LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

The Monsoon Processing Element Architecture Reference

Computation Structures Group Memo 283

March 1, 1988

Gregory M. Papadopoulos

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-84-K-0099.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Contents

1 Scope 3

2 Basic Data Representations 3

2.1 Tokens 3
2.2 Immediates 4
2.3 Pointers 4
2.4 Floats 6
2.5 Integers 6
2.6 Booleans 6

3 Processing Pipeline 7

4 Macro Instructions 9

5 First Level Decode 11

5.1 Effective Address Generation 12
5.2 Type Map 13
5.3 Presence Map 13
5.4 Operand Fetch/Store 15

6 Second Level Decode 15

6.1 Function Unit Control 16
6.2 Floating point, arithmetic and logic unit 18
6.3 Pointer increment unit 21
6.4 Type Propagation Unit 22
6.5 Machine control unit 23
6.6 Next Address Control 24
6.7 Form Token Section 25
6.8 Form Token Control 25
6.9 Stack Management 27

7 Exceptions **28**

8 Statistics **29**

The Monsoon Processor Architecture Reference

1 Scope

The Monsoon processing element is an experimental realization of an **explicit token store** (ETS) dataflow processor that is especially suited for data scheduled numeric and symbolic computation. An instruction fetched in the first pipeline stage is decoded into simple control operations at each subsequent stage. The decode is table-lookup and conditional execution is possible on the basis of data type and matching state. The **microcontrols** for Monsoon are the contents of the decoding memories and state-transition tables.

This document describes the Monsoon processing element microarchitecture in sufficient detail for a programmer to design a new macro instruction set.

2 Basic Data Representations

The Monsoon processor pipeline operates on computation state descriptors called **tokens**. An input token is transformed into zero, one or two output tokens. A side-effect of transforming a token may also be a mutation of the local **frame store**, a large directly-addressed memory that also acts as the instruction and token queue store.

2.1 Tokens

A token is a 144 bit quantity comprising a TAG and a VALUE. The TAG and a VALUE are each 72 bits—8 bits of TYPE and 64 bits of IMMEDIATE, as follows:

Token			
TAG		VALUE	
TYPE 8	POINTER 64	TYPE 8	IMMEDIATE 64

2.2 Immediates

There is special hardware support for the following immediate representations:

Hardware Supported Immediates	
Representation	Operations
Pointer	Increment, Field Set
IEEE Double Precision Float	Add, Sub, Mult, Compare, Convert
64 Bit Integer	Add, Sub, Mult, Shift
64 Bit Boolean	16 ALUs, Bit Rotate

Because there is no predefined meaning to the TYPE fields, the hardware does not check for the representation validity for any operations[1]. For example, this allows instructions to manipulate a pointer using boolean operations. Input operand validity checking is handled orthogonally by the programmable type checking and propagation hardware.

2.3 Pointers

The pointer immediate has a structure which is unique to Monsoon. Pointers *cannot* be manipulated as integers, as is common on many von Neumann machines. A pointer has the following format:

POINTER				
PORT	MAP	IP	PE	FP
1	7	24	10	22

where,

PORT Indicates the l (= 0) or r (= 1) port of the instruction specified by PE:IP.

MAP Alias and interleave control. Increments to FP affect PE as specified by MAP.

IP Instruction pointer. The absolute address of a 32 bit instruction on processor number PE.

[1]Floating-point operations will produce exceptions codes for invalid operand formats and NaNs, however.

PE *Logical* processor number. The physical processor is a lookup on PE. For machines with less than 1024 physical processors, the LSBs of PE can be concatenated with the MSBs of FP, extending the physical address space of each PE

FP Frame pointer. The absolute address of a 72 bit location on processor number PE. PE:FP describes a global address, so a machine is limited to a maximum of 4000 megawords of physical memory.

The map field controls *updates* to the PE number when adding an offset to the FP or IP fields. The map implements the TTDA concept of **subdomains**. A subdomain is a collection of 2^n logical processors starting on modulo 2^n processor number boundary. Thus the lower n bits of PE define the relative processor number within a subdomain whose base processor PE number is obtained by setting these n bits to zero. Interleaving (hashing) of FP or IP is performed on single word boundaries. The internal structure of the map field is as follows:

MAP	
HASH	N
2	5

where,

HASH Selects the hash strategy: FP, aliased FP, base FP, or IP.

N Log of the number of processors in the subdomain.

HASH is encoded as follows:

HASH Field Encoding		
HASH	Strategy	Use
00	FP	Word interleaved data structures
01	aliased FP	Constant data structures
10	base FP	Loop constants
11	IP	Code hashing

Note that the map affects PE only when *incrementing (adding an offset to)* a pointer. The base FP mode *never* affects PE. An example of HASH = FP interleaving for N = 0,1,2 is shown in figure 1. If HASH = aliased FP then N least significant PE bits are ignored—any processor in the subdomain can service the token.

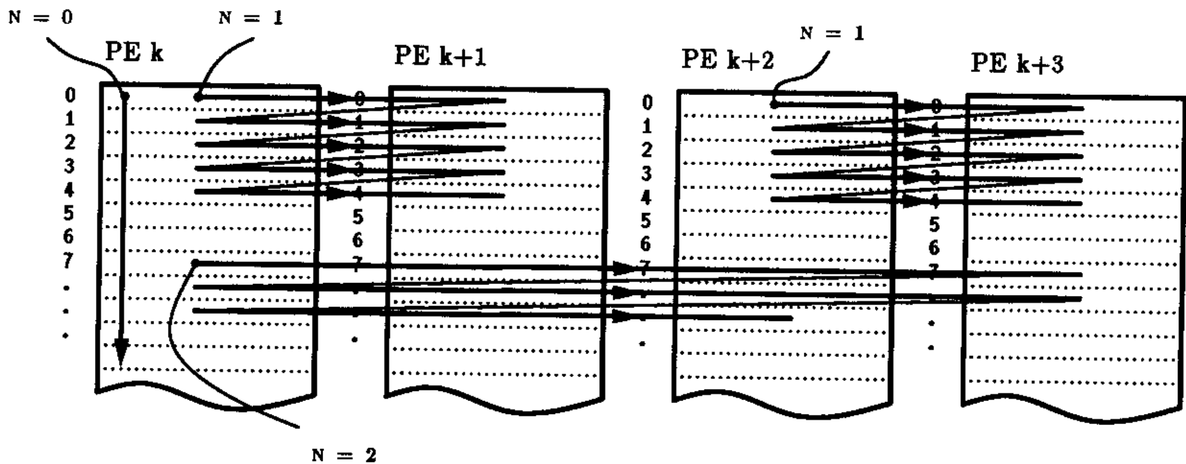

Figure 1: HASH = FP Interleaving for N = 0,1,2

2.4 Floats

Floating point numbers conform to IEEE Standard 754 for double precision. Extended precision floats are not directly supported by the hardware.

Full (Double) Precision Float		
SIGN 1	EXPONENT 11	MANTISSA 52

Rounding mode is a bootstrap configuration option and is not accessible from the microcontrols. Applications programmers should assume "round to nearest" as the default.

2.5 Integers

Integers are 64 bits, unsigned or signed 2's complement notation.

2.6 Booleans

Booleans are 64 bits. The complete set of bitwise "ALUs" (all of the sixteen functions on two bits) are provided as well bit rotates, reversals and shifts.

3 Processing Pipeline

The pipeline pushes tokens. An incoming token either is recirculated from the pipeline output, is popped from a token stack, or is sourced directly from the network. A decoded instruction tracks an incoming token through each stage, performing local operations, ultimately producing a new token(s) that either is directly recirculated, pushed onto a token stack, or sent over the network.

A token's trip through the pipe can involve at most one operand from the local memory. Typically the operand is a **slot** in an **activation frame**. Each word in local memory is 72 bits wide, large enough to hold a tag, a value, or two instructions. In addition, two **presence bits** are associated with each local memory word.

The token processing consists of two logical phases: (1) **operand generation** or **matching**, the calculation of the operand address, the manipulation of the associated presence bits, and the fetch or store of the operand, and (2) **token generation**, the calculation of the new token value part (the ALU/FPU), tag part (next address), and token construction (form token).

The Monsoon processor pipeline consists of six pipeline stages; instruction fetch, effective address calculation, presence bits, operand fetch/store, Next Address and FPU/ALU, and form token. A fetched instruction is subsequently decoded into "horizontal" control points for each stage. This decoding is table-driven, the opcode is used as an address into a memory. The contents of these decoding memories constitute the Monsoon microcontrols. In addition to these microcontrols there are two other tables that are involved in the implementation of an instruction set—an incoming type map and a presence bits state transition table.

The basic functionality of the stages are as follows. Please refer to figure 2.

1. **Instruction Fetch.** The IP field of the incoming token is treated as an absolute address into local memory. A single 32 bit instruction is fetched.

2. **Effective Address.** An effective operand address is computed by one of several modes: EA =FP+r,IP+r, or simply r. Simultaneous type analysis is performed on the VALUE TYPE field. This is accomplished by a table lookup (one of 32 type maps) on the TYPE field (eight bits) concatenated with the incoming PORT bit, resulting in a two bit type dispatch code, TC.

3. **Presence Bits.** The two presence bits for location EA are operated upon. This is accomplished by a table lookup (one of 64 presence state maps) on the concatenation of the current state (two bits), the incoming PORT bit, and the type dispatch code TC (two bits). The FSM issues a new state for location EA, an opcode for the operand fetch/store stage, a four-way microcontrol branch code, and a force-opcode-to-zero override.

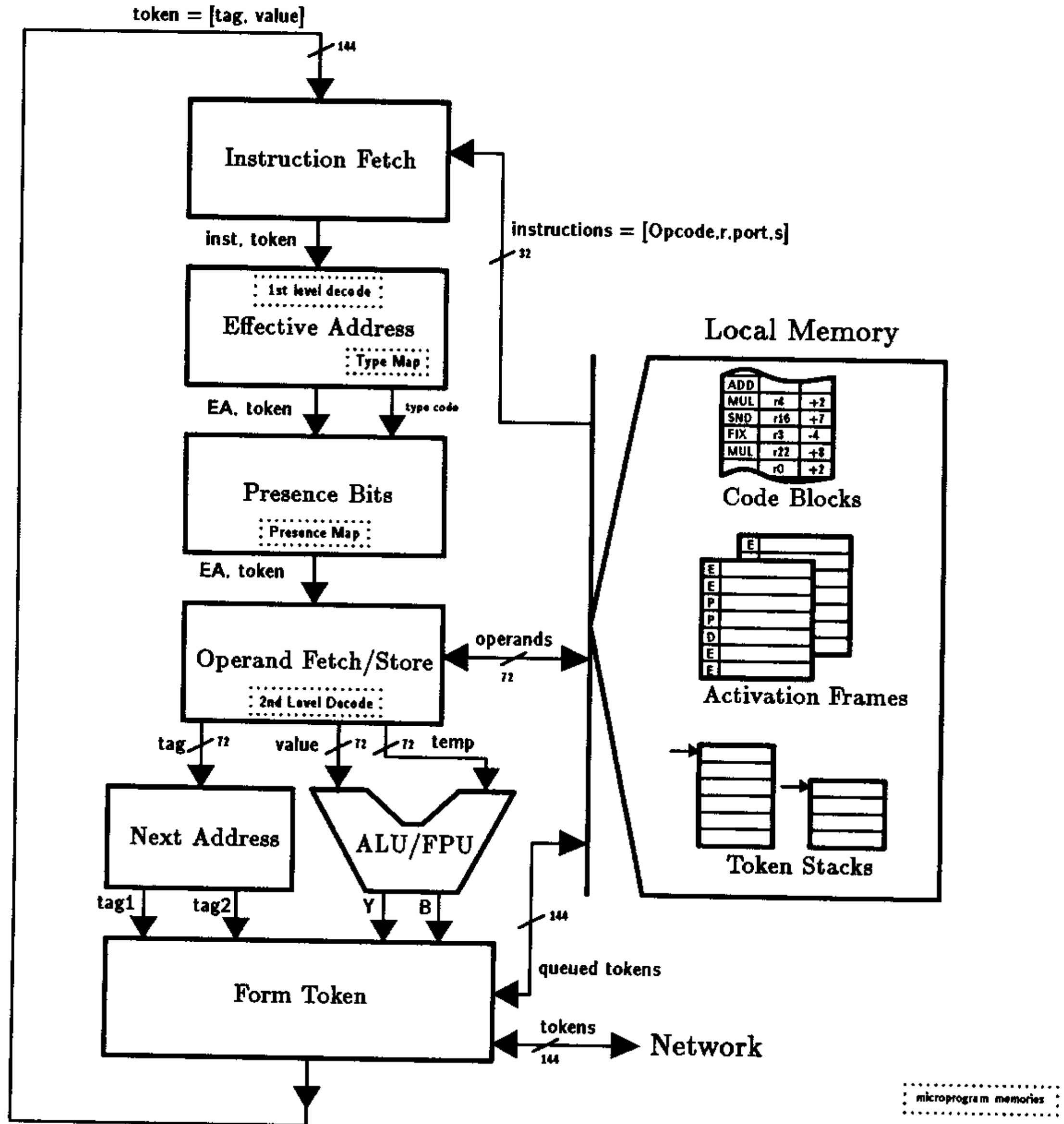

Figure 2: Processing Pipeline Overview

4. **Operand Fetch/Store.** As directed by the presence bits state table output, the memory location at EA is either read, written with the VALUE part of the token, or atomically exchanged with the VALUE part. During either a read or an exchange, the contents at address EA are passed to the ALU/FPU as *temp*.

5. **ALU/FPU.** The ALU/FPU operates upon the VALUE part of the token and *temp*, if applicable. In the case of a dyadic operation the hardware sorts out the left (L) and the right (R) operand from the incoming token and *temp* and presents them to the ALU/FPU as $l \rightarrow A$ and $r \rightarrow B$. Additionally, the microcontrols may call for a "cross-over", $l \rightarrow B$ and $r \rightarrow A$. *Two* results are produced; $Y = A$ OP B and B. In those cases where *temp* is not generated the hardware forces $A = B$. The ALU/FPU also performs type propagation and exception and condition code masking and generation.

6. **Next Address.** The next address unit operates in parallel with the ALU/FPU. It operates on the TAG part of the token whereas the ALU/FPU operates on the VALUE part. Specifically, the next address unit operates on the PORT and IP field (and implicitly the PE field depending upon MAP) to generate *two* new tags. The first tag is obtained by adding the s part of the instruction to the current IP and substituting the instruction-specified PORT. The second tag is generated by incrementing the current IP by either $0, +1, +2$ or $+3$ and by forcing the output port $= l$.

7. **Form Token.** The two values produced by the ALU/FPU and the two tags produced by the next address unit are conditionally assembled into zero, one or two tokens. The form token stage also manages the token stacks and arbitrates the network connection.

4 Macro Instructions

A macro instruction is 32 bits wide. The local memory is 72 bits wide, large enough to hold a tag, a value, or two instructions. Thus, the least significant bit of IP is not part of the address to local memory but is a selector for the upper or lower halfword, permitting instructions to be packed two to a word[2]. The instruction encoding is simple, consisting of just four fields:

Instruction			
OPCODE	r	PORT	s
10	10	1	11

[2]If the LSB of IP = 0 then the instruction is obtained from the least significant halfword of location IP SHR 1.

where,

OPCODE The instruction opcode. OPCODE is the address for the first level microcontrol store.

r An unsigned offset used by the effective address generator.

PORT Defines the l (= 0) or r (= 1) port for one of the destination tags.

s A 2's complement offset to IP for one of the destination tags.

The macro instruction decoding is controlled by the contents of four tables. Please refer to figure 3.

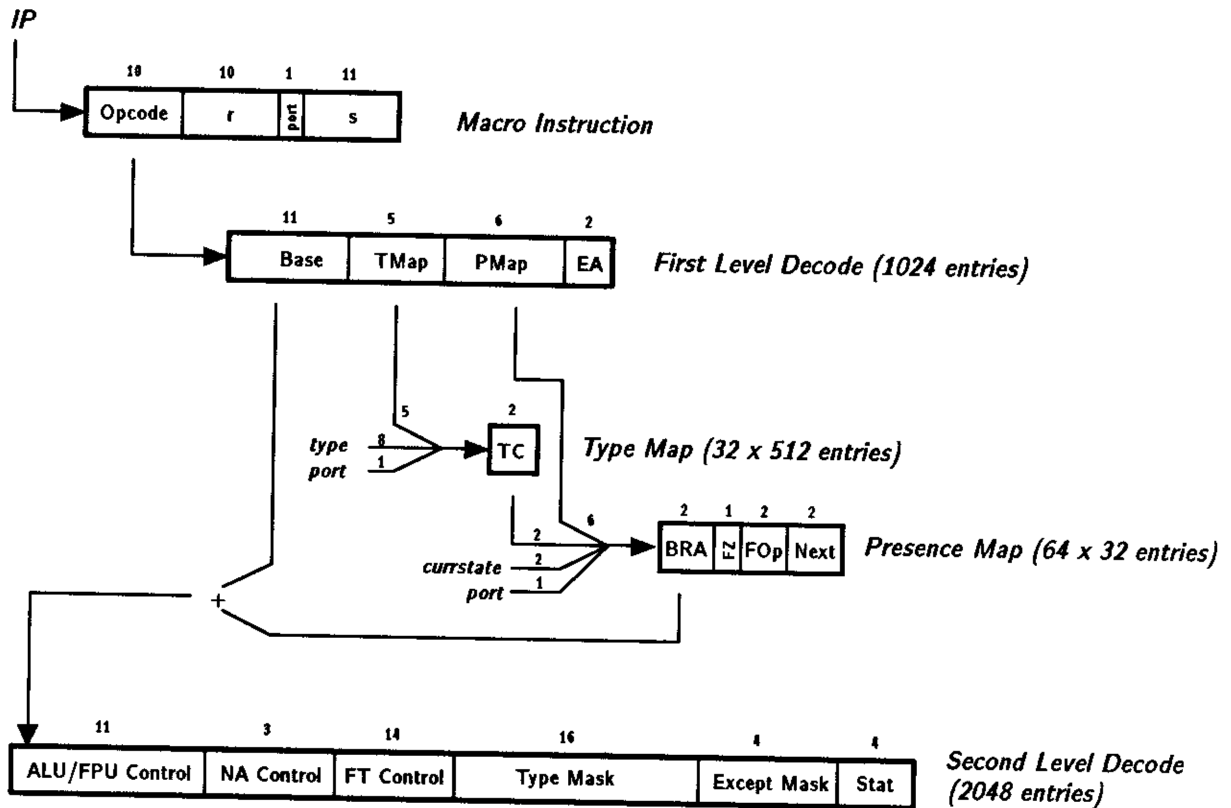

Figure 3: Instruction Decoding Tables and Maps

1. **First Level Decode.** Specifies the effective address generation mode, the type and presence maps, and the base address for the second level decode. The

macro instruction opcode is the address for the first level decode table, so there are $2^{10} = 1024$ entries of 24 bits each.

2. **Type Map.** A lookup on the TYPE and PORT of the VALUE part of the incoming token, yielding a two bit type dispatch code, TC. There are 32 type maps each with 512 two bit entries.

3. **Presence Map.** Maps the current presence state of address EA into a new state. The incoming PORT and two-bit type code TC are additional inputs. The presence map also specifies the frame store operation (read, write, or exchange) and one of eight branch points into the second level decode. There are 64 maps of 32 seven-bit entries.

4. **Second Level Decode.** Specifies the control points for the next address, FPU/ALU, and form token stages. The second level decode address is generated from the logical OR of the base address from the first level decode with the two branch bits from the presence map. If the presence map asserts force-zero, then the base is set to zero and the two branch bits specify the absolute second level decode entries 0, 1, 2 or 3. There are 2048 entries of 56 bits each.

5 First Level Decode

The instruction OPCODE is used as the address for the first level decode. The first level decode specifies the effective address generation mode, the type and presence maps, and the base address for the second level decode.

First Level Decode Entry			
BASE 11	TMAP 5	PMAP 6	EA 2

where,

BASE Specifies the base address for the second level decode.

TMAP Specifies one of 32 type maps.

PMAP Specifies one of 64 presence maps.

EA Specifies the effective address generation mode.

5.1 Effective Address Generation

The operand address is calculated by the effective address unit. The effective address generation always involves adding r to a masked version of FP or IP. Refer to figure 4

The are four encodings of the two EA bits as follows:

EA Field Encoding		
EA	Expression	Use
00	FP $+r$	Activation frame relative (operands)
10	IP $+r$	Instruction relative (literals)
11	r	Absolute (system)
01	mask(FP)$+r$	Active frame base (loop constants)

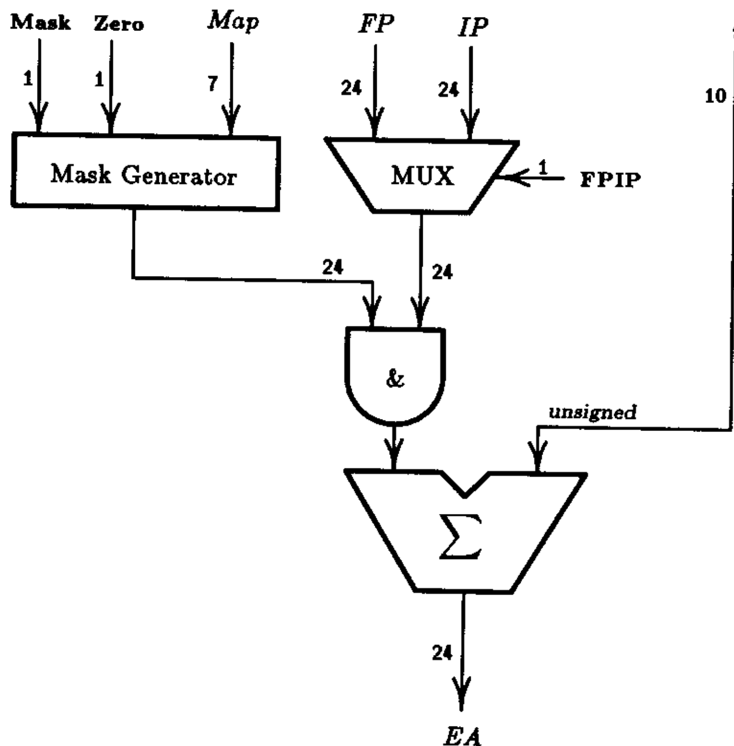

Figure 4: Effective Address Generator

The masking operation aligns FP (or IP) to a power-of-two boundary by forcing the lowest N bits (N is from the MAP field) to zero. For instance, if N = 15 then

the lower 15 bits of FP are set to zero, aligning it to a 32KW boundary. Because r is added *after* masking this mode allows any frame with an FP within the 32KW to "reach down" into the base frame. Remember that this masking only takes place when EA = 01 and N > 0 and HASH = 10. So the mask generator in figure 4 simply sets the lowest N output bits to 0s and remaining to 1s when the above condition is satisfied.

5.2 Type Map

The type map takes the VALUE TYPE and PORT and maps them into a two bit type code. The type code is used as input the presence map.

The TMAP field selects one of 32 type maps. A type map is conceptually a two dimensional array of 256 types × 2 ports. This means that each map has $2^9 = 512$ entries of two bits each. There are no preassigned meanings to the bits, they are purely programmer convention.

Type Map		
TYPE	PORT = l	PORT = r
00000000	$TC_{0,0}$	$TC_{0,1}$
00000001	$TC_{1,0}$	$TC_{1,1}$
00000010	$TC_{2,0}$	$TC_{2,1}$
...	...	...
11111110	$TC_{254,0}$	$TC_{254,1}$
11111111	$TC_{255,0}$	$TC_{255,1}$

5.3 Presence Map

The presence map takes the two presence bits at location EA, the incoming PORT, and the two type code bits TC and maps them into two new presence bits, a frame store operation, and a second level decode branch point. The PMAP field selects one of 64 presence maps. A presence map is a state transition table from the current presence to the new presence state. PORT and type code act as inputs. The frame store operation and branch code are outputs. Each presence map entry, PENT has the following fields:

PENT			
BRA	FZ	FOP	NEXT
2	1	2	2

where,

BRA Four-way branch control for the second level decode. The two BRA bits are ORed into the two least significant BASE bits (BASE is a first level decode field).

FZ Force-to-zero override. When asserted (1) the BASE field is forced to zero *before* the ORing of BRA. This yields four absolute second level decode entries in addition to the four base relative entries, or a possible eight-way branch.

FOP Specifies the operand fetch/store operation, defined below.

NEXT. The new value of the presence bits associated with location EA.

A presence map is consists of $2^{1+2+2} = 32$ entries as follows:

Presence Map					
Inputs		Current State			
PORT	TC	00	01	10	11
l	00	$\text{PENT}_{0,0}$	$\text{PENT}_{0,1}$	$\text{PENT}_{0,2}$	$\text{PENT}_{0,3}$
l	01	$\text{PENT}_{1,0}$	$\text{PENT}_{1,1}$	$\text{PENT}_{1,2}$	$\text{PENT}_{1,3}$
l	10	$\text{PENT}_{2,0}$	$\text{PENT}_{2,1}$	$\text{PENT}_{2,2}$	$\text{PENT}_{2,3}$
l	11	$\text{PENT}_{3,0}$	$\text{PENT}_{3,1}$	$\text{PENT}_{3,2}$	$\text{PENT}_{3,3}$
r	00	$\text{PENT}_{4,0}$	$\text{PENT}_{4,1}$	$\text{PENT}_{4,2}$	$\text{PENT}_{4,3}$
r	01	$\text{PENT}_{5,0}$	$\text{PENT}_{5,1}$	$\text{PENT}_{5,2}$	$\text{PENT}_{5,3}$
r	10	$\text{PENT}_{6,0}$	$\text{PENT}_{6,1}$	$\text{PENT}_{6,2}$	$\text{PENT}_{6,3}$
r	11	$\text{PENT}_{7,0}$	$\text{PENT}_{7,1}$	$\text{PENT}_{7,2}$	$\text{PENT}_{7,3}$

Out of the 64 possible maps defined by PMAP, the first sixteen have reserved meanings as indicated by the following table.

Special Presence Maps		
PMAP	Additional Functionality	
0–3	*NOP*	No frame store operation (FOP ignored)
4–7	*Bulk*	Presence bits of adjacent (modulo 32) words all updated
8–11	*Write-Through*	All operand writes update main memory
12–15	*Non-Cacheable*	Reads and writes avoid operand cache

The actual value of the presence bits have no hardware significance *except* 00. This state is a distinguished **empty state**, the associated word has an arbitrary (and irrelevant) value. This is used in the cache roll-out algorithm—words in the empty state are never written back to memory *even if they are dirty.*

5.4 Operand Fetch/Store

The operand fetch/store stage performs a read, write, or exchange on local memory location EA. In the case of a write or exchange the quantity written is simply the 72 bit VALUE field of the incoming token. In the case of a read or exchange the stage produces an additional 72 bit result called *temp*, which is the contents of $[EA]$. Otherwise *temp* = VALUE. The operation performed is determined by FOP as follows:

FOP Field Encoding		
FOP	operation	
00	*Read*	$temp \leftarrow [EA]$
01	*Write*	$[EA] \leftarrow$ VALUE
10	*Exchange*	$temp \leftarrow [EA]; [EA] \leftarrow$ VALUE
11	*Enqueue*	$temp \leftarrow [EA]; [EA] \leftarrow$ (VALUE.IP + 1)

Note that if PMAP $= 0, 1, 2, 3$ then FOP is preempted, *no* memory operation takes place and *temp* is set to the incoming token VALUE field.

6 Second Level Decode

The second level decode address is computed as (BASE OR BRA) where BASE is the base entry point from the first level decode and BRA is the two bit branch code from the presence map. If the presence map asserts FZ (1) then BASE is forced to zero and the second level decode address is simply BRA, absolute locations 0,1,2 or 3.

The second level decode specifies the ALU/FPU opcode, controls next address generation, and specifies the form token mode. In addition, the second level decode controls type propagation, condition and exception masking, and statistics gathering.

Second Level Decode Entry					
FUCTL	NACTL	FTCTL	TMASK	EMASK	STATS
11	3	12	16	10	4

where,

FUCTL Selects a function unit and specifies its control.

NACTL Controls the next address generation.

FTCTL Specifies the form token mode.

TMASK Specifies operand type checking and propagation.

EMASK Specifies the exception mask.

STATS Specifies an increment for one of 16 instruction mix counters.

6.1 Function Unit Control

The function units operate on the *temp* and VALUE fields produced by the operand fetch/store stage. First, *temp* and VALUE are resolved into a pair of input operands A and B in relation to the incoming PORT. The A and B operands are fed to four function units, only *one* of which is selected by the current microinstruction. The active function unit produces a result, Y, a delayed version of B and condition and exception codes. The Y and B outputs are sent to the form token unit. All of this is under control of FUCTL, which has the following structure:

FUCTL		
FLIP	UNIT	OP
1	2	8

where,

FLIP Specifies the $l,r \rightarrow A,B$ mapping.

UNIT Selects one of the four function units: FALU, PIU, MCU or TPU.

OP Function unit opcode. Interpreted by each function unit.

A schematic of the function unit interconnection is given in figure 5. The FLIP bit determines how *temp* and VALUE are to mapped to A and B. This is a function of the incoming PORT bit as follows:

Determination of A and B		
	PORT	
FLIP	l	r
0	$A \leftarrow$ VALUE $B \leftarrow$ *temp*	$A \leftarrow$ *temp* $B \leftarrow$ VALUE
1	$A \leftarrow$ *temp* $B \leftarrow$ VALUE	$A \leftarrow$ VALUE $B \leftarrow$ *temp*

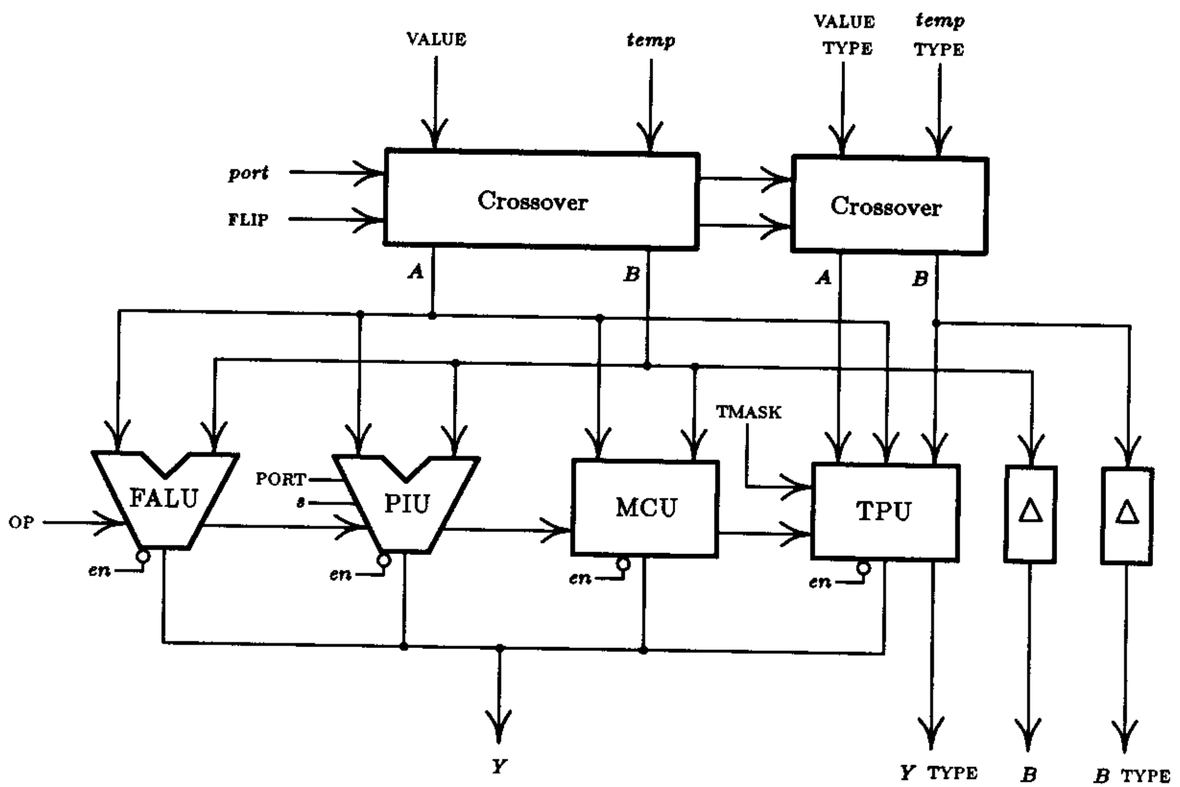

Figure 5: Function Unit Interconnections

So, regardless of how it happens, FLIP specifies the following relationship between l,r and A, B.

Encoding of FLIP	
FLIP	Mapping
0	$A \leftarrow l$ $B \leftarrow r$
1	$A \leftarrow r$ $B \leftarrow l$

The UNIT field selects one of the four parallel function units:

Encoding of UNIT		
UNIT	Function Unit	
00	*FALU*	Floating point, arithmetic and logic unit
01	*PIU*	Pointer increment unit
10	*TPU*	Type propagation unit
11	*MCU*	Machine control unit

6.2 Floating point, arithmetic and logic unit

The floating point, arithmetic and logic unit provides hardware support for floating point numbers, integers, and booleans. The OP field is encoded as follows:

Floating Point Ops UNIT = 00			
OP	Mnemonic	Operation	Types
	FDIV	$Y = A/B$	$float \times float \rightarrow float$
	FSQRT	$Y = \sqrt{A}$	$float \rightarrow float$
	FMUL	$Y = A * B$	$float \times float \rightarrow float$
	FMULAA	$Y = \|A\| * B$	$float \times float \rightarrow float$
	FMULAB	$Y = A * \|B\|$	$float \times float \rightarrow float$
	FMULA	$Y = \|A * B\|$	$float \times float \rightarrow float$
	FMIN	$Y = min(A, B)$	$float \times float \rightarrow float$
	FMAX	$Y = max(A, B)$	$float \times float \rightarrow float$
	FABS	$Y = \|A\|$	$float \rightarrow float$
	FNEG	$Y = -A$	$float \rightarrow float$
	FPASS	$Y = A$	$float \rightarrow float$
	FADD	$Y = A + B$	$float \times float \rightarrow float$
	FADDA	$Y = \|A\| + \|B\|$	$float \times float \rightarrow float$
	FSUB	$Y = A - B$	$float \times float \rightarrow float$
	FSUBR	$Y = B - A$	$float \times float \rightarrow float$
	FSUBA	$Y = \|A\| - \|B\|$	$float \times float \rightarrow float$
	FSUBRA	$Y = \|B\| - \|A\|$	$float \times float \rightarrow float$

Integer Ops UNIT = 00			
OP	Mnemonic	Operation	Types
	IMUL	$Y = A * B$	$integer \times integer \rightarrow integer$
	IMULU	$Y = A * B$	$unsigned \times unsigned \rightarrow unsigned$
	IMULUB	$Y = A * B$	$integer \times unsigned \rightarrow integer$
	IADD	$Y = A + B$	$integer \times integer \rightarrow integer$
	ISUB	$Y = A - B$	$integer \times integer \rightarrow integer$
	ISUBR	$Y = B - A$	$integer \times integer \rightarrow integer$
	IABS	$Y = \|A\|$	$integer \rightarrow integer$
	INEG	$Y = -A$	$integer \rightarrow integer$
	IMAX	$Y = max(A, B)$	$integer \times integer \rightarrow integer$
	IMIN	$Y = min(A, B)$	$integer \times integer \rightarrow integer$
	IMAXU	$Y = max(A, B)$	$unsigned \times unsigned \rightarrow unsigned$
	IMINU	$Y = min(A, B)$	$unsigned \times unsigned \rightarrow unsigned$
	IPASSU	$Y = A$	$unsigned \rightarrow unsigned$
	ISHIFT	$Y = A\, Shift\, B$	$integer \times integer \rightarrow integer$

Conversion Ops UNIT = 00			
OP	Mnemonic	Operation	Types
	FCI	$Y = fix(A)$	$float \rightarrow integer$
	FCU	$Y = fix(A)$	$float \rightarrow unsigned$
	ICF	$Y = float(A)$	$integer \rightarrow float$
	IUCF	$Y = float(A)$	$unsigned \rightarrow float$
	FCTI	$Y = trunc(A)$	$float \rightarrow integer$
	FCTU	$Y = trunc(A)$	$float \rightarrow unsigned$
	FCICF	$Y = float(fix(A))$	$float \rightarrow float$
	FCITCF	$Y = float(trunc(A))$	$float \rightarrow float$

Boolean Ops UNIT = 00			
OP	Mnemonic	Operation	Types
	LS	$Y = A\,Shift\,B$	$boolean \times integer \rightarrow boolean$
	NAND	$Y = \bar{A} + \bar{B}$	$boolean \times boolean \rightarrow boolean$
	ORNA	$Y = \bar{A} + B$	$boolean \times boolean \rightarrow boolean$
	ORNB	$Y = A + \bar{B}$	$boolean \times boolean \rightarrow boolean$
	OR	$Y = A + B$	$boolean \times boolean \rightarrow boolean$
	ANDNA	$Y = \bar{A}B$	$boolean \times boolean \rightarrow boolean$
	ANDNB	$Y = AB$	$boolean \times boolean \rightarrow boolean$
	AND	$Y = AB$	$boolean \times boolean \rightarrow boolean$
	XNOR	$Y = A \bar{\oplus} B$	$boolean \times boolean \rightarrow boolean$
	XOR	$Y = A \oplus B$	$boolean \times boolean \rightarrow boolean$
	SET	$Y = 111 \cdots 1$	$* \rightarrow boolean$
	NOTA	$Y = \bar{A}$	$boolean \rightarrow boolean$
	PASSB	$Y = B$	$boolean \rightarrow boolean$
	PASSA	$Y = A$	$boolean \rightarrow boolean$
	CLR	$Y = 000 \cdots 0$	$* \rightarrow boolean$
	NOTB	$Y = \bar{B}$	$boolean \rightarrow boolean$

The boolean comparison operations have integer or float inputs and produce a boolean output where true is defined as all 1's and false is all 0's.

Comparison Ops UNIT = 00			
OP	Mnemonic	Operation	Types
00011000	FEQ?	$Y = (A = B)$	*float* × *float* → *boolean*
00011001	FLT?	$Y = (A < B)$	*float* × *float* → *boolean*
00011010	FLEQ?	$Y = (A \leq B)$	*float* × *float* → *boolean*
00011100	FNEQ?	$Y = (A \neq B)$	*float* × *float* → *boolean*
00011101	FGEQ?	$Y = (A \geq B)$	*float* × *float* → *boolean*
00011110	FGT?	$Y = (A > B)$	*float* × *float* → *boolean*
10111000	IEQ?	$Y = (A = B)$	*integer* × *integer* → *boolean*
10111001	ILT?	$Y = (A < B)$	*integer* × *integer* → *boolean*
10111010	ILEQ?	$Y = (A \leq B)$	*integer* × *integer* → *boolean*
10111100	INEQ?	$Y = (A \neq B)$	*integer* × *integer* → *boolean*
10111101	IGEQ?	$Y = (A \geq B)$	*integer* × *integer* → *boolean*
10111110	IGT?	$Y = (A > B)$	*integer* × *integer* → *boolean*

6.3 Pointer increment unit

The pointer increment unit (PIU) computes updates to a pointer's IP, FP and PORT fields, and as a side-effect PE as directed by MAP. The PIU assumes that the A operand is of type POINTER. The B operand is assumed to be a signed integer when it is involved in the update. The updates to the IP, FP and PORT fields are independently controlled by OP.

OP when UNIT = 01			
UNDEFINED	PORTOP	IPOP	FPOP
2	2	2	2

where,

PORTOP Controls the generation of the PORT field.

IPOP Controls the generation of the IP field.

FPOP Controls the generation of the FP field.

Note that the resulting MAP field is identical to the A MAP field, and that PE may change as determined by MAP. PORTOP, IPOP and FPOP are encoded as follows:

PORTOP Field Encoding	
PORTOP	Resulting PORT
00	A PORT
01	instruction PORT
10	l
11	r

IPOP Field Encoding	
IPOP	Resulting IP
00	A IP + B
01	A IP + s
10	A IP
11	s

FPOP Field Encoding	
IPOP	Resulting FP
00	A FP + B
01	A FP + s
10	A FP
11	s

6.4 Type Propagation Unit

The type propagation unit controls the generation of the Y TYPE field. This involves two second level decode fields. TMASK is always active and normally used to control the generation of the Y TYPE. Alternatively, when UNIT = 10 the OP field can control the generation of the Y type.

The propagation of the Y TYPE is specified bit-by-bit for each of the eight type bits. Each resulting type can be directly specified as 0 or 1, or inherited from a bit in the same position from either the A or B input types. For each bit position $i = 0, \ldots, 7$ consider the two controls, m_i and v_i, encoded as follows:

Type Propagation Controls		
m_i	v_i	Result TYPE$_i$
0	0	0
0	1	1
1	0	A TYPE$_i$
1	1	B TYPE$_i$

The TMASK field is a set of eight m_i and v_i controls as follows:

TMASK	
$m_7, \ldots, m_0$ 8	$v_7, \ldots, v_0$ 8

TMASK is used to generate the Y TYPE *except* when otherwise specified by a UNIT = 10 OP as follows:

Type Operations UNIT = 10			
OP	Operation	Y TYPE	Y IMMEDIATE
0	SETTYPE	A	A TYPE
1	GETTYPE	TMASK	A TYPE

Note that the type specification for SETTYPE, A, is in the same format as TMASK so SETTYPE can be used to set or reset individual type bits.

6.5 Machine control unit

The machine control unit (MCU) permits program access of machine control settings. There are three classes of controls, (1) stack control, (2) exception control and (3) statistics.

MCU Stack Controls UNIT = 11		
OP	Operation	Description
1 000 0000	SETBASE0	*stack*0 base $\leftarrow A$
1 000 0001	SETBASE1	*stack*1 base $\leftarrow A$
1 000 0010	SETTOS0	*stack*0 TOS $\leftarrow A$
1 000 0011	SETTOS1	*stack*1 TOS $\leftarrow A$
1 000 0100	SETNOPOP0	NOPOP0 $\leftarrow A$
1 000 0101	SETNOPOP1	NOPOP1 $\leftarrow A$
1 000 0110	SETSWAP	STACKSWAP $\leftarrow A$
0 000 0000	GETBASE0	$Y \leftarrow$ *stack*0 base
0 000 0001	GETBASE1	$Y \leftarrow$ *stack*1 base
0 000 0010	GETTOS0	$Y \leftarrow$ *stack*0 TOS
0 000 0011	GETTOS1	$Y \leftarrow$ *stack*1 TOS
0 000 0100	GETNOPOP0	$Y \leftarrow$ NOPOP0
0 000 0101	GETNOPOP1	$Y \leftarrow$ NOPOP1
0 000 0110	GETSWAP	$Y \leftarrow$ STACKSWAP

MCU Exception Controls UNIT = 11		
OP	Operation	Description
0 001 0000	GETA	$Y \leftarrow A_{exception}$
0 001 0001	GETB	$Y \leftarrow B_{exception}$
0 001 0010	GETSTATUS	$Y \leftarrow$ status word
1 001 0011	CLEAR	Clear exception flag

MCU Statistics Controls UNIT = 11		
OP	Operation	Description
0 010 0000	ACTIVITY?	$Y \leftarrow$ *activity flag*; *activity flag* $\leftarrow 0$
0 010 0001	GETCOUNTER	$Y \leftarrow counter(A)$
1 010 0001	SETCOUNTER	$counter(A) \leftarrow B$

Note that the *activity flag* is automatically set by any instruction that doesn't explicitly clear it. Idle instructions should always clear this flag.

6.6 Next Address Control

The next address control field NACTL specifies increments to the current tag IP to generate two new destination tags, *tag*1 and *tag*2. NACTL has two subfields:

NACTL	
NA1	NA2
2	1

where,

NA1 Specifies the *tag*1 IP increment of 0,1,2, or 3. PORT is always set to *l*.

NA2 Specifies the *tag*2 IP increment of 0 or *s*. PORT is set to the instruction PORT or *r*.

Encoding of NA1		
NA1	*tag*1 IP	*tag*1 PORT
00	IP	*l*
01	IP + 1	*l*
10	IP + 2	*l*
11	(IP OR 1) + 3	*l*

Encoding of NA2		
NA2	*tag*2 IP	*tag*2 PORT
0	IP	r
1	IP + s	inst PORT

Note that NA1 = 0, NA2 = 0 produces tags that refer to the left and right ports of the *current* instruction.

6.7 Form Token Section

The form token section assembles zero, one, or two output tokens from the function unit outputs Y and B and the next address outputs *tag*1 and *tag*2. The unit also controls the token stacks and automatically forwards tokens to the network.

6.8 Form Token Control

The FTCTL field has the following structure:

FTCTL							
EN1	EN2	K1	K2	ORD	RECIRC	STACK	ACK
2	2	2	2	1	2	1	1

where,

- **EN1, EN2** Specifies the conditional output predicates for *token*1 and *token*2.
- **K1, K2** Specifies the assembly of *token*1 and *token*2.
- **ORD** Specifies the relative priority of *token*1 and *token*2.
- **RECIRC** Controls the recirculation of the higher priority token.
- **STACK** Specifies the stack for the lower priority token.
- **ACK** Controls acknowledgment for network packets.

Two logical tokens *token*1 and *token*2 are generated from EN1, K1 and EN2, K2 specifications. In the case of two tokens being generated ORD designates *token*1 or *token*2 as the highest priority token, *i.e.*, the token that will be recirculated. RECIRC controls the recirculation of this token. In the case that only one token is formed, or in the case that two are formed and the other is sent over the network, the token recirculation is controlled by RECIRC In the event that two tokens are formed and both remain local to PE then lower priority token is stacked on stack STACK.

Encoding of EN1 and EN2	
EN_i	$token_i$ is generated on ALU condition
00	always
01	$= 0$
10	never
11	$\neq 0$

K1 specifies the construction of *token*1 as follows:

Encoding of K1		
K1	*token*1 TAG	*token*1 VALUE
00	*tag*1	*Y*
01	*tag*1	*B*
10	*Y*	*tag*1
11	*B*	*Y*

K2 specifies the construction of *token*2 as follows:

Encoding of K2		
K2	*token*2 TAG	*token*2 VALUE
00	*tag*2	*Y*
01	*Y*	*B*
10	*tag*1	*tag*2
11	*Y*	*tag*2

ORD specifies the relative ranking of *token*1 and *token*2 in the case that both are generated and both remain local to PE.

Encoding of ORD		
ORD	Recirculated	Stacked
0	*token*1	*token*2
1	*token*2	*token*1

A single token or the higher priority token, as defined by ORD is controlled by RECIRC.

Encoding of RECIRC	
RECIRC	Action
00	Normal recirculation
01	Uninterruptible recirculation
10	Push on *stack*0
11	Push on *stack*1

In the case of two tokens, the lower priority token is *always* stacked even if RECIRC designates stacking of the higher priority token. If both RECIRC and STACK specify the same stack then the lower priority token is pushed first.

Encoding of STACK	
STACK	Action
0	Push on *stack*0
1	Push on *stack*1

Finally, the form token section may demand a positive (*i.e.*, circuit-switched) acknowledgment for a network-bound token.

Encoding of ACK	
ACK	Action
0	Immediately process local token
1	Release local token only after network ack

The local token *must* be designated as low priority *stack*1. The hardware inhibits release of a non-acked local token by inhibiting *stack*1 pops until *all* outstanding acks have been received.

6.9 Stack Management

There are two token stacks, *stack*0 and *stack*1, where *stack*0 has priority over *stack*1. That is, *stack*1 will be popped only if *stack*0 is empty.

In the case of a single processor without a network connection there are three possible results of the form token section:

Form Token Results Without Network	
Form Token Output	Action
no tokens	if *stack*0 not empty then pop *stack*0 else if *stack*1 not empty then pop *stack*1 else insert idle token (IP = 0).
*k*1 or *k*2	if RECIRC = 0X then recirculate token else push on RECIRC stack and insert idle token.
*k*1 and *k*2	if RECIRC = 0X then recirculate ORD token, push other else push non-ORD then push ORD, insert idle.

Note that if the MCU control bits NOPOP0 or NOPOP1 are set then, for purposes of the above algorithm, the associated stack always tests empty. If the MCU STACKSWAP bit is set then roles of *stack*0 and *stack*1 are reversed.

When a network is involved things get a little more complex. An incoming message has priority over everything except an uninterruptible recirculating token. If the form token section produces no tokens, then the incoming token is recirculated (instead of either popping a stack or inserting an idle, as above). If one token is produced, and it is interruptible, then the token is pushed (on *stack*0 by default) and the incoming message is recirculated. If two tokens are produced then both are pushed (first the low priority then the high priority, again the high priority defaulting to *stack*0) and the incoming message is recirculated. If a produced token is uninterruptible then the incoming message is blocked until the next cycle. *Incoming network tokens are never pushed onto a stack and are always processed in FIFO order*

Outgoing messages are routed directly to the network, but to avoid deadlock, they will be pushed onto *stack*0 if the outgoing network connection is presently blocked. This implies that a token popped from a stack may head for the network rather than be recirculated.

7 Exceptions

Exceptions can be generated from a masked set of function unit status outputs. When an exception is detected the current A and B values, along with the status outputs, are recorded into temporary registers that are accessible through the MCU. An exception handler is invoked and is passed the TAG of the offending activity. Exception handlers are entered as critical sections RECIRC = 01 so there is the possibility that there will be as many exception handlers active simultaneously as there are pipeline stages[3]. The hardware sorts this out by maintaining eight distinct contexts for the faulting A and B values and transparently provides the correct value for each active exception.

Similarly, the system programmer must reserve eight activation frames, one for each possible pipeline thread. Aside from these contexts there is no hardware vectoring for exceptions. So the first thing an exception handler must do is determine the exception class by fetching the faulting instruction and examining the saved status bits. The exception mask EMASK has the following structure:

EMASK									
SENSE	ALWAYS	DIVZ	UF	OF	INX	NaN	DEN	ZERO	NEG
1	1	1	1	1	1	1	1	1	1

[3] If an exception occurs within an exception handler before it clears the exception flag in the MCU, an unrecoverable machine check occurs.

where,

SENSE Exception occurs when any unmasked bit is asserted (SENSE = 0) or when no unmasked bit is asserted (SENSE = 1).

ALWAYS Always asserted.

DIVZ Asserted when FPU detects divide by zero.

UF Asserted when FPU or PIU detects an underflow.

OF Asserted when FPU or PIU detects an overflow.

INX Asserted when FPU result is inexact.

NAN Asserted when an FPU input is not a number.

DEN Asserted when an FPU input is denormalized.

ZERO Asserted when FPU result is zero.

NEG Asserted when FPU result is negative.

An exception is masked when the corresponding EMASK bit is set to zero. A masked version of the status word is recorded in the MCU and can be retrieved by a GET-STATUS operation. Similarly, the associated A and B values can be obtained by an MCU GETA and GETB. The exception handler is invoked with a TAG that has FP $= 0$ and an IP $= n + 1$ where n is the logical pipeline thread that received the exception. The VALUE part contains the TAG that caused the exception.

8 Statistics

The hardware monitoring function consists of a set of sixteen 32 bit counters. The STATS field selects one of the counters and increments it. The host processor or MCU can read and/or reset any of the counters.