# The Cellular Automata Research Platform: Communication and Verification

Per Thomas Lundal – Supervised by Gunnar Tufte

***Abstract*—Research into evolvable hardware has long been performed at NTNU. In 2003, a cellular automata research platform was created with the use of an FPGA.**

**In expectation of new hardware with a larger FPGA, the platform was updated and extended in 2014. It was optimized to provide a speedup of 4 for most operations by taking advantage of increased resources. However, the new PCI Express interface was not implemented and the design was only verified in simulation.**

**The goal of this project was to finalize the platform, such that the new features and improved speed could be taken advantage of. The first objective towards that end was to replace the old communication interface. The second was to verify that the new design worked correctly when implemented in hardware.**

**This paper details the implementation of the new communication interface, both in hardware and software. Then, the previously upgraded design is implemented and tested in the new hardware.**

**Implementation of the new communication interface is a success. However, testing shows that there are a lot of issues with the upgraded design. Some issues were possible to fix within the alotted time, but several critical remain.**

## I. Introduction

Evolvable hardware (EHW) is a field of science where evolutionary algorithms (EAs) are used in the creation of specialized hardware. EAs simulate mechanisms found in biological life, such as selection, reproduction and mutation. The goal is to optimize fitness, the ability to survive in the competition for being the best solution. EAs can quickly find approximate solutions to hard problems, and then gradually refine them into good solutions.

An interesting feature is that EAs can find ways to exploit hardware in ways that human designers cannot comprehend [1]. This can be due to complex parallel interactions, or usage of properties that are not fully understood [2].

Technologies related to EHW, including a common EA and hardware platforms are presented in Section II.

Evolvable hardware has been an area of research at NTNU for more than a decade. In 2002, NTNU invested in dedicated FPGA hardware with the intent of building an EHW platform. The purpose was to create a platform for experimentation with evolution and development within a cellular automata (CA).

The initial work was done by Djupdal, before being extended by Aamodt. The CA was implemented as a matrix of sblocks, reprogrammable CA cells designed specifically for usage with evolution. It is connected to a development unit which can simulate growth and change in the sblock matrix. The hardware platform is connected to and controlled by a computer over a PCI connection.

A general use-case for the platform is to have the computer run a genetic algorithm, where the genotype represents the initial CA state and development rules. Then, the CA is initialized and developed to produce a phenotype, which is executed to produce a fitness value.

In expectation of new hardware with a larger FPGA and faster PCI Express connection, Støvneng refurbished the design in 2014. He took advantage of the added resources to greatly improve the performance of the platform, giving a speedup of 4 or more for most operations. Additionally, he extended the CA into 3D and added a DFT. However, since the hardware did not arrive in time, the new design was only tested in simulation and the communication interface was not upgraded.

An in-depth explanation of the platform's features, functionality and iterations is presented in Section III.

The task of this project is to finish the new platform by implementing a new PCI Express communication unit, and to verify that everything is functional in hardware. This will allow the new platform, which is both faster and more feature-rich, to be taken into use. The motives are further detailed in Section IV.

The implementation of the new communication unit is detailed in Section VI, while the verification process and results are detailed in Section VII.

Challenges related to setting up the new hardware platform and testing is detailed in in Section VIII, along with proposed design changes and other future work.

Section IX concludes this paper.

## II. Technology

The field of EHW is comprised of many technologies, but only a few are relevant for this paper. Those are genetic algorithms, development, cellular automata, FPGAs and sblocks. Additionally, PCI Express is relevant for the new communication unit.

### A. Genetic algorithms

A genetic algorithm (GA) is a very common type of EA. It represents each solution as a genotype, a binary string used as a blueprint to create the solution itself, called a phenotype. The genotype is comparable to nature's DNA, and it is this genetic material which is modified in the evolutionary process.

The GA process is shown in Fig. 1. First, a base population with random genotypes is generated. Then, each

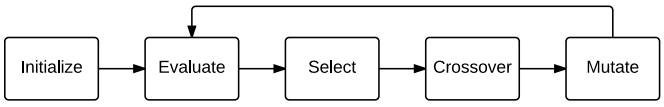


Fig. 1. A genetic algorithm. The cycle is broken when the fitness is above a given threshold.

phenotype is constructed and evaluated using a fitness function. If a solution has a fitness score above a set threshold, the process stops. Otherwise, a new population is created by selecting solutions with high fitness scores, crossing their genotypes, and mutating the results, before repeating the process.

### B. Development

The process that transforms the genotype into a phenotype is called development; it can be regarded as a form of decompression algorithm [3]. In nature, this process is seen when a fertilized egg transforms into a multicellular organism. A conseptual example is shown in Fig. 2.

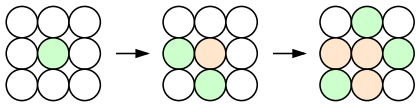

Fig. 2. An example of one cell developing into six. Additionally, some cells change type.

Development also allows individuals to adapt to their environments, making them more robust and scalable [4].

### C. Cellular automata

A cellular automaton (CA) is a structure made up of vast numbers of very simple computational units called cells. The cells are arranged in a grid, with communication only permitted between nearby cells according to a neighborhood. The von Neumann neighborhood is common for two-dimensional CAs; It includes the cells to the north, south, east and west, along with itself (center). An example of this is shown in Fig. 3.

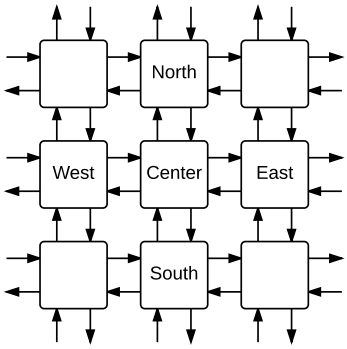


Fig. 3. An excerpt from a 2D CA using a von Neumann neighborhood. All outputs from a given cell carries the same value.

Each cell has a state, which is often a binary number where 1 represents alive and 0 represents dead. At each discrete time step, each cell updates its state based on the states in its neighborhood. The update function is often specified as a look-up-table (LUT), where the next state is defined for all possible neighborhood states[1]. If all cells implement the same update function then the CA is uniform, otherwise it is non-uniform.

The update functions determine how complex patterns and structures emerge within the CA. The emergent behavior can be categorized into one of four classes [5].

1) Homogeneous state.
2) Simple periodic structures.
3) Chaotic patterns.
4) Complex patterns and structures.

The latter is the most interesting, providing both the complexity required for computation and structures required for storage. It exists in the phase change between class 2 and 3, known as the edge of chaos [6].

CAs have been shown to be Turing complete [7] [8], which means they are able to perform any kind of computation.

### D. Field Programmable Gate Array

A Field Programmable Gate Array (FPGA) is a type of reconfigurable hardware. It can implement any desired logical operation by configuring and connecting a number of look-up tables (LUTs) and flip-flops (FFs). FPGAs can also contain dedicated blocks for addition, multiplication, memory, and other functions. These elements are grouped into configurable logic blocks (CLBs), which through a network of interconnects can be connected to each other or input/output pins. An example of this structure is shown in Fig. 4. Note that modern FPGAs consists of thousands of CLBs and hundreds of I/O pins [9].

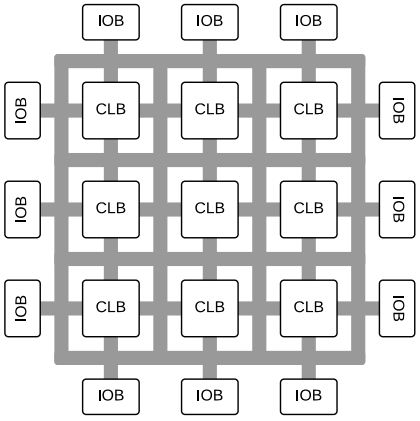


Fig. 4. High-level block diagram of an FPGA. An array of configurable logic blocks (CLBs) and input/output blocks (IOBs) are connected by a network of interconnects.

FPGAs have been the subject of EHW research due to their reconfigurability, and several researchers have been successful in evolving working electronic circuits [10] [1].

[1] The length of the LUT increases exponentially with the size of the neighborhood. $L = S^N$ where $L$ is the LUT length, $S$ is the number of possible states and $N$ is the number of cells in the neighborhood.

However, the resulting circuits have often ended up using intrinsic properties of the silicon and been very sensitive to environmental changes.

The trouble with using modern FPGAs for EHW research is that some configuration bitstrings can destroy the FPGA [11] [12]. This means that the bitstring can not be used directly as the genotype without complicated tests to discard the dangerous bitstrings.

### E. Sblock

The sblock was introduced as part of a new EHW-friendly FPGA architecture in [13]. The architecture is a non-uniform CA with a von Neumann neighborhood, where the update function of each cell is independently configurable at run-time. The cells, known as sblocks, are very simple structures; they consist of a configurable look-up-table (LUT) and a flip-flop (FF), as shown in Fig. 5.

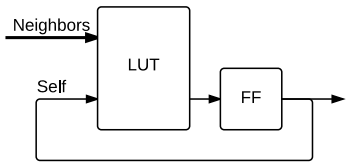


Fig. 5. Detailed block diagram of an sblock. The LUT can be reconfigured on-the-fly to implement any logical function.

The greatest benefit of using sblocks for EHW research is that there is no risk of damage or exploitation of intrinsic properties in the silicon. Additionaly, the simple structure and hardwired signal routing allows for very efficient area usage. The likelihood of a mass-produced sblock-FPGA arriving on the market in the near future is slim. However, it is possible to implement it virtually within an other FPGA.

### F. PCI Express

The PCI Express interface was designed to tackle the arising trouble with clocked parallel buses like PCI. The problem with such buses is that the clock speed can not be increased beyond a given threshold, as the slightly different lengths of the wires causes data to arrive at slightly different times. Reducing the clock period to less than the variation in arrival time means the data will become corrupted. This problem is exacerbated with increasing bus size.

PCI Express is therefore based on serial communication over differential pairs (lanes[2]) without the need for a reference clock [14]. This allows an extremely fast clock speed compared to a parallel bus, and much greater bandwidth in total. PCI Express consists of three layers; the physical layer, the data link layer and the transaction layer, structured as shown in Fig. 6.

[2] PCI Express operates in full duplex mode, which means that each lane has an independent differential pair in each direction. 1, 2, 4, 8, 16 or 32 lanes are supported, but data is striped and thus still transmitted serially.

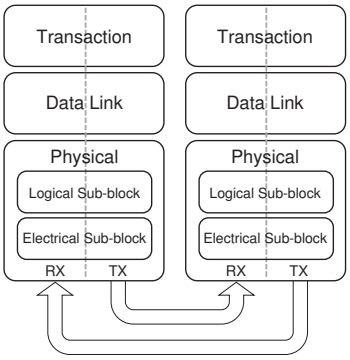


Fig. 6. High-level diagram showing the layered structure of PCI Express. (Reprinted from [14])

The transaction layer's primary responsibility is the creation and parsing of transaction layer packets (TLPs). TLPs are used to trigger events or start various transactions, most commonly to initiate read and write requests[3]. Most requests entail the return of a completion TLP containing the requested data or other information. TLPs consists of multiple 32-bit double words (DW), where the first is a common header describing the type of packet.

The data link layer ensures integrity by adding error detection codes to outgoing TLPs and performing error detection and correction on incoming TLPs. It is also responsible for retransmission if corruption occurs.

The physical layer is responsible for serialization and deserialization of the data stream. Each byte is padded with two extra bits (8b/10b encoding) to allow clock recovery.

## III. Previous Work

The Cellular Automata Research Platform (CARP) has been the subject of three previous master theses at NTNU. The original implementation was made by Djupdal in 2003. It was then extended with a range of various output methods by Aamodt in 2005. Finally, it was further extended and optimized in expectation of new hardware by Støvneng in 2014.

### A. Conception

In 2002, NTNU invested in a CompactPCI computer with a NallaTech BenERA FPGA board to be used for research within the field of evolutionary hardware. The task of developing a platform for the system, based on a matrix of sblocks, fell to Djupdal [15].

An overview of the resulting hardware platform is shown in Fig. 7. It consists of the mentioned sblock matrix, block RAM (BRAM) for storing the state and type of each cell, a development unit, control logic, and a PCI communication unit.

The system is meant to be controlled by a computer running a genetic algoritm. A common flow of operation

[3] Read and write requests are directed at one of up to six base address registers (BARs). They represent internal memory areas that can be anywhere from a few bytes to several gigabytes in size.

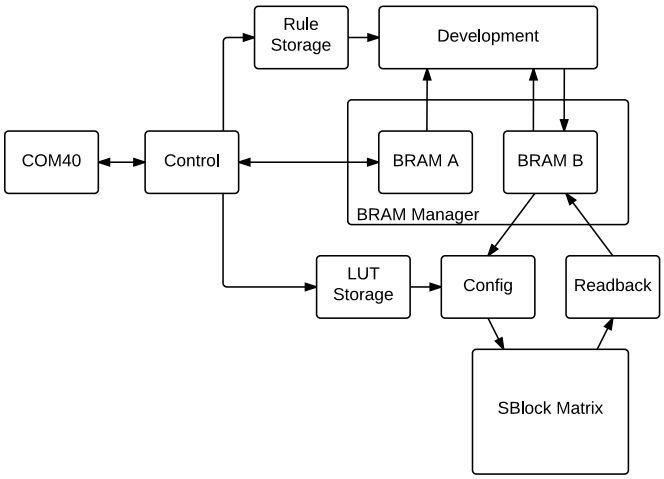


Fig. 7. High-level block diagram of the hardware platform after Djupdal's original work.

is to initialize the system with the genotype, develop it into its phenotype, run the SBM, and send the new states back to the computer. The computer then uses the newly received state data to calculate a fitness score.

The system is initialized by writing states and types to BRAM A, in addition to storing development rules and LUT conversion rules. Then a development step can be performed by reading cell types from BRAM A[4], testing development rules, and writing the (possibly changed) types back to BRAM B. The development unit tests 8 rules on 2 cells each cycle in raster order. Optionally, the BRAMs can be logically swapped and further development steps performed. The SBM can then be configured by translating the types in BRAM B into LUT entries according to the LUT conversion rules, before being run for a desired amount of cycles. Afterwards, the new states in the SBM can be read back into BRAM B, swapped into BRAM A, and sent to the computer.

The design is split into two clock domains; the communication unit uses 40 MHz to be able to interface with PCI, while the rest uses 80 MHz for higher performance.

### B. Extension

There was one major bottleneck in the original design. To calculate the fitness of an individual, the state of each cell had to be transferred to the computer over the PCI interface. Having a dedicated hardware unit would greatly improve the performance. Additionaly, it was desired to have more information about the development process. The task of realizing this fell to Aamodt [16].

An overview of the hardware platform with Aamodt's additions is shown in Fig. 8. The additions consists of a run-step function that calculates the number of live cells, BRAM to store the numbers, a fitness function, and two information outputs from the development unit.

[4] After the first 8 rules have been tested on all cells, center cell types are read from BRAM B instead. This is needed to prevent the result of a rule in an earlier iteration from being deleted if no rules trigger in a later iteration.

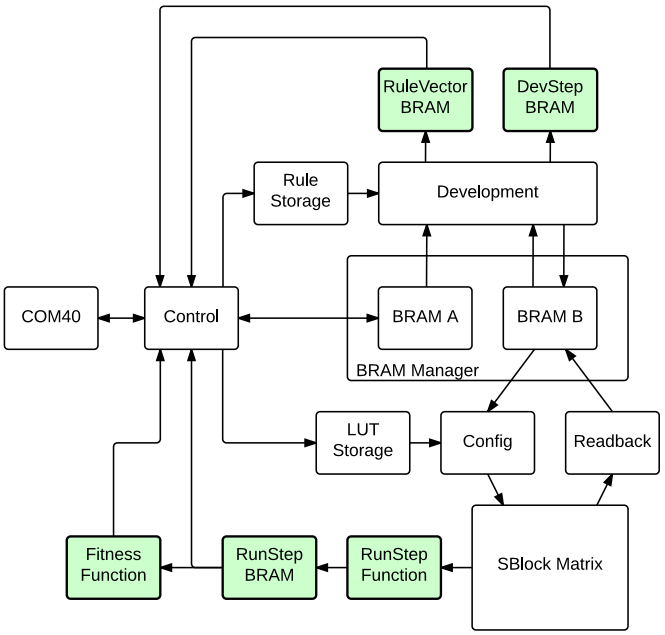


Fig. 8. High-level block diagram of the hardware platform after Aamodt's work. Additions are highlighted in green.

The rule vector BRAM stores lists of which rules were triggered and not for the last 256 development steps. The lists are implemented as bit-vectors where each bit represents the status of a rule for a single development step. The development step BRAM is more detailed; it stores which rule was triggered for each cell. However, it only has storage space for one development step.

The run-step function calculates the number of live cells after each SBM update by using a large adder tree. The numbers are stored in run-step BRAM for later usage by the fitness function, which is replacable.

### C. Renovation

In expectation of receiving new hardware with a larger and faster FPGA, there was a demand to optimize the platform by taking advantage of the increased resource pool. Extending the platform into the third dimension was also a lucrative thought, as doing so allows more complex signal pathways to form within the cellular automata. It was also desired to have a discrete fourier transform (DFT) for interpretation of the RSF data; it should give very useful data according to Berg's research [17]. The task of realizing this was taken on by Støvneng [18].

An overview of the hardware platform with Støvnengs additions and optimizations is shown in Fig. 9. The only addition is the DFT, but nearly all units has been optimized, yielding a speedup of 4 for most operations.

Unfortunately, due to some challenges with manufacturing, Støvneng was unable to get hold of the new hardware for the duration of his project. The system was therefore only verified in simulation, and the PCI communication unit was not upgraded for the PCI Express connection on the new board.

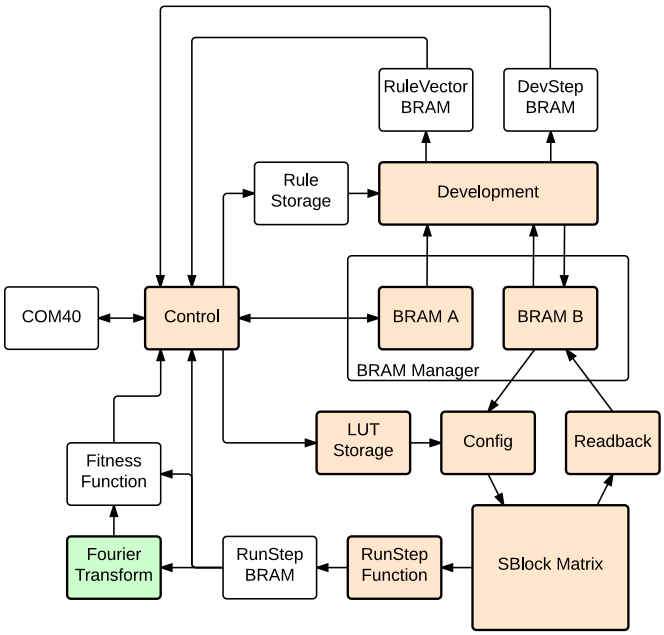


Fig. 9. High-level block diagram of the hardware platform after Støvneng's work. Additions are highlighted in green, and optimizations and 3D modifications in orange.

## IV. Motivation

Currently, CARP is only usable in simulations, which are extremely slow and therefore practically unusable. Implementing a PCI Express communication unit should make the new platform operational, allowing it to be used for research.

The new platform is about 4 times faster than the old and support larger sblock matrices. It also has the new and exciting DFT, which is shown to produce very useful data. 3D is also exiting as it allows much more complex communication pathways to form within the CA.

Verification of previous modifications are necessary since it was only tested in simulation. The path from a behavioral description to an implementation using LUTs and FFs is long and complex; what functions in simulation does not necessarily function when implemented.

## V. Development Platform

Multiple weeks into this project, several months after the end of Støvnengs project, there were still no signs of the new hardware. To prevent the project from halting dead in its tracks, a decision was taken to order slightly different hardware. The significant difference to the original system is reduced size of the FPGA, a Spartan-6 LX45T instead of a Spartan-6 LX150T, which entails around 70% less available resources. Luckily, the hardware design can be scaled down to fit the smaller chip by reducing the size of the sblock matrix, allowing for implementation of PCI Express and verification of the complete system in hardware.

### A. Spartan-6 SP605 Evaluation Platform

The Spartan-6 SP605 Evaluation Platform is essentially a board with the Spartan-6 LX45T FPGA wired to every useful peripheral imaginable. It has connections for PCI Express[5], ethernet, DVI, USB, flash card, JTAG, LEDs, switches, and more. However, the only peripherals utilized in this paper are PCI Express, and JTAG. An overview of the system is shown in Fig. 10.

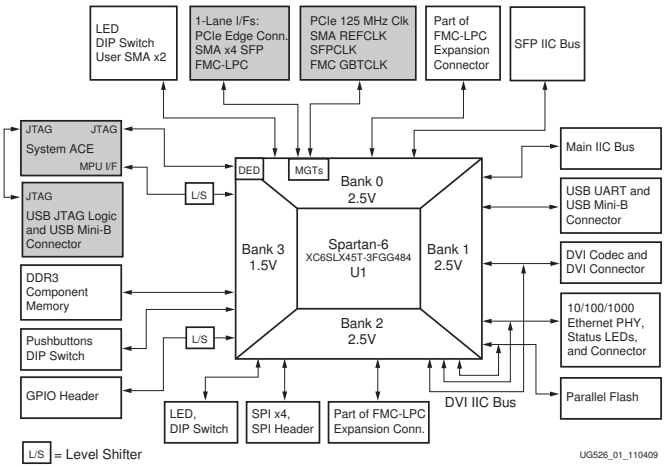


Fig. 10. High-level block diagram of the SP605 and its peripherals. Peripherals utilized in this paper are highlighted in gray. (Modified reprint from [19])

The switch and jumper configurations of the SP605 are set to factory defaults, with the exception of SW1 which is set to 10.

### B. Hardware setup

Due to the experimental nature of testing a new hardware platform, two computers were used in this project, as shown in Fig. 11. One is the main development workstation, used for coding and synthesis; it has a JTAG connection to the SP605 over USB, which allows it to upload new designs. The other is the host for the SP605, which is mounted in a PCI Express expansion slot.

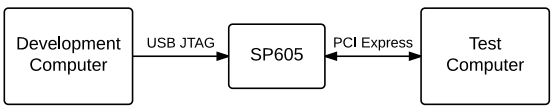


Fig. 11. High-level block diagram of the hardware setup.

The setup allows a new design to be uploaded and tested on the SP605 without disrupting the workflow of the main workstation, due to the power-cycle required to reset the PCI Express connection.

### C. Software setup

The operating system on both computers is Linux Mint; version 16 on the development computer and 17 on the test

[5] Even though the PCI Express finger has lines for power, they are not connected on the SP605. This means an external power source has to be connected.

computer. Linux Mint is currently one of the most popular linux distributions, along with Ubuntu, which it is based upon [20]. This means that procedures and software used and created in this paper should work on most systems.

Xilinx ISE version 13.3 was used for hardware design and synthesis, while ISim was used for simulations. The third-party USB cable driver from [21] was used for JTAG, as explained in Section VIII-A. The software API was compiled with GCC version 4.8.2.

## VI. Implementation

Fig. 12 shows the changes to the hardware platform. The old COM40 unit has been replaced by a new communication unit and a compatibility layer. It communicates with a new software api which uses Linux' built-in drivers for PCI Express.

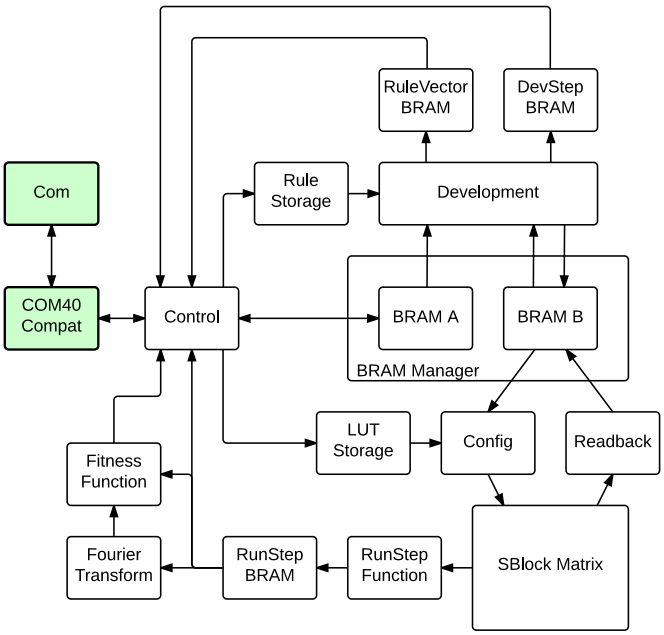


Fig. 12. High-level block diagram of the current hardware platform. Additions are highlighted in green.

### A. Detailed overview

The new communication unit is based on Xilinx' reference PCI Express programmed input/output design. It consists of the Xilinx PCI Express endpoint core, reception and transmission engines, data buffers, and a special request handler, as shown in Fig. 13.

The endpoint core completely handles the physical and data link layers, and all TLPs related to configuration and establishment of the PCI Express connection. Other TLPs, such as read and write requests, are presented on an AXI4-Stream interface [22]. The reception engine is responsible for parsing TLPs and either writing received data to the reception buffer or notifying the transmission engine about a read request. The transmission engine is responsible for building completer TLPs to respond to read requests, using data from the transmission buffer.

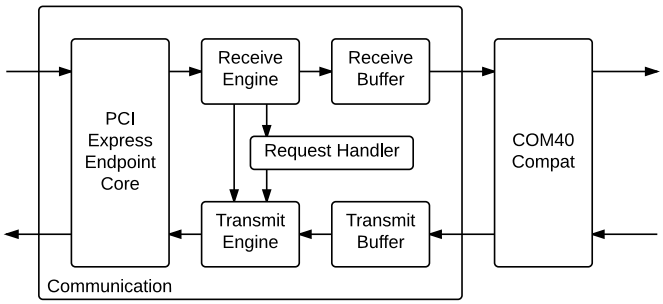


Fig. 13. Detailed block-diagram of the PCI Express communication module.

The request handler listens to the read requests provided by the reception engine, and can override the transmission engine to respond to special requests.

### B. PCI Express Endpoint Core

Several Spartan-6 FPGAs, including the one used in this project, contain a special-purpose hardware block for implementation of PCI Express. The block completely handles the physical and data link layers, with the transaction layer left for the user.

To make use of the block, Xilinx provides the Spartan-6 Integrated PCI Express Endpoint Core; version 2.3 was used in this project. This core additionally takes care of all TLPs related to configuration of the PCI Express connection. Other TLPs, such as read and write requests, are presented on an AXI4-Stream interface [22].

The endpoint core is configured with two memory regions, both 4 kB in size[6]. The first memory region (BAR0) is used for normal communication, while the second (BAR1) is used for special requests. The separation is mostly conseptual as both regions are treated as one data stream. The difference is that the special request handler kicks in for read requests to BAR1.

### C. Reception engine

The reception engine is implemented as a simple state machine, as shown in Fig. 14.

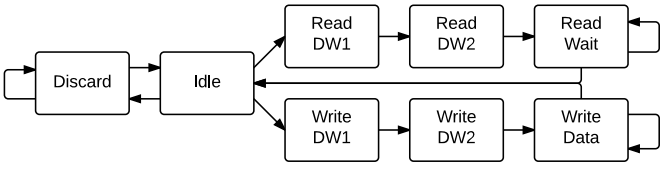


Fig. 14. State machine for the reception engine.

Until the endpoint core presents valid data, the state machine remains in Idle. When it does, the data is stored, and the TLP type is checked. If it is a read or write request, the state machine continues down the corresponding path, otherwise the remaining data is discarded. The remaining

[6] The smallest memory region that can be memory-mapped is one page. The default page size in Linux is 4 kB.

portion of the TLP headers are then parsed in the DW1 and DW2 states. For read requests, the state machine waits in ReadWait until the transmission engine is ready to accept a new read request, and then proceeds to Idle. For write requests, the state machine stays in WriteData, where one DW of data is written to the reception buffer each cycle, for the length of the packet, and then proceeds to Idle.

### D. Transmission engine

The transmission engine is implemented as a simple state machine, as shown in Fig. 15.

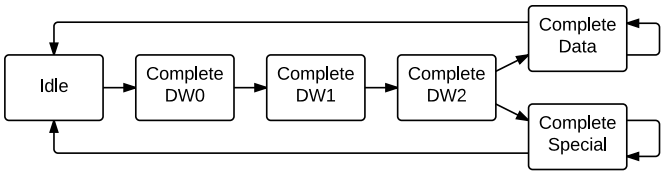


Fig. 15. State machine for the transmission engine.

Until the reception engine signals a read request, the state machine remains in Idle. When a read request is signaled by the reception engine, the state machine begins to traverse the DW path. The DW0, DW1 and DW2 states each transmit one DW of the completer TLP header. Then if the special request signal is set, it procceds to CompleteSpecial, where it transmits data presented by the request handler. Otherwise, it proceeds to CompleteData where it transmits one DW of data from the transmission buffer each cycle. When the requested number of DWs has been transmitted it proceeds back to Idle.

### E. Request handler

The request handler continually listens to the read requests presented by the reception engine. If the request is targeting the primary memory area (BAR 0), it is a normal read request and the transmission engine is allowed to proceed as usual. Otherwise, it is a special request and the transmission engine is overridden.

The kind of special request is determined by the address of the read request, and handled thereafter. There are currently four special requests implemented, as shown in Table I.

TABLE I
SPECIAL REQUESTS

| Address | Request |
|---|---|
| 0x00 | Get transmission buffer data count |
| 0x01 | Get transmission buffer available space |
| 0x02 | Get reception buffer data count |
| 0x03 | Get reception buffer available space |

Note that each of the implemented special requests assumes a read request length of one DW. If the request has a greater length, the returned data is simply repeated to fill the packet.

### F. Buffers

The buffers are implemented as first-in first-out (FIFO) queues using block RAM (BRAM) and two counters. The counters determine the addresses that are written to and read from, and are incremented when the write or read signals are asserted. Fig. 16 shows how the FIFO is used to buffer two words.

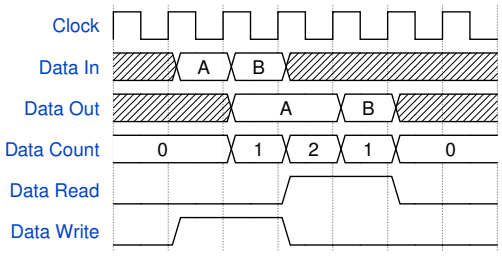


Fig. 16. Wave diagram for the FIFO buffer, showing two consecutive writes followed by two consecutive reads.

Notice how the read signal needs to be asserted before the clock tick when data is read to ensure correct consecutive reads. This is due to the BRAM used in the FIFO, which updates at clock ticks. To have correct data available for a read in the following cycle, the address therefore has to be updated before the clock tick (by asserting the read signal).

### G. Compatibility layer

Due to the difference in wordsize between the new and old communication units (32 vs 64 bits), a compatibility layer is added between the communicaton unit and the control unit. This allows the control unit and the rest of the design to remain unchanged, making the replacement transparent.

The compatibility layer contains two very simple state machines. One combines two 32-bit words from the communication unit into 64-bit words for the control unit. The other splits 64-bit words from the control unit into two 32-bit words for the communication unit.

### H. Software API

The communication part of the new software API is split into two parts.

The first is a general interface for connecting to PCI and PCI Express devices without using a custom driver. It takes advantage of Linux' automatic population of /sys/devices/pci* with files representing the memory regions of all PCI and PCI Express devices. The directory is searched by vendor and device id, and the corresponding memory regions is memory-mapped into the program.

The second is an interface specifically for the communication unit. It provides open, close, read and write functions similar to the old BenERA interface, in addition to implementing all special request functions in Table I. When a read or write operation is initiated, buffers are checked for available data or space. If there is not enough present, the program waits and then rechecks.

## VII. Verification

Due to lack of hardware, Støvneng only verified his changes in simulation. With available hardware and an updated communication unit, the design can finally be properly verified.

### A. Methodology

The communication unit was tested in hardware by connecting the output of the reception buffer to the input of the transmission buffer. Then sample data was sent over PCI Express to the SP605 and then read back.

The updated hardware design was tested with SBM sizes of 8x8 for 2D and 8x8x8 for 3D. The tests are detailed in Appendix A; each one has a short description and a list of the instructions it verifies. Together, the 11 tests cover all instructions.

Note that due to some instructions being dependent on others, it is not always possible to know which instruction is failing.

### B. Results

The communication unit returns the correct data in the correct order, which means it passes its test.

The 2D design passes all tests except for 6 and 8, while the 3D design failes test 1, 3, 6, 7 and 8. This means that the two most crucial components, the SBM and development unit, is working in neither. The 3D design also has some additional issues.

The status of each instruction is listed in Table II, while descriptions of the issues are listed in Section VII-C.

Some issues were solved in the alotted time. They are not included in these results, but are listed in Section VII-D for completeness.

TABLE II
Implementational status of instructions

| Instruction | Works in 2D | Works in 3D |
|---|---|---|
| break | Yes | Yes |
| clearBRAM | Yes | Yes |
| config | Yes | Undecidable |
| devstep | Undecidable | Undecidable |
| doFitness | Undecidable | Undecidable |
| end | Yes | Yes |
| jump | Yes | Yes |
| jumpEqual | Yes | Yes |
| nop | Yes | Yes |
| readback | Yes | Undecidable |
| readFitness | Undecidable | Undecidable |
| readRuleVector | No | No |
| readState | Yes | Yes |
| readStates | Yes | Yes |
| readSums | Undecidable | Undecidable |
| readType | Yes | Yes |
| readTypes | Yes | Yes |
| readUsedRules | Undecidable | No |
| resetDevCounter | Yes | Yes |
| run | Undecidable | Undecidable |
| setNumberOfLastRule | Undecidable | Undecidable |
| startDFT | Undecidable | Undecidable |
| store | Yes | Yes |
| switch | Yes | Yes |
| writeLUTConv | Undecidable | Undecidable |
| writeRule | Undecidable | Undecidable |
| writeState | Yes | Yes |
| writeStates | Yes | No |
| writeType | Yes | Yes |
| writeTypes | Yes | No |

### C. Remaining issues

*ReadStates prints garbage in top 32 bits.* This occurs due to a buffering error in the LSS unit. However, only the least 32 bits are used by the api, which means that this issue only impacts performance and not functionality.

*WriteStates and WriteTypes does nothing in 3D.* This issue is only present on the board, not in any simulations, which makes it hard to track down the cause.

*ReadRuleVector sends incorrect data.* The first execution of the instruction produces an extra word; a repetition of the first. Following executions produces the correct amount of words, but the order is offset by one.

*ReadUsedRules fails for SBM widths less than 16.* The simulator crashes with an index-out-of-bounds error, due to the instruction treating a $[width \cdot 4]$ bits wide signal as if it is 64 bits wide. How ISE is able to implement the design despite illegal indexing is a mystery, but the instruction produces only zeroes when executed on the board.

*Development rules does not activate.* This issue is also only present on the board, making it hard to analyze. The root cause could be with any of the following instructions: devstep, writeRule and setNumberOfLastRule.

*Config does not properly write states in 3D.* Simulations show that the BRAM address fluctuates, causing the states to be overwritten by 0 in the following cycle.

*Runstep causes every state to become zero.* When a runstep is performed, all states in the SBM are reset.

### D. Solved issues

*States and types were written to the wrong location.* When writing single states or types, a half-row is read from BRAM, combined with the new data, and written back to BRAM. Due to the usage of non-implementable code to specify how the data should be combined, the new data always ended up in the middle of the half-row.

*Development ran indefinitely.* Comparison of signals of different widths always return false. Due to the comparison of a parameterized signal with a constant, the development unit would not iterate through the cells, and thus never finish.

*WriteRule did not follow spesification.* When the api transformed a 3D rule struct into an instruction, the position of up/down and north/south were swapped.

*PrintTypes used wrong offsets for decomposition.* When decomposing a row of types into individual types, an offset

of 5 was used instead of 8. This entailed that the printed values appeared as garbage.

*ReadVector and PrintVector used 32-bit words.* The functions were not updated from using 32-bit to 64-bit words. ReadVector would therefore expect twice the number of words provided by the hardware platform, causing the program to wait for nonexistant data.

## VIII. Discussion

### A. Challenges

There was a lot of concern during initial hardware testing, as the SP605 was not detected by the computer. A slightly curved circuit board led to the belief that there might be something wrong with the hardware. Luckily, it proved not to be a hardware fault, but a mistake in the hardware setup guide; the position of SW1 was reversed, causing the board to operate in a completely different mode.

The SP605 was pre-installed with an example design implementing communication over PCI Express with DMA. However, the accompanying driver did not support newer Linux kernels. Additionaly, the design was written in verilog while CARP is written in VHDL, which meant extra effort to integrate the two. There was some effort applied to update the driver, but it was abandoned due to near-untraceable segfaults.

The USB cable driver for usage of JTAG provided by Xilinx also had the problem of not being compatible with newer Linux kernels. Thankfully, a third-party driver found at [21] is compatible and solves the problem.

For unknown reasons, collisions occur on vital signals in all post-map and post-place-and-route simulations[7], causing them to be of no use. This makes it impossible to analyze issues that are present in implementation but not in post-translate simulation. Since ISE will not respect the *keep_hierarchy*[8]attribute for the unit in which the collision is first observed, tracing of the source has been unsuccessful.

### B. Future work

The most important thing going forward is to fix the errors that are preventing the sblockmatrix and development units from working correctly. However, this is no easy task, as large parts of the design are highly complex and difficult to debug. The most extreme cases are the development and LSS units which each consists of a single large file, around 1200 lines long, of complex pipelined code. Simplification and modularization of these units is therefore imperative.

Another reason to simplify the development unit is it's extreme memory bandwidth requirement against the SBM BRAM. Currently, it is designed so that N rules are loaded, applied to every cell, then the N next rules loaded and so on. This means that the BRAM must supply 5 rows each cycle to test 1 row per cycle (or 8 for 2) in 3D, while N rules are needed per matrix iteration. In addition, after the first pass, center cells has to be read from BRAM B instead of BRAM A, since they might have changed.

A simplified process is to read 5 rows, apply one rule to the center row each cycle, then read the next 5 rows, and so on. This will greatly lessen the bandwidth requirements against the BRAM, as new rows can be read in sequence while each rule is being applied to the current. Assuming there are more rules than the number of rows that must be read (highly likely), there is no performance loss. Additionally, this would allow development to only read from BRAM A, simplifying the dataflow.

There are still some remains of having two clock domains, more specificaly a pair of flipflops used for clock-synchronization in the fetch and lss units. It does not affect functionality, and has therefore been of low priority, but it does add a slight delay between the communication unit and the fetch and lss units when reading data.

Currently, the platform only supports SBM sizes that are powers of 2. It would be beneficial to be able to select any size, allowing for fine-tuning of the resource usage, to get most out of the FPGA.

As noted in [18], Støvneng increased the base instruction and data sizes from 32 to 64 bits. Although it is one way to accomodate for longer instructions, the decision is a little odd, considering both PCI and PCI Express are based on 32-bit word sizes. This means that conversion is currently required between the LSS and communication units. Since only 6 out of 30 instructions require more than 32 bits, communication could be simplified and optimized by going back to a 32-bit base size.

The current design makes use of a lot of internal tristate buffers. These are not supported in modern FPGAs [23], and therefore need to be converted into other logic during synthesis. The synthesis tool gently hints at this misuse by producing several warnings. Removing tristate buffers will therefore result in code that more closely relates to its implementation.

Another feature that can beneficially be removed is the global asynchronous reset. Since all Xilinx FPGAs start in a well defined state, a reset signal is only needed in very spesific cases [24] [25]. Otherwise, it only serves to take up valuable resources.

Having the software interface automaticly adapt to the implemented hardware platform would be nifty. This would remove the need to specify the sblock matrix size and type at compilation. It could be accomplished by adding an instruction that returns the size of the sblock matrix.

A feature that could be interesting is the ability to enable or disable wrap-around for the edges of the sblock matrix. Disabling it would mean that the size of the matrix can not be exploited to create an oscillating signal by something continually moving in one direction.

[7] The order of the implementation process is: Synthesize, translate, map, place-and-route.

[8] The *keep_hierarchy* attribute informs the synthesis tool that it should not flatten hardware design units to allow further optimizations. This is useful since the opimizations makes it near-impossible to trace signals.

Finally, a unification of the 2D and 3D designs would give a more generic design and allow less code to be maintained.

With the current need for major fixes, simplification of development and LSS, reducing the need for extreme memory bandwidth, removing tristates, removing resets, and unification, it might be a good idea to rebuild the platform from the ground up. Starting from a clean slate, thoroughly evaluating every part of the design, replacing the bad features and improving the good, will likely result in a greatly improved platform for CA research.

## IX. Conclusion

In this paper, a new hardware platform has been set up. A PCI Express communication unit has been designed and implemented in hardware. A corresponding api has been created in software. It has all been successfully integrated into the the existing platform.

Unfortunately, verification shows that the recently upgraded design has a lot of issues. Some issues could be fixed within the alotted time, but several critical remain. Neither the sblock matrix nor the development unit is functional, which means the platform is currently unusable. These issues along with a long list of proposed changes suggest that a major rewrite might be in order.

## Appendix A
## Test Descriptions

| Test | Description | Verifies |
|---|---|---|
| 0 | Write and read single types | WriteType, ReadType |
| 1 | Write and read multiple types | WriteTypes, ReadTypes |
| 2 | Write and read single states | WriteState, ReadState |
| 3 | Write and read multiple states | WriteStates, ReadStates |
| 4 | Write states and types, clear BRAM, check BRAM is empty | ClearBRAM |
| 5 | Write states and types, switch BRAM, check data is gone, switch again, check data is back | SwitchSBM |
| 6 | Write rules and types, run devstep, check rules have triggered and types have updated | DevStep, WriteRule, ReadRuleVector, ReadUsedRules |
| 7 | Write and read state to/from sblockmatrix | Config, Readback |
| 8 | Write states, types and LUTConv, run sblockmatrix, check states have changed | Run, WriteLUTConv |
| 9 | Store program that prints 1 and then stops, jump to program address 3 times, check for three 1's | Store, End, Jump, Break |
| 10 | Execute program that prints 1, runs devstep and jumps to itself unless 3 devsteps has run, check for three 1's | JumpEqual, ResetDevCounter |

## Appendix B
## Attached Files

*Directory structure:*

- hardware
  - 2D
  - 3D
  - common
    - communication
    - inferers
    - utility
  - ipcore_dir
  - sp605
- software
  - 2D
  - 3D
  - common

*New files:*

- hardware/common/communication
  - *com40_compatibility_layer.vhd*: The compatibility layer between the control unit and the communication unit.
  - *communication.vhd*: The new communication module.
  - *communication_sim.vhd*: A "fake" communication module that provides external access to the buffers, used to allow simulation.
  - *rx_engine.vhd*: The reception engine, responsible for parsing TLPs.
  - *tx_engine.vhd*: The transmission engine, responsible for building TLPs.
  - *rq_special.vhd*: The special request handler.
- hardware/common/utility
  - *combiner.vhd*: A module for inserting an entry into a word, used in the control unit to fix the WriteState and WriteType bug.
  - *fifo.vhd*: A fifo buffer.
  - *shifter.vhd*: A static shifter, used in the dynamic shifter.
  - *shifter_dynamic.vhd*: A dynamic shifter, used in the combiner.
- hardware/ipcore_dir
  - *sp605_pcie.xise*: Project file for the Spartan-6 PCI Express endpoint core.
  - *sp605_pcie.xco*: Core generator file for the Spartan-6 PCI Express endpoint core.
- hardware/sp605
  - *constraints.ucf*: Timing and placement constraints for the SP605.
  - *constraints_sim.ucf*: Timing constraints for simulation.
  - *pcie_wrapper.vhd*: A wrapper for the Spartan-6 PCI Express endpoint core to remove all unneeded signals from cluttering other parts of the design.
- hardware

  - *carp2D.xise*: Project file for the 2D design.
  - *carp2Dsim.xise*: Project file for simulation of the 2D design.
  - *carp3D.xise*: Project file for the 3D design.
  - *carp3Dsim.xise*: Project file for simulation of the 3D design.
- software/common
  - *pci.h*: Header file for pci.c
  - *pci.c*: A general interface to find and memory-map PCI device memory regions.
  - *sp605.h*: Header file for sp605.c
  - *sp605.c*: Provides an open/close/read/write interface towards the communication module implemented on the SP605.

*Modified files:*

- hardware/2D
  - *lss.vhd*: The LoadSendStore unit; a part of the control unit.
- hardware/3D
  - *dev.vhd*: The development unit.
  - *lss.vhd*: The LoadSendStore unit; a part of the control unit.
- software/2D
  - *read_print.c*: Convenience functions for printing states and types.
  - *sblocktest.c*: Tests.
  - *sblocklib.h*: Header file for sblocklib.h
  - *sblocklib.c*: The main API.
- software/3D
  - *read_print.c*: Convenience functions for printing states and types.
  - *sblocktest.c*: Tests.
  - *sblocklib.h*: Header file for sblocklib.h
  - *sblocklib.c*: The main API.

## References


[1] A. Thompson, "An Evolved Circuit, Intrinsic in Silicon, Entwined with Physics," in *Evolvable Systems: From Biology to Hardware*. Springer, 1997, pp. 390–405.

[2] A. Thompson and P. Layzell, "Analysis of Unconventional Evolved Electronics," *Communications of the ACM*, vol. 42, no. 4, pp. 71–79, 1999.

[3] S. Harding and W. Banzhaf, "Artificial Development," in *Organic Computing*. Springer, 2008, pp. 201–219.

[4] G. Tufte, "From Evo to EvoDevo: Mapping and Adaptation in Artificial Development," *Development*, 2008.

[5] S. Wolfram, "Universality and complexity in cellular automata," *Physica D: Nonlinear Phenomena*, vol. 10, no. 1, pp. 1–35, 1984.

[6] C. G. Langton, "Computation at the Edge of Chaos: Phase Transitions and Emergent Computation," *Physica D: Nonlinear Phenomena*, vol. 42, no. 1, pp. 12–37, 1990.

[7] J. von Neumann and A. W. Burks, "Theory of self-reproducing automata," 1966.

[8] E. F. Codd, *Cellular automata*. Academic Press, 1968.

[9] *Spartan-6 Family Overview*, Xilinx, 2011, DS160 (v2.0).

[10] L. Huelsbergen, E. Rietman, and R. Slous, "Evolution of Astable Multivibrators in Silico," in *Evolvable Systems: From Biology to Hardware*. Springer, 1998, pp. 66–77.

[11] *Spartan-6 FPGA Configuration User Guide*, Xilinx, 2014, UG380 (v2.7).

[12] *Virtex Series Configuration Architecture User Guide*, Xilinx, 2004, XAPP151 (v1.7).

[13] P. C. Haddow and G. Tufte, "An evolvable hardware FPGA for adaptive hardware," in *Evolutionary Computation, 2000. Proceedings of the 2000 Congress*, vol. 1. IEEE, 2000, pp. 553–560.

[14] PCI-SIG, "PCI Express Base Specifications Revision 3.0," *PCI-SIG*, 2010.

[15] A. Djupdal, "Konstruksjon av maskinvare for kjøring av sblokkbaserte eksperimenter," 2003.

[16] K. Aamodt, "Kunstig utvikling: Utvidelse av FPGA-basert SBlock-plattform," 2005.

[17] S. Berg, "Evolution of Cellular Automata using Lindenmayer Systems and Fourier Transforms," 2013.

[18] O. M. T. Støvneng, "Extending an sblock platform for a new target hardware," 2014.

[19] *SP605 Hardware User Guide*, Xilinx, 2011, UG526 (v1.6).

[20] DistroWatch, "DistroWatch Page Hit Ranking," http://distrowatch.com/dwres.php?resource=popularity, [Accessed: 2014-12-15].

[21] Michael Gernoth, "XILINX JTAG tools on Linux without proprietary kernel modules," http://rmdir.de/~michael/xilinx/, [Accessed: 2014-12-16].

[22] *Spartan-6 FPGA Integrated Endpoint Block for PCI Express*, Xilinx, 2011, UG672 (v1.1).

[23] D. Koch, C. Haubelt, and J. Teich, "Efficient Reconfigurable On-Chip Buses for FPGAs," in *Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on*. IEEE, 2008, pp. 287–290.

[24] *XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices*, Xilinx, 2012, UG687 (v14.3).

[25] K. Chapman, "Get Smart About Reset: Think Local, Not Global," Xilinx, Tech. Rep., 2008, WP272.