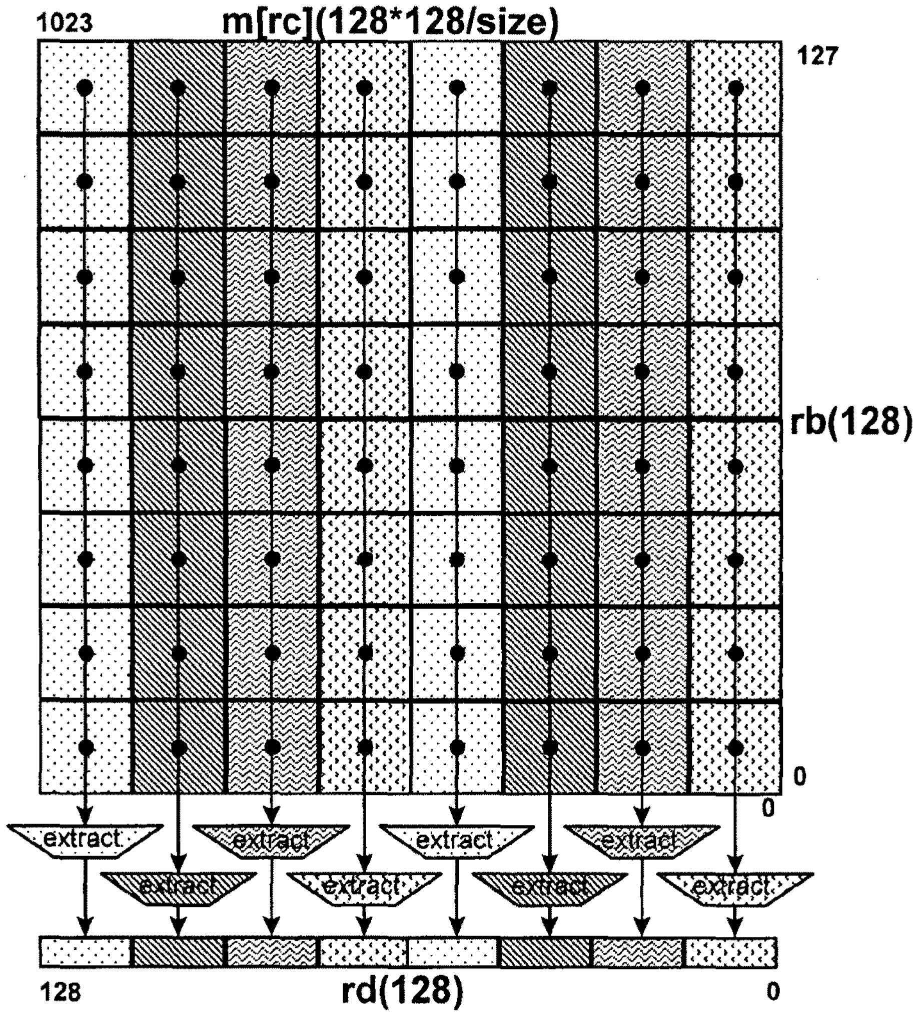
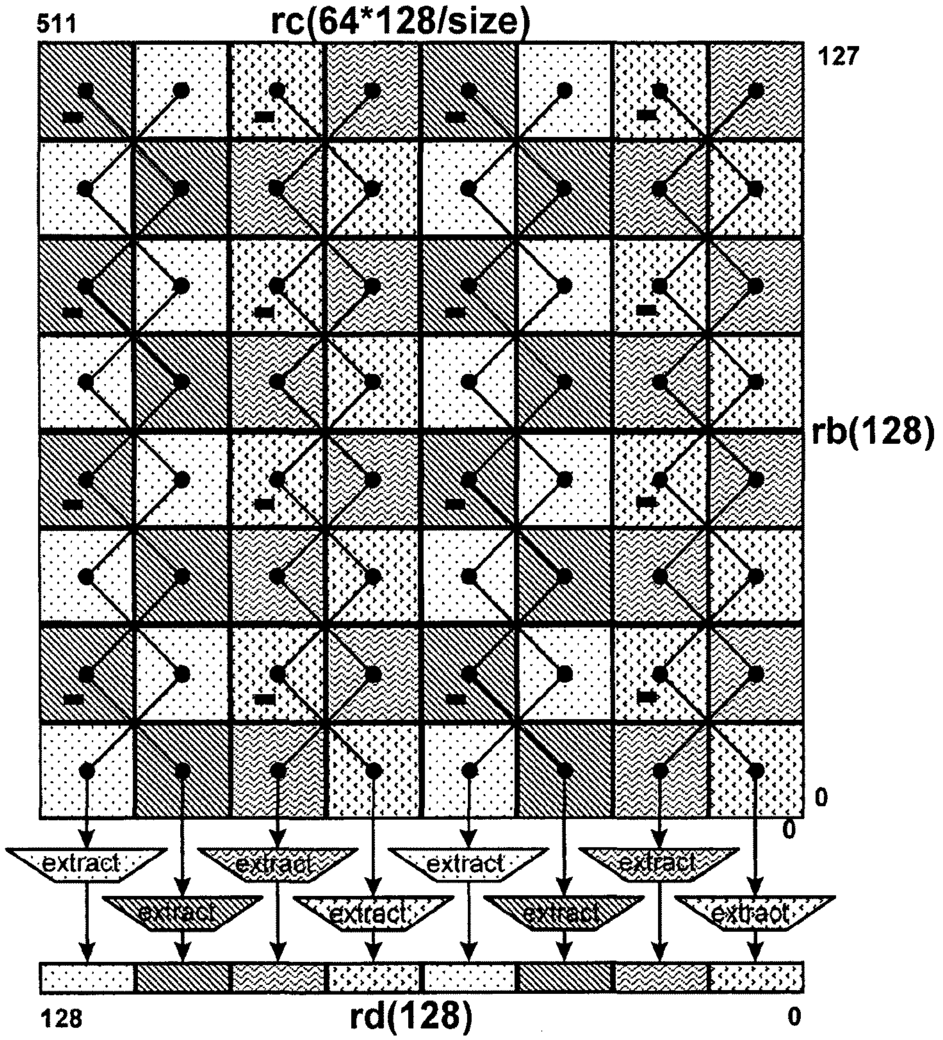
Exhibit U
Part 5



Wide multiply matrix extract immediate doublets

FIG. 102D



Wide multiply matrix extract immediate complex doublets

FIG. 102E

Operation codes

W.MUL.MAT.C.F.16.B	Wide multiply matrix complex floating-point half big-endian
W.MUL.MAT.C.F.16.L	Wide multiply matrix complex floating-point half little-endian
W.MUL.MAT.C.F.32.B	Wide multiply matrix complex floating-point single big-endian
W.MUL.MAT.C.F.32.L	Wide multiply matrix complex floating-point single little-endian
W.MUL.MAT.C.F.64.B	Wide multiply matrix complex floating-point double big-endian
W.MUL.MAT.C.F.64.L	Wide multiply matrix complex floating-point double little-endian
W.MUL.MAT.F.16.B	Wide multiply matrix floating-point half big-endian
W.MUL.MAT.F.16.L	Wide multiply matrix floating-point half little-endian
W.MUL.MAT.F.32.B	Wide multiply matrix floating-point single big-endian
W.MUL.MAT.F.32.L	Wide multiply matrix floating-point single little-endian
W.MUL.MAT.F.64.B	Wide multiply matrix floating-point double big-endian
W.MUL.MAT.F.64.L	Wide multiply matrix floating-point double little-endian

FIG. 103A

Format

M.op.size.order rd=rc,rb

rd=mopsizeorder(rc,rb)

31 24	23 18	17 12	11 6	5 4	3 0
W.MINOR.order	rd	rc	rb	sz	op
8	6	6	6	2	4

FIG. 103B

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 MemoryFloatingPointMultiply(major,op,gsize,rd,rc,rb)
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 128)
    lgsize ← log(gsize)
    switch op of
        W.MUL.MAT.F.16, W.MUL.MAT.F.32, W.MUL.MAT.F.64:
            if $c_{lgsize-4..0} \neq 0$ then
                raise AccessDisallowedByVirtualAddress
            endif
            if $c_{3..lgsize-3} \neq 0$ then
                wsize ← (c and (0-c)) || $0^4$
                t ← c and (c-1)
            else
                wsize ← 128
                t ← c
            endif
            lwsize ← log(wsize)
            if $t_{lwsize+6-lgsize..lwsize-3} \neq 0$ then
                msize ← (t and (0-t)) || $0^4$
                VirtAddr ← t and (t-1)
            else
                msize ← 128*wsize/gsize
                VirtAddr ← t
            endif
            vsize ← msize*gsize/wsize
        W.MUL.MAT.C.F.16, W.MUL.MAT.C.F.32, W.MUL.MAT.C.F.64:
            if $c_{lgsize-4..0} \neq 0$ then
                raise AccessDisallowedByVirtualAddress
            endif
            if $c_{3..lgsize-3} \neq 0$ then
                wsize ← (c and (0-c)) || $0^4$
                t ← c and (c-1)
            else
                wsize ← 128
                t ← c
```

FIG. 103C

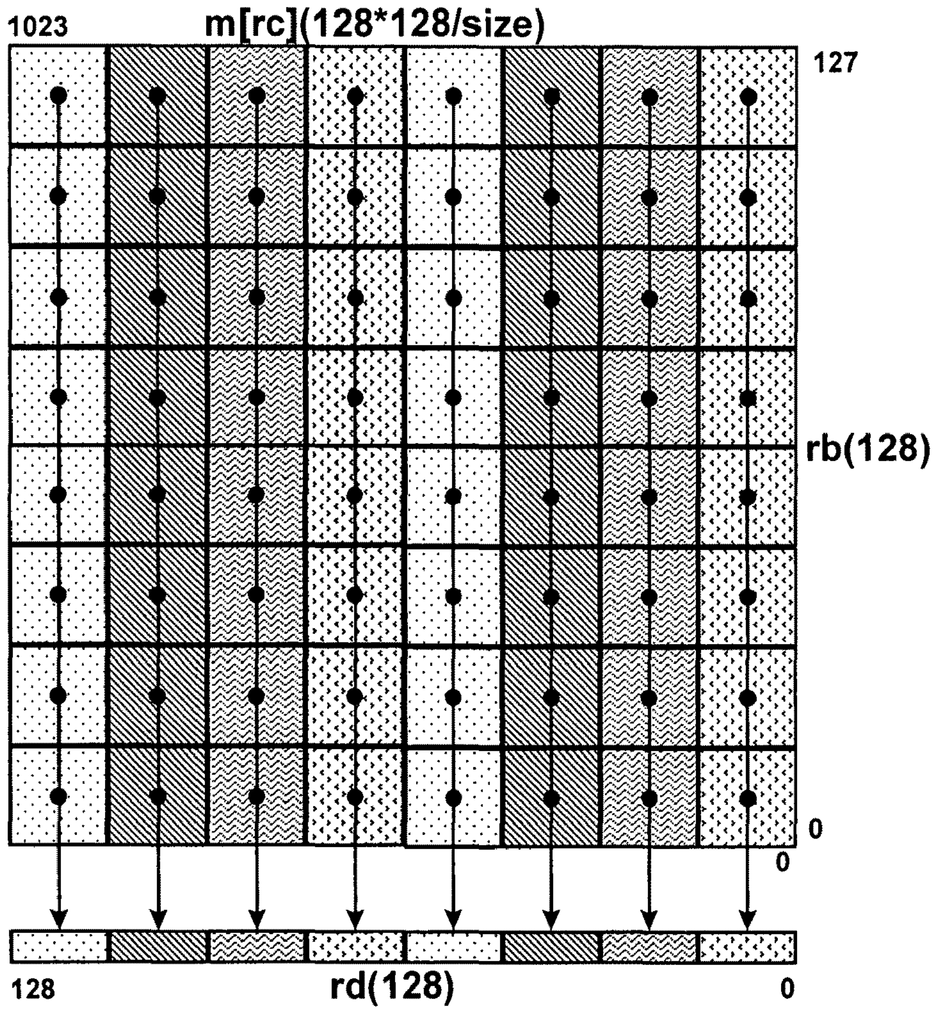
```
                endif
                lwsize ← log(wsize)
                if $t_{lwsize+5-lgsize..lwsize-3}$ ≠ 0 then
                    msize ← (t and (0-t)) || $0^4$
                    VirtAddr ← t and (t-1)
                else
                    msize ← 64*wsize/gsize
                    VirtAddr ← t
                endif
                vsize ← 2*msize*gsize/wsize
        endcase
        case major of
            M.MINOR.B:
                order ← B
            M.MINOR.L:
                order ← L
        endcase
        m ← LoadMemory(c,VirtAddr,msize,order)
        for i ← 0 to wsize-gsize by gsize
            q[0].t ← NULL
            for j ← 0 to vsize-gsize by gsize
                case op of
                    W.MUL.MAT.F.16, W.MUL.MAT.F.32, W.MUL.MAT.F.64:
                        q[j+gsize] ← fadd(q[j], mul(gsize,m,i+wsize*$j_{8..lgsize}$,b,j))
                    W.MUL.MAT.C.F.16, W.MUL.MAT.C.F.32, M.MUL.MAT.C.F.64:
                        if (~i) & j & gsize = 0 then
                            k ← i-(j&gsize)+wsize*$j_{8..lgsize+1}$
                            q[j+gsize] ← fadd(q[j], mul(gsize,m,k,b,j))
                        else
                            k ← i+gsize+wsize*$j_{8..lgsize+1}$
                            q[j+gsize] ← fsub(q[j], mul(gsize,m,k,b,j))
                        endif
                endcase
            endfor
            $a_{gsize-1+i..i}$ ← q[vsize]
        endfor
        $a_{127..wsize}$ ← 0
        RegWrite(rd, 128, a)
    enddef
```

FIG. 103C *continued*

Exceptions

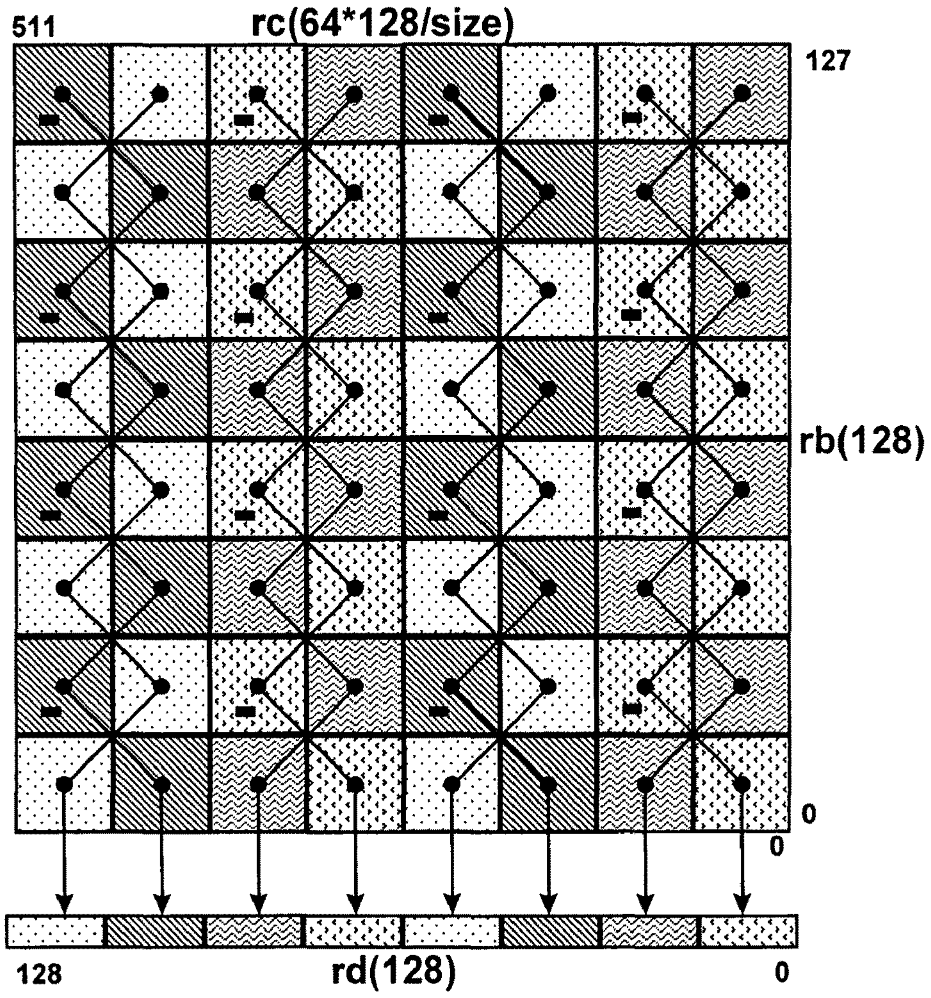
Floating-point arithmetic
Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 103C *continued*



Wide multiply matrix floating-point half

FIG. 103D



Wide multiply matrix complex floating-point half

FIG. 103E

Operation codes

W.MUL.MAT.G.B	Wide multiply matrix Galois big-endian
W.MUL.MAT.G.L	Wide multiply matrix Galois little-endian

FIG. 104A

Format

W.MUL.MAT.G.order ra=rc,rd,rb

ra=mgmorder(rc,rd,rb)

31 24	23 18	17 12	11 6	5 0
W.MULG.order	rd	rc	rb	ra
8	6	6	6	6

FIG. 104B

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 WideMultiplyGalois(op,rd,rc,rb,ra)
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 128)
    gsize ← 8
    lgsize ← log(gsize)
    if $c_{lgsize-4..0} \neq 0$ then
        raise AccessDisallowedByVirtualAddress
    endif
    if $c_{3..lgsize-3} \neq 0$ then
        wsize ← (c and (0-c)) || $0^4$
        t ← c and (c-1)
    else
        wsize ← 128
        t ← c
    endif
    lwsize ← log(wsize)
    if $t_{lwsize+6-lgsize..lwsize-3} \neq 0$ then
        msize ← (t and (0-t)) || $0^4$
        VirtAddr ← t and (t-1)
    else
        msize ← 128*wsize/gsize
        VirtAddr ← t
```

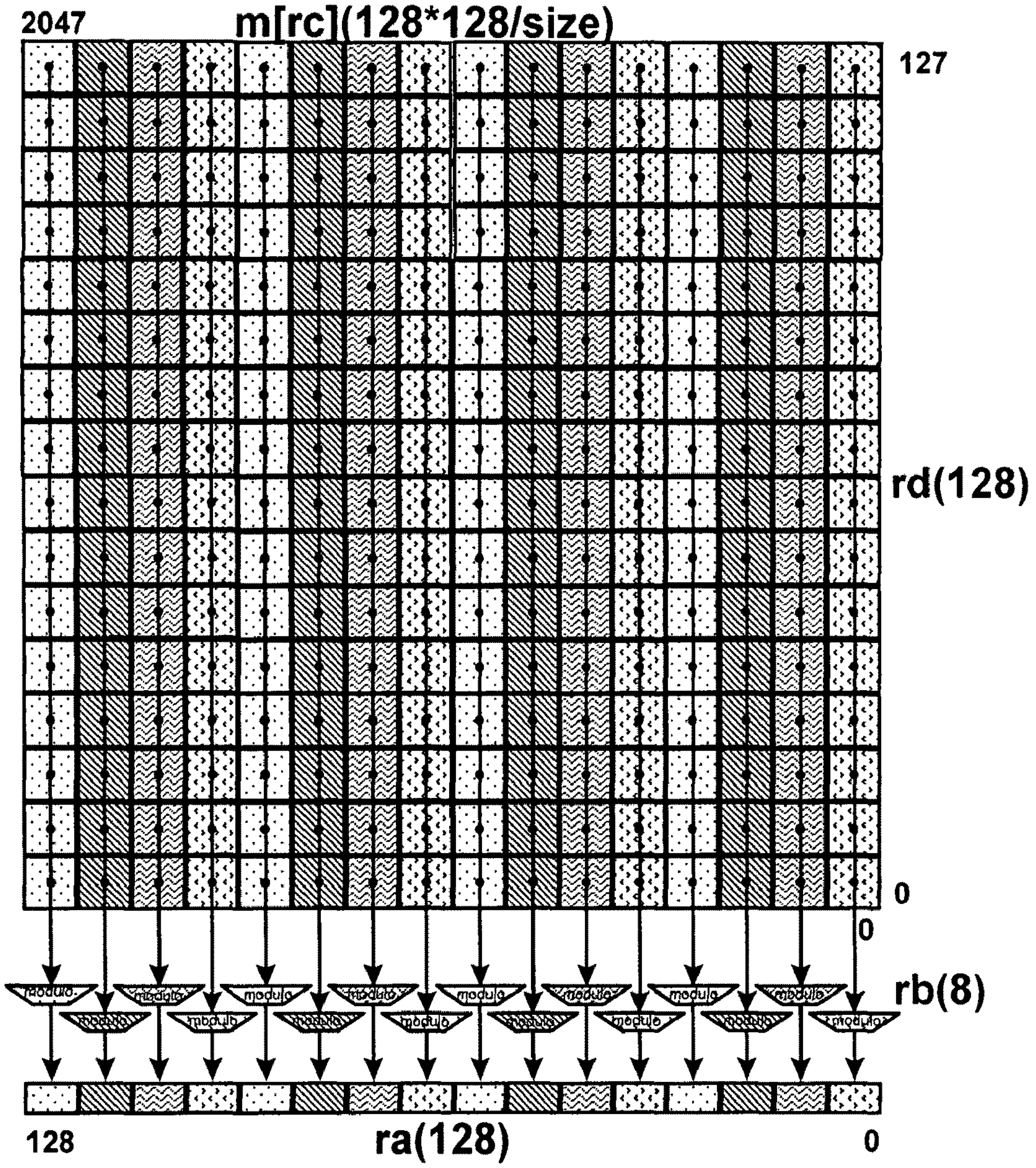
FIG. 104C

```
          endif
          case op of
               W.MUL.MAT.G.B:
                    order ← B
               W.MUL.MAT.G.L:
                    order ← L
          endcase
          m ← LoadMemory(c,VirtAddr,msize,order)
          for i ← 0 to wsize-gsize by gsize
               q[0] ← $0^{2*gsize}$
               for j ← 0 to vsize-gsize by gsize
                    k ← i+wsize*$j_{8..lgsize}$
                    q[j+gsize] ← q[j] ^ PolyMultiply(gsize,$m_{k+gsize-1..k}$,$d_{j+gsize-1..j}$)
               endfor
               $a_{gsize-1+i..i}$ ← PolyResidue(gsize,q[vsize],$b_{gsize-1..0}$)
          endfor
          $a_{127..wsize}$ ← 0
          RegWrite(ra, 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. 104C *continued*



Wide multiply matrix Galois

FIG. 104D

Operation codes

W.SWITCH.B	Wide switch big-endian
W.SWITCH.L	Wide switch little-endian

FIG. 105A

Format

op ra=rc,rd,rb

ra=op(rc,rd,rb)

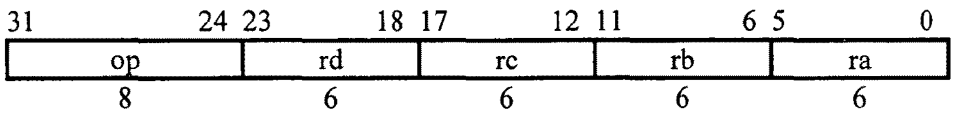


FIG. 105B

Definition

```
def WideSwitch(op,rd,rc,rb,ra)
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 128)
    if $c_{1..0} \neq 0$ then
        raise AccessDisallowedByVirtualAddress
    elseif $c_{6..0} \neq 0$ then
        VirtAddr ← c and (c-1)
        w ← wsize ← (c and (0-c)) || $0^1$
    else
        VirtAddr ← c
        w ← wsize ← 128
    endif
    msize ← 8*wsize
    lwsize ← log(wsize)
    case op of
        W.SWITCH.B:
            order ← B
        W.SWITCH.L:
            order ← L
    endcase
    m ← LoadMemory(c,VirtAddr,msize,order)
    db ← d || b
    for i ← 0 to 127
        j ← 0 || $i_{lwsize-1..0}$
        k ← $m_{7*w+j} || m_{6*w+j} || m_{5*w+j} || m_{4*w+j} || m_{3*w+j} || m_{2*w+j} || m_{w+j} || m_j$
        l ← $i_{7..lwsize} || j_{lwsize-1..0}$
        $a_i$ ← $db_l$
    endfor
    RegWrite(ra, 128, a)
enddef
```

FIG. 105C

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. 105C *continued*

Operation codes

W.TRANSLATE. 8.B	Wide translate bytes big-endian
W.TRANSLATE.16.B	Wide translate doublets big-endian
W.TRANSLATE.32.B	Wide translate quadlets big-endian
W.TRANSLATE.64.B	Wide translate octlets big-endian
W.TRANSLATE. 8.L	Wide translate bytes little-endian
W.TRANSLATE.16.L	Wide translate doublets little-endian
W.TRANSLATE.32.L	Wide translate quadlets little-endian
W.TRANSLATE.64.L	Wide translate octlets little-endian

FIG. 106A

Format

W.TRANSLATE.size.order rd=rc,rb

rd=wtranslatesizeorder(rc,rb)

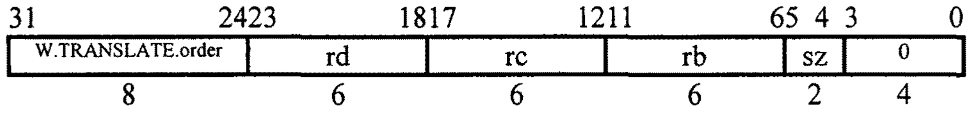


FIG. 106B

Definition

```
def WideTranslate(op,gsize,rd,rc,rb)
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 128)
    lgsize ← log(gsize)
    if $c_{lgsize-4..0}$ ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    endif
    if $c_{4..lgsize-3}$ ≠ 0 then
        wsize ← (c and (0-c)) || $0^3$
        t ← c and (c-1)
    else
        wsize ← 128
        t ← c
    endif
    lwsize ← log(wsize)
    if $t_{lwsize+4..lwsize-2}$ ≠ 0 then
        msize ← (t and (0-t)) || $0^4$
        VirtAddr ← t and (t-1)
    else
        msize ← 256*wsize
        VirtAddr ← t
    endif
    case op of
        W.TRANSLATE.B:
            order ← B
        W.TRANSLATE.L:
            order ← L
    endcase
    m ← LoadMemory(c,VirtAddr,msize,order)
    vsize ← msize/wsize
    lvsize ← log(vsize)
    for i ← 0 to 128-gsize by gsize
        j ← $((order=B)^{lvsize}$^$(b_{lvsize-1+i..i}))$*wsize+$i_{lwsize-1..0}$
        $a_{gsize-1+i..i}$ ← $m_{j+gsize-1..j}$
    endfor
    RegWrite(rd, 128, a)
enddef
```

FIG. 106C

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. 106C *continued*

METHOD AND SOFTWARE FOR GROUP FLOATING-POINT ARITHMETIC OPERATIONS

CROSS-REFERENCES TO RELATED APPLICATIONS

This application is a continuation of U.S. patent application Ser. No. 10/436,340, filed May 13, 2003 now U.S. Pat. No. 7,516,308, which is a continuation of U.S. patent application Ser. No. 09/534,745, filed Mar. 24, 2000, now U.S. Pat. No. 6,643,765, which is a continuation of U.S. patent application Ser. No. 09/382,402, filed Aug. 24, 1999, now U.S. Pat. No. 6,295,599, and which is a continuation-in-part of U.S. patent application Ser. No. 09/169,963, filed Oct. 13, 1998, now U.S. Pat. No. 6,006,318, which is a continuation of U.S. patent application Ser. No. 08/754,827, filed Nov. 22, 1996, now U.S. Pat. No. 5,822,603, which is a division of U.S. patent application Ser. No. 08/516,036, filed Aug. 16, 1995, now U.S. Pat. No. 5,742,840.

This application is a continuation of U.S. patent application Ser. No. 11/511,466, filed Aug. 29, 2006 now abandoned, which is a continuation of U.S. patent application Ser. No. 10/646,787, filed Aug. 25, 2003, now U.S. Pat. No. 7,216,217, which is a continuation of U.S. patent application Ser. No. 09/922,319, filed Aug. 2, 2001 now U.S. Pat. No. 6,725,356, which is a continuation of U.S. patent application Ser. No. 09/382,402, filed Aug. 24, 1999, now U.S. Pat. No. 6,295,599, which claims the benefit of priority to Provisional Application No. 60/097,635 filed Aug. 24, 1998, and is a continuation-in-part of U.S. patent application Ser. No. 09/169,963, filed Oct. 13, 1998, now U.S. Pat. No. 6,006,318, which is a continuation of U.S. patent application Ser. No. 08/754,827, filed Nov. 22, 1996 now U.S. Pat. No. 5,822,603, which is a divisional of U.S. patent application Ser. No. 08/516,036, filed Aug. 16, 1995 now U.S. Pat. No. 5,742,840.

The contents of all the U.S. patent applications and provisional applications listed above are hereby incorporated by reference including their appendices in their entirety.

FIELD OF THE INVENTION

The present invention relates to general purpose processor architectures, and particularly relates to general purpose processor architectures capable of executing group operations.

BACKGROUND OF THE INVENTION

The performance level of a processor, and particularly a general purpose processor, can be estimated from the multiple of a plurality of interdependent factors: clock rate, gates per clock, number of operands, operand and data path width, and operand and data path partitioning. Clock rate is largely influenced by the choice of circuit and logic technology, but is also influenced by the number of gates per clock. Gates per clock is how many gates in a pipeline may change state in a single clock cycle. This can be reduced by inserting latches into the data path: when the number of gates between latches is reduced, a higher clock is possible. However, the additional latches produce a longer pipeline length, and thus come at a cost of increased instruction latency. The number of operands is straightforward; for example, by adding with carry-save techniques, three values may be added together with little more delay than is required for adding two values. Operand and data path width defines how much data can be processed at once; wider data paths can perform more complex functions, but generally this comes at a higher implementation

cost. Operand and data path partitioning refers to the efficient use of the data path as width is increased, with the objective of maintaining substantially peak usage.

SUMMARY OF THE INVENTION

Embodiments of the invention pertain to systems and methods for enhancing the utilization of a general purpose processor by adding classes of instructions. These classes of instructions use the contents of general purpose registers as data path sources, partition the operands into symbols of a specified size, perform operations in parallel, catenate the results and place the catenated results into a general-purpose register. Some embodiments of the invention relate to a general purpose microprocessor which has been optimized for processing and transmitting media data streams through significant parallelism.

Some embodiments of the present invention provide a system and method for improving the performance of general purpose processors by including the capability to execute group operations involving multiple floating-point operands. In one embodiment, a programmable media processor comprises a virtual memory addressing unit, a data path, a register file comprising a plurality of registers coupled to the data path, and an execution unit coupled to the data path capable of executing group-floating point operations in which multiple floating-point operations stored in partitioned fields of one or more of the plurality of registers are operated on to produce catenated results. The group floating-point operations may involve operating on at least two of the multiple floating-point operands in parallel. The catenated results may be returned to a register, and general purpose registers may used as operand and result registers for the floating-point operations. In some embodiments the execution unit may also be capable of performing group floating-point operations on floating-point data of more than one precision. In some embodiments the group floating-point operations may include group add, group subtract, group compare, group multiply and group divide arithmetic operations that operate on catenated floating-point data. In some embodiments, the group floating-point operations may include group multiply-add, group scale-add, and group set operations that operate on catenated floating-point data.

In one embodiment, the execution unit is also capable of executing group integer instructions involving multiple integer operands stored in partitioned fields of registers. The group integer operations may involve operating on at least two of the multiple integer operands in parallel. The group integer operations may include group add, group subtract, group compare, and group multiply arithmetic operations that operate on catenated integer data.

In one embodiment, the execution unit is capable of performing group data handling operations, including operations that copy, operations that shift, operations that rearrange and operations that resize catenated integer data stored in a register and return catenated results. The execution unit may also be configurable to perform group data handling operations on integer data having a symbol width of 8 bits, group data handling operations on integer data having a symbol width of 16 bits, and group data handling operations on integer data having a symbol width of 32 bits. In one embodiment, the operations are controlled by values in a register operand. In one embodiment, the operations are controlled by values in the instruction.

In one embodiment, the multi-precision execution unit is capable of executing a Galois field instruction operation.

In one embodiment, the multi-precision execution unit is configurable to execute a plurality of instruction streams in parallel from a plurality of threads, and the programmable media processor further comprises a register file associated with each thread executing in parallel on the multi-precision execution unit to support processing of the plurality of threads. In some embodiments, the multi-precision execution unit executes instructions from the plurality of instruction streams in a round-robin manner. In some embodiments, the processor ensures only one thread from the plurality of threads can handle an exception at any given time.

Some embodiments of the present invention provide a multiplier array that is fully used for high precision arithmetic, but is only partly used for other, lower precision operations. This can be accomplished by extracting the high-order portion of the multiplier product or sum of products, adjusted by a dynamic shift amount from a general register or an adjustment specified as part of the instruction, and rounded by a control value from a register or instruction portion. The rounding may be any of several types, including round-to-nearest/even; toward zero, floor, or ceiling. Overflows are typically handled by limiting the result to the largest and smallest values that can be accurately represented in the output result.

When an extract is controlled by a register, the size of the result can be specified, allowing rounding and limiting to a smaller number of bits than can fit in the result. This permits the result to be scaled for use in subsequent operations without concern of overflow or rounding. As a result, performance is enhanced. In those instances where the extract is controlled by a register, a single register value defines the size of the operands, the shift amount and size of the result, and the rounding control. By placing such control information in a single register, the size of the instruction is reduced over the number of bits that such an instruction would otherwise require, again improving performance and enhancing processor flexibility. Exemplary instructions are Ensemble Convolve Extract, Ensemble Multiply Extract, Ensemble Multiply Add Extract, and Ensemble Scale Add Extract. With particular regard to the Ensemble Scale Add Extract Instruction, the extract control information is combined in a register with two values used as scalar multipliers to the contents of two vector multiplicands. This combination reduces the number of registers otherwise required, thus reducing the number of bits required for the instruction.

In one embodiment, the processor performs load and store instructions operable to move values between registers and memory. In one embodiment, the processor performs both instructions that verify alignment of memory operands and instructions that permit memory operands to be unaligned. In one embodiment, the processor performs store multiplex instructions operable to move to memory a portion of data contents controlled by a corresponding mask contents. In one embodiment, this masked storage operation is performed by indivisibly reading-modifying-writing a memory operand.

In one embodiment, all processor, memory and interface resources are directly accessible to high-level language programs. In one embodiment, assembler codes and high-level language formats are specified to access enhanced instructions. In one embodiment interface and system state is memory mapped, so that it can be manipulated by compiled code. In one embodiment, software libraries provide other operations required by the ANSI/IEEE floating-point standard. In one embodiment, software conventions are employed at software module boundaries, in order to permit the combination of separately compiled code and to provide standard

interfaces between application, library and system software. In one embodiment, instruction scheduling is performed by a compiler.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **29** illustrates the registers GTBLast, GTBFirst, and GTBBump.

FIG. **30** illustrates a definition for AccessPhysicalGTBRegisters.

FIGS. **31**A-**31**C illustrate Group Boolean instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **31**D-**31**E illustrate Group Multiplex instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **32**A-**32**C illustrate Group Add instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **33**A-**33**C illustrate Group Subtract and Group Set instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **34**A-**34**C illustrate Ensemble Divide and Ensemble Multiply instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **35**A-**35**C illustrate Group Compare instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **36**A-**36**C illustrate Ensemble Unary instructions in accordance with an exemplary embodiment of the present invention.

FIG. **37** illustrates exemplary functions that are defined for use within the detailed instruction definitions in other sections.

FIGS. **38**A-**38**C illustrate Ensemble Floating-Point Add, Ensemble Floating-Point Divide, and Ensemble Floating-Point Multiply instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **38**D-**38**F illustrate Ensemble Floating-Point Multiply Add instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **38**G-**38**I illustrate Ensemble Floating-Point Scale Add instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **39**A-**39**C illustrate Ensemble Floating-Point Subtract instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **39**D-**39**G illustrate Group Set Floating-point instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **40**A-**40**C illustrate Group Compare Floating-point instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **41**A-**41**C illustrate Ensemble Unary Floating-point instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **42**A-**42**D illustrate Ensemble Multiply Galois Field instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **43**A-**43**D illustrate Compress, Expand, Rotate, and Shift instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **43**E-**43**G illustrate Shift Merge instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **43**H-**43**J illustrate Compress Immediate, Expand Immediate, Rotate Immediate, and Shift Immediate instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **43**K-**43**M illustrate Shift Merge Immediate instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **44**A-**44**D illustrate Crossbar Extract instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **44**E-**44**K illustrate Ensemble Extract instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **45**A-**45**F illustrate Deposit and Withdraw instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **45**G-**45**J illustrate Deposit Merge instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **46**A-**46**E illustrate Shuffle instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **47**A-**47**C illustrate Swizzle instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **47**D-**47**E illustrate Select instructions in accordance with an exemplary embodiment of the present invention.

FIG. **48** is a pin summary describing the functions of various pins in accordance with the one embodiment of the present invention.

FIGS. **49**A-**49**G present electrical specifications describing AC and DC parameters in accordance with one embodiment of the present invention.

FIGS. **50**A-**50**C illustrate Load instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **51**A-**51**C illustrate Load Immediate instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **52**A-**52**C illustrate Store and Store Multiplex instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **53**A-**53**C illustrate Store Immediate and Store Multiplex Immediate instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **54**A-**54**E illustrate Data-Handling Operations in accordance with an exemplary embodiment of the present invention.

FIG. **54**F illustrates Procedure Calling Conventions in accordance with an exemplary embodiment of the present invention.

FIG. **54**G illustrates alignment within the dp region in accordance with an exemplary embodiment of the present invention.

FIG. **54**H illustrates gateway with pointers to code and data spaces in accordance with an exemplary embodiment of the present invention.

FIGS. **55**-**56** illustrate an expected rate at which memory requests are serviced in accordance with an exemplary embodiment of the present invention.

FIG. **57** is a pinout diagram in accordance with an exemplary embodiment of the present invention.

FIGS. **58**A-**58**C illustrate Always Reserved instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **59**A-**59**C illustrate Address instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **60**A-**60**C illustrate Address Compare instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **61**A-**61**C illustrate Address Copy Immediate instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **62**A-**62**C illustrate Address Immediate instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **63**A-**63**C illustrate Address Immediate Reversed instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **64**A-**64**C illustrate Address Reversed instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **65**A-**65**C illustrate Address Shift Left Immediate Add instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **66**A-**66**C illustrate Address Shift Left Immediate Subtract instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **67**A-**67**C illustrate Address Shift Left Immediate instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **68**A-**68**C illustrate Address Ternary instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **69**A-**69**C illustrate Branch instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **70**A-**70**C illustrate Branch Back instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **71**A-**71**C illustrate Branch Barrier instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **72**A-**72**C illustrate Branch Conditional instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **73**A-**73**C illustrate Branch Conditional Floating-Point instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **74**A-**74**C illustrate Branch Conditional Visibility Floating-Point instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **75**A-**75**C illustrate Branch Down instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **76**A-**76**C illustrate Branch Gateway instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **77**A-**77**C illustrate Branch Halt instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **78**A-**78**C illustrate Branch Hint instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **79**A-**79**C illustrate Branch Hint Immediate instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **80**A-**80**C illustrate Branch Immediate instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **81**A-**81**C illustrate Branch Immediate Link instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **82**A-**82**C illustrate Branch Link instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **83**A-**83**C illustrate Store Double Compare Swap instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **84**A-**84**C illustrate Store Immediate Inplace instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **85**A-**85**C illustrate Store Inplace instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **86**A-**86**C illustrate Group Add Halve instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **87**A-**87**C illustrate Group Copy Immediate instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **88**A-**88**C illustrate Group Immediate instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **89**A-**89**C illustrate Group Immediate Reversed instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **90**A-**90**C illustrate Group Inplace instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **91**A-**91**C illustrate Group Shift Left Immediate instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **92**A-**92**C illustrate Group Shift Left Immediate Subtract instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **93**A-**93**C illustrate Group Subtract Halve instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **94**A-**94**C illustrate Ensemble instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **95**A-**95**E illustrate Ensemble Convolve Extract Immediate instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **96**A-**96**E illustrate Ensemble Convolve Floating-Point instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **97**A-**97**G illustrate Ensemble Extract Immediate instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **98**A-**98**G illustrate Ensemble Extract Immediate Inplace instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **99**A-**99**C illustrate Ensemble Inplace instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **100**A-**100**E illustrate Wide Multiply Matrix instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **101**A-**101**E illustrate Wide Multiply Matrix Extract instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **102**A-**102**E illustrate Wide Multiply Matrix Extract Immediate instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **103**A-**103**E illustrate Wide Multiply Matrix Floating-Point Immediate instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **104**A-**104**D illustrate Wide Multiply Matrix Galois Immediate instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **105**A-**105**C illustrate Wide Switch Immediate instructions in accordance with an exemplary embodiment of the present invention.

FIGS. **106**A-**106**C illustrate Wide Translate instructions in accordance with an exemplary embodiment of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

Introduction

In various embodiments of the invention, a computer processor architecture, referred to here as MicroUnity's Zeus Architecture is presented. MicroUnity's Zeus Architecture describes general-purpose processor, memory, and interface subsystems, organized to operate at the enormously high bandwidth rates required for broadband applications.

The Zeus processor performs integer, floating point, signal processing and non-linear operations such as Galois field, table lookup and bit switching on data sizes from 1 bit to 128 bits. Group or SIMD (single instruction multiple data) operations sustain external operand bandwidth rates up to 512 bits (i.e., up to four 128-bit operand groups) per instruction even on data items of small size. The processor performs ensemble operations such as convolution that maintain full intermediate precision with aggregate internal operand bandwidth rates up to 20,000 bits per instruction. The processor performs wide operations such as crossbar switch, matrix multiply and table lookup that use caches embedded in the execution units themselves to extend operands to as much as 32768 bits. All instructions produce at most a single 128-bit register result, source at most three 128-bit registers and are free of side effects such as the setting of condition codes and flags. The instruction set design carries the concept of streamlining beyond Reduced Instruction Set Computer (RISC) architectures, to simplify implementations that issue several instructions per machine cycle.

The Zeus memory subsystem provides 64-bit virtual and physical addressing for UNIX, Mach, and other advanced OS environments. Separate address instructions enable the division of the processor into decoupled access and execution units, to reduce the effective latency of memory to the pipeline. The Zeus cache supplies the high data and instruction issue rates of the processor, and supports coherency primitives for scaleable multiprocessors. The memory subsystem includes mechanisms for sustaining high data rates not only in block transfer modes, but also in non-unit stride and scattered access patterns.

The Zeus interface subsystem is designed to match industry-standard "Socket 7" protocols and pin-outs. In this way, Zeus can make use of the immense infrastructure of the PC for building low-cost systems. The interface subsystem is modular, and can be replaced with appropriate protocols and pinouts for lower-cost and higher-performance systems.

The goal of the Zeus architecture is to integrate these processor, memory, and interface capabilities with optimal simplicity and generality. From the software perspective, the entire machine state consists of a program counter, a single bank of 64 general-purpose 128-bit registers, and a linear byte-addressed shared memory space with mapped interface registers. All interrupts and exceptions are precise, and occur with low overhead.

Examples discussed herein are for Zeus software and hardware developers alike, and defines the interface at which their designs must meet. Zeus pursues the most efficient tradeoffs between hardware and software complexity by making all processor, memory, and interface resources directly accessible to high-level language programs.

Conformance

To ensure that Zeus systems may freely interchange data, user-level programs, system-level programs and interface devices, the Zeus system architecture reaches above the processor level architecture.

Optional Areas

Optional areas include:

Number of processor threads

Size of first-level cache memories

Existence of a second-level cache

Size of second-level cache memory

Size of system-level memory

Existence of certain optional interface device interfaces

Upward-Compatible Modifications

Additional devices and interfaces, not covered by this standard may be added in specified regions of the physical memory space, provided that system reset places these devices and interfaces in an inactive state that does not interfere with the operation of software that runs in any conformant system. The software interface requirements of any such additional devices and interfaces must be made as widely available as this architecture specification.

Unrestricted Physical Implementation

Nothing in this specification should be construed to limit the implementation choices of the conforming system beyond the specific requirements stated herein. In particular, a computer system may conform to the Zeus System Architecture while employing any number of components, dissipate any amount of heat, require any special environmental facilities, or be of any physical size.

Common Elements

Notation

The descriptive notation used in this document is summarized in the table below:

descriptive notation	
$x + y$	two's complement addition of x and y. Result is the same size as the operands, and operands must be of equal size.
$x - y$	two's complement subtraction of y from x. Result is the same size as the operands, and operands must be of equal size.
$x * y$	two's complement multiplication of x and y. Result is the same size as the operands, and operands must be of equal size.
x/y	two's complement division of x by y. Result is the same size as the operands, and operands must be of equal size.
$x \& y$	bitwise and of x and y. Result is same size as the operands, and operands must be of equal size.
$x \mid y$	bitwise or of x and y. Result is same size as the operands, and operands must be of equal size.
$x \hat{} y$	bitwise exclusive-OR of x and y. Result is same size as the operands, and operands must be of equal size.
$\sim x$	bitwise inversion of x. Result is same size as the operand.
$x = y$	two's complement equality comparison between x and y. Result is a single bit, and operands must be of equal size.
$x \neq y$	two's complement inequality comparison between x and y. Result is a single bit, and operands must be of equal size.
$x < y$	two's complement less than comparison between x and y. Result is a single bit, and operands must be of equal size.
$x \geqq y$	two's complement greater than or equal comparison between x and y. Result is a single bit, and operands must be of equal size.
$\sqrt{x}$	floating-point square root of x
$x \Vert y$	concatenation of bit field x to left of bit field y
x^y	binary digit x repeated, concatenated y times. Size of result is y.
x_y	extraction of bit y (using little-endian bit numbering) from value x. Result is a single bit.

-continued

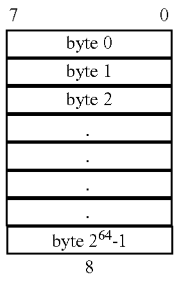
descriptive notation	
$x_{y \ldots z}$	extraction of bit field formed from bits y through z of value x. Size of result is y − z + 1; if z > y, result is an empty string,
x?y:z	value of y, if x is true, otherwise value of z. Value of x is a single bit.
$x \leftarrow y$	bitwise assignment of x to value of y
Sn	signed, two's complement, binary data format of n bytes
Un	unsigned binary data format of n bytes
Fn	floating-point data format of n bytes

Bit Ordering

The ordering of bits in this document is always little-endian, regardless of the ordering of bytes within larger data structures. Thus, the least-significant bit of a data structure is always labeled 0 (zero), and the most-significant bit is labeled as the data structure size (in bits) minus one.

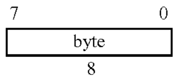
Memory

Zeus memory is an array of 2^{64} bytes, without a specified byte ordering, which is physically distributed among various components.



Byte

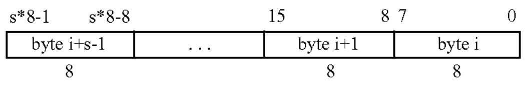
A byte is a single element of the memory array, consisting of 8 bits:



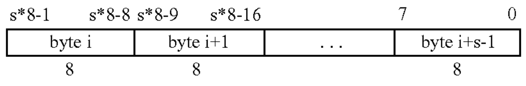
Byte Ordering

Larger data structures are constructed from the concatenation of bytes in either little-endian or big-endian byte ordering. A memory access of a data structure of size s at address i is formed from memory bytes at addresses i through i+s−1. Unless otherwise specified, there is no specific requirement of alignment: it is not generally required that i be a multiple of s. Aligned accesses are preferred whenever possible, however, as they will often require one fewer processor or memory clock cycle than unaligned accesses.

With little-endian byte ordering, the bytes are arranged as:



With big-endian byte ordering, the bytes are arranged as:



Zeus memory is byte-addressed, using either little-endian or big-endian byte ordering. For consistency with the bit ordering, and for compatibility with x86 processors, Zeus uses little-endian byte ordering when an ordering must be selected. Zeus load and store instructions are available for both little-endian and big-endian byte ordering. The selection of byte ordering is dynamic, so that little-endian and big-endian processes, and even data structures within a process, can be intermixed on the processor.

Memory Read/Load Semantics

Zeus memory, including memory-mapped registers, must conform to the following requirements regarding side-effects of read or load operations:

A memory read must have no side-effects on the contents of the addressed memory nor on the contents of any other memory.

Memory Write/Store Semantics

Zeus memory, including memory-mapped registers, must conform to the following requirements regarding side-effects of read or load operations:

A memory write must affect the contents of the addressed memory so that a memory read of the addressed memory returns the value written, and so that a memory read of a portion of the addressed memory returns the appropriate portion of the value written.

A memory write may affect or cause side-effects on the contents of memory not addressed by the write operation, however, a second memory write of the same value to the same address must have no side-effects on any memory; memory write operations must be idempotent.

Zeus store instructions that are weakly ordered may have side-effects on the contents of memory not addressed by the store itself; subsequent load instructions which are also weakly ordered may or may not return values which reflect the side-effects.

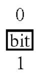
Data

Zeus provides eight-byte (64-bit) virtual and physical address sizes, and eight-byte (64-bit) and sixteen-byte (128-bit) data path sizes, and uses fixed-length four-byte (32-bit) instructions. Arithmetic is performed on two's-complement or unsigned binary and ANSI/IEEE standard 754-1985 conforming binary floating-point number representations.

Fixed-Point Data

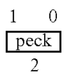
Bit

A bit is a primitive data element:



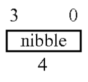
Peck

A peck is the catenation of two bits:



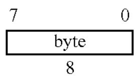
Nibble

A nibble is the catenation of four bits:



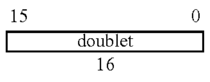
Byte

A byte is the catenation of eight bits, and is a single element of the memory array:



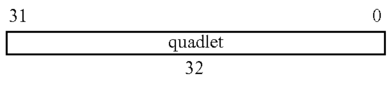
Doublet

A doublet is the catenation of 16 bits, and is the catenation of two bytes:



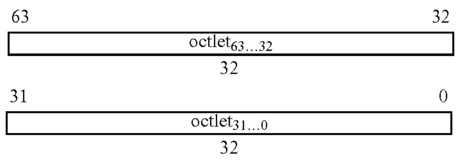
Quadlet

A quadlet is the catenation of 32 bits, and is the catenation of four bytes:



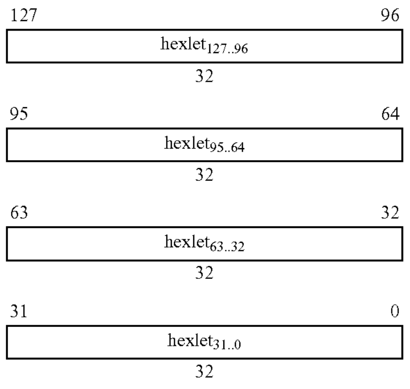
Octlet

An octlet is the catenation of 64 bits, and is the catenation of eight bytes:



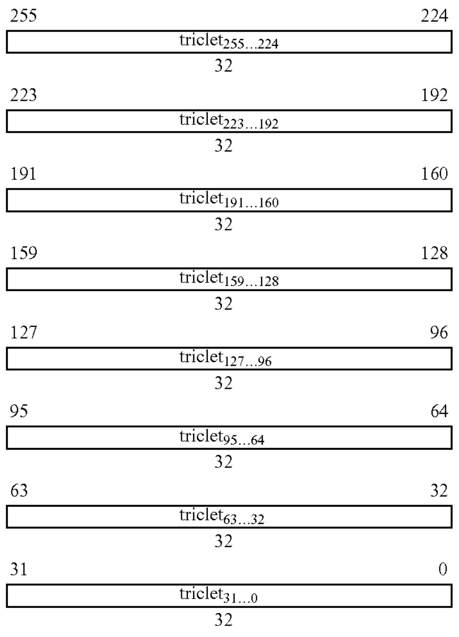
Hexlet

A hexlet is the catenation of 128 bits, and is the catenation of sixteen bytes:



Triclet

A triclet is the catenation of 256 bits, and is the catenation of thirty-two bytes:



Address

Zeus addresses, both virtual addresses and physical addresses, are octlet quantities.

Floating-Point Data

Zeus's floating-point formats are designed to satisfy ANSI/IEEE standard 754-1985: Binary Floating-point Arithmetic. Standard 754 leaves certain aspects to the discretion of implementers: additional precision formats, encoding of quiet and signaling NaN values, details of production and propagation of quiet NaN values. These aspects are detailed below.

Zeus adds additional half-precision and quad-precision formats to standard 754's single-precision and double-precision formats. Zeus's double-precision satisfies standard 754's precision requirements for a single-extended format, and Zeus's quad-precision satisfies standard 754's precision requirements for a double-extended format.

Each precision format employs fields labeled s (sign), e (exponent), and f (fraction) to encode values that are (1) NaN: quiet and signaling, (2) infinities: $(-1)^{\hat{}s}\infty$, (3) normalized numbers: $(-1)^{\hat{}s}2^{\hat{}e-bias}(1.f)$, (4) denormalized numbers: $(-1)^{\hat{}s}2^{\hat{}1-bias}(0.f)$, and (5) zero: $(-1)^{\hat{}s}0$.

Quiet NaN values are denoted by any sign bit value, an exponent field of all one bits, and a non-zero fraction with the most significant bit set. Quiet NaN values generated by default exception handling of standard operations have a zero sign bit, an exponent field of all one bits, a fraction field with the most significant bit set, and all other bits cleared.

Signaling NaN values are denoted by any sign bit value, an exponent field of all one bits, and a non-zero fraction with the most significant bit cleared.

Infinite values are denoted by any sign bit value, an exponent field of all one bits, and a zero fraction field.

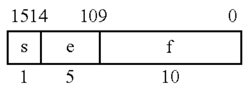
Normalized number values are denoted by any sign bit value, an exponent field that is not all one bits or all zero bits, and any fraction field value. The numeric value encoded is $(-1)^{\hat{}s}2^{\hat{}e-bias}(1.f)$. The bias is equal the value resulting from setting all but the most significant bit of the exponent field, half: 15, single: 127, double: 1023, and quad: 16383.

Denormalized number values are denoted by any sign bit value, an exponent field that is all zero bits, and a non-zero fraction field value. The numeric value encoded is $(-1)^{\hat{}s}2^{\hat{}1-bias}(0.f)$.

Zero values are denoted by any sign bit value, and exponent field that is all zero bits, and a fraction field that is all zero bits. The numeric value encoded is $(-1)^{\hat{}s}0$. The distinction between +0 and −0 is significant in some operations.

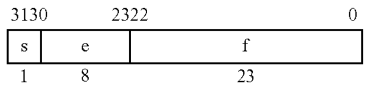
Half-Precision Floating-Point

Zeus half precision uses a format similar to standard 754's requirements, reduced to a 16-bit overall format. The format contains sufficient precision and exponent range to hold a 12-bit signed integer.



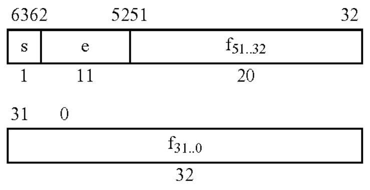
Single-precision Floating-Point

Zeus single precision satisfies standard 754's requirements for "single."



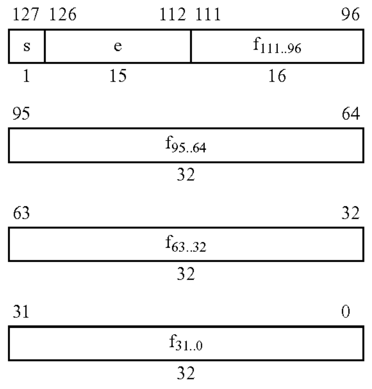
Double-Precision Floating-Point

Zeus double precision satisfies standard 754's requirements for "double."



Quad-Precision Floating-Point

Zeus quad precision satisfies standard 754's requirements for "double extended," but has additional fraction precision to use 128 bits.



Zeus Processor

MicroUnity's Zeus processor provides the general-purpose, high-bandwidth computation capability of the Zeus system. Zeus includes high-bandwidth data paths, register files, and a memory hierarchy. Zeus's memory hierarchy includes on-chip instruction and data memories, instruction and data caches, a virtual memory facility, and interfaces to external devices. Zeus's interfaces in the initial implementation are solely the "Super Socket 7" bus, but other implementations may have different or additional interfaces.

Architectural Framework

The Zeus architecture defines a compatible framework for a family of implementations with a range of capabilities. The following implementation-defined parameters are used in the rest of the document in boldface. The value indicated is for MicroUnity's first Zeus implementation.

Parameter	Interpretation	Value	Range of legal values
T	number of execution threads	4	$1 \leqq \mathbf{T} \leqq 31$
CE	$\log_2$ cache blocks in first-level cache	9	$0 \leqq \mathbf{CE} \leqq 31$
CS	$\log_2$ cache blocks in first-level cache set	2	$0 \leqq \mathbf{CS} \leqq 4$
CT	existence of dedicated tags in first-level cache	1	$0 \leqq \mathbf{CT} \leqq 1$
LE	$\log_2$ entries in local TB	0	$0 \leqq \mathbf{LE} \leqq 3$
LB	Local TB based on base register	1	$0 \leqq \mathbf{LB} \leqq 1$
GE	$\log_2$ entries in global TB	7	$0 \leqq \mathbf{GE} \leqq 15$
GT	$\log_2$ threads which share a global TB	1	$0 \leqq \mathbf{GT} \leqq 3$

Interfaces and Block Diagram

The first implementation of Zeus uses "socket 7" protocols and pinouts.

Instruction

Assembler Syntax

Instructions are specified to Zeus assemblers and other code tools (assemblers) in the syntax of an instruction mnemonic (operation code), then optionally white space (blanks or tabs) followed by a list of operands.

The instruction mnemonics listed in this specification are in upper case (capital) letters, assemblers accept either upper case or lower case letters in the instruction mnemonics. In this specification, instruction mnemonics contain periods (".") to separate elements to make them easier to understand; assemblers ignore periods within instruction mnemonics. The instruction mnemonics are designed to be parsed uniquely without the separating periods.

If the instruction produces a register result, this operand is listed first. Following this operand, if there are one or more source operands, is a separator which may be a comma (","), equal ("="), or at-sign ("@"). The equal separates the result operand from the source operands, and may optionally be expressed as a comma in assembler code. The at-sign indicates that the result operand is also a source operand, and may optionally be expressed as a comma in assembler code. If the instruction specification has an equal-sign, an at-sign in assembler code indicates that the result operand should be repeated as the first source operand (for example, "A.ADD.I r4@5" is equivalent to "A.ADD.I r4=r4,5"). Commas always separate the remaining source operands.

The result and source operands are case-sensitive; upper case and lower case letters are distinct. Register operands are specified by the names r0 (or r00) through r63 (a lower case "r" immediately followed by a one or two digit number from 0 to 63), or by the special designations of "Ip" for "r0," "dp" for "r1," "fp" for "r62," and "sp" for "r63." Integer-valued operands are specified by an optional sign (−) or (+) followed by a number, and assemblers generally accept a variety of integer-valued expressions.

Instruction Structure

A Zeus instruction is specifically defined as a four-byte structure with the little-endian ordering shown below. It is different from the quadlet defined above because the placement of instructions into memory must be independent of the byte ordering used for data structures. Instructions must be aligned on four-byte boundaries; in the diagram below, i must be a multiple of 4.

31 … 24	23 … 16	15 … 8	7 … 0
byte i + 3	byte i + 2	byte i + 1	byte i
8	8	8	8

Gateway

A Zeus gateway is specifically defined as an 8-byte structure with the little-endian ordering shown below. A gateway contains a code address used to securely invoke a system call or procedure at a higher privilege level. Gateways are marked by protection information specified in the TB. Gateways must be aligned on 8-byte boundaries; in the diagram below, i must be a multiple of 8.

63 … 56	55 … 48	47 … 40	39 … 32
byte i + 7	byte i + 6	byte i + 5	byte i + 4
8	8	8	8

31 … 24	23 … 16	15 … 8	7 … 0
byte i + 3	byte i + 2	byte i + 1	byte i
8	8	8	8

The gateway contains two data items within its structure, a code address and a new privilege level:

63 … 2	1 … 0
code address	pl
62	2

The virtual memory system can be used to designate a region of memory as containing gateways. Other data may be placed within the gateway region, provided that if an attempt is made to use additional data as a gateway, that security cannot be violated. For example, 64-data or stack pointers which are aligned to at least 4 bytes and are in little-endian byte order have pl=0 that the privilege level cannot be raised by attempting to use the additional data as a gateway.

User State

The user state consists of hardware data structures that are accessible to all conventional compiled code. The Zeus user state is designed to be as regular as possible, and consists of the general registers, the program counter, and virtual memory. There are no specialized registers for condition codes, operating modes, rounding modes, integer multiply/divide, or floating-point values.

General Registers

Zeus user state includes 64 general registers. All are identical; there is no dedicated zero-valued register, and there are no dedicated floating-point registers.

127 … 0
REG[0]
REG[1]
REG[2]
⋮
REG[62]
REG[63]
128

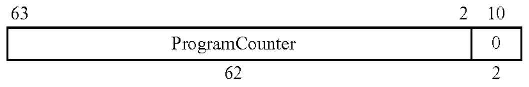
Some Zeus instructions have 64-bit register operands. These operands are sign-extended to 128 bits when written to the register file, and the low-order 64 bits are chosen when read from the register file.

Definition

```
def val ← RegRead(rn, size)
    case size of
        64:
            val ← $REG[rn]_{63..0}$
        128:
            val ← REG[rn]
    endcase
enddef
def RegWrite(rn, size, val)
    case size of
        64:
            REG[rn] ← $val_{63}^{64} \| val_{63..0}$
        128:
            REG[rn] ← $val_{127..0}$
    endcase
enddef
```

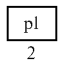
Program Counter

The program counter contains the address of the currently executing instruction. This register is implicitly manipulated by branch instructions, and read by branch instructions that save a return address in a general register.



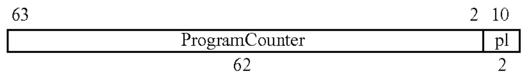
Privilege Level

The privilege level register contains the privilege level of the currently executing instruction. This register is implicitly manipulated by branch gateway and branch down instructions, and read by branch gateway instructions that save a return address in a general register.



Program Counter and Privilege Level

The program counter and privilege level may be packed into a single octlet. This combined data structure is saved by the Branch Gateway instruction and restored by the Branch Down instruction.



System State

The system state consists of the facilities not normally used by conventional compiled code. These facilities provide mechanisms to execute such code in a fully virtual environment. All system state is memory mapped, so that it can be manipulated by compiled code.

Fixed-Point

Zeus provides load and store instructions to move data between memory and the registers, branch instructions to compare the contents of registers and to transfer control from one code address to another, and arithmetic operations to perform computation on the contents of registers, returning the result to registers.

Load and Store

The load and store instructions move data between memory and the registers. When loading data from memory into a register, values are zero-extended or sign-extended to fill the register. When storing data from a register into memory, values are truncated on the left to fit the specified memory region.

Load and store instructions that specify a memory region of more than one byte may use either little-endian or big-endian byte ordering: the size and ordering are explicitly specified in the instruction. Regions larger than one byte may be either aligned to addresses that are an even multiple of the size of the region or of unspecified alignment: alignment checking is also explicitly specified in the instruction.

Load and store instructions specify memory addresses as the sum of a base general register and the product of the size of the memory region and either an immediate value or another general register. Scaling maximizes the memory space which can be reached by immediate offsets from a single base general register, and assists in generating memory addresses within iterative loops. Alignment of the address can be reduced to checking the alignment of the first general register.

The load and store instructions are used for fixed-point data as well as floating-point and digital signal processing data; Zeus has a single bank of registers for all data types.

Swap instructions provide multithread and multiprocessor synchronization, using indivisible operations: add-swap, compare-swap, multiplex-swap, and double-compare-swap. A store-multiplex operation provides the ability to indivisibly write to a portion of an octlet. These instructions always operate on aligned octlet data, using either little-endian or big-endian byte ordering.

Branch

The fixed-point compare-and-branch instructions provide all arithmetic tests for equality and inequality of signed and unsigned fixed-point values. Tests are performed either between two operands contained in general registers, or on the bitwise and of two operands. Depending on the result of the compare, either a branch is taken, or not taken. A taken branch causes an immediate transfer of the program counter to the target of the branch, specified by a 12-bit signed offset from the location of the branch instruction. A non-taken branch causes no transfer; execution continues with the following instruction.

Other branch instructions provide for unconditional transfer of control to addresses too distant to be reached by a 12-bit offset, and to transfer to a target while placing the location following the branch into a register. The branch through gateway instruction provides a secure means to access code at a higher privilege level, in a form similar to a normal procedure call.

Addressing Operations

A subset of general fixed-point arithmetic operations is available as addressing operations. These include add, subtract, Boolean, and simple shift operations. These addressing operations may be performed at a point in the Zeus processor pipeline so that they may be completed prior to or in conjunction with the execution of load and store operations in a "superspring" pipeline in which other arithmetic operations are deferred until the completion of load and store operations.

Execution Operations

Many of the operations used for Digital Signal Processing (DSP), which are described in greater detail below, are also used for performing simple scalar operations. These operations perform arithmetic operations on values of 8-, 16-, 32-, 64-, or 128-bit sizes, which are right-aligned in registers. These execution operations include the add, subtract, boolean and simple shift operations which are also available as addressing operations, but further extend the available set to include three-operand add/subtract, three-operand boolean, dynamic shifts, and bit-field operations.

Floating-Point Zeus provides all the facilities mandated and recommended by ANSI/IEEE standard 754-1985: Binary Floating-point Arithmetic, with the use of supporting software.

Branch Conditionally

The floating-point compare-and-branch instructions provide all the comparison types required and suggested by the IEEE floating-point standard. These floating-point comparisons augment the usual types of numeric value comparisons with special handling for NaN (not-a-number) values. A NaN value compares as "unordered" with respect to any other value, even that of an identical NaN value.

Zeus floating-point compare-branch instructions do not generate an exception on comparisons involving quiet or signaling NaN values. If such exceptions are desired, they can be obtained by combining the use of a floating-point compare-set instruction, with either a floating-point compare-branch instruction on the floating-point operands or a fixed-point compare-branch on the set result.

Because the less and greater relations are anti-commutative, one of each relation that differs from another only by the replacement of an L with a G in the code can be removed by reversing the order of the operands and using the other code. Thus, an L relation can be used in place of a G relation by swapping the operands to the compare-branch or compare-set instruction.

No instructions are provided that branch when the values are unordered. To accomplish such an operation, use the reverse condition to branch over an immediately following unconditional branch, or in the case of an if-then-else clause, reverse the clauses and use the reverse condition.

The E relation can be used to determine the unordered condition of a single operand by comparing the operand with itself.

The following floating-point compare-branch relations are provided as instructions:

compare-branch relations

Mnemonic		Branch taken if values compare as:				Exception if	
code	C-like	Unordered	Greater	Less	Equal	unordered	invalid
E	==	F	F	F	T	no	no
LG	<>	F	T	T	F	no	no
L	<	F	F	T	F	no	no
GE	>=	F	T	F	T	no	no

Compare-Set

The compare-set floating-point instructions provide all the comparison types supported as branch instructions. Zeus compare-set floating-point instructions may optionally generate an exception on comparisons involving quiet or signaling NaNs.

The following floating-point compare-set relations are provided as instructions:

compare-set relations

Mnemonic		Result if values compare as:				Exception if	
code	C-like	Unordered	Greater	Less	Equal	unordered	invalid
E	==	F	F	F	T	no	no
LG	<>	F	T	T	F	no	no
L	<	F	F	T	F	no	no
GE	>=	F	T	F	T	no	no
E.X	==	F	F	F	T	no	yes
LG.X	<>	F	T	T	F	no	yes
L.X	<	F	F	T	F	yes	yes
GE.X	<=	F	T	F	T	yes	yes

Arithmetic Operations

The basic operations supported in hardware are floating-point add, subtract, multiply, divide, square root and conversions among floating-point formats and between floating-point and binary integer formats.

Software libraries provide other operations required by the ANSI/IEEE floating-point standard.

The operations explicitly specify the precision of the operation, and round the result (or check that the result is exact) to the specified precision at the conclusion of each operation. Each of the basic operations splits operand registers into symbols of the specified precision and performs the same operation on corresponding symbols.

In addition to the basic operations, Zeus performs a variety of operations in which one or more products are summed to each other and/or to an additional operand. The instructions include a fused multiply-add (E.MUL.ADD.F), convolve (E.CON.F), matrix multiply (E.MUL.MAT.F), and scale-add (E.SCAL.ADD.F).

The results of these operations are computed as if the multiplies are performed to infinite precision, added as if in infinite precision, then rounded only once. Consequently, these operations perform these operations with no rounding of intermediate results that would have limited the accuracy of the result.

NaN Handling

ANSI/IEEE standard 754-1985 specifies that operations involving a signaling NaN or invalid operation shall, if no trap occurs and if a floating-point result is to be delivered, deliver a quiet NaN as its result. However, it fails to specify what quiet NaN value to deliver.

Zeus operations that produce a floating-point result and do not trap on invalid operations propagate signaling NaN values from operands to results, changing the signaling NaN values to quiet NaN values by setting the most significant fraction bit and leaving the remaining bits unchanged. Other causes of invalid operations produce the default quiet NaN value, where the sign bit is zero, the exponent field is all one bits, the most significant fraction bit is set and the remaining fraction bits are zero bits. For Zeus operations that produce multiple results catenated together, signaling NaN propagation or quiet NaN production is handled separately and independently for each result symbol.

ANSI/IEEE standard 754-1985 specifies that quiet NaN values should be propagated from operand to result by the basic operations. However, it fails to specify which of several quiet NaN values to propagate when more than one operand is a quiet NaN. In addition, the standard does not clearly specify how quiet NaN should be propagated for the multiple-operation instructions provided in Zeus. The standard does not specify the quiet NaN produced as a result of an operand being a signaling NaN when invalid operation exceptions are handled by default. The standard leaves unspecified how quiet and signaling NaN values are propagated though format conversions and the absolute-value, negate and copy operations. This section specifies these aspects left unspecified by the standard.

First of all, for Zeus operations that produce multiple results catenated together, quiet and signaling NaN propagation is handled separately and independently for each result symbol. A quiet or signaling NaN value in a single symbol of an operand causes only those result symbols that are dependent on that operand symbol's value to be propagated as that quiet NaN. Multiple quiet or signaling NaN values in symbols of an operand which influence separate symbols of the result are propagated independently of each other. Any signaling NaN that is propagated has the high-order fraction bit set to convert it to a quiet NaN.

For Zeus operations in which multiple symbols among operands upon which a result symbol is dependent are quiet or signaling NaNs, a priority Rule will determine which NaN is propagated. Priority shall be given to the operand that is specified by a register definition at a lower-numbered (little-endian) bit position within the instruction (rb has priority over rc, which has priority over rd). In the case of operands which are catenated from two registers, priority shall be assigned based on the register which has highest priority (lower-numbered bit position within the instruction). In the case of tie (as when the E.SCAL.ADD scaling operand has two corresponding NaN values, or when a E.MUL.CF operand has NaN values for both real and imaginary components of a value), the value which is located at a lower-numbered (little-endian) bit position within the operand is to receive priority. The identification of a NaN as quiet or signaling shall not confer any priority for selection—only the operand position, though a signaling NaN will cause an invalid operand exception.

The sign bit of NaN values propagated shall be complemented if the instruction subtracts or negates the corresponding operand or (but not and) multiplies it by or divides it by or divides it into an operand which has the sign bit set, even if that operand is another NaN. If a NaN is both subtracted and multiplied by a negative value, the sign bit shall be propagated unchanged.

For Zeus operations that convert between two floating-point formats (INFLATE and DEFLATE), NaN values are propagated by preserving the sign and the most-significant fraction bits, except that the most-significant bit of a signalling NaN is set and (for DEFLATE) the least-significant fraction bit preserved is combined, via a logical- or of all fraction bits not preserved. All additional fraction bits (for INFLATE) are set to zero.

For Zeus operations that convert from a floating-point format to a fixed-point format (SINK), NaN values produce zero values (maximum-likelihood estimate). Infinity values produce the largest representable positive or negative fixed-point value that fits in the destination field. When exception traps are enabled, NaN or Infinity values produce a floating-point exception. Underflows do not occur in the SINK operation, they produce −1, 0 or +1, depending on rounding controls.

For absolute-value, negate, or copy operations, NaN values are propagated with the sign bit cleared, complemented, or copied, respectively. Signalling NaN values cause the Invalid operation exception, propagating a quieted NaN in corresponding symbol locations (default) or an exception, as specified by the instruction.

Floating-Point Functions

Referring to FIG. **37**, the following functions are defined for use within the detailed instruction definitions in the following section. In these functions an internal format represents infinite-precision floating-point values as a four-element structure consisting of (1) s (sign bit): 0 for positive, 1 for negative, (2) t (type): NORM, ZERO, SNAN, QNAN, INFINITY, (3) e (exponent), and (4) f: (fraction). The mathematical interpretation of a normal value places the binary point at the units of the fraction, adjusted by the exponent: $(-1)^{\wedge s}*(2^{\wedge e})*f$. The function F converts a packed IEEE floating-point value into internal format. The function PackF converts an internal format back into IEEE floating-point format, with rounding and exception control.

Digital Signal Processing

The Zeus processor provides a set of operations that maintain the fullest possible use of 128-bit data paths when operating on lower-precision fixed-point or floating-point vector values. These operations are useful for several application areas, including digital signal processing, image processing and synthetic graphics. The basic goal of these operations is to accelerate the performance of algorithms that exhibit the following characteristics:

Low-Precision Arithmetic

The operands and intermediate results are fixed-point values represented in no greater than 64 bit precision. For floating-point arithmetic, operands and intermediate results are of 16, 32, or 64 bit precision.

The fixed-point arithmetic operations include add, subtract, multiply, divide, shifts, and set on compare.

The use of fixed-point arithmetic permits various forms of operation reordering that are not permitted in floating-point arithmetic. Specifically, commutativity and associativity, and distribution identities can be used to reorder operations. Compilers can evaluate operations to determine what intermediate precision is required to get the specified arithmetic result.

Zeus supports several levels of precision, as well as operations to convert between these different levels. These precision levels are always powers of two, and are explicitly specified in the operation code.

When specified, add, subtract, and shift operations may cause a fixed-point arithmetic exception to occur on resulting conditions such as signed or unsigned overflow. The fixed-point arithmetic exception may also be invoked upon a signed or unsigned comparison.

Sequential Access to Data

The algorithms are or can be expressed as operations on sequentially ordered items in memory. Scatter-gather memory access or sparse-matrix techniques are not required.

Where an index variable is used with a multiplier, such multipliers must be powers of two. When the index is of the form: nx+k, the value of n must be a power of two, and the values referenced should have k include the majority of values in the range 0 . . . n−1. A negative multiplier may also be used.

Vectorizable Operations

The operations performed on these sequentially ordered items are identical and independent. Conditional operations are either rewritten to use Boolean variables or masking, or the compiler is permitted to convert the code into such a form.

Data-Handling Operations

The characteristics of these algorithms include sequential access to data, which permit the use of the normal load and store operations to reference the data. Octlet and hexlet loads and stores reference several sequential items of data, the number depending on the operand precision.

The discussion of these operations is independent of byte ordering, though the ordering of bit fields within octlets and hexlets must be consistent with the ordering used for bytes. Specifically, if big-endian byte ordering is used for the loads and stores, the figures below should assume that index values increase from left to right, and for little-endian byte ordering, the index values increase from right to left. For this reason, the figures indicate different index values with different shades, rather than numbering.

When an index of the nx+k form is used in array operands, where n is a power of 2, data memory sequentially loaded contains elements useful for separate operands. The “shuffle” instruction divides a triclet of data up into two hexlets, with alternate bit fields of the source triclet grouped together into the two results. An immediate field, h, in the instruction specifies which of the two regrouped hexlets to select for the result. For example, two X.SHUFFLE.256 rd=rc,rb,32,128,h operations rearrange the source triclet (c,b) into two hexlets as in FIG. **54**A.

In the shuffle operation, two hexlet registers specify the source triclet, and one of the two result hexlets are specified as hexlet register.

The example above directly applies to the case where n is 2. When n is larger, shuffle operations can be used to further subdivide the sequential stream. For example, when n is 4, we need to deal out 4 sets of doublet operands, as shown in FIG. **54**B (An example of the use of a four-way deal is a digital signal processing application such as conversion of color to monochrome).

When an array result of computation is accessed with an index of the form nx+k, for n a power of 2, the reverse of the "deal" operation needs to be performed on vectors of results to interleave them for storage in sequential order. The "shuffle" operation interleaves the bit fields of two octlets of results into a single hexlet. For example a X.SHUFFLE.16 operation combines two octlets of doublet fields into a hexlet as shown in FIG. **54**C.

For larger values of n, a series of shuffle operations can be used to combine additional sets of fields, similarly to the mechanism used for the deal operations. For example, when n is 4, we need to shuffle up 4 sets of doublet operands, as shown in FIG. **54**D (An example of the use of a four-way shuffle is a digital signal processing application such as conversion of monochrome to color).

When the index of a source array operand or a destination array result is negated, or in other words, if of the form nx+k where n is negative, the elements of the array must be arranged in reverse order. The "swizzle" operation can reverse the order of the bit fields in a hexlet. For example, a X.SWIZZLE rd=rc,127,112 operation reverses the doublets within a hexlet as shown in FIG. **47**C.

In some cases, it is desirable to use a group instruction in which one or more operands is a single value, not an array. The "swizzle" operation can also copy operands to multiple locations within a hexlet. For example, a X.SWIZZLE 15,0 operation copies the low-order 16 bits to each double within a hexlet.

Variations of the deal and shuffle operations are also useful for converting from one precision to another. This may be required if one operand is represented in a different precision than another operand or the result, or if computation must be performed with intermediate precision greater than that of the operands, such as when using an integer multiply.

When converting from a higher precision to a lower precision, specifically when halving the precision of a hexlet of bit fields, half of the data must be discarded, and the bit fields packed together. The "compress" operation is a variant of the "deal" operation, in which the operand is a hexlet, and the result is an octlet. An arbitrary half-sized sub-field of each bit field can be selected to appear in the result. For example, a selection of bits **19** . . . **4** of each quadlet in a hexlet is performed by the X.COMPRESS rd=rc,16,4 operation as shown in FIG. **43**D.

When converting from lower-precision to higher-precision, specifically when doubling the precision of an octlet of bit fields, one of several techniques can be used, either multiply, expand, or shuffle. Each has certain useful properties. In the discussion below, m is the precision of the source operand.

The multiply operation, described in detail below, automatically doubles the precision of the result, so multiplication

by a constant vector will simultaneously double the precision of the operand and multiply by a constant that can be represented in m bits.

An operand can be doubled in precision and shifted left with the "expand" operation, which is essentially the reverse of the "compress" operation. For example the X.EXPAND rd=rc,16,4 expands from 16 bits to 32, and shifts 4 bits left as shown in FIG. **54**E.

The "shuffle" operation can double the precision of an operand and multiply it by 1 (unsigned only), 2^m or 2^m+1, by specifying the sources of the shuffle operation to be a zeroed register and the source operand, the source operand and zero, or both to be the source operand. When multiplying by 2 m, a constant can be freely added to the source operand by specifying the constant as the right operand to the shuffle.

Arithmetic Operations

The characteristics of the algorithms that affect the arithmetic operations most directly are low-precision arithmetic, and vectorizable operations. The fixed-point arithmetic operations provided are most of the functions provided in the standard integer unit, except for those that check conditions. These functions include add, subtract, bitwise Boolean operations, shift, set on condition, and multiply, in forms that take packed sets of bit fields of a specified size as operands. The floating-point arithmetic operations provided are as complete as the scalar floating-point arithmetic set. The result is generally a packed set of bit fields of the same size as the operands, except that the fixed-point multiply function intrinsically doubles the precision of the bit field.

Conditional operations are provided only in the sense that the set on condition operations can be used to construct bit masks that can select between alternate vector expressions, using the bitwise Boolean operations. All instructions operate over the entire octlet or hexlet operands, and produce a hexlet result. The sizes of the bit fields supported are always powers of two.

Galois Field Operations

Zeus provides a general software solution to the most common operations required for Galois Field arithmetic. The instructions provided include a polynomial multiply, with the polynomial specified as one register operand. This instruction can be used to perform CRC generation and checking, Reed-Solomon code generation and checking, and spread-spectrum encoding and decoding.

Software Conventions

The following section describes software conventions that are to be employed at software module boundaries, in order to permit the combination of separately compiled code and to provide standard interfaces between application, library and system software. Register usage and procedure call conventions may be modified, simplified or optimized when a single compilation encloses procedures within a compilation unit so that the procedures have no external interfaces. For example, internal procedures may permit a greater number of register-passed parameters, or have registers allocated to avoid the need to save registers at procedure boundaries, or may use a single stack or data pointer allocation to suffice for more than one level of procedure call.

Register Usage

All Zeus registers are identical and general-purpose; there is no dedicated zero-valued register, and no dedicated floating-point registers. However, some procedure-call-oriented instructions imply usage of registers zero (0) and one (1) in a manner consistent with the conventions described below. By software convention, the non-specific general registers are used in more specific ways.

register usage

register number	assembler names	usage	how saved
0	lp, r0	link pointer	caller
1	dp, r1	data pointer	caller
2-9	r2-r9	parameters	caller
10-31	r10-r31	temporary	caller
32-61	r32-r61	saved	callee
62	fp, r62	frame pointer	callee
63	sp, r63	stack pointer	callee

At a procedure call boundary, registers are saved either by the caller or callee procedure, which provides a mechanism for leaf procedures to avoid needing to save registers. Compilers may choose to allocate variables into caller or callee saved registers depending on how their lifetimes overlap with procedure calls.

Procedure Calling Conventions

Procedure parameters are normally allocated in registers, starting from register 2 up to register 9. These registers hold up to 8 parameters, which may each be of any size from one byte to sixteen bytes (hexlet), including floating-point and small structure parameters. Additional parameters are passed in memory, allocated on the stack. For C procedures which use varargs.h or stdarg.h and pass parameters to further procedures, the compilers must leave room in the stack memory allocation to save registers 2 through 9 into memory contiguously with the additional stack memory parameters, so that procedures such as_doprnt can refer to the parameters as an array.

Procedure return values are also allocated in registers, starting from register 2 up to register 9. Larger values are passed in memory, allocated on the stack.

There are several pointers maintained in registers for the procedure calling conventions: lp, sp, dp, fp.

The lp register contains the address to which the callee should return to at the conclusion of the procedure. If the procedure is also a caller, the lp register will need to be saved on the stack, once, before any procedure call, and restored, once, after all procedure calls. The procedure returns with a branch instruction, specifying the lp register.

The sp register is used to form addresses to save parameter and other registers, maintain local variables, i.e., data that is allocated as a LIFO stack. For procedures that require a stack, normally a single allocation is performed, which allocates space for input parameters, local variables, saved registers, and output parameters all at once. The sp register is always hexlet aligned.

The dp register is used to address pointers, literals and static variables for the procedure. The dp register points to a small (approximately 4096-entry) array of pointers, literals, and statically-allocated variables, which is used locally to the procedure. The uses of the dp register are similar to the use of the gp register on a Mips R-series processor, except that each procedure may have a different value, which expands the space addressable by small offsets from this pointer. This is an important distinction, as the offset field of Zeus load and store instructions are only 12 bits. The compiler may use additional registers and/or indirect pointers to address larger regions for a single procedure. The compiler may also share a single dp register value between procedures which are compiled as a single unit (including procedures which are externally callable), eliminating the need to save, modify and restore the dp register for calls between procedures which share the same dp register value.

Load- and store-immediate-aligned instructions, specifying the dp register as the base register, are generally used to obtain values from the dp region. These instructions shift the immediate value by the logarithm of the size of the operand, so loads and stores of large operands may reach farther from the dp register than of small operands. Referring to FIG. **54**F, the size of the addressable region is maximized if the elements to be placed in the dp region are sorted according to size, with the smallest elements placed closest to the dp base. At points where the size changes, appropriate padding is added to keep elements aligned to memory boundaries matching the size of the elements. Using this technique, the maximum size of the dp region is always at least 4096 items, and may be larger when the dp area is composed of a mixture of data sizes.

The dp register mechanism also permits code to be shared, with each static instance of the dp region assigned to a different address in memory. In conjunction with position-independent or pc-relative branches, this allows library code to be dynamically relocated and shared between processes.

To implement an inter-module (separately compiled) procedure call, the lp register is loaded with the entry point of the procedure, and the dp register is loaded with the value of the dp register required for the procedure. These two values are located adjacent to each other as a pair of octlet quantities in the dp region for the calling procedure. For a statically-linked inter-module procedure call, the linker fills in the values at link time. However, this mechanism also provides for dynamic linking, by initially filling in the lp and dp fields in the data structure to invoke the dynamic linker. The dynamic linker can use the contents of the lp and/or dp registers to determine the identity of the caller and callee, to find the location to fill in the pointers and resume execution. Specifically, the lp value is initially set to point to an entry point in the dynamic linker, and the dp value is set to point to itself: the location of the lp and dp values in the dp region of the calling procedure. The identity of the procedure can be discovered from a string following the dp pointer, or a separate table, indexed by the dp pointer.

The fp register is used to address the stack frame when the stack size varies during execution of a procedure, such as when using the GNU C alloca function. When the stack size can be determined at compile time, the sp register is used to address the stack frame and the fp register may be used for any other general purpose as a callee-saved register.

Typical static-linked, intra-module calling sequence:

```
              caller (non-leaf):
caller:   A.ADDI          sp@-size     // allocate caller stack frame
          S.I.64.A        lp,sp,off    // save original lp register
          ... (callee using same dp as caller)
          B.LINK.I        callee
          ...
          ... (callee using same dp as caller)
          B.LINK.I        callee
          ...
          L.I.64.A        lp=sp,off    // restore original lp register
          A.ADDI          sp@size      // deallocate caller stack frame
          B               lp           // return
              callee (leaf):
calLee:   ... (code using dp)
          B               lp           // return
```

Procedures that are compiled together may share a common data region, in which case there is no need to save, load, and restore the dp region in the callee, assuming that the callee does not modify the dp register. The pc-relative addressing of the B.LINK.I instruction permits the code region to be position-independent.

Minimum static-linked, intra-module calling sequence:

```
        caller (non-leaf):
caller:  A.COPY     r31=lp    // save original lp register
         ... (callee using same dp as caller)
         B.LINK.I   callee
         ...
         ... (callee using same dp as caller)
         B.LINK.I   callee
         ...
         B          r31       // return
        callee (leaf):
callee:  ... (code using dp, r31 unused)
         B          lp        // return
```

When all the callee procedures are intra-module, the stack frame may also be eliminated from the caller procedure by using "temporary" caller save registers not utilized by the callee leaf procedures. In addition to the lp value indicated above, this usage may include other values and variables that live in the caller procedure across callee procedure calls.

Typical dynamic-linked, inter-module calling sequence:

```
        caller (non-leaf):
caller: A.ADDI          sp@-size    // allocate caller stack frame
        S.I.64.A        lp,sp,off   // save original lp register
        S.I.64.A        dp,sp,off   // save original dp register
        ... (code using dp)
        L.I.64.A        lp=dp.off   // load lp
        L.I.64.A        dp=dp,off   // load dp
        B.LINK          lp=lp       // invoke callee procedure
        L.I.64.A        dp=sp,off   // restore dp register from stack
        ... (code using dp)
        L.I.64.A        lp=sp,off   // restore original lp register
        A.ADDI          sp=size     // deallocate caller stack frame
        B               lp          // return
        callee (leaf):
callee: ... (code using dp)
B       lp              // return
```

The load instruction is required in the caller following the procedure call to restore the dp register. A second load instruction also restores the lp register, which may be located at any point between the last procedure call and the branch instruction which returns from the procedure.

System and Privileged Library Calls

It is an objective to make calls to system facilities and privileged libraries as similar as possible to normal procedure calls as described above. Rather than invoke system calls as an exception, which involves significant latency and complication, we prefer to use a modified procedure call in which the process privilege level is quietly raised to the required level. To provide this mechanism safely, interaction with the virtual memory system is required.

Such a procedure must not be entered from anywhere other than its legitimate entry point, to prohibit entering a procedure after the point at which security checks are performed or with invalid register contents, otherwise the access to a higher privilege level can lead to a security violation. In addition, the procedure generally must have access to memory data, for which addresses must be produced by the privileged code. To

facilitate generating these addresses, the branch-gateway instruction allows the privileged code procedure to rely the fact that a single register has been verified to contain a pointer to a valid memory region.

The branch-gateway instruction ensures both that the procedure is invoked at a proper entry point, and that other registers such as the data pointer and stack pointer can be properly set. To ensure this, the branch-gateway instruction retrieves a "gateway" directly from the protected virtual memory space. The gateway contains the virtual address of the entry point of the procedure and the target privilege level. A gateway can only exist in regions of the virtual address space designated to contain them, and can only be used to access privilege levels at or below the privilege level at which the memory region can be written to ensure that a gateway cannot be forged.

The branch-gateway instruction ensures that register 1 (dp) contains a valid pointer to the gateway for this target code address by comparing the contents of register 0 (lp) against the gateway retrieved from memory and causing an exception trap if they do not match. By ensuring that register 1 points to the gateway, auxiliary information, such as the data pointer and stack pointer can be set by loading values located by the contents of register 1. For example, the eight bytes following the gateway may be used as a pointer to a data region for the procedure.

Referring to FIG. **54**G, before executing the branch-gateway instruction, register 1 must be set to point at the gateway, and register 0 must be set to the address of the target code address plus the desired privilege level. A "L.I.64.L.A r0=r1, 0" instruction is one way to set register 0, if register 1 has already been set, but any means of getting the correct value into register 0 is permissible.

Similarly, a return from a system or privileged routine involves a reduction of privilege. This need not be carefully controlled by architectural facilities, so a procedure may freely branch to a less-privileged code address. Normally, such a procedure restores the stack frame, then uses the branch-down instruction to return.

Typical dynamic-linked, inter-gateway calling sequence:

```
        caller:
caller: A.ADDI      sp@-size    // allocate caller stack frame
        S.I.64.A    lp,sp,off
        S.I.64.A    dp,sp,off
        ...
        L.I.64.A    lp=dp.off   // load lp
        L.I.64.A    dp=dp,off   // load dp
        B.GATE
        L.I.64.A    dp,sp,off
        ... (code using dp)
        L.I.64.A    lp=sp,off   // restore original lp register
        A.ADDI      sp=size     // deallocate caller stack frame
        B           lp          // return
        callee (non-leaf):
calee:  L.I.64.A    dp=dp,off   // load dp with data pointer
        S.I.64.A    sp,dp,off
        L.I.64.A    sp=dp,off   // new stack pointer
        S.I.64.A    lp,sp,off
        S.I.64.A    dp,sp,off
        ... (using dp)
        L.I.64.A    dp,sp,off
        ... (code using dp)
        L.I.64.A    lp=sp,off   // restore original lp register
        L.I.64.A    sp=sp,off   // restore original sp register
        B.DOWN      lp
        callee (leaf, no stack):
callee: ... (using dp)
        B.DOWN      lp
```

It can be observed that the calling sequence is identical to that of the inter-module calling sequence shown above, except for the use of the B.GATE instruction instead of a B.LINK instruction. Indeed, if a B.GATE instruction is used when the privilege level in the lp register is not higher than the current privilege level, the B.GATE instruction performs an identical function to a B.LINK.

The callee, if it uses a stack for local variable allocation, cannot necessarily trust the value of the sp passed to it, as it can be forged. Similarly, any pointers which the callee provides should not be used directly unless it they are verified to point to regions which the callee should be permitted to address. This can be avoided by defining application programming interfaces (APIs) in which all values are passed and returned in registers, or by using a trusted, intermediate privilege wrapper routine to pass and return parameters. The method described below can also be used.

It can be useful to have highly privileged code call less-privileged routines. For example, a user may request that errors in a privileged routine be reported by invoking a user-supplied error-logging routine. To invoke the procedure, the privilege can be reduced via the branch-down instruction. The return from the procedure actually requires an increase in privilege, which must be carefully controlled. This is dealt with by placing the procedure call within a lower-privilege procedure wrapper, which uses the branch-gateway instruction to return to the higher privilege region after the call through a secure re-entry point. Special care must be taken to ensure that the less-privileged routine is not permitted to gain unauthorized access by corruption of the stack or saved registers, such as by saving all registers and setting up a new stack frame (or restoring the original lower-privilege stack) that may be manipulated by the less-privileged routine. Finally, such a technique is vulnerable to an unprivileged routine attempting to use the re-entry point directly, so it may be appropriate to keep a privileged state variable which controls permission to enter at the re-entry point.

Referring first to FIG. **1**, a general purpose processor is illustrated therein in block diagram form. In FIG. **1**, four copies of an access unit are shown, each with an access instruction fetch queue A-Queue **101-104**. Each access instruction fetch queue A-Queue **101104** is coupled to an access register file AR **105-108**, which are each coupled to two access functional units A **109-116**. In a typical embodiment, each thread of the processor may have on the order of sixty-four general purpose registers (e.g., the AR's **105-108** and ER's **125-128**). The access units function independently for four simultaneous threads of execution, and each compute program control flow by performing arithmetic and branch instructions and access memory by performing load and store instructions. These access units also provide wide operand specifiers for wide operand instructions. These eight access functional units A **109-116** produce results for access register files AR **105-108** and memory addresses to a shared memory system **117-120**.

In one embodiment, the memory hierarchy includes on-chip instruction and data memories, instruction and data caches, a virtual memory facility, and interfaces to external devices. In FIG. **1**, the memory system is comprised of a combined cache and niche memory **117**, an external bus interface **118**, and, externally to the device, a secondary cache **119** and main memory system with I/O devices **120**. The memory contents fetched from memory system **117-120** are combined with execute instructions not performed by the access unit, and entered into the four execute instruction queues E-Queue **121-124**. In accordance with one embodiment of the present invention, from the software perspective, the machine state includes a linear byte-addressed shared memory space. For wide instructions, memory contents fetched from memory system **117-120** are also provided to wide operand microcaches **132-136** by bus **137**. Instructions and memory data from E-queue **121-124** are presented to execution register files **125-128**, which fetch execution register file source operands. The instructions are coupled to the execution unit arbitration unit Arbitration **131**, that selects which instructions from the four threads are to be routed to the available execution functional units E **141** and **149**, X **142** and **148**, G **143-144** and **146-147**, and T **145**. The execution functional units E **141** and **149**, the execution functional units X **142** and **148**, and the execution functional unit T **145** each contain a wide operand microcache **132-136**, which are each coupled to the memory system **117** by bus **137**.

The execution functional units G **143-144** and **146-147** are group arithmetic and logical units that perform simple arithmetic and logical instructions, including group operations wherein the source and result operands represent a group of values of a specified symbol size, which are partitioned and operated on separately, with results catenated together. In a presently preferred embodiment the data path is 128 bits wide, although the present invention is not intended to be limited to any specific size of data path.

The execution functional units X **142** and **148** are crossbar switch units that perform crossbar switch instructions. The crossbar switch units **142** and **148** perform data handling operations on the data stream provided over the data path source operand buses **151-158**, including deal, shuffles, shifts, expands, compresses, swizzles, permutes and reverses, plus the wide operations discussed hereinafter. In a key element of a first aspect of the invention, at least one such operation will be expanded to a width greater than the general register and data path width. Examples of the data manipulation operations are described in another section.

The execution functional units E **141** and **149** are ensemble units that perform ensemble instructions using a large array multiplier, including group or vector multiply and matrix multiply of operands partitioned from data path source operand buses **151-158** and treated as integer, floating-point, polynomial or Galois field values. According to the present embodiment of the invention, a general software solution is provided to the most common operations required for Galois Field arithmetic. The instructions provided include a polynomial multiply, with the polynomial specified as one register operand. This instruction can be used to perform CRC generation and checking, Reed-Solomon code generation and checking, and spread-spectrum encoding and decoding. Also, matrix multiply instructions and other operations described in another section utilize a wide operand loaded into the wide operand microcache **132** and **136**.

The execution functional unit T **145** is a translate unit that performs table-look-up operations on a group of operands partitioned from a register operand, and catenates the result. The Wide Translate instruction included in another section utilizes a wide operand loaded into the wide operand microcache **134**.

The execution functional units E **141**, **149**, execution functional units X-**142**, **148**, and execution functional unit T each contain dedicated storage to permit storage of source operands including wide operands as discussed hereinafter. The dedicated storage **132-136**, which may be thought of as a wide microcache, typically has a width which is a multiple of the width of the data path operands related to the data path source operand buses **151-158**. Thus, if the width of the data path **151-158** is 128 bits, the dedicated storage **132-136** may have a width of 256, 512, 1024 or 2048 bits. Operands which utilize the full width of the dedicated storage are referred to herein as wide operands, although it is not necessary in all instances that a wide operand use the entirety of the width of the dedicated storage; it is sufficient that the wide operand use a portion greater than the width of the memory data path of the output of the memory system **117-120** and the functional unit data path of the input of the execution functional units **141-149**, though not necessarily greater than the width of the two combined. Because the width of the dedicated storage **132-136** is greater than the width of the memory operand bus **137**, portions of wide operands are loaded sequentially into the dedicated storage **132-136**. However, once loaded, the wide operands may then be used at substantially the same time. It can be seen that functional units **141-149** and associated execution registers **125-128** form a data functional unit, the exact elements of which may vary with implementation.

The execution register file ER **125-128** source operands are coupled to the execution units **141-145** using source operand buses **151-154** and to the execution units **145-149** using source operand buses **155-158**. The function unit result operands from execution units **141145** are coupled to the execution register file ER **125-128** using result bus **161** and the function units result operands from execution units **145-149** are coupled to the execution register file using result bus **162**.

The wide operands used in some embodiments of the present invention provide the ability to execute complex instructions such as the wide multiply matrix instruction shown in FIG. **2**, which can be appreciated in an alternative form, as well, from FIG. **3**. As can be appreciated from FIGS. **2** and **3**, a wide operand permits, for example, the matrix multiplication of various sizes and shapes which exceed the data path width. The example of FIG. **2** involves a matrix specified by register rc having a 128*64/size multiplied by a vector contained in register rb having a 128 size, to yield a result, placed in register rd, of 128 bits.

The operands that are substantially larger than the data path width of the processor are provided by using a general-purpose register to specify a memory specifier from which more than one but in some embodiments several data path widths of data can be read into the dedicated storage. The memory specifier typically includes the memory address together with the size and shape of the matrix of data being operated on. The memory specifier or wide operand specifier can be better appreciated from FIG. **5**, in which a specifier **500** is seen to be an address, plus a field representative of the size/2 and a further field representative of width/2, where size is the product of the depth and width of the data. The address is aligned to a specified size, for example sixty-four bytes, so that a plurality of low order bits (for example, six bits) are zero. The specifier **500** can thus be seen to comprise a first field **505** for the address, plus two field indicia **510** within the low order six bits to indicate size and width.

The decoding of the specifier **500** may be further appreciated from FIG. **6** where, for a given specifier **600** made up of an address field **605** together with a field **610** comprising plurality of low order bits. By a series of arithmetic operations shown at steps **615** and **620**, the portion of the field **610** representative of width/2 is developed. In a similar series of steps shown at **625** and **630**, the value of t is decoded, which can then be used to decode both size and address. The portion of the field **610** representative of size/2 is decoded as shown at steps **635** and **640**, while the address is decoded in a similar way at steps **645** and **650**.

The wide function unit may be better appreciated from FIG. **7**, in which a register number **700** is provided to an operand checker **705**. Wide operand, specifier **710** communicates with the operand checker **705** and also addresses

memory **715** having a defined memory width. The memory address includes a plurality of register operands **720**A-n, which are accumulated in a dedicated storage portion **714** of a data functional unit **725**. In the exemplary embodiment shown in FIG. **7**, the dedicated storage **714** can be seen to have a width equal to eight data path widths, such that eight wide operand portions **730**A-H are sequentially loaded into the dedicated storage to form the wide operand. Although eight portions are shown in FIG. **7**, the present invention is not limited to eight or any other specific multiple of data path widths. Once the wide operand portions **730**A-H are sequentially loaded, they may be used as a single wide operand **735** by the functional element **740**, which may be any element(s) from FIG. **1** connected thereto. The result of the wide operand is then provided to a result register **745**, which in a presently preferred embodiment is of the same width as the memory width.

Once the wide operand is successfully loaded into the dedicated storage **714**, a second aspect of the present invention may be appreciated. Further execution of this instruction or other similar instructions that specify the same memory address can read the dedicated storage to obtain the operand value under specific conditions that determine whether the memory operand has been altered by intervening instructions. Assuming that these conditions are met, the memory operand fetch from the dedicated storage is combined with one or more register operands in the functional unit, producing a result. In some embodiments, the size of the result is limited to that of a general register, so that no similar dedicated storage is required for the result. However, in some different embodiments, the result may be a wide operand, to further enhance performance.

To permit the wide operand value to be addressed by subsequent instructions specifying the same memory address, various conditions must be checked and confirmed:

Those conditions include:

1. Each memory store instruction checks the memory address against the memory addresses recorded for the dedicated storage. Any match causes the storage to be marked invalid, since a memory store instruction directed to any of the memory addresses stored in dedicated storage **714** means that data has been overwritten.
2. The register number used to address the storage is recorded. If no intervening instructions have written to the register, and the same register is used on the subsequent instruction, the storage is valid (unless marked invalid by rule #1).
3. If the register has been modified or a different register number is used, the value of the register is read and compared against the address recorded for the dedicated storage. This uses more resources than #1 because of the need to fetch the register contents and because the width of the register is greater than that of the register number itself. If the address matches, the storage is valid. The new register number is recorded for the dedicated storage.

If conditions #2 or #3 are not met, the register contents are used to address the general-purpose processor's memory and load the dedicated storage. If dedicated storage is already fully loaded, a portion of the dedicated storage must be discarded (victimized) to make room for the new value. The instruction is then performed using the newly updated dedicated storage. The address and register number is recorded for the dedicated storage.

By checking the above conditions, the need for saving and restoring the dedicated storage is eliminated. In addition, if the context of the processor is changed and the new context does not employ Wide instructions that reference the same dedicated storage, when the original context is restored, the contents of the dedicated storage are allowed to be used without refreshing the value from memory, using checking rule #3. Because the values in the dedicated storage are read from memory and not modified directly by performing wide operations, the values can be discarded at any time without saving the results into general memory. This property simplifies the implementation of rule #4 above.

An alternate embodiment of the present invention can replace rule #1 above with the following rule:

1a. Each memory store instruction checks the memory address against the memory addresses recorded for the dedicated storage. Any match causes the dedicated storage to be updated, as well as the general memory.

By use of the above rule 1.a, memory store instructions can modify the dedicated storage, updating just the piece of the dedicated storage that has been changed, leaving the remainder intact. By continuing to update the general memory, it is still true that the contents of the dedicated memory can be discarded at any time without saving the results into general memory. Thus rule #4 is not made more complicated by this choice. The advantage of this alternate embodiment is that the dedicated storage need not be discarded (invalidated) by memory store operations.

Referring next to FIG. **9**, an exemplary arrangement of the data structures of the wide microcache or dedicated storage **114** may be better appreciated. The wide microcache contents, wmc.c, can be seen to form a plurality of data path widths **900**A-n, although in the example shown the number is eight. The physical address, wmc.pa, is shown as 64 bits in the example shown, although the invention is not limited to a specific width. The size of the contents, wmc.size, is also provided in a field which is shown as 10 bits in an exemplary embodiment. A "contents valid" flag, wmc.ev, of one bit is also included in the data structure, together with a two bit field for thread last used, or wmc.th. In addition, a six bit field for register last used, wmc.reg, is provided in an exemplary embodiment. Further, a one bit flag for register and thread valid, or wmc.rtv, may be provided.

The process by which the microcache is initially written with a wide operand, and thereafter verified as valid for fast subsequent operations, may be better appreciated from FIG. **8**. The process begins at **800**, and progresses to step **805** where a check of the register contents is made against the stored value wmc.rc. If true, a check is made at step **810** to verify the thread. If true, the process then advances to step **815** to verify whether the register and thread are valid. If step **815** reports as true, a check is made at step **820** to verify whether the contents are valid. If all of steps **805** through **820** return as true, the subsequent instruction is able to utilize the existing wide operand as shown at step **825**, after which the process ends. However, if any of steps **805** 'through **820** return as false, the process branches to step **830**, where content, physical address and size are set. Because steps **805** through **820** all lead to either step **825** or **830**, steps **805** through **820** may be performed in any order or simultaneously without altering the process. The process then advances to step **835** where size is checked. This check basically ensures that the size of the translation unit is greater than or equal to the size of the wide operand, so that a physical address can directly replace the use of a virtual address. The concern is that, in some embodiments, the wide operands may be larger than the minimum region that the virtual memory system is capable of mapping. As a result, it would be possible for a single contiguous virtual address range to be mapped into multiple, disjoint physical address ranges, complicating the task of comparing physical addresses. By determining the size of the wide operand and comparing that size against the size of the virtual address mapping region which is referenced, the instruction is aborted with an exception trap if the wide operand is larger than the mapping region. This ensures secure operation of the processor. Software can then re-map the region using a larger size map to continue execution if desired. Thus, if size is reported as unacceptable at step **835**, an exception is generated at step **840**. If size is acceptable, the process advances to step **845** where physical address is checked. If the check reports as met, the process advances to step **850**, where a check of the contents valid flag is made. If either check at step **845** or **850** reports as false, the process branches and new content is written into the dedicated storage **114**, with the fields thereof being set accordingly. Whether the check at step **850** reported true, or whether new content was written at step **855**, the process advances to step **860** where appropriate fields are set to indicate the validity of the data, after which the requested function can be performed at step **825**. The process then ends.

Referring next to FIGS. **10** and **11**, which together show the operation of the microcache controller from a hardware standpoint, the operation of the microcache controller may be better understood. In the hardware implementation, it is clear that conditions which are indicated as sequential steps in FIGS. **8** and **9** above can be performed in parallel, reducing the delay for such wide operand checking. Further, a copy of the indicated hardware may be included for each wide microcache, and thereby all such microcaches as may be alternatively referenced by an instruction can be tested in parallel. It is believed that no further discussion of FIGS. **10** and **11** is required in view of the extensive discussion of FIGS. **8** and **9**, above.

Various alternatives to the foregoing approach do exist for the use of wide operands, including an implementation in which a single instruction can accept two wide operands, partition the operands into symbols, multiply corresponding symbols together, and add the products to produce a single scalar value or a vector of partitioned values of width of the register file, possibly after extraction of a portion of the sums. Such an instruction can be valuable for detection of motion or estimation of motion in video compression. A further enhancement of such an instruction can incrementally update the dedicated storage if the address of one wide operand is within the range of previously specified wide operands in the dedicated storage, by loading only the portion not already within the range and shifting the in-range portion as required. Such an enhancement allows the operation to be performed over a "sliding window" of possible values. In such an instruction, one wide operand is aligned and supplies the size and shape information, while the second wide operand, updated incrementally, is not aligned.

Another alternative embodiment of the present invention can define additional instructions where the result operand is a wide operand. Such an enhancement removes the limit that a result can be no larger than the size of a general register, further enhancing performance. These wide results can be cached locally to the functional unit that created them, but must be copied to the general memory system before the storage can be reused and before the virtual memory system alters the mapping of the address of the wide result. Data paths must be added so that load operations and other wide operations can read these wide results—forwarding of a wide result from the output of a functional unit back to its input is relatively easy, but additional data paths may have to be introduced if it is desired to forward wide results back to other functional units as wide operands.

As previously discussed, a specification of the size and shape of the memory operand is included in the low-order bits of the address. In a presently preferred implementation, such memory operands are typically a power of two in size and aligned to that size. Generally, one-half the total size is added (or inclusively or'ed, or exclusively or'ed) to the memory address, and one half of the data width is added (or inclusively or'ed, or exclusively or'ed) to the memory address. These bits can be decoded and stripped from the memory address, so that the controller is made to step through all the required addresses. This decreases the number of distinct operands required for these instructions, as the size, shape and address of the memory operand are combined into a single register operand value.

Particular examples of wide operations which are defined by the present invention include the Wide Switch instruction that performs bit-level switching; the Wide Translate instruction which performs byte (or larger) table-lookup; Wide Multiply Matrix, Wide Multiply Matrix Extract and Wide Multiply Matrix Extract Immediate (discussed below), Wide Multiply Matrix Floating-point, and Wide Multiply Matrix Galois (also discussed below). While the discussion below focuses on particular sizes for the exemplary instructions, it will be appreciated that the invention is not limited to a particular width.

The Wide Switch instruction rearranges the contents of up to two registers (256 bits) at the bit level, producing a full-width (128 bits) register result. To control the rearrangement, a wide operand specified by a single register, consisting of eight bits per bit position is used. For each result bit position, eight wide operand bits for each bit position select which of the 256 possible source register bits to place in the result. When a wide operand size smaller than 128 bytes, the high order bits of the memory operand are replaced with values corresponding to the result bit position, so that the memory operand specifies a bit selection within symbols of the operand size, performing the same operation on each symbol.

The Wide Translate instructions use a wide operand to specify a table of depth up to 256 entries and width of up to 128 bits. The contents of a register is partitioned into operands of one, two, four, or eight bytes, and the partitions are used to select values from the table in parallel. The depth and width of the table can be selected by specifying the size and shape of the wide operand as described above.

The Wide Multiply Matrix instructions use a wide operand to specify a matrix of values of width up to 64 bits (one half of register file and data path width) and depth of up to 128 bits/symbol size. The contents of a general register (128 bits) is used as a source operand, partitioned into a vector of symbols, and multiplied with the matrix, producing a vector of width up to 128 bits of symbols of twice the size of the source operand symbols. The width and depth of the matrix can be selected by specifying the size and shape of the wide operand as described above. Controls within the instruction allow specification of signed, mixed-signed, unsigned, complex, or polynomial operands.

The Wide Multiply Matrix Extract instructions use a wide operand to specify a matrix of value of width up to 128 bits (full width of register file and data path) and depth of up to 128 bits/symbol size. The contents of a general register (128 bits) is used as a source operand, partitioned into a vector of symbols, and multiplied with the matrix, producing a vector of width up to 256 bits of symbols of twice the size of the source operand symbols plus additional bits to represent the sums of products without overflow. The results are then extracted in a manner described below (Enhanced Multiply Bandwidth by Result Extraction), as controlled by the contents of a general register specified by the instruction. The general register also specifies the format of the operands: signed, mixed-signed, unsigned, and complex as well as the size of the operands, byte (8 bit), doublet (16 bit), quadlet (32 bit), or hexlet (64 bit).

The Wide Multiply Matrix Extract Immediate instructions perform the same function as above, except that the extraction, operand format and size is controlled by fields in the instruction. This form encodes common forms of the above instruction without the need to initialize a register with the required control information. Controls within the instruction allow specification of signed, mixed-signed, unsigned, and complex operands.

The Wide Multiply Matrix Floating-point instructions perform a matrix multiply in the same form as above, except that the multiplies and additions are performed in floating-point arithmetic. Sizes of half (16-bit), single (32-bit), double (64-bit), and complex sizes of half, single and double can be specified within the instruction.

Wide Multiply Matrix Galois instructions perform a matrix multiply in the same form as above, except that the multiples and additions are performed in Galois field arithmetic. A size of 8 bits can be specified within the instruction. The contents of a general register specify the polynomial with which to perform the Galois field remainder operation. The nature of the matrix multiplication is novel and described in detail below.

In another aspect of the invention, memory operands of either little-endian or big-endian conventional byte ordering are facilitated. Consequently, all Wide operand instructions are specified in two forms, one for little-endian byte ordering and one for big-endian byte ordering, as specified by a portion of the instruction. The byte order specifies to the memory system the order in which to deliver the bytes within units of the data path width (128 bits), as well as the order to place multiple memory words (128 bits) within a larger Wide operand. Each of these instructions is described in greater detail.

Some embodiments of the present invention address extraction of a high order portion of a multiplier product or sum of products, as a way of efficiently utilizing a large multiplier array. Parent U.S. Pat. Nos. 5,742,840 and 5,953,241 describe a system and method for enhancing the utilization of a multiplier array by adding specific classes of instructions to a general-purpose processor. This addresses the problem of making the most use of a large multiplier array that is fully used for high-precision arithmetic—for example a 64.times.64 bit multiplier is fully used by a 64-bit by 64-bit multiply, but only one quarter used for a 32-bit by 32-bit multiply) for (relative to the multiplier data width and registers) low-precision arithmetic operations. In particular, operations that perform a great many low-precision multiplies which are combined (added) together in various ways are specified. One of the overriding considerations in selecting the set of operations is a limitation on the size of the result operand. In an exemplary embodiment, for example, this size might be limited to on the order of 128 bits, or a single register, although no specific size limitation need exist.

The size of a multiply result, a product, is generally the sum of the sizes of the operands, multiplicands and multiplier. Consequently, multiply instructions specify operations in which the size of the result is twice the size of identically-sized input operands. For our prior art design, for example, a multiply instruction accepted two 64-bit register sources and produces a single 128-bit register-pair result, using an entire 64.times.64 multiplier array for 64-bit symbols, or half the multiplier array for pairs of 32-bit symbols, or one-quarter the multiplier array for quads of 16-bit symbols. For all of these cases, note that two register sources of 64 bits are combined, yielding a 128-bit result.

In several of the operations, including complex multiplies, convolve, and matrix multiplication, low-precision multiplier products are added together. The additions further increase the required precision. The sum of two products requires one additional bit of precision; adding four products requires two, adding eight products requires three, adding sixteen products requires four. In some prior designs, some of this precision is lost, requiring scaling of the multiplier operands to avoid overflow, further reducing accuracy of the result.

The use of register pairs creates an undesirable complexity, in that both the register pair and individual register values must be bypassed to subsequent instructions. As a result, with prior art techniques only half of the source operand 128-bit register values could be employed toward producing a single-register 128-bit result.

In some embodiments of the present invention, a high-order portion of the multiplier product or sum of products is extracted, adjusted by a dynamic shift amount from a general register or an adjustment specified as part of the instruction, and, rounded by a control value from a register or instruction portion as round-to-nearest/even, toward zero, floor, or ceiling. Overflows are handled by limiting the result to the largest and smallest values that can be accurately represented in the output result.

In the present invention, when the extract is controlled by a register, the size of the result can be specified, allowing rounding and limiting to a smaller number of bits than can fit in the result. This permits the result to be scaled to be used in subsequent operations without concern of overflow or rounding, enhancing performance.

Also in the present invention, when the extract is controlled by a register, a single register value defines the size of the operands, the shift amount and size of the result, and the rounding control. By placing all this control information in a single register, the size of the instruction is reduced over the number of bits that such a instruction would otherwise require, improving performance and enhancing flexibility of the processor.

The particular instructions included in this aspect of the present invention are Ensemble Convolve Extract, Ensemble Multiply Extract, Ensemble Multiply Add Extract and Ensemble Scale Add Extract, each of which is more thoroughly treated in another section.

An aspect of the present invention defines the Ensemble Scale Add Extract instruction, that combines the extract control information in a register along with two values that are used as scalar multipliers to the contents of two vector multiplicands. This combination reduces the number of registers that would otherwise be required, or the number of bits that the instruction would otherwise require, improving performance.

Several of these instructions (Ensemble Convolve Extract, Ensemble Multiply Add Extract) are typically available only in forms where the extract is specified as part of the instruction. An alternative embodiment can incorporate forms of the operations in which the size of the operand, the shift amount and the rounding can be controlled by the contents of a general register (as they are in the Ensemble Multiply Extract instruction). The definition of this kind of instruction for Ensemble Convolve Extract, and Ensemble Multiply Add Extract would require four source registers, which increases complexity by requiring additional general-register read ports.

Another alternative embodiment can reduce the number of register read ports required for implementation of instructions in which the size, shift and rounding of operands is controlled by a register. The value of the extract control register can be fetched using an additional cycle on an initial execution and retained within or near the functional unit for subsequent executions, thus reducing the amount of hardware required for implementation with a small additional performance penalty. The value retained would be marked invalid, causing a re-fetch of the extract control register, by instructions that modify the register, or alternatively, the retained value can be updated by such an operation. A re-fetch of the extract control register would also be required if a different register number were specified on a subsequent execution. It should be clear that the properties of the above two alternative embodiments can be combined.

Another embodiment of the invention includes Galois field arithmetic, where multiplies are performed by an initial binary polynomial multiplication (unsigned binary multiplication with carries suppressed), followed by a polynomial modulo/remainder operation (unsigned binary division with carries suppressed). The remainder operation is relatively expensive in area and delay. In Galois field arithmetic, additions are performed by binary addition with carries suppressed, or equivalently, a bitwise exclusive-or operation. In this aspect of the present invention, a matrix multiplication is performed using Galois field arithmetic, where the multiplies and additions are Galois field multiples and additions.

Using prior art methods, a 16 byte vector multiplied by a 16.times. 16 byte matrix can be performed as 256 8-bit Galois field multiplies and 16*15=240 8-bit Galois field additions. Included in the 256 Galois field multiplies are 256 polynomial multiplies and 256 polynomial remainder operations. But by use of the present invention, the total computation can be reduced significantly by performing 256 polynomial multiplies, 240 16-bit polynomial additions, and 16 polynomial remainder operations. Note that the cost of the polynomial additions has been doubled, as these are now 16-bit operations, but the cost of the polynomial remainder functions has been reduced by a factor of 16. Overall, this is a favorable tradeoff, as the cost of addition is much lower than the cost of remainder.

In a still further aspect of the present invention, a technique is provided for incorporating floating point information into processor instructions. In U.S. Pat. No. 5,812,439, a system and method are described for incorporating control of rounding and exceptions for floating-point instructions into the instruction itself. The present invention extends this invention to include separate instructions in which rounding is specified, but default handling of exceptions is also specified, for a particular class of floating-point instructions. Specifically, the SINK instruction (which converts floating-point values to integral values) is available with control in the instruction that include all previously specified combinations (default-near rounding and default exceptions, Z—round-toward-zero and trap on exceptions, N—round to nearest and trap on exceptions, F—floor rounding (toward minus infinity) and trap on exceptions, C—ceiling rounding (toward plus infinity) and trap on exceptions, and X—trap on inexact and other exceptions), as well as three new combinations (Z.D—round toward zero and default exception handling, F.D—floor rounding and default exception handling, and C.D—ceiling rounding and default exception handling). (The other combinations: N.D is equivalent to the default, and X.D—trap on inexact but default handling for other exceptions is possible but not particularly valuable).

Instruction Scheduling

The next section describes detailed pipeline organization for Zeus, which has a significant influence on instruction scheduling. Here we will elaborate some general rules for effective scheduling by a compiler. Specific information on numbers of functional units, functional unit parallelism and latency is quite implementation-dependent, values indicated here are valid for Zeus's first implementation.

Separate Addressing from Execution

Zeus has separate function units to perform addressing operations (A, L, S, B instructions) from execution operations (G, X, E, W instructions). When possible, Zeus will execute all the addressing operations of an instruction stream, deferring execution of the execution operations until dependent load instructions are completed. Thus, the latency of the memory system is hidden, so long as addressing operations themselves do not need to wait for memory.

Software Pipeline

Instructions should generally be scheduled so that previous operations can be completed at the time of issue. When this is not possible, the processor inserts sufficient empty cycles to perform the instructions precisely—explicit no-operation instructions are not required.

Multiple Issue

Zeus can issue up to two addressing operations and up to two execution operations per cycle per thread. Considering functional unit parallelism, described below, as many of four instruction issues per cycle are possible per thread.

Functional Unit parallelism

Zeus has separate function units for several classes of execution operations. An A unit performs scalar add, subtract, boolean, and shift-add operations for addressing and branch calculations. The remaining functional units are execution resources, which perform operations subsequent to memory loads and which operate on values in a parallel, partitioned form. A G unit performs add, subtract, boolean, and shift-add operations. An X unit performs general shift operations. An E unit performs multiply and floating-point operations. A T unit performs table-look-up operations.

Each instruction uses one or more of these units, according to the table below.

Instruction	A	G	X	E	T
A.	x				
B	x				
L	x				
S	x				
G		x			
X			x		
E			x	x	
W.TRANSLATE	x				x
W.MULMAT	x		x	x	
W.SWITCH	x		x		

Latency

The latency of each functional unit depends on what operation is performed in the unit, and where the result is used. The aggressive nature of the pipeline makes it difficult to characterize the latency of each operation with a single number. Because the addressing unit is decoupled from the execution unit, the latency of load operations is generally hidden, unless the result of a load instruction must be returned to the addressing unit. Store instructions must be able to compute the address to which the data is to be stored in the addressing unit, but the data will not be irrevocably stored until the data is available and it is valid to retire the store instruction. However, under certain conditions, data may be forwarded from a store instruction to subsequent load instructions, once the data is available.

The latency of each of these units, for the initial Zeus implementation is indicated below:

Unit	instruction	Latency rules
A.	A	1 cycle
	L	Address operands must be ready to issue, 4 cycles to A unit, 0 to G, X, E, T units
	S	Address operands must be ready to issue, Store occurs when data is ready and instruction may be retired.
	B	Conditional branch operands may be provided from the A unit (64-bit values), or the G unit (128-bit values). 4 cycles for mispredicted branch
	W	Address operand must be ready to issue,
G	G	1 cycle
X	X, W.SWITCH	1 cycle for data operands, 2 cycles for shift amount or control operand
E	E, W.MULMAT	4 cycles
T	W.TRANSLATE	1 cycles

Pipelining and Multithreading

As shown in FIG. **4**, some embodiments of the present invention employ both decoupled access from execution pipelines and simultaneous multithreading in a unique way. Simultaneous Multithreaded pipelines have been employed in prior art to enhance the utilization of data path units by allowing instructions to be issued from one of several execution threads to each functional unit (e.g., Susan Eggers, University of Wash, papers on Simultaneous Multithreading).

Decoupled access from execution pipelines have been employed in prior art to enhance the utilization of execution data path units by buffering results from an access unit, which computes addresses to a memory unit that in turn fetches the requested items from memory, and then presenting them to an execution unit (e.g., James E. Smith, paper on Decoupled Access from Execution).

Compared to conventional pipelines, Eggers prior art used an additional pipeline cycle before instructions could be issued to functional units, the additional cycle needed to determine which threads should be permitted to issue instructions. Consequently, relative to conventional pipelines, the prior art design had additional delay, including dependent branch delay.

The embodiment shown in FIG. **4** contains individual access data path units, with associated register files, for each execution thread. These access units produce addresses, which are aggregated together to a common memory unit, which fetches all the addresses and places the memory contents in one or more buffers. Instructions for execution units, which are shared to varying degrees among the threads are also buffered for later execution. The execution units then perform operations from all active threads using functional data path units that are shared.

For instructions performed by the execution units, the extra cycle required for prior art simultaneous multithreading designs is overlapped with the memory data access time from prior art decoupled access from execution cycles, so that no additional delay is incurred by the execution functional units for scheduling resources. For instructions performed by the access units, by employing individual access units for each thread the additional cycle for scheduling shared resources is also eliminated.

This is a favorable tradeoff because, while threads do not share the access functional units, these units are relatively small compared to the execution functional units, which are shared by threads.

FIG. **12** is a timing diagram of a decoupled pipeline structure in accordance with one embodiment of the present invention. As illustrated in FIG. **12**, the time permitted by a pipeline to service load operations may be flexibly extended. Here, various types of instructions are abbreviated as A, L, B, E, and S, representing a register-to-register address calculation, a memory load, a branch, a register-to-register data calculation, and a memory store, respectively. According to the present embodiment, the front of the pipeline, in which A, L and B type instructions are handled, is decoupled from the back of the pipeline, in which E, and S type instructions are handled. This decoupling occurs at the point at which the data cache and its backing memory is referenced; similarly, a FIFO that is filled by the instruction fetch unit decouples instruction cache references from the front of the pipeline shown above. The depth of the FIFO structures is implementation-dependent, i.e. not fixed by the architecture. FIG. **13** further illustrates this pipeline organization. Accordingly, the latency of load instructions can be hidden, as execute instructions are deferred until the results of the load are available. Nevertheless, the execution unit still processes instructions in normal order, and provides precise exceptions. More details relating to this pipeline structure is explained in the "Superspring Pipeline" section.

A difficulty in particular pipeline structures is that dependent operations must be separated by the latency of the pipeline, and for highly pipelined machines, the latency of simple operations can be quite significant. According to one embodiment of the present invention, very highly pipelined implementations are provided by alternating execution of two or more independent threads. In an embodiment, a thread is the state required to maintain an independent execution; the architectural state required is that of the register file contents, program counter, privilege level, local TB, and when required, exception status. In an embodiment, ensuring that only one thread may handle an exception at one time may minimize the latter state, exception status. In order to ensure that all threads make reasonable forward progress, several of the machine resources must be scheduled fairly.

An example of a resource that is critical that it be fairly shared is the data memory/cache subsystem. In one embodiment, the processor may be able to perform a load operation only on every second cycle, and a store operation only on every fourth cycle. The processor schedules these fixed timing resources fairly by using a round-robin schedule for a number of threads that is relatively prime to the resource reuse rates. In one embodiment, five simultaneous threads of execution ensure that resources which may be used every two or four cycles are fairly shared by allowing the instructions which use those resources to be issued only on every second or fourth issue slot for that thread. More details relating to this pipeline structure are explained in the "Superthread Pipeline" section.

Referring back to FIG. **4**, with regard to the sharing of execution units, one embodiment of the present invention employs several different classics of functional units for the execution unit, with varying cost, utilization, and performance. In particular, the G units, which perform simple addition and bitwise operations is relatively inexpensive (in area and power) compared to the other units, and its utilization is relatively high. Consequently, the design employs four such units, where each unit can be shared between two threads. The X unit, which performs a broad class of data switching functions is more expensive and less used, so two units are provided that are each shared among two threads. The T unit, which performs the Wide Translate instruction, is expensive and utilization is low, so the single unit is shared among all four threads. The E unit, which performs the class of Ensemble instructions, is very expensive in area and power compared to the other functional units, but utilization is relatively high, so we provide two such units, each unit shared by two threads.

In FIG. **4**, four copies of an access unit are shown, each with an access instruction fetch queue A-Queue **401-404**, coupled to an access register file AR **405-408**, each of which is, in turn, coupled to two access functional units A **409-416**. The access units function independently for four simultaneous threads of execution. These eight access functional units A **409-416** produce results for access register files AR **405-408** and addresses to a shared memory system **417**. The memory contents fetched from memory system **417** are combined with execute instructions not performed by the access unit and entered into the four execute instruction queues E-Queue **421-424**. Instructions and memory data from E-queue **421-424** are presented to execution register files **425-428**, which fetches execution register file source operands. The instructions are coupled to the execution unit arbitration unit Arbitration **431**, that selects which instructions from the four threads are to be routed to the available execution units E **441** and **449**, X **442** and **448**, G **443-444** and **446-447**, and T **445**. The execution register file source operands ER **425-428** are coupled to the execution units **441-445** using source operand buses **451-454** and to the execution units **445-449** using source operand buses **455-458**. The function unit result operands from execution units **441-445** are coupled to the execution register file using result bus **461** and the function units result operands from execution units **445-449** are coupled to the execution register file using result bus **462**.

In a still further aspect of the present invention, an improved interprivilege gateway is described which involves increased parallelism and leads to enhanced performance. In U.S. application Ser. No. 08/541,416, now U.S. Pat. No. 6,101,590, a system and method is described for implementing an instruction that, in a controlled fashion, allows the transfer of control (branch) from a lower-privilege level to a higher-privilege level. Embodiment of the present invention provides an improved system and method for a modified instruction that accomplishes the same purpose but with specific advantages.

Many processor resources, such as control of the virtual memory system itself, input and output operations, and system control functions are protected from accidental or malicious misuse by enclosing them in a protective, privileged region. Entry to this region must be established only though particular entry points, called gateways, to maintain the integrity of these protected regions.

Prior art versions of this operation generally load an address from a region of memory using a protected virtual memory attribute that is only set for data regions that contain valid gateway entry points, then perform a branch to an address contained in the contents of memory. Basically, three steps were involved: load, branch, then check. Compared to other instructions, such as register-to-register computation instructions and memory loads and stores, and register-based branches, this is a substantially longer operation, which introduces delays and complexity to a pipelined implementation.

In the present invention, the branch-gateway instruction performs two operations in parallel: 1) a branch is performed to the contents of register 0 and 2) a load is performed using the contents of register 1, using a specified byte order (little-endian) and a specified size (64 bits). If the value loaded from memory does not equal the contents of register 0, the instruction is aborted due to an exception. In addition, 3) a return address (the next sequential instruction address following the branch-gateway instruction) is written into register 0, provided the instruction is not aborted. This approach essentially uses a first instruction to establish the requisite permission to allow user code to access privileged code, and then a second instruction is permitted to branch directly to the privileged code because of the permissions issued for the first instruction.

In the present invention, the new privilege level is also contained in register 0, and the second parallel operation does not need to be performed if the new privilege level is not greater than the old privilege level. When this second operation is suppressed, the remainder of the instruction performs an identical function to a branch-link instruction, which is used for invoking procedures that do not require an increase in privilege. The advantage that this feature brings is that the branch-gateway instruction can be used to call a procedure that may or may not require an increase in privilege.

The memory load operation verifies with the virtual memory system that the region that is loaded has been tagged as containing valid gateway data. A further advantage of the present invention is that the called procedure may rely on the fact that register 1 contains the address that the gateway data was loaded from, and can use the contents of register 1 to locate additional data or addresses that the procedure may require. Prior art versions of this instruction required that an additional address be loaded from the gateway region of memory in order to initialize that address in a protected manner—the present invention allows the address itself to be loaded with a "normal" load operation that does not require special protection.

The present invention allows a "normal" load operation to also load the contents of register 0 prior to issuing the branch-gateway instruction. The value may be loaded from the same memory address that is loaded by the branch-gateway instruction, because the present invention contains a virtual memory system in which the region may be enabled for normal load operations as well as the special "gateway" load operation performed by the branch-gateway instruction.

In a further aspect of the present invention, a system and method is provided for performing a three-input bitwise Boolean operation in a single instruction. A novel method described in detail in another section is used to encode the eight possible output states of such an operation into only seven bits, and decoding these seven bits back into the eight states.

In yet a further aspect to the present invention, a system and method is described for improving the branch prediction of simple repetitive loops of code. The method includes providing a count field for indicating how many times a branch is likely to be taken before it is not taken, which enhances the ability to properly predict both the initial and final branches of simple loops when a compiler can determine the number of iterations that the loop will be performed. This improves performance by avoiding misprediction of the branch at the end of a loop.

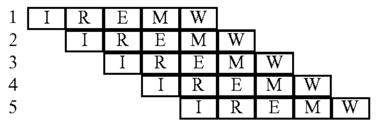
Pipeline Organization

Zeus performs all instructions as if executed one-by-one, in-order, with precise exceptions always available. Consequently, code that ignores the subsequent discussion of Zeus pipeline implementations will still perform correctly. However, the highest performance of the Zeus processor is achieved only by matching the ordering of instructions to the characteristics of the pipeline. In the following discussion, the general characteristics of all Zeus implementations precede discussion of specific choices for specific implementations.

Classical Pipeline Structures

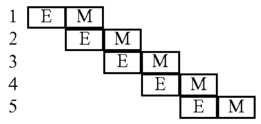
Pipelining in general refers to hardware structures that overlap various stages of execution of a series of instructions so that the time required to perform the series of instructions is less than the sum of the times required to perform each of the instructions separately. Additionally, pipelines carry to connotation of a collection of hardware structures which have a simple ordering and where each structure performs a specialized function.

The diagram below shows the timing of what has become a canonical pipeline structure for a simple RISC processor, with time on the horizontal axis increasing to the right, and successive instructions on the vertical axis going downward. The stages I, R, E, M, and W refer to units which perform instruction fetch, register file fetch, execution, data memory fetch, and register file write. The stages are aligned so that the result of the execution of an instruction may be used as the source of the execution of an immediately following instruction, as seen by the fact that the end of an E stage (bold in line 1) lines up with the beginning of the E stage (bold in line 2) immediately below. Also, it can be seen that the result of a load operation executing in stages E and M (bold in line 3) is not available in the immediately following instruction (line 4), but may be used two cycles later (line 5); this is the cause of the load delay slot seen on some RISC processors.



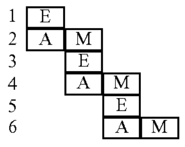
canonical pipeline

In the diagrams below, we simplify the diagrams somewhat by eliminating the pipe stages for instruction fetch, and register file write, which can be understood to precede and follow the pipelines diagrammed. The diagram above is shown again in this new format, showing that the canonical pipeline has very little overlap of the actual execution of instructions.



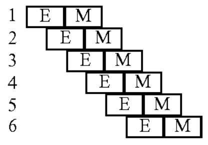
canonical pipeline

A superscalar pipeline is one capable of simultaneously issuing two or more instructions which are independent, in that they can be executed in either order and separately, producing the same result as if they were executed serially. The diagram below shows a two-way superscalar processor, where one instruction may be a register-to-register operation (using stage E) and the other may be a register operation (using stage A) or a memory load or store (using states A and M).



superscalar pipeline

A superpipelined pipeline is one capable is issuing simple instructions frequently enough that the result of a simple instruction must be independent of the immediately following one or more instructions. The diagram below shows a two-cycle superpipelined implementation:



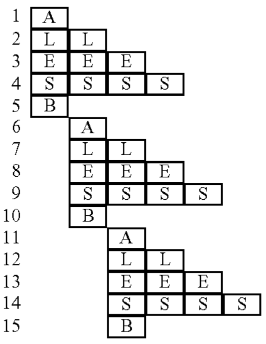
superpipelined pipeline

In the diagrams below, pipeline stages are labelled with the type of instruction that may be performed by that stage. The position of the stage further identifies the function of that stage, as for example a load operation may require several L stages to complete the instruction.

Superstring Pipeline

Zeus architecture provides for implementations designed to fetch and execute several instructions in each clock cycle. For a particular ordering of instruction types, one instruction of each type may be issued in a single clock cycle. The ordering required is A, L, E, S, B; in other words, a register-to-register address calculation, a memory load, a register-to-register data calculation, a memory store, and a branch. Because of the organization of the pipeline, each of these instructions may be serially dependent. Instructions of type E include the fixed-point execute-phase instructions as well as floating-point and digital signal processing instructions. We call this form of pipeline organization "superstring," (readers with a background in theoretical physics may have seen this term in an other, unrelated, context) because of the ability to issue a string of dependent instructions in a single clock cycle, as distinguished from superscalar or superpipelined organizations, which can only issue sets of independent instructions.

These instructions take from one to four cycles of latency to execute, and a branch prediction mechanism is used to keep the pipeline filled. The diagram below shows a box for the interval between issue of each instruction and the completion. Bold letters mark the critical latency paths of the instructions, that is, the periods between the required availability of the source registers and the earliest availability of the result registers. The A-L critical latency path is a special case, in which the result of the A instruction may be used as the base register of the L instruction without penalty. E instructions may require additional cycles of latency for certain operations, such as fixed-point multiply and divide, floating-point and digital signal processing operations.



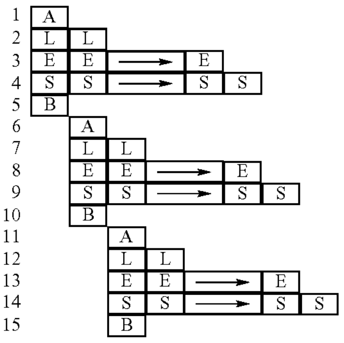
Superstring pipeline

Superspring Pipeline

Zeus architecture provides an additional refinement to the organization defined above, in which the time permitted by the pipeline to service load operations may be flexibly extended. Thus, the front of the pipeline, in which A, L and B type instructions are handled, is decoupled from the back of the pipeline, in which E, and S type instructions are handled. This decoupling occurs at the point at which the data cache and its backing memory is referenced; similarly, a FIFO that is filled by the instruction fetch unit decouples instruction cache references from the front of the pipeline shown above. The depth of the FIFO structures is implementation-dependent, i.e. not fixed by the architecture.

FIG. **13** indicates why we call this pipeline organization feature "superspring," an extension of our superstring organization.

With the super-spring organization, the latency of load instructions can be hidden, as execute instructions are deferred until the results of the load are available. Nevertheless, the execution unit still processes instructions in normal order, and provides precise exceptions.



Superspring pipeline

Superthread Pipeline

This technique is not employed in the initial Zeus implementation, though it was present in an earlier prototype implementation.

A difficulty of superpipelining is that dependent operations must be separated by the latency of the pipeline, and for highly pipelined machines, the latency of simple operations can be quite significant. The Zeus "superthread" pipeline provides for very highly pipelined implementations by alternating execution of two or more independent threads. In this context, a thread is the state required to maintain an independent execution; the architectural state required is that of the register file contents, program counter, privilege level, local TB, and when required, exception status. Ensuring that only one thread may handle an exception at one time may minimize the latter state, exception status. In order to ensure that all threads make reasonable forward progress, several of the machine resources must be scheduled fairly.

An example of a resource that is critical that it be fairly shared is the data memory/cache subsystem. In a prototype implementation, Zeus is able to perform a load operation only on every second cycle, and a store operation only on every fourth cycle. Zeus schedules these fixed timing resources fairly by using a round-robin schedule for a number of threads that is relatively prime to the resource reuse rates. For this implementation, five simultaneous threads of execution ensure that resources which may be used every two or four cycles are fairly shared by allowing the instructions which use those resources to be issued only on every second or fourth issue slot for that thread.

In the diagram below, the thread number which issues an instruction is indicated on each clock cycle, and below it, a list of which functional units may be used by that instruction. The diagram repeats every 20 cycles, so cycle 20 is similar to cycle 0, cycle 21 is similar to cycle 1, etc. This schedule ensures that no resource conflict occur between threads for these resources. Thread 0 may issue an E, L, S or B on cycle 0, but on its next opportunity, cycle 5, may only issue E or B, and on cycle 10 may issue E, L or B, and on cycle 15, may issue E or B.

a common memory system, a common T unit. Pairs of threads share two G units, one X unit, and one E unit. Each thread individually has two A units. A fair allocation scheme balances access to the shared resources by the four threads.

Branch/Fetch Prediction

Zeus does not have delayed branch instructions, and so relies upon branch or fetch prediction to keep the pipeline full around unconditional and conditional branch instructions. In the simplest form of branch prediction, as in Zeus's first implementation, a taken conditional backward (toward a lower address) branch predicts that a future execution of the same branch will be taken. More elaborate prediction may cache the source and target addresses of multiple branches, both conditional and unconditional, and both forward and reverse.

The hardware prediction mechanism is tuned for optimizing conditional branches that close loops or express frequent alternatives, and will generally require substantially more cycles when executing conditional branches whose outcome is not predominately taken or not-taken. For such cases of unpredictable conditional results, the use of code that avoids conditional branches in favor of the use of compare-set and multiplex instructions may result in greater performance.

Under some conditions, the above technique may not be applicable, for example if the conditional branch "guards" code which cannot be performed when the branch is taken. This may occur, for example, when a conditional branch tests for a valid (non-zero) pointer and the conditional code performs a load or store using the pointer. In these cases, the

Superthread pipeline

cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
thread	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4
	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
	L		L		L		L		L		L		L		L		L		L	
	S				S				S				S				S			
	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B

When seen from the perspective of an individual thread, the resource use diagram looks similar to that of the collection. Thus an individual thread may use the load unit every two instructions, and the store unit every four instructions.

conditional branch has a small positive offset, but is unpredictable. A Zeus pipeline may handle this case as if the branch is always predicted to be not taken, with the recovery of a misprediction causing cancellation of the instructions which

Superthread pipeline

cycle	0	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95
thread	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
	L		L		L		L		L		L		L		L		L		L	
	S				S				S				S				S			
	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B

A Zeus Superthread pipeline, with 5 simultaneous threads of execution, permits simple operations, such as register-to-register add (G.ADD), to take 5 cycles to complete, allowing for an extremely deeply pipelined implementation.

Simultaneous Multithreading

The initial Zeus implementation performs simultaneous multithreading among 4 threads. Each of the 4 threads share

have already been issued but not completed which would be skipped over by the taken conditional branch. This "conditional-skip" optimization is performed by the initial Zeus implementation and requires no specific architectural feature to access or implement.

A Zeus pipeline may also perform "branch-return" optimization, in which a branch-link instruction saves a branch target address that is used to predict the target of the next returning branch instruction. This optimization may be implemented with a depth of one (only one return address kept), or as a stack of finite depth, where a branch and link pushes onto the stack, and a branch-register pops from the stack. This optimization can eliminate the misprediction cost of simple procedure calls, as the calling branch is susceptible to hardware prediction, and the returning branch is predictable by the branch-return optimization. Like the conditional-skip optimization described above, this feature is performed by the initial Zeus implementation and requires no specific architectural feature to access or implement.

Zeus implements two related instructions that can eliminate or reduce branch delays for conditional loops, conditional branches, and computed branches. The “branch-hint” instruction has no effect on architectural state, but informs the instruction fetch unit of a potential future branch instruction, giving the addresses of both the branch instruction and of the branch target. The two forms of the instruction specify the branch instruction address relative to the current address as an immediate field, and one form (branch-hint-immediate) specifies the branch target address relative to the current address as an immediate field, and the other (branch-hint) specifies the branch target address from a general register. The branch-hint-immediate instruction is generally used to give advance notice to the instruction fetch unit of a branch-conditional instruction, so that instructions at the target of the branch can be fetched in advance of the branch-conditional instruction reaching the execution pipeline. Placing the branch hint as early as possible, and at a point where the extra instruction will not reduce the execution rate optimizes performance. In other words, an optimizing compiler should insert the branch-hint instruction as early as possible in the basic block where the parcel will contain at most one other “front-end” instruction.

Result Forwarding

When temporally adjacent instructions are executed by separate resources, the results of the first instruction must generally be forwarded directly to the resource used to execute the second instruction, where the result replaces a value which may have been fetched from a register file. Such forwarding paths use significant resources. A Zeus implementation must generally provide forwarding resources so that dependencies from earlier instructions within a string are immediately forwarded to later instructions, except between a first and second execution instruction as described above. In addition, when forwarding results from the execution units back to the data fetch unit, additional delay may be incurred.

Memory Management

This section discusses the caches, the translation mechanisms, the memory interfaces, and how the multiprocessor interface is used to maintain cache coherence.

Overview

FIG. **14** is a diagram illustrating the basic organization of the memory management system according to one embodiment of the invention. In accordance with this embodiment, the Zeus processor provides for both local and global virtual addressing, arbitrary page sizes, and coherent-cache multiprocessing. The memory management system is designed to provide the requirements for implementation of virtual machines as well as virtual memory. All facilities of the memory management system are themselves memory mapped, in order to provide for the manipulation of these facilities by high-level language, compiled code. The translation mechanism is designed to allow full byte-at-a-time

control of access to the virtual address space, with the assistance of fast exception handlers. Privilege levels provide for the secure transition between insecure user code and secure system facilities. Instructions execute at a privilege, specified by a two-bit field in the access information. Zero is the least-privileged level, and three is the most-privileged level.

In general terms, the memory management starts from a local virtual address. The local virtual address is translated to a global virtual address by an LTB (Local Translation Buffer). In turn, the global virtual address is translated to a physical address by a GTB (Global Translation Buffer). One of the addresses, a local virtual address, a global virtual address, or a physical address, is used to index the cache data and cache tag arrays, and one of the addresses is used to check the cache tag array for cache presence. Protection information is assembled from the LTB, GTB, and optionally the cache tag, to determine if the access is legal.

This form varies somewhat, depending on implementation choices made. Because the LTB leaves the lower 48 bits of the address alone, indexing of the cache arrays with the local virtual address is usually identical to cache arrays indexed by the global virtual address. However, indexing cache arrays by the global virtual address rather than the physical address produces a coherence issue if the mapping from global virtual address to physical is many-to-one.

Starting from a local virtual address, the memory management system performs three actions in parallel: the low-order bits of the virtual address are used to directly access the data in the cache, a low-order bit field is used to access the cache tag, and the high-order bits of the virtual address are translated from a local address space to a global virtual address space.

Following these three actions, operations vary depending upon the cache implementation. The cache tag may contain either a physical address and access control information (a physically-tagged cache), or may contain a global virtual address and global protection information (a virtually-tagged cache).

For a physically-tagged cache, the global virtual address is translated to a physical address by the GTB, which generates global protection information. The cache tag is checked against the physical address, to determine a cache hit. In parallel, the local and global protection information is checked.

For a virtually-tagged cache, the cache tag is checked against the global virtual address, to determine a cache hit, and the local and global protection information is checked. If the cache misses, the global virtual address is translated to a physical address by the GTB, which also generates the global protection information.

Local Translation Buffer

The 64-bit global virtual address space is global among all tasks. In a multitask environment, requirements for a task-local address space arise from operations such as the UNIX “fork” function, in which a task is duplicated into parent and child tasks, each now having a unique virtual address space. In addition, when switching tasks, access to one task’s address space must be disabled and another task’s access enabled.

Zeus provides for portions of the address space to be made local to individual tasks, with a translation to the global virtual space specified by four 16-bit registers for each local virtual space. The registers specify a mask selecting which of the high-order 16 address bits are checked to match a particular value, and if they match, a value with which to modify the virtual address. Zeus avoids setting a fixed page size or local address size; these can be set by software conventions.

A local virtual address space is specified by the following:

Local virtual address space specifiers

field name	size	description
lm	16	mask to select fields of local virtual address to perform match over
la	16	value to perform match with masked local virtual address
lx	16	value to xor with local virtual address if matched
lp	16	local protection field (detailed later)

Physical Address

There are as many LTB as threads, and up to 2^3 (8) entries per LTB. Each entry is 128 bits, with the high order 64 bits reserved. FIG. **15** illustrates the physical address of a LTB entry for thread th, entry en, byte b.

Definition

FIG. **16** illustrates a definition for AccessPhysicalLTB.

Entry Format

FIG. **17** illustrates how various 16-bit values are packed together into a 64-bit LTB entry. The LTB contains a separate context of register sets for each thread, indicated by the th index above. A context consists of one or more sets of lm/la/lx/lp registers, one set for each simultaneously accessible local virtual address range, indicated by the en index above. This set of registers is called the "Local TB context," or LTB (Local Translation Buffer) context. The effect of this mechanism is to provide the facilities normally attributed to segmentation. However, in this system there is no extension of the address range, instead, segments are local nicknames for portions of the global virtual address space.

A failure to match a LTB entry results either in an exception or an access to the global virtual address space, depending on privilege level. A single bit, selected by the privilege level active for the access from a four bit control register field, global access, ga determines the result. If ga_{PL} is zero (0), the failure causes an exception, if it is one (1), the failure causes the address to be directly used as a global virtual address without modification.

FIG. **18** illustrates global access as fields of a control register. Usually, global access is a right conferred to highly privilege levels, so a typical system may be configured with ga0 and gal clear (0), but ga2 and ga3 set (1). A single low-privilege (0) task can be safely permitted to have global access, as accesses are further limited by the rwxg privilege fields. A concrete example of this is an emulation task, which may use global addresses to simulate segmentation, such as an x86 emulation. The emulation task then runs as privilege 0, with ga0 set, while most user tasks run as privilege 1, with ga1 clear. Operating system tasks then use privilege 2 and 3 to communicate with and control the user tasks, with ga2 and ga3 set.

For tasks that have global access disabled at their current privilege level, failure to match a LTB entry causes an exception. The exception handler may load a LTB entry and continue execution, thus providing access to an arbitrary number of local virtual address ranges.

When failure to match a LTB entry does not cause an exception, instructions may access any region in the local virtual address space, when a LTB entry matches, and may access regions in the global virtual address space when no

LTB entry matches. This mechanism permits privileged code to make judicious use of local virtual address ranges, which simplifies the manner in which privileged code may manipulate the contents of a local virtual address range on behalf of a less-privileged client. Note, however, that under this model, an LTB miss does not cause an exception directly, so the use of more local virtual address ranges than LTB entries requires more care: the local virtual address ranges should be selected so as not to overlap with the global virtual address ranges, and GTB misses to LVA regions must be detected and cause the handler to load an LTB entry.

Each thread has an independent LTB, so that threads may independently define local translation. The size of the LTB for each thread is implementation dependent and defined as the LE parameter in the architecture description register. LE is the log of the number of entries in the local TB per thread; an implementation may define LE to be a minimum of 0, meaning one LTB entry per thread, or a maximum of 3, meaning eight LTB entries per thread. For the initial Zeus implementation, each thread has two entries and LE=1.

A minimum implementation of a LTB context is a single set of lm/la/lx/lp registers per thread. However, the need for the LTB to translate both code addresses and data addresses imposes some limits on the use of the LTB in such systems. We need to be able to guarantee forward progress. With a single LTB set per thread, either the code or the data must use global addresses, or both must use the same local address range, as must the LTB and GTB exception handler. To avoid this restriction, the implementation must be raised to two sets per thread, at least one for code and one for data, to guarantee forward progress for arbitrary use of local addresses in the user code (but still be limited to using global addresses for exception handlers).

As shown in FIG. **19**, a single-set LTB context may be further simplified by reserving the implementation of the lm and la registers, setting them to a read-only zero value: Note that in such a configuration, only a single LA region can be implemented.

If the largest possible space is reserved for an address space identifier, the virtual address is partitioned as shown in FIG. **20**. Any of the bits marked as "local" below may be used as "offset" as desired.

To improve performance, an implementation may perform the LTB translation on the value of the base register (rc) or unincremented program counter, provided that a check is performed which prohibits changing the unmasked upper 16 bits by the add or increment. If this optimization is provided and the check fails, an AccessDisallowedByVirtualAddress should be signaled. If this optimization is provided, the architecture description parameter LB=1. Otherwise LTB translation is performed on the local address, la, no checking is required, and LB=0.

As shown in FIG. **21**, the LTB protect field controls the minimum privilege level required for each memory action of read (r), write (w), execute (x), and gateway (g), as well as memory and cache attributes of write allocate (wa), detail access (da), strong ordering (so), cache disable (cd), and write through (wt). These fields are combined with corresponding bits in the GTB protect field to control these attributes for the mapped memory region.

Field Description

The meaning of the fields are given by the following table:

name	size	meaning
g	2	minimum privilege required for gateway access
x	2	minimum privilege required for execute access
w	2	minimum privilege required for write access
r	2	minimum privilege required for read access
0	1	reserved
da	1	detail access
so	1	strong ordering
cc	3	cache control

Definition

FIG. **22** illustrates a definition for LocalTranslation.

Global Translation Buffer

Global virtual addresses which fail to be accessed in either the LZC, the MTB, the BTB, or PTB are translated to physical references in a table, here named the "Global Translation Buffer," (GTB).

Each processor may have one or more GTB's, with each GTB shared by one or more threads. The parameter GT, the base-two log of the number of threads which share a GTB, and the parameter T, the number of threads, allow computation of the number of GTBs ($T/2^{GT}$), and the number of threads which share each GTB (2^{GT}).

If there are two GTBs and four threads (GT=1, T=4), GTB 0 services references from threads 0 and 1, and GTB 1 services references from threads 2 and 3. In the first implementation, there is one GTB, shared by all four threads. (GT=2, T=4). The GTB has 128 entries (G=7).

Per clock cycle, each GTB can translate one global virtual address to a physical address, yielding protection information as a side effect.

A GTB miss causes a software trap. This trap is designed to permit a fast handler for GlobalTBMiss to be written in software, by permitting a second GTB miss to occur as an exception, rather than a machine check.

Physical Address

There may be as many GTB as threads, and up to 2^{15} entries per GTB. FIG. **23** illustrates the physical address of a GTB entry for thread th, entry en, byte b. Note that in FIG. **23**, the low-order GT bits of the th value are ignored, reflecting that 2^{GT} threads share a single GTB. A single GTB shared between threads appears multiple times in the address space. Referring to FIG. **24**, GTB entries are packed together so that entries in a GTB are consecutive.

Definition

FIG. **24** illustrates a definition for AccessPhysicalGTB. FIG. **25** illustrates the format of a GTB entry.

Entry Format

As shown, each GTB entry is 128 bits.

Field Description

gs=ga+size/2: 256≦size≦264, ga, global address, is aligned (a multiple of) size.

px=pa^ga. pa, ga, and px are all aligned (a multiple of) size.

The meaning of the fields are given by the following table:

name	size	meaning
gs	57	global address with size
px	56	physical xor
g	2	minimum privilege required for gateway access
x	2	minimum privilege required for execute access
w	2	minimum privilege required for write access
r	2	minimum privilege required for read access
0	1	reserved
da	1	detail access
so	1	strong ordering
cc	3	cache control

If the entire contents of the GTB entry is zero (0), the entry will not match any global address at all. If a zero value is written, a zero value is read for the GTB entry. Software must not write a zero value for the gs field unless the entire entry is a zero value.

It is an error to write GTB entries that multiply match any global address; all GTB entries must have unique, non-overlapping coverage of the global address space. Hardware may produce a machine check if such overlapping coverage is detected, or may produce any physical address and protection information and continue execution.

Limiting the GTB entry size to 128 bits allows up to replace entries atomically (with a single store operation), which is less complex than the previous design, in which the mask portion was first reduced, then other entries changed, then the mask is expanded. However, it is limiting the amount of attribute information or physical address range we can specify. Consequently, we are encoding the size as a single additional bit to the global address in order to allow for attribute information.

Definition

FIG. **26** illustrates a definition for GlobalAddressTranslation.

GTB Registers

Because the processor contains multiple threads of execution, even when taking virtual memory exceptions, it is possible for two threads to nearly simultaneously invoke software GTB miss exception handlers for the same memory region. In order to avoid producing improper GTB state in such cases, the GTB includes access facilities for indivisibly checking and then updating the contents of the GTB as a result of a memory write to specific addresses.

A 128-bit write to the address GTBUpdateFill (fill=1), as a side effect, causes first a check of the global address specified in the data against the GTB. If the global address check results in a match, the data is directed to write on the matching entry. If there is no match, the address specified by GTBLast is used, and GTBLast is incremented. If incrementing GTBLast results in a zero value, GTBLast is reset to GTBFirst, and GTBBump is set. Note that if the size of the updated value is not equal to the size of the matching entry, the global address check may not adequately ensure that no other entries also cover the address range of the updated value. The operation is unpredictable if multiple entries match the global address.

The GTBUpdateFill register is a 128-bit memory-mapped location, to which a write operation performs the operation defined above. A read operation returns a zero value. The format of the GTBUpdateFill register is identical to that of a GTB entry.

An alternative write address, GTBUpdate, (fill=0) updates a matching entry, but makes no change to the GTB if no entry matches. This operation can be used to indivisibly update a GTB entry as to protection or physical address information.

Definition

FIG. **27** illustrates a definition for GTBUpdateWrite.

Physical Address

There may be as many GTB as threads, and up to 2^{11} registers per GTB (5 registers are implemented). FIG. **28** illustrates the physical address of a GTB control register for thread th, register rn, byte b. Note that in FIG. **28**, the low-order GT bits of the th value are ignored, reflecting that 2^{GT} threads share single GTB registers. A single set of GTB registers shared between threads appears multiple times in the address space, and manipulates the GTB of the threads with which the registers are associated.

The GTBUpdate register is a 128-bit memory-mapped location, to which a write operation performs the operation defined above. A read operation returns a zero value. The format of the GTBUpdateFill register is identical to that of a GTB entry. FIG. **29** illustrates the registers GTBLast, GTBFirst, and GTBBump. The registers GTBLast, GTBFirst, and GTBBump are memory mapped. As shown in FIG. **29**, the GTBLast and GTBFirst registers are G bits wide, and the GTBBump register is one bit.

Definition

FIG. **30** illustrates a definition for AccessPhysicalGTBRegisters.

Address Generation

The address units of each of the four threads provide up to two global virtual addresses of load, store, or memory instructions, for a total of eight addresses. LTB units associated with each thread translate the local addresses into global addresses. The LZC operates on global addresses. MTB, BTB, and PTB units associated with each thread translate the global addresses into physical addresses and cache addresses. (A PTB unit associated with each thread produces physical addresses and cache addresses for program counter references. —this is optional, as by limiting address generation to two per thread, the MTB can be used for program references.) Cache addresses are presented to the LOC as required, and physical addresses are checked against cache tags as required.

Memory Banks

The LZC has two banks, each servicing up to four requests. The LOC has eight banks, each servicing at most one request.

Assuming random request addresses, FIG. **55** shows the expected rate at which requests are serviced by multi-bank/multi-port memories that have 8 total ports and divided into 1, 2, 4, or 8 interleaved banks. The LZC is 2 banks, each with 4 ports, and the LOC is 8 banks, each 1 port.

Note a small difference between applying 12 references versus **8** references for the LOC (6.5 vs 5.2), and for the LZC (7.8 vs. 6.9). This suggests that simplifying the system to produce two address per thread (program+load/store or two load/store) will not overly hurt performance. A closer simulation, taking into account the sequential nature of the program and load/store traffic may well yield better numbers, as threads will tend to line up in non-interfering patterns, and program microcaching reduces program fetching.

FIG. **56** shows the rates for both 8 total ports and 16 total ports.

Note significant differences between 8-port systems and 16-port systems, even when used with a maximum of 8 applied references. In particular, a 16-bank 1-port system is better than a 4-bank 2-port system with more than 6 applied references. Current layout estimates would require about a

14% area increase (assuming no savings from smaller/simpler sense amps) to switch to a 16-port LOC, with a 22% increase in 8-reference throughput.

Program Microcache

A program microcache (PMC) which holds only program code for each thread may optionally exist, and does exist for the initial implementation. The program microcache is flushed by reset, or by executing a B.BARRIER instruction. The program microcache is always clean, and is not snooped by writes or otherwise kept coherent, except by flushing as indicated above. The microcache is not altered by writing to the LTB or GTB, and software must execute a B.BARRIER instruction before expecting the new contents of the LTB or GTB to affect determination of PMC hit or miss status on program fetches.

In the initial implementation, the program microcache holds simple loop code. The microcache holds two separately addressed cache lines. Branches or execution beyond this region cause the microcache to be flushed and refilled at the new address, provided that the addresses are executable by the current thread. The program microcache uses the B.HINT and B.HINT.I to accelerate fetching of program code when possible. The program microcache generally functions as a prefetch buffer, except that short forward or backward branches within the region covered maintain the contents of the microcache.

Program fetches into the microcache are requested on any cycle in which less than two load/store addresses are generated by the address unit, unless the microcache is already full. System arbitration logic should give program fetches lower priority than load/store references when first presented, then equal priority if the fetch fails arbitration a certain number of times. The delay until program fetches have equal priority should be based on the expected time the program fetch data will be executed; it may be as small as a single cycle, or greater for fetches which are far ahead of the execution point.

Wide Microcache

A wide microcache (WMC) which holds only data fetched for wide (W) instructions may optionally exist, and does exist for the initial implementation, for each unit which implements one or more wide (W) instructions.

The wide (W) instructions each operate on a block of data fetched from memory and the contents of one or more registers, producing a result in a register. Generally, the amount of data in the block exceeds the maximum amount of data that the memory system can supply in a single cycle, so caching the memory data is of particular importance. All the wide (W) instructions require that the memory data be located at an aligned address, an address that is a multiple of the size of the memory data, which is always a power of two.

The wide (W) instructions are performed by functional units which normally perform execute or "back-end" instructions, though the loading of the memory data requires use of the access or "front-end" functional units. To minimize the use of the "front-end" functional units, special rules are used to maintain the coherence of a wide microcache (WMC).

Execution of a wide (W) instruction has a residual effect of loading the specified memory data into a wide microcache (WMC). Under certain conditions, a future wide (W) instruction may be able to reuse the WMC contents.

First of all, any store or cache coherency action on the physical addresses referenced by the WMC will invalidate the contents. The minimum translation unit of the virtual memory system, 256 bytes, defines the number of physical address blocks which must be checked by any store. A WMC for the W.TABLE instruction may be as large as 4096 bytes, and so requires as many as 16 such physical address blocks to be checked for each WMC entry. A WMC for the W.SWITCH or W.MUL.*instructions need check only one address block for each WMC entry, as the maximum size is 128 bytes.

By making these checks on the physical addresses, we do not need to be concerned about changes to the virtual memory mapping from virtual to physical addresses, and the virtual memory state can be freely changed without invalidating any WMC.

Absent any of the above changes, the WMC is only valid if it contains the contents relevant to the current wide (W) instruction. To check this with minimal use of the front-end units, each WMC entry contains a first tag with the thread and address register for which it was last used. If the current wide (W) instruction uses the same thread and address register, it may proceed safely. Any intervening writes to that address register by that thread invalidates the WMC thread and address register tag.

If the above test fails, the front-end is used to fetch the address register and check its contents against a second WMC tag, with the physical addresses for which it was last used. If the tag matches, it may proceed safely. As detailed above, any intervening stores or cache coherency action by any thread to the physical addresses invalidates the WMC entry.

If both the above tests fail for all relevant WMC entries, there is no alternative but to load the data from the virtual memory system into the WMC. The front-end units are responsible for generating the necessary addresses to the virtual memory system to fetch the entire data block into a WMC.

For the first implementation, it is anticipated that there be eight WMC entries for each of the two X units (for W.SWITCH instructions), eight WMC entries for each of the two E units (for W.MUL instructions), and four WMC entries for the single T unit. The total number of WMC address tags requires is 8*2*1+8*2*1+4*1*16=96 entries.

The number of WMC address tags can be substantially reduced to 32+4=36 entries by making an implementation restriction requiring that a single translation block be used to translate the data address of W.TABLE instructions. With this restriction, each W.TABLE WMC entry uses a contiguous and aligned physical data memory block, for which a single address tag can contain the relevant information. The size of such a block is a maximum of 4096 bytes. The restriction can be checked by examining the size field of the referenced GTB entry.

Level Zero Cache

The innermost cache level, here named the "Level Zero Cache," (LZC) is fully associative and indexed by global address. Entries in the LZC contain global addresses and previously fetched data from the memory system. The LZC is an implementation feature, not visible to the Zeus architecture.

Entries in the LZC are also used to hold the global addresses of store instructions that have been issued, but not yet completed in the memory system. The LZC entry may also contain the data associated with the global address, as maintained either before or after updating with the store data. When it contains the post-store data, results of stores may be forwarded directly to the requested reference.

With an LZC hit, data is returned from the LZC data, and protection from the LZC tag. No LOC access is required to complete the reference.

All loads and program fetches are checked against the LZC for conflicts with entries being used as store buffer. On a LZC hit on such entries, if the post-store data is present, data may

be returned by the LZC to satisfy the load or program fetch. If the post-store data is not present, the load or program fetch must stall until the data is available.

With an LZC miss, a victim entry is selected, and if dirty, the victim entry is written to the LOC. The LOC cache is accessed, and a valid LZC entry is constructed from data from the LOC and tags from the LOC protection information.

All stores are checked against the LZC for conflicts, and further cause a new entry in the LZC, or "take over" a previously clean LZC entry for this purpose. Unaligned stores may require two entries in the LZC. At time of allocation, the address is filled in.

Two operations then occur in parallel—1) for write-back cached references, the remaining bytes of the hexlet are loaded from the LOC (or LZC), and 2) the addressed bytes are filled in with data from data path. If an exception causes the store to be purged before retirement, the LZC entry is marked invalid, and not written back. When the store is retired, the LZC entry can be written back to LOC or external interface.

Structure

The eight memory addresses are partitioned into up to four odd addresses, and four even addresses.

The LZC contains 16 fully associative entries that may each contain a single hexlet of data at even hexlet addresses (LZCE), and another 16 entries for odd hexlet addresses (LZCO). The maximum capacity of the LZC is 16*32=512 bytes.

The tags for these entries are indexed by global virtual address (63 . . . 5), and contain access control information, detailed below.

The address of entries accessed associatively is also encoded into binary and provided as output from the tags for use in updating the LZC, through its write ports.

```
8 bit rwxg
16 bit valid
16 bit dirty
4 bit L0$ address
16 bit protection
def data,protect,valid,dirty,match $\leftarrow$ LevelZeroCacheRead(ga) as
    eo $\leftarrow$ $ga_4$
    match $\leftarrow$ NONE
    for i $\leftarrow$ 0 to LevelZeroCacheEntries/2−1
        if ($ga_{63..5}$ = LevelZeroTag[eo][i] then
            match $\leftarrow$ i
        endif
    endfor
    if match = NONE then
        raise LevelZeroCacheMiss
    else
        data $\leftarrow$ LevelZeroData[eo][match]$_{127..0}$
        valid $\leftarrow$ LevelZeroData[eo][match]$_{143..128}$
        dirty $\leftarrow$ LevelZeroData[eo][match]$_{159..144}$
        protect $\leftarrow$ LevelZeroData[eo][match]$_{167..160}$
    endif
enddef
```

Level One Cache

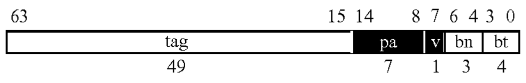
The next cache level, here named the "Level One Cache," (LOC) is four-set-associative and indexed by the physical address. The eight memory addresses are partitioned into up to eight addresses for each of eight independent memory banks. The LOC has a cache block size of 256 bytes, with triclet (32-byte) sub-blocks.

The LOC may be partitioned into two sections, one part used as a cache, and the remainder used as "niche memory." Niche memory is at least as fast as cache memory, but unlike cache, never misses to main memory. Niche memory may be placed at any virtual address, and has physical addresses fixed in the memory map. The nl field in the control register configures the partitioning of LOC into cache memory and niche memory.

The LOC data memory is (256+8)×4×(128+2) bits, depth to hold 256 entries in each of four sets, each entry consisting of one hexlet of data (128 bits), one bit of parity, and one spare bit. The additional 8 entries in each of four sets hold the LOC tags, with 128 bits per entry for ⅛ of the total cache, using 512 bytes per data memory and 4 K bytes total.

There are 128 cache blocks per set, or 512 cache blocks total. The maximum capacity of the LOC is 128 k bytes. Used as a cache, the LOC is partitioned into 4 sets, each 32 k bytes. Physically, the LOC is partitioned into 8 interleaved physical blocks, each holding **16***k* bytes.

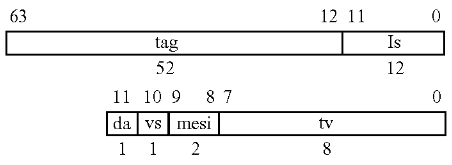
The physical address $pa_{63 \ldots 0}$ is partitioned as below into a 52 to 54 bit tag (three to five bits are duplicated from the following field to accommodate use of portion of the cache as niche), 8-bit address to the memory bank (7 bits are physical address (pa), 1 bit is virtual address (v)), 3 bit memory bank select (bn), and 4-bit byte address (bt). All access to the LOC are in units of 128 bits (hexlets), so the 4-bit byte address (bt) does not apply here. The shaded field (pa,v) is translated via nl to a cache identifier (ci) and set identifier (si) and presented to the LOC as the LOC address to LOC bank bn.



The LOC tag consists of 64 bits of information, including a 52 to 54-bit tag and other cache state information. Only one MTB entry at a time may contain a LOC tag.

With 256 byte cache lines, there are 512 cache blocks. At 64 bits per tag, the cache tags require 4 k bytes of storage. This storage is adjacent to the LOC data memory itself, using physical addresses=1024 . . . 1055. Alternatively (see detailed description below), physical addresses=0 . . . 31 may be used.

The format of a LOC tag entry is shown below.

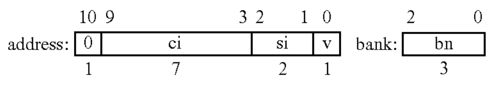


The meaning of the fields are given by the following table:

name	size	meaning
tag	52	physical address tag
da	1	detail access (or physical address bit 11)
vs	1	victim select (or physical address bit 10)
mesi	2	coherency: modified (3), exclusive (2), shared (1), invalid (0)
tv	8	triclet valid (1) or invalid (0)

To access the LOC, a global address is supplied to the Micro-Tag Buffer (MTB), which associatively looks up the global address into a table holding a subset of the LOC tags. In particular, each MTB table entry contains the cache index derived from physical address bits **14** . . . **8**, ci, (7 bits) and set identifier, si, (2 bits) required to access the LOC data. Each MTB table entry also contains the protection information of the LOC tag.

With an MTB hit, protection information is supplied from the MTB. The MTB supplies the resulting cache index (ci, from the MTB), set identifier, si, (2 bits) and virtual address (bit **7**, v, from the LA), which are applied to the LOC data bank selected from bits **6** . . . **4** of the LA. The diagram below shows the address presented to LOC data bank bn.



With an MTB miss, the GTB (described below) is referenced to obtain a physical address and protection information.

To select the cache line, a 7-bit niche limit register nl is compared against the value of $pa_{14 \ldots 8}$ from the GTB. If $pa_{14 \ldots 8}$<nl, a 7-bit address modifier register am is inclusive-or'ed against $pa_{14 \ldots 8}$, producing a cache index, ci. Otherwise, $pa_{14 \ldots 8}$ is used as ci. Cache lines 0 . . . nl−1, and cache tags 0 . . . nl−1, are available for use as niche memory. Cache lines nl . . . 127 and cache tags nl . . . 127 are used as LOC.

$$ci \leftarrow (pa_{14 \ldots 8} < nl) \mathbin{?} (pa_{14 \ldots 8} \| am) : pa_{14 \ldots 8}$$

The address modifier am is $(1^{7-log(128-nl)} \parallel 0^{log(128-nl)})$. The bt field specifies the least-significant bit used for tag, and is (nl<112) ? 12:8+log(128−nl):

nl	am	bt
0	0	12
1 . . . 64	64	12
65 . . . 96	96	12
97 . . . 112	112	12
113 . . . 120	120	11
121 . . . 124	124	10
125 . . . 126	126	9
127	127	8

Values for nl in the range **113** . . . **127** require more than 52 physical address tag bits in the LOC tag and a requisite reduction in LOC features. Note that the presence of bits **14** . . . **10** of the physical address in the LOC tag is a result of the possibility that, with am=64 . . . 127, the cache index value ci cannot be relied upon to supply bit **14** . . . **8**. Bits **9** . . . **8** can be safely inferred from the cache index value ci, so long as nl is in the range **0** . . . **124**. When nl is in the range **113** . . . **127**, the da bit is used for bit **11** of the physical address, so the Tag detail access bit is suppressed. When nl is in the range **121** . . . **127**, the vs bit is used for bit **10** of the physical address, so victim selection is performed without state bits in the LOC tag. When nl is in the range **125** . . . **127**, the set associativity is decreased, so that si_1 is used for bit **9** of the physical address and when nl is 127, si_0 is used for bit **8** of the physical address.

Four tags are fetched from the LOC tags and compared against the PA to determine which of the four sets contain the data. The four tags are contained in two consecutive banks; they may be simultaneously or independently fetched. The diagram below shows the address presented to LOC data bank $(ci_{1 \ldots 0} \| si_1)$.

	10	9 … 5	4 … 0
address:	CT	0	$ci_{6...2}$
	1	5	5

	2 1	0
bank:	$ci_{1...0}$	si_1
	2	1

Note that the CT architecture description variable is present in the above address. CT describes whether dedicated locations exist in the LOC for tags at the next power-of-two boundary above the LOC data. The niche-mapping mechanism can provide the storage for the LOC tags, so the existence of these dedicated tags is optional: If CT=0, addresses at the beginning of the LOC (0 . . . 31 for this implementation) are used for LOC tags, and the nl value should be adjusted accordingly by software.

The LOC address (ci||si) uniquely identifies the cache location, and this LOC address is associatively checked against all MTB entries on changes to the LOC tags, such as by cache block replacement, bus snooping, or software modification. Any matching MTB entries are flushed, even if the MTB entry specifies a different global address—this permits address aliasing (the use of a physical address with more than one global address.

With an LOC miss, a victim set is selected (LOC victim selection is described below), whose contents, if any subblock is modified, is written to the external memory. A new LOC entry is constructed with address and protection information from the GTB, and data fetched from external memory.

The table below shows the contents of LOC data memory banks 0 . . . 7 for addresses 0 . . . 2047:

address	bank 7	. . .	bank 1	bank 0
0	line 0, hexlet 7, set 0		line 0, hexlet 1, set 0	line 0, hexlet 0, set 0
1	line 0, hexlet 15, set 0		line 0, hexlet 9, set 0	line 0, hexlet 8, set 0
2	line 0, hexlet 7, set 1		line 0, hexlet 1, set 1	line 0, hexlet 0, set 1
3	line 0, hexlet 15, set 1		line 0, hexlet 9, set 1	line 0, hexlet 8, set 1
4	line 0, hexlet 7, set 2		line 0, hexlet 1, set 2	line 0, hexlet 0, set 2
5	line 0, hexlet 15, set 2		line 0, hexlet 9, set 2	line 0, hexlet 8, set 2
6	line 0, hexlet 7, set 3		line 0, hexlet 1, set 3	line 0, hexlet 0, set 3
7	line 0, hexlet 15, set 3		line 0, hexlet 9, set 3	line 0, hexlet 8, set 3
8	line 1, hexlet 7, set 0		line 1, hexlet 1, set 0	line 1, hexlet 0, set 0
9	line 1, hexlet 15, set 0		line 1, hexlet 9, set 0	line 1, hexlet 8, set 0
10	line 1, hexlet 7, set 1		line 1, hexlet 1, set 1	line 1, hexlet 0, set 1
11	line 1, hexlet 15, set 1		line 1, hexlet 9, set 1	line 1, hexlet 8, set 1
12	line 1, hexlet 7, set 2		line 1, hexlet 1, set 2	line 1, hexlet 0, set 2
13	line 1, hexlet 15, set 2		line 1, hexlet 9, set 2	line 1, hexlet 8, set 2
14	line 1, hexlet 7, set 3		line 1, hexlet 1, set 3	line 1, hexlet 0, set 3
15	line 1, hexlet 15, set 3		line 1, hexlet 9, set 3	line 1, hexlet 8, set 3
. . .	. . .		. . .	. . .
1016	line 127, hexlet 7, set 0		line 127, hexlet 1, set 0	line 127, hexlet 0, set 0
1017	line 127, hexlet 15, set 0		line 127, hexlet 9, set 0	line 127, hexlet 8, set 0
1018	line 127, hexlet 7, set 1		line 127, hexlet 1, set 1	line 127, hexlet 0, set 1
1019	line 127, hexlet 15, set 1		line 127, hexlet 9, set 1	line 127, hexlet 8, set 1
1020	line 127, hexlet 7, set 2		line 127, hexlet 1, set 2	line 127, hexlet 0, set 2
1021	line 127, hexlet 15, set 2		line 127, hexlet 9, set 2	line 127, hexlet 8, set 2
1022	line 127, hexlet 7, set 3		line 127, hexlet 1, set 3	line 127, hexlet 0, set 3
1023	line 127, hexlet 15, set 3		line 127, hexlet 9, set 3	line 127, hexlet 8, set 3
1024	tag line 3, sets 3 and 2		tag line 0, sets 3 and 2	tag line 0, sets 1 and 0
1025	tag line 7, sets 3 and 2		tag line 4, sets 3 and 2	tag line 4, sets 1 and 0
. . .	. . .		. . .	. . .
1055	tag line 127, sets 3 and 2		tag line 124, sets 3 and 2	tag line 124, sets 1 and 0
1056	reserved		reserved	reserved
. . .	. . .		. . .	. . .
2047	reserved		reserved	reserved

The following table summarizes the state transitions required by the LOC cache:

cc	op	mesi	v	bus op	c	x	mesi	v	w	m	notes
NC	R	x	x	uncached read							
NC	W	x	x	uncached write							
CD	R	I	x	uncached read							
CD	R	x	0	uncached read							
CD	R	MES	1	(hit)							
CD	W	I	x	uncached write							
CD	W	x	0	uncached write							
CD	W	MES	1	uncached write						1	
WT/WA	R	I	x	triclet read	0	x					
WT/WA	R	I	x	triclet read	1	0	S	1			
WT/WA	R	I	x	triclet read	1	1	E	1			
WT/WA	R	MES	0	triclet read	0	x					inconsistent KEN#

-continued

WT/WA	R	S	0	triclet read	1	0		1			
WT/WA	R	S	0	triclet read	1	1		1			E->S: extra sharing
WT/WA	R	E	0	triclet read	1	0		1			
WT/WA	R	E	0	triclet read	1	1	S	1			shared block
WT/WA	R	M	0	triclet read	1	0	S	1			other subblocks M->I
WT/WA	R	M	0	triclet read	1	1		1			E->M: extra dirty
WT/WA	R	MES	1	(hit)							
WT	W	I	x	uncached write							
WT	W	x	0	uncached write							
WT	W	MES	1	uncached write						1	
WA	W	I	x	triclet read	0	x			1		throwaway read
WA	W	I	x	triclet read	1	0	S	1	1	1	
WA	W	I	x	triclet read	1	1	M	1		1	
WA	W	MES	0	triclet read	0	x			1	1	inconsistent KEN#
WA	W	S	0	triclet read	1	0	S	1	1	1	
WA	W	S	0	triclet read	1	1	M	1		1	
WA	W	S	1	write		0	S	1		1	
WA	W	S	1	write		1	S	1		1	E->S: extra sharing
WA	W	E	0	triclet read	1	0	S	1	1	1	
WA	W	E	0	triclet read	1	1	E	1	1	1	
WA	W	E	1	(hit)		x	M	1			E->M: extra dirty
WA	W	M	0	triclet read	1	0	M	1	1	1	
WA	W	M	0	triclet read	1	1	M	1		1	
WA	W	M	1	(hit)		x	M	1			

cc	cache control
op	operation: R = read, W = write
mesi	current mesi state
v	current tv state
bus op	bus operation
c	cachable (triclet) result
x	exclusive result
mesi	new mesi state
v	new tv state
w	cacheable write after read
m	merge store data with cache line data
notes	other notes on transition

Definition

def data,tda←LevelOneCacheAccess (pa,size,lda,gda,cc,op, wd) as

```
def data,tda ← LevelOneCacheAccess(pa,size,lda,gda,cc,op,wd) as
// cache index
am ← $(1^{7-log(128-nl)} || 0^{log(128-nl)})$
ci ← $(pa_{14..8}<nl)$ ? $(pa_{14..8}||am)$ : $pa_{14..8}$
bt ← $(nl \leqq 112)$ ? 12 : 8+log(128−nl)
// fetch tags for all four sets
tag10 ← ReadPhysical($0xFFFFFFFF00000000_{63..19}||CT||0^5||ci||0^1||0^4$, 128)
Tag[0] ← $tag10_{63..0}$
Tag[1] ← $tag10_{127..64}$
tag32 ← □ReadPhysical($0xFFFFFFFF00000000_{63..19}||CT||0^5||ci||1^1||0^4$, 128)
Tag[2] ← $tag32_{63..0}$
Tag[3] ← $tag32_{127..64}$
vsc ← $(Tag[3]_{10} || Tag[2]_{10})$ ^ $(Tag[1]_{10} || Tag[0]_{10})$
// look for matching tag
si ← MISS
for i ← 0 to 3
    if $(Tag[i]_{63..10} || i_{1..0} || 0^7)_{63..bt} = pa_{63..bt}$ then
        si ← i
    endif
endfor
// detail access checking on MISS
if (si = MISS) and (lda ≠ gda) then
    if gda then
        PerformAccessDetail(AccessDetailRequiredByGlobalTB)
    else
        PerformAccessDetail(AccessDetailRequiredByLocalTB)
    endif
endif
```

-continued

```
// if no matching tag or invalid MESI or no sub-block, perform cacheable read/write
bd $\leftarrow$ (si = MISS) or (Tag[si]$_{9..8}$ = I) or ((op=W) and (Tag[si]$_{9..8}$ = S)) or ~Tag[si]$_{pa_{7..5}}$
if bd then
    if (op=W) and (cc $\geqq$ WA) and ((si = MISS) or ~Tag[si]$_{pa_{7..5}}$ or (Tag(si)$_{9..8}$ $\neq$ S)) then
        data,cen,xen $\leftarrow$ AccessPhysical(pa,size,cc,R,0)
        //if cache disabled or shared, do a write through
        if ~cen or ~xen then
            data,cen,xen $\leftarrow$ AccessPhysical(pa,size,cc,W,wd)
        endif
    else
        data,cen,xen $\leftarrow$ AccessPhysical(pa,size,cc,op,wd)
    endif
    al $\leftarrow$ cen
else
    al $\leftarrow$ 0
endif
// find victim set and eject from cache
if al and (si = MISS or Tag[si]$_{9..8}$ = I) then
    case bt of
        12..11:
            si $\leftarrow$ vsc
        10..8:
            gvsc $\leftarrow$ gvsc + 1
            si $\leftarrow$ (bt$\leqq$9) : pa$_9$ : gvsc$_1$ ^ pa$_{11}$ || (bt$\leqq$8) : pa$_8$ : gvsc$_0$ ^ pa$_{10}$
    endcase
    if Tag[si]$_{9..8}$ = M then
        for i $\leftarrow$ 0 to 7
            if Tag[si]$_i$ then
                vca $\leftarrow$ 0xFFFFFFFF00000000$_{63..19}$||0||ci||si||i$_{2..0}$||0$^4$
                vdata $\leftarrow$ ReadPhysical(vca, 256)
                vpa $\leftarrow$ (Tag[si]$_{63..10}$ || si$_{1..0}$ || 0$^7$)$_{63..bt}$||pa$_{bt-1..8}$||i$_{2..0}$||0||0$^4$
                WritePhysical(vpa, 256, vdata)
            endif
        endfor
    endif
    if Tag[vsc+1]$_{9..8}$ = I then
        nvsc $\leftarrow$ vsc + 1
    elseif Tag[vsc+2]$_{9..8}$ = I then
        nvsc $\leftarrow$ vsc + 2
    elseif Tag[vsc+3]$_{9..8}$ = I then
        nvsc $\leftarrow$ vsc + 3
    else
        case cc of
            NC, CD, WT, WA, PF:
                nvsc $\leftarrow$ vsc + 1
            LS, SS:
                nvsc $\leftarrow$ vsc //no change
            endif
        endcase
    endif
    tda $\leftarrow$ 0
    sm $\leftarrow$ 0$^{7-pa_{7..5}}$ || 1$^1$ || 0$^{pa_{7..5}}$
    else
        nvsc $\leftarrow$ vsc
        tda $\leftarrow$ (bt>11) ? Tag[si]$_{11}$ : 0
        if al then
            sm $\leftarrow$ Tag[si]$_{7..1+pa_{7..5}}$ || 1$^1$ || Tag[si]$_{pa_{7..5}}$−1..0
        endif
    endif
    // write new data into cache and update victim selection and other tag fields
    if al then
        if op=R then
            mesi $\leftarrow$ xen ? E : S
        else
            mesi $\leftarrow$ xen ? M : I TODO
        endif
        case bt of
            12:
                Tag[si] $\leftarrow$ pa$_{63..bt}$ || tda || Tag[si ^ 2]$_{10}$ ^ nvsc$_{si_0}$ || mesi || sm
                Tag[si ^ 1]$_{10}$ $\leftarrow$ Tag[si ^ 3]$_{10}$ ^ nvsc$_{1}$ ^ $_{si_0}$
            11:
                Tag[si] $\leftarrow$ pa$_{63..bt}$ || Tag[si ^ 2]$_{10}$ ^ nvsc$_{si_0}$ || mesi || sm
                Tag[si ^ 1]$_{10}$ $\leftarrow$ Tag[si ^ 3]$_{10}$ ^ nvsc$_{1}$ ^ $_{si_0}$
```

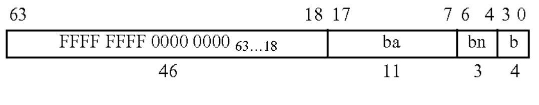
-continued

```
                10:
                    Tag[si] ← $pa_{63..bt}$ || mesi || sm
            endcase
            dt ← 1
            nca ← $0xFFFFFFFF00000000_{63..19}||0||ci||si||pa_{7..5}||0^4$
            WritePhysical(nca, 256, data)
        endif
        // retrieve data from cache
        if ~bd then
            nca ← $0xFFFFFFFF00000000_{63..19}||0||ci||si||pa_{7..5}||0^4$
            data ← ReadPhysical(nca, 128)
        endif
        // write data into cache
        if (op=W) and bd and al then
            nca ← $0xFFFFFFFF00000000_{63..19}||0||ci||si||pa_{7..5}||0^4$
            data ← ReadPhysical(nca, 128)
            mdata ← $data_{127..8*(size+pa3..0)}$ || $wd_{8*(size+pa3..0)-1..8*pa3..0}$ || $data_{8*pa3..0..0}$
            WritePhysical(nca, 128, mdata)
        endif
        // prefetch into cache
        if al=bd and (cc=PF or cc=LS) then
            af ← 0 // abort fetch if af becomes 1
            for i ← 0 to 7
                if ~$Tag[si]_i$ and ~af then
                    data,cen,xen ← AccessPhysical($pa_{63..8}||i_{2..0}||0||0^4$,256,cc,R,0)
                    if cen then
                        nca ← $0xFFFFFFFF00000000_{63..19}||0||ci||si||i_{2..0}||0^4$
                        WritePhysical(nca, 256, data)
                        $Tag[si]_i$ ← 1
                        dt ← 1
                    else
                        af ← 1
                    endif
                endif
            endfor
        endif
        // cache tag writeback if dirty
        if dt then
            nt ← $Tag[si_1||1^1)$ || $Tag[si_1||0^1)$
            WritePhysical($0xFFFFFFFF00000000_{63..19}||CT||0^5||ci||si_1||0^4$, 128, nt)
        endif
enddef
```

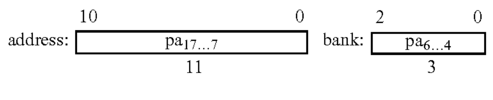
Physical Address

The LOC data memory banks are accessed implicitly by cached memory accesses to any physical memory location as shown above. The LOC data memory banks are also accessed explicitly by uncached memory accesses to particular physical address ranges. The address mapping of these ranges is designed to facilitate use of a contiguous portion of the LOC cache as niche memory.

The physical address of a LOC hexlet for LOC address ba, bank bn, byte b is:



Within the explicit LOC data range, starting from a physical address $pa_{17...0}$, the diagram below shows the LOC address ($pa_{17...7}$) presented to LOC data bank ($pa_{6...4}$).



The table below shows the LOC data memory bank and address referenced by byte address offsets in the explicit LOC data range. Note that this mapping includes the addresses use for LOC tags.

Byte offset	
0	bank 0, address 0
16	bank 1, address 0
32	bank 2, address 0
48	bank 3, address 0
64	bank 4, address 0
80	bank 5, address 0
96	bank 6, address 0
112	bank 7, address 0
128	bank 0, address 1
144	bank 1, address 1
160	bank 2, address 1
176	bank 3, address 1
192	bank 4, address 1
208	bank 5, address 1
224	bank 6, address 1
240	bank 7, address 1
. . .	. . .
262016	bank 0, address 2047
262032	bank 1, address 2047
262048	bank 2, address 2047
262064	bank 3, address 2047

-continued

Byte offset	
262080	bank 4, address 2047
262096	bank 5, address 2047
262112	bank 6, address 2047
262128	bank 7, address 2047

Definition

```
def data ← AccessPhysicalLOC(pa,op,wd) as
    bank ← $pa_{6..4}$
    addr ← $pa_{17..7}$
    case op of
        R:
            rd ← LOCArray[bank][addr]
            crc ← LOCRedundancy[bank]
            data ← (crc and $rd_{130..2}$) or (~crc and $rd_{128..0}$)
            p[0] ← 0
            for i ← 0 to 128 by 1
                p[l+1] ← p[i] ^ $data_i$
            endfor
            if $ControlRegister_{61}$ and (p[129] ≠ 1) then
                raise CacheError
            endif
        W:
            p[0] ← 0
            for l ← 0 to 127 by 1
                p[l+1] ← p[i] ^ $wd_i$
            endfor
            $wd_{128}$ ← ~p[128]
            crc ← LOCRedundancy[bank]
            rdata ← ($crc_{126..0}$ and $wd_{126..0}$) or (~$crc_{126..0}$ and $wd_{128..2}$)
            LOCArray[bank][addr] ← $wd_{128..127}$ || rdata || $wd_{1..0}$
    endcase
enddef
```

Level One Cache Stress Control

LOC cells may be fabricated with marginal parameters, for which changes in clock timing or power supply voltage may cause these LOC cells to fail or pass. When testing the LOC while the part is in a normal circuit environment, rather than a special test environment with changeable power supply levels, cells with marginal parameters may not reliably fail testing.

To combat this problem, two bits of the control register, LOC stress, may be set to stress the circuit environment while testing. Under normal operation, these bits are cleared (00), while during stress testing, one or more of these bits are set (01, 10, 11). Self-testing should be performed in each of the environment settings, and the detected failures combined together to produce a reliable test for cells with marginal parameters.

Level One Cache Redundancy

The LOC contains facilities that can be used to avoid minor defects in the LOC data array.

Each LOC bank has three additional bits of data storage for each 128 bits of memory data (for a total of 131 bits). One of these bits is used to retain odd parity over the 128 bits of memory data, and the other two bits are spare, which can be pressed into service by setting a non-zero value in the LOC redundancy control register for that bank.

Each row of a LOC bank contains 131 bits: 128 bits of memory data, one bit for parity, and two spare bits:

130 129	128	127 0
spare	p	data
2	1	128

LOC redundancy control has 129 bits:

128	127 0
pc	control
1	128

Each bit set in the control word causes the corresponding data bit to be selected from a bit address increased by two:

$$\text{output}\leftarrow(\text{data and}\sim\text{control})\text{or}((\text{spare}_0\|\text{p}\|\text{data}_{127\,\ldots\,2})$$
$$\text{and control})\text{parity}\leftarrow(\text{p and}\sim\text{pc})\text{or}(\text{spare}_1\text{ and pc})$$

The LOC redundancy control register has 129 bits, but is written with a 128-bit value. To set the pc bit in the LOC redundancy control, a value is written to the control with either bit **124** set (1) or bit **126** set (1). To set bit **124** of the LOC redundancy control, a value is written to the control with both bit **124** set (1) and **126** set (1). When the LOC redundancy control register is read, the process is reversed by selecting the pc bit instead of control bit **124** for the value of bit **124** if control bit **126** is zero (0).

This system can remove one defective column at an even bit position and one defective column at an odd bit position within each LOC block. For each defective column location, x, LOC control bit must be set at bits x, x+2, x+4, x+6, . . . If the defective column is in the parity location (bit **128**), then set bit **124** only. The following table defines the control bits for parity, bit **126** and bit **124**: (other control bits are same as values written)

$value_{126}$	$value_{124}$	pc	$control_{126}$	$control_{124}$
0	0	0	0	0
0	1	1	0	0
1	0	1	1	0
1	1	1	1	1

Physical Address

The LOC redundancy controls are accessed explicitly by uncached memory accesses to particular physical address ranges.

The physical address of a LOC redundancy control for LOC bank bn, byte b is:

63 7	6 4	3 0
FFFF FFFF 0900 0000 $_{63...7}$	bn	b
57	3	4

Definition

```
def data ← AccessPhysicalLOCRedundancy(pa,op,wd) as
  bank ← $pa_{6..4}$
  case op of
    R:
      rd ← LOCRedundancy[bank]
```

-continued

```
        data $\leftarrow$ $rd_{127..125} || (rd_{126}\ ?\ rd_{124} : rd_{128}) || rd_{123..0}$
    W:
        rd $\leftarrow$ $(wd_{126}$ or $wd_{124}) || wd_{127..125} || (wd_{126}$ and $wd_{124}) || wd_{123..0}$
        LOCRedundancy[bank] $\leftarrow$ rd
    endcase
enddef
```

Memory Attributes

Fields in the LTB, GTB and cache tag control various attributes of the memory access in the specified region of memory. These include the control of cache consultation, updating, allocation, prefetching, coherence, ordering, victim selection, detail access, and cache prefetching.

Cache Control

The cache may be used in one of five ways, depending on a three-bit cache control field (cc) in the LTB and GTB. The cache control field may be set to one of seven states: NC, CD, WT, WA, PF, SS, and LS:

		read		write		read/write	
State		consult	allocate	update	allocate	victim	prefetch
No Cache	0	No	No	No	No	No	No
Cache Disable	1	Yes	No	Yes	No	No	No
Write Through	2	Yes	Yes	Yes	No	No	No
reserved	3						
Write Allocate	4	Yes	Yes	Yes	Yes	No	No
PreFetch	5	Yes	Yes	Yes	Yes	No	Yes
SubStream	6	Yes	Yes	Yes	Yes	Yes	No
LineStream	7	Yes	Yes	Yes	Yes	Yes	Yes

The Zeus processor controls cc as an attribute in the LTB and GTB, thus software may set this attribute for certain address ranges and clear it for others. A three-bit field indicates the choice of caching, according to the table above. The maximum of the three-bit cache control field (cc) values of the LTB and GTB indicates the choice of caching, according to the table above.

No Cache

No Cache (NC) is an attribute that can be set on a LTB or GTB translation region to indicate that the cache is to be not to be consulted. No changes to the cache state result from reads or writes with this attribute set, (except for accesses that directly address the cache via memory-mapped region).

Cache Disable

Cache Disable (CD) is an attribute that can be set on a LTB or GTB translation region to indicate that the cache is to be consulted and updated for cache lines which are already present, but no new cache lines or sub-blocks are to be allocated when the cache does not already contain the addressed memory contents.

The "Socket 7" bus also provides a mechanism for supporting chip sets to decide on each access whether data is to be cached, using the CACHE# and KEN# signals. Using these signals, external hardware may cause a region selected as WT, WA or PF to be treated as CD. This mechanism is only active on the first such access to a memory region if caching is enabled, as the cache may satisfy subsequent references without a bus transaction.

Write Through

Write Through (WT) is an attribute that can be set on a LTB or GTB translation region to indicate that the writes to the cache must also immediately update backing memory. Reads to addressed memory that is not present in the cache cause cache lines or sub-blocks to be allocated. Writes to addressed memory that is not present in the cache does not modify cache state.

The "Socket 7" bus also provides a mechanism for supporting chip sets to decide on each access whether data is to be written through, using the PWT and WB/WT# signals. Using these signals, external hardware may cause a region selected as WA or PF to be treated as WT. This mechanism is only active on the first write to each region of memory; as on subsequent references, if the cache line is in the Exclusive or Modified state and writeback caching is enabled on the first reference, no subsequent bus operation occurs, at least until the cache line is flushed.

Write Allocate

Write allocate (WA) is an attribute that can be set of a LTB or GTB translation region to indicate that the processor is to allocate a memory block to the cache when the data is not previously present in the cache and the operation to be performed is a store. Reads to addressed memory that is not present in the cache cause cache lines or sub-blocks to be allocated. For cacheable data, write allocate is generally the preferred policy, as allocating the data to the cache reduces further bus traffic for subsequent references (loads or stores) or the data. Write allocate never occurs for data which is not cached. A write allocate brings in the data immediately into the Modified state.

Other "socket 7" processors have the ability to inhibit write allocate to cached locations under certain conditions, related by the address range. K6, for example, can inhibit write allocate in the range of 15-16 Mbyte, or for all addresses above a configurable limit with 4 Mbyte granularity. Pentium has the ability to label address ranges over which write allocate can be inhibited.

PreFetch

Prefetch (PF) is an attribute that can be set on a LTB or GTB translation region to indicate that increased prefetching is appropriate for references in this region. Each program fetch, load or store to a cache line that or does not already contain all the sub-blocks causes a prefetch allocation of the remaining sub-blocks. Cache misses cause allocation of the requested sub-block and prefetch allocation of the remaining sub-blocks. Prefetching does not necessarily fill in the entire cache line, as prefetch memory references are performed at a lower priority to other cache and memory reference traffic. A limited number of prefetches (as low as one in the initial implementation) can be queued; the older prefetch requests are terminated as new ones are created.

In other respects, the PF attribute is handled in the manner of the WA attribute. Prefetching is considered an implementation-dependent feature, and an implementation may choose to implement region with the PF attribute exactly as with the WA attribute.

Implementations may perform even more aggressive prefetching in future versions. Data may be prefetched into the cache in regions that are cacheable, as a result of program fetches, loads or stores to nearby addresses. Prefetches may extend beyond the cache line associated with the nearby address. Prefetches shall not occur beyond the reach of the GTB entry associated with the nearby address. Prefetching is terminated if an attempted cache fill results in a bus response that is not cacheable. Prefetches are implementation-dependent behavior, and such behavior may vary as a result of other memory references or other bus activity.

SubStream

SubStream (SS) is an attribute that can be set on a LTB or GTB translation region to indicate that references in this region are to be selected as the next victim on a cache miss. In particular, cache misses, which normally place the cache line in the last-to-be-victim state, instead place the cache line in the first-to-be-victim state, except relative to cache lines in the I state.

In other respects, the SS attribute is handled in the manner of the WA attribute. SubStream is considered an implementation-dependent feature, and an implementation may choose to implement region with the SS attribute exactly as with the WA attribute.

The SubStream attribute is appropriate for regions which are large data structures in which the processor is likely to reference the memory data just once or a small number of times, but for which the cache permits the data to be fetched using burst transfers. By making it a priority for victimization, these references are less likely to interfere with caching of data for which the cache performs a longer-term storage function.

LineStream

LineStream (LS) is an attribute that can be set on a LTB or GTB translation region to indicate that references in this region are to be selected as the next victim on a cache miss, and to enable prefetching. In particular, cache misses, which normally place the cache line in the last-to-be-victim state, instead place the cache line in the first-to-be-victim state, except relative to cache lines in the I state.

In other respects, the LS attribute is handled in the manner of the PF attribute. LineStream is considered an implementation-dependent feature, and an implementation may choose to implement region with the SS attribute exactly as with the PF or WA attributes.

Like the SubStream attribute, the LineStream attribute is particularly appropriate for regions for which large data structures are used in sequential fashion. By prefetching the entire cache line, memory traffic is performed as large sequential bursts of at least 256 bytes, maximizing the available bus utilization.

Cache Coherence

Cache coherency is maintained by using MESI protocols, for which each cache line (256 bytes) the cache data is kept in one of four states: M, E, S, I:

State		this Cache data	other Cache data	Memory data
Modified	3	Data is held exclusively in this cache.	No data is present in other caches.	The contents of main memory are now invalid.
Exclusive	2	Data is held exclusively in this cache.	No data is present in other caches.	Data is the same as the contents of main memory
Shared	1	Data is held in this cache, and possibly others.	Data is possibly in other caches.	Data is the same as the contents of main memory.
Invalid	0	No data for this location is present in the cache.	Data is possibly in other caches.	Data is possibly present in main memory.

The state is contained in the mesi field of the cache tag.

In addition, because the "Socket 7" bus performs block transfers and cache coherency actions on triclet (32 byte) blocks, each cache line also maintains 8 bits of triclet valid (tv) state. Each bit of tv corresponds to a triclet sub-block of the cache line; bit **0** for bytes **0** . . . **31**, bit **1** for bytes **32** . . . **63**, bit **2** for bytes **64** . . . **95**, etc. If the tv bit is zero (0), the coherence state for that triclet is I, no matter what the value of the mesi field. If the tv bit is one (1), the coherence state is defined by the mesi field. If all the tv bits are cleared (0), the mesi field must also be cleared, indicating an invalid cache line.

Cache coherency activity generally follows the protocols defined by the "Socket 7" bus, as defined by Pentium and K6-2 documentation. However, because the coherence state of a cache line is represented in only 10 bits per 256 bytes (1.25 bits per triclet), a few state transitions are defined differently. The differences are a direct result of attempts to set triclets within a cache line to different MES states that cannot be represented. The data structure allows any triclet to be changed to the I state, so state transitions in this direction match the Pentium processor exactly.

On the Pentium processor, for a cache line in the M state, an external bus Inquiry cycle that does not require invalidation (INV=0) places the cache line in the S state. On the Zeus processor, if no other triclet in the cache line is valid, the mesi field is changed to S. If other triclets in the cache line are valid, the mesi field is left unchanged, and the tv bit for this triclet is turned off, effectively changing it to the I state.

On the Pentium processor, for a cache line in the E state, an external bus Inquiry cycle that does not require invalidation (INV=0) places the cache line in the S state. On the Zeus processor, the mesi field is changed to S. If other triclets in the cache line are valid, the MESI state is effectively changed to the S state for these other triclets.

On the Pentium processor, for a cache line in the S state, an internal store operation causes a write-through cycle and a transition to the E state. On the Zeus processor, the mesi field is changed to E. Other triclets in the cache line are invalidated by clearing the tv bits; the MESI state is effectively changed to the I state for these other triclets.

When allocating data into the cache due to a store operation, data is brought immediately into the Modified state, setting the mesi field to M. If the previous mesi field is S, other triclets which are valid are invalidated by clearing the tv bits. If the previous mesi field is E, other triclets are kept valid and therefore changed to the M state.

When allocating data into the cache due to a load operation, data is brought into the Shared state, if another processor reports that the data is present in its cache or the mesi field is already set to S, the Exclusive state, if no processor reports that the data is present in its cache and the mesi field is currently E or I, or the Modified state if the mesi field is already set to M. The determination is performed by driving PWT low and checking whether WB/WT# is sampled high; if so the line is brought into the Exclusive state. (See page 202 (184) of the K6-2 documentation).

Strong Ordering

Strong ordering (so) is an attribute which permits certain memory regions to be operated with strong ordering, in which all memory operations are performed exactly in the order specified by the program and others to be operated with weak ordering, in which some memory operations may be performed out of program order.

The Zeus processor controls strong ordering as an attribute in the LTB and GTB, thus software may set this attribute for certain address ranges and clear it for others. A one bit field indicates the choice of access ordering. A one (1) bit indicates strong ordering, while a zero (0) bit indicates weak ordering.

With weak ordering, the memory system may retain store operations in a store buffer indefinitely for later storage into the memory system, or until a synchronization operation to any address performed by the thread that issued the store operation forces the store to occur. Load operations may be performed in any order, subject to requirements that they be performed logically subsequent to prior store operations to the same address, and subsequent to prior synchronization operations to any address. Under weak ordering it is permitted to forward results from a retained store operation to a future load operation to the same address. Operations are considered to be to the same address when any bytes of the operation are in common. Weak ordering is usually appropriate for conventional memory regions, which are side-effect free.

With strong ordering, the memory system must perform load and store operations in the order specified. In particular, strong-ordered load operations are performed in the order specified, and all load operations (whether weak or strong) must be delayed until all previous strong-ordered store operations have been performed, which can have a significant performance impact. Strong ordering is often required for memory-mapped I/O regions, where store operations may have a side-effect on the value returned by loads to other addresses. Note that Zeus has memory-mapped I/O, such as the TB, for which the use of strong ordering is essential to proper operation of the virtual memory system.

The EWBE# signal in "Socket 7" is of importance in maintaining strong ordering. When a write is performed with the signal inactive, no further writes to E or M state lines may occur until the signal becomes active. Further details are given in Pentium documentation (K6-2 documentation may not apply to this signal.)

Victim Selection

One bit of the cache tag, the vs bit, controls the selection of which set of the four sets at a cache address should next be chosen as a victim for cache line replacement. Victim selection (vs) is an attribute associated with LOC cache blocks. No vs bits are present in the LTB or GTB.

There are two hexlets of tag information for a cache line, and replacement of a set requires writing only one hexlet. To update priority information for victim selection by writing only one hexlet, information in each hexlet is combined by an exclusive-or. It is the nature of the exclusive- or function that altering either of the two hexlets can change the priority information.

Full Victim Selection Ordering for Four Sets

There are 4*3*2*1=24 possible orderings of the four sets, which can be completely encoded in as few as 5 bits: 2 bits to indicate highest priority, 2 bits for second-highest priority, 1 bit for third-highest priority, and 0 bits for lowest priority. Dividing this up per set and duplicating per hexlet with the exclusive- or scheme above requires three bits per set, which suggests simply keeping track of the three-highest priority sets with 2 bits each, using 6 bits total and three bits per set.

Specifically, vs bits from the four sets are combined to produce a 6-bit value:

$$vsc \leftarrow (vs[3] \| vs[2])\hat{}(vs[1] \| vs[0])$$

The highest priority for replacement is set $vsc_{1 \ldots 0}$, second highest priority is set $vsc_{3 \ldots 2}$, third highest priority is set $vsc_{5 \ldots 4}$, and lowest priority is $vsc_{5 \ldots 4}\hat{}vsc_{3 \ldots 2}\hat{}vsc_{1 \ldots 0}$. When the highest priority set is replaced, it becomes the new lowest priority and the others are moved up, computing a new vsc by:

$$vsc \leftarrow vsc_{5 \ldots 4}\hat{}vsc_{3 \ldots 2}\hat{}vsc_{1 \ldots 0} \| vsc_{5 \ldots 2}$$

When replacing set vsc for a LineStream or SubStream replacement, the priority for replacement is unchanged, unless another set contains the invalid MESI state, computing a new vsc by:

$$\begin{aligned}vsc \leftarrow\ & mesi[vsc_{5 \ldots 4}\hat{}vsc_{3 \ldots 2}\hat{}vsc_{1 \ldots 0}]=I)? \\ & vsc_{5 \ldots 4}\hat{}vsc_{3 \ldots 2}\hat{}vsc_{1 \ldots 0} \| vsc_{5 \ldots 2}: \\ & (mesi[vsc_{5 \ldots 4}]=I)?vsc_{1 \ldots 0} \| vsc_{5 \ldots 2}:(mesi \\ & [vsc_{3 \ldots 2}]=I)?vsc_{5 \ldots 4} \| vsc_{1 \ldots 0} \| vsc_{3 \ldots 2}:vsc\end{aligned}$$

Cache flushing and invalidations can cause cache lines to be cleared out of sequential order. Flushing or invalidating a cache line moves that set to highest priority. If that set is already highest priority, the vsc is unchanged. If the set was second or third highest or lowest priority, the vsc is changed to move that set to highest priority, moving the others down.

$$\begin{aligned}vsc \leftarrow\ & ((fs=vsc_{1 \ldots 0} \text{ or } fs=vsc_{3 \ldots 2})?vsc_{5 \ldots 4}: \\ & vsc_{3 \ldots 2}) \| (fs=vsc_{1 \ldots 0}?vsc_{3 \ldots 2}:vsc_{1 \ldots 0}) \| fs\end{aligned}$$

When updating the hexlet containing vs[1] and vs[0], the new values of vs[1] and vs[0] are:

$$vs[1] \leftarrow vs[3]\hat{}vsc_{5 \ldots 3}$$

$$vs[0] \leftarrow vs[2]\hat{}vsc_{2 \ldots 0}$$

When updating the hexlet containing vs[3] and vs[2], the new values of vs[3] and vs[2] are:

$$vs[3] \leftarrow vs[1]\hat{}vsc_{5 \ldots 3}$$

$$vs[2] \leftarrow vs[0]\hat{}vsc_{2 \ldots 0}$$

Software must initialize the vs bits to a legal, consistent state. For example, to set the priority (highest to lowest) to (0, 1, 2, 3), vsc must be set to 0b100100. There are many legal solutions that yield this vsc value, such as vs[3]←0, vs[2]←0, vs[1]←4,vs[0]←4.

Simplified Victim Selection Ordering for Four Sets

However, the orderings are simplified in the first Zeus implementation, to reduce the number of vs bits to one per set, keeping a two bit vsc state value:

$$vsc \leftarrow (vs[3] \| vs[2])\hat{}(vs[1] \| vs[0])$$

The highest priority for replacement is set vsc, second highest priority is set vsc+1, third highest priority is set vsc+2, and lowest priority is vsc+3. When the highest priority set is replaced, it becomes the new lowest priority and the others are moved up. Priority is given to sets with invalid MESI state, computing a new vsc by:

$$\begin{aligned}vsc \leftarrow\ & mesi[vsc+1]=I)?vsc+1:(mesi[vsc+2]=I)?vsc+2: \\ & (mesi[vsc+3]=I)?vsc+3:vsc+1\end{aligned}$$

When replacing set vsc for a LineStream or SubStream replacement, the priority for replacement is unchanged, unless another set contains the invalid MESI state, computing a new vsc by:

$$\begin{aligned}vsc \leftarrow\ & mesi[vsc+1]=I)?vsc+1:(mesi[vsc+2]=I)?vsc+2: \\ & (mesi[vsc+3]=I)?vsc+3:vsc\end{aligned}$$

Cache flushing and invalidations can cause cache sets to be cleared out of sequential order. If the current highest priority for replacement is a valid set, the flushed or invalidated set is made highest priority for replacement.

$$vsc \leftarrow (mesi[vsc]=I)?vsc:fs$$

When updating the hexlet containing vs[1] and vs[0], the new values of vs[1] and vs[0] are:

$$vs[1] \leftarrow vs[3]\hat{}vsc_1$$

$$vs[0] \leftarrow vs[2]\hat{}vsc_0$$

When updating the hexlet containing vs[3] and vs[2], the new values of vs[3] and vs[2] are:

$$vs[3] \leftarrow vs[1]\hat{\ }vsc_1$$

$$vs[2] \leftarrow vs[0]\hat{\ }vsc_0$$

Software must initialize the vs bits, but any state is legal. For example, to set the priority (highest to lowest) to (0, 1, 2, 3), vsc must be set to 0b00. There are many legal solutions that yield this vsc value, such as vs[3]←0, vs[2]←0, vs[1]←0,vs[0]←0.

Full Victim Selection Ordering for Additional Sets

To extend the full-victim-ordering scheme to eight sets, 3*7=21 bits are needed, which divided among two tags is 11 bits per tag. This is somewhat generous, as the minimum required is 8*7*6*5*4*3*2*1=40320 orderings, which can be represented in as few as 16 bits. Extending the full-victim-ordering four-set scheme above to represent the first 4 priorities in binary, but to use 2 bits for each of the next 3 priorities requires 3+3+3+3+2+2+2=18 bits. Representing fewer distinct orderings can further reduce the number of bits used. As an extreme example, using the simplified scheme above with eight sets requires only 3 bits, which divided among two tags is 2 bits per tag.

Victim Selection Without LOC Tag Bits

At extreme values of the niche limit register (nl in the range **121** . . . **124**), the bit normally used to hold the vs bit is usurped for use as a physical address bit. Under these conditions, no vsc value is maintained per cache line, instead a single, global vsc value is used to select victims for cache replacement. In this case, the cache consists of four lines, each with four sets. On each replacement a new si valus is computed from:

$$gvsc \leftarrow gvsc+1$$

$$si \leftarrow gvsc\hat{\ }pa_{11\,\ldots\,10}$$

The algorithm above is designed to utilize all four sets on sequential access to memory.

Victim Selection Encoding LOC Tag Bits

At even more extreme values of the niche limit register (nl in the range **125** . . . **127**), not only is the bit normally used to hold the vs bit is usurped for use as a physical address bit, but there is a deficit of one or two physical address bits. In this case, the number of sets can be reduced to encode physical address bits into the victim selection, allowing the choice of set to indicate physical address bits **9** or bits **9** . . . **8**. On each replacement a new vsc valus is computed from:

$$gvsc \leftarrow gvsc+1$$

$$si \leftarrow pa_9 \| (n1=127)?pa_8{:}gvsc\hat{\ }pa_{10}$$

The algorithm above is designed to utilize all four sets on sequential access to memory.

Detail Access

Detail access is an attribute which can be set on a cache block or translation region to indicate that software needs to be consulted on each potential access, to determine whether the access should proceed or not. Setting this attribute causes an exception trap to occur, by which software can examine the virtual address, by for example, locating data in a table, and if indicated, causes the processor to continue execution. In continuing, ephemeral state is set upon returning to the re re-execution of the instruction that prevents the exception trap from recurring on this particular re-execution only. The ephemeral state is cleared as soon as the instruction is either completed or subject to another exception, so DetailAccess exceptions can recur on a subsequent execution of the same instruction. Alternatively, if the access is not to proceed, execution has been trapped to software at this point, which can abort the thread or take other correction action.

The detail access attribute permits specification of access parameters over memory region on arbitrary byte boundaries. This is important for emulators, which must prevent store access to code which has been translated, and for simulating machines which have byte granularity on segment boundaries. The detail access attribute can also be applied to debuggers, which have the need to set breakpoints on byte-level data, or which may use the feature to set code breakpoints on instruction boundaries without altering the program code, enabling breakpoints on code contained in ROM.

A one bit field indicates the choice of detail access. A one (1) bit indicates detail access, while a zero (0) bit indicates no detail access. Detail access is an attribute that can be set by the LTB, the GTB, or a cache tag.

The table below indicates the proper status for all potential values of the detail access bits in the LTB, GTB, and Tag:

LTB	GTB	Tag	status
0	0	0	OK - normal
0	0	1	AccessDetailRequiredByTag
0	1	0	AccessDetailRequiredByGTB
0	1	1	OK - GTB inhibited by Tag
1	0	0	AccessDetailRequiredByLTB
1	0	1	OK - LTB inhibited by Tag
1	1	0	OK - LTB inhibited by GTB
1	1	1	AccessDetailRequiredByTag
0	Miss		GTBMiss
1	Miss		AccessDetailRequiredByLTB
0	0	Miss	Cache Miss
0	1	Miss	AccessDetailRequiredByGTB
1	0	Miss	AccessDetailRequiredByLTB
1	1	Miss	Cache Miss

The first eight rows show appropriate activities when all three bits are available. The detail access attributes for the LTB, GTB, and cache tag work together to define whether and which kind of detail access exception trap occurs. Generally, setting a single attribute bit causes an exception, while setting two bits inhibits such exceptions. In this way, a detail access exception can be narrowed down to cause an exception over a specified region of memory: Software generally will set the cache tag detail access bit only for regions in which the LTB or GTB also has a detail access bit set. Because cache activity may flush and refill cache lines implicity, it is not generally useful to set the cache tag detail access bit alone, but if this occurs, the AccessDetailRequiredByTag exception catches such an attempt.

The next two rows show appropriate activities on a GTB miss. On a GTB miss, the detail access bit in the GTB is not present. If the LTB indicates detail access and the GTB misses, the AccessDetailRequiredByLTB exception should be indicated. If software continues from the AccessDetailRequiredByLTB exception and has not filled in the GTB, the GTBMiss exception happens next. Since the GTBMiss exection is not a continuation exception, a re-execution after the GTBMiss exception can cause a reoccurrence of the AccessDetailRequiredByLTB exception. Alternatively, if software continues from the AccessDetailRequiredByLTB exception and has filled in the GTB, the AccessDetailRequiredByLTB exception is inhibited for that reference, no matter what the status of the GTB and Tag detail bits, but the re-executed instruction is still subject to the AccessDetailRequiredByGTB and AccessDetailRequiredByTag exceptions.

The last four rows show appropriate activities for a cache miss. On a cache miss, the detail access bit in the tag is not present. If the LTB or GTB indicates detail access and the cache misses, the AccessDetailRequiredByLTB or AccessDetailRequiredByGTB exception should be indicated. If software continues from these exceptions and has not filled in the cache, a cache miss happens next. If software continues from the AccessDetailRequiredByLTB or AccessDetailRequiredByGTB exception and has filled in the cache, the previous exception is inhibited for that reference, no matter what the status of the Tag detail bit, but is still subject to the AccessDetailRequiredByTag exception. When the detail bit must be created from a cache miss, the initial value filled in is zero. Software may set the bit, thus turning off AccessDetailRequired exceptions per cache line. If the cache line is flushed and refilled, the detail access bit in the cache tag is again reset to zero, and another AccessDetailRequired exception occurs.

Settings of the niche limit parameter to values that require use of the da bit in the LOC tag for retaining the physical address usurp the capability to set the Tag detail access bit. Under such conditions, the Tag detail access bit is effectively always zero (0), so it cannot inhibit AccessDetailRequiredByLTB, inhibit AccessDetailRequiredByGTB, or cause AccessDetailRequiredByTag.

The execution of a Zeus instruction has a reference to one quadlet of instruction, which may be subject to the DetailAccess exceptions, and a reference to data, which may be unaligned or wide. These unaligned or wide references may cross GTB or cache boundaries, and thus involve multiple separate reference that are combined together, each of which may be subject to the DetailAccess exception. There is sufficient information in the DetailAccess exception handler to process unaligned or wide references.

The implementation is free to indicate DetailAccess exceptions for unaligned and wide data references either in combined form, or with each sub-reference separated. For example, in an unaligned reference that crosses a GTB or cache boundary, a DetailAccess exception may be indicated for a portion of the reference. The exception may report the virtual address and size of the complete reference, and upon continuing, may inhibit reoccurrence of the DetailAccess exception for any portion of the reference. Alternatively, it may report the virtual address and size of only a reference portion and inhibit reoccurrence of the DetailAccess exception for only that portion of the reference, subject to another DetailAccess exception occurring for the remaining portion of the reference.

Micro Translation Buffer

The Micro Translation Buffer (MTB) is an implementation-dependent structure which reduces the access traffic to the GTB and the LOC tags. The MTB contains and caches information read from the GTB and LOC tags, and is consulted on each access to the LOC.

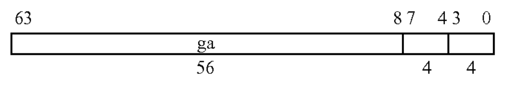
To access the LOC, a global address is supplied to the Micro-Translation Buffer (MTB), which associatively looks up the global address into a table holding a subset of the LOC tags. In addition, each table entry contains the physical address bits **14** . . . **8** (7 bits) and set identifier (2 bits) required to access the LOC data.

In the first Zeus implementation, there are two MTB blocks—MTB 0 is used for threads 0 and 1, and MTB 1 is used for threads 2 and 3. Per clock cycle, each MTB block can check for 4 simultaneous references to the LOC. Each MTB block has 16 entries.

Each MTB entry consists of a bit less than 128 bits of information, including a 56-bit global address tag, 8 bits of

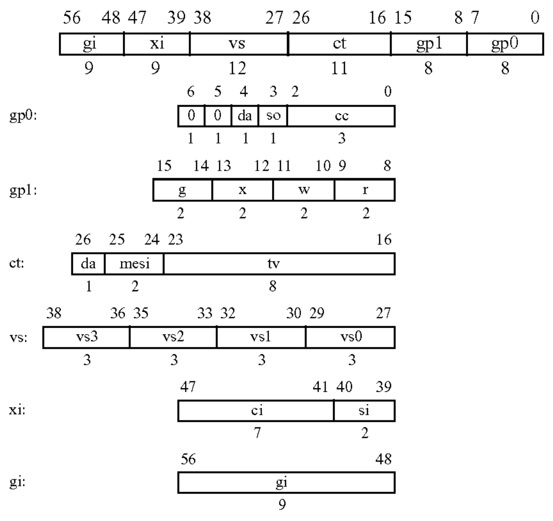
privilege level required for read, write, execute, and gateway access, a detail bit, and 10 bits of cache state indicating for each triclet (32 bytes) sub-block, the MESI state.

Match



Output

The output of the MTB combines physical address and protection information from the GTB and the referenced cache line.



The meaning of the fields are given by the following table:

name	size	meaning
ga	56	global address
gi	9	GTB index
ci	7	cache index
si	2	set index
vs	12	victim select
da	1	detail access (from cache line)
mesi	2	coherency: modified (3), exclusive (2), shared (1), invalid (0)
tv	8	triclet valid (1) or invalid (0)
g	2	minimum privilege required for gateway access
x	2	minimum privilege required for execute access
w	2	minimum privilege required for write access
r	2	minimum privilege required for read access
0	1	reserved
da	1	detail access (from GTB)
so	1	strong ordering
cc	3	cache control

With an MTB hit, the resulting cache index (14 . . . 8 from the MTB, bit **7** from the LA) and set identifier (2 bits from the MTB) are applied to the LOC data bank selected from bits **6** . . . **4** of the GVA. The access protection information (pr and rwxg) is supplied from the MTB.

With an MTB (and BTB) miss, a victim entry is selected for replacement. The MTB and BTB are always clean, so the victim entry is discarded without a writeback. The GTB (described below) is referenced to obtain a physical address and protection information. Depending on the access information in the GTB, either the MTB or BTB is filled.

Note that the processing of the physical address $pa_{14 \ldots 8}$ against the niche limit nl can be performed on the physical address from the GTB, producing the LOC address, ci. The LOC address, after processing against the nl is placed into the MTB directly, reducing the latency of an MTB hit.

Four tags are fetched from the LOC tags and compared against the PA to determine which of the four sets contain the data. If one of the four sets contains the correct physical address, a victim MTB entry is selected for replacement, the MTB is filled and the LOC access proceeds. If none of the four sets is a hit, an LOC miss occurs.

MTB miss	GTB cam	LOC tag	MTB fill
MTB victim			
	LOC miss		

The operation of the MTB is largely not visible to software—hardware mechanisms are responsible for automatically initializing, filling and flushing the MTB. Activity that modifies the GTB or LOC tag state may require that one or more MTB entries are flushed.

A write to the GTBUpdate register that updates a matching entry, a write to the GTBUpdateFill register, or a direct write to the GTB all flush relevant entries from the MTB. MTB flushing is accomplished by searching MTB entries for values that match on the gi field with the GTB entry that has been modified. Each such matching MTB entry is flushed.

The MTB is kept synchronous with the LOC tags, particularly with respect to MESI state. On an LOC miss or LOC snoop, any changes in MESI state update (or flush) MTB entries which physically match the address. If the MTB may contain less than the full physical address: it is sufficient to retain the LOC physical address (ci||v||si).

Block Translation Buffer

Zeus has a per thread "Block Translation Buffer" (BTB). The BTB retains GTB information for uncached address blocks. The BTB is used in parallel with the MTB—exactly one of the BTB or MTB may translate a particular reference. When both the BTB and MTB miss, the GTB is consulted, and depending on the result, the block is filled into either the MTB or BTB as appropriate. In the first Zeus implementation, the BTB has 2 entries for each thread.

BTB entries cover any power-of-two granularity, as they retain the size information from the GTB. BTB entries contain no MESI state, as they only contain uncached blocks.

Each BTB entry consists of 128 bits of information, containing the same information in the same format as a GTB entry.

Niche blocks are indicated by GTB information, and correspond to blocks of data that are retained in the LOC and never miss. A special physical address range indicates niche blocks. For this address range, the BTB enables use of the LOC as a niche memory, generating the "set select" address bits from low-order address bits. There is no checking of the LOC tags for consistent use of the LOC as a niche—the nl field must be preset by software so that LOC cache replacement never claims the LOC niche space, and only BTB miss and protection bits prevent software from using the cache portion of the LOC as niche.

Other address ranges include other on-chip resources, such as bus interface registers, the control register and status register, as well as off-chip memory, accessed through the bus interface. Each of these regions are accessible as uncached memory.

Program Translation Buffer

Later implementations of Zeus may optionally have a per thread "Program Translation Buffer" (PTB). The PTB retains GTB and LOC cache tag information. The PTB enables generation of LOC instruction fetching in parallel with load/store fetching. The PTB is updated when instruction fetching crosses a cache line boundary (each 64 instructions in straight-line code). The PTB functions similarly to a one-entry MTB, but can use the sequential nature of program code fetching to avoid checking the 56-bit match. The PTB is flushed at the same time as the MTB.

The initial implementation of Zeus has no PTB—the MTB suffices for this function.

Global Virtual Cache

The initial implementation of Zeus contains cache which is both indexed and tagged by a physical address. Other prototype implementations have used a global virtual address to index and/or tag an internal cache. This section will define the required characteristics of a global virtually-indexed cache. TODO

Memory Interface

Dedicated hardware mechanisms are provided to fetch data blocks in the levels zero and one caches, provided that a matching entry can be found in the MTB or GTB (or if the MMU is disabled). Dedicated hardware mechanisms are provided to store back data blocks in the level zero and one caches, regardless of the state of the MTB and GTB. When no entry is to be found in the GTB, an exception handler is invoked either to generate the required information from the virtual address, or to place an entry in the GTB to provide for automatic handling of this and other similarly addressed data blocks.

The initial implementation of Zeus accesses the remainder of the memory system through the "Socket 7" interface. Via this interface, Zeus accesses a secondary cache, DRAM memory, external ROM memory, and an I/O system The size and presence of the secondary cache and the DRAM memory array, and the contents of the external ROM memory and the I/O system are variables in the processor environment.

Microarchitecture

Each thread has two address generation units, capable of producing two aligned, or one unaligned load or store operation per cycle. Alternatively, these units may produce a single load or store address and a branch target address.

Each thread has a LTB, which translates the two addresses into global virtual addresses.

Each pair of threads has a MTB, which looks up the four references into the LOC. The PTB provides for additional references that are program code fetches.

In parallel with the MTB, these four references are combined with the four references from the other thread pair and partitioned into even and odd hexlet references. Up to four references are selected for each of the even and odd portions of the LZC. One reference for each of the eight banks of the LOC (four are even hexlets; four are odd hexlets) are selected from the eight load/store/branch references and the PTB references.

Some references may be directed to both the LZC and LOC, in which case the LZC hit causes the LOC data to be ignored. An LZC miss which hits in the MTB is filled from the LOC to the LZC. An LZC miss which misses in the MTB causes a GTB access and LOC tag access, then an MTB fill and LOC access, then an LZC fill.

Priority of access: (highest/lowest) cache dump, cache fill, load, program, store.

Snoop

The "Socket 7" bus requires certain bus accesses to be checked against on-chip caches. On a bus read, the address is checked against the on-chip caches, with accesses aborted when requested data is in an internal cache in the M state, and the E state, the internal cache is changed to the S state. On a bus write, data written must update data in on-chip caches. To meet these requirements, physical bus addresses must be checked against the LOC tags.

The S7 bus requires that responses to inquire cycles occur with fixed timing. At least with certain combinations of bus and processor clock rate, inquire cycles will require top priority to meet the inquire response timing requirement.

Synchronization operations must take into account bus activity—generally a synchronization operation can only proceed on cached data which is in Exclusive or Modified—if cached data in Shared state, ownership must be obtained. Data that is not cached must be accessed using locked bus cycles.

Load

Load operations require partitioning into reads that do not cross a hexlet (128 bit) boundary, checking for store conflicts, checking the LZC, checking the LOC, and reading from memory. Execute and Gateway accesses are always aligned and since they are smaller than a hexlet, do not cross a hexlet boundary.

Note: S7 processors perform unaligned operations LSB first, MSB last, up to 64 bits at a time. Unaligned 128 bit loads need 3 64-bit operations, LSB, octlet, MSB. Transfers which are smaller than a hexlet but larger than an octlet are further divided in the S7 bus unit.

Definition

```
def data ← LoadMemoryX(ba,la,size,order)
    assert (order = L) and ((la and (size/8−1)) = 0) and (size = 32)
    hdata ← TranslateAndCacheAccess(ba,la,size,X,0)
    data ← $hdata_{31+8*(la\ and\ 15)..8*(la\ and\ 15)}$
enddef
def data ← LoadMemoryG(ba,la,size,order)
    assert (order = L) and ((la and (size/8−1)) = 0) and (size = 64)
    hdata ← TranslateAndCacheAccess(ba,la,size,G,0)
    data ← $hdata_{63+8*(la\ and\ 15)..8*(la\ and\ 15)}$
enddef
def data ← LoadMemory(ba,la,size,order)
    if (size > 128) then
        data0 ← LoadMemory(ba, la,size/2, order)
        data1 ← LoadMemory(ba, la+(size/2), size/2, order)
        case order of
            L:
                data ← data1 || data0
            B:
                data ← data0 || data1
        endcase
    else
        bs ← 8*$la_{4..0}$
        be ← bs + size
        if be > 128 then
            data0 ← LoadMemory(ba, la, 128 − bs, order)
            data1 ← LoadMemory(ba, ($la_{63..5}$ + 1) || $0^4$,
            be − 128, order)
            case order of
                L:
                    data ← (data1 || data0)
                B:
                    data ← (data0 || data1)
            endcase
        else
            hdata ← TranslateAndCacheAccess(ba,la,size,R,0)
            for i ← 0 to size−8 by 8
                j ← bs + ((order=L) ? i : size−8−i)
                $data_{i+7..i}$ ← $hdata_{j+7..j}$
            endfor
        endif
    endif
enddef
```

Store

Store operations requires partitioning into stores less than 128 bits that do not cross hexlet boundaries, checking for store conflicts, checking the LZC, checking the LOC, and storing into memory.

Definition

```
def StoreMemory(ba,la,size,order,data)
    bs ← 8*$la_{4..0}$
    be ← bs + size
    if be > 128 then
        case order of
            L:
                data0 ← $data_{127-bs..0}$
                data1 ← $data_{size-1..128-bs}$
            B:
                data0 ← $data_{size-1..be-128}$
                data1 ← $data_{be-129..0}$
        endcase
        StoreMemory(ba, la, 128 − bs, order, data0)
        StoreMemory(ba, ($la_{63..5}$ + 1) || $0^4$, be − 128, order, data1)
    else
        for i ← 0 to size−8 by 8
            j ← bs + ((order=L) ? i : size−8−i)
            $hdata_{j+7..j}$ ← $data_{i+7..i}$
        endfor
        xdata ← TranslateAndCacheAccess(ba, la, size, W, hdata)
    endif
enddef
```

Memory

Memory operations require first translating via the LTB and GTB, checking for access exceptions, then accessing the cache.

Definition

```
def hdata ← TranslateAndCacheAccess(ba,la,size,rwxg,hwdata)
    if $ControlRegister_{62}$ then
        case rwxg of
            R:
                at ← 0
            W:
                at ← 1
            X:
                at ← 2
            G:
                at ← 3
        endcase
        rw ← (rwxg=W) ? W : R
        ga,LocalProtect ← LocalTranslation(th,ba,la,pl)
        if $LocalProtect_{9+2*at..8+2*at}$ < pl then
            raise AccessDisallowedByLTB
        endif
        lda ← $LocalProtect_4$
        pa,GlobalProtect ← GlobalTranslation(th,ga,pl,lda)
        if $GlobalProtect_{9+2*at..8+2*at}$ < pl then
            raise AccessDisallowedByGTB
        endif
        cc ← ($LocalProtect_{2..0}$ > $GlobalProtect_{2..0}$) ?
        $LocalProtect_{2..0}$ : $GlobalProtect_{2..0}$
        so ← $LocalProtect_3$ or $GlobalProtect_3$
        gda ← $GlobalProtect_4$
        hdata,TagProtect ← LevelOneCacheAccess(pa,size,
        lda,gda,cc,rw,hwdata)
        if (lda ^ gda ^ TagProtect) = 1 then
            if TagProtect then
                PerformAccessDetail(AccessDetailRequiredByTag)
            elseif gda then
                PerformAccessDetail(AccessDetailRequiredByGlobalTB)
```

-continued

```
            else
                PerformAccessDetail(AccessDetailRequiredByLocalTB)
            endif
        endif
    else
        case rwxg of
            R, X, G:
                hdata ← ReadPhysical(la,size)
            W:
                WritePhysical(la,size,hwdata)
        endcase
    endif
enddef
```

Rounding and Exceptions

In accordance with one embodiment of the invention, rounding is specified within the instructions explicitly, to avoid explicit state registers for a rounding mode. Similarly, the instructions explicitly specify how standard exceptions (invalid operation, division by zero, overflow, underflow and inexact) are to be handled (U.S. Pat. No. 5,812,439 describes this "Technique of incorporating floating point information into processor instructions.").

In this embodiment, when no rounding is explicitly named by the instruction (default), round to nearest rounding is performed, and all floating-point exception signals cause the standard-specified default result, rather than a trap. When rounding is explicitly named by the instruction (N: nearest, Z: zero, F: floor, C: ceiling), the specified rounding is performed, and floating-point exception signals other than inexact cause a floating-point exception trap. When X (exact, or exception) is specified, all floating-point exception signals cause a floating-point exception trap, including inexact. More details regarding rounding and exceptions are described in the "Rounding and Exceptions" section.

This technique assists the Zeus processor in executing floating-point operations with greater parallelism. When default rounding and exception handling control is specified in floating-point instructions, Zeus may safely retire instructions following them, as they are guaranteed not to cause data-dependent exceptions. Similarly, floating-point instructions with N, Z, F, or C control can be guaranteed not to cause data-dependent exceptions once the operands have been examined to rule out invalid operations, division by zero, overflow or underflow exceptions. Only floating-point instructions with X control, or when exceptions cannot be ruled out with N, Z, F, or C control need to avoid retiring following instructions until the final result is generated.

ANSI/IEEE standard 754-1985 specifies information to be given to trap handlers for the five floating-point exceptions. The Zeus architecture produces a precise exception, (The program counter points to the instruction that caused the exception and all register state is present) from which all the required information can be produced in software, as all source operand values and the specified operation are available.

ANSI/IEEE standard 754-1985 specifies a set of five "sticky-exception" bits, for recording the occurrence of exceptions that are handled by default. The Zeus architecture produces a precise exception for instructions with N, Z, F, or C control for invalid operation, division by zero, overflow or underflow exceptions and with X control for all floating-point exceptions, from which corresponding sticky-exception bits can be set. Execution of the same instruction with default control will compute the default result with round-to-nearest rounding. Most compound operations not specified by the standard are not available with rounding and exception controls.

Instruction Set

This section describes the instruction set in complete architectural detail. Operation codes are numerically defined by their position in the following operation code tables, and are referred to symbolically in the detailed instruction definitions. Entries that span more than one location in the table define the operation code identifier as the smallest value of all the locations spanned. The value of the symbol can be calculated from the sum of the legend values to the left and above the identifier.

Instructions that have great similarity and identical formats are grouped together. Starting on a new page, each category of instructions is named and introduced.

The Operation codes section lists each instruction by mnemonic that is defined on that page. A textual interpretation of each instruction is shown beside each mnemonic.

The Equivalences section lists additional instructions known to assemblers that are equivalent or special cases of base instructions, again with a textual interpretation of each instruction beside each mnemonic. Below the list, each equivalent instruction is defined, either in terms of a base instruction or another equivalent instruction. The symbol between the instruction and the definition has a particular meaning. If it is an arrow (← or →), it connects two mathematically equivalent operations, and the arrow direction indicates which form is preferred and produced in a reverse assembly. If the symbol is a (⇐), the form on the left is assembled into the form on the right solely for encoding purposes, and the form on the right is otherwise illegal in the assembler. The parameters in these definitions are formal; the names are solely for pattern-matching purposes, even though they may be suggestive of a particular meaning.

The Redundancies section lists instructions and operand values that may also be performed by other instructions in the instruction set. The symbol connecting the two forms is a (⇔), which indicates that the two forms are mathematically equivalent, both are legal, but the assembler does not transform one into the other.

The Selection section lists instructions and equivalences together in a tabular form that highlights the structure of the instruction mnemonics.

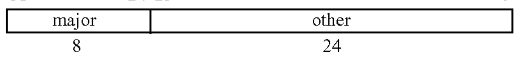
The Format section lists (1) the assembler format, (2) the C intrinsics format, (3) the bit-level instruction format, and (4) a definition of bit-level instruction format fields that are not a one-for-one match with named fields in the assembler format.

The Definition section gives a precise definition of each basic instruction.

The Exceptions section lists exceptions that may be caused by the execution of the instructions in this category.

Major Operation Codes

All instructions are 32 bits in size, and use the high order 8 bits to specify a major operation code.



The major field is filled with a value specified by the following table (Blank table entries cause the Reserved Instruction exception to occur.):

major operation code field values

MAJOR	0	32	64	96	128	160	192	224
0	ARES	BEF16	LI16L	SI16L		XDEPOSIT	EMULXI	WMULMATXIL
1	AADDI	BEF32	LI16B	SI16B	GADDI		EMULXIU	WMULMATXIB
2	AADDI.O	BEF64	LI16AL	SI16AL	GADDI.O		EMULXIM	WMULMATXIUL
3	AADDIU.O	BEF128	LI16AB	SI16AB	GADDIU.O		EMULXIC	WMULMATXIUB
4		BLGF16	LI32L	SI32L		XDEPOSITU	EMULADDXI	WMULMATXIML
5	ASUBI	BLGF32	LI32B	SI32B	GSUBI		EMULADDXIU	WMULMATXIMB
6	ASUBI.O	BLGF64	LI32AL	SI32AL	GSUBI.O		EMULADDXIM	WMULMATXICL
7	ASUBIU.O	BLGF128	LI32AB	SI32AB	GSUBIU.O		EMULADDXIC	WMULMATXICB
8	ASETEI	BLF16	LI64L	SI64L	GSETEI	XWITHDRAW	ECONXIL	
9	ASETNEI	BLF32	LI64B	SI64B	GSETNEI		ECONXIB	
10	ASETANDEI	BLF64	LI64AL	SI64AL	GSETANDEI		ECONXIUL	
11	ASETANDNEI	BLF128	LI64AB	SI64AB	GSETANDNEI		ECONXIUB	
12	ASETLI	BGEF16	LI128L	SI128L	GSETLI	XWITHDRAWU	ECONXIML	
13	ASETGEI	BGEF32	LI128B	SI128B	GSETGEI		ECONXIMB	
14	ASETLIU	BGEF64	LI128AL	SI128AL	GSETLIU		ECONXICL	
15	ASETGEIU	BGEF128	LI128AB	SI128AB	GSETGEIU		ECONXICB	
16	AANDI	BE	LIU16L	SASI64AL	GANDI	XDEPOSITM	ESCALADDF16	WMULMATXL
17	ANANDI	BNE	LIU16B	SASI64AB	GNANDI		ESCALADDF32	WMULMATXB
18	AORI	BANDE	LIU16AL	SCSI64AL	GORI		ESCALADDF64	WMULMATGL
19	ANORI	BANDNE	LIU16AB	SCSI64AB	GNORI		ESCALADDX	WMULMATGB
20	AXORI	BL	LIU32L	SMSI64AL	GXORI	XSWIZZLE	EMULG8	
21	AMUX	BGE	LIU32B	SMSI64AB	GMUX		EMULG64	
22		BLU	LIU32AL	SMUXI64AL	GBOOLEAN		EMULX	
23		BGEU	LIU32AB	SMUXI64AB			EEXTRACT	
24	ACOPYI	BVF32	LIU64L		GCOPYI	XEXTRACT	EEXTRACTI	
25		BNVF32	LIU64B			XSELECT8	EEXTRACTIU	
26		BIF32	LIU64AL					WTABLEL
27		BNIF32	LIU64AB		G8		E.8	WTABLEB
28		BI	LI8	SI8	G16	XSHUFFLE	E.16	WSWITCHL
29		BLINKI	LIU8		G32	XSHIFTI	E.32	WSWITCHB
30		BHINTI			G64	XSHIFT	E.64	WMINORL
31	AMINOR	BMINOR	LMINOR	SMINOR	G128		E.128	WMINORB

Minor Operation Codes

For the major operation field values A.MINOR, B.MINOR, L.MINOR, S.MINOR, G.8, G.16, G.32, G.64, G.128, XSHIFTI, XSHIFT, E.8, E.16, E.32, E.64, E.128, W.MINOR.L and W.MINORR.B, the lowest-order six bits in the instruction specify a minor operation code:

31 24	23 6	5 0
major	other	minor
8	18	6

The minor field is filled with a value from one of the following tables:

minor operation code field values for A.MINOR

A.MINOR	0	8	16	24	32	40	48	56
0		AAND	ASETE	ASETEF		ASHLI	ASHLIADD	
1	AADD	AXOR	ASETNE	ASETLGF				
2	AADDO	AOR	ASETANDE	ASETLF		ASHLIO		
3	AADDUO	AANDN	ASETANDNE	ASETGEF		ASHLIUO		
4		AORN	ASETL/LZ	ASETEF.X			ASHLISUB	
5	ASUB	AXNOR	ASETGE/GEZ	ASETLGF.X				
6	ASUBO	ANOR	ASETLU/GZ	ASETLF.X		ASHRI		
7	ASUBUO	ANAND	ASETGEU/LEZ	ASETGEF.X		ASHRIU		ACOM

minor operation code field values for B.MINOR

B.MINOR	0	8	16	24	32	40	48	56
0	B							
1	BLINK							
2	BHINT							
3	BDOWN							
4	BGATE							
5	BBACK							
6	BHALT							
7	BBARRIER							

minor operation code field values for L.MINOR

L.-MINOR	0	8	16	24	32	40	48	56
0	L16L	L64L	LU16L	LU64L				
1	L16B	L64B	LU16B	LU64B				
2	L16AL	L64AL	LU16AL	LU64AL				
3	L16AB	L64AB	LU16AB	LU64AB				
4	L32L	L128L	LU32L	L8				
5	L32B	L128B	LU32B	LU8				
6	L32AL	L128AL	LU32AL					
7	L32AB	L128AB	LU32AB					

minor operation code field values for S.MINOR

S.MINOR	0	8	16	24	32	40	48	56
0	S16L	S64L	SAS64AL					
1	S16B	S64B	SAS64AB					
2	S16AL	S64AL	SCS64AL	SDCS64AL				
3	S16AB	S64AB	SCS64AB	SDCS64AB				
4	S32L	S128L	SMS64AL	S8				
5	S32B	S128B	SMS64AB					
6	S32AL	S128AL	SMUX64AL					
7	S32AB	S128AB	SMUX64AB					

minor operation code field values for G.size

G.size	0	8	16	24	32	40	48	56
0			GSETE	GSETEF	GADDHN	GSUBHN	GSHLIADD	GADDL
1	GADD		GSETNE	GSETLGF	GADDHZ	GSUBHZ		GADDLU
2	GADDO		GSETANDE	GSETLF	GADDHF	GSUBHF		GAAA
3	GADDUO		GSETANDNE	GSETGEF	GADDHC	GSUBHC		
4			GSETL/LZ	GSETEF.X	GADDHUN	GSUBHUN	GSHLISUB	GSUBL
5	GSUB		GSETGE/GEZ	GSETLGF.X	GADDHUZ	GSUBHUZ		GSUBLU
6	GSUBO		GSETLU/GZ	GSETLF.X	GADDHUF	GSUBHUF		GASA
7	GSUBUO		GSETGEU/LEZ	GSETGEF.X	GADDHUC	GSUBHUC		GCOM

minor operation code field values for XSHIFTI

XSHIFTI	0	8	16	24	32	40	48	56
0	XSHLI	XSHLIO		XSHRI		XEXPANDI		XCOMPRESSI
1								
2								
3								
4	XSHLMI	XSHLIOU	XSHRMI	XSHRIU	XROTLI	XEXPANDIU	XROTRI	XCOMPRESSIU
5								
6								
7								

minor operation code field values for XSHIFT

XSHIFT	0	8	16	24	32	40	48	56
0	XSHL	XSHLO		XSHR		XEXPAND		XCOMPRESS
1								
2								
3								

-continued

minor operation code field values for XSHIFT

XSHIFT	0	8	16	24	32	40	48	56
4	XSHLM	XSHLOU	XSHRM	XSHRU	XROTL	XEXPANDU	XROTR	XCOMPRESSU
5								
6								
7								

minor operation code field values for E.size

E.size	0	8	16	24	32	40	48	56
0	EMULFN	EMULADDFN	EADDFN	ESUBFN	EMUL	EMULADD	EDIVFN	ECON
1	EMULFZ	EMULADDFZ	EADDFZ	ESUBFZ	EMULU	EMULADDU	EDIVFZ	ECONU
2	EMULFF	EMULADDFF	EADDFF	ESUBFF	EMULM	EMULADDM	EDIVFF	ECONM
3	EMULFC	EMULADDFC	EADDFC	ESUBFC	EMULC	EMULADDC	EDIVFC	ECONC
4	EMULFX	EMULADDFX	EADDFX	ESUBFX	EMULSUM	EMULSUB	EDIVFX	EDIV
5	EMULF	EMULADDF	EADDF	ESUBF	EMULSUMU	EMULSUBU	EDIVF	EDIVU
6	EMULCF	EMULADDCF	ECONFL	ECONCFL	EMULSUMM	EMULSUBM	EMULSUMF	EMULP
7	EMULSUMCF	EMULSUBCF	ECONFB	ECONCFB	EMULSUMC	EMULSUBC	EMULSUBF	EUNARY

minor operation code field values for W.MINOR.L or W.MINOR.B

W.MINOR.order	0	8	16	24	32	40	48	56
0	WMULMAT8	WMULMATM8						
1	WMULMAT16	WMULMATM16	WMULMATF16					
2	WMULMAT32	WMULMATM32	WMULMATF32					
3	WMULMAT64	WMULMATM64	WMULMATF64					
4	WMULMATU8	WMULMATC8		WMULMATP8				
5	WMULMATU16	WMULMATC16	WMULMATCF16	WMULMATP16				
6	WMULMATU32	WMULMATC32	WMULMATCF32	WMULMATP32				
7	WMULMATU64	WMULMATC64	WMULMATCF64	WMULMATP64				

For the major operation field values E.MUL.X.I, E.MUL.X.I.U, E.MUL.X.I.M, E.MUL.X.I.C, E.MUL.ADD.X.I, E.MUL.ADD.X.I.U, E.MUL.ADD.X.I.M, E.MUL.ADD.X.I.C, E.CON.X.I.L, E.CON.X.I.B, E.CON.X.I.U.L, E.CON.X.I.U.B, E.CON.X.I.M.L, E.CON.X.I.M.B, E.CON.X.I.C.L, E.CON.X.I.C.B, E.EXTRACT.I, E.EXTRACT.I.U, W.MUL.MAT.X.I.U.L, W.MUL.MAT.X.I.U.B, W.MUL.MAT.X.I.M.L, W.MUL.MAT.X.I.M.B, W.MUL.MAT.X.I.C.L, and W.MUL.MAT.X.I.C.B, another six bits in the instruction specify a minor operation code, which indicates operand size, rounding, and shift amount:

31 24	23 6	5 0
major	other	minor
8	18	6

The minor field is filled with a value from the following table: Note that the shift amount field value shown below is the "sh" value, which is encoded in an instruction-dependent manner from the immediate field in the assembler format.

XI	0	8	16	24	32	40	48	56
0	8.F,0	8.N,0	16.F,0	16.N,0	32.F,0	32.N,0	64.F,0	64.N,0
1	8.F,1	8.N,1	16.F,1	16.N,1	32.F,1	32.N,1	64.F,1	64.N,1
2	8.F,2	8.N,2	16.F,2	16.N,2	32.F,2	32.N,2	64.F,2	64.N,2
3	8.F,3	8.N,3	16.F,3	16.N,3	32.F,3	32.N,3	64.F,3	64.N,3
4	8.Z,0	8.C,0	16.Z,0	16.C,0	32.Z,0	32.C,0	64.Z,0	64.C,0
5	8.Z,1	8.C,1	16.Z,1	16.C,1	32.Z,1	32.C,1	64.Z,1	64.C,1
6	8.Z,2	8.C,2	16.Z,2	16.C,2	32.Z,2	32.C,2	64.Z,2	64.C,2
7	8.Z,3	8.C,3	16.Z,3	16.C,3	32.Z,3	32.C,3	64.Z,3	64.C,3

For the major operation field values GCOPYI, two bits in the instruction specify an operand size:

31 24	23 18	17 16	15
op	rd	sz	imm
8	6	2	16

For the major operation field values G.AND.I, G.NAND.I, G.NOR.I, G.OR.I, G.XOR.I, G.ADD.I, G.ADD.I.O, G.ADD.I.UO, G.SET.AND.E.I, G.SET.AND.NE.I, G.SET.E.I, G.SET.GE.I, G.SET.L.I, G.SET.NE.I, G.SET.GE.I.U, G.SET.L.I.U, G.SUB.I, G.SUB.I.O, G.SUB.I.UO, two bits in the instruction specify an operand size:

31 24	23 18	17 12	11 10	9
op	rd	rc	sz	imm
8	6	6	2	10

The sz field is filled with a value from the following table:

sz	size
0	16
1	32
2	64
3	128

For the major operation field values E.8, E.16, E.32, E.64, E.128, with minor operation field value E.UNARY, another six bits in the instruction specify a unary operation code:

31 24	23 18	17 12	11 6	5 0
major	rd	rc	unary	minor
8	6	6	6	6

The unary field is filled with a value from the following table:

	unary operation code field values for E.UNARY.size							
E.UNARY	0	8	16	24	32	40	48	56
0	ESQRFN	ESUMFN	ESINKFN	EFLOATFN	EDEFLATEFN	ESUM		
1	ESQRFZ	ESUMFZ	ESINKFZ	EFLOATFZ	EDEFLATEFZ	ESUMU	ESINKFZD	
2	ESQRFF	ESUMFF	ESINKFF	EFLOATFF	EDEFLATEFF	ELOGMOST	ESINKFFD	
3	ESQRFC	ESUMFC	ESINKFC	EFLOATFC	EDEFLATEFC	ELOGMOSTU	ESINKFCD	
4	ESQRFX	ESUMFX	ESINKFX	EFLOATFX	EDEFLATEFX			
5	ESQRF	ESUMF	ESINKF	EFLOATF	EDEFLATEF			
6	ERSQRESTFX	ERECESTFX	EABSFX	ENEGFX	EINFLATEFX		ECOPYFX	
7	ERSQRESTF	ERECESTF	EABSF	ENEGF	EINFLATEF		ECOPYF	

For the major operation field values A.MINOR and G.MINOR, with minor operation field values A.COM and G.COM, another six bits in the instruction specify a comparison operation code:

31 24	23 18	17 12	11 6	5 0
major	rd	rc	compare	minor
8	6	6	6	6

The compare field is filled with a value from the following table:

	compare operation code field values for A.COM.op and G.COM.op.size							
x.COM	0	8	16	24	32	40	48	56
0	xCOME	xCOMEF						
1	xCOMNE	xCOMLGF						
2	xCOMANDE	xCOMLF						
3	xCOMANDNE	xCOMGEF						
4	xCOML	xCOMEF.X						
5	xCOMGE	xCOMLGF.X						
6	xCOMLU	xCOMLF.X						
7	xCOMGEU	xCOMGEF.X						

General Forms

The general forms of the instructions coded by a major operation code are one of the following:

31 24	23 0
major	offsett
8	24

31 24	23 18	17 0
major	rd	offsett
8	6	18

31 24	23 18	17 12	11 0
major	rd	rc	offsett
8	6	6	12

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

The general forms of the instructions coded by major and minor operation codes are one of the following:

31 24	23 18	17 12	11 6	5 0
major	rd	rc	rb	minor
8	6	6	6	6

31 24	23 18	17 12	11 6	5 0
major	rd	rc	simm	minor
8	6	6	6	6

The general form of the instructions coded by major, minor, and unary operation codes is the following:

31 24	23 18	17 12	11 6	5 0
major	rd	rc	unary	minor
8	6	6	6	6

Register rd is either a source register or destination register, or both. Registers rc and rb are always source registers. Register ra is always a destination register.

Instruction Fetch

Definition

```
def Thread(th) as
    forever do
        catch exception
            if (EventRegister & EventMask[th]) ≠ 0 then
                if ExceptionState=0 then
                    raise EventInterrupt
                endif
            endif
            inst ← LoadMemoryX(ProgramCounter,ProgramCounter,32,L)
            Instruction(inst)
        endcatch
        case exception of
            EventInterrupt,
            ReservedInstruction,
            AccessDisallowedByVirtualAddress,
            AccessDisallowedByTag,
            AccessDisallowedByGlobalTB,
            AccessDisallowedByLocalTB,
            AccessDetailRequiredByTag,
            AccessDetailRequiredByGlobalTB,
            AccessDetailRequiredByLocalTB,
            MissInGlobalTB,
            MissInLocalTB,
            FixedPointArithmetic,
            FloatingPointArithmetic,
            GatewayDisallowed:
                case ExceptionState of
                    0:
                        PerformException(exception)
                    1:
                        PerformException(SecondException)
                    2:
                        PerformMachineCheck(ThirdException)
                endcase
            TakenBranch:
                ContinuationState ← (ExceptionState=0) ? 0 : ContinuationState
            TakenBranchContinue:
                /* nothing */
```

-continued

```
            none, others:
                ProgramCounter ← ProgramCounter + 4
                ContinuationState ← (ExceptionState=0) ? 0 : ContinuationState
        endcase
    endforever
enddef
```

Perform Exception

Definition

```
def PerformException(exception) as
    v ← (exception > 7) ? 7 : exception
    t ← LoadMemory(ExceptionBase,ExceptionBase+Thread*128+64+8*v,64,L)
    if ExceptionState = 0 then
        u ← RegRead(3,128) || RegRead(2,128) || RegRead(1,128) || RegRead(0,128)
        StoreMemory(ExceptionBase,ExceptionBase+Thread*128,512,L,u)
        RegWrite(0,64,ProgramCounter$_{63..2}$ || PrivilegeLevel
        RegWrite(1,64,ExceptionBase+Thread*128)
        RegWrite(2,64,exception)
        RegWrite(3,64,FailingAddress)
    endif
    PrivilegeLevel ← t$_{1..0}$
    ProgramCounter ← t$_{63..2}$ || 0$^2$
    case exception of
        AccessDetailRequiredByTag,
        AccessDetailRequiredByGlobalTB,
        AccessDetailRequiredByLocalTB:
            ContinuationState ← ContinuationState + 1
        others:
        /* nothing */
    endcase
    ExceptionState ← ExceptionState + 1
enddef
```

Instruction Decode

```
def Instruction(inst) as
    major ← inst$_{31..24}$
    rd ← inst$_{23..18}$
    rc ← inst$_{17..12}$
    simm ← rb ← inst$_{11..6}$
    minor ← ra ← inst$_{5..0}$
    case major of
        A.RES:
            AlwaysReserved
        A.MINOR:
            minor ← inst$_{5..0}$
            case minor of
                A.ADD, A.ADD.O, A.ADD.OU, A.AND, A.ANDN, A.NAND, A.NOR,
                A.OR, A.ORN, A.XNOR, A.XOR:
                    Address(minor,rd,rc,rb)
                A.COM:
                    compare ← inst$_{11..6}$
                    case compare of
                        A.COM.E, A.COM.NE, A.COM.AND.E, A.COM.AND.NE,
                        A.COM.L, A.COM.GE, A.COM.L.U, A.COM.GE.U:
                            AddressCompare(compare,rd,rc)
                        others:
                            raise ReservedInstruction
                    endcase
                A.SUB, A.SUB.O, A.SUB.U.O,
                A.SET.AND.E, A.SET.AND.NE, A.SET.E, A.SET.NE,
                A.SET.L, A.SET.GE, A.SET.L.U, A.SET.GE.U,
                    AddressReversed(minor,rd,rc,rb)
```

-continued

```
        A.SHL.I.ADD..A.SHL.I.ADD+3:
            AddressShiftLeftImmediateAdd(inst$_{1..0}$,rd,rc,rb)
        A.SHL.I.SUB..A.SHL.I.SUB+3:
            AddressShiftLeftImmediateSubtract(inst$_{1..0}$,rd,rc,rb)
        A.SHL.I, A.SHL.I.O, A.SHL.I.U.O, A.SHR.I, A.SHR.I.U, A.ROTR.I:
            AddressShiftImmediate(minor,rd,rc,simm)
        others:
            raise ReservedInstruction
    endcase
A.COPY.I
    AddressCopyImmediate(major,rd,inst$_{17..0}$)
A.ADD.I, A.ADD.I.O, A.ADD.I.U.O, A.AND.I, A.OR.I, A.NAND.I, A.NOR.I, A.XOR.I:
    AddressImmediate(major,rd,rc,inst$_{11..0}$)
A.SET.AND.E.I, A.SET.AND.NE.I, A.SET.E.I, A.SET.NE.I,
A.SET.L.I, E.SET.GE.I, A.SET.LU.I, A.SET.GE.U.I,
A.SUB.I, A.SUB.I.O, A.SUB.I.U.O:
    AddressImmediateReversed(major,rd,rc,inst$_{11..0}$)
A.MUX:
    AddressTernary(major,rd,rc,rb,ra)
B.MINOR:
    case minor of
        B:
            Branch(rd,rc,rb)
        B.BACK:
            BranchBack(rd,rc,rb)
        B.BARRIER:
            BranchBarrier(rd,rc,rb)
        B.DOWN:
            BranchDown(rd,rc,rb)
        B.GATE:
            BranchGateway(rd,rc,rb)
        B.HALT:
            BranchHalt(rd,rc,rb)
        B.HINT:
            BranchHint(rd,inst$_{17..12}$,simm)
        B.LINK:
            BranchLink(rd,rc,rb)
        others:
            raise ReservedInstruction
    endcase
BE, BNE, BL, BGE, BLU, BGE.U, BAND.E, BAND.NE:
    BranchConditional(major,rd,rc,inst$_{11..0}$)
BHINTI:
    BranchHintImmediate(inst$_{23..18}$,inst$_{17..12}$,inst$_{11..0}$)
BI:
    BranchImmediate(inst$_{23..0}$)
BLINKI:
    BranchImmediateLink(inst$_{23..0}$)
BEF16, BLGF16, BLF16, BGEF16,
BEF32, BLGF32, BLF32, BGEF32,
BEF64, BLGF64, BLF64, BGEF64,
BEF128, BLGF128, BLF128, BGEF128:
    BranchConditionalFloatingPoint(major,rd,rc,inst$_{11..0}$)
BIF32, BNIF32, BNVF32, BVF32:
    BranchConditionalVisibilityFloatingPoint(major,rd,rc,inst$_{11..0}$)
L.MINOR
    case minor of
        L16L, LU16L, L32L, LU32L, L64L, LU64L, L128L, L8, LU8,
        L16AL, LU16AL, L32AL, LU32AL, L64AL, LU64AL, L128AL,
        L16B, LU16B, L32B, LU32B, L64B, LU64B, L128B,
        L16AB, LU16AB, L32AB, LU32AB, L64AB, LU64AB, L128AB:
            Load(minor,rd,rc,rb)
        others:
            raise ReservedInstruction
    endcase
LI16L LIU16L, LI32L, LIU32L, LI64L, LIU64L, LI128L, LI8, LIU8,
LI16AL, LIU16AL, LI32AL, LIU32AL, LI64AL, LIU64AL, LI128AL,
LI16B, LIU16B, LI32B, LIU32B, LI64B, LIU64B, LI128B,
LI16AB, LIU16AB, LI32AB, LIU32AB, LI64AB, LIU64AB, LI128AB:
    LoadImmediate(major,rd,rc,inst$_{11..0}$)
S.MINOR
    case minor of
        S16L, S32L, S64L, S128L, S8,
        S16AL, S32AL, S64AL, S128AL,
        SAS64AL, SCS64AL, SMS64AL, SM64AL,
        S16B, S32B, S64B, S128B,
        S16AB, S32AB, S64AB, S128AB,
        SAS64AB, SCS64AB, SMS64AB, SM64AB:
            Store(minor,rd,rc,rb)
```

-continued

```
                SDCS64AB, SDCS64AL:
                    StoreDoubleCompareSwap(minor,rd,rc,rb)
                others:
                    raise ReservedInstruction
            endcase
        SI16L, SI32L, SI64L, SI128L, SI8,
        SI16AL, SI32AL, SI64AL, SI128AL,
        SASI64AL, SCSI64AL, SMSI64AL, SMUXI64AL,
        SI16B, SI32B, SI64B, SI128B,
        SI16AB, SI32AB, SI64AB, SI128AB
        SASI64AB, SCSI64AB, SMSI64AB, SMUXI64AB:
            StoreImmediate(major,rd,rc,inst$_{11..0}$)
        G.8, G.16, G.32, G.64, G.128:
            minor $\leftarrow$ inst$_{5..0}$
            size $\leftarrow$ 0 || 1 || 0$^{3+major-G.8}$
            case minor of
                G.ADD, G.ADD.L, G.ADD.LU, G.ADD.O, G.ADD.OU:
                    Group(minor,size,rd,rc,rb)
                G.ADDHC, G.ADDHF, G.ADDHN, G.ADDHZ,
                G.ADDHUC, G.ADDHUF, G.ADDHUN, G.ADDHUZ:
                    GroupAddHalve(minor,inst$_{1..0}$,size,rd,rc,rb)
                G.AAA, G.ASA:
                    GroupInplace(minor,size,rd,rc,rb)
                G.SET.AND.E, G.SET.AND.NE, G.SET.E, G.SET.NE,
                G.SET.L, G.SET.GE, G.SET.L.U, G.SET.GE.U:
                G.SUB, G.SUB.L, G.SUB.LU, G.SUB.O, G.SUB.U.O:
                    GroupReversed(minor,size,ra,rb,rc)
                G.SET.E.F, G.SET.LG.F, G.SET.GE.F, G.SET.L.F,
                G.SET.E.F.X, G.SET.LG.F.X, G.SET.GE.F.X, G.SET.L.F.X:
                    GroupReversedFloatingPoint(minor.op,.size,
                        minor.round, rd, rc, rb)
                G.SHL.I.ADD..G.SHL.I.ADD+3,
                    GroupShiftLeftImmediateAdd(inst$_{1..0}$,size,rd,rc,rb)
                G.SHL.I.SUB..G.SHL.I.SUB+3,
                    GroupShiftLeftImmediateSubtract(inst$_{1..0}$,size,rd,rc,rb)
                G.SUBHC, G.SUBHF, G.SUBHN, G.SUBHZ,
                G.SUBHUC, G.SUBHUF, G.SUBHUN, G.SUBHUZ:
                    GroupSubtractHalve(minor,inst$_{1..0}$,size,rd,rc,rb)
                G.COM,
                    compare $\leftarrow$ inst$_{11..6}$
                    case compare of
                        G.COM.E, G.COM.NE, G.COM.AND.E, G.COM.AND.NE,
                        G.COM.L, G.COM.GE, G.COM.L.U, G.COM.GE.U:
                            GroupCompare(compare,size,ra,rb)
                        others:
                            raise ReservedInstruction
                    endcase
                others:
                    raise ReservedInstruction
            endcase
        G.BOOLEAN..G.BOOLEAN+1:
            GroupBoolean(major,rd,rc,rb,minor)
        G.COPY.I...G.COPY.I+1:
            size $\leftarrow$ 0 || 1 || 0$^{4+inst_{17..16}}$
            GroupCopyImmediate(major,size,rd,inst$_{15..0}$)
        G.AND.I, G.NAND.I, G.NOR.I, G.OR.I, G.XOR.I,
        G.ADD.I, G.ADD.I.O, G.ADD.I.U.O:
            size $\leftarrow$ 0 || 1 || 0$^{4+inst_{11..10}}$
            GroupImmediate(major,size,rd,rc,inst$_{9..0}$)
        G.SET.AND.E.I, G.SET.AND.NE.I, G.SET.E.I, G.SET.GE.I, G.SET.L.I,
        G.SET.NE.I, G.SET.GE.I.U, G.SET.L.I.U, G.SUB.I, G.SUB.I.O, G.SUB.I.U.O:
            size $\leftarrow$ 0 || 1 || 0$^{4+inst_{11..10}}$
            GroupImmediateReversed(major,size,rd,rc,inst$_{9..0}$)
        G.MUX:
            GroupTernary(major,rd,rc,rb,ra)
        X.SHIFT:
            minor $\leftarrow$ inst$_{5..2}$ || 0$^2$
            size $\leftarrow$ 0 || 1 || 0$^{(inst_{24} || inst_{1..0})}$
            case minor of
                X.EXPAND, X.UEXPAND, X.SHL, X.SHL.O, X.SHL.U.O,
                X.ROTR, X.SHR, X.SHR.U,
                    Crossbar(minor,size,rd,rc,rb)
                X.SHL.M, X.SHR.M:
                    CrossbarInplace(minor,size,rd,rc,rb)
                others:
                    raise ReservedInstruction
            endcase
```

-continued

```
X.EXTRACT:
    CrossbarExtract(major,rd,rc,rb,ra)
X.DEPOSIT, X.DEPOSIT.U X.WITHDRAW X.WITHDRAW.U
    CrossbarField(major,rd,rc,inst$_{11..6}$,inst$_{5..0}$)
X.DEPOSIT.M:
    CrossbarFieldInplace(major,rd,rc,inst$_{11..6}$,inst$_{5..0}$)
X.SHIFT.I:
    minor $\leftarrow$ inst$_{5..0}$
    case minor$_{5..2}$ || $0^2$ of
        X.COMPRESS.I, X.EXPAND.I, X.ROTR.I, X.SHL.I, X.SHL.I.O, X.SHL.I.U.O,
        X.SHR.I, X.COMPRESS.I.U, X.EXPAND.I.U, X.SHR.UI:
            CrossbarShortImmediate(minor,rd,rc,simm)
        X.SHL.M.I, X.SHR.M.I:
            CrossbarShortImmediateInplace(minor,rd,rc,simm)
        others:
            raise ReservedInstruction
    endcase
X.SHUFFLE..X.SHUFFLE+1:
    CrossbarShuffle(major,rd,rc,rb,simm)
X.SWIZZLE..X.SWIZZLE+3:
    CrossbarSwizzle(major,rd,rc, inst$_{11..6}$,inst$_{5..0}$)
X.SELECT.8:
    CrossbarTernary(major,rd,rc,rb,ra)
E.8, E.16, E.32, E.64, E.128:
    minor $\leftarrow$ inst$_{5..0}$
    size $\leftarrow$ 0 || 1 || $0^{3+major-E.8}$
    case minor of
        E.CON., E.CON.U, E.CON.M, E.CON.C,
        E.MUL., E.MUL.U, E.MUL.M, E.MUL.C,
        E.MUL.SUM, E.MUL.SUM.U, E.MUL.SUM.M, E.MUL.SUM.C,
        E.DIV, E.DIV.U, E.MUL.P:
            Ensemble(minor,size,ra,rb,rc)
        E.CON.F.L, E.CON.F.B, E.CON.C.F.L, E.CON.C.F.B:
            EnsembleConvolveFloatingPoint(minor.size,rd,rc,rb)
        E.ADD.F.N, E.MUL.C.F.N, E.MUL.F.N, E.DIV.F.N,
        E.ADD.F.Z, E.MUL.C.F.Z, E.MUL.F.Z, E.DIV.F.Z,
        E.ADD.F.F, E.MUL.C.F.F, E.MUL.F.F, E.DIV.F.F,
        E.ADD.F.C, E.MUL.C.F.C, E.MUL.F.C, E.DIV.F.C,
        E.ADD.F, E.MUL.C.F, E.MUL.F, E.DIV.F,
        E.ADD.F.X, E.MUL.C.F.X, E.MUL.F.X, E.DIV.F.X,
            EnsembleFloatingPoint(minor.op, major.size, minor.round, rd, rc, rb)
        E.MUL.ADD, E.MUL.ADD.U, E.MUL.ADD.M, E.MUL.ADD.C:
            EnsembleInplace(minor,size,rd,rc,rb)
        E.MUL.SUB, E.MUL.SUB.U, E.MUL.SUB.M, E.MUL.SUB.C:
            EnsembleInplaceReversed(minor,size,rd,rc,rb)
        E.MUL.SUB.F, E.MUL.SUB.C.F:
            EnsembleInplaceReversedFloatingPoint(minor,size,rd,rc,rb)
        E.SUB.F.N, E.SUB.F.Z, E.SUB.F.F, E.SUB.F.C, E.SUB.F, E.SUB.F.X:
            EnsembleReversedFloatingPoint(minor.op, major.size,
                minor.round, rd, rc, rb)
        E.UNARY:
            case unary of
                E.SUM, E.SUMU, E.LOG.MOST, E. LOG.MOST.U:
                    EnsembleUnary(unary,rd,rc)
                E.ABS.F, E.ABS.F.X, E.COPY.F, E.COPY.F.X,
                E.DEFLATE.F, E.DEFLATE.F.N, E.DEFLATE.F.Z,
                E.DEFLATE.F.F, E.DEFLATE.F.C, E.DEFLATE.F.X:
                E.FLOAT.F, E.FLOAT.F.N, E.FLOAT.F.Z,
                E.FLOAT.F.F, E.FLOAT.F.C, E.FLOAT.F.X:
                E.INFLATE.F, E.INFLATE.F.X, E.NEG.F, E.NEG.F.X,
                E.RECEST.F, E.RECEST.F.X, E.RSQREST.F, E.RSQREST.F.X,
                E.SQR.F, E.SQR.F.N, E.SQR.F.Z, E.SQR.F.F, E.SQR.F.C, E.SQR.F.X:
                E.SUM.F, E.SUM.F.N, E.SUM.F.Z,
```