UNIVERSITY OF JOENSUU

DEPARTMENT OF COMPUTER SCIENCE

Report Series B

# PRAM EMULATOR

## User's Manual

Pasi Hämäläinen

Report B-1992-2

ACM D.1.3

**TABLE OF CONTENTS**

**LIST OF FIGURES**

# 1. INTRODUCTION

*PRAM emulator* is a software emulator of a *theoretical model of parallel computing* called *Parallel Random Access Machine* or *PRAM* (*P-RAM*) for short. Although there exists no real PRAM machine, the execution of one can be emulated in a sequential computer .

*Parallel computing* is a popular research area in computer science. Parallel computing differs from the conventional *sequential computing* so that when in sequential computing only one processor is used in computation, in parallel computing the problem is divided into several subproblems that are solved using a set of processors that work simultaneously , i.e. *in parallel* .

In conventional sequential computing, at least in the computer technology , very much has already been done that is possible to do to speed up computations. Processor speed and integration density are about to reach their theoretical limits, but the desire for more computing power is still growing. For example, areas likes weather forecasting and complex simulations at large require huge amounts of computing power . And the more computing power is available, the better results are possible to achieve, or at least the results are achieved in shorter time. There are also many theoretical and practical problems the solutions to which are beyond the power of sequential computing. It is also true that when computing power grows, new problems arise that are nearly able to be solved. So the growth of computing power kind of feeds itself.

In parallel computing, a problem is spread to a set of processors that work on the subproblems in parallel. Each processor solves a part of the problem and when the subsolutions are combined, the solution to the whole problem is obtained. Because the subproblems are solved simultaneously , the whole problem can be solved in a fraction of time that would be needed in sequential solution. The minimum theoretical time the parallel solution takes is $T/P$, where $T$ is the time taken by the sequential solution and $P$ is the number of processors in the parallel solution.

One might think that by increasing the number of processors the time needed could be reduced to as short as desired. But the things are not that simple and easy . One of the most difficult and also most important questions in parallel computing is how to divide problems into parts that can be executed in parallel efficiently . *Efficient* , or *cost-efficient* , parallel solution means that the product of the time taken by the parallel solution and the number of processors should not be much larger than the time taken by the best sequential solution. If the product is the same as the sequential time, then the parallel solution is *optimal* . The product of parallel time and number of processors represent the amount of total work done or the efficiency of the usage of the processors. Sometimes the parallel time can be reduced to very short but then the processors are usually great deal of the time idle. The intention is that all the processors would work most of the time and that the time would be as short as possible.

Often an efficient parallel solution is kind of multi-level, for example tree-like. The problem is first divided into subproblems that are further divided into subproblems. Or when combining the subsolutions, a set of subsolutions is first combined to a smaller set of subsolutions. In such solutions the processors should work on several subproblems for the solution to be efficient. This causes problems concerning *processor allocation*.

In parallel computing there is usually very much *interprocessor communication* because problems must be divided and solutions must be combined and intermediate results and some other information must be passed from processor to another. The processors of a parallel computer must be connected together, usually with some kind of network, so that interprocessor communication would be possible. In real parallel computers the interprocessor communication is the most serious bottleneck. Because a complete network between a set of processors is impossible to built, the degree of the network must be bounded. This drops down the amount of data that can be transferred in the network simultaneously. The parallel algorithms should thus minimize the need for interprocessor communication.

The area of parallel computing is very wide and parallelism exists on different levels. The processors inside computers handle the bits of one word in parallel. Some computers watch the sequential instruction stream they are executing and if they find successive instructions that are independent and can be executed in parallel, pass the instructions to a few processors to be executed in parallel. Even computers that have tens of thousands of processors connected together according to some network topology are built and used.

In theoretical computer science, researchers try to develop as efficient parallel algorithms as possible for different theoretical models of parallel computing and even for real machines. The theoretical limits of parallel computation are also studied. One area is the simulation algorithms between theoretical and easy to use models of parallel computing and more practical and easier to implement models of parallel computing. Also good parallel programming languages are tried to develop.

Parallel computing has been practised since 1960's. The best practical applications are usually in some special area; when the number of processors has grown, the general usability of the parallel computer has usually reduced. The structure of a parallel computer, for example the topology of the network connecting the processors, determines what kind of parallel processing is efficient in the machine. Most of the parallel computers are used in raw number crunching. Nowadays much research is done for finding good models for *general purpose parallel computers*.

The idea of *massively parallel computers*, i.e. computers that have tens or hundreds of thousands of processors, is very attractive and much research has been done for massively parallel computing. Unfortunately massively parallel computers are difficult or impossible to implement with the present technology. The amount of wire to connect the processors and the processors itself need quite a lot of space. If the processors are near each other, they produce too much heat to the small space. On the other hand the distance of processors may not be too long because some kind of synchronism is needed; it the distance of the processors grows, then the clock rate of the machine must be reduced so that a signal has time to reach each processor within one clock cycle. Because the amount of interprocessor communication grows when the amount of processors grows and because the degree of the network must be very small with respect to the number of processors, the communication bottleneck becomes very serious and the value of such a computer becomes questionable.

One can, however, newer predict what will be possible in the future. Maybe optical technology makes interprocessor communication more efficient. Maybe some other technology will be developed. And maybe good theoretical results stimulate the development of technology. We undoubtfully need research where were accept the limits of the present technology. Quite a lot can be done with parallel computers that are implementable at present. Theoretical research of strong but unrealistic models of computation does not cause any harm - hopefully. If the models someday become realistic, then the theory is ready. And the research may produce results that are already usable.

There are many theoretical models of parallel computing. Models can be divided into two classes: *distributed memory models* and *shared memory models*. In a distributed memory model, the memory of a parallel computer is distributed among the processors. The processors are connected with some network. If a processor needs some data and the data is in another node, the processor needs to request the data from the other processor. In a shared memory model the processors have one shared memory that is used as a communication link between processors. One such model is the PRAM model.

The PRAM model is quite straightforward generalization, or “parallelization”, of the *RAM model (Random Access Machine)* familiar from sequential computing. In nutshell, PRAM consists of $P$ RAMs that have a shared memory which the RAMs can access independently from each other and in constant time. The shared memory acts as a very efficient communication link between the RAMs: communication between any two processors takes one memory write and one memory read operation, i.e. a constant time. This is the feature that makes the model so strong, general purpose, and (in principle) easy to program and analyse. But this same feature makes the model impossible to implement in large scale, at least if we think of the present technology.

The model has it's theoretical and even practical values, or at least they are worth considering. As one of the strongest models of parallel computing, PRAM is a good model when researching the theoretical limits of parallel computing. There are many parallel algorithms developed for the PRAM model. One practical idea is that PRAM could be the machine model that the programmer sees, and that PRAM could be automatically simulated in some real parallel computer, e.g. in some processor network.

Building large PRAMs (more than hundreds of processors) isn't very feasible at present, but a software emulator for the model can be easily constructed, as was done in this special project. Motivation for the PRAM emulator at the beginning of the project was that it would be nice to be able to write parallel algorithms for PRAM in practice and really test them, and to be able to try to write a high level parallel language compiler for PRAM. Such a compiler for parallel modula-2 has been constructed as another special project (Juvaste 1992a, Juvaste 1992b). The emulator along with a high level parallel language compiler can be used to analyse real PRAM programs to see what kind of properties is required from a real machine that would execute PRAM programs. The emulator can be used as a teaching aid in parallel computing courses.

In practice the emulator is controlled by the user via a simple *assembly style programming language*. The system includes *an assembler* for the language and the actual *emulator* that executes the assembled programs. The PRAM model does not include any input or output facilities but the emulator system contains *an interactive monitor program* that can be used to follow the execution of a PRAM. The the emulator may be used as a target "machine" of a high level parallel programming language compiler.

The emulator system is written in ANSI C programming language (Kernighan and Ritchie, 1988, Plauger and Brodie, 1989) and is distributed in that form. Only the curses terminal screen handling package (Programmer's Reference Manual for Curses on the Sun WorkStation, 1986) is used in addition to the standard ANSI C language and library. Therefore the emulator is portable provided that an ANSI C compiler a curses package are available.

This User's Manual is aimed for the user of the PRAM emulator system. Knowledge of parallel computing is not absolutely necessary, but some might help. The document starts with the definition of the used PRAM model. Then follows a general description of the emulator system and detailed instructions for the user of the emulator system. Installation guide of the emulator system is in appendix. The other appendix contains some references to different PRAM models. The documentation of the PRAM emulator system includes also Programmer's Manual that contains some technical documentation for persons doing possible future updates or persons just interested in the program itself.

## 2. PRAM MODEL

Before an emulator for a model of computing can be designed and implemented or such a system described for the user, the model of computing must be defined. Defining the PRAM model is not as easy as it might seem. Researchers using the PRAM model tend to give to their models slightly different properties than the others. Most differences concern instruction sets and memory size limits. The model is not as stable as the RAM model in sequential computing. Different definitions of the PRAM model may be found in the references listed in Appendix B.

The PRAM model in this project is supposed to be "primitive" or "ascetic" and it should not have "too strong" properties. These requirements are somewhat ambiguous, but the (ambiguous) ideas behind these terms are something like the following. The instruction set should not include operations which are more strong than simple arithmetic functions and memory access operations. The model should not be dependent on any "smart operating system" that has strong features and abilities. But the model should be strong enough to be usable, i.e. programming should be easy enough and, for example, writing a high level language compiler for the PRAM should not be impossible.

The basis of the PRAM model used in this project was the model presented in the lecture notes of a parallel programming course by Jyrki Katajainen (1990). The model got to the final form after discussions with professor Martti Penttonen, the instructor of this work.

### 2.1 Basic Properties

A *PRAM* consists of $P$ processors, a shared memory, and *a common clock*. Here $P$ is an arbitrary (positive) integer. The *processors* are identical RAMs (Random Access Machine), and each processor has $R$ *general read-write registers*, where $R$ is an arbitrary integer greater than or equal to two. General registers are indexed from 1 to $R$. Register number 1 is *the accumulator*. In addition to $R$ general registers, each processor has it's own *program counter* (*PC*) and a read-only *signature register* (*SIG*) containing unique *processor identifier* (processor index). Processors are indexed from 1 to $P$. The value of the program counter can be read into the accumulator with a special command and set with jump-commands (otherwise the program counter is increased by one after each clock cycle). Each RAM contains an identical *program*, which the RAMs execute *synchronously* instruction by instruction. Execution of one instruction takes one *clock cycle*. Because each RAM has a distinctive processor identifier, each RAM can identify itself and branch to a different part of the program than the other RAMs.

*The shared memory* consists of unlimited number of memory cells indexed with 0,1,2,.... Each memory cell can contain an arbitrary signed integer. Each processor can access the shared memory simultaneously . Depending on whether simultaneous reading from a same shared memory cell is allowed or not, two *read models*, *concurrent read* model, or *CR* for short, and *exclusive read* model, or *ER* for short, can be defined. Similarly two *write models*, *concurrent write* model, or *CW*, and *exclusive write* model, or *EW*, can be defined, depending on whether simultaneous writing to a same a shared memory cell is allowed or not. From these two possible read models and two possible write models, four possible PRAM models can be combined: *EREW*, *ERCW*, *CREW*, and *CRCW*.

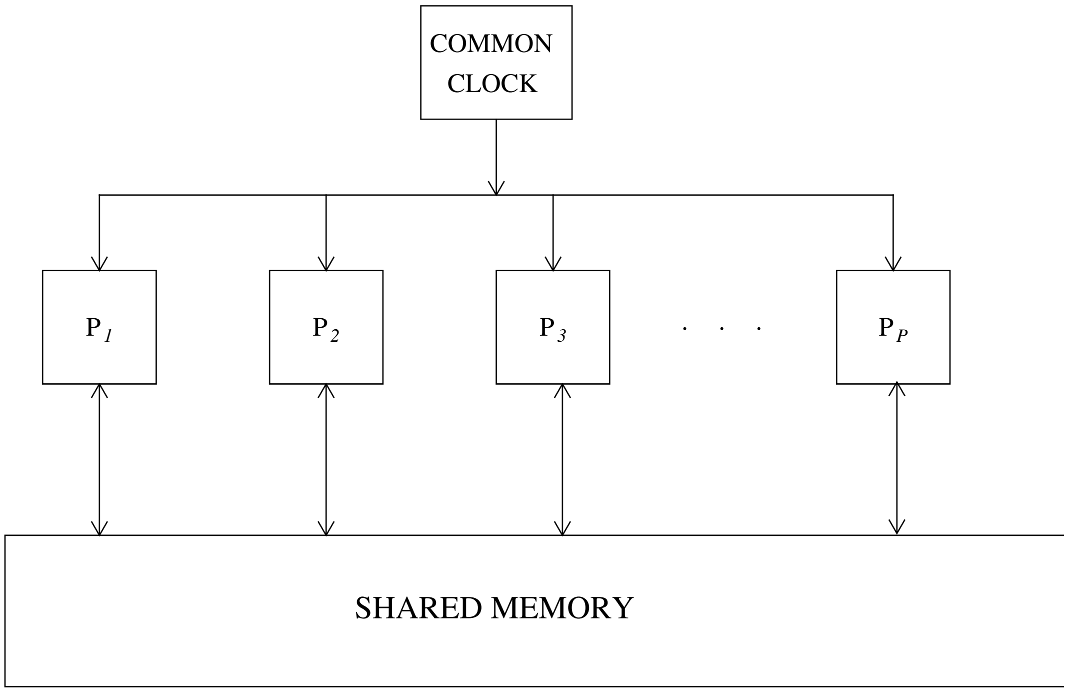



Figure 2.1 The structure of PRAM

Execution of any one instruction takes one clock cycle, i.e. a constant uniform time. One clock cycle is divided into four sequential phases which are synchronous between processors. Different instructions are executed during different phases. In the first phase the program counters (PCs) of all the processors that are not yet halted are increased by one. In the second phase the processors executing instructions not involving shared memory access execute their instructions. In the third phase the processors executing shared memory read instructions execute their read instructions. And finally in the fourth phase the processors executing shared memory write instructions execute their write instructions. Because a clock cycle is divided into phases, reading from and writing to a same shared memory cell simultaneously is not a problem and does not cause any needs to define new models. Read instructions are just executed prior to write instructions.

## 2.1 Concurrent Write Variants

Concurrent write models need further consideration. It has to be defined what happens if two or more processors try to write simultaneously into a same shared memory cell, i.e. it has to be defined what is written to the cell. There are many different *write conflict resolution rules*. Some of the most common ones are presented and defined here. The different write conflict resolution rules define different *variants of the CW PRAM* models.

*Variant* : WEAK

*Definition* : Writing the value zero simultaneously to the same shared memory cell by two or more processors is allowed and the value zero gets stored into the cell. Writing any other value simultaneously to the same shared memory cell by two or more processors is forbidden and the execution of the PRAM ends in a write conflict if such a write is attempted.

*Variant* : COMMON

*Definition* : Writing the same value simultaneously to the same shared memory cell by two or more processors is allowed and the common value gets stored into the cell. Writing two or more different values simultaneously to the same shared memory cell by two or more processors is forbidden and the execution of the PRAM ends in a write conflict if such a write is attempted.

*Variant* : TOLERANT

*Definition* : If two or more processors try to write simultaneously to the same shared memory cell, then the value of the shared memory cell is not changed. The value of a shared memory cell is changed only if just one processor is writing to it at the time. (This variant differs from the EW model so that execution of an EW PRAM would end in a write conflict while a TOLERANT CW PRAM would just ignore the simultaneous write attempt.)

*Variant* : COLLISION

*Definition* : If two or more processors try to write simultaneously to the same shared memory cell, then a special *collision symbol* is written to the shared memory cell. The value of a shared memory cell is changed normally only if just one processor is writing to it at the time.

*Variant* : COLLISION+

*Definition* : If two or more processors try to write simultaneously two or more different values to the same shared memory cell, then a special collision symbol is written to the shared memory cell. The value of a shared memory cell is changed normally if processors are writing the same value to it at the time.

*Variant* : ARBITRARY

*Definition* : If two or more processors try to write simultaneously to the same shared memory cell, then an arbitrarily chosen processor writing to the cell succeeds in writing. There is no way to determine which processor will succeed or succeeded, unless the written value does not uncover the processor that wrote.

*Variant* : PRIORITY

*Definition* : If two or more processors try to write simultaneously to the same shared memory cell, then the processor with the smallest processor identifier , i.e. with the smallest value of the SIG-register , succeeds in writing. In other words, processor identifiers define on unequivocal and processor -wise order of priority of processors so that the processor with the smallest processor identifier has the highest priority .

### 2.2 Computation of PRAM

At the beginning of the *computation of a PRAM* , *the input* is assumed to be somewhere in the shared memory . The contents of all the other shared memory cells is undefined. The signature register (SIG) of each processor is assumed to contain a proper processor identifier , and the program counter (PC) of each processor is assumed to point to the first instruction of the program, i.e. each PC has a value zero. All general registers are assumed to contain a value zero.

The computation starts when all the processors simultaneously and synchronously start to execute the program, beginning from the first instruction (because all PCs have a value zero). Each processor continues the computation, instruction per clock cycle, until it executes a `HALT` instruction, or until it executes the last instruction of the program (if the last instruction of the program is a correct jump instruction, then the execution is not halted), or an error occurs during the computation of the PRAM. From the previous follows that a processor is *correctly halted* when the PC of the processor has a value $C$, where $C$ is the number of instructions in the program.

The computation of a PRAM *ends successfully* when all the processors are correctly halted. The computation of a PRAM *ends to an error* if something forbidden or undefined happens during the computation. This includes, for example, that the PC of a processor would get (as the result of a jump instruction) a value smaller than zero or greater than or equal to the number of instructions in the program, or an instruction gets an invalid argument, or an forbidden or undefined read or write conflict happens. After successful computation the output is in the shared memory (if written there).

### 2.3 Instruction Set

*The instruction set of the PRAM model* includes simple arithmetic operations, bitwise operations, memory access (read from and write to) instructions, register-to-accumulator copy and accumulator-to-register copy instructions, conditional and unconditional jumps, load program counter and load signature register instructions, and a halt instruction. Operation is usually done with the accumulator and the result is stored in the accumulator. Dataflow between a processor and the shared memory is done via the accumulator.

There are five different *addressing modes*: immediate, direct register, indirect register, direct memory, and indirect memory. *Immediate* addressing mode means that the argument is a constant. *Direct register* addressing mode means that the argument is a register indexed with a constant. *Indirect register* addressing mode means that the argument is a register indexed with the value of a register. *Direct memory* addressing mode means that the argument is a shared memory cell indexed with a constant. *Indirect memory* addressing mode means that the argument is a shared memory cell indexed with the value of a register. Different instructions can use different addressing modes; none of the instructions can use every addressing mode.

In the following all the instructions, the possible addressing modes of each instruction, and the semantics of each instruction and addressing mode are presented. Note that the instructions are instructions of a single RAM. Each RAM of a PRAM executes the instructions of the program without a (direct) knowledge of the functioning of the other RAMs. Actually the ability of a RAM to identify itself, i.e. the ability to check the value of the processor identifier, is the only difference between an ordinary RAM and a RAM of a PRAM. (A RAM knows directly that there exists at least $s - 1$ other RAMs in the PRAM, where $s$ is the processor identifier of the RAM.)

First something about the notation:

| | |
|---|---|
| $i$ | is a signed integer |
| $k$ | is a register index, i.e. an integer in range $1, \ldots, R$ |
| $n$ | is a shared memory index, i.e. an unsigned integer $0, 1, 2, \ldots$ |
| $l$ | is a label string |
| $\& l$ | is the index of the instruction labelled with $l$ |
| $C$ | is the number of instructions in the program |
| $r_x$ | refers to the register $x$ |
| SIG | refers to the signature register |
| PC | refers to the program counter |
| $m_x$ | refers to the shared memory cell $x$ |
| log | is the base two logarithmic function |
| and | is the bitwise and operation |
| or | is the bitwise or operation |
| xor | is the bitwise exclusive or operation |
| not | is the bitwise not operation |
| $x \leftarrow y$ | the value of the expression $y$ is evaluated and then set as the value of object $x$ |

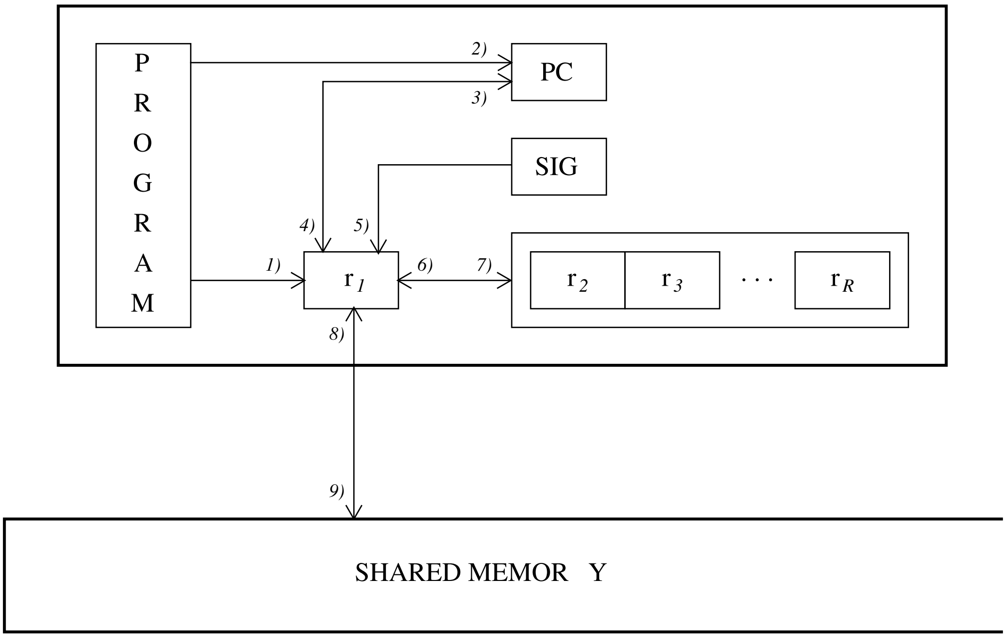
And then the descriptions of the instructions:

| *Instruction* | *Meaning* | *Instruction* | *Meaning* |
|---|---|---|---|
| ADD =$i$ | $r_1 \leftarrow r_1 + i$ | LOADPC | $r_1 \leftarrow \mathrm{PC}$ |
| ADD $k$ | $r_1 \leftarrow r_1 + r_k$ | | |
| ADD *$k$ | $r_1 \leftarrow r_1 + r_{r_k}$ | READ $n$ | $r_1 \leftarrow m_n$ |
| | | READ *$k$ | $r_1 \leftarrow m_{r_k}$ |
| SUB =$i$ | $r_1 \leftarrow r_1 - i$ | | |
| SUB $k$ | $r_1 \leftarrow r_1 - r_k$ | WRITE $n$ | $m_n \leftarrow r_1$ |
| SUB *$k$ | $r_1 \leftarrow r_1 - r_{r_k}$ | WRITE *$k$ | $m_{r_k} \leftarrow r_1$ |
| MUL =$i$ | $r_1 \leftarrow r_1 \cdot i$ | AND =$i$ | $r_1 \leftarrow r_1 \text{ and } i$ |
| MUL $k$ | $r_1 \leftarrow r_1 \cdot r_k$ | AND $k$ | $r_1 \leftarrow r_1 \text{ and } r_k$ |
| MUL *$k$ | $r_1 \leftarrow r_1 \cdot r_{r_k}$ | AND *$k$ | $r_1 \leftarrow r_1 \text{ and } r_{r_k}$ |
| DIV =$i$ | $r_1 \leftarrow \lfloor r_1 / i \rfloor$ | OR =$i$ | $r_1 \leftarrow r_1 \text{ or } i$ |
| DIV $k$ | $r_1 \leftarrow \lfloor r_1 / r_k \rfloor$ | OR $k$ | $r_1 \leftarrow r_1 \text{ or } r_k$ |
| DIV *$k$ | $r_1 \leftarrow \lfloor r_1 / r_{r_k} \rfloor$ | OR *$k$ | $r_1 \leftarrow r_1 \text{ or } r_{r_k}$ |
| MOD =$i$ | $r_1 \leftarrow r_1 - i \cdot \lfloor r_1 / i \rfloor$ | XOR =$i$ | $r_1 \leftarrow r_1 \text{ xor } i$ |
| MOD $k$ | $r_1 \leftarrow r_1 - r_k \cdot \lfloor r_1 / r_k \rfloor$ | XOR $k$ | $r_1 \leftarrow r_1 \text{ xor } r_k$ |
| MOD *$k$ | $r_1 \leftarrow r_1 - r_{r_k} \cdot \lfloor r_1 / r_{r_k} \rfloor$ | XOR *$k$ | $r_1 \leftarrow r_1 \text{ xor } r_{r_k}$ |
| LOG =$i$ | $r_1 \leftarrow \lfloor \log i \rfloor$ | NOT =$i$ | $r_1 \leftarrow \text{not } i$ |
| LOG $k$ | $r_1 \leftarrow \lfloor \log r_k \rfloor$ | NOT $k$ | $r_1 \leftarrow \text{not } r_k$ |
| LOG *$k$ | $r_1 \leftarrow \lfloor \log r_{r_k} \rfloor$ | NOT *$k$ | $r_1 \leftarrow \text{not } r_{r_k}$ |
| SHIFT =$i$ | $r_1 \leftarrow r_1 \cdot 2^i$ | JUMP $l$ | $\mathrm{PC} \leftarrow \& l$ |
| SHIFT $k$ | $r_1 \leftarrow r_1 \cdot 2^{r_k}$ | JUMP $k$ | $\mathrm{PC} \leftarrow r_k$ |
| SHIFT *$k$ | $r_1 \leftarrow r_1 \cdot 2^{r_{r_k}}$ | JUMP *$k$ | $\mathrm{PC} \leftarrow r_{r_k}$ |
| LOAD =$i$ | $r_1 \leftarrow i$ | JPOS $l$ | $r_1 > 0 \Rightarrow \mathrm{PC} \leftarrow \& l$ |
| LOAD $k$ | $r_1 \leftarrow r_k$ | JPOS $k$ | $r_1 > 0 \Rightarrow \mathrm{PC} \leftarrow r_k$ |
| LOAD *$k$ | $r_1 \leftarrow r_{r_k}$ | JPOS *$k$ | $r_1 > 0 \Rightarrow \mathrm{PC} \leftarrow r_{r_k}$ |
| STORE $k$ | $r_k \leftarrow r_1$ | JZERO $l$ | $r_1 = 0 \Rightarrow \mathrm{PC} \leftarrow \& l$ |
| STORE *$k$ | $r_{r_k} \leftarrow r_1$ | JZERO $k$ | $r_1 = 0 \Rightarrow \mathrm{PC} \leftarrow r_k$ |
| | | JZERO *$k$ | $r_1 = 0 \Rightarrow \mathrm{PC} \leftarrow r_{r_k}$ |
| LOADINDEX | $r_1 \leftarrow \mathrm{SIG}$ | | |
| | | HALT | $\mathrm{PC} \leftarrow C$ (i.e. halt) |

The program of a PRAM is described with a list of instructions. Instruction can be labelled with a label string followed by a colon :. For example:

```
Loop: ADD 1
Wait: WaitLoop: MUL *2
```

Execution proceeds in sequential order unless jump instructions don't change the flow of control.




*1)* Literal argument for `ADD`, `SUB`, `MUL`, `DIV`, `MOD`, `LOG`, `SHIFT`, `AND`, `OR`, `XOR`, `NOT`, and `LOAD`. Result is written to the accumulator.

*2)* Literal argument for `JUMP`, `JPOS`, and `JZERO`, i.e. a label. If jump is performed, then the argument is written to PC.

*3)* Direct register or indirect register argument for `JUMP`, `JPOS`, and `JZERO`. If jump is performed, then the argument is written to PC.

*4)* `LOADPC` loads the value of PC to the accumulator.

*5)* `LOADINDEX` loads the value of SIG to the accumulator.

*6)* Direct register or indirect register argument for `ADD`, `SUB`, `MUL`, `DIV`, `MOD`, `LOG`, `SHIFT`, `AND`, `OR`, `XOR`, `NOT`, and `LOAD`. Result is written to the accumulator.

*7)* `STORE` copies the value of the accumulator to a register.

*8)* `READ` reads the value of a shared memory cell to the accumulator.

*9)* `WRITE` writes the value of the accumulator to a shared memory cell.

Figure 2.2 The structure of one RAM of a PRAM and the flow of data

### 2.1 Notes About the PRAM Model

Some may wonder and criticize the infinite shared memory and infinite word length. These are defined infinite just to simplify the definitions of the PRAM model. In every finite computation only a finite number of shared memory cells can be accessed and only a finite number of bits can be used. So, word length and shared memory size is actually bounded for every "incarnation" of the model, i.e. we could parameterize these properties or maybe choose "large enough" constants. But because the size of the shared memory and the word length are considered to be always sufficiently large or as never ending resource, “large enough” constants won’t work. Parameterizing causes extra work, because the parameters must always be determined. The number of general registers of processors is parameterized, because it is expressly required that processors don’t have local, always sufficiently large memories, but just a few necessary registers.

The indexing of processors, registers, shared memory cells, and the instructions of a program may seem confusingly non-uniform. Processors and registers are indexed starting from one, and shared memory cells and instructions are indexed starting from zero. This inconsistency was noticed little too late in the project. Too much had already been implemented in this and in the related parallel modula-2 project (Juvaste 1992a, Juvaste 1992b). But this is just a little beatyspot.

## 3. ABOUT EMULATION OF THE PRAM MODEL

The emulation of the PRAM model in an ordinary one-processor computer, i.e. *a sequential simulation* of the PRAM model, may seem quite simple. The actual simulation of the computation really is quite simple. But when a theoretical model of computation is to be implemented as a software emulator, certain practical issues has to be considered.

Theoretical model usually exists only on paper or in person's mind. It is easy just to state or suppose that there exists, for example, a PRIORITY CRCW PRAM with $P$ processors, each processor has $R$ registers, each processor has the common program, and the input is in the shared memory. For a software emulator these things must be explicitly expressed in some convenient way.

Usually the number of processors is a function of the input. Therefore it is be advisable that the number of processors could be calculated from the input in the emulator, especially if the program should work with different sized inputs for the same problem. This means that the input must be written to the shared memory before the number of processors can be determined. The same emulator should be able to emulate different sized PRAMs and PRAMs with different read and write models, etc. After the computation there must be a way to examine the output that is in the shared memory. Briefly: the input must be written or generated to the shared memory before the size of the PRAM is determined and before the computation starts, it must be expressed to the emulator what kind of PRAM it should emulate, and after the computation it must be possible to examine the output in the shared memory.

The above practical details are not part of the PRAM model and they should be kept separated from the model. The separation of the model and the practical details not including to the model should be easy and obvious for the user of the emulator.

The implementation of the emulator and especially the interface to the outside should also be flexible so that, for example, a high level language compiler is possible to implement. In practice this means a quite low-level interface. A low-level interface has also the advantage that it is simple to implement.

The infinite word length and infinite shared memory size are properties that are in practice impossible to implement. The memory address space of the implementing machine can be easily set as the only restriction on the memory size of the PRAM. Nearly arbitrary long integers are possible to implement, but if at least a moderate efficiency is desired, then the word length of the computer where the emulator is running sets the limits on the word length of the PRAM.

## 4. OVER VIEW OF THE PRAM EMULATOR SYSTEM

The PRAM emulator is controlled via a programming language called *the PRAM emulator language* or *PEL* for short. PEL is an assembly style language and the notation of the instructions is like in the PRAM model. The PRAM model instruction set forms the major part of PEL. The extensions are mainly for the necessary practical details, like initialization of a PRAM and input and output. There are also features that make the monitoring of a PRAM possible. In PEL, the actual algorithm written for the PRAM, the pre-computational initialization of the PRAM, and the post-computational examination of the results are separated in distinct code blocks.

A PEL program can be written in any text editor. It would be possible to interpret a text form PEL program directly, but because the interpretation of the PEL program should be efficient, the PEL program must be converted to a pre-processed and easily interpretable form. Especially the evaluation of jump addresses in advance is almost necessary. *A PEL assembler program* is provided for doing the conversion. The assembler takes a text form PEL program file as input and outputs a compact binary form file.

*The PRAM emulator program* takes an assembled PEL program file as input and executes it, i.e. follows the instructions in the file. The emulator does not produce any output files but has, however, input and output facilities. Input is read from the *standard input stream* (usually the keyboard) and output is written to the *standard output stream* (usually the display). Input and output must be done before or after the execution of the actual PRAM program.

A set of *dumping facilities* has been provided, so that *a monitoring of the execution of a PRAM* would be possible. It is possible to dump the contents of the shared memory, the value of the common clock, and the largest referred to shared memory address to the standard output stream. In addition, an interactive monitor program has been provided for flexible and covering monitoring of a PRAM. All these monitoring aids are controlled with *directives* that can be written among the instructions of a program.

## 5. THE PRAM EMULATOR LANGUAGE PEL

PEL is an assembly style case-sensitive language for controlling the PRAM emulator. A PEL program consists of *an initialization control code block*, *a PRAM code block*, and an optional *termination control code block*. The main task to be done in the initialization control code block is to read in or generate the input for the PRAM and to create and initialize the processors of the PRAM. The PRAM code block contains the actual PRAM algorithm to be executed and emulated. The termination control code block is meant for examining the results of the emulated PRAM algorithm. These code blocks are executed in sequence: first the initialization control code block, then the PRAM code block, and finally the possible termination control code block.

### 5.1 The Control Processor

It is clear that the PRAM code is executed by the processors of the PRAM, and that the PRAM code is represented with the PRAM model instructions. But what about the control code blocks? What notation should one use, and what executes the code? In principle, any notation and an arbitrarily strong processor could have been chosen, because the control code is separate from the PRAM code, and thus from the PRAM model. But a natural and easy solution (at least from the implementor's point of view) is to think of a RAM like *control processor* which has an instruction set almost like the processors of the PRAM.

The control processor needs some extra properties, because the processors of the PRAM don't have any input/output facilities, nor can they "create and initialize" the processors of a PRAM. But a RAM has access to the shared memory and it can do calculations and conditional jumps. One has to remember that the shared memory of the PRAM must exist a period of the execution of the whole PEL program, and that access to the shared memory must be possible within the control code blocks. What about the input and output and creation and initialization of the processors of a PRAM? An easy solution is to think these functions as some kind of *external services*, which can be called from a control code block.

The control processor is almost like the processors (RAMs) of the PRAM. The only structural difference is that the control processor does not have a signature register. Because there exists only one control processor, there is no need to identify between different control processors, and thus the signature register would be useless. The instruction sets of the control processor and the processors of the PRAM are almost alike. The most important difference is that the control processor has an additional instruction, `ECALL`, which is used to call the external services, such as input and output functions and the PRAM initialization function. Because the control processor does not have a signature register, the instruction set of the control processor does not include the `LOADINDEX` instruction. By default, the control processor has 16 general registers.

This default value can be changed with a command line option of the emulator .

### 5.2 Execution of a Program

Control is transferred between the control processor and the processors of a PRAM in the following way . First the control processor starts to execute the initialization control code. When the execution of the initialization control code terminates successfully , control is transferred to the (hopefully) newly created and initialized processors of the PRAM. The PRAM code is executed by the processors of the PRAM. When the execution of the PRAM code terminates successfully , the processors of the PRAM are deleted and control is transferred back to the control processor , which executes the termination control code, if present. Note that the control processor is always initialized when is starts to execute a new control code block. So, the contents of the registers of the control processor are not preserved between the control code blocks. Note however , that the shared memory exists a period of the execution of the whole PEL program. The shared memory is not initialized between the code blocks.

The execution of a control code block terminates successfully when the control processor executes the `HALT` instruction or the control code block ends before any errors occur . If the execution terminates in an error , i.e. an error occurs, the execution of the PEL program terminates in an error . *The PRAM initialization function* must be called in the initialization control code. This function creates and initializes the processors of the PRAM. If the processors of the PRAM are not successfully created and initialized, then an error occurs when control is transferred to the processors of the PRAM.

### 5.3 Preprocessing

As the first phase of the assembling of the PEL program, the source file is first passed through *the C language preprocessor* . Every feature of the C preprocessor may be used in PEL programs. Especially macros and file inclusion are very useful features of the C preprocessor . For example, the instruction set of the PRAM does not include a `NOP` instruction, but one might be useful. The instruction `LOAD 1`, which loads the value of the accumulator to itself, does nothing useful, and can thus be used as a `NOP` instruction. But writing `LOAD 1` is inconvenient and misleading. By writing a macro

```
#define NOP LOAD 1
```

at the start of the program, `NOP` can be written instead of `LOAD 1`.

Comments are written in C style:

```
/* This is a comment */
```

Nested comments are not allowed.

Full descriptions of C preprocessors can be found in C language, C preprocessor, and C compiler manuals, but the following abridged and edited version of the cpp manual pages of the UNIX system (*SunOs Release 4.1 Manual Pages on cpp*) should be sufficient for the user:

Directives

All cpp directives start with a hash symbol (`#`) as the first character on a line. White space (SPACE or TAB characters) can appear after the initial `#` for proper indentation.

`#define` *name token-string*

Replace subsequent instances of *name* with *token-string*.

`#define` *name* `(`*argument* `[,` *argument* `]`... `)` *token-string*

There can be no space between *name* and the '`(`'. Replace subsequent instances of *name*, followed by a parenthesized list of arguments, with *token-string*, where each occurrence of an *argument* in the *token- string* is replaced by the corresponding token in the comma-separated list. When a macro with arguments is expanded, the arguments are placed into the expanded *token-string* unchanged. After the entire *token-string* has been expanded, cpp re-starts its scan for names to expand at the beginning of the newly created token- string.

`#undef` *name*

Remove any definition for the symbol *name*. No additional tokens are permitted on the directive line after *name*.

`#include "`*filename*`"`

Read in the contents of *filename* at this location. This data is processed by cpp as if it were part of the current file. No additional tokens are permitted on the directive line after the final '`"`'.

`#if` *constant-expression*

Subsequent lines up to the matching `#else`, `#elif`, or `#endif` directive, appear in the output only if *constant-expression* yields a nonzero value. All binary non-assignment C operators, including '`&&`', '`||`', and '`,`', are legal in *constant-expression*. The '`?:`' operator, and the unary '`-`', '`!`', and '`~`' operators, are also legal in *constant-expression*.

The precedence of these operators is the same as that for C. In addition, the unary operator defined, can be used in *constant-expression* in these two forms: '`defined (`*name*`)`' or '`defined` *name*'. This allows the effect of `#ifdef` and `#ifndef` directives (described below) in the `#if` directive. Only these operators, integer constants, and names that are known by cpp should be used within *constant-expression*. In particular, the size of operator is not available.

`#ifdef` *name*

Subsequent lines up to the matching `#else`, `#elif`, or `#endif` appear in the output only if *name* has been defined with a `#define` directive, and in the absence of an intervening `#undef` directive. No additional tokens are permitted on the directive line after *name*.

`#ifndef` *name*

Subsequent lines up to the matching `#else`, #elif, or `#endif` appear in the output only if *name* has not been defined, or if its definition has been removed with an `#undef` directive. No additional tokens are permitted on the directive line after *name*.

`#elif` *constant-expression*

Any number of `#elif` directives may appear between an `#if`, `#ifdef`, or `#ifndef` directive and a matching `#else` or `#endif` directive. The lines following the `#elif` directive appear in the output only if all of the following conditions hold:

+ The *constant-expression* in the preceding `#if` directive evaluated to zero, the name in the preceding `#ifdef` is not defined, or the name in the preceding `#ifndef` directive was defined.

+ The *constant-expression* in all intervening `#elif` directives evaluated to zero.

+ The current *constant-expression* evaluates to non-zero.

If the *constant-expression* evaluates to non-zero, subsequent `#elif` and `#else` directives are ignored up to the matching `#endif`. Any *constant-*

*expression* allowed in an `#if` directive is allowed in an *#elif* directive.

`#else`

This inverts the sense of the conditional directive otherwise in effect. If the preceding conditional would indicate that lines are to be included, then lines between the `#else` and the matching `#endif` are ignored. If the preceding conditional indicates that lines would be ignored, subsequent lines are included in the output. Conditional directives and corresponding `#else` directives can be nested.

`#endif`

End a section of lines begun by one of the conditional directives `#if`, `#ifdef`, or `#ifndef`. Each such directive must have a matching `#endif`.

Newline Characters

A NEWLINE character terminates a character constant or quoted string. An escaped NEWLINE (that is, a backslash immediately followed by a NEWLINE) may be used in the body of a `#define` statement to continue the definition onto the next line. The escaped NEWLINE is not included in the macro value.

Diagnostics

The error messages produced by cpp are intended to be self- explanatory. The line number and filename where the error occurred are printed along with the diagnostic.

### 5.4 Syntax

After the preprocessing phase a PEL program should follow the syntax described below. (Actually the C preprocessor inserts line control information lines to the source but these are not visible to the user.) Note that cpp directives are completely different thing from the (PEL) directives in the syntax below. In general: directive means a PEL directive if nothing else is mentioned. The nonterminal symbols are written in italics. The vertical bar '|' separates alternative derivations. The 'ε' means an empty derivation.

| | | |
|---|---|---|
| *Pgm* | $\rightarrow$ | *CtrlCode PRAMCode RestOfPgm* |
| *RestOfPgm* | $\rightarrow$ | ε \| *CtrlCode* |
| | | |
| *CtrlCode* | $\rightarrow$ | *CtrlStartDir InstrSeq CtrlEndDir* |
| *CtrlStartDir* | $\rightarrow$ | `@CONTROL` |

*CtrlEndDir* → `@ENDCONTROL`

*PRAMCode* → *InstrSeq*

*InstrSeq* → ε | *SingleInstr* *InstrSeq*
*SingleInstr* → *Dir* | *SimpleInstr* | *InstrWithLabel*

*Dir* → `@CLOCK` | `@CORE` | `@LASTMEMINDEX` | `@MONITOR`

*InstrWithLabel* → *Label RestOfInstrWithLabel*
*RestOfInstrWithLabel* → *SimpleInstr* | *InstrWithDir* | *InstrWithLabel*
*Label* → *LabelId* **:**
*InstrWithDir* → *Dir RestOfInstrWithDir RestOfInstr*
*WithDir* → *SimpleInstr* | *InstrWithDir*
*SimpleInstr* → *ToAccuInstr* | *ToRegInstr* | *NonArgInstr* | *MemAccInstr* | *JumpInstr*
*ToAccuInstr* → *ToAccuCmd* *ToAccuArg*
*ToAccuCmd* → `ADD` | `SUB` | `MUL` | `DIV` | `MOD` | `LOG` | `SHIFT` | `AND` | `OR` | `XOR` | `NOT` | `LOAD` | `ECALL`
*ToAccuArg* → *ImmediateInt* | *DirectReg* | *IndirectReg*
*ToRegInstr* → *ToRegCmd ToRegArg*
*ToRegCmd* → `STORE`
*ToRegArg* → *DirectReg* | *IndirectReg*
*NonArgInstr* → *NonArgCmd*
*NonArgCmd* → `LOADINDEX` | `LOADPC` | `HALT`
*MemAccInstr* → *MemAccCmd MemAccArg*
*MemAccCmd* → `READ` | `WRITE`
*MemAccArg* → *DirectMem* | *IndirectMem*
*JumpInstr* → *JumpCmd JumpArg*
*JumpCmd* → `JUMP` | `JPOS` | `JZERO`
*JumpArg* → *ImmediateLabel* | *DirectReg* | *IndirectReg*
*ImmediateInt* → `=` *Integer*
*Integer* → *Sign UnsignedInt* | *UnsignedInt*
*Sign* → `-`
*DirectReg* → *Natural*
*IndirectReg* → `*` *Natural*
*DirectMem* → *UnsignedInt*
*IndirectMem* → `*` *Natural*
*ImmediateLabel* → *LabelId*

The only new identifiers the user may present are label identifiers ( *LabelId* ). A label identifier consists of a character string starting with a character `a-z`,`A-Z`, followed by zero or more characters `a-z`,`A-Z`,`0-9`. The identifier must differ from the reserved words.

An unsigned integer ( *UnsignedInt* ) is a string of decimal digits `0-9`. A natural number ( *Natu-*

*ral* ) is an unsigned integer greater than zero.

The syntax separates the instructions by their possible addressing modes, not by the control code - PRAM code separation. The control processor and the processors of a PRAM have, however , slightly different instruction sets. Also few other details are not represented by the syntax. Thus some semantic specifications must be presented.

In a control code block, i.e. between *the code block separation directives* `@CONTROL` and `@ENDCONTROL`, no `LOADINDEX` instructions are allowed, because the `LOADINDEX` instruction does not belong to the instruction set of the control processor . In addition, no directives are allowed in a control code block, because directives are meant for the monitoring of the PRAM itself, not the control processor of the emulator .

The `ECALL` instruction is not allowed in a PRAM code block, because the instruction set of the PRAM model does not include the `ECALL` instruction.

Note that all the code blocks are separate. No jumps between code blocks are allowed. Because code blocks are assembled separately , same labels may be used in different code blocks, although this is not a good programming practice.

### 5.5 Syntax – In Other Words

Because the syntax was designed the implementation of the assembler (parser) in mind, the syntax may seem a little complex. A more human oriented, although less precise, formulation of the syntax or the structure of a PEL program is presented in the following.

The structure of a PEL program is

> `@CONTROL`
> *ControlCode*
> `@ENDCONTROL`
> *PRAMCode*

or

> `@CONTROL`
> *ControlCode*
> `@ENDCONTROL`
> *PRAMCode*
> `@CONTROL`
> *ControlCode*

```
@ENDCONTROL
```

*ControlCode* is a sequence of instructions of the control processor, and *PRAMCode* is a sequence of instructions of the PRAM model. Each instruction, both in *ControlCode* and in *PRAMCode*, may be preceded by one or more labels, like

```
Loop: ADD 2
```

or

```
Loop: Phase1: ADD 2
```

*PRAMCode* may contain directives `@CORE`, `@CLOCK`, `@LASTMEMINDEX`, and `@MONITOR` before and after each instruction. Directives may also appear between labels and instructions, like

```
Loop: @MONITOR ADD 2
```

Note that the possible directive between a label and an instruction must be just before the instruction, i.e. no directives are allowed between two labels.

Note that *ControlCode* may not contain any directives. The code block separation directives `@CONTROL` and `@ENDCONTROL` are just code block separators, not directives that have some special function during the execution of the emulator.

### 5.6 Directives

A PRAM has no input or output facilities. To be able to monitor the execution of a PRAM, a set of dumping functions and an interactive monitor program are provided. The invocation of these monitoring aids is controlled with directives.

A (PEL) directive (`@CORE`, `@CLOCK`, `@LASTMEMINDEX`, or `@MONITOR`) is not an instruction that a processor executes or handles. A directive is an instruction for the emulator to interrupt the execution of a PEL program, and to perform the function the directives suggests, and then continue the execution of the PEL program. The functions the different directives perform are described in the following. The interactive monitor program invoked with the `@MONITOR` directive is described in chapter 8.

*Directive* : `@CLOCK`

*Description* : Directive `@CLOCK` writes the current value of the common clock of the PRAM to the standard output stream in format

`CLOCK:` *Value*

and a newline after it. Note that the execution of the control code blocks have no effect on the value of the common clock; the value of the common clock is always zero at the start of the execution of a PRAM code block.

*Directive* : `@CORE`

*Description* : Directive `@CORE` dumps the shared memory up to the largest referred to shared memory address to the standard output stream in format

| | | |
|---|---|---|
| `CORE:` | `Addr` | `Value` |
| | `0` | *Value* |
| | `1` | *Value* |
| | `2` | *Value* |
| | . | . |
| | . | . |
| | . | . |
| | *Last* | *Value* |
| `END-OF-CORE` | | |

and writes a newline after the dump. Note that the output can be VERY long, so be careful with this directive (even if the output is directed to a file).

*Directive* : `@LASTMEMINDEX`

Description: Directive `@LASTMEMINDEX` writes the largest referred to shared memory address to the standard output stream in format

`LAST MEMORY INDEX:` *Value*

and a newline after it.

*Directive* : `@MONITOR`

*Description* : Directive `@MONITOR` invokes the interactive monitor program. (See chapter 8.)

It is reasonable to think that in a PEL program control flows from instruction to the next instruction (unless a jump is performed) starting from the first instruction – and fortunately this is the way people usually think. When the user wants the execution of a PEL program to be interrupted and some monitoring to be done between two instructions, he or she intuitively puts a proper directive between those instructions. This is approximately how directives are designed to work. When control flows through a directive, the directive triggers an interrupt and the specified function is performed. It is sufficient that just one processor "goes through" a directive so that the directive would be triggered. If many processors "go through" the same directive – the directives may be at different locations – simultaneously , the directive is processed just ones. Note that the processors of a PRAM may branch to different instructions and thus many different directives may be triggered simultaneously . If many different directives are triggered in the same clock cycle, they are processed in the following order: @MONITOR, @CLOCK, @LASTMEMINDEX, and @CORE.

If a directive is placed between a label and an instruction, e.g.

```
...
Wait: @CLOCK LOADINDEX
...
```

the directive is processed every time the instruction is executed, especially if a processor jumps to the labelled instruction. If a directive is placed before the label, e.g.

```
...
STORE 2
@CLOCK Wait: LOADINDEX
...
```

a jump to the instruction does not trigger the directive. (In the example above, if control flows from the STORE instruction to the LOADINDEX instruction, the directive is triggered.)

The situation when a directive is immediately after a jump instruction, e.g.

```
...
JPOS Loop
@CORE MUL =-1
...
```

causes the directive to be processed when control flows through the jump instruction, whether the jump is performed or not and whether the jump is directed forward or backward. This may seem inconsistent, because the directive is triggered even if control does not really flow through the directive. The reason for this kind of an action lies in the implementation: a more consistent implementation would have been much more complicated than the present.

There is one situation when a directive has no effect: if a directive is before the first instruction of a PRAM code block and if the first instruction is preceded by a label, e.g.

```
...
@ENDCONTROL
```

```
@MONITOR LoopStart: LOAD 2
...
```

The idea behind the above is clear . If a processor jumps to the labelled instruction, control is not flowed through the directive. The reason why the directive is not processed at the beginning of the program is that the emulator has no easy way to determine whether a processor jumped to the instruction or not.

The above intuitive idea of directives is just a kind of an approximation of the way how things really work, but the idea is sufficient in practice. Directives are not instructions but are attached to instructions. A directive is processed either before the execution of the instruction or the instruction cycle or after it. This may cause situations when there are both pre-instruction and post-instruction directives to be processed in the same instruction cycle. This may cause slightly confusing effects if the pre-instruction and post-instruction directives are the same. What happens is that the same directive is processed twice in one instruction cycle. This, however , causes no actual harm, but may seem confusing. The rules how directives are attached to instructions are not presented here, because the user does not really have to worry about them. (The rules are presented in the document: *PRAM Emulator – Programmer's Manual* ).

The directives are processed and have some effect only if they are activated with command line options of the emulator . Each of the four directives can be activated separately . So, directives can be written "in advance" to the program, and the program needs to be assembled only once. By changing the command line options of the emulator , different directives can be activated and deactivated in different runs. There is also a "trace mode" command line option which inserts implicit `@MONITOR` directives between every instruction.

### 5.7 External Functions

External functions are used to perform input and output and to create and initialize the processors of a PRAM. External functions may be called only from control code blocks. The external function call has the syntax

`ECALL` *Arg*

where *Arg* is an immediate, a direct register , or an indirect register reference. The value of the argument is the number of the function to be called. If the function gets more than one parameter , the parameters must be stored in successive shared memory cells so that the first argument is in the lowest address. The address of the first argument must be in the accumulator of the control processor . If the function gets at most one argument the possible argument must be in the accumulator of the control processor . If the function returns a value, the value is returned to the accumulator of the control processor .

There are four external functions, and they are presented next.

*Function* : Initialize PRAM

*Description* : Initialize PRAM -function creates and initializes the processors of a PRAM.

*Function number* : 0

*Parameters* : 1: The number of processors
2: The number of general registers in one processor
3: Read model: 0 = exclusive read (ER)
1 = concurrent read (CR)
4: Write model: 0 = exclusive write (EW)
1 = WEAK concurrent write (WEAK CW)
2 = COMMON concurrent write (COMMON CW)
3 = TOLERANT concurrent write (TOLERANT CW)
4 = COLLISION concurrent write (COLLISION CW)
5 = COLLISION+ concurrent write (COLLISION+ CW)
6 = ARBITRAR Y concurrent write (ARBITRAR Y CW)
7 = PRIORITY concurrent write (PRIORITY CW)

*Return value:* None.

*Function* : Input integer

*Description* : Input integer -function reads an integer from the standard input stream and stores the read value to the accumulator of the control processor .

*Function number* : 1

*Parameters* : None.

*Return value:* The integer that is read from the standard input stream is stored in the accumulator .

*Function* : Output integer

*Description* : Output integer -function writes an integer stored in the accumulator of the control processor to the standard output stream. Note that just the integer is written, no newlines or spaces.

*Function number* : 2

*Parameters* : The integer that is written to the standard output stream is in the accumulator .

*Return value:* None.

*Function* : Output character

*Description* : Output character -function writes a character the character code of which (ASCII code, for example) is stored in the accumulator of the control processor to the standard output stream. Note that just the character is written, no extra newlines or spaces.

*Function number* : 3

*Parameters* : The character code of the character that is written to the standard output stream is in the accumulator .

*Return value:* None.

### 5.8 An Example

Usually PRAM algorithms use the number of processors in the computations. When designing an algorithm on paper it is easy just to think that the processors of the PRAM know the number of processors. But in practice this information must be spread to the processors. Often even general spread algorithms assume that the processors know the number of processors. With the CR models the spreading is easy and can be done in constant time. But with the ER models the task is not trivial. An algorithm that spreads the number of processors in logarithmic time and that works in an EREW PRAM is presented here. The algorithm even exits synchronously , i.e. the processors halt simultaneously , which is a most desirable feature if the algorithm is used in other programs.

The algorithm assumes that the input, which is the number of processors, say $P$, is in the shared memory address 0. After the execution of the algorithm, the number of processors ($P$) should be copied into memory cells 1 to $P$. (Now the processors could read the number of processors into their registers simultaneously .)

The algorithm spreads the input in a tree-like fashion. First the processor number 1 copies $P$ from the shared memory address 0 to the address 1. Next the processor number 2 copies $P$ from the shared memory address 1 to the address 2. Next the processors number 3 and 4 copy $P$ from the shared memory addresses 1 and 2 to the addresses 3 and 4. Next processors 5, 6, 7, and 8 copy $P$ from the shared memory addresses 1, 2, 3, and 4 to the addresses 5, 6, 7, and 8. And so on. Before a processor does the copying it waits for it's turn in a wait loop. Because the processor knows it's own processor number , it knows how long to wait for the number of processors to become available. When the processor has done the copying it goes to another wait loop to wait for the others to do the copying. When the processor knows the number of processors, it can determine how many processors are not yet done the copying. When all the processors have done their parts of the spreading they halt simultaneously .

The complete PEL program is:

```
/**************
Problem:          Spread the number of processors
Model:            EREW
```

```
Initial state:     m[0] = P
Final State:       m[1] = m[2] = ... = m[P] = P
                   Processors exit synchronously
Algorithm (# is the processor number):

     if # - 1 > 0 then
          goto <LoopInit>

     m[1] = m[0]
     j = 1
     goto <Wait>   // When #1 is here # >= 3 is at goto <Loop>

     <LoopInit>
          j = 2

     <Loop>        // <Loop>-loop is as long as <Wait>-loop!
          if j + 1 - # > 0 then
               goto <Copy>
          j = 2 * j
          goto <Loop>

     <Copy>        // Copy takes <Wait>-loop + inc of j
          m[#] = m[ # - j / 2 ]

     <Wait>        // <Wait>-loop is as long as <Loop>-loop!
          if j + 1 - P > 0 then
               goto <End>
          j = 2 * j
          goto <Wait>

     <End>

***************/

/* PRAM does not have NOP but r[1] <- r[1] does nothing useful */
#define NOP LOAD 1

@CONTROL
     /* Write '?' for the prompt */
     LOAD =63  /* ASCII-code of '?' */
     ECALL =3  /* Output character */

     ECALL =1  /* Read the number of processors from user... */
     WRITE 0
     LOAD =3   /* Number of registers: 3... */
     WRITE 1
```

```
     LOAD =0   /* Read model: ER... */
     WRITE 2
     LOAD =0   /* Write model: EW... */
     WRITE 3
     LOAD =0   /* Address of first parameter: 0 */
     ECALL =0  /* Initialize PRAM */

     HALT      /* Halt the control processor, not necessary */
@ENDCONTROL

          LOADINDEX     /* Processor index to register 2 */
          STORE 2
          SUB =1        /* Who is number 1?... */
          JPOS LoopInit

          /* For processor number 1 only */
          READ 0    /* Copy m[0] -> m[1]... */
          WRITE 1
          STORE 2   /* r[2] <- P */
          LOAD =1   /* Loopindex j <- 1 */
          STORE 3   /* r[3] <- j */
          NOP       /* Wait for a while... */
          NOP
          JUMP Wait     /* Wait for the others */

          /* For processors number 2,3,...,P */
LoopInit: LOAD =2   /* Loopindex j <- 2 */

Loop:     STORE 3   /* r[3] <- j */
          ADD =1    /* Whose turn to copy?... */
          SUB 2
          JPOS Copy     /* if (j + 1) - # > 0 then goto Copy */

          /* Not yet my turn. Have to wait... */
          LOAD 3    /* j <- j * 2... */
          SHIFT =1
          JUMP Loop

Copy:     LOAD 3    /* r[1] <- j / 2... */
          SHIFT =-1
          MUL =-1   /* r[1] <- -r[1] = -(j/2) */
          ADD 2     /* r[1] <- r[1] + # = # - j/2 */
          READ *1   /* m[r[1]] = m[#-j/2] -> m[r[2]] = m[#]... */
          WRITE *2
          STORE 2   /* r[2] <- P */
          LOAD 3    /* r[1] <- j */
```

```
          NOP       /* Wait for a while... */
          NOP

Wait:     STORE 3   /* Equal to Loop-loop except r[2] does not */
          ADD =1    /* contain # but P! */
          SUB 2
          JPOS End
          LOAD 3
          SHIFT =1
          JUMP Wait

End:      HALT
```

It might be useful to simulate the execution of this program with a pencil and a piece of paper and some small (!) number of processors to get some idea of the PRAM and this algorithm.

### 5.9 Some Notes on the Implementation

The functions of the instructions of the PRAM model were described in chapter 2. Those descriptions are still valid for the PEL instructions in most cases. For some instructions the exactness of the definitions have to be reduced due to restrictions set by the implementation. Also the requirement for the unlimited word length has to be relaxed. Because the emulator was designed and implemented to be portable from machine to machine, these inaccuracies and limitations are mainly machine-dependent features. In each implementation, the exact behaviour can be defined by experimenting.

The emulator was programmed with ANSI C programming language (Kernighan and Ritchie, 1988, Plauger and Brodie, 1989). The restrictions and inaccuracies come directly or indirectly from the definition of ANSI C.

#### *5.9.1 Word Length*

As mentioned before, the unlimited word length is impossible to implement efficiently . The word length in the emulator is machine-dependent. The word length is at least 32 bits and each integer is stored as signed integer . The minimum range of values one word can hold is from $-2^{31}$ to $2^{31}$ not including the endpoints.

Because the word length, and thus the integer range, is limited, arithmetic overflows and underflows can happen. The assembler checks that the constant values appearing in the program text are in the valid integer range. The emulator checks during run-time that an argument of an instruction is sensible and within the domain of the function. But the emulator does not check if an instruction causes an arithmetic underflow or overflow . The hardware and the operating system on which the emulator is run may or may not trap underflows and overflows.

### *5.9.2* `DIV` *Instruction*

For non-negative operands, the division instruction `DIV` truncates the quotient toward zero. Otherwise the direction of the truncation is machine-dependent. The division instruction checks that the divisor is not zero.

### *5.9.3* `MOD` *Instruction*

If both operands of the modulus instruction `MOD` are non-negative, then the remainder is non-negative and smaller than the divisor; if not, it is guaranteed only that the absolute value of the reminder is smaller than the absolute value of the divisor , i.e. the sign is machine-dependent. It is always true that $(a/b) \cdot b + a\% b$ is equal to $a$, where $\%$ is the modulus operator . The modulus instructions checks that the divisor is not zero.

### *5.9.4* `SHIFT` *Instruction*

The result of the shift instruction `SHIFT` is undefined, if the absolute value of the argument (the number of bit positions to be shifted) is greater than or equal to the word length (number of bits in a word). If the argument of the shift instruction is negative and the value to be shifted (the value in accumulator) is non-negative, then zeros fill the vacated bit positions, i.e. an arithmetic shift is performed. If the value to be shifted is negative, then the result of a shift with a negative argument is undefined.

### *5.9.5 Collision Symbol*

The COLLISION and COLLISION+ CR PRAM models write a collision symbol if a write conflict occurs. The collision symbol was not defined in the definition of the PRAM model. The collision symbol the emulator uses is machine-dependent. It is the smallest integer value and is thus dependent on the word length and on the representation of negative integers. The collision symbol can be easily determined with a short PEL program. The following program writes the collision symbol to the standard output stream.

```
@CONTROL
     LOAD =2   /* Two processors... */
     WRITE 0
     LOAD =2   /* Two registers... */
```

```
     WRITE 1
     LOAD =1   /* CR... */
     WRITE 2
     LOAD =4   /* COLLISION CW... */
     WRITE 3
     LOAD =0   /* Address of first parameter */
     ECALL =0 /* Initialize PRAM */
     HALT
@ENDCONTROL

WRITE 0   /* Write simult. to the address 0 => collision symbol */
HALT

@CONTROL
     READ 0    /* Read the collision symbol */
     ECALL =2  /* Output the collision symbol */
     LOAD =10  /* Output a newline... */ ECALL =3
     HALT
@ENDCONTROL
```

# 6. PEL ASSEMBLER

After a PEL program has been written with some text editor and before it can be executed with the emulator it has to be assembled with the PEL assembler provided.

## 6.1 Usage

Usage of the assembler is very simple. The command line (if the assembler is installed with name pa) is

`pa` *Source* *Object*

where *Source* is the name of the PEL program file to be assembled and *Object* is the name of the assembled file. There is no command line options. There is no restrictions on the names of the files or default filenames or default specifiers. An error message

```
Usage: pa <source> <object>
```

is given if an incorrect number of arguments is given to the `pa` command.

The command first calls the C preprocessor and if the source file is successfully preprocessed the preprocessed file is passed to the actual assembler . If the assembling phase is successful, a file named *Object* is created. The created file can be executed in the emulator . The creation of the object file is started when the assembling phase begins and thus a possible existing file named *Object* is deleted even if an error occurs during the assembling phase.

The error messages of the C preprocessor are documented in the manual of the used C preprocessor . The error messages of the assembler are documented in the following section.

## 6.2 PEL Assembler Error Messages

The assembler , among other things, checks that the PEL program is syntactically valid. It also checks certain semantic details. The assembler gives an error message, if it finds an error . The assembler does not stop to the first error but tries to continue the parsing of the program. The output program file is created only if no errors occur .

Certain (assembler) run-time errors can happen during the execution of the assembler . These errors happen due to the limitations or misuse of the underlying hardware and operating system. These errors are described in chapter 9.

The format of an assembler error message is

*File* **:** *Line* **:** *Message*

*File* is the name of the file where the error occurred. Note that the file can be an included file. *Line* is the line number of the file *File* where the error was detected. The line number is relative to the beginning of the file, i.e. the error can be found by finding the line number *Line* from file *File*. *Message* is the actual message. The messages are described below.

*Message* : Unexpected end of file

*Description* : This messages is given when the source file ends before a syntactically complete program or program structure is found.

*Message* : *Token1* expected in place of *Token2*

*Description* : This message implies that due to previous tokens the assembler started to parse some syntactic structure that does not allow *Token2* but expects *Token1* in place of it.

*Message* : Not allowed in this context: *Token*

*Description* : *Token* is in principle correct but it is not allowed in this context. This message is given when a directive or instruction appears in a code block where it is not allowed, or when a directive appears between two labels. In the latter case the latter label is reported to be the token not in proper context. This is because the assembler expects to find a directive or an instruction.

*Message* : No argument for *Instruction* can begin with *Token*

*Description* : A valid argument for *Instruction* cannot begin with *Token*.

*Message* : Integer literal overflow: *Integer*

*Description* : Integer literal *Integer* found in program text represents an integer that is out of the range of possible integers.

*Message* : Undefined label: *Label*

*Description* : A reference to a label *Label* is found, i.e. *Label* is an argument of a jump instruction, but the definition of the label is not found in the code block, i.e. there is no instruction preceded by *Label* (and a **:**).

*Message* : Label redefinition: *Label*

*Description* : Label *Label* is defined twice in the same block, i.e. there is two instructions preceded by the same label *Label* (and a **:**).

*Message* : Invalid constant argument *Constant* for *Instruction*

*Description* : A constant argument (i.e. an immediate reference) *Constant* of an instruction *Instruction* is not valid. This message is given if a `DIV` or a `MOD` instruction gets a constant zero argument, a `LOG` instruction gets a constant argument less than or equal to zero, or an `ECALL` instruction gets a constant argument less than zero.

*Message* : Invalid register index '0' in argument of *Instruction*

*Description* : A reference to a register number zero is found in the argument of instruction *Instruction*. (There isn't such a register!)

# 7. EMULA TOR

When a PEL program has been assembled with the PEL assembler , the created assembled program file can be executed with the emulator . There is not much to say about the controlling of the emulator; most of the control is described with the PEL programs. There are some command line options and error conditions and relating error messages.

## 7.1 Command Line

The format of the command to start the emulator is

`pram [-aclmtM] [-r` *Regs* `]` *ProgramFile*

*ProgramFile* is the name of the file containing the assembled PEL program to be executed. The activation of the directives and the number of the general registers of the control processor can be controlled with the command line options. If no options are specified, all directives are inactive and the control processor has 16 general registers. The command line options can be specified in a any order and options can be mixed. The `-r` directive must, however , be separate from the others. For example, the commands

```
pram -c -t -r 8 test

pram -r 8 -tc test

pram -t -r 8 -c test
```

have the same effect, but the command

```
pram -tcr 8 test
```

is invalid.

### *7.1.1 Command Line Options*

*Option* : `-a`
*Description* : This option activates all directives.

*Option* : `-c`
*Description* : This option activates the `@CORE` directive.

*Option* : `-l`
*Description* : This option activates the `@LASTMEMINDEX` directive.

*Option* : `-m`

*Description* : This option activates the `@MONITOR` directive.

*Option* : `-t`

*Description* : This option activates the `@CLOCK` directive.

*Option* : `-M`

*Description* : This option adds implicit `@MONITOR` directives after each instruction and before the first instruction of the PRAM code block of the program. The monitor program is thus called every clock cycle and a full trace of the execution of the PRAM code is possible and easy. All explicit directives written to the program are deactivated. If the trace mode is turned off in the emulator program, then the implicitly added `@MONITOR` directives are removed (implicitly) and the explicit directives that were specified active with command line options are set active.

*Option* : `-r` *Regs*

*Description* : This option is used to change the default number of general registers in the control processor. The default number is 16. *Regs* is the new number of registers and it must be at least two. There is no upper bound.

### *7.1.2 Command Line Error Messages*

If the command line is invalid, an error message is given. The possible error messages are described here.

*Message* : pram: option ‘r’ usage: [-r <regs>]

*Description* : Something else than a space follows the `-r` option. The `-r` option must be used separate from the others and there must be at least one space between the `-r` and the number of registers (<regs>).

*Message* : pram: invalid argument for option ‘r’

*Description* : The argument following the `-r` option, i.e. *Regs*, does not form a valid integer representation.

*Message* : pram: invalid number of control processor registers requested

*Description* : The requested number of general registers for the control processor, i.e. the argument *Regs*, is less than the minimum two.

*Message* : pram: option ‘r’ in invalid context, options usage: [-aclmtM] [-r <regs>]

*Description* : The option `-r` is not used separately from the others.

*Message* : pram: missing option

*Description* : There is no option character following immediately the '−'.

*Message* : pram: invalid option '*Opt*'

*Description* : An invalid (unknown) option *Opt* is used.

*Message* : Usage: pram [-aclmtM] [-r <regs>] <programfile>

*Description* : There are too few arguments in the command line.

### 7.2 Emulator Error Messages

There exists a few possible run-time error situations. Most of them are checked and reported if an error occurs. Arithmetic underflows and overflows are NOT checked by the emulator for efficiency reasons. The underlying hardware and operating system may or may not trap these errors. Here the errors concerning the actual emulation process are described. These errors are caused by an errorneous PEL program or input. These errors cause the emulator to terminate.

There exists also other error situations concerning mostly the execution of the emulator program. These errors happen due to the limitations or misuse of the underlying hardware and operating system. These errors are described in chapter 9.

Values of the SIG register and the program counter (PC) of a processor are output after most of the messages. The processor is the one that caused the error , or it is the processor with the smallest SIG in the group of processors causing the error . The latter case applies to read and write conflicts where a group of processor cause the error by reading from or writing to the same shared memory cell in a non-valid manner . If an error is caused by the control processor (in a control code block), the SIG has value 0. Note that the value of the program counter of a processor is increased prior to the execution of an instruction. Because the indexing of instructions of a code block start from zero, it follows that the instruction that caused the error is the $n$th instruction in the code block, where $n$ is the value of the PC in the error message.

*Message* : Division by zero: SIG: *Sig*, PC: *Pc*

*Description* : A processor whose SIG is *Sig* attempted a division by zero. Note that the argument (zero) is a direct or indirect register reference; constant zero arguments are checked in assembly-time.

*Message* : Modulus by zero: SIG: *Sig*, PC: *Pc*

*Description* : A processor whose SIG is *Sig* attempted a modulus by zero. Note that the argument (zero) is a direct or indirect register reference; constant zero arguments are checked in assembly-time.

*Message* : Non-positive argument for logarithm: SIG: *Sig*, PC: *Pc*

*Description* : A processor whose SIG is *Sig* attempted to take a logarithm of a non-positive number. Note that the argument is a direct or indirect register reference; constant arguments are checked in assembly-time.

*Message* : Invalid jump address: SIG: *Sig*, PC: *Pc*, Invalid addr: *Addr*

*Description* : A jump directed outside the code block is attempted. *Addr* is the invalid jump address.

*Message* : Illegal register reference: SIG: *Sig*, PC: *Pc*, register reference: *Ref*

*Description* : A reference to a non-existing register appears as an argument or as part of an argument of an instruction. *Ref* is the invalid register index.

*Message* : Illegal read attempt: addr: *Addr*, first proc: SIG: *Sig*, PC: *Pc*

*Description* : A simultaneous read from the same shared memory address is attempted in an ER model. *Addr* is the shared memory address where the processors try to read from. The processor identified is the processor with the smallest SIG in the group of processors attempting the read from the address *Addr*.

*Message* : Illegal write attempt: addr: *Addr*, first proc: SIG: *Sig*, PC: *Pc*

*Description* : A forbidden simultaneous write to the same shared memory address is attempted. *Addr* is the shared memory address where the processors try to write to. The processor identified is the processor with the smallest SIG in the group of processors attempting the write to the address *Addr*.

*Message* : PRAM not correctly initialized

*Description* : The PRAM initialization function is not called in the initialization control code block.

*Message* : Invalid number of RAMs ( *Num* ) requested

*Description* : A smaller than one-processor PRAM is attempted to create with a call to the PRAM initialization function. *Num* is the requested invalid number of RAMs.

*Message* : Invalid number of registers ( *Num* ) requested

*Description* : A smaller than the minimum number of general registers (which is two) is requested for each PRAM processor in a call to the PRAM initialization function. *Num* is the requested invalid number of general registers.

*Message* : Invalid external function #: PC: *Pc*, function #: *FuncNum*

*Description* : A non-existing external function is called in a control code block. *FuncNum* is the invalid external function number.

## 8. MONITOR

In the emulator, no input or output can be done during the execution of a PRAM code block because the processors of the PRAM have no input or output instructions. In control code blocks, external input and output functions can be called with the `ECALL` instruction. It would be very desirable to be able to follow the execution of a PRAM code, not just to examine the results of the PRAM algorithm in the termination control code. A set of directives that interrupt the execution of a PRAM code and trigger the execution of some monitoring aid has been designed and implemented for this reason. The non-interactive directives `@CLOCK`, `@CORE`, and `@LASTMEMINDEX` were described in chapter 5. In this chapter, the interactive monitor program invoked with the `@MONITOR` directive is described.

### 8.1 Introduction

The non-interactive directives just dump some data to the standard output stream. Especially the `@CORE` directive may produce huge amounts of data, which is hard to handle. None of the non-interactive directives provide a (direct) way to examine the registers of the processors of a PRAM. With the interactive monitor program, the desired registers, shared memory locations, etc. can be quite easily selected and viewed.

The monitor is invoked when an active `@MONITOR` directive is encountered. With a command line option of the emulator, a trace mode can be set on. Then an implicit `@MONITOR` directive is added between all PRAM code instructions and before the first and after the last instruction of the PRAM code block and all explicit directives are set as inactive. The trace mode can be toggled on and off in the monitor program. When trace mode is set off, then the explicit directives that were set as active with command line options are again active.

In the monitor program, the user can define *windows* and attach *views* to the windows. *A window* is a portion of the display where the user defined data is shown at. A window is always as wide as the display but the number of rows can be defined. *A header line* is automatically added to each window. *A view* is a description of what data is to be shown and how (in what format). One view can be attached to each window.

If the monitor program is invoked many times during the execution of the PRAM code of a PEL program, the monitor "remembers" the defined windows and attached views between the invocations. This feature makes it possible to easily follow the execution of a PRAM code, even instruction by instruction.

## 8.2 Windows

Most of the display is reserved for the *user defined windows*. The three bottom lines of the display form *the message window* of the monitor. The error messages and some other messages reported by the monitor are displayed in the message window. The line above the message window is t*he user input window*. The user defined windows, or *windows* for short, are placed to the upper part of the display. The first defined window is placed on the top of the display, and the second just below it, etc. Thus the user needs only to define the number of rows in each window, not the placements of the windows.

Windows are numbered staring from one. The topmost window is always number one, the next is number two, etc. The numbering and the placement of windows is dynamic in that sense that if a window is deleted, then the numbers of the windows below the deleted one are decreased by one and the windows are shifted upwards to fill the gap. The number of a window is displayed at the start of the header line of the window in format '#*Num*', where *Num* is the number of the window. One of the windows is always *the default window*. Certain functions invoked with user commands affect the default window or the default window number is given as the default value to certain input requests. The text '`Default`' is displayed on the header line of the default window just after the window number.

## 8.3 User Interaction

When the monitor is invoked, the screen is cleared and if there exists view and window definitions defined during previous invocations of the monitor, the data according to the view and window definitions is displayed. When the program is ready and waiting for a command from the user, a prompt '>' is displayed in the input window and the cursor is just right to the prompt.

*A command* is usually given by pressing *a command key*. Some commands may be preceded by an *unsigned integer argument*. Then the command is given by first typing the argument and then ending the input by pressing the command key. The typed integer digits are echoed on the input window. (A typed unsigned integer ended with a return key changes the default window). If a proper command is given, a corresponding *command function* is invoked; otherwise nothing happens, except that the message window is cleared, and the program starts to wait for a proper command.

The command function may ask some additional *input*. The command can usually be cancelled by giving *an empty input*. The cancellation is not reported; the program just does nothing. If the input is not valid, an error message is usually written on the message window. If a valid non-empty input is given, the program sometimes reports about the successful performance of the function.

When the function ends, successfully or unsuccessfully, the monitor starts to wait for a new command -- unless, of course, *a quit* or *a halt command* is given. If a quit command is given, then the monitor is quitted and the execution of the PRAM code continues. The screen is not cleared. If a halt command is given, then the execution of the emulator is halted. The screen is not cleared.

When the program is waiting for an input other than just a single key stroke, a simple *line editor* is invoked. The user may type the input characters and delete the last typed character by pressing the backspace key. When the program is waiting for a numeric input, only the numeric keys are active. The space bar is never active, because space characters are not needed nor allowed in inputs. The input is usually ended with the return key. When a command preceded by an unsigned integer argument is given, the typing of the argument is ended with the command key. Usually some *prompt* is written, when input is waited for. If the input might be long, the prompt is deleted when the typing of the input starts. Sometimes a default value is given as the default. The default value may be accepted by pressing the return key or it may be edited by using the backspace key and typing new input characters.

The command functions and the commands are described later in this chapter.

## 8.4 Definition of a View

A view can be defined by invoking *the define view function* which asks the definition. A definition is written using a compact *view definition language*. A definition of a view may at first seem cryptic and as hard to interpret string of characters, but the interpretation of a definition is quite simple. The obscurity of a definition of a view is due to compactness of the language: all identifiers consist of one character and no space characters are allowed. A compact form was chosen to simplify the interpretation of the definitions.

A definition of a view consists of *a list of references* to the objects which are to be viewed, i.e. registers, shared memory locations, etc. A reference can be a simple one referring to a single value, or it can refer to a set of objects.

### *8.4.1 Variable References*

The simplest form of a reference is a single *identifier*. For example, the reference

```
t
```

refers to the common clock of the PRAM. A definition of a view can be just this '`t`'. When the monitor program is interpreting the definition, it fetches the value of the common clock and writes it to the window in form

`t:`*T*

where *T* is the value of the common clock. A viewed object is always displayed in the form

*Name* **:** *Value*

where *Name* is the name of the object. There are certain *variables*, some of which are actually constants during the execution of the emulator, that are referred to in the same single character manner. For example, `W` is the number of rows in the window, `P` is the number of processors in the PRAM, `R` is the number of general registers in each RAM of the PRAM, and `M` is the largest referred to shared memory address.

### *8.4.2 Lists of References and Separators*

*A list of references* has the form

*Ref SepLst Ref SepLst ... SepLst Ref*

where *Ref* is a reference and *SepLst* is *a list of separators*. There are two kinds of separators: *a space separator* '`,`' and *a newline separator* '`;`'. Separators are used to determine the form of the output. When the view interpreter meets a space separator, it writes a space character on the window; when it meets a newline separator, it writes a newline character. There can be a list of separators between references, and a list of separators is taken literally: '`,;,,`' produces first a space, then a newline, and then two spaces.

A more complicated definition of a view that the single '`t`' might be

```
t,,,M;P,R
```

which might produce the output

```
t:102 M:1123 P:2048 R:8
```

Actually the space separators are not always taken literally. The monitor program automatically breaks the output to the next line if the list of references does not fit to one line. The break is always done between references; a reference is never broken on two lines. If this kind of *implicit newline separator* is inserted between references, then the possible space separators following the inserted newline separator up to the next reference or to an explicit newline separator are skipped. This simply means that an implicit newline substitutes explicit space separators; there is no need to write spaces at the start of the lines if a space separated list of references is automatically broken on two or more lines. Note that if the definition of a view contains, for example, a list *Ref1* `,,;,` *Ref2*, then a space is written at the start of a line before *Ref2*, because there is an explicit newline separator before the space separator. But if the definition contained the list *Ref1* `,,,,` *Ref2* and only the *Ref1* would fit to the current line, then an implicit newline separator would be added between *Ref1* and *Ref2* and the space separators would be skipped, i.e. *Ref2* would be written to the start of the next line.

### 8.4.3 Simple References with Index Expressions

One cannot refer to a shared memory location with a single identifier. The index or the address of the shared memory location has to be specified somehow. Referring to a shared memory location is simple. The reference has the form

`m`*Expr*

where *Expr* is *an index expression*. For example, the reference

```
m3
```

refers to the shared memory location three and the output might be

```
m3:7
```

The index expression can be more complicated than an unsigned integer constant. The arithmetic operators '`+`' (addition), '`-`' (subtraction or sign), '`*`' multiplication, '`/`' division, and '`%`' reminder may be used. (The operators work as the corresponding PRAM instructions.) The factors of the expression need not be unsigned integer constants. References to objects (or to the values of the objects) may be used as well! For example, in the reference

```
mM
```

the value of the variable M (the largest referred to shared memory address) is used as index expression. This reference can be used as an index expression:

```
mmM
```

And arithmetic operators may be used:

```
m(M−1)/2
```

Lets assume that the variable `M` has value 101, and the shared memory location 50 has value 8. Then the reference

```
m(M−1)/2
```

would produce the output

```
m50:8
```

because `m(M−1)/2` evaluates to `m(101−1)/2` which evaluates to `m100/2` which evaluates to `m50` which produces the output.

One note about references to the shared memory . The whole shared memory is accessible in monitor program, not just the portion up to the largest referred to shared memory address. This means that if there is a reference beyond the largest referred to shared memory address, the largest referred to shared memory address and the value of variable `M` changes! For example, after the reference

```
m(M+1)
```

the value of the variable `M` has increased by one.

An index expression can always be arbitrarily complex, but the value of the expression must, of course, be sensible. The value of the index expression must be within the range of the index. If the index refers to somewhere beyond the limits, the view interpreter gives an error message. The interpreter evaluates the index expression and uses the value as the index when fetching the data value. Also in the output written on the window , the value of the index expression is viewed, not the whole expression.

Identifier '`i`' is used in *references to an instruction* . It can be used in two ways and the first has the same form as the reference to a shared memory location, i.e.

`i`*Expr*

The second form is discussed later . A reference to an instruction does not produce a numeric value but a human readable representation of the instruction the index of which is the value of the index expression. So, a reference to an instruction may not be used in index expressions. The output of the reference

```
it%42
```

is

```
i8:MUL *2
```

if `t` has value 50 and the instruction number 8 of the PRAM code block of the PEL program is '`MUL *2`'.

What about a *reference to a register* of a processor? In such a reference both the processor and the register must be specified. The form of a reference to a general register is

`p`*Expr* `.r`*Expr*

For example, the reference

```
p1.r2
```

refers to the register number 2 of processor number 1. The index expressions can again be arbitrarily complex, but the values must be sensible. A reference

```
pmp1.r1.rR
```

which is equivalent to

```
p(m(p1.r1)).r(R)
```

produces the output

```
p7.r16:0
```

if the register number 1 of the processor number 1 (`p1.r1`) contains the value 33 and the shared memory location number 33 contains the value 7 and each RAM of the PRAM has 16 general registers and the register number 16 of the processor number 7 contains the value 0.

In addition to the general registers, each processor has a program counter and a signature registers. These registers can be referred to with identifiers '`c`' (program counter) and '`s`' (signature register). The forms of the complete references are of course

```
pExpr.c
```

and

```
pExpr.s
```

The reference

```
p1.c
```

refers to the value of the program counter of the processor number 1 and might produce the output

```
p1.c:234
```

The reference to a signature register of a processor is actually useless: the index of the processor is the same as the value of the signature register . So, the reference

```
p2.s
```

produces the output

```
p2.s:2
```

The current PRAM model does not allow a processor to change the value of the signature register . If somebody someday adds this feature to the emulator , then this possibility to reference to the signature register is useful - and already implemented...

The identifier '`i`' may be used with reference to a processor . A reference of the form

`p`*Expr* `.i`

refers to the instruction pointed by the program counter of the processor the index of which is the value of the index expression *Expr* . The output is in a human readable form, not in numeric form. So, the reference

`p`*Expr* `.i`

refers the same instruction as the reference

`i(p`*Expr* `.c)`

(if the index expressions are the same, of course). The output is little different. If the program counter of the processor number 3 has value 66 and the instruction number 66 is '`SHIFT =-1`', then the reference

```
p3.i
```

would produce the output

```
p3.i66:SHIFT =-1
```

and the reference

```
ip3.c
```

would produce the output

```
i66:SHIFT =-1
```

*8.4.4 Range References*

So far , only references to single values has been presented. Often arises needs to view a range of values, for example, some range of shared memory locations. It is of course possible to list all the locations separately , like

```
m0,m1,m2,m3,m4,m5,m6,m7,m8,m9,m10,m11,m12,m13,m14,m15,m16
```

but this is laborious. Fortunately there is an easier way , *a range reference* , i.e. it is possible to refer to a range of objects with one reference statement. A range reference can be obtained by substituting the index expression of a simple reference with a (index) *range expression* . The range expression has the form

[ *Expr1 SepLst Expr2* ]

where *Expr1* is an integer expression the value of which is the lower limit of the index range, *Expr2* is an integer expression the value of which is the upper limit of the index range, and *SepLst* is a separator list consisting of one or more space and newline separators. The separator list both separates the boundary expressions and controls the output. The separators specified in the separator list are written between each two output values. For example, the above list of 17 shared memory references can be substituted with the range reference

```
m[0,16]
```

The output is exactly the same; the values are written separately , like

```
m0:1 m1:23 m3:21 m4:2 m5:1 ...
```

If the range reference was

```
m[0,,,16]
```

then there would be three spaces between the output values, and if the range reference was

```
m[0;16]
```

then the values would be written on separate lines. So, a range reference

*Id* `[`*Expr1 SepLst Expr2* `]`

is just a shorthand for

*Id Expr1 SepLst Id* `(`*Expr1* `+1)` *SepLst...SepLst Id Expr2*

A range reference results in a list of values, so a range reference cannot be used as an index expression; a range reference always produces output. The boundary expressions can be arbitrarily complex but the values must be reasonable and the value of the lower limit expression must be smaller than or equal to the value of the upper limit expression.

Every index expression of a reference that produces output, i.e. an index expression that is not part of a reference within another index expression, can be substituted with a range expression to obtain a range reference. So, the following forms are valid:

`i`*Range*
`m`*Range*
`p`*Range* `.c`
`p`*Range* `.s`
`p`*Range* `.i`
`p`*Range* `.r`*Expr*
`p`*Expr* `.r`*Range*
`p`*Range* `.r`*Range*

where *Range* is a range expression and *Expr* is an index expression. All the forms should be clear , but the form

`p`*Range* `.r`*Range*

might need some specification. It is clear that the reference produces an output consisting of a range of general registers of a range of processors. The output is processorwise, i.e. all the specified registers of one processor are output before the registers of the next processor . The index of the registers "runs faster" than the index of the processors. The separator list in the register range expression is handled between the registers of one processor . The separator list in the processor range expression is handled when the processor index changes. The reference

```
p[1;P].r[1,R]
```

would output all the general registers of all the processor so that the registers of different processors are on separate lines.

### *8.4.5 List References*

The output of the processor range reference of the form

`p`*Range* `.`*Ref*

where *Ref* is some kind of register reference, either simple or range reference, is processorwise and only one type of registers can be referred to in one reference statement. Sometimes it would be desirable to be able to output, for example, general registers and the current instruction of the processor on the same line. There is *a list reference* in the view definition language to make this possible. A list reference has two forms:

`p`*Expr* `.`*Lst*
`p`*Range* `.`*Lst*

where *Expr* is an index expression, *Range* is a range expression and *Lst* is a list expression. A list expression has the form

`(`*Ref SepLst Ref SepLst...SepLst Ref* `)`

where *Ref* is a reference of the forms

`c`
`s`
`i`
`r`*Expr*
`r`*Range*

and *SepLst* is a separator list. The list reference is a way to refer to the different type registers of the same processor without the need to write the same processor reference many times. For example, the reference

```
pP/2.(r[1,3],r[6,R],,,i)
```

is a shorthand for the list of references

```
pP/2.r[1,3],pP/2.r[6,R],,,pP/2.i
```

but for the reference

```
p[1;P].(r[1,R],,i)
```

which writes all the registers and current instructions of all the processors processorwise, i.e. different processors on different lines, it is usually impossible to write an equivalent form without the list reference.

### *8.4.6 Automatic Pausing of the Output*

With range and list references a lot of objects can be referred to with a short definition of a view. The amount of data may be so large that the data does not fit to the window, especially if the PRAM is large. If the view interpreter, when it is interpreting a view and writing the data on the window, notices that the data does not fit to the window, it interrupts the output and writes the prompt "`more...`" on the header line of the window and waits until the user presses a key. When the user has pressed a key, the output continues for the next windowful. If, however, the user presses the key '`q`', then the output and the interpretation of the view is interrupted altogether.

Because of this automatic pausing feature of the view interpreter, viewed data does not accidentally scroll of the window and all the data can be seen. But if there is many windowfuls of data this feature is not adequate, because it provides no way to scroll back. As expected, the view definition language provides a feature to avoid this defect.

### *8.4.7 Index Variables*

Each window has one *index variable* that can be used in the definition of a view. The index variable is referred to with identifier '`x`'. The value of the index variable can be used in index expressions and because the index variable is one of the variables, the value of it can be output to the window with the reference

```
x
```

which produces the output

`x:`*Val*

where *Val* is the current value of the index variable. But before the index variable can be referred to in any way, the range of valid values must be defined. The definition is done in the same definition of a view, where the index variable is used in. The definition of the index variable must precede the actual definition of the view. The definition of a view which contains a definition of the index variable has the form

`x`*Range* **:** *View*

where *Range* is a range expression specifying the range of values the index variable can get and *View* is a list of references. The separator list in the *Range* has no effect on the output; it simply separates the lower and upper limit expressions. The view can be said to be a function of the index variable, if the index variable is used in the definition of the view (if the index variable is defined but it is not used, then the view does not depend on the value of the index variable).

When the view is first interpreted, the range for the index variable is first evaluated and the index variable gets the smallest value defined, i.e. the value of the lower limit expression. Then the actual view is calculated and interpreted using this value of the index expression. The value of the index variable can be increased and decreased and set with monitor commands, and when such a change is done, the view is recalculated and reinterpreted using this new value of the index variable. The range of the index variable is calculated only when the view is defined, when the monitor is re-entered, and when the size of the window is changed (see command and function reference later).

A controlled viewing and scrolling of large amounts of data is possible with the use of the index variable. Lets suppose that the PRAM has very large number of processors and the user wants the view all the general registers and current instructions of all the processors processorwise. The user might write a definition of a view

```
p[1;P].(r[1,R],i)
```

but if the `P` is large, as it is, the data won't fit to one or two windows. With a little thinking the definition of the view can be converted so that the processor range expression is a function of the index variable in such way that the definition of the view refers to `W` successive processors at the time, where `W` is the number of rows in the window. If the change of the index variable by one shifts the processor range with `W` processors, then the processors of the PRAM can be viewed and scrolled `W` successive processors at the time. The definition that does the trick is

```
x[1,P/W]:r[(x-1)*W+1;x*W].(r[1,R],i)
```

Here `P` should be divisible with `W`. When the index variable gets the first value one, the view is

```
r[1;W].(r[1,R],i)
```

as can be easily seen. When the index variable gets the value two, the view is

```
r[W+1;2*W].(r[1,R],i)
```

etc. And finally when the index variable gets the value `P/W`, the view is

```
r[P-W+1;P].(r[1,R],i)
```

The use of the index variable requires some thinking but the user of the monitor program knows or at least should know what she of he wants to be viewed.

### *8.4.8 Elements of a View Definition*

Lets summarise the basic elements of a view definition:

| | |
|---|---|
| `p` | processor |
| `r` | register |
| `c` | program counter, PC |
| `s` | signature register, SIG |
| `i` | instruction in "human readable form" |
| `m` | shared memory |
| `t` | common clock |
| `x` | index variable |
| `W` | window size, number of rows in a window |
| `P` | the number of processors |
| `R` | the number of registers (r) in one processor |
| `M` | the largest referred shared memory index |
| `.` | separates processor and register references |
| `:` | separates an index variable definition from actual view definition |
| `,` | space separator |
| `;` | newline separator |
| `[` | starts a range definition |
| `]` | ends a range definition |
| `(` | starts a register list or a parenthesized expression |
| `)` | ends a register list or a parenthesized expression |
| `+` | addition |
| `-` | subtraction or minus-sign |
| `*` | multiplication |
| `/` | division |
| `%` | reminder |

### *8.4.9 Syntax of a View Definition*

The exact syntax of a definition of a view is presented here.

| | | |
|---|---|---|
| *Def* | → | *IndexVarDef RefLst* \| *RefLst* |
| *IndexVarDef* | → | `x`*Range* `:` |
| *Range* | → | `[`*Expr SepLst Expr* `]` |
| *RefLst* | → | *Ref RestOfRefLst* |
| *RestOfRefLst* | → | ε \| *SepLst RefLst* |
| *SepLst* | → | *Sep RestOfSepLst* |
| *RestOfSepLst* | → | ε \| *SepLst* |
| *Sep* | → | *NLSep* \| *SpaceSep* |
| *NLSep* | → | `;` |
| *SpaceSep* | → | `,` |
| *Ref* | → | *SimpleRef* \| *RangeRef* |
| *SimpleRef* | → | *SimpleMemRef* \| *SimpleRegRef* \| *SimpleProcInstrRef* \| *VarRef* \| *SimpleInstrRef* |
| *SimpleMemRef* | → | `m`*Expr* |
| *SimpleRegRef* | → | *ProcRef RegRef* |
| *ProcRef* | → | `p`*Expr* `.` |
| *RegRef* | → | *GenRegRef* \| *PCRef* \| *SIGRef* |
| *GenRegRef* | → | `r`*Expr* |
| *PCRef* | → | `c` |
| *SIGRef* | → | `s` |
| *SimpleProcInstrRef* | → | *ProcRef InstrRef* |
| *InstrRef* | → | `i` |
| *VarRef* | → | *IndexVarRef* \| *ClockRef* \| *WindowSizeRef* *LastMemIndexRef* \| *LastProcIndexRef* \| *LastRegIndexRef* |
| *IndexVarRef* | → | `x` |
| *ClockRef* | → | `t` |
| *WindowSizeRef* | → | `W` |
| *LastMemIndexRef* | → | `M` |
| *LastProcIndexRef* | → | `P` |
| *LastRegIndexRef* | → | `R` |
| *SimpleInstrRef* | → | `i`*Expr* |
| *RangeRef* | → | *MemRangeRef* \| *ProcRangeRegRef* \| *ProcRegRangeRef* \| *ProcRangeRegRangeRef* \| *ProcRegLstRef* \| *ProcRangeRegLstRef* \| *InstrRangeRef* |
| *MemRangeRef* | → | `m`*Range* |
| *ProcRangeRegRef* | → | *ProcRangeRef RegRef* |
| *ProcRangeRef* | → | `p`*Range* `.` |
| *ProcRegRangeRef* | → | *ProcRef RegRangeRef* |
| *RegRangeRef* | → | `r`*Range* |
| *ProcRangeRegRangeRef* | → | *ProcRangeRef RegRangeRef* |
| *ProcRegLstRef* | → | *ProcRef RegLstRef* |
| *ProcRangeRegLstRef* | → | *ProcRangeRef RegLstRef* |
| *RegLstRef* | → | `(` *RegLst* `)` |
| *RegLst* | → | *RegLstElem RestOfRegLst* |

| | | |
|---|---|---|
| *RestOfRegLst* | $\rightarrow$ | ε \| *SepLst RegLst* |
| *RegLstElem* | $\rightarrow$ | *RegRef* \| *RegRangeRef* \| *InstrRef* |
| *InstrRangeRef* | $\rightarrow$ | `i`*Range* |
| *Expr* | $\rightarrow$ | *Term RestOfExpr* |
| *RestOfExpr* | $\rightarrow$ | ε \| +*Expr* \| −*Expr* |
| *Term* | $\rightarrow$ | *Factor RestOfTerm* |
| *RestOfTerm* | $\rightarrow$ | ε \| `*`*Term* \| `/`*Term* \| `%`*Term* |
| *Factor* | $\rightarrow$ | `(`*Expr* `)` \| −*Factor* \| +*Factor* \| *SimpleVal* |
| *SimpleVal* | $\rightarrow$ | *UnsignedInt* \| *SimpleMemRef* \| *SimpleRegRef* \| *VarRef* |

## 8.5 Monitor Program Functions and Commands

The functions and the commands that are used to invoke the functions are described in the following.

*Function* : Create new window

*Command* : w

*Description* : This function creates a new window . Function first writes the prompt "`New window. Number of rows?`" on the input window and starts to wait for the input, the number of rows in the new window . If the number of rows is not valid, then the error message "`Invalid number if rows`" is written on the message window and function is exited and no window is created. If the input is empty , then the function is exited and no window is created. If the number of rows is valid, then a new window is created and it is placed as the last window and is set as the default window and the message "`New window (number` *Num* `) defined`", where *Num* is the number of the created window , is written on the message window . Next the define view function is automatically called and the prompt "`View?`" is written on the input window and the program starts to wait for the input, the definition of a view . When the typing of the view starts, the prompt is cleared. If the input is empty , then no view is attached. If a non-empty input is given, then the defined view is attached to the window and the message "`A view defined and attached to window number` *Num* ", where *Num* is the number of the created window , is written on the message window . Then the definition of the view is interpreted and the data according to the definition is written on the window . If an error is detected during the interpretation, then an error message is given. A '`^`' character is printed to show the location of the error in the definition string. In syntax error messages the location should be correct and accurate, but in semantic error messages the location may be somewhat approximative due to parsing technique.

*Function* : Define view .

*Command* : v

*Description* : This function attaches a view entered by the user to a window . Function first writes the prompt "`Define view. Window number?`" on the input window and starts to wait for the input, the number of the window to which to attach a view. Function gives the default window number as the default input value. If there exists no windows, this command has no effect. If the window number is not valid, then the message "`Invalid window number`" is written on the message window. If the input is empty, then the function is exited and the command has no effect. If the window number entered is valid, then the possible old view is deleted and the prompt "`View?`" is written on the input window and the program starts to wait for the input, the definition of the view. When the typing of the view starts, the prompt is cleared. If the input is empty, then no view is attached. If a non-empty input is given, then the defined view is attached to the window and the message "`A view defined and attached to window number` *Num*", where *Num* is the number of the created window, is written on the message window. Then the definition of the view is interpreted and the data according to the definition is written on the window. If an error is detected during the interpretation, then an error message is given. A '`^`' character is printed to show the location of the error in the definition string. In syntax error messages the location should be correct and accurate, but in semantic error messages the location may be somewhat approximative due to parsing technique.

*Function* : Change window size.

*Command* : c

*Description* : This function changes the size of an existing window. Function first writes the prompt "`Change window size. Window number?`" on the input window and starts to wait for the input, the number of the window the size of which to change. Function gives the default window number as the default input value. If there exists no windows, this command has no effect. If the window number is not valid, then the message "`Invalid window number`" is written on the message window. If the input is empty, then the function is exited and the command has no effect. If the window number entered is valid, then the prompt "`New number of rows?`" is written on the input window and the program starts to wait for the input, the new number of rows in the window. If the number of rows is not valid, then the error message "`Invalid number if rows`" is written on the message window and function is exited and the command has no effect. If the input is empty, then the function is exited and the command has no effect. If the number of rows is valid, then the size of the window is changed and the placements of the windows following the changed one are adjusted. Finally all defined views are reinterpreted and the data according to the definitions is rewritten on the windows.

*Function* : Delete window.

*Command* : d

*Description* : This function deletes a window. Function first writes the prompt "`Delete window. Window number?`" on the input window and starts to wait for the input, the number of the window which to delete. Function gives the default window number as the default input value. If there exists no windows, this command has no effect. If the window number is not valid, then the message "`Invalid window number`" is written on the message window. If the input is empty, then the function is exited and the command has no effect. If the window number entered is valid, then the window is deleted (along with the possible definition of a view) and the message "`Window number` *Num* `deleted`", where *Num* is the number of the deleted window, is written on the message window. Finally the placements and the numbers of the windows following the deleted one are adjusted and all remaining defined views are reinterpreted and the data according to the definitions is rewritten on the windows. If the deleted window, window number *Num*, is the default window, then the new number *Num* is set as the default window, or if there no more exists a window number *Num*, then the window number *Num* - 1 is set as the default window.

*Function* : Delete all windows.

*Command* : D

*Description* : This function deletes all defined windows (and all defined views). After the deletion, the message "`All windows deleted`" is written on the message window.

*Function* : Scroll forward.

*Command* : [*Unsigned*]f

*Description* : This function increases the index variable of the active window. By default the increment is 1. The increment can be changed by preceding the command with an unsigned integer argument. The unsigned integer typed is the increment. When the typing of the argument has started, the command can be cancelled by deleting the argument and pressing some non-numeric key. If the new value of the index variable would get larger than the largest defined value for the index variable, then the index variable gets the largest defined value. After the value of the index variable is increased, a new view is calculated and interpreted and the data according to the view is displayed on the (default) window.

*Function* : Scroll all forward.

*Command* : [*Unsigned*]F

*Description* : This function increases the index variables of all windows. The effect is as the same scroll forward command was given separately to each window.

*Function* : Scroll backward.

*Command* : [*Unsigned*]b

*Description* : This function decreases the index variable of the active window. By default the decrement is 1. The decrement can be changed by preceding the command with an unsigned integer argument. The unsigned integer typed is the decrement. When the typing of the argument has started, the command can be cancelled by deleting the argument and pressing some non-numeric key. If the new value of the index variable would get smaller than the smallest defined value for the index variable, then the index variable gets the smallest defined value. After the value of the index variable is decreased, a new view is calculated and interpreted and the data according to the view is displayed on the (default) window .

*Function* : Scroll all backward.

*Command* : [*Unsigned* ]B

*Description* : This function decreases the index variables of all windows. The effect is as the same scroll backward command was given separately to each window .

*Function* : Goto.

*Command* : *Unsigned* g

*Description* : This function sets the specified unsigned integer value as the value of the index variable of the default window . The value is the argument preceding the actual command key ‘ `g`’. When the typing of the argument has started, the command can be cancelled by deleting the argument and pressing some non-numeric key . If the new value of the index variable would get larger than the largest defined value for the index variable, then the index variable gets the largest defined value. If the new value of the index variable would get smaller than the smallest defined value for the index variable, then the index variable gets the smallest defined value. After the value of the index variable is changed, then a new view is calculated and interpreted and the data according to the view is displayed on the (default) window .

*Function* : Goto all.

*Command* : *Unsigned* G

*Description* : This function sets the specified unsigned integer value as the value of the index variables of all windows. The effect is as the same goto command was given separately to each window .

*Function* : Set initial index variable value.

*Command* : i

*Description* : This function sets the current value of the index variable of a window as the initial value of that index variable. The initial value is the value the index variable gets when the monitor is invoked. If the initial value is out of range of the valid index variable values, then the index variable gets the smallest value defined. Function first writes the prompt "`Set initial index variable value. Window number?`" on the input window and starts to wait for the input, the number of the window the initial value of the index variable of which is to be set. Function gives the default window number as the default input value. If there exists no windows, this command has no effect. If the window number is not valid, then the message “ *Invalid window number* ” is written on the message window . If the input is empty , then the function is exited and the command has no effect. If the window number entered is valid, then the current value of the index variable of the window is set as the initial value of that index variable. When the monitor is next called, the index variable gets the initial value.

*Function* : Set all initial index variable values.

*Command* : I

*Description* : This function sets the initial values of the index variables of all windows. The effect is as the set initial index variable value command was given separately to each window .

*Function* : Set default window .

*Command* : *Unsigned* *<return>*

*Description* : This function sets the specified window as the default window . The number of the new default window is given as an argument to the command. When the typing of the argument has started, the command can be cancelled by deleting the argument and pressing some non-numeric key . If the entered window number is valid, then the specified window is set as the default window . If the number is not valid, then the default window is not changed. In any case, the function finally writes the message " `Default window is window number` *Num* ", where *Num* is the default window number , on the message window .

*Function* : Toggle trace mode.

*Command* : t

*Description* : This function changes the trace mode status of the emulator . If the trace mode is on, then after this command the trace mode is off; if the trace mode is off, then after this command the trace mode is on. The program writes the new toggle mode status on the message window when this command is given, i.e. either "`Trace ON`" or "`Trace OFF`". When trace mode is on, then there is an implicit `@MONITOR` directive between every instruction of the PRAM code block of the PEL program and before the first and after the last instruction and all explicit directives are deactivated. When trace mode is off, then the explicit directives which were set as active with emulator command line options are again active and no implicit `@MONITOR` directives exist.

*Function* : Quit monitor .

*Command* : Q

*Description* : The function quit the monitor . After this command the execution of the PRAM code continues. The message " `Quit monitor. Execution continues...`" is written on the input window .

*Function* : Halt emulator .

*Command* : H

*Description* : This function sets the execution status of the emulator as fail which causes the emulator to halt. The message " `Execution halted.`" is written on the input window .

## 8.6 View Interpreter Error Messages

The view interpreter interprets the definitions of the views and outputs the data on the windows. The interpreter can detect two kinds of errors: errors in the syntax of the definition and errors in the semantics of the definitions, e.g. invalid value of an index expression.

An error message consists of a message written on the first line of the message window, the erroneous definition of the view written on the second line of the message window, and a ‘^’ character written under the definition of the view to point to the error location.

Syntax error positions are always accurate, but due to the parsing technique, the semantic error positions are somewhat approximative. It is, however, always true, that a semantic error position is somewhere before the pointing ‘^’ character, not after it.

### *8.6.1 Syntax Error Messages*

Message: Invalid definition.
*Description* : An unknown or an out of context symbol was detected.

*Message* : ‘)’ expected.
*Message* : ‘,’ or ‘;’ expected.
*Message* : ‘.’ expected.
*Message* : ‘:’ expected.
*Message* : ‘;’ or ‘,’ or <end of definition> expected.
*Message* : ‘]’ expected.
*Message* : ‘r’ or ‘c’ or ‘s’ expected.
*Message* : ‘r’ or ‘c’ or ‘s’ or ‘i’ expected.
*Description* : The interpreter expects to find the mentioned symbol instead of the one under the pointing ‘^’ character.

### *8.6.2 Semantic Error Messages*

*Message* : Division by zero.
*Description* : The expression on the right of a division operator ‘/’ or a reminder operator ‘%’ evaluates to zero.

*Message* : Invalid memory reference.
*Message* : Invalid processor reference.
*Message* : Invalid program word reference.
*Message* : Invalid register reference.

*Description* : The value of an index expression or the value of a boundary expression of an range expression is not within the domain of that index.

*Message* : Lower limit of index variable greater than upper limit

*Description* : Self explanatory .

*Message* : No index variable defined.

*Description* : A reference to an index variable (' `x`') was detected, but there was no definition of the range of the index variable at the beginning of the definition of the view .

## 9. GENERAL RUN-TIME ERRORS

Both the PEL assembler and the emulator use the same file access and memory allocation services. Some errors may occur during these actions. Most of the errors are rare and indicate a bug in the program or happen due to misuse of the computer system. These errors cause the program to terminate.

There are also traps laid for bugs in the program. These traps, when triggered, cause an internal error . An internal error message is given and a program exits immediately if an internal error occurs.

The general run-time messages and the internal error message are described in the following.

| | |
|---|---|
| *Message* : | Error while opening file '*File*' |
| *Description* : | A file named *File* could not be opened. Usually the reason is that the file named *File* does not exists. |
| | |
| *Message* : | Error while closing file '*File*' |
| *Message* : | Error while removing file '*File*' |
| *Message* : | Error while reading from file '*File*' |
| *Message* : | Error while writing to file '*File*' |
| *Message* : | Error while reading a character from file '*File*' |
| *Message* : | Error while setting file position for file '*File*' |
| *Message* : | Error while getting file position of file '*File*' |
| *Description* : | These error messages are self-explanatory about what went wrong in file access. The *File* in the above messages is the name of the file accessed. If the filename looks strange and unfamiliar in the message, the file is probably a temporary file created by the system. The reasons why the error occurred are system and situation dependent and therefore impossible to list here. These errors are very rare and might indicate that the computer system is used somehow wrong or that there are bugs somewhere. None of these errors occurred during the testing of the PRAM emulator system. The author or the maintenance person of the system or the local system wizard should be consulted if one of these errors occur . |
| | |
| *Message* : | Memory allocation error |
| *Description* : | There is not enough memory in the computer to execute the task requested. |
| | |
| *Message* : | *File*:*Line* : Internal error . Contact author |
| *Description* : | There is a bug in the program and the bug was detected in source file *File* on line *Line* . Report this message along with the input and other necessary information about the run and the system to the author of this system or to the maintenance person of this system. |

## REFERENCES

*SunOs Release 4.1 Manual Pages on cpp.* Sun Microsystems Inc., Mountain View, USA, 1988.

Juvaste S. 1992: *An implementation of the programming language pm2*. University of Joensuu, Department of Computer Science, Report A-1992-1. ISSN 0789-7316. ISBN 951-708-112-X.

Juvaste S. 1992: *Programming language pm2*. University of Joensuu, Department of Computer Science, Report B-1992-1.

Katajainen J. 1990: *Lecture notes on the theory of parallel algorithms* (In Finnish). University of Turku, Technical Report C9. ISSN 0357-8542.

Kernighan B.W., Ritchie D. 1988: *The C Programming Language, 2nd ed.* Prentice Hall. ISBN 0-13-110362-8.

Penttonen M. Personal communication 1991 - 1992.

Plauger P.J., Brodie J. 1989: *Standard C: programmer's quick reference*. Microsoft Press. ISBN 1-55615-158-6.

*Programmer's Reference Manual for Curses on the Sun Workstation.* 1986. Sun Microsystems Inc., Mountain View, USA.

## APPENDIX A: INSTALLATION GUIDE

The PRAM emulator system (the PEL assembler and the emulator itself) is distributed in source code form and the source code has to be compiled and linked in the computer system to which the emulator is to be installed. The compiling and linking of the source to get the two executable programs, the PEL assembler and the emulator, is actually nearly all the installation needed. Different systems may, of course, have their own peculiarities.

### A.1 System Requirements

The PRAM emulator system is written in ANSI C programming language (Kernighan and Ritchie, 1988, Plauger and Brodie, 1989). The computer system must provide an ANSI C compiler.

The user interface of the monitor program of the emulator is written using terminal screen handling package curses (Programmer's Reference Manual for Curses on the Sun Workstation). The system must include the curses package. At least UNIX systems include (should include) this package and public domain version for PCs (PDCurses) is available.

The PEL language and the PEL assembler are designed with the idea that a C language preprocessor is used as the first phase of assembling, i.e. that a C language preprocessor could be used as a preprocessor or a macro processor for the PEL language. UNIX systems usually include a separate C preprocessor or the C compiler can be used to output the preprocessed file.

Some C compilers, especially for PCs, have built-in preprocessors that cannot be used as separate preprocessors or the preprocessors do not accept other than C language programs. At least for PCs, a public domain C preprocessor is available. (The PRAM emulator system was successfully ported to PC using Borland C++ compiler, PDCurses, and a public domain C preprocessor.)

The PEL assembler can be used without a C preprocessor, but then comments and other features that the preprocessor provides cannot be used. In principle, also other preprocessor can be used. The PEL assembler requires that the preprocessed file that the assembler gets as input follows the syntax of the PEL language and may besides the language statements include line control information lines of the format

# *LineNum* "*FileName*"

and

```
# line LineNum
```

and

```
# line LineNum "FileName"
```

These lines are normally included to the start of the source files and when a file is included to the source file (with `#include` directive in C). The control lines inform the assembler about the current filename ( *FileName* ) and line number ( *LineNum* ). This information is necessary in order to the line numbers and file names in the error messages to be correct. The first non white space character of a control line must be the '#' sign. The end of a control line, after the "*FileName*" or the *LineNum*, is ignored.

### A.2 Compiling

The source code for the PRAM emulator system is in the following files:

| | | |
|---|---|---|
| asmdefs.h | genstack.c | pgmmgr .h |
| asmerr .c | genstack.h | pram.c |
| asmerr .h | instrmgr .c | rammgr .c |
| avltree.c | instrmgr .h | rammgr .h |
| avltree.h | labelmgr .c | ramset.c |
| cmdfuncs.c | labelmgr .h | ramset.h |
| cmdfuncs.h | lneditor .c | readmgr .c |
| codemgr .c | lneditor .h | readmgr .h |
| codemgr .h | memmgr .c | reffuncs.c |
| defintrp.c | memmgr .h | reffuncs.h |
| defintrp.h | mondef.h | scanner .c |
| dirmgr .c | monitor .c | scanner .h |
| error .c | monitor .h | smemmgr .c |
| error .h | parser .c | smemmgr .h |
| extmgr .c | parser .h | symtbl.c |
| extmgr .h | pasm.c | symtbl.h |
| filemgr .c | pedefs.h | writemgr .c |
| filemgr .h | pgmmgr .c | writemgr .h |

Some of the source files are part of the PEL assemble, some are part of the emulator , and some implement general services used in both programs. The header files, i.e. the files that have the '.h' specifier , are included to the C files, i.e. the files that have the '.c' specifier , automatically with directives

```
#include "FileName"
```

The header files should be placed in a directory where the C compiler can find them. Usually the same directory where the C files are is proper .

All the C files should be compiled with the ANSI C compiler to get the corresponding object files (files with the specifier .o or .OBJ or something else depending on the system). The compiler may give some warnings but no fatal errors should occur . If fatal errors occur , then the compiler used in not full ANSI C compiler , or there is something wrong with the installation of the compiler , or the computer system is not suitable (for example, there is no curses package; the curses.h header file does not exist). If fatal errors occur , consult the local system and C language wizards.

(The PRAM emulator system also includes files debug.h and debug.c. These files include some debugging aids but they are not used normally , so normally these files are not needed. If the maintenance person wants to use these debugging aids, she or he should compile the source to be debugged along with these debug files with the symbol “debug” defined.)

## A.3 Linking

The PEL assembler and the emulator can be linked together from the object files created. Of course, the standard C library is needed, as is the curses library .

### *A.3.1 Linking the PEL Assembler*

The PEL assembler is linked together from the object files resulting from the C files:

| | | |
|---|---|---|
| asmerr .c | filemgr .c | parser .c |
| avltree.c | genstack.c | pasm.c |
| codemgr .c | labelmgr .c | scanner .c |
| error .c | memmgr .c | symtbl.c |

The name of the resulting executable can be chosen freely . The author uses the name “pasm” (or “pasm.exe” in MS-DOS).

This program (pasm) accepts as input a preprocessed PEL program file. The command to assemble a preprocessed file is

`pasm` *Source* *Object*

where *Source* is the name of the preprocessed file (or the file that does not need preprocessing) and *Object* is the desired name of the assembled file.

The pasm program returns to the target environment the execution status. Usually the execution status is not needed. (Consult the documentation of the computer system and the operating system how to check the return status of a program if the status is needed.) If no errors happen in the execution of the program itself and if no errors are detected in the PEL program, then the status `EXIT_SUCCESS` is returned; otherwise the status `EXIT_FAILURE` is returned. These values are defined in ANSI C and they are implementation dependent. The exact values can be find in the documentation of the used C compiler.

Depending on the computer system, some kind of shell script can be built to make the preprocessing phase seem automatic. For example, in UNIX system the following Bourne shell script can be used (path names may have to be specified):

```
if [ $# -ne 2 ]
then
     echo "Usage: pa <source> <object>"
else
     cpp $1 > pasmtmp$$ && pasm pasmtmp$$ $2
     rm pasmtmp$$
fi
```

Here, it is assumed that '`pa`' is alias for command '`sh` *Script*' or '`pa`' is an executable file including the command '`sh` *Script* `$1 $2`', where *Script* is the name of the above shell script file. The command to assemble a PEL program is thus

`pa` *Source Object*

where *Source* is the name of a source PEL program file and *Object* is the desired named of the assembled file.

### *A.3.2 Linking the Emulator*

The emulator is linked together from the object files resulting from the C files:

| | | |
|---|---|---|
| avltree.c | instrmgr .c | rammgr .c |
| cmdfuncs.c | lneditor .c | ramset.c |
| defintrp.c | memmgr .c | readmgr .c |
| dirmgr .c | monitor .c | reffuncs.c |
| error .c | pgmmgr .c | smemmgr .c |
| extmgr .c | pram.c | writemgr .c |
| filemgr .c | | |

Also the curses library must be linked along. Consult the local curses documentation and the documentation of the used C compiler for information about the curses library and linking of it.

The name of the resulting executable can be chosen freely. The author uses the name "pram" (or "pram.exe" in MS-DOS).

This program (pram) accepts as input an assembled PEL program file, the result of the PEL assembler.

The pram program returns to the target environment the execution status. Usually the execution status is not needed. (Consult the documentation of the computer system and the operating system how to check the return status of a program if the status is needed.) If no errors happen in the execution of a program, then the status EXIT_SUCCESS is returned; if an error happens or the user halts the execution in the monitor program with the halt command, then the status EXIT_FAILURE is returned. These values are defined in ANSI C and they are implementation dependent. The exact values can be find in the documentation of the used C compiler .

### A.4 Some Notes

Some compilers, for example the GNU C compiler used to develop the PRAM emulator system, do not include ANSI C header files. The "standard" header files of the computer system may then let some C language macros undefined. The definitions may have to be written by hand. For example, the following definitions had to be added to the header file "pedefs.h" in order to be able to compile the PRAM emulator system with the GNU C compiler and the UNIX system used:

```
/*
* If stdio.h does not include the following
* definitions, they are presented here.
***********************************************/
#ifndef SEEK_SET
     #define SEEK_SET 0
#endif
/***********************************************/

/*
* If stdlib.h does not include the following
* definitions, they are presented here.
***********************************************/
#ifndef EXIT_SUCCESS
     #define EXIT_SUCCESS 0
#endif
```

```
#ifndef EXIT_FAILURE
     #define EXIT_FAILURE 1
#endif
/**********************************************/
```

Note that these definitions are implementation dependent and are not portable (should be, however , valid in UNIX and MS-DOS systems)!

In principle, the amount of the memory in the computer system, places the only restriction to the size of the PRAM (the number of processors) that can be emulated. Some system, however , may restrict the size of the largest contiguous memory block by some means. For example, an ordinary PC has a fixed-segmented memory and the segments are just 64 kilobytes. The largest contiguous data structure allocated during the execution of the emulator may thus place restrictions to the size of the PRAM. The size of the largest allocated memory block (in practice) can be calculated with the formula $P \cdot (2+R) \cdot I$, where $P$ is the number of processors in the PRAM, $R$ is the number of general register in each processor , and $I$ is the size in bytes of the long int type in the target machine (usually 4). By setting this formula smaller than or equal to the largest contiguous memory block, the maximum size of a PRAM (the number of processors and the number of registers) can be determined

## APPENDIX B: SOME REFERENCES CONCERNING THE PRAM MODEL

In the references listed in this appendix, different kinds of PRAM models are defined and used. Also some "near PRAM" models and ideas before the actual PRAM model are presented.


Blelloch G.E. 1989: Scans as Primitive Parallel Operations. *IEEE Transactions on Computers* **38**,11 1526-1538. ISSN 0018-9340.

- Strong scan operations taken as part of the PRAM model.

Borodin A., Munro I. 1975: *The Computational Complexity of Albegraic and Numeric Problems.* American Elsevier Publishing Company, Inc., New York, ISBN 0-444-00168-9.

- One of the first PRAM models (CREW PRAM).

Chlebus B.S., Diks K., Hagerup T., Radzik T. 1988: Efficient Simulations Between Concurrent-Read Concurrent-Write PRAM Models. *Mathematical Foundations of Computer Science, Proceedings of the 13th Symposium* (Ed. J. Kruska, B. Rovan, J. Wiedermann), 231-239.

- COLLISION+ variant (and many more)

Cook S., Dwork C. 1982: Bounds on the Time for Parallel RAM's to Compute Simple Functions. *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, 231-233. ISBN 0-89791-067-2.

- A formal definition of PRAM.

Fich F.E., Ragde P., Wigderson A. 1988: Relation Between Concurrent-Write Models of Parallel Computation. *SIAM Journal on Computing* **17**,3 606-627. ISSN 0097-5397.

- COLLISION variant.

Fortune S., Wyllie J. 1978: Parallelism in Random Access Machines. *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, 114-118.

- This reference which presents the P-RAM model is perhaps the mostly referred to reference in "PRAM papers". But the P-RAM model has not really been used as is.

Galil Z. 1984: Optimal Parallel Algorithms for String Matching. *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, 240-248. ISBN 0-89791-133-4.

- WRAM = WEAK CRCW PRAM.

Goldschlager L.M. 1982: A Universal Interconnection Pattern for Parallel Computers. *Journal of the ACM* **29**,4 1073-1086. ISSN 0004-5411.

- SIMDAG (single instruction stream, multiple data stream, global memory). This model is very near to the PRIORITY CRCW PRAM. In SIMDAG a single processor executes the sequential instructions and spreads the parallel instructions to parallel processors.

Gottlieb A., Lubachevsky B.D., Rudolph L. 1983: Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors. *ACM Transactions on Programming Languages and Systems* **5**,2 164-189. ISSN 0164-0925.

- A serialization principle presented as a write conflict resolution rule.

Grolmusz V., Ragde P. 1987: Incomparability In Parallel Computation. *28th Annual Symposium on Foundations of Computer Science*, 89-98. ISSN 0272-5428.

- TOLERANT CRCW PRAM.

Hirschberg D.S. 1978: Fast Parallel Sorting Algorithms. *Communications of the ACM* **21**,8 657-661. ISSN 0001-0782.

- First ideas of EREW.

Karp R.M., Miller R.E. 1969: Parallel Program Schemata. *Journal of Computer and System Sciences* **3**,2 147-195.

- Parallel program schemata. Formal and quite practical model but very little used.

Kruskal C.P., Rudolph L., Snir M. 1990: A Complexity Theory of Efficient Parallel Algorithms. *Theoretical Computer Science* **71**,1 95-132. ISSN 0304-3975.

- Strong PRAM (which is very strong) and CROW PRAM (Owner Write).

Kucera L. 1982: Parallel Computation and Conflicts in Memory Access. *Information Processing Letters* **14**,2 93-96. ISSN 0020-0190.

- WPRAM (a kind of WEAK CRCW PRAM) and a kind of COMMON CRCW PRAM.

Lev G.F., Pippenger N., Valiant L. 1981: A Fast Parallel Algorithm for Routing in Permutation Networks. *IEEE Transactions on Computers* **C-30**,2 93-100. ISSN 0018-9340.

- Parallel RAC (Random Access Computer).

Ragde P., Steiger W., Szemerédi E., Wigderson A. 1988: The Parallel Complexity of Element Distinctness is $\Omega(\sqrt{\log n})$. *SIAM Journal on Discrete Mathematics* **1**,3 399-410. ISSN 0895-5397.

- Write Once COMMON PRAM.

Rosenfeld J.L 1969: A Case Study in Programming for Parallel-Processors. *Communications of the ACM* **12**,12 645-656.

- Used a parallel computer simulator.

Savitch W .J., Stimson M.J. 1979: T ime Bounded Random Access Machines with Parallel Processing. *Journal of the ACM* **26**,1 103-1 18. ISSN 0004-541 1.

- k-PRAM which is like P-RAM but does not have shared memory .

Schwartz J. 1966: Large Parallel Computers. *Journal of the ACM* **13**,1 25-32.

- Presented an ideal parallel computer model which has some ideas of the PRAM.

Schwartz J.T . 1980: Ultracomputers. *ACM Transactions on Programming Languages and Systems* **2**,4 484-521. ISSN 0164-0925.

- Paracomputer which is like CRCW PRAM.

Shiloah Y ., Vishkin U. 1981: Finding the Maximum, Merging, and Sorting in a Parallel Computation Model. *Journal of Algorithms* **2** 88-102. ISSN 0196-6774.

- A kind of COMMON CRCW .

Snir M. 1985: On Parallel Searching. *SIAM Journal on Computing* **14**,3 688-708. ISSN 0097-5397.

- BEREW (Broadcast EREW).

Vishkin U. 1983: Implementation of Simultaneous Memory Address Access in Models That Forbid It. *Journal of Algorithms* **4** 45-50. ISSN 0196-6774.

- Mentions the ERCW model.

Vishkin U., W igderson A. 1983: T rade-offs Between Depth and W idth in Parallel Computation. *24th Annual Symposium on Foundations of Computer Science* , 146-153. ISSN 0272-5428.

- CREW PRAM(m), where m is the size of the shared memory .