

A Simulator for the Reconfigurable Mesh Architecture

Carsten Steckel¹, Martin Middendorf¹,
Hossam ElGindy², Hartmut Schmeck¹

¹Institut für Angewandte Informatik und
Formale Beschreibungsverfahren, Universität Karlsruhe
D-76128 Karlsruhe, Germany

²Dept. of Electrical and Computer Engineering,
University of Newcastle, NSW 2308, Australia

Abstract. In this paper we present a simulator for the reconfigurable mesh SIMD architecture. The purpose of the simulator is to assist in the analysis of algorithms and the visualisation of their behaviour. Furthermore, it can be used to demonstrate the potential of reconfiguration in an educational environment.

1 Introduction

The most distinctive feature of a reconfigurable parallel computer is the availability of a variable network topology allowing each node's neighborhood to be changed on the fly. It is able to map closely to the algorithm's communication needs at any given time and to cut the interprocessor communication cost [3]. For a broad range of problems the transfer of calculations into the dynamic reconfiguration of bus structures has led to a significant reduction in computation times (e.g. [4]).

We report on a tool to simulate the reconfigurable mesh architecture, which achieves reconfigurability through the use of locally controlled switches. The goal is to keep the interface between the simulator and the model simple and still be capable of incorporating new restrictions or ideas with little effort. The interface supports visualisation of an implemented algorithm in action to help analyse and understand its properties. It can furthermore be used to demonstrate the capabilities of dynamic reconfiguration in an educational environment.

2 Simulated Model

A reconfigurable mesh consists of an $n \times m$ grid of processing elements (PE). Every PE has four ports (north, east, south, and west) enabling it to connect to its neighbors. The linked ports form the static mesh topology (it can be enhanced by wrap around links to form a torus). Every PE can connect some of its ports by switched lines, thus forming the dynamic topology. The configuration of a

PE is denoted by a set of strings such as $\{NS, EW\}$, where the north and south port and the east and west port are connected.

The PEs are based on the processors defined in the RAM Model. They have a constant number of registers and are capable of executing synchronously RAM instructions as well as communication and configuration commands. The computational cycle is made up from a constant number of RAM assembler instructions usually in the following order: 1. Configure switch, 2. Write to bus, 3. Read from bus, and 4. Compute.

One may assume that a signal can propagate through the interconnecting network in constant time. However, experiments have shown that it might be more realistic to assume the signal to travel a certain length of the bus in each cycle. To model this constraint, the number of bus segments that a signal is able to travel in one cycle may be restricted to a constant [1]. The model allows concurrent reading from a bus and requires exclusive writing (CREW).

The simulated model assumes an SIMD array with one array control unit (ACU). The assembler instructions are fixed and atomic.

An important concept in the design of the simulator is that the computational cycle is a “user defined sequence of instructions”. To mark the end of a computational cycle a special mark is placed in the source code. The CREW bus communication assumption requires that two operations writing onto the same bus are separated by such a marker. Otherwise, a write error will stop the simulation. The concept of user defined computational cycles allows to execute a complex instruction of an abstract machine in one computational cycle. This facilitates the simulation of various computational models and paradigms.

To locate or identify a PE, the PEs have an identification number and a position in the grid. The PEs are numbered rowwise from top to bottom starting with zero. The PE in the upper left corner has identification number zero (PE[0]) and coordinates $(0, 0)$ (P[0][0]).

3 Concept and Design

Here we demonstrate the assembler instruction set, the graphical user interface, and output files containing information about a simulation run. Using a running example we outline the capabilities and limits of the simulator as they occur.

The simulator can be started in two ways. In *display mode*, using the graphical environment, or in *test run mode*, where an algorithm is merely executed and results are written into two output files (One contains the log-messages, the other statistical data). The simulator assumes that each PE has a constant number of registers $R1, R2, \dots$ plus an accumulator AX (or $R0$). Each PE has an idle flag IDL containing the current status of the PE (executing or idle). The four ports of each PE are enumerated clockwise starting with the north port P0.

3.1 Assembler Instructions

Each source file is divided into two sections: the parameter block, defining the properties of the virtual machine, and the program in form of assembler instruc-

tions.

Every Boolean or arithmetic instruction takes one argument (a constant or a register). Every configuration instruction takes two ports as arguments. The *WRITE* to bus instruction takes a value or register and a port. The *READ* from bus command takes a port and a register. Two constants describing the mesh's size are available (*_N* width, *_M* height). The PE's coordinates are (*_X* column, *_Y* row) and their identification number is *PID*. The idle flag can be set by storing a nonzero value to *IDL*. The programmer has to make sure to save the idle flag onto the flag stack, if he wants to recover the state later on. Only nested jumps are allowed, because the idle mask is stored on or retrieved from a stack by the *PUSHF* and *POPF* instructions.

The assembler instructions are now explained in more detail using code fragments taken from an implementation computing the logic OR of n^2 bit values stored in the PEs of an $n \times n$ no wrap around mesh. In case no PE has written to a bus a zero will be read. Each PE uses two registers R_1 and R_2 : the first one stores the bit value, the second one temporarily stores expression results. The program is based on the following algorithm:

- (1) Configure a snake on the mesh.
- (2) **if** a PE contains a '1' **then**
- (3) Break the snake up into segments.
- (4) Write a '1' to the port that points to the head PE[0] of the snake:
 - west for even rows and not the first column,
 - east for odd rows and not the last column,
 - north for even rows and first column, odd rows and last column.
- (5) **end if**
- (6) The head of snake reads from its northern port and stores the result.
- (7) PE[0] broadcasts the result to all PEs.

The parameters needed to define the simulator model for a test run on a mesh of size 10×10 are given as follows:

```
.DIMENSION 10 10      ; size of mesh
.WRAPAROUND N        ; no wrap-around
.BUSWIDTH 16         ; width of buses and registers
.REGISTERS 3         ; 2 registers + accumulator
.KCONSTRAINT 100     ; maximum signal propagation distance per cycle
                    ; in number of bus segments
.BUSDEFAULT 0        ; value to be read, if nothing was written
.INPUTDIR './or.in' ; where does the initial data come from?
```

Following the parameter block is the program. To configure a snake, we divide the set of PEs into five subsets. The 'inner' block are all PEs which are not in the leftmost or rightmost column. The leftmost (rightmost) column is split into PEs in even and odd rows. To identify the 'inner'-block we code

```

LOAD _X      ; load PE's column number into accumulator
CMP 0        ; compare with zero writes -1 iff less, 0 iff
              ; equal and 1 iff greater to accumulator
NOT AX       ; iff AX is zero writes 1, else
              ; writes 0 to accumulator
STORE R2     ; save temporary result in register R2
LOAD _X
ADD 1        ; because counting starts with zero
CMP _N       ; compare with x-dimension
NOT AX
OR R2        ; if in leftmost or in rightmost column
STORE R2     ; save for later use
PUSHF        ; remember the idle state
STORE IDL    ; switch off processors with 1 in AX

```

After determination of the idle mask, set configuration for the 'inner'-block

```

CON P3,P1    ; configure WEST, EAST
POPF
# _CC        ; mark the end of a computational cycle

```

and turn all PEs back on by remembering their last idle state. The other configurations are set in a similar fashion. The next step is to break up the snake into segments at PEs storing a one in register R1.

```

LOAD R1
NOT AX
PUSHF
STORE IDL
CLR          ; clear the switch
POPF

```

PEs containing a one write a signal towards the head of the snake. Here is the code for the case of PEs in even rows and not the leftmost column which write a token to their west port:

```

LOAD _Y
MOD 2
CMP 0
PUSHF
STORE IDL    ; turn off odd rows
LOAD _X
CMP 0
NOT AX
STORE IDL    ; turn off leftmost column
WRITE P3,1   ; write token to west port
POPF
# _CC

```

It remains for PE[0] to read from its northern port and to store the result.

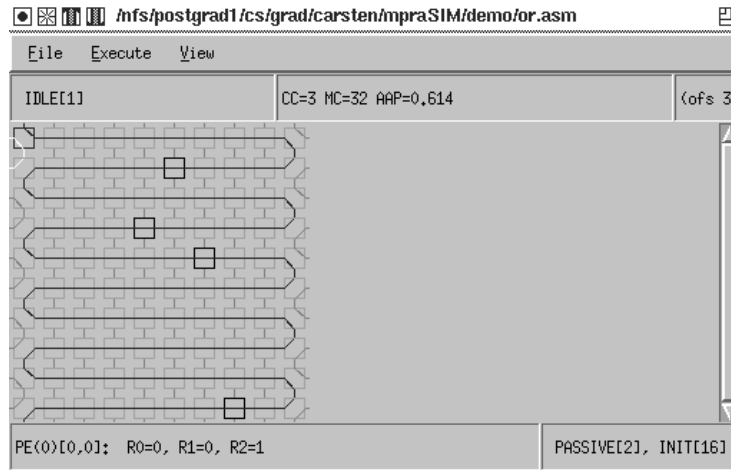


Fig. 1. A configured snake with the PEs active that contain a one

3.2 Graphical Interface and Output Files

The graphical interface consists of two windows: One shows the mesh of PEs and their status, the other displays the source code with the currently executed instruction marked.

Figure 1 shows a snap shot of the first window, while animating through the program. A snake is being configured and PEs containing a one are active (depicted by a black frame whereas the passive PEs have a grey frame). The machine reports its status beneath the menu bar. It is idle (i.e. not executing an assembler instruction) and the last executed assembler instruction has offset 35 in the programm (ofs 35). The machine has used three computational cycles (CC=3), and it is currently in its 32nd machine cycle (MC=32). The average number of active PEs has been 61.4% (AAP=0.614). The bottom message bar indicates the state of the currently selected PE (PE[0,0]; the selected PE is depicted with a red frame), its register values and idle state (it is passive and has been initialised correctly).

Figure 2 shows the source window, which displays the executed assembler program. The user has the option to step through the program, animate the execution or run the program. The last two columns of the source window currently display essentially the same information. This will change, as soon as a high level language is available. The atomic assembler instructions will be displayed in the second last column, and the actual high level source code in the rightmost column.

Each run of the simulator produces two output files. The log file contains all model dependent messages and irregularities of the run. Statistical information about the run is stored in a statistics file.

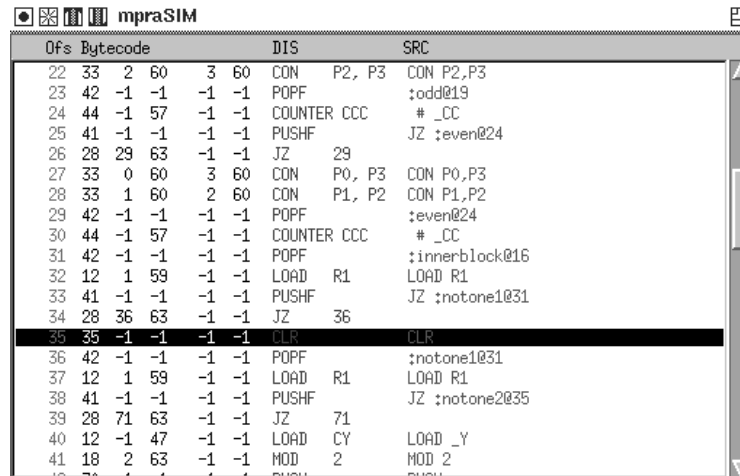


Fig. 2. The source window displaying the current instruction

4 Conclusion

We have reported the design and application of a simulator for the reconfigurable mesh architecture. It is programmed by a fixed instruction set based on the RAM model and enhanced by certain bus communication and configuration commands. The flexible definition of a computational cycle allows to simulate various abstract machines. The use of the simulator was demonstrated by a number of basic computing tasks and also a sparse matrix multiplication algorithm (see the full paper [2]). Future projects are:

- A high level language interface for the simulator should be developed to reduce the time consuming task of implementing algorithms in assembly language.
- The simulator itself should be parallelized in order to speed up the execution and allow the simulation of large meshes in reasonable time.

References

1. B. Beresford-Smith, O. Diessel, H.ElGindy. Optimal algorithms for constrained reconfigurable meshes. *J. Par. Distr. Comput.* 39 (1996) 74-78.
2. C. Steckel, M. Middendorf, H. ElGindy, H. Schmeck. A simulator for the reconfigurable mesh architecture. TR 374, Institute AIFB, University of Karlsruhe (1997).
3. H.Li, Q.F. Stout. Reconfigurable massively parallel computers, *Prentice-Hall* 1991.
4. R. Miller, V.K. Prasanna-Kumar, D.I. Reisis, Q.F. Stout. Parallel computations on reconfigurable meshes. *IEEE Trans. on Computers* 42 (1993) 678-692.

This article was processed using the L^AT_EX macro package with LLNCS style