MIPS® Architecture Base: nanoMIPS32™ Instruction Set Technical Reference Manual

Revision 01.01

April 27, 2018

Public

Document Number: MD01247

Contents

Chapter 1

Introduction

nanoMIPS™ is a variable length ISA containing 16, 32 and 48 bit wide instructions. It is designed to be portable at assembly level with other MIPS™ and microMIPS™ code, but contains a number of changes which enhance code density and efficiency. The key changes are the resizing of immediate values to allocate opcode space to the most frequently used cases, addition of new instructions to execute frequently used code sequences more efficiently, and changes to the optimized register set to target enhancements in the ABI conventions. In addition, branch delay slots and forbidden slots have been removed.

It is legal for a core to implement a specific subset of nanoMIPS™ instructions, known as the ”nanoMIPS Subset” (NMS). Such cores have Config5.NMS (bit 21) set to 1. For some applications, the trade-off of reduced functionality in exchange for the reduced implementation cost of the smaller instruction set may be desirable. Instructions which are not implemented in the nanoMIPS™ subset are marked as ”not available in NMS”.

This document is structured as follows:

The Instruction Summary chapter gives a summary of all the instructions in the base nanoMIPS™ ISA.

The Instruction Definitions chapter defines the behavior of every nanoMIPS™ instruction.

The Shared Pseudocode Functions chapter contains the shared pseudocode functions which are referenced by the instruction definitions.

The Specification Conventions appendix explains the conventions used in the instruction definitions and pseudocode.

The Opcode Map appendix gives a hierarchical view of the nanoMIPS™ opcode map.

The Change Log appendix contains the change log for this document.

Chapter 2

Instruction Summary

This chapter lists all instructions formats in the nanoMIPS™ base ISA, with a brief description of their operation and availability. The information in this chapter is intended as a summary only, and the reader should refer to the Instruction Definitions section for the precise specification.

Instruction	Operation	Availability
ADD `rd, rs, rt`	`rd = rs + rt  # Trap on overflow`	
ADDIU[32] `rt, rs, u16`	`rt = rs + u16`	
ADDIU[48] `rt, s32`	`rt = rt + s32`	Not in NMS
ADDIU[GP48] `rt, gp, s32`	`rt = gp + s32`	Not in NMS Not in P64 mode
ADDIU[GP.B] `rt, gp, u18`	`rt = gp + u18`	Not in P64 mode
ADDIU[GP.W] `rt, gp, u21`	`rt = gp + u21  # Word aligned imm`	Not in P64 mode
ADDIU[R1.SP] `rt3, sp, u8`	`rt3 = sp + u8`	
ADDIU[R2] `rt3, rs3, u5`	`rt3 = rs3 + u5`	
ADDIU[RS5] `rt, s5`	`rt = rt + s5`	
ADDIU[NEG] `rt, rs, -u12`	`rt = rs - u12`	
ADDIUPC[32] `rt, s22`	`rt = pc + s22 # Halfword aligned imm`	Not in P64 mode
ADDIUPC[48] `rt, s32`	`rt = pc + s32`	Not in NMS Not in P64 mode
ADDU[32] `dst, src1, src2`	`dst = src1 + src2`	
ADDU[16] `rd3, rs3, rt3`	`rd3 = rs3 + rt3`	
ADDU[4x4] `rt4, rs4`	`rt4 = rt4 + rs4`	Not in NMS
ALUIPC `rt, %pcrel_hi(addr)`	`rt = pcrel_hi(addr)`	

Instruction	Operation	Availability
AND[32] rd, rs, rt	rd = rs & rt	
AND[16] rt3, rs3	rt3 = rt3 & rs3	
ANDI[32] rt, rs, u12	rt = rs & u12	
ANDI[16] rt3, rs3, imm4	rt3 = rs3 & imm4	
BALC[32] addr	ra = next_pc; pc = addr	
BALC[16] addr	ra = next_pc; pc = addr	
BALRSC rt, rs	rt = next_pc; pc = next_pc + (rs<<1)	
BBEQZC rt, bit, addr	if ((rt>>bit) & 1) == 0: pc = addr	Not in NMS
BBNEZC rt, bit, addr	if ((rt>>bit) & 1) != 0: pc = addr	Not in NMS
BC[16] addr	pc = addr	
BC[32] addr	pc = addr	
BEQC[16] rs3, rt3, addr	if rs3 == rt3: pc = addr	Not in NMS
BEQC[32] rs, rt, addr	if rs == rt: pc = addr	
BEQIC rt, u7, addr	if rt == u7: pc = addr	
BEQZC[16] rt3, addr	if rt3 == 0: pc = addr	
BGEC rs, rt, addr	if rs >= rt: pc = addr	
BGEIC rt, u7, addr	if rt >= u7: pc = addr	
BGEIUC rt, u7, addr	if unsigned(rt) >= u7: pc = addr	
BGEUC rs, rt, addr	if unsigned(rs) >= unsigned(rt): pc = addr	
BLTC rs, rt, addr	if rs < rt: pc = addr	
BLTIC rt, u7, addr	if rt < u7: pc = addr	
BLTIUC rt, u7, addr	if unsigned(rt) < u7: pc = addr	
BLTUC rs, rt, addr	if unsigned(rs) < unsigned(rt): pc = addr	
BNEC[16] rs3, rt3, addr	if rs3 != rt3: pc = addr	Not in NMS
BNEC[32] rs, rt, addr	if rs != rt: pc = addr	
BNEIC rt, u7, addr	if rt != u7: pc = addr	

Instruction	Operation	Availability
BNEZC[16] rt3, addr	if rt3 != 0: pc = addr	
BREAK[16] code3	raise exception('BP')	
BREAK[32] code19	raise exception('BP')	
BRSC rs	pc = next_pc + (rs << 1)	
CACHE op, s9(rs)	cacheop(op, rs + s9)	Needs CP0
CACHEE op, s9((rs)	cacheop(op, rs + s9, eva=True)	If Config5.EVA=1 Needs CP0
CLO rt, rs	rt = count_leading_ones(rs,32)	Not in NMS
CLZ rt, rs	rt = count_leading_zeros(rs,32)	Not in NMS
CRC32B rt, rs	rt = crc32(rt, rs[7:0], 0xedb88320)	If Config5.CRCP=1
CRC32CB rt, rs	rt = crc32(rt, rs[7:0], 0x82f63b78)	If Config5.CRCP=1
CRC32CH rt, rs	rt = crc32(rt, rs[15:0], 0x82f63b78)	If Config5.CRCP=1
CRC32CW rt, rs	rt = crc32(rt, rs[31:0], 0x82f63b78)	If Config5.CRCP=1
CRC32H rt, rs	rt = crc32(rt, rs[15:0], 0xedb88320)	If Config5.CRCP=1
CRC32W rt, rs	rt = crc32(rt, rs[31:0], 0xedb88320)	If Config5.CRCP=1
DERET	pc = C0.DEPC; C0.Debug.DM = 0	If Debug present
DI rt	rt = C0.Status; C0.Status.IE = 0	Needs CP0
DIV rd, rs, rt	rd = rs / rt	
DIVU rd, rs, rt	rd = unsigned(rs) / unsigned(rt)	
DVP rt	rt = C0.VPECtl; C0.VPECtl.DIS = 1	If Config5.VP=1 Needs CP0
EHB	clear_execution_hazards()	
EI rt	rt = C0.Status; C0.Status.IE = 1	Needs CP0
ERET	pc = C0.EPC; C0.Status.EXL = 0; C0.LLAddr.LLB = 0	Needs CP0
ERETNC	pc = C0.EPC; C0.Status.EXL = 0; C0.LLAddr.LLB = <unchanged>	If Config5.LLB!=0 Needs CP0
EVP rt	rt = C0.VPECtl; C0.VPECtl.DIS = 0	If Config5.VP=1 Needs CP0
EXT rt, rs, pos, size	rt = rs[pos+size-1:pos]	Not in NMS

Instruction	Operation	Availability
EXTW rd, rs, rt, shift	rd = (rt[31:0] @ rs[31:0]) >> shift	
GINVI rs	globally_invalidate_icache(target=rs)	If Config5.GI>=2 Needs CP0
GINVT rs, type	globally_invalidate_tlb(va=rs, type)	If Config5.GI=3 Needs CP0
INS rt, rs, pos, size	rt[pos+size-1:size] = rs[size-1:0]	Not in NMS
JALRC.HB rt, rs	pc = rs; rt = next_pc; clear_instruction_hazards()	
JALRC[16] src	pc = src; ra = next_pc	
JALRC[32] dst, src	pc = src; dst = next_pc	
JRC rt	pc = rt	
LBE rt, s9(rs)	rt = MEM(rs + s9, BYTE, eva=True)	If Config5.EVA=1 Needs CP0
LBUE rt, s9(rs)	rt = unsigned(MEM(rs + s9, BYTE, eva=True))	If Config5.EVA=1 Needs CP0
LBUX rd, rs(rt)	rd = unsigned(MEM(rs + rt, BYTE))	
LBU[16] rt3, u2(rs3)	rt3 = unsigned(MEM(rs3 + u2, BYTE))	
LBU[GP] rt, u18(gp)	rt = unsigned(MEM(gp + u18, BYTE))	
LBU[S9] rt, s9(rs)	rt = unsigned(MEM(rs + s9, BYTE))	
LBU[U12] rt, u12(rs)	rt = unsigned(MEM(rs + u12, BYTE))	
LBX rd, rs(rt)	rd = MEM(rs + rt, BYTE)	
LB[16] rt3, u2(rs3)	rt3 = MEM(rs3 + u2, BYTE)	
LB[GP] rt, u18(gp)	rt = MEM(gp + u18, BYTE)	
LB[S9] rt, s9(rs)	rt = MEM(rs + s9, BYTE)	
LB[U12] rt, u12(rs)	rt = MEM(rs + u12, BYTE)	
LHE rt, s9(rs)	rt = MEM(rs + s9, HALF, eva=True)	If Config5.EVA=1 Needs CP0
LHUE rt, s9(rs)	rt = unsigned(MEM(rs + s9, HALF, eva=True))	If Config5.EVA=1 Needs CP0
LHUX rd, rs(rt)	rd = unsigned(MEM(rs + rt, HALF))	
LHUXS rd, rs(rt)	rd = unsigned(MEM(2*rs + rt), HALF))	

Instruction	Operation	Availability
LHU[16] rt3, u3(rs3)	rt3 = unsigned(MEM(rs3 + u3, HALF))	
LHU[GP] rt, u18(gp)	rt = unsigned(MEM(gp + u18, HALF))	
LHU[S9] rt, s9(rs)	rt = unsigned(MEM(rs + s9, HALF))	
LHU[U12] rt, u12(rs)	rt = unsigned(MEM(rs + u12, HALF))	
LHX rd, rs(rt)	rd = MEM(rs + rt, HALF)	
LHXS rd, rs(rt)	rd = MEM(2*rs + rt, HALF)	
LH[16] rt3, u3(rs3)	rt3 = MEM(rs3 + u3, HALF)	
LH[GP] rt, u18(gp)	rt = MEM(gp + u18, HALF)	
LH[S9] rt, s9(rs)	rt = MEM(rs + s9, HALF)	
LH[U12] rt, u12(rs)	rt = MEM(rs + u12, HALF)	
LI[16] rt3, imm7	rt3 = imm7	
LI[48] rt, s32	rt = s32	Not in NMS
LL rt, s9(rs)	rt = MEM(rs + s9, WORD); record_linked_address()	
LLE rt, s9(rs)	rt = MEM(rs + s9, WORD, eva=True); record_linked_address()	If Config5.EVA=1 Needs CP0
LLWP rt, ru, (rs)	rt = MEM(rs, WORD) ru = MEM(rs + 4, WORD) record_linked_address()	Required (Optional in NMS)
LLWPE rt, ru, (rs)	rt = MEM(rs, WORD, eva=True) ru = MEM(rs + 4, WORD, eva=True) record_linked_address()	If Config5.EVA=1 Needs CP0
LSA rd, rs, rt, u2	rd = (rs << u2) + rt	
LUI rt, %hi(imm)	rd = imm # Load upper 20 bits	
LWE rt, s9(rs)	rt = MEM(rs + s9, WORD, eva=True)	If Config5.EVA=1 Needs CP0
LWM rt, s9(rs), count	GPR[rt] = MEM(rs + s9, WORD) GPR[rt+1] = MEM(rs + s9 + 4, WORD) ...	Not in NMS
LWPC[48] rt, addr	rt = MEM(addr, WORD) # PC relative	Not in NMS

Instruction	Operation	Availability
LWX rd, rs(rt)	rd = MEM(rs + rt, WORD)	
LWXS[16] rd3, rs3(rt3)	rd3 = MEM(4*rs3 + rt3, WORD)	
LWXS[32] rd, rs(rt)	rd = MEM(4*rs + rt, WORD)	
LW[16] rt3, u6(rs3)	rt3 = MEM(rs3 + u6, WORD)	
LW[4X4] rt4, u4(rs4)	rt4 = MEM(rs4 + u4, WORD)	Not in NMS
LW[GP16] rt3, u9(gp)	rt3 = MEM(gp + u9, WORD)	
LW[GP] rt, u21(gp)	rt = MEM(gp + u21, WORD)	
LW[S9] rt, s9(rs)	rt = MEM(rs + s9, WORD)	
LW[SP] rt, u7(sp)	rt = MEM(sp + u7, WORD)	
LW[U12] rt, u12(rs)	rt = MEM(rs + u12, WORD)	
MFC0 rt, c0s, sel	rt = CP0(c0s, sel)	Needs CP0
MFHC0 rt, c0s, sel	rt = CP0(c0s, sel, hi=True)	Required (Optional in NMS) Needs CP0
MOD rd, rs, rt	rd = rs % rt	
MODU rd, rs, rt	rd = unsigned(rs) % unsigned(rt)	
MOVE rt, rs	rt = rs	
MOVE.BALC rd1, rt4, addr	rd1 = rt4; pc = addr; ra = next_pc	Not in NMS
MOVEP d1, d2, s1, s2	d1 = s1; d2 = s2	Not in NMS
MOVEP[REV] d1, d2, s1, s2	d1 = s1; d2 = s2	Not in NMS
MOVN rd, rs, rt	rd = rs if rt != 0 else rd	
MOVZ rd, rs, rt	rd = rs if rt == 0 else rd	
MTC0 rt, c0s, sel	CP0(c0s, sel) = rt	Needs CP0
MTHC0 rt, c0s, sel	CP0(c0s, sel, hi=True) = rt	Required (Optional in NMS) Needs CP0
MUH rd, rs, rt	rd = (rs * rt) >> 32	
MUHU rd, rs, rt	rd = (unsigned(rs)*unsigned(rd))>>32	
MULU rd, rs, rt	rd = unsigned(rs) * unsigned(rd)	
MUL[32] dst, src1, src2	dst = src1 * src2	

Instruction	Operation	Availability
MUL[4X4] rt4, rs4	rt4 = rt4 * rs4	Not in NMS
NOP[16]	# No operation	
NOP[32]	# No operation	
NOR rd, rs, rt	rd = ~(rs | rt)	
NOT[16] rt3, rs3	rt3 = ~rs3	
ORI rt, rs, u12	rt = rs | u12	
OR[16] rt3, rs3	rt3 = rs3 | rt3	
OR[32] rd, rs, rt	rd = rs | rt	
PAUSE	while C0.LLAddr.LLB: CPU.in_pause_state = True	
PREFE hint, s9(rs)	prefetch(hint, rs + s9, eva=True)	If Config5.EVA=1 Needs CP0
PREF[S9] hint, s9(rs)	prefetch(hint, rs + s9)	
PREF[U12] hint, u12(rs)	prefetch(hint, rs + u12)	
RDHWR rt, hs, sel	rt = HWRegister(hs, sel)	Not in NMS
RDPGPR rt, rs	rt = SRS[C0.SRSCtl.PSS][rs]	Needs CP0
RESTORE.JRC[16] args...	restore_caller_saved_regs(args...); pc = ra	
RESTORE.JRC[32] args...	restore_caller_saved_regs(args...); pc = ra	
RESTORE[32] args...	restore_caller_saved_regs(args...)	
ROTR rt, rs, shift	rt = (rs[31:0] @ rs[31:0]) >> shift	
ROTRV rd, rs, rt	rd = (rs[31:0] @ rs[31:0]) >> rt	
ROTX rt, rs, sft, sftx, str	rt = rotx(rs, sft, sftx, str)	Not in NMS
SAVE[16] args...	save_caller_saved_regs(args...)	
SAVE[32] args...	save_caller_saved_regs(args...)	
SBE rt, s9(rs)	MEM(rs + s9, BYTE, eva=True) = rt	If Config5.EVA=1 Needs CP0
SBX rd, rs(rt)	MEM(rs + rt, BYTE) = rd	Not in NMS

Instruction	Operation	Availability
SB[16] rt3, u2(rs3)	MEM(rs3 + u2, BYTE) = rt3	
SB[GP] rt, u18(gp)	MEM(gp + u18, BYTE) = rt	
SB[S9] rt, s9(rs)	MEM(rs + s9, BYTE) = rt	
SB[U12] rt, u12(rs)	MEM(rs + u12, BYTE) = rt	

SC rt, s(rs)	if <atomic>:
	  MEM(rs + s, WORD) = rt
	  rt = 1
	else: rt = 0	

SCE rt, s9(rs)	if <atomic>:
	  MEM(rs + s9, WORD, eva=True) = rt
	  rt = 1
	else: rt = 0
		If Config5.EVA=1
		Needs CP0

SCWP rt, ru, (rs)	if <atomic>:
	  MEM(rs, WORD) = rt
	  MEM(rs + 4, WORD) = ru
	  rt = 1
	else: rt = 0
		Required
		(Optional in NMS)

SCWPE rt, ru, (rs)	if atomic:
	  MEM(rs, WORD, eva=True) = rt
	  MEM(rs + 4, WORD, eva=True) = ru
	  rt = 1
	else: rt = 0
		If Config5.EVA=1
		Needs CP0

SDBBP[16] code3	raise debug_exception('BP')	If Debug present
SDBBP[32] code19	raise debug_exception('BP')	If Debug present
SEB rt, rs	rt = sign_extend(rs[7:0])	Not in NMS
SEH rt, rs	rt = sign_extend(rs[15:0])	
SEQI rt, rs, u12	rt = 1 if rs == u12 else 0	
SHE rt, s9(rs)	MEM(rs + s9, HALF, eva=True) = rt	If Config5.EVA=1 Needs CP0
SHX rd, rs(rt)	MEM(rs + rt, HALF) = rd	Not in NMS
SHXS rd, rs(rt)	MEM(2*rs + rt, HALF) = rd	Not in NMS
SH[16] rt3, u3(rs3)	MEM(rs3 + u3, HALF) = rt3	

Instruction	Operation	Availability
SH[GP] rt, u18(gp)	MEM(gp + u18, HALF) = rt	
SH[S9] rt, s9(rs)	MEM(rs + s9, HALF) = rt	
SH[U12] rt, u12(rs)	MEM(rs + u12, HALF) = rt	
SIGRIE code19	raise exception('RI')	
SLLV rd, rs, rt	rd = rs << rt	
SLL[16] rt3, rs3, shift3	rt3 = rs3 << shift3	
SLL[32] rt, rs, shift	rt = rs << shift	
SLT rd, rs, rt	rd = 1 if rs < rt else 0	
SLTI rt, rs, u12	rd = 1 if rs < u12 else 0	
SLTIU rt, rs, u12	rd = 1 if unsigned(rs) < u12 else 0	
SLTU rd, rs, rt	rd = 1 if unsigned(rs) < unsigned(rt) else 0	
SOV rd, rs, rt	rd = 1 if overflows(rs+rt,32) else 0	
SRA rt, rs, shift	rt = rs >> shift	
SRAV rd, rs, rt	rd = rs >> rt	
SRLV rd, rs, rt	rd = unsigned(rs) >> rt	
SRL[16] rt3, rs3, shift3	rt3 = unsigned(rs3) >> shift3	
SRL[32] rt, rs, shift	rt = unsigned(rs) >> shift	
SUB rd, rs, rt	rd = rs - rt # Trap on overflow	Not in NMS
SUBU[16] rd3, rs3, rt3	rd3 = rs3 - rt3	
SUBU[32] rd, rs, rt	rd = rs - rt	
SWE rt, s9(rs)	MEM(rs + s9, WORD, eva=True) = rt	If Config5.EVA=1 Needs CP0
SWM rt, s9(rs), count	MEM(rs + s9, WORD) = GPR[rt] MEM(rs + s9 + 4, WORD) = GPR[rt+1] ...	Not in NMS
SWPC[48] rt, addr	MEM(addr, WORD) = rt # PC relative	Not in NMS
SWX rd, rs(rt)	MEM(rs + rt, WORD) = rd	Not in NMS
SWXS rd, rs(rt)	MEM(4*rs + rt, WORD) = rd	Not in NMS

Instruction	Operation	Availability
SW[16] rt3, u6(rs3)	MEM(rs3 + u6, WORD) = rt3	
SW[4X4] rt4, u4(rs4)	MEM(rs4 + u4, WORD) = rt4	Not in NMS
SW[GP16] rt3, u9(gp)	MEM(gp + u9, WORD) = rt3	
SW[GP] rt, u21(gp)	MEM(gp + u21, WORD) = rt	
SW[S9] rt, s9(rs)	MEM(rs + s9, WORD) = rt	
SW[SP] rt, u7(sp)	MEM(sp + u7, WORD) = rt	
SW[U12] rt, u12(rs)	MEM(rs + u12, WORD) = rt	
SYNC stype	sync_memory_access(stype)	
SYNCIE s9(rs)	sync_icache(rs + s9, eva=True)	If Config5.EVA=1 Needs CP0
SYNCI[S9] s9(rs)	sync_icache(rs + s9)	
SYNCI[U12] u12(rs)	sync_icache(rs + u12)	
SYSCALL[16] code2	raise exception('SYSCALL')	
SYSCALL[32] code18	raise exception('SYSCALL')	
TEQ rs, rt, code5	if rs == rt: raise exception('TRAP')	Not in NMS
TLBINV	tlbinv() # Invalidate TLB	On TLB cores If Config5.IE>=2 Needs CP0
TLBINVF	tlbinv(flush=True) # Flush TLB	On TLB cores If Config5.IE>=2 Needs CP0
TLBP	tlbp() # Probe TLB	On TLB cores Needs CP0
TLBR	tlbr() # Read TLB	On TLB cores Needs CP0
TLBWI	tlbwi() # Write TLB Index	On TLB cores Needs CP0
TLBWR	tlbwr() # Write TLB Random	On TLB cores Needs CP0
TNE rs, rt, code5	if rs != rt: raise exception('TRAP')	Not in NMS

UALH rt, s9(rs)	# Unaligned OK...
	rt = MEM(rs + s9, HALF)
		Not in NMS

Instruction	Operation	Availability

UALWM rt, s9(rs), count	# Unaligned OK...
	GPR[rt] = MEM(rs + s9, WORD)
	GPR[rt+1] = MEM(rs + s9 + 4, WORD)
	...
		Not in NMS

UASH rt, s9(rs)	# Unaligned OK... MEM(rs + s9, HALF) = rt	Not in NMS
UASWM rt, s9(rs), count	# Unaligned OK... MEM(rs + s9, WORD) = GPR[rt] MEM(rs + s9 + 4, WORD) = GPR[rt+1] ...	Not in NMS
WAIT code10	CPU.in_wait_state = True	
WRPGPR rt, rs	rs = SRS[C0.SRSCtl.PSS][rt]	Needs CP0
XORI rt, rs, u12	rt = rs ^ u12	
XOR[16] rt3, rs3	rt3 = rs3 ^ rt3	
XOR[32] rd, rs, rt	rd = rs ^ rt	

Chapter 3

Instruction Definitions

This chapter contains specifications for every instruction in the nanoMIPS™ base ISA. The reader should refer to the Shared Pseudocode Functions chapter for definitions of the shared functions used in the instruction pseudocode, and the Specification Conventions appendix for an explanation of the methodology used to specify the instruction behavior.

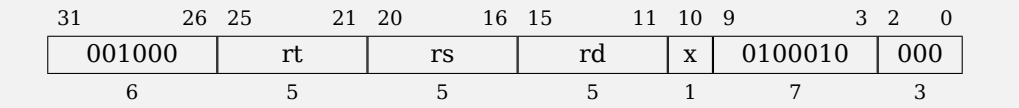
ADD

Assembly: `ADD rd, rs, rt`

Purpose: *Add*. Add two 32-bit signed integers in registers `$rs` and `$rt`, placing the 32-bit result in register `$rd`, and trapping on overflow.

Availability: nanoMIPS, not available in NMS

Format:



Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

sum = GPR[rs] + GPR[rt]
if overflows(sum, nbits=32):
    raise exception('OV')

GPR[rd] = sign_extend(sum, from_nbits=32)
```

Exceptions: Overflow.

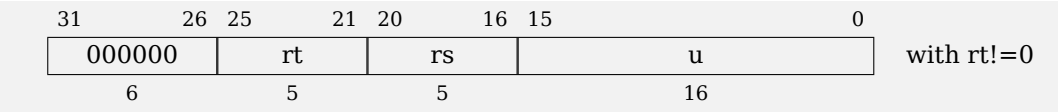
ADDIU

Assembly: `ADDIU rt, rs, imm`

Purpose: *Add Immediate (Untrapped).* Add immediate value `imm` to the 32-bit integer value in register `$rs`, placing the 32-bit result in register `$rt`, and not trapping on overflow.

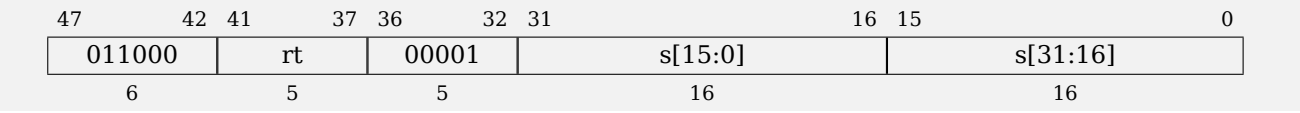
Availability: nanoMIPS, availability varies by format.

Format: ADDIU[32]



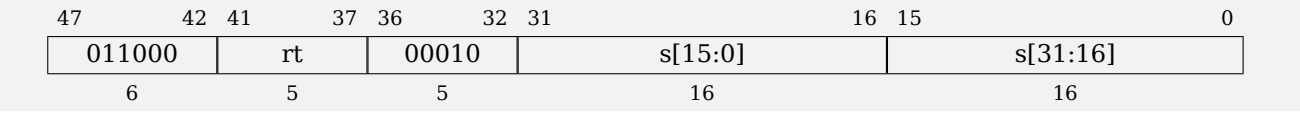

```
imm = u
```

Format: ADDIU[48], not available in NMS



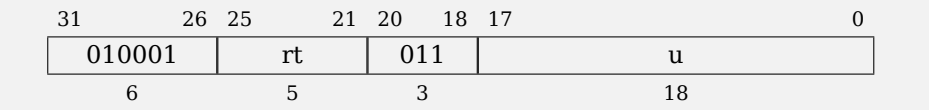

```
if C0.Config5.NMS == 1:
    raise exception('RI')

imm = sign_extend(s, from_nbits=32)
rs = rt
```

Format: ADDIU[GP48], not available in NMS, not available in P64 mode



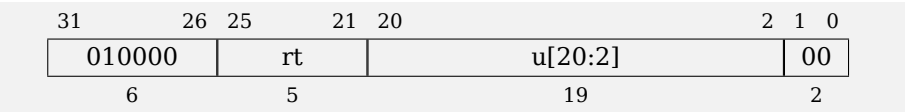

```
if C0.Config5.NMS == 1:
    raise exception('RI')

if pointers_are_64_bits():
    raise behaves_like('DADDIU[GP48]')

imm = sign_extend(s, from_nbits=32)
rs = 28
```

Format: ADDIU[GP.B], not available in P64 mode



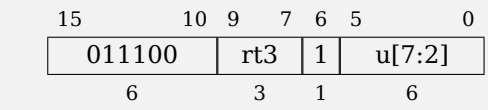

```
if pointers_are_64_bits():
    raise behaves_like('DADDIU[GP.B]')

imm = u
rs = 28
```

Format: ADDIU[GP.W], not available in P64 mode



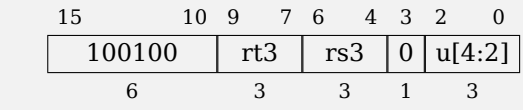

```
if pointers_are_64_bits():
    raise behaves_like('DADDIU[GP.W]')

imm = u
rs = 28
```

Format: ADDIU[R1.SP], not available in P64 mode



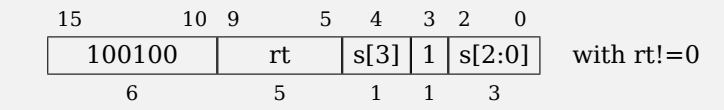

```
if pointers_are_64_bits():
    raise behaves_like('DADDIU[R1.SP]')

rt = decode_gpr(rt3, 'gpr3')
rs = 29
imm = u
```

Format: ADDIU[R2]




```
rt = decode_gpr(rt3, 'gpr3')
rs = decode_gpr(rs3, 'gpr3')
imm = u
```

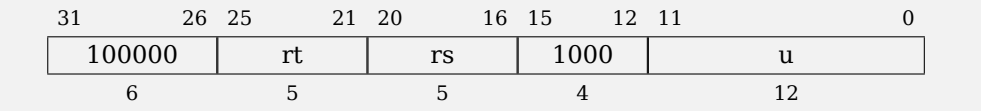
Format: ADDIU[RS5]




```
rs = rt
imm = sign_extend(s, from_nbits=4)
```

ADDIU[RS5] with rt=0 is used to provide a 16 bit NOP instruction.

Format: ADDIU[NEG]




```
imm = -u
```

Operation:

```
sum = GPR[rs] + imm
GPR[rt] = sign_extend(sum, from_nbits=32)
```

Exceptions: Reserved Instruction for ADDIU[48] and ADDIU[GP48] formats on NMS cores.

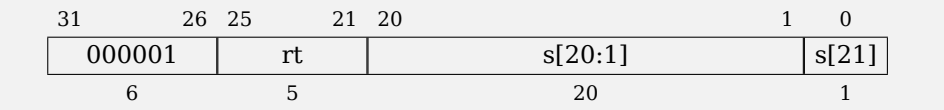
ADDIUPC

Assembly: `ADDIUPC rt, imm`

Purpose: *Add Immediate (Untrapped) to PC*. Compute address by adding immediate value `imm` to the PC and placing the result in register `$rt`, not trapping on overflow.

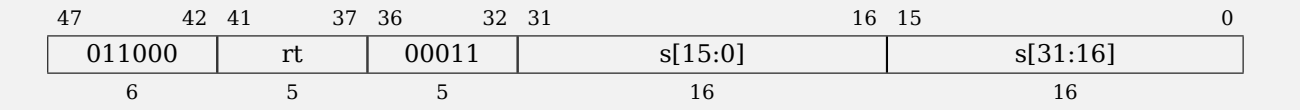
Availability: nanoMIPS, availability varies by format.

Format: ADDIUPC[32], not available in P64 mode




```
if pointers_are_64_bits():
    raise behaves_like('DADDIUPC[32]')

s = sign_extend(s[21] @ s[20:1] @ '0')
imm = s + 4
```

Format: ADDIUPC[48], not available in NMS, not available in P64 mode




```
if C0.Config5.NMS == 1:
    raise exception('RI')

if pointers_are_64_bits():
    raise behaves_like('DADDIUPC[48]')

s = sign_extend(s[31:16] @ s[15:0])
imm = s + 6
```

Operation:

```
GPR[rt] = effective_address(CPU.next_pc, s)
```

Exceptions: Reserved Instruction for ADDIUPC[48] format on NMS cores.

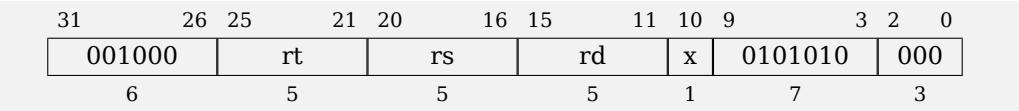
ADDU

Assembly: `ADDU dst, src1, src2`

Purpose: *Add (Untrapped)*. Add two 32-bit integers in registers `$src1` and `$src2`, placing the 32-bit result in register `$dst`, and not trapping on overflow.

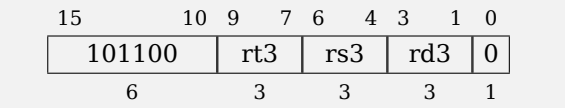
Availability: nanoMIPS, availability varies by format.

Format: ADDU[32]



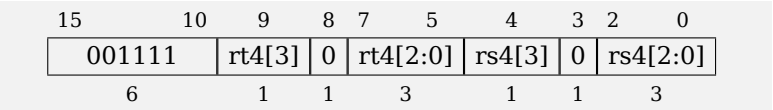

```
dst = rd
src1 = rs
src2 = rt

not_in_nms = False
```

Format: ADDU[16]




```
dst = decode_gpr(rd3, 'gpr3')
src1 = decode_gpr(rs3, 'gpr3')
src2 = decode_gpr(rt3, 'gpr3')

not_in_nms = False
```

Format: ADDU[4X4], not available in NMS




```
dst = decode_gpr(rt4, 'gpr4')
src1 = decode_gpr(rt4, 'gpr4')
src2 = decode_gpr(rs4, 'gpr4')

not_in_nms = True
```

Operation:

```
if not_in_nms and C0.Config5.NMS:
    raise exception('RI')

sum = GPR[src1] + GPR[src2]
GPR[dst] = sign_extend(sum, from_nbits=32)
```

Exceptions: Reserved Instruction for ADDU[4X4] format on NMS cores.

ALIGN (Assembly alias)

Assembly: `ALIGN rd, rs, rt, bp`

Purpose: *Align*. Concatenate the 32 bit values in registers `$rt` and `$rs`, extract the word at specified byte position `bp`, and place the result in register `$rd`.

Availability: Assembly alias

Expansion:

```
bp != 0:
   EXTW rd, rs, rt, (4-bp)<<3

bp == 0:
   MOVE rd, rt
```

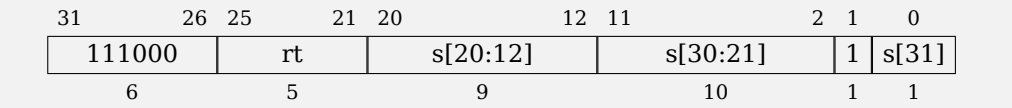
ALUIPC

Assembly: `ALUIPC rt, %pcrel_hi(address)`

Purpose: *Add aLigned Upper Immediate to PC*. Compute a 4KB aligned PC relative address by adding an upper 20 bit immediate value to NextPC, discarding the lower 12 bits, and placing the result in register `$rt`.

Availability: nanoMIPS

Format:




```
offset = sign_extend(s, from_nbits=32)
address = effective_address(CPU.next_pc, offset) & ~0xfff
```

Operation:

```
GPR[rt] = address
```

Exceptions: None.

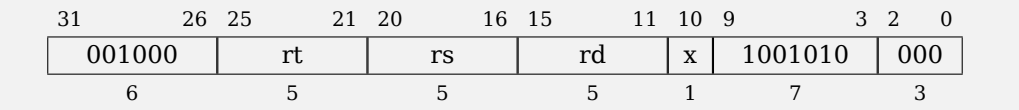
AND

Assembly: `AND rd, rs, rt`

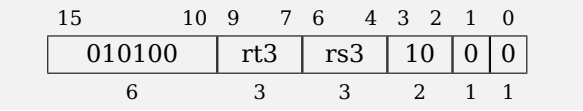
Purpose: *AND*. Compute logical AND of registers `$rs` and `$rt`, placing the result in register `$rd`.

Availability: nanoMIPS

Format: AND[32]



Format: AND[16]




```
rt = decode_gpr(rt3, 'gpr3')
rs = decode_gpr(rs3, 'gpr3')
rd = rt
```

Operation:

```
GPR[rd] = GPR[rs] & GPR[rt]
```

Exceptions: None.

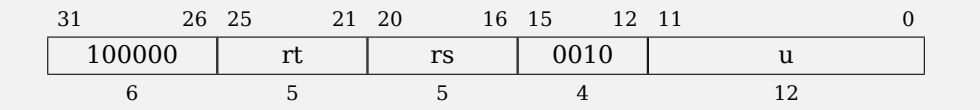
ANDI

Assembly: `ANDI rt, rs, u`

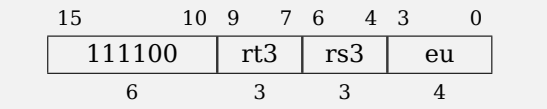
Purpose: *AND Immediate*. Compute logical AND of register `$rs` and immediate `u`, placing the result in register `$rt`.

Availability: nanoMIPS

Format: ANDI[32]



Format: ANDI[16]




```
rt = decode_gpr(rt3, 'gpr3')
rs = decode_gpr(rs3, 'gpr3')
u = (0x00ff if eu == 12 else
     0xffff if eu == 13 else
     eu)
```

Operation:

```
GPR[rt] = GPR[rs] & u
```

Exceptions: None.

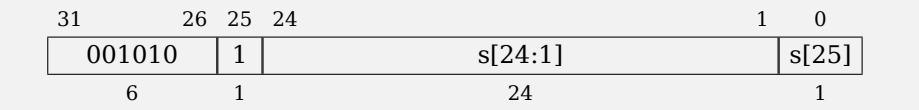
BALC

Assembly: `BALC address`

Purpose: *Branch And Link, Compact.* Unconditional PC relative branch to `address`, placing return address in register `$31`.

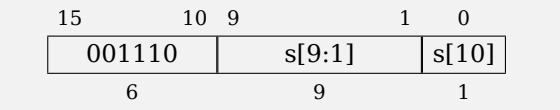
Availability: nanoMIPS

Format: BALC[32]




```
offset = sign_extend(s, from_nbits=26)
```

Format: BALC[16]




```
offset = sign_extend(s, from_nbits=11)
```

Operation:

```
address = effective_address(CPU.next_pc, offset)

GPR[31] = CPU.next_pc
CPU.next_pc = address
```

Exceptions: None.

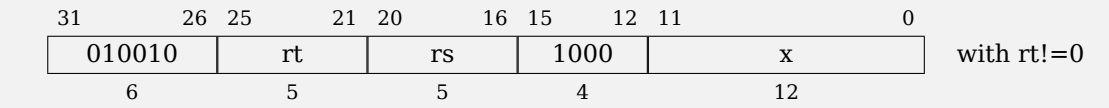
BALRSC

Assembly: `BALRSC rt, rs`

Purpose: *Branch And Link Register Scaled, Compact.* Unconditional branch to address `NextPC + 2*$rs`, placing return address in register `$rt`.

Availability: nanoMIPS

Format:



Operation:

```
address = effective_address(CPU.next_pc, offset=GPR[rs]<<1)

GPR[rt] = CPU.next_pc
CPU.next_pc = address
```

Exceptions: None.

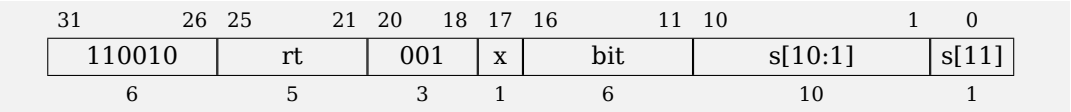
BBEQZC

Assembly: `BBEQZC rt, bit, address`

Purpose: *Branch if Bit Equals Zero, Compact.* PC relative branch to `address` if bit `bit` of register `$rt` is equal to zero.

Availability: nanoMIPS, not available in NMS

Format:




```
offset = sign_extend(s, from_nbits=12)
```

Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

if bit >= 32 and not Are64BitOperationsEnabled():
    raise exception('RI');

address = effective_address(CPU.next_pc, offset)

testbit = (GPR[rt] >> bit) & 1

if testbit == 0:
    CPU.next_pc = address
```

Exceptions: Reserved Instruction on NMS cores.

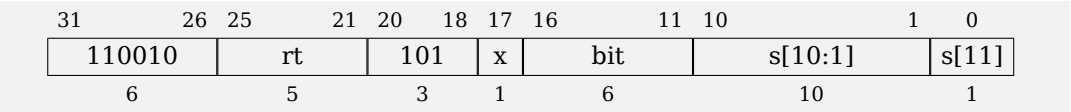
BBNEZC

Assembly: `BBNEZC rt, bit, address`

Purpose: *Branch if Bit Not Equal to Zero, Compact.* PC relative branch to `address` if bit `bit` of register `$rt` is not equal to zero.

Availability: nanoMIPS, not available in NMS

Format:




```
offset = sign_extend(s, from_nbits=12)
```

Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

if bit >= 32 and not Are64BitOperationsEnabled():
    raise exception('RI');

address = effective_address(CPU.next_pc, offset)

testbit = (GPR[rt] >> bit) & 1

if testbit == 1:
    CPU.next_pc = address
```

Exceptions: Reserved Instruction on NMS cores.

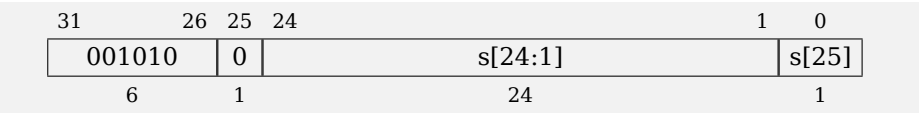
BC

Assembly: `BC address`

Purpose: *Branch, Compact*. Unconditional PC relative branch to `address`.

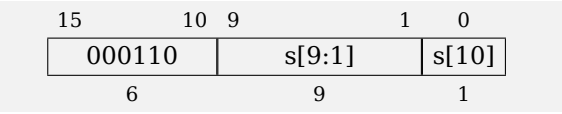
Availability: nanoMIPS

Format: BC[32]




```
offset = sign_extend(s, from_nbits=26)
address = effective_address(CPU.next_pc, offset)
```

Format: BC[16]




```
offset = sign_extend(s, from_nbits=11)
address = effective_address(CPU.next_pc, offset)
```

Operation:

```
CPU.next_pc = address
```

Exceptions: None.

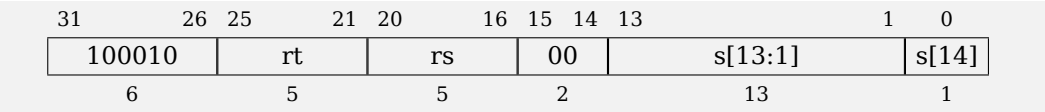
BEQC

Assembly: `BEQC rs, rt, address`

Purpose: *Branch if Equal, Compact.* PC relative branch to `address` if registers `$rs` and `$rt` are are equal.

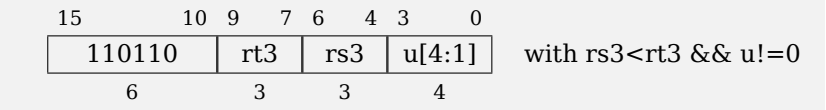
Availability: nanoMIPS, availability varies by format.

Format: BEQC[32]




```
offset = sign_extend(s, from_nbits=15)
address = effective_address(CPU.next_pc, offset)

not_in_mms = False
```

Format: BEQC[16], not available in NMS



with rs3<rt3 && u!=0

```
rs = decode_gpr(rs3, 'gpr3')
rt = decode_gpr(rt3, 'gpr3')

offset = u
address = effective_address(CPU.next_pc, offset)

not_in_mms = True
```

Operation:

```
if not_in_mms and C0.Config5.NMS == 1:
    raise exception('RI')

if GPR[rs] == GPR[rt]:
    CPU.next_pc = address
```

Exceptions: Reserved Instruction for BEQC[16] format on NMS cores.

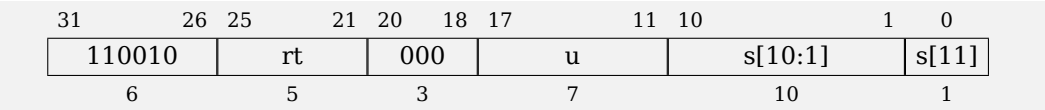
BEQIC

Assembly: `BEQIC rt, u, address`

Purpose: *Branch if Equal to Immediate, Compact.* PC relative branch to `address` if value of register `$rt` is equal to immediate value `u`.

Availability: nanoMIPS

Format:



Operation:

```
offset = sign_extend(s, from_nbits=12)
address = effective_address(CPU.next_pc, offset)

if GPR[rt] == u:
    CPU.next_pc = address
```

Exceptions: None.

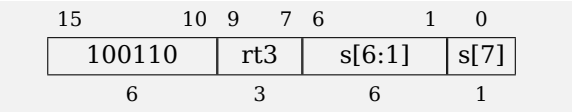
BEQZC

Assembly: `BEQZC rt, address # when rt and address are in range`

Purpose: *Branch if Equal to Zero, Compact.* PC relative branch to `address` if register `$rt` equals zero.

Availability: nanoMIPS

Format: BEQZC[16]



Operation:

```
rt = decode_gpr(rt3, 'gpr3')
offset = sign_extend(s, from_nbits=8)
address = effective_address(CPU.next_pc, offset)

if GPR[rt] == 0:
    CPU.next_pc = address
```

Exceptions: None.

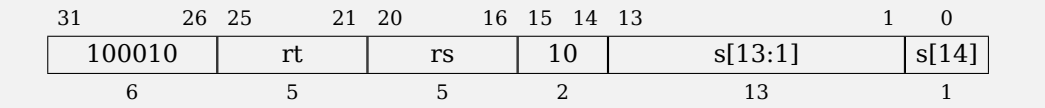
BGEC

Assembly: `BGEC rs, rt, address`

Purpose: *Branch if Greater than or Equal, Compact.* PC relative branch to `address` if register `$rs` is greater than or equal to register `$rt`.

Availability: nanoMIPS

Format:



Operation:

```
offset = sign_extend(s, from_nbits=15)
address = effective_address(CPU.next_pc, offset)

if GPR[rs] >= GPR[rt]:
    CPU.next_pc = address
```

Exceptions: None.

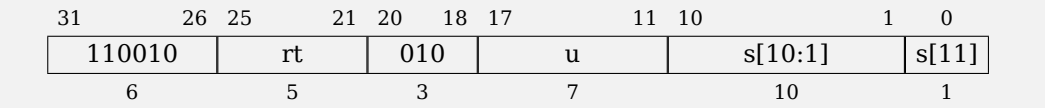
BGEIC

Assembly: `BGEIC rt, u, address`

Purpose: *Branch if Greater than or Equal to Immediate, Compact.* PC relative branch to `address` if signed register value `$rt` is greater than or equal to immediate u.

Availability: nanoMIPS

Format:



Operation:

```
offset = sign_extend(s, from_nbits=12)
address = effective_address(CPU.next_pc, offset)

if GPR[rt] >= u:
    CPU.next_pc = address
```

Exceptions: None.

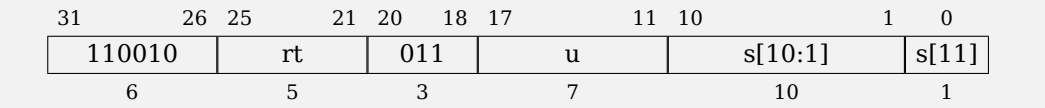
BGEIUC

Assembly: `BGEIUC rt, u, address`

Purpose: *Branch if Greater than or Equal to Immediate Unsigned, Compact*. PC relative branch to `address` if unsigned register `$rt` is greater than or equal to immediate `u`.

Availability: nanoMIPS

Format:



Operation:

```
offset = sign_extend(s, from_nbits=12)
address = effective_address(CPU.next_pc, offset)

if unsigned(GPR[rt]) >= u:
    CPU.next_pc = address
```

Exceptions: None.

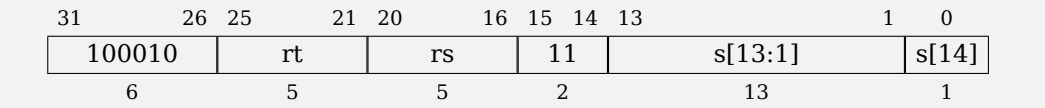
BGEUC

Assembly: `BGEUC rs, rt, address`

Purpose: *Branch if Greater than or Equal to Unsigned, Compact.* PC relative branch to `address` if unsigned register `$rs` is greater than or equal to unsigned register `$rt`.

Availability: nanoMIPS

Format:



Operation:

```
offset = sign_extend(s, from_nbits=15)
address = effective_address(CPU.next_pc, offset)

if unsigned(GPR[rs]) >= unsigned(GPR[rt]):
    CPU.next_pc = address
```

Exceptions: None.

BITREVB (Assembly alias)

Assembly: `BITREVB rt, rs`

Purpose: *Bit Reverse in Bytes*. Reverse bits in each byte of 32-bit value in register `$rs`, placing the result in register `$rt`.

Availability: Assembly alias, not available in NMS

Expansion:

```
ROTX rt, rs, 7, 8, 1
```

BITREVH (Assembly alias)

Assembly: `BITREVH rt, rs`

Purpose: *Bit Reverse in Halfs*. Reverse bits in each halfword of 32-bit value in register `$rs`, placing the result in register `$rt`.

Availability: Assembly alias, not available in NMS

Expansion:

```
ROTX rt, rs, 15, 16
```

BITREVW (Assembly alias)

Assembly: `BITREVW rt, rs`

Purpose: *Bit Reverse in Word*. Reverse all bits in 32 bit register `$rs`, placing the result in register `$rt`.

Availability: Assembly alias, not available in NMS

Expansion:

```
ROTX rt, rs, 31, 0
```

BITSWAP (Assembly alias)

Assembly: `BITSWAP rt, rs`

Purpose: *Bitswap.* Reverse bits in each byte of 32-bit value in register `$rs`, placing the result in register `$rt`.

Availability: Assembly alias, not available in NMS

Expansion:

```
ROTX rt, rs, 7, 8, 1
```

The assembly alias BITSWAP is provided for compatibility with MIPS32™. Its behavior is equivalent to the new assembly alias BITREVB, whose name is chosen to fit consistently with the naming of other reversing instructions in nanoMIPS™.

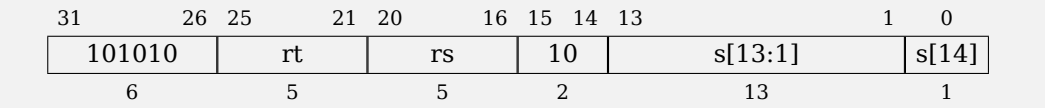
BLTC

Assembly: `BLTC rs, rt, address`

Purpose: *Branch if Less Than, Compact.* PC relative branch to `address` if signed register `$rs` is less than signed register `$rt`.

Availability: nanoMIPS

Format:



Operation:

```
offset = sign_extend(s, from_nbits=15)
address = effective_address(CPU.next_pc, offset)

if GPR[rs] < GPR[rt]:
    CPU.next_pc = address
```

Exceptions: None.

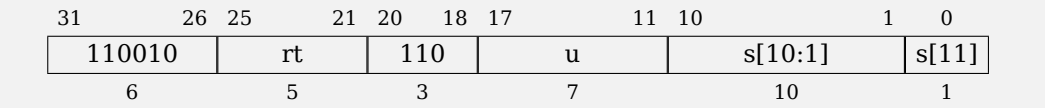
BLTIC

Assembly: `BLTIC rt, u, address`

Purpose: *Branch if Less Than Immediate, Compact.* PC relative branch to `address` if signed register `$rt` is less than immediate `u`.

Availability: nanoMIPS

Format:



Operation:

```
offset = sign_extend(s, from_nbits=12)
address = effective_address(CPU.next_pc, offset)

if GPR[rt] < u:
    CPU.next_pc = address
```

Exceptions: None.

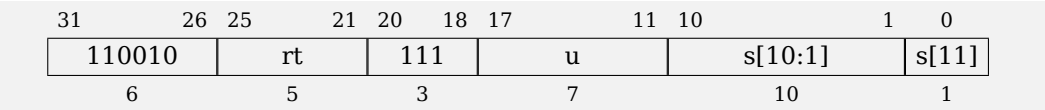
BLTIUC

Assembly: `BLTIUC rt, u, address`

Purpose: *Branch if Less Than Immediate Unsigned Compact*. PC relative branch to `address` if unsigned register `$rt` is less than immediate `u`.

Availability: nanoMIPS

Format:



Operation:

```
offset = sign_extend(s, from_nbits=12)
address = effective_address(CPU.next_pc, offset)

if unsigned(GPR[rt]) < u:
    CPU.next_pc = address
```

Exceptions: None.

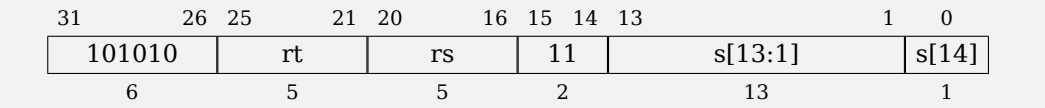
BLTUC

Assembly: `BLTUC rs, rt, address`

Purpose: *Branch if Less Than Unsigned, Compact.* PC relative branch to `address` if unsigned register `$rs` is less than unsigned register `$rt`.

Availability: nanoMIPS

Format:



Operation:

```
offset = sign_extend(s, from_nbits=15)
address = effective_address(CPU.next_pc, offset)

if unsigned(GPR[rs]) < unsigned(GPR[rt]):
    CPU.next_pc = address
```

Exceptions: None.

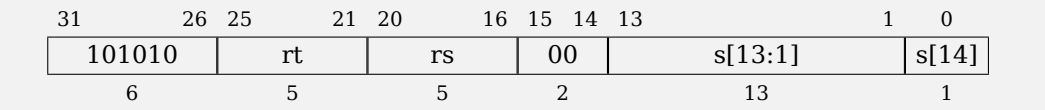
BNEC

Assembly: `BNEC rs, rt, address`

Purpose: *Branch Not Equal, Compact*. PC relative branch to `address` if register `$rs` is not equal to register `$rt`.

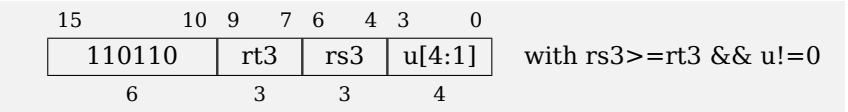
Availability: nanoMIPS, availability varies by format.

Format: BNEC[32]




```
offset = sign_extend(s, from_nbits=15)
```

Format: BNEC[16], not available in NMS



with rs3>=rt3 && u!=0

```
if C0.Config5.NMS == 1:
    raise exception('RI')

rs = decode_gpr(rs3, 'gpr3')
rt = decode_gpr(rt3, 'gpr3')
offset = u
```

Operation:

```
address = effective_address(CPU.next_pc, offset)

if GPR[rs] != GPR[rt]:
    CPU.next_pc = address
```

Exceptions: Reserved Instruction for BNEC[16] format on NMS cores.

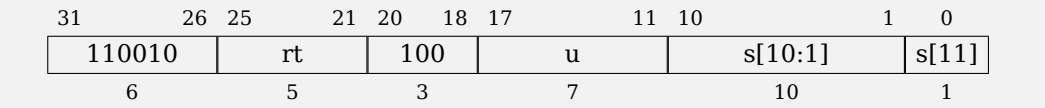
BNEIC

Assembly: `BNEIC rt, u, address`

Purpose: *Branch if Not Equal to Immediate, Compact.* PC relative branch to address if register `$rt` is not equal to immediate `u`.

Availability: nanoMIPS

Format:



Operation:

```
offset = sign_extend(s, from_nbits=12)
address = effective_address(CPU.next_pc, offset)

if GPR[rt] != u:
    CPU.next_pc = address
```

Exceptions: None.

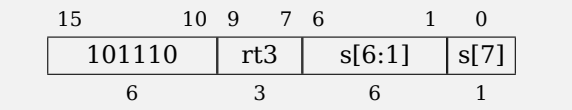
BNEZC

Assembly: `BNEZC rt, address`

Purpose: *Branch if Not Equal to Zero, Compact.* PC relative branch to `address` if register `$rt` is not equal to zero.

Availability: nanoMIPS

Format: BNEZC[16]




```
rt = decode_gpr(rt3, 'gpr3')
offset = sign_extend(s, from_nbits=8)
```

Operation:

```
address = effective_address(CPU.next_pc, offset)

if GPR[rt] != 0:
    CPU.next_pc = address
```

Exceptions: None.

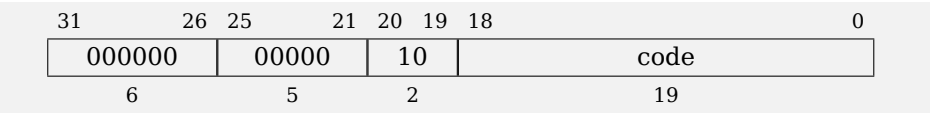
BREAK

Assembly: `BREAK code`

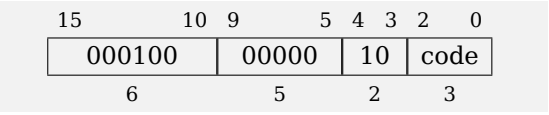
Purpose: *Break*. Cause a Breakpoint exception.

Availability: nanoMIPS

Format: BREAK[32]



Format: BREAK[16]



Operation:

```
raise exception('BP')
```

Exceptions: Breakpoint.

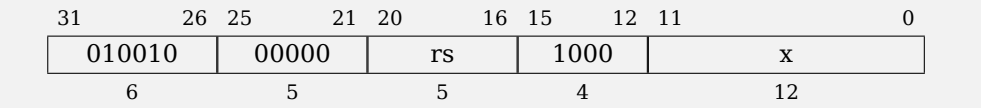
BRSC

Assembly: `BRSC rs`

Purpose: *Branch Register Scaled, Compact.* Unconditional branch to address `NextPC + 2*$rs`.

Availability: nanoMIPS

Format:



Operation:

```
address = effective_address(CPU.next_pc, offset=GPR[rs]<<1)

CPU.next_pc = address
```

Exceptions: None.

BYTEREVH (Assembly alias)

Assembly: `BYTEREVH rt, rs`

Purpose: *Byte Reverse in Halfs*. Reverse bytes in each halfword of 32-bit value in register `$rs`, placing the result in register `$rt`.

Availability: Assembly alias, not available in NMS

Expansion:

```
ROTX rt, rs, 8, 24
```

BYTEREVW (Assembly alias)

Assembly: `BYTEREVW rt, rs`

Purpose: *Byte Reverse in Word*. Reverse each byte in word value in register `$rs`, placing the result in register `$rt`.

Availability: Assembly alias, not available in NMS

Expansion:

```
ROTX rt, rs, 24, 8
```

CACHE/CACHEE

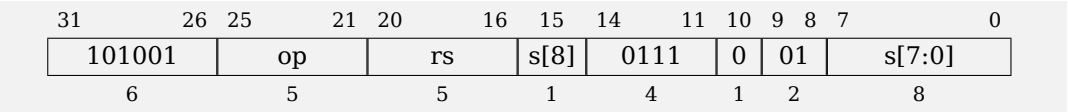
Assembly:

```
CACHE  op, offset(rs)
CACHEE op, offset(rs)
```

Purpose: *Cache operation/Cache operation using EVA addressing.* Perform cache operation of type `op` at address `$rs +offset` (register plus immediate). For CACHEE, translate the virtual address as though the core is in user mode, although it is actually in kernel mode.

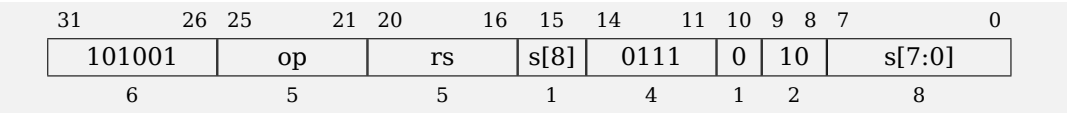
Availability: nanoMIPS. Requires CP0 privilege, availability varies by format.

Format: CACHE




```
offset = sign_extend(s, from_nbits=9)
is_eva = False
```

Format: CACHEE, present when Config5.EVA=1.




```
offset = sign_extend(s, from_nbits=9)
is_eva = True
```

Operation:

```
# NMS core without caches gives RI (not Coprocessor Unusable) exception.
if (C0.Config5.NMS and C0.Config1.DL == 0 and C0.Config1.IL == 0
                   and C0.Config2.SL == 0 and C0.Config2.TL == 0
                   and C0.Config5.L2C == 0):
    raise exception('RI')

if is_eva and not C0.Config5.EVA:
    raise exception('RI')

if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

va = effective_address(GPR[rs], offset, 'Load', eva=is_eva)
```

```
23
24  # Behavior for index cacheops is unpredictable if address is not unmapped.
25  if op <= 11:  # Index cacheop
26      translation_type, description, result_args = decode_va(va, eva=is_eva)
27      if translation_type != 'unmapped':
28          raise UNPREDICTABLE('Index cacheop unpredictable with VA not unmapped')
29
30  pa, cca = va2pa(va, 'Cacheop', eva=is_eva)
31
32  if cca == 2 or cca == 7:
33      if C0.Config.AT >= 2:
34          pass  # Cacheop to uncached address is a nop in R6
35      else:
36          raise UNPREDICTABLE('Cacheop to uncached address is unpredictable')
37
38  else:
39      cacheop(va, pa, op)
```

The CACHE/CACHEE instructions perform the cache operation specified by argument 'op' on the register plus immediate address `$rs +offset`. For CACHEE, the virtual address is translated as though the core is in user mode, although it is actually in kernel mode.

The 'op' argument is a 5 bit value specifying one of the following the possible cache operations, which are described in more detail below:

'op'	Operation	Availability
0	ICache Index Invalidate	Required (if ICache present)
1	DCache Index Writeback Invalidate	Required (if DCache present)
2	TCache Index Writeback Invalidate	Required (if TCache present)
3	SCache Index Writeback Invalidate	Required (if SCache present)
4	ICache Index Load Tag	Recommended (if ICache present)
5	DCache Index Load Tag	Recommended (if DCache present)
6	TCache Index Load Tag	Recommended (if TCache present)
7	SCache Index Load Tag	Recommended (if SCache present)
8	ICache Index Store Tag	Required (if ICache present)
9	DCache Index Store Tag	Required (if DCache present)
10	TCache Index Store Tag	Required (if TCache present)
11	SCache Index Store Tag	Required (if SCache present)
12	ICache Implementation Dependent Op	Optional (if ICache present)
13	DCache Implementation Dependent Op	Optional (if DCache present)
14	TCache Implementation Dependent Op	Optional (if TCache present)
15	SCache Implementation Dependent Op	Optional (if SCache present)
16	ICache Hit Invalidate	Required (if ICache present)
17	DCache Hit Invalidate	Optional (if DCache present)
18	TCache Hit Invalidate	Optional (if TCache present)
19	SCache Hit Invalidate	Optional (if SCache present)
20	ICache Fill	Recommended (if ICache present)
21	DCache Hit Writeback Invalidate	Recommended (if DCache present)

'op'	Operation	Availability
22	TCache Hit Writeback Invalidate	Recommended (if TCache present)
23	SCache Hit Writeback Invalidate	Recommended (if SCache present)
24	Unused	
25	DCache Hit Writeback	Recommended (if DCache present)
26	TCache Hit Writeback	Recommended (if TCache present)
27	SCache Hit Writeback	Recommended (if SCache present)
28	ICache Fetch and Lock	Recommended (if ICache present)
29	DCache Fetch and Lock	Recommended (if DCache present)
30	Unused	
31	Unused	

Index cacheops (those with `op <= 11` and optionally the implementation dependent cases `12 <= op <= 15`) are operations where the input address is treated as an index into the target cache array. The rules for constructing the index are given in the cacheop() function pseudocode.

'Hit' cacheops are operations where the input address is treated as a virtual memory address. The operation will target the cache line containing data for that virtual address, if it is present in the cache.

The operations listed above behave as follows:

- *ICache Index Invalidate* (op=0): Set the state of the instruction cache line at the specified index to invalid.
- *D/T/S Cache Index Writeback Invalidate* (op=1,2,3): If the cache line at the specified index is valid and dirty, write the line back to the memory address specified by the cache tag. Whether or not the line was dirty, set the state of the cache line to invalid. For a write-through cache, the writeback step is not required and this is effectively a Cache Index Invalidate operation. This cache operation is required and may be used by software to invalidate the entire data cache by stepping through all indices. Note that the Index Store Tag operation must be used to initialize the cache at power up.
- *I/D/T/S Cache Index Load Tag* (op=4,5,6,7): Read the tag for the cache line at the specified index into the TagLo and TagHi Coprocessor 0 registers. If the DataLo and DataHi registers are implemented, also read the data corresponding to the byte index into the DataLo and DataHi registers. This operation must not cause a Cache Error Exception. The granularity and alignment of the data read into the DataLo and DataHi registers is implementation-dependent, but is typically the result of an aligned access to the cache, ignoring the appropriate low-order bits of the byte index.
- *I/D/T/S Cache Index Store Tag* (op=8,9,10,11): Write the tag for the cache block at the specified index from the TagLo and TagHi Coprocessor 0 registers. This operation must not cause a Cache Error Exception. This required encoding may be used by software to initialize the entire instruction or data caches by stepping through all valid indices. Doing so requires that the TagLo and TagHi registers associated with the cache be initialized to zero first.
- *I/D/T/S Cache Implementation Dependent Op* (op=12,13,14,15): Available for implementation dependent operation.

- *I/D/T/S Cache Hit Invalidate* (op=16,17,18,19): If the cache block contains the specified address, set the state of the cache block to invalid. This required encoding may be used by software to invalidate a range of addresses from the instruction cache by stepping through the address range by the line size of the cache. In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.
- *ICache FIll* (op=20): Fill the cache from the specified virtual address.
- *D/T/S Hit Writeback Invalidate* (op=21,22,23): For the cache line (if any) which contains the specified address: if the cache line is valid and dirty, write the line back to the memory address specified by the cache tag. Whether or not the line was dirty, set the state of the cache line to invalid. For a write-through cache, the writeback step is not required and this is effectively a Cache Hit Invalidate operation. This cache operation is required and may be used by software to invalidate a range of addresses from the data cache by stepping through the address range by the line size of the cache. In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.
- *D/T/S Hit Writeback* (op=25,26,27): If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After the operation is completed, leave the state of the line valid, but clear the dirty state. For a write-through cache, this operation may be treated as a nop. In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.
- *I/D Fetch and Lock* (op=28,29): If the cache does not contain the specified virtual address, fill it from memory, performing a write-back if required. Set the state to valid and locked. The way selected on a fill from memory is implementation dependent. The lock state may be cleared by executing an Index Invalidate, Index Writeback Invalidate, Hit Invalidate, or Hit Writeback Invalidate operation to the locked line, or via an Index Store Tag operation to the line that clears the lock bit. It is implementation dependent whether a locked line is displaced as the result of an external invalidate or intervention that hits on the locked line. Software must not depend on the locked line remaining in the cache if an external invalidate or intervention would invalidate the line if it were not locked. It is implementation dependent whether a Fetch and Lock operation affects more than one line. For example, more than one line around the referenced address may be fetched and locked. It is recommended that only the single line containing the referenced address be affected.

It is implementation dependent whether the input address for an Index cacheop is converted into a physical address by the MMU, so to avoid the posibility of generating a TLB exception, the index value should always be converted to an unmapped address (such as a kseg0 address by ORing the index with 0x80000000) before being used by the cache instruction. For example, the following code sequence performs a data cache Index Store Tag operation using the index passed in GPR a0:

```
li      a1, 0x80000000       /* Base of kseg0 segment */
or      a0, a0, a1           /* Convert index to kseg0 address */
cache   DCIndexStTag, 0(a1)  /* Perform the index store tag operation */
```

Some CACHE/CACHEE operations may result in a Cache Error exception. For example, if a Writeback operation detects a cache or bus error during the processing of the operation, that error is reported

via a Cache Error exception. Also, a Bus Error Exception may occur if a bus operation invoked by this instruction is terminated in an error. However, cache error exceptions must not be triggered by an Index Load Tag or Index Store tag operation, as these operations are used for initialization and diagnostic purposes.

It is implementation dependent whether a data watch is triggered by a cache instruction whose address matches the Watch register address match conditions. The preferred implementation is not to match on the CACHE/CACHEE instructions.

The operation of the instruction is UNPREDICTABLE if the cache line that contains the CACHE instruction is the target of an invalidate or a writeback invalidate operation.

If this instruction is used to lock all ways of a cache at a specific cache index, the behavior of that cache to subsequent cache misses to that cache index is UNDEFINED.

The effective address may be arbitrarily aligned. The CACHE/CACHEE instructions never causes an Address Error Exception due to a non-aligned address.

The CACHE instruction and the memory transactions which are sourced by the CACHE instruction, such as cache refill or cache writeback, obey the ordering and completion rules of the SYNC instruction. Any use of this instruction that can cause cacheline writebacks should be followed by a subsequent SYNC instruction to avoid hazards where the writeback data is not yet visible at the next level of the memory hierarchy.

For multiprocessor implementations that maintain coherent caches, some of the Hit type operations may optionally affect all coherent caches within the implementation. In this case, if the effective address uses a coherent Cache Coherency Attribute (CCA), then the operation is globalized, meaning it is broadcast to all of the coherent caches within the system. If the effective address does not use one of the coherent CCAs, there is no broadcast of the operation. If multiple levels of caches are to be affected by one CACHE instruction, all of the affected cache levels must be processed in the same manner - either all affected cache levels use the globalized behavior or all affected cache levels use the non-globalized behavior.

Exceptions:

Address Error. Bus Error. Cache Error. Coprocessor Unusable. Reserved Instruction on NMS cores without caches. Reserved Instruction for CACHEE if EVA not implemented. TLB Invalid. TLB Refill.

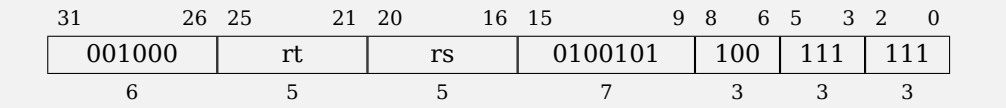
CLO

Assembly: `CLO rt, rs`

Purpose: *Count Leading Ones*. Count leading ones in 32-bit register value `$rs`, placing the result in register `$rt`.

Availability: nanoMIPS, not available in NMS

Format:



Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

input = GPR[rs]

i = 0
while i < 32:
    if input[31 - i] != 1: break
    i += 1

GPR[rt] = i
```

Exceptions: Reserved Instruction on NMS cores.

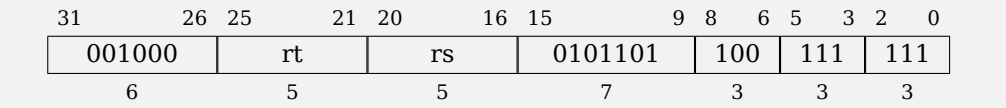
CLZ

Assembly: `CLZ rt, rs`

Purpose: *Count Leading Zeros.* Count leading zeros in 32-bit register value `$rs`, placing the result in register `$rt`.

Availability: nanoMIPS, not available in NMS

Format:



Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

input = GPR[rs]

i = 0
while i < 32:
    if input[31 - i] != 0: break
    i += 1

GPR[rt] = i
```

Exceptions: Reserved Instruction on NMS cores.

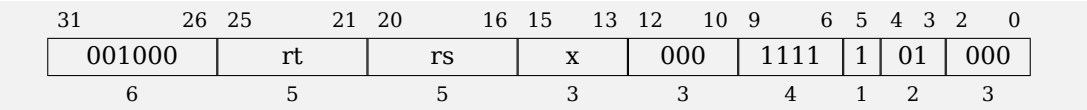
CRC32B

Assembly: `CRC32B rt, rs`

Purpose: *CRC32 Byte*. Generate a 32-bit CRC value `$rt` based on the reversed polynomial 0xEDB88320, using cumulative 32-bit CRC value `$rt` and right-justified byte-sized message `$rs` as inputs.

Availability: nanoMIPS. Optional, present when `Config5.CRCP=1`.

Format:



Operation:

```
if C0.Config5.CRCP == 0:
    raise exception('RI')

result = crc32(value=GPR[rt], message=GPR[rs], nbits=8, poly=0xEDB88320)

GPR[rt] = sign_extend(result, from_nbits=32)
```

Exceptions: Reserved Instruction on cores without CRC support.

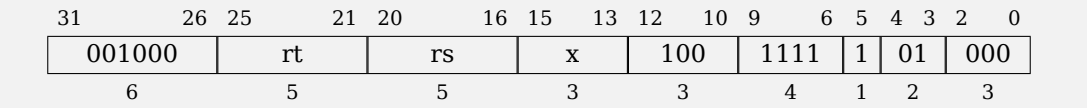
CRC32CB

Assembly: `CRC32CB rt, rs`

Purpose: *CRC32 (Castagnoli) Byte*. Generate a 32-bit CRC value `$rt` based on the reversed polynomial 0x82F63B78, using cumulative 32-bit CRC value `$rt` and right-justified byte-sized message `$rs` as inputs.

Availability: nanoMIPS. Optional, present when `Config5.CRCP=1`.

Format:



Operation:

```
if C0.Config5.CRCP == 0:
    raise exception('RI')

result = crc32(value=GPR[rt], message=GPR[rs], nbits=8, poly=0x82F63B78)

GPR[rt] = sign_extend(result, from_nbits=32)
```

Exceptions: Reserved Instruction on cores without CRC support.

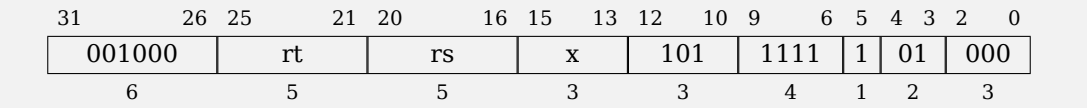
CRC32CH

Assembly: `CRC32CH rt, rs`

Purpose: *CRC32 (Castagnoli) Half*. Generate a 32-bit CRC value `$rt` based on the reversed polynomial 0x82F63B78, using cumulative 32-bit CRC value `$rt` and right-justified halfword-sized message `$rs` as inputs.

Availability: nanoMIPS. Optional, present when `Config5.CRCP=1`.

Format:



Operation:

```
if C0.Config5.CRCP == 0:
    raise exception('RI')

result = crc32(value=GPR[rt], message=GPR[rs], nbits=16, poly=0x82F63B78)

GPR[rt] = sign_extend(result, from_nbits=32)
```

Exceptions: Reserved Instruction on cores without CRC support.

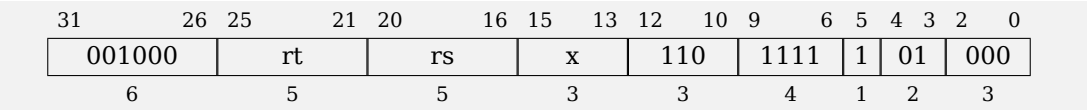
CRC32CW

Assembly: `CRC32CW rt, rs`

Purpose: *CRC32 (Castagnoli) Word*. Generate a 32-bit CRC value `$rt` based on the reversed polynomial 0x82F63B78, using cumulative 32-bit CRC value `$rt` and right-justified word-sized message `$rs` as inputs.

Availability: nanoMIPS. Optional, present when `Config5.CRCP=1`.

Format:



Operation:

```
if C0.Config5.CRCP == 0:
    raise exception('RI')

result = crc32(value=GPR[rt], message=GPR[rs], nbits=32, poly=0x82F63B78)

GPR[rt] = sign_extend(result, from_nbits=32)
```

Exceptions: Reserved Instruction on cores without CRC support.

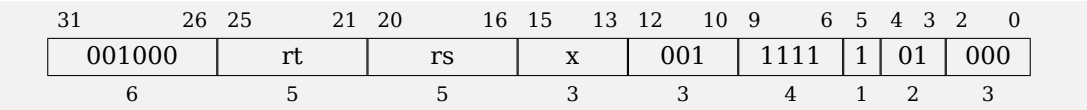
CRC32H

Assembly: `CRC32H rt, rs`

Purpose: *CRC32 Half*. Generate a 32-bit CRC value `$rt` based on the reversed polynomial 0xEDB88320, using cumulative 32-bit CRC value `$rt` and right-justified halfword-sized message `$rs` as inputs.

Availability: nanoMIPS. Optional, present when `Config5.CRCP=1`.

Format:



Operation:

```
if C0.Config5.CRCP == 0:
    raise exception('RI')

result = crc32(value=GPR[rt], message=GPR[rs], nbits=16, poly=0xEDB88320)

GPR[rt] = sign_extend(result, from_nbits=32)
```

Exceptions: Reserved Instruction on cores without CRC support.

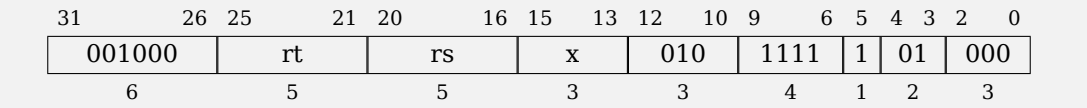
CRC32W

Assembly: `CRC32W rt, rs`

Purpose: *CRC32 Word*. Generate a 32-bit CRC value `$rt` based on the reversed polynomial 0xEDB88320, using cumulative 32-bit CRC value `$rt` and right-justified word-sized message `$rs` as inputs.

Availability: nanoMIPS. Optional, present when `Config5.CRCP=1`.

Format:



Operation:

```
if C0.Config5.CRCP == 0:
    raise exception('RI')

result = crc32(value=GPR[rt], message=GPR[rs], nbits=32, poly=0xEDB88320)

GPR[rt] = sign_extend(result, from_nbits=32)
```

Exceptions: Reserved Instruction on cores without CRC support.

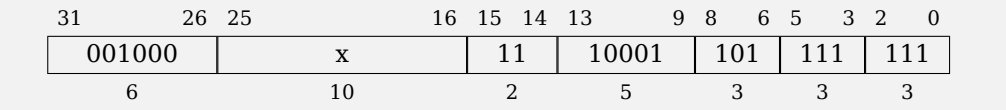
DERET

Assembly: `DERET`

Purpose: *Debug Exception Return*. Return from a debug exception by jumping to the address in the DEPC register, and clearing Debug.DM.

Availability: nanoMIPS. Optional, present when Debug implemented.

Format:



Operation:

```
if C0.Config1.EP == 0:
    raise exception('RI')

if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

if C0.Debug.DM == 0:
    raise exception('RI')

CPU.next_pc = sign_extend(Root.C0.DEPC)
C0.Debug.DM = 0

# If single stepping, forward progress is allowed on the next instruction.
CPU.debug_sst_progress_allowed = True

clear_execution_hazards()
clear_instruction_hazards()
```

The DERET instruction implements a software barrier that resolves all execution and instruction hazards. See the EHB and JALRC.HB instructions for an explanation of execution and instruction hazards respectively, and also the SYNCI/SYNCIE instruction for additional information on resolving instruction hazards created by writing to the instruction stream.

The effects of the DERET barrier are seen starting with the fetch and decode of the instruction at the PC to which the DERET returns. This means, for instance, that if C0.DEPC is modified by an MTC0 instruction prior to a DERET, an EHB is required between the MTC0 and the DERET to ensure that the DERET uses the correct DEPC value.

The DERET instruction is only legal in debug mode and will give a Coprocessor Unusable exception when executed in user mode or a Reserved Instruction exception when executed in kernel mode.

Exceptions:

Coprocessor Unusable. Reserved Instruction when not in Debug Mode or on cores without Debug support.

DI

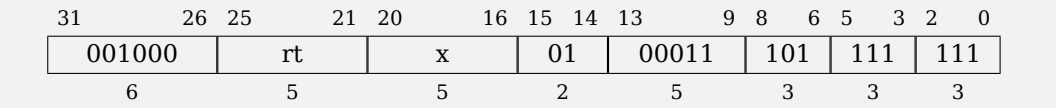
Assembly:

```
DI rt
DI    # rt=0 implied
```

Purpose: *Disable Interrupts*. Disable interrupts by setting Status.IE to 0, and return the previous value of Status register in register `$rt`.

Availability: nanoMIPS. Requires CP0 privilege.

Format:



Operation:

```
if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

GPR[rt] = C0.Status
C0.Status.IE = 0
```

Exceptions: Coprocessor Unusable.

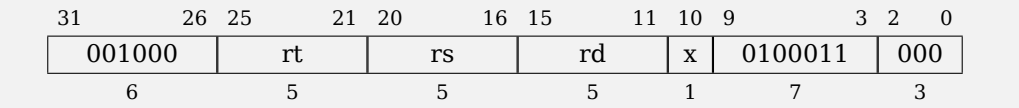
DIV

Assembly: `DIV rd, rs, rt`

Purpose: *Divide*. Divide signed word `$rs` by signed word `$rt` and place the result in `$rd`.

Availability: nanoMIPS

Format:



Operation:

```
numerator = GPR[rs]
denominator = GPR[rt]

if denominator == 0:
    quotient, remainder = (UNKNOWN, UNKNOWN)

else:
    quotient, remainder = divide_integers(numerator, denominator)

GPR[rd] = sign_extend(quotient, from_nbits=32)
```

Exceptions: None.

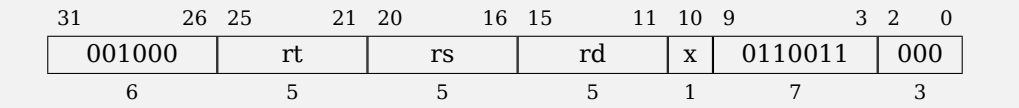
DIVU

Assembly: `DIVU rd, rs, rt`

Purpose: *Divide Unsigned*. Divide unsigned word `$rs` by unsigned word `$rt` and place the result in register `$rd`.

Availability: nanoMIPS

Format:



Operation:

```
numerator = zero_extend(GPR[rs], from_nbits=32)
denominator = zero_extend(GPR[rt], from_nbits=32)

if denominator == 0:
    quotient, remainder = (UNKNOWN, UNKNOWN)

else:
    quotient, remainder = divide_integers(numerator, denominator)

GPR[rd] = sign_extend(quotient, from_nbits=32)
```

Exceptions: None.

DVP

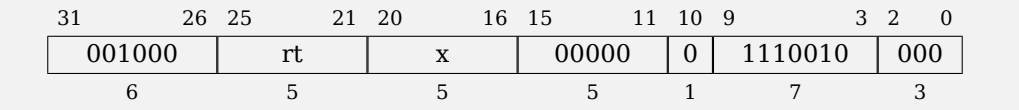
Assembly:

```
DVP rt
DVP    # rt=0 implied
```

Purpose: *Disable Virtual Processors*. Disable all virtual processors in a physical core other than the one that issued the instruction. Set `VPControl.DIS` to 1, and place the previous value of the `VPControl` CP0 register in register `$rt`.

Availability: nanoMIPS. Optional, present when `Config5.VP=1`, otherwise NOP. Requires CP0 privilege.

Format:



Operation:

```
if C0.Config5.VP == 0:
    # No operation when VP not implemented
    pass
else:
    if not IsCoprocessor0Enabled():
        raise coprocessor_exception(0)

    GPR[rt] = C0.VPControl
    C0.VPControl.DIS = 1

    disable_virtual_processors()
```

The DVP instruction is used to halt instruction fetch for all virtual processors in a VP core, other than the one which issued the DVP instruction. Possible uses for DVP include:

- Performing cache operations where the cache state must not be affected by the actions of other threads on the same core.
- Reprogramming virtual processor scheduling priority.

All outstanding instructions for the affected virtual processors must be complete before the DVP itself is allowed to retire. Any outstanding events such as hardware instruction or data prefetch, or page-table walks, must also be terminated.

Memory ordering equivalent to that provided by SYNC(stype=0) is guaranteed between subsequent instructions on the virtual processor which issued the DVP, and instructions which have already graduated on the disabled virtual processors.

If a virtual processor is already disabled by another event, for instance, if it has executed a WAIT or a PAUSE instruction or has been halted by some external hardware event, then the disabled virtual processor will not be re-enabled until both an EVP instruction has been executed on the controlling thread, and an event which would otherwise have woken the virtual processor (such as an interrupt for a WAIT instruction or an interrupt or clearing of the LLBit for a PAUSE instruction) has also occurred.

The effect of a DVP instruction is undone by an EVP instruction, which causes execution to resume immediately (where applicable) on all other virtual processors. From the perspective of the disabled virtual processors, after the EVP, execution continues as though the DVP had not occurred.

If an event occurs in between the DVP and EVP that renders state of a disabled virtual processor UNPREDICTABLE (such as power-gating), then the effect of EVP is UNPREDICTABLE.

A disabled virtual processor cannot be woken by an interrupt or a deferred exception, at least until execution is re-enabled by an EVP instruction on the controlling thread. The virtual processor that executes the DVP, however, continues to be interruptible.

A DVP which is executed when VPControl.DIS=1 will return the current value of the VPControl register but otherwise will leave the other virtual processors in a disabled state. Software should only re-enable virtual processors (via the EVP instruction) if it has verified from the VPControl value returned by the DVP that virtual processors were previously enabled. Performing this check allows DVP/EVP pairs to be safely nested.

In a core with multiple virtual processors, more than one virtual processor may execute a DVP simultaneously. The implementation should ensure that the selection of which virtual processor's DVP successfully graduates is not biased towards any one virtual processor, in order to prevent the possibility of live-lock.

The DVP instruction behaves like a NOP on cores which do not implement virtual processors (i.e. when Config5.VP=0). This behavior allows kernel code to enclose critical sequences within DVP/EVP blocks without first checking whether it is running on a VP core. The encoding of the DVP instruction is equivalent to a SLTU instruction targeting $0, i.e. a NOP, which leads to the correct behavior on non-VP cores with no additional hardware special casing.

Exceptions: Coprocessor Unusable.

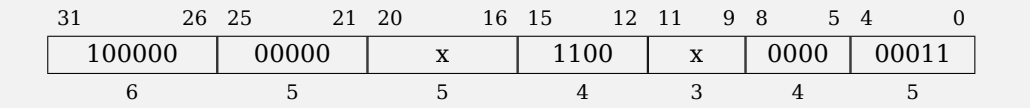
EHB

Assembly: `EHB`

Purpose: *Execution hazard barrier*. Clear all execution hazards before allowing any subsequent instructions to graduate.

Availability: nanoMIPS

Format:



Operation:

```
clear_execution_hazards()
```

The EHB instruction creates an execution hazard barrier, meaning that it ensures that subsequent instructions will be aware of changes to CP0 state caused by prior instructions. Examples of instructions which change CP0 state and which need an execution hazard barrier to ensure that subsequent instructions see those updates are MTC0, EI, DI, TLBR and CACHE/CACHEE

In the absence of an execution hazard barrier, the CP0 register value used as input to an instruction may be out of date, since it may have been read before the write to the CP0 register by a prior instruction has actually been committed.

An execution hazard barrier is sufficient to ensure that a fetched instruction is aware of all prior CP0 updates. However, it is not sufficient to ensure that the correct instruction is being fetched as a result of those CP0 updates. Ensuring that the correct instruction is fetched requires an instruction hazard barrier, which is provided by the JALRC.HB instruction, or any of the exception return instructions ERET/ERETNC or DERET.

Exceptions: None.

EI

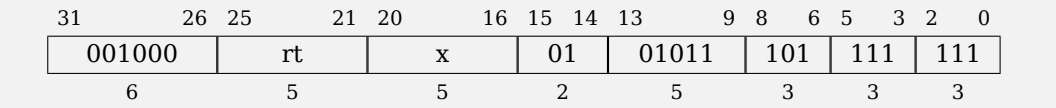
Assembly:

```
EI rt
EI    # rt=0 implied
```

Purpose: *Enable Interrupts*. Enable interrupts by setting `Status.IE` to 1, and return the previous value of `Status` register in register `$rt`.

Availability: nanoMIPS. Requires CP0 privilege.

Format:



Operation:

```
if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

GPR[rt] = C0.Status
C0.Status.IE = 1
```

Exceptions: Coprocessor Unusable.

ERET/ERETNC

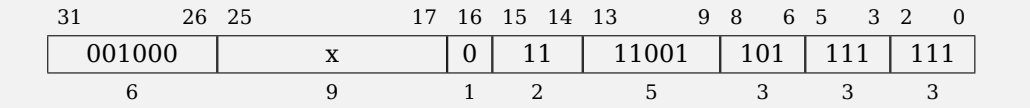
Assembly:

```
ERET
ERETNC
```

Purpose: *Exception Return/Exception Return Not Clearing LLBit*. Return from an exception: either by clearing `Status.ERL` if set and jumping to the address in `ErrorEPC`; otherwise by clearing Status.EXL, jumping to the address in EPC, and updating the current Shadow Register Set to SRSCtl.PSS if required.

Availability: nanoMIPS, availability varies by format.

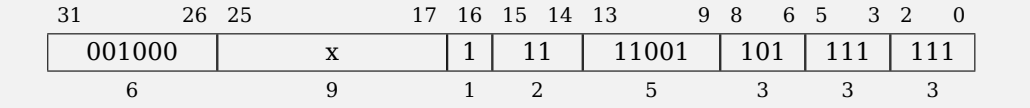
Format: ERET, requires CP0 privilege.



31–26	25–17	16	15–14	13–9	8–6	5–3	2–0
001000	x	0	11	11001	101	111	111
6	9	1	2	5	3	3	3

```
nc = False
```

Format: ERETNC, present when `Config5.LLB=1`. Requires CP0 privilege.



31–26	25–17	16	15–14	13–9	8–6	5–3	2–0
001000	x	1	11	11001	101	111	111
6	9	1	2	5	3	3	3

```
nc = True
```

Operation:

```
if nc and C0.Config5.LLB == 0:
    raise exception('RI')

if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

if C0.Status.ERL == 1:
    effective_epc = sign_extend(C0.ErrorEPC)
    C0.Status.ERL = 0
else:
    effective_epc = sign_extend(C0.EPC)
    C0.Status.EXL = 0

    if C0.SRSCtl.HSS > 0 and C0.Status.BEV == 0:
```

```
24          C0.SRSCtl.CSS = C0.SRSCtl.PSS
25
26  CPU.next_pc = effective_epc
27
28  # clear LLbit unless this is an ERETNC
29  if not nc:
30     C0.LLAddr.LLB = 0
31
32  clear_execution_hazards()
33  clear_instruction_hazards()
```

The ERET/ERETNC instructions implement a software barrier that resolves all execution and instruction hazards. See the EHB and JALRC.HB instructions for an explanation of execution and instruction hazards respectively, and also the SYNCI/SYNCIE instruction for additional information on resolving instruction hazards created by writing to the instruction stream.

The effects of the ERET/ERETNC barrier are seen starting with the fetch and decode of the instruction at the PC to which the ERET returns. This means, for instance, that if C0.EPC is modified by an MTC0 instruction prior to an ERET, an EHB is required between the MTC0 and the ERET to ensure that the ERET uses the correct EPC value.

Config5.LLB indicates support for the ERETNC instruction. It is always 1 for R6 cores, except for those implementing the nanoMIPS™ subset. In other words, ERETNC is required for nanoMIPS™ cores and optional for NMS cores.

Exceptions:

Coprocessor Unusable. Reserved Instruction allowed for ERETNC on NMS cores.

EVP

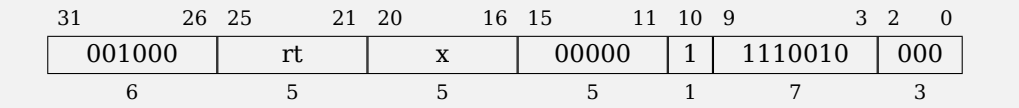
Assembly:

```
EVP rt
EVP    # rt=0 implied
```

Purpose: *Enable Virtual Processors*. Enable all virtual processors in a physical core. Set `VPControl.DIS` to 0, and place the previous value of the `VPControl` CP0 register in register `$rt`.

Availability: nanoMIPS. Optional, present when Config5.VP=1, otherwise NOP. Requires CP0 privilege.

Format:



Operation:

```
if C0.Config5.VP == 0:
    # No operation when VP not implemented
    pass
else:
    if not IsCoprocessor0Enabled():
        raise coprocessor_exception(0)

    GPR[rt] = C0.VPControl
    C0.VPControl.DIS = 0

    enable_virtual_processors()
```

The EVP instruction is used on VP cores to undo the effect of a DVP instruction, and the reader should refer to the DVP description for details regarding its usage.

The EVP instruction behaves like a NOP on cores which do not implement virtual processors (i.e. when Config5.VP=0). This behavior allows kernel code to enclose critical sequences within DVP/EVP blocks without first checking whether it is running on a VP core. The encoding of the EVP instruction is equivalent to a SLTU instruction targeting $0, i.e. a NOP, which leads to the correct behavior on non-VP cores with no additional hardware special casing.

Exceptions: Coprocessor Unusable.

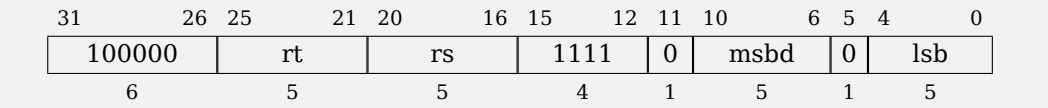
EXT

Assembly: `EXT rt, rs, pos, size`

Purpose: *Extract*. Extract a bit field of size `size` at position `pos` from register `$rs` and store it right-justified into register `$rt`.

Availability: nanoMIPS, not available in NMS

Format:



Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

pos = lsb
size = msbd + 1

if pos + size > 32:
    raise UNPREDICTABLE()

result = zero_extend(GPR[rs] >> pos, from_nbits=size)
GPR[rt] = sign_extend(result, from_nbits=32)
```

Exceptions: Reserved Instruction on NMS cores.

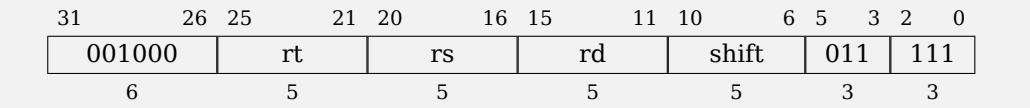
EXTW

Assembly: `EXTW rd, rs, rt, shift`

Purpose: *Extract Word*. Concatenate the 32 bit values in registers `$rt` and `$rs`, extract the word at specified bit position `shift`, and place the result in register `$rd`.

Availability: nanoMIPS

Format:



Operation:

```
tmp = GPR[rt][31:0] @ GPR[rs][31:0]
result = tmp >> shift
GPR[rd] = sign_extend(result, from_nbits=32)
```

Exceptions: None.

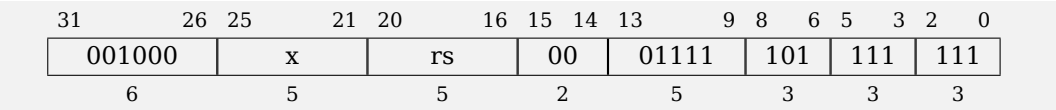
GINVI

Assembly: `GINVI rs`

Purpose: *Globally Invalidate Instruction caches*.

Availability: nanoMIPS. Optional, present when `Config5.GI >= 2`. Requires CP0 privilege.

Format:



Operation:

```
if C0.Config5.GI < 2:
    raise exception('RI')

if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

if GPR[rs] == 0:
    cores = get_all_cores_in_system()
else:
    cores = implementation_dependent_ginvi_cores(GPR[rs])

for core in cores:
    # Find encoded line size, sets, and associativity for the target cache.
    (L, S, A) = get_cache_parameters('I', core)

    num_sets = 2 ** (S + 6)
    num_ways = A + 1

    for way_index in range(num_ways):
        for set_index in range(num_sets):
            cache_line = get_cache_line('I', way_index, set_index, core)
            cache_line.valid = False
```

When $rs is 0, GINVI fully invalidates all instruction caches of all cores in the system, including the local instruction cache. For non-zero $rs values, GINVI invalidates the instruction cache of a specific, implementation dependent core in the system.

The GINVI instruction must be followed by a SYNC (stype=0x14) and an instruction hazard barrier (e.g. JRC.HB) to ensure that all instruction caches in the system have been invalidated.

Exceptions:

Coprocessor Unusable. Reserved Instruction if Global Invalidate I-cache not implemented.

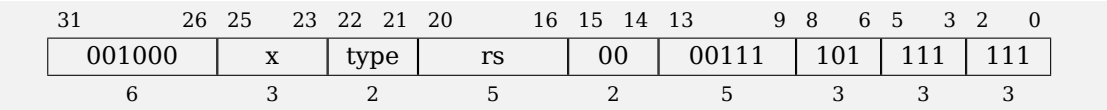
GINVT

Assembly: `GINVT rs, type`

Purpose: *Globally invalidate TLBs.*

Availability: nanoMIPS. Optional, present when `Config5.GI=3`. Requires CP0 privilege.

Format:



Operation:

```
if C0.Config5.GI != 3:
    raise exception('RI')

if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

if not C0.Config5.MI:
    raise exception('RI', 'Config5.MI not set')

ginvt(type, va=GPR[rs])
```

Perform `type` invalidation of all TLBs in the system, where `type` is one of:

- `type=0`: *invALL* - invalidate all non wired entries.
- `type=1`: *invVA*- invalidate all entries which match the VA specified by `$rs`.
- `type=2`: *invMMID* - invalidate all entries which match C0.MemoryMapID.MMID and are not global.
- `type=3`: *invVAMMID* - invalidate all entries which match the VA specified by `$rs` and either match C0.MemoryMapID or are global.

The GINVT instruction must be followed by a SYNC (stype=0x14) and an instruction hazard barrier (e.g. JRC.HB) to ensure that matching entries have been removed from all TLBs in the system and that all instructions in the instruction stream can only access the new context.

invMMID and invVAMMID operations use the C0.MemoryMapID value of the currently running process. The kernel must save/restore C0.MemoryMapID appropriately before it modifies it for the invalidation operation. Between the save and restore, it must utilize unmapped addresses.

Exceptions:

Coprocessor Unusable. Reserved Instruction if Global Invalidate TLB not implemented. Reserved Instruction if MemoryMapID not enabled (i.e. Config5.MI==0).

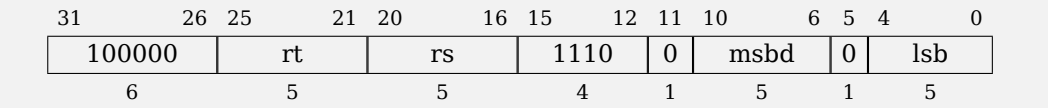
INS

Assembly: `INS rt, rs, pos, size`

Purpose: *Insert.* Merge a right justified bit field of size `size` from register `$rs` into position `pos` of register `$rt`.

Availability: nanoMIPS, not available in NMS

Format:



Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

pos = lsb
size = 1 + msbd - lsb

if size < 1:
    raise UNPREDICTABLE()

merge_mask = ((1<<size) - 1) << pos

result = (GPR[rt] & ~merge_mask
          | (GPR[rs] << pos) & merge_mask)

GPR[rt] = sign_extend(result, from_nbits=32)
```

The INS instruction is not available on NMS cores. It can be emulated using a sequence of three EXTW instructions:

```
INS     rt, rs, pos, size
```

can be emulated using the following sequence of instructions (provided rt is not equal to rs):

```
EXTW    rt, rt, rt, pos
EXTW    rt, rt, rs, size
EXTW    rt, rt, rt, 32 - size - pos
```

Exceptions: Reserved Instruction on NMS cores.

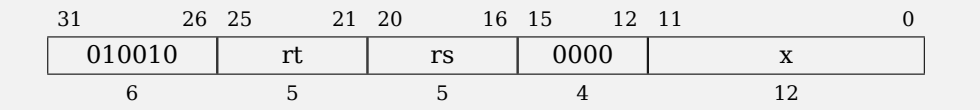
JALRC

Assembly: `JALRC dst, src`

Purpose: *Jump And Link Register, Compact*. Unconditional jump to address in register `$src`, placing the return address in register `$dst`.

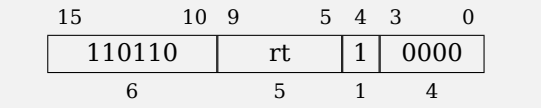
Availability: nanoMIPS

Format: JALRC[32]




```
src = rs
dst = rt
```

Format: JALRC[16]




```
src = rt
dst = 31
```

Operation:

```
address = GPR[src] + 0

GPR[dst] = CPU.next_pc
CPU.next_pc = address
```

Exceptions: None.

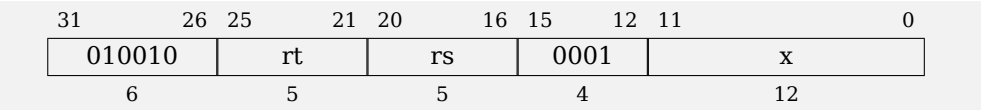
JALRC.HB

Assembly: `JALRC.HB rt, rs`

Purpose: *Jump And Link Register, Compact, with Hazard Barrier*. Unconditional jump to address in register `$rs`, placing the return address in register `$rt`. Clear all instruction and execution hazards before allowing any subsequent instructions to graduate.

Availability: nanoMIPS

Format:



Operation:

```
address = GPR[rs] + 0

GPR[rt] = CPU.next_pc
CPU.next_pc = address

clear_instruction_hazards()
clear_execution_hazards()
```

The JALRC.HB instruction creates an instruction hazard barrier, meaning that it ensures that subsequent instruction fetches will be aware of state changes caused by prior instructions. Examples of state changes which affect instruction fetch and which need an instruction hazard barrier to ensure that subsequent instructions see those updates are:

- Writes to the instruction stream (which must also have been synchronized by a SYNCI/SYNCIE and a SYNC).
- Updates to the TLB.
- Changes in CP0 state which affect addresses mappings.

In the absence of an instruction hazard barrier, the state used as input to an instruction fetch may be out of date, since it may have been read before the updates to that state have actually completed.

JALRC.HB also provides an execution hazard barrier, see the EHB instruction definition for details. An instruction hazard barrier is also provided by any of the exception return instructions ERET/ERETNC, or DERET, but those instructions are only available to privileged software, whereas JALRC.HB is available from all operating modes.

Exceptions: None.

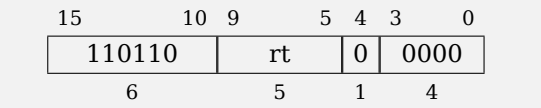
JRC

Assembly: `JRC rt`

Purpose: *Jump Register, Compact*. Unconditional jump to address in register `$rt`.

Availability: nanoMIPS

Format:



Operation:

```
address = GPR[rt]
CPU.next_pc = address
```

Exceptions: None.

LAPC (Assembly alias)

Assembly: `LAPC rt, address`

Purpose: *Load Address, PC relative*. Load PC relative address to register `$rt`.

Availability: Assembly alias. NMS cores restricted to 21 bit signed offset from PC.

Expansion:

```
address = $PC + imm (imm in 21 bit signed range):
    ADDIUPC[32] rt, imm

address = $PC + imm (imm in 32 bit signed range):
    ADDIUPC[48] rt, imm
```

LAPC uses the ADDIUPC instruction to load a PC relative address into register $rt. In order to determine the correct immediate value for the ADDIUPC instruction, the assembler must assume a value for the PC that the instruction will be executed from. If the instruction is executed from a different PC, then the generated address will be shifted by a PC relative amount.

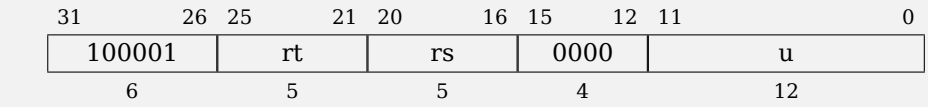
LB

Assembly: `LB rt, offset(rs)`

Purpose: *Load Byte*. Load signed byte to register `$rt` from memory address `$rs + offset` (register plus immediate).

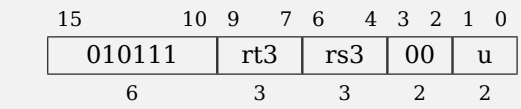
Availability: nanoMIPS

Format: LB[U12]



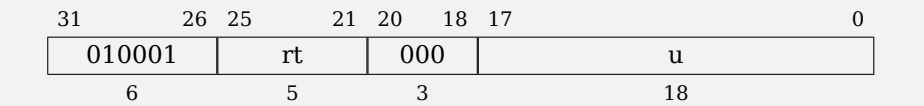

```
offset = u
```

Format: LB[16]



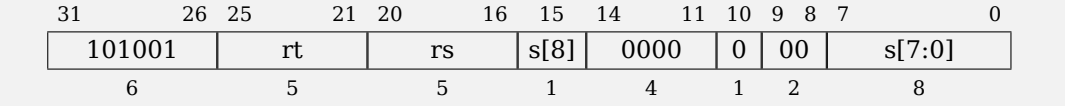

```
rt = decode_gpr(rt3, 'gpr3')
rs = decode_gpr(rs3, 'gpr3')
offset = u
```

Format: LB[GP]




```
rs = 28
offset = u
```

Format: LB[S9]




```
offset = sign_extend(s, from_nbits=9)
```

Operation:

```
va = effective_address(GPR[rs], offset, 'Load')

data = read_memory_at_va(va, nbytes=1)
GPR[rt] = sign_extend(data, from_nbits=8)
```

Exceptions:

Address Error. Bus Error. TLB Invalid. TLB Read Inhibit. TLB Refill. Watch.

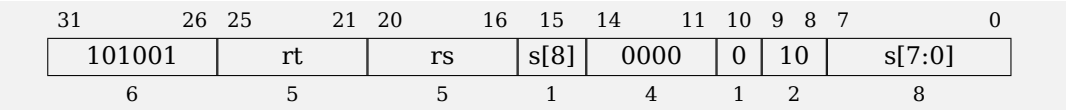
LBE

Assembly: `LBE rt, offset(rs)`

Purpose: *Load Byte using EVA addressing*. Load signed byte to register `$rt` from virtual address `$rs + offset`, translating the virtual address as though the core is in user mode, although it is actually in kernel mode.

Availability: nanoMIPS. Optional, present when Config5.EVA=1. Requires CP0 privilege.

Format:



Operation:

```
offset = sign_extend(s, from_nbits=9)

if not C0.Config5.EVA:
    raise exception('RI')

if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

va = effective_address(GPR[rs], offset, 'Load', eva=True)

data = read_memory_at_va(va, nbytes=1, eva=True)
GPR[rt] = sign_extend(data, from_nbits=8)
```

Exceptions:

Address Error. Bus Error. Coprocessor Unusable. Reserved Instruction if EVA not implemented. TLB Invalid. TLB Read Inhibit. TLB Refill. Watch.

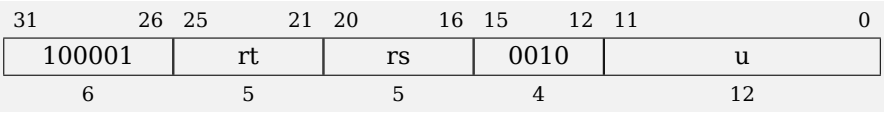
LBU

Assembly: `LBU rt, offset(rs)`

Purpose: *Load Byte Unsigned*. Load unsigned byte to register `$rt` from memory address `$rs` + `offset` (register plus immediate).

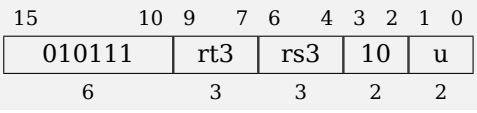
Availability: nanoMIPS

Format: LBU[U12]



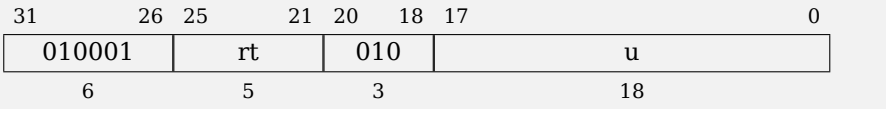

```
offset = u
```

Format: LBU[16]



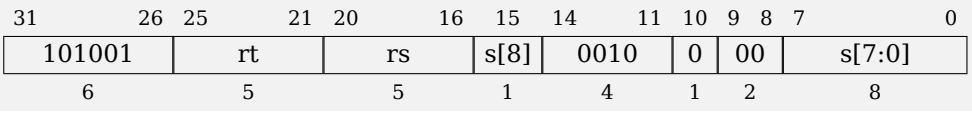

```
rt = decode_gpr(rt3, 'gpr3')
rs = decode_gpr(rs3, 'gpr3')
offset = u
```

Format: LBU[GP]




```
rs = 28
offset = u
```

Format: LBU[S9]




```
offset = sign_extend(s, from_nbits=9)
```

Operation:

```
va = effective_address(GPR[rs], offset, 'Load')
GPR[rt] = read_memory_at_va(va, nbytes=1)
```

Exceptions:

Address Error. Bus Error. TLB Invalid. TLB Read Inhibit. TLB Refill. Watch.

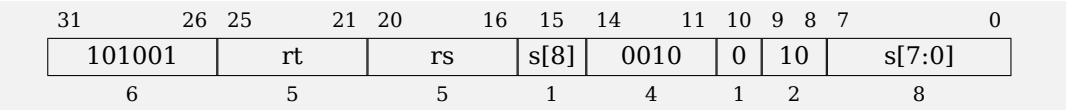
LBUE

Assembly: `LBUE rt, offset(rs)`

Purpose: *Load Byte Unsigned using EVA addressing*. Load unsigned byte to register `$rt` from virtual address `$rs + offset`, translating the virtual address as though the core is in user mode, although it is actually in kernel mode.

Availability: nanoMIPS. Optional, present when Config5.EVA=1. Requires CP0 privilege.

Format:



Operation:

```
offset = sign_extend(s, from_nbits=9)

if not C0.Config5.EVA:
    raise exception('RI')

if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

va = effective_address(GPR[rs], offset, 'Load', eva=True)
GPR[rt] = read_memory_at_va(va, nbytes=1, eva=True)
```

Exceptions:

Address Error. Bus Error. Coprocessor Unusable. Reserved Instruction if EVA not implemented. TLB Invalid. TLB Read Inhibit. TLB Refill. Watch.

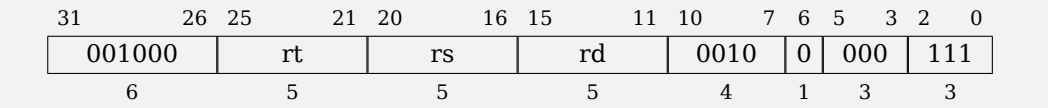
LBUX

Assembly: `LBUX rd, rs(rt)`

Purpose: *Load Byte Unsigned indeXed*. Load unsigned byte to register `$rd` from memory address `$rt + $rs` (register plus register).

Availability: nanoMIPS

Format:



Operation:

```
va = effective_address(GPR[rs], GPR[rt], 'Load')
GPR[rd] = read_memory_at_va(va, nbytes=1)
```

Exceptions:

Address Error. Bus Error. TLB Invalid. TLB Read Inhibit. TLB Refill. Watch.

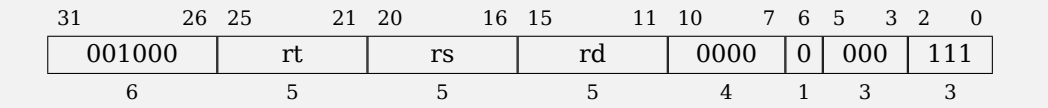
LBX

Assembly: `LBX rd, rs(rt)`

Purpose: *Load Byte indeXed*. Load signed byte to register `$rd` from memory address `$rt` + `$rs` (register plus register).

Availability: nanoMIPS

Format:



Operation:

```
va = effective_address(GPR[rs], GPR[rt], 'Load')

data = read_memory_at_va(va, nbytes=1)
GPR[rd] = sign_extend(data, from_nbits=8)
```

Exceptions:

Address Error. Bus Error. TLB Invalid. TLB Read Inhibit. TLB Refill. Watch.

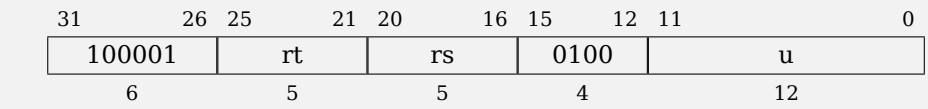
LH

Assembly: `LH rt, offset(rs)`

Purpose: *Load Half.* Load signed halfword to register `$rt` from memory address `$rs` + `offset` (register plus immediate).

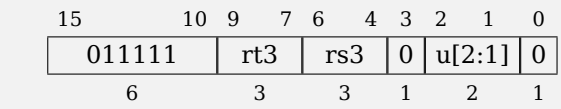
Availability: nanoMIPS

Format: LH[U12]



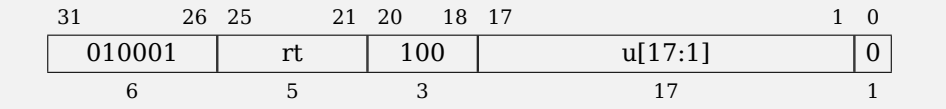

```
offset = u
```

Format: LH[16]



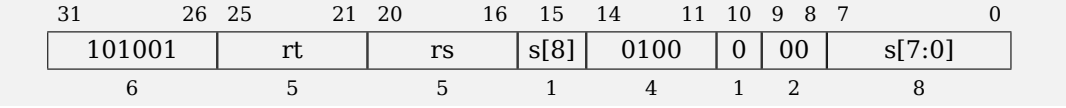

```
rt = decode_gpr(rt3, 'gpr3')
rs = decode_gpr(rs3, 'gpr3')
offset = u
```

Format: LH[GP]




```
rs = 28
offset = u
```

Format: LH[S9]




```
offset = sign_extend(s, from_nbits=9)
```

Operation:

```
va = effective_address(GPR[rs], offset, 'Load')

data = read_memory_at_va(va, nbytes=2)
GPR[rt] = sign_extend(data, from_nbits=16)
```

Exceptions:

Address Error. Bus Error. TLB Invalid. TLB Read Inhibit. TLB Refill. Watch.

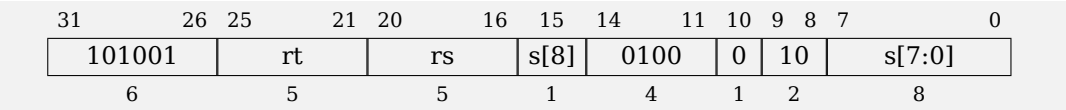
LHE

Assembly: `LHE rt, offset(rs)`

Purpose: *Load Half using EVA addressing.* Load signed halfword to register `$rt` from virtual address `$rs + offset`, translating the virtual address as though the core is in user mode, although it is actually in kernel mode.

Availability: nanoMIPS. Optional, present when Config5.EVA=1. Requires CP0 privilege.

Format:



Operation:

```
if not C0.Config5.EVA:
    raise exception('RI')

if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

offset = sign_extend(s, from_nbits=9)

va = effective_address(GPR[rs], offset, 'Load', eva=True)

data = read_memory_at_va(va, nbytes=2, eva=True)
GPR[rt] = sign_extend(data, from_nbits=16)
```

Exceptions:

Address Error. Bus Error. Coprocessor Unusable. Reserved Instruction if EVA not implemented. TLB Invalid. TLB Read Inhibit. TLB Refill. Watch.

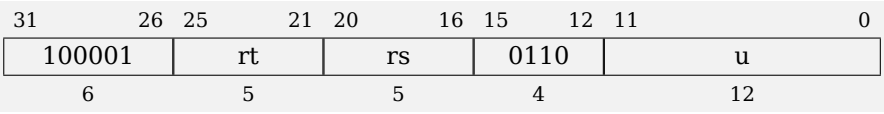
LHU

Assembly: `LHU rt, offset(rs)`

Purpose: *Load Half Unsigned*. Load unsigned halfword to register `$rt` from memory address `$rs` + `offset` (register plus immediate).

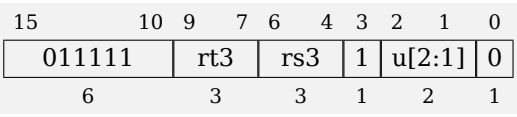
Availability: nanoMIPS

Format: LHU[U12]



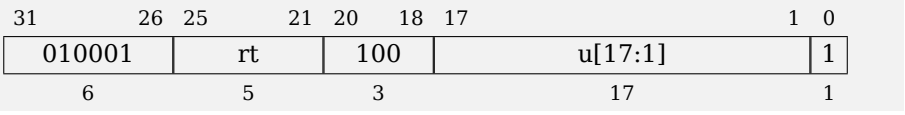

```
offset = u
```

Format: LHU[16]



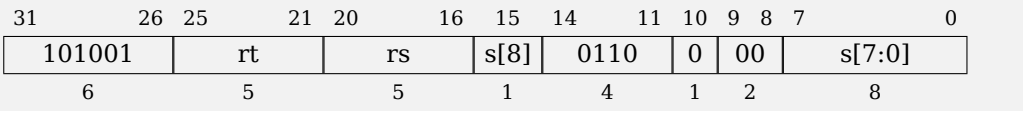

```
rt = decode_gpr(rt3, 'gpr3')
rs = decode_gpr(rs3, 'gpr3')
offset = u
```

Format: LHU[GP]




```
rs = 28
offset = u
```

Format: LHU[S9]




```
offset = sign_extend(s, from_nbits=9)
```

Operation:

```
va = effective_address(GPR[rs], offset, 'Load')
GPR[rt] = read_memory_at_va(va, nbytes=2)
```

Exceptions:

Address Error. Bus Error. TLB Invalid. TLB Read Inhibit. TLB Refill. Watch.

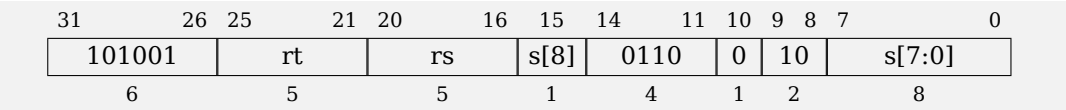
LHUE

Assembly: `LHUE rt, offset(rs)`

Purpose: *Load Half Unsigned using EVA addressing*. Load unsigned halfword to register `$rt` from virtual address `$rs + offset`, translating the virtual address as though the core is in user mode, although it is actually in kernel mode.

Availability: nanoMIPS. Optional, present when Config5.EVA=1. Requires CP0 privilege.

Format:



Operation:

```
if not C0.Config5.EVA:
    raise exception('RI')

if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

offset = sign_extend(s, from_nbits=9)

va = effective_address(GPR[rs], offset, 'Load', eva=True)
GPR[rt] = read_memory_at_va(va, nbytes=2, eva=True)
```

Exceptions:

Address Error. Bus Error. Coprocessor Unusable. Reserved Instruction if EVA not implemented. TLB Invalid. TLB Read Inhibit. TLB Refill. Watch.

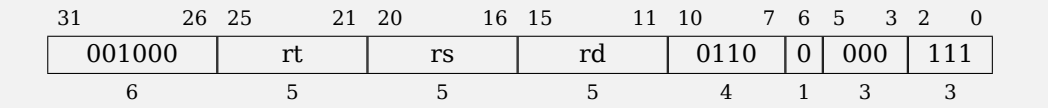
LHUX

Assembly: `LHUX rd, rs(rt)`

Purpose: *Load Half Unsigned indeXed*. Load unsigned halfword to register `$rd` from memory address `$rt + $rs` (register plus register).

Availability: nanoMIPS

Format:



Operation:

```
va = effective_address(GPR[rs], GPR[rt], 'Load')
GPR[rd] = read_memory_at_va(va, nbytes=2)
```

Exceptions:

Address Error. Bus Error. TLB Invalid. TLB Read Inhibit. TLB Refill. Watch.

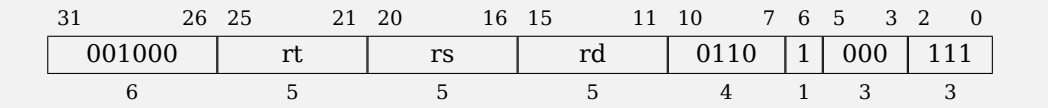
LHUXS

Assembly: `LHUXS rd, rs(rt)`

Purpose: *Load Half Unsigned indeXed Scaled*. Load unsigned halfword to register `$rd` from memory address `$rt + 2*$rs` (register plus scaled register).

Availability: nanoMIPS

Format:



Operation:

```
va = effective_address(GPR[rs]<<1, GPR[rt], 'Load')
GPR[rd] = read_memory_at_va(va, nbytes=2)
```

Exceptions:

Address Error. Bus Error. TLB Invalid. TLB Read Inhibit. TLB Refill. Watch.

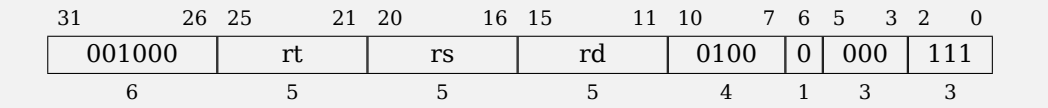
LHX

Assembly: `LHX rd, rs(rt)`

Purpose: *Load Half indeXed*. Load signed halfword to register `$rd` from memory address `$rt + $rs` (register plus register).

Availability: nanoMIPS

Format:



Operation:

```
va = effective_address(GPR[rs], GPR[rt], 'Load')

data = read_memory_at_va(va, nbytes=2)
GPR[rd] = sign_extend(data, from_nbits=16)
```

Exceptions:

Address Error. Bus Error. TLB Invalid. TLB Read Inhibit. TLB Refill. Watch.

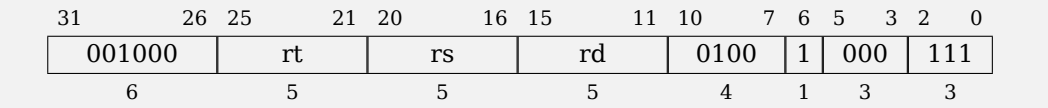
LHXS

Assembly: `LHXS rd, rs(rt)`

Purpose: *Load Half indeXed Scaled*. Load signed halfword to register `$rd` from memory address `$rt + 2*$rs` (register plus scaled register).

Availability: nanoMIPS

Format:



Operation:

```
va = effective_address(GPR[rs]<<1, GPR[rt], 'Load')

data = read_memory_at_va(va, nbytes=2)
GPR[rd] = sign_extend(data, from_nbits=16)
```

Exceptions:

Address Error. Bus Error. TLB Invalid. TLB Read Inhibit. TLB Refill. Watch.

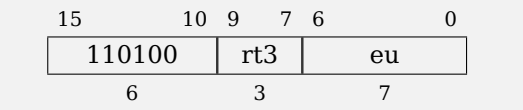
LI

Assembly: `LI rt, s`

Purpose: *Load Immediate*. Load immediate value `s` to register `$rt`.

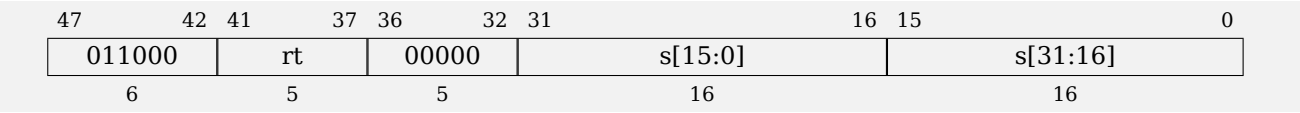
Availability: nanoMIPS, availability varies by format.

Format: LI[16]




```
rt = decode_gpr(rt3, 'gpr3')
s = -1 if eu == 127 else eu

not_in_nms = False
```

Format: LI[48], not available in NMS




```
s = sign_extend(s[31:16] @ s[15:0])

not_in_nms = True
```

Operation:

```
if not_in_nms and C0.Config5.NMS == 1:
    raise exception('RI')

GPR[rt] = s
```

Exceptions: Reserved Instruction for LI[48] format on NMS cores.

LL/LLE/LLWP/LLWPE

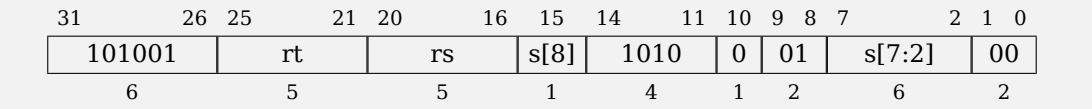
Assembly:

```
LL    rt, offset(rs)
LLE   rt, offset(rs)
LLWP  rt, ru, (rs)
LLWPE rt, ru, (rs)
```

Purpose: *Load Linked word/Load Linked word using EVA addressing/Load Linked Word Pair/Load Linked Word Pair using EVA addressing*. For LL/LLE, load word for atomic RMW to register `$rt` from address `$rs + offset` (register plus immediate). For LLWP/LLWPE, load words for atomic RMW to registers `$rt` and `$ru` from address `$rs`. For LLE/LLWPE, translate the virtual address as though the core is in user mode, although it is actually in kernel mode.

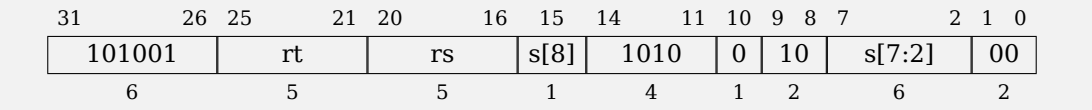
Availability: nanoMIPS, availability varies by format.

Format: LL



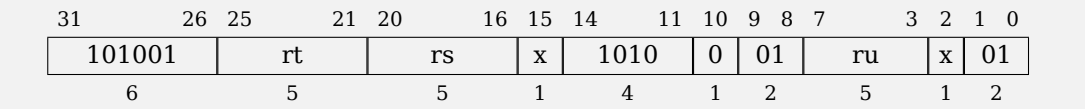

```
offset = sign_extend(s, from_nbits=9)
nbytes = 4
is_eva = False
```

Format: LLE, present when Config5.EVA=1, requires CP0 privilege.



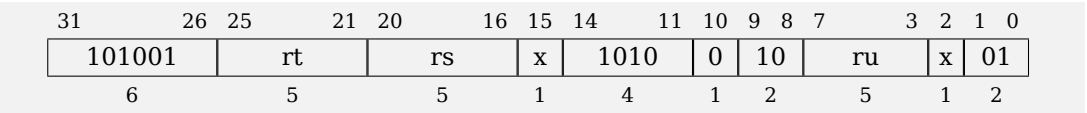

```
offset = sign_extend(s, from_nbits=9)
nbytes = 4
is_eva = True
```

Format: LLWP, required (optional on NMS cores).




```
offset = 0
nbytes = 8
is_eva = False
```

Format: LLWPE, present when Config5.EVA=1. Requires CP0 privilege.




```
offset = 0
nbytes = 8
is_eva = True
```

Operation:

```
if nbytes == 8 and C0.Config5.XNP:
    raise exception('RI', 'LLWP[E] requires word-paired support')

if is_eva and not C0.Config5.EVA:
    raise exception('RI')

va = effective_address(GPR[rs], offset, 'Load', eva=is_eva)

# Linked access must be aligned.
if va & (nbytes-1):
    raise exception('ADEL', badva=va)

pa, cca = va2pa(va, 'Load', eva=is_eva)

if (cca == 2 or cca == 7) and not C0.Config5.ULS:
    raise UNPREDICTABLE('uncached CCA not synchronizable when Config5.ULS=0')
    # (Preferred behavior for non-synchronizable address is Bus Error).

# Indicate that there is an active RMW sequence on this processor.
C0.LLAddr.LLB = 1

# Save target address of active RMW sequence.
record_linked_address(va, pa, cca, nbytes=nbytes)

data = read_memory(va, pa, cca, nbytes=nbytes)

if nbytes == 4:  # LL/LLE
    GPR[rt] = sign_extend(data, from_nbits=32)

else:  # LLWP/LLWPE
    word0 = data[63:32] if C0.Config.BE else data[31:0]
    word1 = data[31:0] if C0.Config.BE else data[63:32]

    if rt == ru:
        raise UNPREDICTABLE()
```

```
45
46      GPR[rt] = sign_extend(word0, from_nbits=32)
47      GPR[ru] = sign_extend(word1, from_nbits=32)
```

The LL/LLE/LLWP/LLWPE instructions are used to initiate an atomic read-modify-write sequence. C0.LLAddr.LLB is set to 1, indicating that there is an active RMW sequence on the current processor, and an implementation dependent set of state is saved which indicates the address and access type of the active RMW sequence. There can be only one active RMW sequence per processor.

The RMW sequence will be completed by a matching SC/SCE/SCWP/SCWPE instruction. The store-conditional instruction will only complete if the system can guarantee that the accessed memory location has not been modified since the load-linked instruction occurred, as discussed in more detail in the SC/SCE/SCWP/SCWPE instruction description.

The address and CCA targeted by the LL/LLE/LLWP/LLWPE must be must be synchronizable by all processors and I/O devices sharing the location; if it is not, the result is UNPREDICTABLE. Which storage is synchronizable is a function of both CPU and system implementations - see the SC/SCE/SCWP/SCWPE instruction for the formal definition. The preferred behavior for a load-linked instruction which attempts to access an address which is not synchronizable is a Bus Error exception.

If Config5.ULS is set, then the system supports uncached load-linked/store-conditional accesses. Otherwise, the result of uncached accesses is unpredictable.

A LL/LLE/LLWP/LLWPE instruction on one processor must not take action that, by itself, causes a store-conditional instruction for the same block on another processor to fail. For example, if an implementation depends on retaining the data in the cache during the RMW sequence, cache misses caused by a load-linked instruction must not fetch data in the exclusive state, since that would remove it from another core's cache if it were present.

An execution of a load-linked instruction does not have to be followed by execution of store-conditional instruction; a program is free to abandon the RMW sequence without attempting a write.

Support for the paired word instructions LLWP/LLWPE is indicated by the Config5.XNP bit. Paired word support is required for nanoMIPS™ cores, except for NMS cores, where it is optional.

The result of LLWP/LLWPE is unpredictable if `$rt` and `$ru` are the same register.

Exceptions:

Address Error. Bus Error. Coprocessor Unusable for LLE/LLWPE. Reserved Instruction for LLE/LLWPE if EVA not implemented. Reserved Instruction for LLWP/LLWPE if load linked pair not implemented. TLB Invalid. TLB Read Inhibit. TLB Refill. Watch.

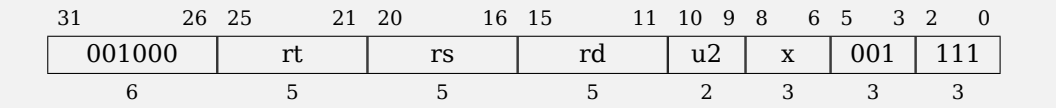
LSA

Assembly: `LSA rd, rs, rt, u2`

Purpose: *Load Scaled Address*. Add register `$rs` scaled by a left shift `u2` to register `$rt` and place the 32 bit result in register `$rd`.

Availability: nanoMIPS

Format:



Operation:

```
sum = (GPR[rs] << u2) + GPR[rt]
GPR[rd] = sign_extend(sum, from_nbits=32)
```

In nanoMIPS™, the shift field directly encodes the shift amount, meaning that the supported shift values are in the range 0 to 3 (instead of 1 to 4 in MIPSR6™).

Exceptions: None.

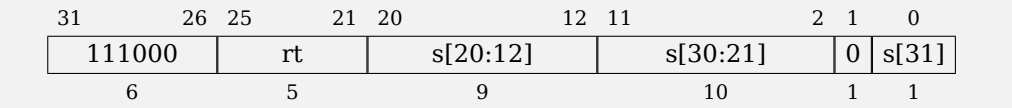
LUI

Assembly: `LUI rt, %hi(imm)`

Purpose: *Load Upper Immediate*. Load upper 20 bits of immediate value `imm` to upper 20 bits of register `$rt`, and set the lower 12 bits to zero.

Availability: nanoMIPS

Format:




```
imm = sign_extend(s, from_nbits=32)
```

Operation:

```
GPR[rt] = imm
```

For backwards compatibility, instances of LUI which use a literal value for the immediate will be treated as containing a 16 bit immediate which should be loaded into the upper 16 bits of the target register. To access the upper 20 bits of the register, the '`%hi(imm)`' form of the immediate must be used.

Exceptions: None.

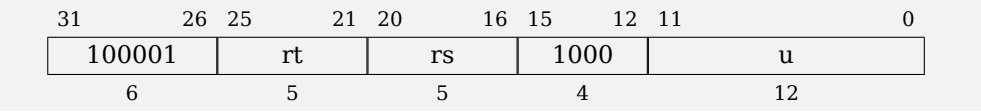
LW

Assembly: `LW rt, offset(rs)`

Purpose: *Load Word*. Load word to register `$rt` from memory address `$rs + offset` (register plus immediate).

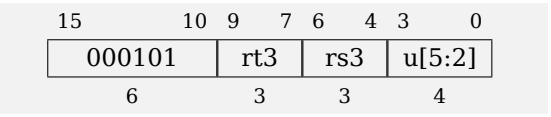
Availability: nanoMIPS, availability varies by format.

Format: LW[U12]



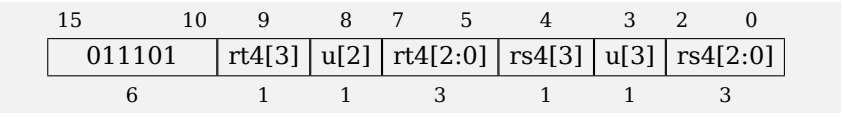

```
offset = u
```

Format: LW[16]



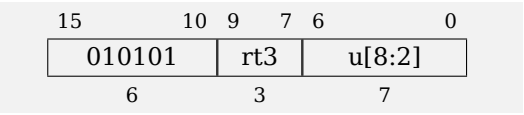

```
rt = decode_gpr(rt3, 'gpr3')
rs = decode_gpr(rs3, 'gpr3')
offset = u
```

Format: LW[4X4], not available in NMS



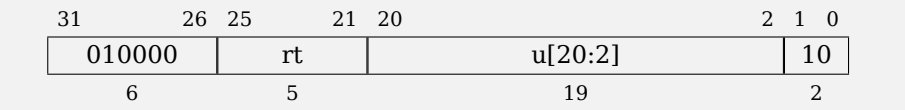

```
if C0.Config5.NMS == 1:
    raise exception('RI')

rt = decode_gpr(rt4[3] @ rt4[2:0], 'gpr4')
rs = decode_gpr(rs4[3] @ rs4[2:0], 'gpr4')
offset = u
```

Format: LW[GP16]



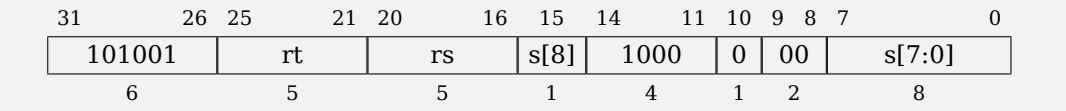
```
rt = decode_gpr(rt3, 'gpr3')
rs = 28
offset = u
```

Format: LW[GP]



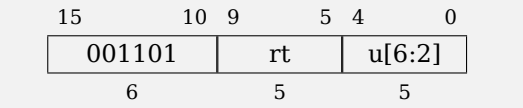

```
rs = 28
offset = u
```

Format: LW[S9]




```
offset = sign_extend(s, from_nbits=9)
```

Format: LW[SP]




```
rs = 29
offset = u
```

Operation:

```
va = effective_address(GPR[rs], offset, 'Load')

data = read_memory_at_va(va, nbytes=4)
GPR[rt] = sign_extend(data, from_nbits=32)
```

Exceptions:

Address Error. Bus Error. Reserved Instruction for LW[4X4] format on NMS cores. TLB Invalid. TLB Read Inhibit. TLB Refill. Watch.

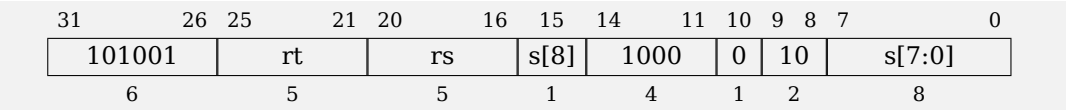
LWE

Assembly: `LWE rt, offset(rs)`

Purpose: *Load Word using EVA addressing*. Load word to register `$rt` from virtual address `$rs + offset`, translating the virtual address as though the core is in user mode, although it is actually in kernel mode.

Availability: nanoMIPS. Optional, present when Config5.EVA=1. Requires CP0 privilege.

Format:



Operation:

```
offset = sign_extend(s, from_nbits=9)

if not C0.Config5.EVA:
    raise exception('RI')

if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

va = effective_address(GPR[rs], offset, 'Load', eva=True)

data = read_memory_at_va(va, nbytes=4, eva=True)
GPR[rt] = sign_extend(data, from_nbits=32)
```

Exceptions:

Address Error. Bus Error. Coprocessor unusable. Reserved Instruction if EVA not implemented. TLB Invalid. TLB Read Inhibit. TLB Refill. Watch.

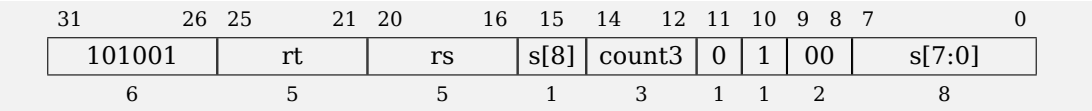
LWM

Assembly: `LWM rt, offset(rs), count`

Purpose: *Load Word Multiple*. Load `count` words of data to registers `$rt`, `$(rt+1)`, ..., `$(rt+count-1)` from consecutive memory address starting at `$rs` + `offset` (register plus immediate).

Availability: nanoMIPS, not available in NMS

Format:




```
offset = sign_extend(s, from_nbits=9)
count = 8 if count3 == 0 else count3
```

Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

i = 0
while i != count:
    this_rt = ( rt + i      if rt + i < 32 else
                rt + i - 16                    )

    this_offset = offset + (i<<2)
    va = effective_address(GPR[rs], this_offset, 'Load')

    data = read_memory_at_va(va, nbytes=4)
    GPR[this_rt] = sign_extend(data, from_nbits=32)

    if this_rt == rs and i != count - 1:
        raise UNPREDICTABLE()

    i += 1
```

LWM loads `count` words to sequentially numbered register from sequential memory addresses. After loading $31, the sequence of registers continues from $16. Some example encodings of the register list are:

- rt=15, count=3: loads [$15, $16, $17]
- rt=31, count=3: loads [$31, $16, $17].

The result is unpredictable if an LWM instruction updates the base register prior to the final load.

LWM must be implemented in such a way as to make the instruction restartable, but the implementation does not need to be fully atomic. For instance, it is allowable for a LWM instruction to be aborted by an exception after a subset of the register updates have occurred. To ensure restartability, any write to GPR `$rs` (which may be used as the final output register) must be completed atomically, that is, the instruction must graduate if and only if that write occurs.

Exceptions:

Address Error. Bus Error. Reserved Instruction on NMS cores. TLB Invalid. TLB Read Inhibit. TLB Refill. Watch.

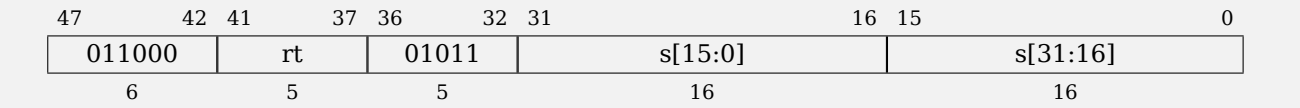
LWPC

Assembly: `LWPC rt, address`

Purpose: *Load Word PC relative*. Load word to register `$rt` from PC relative address `address`.

Availability: nanoMIPS, not available in NMS

Format: LWPC[48]




```
offset = sign_extend(s, from_nbits=32)
```

Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

address = effective_address(CPU.next_pc, offset)

data = read_memory_at_va(address, nbytes=4)
GPR[rt] = sign_extend(data, from_nbits=32)
```

Exceptions:

Address Error. Bus Error. Reserved Instruction on NMS cores TLB Invalid. TLB Read Inhibit. TLB Refill. Watch.

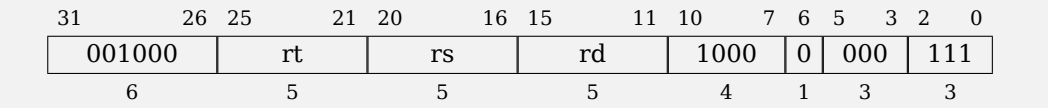
LWX

Assembly: `LWX rd, rs(rt)`

Purpose: *Load Word indeXed*. Load word to register `$rd` from memory address `$rt + $rs` (register plus register).

Availability: nanoMIPS

Format:



Operation:

```
va = effective_address(GPR[rs], GPR[rt], 'Load')

data = read_memory_at_va(va, nbytes=4)
GPR[rd] = sign_extend(data, from_nbits=32)
```

Exceptions:

Address Error. Bus Error. TLB Invalid. TLB Read Inhibit. TLB Refill. Watch.

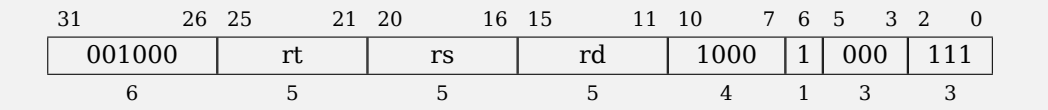
LWXS

Assembly: `LWXS rd, rs(rt)`

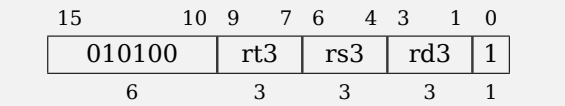
Purpose: *Load Word indeXed Scaled*. Load word to register `$rd` from memory address `$rt + 4*$rs` (register plus scaled register).

Availability: nanoMIPS

Format: LWXS[32]



Format: LWXS[16]




```
rd = decode_gpr(rd3, 'gpr3')
rs = decode_gpr(rs3, 'gpr3')
rt = decode_gpr(rt3, 'gpr3')
```

Operation:

```
va = effective_address(GPR[rs]<<2, GPR[rt], 'Load')

data = read_memory_at_va(va, nbytes=4)
GPR[rd] = sign_extend(data, from_nbits=32)
```

Exceptions:

Address Error. Bus Error. TLB Invalid. TLB Read Inhibit. TLB Refill. Watch.

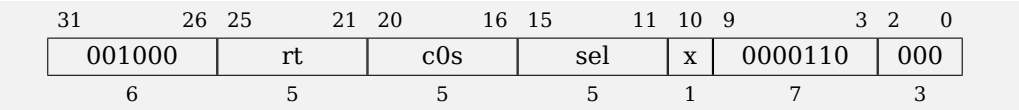
MFC0

Assembly: `MFC0 rt, c0s, sel`

Purpose: *Move From Coprocessor 0*. Write value of CP0 register indexed by `c0s`, `sel` to register `$rt`.

Availability: nanoMIPS. Requires CP0 privilege.

Format:



Operation:

```
if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

value = read_cp0_register(c0s, sel)
GPR[rt] = sign_extend(value, from_nbits=32)
```

An MFC0 which targets a register which is not used on the current core will return zero.

Exceptions: Coprocessor Unusable.

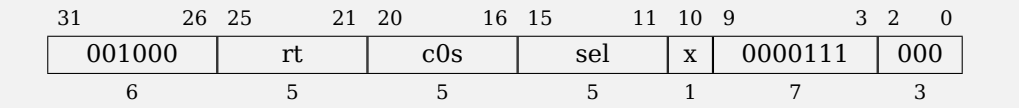
MFHC0

Assembly: `MFHC0 rt, c0s, sel`

Purpose: *Move From High Coprocessor 0*. Write bits 63..32 (when present) of CP0 register indexed by `c0s`, `sel` to register `$rt`.

Availability: nanoMIPS, required. (Optional on NMS cores). Requires CP0 privilege.

Format:



Operation:

```
if C0.Config5.MVH == 0:
    raise exception('RI')

if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

value = read_cp0_register(c0s, sel, h=True)
GPR[rt] = sign_extend(value, from_nbits=32)
```

For certain core configurations, specific nanoMIPS32™ CP0 registers may be extended to be 64 bits wide. The MFHC0 instruction is used to read the upper 32 bits of such registers. An MFHC0 which targets a register for which the 'high' bits are not used will return zero.

This instruction is available when Config5.MVH=1, which is required on nanoMIPS™ cores, except for NMS cores where it is optional.

Exceptions:

Coprocessor Unusable. Reserved Instruction on NMS cores without MVH support.

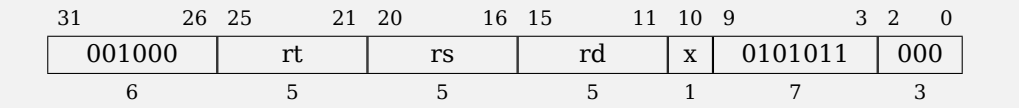
MOD

Assembly: `MOD rd, rs, rt`

Purpose: *Modulo*. Compute signed division of register `$rs` by register `$rt`, and place the remainder in register `$rd`.

Availability: nanoMIPS

Format:



Operation:

```
numerator = GPR[rs]
denominator = GPR[rt]

if denominator == 0:
    quotient, remainder = (UNKNOWN, UNKNOWN)

else:
    quotient, remainder = divide_integers(numerator, denominator)

GPR[rd] = sign_extend(remainder, from_nbits=32)
```

Exceptions: None.

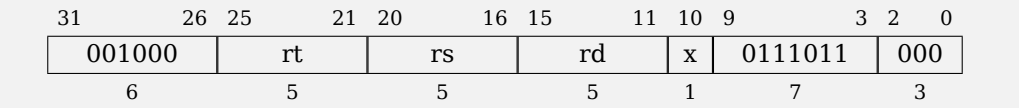
MODU

Assembly: `MODU rd, rs, rt`

Purpose: *Modulo Unsigned*. Compute unsigned division of register `$rs` by register `$rt`, and place the remainder in register `$rd`.

Availability: nanoMIPS

Format:



Operation:

```
numerator = zero_extend(GPR[rs], from_nbits=32)
denominator = zero_extend(GPR[rt], from_nbits=32)

if denominator == 0:
    quotient, remainder = (UNKNOWN, UNKNOWN)

else:
    quotient, remainder = divide_integers(numerator, denominator)

GPR[rd] = sign_extend(remainder, from_nbits=32)
```

Exceptions: None.

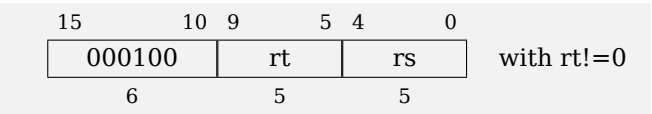
MOVE

Assembly: `MOVE rt, rs`

Purpose: *Move*. Copy value of register `$rs` to register `$rt`.

Availability: nanoMIPS

Format:



Operation:

```
GPR[rt] = GPR[rs]
```

Exceptions: None.

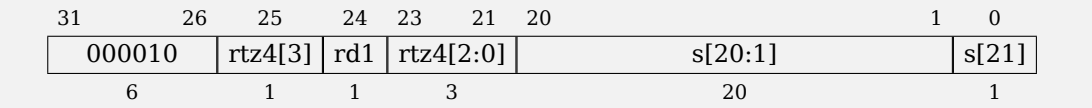
MOVE.BALC

Assembly: `MOVE.BALC rd, rt, address`

Purpose: *Move and Branch and Link, Compact.* Copy value of register `$rt` to register `$rd`, and perform an unconditional PC relative branch to `address`, placing the return address in register `$31`.

Availability: nanoMIPS, not available in NMS

Format:



Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

rd = decode_gpr(rd1, 'gpr1')
rt = decode_gpr(rtz4[3] @ rtz4[2:0], 'gpr4.zero')

offset = sign_extend(s, from_nbits=22)
address = effective_address(CPU.next_pc, offset)

GPR[rd] = GPR[rt]

GPR[31] = CPU.next_pc
CPU.next_pc = address
```

Although this instruction is called MOVE.BALC, the order of the updates to PC, `$31` and `$rd` is invisible to software, and an implementation may choose any order for carring out these steps.

Exceptions: Reserved Instruction on NMS cores.

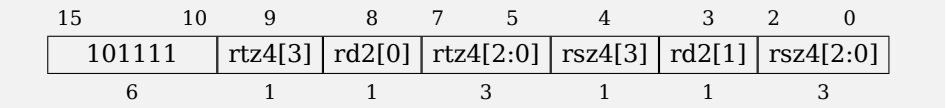
MOVEP

Assembly: `MOVEP dst1, dst2, src1, src2`

Purpose: *Move Pair.* Copy value of register `$src1` to register `$dst1`, and copy value of register `$src2` to register `$dst2`.

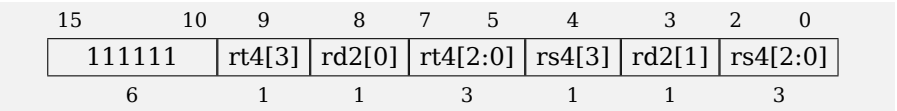
Availability: nanoMIPS, not available in NMS

Format: MOVEP




```
dst1 = decode_gpr(rd2[1] @ rd2[0], 'gpr2.reg1')
dst2 = decode_gpr(rd2[1] @ rd2[0], 'gpr2.reg2')
src1 = decode_gpr(rsz4[3]  @ rsz4[2:0], 'gpr4.zero')
src2 = decode_gpr(rtz4[3]  @ rtz4[2:0], 'gpr4.zero')
```

Format: MOVEP[REV]




```
dst1 = decode_gpr(rs4[3] @ rs4[2:0], 'gpr4')
dst2 = decode_gpr(rt4[3] @ rt4[2:0], 'gpr4')
src1 = decode_gpr(rd2[1] @ rd2[0], 'gpr2.reg1')
src2 = decode_gpr(rd2[1] @ rd2[0], 'gpr2.reg2')
```

Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

if dst1 == src1 or dst1 == src2 or dst2 == src1 or dst2 == src2:
    GPR[dst1] = UNKNOWN
    GPR[dst2] = UNKNOWN
else:
    GPR[dst1] = GPR[src1]
    GPR[dst2] = GPR[src2]
```

The output register values are unpredictable if either of the output registers is also used as an input.

Exceptions: Reserved Instruction on NMS cores.

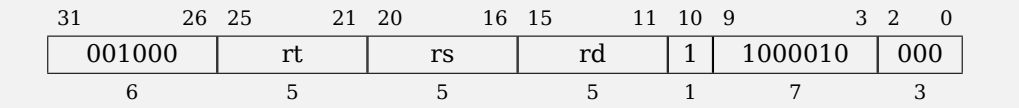
MOVN

Assembly: `MOVN rd, rs, rt`

Purpose: *Move if Not zero*. Copy value of register `$rs` to register `$rd` if register `$rt` is not zero.

Availability: nanoMIPS

Format:



Operation:

```
GPR[rd] = GPR[rs] if GPR[rt] != 0 else GPR[rd]
```

Exceptions: None.

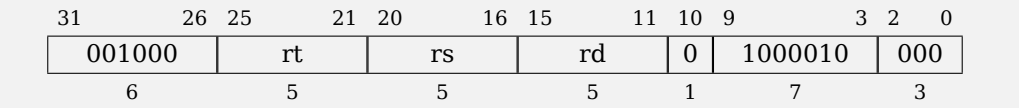
MOVZ

Assembly: `MOVZ rd, rs, rt`

Purpose: *Move if Zero*. Copy value of register `$rs` to register `$rd` if register `$rt` is zero.

Availability: nanoMIPS

Format:



Operation:

```
GPR[rd] = GPR[rs] if GPR[rt] == 0 else GPR[rd]
```

Exceptions: None.

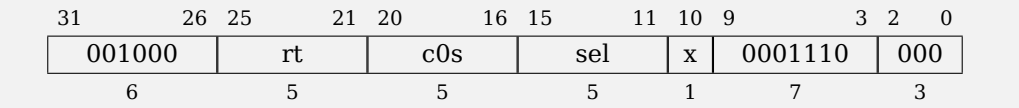
MTC0

Assembly: `MTC0 rt, c0s, sel`

Purpose: *Move To Coprocessor 0*. Write value of register `$rt` to CP0 register indexed by `c0s`, `sel`.

Availability: nanoMIPS. Requires CP0 privilege.

Format:



Operation:

```
if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

write_cp0_register(GPR[rt], c0s, sel)
```

An MTC0 to a register which is not used on the current core is ignored.

When a register is extended to have high bits for a specific configuration (see MTHC0), legacy software which is not aware of the existence of these high bits still needs to function correctly. In such cases, the architecture may require that an MTC0 modifies the high 32 bits of the register as well as the low 32 bits to give the correct legacy behavior.

For this reason, when setting an extended CP0 register, the MTC0 to set the low 32 bits should always precede the MTHC0 to set the high 32 bits. Also, a read-modify-write sequence to set a specific bitfield in the low 32 bits should read both the low 32 and high 32 bits, then do MTC0 followed by MTHC0 to write the modified value back.

Exceptions: Coprocessor Unusable.

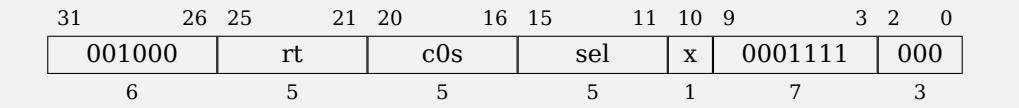
MTHC0

Assembly: `MTHC0 rt, c0s, sel`

Purpose: *Move To High Coprocessor 0*. Write value of register `$rt` to bits 63..32 (when present) of CP0 register indexed by `c0s`, `sel`.

Availability: nanoMIPS, required. (Optional on NMS cores). Requires CP0 privilege.

Format:



Operation:

```
if C0.Config5.MVH == 0:
    raise exception('RI')

if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

write_cp0_register(GPR[rt], c0s, sel, h=True)
```

For certain core configurations, specific nanoMIPS32™ CP0 registers may be extended to be 64 bits wide. The MTHC0 instruction is used to write the upper 32 bits of such registers. An MTHC0 to a register for which the 'high' bits are not used will be ignored.

When a register is extended to have high bits for a specific configuration, legacy software which is not aware of the existence of these high bits still needs to function correctly. In such cases, the architecture may require that an MTC0 modifies the high 32 bits of the register as well as the low 32 bits to give the correct legacy behavior.

For this reason, when setting an extended CP0 register, the MTC0 to set the low 32 bits should always precede the MTHC0 to set the high 32 bits. Also, a read-modify-write sequence to set a specific bitfield in the low 32 bits should read both the low 32 and high 32 bits, then do MTC0 followed by MTHC0 to write the modified value back.

This instruction is available when Config5.MVH=1, which is required on nanoMIPS™ cores, except for NMS cores where it is optional.

Exceptions:

Coprocessor Unusable. Reserved Instruction on NMS cores without MVHm support.

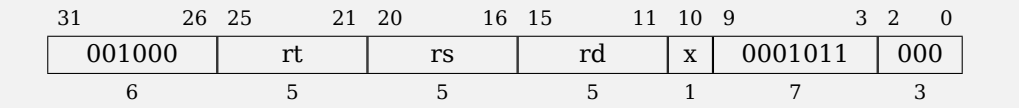
MUH

Assembly: `MUH rd, rs, rt`

Purpose: *Multiply High*. Multiply signed word values from registers `$rs` and `$rt`, and place bits 63..32 of the result in register `$rd`.

Availability: nanoMIPS

Format:



Operation:

```
result = GPR[rs] * GPR[rt]

result_hi = result[63:32]

GPR[rd] = sign_extend(result_hi, from_nbits=32)
```

Exceptions: None.

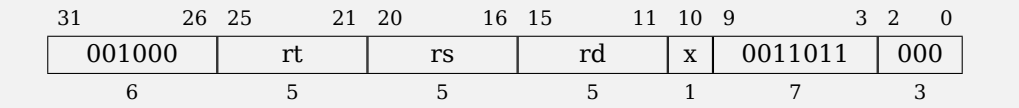
MUHU

Assembly: `MUHU rd, rs, rt`

Purpose: *Multiply High Unsigned*. Multiply unsigned word values in registers `$rs` and `$rt`, and place bits 63..32 of the result in register `$rd`.

Availability: nanoMIPS

Format:



Operation:

```
rs_unsigned = zero_extend(GPR[rs], from_nbits=32)
rt_unsigned = zero_extend(GPR[rt], from_nbits=32)

result = rs_unsigned * rt_unsigned

result_hi = result[63:32]

GPR[rd] = sign_extend(result_hi, from_nbits=32)
```

Exceptions: None.

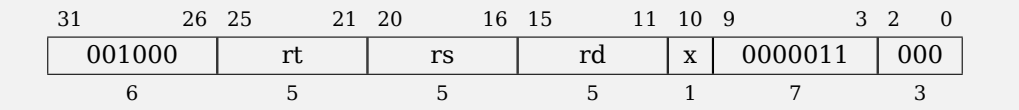
MUL

Assembly: `MUL dst, src1, src2`

Purpose: *Multiply.* Multiply signed word values in registers `$src1` and `$src2`, and place bits 31..0 of the result in register `$dst`.

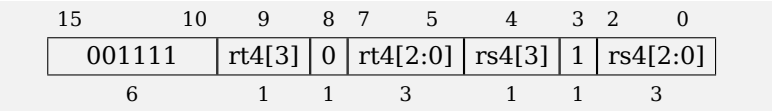
Availability: nanoMIPS, availability varies by format.

Format: MUL[32]




```
dst = rd
src1 = rs
src2 = rt

not_in_mms = False
```

Format: MUL[4X4], not available in NMS




```
dst = decode_gpr(rt4, 'gpr4')
src1 = decode_gpr(rt4, 'gpr4')
src2 = decode_gpr(rs4, 'gpr4')

not_in_mms = True
```

Operation:

```
if not_in_mms and C0.Config5.NMS == 1:
    raise exception('RI')

result = GPR[src1] * GPR[src2]

GPR[dst] = sign_extend(result, from_nbits=32)
```

Exceptions: Reserved Instruction for MUL[4X4] format on NMS cores.

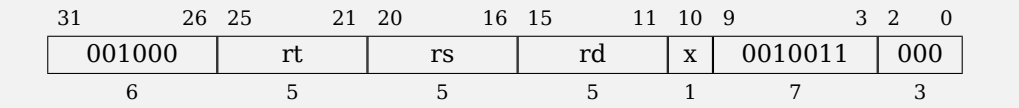
MULU

Assembly: `MULU rd, rs, rt`

Purpose: *Multiply Unsigned*. Multiply unsigned word values in registers `$rs` and `$rt`, and place bits 31..0 of the result in register `$rd`.

Availability: nanoMIPS

Format:



Operation:

```
rs_unsigned = zero_extend(GPR[rs], from_nbits=32)
rt_unsigned = zero_extend(GPR[rt], from_nbits=32)

result = rs_unsigned * rt_unsigned

GPR[rd] = sign_extend(result, from_nbits=32)
```

Exceptions: None.

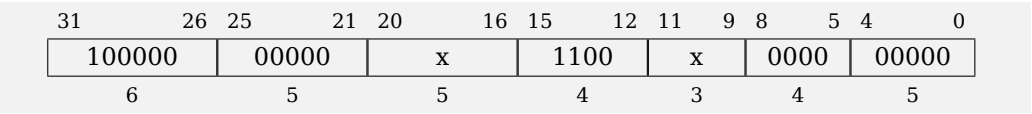
NOP

Assembly: `NOP`

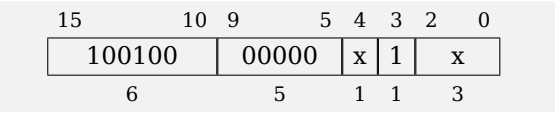
Purpose: *No Operation*.

Availability: nanoMIPS

Format: NOP[32]



Format: NOP[16]



Operation:

```
# No operation
pass
```

The NOP[32] encoding is equivalent to an SLL[32] instruction using `$0` as output and a shift value of 0. The NOP[16] encoding is equivalent to an ADDIU[RS5] instruction using `$0` as output. Therefore NOP does not necessarily need any additional implementation in hardware beyond the normal behavior of the SLL[32] and ADDIU[RS5] instructions.

If software intentionally generates a NOP instruction, it should only generate these specific encodings, rather than other instructions writing to `$0` which would also result in no operation.

If hardware implements a performance counter for nops, it can expect these specific instruction encodings to be used. It should ignore the `x` field of the encoding, treating all values of `x` as representing a valid NOP instruction. Software on the other hand should only generate NOP instructions with an `x` value of 0.

As for all instruction definitions containing `x` fields, this methodology allows for the possibility that the meaning of `x` values other than zero might be enhanced in the future, with the understanding that cores prior to the enhanced definition will treat the `x!=0` encodings as equivalent to the `x==0` instruction.

Exceptions: None.

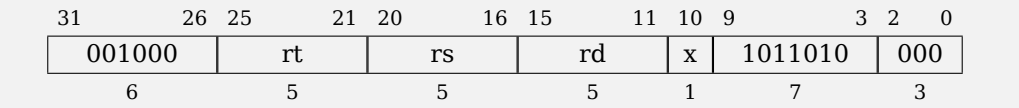
NOR

Assembly: `NOR rd, rs, rt`

Purpose: *NOR*. Compute logical NOR of registers `$rs` and `$rt`, placing the result in register `$rt`.

Availability: nanoMIPS

Format:



Operation:

```
GPR[rd] = ~(GPR[rs] | GPR[rt])
```

Exceptions: None.

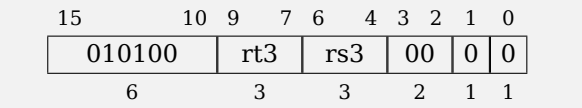
NOT[16]

Assembly: `NOT rt, rs`

Purpose: *NOT*. Write logical inversion of register `$rs` to register `$rt`.

Availability: nanoMIPS

Format:



Operation:

```
rt = decode_gpr(rt3, 'gpr3')
rs = decode_gpr(rs3, 'gpr3')

GPR[rt] = ~GPR[rs]
```

Exceptions: None.

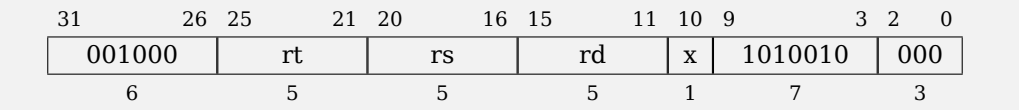
OR

Assembly: `OR rd, rs, rt`

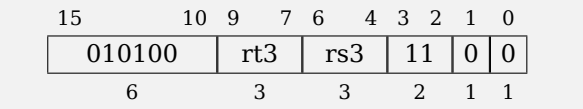
Purpose: *OR*. Compute logical OR of registers `$rs` and `$rt`, placing the result in register `$rt`.

Availability: nanoMIPS

Format: OR[32]



Format: OR[16]




```
rt = decode_gpr(rt3, 'gpr3')
rs = decode_gpr(rs3, 'gpr3')
rd = rt
```

Operation:

```
GPR[rd] = GPR[rs] | GPR[rt]
```

Exceptions: None.

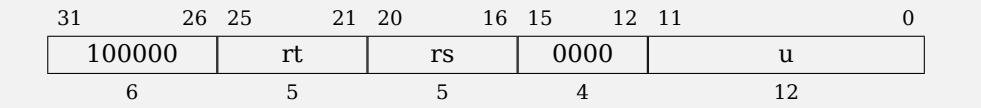
ORI

Assembly: `ORI rt, rs, u`

Purpose: *OR Immediate*. Compute logical OR of register `$rs` with immediate `u`, placing the result in register `$rt`.

Availability: nanoMIPS

Format:



Operation:

```
GPR[rt] = GPR[rs] | u
```

Exceptions: None.

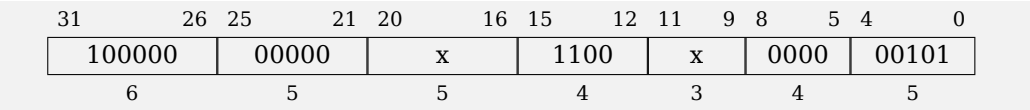
PAUSE

Assembly: `PAUSE`

Purpose: *Pause*. Pause until LL Bit is cleared.

Availability: nanoMIPS

Format:



Operation:

```
if C0.LLAddr.LLB:
    CPU.in_pause_state = True
```

The purpose of the PAUSE instruction is halt a thread (rather than entering a spin loop) when it is waiting to acquire an LL/SC lock. This is particularly useful on multi-threaded processors, since the waiting thread may be using the same instruction pipeline as the thread which currently owns the lock, and hence entering a spin loop will delay the other thread from completing its task and freeing the lock.

When a thread is in the paused state, it should not issue any instructions. The paused state will be cleared either if the LLBit for the thread gets cleared, or if the thread takes an interrupt. If an interrupt occurs, it is implementation dependent whether C0.EPC points to the PAUSE instruction or the instruction after the PAUSE.

In LL/SC lock software, the LLBit of the waiting thread will always be cleared when the thread which owns the lock does a store instruction to the lock address in order to clear the lock. Thus the paused thread will always be woken when it has another opportunity to acquire the lock. After the PAUSE instruction completes, software is expected to attempt to acquire the lock again by re-executing the LL/SC sequence.

It is legal to implement PAUSE as a NOP instruction. In this case, the behavior of LL/SC lock software will be equivalent to executing a spin loop to acquire the lock. Software using PAUSE will still work, but the benefit of having the waiting thread not consume instruction issue slots will be lost.

PAUSE is encoded as an SLL instruction with a shift value of 5, targeting GPR $0. Hence PAUSE will behave as a NOP instruction if no additional behavior beyond that of SLL is implemented.

The following assembly code example shows how the PAUSE instruction can be used to halt a thread while it is waiting to acquire an LL/SC lock.

```
acquire_lock:
        ll      t0, 0(a0)    /* Read software lock, set LLBit. */
        bnezc   t0, acquire_lock_retry /* Branch if software lock is taken. */
        addiu   t0, t0, 1    /* Set the software lock. */
```

```
        sc      t0, 0(a0)    /* Try to store the software lock. */
        bnezc   t0, 10f      /* Branch if lock acquired successfully. */
        sync

acquire_lock_retry:
        pause                /* Wait for LLBIT to clear before retrying. */
        bc      acquire_lock /* Now retry the operation. */
10:

        /* Critical Region Code */
        ...

release_lock:
        sync
        sw      zero, 0(a0)  /* Release software lock, clearing LLBIT
                                for any PAUSEd waiters */
```

Exceptions: None.

PREF/PREFE

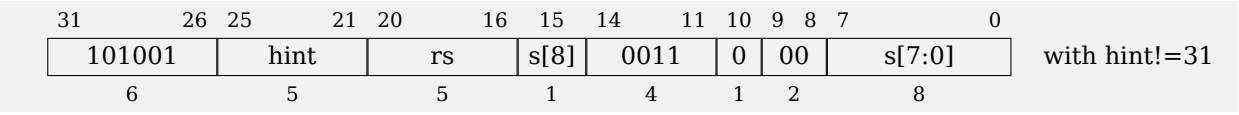
Assembly:

```
PREF  hint, offset(rs)
PREFE hint, offset(rs)
```

Purpose: *Prefetch/Prefetch using EVA addressing*. Perform a prefetch operation of type `hint` at address `$rs + offset` (register plus immediate). For PREFE, translate the virtual address as though the core is in user mode, although it is actually in kernel mode.

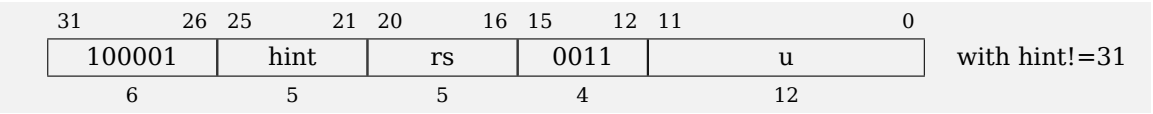
Availability: nanoMIPS, availability varies by format.

Format: PREF[S9]



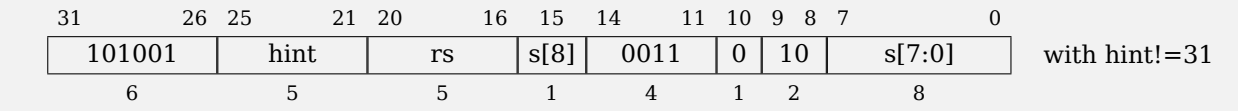

```
offset = sign_extend(s, from_nbits=9)
is_eva = False
```

Format: PREF[U12]




```
offset = u
is_eva = False
```

Format: PREFE, present when Config5.EVA=1, requires CP0 privilege.




```
offset = sign_extend(s, from_nbits=9)
is_eva = True
```

Operation:

```
if is_eva and not C0.Config5.EVA:
    raise exception('RI')

if is_eva and not IsCoprocessor0Enabled():
```

```
    raise coprocessor_exception(0)

va = effective_address(GPR[rs], offset, 'Load', eva=is_eva)

# Perform implementation dependent prefetch actions
pref(va, hint, eva=is_eva)
```

The PREF and PREFE instructions request that the processor take some action to improve program performance in accordance with the intended data usage specified by the `hint` argument. This is typically done by moving data to or from the cache at the specified address. The meanings of `hint` are as follows:

- `hint=0`: *load*
  - Use: Prefetched data is expected to be read (not modified).
  - Action: Fetch data as if for a load.
- `hint=1`: *store*
  - Use: Prefetched data is expected to be stored or modified.
  - Action: Fetch data as if for a store.
- `hint=2`: *L1 LRU hint*
  - Mark the line as LRU in the L1 cache and thus preferred for next eviction. Implementations can choose to writeback and/or invalidate the line as long as no architectural state is modified.
- `hint=3`: Reserved for Implementation
- `hint=4`: *load_streamed*
  - Use: Prefetched data is expected to be read (not modified) but not reused extensively; it ”streams” through cache.
  - Action: Fetch data as if for a load and place it in the cache so that it does not displace data prefetched as ”retained”.
- `hint=5`: *store_streamed*
  - Use: Prefetched data is expected to be stored or modified but not reused extensively; it ”streams” through cache.
  - Action: Fetch data as if for a store and place it in the cache so that it does not displace data prefetched as ”retained”.
- `hint=6`: *load_retained*
  - Use: Prefetched data is expected to be read (not modified) and reused extensively; it should be ”retained” in the cache.

  - Action: Fetch data as if for a load and place it in the cache so that it is not displaced by data prefetched as ”streamed”.

- `hint=7`: *store_retained*

  - Use: Prefetched data is expected to be stored or modified and reused extensively; it should be ”retained” in the cache.
  - Action: Fetch data as if for a store and place it in the cache so that it is not displaced by data prefetched as ”streamed”.

- `hint=8..15`: L2 operation

  - In the Release 6 architecture, hint codes 8..15 are treated the same as hint codes 0..7 respectively, but operate on the L2 cache.

- `hint=16..23`: L3 operation

  - In the Release 6 architecture, hint codes 16..23 are treated the same as hint codes 0..7 respectively, but operate on the L3 cache.

- `hint=24..30`: Reserved for Architecture

  - These hint codes are reserved in nanoMIPS and should act as a NOP. (This is not the same as the MIPSR6 behavior, where these hints give a Reserved Instruction exception). Note that `hint=31` is not listed as that encoding is decoded as a SYNCI instruction.

The action taken for a specific PREF instruction is both system and context dependent. Any action, including doing nothing, is permitted as long as it does not change architecturally visible state or alter the meaning of a program.

PREF does not cause addressing-related exceptions, including TLB exceptions. If the address specified would cause an addressing exception, the exception condition is ignored and no data movement occurs.

For cached addresses, the expected and useful action is for the processor to prefetch a block of data that includes the effective address. The size of the block and the level of the memory hierarchy it is fetched into are implementation specific.

PREF neither generates a memory operation nor modifies the state of a cache line for addresses with an uncached CCA.

Prefetch operations have no effect on cache lines that were previously locked with the CACHE instruction.

In coherent multiprocessor implementations, if the effective address uses a coherent CCA, then the instruction causes a coherent memory transaction to occur. This means a prefetch issued on one processor can cause data to be evicted from the cache in another processor.

The memory transactions which occur as a result of a PREF instruction, such as cache refill or cache writeback, obey the same ordering and completion rules as other load or store instructions.

It is implementation dependent whether a Bus Error or Cache Error exception is reported if such an error is detected as a byproduct of the action taken by the PREF instruction. Implementations are encouraged to report such errors only if there is a specific requirement for high-reliability. Note that

suppressing a bus or cache error in this case may require that the processor communicate to the system that the reference is speculative.

Hint field encodings whose function is described as ”streamed” or ”retained” convey usage intent from software to hardware. Software should not assume that hardware will always prefetch data in an optimal way. If data is to be truly retained, software should use the Cache instruction to lock data into the cache.

It is implementation dependent whether a data watch or EJTAG breakpoint exception is triggered by a prefetch instruction whose address matches the Watch register address match or EJTAG data breakpoint conditions. The preferred implementation is not to match on the prefetch instruction.

Exceptions:

Bus Error. Cache Error. Coprocessor Unusable for PREFE. Reserved Instruction for PREFE if EVA not implemented.

RDHWR

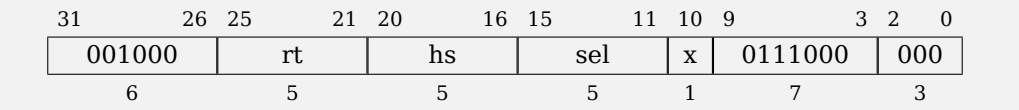
Assembly:

```
RDHWR rt, hs, sel
RDHWR rt, hs
```

Purpose: *Read Hardware Register*. Read specific CP0 privileged state (identified by `hs`, `sel`) to register `$rs`. Kernel code can enable or disable user mode RDHWR accesses by programming the enable bits in the HWREna register.

Availability: nanoMIPS, not available in NMS

Format:



Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

if not IsCoprocessor0Enabled():
   if not C0.HWREna & (1 << hs):
       raise exception('RI', 'Required HWREna bit not set')

if sel and hs != 4:
    raise exception('RI', 'sel field not supported for this hs')

if is_guest_mode():
    check_gpsi('CP0')

if hs == 0:
    GPR[rt] = C0.EBase.CPUNum

elif hs == 1:
    GPR[rt] = synci_step()

elif hs == 2:
    if is_guest_mode():
        check_gpsi('GT')
        GPR[rt] = guest_count()

    else:
        GPR[rt] = C0.Count
```

```
36
37  elif hs == 3:
38      GPR[rt] = CPU.count_resolution
39
40  elif hs == 4:
41      if not C0.Config1.PC:
42          raise exception('RI', 'Perf Counters not implemented')
43
44      GPR[rt] = read_cp0_register(25, sel)  # Performance counter register
45
46  elif hs == 5:
47      GPR[rt] = C0.Config5.XNP
48
49  elif hs == 29:
50      if not C0.Config3.ULRI:
51          raise exception('RI')
52
53      GPR[rt] = sign_extend(C0.UserLocal)
54
55  else:
56      raise exception('RI')
```

Exceptions:

Coprocessor Unusable. Reserved Instruction for unsupported register numbers. Reserved Instruction on NMS cores.

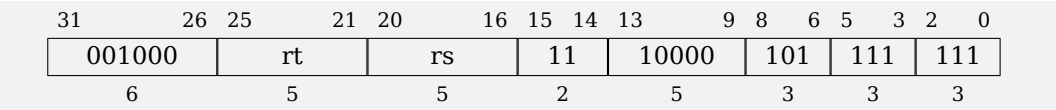
RDPGPR

Assembly: `RDPGPR rt, rs`

Purpose: *Read Previous GPR*. Write the value of register `$rs` from the previous shadow register set (SRSCtl.PSS) to register `$rt` in the current shadow register set (SRSCtl.CSS). If shadow register sets are not implemented, just copy the value from register `$rs` to register `$rt`.

Availability: nanoMIPS. Requires CP0 privilege.

Format:



Operation:

```
if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

if C0.SRSCtl.HSS > 0:
    GPR[rt] = SRS[C0.SRSCtl.PSS][rs]
else:
    GPR[rt] = GPR[rs]
```

Exceptions: Coprocessor Unusable.

RESTORE/RESTORE.JRC

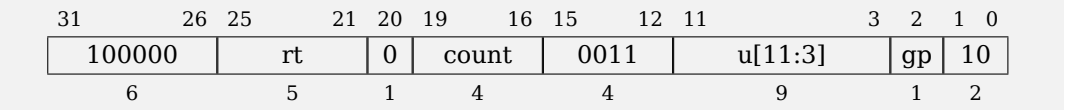
Assembly:

```
RESTORE     u[, dst1 [, dst2 [, ...]]]  # jr=0 implied
RESTORE.JRC u[, dst1 [, dst2 [, ...]]]  # jr=1 implied
```

Purpose: *Restore callee saved registers/Restore callee saved registers and Jump to Return address, Compact*. Restore registers `dst1`, [`dst2`, ...] from addresses at the top of the local stack frame (`$29 + u - 4`, `$29 + u - 8`, ...), then point register `$29` back to the caller's stack frame by adding offset `u`. For RESTORE.JRC, return from the current subroutine by jumping to the address in $31.

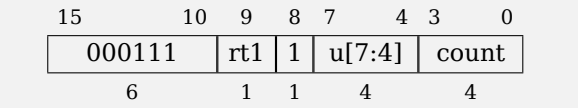
Availability: nanoMIPS, availability varies by format.

Format: RESTORE[32]



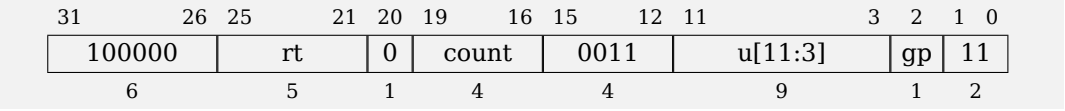

```
jr = 0
```

Format: RESTORE.JRC[16]




```
rt = 30 if rt1 == 0 else 31
gp = 0
jr = 1
```

Format: RESTORE.JRC[32], gp case not available in NMS




```
jr = 1
```

Operation:

```
if gp and C0.Config5.NMS:
    raise exception('RI')

i = 0
while i != count:
    this_rt = ( 28          if gp and (i + 1 == count) else
                rt + i      if rt + i < 32             else
                rt + i - 16                                 )

    this_offset = u - ( (i+1) << 2 )
    va = effective_address(GPR[29], this_offset, 'Load')

    if va & 3:
        raise exception('ADEL', badva=va)

    data = read_memory_at_va(va, nbytes=4)
    GPR[this_rt] = sign_extend(data, from_nbits=32)

    if this_rt == 29:
        raise UNPREDICTABLE()

    i += 1

GPR[29] = effective_address(GPR[29], u)

if jr:
    CPU.next_pc = GPR[31]
```

The purpose of the RESTORE and RESTORE.JRC instructions is to restore callee saved registers from the stack on exit from a subroutine, adjust the stack pointer register $29 to point to the caller's stack frame, and for RESTORE.JRC to return from the subroutine by jumping to the address in register $31. RESTORE/RESTORE.JRC will usually be paired with a matching SAVE instruction at the start of the subroutine, and SAVE and RESTORE take the same arguments.

The arguments for RESTORE/RESTORE.JRC consist of the amount to increment the stack by, and a list of registers to restore from to the stack. The increment is a double word aligned immediate value `u` in the range 0 to 4092. The register list can contain up to 16 consecutive registers. The count of the number of registers is encoded in the instruction's `count` field. The first register in the list is encoded in the `rt` field of the instruction.

The register list is allowed to wrap around from register $31 back to register $16 and still be considered consecutive; this allows fp ($30) and ra ($31) and the saved temporary registers s0-s7 ($16 - $23) to be restored in one instruction.

Additionally, $28 (the global pointer register) will be used in place of last register in the sequence if the 'gp' bit in the instruction encoding is set. This feature (which is not available for NMS cores) makes it possible to treat $28 as a callee saved register for environments such as Linux which require it.

The restored registers are read from memory addresses `$29 + u - 4`, `$29 + u - 8`, `$29 + u - 12`, ... etc, i.e. at the top of the local stack frame. The stack pointer is then adjusted by adding the size `u` of

the local stack frame, so that it points back to the caller's stack frame.

RESTORE.JRC with count=0 adjusts the stack pointer and jumps to the address in `$31`, but does not restore any registers from memory. Thus the RESTORE.JRC[16] instruction format can be used to provide `ADDIU $29, $29, u; JRC $31` behavior using a single 16 bit instruction.

The result of a RESTORE instruction is UNPREDICTABLE if the register list includes register `$29`.

RESTORE/RESTORE.JRC must be implemented in such a way as to make the instructions restartable, but the implementation does not need to be fully atomic. For instance, it is allowable for a RESTORE/RESTORE.JRC instruction to be aborted by an exception after a subset of the register updates have occurred. To ensure restartability, the write to GPR `$29` and the jump (for RESTORE.JRC) must be completed atomically, that is, the instruction must graduate if and only if those writes occur.

Exceptions:

Address Error. Bus Error. Reserved Instruction for gp=1 cases on NMS cores. TLB Invalid. TLB Read Inhibit. TLB Refill. Watch.

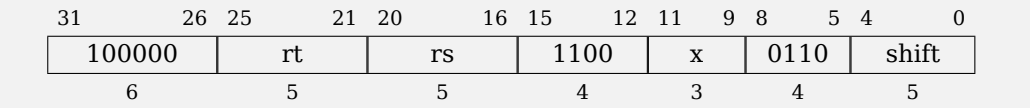
ROTR

Assembly: `ROTR rt, rs, shift`

Purpose: *Rotate Right*. Rotate the word value in register `$rs` by shift value `shift`, and place the result in register `$rt`.

Availability: nanoMIPS

Format:



Operation:

```
tmp = GPR[rs][31:0] @ GPR[rs][31:0]
result = tmp >> shift
GPR[rt] = sign_extend(result, from_nbits=32)
```

Exceptions: None.

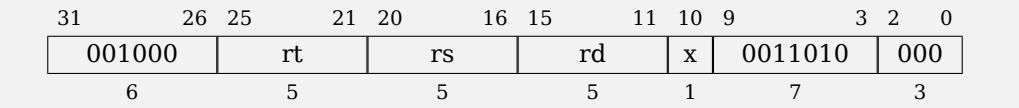
ROTRV

Assembly: `ROTRV rd, rs, rt`

Purpose: *Rotate Right Variable*. Rotate the word value in register `$rs` by the shift value contained in register `$rt`, and place the result in register `$rd`.

Availability: nanoMIPS

Format:



Operation:

```
shift = GPR[rt] & 0x1f
tmp = GPR[rs][31:0] @ GPR[rs][31:0]
result = tmp >> shift
GPR[rd] = sign_extend(result, from_nbits=32)
```

Exceptions: None.

ROTX

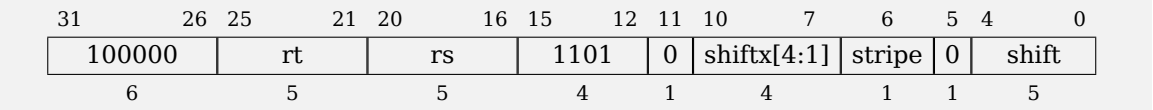
Assembly:

```
ROTX rt, rs, shift, shiftx         # stripe=0 implied
ROTX rt, rs, shift, shiftx, stripe
```

Purpose: *Rotate and eXchange*. Rotate and exchange bits in the word value in register `$rs` and place result in register `$rt`. Specific choices of the `shift`, `shiftx` and `stripe` arguments allow this instruction to perform bit and byte reordering operations including BYTEREVW, BYTEREVH, BITREVW, BITREVH and BITREVB.

Availability: nanoMIPS, not available in NMS

Format:



Operation:

```
if C0.Config5.NMS:
    raise exception('RI')

tmp0 = GPR[rs][31:0] @ GPR[rs][31:0]

tmp1 = tmp0
for i in range(47):  # 0..46
    s = shift if (i & 0b01000) else shiftx
    if stripe and not (i & 0b00100): s = ~s
    if s[4]: tmp1[i] = tmp0[i+16]

tmp2 = tmp1
for i in range(39):  # 0..38
    s = shift if (i & 0b00100) else shiftx
    if s[3]: tmp2[i] = tmp1[i+8]

tmp3 = tmp2
for i in range(35):  # 0..34
    s = shift if (i & 0b00010) else shiftx
    if s[2]: tmp3[i] = tmp2[i+4]

tmp4 = tmp3
for i in range(33):  # 0..32
    s = shift if (i & 0b00001) else shiftx
    if s[1]: tmp4[i] = tmp3[i+2]
```

```
35
36 tmp5 = tmp4
37 for i in range(32):  # 0..31
38     s = shift;
39     if s[0]: tmp5[i] = tmp4[i+1]
40
41 GPR[rt] = sign_extend(tmp5, from_nbits=32)
```

The ROTX instruction can be used to reverse elements of a selected size within blocks of a different selected size. Some example use cases are shown in the table below. The 'Result' shows the output value assuming an input value of `abcdefgh ijklmnop qrstuvwx yz012345`, where each character represents the value of a single bit.

Alias	Operation	Assembly/Result from `abcdefgh ijklmnop qrstuvwx yz012345`
BITREVW	Reverse all bits	ROTX rt, rs, 31, 0 `543210zy xwvutsrq ponmlkji hgfedcba`
BITREVH	Reverse bits in halfs	ROTX rt, rs, 15, 16 `ponmlkji hgfedcba 543210zy xwvutsrq`
BITREVB	Reverse bits in bytes	ROTX rt, rs, 7, 8, 1 `hgfedcba ponmlkji xwvutsrq 543210zy`
BYTEREVW	Reverse all bytes	ROTX rt, rs, 24, 8 `yz012345 qrstuvwx ijklmnop abcdefgh`
BYTEREVH	Reverse bytes in halfs	ROTX rt, rs, 8, 24 `ijklmnop abcdefgh yz012345 qrstuvwx`
	Reverse all nibbles	ROTX rt, rs, 28, 4 `2345yz01 uvwxqrst mnopijkl efghabcd`
	Reverse nibbles in halfs	ROTX rt, rs, 12, 20 `mnopijkl efghabcd 2345yz01 uvwxqrst`
	Reverse nibbles in bytes	ROTX rt, rs, 4, 12, 1 `efghabcd mnopijkl uvwxqrst 2345yz01`

Alias	Operation	Assembly/Result from `abcdefgh ijklmnop qrstuvwx yz012345`
	Reverse all bit pairs	ROTX rt, rs, 30, 2 `452301yz wxuvstqr opmnklij ghefcdab`
	Reverse pairs in halfs	ROTX rt, rs, 14, 18 `opmnklij ghefcdab 452301yz wxuvstqr`
	Reverse pairs in bytes	ROTX rt, rs, 6, 10, 1 `ghefcdab opmnklij wxuvstqr 452301yz`

Assembler aliases are provided for certain cases, as indicated in the table.

The MIPS32™ instructions BITSWAP and WSBH are equivalent to BITREVB and BYTEREVH respectively, and are also provided as assembler aliases to ROTX.

The ROTX instruction is designed to be implementable with minimal overhead using existing logic for the ROTR instruction. ROTR can be implemented using a barrel shifter, where the select signals for the multiplexers at each stage are the bits of the 'shift' argument. For ROTX, the mux select signals depend on the bit position as well as the stage of the shifter, and are a function of the 'shift', 'shiftx' and 'stripe' arguments.

Exceptions: Reserved Instruction on NMS cores.

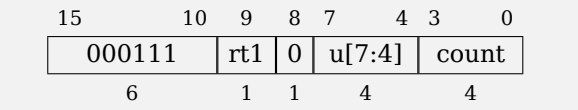
SAVE

Assembly: `SAVE u[, src1 [, src2 [, ...]]]`

Purpose: *Save callee saved registers*. Save registers `src1`, [`src2`, ...] to addresses just below the current stack pointer ($29) address and adjust the stack pointer by subtracting offset u to accommodate the saved registers and the local stack frame.

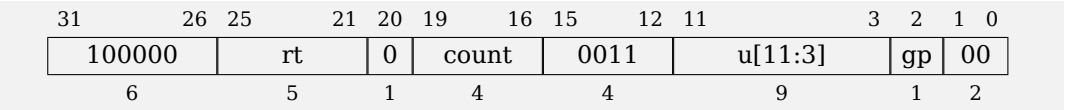
Availability: nanoMIPS, availability varies by format.

Format: SAVE[16]




```
rt = 30 if rt1 == 0 else 31
gp = 0
```

Format: SAVE[32], gp case not available in NMS



Operation:

```
if gp and C0.Config5.NMS:
    raise exception('RI')

i = 0
while i != count:
    this_rt = ( 28          if gp and (i + 1 == count) else
                rt + i      if rt + i < 32             else
                rt + i - 16                                 )

    this_offset = - ( (i+1) << 2 )
    va = effective_address(GPR[29], this_offset, 'Load')

    if va & 3:
        raise exception('ADES', badva=va)

    data = zero_extend(GPR[this_rt], from_nbits=32)
    write_memory_at_va(data, va, nbytes=4)

    i += 1

GPR[29] = effective_address(GPR[29], -u)
```

The purpose of the SAVE instruction is to save callee saved registers to the stack on entry to a subroutine, and adjust the stack pointer register ($29) to accommodate the saved registers and the subroutine's local stack frame.

The instruction specification consists of the amount to decrement the stack by, and a list of registers to save to the stack. The stack decrement is a double word aligned immediate value `u` in the range 0 to 4092. The register list can contain up to 16 consecutive registers. The count of the number of registers in the register list is encoded in the instruction's `count` field. The first register in the list is encoded in the `rt` field of the instruction.

The register list is allowed to wrap around from register $31 back to register $16 and still be considered consecutive; this allows fp ($30) and ra ($31) and the saved temporary registers s0-s7 ($16 - $23) to be saved in one instruction.

Additionally, $28 (the global pointer register) will be used in place of last register in the sequence if the 'gp' bit in the instruction encoding is set. This feature (which is not available for NMS cores) makes it possible to treat $28 as a callee saved register for environments such as Linux which require it.

The saved registers are written to memory addresses $29-4, $29-8, $29-12, ... etc, i.e. just below the current stack pointer address. The stack pointer is then adjusted by subtracting offset u, which should be chosen to accommodate the saved registers and current subroutine's local stack frame, while maintaining the required stack pointer alignment.

SAVE with count=0 adjusts the stack pointer but does not save any registers to memory. Thus the SAVE[16] instruction format can be used to provide `ADDIU16 $29, $29, -u` behavior.

SAVE must be implemented in such a way as to make the instruction restartable, but the implementation does not need to be fully atomic. For instance, it is allowable for a SAVE instruction to be aborted by an exception after a subset of the memory updates have occurred. To ensure restartability, the write to GPR `$29` must be completed atomically, that is, the instruction must graduate if and only if that write occurs.

Exceptions:

Address Error. Bus Error. Reserved Instruction for gp=1 cases on NMS cores. TLB Invalid. TLB Modified. TLB Refill. Watch.

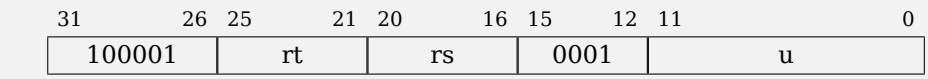
SB

Assembly: `SB rt, offset(rs)`

Purpose: *Store Byte*. Store byte from register `$rt` to memory address `$rs + offset` (register plus immediate).

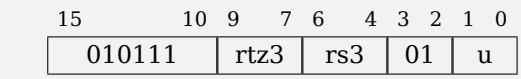
Availability: nanoMIPS

Format: SB[U12]



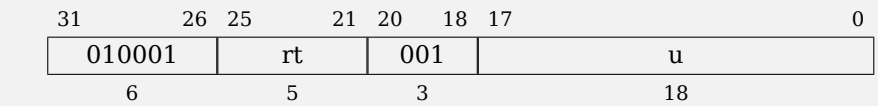

```
offset = u
```

Format: SB[16]



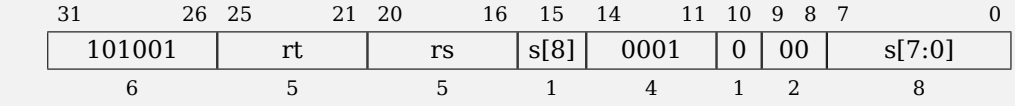

```
rt = decode_gpr(rtz3, 'gpr3.src.store')
rs = decode_gpr(rs3, 'gpr3')
offset = u
```

Format: SB[GP]




```
rs = 28
offset = u
```

Format: SB[S9]




```
offset = sign_extend(s, from_nbits=9)
```

Operation:

```
va = effective_address(GPR[rs], offset, 'Store')

data = zero_extend(GPR[rt], from_nbits=8)
write_memory_at_va(data, va, nbytes=1)
```

Exceptions:

Address Error. Bus Error. TLB Invalid. TLB Modified. TLB Refill. Watch.

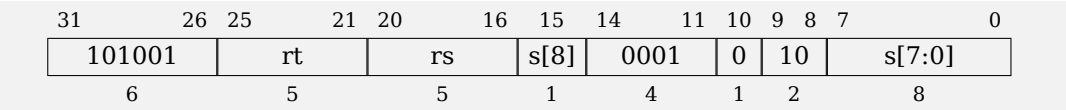
SBE

Assembly: `SBE rt, offset(rs)`

Purpose: *Store Byte using EVA addressing*. Store byte from register `$rt` to virtual address `$rs + offset`, translating the virtual address as though the core is in user mode, although it is actually in kernel mode.

Availability: nanoMIPS. Optional, present when Config5.EVA=1. Requires CP0 privilege.

Format:



Operation:

```
offset = sign_extend(s, from_nbits=9)

if not C0.Config5.EVA:
    raise exception('RI')

if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

va = effective_address(GPR[rs], offset, 'Store', eva=True)

data = zero_extend(GPR[rt], from_nbits=8)
write_memory_at_va(data, va, nbytes=1, eva=True)
```

Exceptions:

Address Error. Bus Error. Coprocessor Unusable. Reserved Instruction if EVA not implemented. TLB Invalid. TLB Modified. TLB Refill. Watch.

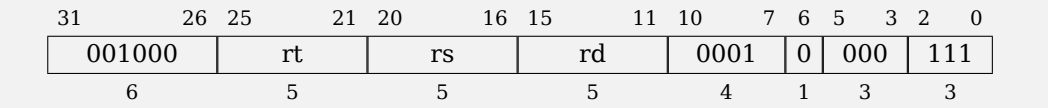
SBX

Assembly: `SBX rd, rs(rt)`

Purpose: *Store Byte indeXed*. Store byte from register `$rt` to memory address `$rt + $rs` (register plus register).

Availability: nanoMIPS, not available in NMS

Format:



Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

va = effective_address(GPR[rs], GPR[rt], 'Store')

data = zero_extend(GPR[rd], from_nbits=8)
write_memory_at_va(data, va, nbytes=1)
```

Exceptions:

Address Error. Bus Error. Reserved Instruction on NMS cores. TLB Invalid. TLB Modified. TLB Refill. Watch.

SC/SCE/SCWP/SCWPE

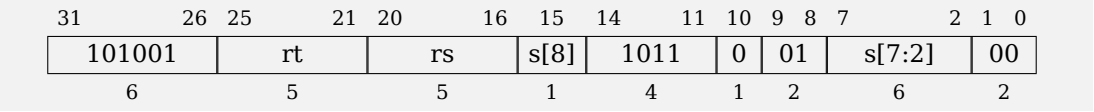
Assembly:

```
SC    rt, offset(rs)
SCE   rt, offset(rs)
SCWP  rt, ru, (rs)
SCWPE rt, ru, (rs)
```

Purpose: *Store Conditional word/Store Conditional word using EVA addressing/Store Conditional Word Pair/Store Conditional Word Pair using EVA addressing*. Store conditionally to complete atomic read-modify-write. For SC/SCE, store from register `$rt` to address `$rs + offset` (register plus offset). For SCWP/SCWPE, store from registers `$rt` and `$ru` to address `$rs`. For SCE/SCWPE, translate the virtual address as though the core is in user mode, although it is actually in kernel mode. Indicate Indicate success or failure by writing 1 or 0 respectively to `$rt`.

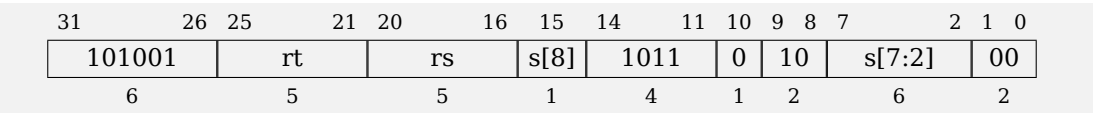
Availability: nanoMIPS, availability varies by format.

Format: SC



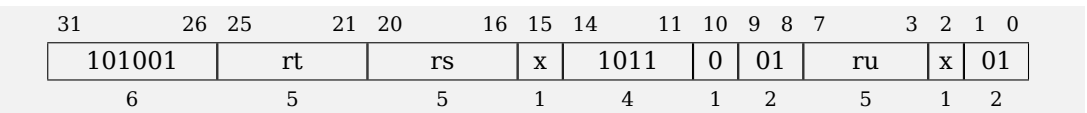

```
offset = sign_extend(s, from_nbits=9)
nbytes = 4
is_eva = False
```

Format: SCE, present when Config5.EVA=1, requires CP0 privilege.



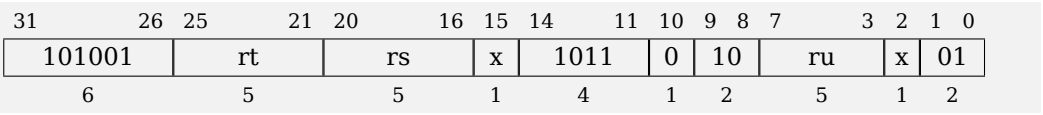

```
offset = sign_extend(s, from_nbits=9)
nbytes = 4
is_eva = True
```

Format: SCWP, required (optional on NMS cores).




```
offset = 0
nbytes = 8
is_eva = False
```

Format: SCWPE, present when Config5.EVA=1. Requires CP0 privilege.




```
offset = 0
nbytes = 8
is_eva = True
```

Operation:

```
if nbytes == 8 and C0.Config5.XNP:
    raise exception('RI', 'SCWP[E] requires word-paired support')

if is_eva and not C0.Config5.EVA:
    raise exception('RI')

va = effective_address(GPR[rs], offset, 'Store', eva=is_eva)

# Linked access must be aligned.
if va & (nbytes-1):
    raise exception('ADES', badva=va)

pa, cca = va2pa(va, 'Store', eva=is_eva)

if (cca == 2 or cca == 7) and not C0.Config5.ULS:
    raise UNPREDICTABLE('uncached CCA not synchronizable when Config5.ULS=0')
    # (Preferred behavior for non-synchronizable address is Bus Error).

if nbytes == 4:  # SC/SCE
    data = zero_extend(GPR[rt], from_nbits=32)

else:  # SCWP/SCWPE
    word0 = GPR[rt][31:0]
    word1 = GPR[ru][31:0]

    data = word0 @ word1 if C0.Config.BE else word1 @ word0

# Write this data to memory, but only if it can be done atomically with
# respect to a prior linked load. The return value indicates whether the write
# occurred.
success = write_memory(data, va, pa, cca, nbytes=nbytes, atomic=True)

if success:
    GPR[rt] = 1
else:
```

```
45      GPR[rt] = 0
46
47  C0.LLAddr.LLB = 0  # SC always clears LLbit regardless of address matches.
```

The SC, SCE, SCWP and SCWPE instructions are used to complete the atomic read-modify-write (RMW) sequence begun by a prior matching LL/LLE/LLWP/LLWPE instruction respectively. If the system can guarantee that the write to memory can be completed prior to any other modification to the targeted data since it was read by the load-linked instruction which initiated the sequence, then the write will complete and register `$rt` will be set to 1, indicating success. Otherwise, the memory write will not occur, and register `$rt` will be set to 0, indicating failure.

If any of the following events occur between a load-linked and a store conditional instruction, the store-conditional will fail:

- The store-conditional will fail if a coherent store is completed (by either the current processor, another processor, or a coherent I/O module) into the block of synchronizable physical memory containing the load-linked data. The size and alignment of the block is implementation-dependent, but it is at least one word and at most the minimum page size. Typically, the synchronizable block size is the size of the largest cache line in use.
- The store-conditional will fail if an ERET instruction has been executed since the preceding load-linked instruction. (Note that nanoMIPS™ also includes the ERETNC instruction, which will not cause the store-conditional instruction to fail.)

If any of the following events occur between a load-linked and a store conditional instruction, the store-conditional may fail when it would otherwise have succeeded. Portable programs should not cause any of these events:

- The store-conditional may fail if a load or store is executed on a processor executing a load-linked/store-conditional sequence, and that load or store is not to the block of synchronizable physical memory containing the load-linked data. This is because the load or store may cause the load-linked data to be evicted from the cache.
- The store-conditional may fail if any PREF instruction is executed a processor executing a load-linked/store-conditional sequence, due to the possibility of the PREF causing a cache eviction.
- The store-conditional may fail on coherent multi-processor systems if a non-coherent store is executed during a load-linked/store-conditional sequence and that store is to the block of synchronizable physical memory containing the linked data.
- The store-conditional may fail if the instructions executed starting with the load-linked instruction and ending with the store-conditional instruction do not lie in a 2048-byte contiguous region of virtual memory. (The region does not have to be aligned, other than the alignment required for instruction words.)
- The store-conditional may fail if a CACHE operation is carried out during the load-linked/store-conditional sequence, due to the possibility of modifying or evicting the line containing the linked data. In addition, non-local CACHE operations may cause a store-conditional instruction to fail on either the local processor or on the remote processor in multiprocessor or multi-threaded systems.

The store-conditional must not fail as a result of any of the following events:

- The store-conditional must not fail as a result of a load that executes on the processor executing a load-linked/store-conditional sequence if the load targets the block of synchronizable physical memory containing the load-linked data.

The outcome of the store-conditional is not predictable (it may succeed or fail) under any of the following conditions:

- The store-conditional result is unpredictable if the store-conditional was not preceded by a matching load-linked instruction. SC must be preceded by LL, SCE must be preceded by LLE, SCWP must be preceded by LLWP, and SCWPE must be preceded by LLWPE.
- The store-conditional result is unpredictable if the load-linked and store-conditional instructions do not target identical virtual addresses, physical addresses and CCAs.
- The store-conditional result is unpredictable if the targeted memory location is not synchronizable. A synchronizable memory location is one that is associated with the state and logic necessary to track RMW atomicity. Whether a memory location is synchronizable depends on the processor and system configurations, and on the memory access type used for the location.
- The store-conditional result is unpredictable if the memory access does not use a CCA which supports atomic RMW for the targeted address.
  - For uniprocessor systems, a cached noncoherent or cached coherent CCA must be used, or additionally an uncached CCA can be used in the case that Config5.ULS=1.
  - For multi-processor systems or systems containing coherent I/O devices, a cached coherent CCA must be used, or additionally an uncached CCA can be used in the case that Config5.ULS=1.

When Config5.ULS=1, uncached load-linked/store-conditional operations are supported, with the following additional constraints:

- The result of a store-conditional which is part of an uncached load-linked/store conditional sequence is unpredictable if during the sequence a local or remote CPU accesses the block of memory containing the targeted data using any other CCA than that used by the load-linked and store-conditional instructions.
- The result of an uncached load-linked/store-conditional sequence is only predicable if it targets an address in the system which supports uncached RMW accesses. In particular, the system must implement a ”monitor”, which is responsible determining whether or not the address can be updated atomically with respect to the prior linked load. In response to a store-conditional instruction, the monitor updates memory where appropriate and communicates the result to the processor that initiated the sequence. It is implementation dependent as to what form the monitor takes. The recommended response for load-linked/store-conditional instructions which target a non-synchronizable uncached address is that the sub-system report a Bus Error to the processor.

- Same processor uncached stores will cause an uncached load-linked/store-conditional sequence to fail if the store address matches that of the sequence.
- A PAUSE instruction is no-op'd when it is preceded by an uncached load-linked instruction. This is because the event which would wake the CPU from the paused state may only be visible to the external monitor, not to the local processor.
- The rules for uncached load-linked/store-conditional atomic operation apply to any uncached CCA including UCA (UnCached Accelerated). An implementation that supports UCA must guarantee that a store-conditional instruction does not participate in store gathering and that it ends any gathering initiated by stores preceding the SC in program order when the SC address coincides with a gathering address.

The effective address of a store-conditional operation must be naturally-aligned, i.e. word aligned for SC and SCE, and double-word aligned for SCWP and SCWPE: Otherwise an address exception occurs.

The following assembly code shows a possible usage of LL and SC to atomically update a memory location:

```
L1:
        ll      t1, 0(t0)     # Load counter.
        addiu   t2, t1, 1     # Increment.
        sc      t2, 0(t0)     # Try to store, checking for atomicity.
        beqc    t2, 0, L1     # If not atomic (0), try again.
```

Exceptions between the load-linked and store-conditional instructions cause the store-conditional to fail, so instructions which can cause persistent exceptions must not be used within the load-linked/store-conditional sequence. Examples of instructions which must be avoided are are arithmetic operations that trap, system calls, and floating point operations that trap or require software emulation assistance.

Load-linked and store-conditional must function correctly on a single processor for cached noncoherent memory so that parallel programs can be run on uniprocessor systems that do not support cached coherent memory access types.

Support for the paired word instructions SCWP/SCWPE is indicated by the Config5.XNP bit. Paired word support is required for nanoMIPS™ cores, except for NMS cores, where it is optional.

Exceptions:

Address Error. Bus Error. Coprocessor Unusable for SCE/SCWPE. Reserved Instruction for SCE/SCWPE if EVA not implemented. Reserved Instruction for SCWP/SCWPE if load linked pair not implemented. TLB Invalid. TLB Modified. TLB Refill. Watch.

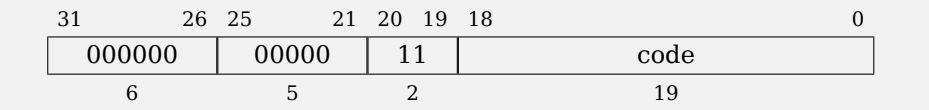
SDBBP

Assembly: `SDBBP code`

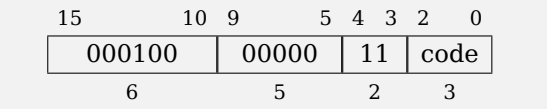
Purpose: *Software Debug Breakpoint.* Cause a Software Debug Breakpoint exception.

Availability: nanoMIPS. Optional, present when Debug implemented.

Format: SDBBP[32]



Format: SDBBP[16]



Operation:

```
if C0.Config1.EP == 0:
    raise exception('RI', 'Debug not implemented')

if C0.Config5.SBRI and EffectiveKSU() != 0:
    raise exception('RI', 'SBRI exception')

if Root.C0.Config5.SBRI and is_guest_mode():
    raise exception('RI', 'Root SBRI exception', g=False)

debug_exception('BP')
Root.C0.Debug.DBp = 1
raise EXCEPTION()
```

Exceptions:

Software Debug Breakpoint. Reserved Instruction if Debug not implemented.

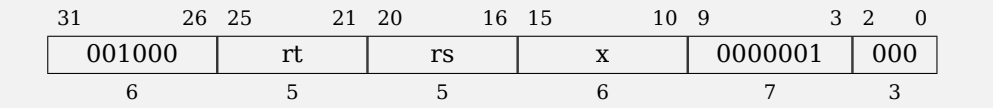
SEB

Assembly: `SEB rt, rs`

Purpose: *Sign Extend Byte*. Take the lower byte of the value in register `$rs`, sign extend it, and place the result in register `$rt`.

Availability: nanoMIPS, not available in NMS

Format:



Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

GPR[rt] = sign_extend(GPR[rs], from_nbits=8)
```

Exceptions: Reserved Instruction on NMS cores.

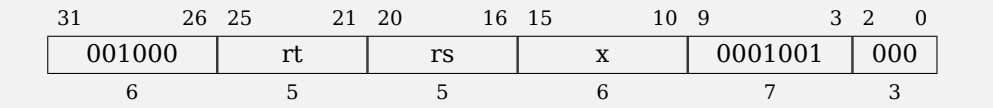
SEH

Assembly: `SEH rt, rs`

Purpose: *Sign Extend Half*. Take the lower halfword of the value in register `$rs`, sign extend it, and place the result in register `$rt`.

Availability: nanoMIPS

Format:



Operation:

```
GPR[rt] = sign_extend(GPR[rs], from_nbits=16)
```

Exceptions: None.

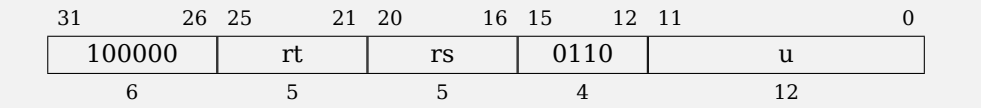
SEQI

Assembly: `SEQI rt, rs, u`

Purpose: *Set on Equal to Immediate*. Set the register `$rt` to 1 if register `$rs` is equal to immediate value `u`, and 0 otherwise.

Availability: nanoMIPS

Format:



Operation:

```
GPR[rt] = 1 if GPR[rs] == u else 0
```

Exceptions: None.

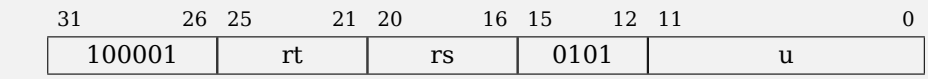
SH

Assembly: `SH rt, offset(rs)`

Purpose: *Store Half.* Store halfword from register `$rt` to memory address `$rs + offset` (register plus immediate).

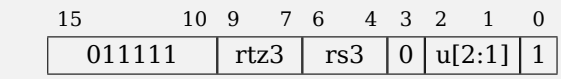
Availability: nanoMIPS

Format: SH[U12]



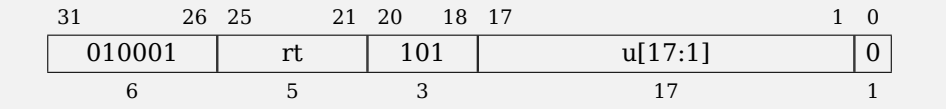

```
offset = u
```

Format: SH[16]



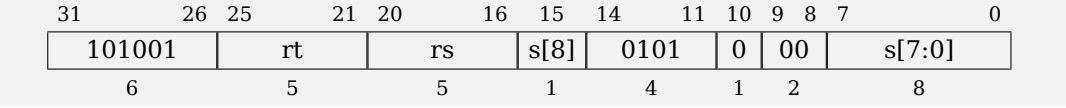

```
rt = decode_gpr(rtz3, 'gpr3.src.store')
rs = decode_gpr(rs3, 'gpr3')
offset = u
```

Format: SH[GP]




```
rs = 28
offset = u
```

Format: SH[S9]




```
offset = sign_extend(s, from_nbits=9)
```

Operation:

```
va = effective_address(GPR[rs], offset, 'Store')

data = zero_extend(GPR[rt], from_nbits=16)
write_memory_at_va(data, va, nbytes=2)
```

Exceptions:

Address Error. Bus Error. TLB Invalid. TLB Modified. TLB Refill. Watch.

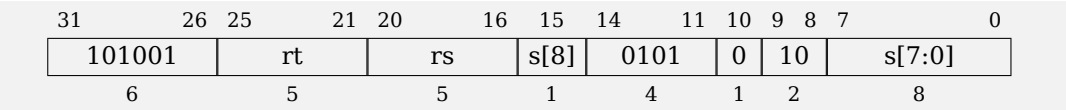
SHE

Assembly: `SHE rt, offset(rs)`

Purpose: *Store Half using EVA addressing*. Store halfword from register `$rt` to virtual address `$rs + offset`, translating the virtual address as though the core is in user mode, although it is actually in kernel mode.

Availability: nanoMIPS. Optional, present when Config5.EVA=1. Requires CP0 privilege.

Format:



Operation:

```
offset = sign_extend(s, from_nbits=9)

if not C0.Config5.EVA:
    raise exception('RI')

if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

va = effective_address(GPR[rs], offset, 'Store', eva=True)

data = zero_extend(GPR[rt], from_nbits=16)
write_memory_at_va(data, va, nbytes=2, eva=True)
```

Exceptions:

Address Error. Bus Error. Coprocessor Unusable. Reserved Instruction if EVA not implemented. TLB Invalid. TLB Modified. TLB Refill. Watch.

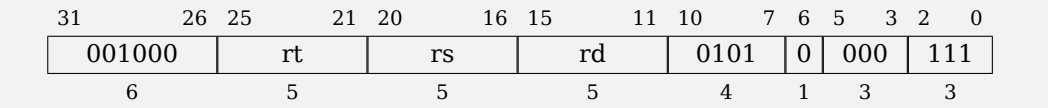
SHX

Assembly: `SHX rd, rs(rt)`

Purpose: *Store Half indeXed*. Store halfword from register `$rt` to memory address `$rt` + `$rs` (register plus register).

Availability: nanoMIPS, not available in NMS

Format:



Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

va = effective_address(GPR[rs], GPR[rt], 'Store')

data = zero_extend(GPR[rd], from_nbits=16)
write_memory_at_va(data, va, nbytes=2)
```

Exceptions:

Address Error. Bus Error. Reserved Instruction on NMS Cores. TLB Invalid. TLB Modified. TLB Refill. Watch.

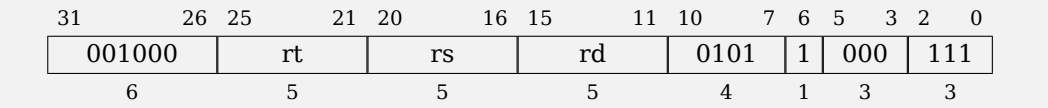
SHXS

Assembly: `SHXS rd, rs(rt)`

Purpose: *Store Half indeXed Scaled*. Store halfword from register `$rt` to memory address `$rt + 2*$rs` (register plus scaled register).

Availability: nanoMIPS, not available in NMS

Format:



Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

va = effective_address(GPR[rs]<<1, GPR[rt], 'Store')

data = zero_extend(GPR[rd], from_nbits=16)
write_memory_at_va(data, va, nbytes=2)
```

Exceptions:

Address Error. Bus Error. Reserved Instruction on NMS Cores. TLB Invalid. TLB Modified. TLB Refill. Watch.

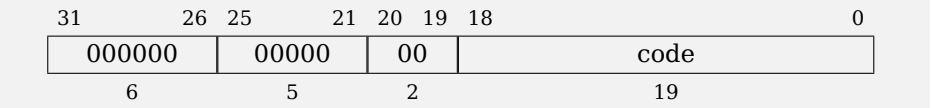
SIGRIE

Assembly: `SIGRIE code`

Purpose: *Signal Reserved Instruction Exception.*

Availability: nanoMIPS

Format:



Operation:

```
raise exception('RI')
```

Exceptions: Reserved Instruction.

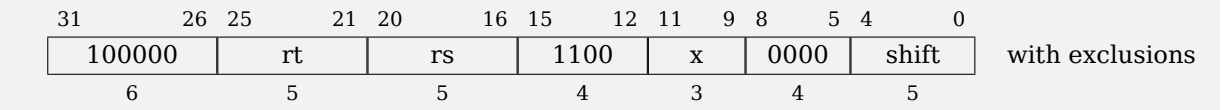
SLL

Assembly: `SLL rt, rs, shift`

Purpose: *Shift Left Logical*. Left shift word value in register `$rs` by amount `shift`, and place the result in register `$rt`.

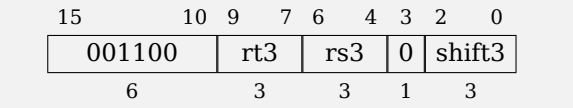
Availability: nanoMIPS

Format: SLL[32]



NOP[32], EHB, PAUSE, and SYNC instruction formats overlap SLL[32]. Opcodes matching those instruction formats should be processed according to the description of those instructions, not as SLL[32].

Format: SLL[16]




```
rt = decode_gpr(rt3, 'gpr3')
rs = decode_gpr(rs3, 'gpr3')
shift = 8 if shift3 == 0 else shift3
```

Operation:

```
result = GPR[rs] << shift
GPR[rt] = sign_extend(result, from_nbits=32)
```

Exceptions: None.

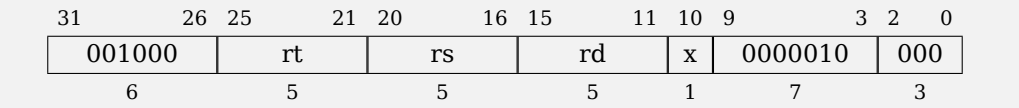
SLLV

Assembly: `SLLV rd, rs, rt`

Purpose: *Shift Left Logical Variable*. Left shift word value in register `$rs` by shift amount in register `$rt`, and place the result in register `$rd`.

Availability: nanoMIPS

Format:



Operation:

```
shift = GPR[rt] & 0x1f
result = GPR[rs] << shift
GPR[rd] = sign_extend(result, from_nbits=32)
```

Exceptions: None.

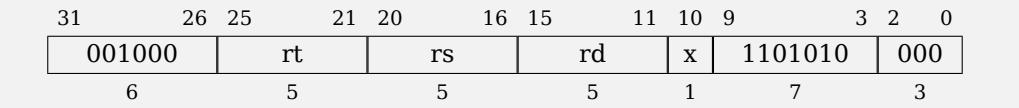
SLT

Assembly: `SLT rd, rs, rt`

Purpose: *Set on Less Than*. Set the register `$rd` to 1 if signed register `$rs` is less than signed register `$rt`, and 0 otherwise.

Availability: nanoMIPS

Format:



Operation:

```
GPR[rd] = 1 if GPR[rs] < GPR[rt] else 0
```

Exceptions: None.

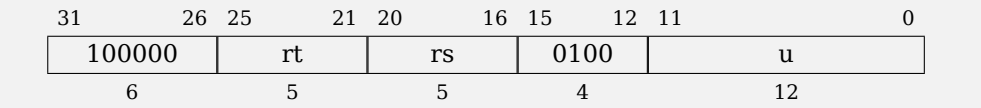
SLTI

Assembly: `SLTI rt, rs, u`

Purpose: *Set on Less Than Immediate*. Set the register `$rt` to 1 if the signed value in register `$rs` is less than immediate `u`, and 0 otherwise.

Availability: nanoMIPS

Format:



Operation:

```
GPR[rt] = 1 if GPR[rs] < u else 0
```

Exceptions: None.

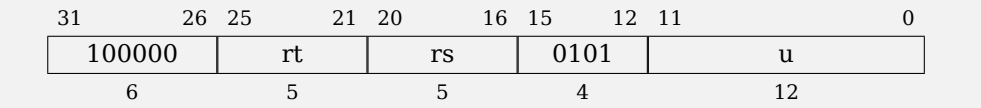
SLTIU

Assembly: `SLTIU rt, rs, u`

Purpose: *Set on Less Than Immediate, Unsigned*. Set the register `$rt` to 1 if the unsigned value in register `$rs` is less than immediate `u`, and 0 otherwise.

Availability: nanoMIPS

Format:



Operation:

```
GPR[rt] = 1 if unsigned(GPR[rs]) < u else 0
```

Exceptions: None.

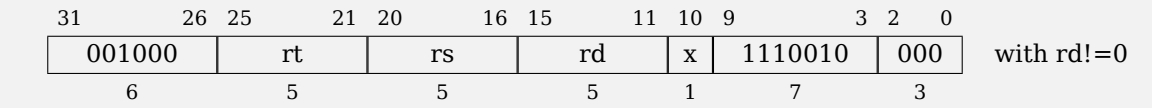
SLTU

Assembly: `SLTU rd, rs, rt`

Purpose: *Set on Less Than, Unsigned*. Set the register `$rd` to 1 if unsigned register `$rs` is less than unsigned register `$rt`, and 0 otherwise.

Availability: nanoMIPS

Format:



Operation:

```
GPR[rd] = 1 if unsigned(GPR[rs]) < unsigned(GPR[rt]) else 0
```

SLTU encodings with rd=0 are used for the DVP and EVP instructions. DVP and EVP are required to behave as NOPs on cores without Virtual Processor (VP) support. This means that no DVP/EVP special casing is required in hardware for non-VP cores, since a SLTU instruction writing to $0 naturally behaves as a NOP.

Exceptions: None.

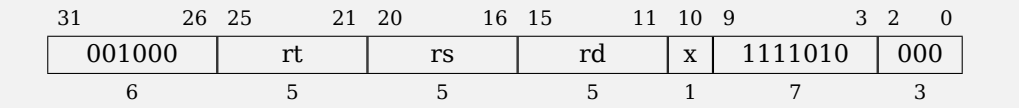
SOV

Assembly: `SOV rd, rs, rt`

Purpose: *Set on Overflow*. Set the register `$rd` to 1 if the signed addition of registers `$rs` and `$rt` overflows 32 bits, and 0 otherwise.

Availability: nanoMIPS

Format:



Operation:

```
sum = GPR[rs] + GPR[rt]
GPR[rd] = 1 if overflows(sum, nbits=32) else 0
```

Exceptions: None.

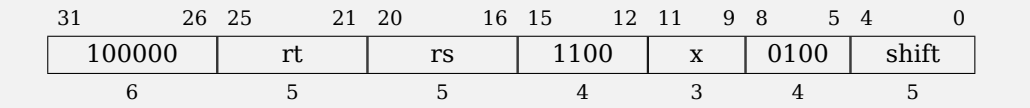
SRA

Assembly: `SRA rt, rs, shift`

Purpose: *Shift Right Arithmetic*. Right shift word value in register `$rs` by amount `shift`, duplicating the sign bit (bit 31) in the emptied bits. Place the result in register `$rt`.

Availability: nanoMIPS

Format:



Operation:

```
GPR[rt] = GPR[rs] >> shift
```

Exceptions: None.

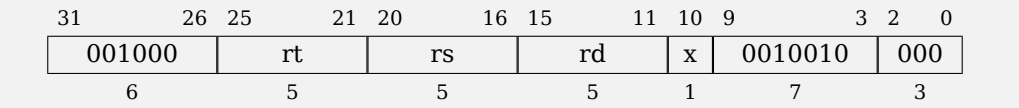
SRAV

Assembly: `SRAV rd, rs, rt`

Purpose: *Shift Right Arithmetic Variable*. Right shift word value in register `$rs` by shift amount in register `$rt`, duplicating the sign bit (bit 31) in the emptied bits. Place the result in register `$rd`.

Availability: nanoMIPS

Format:



Operation:

```
shift = GPR[rt] & 0x1f
GPR[rd] = GPR[rs] >> shift
```

Exceptions: None.

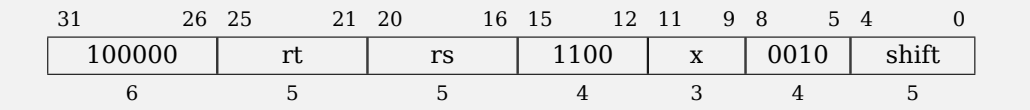
SRL

Assembly: `SRL rt, rs, shift`

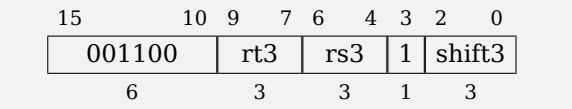
Purpose: *Shift Right Logical*. Right shift word value in register `$rs` by amount `shift`, filling the emptied bits with zeroes. Place the result in register `$rt`.

Availability: nanoMIPS

Format: SRL[32]



Format: SRL[16]




```
rt = decode_gpr(rt3, 'gpr3')
rs = decode_gpr(rs3, 'gpr3')
shift = 8 if shift3 == 0 else shift3
```

Operation:

```
result = zero_extend(GPR[rs], from_nbits=32) >> shift
GPR[rt] = sign_extend(result, from_nbits=32)
```

Exceptions: None.

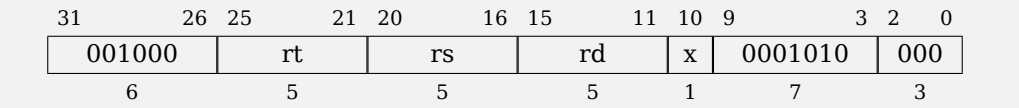
SRLV

Assembly: `SRLV rd, rs, rt`

Purpose: *Shift Right Logical Variable*. Right shift word value in register `$rs` by shift amount in register `$rt`, filling the emptied bits with zeros. Place the result in register `$rd`.

Availability: nanoMIPS

Format:



Operation:

```
shift = GPR[rt] & 0x1f

result = zero_extend(GPR[rs], from_nbits=32) >> shift
GPR[rd] = sign_extend(result, from_nbits=32)
```

Exceptions: None.

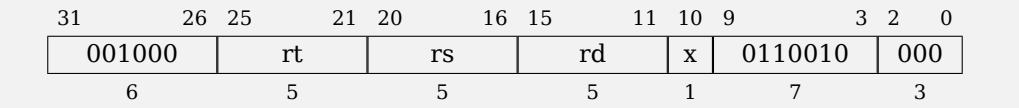
SUB

Assembly: `SUB rd, rs, rt`

Purpose: *Subtract.* Subtract the 32-bit signed integer in register `$rt` from the 32-bit signed integer in register `$rs`, placing the 32-bit result in register `$rd`, and trapping on overflow.

Availability: nanoMIPS, not available in NMS

Format:



Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

result = GPR[rs] - GPR[rt]
if overflows(result, nbits=32):
    raise exception('OV')

GPR[rd] = sign_extend(result, from_nbits=32)
```

Exceptions: None.

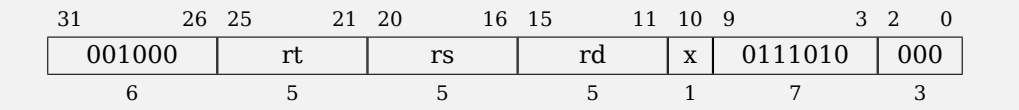
SUBU

Assembly: `SUBU rd, rs, rt`

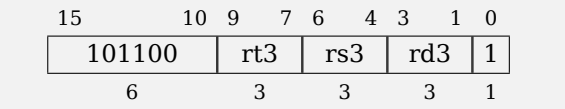
Purpose: *Subtract (Untrapped)*. Subtract the 32-bit integer in register `$rt` from the 32-bit integer in register `$rs`, placing the 32-bit result in register `$rd`, and not trapping on overflow.

Availability: nanoMIPS

Format: SUBU[32]



Format: SUBU[16]




```
rd = decode_gpr(rd3, 'gpr3')
rs = decode_gpr(rs3, 'gpr3')
rt = decode_gpr(rt3, 'gpr3')
```

Operation:

```
result = GPR[rs] - GPR[rt]
GPR[rd] = sign_extend(result, from_nbits=32)
```

Exceptions: None.

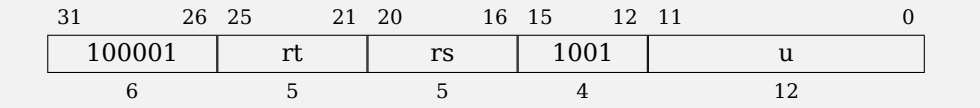
SW

Assembly: `SW rt, offset(rs)`

Purpose: *Store Word*. Store word from register `$rt` to memory address `$rs + offset` (register plus immediate).

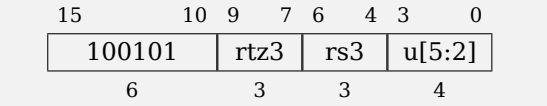
Availability: nanoMIPS, availability varies by format.

Format: SW[U12]



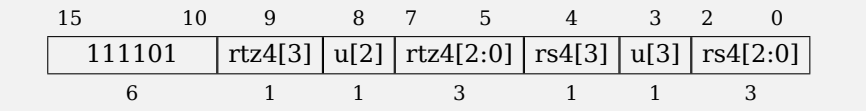

```
offset = u
```

Format: SW[16]



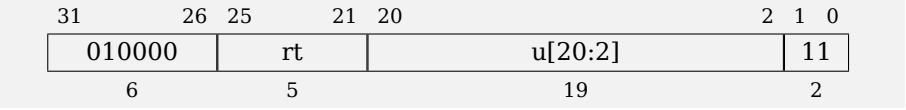

```
rt = decode_gpr(rtz3, 'gpr3.src.store')
rs = decode_gpr(rs3, 'gpr3')
offset = u
```

Format: SW[4X4], not available in NMS



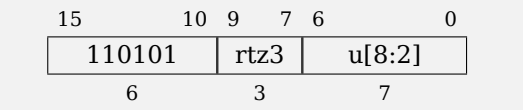

```
if C0.Config5.NMS == 1:
    raise exception('RI')

rt = decode_gpr(rtz4[3] @ rtz4[2:0], 'gpr4.zero')
rs = decode_gpr(rs4[3] @ rs4[2:0], 'gpr4')
offset = u
```

Format: SW[GP]



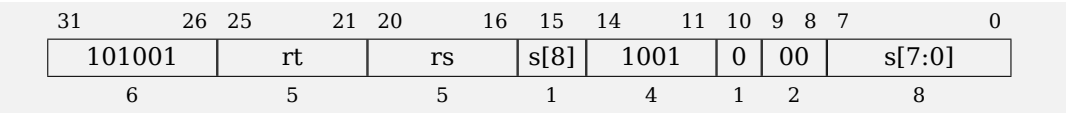
```
rs = 28
offset = u
```

Format: SW[GP16]



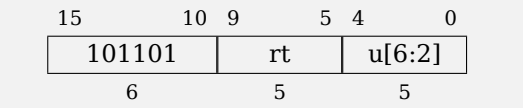

```
rt = decode_gpr(rtz3, 'gpr3.src.store')
rs = 28
offset = u
```

Format: SW[S9]




```
offset = sign_extend(s, from_nbits=9)
```

Format: SW[SP]




```
rs = 29
offset = u
```

Operation:

```
va = effective_address(GPR[rs], offset, 'Store')

data = zero_extend(GPR[rt], from_nbits=32)
write_memory_at_va(data, va, nbytes=4)
```

Exceptions:

Address Error. Bus Error. Reserved Instruction for SW[4X4] format on NMS Cores. TLB Invalid. TLB Modified. TLB Refill. Watch.

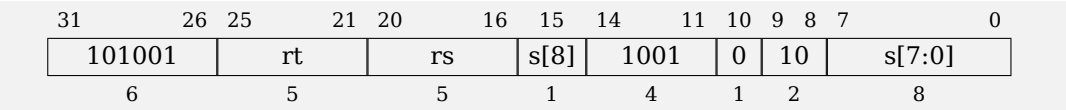
SWE

Assembly: `SWE rt, offset(rs)`

Purpose: *Store Word using EVA addressing*. Store word from register `$rt` to virtual address `$rs + offset`, translating the virtual address as though the core is in user mode, although it is actually in kernel mode.

Availability: nanoMIPS. Optional, present when Config5.EVA=1. Requires CP0 privilege.

Format:



Operation:

```
offset = sign_extend(s, from_nbits=9)

if not C0.Config5.EVA:
    raise exception('RI')

if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

va = effective_address(GPR[rs], offset, 'Store', eva=True)

data = zero_extend(GPR[rt], from_nbits=32)
write_memory_at_va(data, va, nbytes=4, eva=True)
```

Exceptions:

Address Error. Bus Error. Coprocessor Unusable. Reserved Instruction if EVA not implemented. TLB Invalid. TLB Modified. TLB Refill. Watch.

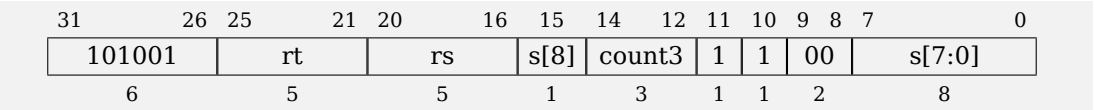
SWM

Assembly: `SWM rt, offset(rs), count`

Purpose: *Store Word Multiple*. Store `count` words of data from registers `$rt`, `$(rt+1)`, ..., `$(rt+count-1)` to consecutive memory addresses starting at `$rs + offset` (register plus immediate).

Availability: nanoMIPS, not available in NMS

Format:




```
offset = sign_extend(s, from_nbits=9)
count = 8 if count3 == 0 else count3
```

Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

i = 0
while i != count:
    this_rt = ( 0           if rt == 0     else
                rt + i      if rt + i < 32 else
                rt + i - 16                   )

    this_offset = offset + (i<<2)
    va = effective_address(GPR[rs], this_offset, 'Store')

    data = zero_extend(GPR[this_rt], from_nbits=32)
    write_memory_at_va(data, va, nbytes=4)

    i += 1
```

SWM stores `count` words from sequentially numbered registers to sequential memory addresses. After storing $31, the sequence of registers continues from $16. If rt=0, then $0 is stored for all `count` steps of the instruction. Some example encodings of the register list are:

- rt=15, count=3: loads [$15, $16, $17]
- rt=31, count=3: saves [$31, $16, $17]
- rt=0, count=3: saves [$0, $0, $0].

If a TLB exception or interrupt occurs during the execution of this instruction, a subset of the required memory updates may have occurred. A full restart of the instruction will be performed on return from the exception.

Exceptions:

Address Error. Bus Error. Reserved Instruction on NMS cores. TLB Invalid. TLB Modified. TLB Refill. Watch.

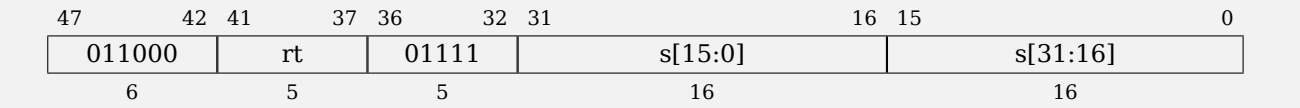
SWPC

Assembly: `SWPC rt, address`

Purpose: *Store Word PC relative*. Store word from register `$rt` to PC relative address `address`.

Availability: nanoMIPS, not available in NMS

Format: SWPC[48]




```
offset = sign_extend(s, from_nbits=32)
```

Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

address = effective_address(CPU.next_pc, offset, 'Store')

data = zero_extend(GPR[rt], from_nbits=32)
write_memory_at_va(data, address, nbytes=4)
```

Exceptions:

Address Error. Bus Error. Reserved Instruction on NMS cores. TLB Invalid. TLB Modified. TLB Refill. Watch.

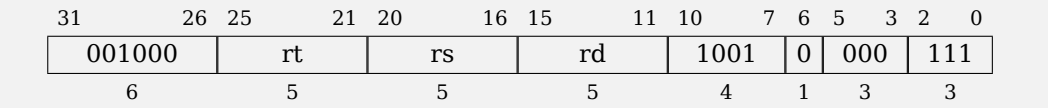
SWX

Assembly: `SWX rd, rs(rt)`

Purpose: *Store Word indeXed*. Store word from register `$rt` to memory address `$rt + $rs` (register plus register).

Availability: nanoMIPS, not available in NMS

Format:



Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

va = effective_address(GPR[rs], GPR[rt], 'Store')

data = zero_extend(GPR[rd], from_nbits=32)
write_memory_at_va(data, va, nbytes=4)
```

Exceptions:

Address Error. Bus Error. Reserved Instruction on NMS cores. TLB Invalid. TLB Modified. TLB Refill. Watch.

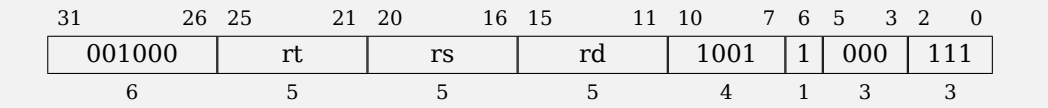
SWXS

Assembly: `SWXS rd, rs(rt)`

Purpose: *Store Word indeXed Scaled*. Store word from register `$rt` to memory address `$rt + 4*$rs` (register plus scaled register).

Availability: nanoMIPS, not available in NMS

Format:



Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

va = effective_address(GPR[rs]<<2, GPR[rt], 'Store')

data = zero_extend(GPR[rd], from_nbits=32)
write_memory_at_va(data, va, nbytes=4)
```

Exceptions:

Address Error. Bus Error. Reserved Instruction on NMS cores. TLB Invalid. TLB Modified. TLB Refill. Watch.

SYNC

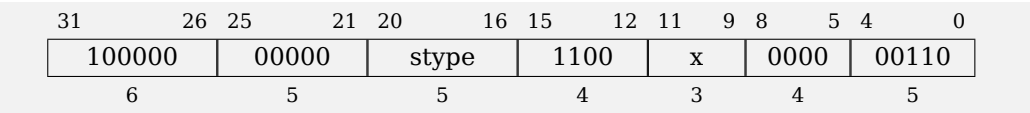
Assembly:

```
SYNC stype
SYNC        # stype=0 implied
```

Purpose: *Sync*. Impose ordering constraints of type `stype` on prior and subsequent memory operations.

Availability: nanoMIPS

Format:



Operation:

```
10 sync_memory_access(stype)
```

The SYNC instruction is used to order loads and stores for shared memory, and also to order operations with respect to the global invalidate instructions GINVI and GINVT. The following types of ordering guarantees are available with different stypes.

- *Completion Barriers*: A completion barrier provides a guarantee that any of the specified memory instructions before the SYNC are completed and globally performed before any of the specified memory instructions after the SYNC are performed to any extent. Loads are completed when the destination register is written. Stores are completed when the stored value is visible to every other processor in the system.
- *Ordering Barriers*: An ordering barrier provides a guarantee in the system that any specified memory instructions before the SYNC are ordered before any of the specified memory instructions after the SYNC. The ordering SYNC is considered complete when the memory instructions before and after the SYNC are guaranteed thereafter to retain their order relative to the SYNC, i.e. when it is guaranteed that all specified memory instructions before the SYNC will be globally performed before any of the specified memory accesses after the SYNC are performed to any extent. It is helpful to think of a global ordering point in a coherence domain, which is a point where once an instruction reaches, it can be guaranteed to retain its order relative to any memory instruction that reaches the point after it. The ordering SYNC thus can not complete before all older specified memory instructions reach the global ordering point.

The following table shows the behavior of the SYNC instruction for each `stype` value. Operation types listed in the 'What *reaches* before' column are subject to a pre-SYNC *ordering* barrier: such operations, when younger, must reach the global ordering point before the SYNC instruction completes. Operation types listed in the 'What *reaches* after' column are subject to a post-SYNC *ordering* barrier: such operations, when older, must reach the global ordering point only after the SYNC instruction completes. Operation types listed in the 'What *completes* before' column are subject to a *completion* barrier, that is, they must be globally performed when the SYNC instruction completes.

stype	Name	What *reaches* before	What *reaches* after	What *completes* before	Availability
0x0	SYNC	Loads, Stores	Loads, Stores	Loads, Stores	Required.
0x1-0x3					Impl./vendor specific.
0x4	SYNC_WMB	Stores	Stores		Optional.
0x5-0xF					Impl./vendor specific.
0x10	SYNC_MB	Loads, Stores	Loads, Stores		Optional.
0x11	SYNC_ACQUIRE	Loads	Loads, Stores		Optional.
0x12	SYNC_RELEASE	Loads, Stores	Loads		Optional.
0x13	SYNC_RMB	Loads	Loads		Optional.
0x14	SYNC_GINV	Loads, Stores	Loads, Stores	GINVI, GINVT, SYNCI	Config5.GI=2,3.
0x15-0x1F					Reserved for Architecture.

SYNC barriers affect only uncached and cached coherent loads and stores and do not affect the order in which instruction fetches are performed. For the purposes of this description, the CACHE, PREF and SYNCI instructions are treated as loads and stores. In addition, the optional Global Invalidate instructions are synchronizable through SYNC (stype=0x14).

The effect of SYNC on the global order of loads and stores for memory access types other than uncached and cached coherent is UNPREDICTABLE.

A completion barrier may have an adverse impact on performance compared to an ordering barrier due to the constraint of completion. An implementation may optimize the ordering of memory instructions such that the ordering barrier completes before a completion barrier under the same circumstance. The magnitude of the impact is implementation-dependent but an implementation must ensure that an ordering barrier is not worse performing than the equivalent completion barrier. Software thus needs to use completion and ordering barriers for the appropriate conditions.

An stype of 0 is used to define the SYNC instruction with completion barrier semantics. Non-zero values of stype may be defined by the architecture or specific implementations to perform synchronization behaviors that are less complete than that of stype=0. If an implementation does not use one of these non-zero values to define a different synchronization behavior, then that non-zero value of stype must map to a completion barrier. This allows software written for an implementation with a lighter-weight barrier to work on another implementation which only implements the stype=0 completion barrier.

The Acquire and Release barrier types are used to minimize the memory ordering that must be maintained and still have software synchronization work.

A completion barrier is required, potentially in conjunction with an EHB instruction, to guarantee that memory reference results are visible across operating mode changes. For example, a completion barrier is required on some implementations on entry to and exit from Debug Mode to guarantee that memory effects are handled correctly.

If Global Invalidate instructions are supported, then SYNC (stype=0x14) acts as a completion barrier with respect to any preceding GINVI or GINVT instructions. This SYNC instruction is globalized and

only completes if all preceding GINVI or GINVT operations related to the same program have completed in the system. (Any references to GINVT also imply GINVGT, available in a virtualized MIPS system.)

A system that implements the Global Invalidates also requires that the completion of SYNC (stype=0x14) be constrained by legacy SYNCI operations. Thus SYNC (stype=0x14) can also be used to enforce synchronization of SYNCI instructions. In the typical use cases, a single GINVI is used by itself to invalidate caches and would be followed by a SYNC (stype=0x14). In the case of GINVT, multiple GINVT could be used to invalidate multiple TLB mappings, and the SYNC (stype=0x14) would be used to guaranteed completion of any number of GINVTs preceding it.

Terms:

Synchronizable: A load or store instruction is synchronizable if the load or store occurs to a physical location in shared memory using a virtual address with a memory access type of either uncached or cached coherent .

Shared memory: Memory that can be accessed by more than one processor or by a coherent I/O system module.

Performed load: A load instruction is performed when the value returned by the load has been determined. The result of a load on processor A has been determined with respect to processor or coherent I/O module B when a subsequent store to the location by B cannot affect the value returned by the load. The store by B must use the same memory access type as the load.

Performed store: A store instruction is performed when the store is observable. A store on processor A is observable with respect to processor or coherent I/O module B when a subsequent load of the location by B returns the value written by the store. The load by B must use the same memory access type as the store.

Globally performed load: A load instruction is globally performed when it is performed with respect to all processors and coherent I/O modules capable of storing to the location.

Globally performed store: A store instruction is globally performed when it is globally observable. It is globally observable when it is observable by all processors and I/O modules capable of loading from the location.

Global ordering point: A point in the coherence domain where when a memory instruction reaches, it can be guaranteed to retain its order relative to any memory instruction that reaches the point after it.

Coherent I/O module: A coherent I/O module is an Input/Output system component that performs coherent Direct Memory Access (DMA). It reads and writes memory independently as though it were a processor doing loads and stores to locations with a memory access type of cached coherent.

Programming Notes:

A processor executing load and store instructions observes the order in which loads and stores using the same memory access type occur in the instruction stream; this is known as program order.

A *parallel program* has multiple instruction streams that can execute simultaneously on different processors. In multiprocessor (MP) systems, the order in which the effects of loads and stores are observed by other processors - the global order of the loads and store - determines the actions necessary to reliably share data in parallel programs.

When all processors observe the effects of loads and stores in program order, the system is strongly ordered. On such systems, parallel programs can reliably share data without explicitly using a SYNC.

Executing SYNC on such a system is not necessary, will not cause an error, but may reduce overall performance.

If a multiprocessor system is not strongly ordered, the effects of load and store instructions executed by one processor may be observed out of program order by other processors. On such systems, parallel programs must use SYNC to reliably share data at critical points in the program. SYNC separates the loads and stores executed on the processor into two groups, and the effect of all loads and stores in one group is seen by all processors before the effect of any load or store in the subsequent group. In effect, SYNC causes the system to be strongly ordered for the executing processor at the instant that the SYNC is executed.

The hardware ordering support provided in a MIPS-based multiprocessor system is implementation dependent. A parallel program that does not use SYNC generally does not operate on a system that is not strongly ordered. However, a program that does use SYNC works on both types of systems. (System-specific documentation describes the actions needed to reliably share data in parallel programs for that system.)

The behavior of a load or store using one memory access type is UNPREDICTABLE if a load or store was previously made to the same physical location using a different memory access type. The presence of a SYNC between the references does not alter this behavior.

SYNC affects the order in which the effects of load and store instructions appear to all processors; it does not generally affect the physical memory-system ordering or synchronization issues that arise in system programming. The effect of SYNC on implementation-specific aspects of the cached memory system, such as writeback buffers, is not defined.

The code fragments below show how SYNC can be used to coordinate the use of shared data between separate writer and reader instruction streams in a multiprocessor environment. The FLAG location is used by the instruction streams to determine whether the shared data item DATA is valid. The SYNC executed by processor A forces the store of DATA to be performed globally before the store to FLAG is performed. The SYNC executed by processor B ensures that DATA is not read until after the FLAG value indicates that the shared data is valid.

```
# Processor A (writer)
# Conditions at entry:
# The value 0 has been stored in FLAG and that value is observable by B
SW     R1, DATA       # change shared DATA value
LI     R2, 1
SYNC                  # Perform DATA store before performing FLAG store
SW     R2, FLAG       # say that the shared DATA value is valid

    # Processor B (reader)
    LI     R2, 1
 1: LW     R1, FLAG   # Get FLAG
    BNEC   R2, R1, 1B # if it says that DATA is not valid, poll again
    NOP
    SYNC              # FLAG value checked before doing DATA read
    LW     R1, DATA   # Read (valid) shared DATA value
    SYNC
```

Exceptions: None.

SYNCI/SYNCIE

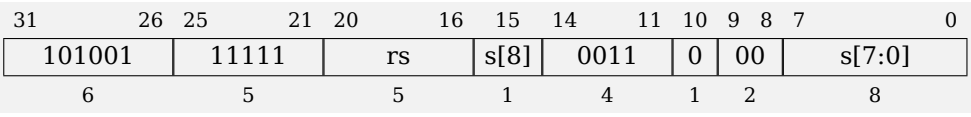
Assembly:

```
SYNCI  offset(rs)
SYNCIE offset(rs)
```

Purpose: *SYNChronize Instruction cache/SYNChronize Instruction cache using EVA addressing*. Synchronize the caches to make instructions writes at address `$rs + offset` (register plus immediate) effective. For SYNCIE, translate the virtual address as though the core is in user mode, although it is actually in kernel mode.

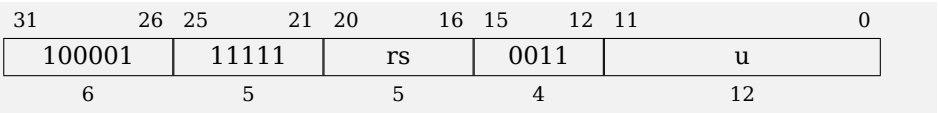
Availability: nanoMIPS, availability varies by format.

Format: SYNCI[S9]



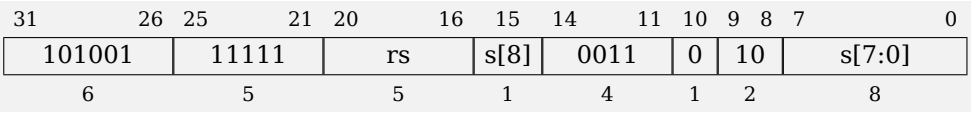

```
offset = sign_extend(s, from_nbits=9)
is_eva = False
```

Format: SYNCI[U12]




```
offset = u
is_eva = False
```

Format: SYNCIE, present when Config5.EVA=1, requires CP0 privilege.




```
offset = sign_extend(s, from_nbits=9)
is_eva = True
```

Operation:

```
if is_eva and not C0.Config5.EVA:
    raise exception('RI')

if is_eva and not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

va = effective_address(GPR[rs], offset, 'Load', eva=is_eva)
pa, cca = va2pa(va, 'Cacheop', eva=is_eva)

# Make data writes at address=PA visible to the instruction stream (for all
# coherent cores in the system)...

# The precise details of the operation are implementation dependent, and will
# depend on the cache hierarchy and coherency behavior of the system. The
# following code shows a sample implementation for a system where the memory
# hierarchy is unified beyond the L1 instruction and data caches.

# Find index where address is present in D cache, if any.
dcache_hit_index = cache_lookup_index('D', va, pa)

if dcache_hit_index:
    way_index, set_index = dcache_hit_index
    dcache_line = get_cache_line('D', way_index, set_index)
    if dcache_line.valid and dcache_line.dirty:
        dcache_line.write_back()
        # Implementation may or may not invalidate line too, see below.

for core in get_all_cores_in_system():
    # Find index where address is present in this core's I cache, if any.
    icache_hit_index = cache_lookup_index('I', va, pa, core)

    if icache_hit_index:
        way_index, set_index = icache_hit_index
        icache_line = get_cache_line('I', way_index, set_index, core)
        if not icache_line.locked:
            icache_line.valid = 0
```

SYNCI is a user privilege instruction for synchronizing the caches to make instruction writes to address `$rs + offset` effective. SYNCI must be followed by a SYNC instruction and an instruction hazard barrier to guarantee that subsequent instruction fetches see the updated instructions. One SYNCI instruction is required for every cache line that was written. The size of the cache line can be determined by the RDHWR instruction.

SYNCI can cause TLB Refill and TLB invalid exceptions (with cause code TLBL). It does not cause TLBRI exceptions. A Cache Error or Bus Error exception may occur as a result of a writeback triggered by the instruction.

An Address Error Exception (with cause code equal ADEL) may occur if a SYNCI targets an address which is not accessible from the current operating mode. It is implementation dependent whether such

an exception does occur, but the instruction should not affect cache lines which are not accessible from the current operating mode.

It is implementation dependent whether a data watch exception is triggered by a SYNCI instruction whose address matches the Watch register address match conditions. The preferred implementation is not to match on the SYNCI instruction.

The operation of the processor is UNPREDICTABLE if the effective address of the SYNCI targets any instruction cache line that contains instructions to be executed between the SYNCI and the subsequent JALRC.HB, JRC.HB, or ERET instruction required to clear the instruction hazard.

The SYNCI instruction has no effect on cache lines that were previously locked with the CACHE instruction. If correct software operation depends on the state of a locked line, the CACHE instruction must be used to synchronize the caches.

In multi-processor systems, a SYNCI to an address with a coherent CCA must guarantee synchronization of all coherent instruction caches in the system. (Prior to Release 6 of the MIPS™ Architecture, this behavior was recommended but not required).

The manner in which SYNCI is implemented will depend on the cache hierarchy of the processor. Typically, all caches out to the point at which both instruction and data references become unified are processed. If no caches exist or if instruction cache coherency is already guaranteed, the instruction must be implemented as a NOP.

In a typical implementation in which only the L1 instruction and data caches are affected, this instruction would perform a Hit Invalidate operation on the instruction cache and a Hit Writeback or Hit Writeback Invalidate on the data cache. The decision to invalidate the data cache line is implementation dependent, but should be made under the assumption that the data will not be written again soon. If a Hit Writeback Invalidate (as opposed to a Hit Writeback) would cause the line to be selected for replacement, the invalidate option might be selected.

The following example shows a routine which could be called after the new instruction stream is written to make those changes effective.

```
/*
 * This routine makes changes to the instruction stream effective to the
 * hardware. It should be called after the instruction stream is written.
 * On return, the new instructions are effective.
 *
 * Inputs:
 *    a0 = Start address of new instruction stream
 *    a1 = Size in bytes of new instruction stream
 */
       beqc    a1, zero, 20f     /* If size==0, branch around. */
       addu    a1, a0, a1        /* Calculate end address + 1. */
       rdhwr   v0, HW_SYNCI_Step /* Get step size for SYNCI. */
       beqc    v0, zero, 20f     /* Nothing to do if no caches. */
10:    synci   0(a0)             /* Sync all caches around address. */
       addu    a0, a0, v0        /* Add step size. */
       sltu    v1, a0, a1        /* Not past the end address? */
       bnec    v1, zero, 10b     /* Branch if more to do. */
```

```
        sync                    /* Clear memory hazards. */
20:     jrc.hb  ra              /* Return, clearing instruction hazards. */
```

Exceptions:

Address Error. Bus Error. Cache Error. Coprocessor Unusable for SYNCIE. Reserved Instruction for SYNCIE if EVA not implemented. TLB Invalid. TLB Refill.

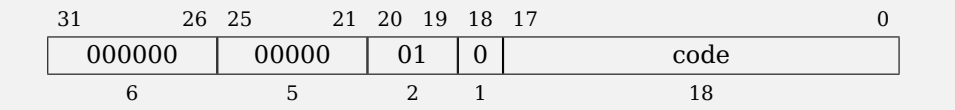
SYSCALL

Assembly: `SYSCALL code`

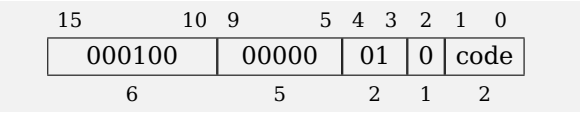
Purpose: *System Call*. Cause a System Call exception.

Availability: nanoMIPS

Format: SYSCALL[32]



Format: SYSCALL[16]



Operation:

```
raise exception('SYSCALL')
```

Exceptions: System Call.

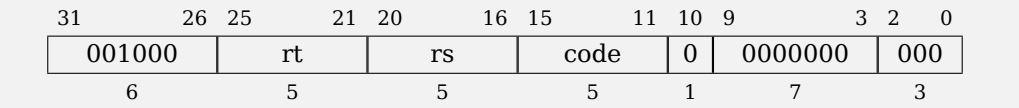
TEQ

Assembly: `TEQ rs, rt, code`

Purpose: *Trap if Equal*. Cause a Trap exception if registers `$rs` and `$rt` are equal.

Availability: nanoMIPS, not available in NMS

Format:



Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

if GPR[rs] == GPR[rt]:
    raise exception('TRAP')
```

Exceptions: Trap.

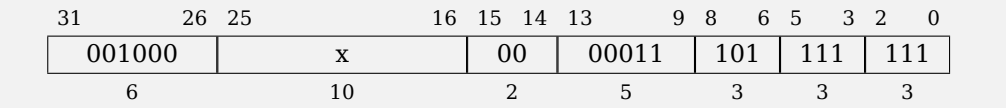
TLBINV

Assembly: `TLBINV`

Purpose: *TLB Invalidate*. Invalidate a set of TLB entries based on ASID match.

Availability: nanoMIPS. Required on TLB cores, unless `Config5.IE<2`. Requires CP0 privilege.

Format:



Operation:

```
if C0.Config4.IE < 2:
    raise exception('RI')

if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

tlbinv()
```

Exceptions:

Coprocessor Unusable. Reserved Instruction if TLB invalidate not implemented.

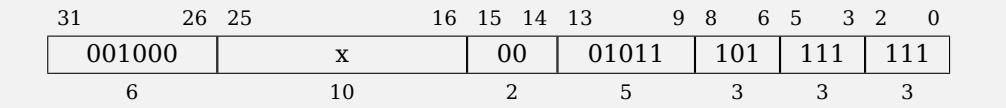
TLBINVF

Assembly: `TLBINVF`

Purpose: *TLB Invalidate Flush*. Invalidate a set of TLB entries, ignoring ASID match.

Availability: nanoMIPS. Required on TLB cores, unless `Config5.IE<2`. Requires CP0 privilege.

Format:



Operation:

```
if C0.Config4.IE < 2:
    raise exception('RI')

if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

tlbinv(flush=True)
```

Exceptions:

Coprocessor Unusable. Reserved Instruction if TLB invalidate not implemented.

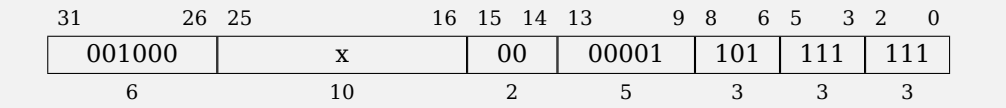
TLBP

Assembly: `TLBP`

Purpose: *TLB Probe*. Probe the TLB for an entry matching `C0.EntryHi`. If found, write the index of the matching entry to `C0.Index`, otherwise set `C0.Index.P` to 1.

Availability: nanoMIPS. Required on TLB cores. Requires CP0 privilege.

Format:



Operation:

```
if not got_tlb():
    raise exception('RI')

if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

tlbp()
```

Exceptions:

Coprocessor Unusable. Reserved Instruction if TLB not implemented.

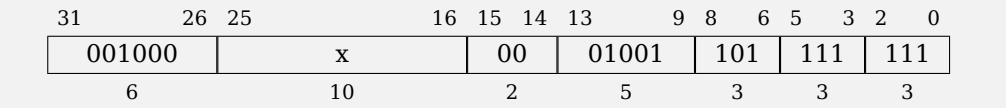
TLBR

Assembly: `TLBR`

Purpose: *TLB Read*. Read the TLB entry indexed by `C0.Index` into the TLB CP0 registers `EntryHi`, `EntryLo0`, `EntryLo1`, `PageMask`.

Availability: nanoMIPS. Required on TLB cores. Requires CP0 privilege.

Format:



Operation:

```
if not got_tlb():
    raise exception('RI')

if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

tlbr()
```

Exceptions:

Coprocessor Unusable. Reserved Instruction if TLB not implemented.

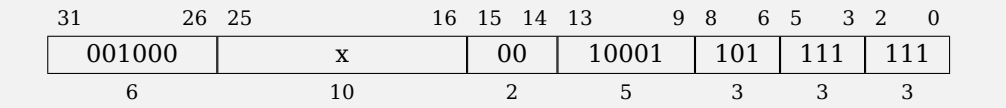
TLBWI

Assembly: `TLBWI`

Purpose: *TLB Write Indexed*. Write the TLB entry indexed by `C0.Index` using the values in the TLB CP0 registers `EntryHi`, `EntryLo0`, `EntryLo1`, `PageMask`.

Availability: nanoMIPS. Required on TLB cores. Requires CP0 privilege.

Format:



Operation:

```
if not got_tlb():
    raise exception('RI')

if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

tlbwi(C0.Index.Index)
```

Exceptions:

Coprocessor Unusable. Reserved Instruction if TLB not implemented.

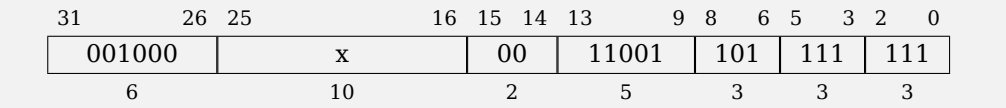
TLBWR

Assembly: `TLBWR`

Purpose: *TLB Write Random*. Write a randomly chosen TLB entry using the values in the TLB CP0 registers `EntryHi`, `EntryLo0`, `EntryLo1`, `PageMask`.

Availability: nanoMIPS. Required on TLB cores. Requires CP0 privilege.

Format:



Operation:

```
if not got_tlb():
    raise exception('RI')

if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

tlbwr()
```

Exceptions:

Coprocessor Unusable. Reserved Instruction if TLB not implemented.

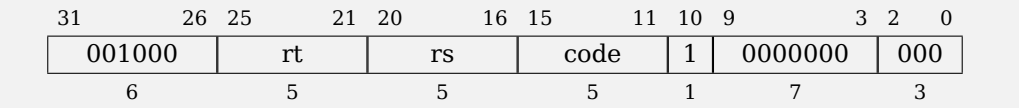
TNE

Assembly: `TNE rs, rt, code`

Purpose: *Trap if Not Equal*. Cause a Trap exception if registers `$rs` and `$rt` are not equal.

Availability: nanoMIPS, not available in NMS

Format:



Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

if GPR[rs] != GPR[rt]:
    raise exception('TRAP')
```

Exceptions: Trap.

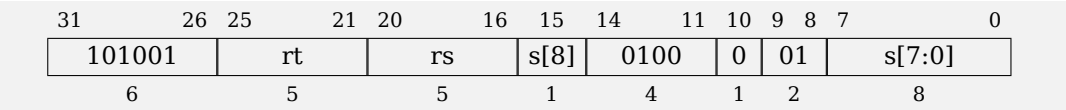
UALH

Assembly: `UALH rt, offset(rs)`

Purpose: *Unaligned Load Half*. Load signed halfword to register `$rt` from memory address `$rs + offset` (register plus immediate), guaranteeing that the operation completes even if the address is not halfword aligned.

Availability: nanoMIPS, not available in NMS

Format:



Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

offset = sign_extend(s, from_nbits=9)

va = effective_address(GPR[rs], offset, 'Load')

data = read_memory_at_va(va, nbytes=2, unaligned_support='always')
GPR[rt] = sign_extend(data, from_nbits=16)
```

UALH will not cause an Address Error exception for unaligned addresses.

An unaligned load/store instruction may be implemented using more than one memory transaction. It is possible for a subset of these memory transactions to have completed and then for a TLB exception to occur on a remaining transaction. It is also possible that memory could be modified by another thread or device in between the completion of the memory transactions. This behavior is equivalent to what might occur if the unaligned load/store was carried out in software using a series of separate aligned instructions, for instance using LWL/LWR on a pre-R6 MIPS™ core. Software should take equivalent steps to accommodate this lack of guaranteed atomicity as it would for the multiple instruction case.

Exceptions:

Bus Error. Reserved Instruction on NMS cores. TLB Invalid. TLB Refill. TLB Read Inhibit. Watch.

UALW (Assembly alias)

Assembly: `UALW rt, offset(rs)`

Purpose: *Unaligned Load Word*. Load word to register `$rt` from memory address `$rs` + `offset` (register plus immediate), guaranteeing that the operation completes even if the address is not word aligned.

Availability: Assembly alias, not available in NMS

Expansion:

```
UALWM rt, offset(rs), 1
```

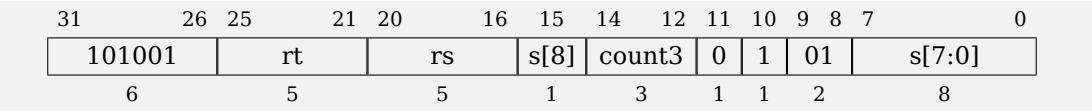
UALWM

Assembly: `UALWM rt, offset(rs), count`

Purpose: *Unaligned Load Word Multiple*. Load `count` words of data to registers `$rt`, `$(rt+1)`, ..., `$(rt+count-1)` from consecutive memory address starting at `$rs + offset` (register plus immediate). Guarantee that the operation completes even if the address is not word aligned.

Availability: nanoMIPS, not available in NMS

Format:




```
offset = sign_extend(s, from_nbits=9)
count = 8 if count3 == 0 else count3
```

Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

i = 0
while i != count:
    this_rt = ( rt + i      if rt + i < 32 else
                rt + i - 16                    )

    this_offset = offset + (i<<2)
    va = effective_address(GPR[rs], this_offset, 'Load')

    data = read_memory_at_va(va, nbytes=4, unaligned_support='always')
    GPR[this_rt] = sign_extend(data, from_nbits=32)

    if this_rt == rs and i != count - 1:
        raise UNPREDICTABLE()

    i += 1
```

UALWM loads `count` words to sequentially numbered registers from sequential memory addresses which are potentially unaligned. After loading $31, the sequence of registers continues from $16. See LWM for example encodings of the register list.

UALWM will not cause an Address Error exception for unaligned addresses.

The result is unpredictable if an UALWM instruction updates the base register prior to the final load.

If a TLB exception or interrupt occurs during the execution of this instruction, a subset of the required register updates may have occurred.

An unaligned load/store instruction may be implemented using more than one memory transaction. It is possible for a subset of these memory transactions to have completed and then for a TLB exception to occur on a remaining transaction. It is also possible that memory could be modified by another thread or device in between the completion of the memory transactions. This behavior is equivalent to what might occur if the unaligned load/store was carried out in software using a series of separate aligned instructions, for instance using LWL/LWR on a pre-R6 MIPS™ core. Software should take equivalent steps to accommodate this lack of guaranteed atomicity as it would for the multiple instruction case.

UALWM must be implemented in such a way as to make the instruction restartable, but the implementation does not need to be fully atomic. For instance, it is allowable for a UALWM instruction to be aborted by an exception after a subset of the register updates have occurred. To ensure restartability, any write to GPR `$rs` (which may be used as the final output register) must be completed atomically, that is, the instruction must graduate if and only if that write occurs.

Exceptions:

Bus Error. Reserved Instruction on NMS cores. TLB Invalid. TLB Read Inhibit. TLB Refill. Watch.

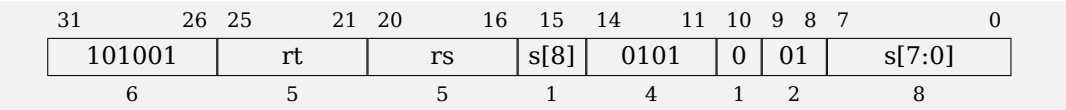
UASH

Assembly: `UASH rt, offset(rs)`

Purpose: *Unaligned Store Half*. Store halfword from register `$rt` to memory address `$rs + offset` (register plus immediate), guaranteeing that the operation completes even if the address is not halfword aligned.

Availability: nanoMIPS, not available in NMS

Format:



Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

offset = sign_extend(s, from_nbits=9)

va = effective_address(GPR[rs], offset, 'Store')

data = zero_extend(GPR[rt], from_nbits=16)
write_memory_at_va(data, va, nbytes=2, unaligned_support='always')
```

UASH will not cause an Address Error exception for unaligned addresses.

An unaligned load/store instruction may be implemented using more than one memory transaction. It is possible for a subset of these memory transactions to have completed and then for a TLB exception to occur on a remaining transaction. It is also possible that memory could be modified by another thread or device in between the completion of the memory transactions. This behavior is equivalent to what might occur if the unaligned load/store was carried out in software using a series of separate aligned instructions, for instance using LWL/LWR on a pre-R6 MIPS™ core. Software should take equivalent steps to accommodate this lack of guaranteed atomicity as it would for the multiple instruction case.

Exceptions:

Bus Error. Reserved Instruction on NMS cores. TLB Invalid. TLB Modified. TLB Refill. Watch.

UASW (Assembly alias)

Assembly: `UASW rt, offset(rs)`

Purpose: *Unaligned Store Word*. Store word from register `$rt` to memory address `$rs + offset` (register plus immediate), guaranteeing that the operation completes even if the address is not word aligned.

Availability: Assembly alias, not available in NMS

Expansion:

```
UASWM rt, offset(rs), 1
```

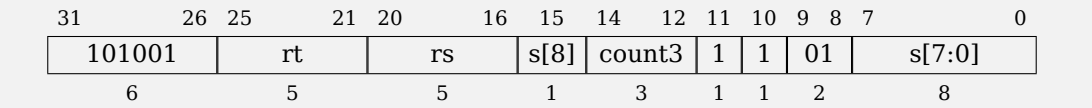
UASWM

Assembly: `UASWM rt, offset(rs), count`

Purpose: *Unaligned Store Word Multiple*. Store `count` words of data from registers `$rt`, `$(rt+1)`, ..., `$(rt+count-1)` to consecutive memory addresses starting at `$rs + offset` (register plus immediate). Guarantee that the operation completes even if the address is not word aligned.

Availability: nanoMIPS, not available in NMS

Format:




```
offset = sign_extend(s, from_nbits=9)
count = 8 if count3 == 0 else count3
```

Operation:

```
if C0.Config5.NMS == 1:
    raise exception('RI')

i = 0
while i != count:
    this_rt = ( 0           if rt == 0     else
                rt + i      if rt + i < 32 else
                rt + i - 16                   )

    this_offset = offset + (i<<2)
    va = effective_address(GPR[rs], this_offset, 'Store')

    data = zero_extend(GPR[this_rt], from_nbits=32)
    write_memory_at_va(data, va, nbytes=4, unaligned_support='always')

    i += 1
```

UASWM stores `count` words from sequentially numbered registers to sequential memory addresses which are potentially unaligned. After storing $31, the sequence of registers continues from $16. If rt=0, then $0 is stored for all `count` steps of the instruction. See SWM for example encodings of the register list.

UASWM will not cause an Address Error exception for unaligned addresses.

If a TLB exception or interrupt occurs during the execution of this instruction, a subset of the required memory updates may have occurred. A full restart of the instruction will be performed on return from the exception.

An unaligned load/store instruction may be implemented using more than one memory transaction. It is possible for a subset of these memory transactions to have completed and then for a TLB exception to occur on a remaining transaction. It is also possible that memory could be modified by another thread or device in between the completion of the memory transactions. This behavior is equivalent to what might occur if the unaligned load/store was carried out in software using a series of separate aligned instructions, for instance using LWL/LWR on a pre-R6 MIPS™ core. Software should take equivalent steps to accommodate this lack of guaranteed atomicity as it would for the multiple instruction case.

Exceptions:

Bus Error. Reserved Instruction on NMS cores. TLB Invalid. TLB Modified. TLB Refill. Watch.

WAIT

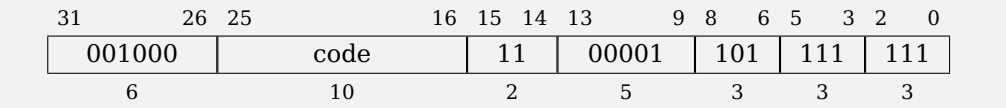
Assembly:

```
WAIT code
WAIT       # code=0 implied
```

Purpose: *Wait*. Enter wait state.

Availability: nanoMIPS

Format:



Operation:

```
if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

CPU.in_wait_state = True
```

Exceptions: Coprocessor Unusable.

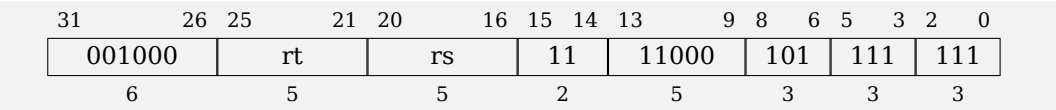
WRPGPR

Assembly: `WRPGPR rt, rs`

Purpose: *Write Previous GPR*. Write the value of register `$rs` from the current shadow register set (SRSCtl.CSS) to register `$rt` in the previous shadow register set (SRSCtl.PSS). If shadow register sets are not implemented, just copy the value from register `$rs` to register `$rt`.

Availability: nanoMIPS. Requires CP0 privilege.

Format:



Operation:

```
if not IsCoprocessor0Enabled():
    raise coprocessor_exception(0)

if C0.SRSCtl.HSS > 0:
    SRS[C0.SRSCtl.PSS][rt] = GPR[rs]
else:
    GPR[rt] = GPR[rs]
```

Exceptions: Coprocessor Unusable.

WSBH (Assembly alias)

Assembly: `WSBH rt, rs`

Purpose: *Word Swap Byte Half*. Swap the bytes within both halfs of the word value in register `$rs`, and write the result to register `$rt`.

Availability: Assembly alias, not available in NMS

Expansion:

```
ROTX rt, rs, 8, 24
```

The assembly alias WSHB is provided for compatibility with MIPS32™. Its behavior is equivalent to the new assembly alias BYTEREVH, whose name is chosen to fit consistently with the naming of other reversing instructions in nanoMIPS™.

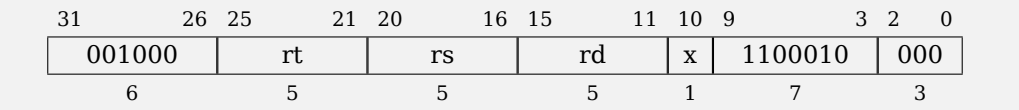
XOR

Assembly: `XOR rd, rs, rt`

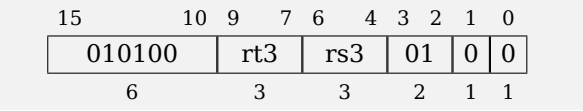
Purpose: *XOR*. Compute logical XOR of registers `$rs` and `$rt`, placing the result in register `$rt`.

Availability: nanoMIPS

Format: XOR[32]



Format: XOR[16]




```
rt = decode_gpr(rt3, 'gpr3')
rs = decode_gpr(rs3, 'gpr3')
rd = rt
```

Operation:

```
GPR[rd] = GPR[rs] ^ GPR[rt]
```

Exceptions: None.

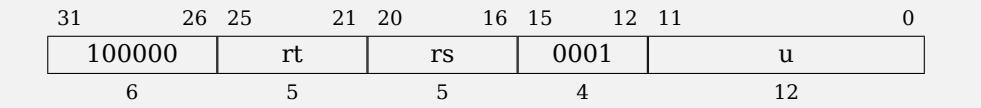
XORI

Assembly: `XORI rt, rs, u`

Purpose: *XOR Immediate*. Compute logical XOR of register `$rs` with immediate `u`, placing the result in register `$rt`.

Availability: nanoMIPS

Format:



Operation:

```
GPR[rt] = GPR[rs] ^ u
```

Exceptions: None.

Chapter 4

Shared Pseudocode Functions

4.1 Are64BitOperationsEnabled()

Are 64 bit operations enabled?

```
def Are64BitOperationsEnabled():
    return EffectiveKSU() != 2 or C0.Status.PX or C0.Status.UX
```

4.2 cacheop()

The `cacheop(op, va, pa)` function performs the cache operation specified by operation 'op' on the address specified by arguments 'va' and 'pa'.

```
def cacheop(va, pa, op):

    # Figure out the target cache for this cache operation.
    target_cache = ('I' if op[1:0] == 0 else
                    'D' if op[1:0] == 1 else
                    'T' if op[1:0] == 2 else
                    'S' if op[1:0] == 3 else FatalError("unexpected cacheop"))

    # Find encoded line size, sets, and associativity for the target cache.
    (L, S, A) = get_cache_parameters(target_cache)

    # Decode the encoded line size, sets and associativity.
    line_size = 2 ** (L + 1) if L else 0
    num_sets = 2 ** (S + 6)
    num_ways = A + 1

    # Deal with case where target cache is not present on this core.
    if line_size == 0:
       if C0.Config.AR >= 2:
```

```
            return  # For R6 cores, cacheop to non-existent cache is a NOP
        else:
            raise UNPREDICTABLE('cacheop to non-existent cache is unpredictable')

    # Find the set index and way index which this cacheop should target.
    if op <= 12 or (op <= 16 and CPU.imp_cacheop_is_indexed):  # Index cacheop
        # Figure out the bit positions for lines, sets and ways in the address.
        line_bits = (line_size - 1).bit_length()
        set_bits = (line_size * num_sets - 1).bit_length()
        way_bits = (line_size * num_sets * num_ways - 1).bit_length()

        set_index = va[set_bits - 1: line_bits]
        if num_ways > 1:
            way_index = va[way_bits - 1: set_bits]
        else:
            way_index = 0

    else:  # Hit cacheop
        # Find index which stores this address in the cache (if any).
        hit_index = cache_lookup_index(target_cache, va, pa)

        if hit_index is None:
            if (   op == 20  # ICache Fill. (Recommended).
                or op == 28  # ICache Fetch and Lock. (Recommended).
                or op == 29  # DCache Fetch and Lock. (Recommended).
                ):

                # For operations which can fill the cache from memory, fill it
                # here (having written the previous contents of the chosen
                # line to memory if it was dirty).
                hit_index = fill_cache_from_memory(target_cache, va, pa)
            else:
                return  # NOP if this address is not found in the cache

        (set_index, way_index) = hit_index

    # Get cache_line object representing the cache line at the required index.
    cache_line = get_cache_line(target_cache, way_index, set_index)

    if op[4:2] == 0:
        if target_cache == 'I':  # ICache Index Invalidate.
            cache_line.valid = False  # Invalidate the cache line.
            cache_line.locked = False  # Unlock cache line (when implemented).

        else:  # Index Writeback Invalidate
            if cache_line.valid and cache_line.dirty:
                cache_line.write_back()  # Write back if dirty.
            cache_line.valid = False  # Invalidate the cache line.
```

```
77              cache_line.locked = False  # Unlock cache line (when implemented).
78
79      elif op[4:2] == 1: # Index Load Tag.
80          # Implementation dependent read from cache_line tag/data to C0.TagLo,
81          # C0.TagHi, etc. registers. (Recommended).
82          index_load_tag(cache_line)
83
84      elif op[4:2] == 2:  # Index Store Tag.
85          # If the C0.TagLo/C0.TagHi registers are initialized to 0, the
86          # Index Store Tag cacheop must initialize the cache line by setting it
87          # to the 'invalid' state. (Required).
88          if (    C0.TagLo[0] == 0 and C0.TagLo[2] == 0
89              and C0.TagHi[0] == 0 and C0.TagHi[2] == 0):
90              cache_line.valid = False  # Initialize (invalidate) the cache line.
91              cache_line.locked = False  # Unlock cache line (when implemented).
92          else:
93              # Implementation dependent operation to write cache tag/data from
94              # C0.TagLo, C0.TagHi, etc. registers. (Recommended).
95              index_store_tag(cache_line)
96
97      elif op[4:2] == 3:  # Implementation dependent cacheop.
98          pass # Placeholder for implementation dependent cacheop.
99
100     elif op[4:2] == 4:  # Hit Invalidate.
101         # Required for I cache, optional otherwise.
102         cache_line.valid = False  # Invalidate the cache line.
103         cache_line.locked = False  # Unlock the cache line (when implemented).
104
105     elif op[4:2] == 5:
106         if target_cache == 'I':  # Fill
107             pass  # Cache line was filled for ICache Fill earlier in pseudocode.
108
109         else:  # Hit Writeback Invalidate
110            if cache_line.valid and cache_line.dirty:
111                cache_line.write_back()  # Write back if dirty.
112            cache_line.valid = False  # Invalidate the cache line.
113            cache_line.locked = False  # Unlock cache line (when implemented).
114
115     elif op[4:2] == 6:  # Hit Writeback. (Recommended).
116         if target_cache != 'I' and cache_line.valid and cache_line.dirty:
117             cache_line.write_back()  # Write back if dirty.
118
119     elif op[4:2] == 7:
120         if target_cache == 'I' or target_cache == 'D': # Fetch and Lock. (Recommended).
121             # Cache line already fetched earlier in pseudocode.
122             cache_line.locked = True
123
124     else:
```

```
125         raise ValueError('Unexpected cacheop operation')
```

4.3 coprocessor_exception()

Perform state updates associated with a coprocessor unusable exception.

```
10  def coprocessor_exception(coprocessor_number):
11      exception('CPU')
12      C0.Cause.CE = coprocessor_number
13      raise EXCEPTION
```

4.4 count_leading_zeros()

Return the number of leading zeros in the `nbits` least significant bits of value `value`.

```
10  def count_leading_zeros(value, nbits):
11      count = 0;
12      bit = nbits - 1
13      while bit >= 0:
14          if value & (1 << bit):
15              return count
16          count += 1
17          bit -= 1
```

4.5 crc32()

Compute the 32-bit CRC value from the following inputs:

- `value`: right-justified current 32-bit CRC value,
- `message`: right-justified message,
- `nbits`: size of `message` in bits,
- `poly`: 32-bit reversed polynomial.

```
10  def crc32(value, message, nbits, poly):
11      value = value[31:0] ^ message[nbits-1:0]
12
13      for i in range(nbits):
14          if value & 1:
15              value = (value >> 1) ^ poly
16          else:
17              value = (value >> 1)
18
19      return value
```

4.6 debug_exception()

Perform the state updates associated with a debug exception. Arguments are:

- `cause`: The Debug.DExcCode value (if any) for this exception.
- `comment`: A comment about the reason for the exception, for debug and explanation purposes only.

```
def debug_exception(cause=None, comment=None):

    # Trace a comment for this exception for explanation purposes only.
    comment_text = "Debug exception"
    if cause:
        comment_text += f", {cause}"
    if comment:
        comment_text += f", {comment}"
    trace(comment_text)

    if cause is not None:
        Root.C0.Debug.DExcCode = cause

    epc = CPU.prior_pc if CPU.in_bds else CPU.pc
    Root.C0.DEPC = zero_extend(epc, from_nbits=64 if C0.Config.AT==2 else 32)
    Root.C0.Debug.DBD = CPU.in_bds

    Root.C0.Debug.DM = 1

    # Clear various flag bits. If any are to be set to 1, let the caller do it
    # after calling this function.
    Root.C0.Debug.DBp = 0
    Root.C0.Debug.DSS = 0

    exception_va = sign_extend(0xbfc00480, from_nbits=32)

    if C0.Config3.ISA & 1 and not is_nanomips():
        exception_va |= 1

    CPU.next_pc = exception_va
    CPU.got_next_next_pc = False  # For branch delay slot handling in simulator.

    return EXCEPTION()  # Makes the "raise exception(...)" syntax work.
```

4.7 decode_gpr()

Return the GPR index for different types of nanoMIPS™ encoded GPR value.

```
def decode_gpr(encoded_gpr, encoding_type):
    if encoding_type == 'gpr3':
        # Standard list of 8 GPRs used by most 16 bit instructions.
        gpr_list = [16, 17, 18, 19, 4, 5, 6, 7]
    elif encoding_type == 'gpr3.src.store':
        # Source GPR used by 16 bit store instructions.
        # ($0 used instead of S0).
        gpr_list = [0, 17, 18, 19, 4, 5, 6, 7]
    elif encoding_type == 'gpr1':
        # Destination GPR used by MOVE.BALC.
        gpr_list = [4, 5]
    elif encoding_type == 'gpr2.reg1':
        # First GPR used by MOVEP and MOVEP[REV].
        gpr_list = [4, 5, 6, 7]
    elif encoding_type == 'gpr2.reg2':
        # Second GPR used by MOVEP and MOVEP[REV].
        gpr_list = [5, 6, 7, 8]
    elif encoding_type == 'gpr4':
        # Standard list of 16 GPRs used by 16 bit instructions not in MMS.
        gpr_list = [8, 9, 10, 11, 4, 5, 6, 7, 16, 17, 18, 19, 20, 21, 22, 23]
    elif encoding_type == 'gpr4.zero':
        # Source GPR used by store and movep instructions.
        # Use $0 instead of a7
        gpr_list = [8, 9, 10, 0, 4, 5, 6, 7, 16, 17, 18, 19, 20, 21, 22, 23]
    else:
        FatalError('unknown encoding type')

    return gpr_list[encoded_gpr]
```

4.8 decode_va()

Find the address translation type associated with the specified virtual address. Arguments are:

- `va`: Virtual address to be translated.
- `eva`: Are we translating for an EVA instruction, i.e. a kernel privilege instruction which translates addresses as though in user mode?

The return values are `(translation_type, comment, args)`. `comment` is a string describing the current translation rules in use (for explanation and debug purposes only) and `translation_type` is one of:

- 'ade': Address privilege error. `args` is null in this case.
- 'mapped': Means that the address needs to be mapped through the TLB. In this case, `args` is a single element list `(use_xtlb_vector)` which specifies whether an XTLB miss (as opposed to a TLB miss) should be taken if the address misses in the TLB.

- 'unmapped': Means that the address is not mapped through the TLB. In this case `args` contains `(pa, cca)` for the current translation.

```
def decode_va(va, eva=False):

    if overflows(va, 32):
        return decode_va_mips64(va, eva)

    if C0.Config3.SC:
        return decode_va_segctl(va, eva)

    ksu = EffectiveKSU(eva)
    is_fm = C0.Config.MT == 'FMMMU'

    comment = ''

    va_u = zero_extend(va, from_nbits=32)

    if va_u <= 0x7fffffff:
        comment += 'useg'
        priv = 2

        if C0.Status.ERL:
            mapped = False
            pa = va
            cca = 2
        elif is_fm:
            mapped = False
            pa = va + 0x40000000
            cca = C0.Config.KU
        else:
            mapped = True
            use_xtlb_vector = C0.Status.UX

    elif va_u <= 0x9fffffff:
        comment += 'kseg0'
        priv = 0
        mapped = False
        pa = va & 0x1fffffff
        cca = C0.Config.K0

    elif va_u <= 0xbfffffff:
        comment += 'kseg1'
        priv = 0
        mapped = False
        pa = va & 0x1fffffff
        cca = 2

```

```
    elif va_u <= 0xdfffffff:
        comment += 'sseg'
        priv = 1

        if is_fm:
            mapped = False
            pa = va
            cca = C0.Config.K23
        else:
            mapped = True
            use_xtlb_vector = C0.Status.KX

    elif va_u <= 0xffffffff:
        comment += 'kseg3'
        priv = 0

        if is_fm:
            mapped = False
            pa = va
            cca = C0.Config.K23
        else:
            mapped = True
            use_xtlb_vector = C0.Status.KX

    else:
        FatalError("unexpected va")

    if ksu > priv:
        return ('ade', "privilege error, " + comment, ())

    if mapped:
        return ('map', comment, (use_xtlb_vector))
    else:
        return ('unmapped', comment, (pa, cca))
```

4.9 decode_va_mips64()

decode_va_mips64() is called by decode_va() to process VAs which are outside of the MIPS32 compatibility range. The arguments and return values have the same format as decode_va().

```
def decode_va_mips64(va, eva):

    if C0.Config.AT != 2:
        FatalError(f"VA {va:x} out of 32 bit range but no m64 address support")

    ksu = EffectiveKSU(eva)
```

```
    segbits = CPU.SEGBITS

    va62 = va & (2**62-1)
    xseg = (va >> 62) & 3

    if xseg == 0:  # xuseg
        if C0.Status.UX == 0:
            return ('ade', 'xuseg access with UX=0', ())

        if va62 >> segbits:
            return ('ade', 'xuseg access with va[61..SEGBITS] != 0', ())

        return ('map', 'xuseg', ('x'))

    elif xseg == 1:  # xsseg
        if C0.Status.SX == 0:
            return ('ade', 'xsseg access with SX=0', ())

        if ksu > 1:
            return ('ade', 'privilege error, xsseg', ())

        if va62 >> segbits:
            return ('ade', 'xsseg access with va[61..SEGBITS] != 0', ())

        return ('map', 'xsseg', ('x'))

    elif xseg == 2:  # xkphys
        if C0.Status.KX == 0:
            return ('ade', 'xkphys access with KX=0', ())

        if ksu > 0:
            return ('ade', 'privilege error, xkphys', ())

        pa = zero_extend(va62, from_nbits=59)
        cca = (va62 >> 59) & 3

        if pa >= 2 ** 32 and is_guest_mode() and Root.C0.PageGrain.ELPA == 0:
            return ('ade', 'PA > 32 bits, Root.ELPA=0', ())

        if pa >= 2 ** CPU.PABITS:
            return ('ade', 'PA > PABITS', ())

        return ('unmapped', 'xkphys', (pa, cca))

    elif xseg == 3: # xkseg
        # special considerations for size of xkseg segment
        if va62 >= 2**segbits - 2**31:
            return ('ade', 'xkseg va out of range', ())
```

```
64
65          if C0.Status.KX == 0:
66              return ('ade', 'xkseg access with KX=0', ())
67
68          if ksu > 0:
69              return ('ade', 'privilege error, xkseg', ())
70
71          return ('map', 'xkseg', ('x'))
72
73      else:
74          FatalError("unknown xseg " + xseg)
```

4.10 decode_va_segctl()

decode_va_segctl() is called by decode_va() for addresses in the MIPS32 compatibility range on cores which implement segmentation control. The arguments and return values have the same format as decode_va().

```
10  def decode_va_segctl(va, eva):
11
12      # Look for a possible overlay segment.
13      overlay = decode_overlay(va, eva)
14      if overlay:
15          return overlay
16
17      comment = ''
18
19      ksu = EffectiveKSU(eva)
20
21      # Figure out the segment information.
22      va_u = zero_extend(va, from_nbits=32)
23
24      if va_u <= 0x3fffffff:
25          comment += 'seg5'
26          am = C0.SegCtl2.AM5
27          eu = C0.SegCtl2.EU5
28          cca = C0.SegCtl2.C5
29          pa = (C0.SegCtl2.PA5 >> 1) @ va[29:0]
30
31      elif va_u <= 0x7fffffff:
32          comment += 'seg4'
33          am = C0.SegCtl2.AM4
34          eu = C0.SegCtl2.EU4
35          cca = C0.Config.K0 if C0.Config5.K else C0.SegCtl2.C4
36          pa = (C0.SegCtl2.PA4 >> 1) @ va[29:0]
37
```

```
    elif va_u <= 0x9fffffff:
        comment += 'seg3'
        am = C0.SegCtl1.AM3
        eu = C0.SegCtl1.EU3
        cca = C0.SegCtl1.C3
        pa = C0.SegCtl1.PA3 @ va[28:0]

    elif va_u <= 0xbfffffff:
        comment += 'seg2'
        am = C0.SegCtl1.AM2
        eu = C0.SegCtl1.EU2
        cca = C0.SegCtl1.C2
        pa = C0.SegCtl1.PA2 @ va[28:0]

    elif va_u <= 0xdfffffff:
        comment += 'seg1'
        am = C0.SegCtl0.AM1
        eu = C0.SegCtl0.EU1
        cca = C0.SegCtl0.C1
        pa = C0.SegCtl0.PA1 @ va[28:0]

    elif va_u <= 0xffffffff:
        comment += 'seg0'
        am = C0.SegCtl0.AM0
        eu = C0.SegCtl0.EU0
        cca = C0.SegCtl0.C0
        pa = C0.SegCtl0.PA0 @ va[28:0]

    else:
        FatalError("unexpected va")

    # Interpret the access mode.
    if am == 0:
        comment += ", UK"
        priv = 0
        mapped = False

    elif am == 1:
        comment += ", MK"
        priv = 0
        mapped = True
        use_xtlb_vector = C0.Status.KX

    elif am == 2:
        comment += ", MSK"
        priv = 1
        mapped = True
        use_xtlb_vector = C0.Status.KX
```

```
86
87      elif am == 3:
88          comment += ", MUSK"
89          priv = 2
90          mapped = True
91          use_xtlb_vector = C0.Status.UX
92
93      elif am == 4:
94          priv = 2
95
96          if ksu == 0:
97              comment += ", MUSK(K)"
98              mapped = False
99          else:
100             comment += ", MUSK(not K)"
101             mapped = True
102             use_xtlb_vector = C0.Status.UX
103
104     elif am == 5:
105         priv = 1
106         mapped = False
107         comment += ", USK"
108
109     elif am == 7:
110         priv = 2
111         mapped = False
112         comment += ", UUSK"
113
114     else:
115         FatalError("unknown am " + am)
116
117     # Check for privilege exception.
118     if ksu > priv:
119         return ('ade', "privilege error, " + comment, ())
120
121     # Override when ERL=1 and EU is set
122     if eu and C0.Status.ERL and not eva:
123         comment += ', EU'
124         mapped = False
125         cca = 2
126
127     if mapped:
128         return ('map', comment, (use_xtlb_vector))
129     else:
130         return ('unmapped', comment, (pa, cca))
```

4.11 divide_integers()

Return the quotient and remainder obtained by dividing integer numerator by integer denominator.

```
def divide_integers(numerator, denominator):

    # For clarity, we explicitly show the behavior for each possible
    # combination of the signs of the input values. Division is carried out
    # using the python '//' (floor division) operator, which returns the
    # highest integer which is less than or equal to the exact result.

    if denominator == 0:
        # Division by zero must be filtered out before calling this function.
        FatalError('division by zero')

    if numerator >= 0 and denominator >= 0:
        quotient = numerator // denominator

    elif numerator >= 0 and denominator < 0:
        quotient = -(numerator // abs(denominator))

    elif numerator < 0 and denominator >= 0:
        quotient = -(abs(numerator) // denominator)

    elif numerator < 0 and denominator < 0:
        quotient = abs(numerator) // abs(denominator)

    remainder = numerator - quotient * denominator

    return quotient, remainder
```

4.12 effective_address()

Compute the effective address to be used when a load, store or fetch adds an offset to a base address. Depending on the addressing mode, the address will wrap at 32 or 64 bits.

On a 32 bit core, effective_address() is just a 32 bit addition.

An address exception will be signaled on a load or store executing in 32 bit mode if the base or offset is not within 32 bit range.

The base and offset arguments are used symmetrically, so for instance for an indexed load/store instruction, there is no particular significance to which value is passed as the base, and which is passed as the offset.

```
def effective_address(base, offset, access_type='Fetch', eva=False):

```

```
    if C0.Config.AT != 2:
        return sign_extend(base + offset, from_nbits=32)

    # effective_address will be either the 32 or 64 bit wrapped value. Compute
    # them both here for convenience.
    ea64 = sign_extend(base + offset, from_nbits=64)
    ea32 = sign_extend(base + offset, from_nbits=32)

    addressing_mode = 64 if pointers_are_64_bits(access_type, eva) else 32

    # In R6 or EVA, special behavior occurs if we are in 32 bit addressing
    # mode but the base or offset is not in 32 bit range.
    if is_r6() or C0.Config5.EVA and addressing_mode == 32:
        if overflows(base, 32) or overflows(offset, 32):
            # For loads and stores, take an address exception now.
            if access_type == 'Load':
                raise exception('ADEL', badva=ea64)
            elif access_type == 'Store':
                raise exception('ADES', badva=ea64)

            # For fetches, don't wrap the effective address at 32 bits.
            # This may lead to an address error on the fetch.
            return ea64

    if addressing_mode == 64:
        return ea64
    else:
        return ea32
```

4.13 EffectiveKSU()

Is the processor currently operating in Kernel, Supervisor or User mode? If the eva argument is set, then return the effective operation mode for an EVA instruction (i.e. a kernel privilege instruction which translates addresses as though in user mode).

```
def EffectiveKSU(eva=False):
    if eva:
        return 2  # EVA instructions act as though in user mode
    elif C0.Status.EXL or C0.Status.ERL or C0.Debug.DM:
        return 0
    else:
        return C0.Status.KSU
```

4.14 exception()

Perform the state updates associated with an exception. Arguments are:

- `cause`: The Cause.ExcCode value for this exception.
- `comment`: A comment about the reason for the exception, for debug and explanation purposes only.
- `g`: For VZ cores: If `g` is True, take an exception in guest mode. If `g` is False, take an exception in Root mode. If `g` is None, take an exception in the current operating mode (Root or Guest).
- `offset`: The exception vector offset. By default, use the general exception vector (0x180).
- `ess`: The exception shadow register set to use. By default, C0.SRSCtl.ESS will be used, but the `ess` arg allows a different value to be specified for vectored and EIC interrupts.
- `badva`: If specified, then write the specified value into C0.BadVAddr. Provides a convenient syntax for specifying the BadVaddr update on address error exceptions.

```
def exception(cause, comment=None, g=None, offset=None, ess=None, badva=None):
    clear_execution_hazards() # Clear all pending cp0 updates needing ehb

    comment_text = G(g,'Guest ') + cause + " exception"
    if comment:
        comment_text += ", " + comment

    # Check for guest reserved redirect exception.
    if (cause == 'RI' and (g or g is None and is_guest_mode()) and C0.GuestCtl0.RI):
        raise hypervisor_exception('GRR', comment_text + " (Guest Reserved Redirect)")

    # Exceptions in debug mode are processed as Debug Exceptions.
    if C0.Debug.DM:
        return debug_exception(cause, comment_text + " taken in debug mode")

    trace(comment_text)

    # In VZ, the EXL change may also trigger a GHFC exception on next
    # instruction. If so, record it here.
    if is_guest_mode() and g is not None:
        if C0.GuestCtl0.MC and not C0.GuestCtl0Ext.FCD:
            trace("GHFC (Status.EXL 0->1)")
            CPU.pending_ghfc_exception = 1

    if not C0.Status.EXL and not (C0.Config3.MCU and CPU.in_iae):
        epc = CPU.prior_pc if CPU.in_bds else CPU.pc
        C0.EPC = zero_extend(epc, from_nbits=64 if C0.Config.AT==2 else 32)
        C0.Cause.BD = CPU.in_bds

    C0.Cause.ExcCode = cause

    # Clear Cause.CE if this is not a CPU exception.
    if cause != 'CPU' and C0.Cause.CE:
```

```
        C0.Cause.CE = 0

    if C0.Config3.BI and not CPU.in_fetch:
        opcode = CPU.opcode
        inst_bytes = CPU.inst_bytes

        C0.BadInstr = (opcode << 16 if inst_bytes == 2 else
                       opcode       if inst_bytes == 4 else
                       opcode >> 16 if inst_bytes == 6 else
                       FatalError("unexpected inst_bytes"))

        if (inst_bytes == 6):
            C0.BadInstrX = (opcode & 0x0000ffff) << 16

    if CPU.in_bds and C0.Config3.BP:
        opcode = CPU.prior_opcode
        inst_bytes = CPU.prior_inst_bytes

        C0.BadInstrP = (opcode << 16 if inst_bytes == 2 else
                        opcode       if inst_bytes == 4 else
                        FatalError("unexpected inst_bytes"))

    if C0.Status.BEV:
        ebase = sign_extend(0xbfc00200, from_nbits=32)
        if (C0.Config3.ISA & 1) and not is_nanomips():
            ebase |= 1
    elif C0.Config.AR == 0:
        ebase = sign_extend(0x80000000, from_nbits=32)
    else:
        ebase = sign_extend(C0.EBase.VA << C0.EBase.VA.shift, from_nbits=32)
        if C0.Config3.ISAOnExc and not is_nanomips():
            ebase |= 1

    if offset is None:
        offset = 0x180

    CPU.next_pc = ebase + offset
    CPU.got_next_next_pc = False  # For branch delay slot handling in simulator.

    if not C0.Status.EXL and not C0.Status.BEV and C0.SRSCtl.HSS:
        C0.SRSCtl.PSS = C0.SRSCtl.CSS
        C0.SRSCtl.CSS = C0.SRSCtl.ESS if ess is None else ess

    C0.Status.EXL = 1

    # For MCU ASE, special processing for interrupt prologue or epilogue.
    if C0.Config3.MCU:
        C0.Cause.AP = CPU.in_iap
```

```
91
92          if CPU.in_iap or CPU.in_iae:
93              # CSS still needs updating even though EXL=1.
94              C0.SRSCtl.CSS = C0.SRSCtl.ESS if ess is None else ess
95
96              CPU.in_iap = 0
97              CPU.in_iae = 0
98
99      if is_mt_target():
100         C0.VPEConf0.XTC = C0.TCBind.CurTC
101
102     if badva is not None:
103         va_bits = 64 if C0.Config.AT==2 else 32
104         C0.BadVAddr = zero_extend(badva, from_nbits=va_bits)
105
106     return EXCEPTION()  # Makes the "raise exception(...)" syntax work.
```

4.15 get_asid()

Return the effective ASID value associated with the current operating mode.

For VZ cores: If the argument `g` is True, return the value for guest. If the argument `g` is False, return the value for root. If the argument `g` is None (the default), return the value for the current operating mode (root or guest).

```
10  def get_asid(g):
11      C0 = CP0(g)
12
13      if C0.Config5.MI:
14          return C0.MemoryMapID.MMID
15
16      if C0.Config4.AE:
17          return C0.EntryHi.ASIDX @ C0.EntryHi.ASID
18      else:
19          return C0.EntryHi.ASID
```

4.16 get_c0_context_value()

Get the C0.Context value for a given bad virtual address.

For VZ cores: If the argument `g` is True, return the value for guest. If the argument `g` is False, return the value for root. If the argument `g` is None (the default) return the value for the current operating mode (root or guest).

```
10  def get_c0_context_value(bva, g):
11      C0 = CP0(g)
```

```
12
13      mask = C0.ContextConfig if C0.Config3.CTXTC else 0x007ffff0
14
15      if mask == 0:
16          return C0.Context
17
18      lz = count_leading_zeros(mask, 32)
19
20      context = C0.Context & ~(2 ** (32 - lz) - 1) | (bva >> lz) & mask
21
22      return context
```

4.17 get_cache_parameters()

Return the encoded cache size parameters for the cache specified by the 'cache_type' argument. 'cache_type' is either 'I', 'D', 'S' or 'T' for the instruction, data, secondary or tertiary caches respectively. The `context` argument can be used to target root or guest on a specific thread, otherwise the current thread and operating mode are used.

The return values are (L, S, A), where L is the encoded line size, S is the encoded sets per way, and A is the encoded associativity (number of ways). The corresponding decoded sizes are:

- `line_size = 2 ** (L + 1) if L else 0`
- `num_sets = 2 ** (S + 6)`
- `num_ways = A + 1`

```
10  def get_cache_parameters(cache_type, context=None):
11      C0 = CP0(context)
12
13      if cache_type == 'I':
14          L = C0.Config1.IL
15          S = C0.Config1.IS if C0.Config1.IS != 7 else -1
16          A = C0.Config1.IA
17
18      elif cache_type == 'D':
19          L = C0.Config1.DL
20          S = C0.Config1.DS if C0.Config1.DS != 7 else -1
21          A = C0.Config1.DA
22
23      elif cache_type == 'S':
24          if C0.Config5.L2C == 0:
25              L = C0.Config2.SL
26              S = C0.Config2.SS
27              A = C0.Config2.SA
28
29          else:  # L2 configuration is specified in GCRs.
30              L = GCR.L2_CONFIG.LINE_SIZE
```

```
            S = GCR.L2_CONFIG.SET_SIZE
            A = GCR.L2_CONFIG.ASSOC

    elif cache_type == 'T':
        if C0.Config5.L2C == 0:
            L = C0.Config2.TL
            S = C0.Config2.TS
            A = C0.Config2.TA

        else:  # L3 configuration is specified in GCRs.
            L = GCR.L3_CONFIG.LINE_SIZE
            S = GCR.L3_CONFIG.SET_SIZE
            A = GCR.L3_CONFIG.ASSOC

    else:
        raise ValueError(f"Invalid cachetype {cache_type}")

    return (L, S, A)
```

4.18 get_num_ftlb_entries()

Return the number of FTLB entries on the current core. The `context` argument can be used to target root or guest on a specific thread, otherwise the current thread and operating mode are used.

```
def get_num_ftlb_entries(context=None):
    C0 = CP0(context)

    if C0.Config.MT == 'FTLBMMU':
        return  (C0.Config4.FTLBWays + 2) * (1 << C0.Config4.FTLBSets)
    else:
        return 0
```

4.19 get_num_tlb_entries()

Return the total number of TLB entries (VTLB + FTLB). The `context` argument can be used to target root or guest on a specific thread, otherwise the current thread and operating mode are used.

```
def get_num_tlb_entries(context=None):
    return get_num_vtlb_entries(context) + get_num_ftlb_entries(context)
```

4.20 get_num_vtlb_entries()

Return the number of VTLB entries on the current core. The `context` argument can be used to target root or guest on a specific thread, otherwise the current thread and operating mode are used.

```
def get_num_vtlb_entries(context=None):
    C0 = CP0(context)

    if is_r6():
        return (C0.Config4.VTLBSizeExt @ C0.Config1.MMUSize) + 1
    else:
        return C0.Config1.MMUSize + 1
```

4.21 ginvt()

Globally invalidate TLBs. Arguments are:

- type - the type of invalidation to perform:
  - type=0: 'invALL - invalidate all non wired entries.
  - type=1: invVA - invalidate all entries which match the specified VA.
  - type=2: invMMID - invalidate all entries which match C0.MemoryMapID.MMID and are not global.
  - type=3: invVAMMID - invalidate all entries which match the specified VA and either match C0.MemoryMapID or are global.
- va: The VA to compare with for invVA or invVAMMID type invalidations.
- context: The targeted context, i.e. root or guest. If 'None' (the default), target the current operating mode.

```
def ginvt(type, va, context=None):
    C0 = CP0(context)

    gid = Root.C0.GuestCtl1.ID if is_guest_mode() else Root.C0.GuestCtl1.RID

    for thread in get_all_threads_in_system(context):

        for index in range( get_num_tlb_entries(thread) ):
            tlb_entry = thread.TLB[index]

            # Nothing to do if entry is already invalid.
            if tlb_entry.ehinv:
                continue

            # For VZ, nothing to do if entry belongs to a different guest.
            if tlb_entry.gid != gid:
                continue

            va_match = not (tlb_entry.va ^ va) & tlb_entry.va_match_mask
            mmid_match = tlb_entry.asid == C0.MemoryMapID.MMID

```

```
            if (type == 0):  # invAll
                invalidate = index >= thread.C0.Wired.Index

            elif (type == 1):  # invVA
                invalidate = va_match

            elif (type == 2):  # invMMID
                invalidate = mmid_match and not tlb_entry.is_global

            elif (type == 3):  # invVAMMID
                invalidate = va_match and (mmid_match or tlb_entry.is_global)

            else:
                FatalError(f"ERROR: invalid ginvt type '{type}'")

            if invalidate:
                thread.TLB[index] = invalid_tlb_entry()
```

4.22 IsCoprocessor0Enabled()

Are Coprocessor 0 operatiions currently enabled?

```
def IsCoprocessor0Enabled():
    if C0.Config.AR < 2 and C0.Status.CU0:
        return True

    return EffectiveKSU() == 0;
```

4.23 is_r6()

Does this core implement Release 6 or later of the MIPS architecture?

```
def is_r6():
    return C0.Config.AR >= 2
```

4.24 overflows()

Does 'value' overflow when we try to represent it as nbits bits signed or unsigned number?

```
def overflows(value, nbits, unsigned=False):
    if unsigned:
        return zero_extend(value, nbits) != value
    else:
        return sign_extend(value, nbits) != value
```

4.25 pointers_are_64_bits()

Are address pointers 64 bits wide for the current operating mode? On MIPS32 cores, pointers_are_64_bits() always returns False.

- The access_type argument specifies what type of memory access we are considering, with allowed values 'Load', 'Store', 'Fetch' and 'Cacheop'.
- The eva argument specifies whether the access in question is for an EVA instruction, i.e. a kernel privilege instruction which translates addresses as though in user mode.

```
def pointers_are_64_bits(access_type=None, eva=False):
    ksu = EffectiveKSU(eva)

    # No 64 bit pointers on a core which does not support 64 bit addressing
    if C0.Config.AT < 2:
        return False

    # On R6 or EVA cores, KX, SX and UX bits are all considered.
    if is_r6() or C0.Config5.EVA:
        if (   (ksu == 0 and C0.Status.KX)
            or (ksu == 1 and C0.Status.SX)
            or (ksu == 2 and C0.Status.UX) ):
            return True
        else:
            return False

    # On Pre-R6 cores with 64 bit addressing, we use 32 bit addressing for
    # user mode loads and stores when UX = 0

    if ( ksu == 2
        and (access_type == 'Load' or access_type == 'Store')
        and not C0.Status.UX):
        return False
    else:
        return True
```

4.26 read_memory_at_va()

Read nbytes bytes of data from virtual address va. If a TLB exception or address error exception occurs as a result of the read, raise it.

The eva argument specifies whether the access in question is for an EVA instruction, i.e. a kernel privilege instruction which translates addresses as though in user mode.

If unaligned_support is 'optional' (the default) and va is unaligned, then an implementation may choose to signal an address error exception. Otherwise, the read will be split into one or more memory ac-

cesses and the results of those accesses will be combined to form the final data value. Any of the memory accesses may cause a TLB or privilege exception.

If unaligned_support is 'always' and va is unaligned, then an implementation may *not* choose to signal an address error exception and the read will always be split into one or more memory accesses with results of those accesses being combined to form the final data value.

```
def read_memory_at_va(va, nbytes, unaligned_support='optional', eva=False):

    if unaligned_support == 'optional':
        # Implementation dependent whether an unaligned access is supported.
        if va & (nbytes-1) and not allow_unaligned_access(va, nbytes, eva):
            raise exception('ADEL', badva=va)
    elif unaligned_support == 'always':
        pass  # No unaligned exception allowed.
    else:
        FatalError("Unknown unaligned_support value")

    accesses = va2pa_split(va, nbytes, 'Load', eva)

    data = 0
    for access in accesses:
        (va, pa, cca, this_nbytes, shift) = access
        this_data = read_memory(va, pa, cca, this_nbytes)
        data |= this_data << shift

    return data
```

4.27 sign_extend()

Sign extend from 'from_nbits' bits and above, then interpret the result as a twos-complement value.

```
def sign_extend(value, from_nbits=None):
    if from_nbits is None:
        from_nbits = value.nbits

    sign_bit_mask = 1 << (from_nbits-1)
    low_bits_mask = sign_bit_mask - 1

    return (value & low_bits_mask) - (value & sign_bit_mask)
```

4.28 synci_step()

Compute the step size which can be used by the synci instruction.

```
def synci_step():
    if C0.Config1.DL == 0 or C0.Config1.IL and C0.Config1.IL < C0.Config1.DL:
        L = C0.Config1.IL
    else:
        L = C0.Config1.DL

    return 2 ** (L + 1) if L else 0
```

4.29 tlb_exception()

Perform the state updates associated with a TLB exception. Arguments are:

- `cause`: The Cause.ExcCode value for this exception.
- `bva`: The bad virtual address value for this exception.
- `gva`: Will contain to the guest virtual address for GPA->RPA lookups on VZ cores. If this is not a GPA->RPA lookup, `gva` will be None. `gva` will be used in place of `bva` on implementations which use GuestCtl0.GExcCode == 'GVA' for the current exception type.
- `comment`: A comment about the reason of the exception, for debug and explanation purposes only.
- `g`: For VZ cores: If `g` is True, take an exception in guest mode. If `g` is False, take an exception in Root mode. If `g` is None, take an exception in the current operating mode (Root or Guest).
- `offset`: The exception vector offset. By default, use the general exception vector (0x180).

```
def tlb_exception(cause, bva, gva, comment, g, offset=None):

    # Make all the exception state updates not specific to TLB exceptions.
    exception(cause, comment, g, offset)

    # No further state changes for TLB exceptions in debug mode.
    if C0.Debug.DM:
        return EXCEPTION()

    # If this is a GPR->RPA lookup on a VZ core...
    if gva is not None:
        # ... it is implementation dependent whether bva gets updated with GVA or GPA.
        if CPU.use_gva:
            Root.C0.GuestCtl0.GExcCode = 'GVA'
            bva = gva
        else:
            Root.C0.GuestCtl0.GExcCode = 'GPA'

    C0.EntryHi.VPN2 = bva[CPU.SEGBITS-1:0] >> 13

    bva_nbits = 64 if C0.Config.AT == 2 else 32
```

```
    C0.BadVAddr = zero_extend(bva, from_nbits=bva_nbits)

    bva_aln = (bva >> 13) << 13
    C0.Context = get_c0_context_value(bva_aln, g)

    if C0.Config.AT == 2:
        C0.XContext = get_c0_xcontext_value(bva_aln, g)

        EntryHi.R64 = (bva >> 62) & 3

    return EXCEPTION()  # Makes the "raise tlb_exception(...)" syntax work.
```

4.30 tlb_lookup_index()

Return the tlb index which maps the specified VA in the current operating mode. If there is no mapping, return None.

For VZ cores: If the argument `guest` is True, do the lookup for guest. If the argument `guest` is False, do the lookup for root. If the argument `guest` is None, do the lookup for the current operating mode (root or guest).

For VZ cores, if the `probe` argument is True, this means the lookup is being carried out for a TLB probe instruction, which means that Root.C0.GuestCtl1.RID is used as the source of the `gid` value if the core is in root mode.

```
def tlb_lookup_index(va, guest, probe=False):
    asid = get_asid(guest)

    if is_guest_mode():
        gid = Root.C0.GuestCtl1.ID
    elif probe:
        # For Root TLBP on VZ cores, we look for match to RID not ID.
        gid = Root.C0.GuestCtl1.RID
    else:
        gid = 0

    index = None

    for i in range( get_num_tlb_entries(guest) ):

        tlb_entry = Context(guest).TLB[i]

        # No match if EHINV bit set for this entry.
        if tlb_entry.ehinv:
            continue

        # No match if ASID does not match and this entry is not global.
```

```
        if tlb_entry.asid != asid and not tlb_entry.is_global:
            continue

        # No match if non-masked bits of VA do not match the VPN2 value.
        if (tlb_entry.va ^ va) & tlb_entry.va_match_mask:
            continue

        # For VZ, no match if the gid does not match.
        if tlb_entry.gid != gid:
            continue

        # Behavior when multiple TLB entries match the same address is
        # implementation dependent. The preferred implementation is a machine
        # check exception on a TLBW which generates an overlapping entry. In
        # in some implementations it may not be possible to detect the
        # overlapping entries until the lookup is performed, in which case the
        # preferred implementation is a machine check exception on the lookup.
        if index is not None:
            raise exception('MCHECK')

        index = i

    return index
```

4.31 tlb_map()

Translate the specified virtual address through the TLB, returning the physical address and the CCA in the case that a valid mapping is present, or raising a TLB exception in the case that the mapping is not present or is not valid.

Arguments are:

- `va`: The virtual address which is to be translated.
- `access_type`: Specifies what type of memory access we are considering, with allowed values 'Load', 'Store', 'Fetch' and 'Cacheop'.
- `use_xtlb_vector`: If a TLB miss exception occurs, should we use the TLB miss exception vector (0x00) or the XTLB miss exception vector (0x80)?
- `guest`: For VZ cores, perform a guest TLB lookup if `guest` is True, perform a root TLB lookup if `guest` is False, else perform a TLB lookup for the current operating mode (root or guest).
- `in_pw`: Is the TLB lookup being triggered as a result of a hardware page table walk?
- `gva`: Specifies the guest virtual address if the lookup is for a GPA->RPA translation on a VZ core. Set to None otherwise.

```
def tlb_map(va, access_type, use_xtlb_vector, guest, in_pw, comment, gva=None):

    # Find the index (if any) of the entry matching this VA.
    index = tlb_lookup_index(va, guest)

    # Trigger hardware page table walk on a TLB miss if PW present and enabled.
    if (index is None
        and (not in_pw or gva is not None)
        and C0.Config3.PW and C0.PWCtl.PWen):
        page_walk(va, guest, use_xtlb_vector)
        index = tlb_lookup_index(va, guest)

    if (index is None):
        if in_pw:
            return

        cause = 'TLBS' if access_type == 'Store' else 'TLBL'

        if C0.Status.EXL == 0 or C0.Config3.MCU and (CPU.in_iae or CPU.in_iap):
            offset = 0x80 if use_xtlb_vector else 0x00  # TLB or XTLB vector?
        else:
            offset = None  # Use default (general) exception vector.

        raise tlb_exception(cause, va, gva, comment, guest, offset)

    entry = Context(guest).TLB[index]

    even_odd_bit = 1 if va & entry.even_odd_bit else 0
    entrylo = entry.entrylo1 if even_odd_bit else entry.entrylo0

    comment += f" idx[{index},{even_odd_bit}]"

    if in_pw and (not entrylo.v or entrylo.ri):
        return

    if not entrylo.v:
        comment += " not valid"
        cause = 'TLBS' if access_type == 'Store' else 'TLBL'
        raise tlb_exception(cause, va, gva, comment, guest)

    if not entrylo.d and access_type == 'Store':
        raise tlb_exception("MOD", va, gva, comment, guest)

    if (entrylo.ri and access_type == 'Load'):
        comment += " read inhibit"
        if C0.PageGrain.IEC:
            raise tlb_exception("TLBRI", va, gva, comment, guest)

```

```
58          raise tlb_exception("TLBL", va, gva, comment, guest)
59      elif (entrylo.xi and access_type == 'Fetch'):
60          comment += " execute inhibit"
61          if C0.PageGrain.IEC:
62              raise tlb_exception("TLBXI", va, gva, comment, guest)
63
64          raise tlb_exception("TLBL", va, gva, comment, guest)
65
66      pa = entrylo.pa | va & (entry.even_odd_bit - 1)
67      cca = entrylo.c
68
69      # direct GPA->RPA mapping if we are using RPU
70      if gva is not None and C0.Config.MT == 'FMTMMU':
71          pa = va
72
73      if guest:
74          comment += f" GPA={xpa_hex(pa)} GCCA={cca}"
75
76      trace(comment)
77
78      return pa, cca
```

4.32 tlbinv()

Carry out a TLBINV operation (when `flush` argument is False) or a TLBINVF operation (when `flush` argument is True).

For VZ cores: If the argument `g` is True, carry out a guest tlb invalidate. If the argument `g` is False, carry out a root tlb invalidate. If the argument `g` is None (the default), carry out a tlb invalidate for the current operating mode (root or guest).

```
10  def tlbinv(flush=False, g=None):
11      C0 = CP0(g)
12
13      if C0.Config4.IE == 2:  # SW FTLB walk.
14          vtlb_size = get_num_vtlb_entries(g)
15
16          if (C0.Index.Index < vtlb_size):
17              # When index targets VTLB, target all entries in VTLB.
18              indexes = range(vtlb_size)
19          else:
20              # When index targets FTLB, target one index per FLTB set.
21              sets = 1 << C0.Config4.FTLBSets
22              ways = 2 + C0.Config4.FTLBWays
23
24              set = (index - vtlb_size) % sets
25
```

```
            indexes = [(vtlb_size + set + way * sets) for way in range(ways)]
    else:
        # Target all TLB indexes.
        indexes = range(get_num_tlb_entries(g))

    asid = get_asid(g)
    gid = Root.C0.GuestCtl1.ID if is_guest_mode() else Root.C0.GuestCtl1.RID

    for index in indexes:
        tlb_entry = Context(g).TLB[index]

        if tlb_entry.ehinv:  # No need to invalidate an invalid entry.
            continue

        if tlb_entry.gid != gid:  # Don't invalidate another guest's entries.
            continue

        if flush or (not tlb_entry.is_global and tlb_entry.asid == asid):
            Context(g).TLB[index] = invalid_tlb_entry()
```

4.33 tlbp()

Carry out a TLBP operation.

For VZ cores: If the argument `g` is True, carry out a guest TLBP. If the argument `g` is False, carry out a root TLBP. If the argument `g` is None (the default), carry out a TLBP for the current operating mode (root or guest).

```
def tlbp(g=None):
    C0 = CP0(g)

    va = C0.EntryHi.VPN2 << 13

    if C0.Config.AT == 2:
        va |= C0.EntryHi.R64 << 62

        if not is_guest_mode() and C0.GuestCtl1.RID and not C0.PageGrain.ELPA:
            if overflows(va, 32, unsigned=True):
                comment("Truncate EntryHi to 32 bits on ELPA=0 GPA->RPA tlbp")
                va = zero_extend(va, from_nbits=32)

    index = tlb_lookup_index(va, g, probe=True)

    if index is None:
        C0.Index.P = 1
        C0.Index.Index = 0
    else:
```

```
        C0.Index.P = 0
        C0.Index.Index = index
```

4.34 tlbr()

Carry out a TLBR operation, that is, read the TLB entry information associated with the current C0.Index value into the TLB interface registers (C0. EntryHi, C0.EntryLo0, C0.EntryLo1, C0.PageMask, etc.).

For VZ cores: If the argument g is True, carry out a guest TLBR. If the argument g is False, carry out a root TLBR. If the argument g is None (the default), carry out a TLBR for the current operating mode (root or guest).

```
def tlbr(g=None):
    C0 = CP0(g)

    tlb_entry = Context(g).TLB[C0.Index.Index]

    # guest cannot read another guest's entries
    if is_guest_mode() and tlb_entry.gid != Root.C0.GuestCtl1.ID:
        tlb_entry = invalid_tlb_entry()

    C0.EntryHi.EHINV = tlb_entry.ehinv
    C0.EntryHi.VPN2 = tlb_entry.vpn2
    C0.PageMask = tlb_entry.pagemask

    if C0.Config.AT == 2:
        C0.EntryHi.R64 = tlb_entry.r64

    if not is_guest_mode():
        Root.C0.GuestCtl1.RID = tlb_entry.gid

    C0.EntryLo0.PFN = tlb_entry.entrylo0.pfn
    C0.EntryLo0.C = tlb_entry.entrylo0.c
    C0.EntryLo0.V = tlb_entry.entrylo0.v
    C0.EntryLo0.D = tlb_entry.entrylo0.d
    C0.EntryLo0.G = tlb_entry.entrylo0.g

    C0.EntryLo1.PFN = tlb_entry.entrylo1.pfn
    C0.EntryLo1.C = tlb_entry.entrylo1.c
    C0.EntryLo1.V = tlb_entry.entrylo1.v
    C0.EntryLo1.D = tlb_entry.entrylo1.d
    C0.EntryLo1.G = tlb_entry.entrylo1.g

    if C0.PageGrain.ELPA:
        C0.EntryLo0.RI64 = tlb_entry.entrylo0.ri
        C0.EntryLo0.XI64 = tlb_entry.entrylo0.xi
```

```
        C0.EntryLo0.RI = 0
        C0.EntryLo0.XI = 0

        C0.EntryLo1.RI64 = tlb_entry.entrylo1.ri
        C0.EntryLo1.XI64 = tlb_entry.entrylo1.xi
        C0.EntryLo1.RI = 0
        C0.EntryLo1.XI = 0
    else:
        C0.EntryLo0.RI = tlb_entry.entrylo0.ri
        C0.EntryLo0.XI = tlb_entry.entrylo0.xi
        C0.EntryLo0.RI64 = 0
        C0.EntryLo0.XI64 = 0

        C0.EntryLo1.RI = tlb_entry.entrylo1.ri
        C0.EntryLo1.XI = tlb_entry.entrylo1.xi
        C0.EntryLo1.RI64 = 0
        C0.EntryLo1.XI64 = 0

    # Set the relevent asid bits for this core configuration.
    if C0.Config5.MI:
        C0.MemoryMapID.MMID = tlb_entry.asid
    elif C0.Config4.AE:
        C0.EntryHi.ASID = tlb_entry.asid & C0.EntryHi.ASID.mask
        C0.EntryHi.ASIDX = tlb_entry.asid >> C0.EntryHi.ASIDX.shift
    else:
        C0.EntryHi.ASID = tlb_entry.asid
```

4.35 tlbwi()

Write the TLB entry information as specified in the TLB interface registers (C0.EntryHi, C0.EntryLo0, C0.EntryLo1, C0.PageMask, etc.) to the TLB entry indexed by the argument `index`.

For VZ cores: If the argument `g` is True, write to the guest TLB. If the argument `g` is False, write to the root TLB. If the argument `g` is None (the default), write to the TLB for the current operating mode (root or guest).

```
def tlbwi(index, g=None):
    C0 = CP0(g)

    is_m64 = C0.Config.AT == 2

    # Find the alignment of EntryLo0/1.PFN according to the small page settings.
    if C0.Config3.SP and C0.PageGrain.ESP:
        pfn_shift = 11
    elif C0.Config3.SM:
        if C0.PageGrain.Mask == 0:
            pfn_shift = 11
```

```
        elif C0.PageGrain.Mask == 1:
            pfn_shift = 12
        elif C0.PageGrain.Mask == 3:
            pfn_shift = 13
        else:
            FatalError("Unexpected PageGrain.Mask value")
    else:
        pfn_shift = 13

    # Find the gid (guest ID) value for the current operating mode.
    if Root.C0.Config3.VZ == 0:
        gid = 0
    elif is_guest_mode():
        gid = Root.C0.GuestCtl1.ID
    else:
        gid = Root.C0.GuestCtl1.RID

    is_global = C0.EntryLo0.G and C0.EntryLo1.G

    if not g and gid and not is_guest_mode():
        is_global = 1

    va = C0.EntryHi.VPN2 << 13
    if is_m64:
        va |= C0.EntryHi.R64 << 62

    page_mask = C0.PageMask | (2 ** pfn_shift - 1)
    va_match_mask = zero_extend(~page_mask, from_nbits=CPU.SEGBITS)
    if is_m64:
        va_match_mask |= C0.EntryHi.R64.mask

    even_odd_bit = page_mask ^ (page_mask >> 1)

    # Base PAs for the even and odd pages
    pa0 = (C0.EntryLo0.PFN << (pfn_shift-1)) & ~(even_odd_bit - 1)
    pa1 = (C0.EntryLo1.PFN << (pfn_shift-1)) & ~(even_odd_bit - 1)

    if is_m64 and not G(g) and gid and not Root.C0.PageGrain.ELPA:
        if overflows(va, 32, unsigned=True):
            va = zero_extend(va, from_nbits=32)
            comment("Truncating EntryHi to 32 bits on ELPA=0 tlbw of GPA->RPA")

    # 'tlb_entry' is an object storing all the information associated with a
    # TLB entry. Start with an invalid entry (EHINV==1, all other bits 0)
    tlb_entry = invalid_tlb_entry()

    # If we're creating a valid TLB entry, fill in all the state details.
    if C0.EntryHi.EHINV == 0:
```

```
        tlb_entry.ehinv = 0
        tlb_entry.r64 = C0.EntryHi.R64 if is_m64 else 0
        tlb_entry.vpn2 = C0.EntryHi.VPN2
        tlb_entry.asid = get_asid(g)
        tlb_entry.is_global = is_global
        tlb_entry.is_m64 = is_m64
        tlb_entry.gid = gid
        tlb_entry.pagemask = C0.PageMask
        tlb_entry.va = va
        tlb_entry.va_match_mask = va_match_mask
        tlb_entry.pfn_shift = pfn_shift
        tlb_entry.even_odd_bit = even_odd_bit

        tlb_entry.entrylo0.pa = pa0
        tlb_entry.entrylo0.pa = pa0
        tlb_entry.entrylo0.pfn = C0.EntryLo0.PFN
        tlb_entry.entrylo0.c = C0.EntryLo0.C
        tlb_entry.entrylo0.v = C0.EntryLo0.V
        tlb_entry.entrylo0.d = C0.EntryLo0.D
        tlb_entry.entrylo0.g = is_global
        tlb_entry.entrylo0.ri = C0.EntryLo0.RI | C0.EntryLo0.RI64
        tlb_entry.entrylo0.xi = C0.EntryLo0.XI | C0.EntryLo0.XI64

        tlb_entry.entrylo1.pa = pa1
        tlb_entry.entrylo1.pa = pa1
        tlb_entry.entrylo1.pfn = C0.EntryLo1.PFN
        tlb_entry.entrylo1.c = C0.EntryLo1.C
        tlb_entry.entrylo1.v = C0.EntryLo1.V
        tlb_entry.entrylo1.d = C0.EntryLo1.D
        tlb_entry.entrylo1.g = is_global
        tlb_entry.entrylo1.ri = C0.EntryLo1.RI | C0.EntryLo1.RI64
        tlb_entry.entrylo1.xi = C0.EntryLo1.XI | C0.EntryLo1.XI64

    Context(g).TLB[index] = tlb_entry
```

4.36 tlbwr()

Choose a valid random tlb index, and write the TLB entry information as specified in the TLB interface registers (C0.EntryHi, C0.EntryLo0, C0.EntryLo1, C0.PageMask, etc.) to that TLB entry.

For VZ cores: If the argument `g` is True, write to the guest TLB. If the argument `g` is False, write to the root TLB. If the argument `g` is None (the default), write to the TLB for the current operating mode (root or guest).

The pseudocode presented here shows a specific implementation of the TLBWR operation. It illustrates the rules for determining which TLB indexes are possible targets for TLBWR as a function of the configuration and operation mode of the processor, but otherwise is provided for reference purposes only.

```
def tlbwr(g=None):
    C0 = CP0(g)

    num_vtlb_entries = get_num_vtlb_entries(g)
    num_ftlb_entries = get_num_ftlb_entries(g)

    if num_ftlb_entries:
        # FTLB not available if PageMask does not match FTLBPageSize
        page_aln = 31 - count_leading_zeros(C0.PageMask | 0x1fff, 32)
        if page_aln != 2 * C0.Config4.FTLBPageSize + 10:
            num_ftlb_entries = 0

    min_index = C0.Wired.Index
    max_index = num_vtlb_entries + num_ftlb_entries - 1

    # In this illustrative implementation, we will keep trying until we find a
    # random index which is consistent with the constraints. We impose an
    # arbitrary limit on the number of tries to avoid a simulator hang in case
    # the index constraints are inconsistent.
    try_count = 0
    while 1:
        if ++try_count > 99999:
            FatalError("Failed to find valid index for tlbwr")

        # The random distribution for the index is implementation dependent,
        # and could include factors such as "least recently used", and
        # different probabilities of choosing an FTLB vs a VTLB entry. This
        # illustrative implementation uses a uniform distribution for
        # simplicity.
        random_index = random.randint(min_index, max_index)

        # In R6, the entry pointed to by Index is forbidden if Index.P=0.
        if is_r6() and not C0.Index.P and random_index == C0.Index.Index:
            continue  # The index was bad, try again.

        # Disallow FTLB indexes which are not valid for the current TLB entry
        # parameters.
        if is_invalid_ftlb_index(random_index):
            continue  # The index was bad, try again.

        break  # The index was ok, exit the loop.

    tlbwi(random_index, g)
```

4.37 unsigned()

Convert a GPR sized signed value to an unsigned value.

```
def unsigned(value):
    nbits = 32 if C0.Config.AT == 0 else 64
    return zero_extend(value, from_nbits=nbits)
```

4.38 va2pa()

Get the PA and CCA for a given VA, as a function of the current state of the processor. If the address lookup causes an exception, signal it.

- The va argument specifies the virtual address to be translated.
- The access_type argument specifies what type of memory access we are considering, with allowed values 'Load', 'Store', 'Fetch' and 'Cacheop'.
- The eva argument specifies whether the access in question is for an EVA instruction, i.e. a kernel privilege instruction which translates addresses as though in user mode.
- The in_pw argument species whether the current lookup is occurring as a result of a hardware page table walk.

```
def va2pa(va, access_type, eva=False, in_pw=False):
    g = is_guest_mode()

    # check for possible hit in mpu
    mpu_check(va, access_type, eva)

    # check for possible hit in watch registers
    watch_check(va, access_type, g)

    translation_type, description, result_args = decode_va(va, eva)

    decode_comment = access_type + " " + description
    if g:
        decode_comment = "Guest " + decode_comment

    if translation_type == 'ade':
        if in_pw:
            raise RuntimeError("unexpected address error on pw va2pa")

        cause = "ADES" if access_type == 'Store' else "ADEL"
        raise exception(cause, badva=va, comment=decode_comment)

    elif translation_type == 'map':
        (use_xtlb_vector) = result_args

        (pa, cca) = tlb_map(
            va              = va,
```

```
            access_type     = access_type,
            use_xtlb_vector = use_xtlb_vector,
            comment         = decode_comment,
            guest           = g,
            in_pw           = in_pw)

        if in_pw and pa is None:
            return

    elif translation_type == 'unmapped':
        (pa, cca) = result_args

    else:
        FatalError("unknown translation_type " + translation_type)

    if g:
        (rpa, rcca) = tlb_map(
            va              = pa,
            gva             = va,
            access_type     = access_type,
            use_xtlb_vector = Root.C0.Status.KX,
            comment         = access_type + " GPA->RPA",
            guest           = FALSE,
            in_pw           = in_pw)

        if in_pw and rpa is None:
            return

        ncc = Root.C0.GuestCtl0Ext.NCC
        if ncc == 0:
            pass # cca unmodified
        elif ncc == 1:
            if rcca == 2 or (rcca == 7 and cca != 2):
                cca = rcca
        elif ncc == 2:
            if rcca != 0 and not (rcca == 7 and cca == 2):
                cca = rcca
        else:
            FatalError("unknown NCC value " + ncc)

        return (rpa, cca)

    return (pa, cca)
```

4.39 va2pa_split()

Split a potentially unaligned memory access for nbytes bytes at VA=va into sub-accesses which are guaranteed to share the same va2pa mapping. Signal any TLB or address privilege errors which occur for either mapping.

This function demonstrates one specific implementation which guarantees that the memory access gets split into groups of bytes which all share the same address mapping. There are other implementation choices which could achieve the same goal.

```
def va2pa_split(va, nbytes, access_type, eva):

    pa, cca = va2pa(va, access_type, eva)

    # The first VA which *might* need a different mapping.
    va_of_next_1k_page = va + 1024 - (va & 1023)

    if va + nbytes - 1 < va_of_next_1k_page:
        # Only one mapping required.
        shift = 0
        return ( (va, pa, cca, nbytes, shift), )
    else:
        # Two mappings might be necessary, split into two accesses.

        # Take care of possible address wrapping for the second access.
        va2 = effective_address(va_of_next_1k_page-1, 1, access_type, eva)

        trace("va =" + _hex(va,16))
        trace("va2=" + _hex(va2,16))

        # Get mapping for the second access.
        (pa2, cca2) = va2pa(va2, access_type, eva)

        nbytes1 = va_of_next_1k_page - va
        nbytes2 = nbytes - nbytes1

        (shift1, shift2) = ((8*nbytes2, 0) if C0.Config.BE else (0, 8*nbytes1))

        return (
            (va,  pa,  cca,  nbytes1, shift1),
            (va2, pa2, cca2, nbytes2, shift2),
        )
```

4.40 write_memory_at_va()

Write the nbytes byte data value value to virtual address va. If a TLB exception or address error exception occurs as a result of the write, raise it.

The eva argument specifies whether the access in question is for an EVA instruction, i.e. a kernel privilege instruction which translates addresses as though in user mode.

If unaligned_support is 'optional' (the default) and va is unaligned, then an implementation may choose to signal an address error exception. Otherwise, the write will be split into one or more memory accesses. Any of the memory accesses may cause a TLB or privilege exception. If such an exception occurs on any of the accesses, none of the memory writes associated with any of the accesses may occur.

If unaligned_support is 'always' and va is unaligned, then an implementation may *not* choose to signal an address error exception and the write will always be split into one or more memory accesses. Although it is not shown explicitly in the pseudocode, if a TLB or address exception occurs on one of the accesses, it is possible that one of the other accesses will already have committed its result to memory.

```
def write_memory_at_va(value, va, nbytes, unaligned_support='optional', eva=False):
    if unaligned_support == 'optional':
        # Implementation dependent whether an unaligned access is supported.
        if va & (nbytes-1) and not allow_unaligned_access(va, nbytes, eva):
            raise exception('ADES', badva=va)

    elif unaligned_support == 'always':
        pass  # No unaligned exception allowed.
    else:
        FatalError("Unknown unaligned_support value")

    accesses = va2pa_split(va, nbytes, 'Store', eva)

    for access in accesses:
        (va, pa, cca, this_nbytes, shift) = access

        this_data = (value >> shift) & (2**(8*this_nbytes) - 1)
        write_memory(this_data, va, pa, cca, this_nbytes)
```

4.41 zero_extend()

Zero extend from 'from_nbits' bits and above, then interpret the result as an unsigned value.

```
def zero_extend(value, from_nbits):
    low_bits_mask = (1 << from_nbits) - 1
    return value & low_bits_mask
```

Appendix A

Specification Conventions

A.1 Assembly Instruction vs. Hardware Format

Instruction definitions in this manual are grouped alphabetically by assembly mnemonic name. For each assembly mnemonic, there will be one or more hardware formats which can execute the instruction. Different hardware formats for the same instruction are distinguished by a string suffix in square parentheses. For instance, the ANDI assembly instruction has two hardware formats: ANDI[16] and ANDI[32]. The assembler will convert an instance of an ANDI instruction in assembly code to the hardware format which it considers most appropriate, taking into account the fact that some hardware formats only support a specific subset of registers or range of immediate values. By default, a hardware format with the smallest possible instruction length will be used. The user may force an instruction format with a specific length to be used by appending the instruction size in bits to the end of the instruction name. For instance, using 'ADDIU16', 'ADDIU32' or 'ADDIU48' will force the assembler to pick a 16, 32 or 48 bit ADDIU instruction respectively. If no hardware format of the correct length for the requested instruction and arguments is available, the assembler will report an error.

For completeness, it is worth noting that in some cases the assembler may also expand an assembler instruction into a sequence of more than one hardware instruction, unless assembler options are turned on which explicitly disable macro expansion, or a specific instruction length is required by use of a '16', '32' or '48' suffix in the instruction name.

A.2 Instruction byte ordering and endianness

nanoMIPS™ instructions are encoded as 2, 4 or 6 byte data values, with the appropriate encoding value for each instruction shown in the various "Format" sections in this manual. Once the instruction data value is known, it remains to specify the order in which the bytes of the instruction are to be stored in memory.

Instructions are stored as one or more 16 bit parts, as follows:

- 16 bit instructions are stored as one 16-bit part located at the instruction address.
- 32 bit instructions are stored as two 16-bit parts, with the most significant halfword located at the instruction address, and the least significant halfword located at the instruction address + 2.

- 48 bit instructions are stored as three 16-bit parts, with the most significant halfword located at the instruction address, the next most significant halfword located at the instruction address + 2, and the least significant halfword located at the instruction address + 4.

For each 16-bit part, the order of the two bytes within the halfword depends on the endianness of the machine.

- For big-endian machines, the most significant byte is stored at the halfword address, and the least significant byte is stored at the halfword address + 1.
- For little-endian machines, the least significant byte is stored at the halfword address, and the most significant bye is stored at the halfword address + 1.

The procedure for reading or writing a nanoMIPS™ instruction to memory from a running program is the same for either a big-endian or a little-endian CPU, and is as follows.

- The most significant 16 bits of the instruction encoding can be read/written using a LHU/SH instruction targeting the instruction address.
- The next most significant 16 bits of the instruction encoding (for instructions which are 32 or more bits long) can be read/written using an LHU/SH instruction targeting the instruction address + 2.
- The least significant 16 bits of the instruction (for instructions which are 48 bits long) can be read/written using an LHU/SH instruction targeting the instruction address + 4.

As an example, consider the instruction ”MOVZ r1, r2, r3”. As per the MOVZ format, this is encoded as the value 0x20620a10. The most significant 16 bits of the instruction are 0x2062, and this halfword is stored at the instruction address. The least significant 16 bits of the instruction are 0x0a10, and this halfword is stored at the instruction address + 2. The contents of the 4 bytes starting at the instruction address on big and little endian machines are therefore:

```
Instruction: MOVZ r1, r2, r3
Encoding: 0x20620a10

  Byte address:               PC+0 PC+1 PC+2 PC+3
  Byte value (Big-endian):    0x20 0x62 0x0a 0x10
  Byte value (Little-endian): 0x62 0x20 0x10 0x0a
```

A.3 Pseudocode methodology

A.3.1 Python for pseudocode

The pseudocode in this document is executable Python, and the reader should refer to Python documentation to understand the basic syntax.

Formally, all integers in the pseudocode are unbounded, that is, they have a potentially infinite number of bits. The hardware should be implemented in such a way as to get the same result for all architecturally visible state that the pseudocode (with potentially infinite bits) does. In practice, hardware will do this using a finite number of bits, and the designer should use the most efficient way to get the same result as the pseudocode.

A.3.2 The 'raise' keyword

The "raise" keyword in the pseudocode indicates that execution of the current instruction will terminate on completion of the current expression. See for instance the ADD instruction definition. The code

```
if overflows(sum, nbits=32):
    raise exception('OV')
```

means: In the case that overflow occurs, call the `exception()` function, passing "`OV`" as argument, then execute no more pseudocode for this instruction. "raise" may also be called from within shared pseudocode functions.

A.3.3 Bitfield syntax using the "[]" (slicing) operator

The "[]" (slicing) operator in python is usually applied to list type objects to read or write a specific range of list elements. In our pseudocode, we make use of the slicing operator with integers to provide a Verilog style syntax for accessing bit ranges. For instance, "s[21]" means bit 21 of integer value s, and "s[20:1]" means the bitfield of s starting with most significant bit 20 and ending with least significant bit 1. Bit ranges specified in this way can be both read and written to.

The bit range operation always returns an unsigned (i.e. zero extended) value. To get a signed value, the result must be explicitly sign extended (for example "sign_extend(s[20:0])").

A.3.4 Bitstring concatenation using the '@' operator

The '@' operator in python is usually used for matrix multiplication, but we don't need to do that in our CPU pseudocode, so instead we make use of '@' as a convenient shorthand for concatenating binary bit strings to generate a number.

For instance, the expression

```
s[21] @ s[20:1] @ '0'
```

means "concatenate bit 21 of integer s, bit range [20:1] of integer s, and the binary value '0' together to form an integer value".

Using the '@' (concatenation) operator only makes sense when the value on the right hand side has a known width. Hence '@' is only used when the width of the right hand value can be inferred from its context.

A.3.5 Signed vs unsigned values

The pseudocode treats the values stored in GPRs as signed integers. For instructions where the operation takes an unsigned GPR value as input, this will be shown explicitly in the pseudocode by converting the signed GPR value to an unsigned value before carrying out the operation. For instance, in the BGEUC instruction definition, the branch condition pseudocode is:

```
10 if unsigned(GPR[rs]) >= unsigned(GPR[rt]):
11     CPU.next_pc = address
```

The values stored in CP0 registers and CP0 bitfields are treated as unsigned integers. For cases where the value needs to be interpreted as signed, this will be shown explicitly in the pseudocode by converting the unsigned value to a signed value.

A.3.6 Decode vs. Operation Pseudocode

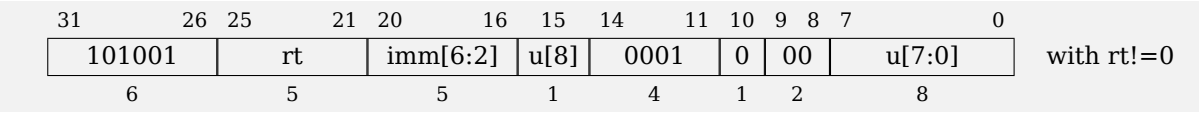
Different hardware formats for the same instruction will typically have different decode behavior, combined with common behavior in a shared execution stage. This fact is reflected the way the operation pseudocode is specified this document. Each hardware ”Format” may have its own ’decode’ pseudocode, where common arguments are derived. There will also be shared pseudocode in the ”Operation” definition for the instruction. The effective behavior for each hardware format is the combination of the ”Format” and ”Operation” pseudocode sections.

A.3.7 Shared pseudocode functions

Shared pseudocode functions referenced by the instruction definitions are given in the Shared Pseudocode Functions section.

A.4 Instruction arguments

Hardware instruction format specifications are shown in encoding diagrams like this example:



Binary values in the hardware encoding diagrams represent specific bit patterns which an instruction bitstring must match to be decoded as an instance of the current instruction. The remaining fields are named instruction arguments.

The ”with rt!=0” qualifier indicates an additional condition which the arguments must meet in order for this instruction bitstring to be decoded as an instance of the current instruction.

The instruction pseudocode describes the behavior of the instruction in terms of operations using the named instruction arguments as input. Some details about the way instructions arguments are specified are presented in the following subsections.

A.4.1 Argument alignment

For arguments which are not right justified, that is, where the bitfield does not represent the least significant bits of the corresponding argument, then the bit range is shown in square parentheses. For instance, the field 'imm[6:2]' in the above example represents an argument 'imm', where 'imm = imm[6:2] << 2'.

A single argument may be specified using multiple fields in the instruction encoding, with the bit range of each field shown square parentheses. For instance the fields 'u[7:0]' and 'u[8]' in the above example represent two parts of a single argument 'u', where `u = (u[8] << 8) | u[7:0]`.

If no explicit bit range is specified for an argument, as for 'rt' in the above example, then the argument represents the least significant bits of the value.

A.4.2 Signed vs unsigned arguments

By convention, the hardware argument name 's' is always used for instruction immediate values which will be sign extended in the instruction operation. The sign extension of 's' values is also always shown explicitly in the instruction pseudocode. Hardware arguments with any other name are always unsigned.

A.4.3 'x' fields

Some instructions have argument fields called 'x'. These fields should be ignored by hardware, and set to zero by software. This methodology allows for the possibility that the meaning of `x` values other than zero might be enhanced in future revisions of the ISA. Any future enhancement will be made with the understanding that cores prior to the enhanced definition will treat the `x!=0` encodings as equivalent to the `x==0` instruction. Older software will never inadvertently trigger future enhanced behavior provided it follows the "only use `x==0` encodings" rule.

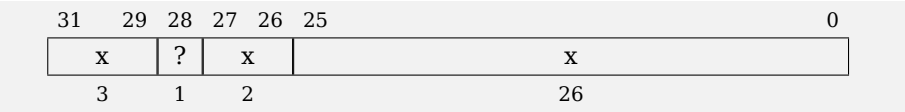
A.4.4 Hardware vs. assembler arguments

Arguments in assembler formats either correspond directly to the hardware arguments or are derived from those arguments. In the case that the assembler arguments are derived, the rules for deriving them will be given as part of the instruction pseudocode. See for example the ANDI[16] format definition for the ANDI instruction. The hardware format contains a field called "`eu`". The assembler format contains a field called "`u`". The ANDI[16] pseudocode specifies the rule for deriving `u` from `eu`. When writing this instruction in assembly code, the non-encoded value `u` should be specified. The assembler will invert the rule specified by the pseudocode to infer the `eu` value to be used in the instruction encoding.

Appendix B

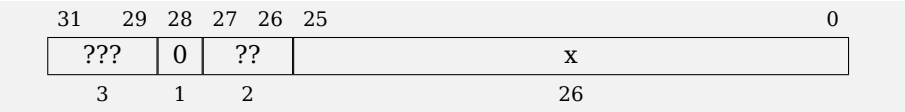
Opcode Map

B.1 MAJOR pool



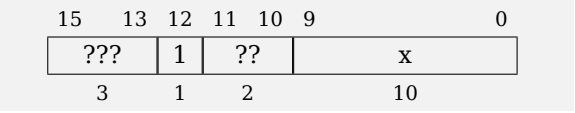
0	1
P32	P16

B.2 P32 pool



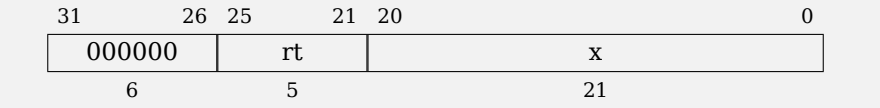
	???00	???01	???10	???11
000??	P.ADDIU	ADDIUPC[32]	MOVE.BALC	*
001??	P32A	*	P.BAL	*
010??	P.GP.W	P.GP.BH	P.J	*
011??	P48I	*	*	*
100??	P.U12	P.LS.U12	P.BR1	*
101??	*(CP1)	P.LS.S9	P.BR2	*
110??	*(MIPS64)	*	P.BRI	*
111??	P.LUI	*	*	*

B.3 P16 pool



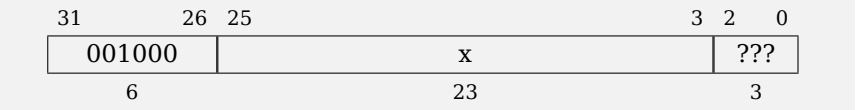
	???00	???01	???10	???11
000??	P16.MV	LW[16]	BC[16]	P16.SR
001??	P16.SHIFT	LW[SP]	BALC[16]	P16.4X4
010??	P16C	LW[GP16]	*	P16.LB
011??	P16.A1	LW[4X4]	*	P16.LH
100??	P16.A2	SW[16]	BEQZC[16]	*
101??	P16.ADDU	SW[SP]	BNEZC[16]	MOVEP
110??	LI[16]	SW[GP16]	P16.BR	*
111??	ANDI[16]	SW[4X4]	*	MOVEP[REV]

B.4 P.ADDIU pool



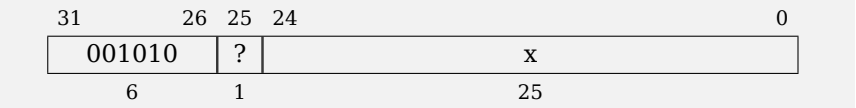
- P.RI: rt==0
- ADDIU[32]: rt!=0

B.5 P32A pool



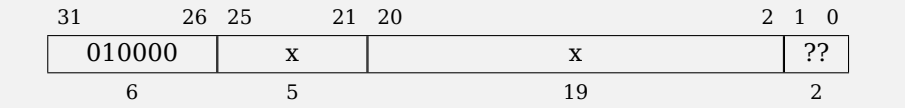
000	_POOL32A0
001	*(UDI)
010	*(CP2)
011	*(UDI)
100	*
101	*(DSP)
110	*
111	_POOL32A7

B.6 P.BAL pool



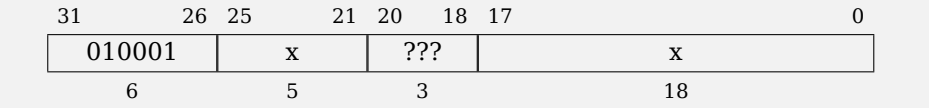
0	1
BC[32]	BALC[32]

B.7 P.GP.W pool



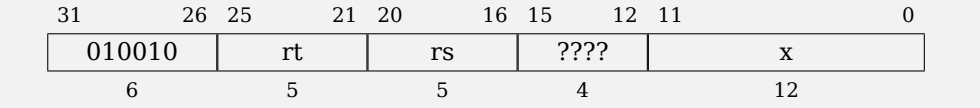
00	01	10	11
ADDIU[GP.W]	*(MIPS64)	LW[GP]	SW[GP]

B.8 P.GP.BH pool



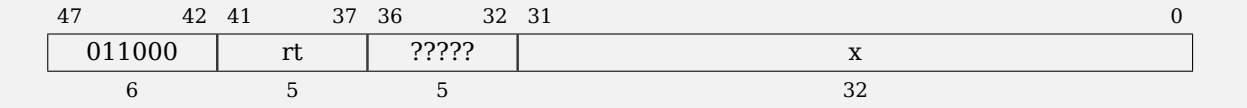
	?00	?01	?10	?11
0??	LB[GP]	SB[GP]	LBU[GP]	ADDIU[GP.B]
1??	P.GP.LH	P.GP.SH	*(CP1)	*(MIPS64)

B.9 P.J pool



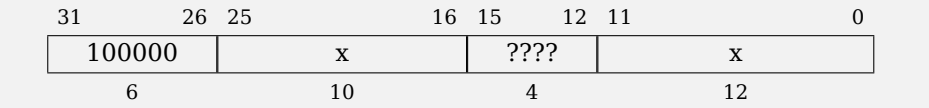
	??00	??01	??10	??11
00??	JALRC[32]	JALRC.HB	*	*
01??	*	*	*	*
10??	P.BALRSC	*	*	*
11??	*	*	*	*

B.10 P48I pool



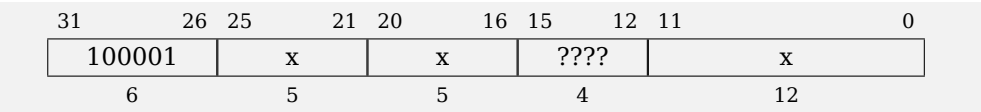
	???00	???01	???10	???11
000??	LI[48]	ADDIU[48]	ADDIU[GP48]	ADDIUPC[48]
001??	*	*	*	*
010??	*	*	*	LWPC[48]
011??	*	*	*	SWPC[48]
100??	*	*(MIPS64)	*	*
101??	*(MIPS64)	*	*	*
110??	*	*	*	*(MIPS64)
111??	*	*	*	*(MIPS64)

B.11 P.U12 pool



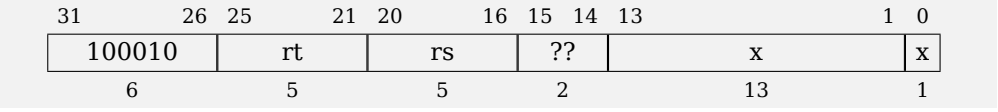
	??00	??01	??10	??11
00??	ORI	XORI	ANDI[32]	P.SR
01??	SLTI	SLTIU	SEQI	*
10??	ADDIU[NEG]	*(MIPS64)	*(MIPS64)	*(MIPS64)
11??	P.SHIFT	P.ROTX	P.INS	P.EXT

B.12 P.LS.U12 pool



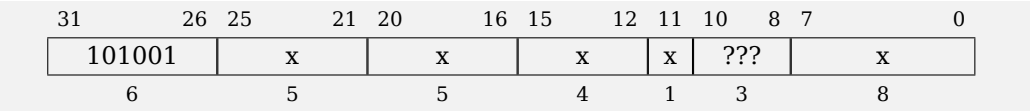
	??00	??01	??10	??11
00??	LB[U12]	SB[U12]	LBU[U12]	P.PREF[U12]
01??	LH[U12]	SH[U12]	LHU[U12]	*(MIPS64)
10??	LW[U12]	SW[U12]	*(CP1)	*(CP1)
11??	*(MIPS64)	*(MIPS64)	*(CP1)	*(CP1)

B.13 P.BR1 pool



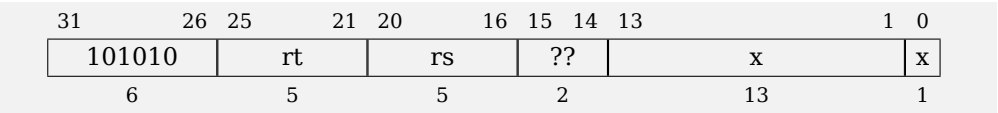
00	01	10	11
BEQC[32]	P.BR3A	BGEC	BGEUC

B.14 P.LS.S9 pool



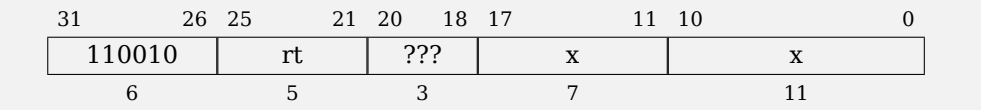
	?00	?01	?10	?11
0??	P.LS.S0	P.LS.S1	P.LS.E0	*
1??	P.LS.WM	P.LS.UAWM	*(MIPS64)	*(MIPS64)

B.15 P.BR2 pool



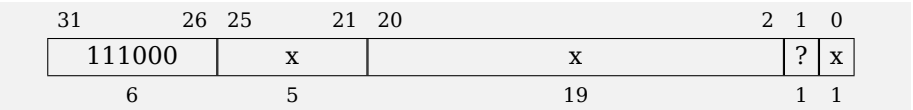
00	01	10	11
BNEC[32]	*	BLTC	BLTUC

B.16 P.BRI pool



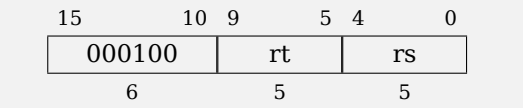
	?00	?01	?10	?11
0??	BEQIC	BBEQZC	BGEIC	BGEIUC
1??	BNEIC	BBNEZC	BLTIC	BLTIUC

B.17 P.LUI pool



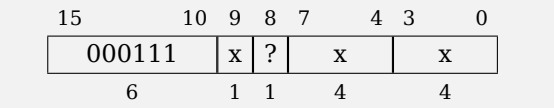
0	1
LUI	ALUIPC

B.18 P16.MV pool



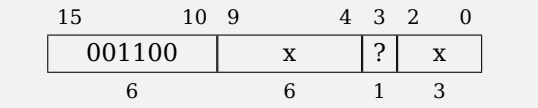
- P16.RI: rt==0
- MOVE: rt!=0

B.19 P16.SR pool



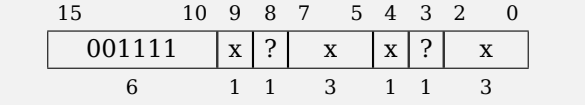
0	1
SAVE[16]	RESTORE.JRC[16]

B.20 P16.SHIFT pool



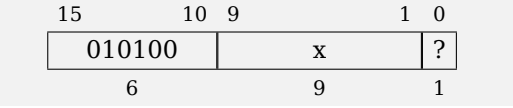
0	1
SLL[16]	SRL[16]

B.21 P16.4X4 pool



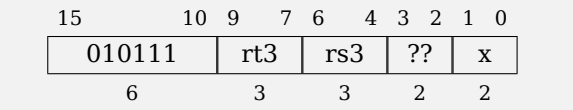
00	01	10	11
ADDU[4X4]	MUL[4X4]	*	*

B.22 P16C pool



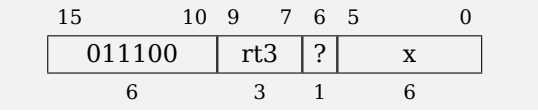
0	1
POOL16C_0	LWXS[16]

B.23 P16.LB pool



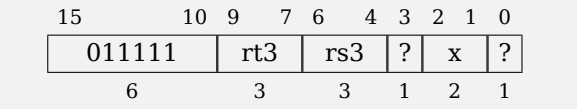
00	01	10	11
LB[16]	SB[16]	LBU[16]	*

B.24 P16.A1 pool



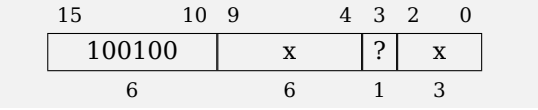
0	1
*	ADDIU[R1.SP]

B.25 P16.LH pool



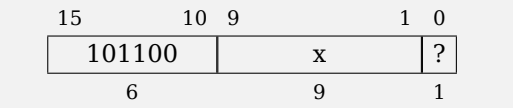
00	01	10	11
LH[16]	SH[16]	LHU[16]	*

B.26 P16.A2 pool



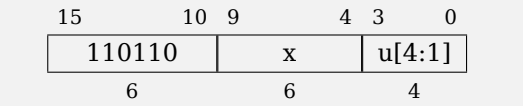
0	1
ADDIU[R2]	P.ADDIU[RS5]

B.27 P16.ADDU pool



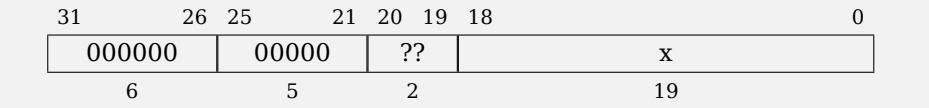
0	1
ADDU[16]	SUBU[16]

B.28 P16.BR pool



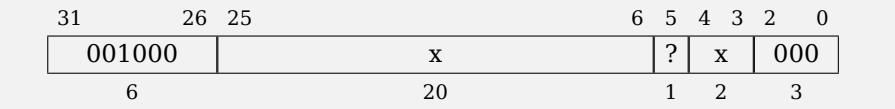
- P16.JRC: u==0
- P16.BR1: u!=0

B.29 P.RI pool



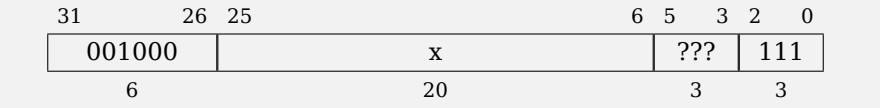
00	01	10	11
SIGRIE	P.SYSCALL	BREAK[32]	SDBBP[32]

B.30 _POOL32A0 pool



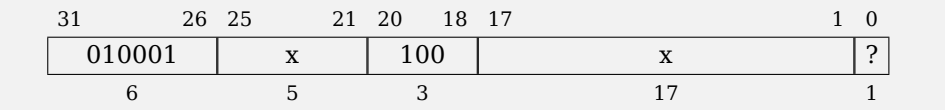
0	1
_POOL32A0_0	_POOL32A0_1

B.31 _POOL32A7 pool



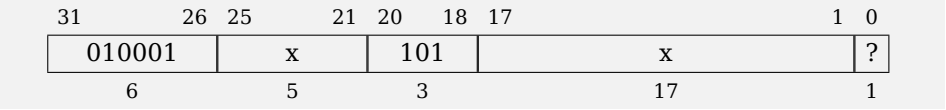
000	001	010	011	100	101	110	111
P.LSX	LSA	*	EXTW	*	*	*	POOL32Axf

B.32 P.GP.LH pool



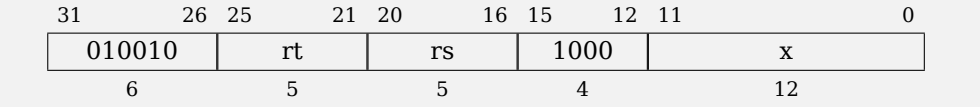
0	1
LH[GP]	LHU[GP]

B.33 P.GP.SH pool



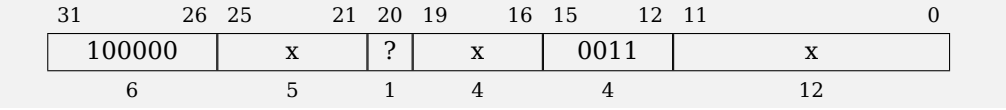
0	1
SH[GP]	*

B.34 P.BALRSC pool



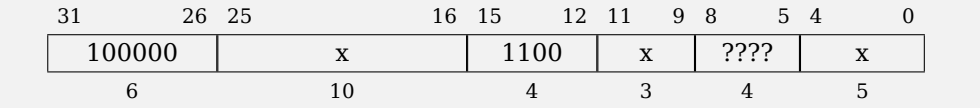
- BRSC: rt==0
- BALRSC: rt!=0

B.35 P.SR pool



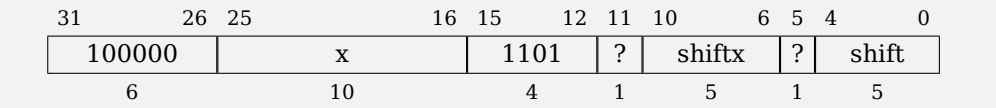
0	1
PP.SR	*(CP1)

B.36 P.SHIFT pool



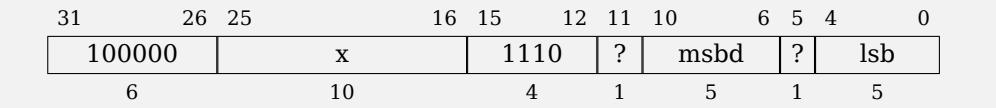
	?000	?001	?010	?011	?100	?101	?110	?111
0???	P.SLL	*	SRL[32]	*	SRA	*	ROTR	*
1???	*(MIPS64)	*(MIPS64)	*(MIPS64)	*(MIPS64)	*(MIPS64)	*(MIPS64)	*(MIPS64)	*(MIPS64)

B.37 P.ROTX pool



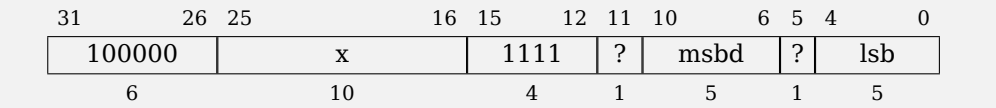
00	01	10	11
ROTX	*	*	*

B.38 P.INS pool



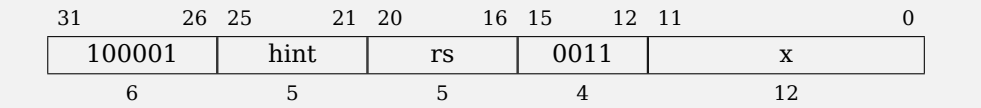
00	01	10	11
INS	*(MIPS64)	*(MIPS64)	*(MIPS64)

B.39 P.EXT pool



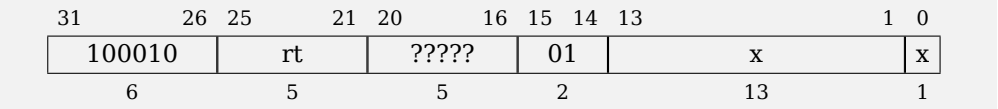
00	01	10	11
EXT	*(MIPS64)	*(MIPS64)	*(MIPS64)

B.40 P.PREF[U12] pool



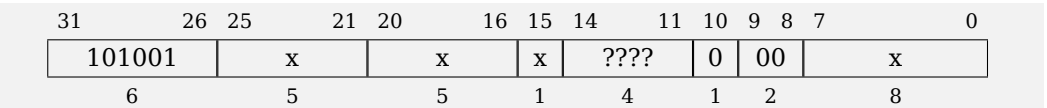
- SYNCI[U12]: hint==31
- PREF[U12]: hint!=31

B.41 P.BR3A pool



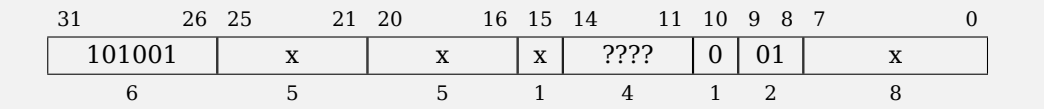
	???00	???01	???10	???11
000??	*(CP1)	*(CP1)	*(CP2)	*(CP2)
001??	*(DSP)	*	*	*
010??	*	*	*	*
011??	*	*	*	*
100??	*	*	*	*
101??	*	*	*	*
110??	*	*	*	*
111??	*	*	*	*

B.42 P.LS.S0 pool



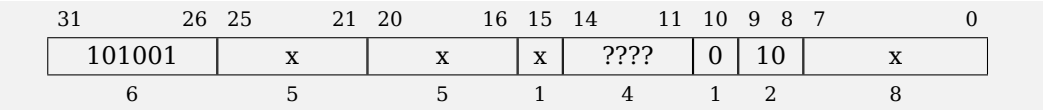
	??00	??01	??10	??11
00??	LB[S9]	SB[S9]	LBU[S9]	P.PREF[S9]
01??	LH[S9]	SH[S9]	LHU[S9]	*(MIPS64)
10??	LW[S9]	SW[S9]	*(CP1)	*(CP1)
11??	*(MIPS64)	*(MIPS64)	*(CP1)	*(CP1)

B.43 P.LS.S1 pool



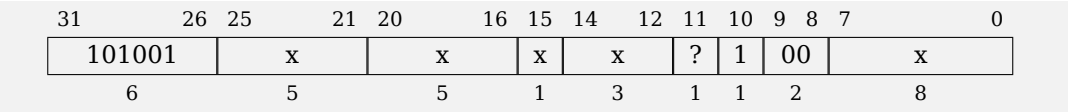
	??00	??01	??10	??11
00??	*	*	*(MCU)	*
01??	UALH	UASH	*	CACHE
10??	*(CP2)	*(CP2)	P.LL	P.SC
11??	*(CP2)	*(CP2)	*(MIPS64)	*(MIPS64)

B.44 P.LS.E0 pool



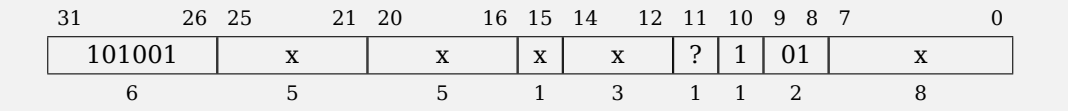
	??00	??01	??10	??11
00??	LBE	SBE	LBUE	P.PREFE
01??	LHE	SHE	LHUE	CACHEE
10??	LWE	SWE	P.LLE	P.SCE
11??	*	*	*	*

B.45 P.LS.WM pool



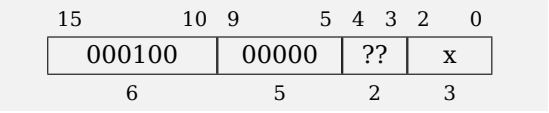
0	1
LWM	SWM

B.46 P.LS.UAWM pool



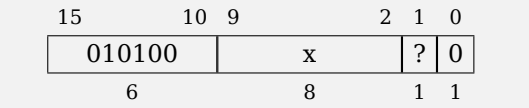
0	1
UALWM	UASWM

B.47 P16.RI pool



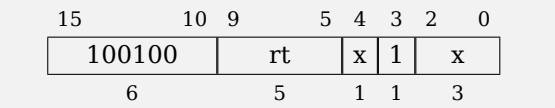
00	01	10	11
*	P16.SYSCALL	BREAK[16]	SDBBP[16]

B.48 POOL16C_0 pool



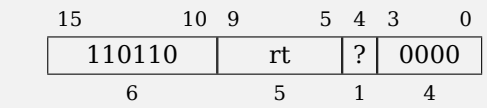
0	1
POOL16C_00	*

B.49 P.ADDIU[RS5] pool



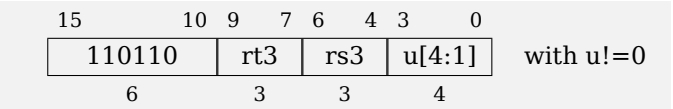
- NOP[16]: rt==0
- ADDIU[RS5]: rt!=0

B.50 P16.JRC pool



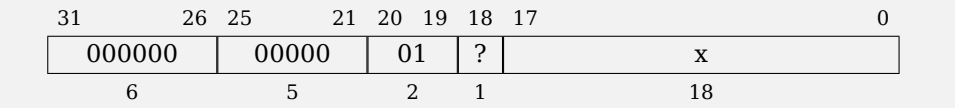
0	1
JRC	JALRC[16]

B.51 P16.BR1 pool



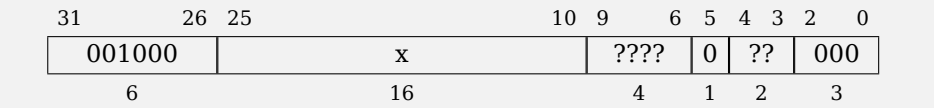
- BEQC[16]: rs3<rt3
- BNEC[16]: rs3>=rt3

B.52 P.SYSCALL pool



0	1
SYSCALL[32]	*(VZ)

B.53 _POOL32A0_0 pool

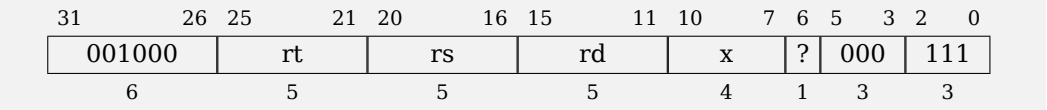


	????00	????01	????10	????11
0000??	P.TRAP	SEB	SLLV	MUL[32]
0001??	*	SEH	SRLV	MUH
0010??	*	*	SRAV	MULU
0011??	*	*	ROTRV	MUHU
0100??	*	*	ADD	DIV
0101??	*	*	ADDU[32]	MOD
0110??	*	*	SUB	DIVU
0111??	RDHWR	*	SUBU[32]	MODU
1000??	*	*	P.CMOVE	*
1001??	*	*	AND[32]	*
1010??	*	*	OR[32]	*
1011??	*	*	NOR	*
1100??	*	*	XOR[32]	*
1101??	*	*	SLT	*
1110??	*	*	P.SLTU	*
1111??	*	*	SOV	*

B.54 _POOL32A0_1 pool

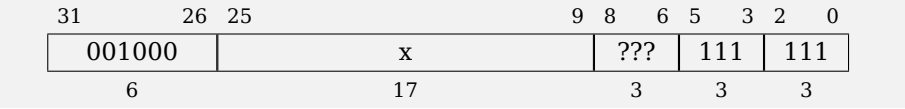
	????00	????01	????10	????11
0000??	*	*	MFC0	MFHC0
0001??	*	*	MTC0	MTHC0
0010??	*	*	*(VZ)	*(VZ)
0011??	*	*	*(VZ)	*(VZ)
0100??	*	*	*(MIPS64)	*
0101??	*	*	*(MIPS64)	*
0110??	*	*	*(VZ)	*
0111??	*	*	*(VZ)	*
1000??	*	*(MT)	*(MT)	MFTR
1001??	*	*(MT)	*(MT)	MTTR
1010??	*	*	*(MT)	*
1011??	*	*	*	*
1100??	*	*	*	*
1101??	*	*	*	*
1110??	*	*	*	*
1111??	*	CRC32	*	*

B.55 P.LSX pool



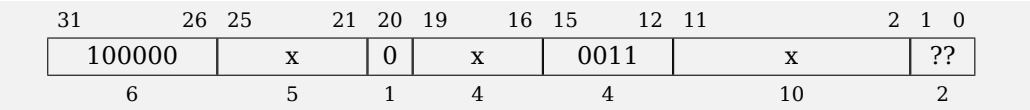
0	1
PP.LSX	PP.LSXS

B.56 POOL32Axf pool



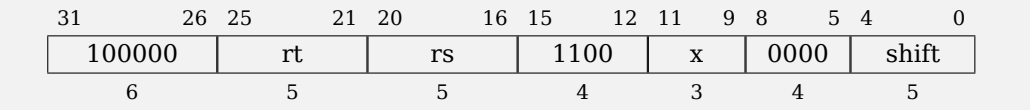
000	*
001	*(DSP)
010	*(DSP)
011	*
100	POOL32Axf_4
101	POOL32Axf_5
110	*
111	*(DSP)

B.57 PP.SR pool



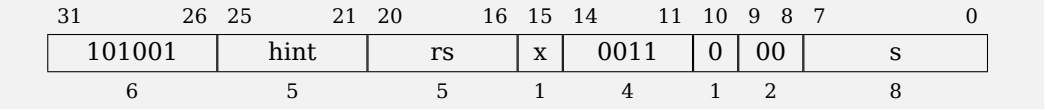
00	01	10	11
SAVE[32]	*	RESTORE[32]	RESTORE.JRC[32]

B.58 P.SLL pool



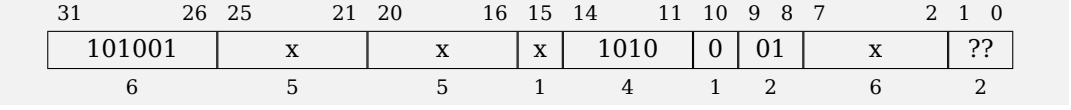
- NOP[32]: rt==0 && shift==0
- EHB: rt==0 && shift==3
- PAUSE: rt==0 && shift==5
- SYNC: rt==0 && shift==6
- SLL[32]: exclusions

B.59 P.PREF[S9] pool



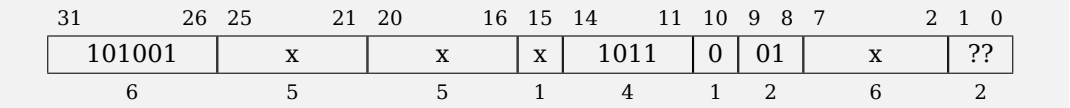
- SYNCI[S9]: hint==31
- PREF[S9]: hint!=31

B.60 P.LL pool



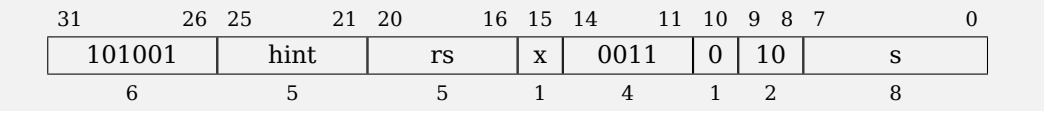
00	01	10	11
LL	LLWP	*	*

B.61 P.SC pool



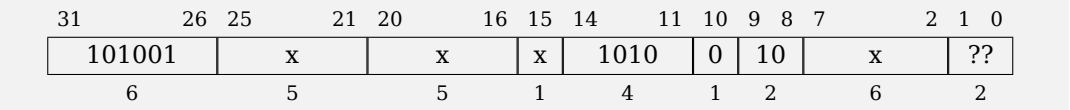
00	01	10	11
SC	SCWP	*	*

B.62 P.PREFE pool



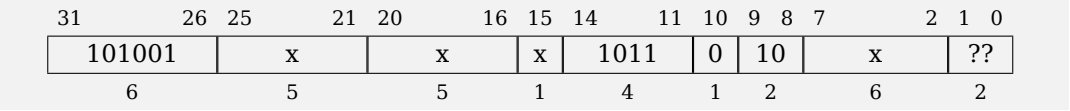
- SYNCIE: hint==31
- PREFE: hint!=31

B.63 P.LLE pool



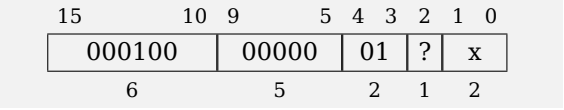
00	01	10	11
LLE	LLWPE	*	*

B.64 P.SCE pool



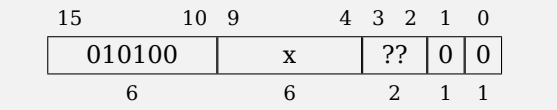
00	01	10	11
SCE	SCWPE	*	*

B.65 P16.SYSCALL pool



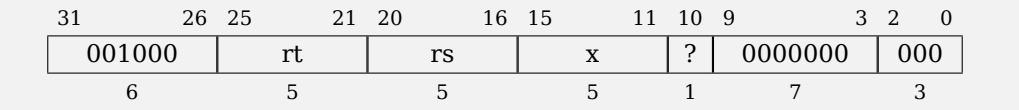
0	1
SYSCALL[16]	*(VZ)

B.66 POOL16C_00 pool



00	01	10	11
NOT[16]	XOR[16]	AND[16]	OR[16]

B.67 P.TRAP pool

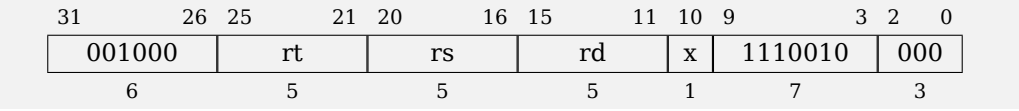


0	1
TEQ	TNE

B.68 P.CMOVE pool

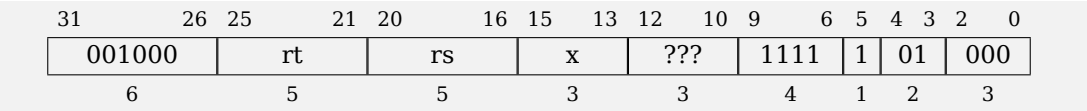
0	1
MOVZ	MOVN

B.69 P.SLTU pool



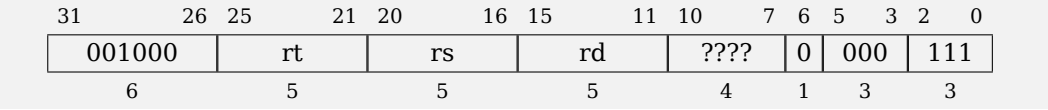
- P.DVP: rd==0
- SLTU: rd!=0

B.70 CRC32 pool



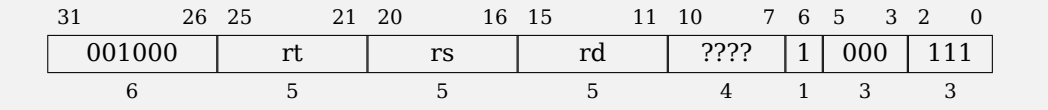
000	001	010	011	100	101	110	111
CRC32B	CRC32H	CRC32W	*(MIPS64)	CRC32CB	CRC32CH	CRC32CW	*(MIPS64)

B.71 PP.LSX pool



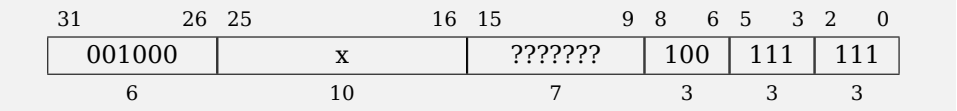
	??00	??01	??10	??11
00??	LBX	SBX	LBUX	*
01??	LHX	SHX	LHUX	*(MIPS64)
10??	LWX	SWX	*(CP1)	*(CP1)
11??	*(MIPS64)	*(MIPS64)	*(CP1)	*(CP1)

B.72 PP.LSXS pool



	??00	??01	??10	??11
00??	*	*	*	*
01??	LHXS	SHXS	LHUXS	*(MIPS64)
10??	LWXS[32]	SWXS	*(CP1)	*(CP1)
11??	*(MIPS64)	*(MIPS64)	*(CP1)	*(CP1)

B.73 POOL32Axf_4 pool



	????000	????001	????010	????011	????100	????101	????110	????111
0000???	*(DSP)	*(DSP)	*	*	*	*	*	*
0001???	*(DSP)	*(DSP)	*	*	*	*	*	*
0010???	*(DSP)	*	*	*	*	*	*	*
0011???	*(DSP)	*	*	*	*	*	*	*
0100???	*(DSP)	*	*	*	*	CLO	*(CP2)	*
0101???	*(DSP)	*	*	*	*	CLZ	*(CP2)	*
0110???	*(DSP)	*	*	*	*	*	*(CP2)	*
0111???	*(DSP)	*(DSP)	*	*	*	*	*(CP2)	*
1000???	*	*	*	*	*	*	*(CP2)	*
1001???	*(DSP)	*(DSP)	*	*	*	*	*(CP2)	*
1010???	*	*	*	*	*	*	*	*
1011???	*(DSP)	*(DSP)	*	*	*	*	*	*
1100???	*	*	*	*	*	*	*(CP2)	*
1101???	*(DSP)	*(DSP)	*	*	*	*	*(CP2)	*
1110???	*	*	*	*	*	*	*	*
1111???	*(DSP)	*	*	*	*	*	*	*

B.74 POOL32Axf_5 pool

31–26	25–16	15–14	13–9	8–6	5–3	2–0
001000	x	??	x	101	111	111
6	10	2	5	3	3	3

00	POOL32Axf_5_group0
01	POOL32Axf_5_group1
10	*
11	POOL32Axf_5_group3

B.75 P.DVP pool

31–26	25–21	20–16	15–11	10	9–3	2–0
001000	rt	rs	00000	?	1110010	000
6	5	5	5	1	7	3

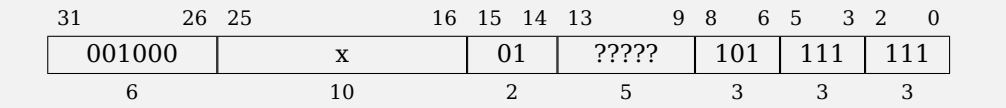
0	1
DVP	EVP

B.76 POOL32Axf_5_group0 pool

31–26	25–16	15–14	13–9	8–6	5–3	2–0
001000	x	00	?????	101	111	111
6	10	2	5	3	3	3

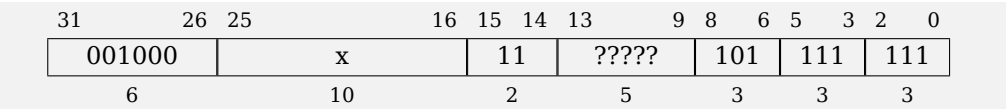
	??000	??001	??010	??011	??100	??101	??110	??111
00???	*(VZ)	TLBP	*(VZ)	TLBINV	*	*	*(VZ)	GINVT
01???	*(VZ)	TLBR	*(VZ)	TLBINVF	*	*	*	GINVI
10???	*(VZ)	TLBWI	*	*	*	*	*	*
11???	*(VZ)	TLBWR	*	*	*	*	*	*

B.77 POOL32Axf_5_group1 pool



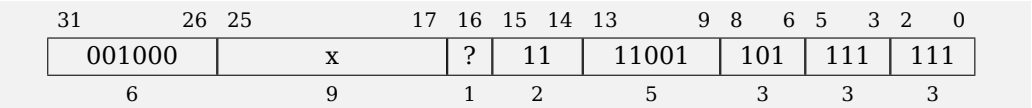
	??000	??001	??010	??011	??100	??101	??110	??111
00???	*	*	*	DI	*	*	*	*
01???	*	*	*	EI	*	*	*	*
10???	*	*	*	*	*	*	*	*
11???	*	*	*	*	*	*	*	*

B.78 POOL32Axf_5_group3 pool



	??000	??001	??010	??011	??100	??101	??110	??111
00???	*	WAIT	*	*	*	*	*	*
01???	*	*(MCU)	*	*	*	*	*	*
10???	RDPGPR	DERET	*	*	*	*	*	*
11???	WRPGPR	ERETx	*	*	*	*	*	*

B.79 ERETx pool



0	1
ERET	ERETNC

Appendix C

Change Log

- 6/27/17: Revision 0.9.7. ”Alpha” release.
- 7/28/17: Revision 0.9.8.
  - Added consistently named aliases for ROTX:
    * BITREVB (equivalent to legacy BITSWAP)
    * BITREVH (renamed from BITSWAP.H)
    * BITREVW (renamed from BITREV to avoid conflict with DSP instruction)
    * BYTEREVH (equivalent to legacy WSBH)
    * BYTEREVW (renamed from BYTEREV).
  - Added GINVI, GINVT instructions.
- 8/22/17: Revision 0.9.9.
  - Added CRC instructions: CRC32B, CRC32H, CRC32W, CRC32CB, CRC32CH CRC32CW.
  - Added EVA instructions: CACHEE, LBE, LBUE, LHE, LHUE, LLE, LLWPE, LWE, PREFE, SBE, SCE, SCWPE, SHE, SWE.
  - Expanded ”Purpose” sections to contain an expansion into words of the instruction mnemonic (in italics), plus as full an explanation of the operation of the instruction as is possible in one or two sentences.
  - Updated wording of ”Availability” sections to be more easily readable by humans.
  - Added more top level explanation in SAVE/RESTORE instructions description.
  - Aesthetic changes to SAVE, RESTORE, LWM, SWM, UALWM, UASWM pseudocode.
  - Reordered all instructions alphabetically.
- 11/22/17: Revision 0.9.10
  - Various minor consistency and bug fixes in the pseudocode and assembly formats, and clarifications in the instruction descriptions. No functional changes.
  - Explicit pseudocode added in several places which were previously marked as ”Operation equivalent to MIPS Release 6”.
  - Added sections explaining the use of ’@’ (concatenation) and ’[]’ (slicing) operators in the pseudocode.

- 1/30/18: Revision 00.20
  - Added ”Exceptions” section listing possible exception types for all instructions.
  - Added statement that LLWP and LLWPE are unpredictable if $rt and $ru are the same register.
  - Document revision number format changed to be consistent with company conventions.
- 3/21/18: Revision 00.21
  - Added full pseudocode and description for all instructions where the reader was previously referred to the base R6 specification. Updated definitions include TLB instructions, CACHE, CACHEE, CRC instructions, EHB, ERET, ERETNC, GINVI, GIVNT, JALRC.HB, LL, LLE, LLWP, LLWPE, MFC0, MFHC0, MTC0, MTHC0, PREF, PREFE, SC, SCE, SCWP, SCWPE, SYNC, SYNCI, SYNCIE.
  - Added full pseudocode for common functions where the reader was previously referred to the base R6 specification. Updated functions include those related to address translation, exceptions, segmentation control and EVA instructions.
  - Updated RDHWR ’sel’ field to be 5 bits wide (not 3).
  - Marked MTHC0/MFHC0 as required (except in NMS where they are optional).
  - Marked LLWP/SCWP as required (except in NMS where they are optional).
  - Fixed error in synci_step() function.
  - Various formatting changes:
    * Added ”Instruction Summary” chapter, and changed the chapter order so that the instruction definitions appear first and the supporting sections appear later (in some cases in the appendix).
    * Instruction definitions now always start on a new page.
    * Instruction definition sections are no longer numbered.
    * Individual instruction formats now appear in the table of contents.
    * Opcode diagrams appear on a pale gray background for contrast.
- 3/29/18: Revision 01.00
  - Added LWPC assembly alias.
  - Changed name of grouped instruction sections from e.g. ”CACHE[E]” to ”CACHE/CACHEE” etc. to avoid confusion with ”[]” qualifiers in format names.
- 4/27/18: Revision 01.01
  - Updated security class from ”Strictly Confidential” to ”Public”.