January 1991

Symbolic Simulator/Debugger for the Systolic/Cellular Array Processor

Janez Funda
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-91-07.

Symbolic Simulator/Debugger for the Systolic/Cellular Array Processor

Abstract

This document describes an implementation of a symbolic simulator/debugger for the Systolic/Cellular Array Processor (SCAP), which is currently being built at Hughes Research Laboratories. The SCAP system is a parallel computer with 256 identical processing elements (PEs) connected using a mesh interconnection network in a 16 x 16 grid. Each PE features a two-bus internal architecture, with seven functional units, and four I/O ports used to communicate with its four neighboring PEs. All functional units operate on 32-bit fixed point data. The reader is refered to [Przytula,88] for a detailed description of SCAP's architecture, data representation format, and machine level operation of the system.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-91-07.

Symbolic Simulator/Debugger For The Systolic/Cellular Array Processor

MS-CIS-91-07
GRASP LAB 252

Janez Funda

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104-6389

January 1991

Symbolic Simulator/Debugger for the Systolic/Cellular Array Processor

Janez Funda

University of Pennsylvania
Department of Computer and Information Science
Philadelphia, PA 19104

January 10, 1991

Contents

1 Introduction 2

2 Description of the System 3

2.1 General 3

2.2 User interface in **sim** 4

2.3 Data file format 7

2.4 Disassembly conventions 8

2.5 Processor display format 11

3 Discussion and Suggested Improvements 13

4 Some Implementational Details 14

A A sample SCAP assembly language program and data file 17

B A sample run of the simulator/debugger 18

1 Introduction

This document describes an implementation of a symbolic simulator/debugger for the Systolic/Cellular Array Processor (SCAP), which is currently being built at Hughes Research Laboratories. The SCAP system is a parallel computer with 256 identical processing elements (PEs) connected using a mesh interconnection network in a 16×16 grid. Each PE features a two-bus internal architecture, with seven functional units, and four I/O ports used to communicate with its four neighboring PEs. All functional units operate on 32-bit fixed point data. The reader is refered to [Przytula,88] for a detailed description of SCAP's architecture, data representation format, and machine level operation of the system.

The construction of a symbolic simulator/debugger for the SCAP system was motivated by the need to run, test and debug some assembly language programs that were being developed in the General Robotics and Active Sensory Perception (GRASP) Laboratory here at the University of Pennsylvania. The Laboratory is expecting a delivery of the actual computer some time this year. However, we felt that availability of a simulator/debugger prior to the actual delivery of the system would increase the utilization of the harware, once installed.

The scope of the simulator/debugger project included (1) an implementation of a simulator, that corresponds as closely as possible to the operation of the actual hardware, (2) a loader module to load user data into memory prior to program execution, and (3) an easy-to-use basic user interface, allowing the user to trace the execution of the program, single-step through the code, examine all relevant portions of the system, and make references to the symbolic information in the source program (*e.g.*, symbolic labels, processor mask names, *etc*). The current implementation meets all aspects of the above objectives. It does not, however, strive for a sophisticated visual interface with the user. Therefore, no graphics front end interaction has been implemented. This would be a useful extension to the system, should the simulator/debugger be used more than currently anticipated.

The SCAP simulator/debugger (hereforth refered to as **sim**) described in this document is designed to operate in conjunction with the corresponding SCAP assembler (**ass**), developed by Hartholtz [Hartholtz,88]. The reader is encouraged to refer to [Hartholz,88] for a description of the SCAP assembly language syntax, facilities, and

features.

Simulating the execution of a given SCAP assembly language program is therefore a two stage process. The program is first assembled using **ass**, producing a file containing non-relocatable absolute object code and the symbol table information [Hartholz,88]. This file plus a file containing the data to be operated on are then passed as arguments to **sim**. The details of the data file format as well as user interaction with the system are described below.

This document is intended as a brief user's guide for **sim**. To illustrate some of **sim**'s features, I have added a couple of appendices at the end of this document, showing an actual interaction with the system. Appendix A shows an assembly language source program and the corresponding data file, while Appendix B contains the transcript of a **sim** session, performing a simulation of the program on the given data. Note that the program has been written to exhibit a wide variety of instructions for testing and demonstrational purposes, and is *not* intended to compute any particular useful function.

2 Description of the System

2.1 General

NAME

sim — SCAP symbolic simulator/debugger

SYNOPSIS

sim [**-w**] *codefile datafile*

DESCRIPTION

sim is a symbolic simulator/debugger for the Systolic/Cellular Array Processor currently being designed and built at Hughes Research Laboratories.

codefile is a binary file containing non-relocatable object code, produced by running **ass** on a SCAP assembly language source program. *codefile* also contains symbol information about the original source program.

datafile is an ascii text file containing the data to be manipulated by the program stored in *codefile*. See Section 2.3 for a detailed description of the *datafile* format.

The following *option* is recognized by **sim**:

-w Enable the reporting of run-time warnings. Most of the warnings, which are suppressed by default, are concerned with alerting the user to the potentially undesirable side effects of inter-processor I/O (see [Przytula,88]).

sim operates by reading *codefile* and *datafile* and initializing its internal data structures corresponding to the Instruction Memory (also refred to as Program Memory) and Data Memory accordingly. Information about the symbols defined in the original source program is also obtained from *codefile* and appropriate internal symbol tables are constructed. **sim** then enters a *prompt-execute* loop where the user is repeatedly prompted for commands, which are in turn parsed, interpreted, and executed by the simulator. Various self-explanatory error and warning diagnostics at both the command and execution levels are issued when appropriate.

2.2 User interface in sim

Upon invoking **sim**, the simulator displays a message indicating that loading has been completed and prints the size (in machine instructions) of the loaded program. It then displays the prompt '**sim>**' and awaits a user command. The following commands are recognized by **sim**:

? or **help**

Print a summary of all available commands.

run or **cont**

Run the program or resume execution after a breakpoint. Execution continues until one of the following conditions becomes true:

- a fatal run-time error has occured in one of the active processors (*e.g.*, multiplication overflow)
- a breakpoint has been encountered, or
- the end of the program has been reached.

step [*j*]

Execute the next j instructions. If the specification of j is absent, a step of one instruction is assumed. Termination conditions are equivalent to the ones in the case of **run** or **cont** above.

pc

Show the contents of the program counter (pc). The program counter is displayed in hexadecimal, octal, and decimal formats.

readaddr

Show the contents of the read-address counter. This value will be used as the source address during the next memory read (*i.e.*, shift north/south with memory read).

writeaddr

Show the contents of write-address counter. This value will be used as the destination address during the next memory write (*i.e.*, shift north/south with memory write).

pfifo

Show the contents of the Program Queue (pfifo). The list of Program Memory addresses comprising the queue are displayed along with the status of the queue (empty/full/ok).

wfifo

Show the contents of the Write Queue (wfifo). The list of Data Memory addresses comprising the queue are displayed along with the status of the queue (empty/full/ok).

rfifo

Show the contents of the Read Queue (rfifo). The list of Data Memory addresses comprising the queue are displayed along with the status of the queue (empty/full/ok).

proc i j [f]

Show the contents of the processor in row i and column j. The default format is to display the contents of the processor's registers and ports as real values (format = %19.16f). If the -f option is specified, then the registers and ports are printed as the hexadecimal equivalent of the 32 bit fixed point internal representation of data (format = %8x). The first format lends itself to a more natural inspection by the user, but contains small roundoff errors incurred by the conversion. Conversely, the hexadecimal format corresponds exactly to the internal data representation, but is difficult to interpret. See Section 2.5 for a detailed description of the processor display format.

proglen

Display the length of the program. The length is given in the number of machine instructions and is printed in hexadecimal, octal, and decimal format.

memory i [j] [f]

Display rows i through j of Data Memory. If the specification of j is absent, only the row i is displayed. Each row is preceded by a decimal address of the row. The default format is to print the 16 elements of the row as real values (format = %13.10f), 4 per line. If the -f option is specified, the elements are printed as hexadecimal equivalents of the internal fixed point data representation (format = %8x), 8 per line. In either format, printing of identical rows is suppressed.

instr i [j]

Display rows i through j of Program Memory. If the specification of j is absent, only the instruction at address i is displayed. Instructions are disassembled and printed in symbolic notation, closely resembling that of the assembly language syntax. Some departures were necessary because certain assembly language instructions translate into more than one machine instruction (*e.g.*, inter-processor I/O instructions). See Section 2.4 for more detail.

mask *label*

Display the mask with the symbolic name *label.* A simple graphical representation of the processor array is printed and the enabled processors are

marked with "x".

sysfld *code*

Decode the hexadecimal system field code *code*. The names of the eight system signals are given and the asserted signals are marked with "X".

dump

Produce a system dump in file "core.dump". Data Memory, Program Memory, Read, Write, and Program Queues (fifo's), as well as the contents of all 256 processors are written out to a text file for later inspection. Default printing formats are used for the Data and Program Memory.

break [*label*]

Set a breakpoint at the symbolic label *label*. Only one breakpoint can be set at a given label. A total of 20 breakpoints can be active in the system at any given time. If no *label* argument is given, all currently active breakpoints are listed.

unbreak *label*

Remove a breakpoint at the symbolic label *label*.

exit or **quit**

Exit the simulator.

2.3 Data file format

All data in the SCAP system is organized into named data queues of fixed size. Each queue is composed of a non-zero number of rows of data. The system distinguishes between three different kinds of data queues – *ascending*, *descending*, and *constant* queues [Przytula,88]. Ascending queues grow towards higher Data Memory addresses, descending queues grow toward lower Data Memory addresses, and constant queues are comprised of a single row of data.

The simulator obtains the information about the sizes and types of all data queues defined in the original source program from the object code symbol tables. However,

in order for the loader (*i.e.*, the first stage of the simulator) to be able to correctly load data into Data Memory, the user must list the data queues in the data file (*datafile*) in the *same order* in which the corresponding type/size definitions appear in the original source program. Each queue must be listed as a sequence of *real values* in the range $(-2.0, +2.0)$ and it is the user's responsibility to ensure that the sizes of the given data queues agree with those specified in the queue definitions in the source program. All queues should be listed from head to tail, regardless of the type of the queue. The loader phase of the simulator will account for a particular type of a data queue and load the data in the correct memory locations.

In the absence of any information about the exact placement of the queues in the Data Memory (only the sizes of the queues are known), the simulator assumes that data is to be loaded into Data Memory starting at address 0. Therefore, all input data is loaded into the top portion of the memory and extends as far down (towards higher addresses) as necessary. This convention may change in response to a possible future change in the way data queues are declared in the source program — if the syntax of **ass** is modified to allow the programmer to explicitly specify where in memory particular queues should be located, then the simulator's loader (as well as the actual loader) can be modified to account for that.

2.4 Disassembly conventions

In this section I will briefly present the mnemonics used in the symbolic disassembly, employed by **sim** to display instructions stored in the Program Memory. A typical disassembled instruction appears as follows:

```
                              external (ph1/ph2)    internal (ph1/ph2)
 pc  sel   mask    sysfld  i/o     src    dst     i/o     src    dst
-------------------------------------------------------------------
0024 R/C 0F0FF0F0  00FF    NOP     CSUM2A NOP     NOP     CSUM2A NOP
                           NOP     B7     ADD2    NOP     B7     ADD2
```

where the first four fields (from left to right) correspond to

1. the decimal value of the program counter associated with the instruction,
2. mask type indicator (R/C = row/column; D = diagonal),

3. hexadecimal encoding of the mask (see [Hartholz,88]) — use **mask** command (Section 2.2) to decode,

4. hexadecimal encoding of the system field bits (see [Przytula,88]) — use **sysfld** command (Section 2.2) to decode

The remaining fields encode both phases of the actual instruction. The top line corresponds to Phase 1 and the second line to Phase 2 of the instruction cycle. Moreover (as is suggested by the header), the first three of these fields refer to the external processors (column 1 of the processor array), whereas the second triple of fields refers to the internal processors (columns 2 through 16).

As mentioned above, the mnemonics used in disassembly have been chosen to correspond as closely as possible to the ones used in the assembly language. An important exception are the mnemonics for inter-processor I/O, because a single line of assembly language code expands into two or three (depending on the type of I/O) machine level instructions (see [Przytula,88]). There are a few other minor exceptions. However, all of the additional mnemonics, not present at the assembly language level, are easy to identify and interpret, and a detailed description of the differences will therefore be omitted in this document. Instead, a complete list of the mnemonics, as used by **sim**, is given below. For convenience, the mnemonics have been grouped into functional categories.

1. **registers**: (src or dst)

   AB0 ... AB7 : registers accessible from both buses
   A0 ... A7 : registers accessible from bus A
   B0 ... B7 : registers accessible from bus B

2. **I/O ports**: (src or dst)

   E:N : to/from East port over bus A, North port over bus B
   N:E : to/from North port over bus A, East port over bus B
   S:W : to/from South port over bus A, West port over bus B
   W:S : to/from West port over bus A, South port over bus B

3. **functional unit output registers**: (src only)

SUM1A	:	sum from Adder1 accessible from bus A
CSUM1B	:	1's compl. of sum from Adder1 accessible from bus B
SUM2B	:	sum from Adder2 accessible from bus B
CSUM2A	:	1's compl. of sum from Adder2 accessible from bus B
QUOTA	:	quotient of the Divider unit accessible from bus A
HIGHA	:	maximum output of Sorter unit accessible from bus A
LOWB	:	minimum output of Sorter unit accessible from bus B
SFTA&B	:	outputs of the Shifter unit (accessed together only)

4. **Arithmetic operation codes:**

ADD2	:	addition in Adder2
ADDD	:	addition in both Adder1 and Adder2
MULTF1	:	multiplication in Multiplier1, first stage
MULTS1	:	multiplication in Multiplier1, second stage
MULTF2	:	multiplication in Multiplier2, first stage
MULTS2	:	multiplication in Multiplier2, second stage
DIVS	:	division with inputs coming from the Shifter unit
DIV	:	division with inputs coming from the buses
SORT	:	sort/comparison in the Sorter unit
SHIFT	:	shift/normalization in the Shifter unit

5. **I/O operation codes:**

SWL_IN	:	receive low-order word from South and West neighbors
SWH_IN	:	receive high-order word from South and West neighbors
NEL_IN	:	receive low-order word from North and East neighbors
NEH_IN	:	receive high-order word from North and East neighbors
SWL_OUT	:	send low-order word to South and West neighbors
SWH_OUT	:	send high-order word to South and West neighbors
NEL_OUT	:	send low-order word to North and East neighbors
NEH_OUT	:	send high-order word to North and East neighbors
SWH&NEL	:	receive high-order word from South and West neighbors, and send low-order word to North and East neighbors
NEH&SWL	:	receive high-order word from North and East neighbors, and send low-order word to South and West neighbors

2.5 Processor display format

As mentioned in Section 2.2, the contents of a processor's I/O ports and/or general purpose registers can be displayed in two different formats — fixed point format (unsigned long) and floating point format (double). An example of a floating point format processor information display (taken from Appendix B) is shown below.

```
+-------------------------------------------------------------------------+
|           N.in =  0.0199999995529652  N.out =  0.0000000000000000       |
| W.in  =  0.0000000000000000                    E.in  = -0.0000000009313226 |
| W.out = -0.0000000009313226                    E.out =  0.0000000000000000 |
|          S.in =  0.0000000000000000  S.out = -0.0299999993294477         |
+-------------------------------------------------------------------------+
| AB0:  0.0199999995529652  A0:  0.0199999995529652  B0:  0.0199999995529652 |
| AB1:  0.0000000000000000  A1:  0.0399999991059303  B1:  0.0399999991059303 |
| AB2:  0.0000000000000000  A2:  1.2799999713897700  B2:  0.6399999856948850 |
| AB3:  0.0799999982118606  A3:  0.0000000000000000  B3:  0.0000000000000000 |
| AB4:  0.0000000000000000  A4:  0.0000000000000000  B4:  0.0031999992206693 |
| AB5:  0.0000000000000000  A5:  0.0000000000000000  B5:  0.0000000000000000 |
| AB6:  0.0000000000000000  A6:  0.0000000000000000  B6:  0.0000000000000000 |
| AB7:  0.0000000000000000  A7:  0.0000000000000000  B7:  0.9999999999999996 |
+-----------------------------------+-----------------------------------+
| Adder1:          (dyn. clock =  -1) | Multiplier1:      (dyn. clock =  -1) |
|   SUM1A  =       0.0399999991059303 |   PAR_PROD & CARY : inaccessible     |
|   CSUM1B =      -0.0400000000372529 |   ready  =        y (mul_clock = -1) |
+-----------------------------------+-----------------------------------+
| Adder2:          (dyn. clock =   4) | Multiplier2:      (dyn. clock =   4) |
|   CSUM2A =      -0.0032000001519918 |   PAR_PROD & CARY : inaccessible     |
|   SUM2B  =       0.0031999992206693 |   ready  =        y (mul_clock = -1) |
+-----------------------------------+-----------------------------------+
| Divider:         (dyn. clock =  -1) | Shifter:          (dyn. clock =  -1) |
|   QUOTA  =       0.0000000000000000 |   SHIFTA =        1.2799999713897700 |
|   ready  =       y (div_clock = -1) |   SHIFTB =        0.6399999856948850 |
+-----------------------------------+-----------------------------------+
| Sorter:          (dyn. clock =  -1) | row ........................ 0002 |
|   HIGHA  =       0.0000000000000000 | column ..................... 0016 |
|   LOWB   =       0.0000000000000000 | enabled? ..................  yes |
+-----------------------------------+-----------------------------------+
```

The topmost box gives the contents of the processor's I/O port registers. The values of both *input* and *output* registers (32 bits) for each of the processor's four ports are shown (see [Przytula,88] for a discussion of PE architecture). Following the listing

of the general purpose register contents, the template shows the current status of the seven functional units of the given processor. All functional unit output registers (their names appear capitalized in the template) are dynamic registers, which means that the information they store becomes unreliable after 5 clock cycles. Therefore, each functional unit has an associated *dynamic clock*, which shows the number of cycles *remaining* until the expiration of the register contents. The value of -1 denotes an expired clock.

Because division and multiplication are multi-cycle operations, two additional *global* system clocks are maintained by **sim** — `mul_clock` and `div_clock`. Following the loading of the operands, the SCAP system requires 4 cycles to perform a multiplication, and 9 cycles to perform a division. Therefore, the two clocks are set to 4 and 9, respectively, in the cycle in which the corresponding operation was initiated. When the operation has completed, *i.e.*, when the corresponding clock has expired, the results become available in the dynamic output registers, the `ready` flags are set to `y` (yes), and the corresponding dynamic clocks are set to 5.

As decribed in [Przytula,88], the two multiplier units of the SCAP system produce *partial products* and *cary values*, rather than final products. For each multiplier, the two partial results (stored in dynamic registers PAR_PROD and CARY, respectively), must then be added together to yield the final product. Note that PAR_PROD and CARY are not connected to the buses and thus can not be accessed by the user directly. The reader will notice that the boxes in the above template corresponding to the multiplier units do not give the contents of the PAR_PROD and CARY registers. This is due to the fact that **sim** simulates multiplication as a monolithic operation, and therefore computes the final product directly, without producing intermediate partial product and cary values. However, this in no way compromises the correctness or usefulness of the simulator, as the SCAP hardware provides no way for the user to access and use these intermediate results. Moreover, all timing information concerning availability and maturity of the result as given by **sim** is consistent with the actual hardware behavior.

A final note to the user — **sim** updates all clocks *at the beginning* of each cycle. The clock values displayed at any given point during program execution (*i.e.*, simulation) will therefore be updated in the upcoming cycle *before* the corresponding instruction is executed.

3 Discussion and Suggested Improvements

As of this writing, a couple of fixes to **sim** are still pending due to the prerequisite corrections/modifications that need to be done to the corresponding assembler. The most important of these concerns the loading stage of the simulator. With the current format of the assembler symbol tables, the simulator can not load descending data queues correctly, because the sizes of the queues are not given as part of the symbol table information. This should be fixed shortly.

Data is represented internally as unsigned long 32-bit integer data (see [Przytula,88] for a description of the format). All inter-processor I/O, intra-processor data movements, and even some of the functional units (*e.g.*, shifter) work with this format. Only when data is passed to arithmetic functional units, such as adders, multipliers, the divider, or the sorter, the representation changes and the data is converted to floating point format (double). The current conversion scheme effects the conversion of an unsigned long (representing 32-bit fixed point format) into a floating point format by casting the unsigned long into a double and scaling the result by 2^{-30}. The reverse conversion is analogous. Each such format conversion introduces a slight error, and therefore the error compounds as successive arithmetic computations perform futher conversions. An initial error in data representation is introduced by the conversions at loading time, where real values are read from *datafile* and stored internally as fixed point unsigned longs. Whereas we can clearly not escape some inaccuracies in the presence of multiple representations of data, we may be able to improve on the present scheme.

The current implementation of the simulator (*i.e.*, loader) assumes that scaling of the data into the appropriate range $(-2.0, +2.0)$ has been done by the user and so the data appearing in *datafile* are expected to be in the correct range. Loader warnings are issued if this is not the case, and zeroes are loaded in place of out-of-range values. Whereas this assumption may seem limiting, it also gives the user greater flexibility in choosing her own scaling factor, which minimizes the loss of precision due to scaling for the particular application. The optimal scaling factor for an application which uses a relatively narrow numerical range of data will be different than that corresponding to an application where a very broad range of data and/or results is expected.

As mentioned above, all data is loaded into the top portion of the Data Memory.

The current syntax of the assembly language enables the user to specify a set of data queues, their types, and their sizes. However, the user is not given the option of specifying the starting addresses of where these data queues should appear in memory. Consequently, the data is loaded into Data Memory sequentially, starting at address 0. Should a future extension to the assembler modify the queue declaration mechanism (and reflect the additional information in the object code symbol tables), the simulator's loader could be easily adapted to accomodate this new information.

In Section 2.5 we noted that multiplication in **sim** is handled differently from the actual hardware multiplication. The first stage of the multiplication operation on SCAP hardware produces the *partial product* and the *cary*, which are then added in the second stage to give the final result. **sim**, on the other hand (for reasons of simplicity) computes the final product already in the first stage and simulates the second stage of multiplication as a simple "move" to the appropriate adder output registers, rather than an "add" of the partial product and the cary. As mentioned, this has no effect on the correctness of the results and is completely transparent to the user, as the partial product (PAR_PROD) and cary (CARY) dynamic registers are not accessible from the buses and thus can not be manipulated by the user. However, in keeping with the goal of faithfully adhering to the architecture and workings of the actual machine, this should perhaps be changed, so that multiplication in **sim** in fact mimics SCAP's two-stage multiplication process exactly.

Finally, due to the size and internal complexity of the simulator/debugger system, the system has been only marginally tested. A reasonable stabilizing period of continued usage, testing, and debugging is to be expected.

4 Some Implementational Details

The simulator/debugger is implemented entirely in C and is currently running on a VAX-11/785 under Ultrix V2.0-1. The source code occupies 230 Kbytes of storage ($\approx$ 7,000 source lines of code), and the optimized object code takes up 66 Kbytes of storage.

References

[1] Miriam A. Hartholz. *The Systolic/Cellular System Assembler: User's Guide.* Master's thesis, University of Pennsylvania, August 1988.

[2] K. Wojtek Przytula. *Systolic/Cellular System.* Hughes Research Laboratories, March 1988. Technical reference manual.

A A sample SCAP assembly language program and data file

```
DEFQUEUE Q1    16                       ; data queue  (16 rows, asc)
DEFQUEUE Q2     1                       ; data queue  (1 row, constant)

DEFMASK M1 (::2-7,13-17,23-28)          ; dummy mask (diagonal)
DEFMASK M2 (1-4,9-12:5-8,13-16:)        ; processor mask (row/column)

        NOP                             ;
        READQ Q2                        ; read from constant queue Q2
L0:     GETNR(B7,B7)                    ;
        LOOP 15 L0                      ; every proc has 1 in B7

        NOP                             ;
        READQ Q1                        ; set read address to Q1
L1:     GETNR(A0,A0)                    ; read from ascending queue Q1
        LOOP 15 L1                      ; read 16 rows of data

        ; internal data movements (masked)

L2:     MOV(A0,AB0:)              M2    ;
        MOV(:AB0,B0)              M2    ; A0 = B0 = AB0 = x (x = local data)
        ADDD(A0,B0)               M2    ; addition in both adders
        MOV(SUM1A,A1:SUM2B,B1)    M2    ; A1 = B1 = 2*x
        SHIFT(A1,B0)              M2    ; shifting => 1 <= A1 < 2
        MOV(SHIFTA,A2:SHIFTB,B2)  M2    ; A2 = shift(A1), B2 = shift(B0)
        ADD2(A1,B1)               M2    ; addition in Adder 2
        MOV(:SUM2B,AB3)           M2    ; AB3 = 4*x

L3:     MULTF2(A1,SUM2B)          M2    ; multiplication in Multiplier 2 (load)
        NOP                       M2    ; 4 cycles of computation
        NOP                       M2    ;
        NOP                       M2    ;
        NOP                       M2    ;
        MULTS2                    M2    ; second stage of mult. in Multiplier 2
        MOV(:PROD2B,B4)           M2    ; B4 = (2*x)*(4*x) = 8*x^2

                                        ; subtract : AB3 - B1 (4*x - 2*x)
L4:     ADD2(_,B1)                M2    ; CSUM2A = ~(2*x + (-2^-30)) = -2*x
        ADD2(CSUM2A,AB3)          M2    ; SUM2B = -2*x + 4*x = 2*x
        MOV(:SUM2B,AB4)           M2    ; AB4 = 2*x (the difference)

L5:     DIVF(A1,B0)               M2    ; division : normalize
        DIVS                      M2    ; division : divide B0/A1
        NOP                       M2    ; 1
        NOP                       M2    ; 2
        NOP                       M2    ; 3
        NOP                       M2    ; 4
        SORT(AB4,B2)              M2    ; 5 (intermediate sort operation)
        NOP                       M2    ; 6
        NOP                       M2    ; 7
        NOP                       M2    ; 8
        NOP                       M2    ; 9
        MOV(QUOTA,A3:)            M2    ; A3 = 0.5 (the quotient)

        STOP                            ; return
        END                             ;
```



```
0.16    0.16    0.16    0.16    0.16    0.16    0.16    0.16
0.16    0.16    0.16    0.16    0.16    0.16    0.16    0.16

-0.15   -0.15   -0.15   -0.15   -0.15   -0.15   -0.15   -0.15
-0.15   -0.15   -0.15   -0.15   -0.15   -0.15   -0.15   -0.15

0.14    0.14    0.14    0.14    0.14    0.14    0.14    0.14
0.14    0.14    0.14    0.14    0.14    0.14    0.14    0.14

-0.13   -0.13   -0.13   -0.13   -0.13   -0.13   -0.13   -0.13
-0.13   -0.13   -0.13   -0.13   -0.13   -0.13   -0.13   -0.13

0.12    0.12    0.12    0.12    0.12    0.12    0.12    0.12
0.12    0.12    0.12    0.12    0.12    0.12    0.12    0.12

-0.11   -0.11   -0.11   -0.11   -0.11   -0.11   -0.11   -0.11
-0.11   -0.11   -0.11   -0.11   -0.11   -0.11   -0.11   -0.11

0.10    0.10    0.10    0.10    0.10    0.10    0.10    0.10
0.10    0.10    0.10    0.10    0.10    0.10    0.10    0.10

-0.09   -0.09   -0.09   -0.09   -0.09   -0.09   -0.09   -0.09
-0.09   -0.09   -0.09   -0.09   -0.09   -0.09   -0.09   -0.09

0.08    0.08    0.08    0.08    0.08    0.08    0.08    0.08
0.08    0.08    0.08    0.08    0.08    0.08    0.08    0.08

-0.07   -0.07   -0.07   -0.07   -0.07   -0.07   -0.07   -0.07
-0.07   -0.07   -0.07   -0.07   -0.07   -0.07   -0.07   -0.07

0.06    0.06    0.06    0.06    0.06    0.06    0.06    0.06
0.06    0.06    0.06    0.06    0.06    0.06    0.06    0.06

-0.05   -0.05   -0.05   -0.05   -0.05   -0.05   -0.05   -0.05
-0.05   -0.05   -0.05   -0.05   -0.05   -0.05   -0.05   -0.05

0.04    0.04    0.04    0.04    0.04    0.04    0.04    0.04
0.04    0.04    0.04    0.04    0.04    0.04    0.04    0.04

-0.03   -0.03   -0.03   -0.03   -0.03   -0.03   -0.03   -0.03
-0.03   -0.03   -0.03   -0.03   -0.03   -0.03   -0.03   -0.03

0.02    0.02    0.02    0.02    0.02    0.02    0.02    0.02
0.02    0.02    0.02    0.02    0.02    0.02    0.02    0.02

-0.01   -0.01   -0.01   -0.01   -0.01   -0.01   -0.01   -0.01
-0.01   -0.01   -0.01   -0.01   -0.01   -0.01   -0.01   -0.01

1       1       1       1       1       1       1       1
1       1       1       1       1       1       1       1
```

B A sample run of the simulator/debugger

```
% sim ep7.exe ep7.dat

program "ep7.exe" loaded: 39 (dec) instructions.

sim> ?

      ?, help          -- print this message.
      run, cont        -- run the program, resume execution after a breakpoint.
      step [j]         -- execute next j instructions (default: j = 1).
      pc               -- show contents of program counter.
      readaddr         -- show contents of read-address counter.
      writeaddr        -- show contents of write-address counter.
      pfifo            -- show contents of program FIFO.
      wfifo            -- show contents of write FIFO.
      rfifo            -- show contents of read FIFO.
      proc i j [f]     -- show contents of processor in row i and column j.
                          (f = fixed-point format [default = floating-point])
      proglen          -- display program length.
      memory i [j] [f] -- display rows i through j of data memory.
                          (f = fixed-point format [default = floating-point])
      instr i [j]      -- display rows i through j of program memory.
      mask <lbl>       -- display mask with symbolic name <lbl>.
      sysfld <code>    -- decode the (hex) system field <code>.
      dump             -- produce a system dump in file "core.dump".
      break [<lbl>]    -- set a breakpoint at <lbl>, show all breakpoints.
      unbreak <lbl>    -- remove the breakpoint at <lbl>.
      exit, quit       -- exit the simulator.

sim> instr 0 12
                          external (ph1/ph2)    internal (ph1/ph2)
 pc  sel   mask   sysfld  i/o     src    dst    i/o     src    dst
-------------------------------------------------------------------
0000  D  00000000  007F   NOP     NOP    NOP    NOP     NOP    NOP
                         NOP     NOP    NOP    NOP     NOP    NOP
0001  D  00000000  00BF   NEL_IN  NOP    W:S    NEL_IN  NOP    W:S
                         SWL_OUT B7     NOP    SWL_OUT B7     NOP
0002  D  00000000  00BF   NOP     NOP    NOP    NOP     NOP    NOP
                         NEH_IN  NOP    NOP    NEH_IN  NOP    NOP
0003  D  00000000  00B7   SWH_OUT E:N    NOP    SWH_OUT E:N    NOP
                         NOP     NOP    B7     NOP     NOP    B7
0004  D  00000000  007F   NOP     NOP    NOP    NOP     NOP    NOP
                         NOP     NOP    NOP    NOP     NOP    NOP
0005  D  00000000  00BF   NEL_IN  A0     S:W    NEL_IN  A0     S:W
                         SWL_OUT NOP    NOP    SWL_OUT NOP    NOP
0006  D  00000000  00BF   NOP     NOP    NOP    NOP     NOP    NOP
                         NEH_IN  NOP    NOP    NEH_IN  NOP    NOP
0007  D  00000000  00B7   SWH_OUT N:E    A0     SWH_OUT N:E    A0
                         NOP     NOP    NOP    NOP     NOP    NOP
0008 R/C 0F0FF0F0  00FF   NOP     A0     AB0    NOP     A0     AB0
                         NOP     NOP    NOP    NOP     NOP    NOP
0009 R/C 0F0FF0F0  00FF   NOP     NOP    NOP    NOP     NOP    NOP
                         NOP     AB0    B0     NOP     AB0    B0
0010 R/C 0F0FF0F0  00FF   NOP     A0     NOP    NOP     A0     NOP
                         NOP     B0     ADDD   NOP     B0     ADDD
0011 R/C 0F0FF0F0  00FF   NOP     SUM1A  A1     NOP     SUM1A  A1
                         NOP     SUM2B  B1     NOP     SUM2B  B1
0012 R/C 0F0FF0F0  00FF   NOP     A1     NOP    NOP     A1     NOP
                         NOP     B0     SHIFT  NOP     B0     SHIFT
sim> pfifo

     0001 0001 0001 0001 0001 0001 0001 0001 0001 0001 0001 0001 0001 0001 0001
     0001 0004 0005 0005 0005 0005 0005 0005 0005 0005 0005 0005 0005 0005
     0005 0005 0005 0008
```



```
     Number of items = 32
     FIFO status = OK

sim> rfifo

     4010 0000

     Number of items = 2
     FIFO status = OK

sim> wfifo

     Number of items = 0
     FIFO status = EMPTY

sim> readaddr
     read address = 0000 (hex) = 0000 (oct) = 0000 (dec)
sim> memory 13 100
0013 -0.0299999993 -0.0299999993 -0.0299999993 -0.0299999993
     -0.0299999993 -0.0299999993 -0.0299999993 -0.0299999993
     -0.0299999993 -0.0299999993 -0.0299999993 -0.0299999993
     -0.0299999993 -0.0299999993 -0.0299999993 -0.0299999993
0014  0.0199999996  0.0199999996  0.0199999996  0.0199999996
      0.0199999996  0.0199999996  0.0199999996  0.0199999996
      0.0199999996  0.0199999996  0.0199999996  0.0199999996
      0.0199999996  0.0199999996  0.0199999996  0.0199999996
0015 -0.0099999998 -0.0099999998 -0.0099999998 -0.0099999998
     -0.0099999998 -0.0099999998 -0.0099999998 -0.0099999998
     -0.0099999998 -0.0099999998 -0.0099999998 -0.0099999998
     -0.0099999998 -0.0099999998 -0.0099999998 -0.0099999998
0016  1.0000000000  1.0000000000  1.0000000000  1.0000000000
      1.0000000000  1.0000000000  1.0000000000  1.0000000000
      1.0000000000  1.0000000000  1.0000000000  1.0000000000
      1.0000000000  1.0000000000  1.0000000000  1.0000000000
0017  0.0000000000  0.0000000000  0.0000000000  0.0000000000
      0.0000000000  0.0000000000  0.0000000000  0.0000000000
      0.0000000000  0.0000000000  0.0000000000  0.0000000000
      0.0000000000  0.0000000000  0.0000000000  0.0000000000
      *
0100  0.0000000000  0.0000000000  0.0000000000  0.0000000000
      0.0000000000  0.0000000000  0.0000000000  0.0000000000
      0.0000000000  0.0000000000  0.0000000000  0.0000000000
      0.0000000000  0.0000000000  0.0000000000  0.0000000000
sim> break L2
sim> break L5
sim> break L3
sim> break L1
sim> break L4
sim> unbreak L1
sim> break
Currently active breakpoints:
  breakpoint "L2" at address 0008 (hex) = 0008 (dec)
  breakpoint "L3" at address 0010 (hex) = 0016 (dec)
  breakpoint "L4" at address 0017 (hex) = 0023 (dec)
  breakpoint "L5" at address 001a (hex) = 0026 (dec)
sim> pc
     pc = 0000 (hex) = 0000 (oct) = 0000 (dec)
sim> run
breakpoint (L2): pc = 0008 (hex) = 0008 (dec)
sim> readaddr
     read address = 0010 (hex) = 0020 (oct) = 0016 (dec)
```

```
sim> proc 13 2
+-----------------------------------------------------------------------+
|          N.in = -0.1299999952316284  N.out =  0.0000000000000000     |
| W.in  =  0.0000000000000000                   E.in  = -0.0000000009313226 |
| W.out = -0.0000000009313226                   E.out =  0.0000000000000000 |
|         S.in =  0.0000000000000000  S.out =  0.1400000005960464       |
+-----------------------------------------------------------------------+
| AB0:  0.0000000000000000  A0: -0.1299999952316284  B0:  0.0000000000000000 |
| AB1:  0.0000000000000000  A1:  0.0000000000000000  B1:  0.0000000000000000 |
| AB2:  0.0000000000000000  A2:  0.0000000000000000  B2:  0.0000000000000000 |
| AB3:  0.0000000000000000  A3:  0.0000000000000000  B3:  0.0000000000000000 |
| AB4:  0.0000000000000000  A4:  0.0000000000000000  B4:  0.0000000000000000 |
| AB5:  0.0000000000000000  A5:  0.0000000000000000  B5:  0.0000000000000000 |
| AB6:  0.0000000000000000  A6:  0.0000000000000000  B6:  0.0000000000000000 |
| AB7:  0.0000000000000000  A7:  0.0000000000000000  B7:  0.9999999999999996 |
+-----------------------------------+-----------------------------------+
| Adder1:           (dyn. clock =  -1) | Multiplier1:      (dyn. clock =  -1) |
|   SUM1A =        0.0000000000000000 |   PAR_PROD & CARY : inaccessible     |
|   CSUM1B =       0.0000000000000000 |   ready  =        y (mul_clock = -1) |
+-----------------------------------+-----------------------------------+
| Adder2:           (dyn. clock =  -1) | Multiplier2:      (dyn. clock =  -1) |
|   CSUM2A =       0.0000000000000000 |   PAR_PROD & CARY : inaccessible     |
|   SUM2B  =       0.0000000000000000 |   ready  =        y (mul_clock = -1) |
+-----------------------------------+-----------------------------------+
| Divider:          (dyn. clock =  -1) | Shifter:          (dyn. clock =  -1) |
|   QUOTA  =       0.0000000000000000 |   SHIFTA =       0.0000000000000000 |
|   ready  =       y (div_clock = -1) |   SHIFTB =       0.0000000000000000 |
+-----------------------------------+-----------------------------------+
| Sorter:           (dyn. clock =  -1) | row ........................ 0013 |
|   HIGHA  =       0.0000000000000000 | column ..................... 0002 |
|   LOWB   =       0.0000000000000000 | enabled? ..................  yes |
+-----------------------------------+-----------------------------------+
sim> mask M1

mask: f03e0f81 (diagonal) -- processors marked as "x" are enabled.

      1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
 1    x  x  x  x                    x  x  x  x  x  x
 2    x  x  x  x  x                    x  x  x  x  x  x
 3       x  x  x  x  x                    x  x  x  x  x
 4          x  x  x  x  x                    x  x  x  x
 5             x  x  x  x  x                    x  x  x
 6                x  x  x  x  x                    x  x
 7                   x  x  x  x  x                    x
 8  x                   x  x  x  x  x
 9  x  x                   x  x  x  x  x
10  x  x  x                   x  x  x  x
11  x  x  x  x                    x  x  x  x  x
12  x  x  x  x  x                    x  x  x  x  x
13  x  x  x  x  x  x                    x  x  x  x  x
14     x  x  x  x  x  x                    x  x  x  x
15        x  x  x  x  x  x                    x  x  x
16           x  x  x  x  x  x                    x  x

sim> mask M2

mask: 0f0ff0f0 (row/column) -- processors marked as "x" are enabled.

      1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
 1              x  x  x  x                 x  x  x  x
 2              x  x  x  x                 x  x  x  x
 3              x  x  x  x                 x  x  x  x
 4              x  x  x  x                 x  x  x  x
```



```
 5
 6
 7
 8
 9              x  x  x  x                 x  x  x  x
10              x  x  x  x                 x  x  x  x
11              x  x  x  x                 x  x  x  x
12              x  x  x  x                 x  x  x  x
13
14
15
16

sim> pc
     pc = 0008 (hex) = 0010 (oct) = 0008 (dec)
sim> instr 8
                                 external (ph1/ph2)    internal (ph1/ph2)
 pc  sel   mask   sysfld  i/o     src     dst     i/o     src     dst
-------------------------------------------------------------------------
0008 R/C 0F0FF0F0  00FF   NOP     A0      AB0     NOP     A0      AB0
                          NOP     NOP     NOP     NOP     NOP     NOP
sim> cont
breakpoint (L3): pc = 0010 (hex) = 0016 (dec)
sim> instr 16
                                 external (ph1/ph2)    internal (ph1/ph2)
 pc  sel   mask   sysfld  i/o     src     dst     i/o     src     dst
-------------------------------------------------------------------------
0016 R/C 0F0FF0F0  00FB   NOP     A1      MULTF2  NOP     A1      MULTF2
                          NOP     SUM2B   NOP     NOP     SUM2B   NOP
sim> sysfld fb

system field = 00fb : asserted signals are denoted by "X".

load_read_addrs  read  load_write_addrs  write  load_pc  multiply  divide  stop
---------------  ----  ----------------  -----  -------  --------  ------  ----
                                                           X

sim> proc 10 14
+-----------------------------------------------------------------------+
|          N.in =  0.1000000014901161  N.out =  0.0000000000000000     |
| W.in  =  0.0000000000000000                   E.in  = -0.0000000009313226 |
| W.out = -0.0000000009313226                   E.out =  0.0000000000000000 |
|         S.in =  0.0000000000000000  S.out = -0.1099999994039535       |
+-----------------------------------------------------------------------+
| AB0:  0.1000000014901161  A0:  0.1000000014901161  B0:  0.1000000014901161 |
| AB1:  0.0000000000000000  A1:  0.2000000029802322  B1:  0.2000000029802322 |
| AB2:  0.0000000000000000  A2:  1.6000000238418573  B2:  0.8000000119209287 |
| AB3:  0.4000000059604643  A3:  0.0000000000000000  B3:  0.0000000000000000 |
| AB4:  0.0000000000000000  A4:  0.0000000000000000  B4:  0.0000000000000000 |
| AB5:  0.0000000000000000  A5:  0.0000000000000000  B5:  0.0000000000000000 |
| AB6:  0.0000000000000000  A6:  0.0000000000000000  B6:  0.0000000000000000 |
| AB7:  0.0000000000000000  A7:  0.0000000000000000  B7:  0.9999999999999996 |
+-----------------------------------+-----------------------------------+
| Adder1:           (dyn. clock =   0) | Multiplier1:      (dyn. clock =  -1) |
|   SUM1A =        0.2000000029802322 |   PAR_PROD & CARY : inaccessible     |
|   CSUM1B =      -0.2000000039115547 |   ready  =        y (mul_clock = -1) |
+-----------------------------------+-----------------------------------+
| Adder2:           (dyn. clock =   4) | Multiplier2:      (dyn. clock =  -1) |
|   CSUM2A =      -0.4000000068917869 |   PAR_PROD & CARY : inaccessible     |
|   SUM2B  =       0.4000000059604643 |   ready  =        y (mul_clock = -1) |
+-----------------------------------+-----------------------------------+
| Divider:          (dyn. clock =  -1) | Shifter:          (dyn. clock =   2) |
|   QUOTA  =       0.0000000000000000 |   SHIFTA =       1.6000000238418573 |
|   ready  =       y (div_clock = -1) |   SHIFTB =       0.8000000119209287 |
```

```
+-----------------------------------+-----------------------------------+
| Sorter:          (dyn. clock =  -1) | row ........................ 0010 |
|   HIGHA  =        0.0000000000000000 | column ..................... 0014 |
|   LOWB   =        0.0000000000000000 | enabled? ..................  yes |
+-----------------------------------+-----------------------------------+
sim> cont
breakpoint (L4): pc = 0017 (hex) = 0023 (dec)
sim> instr 23
                          external (ph1/ph2)    internal (ph1/ph2)
 pc  sel   mask   sysfld  i/o     src    dst    i/o     src    dst
-------------------------------------------------------------------
0023 R/C 0F0FF0F0  00FF   NOP     NOP    NOP    NOP     NOP    NOP
                          NOP     B1     ADD2   NOP     B1     ADD2
sim> proc 2 16
+-------------------------------------------------------------------------+
|          N.in =  0.0199999995529652  N.out =  0.0000000000000000        |
| W.in  =  0.0000000000000000                    E.in  = -0.0000000009313226 |
| W.out = -0.0000000009313226                    E.out =  0.0000000000000000 |
|         S.in =  0.0000000000000000  S.out = -0.0299999993294477          |
+-------------------------------------------------------------------------+
| AB0:  0.0199999995529652  A0:  0.0199999995529652  B0:  0.0199999995529652 |
| AB1:  0.0000000000000000  A1:  0.0399999991059303  B1:  0.0399999991059303 |
| AB2:  0.0000000000000000  A2:  1.2799999713897700  B2:  0.6399999856948850 |
| AB3:  0.0799999982118606  A3:  0.0000000000000000  B3:  0.0000000000000000 |
| AB4:  0.0000000000000000  A4:  0.0000000000000000  B4:  0.0031999992206693 |
| AB5:  0.0000000000000000  A5:  0.0000000000000000  B5:  0.0000000000000000 |
| AB6:  0.0000000000000000  A6:  0.0000000000000000  B6:  0.0000000000000000 |
| AB7:  0.0000000000000000  A7:  0.0000000000000000  B7:  0.9999999999999996 |
+-----------------------------------+-----------------------------------+
| Adder1:          (dyn. clock =  -1) | Multiplier1:      (dyn. clock =  -1) |
|   SUM1A  =        0.0399999991059303 |   PAR_PROD & CARY : inaccessible     |
|   CSUM1B =       -0.0400000000372529 |   ready  =        y (mul_clock = -1) |
+-----------------------------------+-----------------------------------+
| Adder2:          (dyn. clock =   4) | Multiplier2:      (dyn. clock =   4) |
|   CSUM2A =       -0.0032000001519918 |   PAR_PROD & CARY : inaccessible     |
|   SUM2B  =        0.0031999992206693 |   ready  =        y (mul_clock = -1) |
+-----------------------------------+-----------------------------------+
| Divider:         (dyn. clock =  -1) | Shifter:          (dyn. clock =  -1) |
|   QUOTA  =        0.0000000000000000 |   SHIFTA =        1.2799999713897700 |
|   ready  =        y (div_clock = -1) |   SHIFTB =        0.6399999856948850 |
+-----------------------------------+-----------------------------------+
| Sorter:          (dyn. clock =  -1) | row ........................ 0002 |
|   HIGHA  =        0.0000000000000000 | column ..................... 0016 |
|   LOWB   =        0.0000000000000000 | enabled? ..................  yes |
+-----------------------------------+-----------------------------------+
sim> step
sim> step 12
breakpoint (L5): pc = 001a (hex) = 0026 (dec)
sim> instr 26
                          external (ph1/ph2)    internal (ph1/ph2)
 pc  sel   mask   sysfld  i/o     src    dst    i/o     src    dst
-------------------------------------------------------------------
0026 R/C 0F0FF0F0  00FF   NOP     A1     NOP    NOP     A1     NOP
                          NOP     B0     SHIFT  NOP     B0     SHIFT
sim> proc 3 6
+-------------------------------------------------------------------------+
|          N.in = -0.0299999993294477  N.out =  0.0000000000000000        |
| W.in  =  0.0000000000000000                    E.in  = -0.0000000009313226 |
| W.out = -0.0000000009313226                    E.out =  0.0000000000000000 |
|         S.in =  0.0000000000000000  S.out =  0.0399999991059303          |
+-------------------------------------------------------------------------+
| AB0: -0.0299999993294477  A0: -0.0299999993294477  B0: -0.0299999993294477 |
| AB1:  0.0000000000000000  A1: -0.0599999986588955  B1: -0.0599999986588955 |
| AB2:  0.0000000000000000  A2: -1.9199999570846551  B2: -0.9599999785423275 |
```



```
| AB3: -0.1199999973177909  A3:  0.0000000000000000  B3:  0.0000000000000000 |
| AB4: -0.0599999986588955  A4:  0.0000000000000000  B4:  0.0071999989449978 |
| AB5:  0.0000000000000000  A5:  0.0000000000000000  B5:  0.0000000000000000 |
| AB6:  0.0000000000000000  A6:  0.0000000000000000  B6:  0.0000000000000000 |
| AB7:  0.0000000000000000  A7:  0.0000000000000000  B7:  0.9999999999999996 |
+-----------------------------------+-----------------------------------+
| Adder1:          (dyn. clock =  -1) | Multiplier1:      (dyn. clock =  -1) |
|   SUM1A  =       -0.0599999986588955 |   PAR_PROD & CARY : inaccessible     |
|   CSUM1B =        0.0599999977275729 |   ready  =        y (mul_clock = -1) |
+-----------------------------------+-----------------------------------+
| Adder2:          (dyn. clock =   4) | Multiplier2:      (dyn. clock =   1) |
|   CSUM2A =        0.0599999977275729 |   PAR_PROD & CARY : inaccessible     |
|   SUM2B  =       -0.0599999986588955 |   ready  =        y (mul_clock = -1) |
+-----------------------------------+-----------------------------------+
| Divider:         (dyn. clock =  -1) | Shifter:          (dyn. clock =  -1) |
|   QUOTA  =        0.0000000000000000 |   SHIFTA =       -1.9199999570846551 |
|   ready  =        y (div_clock = -1) |   SHIFTB =       -0.9599999785423275 |
+-----------------------------------+-----------------------------------+
| Sorter:          (dyn. clock =  -1) | row ........................ 0003 |
|   HIGHA  =        0.0000000000000000 | column ..................... 0006 |
|   LOWB   =        0.0000000000000000 | enabled? ..................  yes |
+-----------------------------------+-----------------------------------+
sim> cont
STOP reached -- normal termination.
sim> proc 12 8
+-------------------------------------------------------------------------+
|          N.in =  0.1199999973177909  N.out =  0.0000000000000000        |
| W.in  =  0.0000000000000000                    E.in  = -0.0000000009313226 |
| W.out = -0.0000000009313226                    E.out =  0.0000000000000000 |
|         S.in =  0.0000000000000000  S.out = -0.1299999952316284          |
+-------------------------------------------------------------------------+
| AB0:  0.1199999973177909  A0:  0.1199999973177909  B0:  0.1199999973177909 |
| AB1:  0.0000000000000000  A1:  0.2399999946355819  B1:  0.2399999946355819 |
| AB2:  0.0000000000000000  A2:  1.9199999570846551  B2:  0.9599999785423275 |
| AB3:  0.4799999892711638  A3:  0.4999999999999998  B3:  0.0000000000000000 |
| AB4:  0.2399999946355819  A4:  0.0000000000000000  B4:  0.1151999942958355 |
| AB5:  0.0000000000000000  A5:  0.0000000000000000  B5:  0.0000000000000000 |
| AB6:  0.0000000000000000  A6:  0.0000000000000000  B6:  0.0000000000000000 |
| AB7:  0.0000000000000000  A7:  0.0000000000000000  B7:  0.9999999999999996 |
+-----------------------------------+-----------------------------------+
| Adder1:          (dyn. clock =  -1) | Multiplier1:      (dyn. clock =  -1) |
|   SUM1A  =        0.2399999946355819 |   PAR_PROD & CARY : inaccessible     |
|   CSUM1B =       -0.2399999955669045 |   ready  =        y (mul_clock = -1) |
+-----------------------------------+-----------------------------------+
| Adder2:          (dyn. clock =  -1) | Multiplier2:      (dyn. clock =  -1) |
|   CSUM2A =       -0.2399999955669045 |   PAR_PROD & CARY : inaccessible     |
|   SUM2B  =        0.2399999946355819 |   ready  =        y (mul_clock = -1) |
+-----------------------------------+-----------------------------------+
| Divider:         (dyn. clock =   5) | Shifter:          (dyn. clock =  -1) |
|   QUOTA  =        0.4999999999999998 |   SHIFTA =        1.9199999570846551 |
|   ready  =        y (div_clock = -1) |   SHIFTB =        0.9599999785423275 |
+-----------------------------------+-----------------------------------+
| Sorter:          (dyn. clock =   0) | row ........................ 0012 |
|   HIGHA  =        0.9599999785423275 | column ..................... 0008 |
|   LOWB   =        0.2399999946355819 | enabled? ..................  yes |
+-----------------------------------+-----------------------------------+
sim> proc 12 8 f
+-------------------------------------------------------------------------+
| W.in  = 00000000    N.in  = 07ae1478    N.out = 00000000    E.in  = ffffffff |
| W.out = ffffffff    S.in  = 00000000    S.out = f7ae1480    E.out = 00000000 |
+-------------------------------------------------------------------------+
| AB: 07ae1478 00000000 00000000 1eb851e0 0f5c28f0 00000000 00000000 00000000 |
| A : 07ae1478 0f5c28f0 7ae14780 20000000 00000000 00000000 00000000 00000000 |
| B : 07ae1478 0f5c28f0 3d70a3c0 00000000 075f6fcc 00000000 00000000 40000000 |
```

```
+-----------------+-----------------+-----------------+-----------------+
| Adder1:     (-1) | Multiplier1: (-1) | Adder2:     (-1) | Multiplier2: (-1) |
| SUM1A = 0f5c28f0 | PAR_PROD  &  CARY | CSUM2A= f0a3d70f | PAR_PROD  &  CARY |
| CSUM1B= f0a3d70f | ready  = yes (-1) | SUM2B = 0f5c28f0 | ready  = yes (-1) |
+-----------------+-----------------+-----------------+-----------------+
| Divider:    (05) | Shifter:     (-1) | Sorter:     (00) | row     =    0012 |
| QUOTA = 20000000 | SHIFTA = 7ae14780 | HIGHA = 3d70a3c0 | col     =    0008 |
| ready = yes (-1) | SHIFTB = 3d70a3c0 | LOWB  = 0f5c28f0 | enabled =    TRUE |
+-----------------+-----------------+-----------------+-----------------+
sim> dump
Dumping coprocessor contents to file "core.dump" ... done.
sim> quit
%
```