Hexagon V5x Instruction Quick Reference

Hexagon™ processor registers

Name	Description
R0-R31	General registers
Rx:Ry	64-bit register pair
Rx.H-Rx.L	Most and least significant halfwords
SP	Stack pointer (R29)
FP	Frame pointer (R30)
LR	Link register (R31)
P0-P3	Predicate registers
LC0-LC1	Loop count
SA0-SA1	Loop start address
M0-M1	Modify registers
CS0-CS1	Circular start registers
USR	User status register
UGP	User general pointer
GP	Global pointer
PC	Program counter
UPCYCLELO	Cycle count register (low)
UPCYCLEHI	Cycle count register (high)

Register operands

Symbol	Description
Rs, Rt, Ru	32-bit source
Rd	32-bit destination
Rx	32-bit source/destination
Rss, Rtt, Ruu	64-bit source
Rdd	64-bit destination
Rxx	64-bit source/destination

Constant operands

Symbol	Description
#u*N*	*N*-bit unsigned value
#s*N*	*N*-bit signed value
#m*N*	Same as #sN, but minimum value is $-(2^{(N-1)}-1)$
#r*N*	*N*-bit PC-relative value
#x*N*:*M*	Bit field *N*:*M* in *N*-bit value
##	Same as #, but associated value is 32 bits

Instruction packets

Slot 0	Slot 1	Slot 2	Slot 3
LD ST ALU32 MEMOP NV SYSTEM	LD ST ALU32	XTYPE ALU32 J JR	XTYPE ALU32 J CR

ALU32

ALU32/ALU

Add
```
Rd=add(Rs,#s16)
Rd=add(Rs,Rt)
Rd=add(Rs,Rt):sat
```
Logical operations
```
Rd=and(Rs,#s10)
Rd=and(Rs,Rt)
Rd=and(Rt,~Rs)
Rd=not(Rs)
Rd=or(Rs,#s10)
Rd=or(Rs,Rt)
Rd=or(Rt,~Rs)
Rd=xor(Rs,Rt)
```
Negate
```
Rd=neg(Rs)
```
Nop
```
nop
```
Subtract
```
Rd=sub(#s10,Rs)
Rd=sub(Rt,Rs)
Rd=sub(Rt,Rs):sat
```
Sign extend
```
Rd=sxtb(Rs)
Rd=sxth(Rs)Transfer immediate
Rd=#s16
Rdd=#s8
Rx.[HL]=#u16
```
Transfer register
```
Rd=Rs
Rdd=Rss
```
Vector add halfwords
```
Rd=vaddh(Rs,Rt)[:sat]
Rd=vadduh(Rs,Rt):sat
```
Vector average halfwords
```
Rd=vavgh(Rs,Rt)
Rd=vavgh(Rs,Rt):rnd
Rd=vnavgh(Rt,Rs)
```
Vector subtract halfwords
```
Rd=vsubh(Rt,Rs)[:sat]
Rd=vsubuh(Rt,Rs):sat
```
Zero extend
```
Rd=zxtb(Rs)
Rd=zxth(Rs)
```
ALU32/PERM

Combine words into doubleword
```
Rd=combine(Rt.[HL],Rs.[HL])
Rdd=combine(#s8,#S8)
Rdd=combine(#s8,#U6)
Rdd=combine(#s8,Rs)
Rdd=combine(Rs,#s8)
Rdd=combine(Rs,Rt)
```
Mux
```
Rd=mux(Pu,#s8,#S8)
Rd=mux(Pu,#s8,Rs)
Rd=mux(Pu,Rs,#s8)
Rd=mux(Pu,Rs,Rt)
```
Shift word by 16
```
Rd=aslh(Rs)
Rd=asrh(Rs)
```
Pack high and low halfwords
```
Rdd=packhl(Rs,Rt)
```
ALU32/PRED

Conditional add
```
if ([!]Pu[.new]) Rd=add(Rs,#s8)
if ([!]Pu[.new]) Rd=add(Rs,Rt)
```
Conditional shift halfword
```
if ([!]Pu[.new]) Rd=aslh(Rs)
if ([!]Pu[.new]) Rd=asrh(Rs)
```
Conditional combine
```
if ([!]Pu[.new]) Rdd=combine(Rs,Rt)
```
Conditional logical operations
```
if ([!]Pu[.new]) Rd=and(Rs,Rt)
if ([!]Pu[.new]) Rd=or(Rs,Rt)
if ([!]Pu[.new]) Rd=xor(Rs,Rt)
```
Conditional subtract
```
if ([!]Pu[.new]) Rd=sub(Rt,Rs)
```
Conditional sign extend
```
if ([!]Pu[.new]) Rd=sxtb(Rs)
if ([!]Pu[.new]) Rd=sxth(Rs)
```
Conditional transfer
```
if ([!]Pu[.new]) Rd=#s12
if ([!]Pu[.new]) Rd=Rs
if ([!]Pu[.new]) Rdd=Rss
```
Conditional zero extend
```
if ([!]Pu[.new]) Rd=zxtb(Rs)
if ([!]Pu[.new]) Rd=zxth(Rs)
```
Compare
```
Pd=[!]cmp.eq(Rs,#s10)
Pd=[!]cmp.eq(Rs,Rt)
Pd=[!]cmp.gt(Rs,#s10)
Pd=[!]cmp.gt(Rs,Rt)
Pd=[!]cmp.gtu(Rs,#u9)
Pd=[!]cmp.gtu(Rs,Rt)
Pd=cmp.ge(Rs,#s8)
Pd=cmp.geu(Rs,#u8)
Pd=cmp.lt(Rs,Rt)
Pd=cmp.ltu(Rs,Rt)
```
Compare to general register
```
Rd=[!]cmp.eq(Rs,#s8)
Rd=[!]cmp.eq(Rs,Rt)
```
CR

End loop instructions
```
endloop0
endloop01
endloop1
```
Corner detection acceleration
```
Pd=[!]fastcorner9(Ps,Pt)
```
Logical reductions on predicates
```
Pd=all8(Ps)
Pd=any8(Ps)
```
Looping instructions
```
loop0(#r7:2,#U10)
loop0(#r7:2,Rs)
loop1(#r7:2,#U10)
loop1(#r7:2,Rs)
```
Add to PC
```
Rd=add(pc,#u6)
```
Pipelined loop instructions
```
p3=sp1loop0(#r7:2,#U10)
p3=sp1loop0(#r7:2,Rs)
p3=sp2loop0(#r7:2,#U10)
p3=sp2loop0(#r7:2,Rs)
p3=sp3loop0(#r7:2,#U10)
p3=sp3loop0(#r7:2,Rs)
```


Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Logical operations on predicates

```
Pd=Ps
Pd=and(Ps,and(Pt,[!]Pu))
Pd=and(Ps,or(Pt,[!]Pu))
Pd=and(Pt,[!]Ps)
Pd=not(Ps)
Pd=or(Ps,and(Pt,[!]Pu))
Pd=or(Ps,or(Pt,[!]Pu))
Pd=or(Pt,[!]Ps)
Pd=xor(Ps,Pt)
```

User control register transfer

```
Cd=Rs
Cdd=Rss
Rd=Cs
Rdd=Css
```

JR

Call subroutine from register

```
callr Rs
if ([!]Pu) callr Rs
```

Hint an indirect jump address

```
hintjr(Rs)
```

Jump to address from register

```
if ([!]Pu) jumpr Rs
if ([!]Pu.new) jumpr:<hint> Rs
jumpr Rs
```

J

Call subroutine

```
call #r22:2
if ([!]Pu) call #r15:2
```

Compare and jump

```
p[01]=cmp.eq(Rs,#-1); if ([!]p[01].new)
    jump:<hint> #r9:2
p[01]=cmp.eq(Rs,#U5); if ([!]p[01].new)
    jump:<hint> #r9:2
p[01]=cmp.eq(Rs,Rt); if ([!]p[01].new)
    jump:<hint> #r9:2
p[01]=cmp.gt(Rs,#-1); if ([!]p[01].new)
    jump:<hint> #r9:2
p[01]=cmp.gt(Rs,#U5); if ([!]p[01].new)
    jump:<hint> #r9:2
p[01]=cmp.gt(Rs,Rt); if ([!]p[01].new)
    jump:<hint> #r9:2
p[01]=cmp.gtu(Rs,#U5); if ([!]p[01].new)
    jump:<hint> #r9:2
p[01]=cmp.gtu(Rs,Rt); if ([!]p[01].new)
    jump:<hint> #r9:2
p[01]=tstbit(Rs,#0); if ([!]p[01].new)
    jump:<hint> #r9:2
```

Jump to address

```
if ([!]Pu) jump #r15:2
jump #r22:2
```

Jump to address conditioned on new predicate

```
if ([!]Pu.new) jump:<hint> #r15:2
```

Jump to address condition on register value

```
if (Rs!=#0) jump:nt #r13:2
if (Rs!=#0) jump:t #r13:2
if (Rs<=#0) jump:nt #r13:2
if (Rs<=#0) jump:t #r13:2
if (Rs==#0) jump:nt #r13:2
if (Rs==#0) jump:t #r13:2
if (Rs>=#0) jump:nt #r13:2
if (Rs>=#0) jump:t #r13:2
```

Transfer and jump

```
Rd=#U6 ; jump #r9:2
Rd=Rs ; jump #r9:2
```

LD

Load doubleword

```
Rdd=memd(Re=#U6)
Rdd=memd(Rs+#s11:3)
Rdd=memd(Rs+Rt<<#u2)
Rdd=memd(Rt<<#u2+#U6)
Rdd=memd(Rx++#s4:3)
Rdd=memd(Rx++#s4:3:circ(Mu))
Rdd=memd(Rx++I:circ(Mu))
Rdd=memd(Rx++Mu)
Rdd=memd(Rx++Mu:brev)
Rdd=memd(gp+#u16:3)
```

Load doubleword conditionally

```
if ([!]Pt[.new]) Rdd=memd(#u6)
if ([!]Pt[.new]) Rdd=memd(Rs+#u6:3)
if ([!]Pt[.new]) Rdd=memd(Rx++#s4:3)
if ([!]Pv[.new]) Rdd=memd(Rs+Rt<<#u2)
```

Load byte

```
Rd=memb(Re=#U6)
Rd=memb(Rs+#s11:0)
Rd=memb(Rs+Rt<<#u2)
Rd=memb(Rt<<#u2+#U6)
Rd=memb(Rx++#s4:0)
Rd=memb(Rx++#s4:0:circ(Mu))
Rd=memb(Rx++I:circ(Mu))
Rd=memb(Rx++Mu)
Rd=memb(Rx++Mu:brev)
Rd=memb(gp+#u16:0)
```

Load byte conditionally

```
if ([!]Pt[.new]) Rd=memb(#u6)
if ([!]Pt[.new]) Rd=memb(Rs+#u6:0)
if ([!]Pt[.new]) Rd=memb(Rx++#s4:0)
if ([!]Pv[.new]) Rd=memb(Rs+Rt<<#u2)
```

Load byte into shifted vector

```
Ryy=memb_fifo(Re=#U6)
Ryy=memb_fifo(Rs)
Ryy=memb_fifo(Rs+#s11:0)
Ryy=memb_fifo(Rt<<#u2+#U6)
Ryy=memb_fifo(Rx++#s4:0)
Ryy=memb_fifo(Rx++#s4:0:circ(Mu))
Ryy=memb_fifo(Rx++I:circ(Mu))
Ryy=memb_fifo(Rx++Mu)
Ryy=memb_fifo(Rx++Mu:brev)
```

Load half into shifted vector

```
Ryy=memh_fifo(Re=#U6)
Ryy=memh_fifo(Rs)
Ryy=memh_fifo(Rs+#s11:1)
Ryy=memh_fifo(Rt<<#u2+#U6)
Ryy=memh_fifo(Rx++#s4:1)
Ryy=memh_fifo(Rx++#s4:1:circ(Mu))
Ryy=memh_fifo(Rx++I:circ(Mu))
Ryy=memh_fifo(Rx++Mu)
Ryy=memh_fifo(Rx++Mu:brev)
```

Load halfword

```
Rd=memh(Re=#U6)
Rd=memh(Rs+#s11:1)
Rd=memh(Rs+Rt<<#u2)
Rd=memh(Rt<<#u2+#U6)
Rd=memh(Rx++#s4:1)
Rd=memh(Rx++#s4:1:circ(Mu))
Rd=memh(Rx++I:circ(Mu))
Rd=memh(Rx++Mu)
Rd=memh(Rx++Mu:brev)
Rd=memh(gp+#u16:1)
```

Load halfword conditionally

```
if ([!]Pt[.new]) Rd=memh(#u6)
if ([!]Pt[.new]) Rd=memh(Rs+#u6:1)
if ([!]Pt[.new]) Rd=memh(Rx++#s4:1)
if ([!]Pv[.new]) Rd=memh(Rs+Rt<<#u2)
```

Load unsigned byte

```
Rd=memub(Re=#U6)
Rd=memub(Rs+#s11:0)
Rd=memub(Rs+Rt<<#u2)
Rd=memub(Rt<<#u2+#U6)
Rd=memub(Rx++#s4:0)
Rd=memub(Rx++#s4:0:circ(Mu))
Rd=memub(Rx++I:circ(Mu))
Rd=memub(Rx++Mu)
Rd=memub(Rx++Mu:brev)
Rd=memub(gp+#u16:0)
```

Load unsigned byte conditionally

```
if ([!]Pt[.new]) Rd=memub(#u6)
if ([!]Pt[.new]) Rd=memub(Rs+#u6:0)
if ([!]Pt[.new]) Rd=memub(Rx++#s4:0)
if ([!]Pv[.new]) Rd=memub(Rs+Rt<<#u2)
```

Load unsigned halfword

```
Rd=memuh(Re=#U6)
Rd=memuh(Rs+#s11:1)
Rd=memuh(Rs+Rt<<#u2)
Rd=memuh(Rt<<#u2+#U6)
Rd=memuh(Rx++#s4:1)
Rd=memuh(Rx++#s4:1:circ(Mu))
Rd=memuh(Rx++I:circ(Mu))
Rd=memuh(Rx++Mu)
Rd=memuh(Rx++Mu:brev)
Rd=memuh(gp+#u16:1)
```

Load unsigned halfword conditionally

```
if ([!]Pt[.new]) Rd=memuh(#u6)
if ([!]Pt[.new]) Rd=memuh(Rs+#u6:1)
if ([!]Pt[.new]) Rd=memuh(Rx++#s4:1)
if ([!]Pv[.new]) Rd=memuh(Rs+Rt<<#u2)
```

Load word

```
Rd=memw(Re=#U6)
Rd=memw(Rs+#s11:2)
Rd=memw(Rs+Rt<<#u2)
Rd=memw(Rt<<#u2+#U6)
Rd=memw(Rx++#s4:2)
Rd=memw(Rx++#s4:2:circ(Mu))
Rd=memw(Rx++I:circ(Mu))
Rd=memw(Rx++Mu)
Rd=memw(Rx++Mu:brev)
Rd=memw(gp+#u16:2)
```

Load word conditionally

```
if ([!]Pt[.new]) Rd=memw(#u6)
if ([!]Pt[.new]) Rd=memw(Rs+#u6:2)
if ([!]Pt[.new]) Rd=memw(Rx++#s4:2)
if ([!]Pv[.new]) Rd=memw(Rs+Rt<<#u2)
```

Deallocate stack frame

```
deallocframe
```

Deallocate frame and return

```
dealloc_return
if ([!]Ps) dealloc_return
if ([!]Ps.new) dealloc_return:nt
if ([!]Ps.new) dealloc_return:t
```

Load and unpack bytes to halfwords

```
Rd=membh(Re=#U6)
Rd=membh(Rs)
Rd=membh(Rs+#s11:1)
Rd=membh(Rt<<#u2+#U6)
Rd=membh(Rx++#s4:1)
Rd=membh(Rx++#s4:1:circ(Mu))
Rd=membh(Rx++I:circ(Mu))
Rd=membh(Rx++Mu)
Rd=membh(Rx++Mu:brev)
Rd=memubh(Re=#U6)
Rd=memubh(Rs+#s11:1)
Rd=memubh(Rt<<#u2+#U6)
Rd=memubh(Rx++#s4:1)
Rd=memubh(Rx++#s4:1:circ(Mu))
Rd=memubh(Rx++I:circ(Mu))
Rd=memubh(Rx++Mu)
Rd=memubh(Rx++Mu:brev)
```

MEMOP

Operation on memory byte

```
memb(Rs+#u6:0)=clrbit(#U5)
memb(Rs+#u6:0)=setbit(#U5)
memb(Rs+#u6:0)[+-]=#U5
memb(Rs+#u6:0)[+-|&]=Rt
```

Operation on memory halfword

```
memh(Rs+#u6:1)=clrbit(#U5)
memh(Rs+#u6:1)=setbit(#U5)
memh(Rs+#u6:1)[+-]=#U5
memh(Rs+#u6:1)[+-|&]=Rt
```

Operation on memory word

```
memw(Rs+#u6:2)=clrbit(#U5)
memw(Rs+#u6:2)=setbit(#U5)
memw(Rs+#u6:2)[+-]=#U5
memw(Rs+#u6:2)[+-|&]=Rt
```

NV

NV/J

Jump to address condition on new register value

```
if ([!]cmp.eq(Ns.new,#-1)) jump:<hint>
    #r9:2
if ([!]cmp.eq(Ns.new,#U5)) jump:<hint>
    #r9:2
if ([!]cmp.eq(Ns.new,Rt)) jump:<hint>
    #r9:2
if ([!]cmp.gt(Ns.new,#-1)) jump:<hint>
    #r9:2
if ([!]cmp.gt(Ns.new,#U5)) jump:<hint>
    #r9:2
if ([!]cmp.gt(Ns.new,Rt)) jump:<hint>
    #r9:2
if ([!]cmp.gt(Rt,Ns.new)) jump:<hint>
    #r9:2
if ([!]cmp.gtu(Ns.new,#U5)) jump:<hint>
    #r9:2
if ([!]cmp.gtu(Ns.new,Rt)) jump:<hint>
    #r9:2
if ([!]cmp.gtu(Rt,Ns.new)) jump:<hint>
    #r9:2
if ([!]tstbit(Ns.new,#0)) jump:<hint>
    #r9:2
```

NV/ST

Store new-value byte

```
memb(Re=#U6)=Nt.new
memb(Rs+#s11:0)=Nt.new
memb(Rs+Ru<<#u2)=Nt.new
memb(Ru<<#u2+#U6)=Nt.new
memb(Rx++#s4:0)=Nt.new
memb(Rx++#s4:0:circ(Mu))=Nt.new
memb(Rx++I:circ(Mu))=Nt.new
memb(Rx++Mu)=Nt.new
memb(Rx++Mu:brev)=Nt.new
memb(gp+#u16:0)=Nt.new
```

Store new-value byte conditionally

```
if ([!]Pv[.new]) memb(#u6)=Nt.new
if ([!]Pv[.new]) memb(Rs+#u6:0)=Nt.new
if ([!]Pv[.new])
    memb(Rs+Ru<<#u2)=Nt.new
if ([!]Pv[.new])
    memb(Rx++#s4:0)=Nt.new
```

Store new-value halfword

```
memh(Re=#U6)=Nt.new
memh(Rs+#s11:1)=Nt.new
memh(Rs+Ru<<#u2)=Nt.new
memh(Ru<<#u2+#U6)=Nt.new
memh(Rx++#s4:1)=Nt.new
memh(Rx++#s4:1:circ(Mu))=Nt.new
memh(Rx++I:circ(Mu))=Nt.new
memh(Rx++Mu)=Nt.new
memh(Rx++Mu:brev)=Nt.new
memh(gp+#u16:1)=Nt.new
```

Store new-value halfword conditionally

```
if ([!]Pv[.new]) memh(#u6)=Nt.new
if ([!]Pv[.new]) memh(Rs+#u6:1)=Nt.new
if ([!]Pv[.new])
    memh(Rs+Ru<<#u2)=Nt.new
if ([!]Pv[.new])
    memh(Rx++#s4:1)=Nt.new
```

Store new-value word

```
memw(Re=#U6)=Nt.new
memw(Rs+#s11:2)=Nt.new
memw(Rs+Ru<<#u2)=Nt.new
memw(Ru<<#u2+#U6)=Nt.new
memw(Rx++#s4:2)=Nt.new
memw(Rx++#s4:2:circ(Mu))=Nt.new
memw(Rx++I:circ(Mu))=Nt.new
memw(Rx++Mu)=Nt.new
memw(Rx++Mu:brev)=Nt.new
memw(gp+#u16:2)=Nt.new
```

Store new-value word conditionally

```
if ([!]Pv[.new]) memw(#u6)=Nt.new
if ([!]Pv[.new]) memw(Rs+#u6:2)=Nt.new
if ([!]Pv[.new])
    memw(Rs+Ru<<#u2)=Nt.new
if ([!]Pv[.new])
    memw(Rx++#s4:2)=Nt.new
```

ST

Store doubleword

```
memd(Re=#U6)=Rtt
memd(Rs+#s11:3)=Rtt
memd(Rs+Ru<<#u2)=Rtt
memd(Ru<<#u2+#U6)=Rtt
memd(Rx++#s4:3)=Rtt
memd(Rx++#s4:3:circ(Mu))=Rtt
memd(Rx++I:circ(Mu))=Rtt
memd(Rx++Mu)=Rtt
memd(Rx++Mu:brev)=Rtt
memd(gp+#u16:3)=Rtt
```

Store doubleword conditionally

```
if ([!]Pv[.new]) memd(#u6)=Rtt
if ([!]Pv[.new]) memd(Rs+#u6:3)=Rtt
if ([!]Pv[.new]) memd(Rs+Ru<<#u2)=Rtt
if ([!]Pv[.new]) memd(Rx++#s4:3)=Rtt
```

Store byte

```
memb(Re=#U6)=Rt
memb(Rs+#s11:0)=Rt
memb(Rs+#u6:0)=#S8
memb(Rs+Ru<<#u2)=Rt
memb(Ru<<#u2+#U6)=Rt
memb(Rx++#s4:0)=Rt
memb(Rx++#s4:0:circ(Mu))=Rt
memb(Rx++I:circ(Mu))=Rt
memb(Rx++Mu)=Rt
memb(Rx++Mu:brev)=Rt
memb(gp+#u16:0)=Rt
```

Store byte conditionally
if ([!]Pv[.new]) memb(#u6)=Rt
if ([!]Pv[.new]) memb(Rs+#u6:0)=#S6
if ([!]Pv[.new]) memb(Rs+#u6:0)=Rt
if ([!]Pv[.new]) memb(Rs+Ru<<#u2)=Rt
if ([!]Pv[.new]) memb(Rx++#s4:0)=Rt

Store halfword
memh(Re=#U6)=Rt.H
memh(Re=#U6)=Rt
memh(Rs+#s11:1)=Rt.H
memh(Rs+#s11:1)=Rt
memh(Rs+#u6:1)=#S8
memh(Rs+Ru<<#u2)=Rt.H
memh(Rs+Ru<<#u2)=Rt
memh(Ru<<#u2+#U6)=Rt.H
memh(Ru<<#u2+#U6)=Rt
memh(Rx++#s4:1)=Rt.H
memh(Rx++#s4:1)=Rt
memh(Rx++#s4:1:circ(Mu))=Rt.H
memh(Rx++#s4:1:circ(Mu))=Rt
memh(Rx++I:circ(Mu))=Rt.H
memh(Rx++I:circ(Mu))=Rt
memh(Rx++Mu)=Rt.H
memh(Rx++Mu)=Rt
memh(Rx++Mu:brev)=Rt.H
memh(Rx++Mu:brev)=Rt
memh(gp+#u16:1)=Rt.H
memh(gp+#u16:1)=Rt

Store halfword conditionally.
if ([!]Pv[.new]) memh(#u6)=Rt.H
if ([!]Pv[.new]) memh(#u6)=Rt
if ([!]Pv[.new]) memh(Rs+#u6:1)=#S6
if ([!]Pv[.new]) memh(Rs+#u6:1)=Rt.H
if ([!]Pv[.new]) memh(Rs+#u6:1)=Rt
if ([!]Pv[.new])
memh(Rs+Ru<<#u2)=Rt.H
if ([!]Pv[.new]) memh(Rs+Ru<<#u2)=Rt
if ([!]Pv[.new]) memh(Rx++#s4:1)=Rt.H
if ([!]Pv[.new]) memh(Rx++#s4:1)=Rt

Store word
memw(Re=#U6)=Rt
memw(Rs+#s11:2)=Rt
memw(Rs+#u6:2)=#S8
memw(Rs+Ru<<#u2)=Rt
memw(Ru<<#u2+#U6)=Rt
memw(Rx++#s4:2)=Rt
memw(Rx++#s4:2:circ(Mu))=Rt
memw(Rx++I:circ(Mu))=Rt
memw(Rx++Mu)=Rt
memw(Rx++Mu:brev)=Rt
memw(gp+#u16:2)=Rt

Store word conditionally
if ([!]Pv[.new]) memw(#u6)=Rt
if ([!]Pv[.new]) memw(Rs+#u6:2)=#S6
if ([!]Pv[.new]) memw(Rs+#u6:2)=Rt
if ([!]Pv[.new]) memw(Rs+Ru<<#u2)=Rt
if ([!]Pv[.new]) memw(Rx++#s4:2)=Rt

Allocate stack frame
allocframe(#u11:3)

SYSTEM

SYSTEM/USER

Load locked
Rd=memw_locked(Rs)
Rdd=memd_locked(Rs)

Store conditional
memd_locked(Rs,Pd)=Rtt
memw_locked(Rs,Pd)=Rt

Zero a cache line
dczeroa(Rs)

Memory barrier
barrier

Breakpoint
brkpt

Data cache prefetch
dcfetch(Rs)
dcfetch(Rs+#u11:3)

Data cache maintenance user operations
dccleana(Rs)
dccleaninva(Rs)
dcinva(Rs)

Instruction cache maintenance user operations
icinva(Rs)

Instruction synchronization
isync

L2 cache prefetch
l2fetch(Rs,Rt)
l2fetch(Rs,Rtt)

Pause
pause(#u8)

Memory thread synchronization
syncht

Send value to ETM trace
trace(Rs)

Trap
trap0(#u8)
trap1(#u8)

XTYPE

XTYPE/ALU

Absolute value doubleword
Rdd=abs(Rss)

Absolute value word
Rd=abs(Rs)[:sat]

Add and accumulate
Rd=add(Rs,add(Ru,#s6))
Rd=add(Rs,sub(#s6,Ru))
Rx+=add(Rs,#s8)
Rx+=add(Rs,Rt)
Rx-=add(Rs,#s8)
Rx-=add(Rs,Rt)

Add doublewords
Rd=add(Rs,Rt):sat:deprecated
Rdd=add(Rs,Rtt)
Rdd=add(Rss,Rtt)
Rdd=add(Rss,Rtt):raw:hi
Rdd=add(Rss,Rtt):raw:lo
Rdd=add(Rss,Rtt):sat

Add halfword
Rd=add(Rt.L,Rs.[HL])[:sat]
Rd=add(Rt.[HL],Rs.[HL])[:sat]:<<16

Add or subtract doublewords with carry
Rdd=add(Rss,Rtt,Px):carry
Rdd=sub(Rss,Rtt,Px):carry

Logical doublewords
Rdd=and(Rss,Rtt)
Rdd=and(Rtt,~Rss)
Rdd=not(Rss)
Rdd=or(Rss,Rtt)
Rdd=or(Rtt,~Rss)
Rdd=xor(Rss,Rtt)

Logical-logical doublewords
Rxx^=xor(Rss,Rtt)

Logical-logical words
Rx=or(Ru,and(Rx,#s10))
Rx[&|^]=and(Rs,Rt)
Rx[&|^]=and(Rs,~Rt)
Rx[&|^]=or(Rs,Rt)
Rx[&|^]=xor(Rs,Rt)
Rx|=and(Rs,#s10)
Rx|=or(Rs,#s10)

Maximum words
Rd=max(Rs,Rt)
Rd=maxu(Rs,Rt)

Maximum doublewords
Rdd=max(Rss,Rtt)
Rdd=maxu(Rss,Rtt)

Minimum words
Rd=min(Rt,Rs)
Rd=minu(Rt,Rs)

Minimum doublewords
Rdd=min(Rtt,Rss)
Rdd=minu(Rtt,Rss)

Modulo wrap
Rd=modwrap(Rs,Rt)

Negate
Rd=neg(Rs):sat
Rdd=neg(Rss)

Round
Rd=cround(Rs,#u5)
Rd=cround(Rs,Rt)
Rd=round(Rs,#u5)[:sat]
Rd=round(Rs,Rt)[:sat]
Rd=round(Rss):sat

Subtract doublewords
Rd=sub(Rt,Rs):sat:deprecated
Rdd=sub(Rtt,Rss)

Subtract and accumulate words
Rx+=sub(Rt,Rs)

Subtract halfword
Rd=sub(Rt.L,Rs.[HL])[:sat]
Rd=sub(Rt.[HL],Rs.[HL])[:sat]:<<16

Sign extend word to doubleword
Rdd=sxtw(Rs)

Vector absolute value halfwords
Rdd=vabsh(Rss)
Rdd=vabsh(Rss):sat

Vector absolute value words
Rdd=vabsw(Rss)
Rdd=vabsw(Rss):sat

Vector absolute difference halfwords
Rdd=vabsdiffh(Rtt,Rss)

Vector absolute difference words
Rdd=vabsdiffw(Rtt,Rss)

Vector add halfwords
Rdd=vaddh(Rss,Rtt)[:sat]
Rdd=vadduh(Rss,Rtt):sat

Vector add halfwords with saturate and pack to unsigned bytes
Rd=vaddhub(Rss,Rtt):sat

Vector reduce add unsigned bytes
Rdd=vraddub(Rss,Rtt)
Rxx+=vraddub(Rss,Rtt)

Vector reduce add halfwords
Rd=vraddh(Rss,Rtt)
Rd=vradduh(Rss,Rtt)

Vector add bytes
Rdd=vaddb(Rss,Rtt)
Rdd=vaddub(Rss,Rtt)[:sat]

Vector add words
Rdd=vaddw(Rss,Rtt)[:sat]

Vector average halfwords
Rdd=vavgh(Rss,Rtt)
Rdd=vavgh(Rss,Rtt):crnd
Rdd=vavgh(Rss,Rtt):rnd
Rdd=vavguh(Rss,Rtt)
Rdd=vavguh(Rss,Rtt):rnd
Rdd=vnavgh(Rtt,Rss)
Rdd=vnavgh(Rtt,Rss):crnd:sat
Rdd=vnavgh(Rtt,Rss):rnd:sat

Vector average unsigned bytes
Rdd=vavgub(Rss,Rtt)
Rdd=vavgub(Rss,Rtt):rnd

Vector average words
Rdd=vavguw(Rss,Rtt)[:rnd]
Rdd=vavgw(Rss,Rtt):crnd
Rdd=vavgw(Rss,Rtt)[:rnd]
Rdd=vnavgw(Rtt,Rss)
Rdd=vnavgw(Rtt,Rss):crnd:sat
Rdd=vnavgw(Rtt,Rss):rnd:sat

Vector conditional negate
Rdd=vcnegh(Rss,Rt)
Rxx+=vrcnegh(Rss,Rt)

Vector maximum bytes
Rdd=vmaxb(Rtt,Rss)
Rdd=vmaxub(Rtt,Rss)

Vector maximum halfwords
Rdd=vmaxh(Rtt,Rss)
Rdd=vmaxuh(Rtt,Rss)

Vector reduce maximum halfwords
Rxx=vrmaxh(Rss,Ru)
Rxx=vrmaxuh(Rss,Ru)

Vector reduce maximum words
Rxx=vrmaxuw(Rss,Ru)
Rxx=vrmaxw(Rss,Ru)

Vector maximum words
Rdd=vmaxuw(Rtt,Rss)
Rdd=vmaxw(Rtt,Rss)

Vector minimum bytes
Rdd=vminb(Rtt,Rss)
Rdd=vminub(Rtt,Rss)

Vector minimum halfwords
Rdd=vminh(Rtt,Rss)
Rdd=vminuh(Rtt,Rss)

Vector reduce minimum halfwords
Rxx=vrminh(Rss,Ru)
Rxx=vrminuh(Rss,Ru)

Vector reduce minimum words
Rxx=vrminuw(Rss,Ru)
Rxx=vrminw(Rss,Ru)

Vector minimum words
Rdd=vminuw(Rtt,Rss)
Rdd=vminw(Rtt,Rss)

Vector sum of absolute differences unsigned bytes
Rdd=vrsadub(Rss,Rtt)
Rxx+=vrsadub(Rss,Rtt)

Vector subtract halfwords
Rdd=vsubh(Rtt,Rss)[:sat]
Rdd=vsubuh(Rtt,Rss):sat

Vector subtract bytes
Rdd=vsubb(Rss,Rtt)
Rdd=vsubub(Rtt,Rss)[:sat]

Vector subtract words
Rdd=vsubw(Rtt,Rss)[:sat]

Vector add compare and select maximum halfwords
Rxx,Pe=vacsh(Rss,Rtt)

XTYPE/BIT

Count leading
Rd=add(clb(Rs),#s6)
Rd=add(clb(Rss),#s6)
Rd=cl0(Rs)
Rd=cl0(Rss)
Rd=cl1(Rs)
Rd=cl1(Rss)
Rd=clb(Rs)
Rd=clb(Rss)
Rd=normamt(Rs)
Rd=normamt(Rss)

Count population
Rd=popcount(Rss)

Count trailing
Rd=ct0(Rs)
Rd=ct0(Rss)
Rd=ct1(Rs)
Rd=ct1(Rss)

Extract bitfield
Rd=extract(Rs,#u5,#U5)
Rd=extract(Rs,Rtt)
Rd=extractu(Rs,#u5,#U5)
Rd=extractu(Rs,Rtt)
Rdd=extract(Rss,#u6,#U6)
Rdd=extract(Rss,Rtt)
Rdd=extractu(Rss,#u6,#U6)
Rdd=extractu(Rss,Rtt)

Insert bitfield
Rx=insert(Rs,#u5,#U5)
Rx=insert(Rs,Rtt)
Rxx=insert(Rss,#u6,#U6)
Rxx=insert(Rss,Rtt)

Interleave/deinterleave
Rdd=deinterleave(Rss)
Rdd=interleave(Rss)

Linear feedback-shift iteration
Rdd=lfs(Rss,Rtt)

Masked parity
Rd=parity(Rs,Rt)
Rd=parity(Rss,Rtt)

Bit reverse
Rd=brev(Rs)
Rdd=brev(Rss)

Set/clear/toggle bit
Rd=clrbit(Rs,#u5)
Rd=clrbit(Rs,Rt)
Rd=setbit(Rs,#u5)
Rd=setbit(Rs,Rt)
Rd=togglebit(Rs,#u5)
Rd=togglebit(Rs,Rt)

Split bitfield
Rdd=bitsplit(Rs,#u5)
Rdd=bitsplit(Rs,Rt)

Table index
Rx=tableidxb(Rs,#u4,#S6):raw
Rx=tableidxb(Rs,#u4,#U5)
Rx=tableidxd(Rs,#u4,#S6):raw
Rx=tableidxd(Rs,#u4,#U5)
Rx=tableidxh(Rs,#u4,#S6):raw
Rx=tableidxh(Rs,#u4,#U5)
Rx=tableidxw(Rs,#u4,#S6):raw
Rx=tableidxw(Rs,#u4,#U5)

XTYPE/COMPLEX

Complex add/sub halfwords
Rdd=vxaddsubh(Rss,Rtt):rnd:>>1:sat
Rdd=vxaddsubh(Rss,Rtt):sat
Rdd=vxsubaddh(Rss,Rtt):rnd:>>1:sat
Rdd=vxsubaddh(Rss,Rtt):sat

Complex add/sub words
Rdd=vxaddsubw(Rss,Rtt):sat
Rdd=vxsubaddw(Rss,Rtt):sat

Complex multiply
Rdd=cmpy(Rs,Rt)[:<<1]:sat
Rdd=cmpy(Rs,Rt*)[:<<1]:sat
Rxx+=cmpy(Rs,Rt)[:<<1]:sat
Rxx+=cmpy(Rs,Rt*)[:<<1]:sat
Rxx-=cmpy(Rs,Rt)[:<<1]:sat
Rxx-=cmpy(Rs,Rt*)[:<<1]:sat

Complex multiply real or imaginary
Rdd=cmpyi(Rs,Rt)
Rdd=cmpyr(Rs,Rt)
Rxx+=cmpyi(Rs,Rt)
Rxx+=cmpyr(Rs,Rt)

Complex multiply with round and pack
Rd=cmpy(Rs,Rt)[:<<1]:rnd:sat
Rd=cmpy(Rs,Rt*)[:<<1]:rnd:sat

Complex multiply 32x16
Rd=cmpyiwh(Rss,Rt):<<1:rnd:sat
Rd=cmpyiwh(Rss,Rt*):<<1:rnd:sat
Rd=cmpyrwh(Rss,Rt):<<1:rnd:sat
Rd=cmpyrwh(Rss,Rt*):<<1:rnd:sat

Vector complex multiply real or imaginary
Rdd=vcmpyi(Rss,Rtt)[:<<1]:sat
Rdd=vcmpyr(Rss,Rtt)[:<<1]:sat
Rxx+=vcmpyi(Rss,Rtt):sat
Rxx+=vcmpyr(Rss,Rtt):sat

Vector complex conjugate
Rdd=vconj(Rss):sat

Vector complex rotate
Rdd=vcrotate(Rss,Rt)

Vector reduce complex multiply real or imaginary
Rdd=vrcmpyi(Rss,Rtt)
Rdd=vrcmpyi(Rss,Rtt*)
Rdd=vrcmpyr(Rss,Rtt)
Rdd=vrcmpyr(Rss,Rtt*)
Rxx+=vrcmpyi(Rss,Rtt)
Rxx+=vrcmpyi(Rss,Rtt*)
Rxx+=vrcmpyr(Rss,Rtt)
Rxx+=vrcmpyr(Rss,Rtt*)

Vector reduce complex multiply by scalar
Rdd=vrcmpys(Rss,Rt):<<1:sat
Rdd=vrcmpys(Rss,Rtt):<<1:sat:raw:hi
Rdd=vrcmpys(Rss,Rtt):<<1:sat:raw:lo
Rxx+=vrcmpys(Rss,Rt):<<1:sat
Rxx+=vrcmpys(Rss,Rtt):<<1:sat:raw:hi
Rxx+=vrcmpys(Rss,Rtt):<<1:sat:raw:lo

Vector reduce complex multiply by scalar with round and pack
Rd=vrcmpys(Rss,Rt):<<1:rnd:sat
Rd=vrcmpys(Rss,Rtt):<<1:rnd:sat:raw:hi
Rd=vrcmpys(Rss,Rtt):<<1:rnd:sat:raw:lo

Vector reduce complex rotate
Rdd=vrcrotate(Rss,Rt,#u2)
Rxx+=vrcrotate(Rss,Rt,#u2)

XTYPE/FP

Floating point addition
Rd=sfadd(Rs,Rt)

Classify floating-point value
Pd=dfclass(Rss,#u5)
Pd=sfclass(Rs,#u5)

Compare floating-point value
Pd=dfcmp.eq(Rss,Rtt)
Pd=dfcmp.ge(Rss,Rtt)
Pd=dfcmp.gt(Rss,Rtt)
Pd=dfcmp.uo(Rss,Rtt)
Pd=sfcmp.eq(Rs,Rt)
Pd=sfcmp.ge(Rs,Rt)
Pd=sfcmp.gt(Rs,Rt)
Pd=sfcmp.uo(Rs,Rt)

Convert floating-point value to other format
Rd=convert_df2sf(Rss)
Rdd=convert_sf2df(Rs)

Convert integer to floating-point value
Rd=convert_d2sf(Rss)
Rd=convert_ud2sf(Rss)
Rd=convert_uw2sf(Rs)
Rd=convert_w2sf(Rs)
Rdd=convert_d2df(Rss)
Rdd=convert_ud2df(Rss)
Rdd=convert_uw2df(Rs)
Rdd=convert_w2df(Rs)

Convert floating-point value to integer
Rd=convert_df2uw(Rss)
Rd=convert_df2uw(Rss):chop
Rd=convert_df2w(Rss)
Rd=convert_df2w(Rss):chop
Rd=convert_sf2uw(Rs)
Rd=convert_sf2uw(Rs):chop
Rd=convert_sf2w(Rs)
Rd=convert_sf2w(Rs):chop
Rdd=convert_df2d(Rss)
Rdd=convert_df2d(Rss):chop
Rdd=convert_df2ud(Rss)
Rdd=convert_df2ud(Rss):chop
Rdd=convert_sf2d(Rs)
Rdd=convert_sf2d(Rs):chop
Rdd=convert_sf2ud(Rs)
Rdd=convert_sf2ud(Rs):chop

Floating point extreme value assistance
Rd=sffixupd(Rs,Rt)
Rd=sffixupn(Rs,Rt)
Rd=sffixupr(Rs)

Floating point fused multiply-add
Rx+=sfmpy(Rs,Rt)
Rx-=sfmpy(Rs,Rt)

Floating point fused multiply-add with scaling
Rx+=sfmpy(Rs,Rt,Pu):scale

Floating point reciprocal square root approximation
Rd,Pe=sfinvsqrta(Rs)

Floating point fused multiply-add for library routines
Rx+=sfmpy(Rs,Rt):lib
Rx-=sfmpy(Rs,Rt):lib

Create floating-point constant
Rd=sfmake(#u10):neg
Rd=sfmake(#u10):pos
Rdd=dfmake(#u10):neg
Rdd=dfmake(#u10):pos

Floating point maximum
Rd=sfmax(Rs,Rt)

Floating point minimum
Rd=sfmin(Rs,Rt)

Floating point multiply
Rd=sfmpy(Rs,Rt)

Floating point reciprocal approximation
Rd,Pe=sfrecipa(Rs,Rt)

Floating point subtraction
Rd=sfsub(Rs,Rt)

XTYPE/MPY

Multiply and use lower result
Rd=+mpyi(Rs,#u8)
Rd=-mpyi(Rs,#u8)
Rd=add(#u6,mpyi(Rs,#U6))
Rd=add(#u6,mpyi(Rs,Rt))
Rd=add(Ru,mpyi(#u6:2,Rs))
Rd=add(Ru,mpyi(Rs,#u6))
Rd=mpyi(Rs,#m9)
Rd=mpyi(Rs,Rt)
Rd=mpyui(Rs,Rt)
Rx+=mpyi(Rs,#u8)
Rx+=mpyi(Rs,Rt)
Rx-=mpyi(Rs,#u8)
Ry=add(Ru,mpyi(Ry,Rs))

Vector multiply word by signed half (32x16)
Rdd=vmpyweh(Rss,Rtt)[:<<1]:rnd:sat
Rdd=vmpyweh(Rss,Rtt)[:<<1]:sat
Rdd=vmpywoh(Rss,Rtt)[:<<1]:rnd:sat
Rdd=vmpywoh(Rss,Rtt)[:<<1]:sat
Rxx+=vmpyweh(Rss,Rtt)[:<<1]:rnd:sat
Rxx+=vmpyweh(Rss,Rtt)[:<<1]:sat
Rxx+=vmpywoh(Rss,Rtt)[:<<1]:rnd:sat
Rxx+=vmpywoh(Rss,Rtt)[:<<1]:sat

Vector multiply word by unsigned half (32x16)
Rdd=vmpyweuh(Rss,Rtt)[:<<1]:rnd:sat
Rdd=vmpyweuh(Rss,Rtt)[:<<1]:sat
Rdd=vmpywouh(Rss,Rtt)[:<<1]:rnd:sat
Rdd=vmpywouh(Rss,Rtt)[:<<1]:sat
Rxx+=vmpyweuh(Rss,Rtt)[:<<1]:rnd:sat
Rxx+=vmpyweuh(Rss,Rtt)[:<<1]:sat
Rxx+=vmpywouh(Rss,Rtt)[:<<1]:rnd:sat
Rxx+=vmpywouh(Rss,Rtt)[:<<1]:sat

Multiply signed halfwords
Rd=mpy(Rs.[HL],Rt.[HL])[:<<1][:rnd][:sat]
Rdd=mpy(Rs.[HL],Rt.[HL])[:<<1][:rnd]
Rx+=mpy(Rs.[HL],Rt.[HL])[:<<1][:sat]
Rx-=mpy(Rs.[HL],Rt.[HL])[:<<1][:sat]
Rxx+=mpy(Rs.[HL],Rt.[HL])[:<<1]
Rxx-=mpy(Rs.[HL],Rt.[HL])[:<<1]

Multiply unsigned halfwords
Rd=mpyu(Rs.[HL],Rt.[HL])[:<<1]
Rdd=mpyu(Rs.[HL],Rt.[HL])[:<<1]
Rx+=mpyu(Rs.[HL],Rt.[HL])[:<<1]
Rx-=mpyu(Rs.[HL],Rt.[HL])[:<<1]
Rxx+=mpyu(Rs.[HL],Rt.[HL])[:<<1]
Rxx-=mpyu(Rs.[HL],Rt.[HL])[:<<1]

Polynomial multiply words
Rdd=pmpyw(Rs,Rt)
Rxx^=pmpyw(Rs,Rt)

Vector reduce multiply word by signed half (32x16)
Rdd=vrmpyweh(Rss,Rtt)[:<<1]
Rdd=vrmpywoh(Rss,Rtt)[:<<1]
Rxx+=vrmpyweh(Rss,Rtt)[:<<1]
Rxx+=vrmpywoh(Rss,Rtt)[:<<1]

Multiply and use upper result
Rd=mpy(Rs,Rt.H):<<1:rnd:sat
Rd=mpy(Rs,Rt.H):<<1:sat
Rd=mpy(Rs,Rt.L):<<1:rnd:sat
Rd=mpy(Rs,Rt.L):<<1:sat
Rd=mpy(Rs,Rt)
Rd=mpy(Rs,Rt):<<1
Rd=mpy(Rs,Rt):<<1:sat
Rd=mpy(Rs,Rt):rnd
Rd=mpysu(Rs,Rt)
Rd=mpyu(Rs,Rt)
Rx+=mpy(Rs,Rt):<<1:sat
Rx-=mpy(Rs,Rt):<<1:sat

Multiply and use full result
Rdd=mpy(Rs,Rt)
Rdd=mpyu(Rs,Rt)
Rxx[+-]=mpy(Rs,Rt)
Rxx[+-]=mpyu(Rs,Rt)

Vector dual multiply
Rdd=vdmpy(Rss,Rtt):<<1:sat
Rdd=vdmpy(Rss,Rtt):sat
Rxx+=vdmpy(Rss,Rtt):<<1:sat
Rxx+=vdmpy(Rss,Rtt):sat

Vector dual multiply with round and pack
Rd=vdmpy(Rss,Rtt)[:<<1]:rnd:sat

Vector reduce multiply bytes
Rdd=vrmpybsu(Rss,Rtt)
Rdd=vrmpybu(Rss,Rtt)
Rxx+=vrmpybsu(Rss,Rtt)
Rxx+=vrmpybu(Rss,Rtt)

Vector dual multiply signed by unsigned bytes
Rdd=vdmpybsu(Rss,Rtt):sat
Rxx+=vdmpybsu(Rss,Rtt):sat

Vector multiply even halfwords
Rdd=vmpyeh(Rss,Rtt):<<1:sat
Rdd=vmpyeh(Rss,Rtt):sat
Rxx+=vmpyeh(Rss,Rtt)
Rxx+=vmpyeh(Rss,Rtt):<<1:sat
Rxx+=vmpyeh(Rss,Rtt):sat

Vector multiply halfwords
Rdd=vmpyh(Rs,Rt)[:<<1]:sat
Rxx+=vmpyh(Rs,Rt)
Rxx+=vmpyh(Rs,Rt)[:<<1]:sat

Vector multiply halfwords with round and pack
Rd=vmpyh(Rs,Rt)[:<<1]:rnd:sat

Vector multiply halfwords, signed by unsigned
Rdd=vmpyhsu(Rs,Rt)[:<<1]:sat
Rxx+=vmpyhsu(Rs,Rt)[:<<1]:sat

Vector reduce multiply halfwords
Rdd=vrmpyh(Rss,Rtt)
Rxx+=vrmpyh(Rss,Rtt)

Vector multiply bytes
Rdd=vmpybsu(Rs,Rt)
Rdd=vmpybu(Rs,Rt)
Rxx+=vmpybsu(Rs,Rt)
Rxx+=vmpybu(Rs,Rt)

Vector polynomial multiply halfwords
Rdd=vpmpyh(Rs,Rt)
Rxx^=vpmpyh(Rs,Rt)

XTYPE/PERM

CABAC decode bin
Rdd=decbin(Rss,Rtt)

Saturate
Rd=sat(Rss)
Rd=satb(Rs)
Rd=sath(Rs)
Rd=satub(Rs)
Rd=satuh(Rs)

Swizzle bytes
Rd=swiz(Rs)

Vector align
Rdd=valignb(Rtt,Rss,#u3)
Rdd=valignb(Rtt,Rss,Pu)

Vector round and pack
Rd=vrndwh(Rss)
Rd=vrndwh(Rss):sat

Vector saturate and pack
Rd=vsathb(Rs)
Rd=vsathb(Rss)
Rd=vsathub(Rs)
Rd=vsathub(Rss)
Rd=vsatwh(Rss)
Rd=vsatwuh(Rss)

Vector saturate without pack
Rdd=vsathb(Rss)
Rdd=vsathub(Rss)
Rdd=vsatwh(Rss)
Rdd=vsatwuh(Rss)

Vector shuffle
Rdd=shuffeb(Rss,Rtt)
Rdd=shuffeh(Rss,Rtt)
Rdd=shuffob(Rtt,Rss)
Rdd=shuffoh(Rtt,Rss)

Vector splat bytes
Rd=vsplatb(Rs)

Vector splat halfwords
Rdd=vsplath(Rs

Vector splice
Rdd=vspliceb(Rss,Rtt,#u3)
Rdd=vspliceb(Rss,Rtt,Pu)

Vector sign extend
Rdd=vsxtbh(Rs)
Rdd=vsxthw(Rs)

Vector truncate
Rd=vtrunehb(Rss)
Rd=vtrunohb(Rss)
Rdd=vtrunewh(Rss,Rtt)
Rdd=vtrunowh(Rss,Rtt)

Vector zero extend
Rdd=vzxtbh(Rs)
Rdd=vzxthw(Rs)

XTYPE/PRED

Bounds check
Pd=boundscheck(Rs,Rtt)
Pd=boundscheck(Rss,Rtt):raw:hi
Pd=boundscheck(Rss,Rtt):raw:lo

Compare byte
Pd=cmpb.eq(Rs,#u8)
Pd=cmpb.eq(Rs,Rt)
Pd=cmpb.gt(Rs,#s8)
Pd=cmpb.gt(Rs,Rt)
Pd=cmpb.gtu(Rs,#u7)
Pd=cmpb.gtu(Rs,Rt)

Compare half
Pd=cmph.eq(Rs,#s8)
Pd=cmph.eq(Rs,Rt)
Pd=cmph.gt(Rs,#s8)
Pd=cmph.gt(Rs,Rt)
Pd=cmph.gtu(Rs,#u7)
Pd=cmph.gtu(Rs,Rt)

Compare doublewords
Pd=cmp.eq(Rss,Rtt)
Pd=cmp.gt(Rss,Rtt)
Pd=cmp.gtu(Rss,Rtt)

Compare bit mask
Pd=[!]bitsclr(Rs,#u6)
Pd=[!]bitsclr(Rs,Rt)
Pd=[!]bitsset(Rs,Rt)

Mask generate from predicate
Rdd=mask(Pt)

Check for TLB match
Pd=tlbmatch(Rss,Rt)

Predicate transfer
Pd=Rs
Rd=Ps

Test bit

Pd=[!]tstbit(Rs,#u5)
Pd=[!]tstbit(Rs,Rt)

Vector compare halfwords

Pd=vcmph.eq(Rss,#s8)
Pd=vcmph.eq(Rss,Rtt)
Pd=vcmph.gt(Rss,#s8)
Pd=vcmph.gt(Rss,Rtt)
Pd=vcmph.gtu(Rss,#u7)
Pd=vcmph.gtu(Rss,Rtt)

Vector compare bytes for any match

Pd=any8(vcmpb.eq(Rss,Rtt))

Vector compare bytes

Pd=vcmpb.eq(Rss,#u8)
Pd=vcmpb.eq(Rss,Rtt)
Pd=vcmpb.gt(Rss,#s8)
Pd=vcmpb.gt(Rss,Rtt)
Pd=vcmpb.gtu(Rss,#u7)
Pd=vcmpb.gtu(Rss,Rtt)

Vector compare words

Pd=vcmpw.eq(Rss,#s8)
Pd=vcmpw.eq(Rss,Rtt)
Pd=vcmpw.gt(Rss,#s8)
Pd=vcmpw.gt(Rss,Rtt)
Pd=vcmpw.gtu(Rss,#u7)
Pd=vcmpw.gtu(Rss,Rtt)

Viterbi pack even and odd predicate bits

Rd=vitpack(Ps,Pt)

Vector mux

Rdd=vmux(Pu,Rss,Rtt)

XTYPE/SHIFT

Shift by immediate

Rd=asl(Rs,#u5)
Rd=asr(Rs,#u5)
Rd=lsr(Rs,#u5)
Rdd=asl(Rss,#u6)
Rdd=asr(Rss,#u6)
Rdd=lsr(Rss,#u6)

Shift by immediate and accumulate

Rx=add(#u8,asl(Rx,#U5))
Rx=add(#u8,lsr(Rx,#U5))
Rx=sub(#u8,asl(Rx,#U5))
Rx=sub(#u8,lsr(Rx,#U5))
Rx[+-]=asl(Rs,#u5)
Rx[+-]=asr(Rs,#u5)
Rx[+-]=lsr(Rs,#u5)
Rxx[+-]=asl(Rss,#u6)
Rxx[+-]=asr(Rss,#u6)
Rxx[+-]=lsr(Rss,#u6)

Shift by immediate and add

Rd=addasl(Rt,Rs,#u3)

Shift by immediate and logical

Rx=and(#u8,asl(Rx,#U5))
Rx=and(#u8,lsr(Rx,#U5))
Rx=or(#u8,asl(Rx,#U5))
Rx=or(#u8,lsr(Rx,#U5))
Rx[&|]=asl(Rs,#u5)
Rx[&|]=asr(Rs,#u5)
Rx[&|]=lsr(Rs,#u5)
Rx^=asl(Rs,#u5)
Rx^=lsr(Rs,#u5)
Rxx[&|]=asl(Rss,#u6)
Rxx[&|]=asr(Rss,#u6)
Rxx[&|]=lsr(Rss,#u6)
Rxx^=asl(Rss,#u6)
Rxx^=lsr(Rss,#u6)

Shift right by immediate with rounding

Rd=asr(Rs,#u5):rnd
Rd=asrrnd(Rs,#u5)
Rdd=asr(Rss,#u6):rnd
Rdd=asrrnd(Rss,#u6)

Shift left by immediate with saturation

Rd=asl(Rs,#u5):sat

Shift by register

Rd=asl(Rs,Rt)
Rd=asr(Rs,Rt)
Rd=lsl(#s6,Rt)
Rd=lsl(Rs,Rt)
Rd=lsr(Rs,Rt)
Rdd=asl(Rss,Rt)
Rdd=asr(Rss,Rt)
Rdd=lsl(Rss,Rt)
Rdd=lsr(Rss,Rt)

Shift by register and accumulate

Rx[+-]=asl(Rs,Rt)
Rx[+-]=asr(Rs,Rt)
Rx[+-]=lsl(Rs,Rt)
Rx[+-]=lsr(Rs,Rt)
Rxx[+-]=asl(Rss,Rt)
Rxx[+-]=asr(Rss,Rt)
Rxx[+-]=lsl(Rss,Rt)
Rxx[+-]=lsr(Rss,Rt)

Shift by register and logical

Rx[&|]=asl(Rs,Rt)
Rx[&|]=asr(Rs,Rt)
Rx[&|]=lsl(Rs,Rt)
Rx[&|]=lsr(Rs,Rt)
Rxx[&|]=asl(Rss,Rt)
Rxx[&|]=asr(Rss,Rt)
Rxx[&|]=lsl(Rss,Rt)
Rxx[&|]=lsr(Rss,Rt)
Rxx^=asl(Rss,Rt)
Rxx^=asr(Rss,Rt)
Rxx^=lsl(Rss,Rt)
Rxx^=lsr(Rss,Rt)

Shift by register with saturation

Rd=asl(Rs,Rt):sat
Rd=asr(Rs,Rt):sat

Vector shift halfwords by immediate

Rdd=vaslh(Rss,#u4)
Rdd=vasrh(Rss,#u4)
Rdd=vlsrh(Rss,#u4)

Vector arithmetic shift halfwords with round

Rdd=vasrh(Rss,#u4):raw
Rdd=vasrh(Rss,#u4):rnd

Vector arithmetic shift halfwords with saturate and pack

Rd=vasrhub(Rss,#u4):raw
Rd=vasrhub(Rss,#u4):rnd:sat
Rd=vasrhub(Rss,#u4):sat

Vector shift halfwords by register

Rdd=vaslh(Rss,Rt)
Rdd=vasrh(Rss,Rt)
Rdd=vlslh(Rss,Rt)
Rdd=vlsrh(Rss,Rt)

Vector shift words by immediate

Rdd=vaslw(Rss,#u5)
Rdd=vasrw(Rss,#u5)
Rdd=vlsrw(Rss,#u5)

Vector shift words by register

Rdd=vaslw(Rss,Rt)
Rdd=vasrw(Rss,Rt)
Rdd=vlslw(Rss,Rt)
Rdd=vlsrw(Rss,Rt)

Vector shift words with truncate and pack

Rd=vasrw(Rss,#u5)
Rd=vasrw(Rss,Rt)