options (not present in this configuration) would be separated by <LS> (List Separator Symbol).
- The configuration ends to <SS> (String Separator) symbol.

The generator, which builds the description from textual representation, appends a <NIL> symbol after each configuration, although this would not be necessary.

```

1  #-----
2  # Description of the tape
3  #-----
4  By-pass: +z                      10<SS>
5  By-pass: +x                      13<SS>
6  Rep    3: +y                      0304<SS>
7  By-pass: -z Move[] (-y)          111<FS><FS>5<SS>
8  Rep    9: +y Move[] (-y)          09041<FS><FS>5<SS>
9  By-pass: +z Move[] (-z)          101<FS><FS>1<SS>
10 Rep    5: -y Move[] (+y)          05051<FS><FS>4<SS>
11 By-pass: +x Move[] (-x)          131<FS><FS>2<SS>
12 Rep    5: +y Move[] (-y)          05041<FS><FS>5<SS>
13 By-pass: -z Move[] (+z)          111<FS><FS>0<SS>
14 Rep    5: -y Move[] (+y)          05051<FS><FS>4<SS>
15 By-pass: +x Move[] (-x)          131<FS><FS>2<SS>
16 Rep    5: +y Move[] (-y)          05041<FS><FS>5<SS>
17 By-pass: +z Move[] (-z)          101<FS><FS>1<SS>
18 Rep    5: -y Move[] (+y)          05051<FS><FS>4<SS>

```

19	By-pass: +x Move[] (-x)	131<FS><FS>2<SS>
20	Rep 5: +y Move[] (-y)	05041<FS><FS>5<SS>
21	By-pass: -z Move[] (+z)	111<FS><FS>0<SS>
22	Rep 5: -y Move[] (+y)	05051<FS><FS>4<SS>
23	By-pass: +x Move[] (-x)	131<FS><FS>2<SS>
24	Rep 5: +y Move[] (-y)	05041<FS><FS>5<SS>
25	By-pass: +z Move[] (-z)	101<FS><FS>1<SS>
26	Rep 5: -y Move[] (+y)	05051<FS><FS>4<SS>
27	By-pass: +x Move[] (-x)	131<FS><FS>2<SS>
28	Rep 5: +y Move[] (-y)	05041<FS><FS>5<SS>
29	By-pass: -z Move[] (+z)	111<FS><FS>0<SS>
30	Rep 5: -y Move[] (+y)	05051<FS><FS>4<SS>
31	By-pass: +x Move[] (-x)	131<FS><FS>2<SS>
32	Rep 5: +y Move[] (-y)	05041<FS><FS>5<SS>
33	By-pass: +z Move[] (-z)	101<FS><FS>1<SS>
34	Rep 9: -y Move[] (+y)	09051<FS><FS>4<SS>
35	By-pass: -y Buffer[1024] (+y)	153<FS>4201<FS>4<SS>
36	By-pass: -z Sync[] (+z, -y)	112<FS><FS>05<SS>
37	By-pass: -y Buffer[32] (+x)	153<FS>23<FS>3<SS>
38	By-pass: -y Config[] (-x)	150<FS><FS>2<SS>
39	By-pass: +x Reserved[] ()	1314<FS><FS><SS>
40	By-pass: +y Mix[] (-y, +y)	14a<FS><FS>54<SS>
41	By-pass: +y Move[] (-x)	141<FS><FS>2<SS>
42	Break	1
43		
44	#-----	
45	# Description of the decoding machine	
46	#-----	
47	By-pass: +z	10<SS>
48	Rep 2: +x	0203<SS>
49	By-pass: -z Move[] (+y)	111<FS><FS>4<SS>
50	By-pass: +x Move[] (-x)	131<FS><FS>2<SS>
51	By-pass: +x Move[] (-x)	131<FS><FS>2<SS>
52	By-pass: +x Mix[] (-x, +y)	13a<FS><FS>24<SS>
53	By-pass: +x Move[] (-x)	131<FS><FS>2<SS>
54	By-pass: +y Move[] (+y)	141<FS><FS>4<SS>
55	By-pass: -x Move[] (+x)	121<FS><FS>3<SS>
56	By-pass: -x	12<SS>
57	By-pass: -x Move[] (-y)	121<FS><FS>5<SS>
58	By-pass: -x Block[0] (+y)	1211<FS>0<FS>4<SS>
59	By-pass: +y Sync[] (-x, +x)	142<FS><FS>23<SS>
60	By-pass: +y Pass[0] (-y)	1410<FS>0<FS>5<SS>
61	By-pass: +x Move[] (-x)	131<FS><FS>2<SS>
62	By-pass: +y Pick[11] (-y)	148<FS>11<FS>5<SS>
63	By-pass: +y ForEach[FF] (+y)	14f<FS>FF<FS>4<SS>
64	By-pass: +y Move[] (+x)	141<FS><FS>3<SS>
65	By-pass: +x Move[] (+x)	131<FS><FS>3<SS>
66	By-pass: -y Add[] (-x, +y)	154<FS><FS>24<SS>
67	By-pass: -y Mix[] (-x, +y)	15a<FS><FS>24<SS>
68	By-pass: +x Move[] (-x)	131<FS><FS>2<SS>
69	By-pass: +y Join[] (+x, -y)	14c<FS><FS>35<SS>
70	By-pass: +y Block[F] (-y)	1411<FS>F<FS>5<SS>
71	By-pass: +x	13<SS>
72	By-pass: -y Move[] (-y)	151<FS><FS>5<SS>
73	By-pass: -y IsZ[] (-x)	157<FS><FS>2<SS>

```

74 By-pass: +z Pass[F] (-z)          1010<FS>F<FS>1<SS>
75 Rep    3: -x Move[] (+x)         03021<FS><FS>3<SS>
76 Rep    2: -y Move[] (+y)         02051<FS><FS>4<SS>
77 By-pass: -z Mix[] (+z,-y,+x)     11a<FS><FS>053<SS>
78 By-pass: +x Input[] ()           1316<FS><FS><SS>
79 By-pass: +x Move[] (-y)          131<FS><FS>5<SS>
80 By-pass: +x Block[F] (+y)        1311<FS>F<FS>4<SS>
81 By-pass: +y Join[] (+y,-x)       14c<FS><FS>42<SS>
82 By-pass: -x Mix[] (-y,-x)        12a<FS><FS>52<SS>
83 By-pass: -x Remove[11] (-x)      129<FS>11<FS>2<SS>
84 Break                             1
85
86 #-----
87 # Enabling the generated machine
88 #-----
89 By-pass: +x Input[] ()           1316<FS><FS><SS>
90 Break                             1

```

Appendix C. MLI Registers and Instruction Set

Register Set. The register bank of the MLI contains 15 registers;

d0 - d3	General purpose registers
a0 - a3	General purpose registers
t0 - t3	General purpose registers
sp	Stack Pointer
fp	Frame Pointer
pc	Program Counter

Two register indexes are special; 1) if the register field in the instruction is "IMM", the immediate field is passed to function unit, and 2) the register "NC" (Not Connected) is read as zero and it is not written, if specified as destination (i.e. discard the result).

The naming of the general purpose registers is selected to help separating data values (dx) and addresses (ax) in assembler source files. The temporary registers (tx) are meant to store temporary values during calculation. Stack pointers (sp and fp) are named for clarifying stack addressing. Although these registers have a designed function, from the processor point of view they are all general purpose.

Instruction Set. The implemented instructions are;

<code>cmp</code>	<code>a, b</code>	Compares two values. In the result, the first bit is set, if values are equal. The second bit is set, if the first value (A) is greater than the second value (B).
<code>ifeq</code>	<code>a, b</code>	The first operand A is supposed to be a result from <code>cmp</code> instruction. If the first bit is set, the second operand (B) is loaded to the destination register — otherwise, the result is discarded.
<code>ifne</code>	<code>a, b</code>	The first operand A is supposed to be a result from <code>cmp</code> instruction. If the first bit is not set, the second operand (B) is loaded to the destination register — otherwise, the result is discarded.
<code>ifle</code>	<code>a, b</code>	The first operand A is supposed to be a result from <code>cmp</code> instruction. If the second bit is not set, the second operand (B) is loaded to the destination register — otherwise, the result is discarded.

<code>ifgt</code>	<code>a, b</code>	The first operand A is supposed to be a result from <code>cmp</code> instruction. If the second bit is set, the second operand (B) is loaded to the destination register — otherwise, the result is discarded.
<code>ifge</code>	<code>a, b</code>	The first operand A is supposed to be a result from <code>cmp</code> instruction. If either first or second bit is set, the second operand (B) is loaded to the destination register — otherwise, the result is discarded.
<code>iflt</code>	<code>a, b</code>	The first operand A is supposed to be a result from <code>cmp</code> instruction. If both the first and second bits are not set, the second operand (B) is loaded to the destination register — otherwise, the result is discarded.
<code>ifz</code>	<code>a, b</code>	If the first operand (A) is zero, the second operand (B) is loaded to destination register. Otherwise, the result is discarded i.e. no operation. This is used for conditional branching (<code>PC = ifz A, label</code>).
<code>ifnz</code>	<code>a, b</code>	If the first operand (A) is non-zero, the second operand (B) is loaded to destination register. Otherwise, the result is discarded. Used to create conditional branching.
<code>load</code>	<code>a</code>	Loads a value from data memory to destination. The operand A is used for address.
<code>store</code>	<code>a, b</code>	Stores the operand B to data memory. Operand A is the address.
<code>move</code>	<code>a</code>	The destination is set to operand A.
<code>not</code>	<code>a</code>	The destination is set to NOT(A).
<code>inc</code>	<code>a</code>	The destination is set to $A + 1$.
<code>dec</code>	<code>a</code>	The destination is set to $A - 1$.
<code>and</code>	<code>a, b</code>	The destination is set to A AND B.
<code>or</code>	<code>a, b</code>	The destination is set to A OR B.

xor	a, b	The destination is set to A XOR B.
add	a, b	The destination is set to A + B.
sub	a, b	The destination is set to A - B.
nop		No operation.
stop		The processor stops executing instructions.
outx	a	For debugging purposes. Outputs operand A as hexadecimal value.

Appendix D. Eight Queens' Problem, C Source

```

1  /*****
2  *
3  * Queens.c: Find solutions to N-queens problem
4  *
5  *****/
6
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <string.h>
10
11 /*****
12 *
13 * Maximum counts
14 *
15 *****/
16
17 #define MAXQUEENS    100
18 #define MAXDIAGS     (2*MAXQUEENS-1)
19 #define EMPTY        (MAXQUEENS+1)
20
21 /*****
22 *
23 * Global variables
24 *
25 *****/
26
27 /* Number of queens in the current puzzle */
28 int queens;
29 #define rows queens
30 #define cols queens
31
32 /* Occupation tables */
33 int queen  [MAXQUEENS];
34 int columns[MAXQUEENS];
35 int fordiag[MAXDIAGS];
36 int bakdiag[MAXDIAGS];
37
38 /* Solutions found this far */
39 unsigned long solutions = 0;
40
41 /*****
42 *
43 * Solve: Recursive function to find solutions
44 *
45 *****/
46
47 void solve(int level)

```



```

48 {
49     int i;
50
51     /*****
52      * Have we set all queens? If yes, increase solutions
53      * and return.
54      *****/
55
56     if(level == queens)
57     {
58         solutions++;
59         return;
60     }
61
62     /*****
63      * At the level (i.e. column), go through all
64      * possibilities and recurse deeper, if the queen
65      * position fits to previously set queens.
66      *****/
67
68     for(i = 0; i < queens; i++)
69     {
70         /* Get the queens positioned at current column and
71          * diagonals */
72         int *colptr = &columns[i];
73         int *fdgptr = &fordiag[level + i];
74         int *bdgptr = &bakdiag[level + cols - i - 1];
75
76         /* If the locations are empty, place a queen here
77          * and go deeper.
78          */
79         if( *colptr >= level &&
80            *fdgptr >= level &&
81            *bdgptr >= level)
82         {
83             queen[level] = i;
84             *colptr = *fdgptr = *bdgptr = level;
85             solve(level + 1);
86             *colptr = *fdgptr = *bdgptr = EMPTY;
87         }
88     }
89 }
90
91 /*****
92  *
93  * Main
94  *
95  *****/
96
97 int main(int argc, char **argv)
98 {
99     /*****
100      * Initialization
101      *****/
102

```

```

103     int i = 0;
104     solutions = 0;
105     for (i=0; i<MAXQUEENS; ++i) columns[i] = EMPTY;
106     for (i=0; i<MAXDIAGS; ++i)   fordiag[i] = bakdiag[i] = EMPTY;
107
108     /*****
109      * Process arguments
110      *****/
111
112     if(argc < 2)
113     {
114         printf("Usage: queens <no. of queens>\n");
115         return -1;
116     }
117
118     queens = atoi(argv[1]);
119
120     printf("%d queen%s on a %dx%d board...\n",
121           queens,
122           (queens > 1) ? "s" : "",
123           cols, rows
124     );
125
126     /*****
127      * Find all solutions (begin recursion)
128      *****/
129
130     solve(0);
131
132     /*****
133      * Report results
134      *****/
135
136     if (solutions == 1)
137     {
138         printf("There is 1 solution\n");
139     }
140     else
141     {
142         printf("There are %ld solutions\n", solutions);
143     }
144
145     return 0;
146 }

```

Appendix E. Eight Queens' Problem, MLI Source

The MLI assembler program listing for solving eight queens' problem, manually translated from the C source code. The statistical information of compilation of the source code is shown in Table 1.

Table 1. MLI eight queens' problem solver statistics.

Parameter	MLI
Code size (instructions)	87
Data size (words)	112

The MLI assembler source code;

```

1  #####
2  #
3  # Queens.mli: Assembler source code for finding all
4  # solutions (92) to eight queen problem.
5  #
6  # NOTES:
7  # - All main level identifiers are declared to global
8  #   (.global) to be able to view them in the symbol tables.
9  #
10 #####
11
12 #-----
13 #
14 # Boot code
15 #
16 #-----
17
18 .code
19 .global boot:
20 {
21     sp = stack      # Initialize stack
22     fp = sp         # Initialize stack frame ptr
23     pc = main       # Goto main
24 }
25
26 #-----
27 #
28 # Reserving stack
29 #
30 #-----
31
```

```

32 .data
33     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
34     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
35     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
36     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
37 .global stack:
38
39 #-----
40 #
41 # Reserving data
42 #
43 #-----
44
45 .data
46
47     .global queens:      # Number of queens
48     .global rows:       # Number of rows (= # of queens)
49     .global cols:       8 # Number of cols (= # of queens)
50
51     .global solutions: 0  # Solutions found
52
53 #-----
54 #
55 # Main function
56 #
57 #-----
58 .code
59 .global main:
60 {
61
62     #-----
63     # Call the solve() function to initiate the solution
64     # finding.
65     #-----
66     {
67         sp = dec    sp
68             store sp, 0      # Level = 0
69         sp = dec    sp
70             store sp, return  # Store return address
71         pc = solve      # Goto solve
72     return:
73         sp = add sp, 2
74     }
75
76     #-----
77     # Print out the number of solutions found
78     #-----
79
80     d0 = load solutions
81         outd d0
82         stop
83 }
84
85 #-----
86 #

```

```

87 # Data for solution finding; this tables are filled in
88 # recursive loop and they indicate the location of queens
89 # set by the earlier phase of the loop.
90 #
91 #-----
92
93 .data
94     .global queen:      0, 0, 0, 0, 0, 0, 0, 0
95
96     .global column:     255, 255, 255, 255, 255, 255, 255, 255
97
98     .global fdiag:      255, 255, 255, 255, 255, 255, 255, 255
99                         255, 255, 255, 255, 255, 255, 255
100
101     .global bdiag:      255, 255, 255, 255, 255, 255, 255, 255
102                         255, 255, 255, 255, 255, 255, 255
103
104 #-----
105 #
106 # solve(): Finding solutions
107 #
108 # args...:    FP+2 = level
109 # registers: D0  = level
110 #             D3  = column counter (0 ... 7)
111 #             A1  = ptr to column table
112 #             A2  = ptr to forward diagonal table
113 #             A3  = ptr to backward diagonal table
114 #
115 #-----
116 .code
117 .global solve:
118 {
119     #-----
120     # Set up a local stack frame
121     #-----
122     sp = dec    sp
123         store sp, fp
124     fp = sp
125
126     #-----
127     # Check, if we have find a solution (level == queens)
128     #-----
129
130     a0 = add    fp, 2          # A0 = FP+2
131     d0 = load   a0             # D0 = level [FP + 2]
132
133     d1 = load   queens         # D1 = [# of queens]
134     d1 = cmp    d0, d1         #
135     pc = iflt   d1, loop       # if level(D0) < queens(D1),
136                                 # goto looping
137
138     d0 = load   solutions      # Load "# of solutions" to D0
139     d0 = inc    d0             # Increment
140     store      solutions, d0   # Store new value...
141     outd d0

```

```

142     pc = return                # ...And return
143
144 loop:
145     #-----
146     # Loop the column 0 ... 7
147     #-----
148
149     a0 = add fp, 2
150     d0 = load a0                # D0 = level
151
152     a1 = column                # A1 = &column[0]
153
154     a2 = fdiag                 # A2 = &fdiag[level]
155     a2 = add a2, d0
156
157     a3 = bdiag                 # A3 = &bdiag[level + columns - 1]
158     a3 = add a3, d0
159     d1 = load cols
160     a3 = add a3, d1
161     a3 = dec a3
162
163     d3 = 0                     # D3 = 0
164
165     again:
166     {
167         #-----
168         # From occupation tables, check, that the queen can
169         # be placed to given column.
170         #-----
171
172         d1 = load a1            # if (*col < level), next
173         d1 = cmp d1, d0
174         pc = iflt d1, next
175
176         d1 = load a2            # if (*fwddiag < level), next
177         d1 = cmp d1, d0
178         pc = iflt d1, next
179
180         d1 = load a3            # if (*backdiag < level), next
181         d1 = cmp d1, d0
182         pc = iflt d1, next
183
184         #-----
185         # The place was free - mark it occupied and recurse
186         # to next row.
187         #-----
188     recurse:
189         store a1, d0
190         store a2, d0
191         store a3, d0
192
193         #-----
194         # Recurse the solve() function
195         #-----
196         {

```

```

197      # -----
198      # Push current values of loop variables
199      # -----
200      sp = dec    sp
201          store sp, a1
202      sp = dec    sp
203          store sp, a2
204      sp = dec    sp
205          store sp, a3
206      sp = dec    sp
207          store sp, d3
208
209      # -----
210      # Set up the function call
211      # -----
212      sp = dec    sp
213      d0 = inc    d0
214          store sp, d0          # Level = Level + 1
215      sp = dec    sp
216          store sp, return     # Store return address
217      pc = solve
218  return:
219      sp = add sp, 2
220
221      # -----
222      # Restore loop variables
223      # -----
224      a0 = add    fp, 2          # D0 = level
225      d0 = load  a0
226
227      d3 = load  sp              # D3 = counter
228      sp = inc   sp
229      a3 = load  sp              # A3 = ptr to backward
230                                  # diagonal tbl
231      sp = inc   sp
232      a2 = load  sp              # A2 = ptr to forward
233                                  # diagonal tbl
234      sp = inc   sp
235      a1 = load  sp              # A1 = ptr to column tbl
236      sp = inc   sp
237  }
238
239  #-----
240  # Mark the columns & diagonals free again
241  #-----
242      store a1, 255
243      store a2, 255
244      store a3, 255
245
246  #-----
247  # Next column
248  #-----
249  next:
250      d3 = inc d3
251      a1 = inc a1

```

```

252         a2 = inc a2
253         a3 = dec a3
254
255         # Loop until D3 >= queens
256         d1 = load cols
257         d1 = cmp d3, d1
258         pc = iflt d1, again
259     }
260
261     #-----
262     # Return from function
263     #-----
264 return:
265     sp = fp           # Restore SP
266     fp = load fp      # Load previous stack frame
267     sp = inc sp       #
268     pc = load sp       # Jump to the return address
269 }

```


Appendix F. VNP, Von Neumann Processor

During the making of the Thesis, a variation to MLI processor called VNP (Von Neumann Processor) was designed. An emulator and assembler compiler was written with D programming language to PC. VNP was aimed to be used for implementing a self-replicating machine, but later a simpler machine was implemented for that purpose.

The key feature in VNP is that instead of having a separate instruction memory, it has an instruction decoder, which allows encoding the instructions to data memory, see Figure 1. Thus it is capable of reading and writing its own program.

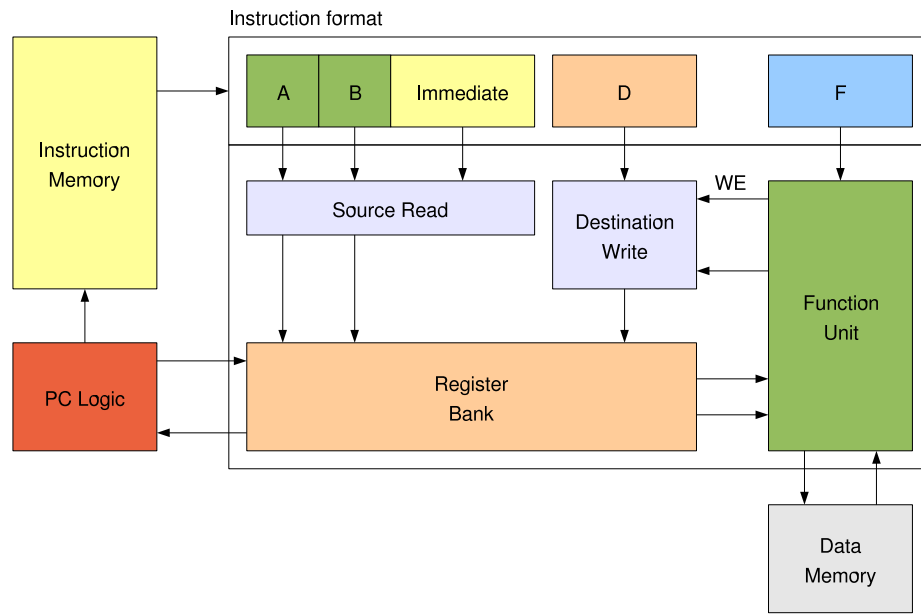
The memory width is 16 bits (4 cells); each instruction is encoded to one 16-bit value. Instruction decoder decodes the instructions to the fields, which are equal to the MLI instructions.

To make development easier and faster, the VNP assembler was designed so, that it compiles unmodified MLI assembler source files. Each MLI instruction is encoded to one or more VNP instructions. It would also be possible to write a VNP emulator with MLI assembler, and emulate the VNP processor with MLI processor model.

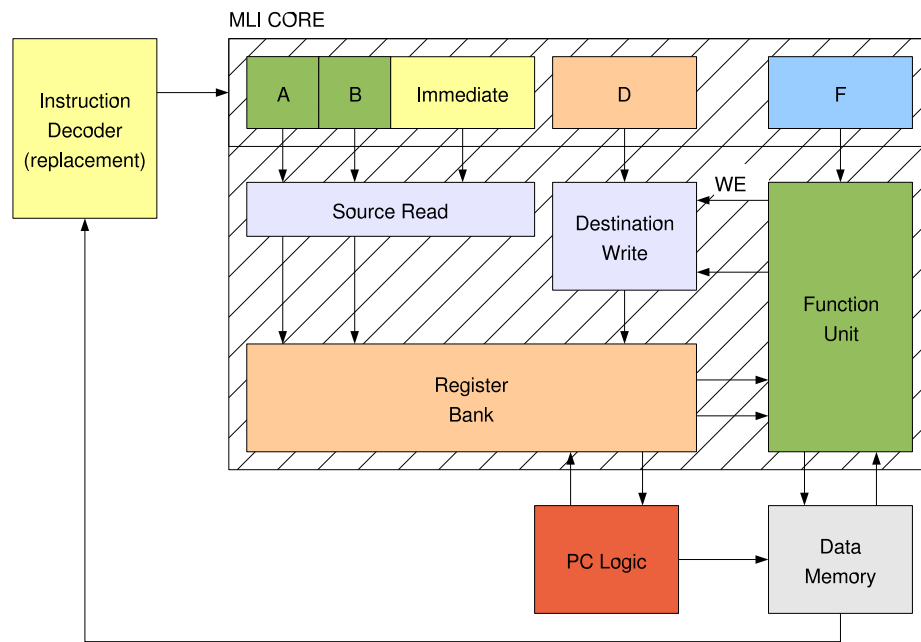
Although VNP was finally not synthesized to CVM, an emulator for PC was written with D programming language. The eight queens problem solving algorithm written with MLI assembler were compiled and executed. The results compared to MLI are shown in Table 1.

Table 1. MLI and VNP eight queens' problem solver statistics.

Parameter	MLI	VNP
Code size (instructions)	87	134
Data size (words)	112	112
Instruction fetches	294,491	447,809
Data memory accesses (tot)	89,722	89,722
Data memory reads	62,899	62,899
Data memory writes	26,823	26,823



(a) Reference MLI



(b) VNP

Figure 1. VNP architecture. (a) MLI architecture for comparison, (b) VNP architecture. In VNP, the instruction memory is replaced with instruction decoder. The PC logic is connected to the data memory bank, and the retrieved result is fed to the decoder.

Instruction Set. The instructions are formed from four nibbles (four bit) fields. The encoding is following;

```

xxxx xxxx xxxx xxxx  Three-operand instructions
0000 dddd aaaa bbbb  d = a + b (add)
0001 dddd aaaa bbbb  d = a - b (sub)
0010 dddd aaaa bbbb  d = a AND b
0011 dddd aaaa bbbb  d = a OR b
0100 dddd aaaa bbbb  d = a XOR b
0101 dddd aaaa bbbb  d = CMP(A, b)
0110 dddd aaaa bbbb
0111 dddd aaaa bbbb
1000 dddd aaaa bbbb
1001 dddd aaaa bbbb
1010 dddd aaaa bbbb
1011 dddd aaaa bbbb
1100 dddd aaaa bbbb
1101 dddd iiii iiii  move d, i8
1110 dddd iiii iiii  move.shl d, i8
1111 xxxx xxxx xxxx  Two-operand instructions
1111 0000 aaaa bbbb  ifz(a) jmp b
1111 0001 aaaa bbbb  ifnz(a) jmp b
1111 0010 aaaa bbbb  ifeq(a) jmp b
1111 0011 aaaa bbbb  ifne(a) jmp b
1111 0100 aaaa bbbb  iflt(a) jmp b
1111 0101 aaaa bbbb  ifle(a) jmp b
1111 0110 aaaa bbbb  ifgt(a) jmp b
1111 0111 aaaa bbbb  ifge(a) jmp b
1111 1000 dddd aaaa  d = a (Register move)
1111 1001 dddd aaaa  d = mem[a] (Register load)
1111 1010 aaaa bbbb  mem[a] = b (Register store)
1111 1011 dddd aaaa  d = a + 1 (inc)
1111 1100 dddd aaaa  d = a - 1 (dec)
1111 1101 dddd aaaa  d = not(a)
1111 1110 **** ****
1111 1111 xxxx xxxx  Miscellaneous instructions
1111 1111 0000 aaaa  outx a
1111 1111 0001 aaaa  outd a
1111 1111 0010 aaaa  outb a
1111 1111 0011 ****
1111 1111 0100 ****
1111 1111 0101 ****
1111 1111 0110 ****
1111 1111 0111 ****
1111 1111 1000 ****
1111 1111 1001 ****

```

1111	1111	1010	****	
1111	1111	1011	****	
1111	1111	1100	****	
1111	1111	1101	****	
1111	1111	1110	****	
1111	1111	1111	xxxx	No-operand instructions
1111	1111	1111	0000	nop
1111	1111	1111	0001	
1111	1111	1111	0010	
1111	1111	1111	0011	
1111	1111	1111	0100	
1111	1111	1111	0101	
1111	1111	1111	0110	
1111	1111	1111	0111	
1111	1111	1111	1000	
1111	1111	1111	1001	
1111	1111	1111	1010	
1111	1111	1111	1011	
1111	1111	1111	1100	
1111	1111	1111	1101	
1111	1111	1111	1110	
1111	1111	1111	1111	stop

Appendix G. “Glasswalls”: A Failed Protection Mechanism

The drawback of the mechanism presented in the Thesis is that it does not allow applications to perform any kind of reconfigurations, which could be very valuable for implementing certain dynamically extending and adapting structures. As an attempt to try to allow applications freely reconfigure the cellular space reserved for them and protecting the malfunctional attempts causing vulnerabilities, a protection model called “glasswalls” was designed.

Assume, that the reconfiguration controller in the cell contains a bitmask, which tells from which directions it accepts reconfiguration requests. This way, the operating system might cover the application area with a *wall*, which does not accept reconfiguration requests from the application side. It was named to *glasswall*, because it is transparent; data goes through it without any problems, and so do the reconfiguration requests from other side. It just blocks requests from “wrong” side.

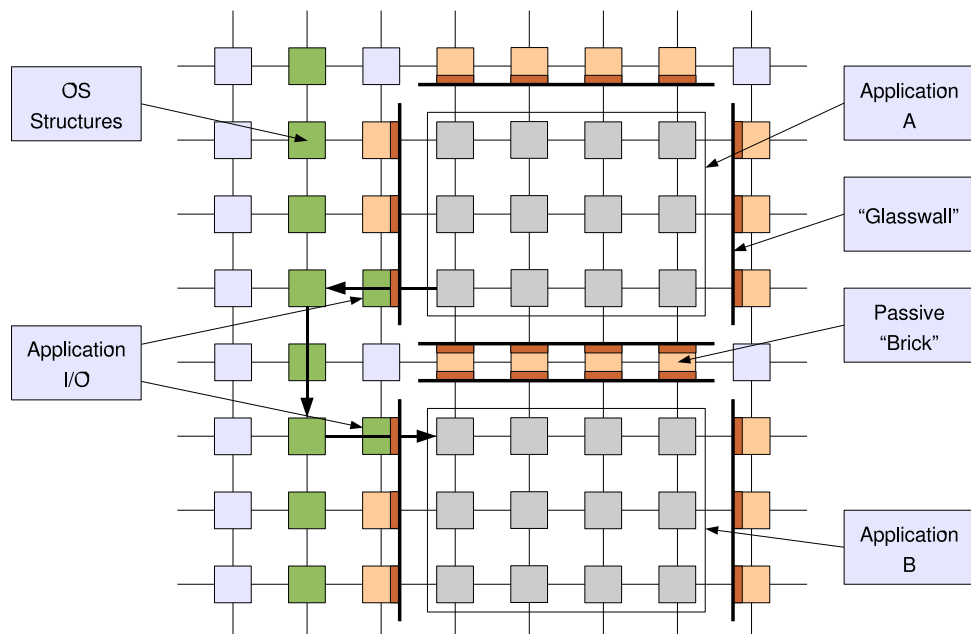


Figure 1. Glasswalls.

Figure 1 shows a situation, where OS (Operating System) has configured two applications, A and B. The applications are surrounded by a *brickwall*, passive cells which prevent the applications to unintentionally “listen” the others. Only the *holes* (Application I/O) created by OS allow the application to communicate, and all communication is supervised

by the OS. This brickwall is configured so that the cells block the reconfiguration requests coming from the application side (darker blocks in the cells), which then creates a glasswall. So, no matter what kind of reconfigurations applications perform, they can not penetrate the wall and cause harm to OS or to other applications.

So, everything is OK? No. In Figure 2, a malicious application fortify itself by creating its own glasswall; although it is not able to break the wall to the OS side to cause harmful actions, it can prevent the OS to wipe it out — the application stays permanently in the array, no matter what means are tried to remove it (configuration is non-volatile). Also, it should be noted, that if the glasswall is breakable from outside (i.e. operating system could somehow break the fortification), it may also be an exploitable hole for applications to break the protection.

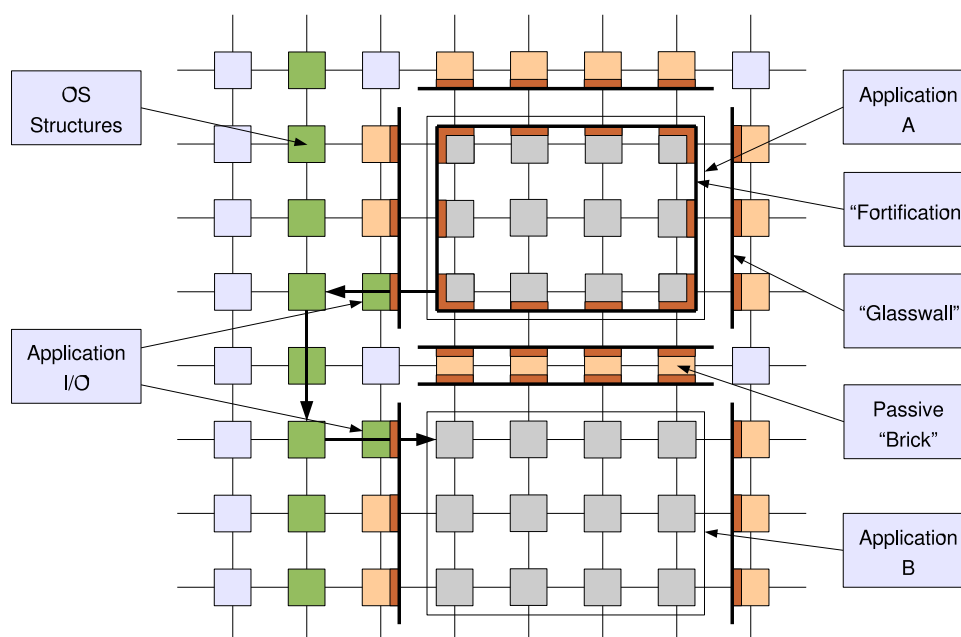


Figure 2. Abusing glasswalls.

The reason why this non-working protection mechanism is presented here is that it was thought in the very beginning, and this problem was found. Months later the reason why this was rejected was forgotten and this was reconsidered, only to find out that it won't work in this form. So, this mechanism is presented here to prevent coming back to it without having a solution against these *fortified applications*.

The failure of the glasswall raised a question, if the applications should be able to do reconfigurations by themselves, without using OS to do it. The answer is probably no. There are two reasons to suspect this;

1. Reconfiguration needs knowledge about the array topology. Thus portable applications need to compute the reconfiguration streams from netlists, i.e. that is not a trivial task to do.

2. The defect tolerance strategy is that if there are damaged cells in the system, they are not used by the OS. This means that when reconfiguring an application, the OS need to perform automated placement and routing; the defected cells make the array irregular. This also means that the configurations are more probably stored to the array as netlists and they are processed to reconfiguration streams while making an instance of an application. The applications would need this information (defected cells) when doing reconfigurations by themselves.

It seems that the simplest way to protect the system is to prevent applications for performing reconfigurations and let only the OS to do that. The OS contains a *bad block detection* mechanism to find out defected cells in the area under reconfiguration and it will do the placement and routing.