August 1988

The Systolic/Cellular System Assembler: User's Guide

Miriam A. Hartholz
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-39.

The Systolic/Cellular System Assembler: User's Guide

Abstract

As components are getting cheaper and smaller, computer systems are getting larger (in number of components) and more complex. In the new age of parallel computing comes entirely new domains of problems to solve. There are two ways to parallelize a problem. One is to restructure a known algorithm so that independent parts run in parallel. The other method is to restructure the problem so that it fits well onto parallel architectures. The Systolic/Cellular System is an array of processors which run in parallel. Its architecture was designed to implement a particular algorithm for matrix manipulation very well. This algorithm, called the Faddeev Algorithm, is well suited to solve a wide variety of operations such as matrix inverse, matrix multiplication, and matrix addition. It can also be used to calculate more complex problems such as the least squares problem and the inverse Jacobian. To efficiently implement this and other algorithms, it is necessary to program as close as possible to the architecture. The obvious way to do this is in machine code, but machine code is hard to read, tedious to write, and almost impossible to debug. The next step is to write an Assembler, and give mnemonics to the various operations, and making the system easier to program. This was the goal of my project. In this document you will find a user's manual for an Assembler for the Systolic/Cellular System. In it, I have described the architecture, issues involved in programming this machine, the input requirements of the Assembler, and a brief discussion on the architecture and how it can be improved to make it an easier machine to program.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-39.

THE SYSTOLIC/CELLULAR SYSTEM ASSEMBLER: USER'S GUIDE

Miriam A. Hartholz

MS-CIS-88-39
GRASP LAB 143

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

June 1988

Acknowledgements: This research was supported in part by NSF grants MIP-8714689, MCS-8219196-CER, IRI84-10413-AO2 and U.S. Army grants DAA29-84-K-0061, DAA29-84-9-0027.

UNIVERSITY OF PENNSYLVANIA
THE MOORE SCHOOL OF ELECTRICAL ENGINEERING
SCHOOL OF ENGINEERING AND APPLIED SCIENCE

THE SYSTOLIC/CELLULAR SYSTEM ASSEMBLER: USER'S GUIDE

Miriam A. Hartholz

Philadelphia, Pennsylvania
August, 1988

A thesis presented to the Faculty of Engineering and Applied Science of the University of Pennsylvania in partial fulfillment of the requirements for the degree of Master of Science in Engineering for graduate work in Electrical Engineering.

Dr. Richard Paul

Dr. Sohrab Rabii

Abstract

As components are getting cheaper and smaller, computer systems are getting larger (in number of components) and more complex. In the new age of parallel computing comes entirely new domains of problems to solve. There are two ways to parallelize a problem. One is to restructure a known algorithm so that independent parts run in parallel. The other method is to restructure the problem so that it fits well onto parallel architectures. The Systolic/Cellular System is an array of processors which run in parallel. Its architecture was designed to implement a particular algorithm for matrix manipulation very well. This algorithm, called the Faddeev Algorithm, is well suited to solve a wide variety of operations such as matrix inverse, matrix multiplication, and matrix addition. It can also be used to calculate more complex problems such as the least squares problem and the inverse Jacobian. To efficiently implement this and other algorithms, it is necessary to program as close as possible to the architecture. The obvious way to do this is in machine code, but machine code is hard to read, tedious to write, and almost impossible to debug. The next step is to write an Assembler, and give mnemonics to the various operations, and making the system easier to program. This was the goal of my project. In this document you will find a user's manual for an Assembler for the Systolic/Cellular System. In it, I have described the architecture, issues involved in programming this machine, the input requirements of the Assembler, and a brief discussion on the architecture and how it can be improved to make it an easier machine to program.

Acknowledgements

I would like to give my sincere thanks to the following people for their help in making this project possible:

Dr. Richard Paul: my advisor, for his advice, support, and trust.

Wojtek Przytula: of the research team at Hughes Research Laboratories developing the system, for his time and patience in describing the architecture of the processor, answering all my questions, and giving me feedback on my ideas.

The members of the *Hughes* group at Penn: Yehong Zhang, Roberto Shironoshita, and Janez Funda for listening, for their input, and for their moral support.

Finally, I would like to thank my family and friends for their moral support, even though they had absolutely no idea what this project was about.

Table Of Contents

Chapter 1
Introduction

The prototype of the Systolic/Cellular System is being developed at Hughes Research Laboratories in Malibu California. It consists of the Systolic/Cellular Co-processor, which is a programmable multiprocessor computer, connected to a general purpose computer which will serve as the HOST. The Co-processor was designed for digital signal processing and image processing applications [PRZ88], but it will be used in the field of robotics to compute the Inverse Jacobian Matrix necessary for computing joint coordinates for an experimental robot arm at the University of Pennsylvania.

The Systolic/Cellular Co-processor can be programmed either directly in machine code, or by using the Systolic/Cellular System Assembly language described in this document. The SCS Cross-Assembler can be run on any UNIX[1] compatible system then downloaded from the HOST to the Co-processor while the Co-processor is in a HALT state.

Chapter two gives a general description of the machine's architecture in order to give the programmer enough of a background to program the system. It describes the system architecture down to the processing element's architecture. Chapter three discusses how to program the system, describing the assembly code file format, the Assembler directives, and the output file format. Chapter four gives a detailed description of the opcodes, their functions and their machine code

[1] UNIX is a trademark of Bell Laboratories.

equivalents. Chapter five talks about the architecture in general and how the current architecture could be improved to simplify the programming task. Finally there is the conclusion which discusses the general areas where the Assembler could use some improvement.

The purpose of this document is to serve mainly as a reference manual for programmers of the Systolic/Cellular System. Therefore, the reader will find some issues are repeated quite frequently, especially those issues regarding the side effects and limitations of the operations. This is so the user can look up an item and know that all the relevant information about the operation is supplied locally.

Chapter 2
The Co-processor Description

This chapter gives a brief overview of the system architecture to give the user sufficient background in order to program it. For greater detail, please refer to [PRZ88]. The references for this chapter are [PRZ88] and [SHI88]. Some sentences are copied directly from [PRZ88].

The Co-processor consists of three major parts, the **Processor Array**, the **Array Memory** (also called Data Memory), and the **Controller**. See Figure 2.0.1. The HOST can access the **Controller** to load the FIFOs and the program memory, to read the status register, and to start and stop the Co-processor. The HOST can access the WRITE port of the **Array Memory** to load and unload data. All loading and unloading by the HOST must be executed when the Co-processor is in the HALT state. The HOST has no direct access to the **Processor Array** at all.

2.1 The Processor Array

The **Processor Array** is a 16 × 16 array of identical custom processors[2] connected as a mesh (nearest neighbor connections) with horizontal wrap-around. See Figure 2.1.1. The processors are labeled by rows and columns; the top row is row 1 and the leftmost column is column 0. The PEs in row 1 and row 16 are connected to **Data Memory** (or the **Array Memory**). Row 1 has *read only* memory access via the **READ port** and row 16 has *write only* memory access via

[2] sometimes referred to in this document as *processing elements* or PEs.

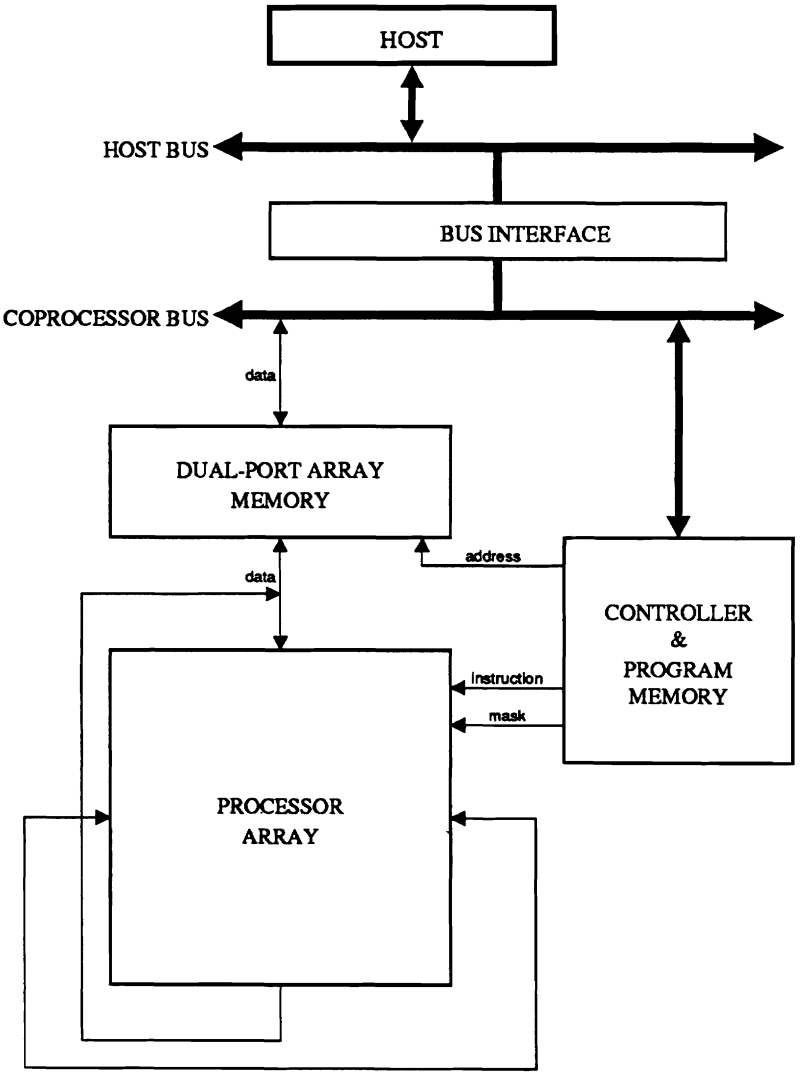

FIGURE 2.0.1 Systolic/Cellular System Diagram

the **WRITE port**. General data flow from memory through the **Array** and back to memory is in a North to South direction, however within the **Array** itself, data can flow northward, southward, eastward or westward.

The **Array** is controlled exclusively by the **Controller**. Each processor receives its instructions, enable/disable information, and the system, multiplier, and divider clocks from the **Controller**. There is no control logic in the processing element.

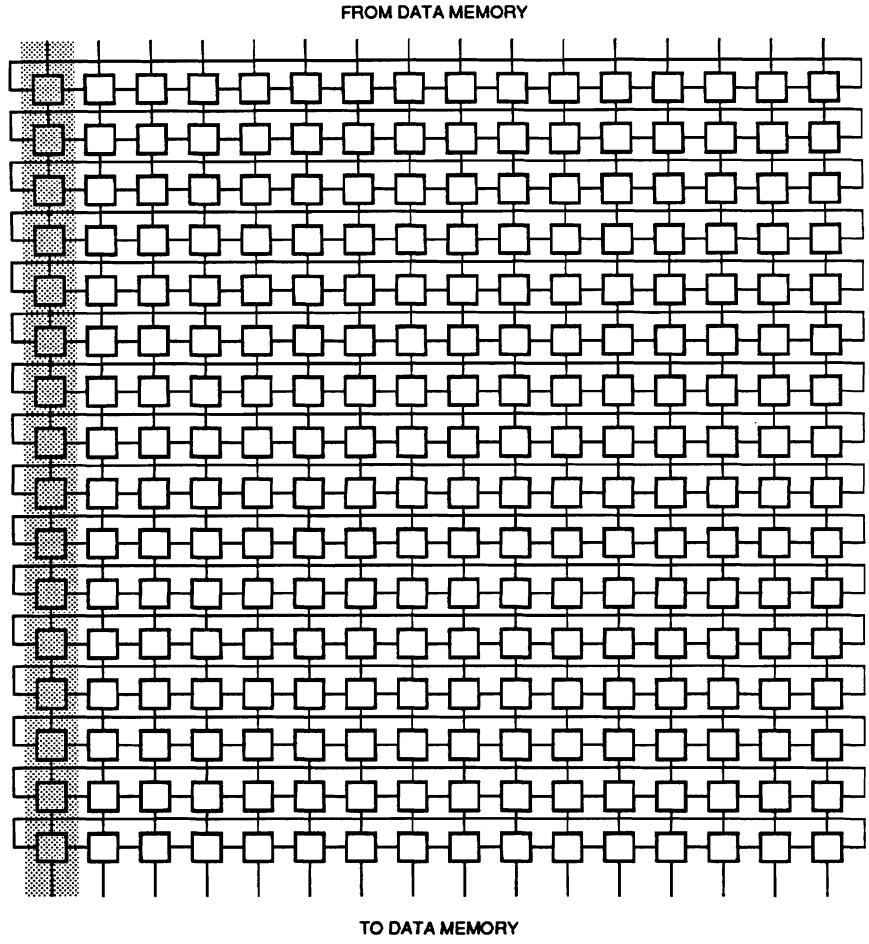

FIGURE 2.1.1 The Processor Array

The **Array** is divided into two sets of processors, **External** and **Internal**. The **External** processors[3] are those in column 1 of the **Processor Array** shown in the shaded area of Figure 2.1.1. The PEs in columns 2–16 are the **Internal** processors. Each set of processors receives different operation codes (opcodes) from the **Controller**. These codes are found in different fields of the program instruction enabling two sequences of instructions to run in parallel on the **Array**. These sequences may be the same or different, but they are not independent. Both are bound by the same system instructions, such as program flow, memory access and masking. Masking enables the user to choose which processors will be enabled for

[3] Sometimes referred to as **Boundary** processors in other documentation.

any given instruction, but not all effects of the program instruction can be disabled. Masking is described in Sections 3.1.2 and 4.3 .

2.2 The Data Memory

The **Data Memory** has two ports, the **WRITE** (or TOP) port which is accessed by the HOST and row 16 of the **Processor Array**, and the **READ** (or BOTTOM) port which is accessed only by row 1 of the **Processor Array**. The HOST can access **Data Memory** to load and unload data only when the Co-processor is in HALT mode. The **Array** can access **Data Memory** only when the Co-processor is in RUN mode. Associated with each port is an Address Counter and an Address FIFO. Each Address FIFO can hold up to 512 addresses.

The **Data Memory** stores data in rows of 16 32-bit words and holds up to 2048 rows of data. The system is designed for *structured data programming.* The basic data structure is a queue of data rows. A queue is chosen by loading the address of the head of the queue from the port's FIFO to its corresponding Address Counter. The data is then accessed sequentially; the Address Counter automatically increments or decrements the address depending on whether the queue is ascending or descending. Another allowable data structure is a single row which is accessed repeatedly. The type of data structure, ascending or descending queue or single row is encoded in the two most significant bits of the address stored in the FIFO. The FIFOs are not overwritten during program execution, so if the program is to be used multiple times, the head pointers can be reset by the HOST to the first item in the FIFO without reloading the entire FIFO from the HOST.

2.3 The Controller

The **Controller** is in charge of running and monitoring the Co-processor. It receives signals from the HOST, reads selected bits from the program instruction, and monitors some system flags. It maintains a status register which is accessible by

the HOST. This status registers tells the states of the FIFOs (full/empty/neither) and operation mode of the Co-processor (RUN/HALT). The **Controller** maintains the three global clocks, and operates the program sequencer.

For details on the **Controller**'s interaction with the HOST, see [WOJ88].

From the SYSTEM FIELD of the program instruction (See Section 3.1) the **Controller** determines whether to put the system into HALT state, to start one or both fast clocks, to load an address counter, or to enable memory access.

The **System Clock** is an 8 MHz two-phased, nonoverlapping, asymmetric clock. Its cycle, *the instruction cycle*, is a basic unit of the overall system operation. The system clock is in continual operation during the power-on state of the Co-processor.

Both the **Multiplier Clock** and the **Divider Clock** are two-phased, nonoverlapping, asymmetric clocks, each controlled by a bit in the SYSTEM field of the user program. The **Multiplier Clock** has a frequency of 22 MHz. In the instruction cycle following the execution of an instruction whose **MULTIPLY** bit is set, the **Multiplier Clock** starts, produces 17 cycles of pulses (the number necessary to perform a multiplication operation), and then stops. The **Divider Clock** has a frequency of 17 MHz. In the instruction cycle following the execution of an instruction whose **DIVIDE** bit is set, the **Divider Clock** starts, produces 30 cycles of pulses (the number necessary to perform a division), and then stops.

The program sequencer maintains the program counter (PC), loads the next program instruction into the instruction register, and sends the appropriate bits to the **Program Array**. The PC is always incremented unless a new address is loaded from the Program FIFO. Program execution is started by the HOST loading the starting address from the FIFO and then setting the system into RUN mode. When an instruction with the **STOP** bit set is encountered, program execution stops and the Co-processor is put into HALT mode. The address of this instruction remains

in the PC and the last instruction remains in the instruction register.

2.4 The Processing Element

All 256 PEs in the **Array** are identical. Each contains 24 static random access 32-bit registers of local memory, four bidirectional I/O ports, and seven functional units; there are two busses to interconnect them all. See Figure 2.4.1. There is no control logic in the Processing Element. The PEs are hard-wired and are controlled by signals from the **Controller**. These signals include the System Clock; the Opcode–one for each phase of the System Clock cycle; the appropriate mask bits–determines whether the PE is enbled or not; the Multiplier Clock; and the Divider Clock.

The two busses are labeled BUS A and BUS B. Of the 24 registers, eight are connected to only BUS A, eight only to BUS B, and the rest are connected to both busses. The seven functional units include two adders, two multipliers, a sorter, a shifter and a divider. See Figure 2.4.1. The functional units implement *arithmetic operations* which require two operands each. These operands are loaded simultaneously, one via BUS A and the other via BUS B. The functional units can all run simultaneously, but only one can be loaded at a time, therefore practically, only those whose execution time is longer then a single cycle can run in parallel with another functional unit in the same processor. For single cycle functional units, execution is initiated as soon as the operands are loaded. For multi-cycle functional units, execution is initiated when its fast clock starts. The results of the functional units are stored in dynamic output registers. The **Sorter** and the **Shifter** each have two results; one is stored in an output register connected to BUS A and the other is stored in an output register connected to BUS B. The **Divider** outputs the quotient to an output register connected to BUS A. The **Adders** output the sum to one bus and the ones complement of the sum to the other bus.

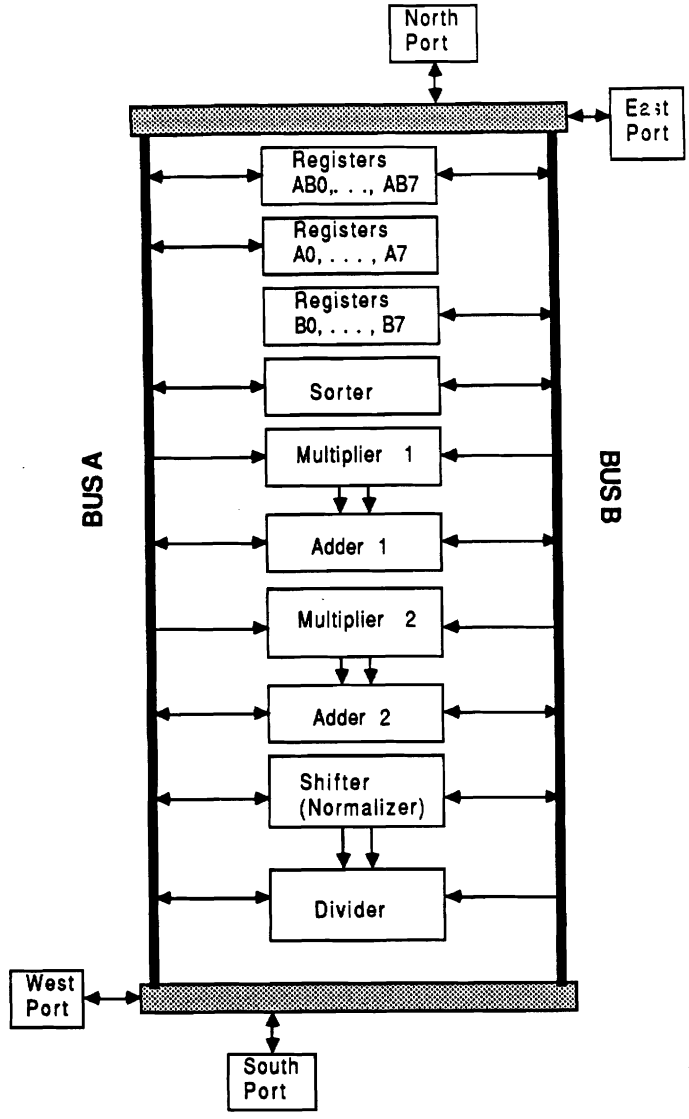

FIGURE 2.4.1 Processing Element Diagram

Some arithmetic operations, i.e. multiplication and division, may require the sequential use of two functional units to acquire the final result. Multiplication requires use of the Multiplier then the Adder, so the output registers of the Multipliers are connected directly to the corresponding Adders, the product (and its ones complement) are available from the Adder outputs. Division may require its inputs to be normalized before executing division, so the inputs can first be passed through the Shifter, before being sent to the Divider. Therefore, the outputs of the Shifter

are connected directly to the Divider as well as to the busses. More details on the functional units may be found in Chapter 4where the opcodes of these functions are defined.

The functional units operate on 32-bit, fixed-point, signed values, where the bit 31 (msb) is the sign bit, bit 30 is the integer bit, and bits 0–29 are the fractional part. The format is as follows:

SB INT [.] F F F F F F F F F F F F F F F F F F F F F F F F F F F LSB

Where:

SB –(msb) sign bit ($0 \Longrightarrow$ a positive number, $1 \Longrightarrow$ a negative number).

INT –integer part of the number.

[.] –decimal point.

F –bits of the fractional part of the number.

LSB –least significant bit of the fractional part of the number.

These discrete values are 2^{-30}'s complement quantities which fall in the range -2 to $2 - 2^{-30}$.

Chapter 3
Programming the Co-processor

A program for the Systolic/Cellular System consists of four parts: the machine code program; the contents of the program FIFO, for program control; the contents of the two data memory port FIFOs, for data access control; and the actual data. The Assembler will generate, the first three parts automatically from the assembly program.[4] *Regular Instructions* will generate machine instructions[5] and *Special Instructions* will generate the FIFO's and set bits in the SYSTEM FIELD of the machine instruction. These two types of instructions are described in depth in the next chapter.

3.1 The Machine Instruction

The machine instruction is a 112-bit binary value divided conceptually into seven fields of 16 bits each. There are two Mask Fields, two fields for Internal processor instructions, two fields for External processor instructions, and one field for system control operations. The format of the machine instruction can be found on the next three pages. Only 99 bits are actually used, unused bits are named "X". The bits are numbered from the least significant bit to the most as well as its position in its 16-bit field. The machine instructions are divided into subfields. For each Phase there is an I/O part (4 bits) and a SOURCE-DESTINATION part (10

[4] In future versions, the Assembler should be able to generate a data file containing initialization data.

[5] machine instructions are also called program instructions.

bits). When executing the instruction, only the SOURCE-DESTINATION part of the processor instruction is affected by the mask.

3.1.1 Masking

It is not necessary for every processor to execute every instruction. The processors are enabled by signals from the two mask fields of the machine instruction. There are two types of masking modes available: **row/column**, and **diagonal**. The two modes may not be combined in a single instruction. A single mask is valid for both **External** and **Internal** processors during both Phases of the clock cycle.

Row/column masking enables the processors which lie on the intersection of the selected rows and columns, where a 0 means selected. The **row** field consists of bits 0-15 of the machine instruction and the **column** field consists of bits 16-31 of the machine instruction. For **row/column** masking, bit-43 (SEL D/RC) is set to 0. See the first page of the machine instruction format. The rows are numbered 1 through 16 from top to bottom and the columns are numbered 1 through 16 from left to right, see Figure 3.1.1. **Row/column** masking mode allows the user to enable rectangular regions on the array, as many as desired with one restriction: ALL intersections of ALL the rows and columns selected are enabled. See Section 4.3 example 2.

Diagonal masking enables the processors which lie on the selected diagonals of the array, where a 0 means selected. The **diagonal** field consists of bits 0-30 of the machine instruction; bit 31 of the higher order mask field is not used. For **diagonal** masking, bit-43 (SEL D/RC) is set to 1. See the first page of the machine instruction format. The diagonals run northwest to southeast and are numbered 1 through 30 from the northeast corner to the southwest corner, see Figure 3.1.2. **Diagonal** masking mode allows the user to enable any combination of diagonal bands of processors. See Section 4.3 example 3.

Instr Bit	Field Bit	Bit Name	Subfield	Field
0	0	R1/D1	ROW	MASK
1	1	R2/D2	MASK	
2	2	R3/D3		
3	3	R4/D4		
4	4	R5/D5		
5	5	R6/D6		
6	6	R7/D7		
7	7	R8/D8		
8	8	R9/D9		
9	9	R10/D10		
10	10	R11/D11		
11	11	R12/D12		
12	12	R13/D13		
13	13	R14/D14		
14	14	R15/D15		
15	15	R16/D16		
16	0	C1/D17	COLUMN	
17	1	C2/D18	MASK	
18	2	C3/D19		
19	3	C4/D20		
20	4	C5/D21		
21	5	C6/D22		
22	6	C7/D23		
23	7	C8/D24		
24	8	C9/D25		
25	9	C10/D26		
26	10	C11/D27		
27	11	C12/D28		
28	12	C13/D29		
29	13	C14/D30		
30	14	C15/D31		
31	15	C16/X		

X - stands for an unused bit.

Instr Bit	Field Bit	Bit Name	Subfield	Field
32	0	S1	OP-CODE	INTERNAL
33	1	S2		INSTRUCTION
34	2	S3		PHASE 1
35	3	S4		
36	4	S5		
37	5	D1		
38	6	D2		
39	7	D3		
40	8	D4		
41	9	D5		
42	10	X		
43	11	SEL D/$\overline{RC}$		
44	12	$\overline{I/O}$	I/O CODE	
45	13	$\overline{A/B}$		
46	14	$\overline{RS}$		
47	15	$\overline{LS}$		
48	0	S1	OP-CODE	INTERNAL
49	1	S2		INSTRUCTION
50	2	S3		PHASE 2
51	3	S4		
52	4	S5		
53	5	D1		
54	6	D2		
55	7	D3		
56	8	D4		
57	9	D5		
58	10	$\overline{X}$		
59	11	$\overline{X}$		
60	12	$\overline{I/O}$	I/O CODE	
61	13	$\overline{A/B}$		
62	14	$\overline{RS}$		
63	15	$\overline{LS}$		

X - stands for an unused bit.

Instr Bit	Field Bit	Bit Name	Subfield	Field
64	0	S1	OP-CODE	EXTERNAL
65	1	S2		(BOUNDARY)
66	2	S3		INSTRUCTION
67	3	S4		PHASE 1
68	4	S5		
69	5	D1		
70	6	D2		
71	7	D3		
72	8	D4		
73	9	D5		
74	10	X		
75	11	X		
76	12	I/O	I/O CODE	
77	13	A/B		
78	14	RS		
79	15	LS		
80	0	S1	OP-CODE	EXTERNAL
81	1	S2		(BOUNDARY)
82	2	S3		INSTRUCTION
83	3	S4		PHASE 2
84	4	S5		
85	5	D1		
86	6	D2		
87	7	D3		
88	8	D4		
89	9	D5		
90	10	X		
91	11	X		
92	12	I/O	I/O CODE	
93	13	A/B		
94	14	RS		
95	15	LS		

X - stands for an unused bit.

Instr Bit	Field Bit	Bit Name	Subfield	Field
96	0	STOP		SYSTEM
97	1	DIVIDE		OPERATION
98	2	MULTIPLY		
99	3	LOAD PC		
100	4	WRITE		
101	5	LD WRITE ADDR		
102	6	READ		
103	7	LD READ ADDR		
104	8	X		
105	9	X		
106	10	X		
107	11	X		
108	12	X		
109	13	X		
110	14	X		
111	15	X		

X - stands for an unused bit.

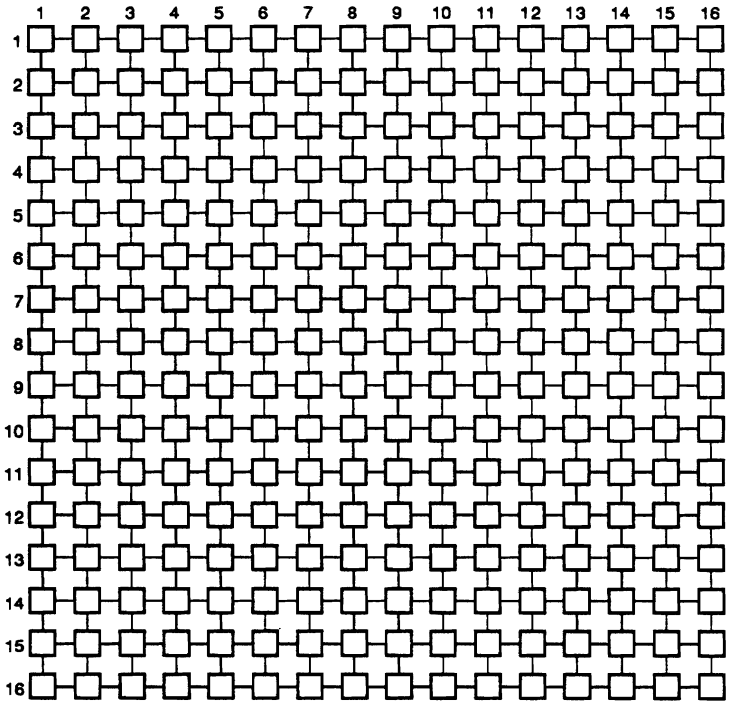

FIGURE 3.1.1 Rows and Columns of the Processor Array

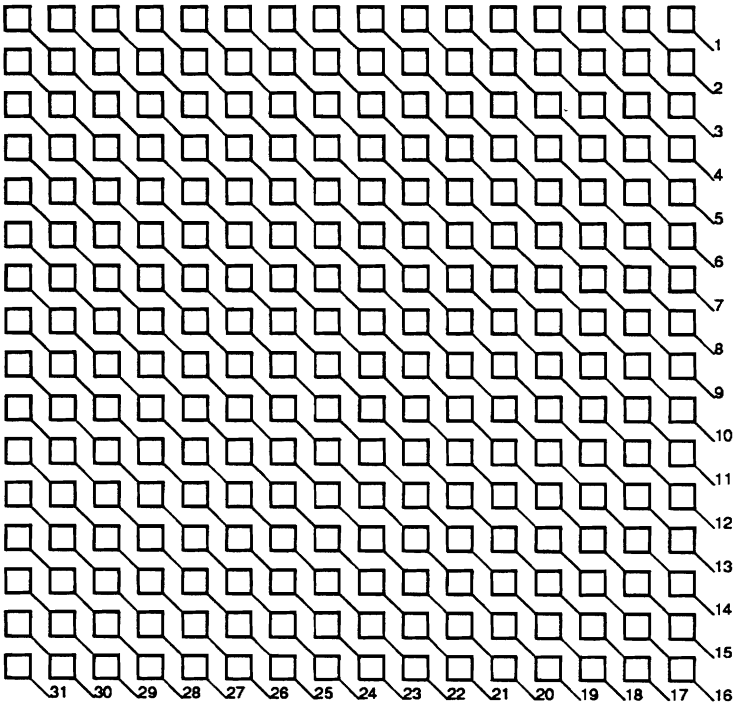

FIGURE 3.1.2 Diagonals of the Processor Array

The masks do not effect every part of the machine instruction. Only execution of the SOURCE-DESTINATION codes (for both Phases in both sets of processors) is masked. The I/O instructions always execute in every processor in the set. The fast clocks, when enabled, effect every processor in both sets. See Sections 4.1.3 4.1.2 , and 4.1.6 on Inter-processor Communication, Division and Multiplication and the NOTE in section 4.3 .

3.1.2 External and Internal Processors

The processors in the **Array** are divided into two sets. The **External** processors (sometimes called **Boundary** processors in other documentation) consist of the processors in column 1; the rest are called **Internal** processors. The two sets have their own sequences of instructions, although the two sequences are not 100% independent, both are effected by a single mask and by the actions initiated by bits set in the SYSTEM field of the machine instruction. For example, if the mask is such that only row 1 is enabled, then the top External processor will execute the External instruction and the top row of Internal processors will execute the Internal instruction. If a fast clock is started, it affects all processors in both sets. etc.

The instructions for each set are located in different fields of the program instruction and may be different or the same. For Internal Instructions, the Phase 1 field consists of bits 32-47 and the Phase 2 field consists of bits 48-63. For External Instructions, the Phase 1 field consists of bits 64-79 and the Phase 2 field consists of bits 80-95. See pages two and three of the instruction format above. The Phase fields can be further subdivided into subfields, the SOURCE-DESTINATION opcode, bits 0-9 of the field and the I/O code, bits 12-15 of the field. The SOURCE-DESTINATION codes control data transfers within the processor. Bits labeled S_1 through S_5 denote the source register with S_1 being the least significant bit (lsb). Bits labeled D_1 through D_5 denote the destination register or functional unit with

D_1 being the lsb. The I/O code controls data transfer between processors, and between processors and Memory.

3.1.3 System Operation Field

The 8 utilized bits in the SYSTEM field control interaction between different parts of the Systolic/Cellular System. They consist of bits 96-103, where a 0 means enable the operation. The LOAD bits control loading the address and program counters from the FIFOs; the READ and WRITE bits control memory access; the MULTIPLY and DIVIDE bits start the respective fast clocks, and the STOP bit places the Co-processor in HALT mode. See the fourth page of the instruction format above.

3.2 Assembly Program File

From the Assembly Program, the Assembler must construct the machine code program and the three fifos. The format of the Assembly Program is as follows:

Data Queue Definitions (*mandatory*)

Mask Definitions (*optional*)

Program Body (*mandatory*)

The Assembly Directive: END; (*mandatory*)

Comments may appear anywhere in the file. There are two types and formats for comments. Comments may appear anywhere in the code (even within an instruction) if it lies between curly brackets "" and ""; it may extend over more than one line of the file. Comments may also appear anywhere between a SEMICOLON (the end-of-assembly-instruction symbol) and an end-of-line character (a ¡cr¿). The user should take care not to use the SEMICOLON in the middle of an assembly instruction.

3.2.1 Queue Definitions

The user must know where his data lies in memory in order to access it. Currently, (May 1988) the Assembler generates memory addresses from the size of the data queues defined. Future versions of the Assembler will require that the user explicitly give the memory addresses, which will allow more flexibility in data access as well as allowing more programs with different data to be resident in the processor at any given time, and shared data between programs.

The format of queue definitions which is accepted by the current implementation is as follows:

DEFQUEUE QNAME SIZE

where QNAME is a user chosen alphanumeric string (beginning with a letter, and where all letters are capitalized), and SIZE is a positive or negative integer $\neq 0$. If SIZE has magnitude of 1, it is considered a single element data structure whose address remains constant until the next one is loaded. If SIZE < 0 then the data queue is a descending queue. If SIZE > 0 then the data queue is an ascending queue.

The format of the future queue definition may look as follows:

DEFQUEUE QNAME STARTADDR ENDADDR VALLIST

where STARTADDR and ENDADDR are actual memory addresses, STARTADDR points to the head of the data queue and ENDADDR to its tail. If STARTADDR = ENDADDR then the data structure is considered to be single element. Otherwise the relationship between STARTADDR and ENDADDR determine the direction of the queue. VALLIST will be an optional list of initialization data whose format is currently undetermined. This data would be loaded to **Data Memory** every time the program is loaded into the Co-processor.

The starting address and the type of queue is stored in a symbol table with

the QNAME for reference later in the program, when QNAME is used in READQ and/or WRITEQ instructions.

3.2.2 Mask Definitions

Due to possible frequent use of specific masks throughout a program, it was decided to allow a user to associate a mask with an identifier and allow the identifier to be used in the program body instead of the full mask syntax. This construct will also simplify the task of changing a mask if necessary. The format of the mask definition is as follows:

DEFMASK MNAME MASK

where MNAME is an alphanumeric string as defined above, and MASK conforms to the syntax described in Section 4.3 , (RList:CList:DList).

3.2.3 Program Body

The program body consists of a sequence of Instructions, where each Instruction is followed by a SEMICOLON (;), this allows an Instruction to be spread out over more than one line of the file, if desired. An Instruction can have one of nine forms:

1 LABEL: EXT-OPCODE INT-OPCODE THEMASK

2 LABEL: EXT-OPCODE INT-OPCODE

3 EXT-OPCODE INT-OPCODE THEMASK

4 EXT-OPCODE INT-OPCODE

5 LABEL: OPCODE THEMASK

6 LABEL: OPCODE

7 OPCODE THEMASK

8 OPCODE

9 SPECIAL-INSTRUCTION

LABEL is an alphanumeric string, and if present, must be followed by a colon. EXT-OPCODE and INT-OPCODE refer to the opcodes for the External and Internal processors respectively. THEMASK can be either a Mask Identifier as defined at the beginning of the user file in a DEFMASK command, or a MASK as described in Section 4.3 . OPCODE can be used to signal SIMD mode, i.e. when the External and Internal processors will execute the same instruction. Instructions 5–8 are shorthand notation for the special case of 1–4 where EXT-OPCODE is identical to INT-OPCODE. The SPECIAL-INSTRUCTIONs are described in Section 4.4.

Instructions 1 through 8 above are classified as **Regular Instructions** and instruction 9 is classified as a **Special Instruction.** The differences are described in Chapter 4. One thing to note is that the **Special Instruction** does not take either a LABEL or a Mask.

3.3 Assembler's output file

If assembly succeeds with no errors, then the Assembler outputs the generated code and the Label, Queue, and Mask symbol tables which can be useful when running the code on the Simulator/Debugger [SHI88]. The output file format is as follows:

Magic Number (16 bits) (LSB = 023_8 MSB = 007_8)

Length of Program in # of Machine Instrs (16 bits)

Program

Length of Program-FIFO (16 bits)

Program-FIFO

Length of Write Address-FIFO (16 bits)

Write Address-FIFO

Length of Read Address-FIFO (16 bits)

Read Address-FIFO

Data * not implemented yet.

Label Symbol Table

Queue Symbol Table

Mask Symbol Table

The **Magic Number** is used as a file ID; any input file to the Loader of the Co-processor or to the Simulator/Debugger should begin with these two bytes in the correct order. Since the VAX and the SUN have different byte ordering, if the **Magic Number** is read in the reverse order, then it can be assumed that the file was generated from an Assembler on the other machine. The Loader and Simulator can check for this and swap the bytes on the relevant words if necessary. The third and forth bytes together give the size of the **Program**, in number of Machine Instructions (112-bit words).

The **Program** format is as follows:

Row Mask Fields (Program-Length × 16-bits)

Col Mask Fields (Program-Length × 16-bits)

Internal Instruction f_1 Fields (Program-Length × 16-bits)

Internal Instruction f_2 Fields (Program-Length × 16-bits)

External Instruction f_1 Fields (Program-Length × 16-bits)

External Instruction f_2 Fields (Program-Length × 16-bits)

System Operation Fields (Program-Length × 16-bits)

Each FIFO is listed in Head to Tail order.

Each Table is of the form:

Hash-value (8 bits)

Item-List

MARK (8 bit value ¿ largest hash value)

Each item in an Item List contains an eight-bit string length, a string and the value attributed to that string. For the Label and Queue Address Tables, the

value consists of a 16-bit address. For the Mask Table, the value consists of a character depicting Mask type (8 bits), the Low Order (Row) Mask (16 bits) and then the High Order (Column) Mask (16 bits). Every Item List is followed by a MARK which is a number greater than the highest hash value. Each Table is followed by another MARK.

The Loader to the Co-processor and the Simulator/Debugger will be designed to read in this format.

3.4 Running the Assembler

To run the Systolic/Cellular System Cross-Assembler, you issue the command to the shell:

xscs file1 [file2]

where **file1** is the name of the input file and the optional **file2** is the name to assign to the output file. If **file2** is not specified, then the output file is named *scs.exe*.

Future versions of the Assembler will generate the symbol tables at the end of output file only if it was run with a **-g** switch. The current Assembler (May 1988) always outputs the tables.

3.5 Programming the Systolic/Cellular SYSTEM

When programming the Systolic/Cellular System the user needs more than just the Co-processor program as described above. It is necessary to use an interface program in the HOST.[6] For a short simple program, this is simple; you need to download the program, FIFOs, and data and upload the results when execution is complete. But it is possible to run longer programs, programs greater than will fit in program memory and/or with FIFOs that are too long. It is also possible to run programs with more data than will fit in the Array Memory. To do this, the

[6] To date this has not been implemented.

program must STOP in the middle and allow the HOST to load the Co-processor. There should be a program that will do this automatically for the user, possibly requiring extensions to the Assembler. However, the user should take care that if the program will be overwritten, that the STOP does not occur within a loop and that all necessary outputs of functional units are stored before halting the system.

Chapter 4
Assembly Code: Opcodes

Below is a list of the assembly codes recognized by the SCS cross-assembler. The *opcodes* can be divided into 2 major categories:

Regular instructions: those which translate into one or more lines of *machine code*,[7] These are listed in section 4.1.

Special instructions: those which only affect the system control field of the program instruction. These are also sometimes referred to as *Control instructions*. These are listed in section 4.4.

There are two types of registers, static and dynamic, described in section 4.2. Source registers and operands to Arithmetic operations may be of either type. Destination registers on the other hand, may only be of the static type. Register names with their corresponding opcodes are also listed in section 4.2.

Regular instructions may be masked so that only a specified set of the processors execute the operation. The mask format is described in section 4.3.

4.1 Regular Operation Codes

There are three types of regular instructions: arithmetic operations; internal (to the processing element (PE)) memory storage; and communication between processors.

[7] Each line of machine code, a 112-bit binary value, is also called a ***program instruction*** or a ***machine instruction***. See Section 3.1.

An arithmetic operation is one that utilizes one of the *functional units* of a processing element. A functional unit takes 2 operands, either one via each bus, or both from another functional unit, and outputs to registers accessible to one or both busses and/or directly to another functional unit. These output registers are **dynamic**, meaning that the values stored within decay with time. The values stored in these **dynamic** registers are valid up to 5 cycles. See Section 4.2 for more details. There are three types of arithmetic operations:

1. operations with no operands consisting mainly of *stage 2* operations with input from another functional unit;
2. operations with 2 operands; and
3. operations with 4 operands (input into 2 functional units).

The other two types of operations involve data movement, inside the processing element and outside, and are described in the appropriate subsections below.

There are two timing considerations; both are in terms of *cycles*. A *cycle* or *instruction cycle* refers to the time it takes for the co-processor to interpret one 112-bit machine instruction. An *instruction cycle* is made up of 2 phases labeled f_1 and f_2. During each phase, a different part of the operation is executed, thus the code for each phase is situated in a different field of the machine instruction. One timing consideration is the number of machine instructions into which a given operation translates. This can be noted by the number of f_1, f_2 pairs there are in the code description. Each f_1, f_2 pair (i.e. machine instruction) is separated by a heavy horizontal line. The other timing consideration is the time (in instruction cycles) it takes to complete the execution of an operation and is noted for each operation after the words *Execution Cycles*. This value is greater than or equal to the number of machine instructions generated.

The codes are listed below in alphabetical order of function.

4.1.1 Addition

There are two types of addition available, single addition using **Adder 2**, and double addition using both **Adder 1** and **Adder 2** on the same operands. The advantage to double addition is that both the result and its one's complement are accessible from both busses.

Multiplier 2 in the same processors must be stored in a static register before executing the second stage of multiplication, else the sum will be lost.

Double Addition

Mnemonic Code: **ADDD (X,Y)**

Execution Cycles: **1**

Operands:

X any source register accessible by BUS A.

Y any source register accessible by BUS B.

Machine Code:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 1 1 1	1 1 1 1 1	X
f_1	1 1 1 1	1 1 0 1 0	Y

Results: available in the next cycle following the cycle in which the unit is loaded

SUM1A Accessible from BUS A. The sum of the values stored at registers X and Y. It is identical to the register named PROD2B, see section 4.1.6 on Multiplication.

SUM2B Accessible from BUS B. The sum of the values stored at registers X and Y. It is identical to the register named PROD2B, see section 4.1.6 on Multiplication.

CSUM1B Accessible from BUS B. The ones complement of the sum of the values stored at registers X and Y. It is identical to the register named CPROD2A, see section 4.1.6 on Multiplication.

CSUM2A Accessible from BUS A. The ones complement of the sum of the values stored at registers X and Y. It is identical to the register named CPROD2A, see section 4.1.6 on Multiplication.

Comments:

Moves the data stored in X via BUS A and Y via BUS B to **Adder 1** and **Adder 2** which perform the addition operation simultaneously in a single cycle. The sum from **Adder 1** is stored in

the dynamic register SUM1A and its one's complement in the dynamic register CSUM1B. The sum from **Adder 2** is stored in the dynamic register SUM2B and its one's complement in the dynamic register CSUM2A.

NOTE:

Since the second stage of multiplication in **Multiplier 1** and **Multiplier 2** use **Adder 1** and **Adder 2** respectively, SUM2B, and CSUM2A get overwritten when executing a MULTS2 in the enabled processors, and SUM2B, CSUM2A, SUM1A, and CSUM1B get overwritten when executing a MULTSD in the enabled processors. Therefore, any sum generated by an ADDD during execution of multiplication in the same processors, must be stored in a static register before executing the second stage of multiplication, else the sum will be lost.

4.1.2 Division

The **Divider** requires its operands to be in a normalized format i.e. the divisor (Input from BUS A) must be in the range $1 \leq X < 2$.[7] To ensure the correct format, the input data should pass through the **Shifter**[8] before being passed to the **Divider**. This can be implemented in a two-stage division; the first stage normalizes the divisor to its correct format, shifting the dividend simultaneously. The second stage loads the normalized values directly from the **Shifter** and performs the division. See DIVF and DIVS below. Alternatively, if the User knows that the divisor is in the correct format, then the inputs may be loaded directly to the **Divider**. See DIV below.

Division takes a single cycle to load the data but 10 cycles total to execute. The result of this is that the quotient may not be accessed until the end of execution, but other operations may run in the interim.

There is only a single **Divider clock** for both Internal and External processors. For division to run on both sets of processors, the dividers must be started simultaneously, because once the clock has started, another division may not commence until the clock is free. Once the clock has started, the value stored in the dynamic output register of the divider is affected, therefore the result of the old division must be used or saved in a static register before the next division commences, else the value will be lost. The clock inputs to the processors are NOT masked, this means that once the clock starts, EVERY processor's divider output will be affected, internal and external, enabled and disabled.

---

[7] See Section 2.4 on data type.

[8] In other documentation, it is sometimes referred to as the **Normalizer** because of this function.

Division—Normalized Operands

Mnemonic Code: **DIV (X,Y)**

Execution Cycles: **10**

Operands:

X any source register accessible by BUS A containing a value greater than or equal to 1 and less than 2. The **divisor.**

Y any source register accessible by BUS B. The **dividend.**

Machine Code:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 1 1 1	1 1 1 1 1	X
f_1	1 1 1 1	1 1 1 1 0	Y

Enables the **DIVIDE** bit in the system field of the current instruction by setting bit 97 to 0.

Results: available in the 10 th cycle following the cycle in which the unit is loaded

QUOTA The quotient of Y ÷ X. Accessible from BUS A.

Comments:

Moves the (assumed normalized) data stored in X via BUS A and the data stored in Y via BUS B to the **Divider** and starts execution of division Y ÷ X (by starting the **Divider clock**). After 10 cycles, the quotient is stored in the dynamic register QUOTA.

NOTE:

There is only a single **Divider clock** for both Internal and External processors. Thus once the clock has started, another division may not commence, even in a different set of processors, until the current one has been completed.

The **Divider clock** changes the value stored in QUOTA, from its first pulse, therefore, the old value of QUOTA must be used or

saved before the **Divider clock** is restarted (as well as before the old value decays).

The **Divider clock** is NOT masked, therefore the QUOTA registers in EVERY processor (internal and external, enabled and disabled) is changed.

Division—Non-Normalized Operands: Stage 1

Mnemonic Code: **DIVF (X,Y)**

Execution Cycles: **1**

Operands:

X any source register accessible by BUS A. The **divisor.**

Y any source register accessible by BUS B. The **dividend.**

Machine Code:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 1 1 1	1 1 1 1 1	X
f_1	1 1 1 1	1 1 1 0 1	Y

Results: available in the next cycle following the cycle in which the unit is loaded

SHIFTA The shifted value of X, normalized to be a value greater than or equal to 1 and less than 2. Dynamic register accessible from BUS A.

SHIFTB The shifted value of Y. Dynamic register accessible from BUS B.

The values stored in SHIFTA and SHIFTB are also directly accessible by the **Divider.**

Comments:

Moves the data stored in X via BUS A and the data stored in Y via BUS B to the **Shifter** and executes normalization, by shifting both values *left* until the X input falls in the range greater than or equal to 1 and less than 2. The shifted X value is stored in dynamic register SHIFTA, and the shifted Y value is stored in dynamic register SHIFTB. Both SHIFTA and SHIFTB are also connected directly to the **Divider.**

NOTE:

A DIVF is usually followed (within 5 cycles) by a DIVS opera-

tion. Examples:

```
DIVF (X,Y)
DIVS
```

or:

```
DIVF (X,Y)
NOP
NOP
NOP
NOP
DIVS
```

The NOPs above may be replaced by any operations that do not involve the **Shifter**, else the values will be overwritten.

The outputs SHIFTA and SHIFTB are not independent, a single access code accesses both values simultaneously. This is a hardware or firmware restriction. A single code accessing these registers ties up both busses. The limiting result of this is that when moving the value of one of these outputs to a static register or as input to another functional unit, there may not be any other data movement. For example: (let $n, m = 1, 2, \ldots, 7$).

The following are examples of **valid** operations:

MOV(SHIFTA,An:SHIFTB,Bn)

MOV(:SHIFTB,Bn)

MOV(SHIFTA,ABn:)

ADD2(SHIFTA,SHIFTB)

DIV(SHIFTA,SHIFTB) *identical operation as DIVS.*

The following are examples of **invalid** operations:

MOV(SHIFTA,An:Bn,ABn)

MOV(An,Am:SHIFTB,ABn)

MULTS1(SHIFTA,SUMB2)

Division—Non-Normalized Operands: Stage 2

Mnemonic Code: **DIVS**

Execution Cycles: **10**

Operands:

Normalized operands come directly from dynamic output registers of the **Shifter**.

Machine Code:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 1 1 1	1 1 1 1 0	1 1 1 1 1
f_1	1 1 1 1	1 1 1 1 1	1 1 1 1 1

Enables the **DIVIDE** bit in the system field of the current instruction by setting bit 97 to 0.

Results: available in the 10 th cycle following the cycle in which the unit is loaded

QUOTA The quotient of Y ÷ X, where X and Y are the inputs to the **Shifter** in a previous SHIFT(X,Y) or DIVF(X,Y) command. Accessible from BUS A.

Comments:

Moves the data, the normalized dividend and divisor, stored in the dynamic output register of the **Shifter** to the **Divider** and starts execution of division (by starting the **Divider clock**). After 10 cycles, the quotient is stored in the dynamic register QUOTA.

Example:

```
DIVF (X,Y)
DIVS
NOP
NOP
NOP
```

```
NOP
NOP
NOP
NOP
NOP
NOP
MOV (QUOTA,A4:)
```

The NOPs above may be replaced by any operations that do not involve the **Divider** or the register QUOTA. Once the **Divider** has been loaded, the **Shifter** is available.

NOTE:

The **Shifter** must have been executed within 5 cycles previously in order for **Divider** to have a reliable output.

There is only a single **Divider clock** for both Internal and External processors. Thus once the clock has started, another division may not commence, even in a different set of processors, until the current one has been completed.

The **Divider clock** changes the value stored in QUOTA, from its first pulse, therefore, the old value of QUOTA must be used or saved before the **Divider clock** is restarted (as well as before the old value decays).

The **Divider clock** is NOT masked, therefore the QUOTA registers in EVERY processor (internal and external, enabled and disabled) is changed.

4.1.3 Inter-Processor Communication

There are four types of inter-processor communication within the processor array, East to West, North to South, South to North, and West to East. Because of pin limitations, it is possible to physically move only 16 bits at a time between processors, therefore, since the data is 32 bits long, I/O requires execution of two machine instructions. Execution is in two parts, moving data to and from the ports and moving data between processors. The code for the former is located in the source/destination parts of the f_1 and f_2 fields of the machine instruction, the latter is located in the I/O parts. Due to complex issues regarding the I/O ports and the difference in I/O code for communication with memory access and communication without memory access, it was decided that inter-processor communication should require use of a single instruction by the programmer.

The user decides which processors should *receive* data, and those are the ones which are masked. **GETE(S,D)**, for example, means each enabled processor will receive the value stored in its east neighbor's register S and place it in its own register D. From start to finish this works as follows:

1. **ALL** processors move the data stored in register S (via either bus)[9] to their WEST output port.
2. Within two cycles, data is moved from the WEST port of **EACH** processor to the EAST port of its west neighbor.
3. The **enabled** processors move data stored in their EAST ports to the destination register D (via either bus).[8] The other instructions, including communication with memory access work in the same manner (but with different ports).

The side effects of this implementation is that the mask is not applied to

[9] The assembler determines which bus.

the first machine instruction generated. Thus if the *External* instruction is Communication and the *Internal* is not, or vice versa, and the instruction is masked, then the programmer should note that the first instruction generated for the non-communication operation will be executed by **all** processors. If it translates to more than one machine instruction, the second, etc. instructions will be executed by **only** the enabled processors. For more details on this and other side effects, see the next chapter on Limitations.

East To West Data Flow

Mnemonic Code: **GETE (S,D)**

Execution Cycles: **2**

Parameters:

S any source register of the **source processor** accessible by either BUS A or BUS B

D any static register of the **destination processor** accessible by either BUS A or BUS B

Machine Code:

S via BUS A to West port and East port to D via BUS A:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 0 0 0	1 0 0 1 0	S
f_1	0 0 1 0	1 1 1 1 1	1 1 1 1 1
f_2	0 1 1 0	D	1 0 1 0 0
f_1	1 1 1 1	1 1 1 1 1	1 1 1 1 1

S via BUS A to West port and East port to D via BUS B:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 0 0 0	1 0 0 1 0	S
f_1	0 0 1 0	1 1 1 1 1	1 1 1 1 1
f_2	0 1 1 0	1 1 1 1 1	1 0 1 1 0
f_1	1 1 1 1	D	1 1 1 1 1

S via BUS B to West port and East port to D via BUS A:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 0 0 0	1 0 0 0 0	1 1 1 1 1
f_1	0 0 1 0	1 1 1 1 1	S
f_2	0 1 1 0	D	1 0 1 0 0
f_1	1 1 1 1	1 1 1 1 1	1 1 1 1 1

S via BUS B to West port and East port to D via BUS B:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 0 0 0	1 0 0 0 0	1 1 1 1 1
f_1	0 0 1 0	1 1 1 1 1	S
f_2	0 1 1 0	1 1 1 1 1	1 0 1 1 0
f_1	1 1 1 1	D	1 1 1 1 1

Comments:

All processors move the data stored in register S (via the appropriate bus) to their WEST output port. The data is passed from every processor WEST output port to its west neighbor's EAST input port. In the second cycle, the enabled processors pass the data in their EAST input port to register D (via the appropriate bus).

NOTE:

Only the second cycle instruction is generated with a mask, the first is generated with ALL processors ENABLED. Thus the programmer must take care when executing I/O in one field of processors and some other operation in the other.

North To South Data Flow

Mnemonic Code: **GETN (S,D)**

Execution Cycles: **2**

Parameters:

X any source register of the **source processor** accessible by either BUS A or BUS B

Y any static register of the **destination processor** accessible by either BUS A or BUS B

Machine Code:

S via BUS A to South port and North port to D via BUS A:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 0 0 0	1 0 0 0 0	S
f_1	0 0 1 0	1 1 1 1 1	1 1 1 1 1
f_2	0 1 1 0	D	1 0 1 1 0
f_1	1 1 1 1	1 1 1 1 1	1 1 1 1 1

S via BUS A to South port and North port to D via BUS B:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 0 0 0	1 0 0 0 0	S
f_1	0 0 1 0	1 1 1 1 1	1 1 1 1 1
f_2	0 1 1 0	1 1 1 1 1	1 0 1 0 0
f_1	1 1 1 1	D	1 1 1 1 1

S via BUS B to South port and North port to D via BUS A:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 0 0 0	1 0 0 1 0	1 1 1 1 1
f_1	0 0 1 0	1 1 1 1 1	S
f_2	0 1 1 0	D	1 0 1 1 0
f_1	1 1 1 1	1 1 1 1 1	1 1 1 1 1

S via BUS B to South port and North port to D via BUS B:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 0 0 0	1 0 0 1 0	1 1 1 1 1
f_1	0 0 1 0	1 1 1 1 1	S
f_2	0 1 1 0	1 1 1 1 1	1 0 1 0 0
f_1	1 1 1 1	D	1 1 1 1 1

Comments:

All processors move the data stored in register S (via the appropriate bus) to their SOUTH output port. The data is passed from every processor's SOUTH output port to its south neighbor's NORTH input port. In the second cycle, the enabled processors pass the data in their NORTH input port to register D (via the appropriate bus).

NOTE:

Only the second cycle instruction is generated with a mask, the first is generated with ALL processors ENABLED. Thus the programmer must take care when executing I/O in one field of processors and some other operation in the other.

If the programmer executes a GETN(S,D) in conjunction with a memory access instruction, and the top and/or bottom processors corresponding to the GETN(S,D) instruction are not masked out, then some unreliable data will be read from and/or written to Memory in the corresponding columns.

South To North Data Flow

Mnemonic Code: **GETS (S,D)**

Execution Cycles: **2**

Parameters:

X any source register of the **source processor** accessible by either BUS A or BUS B

Y any static register of the **destination processor** accessible by either BUS A or BUS B

Machine Code:

S via BUS A to North port and South port to D via BUS A:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	0 1 0 1	1 0 1 1 0	S
f_1	0 0 1 1	1 1 1 1 1	1 1 1 1 1
f_2	1 0 1 1	D	1 0 0 0 0
f_1	1 1 1 1	1 1 1 1 1	1 1 1 1 1

S via BUS A to North port and South port to D via BUS B:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	0 1 0 1	1 0 1 1 0	S
f_1	0 0 1 1	1 1 1 1 1	1 1 1 1 1
f_2	1 0 1 1	1 1 1 1 1	1 0 0 1 0
f_1	1 1 1 1	D	1 1 1 1 1

S via BUS B to North port and South port to D via BUS A:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	0 1 0 1	1 0 1 0 0	1 1 1 1 1
f_1	0 0 1 1	1 1 1 1 1	S
f_2	1 0 1 1	D	1 0 0 0 0
f_1	1 1 1 1	1 1 1 1 1	1 1 1 1 1

S via BUS B to North port and South port to D via BUS B:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	0 1 0 1	1 0 1 0 0	1 1 1 1 1
f_1	0 0 1 1	1 1 1 1 1	S
f_2	1 0 1 1	1 1 1 1 1	1 0 0 1 0
f_1	1 1 1 1	D	1 1 1 1 1

Comments:

All processors move the data stored in register S (via the appropriate bus) to their NORTH output port. The data is passed from every processor's NORTH output port to its north neighbor's SOUTH input port. In the second cycle, the enabled processors pass the data in their SOUTH input port to register D (via the appropriate bus).

NOTE:

Only the second cycle instruction is generated with a mask, the first is generated with ALL processors ENABLED. Thus the programmer must take care when executing I/O in one field of processors and some other operation in the other.

West To East Data Flow

Mnemonic Code: **GETW (S,D)**

Execution Cycles: **2**

Parameters:

X any source register of the **source processor** accessible by either BUS A or BUS B

Y any static register of the **destination processor** accessible by either BUS A or BUS B

Machine Code:

S via BUS A to East port and West port to D via BUS A:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	0 1 0 1	1 0 1 0 0	S
f_1	0 0 1 1	1 1 1 1 1	1 1 1 1 1
f_2	1 0 1 1	D	1 0 0 1 0
f_1	1 1 1 1	1 1 1 1 1	1 1 1 1 1

S via BUS A to East port and West port to D via BUS B:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	0 1 0 1	1 0 1 0 0	S
f_1	0 0 1 1	1 1 1 1 1	1 1 1 1 1
f_2	1 0 1 1	1 1 1 1 1	1 0 0 0 0
f_1	1 1 1 1	D	1 1 1 1 1

S via BUS B to East port and West port to D via BUS A:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	0 1 0 1	1 0 1 1 0	1 1 1 1 1
f_1	0 0 1 1	1 1 1 1 1	S
f_2	1 0 1 1	D	1 0 0 1 0
f_1	1 1 1 1	1 1 1 1 1	1 1 1 1 1

S via BUS B to East port and West port to D via BUS B:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	0 1 0 1	1 0 1 1 0	1 1 1 1 1
f_1	0 0 1 1	1 1 1 1 1	S
f_2	1 0 1 1	1 1 1 1 1	1 0 0 0 0
f_1	1 1 1 1	D	1 1 1 1 1

Comments:

All processors move the data stored in register S (via the appropriate bus) to their EAST output port. The data is passed from every processor's EAST output port to its east neighbor's WEST input port. In the second cycle, the enabled processors pass the data in their WEST input port to register D (via the appropriate bus).

NOTE:

Only the second cycle instruction is generated with a mask, the first is generated with ALL processors ENABLED. Thus the programmer must take care when executing I/O in one field of processors and some other operation in the other.

4.1.4 Inter-Processor Communication: With Memory Access

The only link between the Processor Array and the "outside world," namely the Host, is through Data Memory. The Processor Array is connected via two ports to the Data Memory. The Read Port connects to the top row of the Array and its sole function is to transfer data **from** Memory **to** the Array. The Write Port connects to the bottom row of the Array and its sole function is to transfer data **from** the Array **to** Memory. Either or both these operations occur in conjunction with North to South data flow within the Array. See section 4.1.3. Since Memory access takes 3 cycles to execute, the I/O codes for North to South data flow has been stretched out to fill 3 cycles.

There are three possible instructions, Communication with a Read, Communication with a Write, and Communication with both Read and Write. The three differ only in which System Operation bits get set.

North To South Data Flow With Memory Access

Mnemonic Codes:

GETNRD (S,D)

GETNWT (S,D)

GETNRDWT (S,D)

Execution Cycles: **3**

Parameters:

X any source register of the **source processor** accessible by either BUS A or BUS B

Y any static register of the **destination processor** accessible by either BUS A or BUS B

Machine Code:

S via BUS A to South port and North port to D via BUS A:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 0 0 0	1 0 0 0 0	S
f_1	0 1 0 0	1 1 1 1 1	1 1 1 1 1
f_2	1 1 1 1	1 1 1 1 1	1 1 1 1 1
f_1	1 0 1 0	1 1 1 1 1	1 1 1 1 1
f_2	0 1 1 0	D	1 0 1 1 0
f_1	1 1 1 1	1 1 1 1 1	1 1 1 1 1

S via BUS A to South port and North port to D via BUS B:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 0 0 0	1 0 0 0 0	S
f_1	0 1 0 0	1 1 1 1 1	1 1 1 1 1
f_2	1 1 1 1	1 1 1 1 1	1 1 1 1 1
f_1	1 0 1 0	1 1 1 1 1	1 1 1 1 1
f_2	0 1 1 0	1 1 1 1 1	1 0 1 0 0
f_1	1 1 1 1	D	1 1 1 1 1

S via BUS B to South port and North port to D via BUS A:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 0 0 0	1 0 0 1 0	1 1 1 1 1
f_1	0 1 0 0	1 1 1 1 1	S
f_2	1 1 1 1	1 1 1 1 1	1 1 1 1 1
f_1	1 0 1 0	1 1 1 1 1	1 1 1 1 1
f_2	0 1 1 0	D	1 0 1 1 0
f_1	1 1 1 1	1 1 1 1 1	1 1 1 1 1

S via BUS B to South port and North port to D via BUS B:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 0 0 0	1 0 0 1 0	1 1 1 1 1
f_1	0 1 0 0	1 1 1 1 1	S
f_2	1 1 1 1	1 1 1 1 1	1 1 1 1 1
f_1	1 0 1 0	1 1 1 1 1	1 1 1 1 1
f_2	0 1 1 0	1 1 1 1 1	1 0 1 0 0
f_1	1 1 1 1	D	1 1 1 1 1

GETNRD Enables the **READ** bit in the System field of the above three instructions by setting bit 102 to 0.

GETNWT Enables the **WRITE** bit in the System field of the above three instructions by setting bit 100 to 0.

GETNRD Enables the **READ** and **WRITE** bits in the System field of the above three instructions by setting bits 100 and 102 to 0.

Comments:

All processors move the data stored in register S (via the appropriate bus) to their SOUTH output port. If the **READ** bit is set (= 0), data is transferred from the location in Memory pointed to by the Read Address Counter, to the NORTH input port of all the processors in the top row of the array. Data is passed from every processor's SOUTH output port to its south neighbor's NORTH input port. If the **WRITE** bit is set (= 0), data is transferred from the SOUTH output port of all the processors in the bottom row of

the array to Memory at the location pointed to by the Write Address Counter. In the third cycle, the enabled processors pass the data in their NORTH input port to register D (via the appropriate bus).

NOTE:

Only the third cycle instruction is generated with a mask, the first two are generated with ALL processors ENABLED. Thus the programmer must take care when executing I/O in one field of processors and some other operation in the other.

If the programmer executes a GETN(S,D) in conjunction with a memory access instruction, and the top and/or bottom processors corresponding to the GETN(S,D) instruction are not masked out, then some unreliable data will be read from and/or written to Memory in the corresponding columns.

4.1.5 Local Memory Storage/Retrieval

Moving Data Between Registers

Mnemonic Code:

MOV(X,W:Y,Z)

MOV(:Y,Z)

MOV(X,W:)

Execution Cycles: **1**

Operands:

X any source register accessible by BUS A

W any static register accessible by BUS A

Y any source register accessible by BUS B

Z any static register accessible by BUS B

Machine Code:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 1 1 1	W	X
f_1	1 1 1 1	Z	Y

Results: available in the next cycle following the cycle in which the unit is loaded

Register W contains the same data as register X.

Register Z contains the same data as register Y.

Comments:

Move the contents of X via BUS A to the static register W, and/or move the contents of Y via BUS B to the static register Z.

NOTE:

The NULL register _ may be used in both the source and destination positions. When used as a source, the contents of the precharged bus is transferred; this is equivalent to loading a value of -2^{-30}. As a destination, the data from the source is not latched

anywhere. This is useful when storing the result of the **Shifter** when only one result is desired.

MOV(:) and MOV(_,_:_,_) are both valid and are equivalent to a NOP.

4.1.6 Multiplication

Multiplication is implemented in two stages. In the first stage, a Multiplier is loaded and the fast Multiplier Clock is started. After 6 cycles, the partial product and the carry values are available. The full product is the sum of these values, so the second stage of multiplication loads the partial product and carry into an Adder and performs the addition. The product (and its ones complement) are available from the Adder outputs, thus any previous Sum (and its compliment) are overwritten.

Each stage must be specified in the program. This allows other operations (except another multiplication) to be executed in parallel with the Multiplier, even in the same processor as the Multiplier. This also reminds the programmer that the second stage takes place in a different functional unit, namely the Adder.

There are two Multipliers, **Multiplier 1** and **Multiplier 2**. The partial product and carry from **Multiplier 1** are loaded directly into **Adder 1**, and the partial product and carry from **Multiplier 2** are loaded directly into **Adder 2**. There are three types of multiplication available, single multiplication using **Multiplier 1**, single multiplication using **Multiplier 2**, and double multiplication of two sets of operands using both multipliers. For double multiplication, both multipliers are loaded before the Multiplier clock is started.

There are three different mnemonics for first stage of multiplication, but only two for the second stage. The second stage for single multiplication in **Multiplier 1** is the same as for double multiplication and affects both Adders. The reason for this is that there is no machine code to operate **Adder 1** independently from **Adder 2**.

There is only a single **Multiplier clock** for both Multipliers in ALL processors, Internal and External. For multiplication to run in both sets of processors, the multipliers must be started simultaneously, because once the clock has started, another multiplication may not commence until the clock is free. Once the clock

has started, the value stored in the dynamic output register of both multipliers are affected, therefore the result of the old multiplication must be used or saved in a static register before the next multiplication commences, else the value will be lost. The clock inputs to the processors are NOT masked, this means that once the clock starts, BOTH multipliers' outputs will be affected in ALL processors, internal and external, enabled and disabled.

Single Multiplication in Multiplier 1: First Stage

Mnemonic Code: **MULTF1 (X,Y)**

Execution Cycles: **6**

Operands:

X any source register accessible by BUS A

Y any source register accessible by BUS B

machine code:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 1 1 1	1 1 0 0 1	X
f_1	1 1 1 1	1 1 1 1 1	Y

Enables the MULTIPLY bit in the system field of the current instruction by setting bit 98 to 0.

Results: available in the 6 th cycle following the cycle in which the unit is loaded.

A **partial product** and a **carry** which are stored in dynamic registers accessible directly by **Adder 1**. The **partial product** and **carry** must be added in order to obtain the product.

Comments:

Moves the value stored in X via BUS A and the value stored in Y via BUS B to **Multiplier 1** and starts execution of multiplication (by starting the **Multiplier clock**). After 6 cycles, the partial product and carry values are stored in dynamic registers connected directly to **Adder 1**. These values must be then added (See Second stage multiplication) to obtain the product.

NOTE:

A MULTF1 is usually followed by a MULTSD operation within 5 cycles after the results of the multiplier are ready. Examples:

MULTF1 (X,Y)

```
NOP
NOP
NOP
NOP
NOP
MULTSD
```

or:

```
MULTF1 (X,Y)
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
MULTSD
```

The NOPs above may be replaced by any operations that do not involve multiplication.

There is only a single **Multiplier clock** for both Multipliers in all processors, Internal and External. Once the clock has been started, another multiplication may not commence, even a different Multiplier, even in a different set of processors, until the current execution has been completed.

The **Multiplier clock** changes the values stored in the output

registers of both **Multipliers** from its first pulse, therefore, the old values must be used (i.e. a MULTSD executed) before the **Multiplier clock** is restarted (as well as before the old values decay).

The **Multiplier clock** is NOT masked, therefore the output registers of Both multipliers in EVERY processor (internal and external, enabled and disabled) are changed.

Single Multiplication in Multiplier 2: First Stage

Mnemonic Code: **MULTF2 (X,Y)**

Execution Cycles: **6**

Operands:

X any source register accessible by BUS A

Y any source register accessible by BUS B

machine code:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 1 1 1	1 1 0 1 1	X
f_1	1 1 1 1	1 1 1 1 1	Y

Enables the **MULTIPLY** bit in the system field of the current instruction by setting bit 98 to 0.

Results: available in the 6 th cycle following the cycle in which the unit is loaded.

A **partial product** and a **carry** which are stored in dynamic registers accessible directly by **Adder 2**. The **partial product** and **carry** must be added in order to obtain the product.

Comments:

Moves the value stored in X via BUS A and the value stored in Y via BUS B to **Multiplier 2** and starts execution of multiplication (by starting the **Multiplier clock**). After 6 cycles, the partial product and carry values are stored in dynamic registers connected directly to **Adder 2**. These values must be then added (See Second stage multiplication) to obtain the product.

NOTE:

A MULTF2 is usually followed by a MULTS2 operation within 5 cycles after the results of the multiplier are ready. Examples:

MULTF2 (X,Y)

```
NOP
NOP
NOP
NOP
NOP
MULTS2
```

or:

```
MULTF2 (X,Y)
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
MULTS2
```

The NOPs above may be replaced by any operations that do not involve multiplication.

There is only a single **Multiplier clock** for both Multipliers in all processors, Internal and External. Once the clock has been started, another multiplication may not commence, even a different Multiplier, even in a different set of processors, until the current execution has been completed.

The **Multiplier clock** changes the values stored in the output

registers of both **Multipliers** from its first pulse, therefore, the old values must be used (i.e. a MULTS2 executed) before the **Multiplier clock** is restarted (as well as before the old values decay).

The **Multiplier clock** is NOT masked, therefore the output registers of Both multipliers in EVERY processor (internal and external, enabled and disabled) are changed.

Double Multiplication: First Stage

Mnemonic Code: **MULTFD (X,Y:W,Z)**

Execution Cycles: **7**

Operands:

X any source register accessible by BUS A. Input to **Multiplier 1.**

Y any source register accessible by BUS B. Input to **Multiplier 1.**

W any source register accessible by BUS A. Input to **Multiplier 2.**

Z any source register accessible by BUS B. Input to **Multiplier 2.**

machine code:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 1 1 1	1 1 0 0 1	X
f_1	1 1 1 1	1 1 1 1 1	Y
f_2	1 1 1 1	1 1 0 1 1	W
f_1	1 1 1 1	1 1 1 1 1	Z

Enables the MULTIPLY bit in the system field of the second instruction generated by setting bit 98 to 0.

Results: available in the 7 th cycle following the cycle in which the first unit is loaded.

The **partial products** and the **carries** which are stored in dynamic registers accessible directly by **Adder 1** and **Adder 2**. The **partial product** and **carry** must be added in order to obtain the product.

Comments:

Moves the value stored in X via BUS A and the value stored in Y via BUS B to **Multiplier 1** in one cycle, then moves the value stored in W via BUS A and the value stored in Z via BUS B to **Multiplier 2** and starts execution of multiplication (by starting the **Multiplier clock**). After 7 cycles from loading **Multiplier 1**, the partial product and carry values are stored in dynamic registers

connected directly to **Adder 1** and **Adder 2** from **Multiplier 1** and **Multiplier 2** respectively. These values must be then added (See Second stage multiplication) to obtain the product.

NOTE:

A MULTFD is usually followed by a MULTSD operation within 5 cycles after the results of the multiplier are ready. Examples:

```
MULTFD (X,Y)
NOP
NOP
NOP
NOP
NOP
MULTSD
```

or:

```
MULTFD (X,Y)
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
MULTSD
```

The NOPs above may be replaced by any operations that do not

involve multiplication.

There is only a single **Multiplier clock** for both Multipliers in all processors, Internal and External, therefore the clock is started after both Multipliers have been loaded. Once the clock has been started, another multiplication may not commence, even in a different set of processors, until the current execution has been completed.

The **Multiplier clock** changes the values stored in the output registers of both **Multipliers** from its first pulse, therefore, the old values must be used (i.e. a MULTSD executed) before the **Multiplier clock** is restarted (as well as before the old values decay).

The **Multiplier clock** is NOT masked, therefore the output registers of Both multipliers in EVERY processor (internal and external, enabled and disabled) are changed.

Single Multiplication in Multiplier 2: Second Stage

Mnemonic Code: **MULTS2**

Execution Cycles: **1**

Operands:

None

machine code:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 1 1 1	1 1 1 0 0	1 1 1 1 1
f_1	1 1 1 1	1 1 1 1 1	1 1 1 1 1

Results: available in the next cycle following the cycle in which t he unit is loaded

PROD2B Accessible from BUS B. The sum of the values stored at registers X and Y. It is identical to the register named SUM2B, see section 4.1.1 on Addition.

CPROD2A Accessible from BUS A. The ones complement of the sum of the values stored at registers X and Y. It is identical to the register named CSUM2A, see section 4.1.1 on Addition.

Comments:

Move the partial product and carry, stored in the dynamic output registers of **Multiplier 2**, to **Adder 2** and perform the addition operation. The sum from **Adder 2** is stored in the dynamic register PROD2B and its one's complement in the dynamic register CPROD2A.

Example:

```
MULTF2 (X,Y)
NOP
NOP
NOP
```

```
NOP
NOP
MULTS2
MOV (:PROD2B,B1)
```

The NOPs above may be replaced by any operations that do not involve **Multiplier 1** or **Multiplier 2**. Once **Adder 2** has been loaded (by a MULTS2 instruction), **Multiplier 2** is available.

NOTE:

Multiplier 2 must have finished execution within 5 cycles prior to calling MULTS2 in order for the product to be correct.

Single Multiplication in Multiplier 1: Second Stage

Double Multiplication: Second Stage

Mnemonic Code: **MULTSD**

Execution Cycles: **1**

Operands:

None

machine code:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 1 1 1	1 1 0 1 0	1 1 1 1 1
f_1	1 1 1 1	1 1 1 1 1	1 1 1 1 1

Results: available in the next cycle following the cycle in which t he unit is loaded

PROD1A Accessible from BUS A. The sum of the values stored at registers X and Y. It is identical to the register named SUM2B, see section 4.1.1 on Addition.

PROD2B Accessible from BUS B. The sum of the values stored at registers X and Y. It is identical to the register named SUM2B, see section 4.1.1 on Addition.

CPROD1B Accessible from BUS B. The ones complement of the sum of the values stored at registers X and Y. It is identical to the register named CSUM2A, see section 4.1.1 on Addition.

CPROD2A Accessible from BUS A. The ones complement of the sum of the values stored at registers X and Y. It is identical to the register named CSUM2A, see section 4.1.1 on Addition.

Comments:

Move the partial product and carry, stored in the dynamic output registers of **Multiplier 1** and **Multiplier 2**, to **Adder 1** and **Adder 2** respectively and perform the addition operation simul-

taneously in a single cycle. The sum from **Adder 1** is stored in the dynamic register PROD1A and its one's complement in the dynamic register CPROD1B. The sum from **Adder 2** is stored in the dynamic register PROD2B and its one's complement in the dynamic register CPROD2A.

Examples:

```
MULTF1 (X,Y)
NOP
NOP
NOP
NOP
NOP
MULTSD
MOV (PROD1A,A1:)

MULTFD (X,Y:W,Z)
NOP
NOP
NOP
NOP
NOP
MULTSD
MOV (PROD1A,A1:PROD2B,B1)
```

The NOPs above may be replaced by any operations that do not involve **Multiplier 1** or **Multiplier 2**. Once the Adders have been loaded (by a MULTSD instruction), the Multipliers are available.

NOTE:

Multiplier 1 or both **Multiplier 1** and **Multiplier 2** must have finished execution within 5 cycles prior to calling MULTSD in order for the product(s) to be correct.

Although there is a first stage single multiplication using **Multiplier 1** there is no single addition instruction for **Adder 1**, so the second stage must be executed in parallel with **Adder 2**. The side effect is that the output registers of **Adder 2** are overwritten.

4.1.7 No-Operation

As in other assembly codes, there is an operation which does nothing. It is equivalent to "moving nothing to nowhere." It is useful to execute only internal processors or only external processors. It is also useful to use up cycles while waiting for an operation with a multi-cycle execution time to finish executing.

No-Operation

Mnemonic Code: **NOP**

Execution Cycles: **1**

Operands:

None

Machine Code:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 1 1 1	1 1 1 1 1	1 1 1 1 1
f_1	1 1 1 1	1 1 1 1 1	1 1 1 1 1

Results:

None

Comments:

No new operation initiated in the current cycle.

NOTE:

4.1.8 Shifting

Shifting

Mnemonic Code: **SHIFT (X,Y)**

Execution Cycles: 1

Operands:

X any source register accessible by BUS A.

Y any source register accessible by BUS B.

Machine Code:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,...,D_1)	source (S_5,S_4,...,S_1)
f_2	1 1 1 1	1 1 1 1 1	X
f_1	1 1 1 1	1 1 1 0 1	Y

Results: available in the next cycle following the cycle in which the unit is loaded

SHIFTA The shifted value of X, normalized to be a value greater than or equal to 1 and less than 2. Dynamic register accessible from BUS A.

SHIFTB The shifted value of Y. Dynamic register accessible from BUS B.

Both values stored in SHIFTA and SHIFTB are also stored in dynamic registers accessible directly by the **Divider**.

Comments:

Moves the data stored in X via BUS A and the data stored in Y via BUS B to the **Shifter** and executes an arithmetic *left* shift, by shifting both values until the X input falls in the range greater than or equal to 1 and less than 2. To execute n shifts left on the Y input, the X input should be greater than or equal to 2^{-n} and less than $2^{-(n-1)}$. The shifted X value is stored in dynamic register SHIFTA as well as in a dynamic register connected to the **Divider**, and the shifted Y value is stored in dynamic register SHIFTB as

well as in a dynamic register connected to the **Divider.**

NOTE:

The outputs SHIFTA and SHIFTB are not independent, a single access code accesses both values simultaneously. This is a hardware or firmware restriction. A single code accessing these registers ties up both busses. The limiting result of this is that when moving the value of one of these outputs to a static register or as input to another functional unit, there may not be any other data movement. For example: (let $n, m = 1, 2, \ldots, 7$).

The following are examples of **valid** operations:

MOV(SHIFTA,An:SHIFTB,Bn)

MOV(:SHIFTB,Bn)

MOV(SHIFTA,ABn:)

ADD2(SHIFTA,SHIFTB)

DIV(SHIFTA,SHIFTB) *identical operation as DIVS.*

The following are examples of **invalid** operations:

MOV(SHIFTA,An:Bn,ABn)

MOV(An,Am:SHIFTB,ABn)

MULTS1(SHIFTA,SUMB2)

4.1.9 Sorting

Sorting or Comparing

Mnemonic Code: **SORT (X,Y)**

Execution Cycles: **1**

Operands:

X any source register accessible by BUS A

Y any source register accessible by BUS B

Machine Code:

phase	I/O code (LS, RS, A/B, I/O)	destination (D_5,D_4,. . . ,D_1)	source (S_5,S_4,. . . ,S_1)
f_2	1 1 1 1	1 1 1 1 1	X
f_1	1 1 1 1	1 1 0 0 0	Y

Results: available in the next cycle following the cycle in which the unit is loaded

HIGHA The higher value of X and Y. Accessible from BUS A.

LOWB The lower value of X and Y. Accessible from BUS B.

Comments:

Moves the data stored in X via BUS A and Y via BUS B to the **Sorter** which compares the two values. The greater (or higher) value is stored in the dynamic register HIGHA, and the smaller (or lower) value is stored in the dynamic register LOWB.

NOTE:

4.1.10 Subtraction

Subtraction can be performed by using one of the two following sequences of Additions. The examples are using Double Addition, but Single Addition may also be used for Y − X, or for the second part of X − Y. For X via BUS A and Y via BUS B:

Y − X :

```
ADDD(X,_)
ADDD(CSUM2A,Y)
```

X − Y :

```
ADDD(_,Y)
ADDD(X,CSUM1B)
```

4.2 Registers

There are two types of registers, **static** and **dynamic**. Local memory of each processor consists of 24 **static** registers. Eight can be accessed solely via BUS A, eight solely via BUS B, and eight via either bus. They are called **static** because once a value is stored, it remains valid until it is over-written. The register names and their machine codes are listed in the table of Figure 4.2.1. **Static** registers may be used as either the **source** or the **destination** register (where S_i stands for the i^{th} bit of the source code and D_i stands for the i^{th} bit of the destination code). The **phase** information tells which phase field to put the code. Registers placed in phase f_2 are transferred over BUS A and those placed in phase f_1 are transferred over BUS B.

The **NULL register** can be used inn place of any **static** register. When used as a **source**, the contents of the pre-charged bus is transferred to the destination register or functional unit. This is equivalent to loading a -2^{-30}. As a destination, the data placed on the bus is not latched anywhere, thus it is functionally equivalent to a NOP operation on that bus. This is useful when storing the result of a SHIFT when only one output is wanted, but the programmer wants an explicit reminder that the other bus is not free (See Section 4.1.8 on Shifting).

Table of Static Registers

Register	S_5/D_5	S_4/D_4	S_3/D_3	S_2/D_2	S_1/D_1	Phase
Bus A						
A0	0	1	0	0	0	f_2
A1	0	1	0	0	1	f_2
A2	0	1	0	1	0	f_2
A3	0	1	0	1	1	f_2
A4	0	1	1	0	0	f_2
A5	0	1	1	0	1	f_2
A6	0	1	1	1	0	f_2
A7	0	1	1	1	1	f_2
Bus B						
B0	0	1	0	0	0	f_1
B1	0	1	0	0	1	f_1
B2	0	1	0	1	0	f_1
B3	0	1	0	1	1	f_1
B4	0	1	1	0	0	f_1
B5	0	1	1	0	1	f_1
B6	0	1	1	1	0	f_1
B7	0	1	1	1	1	f_1
Either Bus A or Bus B						
AB0	0	0	0	0	0	f_1/f_2
AB1	0	0	0	0	1	f_1/f_2
AB2	0	0	0	1	0	f_1/f_2
AB3	0	0	0	1	1	f_1/f_2
AB4	0	0	1	0	0	f_1/f_2
AB5	0	0	1	0	1	f_1/f_2
AB6	0	0	1	1	0	f_1/f_2
AB7	0	0	1	1	1	f_1/f_2
_ *	1	1	1	1	1	f_1/f_2

*** The Null register. As source, reads bus whose lines are pulled high. As destination, value is not latched to anything.**

FIGURE 4.2.1 Static Registers

The outputs of the functional units are **dynamic** registers. They are called **dynamic** because their values decay over time. The data in a **dynamic** output register is valid for up to 5 cycles after being loaded; therefore, the programmer should be careful to use or store the results of an arithmetic operation within this

time limit.

If the co-processor is stopped (i.e. the STOP instruction is executed), when it is restarted, the old values in the **dynamic** registers are lost. The programmer should be careful to store any necessary results before issuing the STOP instruction.

The names of the **dynamic** registers are listed in the table of Figure 4.2.2. Please note that the "product" register names (PROD2A, etc.) are aliases for the output registers of the **Adders** (named SUM2A, etc.). Also note the unusual case of the **Shifter** output registers; that they must be accessed simultaneously, (and may not be accessed individually). See Section 4.1.8 for more details.

Table of Dynamic Output Registers

Register	S_5/D_5	S_4/D_4	S_3/D_3	S_2/D_2	S_1/D_1	Phase	Bus
Adder1							
SUM1A	1	1	0	1	0	f_2	A
PROD1A	1	1	0	1	0	f_2	A
CSUM1B	1	1	0	1	0	f_1	B
CPROD1B	1	1	0	1	0	f_1	B
Adder2							
SUM2B	1	1	0	1	0	f_1	B
CSUM2B	1	1	0	1	0	f_1	B
PROD2A	1	1	0	1	0	f_2	A
CPROD2A	1	1	0	1	0	f_2	A
Divider							
QUOTA	1	1	1	1	0	f_2	A
Shifter*							
SHIFTA	1	1	1	0	1	f_1	A
SHIFTB	1	1	1	0	1	f_1	B
Sorter							
HIGHA	1	1	0	0	0	f_2	A
LOWB	1	1	0	0	1	f_1	B

*** Shifter outputs are accessed only as a pair**

FIGURE 4.2.2 Dynamic Registers

4.3 Masks

Masks can be used after any Regular Instruction to enable only a subset of the PEs during execution of the given instruction. A single mask applies to both external and internal processors, which evokes certain restrictions, see the next chapter, on Limitations. A restriction to note is that when a Communication operation is masked, the first line of machine code generated does not contain that mask; see Section 4.1.1 and the chapter on Limitations. Masks may be predefined at the beginning of the file, and symbolic names used in the program.

There are two types of masking *row/column* masking and *diagonal* masking. Masked processors are **enabled**. The absence of the mask after a Regular Instruction means **all** processors are **enabled**. The presence of a mask means the listed processors are **enabled**; if the mask is empty, or incomplete (see below), then **no** processors are **enabled**.

Row/Column masking consists of a list of rows and a list of columns. The processors in the array which lie on the intersection of the listed rows and columns are **enabled**. The rows of the processor array are numbered from top to bottom, 1 through 16 (The top row, row 1, is connected to the Read port of Memory, and the bottom row, row 16 is connected to the Write port of Memory). See Figure 4.3.1. The columns are numbered from left to right, 1 through 16 (col 1 consists of the External processors).

Diagonal masking consists of a list of diagonals. The processors which lie on the listed diagonals are **enabled**. The diagonals lie slanting left i.e. from northwest to southeast. They are numbered starting with the single processor diagonal on the top left corner of the array and ending with the single processor diagonal on the bottom right corner of the array, 1 through 31. Diagonal 16 is the main diagonal of the array containing processors (i, i) for $i = 1, 2, \ldots, 16$. See Figure 4.3.1.

Masks can be either row/column OR diagonal, not both. Bit 43, SEL D/RC

determines the type of mask.

Masking

Mnemonic Code: **(Rlist:Clist:Dlist)**

Machine Code:

In a machine instruction, bits 0–15 contain the Row Mask *(bit 0 = Row 1,..., bit 15 = Row 16)* or the low order Diagonal Mask *(bit 0 = Diag 1,..., bit 15 = Diag 16)* , bits 16–31 contain the Column Mask *(bit 16 = Col 1,..., Bit 31 = Col 16)* or the high order Diagonal Mask *(bit 16 = Diag 17,..., bit 30 = Diag 31, and bit 31 is unused)*, and bit 43 is the SEL D/RC bit.

For a Row/Column mask: bit 43 = 0

For a Diagonal mask: bit 43 = 1

For the mask fields (bits 0–15 and 16–31) a 0 enables and a 1 disables. See Examples below.

Comments:

A mask consists of three fields within parentheses, the Row field, the Column field and the Diagonal field. For a given mask, either both the Row and Column fields may have values and the Diagonal field is empty, or only the Diagonal field may have values, and the other fields are empty. The Diagonal field has priority, meaning that if the Diagonal field has values and one or both of the other two also have values, then the Diagonal values become the mask and the other two are ignored. If the Diagonal field is empty and one of the Row and Column fields are empty (an *incomplete* mask), then the mask is still generated but the result is that **no** processors will be enabled since no processors lie on an intersection. If all three fields are empty, a mask is still generated, no processors are enabled.

The values within each field are a list of numbers and ranges of

numbers between 1 and 31. Element in the list are separated by comma. An element can be a number or a range. A range is of the form n-m, where n is the first value in the range and m is the last. The list may be in any order (increasing, decreasing, mixed).

Examples:

1 (1,16:1-16:)

A row/column mask enabling the top and bottom rows.

Translates to: bit 43 = 0

Column (msb,...,lsb)	Row (msb,...,lsb)
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0

2 (1-4,9-14:1,6-12:)

A row/column mask enabling 4 rectangular regions of processors. In the array below, o and $\dot{o}$ mean enabled and x means disabled. It is impossible with this masking technique to simultaneously enable regions that don't share all rows and columns specified; i.e. it is impossible to ONLY enable the processors below labeled $\dot{o}$ without also enabling those labeled o.

Translates to: bit 43 = 0

Column (msb,...,lsb)	Row (msb,...,lsb)
1 1 1 1 0 0 0 0 0 0 1 1 1 1 0	1 1 0 0 0 0 0 0 1 1 1 1 0 0 0 0

o	x	x	x	x	ȯ	ȯ	ȯ	ȯ	ȯ	ȯ	ȯ	x	x	x	x
o	x	x	x	x	ȯ	ȯ	ȯ	ȯ	ȯ	ȯ	ȯ	x	x	x	x
o	x	x	x	x	ȯ	ȯ	ȯ	ȯ	ȯ	ȯ	ȯ	x	x	x	x
o	x	x	x	x	ȯ	ȯ	ȯ	ȯ	ȯ	ȯ	ȯ	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
ȯ	x	x	x	x	o	o	o	o	o	o	o	x	x	x	x
ȯ	x	x	x	x	o	o	o	o	o	o	o	x	x	x	x
ȯ	x	x	x	x	o	o	o	o	o	o	o	x	x	x	x
ȯ	x	x	x	x	o	o	o	o	o	o	o	x	x	x	x
ȯ	x	x	x	x	o	o	o	o	o	o	o	x	x	x	x
ȯ	x	x	x	x	o	o	o	o	o	o	o	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

3 (::14-18)

A diagonal mask enabling the 5 diagonals around the main diagonal. In the array below, *o* means enabled and *x* means disabled.

o	o	o	x	x	x	x	x	x	x	x	x	x	x	x	x
o	o	o	o	x	x	x	x	x	x	x	x	x	x	x	x
o	o	o	o	o	x	x	x	x	x	x	x	x	x	x	x
x	o	o	o	o	o	x	x	x	x	x	x	x	x	x	x
x	x	o	o	o	o	o	x	x	x	x	x	x	x	x	x
x	x	x	o	o	o	o	o	x	x	x	x	x	x	x	x
x	x	x	x	o	o	o	o	o	x	x	x	x	x	x	x
x	x	x	x	x	o	o	o	o	o	x	x	x	x	x	x
x	x	x	x	x	x	o	o	o	o	o	x	x	x	x	x
x	x	x	x	x	x	x	o	o	o	o	o	x	x	x	x
x	x	x	x	x	x	x	x	o	o	o	o	o	x	x	x
x	x	x	x	x	x	x	x	x	o	o	o	o	o	x	x
x	x	x	x	x	x	x	x	x	x	o	o	o	o	o	x
x	x	x	x	x	x	x	x	x	x	x	o	o	o	o	o
x	x	x	x	x	x	x	x	x	x	x	x	o	o	o	o
x	x	x	x	x	x	x	x	x	x	x	x	x	o	o	o

Translates to: bit 43 = 1

Column (msb,...,lsb) Row (msb,...,lsb)

1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1

4 (:1,5-8,13:)

A row/column mask enabling NO processors.

Translates to: bit 43 = 0

Column (msb,...,lsb) Row (msb,...,lsb)

1 1 1 0 1 1 1 1 0 0 0 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

NOTE:

A single mask applies for both External and Internal Processors in a given instruction.

For a Row/Column mask, the number of distinct enabled regions of processors is equal to the number of distinct regions enumerated in the Row Field TIMES the number of distinct regions enumerated in the Column Field. See Example 2 above—2 Row regions × 2 Column regions = 4 regions enabled.

Masks ONLY effect the Source–Destination parts of the instruction. I/O is not affected by the mask, neither is the system control field. This means that I/O is executed by every processor specified by the instruction (if I/O is in the External field, ALL External processors send and receive from and to the specified ports. If I/O is in the Internal Field, the ALL Internal processors send and receive from and to the specified ports. This is one reason why the ports were made transparent to the user. Similarly, the Multiply and Divide clocks are activate the Multipliers and Dividers in ALL processors (this time NO differentiation between Internal and External, since both use the same clocks). Since the start of the clock changes the value of the output registers of the corresponding

functional units, ALL outputs of ALL the Multipliers (Dividers), Internal AND External, are affected when the Multiply (Divide) clock is started, regardless of the mask.

4.4 Special Instructions

Special Instructions are those which only effect the System Field of the machine instruction. The Assembler also uses information contained in them to generate the program FIFO, and the two memory address FIFOs. Since the System Field is not affected by the mask, these instructions are not masked. As currently implemented, Special Instructions, except for STOP, modify the System Field of the previous instruction thus they may not be preceded by a label.

Co-processor Execution Control

Mnemonic Code: **STOP**

Operands:

none

Sets bit 96 **STOP** equal to 0.

Comments:

Generates a NOP instruction with the **STOP** bit set to 0. During execution, the co-processor is placed into **HALT** mode in the current cycle. The co-processor can be put back into **RUN** mode only by the HOST.

NOTE:

Be sure to store any necessary results of arithmetic operations before generating the **STOP** instruction since the **dynamic** registers decay.

Mnemonic Code: **LOOP N LABEL**

Operands:

N The number of times to jump back to the instruction labeled LABEL.

LABEL The label name given to the first instruction of the loop (must be a **regular** instruction in the current implementation of the Assembler (May 1988)). See Section 3.2.3 for how to define a label. The program address associated with LABEL must be less than the current program address.

Sets bit 99 **LOAD PC** equal to 0.

Comments:

The **LOAD PC** bit of the previous instruction is set to 0 and the address associated with LABEL is entered N times into the **Program FIFO** followed by the program address of the next **regular** instruction. When executing, the loop gets executed once, then during the last cycle of the loop, the PC gets loaded from the **Program FIFO** with LABEL instead of being incremented. This occurs N times, after the N +1st traversal of the loop code, the PC gets loaded with the address of the next instruction and the program continues.

Example:

```
        SORT(A1,B1) ;
LABEL1: GETW(HIGHA,B2) ;
        SORT(HIGHA,B2) ;
        LOOP 14 LAB1 ;
LABEL2: MOV(HIGHA,AB0:);
```

The program segment above should store in each processors AB0 register the maximum of the values in registers A1 and B1 of all

processors in its own row. During Assembly time, the address LABEL1 gets stored on the program FIFO 14 times followed by the address LABEL2. The instructions GETW and the 2nd SORT will be executed a total of 15 times.

NOTE:

There is NO comma between N and LABEL.

The instructions within the loop are executed N +1 times.

In the current implementation of the Assembler (May 1988), there should be only **Regular** instructions within the loop, i.e. no READQs or WRITEQs nor other LOOPs inside a loop. Future versions of the Assembler should be able to handle single nested loops as well as READQs and WRITEQs within the loop.

If READQs or WRITEQs are place after a LOOP with no **Regular** instruction in between, then it is as if they were placed within the loop. The corresponding FIFOs will be popped N times. This is incorrect programming since only a single value would have been placed onto the FIFO. But, should the user want to execute the loop on N +1 different data queues (one for each iteration of the loop) then the user can precede the loop (in the instruction before the LABEL) by a READQ and/or WRITEQ, AND follow the LOOP instruction immediately by N READQs and/or N WRITEQs. This technique is only valid for the current version of the Assembler (May 1988) and may not work in future versions.

Mnemonic Code: **READQ Qname**

WRITEQ Qname

Operands:

Qname The name of the data queue which was defined at the beginning of the program file. See Section 3.2.1 for details.

READQ: Sets bit 103 **LD READ ADDR** equal to 0.

WRITEQ: Sets bit 101 **LD WRITE ADDR** equal to 0.

Comments:

The READQ instruction tells the assembler to set the **LD READ ADDR** bit to 0, and to enter the value of the previously defined Qname onto the Read Data Address FIFO.

The WRITEQ instruction tells the assembler to set the **LD WRITE ADDR** bit to 0, and to enter the value of the previously defined Qname onto the Write Data Address FIFO.

This sets up the corresponding memory port to point to the start of the next data queue the user wants to access. The Qname is defined at the beginning of the user program file. The same data queue may be used for either reading or writing or both (but not simultaneously although this is not checked in the assembler).

NOTE:

In the current implementation of the Assembler (May 1988), READQs and WRITEQs are not permitted inside a loop. Future versions of the Assembler should be able to handle READQs and WRITEQs within the loop.

If READQs or WRITEQs are place after a LOOP instruction with no **Regular** instruction in between, then it is as if they were

placed within the loop. The corresponding FIFOs will be popped N times. This is incorrect programming since only a single value would have been placed onto the FIFO. But, should the user want to execute the loop on N +1 different data queues (one for each iteration of the loop) then the user can precede the loop (in the instruction before the LABEL) by a READQ and/or WRITEQ, AND follow the LOOP instruction immediately by N READQs and/or N WRITEQs. This technique is only valid for the current version of the Assembler (May 1988) and may not work in future versions.

Chapter 5
Programming Limitations due to Hardware

There are many limitation the user must keep in mind while programming the Systolic/Cellular System. These limitations fall into three categories, the first category consists of those limitations due to system design and hardware. The user must accept these limitations since they are an integral part of the current version of the system. The second category is Assembly Language dependent. These limitations were imposed as a compromise between maximizing the flexibility of the user to program using as many levels of parallelism as possible while minimizing the complexity of writing the programs. Should a user want absolute maximum performance, the user should program directly in machine code, or try to optimize the assembled code. The third category consist of those limitations imposed by the Assembler in its current state (May 1988). These limitations should be eliminated in future versions of the Assembler. The latter two categories have been covered in previous chapters.

5.1 Architecture

The general architectural design of the Systolic/Cellular System allows for very limited applications. Some of these limiting features are the unidirectional memory access, and the division of processors into asymmetric sets, namely one column and 15 columns. Most conceptual systolic designs I have seen have data input from two sides of the processor array. By not having this feature, the Systolic/Cellular System will require much more time loading and positioning data.

5.2 Control

There are many features of the control mechanisms which place restrictions on the programmer. One of these is the fact that a disabled processor is not totally disabled. By disabling a processor, the user can only be assured that the 24 Static Local Memory Registers will remain unchanged. Side effects of the I/O execution and Clock inputs make the I/O port and the Multiplier and Divider output registers less than 100are valid, since another operation, even from a different processor can effect their values.

Another problem is due to the deterministic characteristic of machine execution. There is no data-dependent control at all, thus for example, there is no way to determine if a divisor is equal to 0 nor a way to stop execution of division in a given processor whose divisor is equal to 0, if the value is not known at coding time and the processor masked disabled.

5.3 Processor Design and Hardware

Communication between Processors is limited due to packaging constraints. The Processor chip has 100 pins thus it can support 2 32 bit ports or 4 16 bit ports. Due to this, I/O between nearest neighbors requires two cycles to execute. Furthermore, the I/O code for data flow in a single direction is not unique, the same code moves data from North to South as from East to West, and the same code moves data from South to North as from West to East. This feature makes it difficult for the programmer to use the I/O ports for storage since the values can change due to side effects of another instruction.

Within the processor there are two busses, but only 8 of the internal registers have access to both busses. This imposes local data management on the programmer.

Almost all the functional unit outputs can be accessed independently, ex-

cept for the **Shifter** output. This unit always outputs it A output register simultaneously with its B output register, tying up both busses. The result is that a single output from the **Shifter** may not be used as input to another functional unit directly. However, there is a positive side effect of this. If another register happens to be placed on the BUS at the same time as the **Shifter** outputs, the result is a bitwise AND of the values on BUS A.

The the functional units are designed for a fixed point data value whose magnitude is less than two. This greatly limits the applications by requiring the user to limit the values of the data.

5.4 Version II of the Systolic/Cellular System

If the prototype of the Systolic/Cellular System (version I) proves to be successful, a new and improved version will be developped. From the above discussion we can suggest some areas for improvement. These would be:

Functional units that can handle floating point arithmetic, instead of 2^{-30}'s complement.

Faster Memory access to keep up with I/O.

Independent I/O ports to simplify communications.

Masked fast clocks, so that the old results in disabled processors do not get erased when enabled processors execute multiplication or division.

A more sophisticated program control to enable some level of data dependent execution. Perhaps make the PE intelligent enough to be able to set clear its own enable bit for the next instruction. This might be done by having a comparison to some value or threshold and based on the result, set or clear an internal enable bit. If a control bit from the **Controller** says to use local control, then that enable bit gets ORed with the mask bits from the **Controller**. If the control bit says use only global control,

then the local enable bit is ignored. This type of construct would be useful to prevent division by zero and other illegal operations (such as those that might cause and overflow etc.).

Independent access for the SHIFTA and SHIFTB output registers so that functionally they become identical to any other output register.

Separate fast clocks for the External and Internal sets of processors. This will enable greater parallelism and less dependence between the two instruction sequences.

The designers at Hughes have already many plans for improvements to the Systolic/Cellular System. Many of the ideas stated above are included. The list of changes for the next version of the System include:

Independent I/O

New Boundary (Internal) processor chips, different from the Internal processor chips.

New floating point internal chips.

A new Boundary memory feeding into the array through column 1.

A new **Controller.**

etc.

Most of these intended improvements will answer many of the limitations that have been discussed throughout this document. If implemented, they will certainly enhance the machine by extending possible applications, simplify the Assembler (fewer checks to be made) and make the programmers job easier.

Chapter 6
Conclusion

This document described the programming issues relevant to the first version of the Cross-Assembler for the Hughes Systolic/Cellular System. The Assembler runs and assembles correct code. However, there are a few alterations and enhancements that need to be implemented.

One major change that must occur in the assembly code will be in the definition of a data queue. Currently, the user specifies the size and direction of the queue and the Assembler generates the addresses. After much discussion it was decided that the programmer should have more control over where the data queues are positioned, so the new declaration will include a user specified address along with some way to indicate queue direction, either be specification of a tail address, or an explicit direction specification.

An enhancement to the assembly code will be to implement a nested loop as well as to implement data queue specifications within loops (and in nested loops).

There are many fine points in programming this processor that the programmer must always keep in mind. One example is the decaying dynamic registers. Another is the extended execution times of multiplication and division. For both cases, the user must keep track of the number of instruction cycles that have gone by between initializing execution and accessing the results. The Assembler should be enhanced to keep track of how many instructions ago an operation was instigated when it sees its output register being accessed. If there is a discrepancy between the time the data is valid and when it is accessed, then a warning should

be generated. Similarly, where there is an exception to a general rule, such as with masking I/O commands, warnings should be generated if relevant (i.e. for an SIMD mode instruction, a warning about the first cycle not being masked is irrelevant; the same is true for the case where there is no mask specified).

In summary, there is still much work to be done to make this Assembler optimal (in terms of code generated) and user friendly (in terms of debugging tools). But with this tool, it is now possible to convert algorithms to run on the Systolic/Cellular System into readable, and debuggable. programs.

REFERENCES

[HUG] Hughes Research Laboratories *Faddeev Algorithm* SLIDES

[KER78] Kernighan, Brian W., Ritchie, Dennis M. *The C Programming Language*, Prentice-Hall Software Series, 1978

[NAS84] Nash, J. G., Hansen, S. *Modified Faddeev Algorithm for Matrix Manipulation*, Hughes Research Laboratories, Malibu, CA Aug. 1984

[NASa] Nash, J. G., Hansen, S. *Modified Faddeev Algorithm for Concurrent Execution of Linear Algebraic Operations*, Hughes Research Laboratories, Malibu, CA Aug. 1984

[NASb] Nash, J. G., *Linear Algebraic Processor*, Hughes Research Laboratories, Malibu, CA Aug. 1984

[NASc] Nash, J. G., et al *Systolic/Cellular Processor*, Hughes Research Laboratories, Malibu, CA Aug. 1984

[PRZ85a] Przytula, Wojtek *Inverse Jacobian Problem*, Hughes Research Laboratories, Malibu, CA Nov. 1985

[PRZ85b] Przytula, Wojtek *Assembler*, Hughes Research Laboratories, Malibu, CA Dec. 1985 DRAFT

[PRZ85c] Przytula, Wojtek *Parallelism Versus Time For Inverse Jacobian*, Hughes Research Laboratories, Malibu, CA Dec. 1985

[PRZ86a] Przytula, Wojtek *Assembly Code*, Hughes Research Laboratories, Malibu, CA Sept. 1986 UPDATE

[PRZ86b] Przytula, Wojtek *Controller Notes* , Hughes Research Laboratories, Malibu, CA Aug. 1986

[PRZ87] Przytula, Wojtek *Assembler*, Hughes Research Laboratories, Malibu, CA Dec. 1987 DRAFT

[PRZ88] Przytula, Wojtek *Systolic/Cellular System of Hughes Research Laboratories*, Hughes Research Laboratories, Malibu, CA Dec 1987, Mar 1, 1988, Mar 28, 1988 DRAFTS

[PRZ] Przytula, Wojtek *High Level Language Program for the Main Loop of the Inverse-Jacobian Problem in Semi-Sequential mode*, Hughes Research Laboratories, Malibu, CA

[SHI88] Shironoshita, Roberto *Systolic/Cellular Processor Simulation*, Senior Project, U. of Pennsylvania, Philadelphia, PA Apr 1988

[] Computer Science Division, Department of E.E. and C.S. *Unix Programmer's Manual: Supplementary Documents* University of California, Berkeley, CA Mar 1984