UNIVERSITY OF JOENSUU
DEPARTMENT OF COMPUTER SCIENCE
Report Series B

AN F-PRAM EMULATOR

Jukka Veräjäntausta

Report B-1998-1

ACM D.1.3

TABLE OF CONTENTS

ABSTRACT **1**
1. INTRODUCTION **2**
2. OVERVIEW OF THE F-PRAM MODEL **3**
2.1. Components of the model **3**
2.2. Parameters of the model **4**
2.3. The definition of the future **5**
3. THE F-PRAM EMULATOR LANGUAGE FASM **6**
3.1. Preprocessing **6**
3.2. Syntax **6**
3.3. Directives **7**
3.4. Instruction set **9**
4. USAGE OF THE ASSEMBLER **12**
5. USAGE OF THE EMULATOR **13**
5.1. Principle of functioning **13**
5.2. Command line of the emulator **13**
5.2.1. Command line options *14*
5.2.2. Command line error messages *15*
6. MODULE REFERENCE **16**
6.1. Main modules **16**
6.2. Other modules **16**
7. GENERAL PROGRAMMING STYLE AND TECHNIQUE ISSUES **21**
7.1. Naming of the identifiers **21**
7.2. Header files **21**
7.3. Data structures **21**
7.4. Function headers **22**
7.5. Run-time error handling **22**
7.6. Debugging **22**
8. STRUCTURE OF THE FASM ASSEMBLER PROGRAM **24**
8.1. General structure of the FASM assembler **24**
8.2. Solving of the label references **24**
8.3. Directive assignment **25**
9. STRUCTURE OF THE EMULATOR PROGRAM **26**
9.1. The implementation of the shared memory **26**
9.2. The implementation of the interconnection network **26**
9.3. The implementation of the synchronization network **27**
9.4. The implementation of the input/output handling **27**
9.5. The implementation of the directive manager **27**
9.6. One RAM **27**
9.7. Command functions **28**
9.8. The implementation of the reference functions **28**
10. ASSEMBLED FASM FILE **30**
11. CHANGING THE EMULATOR SYSTEM **32**
11.1. Changing constant values **32**
11.1.1. Memory segment size *32*
11.1.2. Preprocessor *32*
11.1.3. Future ready - value *32*
11.2. Changing instruction set **32**
11.3. Changing random function **33**

12. INSTALLING THE EMULATOR SYSTEM 36
12.1. System requirements 36
12.2. Makefile 36
12.3. Compiling 36
12.4. Linking 37
13. ERROR MESSAGES 39
13.1. FASM assembler error messages 39
13.2. Emulator error messages 40
13.3. General run-time error messages 41
REFERENCES 43

ABSTRACT

The F-PRAM emulator system is a tool for simulating parallel computation in a sequential computer. The emulator gives the user a possibility to modify the emulator for one's own needs. The system is based on a theoretical parallel computation model called F-PRAM (Future PRAM) and implements the components the model describes.

The emulator system consists of an FASM assembler and an F-PRAM -emulator. The FASM assembler converts an assembler style program into a form that the emulator understands. The user can define the amount of virtual processors the emulator uses and each of the processors executes the FASM program asynchronously instruction by instruction until they reach the end of the code.

The emulator can be directed (by user) to return not only the results of calculations, but also the amount of used clock cycles and shared memory. This facilitates the analysation of the FASM programs. The settings of the emulator, that describe the features of the parallel computer, can also be altered. This facilitates the comparison of different parallel computers.

Keywords: F-PRAM, Emulator, Parallel, Programming

1. INTRODUCTION

Parallel computing is an important research area in computer science because parallelism gives one possibility to improve the performance of computers. 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.

An F-PRAM emulator is a software emulator of a theoretical model of parallel computing called Future Parallel Random Access Machine (F-PRAM). The model introduces a set of parameters that describe the qualities of a parallel machine. By modifying these parameters the user can create a parallel machine that matches his/hers needs.

In practice the emulator is controlled by the user via a simple assembler style programming language. The system includes an assembler for the language and the actual emulator that executes the assembled programs. The emulator system is written in ANSI C (Kerninghan & Ritchie 1988) programming language and is distributed in that form. Therefore the emulator is portable provided that an ANSI C compiler is available.

First five chapters of this paper describe the emulator system from the user's viewpoint. The chapters from six to eleven specify the structure of the system and gives some necessary information for the programmers who wish to modify the system. The rest of this paper describes the installation procedure and the error messages of the system.

2. OVERVIEW OF THE F-PRAM MODEL

The F-PRAM model is a parallel computation model which is based on the widely used PRAM model (JáJá 1992). In the F-PRAM model processors work *asynchronously* and communicate by means of *shared memory*. All the memory references to the shared memory go through the *interconnection network*. The F-PRAM model uses a set of parameters to characterize the parallel computers.

2.1. Components of the model

Components of the model are presented in figure 1. The *P* processors execute the same program code asynchronously. The processors use mainly *local memories* in their normal computation, but the communication between processors is handled via the shared memory. The processors send write and read (future) requests to the shared memory via the interconnection network.

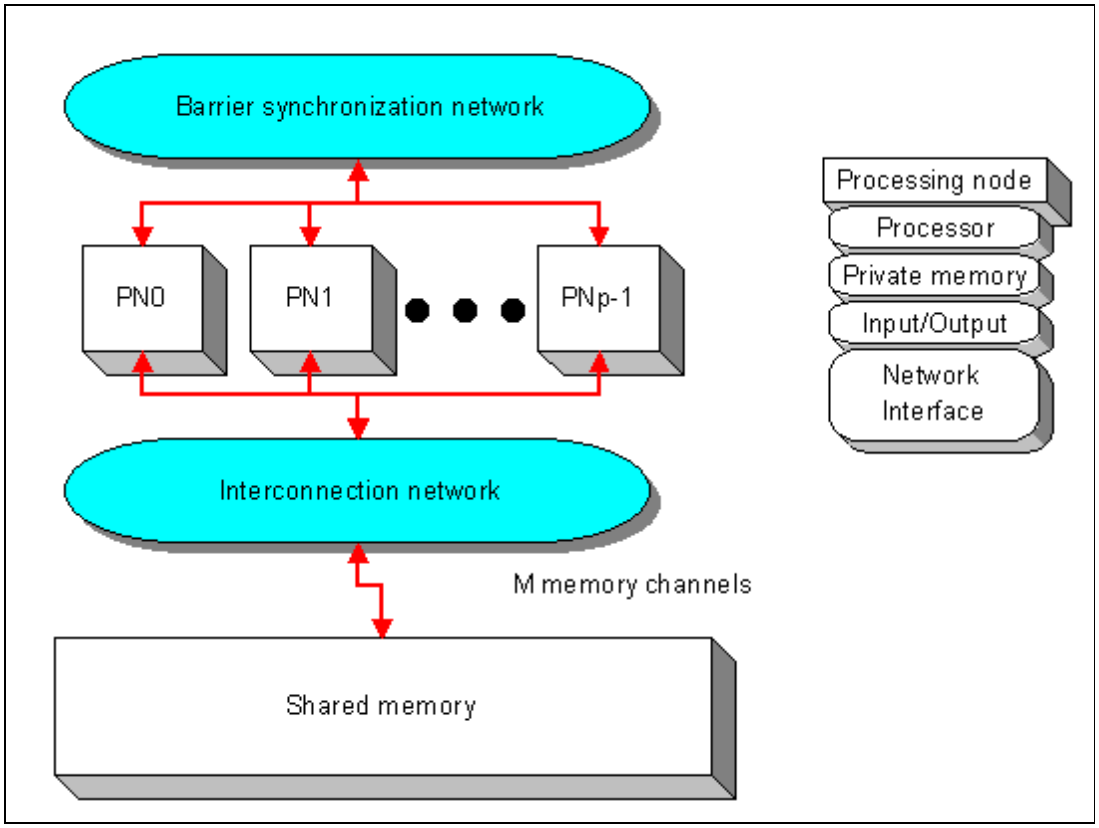

Figure 1. Component of the F-PRAM model (Juvaste 1996).

The processors work asynchronously executing the program code at their own rate. This may cause a need to *synchronize* the processors and to ensure that all the memory references have been completed before continuing the execution of the program code. Synchronization is accomplished with a so-called barrier synchronization routine which makes all the processors wait until all of them have finished the synchronization procedure.

2.2. Parameters of the model

The F-PRAM model uses a set of parameters to model the parallel computers. Parameters are divided into two categories, *primary* and *secondary parameters* (table 1). Primary parameters describe the most important qualities of a parallel machine and these are necessary to take into account when analysing algorithms. Secondary parameters are mostly adjusting the primary ones and these are used to patch gaps in main analysation areas.

Table 1. The parameters of the F-PRAM model (Juvaste 1997a).

Primary parameters		**Typical values**
P	The number of processors	
L	The latency of shared memory references	>= 2 x ∅ of interconnection network
B	The bandwidth inefficiency	>= 1, e.g. P/M x $speed_p/Speed_M$
S	The synchronization delay	>= L, even L x logP
Secondary parameters		
B_v	The single variable bandwidth	<= 1
B_M	The single memory module bandwidth	<= 1
M	The number of shared memory modules	P
B_{IO}	The input/output bandwidth inefficiency	>= 1
B_p	The processor communication overhead	>= 1
B_B	The block reference bandwidth inefficiency	< B

In a real parallel computer, the number of processors (P) is always finite. Normally algorithms can be written using more processors than really exists, because the compiler should be able to share the execution of the algorithm to P processors. The latency (*L*) of shared memory references means the worst expected time that takes to complete a shared memory reference. The bandwidth inefficiency (*B*) means the ability of the network to fullfill the shared memory references. The value B is the minimum possible time between requested memory references without saturating the network. The synchronization delay (*S*) is a time starting from the moment, when the last

processor has begun to execute the synchronization command, and ending, when all the processors have started executing the program code again.

The secondary parameters can normally be ignored (which equals to using their default values). They are useful only when there is need for more refined analysation of the algorithms.

2.3. The definition of the future

The communication between the processors is handled via the interconnection network. When the processor needs a value from the shared memory, it issues a shared memory reference, i.e. a future. The syntax of the future in programming language notation is as follows:

future *local_variable* := *shared_variable*;

where the local variable *local_variable* do not yet get the value of the *shared_variable*. The local variable is assigned a placeholder, a future, which is not a usable value (Juvaste 1997a). After issuing the future, the processor continues its execution. The interconnecting network and the shared memory are responsible for handling the memory request and within the L time units the placeholder should be replaced by the actual value of the shared variable.

3. THE F-PRAM EMULATOR LANGUAGE FASM

The F-PRAM emulator is controlled via a programming language called FASM. FASM is an assembler style case-sensitive language and it can be written in any text editor that produces normal ASCII text. This FASM text file can also be produced by some higher level programming language compiler, such as the FPM compiler by Juvaste (1997b).

3.1. Preprocessing

As the first phase of the assembling of a FASM program, the source file is first passed through the C language preprocessor. Every feature of the C preprocessor may be used in FASM programs. Especially macros and file inclusion are very useful features of the C preprocessor. For example, the instruction set of the FASM language 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 description of the C preprocessors can be found in C language, C preprocessor, and C compiler manuals.

3.2. Syntax

After the preprocessing phase an FASM program should follow the syntax described below (actually the C preprocessor inserts line number information lines to the source but these are not visible to the user). The nonterminal symbols are written in italics. The vertical bar '|' separates alternative derivations. The epsilon (e) means an empty derivation.

Pgm	->	*FASMCode*
FASMCode	->	*InstrSeq*
InstrSeq	->	e | *SingleInstr InstrSeq*
SingleInstr	->	*Dir* | *SimpleInstr* | *InstrWithLabel*
Dir	->	@CLOCK | @LASTMEMINDEX
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 | FCHECK | INC | DEC | EXP | LOGB
ToAccuArg	->	*ImmediateInt* | *DirectReg* | *IndirectReg*
ToRegInstr	->	*ToRegCmd ToRegArg*
ToRegCmd	->	STORE
ToRegArg	->	*DirectReg* | *IndirectReg*
NonArgInstr	->	*NonArgCmd*
NonArgCmd	->	LOADINDEX | LOADPC | HALT | SYNCHRONIZE
MemAccInstr	->	*MemAccCmd MemAccArg*
MemAccCmd	->	READ | WRITE | FWRITE | FUTURE
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 identifiers must differ from the reserved words. An unsigned integer (*UnsignedInt*) is a string of decimal digits 0-9. A natural number (*Natural*) is an unsigned integer greater than zero.

3.3. Directives

Directives are an important tools for analysing the F-PRAM program and a system created with the emulator parameters. A directive (@CLOCK, @LASTMEMINDEX) is not an instruction that a processor executes or handles, but a hint for the emulator to interrupt the execution of an FASM program, and to perform the function the directive suggests, and then continue the execution of the FASM program. The functions the different directives perform are described in the following.

Directive: @CLOCK
Description: Directive @CLOCK writes the current value of the common clock of the F-PRAM to the output queue of the current processor in format

@CLOCK: Value

and a space after it.

Directive: @LASTMEMINDEX
Description: Directive @LASTMEMINDEX writes the largest shared memory address to the output queue in format

@LASTMEMNDX: Value

and a space after it.

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
@CLOCK 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 one.

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

```
@CLOCK 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.

3.4. Instruction set

An FASM program is described with a list of instructions. Instructions can be labelled with one or more label strings followed by a colon ":". For example

```
loop: ADD 1
wait: waitloop: MUL *2
```

Execution proceeds in sequential order unless a jump instruction doesn't change the flow of control. In the following list are described all available instructions.

ADD = *i*	$r_1 \leftarrow r_1 + i$	ECALL =0	$r_1 \leftarrow$Input int
ADD *k*	$r_1 \leftarrow r_1 + r_k$	ECALL =1	$r_1 \leftarrow$Input char
ADD **k*	$r_1 \leftarrow r_1 + r_{r_k}$	ECALL =2	Output int $\leftarrow r_1$
		ECALL =3	Output char $\leftarrow r_1$
AND =*i*	$r_1 \leftarrow r_1$ and *i*		
AND *k*	$r_1 \leftarrow r_1$ and r_k	EXP =*i*	$r_1 \leftarrow i^r 1$
AND **k*	$r_1 \leftarrow r_1$ and r_{r_k}	EXP k	$r_1 \leftarrow r_k{}^r 1$
		EXP *k	$r_1 \leftarrow r_{r_k}{}^r 1$
DEC *k*	$r_k \leftarrow r_k - 1$		
DEC **k*	$r_{r_k} \leftarrow r_{r_k} - 1$	FCHECK *k*	$r_1 \leftarrow 1 \parallel 0$
		FCHECK **k*	$r_1 \leftarrow 1 \parallel 0$
DIV =*i*	$r_1 \leftarrow \lfloor r_1 / i \rfloor$		
DIV *k*	$r_1 \leftarrow \lfloor r_1 / r_k \rfloor$	FUTURE *n*	$pm_{r1} \leftarrow$future(sm_n)
DIV **k*	$r_1 \leftarrow \lfloor r_1 / r_{r_k} \rfloor$	FUTURE **k*	$pm_{r1} \leftarrow$future(sm_{r_k})

FWRITE *k	$sm_{r1} \leftarrow r_k$
HALT	PC←C (i.e.halt)
INC k	$r_k \leftarrow r_k + 1$
INC *k	$r_{r_k} \leftarrow r_{r_k} + 1$
JPOS *label*	$r_1 > 0 \Rightarrow$ PC-&*label*
JPOS k	$r_1 > 0 \Rightarrow$ PC- r_k
JPOS *k	$r_1 > 0 \Rightarrow$ PC- r_{r_k}
JUMP *label*	PC ←&*label*
JUMP k	PC ←r_k
JUMP *k	PC ←r_{r_k}
JZERO *label*	$r_1 = 0 \Rightarrow$ PC-&*label*
JZERO k	$r_1 = 0 \Rightarrow$ PC- r_k
JZERO *k	$r_1 = 0 \Rightarrow$ PC- r_{r_k}
LOAD =i	$r_1 \leftarrow i$
LOAD k	$r_1 \leftarrow r_k$
LOAD *k	$r_1 \leftarrow r_{r_k}$
LOADINDEX	$r_1 \leftarrow$SIG
LOADPC	$r_1 \leftarrow$PC
LOG =i	$r_1 \leftarrow \lfloor \log i \rfloor$
LOG k	$r_1 \leftarrow \lfloor \log r_k \rfloor$
LOG *k	$r_1 \leftarrow \lfloor \log r_{r_k} \rfloor$
LOGB =i	$r_1 \leftarrow \lfloor \log_i r_1 \rfloor$
LOGB k	$r_1 \leftarrow \lfloor \log_{r_k} r_1 \rfloor$
LOGB *k	$r_1 \leftarrow \lfloor \log_{r_{r_k}} r_1 \rfloor$
MOD =i	$r_1 \leftarrow r_1 - i * \lfloor r_1 / i \rfloor$
MOD k	$r_1 \leftarrow r_1 - r_k * \lfloor r_1 / r_k \rfloor$
MOD *k	$r_1 \leftarrow r_1 - r_{r_k} * \lfloor r_1 / r_{r_k} \rfloor$
MUL =i	$r_1 \leftarrow r_1 *$ i
MUL k	$r_1 \leftarrow r_1 * r_k$
MUL *k	$r_1 \leftarrow r_1 * r_{r_k}$
NOT =i	$r_1 \leftarrow$not i
NOT k	$r_1 \leftarrow$not r_k
NOT *k	$r_1 \leftarrow$not r_{r_k}
OR =i	$r_1 \leftarrow r_1$ or i
OR k	$r_1 \leftarrow r_1$ or r_k
OR *k	$r_1 \leftarrow r_1$ or r_{r_k}
READ n	$r_1 \leftarrow pm_n$
READ *k	$r_1 \leftarrow pm_{r_k}$
SHIFT =i	$r_1 \leftarrow r_1 * 2^i$
SHIFT k	$r_1 \leftarrow r_1 * 2^{r_k}$
SHIFT *k	$r_1 \leftarrow r_1 * 2^{r_{r_k}}$
STORE k	$r_k \leftarrow r_1$
STORE *k	$r_{r_k} \leftarrow r_1$
SUB =i	$r_1 \leftarrow r_1 - i$
SUB k	$r_1 \leftarrow r_1 - r_k$
SUB *k	$r_1 \leftarrow r_1 - r_{r_k}$
SYNCHRONIZE	synchronization
WRITE n	$pm_n \leftarrow r_1$
WRITE *k	$pm_{r_k} \leftarrow r_1$
XOR =i	$r_1 \leftarrow r_1$ xor i
XOR k	$r_1 \leftarrow r_1$ xor r_k
XOR *k	$r_1 \leftarrow r_1$ xor r_r

Next some explanations about the notation of instructions:

i	a signed integer
k	a register
n	a memory index
pm	a private memory
sm	a shared memory
label	a label string
&*label*	the index of the instruction labelled with *label*
C	the number of instructions in the program
r_x	refers to register x
SIG	refers to the signature register
PC	refers to the program counter
m_x	refers to memory cell x
x ←y	the value of the expression y is evaluated and then set as a value of x
x ↵ y	the value of expression y is evaluated and set as a value y, but not immediately. The value of y is first requested from shared memory or it is sent to shared memory and therefore goes through the interconnection network.

For some instructions the exactness of the definitions has 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. Normally the exact behaviour can be found from file *limits.h*.

4. USAGE OF THE ASSEMBLER

The FASM assembler program converts a text file to pre-processed and easily interpretable binary file. A text file can be written with some ordinary text editor.

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

fasm *Source Object*

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

Usage: fasm <source> <object>

is given if an incorrect number of arguments are given to the fasm 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 chapter 13.

5. USAGE OF THE EMULATOR

When an FASM program has been assembled with the FASM assembler, the created assembled program file can be executed with the emulator. With command line options user can change the system parameters. With these parameters, the user can modify the settings of the emulator for his/hers own needs.

5.1. Principle of functioning

The basic principle of functioning is simple. Every processor in turns executes an instruction. This processing cycle will be continued until every processor has completed executing the FASM program currently loaded. After every cycle will also be some memory references serviced. After all processors have halted (executed HALT instruction) their execution, outputs of the processors will be printed out. Algorithm 1 demonstrates this procedure.

Algorithm 1. Functioning of the emulator.

```
Begin
   Initialize emulator
   While not All processors halted
   do
      For All processors do
         Read instruction
         Execute instruction
      End for
      Clock := Clock + 1
      Complete a set of  memory references
   End while
   Print outputs
End
```

5.2. Command line of the emulator

The format of the command to start the emulator is

fpram [options] *ProgramFile*

ProgramFile is the name of the file containing the assembled F-PRAM program to be executed. The command line options can be specified in any order. For example, the command

fpram -pf *Test.init* -i *Test.input* -reg 32 -p 16 *TestProgram.obj*

is valid if files *Test.init*, *Test.input* and *TestProgram.obj* exist.

5.2.1. Command line options

Option: -p RAMs
Description: Number of used processors. Default value of this option is 1.

Option: -reg Regs
Description: Number of general registers. The amount must be at least two. Default value is 16.

Option: -b Value
Description: Bandwidth inefficiency. Value b is the minimum possible time between requested memory references without saturating the network. The width of the memory reference queue is P/b. Default value is five.

Option: -s Value
Description: Synchronization delay. Value s is the length of the delay whenever the synchronization procedure is executed. Default value is 10.

Option: -bp Value
Description: Processor bandwidth inefficiency. Value bp is the time that it takes to issue a shared memory reference. Default value is one.

Option: -l Value
Description: Shared memory reference latency. Value l is the worst expected time that takes to complete a shared memory reference. Default value is 10.

Option: -i Inputfile
Description: Presents the file consisting inputs for processors. Inputs must be in the following form:

#RAMNDX *Input1 Input2*

where RAMNDX is the index of processor and *Input* is the input value for processor RAMNDX. The character ”#” separates the ramindex from other input data.

Option: -pf Parameterfile
Description: Presents the file consisting new parameter values for the emulator. New parameter values must appear in the file in the same form than in the command line.

5.2.2. Command line error messages

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

Message: fpram: invalid option
Description: Option was not legal. After the message a help page of commands is printed. Help page is as follows:

```
Usage:
fpram [options] programfile
Options:
-pf initfile     : Parameter initialization file
-p #n            : #n processors
-reg #n          : #n registers
-b #n            : Bandwidth inefficiency
-l #n            : Shared memory latency
-s #n            : Synchronization delay
-bp #n           : Processor bandwidth inefficiency
-i inputfile     : Input file
Example:
fpram -pf example.init -p 64 -l 20 example.obj
```

6. MODULE REFERENCE

The emulator system is divided into modules that implement some services or parts of the system. Some modules are common for both the emulator program and the assembler program. A program consists of a main module and modules that implement some service or part of the program or that just provides some type and macro definitions. The modules consist of a header file (.h) and a possible source file (.c). The header file contains the declarations of the objects and the definitions of the types and possible some macro definitions. The client module that uses the services and definitions of the servant module includes the header file of the servant module. The source file contains the definitions of the external objects and the local declarations and definitions of the module.

The following module reference contains a short description of each module, modules specifier and in which files it is used.

6.1. Main modules

Module: fasm.c
Description: This module is the main module for FASM assembler. The module implements the entry and exit points of the assembler.

Module: fpram.c
Description: This module is the main module for the F-PRAM emulator. The module implements the entry and exit points of the emulator. The module controls the execution of FASM program.

6.2. Other modules

Module: asmdefs.h
Specifier: ASM
Description: This module contains some macro and type definitions that are common to the FASM assembler.
Used in files: asmerr.h, labelmgr.h, scanner.h, symtbl.h, asmerr.c, labelmgr.c, parser.c, scanner.c, symtbl.c

Module: asmerr.h, asmerr.c
Specifier: AER
Description: This module implements the FASM assembler error reporting system.
Used in files: codemgr.c, fasm.c, labelmgr.c, parser.c, scanner.c

Module: avltree.h, avltree.c
Specifier: AVL
Description: This module implements the generic AVL-search-tree. Operations implemented are create tree, insert node, search node, remove all nodes and delete tree. AVL-tree is used to implement the symbol table of the FASM assembler.
Used in files: symtbl.c

Module: cmdfuncs.h, cmdfuncs.c
Specifier: CMD
Description: This module contains the functions that implement the instructions of the emulator.
Used in files: rammgr.c

Module: codemgr.h, codemgr.c
Specifier: CDM
Description: This module implements the code generation functions of the FASM assembler.
Used in files: fasm.c, labelmgr.c, parser.c

Module: dirmgr.h, dirmgr.c
Specifier: DIR
Description: This module implements the directive handler for the run-time directives of the FASM language.
Used in files: fpram.c, ramset.c

Module: error.h, error.c
Specifier: ERR
Description: This module implements the general error reporting system. The system is used to report about the general errors that happen during a program execution, about the errors that are detected during the execution of an FASM program in the emulator, and about the internal errors that happen during a program execution.
Used in files: asmerr.c, avltree.c, cmdfuncs.c, codemgr.c, debug.c, filemgr.c, fpram.c, genstack.c, instrmgr.c, labelmgr.c, memmgr.c, parser.c, pgrmgr.c, rammgr.c, ramset.c, reffuncs.c, scanner.c, symtbl.c

Module: extmgr.h, extmgr.c
Specifier: EXT
Description: This module implements the external variables used by the emulator.
Used in files: cmdfuncs.c, fpram.c, pmemmgr.c, ramset.c, smemmgr.c

Module: filemgr.h, filemgr.c
Specifier: FLM
Description: This module implements the general file handling functions used in all file handling in the system.
Used in files: codemgr.c, pgmmgr.c, scanner.c

Module: genstack.h, genstack.c
Specifier: GST
Description: This module implements a generic stack. Stacks are used while solving the label references during the assembling of an FASM program.
Used in files: asmdefs.h, labelmgr.c

Module: instrmgr.h, instrmgr.c
Specifier: INS
Description: This module implements some type and macro definitions for instruction handling and a function for converting the binary representation of an instruction to a human readable form. Note that the file instrmgr.c is included only in the debugging purpose.
Used in files: pgmmgr.h, codemgr.c, pgmmgr.c, rammgr.c, ramset.c

Module: iomgr.h, iomgr.c
Specifier: IOM
Description: This module implements the input/output queues of the processors.
Used in files: dirmgr.h, ramset.h, cmdfuncs.c, dirmgr.c, fpram.c, ramset.c,

Module: lblmgr.h, lblmgr.c
Specifier: LBL
Description: This module implements the functions for solving the label references during the assembling of an FASM program.
Used in files: fasm.c, parser.c,

Module: memmgr.h, memmgr.c
Specifier: MEM
Description: This module implements the general memory allocation and deallocation functions. All dynamic memory allocation is done through this module.
Used in files: avltree.c, codemgr.c, filemgr.c, genstack.c, iomgr.c, pgmmgr.c, pmemmgr.c, rammgr.c, ramset.c, scanner.c smemmgr.c

Module: parser.h, parser.c
Specifier: PAR
Description: This module implements the lexical parser of the FASM assembler.
Used in files: fasm.c

Module: pedefs.h
Specifier: PE
Description: This module contains the general definitions of the F-PRAM emulator system.
Used in files: asmdefs.h, codemgr.h, instrmgr.h, asmerr.c, avltree.c, cmdfuncs.c, codemgr.c, dirmgr.c, extmgr.c, fasm.c, filemgr.c, fpram.c, genstack.c, instrmgr.c, labelmgr.c, memmgr.c, parser.c, pgmmgr.c, pmemmgr.c, rammgr.c, ramset.c, reffuncs.c, scanner.c, smemmgr.c, symtbl.c

Module: pgmmgr.h, pgmmgr.c
Specifier: PGM
Description: This module contains the functions for loading and deleting of assembled FASM program code and a macro for fetching an instruction from the loaded program code.
Used in files: extmgr.h, rammgr.h, ramset.h, cmdfuncs.c, extmgr.c, fpram.c, rammgr.c, ramset.c

Module: pmemmgr.h, pmemmgr.c
Specifier: PMM
Description: This module implements the private memories of processors.
Used in files: rammgr.h, cmdfuncs.c, rammgr.c, ramset.c, reffuncs.c

Module: queuemgr.h, queuemgr.c
Specifier: QM
Description: This module contains the functions for calculating the dimensions of the queue that implements the interconnecting network.
Used in files: fpram.c, smemmgr.c

Module: rammgr.h, rammgr.c
Specifier: RAM
Description: This module contains the functions for handling of one virtual processor.
Used in files: cmdfuncs.h, extmgr.h, ramset.h, reffuncs.h, cmdfuncs.c, extmgr.c, fpram.c, ramset.c, reffuncs.c

Module: ramset.h , ramset.c
Specifier: RMS
Description: This module implements the functions for handling a set of RAMs.
Used in files: dirmgr.h, extmgr.h, dirmgr.c, extmgr.c, fpram.c

Module: reffuncs.h, reffuncs.c
Specifier: REF
Description: This module contains the functions that implement the different reference modes of the F-PRAM instructions.
Used in files: rammgr.c, ramset.c

Module: scanner.h, scanner.c
Specifier: SCN
Description: This module implements the lexical scanner of the FASM assembler.
Used in files: asmerr.c, fasm.c, parser.c

Module: smemmgr.h, smemmgr.c
Specifier: SMM
Description: This module implements the shared memory and the interconnection network of the F-PRAM.
Used in files: dirmgr.c, extmgr.c, fpram.c, ramset.c, reffuncs.c

Module: symtbl.h, symtbl.c
Specifier: SYM
Description: This module implements the symbol table of the FASM assembler.
Used in files: fasm.c, parser.c, scanner.c

Module: debug.h, debug.c
Specifier: DBG
Description: This module implements some debugging aid functions. This module is normally not used. The debug function calls can be activated by defining the macro debug when compiling the module that needs debugging. Also this module must be compiled with the macro debug defined.
Used in files: asmerr.c, avltree.c, cmdfuncs.c, codemgr.c, dirmgr.c, error.c, extmgr.c, filemgr.c, genstack.c, instrmgr.c, iomgr.c, labelmgr.c, memmgr.c, parser.c, pgmmgr.c, pmemmgr.c, rammgr.c, rammgr.c, ramset.c, reffuncs.c, scanner.c, smemmgr.c, symtbl.c

7. GENERAL PROGRAMMING STYLE AND TECHNIQUE ISSUES

This chapter presents some general programming style and technical issues concerning the emulator system. These issues give a general guide for reading the source code of the emulator system.

7.1. Naming of the identifiers

The names of the identifiers are supposed to be illustrative. Names are formed using mixed-case style. Identifiers are normally abbreviations. Each word in an identifier starts with an upper-case letter and the rest of the word is written with lowercase letters. External identifiers. i.e. the names of the objects that are supposed to be visible outside the module, are prefixed with the module specifier followed by an underscore '_'. E.g. a macro found from file rammgr.h is RAM_GetNumOfRegs and it returns number of registers in a RAM.

7.2. Header files

Header files contain the declaration of the objects and definition of the types and macros that are globally visible. If an identifier that is meant to be globally visible is referred in a file, then the header file that declares the identifier is included in the file. The same practice applies to the standard C library headers.

To suppress multiple inclusions of header files, each header file defines a macro

InclName

when the header file is included the first time. Here Name is the upper-case file name of the module without an extension. The definition status of the macro is checked in the header file. If the macro is defined, then the content of the header file is skipped. This way the order in which header files are included is not important because a header file is included in every file that uses the identifiers declared in the header file. And one does not have to think whether a header file has already been included or not.

7.3. Data structures

In principle, the members of an object of a complex data type are not accessed directly outside the module that defines the data type. Functions and macros provide the access to these structures. C language does not provide a natural way to hide the details of a data type. Therefore globally visible types are defined completely in header files, but it is meant that only the names of the types are visible outside the module. It is a matter of programming discipline whether this "visibility

rule" is obeyed. Inside the module that defines the data type, the members of an object of that type are often accessed directly.

The token descriptor structure defined in header asmdefs.h is an exception to the above practice. The type is quite complex and the number of macros for accessing different members of the structures and unions would have been large if they were implemented. The naming of the macros would have been a problem. Furthermore, it is not very probable that the data type would change. The members of the structures and unions of the type are accessed directly in the FASM assembler program.

7.4. Function headers

The prototypes of the functions in the function declarations are copied one-to-one from the function definitions. The prototypes of external functions are gathered to the header file of the module. The prototypes of internal functions of a module (functions with static specifier) are gathered to the beginning of the source file.

7.5. Run-time error handling

Run-time errors can happen during dynamic memory allocation and file handling, as well as during the execution of the emulator due to a semantically invalid FASM program or input.

When a run-time error is detected, it is reported immediately, i.e. the function, where the error occurred, calls the error reporting function. Each function in which a run-time error might happen returns an integer status value that indicates whether error occurred -status is fail or false or not status is ok or true. Each function that might call a function that returns a status value, returns a status value. When a function receives a fail return status, it tries to return as soon as possible and return the fail status to the caller. This means that when a run-time error occurs, it is reported immediately and the program terminates. An error does not raise other errors because the execution status is set to fail when the error is detected.

The source code contains some checking for situations that should not happen. Usually these verifications are parameter validity checks. If such a check fails, then there is a bug in the program and the internal error reporting function is called immediately. This function reports about internal error and calls the exit function.

7.6. Debugging

Most of the functions contain some debugging statements that are not compiled by default but that can be compiled along by defining the macro *debug*. If a function is compiled with the macro debug defined, then calls to the function that reports about entry to the function are included to the

beginning of the function and calls to functions that report about the exit from the function are included to the end of the function. These functions write the name of the function and possibly some parameter and return value information to the standard output stream. The function call level is denoted with indention and direction of the flow of control - entry or exit - with arrow symbols.

The module that is to be debugged with the help of the debugging statements must be compiled with the macro debug defined. Also the file “debug.c” must be compiled with the macro debug defined and linked along with the other object files that form the program. Normally the debug module is not needed.

8. STRUCTURE OF THE FASM ASSEMBLER PROGRAM

The purpose of the FASM assembler is to convert a text form FASM program to an easily interpretable binary form. The assembler checks the syntax and static semantics of the FASM program and, what is most important, solves the label references. If a text form FASM program was interpreted directly, then the determining of the jump addresses would be quite inefficient. The solving of the label references is the most complex part of the assembling procedure, because the assembling is done with one pass through the FASM source. Otherwise, the assembling is quite straightforward.

8.1. General structure of the FASM assembler

The FASM assembler is a quite ordinary one pass compiler that uses a recursively descending parsing technique. The assembler consists of the usual parts of a compiler: lexical scanner, parser, code manager, and symbol table manager. In addition there is label manager which provides functions that are used in solving the label references. These parts are implemented in separate modules.

The lexical scanner reads the FASM source file character by character and tries to form tokens from the characters. The symbol table contains the reserved words of the FASM language. New label identifiers are added to the symbol table when they are met the first time. When the scanner has extracted an identifier token, it checks from the symbol table, whether the identifier is concluded to be a new label identifier and is added to the symbol table. The parser checks whether the tokens it gets from the scanner, form syntactically and semantically valid program structure, and if so, calls the functions of the code manager to generate instructions to the binary program file that is the output of the assembler.

8.2. Solving of the label references

If the parser meets a label reference as a parameter of a jump instruction before the definition of the label, then the file position of the instruction is pushed on a stack. In addition, a pointer to the label is pushed to a stack that holds the pointers of all the labels whose definition is pending. This stack is used to check at the end of a code that all the labels are defined.

For the jump instruction the assembler generates code that has an undefined label as a parameter. The value of the argument (the address of the instruction where to jump) is set to zero. When the definition of the label is later met, the argument values of the undefined jump instructions are updated. The descriptor of the label in the symbol table holds the stack that contains the file positions of the pending instructions. The positions are popped one at the time. For each position the file position pointer is moved to the position and the argument value of the jump instruction is updated.

8.3. Directive assignment

The directives @CLOCK and @LASTMEMINDEX are handled intuitively in the following way: when control flows through a directive it is taken into account. In practice, the directives are part of the instructions and are assigned as pre-instruction or post-instruction directives, i.e. are handled either prior to the execution of the instruction or after the execution of the instruction. The exact rules, when directive is assigned as pre-instruction and when as post-instruction directives, are presented in the following.

If a directive immediately follows an F-PRAM code instruction, then the directive is assigned as a post-instruction directive to the instruction. If a directive is between a label (or labels) preceding an F-PRAM code instruction, then the directive is assigned as a pre-instruction directive to the instruction. If a directive is before the first instruction of an F-PRAM code block and the first instruction is not preceded with labels, then the directive is assigned as a pre-instruction directive; if the first instruction is preceded with a label, then the directive is not assigned to anywhere (it has no effect).

9. STRUCTURE OF THE EMULATOR PROGRAM

The emulator program implements the actual F-PRAM emulator. The emulator reads, as an input, a file containing an assembled FASM program and executes the instructions in the file.

The emulator is basically a simple interpreter that fulfills the instructions in the FASM program file. Different parts of the emulator have been separated to modules. At the lowest level there are modules that implement the parts of the F-PRAM machine: the shared and the private memories, the interconnection network, the RAM manager, the input/output handling and the reference modes of the instructions. The RAM-set module implements an F-PRAM using the RAM modules and the module implementing the directives.

9.1. The implementation of the shared memory

The shared memory is implemented as a set of blocks of shared memory. One shared memory block consists of 1024 contiguous shared memory cells. A pointer to a block is saved into a vector containing pointers to all of the blocks. A block pointer is added to the vector and a necessary amount of memory is allocated when to some undefined memory block is referred during the execution of the FASM program.

When a shared memory cell is referred to, a segment and offset addresses are calculated from the cell address with the usual divide and reminder operations. The segment address is the address of the memory block in the vector and the offset address is the address of the cell within the block. A pointer to the cell is returned to the caller.

9.2. The implementation of the interconnection network

The interconnection network is often the most ineffective part of a parallel computer. However, it is indispensable part of it, because a parallel machine communicates through the shared memory which is reached through the interconnection network. When a processor wants to write to the shared memory or read from it, it sends a memory reference to the network. The speed of the network is measured with a latency L. It is the worst expected time that it takes to complete a memory reference in an empty (non-full) network.

The network is implemented as a queue of blocks of memory references, queue implemented using a ring. Each block has a finite amount of places. The block where a new memory reference is put is selected using a random function that returns values within an acceptable range. In each clock cycle, a set of memory references are handled, and the rest are advanced in the future.

9.3. The implementation of the synchronization network

The idea of a synchronization is to get asynchronously working processors to the same phase of the FASM program execution. Synchronization is sometimes necessary to verify that all the information in the shared memory is up-to-date. The effectiveness of a synchronization network is described by the value of synchronization delay. It is the time starting from the moment all processors have executed the synchronization instruction and ending when the execution of the program has started again.

The synchronization network has not been implemented as a separate data structure. The functions of this network have been combined to the normal execution of the instructions. Synchronization is started when the first processor reaches the synchronize-instruction. During synchronization all pending memory references will also be completed.

9.4. The implementation of the input/output handling

Each processor has an input and an output queue. When processor needs input from the user, it first checks its input queue. If the queue is empty, it reads the input from stdin or from an input file. The input must be given in form

#RAMNDX $Input_1$ $Input_2$ ...

where RAMNDX is the index of the processor and $Input_i$ is the value of the input. Input is read from line to line until current processors input is found. All inputs are stored to the i/o queue that RAMNDX orders.

Normally a program outputs the results of its calculations. When the processor reaches the instruction that outputs a value, it stores the output value to its output queue. At the end of the execution of the program, all output queues are printed out to the stdout.

9.5. The implementation of the directive manager

The directives are handled through the directive manager module. The module is very simple. The directive handling functions get as their parameters a pointer to the descriptor of the RAM-set (F-PRAM).

9.6. One RAM

The RAM manager implements a single RAM of an F-PRAM. A descriptor of a RAM consists of a member containing a pointer to an array containing the registers of the RAM, a member

containing the number of registers of the RAM, a member containing the index of the RAM, a member pointing to the private memory of the RAM and a member containing a pointer to the program code.

Within the RAM manager, pointers to the functions implementing the instructions of a RAM and to the functions implementing the reference modes of the instructions are gathered to two arrays: command function array (instruction consists of a command and an argument) and reference function array. The coding of the instructions can be easily used to index the command function and reference function arrays and thus the calling of the right function is easy.

9.7. Command functions

The command functions implement the instructions of the RAMs of the F-PRAM. Pointers to the command functions are gathered to an array in the RAM manager. The command functions are called by indexing the table by a part of the descriptor of an instruction.

The command functions get as their parameter a pointer to the RAM descriptor of the RAM that executes the instruction and a pointer to the actual argument of the instruction, i.e. to some memory cell or to some register of the RAM - or NULL if there is no argument.

9.8. The implementation of the reference functions

Reference functions implement the different reference modes of the instructions. In practice, a reference function is used to get a pointer to the actual argument of an instruction. Pointers to the reference functions are gathered to an array in RAM manager and the reference functions are called through that table using a part of the instruction as an index.

There are eight different reference functions. “No argument” function is used as a dummy reference function with instruction HALT. “Immediate reference” function is used with instructions that have an immediate argument (including a jump to a label). “Direct register reference” function is used with the instructions that have an argument that refers directly to a register. “Indirect register reference” function is used with instructions that have an argument that refers indirectly to a register. “Direct memory reference” function is used with instructions that have an argument that refers directly to a private memory cell. “Indirect memory reference” function is used with instructions that have an argument that refers indirectly (via a register) to a private memory cell. “Direct shared memory reference” function is used with instructions that have an argument that refers directly to a shared memory cell. “Indirect shared memory reference” function is used with instructions that have an argument that refers indirectly to a shared memory cell. “Direct PC reference” function, which may seem odd, is used with LOADPC instruction. The LOADPC instruction is implemented as reverse assignment, as write of the argument value to the accumulator. In the same way, “Direct SIG reference” function is used with LOADINDEX instruction.

The reference functions get as parameters a pointer to the descriptor of the RAM that executes the instruction and a pointer to the argument value member of the descriptor of the instruction. They return via a parameter a pointer to the actual argument.

10. ASSEMBLED FASM FILE

The FASM assembler converts a text file containing a valid F-PRAM program to an assembled F-PRAM program file that can be given as an input to the emulator for execution. The assembled F-PRAM program file is a binary file and the structure of the file is described here.

A program segment has the format

Num $Instr_1$ $Instr_2$ ... $Instr_{Num}$

where Num is a long int containing the number of instructions in the segment and *Instr*'s are the instructions. If Num has value 0L, then there are no *Instr*'s.

Each instruction is of structure type INS_InstrWordDescr (defined in file instrmgr.h):

```
typedef struct {
        int         DirCmdRef;
        long        Arg;
} INS_InstrWordDescr;
```

The member DirCmdRef contains information about command (instruction consists of a command and a possible argument), the reference mode of the instruction, and the possible directives. The member Arg holds the argument value of the instruction. The bit positions 0-4 of the member DirCmdRef are reserved for the command, the bit positions 5-8 for the reference mode, and the bit positions 9-15 for the directives.

The commands are coded in bit positions 0-4 of the member DirCmdRef as follows (in hexadecimal and binary):

ADD	0x00 = 0000 0000
AND	0x01 = 0000 0001
DIV	0x02 = 0000 0010
ECALL	0x03 = 0000 0011
HALT	0x04 = 0000 0100
JPOS	0x05 = 0000 0101
JUMP	0x06 = 0000 0110
JZERO	0x07 = 0000 0111
LOAD	0x08 = 0000 1000
LOADNDX	0x09 = 0000 1001
LOADPC	0x0A = 0000 1010
LOG	0x0B = 0000 1011
MOD	0x0C = 0000 1100
MUL	0x0D = 0000 1101
NOT	0x0E = 0000 1110
OR	0x0F = 0000 1111

READ	0x10 = 0001 0000
SHIFT	0x11 = 0001 0001
STORE	0x12 = 0001 0010
SUB	0x13 = 0001 0011
WRITE	0x14 = 0001 0100
XOR	0x15 = 0001 0101
FUTURE	0x16 = 0001 0110
FWRITE	0x17 = 0001 0111
FCHECK	0x18 = 0001 1000
BFUTURE	0x19 = 0001 1001 (not implemented)
BWRITE	0x1A = 0001 1010 (not implemented)
LOGB	0x1B = 0001 1011
EXP	0x1C = 0001 1100
INC	0x1D = 0001 1101
DEC	0x1E = 0001 1110
SYNCHRONIZE	0x1F = 0001 1111

The reference modes of an instruction are coded in bit positions 5-8 of the member DirCmdRef as follows (in hexadecimal and binary):

no argument (e.g. HALT)	0x00 = 0000 0000 0000
immediate (e.g. MUL = -1)	0x20 = 0000 0010 0000
direct register reference (e.g. ADD 1)	0x40 = 0000 0100 0000
indirect register reference (e.g. OR *2)	0x60 = 0000 0110 0000
direct memory reference (e.g. READ 5)	0x80 = 0000 1000 0000
indirect memory reference	0xA0 = 0000 1010 0000
direct PC reference (LOADPC)	0xC0 = 0000 1100 0000
direct SIG reference (LOADINDEX)	0xE0 = 0000 1110 0000
direct shared memory read reference	0x100 = 0001 0000 0000
indirect shared memory read reference	0x120 = 0001 0010 0000
direct shared memory write reference	0x140 = 0001 0100 0000
direct shared memory write reference	0x160 = 0001 0110 0000

The pre-instruction directives assigned to an instruction are coded in bit positions 9-12 and the post-instruction directives assigned to an instruction are coded in bit positions 13-15 of the member DirCmdRef as follows (in hexadecimal and binary):

precommand @CLOCK	0x0200 = 0000 0010 0000 0000
precommand @LASTMEMINDEX	0x0800 = 0000 1000 0000 0000
postcommand @CLOCK	0x2000 = 0010 0000 0000 0000
postcommand @LASTMEMINDEX	0x8000 = 1000 0000 0000 0000

11. CHANGING THE EMULATOR SYSTEM

The emulator system is based on the parallel computation model F-PRAM. So the possibilities to change the system are limited. On the other hand, all larger entities are divided into different modules, so if it is necessary to change the implementation of certain operations, it is possible as long as the necessary services are provided into other modules.

11.1. Changing constant values

Some of the constant values can be changed. Next follows a description of these changeable features. These values can be found from a file *config.h*.

11.1.1. Memory segment size

Memories (private and shared) consist of memory segments. The size of the segment is by default 1024 (#define SMM_SegSize 1024), but it can be changed to any value larger that zero. Bigger segments may speed up the execution time (of course doesn't affect to clock cycle values returned by emulator) slightly but it also may waste more memory.

11.1.2. Preprocessor

The FASM source file is first passed through the C language preprocessor. Default name of the preprocessor is cpp. If the name differs from that, it must be changed to correct one. Note that after the name of the preprocessor must be one space, e.g. #define PE_PreProsCmd "cpp ".

11.1.3. Future ready - value

Instruction FCHECK checks if the shared memory value requested (with future) earlier has arrived. Instruction FUTURE sets the value of the private memory cell to value defined in *config.h* (#define CMD_CheckValue MAXLONG - 1) and FCHECK returns PE_No if the value in a memory cell is equivalent to MAXLONG-1. This checking value can be changed to any value within type long range.

11.2. Changing instruction set

The easiest way to change instruction set is to replace some of the existing instruction and its function calls with new ones. If this is not possible, also a new instruction can be added. This, however, requires changes to several modules:

cmdfuncs.c ,h	: adding new extern function, header of the command is int CMD_CmdFunc(RAM_RAMDescr *RAM, long *ArgPtr);
instrmgr.h	: checking the bit masks of instructions, directives and
pedefs.h	: adding new command and checking the bit masks of
rammgr.c	: initializing a command function table in function RAM_InitRAMMgr,
symbtbl.c	:initializing a symbol table in function SYM_InitSymTbl.

11.3. Changing random function

The interconnection network is an important part when analysing parallel computers. In the emulator, the network is implemented with a queue of blocks of pending references between the processors and the shared memory (figure 2). A random function shows where to place a new memory reference in the queue. The random function gives a value between 1-L. The reference is inserted into the block that the value returned by the function shows. If the block is already full, the value is inserted into the next block having free place for reference. Reference is placed into the same kind of queue when the value is returning to the processor it requested. Random function actually describes the nature of the interconnection network.

The random function the emulator uses is a Gaussian function by default. It returns a random value where a middle point is the most probable returning value (figure 3). The more far from the middle point the value is, the less probable it is to get it. The middle point is calculated from the parameter L so that the middle point is L / 2.5 . The value of a sigma describes how wide is the range of the values the Gaussian random function returns. The value of the sigma is calculated so that sigma is latency divided with SMM_Sigma. The default value of SMM_Sigma is 5, but it can be changed.

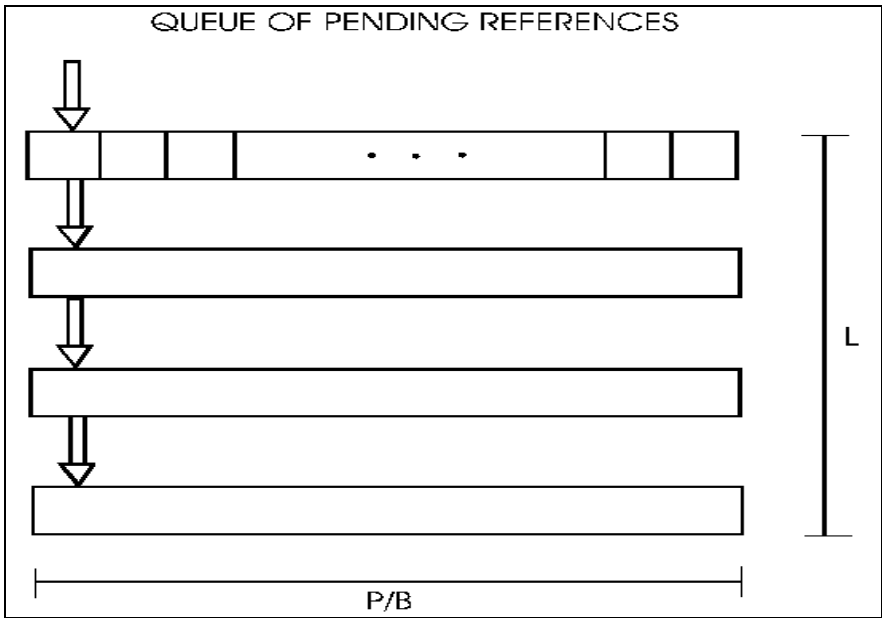

Figure 2. Queue of pending references.

Error! Bookmark not defined.

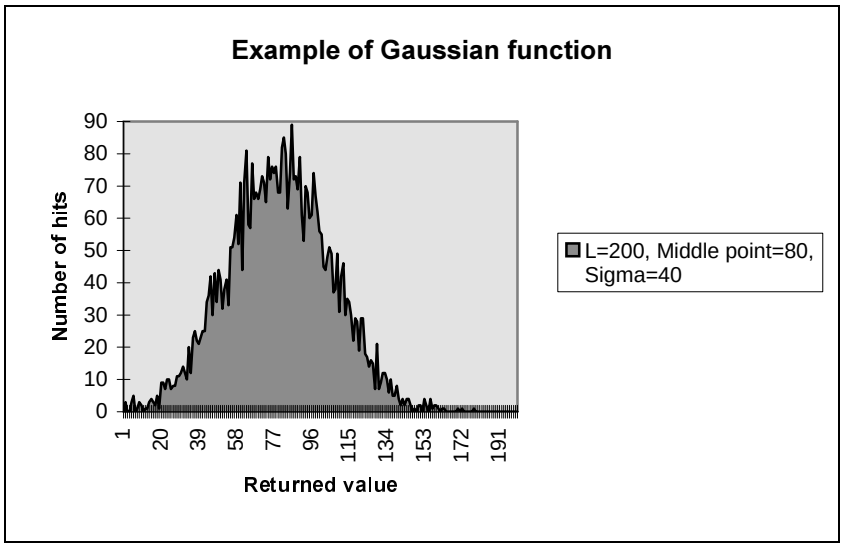

Figure 3. Gaussian function.

The user can also use his/hers own random functions. In a file *smemmgr.h* is defined five slots for random functions. One is used (Gaussian function) but four is unused. The procedure to use own random function is as follows:

i) Write the function to file *smemmgr.c* and its introduction to the beginning of the file (or to some different module and header file included in *smemmgr.c*). The function has certain limitations. Its definition must follow the next formula:

int random_function(long *LowBound*, long *HighBound*, long **ReturnValue*)

where *LowBound* is the smallest possible value, *HighBound* is highest possible value and **ReturnValue* is a pointer to returning value.

ii) Add the name of the function to the *config.h* file into a free random function slot (e.g. #define SMM_RANDOMFUNC1 NameOfUserFunction).

iii) Change the value of SMM_RANDFUNC (in *config.h*) to point the user defined function.

iv) Compile the program again.

12. INSTALLING THE EMULATOR SYSTEM

The F-PRAM emulator system (the FASM 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 FASM assembler and the emulator, is actually all the installation needs.

12.1. System requirements

The F-PRAM emulator system is written in ANSI C programming language. The computer system must provide an ANSI C compiler. The FASM assembler is 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 FASM language. UNIX systems usually include a separate C preprocessor or the C compiler can be used to output the preprocessed file.

12.2. Makefile

An emulator package includes one makefile to compile and link modules in UNIX systems. That can be used with a program make found in most UNIX systems. The syntax of the command is

```
make
```

where make uses the file makefile as a guide to complete the compile- and link procedures. Makefile included compiles and links both the assembler and the emulator programs.

12.3. Compiling

The best way to compile the emulator system is using the makefile, but if this is not possible (system does not contain make program) the source code can also be compiled some other way. The source code for the F-PRAM emulator system is in the following files.

Files that form the FASM assembler are:

asmdefs.h	config.h	intstrmgr.h
asmerr.h	error.h	labelmgr.h
asmerr.c	error.c	labelmgr.c
avltree.h	filemgr.h	memmgr.h
avltree.c	filemgr.c	memmgr.c
codemgr.h	genstack.h	parser.h
codemgr.c	genstack.c	parser.c

pasm.c	scanner.h	symtbl.h
pedefs.h	scanner.c	symtbl.c

Note that the file instrmgr.c is not needed in the FASM assembler, just the instrmgr.h.

Files that form the emulator are:

cmdfuncs.h	fpram.h	pmemmgr.c
cmdfuncs.c	fpram.c	queuemgr.h
config.h	instrmgr.h	queuemgr.c
dirmgr.h	iomgr.h	rammgr.h
dirmgr.c	iomgr.c	rammgr.c
error.h	memmgr.h	ramset.h
error.c	memmgr.c	ramset.c
extmgr.h	pedefs.h	reffuncs.h
extmgr.h	pgmmgr.h	reffuncs.c
filemgr.h	pgmmgr.c	smemmgr.h
filemgr.c	pmemmgr.h	smemmgr.c

Files that are common to both the FASM assembler and the emulator programs are:

config.h	filemgr.h	memmgr.h
error.h	filemgr.c	memmgr.c
error.c	instrmgr.h	pedefs.h

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

All the C files should be compiled with the ANSI C compiler to get the corresponding object files. The compiler may give some warnings but no fatal errors should occur. If a fatal error occurs, then the compiler used is not full ANSI C compiler, or there is something wrong with the installation of the compiler, or the computer system is not suitable.

The F-PRAM emulator system also includes files *debug.h* and *debug.c*. These files include some debugging aids but they are not normally needed. If there is a need to debug the system, the source should be compiled along with these debug files with the symbol “debug” defined.

12.4. Linking

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

The FASM assembler is linked together from object files resulting from the assembler C files. The name of resulting executable can be chosen freely. The author uses the name “fasm”.

The emulator is linked together from object files resulting from the emulator C files. The name of the program can be chosen freely. The author uses the name “fpram”.

13. ERROR MESSAGES

This chapter describes the error messages produced by the assembler and the emulator. Run-time error messages are divided into their own category.

13.1. FASM assembler error messages

The assembler, among other things, checks that the FASM 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.

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 message is given if the source file ends before a syntactically complete program or program structure was 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 correct but it is not allowed in this context. This message is given when a directive or instruction appears in a code 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, i.e. there is no instruction preceded by *Label* (and a :).

Message: Label redefinition: *Label*
Description: Label *Label* is defined twice in the code, i.e. there are 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 is 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 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!)

13.2. Emulator error messages

There exist a few possible run-time error situations. Most of them are checked and reported if error occurs. Arithmetic under flows and overflows are not checked by the emulator. The operating system may or may not trap these errors. These errors cause the emulator to terminate.

Message: Division by zero: SIG:*Sig*, PC:*Pc*
Description: A processor whose signature (SIG) is *Sig* attempted a division by zero.

Message: Error while inserting value to queue
Description: Error occurred while inserting value to i/o queue. Normally happens only when i/o manager fails to allocate memory.

Message: Modulus by zero: SIG:*Sig*, PC:*Pc*
Description: A processor whose SIG is *Sig* attempted a modulus by zero.

Message: Illegal register reference:SIG:*Sig*, PC: *Pc*, Register reference: *Ref*
Description: A reference to a nonexisting register appears as an argument or as part of an argument of an instruction. *Ref* is the invalid register index.

Message: Invalid external function:PC:*Pc*, FUNCTION:*Function*
Description: The argument of an external function is out of range. Valid range is from 0 to 3.

Message: Invalid jump address:SIG: *Sig*, PC: *Pc*
Description: A jump directed outside the code block is attempted. Addr is the invalid jump address.

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.

13.3. General run-time error messages

Both the FASM 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 exist.

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 a 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.

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

Hämäläinen P. 1992: *Pram-emulator, User's Manual*, Report B-1992-2, University of Joensuu, Department of Computer Science, Finland.

JáJá J. 1992: *An Introduction to Parallel Algorithms*, Addison-Wesley Publishing Company, Inc.ISBN 0-201-54856-9.

Juvaste S. 1996: *Reasoning of a Parallel Computation Model*, Licenciate Thesis, University of Joensuu, Department of Computer Science, Finland.

Juvaste S. 1997a: *Programming with a Parametrized Parallel Programming Model*, Report A-1997-4, University of Joensuu, Department of Computer Science, Finland.

Juvaste S. 1997b: *Programming language FPM for F-PRAM*. University of Joensuu, Department of Computer Science, Report B-1997-X, to appear.

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