Exhibit U
Part 2

```
        elseif a.t=ZERO or b.t=ZERO then
                c.s ← a.s ^ b.s
                c.t ← ZERO
        else
                assert FALSE // should have covered al the cases above
        endif
enddef

def c ← fdivr(a,b) as
        if a.t=NORM and b.t=NORM then
                c.s ← a.s ^ b.s
                c.t ← NORM
                c.e ← a.e - b.e + 256
                c.f ← (a.f || 0^256) / b.f
        // priority is given to b operand for NaN propagation
        elseif (b.t=SNAN) or (b.t=QNAN) then
                c.s ← a.s ^ b.s
                c.t ← b.t
                c.e ← b.e
                c.f ← b.f
        elseif (a.t=SNAN) or (a.t=QNAN) then
                c.s ← a.s ^ b.s
                c.t ← a.t
                c.e ← a.e
                c.f ← a.f
        elseif a.t=ZERO and b.t=ZERO then
                c ← DEFAULTSNAN // Invalid
        elseif a.t=INFINITY and b.t=INFINITY then
                c ← DEFAULTSNAN // Invalid
        elseif a.t=ZERO then
                c.s ← a.s ^ b.s
                c.t ← ZERO
        elseif a.t=INFINITY then
                c.s ← a.s ^ b.s
                c.t ← INFINITY
        else
                assert FALSE // should have covered al the cases above
        endif
enddef

def msb ← findmsb(a) as
        MAXF ← 2^18 // Largest possible f value after matrix multiply
        for j ← 0 to MAXF
                if a_(MAXF-1..j) = (0^(MAXF-1-j) || 1) then
                        msb ← j
                endif
        endfor
enddef

def ai ← PackF(prec,a,round) as
        case a.t of
                NORM:
                        msb ← findmsb(a.f)
                        m ← msb-1-fbits(prec) // lsb for normal
                        rdn ← -ebias(prec)-a.e-1-fbits(prec) // lsb if a denormal
                        rb ← (m > rdn) ? m : rdn
```

FIG. 37 *continued*

```
        if rb ≤ 0 then
            aifr ← a.f_(msb-1..0) || 0^(-rb)
            eadj ← 0
        else
            case round of
                C:
                    s ← 0^(msb-rb) || (~a.s)^(rb)
                F:
                    s ← 0^(msb-rb) || (a.s)^(rb)
                N, NONE:
                    s ← 0^(msb-rb) || ~a.f_(rb) || a.f_(rb)^(rb-1)
                X:
                    if a.f_(rb-1..0) ≠ 0 then
                        raise FloatingPointArithmetic // Inexact
                    endif
                    s ← 0
                Z:
                    s ← 0
            endcase
            v ← (0||a.f_(msb..0)) + (0||s)
            if v_(msb) = 1 then
                aifr ← v_(msb-1..rb)
                eadj ← 0
            else
                aifr ← 0^(fbits(prec))
                eadj ← 1
            endif
        endif
        aien ← a.e + msb − 1 + eadj + ebias(prec)
        if aien ≤ 0 then
            if round = NONE then
                ai ← a.s || 0^(ebits(prec)) || aifr
            else
                raise FloatingPointArithmetic //Underflow
            endif
        elseif aien ≥ 1^(ebits(prec)) then
            if round = NONE then
                //default: round-to-nearest overflow handling
                ai ← a.s || 1^(ebits(prec)) || 0^(fbits(prec))
            else
                raise FloatingPointArithmetic //Underflow
            endif
        else
            ai ← a.s || aien_(ebits(prec)-1..0) || aifr
        endif
    SNAN:
        if round ≠ NONE then
            raise FloatingPointArithmetic //Invalid
        endif
        if −a.e < fbits(prec) then
            ai ← a.s || 1^(ebits(prec)) || a.f_(-a.e-1..0) || 0^(fbits(prec)+a.e)
```

FIG. 37 *continued*

```
                else
                        lsb ← a.f-a.e-1-fbits(prec)+1..0 ≠ 0
                        ai ← a.s || 1^ebits(prec) || a.f-a.e-1..-a.e-1-fbits(prec)+2 || lsb
                endif
        QNAN:
                if -a.e < fbits(prec) then
                        ai ← a.s || 1^ebits(prec) || a.f-a.e-1..0 || 0^fbits(prec)+a.e
                else
                        lsb ← a.f-a.e-1-fbits(prec)+1..0 ≠ 0
                        ai ← a.s || 1^ebits(prec) || a.f-a.e-1..-a.e-1-fbits(prec)+2 || lsb
                endif
        ZERO:
                ai ← a.s || 0^ebits(prec) || 0^fbits(prec)
        INFINITY:
                ai ← a.s || 1^ebits(prec) || 0^fbits(prec)
    endcase
enddef

def ai ← fsinkr(prec, a, round) as
    case a.t of
        NORM:
                msb ← findmsb(a.f)
                rb ← -a.e
                if rb ≤ 0 then
                        aifr ← a.f_msb..0 || 0^-rb
                        aims ← msb - rb
                else
                        case round of
                                C, C.D:
                                        s ← 0^msb-rb || (~ai.s)^rb
                                F, F.D:
                                        s ← 0^msb-rb || (ai.s)^rb
                                N, NONE:
                                        s ← 0^msb-rb || ~ai.f_rb || ai.f_rb^rb-1
                                X:
                                        if ai.f_rb-1..0 ≠ 0 then
                                                raise FloatingPointArithmetic // Inexact
                                        endif
                                        s ← 0
                                Z, Z.D:
                                        s ← 0
                        endcase
                        v ← (0||a.f_msb..0) + (0||s)
                        if v_msb = 1 then
                                aims ← msb + 1 - rb
                        else
                                aims ← msb - rb
                        endif
                        aifr ← v_aims..rb
                endif
                if aims > prec then
                        case round of
                                C.D, F.D, NONE, Z.D:
                                        ai ← a.s || (~as)^prec-1
```

FIG. 37 *continued*

```
                        C, F, N, X, Z:
                            raise FloatingPointArithmetic // Overflow
                    endcase
                elseif a.s = 0 then
                    ai ← aifr
                else
                    ai ← -aifr
                endif
            ZERO:
                ai ← 0^prec
            SNAN, QNAN:
                case round of
                    C.D, F.D, NONE, Z.D:
                        ai ← 0^prec
                    C, F, N, X, Z:
                        raise FloatingPointArithmetic // Invalid
                endcase
            INFINITY:
                case round of
                    C.D, F.D, NONE, Z.D:
                        ai ← a.s || (~as)^prec-1
                    C, F, N, X, Z:
                        raise FloatingPointArithmetic // Invalid
                endcase
    endcase
enddef

def c ← frecrest(a) as
    b.s ← 0
    b.t ← NORM
    b.e ← 0
    b.f ← 1
    c ← fest(fdiv(b,a))
enddef

def c ← frsqrest(a) as
    b.s ← 0
    b.t ← NORM
    b.e ← 0
    b.f ← 1
    c ← fest(fsqr(fdiv(b,a)))
enddef

def c ← fest(a) as
    if (a.t=NORM) then
        msb ← findmsb(a.f)
        a.e ← a.e + msb - 13
        a.f ← a.f_msb..msb-12 || 1
    else
        c ← a
    endif
enddef

def c ← fsqr(a) as
    if (a.t=NORM) and (a.s=0) then
        c.s ← 0
        c.t ← NORM
        if (a.e_0 = 1) then
```

FIG. 37 *continued*

```
                c.e ← (a.e-127) / 2
                c.f ← sqr(a.f || 0^127)
            else
                c.e ← (a.e-128) / 2
                c.f ← sqr(a.f || 0^128)
            endif
        elseif (a.t=SNAN) or (a.t=QNAN) or a.t=ZERO or ((a.t=INFINITY) and (a.s=0)) then
            c ← a
        elseif ((a.t=NORM) or (a.t=INFINITY)) and (a.s=1) then
            c ← DEFAULTSNAN // Invalid
        else
            assert FALSE // should have covered al the cases above
        endif
enddef
```

FIG. 37 *continued*

E.ADD.F.16	Ensemble add floating-point half
E.ADD.F.16.C	Ensemble add floating-point half ceiling
E.ADD.F.16.F	Ensemble add floating-point half floor
E.ADD.F.16.N	Ensemble add floating-point half nearest
E.ADD.F.16.X	Ensemble add floating-point half exact
E.ADD.F.16.Z	Ensemble add floating-point half zero
E.ADD.F.32	Ensemble add floating-point single
E.ADD.F.32.C	Ensemble add floating-point single ceiling
E.ADD.F.32.F	Ensemble add floating-point single floor
E.ADD.F.32.N	Ensemble add floating-point single nearest
E.ADD.F.32.X	Ensemble add floating-point single exact
E.ADD.F.32.Z	Ensemble add floating-point single zero
E.ADD.F.64	Ensemble add floating-point double
E.ADD.F.64.C	Ensemble add floating-point double ceiling
E.ADD.F.64.F	Ensemble add floating-point double floor
E.ADD.F.64.N	Ensemble add floating-point double nearest
E.ADD.F.64.X	Ensemble add floating-point double exact
E.ADD.F.64.Z	Ensemble add floating-point double zero
E.ADD.F.128	Ensemble add floating-point quad
E.ADD.F.128.C	Ensemble add floating-point quad ceiling
E.ADD.F.128.F	Ensemble add floating-point quad floor
E.ADD.F.128.N	Ensemble add floating-point quad nearest
E.ADD.F.128.X	Ensemble add floating-point quad exact
E.ADD.F.128.Z	Ensemble add floating-point quad zero
E.DIV.F.16	Ensemble divide floating-point half
E.DIV.F.16.C	Ensemble divide floating-point half ceiling
E.DIV.F.16.F	Ensemble divide floating-point half floor
E.DIV.F.16.N	Ensemble divide floating-point half nearest
E.DIV.F.16.X	Ensemble divide floating-point half exact
E.DIV.F.16.Z	Ensemble divide floating-point half zero
E.DIV.F.32	Ensemble divide floating-point single
E.DIV.F.32.C	Ensemble divide floating-point single ceiling
E.DIV.F.32.F	Ensemble divide floating-point single floor
E.DIV.F.32.N	Ensemble divide floating-point single nearest
E.DIV.F.32.X	Ensemble divide floating-point single exact
E.DIV.F.32.Z	Ensemble divide floating-point single zero
E.DIV.F.64	Ensemble divide floating-point double

FIG. 38A

E.DIV.F.64.C	Ensemble divide floating-point double ceiling
E.DIV.F.64.F	Ensemble divide floating-point double floor
E.DIV.F.64.N	Ensemble divide floating-point double nearest
E.DIV.F.64.X	Ensemble divide floating-point double exact
E.DIV.F.64.Z	Ensemble divide floating-point double zero
E.DIV.F.128	Ensemble divide floating-point quad
E.DIV.F.128.C	Ensemble divide floating-point quad ceiling
E.DIV.F.128.F	Ensemble divide floating-point quad floor
E.DIV.F.128.N	Ensemble divide floating-point quad nearest
E.DIV.F.128.X	Ensemble divide floating-point quad exact
E.DIV.F.128.Z	Ensemble divide floating-point quad zero
E.MUL.C.F.16	Ensemble multiply complex floating-point half
E.MUL.C.F.32	Ensemble multiply complex floating-point single
E.MUL.C.F.64	Ensemble multiply complex floating-point double
E.MUL.F.16	Ensemble multiply floating-point half
E.MUL.F.16.C	Ensemble multiply floating-point half ceiling
E.MUL.F.16.F	Ensemble multiply floating-point half floor
E.MUL.F.16.N	Ensemble multiply floating-point half nearest
E.MUL.F.16.X	Ensemble multiply floating-point half exact
E.MUL.F.16.Z	Ensemble multiply floating-point half zero
E.MUL.F.32	Ensemble multiply floating-point single
E.MUL.F.32.C	Ensemble multiply floating-point single ceiling
E.MUL.F.32.F	Ensemble multiply floating-point single floor
E.MUL.F.32.N	Ensemble multiply floating-point single nearest
E.MUL.F.32.X	Ensemble multiply floating-point single exact
E.MUL.F.32.Z	Ensemble multiply floating-point single zero
E.MUL.F.64	Ensemble multiply floating-point double
E.MUL.F.64.C	Ensemble multiply floating-point double ceiling
E.MUL.F.64.F	Ensemble multiply floating-point double floor
E.MUL.F.64.N	Ensemble multiply floating-point double nearest
E.MUL.F.64.X	Ensemble multiply floating-point double exact
E.MUL.F.64.Z	Ensemble multiply floating-point double zero
E.MUL.F.128	Ensemble multiply floating-point quad
E.MUL.F.128.C	Ensemble multiply floating-point quad ceiling
E.MUL.F.128.F	Ensemble multiply floating-point quad floor
E.MUL.F.128.N	Ensemble multiply floating-point quad nearest
E.MUL.F.128.X	Ensemble multiply floating-point quad exact
E.MUL.F.128.Z	Ensemble multiply floating-point quad zero

FIG. 38A *continued*

Selection

class	op	prec	round/trap
add	EADDF	16 32 64 128	NONE C F N X Z
divide	EDIVF	16 32 64 128	NONE C F N X Z
multiply	EMULF	16 32 64 128	NONE C F N X Z
complex multiply	EMUL.CF	16 32 64	NONE

Format

E.op.prec.round rd=rc,rb

rd=eopprecround(rc,rb)

31 24	23 18	17 12	11 6	5 0
E.prec	rd	rc	rb	op.round
8	6	6	6	6

FIG. 38B

Definition

```
def mul(size,v,i,w,j) as
      mul ← fmul(F(size,v_{size-1+i..i}),F(size,w_{size-1+j..j}))
enddef

def EnsembleFloatingPoint(op,prec,round,ra,rb,rc) as
      c ← RegRead(rc, 128)
      b ← RegRead(rb, 128)
      for i ← 0 to 128-prec by prec
            ci ← F(prec,c_{i+prec-1..i})
            bi ← F(prec,b_{i+prec-1..i})
            case op of
                  E.ADD.F:
                        ai ← faddr(ci,bi,round)
                  E.MUL.F:
                        ai ← fmul(ci,bi)
                  E.MUL.C.F:
                        if (i and prec) then
                              ai ← fadd(mul(prec,c,i,b,i-prec), mul(prec,c,i-prec,b,i))
                        else
                              ai ← fsub(mul(prec,c,i,b,i), mul(prec,c,i+prec,b,i+prec))
                        endif
                  E.DIV.F.:
                        ai ← fdiv(ci,bi)
            endcase
            a_{i+prec-1..i} ← PackF(prec, ai, round)
      endfor
      RegWrite(rd, 128, a)
enddef
```

Exceptions

Floating-point arithmetic

FIG. 38C

Operation codes

E.MUL.ADD.C.F.16	Ensemble multiply add complex floating-point half
E.MUL.ADD.C.F.32	Ensemble multiply add complex floating-point single
E.MUL.ADD.C.F.64	Ensemble multiply add complex floating-point double
E.MUL.ADD.F.16	Ensemble multiply add floating-point half
E.MUL.ADD.F.16.C	Ensemble multiply add floating-point half ceiling
E.MUL.ADD.F.16.F	Ensemble multiply add floating-point half floor
E.MUL.ADD.F.16.N	Ensemble multiply add floating-point half nearest
E.MUL.ADD.F.16.X	Ensemble multiply add floating-point half exact
E.MUL.ADD.F.16.Z	Ensemble multiply add floating-point half zero
E.MUL.ADD.F.32	Ensemble multiply add floating-point single
E.MUL.ADD.F.32.C	Ensemble multiply add floating-point single ceiling
E.MUL.ADD.F.32.F	Ensemble multiply add floating-point single floor
E.MUL.ADD.F.32.N	Ensemble multiply add floating-point single nearest
E.MUL.ADD.F.32.X	Ensemble multiply add floating-point single exact
E.MUL.ADD.F.32.Z	Ensemble multiply add floating-point single zero
E.MUL.ADD.F.64	Ensemble multiply add floating-point double
E.MUL.ADD.F.64.C	Ensemble multiply add floating-point double ceiling
E.MUL.ADD.F.64.F	Ensemble multiply add floating-point double floor
E.MUL.ADD.F.64.N	Ensemble multiply add floating-point double nearest
E.MUL.ADD.F.64.X	Ensemble multiply add floating-point double exact
E.MUL.ADD.F.64.Z	Ensemble multiply add floating-point double zero
E.MUL.ADD.F.128	Ensemble multiply add floating-point quad
E.MUL.ADD.F.128.C	Ensemble multiply add floating-point quad ceiling
E.MUL.ADD.F.128.F	Ensemble multiply add floating-point quad floor
E.MUL.ADD.F.128.N	Ensemble multiply add floating-point quad nearest
E.MUL.ADD.F.128.X	Ensemble multiply add floating-point quad exact
E.MUL.ADD.F.128.Z	Ensemble multiply add floating-point quad zero
E.MUL.SUB.C.F.16	Ensemble multiply subtract complex floating-point half
E.MUL.SUB.C.F.32	Ensemble multiply subtract complex floating-point single
E.MUL.SUB.C.F.64	Ensemble multiply subtract complex floating-point double
E.MUL.SUB.F.16	Ensemble multiply subtract floating-point half
E.MUL.SUB.F.32	Ensemble multiply subtract floating-point single
E.MUL.SUB.F.64	Ensemble multiply subtract floating-point double
E.MUL.SUB.F.128	Ensemble multiply subtract floating-point quad

FIG. 38D

Selection

class	op	type	prec	round/trap
multiply add	E.MUL.ADD	F	16 32 64 128	NONE C F N X Z
		C.F	16 32 64	NONE
multiply subtract	E.MUL.SUB	F	16 32 64 128	NONE
		C.F	16 32 64	NONE

Format

E.op.size rd@rc,rb

rd=eopsize(rd,rc,rb)

31..24	23..18	17..12	11..6	5..0
E.size	rd	rc	rb	op
8	6	6	6	6

FIG. 38E

Definition

```
def mul(size,v,i,w,j) as
    mul ← fmul(F(size,v_{size-1+i..i}),F(size,w_{size-1+j..j}))
enddef

def EnsembleInplaceFloatingPoint(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
        di ← F(prec,d_{i+prec-1..i})
        case op of
            E.MUL.ADD.F:
                ai ← fadd(di, mul(prec,c,i,b,i))
            E.MUL.ADD.C.F:
                if (i and prec) then
                    ai ← fadd(di, fadd(mul(prec,c,i,b,i-prec), mul(c,i-prec,b,i)))
                else
                    ai ← fadd(di, fsub(mul(prec,c,i,b,i), mul(prec,c,i+prec,b,i+prec)))
                endif
            E.MUL.SUB.F:
                ai ← frsub(di, mul(prec,c,i,b,i))
            E.MUL.SUB.C.F:
                if (i and prec) then
                    ai ← frsub(di, fadd(mul(prec,c,i,b,i-prec), mul(c,i-prec,b,i)))
                else
                    ai ← frsub(di, fsub(mul(prec,c,i,b,i), mul(prec,c,i+prec,b,i+prec)))
                endif
        endcase
        a_{i+prec-1..i} ← PackF(prec, ai, round)
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

none

FIG. 38F

Operation codes

E.SCAL.ADD.F.16	Ensemble scale add floating-point half
E.SCAL.ADD.F.32	Ensemble scale add floating-point single
E.SCAL.ADD.F.64	Ensemble scale add floating-point double

FIG. 38G

Selection

class-	op	prec
scale add	E.SCAL.ADD.F	16 32 64

Format

E.SCAL.ADD.F.size ra=rd,rc,rb

ra=escaladdfsize(rd,rc,rb)

31 24	23 18	17 12	11 6	5 0
op	rd	rc	rb	ra
8	6	6	6	6

FIG. 38H

Definition

def EnsembleFloatingPointTernary(op,prec,rd,rc,rb,ra) as
  d ← RegRead(rd, 128)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  for i ← 0 to 128-prec by prec
    di ← F(prec,$d_{i+prec-1..i}$)
    ci ← F(prec,$c_{i+prec-1..i}$)
    ai ← fadd(fmul(di, F(prec,$b_{prec-1..0}$)), fmul(ci, F(prec,$b_{2*prec-1..prec}$)))
    $a_{i+prec-1..i}$ ← PackF(prec, ai, none)
  endfor
  RegWrite(ra, 128, a)
enddef

Exceptions

none

FIG. 38I

E.SUB.F.16	Ensemble subtract floating-point half
E.SUB.F.16.C	Ensemble subtract floating-point half ceiling
E.SUB.F.16.F	Ensemble subtract floating-point half floor
E.SUB.F.16.N	Ensemble subtract floating-point half nearest
E.SUB.F.16.Z	Ensemble subtract floating-point half zero
E.SUB.F.16.X	Ensemble subtract floating-point half exact
E.SUB.F.32	Ensemble subtract floating-point single
E.SUB.F.32.C	Ensemble subtract floating-point single ceiling
E.SUB.F.32.F	Ensemble subtract floating-point single floor
E.SUB.F.32.N	Ensemble subtract floating-point single nearest
E.SUB.F.32.Z	Ensemble subtract floating-point single zero
E.SUB.F.32.X	Ensemble subtract floating-point single exact
E.SUB.F.64	Ensemble subtract floating-point double
E.SUB.F.64.C	Ensemble subtract floating-point double ceiling
E.SUB.F.64.F	Ensemble subtract floating-point double floor
E.SUB.F.64.N	Ensemble subtract floating-point double nearest
E.SUB.F.64.Z	Ensemble subtract floating-point double zero
E.SUB.F.64.X	Ensemble subtract floating-point double exact
E.SUB.F.128	Ensemble subtract floating-point quad
E.SUB.F.128.C	Ensemble subtract floating-point quad ceiling
E.SUB.F.128.F	Ensemble subtract floating-point quad floor
E.SUB.F.128.N	Ensemble subtract floating-point quad nearest
E.SUB.F.128.Z	Ensemble subtract floating-point quad zero
E.SUB.F.128.X	Ensemble subtract floating-point quad exact

FIG. 39A

Selection

class	op	prec	round/trap
set	SET. E LG L GE	16 32 64 128	NONE X
subtract	SUB	16 32 64 128	NONE C F N X Z

Format

E.op.prec.round rd=rb,rc

rd=eopprecround(rb,rc)

31 24	23 18	17 12	11 6	5 0
E.prec	rd	rc	rb	op.round
8	6	6	6	6

FIG. 39B

Definition

```
def EnsembleReversedFloatingPoint(op,prec,round,rd,rc,rb) as
        c ← RegRead(rc, 128)
        b ← RegRead(rb, 128)
        for i ← 0 to 128-prec by prec
                ci ← F(prec,c_{i+prec-1..i})
                bi ← F(prec,b_{i+prec-1..i})
                ai ← frsubr(ci,-bi, round)
                a_{i+prec-1..i} ← PackF(prec, ai, round)
        endfor
        RegWrite(rd, 128, a)
enddef
```

Exceptions

Floating-point arithmetic

FIG. 39C

Operation codes

G.SET.E.F.16	Group set equal floating-point half
G.SET.E.F.16.X	Group set equal floating-point half exact
G.SET.E.F.32	Group set equal floating-point single
G.SET.E.F.32.X	Group set equal floating-point single exact
G.SET.E.F.64	Group set equal floating-point double
G.SET.E.F.64.X	Group set equal floating-point double exact
G.SET.E.F.128	Group set equal floating-point quad
G.SET.E.F.128.X	Group set equal floating-point quad exact
G.SET.GE.F.16.X	Group set greater equal floating-point half exact
G.SET.GE.F.32.X	Group set greater equal floating-point single exact
G.SET.GE.F.64.X	Group set greater equal floating-point double exact
G.SET.GE.F.128.X	Group set greater equal floating-point quad exact
G.SET.LG.F.16	Group set less greater floating-point half
G.SET.LG.F.16.X	Group set less greater floating-point half exact
G.SET.LG.F.32	Group set less greater floating-point single
G.SET.LG.F.32.X	Group set less greater floating-point single exact
G.SET.LG.F.64	Group set less greater floating-point double
G.SET.LG.F.64.X	Group set less greater floating-point double exact
G.SET.LG.F.128	Group set less greater floating-point quad
G.SET.LG.F.128.X	Group set less greater floating-point quad exact
G.SET.L.F.16	Group set less floating-point half
G.SET.L.F.16.X	Group set less floating-point half exact
G.SET.L.F.32	Group set less floating-point single
G.SET.L.F.32.X	Group set less floating-point single exact
G.SET.L.F.64	Group set less floating-point double
G.SET.L.F.64.X	Group set less floating-point double exact
G.SET.L.F.128	Group set less floating-point quad
G.SET.L.F.128.X	Group set less floating-point quad exact
G.SET.GE.F.16	Group set greater equal floating-point half
G.SET.GE.F.32	Group set greater equal floating-point single
G.SET.GE.F.64	Group set greater equal floating-point double
G.SET.GE.F.128	Group set greater equal floating-point quad

FIG. 39D

Equivalencies

G.SET.LE.F.16.X	Group set less equal floating-point half exact
G.SET.LE.F.32.X	Group set less equal floating-point single exact
G.SET.LE.F.64.X	Group set less equal floating-point double exact
G.SET.LE.F.128.X	Group set less equal floating-point quad exact
G.SET.G.F.16	Group set greater floating-point half
G.SET.G.F.16.X	Group set greater floating-point half exact
G.SET.G.F.32	Group set greater floating-point single
G.SET.G.F.32.X	Group set greater floating-point single exact
G.SET.G.F.64	Group set greater floating-point double
G.SET.G.F.64.X	Group set greater floating-point double exact
G.SET.G.F.128	Group set greater floating-point quad
G.SET.G.F.128.X	Group set greater floating-point quad exact
G.SET.LE.F.16	Group set less equal floating-point half
G.SET.LE.F.32	Group set less equal floating-point single
G.SET.LE.F.64	Group set less equal floating-point double
G.SET.LE.F.128	Group set less equal floating-point quad

G.SET.G.F.prec rd=rb,rc	→	G.SET.L.F.prec rd=rc,rb
G.SET.G.F.prec.X rd=rb,rc	→	G.SET.L.F.prec.X rd=rc,rb
G.SET.LE.F.prec rd=rb,rc	→	G.SET.GE.F.prec rd=rc,rb
G.SET.LE.F.prec.X rd=rb,rc	→	G.SET.GE.F.prec.X rd=rc,rb

FIG. 39E

Selection

class	op	prec	round/trap
set	SET. E LG L GE *G LE*	16 32 64 128	NONE X

Format

G.op.prec.round rd=rb,rc

rc=gopprecround(rb,ra)

31 24	23 18	17 12	11 6	5 0
G.prec	rd	rc	rb	op.round
8	6	6	6	6

FIG. 39F

Definition

```
def GroupFloatingPointReversed(op,prec,round,rd,rc,rb) as
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    for i ← 0 to 128-prec by prec
        ci ← F(prec,c_{i+prec-1..i})
        bi ← F(prec,b_{i+prec-1..i})
        if round≠NONE then
            if (di.t = SNAN) or (ci.t = SNAN) then
                raise FloatingPointArithmetic
            endif
            case op of
                G.SET.L.F, G.SET.GE.F:
                    if (di.t = QNAN) or (ci.t = QNAN) then
                        raise FloatingPointArithmetic
                    endif
                others: //nothing
            endcase
        endif
        case op of
            G.SET.L.F:
                ai ← bi?≥ci
            G.SET.GE.F:
                ai ← bi!?<ci
            G.SET.E.F:
                ai ← bi=ci
            G.SET.LG.F:
                ai ← bi≠ci
        endcase
        a_{i+prec-1..i} ← ai^prec
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Floating-point arithmetic

FIG. 39G

Operation codes

G.COM.E.F. 16	Group compare equal floating-point half
G.COM.E.F. 16.X	Group compare equal floating-point half exact
G.COM.E.F. 32	Group compare equal floating-point single
G.COM.E.F. 32.X	Group compare equal floating-point single exact
G.COM.E.F. 64	Group compare equal floating-point double
G.COM.E.F. 64.X	Group compare equal floating-point double exact
G.COM.E.F.128	Group compare equal floating-point quad
G.COM.E.F.128.X	Group compare equal floating-point quad exact
G.COM.GE.F. 16	Group compare greater or equal floating-point half
G.COM.GE.F. 16.X	Group compare greater or equal floating-point half exact
G.COM.GE.F. 32	Group compare greater or equal floating-point single
G.COM.GE.F. 32.X	Group compare greater or equal floating-point single exact
G.COM.GE.F. 64	Group compare greater or equal floating-point double
G.COM.GE.F. 64.X	Group compare greater or equal floating-point double exact
G.COM.GE.F.128	Group compare greater or equal floating-point quad
G.COM.GE.F.128.X	Group compare greater or equal floating-point quad exact
G.COM.L.F. 16	Group compare less floating-point half
G.COM.L.F. 16.X	Group compare less floating-point half exact
G.COM.L.F. 32	Group compare less floating-point single
G.COM.L.F. 32.X	Group compare less floating-point single exact
G.COM.L.F. 64	Group compare less floating-point double
G.COM.L.F. 64.X	Group compare less floating-point double exact
G.COM.L.F.128	Group compare less floating-point quad
G.COM.L.F.128.X	Group compare less floating-point quad exact
G.COM.LG.F. 16	Group compare less or greater floating-point half
G.COM.LG.F. 16.X	Group compare less or greater floating-point half exact
G.COM.LG.F. 32	Group compare less or greater floating-point single
G.COM.LG.F. 32.X	Group compare less or greater floating-point single exact
G.COM.LG.F. 64	Group compare less or greater floating-point double
G.COM.LG.F. 64.X	Group compare less or greater floating-point double exact
G.COM.LG.F.128	Group compare less or greater floating-point quad
G.COM.LG.F.128.X	Group compare less or greater floating-point quad exact

FIG. 40A

Equivalencies

G.COM.G.F. 16	Group compare greater floating-point half
G.COM.G.F. 16.X	Group compare greater floating-point half exact
G.COM.G.F. 32	Group compare greater floating-point single
G.COM.G.F. 32.X	Group compare greater floating-point single exact
G.COM.G.F. 64	Group compare greater floating-point double
G.COM.G.F. 64.X	Group compare greater floating-point double exact
G.COM.G.F.128	Group compare greater floating-point quad
G.COM.G.F.128.X	Group compare greater floating-point quad exact
G.COM.LE.F. 16	Group compare less equal floating-point half
G.COM.LE.F. 16.X	Group compare less equal floating-point half exact
G.COM.LE.F. 32	Group compare less equal floating-point single
G.COM.LE.F. 32.X	Group compare less equal floating-point single exact
G.COM.LE.F. 64	Group compare less equal floating-point double
G.COM.LE.F. 64.X	Group compare less equal floating-point double exact
G.COM.LE.F.128	Group compare less equal floating-point quad
G.COM.LE.F.128.X	Group compare less equal floating-point quad exact

G.COM.G.F.prec rd,rc	→	G.COM.L.F.prec rc,rd
G.COM.G.F.prec.X rd,rc	→	G.COM.L.F.prec.X rc,rd
G.COM.LE.F.prec rd,rc	→	G.COM.GE.F.prec rc,rd
G.COM.LE.F.prec.X rd,rc	→	G.COM.GE.F.prec.X rc,rd

FIG. 40A *continued*

Selection

class	op	cond	type	prec	round/trap
set	COM	E LG L GE *G LE*	F	16 32 64 128	NONE X

Format

G.COM.op.prec.round rd,rc

rc=gcomopprecround(rd,rc)

31 24	23 18	17 12	11 6	5 0
G.prec	rd	rc	op	GCOM
8	6	6	6	6

FIG. 40B

Definition

```
def GroupCompareFloatingPoint(op,prec,round,rd,rc) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    for i ← 0 to 128-prec by prec
        di ← F(prec,d_{i+prec-1..i})
        ci ← F(prec,c_{i+prec-1..i})
        if round≠NONE then
            if (di.t = SNAN) or (ci.t = SNAN) then
                raise FloatingPointArithmetic
            endif
            case op of
                G.COM.L.F, G.COM.GE.F:
                    if (di.t = QNAN) or (ci.t = QNAN) then
                        raise FloatingPointArithmetic
                    endif
                others: //nothing
            endcase
        endif
        case op of
            G.COM.L.F:
                ai ← di?≥ci
            G.COM.GE.F:
                ai ← di!?<ci
            G.COM.E.F:
                ai ← di=ci
            G.COM.LG.F:
                ai ← di≠ci
        endcase
        a_{i+prec-1..i} ← ai
    endfor
    if (a ≠ 0) then
        raise FloatingPointArithmetic
    endif
enddef
```

Exceptions

Floating-point arithmetic

FIG. 40C

E.ABS.F.16	Ensemble absolute value floating-point half
E.ABS.F.16.X	Ensemble absolute value floating-point half exception
E.ABS.F.32	Ensemble absolute value floating-point single
E.ABS.F.32.X	Ensemble absolute value floating-point single exception
E.ABS.F.64	Ensemble absolute value floating-point double
E.ABS.F.64.X	Ensemble absolute value floating-point double exception
E.ABS.F.128	Ensemble absolute value floating-point quad
E.ABS.F.128.X	Ensemble absolute value floating-point quad exception
E.COPY.F.16	Ensemble copy floating-point half
E.COPY.F.16.X	Ensemble copy floating-point half exception
E.COPY.F.32	Ensemble copy floating-point single
E.COPY.F.32.X	Ensemble copy floating-point single exception
E.COPY.F.64	Ensemble copy floating-point double
E.COPY.F.64.X	Ensemble copy floating-point double exception
E.COPY.F.128	Ensemble copy floating-point quad
E.COPY.F.128.X	Ensemble copy floating-point quad exception
E.DEFLATE.F.32	Ensemble convert floating-point half from single
E.DEFLATE.F.32.C	Ensemble convert floating-point half from single ceiling
E.DEFLATE.F.32.F	Ensemble convert floating-point half from single floor
E.DEFLATE.F.32.N	Ensemble convert floating-point half from single nearest
E.DEFLATE.F.32.X	Ensemble convert floating-point half from single exact
E.DEFLATE.F.32.Z	Ensemble convert floating-point half from single zero
E.DEFLATE.F.64	Ensemble convert floating-point single from double
E.DEFLATE.F.64.C	Ensemble convert floating-point single from double ceiling
E.DEFLATE.F.64.F	Ensemble convert floating-point single from double floor
E.DEFLATE.F.64.N	Ensemble convert floating-point single from double nearest
E.DEFLATE.F.64.X	Ensemble convert floating-point single from double exact
E.DEFLATE.F.64.Z	Ensemble convert floating-point single from double zero
E.DEFLATE.F.128	Ensemble convert floating-point double from quad
E.DEFLATE.F.128.C	Ensemble convert floating-point double from quad ceiling
E.DEFLATE.F.128.F	Ensemble convert floating-point double from quad floor
E.DEFLATE.F.128.N	Ensemble convert floating-point double from quad nearest
E.DEFLATE.F.128.X	Ensemble convert floating-point double from quad exact
E.DEFLATE.F.128.Z	Ensemble convert floating-point double from quad zero
E.FLOAT.F.16	Ensemble convert floating-point half from doublets
E.FLOAT.F.16.C	Ensemble convert floating-point half from doublets ceiling
E.FLOAT.F.16.F	Ensemble convert floating-point half from doublets floor
E.FLOAT.F.16.N	Ensemble convert floating-point half from doublets nearest
E.FLOAT.F.16.X	Ensemble convert floating-point half from doublets exact
E.FLOAT.F.16.Z	Ensemble convert floating-point half from doublets zero

FIG. 41A

E.FLOAT.F.32	Ensemble convert floating-point single from quadlets
E.FLOAT.F.32.C	Ensemble convert floating-point single from quadlets ceiling
E.FLOAT.F.32.F	Ensemble convert floating-point single from quadlets floor
E.FLOAT.F.32.N	Ensemble convert floating-point single from quadlets nearest
E.FLOAT.F.32.X	Ensemble convert floating-point single from quadlets exact
E.FLOAT.F.32.Z	Ensemble convert floating-point single from quadlets zero
E.FLOAT.F.64	Ensemble convert floating-point double from octlets
E.FLOAT.F.64.C	Ensemble convert floating-point double from octlets ceiling
E.FLOAT.F.64.F	Ensemble convert floating-point double from octlets floor
E.FLOAT.F.64.N	Ensemble convert floating-point double from octlets nearest
E.FLOAT.F.64.X	Ensemble convert floating-point double from octlets exact
E.FLOAT.F.64.Z	Ensemble convert floating-point double from octlets zero
E.FLOAT.F.128	Ensemble convert floating-point quad from hexlet
E.FLOAT.F.128.C	Ensemble convert floating-point quad from hexlet ceiling
E.FLOAT.F.128.F	Ensemble convert floating-point quad from hexlet floor
E.FLOAT.F.128.N	Ensemble convert floating-point quad from hexlet nearest
E.FLOAT.F.128.X	Ensemble convert floating-point quad from hexlet exact
E.FLOAT.F.128.Z	Ensemble convert floating-point quad from hexlet zero
E.INFLATE.F.16	Ensemble convert floating-point single from half
E.INFLATE.F.16.X	Ensemble convert floating-point single from half exception
E.INFLATE.F.32	Ensemble convert floating-point double from single
E.INFLATE.F.32.X	Ensemble convert floating-point double from single exception
E.INFLATE.F.64	Ensemble convert floating-point quad from double
E.INFLATE.F.64.X	Ensemble convert floating-point quad from double exception
E.NEG.F.16	Ensemble negate floating-point half
E.NEG.F.16.X	Ensemble negate floating-point half exception
E.NEG.F.32	Ensemble negate floating-point single
E.NEG.F.32.X	Ensemble negate floating-point single exception
E.NEG.F.64	Ensemble negate floating-point double
E.NEG.F.64.X	Ensemble negate floating-point double exception
E.NEG.F.128	Ensemble negate floating-point quad
E.NEG.F.128.X	Ensemble negate floating-point quad exception
E.RECEST.F.16	Ensemble reciprocal estimate floating-point half
E.RECEST.F.16.X	Ensemble reciprocal estimate floating-point half exception
E.RECEST.F.32	Ensemble reciprocal estimate floating-point single
E.RECEST.F.32.X	Ensemble reciprocal estimate floating-point single exception
E.RECEST.F.64	Ensemble reciprocal estimate floating-point double
E.RECEST.F.64.X	Ensemble reciprocal estimate floating-point double exception
E.RECEST.F.128	Ensemble reciprocal estimate floating-point quad
E.RECEST.F.128.X	Ensemble reciprocal estimate floating-point quad exception

FIG. 41A *continued*

E.RSQREST.F.16	Ensemble floating-point reciprocal square root estimate half
E.RSQREST.F.16.X	Ensemble floating-point reciprocal square root estimate half exact
E.RSQREST.F.32	Ensemble floating-point reciprocal square root estimate single
E.RSQREST.F.32.X	Ensemble floating-point reciprocal square root estimate single exact
E.RSQREST.F.64	Ensemble floating-point reciprocal square root estimate double
E.RSQREST.F.64.X	Ensemble floating-point reciprocal square root estimate double exact
E.RSQREST.F.128	Ensemble floating-point reciprocal square root estimate quad
E.RSQREST.F.128.X	Ensemble floating-point reciprocal square root estimate quad exact
E.SINK.F.16	Ensemble convert floating-point doublets from half nearest default
E.SINK.F.16.C	Ensemble convert floating-point doublets from half ceiling
E.SINK.F.16.C.D	Ensemble convert floating-point doublets from half ceiling default
E.SINK.F.16.F	Ensemble convert floating-point doublets from half floor
E.SINK.F.16.F.D	Ensemble convert floating-point doublets from half floor default
E.SINK.F.16.N	Ensemble convert floating-point doublets from half nearest
E.SINK.F.16.X	Ensemble convert floating-point doublets from half exact
E.SINK.F.16.Z	Ensemble convert floating-point doublets from half zero
E.SINK.F.16.Z.D	Ensemble convert floating-point doublets from half zero default
E.SINK.F.32	Ensemble convert floating-point quadlets from single nearest default
E.SINK.F.32.C	Ensemble convert floating-point quadlets from single ceiling
E.SINK.F.32.C.D	Ensemble convert floating-point quadlets from single ceiling default
E.SINK.F.32.F	Ensemble convert floating-point quadlets from single floor
E.SINK.F.32.F.D	Ensemble convert floating-point quadlets from single floor default
E.SINK.F.32.N	Ensemble convert floating-point quadlets from single nearest
E.SINK.F.32.X	Ensemble convert floating-point quadlets from single exact
E.SINK.F.32.Z	Ensemble convert floating-point quadlets from single zero
E.SINK.F.32.Z.D	Ensemble convert floating-point quadlets from single zero default
E.SINK.F.64	Ensemble convert floating-point octlets from double nearest default
E.SINK.F.64.C	Ensemble convert floating-point octlets from double ceiling
E.SINK.F.64.C.D	Ensemble convert floating-point octlets from double ceiling default
E.SINK.F.64.F	Ensemble convert floating-point octlets from double floor
E.SINK.F.64.F.D	Ensemble convert floating-point octlets from double floor default
E.SINK.F.64.N	Ensemble convert floating-point octlets from double nearest
E.SINK.F.64.X	Ensemble convert floating-point octlets from double exact
E.SINK.F.64.Z	Ensemble convert floating-point octlets from double zero
E.SINK.F.64.Z.D	Ensemble convert floating-point octlets from double zero default
E.SINK.F.128	Ensemble convert floating-point hexlet from quad nearest default
E.SINK.F.128.C	Ensemble convert floating-point hexlet from quad ceiling
E.SINK.F.128.C.D	Ensemble convert floating-point hexlet from quad ceiling default
E.SINK.F.128.F	Ensemble convert floating-point hexlet from quad floor
E.SINK.F.128.F.D	Ensemble convert floating-point hexlet from quad floor default

FIG. 41A *continued*

E.SINK.F.128.N	Ensemble convert floating-point hexlet from quad nearest
E.SINK.F.128.X	Ensemble convert floating-point hexlet from quad exact
E.SINK.F.128.Z	Ensemble convert floating-point hexlet from quad zero
E.SINK.F.128.Z.D	Ensemble convert floating-point hexlet from quad zero default
E.SQR.F.16	Ensemble square root floating-point half
E.SQR.F.16.C	Ensemble square root floating-point half ceiling
E.SQR.F.16.F	Ensemble square root floating-point half floor
E.SQR.F.16.N	Ensemble square root floating-point half nearest
E.SQR.F.16.X	Ensemble square root floating-point half exact
E.SQR.F.16.Z	Ensemble square root floating-point half zero
E.SQR.F.32	Ensemble square root floating-point single
E.SQR.F.32.C	Ensemble square root floating-point single ceiling
E.SQR.F.32.F	Ensemble square root floating-point single floor
E.SQR.F.32.N	Ensemble square root floating-point single nearest
E.SQR.F.32.X	Ensemble square root floating-point single exact
E.SQR.F.32.Z	Ensemble square root floating-point single zero
E.SQR.F.64	Ensemble square root floating-point double
E.SQR.F.64.C	Ensemble square root floating-point double ceiling
E.SQR.F.64.F	Ensemble square root floating-point double floor
E.SQR.F.64.N	Ensemble square root floating-point double nearest
E.SQR.F.64.X	Ensemble square root floating-point double exact
E.SQR.F.64.Z	Ensemble square root floating-point double zero
E.SQR.F.128	Ensemble square root floating-point quad
E.SQR.F.128.C	Ensemble square root floating-point quad ceiling
E.SQR.F.128.F	Ensemble square root floating-point quad floor
E.SQR.F.128.N	Ensemble square root floating-point quad nearest
E.SQR.F.128.X	Ensemble square root floating-point quad exact
E.SQR.F.128.Z	Ensemble square root floating-point quad zero
E.SUM.F.16	Ensemble sum floating-point half
E.SUM.F.16.C	Ensemble sum floating-point half ceiling
E.SUM.F.16.F	Ensemble sum floating-point half floor
E.SUM.F.16.N	Ensemble sum floating-point half nearest
E.SUM.F.16.X	Ensemble sum floating-point half exact
E.SUM.F.16.Z	Ensemble sum floating-point half zero
E.SUM.F.32	Ensemble sum floating-point single
E.SUM.F.32.C	Ensemble sum floating-point single ceiling
E.SUM.F.32.F	Ensemble sum floating-point single floor
E.SUM.F.32.N	Ensemble sum floating-point single nearest
E.SUM.F.32.X	Ensemble sum floating-point single exact
E.SUM.F.32.Z	Ensemble sum floating-point single zero

FIG. 41A *continued*

E.SUM.F.64	Ensemble sum floating-point double
E.SUM.F.64.C	Ensemble sum floating-point double ceiling
E.SUM.F.64.F	Ensemble sum floating-point double floor
E.SUM.F.64.N	Ensemble sum floating-point double nearest
E.SUM.F.64.X	Ensemble sum floating-point double exact
E.SUM.F.64.Z	Ensemble sum floating-point double zero
E.SUM.F.128	Ensemble sum floating-point quad
E.SUM.F.128.C	Ensemble sum floating-point quad ceiling
E.SUM.F.128.F	Ensemble sum floating-point quad floor
E.SUM.F.128.N	Ensemble sum floating-point quad nearest
E.SUM.F.128.X	Ensemble sum floating-point quad exact
E.SUM.F.128.Z	Ensemble sum floating-point quad zero

Selection

	op	prec				round/trap
copy	COPY	16	32	64	128	NONE X
absolute value	ABS	16	32	64	128	NONE X
float from integer	FLOAT	16	32	64	128	NONE C F N X Z
integer from float	SINK	16	32	64	128	NONE C F N X Z C.D F.D Z.D
increase format precision	INFLATE	16	32	64		NONE X
decrease format precision	DEFLATE		32	64	128	NONE C F N X Z
negate	NEG	16	32	64	128	NONE X
reciprocal estimate	RECEST	16	32	64	128	NONE X
reciprocal square root estimate	RSQREST	16	32	64	128	NONE X
square root	SQR	16	32	64	128	NONE C F N X Z
sum	SUM	16	32	64	128	NONE C F N X Z

FIG. 41A *continued*

Format

E.op.prec.round rd=rc

rd=eopprecround(rc)

31 24	23 18	17 12	11 6	5 0
E.prec	rd	rc	op	E.UNARY
8	6	6	6	6

FIG. 41B

Definition

```
def EnsembleUnaryFloatingPoint(op,prec,round,rd,rc) as
    c ← RegRead(rc, 128)
    case op of
        E.ABS.F, E.NEG.F, E.SQR.F:
            for i ← 0 to 128-prec by prec
                ci ← F(prec,c_{i+prec-1..i})
                case op of
                    E.ABS.F:
                        ai.t ← ci.t
                        ai.s ← 0
                        ai.e ← ci.e
                        ai.f ← ci.f
                    E.COPY.F:
                        ai ← ci
                    E.NEG.F:
                        ai.t ← ci.t
                        ai.s ← ~ci.s
                        ai.e ← ci.e
                        ai.f ← ci.f
                    E.RECEST.F:
                        ai ← frecest(ci)
                    E.RSQREST.F:
                        ai ← frsqrest(ci)
                    E.SQR.F:
                        ai ← fsqr(ci)
                endcase
                a_{i+prec-1..i} ← PackF(prec, ai, round)
            endfor
        E.SUM.F:
            p[0].t ← NULL
            for i ← 0 to 128-prec by prec
                p[i+prec] ← fadd(p[i], F(prec,c_{i+prec-1..i}))
            endfor
            a ← PackF(prec, p[128], round)
        E.SINK.F:
            for i ← 0 to 128-prec by prec
                ci ← F(prec,c_{i+prec-1..i})
                a_{i+prec-1..i} ← fsinkr(prec, ci, round)
            endfor
        E.FLOAT.F:
            for i ← 0 to 128-prec by prec
                ci.t ← NORM
                ci.e ← 0
                ci.s ← c_{i+prec-1}
                ci.f ← ci.s ? 1+~c_{i+prec-2..i} : c_{i+prec-2..i}
                a_{i+prec-1..i} ← PackF(prec, ci, round)
            endfor
```

FIG. 41C

```
            E.INFLATE.F:
                for i ← 0 to 64-prec by prec
                    ci ← F(prec,c_{i+prec-1..i})
                    a_{i+i+prec+prec-1..i+i} ← PackF(prec+prec, ci, round)
                endfor
            E.DEFLATE.F:
                for i ← 0 to 128-prec by prec
                    ci ← F(prec,c_{i+prec-1..i})
                    a_{i/2+prec/2-1..i/2} ← PackF(prec/2, ci, round)
                endfor
                a_{127..64} ← 0
        endcase
        RegWrite[rd, 128, a]
enddef
```

Exceptions

Floating-point arithmetic

FIG. 41C *continued*

E.MUL.G.8	Ensemble multiply Galois field byte
E.MUL.G.64	Ensemble multiply Galois field octlet

FIG. 42A

Format

E.MUL.G.size ra=rd,rc,rb

ra=emulgsize(rd,rc,rb)

31 ... 24	23 ... 18	17 ... 12	11 ... 6	5 ... 0
E.MUL.G.size	rd	rc	rb	ra
8	6	6	6	6

FIG. 42B

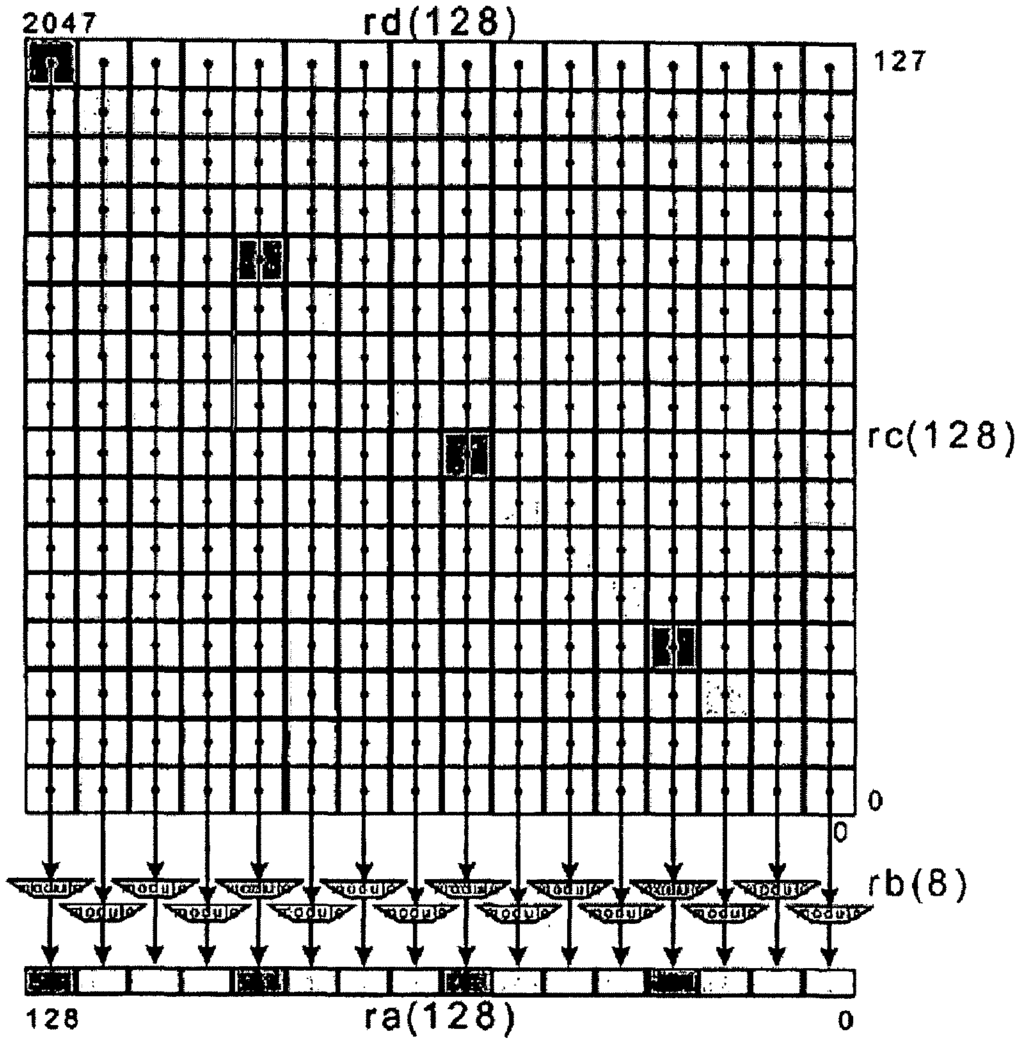
Definition

```
def c ← PolyMultiply(size,a,b) as
    p[0] ← 0^(2*size)
    for k ← 0 to size-1
        p[k+1] ← p[k] ^ a_k ? (0^(size-k) || b || 0^k) : 0^(2*size)
    endfor
    c ← p[size]
enddef

def c ← PolyResidue(size,a,b) as
    p[0] ← a
    for k ← size-1 to 0 by -1
        p[k+1] ← p[k] ^ p[0]_(size+k) ? (0^(size-k) || 1^1 || b || 0^k) : 0^(2*size)
    endfor
    c ← p[size]_(size-1..0)
enddef

def EnsembleTernary(op,size,rd,rc,rb,ra) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case op of
        E.MUL.G:
            for i ← 0 to 128-size by size
                a_(size-1+i..i) ← PolyResidue(size,PolyMul(size,c_(size-1+i..i),b_(size-1+i..i)),d_(size-1+i..i))
            endfor
    endcase
    RegWrite(ra, 128, a)
enddef
```

Exceptions

none

FIG. 42C



Ensemble multiply Galois field bytes

FIG. 42D

X.COMPRESS.2	Crossbar compress signed pecks
X.COMPRESS.4	Crossbar compress signed nibbles
X.COMPRESS.8	Crossbar compress signed bytes
X.COMPRESS.16	Crossbar compress signed doublets
X.COMPRESS.32	Crossbar compress signed quadlets
X.COMPRESS.64	Crossbar compress signed octlets
X.COMPRESS.128	Crossbar compress signed hexlet
X.COMPRESS.U.2	Crossbar compress unsigned pecks
X.COMPRESS.U.4	Crossbar compress unsigned nibbles
X.COMPRESS.U.8	Crossbar compress unsigned bytes
X.COMPRESS.U.16	Crossbar compress unsigned doublets
X.COMPRESS.U.32	Crossbar compress unsigned quadlets
X.COMPRESS.U.64	Crossbar compress unsigned octlets
X.COMPRESS.U.128	Crossbar compress unsigned hexlet
X.EXPAND.2	Crossbar expand signed pecks
X.EXPAND.4	Crossbar expand signed nibbles
X.EXPAND.8	Crossbar expand signed bytes
X.EXPAND.16	Crossbar expand signed doublets
X.EXPAND.32	Crossbar expand signed quadlets
X.EXPAND.64	Crossbar expand signed octlets
X.EXPAND.128	Crossbar expand signed hexlet
X.EXPAND.U.2	Crossbar expand unsigned pecks
X.EXPAND.U.4	Crossbar expand unsigned nibbles
X.EXPAND.U.8	Crossbar expand unsigned bytes
X.EXPAND.U.16	Crossbar expand unsigned doublets
X.EXPAND.U.32	Crossbar expand unsigned quadlets
X.EXPAND.U.64	Crossbar expand unsigned octlets
X.EXPAND.U.128	Crossbar expand unsigned hexlet
X.ROTL.2	Crossbar rotate left pecks
X.ROTL.4	Crossbar rotate left nibbles
X.ROTL.8	Crossbar rotate left bytes
X.ROTL.16	Crossbar rotate left doublets
X.ROTL.32	Crossbar rotate left quadlets
X.ROTL.64	Crossbar rotate left octlets
X.ROTL.128	Crossbar rotate left hexlet
X.ROTR.2	Crossbar rotate right pecks
X.ROTR.4	Crossbar rotate right nibbles
X.ROTR.8	Crossbar rotate right bytes
X.ROTR.16	Crossbar rotate right doublets

FIG. 43A

X.ROTR.64	Crossbar rotate right octlets
X.ROTR.128	Crossbar rotate right hexlet
X.SHL.2	Crossbar shift left pecks
X.SHL.2.O	Crossbar shift left signed pecks check overflow
X.SHL.4	Crossbar shift left nibbles
X.SHL.4.O	Crossbar shift left signed nibbles check overflow
X.SHL.8	Crossbar shift left bytes
X.SHL.8.O	Crossbar shift left signed bytes check overflow
X.SHL.16	Crossbar shift left doublets
X.SHL.16.O	Crossbar shift left signed doublets check overflow
X.SHL.32	Crossbar shift left quadlets
X.SHL.32.O	Crossbar shift left signed quadlets check overflow
X.SHL.64	Crossbar shift left octlets
X.SHL.64.O	Crossbar shift left signed octlets check overflow
X.SHL.128	Crossbar shift left hexlet
X.SHL.128.O	Crossbar shift left signed hexlet check overflow
X.SHL.U.2.O	Crossbar shift left unsigned pecks check overflow
X.SHL.U.4.O	Crossbar shift left unsigned nibbles check overflow
X.SHL.U.8.O	Crossbar shift left unsigned bytes check overflow
X.SHL.U.16.O	Crossbar shift left unsigned doublets check overflow
X.SHL.U.32.O	Crossbar shift left unsigned quadlets check overflow
X.SHL.U.64.O	Crossbar shift left unsigned octlets check overflow
X.SHL.U.128.O	Crossbar shift left unsigned hexlet check overflow
X.SHR.2	Crossbar signed shift right pecks
X.SHR.4	Crossbar signed shift right nibbles
X.SHR.8	Crossbar signed shift right bytes
X.SHR.16	Crossbar signed shift right doublets
X.SHR.32	Crossbar signed shift right quadlets
X.SHR.64	Crossbar signed shift right octlets
X.SHR.128	Crossbar signed shift right hexlet
X.SHR.U.2	Crossbar shift right unsigned pecks
X.SHR.U.4	Crossbar shift right unsigned nibbles
X.SHR.U.8	Crossbar shift right unsigned bytes
X.SHR.U.16	Crossbar shift right unsigned doublets
X.SHR.U.32	Crossbar shift right unsigned quadlets
X.SHR.U.64	Crossbar shift right unsigned octlets
X.SHR.U.128	Crossbar shift right unsigned hexlet

FIG. 43A *continued*

Selection

class	op	size
precision	EXPAND EXPAND.U COMPRESS COMPRESS.U	2 4 8 16 32 64 128
shift	ROTR ROTL SHR SHL SHL.O SHL.U.O SHR.U	2 4 8 16 32 64 128

Format

X.op.size rd=rc,rb

rd=xopsize(rc,rb)

31 25	24	23 18	17 12	11 6	5 2	1 0
XSHIFT	s	rd	rc	rb	op	sz
7	1	6	6	6	4	2

$lsize \leftarrow \log(size)$
$s \leftarrow lsize_2$
$sz \leftarrow lsize_{1..0}$

FIG. 43B

Definition

def Crossbar(op,size,rd,rc,rb)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    shift ← b and (size-1)
    case $op_{5..2} \| 0^2$ of
        X.COMPRESS:
            hsize ← size/2
            for i ← 0 to 64-hsize by hsize
                if shift ≤ hsize then
                    $a_{i+hsize-1..i} \leftarrow c_{i+i+shift+hsize-1..i+i+shift}$
                else
                    $a_{i+hsize-1..i} \leftarrow c_{i+i+size-1}^{shift-hsize} \| c_{i+i+size-1..i+i+shift}$
                endif
            endfor
            $a_{127..64} \leftarrow 0$
        X.COMPRESS.U:
            hsize ← size/2
            for i ← 0 to 64-hsize by hsize
                if shift ≤ hsize then
                    $a_{i+hsize-1..i} \leftarrow c_{i+i+shift+hsize-1..i+i+shift}$
                else
                    $a_{i+hsize-1..i} \leftarrow 0^{shift-hsize} \| c_{i+i+size-1..i+i+shift}$
                endif
            endfor
            $a_{127..64} \leftarrow 0$
        X.EXPAND:
            hsize ← size/2
            for i ← 0 to 64-hsize by hsize
                if shift ≤ hsize then
                    $a_{i+i+size-1..i+i} \leftarrow c_{i+hsize-1}^{hsize-shift} \| c_{i+hsize-1..i} \| 0^{shift}$
                else
                    $a_{i+i+size-1..i+i} \leftarrow c_{i+size-shift-1..i} \| 0^{shift}$
                endif
            endfor
        X.EXPAND.U:
            hsize ← size/2
            for i ← 0 to 64-hsize by hsize
                if shift ≤ hsize then
                    $a_{i+i+size-1..i+i} \leftarrow 0^{hsize-shift} \| c_{i+hsize-1..i} \| 0^{shift}$
                else
                    $a_{i+i+size-1..i+i} \leftarrow c_{i+size-shift-1..i} \| 0^{shift}$
                endif
            endfor
        X.ROTL:
            for i ← 0 to 128-size by size
                $a_{i+size-1..i} \leftarrow c_{i+size-1-shift..i} \| c_{i+size-1..i+size-1-shift}$
            endfor

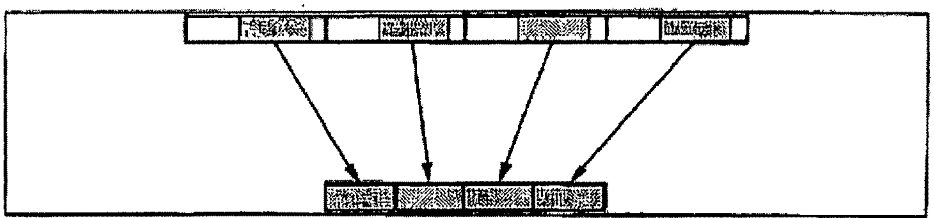
FIG. 43C

```
            X.ROTR:
                for i ← 0 to 128-size by size
                    a_{i+size-1..i} ← c_{i+shift-1..i} || c_{i+size-1..i+shift}
                endfor
            X.SHL:
                for i ← 0 to 128-size by size
                    a_{i+size-1..i} ← c_{i+size-1-shift..i} || 0^{shift}
                endfor
            X.SHL.O:
                for i ← 0 to 128-size by size
                    if c_{i+size-1..i+size-1-shift} ≠ c^{shift+1}_{i+size-1-shift} then
                        raise FixedPointArithmetic
                    endif
                    a_{i+size-1..i} ← c_{i+size-1-shift..i} || 0^{shift}
                endfor
            X.SHL.U.O:
                for i ← 0 to 128-size by size
                    if c_{i+size-1..i+size-shift} ≠ 0^{shift} then
                        raise FixedPointArithmetic
                    endif
                    a_{i+size-1..i} ← c_{i+size-1-shift..i} || 0^{shift}
                endfor
            X.SHR:
                for i ← 0 to 128-size by size
                    a_{i+size-1..i} ← c^{shift}_{i+size-1} || c_{i+size-1..i+shift}
                endfor
            X.SHR.U:
                for i ← 0 to 128-size by size
                    a_{i+size-1..i} ← 0^{shift} || c_{i+size-1..i+shift}
                endfor
        endcase
        RegWrite(rd, 128, a)
enddef
```

Exceptions

Fixed-point arithmetic

FIG. 43C *continued*



Compress 32 bits to 16, with 4-bit right shift

FIG. 43D

Operation codes

X.SHL.M.2	Crossbar shift left merge pecks
X.SHL.M.4	Crossbar shift left merge nibbles
X.SHL.M.8	Crossbar shift left merge bytes
X.SHL.M.16	Crossbar shift left merge doublets
X.SHL.M.32	Crossbar shift left merge quadlets
X.SHL.M.64	Crossbar shift left merge octlets
X.SHL.M.128	Crossbar shift left merge hexlet
X.SHR.M.2	Crossbar shift right merge pecks
X.SHR.M.4	Crossbar shift right merge nibbles
X.SHR.M.8	Crossbar shift right merge bytes
X.SHR.M.16	Crossbar shift right merge doublets
X.SHR.M.32	Crossbar shift right merge quadlets
X.SHR.M.64	Crossbar shift right merge octlets
X.SHR.M.128	Crossbar shift right merge hexlet

FIG. 43E

Format

X.op.size rd@rc,rb

rd=xopsize(rd,rc,rb)

31 25	24	23 18	17 12	11 6	5 2	1 0
XSHIFT	s	rd	rc	rb	op	sz
7	1	6	6	6	4	2

lsize ← log(size)
s ← $lsize_2$
sz ← $lsize_{1..0}$

FIG. 43F

Definition

```
def CrossbarInplace(op,size,rd,rc,rb) as
        d ← RegRead(rd, 128)
        c ← RegRead(rc, 128)
        b ← RegRead(rb, 128)
        shift ← b and (size-1)
        for i ← 0 to 128-size by size
                case op of
                        X.SHR.M:
                                a[i+size-1..i] ← c[i+shift-1..i] || d[i+size-1..i+shift]
                        X.SHL.M:
                                a[i+size-1..i] ← d[i+size-1-shift..i] || c[i+shift-1..i]
        endfor
        RegWrite(rd, 128, a)
enddef
```

Exceptions

none

FIG. 43G

Operation codes

X.COMPRESS.I. 2	Crossbar compress immediate signed pecks
X.COMPRESS.I. 4	Crossbar compress immediate signed nibbles
X.COMPRESS.I. 8	Crossbar compress immediate signed bytes
X.COMPRESS.I. 16	Crossbar compress immediate signed doublets
X.COMPRESS.I. 32	Crossbar compress immediate signed quadlets
X.COMPRESS.I. 64	Crossbar compress immediate signed octlets
X.COMPRESS.I.128	Crossbar compress immediate signed hexlet
X.COMPRESS.I.U. 2	Crossbar compress immediate unsigned pecks
X.COMPRESS.I.U. 4	Crossbar compress immediate unsigned nibbles
X.COMPRESS.I.U. 8	Crossbar compress immediate unsigned bytes
X.COMPRESS.I.U. 16	Crossbar compress immediate unsigned doublets
X.COMPRESS.I.U. 32	Crossbar compress immediate unsigned quadlets
X.COMPRESS.I.U. 64	Crossbar compress immediate unsigned octlets
X.COMPRESS.I.U.128	Crossbar compress immediate unsigned heXlet
X.EXPAND.I. 2	Crossbar expand immediate signed pecks
X.EXPAND.I. 4	Crossbar expand immediate signed nibbles
X.EXPAND.I. 8	Crossbar expand immediate signed bytes
X.EXPAND.I. 16	Crossbar expand immediate signed doublets
X.EXPAND.I. 32	Crossbar expand immediate signed quadlets
X.EXPAND.I. 64	Crossbar expand immediate signed octlets
X.EXPAND.I.128	Crossbar expand immediate signed hexlet
X.EXPAND.I.U. 2	Crossbar expand immediate unsigned pecks
X.EXPAND.I.U. 4	Crossbar expand immediate unsigned nibbles
X.EXPAND.I.U. 8	Crossbar expand immediate unsigned bytes
X.EXPAND.I.U. 16	Crossbar expand immediate unsigned doublets
X.EXPAND.I.U. 32	Crossbar expand immediate unsigned quadlets
X.EXPAND.I.U. 64	Crossbar expand immediate unsigned octlets
X.EXPAND.I.U.128	Crossbar expand immediate unsigned hexlet
X.ROTL.I. 2	Crossbar rotate left immediate pecks
X.ROTL.I. 4	Crossbar rotate left immediate nibbles
X.ROTL.I. 8	Crossbar rotate left immediate bytes
X.ROTL.I. 16	Crossbar rotate left immediate doublets
X.ROTL.I. 32	Crossbar rotate left immediate quadlets
X.ROTL.I. 64	Crossbar rotate left immediate octlets
X.ROTL.I.128	Crossbar rotate left immediate hexlet
X.ROTR.I. 2	Crossbar rotate right immediate pecks
X.ROTR.I. 4	Crossbar rotate right immediate nibbles
X.ROTR.I. 8	Crossbar rotate right immediate bytes
X.ROTR.I. 16	Crossbar rotate right immediate doublets
X.ROTR.I. 32	Crossbar rotate right immediate quadlets
X.ROTR.I. 64	Crossbar rotate right immediate octlets
X.ROTR.I.128	Crossbar rotate right immediate hexlet

FIG. 43H

X.SHL.I. 2	Crossbar shift left immediate pecks
X.SHL.I. 2.O	Crossbar shift left immediate signed pecks check overflow
X.SHL.I. 4	Crossbar shift left immediate nibbles
X.SHL.I. 4.O	Crossbar shift left immediate signed nibbles check overflow
X.SHL.I. 8	Crossbar shift left immediate bytes
X.SHL.I. 8.O	Crossbar shift left immediate signed bytes check overflow
X.SHL.I. 16	Crossbar shift left immediate doublets
X.SHL.I. 16.O	Crossbar shift left immediate signed doublets check overflow
X.SHL.I. 32	Crossbar shift left immediate quadlets
X.SHL.I. 32.O	Crossbar shift left immediate signed quadlets check overflow
X.SHL.I. 64	Crossbar shift left immediate octlets
X.SHL.I. 64.O	Crossbar shift left immediate signed octlets check overflow
X.SHL.I.128	Crossbar shift left immediate hexlet
X.SHL.I.128.O	Crossbar shift left immediate signed hexlet check overflow
X.SHL.I.U. 2.O	Crossbar shift left immediate unsigned pecks check overflow
X.SHL.I.U. 4.O	Crossbar shift left immediate unsigned nibbles check overflow
X.SHL.I.U. 8.O	Crossbar shift left immediate unsigned bytes check overflow
X.SHL.I.U. 16.O	Crossbar shift left immediate unsigned doublets check overflow
X.SHL.I.U. 32.O	Crossbar shift left immediate unsigned quadlets check overflow
X.SHL.I.U. 64.O	Crossbar shift left immediate unsigned octlets check overflow
X.SHL.I.U.128.O	Crossbar shift left immediate unsigned hexlet check overflow
X.SHR.I. 2	Crossbar signed shift right immediate pecks
X.SHR.I. 4	Crossbar signed shift right immediate nibbles
X.SHR.I. 8	Crossbar signed shift right immediate bytes
X.SHR.I. 16	Crossbar signed shift right immediate doublets
X.SHR.I. 32	Crossbar signed shift right immediate quadlets
X.SHR.I. 64	Crossbar signed shift right immediate octlets
X.SHR.I.128	Crossbar signed shift right immediate hexlet
X.SHR.I.U. 2	Crossbar shift right immediate unsigned pecks
X.SHR.I.U. 4	Crossbar shift right immediate unsigned nibbles
X.SHR.I.U. 8	Crossbar shift right immediate unsigned bytes
X.SHR.I.U. 16	Crossbar shift right immediate unsigned doublets
X.SHR.I.U. 32	Crossbar shift right immediate unsigned quadlets
X.SHR.I.U. 64	Crossbar shift right immediate unsigned octlets
X.SHR.I.U.128	Crossbar shift right immediate unsigned hexlet

Equivalencies

X.COPY	Crossbar copy
X.NOP	Crossbar no operation

X.COPY rd=rc	←	X.ROTL.I.128 rd=rc,0
X.NOP	←	X.COPY r0=r0

FIG. 43H *continued*

Redundancies

X.ROTL.I.gsize rd=rc,0	⇔	*X.COPY rd=rc*
X.ROTR.I.gsize rd=rc,0	⇔	*X.COPY rd=rc*
X.ROTR.I.gsize rd=rc,shift	⇔	*X.ROTL.I.gsize rd=rc,gsize-shift*
X.SHL.I.gsize rd=rc,0	⇔	*X.COPY rd=rc*
X.SHL.I.gsize.O rd=rc,0	⇔	*X.COPY rd=rc*
X.SHL.I.U.gsize.O rd=rc,0	⇔	*X.COPY rd=rc*
X.SHR.I.gsize rd=rc,0	⇔	*X.COPY rd=rc*
X.SHR.I.U.gsize rd=rc,0	⇔	*X.COPY rd=rc*

Selection

class	op	size
precision	COMPRESS.I COMPRESS.I.U EXPAND.I EXPAND.I.U	2 4 8 16 32 64 128
shift	ROTL.I ROTR.I SHL.I SHL.I.O SHL.I.U.O SHR.I SHR.I.U	2 4 8 16 32 64 128
copy	*COPY*	

Format

X.op.size rd=rc,shift

rd=xopsize(rc,shift)

31 24	23 18	17 12	11 6	5 0
XSHIFTI	rd	rc	simm	op
8	6	6	6	6

$t \leftarrow 256\text{-}2*\text{size}+\text{shift}$

$op_{1..0} \leftarrow t_{7..6}$

$simm \leftarrow t_{5..0}$

FIG. 43I

Definition

```
def CrossbarShortImmediate(op,rd,rc,simm)
    case (op_{1..0} || simm) of
        0..127:
            size ← 128
        128..191:
            size ← 64
        192..223:
            size ← 32
        224..239:
            size ← 16
        240..247:
            size ← 8
        248..251:
            size ← 4
        252..253:
            size ← 2
        254..255:
            raise ReservedInstruction
    endcase
    shift ← (op_0 || simm) and (size-1)
    c ← RegRead(rc, 128)
    case (op_{5..2} || 0^2) of
        X.COMPRESS.I:
            hsize ← size/2
            for i ← 0 to 64-hsize by hsize
                if shift ≤ hsize then
                    a_{i+hsize-1..i} ← c_{i+i+shift+hsize-1..i+i+shift}
                else
                    a_{i+hsize-1..i} ← c_{i+i+size-1}^{shift-hsize} || c_{i+i+size-1..i+i+shift}
                endif
            endfor
            a_{127..64} ← 0
        X.COMPRESS.I.U:
            hsize ← size/2
            for i ← 0 to 64-hsize by hsize
                if shift ≤ hsize then
                    a_{i+hsize-1..i} ← c_{i+i+shift+hsize-1..i+i+shift}
                else
                    a_{i+hsize-1..i} ← 0^{shift-hsize} || c_{i+i+size-1..i+i+shift}
                endif
            endfor
            a_{127..64} ← 0
```

FIG. 43J

```
X.EXPAND.I:
    hsize ← size/2
    for i ← 0 to 64-hsize by hsize
        if shift ≤ hsize then
            a_{i+i+size-1..i+i} ← c_{i+hsize-1}^{hsize-shift} || c_{i+hsize-1..i} || 0^{shift}
        else
            a_{i+i+size-1..i+i} ← c_{i+size-shift-1..i} || 0^{shift}
        endif
    endfor
X.EXPAND.I.U:
    hsize ← size/2
    for i ← 0 to 64-hsize by hsize
        if shift ≤ hsize then
            a_{i+i+size-1..i+i} ← 0^{hsize-shift} || c_{i+hsize-1..i} || 0^{shift}
        else
            a_{i+i+size-1..i+i} ← c_{i+size-shift-1..i} || 0^{shift}
        endif
    endfor
X.SHL.I:
    for i ← 0 to 128-size by size
        a_{i+size-1..i} ← c_{i+size-1-shift..i} || 0^{shift}
    endfor
X.SHL.I.O:
    for i ← 0 to 128-size by size
        if c_{i+size-1..i+size-1-shift} ≠ c_{i+size-1-shift}^{shift+1} then
            raise FixedPointArithmetic
        endif
        a_{i+size-1..i} ← c_{i+size-1-shift..i} || 0^{shift}
    endfor
X.SHL.I.U.O:
    for i ← 0 to 128-size by size
        if c_{i+size-1..i+size-shift} ≠ 0^{shift} then
            raise FixedPointArithmetic
        endif
        a_{i+size-1..i} ← c_{i+size-1-shift..i} || 0^{shift}
    endfor
```

FIG. 43J *continued*

```
            X.ROTR.I:
                  for i ← 0 to 128-size by size
                        a[i+size-1..i] ← c[i+shift-1..i] || c[i+size-1..i+shift]
                  endfor
            X.SHR.I:
                  for i ← 0 to 128-size by size
                        a[i+size-1..i] ← c[i+size-1]^shift || c[i+size-1..i+shift]
                  endfor
            X.SHR.I.U:
                  for i ← 0 to 128-size by size
                        a[i+size-1..i] ← 0^shift || c[i+size-1..i+shift]
                  endfor
      endcase
      RegWrite(rd, 128, a)
enddef
```

Exceptions

Fixed-point arithmetic
Reserved Instruction

FIG. 43J *continued*

Operation codes

X.SHL.M.I.2	Crossbar shift left merge immediate pecks
X.SHL.M.I.4	Crossbar shift left merge immediate nibbles
X.SHL.M.I.8	Crossbar shift left merge immediate bytes
X.SHL.M.I.16	Crossbar shift left merge immediate doublets
X.SHL.M.I.32	Crossbar shift left merge immediate quadlets
X.SHL.M.I.64	Crossbar shift left merge immediate octlets
X.SHL.M.I.128	Crossbar shift left merge immediate hexlet
X.SHR.M.I.2	Crossbar shift right merge immediate pecks
X.SHR.M.I.4	Crossbar shift right merge immediate nibbles
X.SHR.M.I.8	Crossbar shift right merge immediate bytes
X.SHR.M.I.16	Crossbar shift right merge immediate doublets
X.SHR.M.I.32	Crossbar shift right merge immediate quadlets
X.SHR.M.I.64	Crossbar shift right merge immediate octlets
X.SHR.M.I.128	Crossbar shift right merge immediate hexlet

FIG. 43K

Format

X.op.size rd@rc,shift

rd=xopsize(rc,shift)

31 24	23 18	17 12	11 6	5 0
XSHIFTI	rd	rc	simm	op
8	6	6	6	6

$t \leftarrow 256\text{-}2{*}size{+}shift$

$op_{1..0} \leftarrow t_{7..6}$

$simm \leftarrow t_{5..0}$

FIG. 43L

Definition

```
def CrossbarShortImmediateInplace(op,rd,rc,simm)
    case (op1..0 || simm) of
        0..127:
            size ← 128
        128..191:
            size ← 64
        192..223:
            size ← 32
        224..239:
            size ← 16
        240..247:
            size ← 8
        248..251:
            size ← 4
        252..253:
            size ← 2
        254..255:
            raise ReservedInstruction
    endcase
    shift ← (op0 || simm) and (size-1)
    c ← RegRead(rc, 128)
    d ← RegRead(rd, 128)
    for i ← 0 to 128-size by size
        case (op5..2 || 0^2) of
            X.SHR.M.I:
                a[i+size-1..i] ← c[i+shift-1..i] || d[i+size-1..i+shift]
            X.SHL.M.I:
                a[i+size-1..i] ← d[i+size-1-shift..i] || c[i+shift-1..i]
        endcase
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Reserved Instruction

FIG. 43M

Operation codes

X.EXTRACT	Crossbar extract

Format

X.EXTRACT ra=rd,rc,rb

ra=xextract(rd,rc,rb)

31 24	23 18	17 12	11 6	5 0
op	rd	rc	rb	ra
8	6	6	6	6

FIG. 44A

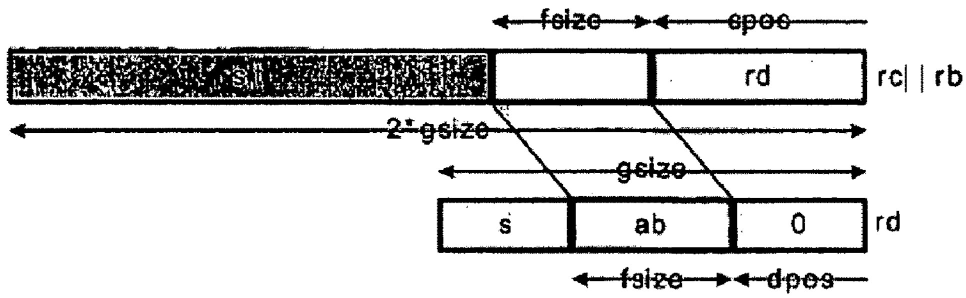
Definition

```
def CrossbarExtract(op,ra,rb,rc,rd) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case b_{8..0} of
        0..255:
            gsize ← 128
        256..383:
            gsize ← 64
        384..447:
            gsize ← 32
        448..479:
            gsize ← 16
        480..495:
            gsize ← 8
        496..503:
            gsize ← 4
        504..507:
            gsize ← 2
        508..511:
            gsize ← 1
    endcase
    m ← b_{12}
    as ← signed ← b_{14}
    h ← (2-m)*gsize
    spos ← (b_{8..0}) and ((2-m)*gsize-1)
    dpos ← (0 || b_{23..16}) and (gsize-1)
    sfsize ← (0 || b_{31..24}) and (gsize-1)
    tfsize ← (sfsize = 0) or ((sfsize+dpos) > gsize) ? gsize-dpos : sfsize
    fsize ← (tfsize + spos > h) ? h - spos : tfsize
    for i ← 0 to 128-gsize by gsize
        case op of
            X.EXTRACT:
                if m then
                    p ← d_{gsize+i-1..i}
                else
                    p ← (d || c)_{2*(gsize+i)-1..2*i}
                endif
        endcase
        v ← (as & p_{h-1})||p
        w ← (as & v_{spos+fsize-1})^{gsize-fsize-dpos} || v_{fsize-1+spos..spos} || 0^{dpos}
        if m then
            a_{size-1+i..i} ← c_{gsize-1+i..dpos+fsize+i} || w_{dpos+fsize-1..dpos} || c_{dpos-1+1..i}
        else
            a_{size-1+i..i} ← w
        endif
    endfor
    RegWrite(ra, 128, a)
enddef
```

Exceptions

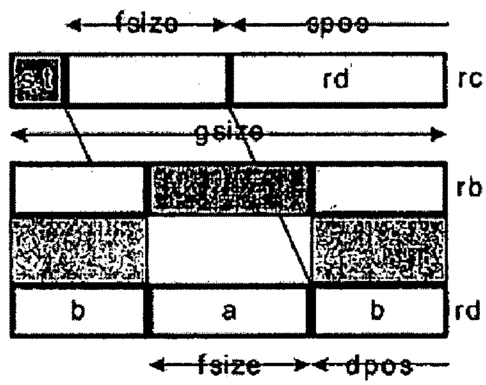
none

FIG. 44B



Crossbar extract

FIG. 44C



Crossbar merge extract

FIG. 44D

Operation codes

E.MUL.X	Ensemble multiply extract
E.EXTRACT	Ensemble extract
E.SCAL.ADD.X	Ensemble scale add extract

FIG. 44E

Format

E.op ra=rd,rc,rb

ra=eop(rd,rc,rb)

31 24	23 18	17 12	11 6	5 0
op	rd	rc	rb	ra
8	6	6	6	6

FIG. 44F

```
def mul(size,h,vs,v,i,ws,w,j) as
    mul ← ((vs&v_{size-1+i})^{h-size} || v_{size-1+i..i}) * ((ws&w_{size-1+j})^{h-size} || w_{size-1+j..j})
enddef

def EnsembleExtract(op,ra,rb,rc,rd) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case b_{8..0} of
        0..255:
            sgsize ← 128
        256..383:
            sgsize ← 64
        384..447:
            sgsize ← 32
        448..479:
            sgsize ← 16
        480..495:
            sgsize ← 8
        496..503:
            sgsize ← 4
        504..507:
            sgsize ← 2
        508..511:
            sgsize ← 1
    endcase
    l ← b_11
    m ← b_12
    n ← b_13
    signed ← b_14
    case op of
        E.EXTRACT:
            gsize ← sgsize
            h ← (2-m)*gsize
            as ← signed
            spos ← (b_{8..0}) and ((2-m)*gsize-1)
        E.SCAL.ADD.X:
            if (sgsize < 8) then
                gsize ← 8
            elseif (sgsize*(n+1) > 32) then
                gsize ← 32/(n+1)
            else
                gsize ← sgsize
            endif
            ds ← cs ← signed
            bs ← signed ^ m
            as ← signed or m or n
            h ← (2*gsize) + 1 + n
            spos ← (b_{8..0}) and (2*gsize-1)
```

FIG. 44G

```
        E.MUL.X:
            if (sgsize < 8) then
                gsize ← 8
            elseif (sgsize*(n+1) > 128) then
                gsize ← 128/(n+1)
            else
                gsize ← sgsize
            endif
            ds ← signed
            cs ← signed ^ m
            as ← signed or m or n
            h ← (2*gsize) + n
            spos ← (b8..0) and (2*gsize-1)
endcase
dpos ← (0 || b23..16) and (gsize-1)
r ← spos
sfsize ← (0 || b31..24) and (gsize-1)
tfsize ← (sfsize = 0) or ((sfsize+dpos) > gsize) ? gsize-dpos : sfsize
fsize ← (tfsize + spos > h) ? h - spos : tfsize
if (b10..9 = Z) and not as then
    rnd ← F
else
    rnd ← b10..9
endif
for i ← 0 to 128-gsize by gsize
    case op of
        E.EXTRACT:
            if m then
                p ← dgsize+i-1..i
            else
                p ← (d || c)2*(gsize+i)-1..2*i
            endif
        E.MUL.X:
            if n then
                if (i and gsize) = 0 then
                    p ← mul(gsize,h,ds,d,i,cs,c,i) - mul(gsize,h,ds,d,i+size,cs,c,i+size)
                else
                    p ← mul(gsize,h,ds,d,i,cs,c,i+size) + mul(gsize,h,ds,d,i,cs,c,i+size)
                endif
            else
                p ← mul(gsize,h,ds,d,i,cs,c,i)
            endif
```

FIG. 44G *continued*

```
                E.SCAL.ADD.X:
                    if n then
                        if (i and gsize) = 0 then
                            p ← mul(gsize,h,ds,d,i,bs,b,64+2*gsize)
                                + mul(gsize,h,cs,c,i,bs,b,64)
                                - mul(gsize,h,ds,d,i+gsize,bs,b,64+3*gsize)
                                - mul(gsize,h,cs,c,i+gsize,bs,b,64+gsize)
                        else
                            p ← mul(gsize,h,ds,d,i,bs,b,64+3*gsize)
                                + mul(gsize,h,cs,c,i,bs,b,64+gsize)
                                + mul(gsize,h,ds,d,i+gsize,bs,b,64+2*gsize)
                                + mul(gsize,h,cs,c,i+gsize,bs,b,64)
                        endif
                    else
                        p ← mul(gsize,h,ds,d,i,bs,b,64+gsize) + mul(gsize,h,cs,c,i,bs,b,64)
                    endif
            endcase
            case rnd of
                N:
                    s ← 0^(h-r) || ~p_r || p_r^(r-1)
                Z:
                    s ← 0^(h-r) || p_(h-1)^r
                F:
                    s ← 0^h
                C:
                    s ← 0^(h-r) || 1^r
            endcase
            v ← ((as & p_(h-1))||p) + (0||s)
            if (v_(h..r+fsize) = (as & v_(r+fsize-1))^(h+1-r-fsize)) or not (l and (op = E.EXTRACT)) then
                w ← (as & v_(r+fsize-1))^(gsize-fsize-dpos) || v_(fsize-1+r..r) || 0^dpos
            else
                w ← (s ? (v_h || ~v_h^(gsize-dpos-1)) : 1^(gsize-dpos)) || 0^dpos
            endif
            if m and (op = E.EXTRACT) then
                a_(size-1+i..i) ← c_(gsize-1+i..dpos+fsize+i) || w_(dpos+fsize-1..dpos) || c_(dpos-1+1..i)
            else
                a_(size-1+i..i) ← w
            endif
        endfor
        RegWrite(ra, 128, a)
enddef
```

Exceptions

none

FIG. 44G *continued*

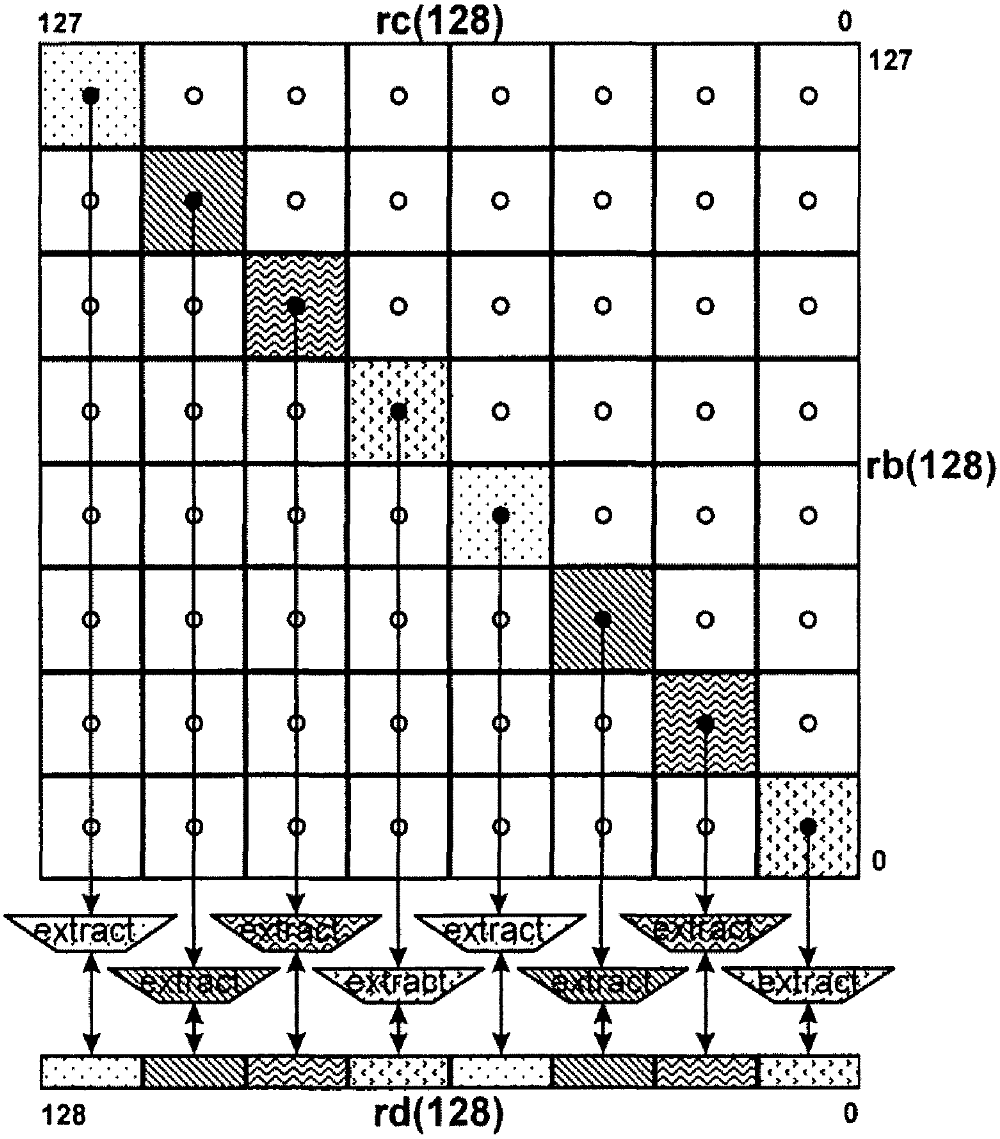
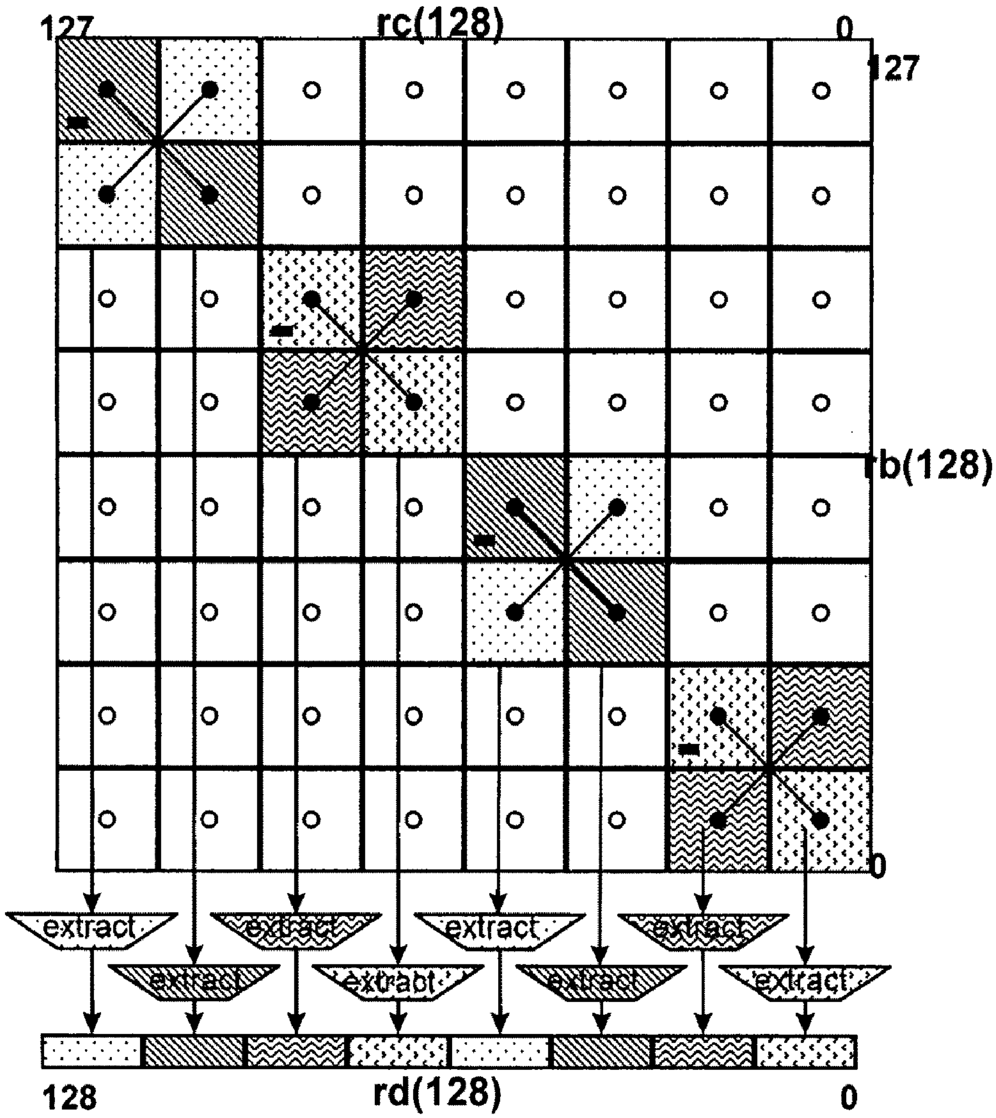
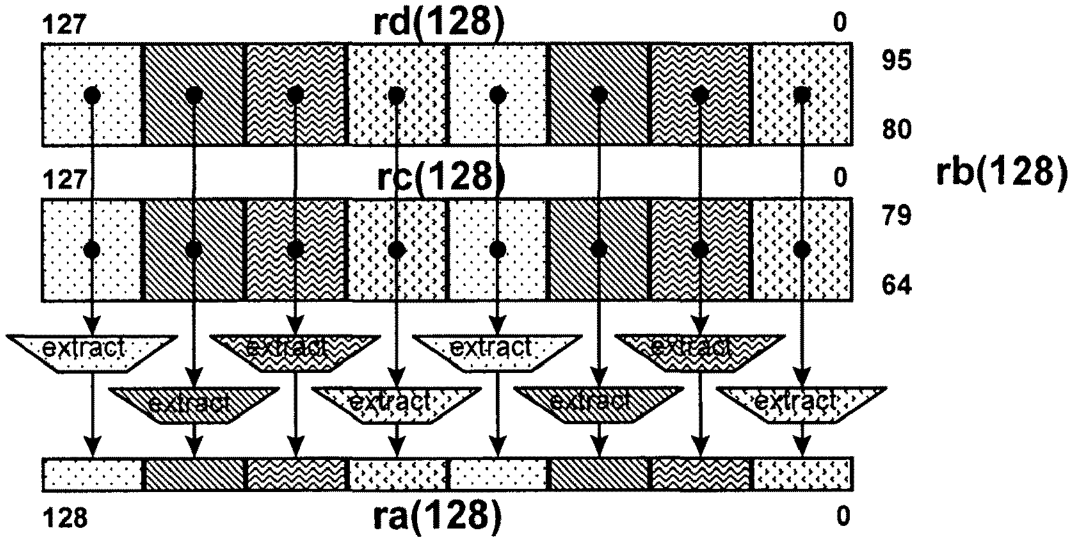

127
rc(128)
0
127
rb(128)
0
extract
extract
extract
extract
extract
extract
extract
extract
128
rd(128)
0

FIG. 44H



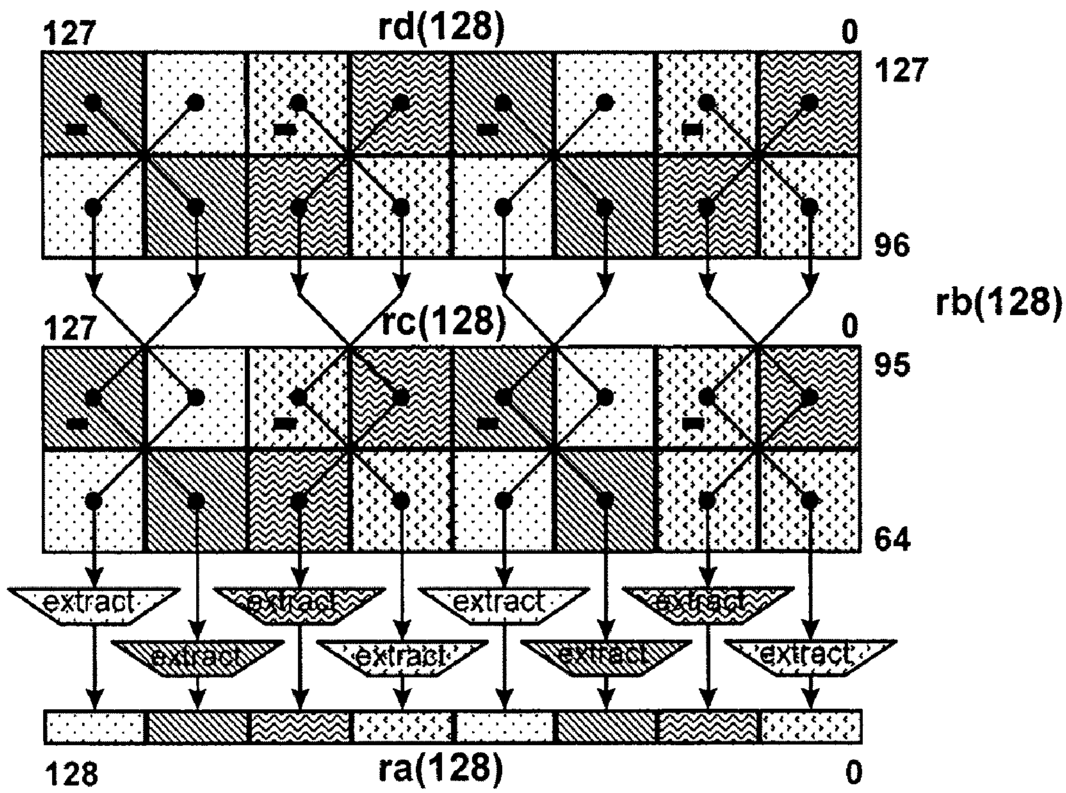
Ensemble complex multiply extract doublets

FIG. 44I



Ensemble scale add extract doublets

FIG. 44J



Ensemble complex scale add extract doublets

FIG. 44K

X.DEPOSIT.2	Crossbar deposit signed pecks
X.DEPOSIT.4	Crossbar deposit signed nibbles
X.DEPOSIT.8	Crossbar deposit signed bytes
X.DEPOSIT.16	Crossbar deposit signed doublets
X.DEPOSIT.32	Crossbar deposit signed quadlets
X.DEPOSIT.64	Crossbar deposit signed octlets
X.DEPOSIT.128	Crossbar deposit signed hexlet
X.DEPOSIT.U.2	Crossbar deposit unsigned pecks
X.DEPOSIT.U.4	Crossbar deposit unsigned nibbles
X.DEPOSIT.U.8	Crossbar deposit unsigned bytes
X.DEPOSIT.U.16	Crossbar deposit unsigned doublets
X.DEPOSIT.U.32	Crossbar deposit unsigned quadlets
X.DEPOSIT.U.64	Crossbar deposit unsigned octlets
X.DEPOSIT.U.128	Crossbar deposit unsigned hexlet
X.WITHDRAW.U.2	Crossbar withdraw unsigned pecks
X.WITHDRAW.U.4	Crossbar withdraw unsigned nibbles
X.WITHDRAW.U.8	Crossbar withdraw unsigned bytes
X.WITHDRAW.U.16	Crossbar withdraw unsigned doublets
X.WITHDRAW.U.32	Crossbar withdraw unsigned quadlets
X.WITHDRAW.U.64	Crossbar withdraw unsigned octlets
X.WITHDRAW.U.128	Crossbar withdraw unsigned hexlet
X.WITHDRAW.2	Crossbar withdraw pecks
X.WITHDRAW.4	Crossbar withdraw nibbles
X.WITHDRAW.8	Crossbar withdraw bytes
X.WITHDRAW.16	Crossbar withdraw doublets
X.WITHDRAW.32	Crossbar withdraw quadlets
X.WITHDRAW.64	Crossbar withdraw octlets
X.WITHDRAW.128	Crossbar withdraw hexlet

FIG. 45A

Equivalencies

X.SEX.I.2	Crossbar extend immediate signed pecks
X.SEX.I.4	Crossbar extend immediate signed nibbles
X.SEX.I.8	Crossbar extend immediate signed bytes
X.SEX.I.16	Crossbar extend immediate signed doublets
X.SEX.I.32	Crossbar extend immediate signed quadlets
X.SEX.I.64	Crossbar extend immediate signed octlets
X.SEX.I.128	Crossbar extend immediate signed hexlet
X.ZEX.I.2	Crossbar extend immediate unsigned pecks
X.ZEX.I.4	Crossbar extend immediate unsigned nibbles
X.ZEX.I.8	Crossbar extend immediate unsigned bytes
X.ZEX.I.16	Crossbar extend immediate unsigned doublets
X.ZEX.I.32	Crossbar extend immediate unsigned quadlets
X.ZEX.I.64	Crossbar extend immediate unsigned octlets
X.ZEX.I.128	Crossbar extend immediate unsigned hexlet

X.SHL.I.gsize rd=rc,i	→	X.DEPOSIT.gsize rd=rc,size-i,i
X.SHR.I.gsize rd=rc,i	→	X.WITHDRAW.gsize rd=rc,size-i,i
X.SHRU.I.gsize rd=rc,i	→	X.WITHDRAW.U.gsize rd=rc,size-i,i
X.SEX.I.gsize rd=rc,i	→	X.DEPOSIT.gsize rd=rc,i,0
X.ZEX.I.gsize rd=rc,i	→	X.DEPOSIT.U.gsize rd=rc,i,0

Redundancies

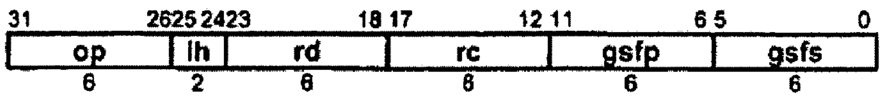
X.DEPOSIT.gsize rd=rc,gsize,0	⇔	X.COPY rd=rc
X.DEPOSIT.U.gsize rd=rc,gsize,0	⇔	X.COPY rd=rc
X.WITHDRAW.gsize rd=rc,gsize,0	⇔	X.COPY rd=rc
X.WITHDRAW.U.gsize rd=rc,gsize,0	⇔	X.COPY rd=rc

FIG. 45A *continued*

Format

X.op.gsize rd=rc,isize,ishift

rd=xopgsize(rc,isize,ishift)



assert isize+ishift ≤ gsize
assert isize≥1
ih_0 || gsfs ← 128-gsize+isize-1
ih_1 || gsfp ← 128-gsize+ishift

FIG. 45B

Definition

```
def CrossbarField(op,rd,rc,gsfp,gsfs) as
    c ← RegRead(rc, 128)
    case ((op_1 || gsfp) and (op_0 || gsfs)) of
        0..63:
            gsize ← 128
        64..95:
            gsize ← 64
        96..111:
            gsize ← 32
        112..119:
            gsize ← 16
        120..123:
            gsize ← 8
        124..125:
            gsize ← 4
        126:
            gsize ← 2
        127:
            raise ReservedInstruction
    endcase
    ishift ← (op_1 || gsfp) and (gsize-1)
    isize ← ((op_0 || gsfs) and (gsize-1))+1
    if (ishift+isize>gsize)
        raise ReservedInstruction
    endif
    case op of
        X.DEPOSIT:
            for i ← 0 to 128-gsize by gsize
                a_{i+gsize-1..i} ← c_{i+isize-1}^{gsize-isize-ishift} || c_{i+isize-1..i} || 0^{ishift}
            endfor
        X.DEPOSIT.U:
            for i ← 0 to 128-gsize by gsize
                a_{i+gsize-1..i} ← 0^{gsize-isize-ishift} || c_{i+isize-1..i} || 0^{ishift}
            endfor
        X.WITHDRAW:
            for i ← 0 to 128-gsize by gsize
                a_{i+gsize-1..i} ← c_{i+isize+ishift-1}^{gsize-isize} || c_{i+isize+ishift-1..i+ishift}
            endfor
        X.WITHDRAW.U:
            for i ← 0 to 128-gsize by gsize
                a_{i+gsize-1..i} ← 0^{gsize-isize} || c_{i+isize+ishift-1..i+ishift}
            endfor
    endcase
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Reserved instruction

FIG. 45C

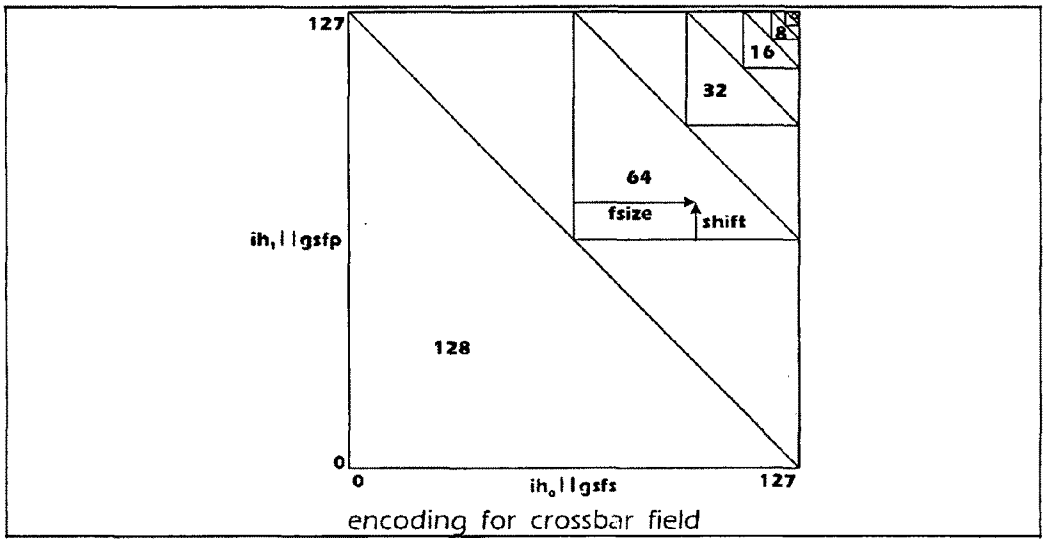


FIG. 45D

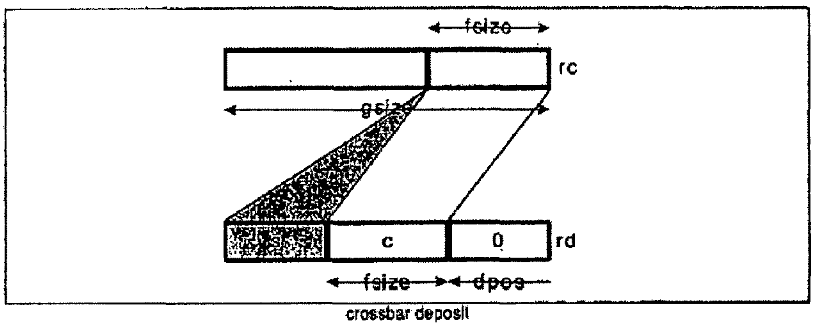


FIG. 45E

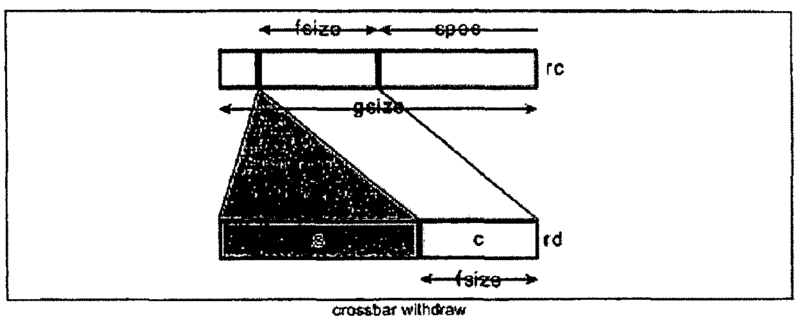


FIG. 45F

Operation codes

X.DEPOSIT.M. 2	Crossbar deposit merge pecks
X.DEPOSIT.M. 4	Crossbar deposit merge nibbles
X.DEPOSIT.M. 8	Crossbar deposit merge bytes
X.DEPOSIT.M. 16	Crossbar deposit merge doublets
X.DEPOSIT.M. 32	Crossbar deposit merge quadlets
X.DEPOSIT.M. 64	Crossbar deposit merge octlets
X.DEPOSIT.M.128	Crossbar deposit merge hexlet

Equivalencies

X.DEPOSIT.M. 1	Crossbar deposit merge bits

X.DEPOSIT.M.1 rd@rc,1,0	→	X.COPY rd=rc

FIG. 45G

Redundancies

X.DEPOSIT.M.gsize rd@rc,gsize,0	⇔	X.COPY rd=rc

Format

X.op.gsize rd@rc,isize,ishift

rd=xopgsize(rd,rc,isize,ishift)

31 26	25 24	23 18	17 12	11 6	5 0
op	ih	rd	rc	gsfp	gsfs
6	2	6	6	6	6

assert isize+ishift ≤ gsize
assert isize≥1
ih_0 || gsfs ← 128-gsize+isize-1
ih_1 || gsfp ← 128-gsize+ishift

FIG. 45H

Definition

```
def CrossbarFieldInplace(op,rd,rc,gsfp,gsfs) as
    c ← RegRead(rc, 128)
    d ← RegRead(rd, 128)
    case ((op1 || gsfp) and (op0 || gsfs)) of
        0..63:
            gsize ← 128
        64..95:
            gsize ← 64
        96..111:
            gsize ← 32
        112..119:
            gsize ← 16
        120..123:
            gsize ← 8
        124..125:
            gsize ← 4
        126:
            gsize ← 2
        127:
            raise ReservedInstruction
    endcase
    ishift ← (op1 || gsfp) and (gsize-1)
    isize ← ((op0 || gsfs) and (gsize-1))+1
    if (ishift+isize>gsize)
        raise ReservedInstruction
    endif
    for i ← 0 to 128-gsize by gsize
        a[i+gsize-1..i] ← d[i+gsize-1..i+isize+ishift] || c[i+isize-1..i] || d[i+ishift-1..i]
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Reserved instruction

FIG. 45I

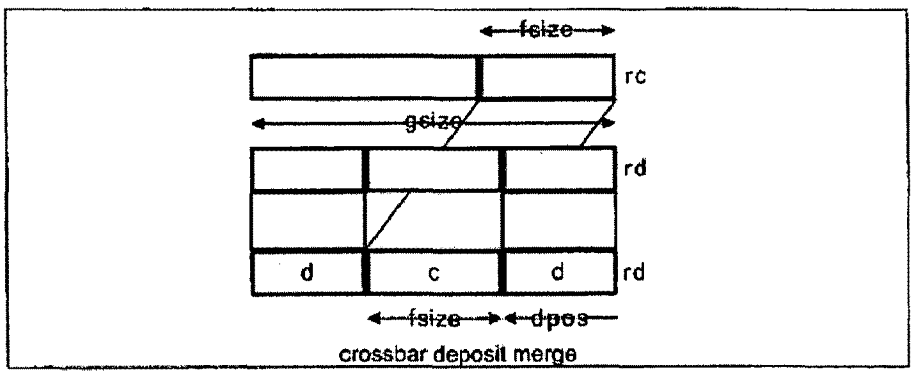


FIG. 45J

X.SHUFFLE.4	Crossbar shuffle within pecks
X.SHUFFLE.8	Crossbar shuffle within bytes
X.SHUFFLE.16	Crossbar shuffle within doublets
X.SHUFFLE.32	Crossbar shuffle within quadlets
X.SHUFFLE.64	Crossbar shuffle within octlets
X.SHUFFLE.128	Crossbar shuffle within hexlet
X.SHUFFLE.256	Crossbar shuffle within triclet

FIG. 46A

Format

X.SHUFFLE.256 rd=rc,rb,v,w,h
X.SHUFFLE.size rd=rcb,v,w

rd=xshuffle256(rc,rb,v,w,h)
rd=xshufflesize(rcb,v,w)

31 … 24	23 … 18	17 … 12	11 … 6	5 … 0
X.SHUFFLE	rd	rc	rb	op
8	6	6	6	6

$rc \leftarrow rb \leftarrow rcb$

$x \leftarrow \log_2(size)$

$y \leftarrow \log_2(v)$

$z \leftarrow \log_2(w)$

$op \leftarrow ((x*x*x-3*x*x-4*x)/6-(z*z-z)/2+x*z+y) + (size=256)*(h*32-56)$

FIG. 46B

Definition

```
def CrossbarShuffle(major,rd,rc,rb,op)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    if rc=rb then
        case op of
            0..55:
                for x ← 2 to 7; for y ← 0 to x-2; for z ← 1 to x-y-1
                    if op = ((x*x*x-3*x*x-4*x)/6-(z*z-z)/2+x*z+y) then
                        for i ← 0 to 127
                            a_i ← c_(i_6..x || i_y+z-1..y || i_x-1..y+z || i_y-1..0)
                        end
                    endif
                endfor; endfor; endfor
            56..63:
                raise ReservedInstruction
        endcase
    elseif
        case op_4..0 of
            0..27:
                cb ← c || b
                x ← 8
                h ← op_5
                for y ← 0 to x-2; for z ← 1 to x-y-1
                    if op_4..0 = ((17*z-z*z)/2-8+y) then
                        for i ← h*128 to 127+h*128
                            a_i-h*128 ← cb_(i_y+z-1..y || i_x-1..y+z || i_y-1..0)
                        end
                    endif
                endfor; endfor
            28..31:
                raise ReservedInstruction
        endcase
    endif
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Reserved Instruction

FIG. 46C

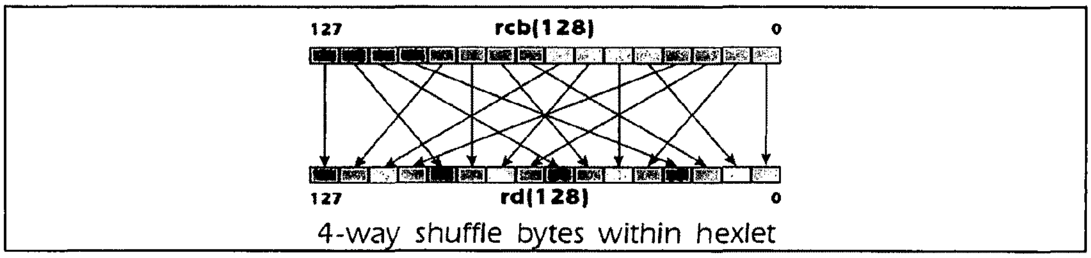


FIG. 46D

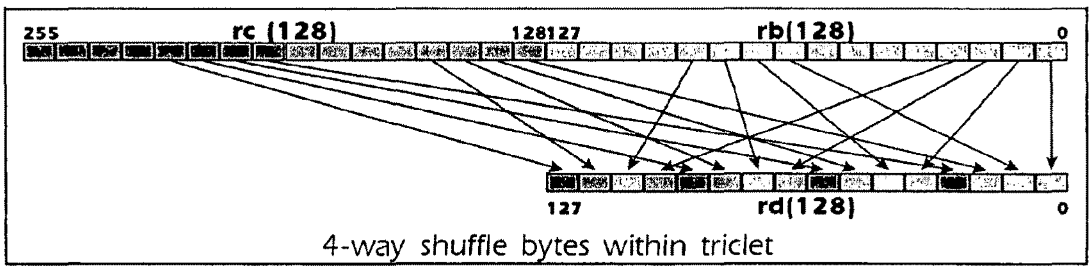


FIG. 46E

Operation codes

X.SWIZZLE	Crossbar swizzle

Format

X.SWIZZLE rd=rc,icopy,iswap

rd=xswizzle(rc,icopy,iswap)

31 26	25 24	23 18	17 12	11 6	5 0
X.SWIZZLE	ih	rd	rc	icopya	iswapa
6	2	6	6	6	6

icopya ← $icopy_{5..0}$
iswapa ← $iswap_{5..0}$
ih ← $icopy_6$ || $iswap_6$

FIG. 47A

Definition

```
def GroupSwizzleImmediate(ih,rd,rc,icopya,iswapa) as
    icopy ← ih1 || icopya
    iswap ← ih0 || iswapa
    c ← RegRead(rc, 128)
    for i ← 0 to 127
        ai ← c(i & icopy) ^ iswap
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

none

FIG. 47B

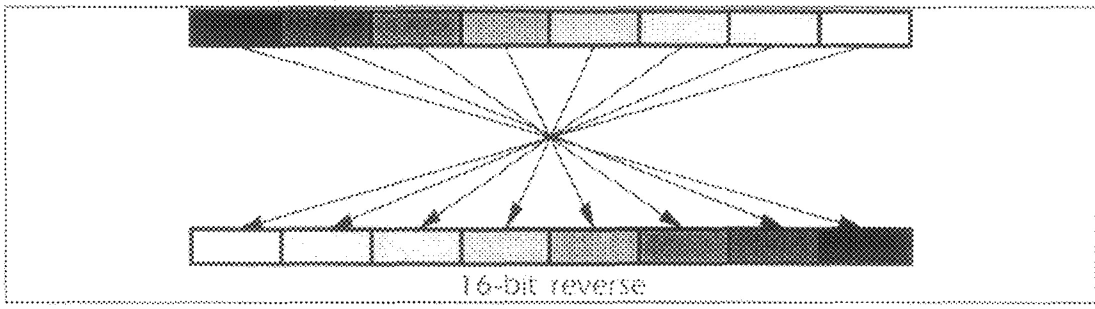


FIG. 47C

X.SELECT.8	Crossbar select bytes

Format

op ra=rd,rc,rb

ra=op(rd,rc,rb)

31 24	23 18	17 12	11 6	5 0
op	rd	rc	rb	ra
8	6	6	6	6

FIG. 47D

Definition

```
def CrossbarTernary(op,rd,rc,rb,ra) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    dc ← d || c
    for i ← 0 to 15
        j ← b8*i+4..8*i
        a8*i+7..8*i ← dc8*j+7..8*j
    endfor
    RegWrite(ra, 128, a)
enddef
```

Exceptions

none

FIG. 47E