Exhibit U
Part 3

Pin summary

A20M#	I	**Address bit 20 Mask** is an emulator signal.
A31..A3	IO	**Address**, in combination with **byte enable**, indicate the physical addresses of memory or device that is the target of a bus transaction. This signal is an output, when the processor is initiating the bus transaction, and an input when the processor is receiving an inquire transaction or snooping another processor's bus transaction.
ADS#	IO	**ADdress Strobe**, when asserted, indicates new bus transaction by the processor, with valid **address** and **byte enable** simultaneously driven.
ADSC#	O	**Address Strobe Copy** is driven identically to **address strobe**
AHOLD	I	**Address HOLD**, when asserted, causes the processor to cease driving **address** and **address parity** in the next bus clock cycle.
AP	IO	**Address Parity** contains even parity on the same cycle as **address**. **Address parity** is generated by the processor when **address** is an output, and is checked when **address** is an input. A parity error causes a bus error machine check.
APCHK#	O	**Address Parity CHecK** is asserted two bus clocks after EADS# if **address parity** is not even parity of **address**.
APICEN	I	**Advanced Programmable Interrupt Controller ENable** is not implemented.
BE7#..BE0#	IO	**Byte Enable** indicates which bytes are the subject of a read or write transaction and are driven on the same cycle as **address**.
BF1..BF0	I	**Bus Frequency** is sampled to permit software to select the ratio of the processor clock to the bus clock.
BOFF#	I	**BackOFF** is sampled on the rising edge of each bus clock, and when asserted, the processor floats bus signals on the next bus clock and aborts the current bus cycle, until the backoff signal is sampled negated.
BP3..BP0	O	**BreakPoint** is an emulator signal.
BRDY#	I	**Bus ReaDY** indicates that valid data is present on data on a read transaction, or that data has been accepted on a write transaction.
BRDYC#	I	**Bus ReaDY Copy** is identical to BRDY#; asserting either signal has the same effect.
BREQ	O	**Bus REQuest** indicates a processor initiated bus request.

FIG. 48

BUSCHK#	I	**BUS CHecK** is sampled on the rising edge of the bus clock, and when asserted, causes a bus error machine check.
CACHE#	O	**CACHE**, when asserted, indicates a cacheable read transaction or a burst write transaction.
CLK	I	bus **CLocK** provides the bus clock timing edge and the frequency reference for the processor clock.
CPUTYP	I	**CPU TYPe**, if low indicates the primary processor, if high, the dual processor.
D/C#	I	**Data/Code** is driven with the address signal to indicate data, code, or special cycles.
D63..D0	IO	**Data** communicates 64 bits of data per **bus clock**.
D/P#	O	**Dual/Primary** is driven (asserted, low) with **address** on the primary processor
DP7..DP0	IO	**Data Parity** contains even parity on the same cycle as **data**. A parity error causes a bus error machine check.
DPEN#	IO	**Dual Processing Enable** is asserted (driven low) by a Dual processor at reset and sampled by a Primary processor at the falling edge of reset.
EADS#	I	**External Address Strobe** indicates that an external device has driven **address** for an inquire cycle.
EWBE#	I	**External Write Buffer Empty** indicates that the external system has no pending write.
FERR#	O	**Floating point ERRor** is an emulator signal.
FLUSH#	I	**cache FLUSH** is an emulator signal.
FRCMC#	I	**Functional Redundancy Checking Master/Checker** is not implemented.
HIT#	IO	**HIT** indicates that an inquire cycle or cache snoop hits a valid line.
HITM#	IO	**HIT to a Modfied line** indicates that an inquire cycle or cache snoop hits a sub-block in the M cache state.
HLDA	O	**bus HoLD Acknowlege** is asserted (driven high) to acknowlege a **bus hold request**
HOLD	I	**bus HOLD request** causes the processor to float most of its pins and assert **bus hold acknowlege** after completing all outstanding bus transactions, or during reset.
IERR#	O	**Internal ERRor** is an emulator signal.
IGNNE#	I	**IGNore Numeric Error** is an emulator signal.
INIT	I	**INITialization** is an emulator signal.
INTR	I	**maskable INTeRrupt** is an emulator signal.
INV	I	**INValidation** controls whether to invalidate the addressed cache sub-block on an inqure transaction.

FIG. 48 *continued*

KEN#	I	**Cache ENable** is driven with **address** to indicate that the read or write transaction is cacheable.
LINT1..LINT0	I	**Local INTerrupt** is not implemented.
LOCK#	O	**bus LOCK** is driven starting with **address** and ending after **bus ready** to indicate a locked series of bus transactions.
M/IO#	O	**Memory/Input Output** is driven with **address** to indicate a memory or I/O transaction.
NA#	I	**Next Address** indicates that the external system will accept an **address** for a new bus cycle in two bus clocks.
NMI	I	**Non Maskable Interrupt** is an emulator signal.
PBGNT#	IO	**Private Bus GraNT** is driven between Primary and Dual processors to indicate that bus arbitration has completed, granting a new master access to the bus.
PBREQ#	IO	**Private Bus REQuest** is driven between Primary and Dual processors to request a new master access to the bus.
PCD	O	**Page Cache Disable** is driven with address to indicate a not cacheable transaction.
PCHK#	O	**Parity CHecK** is asserted (driven low) two bus clocks after **data** appears with odd parity on enabled bytes.
PHIT#	IO	**Private HIT** is driven between Primary and Dual processors to indicate that the current read or write transaction addresses a valid cache sub-block in the slave processor.
PHITM#	IO	**Private HIT Modified** is driven between Primary and Dual processors to indicate that the current read or write transaction addresses a modified cache sub-block in the slave processor.
PICCLK	I	**Programmable Interrupt Controller CLocK** is not implemented.
PICD1..PICD0	IO	**Programmable Interrupt Controller Data** is not implemented.
PEN#	I	**Parity Enable**, if active on the data cycle, allows a parity error to cause a bus error machine check.
PM1..PM0	O	**Performance Monitoring** is an emulator signal.
PRDY	O	**Probe ReaDY** is not implemented.
PWT	O	**Page Write Through** is driven with **address** to indicate a not write allocate transaction.
R/S#	I	**Run/Stop** is not implemented.
RESET	I	**RESET** causes a processor reset.
SCYC	O	**Split CYCle** is asserted during **bus lock** to indicate that more than two transactions are in the series of bus transactions.

FIG. 48 *continued*

SMI#	I	**System Managem nt Interrupt** is an emulator signal.
SMIACT#	O	**System Management Interrupt ACTive** is an emulator signal.
STPCLK#	I	**SToP CLocK** is an emulator signal.
TCK	I	**Test CLocK** follows IEEE 1149.1.
TDI	I	**Test Data Input** follows IEEE 1149.1.
TDO	O	**Test Data Output** follows IEEE 1149.1.
TMS	I	**Test Mode Select** follows IEEE 1149.1.
TRST#	I	**Test ReSeT** follows IEEE 1149.1.
VCC2	I	VCC of 2.8V at 25 pins
VCC3	I	VCC of 3.3V at 28 pins
VCC2DET#	O	VCC2 DETect sets appropriate VCC2 voltage level.
VSS	I	VSS supplied at 53 pins
W/R#	O	**Write/Read** is driven with address to indicate write vs. read transaction.
WB/WT#	I	**Write Back/Write Through** is returned to indicate that data is permitted to be cached as write back.

FIG. 48 *continued*

Electrical Specifications

Clock rate	66 MHz		75 MHz		100 MHz		133 MHz		
Parameter	min	max	min	max	min	max	min	max	unit
CLK frequency	33.3	66.7	37.5	75	50	100		133	MHz
CLK period	15.0	30.0	13.3	26.3	10.0	20.0			ns
CLK high time (≥2v)	4.0		4.0		3.0				ns
CLK low time (≤0.8V)	4.0		4.0		3.0				ns
CLK rise time (0.8V->2V)	0.15	1.5	0.15	1.5	0.15	1.5			ns
CLK fall time (2V->0.8V)	0.15	1.5	0.15	1.5	0.15	1.5			ns
CLK period stability		250		250		250			ps

FIG. 49A

A31..3 valid delay	1.1	6.3	1.1	4.5	1.1	4.0			ns
A31..3 float delay		10.0		7.0		7.0			ns
ADS# valid delay	1.0	6.0	1.0	4.5	1.0	4.0			ns
ADS# float delay		10.0		7.0		7.0			ns
ADSC# valid delay	1.0	7.0	1.0	4.5	1.0	4.0			ns
ADSC# float delay		10.0		7.0		7.0			ns
AP valid delay	1.0	8.5	1.0	5.5	1.0	5.5			ns
AP float delay		10.0		7.0		7.0			ns
APCHK# valid delay	1.0	8.3	1.0	4.5	1.0	4.5			ns
BE7..0# valid delay	1.0	7.0	1.0	4.5	1.0	4.0			ns
BE7..0# float delay		10.0		7.0		7.0			ns
BP3..0 valid delay	1.0	10.0							ns
BREQ valid delay	1.0	8.0	1.0	4.5	1.0	4.0			ns
CACHE# valid delay	1.0	7.0	1.0	4.5	1.0	4.0			ns
CACHE# float delay		10.0		7.0		7.0			ns
D/C# valid delay	1.0	7.0	1.0	4.5	1.0	4.0			ns
D/C# float delay		10.0		7.0		7.0			ns
D63..0 write data valid delay	1.3	7.5	1.3	4.5	1.3	4.5			ns
D63..0 write data float delay		10.0		7.0		7.0			ns
DP7..0 write data valid delay	1.3	7.5	1.3	4.5	1.3	4.5			ns
DP7..0 write data float delay		10.0		7.0		7.0			ns
FERR# valid delay	1.0	8.3	1.0	4.5	1.0	4.5			ns
HIT# valid delay	1.0	6.8	1.0	4.5	1.0	4.0			ns
HITM# valid delay	1.1	6.0	1.1	4.5	1.1	4.0			ns
HLDA valid delay	1.0	6.8	1.0	4.5	1.0	4.0			ns
IERR# valid delay	1.0	8.3							ns
LOCK# valid delay	1.1	7.0	1.1	4.5	1.1	4.0			ns
LOCK# float delay		10.0		7.0		7.0			ns
M/IO# valid delay	1.0	5.9	1.0	4.5	1.0	4.0			ns
M/IO# float delay		10.0		7.0		7.0			ns
PCD valid delay	1.0	7.0	1.0	4.5	1.0	4.0			ns
PCD float delay		10.0		7.0		7.0			ns
PCHK# valid delay	1.0	7.0	1.0	4.5	1.0	4.5			ns
PM1..0 valid delay	1.0	10.0							ns
PRDY valid delay	1.0	8.0							ns
PWT valid delay	1.0	7.0	1.0	4.5	1.0	4.0			ns
PWT float delay		10.0		7.0		7.0			ns
SCYC valid delay	1.0	7.0	1.0	4.5	1.0	4.0			ns
SCYC float delay		10.0		7.0		7.0			ns
SMIACT# valid delay	1.0	7.3	1.0	4.5	1.0	4.0			ns
W/R# valid delay	1.0	7.0	1.0	4.5	1.0	4.0			ns
W/R# float delay		10.0		7.0		7.0			ns

FIG. 49B

A31..5 setup time	6.0		3.0		3.0				ns
A31..5 hold time	1.0		1.0		1.0				ns
A20M# setup time	5.0		3.0		3.0				ns
A20M# hold time	1.0		1.0		1.0				ns
AHOLD setup time	5.5		3.5		3.5				ns
AHOLD hold time	1.0		1.0		1.0				ns
AP setup time	5.0		1.7		1.7				ns
AP hold time	1.0		1.0		1.0				ns
BOFF# setup time	5.5		3.5		3.5				ns
BOFF# hold time	1.0		1.0		1.0				ns
BRDY# setup time	5.0		3.0		3.0				ns
BRDY# hold time	1.0		1.0		1.0				ns
BRDYC# setup time	5.0		3.0		3.0				ns
BRDYC# hold time	1.0		1.0		1.0				ns
BUSCHK# setup time	5.0		3.0		3.0				ns
BUSCHK# hold time	1.0		1.0		1.0				ns
D63..0 read data setup time	2.8		1.7		1.7				ns
D63..0 read data hold time	1.5		1.5		1.5				ns
DP7..0 read data setup time	2.8		1.7		1.7				ns
DP7..0 read data hold time	1.5		1.5		1.5				ns
EADS# setup time	5.0		3.0		3.0				ns
EADS# hold time	1.0		1.0		1.0				ns
EWBE# setup time	5.0		1.7		1.7				ns
EWBE# hold time	1.0		1.0		1.0				ns
FLUSH# setup time	5.0		1.7		1.7				ns
FLUSH# hold time	1.0		1.0		1.0				ns
FLUSH# async pulse width	2		2		2				CLK
HOLD setup time	5.0		1.7		1.7				ns
HOLD hold time	1.5		1.5		1.5				ns
IGNNE# setup time	5.0		1.7		1.7				ns
IGNNE# hold time	1.0		1.0		1.0				ns
IGNNE# async pulse width	2		2		2				CLK
INIT setup time	5.0		1.7		1.7				ns
INIT hold time	1.0		1.0		1.0				ns
INIT async pulse width	2		2		2				CLK
INTR setup time	5.0		1.7		1.7				ns
INTR hold time	1.0		1.0		1.0				ns
INV setup time	5.0		1.7		1.7				ns
INV hold time	1.0		1.0		1.0				ns
KEN# setup time	5.0		3.0		3.0				ns
KEN# hold time	1.0		1.0		1.0				ns
NA# setup time	4.5		1.7		1.7				ns

FIG. 49C

NA# hold time	1.0		1.0		1.0				ns
NMI setup time	5.0		1.7		1.7				ns
NMI hold time	1.0		1.0		1.0				ns
NMI async pulse width	2		2		2				CLK
PEN# setup time	4.8		1.7		1.7				ns
PEN# hold time	1.0		1.0		1.0				ns
R/S# setup time	5.0		1.7		1.7				ns
R/S# hold time	1.0		1.0		1.0				ns
R/S# async pulse width	2		2		2				CLK
SMI# setup time	5.0		1.7		1.7				ns
SMI# hold time	1.0		1.0		1.0				ns
SMI# async pulse width	2		2		2				CLK
STPCLK# setup time	5.0		1.7		1.7				ns
STPCLK# hold time	1.0		1.0		1.0				ns
WB/WT# setup time	4.5		1.7		1.7				ns
WB/WT# hold time	1.0		1.0		1.0				ns

FIG. 49C *continued*

RESET setup time	5.0		1.7		1.7				ns
RESET hold time	1.0		1.0		1.0				ns
RESET pulse width	15		15		15				CLK
RESET active	1.0		1.0		1.0				ms
BF2..0 setup time	1.0		1.0		1.0				ms
BF2..0 hold time	2		2		2				CLK
BRDYC# hold time	1.0		1.0		1.0				ns
BRDYC# setup time	2		2		2				CLK
BRDYC# hold time	2		2		2				CLK
FLUSH# setup time	5.0		1.7		1.7				ns
FLUSH# hold time	1.0		1.0		1.0				ns
FLUSH# setup time	2		2		2				CLK
FLUSH# hold time	2		2		2				CLK

FIG. 49D

PBREQ# flight time	0	2.0							ns
PBGNT# flight time	0	2.0							ns
PHIT# flight time	0	2.0							ns
PHITM# flight time	0	1.8							ns
A31..5 setup time	3.7								ns
A31..5 hold time	0.8								ns
D/C# setup time	4.0								ns
D/C# hold time	0.8								ns
W/R# setup time	4.0								ns
W/R# hold time	0.8								ns
CACHE# setup time	4.0								ns
CACHE# hold time	1.0								ns
LOCK# setup time	4.0								ns
LOCK# hold time	0.8								ns
SCYC setup time	4.0								ns
SCYC hold time	0.8								ns
ADS# setup time	5.8								ns
ADS# hold time	0.8								ns
M/IO# setup time	5.8								ns
M/IO# hold time	0.8								ns
HIT# setup time	6.0								ns
HIT# hold time	1.0								ns
HITM# setup time	6.0								ns
HITM# hold time	0.7								ns
HLDA setup time	6.0								ns
HLDA hold time	0.8								ns
DPEN# valid time		10.0							CLK
DPEN# hold time	2.0								CLK
D/P# valid delay (primary)	1.0	8.0							ns

FIG. 49E

TCK frequency		25				25			MHz
TCK period	40.0				40.0				ns
TCK high time (≥2v)	14.0				14.0				ns
TCK low time (≤0.8V)	14.0				14.0				ns
TCK rise time (0.8V->2V)		5.0				5.0			ns
TCK fall time (2V->0.8V)		5.0				5.0			ns
TRST# pulse width	30.0				30.0				ns

FIG. 49F

TDI setup time	5.0				5.0				ns
TDI hold time	9.0				9.0				ns
TMS setup time	5.0				5.0				ns
TMS hold time	9.0				9.0				ns
TDO valid delay	3.0	13.0			3.0	13.0			ns
TDO float delay		16.0				16.0			ns
all outputs valid delay	3.0	13.0			3.0	13.0			ns
all outputs float delay		16.0				16.0			ns
all inputs setup time	5.0				5.0				ns
all inputs hold time	9.0				9.0				ns

FIG. 49G

Operation codes

L. 8[1]	Load signed byte
L. 16.B	Load signed doublet big-endian
L. 16.A.B	Load signed doublet aligned big-endian
L. 16.L	Load signed doublet little-endian
L. 16.A.L	Load signed doublet aligned little-endian
L. 32.B	Load signed quadlet big-endian
L. 32.A.B	Load signed quadlet aligned big-endian
L. 32.L	Load signed quadlet little-endian
L. 32.A.L	Load signed quadlet aligned little-endian
L. 64.B	Load signed octlet big-endian
L. 64.A.B	Load signed octlet aligned big-endian
L. 64.L	Load signed octlet little-endian
L. 64.A.L	Load signed octlet aligned little-endian
L.128.B[2]	Load hexlet big-endian
L.128.A.B[3]	Load hexlet aligned big-endian
L.128.L[4]	Load hexlet little-endian
L.128.A.L[5]	Load hexlet aligned little-endian
L.U. 8[6]	Load unsigned byte
L.U. 16.B	Load unsigned doublet big-endian
L.U. 16.A.B	Load unsigned doublet aligned big-endian
L.U. 16.L	Load unsigned doublet little-endian
L.U. 16.A.L	Load unsigned doublet aligned little-endian
L.U. 32.B	Load unsigned quadlet big-endian
L.U. 32.A.B	Load unsigned quadlet aligned big-endian
L.U. 32.L	Load unsigned quadlet little-endian
L.U. 32.A.L	Load unsigned quadlet aligned little-endian
L.U. 64.B	Load unsigned octlet big-endian
L.U. 64.A.B	Load unsigned octlet aligned big-endian
L.U. 64.L	Load unsigned octlet little-endian
L.U. 64.A.L	Load unsigned octlet aligned little-endian

FIG 50A

Selection

number format	type	size	alignment	ordering
signed byte		8		
unsigned byte	U	8		
signed integer		16 32 64		L B
signed integer aligned		16 32 64	A	L B
unsigned integer	U	16 32 64		L B
unsigned integer aligned	U	16 32 64	A	L B
register		128		L B
register aligned		128	A	L B

Format

op rd=rc,rb

rd=op(rc,rb)

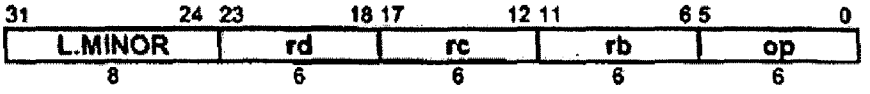


FIG. 50B

Definition

```
def Load(op,rd,rc,rb) as
    case op of
        L16L, L32L, L8, L16AL, L32AL, L16B, L32B, L16AB, L32AB,
        L64L, L64AL, L64B, L64AB:
            signed ← true
        LU16L, LU32L, LU8, LU16AL, LU32AL, LU16B, LU32B, LU16AB, LU32AB,
        LU64L, LU64AL, LU64B, LU64AB:
            signed ← false
        L128L, L128AL, L128B, L128AB:
            signed ← undefined
    endcase
    case op of
        L8, LU8:
            size ← 8
        L16L, LU16L, L16AL, LU16AL, L16B, LU16B, L16AB, LU16AB:
            size ← 16
        L32L, LU32L, L32AL, LU32AL, L32B, LU32B, L32AB, LU32AB:
            size ← 32
        L64L, LU64L, L64AL, LU64AL, L64B, LU64B, L64AB, LU64AB:
            size ← 64
        L128L, L128AL, L128B, L128AB:
            size ← 128
    endcase
    lsize ← log(size)
    case op of
        L16L, LU16L, L32L, LU32L, L64L, LU64L, L128L,
        L16AL, LU16AL, L32AL, LU32AL, L64AL, LU64AL, L128AL:
            order ← L
        L16B, LU16B, L32B, LU32B, L64B, LU64B, L128B,
        L16AB, LU16AB, L32AB, LU32AB, L64AB, LU64AB, L128AB:
            order ← B
        L8, LU8:
            order ← undefined
    endcase
```

FIG. 50C

```
        c ← RegRead(rc, 64)
        b ← RegRead(rb, 64)
        VirtAddr ← c + (b66-lsize..0 || 0lsize-3)
        case op of
            L16AL, LU16AL, L32AL, LU32AL, L64AL, LU64AL, L128AL,
            L16AB, LU16AB, L32AB, LU32AB, L64AB, LU64AB, L128AB:
                if (clsize-4..0 ≠ 0 then
                    raise AccessDisallowedByVirtualAddress
                endif
            L16L, LU16L, L32L, LU32L, L64L, LU64L, L128L,
            L16B, LU16B, L32B, LU32B, L64B, LU64B, L128B:
            L8, LU8:
        endcase
        m ← LoadMemory(c,VirtAddr,size,order)
        a ← (msize-1 and signed)128-size || m
        RegWrite(rd, 128, a)
enddef
```

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 50C *continued*

Operation codes

L.I. 8[1]	Load immediate signed byte
L.I. 16.A.B	Load immediate signed doublet aligned big-endian
L.I. 16.B	Load immediate signed doublet big-endian
L.I. 16.A.L	Load immediate signed doublet aligned little-endian
L.I. 16.L	Load immediate signed doublet little-endian
L.I. 32.A.B	Load immediate signed quadlet aligned big-endian
L.I. 32.B	Load immediate signed quadlet big-endian
L.I. 32.A.L	Load immediate signed quadlet aligned little-endian
L.I. 32.L	Load immediate signed quadlet little-endian
L.I. 64.A.B	Load immediate signed octlet aligned big-endian
L.I. 64.B	Load immediate signed octlet big-endian
L.I. 64.A.L	Load immediate signed octlet aligned little-endian
L.I. 64.L	Load immediate signed octlet little-endian
L.I.128.A.B[2]	Load immediate hexlet aligned big-endian
L.I.128.B[3]	Load immediate hexlet big-endian
L.I.128.A.L[4]	Load immediate hexlet aligned little-endian
L.I.128.L[5]	Load immediate hexlet little-endian
L.I.U. 8[6]	Load immediate unsigned byte
L.I.U. 16.A.B	Load immediate unsigned doublet aligned big-endian
L.I.U. 16.B	Load immediate unsigned doublet big-endian
L.I.U. 16.A.L	Load immediate unsigned doublet aligned little-endian
L.I.U. 16.L	Load immediate unsigned doublet little-endian
L.I.U. 32.A.B	Load immediate unsigned quadlet aligned big-endian
L.I.U. 32.B	Load immediate unsigned quadlet big-endian
L.I.U. 32.A.L	Load immediate unsigned quadlet aligned little-endian
L.I.U. 32.L	Load immediate unsigned quadlet little-endian
L.I.U. 64.A.B	Load immediate unsigned octlet aligned big-endian
L.I.U. 64.B	Load immediate unsigned octlet big-endian
L.I.U. 64.A.L	Load immediate unsigned octlet aligned little-endian
L.I.U. 64.L	Load immediate unsigned octlet little-endian

FIG 51A

Selection

number format	type	size	alignment	ordering
signed byte		8		
unsigned byte	U	8		
signed integer		16 32 64		L B
signed integer aligned		16 32 64	A	L B
unsigned integer	U	16 32 64		L B
unsigned integer aligned	U	16 32 64	A	L B
register		128		L B
register aligned		128	A	L B

Format

op rd=rc,offset

rd=op(rc,offset)

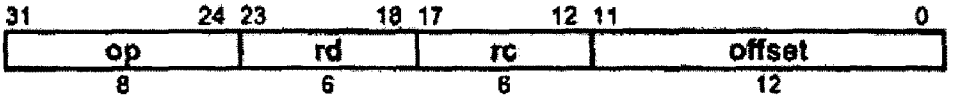


FIG. 51B

Definition

```
def LoadImmediate(op,rd,rc,offset) as
    case op of
        LI16L, LI32L, LI8, LI16AL, LI32AL, LI16B, LI32B, LI16AB, LI32AB:
        LI64L, LI64AL, LI64B, LI64AB:
            signed ← true
        LIU16L, LIU32L, LIU8, LIU16AL, LIU32AL,
        LIU16B, LIU32B, LIU16AB, LIU32AB:
        LIU64L, LIU64AL, LIU64B, LIU64AB:
            signed ← false
        LI128L, LI128AL, LI128B, LI128AB:
            signed ← undefined
    endcase
    case op of
        LI8, LIU8:
            size ← 8
        LI16L, LIU16L, LI16AL, LIU16AL, LI16B, LIU16B, LI16AB, LIU16AB:
            size ← 16
        LI32L, LIU32L, LI32AL, LIU32AL, LI32B, LIU32B, LI32AB, LIU32AB:
            size ← 32
        LI64L, LIU64L, LI64AL, LIU64AL, LI64B, LIU64B, LI64AB, LIU64AB:
            size ← 64
        LI128L, LI128AL, LI128B, LI128AB:
            size ← 128
    endcase
    lsize ← log(size)
    case op of
        LI16L, LIU16L, LI32L, LIU32L, LI64L, LIU64L, LI128L,
        LI16AL, LIU16AL, LI32AL, LIU32AL, LI64AL, LIU64AL, LI128AL:
            order ← LI
        LI16B, LIU16B, LI32B, LIU32B, LI64B, LIU64B, LI128B,
        LI16AB, LIU16AB, LI32AB, LIU32AB, LI64AB, LIU64AB, LI128AB:
            order ← B
        LI8, LIU8:
            order ← undefined
    endcase
```

FIG. 51C

```
    c ← RegRead(rc, 64)
    VirtAddr ← c + (offset[55-lsize] || offset || 0^(lsize-3))
    case op of
        LI16AL, LIU16AL, LI32AL, LIU32AL, LI64AL, LIU64AL, LI128AL,
        LI16AB, LIU16AB, LI32AB, LIU32AB, LI64AB, LIU64AB, LI128AB:
            if (c_(lsize-4..0) ≠ 0 then
                raise AccessDisallowedByVirtualAddress
            endif
        LI16L, LIU16L, LI32L, LIU32L, LI64L, LIU64L, LI128L,
        LI16B, LIU16B, LI32B, LIU32B, LI64B, LIU64B, LI128B:
        LI8, LIU8:
    endcase
    m ← LoadMemory(c,VirtAddr,size,order)
    a ← (m_(size-1) and signed)^(128-size) || m
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 51C *continued*

Operation codes

S. 8[1]	Store byte
S. 16.B	Store double big-endian
S. 16.A.B	Store double aligned big-endian
S. 16.L	Store double little-endian
S. 16.A.L	Store double aligned little-endian
S. 32.B	Store quadlet big-endian
S. 32.A.B	Store quadlet aligned big-endian
S. 32.L	Store quadlet little-endian
S. 32.A.L	Store quadlet aligned little-endian
S. 64.B	Store octlet big-endian
S. 64.A.B	Store octlet aligned big-endian
S. 64.L	Store octlet little-endian
S. 64.A.L	Store octlet aligned little-endian
S.128.B	Store hexlet big-endian
S.128.A.B	Store hexlet aligned big-endian
S.128.L	Store hexlet little-endian
S.128.A.L	Store hexlet aligned little-endian
S.MUX.64.A.B	Store multiplex octlet aligned big-endian
S.MUX.64.A.L	Store multiplex octlet aligned little-endian

FIG 52A

Selection

number format	op	size	alignment	ordering
byte		8		
integer		16 32 64 128		L B
integer aligned		16 32 64 128	A	L B
multiplex	MUX	64	A	L B

Format

op rd,rc,rb

op(rd,rc,rb)

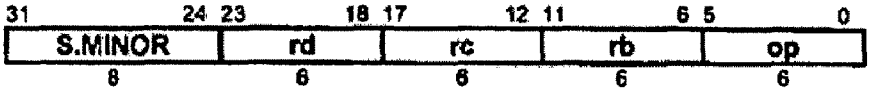


FIG. 52B

Definition

```
def Store(op,rd,rc,rb) as
    case op of
        S8:
            size ← 8
        S16L, S16AL, S16B, S16AB:
            size ← 16
        S32L, S32AL, S32B, S32AB:
            size ← 32
        S64L, S64AL, S64B, S64AB,
        SMUX64AB, SMUX64AL:
            size ← 64
        S128L, S128AL, S128B, S128AB:
            size ← 128
    endcase
    lsize ← log(size)
    case op of
        S8:
            order ← undefined
        S16L, S32L, S64L, S128L,
        S16AL, S32AL, S64AL, S128AL, SMUX64ALI:
            order ← L
        S16B, S32B, S64B, S128B,
        S16AB, S32AB, S64AB, S128AB, SMUX64ABI:
            order ← B
    endcase
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 64)
    VirtAddr ← c + (b66-lsize..0 || 0lsize-3)
    case op of
        S16AL, S32AL, S64AL, S128AL,
        S16AB, S32AB, S64AB, S128AB,
        SMUX64AB, SMUX64AL:
            if (clsize-4..0 ≠ 0 then
                raise AccessDisallowedByVirtualAddress
            endif
        S16L, S32L, S64L, S128L,
        S16B, S32B, S64B, S128B:
        S8:
    endcase
```

FIG. 52C

```
    d ← RegRead(rd, 128)
    case op of
        S8,
        S16L, S16AL, S16B, S16AB,
        S32L, S32AL, S32B, S32AB,
        S64L, S64AL, S64B, S64AB,
        S128L, S128AL, S128B, S128AB:
            StoreMemory(c,VirtAddr,size,order,d_size-1..0)
        SMUX64AB, SMUX64AL:
            lock
                a ← LoadMemoryW(c,VirtAddr,size,order)
                m ← (d_127..64 & d_63..0) | (a & ~d_63..0)
                StoreMemory(c,VirtAddr,size,order,m)
            endlock
    endcase
enddef
```

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 52C *continued*

Operation codes

S.I. 8[1]	Store immediate byte
S.I. 16.A.B	Store immediate double aligned big-endian
S.I. 16.B	Store immediate double big-endian
S.I. 16.A.L	Store immediate double aligned little-endian
S.I. 16.L	Store immediate double little-endian
S.I. 32.A.B	Store immediate quadlet aligned big-endian
S.I. 32.B	Store immediate quadlet big-endian
S.I. 32.A.L	Store immediate quadlet aligned little-endian
S.I. 32.L	Store immediate quadlet little-endian
S.I. 64.A.B	Store immediate octlet aligned big-endian
S.I. 64.B	Store immediate octlet big-endian
S.I. 64.A.L	Store immediate octlet aligned little-endian
S.I. 64.L	Store immediate octlet little-endian
S.I.128.A.B	Store immediate hexlet aligned big-endian
S.I.128.B	Store immediate hexlet big-endian
S.I.128.A.L	Store immediate hexlet aligned little-endian
S.I.128.L	Store immediate hexlet little-endian
S.MUXI.64.A.B	Store multiplex immediate octlet aligned big-endian
S.MUXI.64.A.L	Store multiplex immediate octlet aligned little-endian

FIG 53A

Selection

number format	op	size	alignment	ordering
byte		8		
integer		16 32 64 128		L B
integer aligned		16 32 64 128	A	L B
multiplex	MUX	64	A	L B

Format

S.op.l.size.align.order rd,rc,offset

sopisizealignorder(rd,rc,offset)

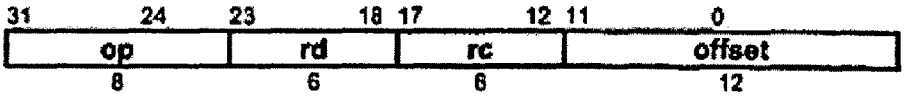


FIG. 53B

Definition

```
def StoreImmediate(op,rd,rc,offset) as
    case op of
        SI8:
            size ← 8
        SI16L, SI16AL, SI16B, SI16AB:
            size ← 16
        SI32L, SI32AL, SI32B, SI32AB:
            size ← 32
        SI64L, SI64AL, SI64B, SI64AB, SMUXI64AB, SMUXI64AL:
            size ← 64
        SI128L, SI128AL, SI128B, SI128AB:
            size ← 128
    endcase
    lsize ← log(size)
    case op of
        SI8:
            order ← undefined
        SI16L, SI32L, SI64L, SI128L,
        SI16AL, SI32AL, SI64AL, SI128AL, SMUXI64AL:
            order ← L
        SI16B, SI32B, SI64B, SI128B,
        SI16AB, SI32AB, SI64AB, SI128AB, SMUXI64AB:
            order ← B
    endcase
    c ← RegRead(rc, 64)
    VirtAddr ← c + (offset[55-lsize] || offset || 0^(lsize-3))
    case op of
        SI16AL, SI32AL, SI64AL, SI128AL,
        SI16AB, SI32AB, SI64AB, SI128AB,
        SMUXI64AB, SMUXI64AL:
            if (c[lsize-4..0] ≠ 0 then
                raise AccessDisallowedByVirtualAddress
            endif
        SI16L, SI32L, SI64L, SI128L,
        SI16B, SI32B, SI64B, SI128B:
        SI8:
    endcase
```

FIG. 53C

```
    d ← RegRead(rd, 128)
    case op of
        SI8,
        SI16L, SI16AL, SI16B, SI16AB,
        SI32L, SI32AL, SI32B, SI32AB,
        SI64L, SI64AL, SI64B, SI64AB,
        SI128L, SI128AL, SI128B, SI128AB:
            StoreMemory(c,VirtAddr,size,order,d_size-1..0)
        SMUXI64AB, SMUXI64AL:
            lock
                a ← LoadMemoryW(c,VirtAddr,size,order)
                m ← (d_127..64 & d_63..0) | (a & ~d_63..0)
                StoreMemory(c,VirtAddr,size,order,m)
            endlock
    endcase
enddef
```

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 53C *continued*

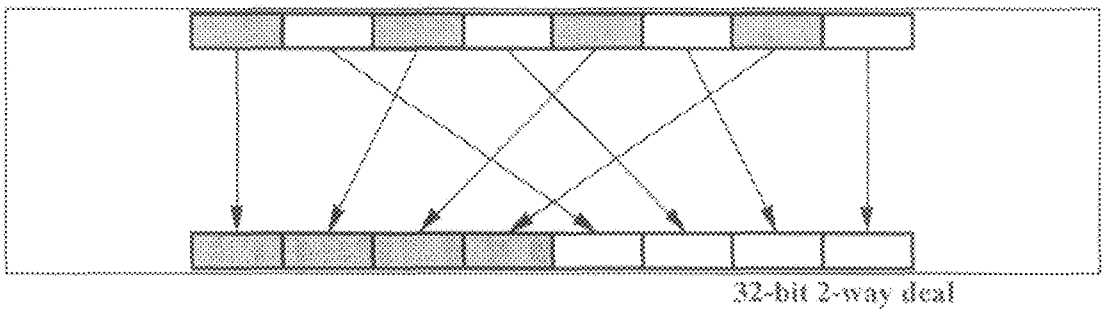


FIG. 54A

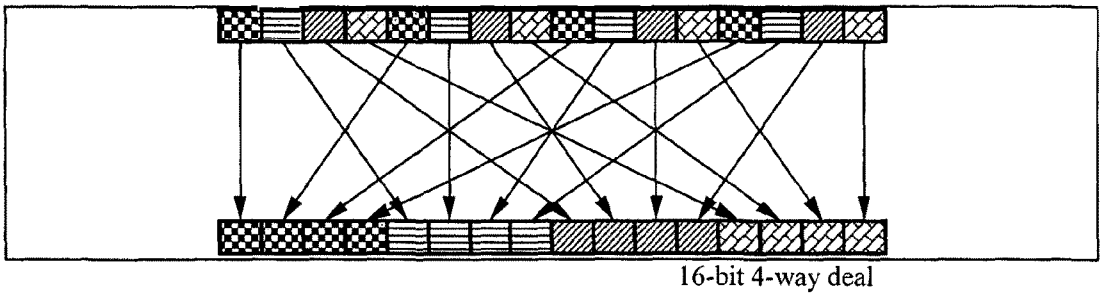


FIG. 54B

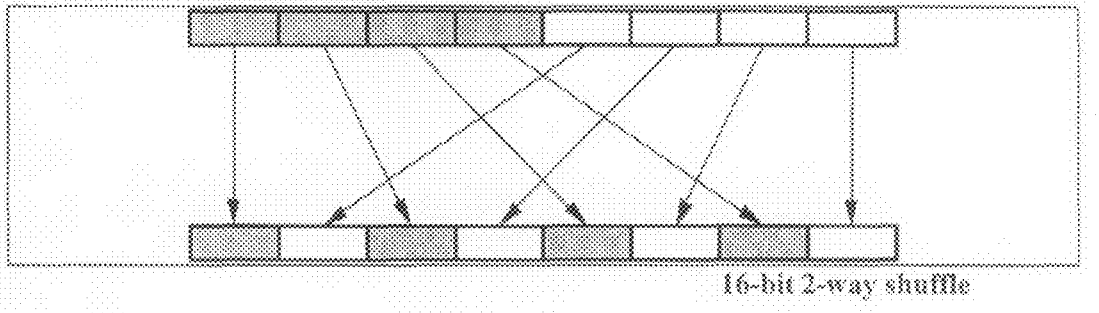


FIG. 54C

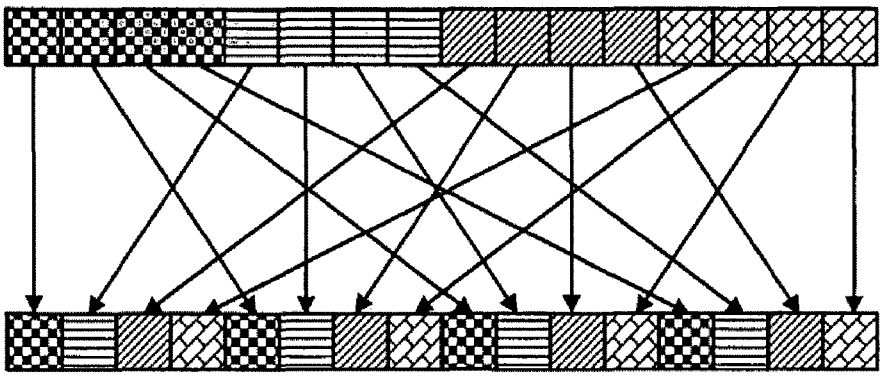


FIG. 54D

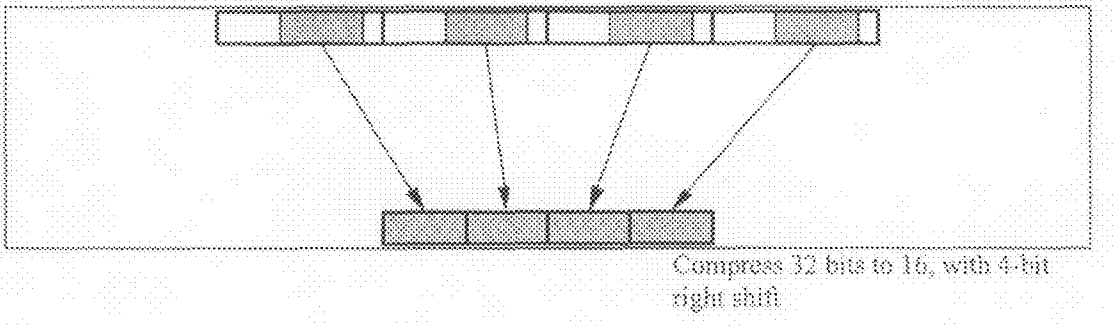


FIG. 34E

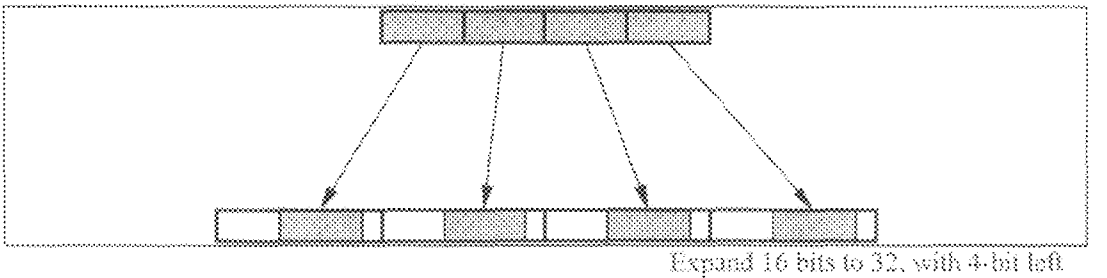


FIG. 54F

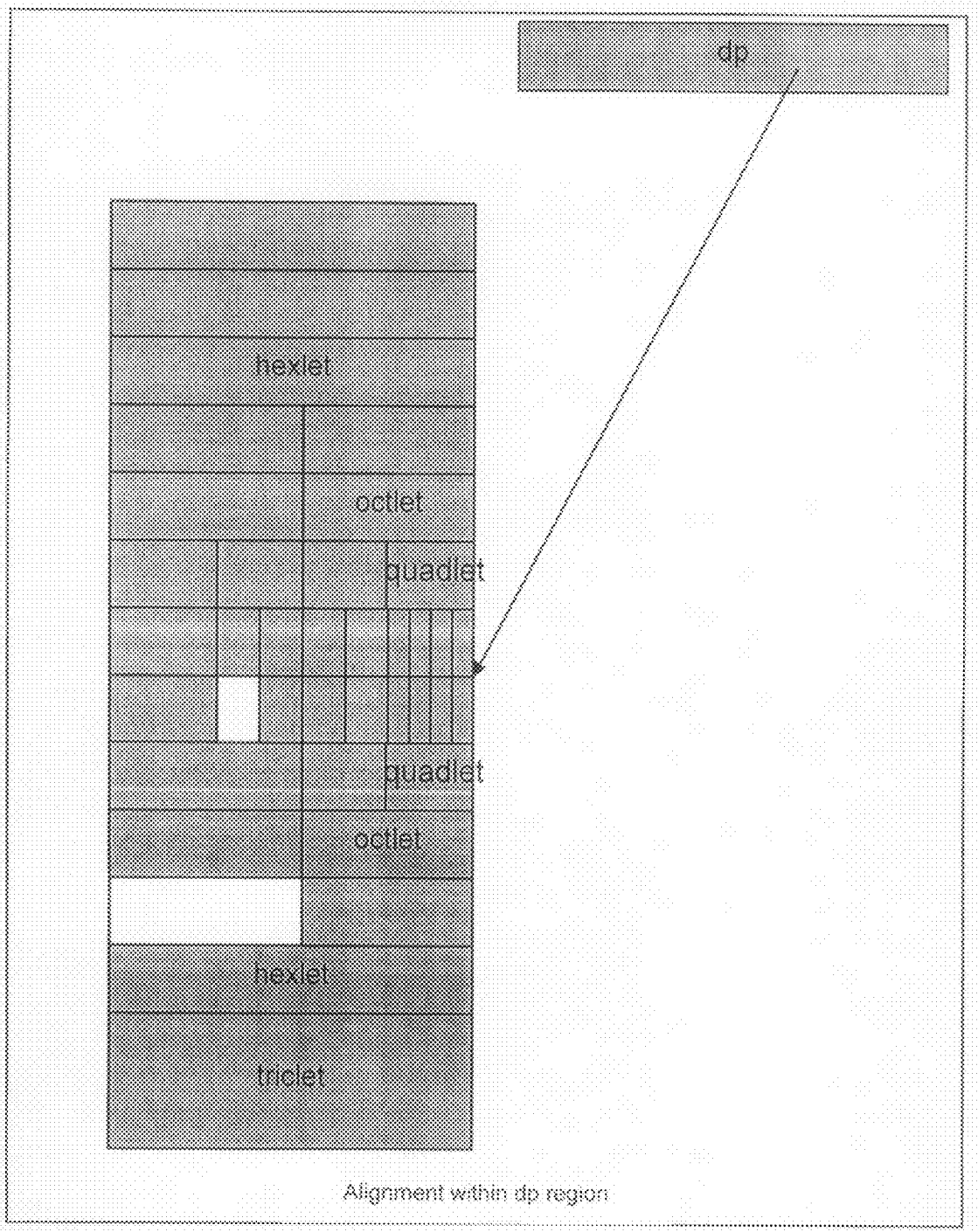

dp
hexlet
octlet
quadlet
quadlet
octlet
hexlet
triclet
Alignment within dp region

FIG. 54G

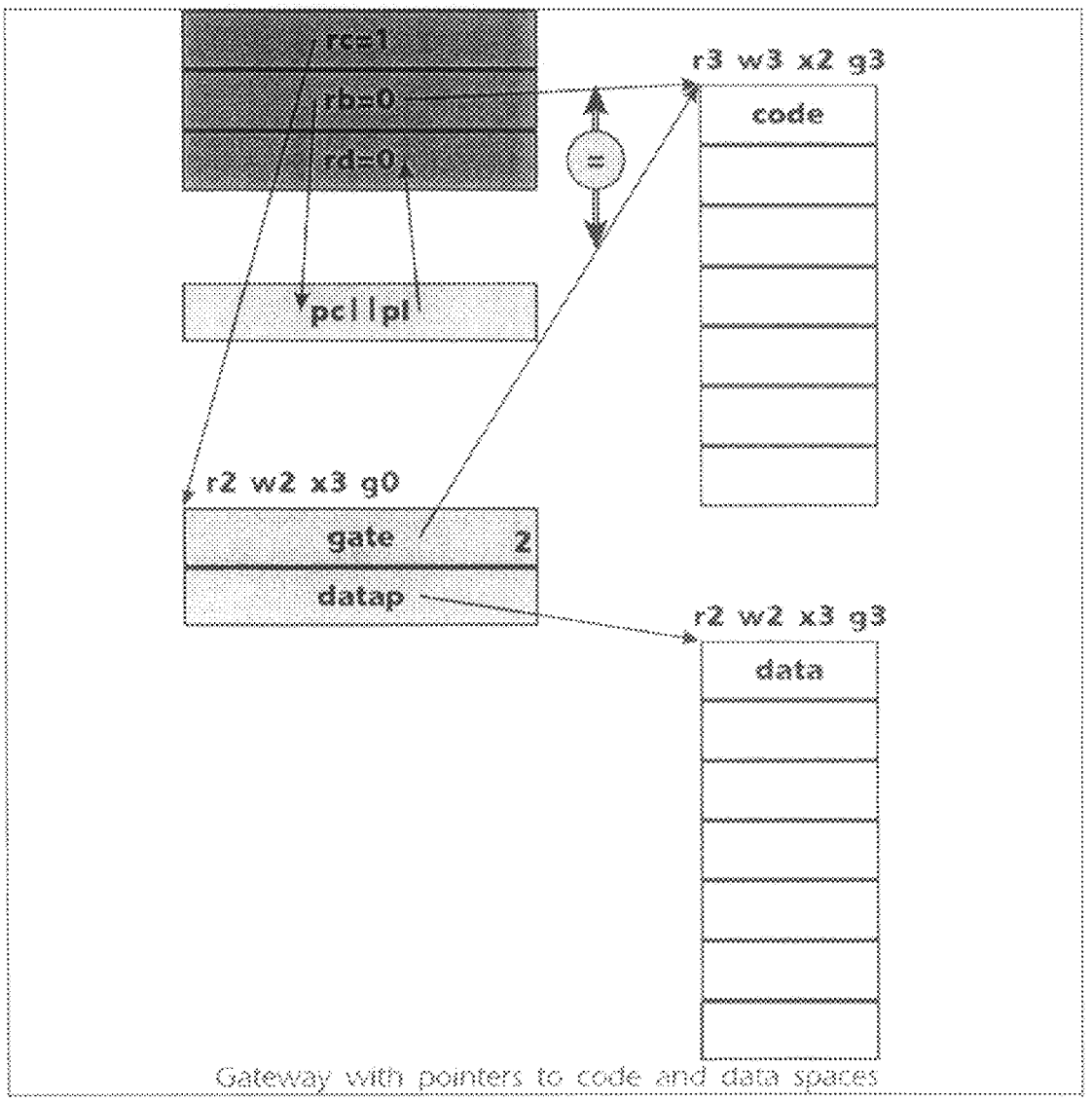

rc=1
rb=0
rd=0
=
pc | pl
r3 w3 x2 g3
code
r2 w2 x3 g0
gate
2
datap
r2 w2 x3 g3
data
Gateway with pointers to code and data spaces

FIG. 54E

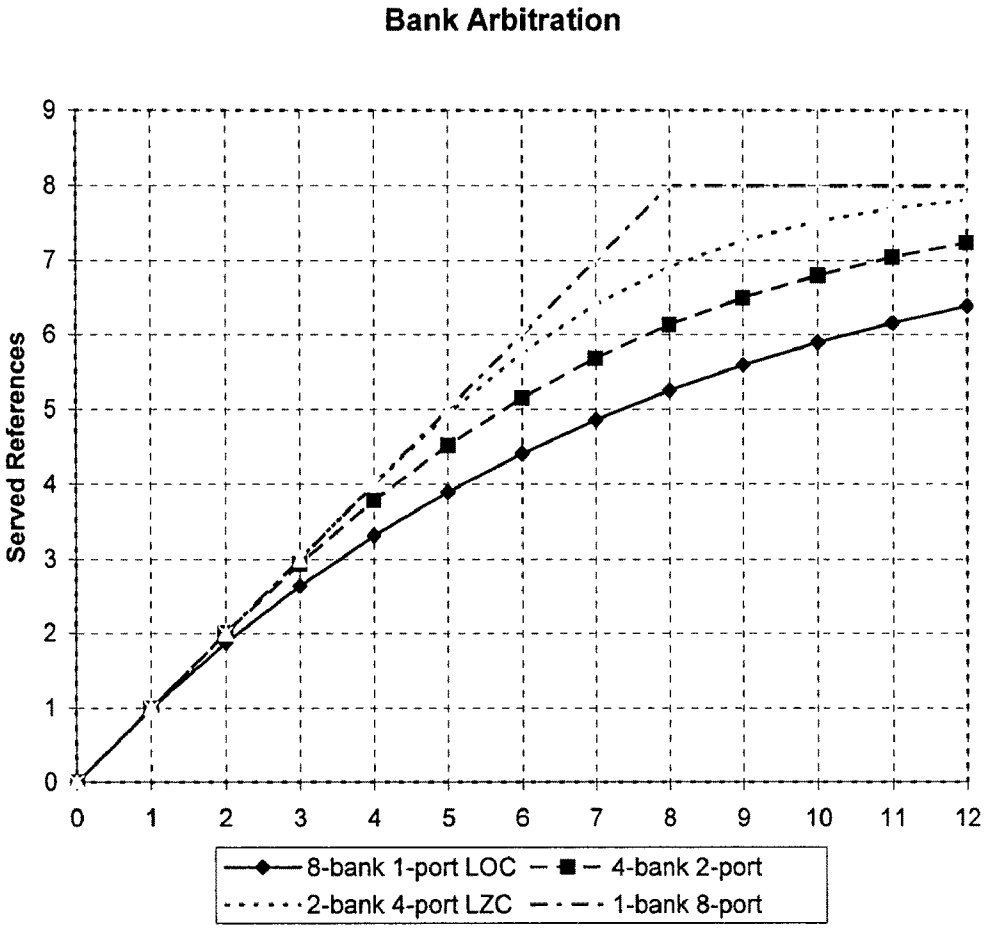

Bank Arbitration
Served References
0
1
2
3
4
5
6
7
8
9
0
1
2
3
4
5
6
7
8
9
10
11
12
8-bank 1-port LOC
4-bank 2-port
2-bank 4-port LZC
1-bank 8-port

FIG. 55

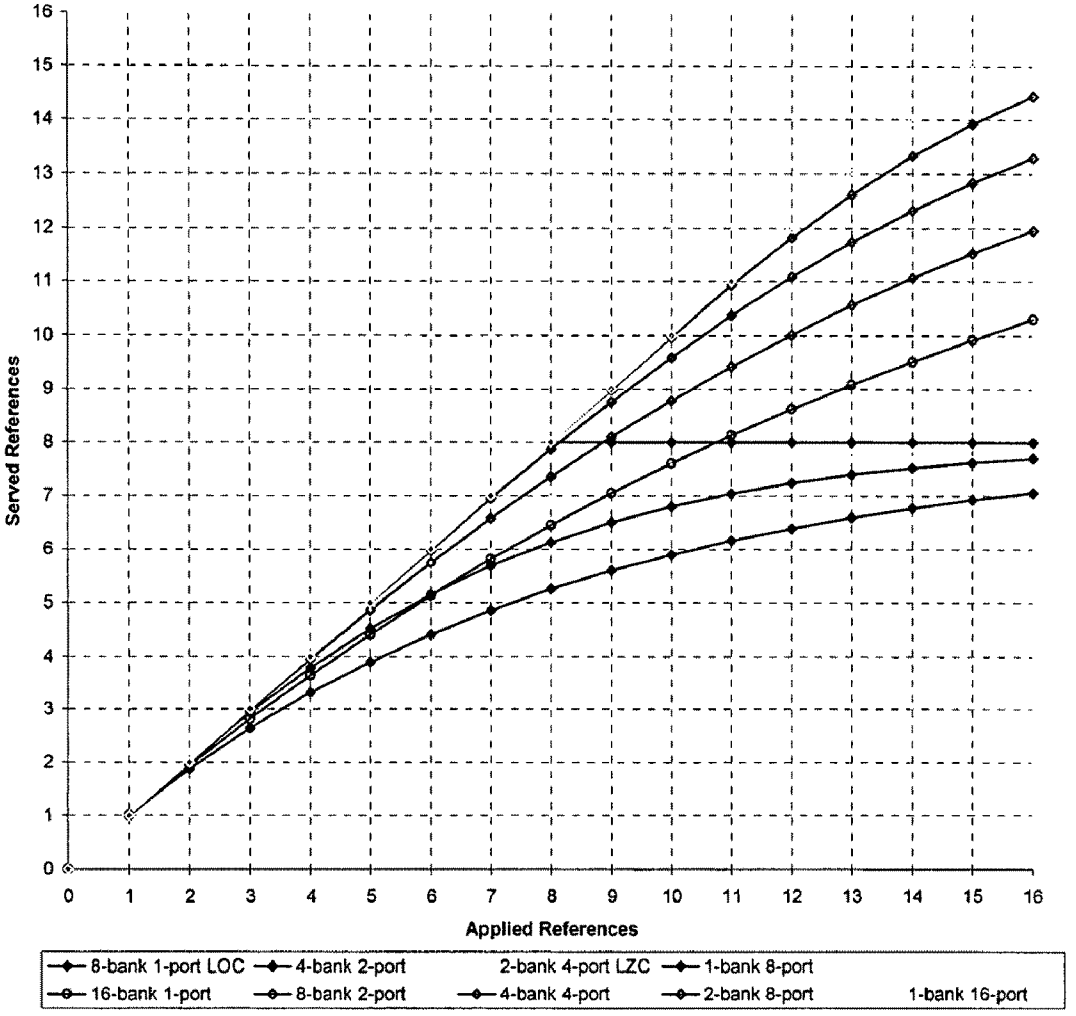
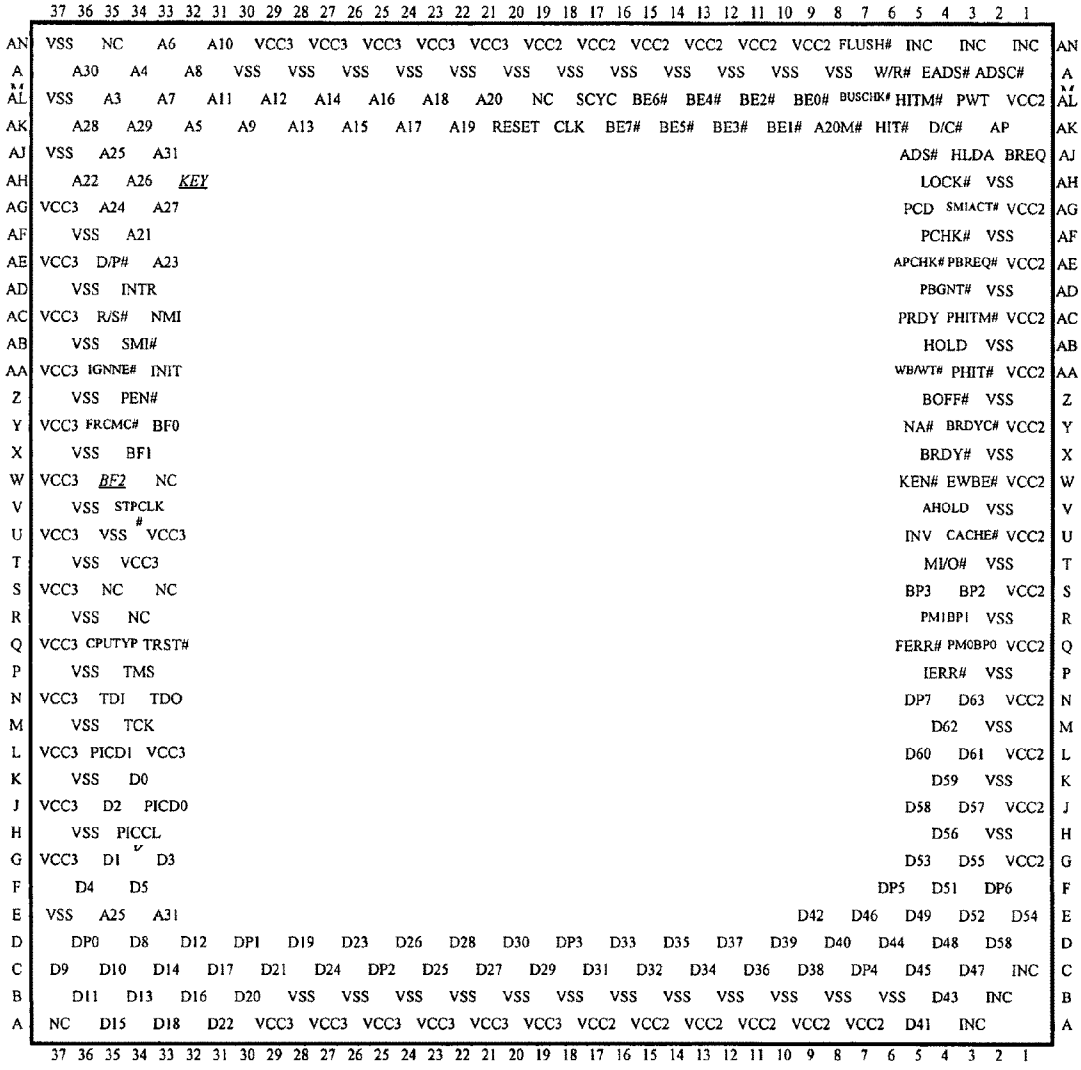

Served References
Applied References
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
8-bank 1-port LOC
4-bank 2-port
2-bank 4-port LZC
1-bank 8-port
16-bank 1-port
8-bank 2-port
4-bank 4-port
2-bank 8-port
1-bank 16-port

FIG. 56



	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	
AN	VSS	NC	A6	A10	VCC3	VCC3	VCC3	VCC3	VCC3	VCC2	VCC2	VCC2	VCC2	VCC2	VCC2	FLUSH#	INC	INC	INC	AN
A	A30	A4	A8	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	W/R#	EADS#	ADSC#	A
AL	VSS	A3	A7	A11	A12	A14	A16	A18	A20	NC	SCYC	BE6#	BE4#	BE2#	BE0#	BUSCHK#	HITM#	PWT	VCC2	AL
AK	A28	A29	A5	A9	A13	A15	A17	A19	RESET	CLK	BE7#	BE5#	BE3#	BE1#	A20M#	HIT#	D/C#	AP	AK
AJ	VSS	A25	A31	ADS#	HLDA	BREQ	AJ
AH	A22	A26	KEY	LOCK#	VSS	AH
AG	VCC3	A24	A27	PCD	SMIACT#	VCC2	AG
AF	VSS	A21	PCHK#	VSS	AF
AE	VCC3	D/P#	A23	APCHK#	PBREQ#	VCC2	AE
AD	VSS	INTR	PBGNT#	VSS	AD
AC	VCC3	R/S#	NMI	PRDY	PHITM#	VCC2	AC
AB	VSS	SMI#	HOLD	VSS	AB
AA	VCC3	IGNNE#	INIT	WB/WT#	PHIT#	VCC2	AA
Z	VSS	PEN#	BOFF#	VSS	Z
Y	VCC3	FRCMC#	BF0	NA#	BRDYC#	VCC2	Y
X	VSS	BF1	BRDY#	VSS	X
W	VCC3	BE2	NC	KEN#	EWBE#	VCC2	W
V	VSS	STPCLK	AHOLD	VSS	V
U	VCC3	VSS	VCC3	INV	CACHE#	VCC2	U
T	VSS	VCC3	MI/O#	VSS	T
S	VCC3	NC	NC	BP3	BP2	VCC2	S
R	VSS	NC	PM1BP1	VSS	R
Q	VCC3	CPUTYP	TRST#	FERR#	PM0BP0	VCC2	Q
P	VSS	TMS	IERR#	VSS	P
N	VCC3	TDI	TDO	DP7	D63	VCC2	N
M	VSS	TCK	D62	VSS	M
L	VCC3	PICD1	VCC3	D60	D61	VCC2	L
K	VSS	D0	D59	VSS	K
J	VCC3	D2	PICD0	D58	D57	VCC2	J
H	VSS	PICCL	D56	VSS	H
G	VCC3	D1	D3	D53	D55	VCC2	G
F	D4	D5	DP5	D51	DP6	F
E	VSS	A25	A31	D42	D46	D49	D52	D54	E
D	DP0	D8	D12	DP1	D19	D23	D26	D28	D30	DP3	D33	D35	D37	D39	D40	D44	D48	D58	D
C	D9	D10	D14	D17	D21	D24	DP2	D25	D27	D29	D31	D32	D34	D36	D38	DP4	D45	D47	INC	C
B	D11	D13	D16	D20	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	D43	INC	B
A	NC	D15	D18	D22	VCC3	VCC3	VCC3	VCC3	VCC3	VCC3	VCC2	VCC2	VCC2	VCC2	VCC2	VCC2	D41	INC	A
	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	

FIG. 57

Operation code

A.RES	Always reserved

FIG. 58A

Format

A.RES imm

ares(imm)

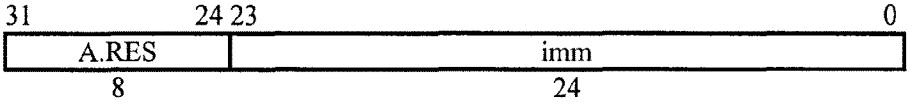


FIG. 58B

Definition

```
def AlwaysReserved as
    raise ReservedInstruction
enddef
```

Exceptions

Reserved Instruction

FIG. 58C

Operation codes

A.ADD	Address add
A.ADD.O	Address add signed check overflow
A.ADD.U.O	Address add unsigned check overflow
A.AND	Address and
A.ANDN	Address and not
A.NAND	Address not and
A.NOR	Address not or
A.OR	Address or
A.ORN	Address or not
A.XNOR	Address exclusive nor
A.XOR	Address xor

Redundancies

A.OR rd=rc,rc	⇔	*A.COPY rd=rc*
A.AND rd=rc,rc	⇔	*A.COPY rd=rc*
A.NAND rd=rc,rc	⇔	*A.NOT rd=rc*
A.NOR rd=rc,rc	⇔	*A.NOT rd=rc*
A.XNOR rd=rc,rc	⇔	*A.SET rd*
A.XOR rd=rc,rc	⇔	*A.ZERO rd*
A.ADD rd=rc,rc	⇔	A.SHL.I rd=rc,1
A.ADD.O rd=rc,rc	⇔	A.SHL.I.O rd=rc,1
A.ADD.U.O rd=rc,rc	⇔	A.SHL.I.U.O rd=rc,1

FIG. 59A

Selection

class	operation	check
arithmetic	ADD	NONE O U.O
bitwise	OR AND XOR ANDN NOR NAND XNOR ORN	

Format

op rd=rc,rb

rd=op(rc,rb)

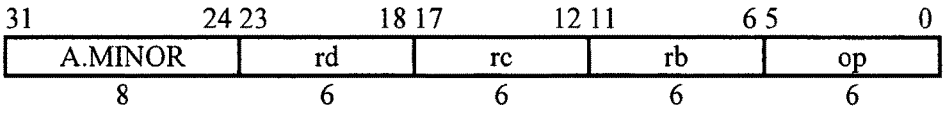


FIG. 59B

Definition

```
def Address(op,rd,rc,rb) as
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 64)
    case op of
        A.ADD:
            a ← c + b
        A.ADD.O:
            t ← (c63 || c) + (b63 || b)
            if t64 ≠ t63 then
                raise FixedPointArithmetic
            endif
            a ← t63..0
        A.ADD.UO:
            t ← (0^1 || c) + (0^1 || b)
            if t64 ≠ 0 then
                raise FixedPointArithmetic
            endif
            a ← t63..0
        A.AND:
            a ← c and b
        A.OR:
            a ← c or b
        A.XOR:
            a ← c xor b:
        A.ANDN:
            a ← c and not b
        A.NAND:
            a ← not (c and b)
        A.NOR:
            a ← not (c or b)
        A.XNOR:
            a ← not (c xor b)
        A.ORN:
            a ← c or not b
    endcase
    RegWrite(rd, 64, a)
enddef
```

Exceptions

Fixed-point arithmetic

FIG. 59C

Operation codes

A.COM.AND.E	Address compare and equal zero
A.COM.AND.NE	Address compare and not equal zero
A.COM.E	Address compare equal
A.COM.GE	Address compare greater equal signed
A.COM.GE.U	Address compare greater equal unsigned
A.COM.L	Address compare less signed
A.COM.L.U	Address compare less unsigned
A.COM.NE	Address compare not equal

Equivalencies

A.COM.E.Z	Address compare equal zero
A.COM.G.Z	Address compare greater zero signed
A.COM.GE.Z	Address compare greater equal zero signed
A.COM.L.Z	Address compare less zero signed
A.COM.LE.Z	Address compare less equal zero signed
A.COM.NE.Z	Address compare not equal zero
A.COM.G	Address compare greater signed
A.COM.G.U	Address compare greater unsigned
A.COM.LE	Address compare less equal signed
A.COM.LE.U	Address compare less equal unsigned
A.FIX	Address fixed point arithmetic exception
A.NOP	Address no operation

A.COM.E.Z rc	←	A.COM.AND.E rc,rc
A.COM.G.Z rc	⇐	A.COM.L.U rc,rc
A.COM.GE.Z rc	⇐	A.COM.GE rc,rc
A.COM.L.Z rc	⇐	A.COM.L rc,rc
A.COM.LE.Z rc	⇐	A.COM.GE.U rc,rc
A.COM.NE.Z rc	←	A.COM.AND.NE rc,rc
A.COM.G rc,rd	→	A.COM.L rd,rc
A.COM.G.U rc,rd	→	A.COM.L.U rd,rc
A.COM.LE rc,rd	→	A.COM.GE rd,rc
A.COM.LE.U rc,rd	→	A.COM.GE.U rd,rc
A.FIX	←	A.COM.E 0,0
A.NOP	←	A.COM.NE 0,0

Redundancies

A.COM.E rd,rd	⇔	*A.FIX*
A.COM.NE rd,rd	⇔	*A.NOP*

FIG. 60A

Selection

class	operation	cond	operand
boolean	COM.AND COM	E NE	
arithmetic	COM	L GE *G LE*	NONE U
	COM	*L GE G LE E NE*	*Z*

Format

A.COM.op rd,rc

acomop(rd,rc)
acomopz(rcd)

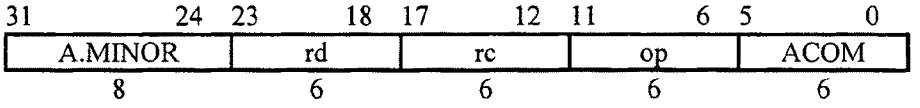


FIG. 60B

Definition

```
def AddressCompare(op,rd,rc) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    case op of
        A.COM.E:
            a ← d = c
        A.COM.NE:
            a ← d ≠ c
        A.COM.AND.E:
            a ← (d and c) = 0
        A.COM.AND.NE:
            a ← (d and c) ≠ 0
        A.COM.L:
            a ← (rd = rc) ? (c < 0) : (d < c)
        A.COM.GE:
            a ← (rd = rc) ? (c ≥ 0) : (d ≥ c)
        A.COM.L.U:
            a ← (rd = rc) ? (c > 0) : ((0 || d) < (0 || c))
        A.COM.GE.U:
            a ← (rd = rc) ? (c ≤ 0) : ((0 || d) ≥ (0 || c))
    endcase
    if a then
        raise FixedPointArithmetic
    endif
enddef
```

Exceptions

Fixed-point arithmetic

FIG. 60C

Operation codes

A.COPY.I	Address copy immediate

Equivalencies

A.SET	Address set
A.ZERO	Address zero

A.SET rd	←	A.COPY.I rd=-1
A.ZERO rd	←	A.COPY.I rd=0

FIG. 61A

Format

A.COPY.I rd=imm

rd=acopyi(imm)

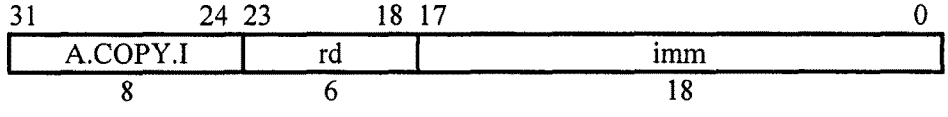


FIG. 61B

Definition

```
def AddressCopyImmediate(op,rd,imm) as
    a ← (imm⟨7⟩^10 || imm)
    RegWrite(rd, 128, a)
enddef
```

Exceptions

none

FIG. 61C

Operation codes

A.ADD.I	Address add immediate
A.ADD.I.O	Address add immediate signed check overflow
A.ADD.I.U.O	Address add immediate unsigned check overflow
A.AND.I	Address and immediate
A.NAND.I	Address not and immediate
A.NOR.I	Address not or immediate
A.OR.I	Address or immediate
A.XOR.I	Address xor immediate

Equivalencies

A.ANDN.I	Address and not immediate
A.COPY	Address copy
A.NOT	Address not
A.ORN.I	Address or not immediate
A.XNOR.I	Address xnor immediate

A.ANDN.I rd=rc.imm	→	A.AND.I rd=rc,~imm
A.COPY rd=rc	←	A.OR.I rd=rc,0
A.NOT rd=rc	←	A.NOR.I rd=rc,0
A.ORN.I rd=rc.imm	→	A.OR.I rd=rc,~imm
A.XNOR.I rd=rc.imm	→	A.XOR.I rd=rc,~imm

Redundancies

A.ADD.I rd=rc,0	⇔	*A.COPY rd=rc*
A.ADD.I.O rd=rc,0	⇔	*A.COPY rd=rc*
A.ADD.I.U.O rd=rc,0	⇔	*A.COPY rd=rc*
A.AND.I rd=rc,0	⇔	*A.ZERO rd*
A.AND.I rd=rc,-1	⇔	*A.COPY rd=rc*
A.NAND.I rd=rc,0	⇔	*A.SET rd*
A.NAND.I rd=rc,-1	⇔	*A.NOT rd=rc*
A.OR.I rd=rc,-1	⇔	*A.SET rd*
A.NOR.I rd=rc,-1	⇔	*A.ZERO rd*
A.XOR.I rd=rc,0	⇔	*A.COPY rd=rc*
A.XOR.I rd=rc,-1	⇔	*A.NOT rd=rc*

FIG. 62A

Selection

class	operation	check
arithmetic	ADD	NONE O UO
bitwise	AND OR NANDNOR XOR	

Format

op rd=rc,imm

rd=op(rc,imm)

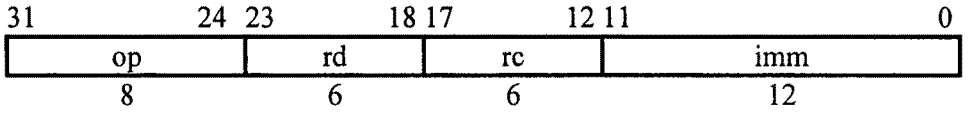


FIG. 62B

Definition

```
def AddressImmediate(op,rd,rc,imm) as
    i ← imm[52] || imm
    c ← RegRead(rc, 64)
    case op of
        A.AND.I:
            a ← c and i
        A.OR.I:
            a ← c or i
        A.NAND.I:
            a ← c nand i
        A.NOR.I:
            a ← c nor i
        A.XOR.I:
            a ← c xor i:
        A.ADD.I:
            a ← c + i
        A.ADD.I.O:
            t ← (c63 || c) + (i63 || i)
            if t64 ≠ t63 then
                raise FixedPointArithmetic
            endif
            a ← t63..0
        A.ADD.I.U.O:
            t ← (c63 || c) + (i63 || i)
            if t64 ≠ 0 then
                raise FixedPointArithmetic
            endif
            a ← t63..0
    endcase
    RegWrite(rd, 64, a)
enddef
```

Exceptions

Fixed-point arithmetic

FIG. 62C

Operation codes

A.SET.AND.E.I	Address set and equal immediate
A.SET.AND.NE.I	Address set and not equal immediate
A.SET.E.I	Address set equal immediate
A.SET.GE.I	Address set greater equal immediate signed
A.SET.L.I	Address set less immediate signed
A.SET.NE.I	Address set not equal immediate
A.SET.GE.I.U	Address set greater equal immediate unsigned
A.SET.L.I.U	Address set less immediate unsigned
A.SUB.I	Address subtract immediate
A.SUB.I.O	Address subtract immediate signed check overflow
A.SUB.I.U.O	Address subtract immediate unsigned check overflow

Equivalencies

A.NEG	Address negate
A.NEG.O	Address negate signed check overflow
A.SET.G.I.U	Address set greater immediate unsigned
A.SET.LE.I	Address set less equal immediate signed
A.SET.LE.I.U	Address set less equal immediate unsigned

A.NEG rd=rc	→	A.SUB.I rd=0,rc
A.NEG.O rd=rc	→	A.SUB.I.O rd=0,rc
A.SET.G.I rd=imm,rc	→	A.SET.GE.I rd=imm+1,rc
A.SET.G.I.U rd=imm,rc	→	A.SET.GE.I.U rd=imm+1,rc
A.SET.LE.I rd=imm,rc	→	A.SET.L.I rd=imm-1,rc
A.SET.LE.I.U rd=imm,rc	→	A.SET.L.I.U rd=imm-1,rc

Redundancies

A.SET.AND.E.I rd=rc,0	⇔	*A.SET rd*
A.SET.AND.NE.I rd=rc,0	⇔	*A.ZERO rd*
A.SET.AND.E.I rd=rc,-1	⇔	*A.SET.E.Z rd=rc*
A.SET.AND.NE.I rd=rc,-1	⇔	*A.SET.NE.Z rd=rc*
A.SET.E.I rd=rc,0	⇔	*A.SET.E.Z rd=rc*
A.SET.GE.I rd=rc,0	⇔	*A.SET.GE.Z rd=rc*
A.SET.L.I rd=rc,0	⇔	*A.SET.L.Z rd=rc*
A.SET.NE.I rd=rc,0	⇔	*A.SET.NE.Z rd=rc*
A.SET.GE.I.U rd=rc,0	⇔	*A.SET.GE.U.Z rd=rc*
A.SET.L.I.U rd=rc,0	⇔	*A.SET.L.U.Z rd=rc*

FIG. 63A

Selection

class	operation	cond	form	type	check
arithmetic	SUB		I		
				NONE U	O
boolean	SET.AND SET	E NE	I		
	SET	L GE *G LE*	I	NONE U	

Format

op rd=imm,rc

rd=op(imm,rc)

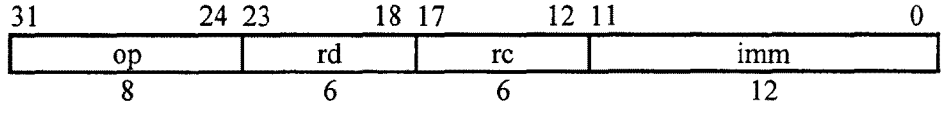


FIG. 63B

Definition

```
def AddressImmediate(op,rd,rc,imm) as
    i ← imm_11^52 || imm
    c ← RegRead(rc, 64)
    case op of
        A.SUB.I:
            a ← i - c
        A.SUB.I.O:
            t ← (i_63 || i) - (c_63 || c)
            if t_64 ≠ t_63 then
                raise FixedPointArithmetic
            endif
            a ← t_63..0
        A.SUB.I.U.O:
            t ← (i_63 || i) - (c_63 || c)
            if t_64 ≠ 0 then
                raise FixedPointArithmetic
            endif
            a ← t_63..0
        A.SET.AND.E.I:
            a ← ((i and c) = 0)^64
        A.SET.AND.NE.I:
            a ← ((i and c) ≠ 0)^64
        A.SET.E.I:
            a ← (i = c)^64
        A.SET.NE.I:
            a ← (i ≠ c)^64
        A.SET.L.I:
            a ← (i < c)^64
        A.SET.GE.I:
            a ← (i ≥ c)^64
        A.SET.L.I.U:
            a ← ((0 || i) < (0 || c))^64
        A.SET.GE.I.U:
            a ← ((0 || i) ≥ (0 || c))^64
    endcase
    RegWrite(rd, 64, a)
enddef
```

Exceptions

Fixed-point arithmetic

FIG. 63C

Operation codes

A.SET.AND.E	Address set and equal zero
A.SET.AND.NE	Address set and not equal zero
A.SET.E	Address set equal
A.SET.GE	Address set greater equal signed
A.SET.GE.U	Address set greater equal unsigned
A.SET.L	Address set less signed
A.SET.L.U	Address set less unsigned
A.SET.NE	Address set not equal
A.SUB	Address subtract
A.SUB.O	Address subtract signed check overflow
A.SUB.U.O	Address subtract unsigned check overflow

Equivalencies

A.SET.E.Z	Address set equal zero
A.SET.G.Z	Address set greater zero signed
A.SET.GE.Z	Address set greater equal zero signed
A.SET.L.Z	Address set less zero signed
A.SET.LE.Z	Address set less equal zero signed
A.SET.NE.Z	Address set not equal zero
A.SET.G	Address set greater signed
A.SET.G.U	Address set greater unsigned
A.SET.LE	Address set less equal signed
A.SET.LE.U	Address set less equal unsigned

A.SET.E.Z rd=rc	←	A.SET.AND.E rd=rc,rc
A.SET.G.Z rd=rc	⇐	A.SET.L.U rd=rc,rc
A.SET.GE.Z rd=rc	⇐	A.SET.GE rd=rc,rc
A.SET.L.Z rd=rc	⇐	A.SET.L rd=rc,rc
A.SET.LE.Z rd=rc	⇐	A.SET.GE.U rd=rc,rc
A.SET.NE.Z rd=rc	←	A.SET.AND.NE rd=rc,rc
A.SET.G rd=rb,rc	→	A.SET.L rd=rc,rb
A.SET.G.U rd=rb,rc	→	A.SET.L.U rd=rc,rb
A.SET.LE rd=rb,rc	→	A.SET.GE rd=rc,rb
A.SET.LE.U rd=rb,rc	→	A.SET.GE.U rd=rc,rb

Redundancies

A.SET.E rd=rc,rc	⇔	*A.SET rd*
A.SET.NE rd=rc,rc	⇔	*A.ZERO rd*

FIG. 64A

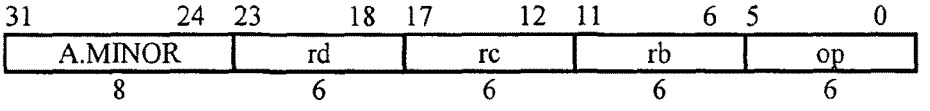
Selection

class	operation	cond	operand	check
arithmetic	SUB			
			NONE U	O
boolean	SET.AND SET	E NE		
	SET	L GE *G LE*	NONE U	
	SET	*L GE G LE E NE*	Z	

Format

op rd=rb,rc

rd=op(rb,rc)
rd=opz(rcb)



rc ← rb ← rcb

FIG. 64B

Definition

```
def AddressReversed(op,rd,rc,rb) as
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case op of
        A.SET.E:
            a ← (b = c)^64
        A.SET.NE:
            a ← (b ≠ c)^64
        A.SET.AND.E:
            a ← ((b and c) = 0)^64
        A.SET.AND.NE:
            a ← ((b and c) ≠ 0)^64
        A.SET.L:
            a ← ((rc = rb) ? (b < 0) : (b < c))^64
        A.SET.GE:
            a ← ((rc = rb) ? (b ≥ 0) : (b ≥ c))^64
        A.SET.L.U:
            a ← ((rc = rb) ? (b > 0) : ((0 || b) < (0 || c))^64
        A.SET.GE.U:
            a ← ((rc = rb) ? (b ≤ 0) : ((0 || b) ≥ (0 || c))^64
        A.SUB:
            a ← b - c
        A.SUB.O:
            t ← (b63 || b) - (c63 || c)
            if t64 ≠ t63 then
                raise FixedPointArithmetic
            endif
            a ← t63..0
        A.SUB.U.O:
            t ← (0^1 || b) - (0^1 || c)
            if t64 ≠ 0 then
                raise FixedPointArithmetic
            endif
            a ← t63..0
    endcase
    RegWrite(rd, 64, a)
enddef
```

Exceptions

Fixed-point arithmetic

FIG. 64C

Operation codes

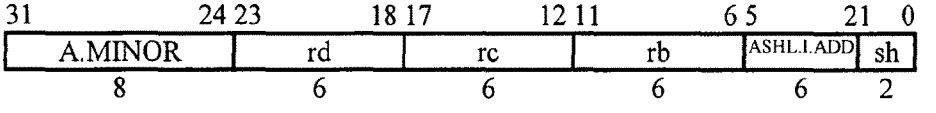
A.SHL.I.ADD	Address shift left immediate add

FIG. 65A

Format

A.SHL.I.ADD rd=rc,rb,i

rc=op(ra,rb,i)



assert $1 \leq i \leq 4$
sh $\leftarrow$ i-1

FIG. 65B

Definition

```
def AddressShiftLeftImmediateAdd(sh,rd,rc,rb) as
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 64)
    a ← c + (b62-sh..0 || 01+sh)
    RegWrite(rd, 64, a)
enddef
```

Exceptions

none

FIG. 65C

Operation codes

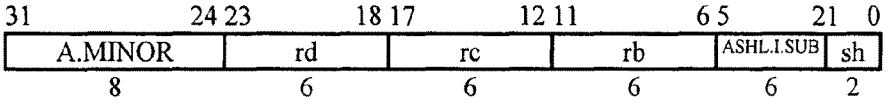
A.SHL.I.SUB	Address shift left immediate subtract

FIG. 66A

Format

ASHL.I.SUB rd=rb,i,rc

rd=op(rb,i,rc)



assert $1 \leq i \leq 4$
sh $\leftarrow$ i-1

FIG. 66B

Definition

```
def AddressShiftLeftImmediateSubtract(op,rd,rc,rb) as
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    a ← (b62-sh..0 || 01+sh) - c
    RegWrite(rd, 64, a)
enddef
```

Exceptions

none

FIG. 66C

Operation codes

A.SHL.I	Address shift left immediate
A.SHL.I.O	Address shift left imMediate signed check overflow
A.SHL.I.U.O	Address shift left immediate unsigned check overflow
A.SHR.I	Address signed shift right immediate
A.SHR.I.U	Address shift right immediate unsigned

Rèdundancies

A.SHL.I rd=rc,1	⇔	A.ADD rd=rc,rc
A.SHL.I.O rd=rc,1	⇔	A.ADD.O rd=rc,rc
A.SHL.I.U.O rd=rc,1	⇔	A.ADD.U.O rd=rc,rc
A.SHL.I rd=rc,0	⇔	*A.COPY rd=rc*
A.SHL.I.O rd=rc,0	⇔	*A.COPY rd=rc*
A.SHL.I.U.O rd=rc,0	⇔	*A.COPY rd=rc*
A.SHR.I rd=rc,0	⇔	*A.COPY rd=rc*
A.SHR.I.U rd=rc,0	⇔	*A.COPY rd=rc*

FIG. 67A

Selection

class	operation	form	operand	check
shift	SHL	I		
			NONE U	O
	SHR	I	NONE U	

Format

op rd=rc,simm

rd=op(rc,simm)

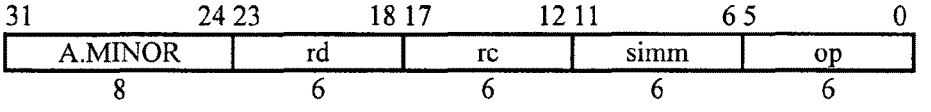


FIG. 67B

Definition

```
def AddressShiftImmediate(op,rd,rc,simm) as
    c ← RegRead(rc, 64)
    case op of
        A.SHL.I:
            a ← c63-simm..0 || 0simm
        A.SHL.I.O:
            if c63..63-simm ≠ c63simm+1 then
                raise FixedPointArithmetic
            endif
            a ← c63-simm..0 || 0simm
        A.SHL.I.U.O:
            if c63..64-simm ≠ 0 then
                raise FixedPointArithmetic
            endif
            a ← c63-simm..0 || 0simm
        A.SHR.I:
            a ← a63simm || c63..simm
        A.SHR.I.U:
            a ← 0simm || c63..simm
    endcase
    RegWrite(rd, 64, a)
enddef
```

Exceptions

Fixed-point arithmetic

FIG. 67C

Operation codes

A.MUX	Address multiplex

FIG. 68A

Format

op ra=rd,rc,rb

ra=amux(rd,rc,rb)

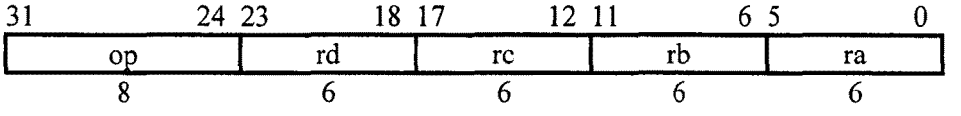


FIG. 68B

Definition

```
def AddressTernary(op,rd,rc,rb,ra) as
    d ← RegRead(rd, 64)
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 64)
    endcase
    case op of
        A.MUX:
            a ← (c and d) or (b and not d)
    endcase
    RegWrite(ra, 64, a)
enddef
```

Exceptions

none

FIG. 68C

Operation codes

B	Branch

FIG. 69A

Format

B rd

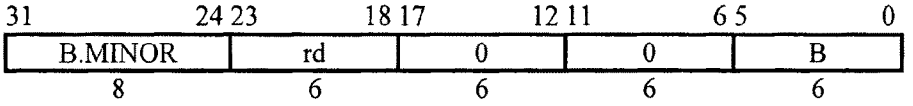


FIG. 69B

Definition

```
def Branch(rd,rc,rb) as
    if (rc ≠ 0) or (rb ≠ 0) then
        raise ReservedInstruction
    endif
    d ← RegRead(rd, 64)
    if (d1..0) ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    endif
    ProgramCounter ← d63..2 || 0^2
    raise TakenBranch
enddef
```

Exceptions

Reserved Instruction
Access disallowed by virtual address

FIG. 69C

Operation codes

B.BACK	Branch back

FIG. 70A

Format

B.BACK

bback()

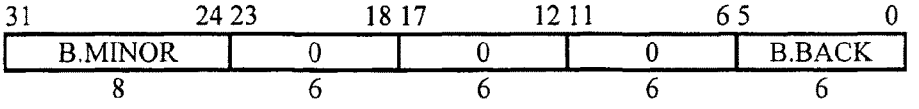


FIG. 70B

Definition

```
def BranchBack(rd,rc,rb) as
    c ← RegRead(rc, 128)
    if (rd ≠ 0) or(rc ≠ 0) or (rb ≠ 0) then
        raise ReservedInstruction
    endif
    a ← LoadMemory(ExceptionBase,ExceptionBase+Thread*128,128,L)
    if PrivilegeLevel > c1..0 then
        PrivilegeLevel ← c1..0
    endif
    ProgramCounter ← c63..2 || 0^2
    ExceptionState ← 0
    RegWrite(rd,128,a)
    raise TakenBranchContinue
enddef
```

Exceptions

Reserved Instruction
Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 70C

Operation codes

B.BARRIER	Branch barrier

FIG. 71A

Format

B.BARRIER rd

bbarrier(rd)

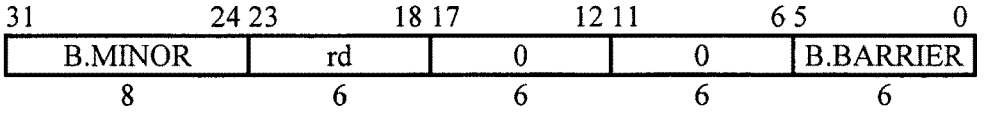


FIG. 71B

Definition

```
def BranchBarrier(rd,rc,rb) as
    if (rc ≠ 0) or (rb ≠ 0) then
        raise ReservedInstruction
    endif
    d ← RegRead(rd, 64)
    if (d1..0) ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    endif
    ProgramCounter ← d63..2 || 0^2
    FetchBarrier()
    raise TakenBranch
enddef
```

Exceptions

Reserved Instruction

FIG. 71C

Operation codes

B.AND.E	Branch and equal zero
B.AND.NE	Branch and not equal zero
B.E	Branch equal
B.GE	Branch greater equal signed
B.L	Branch signed less
B.NE	Branch not equal
B.GE.U	Branch greater equal unsigned
B.L.U	Branch less unsigned

Equivalencies

B.E.Z	Branch equal zero
B.G.Z[1]	Branch greater zero signed
B.GE.Z[2]	Branch greater equal zero signed
B.L.Z[3]	Branch less zero signed
B.LE.Z[4]	Branch less equal zero signed
B.NE.Z	Branch not equal zero
B.LE	Branch less equal signed
B.G	Branch greater signed
B.LE.U	Branch less equal unsigned
B.G.U	Branch greater unsigned
B.NOP	Branch no operation

B.E.Z rc,target	←	B.AND.E rc,rc,target
B.G.Z rc,target	⇐	B.L.U rc,rc,target
B.GE.Z rc,target	⇐	B.GE rc,rc,target
B.L.Z rc,target	⇐	B.L rc,rc,target
B.LE.Z rc,target	⇐	B.GE.U rc,rc,target
B.NE.Z rc,target	←	B.AND.NE rc,rc,target
B.LE rc,rd,target	→	B.GE rd,rc,target
B.G rc,rd,target	→	B.L rd,rc,target
B.LE.U rc,rd,target	→	B.GE.U rd,rc,target
B.G.U rc,rd,target	→	B.L.U rd,rc,target
B.NOP	←	B.NE r0,r0,$

Redundancies

B.E rc,rc,target	⇔	B.I target
B.NE rc,rc,target	⇔	*B.NOP*

FIG. 72A

Selection

class	op	compare	type
arithmetic		L GE *G* *LE*	NONE U
vs. zero		*L* *GE* *G* *LE* *E* *NE*	*Z*
bitwise	none AND	E NE	

Format

op rd,rc,target

if (op(rd,rc)) goto target;

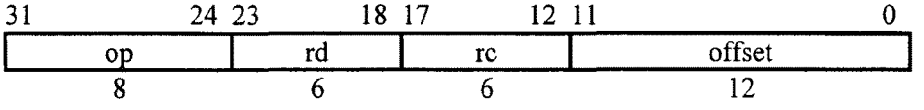


FIG. 72B

Definition

```
def BranchConditionally(op,rd,rc,offset) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    case op of
        B.E:
            a ← d = c
        B.NE:
            a ← d ≠ c
        B.AND.E:
            a ← (d and c) = 0
        BAND.NE:
            a ← (d and c) ≠ 0
        B.L:
            a ← (rd = rc) ? (c < 0): (d < c)
        B.GE:
            a ← (rd = rc) ? (c ≥ 0): (d ≥ c)
        B.L.U:
            a ← (rd = rc) ? (c > 0): ((0 || d) < (0 || c))
        B.GE.U:
            a ← (rd = rc) ? (c ≤ 0): ((0 || d) ≥ (0 || c))
    endcase
    if a then
        ProgramCounter ← ProgramCounter + (offset[59] || offset || 0^2)
        raise TakenBranch
    endif
enddef
```

Exceptions

none

FIG. 72C

Operation codes

B.E.F. 16	Branch equal floating-point half
B.E.F. 32	Branch equal floating-point single
B.E.F. 64	Branch equal floating-point double
B.E.F.128	Branch equal floating-point quad
B.GE.F. 16	Branch greater equal floating-point half
B.GE.F. 32	Branch greater equal floating-point single
B.GE.F. 64	Branch greater equal floating-point double
B.GE.F.128	Branch greater equal floating-point quad
B.L.F. 16	Branch less floating-point half
B.L.F. 32	Branch less floating-point single
B.L.F. 64	Branch less floating-point double
B.L.F.128	Branch less floating-point quad
B.LG.F. 16	Branch less greater floating-point half
B.LG.F. 32	Branch less greater floating-point single
B.LG.F. 64	Branch less greater floating-point double
B.LG.F.128	Branch less greater floating-point quad

Equivalencies

B.LE.F. 16	Branch less equal floating-point half
B.LE.F. 32	Branch less equal floating-point single
B.LE.F. 64	Branch less equal floating-point double
B.LE.F.128	Branch less equal floating-point quad
B.G.F. 16	Branch greater floating-point half
B.G.F. 32	Branch greater floating-point single
B.G.F. 64	Branch greater floating-point double
B.G.F.128	Branch greater floating-point quad

B.LE.F.size rc,rd,target	→	B.GE.F.size rd,rc,target
B.G.F.size rc,rd,target	→	B.L.F.size rd,rc,target

FIG. 73A

Selection

number format	type	compare					size	
floating-point	F	E	LG	L	GE	*G*	16	32
			LE					64
								128

Format

op rd,rc,target

if (op(rd,rc)) goto target;

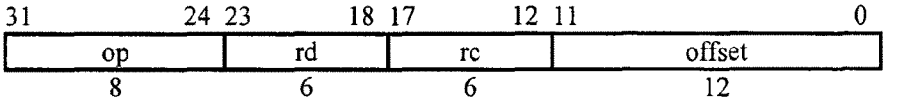


FIG. 73B

Definition

```
def BranchConditional(FloatingPointop,rd,rc,offset) as
    case op of
        B.E.F.16, B.LG.F.16, B.L.F.16, B.GE.F.16:
            size ← 16
        B.E.F.32, B.LG.F.32, B.L.F.32, B.GE.F.32:
            size ← 32
        B.E.F.64, B.LG.F.64, B.L.F.64, B.GE.F.64:
            size ← 64
        B.E.F.128, B.LG.F.128, B.L.F.128, B.GE.F.128:
            size ← 128
    endcase
    d ← F(size,RegRead(rd, 128))
    c ← F(size,RegRead(rc, 128))
    v ← fcom(d, c)
    case op of
        BEF16, BEF32, BEF64, BEF128:
            a ← (v = E)
        BLGF16, BLGF32, BLGF64, BLGF128:
            a ← (v = L) or (v = G)
        BLF16, BLF32, BLF64, BLF128:
            a ← (v = L)
        BGEF16, BGEF32, BGEF64, BGEF128:
            a ← (v = G) or (v = E)
    endcase
    if a then
        ProgramCounter ← ProgramCounter + (offset[50] || offset || 0^2)
        raise TakenBranch
    endif
enddef
```

Exceptions

none

FIG. 73C

Operation codes

B.I.F. 32	Branch invisible floating-point single
B.NI.F. 32	Branch not invisible floating-point single
B.NV.F. 32	Branch not visible floating-point single
B.V.F. 32	Branch visible floating-point single

FIG. 74A

Selection

number format	type	compare	size
floating-point	F	I NI NV V	32

Format

op rc,rd,target

if (op(rc,rd)) goto target;

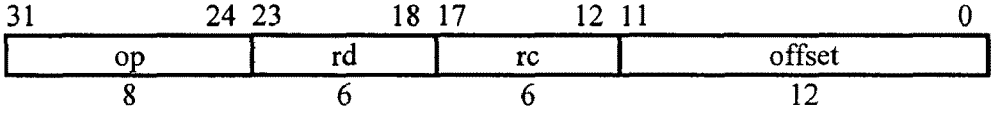


FIG. 74B

Definition

```
def n(a) as (a.t=QNAN) or (a.t=SNAN) enddef

def less(a,b) as fcom(a,b)=L enddef

def trxya,b,c,d) as (fcom(fabs(a),b)=G) and (fcom(fabs(c),d)=G) and (a.s=c.s) enddef


def BranchConditionalVisibilityFloatingPoint(op,rd,rc,offset) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    dx ← F(32,d31..0)
    cx ← F(32,c31..0)
    dy ← F(32,d63..32)
    cy ← F(32,c63..32)
    dz ← F(32,d95..64)
    cz ← F(32,c95..64)
    dw ← F(32,d127..96)
    cw ← F(32,c127..96)
    fl ← F(32,0x7f000000) // floating-point 1.0
    if (n(dx) or n(dy) or n(dz) or n(dw) or n(cx) or n(cy) or n(cz) or n(cw)) then
        a ← false
    else
        dv ← less(fabs(dx),dz) and less(fabs(dy),dz) and less(dz,fl) and (dz.s=0)
        cv ← less(fabs(cx),cz) and less(fabs(cy),cz) and less(cz,fl) and (cz.s=0)
        trz ← (less(fl,dz) and less(fl,cz)) or ((dz.s=1 and cz.s=1))
        tr ← trxy(dx,dz,cx,cz) or trxy(dy,dz,cy,cz) or trz
        case op of
            B.I.F.32:
                a ← tr
            B.NI.F.32:
                a ← not tr
            B.NV.F.32:
                a ← not (dv and cv)
            B.V.F.32:
                a ← dv and cv
        endcase
    endif
    if a then
        ProgramCounter ← ProgramCounter + (offset[50] || offset || 0[2])
```

FIG. 74C

```
            raise TakenBranch
        endif
    enddef
```

Exceptions

none

FIG. 74C *continued*

Operation codes

B.DOWN	Branch down

FIG. 75A

Format

B.DOWN rd

bdown(rd)

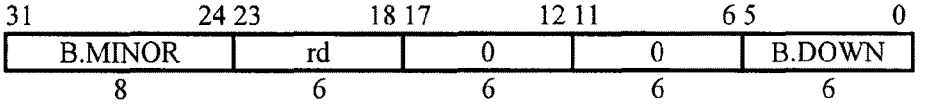


FIG. 75B

Definition

```
def BranchDown(rd,rc,rb) as
    if (rc ≠ 0) or (rb ≠ 0) then
        raise ReservedInstruction
    endif
    d ← RegRead(rd, 64)
    if PrivilegeLevel > d1..0 then
        PrivilegeLevel ← d1..0
    endif
    ProgramCounter ← d63..2 || 0^2
    raise TakenBranch
enddef
```

Exceptions

Reserved Instruction

FIG. 75C

Operation codes

B.GATE	Branch gateway

Equivalencies

B.GATE	←	B.GATE 0

FIG. 76A

Format

B.GATE rb

bgate(rb)

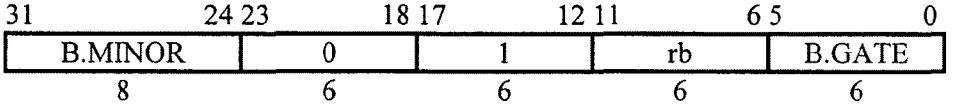


FIG. 76B

Definition

```
def BranchGateway(rd,rc,rb) as
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 64)
    if (rd ≠ 0) or (rc ≠ 1) then
        raise ReservedInstruction
    endif
    if c2..0 ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    endif
    d ← ProgramCounter63..2+1 || PrivilegeLevel
    if PrivilegeLevel < b1..0 then
        m ← LoadMemoryG(c,c,64,L)
        if b ≠ m then
            raise GatewayDisallowed
        endif
        PrivilegeLevel ← b1..0
    endif
    ProgramCounter ← b63..2 || 02
    RegWrite(rd, 64, d)
    raise TakenBranch
enddef
```

Exceptions

Reserved Instruction
Gateway disallowed
Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 76C

Operation codes

B.HALT	Branch halt

FIG. 77A

Format

B.HALT

bhalt()

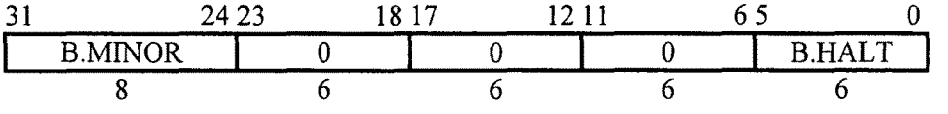


FIG. 77B

Definition

```
def BranchHalt(rd,rc,rb) as
    if (rd ≠ 0) or (rc ≠ 0) or (rb ≠ 0) then
        raise ReservedInstruction
    endif
    FetchHalt()
enddef
```

Exceptions

Reserved Instruction

FIG. 77C

Operation codes

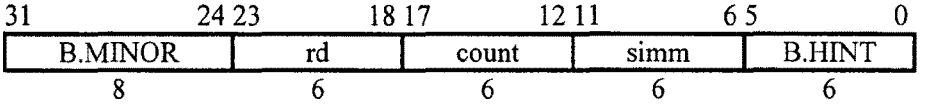
B.HINT	Branch Hint

FIG. 78A

Format

B.HINT badd,count,rd

bhint(badd,count,rd)



simm ← badd-pc-4

FIG. 78B

Definition

```
def BranchHint(rd,count,simm) as
    d ← RegRead(rd, 64)
    if (d1..0) ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    endif
    FetchHint(ProgramCounter +4 + (0 || simm || 0^2), d63..2 || 0^2, count)
enddef
```

Exceptions

Access disallowed by virtual address

FIG. 78C

Operation codes

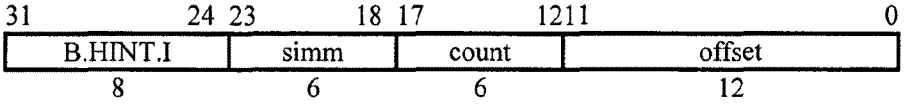
B.HINT.I	Branch Hint Immediate

FIG. 79A

Format

B.HINT.I badd,count,target

bhinti(badd,count,target)



simm ← badd-pc-4

FIG. 79B

Definition

```
def BranchHintImmediate(simm,count,offset) as
    BranchHint(ProgramCounter + 4 + (0 || simm || 0^2), count,
            ProgramCounter + (offset^44 || offset || 0^2))
enddef
```

Exceptions

none

FIG. 79C

Operation codes

B.I	Branch immediate

Redundancies

B.I target	⇔	B.E rc,rc,target

FIG. 80A

Format

B.I target

bi(target)

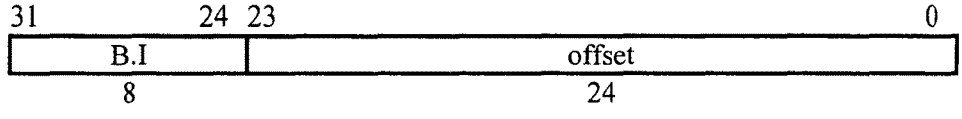


FIG. 80B

Definition

```
def BranchImmediate(offset) as
    ProgramCounter ← ProgramCounter + (offset[38] || offset || 0^2)
    raise TakenBranch
enddef
```

Exceptions

none

FIG. 80C

Operation codes

B.LINK.I	Branch immediate link

FIG. 81A

Format

B.LINK.I target

blinki(target)

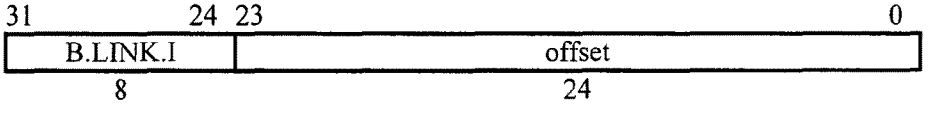


FIG. 81B

Definition

```
def BranchImmediateLink(offset) as
    RegWrite(0, 64, ProgramCounter + 4)
    ProgramCounter ← ProgramCounter + (offset[38:23] || offset || 0^2)
    raise TakenBranch
enddef
```

Exceptions

none

FIG. 81C

Operation codes

B.LINK	Branch link

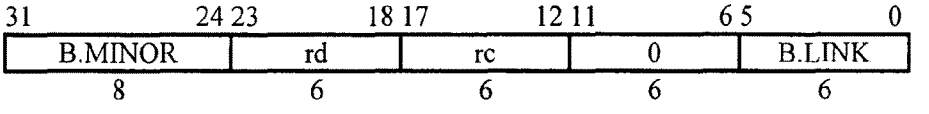
Equivalencies

B.LINK	←	B.LINK 0=0
B.LINK rc	←	B.LINK 0=rc

FIG. 82A

Format

B.LINK rd=rc



rb ← 0

FIG. 82B

Definition

```
def BranchLink(rd,rc,rb) as
    if rb ≠ 0 then
        raise ReservedInstruction
    endif
    c ← RegRead(rc, 64)
    if (c and 3) ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    endif
    RegWrite(rd, 64, ProgramCounter + 4)
    ProgramCounter ← c63..2 || 0^2
    raise TakenBranch
enddef
```

Exceptions

Reserved Instruction
Access disallowed by virtual address

FIG. 82C

Operation codes

S.D.C.S.64.A.B	Store double compare swap octlet aligned big-endian
S.D.C.S.64.A.L	Store double compare swap octlet aligned little-endian

FIG. 83A

Format

op rd@rc,rb

rd=op(rd,rc,rb)

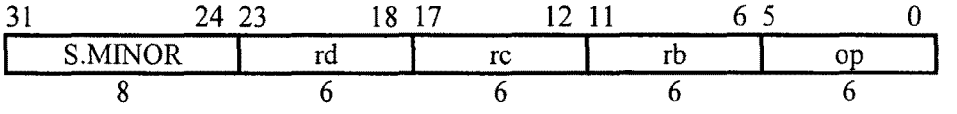


FIG. 83B

Definition

```
def StoreDoubleCompareSwap(op,rd,rc,rb) as
    size ← 64
    lsize ← log(size)
    case op of
        SDCS64AL:
            order ← L
        SDCS64AB:
            order ← B
    endcase
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    d ← RegRead(rd, 128)
    if (c2..0 ≠ 0) or (b2..0 ≠ 0) then
        raise AccessDisallowedByVirtualAddress
    endif
    lock
        a ← LoadMemoryW(c63..0,c63..0,64,order) || LoadMemoryW(b63..0,b63..0,64,order)
        if ((c127..64 || b127..64.) = a) then
            StoreMemory((c63..0,c63..0,64,order,d127..64)
            StoreMemory(b63..0,b63..0,64,order,d63..0)
        endif
    endlock
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 83C

Operation codes

S.A.S.I.64.A.B	Store add swap immediate octlet aligned big-endian
S.A.S.I.64.A.L	Store add swap immediate octlet aligned little-endian
S.C.S.I.64.A.B	Store compare swap immediate octlet aligned big-endian
S.C.S.I.64.A.L	Store compare swap immediate octlet aligned little-endian
S.M.S.I.64.A.B	Store multiplex swap immediate octlet aligned big-endian
S.M.S.I.64.A.L	Store multiplex swap immediate octlet aligned little-endian

FIG. 84A

Selection

number format	op	size	alignment	ordering
add-swap	AS	64	A	L B
compare-swap	CS	64	A	L B
multiplex-swap	MS	64	A	L B

Format

S.op.I.64.align.order rd@rc,offset

rd=sopi64alignorder(rd,rc,offset)

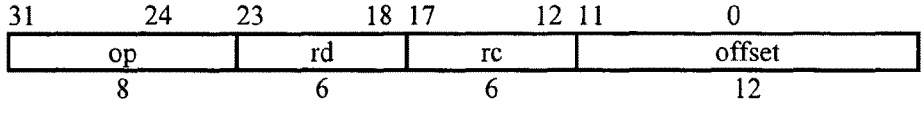


FIG. 84B

Definition

```
def StoreImmediateInplace(op,rd,rc,offset) as
    sizE ← 64
    lsize ← log(size)
    case op of
        SASI64AL, SCSI64AL, SMSI64AL:
            order ← L
        SASI64AB, SCSI64AB, SMSI64AB:
            order ← B
    endcase
    c ← RegRead(rc, 64)
    VirtAddr ← c + (offset[63]^(-lsize) || offset || 0^(lsize-3))
    if (c[lsize-4..0] ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    endif
    d ← RegRead(rd, 128)
    case op of
        SASI64AB, SASI64AL:
            lock
                a ← LoadMemoryW(c,VirtAddr,size,order)
                StoreMemory(c,VirtAddr,size,order,d63..0+a)
            endlock
        SCSI64AB, SCSI64AL:
            lock
                a ← LoadMemoryW(c,VirtAddr,size,order)
                if (a = d63..0) then
                    StoreMemory(c,VirtAddr,size,order,d127..64)
                endif
            endlock
        SMSI64AB, SMSI64AL:
            lock
                a ← LoadMemoryW(c,VirtAddr,size,order)
                m ← (d127..64 & d63..0) | (a & ~d63..0)
                StoreMemory(c,VirtAddr,size,order,m)
            endlock
    endcase
    RegWrite(rd, 64, a)
enddef
```

FIG. 84C

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 84C *continued*

Operation codes

S.A.S.64.A.B	Store add swap octlet aligned big-endian
S.A.S.64.A.L	Store add swap octlet aligned little-endian
S.C.S.64.A.B	Store compare swap octlet aligned big-endian
S.C.S.64.A.L	Store compare swap octlet aligned little-endian
S.M.S.64.A.B	Store multiplex swap octlet aligned big-endian
S.M.S.64.A.L	Store multiplex swap octlet aligned little-endian

FIG. 85A

Selection

number format	op	size	alignment	ordering
add-swap	A.S	64	A	L B
compare-swap	C.S	64	A	L B
multiplex-swap	M.S	64	A	L B

Format

op rd@rc,rb

rd=op(rd,rc,rb)

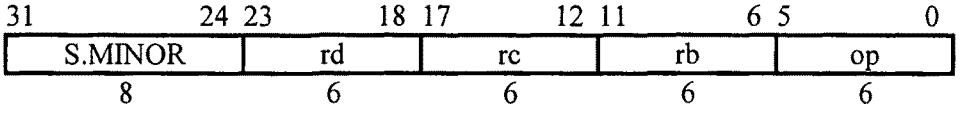


FIG. 85B

Definition

```
def StoreInplace(op,rd,rc,rb) as
    size ← 64
    lsize ← log(size)
    case op of
        SAS64AL, SCS64AL, SMS64AL:
            order ← L
        SAS64AB, SCS64AB, SMS64AB:
            order ← B
    endcase
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 64)
    VirtAddr ← c + (b66-lsize..0 || 0lsize-3)
    if (clsize-4..0 ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    endif
    d ← RegRead(rd, 128)
    case op of
        SAS64AB, SAS64AL:
            lock
                a ← LoadMemoryW(c,VirtAddr,size,order)
                StoreMemory(c,VirtAddr,size,order,d63..0+a)
            endlock
        SCS64AB, SCS64AL:
            lock
                a ← LoadMemoryW(c,VirtAddr,size,order)
                if (a = d63..0) then
                    StoreMemory(c,VirtAddr,size,order,d127..64)
                endif
            endlock
        SMS64AB, SMS64AL:
            lock
                a ← LoadMemoryW(c,VirtAddr,size,order)
                m ← (d127..64 & d63..0) | (a & ~d63..0)
                StoreMemory(c,VirtAddr,size,order,m)
            endlock
    endcase
    RegWrite(rd, 64, a)
enddef
```

FIG. 85C

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 85C *continued*

Operation codes

G.ADD.H. 8.C	Group add halve signed bytes ceiling
G.ADD.H. 8.F	Group add halve signed bytes floor
G.ADD.H. 8.N	Group add halve signed bytes nearest
G.ADD.H. 8.Z	Group add halve signed bytes zero
G.ADD.H. 16.C	Group add halve signed doublets ceiling
G.ADD.H. 16.F	Group add halve signed doublets floor
G.ADD.H. 16.N	Group add halve signed doublets nearest
G.ADD.H. 16.Z	Group add halve signed doublets zero
G.ADD.H. 32.C	Group add halve signed quadlets ceiling
G.ADD.H. 32.F	Group add halve signed quadlets floor
G.ADD.H. 32.N	Group add halve signed quadlets nearest
G.ADD.H. 32.Z	Group add halve signed quadlets zero
G.ADD.H. 64.C	Group add halve signed octlets ceiling
G.ADD.H. 64.F	Group add halve signed octlets floor
G.ADD.H. 64.N	Group add halve signed octlets nearest
G.ADD.H. 64.Z	Group add halve signed octlets zero
G.ADD.H.128.C	Group add halve signed hexlet ceiling
G.ADD.H.128.F	Group add halve signed hexlet floor
G.ADD.H.128.N	Group add halve signed hexlet nearest
G.ADD.H.128.Z	Group add halve signed hexlet zero
G.ADD.H.U. 8.C	Group add halve unsigned bytes ceiling
G.ADD.H.U. 8.F	Group add halve unsigned bytes floor
G.ADD.H.U. 8.N	Group add halve unsigned bytes nearest
G.ADD.H.U. 16.C	Group add halve unsigned doublets ceiling
G.ADD.H.U. 16.F	Group add halve unsigned doublets floor
G.ADD.H.U. 16.N	Group add halve unsigned doublets nearest
G.ADD.H.U. 32.C	Group add halve unsigned quadlets ceiling
G.ADD.H.U. 32.F	Group add halve unsigned quadlets floor
G.ADD.H.U. 32.N	Group add halve unsigned quadlets nearest
G.ADD.H.U. 64.C	Group add halve unsigned octlets ceiling
G.ADD.H.U. 64.F	Group add halve unsigned octlets floor
G.ADD.H.U. 64.N	Group add halve unsigned octlets nearest
G.ADD.H.U.128.C	Group add halve unsigned hexlet ceiling
G.ADD.H.U.128.F	Group add halve unsigned hexlet floor
G.ADD.H.U.128.N	Group add halve unsigned hexlet nearest

Redundancies

G.ADD.H.size.rnd rd=rc,rc	⇔	G.COPY rd=rc
G.ADD.H.U.size.rnd rd=rc,rc	⇔	G.COPY rd=rc

FIG. 86A

Format

G.op.size.rnd rd=rc,rb

rd=gopsizernd(rc,rb)

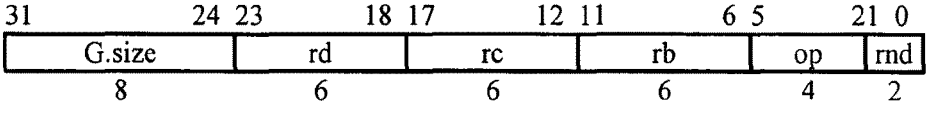


FIG. 86B

Definition

```
def GroupAddHalve(op,rnd,size,rd,rc,rb)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case op of
        G.ADDHC, G.ADDHF, G.ADDHN, G.ADDHZ:
            as ← cs ← bs ← 1
        G.ADDHUC, G.ADDHUF, G.ADDHUN, G.ADDHUZ
            as ← cs ← bs ← 0
            if rnd = Z then
                raise ReservedInstruction
            endif
    endcase
    h ← size+1
    r ← 1
    for i ← 0 to 128-size by size
        p ← ((cs and c_size-1) || c_size-1+i..i) + ((bs and b_size-1) || b_size-1+i..i)
        case rnd of
            none, N:
                s ← 0^size || ~p_1
            Z:
                s ← 0^size || p_size
            F:
                s ← 0^size+1
            C:
                s ← 0^size || 1^1
        endcase
        v ← ((as & p_size)||p) + (0||s)
        a_size-1+i..i ← v_size..1
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

ReservedInstruction

FIG. 86C

Operation codes

G.COPY.I.16	Group copy immediate doublet
G.COPY.I.32	Group signed copy immediate quadlet
G.COPY.I.64	Group signed copy immediate octlet
G.COPY.I.128	Group signed copy immediate hexlet

Equivalencies

G.COPY.I.8	Group copy immediate byte
G.SET	Group set
G.ZERO	Group zero

G.COPY.I.8 rd=(i$_7^9$ || i$_{7..0}$)	←	G.COPY.I.16 rd=(0 || i$_{7..0}$ || i$_{7..0}$)
G.SET rd	←	G.COPY.I.128 rd=-1
G.ZERO rd	←	G.COPY.I.128 rd=0

Redundancies

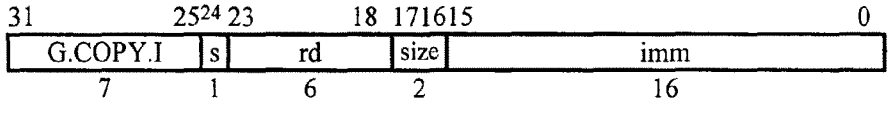
G.COPY.I.size rd=-1	⇔	*G.SET rd*
G.COPY.I.size rd=0	⇔	*G.ZERO rd*

FIG. 87A

Format

G.COPY.I.size rd=i

rd=gcopyisize(i)



$s \leftarrow i_{16}$
$imm \leftarrow i_{15..0}$

FIG. 87B

Definition

```
def GroupCopyImmediate(op,size,rd,imm) as
    s ← op0
    case size of
        16:
            If s then
                ReservedInstruction
            endif
            a ← imm || imm || imm || imm || imm || imm || imm || imm
        32:
            a ← s^16 || imm || s^16 || imm || s^16 || imm || s^16 || imm
        64:
            a ← s^48 || imm || s^48 || imm
        128:
            a ← s^112 || imm
    endcase
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Reserved Instruction

FIG. 87C

Operation codes

G.ADD.I. 16	Group add immediate doublet
G.ADD.I. 16.O	Group add immediate signed doublet check overflow
G.ADD.I. 32	Group add immediate quadlet
G.ADD.I. 32.O	Group add immediate signed quadlet check overflow
G.ADD.I. 64	Group add immediate octlet
G.ADD.I. 64.O	Group add immediate signed octlet check overflow
G.ADD.I.128	Group add immediate hexlet
G.ADD.I.128.O	Group add immediate signed hexlet check overflow
G.ADD.I.U. 16.O	Group add immediate unsigned doublet check overflow
G.ADD.I.U. 32.O	Group add immediate unsigned quadlet check overflow
G.ADD.I.U. 64.O	Group add immediate unsigned octlet check overflow
G.ADD.I.U.128.O	Group add immediate unsigned hexlet check overflow
G.AND.I. 16	Group and immediate doublet
G.AND.I. 32	Group and immediate quadlet
G.AND.I. 64	Group and immediate octlet
G.AND.I.128	Group and immediate hexlet
G.NAND.I. 16	Group not and immediate doublet
G.NAND.I. 32	Group not and immediate quadlet
G.NAND.I. 64	Group not and immediate octlet
G.NAND.I.128	Group not and immediate hexlet
G.NOR.I. 16	Group not or immediate doublet
G.NOR.I. 32	Group not or immediate quadlet
G.NOR.I. 64	Group not or immediate octlet
G.NOR.I.128	Group not or immediate hexlet
G.OR.I. 16	Group or immediate doublet
G.OR.I. 32	Group or immediate quadlet
G.OR.I. 64	Group or immediate octlet
G.OR.I.128	Group or immediate hexlet
G.XOR.I. 16	Group exclusive-or immediate doublet
G.XOR.I. 32	Group exclusive-or immediate quadlet
G.XOR.I. 64	Group exclusive-or immediate octlet
G.XOR.I.128	Group exclusive-or immediate hexlet

FIG. 88A

Equivalencies

G.ANDN.I. 16	Group and not immediate doublet
G.ANDN.I. 32	Group and not immediate quadlet
G.ANDN.I. 64	Group and not immediate octlet
G.ANDN.I.128	Group and not immediate hexlet
G.COPY	Group copy
G.NOT	Group not
G.ORN.I. 16	Group or not immediate doublet
G.ORN.I. 32	Group or not immediate quadlet
G.ORN.I. 64	Group or not immediate octlet
G.ORN.I.128	Group or not immediate hexlet
G.XNOR.I. 16	Group exclusive-nor immediate doublet
G.XNOR.I. 32	Group exclusive-nor immediate quadlet
G.XNOR.I. 64	Group exclusive-nor immediate octlet
G.XNOR.I.128	Group exclusive-nor immediate hexlet

G.ANDN.I.size rd=rc.imm	→	G.AND.I.size rd=rc,~imm
G.COPY rd=rc	←	G.OR.I.128 rd=rc,0
G.NOT rd=rc	←	G.NOR.I.128 rd=rc,0
G.ORN.I.size rd=rc.imm	→	G.OR.I.size rd=rc,~imm
G.XNOR.I.size rd=rc.imm	→	G.XOR.I.size rd=rc,~imm

Redundancies

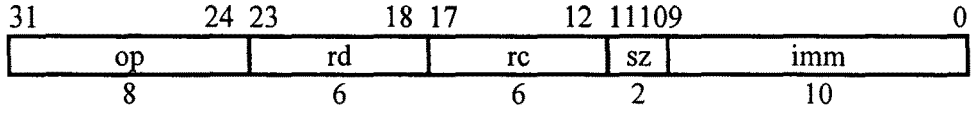
G.ADD.I.size rd=rc,0	⇔	*G.COPY rd=rc*
G.ADD.I.size.O rd=rc,0	⇔	*G.COPY rd=rc*
G.ADD.I.U.size.O rd=rc,0	⇔	*G.COPY rd=rc*
G.AND.I.size rd=rc,0	⇔	*G.ZERO rd*
G.AND.I.size rd=rc,-1	⇔	*G.COPY rd=rc*
G.NAND.I.size rd=rc,0	⇔	*G.SET rd*
G.NAND.I.size rd=rc,-1	⇔	*G.NOT rd=rc*
G.OR.I.size rd=rc,-1	⇔	*G.SET rd*
G.NOR.I.size rd=rc,-1	⇔	*G.ZERO rd*
G.XOR.I.size rd=rc,0	⇔	*G.COPY rd=rc*
G.XOR.I.size rd=rc,-1	⇔	*G.NOT rd=rc*

FIG. 88A *continued*

Format

op.size rd=rc,imm

rd=opsize(rc,imm)



sz ← log(size)-4

FIG. 88B

Definition

```
def GroupImmediate(op,size,rd,rc,imm) as
    c ← RegRead(rc, 128)
    s ← imm9
    case size of
        16:
            i16 ← s^7 || imm
            b ← i16 || i16 || i16 || i16 || i16 || i16 || i16 || i16
        32:
            b ← s^22 || imm || s^22 || imm || s^22 || imm || s^22 || imm
        64:
            b ← s^54 || imm || s^54 || imm
        128:
            b ← s^118 || imm
    endcase
    case op of
        G.AND.I:
            a ← c and b
        G.OR.I:
            a ← c or b
        G.NAND.I:
            a ← c nand b
        G.NOR.I:
            a ← c nor b
        G.XOR.I:
            a ← c xor b
        G.ADD.I:
            for i ← 0 to 128-size by size
                a_i+size-1..i ← c_i+size-1..i + b_i+size-1..i
            endfor
        G.ADD.I.O:
            for i ← 0 to 128-size by size
                t ← (c_i+size-1 || c_i+size-1..i) + (b_i+size-1 || b_i+size-1..i)
                if t_size ≠ t_size-1 then
                    raise FixedPointArithmetic
                endif
                a_i+size-1..i ← t_size-1..0
            endfor
```

FIG. 88C

```
        G.ADD.I.U.O:
            for i ← 0 to 128-size by size
                t ← (0^1 || c_i+size-1..i) + (0^1 || b_i+size-1..i)
                if t_size ≠ 0 then
                    raise FixedPointArithmetic
                endif
                a_i+size-1..i ← t_size-1..0
            endfor
    endcase
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Fixed-point arithmetic

FIG. 88C *continued*

Operation codes

G.SET.AND.E.I. 16	Group set and equal zero immediate doublets
G.SET.AND.E.I. 32	Group set and equal zero immediate quadlets
G.SET.AND.E.I. 64	Group set and equal zero immediate octlets
G.SET.AND.E.I.128	Group set and equal zero immediate hexlet
G.SET.AND.NE.I. 16	Group set and not equal zero immediate doublets
G.SET.AND.NE.I. 32	Group set and not equal zero immediate quadlets
G.SET.AND.NE.I. 64	Group set and not equal zero immediate octlets
G.SET.AND.NE.I.128	Group set and not equal zero immediate hexlet
G.SET.E.I. 16	Group set equal immediate doublets
G.SET.E.I. 32	Group set equal immediate quadlets
G.SET.E.I. 64	Group set equal immediate octlets
G.SET.E.I.128	Group set equal immediate hexlet
G.SET.GE.I. 16	Group set greater equal immediate signed doublets
G.SET.GE.I. 32	Group set greater equal immediate signed quadlets
G.SET.GE.I. 64	Group set greater equal immediate signed octlets
G.SET.GE.I.128	Group set greater equal immediate signed hexlet
G.SET.GE.I.U. 16	Group set greater equal immediate unsigned doublets
G.SET.GE.I.U. 32	Group set greater equal immediate unsigned quadlets
G.SET.GE.I.U. 64	Group set greater equal immediate unsigned octlets
G.SET.GE.I.U.128	Group set greater equal immediate unsigned hexlet
G.SET.L.I. 16	Group set signed less immediate doublets
G.SET.L.I. 32	Group set signed less immediate quadlets
G.SET.L.I. 64	Group set signed less immediate octlets
G.SET.L.I.128	Group set signed less immediate hexlet
G.SET.L.I.U. 16	Group set less immediate signed doublets
G.SET.L.I.U. 32	Group set less immediate signed quadlets
G.SET.L.I.U. 64	Group set less immediate signed octlets
G.SET.L.I.U.128	Group set less immediate signed hexlet
G.SET.NE.I. 16	Group set not equal immediate doublets
G.SET.NE.I. 32	Group set not equal immediate quadlets
G.SET.NE.I. 64	Group set not equal immediate octlets
G.SET.NE.I.128	Group set not equal immediate hexlet
G.SUB.I. 16	Group subtract immediate doublet
G.SUB.I. 16.O	Group subtract immediate signed doublet check overflow
G.SUB.I. 32	Group subtract immediate quadlet
G.SUB.I. 32.O	Group subtract immediate signed quadlet check overflow
G.SUB.I. 64	Group subtract immediate octlet
G.SUB.I. 64.O	Group subtract immediate signed octlet check overflow
G.SUB.I.128	Group subtract immediate hexlet
G.SUB.I.128.O	Group subtract immediate signed hexlet check overflow

FIG. 89A

G.SUB.I.U. 16.O	Group subtract immediate unsigned doublet check overflow
G.SUB.I.U. 32.O	Group subtract immediate unsigned quadlet check overflow
G.SUB.I.U. 64.O	Group subtract immediate unsigned octlet check overflow
G.SUB.I.U.128.O	Group subtract immediate unsigned hexlet check overflow

Equivalencies

G.NEG. 16	Group negate doublet
G.NEG. 16.O	Group negate signed doublet check overflow
G.NEG. 32	Group negate quadlet
G.NEG. 32.O	Group negate signed quadlet check overflow
G.NEG. 64	Group negate octlet
G.NEG. 64.O	Group negate signed octlet check overflow
G.NEG.128	Group negate hexlet
G.NEG.128.O	Group negate signed hexlet check overflow
G.SET.LE.I. 16	Group set less equal immediate signed doublets
G.SET.LE.I. 32	Group set less equal immediate signed quadlets
G.SET.LE.I. 64	Group set less equal immediate signed octlets
G.SET.LE.I.128	Group set less equal immediate signed hexlet
G.SET.LE.I.U. 16	Group set less equal immediate unsigned doublets
G.SET.LE.I.U. 32	Group set less equal immediate unsigned quadlets
G.SET.LE.I.U. 64	Group set less equal immediate unsigned octlets
G.SET.LE.I.U.128	Group set less equal immediate unsigned hexlet
G.SET.G.I. 16	Group set immediate signed greater doublets
G.SET.G.I. 32	Group set immediate signed greater quadlets
G.SET.G.I. 64	Group set immediate signed greater octlets
G.SET.G.I.128	Group set immediate signed greater hexlet
G.SET.G.I.U. 16	Group set greater immediate unsigned doublets
G.SET.G.I.U. 32	Group set greater immediate unsigned quadlets
G.SET.G.I.U. 64	Group set greater immediate unsigned octlets
G.SET.G.I.U.128	Group set greater immediate unsigned hexlet

G.NEG.size rd=rc	→	A.SUB.I.size rd=0,rc
G.NEG.size.O rd=rc	→	A.SUB.I.size.O rd=0,rc
G.SET.G.I.size rd=imm,rc	→	G.SET.GE.I.size rd=imm+1,rc
G.SET.G.I.U.size rd=imm,rc	→	G.SET.GE.I.U.size rd=imm+1,rc
G.SET.LE.I.size rd=imm,rc	→	G.SET.L.I.size rd=imm-1,rc
G.SET.LE.I.U.size rd=imm,rc	→	G.SET.L.I.U.size rd=imm-1,rc

FIG. 89A *continued*

Redundancies

G.SET.AND.E.I.size rd=rc,0	⇔	*G.SET.size rd*
G.SET.AND.NE.I.size rd=rc,0	⇔	*G.ZERO rd*
G.SET.AND.E.I.size rd=rc,-1	⇔	*G.SET.E.Z.size rd=rc*
G.SET.AND.NE.I.size rd=rc,-1	⇔	*G.SET.NE.Z.size rd=rc*
G.SET.E.I.size rd=rc,0	⇔	*G.SET.E.Z.size rd=rc*
G.SET.GE.I.size rd=rc,0	⇔	*G.SET.GE.Z.size rd=rc*
G.SET.L.I.size rd=rc,0	⇔	*G.SET.L.Z.size rd=rc*
G.SET.NE.I.size rd=rc,0	⇔	*G.SET.NE.Z.size rd=rc*
G.SET.GE.I.U.size rd=rc,0	⇔	*G.SET.GE.U.Z.size rd=rc*
G.SET.L.I.U.size rd=rc,0	⇔	*G.SET.L.U.Z.size rd=rc*

FIG. 89A *continued*

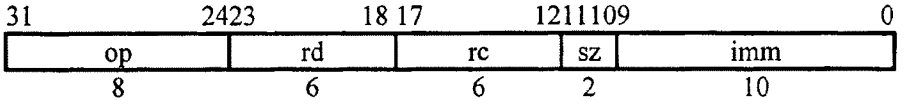
Selection

class	operation	cond	form	operand	size	check
arithmetic	SUB		I		16 32 64 128	
				NONE U	16 32 64 128	O
boolean	SET.AND SET	E NE	I		16 32 64 128	
	SET	L GE *G LE*	I	NONE U	16 32 64 128	

Format

op.size rd=imm,rc

rd=opsize(imm,rc)



sz ← log(size)-4

FIG. 89B

Definition

```
def GroupImmediateReversed(op,size,ra,imm) as
    c ← RegRead(rc, 128)
    s ← imm9
    case size of
        16:
            i16 ← s^7 || imm
            b ← i16 || i16 || i16 || i16 || i16 || i16 || i16 || i16
        32:
            b ← s^22 || imm || s^22 || imm || s^22 || imm || s^22 || imm
        64:
            b ← s^54 || imm || s^54 || imm
        128:
            b ← s^118 || imm
    endcase
    case op of
        G.SUB.I:
            for i ← 0 to 128-size by size
                a_i+size-1..i ← b_i+size-1..i - c_i+size-1..i
            endfor
        G.SUB.I.O:
            for i ← 0 to 128-size by size
                t ← (b_i+size-1 || b_i+size-1..i) - (c_i+size-1 || c_i+size-1..i)
                if (t_size ≠ t_size-1 then
                    raise FixedPointArithmetic
                endif
                a_i+size-1..i ← t_size-1..0
            endfor
        G.SUB.I.U.O:
            for i ← 0 to 128-size by size
                t ← (0^1 || b_i+size-1..i) - (0^1 || c_i+size-1..i)
                if (t_size ≠ 0 then
                    raise FixedPointArithmetic
                endif
                a_i+size-1..i ← t_size-1..0
            endfor
        G.SET.E.I:
            for i ← 0 to 128-size by size
                a_i+size-1..i ← (b_i+size-1..i = c_i+size-1..i)^size
            endfor
```

FIG. 89C

```
        G.SET.NE.I:
            for i ← 0 to 128-size by size
                a[i+size-1..i] ← (b[i+size-1..i] ≠ c[i+size-1..i])^size
            endfor
        G.SET.AND.E.I:
            for i ← 0 to 128-size by size
                a[i+size-1..i] ← ((b[i+size-1..i] and c[i+size-1..i]) = 0)^size
            endfor
        G.SET.AND.NE.I:
            for i ← 0 to 128-size by size
                a[i+size-1..i] ← ((b[i+size-1..i] and c[i+size-1..i]) ≠ 0)^size
            endfor
        G.SET.L.I:
            for i ← 0 to 128-size by size
                a[i+size-1..i] ← (b[i+size-1..i] < c[i+size-1..i])^size
            endfor
        G.SET.GE.I:
            for i ← 0 to 128-size by size
                a[i+size-1..i] ← (b[i+size-1..i] ≥ c[i+size-1..i])^size
            endfor
        G.SET.L.I.U:
            for i ← 0 to 128-size by size
                a[i+size-1..i] ← ((0 || b[i+size-1..i]) < (0 || c[i+size-1..i]))^size
            endfor
        G.SET.GE.I.U:
            for i ← 0 to 128-size by size
                a[i+size-1..i] ← ((0 || b[i+size-1..i]) ≥ (0 || c[i+size-1..i]))^size
            endfor
    endcase
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Fixed-point arithmetic

FIG. 89C *continued*

Operation codes

G.AAA.8	Group add add add bytes
G.AAA.16	Group add add add doublets
G.AAA.32	Group add add add quadlets
G.AAA.64	Group add add add octlets
G.AAA.128	Group add add add hexlet
G.ASA.8	Group add subtract add bytes
G.ASA.16	Group add subtract add doublets
G.ASA.32	Group add subtract add quadlets
G.ASA.64	Group add subtract add octlets
G.ASA.128	Group add subtract add hexlet

Equivalencies

G.AAS.8	Group add add subtract bytes
G.AAS.16	Group add add subtract doublets
G.AAS.32	Group add add subtract quadlets
G.AAS.64	Group add add subtract octlets
G.AAS.128	Group add add subtract hexlet

Redundancies

G.AAA.size rd@rc,rc	⇔	G.SHL.I.ADD.size rd=rd,rc,1
G.ASA.size rd@rc,rc	⇔	G.NOP

FIG. 90A

Format

G.op.size rd@rc,rb

rd=gopsize(rd,rc,rb)

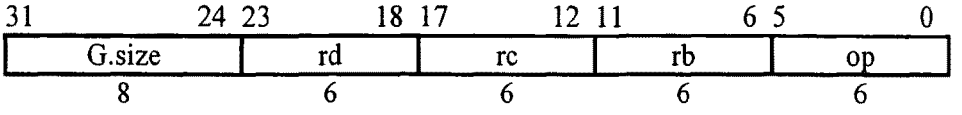


FIG. 90B

Definition

```
def GroupInplace(op,size,rd,rc,rb) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    for i ← 0 to 128-size by size
        case op of
            G.AAA:
                a[i+size-1..i] ← + d[i+size-1..i] + c[i+size-1..i] + b[i+size-1..i]
            G.ASA:
                a[i+size-1..i] ← + d[i+size-1..i] - c[i+size-1..i] + b[i+size-1..i]
        endcase
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

none

FIG. 90C

Operation codes

G.SHL.I.ADD. 8	Group shift left immediate add bytes
G.SHL.I.ADD. 16	Group shift left immediate add doublets
G.SHL.I.ADD. 32	Group shift left immediate add quadlets
G.SHL.I.ADD. 64	Group shift left immediate add octlets
G.SHL.I.ADD.128	Group shift left immediate add hexlet

Redundancies

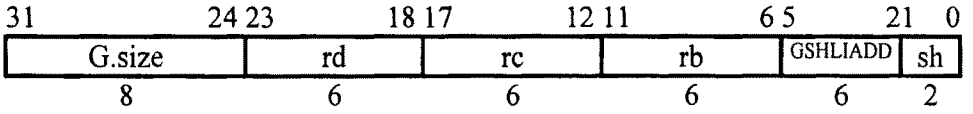
G.SHL.I.ADD.size rd=rd,rc,1	⇔	G.AAA.size rd@rc,rc

FIG. 91A

Format

G.op.size rd=rc,rb,i

rd=gopsize(rc,rb,i)



assert $1 \leq i \leq 4$
sh ← i-1

FIG. 91B

Definition

```
def GroupShiftLeftImmediateAdd(sh,size,ra,rb,rc)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    for i ← 0 to 128-size by size
        a_{i+size-1..i} ← c_{i+size-1..i} + (b_{i+size-1-sh..i} || 0^{1+sh})
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

none

FIG. 91C

Operation codes

G.SHL.I.SUB. 8	Group shift left immediate subtract bytes
G.SHL.I.SUB. 16	Group shift left immediate subtract doublets
G.SHL.I.SUB. 32	Group shift left immediate subtract quadlets
G.SHL.I.SUB. 64	Group shift left immediate subtract octlets
G.SHL.I.SUB.128	Group shift left immediate subtract hexlet

Redundancies

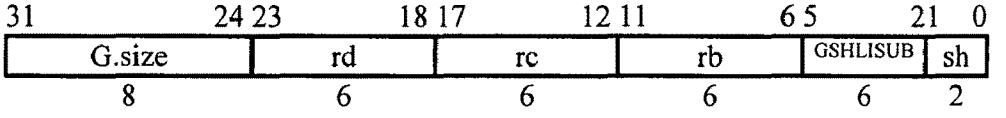
G.SHL.I.SUB.size rd=rc,1,rc	⇔	G.COPY rd=rc

FIG. 92A

Format

G.op.size rd=rb,i,rc

rd=gopsize(rb,i,rc)



assert 1≤i≤4
sh ← i-1

FIG. 92B

Definition

```
def GroupShiftLeftImmediateSubtract(sh,size,ra,rb,rc)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    for i ← 0 to 128-size by size
        a(i+size-1..i) ← (b(i+size-1-sh..i) || 0^(1+sh)) - c(i+size-1..i)
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

none

FIG. 92C