The Cellular Processor Architecture CEPRA–1X and its Configuration by CDL

Christian Hochberger[1], Rolf Hoffmann[2], Klaus–Peter Völkmann[2], and Stefan Waldschmidt[2]

[1] University of Rostock, 18059 Rostock, Germany,
hochberg@informatik.uni-rostock.de
[2] Darmstadt University of Technology, 64283 Darmstadt, Germany,
(hoffmann,voelk,waldsch)@informatik.tu-darmstadt.de

Abstract. The configurable coprocessor CEPRA–1X was developed as a PC plug–in card in order to speed up cellular processing significantly. *Cellular Processing* is an attractive and simple massive parallel processing model. To increase its general acceptance and usability it must be supported by a software environment, an efficient simulator and a special language. For this purpose the cellular description language CDL was defined and implemented. With CDL complex cellular algorithms can be described in a concise and readable form. A CDL program can automatically be transformed into a logical design for the CEPRA–1X. The design is loaded into field programmable gate arrays for the computation of the state transition of the cells. For time dependent or complex rules the design may be reconfigured between consecutive generations. An example is presented to show the generation of logic code.

1 Introduction

Cellular Processing is based on the processing model of Cellular Automata. All cells obey in parallel to the same local rule, which results in a global transformation of the whole generation. The cells are connected to their adjacent cells only. In the two dimensional case 4 neighbours (von Neumann neighbourhood) or 8 neighbours (Moore neighbourhood) are considered. In the three dimensional case up to 26 neighbours can be taken into consideration.

Typical applications are: crystal growth, biological growth, simulation of digital logic, neuronal switching, electrodynamic fields, diffusion, temperature distributions, movement and collision of particles, lattice gas models, liquid flow, wave optics, Ising systems, image processing, pattern recognition and numerical applications.

Cellular algorithms are described in a concise and readable form in the language CDL (Cellular Description Language). CDL has been proved to be very useful for the description of complex cellular algorithms[1]. One version of the compiler generates C or Java code for the software simulator, another version generates a hardware description for field programmable gate array which we use in our coprocessor CEPRA–1X[2].

Cellular processing on a conventional computer is time consuming especially for a large number of cells, complex rules and experiments with parameter variations. Special hardware support is necessary to speed up the computation and for realtime visualisation on the fly.

2 Target Architectures

In the course of the cellular processing project at the Technical University of Darmstadt different architectures have been developed, in particular the CEPRA–8L[3], the CEPRA–1X[2], and the CEPRA–3D[4]. A new designed machine CEPRA–S for general purposes is under development. The advantage of the CEPRA processors compared to CAM[5] machines is that complex and probabilistic rules can be computed in one step, whereas the CAM machines must split the problem into cascaded look–up tables.

Coprocessor CEPRA–1X. The CEPRA–1X coprocessor is a plug–in card for the PCI bus. It was designed for 2D cellular processing with visualisation support, but it can be used as a general data stream processor. The cellular field data is stored in the host. For the computation of a new generation the cell states are streamed to the coprocessor, the rule is computed for all the cells in the stream and the new cell states are streamed back to the host.
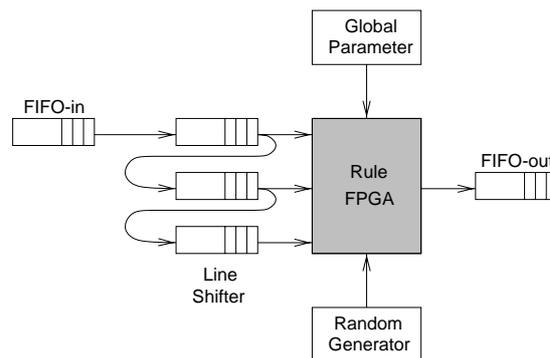

Fig. 1. three line FIFOs

The rule is computed by an FPGA (field programmable gate array) which has to be loaded with a configuration, describing the logic design of the rule. Because three lines are buffered (implemented as FIFOs) each cell has to be read and written exactly once. With the PCI–bus–performance of 133 MByte/sec the performance is 30 million 2D–16–Bit cell operations per second with 9 neighbours. Considering the Belousov–Zhabotinsky reaction described later this is a speed up of about 40 in comparison to a 133 MHz PC. More complex rules will yield higher speed ups, because we use hardware pipelining in the CEPRA–1X. Therefore the computation time is independent of the the rule complexity.

The logic design which has to be loaded into the FPGA is generated by the CDL hardware compiler. The compiler generates intermediate logic code

(*VERILOG*) which is transformed into FPGA configuration data by a tool from XILINX.

The logic description of different rules can be reloaded between the computation of the generations. By this technique time dependent rules can be computed. Complex rules which do not fit into the FPGA can be broken into a sequence of phase rules. The phase rules are loaded between the phases of the generations. The time to reload the FPGA (parallel mode, 8MHz) is 15% of the computation time for a cell field of size 1024×1024.
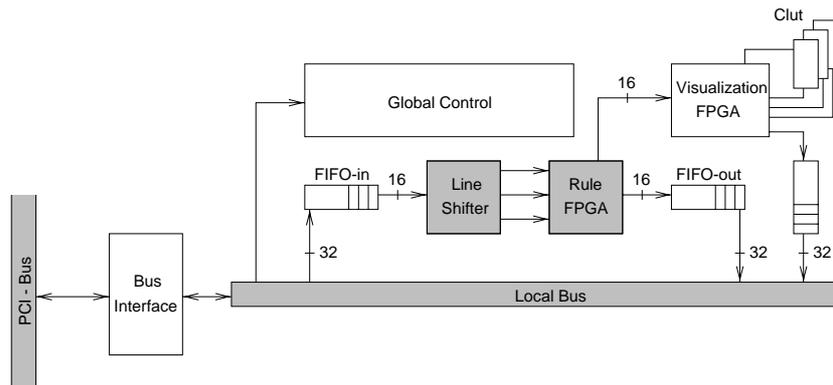

Fig. 2. CEPRA–1X architecture

Software Simulator. For the evaluation of cellular algorithms we have developed a simulator software. Experiments for this simulator consist of three basic parts: the description of the rule, the initial state of the cells in the array and some information about the visualisation.

The simulator allows the user to store the cell state in a structured datatype. One of the easiest and most often used visualisation concepts is the assignment of colours to cell states. Thus the simulator provides a visualisation tool that uses one of the cells components as an index into a colourmap.

The rule is written in C or Java and is linked with a kernel which controls the simulation. The kernel provides a `neighbour` function for the access to the neighbours. The kernel is capable of calling different rules depending on the position within the cellular field. By this technique special rules for borders and corners can be defined.

3 CDL, a Language for Cellular Processing

Until now cellular algorithms are programmed in simulator dependent special languages and data structures. Thus the programmer needs special knowledge of the target architecture, which makes programming a tedious task. The CEPRA–1X processor is programmed in VERILOG, whereas the software simulator is programmed in C or Java. Neither of those languages is convenient and adequate for the programmer to describe cellular algorithms. Also both languages contain

elements that are not required for this purpose (e.g. pointer and dynamic memory allocation in C).

The new language CDL was defined with respect to readability, conciseness and portability. While developing a cellular algorithm it is desired to have short turn–around cycles. Thus the usage of a highly interactive software simulator is recommended during the development process. After having tested the algorithm on the software simulator it can be transferred to the CEPRA–1X for fast execution and realtime visualisation.

Features of the Language. The language CDL is intended to serve as an architecture independent language for cellular algorithms. The programmer's benefit is obvious: Switching the target architecture does not require more than just a new compiler run. Moreover CDL contains special elements that make the description of complex conditions very easy (groups, special loop constructs). These elements allow the description of situations like:

- Is there any neighbour that fulfils a certain condition? (`one()`)
- Do all neighbours fulfil a certain condition? (`all()`)
- How many neighbours are in a certain state? (`num()`)

CDL does not contain conditional loops, which has two positive side effects. (1) It enforces the termination of the rule because it is impossible to write endless loops and (2) it enables the compiler to unroll all statements which is extremely important for the synthesis of hardware. CDL allows the user to describe the cell state as a record of arbitrary types. All common data types are available in CDL (integer, boolean, float, etc.). In addition the user can define new types (enumerations and subranges of integers or enumerations).

Example. To give an impression of a CDL program we present the Belousov–Zhabotinsky reaction[6]. It does not show all the special features of CDL, but demonstrates some of the problems that have to be handled quite differently on hardware and software simulators.

```
(1) cellular automaton Belousov_Zhabotinsky ;
(2)
(3) const dimension = 2 ;  // a two-dimensional grid
(4)       distance  = 1 ;  // allow/restrict Moore-neighbourhood
(5)       maxtimer  = 7 ;  // a local constant
(6)       cell      = [0,0]; // relative address of actual cell
(7)              // *[0,0]   means the contents of the cell
(8) type celltype = record  // celltype defines possible states
(9)         active : boolean;
(10)        alarm  : boolean;
(11)        timer  : 0..maxtimer;
(12)      end;
(13)                     // addresses of all 8 Moore-neighbours
(14) group   neighbours={[-1,0],[ 1,0],[0, 1],[ 0,-1],
(15)                     [ 1,1],[-1,1],[1,-1],[-1,-1]};
```

```
(16) colour                    // description of visualisation
(17)    [0  , 255, 0]      ˜      *cell.active and     *cell.alarm;
(18)    [255,   0, 0]      ˜      *cell.active and not *cell.alarm;
(19)    [*cell.timer * 255 div maxtimer,0,0]   ˜ not *cell.active;
(20)
(21) var   neighbour : celladdress;     // local loop variable
(22)
(23) rule begin
(24)    *cell.active := *cell.timer=0;     // is actual timer==0?
(25)    *cell.alarm  :=        // count neighbours in active state
(26)         num(neighbour in neighbours : *neighbour.active)
(27)                                        in {2,4..8};
(28)    if *cell.active and *cell.alarm and (*cell.timer=0)
(29)      then *cell.timer:=maxtimer
(30)      else if *cell.timer!=0 then *cell.timer:=*cell.timer-1;
(31) end;
```

The type celladdress, as used in line (21), is implicitly defined by the compiler from the two constants dimension and distance. They define how many dimensions the model uses and how far the access to other cells reaches. Both constants must be supplied by the programmer. The type celladdress is a record with as many components as the model has dimensions. Each component can have a value between -distance and +distance. Lines (14) and (15) show the celladdresses of all eight Moore neighbours. The name of this enumeration does not have any meaning for the compiler. The elements are used in the iterative num–loop in line (26).

4   Transformation into a Hardware Description

Even simulators that are based on specialised hardware are supported by CDL. The CEPRA–1X simulator has been chosen as an example during the design phase of CDL.

The most important restrictions of a hardware simulator are the limited number of cell states and the limitations in the rule complexity. Although floating point numbers are desired and should be included in a cellular language, they are usually not implemented in a specialised hardware simulator because of hardware costs.

Celltype. In the case of CEPRA–1X the states of the cell must be coded with 16 bits. If the celltype is a record (as in lines (08)–(12)) it would me more easy to reserve bit groups for the subtypes of this record (one bit for each boolean in lines (09)–(10) and three bits for the integer subrange in line (11)). Usually, this will simplify the logic for the rules, because often the rules access only components of the cell record (e.g. line (28)). On the other hand, this may lead to a state coding, where not all 2^{16} states can be used (e.g. if the integer subrange does not have power of two elements). Enumerating all possible cell states (the power set of the components) will not waste any of the states, but will
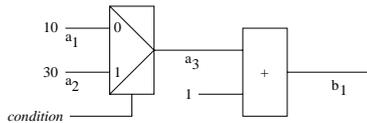
Fig. 3. The implementation of local variables and assignments

increase implementation cost. The CDL compiler decides itself which method to use.

Variables. The classical synthesis approach uses registers to represent variables. The data paths between these registers are controlled by a finite state machine. For the CEPRA–1X machine this is not desired, because it would imply the usage of a clock signal. The number of clocks required to complete the calculation would then depend on the data. The varying time could stall the pipeline and slow down the calculation speed.

To simulate CDL variables, they are represented by local signals. Because a new value can be assigned to signals only once, for each assignment new signals must be created.

The following CDL fragment

```
(1) a:=10;
(2) if condition then a:=30;
(3) b:=a+1;
```

produces the local variables a_1, a_2, and a_3, a multiplexer driven by `condition`, and a following adder which calculates the value of signal b_1 (Fig. 3).

Optimisations. The hardware resources inside a FPGA are limited. Therefore optimisation is necessary. The optimisation supported by the VERILOG compiler is good but not sufficient. The CDL compiler already should keep an eye on the complexity of the description. It should not use too many local signals and avoid generating unused code. To reduce implementation cost, early expression and condition evaluation is necessary and was implemented. The compiler evaluates constant expressions during compilation, taking special properties of the operation into consideration. The `or` operation, for example, with one operand being constant `true` is evaluated during compilation and translated into the constant `true`.

Usually a data type is represented by a fixed number of bytes on common computers. To reduce implementation cost, the compiler should use single bits instead of bytes as the smallest unit. In addition, the size of a data type may vary. For example a variable of an integer subrange type, which is divided by two will need one bit less after the division. Therefore it is useful to know the exact range of possible values for each variable and expression.

Loops. To simulate the behaviour of a loop, hardware must be generated for each iteration. Conditional loops are not available because the number of iterations can not be determined during compilation. (This is equivalent to the demand that calculation must always terminate.)

The `num` expression in line `(30)` can be interpreted as a loop. The constants of the group `neighbours` are assigned to the variable `neighbour` one after the

other. After each assignment the expression `*neighbour.active` is evaluated and the result is assigned to a new local signal. After the eight iterations, the eight signals are connected to a logic, which sums up the conditions that are true. The sum is the result of this expression.

Conditional Statements. The only statement which has a permanent effect is the assignment of a value to a variable or the cell state (e.g. line `(28)`). For this reason the assignment statement is affected by the corresponding condition. Have a look at line `(35)`. Only if the condition is true, the assignment shall have an effect. Therefore each assignment is implemented as a two–to–one multiplexer, where one input is the old value and the other is the new value. The select signal of this multiplexer is connected to the condition of the surrounding conditional statement. For nested conditional statements their conditions are combined using the logical `and` operation. An `else` part can be realized using the inverted condition and a `case` statement using different cascaded conditions.
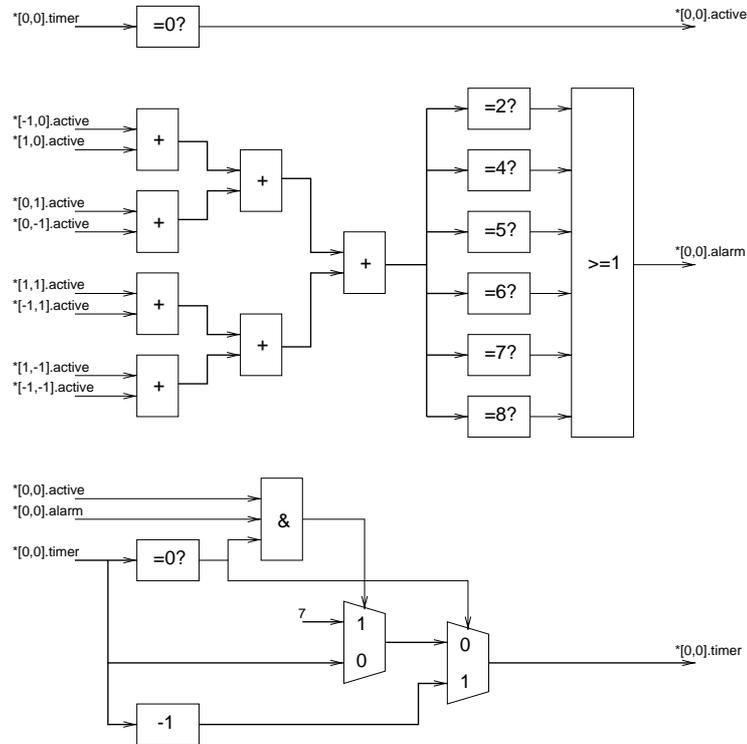

Fig. 4. result of synthesis

Complete Example. The CDL program describing the Belousov–Zhabotinsky reaction from the previous section results in a hardware structure shown in Fig. 4. Obviously program line `(24)` corresponds with the upper part of the logic. The

middle part corresponds to lines (25)-(27). And the lower part of the logic has been synthesised form lines (28)-(30).

Recognise the power of the num() statement. Only three lines of code result in the large amount of the middle part of the logic.

Colour. The colour definition must be loaded into the CRT controller as a look–up–table. To create this look–up–table during compilation, each possible cell state is associated with the contents of the cell (*[0,0]) and the expressions in the colour definition (lines (17)-(19)+) are evaluated.

5   Conclusion

The CEPRA–1X is a configurable coprocessor which speeds up cellular processing significantly. As it processes data streams it can also be used for other applications. The resulting pixel stream can be coloured and visualised in real-time. Complex rules and time dependent rules can be computed by reloading the FPGA between the generations.

CDL is an implemented language for the concise, readable and portable description of cellular algorithms. One version of the compiler generates C/Java–code for the software simulator. Another version generates logic equations for the field programmable gate arrays of the CEPRA–1X machine. The logic equations are partly minimised by the compiler and partly by a commercial available design system.

Main features of the language are records, unions, groups and the loop construct for testing complex conditions. The language can be used to describe complex cellular algorithms of practical relevance. Based on the experience the language was extended to CDL++[7] for the description of moving objects.

References

[1] Christian Hochberger, Rolf Hoffmann, Klaus-Peter Völkmann, and Stefan Waldschmidt. Cellular processing environment. In Bogusław Butryło, editor, *International Conference on Parallel Computing in Electrical Engineering (PARELEC 98)*, number 1, pages 171–174, Białystok, Poland, 1998. Technical University of Białystok.

[2] Christian Hochberger, Rolf Hoffmann, Klaus-Peter Völkmann, and Jens Steuerwald. The CEPRA–1X cellular processor. In Rainer W. Hartenstein and Viktor K. Prasanna, editors, *Reconfigurable Architectures, High Performance by Configware*. IT Press, Bruchsal, 1997.

[3] Rolf Hoffmann, Klaus-Peter Völkmann, and Marek Sobolewski. The cellular processing machine CEPRA–8L. *Mathematical Research*, 81:179–188, 1994.

[4] R. Hoffmann and K.-P Voelkmann. Hardware support for 3D cellular processing. *Lecture Notes in Computer Science*, 1277:322–??, 1997.

[5] Norman H. Margolus. CAM–8: a computer architecture based on cellular automata. Technical Report 01239, MIT Lab. for Computer Science, December 1993.

[6] A. Zaikin and A. Zhabotinsky. *Nature*, (225):535–, 1970.

[7] Christian Hochberger. *CDL — Eine Sprache für die Zellularverarbeitung auf verschiedenen Zielplattformen*. PhD thesis, Darmstadt University of Technology, 1999.