


EE Core Instruction Set Manual

Copyright © 2001 Sony Computer Entertainment Inc.
All Rights Reserved.

© 2001 Sony Computer Entertainment Inc.
Publication date: October 2001

Sony Computer Entertainment Inc.
1-1, Akasaka 7-chome, Minato-ku
Tokyo 107-0052 Japan

Unauthorized reproduction or distribution, in whole or in part, of this book is expressly prohibited by law.

 and PlayStation are registered trademarks of Sony Computer Entertainment Inc. All other trademarks are property of their respective owners.

About This Manual

The "EE Core Instruction Set Manual" is a comprehensive reference for the Emotion Engine instruction set.

- Chapter 1 "Notational Convention" describes the format used in this manual to describe the instructions.
- Chapter 2 "CPU Instruction Set" describes the 64-bit instructions that conform to the MIPS architecture, in alphabetical order.
- Chapter 3 "EE Core-Specific Instruction Set" describes the instructions that are extensions to the MIPS architecture, including 128-bit multimedia instructions, in alphabetical order.
- Chapter 4 "System Control Coprocessor (COP0) Instruction Set" describes the COP0 instructions, which perform address translation, cache control, exception handling and breakpoint functions, in alphabetical order.
- Chapter 5 "COP1 (FPU) Instruction Set" describes the COP1 instructions, which perform floating-point operations, in alphabetical order. Note: the EE Core also executes COP2 instructions which perform four-parallel floating-point operations (vector operations). For COP2 instructions, refer to the "VU User's Manual".
- Chapter 6 "Appendix" shows the instructions (including COP2 instructions) classified by functions.
- Chapter 7 "Appendix" shows the instructions organized by bit pattern of the instruction codes.

(This page is left blank intentionally)

Glossary

Term	Definition
EE	Emotion Engine. CPU of the PlayStation 2.
EE Core	Generalized computation and control unit of EE. Core of the CPU.
COP0	EE Core system control coprocessor.
COP1	EE Core floating-point operation coprocessor. Also referred to as FPU.
COP2	Vector operation unit coupled as a coprocessor of EE Core. VPU0.
GS	Graphics Synthesizer. Graphics processor connected to EE.
GIF	EE Interface unit to GS.
IOP	Processor connected to EE for controlling input/output devices.
SBUS	Bus connecting EE to IOP.
VPU (VPU0/VPU1)	Vector operation unit. EE contains 2 VPUs: VPU0 and VPU1.
VU (VU0/VU1)	VPU core operation unit.
VIF (VIF0/VIF1)	VPU data decompression unit.
VIFcode	Instruction code for VIF.
SPR	Quick-access data memory built into EE Core (Scratchpad memory).
IPU	EE Image processor unit.
word	Unit of data length: 32 bits
qword	Unit of data length: 128 bits
Slice	Physical unit of DMA transfer: 8 qwords or less
Packet	Data to be handled as a logical unit for transfer processing.
Transfer list	A group of packets transferred in serial DMA transfer processing.
Tag	Additional data indicating data size and other attributes of packets.
DMAtag	Tag positioned first in DMA packet to indicate address/size of data and address of the following packet.
GS primitive	Data to indicate image elements such as point and triangle.
Context	A set of drawing information (e.g. texture, distant fog color, and dither matrix) applied to two or more primitives uniformly. Also referred to as the drawing environment.
GIFtag	Additional data to indicate attributes of GS primitives.
Display list	A group of GS primitives to indicate batches of images.

(This page is left blank intentionally)

Contents

1. Notational Convention.....	17
1.1. Instruction Format of Each Instruction.....	18
1.1.1. Mnemonic	18
1.1.2. Instruction Encoding Picture	18
1.1.3. Format.....	19
1.1.4. Description Section	19
1.1.5. Restrictions Section	19
1.1.6. Exception Section.....	19
1.1.7. Operation Section	19
1.1.8. Programming Notes Section.....	19
1.2. Notational Convention of Pseudocode	20
1.2.1. Pseudocode Symbol.....	20
1.2.2. Pseudocode Functions.....	21
2. CPU Instruction Set.....	23
ADD : Add Word	24
ADDI : Add Immediate Word	25
ADDIU : Add Immediate Unsigned Word.....	26
ADDU : Add Unsigned Word.....	27
AND : And.....	28
ANDI : Add Immediate.....	29
BEQ : Branch on Equal.....	30
BEQL : Branch on Equal Likely	31
BGEZ : Branch on Greater Than or Equal to Zero	32
BGEZAL : Branch on Greater Than or Equal to Zero and Link	33
BGEZALL : Branch on Greater Than or Equal to Zero and Link Likely.....	34
BGEZL : Branch on Greater Than or Equal to Zero Likely.....	35
BGTZ : Branch on Greater Than Zero	36
BGTZL : Branch on Greater Than Zero Likely	37
BLEZ : Branch on Less Than or Equal to Zero	38
BLEZL : Branch on Less Than or Equal to Zero Likely	39
BLTZ : Branch on Less Than Zero.....	40
BLTZAL : Branch on Less Than Zero and Link.....	41
BLTZALL : Branch on Less Than Zero and Link Likely	42
BLTZL : Branch on Less Than Zero Likely	43
BNE : Branch on Not Equal	44
BNEL : Branch on Not Equal Likely.....	45
BREAK : Breakpoint	46
DADD : Doubleword Add	47
DADDI : Doubleword Add Immediate.....	48
DADDIU : Doubleword Add Immediate Unsigned	49

DADDU : Doubleword Add Unsigned	50
DIV : Divide Word	51
DIVU : Divide Unsigned Word.....	53
DSLL : Doubleword Shift Left Logical.....	54
DSLL32 : Doubleword Shift Left Logical Plus 32	55
DSLLV : Doubleword Shift Left Logical Variable.....	56
DSRA : Doubleword Shift Right Arithmetic.....	57
DSRA32 : Doubleword Shift Right Arithmetic Plus 32	58
DSRAV : Doubleword Shift Right Arithmetic Variable.....	59
DSRL : Doubleword Shift Right Logical	60
DSRL32 : Doubleword Shift Right Logical Plus 32.....	61
DSRLV : Doubleword Shift Right Logical Variable	62
DSUB : Doubleword Subtract.....	63
DSUBU : Doubleword Subtract Unsigned.....	64
J : Jump.....	65
JAL : Jump and Link	66
JALR : Jump and Link Register.....	67
JR : Jump Register.....	68
LB : Load Byte	69
LBU : Load Byte Unsigned.....	70
LD : Load Doubleword.....	71
LDL : Load Doubleword Left.....	72
LDR : Load Doubleword Right.....	74
LH : Load Halfword.....	76
LHU : Load Halfword Unsigned	77
LUI : Load Upper Immediate.....	78
LW : Load Word.....	79
LWL : Load Word Left.....	80
LWR : Load Word Right	82
LWU : Load Word Unsigned	84
MFHI : Move from HI Register	85
MFLO : Move from LO Register.....	86
MOVN : Move Conditional on Not Zero	87
MOVZ : Move Conditional on Zero	88
MTHI : Move to HI Register	89
MTLO : Move to LO Register	90
MULT : Multiply Word	91
MULTU : Multiply Unsigned Word.....	92
NOR : Not Or.....	93
OR : Or.....	94
ORI : Or immediate	95
PREF : Prefetch.....	96
SB : Store Byte.....	97
SD : Store Doubleword	98
SDL : Store Doubleword Left.....	99
SDR : Store Doubleword Right	101

SH : Store Halfword	103
SLL : Shift Word Left Logical	104
SLLV : Shift Word Left Logical Variable	105
SLT : Set on Less Than	106
SLTI : Set on Less Than Immediate	107
SLTIU : Set on Less Than Immediate Unsigned	108
SLTU : Set on Less Than Unsigned	109
SRA : Shift Word Right Arithmetic	110
SRAV : Shift Word Right Arithmetic Variable	111
SRL : Shift Word Right Logical	112
SRLV : Shift Word Right Logical Variable	113
SUB : Subtract Word	114
SUBU : Subtract Unsigned Word	115
SW : Store Word	116
SWL : Store Word Left	117
SWR : Store Word Right	119
SYNC.type : Synchronize Shared Memory	121
SYSCALL : System Call	122
TEQ : Trap if Equal	123
TEQI : Trap if Equal Immediate	124
TGE : Trap if Greater or Equal	125
TGEI : Trap if Greater or Equal Immediate	126
TGEIU : Trap if Greater or Equal Immediate Unsigned	127
TGEU : Trap if Greater or Equal Unsigned	128
TLT : Trap if Less Than	129
TLTI : Trap if Less Than Immediate	130
TLTIU : Trap if Less Than Immediate Unsigned	131
TLTU : Trap if Less Than Unsigned	132
TNE : Trap if Not Equal	133
TNEI : Trap if Not Equal Immediate	134
XOR : Exclusive OR	135
XORI : Exclusive OR Immediate	136
3. EE Core-Specific Instruction Set	137
DIV1 : Divide Word Pipeline 1	138
DIVU1 : Divide Unsigned Word Pipeline 1	140
LQ : Load Quadword	141
MADD : Multiply-Add word	142
MADD1 : Multiply-Add word Pipeline 1	143
MADDU : Multiply-Add Unsigned word	144
MADDU1 : Multiply-Add Unsigned word Pipeline 1	145
MFHI1 : Move From HI1 Register	146
MFLO1 : Move From LO1 Register	147
MFSA : Move from Shift Amount Register	148
MTHI1 : Move To HI1 Register	149
MTLO1 : Move To LO1 Register	150
MTSA : Move to Shift Amount Register	151

MTSAB : Move Byte Count to Shift Amount Register	152
MTSAH : Move Halfword Count to Shift Amount Register	153
MULT : Multiply Word	154
MULT1 : Multiply Word Pipeline 1	155
MULTU : Multiply Unsigned Word	156
MULTU1 : Multiply Unsigned Word Pipeline 1	157
PABSH : Parallel Absolute Halfword	158
PABSW : Parallel Absolute Word	159
PADDB : Parallel Add Byte	160
PADDH : Parallel Add Halfword	161
PADDSB : Parallel Add with Signed saturation Byte	162
PADDSH : Parallel Add with Signed saturation Halfword	164
PADDSW : Parallel Add with Signed saturation Word	166
PADDUB : Parallel Add with Unsigned saturation Byte	168
PADDUH : Parallel Add with Unsigned saturation Halfword	170
PADDUW : Parallel Add with Unsigned saturation Word	172
PADDW : Parallel Add Word	174
PADSBH : Parallel Add/Subtract Halfword	175
PAND : Parallel And	176
PCEQB : Parallel Compare for Equal Byte	177
PCEQH : Parallel Compare for Equal Halfword	179
PCEQW : Parallel Compare for Equal Word	181
PCGTB : Parallel Compare for Greater Than Byte	183
PCGTH : Parallel Compare for Greater Than Halfword	185
PCGTW : Parallel Compare for Greater Than Word	187
PCPYH : Parallel Copy Halfword	189
PCPYLD : Parallel Copy Lower Doubleword	190
PCPYUD : Parallel Copy Upper Doubleword	191
PDIVBW : Parallel Divide Broadcast Word	192
PDIVUW : Parallel Divide Unsigned Word	194
PDIVW : Parallel Divide Word	196
PEXCH : Parallel Exchange Center Halfword	198
PEXCW : Parallel Exchange Center Word	199
PEXEH : Parallel Exchange Even Halfword	200
PEXEW : Parallel Exchange Even Word	201
PEXT5 : Parallel Extend from 5 bits	202
PEXTLB : Parallel Extend Lower from Byte	203
PEXTLH : Parallel Extend Lower from Halfword	204
PEXTLW : Parallel Extend Lower from Word	205
PEXTUB : Parallel Extend Upper from Byte	206
PEXTUH : Parallel Extend Upper from Halfword	207
PEXTUW : Parallel Extend Upper from Word	208
PHMADH : Parallel Horizontal Multiply-Add Halfword	209
PHMSBH : Parallel Horizontal Multiply-Subtract Halfword	211
PINTEH : Parallel Interleave Even Halfword	213
PINTH : Parallel Interleave Halfword	214

PLZCW : Parallel Leading Zero or one Count Word.....	215
PMADDH : Parallel Multiply-Add Halfword.....	216
PMADDUW : Parallel Multiply-Add Unsigned Word.....	218
PMADDW : Parallel Multiply-Add Word.....	220
PMAXH : Parallel Maximize Halfword.....	222
PMAXW : Parallel Maximize Word.....	223
PMFHI : Parallel Move From HI Register.....	225
PMFHL.LH : Parallel Move From HI/LO Register.....	226
PMFHL.LW : Parallel Move From HI/LO Register.....	227
PMFHL.SH : Parallel Move From HI/LO Register.....	228
PMFHL.SLW : Parallel Move From HI/LO Register.....	230
PMFHL.UW : Parallel Move From HI/LO Register.....	232
PMFLO : Parallel Move From LO Register.....	233
PMINH : Parallel Minimize Halfword.....	234
PMINW : Parallel Minimize Word.....	235
PMSUBH : Parallel Multiply-Subtract Halfword.....	237
PMSUBW : Parallel Multiply-Subtract Word.....	239
PMTHI : Parallel Move To HI Register.....	241
PMTHL.LW : Parallel Move To HI/LO Register.....	242
PMTLO : Parallel Move To LO Register.....	243
PMULTH : Parallel Multiply Halfword.....	244
PMULTUW : Parallel Multiply Unsigned Word.....	246
PMULTW : Parallel Multiply Word.....	248
PNOR : Parallel Not Or.....	250
POR : Parallel Or.....	251
PPAC5 : Parallel Pack to 5 bits.....	252
PPACB : Parallel Pack to Byte.....	254
PPACH : Parallel Pack to Halfword.....	255
PPACW : Parallel Pack to Word.....	256
PREVH : Parallel Reverse Halfword.....	257
PROT3W : Parallel Rotate 3 Words Left.....	258
PSLLH : Parallel Shift Left Logical Halfword.....	259
PSLLVW : Parallel Shift Left Logical Variable Word.....	260
PSLLW : Parallel Shift Left Logical Word.....	261
PSRAH : Parallel Shift Right Arithmetic Halfword.....	262
PSRAVW : Parallel Shift Right Arithmetic Variable Word.....	263
PSRAW : Parallel Shift Right Arithmetic Word.....	264
PSRLH : Parallel Shift Right Logical Halfword.....	265
PSRLVW : Parallel Shift Right Logical Variable Word.....	266
PSRLW : Parallel Shift Right Logical Word.....	267
PSUBB : Parallel Subtract Byte.....	268
PSUBH : Parallel Subtract Halfword.....	269
PSUBSB : Parallel Subtract with Signed saturation Byte.....	270
PSUBSH : Parallel Subtract with Signed Saturation Halfword.....	272
PSUBSW : Parallel Subtract with Signed Saturation Word.....	274
PSUBUB : Parallel Subtract with Unsigned Saturation Byte.....	276

PSUBUH : Parallel Subtract with Unsigned Saturation Halfword	278
PSUBUW : Parallel Subtract with Unsigned Saturation Word.....	280
PSUBW : Parallel Subtract Word.....	282
PXOR : Parallel Exclusive OR.....	283
QFSRV : Quadword Funnel Shift Right Variable	284
SQ : Store Quadword.....	285
4. System Control Coprocessor (COP0) Instruction Set.....	287
BC0F : Branch on Coprocessor 0 False	288
BC0FL : Branch on Coprocessor 0 False Likely.....	289
BC0T : Branch on Coprocessor 0 True.....	290
BC0TL : Branch on Coprocessor 0 True Likely	291
CACHE BFH : Cache Operation (BTAC Flush)	292
CACHE BHINBT : Cache Operation (Hit Invalidate BTAC)	293
CACHE BXLBT : Cache Operation (Index Load BTAC)	294
CACHE BXSBT : Cache Operation (Index Store BTAC)	295
CACHE DHIN : Cache Operation (Hit Invalidate)	296
CACHE DHWBIN : Cache Operation (Hit Writeback Invalidate).....	297
CACHE DHWOIN : Cache Operation (Hit Writeback Without Invalidate).....	298
CACHE DXIN : Cache Operation (Index Invalidate).....	299
CACHE DXLDT : Cache Operation (Index Load Data).....	300
CACHE DXLTG : Cache Operation (Index Load Tag).....	301
CACHE DXSDT : Cache Operation (Index Store Data).....	302
CACHE DXSTG : Cache Operation (Index Store Tag).....	303
CACHE DXWBIN : Cache Operation (Index Writeback Invalidate)	304
CACHE IFL : Cache Operation (Fill)	305
CACHE IHIN : Cache Operation (Hit Invalidate).....	306
CACHE IXIN : Cache Operation (Index Invalidate).....	307
CACHE IXLDT : Cache Operation (Index Load Data).....	308
CACHE IXLTG : Cache Operation (Index Load Tag)	309
CACHE IXSDT : Cache Operation (Index Store Data).....	310
CACHE IXSTG : Cache Operation (Index Store Tag).....	311
DI : Disable Interrupt.....	312
EI : Enable Interrupt.....	313
ERET : Exception Return.....	314
MFBPC : Move from Breakpoint Control Register	315
MFC0 : Move from System Control Coprocessor	316
MFDAB : Move from Data Address Breakpoint Register.....	317
MFDABM : Move from Data Address Breakpoint Mask Register.....	318
MFDVB : Move from Data value Breakpoint Register.....	319
MFDVBM : Move from Data Value Breakpoint Mask Register	320
MFIAB : Move from Instruction Address Breakpoint Register	321
MFIABM : Move from Instruction Address Breakpoint Mask Register.....	322
MFPC : Move from Performance Counter.....	323
MFPS : Move from Performance Event Specifier.....	324
MTBPC : Move to Breakpoint Control Register	325
MTC0 : Move to System Control Coprocessor	326

MTDAB : Move to Data Address Breakpoint Register.....	327
MTDABM : Move to Data Address Breakpoint Mask Register.....	328
MTDVB : Move to Data Value Breakpoint Register.....	329
MTDVBM : Move to Data Value Breakpoint Mask Register.....	330
MTIAB : Move to Instruction Address Breakpoint Register.....	331
MTIABM : Move to Instruction Address Breakpoint Mask Register.....	332
MTPC : Move to Performance Counter.....	333
MTPS : Move to Performance Event Specifier.....	334
TLBP : Probe TLB for Matching Entry	335
TLBR : Read Indexed TLB Entry.....	336
TLBWI : Write Index TLB Entry	337
TLBWR : Write Random TLB Entry.....	338
5. COP1 (FPU) Instruction Set.....	339
ABS.S : Floating Point Absolute Value.....	340
ADD.S : Floating Point ADD	341
ADDAS : Floating Point Add to Accumulator	342
BC1F : Branch on FP False.....	343
BC1FL : Branch on FP False Likely.....	344
BC1T : Branch on FP True.....	345
BC1TL : Branch on FP True Likely.....	346
C.EQ.S : Floating Point Compare	347
C.F.S : Floating Point Compare	348
C.LE.S : Floating Point Compare	349
C.LT.S : Floating Point Compare.....	350
CFC1 : Move Control Word from Floating Point	351
CTC1 : Move Control Word to Floating Point.....	352
CVT.S.W : Fixed-point Convert to Single Floating Point.....	353
CVT.W.S : Floating Point Convert to Word Fixed-point	354
DIV.S : Floating Point Divide	355
LWC1 : Load Word to Floating Point	356
MADD.S : Floating Point Multiply-ADD.....	357
MADDAS : Floating Point Multiply-Add.....	359
MAX.S : Floating Point Maximum.....	361
MFC1 : Move Word from Floating Point	362
MIN.S : Floating Point Minimum.....	363
MOV.S : Floating Point Move.....	364
MSUB.S : Floating Point Multiply and Subtract.....	365
MSUBAS : Floating Point Multiply and Subtract from Accumulator.....	367
MTC1 : Move Word to Floating Point	369
MUL.S : Floating Point Multiply.....	370
MULAS : Floating Point Multiply to Accumulator	371
NEG.S : Floating Point Negate.....	372
RSQRT.S : Floating Point Reciprocal Square Root.....	373
SQRT.S : Floating Point Square Root.....	374
SUB.S : Floating Point Subtract	375
SUBAS : Floating Point Subtract to Accumulator.....	376

SWC1 : Store Word from Floating Point	377
6. Appendix Instruction Set List	379
6.1. Computational Instructions	380
6.1.1. Integer Addition and Subtraction	380
6.1.2. Floating Point Addition and Subtraction	381
6.1.3. Integer Multiplication and Division	381
6.1.4. Floating Point Multiplication and Division	381
6.1.5. Integer Multiply-Add	382
6.1.6. Floating Point Multiply-Add	382
6.1.7. Shift Operation	383
6.1.8. Logical Operation	383
6.1.9. Comparison Operation	384
6.1.10. Maximum / Minimum Value	384
6.1.11. Data Format Conversion	384
6.1.12. Exchange	385
6.1.13. Random Number	385
6.1.14. Other Operations	385
6.2. Data Transfer Instructions	386
6.2.1. Instructions for Transferring between Registers	386
6.2.2. Load	387
6.2.3. Store	387
6.2.4. Special Data Transfer	388
6.3. Program Control Instructions	389
6.3.1. Conditional Branch	389
6.3.2. Jump	389
6.3.3. Subroutine Call	389
6.3.4. Break / Trap	390
6.4. Other Instructions	391
7. Appendix OpCode Encoding	393
7.1. CPU Instructions	394
7.1.1. Instructions encoded by OpCode field	394
7.1.2. SPECIAL Instruction Class	395
7.1.3. REGIMM Instruction Class	396
7.2. EE Core-Specific Instructions	397
7.2.1. MMI Instruction Class	397
7.2.2. MMI0 Instruction Class	398
7.2.3. MMI1 Instruction Class	399
7.2.4. MMI2 Instruction Class	400
7.2.5. MMI3 Instruction Class	401
7.3. COP0 Instructions	402
7.3.1. COP0 Instruction Class	402
7.3.2. BC0 Instruction Class	402
7.3.3. C0 Instruction Class	403
7.4. COP1 Instructions	404
7.4.1. COP1 Instruction Class	404
7.4.2. BC1 Instruction Class	404

7.4.3. S Instruction Class	405
7.4.4. W Instruction Class	406

(This page is left blank intentionally)

1. Notational Convention

1.1.3. Format

This section indicates the instruction formats for the assembler. Lower case indicates variables, corresponding to variable fields in the encoding picture.

1.1.4. Description Section

This section describes the instruction function and operation. If possible, the outline of the operation is expressed in one-line pseudocode. The notational conventions of pseudocode are described later.

1.1.5. Restrictions Section

This section shows the restrictions on instruction execution. These include alignment for memory addresses, the range of valid values of operands, order of execution with other instructions, and so on.

1.1.6. Exception Section

This section shows the exceptions that can be caused by the instructions. However, it omits exceptions caused by instruction fetch, performance counters, breakpoints, asynchronous external events (e.g. interrupt) and Bus Error.

1.1.7. Operation Section

This section describes the instruction operations in pseudocode, resembling Pascal. The notational conventions of pseudocode are described later.

1.1.8. Programming Notes Section

This section shows the supplementary information about programming when using the instruction.

1.2. Notational Convention of Pseudocode

The "Description" and "Operation" sections describe the operations that each instruction performs using a pseudocode resembling Pascal. The notational conventions of the pseudocode are as follows.

1.2.1. Pseudocode Symbol

Special symbols used in the pseudocode notation are as follows.

Symbol	Meaning
\leftarrow	Assignment.
$= \neq$	Tests of equality and inequality.
$ $	Bit string concatenation.
X^y	Bit string formed by y copies of 1-bit value X.
$X_{y.z}$	Substring of bit-y through bit-z of bit string X.
$+ -$	Twos complement or floating point add and subtraction.
\times	Two's complement or floating point multiplication.
DIV	Two's complement integer division.
MOD	Two's complement modulo.
$/$	Floating point division.
$<$	Two's complement comparison operation (less than).
NOT	Bitwise logical NOT.
NOR	Bitwise logical NOR.
XOR	Bitwise logical XOR.
AND	Bitwise logical AND.
OR	Bitwise logical OR.
GPRLEN	The length in bits of the CPU general purpose registers.
$\text{GPR}[x]$	CPU general purpose register x.
HI0,LO0	Lower 64 bits of both HI and LO registers that are extended to 128 bits.
HI1,LO1	Upper 64 bits of both HI and LO registers that are extended to 128 bits.
HI,LO	128-bit HI and LO registers. If clear in the context, HI0 and LO0 registers may be described as simply HI and LO according to the existing MIPS term.
$\text{CPR}[z,x]$	General purpose register of Coprocessor unit z
$\text{CCR}[z,x]$	Control register x of Coprocessor unit z
$\text{CPCOND}[z]$	Conditional signal of Coprocessor unit z
$I, I+1:$	Label When the timing between the instruction operation and the operation prior to and subsequent to the instruction is required to be stipulated (e.g. the branch delay slot during the branch instruction), it is shown in the beginning of each line.
PC	The Program Counter Value. It is the address of the instruction word and automatically incremented by 4 every time an instruction is executed. When any values are assigned to a pseudocode, the instruction in the address is performed following the instruction in the branch delay slot.
PSIZE	Bit length of Physical address (=32)

1.2.2. Pseudocode Functions

The main functions used in the pseudocode are described below.

(pAddr, CCA) ← AddressTranslation (vAddr, IorD, LorS) Address Translation

pAddr: Physical Address
 CCA: Cache Coherence Algorithm
 vAddr: Virtual Address
 IorD: Instruction access or Data access
 LorS: Load access or Store access

Translates a virtual address to a physical address and cache coherence algorithm (that determines which cache line is used). If the virtual address is in the unmapped address space, the physical address and CCA are determined directly by the virtual address. If the virtual address is in the mapped address space, the TLB is used to determine the physical address and CCA. An exception is taken if a page fault or a breach of access right occurs.

MemElem ← LoadMemory(CCA, AccessLength, pAddr, vAddr, IorD) Loads Data

MemElem: Data that is aligned with 128-bit width
 CCA: Cache Coherence Algorithm
 AccessLength: Data Size (in bytes)
 pAddr: Physical Address
 vAddr: Virtual Address
 IorD: Instruction access or Data access

Loads a value from memory.

Uses the address of the minimum value of the byte addresses in the data object. The low-order 2 to 4 bits, together with AccessLength, indicate which bytes within MemElem are given to the processor. If the access is an uncached type, only applicable bytes are read from memory and valid within MemElem. If the access is a cached type and the data is not present in the cache, data of the cache line size is read from memory.

StoreMemory (CCA, AccessLength, MemElem, pAddr, vAddr)

CCA: Cache Coherence Algorithm
 AccessLength: Data Size (in bytes)
 MemElem: 128-bit Data
 pAddr: Physical Address
 vAddr: Virtual Address

Store a value to memory.

MemElem is aligned to fixed-width data. If the store is partial, only applicable bytes are valid. The low-order three bits and AccessLength indicate the valid bytes.

SignalException (Exception)

Exception; The exception condition that exists.

Sends exception condition signals. An exception which suspends the instruction occurs and the pseudocode does not return from this function call.

UndefinedResult()

Indicates that the result of the operation is undefined.

NullifyCurrentInstruction ()

Nullifies the present instruction. The instructions in the delay slot of Branch-Likely instructions are nullified when not branching but is used for describing the operation.

(This page is left blank intentionally)

2. CPU Instruction Set

This chapter describes the instructions that can be performed in User mode, in alphabetical order. Instructions shown in this chapter conform to the MIPS architecture and, the general-purpose registers are 64-bit wide.

ADD : Add Word

MIPS I

To add 32-bit integers. Traps if overflow occurs.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000	rs	rt	rd	0 00000	ADD 100000						
6	5	5	5	5	6						

Format

ADD rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] + GPR[rt]$

Adds the 32-bit value in GPR[rt] to the 32-bit value in GPR[rs]. The result is stored in GPR[rd]. If the addition results in 32-bit 2's complement overflow, then the contents of GPR[rd] are not changed and an Integer Overflow exception occurs.

Restrictions

If GPR[rt] and GPR[rs] are not sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Exceptions

Integer Overflow

Operation

```

if (NotWordValue(GPR[rs]63..0) or NotWordValue(GPR[rt]63..0)) then UndefinedResult() endif
temp  $\leftarrow$  GPR[rs]63..0 + GPR[rt]63..0
if (32_bit_arithmetic_overflow) then
    SignalException (IntegerOverflow)
else
    GPR[rd]63..0  $\leftarrow$  sign_extend(temp31..0)
endif

```

Programming Notes

ADDU performs the same arithmetic operation but does not trap on overflow.

ADDI : Add Immediate Word

MIPS I

To add a constant to a 32-bit integer. Traps if overflow occurs.

Operation Code

31	26	25	21	20	16	15	0
SPECIAL 000000		rs		rt		immediate	
6		5		5		16	

Format

ADDI rt, rs, immediate

Description

$GPR[rt] \leftarrow GPR[rs] + \text{immediate}$

Adds the value of the immediate field as a 16-bit signed integer to the 32-bit integer value in GPR[rs]. The result is stored in GPR[rd]. If the addition results in 32-bit 2's complement overflow, then the contents of GPR[rt] are not changed and an Integer Overflow exception occurs.

Restrictions

If GPR[rs] is not a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is undefined.

Exceptions

Integer Overflow

Operation

```

if(NotWordValue(GPR[rs]_63..0)) then UndefinedResult() endif
temp ← GPR[rs]_63..0 + sign_extend(immediate)
if (32_bit_arithmetic_overflow) then
    SignalException(IntegerOverflow)
else
    GPR[rt]_63..0 ← sign_extend(temp_31..0)
endif

```

Programming Notes

ADDIU performs the same arithmetic operation but does not trap on overflow.

ADDIU : Add Immediate Unsigned Word

MIPS I

To add a constant to a 32-bit integer.

Operation Code

31	26	25	21	20	16	15	0
ADDIU						rs	
001001						rt	
						immediate	
6						5	
						5	
						16	

Format

ADDIU rt, rs, immediate

Description

$GPR[rt] \leftarrow GPR[rs] + \text{immediate}$

Adds the value of the immediate field as a 16-bit signed integer to the 32-bit integer value in GPR[rs]. The result is stored in GPR[rt]. If overflow occurs, it is ignored.

Restrictions

If GPR[rs] is not a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is undefined.

Exceptions

None

Operation

```
if (NotWordValue(GPR[rs]_63..0)) then UndefinedResult() endif
temp ← GPR[rs]_63..0 + sign_extend(immediate)
GPR[rt]_63..0 ← sign_extend(temp_31..0)
```

Programming Notes

This instruction is not an unsigned operation in the strict sense and performs 32-bit modulo arithmetic that ignores overflow. It is appropriate for integer arithmetic that ignores overflow such as address or C language arithmetic.

ADDU : Add Unsigned Word

MIPS I

To add 32-bit integers.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000	rs				rt				rd		0 00000
6	5				5				5		6

Format

ADDU rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] + GPR[rt]$

Adds the 32-bit value in GPR[rt] to the 32-bit value in GPR[rs]. The result is stored in GPR[rd]. If overflow occurs, it is ignored.

Restrictions

If GPR[rt] and GPR[rs] are not sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Exceptions

None

Operation

```
if (NotWordValue(GPR[rs] 63..0) or NotWordValue(GPR[rt] 63..0)) then UndefinedResult() endif
temp ← GPR[rs] 63..0 + GPR[rt] 63..0
GPR[rt] 63..0 ← sign_extend(temp 31..0)
```

Programming Notes

This instruction is not an unsigned operation in the strict sense and performs 32-bit modulo arithmetic that ignores overflow. It is appropriate for integer arithmetic that ignores overflow such as address or C language arithmetic.

AND : And

MIPS I

To perform a bitwise AND.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000	rs	rt	rd	0 00000	AND 100100						
6	5	5	5	5	6						

Format

AND rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] \text{ AND } GPR[rt]$

Performs a bitwise AND between GPR[rs] and GPR[rt]. The result is stored in GPR[rd].

The truth table value for AND is as follows;

X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

Exceptions

None

Operation

$GPR[rd]_{63..0} \leftarrow GPR[rs]_{63..0} \text{ AND } GPR[rt]_{63..0}$

ANDI : Add Immediate

MIPS I

To perform a bitwise AND.

Operation Code

31	26	25	21	20	16	15	0
ANDI 001100						rs	rt
						immediate	
6						5	5
						16	

Format

ANDI rt, rs, immediate

Description

$GPR[rt] \leftarrow GPR[rs] \text{ AND immediate}$

Performs a bitwise AND between the contents of GPR[rs] and the value of a zero- extended immediate value. The result is stored in GPR[rt].

Exceptions

None

Operation

$GPR[rt]_{63..0} \leftarrow (0^{48} \mid \mid \text{immediate}) \text{ AND } GPR[rs]_{63..0}$

BEQ : Branch on Equal

MIPS I

To compare two GPRs and do a PC-relative conditional branch according to the result.

Operation Code

31	26	25	21	20	16	15	0
BEQ 000100		rs		rt		offset	
6		5		5		16	

Format

BEQ rs, rt, offset

Description

if (GPR[rs] = GPR[rt]) then branch

Compares the contents of GPR[rs] and GPR[rt]. If they are equal, branches to the target address after the instruction in the branch delay slot is executed. If they are not equal, continues execution including the instruction in the branch delay slot.

The target address is the address obtained from adding an 18-bit signed offset, (the offset field shifted left 2 bits) to the address of the instruction in the branch delay slot (the instruction following the BEQ instruction).

Exceptions

None

Operation

```

I:      tgt_offset ← sign_extend(offset || 0²)
        condition ← (GPR[rs]63..0 = GPR[rt]63..0)
I+1:    if condition then
            PC ← PC + tgt_offset
        endif

```

Programming Notes

Since the offset is an 18-bit signed offset, the conditional branch range is ± 128 KB. Use J or JR instructions to branch to more distant addresses.

BEQL : Branch on Equal Likely

MIPS II

To compare two GPRs and do a PC-relative conditional branch according to the result. Executes the delay slot only if the branch is taken.

Operation Code

31	26	25	21	20	16	15	0
BEQL 010100		rs	rt	offset			
6		5	5	16			

Format

BEQL rs, rt, offset

Description

if (GPR[rs] = GPR[rt]) then branch_likely

Compares the contents of GPR[rs] and GPR[rt]. If they are equal, branches to the target address after the instruction in the branch delay slot is executed. If they are not equal, cancels the instruction in the branch delay slot.

The target address is the address obtained from adding an 18-bit signed offset (the offset field shifted left 2 bits), to the address of the instruction in the branch delay slot (the instruction following the BEQL instruction).

Exceptions

None

Operation

```

I:      tgt_offset ← sign_extend(offset || 02)
        condition ← (GPR[rs]63:0 = GPR[rt]63:0)
I+1:    if condition then
            PC ← PC + tgt_offset
        else
            NullifyCurrentInstruction ()
        endif

```

Programming Notes

Since the offset is an 18-bit signed offset, the conditional branch range is ± 128 KB. Use J or JR instructions to branch to more distant addresses.

BGEZ : Branch on Greater Than or Equal to Zero

MIPS I

To do a PC-relative branch according to the values of a GPR.

Operation Code

31	26	25	21	20	16	15	0
REGIMM 000001						rs	BGEZ 00001
offset							
6						5	5
						16	

Format

BGEZ rs, offset

Description

if (GPR[rs] >= 0) then branch

If the value of GPR[rs] is greater than or equal to zero (sign bit is 0), branches to the target address after the instruction in the delay slot is executed. If the value of GPR[rs] is less than zero, continues execution, including the instruction in the branch delay slot.

The target address is the address obtained from adding an 18-bit signed offset (the offset field shifted left 2 bits) to the address of the instruction in the branch delay slot (the instruction following the BGEZ instruction).

Exceptions

None

Operation

```

I:      tgt_offset ← sign_extend(offset || 02)
        condition ← GPR[rs]63:0 >= 0
I+1:    if condition then
        PC ← PC + tgt_offset
        endif

```

Programming Notes

Since the offset is an 18-bit signed offset, the conditional branch range is ± 128 KB. Use J or JR instructions to branch to more distant addresses.

BGEZAL : Branch on Greater Than or Equal to Zero and Link

MIPS I

To do a PC-relative conditional procedure call according to the values of a GPR.

Operation Code

31	26	25	21	20	16	15	0
REGIMM	rs		BGEZAL		offset		
000001			10001				
6	5		5		16		

Format

BGEZAL rs, offset

Description

if (GPR[rs] >= 0) then procedure_call

Stores the address of the instruction following the branch delay slot as the return address link in GPR[31]. If the value of GPR[rs] is greater than or equal to zero (sign bit is 0), branches to the target address after the instruction in the delay slot is executed. If the value of GPR[rs] is less than zero, continues execution including the instruction in the branch delay slot.

The target address is the address obtained from adding an 18-bit signed offset (the offset field shifted 2 bits) to the address of the instruction in the branch delay slot (the instruction following the BGEZAL instruction).

Restrictions

GPR[31] must not be used for the source register rs and the result of such an execution is undefined. This restriction permits an exception handler to resume execution by re-executing the branch even when an exception occurs in the branch delay slot.

Exceptions

None

Operation

```

I:      tgt_offset ← sign_extend(offset || 02)
        condition ← GPR[rs]63:0 >= 0
        GPR[31]63:0 ← PC + 8
I+1:    if condition then
        PC ← PC + tgt_offset
        endif

```

Programming Notes

Since the offset is an 18-bit signed offset, the conditional branch range is ± 128 KB. Use JAL or JALR instructions to call a subroutine to more distant addresses.

BGEZALL : Branch on Greater Than or Equal to Zero and Link Likely

MIPS II

To do a PC-relative conditional procedure call according to the values of a GPR. Executes the instruction in the branch delay slot only if the branch is taken.

Operation Code

31	26	25	21	20	16	15	0
REGIMM 000001			rs		BGEZALL 10011		offset
6			5		5		16

Format

BGEZALL rs, offset

Description

if (GPR[rs] >= 0) then procedure_call_likely

Stores the address of the instruction following the branch delay slot as the return address link in GPR[31]. If the value of GPR[rs] is greater than or equal to zero (sign bit is 0), branches to the target address after the instruction in the branch delay slot is executed. If the value of GPR[rs] is not greater than or equal to zero, cancels the instruction in the branch delay slot and continues execution.

The target address is the address obtained from adding an 18-bit signed offset (the offset field shifted 2 bits) to the address of the instruction in the branch delay slot (the instruction following the BGEZALL instruction).

Restrictions

GPR[31] must not be used for the source register rs and the result of such execution is undefined. This restriction permits an exception handler to resume execution by re-executing the branch even when an exception occurs in the branch delay slot.

Exceptions

None

Operation

```

I:      tgt_offset ← sign_extend (offset | | 02)
        condition ← GPR[rs]63..0 >= 0
        GPR[31]63..0 ← PC + 8
I+1:    if condition then
        PC ← PC + tgt_offset
        else
        NullifyCurrentInstruction ()
        endif

```

Programming Notes

Since the offset is an 18-bit signed offset, the conditional branch range is ± 128 KB. Use JAL or JALR instructions to call a subroutine to more distant addresses.

BGEZL : Branch on Greater Than or Equal to Zero Likely

MIPS II

To do a PC-relative branch according to the values of a GPR. Executes the instruction in the branch delay slot only if the branch is taken.

Operation Code

31	26	25	21	20	16	15	0
REGIMM		rs		BGEZL		offset	
000001				00011			
6		5		5		16	

Format

BGEZL rs, offset

Description

if (GPR[rs] >= 0) then branch_likely

If the value of GPR[rs] is greater than or equal to zero (sign bit is 0), branches to the target address after the instruction in the delay slot is executed. If the value of GPR[rs] is not greater than or equal to zero, cancels the instruction in the branch delay slot and continues execution.

The target address is the address obtained from adding an 18-bit signed offset (the offset field shifted left 2 bits) to the address of the instruction in the branch delay slot (the instruction following the BGEZL instruction).

Exceptions

None

Operation

```

I:    tgt_offset ← sign_extend (offset || 02)
      condition ← GPR[rs]63:0 >= 0
I+1:  if condition then
      PC ← PC + tgt_offset
      else
      NullifyCurrentInstruction ()
      endif

```

Programming Notes

Since the offset is an 18-bit signed offset, the conditional branch range is ± 128 KB. Use J or JR instructions to branch to more distant addresses.

BGTZ : Branch on Greater Than Zero

MIPS I

To do a PC-relative branch according to the values of a GPR.

Operation Code

31	26	25	21	20	16	15	0
BGTZ	rs	0	offset				
000111		00000					
6	5	5	16				

Format

BGTZ rs, offset

Description

if (GPR[rs] > 0) then branch

If the value of GPR[rs] is greater than zero (sign bit is zero but value is not zero), branches to the target address after the instruction in the delay slot is executed. If the value of GPR[rs] is less than or equal to zero, continues execution including the instruction in the branch delay slot.

The target address is the address obtained from adding an 18-bit signed offset (the offset field shifted left 2 bits) to the address of the instruction in the branch delay slot (the instruction following the BGTZ instruction).

Exceptions

None

Operation

```

I:      tgt_offset ← sign_extend(offset || 02)
        condition ← GPR[rs]63:0 > 0
I+1:    if condition then
        PC ← PC + tgt_offset
        endif

```

Programming Notes

Since the offset is an 18-bit signed offset, the conditional branch range is ± 128 KB. Use J or JR instructions to branch to more distant addresses.

BGTZL : Branch on Greater Than Zero Likely

MIPS II

To do a PC-relative branch according to the values of a GPR. Executes the instruction in the branch delay slot only if the branch is taken.

Operation Code

31	26	25	21	20	16	15	0
BGTZL		rs		0			offset
010111				00000			
6		5		5			16

Format

BGTZL rs, offset

Description

if (GPR[rs] > 0) then branch_likely

If the value of GPR[rs] is greater than zero (sign bit is zero but value not zero), branches to the target address after the instruction in the delay slot is executed. If the value of GPR[rs] is less than or equal to zero, cancels the instruction in the branch delay slot and continues execution.

The target address is the address obtained from adding an 18-bit signed offset (the offset field shifted left 2 bits) to the address of the instruction in the branch delay slot (the instruction following the BGTZL instruction).

Exceptions

None

Operation

```

I:      tgt_offset ← sign_extend (offset || 02)
        condition ← GPR[rs]63:0 > 0
I+1:    if condition then
            PC ← PC + tgt_offset
        else
            NullifyCurrentInstruction ()
        endif

```

Programming Notes

Since the offset is an 18-bit signed offset, the conditional branch range is ± 128 KB. Use J or JR instructions to branch to more distant addresses.

BLEZ : Branch on Less Than or Equal to Zero

MIPS I

To do a PC-relative branch according to the values of a GPR.

Operation Code

31	26	25	21	20	16	15	0
BLEZ	rs	0	offset				
000110		00000					
6	5	5	16				

Format

BLEZ rs, offset

Description

if (GPR[rs] <= 0) then branch

If the value of GPR[rs] is less than or equal to zero (sign bit is one or value is zero), branches to the target address after the instruction in the delay slot is executed. If the value of GPR[rs] is greater than zero, continues execution including the instruction in the branch delay slot.

The target address is the address obtained from adding an 18-bit signed offset (the offset field shifted 2 bits) to the address of the instruction in the branch delay slot (the instruction following the BLEZ instruction).

Exceptions

None

Operation

```

I:      tgt_offset ← sign_extend (offset || 02)
        condition ← GPR[rs]63..0 <= 0
I+1:    if condition then
        PC ← PC + tgt_offset
        endif

```

Programming Notes

Since the offset is an 18-bit signed offset, the conditional branch range is ± 128 KB. Use J or JR instructions to branch to more distant addresses.

BLEZL : Branch on Less Than or Equal to Zero Likely

MIPS I

To do a PC-relative branch according to the values of a GPR. Executes the instruction in the branch delay slot only if the branch is taken.

Operation Code

31	26	25	21	20	16	15	0
BLEZL		rs		0			offset
010110				00000			
6		5		5			16

Format

BLEZL rs, offset

Description

If the value of GPR[rs] is less than or equal to zero (sign bit is one or value is zero), branches to the target address after the instruction in the delay slot is executed. If the value of GPR[rs] is greater than zero, cancels the instruction in the branch delay slot and continues execution.

The target address is the address obtained from adding an 18-bit signed offset (the offset field shifted 2 bits) to the address of the instruction in the branch delay slot (the instruction following the BLEZL instruction).

Exceptions

None

Operation

```

I:      tgt_offset ← sign_extend(offset || 02)
        condition ← GPR[rs]63:0 ≤ 0
I+1:    if condition then
            PC ← PC + tgt_offset
        else
            NullifyCurrentInstruction ()
        endif

```

Programming Notes

Since the offset is an 18-bit signed offset, the conditional branch range is ± 128 KB. Use J or JR instructions to branch to more distant addresses.

BLTZ : Branch on Less Than Zero

MIPS I

To do a PC-relative branch according to the values of a GPR.

Operation Code

31	26	25	21	20	16	15	0
REGIMM 000001						rs	BLTZ 00000
6						5	5
						offset	
						16	

Format

BLTZ rs, offset

Description

if (GPR[rs] < 0) then branch

If the value of GPR[rs] is less than zero (sign bit is one and value is not zero), branches to the target address after the instruction in the delay slot is executed. If the value of GPR[rs] is greater than or equal to zero, continues execution including the instruction in the branch delay slot.

The target address is the address obtained from adding an 18-bit signed offset (the offset field shifted 2 bits) to the address of the instruction in the branch delay slot (the instruction following the BLTZ instruction).

Exceptions

None

Operation

```

I:      tgt_offset ← sign_extend (offset || 02)
        condition ← GPR[rs]63..0 < 0
I+1:    if condition then
        PC ← PC + tgt_offset
        endif

```

Programming Notes

Since the offset is an 18-bit signed offset, the conditional branch range is ± 128 KB. Use J or JR instructions to branch to more distant addresses.

BLTZAL : Branch on Less Than Zero and Link

MIPS I

To do a PC-relative conditional procedure call according to the values of a GPR.

Operation Code

31	26	25	21	20	16	15	0
REGIMM	rs		BLTZAL		offset		
000001			10000				
6	5		5		16		

Format

BLTZAL rs, offset

Description

if (GPR[rs] < 0) then procedure_call

Stores the address of the instruction following the branch delay slot as the return address link in GPR[31]. If the value of GPR[rs] is less than zero (sign bit is 1), branches to the target address after the instruction in the delay slot is executed. If the value of GPR[rs] is greater than or equal to zero, continues execution including the instruction in the branch delay slot.

The target address is the address obtained from adding an 18-bit signed offset (the offset field shifted 2 bits) to the address of the instruction in the branch delay slot (the instruction following the BLTZAL instruction).

Restrictions

GPR[31] must not be used for the source register rs and the result of such execution is undefined. This restriction permits an exception handler to resume execution by re-executing the branch even when an exception occurs in the branch delay slot.

Exceptions

None

Operation

```

I:      tgt_offset ← sign_extend (offset | | 02)
        condition ← GPR[rs]63..0 < 0
        GPR[31]63..0 ← PC + 8
I+1:    if condition then
        PC ← PC + tgt_offset
        endif

```

Programming Notes

Since the offset is an 18-bit signed offset, the conditional branch range is ± 128 KB. Use JAL or JALR instructions to call a subroutine to more distant addresses.

BLTZALL : Branch on Less Than Zero and Link Likely

MIPS II

To do a PC-relative conditional procedure call according to the values of a GPR. Executes the instruction in the branch delay slot only if the branch is taken.

Operation Code

31	26	25	21	20	16	15	0
REGIMM 000001			rs		BLTZALL 10010		offset
6			5		5		16

Format

BLTZALL rs, offset

Description

if (GPR[rs] < 0) then procedure_call_likely

Stores the address of the instruction following the branch delay slot as the return address link in GPR[31]. If the value of GPR[rs] is less than zero (sign bit is 1), branches to the target address after the instruction in the delay slot is executed. If the value of GPR[rs] is greater than or equal to zero, continues execution including the instruction in the branch delay slot.

The target address is the address obtained from adding an 18-bit signed offset (the offset field shifted 2 bits) to the address of the instruction in the branch delay slot (the instruction following the BLTZALL instruction).

Restrictions

GPR[31] must not be used for the source register rs and the result of such execution is undefined. This restriction permits an exception handler to resume execution by re-executing the branch even when an exception occurs in the branch delay slot.

Exceptions

None

Operation

```

I:      tgt_offset ← sign_extend (offset | 02)
        condition ← GPR[rs]63..0 < 0
        GPR[31]63..0 ← PC+8
I+1:    if condition then
        PC ← PC + tgt_offset
        else
        NullifyCurrentInstruction ()
        endif

```

Programming Notes

Since the offset is an 18-bit signed offset, the conditional branch range is ± 128 KB. Use JAL or JALR instructions to call a subroutine to more distant addresses.

BLTZL : Branch on Less Than Zero Likely

MIPS II

To do a PC-relative conditional branch according to the values of a GPR. Executes the instruction in the branch delay slot only if the branch is taken.

Operation Code

31	26	25	21	20	16	15	0
REGIMM 000001	rs				BLTZL 00010	offset	
6	5				5	16	

Format

BLTZL rs, offset

Description

if (GPR[rs] < 0) then branch_likely

If the value of GPR[rs] is less than zero (sign bit is 1), branches to the target address after the instruction in the delay slot is executed. If the value of GPR[rs] is greater than or equal to zero, cancels the instruction in the branch delay slot and continues execution.

The target address is the address obtained from adding an 18-bit signed offset (the offset field shifted 2 bits) to the address of the instruction in the branch delay slot (the instruction following the BLTZL instruction).

Exceptions

None

Operation

```

I:      tgt_offset ← sign_extend (offset | | 02)
        condition ← GPR[rs]63:0 < 0
I+1:    if condition then
            PC ← PC + tgt_offset
        else
            NullifyCurrentInstruction ()
        endif

```

Programming Notes

Since the offset is an 18-bit signed offset, the conditional branch range is ± 128 KB. Use J or JR instructions to branch to more distant addresses.

BNE : Branch on Not Equal

MIPS I

To compare the value of GPRs and then do a PC-relative conditional branch according to the result.

Operation Code

31	26	25	21	20	16	15	0
BNE 000101		rs		rt		offset	
6		5		5		16	

Format

BNE rs, rt, offset

Description

if (GPR[rs] \neq GPR[rt]) then branch

If the values of GPR[rs] and GPR[rt] are not equal, branches to the target address after the instruction in the branch delay slot is executed. If they are equal, continues execution including the instruction in the branch delay slot.

The target address is the address obtained from adding an 18-bit signed offset (the offset field shifted 2 bits) to the address of the instruction in the branch delay slot (the instruction following the BNE instruction).

Exceptions

None

Operation

```

I:      tgt_offset  $\leftarrow$  sign_extend(offset | 02)
        condition  $\leftarrow$  (GPR[rs]63..0  $\neq$  GPR[rt]63..0)
I+1:    if condition then
        PC  $\leftarrow$  PC + tgt_offset
        endif

```

Programming Notes

Since the offset is an 18-bit signed offset, the conditional branch range is ± 128 KB. Use J or JR instructions to branch to more distant addresses.

BNEL : Branch on Not Equal Likely

MIPS II

To compare the value of GPRs and then do a PC-relative conditional branch according to the result. Executes the instruction in the delay slot only if the branch is taken.

Operation Code

31	26	25	21	20	16	15	0
BNEL 010101		rs		rt		offset	
6		5		5		16	

Format

BNEL rs, rt, offset

Description

if (GPR[rs] \neq GPR[rt]) then branch_likely

If the values of GPR[rs] and GPR[rt] are not equal, branches to the target address after the instruction in the branch delay slot is executed. If they are equal, cancels the instruction in the branch delay slot and continues execution.

The target address is the address obtained from adding an 18-bit signed offset (the offset field shifted 2 bits) to the address of the instruction in the branch delay slot (the instruction following the BNEL instruction).

Exceptions

None

Operation

```

I:      tgt_offset  $\leftarrow$  sign_extend (offset | 02)
        condition  $\leftarrow$  (GPR[rs]63..0  $\neq$  GPR[rt]63..0)

I+1:    if condition then
        PC  $\leftarrow$  PC + tgt_offset
    else
        NullifyCurrentInstruction ()
    endif

```

Programming Notes

Since the offset is an 18-bit signed offset, the conditional branch range is ± 128 KB. Use J or JR instructions to branch to more distant addresses.

BREAK : Breakpoint

MIPS I

To cause a Breakpoint exception.

Operation Code

31	26	25		6	5	0
SPECIAL 000000			code			BREAK 001101
6			20			6

Format

BREAK

Description

A breakpoint exception occurs, immediately and unconditionally transferring control to the exception handler.

The code field is available to be used for software parameters.

However, no special way for the exception handler to acquire the value of the code field is provided. It must be retrieved by determining the address of an instruction word from the EPC register, etc.

Exceptions

Breakpoint

Operation

SignalException (Breakpoint)

DADD : Doubleword Add

MIPS III

To add 64-bit integers. Traps if overflow occurs.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000	rs				rt				rd		0 00000
6	5				5				5		6
											DADD 101100

Format

DADD rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] + GPR[rt]$

Adds the 64-bit value in GPR[rt] to the 64-bit value in GPR[rs]. The result is stored in GPR[rd]. If the addition results in 64-bit 2's complement arithmetic overflow, then the contents of GPR[rd] are not changed and an Integer Overflow exception occurs.

Exceptions

Integer Overflow

Operation

```
temp ← GPR[rs]63..0 + GPR[rt]63..0
if (64_bit_arithmetic_overflow) then
    SignalException (IntegerOverflow)
else
    GPR[rd]63..0 ← temp
endif
```

Programming Notes

DADDU performs the same arithmetic operation but does not trap on overflow.

DADDI : Doubleword Add Immediate

MIPS III

To add a 16-bit immediate value to a 64-bit integers. Traps if overflow occurs.

Operation Code

31	26	25	21	20	16	15	0
DADDI						rs	
011000						rt	
						immediate	
6						5	
						5	
						16	

Format

DADDI rt, rs, immediate

Description

$GPR[rt] \leftarrow GPR[rs] + \text{immediate}$

Adds the value of the immediate field as a 16-bit signed integer to the 64-bit integer value in GPR[rs]. The result is stored in GPR[rt]. If the addition results in 64-bit 2's complement overflow, then the contents of GPR[rt] are not changed and an Integer Overflow exception occurs.

Exceptions

Integer Overflow

Operation

```
temp ← GPR[rs]63..0 + sign_extend(immediate)
if (64_bit_arithmetic_overflow) then
    SignalException (IntegerOverflow)
else
    GPR[rt]63..0 ← temp
endif
```

Programming Notes

DADDIU performs the same arithmetic operation but does not trap on overflow.

DADDIU : Doubleword Add Immediate Unsigned

MIPS III

To add a 16-bit immediate value to a 64-bit integer.

Operation Code

31	26	25	21	20	16	15	0
DADDIU	rs	rt	immediate				
011001							
6	5	5	16				

Format

DADDIU rt, rs, immediate

Description

$GPR[rt] \leftarrow GPR[rs] + \text{immediate}$

Adds the value of the immediate field as a 16-bit signed integer to the 64-bit integer value in GPR[rs]. The result is stored in GPR[rt]. Regardless of the result, an Integer Overflow exception does not occur.

Exceptions

None

Operation

$GPR[rt]_{63..0} \leftarrow GPR[rs]_{63..0} + \text{sign_extend}(\text{immediate})$

Programming Notes

This instruction is not an unsigned operation in the strict sense and performs 64-bit modulo arithmetic that ignores overflow. It is appropriate for integer arithmetic that ignores overflow such as address or C language arithmetic.

DADDU : Doubleword Add Unsigned

MIPS III

To add 64-bit integers.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000	rs	rt	rd	0 00000	DADDU 101101						
6	5	5	5	5	6						

Format

DADDU rd, rs, rt

Description

$rd \leftarrow GPR[rs] + rt$

Adds the 64-bit value in GPR[rt] to the 64-bit value in GPR[rs]. The result is stored in GPR[rd]. Regardless of the result, an Integer Overflow exception does not occur.

Exceptions

None

Operation

$GPR[rd]_{63..0} \leftarrow GPR[rs]_{63..0} + GPR[rt]_{63..0}$

Programming Notes

This instruction is not an unsigned operation in the strict sense and performs 64-bit modulo arithmetic that ignores overflow. It is appropriate for integer arithmetic that ignores overflow such as address or C language arithmetic.

DIV : Divide Word

MIPS I

To divide 32-bit signed integers.

Operation Code

31	26	25	21	20	16	15		6	5	0	
SPECIAL 000000			rs		rt		0 00 0000 0000			DIV 011010	
6			5		5		10			6	

Format

DIV rs, rt

Description

$(LO, HI) \leftarrow GPR[rs] / GPR[rt]$

The 32-bit value in GPR[rs] is divided by the 32-bit value in GPR[rt]. The 32-bit quotient is stored in the LO register and the 32-bit remainder is stored in the HI register. Both GPR[rs] and GPR[rt] are treated as signed values. The sign of the quotient and remainder are determined as shown in the table below.

Dividend GPR[rs]	Divisor GPR[rt]	Quotient LO	Remainder HI
Positive	Positive	Positive	Positive
Positive	Negative	Negative	Positive
Negative	Positive	Negative	Negative
Negative	Negative	Positive	Negative

No arithmetic exception such as divide-by-zero or overflow occurs under any circumstances.

Restrictions

If GPR[rt] or GPR[rs] are not sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined (Bits 63 to 32 are not used for the operation itself, but this restriction is applied). Also, if the value of GPR[rt] is zero, the value of the arithmetic result is undefined.

Exceptions

None

Operation

if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif

quotient $\leftarrow GPR[rs]_{31..0} \text{ DIV } GPR[rt]_{31..0}$

$LO_{63..0} \leftarrow \text{sign_extend}(\text{quotient}_{31..0})$

remainder $\leftarrow GPR[rs]_{31..0} \text{ MOD } GPR[rt]_{31..0}$

$HI_{63..0} \leftarrow \text{sign_extend}(\text{remainder}_{31..0})$

Programming Notes

In the EE Core, the integer divide operation proceeds asynchronously. An attempt to read the contents of the LO or HI register before the divide operation finishes will result in interlock. Other CPU instructions can execute without delay. Therefore, scheduling the divide operation appropriately will improve performance.

Normally, when 0x80000000 (−2147483648), the signed minimum value, is divided by 0xFFFFFFFF (−1), the operation will result in overflow. However, in this instruction an overflow exception does not occur and the result will be as follows;

Quotient: 0x80000000 (−2147483648), and remainder: 0x00000000 (0)

If overflow or divide-by-zero must be detected, then add an instruction that detects these conditions following the divide instruction. Since the divide instruction is asynchronous, the divide operation and check can be executed in parallel. If overflow or divide-by-zero is detected, then to signal the problem to the system software is possible by generating exceptions using an appropriate code value with a BREAK instruction.

DIVU : Divide Unsigned Word

MIPS I

To divide 32-bit unsigned integers.

Operation Code

31	26	25	21	20	16	15		6	5	0
SPECIAL		rs		rt		0			DIVU	
000000						00 0000 0000			011011	
6		5		5		10			6	

Format

DIVU rs, rt

Description

$(LO, HI) \leftarrow GPR[rs] / GPR[rt]$

The 32-bit value in GPR[rs] is divided by the 32-bit value in GPR[rt]. The 32-bit quotient is stored in the LO register and the 32-bit remainder is stored in the HI register. Both GPR[rs] and GPR[rt] are treated as unsigned values.

No arithmetic exception such as divide-by-zero or overflow occurs under any circumstances.

Restrictions

If GPR[rt] or GPR[rs] are not sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined (Bits 63 to 32 are not used for the operation itself, but this restriction is applied). Also, if the value of GPR[rt] is zero, the value of arithmetic operation is undefined.

Exceptions

None

Operation

if (NotWordValue(GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif

quotient $\leftarrow (0 \mid \mid GPR[rs]_{31..0}) \text{ DIV } (0 \mid \mid GPR[rt]_{31..0})$

$LO_{63..0} \leftarrow \text{sign_extend}(\text{quotient}_{31..0})$

remainder $\leftarrow (0 \mid \mid GPR[rs]_{31..0}) \text{ MOD } (0 \mid \mid GPR[rt]_{31..0})$

$HI_{63..0} \leftarrow \text{sign_extend}(\text{remainder}_{31..0})$

Programming Notes

See "Programming Notes" for the DIV instruction.

DSLL : Doubleword Shift Left Logical

MIPS III

To left shift a doubleword. The shift amount is a fixed value (0-31 bits) specified by sa.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000	0 00000	rt	rd	sa	DSLL 111000						
6	5	5	5	5	6						

Format

DSLL rd, rt, sa

Description

$GPR[rd] \leftarrow GPR[rt] \ll sa$

Shifts the 64-bit data in GPR[rt] left by the bit count specified by sa, inserting zeros into the emptied bits.

The result is stored in GPR[rd].

Exceptions

None

Operation

$s \leftarrow 0 \mid sa$

$GPR[rd]_{63..0} \leftarrow GPR[rt]_{(63-s)..0} \mid 0^s$

DSLL32 : Doubleword Shift Left Logical Plus 32

MIPS III

To left shift a doubleword. The shift amount is a fixed value (32-63 bits) specified by sa.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL	0	rt	rd	sa	DSLL32						
000000	00000										111100
6	5	5	5	5	6						

Format

DSLL32 rd, rt, sa

Description

$GPR[rd] \leftarrow GPR[rt] \ll (sa + 32)$

Shifts the 64-bit data in GPR[rt] left by the bit count specified by sa + 32, inserting zeros into the emptied bits. The result is stored in GPR[rd].

Exceptions

None

Operation

$s \leftarrow 1 \mid \mid sa$ /* s = 32 + sa */
 $GPR[rd]_{63..0} \leftarrow GPR[rt]_{(63-s)..0} \mid \mid 0^s$

DSLLV : Doubleword Shift Left Logical Variable

MIPS III

To left shift a doubleword. The shift amount is a variable (specified by a GPR).

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000		rs		rt		rd		0 00000		DSLLV 010100	
6		5		5		5		5		6	

Format

DSLLV rd, rt, rs

Description

$GPR[rd] \leftarrow GPR[rt] \ll GPR[rs]$

Shifts the 64-bit data in GPR[rt] left by the bit count (0-63) specified by the low-order 6 bits in GPR[rs], inserting zeros into the emptied bits. The result is stored in GPR[rd].

Exceptions

None

Operation

$s \leftarrow 0 \mid \mid GPR[rs]_{5..0}$

$GPR[rd]_{63..0} \leftarrow GPR[rt]_{(63-s)..0} \mid \mid 0^s$

DSRA : Doubleword Shift Right Arithmetic

MIPS III

To arithmetic right shift a doubleword. The shift amount is a fixed value (0-31 bits) specified by sa.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL	0		rt		rd		sa		DSRA		
000000	00000								111011		
6	5		5		5		5		6		

Format

DSRA rd, rt, sa

Description

$GPR[rd] \leftarrow GPR[rt] \gg sa$ (arithmetic)

Shifts the 64-bit data in GPR[rt] right by the bit count (0-31) specified by sa, duplicating the sign bit (bit 63) into the emptied bits. The result is stored in GPR[rd].

Exceptions

None

Operation

$s \leftarrow 0 \mid \mid sa$

$GPR[rd]_{63..0} \leftarrow (GPR[rt]_{63})^s \mid \mid GPR[rt]_{63..s}$

DSRA32 : Doubleword Shift Right Arithmetic Plus 32

MIPS III

To arithmetic right shift a doubleword. The shift amount is a fixed value (32-63 bits) specified by sa.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000	0 00000	rt	rd	sa	DSRA32 111111						
6	5	5	5	5	6						

Format

DSRA32 rd, rt, sa

Description

$GPR[rd] \leftarrow GPR[rt] \gg (sa + 32)$ (arithmetic)

Shifts the 64-bit data in GPR[rt] right by the bit count (0-31) specified by sa + 32, duplicating the sign bit (bit 63) into the emptied bits. The result is stored in GPR[rd].

Exceptions

None

Operation

$s \leftarrow 1 \mid \mid sa$ /* s = 32 + sa */
 $GPR[rd]_{63..0} \leftarrow (GPR[rt]_{63})^s \mid \mid GPR[rt]_{63..s}$

DSRAV : Doubleword Shift Right Arithmetic Variable

MIPS III

To arithmetic right shift a doubleword. The shift amount is a variable value (0-63 bits) specified by a GPR.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000		rs		rt		rd		0 00000		DSRAV 010111	
6		5		5		5		5		6	

Format

DSRAV rd, rt, rs

Description

$GPR[rd] \leftarrow GPR[rt] \gg GPR[rs]$ (arithmetic)

Shifts the 64-bit data in GPR[rt] right by the bit count specified by the low-order 6-bits in GPR[rs], duplicating the sign bit (bit 63) into the emptied bits. The result is stored in GPR[rd].

Exceptions

None

Operation

$s \leftarrow GPR[rs]_{5..0}$

$GPR[rd]_{63..0} \leftarrow (GPR[rt]_{63})^s \mid \mid GPR[rt]_{63..s}$

DSRL : Doubleword Shift Right Logical

MIPS III

To logical right shift a doubleword. The shift amount is a fixed value (0-31 bit) specified by sa.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000	0 00000	rt	rd	sa	DSRL 111010						
6	5	5	5	5	6						

Format

DSRL rd, rt, sa

Description

$GPR[rd] \leftarrow GPR[rt] \gg sa$ (logical)

Shifts the 64-bit data in GPR[rt] right by the bit count (0-31) specified by sa, inserting zeros into the emptied bits. The result is stored in GPR[rd].

Exceptions

None

Operation

$s \leftarrow 0 \mid sa$

$GPR[rd]_{63..0} \leftarrow 0^s \mid GPR[rt]_{63..s}$

DSRL32 : Doubleword Shift Right Logical Plus 32

MIPS III

To logical right shift a doubleword. The shift amount is a fixed value (32-63 bits) specified by sa.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL	0		rt		rd		sa		DSRL32		
000000	00000								111110		
6	5		5		5		5		6		

Format

DSRL32 rd, rt, sa

Description

$GPR[rd] \leftarrow GPR[rt] \gg (sa + 32)$ (logical)

Shifts the 64-bit data in GPR[rt] right by the bit count (32-63) specified by sa, inserting zeros into the emptied bits. The result is stored in GPR[rd].

Exceptions

None

Operation

$s \leftarrow 1 \mid \mid sa$

$/* s = 32 + sa */$

$GPR[rd]_{63:0} \leftarrow 0^s \mid \mid GPR[rt]_{63:s}$

DSRLV : Doubleword Shift Right Logical Variable

MIPS III

To logical right shift a doubleword. The shift amount is a variable value (0-63 bits) specified by a GPR.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000						rs		rt		rd	
6						5		5		5	
						0 00000		DSRLV 010110		6	

Format

DSRLV rd, rt, rs

Description

$GPR[rd] \leftarrow GPR[rt] \gg GPR[rs]$ (logical)

Shifts the 64-bit data in GPR[rt] right by the bit count (0-63) specified by the low-order 6 bits in GPR[rs], inserting zeros into the emptied bits. The result is stored in GPR[rd].

Exceptions

None

Operation

$s \leftarrow GPR[rs]_{5:0}$

$GPR[rd]_{63:0} \leftarrow 0^s \parallel GPR[rt]_{63:s}$

DSUB : Doubleword Subtract

MIPS III

To subtract 64-bit integers. Traps if overflow occurs.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000		rs		rt		rd		0 00000		DSUB 101110	
6		5		5		5		5		6	

DSUBU : Doubleword Subtract Unsigned

MIPS III

To subtract 64-bit integers.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000						rs		rt		rd	
000000						000000		DSUBU 101111			
6						5		5		5	

Format

DSUBU rd, rs, rt

Description

$$\text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rs}] - \text{GPR}[\text{rt}]$$

Subtracts the 64-bit value in GPR[rt] from the 64-bit value in GPR[rs]. The result is stored in GPR[rd].

Regardless of the arithmetic result, an Integer Overflow exception does not occur.

Exceptions

None

Operation

$$\text{GPR}[\text{rd}]_{63..0} \leftarrow \text{GPR}[\text{rs}]_{63..0} - \text{GPR}[\text{rt}]_{63..0}$$

Programming Notes

This instruction is not an unsigned operation in the strict sense and performs 64-bit modulo arithmetic that ignores overflow. It is appropriate for integer arithmetic that ignores overflow such as address or C language arithmetic.

J : Jump

MIPS I

To do an unconditional branch by an absolute address within a 256 MB region.

Operation Code

31	26	25	0
J	offset		
000010			
6	26		

Format

J target

Description

Executes the instruction in the branch delay slot and then branches to the target address unconditionally.

The target address is the address adding the high-order bits of the instruction address in the branch delay slot (the address of the J instruction itself + 4) to the 28-bit value obtained by shifting the offset left 2 bits. That is, though it is not PC-relative, the branch target is restricted to a memory region aligned on a 256 MB boundary, the same region as the current PC region.

Exceptions

None

Operation

I: (Explicit Operation: None)

I+1: $PC \leftarrow PC_{31..28} || \text{offset} || 0^s$

Programming Notes

The J instruction can branch to any addresses within a 256 MB region, while conditional branch instructions such as BEQ are limited to ± 128 KB.

If the J instruction is in the last word of a 256 MB region, the branch target is restricted to the following 256 MB region, because the basis of the branch is not the J instruction itself but the following instruction. Note that this is an exceptional case.

JAL : Jump and Link

MIPS I

To call a subroutine by an absolute address within a 256 MB region.

Operation Code

31	26	25	0
JAL 000011	instr_index		
6	26		

Format

JAL target

Description

Stores the address of the instruction following the branch delay slot (the address of the J instruction itself + 8) in GPR[31] and branches to the target address after the instruction in the delay slot is executed.

The target address is the address adding the high-order bits of the instruction address in the branch delay slot (the address of J instruction itself + 4) to the 28-bit value obtained by shifting the instr_index left 2 bits. That is, though it is not PC-relative, the branch target is restricted to a memory region aligned on a 256 MB boundary, the same region as the current PC region.

Exceptions

None

Operation

I: $\text{GPR}[31]_{63..0} \leftarrow \text{zero_extend}(\text{PC} + 8)$
 I+1: $\text{PC} \leftarrow \text{PC}_{31..28} \parallel \text{instr_index} \parallel 0^s$

Programming Notes

The JAL instruction can call subroutines in any addresses within a 256 MB region, while the range of conditional branch instructions such as BGEZAL are limited to ± 128 KB.

If the JAL instruction is in the last word of a 256 MB region, the branch target is restricted to the following 256 MB region, because the basis of the branch is not the JAL instruction itself but the following instruction. Note that this is an exceptional case.

JALR : Jump and Link Register

MIPS I

To call a subroutine at the address specified by a GPR.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000	rs				0 00000	rd				0 00000	JALR 001001
6	5				5	5				5	6

Format

JALR rs (rd = 31 implied)

JALR rd, rs

Description

$\text{GPR}[\text{rd}] \leftarrow \text{return_addr}$, $\text{PC} \leftarrow \text{GPR}[\text{rs}]$

Stores the address of the instruction following the branch delay slot (the address of the J instruction itself + 8) in GPR[rd] and branches to the target address after the instruction in the delay slot is executed. The target address is the value of GPR[rs].

Restrictions

rs and rd have to specify different registers. If they specify the same register, the result of the execution is undefined. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

The value of GPR[rs] must be naturally aligned. If either of the two least-significant bits is not zero, an Address Error exception occurs when the instruction of the branch target is fetched (not when the JALR instruction is executed).

Exceptions

None

Operation

I: $\text{temp} \leftarrow \text{GPR}[\text{rs}]_{31..0}$
 $\text{GPR}[\text{rd}]_{63..0} \leftarrow \text{zero_extend}(\text{PC} + 8)$
 I+1: $\text{PC} \leftarrow \text{temp}$

Programming Notes

The JALR instruction is the only instruction that can specify the link register. All other link instructions use GPR[31]. The default register for the link register (GPR[rd]) is GPR[31] in the JALR instruction.

JR : Jump Register

MIPS I

To branch to the address specified by a GPR.

Operation Code

31	26	25	21	20	6	5	0
SPECIAL						rs	
000000						0	
						000 0000 0000 0000	
						JR	
						001000	
6						5	
						15	
						6	

Format

JR rs

Description

$PC \leftarrow GPR[rs]$

Branches to the target address after the instruction in the delay slot is executed. The target address is the value of GPR[rs].

Restrictions

The value of GPR[rs] must be naturally aligned. If either of the two least-significant bits is not zero, an Address Error exception occurs when the instruction of the branch target is fetched (not when the JR instruction is executed).

Exceptions

None

Operation

I: $temp \leftarrow GPR[rs]_{31..0}$

I+1: $PC \leftarrow temp$

LB : Load Byte

MIPS I

To load a byte from memory as a signed value.

Operation Code

31	26	25	21	20	16	15	0
LB						base	
100000						rt	
						offset	
6						5	
						5	
						16	

Format

LB rt, offset (base)

Description

$GPR[rt] \leftarrow \text{memory}[GPR[base] + \text{offset}]$

Adds the offset as a 16-bit signed number to the value of GPR[base] to form the effective address. Fetches the byte at the address, sign-extends it and stores it in GPR[rt].

Exceptions

TLB Refill, TLB Invalid, Address Error

Operation (128-bit bus)

$vAddr \leftarrow \text{sign_extend}(\text{offset}) + GPR[base]_{31..0}$

$(pAddr, \text{uncached}) \leftarrow \text{AddressTranslation}(vAddr, \text{DATA}, \text{LOAD})$

$pAddr \leftarrow pAddr_{(PSIZE-1)..0}$

$\text{memquad} \leftarrow \text{LoadMemory}(\text{uncached}, \text{BYTE}, pAddr, vAddr, \text{DATA})$

$\text{byte} \leftarrow vAddr_{3..0}$

$GPR[rt]_{63..0} \leftarrow \text{sign_extend}(\text{memquad}_{(7+8*\text{byte})..8*\text{byte}})$

LBU : Load Byte Unsigned

MIPS I

To load a byte from memory as an unsigned value.

Operation Code

31	26	25	21	20	16	15	0
LBU						base	
100100						rt	
						offset	
6						5	
						5	
						16	

Format

LBU rt, offset(base)

Description

$\text{GPR}[\text{rt}] \leftarrow \text{memory}[\text{GPR}[\text{base}] + \text{offset}]$

Adds the offset as a 16-bit signed number to the value of GPR[base] to form the effective address. Fetches the byte at the address, zero-extends it and stores it in GPR[rt].

Exceptions

TLB Refill, TLB Invalid, Address Error

Operation (128-bit bus)

$\text{vAddr} \leftarrow \text{sign_extend}(\text{offset}) + \text{GPR}[\text{base}]_{31..0}$

$(\text{pAddr}, \text{uncached}) \leftarrow \text{AddressTranslation}(\text{vAddr}, \text{DATA}, \text{LOAD})$

$\text{pAddr} \leftarrow \text{pAddr}_{(\text{PSIZE}-1)..0}$

$\text{memquad} \leftarrow \text{LoadMemory}(\text{uncached}, \text{BYTE}, \text{pAddr}, \text{vAddr}, \text{DATA})$

$\text{byte} \leftarrow \text{vAddr}_{3..0}$

$\text{GPR}[\text{rt}]_{63..0} \leftarrow \text{zero_extend}(\text{memquad}_{(7+8*\text{byte})..8*\text{byte}})$

LD : Load Doubleword

MIPS III

To load a doubleword from memory.

Operation Code

31	26	25	21	20	16	15	0
LD	base	rt	offset				
110111							
6	5	5	16				

Format

LD rt, offset (base)

Description

$GPR[rt] \leftarrow \text{memory}[GPR[base] + \text{offset}]$

Adds the offset as a 16-bit signed number to the value of $GPR[base]$ to form the effective address. Fetches the doubleword at the address and stores it in $GPR[rt]$.

Restrictions

The effective address must be aligned on a doubleword boundary. If any of the three least-significant bits of the effective addresses are non-zero, an Address Error exception occurs.

Exceptions

TLB Refill, TLB Invalid, Address Error

Operation (128-bit bus)

$vAddr \leftarrow \text{sign_extend}(\text{offset}) + GPR[base]_{31..0}$

if $(vAddr_{2..0}) \neq 0^3$ then $\text{SignalException}(\text{AddressError})$ endif

$(pAddr, \text{uncached}) \leftarrow \text{AddressTranslation}(vAddr, \text{DATA}, \text{LOAD})$

$pAddr \leftarrow pAddr_{(PSIZE-1)..0}$

$\text{byte} \leftarrow vAddr_{3..0}$

$\text{memquad} \leftarrow \text{LoadMemory}(\text{uncached}, \text{DOUBLEWORD}, pAddr, vAddr, \text{DATA})$

$GPR[rt]_{63..0} \leftarrow \text{memquad}_{(63+8*\text{byte})..8*\text{byte}}$

LDL : Load Doubleword Left

MIPS III

To load the upper part of an unaligned doubleword.

Operation Code

31	26	25	21	20	16	15	0
LDL	base	rt	offset				
011010							
6	5	5	16				

Format

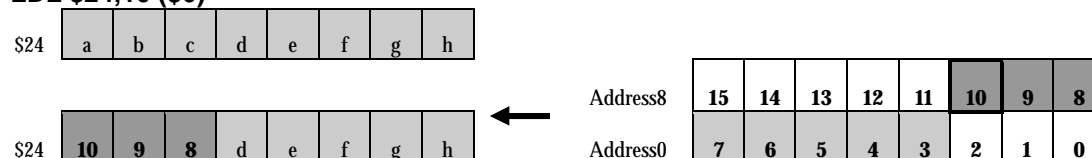
LDL rt, offset (base)

Description

$GPR[rt] \leftarrow GPR[rt] \text{ MERGE memory } [GPR[base] + \text{offset}]$

Adds the offset as a 16-bit signed number to the value of $GPR[base]$ to form the effective address. Stores the low-order bytes starting from the effective address in the aligned doubleword including the address into the upper part of $GPR[rt]$. The low-order bytes of $GPR[rt]$ that are not loaded are not changed.

LDL \$24,10 (\$0)



An Address Error exception due to alignment of the effective address does not occur.

LDL and LDR instructions in a pair are used to load an 8-byte block that does not conform to the doubleword alignment.

Exceptions

TLB Refill, TLB Invalid, Address Error

Operation (128-bit bus)

$vAddr \leftarrow \text{sign_extend}(\text{offset}) + GPR[base]_{31..0}$

$(pAddr, \text{uncached}) \leftarrow \text{AddressTranslation}(vAddr, \text{DATA}, \text{LOAD})$

$pAddr \leftarrow pAddr_{(PSIZE-1)..3} \parallel 0^3$

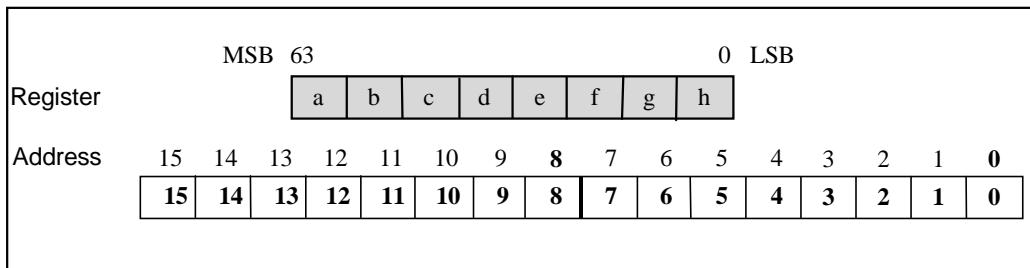
$\text{byte} \leftarrow 0 \parallel vAddr_{2..0}$

$\text{doubleword} \leftarrow vAddr_3$

$\text{memquad} \leftarrow \text{LoadMemory}(\text{uncached}, \text{byte}, pAddr, vAddr, \text{DATA})$

$GPR[rt]_{63..0} \leftarrow \text{memquad}_{(7+8*\text{byte}+64*\text{doubleword})..(64*\text{doubleword})} \parallel GPR[rt]_{(55-8*\text{byte})..0}$

The relation between the low-order 4 bits of the effective address $vAddr$ and bytes that are to be loaded is illustrated below.



vAddr _{3,0}	Contents of registers after instruction (Shaded is unchanged)								Access Type	pAddr _{3,0}
	bit 63							bit 0		
0	0	b	c	d	e	f	g	h	7	0
1	1	0	c	d	e	f	g	h	6	0
2	2	1	0	d	e	f	g	h	5	0
3	3	2	1	0	e	f	g	h	4	0
4	4	3	2	1	0	f	g	h	3	0
5	5	4	3	2	1	0	g	h	2	0
6	6	5	4	3	2	1	0	h	1	0
7	7	6	5	4	3	2	1	0	0	0
8	8	b	c	d	e	f	g	h	7	8
9	9	8	c	d	e	f	g	h	6	8
10	10	9	8	d	e	f	g	h	5	8
11	11	10	9	8	e	f	g	h	4	8
12	12	11	10	9	8	f	g	h	3	8
13	13	12	11	10	9	8	g	h	2	8
14	14	13	12	11	10	9	8	h	1	8
15	15	14	13	12	11	10	9	8	0	8

Programming Notes

In the LDL instruction, the contents of GPR[rt] are referenced. However, since bypassing is performed internally, even when loading the value in GPR[rt] in the preceding instruction, a NOP does not need to be inserted between the instruction and the LDL instruction.

LDR : Load Doubleword Right

MIPS III

To load the lower part of an unaligned doubleword.

Operation Code

31	26	25	21	20	16	15	0
LDR	base	rt	offset				
011011							
6	5	5	16				

Format

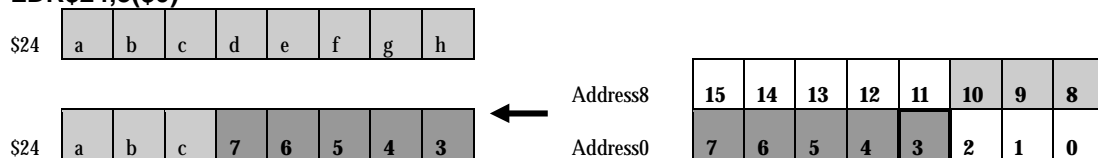
LDR rt, offset (base)

Description

$GPR[rt] \leftarrow GPR[rt] \text{ MERGE memory } [GPR[base] + \text{offset}]$

Adds the offset as a 16-bit signed number to the value of $GPR[base]$ to form the effective address. Stores the high-order bytes starting from the effective address in the aligned doubleword including the address into the lower part of $GPR[rt]$. The high-order bytes of $GPR[rt]$ that are not loaded are not changed.

LDR\$24,3(\$0)



An Address Error exception due to alignment of the effective address does not occur.

LDR and LDL instructions in a pair are used to load an 8-byte block that does not conform to doubleword alignment.

Exceptions

TLB Refill, TLB Invalid, Address Error

Operation (128-bit bus)

$vAddr \leftarrow \text{sign_extend}(\text{offset}) + GPR[base]_{31..0}$

$(pAddr, \text{uncached}) \leftarrow \text{AddressTranslation}(vAddr, \text{DATA}, \text{LOAD})$

$pAddr \leftarrow pAddr_{(PSIZE-1)..0}$

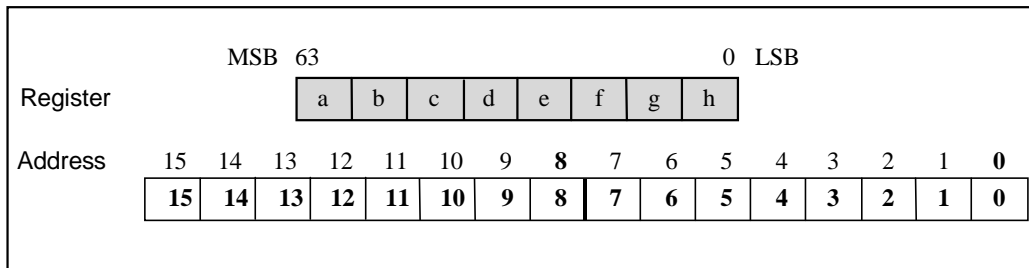
$\text{byte} \leftarrow 0 \mid \mid vAddr_{2..0}$

$\text{doubleword} \leftarrow vAddr_3$

$\text{memquad} \leftarrow \text{LoadMemory}(\text{uncached}, \text{byte}, pAddr, vAddr, \text{DATA})$

$GPR[rt]_{63..0} \leftarrow GPR[rt]_{63..(64-8*\text{byte})} \mid \mid \text{memquad}_{(63+64*\text{doubleword})..(64*\text{doubleword}+8*\text{byte})}$

The relation between the low-order 4 bits of the effective address $vAddr$ and bytes that are to be loaded is illustrated below.



vAddr _{3..0}	Contents of registers after instruction (Shaded is unchanged)									Access Type	pAddr _{3..0}
	bit 63								bit 0		
0	7	6	5	4	3	2	1	0		7	0
1	a	7	6	5	4	3	2	1		6	1
2	a	b	7	6	5	4	3	2		5	2
3	a	b	c	7	6	5	4	3		4	3
4	a	b	c	d	7	6	5	4		3	4
5	a	b	c	d	e	7	6	5		2	5
6	a	b	c	d	e	f	7	6		1	6
7	a	b	c	d	e	f	g	7		0	7
8	15	14	13	12	11	10	9	8		7	8
9	a	15	14	13	12	11	10	9		6	9
10	a	b	15	14	13	12	11	10		5	10
11	a	b	c	15	14	13	12	11		4	11
12	a	b	c	d	15	14	13	12		3	12
13	a	b	c	d	e	15	14	13		2	13
14	a	b	c	d	e	f	15	14		1	14
15	a	b	c	d	e	f	g	15		0	15

Programming Notes

In the LDR instruction, the contents of GPR[rt] are referenced. However, since bypassing is performed internally, even when loading the value in GPR[rt] in the preceding instruction, a NOP does not need to be inserted between the instruction and the LDR instruction.

LH : Load Halfword

MIPS I

To load a halfword from memory as a signed value.

Operation Code

31	26	25	21	20	16	15	0
LH	base	rt	offset				
100001							
6	5	5	16				

Format

LH rt, offset(base)

Description

$\text{GPR}[\text{rt}] \leftarrow \text{memory}[\text{GPR}[\text{base}] + \text{offset}]$

Adds the offset as a 16-bit signed number to the value of GPR[base] to form the effective address. Loads the halfword data at the address, sign-extends it and stores it in GPR[rt].

Restrictions

The effective address must conform to halfword alignment. That is, if the least-significant bit of the address is non-zero, an Address Error exception occurs.

Exceptions

TLB Refill, TLB Invalid, Address Error

Operation (128-bit bus)

```

vAddr ← sign_extend(offset) + GPR[base]31..0
if (vAddr0) ≠ 0 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation(vAddr, DATA, LOAD)
pAddr ← pAddr(PSIZE-1)..0
memquad ← LoadMemory(uncached, HALFWORD, pAddr, vAddr, DATA)
byte ← vAddr3..0
GPR[rt]63..0 ← sign_extend(memquad(15+8*byte)..8*byte)

```

LHU : Load Halfword Unsigned

MIPS I

To load a halfword from memory as an unsigned value.

Operation Code

31	26	25	21	20	16	15	0
LHU	base	rt	offset				
100101							
6	5	5	16				

Format

LHU rt, offset (base)

Description

$GPR[rt] \leftarrow \text{memory}[GPR[base] + \text{offset}]$

Adds the offset as a 16-bit signed number to the value of $GPR[base]$ to form the effective address. Loads the halfword at the address, zero-extends it and stores it in $GPR[rt]$.

Restrictions

The effective address must conform to halfword alignment. That is, if the least-significant bit of the address is non-zero, an Address Error exception occurs.

Exceptions

TLB Refill, TLB Invalid, Address Error

Operation (128-bit bus)

```

vAddr ← sign_extend(offset) + GPR[base]31..0
if (vAddr0) ≠ 0 then SignalException (AddressError) endif
(pAddr, uncached) ← AddressTranslation(vAddr, DATA, LOAD)
pAddr ← pAddr(PSIZE-1)..0
memquad ← LoadMemory(uncached, HALFWORD, pAddr, vAddr, DATA)
byte ← vAddr3..0
GPR[rt]63..0 ← zero_extend(memquad(15+8*byte)..8*byte)

```

LUI : Load Upper Immediate

MIPS I

To load a constant into the upper half of a word.

Operation Code

31	26	25	21	20	16	15	0
LUI	0		rt				immediate
001111	00000						
6	5		5				16

Format

LUI rt, immediate

Description

$GPR[rt] \leftarrow \text{immediate} \parallel 0^{16}$

Shifts the 16-bit immediate value left 16 bits, adds 16 bits of low-order zeros, sign-extends it and stores it in GPR[rt].

Exceptions

None

Operation

$GPR[rt]_{63..0} \leftarrow \text{sign_extend}(\text{immediate} \parallel 0^{16})$

LW : Load Word

MIPS I

To load a word from memory as a signed value.

Operation Code

31	26	25	21	20	16	15	0
LW		base		rt		offset	
100011							
6		5		5		16	

Format

LW rt, offset (base)

Description

$GPR[rt] \leftarrow \text{memory}[GPR[base] + \text{offset}]$

Adds the offset as a 16-bit signed number to the value of $GPR[base]$ to form the effective address. Loads the word data at the address, sign-extends it and stores it in $GPR[rt]$.

Restrictions

The effective address must conform to halfword alignment. That is, if the two least-significant bits of the address are non-zero, an Address Error exception occurs.

Exceptions

TLB Refill, TLB Invalid, Address Error

Operation (128-bit bus)

```

vAddr ← sign_extend(offset) + GPR[base]31..0
if (vAddr1..0) ≠ 02 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation(vAddr, DATA, LOAD)
pAddr ← pAddr(PSIZE-1)..0
memquad ← LoadMemory(uncached, WORD, pAddr, vAddr, DATA)
byte ← vAddr3..0
GPR[rt]63..0 ← sign_extend(memquad(31+8*byte)..8*byte)

```

LWL : Load Word Left

MIPS I

To load the upper part of an unsigned word.

Operation Code

31	26	25	21	20	16	15	0
LWL 100010	base				rt		offset
6	5				5		16

Format

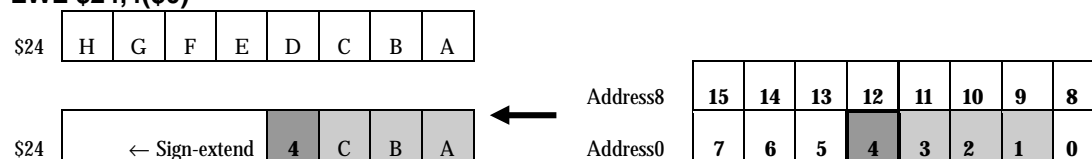
LWL rt, offset (base)

Description

$GPR[rt] \leftarrow GPR[rt] \text{ MERGE memory } [GPR[base] + \text{offset}]$

Adds the offset as a 16-bit signed number to the value of $GPR[base]$ to form the effective address that is considered to be the most-significant byte of the target word. Loads the low-order bytes of the aligned word that contains the most-significant byte into the corresponding bytes of $GPR[rt]$. The high-order words of $GPR[rt]$ are sign-extended and the low-order bytes of $GPR[rt]$ that are not loaded are not changed.

LWL \$24,4(\$0)



An Address Error exception due to alignment of the effective address does not occur.

LWL and LWR instructions in a pair are used to load the 4-byte blocks that do not conform to word alignment.

Exceptions

TLB Refill, TLB Invalid, Address Error

Operation (128-bit bus)

$vAddr \leftarrow \text{sign_extend}(\text{offset}) + GPR[base]_{31:0}$

$(pAddr, \text{uncached}) \leftarrow \text{AddressTranslation}(vAddr, \text{DATA}, \text{LOAD})$

$pAddr \leftarrow pAddr_{(PSIZE-1):3} \mid 0^3$

$\text{byte} \leftarrow 0^2 \mid vAddr_{1:0}$

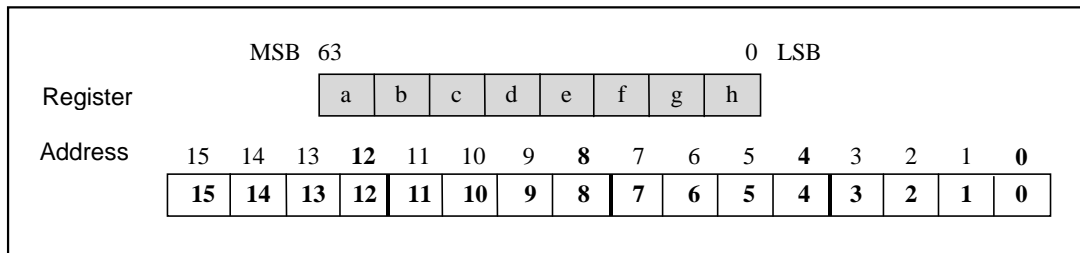
$\text{word} \leftarrow vAddr_{3:2}$

$\text{memquad} \leftarrow \text{LoadMemory}(\text{uncached}, \text{byte}, pAddr, vAddr, \text{DATA})$

$\text{temp} \leftarrow \text{memquad}_{(32*\text{word}+8*\text{byte}+7):32*\text{word}} \mid GPR[rt]_{(23-8*\text{byte}):0}$

$GPR[rt]_{63:0} \leftarrow (\text{temp}_{31})^{32} \mid \text{temp}$

The relation between the low-order 4 bits of the effective address $vAddr$ and bytes that are to be loaded is illustrated below.



vAddr _{3,0}	Contents of registers after instruction (Shaded is unchanged)						Access Type	pAddr _{3,0}
	bit 63	32	31			bit 0		
0	Sign extension of 0←	0		f	g	h	0	0
1	Sign extension of 1←	1	0		g	h	1	0
2	Sign extension of 2←	2	1	0		h	2	0
3	Sign extension of 3←	3	2	1	0		3	0
4	Sign extension of 4←	4	f	g	h		0	4
5	Sign extension of 5←	5	4	g	h		1	4
6	Sign extension of 6←	6	5	4	h		2	4
7	Sign extension of 7←	7	6	5	4		3	4
8	Sign extension of 8←	8	f	g	h		0	8
9	Sign extension of 9←	9	8	g	h		1	8
10	Sign extension of 10←	10	9	8	h		2	8
11	Sign extension of 11←	11	10	9	8		3	8
12	Sign extension of 12←	12	f	g	h		0	12
13	Sign extension of 13←	13	12	g	h		1	12
14	Sign extension of 14←	14	13	12	h		2	12
15	Sign extension of 15←	15	14	13	12		3	12

Programming Notes

In the LWL instruction, the contents of GPR[rt] are referenced. However, since bypassing is performed internally, even when loading the value in GPR[rt] in the preceding instruction, a NOP does not need to be inserted between the instruction and the LWL instruction.

An instruction that treats an unaligned word as unsigned is not provided.

LWR : Load Word Right

MIPS I

To load the lower part of an unsigned word.

Operation Code

31	26	25	21	20	16	15	0
LWR	base	rt	offset				
100110							
6	5	5	16				

Format

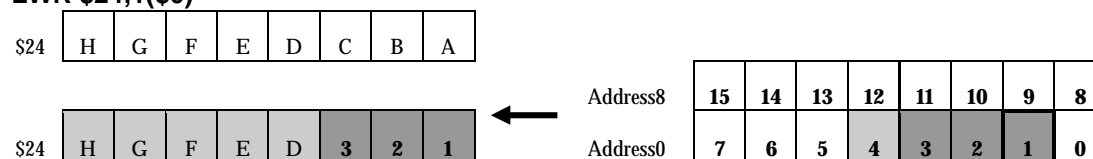
LWR rt, offset (base)

Description

$GPR[rt] \leftarrow GPR[rt] \text{ MERGE memory } [GPR[base] + \text{offset}]$

Adds the offset as a 16-bit signed number to the value of $GPR[base]$ to form the effective address that is considered to be the least-significant byte of the target word. Loads the high-order bytes in the aligned word that contains the least-significant byte into the corresponding bytes of $GPR[rt]$. Bytes of $GPR[rt]$ that are not loaded are not changed. But if the sign bit (bit 31) is loaded, they are sign-extended to bits 63 to 32.

LWR \$24,1(\$0)



An Address Error exception due to alignment of the effective address does not occur.

LWR and LWL instructions in a pair are used to load 4-byte blocks that do not conform to word alignment.

Exceptions

TLB Refill, TLB Invalid, Address Error

Operation (128-bit bus)

$vAddr \leftarrow \text{sign_extend}(\text{offset}) + GPR[base]_{31:0}$

$(pAddr, \text{uncached}) \leftarrow \text{AddressTranslation}(vAddr, \text{DATA}, \text{LOAD})$

$pAddr \leftarrow pAddr_{(PSIZE-1):0}$

$\text{byte} \leftarrow 0 \mid \mid vAddr_{1:0}$

$\text{word} \leftarrow vAddr_{3:2}$

$\text{memquad} \leftarrow \text{LoadMemory}(\text{uncached}, \text{byte}, pAddr, vAddr, \text{DATA})$

$\text{temp} \leftarrow GPR[rt]_{31:(32-8*\text{byte})} \mid \mid \text{memquad}_{(31+32*\text{word}):((32*\text{word}+8*\text{byte}))}$

if byte = 4 then

$\text{utemp} \leftarrow (\text{temp}_{31})^{32}$ /* If loads bit 31, then sign-extends */

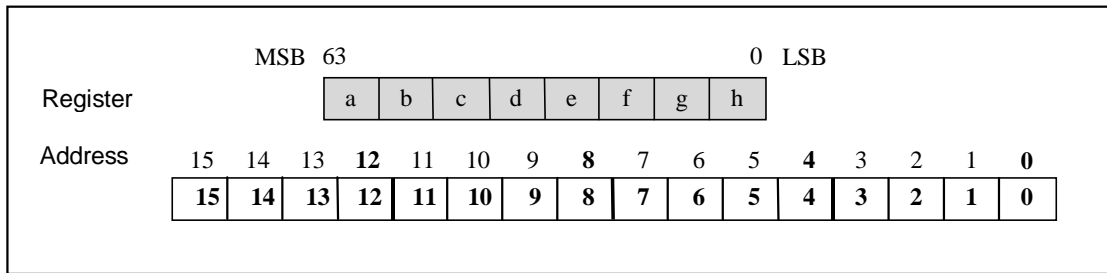
else

$\text{utemp} \leftarrow GPR[rt]_{63:32}$ /* Otherwise, the high-order 4 bytes are not changed */

endif

$GPR[rt]_{63:0} \leftarrow \text{utemp} \mid \mid \text{temp}$

The relation between the low-order 4 bits of the effective address $vAddr$ and bytes that are to be loaded is illustrated below.



vAddr _{3,0}	Contents of registers after instruction (Shaded is unchanged)								Access Type	pAddr _{3,0}
	bit 63			32	31			bit 0		
0	Sign extension of 3←				3	2	1	0	3	0
1	a	b	c	d	e	3	2	1	2	1
2	a	b	c	d	e	f	3	2	1	2
3	a	b	c	d	e	f	g	3	0	3
4	Sign extension of 7←				7	6	5	4	3	4
5	a	b	c	d	e	7	6	5	2	5
6	a	b	c	d	e	f	7	6	1	6
7	a	b	c	d	e	f	g	7	0	7
8	Sign extension of 11←				11	10	9	8	3	8
9	a	b	c	d	e	11	10	9	2	9
10	a	b	c	d	e	f	11	10	1	10
11	a	b	c	d	e	f	g	11	0	11
12	Sign extension of 15←				15	14	13	12	3	12
13	a	b	c	d	e	15	14	13	2	13
14	a	b	c	d	e	f	15	14	1	14
15	a	b	c	d	e	f	g	15	0	15

Programming Notes

In the LWL instruction, the contents of GPR[rt] are referenced. However, since bypassing is performed internally, even when loading the value in GPR[rt] in the preceding instruction, a NOP does not need to be inserted between the instruction and the LWL instruction.

An instruction that treats an unaligned word data as unsigned is not provided.

LWU : Load Word Unsigned

MIPS I

To load a word from memory as an unsigned value.

Operation Code

31	26	25	21	20	16	15	0
LWU 100111		base		rt		offset	
6		5		5		16	

Format

LWU rt, offset (base)

Description

$GPR[rt] \leftarrow \text{memory}[GPR[base] + \text{offset}]$

Adds the offset as a 16-bit signed number to the value of GPR[base] to form the effective address. Loads the word data at the address, zero-extends it and stores it in GPR[rt].

Restrictions

The effective address must conform to word alignment. That is, if the two least-significant bits of the address are non-zero, an Address Error exception occurs.

Exceptions

TLB Refill, TLB Invalid, Address Error

Operation (128-bit bus)

$vAddr \leftarrow \text{sign_extend}(\text{offset}) + GPR[base]_{31..0}$

if ($vAddr_{1..0} \neq 0^2$) then SignalException (AddressError) endif

$(pAddr, \text{uncached}) \leftarrow \text{AddressTranslation}(vAddr, \text{DATA}, \text{LOAD})$

$pAddr \leftarrow pAddr_{(PSIZE-1)..0}$

$\text{memquad} \leftarrow \text{LoadMemory}(\text{uncached}, \text{WORD}, pAddr, vAddr, \text{DATA})$

$\text{byte} \leftarrow vAddr_{3..0}$

$GPR[rt]_{63..0} \leftarrow 0^{32} \mid \mid \text{memquad}_{(31+8*\text{byte})..8*\text{byte}}$

MFHI : Move from HI Register

MIPS I

To move the contents of the HI register to a GPR.

Operation Code

31	26	25	16	15	11	10	6	5	0
SPECIAL	0		rd		0		MFHI		
000000	00 0000 0000				00000		010000		
6	10		5		5		6		

Format

MFHI rd

Description

$\text{GPR}[\text{rd}] \leftarrow \text{HI}$

Stores the contents of HI register, which keeps the results of multiplication and division, in GPR[rd].

Exceptions

None

Operation

$\text{GPR}[\text{rd}]_{63..0} \leftarrow \text{HI}_{63..0}$

Programming Notes

The HI register holds the upper part of the result from multiplication or multiply-accumulate or the remainder from division.

Since an interlock works in EE Core, multiplication and division instructions can be directly followed by an MFHI instruction.

MFLO : Move from LO Register

MIPS I

To move the contents of the LO register to a GPR.

Operation Code

31	26	25	16	15	11	10	6	5	0
SPECIAL	0		rd		0		MFLO		
000000	00 0000 0000				00000		010010		
6	10		5		5		6		

Format

MFLO rd

Description

$rd \leftarrow LO$

Stores the contents of the LO register, which keeps the results of multiplication and division, in GPR[rd].

Exceptions

None

Operation

$GPR[rd]_{63..0} \leftarrow LO_{63..0}$

Programming Notes

The LO register holds the lower part of the result from multiplication or multiply accumulate or the quotient from division.

Since an interlock works in EE Core, multiplication and division instructions can be directly followed by an MFLO instruction.

MOVN : Move Conditional on Not Zero

MIPS IV

To move data between GPRs according to the value of a GPR.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000	rs	rt	rd	0 00000	MOVN 001011						
6	5	5	5	5	6						

Format

MOVN rd, rs, rt

Description

if (GPR[rt] \neq 0) then GPR[rd] \leftarrow GPR[rs]

Checks the value of GPR[rt]. If it is not equal to zero, moves the contents of GPR[rs] to GPR[rd].

Exceptions

None

Operation

if GPR[rt]_{63..0} \neq 0 then

GPR[rd]_{63..0} \leftarrow GPR[rs]_{63..0}

endif

MOVZ : Move Conditional on Zero

MIPS IV

To move data between GPRs according to the value of a GPR.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000	rs	rt	rd	0 00000	MOVZ 001010						
6	5	5	5	5	6						

Format

MOVZ rd, rs, rt

Description

if (GPR[rt] = 0) then GPR[rd] \leftarrow GPR[rs]

Checks the value of GPR[rt]. If the value is equal to zero, moves the contents of GPR[rs] to GPR[rd].

Exceptions

None

Operation

if GPR[rt]_{63..0} = 0 then

GPR[rd]_{63..0} \leftarrow GPR[rs]_{63..0}

endif

MTHI : Move to HI Register

MIPS I

To move the contents a GPR to the HI register.

Operation Code

31	26	25	21	20	6	5	0
SPECIAL	rs		0			MTHI	
000000			000 0000 0000 0000			010001	
6	5		15			6	

Format

MTHI rs

Description

$HI \leftarrow GPR[rs]$

Stores the contents of GPR[rs] to the HI register.

Exceptions

None

Operation

$HI_{63..0} \leftarrow GPR[rs]_{63..0}$

Programming Notes

The HI register holds the upper part of the result from multiplication or multiply-accumulate or the remainder from division.

MTLO : Move to LO Register

MIPS I

To move the contents a GPR to the LO register.

Operation Code

31	26	25	21	20	6	5	0
SPECIAL	rs		0			MTLO	
000000			000 0000 0000 0000			010011	
6	5		15			6	

Format

MTLO rs

Description

$LO \leftarrow GPR[rs]$

Stores the contents of GPR[rs] in the LO register.

Exceptions

None

Operation

$LO_{63..0} \leftarrow GPR[rs]_{63..0}$

Programming Notes

The LO register holds the lower part of the result from multiplication or multiply-accumulate or the quotient from division.

MULT : Multiply Word

MIPS I

To multiply 32-bit signed integers.

Operation Code

31	26	25	21	20	16	15		6	5	0
SPECIAL		rs		rt		0			MULT	
000000						00 0000 0000			011000	
6		5		5		10			6	

Format

MULT rs, rt

Description

$(LO, HI) \leftarrow GPR[rs] \times GPR[rt]$

Multiplies the 32-bit value in GPR[rt] by the 32-bit value in GPR[rs] as signed integer values. The low-order 32 bits and the high-order 32 bits of the 64-bit result are stored in the LO and HI registers respectively.

Restrictions

If GPR[rt] and GPR[rs] are not sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Exceptions

None. No arithmetic exception occurs.

Operation

if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif

$prod \leftarrow GPR[rs]_{31..0} \times GPR[rt]_{31..0}$

$LO_{63..0} \leftarrow (prod_{31})^{32} \mid \mid prod_{31..0}$

$HI_{63..0} \leftarrow (prod_{63})^{32} \mid \mid prod_{63..32}$

Programming Notes

In the EE Core, the integer multiply operation proceeds asynchronously. An attempt to read the contents of the LO or HI register before the multiply operation finishes will result in an interlock. Other CPU instructions can execute without delay. Therefore, scheduling the multiply operation appropriately will improve performance.

Even when the result of the multiply operation overflows, the Overflow exception does not occur. If Overflow must be detected, an explicit check is necessary.

In the EE Core, the MULT instruction has been extended to three-operand instruction that can store the result of operation in a GPR as well. See "3. EE Core-Specific Instruction Set " about the usage as a three-operand instruction.

MULTU : Multiply Unsigned Word

MIPS I

To multiply 32-bit unsigned integers.

Operation Code

31	26	25	21	20	16	15	6	5	0
SPECIAL 000000		rs		rt		0 00 0000 0000		MULTU 011001	
6		5		5		10		6	

Format

MULTU rs, rt

Description

$(LO, HI) \leftarrow GPR[rs] \times GPR[rt]$

Multiplies the 32-bit value in GPR[rt] by the 32-bit value in GPR[rs] as unsigned integer values. The low-order 32-bit and the high-order 32-bit of a 64-bit result are stored in the LO and HI registers respectively.

Restrictions

GPR[rt] and GPR[rs] are not sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Exceptions

None. No arithmetic exception occurs.

Operation

if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif

$prod \leftarrow (0 \mid \mid GPR[rs]_{31..0}) \times (0 \mid \mid GPR[rt]_{31..0})$

$LO_{63..0} \leftarrow (prod_{31})^{32} \mid \mid prod_{31..0}$

$HI_{63..0} \leftarrow (prod_{63})^{32} \mid \mid prod_{63..32}$

Programming Notes

In the EE Core, the integer multiply operation proceeds asynchronously. An attempt to read the contents of the LO or HI register before the multiply operation finishes will result in an interlock. Other CPU instructions can execute without delay. Therefore, scheduling the multiply operation appropriately will improve performance.

Even when the result of a multiply operation overflows, an Overflow exception does not occur. If Overflow must be detected, an explicit check is necessary.

In EE Core, the MULTU instruction has been extended to be a three-operand instruction that can store the result of operation in a GPR as well. See "3. EE Core-Specific Instruction Set" about the usage as a three-operand instruction.

NOR : Not Or

MIPS I

To perform a bitwise logical NOT OR.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000	rs	rt	rd	0 00000	NOR 100111						
6	5	5	5	5	6						

Format

NOR rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] \text{ NOR } GPR[rt]$

Performs a bitwise logical NOR between the contents of GPR[rs] and GPR[rt]. The result is stored in GPR[rd].

The truth table value for NOR is as follows;

X	Y	X NOR Y
0	0	1
0	1	0
1	0	0
1	1	0

Exceptions

None

Operation

$GPR[rd]_{63..0} \leftarrow GPR[rs]_{63..0} \text{ NOR } GPR[rt]_{63..0}$

OR : or

MIPS I

To perform a bitwise logical OR.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000	rs	rt	rd	0 00000	OR 100101						
6	5	5	5	5	6						

Format

OR rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] \text{ OR } GPR[rt]$

Performs a bitwise logical OR between the contents of GPR[rs] and GPR[rt]. The result is stored in GPR[rd].

The truth table value for OR is as follows;

X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

Exceptions

None

Operation

$GPR[rd]_{63..0} \leftarrow GPR[rs]_{63..0} \text{ OR } GPR[rt]_{63..0}$

ORI : Or immediate

MIPS I

To perform a bitwise logical OR between a constant and a GPR.

Operation Code

31	26	25	21	20	16	15	0
ORI						rs	
001101						rt	
						immediate	
6						5	
						5	
						16	

Format

ORI rt, rs, immediate

Description

$GPR[rt] \leftarrow GPR[rs] \text{ OR immediate}$

Performs a bitwise logical OR between the sign-extended value of the 16-bit immediate field and the contents of GPR[rs]. The result is stored in GPR[rt].

The truth table value for OR is as follows;

X	Y	X OR Y
0	0	0
0	1	0
1	0	0
1	1	1

Exceptions

None

Operation

$GPR[rt]_{63..0} \leftarrow (0^{48} \mid \mid \text{immediate}) \text{ OR } GPR[rs]_{63..0}$

PREF : Prefetch

MIPS IV

To prefetch data from memory.

Operation Code

31	26	25	21	20	16	15	0
PREF 110011		base		hint		offset	
6		5		5		16	

Format

PREF hint, offset(base)

Description

prefetch_memory(GPR[base]+offset)

The data at the effective address, obtained by adding the offset as a signed integer to the value of GPR[base], is read into the cache, if possible. It does not affect the meaning of the program but can help improve its performance.

The value of hint is to specify the details of the Prefetch operation as defined in the following table.

Value	Name	Prefetch Operation
0	load	Read into cache for loading data.
1-31	(Reserved)	(Same in case specifies 0)

Addressing-related exceptions do not occur. If an exception should be generated, it is ignored and Prefetch does not take place. However, operations such as a writeback of a dirty cache line might take place.

Exceptions

None

Operation

$vAddr \leftarrow GPR[base] + \text{sign_extend}(\text{offset})$

$(pAddr, \text{uncached}) \leftarrow \text{AddressTranslation}(vAddr, \text{DATA}, \text{LOAD})$

Prefetch(uncached, pAddr, vAddr, DATA, hint)

Programming Notes

Prefetch does not take place on uncached memory access locations. Prefetch, from memory locations not present in the TLB, is not allowed. Memory pages that have not been accessed recently may not present in the TLB, so prefetch may not be effective. In addition, prefetch may not take place when the bus is used for read operations such as data cache miss, uncached load, and load to the uncached accelerated buffer.

SB : Store Byte

MIPS I

To store a byte in a GPR to memory.

Operation Code

31	26	25	21	20	16	15	0
SB		base			rt		offset
101000							
6		5			5		16

Format

SB rt, offset (base)

Description

memory [GPR[base] + offset] \leftarrow GPR[rt]

Stores the least-significant byte of GPR[rt] in memory at the effective address obtained by adding the offset as a 16-bit signed integer to the value of GPR[base].

Exceptions

TLB Refill, TLB Invalid, TLB Modified, Address Error

Operation (128-bit bus)

vAddr \leftarrow sign_extend (offset) + GPR[base]_{31..0}

(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA, STORE)

pAddr \leftarrow pAddr_{(PSIZE-1)..0}

byte \leftarrow vAddr_{3..0}

dataquad \leftarrow GPR[rt]_{(127-8*byte)..0} | 0^{8*byte}

StoreMemory (uncached, BYTE, dataquad, pAddr, vAddr, DATA)

SD : Store Doubleword

MIPS III

To store a doubleword in a GPR to memory.

Operation Code

31	26	25	21	20	16	15	0
SD 111111		base		rt		offset	
6		5		5		16	

Format

SD rt, offset (base)

Description

memory [GPR[base] + offset] \leftarrow GPR[rt]

Stores the 64-bit value of GPR[rt] in memory at the effective address obtained by adding the offset as a 16-bit signed integer to the value of GPR[base].

Restrictions

The effective address must conform to doubleword alignment. That is, if any of the three least-significant bits of the address are non-zero, an Address Error exception occurs.

Exceptions

TLB Refill, TLB Invalid, TLB Modified, Address Error

Operation (128-bit bus)

vAddr \leftarrow sign_extend (offset) + GPR[base]

if (vAddr_{2..0}) \neq 0³ then SignalException (AddressError) endif

(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA, STORE)

pAddr \leftarrow pAddr_{(PSIZE-1)..₀}

byte \leftarrow vAddr_{3..0}

dataquad \leftarrow GPR[rt]_{(127-8*byte)..₀} | 0^{8*byte}

StoreMemory (uncached, DOUBLEWORD, dataquad, pAddr, vAddr, DATA)

SDL : Store Doubleword Left

MIPS III

To load the upper part of a doubleword to an unaligned memory address.

Operation Code

31	26	25	21	20	16	15	0
SDL						base	
101100						rt	
						offset	
6						5	
						5	
						16	

Format

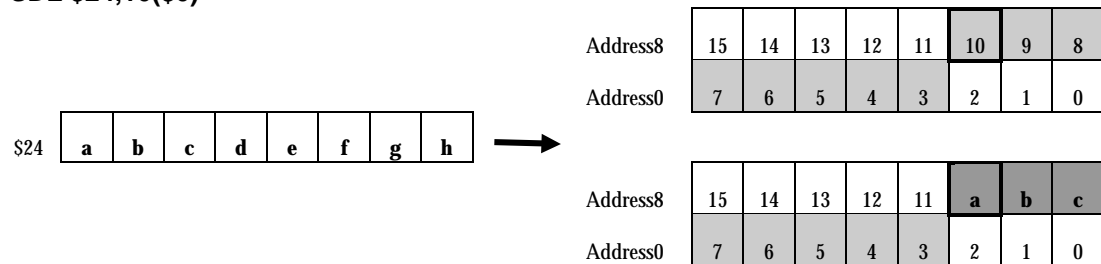
SDL rt, offset (base)

Description

memory [GPR[base] + offset] \leftarrow GPR[rt]

Adds the offset as a 16-bit signed number to the value of GPR[base] to form the effective address. Stores the high-order bytes of GPR[rt] in the lower part of the aligned doubleword starting with the effective address.

SDL \$24,10(\$0)



Address Error exceptions due to the alignment of the effective address do not occur.

SDL and SDR instructions in a pair are used to store doubleword data in an 8-byte block that does not conform to doubleword alignment.

Exceptions

TLB Refill, TLB Invalid, TLB Modified exception, Address Error

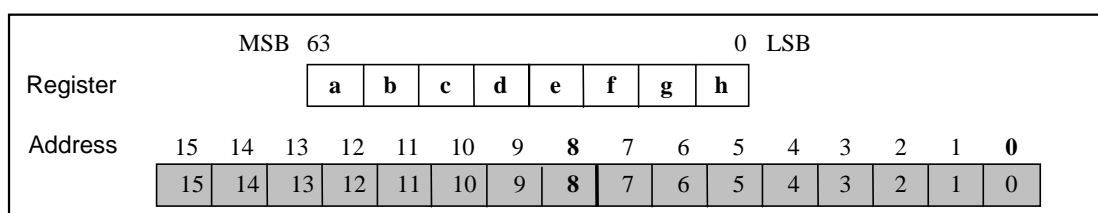
Operation (128-bit bus)

```

vAddr  $\leftarrow$  sign_extend(offset) + GPR[base]31..0
(pAddr, uncached)  $\leftarrow$  AddressTranslation(vAddr, DATA, STORE)
pAddr  $\leftarrow$  pAddr(PSIZE-1)..3 | 03
byte  $\leftarrow$  0 | vAddr2..0
if (vAddr3 = 0) then
    dataquad  $\leftarrow$  064 | 0(56-8*byte) | GPR[rt]63..(56-8*byte)
else
    dataquad  $\leftarrow$  0(56-8*byte) | GPR[rt]63..(56-8*byte) | 064
endif
StoreMemory(uncached, byte, dataquad, pAddr, vAddr, DATA)

```

The relation between the low-order 4 bits of the effective address vAddr and bytes that are to be stored is illustrated below.



Vaddr _{3..0}	Contents of registers after instruction (Shaded is unchanged)																Access Type	pAddr _{3..0}
	bit 63																	
0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	a	0	0
1	15	14	13	12	11	10	9	8	7	6	5	4	3	2	a	b	1	0
2	15	14	13	12	11	10	9	8	7	6	5	4	3	a	b	c	2	0
3	15	14	13	12	11	10	9	8	7	6	5	4	a	b	c	d	3	0
4	15	14	13	12	11	10	9	8	7	6	5	a	b	c	d	e	4	0
5	15	14	13	12	11	10	9	8	7	6	a	b	c	d	e	f	5	0
6	15	14	13	12	11	10	9	8	7	a	b	c	d	e	f	g	6	0
7	15	14	13	12	11	10	9	8	a	b	c	d	e	f	g	h	7	0
8	15	14	13	12	11	10	9	a	7	6	5	4	3	2	1	0	8	8
9	15	14	13	12	11	10	a	b	7	6	5	4	3	2	1	0	9	8
10	15	14	13	12	11	a	b	c	7	6	5	4	3	2	1	0	10	8
11	15	14	13	12	a	b	c	d	7	6	5	4	3	2	1	0	11	8
12	15	14	13	a	b	c	d	e	7	6	5	4	3	2	1	0	12	8
13	15	14	a	b	c	d	e	f	7	6	5	4	3	2	1	0	13	8
14	15	a	b	c	d	e	f	g	7	6	5	4	3	2	1	0	14	8
15	a	b	c	d	e	f	g	h	7	6	5	4	3	2	1	0	15	8

SDR : Store Doubleword Right

MIPS III

To load the lower part of a doubleword to an unaligned memory address.

Operation Code

31	26	25	21	20	16	15	0
SDR	base	rt	offset				
101101							
6	5	5	16				

Format

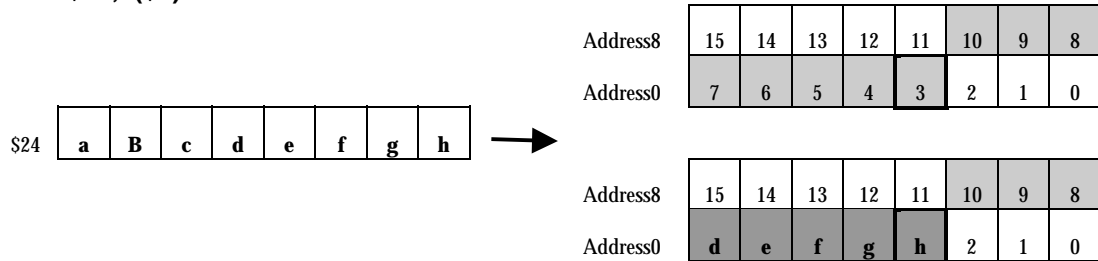
SDR rt, offset (base)

Description

memory [GPR[base] + offset] \leftarrow GPR[rt]

Adds the offset as a 16-bit signed number to the value of GPR[base] to form the effective address. Stores the lower-order bytes of GPR[rt] in the upper part of the aligned doubleword starting with the effective address.

SDL \$24,3(\$0)



Address Error exceptions due to alignment of the effective address do not occur.

SDR and SDL instructions in a pair are used to store the doubleword data in an 8-byte block that does not conform to doubleword alignment.

Exceptions

TLB Refill, TLB Invalid, TLB Modified exception, Address Error

Operation (128-bit bus)

$vAddr \leftarrow \text{sign_extend}(\text{offset}) + \text{GPR}[\text{base}]_{31..0}$

$(pAddr, \text{uncached}) \leftarrow \text{AddressTranslation}(vAddr, \text{DATA}, \text{STORE})$

$pAddr \leftarrow pAddr_{(PSIZE-1)..3} \mid 0^3$

$\text{byte} \leftarrow vAddr_{2..0}$

if($vAddr_3 = 0$) then

$\text{dataquad} \leftarrow 0^{64} \mid \text{GPR}[\text{rt}]_{(63-8*\text{byte})..0} \mid 0^{8*\text{byte}}$

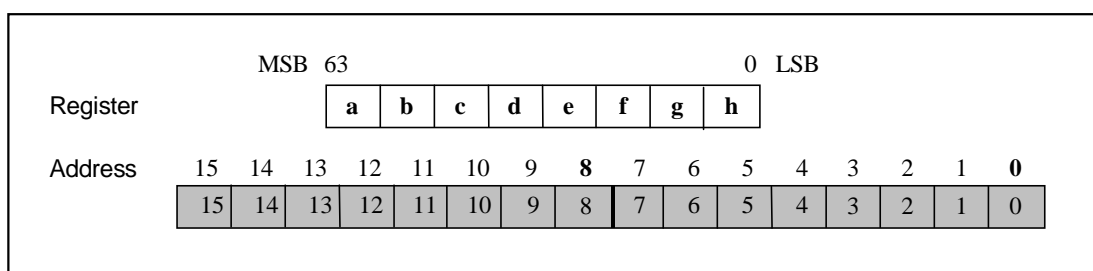
else

$\text{dataquad} \leftarrow \text{GPR}[\text{rt}]_{(63-8*\text{byte})..0} \mid 0^{8*\text{byte}} \mid 0^{64}$

endif

StoreMemory(uncached, DOUBLEWORD-byte, dataquad, pAddr, vAddr, DATA)

The relation between the low-order 4 bits of the effective address vAddr and bytes that are to be stored is illustrated below.



vAddr _{3..0}	Contents of registers after instruction (Shaded is unchanged)																Access Type	pAddr _{3..0}
	bit 63																	
0	15	14	13	12	11	10	9	8	a	b	c	d	e	f	g	h	0	8
1	15	14	13	12	11	10	9	8	b	c	d	e	f	g	h	0	1	8
2	15	14	13	12	11	10	9	8	c	d	e	f	g	h	1	0	2	8
3	15	14	13	12	11	10	9	8	d	e	f	g	h	2	1	0	3	8
4	15	14	13	12	11	10	9	8	e	f	g	h	3	2	1	0	4	8
5	15	14	13	12	11	10	9	8	f	g	h	4	3	2	1	0	5	8
6	15	14	13	12	11	10	9	8	g	h	5	4	3	2	1	0	6	8
7	15	14	13	12	11	10	9	8	h	6	5	4	3	2	1	0	7	8
8	a	b	c	d	e	f	g	h	7	6	5	4	3	2	1	0	8	0
9	b	c	d	e	f	g	h	8	7	6	5	4	3	2	1	0	9	0
10	c	d	e	f	g	h	9	8	7	6	5	4	3	2	1	0	10	0
11	d	e	f	g	h	10	9	8	7	6	5	4	3	2	1	0	11	0
12	e	f	g	h	11	10	9	8	7	6	5	4	3	2	1	0	12	0
13	f	g	h	12	11	10	9	8	7	6	5	4	3	2	1	0	13	0
14	g	h	13	12	11	10	9	8	7	6	5	4	3	2	1	0	14	0
15	h	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	0

SH : Store Halfword

MIPS I

To store a halfword in memory.

Operation Code

31	26	25	21	20	16	15	0
SH		base		rt		offset	
101001							
6		5		5		16	

Format

SH rt, offset (base)

Description

memory [GPR[base] + offset] \leftarrow GPR[rt]

Stores the least-significant 16-bit values of GPR[rt] in memory at the effective address obtained by adding the offset as a 16-bit signed integer to the value of GPR[base].

Restrictions

The effective address must conform to halfword alignment. If the least-significant bit of the address is not zero, an Address Error exception occurs.

Exceptions

TLB Refill, TLB Modified, Address Error

Operation (128-bit bus)

```

vAddr  $\leftarrow$  sign_extend(offset) + GPR[base]31..0
if (vAddr0)  $\neq$  0 then SignalException (AddressError) endif
(pAddr, uncached)  $\leftarrow$  AddressTranslation (vAddr, DATA, STORE)
pAddr  $\leftarrow$  pAddr(PSIZE-1)..0
byte  $\leftarrow$  vAddr3..0
dataquad  $\leftarrow$  GPR[rt](127-8*byte)..0 || 08*byte
StoreMemory (uncached, HALFWORD, dataquad, pAddr, vAddr, DATA)

```

SLL : Shift Word Left Logical

MIPS I

To left shift a word. The shift amount is a fixed value (0-31 bits) specified by sa.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0	
SPECIAL 000000			0 00000		rt		rd		sa		SLL 000000	
6			5		5		5		5		6	

Format

SLL rd, rt, sa

Description

$GPR[rd] \leftarrow GPR[rt] \ll sa$

Shifts the 32-bit data in GPR[rt] left by the bit count specified by sa, inserting zeros into the emptied bits.

The 32-bit result is sign-extended into 64-bit destination register and stored in GPR[rd].

Restrictions

None. Unlike nearly all other word operations, the input operand does not have to be a sign-extended 32-bit value (bits 63..32 equal).

Exceptions

None

Operation

$s \leftarrow sa$

$temp \leftarrow GPR[rt]_{(31-s)..0} \parallel 0^s$

$GPR[rd]_{63..0} \leftarrow sign_extend(temp_{31..0})$

Programming Notes

If sa is zero, GPR[rt] is truncated to 32-bits and sign extended to 64-bits to store in GPR[rd].

SLLV : Shift Word Left Logical Variable

MIPS I

To left shift a word. The shift amount is specified by a GPR (0-31 bits).

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000	rs	rt	rd	0 00000	SLLV 000100						
6	5	5	5	5	6						

Format

SLLV rd, rt, rs

Description

$GPR[rd] \leftarrow GPR[rt] \ll GPR[rs]$

Shifts the lower 32-bit of GPR[rt] left by the bit count specified by the low-order five bits of GPR[rs], inserting zeros into the emptied bits. The 32-bit result is sign-extended and stored in GPR[rd].

Restrictions

None. Unlike nearly all other word operations, the input operand does not have to be a sign-extended 32-bit value (bits 63..32 equal).

Exceptions

None

Operation

$s \leftarrow GPR[rs]_{4..0}$

$temp \leftarrow GPR[rt]_{(31-s)..0} \mid 0^s$

$GPR[rd]_{63..0} \leftarrow sign_extend(temp_{31..0})$

SLT : Set on Less Than

MIPS I

To compare the value of GPRs as signed integers.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000						rs		rt		rd	
000000						00000		SLT 101010			
6						5		5		5	

Format

SLT rd, rs, rt

Description

$GPR[rd] \leftarrow (GPR[rs] < GPR[rt])$

Compares the contents of GPR[rs] and GPR[rt] as signed integers. If GPR[rs] is less than GPR[rt], stores 1 in GPR[rd]. Otherwise, stores 0 in GPR[rd].

Exceptions

None. An Integer Overflow exception does not occur due to the arithmetic comparison.

Operation

```

if GPR[rs]63..0 < GPR[rt]63..0 then
    GPR[rd]63..0 ← 0GPRLEN-1 || 1
else
    GPR[rd]63..0 ← 0GPRLEN
endif

```

SLTI : Set on Less Than Immediate

MIPS I

To compare a GPR with a constant as signed integers.

Operation Code

31	26	25	21	20	16	15	0
SLTI	rs	rt	immediate				
001010							
6	5	5	16				

Format

SLTI rt, rs, immediate

Description

$GPR[rt] \leftarrow (GPR[rs] < \text{immediate})$

Compares the contents of GPR[rs] with immediate value as signed integers. If GPR[rs] is less than immediate, stores 1 in GPR[rd]. Otherwise, stores 0 in GPR[rd].

Exceptions

None. An Integer Overflow exception does not occur due to the arithmetic comparison.

Operation

if $GPR[rs]_{63:0} < \text{sign_extend}(\text{immediate})$ then

$GPR[rd]_{63:0} \leftarrow 0^{GPREN-1} \mid 1$

else

$GPR[rd]_{63:0} \leftarrow 0^{GPREN}$

endif

SLTIU : Set on Less Than Immediate Unsigned

MIPS I

To compare a GPR with a constant unsigned integer.

Operation Code

31	26	25	21	20	16	15	0
SLTIU						rs	
001011						rt	
						immediate	
6						5	
						5	
						16	

Format

SLTIU rt, rs, immediate

Description

$GPR[rt] \leftarrow (GPR[rs] < \text{immediate})$

Compares the contents of GPR[rs] with the sign-extended immediate value as unsigned integers. If GPR[rs] is less than immediate, stores 1 in GPR[rd]. Otherwise, stores 0 in GPR[rd].

Exceptions

None. An Integer Overflow exception does not occur due to the arithmetic comparison.

Operation

if $(0 \mid\mid GPR[rs]_{63..0}) < (0 \mid\mid \text{sign_extend}(\text{immediate}))$ then

$GPR[rd]_{63..0} \leftarrow 0^{GPRLEN-1} \mid\mid 1$

else

$GPR[rd]_{63..0} \leftarrow 0^{GPRLEN}$

endif

Programming Notes

Because the 16-bit immediate is sign-extended before comparison, the range of numeric values that the immediate represents is not sequential, but split into two areas; around the smallest and largest 64-bit unsigned integers. That is [0,32767] and [max_unsigned-32767, max_unsigned], respectively.

SLTU : Set on Less Than Unsigned

MIPS I

To compare the value of GPRs as unsigned integers.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000	rs	rt	rd	0 00000	SLTU 101011						
6	5	5	5	5	6						

Format

SLTU rd, rs, rt

Description

$GPR[rd] \leftarrow (GPR[rs] < GPR[rt])$

Compares the contents of GPR[rs] and GPR[rt] as unsigned integers. If GPR[rs] is less than GPR[rt], stores 1 in GPR[rd]. Otherwise, stores 0 in GPR[rd].

Exceptions

None. An Integer Overflow exception due to the arithmetic comparison does not occur.

Operation

if $(0 \mid \mid GPR[rs]_{63..0}) < (0 \mid \mid GPR[rt]_{63..0})$ then

$GPR[rd]_{63..0} \leftarrow 0^{GPREN-1} \mid 1$

else

$GPR[rd]_{63..0} \leftarrow 0^{GPREN}$

endif

SRA : Shift Word Right Arithmetic

MIPS I

To arithmetic right shift a word. The shift amount is a fixed value (0-31 bits) specified by sa.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000	0 00000	rt	rd	sa	SRA 000011						
6	5	5	5	5	6						

Format

SRA rd, rt sa

Description

$GPR[rd] \leftarrow GPR[rt] \gg sa$ (arithmetic)

Shifts the lower 32 bits of GPR[rt] right by the bit count specified by sa, duplicating the sign bit (bit 31) into the emptied bits. The 32-bit result is sign-extended and stored in GPR[rd].

Restrictions

If GPR[rt] is not a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is undefined.

Exceptions

None

Operation

if (NotWordValue (GPR[rt]_{63..0})) then UndefinedResult () endif

$s \leftarrow sa$

$temp \leftarrow (GPR[rt]_{31})^s \parallel GPR[rt]_{31..s}$

$GPR[rd]_{63..0} \leftarrow sign_extend(temp_{31..0})$

SRAV : Shift Word Right Arithmetic Variable

MIPS I

To arithmetic right shift a word. The shift amount is specified by a GPR (0-31 bits).

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000		rs		rt		rd		0 00000		SRAV 000111	
6		5		5		5		5		6	

Format

SRAV rd, rt, rs

Description

$GPR[rd] \leftarrow GPR[rt] \gg GPR[rs]$ (arithmetic)

Shifts the lower 32 bits of GPR[rt] right by the bit count specified by the low-order five bits of GPR[rs], inserting the sign bit (bit 31) into the emptied bits. The 32-bit result is sign-extended and stored in GPR[rd].

Restrictions

If the value of GPR[rt] is not a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is undefined.

Exceptions

None

Operation

```
if (NotWordValue (GPR[rt] 63..0 )) then UndefinedResult () endif
s ← GPR[rs] 4..0
temp ← (GPR[rt] 31s | GPR[rt] 31..s
GPR[rd] 63..0 ← sign_extend (temp 31..0)
```

SRL : Shift Word Right Logical

MIPS I

To logical right shift a word. The shift amount is a fixed value (0-31 bits) specified by sa.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000	0 00000	rt	rd	sa	SRL 000010						
6	5	5	5	5	6						

Format

SRL rd, rt, sa

Description

$GPR[rd] \leftarrow GPR[rt] \gg sa$ (logical)

Shifts the low-order 32 bits of GPR[rt] right by the bit count specified by sa, inserting zeros into the emptied bits. The 32-bit result is sign-extended and stored in GPR[rd]

Restrictions

If the value of GPR[rt] is not a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is undefined.

Exceptions

None

Operation

if (NotWordValue (GPR[rt]_{63..0})) then UndefinedResult () endif

$s \leftarrow sa$

$temp \leftarrow 0^s \parallel GPR[rt]_{31..s}$

$GPR[rd]_{63..0} \leftarrow sign_extend(temp_{31..0})$

SRLV : Shift Word Right Logical Variable

MIPS I

To logical right shift a word. The shift amount is specified by a GPR (0-31 bits).

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000	rs				rt		rd		0 00000		SRLV 000110
6	5				5		5		5		6

Format

SRLV rd, rt, rs

Description

$GPR[rd] \leftarrow GPR[rt] \gg GPR[rs]$ (logical)

Shifts the low-order 32 bits of GPR[rt] right by the bit count specified by the low-order five bits of GPR[rs], inserting zeros into the emptied bits. The 32-bit result is sign-extended and stored in GPR[rd].

Restrictions

If the value of GPR[rt] is not a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is undefined.

Exceptions

None

Operation

```
if (NotWordValue (GPR[rt] 63..0)) then UndefinedResult () endif
s ← GPR[rs] 4..0
temp ← 0s || GPR[rt] 31..s
GPR[rd] 63..0 ← sign_extend (temp 31..0)
```

SUB : Subtract Word

MIPS I

To subtract 32-bit integers. Traps if overflow occurs.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000	rs	rt	rd	0 00000	SUB 100010						
6	5	5	5	5	6						

Format

SUB rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] - GPR[rt]$

Subtracts the low-order 32-bit value in GPR[rt] from the lower 32-bit value in GPR[rs]. The 32-bit result is sign-extended and stored in GPR[rd]. If the subtraction results in 32-bit 2's complement arithmetic overflow, then the contents of GPR[rd] are not changed and an Integer Overflow exception occurs.

Restrictions

If the values of GPR[rs] and GPR[rt] are not sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Exceptions

Integer Overflow

Operation

```

if (NotWordValue (GPR[rs]63..0) or NotWordValue (GPR[rt]63..0)) then UndefinedResult () endif
temp ← GPR[rs]63..0 — GPR[rt]63..0
if (32_bit_arithmetic_overflow) then
    SignalException (IntegerOverflow)
else
    GPR[rd]63..0 ← sign_extend (temp31..0)
endif

```

Programming Notes

SUBU performs the same arithmetic operation, but does not trap on overflow.

SUBU : Subtract Unsigned Word

MIPS I

To subtract 32-bit integers and ignore overflow.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000	rs				rt				0 00000		SUBU 100011
6	5				5				5		6

Format

SUBU rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] - GPR[rt]$

Subtracts the low-order 32-bit value in GPR[rt] from the low-order 32-bit value in GPR[rs]. The 32-bit result is sign-extended and stored in GPR[rd]. If overflow occurs, ignores it.

Restrictions

If the values of GPR[rs] and GPR[rt] are not sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Exceptions

None

Operation

```
if (NotWordValue (GPR[rs] 63..0) or NotWordValue (GPR[rt] 63..0)) then UndefinedResult () endif
temp ← GPR[rs] 63..0 — GPR[rt] 63..0
GPR[rd] 63..0 ← sign_extend (temp 31..0)
```

Programming Notes

This instruction is not an unsigned operation in the strict sense, and performs 32-bit modulo arithmetic that ignores overflow. It is appropriate for integer arithmetic that ignores overflow such as address or C language arithmetic.

SW : Store Word

MIPS I

To store a word data in memory.

Operation Code

31	26	25	21	20	16	15	0
SW 101011		base		rt		offset	
6		5		5		16	

Format

SW rt, offset (base)

Description

memory [GPR[base] + offset] \leftarrow GPR[rt]

Stores the low-order 32-bit value of GPR[rt] in memory at the effective address obtained by adding the offset as a 16-bit signed integer to the contents of GPR[base].

Restrictions

The effective address must conform to word alignment. If both of the least-significant bits of the address are non-zero, an Address Error exception occurs.

Exceptions

TLB Refill, TLB Invalid, TLB Modified, Address Error

Operation (128-bit bus)

vAddr \leftarrow sign_extend(offset) + GPR[base]

if (vAddr_{1..0} \neq 0² then SignalException (AddressError) endif

(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA, STORE)

pAddr \leftarrow pAddr_{(PSIZE-1)..0}

byte \leftarrow vAddr_{3..0}

dataquad \leftarrow GPR[rt]_{(127-8*byte)..0} | 0^{8*byte}

StoreMemory (uncached, WORD, dataquad, pAddr, vAddr, DATA)

SWL : Store Word Left

MIPS I

To store the upper part of a word at an unaligned memory address.

Operation Code

31	26	25	21	20	16	15	0
SWL						base	
101010						rt	
						offset	
6						5	
						5	
						16	

Format

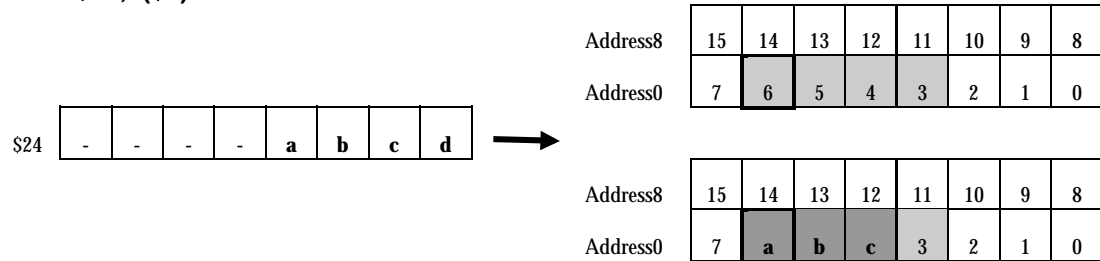
SWL rt, offset (base)

Description

memory [GPR[base] + offset] \leftarrow GPR[rt]

Adds the offset as a 16-bit signed number to the value of GPR[base] to form the effective address. Stores the high-order bytes of GPR[rt] in the lower part of the address in an aligned word including the address.

SWL \$24,6(\$0)



An Address Error exception due to alignment of the effective address does not occur.

SWL and SWR instructions in a pair are used to store the word data in a 4-byte block that does not conform to word alignment.

Exceptions

TLB Refill, TLB Invalid, TLB Modified, Address Error

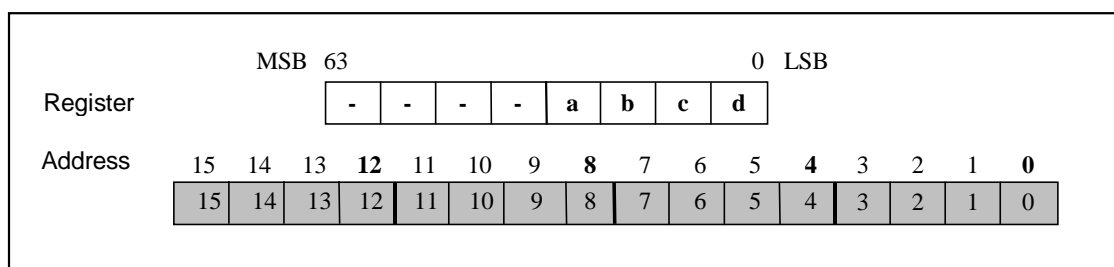
Operation

```

vAddr  $\leftarrow$  sign_extend (offset) + GPR[base]
(pAddr, uncached)  $\leftarrow$  AddressTranslation (vAddr, DATA, STORE)
pAddr  $\leftarrow$  pAddr(PSIZE-1)..2 | 02
byte  $\leftarrow$  vAddr1..0
if (vAddr3..2 = 002) then
    dataquad  $\leftarrow$  096 | 0(24-8*byte) | GPR[rt]31..(24-8*byte)
elseif (vAddr3..2 = 012) then
    dataquad  $\leftarrow$  064 | 0(24-8*byte) | GPR[rt]31..(24-8*byte) | 032
elseif (vAddr3..2 = 102) then
    dataquad  $\leftarrow$  032 | 0(24-8*byte) | GPR[rt]31..(24-8*byte) | 032
elseif (vAddr3..2 = 112) then
    dataquad  $\leftarrow$  0(24-8*byte) | GPR[rt]31..(24-8*byte) | 064
endif
StoreMemory (uncached, byte, dataquad, pAddr, vAddr, DATA)

```

The relation between the low-order 4 bits of the effective address vAddr and bytes that are to be stored is illustrated below.



vAddr _{3..0}	Contents of registers after instruction (Shaded is unchanged)																Access Type	pAddr _{3..0}
	bit 63															bit 0		
0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	a	0	0
1	15	14	13	12	11	10	9	8	7	6	5	4	3	2	a	b	1	0
2	15	14	13	12	11	10	9	8	7	6	5	4	3	a	b	c	2	0
3	15	14	13	12	11	10	9	8	7	6	5	4	a	b	c	d	3	0
4	15	14	13	12	11	10	9	8	7	6	5	a	3	2	1	0	0	4
5	15	14	13	12	11	10	9	8	7	6	a	b	3	2	1	0	1	4
6	15	14	13	12	11	10	9	8	7	a	b	c	3	2	1	0	2	4
7	15	14	13	12	11	10	9	8	a	b	c	d	3	2	1	0	3	4
8	15	14	13	12	11	10	9	a	7	6	5	4	3	2	1	0	0	8
9	15	14	13	12	11	10	a	b	7	6	5	4	3	2	1	0	1	8
10	15	14	13	12	11	a	b	c	7	6	5	4	3	2	1	0	2	8
11	15	14	13	12	a	b	c	d	7	6	5	4	3	2	1	0	3	8
12	15	14	13	a	11	10	9	8	7	6	5	4	3	2	1	0	0	12
13	15	14	a	b	11	10	9	8	7	6	5	4	3	2	1	0	1	12
14	15	a	b	c	11	10	9	8	7	6	5	4	3	2	1	0	2	12
15	a	b	c	d	11	10	9	8	7	6	5	4	3	2	1	0	3	12

SWR : Store Word Right

MIPS I

To store the lower part of a word at an unaligned memory address.

Operation Code

31	26	25	21	20	16	15	0
SWR	base	rt	offset				
101110							
6	5	5	16				

Format

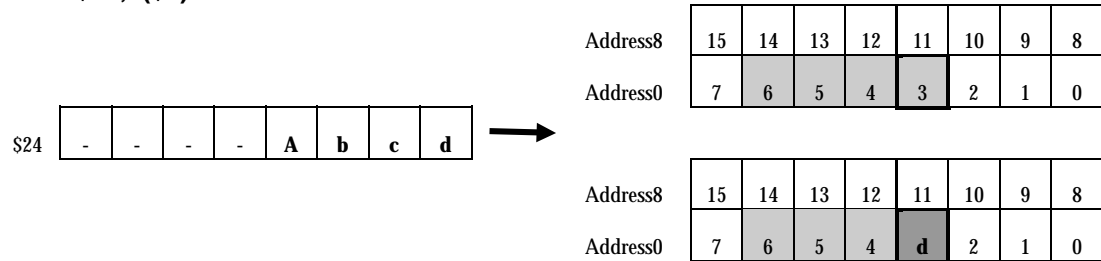
SWR rt, offset (base)

Description

memory [GPR[base] + offset] \leftarrow GPR[rt]

Adds the offset as a 16-bit signed number to the value of GPR[base] to form the effective address. Stores the low-order bytes of GPR[rt] in the upper part of the address in an aligned word including the address.

SWL \$24,3(\$0)



An Address Error exception due to alignment of the effective address does not occur.

SWR and SWL instructions in a pair are used to store the word in a 4-byte block that does not conform to word alignment.

Exceptions

TLB Refill, TLB Invalid, TLB Modified, Address Error

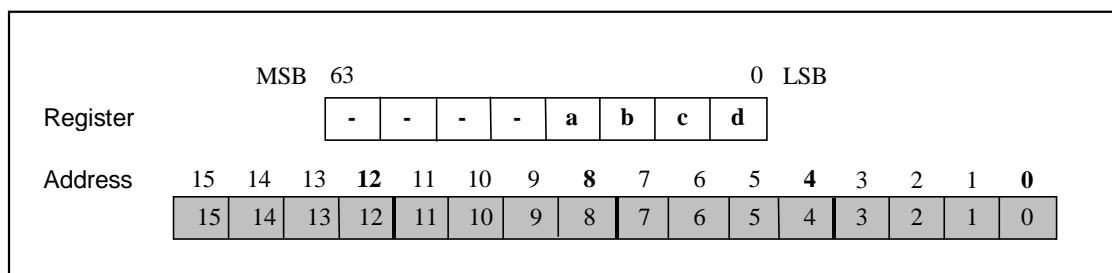
Operation

```

vAddr  $\leftarrow$  sign_extend (offset) + GPR[base]
(pAddr, uncached)  $\leftarrow$  AddressTranslation (vAddr, DATA, STORE)
pAddr  $\leftarrow$  pAddr(PSIZE-1)..2 | 02
byte  $\leftarrow$  vAddr1..0
if (vAddr3..2 = 002) then
    dataquad  $\leftarrow$  096 | GPR[rt](31-8*byte)..0 | 08*byte
else if (vAddr3..2 = 012) then
    dataquad  $\leftarrow$  064 | GPR[rt](31-8*byte)..0 | 08*byte | 032
else if (vAddr3..2 = 102) then
    dataquad  $\leftarrow$  032 | GPR[rt](31-8*byte)..0 | 08*byte | 064
else if (vAddr3..2 = 112) then
    dataquad  $\leftarrow$  GPR[rt](31-8*byte)..0 | 08*byte | 096
endif
StoreMemory (uncached, WORD-byte, dataquad, pAddr, vAddr, DATA)

```

The relation between the low-order 4 bits of the effective address vAddr and bytes that are to be stored is illustrated below.



vAddr _{3..0}	Contents of registers after instruction (Shaded is unchanged)																Access Type	pAddr _{3..0}	
	bit 63															bit 0			
0	15	14	13	12	11	10	9	8	7	6	5	4	a	b	c	d	3	0	
1	15	14	13	12	11	10	9	8	7	6	5	4	b	c	d	0	2	1	
2	15	14	13	12	11	10	9	8	7	6	5	4	c	d	1	0	1	2	
3	15	14	13	12	11	10	9	8	7	6	5	4	d	2	1	0	0	3	
4	15	14	13	12	11	10	9	8	a	B	c	d	3	2	1	0	3	4	
5	15	14	13	12	11	10	9	8	b	C	d	4	3	2	1	0	2	5	
6	15	14	13	12	11	10	9	8	c	D	5	4	3	2	1	0	1	6	
7	15	14	13	12	11	10	9	8	d	6	5	4	3	2	1	0	0	7	
8	15	14	13	12	a	b	c	d	7	6	5	4	3	2	1	0	3	8	
9	15	14	13	12	b	c	d	8	7	6	5	4	3	2	1	0	2	9	
10	15	14	13	12	c	d	9	8	7	6	5	4	3	2	1	0	1	10	
11	15	14	13	12	d	10	9	8	7	6	5	4	3	2	1	0	0	11	
12	a	b	c	d	11	10	9	8	7	6	5	4	3	2	1	0	3	12	
13	b	c	d	12	11	10	9	8	7	6	5	4	3	2	1	0	2	13	
14	c	d	13	12	11	10	9	8	7	6	5	4	3	2	1	0	1	14	
15	d	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	0	15	

SYNC.stype : Synchronize Shared Memory

MIPS II

To wait until a memory access or a pipeline operation during the execution is completed.

Operation Code

31	26	25		11	10	6	5	0
SPECIAL			0			stype		SYNC
000000			000 0000 0000 0000					001111
6			15			5		6

Format

SYNC	(stype = 0xxxx)
SYNC.L	(stype = 0xxxx)
SYNC.P	(stype = 1xxxx)

Description

The SYNC instruction synchronizes memory accesses or pipeline operations.

The SYNC and SYNC.L instructions wait until the preceding loads or stores are completed. The completion of loads indicates when the data is written into the destination register and the completion of stores indicates when the data is written into the data cache or the scratch-pad RAM or when the data is sent on the processor bus and the SYSDACK signal is asserted. Also, they flush the uncached accelerated buffer and writeback buffer. In this way, load and store instructions issued before SYNC or SYNC.L are guaranteed to execute before load and store instructions following SYNC or SYNC.L are executed, in orderly sequence.

The SYNC.P instruction waits until the preceding instruction is completed with the exception of multiply, divide, multicycle COP1 or COP2 operations or a pending load.

Restrictions

The SYNC instruction (SYNC.P or SYNC.L) is not allowed to execute in a branch delay slot, (the instruction immediately following a branch instruction).

Exceptions

None

Operation

SyncOperation(stype)

SYSCALL : System Call

MIPS I

To cause a System Call exception.

Operation Code

31	26	25		6	5	0
SPECIAL 000000			code			SYSCALL 001100
6			20			6

Format

SYSCALL

Description

A system call exception occurs, immediately and unconditionally transferring control to the exception handler. The code field is available and can be used for software parameters.

However, no special way for the exception handler to acquire the value of the code field is provided. It must be retrieved by determining the address of an instruction word from the EPC register, etc.

Exceptions

System Call

Operation

SignalException (SystemCall)

TEQ : Trap if Equal

MIPS II

To compare the values of two GPRs and take a Trap exception according to the result.

Operation Code

31	26	25	21	20	16	15	6	5	0
SPECIAL 000000	rs					rt	code		TEQ 110100
6	5					5	10		6

Format

TEQ rs, rt

Description

if (GPR[rs] = GPR[rt]) then Trap

Compares the contents of GPR[rs] and GPR[rt]. If they are equal, takes a trap exception.

The code field is available and can be used for software parameters.

However, no special way for the exception handler to acquire the value of the code field is provided. It must be retrieved by determining the address of an instruction word from the EPC register, etc.

Exceptions

Trap

Operation

if GPR[rs]_{63:0} = GPR[rt]_{63:0} then

 SignalException (Trap)

endif

TEQI : Trap if Equal Immediate

MIPS II

To compare a GPR with a constant and take a Trap exception according to the result.

Operation Code

31	26	25	21	20	16	15	0
REGIMM	rs	TEQI	immediate				
000001		01100					
6	5	5	16				

Format

TEQI rs, immediate

Description

if (GPR[rs] = immediate) then Trap

Compares the contents of GPR[rs] with the value of a sign-extended immediate. If they are equal, takes a Trap exception.

Exceptions

Trap

Operation

```

if GPR[rs] 63..0 = sign_extend (immediate) then
    SignalException (Trap)
endif

```

TGE : Trap if Greater or Equal

MIPS II

To compare the values of two GPRs and take a Trap exception according to the result.

Operation Code

31	26	25	21	20	16	15	6	5	0
SPECIAL 000000	rs	rt	code	TGE 110000					
6	5	5	10	6					

Format

TGE rs, rt

Description

if (GPR[rs] >= GPR[rt]) then Trap

Compares the values of GPR[rs] and GPR[rt]. If GPR[rs] is greater than or equal to GPR[rt], takes a trap exception.

The code field is available and can be used as software parameters.

However, no special way for the exception handler to acquire the value of the code is provided. The value of the code field must be retrieved by determining the address of an instruction word from the EPC register, etc.

Exceptions

Trap

Operation

```
if GPR[rs]63:0 >= GPR[rt]63:0 then
    SignalException(Trap)
endif
```

TGEI : Trap if Greater or Equal Immediate

MIPS II

To compare a GPR with a constant and take a Trap exception according to the result.

Operation Code

31	26	25	21	20	16	15	0
REGIMM						rs	
000001						TGEI	
						01000	
						immediate	
6						5	
						5	
						16	

Format

TGEI rs, immediate

Description

if (GPR[rs] >= immediate) then Trap

Compares the values of GPR[rs] with the value of the sign-extended immediate field. If GPR[rs] is greater than or equal to immediate, takes a Trap exception.

Exceptions

Trap

Operation

```

if GPR[rs]63:0 >= sign_extend(immediate) then
    SignalException(Trap)
endif

```

TGEIU : Trap if Greater or Equal Immediate Unsigned

MIPS II

To compare a GPR with a constant and take a Trap exception according to the result.

Operation Code

31	26	25	21	20	16	15	0
REGIMM		rs		TGEIU		immediate	
000001				01001			
6		5		5		16	

Format

TGEIU rs, immediate

Description

if (GPR[rs] >= immediate) then Trap

Compares the value of GPR[rs] with the value of the sign-extended immediate field as unsigned integers. If GPR[rs] is greater than or equal to immediate, takes a Trap exception.

Exceptions

Trap

Operation

```
if (0 || GPR[rs]63:0) >= (0 || sign_extend(immediate)) then
    SignalException(Trap)
endif
```

Programming Notes

Because the immediate is treated as an unsigned integer after it is sign-extended, the range of numeric values that the immediate represents is not sequential, but split into two areas; around the smallest and largest 64-bit unsigned integers. That is [0,32767] and max_unsigned-32767, max_unsigned], respectively.

TGEU : Trap if Greater or Equal Unsigned

MIPS II

To compare the values of two GPRs and take a Trap exception according to the result.

Operation Code

31	26	25	21	20	16	15	6	5	0
SPECIAL 000000	rs	rt	code	TGEU 110001					
6	5	5	10	6					

Format

TGEU rs, rt

Description

if (GPR[rs] >= GPR[rt]) then Trap

Compares the values of GPR[rs] and GPR[rt] as unsigned integers. If GPR[rs] is greater than or equal to GPR[rt], takes a trap exception.

The code field is available and can be used for software parameters.

However, no special way for the exception handler to acquire the value of the code is provided. It must be retrieved by determining the address of an instruction word from the EPC register, etc.

Exceptions

Trap

Operation

```
if (0 || GPR[rs]63..0) >= (0 || GPR[rt]63..0) then
    SignalException(Trap)
endif
```


TLT : Trap if Less Than

MIPS II

To compare the values of two GPRs and take a Trap exception according to the result.

Operation Code

31	26	25	21	20	16	15	6	5	0
SPECIAL 000000	rs	rt	code	TLT 110010					
6	5	5	10	6					

Format

TLT rs, rt

Description

if (GPR[rs] < GPR[rt]) then Trap

Compares the values of GPR[rs] and GPR[rt]. If GPR[rs] is less than GPR[rt], takes a trap exception.

The code field is available and can be used for software parameters.

However, no special way for the exception handler to acquire the value of the code is provided. It must be retrieved by determining the address of an instruction word from the EPC register, etc.

Exceptions

Trap

Operation

```
if GPR[rs]63..0 < GPR[rt]63..0 then
    SignalException(Trap)
endif
```

TLTI : Trap if Less Than Immediate

MIPS II

To compare a GPR with a constant and take a Trap exception according to the result.

Operation Code

31	26	25	21	20	16	15	0
REGIMM						rs	
000001						TLTI	
						01010	
						immediate	
6						5	
						5	
						16	

Format

TLTI rs, immediate

Description

if (GPR[rs] < immediate) then Trap

Compares the value of GPR[rs] with the value of a sign-extended immediate. If GPR[rs] is less than immediate, takes a Trap exception.

Exceptions

Trap

Operation

```

if GPR[rs]63:0 < sign_extend(immediate) then
    SignalException(Trap)
endif

```

TLTIU : Trap if Less Than Immediate Unsigned

MIPS II

To compare a GPR with a constant and take a Trap exception according to the result.

Operation Code

31	26	25	21	20	16	15	0
REGIMM	rs		TLTIU	immediate			
000001			01011				
6	5		5	16			

Format

TLTIU rs, immediate

Description

if (GPR[rs] < immediate) then Trap

Compares the values of GPR[rs] with the value of a sign-extended immediate as unsigned integers. If GPR[rs] is less than immediate, takes a Trap exception.

Exceptions

Trap

Operation

```
if (0 || GPR[rs]63:0) < (0 || sign_extend(immediate)) then
    SignalException(Trap)
endif
```

Programming Notes

Because the immediate field is treated as an unsigned integer after it is sign-extended, the range of numeric values that the immediate represents is not sequential, but split into two areas; around the smallest and largest 64-bit unsigned integers. That is [0,32767] and [max_unsigned-32767, max_unsigned], respectively.

TLTU : Trap if Less Than Unsigned

MIPS II

To compare the values of two GPRs and take a Trap exception according to the result.

Operation Code

31	26	25	21	20	16	15	6	5	0
SPECIAL 000000	rs	rt	code	TLTU 110011					
6	5	5	10	6					

Format

TLTU rs, rt

Description

if (GPR[rs] < GPR[rt]) then Trap

Compares the values of GPR[rs] and GPR[rt] as unsigned integers. If GPR[rs] is less than GPR[rt], takes a trap exception.

The code field is available and can be used for software parameters.

However, no special way for the exception handler to acquire the value of the code is provided. It must be retrieved by determining the address of an instruction word from the EPC register, etc.

Exceptions

Trap

Operation

```
if (0 || GPR[rs]63..0) < (0 || GPR[rt]63..0) then
    SignalException(Trap)
endif
```

TNE : Trap if Not Equal

MIPS II

To compare the values of two GPRs and take a Trap exception according to the result.

Operation Code

31	26	25	21	20	16	15	6	5	0
SPECIAL 000000	rs	rt	code	TNE 110110					
6	5	5	10	6					

Format

TNE rs, rt

Description

if (GPR[rs] \neq GPR[rt]) then Trap

Compares the values of GPR[rs] and GPR[rt]. If they are not equal, takes a trap exception.

The code field is available and can be used for software parameters.

However, no special way for the exception handler to acquire the value of the code is provided. It must be retrieved by determining the address of an instruction word from the EPC register, etc.

Exceptions

Trap

Operation

```
if GPR[rs]63..0  $\neq$  GPR[rt]63..0 then
    SignalException(Trap)
endif
```

TNEI : Trap if Not Equal Immediate

MIPS II

To compare a GPR with a constant and take a Trap exception according to the result.

Operation Code

31	26	25	21	20	16	15	0
REGIMM						rs	
000001						TNEI	
						01110	
						immediate	
6						5	
						5	
						16	

Format

TNEI rs, immediate

Description

if (rs \neq immediate) then Trap

Compares the value of GPR[rs] with the value of a sign-extended immediate. If they are not equal, takes a Trap exception.

Exceptions

Trap

Operation

if GPR[rs]_{63:0} \neq sign_extend(immediate) then

 SignalException(Trap)

endif

XOR : Exclusive OR

MIPS I

To calculate a bitwise logical EXCLUSIVE OR.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000	rs	rt	rd	0 00000	XOR 100110						
6	5	5	5	5	6						

Format

XOR rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] \text{ XOR } GPR[rt]$

Calculates a bitwise logical XOR between the contents of GPR[rs] and GPR[rt]. The result is stored in GPR[rd].

The truth table values for XOR are as follows;

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Exceptions

None

Operation

$GPR[rd]_{63..0} \leftarrow GPR[rs]_{63..0} \text{ XOR } GPR[rt]_{63..0}$

XORI : Exclusive OR Immediate

MIPS I

To calculate a bitwise logical EXCLUSIVE OR.

Operation Code

31	26	25	21	20	16	15	0
XORI						rs	rt
001110						immediate	
6						5	5
						16	

Format

XORI rt, rs, immediate

Description

$GPR[rt] \leftarrow GPR[rs] \text{ XOR immediate}$

Calculates a bitwise logical XOR between the value of the zero-extended immediate and contents of GPR[rs]. The result is stored in GPR[rt].

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Exceptions

None

Operation

$GPR[rt]_{63..0} \leftarrow GPR[rs]_{63..0} \text{ XOR zero_extend (immediate)}$

3. EE Core-Specific Instruction Set

This chapter describes the details of special CPU instructions that are part of the extended EE Core instruction set. These instructions are classified into the following three types;

- Three-operand Multiply and Multiply-Add instructions
- Multiply and Multiply-Add instructions using logical pipeline 1 (I1 pipe)
- Multimedia instructions (128-bit instructions)

DIV1 : Divide Word Pipeline 1

EE Core

To divide 32-bit signed integers. This operation is executed in logical pipeline 1.

Operation Code

31	26	25	21	20	16	15	6	5	0
MMI		rs		rt		0			DIV1
011100						00 0000 0000			011010
6		5		5		10			6

Format

DIV1 rs, rt

Description

$(LO1, HI1) \leftarrow GPR[rs] / GPR[rt]$

Divides the 32-bit value in GPR[rs] by the 32-bit value in GPR[rt]. The 32-bit quotient and the 32-bit remainder are stored in the LO1(LO_{127...64}) and HI1(HI_{127...64}) registers respectively. Both GPR[rs] and GPR[rt] are treated as signed values. The sign of the quotient and remainder are determined as shown in the following table;

Dividend GPR[rs]	Divisor GPR[rt]	Quotient LO	Remainder HI
Positive	Positive	Positive	Positive
Positive	Negative	Negative	Positive
Negative	Positive	Negative	Negative
Negative	Negative	Positive	Negative

Restrictions

If GPR[rt] and GPR[rs] are not sign-extended 32-bit values (bits 63..31 equal), then the result is undefined.
Also, if the value in GPR[rt] is zero, the arithmetic result is undefined.

Exceptions

None. If the divisor is zero, an exception does not occur on overflow.

Operation

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif

quotient $\leftarrow GPR[rs]_{31..0} \text{ DIV } GPR[rt]_{31..0}$

remainder $\leftarrow GPR[rs]_{31..0} \text{ MOD } GPR[rt]_{31..0}$

LO_{127..64} $\leftarrow (\text{quotient}_{31})^{32} \parallel \text{quotient}_{31..0}$

HI_{127..64} $\leftarrow (\text{remainder}_{31})^{32} \parallel \text{remainder}_{31..0}$

Programming Notes

In the EE Core, the integer divide operation proceeds asynchronously. An attempt to read the contents of the LO or HI register before the divide operation finishes will result in interlock. Other CPU instructions can execute without delay. Therefore, scheduling the divide operation appropriately can improve performance.

When 0x80000000(−2147483648), the signed minimum value, is divided by 0xFFFFFFFF(−1), the operation will result in an overflow. However, in this instruction an overflow exception does not occur and the following results will be returned.

Quotient: 0x80000000 (−2147483648), and remainder: 0x00000000 (0)

If an overflow or divide-by-zero is required to be detected, then add an instruction that detects these conditions following the divide instruction. Since the divide instruction is asynchronous, the divide operation and check can be executed in parallel. If an overflow or divide-by-zero is detected, then the system software can be informed of the problem by generating an exception using an appropriate code value with a BREAK instruction.

DIVU1 : Divide Unsigned Word Pipeline 1

EE Core

To divide 32-bit unsigned integers. This operation is executed in logical pipeline 1.

Operation Code

31	26	25	21	20	16	15	6	5	0
MMI		rs		rt		0			DIVU1
011100						00 0000 0000			011011
6		5		5		10			6

Format

DIVU1 rs, rt

Description

$(LO1, HI1) \leftarrow GPR[rs] / GPR[rt]$

Divides the 32-bit value in GPR[rs] by the 32-bit value in GPR[rt]. The 32-bit quotient and the 32-bit remainder are stored in the LO1(LO_{127...64}) and HI1(HI_{127...64}) registers respectively. Both GPR[rs] and GPR[rt] are treated as unsigned values.

Restrictions

If GPR[rt] and GPR[rs] are not sign-extended 32-bit values (bits 63..31 equal), then the result is undefined.

Exceptions

None. Even if the divisor is zero, an exception does not occur.

Operation

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif

quotient $\leftarrow (0 \mid \mid GPR[rs]_{31..0}) \text{ DIV } (0 \mid \mid GPR[rt]_{31..0})$

remainder $\leftarrow (0 \mid \mid GPR[rs]_{31..0}) \text{ MOD } (0 \mid \mid GPR[rt]_{31..0})$

LO_{127..64} $\leftarrow (\text{quotient}_{31})^{32} \mid \mid \text{quotient}_{31..0}$

HI_{127..64} $\leftarrow (\text{remainder}_{31})^{32} \mid \mid \text{remainder}_{31..0}$

Programming Notes

In the EE Core, the integer divide operation proceeds asynchronously. An attempt to read the contents of the LO or HI register before the divide operation finishes will result in interlock. Other CPU instructions can execute without delay. Therefore, scheduling the divide operation appropriately can improve performance.

If divide-by-zero is required to be detected, then add an instruction that detects this condition following the divide instruction. Since the divide instruction is asynchronous, the divide operation and check can be executed in parallel. If an overflow or divide-by-zero is detected, then the system software can be informed of the problem by generating an exception using an appropriate code value with a BREAK instruction.

LQ : Load Quadword

128-bit MMI

To load a 128-bit data from memory.

Operation Code

31	26	25	21	20	16	15	0
LQ						base	
011110						rt	
						offset	
6						5	
						5	
						16	

Format

LQ rt, offset (base)

Description

$GPR[rt] \leftarrow \text{memory}[GPR[base] + \text{offset}]$

Adds the offset as a 16-bit signed number to the value in GPR[base] to form the effective address. Loads the 128-bit data at the address and stores it in GPR[rt].

The least-significant four bits of the effective address are masked to zero when accessing memory.

Therefore, the effective address does not have to conform to the natural alignment.

Exceptions

TLB Refill, TLB Invalid, Address Error (excluding Address Error due to alignment)

Operation

$vAddr \leftarrow \text{sign_extend}(\text{offset}) + GPR[base]$

$vAddr_{3..0} = 0^4$

$(pAddr, \text{uncached}) \leftarrow \text{AddressTranslation}(vAddr, \text{DATA}, \text{LOAD})$

$\text{memquad} \leftarrow \text{LoadMemory}(\text{uncached}, \text{QUADWORD}, pAddr, vAddr, \text{DATA})$

$GPR[rt]_{127..0} \leftarrow \text{memquad}$

MADD : Multiply-Add word

EE Core

To multiply 32-bit values in GPRs and add to the HI and LO registers.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI 011100						rs		rt		rd	
6						5		5		5	
										0 00000	
										MADD 000000	
										6	

Format

MADD rs, rt

MADD rd, rs, rt

Description

$(\text{GPR}[\text{rd}], \text{HI}, \text{LO}) \leftarrow (\text{HI}, \text{LO}) + \text{GPR}[\text{rs}] \times \text{GPR}[\text{rt}]$

Multiplies the 32-bit value in GPR[rs] by the 32-bit value in GPR[rt] as signed integers. Adds the resulting 64-bit product to the values of the HI and LO registers and stores the high-order 32-bit result in HI0 and the low-order 32-bit result in LO0 and GPR[rd].

If rd is omitted in assembly language, zero is used for the default value. Since GPR[0] is the register whose value is fixed to zero, the arithmetic result will be stored only in the HI and LO registers.

Restrictions

If GPR[rt] and GPR[rs] are not sign-extended 32-bit values (bits 63..31 equal), then the result is undefined.

Exceptions

None

Operation

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif

prod $\leftarrow (\text{HI}_{31..0} \parallel \text{LO}_{31..0}) + \text{GPR}[\text{rs}]_{31..0} \times \text{GPR}[\text{rt}]_{31..0}$

LO_{63..0} $\leftarrow (\text{prod}_{31})^{32} \parallel \text{prod}_{31..0}$

HI_{63..0} $\leftarrow (\text{prod}_{63})^{32} \parallel \text{prod}_{63..32}$

GPR[rd]_{63..0} $\leftarrow (\text{prod}_{31})^{32} \parallel \text{prod}_{31..0}$

Programming Notes

In the EE Core, the multiply accumulate operation proceeds asynchronously. An attempt to read the contents of the LO/HI/GPR[rd] registers before the operation finishes will result in interlock. Other CPU instructions can execute in parallel with the multiply accumulate operation. Therefore, scheduling the multiply accumulate operation appropriately can improve performance of the software.

MADD1 : Multiply-Add word Pipeline 1

EE Core

To multiply the 32-bit values in GPRs and add to the HI and LO registers. This operation is executed in logical pipeline 1.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI 011100	rs					rt					0 00000
6	5					5					6

Format

MADD1 rs, rt

MADD1 rd, rs, rt

Description

$$(GPR[rd], HI1, LO1) \leftarrow (HI1, LO1) + GPR[rs] \times GPR[rt]$$

Multiplies the 32-bit value in GPR[rs] by the 32-bit value in GPR[rt] as signed integers. Adds the resulting 64-bit product to the values of the HI1(HI_{127..64}) and LO1(LO_{127..64}) registers and stores the high-order 32-bit result in the HI0 register and the low-order 32-bit result in the LO0 register and GPR[rd].

If rd is omitted in assembly language, zero is used for the default value. Since GPR[0] is the register whose value is fixed to zero, the arithmetic result will be stored only in the HI1 and LO1 registers.

Restrictions

If GPR[rt] and GPR[rs] are not sign-extended 32-bit values (bits 63..31 equal), then the result is undefined.

Exceptions

None

Operation

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif

$$\text{prod} \leftarrow (HI_{95..64} \parallel LO_{95..64}) + GPR[rs]_{31..0} \times GPR[rt]_{31..0}$$

$$LO_{127..64} \leftarrow (\text{prod}_{31})^{32} \parallel \text{prod}_{31..0}$$

$$HI_{127..64} \leftarrow (\text{prod}_{63})^{32} \parallel \text{prod}_{63..32}$$

$$GPR[rd]_{63..0} \leftarrow (\text{prod}_{31})^{32} \parallel \text{prod}_{31..0}$$

Programming Notes

In the EE Core, the multiply accumulate operation proceeds asynchronously. An attempt to read the contents of the LO/HI/GPR[rd] registers before the operation finishes will result in interlock. Other CPU instructions can execute in parallel with the multiply accumulate operation. Therefore, scheduling the multiply accumulate operation appropriately can improve performance of the software.

MADDU : Multiply-Add Unsigned word

EE Core

To multiply the 32-bit values in GPRs as unsigned integers and add to the HI and LO registers.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	0	MADDU						
011100				00000	000001						
6	5	5	5	5	6						

Format

MADDU rs, rt

MADDU rd, rs, rt

Description

$(\text{GPR}[\text{rd}], \text{HI}, \text{LO}) \leftarrow (\text{HI}, \text{LO}) + \text{GPR}[\text{rs}] \times \text{GPR}[\text{rt}]$

Multiplies the 32-bit value in GPR[rs] by the 32-bit value in GPR[rt] as unsigned integers. Adds the resulting 64-bit product to the values of the HI and LO registers and stores the high-order 32-bit result in the HI0 register and the low-order 32-bit result in the LO0 register and GPR[rd].

If rd is omitted in assembly language, zero is used for the default value. Since GPR[0] is the register whose value is fixed to zero, the arithmetic result will be stored only in the HI and LO registers.

Restrictions

If GPR[rt] and GPR[rs] are not sign-extended 32-bit values (bits 63..31 equal), then the result is undefined.

Exceptions

None

Operation

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif
prod      ← (HI31..0 || LO31..0) + (0 || GPR[rs]31..0) × (0 || GPR[rt]31..0)
LO63..0   ← (prod31)32 || prod31..0
HI63..0   ← (prod63)32 || prod63..32
GPR[rd]63..0 ← (prod31)32 || prod31..0
```

Programming Notes

In the EE Core, the multiply accumulate operation proceeds asynchronously. An attempt to read the contents of the LO/HI/GPR[rd] registers before the operation finishes will result in interlock. Other CPU instructions can execute in parallel with the multiply accumulate operation. Therefore, scheduling the multiply accumulate operation appropriately can improve performance of the software.

MADDU1 : Multiply-Add Unsigned word Pipeline 1

EE Core

To multiply the 32-bit values in GPRs as unsigned integers and add to the HI and LO registers. This operation is executed in pipeline 1.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI 011100	rs					rt					0 00000
6	5					5					6

Format

MADDU1 rs, rt

MADDU1 rd, rs, rt

Description

$$(GPR[rd], HI1, LO1) \leftarrow (HI1, LO1) + GPR[rs] \times GPR[rt]$$

Multiplies the 32-bit value in GPR[rs] by the 32-bit value in GPR[rt] as unsigned integers. Adds the resulting 64-bit product to the values of the HI1 (HI_{127..64}) and LO1 (LO_{127..64}) registers and stores the high-order 32-bit result in the HI1 register and the low-order 32-bit result in the LO1 register and GPR[rd].

If rd is omitted in assembly language, zero is used for the default value. Since GPR[0] is the register whose value is fixed to zero, the arithmetic result will be stored only in the HI1 and LO1 registers.

Restrictions

If GPR[rt] and GPR[rs] are not sign-extended 32-bit values (bits 63..31 equal), then the result is undefined.

Exceptions

None

Operation

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif

$$\text{prod} \leftarrow (HI_{95..64} \parallel LO_{95..64}) + (0 \parallel GPR[rs]_{31..0}) \times (0 \parallel GPR[rt]_{31..0})$$

$$LO_{127..64} \leftarrow (\text{prod}_{31})^{32} \parallel \text{prod}_{31..0}$$

$$HI_{127..64} \leftarrow (\text{prod}_{63})^{32} \parallel \text{prod}_{63..32}$$

$$GPR[rd]_{63..0} \leftarrow (\text{prod}_{31})^{32} \parallel \text{prod}_{31..0}$$

Programming Notes

In the EE Core, the multiply accumulate operation proceeds asynchronously. An attempt to read the contents of the LO/HI/GPR[rd] registers before the operation finishes will result in interlock. Other CPU instructions can execute in parallel with the multiply accumulate operation. Therefore, scheduling the multiply accumulate operation appropriately can improve performance of the software.

MFHI1 : Move From HI1 Register

EE Core

To move the contents of the HI1 register to a GPR.

Operation Code

31	26	25	16	15	11	10	6	5	0
MMI	0				rd	0	MFHI1		
011100	00 0000 0000					00000	010000		
6	10				5	5	6		

Format

MFHI1 rd

Description

$GPR[rd] \leftarrow HI1$

Copies the contents of the HI1 (=HI_{127...64}) register in GPR[rd].

Exceptions

None

Operation

$GPR[rd]_{63..0} \leftarrow HI_{127..64}$

MFLO1 : Move From LO1 Register

EE Core

To move the contents of the LO1 register to a GPR.

Operation Code

31	26	25	16	15	11	10	6	5	0
MMI	0		rd	0		MFLO1			
011100	00 0000 0000			00000		010010			
6	10		5	5		6			

Format

MFLO1 rd

Description

$GPR[rd] \leftarrow LO1$

Copies the contents of the LO1 (=LO_{127...64}) register in GPR[rd].

Exceptions

None

Operation

$GPR[rd]_{63..0} \leftarrow LO_{127..64}$

MFSA : Move from Shift Amount Register

EE Core

To save the contents of the SA register in a GPR.

Operation Code

31	26	25	16	15	11	10	6	5	0
SPECIAL 000000	0 00 0000 0000					rd	0 00000	MFSA 101000	
6	10					5	5	6	

Format

MFSA rd

Description

$GPR[rd] \leftarrow SA$

Copies the contents of the SA register, which holds the funnel shift amount, in GPR[rd]. This instruction is provided for saving the SA register during a context switch. Since the value of the SA register is encoded in a special manner, the software cannot use the resulting value in GPR[rd]. Uses the MTSA instruction to restore the saved values in SA.

Exceptions

None

Operation

$GPR[rd]_{63:0} \leftarrow SA$

Programming Notes

This instruction operates only in pipeline 0.

MTHI1 : Move To HI1 Register

EE Core

To move the value of a GPR to the HI1 register.

Operation Code

31	26	25	21	20	6	5	0
MMI	rs			0			MTHI1
011100				000 0000 0000 0000			010001
6	5			15			6

Format

MTHI1 rs

Description

$HI1 \leftarrow GPR[rs]$

Copies the contents of GPR[rs] to the HI1 (=HI_{127...64}) register.

Exceptions

None

Operation

$HI_{127..64} \leftarrow GPR[rs]_{63..0}$

MTLO1 : Move To LO1 Register

EE Core

To move the value of a GPR to the LO 1 register.

Operation Code

31	26	25	21	20	6	5	0
MMI	rs		0			MTLO1	
011100			000 0000 0000 0000			010001	
6	5		15			6	

Format

MTLO1 rs

Description

$LO1 \leftarrow GPR[rs]$

Copies the contents of GPR[rs] to the LO1 (=LO_{127...64}) register.

Exceptions

None

Operation

$LO_{127..64} \leftarrow GPR[rs]_{63..0}$

MTSA : Move to Shift Amount Register

EE Core

To restore the saved values in a GPR into the SA register.

Operation Code

31	26	25	21	20	6	5	0
SPECIAL	rs			0		MTSA	
000000				000 0000 0000 0000		101001	
6	5			15		6	

Format

MTSA rs

Description

$SA \leftarrow GPR[rs]$

Copies the contents of GPR[rs] into the SA register, which holds the funnel shift amount.

This instruction is provided for restoring the values of SA saved with the MFSA instruction during a context switch.

The contents of GPR[rs] must be the value saved with the MFSA instruction. Otherwise, the result of the MTSA instruction is undefined. That is, setting the funnel shift amount newly with the MTSA instruction is not allowed. Use the MTSAB and MTSABH instructions to do this.

Restrictions

The three instructions prior to the MTSA instruction must not access SA register. That is, placing a MFSA, MTSAB, MTSABH or QFSRV instruction in the three steps preceding the MTSA instruction is not allowed.

Exceptions

None

Operation

$SA \leftarrow GPR[rs]_{63..0}$

Programming Notes

The MTSA instruction operates only in logical pipeline 0.

MTSAB : Move Byte Count to Shift Amount Register

EE Core

To set a byte shift count in the SA register.

Operation Code

31	26	25	21	20	16	15	0
REGIMM 000001						rs	11000
6						5	5
						immediate	
						16	

Format

MTSAB rs, immediate

Description

$$SA \leftarrow (GPR[rs] \text{ XOR } \text{immediate}) \times 8$$

Calculates a bitwise logical XOR between the least-significant four bits of GPR[rs] and those of the immediate value. The result is stored in SA as a byte shift amount.

Restrictions

The three instructions prior to the MTSAB instruction must not read the SA register; that is, they must not be the MFSA or QFSRV instruction.

Exceptions

None

Operation

$$SA \leftarrow (GPR[rs]_{3:0} \text{ XOR } \text{immediate}_{3:0}) \times 8$$

Programming Notes

The MTSAB instruction operates only in logical pipeline 0.

Specifying rs or immediate differs as follows;

mtsab 0, 5 // Sets shifts amount to "5 bytes" .

mtsab 5, 0 // Sets the contents of GPR[5] as a byte shift amount.

MTSAH : Move Halfword Count to Shift Amount Register

EE Core

To set a halfword shift count in the SA register.

Operation Code

31	26	25	21	20	16	15	0
REGIMM 000001	rs		11001		immediate		
6	5		5		16		

Format

MTSAH rs, immediate

Description

$$SA \leftarrow (GPR[rs] \text{ XOR } \text{immediate}) \times 16$$

Calculates a bitwise logical XOR between the least-significant three bits of GPR[rs] and those of the immediate value. The result is stored into SA as a halfword shift amount.

Restrictions

The three instructions prior to the MTSAB instruction must not read the SA register; that is, they must not be the MFSA or QFSRV instruction.

Exceptions

None

Operation

$$SA \leftarrow (GPR[rs]_{2..0} \text{ XOR } \text{immediate}_{2..0}) \times 16$$

Programming Notes

The MTSAH instruction operates only in logical pipeline 0.

Specifying rs or immediate differs as follows;

mtsab 0, 5 // Sets shifts amount to "5 halfwords"

mtsab 5, 0 // Sets the contents of GPR[5] as a byte shift amount.

MULT : Multiply Word

EE Core

To multiply 32-bit signed integers.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000		rs		rt		rd		0 00000		MULT 011000	
6		5		5		5		5		6	

Format

MULT rd, rs, rt

MULT rs, rt

Description

$(\text{GPR}[\text{rd}], \text{LO}, \text{HI}) \leftarrow \text{GPR}[\text{rs}] \times \text{GPR}[\text{rt}]$

Multiplies the 32-bit value in GPR[rt] by the 32-bit value in GPR[rs] as signed integers. The low-order 32 bits and the high-order 32 bits of the 64-bit result are stored in the LO register and GPR[rd], and the HI register, respectively.

Restrictions

If GPR[rt] and GPR[rs] are not sign-extended 32-bit values (bits 63..31 equal), then the result is undefined.

Exceptions

None. No arithmetic exception occurs.

Operation

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif

prod $\leftarrow \text{GPR}[\text{rs}]_{31..0} \times \text{GPR}[\text{rt}]_{31..0}$

LO_{63..0} $\leftarrow (\text{prod}_{31})^{32} \mid \mid \text{prod}_{31..0}$

HI_{63..0} $\leftarrow (\text{prod}_{63})^{32} \mid \mid \text{prod}_{63..32}$

GPR[rd]_{63..0} $\leftarrow (\text{prod}_{31})^{32} \mid \mid \text{prod}_{31..0}$

Programming Notes

In the EE Core, the integer multiply operation proceeds asynchronously. An attempt to read the contents of the LO/HI/GPR[rd] registers before the multiply operation finishes will result in interlock. Other CPU instructions can execute in parallel. Therefore, scheduling the multiply operation appropriately can improve performance.

Even when the result of the multiply operation overflows, an overflow exception does not occur. If an overflow is required to be detected, an explicit check is necessary.

If rd is omitted in assembly language, zero is used for the default value. Since GPR[0] is the register whose value is fixed to zero, the arithmetic result will be stored only in the HI and LO registers. That is, the result is the same as the MULT instruction in the MIPS I level.

MULT1 : Multiply Word Pipeline 1

EE Core

To multiply 32-bit signed integers. This operation is executed in logical pipeline 1.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	0	MULT1						
011100					00000				011000		
6	5	5	5	5	5	6					

Format

MULT1 rd, rs, rt

MULT1 rs, rt

Description

$(\text{GPR}[\text{rd}], \text{LO1}, \text{HI1}) \leftarrow \text{GPR}[\text{rs}] \times \text{GPR}[\text{rt}]$

Multiplies the 32-bit value in GPR[rt] by the 32-bit value in GPR[rs] as signed integer values. The low-order 32 bits and the high-order 32 bits of the resulting 64-bit value are stored in the LO1(LO_{127..64}) register and GPR[rd], and the HI1(HI_{127..64}) register respectively.

Restrictions

If GPR[rt] and GPR[rs] are not sign-extended 32-bit values (bits 63..31 equal), then the result is undefined.

Exceptions

None. No arithmetic exception occurs.

Operation

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif

prod $\leftarrow \text{GPR}[\text{rs}]_{31..0} \times \text{GPR}[\text{rt}]_{31..0}$

LO_{127..64} $\leftarrow (\text{prod}_{31})^{32} \parallel \text{prod}_{31..0}$

HI_{127..64} $\leftarrow (\text{prod}_{63})^{32} \parallel \text{prod}_{63..32}$

GPR[rd]_{63..0} $\leftarrow (\text{prod}_{31})^{32} \parallel \text{prod}_{31..0}$

Programming Notes

In the EE Core, the integer multiply operation proceeds asynchronously. An attempt to read the contents of the LO/HI/GPR[rd] registers before the multiply operation finishes will result in interlock. Other CPU instructions can execute in parallel. Therefore, scheduling the multiply operation appropriately can improve performance.

Even when the result of the multiply operation overflows, an overflow exception does not occur. If Overflow is required to be detected, an explicit check is necessary.

MULTU : Multiply Unsigned Word

EE Core

To multiply 32-bit signed integers.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000		rs		rt		rd		0 00000		MULTU 011001	
6		5		5		5		5		6	

Format

MULTU rd, rs, rt

MULTU rs, rt

Description

$(LO, HI) \leftarrow GPR[rs] \times GPR[rt]$

Multiplies the 32-bit value in GPR[rt] by the 32-bit value in GPR[rs] as unsigned integer values. The low-order 32 bits and the high-order 32 bits of the resulting 64-bit value are stored in the LO and HI registers respectively.

No arithmetic exception occurs under any circumstances.

Restrictions

If GPR[rt] and GPR[rs] are not sign-extended 32-bit values (bits 63..31 equal), then the result is undefined.

Exceptions

None

Operation

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif

prod $\leftarrow (0 \mid \mid GPR[rs]_{31..0}) \times (0 \mid \mid GPR[rt]_{31..0})$

LO_{63..0} $\leftarrow (prod_{31})^{32} \mid \mid prod_{31..0}$

HI_{63..0} $\leftarrow (prod_{63})^{32} \mid \mid prod_{63..32}$

GPR[rd]_{63..0} $\leftarrow (prod_{31})^{32} \mid \mid prod_{31..0}$

Programming Notes

See "Programming Notes" for the MULT instruction.

MULTU1 : Multiply Unsigned Word Pipeline 1

EE Core

To multiply 32-bit unsigned integers. This operation is executed in logical pipeline 1.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	0	MULTU1						
011100				00000	011001						
6	5	5	5	5	6						

Format

MULTU1 rd, rs, rt

MULTU1 rs, rt

Description

$(\text{GPR}[\text{rd}], \text{LO1}, \text{HI1}) \leftarrow \text{GPR}[\text{rs}] \times \text{GPR}[\text{rt}]$

Multiplies the 32-bit value in GPR[rt] by the 32-bit value in GPR[rs] as unsigned integers. The low-order 32 bits and the high-order 32 bits of the resulting 64-bit value are stored in the LO1(LO_{127..64}) register and GPR[rd], and the HI1(HI_{127..64}) register, respectively.

Restrictions

If GPR[rt] and GPR[rs] are not sign-extended 32-bit values (bits 63..31 equal), then the result is undefined.

Exceptions

None

Operation

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif

prod $\leftarrow (0 \mid \mid \text{GPR}[\text{rs}]_{31..0}) \times (0 \mid \mid \text{GPR}[\text{rt}]_{31..0})$

LO_{127..64} $\leftarrow (\text{prod}_{31})^{32} \mid \mid \text{prod}_{31..0}$

HI_{127..64} $\leftarrow (\text{prod}_{63})^{32} \mid \mid \text{prod}_{63..32}$

GPR[rd]_{63..0} $\leftarrow (\text{prod}_{31})^{32} \mid \mid \text{prod}_{31..0}$

Programming Notes

See "Programming Notes" for the MULT1 instruction.

PABSH : Parallel Absolute Halfword

128-bit MMI

To calculate the absolute value of 8 16-bit integers in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0	
MMI 011100			0 00000		rt		rd		PABSH 00101		MMI1 101000	
6			5		5		5		5		6	

Format

PABSH rd, rt

Description

$\text{GPR}[\text{rd}] \leftarrow |\text{GPR}[\text{rt}]|$

Splits the 128-bit value in GPR[rt] into eight 16-bit signed integers, calculates their absolute values and stores them in the corresponding halfwords in GPR[rd].

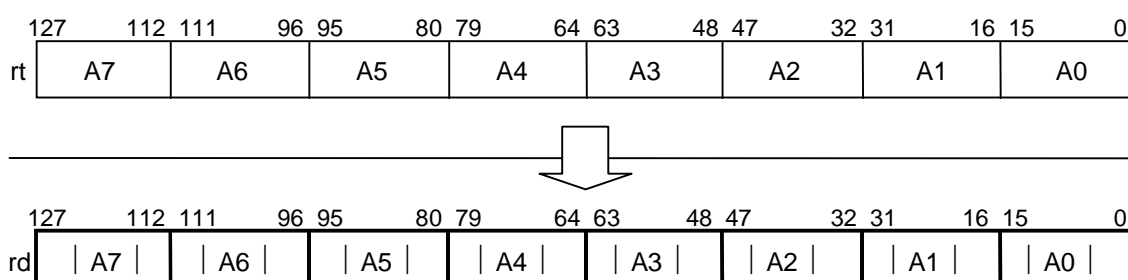
If a value is 0x8000(−32768), the operation will result in an overflow. However, the result is truncated to 0x7FFF(+32767) and an overflow exception does not occur.

Exceptions

None

Operation

$\text{GPR}[\text{rd}]_{15..0} \leftarrow |\text{GPR}[\text{rt}]_{15..0}|$
 $\text{GPR}[\text{rd}]_{31..16} \leftarrow |\text{GPR}[\text{rt}]_{31..16}|$
 $\text{GPR}[\text{rd}]_{47..32} \leftarrow |\text{GPR}[\text{rt}]_{47..32}|$
 $\text{GPR}[\text{rd}]_{63..48} \leftarrow |\text{GPR}[\text{rt}]_{63..48}|$
 $\text{GPR}[\text{rd}]_{79..64} \leftarrow |\text{GPR}[\text{rt}]_{79..64}|$
 $\text{GPR}[\text{rd}]_{95..80} \leftarrow |\text{GPR}[\text{rt}]_{95..80}|$
 $\text{GPR}[\text{rd}]_{111..96} \leftarrow |\text{GPR}[\text{rt}]_{111..96}|$
 $\text{GPR}[\text{rd}]_{127..112} \leftarrow |\text{GPR}[\text{rt}]_{127..112}|$



PABSW : Parallel Absolute Word

EE Core

To calculate the absolute value of 4 32-bit integers in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	0	rt	rd	PABSW	MMI1						
011100	00000			00001	101000						
6	5	5	5	5	6						

Format

PABSW rd, rt

Description

$$\text{GPR}[\text{rd}] \leftarrow |\text{GPR}[\text{rt}]|$$

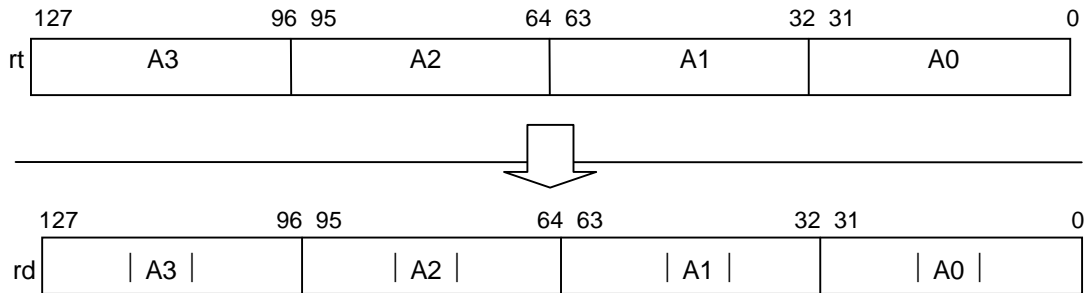
Splits the 128-bit value in GPR[rt] into four 32-bit signed integers, calculates their absolute values and stores them in the corresponding words in GPR[rd].

If a value is 0x80000000 (−2147483648), the operation will result in an overflow. However, the result is truncated to 0x7FFFFFFF (+2147483647) and an overflow exception does not occur.

Exceptions

None

Operation

$$\begin{aligned} \text{GPR}[\text{rd}]_{31..0} &\leftarrow |\text{GPR}[\text{rt}]_{31..0}| \\ \text{GPR}[\text{rd}]_{63..32} &\leftarrow |\text{GPR}[\text{rt}]_{63..32}| \\ \text{GPR}[\text{rd}]_{95..64} &\leftarrow |\text{GPR}[\text{rt}]_{95..64}| \\ \text{GPR}[\text{rd}]_{127..96} &\leftarrow |\text{GPR}[\text{rt}]_{127..96}| \end{aligned}$$


PADDB : Parallel Add Byte

128-bit MMI

To add 16 pairs of 8-bit integers in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PADDB	MMI0						
011100				01000	001000						
6	5	5	5	5	6						

Format

PADDB rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] + GPR[rt]$

Splits the 128-bit values in GPR[rs] and GPR[rt] into sixteen 8-bit integers, adds the data in GPR[rs] to the corresponding data in GPR[rt] and stores them in the corresponding bytes in GPR[rd].

Exceptions

Even when the result of the arithmetic operation overflows or underflows, an overflow exception does not occur.

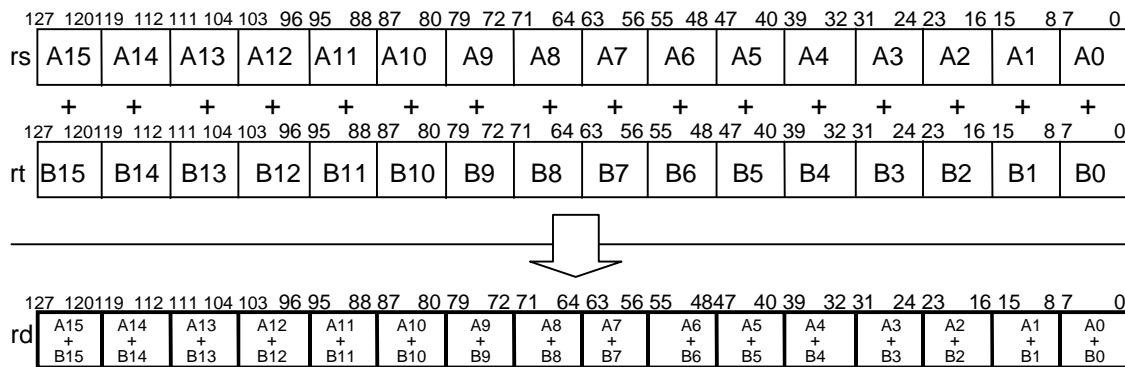
Operation

$GPR[rd]_{7..0} \leftarrow (GPR[rs]_{7..0} + GPR[rt]_{7..0})_{7..0}$

$GPR[rd]_{15..8} \leftarrow (GPR[rs]_{15..8} + GPR[rt]_{15..8})_{7..0}$

(The same operations follow every 8 bits)

$GPR[rd]_{127..120} \leftarrow (GPR[rs]_{127..120} + GPR[rt]_{127..120})_{7..0}$



PADDH : Parallel Add Halfword

128-bit MMI

To add 8 pairs of 16-bit integers in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PADDH	MMI0						
011100				00100	001000						
6	5	5	5	5	6						

Format

PADDH rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] + GPR[rt]$

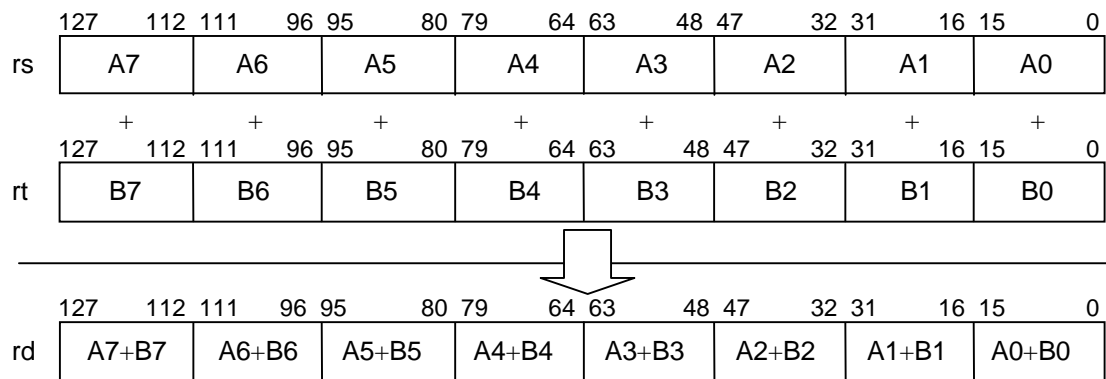
Splits the 128-bit values in GPR[rs] and GPR[rt] into eight 16-bit integers, adds the data in GPR[rs] to the corresponding data in GPR[rt] and stores them in the corresponding halfwords in GPR[rd].

Exceptions

None

Operation

$GPR[rd]_{15..0} \leftarrow (GPR[rs]_{15..0} + GPR[rt]_{15..0})_{15..0}$
 $GPR[rd]_{31..16} \leftarrow (GPR[rs]_{31..16} + GPR[rt]_{31..16})_{15..0}$
 $GPR[rd]_{47..32} \leftarrow (GPR[rs]_{47..32} + GPR[rt]_{47..32})_{15..0}$
 $GPR[rd]_{63..48} \leftarrow (GPR[rs]_{63..48} + GPR[rt]_{63..48})_{15..0}$
 $GPR[rd]_{79..64} \leftarrow (GPR[rs]_{79..64} + GPR[rt]_{79..64})_{15..0}$
 $GPR[rd]_{95..80} \leftarrow (GPR[rs]_{95..80} + GPR[rt]_{95..80})_{15..0}$
 $GPR[rd]_{111..96} \leftarrow (GPR[rs]_{111..96} + GPR[rt]_{111..96})_{15..0}$



PADDSB : Parallel Add with Signed saturation Byte

128-bit MMI

To add 16 pairs of 8-bit signed integers with saturation in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PADDSB	MMI0						
011100				11000	001000						
6	5	5	5	5	6						

Format

PADDSB rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] + GPR[rt]$

Splits the 128-bit values in GPR[rs] and GPR[rt] into sixteen 8-bit signed integers, adds the data in GPR[rs] to the corresponding data in GPR[rt] and stores them in the corresponding bytes in GPR[rd].

Arithmetic results beyond the range of a signed 8-bit integer are saturated as follows;

Overflow : $\rightarrow 0x7F$

Underflow : $\rightarrow 0x80$

Exceptions

None

Operation

```

if ((GPR[rs]7..0 + GPR[rt]7..0) > 0x7F) then
    GPR[rd]7..0 ← 0x7F
else if (0x100 ≤ (GPR[rs]7..0 + GPR[rt]7..0) < 0x180) then
    GPR[rd]7..0 ← 0x80
else
    GPR[rd]7..0 ← (GPR[rs]7..0 + GPR[rt]7..0)7..0
endif

```

```

if ((GPR[rs]15..8 + GPR[rt]15..8) > 0x7F) then
    GPR[rd]15..8 ← 0x7F
else if (0x100 ≤ (GPR[rs]15..8 + GPR[rt]15..8) < 0x180) then
    GPR[rd]15..8 ← 0x80
else
    GPR[rd]15..8 ← (GPR[rs]15..8 + GPR[rt]15..8)7..0
endif

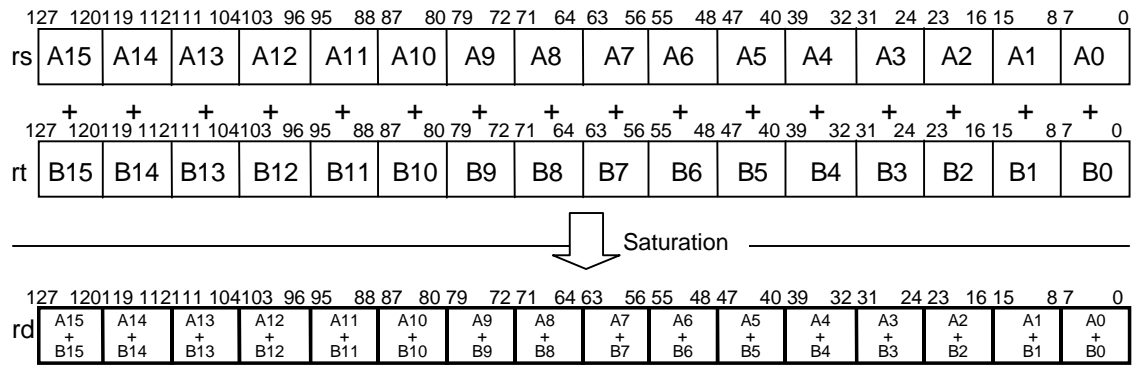
```

(The same operations follow every 8 bits)

```

if ((GPR[rs]127..120 + GPR[rt]127..120) > 0x7F) then
    GPR[rd]127..120 ← 0x7F
else if (0x100 ≤ (GPR[rs]127..120 + GPR[rt]127..120) < 0x180) then
    GPR[rd]127..120 ← 0x80
else
    GPR[rd]127..120 ← (GPR[rs]127..120 + GPR[rt]127..120)7..0
endif

```



PADDSH : Parallel Add with Signed saturation Halfword

128-bit MMI

To add 8 pairs of 16-bit signed integers with saturation in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PADDSH	MMI0						
011100				10100	001000						
6	5	5	5	5	6						

Format

PADDSH rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] + GPR[rt]$

Splits the 128-bit values in GPR[rs] and GPR[rt] into eight 16-bit signed integers, adds the data in GPR[rs] to the corresponding data in GPR[rt] and stores them in the corresponding halfwords in GPR[rd].

Arithmetic results beyond the range of a signed 16-bit integer are saturated as follows;

Overflow : $\rightarrow 0x7FFF$

Underflow : $\rightarrow 0x8000$

Exceptions

None

Operation

if $((GPR[rs]_{15..0} + GPR[rt]_{15..0}) > 0x7FFF)$ then

$GPR[rd]_{15..0} \leftarrow 0x7FFF$

else if $(0x10000 \leq (GPR[rs]_{15..0} + GPR[rt]_{15..0}) < 0x18000)$ then

$GPR[rd]_{15..0} \leftarrow 0x8000$

else

$GPR[rd]_{15..0} \leftarrow (GPR[rs]_{15..0} + GPR[rt]_{15..0})_{15..0}$

endif

if $((GPR[rs]_{31..16} + GPR[rt]_{31..16}) > 0x7FFF)$ then

$GPR[rd]_{31..16} \leftarrow 0x7FFF$

else if $(0x10000 \leq (GPR[rs]_{31..16} + GPR[rt]_{31..16}) < 0x18000)$ then

$GPR[rd]_{31..16} \leftarrow 0x8000$

else

$GPR[rd]_{31..16} \leftarrow (GPR[rs]_{31..16} + GPR[rt]_{31..16})_{15..0}$

endif

(The same operations follow every 16 bits)

if $((GPR[rs]_{127..112} + GPR[rt]_{127..112}) > 0x7FFF)$ then

$GPR[rd]_{127..112} \leftarrow 0x7FFF$

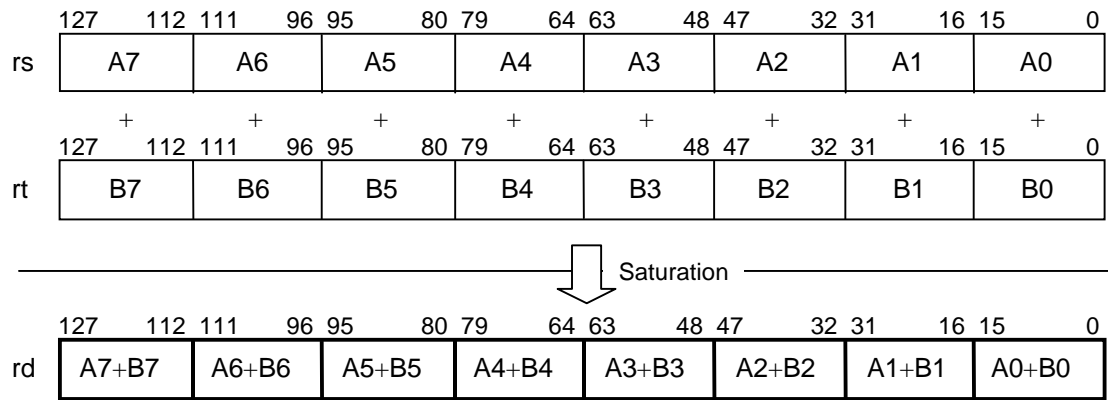
else if $(0x10000 \leq (GPR[rs]_{127..112} + GPR[rt]_{127..112}) < 0x18000)$ then

$GPR[rd]_{127..112} \leftarrow 0x8000$

else

$GPR[rd]_{127..112} \leftarrow (GPR[rs]_{127..112} + GPR[rt]_{127..112})_{15..0}$

endif



PADDSW : Parallel Add with Signed saturation Word

128-bit MMI

To add 4 pairs of 32-bit signed integers with saturation in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PADDSW	MMI0						
011100				10000	001000						
6	5	5	5	5	6						

Format

PADDSW rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] + GPR[rt]$

Splits the 128-bit values in GPR[rs] and GPR[rt] into four 32-bit signed integers, adds the data in GPR[rs] to the corresponding data in GPR[rt] and stores them in the corresponding words in GPR[rd].

Arithmetic results beyond the range of a signed 32-bit integer are saturated as follows;

Overflow : $\rightarrow 0x7FFFFFFF$

Underflow : $\rightarrow 0x80000000$

Exceptions

None

Operation

if $((GPR[rs]_{31..0} + GPR[rt]_{31..0}) > 0x7FFFFFFF)$ then

$GPR[rd]_{31..0} \leftarrow 0x7FFFFFFF$

else if $(0x100000000 \leq (GPR[rs]_{31..0} + GPR[rt]_{31..0}) < 0x800000000)$ then

$GPR[rd]_{31..0} \leftarrow 0x80000000$

else

$GPR[rd]_{31..0} \leftarrow (GPR[rs]_{31..0} + GPR[rt]_{31..0})_{31..0}$

endif

if $((GPR[rs]_{63..32} + GPR[rt]_{63..32}) > 0x7FFFFFFF)$ then

$GPR[rd]_{63..32} \leftarrow 0x7FFFFFFF$

else if $(0x100000000 \leq (GPR[rs]_{63..32} + GPR[rt]_{63..32}) < 0x800000000)$ then

$GPR[rd]_{63..32} \leftarrow 0x80000000$

else

$GPR[rd]_{63..32} \leftarrow (GPR[rs]_{63..32} + GPR[rt]_{63..32})_{31..0}$

endif

if $((GPR[rs]_{95..64} + GPR[rt]_{95..64}) > 0x7FFFFFFF)$ then

$GPR[rd]_{95..64} \leftarrow 0x7FFFFFFF$

else if $(0x100000000 \leq (GPR[rs]_{95..64} + GPR[rt]_{95..64}) < 0x800000000)$ then

$GPR[rd]_{95..64} \leftarrow 0x80000000$

else

$GPR[rd]_{95..64} \leftarrow (GPR[rs]_{95..64} + GPR[rt]_{95..64})_{31..0}$

endif

if $((GPR[rs]_{127..96} + GPR[rt]_{127..96}) > 0x7FFFFFFF)$ then

$GPR[rd]_{127..96} \leftarrow 0x7FFFFFFF$

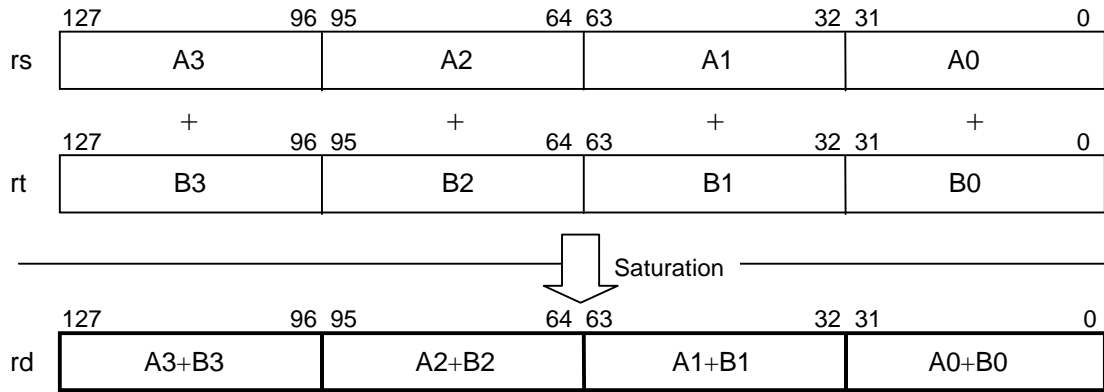
else if $(0x100000000 \leq (GPR[rs]_{127..96} + GPR[rt]_{127..96}) < 0x800000000)$ then

$GPR[rd]_{127..96} \leftarrow 0x80000000$

```

else
    GPR[rd]127..96 ← (GPR[rs]127..96 + GPR[rt]127..96)31..0
endif

```



PADDUB : Parallel Add with Unsigned saturation Byte

128-bit MMI

To add 16 pairs of 8-bit unsigned integers with saturation in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PADDUB	MMI1						
011100				11000	101000						
6	5	5	5	5	6						

Format

PADDUB rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] + GPR[rt]$

Splits the 128-bit values in GPR[rs] and GPR[rt] into sixteen 8-bit unsigned integers, adds the data in GPR[rs] to the corresponding data in GPR[rt] and stores them in the corresponding bytes in GPR[rd].

Arithmetic results beyond the range of an unsigned 8-bit integer are saturated as follows;

Overflow : $\rightarrow 0xFF$

Exceptions

None

Operation

if $((GPR[rs]_{7..0} + GPR[rt]_{7..0}) > 0xFF)$ then

$GPR[rd]_{7..0} \leftarrow 0xFF$

else

$GPR[rd]_{7..0} \leftarrow (GPR[rs]_{7..0} + GPR[rt]_{7..0})_{7..0}$

endif

if $((GPR[rs]_{15..8} + GPR[rt]_{15..8}) > 0xFF)$ then

$GPR[rd]_{15..8} \leftarrow 0xFF$

else

$GPR[rd]_{15..8} \leftarrow (GPR[rs]_{15..8} + GPR[rt]_{15..8})_{7..0}$

endif

(The same operations follow every 8 bits)

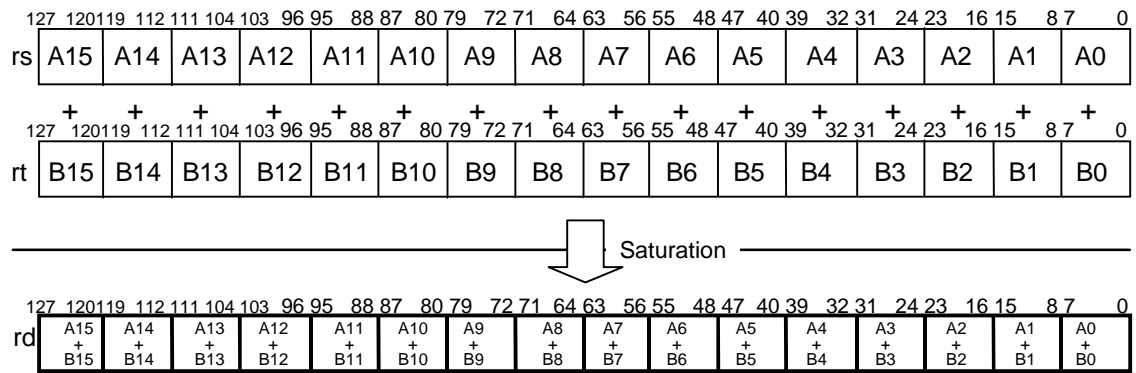
if $((GPR[rs]_{127..120} + GPR[rt]_{127..120}) > 0xFF)$ then

$GPR[rd]_{127..120} \leftarrow 0xFF$

else

$GPR[rd]_{127..120} \leftarrow (GPR[rs]_{127..120} + GPR[rt]_{127..120})_{7..0}$

endif



PADDUH : Parallel Add with Unsigned saturation Halfword

128-bit MMI

To add 8 pairs of 16-bit unsigned integers with saturation in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PADDUH	MMI1						
011100				10100	101000						
6	5	5	5	5	6						

Format

PADDUH rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] + GPR[rt]$

Splits the 128-bit values in GPR[rs] and GPR[rt] into eight 16-bit unsigned integers, adds the data in GPR[rs] to the corresponding data in GPR[rt] and stores them in the corresponding bytes in GPR[rd].

Arithmetic results beyond the range of an unsigned 16-bit integer are saturated as follows;

Overflow : $\rightarrow 0xFFFF$

Exceptions

None

Operation

if $((GPR[rs]_{15..0} + GPR[rt]_{15..0}) > 0xFFFF)$ then

$GPR[rd]_{15..0} \leftarrow 0xFFFF$

else

$GPR[rd]_{15..0} \leftarrow (GPR[rs]_{15..0} + GPR[rt]_{15..0})_{15..0}$

endif

if $((GPR[rs]_{31..16} + GPR[rt]_{31..16}) > 0xFFFF)$ then

$GPR[rd]_{31..16} \leftarrow 0xFFFF$

else

$GPR[rd]_{31..16} \leftarrow (GPR[rs]_{31..16} + GPR[rt]_{31..16})_{15..0}$

endif

(The same operations follow every 16 bits)

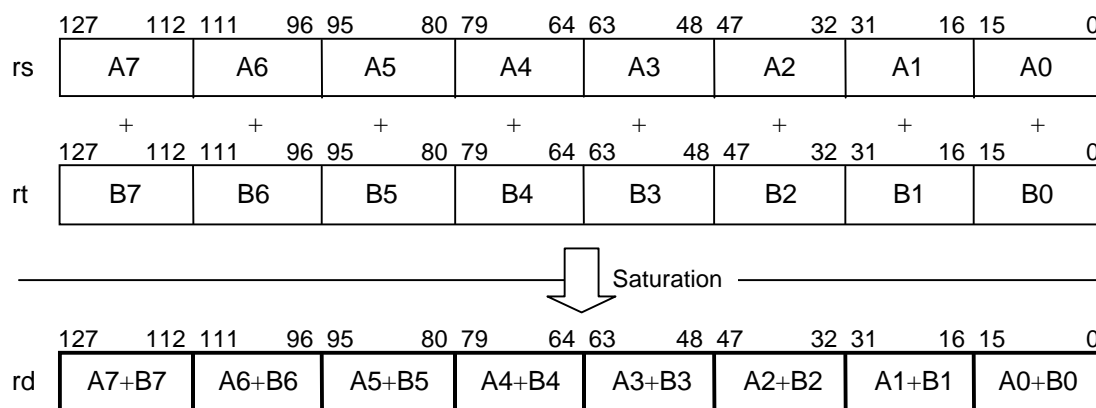
if $((GPR[rs]_{127..112} + GPR[rt]_{127..112}) > 0xFFFF)$ then

$GPR[rd]_{127..112} \leftarrow 0xFFFF$

else

$GPR[rd]_{127..112} \leftarrow (GPR[rs]_{127..112} + GPR[rt]_{127..112})_{15..0}$

endif



PADDUW : Parallel Add with Unsigned saturation Word

128-bit MMI

To add 4 pairs of 32-bit unsigned integers with saturation in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI		rs		rt		rd		PADDUW		MMI1	
011100								10000		101000	
6		5		5		5		5		6	

Format

PADDUW rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] + GPR[rt]$

Splits the 128-bit values in GPR[rs] and GPR[rt] into four 32-bit unsigned integers, adds the data in GPR[rs] to the corresponding data in GPR[rt] and stores them in the corresponding words in GPR[rd].

Arithmetic results beyond the range of an unsigned 32-bit integer are saturated as follows;

Overflow : $\rightarrow 0xFFFFFFFF$

Exceptions

None

Operation

if $((GPR[rs]_{31..0} + GPR[rt]_{31..0}) > 0xFFFFFFFF)$ then

$GPR[rd]_{31..0} \leftarrow 0xFFFFFFFF$

else

$GPR[rd]_{31..0} \leftarrow (GPR[rs]_{31..0} + GPR[rt]_{31..0})_{31..0}$

endif

if $((GPR[rs]_{63..32} + GPR[rt]_{63..32}) > 0xFFFFFFFF)$ then

$GPR[rd]_{63..32} \leftarrow 0xFFFFFFFF$

else

$GPR[rd]_{63..32} \leftarrow (GPR[rs]_{63..32} + GPR[rt]_{63..32})_{31..0}$

endif

if $((GPR[rs]_{95..64} + GPR[rt]_{95..64}) > 0xFFFFFFFF)$ then

$GPR[rd]_{95..64} \leftarrow 0xFFFFFFFF$

else

$GPR[rd]_{95..64} \leftarrow (GPR[rs]_{95..64} + GPR[rt]_{95..64})_{31..0}$

endif

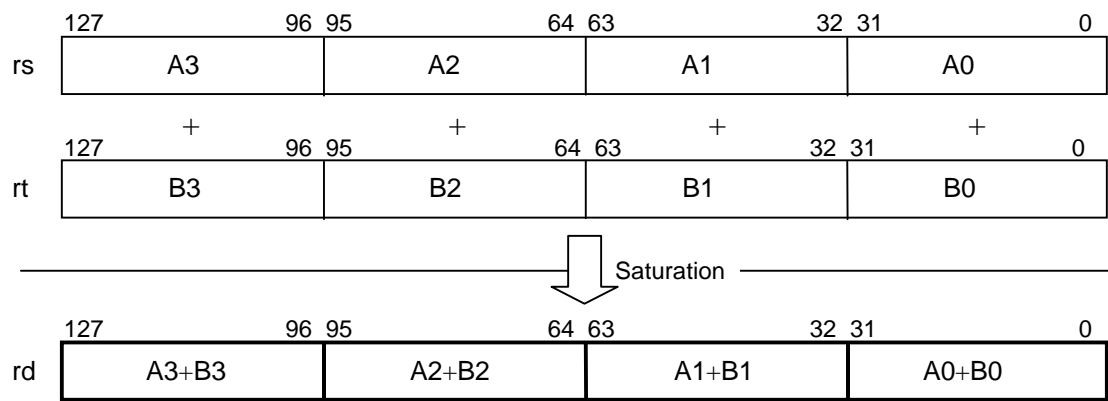
if $((GPR[rs]_{127..96} + GPR[rt]_{127..96}) > 0xFFFFFFFF)$ then

$GPR[rd]_{127..96} \leftarrow 0xFFFFFFFF$

else

$GPR[rd]_{127..96} \leftarrow (GPR[rs]_{127..96} + GPR[rt]_{127..96})_{31..0}$

endif



PADDW : Parallel Add Word

128-bit MMI

To add 4 pairs of 32-bit integers in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PADDW	MMI0						
011100				00000	001000						
6	5	5	5	5	6						

Format

PADDW rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] + GPR[rt]$

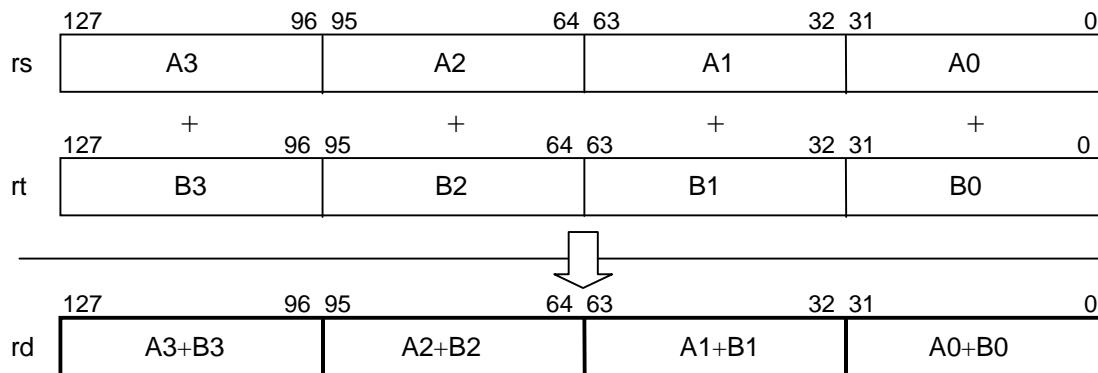
Splits the 128-bit values in GPR[rs] and GPR[rt] into four 32-bit integers, adds the data in GPR[rs] to the corresponding data in GPR[rt] and stores them in the corresponding words in GPR[rd].

Exceptions

None

Operation

$GPR[rd]_{31..0} \leftarrow (GPR[rs]_{31..0} + GPR[rt]_{31..0})_{31..0}$
 $GPR[rd]_{63..32} \leftarrow (GPR[rs]_{63..32} + GPR[rt]_{63..32})_{31..0}$
 $GPR[rd]_{95..64} \leftarrow (GPR[rs]_{95..64} + GPR[rt]_{95..64})_{31..0}$
 $GPR[rd]_{127..96} \leftarrow (GPR[rs]_{127..96} + GPR[rt]_{127..96})_{31..0}$



PADSBH : Parallel Add/Subtract Halfword

128-bit MMI

To add/subtract 8 pairs of 16-bit integers in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PADSBH	MMI1						
011100				00100	101000						
6	5	5	5	5	6						

Format

PADSBH rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] \pm GPR[rt]$

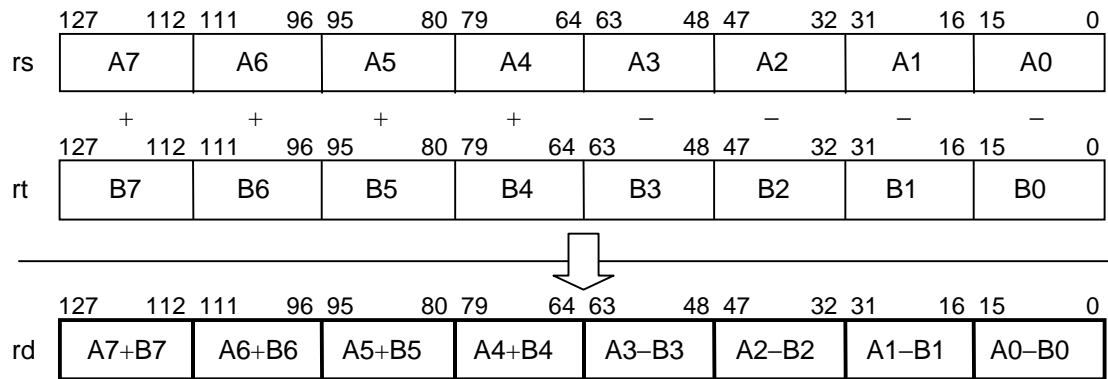
Splits the 128-bit values in GPR[rs] and GPR[rt] into eight 16-bit integers, adds the high-order four pairs and subtracts the low-order four pairs, and stores them in the corresponding halfwords in GPR[rd].

Exceptions

None. When it overflows or underflows, simply ignored and an exception do not occur.

Operation

$GPR[rd]_{15..0} \leftarrow (GPR[rs]_{15..0} - GPR[rt]_{15..0})_{15..0}$
 $GPR[rd]_{31..16} \leftarrow (GPR[rs]_{31..16} - GPR[rt]_{31..16})_{15..0}$
 $GPR[rd]_{47..32} \leftarrow (GPR[rs]_{47..32} - GPR[rt]_{47..32})_{15..0}$
 $GPR[rd]_{63..48} \leftarrow (GPR[rs]_{63..48} - GPR[rt]_{63..48})_{15..0}$
 $GPR[rd]_{79..64} \leftarrow (GPR[rs]_{79..64} + GPR[rt]_{79..64})_{15..0}$
 $GPR[rd]_{95..80} \leftarrow (GPR[rs]_{95..80} + GPR[rt]_{95..80})_{15..0}$
 $GPR[rd]_{111..96} \leftarrow (GPR[rs]_{111..96} + GPR[rt]_{111..96})_{15..0}$
 $GPR[rd]_{127..112} \leftarrow (GPR[rs]_{127..112} + GPR[rt]_{127..112})_{15..0}$



PAND : Parallel And

128-bit MMI

To calculate a bitwise logical AND.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI 011100						rs			rt		
						rd			PAND 10010		
									MMI2 001001		
6						5			5		

Format

PAND rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] \text{ AND } GPR[rt]$

Calculates a bitwise logical AND between the 128-bit values of GPR[rs] and GPR[rt]. The result is stored in GPR[rd].

The truth table values for AND are as follows;

X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

Exceptions

None

Operation

$GPR[rd]_{127..0} \leftarrow GPR[rs]_{127..0} \text{ AND } GPR[rt]_{127..0}$

PCEQB : Parallel Compare for Equal Byte

128-bit MMI

To compare 16 pairs of byte data in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PCEQB	MMI1						
011100				01010	101000						
6	5	5	5	5	6						

Format

PCEQB rd, rs, rt

Description

$GPR[rd] \leftarrow (GPR[rs] = GPR[rt])$

Splits the 128-bit values in GPR[rs] and GPR[rt] into sixteen bytes and compares the data in GPR[rs] with the corresponding data in GPR[rt]. If the results are equal, stores 0xFF and if not equal, stores 0x00 in the corresponding bytes in GPR[rd].

Exceptions

None

Operation

if ($GPR[rs]_{7..0} = GPR[rt]_{7..0}$) then

$GPR[rd]_{7..0} \leftarrow 1^8$

else

$GPR[rd]_{7..0} \leftarrow 0^8$

endif

if ($GPR[rs]_{15..8} = GPR[rt]_{15..8}$) then

$GPR[rd]_{15..8} \leftarrow 1^8$

else

$GPR[rd]_{15..8} \leftarrow 0^8$

endif

(The same operations follow every 8 bits)

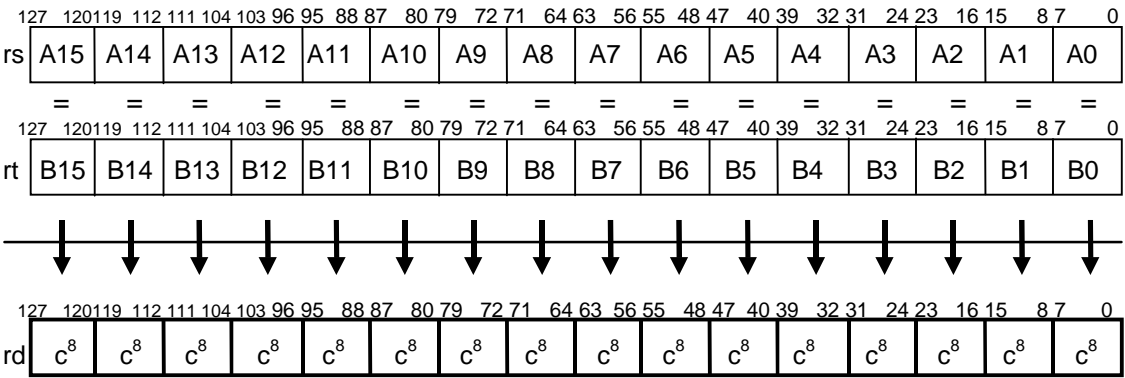
if ($GPR[rs]_{127..120} = GPR[rt]_{127..120}$) then

$GPR[rd]_{127..120} \leftarrow 1^8$

else

$GPR[rd]_{127..120} \leftarrow 0^8$

endif



PCEQH : Parallel Compare for Equal Halfword

128-bit MMI

To compare 8 pairs of halfword data in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PCEQH	MMI1						
011100				00110	101000						
6	5	5	5	5	6						

Format

PCEQH rd, rs, rt

Description

$GPR[rd] \leftarrow (GPR[rs] = GPR[rt])$

Splits the 128-bit values in GPR[rs] and GPR[rt] into eight halfwords and compares the data in GPR[rs] with the corresponding data in GPR[rt]. If the results are equal, stores 0xFFFF and if not equal, stores 0x0000 in the corresponding halfwords in GPR[rd].

Exceptions

None

Operation

if ($GPR[rs]_{15..0} = GPR[rt]_{15..0}$) then

$GPR[rd]_{15..0} \leftarrow 1^{16}$

else

$GPR[rd]_{15..0} \leftarrow 0^{16}$

endif

if ($GPR[rs]_{31..16} = GPR[rt]_{31..16}$) then

$GPR[rd]_{31..16} \leftarrow 1^{16}$

else

$GPR[rd]_{31..16} \leftarrow 0^{16}$

endif

(The same operations follow every 16 bits)

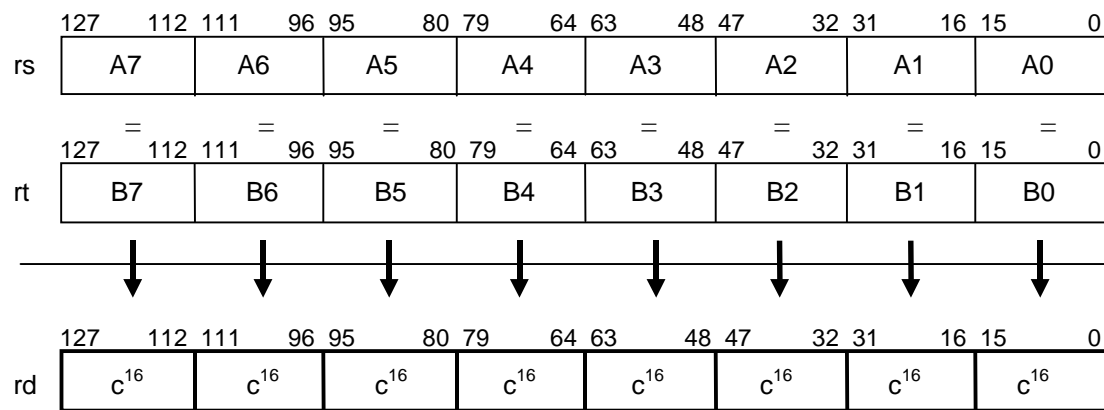
if ($GPR[rs]_{127..112} = GPR[rt]_{127..112}$) then

$GPR[rd]_{127..112} \leftarrow 1^{16}$

else

$GPR[rd]_{127..112} \leftarrow 0^{16}$

endif



PCEQW : Parallel Compare for Equal Word

128-bit MMI

To compare 4 pairs of word data in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PCEQW	MMI1						
011100				00010	101000						
6	5	5	5	5	6						

Format

PCEQW rd, rs, rt

Description

$GPR[rd] \leftarrow (GPR[rs] = GPR[rt])$

Splits the 128-bit values in GPR[rs] and GPR[rt] into four words and compares the data in GPR[rs] with the corresponding data in GPR[rt]. If the results are equal, stores 0xFFFFFFFF and if not equal, stores 0x00000000 in the corresponding words in GPR[rd].

Exceptions

None

Operation

if ($GPR[rs]_{31..0} = GPR[rt]_{31..0}$) then

$GPR[rd]_{31..0} \leftarrow 1^{32}$

else

$GPR[rd]_{31..0} \leftarrow 0^{32}$

endif

if ($GPR[rs]_{63..32} = GPR[rt]_{63..32}$) then

$GPR[rd]_{63..32} \leftarrow 1^{32}$

else

$GPR[rd]_{63..32} \leftarrow 0^{32}$

endif

if ($GPR[rs]_{95..64} = GPR[rt]_{95..64}$) then

$GPR[rd]_{95..64} \leftarrow 1^{32}$

else

$GPR[rd]_{95..64} \leftarrow 0^{32}$

endif

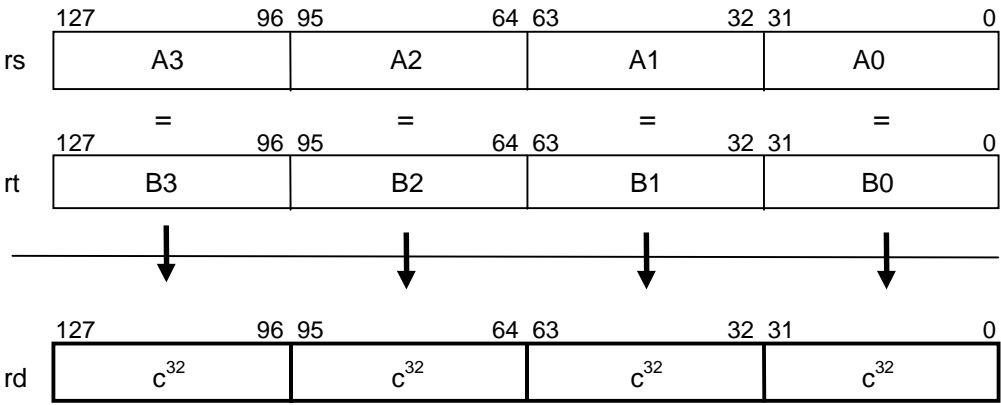
if ($GPR[rs]_{127..96} = GPR[rt]_{127..96}$) then

$GPR[rd]_{127..96} \leftarrow 1^{32}$

else

$GPR[rd]_{127..96} \leftarrow 0^{32}$

endif



PCGTB : Parallel Compare for Greater Than Byte

128-bit MMI

To compare 16 pairs of byte data in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI			rs		rt		rd		PCGTB		MMI0
011100									01010		001000
6			5		5		5		5		6

Format

PCGTB rd, rs, rt

Description

$GPR[rd] \leftarrow (GPR[rs] > GPR[rt])$

Splits the 128-bit values in GPR[rs] and GPR[rt] into sixteen 8-bit signed integers and compares the data in GPR[rs] with the corresponding data in GPR[rt]. If GPR[rs] is greater than GPR[rt], stores 0xFF and otherwise, stores 0x00 in the corresponding bytes in GPR[rd].

Exceptions

None

Operation

if ($GPR[rs]_{7..0} > GPR[rt]_{7..0}$) then

$GPR[rd]_{7..0} \leftarrow 1^8$

else

$GPR[rd]_{7..0} \leftarrow 0^8$

endif

if ($GPR[rs]_{15..8} > GPR[rt]_{15..8}$) then

$GPR[rd]_{15..8} \leftarrow 1^8$

else

$GPR[rd]_{15..8} \leftarrow 0^8$

endif

(The same operations follow every 8 bits)

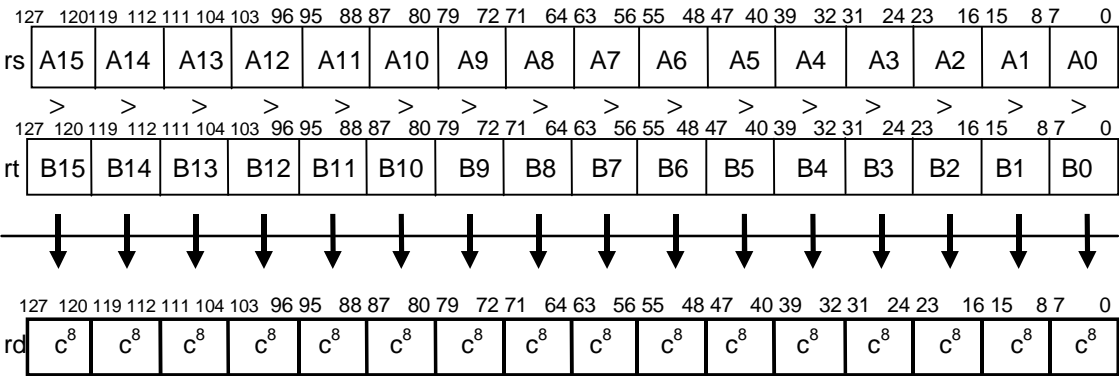
if ($GPR[rs]_{127..120} > GPR[rt]_{127..120}$) then

$GPR[rd]_{127..120} \leftarrow 1^8$

else

$GPR[rd]_{127..120} \leftarrow 0^8$

endif



PCGTH : Parallel Compare for Greater Than Halfword

128-bit MMI

To compare 8 pairs of halfword data in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI 011100			rs		rt		rd		PCGTH 00110		MMI0 001000
6			5		5		5		5		6

Format

PCGTH rd, rs, rt

Description

$GPR[rd] \leftarrow (GPR[rs] > GPR[rt])$

Splits the 128-bit values in GPR[rs] and GPR[rt] into eight 16-bit signed integers and compares the data in GPR[rs] with the corresponding data in GPR[rt]. If GPR[rs] is greater than GPR[rt], stores 0xFFFF and otherwise, stores 0x0000 in the corresponding halfwords in GPR[rd].

Exceptions

None

Operation

if ($GPR[rs]_{15..0} > GPR[rt]_{15..0}$) then

$GPR[rd]_{15..0} \leftarrow 1^{16}$

else

$GPR[rd]_{15..0} \leftarrow 0^{16}$

endif

if ($GPR[rs]_{31..16} > GPR[rt]_{31..16}$) then

$GPR[rd]_{31..16} \leftarrow 1^{16}$

else

$GPR[rd]_{31..16} \leftarrow 0^{16}$

endif

(The same operations follow every 16 bits)

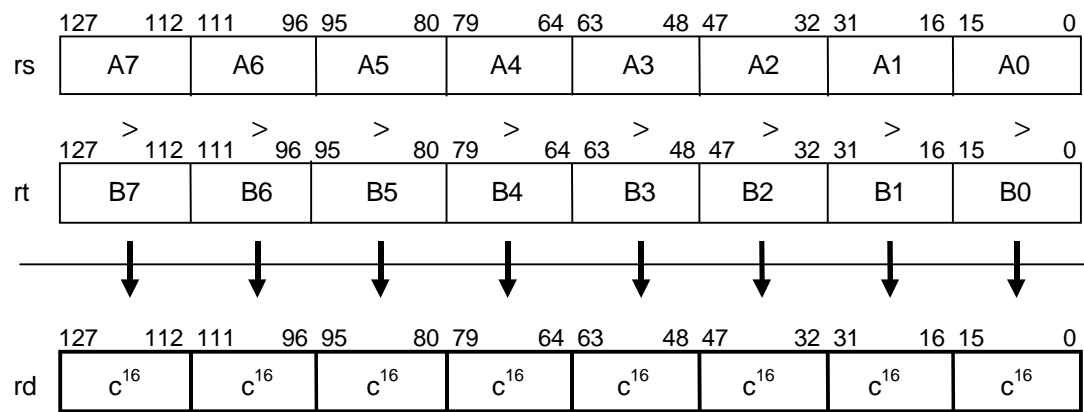
if ($GPR[rs]_{127..112} > GPR[rt]_{127..112}$) then

$GPR[rd]_{127..112} \leftarrow 1^{16}$

else

$GPR[rd]_{127..112} \leftarrow 0^{16}$

endif



PCGTW : Parallel Compare for Greater Than Word

128-bit MMI

To compare 4 pairs of word data in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PCGTW	MMI0						
011100				00010	001000						
6	5	5	5	5	6						

Format

PCGTW rd, rs, rt

Description

$GPR[rd] \leftarrow (GPR[rs] > GPR[rt])$

Splits the 128-bit values in GPR[rs] and GPR[rt] into four 32-bit signed integers and compares the data in GPR[rs] with the corresponding data in GPR[rt]. If GPR[rs] is greater than GPR[rt], stores 0xFFFFFFFF, and otherwise stores 0x00000000 in the corresponding words in GPR[rd].

Exceptions

None

Operation

if ($GPR[rs]_{31..0} > GPR[rt]_{31..0}$) then

$GPR[rd]_{31..0} \leftarrow 1^{32}$

else

$GPR[rd]_{31..0} \leftarrow 0^{32}$

endif

if ($GPR[rs]_{63..32} > GPR[rt]_{63..32}$) then

$GPR[rd]_{63..32} \leftarrow 1^{32}$

else

$GPR[rd]_{63..32} \leftarrow 0^{32}$

endif

if ($GPR[rs]_{95..64} > GPR[rt]_{95..64}$) then

$GPR[rd]_{95..64} \leftarrow 1^{32}$

else

$GPR[rd]_{95..64} \leftarrow 0^{32}$

endif

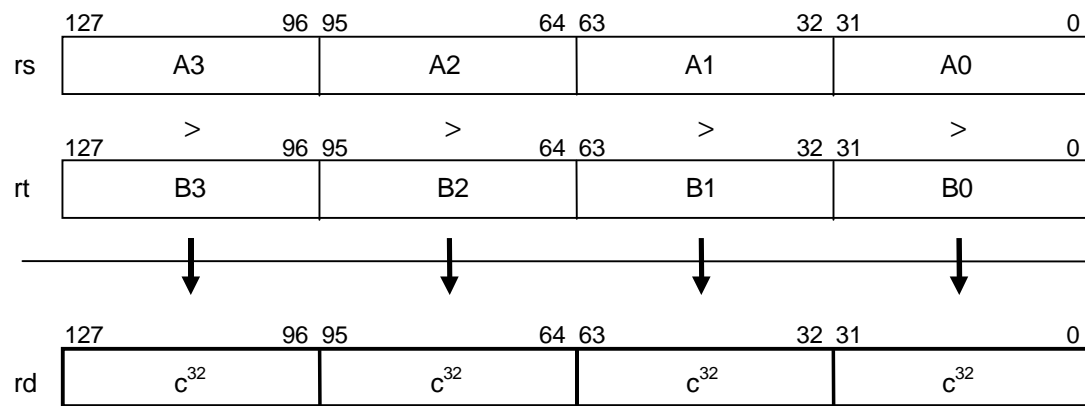
if ($GPR[rs]_{127..96} > GPR[rt]_{127..96}$) then

$GPR[rd]_{127..96} \leftarrow 1^{32}$

else

$GPR[rd]_{127..96} \leftarrow 0^{32}$

endif



PCPYH : Parallel Copy Halfword

128-bit MMI

To copy halfword data in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	0	rt	rd	PCPYH	MMI3						
011100	00000			11011	101001						
6	5	5	5	5	6						

Format

PCPYH rd, rt

Description

$GPR[rd] \leftarrow GPR[rt]$

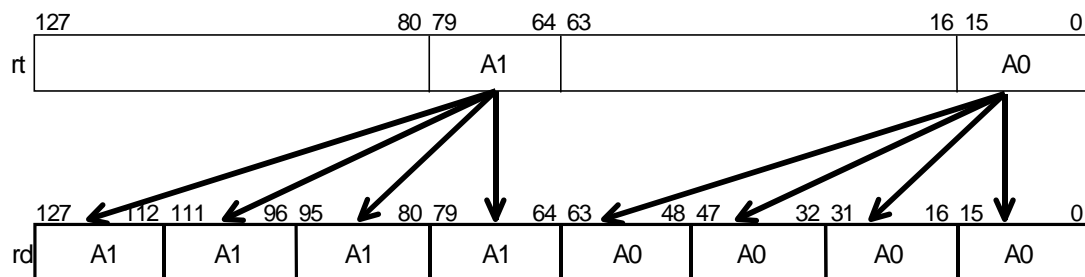
Splits $GPR[rt]$ into the high-order and low-order 64 bits. Copies each of the least-significant halfwords into each of the halfwords of the two doublewords of $GPR[rd]$.

Exceptions

None

Operation

$GPR[rd]_{15..0} \leftarrow GPR[rt]_{15..0}$
 $GPR[rd]_{31..16} \leftarrow GPR[rt]_{15..0}$
 $GPR[rd]_{47..32} \leftarrow GPR[rt]_{15..0}$
 $GPR[rd]_{63..48} \leftarrow GPR[rt]_{15..0}$
 $GPR[rd]_{79..64} \leftarrow GPR[rt]_{79..64}$
 $GPR[rd]_{95..80} \leftarrow GPR[rt]_{79..64}$
 $GPR[rd]_{111..96} \leftarrow GPR[rt]_{79..64}$
 $GPR[rd]_{127..112} \leftarrow GPR[rt]_{79..64}$



PCPYLD : Parallel Copy Lower Doubleword

128-bit MMI

To combine 2 doublewords.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PCPYLD	MMI2						
011100				01110	001001						
6	5	5	5	5	6						

Format

PCPYLD rd, rs, rt

Description

$GPR[rd] \leftarrow \text{copy}(GPR[rs], GPR[rt])$

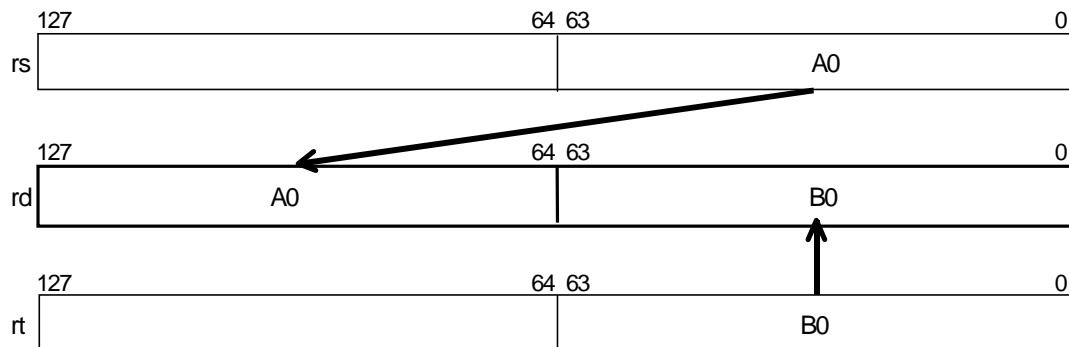
To calculate a 128-bit value, in which the high-order and low-order 64 bits correspond to the low-order 64 bits of GPR[rs] and low-order 64 bits in GPR[rt] respectively, and stores it in GPR[rd].

Exceptions

None

Operation

$GPR[rd]_{127:0} \leftarrow GPR[rs]_{63:0} \parallel GPR[rt]_{63:0}$



PCPYUD : Parallel Copy Upper Doubleword

128-bit MMI

To combine 2 doublewords.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PCPYUD	MMI3						
011100				01110	101001						
6	5	5	5	5	6						

Format

PCPYUD rd, rs, rt

Description

$GPR[rd] \leftarrow \text{copy}(GPR[rs], GPR[rt])$

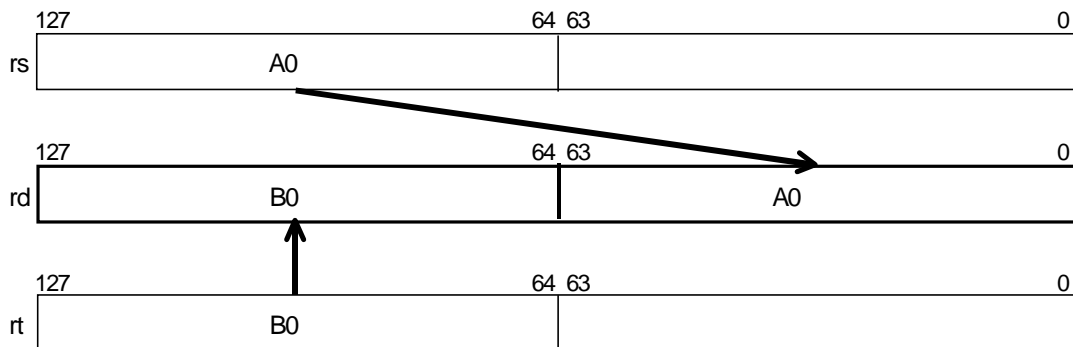
To calculate a 128-bit value, in which the low-order and high-order 64 bits correspond to the high-order 64 bits of GPR[rs] and high-order 64 bits of GPR[rt] respectively, and stores it in GPR[rd].

Exceptions

None

Operation

$GPR[rd]_{127..0} \leftarrow GPR[rt]_{127..64} \parallel GPR[rs]_{127..64}$



PDIVBW : Parallel Divide Broadcast Word

128-bit MMI

To divide 4 32-bit signed integers by a 16-bit signed integer in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	0	PDIVBW	MMI2						
011100			00000	11101	001001						
6	5	5	5	5	6						

Format

PDIVBW rs, rt

Description

$(LO, HI) \leftarrow GPR[rs] / GPR[rt]$

Splits GPR[rs] into four signed 32-bit integers and divides each of them by the least-significant halfword of GPR[rt]. The resulting four quotients (32-bit integers) are stored in the words corresponding to GPR[rs] in the LO register and the four remainders (16-bit integers) are zero-extended and stored in the words corresponding to GPR[rs] in the HI register.

Restrictions

If the least-significant halfword in GPR[rt] is zero, the arithmetic result is undefined. (An exception does not occur.)

Exceptions

None. If the divisor is 0, an exception does not occur on overflow.

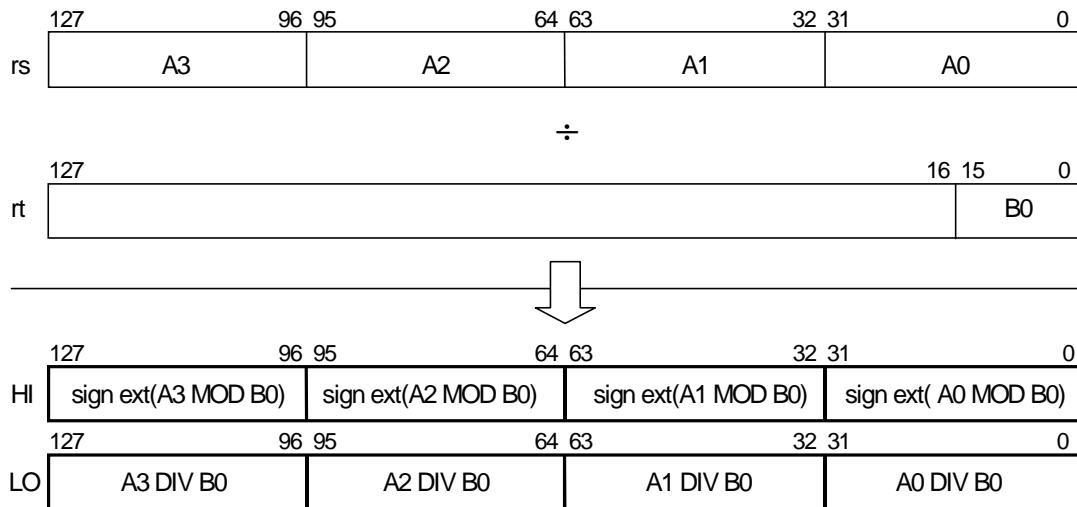
Operation

$q0 \leftarrow GPR[rs]_{31..0} \text{ DIV } GPR[rt]_{15..0}$
 $r0 \leftarrow GPR[rs]_{31..0} \text{ MOD } GPR[rt]_{15..0}$
 $LO_{31..0} \leftarrow q0_{31..0}$
 $HI_{31..0} \leftarrow (r0_{15})^{16} \parallel r0_{15..0}$

$q1 \leftarrow GPR[rs]_{63..32} \text{ DIV } GPR[rt]_{15..0}$
 $r1 \leftarrow GPR[rs]_{63..32} \text{ MOD } GPR[rt]_{15..0}$
 $LO_{63..32} \leftarrow q1_{31..0}$
 $HI_{63..32} \leftarrow (r1_{15})^{16} \parallel r1_{15..0}$

$q2 \leftarrow GPR[rs]_{95..64} \text{ div } GPR[rt]_{15..0}$
 $r2 \leftarrow GPR[rs]_{95..64} \text{ mod } GPR[rt]_{15..0}$
 $LO_{95..64} \leftarrow q2_{31..0}$
 $HI_{95..64} \leftarrow (r2_{15})^{16} \parallel r2_{15..0}$

$q3 \leftarrow GPR[rs]_{127..96} \text{ div } GPR[rt]_{15..0}$
 $r3 \leftarrow GPR[rs]_{127..96} \text{ mod } GPR[rt]_{15..0}$
 $LO_{127..96} \leftarrow q3_{31..0}$
 $HI_{127..96} \leftarrow (r3_{15})^{16} \parallel r3_{15..0}$



Programming Notes

In the EE Core, the integer divide operation proceeds asynchronously. An attempt to read the contents of the LO or HI register before the divide operation finishes will result in interlock. Other CPU instructions can execute without delay. Therefore, scheduling the divide operation appropriately can improve performance.

When 0x80000000 (−2147483648), the signed minimum value, is divided by 0xFFFF (−1), the operation will result in an overflow. However, in this instruction an overflow exception does not occur and the following results will be returned.

Quotient: 0x80000000 (−2147483648), Remainder: 0x00000000 (0)

If an overflow or divide-by-zero is required to be detected, then add an instruction that detects these conditions following the divide instruction. Since the divide instruction is asynchronous, the divide operation and check can be executed in parallel. If an overflow or divide-by-zero is detected, then the system software can be informed of the problem by generating an exception using an appropriate code value with a BREAK instruction.

PDIVUW : Parallel Divide Unsigned Word

128-bit MMI

To divide 2 pairs of 32-bit unsigned integers in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	0	PDIVUW	MMI3						
011100			00000	01101	101001						
6	5	5	5	5	6						

Format

PDIVUW rs, rt

Description

$(LO, HI) \leftarrow GPR[rs] / GPR[rt]$

Divides bits 31..0 of GPR[rs] by bits 31..0 of GPR[rt]. Both are treated as 32-bit unsigned integers. The resulting quotients and remainders are sign-extended and stored in bits 63..0 of the LO and HI registers respectively. Similarly, divides bits 95..64 of GPR[rs] by bits 95..64 of GPR[rt] and stores the results in bits 127..64 of LO and HI respectively.

Restrictions

If GPR[rt] and GPR[rs] are not sign-extended 32-bit values (bits 127..95 equal and bits 63..31 equal), then the result is undefined.

If the divisor is 0, an exception does not occur on overflow.

Exceptions

None. Even if the divisor is zero, a divide-by-zero exception does not occur.

Operation

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif

$q0 \leftarrow (0 \parallel GPR[rs]_{31..0}) \text{ DIV } (0 \parallel GPR[rt]_{31..0})$

$r0 \leftarrow (0 \parallel GPR[rs]_{31..0}) \text{ MOD } (0 \parallel GPR[rt]_{31..0})$

$LO_{63..0} \leftarrow (q0_{31})^{32} \parallel q0_{31..0}$

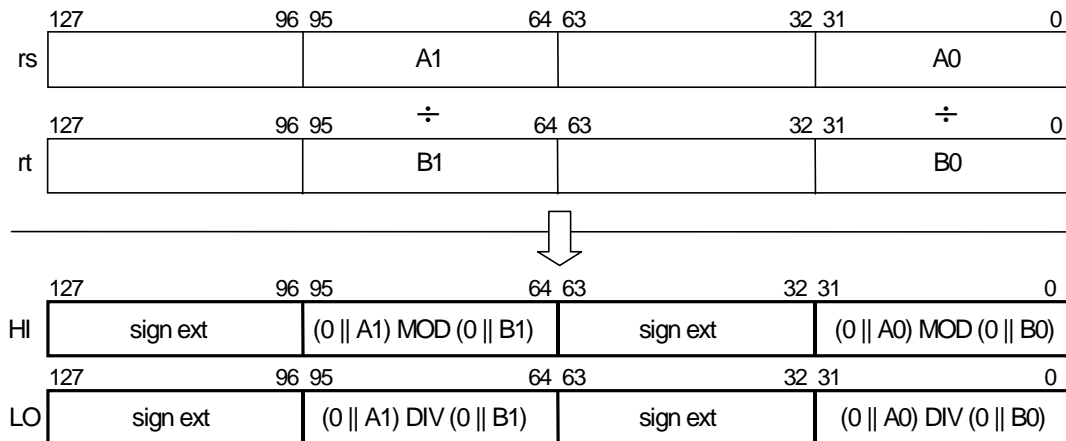
$HI_{63..0} \leftarrow (r0_{31})^{32} \parallel r0_{31..0}$

$q1 \leftarrow (0 \parallel GPR[rs]_{95..64}) \text{ DIV } (0 \parallel GPR[rt]_{95..64})$

$r1 \leftarrow (0 \parallel GPR[rs]_{95..64}) \text{ MOD } (0 \parallel GPR[rt]_{95..64})$

$LO_{127..64} \leftarrow (q1_{31})^{32} \parallel q1_{31..0}$

$HI_{127..64} \leftarrow (r1_{31})^{32} \parallel r1_{31..0}$



Programming Notes

In the EE Core, the integer divide operation proceeds asynchronously. An attempt to read the contents of the LO or HI register before the divide operation finishes will result in interlock. Other CPU instructions can execute without delay. Therefore, scheduling the divide operation appropriately can improve performance.

If divide-by-zero is required to be detected, then add an instruction that detects this condition following the divide instruction. Since the divide instruction is asynchronous, the divide operation and check can be executed in parallel. If divide-by-zero is detected, then the system software can be informed of the problem by generating an exception using an appropriate code value with a BREAK instruction.

PDIVW : Parallel Divide Word

128-bit MMI

To divide 2 pairs of 32-bit signed integers in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	0	PDIVW	MMI2						
011100			00000	01101	001001						
6	5	5	5	5	6						

Format

PDIVW rs, rt

Description

$(LO, HI) \leftarrow GPR[rs] / GPR[rt]$

Divides bits 31..0 of GPR[rs] by bits 31..0 of GPR[rt]. Both are treated as 32-bit signed integers. The resulting quotients and remainders are sign-extended and stored in bits 63..0 of the LO and HI registers respectively. Similarly, divides bits 95..64 of GPR[rs] by bits 95..64 of GPR[rt] and stores the results in bits 127..64 of LO and HI, respectively.

Restrictions

If GPR[rt] and GPR[rs] are not sign-extended 32-bit values (bits 127..95 equal and bits 63..31 equal), then the result is undefined.

If the divisor is 0, an exception does not occur on overflow. (An exception does not occur.)

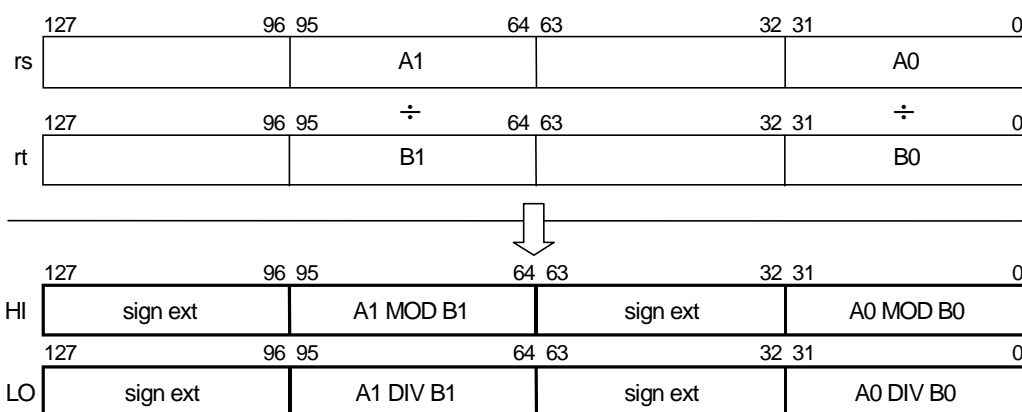
Exceptions

None. If the divisor is zero, an exception does not occur when the arithmetic result overflows.

Operation

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif

$q0 \leftarrow GPR[rs]_{31..0} \text{ DIV } GPR[rt]_{31..0}$
 $r0 \leftarrow GPR[rs]_{31..0} \text{ MOD } GPR[rt]_{31..0}$
 $q1 \leftarrow GPR[rs]_{95..64} \text{ DIV } GPR[rt]_{95..64}$
 $r1 \leftarrow GPR[rs]_{95..64} \text{ MOD } GPR[rt]_{95..64}$
 $LO_{63..0} \leftarrow (q0_{31})^{32} \parallel q0_{31..0}$
 $HI_{63..0} \leftarrow (r0_{31})^{32} \parallel r0_{31..0}$
 $LO_{127..64} \leftarrow (q1_{31})^{32} \parallel q1_{31..0}$
 $HI_{127..64} \leftarrow (r1_{31})^{32} \parallel r1_{31..0}$



Programming Notes

In the EE Core, the integer divide operation proceeds asynchronously. An attempt to read the contents of the LO or HI register before the divide operation finishes will result in interlock. Other CPU instructions can execute without delay. Therefore, scheduling the divide operation appropriately can improve performance.

When 0x80000000(-2147483648), the signed minimum value, is divided by 0xFFFFFFFF(-1), the operation will result in an overflow. However, in this instruction an overflow exception does not occur and the following results will be returned;

Quotient: 0x80000000(-2147483648), Remainder: 0x00000000(0)

If an overflow or divide-by-zero is required to be detected, then add an instruction that detects these conditions following the divide instruction. Since the divide instruction is asynchronous, the divide operation and check can be executed in parallel. If an overflow or divide-by-zero is detected, then the system software can be informed of the problem by generating an exception using an appropriate code value with a BREAK instruction.

PEXCH : Parallel Exchange Center Halfword

128-bit MMI

To exchange the position of halfwords in 128-bit data.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	0	rt	rd	PEXCH	MMI3						
011100	00000			11010	101001						
6	5	5	5	5	6						

Format

PEXCH rd, rt

Description

$GPR[rd] \leftarrow \text{exchange}(GPR[rt])$

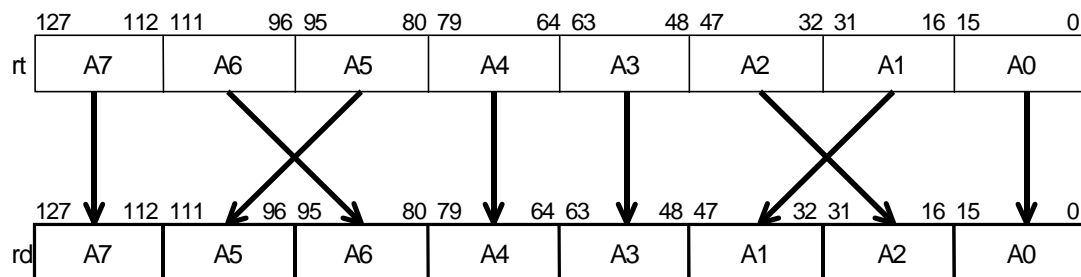
Splits the 128-bit data in GPR[rt] into eight halfwords, exchanges the central halfwords of each doubleword, and stores them in GPR[rd]. See "Operation" about the details of the exchange.

Exceptions

None

Operation

$GPR[rd]_{15..0} \leftarrow GPR[rt]_{15..0}$
 $GPR[rd]_{31..16} \leftarrow GPR[rt]_{47..32}$
 $GPR[rd]_{47..32} \leftarrow GPR[rt]_{31..16}$
 $GPR[rd]_{63..48} \leftarrow GPR[rt]_{63..48}$
 $GPR[rd]_{79..64} \leftarrow GPR[rt]_{79..64}$
 $GPR[rd]_{95..80} \leftarrow GPR[rt]_{111..96}$
 $GPR[rd]_{111..96} \leftarrow GPR[rt]_{95..80}$
 $GPR[rd]_{127..112} \leftarrow GPR[rt]_{127..112}$



PEXCW : Parallel Exchange Center Word

128-bit MMI

To exchange the position of words in 128-bit data.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI						0					
011100						00000					
6						5					
rt						rd					
5						5					
PEXCW						MMI3					
11110						101001					
5						6					

Format

PEXCW rd, rt

Description

$GPR[rd] \leftarrow \text{exchange}(GPR[rt])$

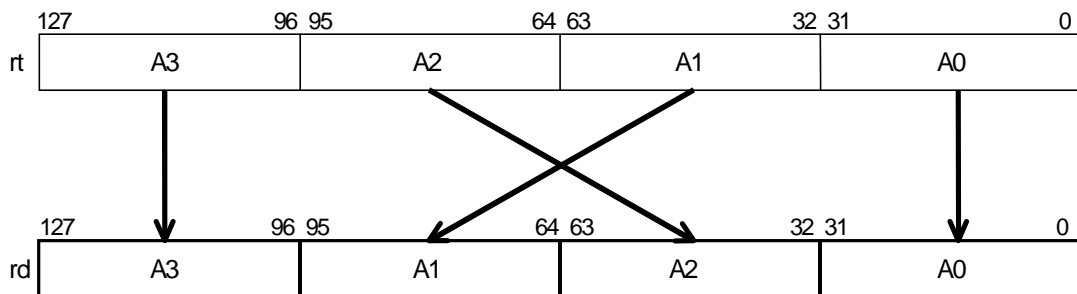
Splits a 128-bit value in GPR[rt] into four words, exchanges the two central words, and stores the result in GPR[rd]. See "Operation" about the details of the exchange.

Exceptions

None

Operation

$GPR[rd]_{31..0} \leftarrow GPR[rt]_{31..0}$
 $GPR[rd]_{63..32} \leftarrow GPR[rt]_{95..64}$
 $GPR[rd]_{95..64} \leftarrow GPR[rt]_{63..32}$
 $GPR[rd]_{127..96} \leftarrow GPR[rt]_{127..96}$



PEXEH : Parallel Exchange Even Halfword

128-bit MMI

To exchange the position of halfwords in 128-bit data.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0	
MMI 011100			0 00000		rt		rd		PEXEH 11010		MMI2 001001	
6			5		5		5		5		6	

Format

PEXEH rd, rt

Description

$GPR[rd] \leftarrow \text{exchange}(GPR[rt])$

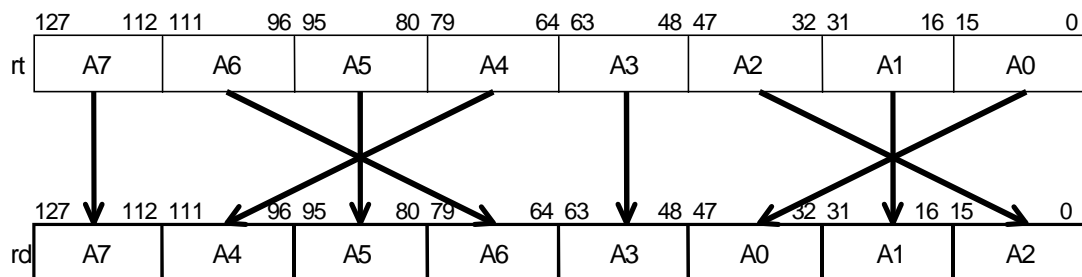
Splits a 128-bit value in GPR[rt] into eight halfwords, exchanges the sequence partially and stores the result in GPR[rd]. See "Operation" about the details of the exchange.

Exceptions

None

Operation

$GPR[rd]_{15..0} \leftarrow GPR[rt]_{47..32}$
 $GPR[rd]_{31..16} \leftarrow GPR[rt]_{31..16}$
 $GPR[rd]_{47..32} \leftarrow GPR[rt]_{15..0}$
 $GPR[rd]_{63..48} \leftarrow GPR[rt]_{63..48}$
 $GPR[rd]_{79..64} \leftarrow GPR[rt]_{111..96}$
 $GPR[rd]_{95..80} \leftarrow GPR[rt]_{95..80}$
 $GPR[rd]_{111..96} \leftarrow GPR[rt]_{79..64}$
 $GPR[rd]_{127..112} \leftarrow GPR[rt]_{127..112}$



PEXEW : Parallel Exchange Even Word

128-bit MMI

To exchange the position of words in 128-bit data.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	0	rt	rd	PEXEW	MMI2						
011100	00000			11110	001001						
6	5	5	5	5	6						

Format

PEXEW rd, rt

Description

$GPR[rd] \leftarrow \text{exchange}(GPR[rt])$

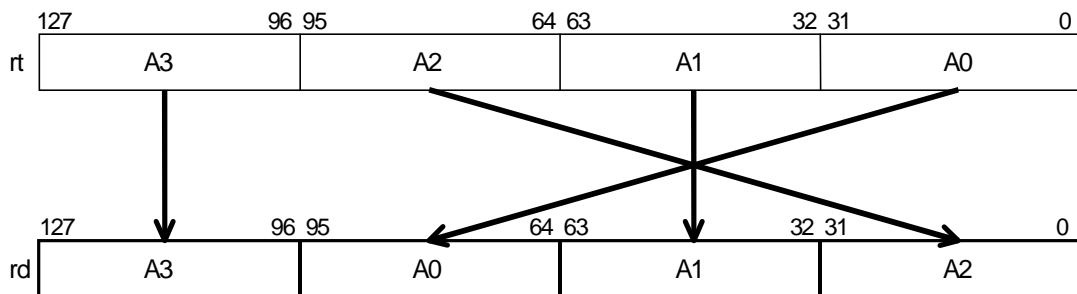
Splits a 128-bit value in GPR[rt] into four words, exchanges the two low-order words of each doubleword, and stores the result in GPR[rd]. See "Operation" about the details of the exchange.

Exceptions

None

Operation

$GPR[rd]_{31..0} \leftarrow GPR[rt]_{95..64}$
 $GPR[rd]_{63..32} \leftarrow GPR[rt]_{63..32}$
 $GPR[rd]_{95..64} \leftarrow GPR[rt]_{31..0}$
 $GPR[rd]_{127..96} \leftarrow GPR[rt]_{127..96}$



PEXT5 : Parallel Extend from 5 bits

128-bit MMI

To extend 4 bytes in the 1-5-5-5 bit format to the 8-8-8-8 bit format.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	0	rt	rd	PEXT5	MMI0						
011100	00000			11110	001000						
6	5	5	5	5	6						

Format

PEXT5 rd, rt

Description

$GPR[rd] \leftarrow \text{extend}(GPR[rt])$

Splits the 128-bit data in GPR[rt] into four words. Each of the low-order 16 bits are considered to be in the 1-5-5-5 bit data format, and they are extended to four words in the to 8-8-8-8 bit data format, as illustrated in "Operation". The resulting value is stored in GPR[rd].

Exceptions

None

Operation

$$GPR[rd]_{31..0} \leftarrow GPR[rt]_{15} \parallel 0^7 \parallel GPR[rt]_{14..10} \parallel 0^3 \parallel GPR[rt]_{9..5} \parallel 0^3$$

$$\parallel GPR[rt]_{4..0} \parallel 0^3$$

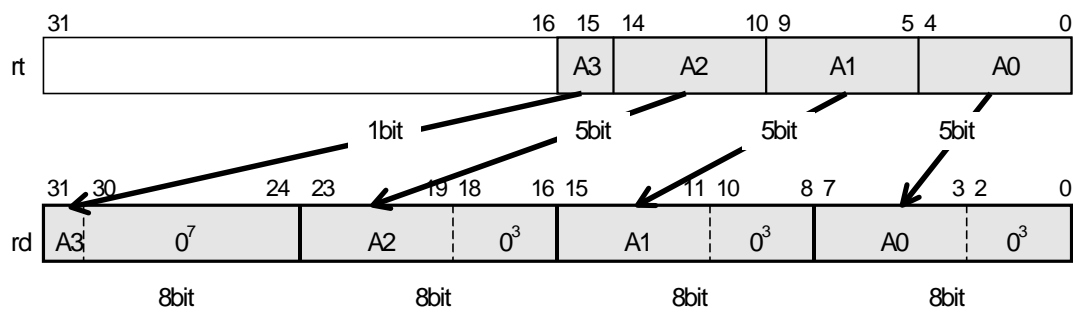
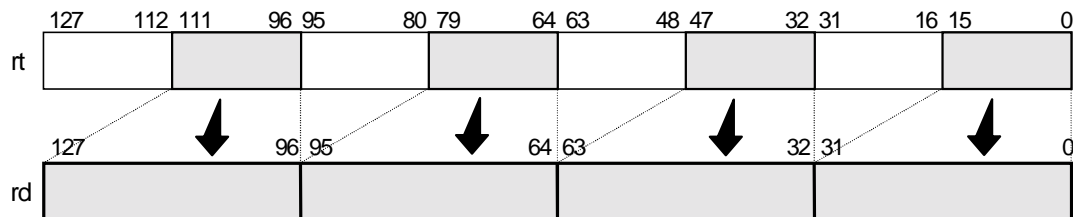
$$GPR[rd]_{63..32} \leftarrow GPR[rt]_{47} \parallel 0^7 \parallel GPR[rt]_{46..42} \parallel 0^3 \parallel GPR[rt]_{41..37} \parallel 0^3$$

$$\parallel GPR[rt]_{36..32} \parallel 0^3$$

$$GPR[rd]_{95..64} \leftarrow GPR[rt]_{79} \parallel 0^7 \parallel GPR[rt]_{78..74} \parallel 0^3 \parallel GPR[rt]_{73..69} \parallel 0^3$$

$$\parallel GPR[rt]_{68..64} \parallel 0^3$$

$$GPR[rd]_{127..96} \leftarrow GPR[rt]_{111} \parallel 0^7 \parallel GPR[rt]_{110..106} \parallel 0^3 \parallel GPR[rt]_{105..101} \parallel 0^3$$

$$\parallel GPR[rt]_{100..96} \parallel 0^3$$


PEXTLB : Parallel Extend Lower from Byte

128-bit MMI

To combine two doublewords interleaving by bytes.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI		rs		rt		rd		PEXTLB		MMI0	
011100								11010		001000	
6		5		5		5		5		6	

Format

PEXTLB rd, rs, rt

Description

$GPR[rd] \leftarrow \text{combine}(GPR[rs], GPR[rt])$

Splits the low-order 64 bits of GPR[rs] and GPR[rt] into byte data and stores them in GPR[rd] in an interleaved manner.

Exceptions

None

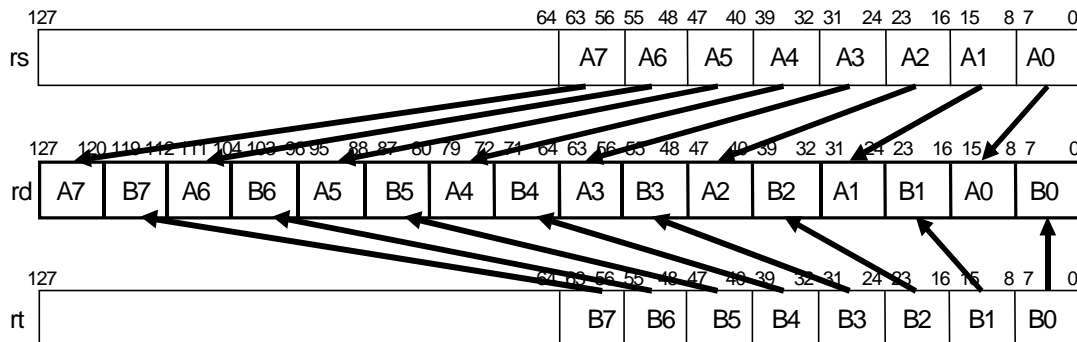
Operation

$GPR[rd]_{15..0} \leftarrow GPR[rs]_{7..0} \parallel GPR[rt]_{7..0}$

$GPR[rd]_{31..16} \leftarrow GPR[rs]_{15..8} \parallel GPR[rt]_{15..8}$

(The same operations follow every 16 bits)

$GPR[rd]_{127..112} \leftarrow GPR[rs]_{63..56} \parallel GPR[rt]_{63..56}$



PEXTLH : Parallel Extend Lower from Halfword

128-bit MMI

To combine two doublewords interleaving by halfwords.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PEXTLH	MMI0						
011100				10110	001000						
6	5	5	5	5	6						

Format

PEXTLH rd, rs, rt

Description

$GPR[rd] \leftarrow \text{combine}(GPR[rs], GPR[rt])$

Splits the low-order 64 bits of GPR[rs] and GPR[rt] into halfwords and stores them in GPR[rd] in an interleaved manner.

Exceptions

None

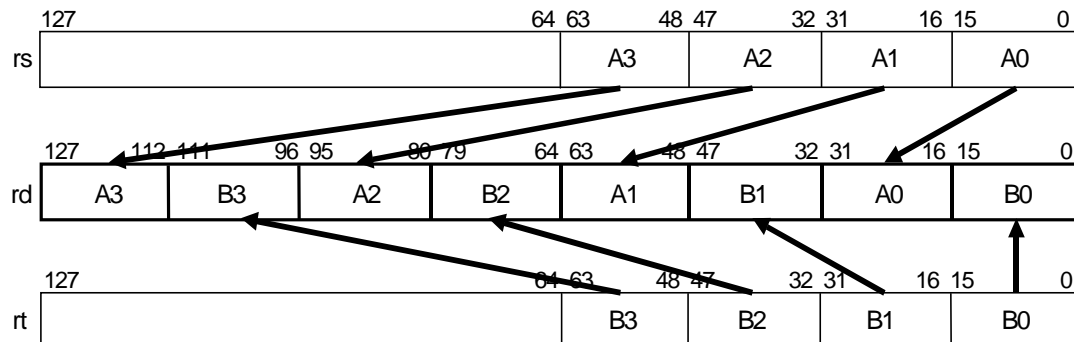
Operation

$GPR[rd]_{31..0} \leftarrow GPR[rs]_{15..0} \parallel GPR[rt]_{15..0}$

$GPR[rd]_{63..32} \leftarrow GPR[rs]_{31..16} \parallel GPR[rt]_{31..16}$

(The same operations follow every 32 bits)

$GPR[rd]_{127..112} \leftarrow GPR[rs]_{63..48} \parallel GPR[rt]_{63..48}$



PEXTLW : Parallel Extend Lower from Word

128-bit MMI

To combine two doublewords interleaving by words.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI		rs		rt		rd		PEXTLW		MMI0	
011100								10010		001000	
6		5		5		5		5		6	

Format

PEXTLW rd, rs, rt

Description

$GPR[rd] \leftarrow \text{combine}(GPR[rs], GPR[rt])$

Splits the low-order 64 bits of GPR[rs] and GPR[rt] into words and stores them in GPR[rd] in an interleaved manner.

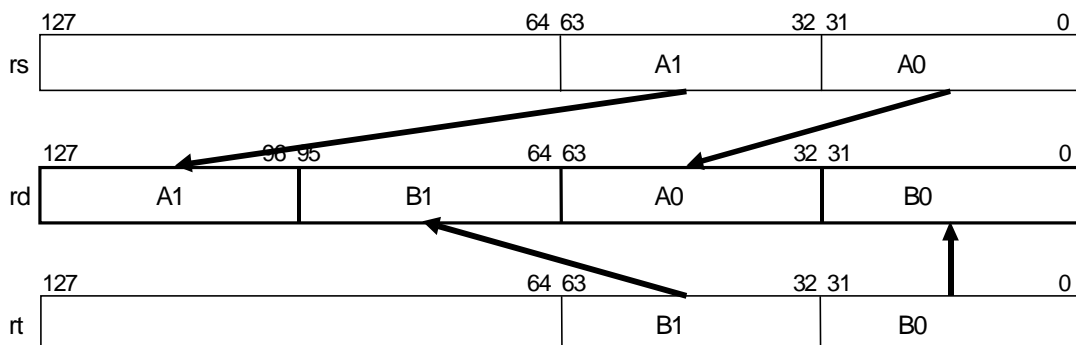
Exceptions

None

Operation

$GPR[rd]_{63..0} \leftarrow GPR[rs]_{31..0} \parallel GPR[rt]_{31..0}$

$GPR[rd]_{127..64} \leftarrow GPR[rs]_{63..32} \parallel GPR[rt]_{63..32}$



PEXTUB : Parallel Extend Upper from Byte

128-bit MMI

To combine two doublewords interleaving by bytes.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PEXTUB	MMI1						
011100				11010	101000						
6	5	5	5	5	6						

Format

PEXTUB rd, rs, rt

Description

$GPR[rd] \leftarrow \text{combine}(GPR[rs], GPR[rt])$

Splits the high-order 64 bits of GPR[rs] and GPR[rt] into bytes and stores them in GPR[rd] in an interleaved manner.

Exceptions

None

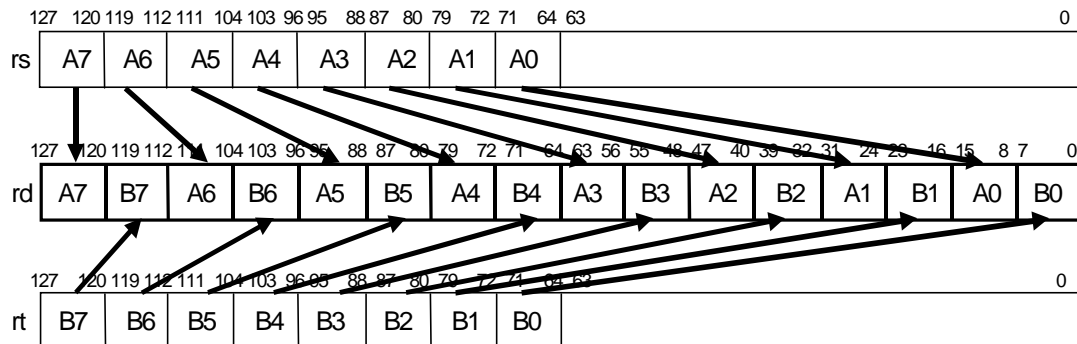
Operation

$GPR[rd]_{15..0} \leftarrow GPR[rs]_{71..64} \parallel GPR[rt]_{71..64}$

$GPR[rd]_{31..16} \leftarrow GPR[rs]_{79..72} \parallel GPR[rt]_{79..72}$

(The same operations follow every 16 bits)

$GPR[rd]_{127..112} \leftarrow GPR[rs]_{127..120} \parallel GPR[rt]_{127..120}$



PEXTUH : Parallel Extend Upper from Halfword

128-bit MMI

To combine two doublewords interleaving by halfwords.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PEXTUH	MMI1						
011100				10110	101000						
6	5	5	5	5	6						

Format

PEXTUH rd, rs, rt

Description

$GPR[rd] \leftarrow \text{combine}(GPR[rs], GPR[rt])$

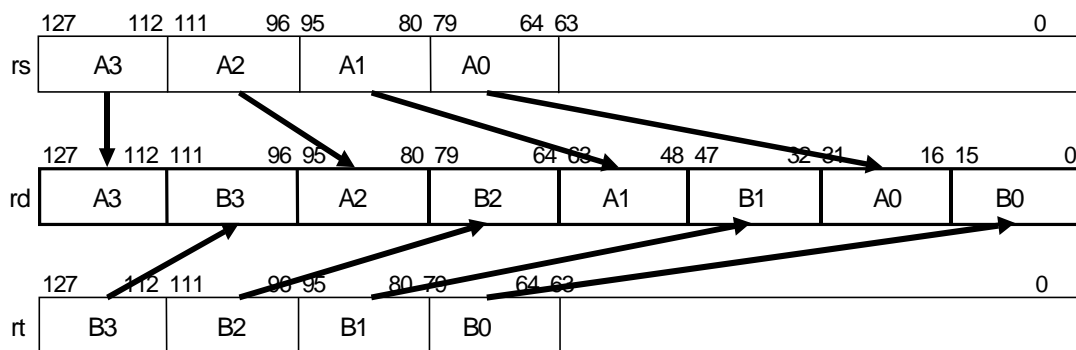
Splits the high-order 64 bits of GPR[rs] and GPR[rt] into halfwords and stores them in GPR[rd] in an interleaved manner.

Exceptions

None

Operation

$GPR[rd]_{31..0} \leftarrow GPR[rs]_{79..64} \parallel GPR[rt]_{79..64}$
 $GPR[rd]_{63..32} \leftarrow GPR[rs]_{95..80} \parallel GPR[rt]_{95..80}$
 $GPR[rd]_{95..64} \leftarrow GPR[rs]_{111..96} \parallel GPR[rt]_{111..96}$
 $GPR[rd]_{127..96} \leftarrow GPR[rs]_{127..112} \parallel GPR[rt]_{127..112}$



PEXTUW : Parallel Extend Upper from Word

128-bit MMI

To combine two doublewords interleaving by words.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PEXTUW	MMI1						
011100				10010	101000						
6	5	5	5	5	6						

Format

PEXTUW rd, rs, rt

Description

$GPR[rd] \leftarrow \text{combine}(GPR[rs], GPR[rt])$

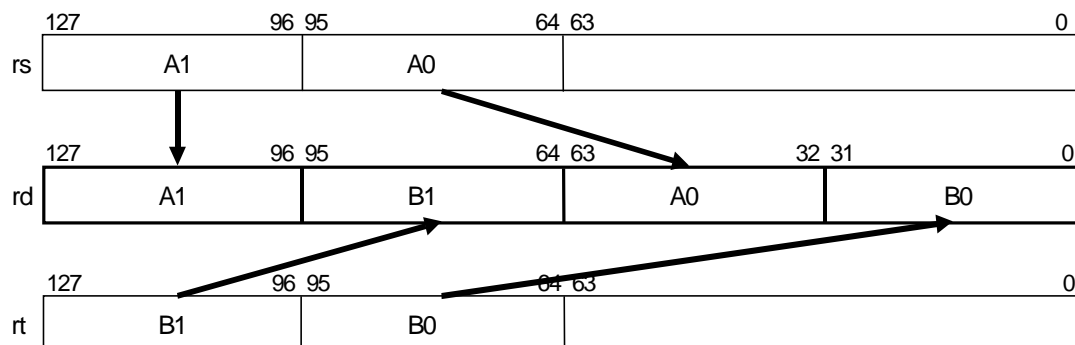
Splits the high-order 64 bits of GPR[rs] and GPR[rt] into words and stores them in GPR[rd] in an interleaved manner.

Exceptions

None

Operation

$GPR[rd]_{63..0} \leftarrow GPR[rs]_{95..64} \parallel GPR[rt]_{95..64}$
 $GPR[rd]_{127..64} \leftarrow GPR[rs]_{127..96} \parallel GPR[rt]_{127..96}$



PHMADH : Parallel Horizontal Multiply-Add Halfword

128-bit MMI

To multiply 8 pairs of 16-bit signed integers and horizontally add.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PHMADH	MMI2						
011100				10001	001001						
6	5	5	5	5	6						

Format

PHMADH rd, rs, rt

Description

$$(GPR[rd], HI, LO) \leftarrow GPR[rs] \times GPR[rt] + GPR[rs] \times GPR[rt]$$

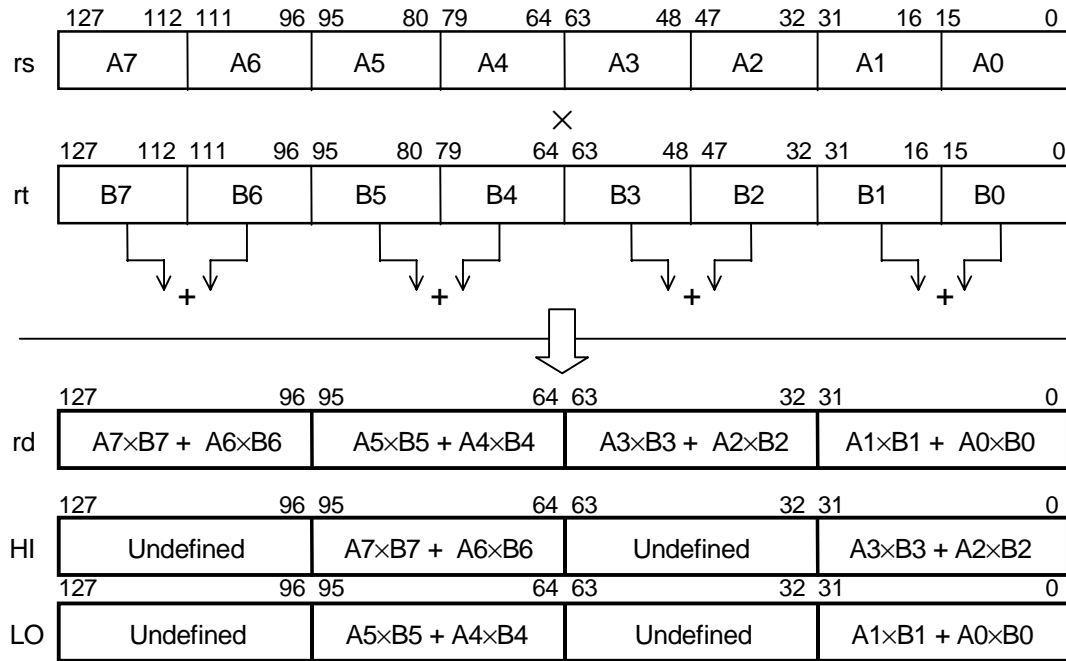
Splits the 128-bit data in GPR[rs] and GPR[rt] into eight 16-bit signed integers. Multiplies the halfwords in GPR[rs] by the corresponding halfwords in GPR[rt], then adds the result of adjacent multiplications and stores them in the GPR[rd], HI and LO registers, as described in "Operation".

Exceptions

None. Even when the result of the arithmetic operation overflows, an overflow exception does not occur.

Operation

prod0	$\leftarrow GPR[rs]_{31..16} \times GPR[rt]_{31..16} + GPR[rs]_{15..0} \times GPR[rt]_{15..0}$
LO _{31..0}	$\leftarrow prod0_{31..0}$
GPR[rd] _{31..0}	$\leftarrow prod0_{31..0}$
prod1	$\leftarrow GPR[rs]_{63..48} \times GPR[rt]_{63..48} + GPR[rs]_{47..32} \times GPR[rt]_{47..32}$
HI _{31..0}	$\leftarrow prod1_{31..0}$
GPR[rd] _{63..32}	$\leftarrow prod1_{31..0}$
prod2	$\leftarrow GPR[rs]_{95..80} \times GPR[rt]_{95..80} + GPR[rs]_{79..64} \times GPR[rt]_{79..64}$
LO _{95..64}	$\leftarrow prod2_{31..0}$
GPR[rd] _{95..64}	$\leftarrow prod2_{31..0}$
prod3	$\leftarrow GPR[rs]_{127..112} \times GPR[rt]_{127..112} + GPR[rs]_{111..96} \times GPR[rt]_{111..96}$
HI _{95..64}	$\leftarrow prod3_{31..0}$
GPR[rd] _{127..96}	$\leftarrow prod3_{31..0}$



Programming Notes

In the EE Core, the integer multiply operation proceeds asynchronously. An attempt to read the contents of the LO/HI/GPR[rd] registers before the multiply operation finishes will result in interlock. Other CPU instructions can execute without delay. Therefore, scheduling the multiply operation appropriately can improve performance.

Even when the result of a multiply operation overflows, an overflow exception does not occur. If an overflow is required to be detected, an explicit check is necessary.

PHMSBH : Parallel Horizontal Multiply-Subtract Halfword

128-bit MMI

To multiply 8 pairs of 16-bit signed integers and horizontally subtract.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PHMSBH	MMI2						
011100				10101	001001						
6	5	5	5	5	6						

Format

PHMSBH rd, rs, rt

Description

$$(GPR[rd], HI, LO) \leftarrow GPR[rs] \times GPR[rt] - GPR[rs] \times GPR[rt]$$

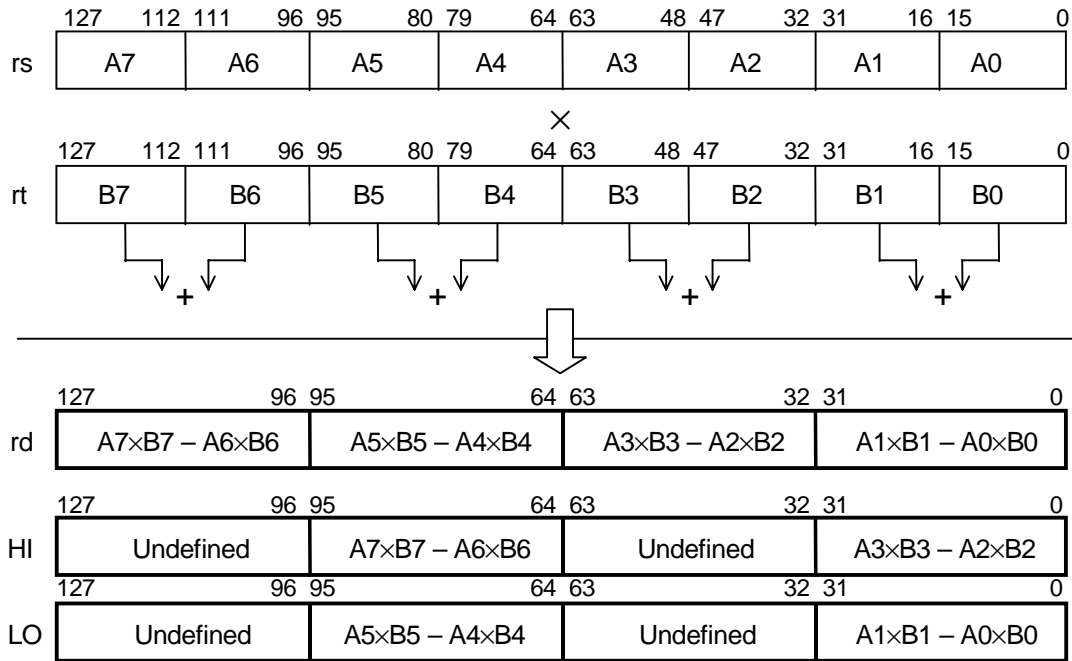
Splits the 128-bit data in GPR[rs] and GPR[rt] into eight 16-bit signed integers. Multiplies the halfwords in GPR[rs] by the corresponding halfwords in GPR[rt], then subtracts the results of adjacent multiplications and stores them in the GPR[rd], HI and LO registers, as described in "Operation".

Exceptions

None

Operation

prod0	$\leftarrow GPR[rs]_{31..16} \times GPR[rt]_{31..16} - GPR[rs]_{15..0} \times GPR[rt]_{15..0}$
LO _{31..0}	$\leftarrow prod0_{31..0}$
GPR[rd] _{31..0}	$\leftarrow prod0_{31..0}$
prod1	$\leftarrow GPR[rs]_{63..48} \times GPR[rt]_{63..48} - GPR[rs]_{47..32} \times GPR[rt]_{47..32}$
HI _{31..0}	$\leftarrow prod1_{31..0}$
GPR[rd] _{63..32}	$\leftarrow prod1_{31..0}$
prod2	$\leftarrow GPR[rs]_{95..80} \times GPR[rt]_{95..80} - GPR[rs]_{79..64} \times GPR[rt]_{79..64}$
LO _{95..64}	$\leftarrow prod2_{31..0}$
GPR[rd] _{95..64}	$\leftarrow prod2_{31..0}$
prod3	$\leftarrow GPR[rs]_{127..112} \times GPR[rt]_{127..112} - GPR[rs]_{111..96} \times GPR[rt]_{111..96}$
HI _{95..64}	$\leftarrow prod3_{31..0}$
GPR[rd] _{127..96}	$\leftarrow prod3_{31..0}$



Programming Notes

In the EE Core, the integer multiply operation proceeds asynchronously. An attempt to read the contents of the LO/HI/GPR[rd] registers before the multiply operation finishes will result in interlock. Other CPU instructions can execute without delay. Therefore, scheduling the multiply operation appropriately can improve performance.

Even when the result of the multiply operation overflows, an overflow exception does not occur. If an overflow is required to be detected, an explicit check is necessary.

PINTEH : Parallel Interleave Even Halfword

128-bit MMI

To combine halfwords in a halfword wide interleaved operation.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PINTEH	MMI3						
011100				01010	101001						
6	5	5	5	5	6						

Format

PINTEH rd, rs, rt

Description

$GPR[rd] \leftarrow \text{interleave}(GPR[rs], GPR[rt])$

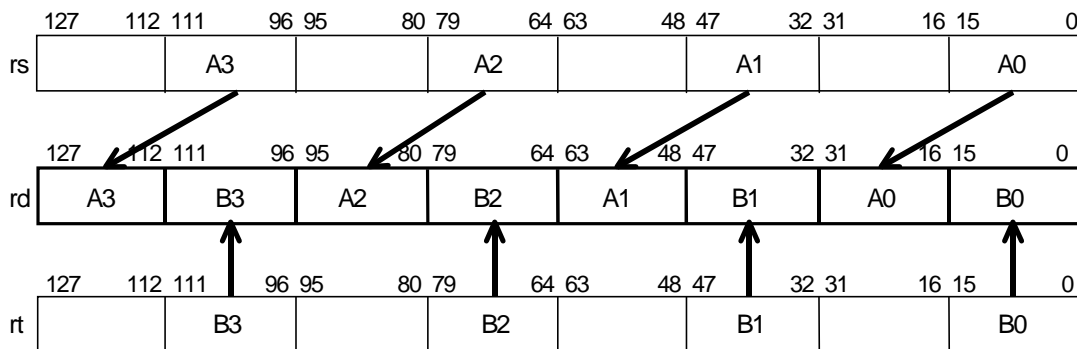
Splits the GPR[rs] and GPR[rt] into words, and stores each of the low-order halfwords in GPR[rd] in an interleaved manner.

Exceptions

None

Operation

$GPR[rd]_{31..0} \leftarrow GPR[rs]_{15..0} \parallel GPR[rt]_{15..0}$
 $GPR[rd]_{63..32} \leftarrow GPR[rs]_{47..32} \parallel GPR[rt]_{47..32}$
 $GPR[rd]_{95..64} \leftarrow GPR[rs]_{79..64} \parallel GPR[rt]_{79..64}$
 $GPR[rd]_{127..96} \leftarrow GPR[rs]_{111..96} \parallel GPR[rt]_{111..96}$



PINTH : Parallel Interleave Halfword

128-bit MMI

To combine 2 doublewords in a halfword wide interleaved operation.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI		rs		rt		rd		PINTH		MMI2	
011100								01010		001001	
6		5		5		5		5		6	

Format

PINTH rd, rs, rt

Description

$GPR[rd] \leftarrow \text{interleave}(GPR[rs], GPR[rt])$

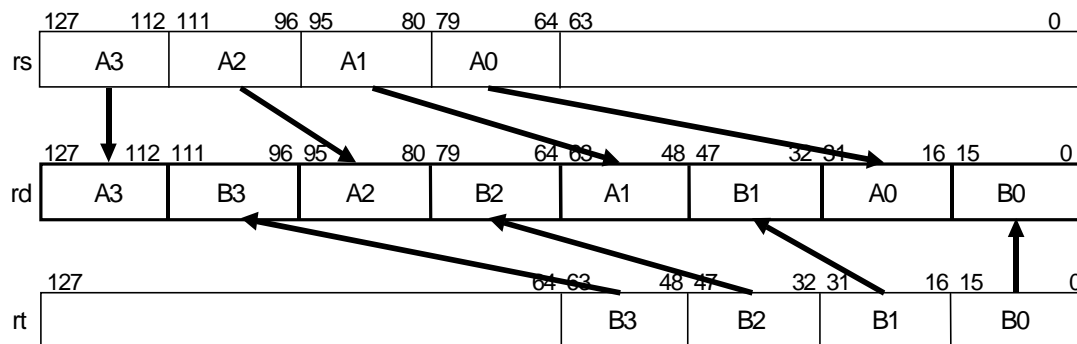
Splits the high-order 64 bits of GPR[rs] and the low-order 64 bits of GPR[rt] into halfwords and stores them in GPR[rd] in an interleaved manner.

Exceptions

None

Operation

$GPR[rd]_{31..0} \leftarrow GPR[rs]_{79..64} \parallel GPR[rt]_{15..0}$
 $GPR[rd]_{63..32} \leftarrow GPR[rs]_{95..80} \parallel GPR[rt]_{31..16}$
 $GPR[rd]_{95..64} \leftarrow GPR[rs]_{111..96} \parallel GPR[rt]_{47..32}$
 $GPR[rd]_{127..96} \leftarrow GPR[rs]_{127..112} \parallel GPR[rt]_{63..48}$



PLZCW : Parallel Leading Zero or one Count Word

EE Core

To count leading zeros or ones.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	0	rd	0	PLZCW						
011100		00000			00000				000100		
6	5	5	5		5				6		

Format

PLZCW rd, rs

Description

$GPR[rd] \leftarrow LZC(GPR[rs]) - 1$

Splits the 64-bit value in GPR[rs] into two words and counts the number of leading bits that have the same value as the highest-order bit (either zero or one) in both words. The result minus 1 is stored in the corresponding words in GPR[rd].

Exceptions

None

Operation

$GPR[rd]_{31..0} \leftarrow LZC(GPR[rs]_{31..0}) - 1$

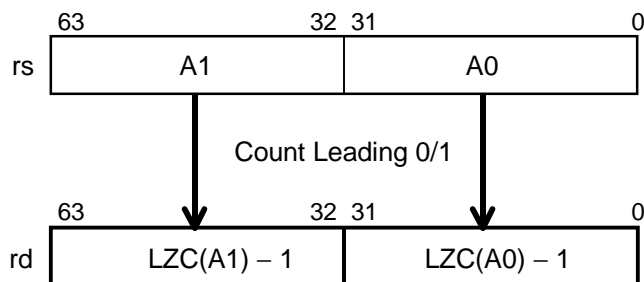
$GPR[rd]_{63..32} \leftarrow LZC(GPR[rs]_{63..32}) - 1$

If $GPR[1] == 0x000F:FF0F:FF0F:F00F$

Then

PLZCW 1, 1

$GPR[1] == 0x0000:000B:0000:0007$



PMADDH : Parallel Multiply-Add Halfword

128-bit MMI

To multiply 8 pairs of 16-bit signed integers and accumulate in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI		rs		rt		rd		PMADDH		MMI2	
011100								10000		001001	
6		5		5		5		5		6	

Format

PMADDH rd, rs, rt

Description

$(\text{GPR}[\text{rd}], \text{HI}, \text{LO}) \leftarrow (\text{HI}, \text{LO}) + \text{GPR}[\text{rs}] \times \text{GPR}[\text{rt}]$

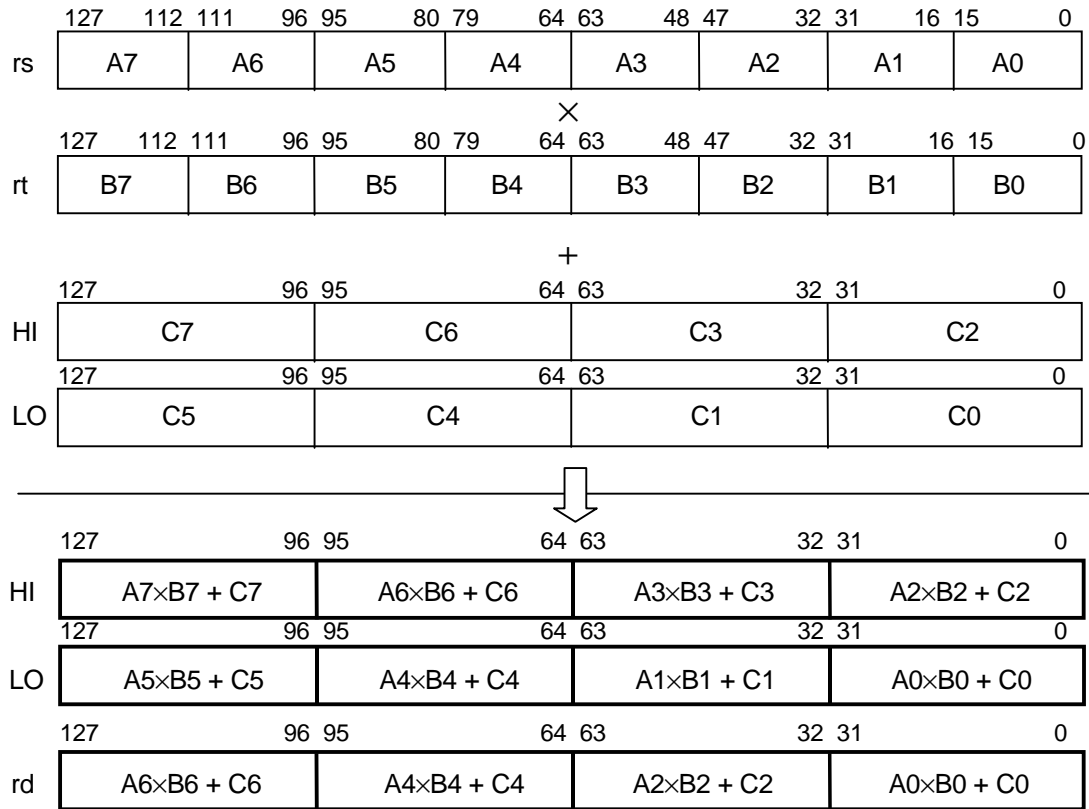
Splits GPR[rs] and GPR[rt] into eight 16-bit signed integers, multiplies each halfword in GPR[rs] by the corresponding halfword in GPR[rt], and adds the results to the corresponding words in the HI and LO registers. The result are stored in HI, LO, and GPR[rd].

Exceptions

None. Even when the result of the arithmetic operation overflows, an exception does not occur.

Operation

prod0	$\leftarrow \text{LO}_{31..0} + \text{GPR}[\text{rs}]_{15..0} \times \text{GPR}[\text{rt}]_{15..0}$
LO _{31..0}	$\leftarrow \text{prod0}_{31..0}$
GPR[rd] _{31..0}	$\leftarrow \text{prod0}_{31..0}$
prod1	$\leftarrow \text{LO}_{63..32} + \text{GPR}[\text{rs}]_{31..16} \times \text{GPR}[\text{rt}]_{31..16}$
LO _{63..32}	$\leftarrow \text{prod1}_{31..0}$
prod2	$\leftarrow \text{HI}_{31..0} + \text{GPR}[\text{rs}]_{47..32} \times \text{GPR}[\text{rt}]_{47..32}$
HI _{31..0}	$\leftarrow \text{prod2}_{31..0}$
GPR[rd] _{63..32}	$\leftarrow \text{prod2}_{31..0}$
prod3	$\leftarrow \text{HI}_{63..32} + \text{GPR}[\text{rs}]_{63..48} \times \text{GPR}[\text{rt}]_{63..48}$
HI _{63..32}	$\leftarrow \text{prod3}_{31..0}$
prod4	$\leftarrow \text{LO}_{95..64} + \text{GPR}[\text{rs}]_{79..64} \times \text{GPR}[\text{rt}]_{79..64}$
LO _{95..64}	$\leftarrow \text{prod4}_{31..0}$
GPR[rd] _{95..64}	$\leftarrow \text{prod4}_{31..0}$
prod5	$\leftarrow \text{LO}_{127..96} + \text{GPR}[\text{rs}]_{95..80} \times \text{GPR}[\text{rt}]_{95..80}$
LO _{127..96}	$\leftarrow \text{prod5}_{31..0}$
prod6	$\leftarrow \text{HI}_{95..64} + \text{GPR}[\text{rs}]_{111..96} \times \text{GPR}[\text{rt}]_{111..96}$
HI _{95..64}	$\leftarrow \text{prod6}_{31..0}$
GPR[rd] _{127..96}	$\leftarrow \text{prod6}_{31..0}$
prod7	$\leftarrow \text{HI}_{127..96} + \text{GPR}[\text{rs}]_{127..112} \times \text{GPR}[\text{rt}]_{127..112}$
HI _{127..96}	$\leftarrow \text{prod7}_{31..0}$



Programming Notes

In the EE Core, the integer multiply operation proceeds asynchronously. An attempt to read the contents of the LO/HI/GPR[rd] registers before the multiply operation finishes will result in interlock. Other CPU instructions can execute without delay. Therefore, scheduling the multiply operation appropriately can improve performance.

Even when the result of the multiply operation overflows, an overflow exception does not occur. If an overflow is required to be detected, an explicit check is necessary.

PMADDUW : Parallel Multiply-Add Unsigned Word

128-bit MMI

To multiply 2 pairs of 32-bit unsigned integers and accumulate in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI		rs		rt		rd		PMADDUW		MMI3	
011100								00000		101001	
6		5		5		5		5		6	

Format

PMADDUW rd, rs, rt

Description

$(\text{GPR}[\text{rd}], \text{HI}, \text{LO}) \leftarrow (\text{HI}, \text{LO}) + \text{GPR}[\text{rs}] \times \text{GPR}[\text{rt}]$

Multiplies bits 95..64 of GPR[rs] by bits 95..64 of GPR[rt] and bits 31..0 of GPR[rs] by bits 31..0 of GPR[rt] as unsigned 32-bit integers and adds the resulting 64-bit values to the corresponding words in the HI and LO registers. A part of the result is stored in GPR[rd]. See "Operation" for details.

Restrictions

If the contents of GPR[rt] and GPR[rs] are not sign-extended 32-bit values (bits 127..95 equal and bits 63..31 equal), then the result is undefined.

Exceptions

None. Even when the result of the arithmetic operation overflows, an exception does not occur.

Operation

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif

prod0 $\leftarrow (\text{HI}_{31..0} \parallel \text{LO}_{31..0}) + (0 \parallel \text{GPR}[\text{rs}]_{31..0}) \times (0 \parallel \text{GPR}[\text{rt}]_{31..0})$

LO_{63..0} $\leftarrow (\text{prod0}_{31})^{32} \parallel \text{prod0}_{31..0}$

HI_{63..0} $\leftarrow (\text{prod0}_{63})^{32} \parallel \text{prod0}_{63..32}$

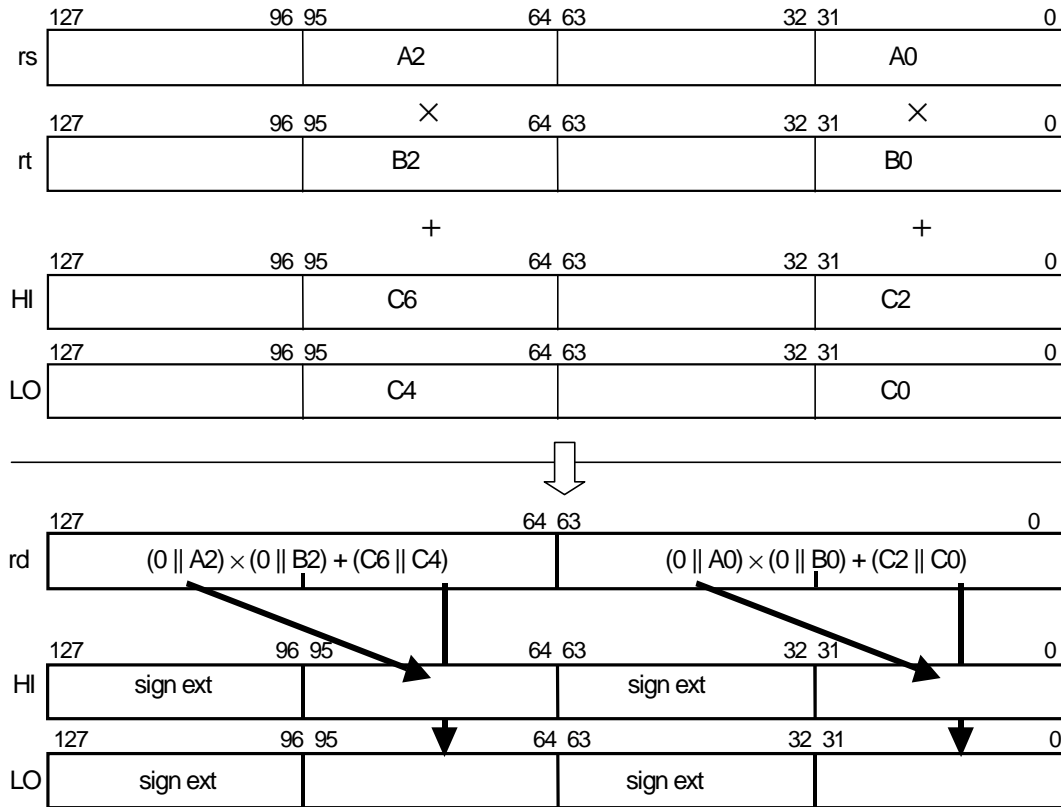
GPR[rd]_{63..0} $\leftarrow \text{prod0}_{63..0}$

prod1 $\leftarrow (\text{HI}_{95..64} \parallel \text{LO}_{95..64}) + (0 \parallel \text{GPR}[\text{rs}]_{95..64}) \times (0 \parallel \text{GPR}[\text{rt}]_{95..64})$

LO_{127..64} $\leftarrow (\text{prod1}_{31})^{32} \parallel \text{prod1}_{31..0}$

HI_{127..64} $\leftarrow (\text{prod1}_{63})^{32} \parallel \text{prod1}_{63..32}$

GPR[rd]_{127..64} $\leftarrow \text{prod1}_{63..0}$



Programming Notes

In the EE Core, the integer multiply operation proceeds asynchronously. An attempt to read the contents of the LO/HI/GPR[rd] registers before the multiply operation finishes will result in interlock. Other CPU instructions can execute without delay. Therefore, scheduling the multiply operation appropriately can improve performance.

Even when the result of the multiply operation overflows, an overflow exception does not occur. If an overflow is required to be detected, an explicit check is necessary.

PMADDW : Parallel Multiply-Add Word

128-bit MMI

To multiply 2 pairs of 32-bit signed integers and accumulate in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PMADDW	MMI2						
011100				00000	001001						
6	5	5	5	5	6						

Format

PMADDW rd, rs, rt

Description

$(\text{GPR}[\text{rd}], \text{HI}, \text{LO}) \leftarrow (\text{HI}, \text{LO}) + \text{GPR}[\text{rs}] \times \text{GPR}[\text{rt}]$

Multiplies bits 95..64 of GPR[rs] by bits 95..64 of GPR[rt] and bits 31..0 of GPR[rs] by bits 31..0 of GPR[rt] as signed 32-bit integers and adds the resulting 64-bit value to the corresponding word positions in the HI and LO registers. A part of the result is stored in GPR[rd]. See "Operation" for details.

Restrictions

If the contents of GPR[rt] and GPR[rs] are not sign-extended 32-bit values (bits 127..95 equal and bits 63..31 equal), then the result is undefined.

Exceptions

None

Operation

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif

prod0 $\leftarrow (\text{HI}_{31..0} \parallel \text{LO}_{31..0}) + \text{GPR}[\text{rs}]_{31..0} \times \text{GPR}[\text{rt}]_{31..0}$

LO_{63..0} $\leftarrow (\text{prod0}_{31})^{32} \parallel \text{prod0}_{31..0}$

HI_{63..0} $\leftarrow (\text{prod0}_{63})^{32} \parallel \text{prod0}_{63..32}$

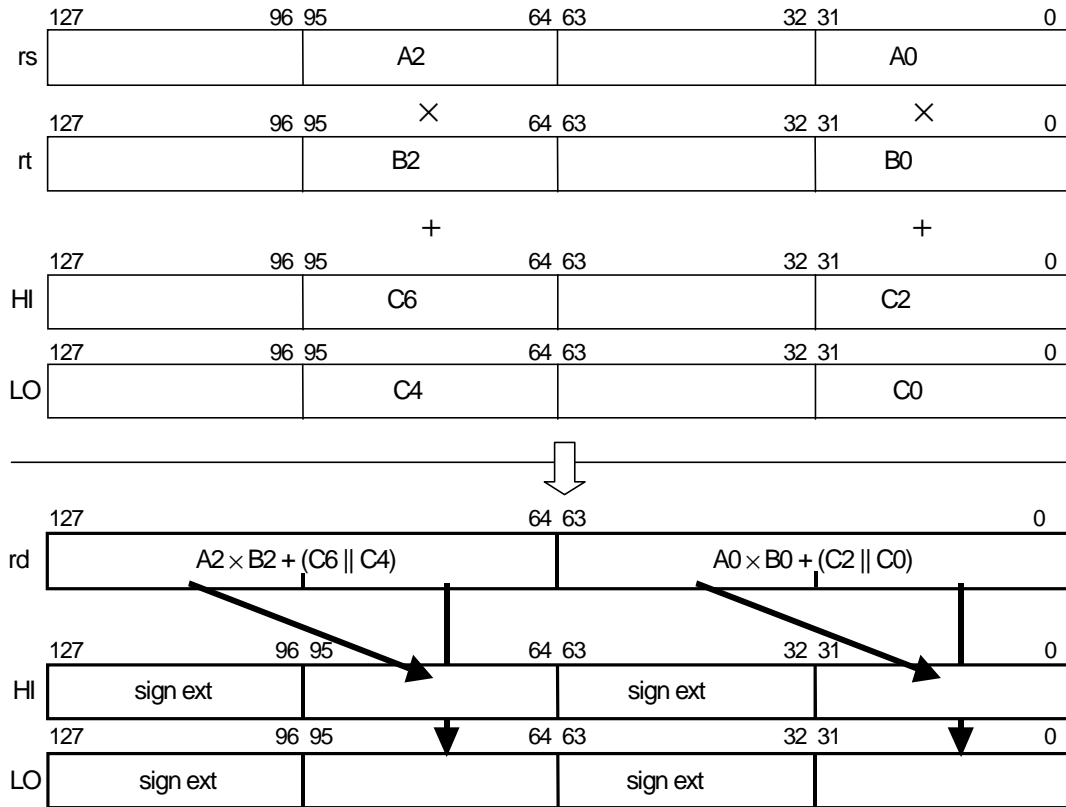
GPR[rd]_{63..0} $\leftarrow \text{prod0}_{63..0}$

prod1 $\leftarrow (\text{HI}_{95..64} \parallel \text{LO}_{95..64}) + \text{GPR}[\text{rs}]_{95..64} \times \text{GPR}[\text{rt}]_{95..64}$

LO_{127..64} $\leftarrow (\text{prod1}_{31})^{32} \parallel \text{prod1}_{31..0}$

HI_{127..64} $\leftarrow (\text{prod1}_{63})^{32} \parallel \text{prod1}_{63..32}$

GPR[rd]_{127..64} $\leftarrow \text{prod1}_{63..0}$



Programming Notes

In the EE Core, the integer multiply operation proceeds asynchronously. An attempt to read the contents of the LO/HI/GPR[rd] registers before the multiply operation finishes will result in interlock. Other CPU instructions can execute without delay. Therefore, scheduling the multiply operation appropriately can improve performance.

Even when the result of the multiply operation overflows, an overflow exception does not occur. If an overflow is required to be detected, an explicit check is necessary.

PMAXH : Parallel Maximize Halfword

128-bit MMI

To compare 16-bit signed integers and calculate the maximum value (8 parallel operations).

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PMAXH	MMI0						
011100				00111	001000						
6	5	5	5	5	6						

Format

PMAXH rd, rs, rt

Description

$GPR[rd] \leftarrow \max(GPR[rs], GPR[rt])$

Splits $GPR[rs]$ and $GPR[rt]$ into eight 16-bit signed integers and compares the data in $GPR[rs]$ with the corresponding data in $GPR[rt]$ and stores the maximum value in the corresponding halfwords in $GPR[rd]$.

Exceptions

None

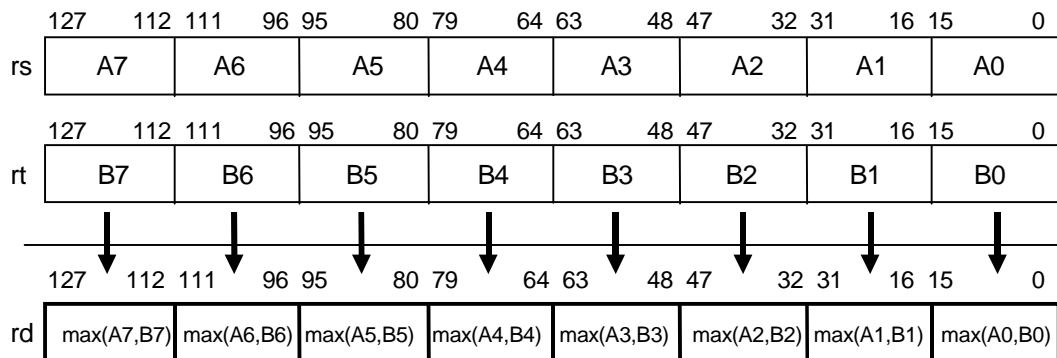
Operation

```
if ((GPR[rs]15..0 > GPR[rt]15..0) then
    GPR[rd]15..0 ← GPR[rs]15..0
else
    GPR[rd]15..0 ← GPR[rt]15..0
endif
```

```
if ((GPR[rs]31..16 > GPR[rt]31..16) then
    GPR[rd]31..16 ← GPR[rs]31..16
else
    GPR[rd]31..16 ← GPR[rt]31..16
endif
```

(The same operations follow every 16 bits)

```
if ((GPR[rs]127..112 > GPR[rt]127..112) then
    GPR[rd]127..112 ← GPR[rs]127..112
else
    GPR[rd]127..112 ← GPR[rt]127..112
endif
```



PMAXW : Parallel Maximize Word

128-bit MMI

To compare 32-bit signed integers and calculate the maximum value (4 parallel operations).

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PMAXW	MMI0						
011100				00011	001000						
6	5	5	5	5	6						

Format

PMAXW rd, rs, rt

Description

$GPR[rd] \leftarrow \max(GPR[rs], GPR[rt])$

Splits GPR[rs] and GPR[rt] into four 32-bit signed integers and compares the data in GPR[rs] with the corresponding data in GPR[rt] and stores the maximum value in the corresponding words in GPR[rd].

Exceptions

None

Operation

```

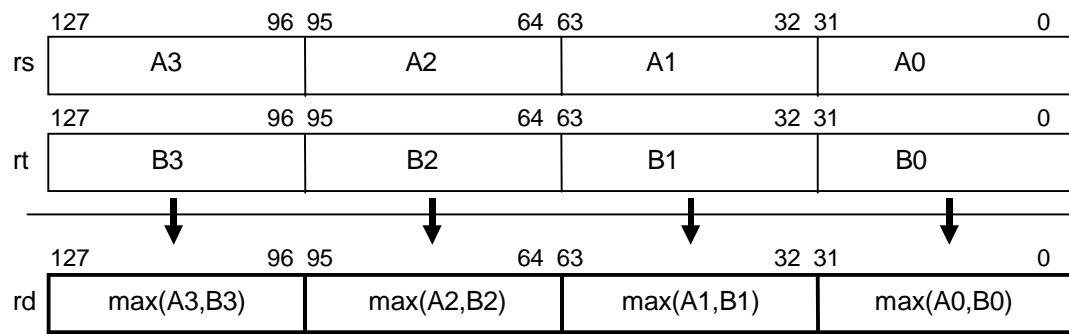
if ((GPR[rs]31..0 > GPR[rt]31..0) then
    GPR[rd]31..0 ← GPR[rs]31..0
else
    GPR[rd]31..0 ← GPR[rt]31..0
endif

if ((GPR[rs]63..32 > GPR[rt]63..32) then
    GPR[rd]63..32 ← GPR[rs]63..32
else
    GPR[rd]63..32 ← GPR[rt]63..32
endif

if ((GPR[rs]95..64 > GPR[rt]95..64) then
    GPR[rd]95..64 ← GPR[rs]95..64
else
    GPR[rd]95..64 ← GPR[rt]95..64
endif

if ((GPR[rs]127..96 > GPR[rt]127..96) then
    GPR[rd]127..96 ← GPR[rs]127..96
else
    GPR[rd]127..96 ← GPR[rt]127..96
endif

```



PMFHI : Parallel Move From HI Register

128-bit MMI

To copy the 128-bit value in the HI register to a GPR.

Operation Code

31	26	25	16	15	11	10	6	5	0
MMI	0				rd	PMFHI	MMI2		
011100	00 0000 0000					01000	001001		
6	10				5	5	6		

Format

PMFHI rd

Description

$GPR[rd] \leftarrow HI$

Stores the 128-bit value in the HI register into GPR[rd].

Exceptions

None

Operation

$GPR[rd]_{127..0} \leftarrow HI_{127..0}$

PMFHL.LH : Parallel Move From HI/LO Register

128-bit MMI

To copy the contents of the HI and LO registers to a GPR in halfword units.

Operation Code

31	26	25	16	15	11	10	6	5	0
MMI	0	rd	fmt	PMFHL					
011100	00 0000 0000		00011	110000					
6	10	5	5	6					

Format

PMFHL.LH rd

Description

$GPR[rd] \leftarrow HI / LO$

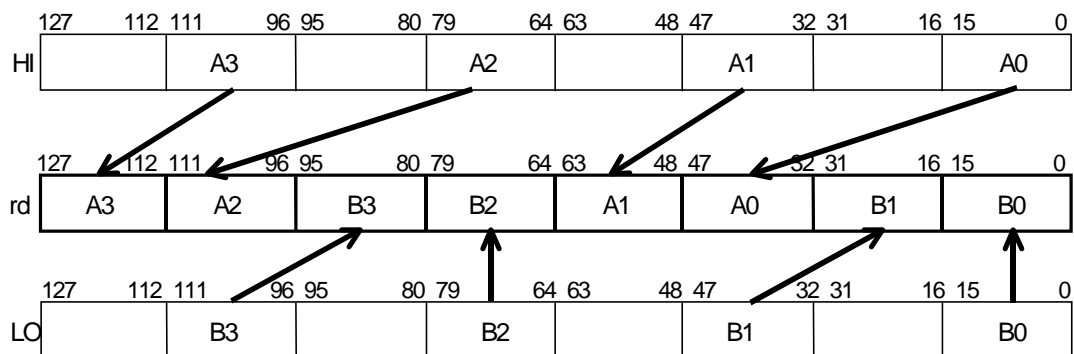
Splits the contents of the HI and LO registers into halfwords, rearranges them as shown in "Operation", and stores them in GPR[rd].

Exceptions

None

Operation

$GPR[rd]_{15..0} \leftarrow LO_{15..0}$
 $GPR[rd]_{31..16} \leftarrow LO_{47..32}$
 $GPR[rd]_{47..32} \leftarrow HI_{15..0}$
 $GPR[rd]_{63..48} \leftarrow HI_{47..32}$
 $GPR[rd]_{79..64} \leftarrow LO_{79..64}$
 $GPR[rd]_{95..80} \leftarrow LO_{111..96}$
 $GPR[rd]_{111..96} \leftarrow HI_{79..64}$
 $GPR[rd]_{127..112} \leftarrow HI_{111..96}$



PMFHL.LW : Parallel Move From HI/LO Register

128-bit MMI

To copy the contents of the HI and LO registers to a GPR in word units.

Operation Code

31	26	25	16	15	11	10	6	5	0
MMI	0				rd	fmt	PMFHL		
011100	00 0000 0000					00000	110000		
6	10				5	5	6		

Format

PMFHL.LW rd

Description

$GPR[rd] \leftarrow HI / LO$

Splits the contents of the HI and LO registers into words, rearranges them as shown in "Operation", and stores them in GPR[rd].

Exceptions

None

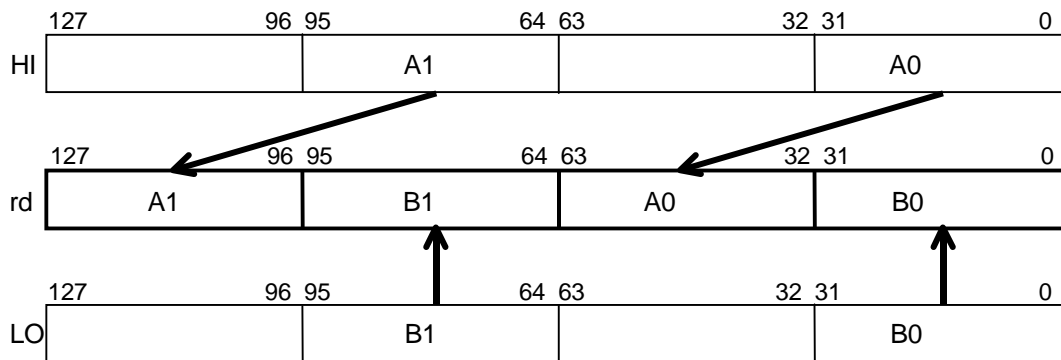
Operation

$GPR[rd]_{31..0} \leftarrow LO_{31..0}$

$GPR[rd]_{63..32} \leftarrow HI_{31..0}$

$GPR[rd]_{95..64} \leftarrow LO_{95..64}$

$GPR[rd]_{127..96} \leftarrow HI_{95..64}$



PMFHL.SH : Parallel Move From HI/LO Register

128-bit MMI

To saturate the contents of HI/LO register to signed halfwords and copy to a GPR.

Operation Code

31	26	25	16	15	11	10	6	5	0
MMI	0				rd	fmt	PMFHL		
011100	00 0000 0000					00100	110000		
6	10				5	5	6		

Format

PMFHL.SH rd

Description

$GPR[rd] \leftarrow HI / LO$

Splits the contents of HI/LO registers into words, saturates them to 16-bit signed integers, rearranges them as shown in "Operation", and stores them in GPR[rd].

Exceptions

None

Operation

```

GPR[rd]15..0      ← clamp(LO31..0)
GPR[rd]31..16     ← clamp(LO63..32)
GPR[rd]47..32     ← clamp(HI31..0)
GPR[rd]63..48     ← clamp(HI63..32)
GPR[rd]79..64     ← clamp(LO95..64)
GPR[rd]95..80     ← clamp(LO127..96)
GPR[rd]111..96    ← clamp(HI95..64)
GPR[rd]127..112   ← clamp(HI127..96)

```

clamp(X)

begin

if (X > 0x00007FFF) then

 clamp := 0x7FFF

else if (X < 0xFFFF8000) then

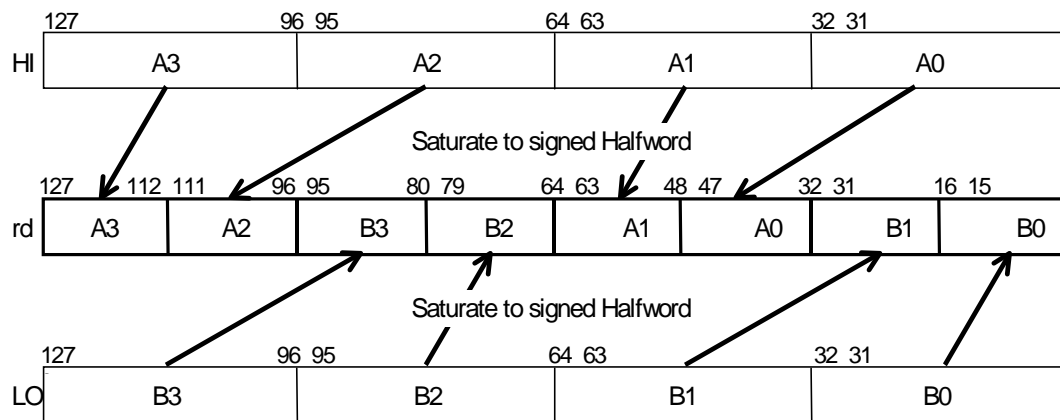
 clamp := 0x8000

else

 clamp := X

endif

end



PMFHL.SLW : Parallel Move From HI/LO Register

128-bit MMI

To copy the contents of HI/LO register to a GPR in word units.

Operation Code

31	26	25	16	15	11	10	6	5	0
MMI	0				rd		fmt		PMFHL
011100	00 0000 0000						00010		110000
6	10				5		5		6

Format

PMFHL.SLW rd

Description

$GPR[rd] \leftarrow HI / LO$

Splits the contents of the HI and LO registers into word data and combines them. Then saturates to 32-bit signed integers and stores them in GPR[rd].

Exceptions

None

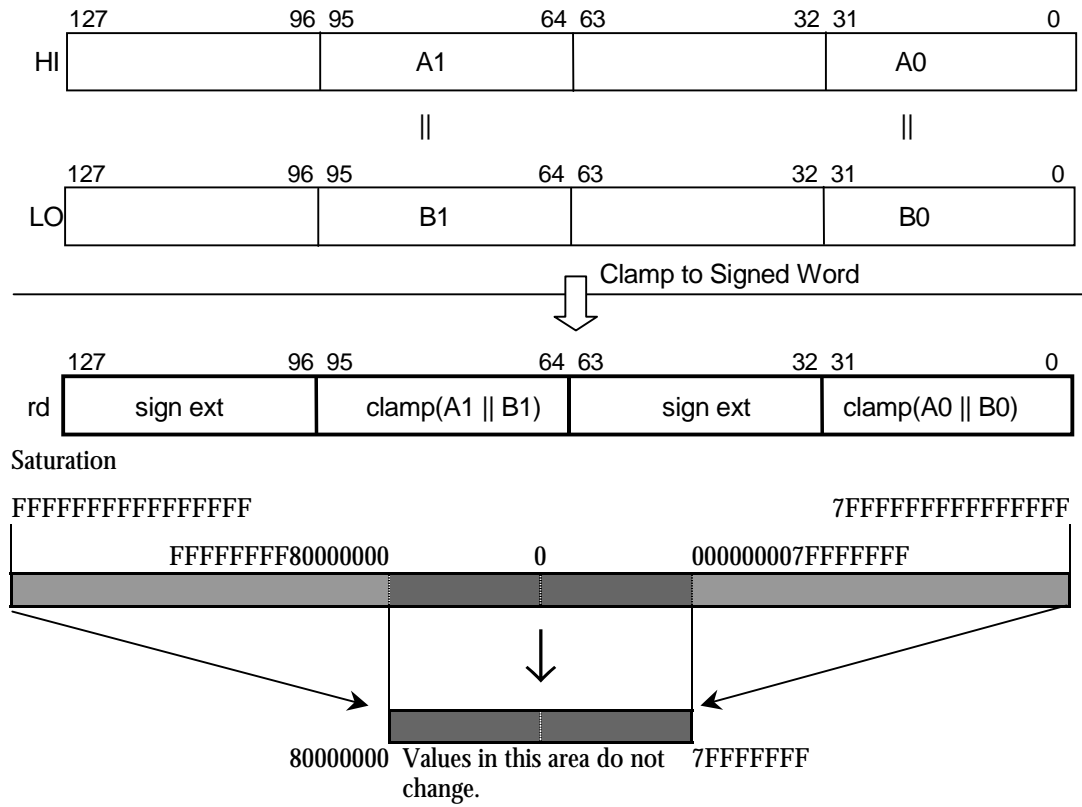
Operation

```

if ( (HI31..0 | LO31..0) >= 0x000000007FFFFFFF ) then
    GPR[rd]63..0 ← 0x000000007FFFFFFF
else if ( (HI31..0 | LO31..0) <= 0xFFFFFFFF80000000 ) then
    GPR[rd]63..0 ← 0xFFFFFFFF80000000
else
    GPR[rd]63..0 ← (LO31)32 | LO31..0
endif

if ( (HI95..64 | LO95..64) >= 0x000000007FFFFFFF ) then
    GPR[rd]127..64 ← 0x000000007FFFFFFF
else if ( (HI95..64 | LO95..64) <= 0xFFFFFFFF80000000 ) then
    GPR[rd]127..64 ← 0xFFFFFFFF80000000
else
    GPR[rd]127..64 ← (LO95)32 | LO95..64
endif

```



PMFHL.UW : Parallel Move From HI/LO Register

128-bit MMI

To copy the contents of the HI/LO registers to a GPR in word units.

Operation Code

31	26	25	16	15	11	10	6	5	0
MMI	0				rd	fmt	PMFHL		
011100	00 0000 0000					00001	110000		
6	10				5	5	6		

Format

PMFHL.UW rd

Description

$GPR[rd] \leftarrow HI / LO$

Splits the contents of HI/LO registers into word data, rearranges them as shown in "Operation", and stores them in GPR[rd].

Exceptions

None

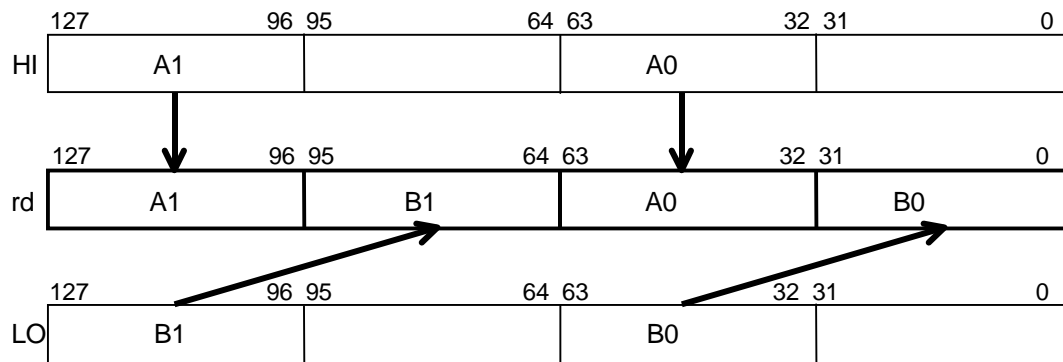
Operation

$GPR[rd]_{31..0} \leftarrow LO_{63..32}$

$GPR[rd]_{63..32} \leftarrow HI_{63..32}$

$GPR[rd]_{95..64} \leftarrow LO_{127..96}$

$GPR[rd]_{127..96} \leftarrow HI_{127..96}$



PMFLO : Parallel Move From LO Register

128-bit MMI

To copy the 128-bit value in the LO register to a GPR.

Operation Code

31	26	25	16	15	11	10	6	5	0
MMI	0			rd	PMFLO		MMI2		
011100	00 0000 0000				01001		001001		
6	10			5	5		6		

Format

PMFLO rd

Description

$GPR[rd] \leftarrow LO$

Stores the 128-bit value in the LO register into GPR[rd].

Exceptions

None

Operation

$GPR[rd]_{127..0} \leftarrow LO_{127..0}$

PMINH : Parallel Minimize Halfword

128-bit MMI

To compare 16-bit signed integers and calculate minimum value (8 parallel operations).

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PMINH	MMI1						
011100				00111	101000						
6	5	5	5	5	6						

Format

PMINH rd, rs, rt

Description

$GPR[rd] \leftarrow \min(GPR[rs], GPR[rt])$

Splits $GPR[rs]$ and $GPR[rt]$ into eight 16-bit signed integers, compares the data in $GPR[rs]$ with the corresponding data in $GPR[rt]$, and stores the minimum value in the corresponding halfwords in $GPR[rd]$.

Exceptions

None

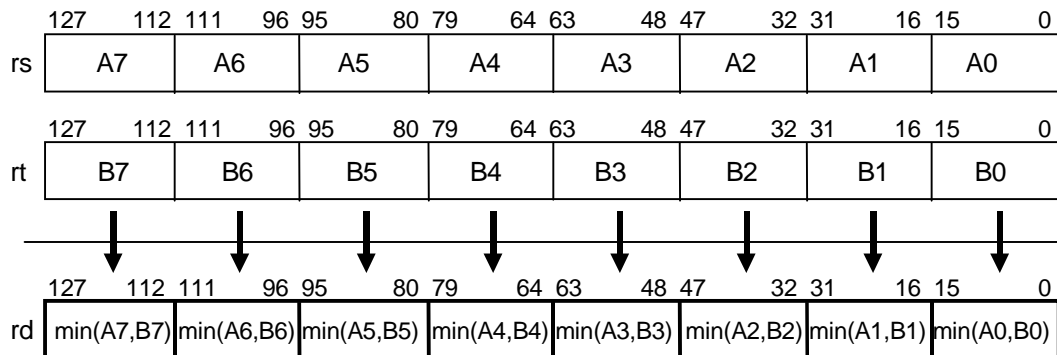
Operation

```
if ((GPR[rs]15:0 > GPR[rt]15:0) then
    GPR[rd]15:0 ← GPR[rt]15:0
else
    GPR[rd]15:0 ← GPR[rs]15:0
endif
```

```
if ((GPR[rs]31:16 > GPR[rt]31:16) then
    GPR[rd]31:16 ← GPR[rt]31:16
else
    GPR[rd]31:16 ← GPR[rs]31:16
endif
```

(The same operations follow every 16 bits)

```
if ((GPR[rs]127:112 > GPR[rt]127:112) then
    GPR[rd]127:112 ← GPR[rt]127:112
else
    GPR[rd]127:112 ← GPR[rs]127:112
endif
```



PMINW : Parallel Minimize Word

128-bit MMI

To compare 32-bit signed integers and calculate minimum value (4 parallel operations).

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PMINW	MMI1						
011100				00011	101000						
6	5	5	5	5	6						

Format

PMINW rd, rs, rt

Description

$GPR[rd] \leftarrow \min(GPR[rs], GPR[rt])$

Splits $GPR[rs]$ and $GPR[rt]$ into four 32-bit signed integers and compares the data in $GPR[rs]$ with the corresponding data in $GPR[rt]$ and stores the minimum value in the corresponding words in $GPR[rd]$.

Exceptions

None

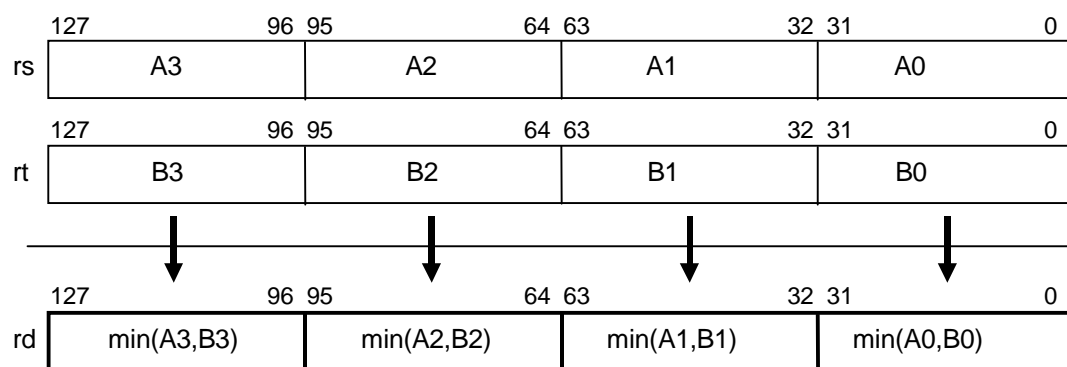
Operation

```
if ((GPR[rs]31..0 > GPR[rt]31..0) then
    GPR[rd]31..0 ← GPR[rt]31..0
else
    GPR[rd]31..0 ← GPR[rs]31..0
endif
```

```
if ((GPR[rs]63..32 > GPR[rt]63..32) then
    GPR[rd]63..32 ← GPR[rt]63..32
else
    GPR[rd]63..32 ← GPR[rs]63..32
endif
```

```
if ((GPR[rs]95..64 > GPR[rt]95..64) then
    GPR[rd]95..64 ← GPR[rt]95..64
else
    GPR[rd]95..64 ← GPR[rs]95..64
endif
```

```
if ((GPR[rs]127..96 > GPR[rt]127..96) then
    GPR[rd]127..96 ← GPR[rt]127..96
else
    GPR[rd]127..96 ← GPR[rs]127..96
endif
```



PMSUBH : Parallel Multiply-Subtract Halfword

128-bit MMI

To multiply 8 pairs of 16-bit signed integers and subtract in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PMSUBH	MMI2						
011100				10100	001001						
6	5	5	5	5	6						

Format

PMSUBH rd, rs, rt

Description

$(\text{GPR}[\text{rd}], \text{HI}, \text{LO}) \leftarrow (\text{HI}, \text{LO}) - \text{GPR}[\text{rs}] \times \text{GPR}[\text{rt}]$

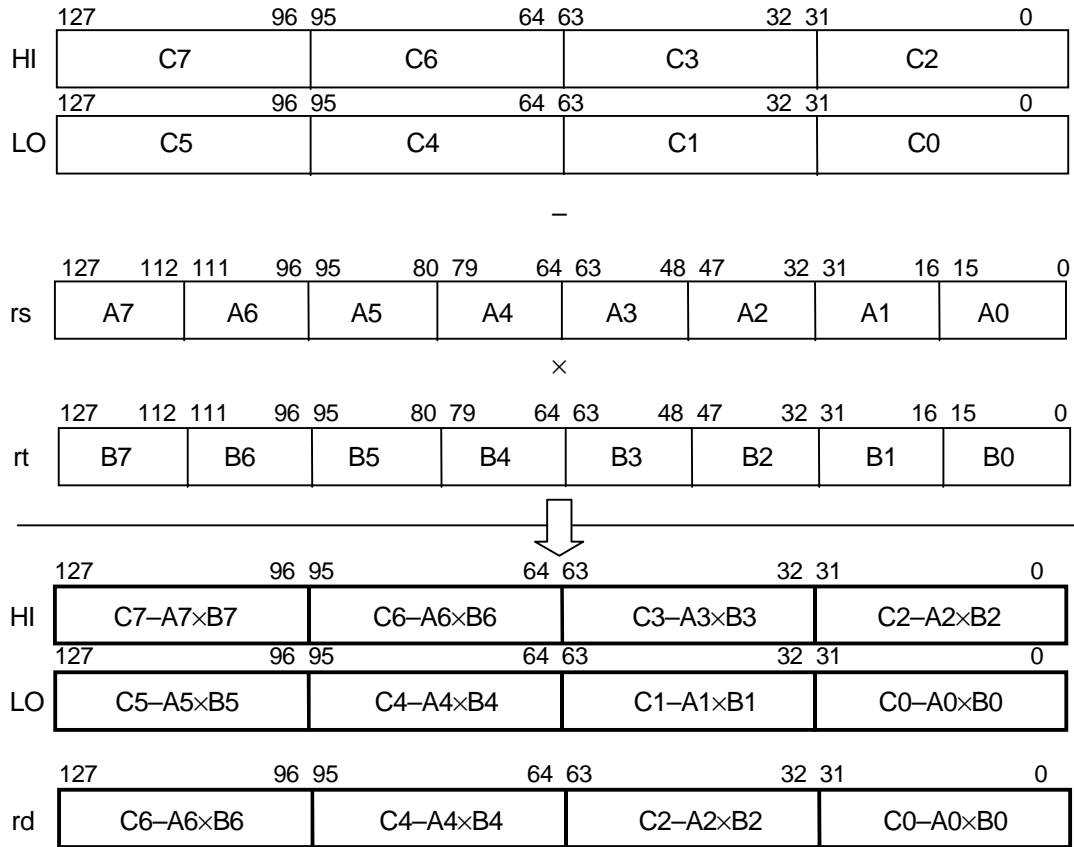
Splits GPR[rs] and GPR[rt] into eight 16-bit signed integers and multiplies each halfword in GPR[rs] by the corresponding halfword in GPR[rt] and subtracts the resulting 32-bit value from the corresponding word in the HI/LO registers. The results of the subtraction are written back to HI/LO registers and a part of them are stored in GPR[rd].

Exceptions

None. Even when the result of the arithmetic operation overflows, an exception does not occur.

Operation

prod0	$\leftarrow \text{LO}_{31..0} - \text{GPR}[\text{rs}]_{15..0} \times \text{GPR}[\text{rt}]_{15..0}$
LO _{31..0}	$\leftarrow \text{prod0}_{31..0}$
GPR[rd] _{31..0}	$\leftarrow \text{prod0}_{31..0}$
prod1	$\leftarrow \text{LO}_{63..32} - \text{GPR}[\text{rs}]_{31..16} \times \text{GPR}[\text{rt}]_{31..16}$
LO _{63..32}	$\leftarrow \text{prod1}_{31..0}$
prod2	$\leftarrow \text{HI}_{31..0} - \text{GPR}[\text{rs}]_{47..32} \times \text{GPR}[\text{rt}]_{47..32}$
HI _{31..0}	$\leftarrow \text{prod2}_{31..0}$
GPR[rd] _{63..32}	$\leftarrow \text{prod2}_{31..0}$
prod3	$\leftarrow \text{HI}_{63..32} - \text{GPR}[\text{rs}]_{63..48} \times \text{GPR}[\text{rt}]_{63..48}$
HI _{63..32}	$\leftarrow \text{prod3}_{31..0}$
prod4	$\leftarrow \text{LO}_{95..64} - \text{GPR}[\text{rs}]_{79..64} \times \text{GPR}[\text{rt}]_{79..64}$
LO _{95..64}	$\leftarrow \text{prod4}_{31..0}$
GPR[rd] _{95..64}	$\leftarrow \text{prod4}_{31..0}$
prod5	$\leftarrow \text{LO}_{127..96} - \text{GPR}[\text{rs}]_{95..80} \times \text{GPR}[\text{rt}]_{95..80}$
LO _{127..96}	$\leftarrow \text{prod5}_{31..0}$
prod6	$\leftarrow \text{HI}_{95..64} - \text{GPR}[\text{rs}]_{111..96} \times \text{GPR}[\text{rt}]_{111..96}$
HI _{95..64}	$\leftarrow \text{prod6}_{31..0}$
GPR[rd] _{127..96}	$\leftarrow \text{prod6}_{31..0}$
prod7	$\leftarrow \text{HI}_{127..96} - \text{GPR}[\text{rs}]_{127..112} \times \text{GPR}[\text{rt}]_{127..112}$
HI _{127..96}	$\leftarrow \text{prod7}_{31..0}$



Programming Notes

In the EE Core, the integer multiply operation proceeds asynchronously. An attempt to read the contents of the LO/HI/GPR[rd] registers before the multiply operation finishes will result in interlock. Other CPU instructions can execute without delay. Therefore, scheduling the multiply operation appropriately can improve performance.

Even when the result of the multiply operation overflows, an overflow exception does not occur. If an overflow is required to be detected, an explicit check is necessary.

PMSUBW : Parallel Multiply-Subtract Word

128-bit MMI

To multiply 2 pairs of 32-bit signed integers and subtract in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PMSUBW	MMI2						
011100				00100	001001						
6	5	5	5	5	6						

Format

PMSUBW rd, rs, rt

Description

$$(GPR[rd], HI, LO) \leftarrow (HI, LO) - GPR[rs] \times GPR[rt]$$

Multiplies bits 95..64 of GPR[rs] by bits 95..64 of GPR[rt] and bits 31..0 of GPR[rs] by bits 31..0 of GPR[rt] as signed 32-bit integers and subtracts the resulting 64-bit value from the value that combines the corresponding word data in the HI and LO registers. A part of the result of subtraction is stored in GPR[rd]. See "Operation" for details.

Restrictions

If the contents of GPR[rt] and GPR[rs] are not sign-extended 32-bit values (bits 127..95 equal and bits 63..31 equal), then the result is undefined.

Exceptions

None. Even when the result of the arithmetic operation overflows, an exception does not occur.

Operation

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif

$$\text{prod0} \leftarrow (HI_{31..0} \parallel LO_{31..0}) - GPR[rs]_{31..0} \times GPR[rt]_{31..0}$$

$$LO_{63..0} \leftarrow (\text{prod0}_{31})^{32} \parallel \text{prod0}_{31..0}$$

$$HI_{63..0} \leftarrow (\text{prod0}_{63})^{32} \parallel \text{prod0}_{63..32}$$

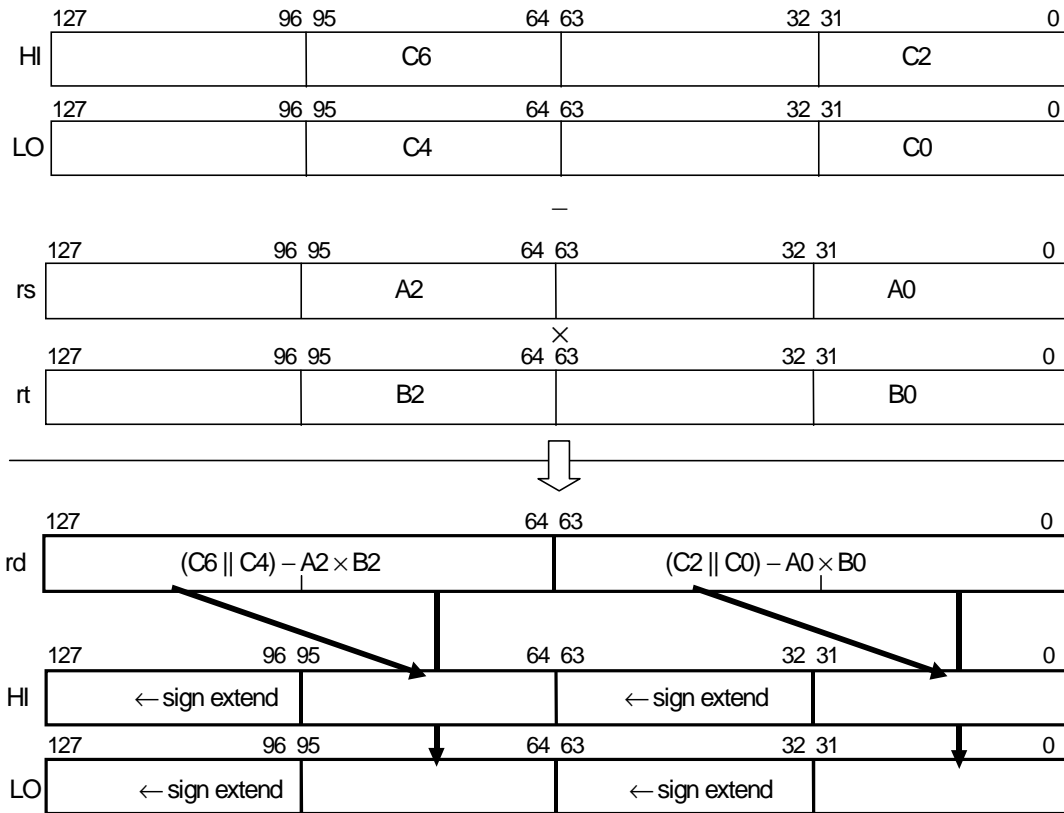
$$GPR[rd]_{63..0} \leftarrow \text{prod0}_{63..0}$$

$$\text{prod1} \leftarrow (HI_{95..64} \parallel LO_{95..64}) - GPR[rs]_{95..64} \times GPR[rt]_{95..64}$$

$$LO_{127..64} \leftarrow (\text{prod1}_{31})^{32} \parallel \text{prod1}_{31..0}$$

$$HI_{127..64} \leftarrow (\text{prod1}_{63})^{32} \parallel \text{prod1}_{63..32}$$

$$GPR[rd]_{127..64} \leftarrow \text{prod1}_{63..0}$$



Programming Notes

In the EE Core, the integer multiply operation proceeds asynchronously. An attempt to read the contents of the LO/HI/GPR[rd] registers before the multiply operation finishes will result in interlock. Other CPU instructions can execute without delay. Therefore, scheduling the multiply operation appropriately can improve performance.

Even when the result of the multiply operation overflows, an overflow exception does not occur. If an overflow is required to be detected, an explicit check is necessary.

PMTHI : Parallel Move To HI Register

128-bit MMI

To copy the value of a GPR to the HI register.

Operation Code

31	26	25	21	20	11	10	6	5	0
MMI						PMTHI		MMI3	
011100						01000		101001	
6						5		6	

Format

PMTHI rs

Description

$HI \leftarrow GPR[rs]$

To store the contents of GPR[rs] in the HI register.

Exceptions

None

Operation

$HI_{127..0} \leftarrow GPR[rs]_{127..0}$

PMTHL.LW : Parallel Move To HI/LO Register

128-bit MMI

To copy the value of a GPR to the HI/LO registers in word units.

Operation Code

31	26	25	21	20	11	10	6	5	0
MMI		rs		0		fmt		PMTHL	
011100				00 0000 0000		00000		110001	
6		5		10		5		6	

Format

PMTHL.LW rs

Description

HI / LO \leftarrow GPR[rs]

Splits GPR[rs] into four words and stores them in the HI and LO registers, as shown in "Operation".

Exceptions

None

Operation

LO_{31..0} \leftarrow GPR[rs]_{31..0}

LO_{63..32} \leftarrow LO_{63..32}

HI_{31..0} \leftarrow GPR[rs]_{63..32}

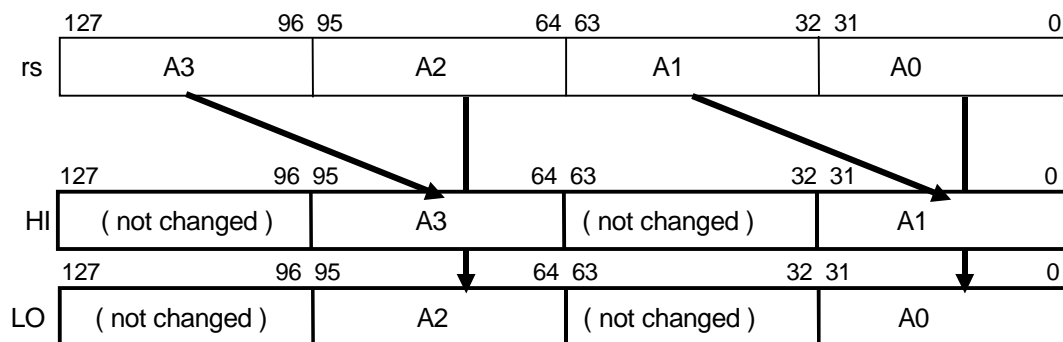
HI_{63..32} \leftarrow HI_{63..32}

LO_{95..64} \leftarrow GPR[rs]_{95..64}

LO_{127..96} \leftarrow LO_{127..96}

HI_{95..64} \leftarrow GPR[rs]_{127..96}

HI_{127..96} \leftarrow HI_{127..96}



PMTLO : Parallel Move To LO Register

128-bit MMI

To copy the value of a GPR to LO register.

Operation Code

31	26	25	21	20	11	10	6	5	0
MMI						rs		0	
011100						00 0000 0000		PMTLO	
6						5		MMI3	
						01001		101001	
						5		6	

Format

PMTLO rs

Description

$LO \leftarrow GPR[rs]$

Stores the contents of GPR[rs] in the LO register.

Exceptions

None

Operation

$LO_{127..0} \leftarrow GPR[rs]_{127..0}$

PMULTH : Parallel Multiply Halfword

128-bit MMI

To multiply 8 pairs of 16-bit signed integers in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PMULTH	MMI2						
011100				11100	001001						
6	5	5	5	5	6						

Format

PMULTH rd, rs, rt

Description

$(\text{GPR}[\text{rd}], \text{LO}, \text{HI}) \leftarrow \text{GPR}[\text{rs}] \times \text{GPR}[\text{rt}]$

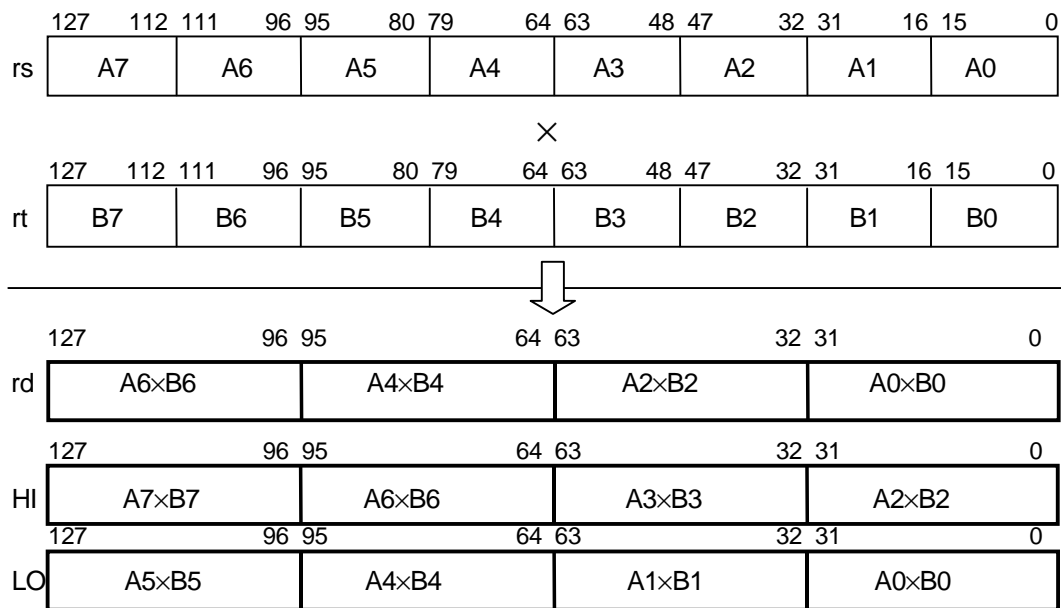
Splits GPR[rs] and GPR[rt] into eight 16-bit signed integers and multiplies the data in GPR[rs] by the corresponding data in GPR[rt] and stores the resulting 32-bit signed integers into the HI/LO registers and GPR[rd], as shown in "Operation".

Exceptions

None

Operation

prod0	$\leftarrow \text{GPR}[\text{rs}]_{15..0} \times \text{GPR}[\text{rt}]_{15..0}$
LO _{31..0}	$\leftarrow \text{prod0}_{31..0}$
GPR[rd] _{31..0}	$\leftarrow \text{prod0}_{31..0}$
prod1	$\leftarrow \text{GPR}[\text{rs}]_{31..16} \times \text{GPR}[\text{rt}]_{31..16}$
LO _{63..32}	$\leftarrow \text{prod1}_{31..0}$
prod2	$\leftarrow \text{GPR}[\text{rs}]_{47..32} \times \text{GPR}[\text{rt}]_{47..32}$
HI _{31..0}	$\leftarrow \text{prod2}_{31..0}$
GPR[rd] _{63..32}	$\leftarrow \text{prod2}_{31..0}$
prod3	$\leftarrow \text{GPR}[\text{rs}]_{63..48} \times \text{GPR}[\text{rt}]_{63..48}$
HI _{63..32}	$\leftarrow \text{prod3}_{31..0}$
prod4	$\leftarrow \text{GPR}[\text{rs}]_{79..64} \times \text{GPR}[\text{rt}]_{79..64}$
LO _{95..64}	$\leftarrow \text{prod4}_{31..0}$
GPR[rd] _{95..64}	$\leftarrow \text{prod4}_{31..0}$
prod5	$\leftarrow \text{GPR}[\text{rs}]_{95..80} \times \text{GPR}[\text{rt}]_{95..80}$
LO _{127..96}	$\leftarrow \text{prod5}_{31..0}$
prod6	$\leftarrow \text{GPR}[\text{rs}]_{111..96} \times \text{GPR}[\text{rt}]_{111..96}$
HI _{95..64}	$\leftarrow \text{prod6}_{31..0}$
GPR[rd] _{127..96}	$\leftarrow \text{prod6}_{31..0}$
prod7	$\leftarrow \text{GPR}[\text{rs}]_{127..112} \times \text{GPR}[\text{rt}]_{127..112}$
HI _{127..96}	$\leftarrow \text{prod7}_{31..0}$

**Programming Notes**

In the EE Core, the integer multiply operation proceeds asynchronously. An attempt to read the contents of the LO/HI/GPR[rd] registers before the multiply operation finishes will result in interlock. Other CPU instructions can execute without delay. Therefore, scheduling the multiply operation appropriately can improve performance.

PMULTUW : Parallel Multiply Unsigned Word

128-bit MMI

To multiply 2 pairs of 32-bit unsigned integers in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PMULTUW	MMI3						
011100				01100	101001						
6	5	5	5	5	6						

Format

PMULTUW rd, rs, rt

Description

Multiplies bits 95-64 of GPR[rs] by bits 95-64 of GPR[rt] and bits 31-0 of GPR[rs] by bits 31-0 of GPR[rt] as unsigned 32-bit integers. The resulting 64-bit value is stored in GPR[rd] and also in the corresponding words in the HI and LO registers, as shown in "Operation".

Restrictions

If the contents of GPR[rt] and GPR[rs] are not sign-extended 32-bit values (bits 127-95 equal and bits 63-31 equal), then the result is undefined.

Exceptions

None

Operation

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif

prod0 $\leftarrow (0 \parallel \text{GPR[rs]}_{31..0}) \times (0 \parallel \text{GPR[rt]}_{31..0})$

LO_{63..0} $\leftarrow (\text{prod0}_{31})^{32} \parallel \text{prod0}_{31..0}$

HI_{63..0} $\leftarrow (\text{prod0}_{63})^{32} \parallel \text{prod0}_{63..32}$

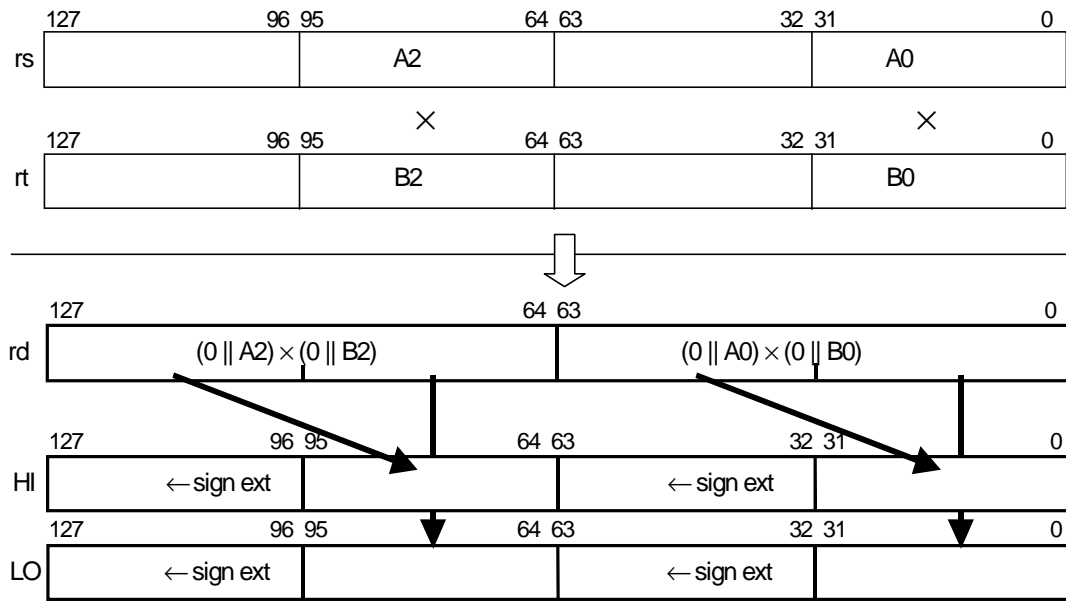
GPR[rd]_{63..0} $\leftarrow \text{prod0}_{63..0}$

prod1 $\leftarrow (0 \parallel \text{GPR[rs]}_{95..64}) \times (0 \parallel \text{GPR[rt]}_{95..64})$

LO_{127..64} $\leftarrow (\text{prod1}_{31})^{32} \parallel \text{prod1}_{31..0}$

HI_{127..64} $\leftarrow (\text{prod1}_{63})^{32} \parallel \text{prod1}_{63..32}$

GPR[rd]_{127..64} $\leftarrow \text{prod1}_{63..0}$



Programming Notes

In the EE Core, the integer multiply operation proceeds asynchronously. An attempt to read the contents of the LO/HI/GPR[rd] registers before the multiply operation finishes will result in interlock. Other CPU instructions can execute without delay. Therefore, scheduling the multiply operation appropriately can improve performance.

PMULTW : Parallel Multiply Word

128-bit MMI

To multiply 2 pairs of 32-bit signed integers in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PMULTW	MMI2						
011100				01100	001001						
6	5	5	5	5	6						

Format

PMULTW rd, rs, rt

Description

Multiplies bits 95-.64 of GPR[rs] by bits 95-64 of GPR[rt] and bits 31-0 of GPR[rs] by bits 31-0 of GPR[rt] as signed 32-bit integers and the resulting 64-bit value is stored into GPR[rd] and the corresponding words in the HI and LO registers, as shown in "Operation".

Restrictions

If the contents of GPR[rt] and GPR[rs] are not sign-extended 32-bit values (bits 127..95 equal and bits 63..31 equal), then the result is undefined.

Exceptions

None

Operation

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif

prod0 \leftarrow GPR[rs]_{31..0} \times GPR[rt]_{31..0}

LO_{63..0} \leftarrow (prod0₃₁)³² || prod0_{31..0}

HI_{63..0} \leftarrow (prod0₆₃)³² || prod0_{63..32}

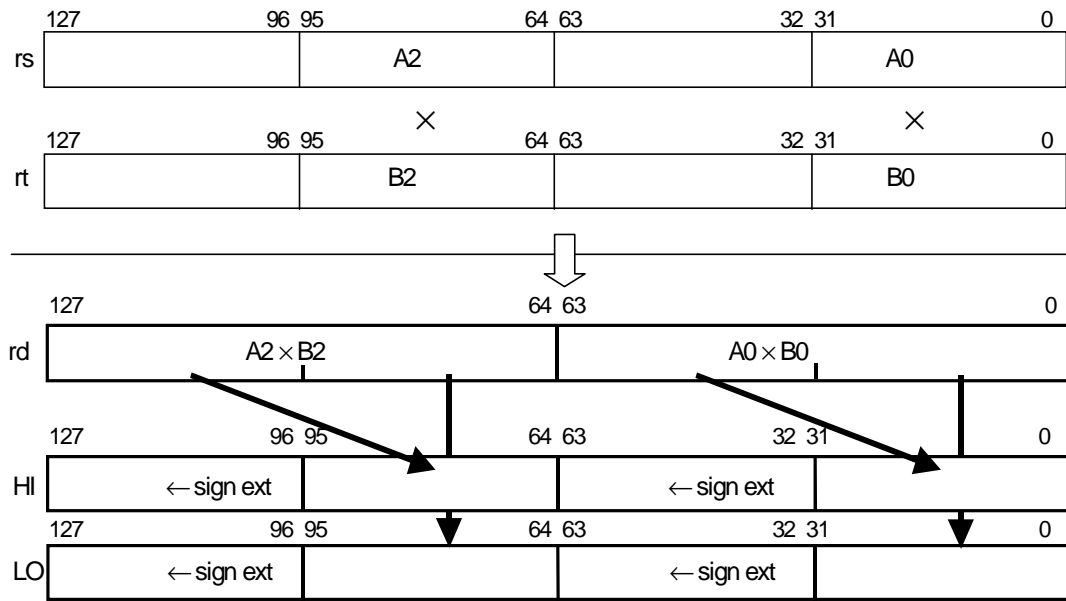
GPR[rd]_{63..0} \leftarrow prod0_{63..0}

prod1 \leftarrow GPR[rs]_{95..64} \times GPR[rt]_{95..64}

LO_{127..64} \leftarrow (prod1₃₁)³² || prod1_{31..0}

HI_{127..64} \leftarrow (prod1₆₃)³² || prod1_{63..32}

GPR[rd]_{127..64} \leftarrow prod1_{63..0}



Programming Notes

In the EE Core, the integer multiply operation proceeds asynchronously. An attempt to read the contents of the LO/HI/GPR[rd] registers before the multiply operation finishes will result in interlock. Other CPU instructions can execute without delay. Therefore, scheduling the multiply operation appropriately can improve performance.

PNOR : Parallel Not Or

128-bit MMI

To calculate a bitwise logical NOT OR.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI 011100		rs		rt		rd		PNOR 10011		MMI3 101001	
6		5		5		5		5		6	

Format

PNOR rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] \text{ NOR } GPR[rt]$

Calculates a bitwise logical NOR between the contents of GPR[rs] and GPR[rt]. The result is stored in GPR[rd].

The truth table values for NOR are as follows;

X	Y	X NOR Y
0	0	1
0	1	0
1	0	0
1	1	0

Exceptions

None

Operation

$GPR[rd]_{127..0} \leftarrow GPR[rs]_{127..0} \text{ NOR } GPR[rt]_{127..0}$

POR : Parallel Or

128-bit MMI

To calculate a bitwise logical OR.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI 011100						rs			rt		
						rd			POR 10010		
									MMI3 101001		
6						5			5		

Format

POR rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] \text{ OR } GPR[rt]$

Calculates a bitwise logical OR between the contents of GPR[rs] and GPR[rt]. The result is stored into GPR[rd].

The truth table values for OR are as follows;

X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

Exceptions

None

Operation

$GPR[rd]_{127..0} \leftarrow GPR[rs]_{127..0} \text{ OR } GPR[rt]_{127..0}$

PPAC5 : Parallel Pack to 5 bits

128-bit MMI

To pack 4 words in the 8-8-8-8 bit format into 4 halfwords in the 1-5-5-5 bit format.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	0	rt	rd	PPAC5	MMI0						
011100	00000			11111	001000						
6	5	5	5	5	6						

Format

PPAC5 rd, rt

Description

$GPR[rd] \leftarrow \text{pack}(GPR[rt])$

Splits the 128-bit data in GPR[rt] into four words in the 8-8-8-8 bit format. Each of the low-order bits are truncated and packed into four halfwords in the 1-5-5-5 bit format. The result is stored in GPR[rd].

Exceptions

None

Operation

$GPR[rd]_{15..0} \leftarrow GPR[rt]_{31} \mid \mid GPR[rt]_{23..19} \mid \mid GPR[rt]_{15..11} \mid \mid GPR[rt]_{7..3}$

$GPR[rd]_{31..16} \leftarrow 0^{16}$

$GPR[rd]_{47..32} \leftarrow GPR[rt]_{63} \mid \mid GPR[rt]_{55..51} \mid \mid GPR[rt]_{47..43} \mid \mid GPR[rt]_{39..35}$

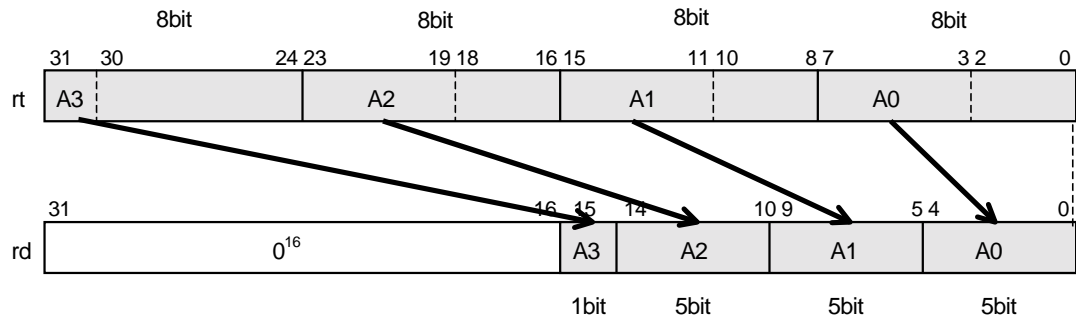
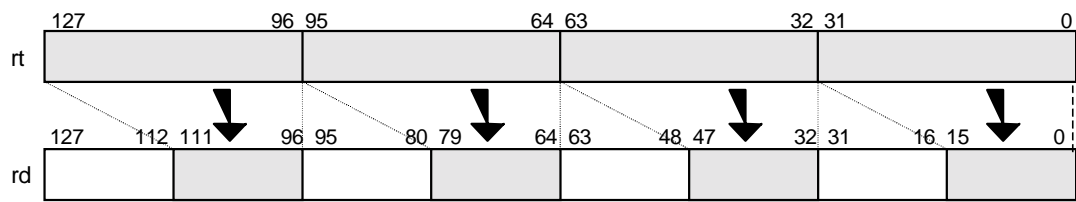
$GPR[rd]_{63..48} \leftarrow 0^{16}$

$GPR[rd]_{79..64} \leftarrow GPR[rt]_{95} \mid \mid GPR[rt]_{87..83} \mid \mid GPR[rt]_{79..75} \mid \mid GPR[rt]_{71..67}$

$GPR[rd]_{95..80} \leftarrow 0^{16}$

$GPR[rd]_{111..96} \leftarrow GPR[rt]_{127} \mid \mid GPR[rt]_{119..115} \mid \mid GPR[rt]_{111..107} \mid \mid GPR[rt]_{103..99}$

$GPR[rd]_{127..112} \leftarrow 0^{16}$



PPACB : Parallel Pack to Byte

128-bit MMI

To pack the data in two GPRs into consecutive bytes.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI		rs		rt		rd		PPACB		MMI0	
011100								11011		001000	
6		5		5		5		5		6	

Format

PPACB rd, rs, rt

Description

$GPR[rd] \leftarrow \text{pack}(GPR[rs], GPR[rt])$

Splits $GPR[rs]$ into eight halfwords, takes the low-order bytes of each of the halfwords and stores them in the high-order 64 bits of $GPR[rd]$. Likewise, takes the low-order bytes of each of the halfwords of $GPR[rt]$ and stores them in the low-order 64 bits of $GPR[rd]$.

Exceptions

None

Operation

$GPR[rd]_{7..0} \leftarrow GPR[rt]_{7..0}$

$GPR[rd]_{15..8} \leftarrow GPR[rt]_{23..16}$

(The same operations follow every 8 bits)

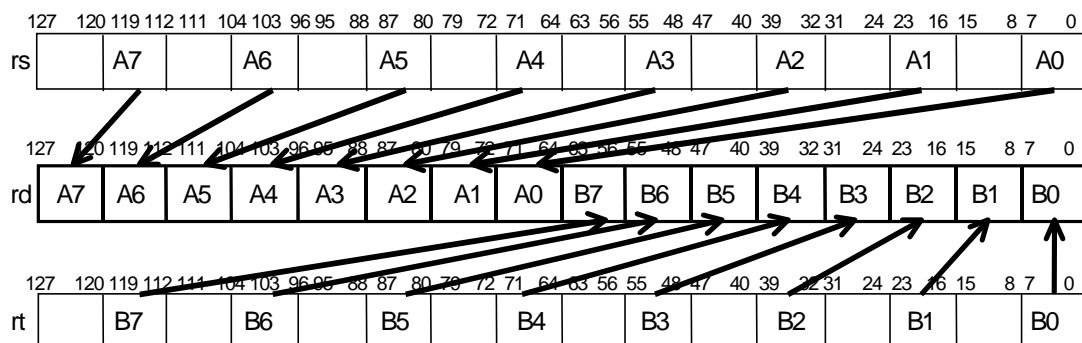
$GPR[rd]_{63..56} \leftarrow GPR[rt]_{119..112}$

$GPR[rd]_{71..64} \leftarrow GPR[rs]_{7..0}$

$GPR[rd]_{79..72} \leftarrow GPR[rs]_{23..16}$

(The same operations follow every 8 bits)

$GPR[rd]_{127..120} \leftarrow GPR[rs]_{119..112}$



PPACH : Parallel Pack to Halfword

128-bit MMI

To pack the data in two GPRs into consecutive halfwords.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PPACH	MMI0						
011100				10111	001000						
6	5	5	5	5	6						

Format

PPACH rd, rs, rt

Description

$GPR[rd] \leftarrow \text{pack}(GPR[rs], GPR[rt])$

Splits $GPR[rs]$ into four words, takes the low-order halfword of each of the words and stores them in the high-order 64 bits of $GPR[rd]$. Likewise, takes the low-order halfwords of each of the words in $GPR[rt]$ and stores them in the low-order 64 bits of $GPR[rd]$.

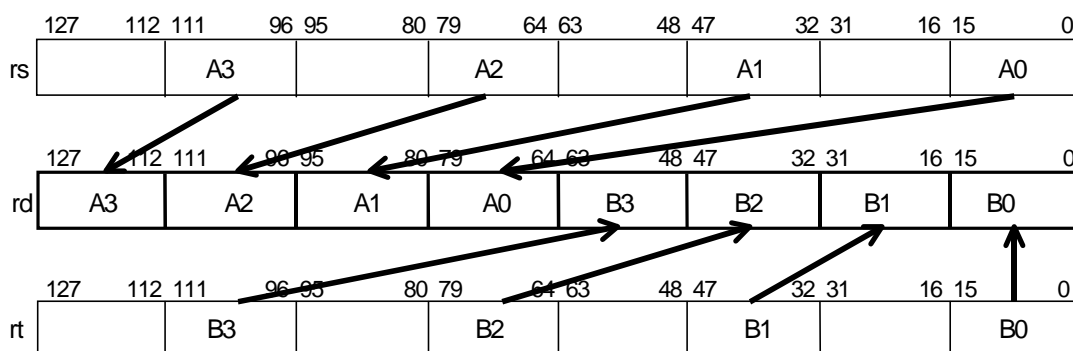
Exceptions

None

Operation

$GPR[rd]_{15..0} \leftarrow GPR[rt]_{15..0}$
 $GPR[rd]_{31..16} \leftarrow GPR[rt]_{47..32}$
 $GPR[rd]_{47..32} \leftarrow GPR[rt]_{79..64}$
 $GPR[rd]_{63..48} \leftarrow GPR[rt]_{111..96}$

$GPR[rd]_{79..64} \leftarrow GPR[rs]_{15..0}$
 $GPR[rd]_{95..80} \leftarrow GPR[rs]_{47..32}$
 $GPR[rd]_{111..96} \leftarrow GPR[rs]_{79..64}$
 $GPR[rd]_{127..112} \leftarrow GPR[rs]_{111..96}$



PPACW : Parallel Pack to Word

128-bit MMI

To pack the data in two GPRs into consecutive words.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI		rs		rt		rd		PPACW		MMI0	
011100								10011		001000	
6		5		5		5		5		6	

Format

PPACW rd, rs, rt

Description

$GPR[rd] \leftarrow \text{pack}(GPR[rs], GPR[rt])$

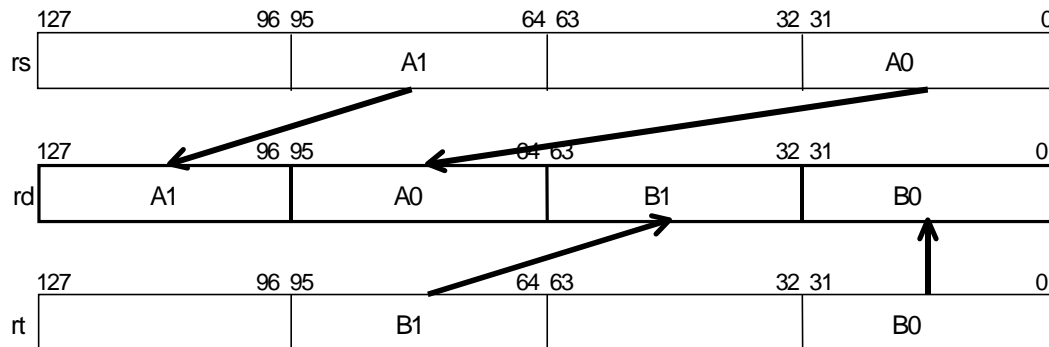
Splits $GPR[rs]$ into two doublewords, takes the low-order word of each of the doublewords and stores them in the high-order 64 bits of $GPR[rd]$. Likewise, takes the low-order word of each of the doublewords of $GPR[rt]$ and stores them in the low-order 64-bit in $GPR[rd]$.

Exceptions

None

Operation

$GPR[rd]_{31..0} \leftarrow GPR[rt]_{31..0}$
 $GPR[rd]_{63..32} \leftarrow GPR[rt]_{95..64}$
 $GPR[rd]_{95..64} \leftarrow GPR[rs]_{31..0}$
 $GPR[rd]_{127..96} \leftarrow GPR[rs]_{95..64}$



PREVH : Parallel Reverse Halfword

128-bit MMI

To exchange the position of halfwords.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI						0		rt		rd	
011100						00000		11011		MMI2	
6						5		5		6	

Format

PREVH rd, rt

Description

$GPR[rd] \leftarrow \text{exchange}(GPR[rt])$

Splits $GPR[rt]$ into eight halfwords, exchanges their sequence and stores them in $GPR[rd]$.

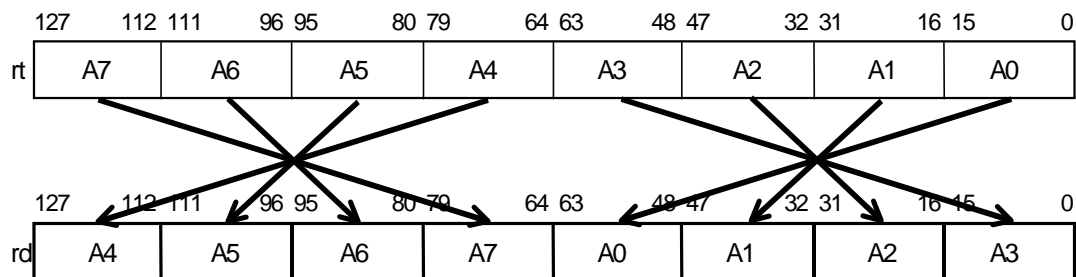
See "Operation" about the details of the exchange.

Exceptions

None

Operation

$GPR[rd]_{15..0} \leftarrow GPR[rt]_{63..48}$
 $GPR[rd]_{31..16} \leftarrow GPR[rt]_{47..32}$
 $GPR[rd]_{47..32} \leftarrow GPR[rt]_{31..16}$
 $GPR[rd]_{63..48} \leftarrow GPR[rt]_{15..0}$
 $GPR[rd]_{79..64} \leftarrow GPR[rt]_{127..112}$
 $GPR[rd]_{95..80} \leftarrow GPR[rt]_{111..96}$
 $GPR[rd]_{111..96} \leftarrow GPR[rt]_{95..80}$
 $GPR[rd]_{127..112} \leftarrow GPR[rt]_{79..64}$



PROT3W : Parallel Rotate 3 Words Left

128-bit MMI

To exchange the sequence of 3 words in 128-bit data.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI						0		rt		rd	
011100						00000		11111		MMI2	
6						5		5		6	

Format

PROT3W rd, rt

Description

$GPR[rd] \leftarrow \text{rotate}(GPR[rt])$

Splits $GPR[rt]$ into four words, rotates the positions of the low-order three words, and stores them in $GPR[rd]$.

Exceptions

None

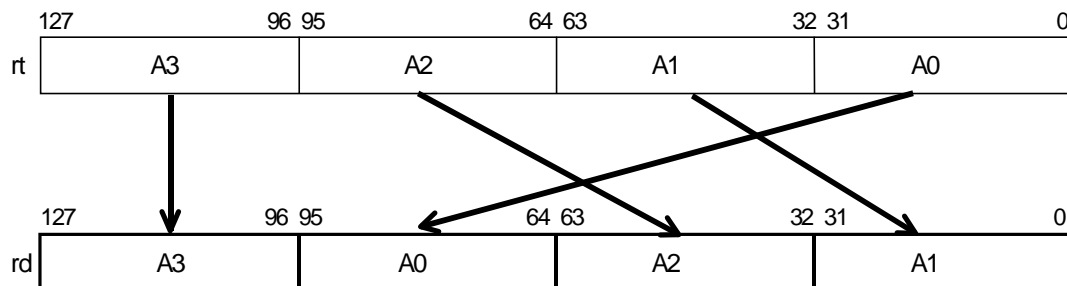
Operation

$GPR[rd]_{31..0} \leftarrow GPR[rt]_{63..32}$

$GPR[rd]_{63..32} \leftarrow GPR[rt]_{95..64}$

$GPR[rd]_{95..64} \leftarrow GPR[rt]_{31..0}$

$GPR[rd]_{127..96} \leftarrow GPR[rt]_{127..96}$



PSLLH : Parallel Shift Left Logical Halfword

128-bit MMI

To logical shift left halfwords in 128-bit data. The shift amount is a fixed value (0-15 bits) specified by sa.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	0	rt	rd	sa	PSLLH						
011100	00000				110100						
6	5	5	5	5	6						

Format

PSLLH rd, rt, sa

Description

$GPR[rd] \leftarrow GPR[rt] \ll sa$ (logical)

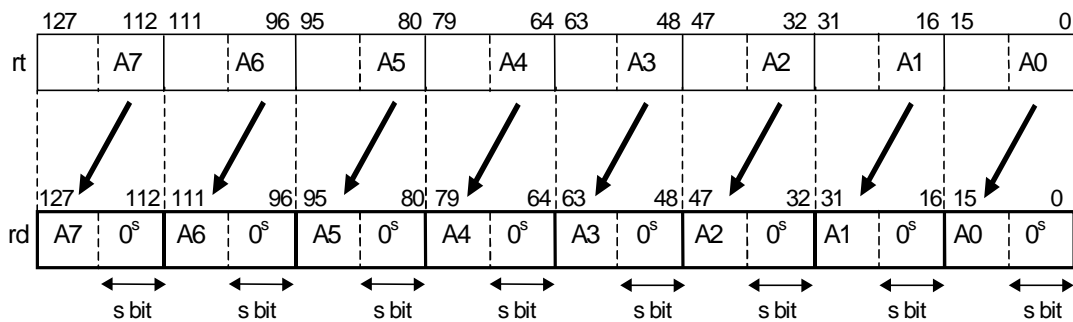
Splits $GPR[rt]$ into eight halfwords, shifts them left by the number of bits specified by the low-order 4 bits of sa and inserts zeros in the emptied bits. The resulting value is stored in the corresponding halfwords in $GPR[rd]$.

Exceptions

None

Operation

$s \leftarrow sa_{3:0}$
 $GPR[rd]_{15:0} \leftarrow GPR[rt]_{15:s..0} \parallel 0^s$
 $GPR[rd]_{31:16} \leftarrow GPR[rt]_{31:s..16} \parallel 0^s$
 $GPR[rd]_{47:32} \leftarrow GPR[rt]_{47:s..32} \parallel 0^s$
 $GPR[rd]_{63:48} \leftarrow GPR[rt]_{63:s..48} \parallel 0^s$
 $GPR[rd]_{79:64} \leftarrow GPR[rt]_{79:s..64} \parallel 0^s$
 $GPR[rd]_{95:80} \leftarrow GPR[rt]_{95:s..80} \parallel 0^s$
 $GPR[rd]_{111:96} \leftarrow GPR[rt]_{111:s..96} \parallel 0^s$
 $GPR[rd]_{127:112} \leftarrow GPR[rt]_{127:s..112} \parallel 0^s$



PSLLVW : Parallel Shift Left Logical Variable Word

128-bit MMI

To shift left 2 words in 128-bit data. The shift amount is specified by a GPR.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PSLLVW	MMI2						
011100				00010	001001						
6	5	5	5	5	6						

Format

PSLLVW rd, rt, rs

Description

$GPR[rd] \leftarrow GPR[rt] \ll GPR[rs]$ (logical)

Splits $GPR[rt]$ into two doublewords and shifts each of the low-order words left. The shift amount is specified by the low-order 5 bits of the corresponding word in $GPR[rs]$. Inserts zeros in the emptied bits.

The resulting two 32-bit values are sign-extended and stored in $GPR[rd]$.

Exceptions

None

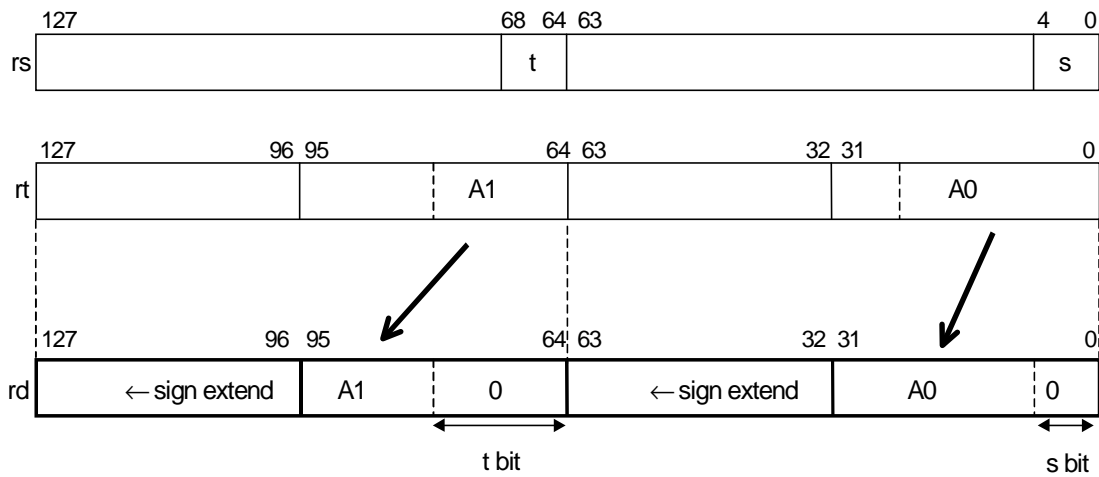
Operation

$s \leftarrow GPR[rs]_{4:0}$

$t \leftarrow GPR[rs]_{68:64}$

$GPR[rd]_{63:0} \leftarrow (GPR[rt]_{31:s})^{32} \parallel GPR[rt]_{31:s..0} \parallel 0^s$

$GPR[rd]_{127:64} \leftarrow (GPR[rt]_{95:t})^{32} \parallel GPR[rt]_{95:t..64} \parallel 0^t$



PSLLW : Parallel Shift Left Logical Word

128-bit MMI

To logically shift left four words in 128-bit data. The shift amount is a fixed value specified by sa.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	0	rt	rd	sa	PSLLW						
011100	00000				111100						
6	5	5	5	5	6						

Format

PSLLW rd, rt, sa

Description

$GPR[rd] \leftarrow GPR[rt] \ll sa$ (logical)

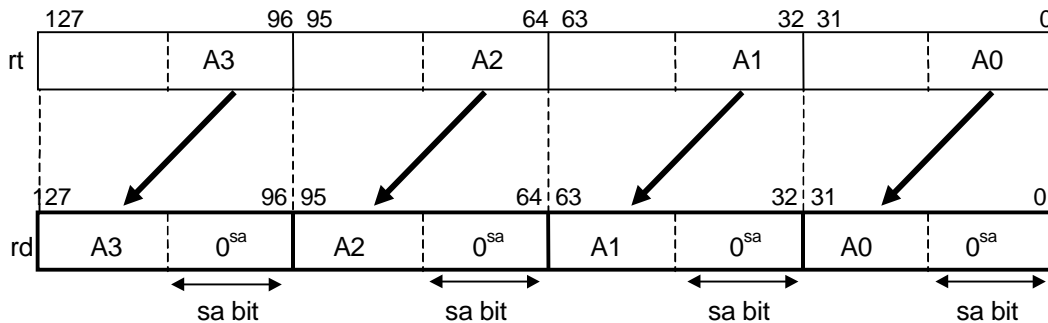
Splits GPR[rt] into four words, shifts left by the number of bits specified by sa and inserts zeros in the emptied bits. The resulting value is stored in the corresponding words in GPR[rd].

Exceptions

None

Operation

$GPR[rd]_{31..0} \leftarrow GPR[rt]_{31-sa..0} \parallel 0^{sa}$
 $GPR[rd]_{63..32} \leftarrow GPR[rt]_{63-sa..32} \parallel 0^{sa}$
 $GPR[rd]_{95..64} \leftarrow GPR[rt]_{95-sa..64} \parallel 0^{sa}$
 $GPR[rd]_{127..96} \leftarrow GPR[rt]_{127-sa..96} \parallel 0^{sa}$



PSRAH : Parallel Shift Right Arithmetic Halfword

128-bit MMI

To arithmetically shift right 8 halfwords in 128-bit data. The shift amount is a fixed value (0-15 bits) specified by sa.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	0		rt		rd		sa		PSRAH		
011100	00000								110111		
6	5		5		5		5		5		6

Format

PSRAH rd, rt, sa

Description

$GPR[rd] \leftarrow GPR[rt] \gg sa$ (arithmetic)

Splits $GPR[rt]$ into eight halfword data, shifts right by the bit number specified by the low-order 4 bits of sa and duplicates the sign bits in the emptied bits. The resulting values are stored in the corresponding halfwords in $GPR[rd]$.

Restrictions

The value of sa must be within the range from 0 to 15. Otherwise, the result is undefined.

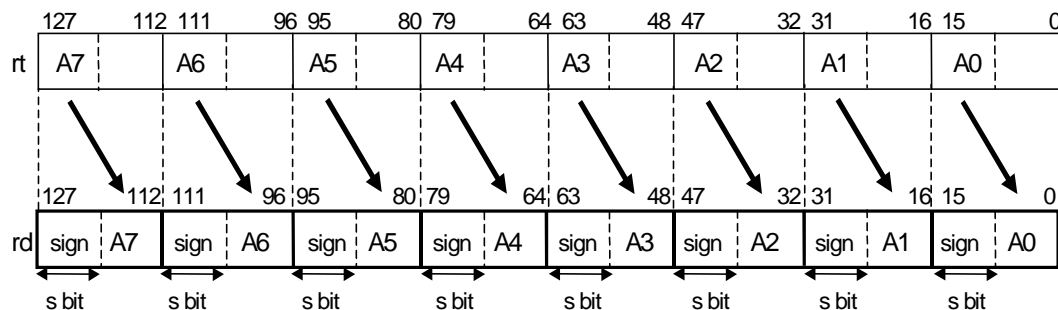
Exceptions

None

Operation

$s \leftarrow sa_{3:0}$

$GPR[rd]_{15:0} \leftarrow (GPR[rt]_{15})^s \parallel GPR[rt]_{15:s}$
 $GPR[rd]_{31:16} \leftarrow (GPR[rt]_{31})^s \parallel GPR[rt]_{31:16+s}$
 $GPR[rd]_{47:32} \leftarrow (GPR[rt]_{47})^s \parallel GPR[rt]_{47:32+s}$
 $GPR[rd]_{63:48} \leftarrow (GPR[rt]_{63})^s \parallel GPR[rt]_{63:48+s}$
 $GPR[rd]_{79:64} \leftarrow (GPR[rt]_{79})^s \parallel GPR[rt]_{79:64+s}$
 $GPR[rd]_{95:80} \leftarrow (GPR[rt]_{95})^s \parallel GPR[rt]_{95:80+s}$
 $GPR[rd]_{111:96} \leftarrow (GPR[rt]_{111})^s \parallel GPR[rt]_{111:96+s}$
 $GPR[rd]_{127:112} \leftarrow (GPR[rt]_{127})^s \parallel GPR[rt]_{127:112+s}$



PSRAVW : Parallel Shift Right Arithmetic Variable Word

128-bit MMI

To arithmetically shift right 2 words in a 128-bit data. The shift amount is specified by a GPR.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PSRAVW	MMI3						
011100				00011	101001						
6	5	5	5	5	6						

Format

PSRAVW rd, rt, rs

Description

$GPR[rd] \leftarrow GPR[rt] \gg GPR[rs]$ (arithmetic)

Splits $GPR[rt]$ into two doublewords and shifts each of the low-order words right. The shift amount is specified by the low-order 5 bits of the corresponding word in $GPR[rs]$. Duplicates the sign bits into the emptied bits. The resulting two 32-bit values are sign-extended and stored in $GPR[rd]$.

Exceptions

None

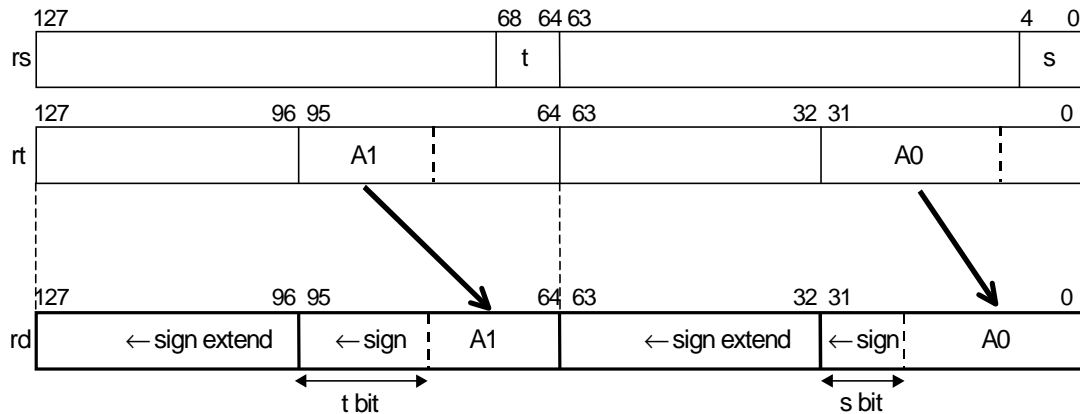
Operation

$s \leftarrow GPR[rs]_{4..0}$

$t \leftarrow GPR[rs]_{68..64}$

$GPR[rd]_{63..0} \leftarrow (GPR[rt]_{31})^{32} \parallel (GPR[rt]_{31})^s \parallel GPR[rt]_{31..s}$

$GPR[rd]_{127..64} \leftarrow (GPR[rt]_{95})^{32} \parallel (GPR[rt]_{95})^t \parallel GPR[rt]_{95..64+t}$



PSRAW : Parallel Shift Right Arithmetic Word

128-bit MMI

To arithmetically shift right 4 words in 128-bit data. The shift amount is a fixed value specified by sa.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	0	rt	rd	sa	PSRAW						
011100	00000				11111						
6	5	5	5	5	6						

Format

PSRAW rd, rt, sa

Description

$GPR[rd] \leftarrow GPR[rt] \gg sa$ (arithmetic)

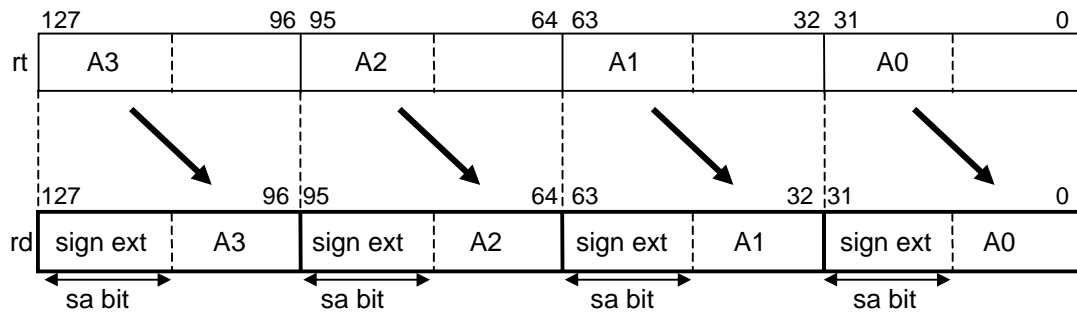
Splits GPR[rt] into four words, shifts them right by the number of bits specified by sa and duplicates the sign bits in the emptied bits. The resulting value is stored in the corresponding words of GPR[rd].

Exceptions

None

Operation

$GPR[rd]_{31..0} \leftarrow (GPR[rt]_{31})^{sa} \parallel GPR[rt]_{31..sa}$
 $GPR[rd]_{63..32} \leftarrow (GPR[rt]_{63})^{sa} \parallel GPR[rt]_{63..32+sa}$
 $GPR[rd]_{95..64} \leftarrow (GPR[rt]_{95})^{sa} \parallel GPR[rt]_{95..64+sa}$
 $GPR[rd]_{127..96} \leftarrow (GPR[rt]_{127})^{sa} \parallel GPR[rt]_{127..96+sa}$



PSRLH : Parallel Shift Right Logical Halfword

128-bit MMI

To logically shift right 8 halfwords in 128-bit data. The shift amount is a fixed value (0-15 bits) specified by sa.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	0	rt	rd	sa	PSRLH						
011100	00000				110110						
6	5	5	5	5	6						

Format

PSRLH rd, rt, sa

Description

$GPR[rd] \leftarrow GPR[rt] \gg sa$ (logical)

Splits $GPR[rt]$ into eight halfwords, shifts right by the number of bits specified by the low-order 4 bits of sa and inserts zeros into the emptied bits. The resulting value is stored in the corresponding halfwords in $GPR[rd]$.

Restrictions

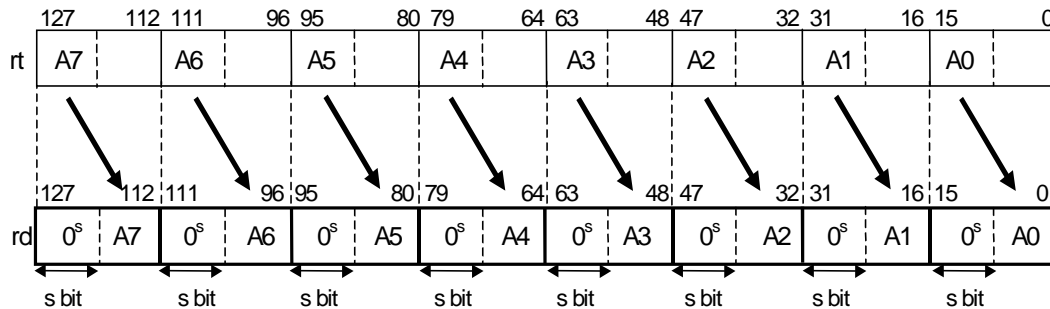
The value of sa must be within the range from 0 to 15. Otherwise, the result is undefined.

Exceptions

None

Operation

$s \leftarrow sa_{3,0}$
 $GPR[rd]_{15..0} \leftarrow 0^s \parallel GPR[rt]_{15..s}$
 $GPR[rd]_{31..16} \leftarrow 0^s \parallel GPR[rt]_{31..16+s}$
 $GPR[rd]_{47..32} \leftarrow 0^s \parallel GPR[rt]_{47..32+s}$
 $GPR[rd]_{63..48} \leftarrow 0^s \parallel GPR[rt]_{63..48+s}$
 $GPR[rd]_{79..64} \leftarrow 0^s \parallel GPR[rt]_{79..64+s}$
 $GPR[rd]_{95..80} \leftarrow 0^s \parallel GPR[rt]_{95..80+s}$
 $GPR[rd]_{111..96} \leftarrow 0^s \parallel GPR[rt]_{111..96+s}$
 $GPR[rd]_{127..112} \leftarrow 0^s \parallel GPR[rt]_{127..112+s}$



PSRLVW : Parallel Shift Right Logical Variable Word

128-bit MMI

To logically shift right 2 words in 128-bit data. The shift amount is specified by a GPR.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI 011100						rs				rt	
6						5				5	
						rd				PSRLVW 00011	
										5	
										MMI2 001001	
										6	

Format

PSRLVW rd, rt, rs

Description

$GPR[rd] \leftarrow GPR[rt] \gg GPR[rs]$ (logical)

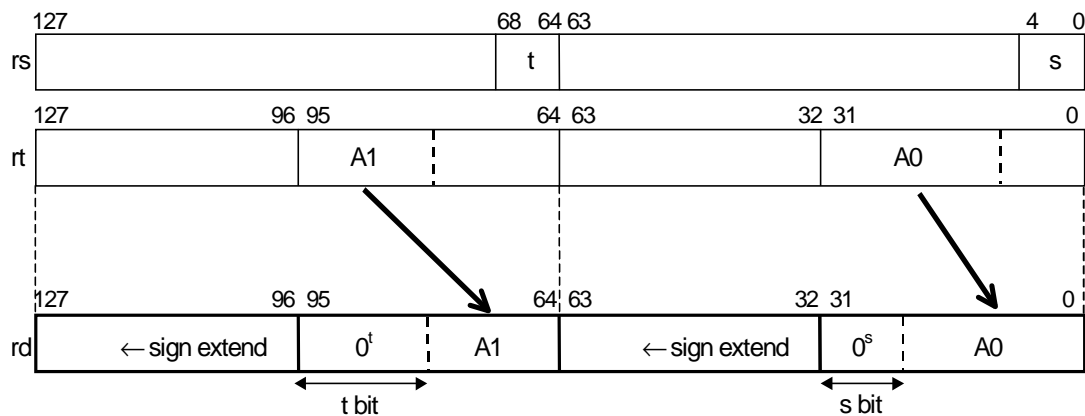
Splits $GPR[rt]$ into two doublewords and shifts each of the low-order words right. The shift amount is specified by the low-order five bits of the corresponding word in $GPR[rs]$. Inserts zeros into the emptied bits. The resulting two 32-bit values are sign-extended and stored in $GPR[rd]$.

Exceptions

None

Operation

$s \leftarrow GPR[rs]_{4..0}$
 $t \leftarrow GPR[rs]_{68..64}$
 $temp0 \leftarrow 0^s \parallel GPR[rt]_{31..s}$
 $temp1 \leftarrow 0^t \parallel GPR[rt]_{95..64+t}$
 $GPR[rd]_{63..0} \leftarrow (temp0_{31})^{32} \parallel temp0_{31..0}$
 $GPR[rd]_{127..64} \leftarrow (temp1_{31})^{32} \parallel temp1_{31..0}$



PSRLW : Parallel Shift Right Logical Word

128-bit MMI

To arithmetically shift right 4 words in 128-bit data. The shift amount is a fixed value specified by sa.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	0	rt	rd	sa	PSRLW						
011100	00000										111110
6	5	5	5	5	6						

Format

PSRLW rd, rt, sa

Description

$GPR[rd] \leftarrow GPR[rt] \gg sa$ (logical)

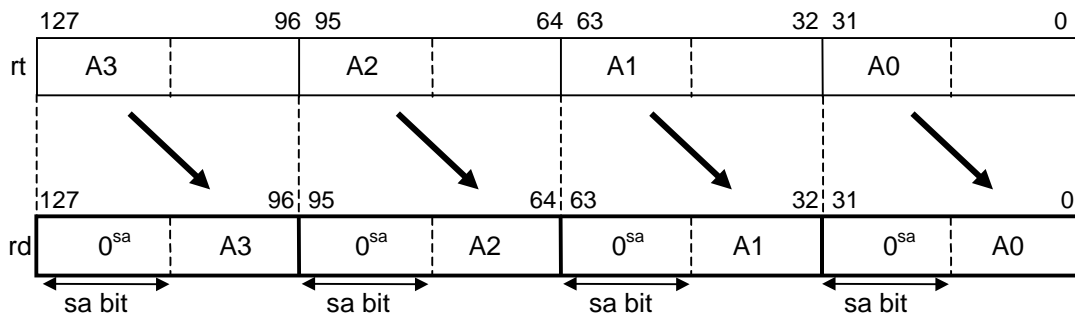
Splits $GPR[rt]$ into four words, shifts them right by the number of bits specified by sa and inserts zeros into the emptied bits. The resulting value is stored in the corresponding words in $GPR[rd]$.

Exceptions

None

Operation

$GPR[rd]_{31..0} \leftarrow 0^{sa} \parallel GPR[rt]_{31..sa}$
 $GPR[rd]_{63..32} \leftarrow 0^{sa} \parallel GPR[rt]_{63..32+sa}$
 $GPR[rd]_{95..64} \leftarrow 0^{sa} \parallel GPR[rt]_{95..64+sa}$
 $GPR[rd]_{127..96} \leftarrow 0^{sa} \parallel GPR[rt]_{127..96+sa}$



PSUBB : Parallel Subtract Byte

128-bit MMI

To subtract 16 pairs of 8-bit integers in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PSUBB	MMI0						
011100				01001	001000						
6	5	5	5	5	6						

Format

PSUBB rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] - GPR[rt]$

Splits $GPR[rs]$ and $GPR[rt]$ into sixteen 8-bit integers, subtracts the data in $GPR[rt]$ from the corresponding data in $GPR[rs]$ and stores them in the corresponding bytes in $GPR[rd]$. If an overflow or underflow occurs, the results are just truncated.

Exceptions

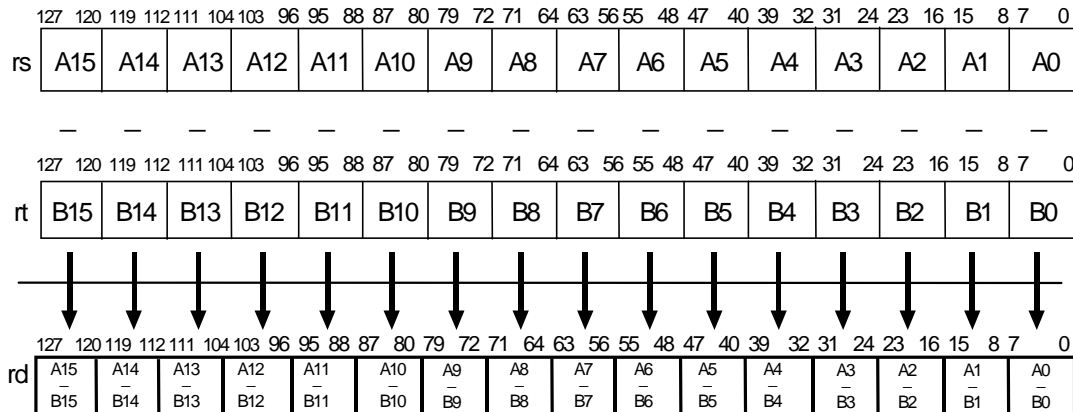
None. Even when the result of the arithmetic operation overflows or underflows, an exception does not occur.

Operation

$GPR[rd]_{7..0} \leftarrow (GPR[rs]_{7..0} - GPR[rt]_{7..0})_{7..0}$

$GPR[rd]_{15..8} \leftarrow (GPR[rs]_{15..8} - GPR[rt]_{15..8})_{7..0}$

(The same operations follow every 8 bits)



PSUBH : Parallel Subtract Halfword

128-bit MMI

To subtract 8 pairs of 16-bit integers in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PSUBH	MMI0						
011100				00101	001000						
6	5	5	5	5	6						

Format

PSUBH rd, rs, rt

Description

$$\text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rs}] - \text{GPR}[\text{rt}]$$

Splits the 128-bit value in GPR[rs] and GPR[rt] into eight 16-bit integers, subtracts the data in GPR[rt] from the corresponding data in GPR[rs] and stores them in the corresponding halfwords in GPR[rd]. If an overflow or underflow occurs, the results are just truncated.

Exceptions

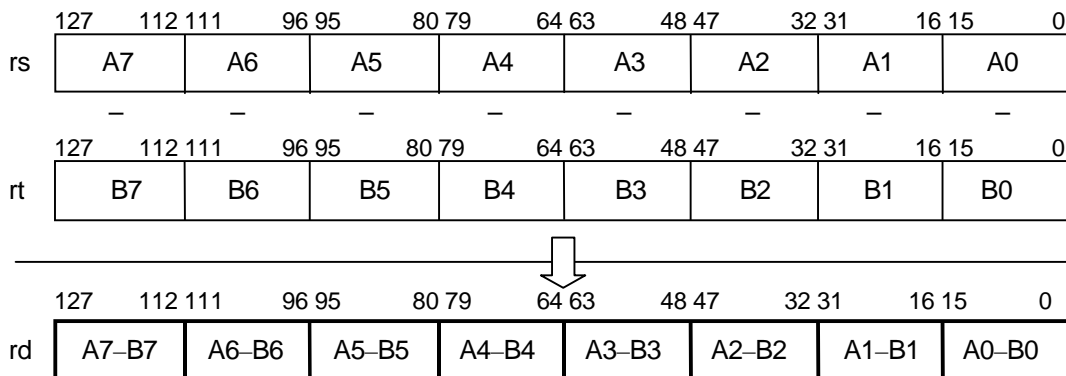
None

Operation

$$\text{GPR}[\text{rd}]_{15..0} \leftarrow (\text{GPR}[\text{rs}]_{15..0} - \text{GPR}[\text{rt}]_{15..0})_{15..0}$$

$$\text{GPR}[\text{rd}]_{31..16} \leftarrow (\text{GPR}[\text{rs}]_{31..16} - \text{GPR}[\text{rt}]_{31..16})_{15..0}$$

(The same operations follow every 16 bits)

$$\text{GPR}[\text{rd}]_{127..112} \leftarrow (\text{GPR}[\text{rs}]_{127..112} - \text{GPR}[\text{rt}]_{127..112})_{15..0}$$


PSUBSB : Parallel Subtract with Signed saturation Byte

128-bit MMI

To subtract 16 pairs of 8-bit signed integers with saturation in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI		rs		rt		rd		PSUBSB		MMI0	
011100								11001		001000	
6		5		5		5		5		6	

Format

PSUBSB rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] - GPR[rt]$

Splits $GPR[rs]$ and $GPR[rt]$ into sixteen 8-bit signed integers, subtracts the data in $GPR[rt]$ from the corresponding data in $GPR[rs]$ with saturation and stores them in the corresponding bytes in $GPR[rd]$.

Exceptions

None

Operation

```

if ((GPR[rs]7..0 - GPR[rt]7..0) >= 0x7F) then
  GPR[rd]7..0 ← 0x7F
else if (0x100 <= (GPR[rs]7..0 - GPR[rt]7..0) < 0x180) then
  GPR[rd]7..0 ← 0x80
else
  GPR[rd]7..0 ← (GPR[rs]7..0 - GPR[rt]7..0)7..0
endif

```

```

if ((GPR[rs]15..8 - GPR[rt]15..8) >= 0x7F) then
  GPR[rd]15..8 ← 0x7F
else if (0x100 <= (GPR[rs]15..8 - GPR[rt]15..8) < 0x180) then
  GPR[rd]15..8 ← 0x80
else
  GPR[rd]15..8 ← (GPR[rs]15..8 - GPR[rt]15..8)7..0
endif

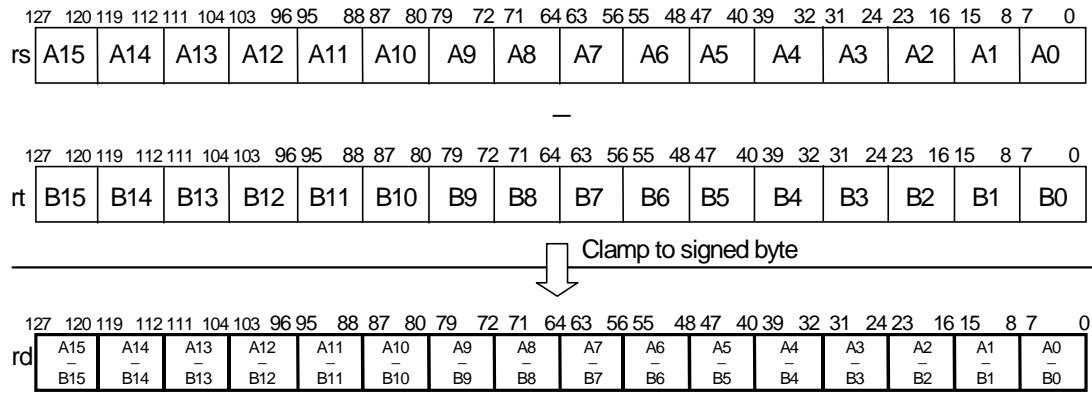
```

(The same operations follow every 8 bits)

```

if ((GPR[rs]127..120 - GPR[rt]127..120) >= 0x7F) then
  GPR[rd]127..120 ← 0x7F
else if (0x100 <= (GPR[rs]127..120 - GPR[rt]127..120) < 0x180) then
  GPR[rd]127..120 ← 0x80
else
  GPR[rd]127..120 ← (GPR[rs]127..120 - GPR[rt]127..120)7..0
endif

```



PSUBSH : Parallel Subtract with Signed Saturation Halfword

128-bit MMI

To subtract 8 pairs of 16-bit integers in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PSUBSH	MMI0						
011100				10101	001000						
6	5	5	5	5	6						

Format

PSUBSH rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] - GPR[rt]$

Splits $GPR[rs]$ and $GPR[rt]$ into eight 16-bit signed integers, subtracts the data in $GPR[rt]$ from the corresponding data in $GPR[rs]$ and stores them in the corresponding halfwords in $GPR[rd]$. If an overflow or underflow occurs, the results are just truncated.

Exceptions

None

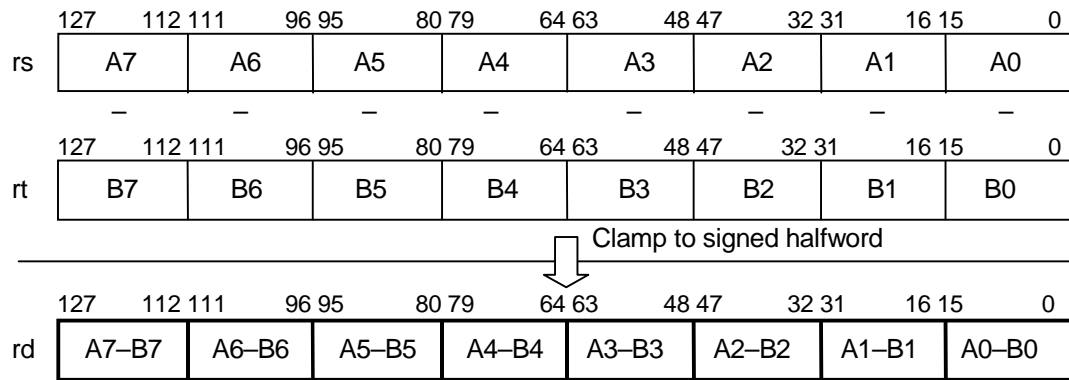
Operation

```
if ((GPR[rs]15..0 - GPR[rt]15..0) >= 0x7FFF) then
  GPR[rd]15..0 ← 0x7FFF
else if (0x10000 <= (GPR[rs]15..0 - GPR[rt]15..0) < 0x18000) then
  GPR[rd]15..0 ← 0x8000
else
  GPR[rd]15..0 ← (GPR[rs]15..0 - GPR[rt]15..0)15..0
endif
```

```
if ((GPR[rs]31..16 - GPR[rt]31..16) >= 0x7FFF) then
  GPR[rd]31..16 ← 0x7FFF
else if (0x10000 <= (GPR[rs]31..16 - GPR[rt]31..16) < 0x18000) then
  GPR[rd]31..16 ← 0x8000
else
  GPR[rd]31..16 ← (GPR[rs]31..16 - GPR[rt]31..16)15..0
endif
```

(The same operations follow every 16 bits)

```
if ((GPR[rs]127..112 - GPR[rt]127..112) >= 0x7FFF) then
  GPR[rd]127..112 ← 0x7FFF
else if (0x10000 <= (GPR[rs]127..112 - GPR[rt]127..112) < 0x18000) then
  GPR[rd]127..112 ← 0x8000
else
  GPR[rd]127..112 ← (GPR[rs]127..112 - GPR[rt]127..112)15..0
endif
```

PSUBSW : Parallel Subtract with Signed Saturation Word

128-bit MMI

To subtract 4 pairs of 32-bit signed integers with saturation in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PSUBSW	MMIO						
011100				10001	001000						
6	5	5	5	5	6						

Format

PSUBSW rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] - GPR[rt]$

Splits $GPR[rs]$ and $GPR[rt]$ into four 32-bit signed integers, subtracts the data in $GPR[rt]$ from the corresponding data in $GPR[rs]$ with saturation and stores them in the corresponding words in $GPR[rd]$.

Exceptions

None

Operation

```

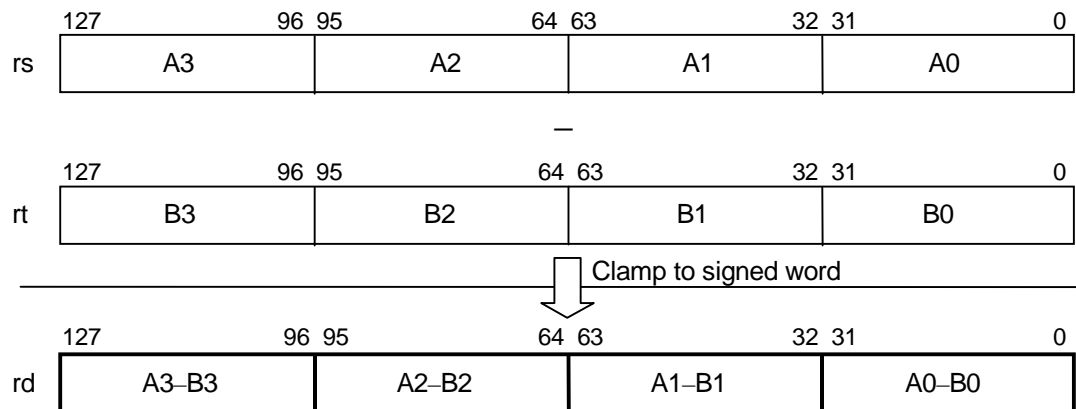
if ((GPR[rs]31..0 - GPR[rt]31..0) >= 0x7FFFFFFF) then
    GPR[rd]31..0 ← 0x7FFFFFFF
else if (0x100000000 <= (GPR[rs]31..0 - GPR[rt]31..0) < 0x180000000) then
    GPR[rd]31..0 ← 0x80000000
else
    GPR[rd]31..0 ← (GPR[rs]31..0 - GPR[rt]31..0)31..0
endif

if ((GPR[rs]63..32 - GPR[rt]63..32) >= 0x7FFFFFFF) then
    GPR[rd]63..32 ← 0x7FFFFFFF
else if (0x100000000 <= (GPR[rs]63..32 - GPR[rt]63..32) < 0x180000000) then
    GPR[rd]63..32 ← 0x80000000
else
    GPR[rd]63..32 ← (GPR[rs]63..32 - GPR[rt]63..32)31..0
endif

if ((GPR[rs]95..64 - GPR[rt]95..64) >= 0x7FFFFFFF) then
    GPR[rd]95..64 ← 0x7FFFFFFF
else if (0x100000000 <= (GPR[rs]95..64 - GPR[rt]95..64) < 0x180000000) then
    GPR[rd]95..64 ← 0x80000000
else
    GPR[rd]95..64 ← (GPR[rs]95..64 - GPR[rt]95..64)31..0
endif

if ((GPR[rs]127..96 - GPR[rt]127..96) >= 0x7FFFFFFF) then
    GPR[rd]127..96 ← 0x7FFFFFFF
else if (0x100000000 <= (GPR[rs]127..96 - GPR[rt]127..96) < 0x180000000) then
    GPR[rd]127..96 ← 0x80000000
else
    GPR[rd]127..96 ← (GPR[rs]127..96 - GPR[rt]127..96)31..0
endif

```



PSUBUB : Parallel Subtract with Unsigned Saturation Byte

128-bit MMI

To subtract 16 pairs of 8-bit unsigned integers with saturation in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI		rs		rt		rd		PSUBUB		MMI1	
011100								11001		101000	
6		5		5		5		5		6	

Format

PSUBUB rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] - GPR[rt]$

Splits $GPR[rs]$ and $GPR[rt]$ into sixteen unsigned bytes, subtracts the data in $GPR[rt]$ from the corresponding data in $GPR[rs]$ with saturation and stores them in the corresponding bytes in $GPR[rd]$.

Exceptions

None

Operation

if $((GPR[rs]_{7..0} - GPR[rt]_{7..0}) \leq 0x00)$ then

$GPR[rd]_{7..0} \leftarrow 0x00$

else

$GPR[rd]_{7..0} \leftarrow (GPR[rs]_{7..0} - GPR[rt]_{7..0})_{7..0}$

endif

if $((GPR[rs]_{15..8} - GPR[rt]_{15..8}) \leq 0x00)$ then

$GPR[rd]_{15..8} \leftarrow 0x00$

else

$GPR[rd]_{15..8} \leftarrow (GPR[rs]_{15..8} - GPR[rt]_{15..8})_{7..0}$

endif

(The same operations follow every 8 bits)

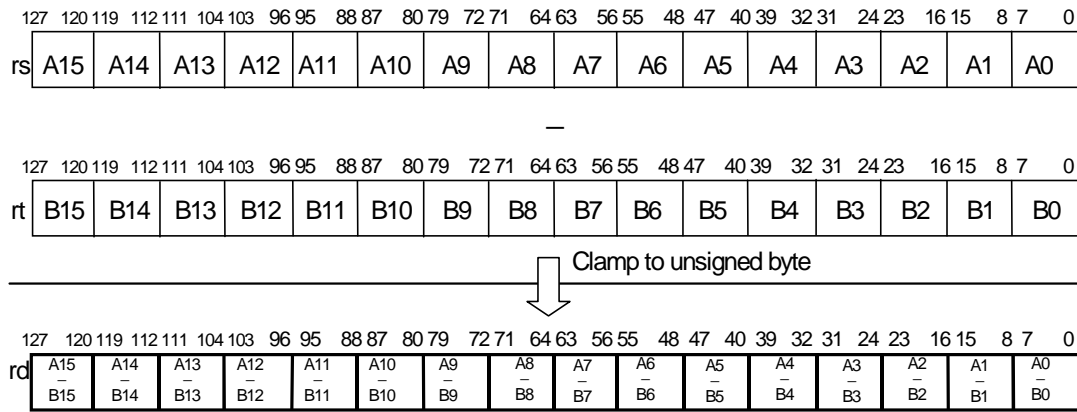
if $((GPR[rs]_{127..120} - GPR[rt]_{127..120}) \leq 0x00)$ then

$GPR[rd]_{127..120} \leftarrow 0x00$

else

$GPR[rd]_{127..120} \leftarrow (GPR[rs]_{127..120} - GPR[rt]_{127..120})_{7..0}$

endif



PSUBUH : Parallel Subtract with Unsigned Saturation Halfword

128-bit MMI

To subtract 8 pairs of 16-bit unsigned integers with saturation in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI		rs		rt		rd		PSUBUH		MMI1	
011100								10101		101000	
6		5		5		5		5		6	

Format

PSUBUH rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] - GPR[rt]$

Splits GPR[rs] and GPR[rt] into eight unsigned halfword data, subtracts the data in GPR[rt] from the corresponding data in GPR[rs] with saturation and stores them in the corresponding halfwords in GPR[rd].

Exceptions

None

Operation

if $((GPR[rs]_{15..0} - GPR[rt]_{15..0}) \leq 0x0000)$ then

$GPR[rd]_{15..0} \leftarrow 0x0000$

else

$GPR[rd]_{15..0} \leftarrow (GPR[rs]_{15..0} - GPR[rt]_{15..0})_{15..0}$

endif

if $((GPR[rs]_{31..16} - GPR[rt]_{31..16}) \leq 0x0000)$ then

$GPR[rd]_{31..16} \leftarrow 0x0000$

else

$GPR[rd]_{31..16} \leftarrow (GPR[rs]_{31..16} - GPR[rt]_{31..16})_{15..0}$

endif

(The same operations follow every 16 bits)

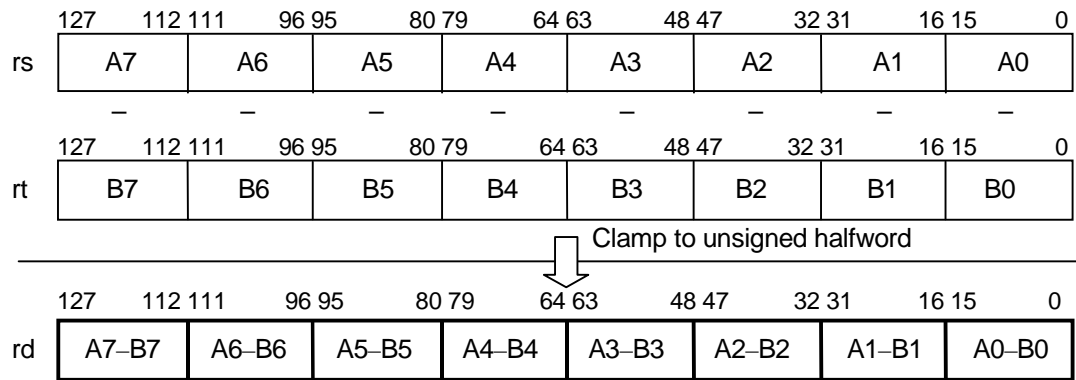
if $((GPR[rs]_{127..112} - GPR[rt]_{127..112}) \leq 0x0000)$ then

$GPR[rd]_{127..112} \leftarrow 0x0000$

else

$GPR[rd]_{127..112} \leftarrow (GPR[rs]_{127..112} - GPR[rt]_{127..112})_{15..0}$

endif



PSUBUW : Parallel Subtract with Unsigned Saturation Word

128-bit MMI

To subtract 4 pairs of 32-bit unsigned integers with saturation in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI		rs		rt		rd		PSUBUW		MMI1	
011100								10001		101000	
6		5		5		5		5		6	

Format

PSUBUW rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] - GPR[rt]$

Splits GPR[rs] and GPR[rt] into four unsigned words, subtracts the data in GPR[rt] from the corresponding data in GPR[rs] with saturation and stores them in the corresponding words in GPR[rd].

Exceptions

None

Operation

if $((GPR[rs]_{31..0} - GPR[rt]_{31..0}) \leq 0x00000000)$ then

$GPR[rd]_{31..0} \leftarrow 0x00000000$

else

$GPR[rd]_{31..0} \leftarrow (GPR[rs]_{31..0} - GPR[rt]_{31..0})_{31..0}$

endif

if $((GPR[rs]_{63..32} - GPR[rt]_{63..32}) \leq 0x00000000)$ then

$GPR[rd]_{63..32} \leftarrow 0x00000000$

else

$GPR[rd]_{63..32} \leftarrow (GPR[rs]_{63..32} - GPR[rt]_{63..32})_{31..0}$

endif

if $((GPR[rs]_{95..64} - GPR[rt]_{95..64}) \leq 0x00000000)$ then

$GPR[rd]_{95..64} \leftarrow 0x00000000$

else

$GPR[rd]_{95..64} \leftarrow (GPR[rs]_{95..64} - GPR[rt]_{95..64})_{31..0}$

endif

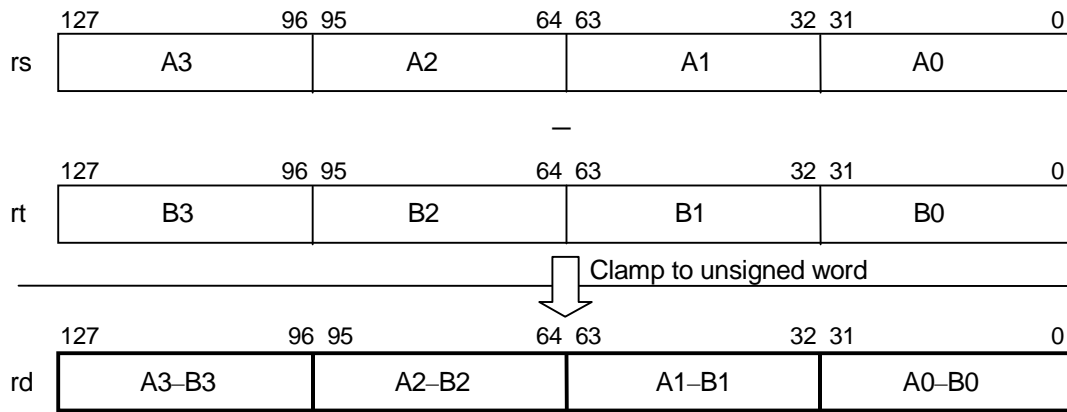
if $((GPR[rs]_{127..96} - GPR[rt]_{127..96}) \leq 0x00000000)$ then

$GPR[rd]_{127..96} \leftarrow 0x00000000$

else

$GPR[rd]_{127..96} \leftarrow (GPR[rs]_{127..96} - GPR[rt]_{127..96})_{31..0}$

endif



PSUBW : Parallel Subtract Word

128-bit MMI

To subtract 4 pairs of 32-bit integers in parallel.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI	rs	rt	rd	PSUBW	MMI0						
011100				00001	001000						
6	5	5	5	5	6						

Format

PSUBW rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] - GPR[rt]$

Splits $GPR[rs]$ and $GPR[rt]$ into four 32-bit signed integers, subtracts the data in $GPR[rt]$ from the corresponding data in $GPR[rs]$ and stores them in the corresponding words in $GPR[rd]$.

If an overflow or underflow occurs, the results are just truncated.

Exceptions

None

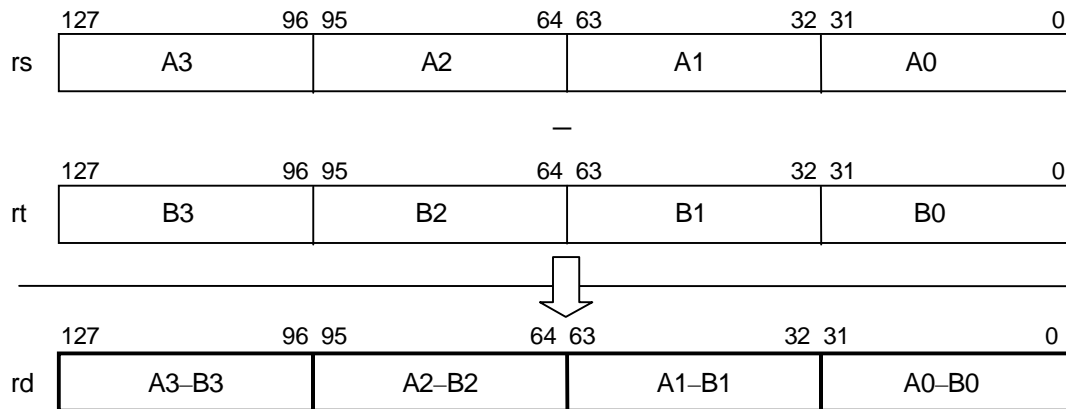
Operation

$GPR[rd]_{31..0} \leftarrow (GPR[rs]_{31..0} - GPR[rt]_{31..0})_{31..0}$

$GPR[rd]_{63..32} \leftarrow (GPR[rs]_{63..32} - GPR[rt]_{63..32})_{31..0}$

$GPR[rd]_{95..64} \leftarrow (GPR[rs]_{95..64} - GPR[rt]_{95..64})_{31..0}$

$GPR[rd]_{127..96} \leftarrow (GPR[rs]_{127..96} - GPR[rt]_{127..96})_{31..0}$



PXOR : Parallel Exclusive OR

128-bit MMI

To calculate a bitwise logical EXCLUSIVE OR.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0	
MMI 011100			rs		rt		rd		PXOR 10011		MMI2 001001	
6			5		5		5		5		6	

Format

PXOR rd, rs, rt

Description

$GPR[rd] \leftarrow GPR[rs] \text{ XOR } GPR[rt]$

Calculates a 128-bit bitwise logical XOR between GPR[rs] and GPR[rt]. The result is stored in GPR[rd].

The truth table values for XOR are as follows;

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Exceptions

None

Operation

$GPR[rd]_{127..0} \leftarrow GPR[rs]_{127..0} \text{ XOR } GPR[rt]_{127..0}$

QFSRV : Quadword Funnel Shift Right Variable

128-bit MMI

To right shift a 128-bit value. The shift amount is specified by the SA register.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
MMI		rs		rt		rd		QFSRV		MMI1	
011100								11011		101000	
6		5		5		5		5		6	

Format

QFSRV rd, rs, rt

Description

$GPR[rd] \leftarrow (GPR[rs], GPR[rt]) \gg SA$

Shifts right the 256-bit concatenation of GPR[rs] with GPR[rt] by the number of bits specified by the SA register, and stores the low-order 128 bits of the result in GPR[rd].

Since the value of the SA register is set using the MTSAB or MTSABH instruction, the shift amount with the QFSRV instruction must be a multiple of bytes or halfwords.

Exceptions

None

Operation

if (SA = 0) then

$GPR[rd]_{127..0} \leftarrow GPR[rt]_{127..0}$

else

$GPR[rd]_{127..0} \leftarrow GPR[rs]_{SA-1..0} \parallel GPR[rt]_{127..SA}$

endif

Programming Notes

A left funnel shift is made possible by the value set in the SA register. To left shift s bytes and s halfwords, specify 16-s in the MTSAB instruction and 8-s in the MTSABH instruction respectively (0<s<16). A quick way to perform this computation is as follows;

// Register %sal contains the left shift amount

subi %samt, %sal, 1

mtsab %samt, -1

// The following QFSRV does a shift left by %sal bytes

qfsrv %dst, %src1, %src2

The instruction can be used to rotate 128-bit data by specifying the same register in rs and rt. For example, the following code rotates right the contents of GPR[5] by three halfwords (=48-bit) and places the result in GPR[6].

mtsah r0, 3

qfsrv r6, r5, r5

SQ : Store Quadword

128-bit MMI

To store 128-bit data in memory.

Operation Code

31	26	25	21	20	16	15	0
SQ						base	
011111						rt	
						offset	
6						5	
						5	
						16	

Format

SQ rt, offset (base)

Description

memory [GPR[base] + offset] \leftarrow GPR[rt]

Adds the offset as a 16-bit number to the value in GPR[base] to form the effective address. Stores the 128-bit data in GPR[rt] at the address.

The least-significant four bits of the effective address are masked to zero when accessing memory.

Therefore, the effective address does not have to conform to the natural alignment.

Exceptions

TLB Refill, TLB Invalid, Address Error (excluding Address Error due to alignment)

Operation

vAddr \leftarrow sign_extend (offset) + GPR[base]

vAddr_{3..0} = 0⁴

(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA, STORE)

quadword \leftarrow GPR[rt]_{127..0}

(This page is left blank intentionally)

4. System Control Coprocessor (COP0) Instruction Set

This chapter describes the system control coprocessor (COP0) instructions.

COP0 instructions can operate on the system control coprocessor register to perform memory control, cache control, exception handling, and breakpoint operation for the EE Core. A COP0 instruction is valid when either the EE Core is in kernel mode or bit 28 (CU[0]) of the status register is 1. If a COP0 instruction is executed in other cases, a Coprocessor Unusable exception occurs.

BC0F : Branch on Coprocessor 0 False

MIPS I

To branch according to the coprocessor 0's condition signal.

Operation Code

31	26	25	21	20	16	15	0
COP0						BC0	
010000						01000	
						BC0F	
						00000	
						offset	
6						5	
						5	
						16	

Format

BC0F offset

Description

Checks the CPCOND0 signal (coprocessor 0's condition signal) when the BC0F instruction is fetched. If the CPCOND0 signal is false, executes the subsequent instruction (branch delay slot instruction), then the program branches to the target address, which is the offset shifted left by 2 bits and added to the address of the instruction in the delay slot.

Exceptions

Coprocessor unusable.

Operation

I: $\text{target} \leftarrow (\text{offset}_{15})^{46} \mid \mid \text{offset} \mid \mid 0^2$
 $\text{condition} \leftarrow \text{NOT CPCOND0}$

I+1: if condition then
 $\text{PC} \leftarrow \text{PC} + \text{target}$
 endif

Programming Notes

Coprocessor 0's condition signal is a DMA transfer termination signal. Since it is an external signal, it varies asynchronous to the execution of the instruction.

BC0FL : Branch on Coprocessor 0 False Likely

MIPS I

To branch according to the coprocessor 0's condition signal.

Operation Code

31	26	25	21	20	16	15	0
COP0	BC0	BC0FL	offset				
010000	01000	00010					
6	5	5	16				

Format

BC0FL offset

Description

Checks the CPCOND0 signal (coprocessor 0's condition signal) when the BC0FL instruction is fetched. If the CPCOND0 signal is false, executes the subsequent instruction (branch delay slot instruction), then the program branches to the target address, which is the offset shifted left by 2 bits and added to the address of the instruction in the delay slot.

If the CPCOND0 signal is true, nullifies the branch delay slot instruction.

Exceptions

Coprocessor unusable

Operation

```

I:      target ← (offset15)46 || offset || 02
        condition ← NOT CPCOND0
I+1:    if condition then
            PC ← PC + target
        else
            NullifyCurrentInstruction()
        endif

```

Programming Notes

The coprocessor 0's condition signal is a DMA transfer termination signal. Since it is an external signal, it varies asynchronous to the execution of the instruction.

BC0T : Branch on Coprocessor 0 True

MIPS I

To branch according to the coprocessor 0's condition signal.

Operation Code

31	26	25	21	20	16	15	0
COP0	BC0	BC0T	offset				
010000	01000	00001					
6	5	5	16				

Format

BC0T offset

Description

Checks the CPCOND0 signal (coprocessor 0's condition signal) when the BC0T instruction is fetched. If the CPCOND0 signal is true, executes the subsequent instruction (branch delay slot instruction), then the program branches to the target address, which is the offset shifted left by 2 bits and added to the address of the instruction in the delay slot.

Exceptions

Coprocessor unusable

Operation

```

I:      target ← (offset15)46 || offset || 02
        condition ← CPCOND0
I+1:    if condition then
            PC ← PC + target
        endif

```

Programming Notes

The coprocessor 0's condition signal is a DMA transfer termination signal. Since it is an external signal, it varies asynchronous to the execution of the instruction.

BC0TL : Branch on Coprocessor 0 True Likely

MIPS I

To branch according to the coprocessor 0's condition signal.

Operation Code

31	26	25	21	20	16	15	0
COP0						BC0	
010000						01000	
						BC0TL	
						00011	
						offset	
6						5	
						5	
						16	

Format

BC0TL offset

Description

Checks the CPCODE0 signal (coprocessor 0's condition signal) when the BC0TL instruction is fetched. If the CPCODE0 signal is true, executes the subsequent instruction (branch delay slot instruction), then the program branches to the target address, which is the offset shifted left by 2 bits and added to the address of the instruction in the delay slot. If the CPCODE0 signal is false, nullifies the branch delay slot instruction.

Exceptions

Coprocessor unusable

Operation

I: $\text{target} \leftarrow (\text{offset}_{15})^{46} \mid \mid \text{offset} \mid \mid 0^2$
 $\text{condition} \leftarrow \text{CPCODE0}$

I+1: if condition then
 $\text{PC} \leftarrow \text{PC} + \text{target}$
 else
 NullifyCurrentInstruction()
 endif

Programming Notes

The coprocessor 0's condition signal is a DMA transfer termination signal. Since it is an external signal, it varies asynchronous to the execution of the instruction.

CACHE BFH : Cache Operation (BTAC Flush)

MIPS III

To flush BTAC.

Operation Code

31	26	25	21	20	16	15	0
CACHE	base		op				offset
101111			01100				
6	5		5				16

Format

CACHE 0x0c, offset(base)

Description

Invalidates all BTAC (Branch Target Address Cache) entries.

The offset and base values are meaningless.

Restrictions

A sequence of CACHE instructions including CACHE BFH has to be directly preceded and followed by a SYNC.P instruction.

Exceptions

Coprocessor unusable.

Operation

$vAddr \leftarrow (offset_{15})^{16} \mid offset_{15..0} + GPR[base]_{31..0}$
 $(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$
 CacheOp (op, vAddr, pAddr)

Programming Notes

This instruction operates regardless of the BPE bit value of the CONFIG register.

CACHE BHINBT : Cache Operation (Hit Invalidate BTAC)

MIPS III

To partially invalidate BTAC.

Operation Code

31	26	25	21	20	16	15	0
CACHE 101111		base		op 01010		offset	
6		5		5		16	

Format

CACHE 0x0a, offset(base)

Description

Obtains a virtual address (vAddr) from the GPR[base] value and the offset, and invalidates BTAC (Branch Target Address Cache) entries which match vAddr[31:3].

Restriction

The operation is undefined if there are plural of BTAC entries that match the specified address.

The operation of this instruction is also undefined when specifying uncached, uncached accelerated and SPRAM addresses.

A sequence of CACHE instructions including CACHE BHINBT has to be directly preceded and followed by a SYNC.P instruction.

Exceptions

Coprocessor unusable.

Operation

$vAddr \leftarrow (offset_{15})^{16} \mid offset_{15..0} + GPR[base]_{31..0}$
 $(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$
 CacheOp (op, vAddr, pAddr)

Programming Notes

This instruction operates regardless of the BPE bit value of the CONFIG register.

CACHE BXLBT : Cache Operation (Index Load BTAC)

MIPS III

To read BTAC (Branch Target Address Cache) entries.

Operation Code

31	26	25	21	20	16	15	0
CACHE 101111						base	op 00010
						offset	
6						5	5
						16	

Format

CACHE 0x02, offset(base)

Description

Obtains a virtual address (vAddr) from the GPR[base] value and the offset, reads the BTAC entry indicated by vAddr[5:0], and stores it in the COP0 TagHI/TagLO registers.

Restrictions

A sequence of CACHE instructions including CACHE BXLBT has to be directly preceded and followed by a SYNC.P instruction.

Exceptions

Coprocessor unusable.

Operation

$vAddr \leftarrow (offset_{15})^{16} \mid offset_{15..0} + GPR[base]_{31..0}$
 $(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$
 TagLO[0] = Valid Bit
 TagLO[31:3] = FetchAddress[28:0]
 TagHI[31:2] = TargetAddress[29:0]

Programming Notes

This instruction operates regardless of the BPE bit value of the CONFIG register.

CACHE BXSBT : Cache Operation (Index Store BTAC)

MIPS III

To write to BTAC (Branch Target Address Cache) entries.

Operation Code

31	26	25	21	20	16	15	0
CACHE	base	op	offset				
101111		00110					
6	5	5	16				

Format

CACHE 0x06, offset(base)

Description

Obtains a virtual address (vAddr) from the GPR[base] value and the offset and stores the contents of the COP0 TagHI/TagLO registers in the BTAC entry indicated by vAddr[5:0].

Restrictions

A sequence of CACHE instructions including CACHE BXSBT has to be directly preceded and followed by a SYNC.P instruction.

Exceptions

Coprocessor unusable.

Operation

$vAddr \leftarrow (offset_{15})^{16} \mid offset_{15..0} + GPR[base]_{31..0}$
 $(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$
 Valid Bit = TagLO[0]
 FetchAddress[28:0] = TagLO[31:3]
 TargetAddress[29:0] = TagHI[31:2]

Programming Notes

This instruction operates regardless of the BPE bit value of the CONFIG register.

CACHE DHIN : Cache Operation (Hit Invalidate)

MIPS III

To invalidate data cache entries.

Operation Code

31	26	25	21	20	16	15	0
CACHE 101111	base				op 11010	offset	
6	5				5	16	

Format

CACHE 0x1a, offset(base)

Description

CACHE DHIN obtains a virtual address (vAddr) and a physical address (pAddr) from the GPR[base] value and the offset, and invalidates the corresponding entry in the data cache if there is a hit.

That is, when the tags at vAddr[11:6] are read from both ways of the data cache, if the Valid bit is 1 and the PFN of the tag matches the pAddr, CACHE DHIN changes the Lock bit, Valid bit, and Dirty bit to 0. The LRF bit is not changed.

Restrictions

The operation is undefined when specifying uncached, uncached accelerated and SPRAM addresses.

A sequence of CACHE instructions including CACHE DHIN has to be directly preceded by a SYNC.L instruction.

Exceptions

Coprocessor unusable, TLB Refill, TLB Invalid, Address Error

Operation

$vAddr \leftarrow (offset_{15})^{16} \mid offset_{15..0} + GPR[base]_{31..0}$
 $(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$
 CacheOp (op, vAddr, pAddr)

Programming Notes

This instruction operates regardless of the DCE bit value of the CONFIG register.

CACHE DHWBIN : Cache Operation (Hit Writeback Invalidate)

MIPS III

To write data cache entries back to memory and invalidate them.

Operation Code

31	26	25	21	20	16	15	0
CACHE		base		op		offset	
101111				11000			
6		5		5		16	

Format

CACHE 0x18, offset(base)

Description

CACHE DHWBIN obtains a virtual address (vAddr) and a physical address (pAddr) from the GPR[base] value and the offset, invalidates the corresponding entry in the data cache if there is a hit, and writes dirty data back to memory.

That is, when the tags at vAddr[11:6] are read from both ways of the data cache, if the Valid bit is 1 and the PFN of the tag matches the pAddr, CACHE DHWBIN changes the Lock bit, Valid bit, and Dirty bit to 0. (The LRF bit is not changed.) If the Dirty bit is 1, this instruction writes the cache line data to the physical address obtained from vAddr[11:6] and PFN.

Restrictions

The operation is undefined when specifying uncached, uncached accelerated and SPRAM addresses.

CACHE DHWBIN has to be directly followed by a SYNC.L instruction. A sequence of CACHE instructions including CACHE DHWBIN has to be directly preceded by a SYNC.L instruction.

Exceptions

Coprocessor unusable, TLB Refill, TLB Invalid, Address Error

Operation

$$vAddr \leftarrow (offset_{15})^{16} \mid offset_{15..0} + GPR[base]_{31..0}$$

$$(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$$

$$CacheOp(op, vAddr, pAddr)$$

Programming Notes

This instruction operates regardless of the DCE bit value of the CONFIG register.

CACHE DHWOIN : Cache Operation (Hit Writeback Without Invalidate)

MIPS III

To write data cache entries back to memory.

Operation Code

31	26	25	21	20	16	15	0
CACHE 101111	base					op 11100	offset
6	5					5	16

Format

CACHE 0x1c, offset(base)

Description

CACHE DHWOIN obtains a virtual address (vAddr) and a physical address (pAddr) from the GPR[base] value and the offset, and writes dirty data back to memory if it is contained in the hit entry.

That is, when the tags at vAddr[11:6] are read from both ways of the data cache, if the Valid bit and Dirty bit are 1 and the PFN of the tag matches the pAddr, CACHE DHWOIN writes the cache line data to the physical address obtained from vAddr[11:6] and PFN, and changes the Dirty bit to 0.

Restrictions

The operation is undefined when specifying uncached, uncached accelerated and SPRAM addresses.

CACHE DHWOIN has to be directly followed by a SYNC.L instruction. A sequence of CACHE instructions including CACHE DHWOIN has to be directly preceded by a SYNC.L instruction.

Exceptions

Coprocessor unusable, TLB Refill, TLB Invalid, Address Error

Operation

$vAddr \leftarrow (offset_{15})^{16} \mid offset_{15..0} + GPR[base]_{31..0}$
 $(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$
 CacheOp (op, vAddr, pAddr)

Programming Notes

This instruction operates regardless of the DCE bit value of the CONFIG register.

CACHE DXIN : Cache Operation (Index Invalidate)

MIPS III

To invalidate specified data cache entries.

Operation Code

31	26	25	21	20	16	15	0
CACHE 101111	base				op 10110	offset	
6	5				5	16	

Format

CACHE 0x16, offset(base)

Description

CACHE DXIN obtains a virtual address (vAddr) from the GPR[base] value and the offset, and invalidates the corresponding data cache line. vAddr[11:6] shows the index of the line and vAddr[0] shows the way. The Lock bit, Valid bit, and Dirty bit are changed to 0. The LRF bit is not changed.

Restrictions

A sequence of CACHE instructions including CACHE DXIN has to be directly preceded by a SYNC.L instruction.

Exceptions

Coprocessor unusable.

Operation

$vAddr \leftarrow (offset_{15})^{16} \mid offset_{15,0} + GPR[base]_{31,0}$
 $(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$
 CacheOp (op, vAddr, pAddr)

Programming Notes

This instruction operates regardless of the DCE bit value of the CONFIG register.

CACHE DXLDT : Cache Operation (Index Load Data)

MIPS III

To read specified data cache entries.

Operation Code

31	26	25	21	20	16	15	0
CACHE 101111	base				op 10001	offset	
6	5				5	16	

Format

CACHE 0x11, offset(base)

Description

CACHE DXLDT obtains a virtual address (vAddr) from the GPR[base] value and the offset, reads the 32-bit data corresponding to the address from the data cache, and stores it in the COP0 TagLO register.

vAddr[11:2] shows the index of the tag and vAddr[0] shows the way.

Restrictions

A sequence of CACHE instructions including CACHE DXLDT has to be directly preceded by a SYNC.L instruction.

Exceptions

Coprocessor unusable.

Operation

$$vAddr \leftarrow (offset_{15})^{16} \mid offset_{15..0} + GPR[base]_{31..0}$$

$$(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$$

$$TagLO[31:0] \leftarrow 32bit-data$$

Programming Notes

This instruction operates regardless of the DCE bit value of the CONFIG register.

CACHE DXLTG : Cache Operation (Index Load Tag)

MIPS III

To read specified data cache entries.

Operation Code

31	26	25	21	20	16	15	0
CACHE	base	op	offset				
101111		10000					
6	5	5	16				

Format

CACHE 0x10, offset(base)

Description

CACHE DXLTG obtains a virtual address (vAddr) from the GPR[base] value and the offset, and reads the corresponding data cache tag into the COP0 TagLO register. vAddr[11:6] specifies the index of the tag and vAddr[0] specifies the way.

Restrictions

A sequence of CACHE instructions including CACHE DXLTG has to be directly preceded by a SYNC.L instruction.

Exceptions

Coprocessor unusable.

Operation

$vAddr \leftarrow (offset_{15})^{16} \mid offset_{15,0} + GPR[base]_{31,0}$
 $(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$

tagdata $\leftarrow Tag(vAddr)$
 TagLO[3] $\leftarrow tagdata.Lock$
 TagLO[4] $\leftarrow tagdata.LRF$
 TagLO[5] $\leftarrow tagdata.Valid$
 TagLO[6] $\leftarrow tagdata.Dirty$
 TagLO[31:12] $\leftarrow tagdata.PFN$
 TagLO bits except above are undefined.

Programming Notes

This instruction operates regardless of the DCE bit value of the CONFIG register.

CACHE DXSDT : Cache Operation (Index Store Data)

MIPS III

To write to specified data cache entries.

Operation Code

31	26	25	21	20	16	15	0
CACHE 101111	base				op 10011	offset	
6	5				5	16	

Format

CACHE 0x13, offset(base)

Description

CACHE DXSDT obtains a virtual address (vAddr) from the GPR[base] value and the offset, and stores the contents of the COP0 TagLO register in the corresponding location of the data cache. vAddr[11:2] shows the index of the tag and vAddr[0] shows the way.

Restrictions

A sequence of CACHE instructions including CACHE DXSDT has to be directly preceded by a SYNC.L instruction.

Exceptions

Coprocessor unusable.

Operation

$$vAddr \leftarrow (offset_{15})^{16} \mid \mid offset_{15..0} + GPR[base]_{31..0}$$

$$(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$$

$$32bit-data \leftarrow TagLO[31:0]$$

Programming Notes

This instruction operates regardless of the DCE bit value of the CONFIG register.

CACHE DXSTG : Cache Operation (Index Store Tag)

MIPS III

To write to specified data cache entries.

Operation Code

31	26	25	21	20	16	15	0
CACHE 101111						base	op 10010
6						5	5
						offset	
						16	

Format

CACHE 0x12, offset(base)

Description

CACHE DXSTG obtains a virtual address (vAddr) from the GPR[base] value and the offset, and stores the contents of the COP0 TagLO register in the corresponding data cache tag. vAddr[11:6] specifies the index of the tag and vAddr[0] specifies the way.

Restrictions

A sequence of CACHE instructions including CACHE DXSTG has to be directly preceded by a SYNC.L instruction.

Exceptions

Coprocessor unusable, TLB Refill, TLB Invalid, Address Error

Operation

$$vAddr \leftarrow (offset_{15})^{16} \mid offset_{15,0} + GPR[base]_{31,0}$$

$$(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$$

$$tag \leftarrow Tag(vAddr)$$

$$tag.Lock \leftarrow TagLO[3]$$

$$tag.LRF \leftarrow TagLO[4]$$

$$tag.Valid \leftarrow TagLO[5]$$

$$tag.Dirty \leftarrow TagLO[6] \ \& \ TagLO[6]$$

$$tag.PFN \leftarrow TagLO[31:12]$$

Programming Notes

This instruction operates regardless of the DCE bit value of the CONFIG register.

CACHE DXWBIN : Cache Operation (Index Writeback Invalidate)

MIPS III

To write specified data cache entries back to memory, and invalidate them.

Operation Code

31	26	25	21	20	16	15	0
CACHE 101111	base				op 10100	offset	
6	5				5	16	

Format

CACHE 0x14, offset(base)

Description

CACHE DXWBIN obtains a virtual address (vAddr) and a physical address (pAddr) from the GPR[base] value and the offset, invalidates the corresponding cache line in the data cache and writes back any dirty data to memory. The Valid bit, Lock bit, and Dirty bit are changed to 0, but the LRF bit does not change. vAddr[11:6] specifies the index and vAddr[0] specifies the way of the data cache line to be invalidated.

Restrictions

CACHE DXWBIN has to be directly followed by a SYNC.L instruction. A sequence of CACHE instructions including CACHE DXWBIN has to be directly preceded by a SYNC.L instruction.

Exceptions

Coprocessor unusable.

Operation

$$vAddr \leftarrow (offset_{15})^{16} \mid offset_{15..0} + GPR[base]_{31..0}$$

$$(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$$

$$CacheOp(op, vAddr, pAddr)$$

Programming Notes

This instruction operates regardless of the DCE bit value of the CONFIG register.

CACHE IFL : Cache Operation (Fill)

MIPS III

To read data from memory into specified instruction cache entries.

Operation Code

31	26	25	21	20	16	15	0
CACHE 101111						base	op 01110
						offset	
6						5	5
						16	

Format

CACHE 0x0e, offset(base)

Description

CACHE IFL obtains a virtual address (vAddr) and a physical address (pAddr) from the GPR[base] value and the offset, and reads the memory data (instruction code) indicated by the physical address into the instruction cache. The line is loaded into the cache line addressed by vAddr[12:6] and the way of the cache is defined by the LRF bits. The PFN is loaded into the corresponding instruction cache tag and the Valid bit is set to 1.

Restrictions

The operation is undefined when specifying uncached, uncached accelerated and SPRAM addresses.

CACHE IFL has to be directly followed by a SYNC.P instruction. A sequence of CACHE instructions including CACHE IFL has to be directly preceded by a SYNC.P instruction.

Exceptions

Coprocessor unusable, TLB Refill, TLB Invalid, Address Error

Operation

$vAddr \leftarrow (offset_{15})^{16} \mid offset_{15..0} + GPR[base]_{31..0}$
 $(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$
 CacheOp (op, vAddr, pAddr)

Programming Notes

This instruction operates regardless of the ICE bit value of the CONFIG register.

CACHE IHIN : Cache Operation (Hit Invalidate)

MIPS III

To invalidate instruction cache entries.

Operation Code

31	26	25	21	20	16	15	0
CACHE 101111						base	op 01011
						offset	
6						5	5
						16	

Format

CACHE 0x0b, offset(base)

Description

CACHE IHIN obtains a virtual address (vAddr) and a physical address (pAddr) from the GPR[base] value and the offset, and invalidates an instruction cache line which matches the PA[31:6].

That is, when the tags at vAddr[12:6] are read from both ways of the instruction cache, if the Valid bit of one of the entries is 1 and the PFN of the tag matches the pAddr, CACHE IHIN changes the Valid bit to 0. The LRF bit is not changed.

This instruction also invalidates BTAC entries which match vAddr[31:6].

Restrictions

The operation is undefined when specifying uncached, uncached accelerated and SPRAM addresses.

A sequence of CACHE instructions including CACHE IHIN has to be directly preceded by a SYNC.P instruction.

Exceptions

Coprocessor unusable, TLB Refill, TLB Invalid, Address Error

Operation

$vAddr \leftarrow (offset_{15})^{16} \mid offset_{15..0} + GPR[base]_{31..0}$

$(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$

CacheOp (op, vAddr, pAddr)

Programming Notes

This instruction operates regardless of the ICE bit value of the CONFIG register.

CACHE IXIN : Cache Operation (Index Invalidate)

MIPS III

To invalidate specified instruction cache entries.

Operation Code

31	26	25	21	20	16	15	0
CACHE	base				op	offset	
101111					00111		
6	5				5	16	

Format

CACHE 0x07, offset(base)

Description

CACHE IXIN obtains a virtual address (vAddr) from the GPR[base] value and the offset, and invalidates the corresponding instruction cache line. vAddr[12:6] shows the index of the line and vAddr[0] shows the way.

The Valid bit is changed to 0, but the LRF bit is not changed.

Restrictions

A sequence of CACHE instructions including CACHE IXIN has to be directly preceded by a SYNC.P instruction.

Exceptions

Coprocessor unusable.

Operation

$$vAddr \leftarrow (offset_{15})^{16} \mid offset_{15,0} + GPR[base]_{31,0}$$

$$(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$$

$$CacheOp(op, vAddr, pAddr)$$

Programming Notes

This instruction operates regardless of the ICE bit value of the CONFIG register.

CACHE IXLDT : Cache Operation (Index Load Data)

MIPS III

To read data from specified instruction cache entries.

Operation Code

31	26	25	21	20	16	15	0
CACHE 101111	base				op 00001	offset	
6	5				5	16	

Format

CACHE 0x01, offset(base)

Description

CACHE IXLDT obtains a virtual address (vAddr) from the GPR[base] value and the offset, reads the 32-bit data (instruction code) corresponding to the address from the cache line of the instruction cache, and stores it in the COP0 TagLO and COP0 TagHI registers. vAddr[12:2] specifies the index and vAddr[0] specifies the way of the instruction cache line to be read.

Restrictions

A sequence of CACHE instructions including CACHE IXLDT has to be directly preceded by a SYNC.P instruction.

Exceptions

Coprocessor unusable.

Operation

$$vAddr \leftarrow (offset_{15})^{16} \mid offset_{15..0} + GPR[base]_{31..0}$$

$$(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$$

TagLO[31:0] = 32-bit instruction
 TagHI[3:0] = SteeringBits[3:0]
 TagHI[5:4] = BHT[1:0]
 TagHI bits except above are undefined.

Programming Notes

This instruction operates regardless of the ICE bit value of the CONFIG register.

CACHE IXLTG : Cache Operation (Index Load Tag)

MIPS III

To read tags from specified instruction cache entries.

Operation Code

31	26	25	21	20	16	15	0
CACHE	base	op	offset				
101111		00000					
6	5	5	16				

Format

CACHE 0x00, offset(base)

Description

CACHE IXLTG obtains a virtual address (vAddr) from the GPR[base] value and the offset, and reads the corresponding instruction cache tag into the COP0 TagLO register. vAddr[12:6] specifies the index of the tag and vAddr[0] specifies the way.

Restrictions

A sequence of CACHE instructions including CACHE IXLTG has to be directly preceded by a SYNC.P instruction.

Exceptions

Coprocessor unusable.

Operation

$$vAddr \leftarrow (offset_{15})^{16} \mid offset_{15..0} + GPR[base]_{31..0}$$

$$(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$$

$$tag \leftarrow Tag(vAddr)$$

$$TagLO[4] \leftarrow tag.LRF$$

$$TagLO[5] \leftarrow tag.VALID$$

$$TagLO[31:12] \leftarrow tag[19:0]$$

TagLO bits except above are undefined.

Programming Notes

This instruction operates regardless of the ICE bit value of the CONFIG register.

CACHE IXSDT : Cache Operation (Index Store Data)

MIPS III

To write tags to specified instruction cache entries.

Operation Code

31	26	25	21	20	16	15	0
CACHE 101111	base				op 00101	offset	
6	5				5	16	

Format

CACHE 0x05, offset(base)

Description

CACHE IXSDT obtains a virtual address (vAddr) from the GPR[base] value and the offset, and stores the contents of the COP0 TagLO and TagHI registers in the corresponding cache line in the instruction cache. vAddr[12:2] specifies the index and vAddr[0] specifies the way.

This instruction invalidates all BTAC entries.

Restrictions

A sequence of CACHE instructions including CACHE IXSDT has to be directly preceded by a SYNC.P instruction.

Exceptions

Coprocessor unusable.

Operation

$vAddr \leftarrow (offset_{15})^{16} \mid \mid offset_{15..0} + GPR[base]_{31..0}$
 $(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$

32-bit instruction $\leftarrow TagLO[31:0]$

SteeringBits[3:0] $\leftarrow TagHI[3:0]$

BHT[1:0] $\leftarrow TagHI[5:4]$

Programming Notes

This instruction operates regardless of the ICE bit value of the CONFIG register.

CACHE IXSTG : Cache Operation (Index Store Tag)

MIPS III

To write tags to specified instruction cache entries.

Operation Code

31	26	25	21	20	16	15	0
CACHE		base		op		offset	
101111				00100			
6		5		5		16	

DI : Disable Interrupt

MIPS I

To disable an interrupt.

Operation Code

31	26	25	21	20	6	5	0
COP0	CO			0		DI	
010000	10000			000 0000 0000 0000		111001	
6	5			15		6	

Format

DI

Description

Disables all interrupts except the NMI and SIO.

Exceptions

None

Operation

```
If (Status.EDI=1) || (Status.EXL=1) || (Status.ERL=1) || (Status.KSU=002) then
    Status.EIE ← 0
endif
```

Programming Notes

When the EDI bit in the Status register is 1, the DI instruction operates in all modes. When this bit is 0, the DI instruction operates only in Kernel mode, and works as a NOP in User and Supervisor modes.

EI : Enable Interrupt

MIPS I

To enable an interrupt.

Operation Code

31	26	25	21	20	6	5	0
COP0						CO	
010000						10000	
						0	
						000 0000 0000 0000	
						EI	
						111000	
6						5	
						15	
						6	

Format

EI

Description

Enables all interrupts.

Exceptions

None

Operation

```
If (Status.EDI=1) || (Status.EXL=1) || (Status.ERL=1) || (Status.KSU=002) then
    Status.EIE ← 1
endif
```

Programming Notes

When the EDI bit in the Status register is 1, the EI instruction operates in all modes. When this bit is 0, the EI instruction operates only in Kernel mode, and works as a NOP in User and Supervisor modes.

ERET : Exception Return

MIPS III

To return from an exception handler.

Operation Code

31	26	25	21	20	6	5	0
COP0	CO			0		ERET	
010000	10000			000 0000 0000 0000		011000	
6	5			15		6	

Format

ERET

Description

ERET is an instruction for returning from an interrupt, exception or error trap.

Unlike a branch or jump instruction, ERET does not execute the subsequent instruction. ERET flushes the effective pipelines of the CPU before fetching the instruction from the jump destination. Any pending loads or stores and ongoing multiplications, divisions, product-sum operations, COP1 and COP2 instructions are not flushed.

Restrictions

An ERET instruction must not be placed in a branch delay slot.

Exceptions

Coprocessor unusable

Operation

```

if Status.ERL = 1 then
    PC ← ErrorEPC
    Status.ERL ← 0
else
    PC ← EPC
    Status.EXL ← 0
endif

```

MFBPC : Move from Breakpoint Control Register

MIPS I

To transfer the contents of the breakpoint control register to a general-purpose register.

Operation Code

31	26	25	21	20	16	15	11	10	0
COP0	MF0		rt		11000			0	
010000	00000							000 0000 0000	
6	5		5		5			11	

Format

MFBPC rt

Description

$GPR[rt] \leftarrow COP0\ BPC$

Copies the contents of the breakpoint control register (one of the COP0 debug registers) to GPR[rt].

Exceptions

Coprocessor unusable

Operation

$data \leftarrow CPR[0, \text{Breakpoint Control}]$

$GPR[rt] \leftarrow (data_{31})^{32} \mid data_{31..0}$

MFC0 : Move from System Control Coprocessor

MIPS I

To transfer a COP0 register to a general-purpose register.

Operation Code

31	26	25	21	20	16	15	11	10	0
COP0	MF0	rt	rd	0					
010000	00000			000 0000 0000					
6	5	5	5	11					

Format

MFC0 rt, rd

Description

$GPR[rt] \leftarrow CPR[0,rd]$

Copies the contents of COP0 coprocessor register rd to GPR[rt].

Exceptions

Coprocessor unusable

Operation

$data \leftarrow CPR[0, rd]$

$GPR[rt] \leftarrow (data_{31})^{32} \parallel data_{31..0}$

MFDAB : Move from Data Address Breakpoint Register

MIPS I

To transfer the data address breakpoint register to a general-purpose register.

Operation Code

31	26	25	21	20	16	15	11	10	0
COP0	MF0		rt		11000			0	
010000	00000							000 0000 0100	
6	5		5		5			11	

Format

MFDAB rt

Description

$GPR[rt] \leftarrow COP0\ DAB$

Copies the contents of the data address breakpoint register (one of the COP0 debug registers) to GPR[rt].

Exceptions

Coprocessor unusable

Operation

$data \leftarrow CPR[0, \text{Data Address Breakpoint}]$

$GPR[rt] \leftarrow (data_{31})^{32} \mid data_{31..0}$

MFDABM : Move from Data Address Breakpoint Mask Register

MIPS I

To transfer the data address breakpoint mask register to a general-purpose register.

Operation Code

31	26	25	21	20	16	15	11	10	0
COP0	MF0		rt		11000			0	
010000	00000							000 0000 0101	
6	5		5		5			11	

Format

MFDABM rt

Description

$GPR[rt] \leftarrow COP0 \text{ DABM}$

Copies the contents of the data address breakpoint mask register (one of the COP0 debug registers) to $GPR[rt]$.

Exceptions

Coprocessor unusable

Operation

$data \leftarrow CPR[0, \text{Data Address Breakpoint Mask}]$

$GPR[rt] \leftarrow (data_{31})^{32} \mid \mid data_{31..0}$

MFDVB : Move from Data value Breakpoint Register

MIPS I

To transfer the data value breakpoint register to a general-purpose register.

Operation Code

31	26	25	21	20	16	15	11	10	0
COP0	MF0		rt		11000		0		
010000	00000						000 0000 0110		
6	5		5		5		11		

Format

MFDVB rt

Description

$GPR[rt] \leftarrow COP0\ DVB$

Copies the contents of the data value breakpoint register (one of the COP0 debug registers) to GPR[rt].

Exceptions

Coprocessor unusable

Operation

$data \leftarrow CPR[0, \text{Data Value Breakpoint}]$

$GPR[rt] \leftarrow (data_{31})^{32} \mid data_{31..0}$

MFDVBM : Move from Data Value Breakpoint Mask Register

MIPS I

To transfer the data value breakpoint mask register to a general-purpose register.

Operation Code

31	26	25	21	20	16	15	11	10	0
COP0	MF0		rt		11000			0	
010000	00000							000 0000 0111	
6	5		5		5			11	

Format

MFDVBM rt

Description

$GPR[rt] \leftarrow COP0\ DVBM$

Copies the contents of the data value breakpoint mask (one of the COP0 debug registers) register to $GPR[rt]$.

Exceptions

Coprocessor unusable

Operation

$data \leftarrow CPR[0, \text{Data Value Breakpoint Mask}]$

$GPR[rt] \leftarrow (data_{31})^{32} \parallel data_{31..0}$

MFIAB : Move from Instruction Address Breakpoint Register

MIPS I

To transfer the instruction address breakpoint register to a general-purpose register.

Operation Code

31	26	25	21	20	16	15	11	10	0
COP0	MF0	rt	11000	0	000 0000 0010				
010000	00000								
6	5	5	5						11

Format

MFIAB rt

Description

$\text{GPR}[\text{rt}] \leftarrow \text{COP0 IAB}$

Copies the contents of the instruction address breakpoint register (one of the COP0 debug registers) to GPR[rt].

Exceptions

Coprocessor unusable

Operation

$\text{data} \leftarrow \text{CPR}[0, \text{Instruction Address Breakpoint}]$

$\text{GPR}[\text{rt}] \leftarrow (\text{data}_{31})^{32} \mid \text{data}_{31..0}$

MFIABM : Move from Instruction Address Breakpoint Mask Register

MIPS I

To transfer the instruction address breakpoint mask register to a general-purpose register.

Operation Code

31	26	25	21	20	16	15	11	10	0
COP0	MF0		rt		11000			0	
010000	00000							000 0000 0011	
6	5		5		5			11	

Format

MFIABM rt

Description

$\text{GPR}[\text{rt}] \leftarrow \text{COP0 IABM}$

Copies the contents of the instruction address breakpoint (one of the COP0 debug registers) mask register to $\text{GPR}[\text{rt}]$.

Exceptions

Coprocessor unusable

Operation

$\text{data} \leftarrow \text{CPR}[0, \text{Instruction Address Breakpoint Mask}]$

$\text{GPR}[\text{rt}] \leftarrow (\text{data}_{31})^{32} \mid \mid \text{data}_{31..0}$

MFPC : Move from Performance Counter

MIPS I

To transfer a performance counter register to a general-purpose register.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	1	0
COP0	MF0		rt		11001		0		reg		1	
010000	00000						00000					
6	5		5		5		5		5		1	

Format

MFPC rt, reg

Description

$GPR[rt] \leftarrow CPR[0, CTR[reg]]$

Copies the contents of a COP0 performance counter register specified by reg to GPR[rt]. The reg field indicates the performance counter number, and only 0 and 1 are valid.

Exceptions

Coprocessor unusable

Operation

$data \leftarrow CPR[0, \text{Performance Counter}(reg)]$

$GPR[rt] \leftarrow (data_{31})^{32} \mid data_{31..0}$

MFPS : Move from Performance Event Specifier

MIPS I

To transfer the performance counter control register to a general-purpose register.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	1	0
COP0	MF0		rt		11001		0		reg		0	
010000	00000						00000		00000			
6	5		5		5		5		5		1	

Format

MFPS rt, reg

Description

$GPR[rt] \leftarrow COP0\ CCR$

Copies the contents of the COP0 performance counter control register to GPR[rt].

The reg field indicates the performance control register number, and 0 must be specified.

Exceptions

Coprocessor unusable

Operation

$data \leftarrow CPR[0, \text{Performance Control}(\text{reg})]$

$GPR[rt] \leftarrow (data_{31})^{32} \mid \mid data_{31..0}$

MTBPC : Move to Breakpoint Control Register

MIPS I

To copy a general-purpose register value to the breakpoint control register.

Operation Code

31	26	25	21	20	16	15	11	10	0
COP0	MT0	rt	11000	0	000 0000 0000				
010000	00100								
6	5	5	5						11

Format

MTBPC rt

Description

Transfers the lower 32 bits of GPR[rt] to the breakpoint control register (one of the COP0 debug registers).

Exceptions

Coprocessor unusable

Operation

$\text{CPR}[0, \text{Breakpoint Control}] \leftarrow \text{GPR}[\text{rt}]_{31..0}$

Note

To guarantee COP0 register update, it is necessary to place a SYNC.P instruction after the MTBPC instruction.

MTC0 : Move to System Control Coprocessor

MIPS I

To copy a general-purpose register value to the COP0 register.

Operation Code

31	26	25	21	20	16	15	11	10	0
COP0	MT0		rt		rd				0
010000	00100								000 0000 0000
6	5		5		5				11

Format

MTC0 rt, rd

Description

Transfers the lower 32 bits of GPR[rt] to CPR[0,rd].

Exceptions

Coprocessor unusable

Operation

$CPR[0, rd] \leftarrow GPR[rt]_{31..0}$

Note

To guarantee COP0 register update, it is necessary to place a SYNC.P instruction after the MTC0 instruction. Due to a hardware interlock, however, data transfer to the EntryHi register via an MTC0 instruction can be immediately followed by a TLBWI or a TLBWR instruction without having a SYNC.P instruction.

The MTC0 instruction, which transfers data to the EntryHi register, must be executed either in unmapped space or in global mapped space (which is mapped by a TLB entry with the G bit set to 1). In addition, the BTAC is flushed whenever the EntryHi register is updated.

An MTC0 instruction should not be executed in kseg0 space to modify the K0 bit of the CONFIG register. A SYNC.L instruction is needed before executing an MTC0 instruction which modifies NBE or DCE of the CONFIG register.

Setting the performance counters via an MTC0 instruction while the performance counters are enabled will result in undefined counter values.

MTDAB : Move to Data Address Breakpoint Register

MIPS I

To copy a general-purpose register value to the data address breakpoint register.

Operation Code

31	26	25	21	20	16	15	11	10	0
COP0	MT0	rt	11000	0	000 0000 0100				
010000	00100								
6	5	5	5						11

Format

MTDAB rt

Description

Transfers the lower 32 bits of GPR[rt] to the data address breakpoint register (one of the COP0 debug registers).

Exceptions

Coprocessor unusable

Operation

CPR[0, Data Address Breakpoint] \leftarrow GPR[rt]_{31..0}

Note

To guarantee COP0 register update, it is necessary to place a SYNC.P instruction after the MTDAB instruction.

MTDABM : Move to Data Address Breakpoint Mask Register

MIPS I

To copy a general-purpose register value to the data address breakpoint mask register.

Operation Code

31	26	25	21	20	16	15	11	10	0
COP0	MT0	rt	11000	0	000 0000 0101				
010000	00100								
6	5	5	5						11

Format

MTDABM rt

Description

Transfers the lower 32 bits of GPR[rt] to a data address breakpoint mask register (one of the COP0 debug registers).

Exceptions

Coprocessor unusable

Operation

$\text{CPR}[0, \text{Data Address Breakpoint Mask}] \leftarrow \text{GPR}[\text{rt}]_{31..0}$

Note

To guarantee COP0 register update, it is necessary to place a SYNC.P instruction after the MTDABM instruction.

MTDVB : Move to Data Value Breakpoint Register

MIPS I

To copy a general-purpose register value to the data value breakpoint register.

Operation Code

31	26	25	21	20	16	15	11	10	0
COP0	MT0		rt		11000			0	
010000	00100							000 0000 0110	
6	5		5		5			11	

Format

MTDVB rt

Description

Transfers the lower 32 bits of GPR[rt] to the data value breakpoint register (one of the COP0 debug registers).

Exceptions

Coprocessor unusable

Operation

$\text{CPR}[0, \text{Data Value Breakpoint}] \leftarrow \text{GPR}[\text{rt}]_{31..0}$

Note

To guarantee COP0 register update, it is necessary to place a SYNC.P instruction after the MTDVB instruction.

MTDVBM : Move to Data Value Breakpoint Mask Register

MIPS I

To copy a general-purpose register value to the data value breakpoint mask register.

Operation Code

31	26	25	21	20	16	15	11	10	0
COP0	MT0	rt	11000	0	000 0000 0111				
010000	00100								
6	5	5	5						11

Format

MTDVBM rt

Description

Transfers the lower 32 bits of GPR[rt] to the data value breakpoint mask register (one of the COP0 debug registers).

Exceptions

Coprocessors unusable

Operation

$\text{CPR}[0, \text{Data Value Breakpoint Mask}] \leftarrow \text{GPR}[\text{rt}]_{31..0}$

Note

To guarantee the COP0 register update, it is necessary to place the SYNC.P instruction after the MTDVBM instruction.

MTIAB : Move to Instruction Address Breakpoint Register

MIPS I

To copy a general-purpose register value to an instruction address breakpoint register.

Operation Code

31	26	25	21	20	16	15	11	10	0
COP0	MT0	rt	11000	0	000 0000 0010				
010000	00100								
6	5	5	5						11

Format

MTIAB rt

Function

Description

Transfers the lower 32 bits of GPR[rt] to the instruction address breakpoint register (one of the COP0 debug registers).

Exceptions

Coprocessor unusable

Operation

$\text{CPR}[0, \text{Instruction Address Breakpoint}] \leftarrow \text{GPR}[\text{rt}]_{31:0}$

Note

To guarantee COP0 register update, it is necessary to place a SYNC.P instruction after the MTIAB instruction.

MTIABM : Move to Instruction Address Breakpoint Mask Register

MIPS I

To copy a general-purpose register value to the instruction address breakpoint mask register.

Operation Code

31	26	25	21	20	16	15	11	10	0
COP0	MT0	rt	11000	0	000 0000 0011				
010000	00100								
6	5	5	5						11

Format

MTIABM rt

Description

Transfers the lower 32 bits of GPR[rt] to the instruction address breakpoint mask register (one of the COP0 debug registers).

Exceptions

Coprocessor unusable

Operation

$\text{CPR}[0, \text{Instruction Address Breakpoint Mask}] \leftarrow \text{GPR}[\text{rt}]_{31..0}$

Note

To guarantee the COP0 register update, it is necessary to place a SYNC.P instruction after the MTIABM instruction.

MTPC : Move to Performance Counter

MIPS I

To copy a general-purpose register value to a performance counter.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	1	0
COP0	MT0	rt	11001	0	reg	1						
010000	00100				00000							
6	5	5	5	5	5	5	5	5	5	5	1	

Format

MTPC rt, reg

Description

Copies the lower 32 bits of GPR[rt] to the COP0 performance counter register number specified by reg. Only 0 and 1 are valid values for reg.

Exceptions

Coprocessor unusable

Operation

$CPR[0, CTR[reg]] \leftarrow GPR[rt]_{31..0}$

Note

To guarantee COP0 register update, it is necessary to place a SYNC.P instruction after the MTPC instruction. If the performance counter is set by the MTPC instruction when the performance counter is in operation, the counter values will be undefined.

MTPS : Move to Performance Event Specifier

MIPS I

To copy a general-purpose register value to a performance counter control register.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	1	0
COP0	MT0	rt	11001	0	reg	0						
010000	00100					00000	00000					
6	5	5	5	5	5	5	5	1				

Format

MTPS rt, reg

Description

Transfers the lower 32 bits of GPR[rt] to a performance counter control register specified by reg. Only 0 is valid for reg.

Exceptions

Coprocessor unusable

Operation

$CPR[0, CCR] \leftarrow GPR[rt]_{31:0}$

Note

To guarantee COP0 register update , it is necessary to place a SYNC.P instruction after the MTPS instruction.

TLBP : Probe TLB for Matching Entry

MIPS I

To probe the TLB entry whose contents match the contents of the EntryHi register.

Operation Code

31	26	25	21	20	16	15	0
COP0	C0			0		TLBP	
010000	10000			000 0000 0000 0000		001000	
6	5			15		6	

Format

TLBP

Description

Probes TLB entries and stores the index of the entry whose contents match the contents of the EntryHi register. Sets the high-order bit of the Index register to 1 if no TLB entry matches.

The contents (virtual address) of the EntryHi register is masked with the corresponding mask field of the TLB entry prior to comparison.

Exceptions

Coprocessor unusable.

Operation

```

Index ← 1 | 025 | undefined6
for i in 0..TLBEntries - 1
  if (TLB[i]95..77 = ( (NOT TLB[i]127..109) AND EntryHi31..13 )
    AND (TLB[i]76 OR (TLB[i]71..64 = EntryHi7..0)) then
    Index ← 026 | i5..0
  endif
endfor

```

Note

The operation is indeterminate if more than one TLB entry matches.

The operation is undefined when memory is accessed with the instruction immediately after a TLBP instruction.

TLBR : Read Indexed TLB Entry

MIPS I

To read the TLB entry pointed at by the contents of the Index register.

Operation Code

31	26	25	21	20	16	15	0
COP0	C0	0				TLBR	
010000	10000	000 0000 0000 0000				000001	
6	5	15				6	

Format

TLBR

Description

Reads the contents of the TLB entry pointed at by the contents of the Index register, and stores them in PageMask, EntryHi, EntryLo0 and EntryLo1 registers.

The G bit of TLB is written into both of the EntryLo0 and EntryLo1 registers.

Restrictions

This instruction must be executed in either unmapped space or global mapped space (memory area mapped with a TLB entry which has the G bit set to 1).

The TLBR instruction must be immediately followed by SYNC.P or ERET instruction.

Exceptions

Coprocessor unusable.

Operation

$\text{PageMask} \leftarrow \text{TLB}[\text{Index}_{5.0}]_{127..96}$

$\text{EntryHi} \leftarrow (\text{TLB}[\text{Index}_{5.0}]_{95..77} \mid \mid 0^5 \mid \mid \text{TLB}[\text{Index}_{5.0}]_{71..64}) \text{ AND } (\text{NOT TLB}[\text{Index}_{5.0}]_{127..96})$

$\text{EntryLo0} \leftarrow \text{TLB}[\text{Index}_{5.0}]_{63..33} \mid \mid \text{TLB}[\text{Index}_{5.0}]_{76}$

$\text{EntryLo1} \leftarrow \text{TLB}[\text{Index}_{5.0}]_{31..1} \mid \mid \text{TLB}[\text{Index}_{5.0}]_{76}$

Note

Depending on the value in the PageMask register, the value read from the TLB entry may be different from what was written by TLBWI/TLBWR instruction.

TLBWI : Write Index TLB Entry

MIPS I

To write to the TLB entry pointed at by the Index register.

Operation Code

31	26	25	21	20	16	15	0
COP0	C0	0				TLBWI	
010000	10000	000 0000 0000 0000				000010	
6	5	15				6	

Format

TLBWI

Description

Stores the contents of the PageMask, EntryHi, EntryLo0, and EntryLo1 registers in the TLB entry pointed at by the Index register.

The logical AND of the G bits in the EntryLo0 and EntryLo1 registers is written into the G bit of TLB.

Restrictions

The operation is indeterminate if the contents of the Index register are greater than the number of TLB entries .

This instruction must be executed in either unmapped space or global mapped space (memory area mapped with a TLB entry which has the G bit set to 1).

The TLBWI instruction must be followed by a SYNC.P or an ERET instruction to insure TLB update.

Exceptions

Coprocessor unusable.

Operation

$$\begin{aligned} \text{TLB}[\text{Index}_{5..0}] &\leftarrow \text{PageMask} \\ &|| ((\text{EntryHi}_{31..13} || (\text{EntryLo0}_0 \text{ AND } \text{EntryLo1}_0) || \text{EntryHi}_{11..0}) \\ &\quad \text{AND (NOT PageMask)}) \\ &|| \text{EntryLo0}_{31..1} || 0_1 || \text{EntryLo1}_{31..1} || 0_1 \end{aligned}$$

Programming Notes

Depending on the page size specified in the PageMask register, the lower bits of PFN may not be used for address translation.

The lower bits of the VPN2 field of the EntryHi register are partially masked by the contents of the PageMask register. Therefore, a TLB entry read by a TLBR instruction may be different from what was originally written.

TLBWR : Write Random TLB Entry

MIPS I

To write to the TLB entry pointed at by the Random register.

Operation Code

31	26	25	21	20	16	15	0
COP0	C0	0				TLBWR	
010000	10000	000 0000 0000 0000				000110	
6	5	15				6	

Format

TLBWR

Description

Stores the contents of the PageMask, EntryHi, EntryLo0, and EntryLo1 registers in the TLB entry pointed at by the Index register.

The logical AND of the G bits in the EntryLo0 and EntryLo1 registers is written to the G bit of TLB.

Restrictions

This instruction must be executed in either unmapped space or global mapped space (memory area mapped with a TLB entry which has the G bit set to 1).

The TLBWR instruction must be followed by a SYNC.P or an ERET instruction to insure TLB update.

Exceptions

Coprocessor unusable.

Operation

$$\begin{aligned} \text{TLB}[\text{Random}_{5..0}] &\leftarrow \text{PageMask} \\ &|| ((\text{EntryHi}_{31..13} || (\text{EntryLo0}_0 \text{ AND } \text{EntryLo1}_0) || \text{EntryHi}_{11..0}) \\ &\quad \text{AND } (\text{NOT PageMask})) \\ &|| \text{EntryLo0}_{31..1} || 0_1 || \text{EntryLo1}_{31..1} || 0_1 \end{aligned}$$

Programming Notes

Depending on the page size specified in the PageMask register, the lower bits of PFN may not be used for address translation.

The lower bits of the VPN2 field of the EntryHi register are partially masked by the contents of the PageMask register. Therefore, a TLB entry read by a TLBR instruction may be different from what was originally written.

5. COP1 (FPU) Instruction Set

This chapter describes the COP1 coprocessor instructions that are Floating-Point Unit (FPU). The FPU consists of 32 Floating-Point Registers (FPR) of 32-bit each, an Accumulator (ACC), and two Control Registers (FCR). The FPU can handle both single-precision floating-point values and fixed-point values (32-bit signed integers). The COP1 instruction is categorized as the following:

- Addition, subtraction, multiplication and division of floating-point values
- Product-sum or difference operation
- Comparison of floating-point values
- Branch by the comparison
- Numerical format conversion
- Data transfer to/from the CPU
- Load/Store between memory and FPRs

Flags that reflect the status of the results are allocated to bits of FCR31 as shown below. Flags that are not indicated in the instructions are not changed.

Flag		Description	Bit
Condition Flag	Flag C	Comparison Flag	FCR31 ₂₃
Cause Flag	Flag I	Invalid Operation Flag	FCR31 ₁₇
	Flag D	0 Division Flag	FCR31 ₁₆
	Flag O	Overflow Flag	FCR31 ₁₅
	Flag U	Underflow Flag	FCR31 ₁₄
Sticky Flag	Flag SI	Invalid Operation Cumulative Flag	FCR31 ₆
	Flag SD	0 Division Cumulative Flag	FCR31 ₅
	Flag SO	Overflow Cumulative Flag	FCR31 ₄
	Flag SU	Underflow Cumulative Flag	FCR31 ₃

ABS.S : Floating Point Absolute Value

MIPS I

To obtain the absolute value of a single-precision floating-point value.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
COP1	S	0	fs	fd	ABS						
010001	10000	00000								000101	
6	5	5	5	5	5	6					

Format

ABS.S fd, fs

Description

$FPR[fd] \leftarrow ABS(FPR[fs])$

Obtains the single-precision floating-point absolute value of FPR[fs] and stores it in FPR[fd].

Exceptions

Coprocessor unusable

Operation

$FPR[fd] \leftarrow ABS(FPR[fs])$

Flag O $\leftarrow 0$

Flag U $\leftarrow 0$

ADD.S : Floating Point ADD

MIPS I

To add single-precision floating-point values.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
COP1	S		ft		fs		fd		ADD		
010001	10000								000000		
6	5		5		5		5		5		6

Format

ADD.S fd, fs, ft

Description

$FPR[fd] \leftarrow FPR[fs] + FPR[ft]$

Adds the single-precision floating-point values of FPR[fs] and FPR[ft], then stores the result in FPR[fd].

When an exponent overflow occurs, Flag O and Flag SO are set to 1 and + maximum or - maximum is stored in FPR[fd] as the result. When an exponent underflow occurs, Flag U and Flag SU are set to 1 and +0 or -0 is stored in FPR[fd] as the result.

Exceptions

Coprocessor unusable. Floating-point exceptions such as invalid operation, inexact, overflow and underflow are not generated by this instruction.

Operation

$FPR[fd] \leftarrow FPR[fs] + FPR[ft]$

Flag O \leftarrow 1 if exponent overflows.

Flag U \leftarrow 1 if exponent underflows.

Flag SO \leftarrow 1 if exponent overflows.

Flag SU \leftarrow 1 if exponent underflows.

ADDA.S : Floating Point Add to Accumulator

EE Core

To add single-precision floating-point values and store the result in an accumulator.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0	
COP1			S		ft		fs		0		ADDA	
010001			10000						00000		011000	
6			5		5		5		5		6	

BC1F : Branch on FP False

MIPS I

To branch according to the result of the immediately preceding floating-point comparison operation.

Operation Code

31	26	25	21	20	16	15	0
COP1	BC1	BC1F	offset				
010001	01000	00000					
6	5	5	16				

Format

BC1F offset

Description

if (C = 0) then branch

Checks Flag C (FCR31₂₃), which shows the result of the immediately preceding floating-point comparison operation. In the case of 0 (false), branches to the target address after execution of the branch delay instruction. The target address is obtained by adding an 18-bit signed offset (the offset field left-shifted by 2 bits) to the instruction address of the branch delay slot.

Exceptions

Coprocessor unusable

Operation

```

I:      condition ← (FCR3123 = 0)
        target_offset ← (offset15)GPRLEN-(16+2) || offset || 02
I+1:    if condition then
            PC ← PC + target_offset
        endif

```

Programming Notes

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB.

To branch to more distant addresses, use the J or JR instructions.

BC1FL : Branch on FP False Likely

MIPS II

To branch according to the result of the immediately preceding floating-point comparison operation. Executes the branch delay slot instruction only when the branch is taken.

Operation Code

31	26	25	21	20	16	15	0
COP1						BC1	
010001						01000	
						BC1FL	
						00010	
						offset	
6						5	
						5	
						16	

Format

BC1FL offset

Description

if ($C = 0$) then branch_likely

Checks Flag C ($FCR31_{23}$), which shows the result of the immediately preceding floating-point comparison operation. In the case of 0 (false), branches to the target address after execution of the branch delay instruction. In the case of 1 (true), nullifies the branch delay slot instruction. The target address is obtained by adding an 18-bit signed offset (the offset field left-shifted by 2 bits) to the instruction address of the branch delay slot.

Exceptions

Coprocessor unusable

Operation

```

I:      condition ← ( $FCR31_{23} = 0$ )
        target_offset ← ( $offset_{15}$ )GPRELEN-(16+2) || offset || 02
I+1:    if condition then
            PC ← PC + target_offset
        else
            NullifyCurrentInstruction()
        endif

```

Programming Notes

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB.

To branch to more distant addresses, use the J or JR instructions.

BC1T : Branch on FP True

MIPS I

To branch according to the result of the immediately preceding floating-point comparison operation.

Operation Code

31	26	25	21	20	16	15	0
COP1	BC1	BC1T	offset				
010001	01000	00001					
6	5	5	16				

Format

BC1T offset

Description

if (C = 1) then branch

Checks Flag C (FCR31₂₃), which shows the result of the immediately preceding floating-point comparison operation. In the case of 1 (true), branches to the target address after execution of the branch delay instruction. The target address is obtained by adding an 18-bit signed offset (the offset field left-shifted by 2 bits) to the instruction address of the branch delay slot.

Exceptions

Coprocessor unusable

Operation

```

I:      condition ← (FCR3123 = 1)
        target_offset ← (offset15)GPRLEN-(16+2) || offset || 02
I+1:    if condition then
            PC ← PC + target_offset
        endif

```

Programming Notes

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB.

To branch to more distant addresses, use the J or JR instructions.

BC1TL : Branch on FP True Likely

MIPS II

To branch according to the result of the immediately preceding floating-point comparison operation. Executes the branch delay slot instruction only when the branch is taken.

Operation Code

31	26	25	21	20	16	15	0
COP1 010001	BC1 01000	BC1TL 00011	offset				
6	5	5	16				

Format

BC1TL offset

Description

if (C = 1) then branch_likely

Checks Flag C (FCR31₂₃), which shows the result of the immediately preceding floating-point comparison operation. In the case of 1 (true), branches to the target address after execution of the branch delay instruction. The target address is obtained by adding an 18-bit signed offset (the offset field left-shifted by 2 bits) to the instruction address of the branch delay slot.

Exceptions

Coprocessor unusable

Operation

```

I:      condition ← (FCR3123 = 1)
        target_offset ← (offset15)GPRELEN-(16+2) || offset || 02
I+1:    if condition then
            PC ← PC + target_offset
        else
            NullifyCurrentInstruction()
        endif

```

Programming Notes

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB.

To branch to more distant addresses, use the J or JR instructions.

C.EQ.S : Floating Point Compare

MIPS I

To compare single-precision floating-point values and record the result in Flag C.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	4	3	2	1	0
COP1	S		ft		fs			0	FC	0	cond	0			
010001	10000							00000	11		01				
6	5		5		5			5	2	1	2	1			

Format

C.EQ.S fs, ft

Description

$C \leftarrow (FPR[fs] = FPR[ft])$

Compares the single-precision floating-point values of FPR[fs] and FPR[ft]. If they are equal, sets Flag C (FCR31₂₃) to 1 (true) and sets it to 0 (false) otherwise. The comparison is exact; therefore no overflow nor underflow occurs. Note that the FPU does not support NaN (non-numeric). $+0 = -0$, as a zero sign is disregarded.

Exceptions

Coprocessor unusable

Operation

```

if (FPR[fs] = FPR[ft]) then
    FCR3123    ← 1
else
    FCR3123    ← 0
endif

```

C.F.S : Floating Point Compare

MIPS I

To record the comparison of single-precision floating-point values in Flag C.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	4	3	2	1	0
COP1	S		ft		fs			0	FC	0	cond	0			
010001	10000							00000	11		00				
6	5		5		5			5	2	1	2	1			

Format

C.F.S fs, ft

Description

$C \leftarrow 0$

Compares the single-precision floating-point values of FPR[fs] and FPR[ft].

Sets Flag C (FCR31₂₃) to 0 (false) regardless of the result.

Exceptions

Coprocessor unusable

Operation

FCR31₂₃ $\leftarrow 0$

C.LE.S : Floating Point Compare

MIPS I

To compare single-precision floating-point values and record the result in Flag C.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	4	3	2	1	0
COP1	S		ft		fs			0	FC	0	cond	0			
010001	10000							00000	11		11				
6	5		5		5			5	2	1	2	1			

Format

C.LE.S fs, ft

Description

$C \leftarrow (FPR[fs] \leq FPR[ft])$

Compares the single-precision floating-point values of FPR[fs] and FPR[ft]. If the value of FPR[fs] is smaller than the value of FPR[ft], sets Flag C (FCR31₂₃) to 1 (true) and sets it to 0 (false) otherwise. The comparison is exact; therefore no overflow nor underflow occurs. $+0 = -0$, as a zero sign is disregarded.

Exceptions

Coprocessor unusable

Operation

```

if (FPR[fs] <= FPR[ft]) then
    FCR3123 ← 1
else
    FCR3123 ← 0
endif

```

C.LT.S : Floating Point Compare

MIPS I

To compare single-precision floating-point values and record the result in Flag C.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	4	3	2	1	0
COP1	S		ft		fs			0	FC	0	cond	0			
010001	10000							00000	11		10				
6	5		5		5			5	2	1	2	1			

Format

C.LT.S fs, ft

Description

$C \leftarrow (FPR[fs] < FPR[ft])$

Compares the single-precision floating-point values of FPR[fs] and FPR[ft]. If the value of FPR[fs] is smaller than the value of FPR[ft], sets Flag C (FCR31₂₃) to 1 (true) and sets it to 0 (false) otherwise. The comparison is exact; therefore no overflow nor underflow occurs. $+0 = -0$, as a zero sign is disregarded.

Exceptions

Coprocessor unusable

Operation

```

if (FPR[fs] < FPR[ft]) then
    FCR3123 ← 1
else
    FCR3123 ← 0
endif

```

CFC1 : Move Control Word from Floating Point

MIPS I

To copy the contents of an FPU control register to a GPR.

Operation Code

31	26	25	21	20	16	15	11	10	0
COP1	CFC1		rt		fs				0
010001	00010								000 0000 0000
6	5		5		5				11

Format

CFC1 rt, fs

Description

$GPR[rt] \leftarrow FCR[fs]$

Sign-extends the contents of the COP1 (FPU) control register FCR[fs] and stores them in GPR[rt].

Exceptions

Coprocessor unusable

Operation

$GPR[rt] \leftarrow \text{sign_extend}(FCR[fs])$

Programming Notes

FCR[0] is the Implementation / Revision register and FCR[31] is the Control / Status register.

CTC1 : Move Control Word to Floating Point

MIPS I

To copy the contents of an FPU control register to a general-purpose register.

Operation Code

31	26	25	21	20	16	15	11	10	0
COP1	CTC1		rt		fs			0	
010001	00110							000 0000 0000	
6	5		5		5			11	

Format

CTC1 rt, fs

Description

$\text{FCR}[\text{fs}] \leftarrow \text{GPR}[\text{rt}]$

Transfers the lower 32 bits of GPR[rt] to the control register FCR[fs] of the COP1 (FPU).

Restrictions

This operation is only defined when fs is 0 or 31.

Exceptions

Coprocessor unusable

Operation

$\text{FCR}[\text{fs}] \leftarrow \text{GPR}[\text{rt}]_{31..0}$

CVT.S.W : Fixed-point Convert to Single Floating Point

MIPS I

To convert a 32-bit signed integer to a single-precision floating-point value.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
COP1	W		0		fs		fd		CVTS		
010001	10100		00000						100000		
6	5		5		5		5		6		

Format

CVT.S.W fd, fs

Description

$FPR[fd] \leftarrow \text{convert_and_round}(FPR[fs])$

Converts the value of FPR[fs] to a single-precision floating-point value considered to be a 32-bit signed integer and stores it in FPR[fd].

Exceptions

Coprocessor unusable

Operation

$FPR[fd] \leftarrow \text{convert}(FPR[fs], S)$

CVT.W.S : Floating Point Convert to Word Fixed-point

MIPS I

To convert a 32-bit signed integer to a single-precision fixed-point value.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
COP1	S	0	fs	fd	CVTW						
010001	10000	00000									100100
6	5	5	5	5	5	6					

Format

CVT.W.S fd, fs

Description

$\text{FPR}[\text{fd}] \leftarrow \text{convert_and_round}(\text{FPR}[\text{fs}])$

Converts the value of FPR[fs] to a 32-bit signed integer considered to be a single-precision floating-point value and stores it in FPR[fd]. If the biased exponent of FPR[fs] exceeds 0x9d, clamping is performed.

Exceptions

Coprocessor unusable

Operation

```

if (FPR[fs]30..23 <= 0x9d) then
    FPR[fd] ← convert(FPR[fs], W)
else if (FPR[fs]31 = 0) then
    FPR[fd] ← 0x7FFFFFFF
else
    FPR[fd] ← 0x80000000
endif

```

DIV.S : Floating Point Divide

MIPS I

To divide single-precision floating-point values .

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
COP1	S		ft		fs		fd			DIV	
010001	10000									000011	
6	5		5		5		5			6	

Format

DIV.S fd, fs, ft

Description

$FPR[fd] \leftarrow FPR[fs] / FPR[ft]$

Divides the value of FPR[fs] by the value of FPR[ft] and stores the result in FPR[fd]. Every value is handled as a single-precision floating-point value.

When FPR[ft] is zero and FPR[fs] is not zero, Flag D and Flag SD are set to 1 with the value of +maximum or -maximum as the result. When both operands are zeros(0/0), Flag I and Flag SI are set to 1 with the value of +maximum or -maximum as the result. When an exponent overflow occurs, the value is +maximum or -maximum. When an exponent underflow occurs, the result value is +0 or -0.

Exceptions

Coprocessor unusable. Floating-point exceptions such as invalid operation, inexact, overflow and underflow are not generated by this instruction.

Operation

$FPR[fd] \leftarrow FPR[fs] / FPR[ft]$
 Flag I $\leftarrow 1$ if $FPR[fs] = 0$ AND $FPR[ft] = 0$
 Flag D $\leftarrow 1$ if $FPR[fs] \neq 0$ AND $FPR[ft] = 0$
 Flag SI $\leftarrow 1$ if $FPR[fs] = 0$ AND $FPR[ft] = 0$
 Flag SD $\leftarrow 1$ if $FPR[fs] \neq 0$ AND $FPR[ft] = 0$

LWC1 : Load Word to Floating Point

MIPS I

To load a word from memory into a floating-point register.

Operation Code

31	26	25	21	20	16	15	0
LWC1	base	ft	offset				
110001							
6	5	5	16				

Format

LWC1 ft, offset(base)

Description

$FPR[ft] \leftarrow \text{memory}[GPR[base] + \text{offset}]$

Adds the 16-bit signed offset to the value of GPR[base] and the word data at the obtained effective address is read into FPR[ft].

The above is a blocking load. If a data cache miss occurs, the pipeline stalls while loading data from memory and transferring it to FPR[ft].

Restrictions

The effective address must comply with word alignment. In other cases, an address error exception occurs if the lower 2 bits of the effective address are not 0.

Exceptions

Coprocessor unusable
 TLB Refill
 TLB Invalid
 Address Error

Operation (128-bit bus)

```

vAddr      ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
  SignalException(AddressError)
endif
(pAddr, uncached) ← AddressTranslation(vAddr, DATA, LOAD)
memquad     ← LoadMemory(uncached, WORD, pAddr, vAddr, DATA)
byte        ← vAddr3..0
FPR[ft]     ← memquad31+8*byte..8*byte
  
```

MADD.S : Floating Point Multiply-ADD

MIPS I

To multiply single-precision floating-point values and add to the accumulator.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
COP1	S		ft		fs		fd			MADD	
010001	10000									011100	
6	5		5		5		5		5	6	

Format

MADD.S fd, fs, ft

Description

$FPR[fd] \leftarrow ACC + FPR[fs] \times FPR[ft]$

Multiplies the value of FPR[fs] by the value of FPR[ft] and adds the result to the value of accumulator ACC. Stores the result in FPR[fd]. The results are all single-precision floating-point values. When an exponent overflow occurs, Flag O and Flag SO are set to 1 with the value of +maximum or -maximum as the result. When an exponent underflow occurs, Flag U and Flag SU are set to 1 with the value of +0 or -0 as the result.

Exceptions

Coprocessor unusable. Floating-point exceptions such as invalid operation, inexact, overflow and underflow are not generated by this instruction.

Operation

```

TEMP ← fs × ft
if (TEMP.exp = plusminuszero) then // multiply underflow occurred
    FCR31.SU ← 1
if (ACC.exp >= 0xFF) then // Exp of ACC is in overflow state at instruction entry
    if (TEMP.exp >= 0xFF) then
        FPR[fd] ← maxmin(TEMP.sign)
    else
        FPR[fd] ← ACC
endif
FCR31.SO ← 1
FCR31.O ← 1
else // Exp of ACC is normal or in underflow state at instruction entry
    if (TEMP.exp >= 0xFF) then
        FPR[fd] ← maxmin(TEMP.sign)
        FCR31.SO ← 1
        FCR31.O ← 1
    else
        if (TEMP.exp = 0x00) then
            FPR[fd] ← ACC
        else
            TEMP1 ← ACC + TEMP
            if (TEMP1.exp >= 0xFF) then
                FPR[fd] ← maxmin(TEMP1.sign)
                FCR31.SO ← 1
                FCR31.O ← 1
            else
                if (TEMP1.exp = 0x00) then

```

```

                                FPR[fd] ← signedzero(TEMP1.sign)
                                FCR31.SU ← 1
                                FCR31.U ← 1
                                else
                                FPR[fd] ← TEMP1
                                endif
                            endif
                        endif
                    endif
                endif
            endif
        endif
    endif
end

maxmin(sign)
begin
    if (sign = 1)
        maxmin ← 0xFFFFFFFF
    else
        maxmin ← 0x7FFFFFFF
    end
end

signedzero(sign)
begin
    if (sign = 1)
        signedzero ← 0x80000000
    else
        signedzero ← 0x00000000
    end
end
```

MADDA.S : Floating Point Multiply-Add

EE Core

To multiply Single-precision floating-point values and add to the accumulator.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
COP1	S	ft	fs	0	MADDA						
010001	10000				00000	011110					
6	5	5	5	5	5	6					

Format

MADDA.S fs, ft

Description

$$ACC \leftarrow ACC + FPR[fs] \times FPR[ft]$$

Multiplies the value of FPR[fs] by the value of FPR[ft] and adds the result to the Accumulator ACC; that result is stored in the Accumulator ACC. The results are all single-precision floating-point values. When an exponent overflow occurs, Flag O and Flag SO are set to 1 with the value of +maximum or -maximum as a result. When an exponent underflow occurs, Flag U and Flag SU are set to 1 with the value of +0 or -0 as a result.

Exceptions

Coprocessor unusable. Floating-point exceptions such as invalid operation, inexact, overflow and underflow are not generated by this instruction.

Operation

```

TEMP ← fs × ft
if (TEMP.exp = plusminuszero) then // multiply underflow occurred
    FCR31.SU ← 1
if (ACC.exp >= 0xFF) then // Exp of ACC is in overflow state at instruction entry
    if (TEMP.exp >= 0xFF) then
        ACC ← maxmin(TEMP.sign)
    else
        ACC ← ACC
endif
FCR31.SO ← 1
FCR31.O ← 1
else // Exp of ACC is normal or in underflow state at instruction entry
    if (TEMP.exp >= 0xFF) then
        ACC ← maxmin(TEMP.sign)
        FCR31.SO ← 1
        FCR31.O ← 1
    else
        if (TEMP.exp = 0x00) then
            ACC ← ACC
        else
            TEMP1 ← ACC + TEMP
            if (TEMP1.exp >= 0xFF) then
                ACC ← maxmin(TEMP1.sign)
                FCR31.SO ← 1
                FCR31.O ← 1
            else

```

```
        if (TEMP1.exp = 0x00) then
            ACC ← signedzero(TEMP1.sign)
            FCR31.SU ← 1
            FCR31.U ← 1
        else
            ACC ← TEMP1
        endif
    endif
endif

maxmin(sign)
begin
    if (sign = 1)
        maxmin ← 0xFFFFFFFF
    else
        maxmin ← 0x7FFFFFFF
    end
end

signedzero(sign)
begin
    if (sign = 1)
        signedzero ← 0x80000000
    else
        signedzero ← 0x00000000
    end
end
```


MAX.S : Floating Point Maximum

EE Core

To obtain the maximum of two single-precision floating-point values.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
COP1	S		ft		fs		fd			MAX	
010001	10000									101000	
6	5		5		5		5			6	

Format

MAX.S fd, fs, ft

Description

$FPR[fd] \leftarrow \max(FPR[fs], FPR[ft])$

Compares the values of FPR[fs] and FPR[ft] and stores the greater value in FPR[fd]. Flag O and Flag U are cleared to zero.

Exceptions

Coprocessor unusable

Operation

```

if (FPR[fs] >= FPR[ft]) then
    FPR[fd] ← FPR[fs]
else
    FPR[fd] ← FPR[ft]
endif
Flag O ← 0
Flag U ← 0

```

MFC1 : Move Word from Floating Point

MIPS I

To copy the contents of a floating-point register (FPR) to a general-purpose register (GPR).

Operation Code

31	26	25	21	20	16	15	11	10	0
COP1	MFC1		rt		fs			0	
010001	00000							000 0000 0000	
6	5		5		5			11	

Format

MFC1 rt, fs

Description

$GPR[rt] \leftarrow FPR[fs]$

Sign-extends the FPR[fs] value and stores it in GPR[rt].

Exceptions

Coprocessor unusable

Operation

$GPR[rt] \leftarrow \text{sign_extend}(FPR[fs])$

MIN.S : Floating Point Minimum

EE Core

To obtain the minimum of two single-precision floating-point values.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
COP1	S		ft		fs		fd			MIN	
010001	10000									110000	
6	5		5		5		5			6	

Format

MIN.S fd, fs, ft

Description

$FPR[fd] \leftarrow \min(FPR[fs], FPR[ft])$

Compares the values of FPR[fs] and FPR[ft] and stores the lesser value in FPR[fd].

Flag O and Flag U are cleared to zero.

Exceptions

Coprocessor unusable

Operation

if (FPR[fs] <= FPR[ft]) then

 FPR[fd] ← FPR[fs]

else

 FPR[fd] ← FPR[ft]

endif

Flag O ← 0

Flag U ← 0

MOV.S : Floating Point Move

MIPS I

To move data between floating-point registers (FPR).

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
COP1	S	0	fs	fd	MOV						
010001	10000	00000									000110
6	5	5	5	5	5	5	5	5	5	5	6

Format

MOV.S fd, fs

Description

$FPR[fd] \leftarrow FPR[fs]$

Stores the value of FPR[fs] in FPR[fd].

Exceptions

Coprocessor unusable

Operation

$FPR[fd] \leftarrow FPR[fs]$

MSUB.S : Floating Point Multiply and Subtract

MIPS I

To multiply single-precision floating-point values and subtract from Accumulator.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
COP1	S	ft	fs	fd	MSUB						
010001	10000				011101						
6	5	5	5	5	6						

Format

MSUB.S fd, fs, ft

Description

$FPR[fd] \leftarrow ACC - FPR[fs] \times FPR[ft]$

Multiplies the value of FPR[fs] by the value of FPR[ft] and subtracts the product from the Accumulator ACC, then stores the result in FPR[fd]. The results are all single-precision floating-point values. When an exponent overflow occurs, Flag O and Flag SO are set to 1 with the value of +maximum or -maximum as a result. When an exponent underflow occurs, Flag U and Flag SU are set to 1 with the value of +0 or -0 as a result.

Exceptions

Coprocessor unusable. Floating-point exceptions such as invalid operation, inexact, overflow and underflow are not generated by this instruction.

Operation

```

TEMP ← fs × ft
if (TEMP.exp = plusminuszero) then // multiply underflow occurred
    FCR31.SU ← 1
if (ACC.exp >= 0xFF) then // Exp of ACC is in overflow state at instruction entry
    if (TEMP.exp >= 0xFF) then
        FPR[fd] ← maxmin(TEMP.sign)
    else
        FPR[fd] ← ACC
endif
FCR31.SO ← 1
FCR31.O ← 1
else // Exp of ACC is normal or in underflow state at instruction entry
    if (TEMP.exp >= 0xFF) then
        FPR[fd] ← maxmin(TEMP.sign)
        FCR31.SO ← 1
        FCR31.O ← 1
    else
        if (TEMP.exp = 0x00) then
            FPR[fd] ← ACC
        else
            TEMP1 ← ACC - TEMP
            if (TEMP1.exp >= 0xFF) then
                FPR[fd] ← maxmin(TEMP1.sign)
                FCR31.SO ← 1
                FCR31.O ← 1
            else

```

```
        if (TEMP1.exp = 0x00) then
            FPR[fd] ← signedzero(TEMP1.sign)
            FCR31.SU ← 1
            FCR31.U ← 1
        else
            FPR[fd] ← TEMP1
        endif
    endif
endif

maxmin(sign)
begin
    if (sign = 1)
        maxmin ← 0xFFFFFFFF
    else
        maxmin ← 0x7FFFFFFF
    end
end

signedzero(sign)
begin
    if (sign = 1)
        signedzero ← 0x80000000
    else
        signedzero ← 0x00000000
    end
end
```

MSUBA.S : Floating Point Multiply and Subtract from Accumulator

EE Core

To multiply single-precision floating-point values and subtract from Accumulator.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
COP1	S	ft	fs	0	MSUBA						
010001	10000				00000						011111
6	5	5	5	5	5	6					

Format

MSUBA.S fs, ft

Description

$$ACC \leftarrow ACC - FPR[fs] \times FPR[ft]$$

Multiplies the value of FPR[fs] by the value of the FPR[ft], subtracts the product from the Accumulator ACC, then writes the result back to the ACC. The results are all single-precision floating-point values.

When an exponent overflow occurs, Flag O and Flag SO are set to 1 with +maximum or -maximum as a result. When an exponent underflow occurs, Flag U and Flag SU are set to 1 and the result value is +0 or -0.

Exceptions

Coprocessor unusable. Floating-point exceptions such as invalid operation, inexact, overflow and underflow are not generated by this instruction.

Operation

```

TEMP ← fs × ft
if (TEMP.exp = plusminuszero) then // multiply underflow occurred
    FCR31.SU ← 1
if (ACC.exp >= 0xFF) then // Exp of ACC is in overflow state at instruction entry
    if (TEMP.exp >= 0xFF) then
        ACC ← maxmin(TEMP.sign)
    else
        ACC ← ACC
endif
FCR31.SO ← 1
FCR31.O ← 1
else // Exp of ACC is normal or in underflow state at instruction entry
    if (TEMP.exp >= 0xFF) then
        ACC ← maxmin(TEMP.sign)
        FCR31.SO ← 1
        FCR31.O ← 1
    else
        if (TEMP.exp = 0x00) then
            ACC ← ACC
        else
            TEMP1 ← ACC - TEMP
            if (TEMP1.exp >= 0xFF) then
                ACC ← maxmin(TEMP1.sign)
                FCR31.SO ← 1
                FCR31.O ← 1
            else
                if (TEMP1.exp = 0x00) then

```

```

                                ACC ← signedzero(TEMP1.sign)
                                FCR31.SU ← 1
                                FCR31.U ← 1
                                else
                                ACC ← TEMP1
                                endif
                            endif
                        endif
                    endif
                endif
            endif
        endif
    endif
end

maxmin(sign)
begin
    if (sign = 1)
        maxmin ← 0xFFFFFFFF
    else
        maxmin ← 0x7FFFFFFF
    end
end

signedzero(sign)
begin
    if (sign = 1)
        signedzero ← 0x80000000
    else
        signedzero ← 0x00000000
    end
end
```


MTC1 : Move Word to Floating Point

MIPS I

To copy the contents of a general-purpose register (GPR) to a floating-point register (FPR).

Operation Code

31	26	25	21	20	16	15	11	10	0
COP1	MTC1		rt		fs				0
010001	00100								000 0000 0000
6	5		5		5				11

Format

MTC1 rt, fs

Description

$FPR[fs] \leftarrow GPR[rt]$

Stores the lower 32 bits of GPR[rt] in a floating-point register FPR[fs].

Exceptions

Coprocessor unusable

Operation

$FPR[fs] \leftarrow GPR[rt]_{31..0}$

MUL.S : Floating Point Multiply

MIPS I

To multiply single-precision floating-point values.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
COP1	S	ft	fs	fd	MUL						
010001	10000				000010						
6	5	5	5	5	6						

Format

MUL.S fd, fs, ft

Description

$FPR[fd] \leftarrow FPR[fs] \times FPR[ft]$

Multiplies the value of FPR[fs] by the value of FPR[ft] and stores the product in FPR[fd]. When an exponent overflow occurs, Flag O and Flag SO are set to 1 with the value of +maximum or -maximum as a result. When an exponent underflow occurs, Flag U and Flag SU are set to 1 with the value of +0 or -0 as a result.

Exceptions

Coprocessor unusable. Floating-point exceptions such as invalid operation, inexact, overflow and underflow are not generated by this instruction.

Operation

$FPR[fd] \leftarrow FPR[fs] \times FPR[ft]$

Flag O \leftarrow 1 if exponent overflows.

Flag U \leftarrow 1 if exponent underflows.

Flag SO \leftarrow 1 if exponent overflows.

Flag SU \leftarrow 1 if exponent underflows.

MULA.S : Floating Point Multiply to Accumulator

EE Core

To multiply single-precision floating-point values and store in Accumulator.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
COP1	S		ft		fs		0		MULA		
010001	10000						00000		011010		
6	5		5		5		5		6		

Format

MULA.S fs, ft

Description

$$ACC \leftarrow FPR[fs] \times FPR[ft]$$

Multiplies the value of FPR[fs] by the value of FPR[ft] as single-precision floating-point values and stores the product in the ACC register. When an exponent overflow occurs, Flag O and Flag SO are set to 1 and +maximum or -maximum value is stored in the ACC register as the result. When an exponent underflow occurs, Flag U and Flag SU are set to 1 and the value of +0 or -0 is stored in the ACC register as the result.

Exceptions

Coprocessor unusable. Floating-point exceptions such as invalid operation, inexact, overflow and underflow are not generated by this instruction.

Operation

$ACC \leftarrow FPR[fs] \times FPR[ft]$
 Flag O \leftarrow 1 if exponent overflows.
 Flag U \leftarrow 1 if exponent underflows.
 Flag SO \leftarrow 1 if exponent overflows.
 Flag SU \leftarrow 1 if exponent underflows.

NEG.S : Floating Point Negate

MIPS I

To negate a floating-point value.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
COP1 010001		S 10000		0 00000		fs		fd		NEG 000111	
6		5		5		5		5		6	

Format

NEG.S fd, fs

Description

$FPR[fd] \leftarrow -FPR[fs]$

Stores the value which is obtained by reversing the sign bit of FPR[fs] in FPR[fd]. Flag O and Flag U are cleared to zero.

Exceptions

Coprocessor unusable

Operation

$FPR[fd] \leftarrow -FPR[fs]$

Flag O $\leftarrow 0$

Flag U $\leftarrow 0$

RSQRT.S : Floating Point Reciprocal Square Root

MIPS IV

To obtain the reciprocal of the square root of a single-precision floating-point value.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
COP1	S	ft	fs	fd	RSQRT						
010001	10000				010110						
6	5	5	5	5	6						

Format

RSQRT.S fd, fs, ft

Description

$FPR[fd] \leftarrow FPR[fs] / \text{SQRT}(FPR[ft])$

Divides the value of FPR[fs] by the square root of FPR[ft] and stores the result in FPR[fd]. Values are handled as single-precision floating-point. When the value of FPR[ft] is 0, Flag D and Flag SD are set to 1 and +maximum or -maximum is stored in FPR[fd] as the result. When the value of FPR[ft] is negative, Flag I and Flag SI are set to 1 and the value of FPR[fs] divided by $\text{SQRT}(\text{ABS}(FPR[ft]))$ is stored in FPR[fd] as the result. When an exponent overflow occurs, the result is +maximum or -maximum. When an exponent underflow occurs, the result is +0 or -0.

Exceptions

Coprocessor unusable. Floating-point exceptions, such as invalid operation or inexact are not generated.

Operation

$FPR[fd] \leftarrow FPR[fs] / \text{SQRT}(FPR[ft])$

Flag I $\leftarrow 1$ if $FPR[ft] < 0$

Flag D $\leftarrow 1$ if $FPR[ft] = 0$

Flag SI $\leftarrow 1$ if $FPR[ft] < 0$

Flag SD $\leftarrow 1$ if $FPR[ft] = 0$

SQRT.S : Floating Point Square Root

MIPS II

To obtain the square root of a single-precision floating-point value.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
COP1		S		ft		0		fd		SQRT	
010001		10000				00000				000100	
6		5		5		5		5		6	

Format

SQRT.S fd, ft

Description

$FPR[fd] \leftarrow SQRT(FPR[ft])$

Calculates the single-precision floating-point square root value of FPR[ft] and stores the result in FPR[fd]. If the value of FPR[ft] is -0, the result is -0.

If the value of FPR[ft] is less than 0, Flag I and Flag SI are set to 1 and the value of $SQRT(ABS(FPR[ft]))$ is stored in FPR[fd] as the result.

Exceptions

Coprocessor unusable. No floating-point exceptions, such as invalid operation, inexact and so on.

Operation

$FPR[fd] \leftarrow SQRT(FPR[ft])$

Flag I $\leftarrow 1$ if $(FPR[ft] < 0)$

Flag D $\leftarrow 0$

Flag SI $\leftarrow 1$ if $(FPR[ft] < 0)$

SUB.S : Floating Point Subtract

MIPS I

To subtract single-precision floating-point values.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
COP1	S		ft		fs		fd			SUB	
010001	10000									000001	
6	5		5		5		5			6	

Format

SUB.S fd, fs, ft

Description

$FPR[fd] \leftarrow FPR[fs] - FPR[ft]$

Subtracts the value of FPR[ft] from the value of FPR[fs] and stores the result in FPR[fd]. Values are handled as single-precision floating-point. When an exponent overflow occurs, Flag O and Flag SO are set to 1 and +maximum or -maximum is stored in FPR[fd] as the result. When an exponent underflow occurs, Flag U and Flag SU are set to 1 and +0 or -0 is stored in FPR[fd] as the result.

Exceptions

Coprocessor unusable. Floating-point exceptions such as invalid operation, inexact, overflow and underflow are not generated by this instruction.

Operation

$FPR[fd] \leftarrow FPR[fs] - FPR[ft]$

Flag O \leftarrow 1 if exponent overflows.

Flag U \leftarrow 1 if exponent underflows.

Flag SO \leftarrow 1 if exponent overflows.

Flag SU \leftarrow 1 if exponent underflows.

SUBA.S : Floating Point Subtract to Accumulator

EE Core

To subtract single-precision floating-point values and store in Accumulator.

Operation Code

31	26	25	21	20	16	15	11	10	6	5	0
COP1						S		ft		fs	
010001						10000				0	
010001						00000		SUBA		011001	
6						5		5		5	
										6	

Format

SUBA.S fs, ft I

Description

$ACC \leftarrow FPR[fs] - FPR[ft]$

Subtracts the value of FPR[ft] from the value of FPR[fs] and stores the result in the ACC register. Values are handled as single-precision floating-point. When an exponent overflow occurs, Flag O and Flag SO are set to 1 and +maximum or -maximum is stored in the ACC register as the result. When an exponent underflow occurs, Flag U and Flag SU are set to 1 and +0 or -0 is stored in the ACC register as the result.

Exceptions

Coprocessor unusable. Floating-point exceptions such as invalid operation, inexact, overflow and underflow are not generated by this instruction.

Operation

$ACC \leftarrow FPR[fs] - FPR[ft]$

Flag O \leftarrow 1 if exponent overflows.

Flag U \leftarrow 1 if exponent underflows.

Flag SO \leftarrow 1 if exponent overflows.

Flag SU \leftarrow 1 if exponent underflows.

SWC1 : Store Word from Floating Point

MIPS I

To store the contents of a floating-point register in memory.

Operation Code

31	26	25	21	20	16	15	0
SWC1	base	ft	offset				
111001							
6	5	5	16				

Format

SWC1 ft, offset(base)

Description

$\text{memory}[\text{GPR}[\text{base}] + \text{offset}] \leftarrow \text{FPR}[\text{ft}]$

Adds the 16-bit signed offset to the value of GPR[base] and stores the contents of FPR[ft] in memory at the obtained effective address.

Restrictions

The effective address must comply with word alignment. Otherwise, an address error exception occurs if the lower 2 bits of effective address are not 0.

Exceptions

Coprocessor unusable
 TLB Refill
 TLB Invalid
 Address error

Operation (128-bit bus)

```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr1:0 ≠ 02 then
  SignalException(AddressError)
endif
(pAddr, uncached) ← AddressTranslation(vAddr, DATA, LOAD)
byte ← vAddr3:0
dataquad ← 096-8*byte || FPR[ft] || 08*byte
StoreMemory(uncached, WORD, dataquad, pAddr, vAddr, DATA)
  
```

(This page is left blank intentionally)

6. Appendix Instruction Set List

The instructions of the EE Core, COP0, COP1 (FPU) and COP2 (VU macro instructions) are classified according to the following functions. The chapter number in this book where each instruction is described is shown in the table. However, for the VU macro instructions indicated as VPU0, see a supplementary volume "VU User's Manual".

- Computational Instructions

- Integer add / subtraction and Floating point add / subtraction

- Integer multiplication / division and Floating point multiplication / division

- Integer multiply-add and Floating point multiply-add

- Shift operation / Logical operation / Comparison operation

- Maximum / Minimum value

- Data Format Conversion

- Exchange

- Random Numbers

- Other Operations (e.g. absolute value, square-root)

- Data transfer instructions

- Instructions for transferring between registers

- Load / Store

- Data transfer with special registers

- Program Control Instructions

- Conditional Branch / Jump

- Subroutine Call

- Break / Trap

- Other Instructions

6.1. Computational Instructions

6.1.1. Integer Addition and Subtraction

Inst.	Description	Ref.
ADD	Add	2
ADDI	Add Immediate	2
ADDIU	Add Immediate Unsigned	2
ADDU	Add Unsigned	2
DADD	Doubleword Add	2
DADDI	Doubleword Add Immediate	2
DADDIU	Doubleword Add Immediate (Unsigned)	2
DADDU	Doubleword Add Unsigned	2
DSUB	Doubleword Subtract	2
DSUBU	Doubleword Subtract Unsigned	2
SUB	Subtract	2
SUBU	Subtract Unsigned	2
PADDB	Parallel Add Byte	3
PADDH	Parallel Add Halfword	3
PADDSB	Parallel Add with Signed Saturation Byte	3
PADDSH	Parallel Add with Signed Saturation Halfword	3
PADDSW	Parallel Add with Signed Saturation Word	3
PADDUB	Parallel Add with Unsigned Saturation Byte	3
PADDUH	Parallel Add with Unsigned Saturation Halfword	3
PADDUW	Parallel Add with Unsigned Saturation Word	3
PADDW	Parallel Add Word	3
PADSBH	Parallel Add/Subtract Halfword	3
PSUBB	Parallel Subtract Byte	3
PSUBH	Parallel Subtract Halfword	3
PSUBSB	Parallel Subtract with Signed Saturation Byte	3
PSUBSH	Parallel Subtract with Signed Saturation Halfword	3
PSUBSW	Parallel Subtract with Signed Saturation Word	3
PSUBUB	Parallel Subtract with Unsigned Saturation Byte	3
PSUBUH	Parallel Subtract with Unsigned Saturation Halfword	3
PSUBUW	Parallel Subtract with Unsigned Saturation Word	3
PSUBW	Parallel Subtract Word	3
VIADD	Integer Add	VPU0
VIADDI	Integer Add Immediate	VPU0
VISUB	Integer Subtract	VPU0

6.1.2. Floating Point Addition and Subtraction

Inst.	Description	Ref.
ADD.S	Single Floating Point Add	5
ADDA.S	Single Floating Point Add to Accumulator	5
SUB.S	Single Floating Point Subtract	5
SUBA.S	Single Floating Point Subtract to Accumulator	5
VADD	Addition	VPU0
VADDA	ADD output to ACC	VPU0
VADDAbc	ADD broadcast bc field	VPU0
VADDAi	ADD output to ACC broadcast I register	VPU0
VADDAq	ADD output to ACC broadcast Q register	VPU0
VADDbc	ADD output to ACC broadcast bc field	VPU0
VADDi	ADD broadcast I register	VPU0
VADDq	ADD broadcast Q register	VPU0
VSUB	Subtraction	VPU0
VSUBA	SUB output ACC	VPU0
VSUBAbc	SUB output to ACC broadcast bc field	VPU0
VSUBAi	SUB output to ACC broadcast I register	VPU0
VSUBAq	SUB output to ACC broadcast Q register	VPU0
VSUBbc	SUB broadcast bc field	VPU0
VSUBi	SUB broadcast I register	VPU0
VSUBq	SUB broadcast Q register	VPU0

6.1.3. Integer Multiplication and Division

Inst.	Description	Ref.
DIV	Divide	2
DIVU	Divide Unsigned	2
MULT	Multiply	2
MULTU	Multiply Unsigned	2
DIV1	Divide 1	3
DIVU1	Divide Unsigned 1	3
MULT	Multiply (3-operand)	3
MULT1	Multiply 1	3
MULTU	Multiply Unsigned (3-operand)	3
MULTU1	Multiply unsigned 1	3
PDIVBW	Parallel Divide Broadcast Word	3
PDIVUW	Parallel Divide Unsigned Word	3
PDIVW	Parallel Divide Word	3
PMULTH	Parallel Multiply Halfword	3
PMULTUW	Parallel Multiply Unsigned Word	3
PMULTW	Parallel Multiply Word	3

6.1.4. Floating Point Multiplication and Division

Inst.	Description	Ref.
DIV.S	Single Floating Point Divide	5
MUL.S	Single Floating Point Multiply	5
MULA.S	Single Floating Point Multiply to Accumulator	5
VDIV	Floating Divide	VPU0

Inst.	Description	Ref.
VMUL	Multiply	VPU0
VMULA	MUL output to ACC	VPU0
VMULAbc	MUL output to ACC broadcast bc field	VPU0
VMULAi	MUL output to ACC broadcast I register	VPU0
VMULAq	MUL output to ACC broadcast Q register	VPU0
VMULbc	MUL broadcast bc field	VPU0
VMULi	MUL broadcast I register	VPU0
VMULq	MUL broadcast Q register	VPU0

6.1.5. Integer Multiply-Add

Inst.	Description	Ref.
MADD	Multiply / Add	3
MADD1	Multiply / Add 1	3
MADDU	Multiply / Add Unsigned	3
MADDU1	Multiply / Add Unsigned 1	3
PHMADH	Parallel Horizontal Multiply / Add Halfword	3
PHMSBH	Parallel Horizontal Multiply / Subtract Halfword	3
PMADDH	Parallel Multiply / Add Halfword	3
PMADDUW	Parallel Multiply / Add Unsigned Word	3
PMADDW	Parallel Multiply / Add Word	3
PMSUBH	Parallel Multiply / Subtract Halfword	3
PMSUBW	Parallel Multiply / Subtract Word	3

6.1.6. Floating Point Multiply-Add

Inst.	Description	Ref.
MADD.S	Single Floating Point Multiply and Add	5
MADDA.S	Single Floating Point Multiply and Add to Accumulator	5
MSUB.S	Single Floating Point Multiply and Subtract	5
MSUBA.S	Single Floating Point Multiply and Subtract from Accumulator	5
VMADD	MUL and ADD (SUB)	VPU0
VMADDA	MUL and ADD (SUB) output to ACC	VPU0
VMADDAbc	MUL and ADD (SUB) output to ACC broadcast bc field	VPU0
VMADDAi	MUL and ADD (SUB) output to ACC broadcast I register	VPU0
VMADDAq	MUL and ADD (SUB) output to ACC broadcast Q register	VPU0
VMADDbc	MUL and ADD (SUB) broadcast bc field	VPU0
VMADDi	MUL and ADD (SUB) broadcast I register	VPU0
VMADDq	MUL and ADD (SUB) broadcast Q register	VPU0
VMSUB	Multiply and SUB	VPU0
VMSUBA	Multiply and SUB output to ACC	VPU0
VMSUBAbc	Multiply and SUB output to ACC bc field	VPU0
VMSUBAi	Multiply and SUB output to ACC I register	VPU0
VMSUBAq	Multiply and SUB output to ACC Q register	VPU0
VMSUBbc	Multiply and SUB broadcast bc field	VPU0
VMSUBi	Multiply and SUB broadcast I register	VPU0
VMSUBq	Multiply and SUB broadcast Q register	VPU0

6.1.7. Shift Operation

Inst.	Description	Ref.
DSRA	Doubleword Shift Right Arithmetic	2
DSLL	Doubleword Shift Left Logical	2
DSLL32	Doubleword Shift Left Logical + 32	2
DSLLV	Doubleword Shift Left Logical Variable	2
DSRA32	Doubleword Shift Right Arithmetic + 32	2
DSRAV	Doubleword Shift Right Arithmetic	2
DSRL	Doubleword Shift Right Logical	2
DSRL32	Doubleword Shift Right Logical + 32	2
DSRLV	Doubleword Shift Right Logical Variable	2
SLL	Shift Left Logical	2
SLLV	Shift Left Logical Variable	2
SRA	Shift Right Arithmetic	2
SRAV	Shift Right Arithmetic Variable	2
SRL	Shift Right Logical	2
SRLV	Shift Right Logical Variable	2
PSLLH	Parallel Shift Left Logical Halfword	3
PSLLVW	Parallel Shift Left Logical Variable Word	3
PSLLW	Parallel Shift Left Logical Word	3
PSRAH	Parallel Shift Right Arithmetic Halfword	3
PSRAVW	Parallel Shift Right Arithmetic Variable Word	3
PSRAW	Parallel Shift Right Arithmetic Word	3
PSRLH	Parallel Shift Right Logical Halfword	3
PSRLVW	Parallel Shift Right Logical Variable Word	3
PSRLW	Parallel Shift Right Logical Word	3
QFSRV	Quadword Funnel Shift Right Variable	3

6.1.8. Logical Operation

Inst.	Description	Ref.
AND	AND	2
ANDI	AND Immediate	2
NOR	NOR	2
OR	OR	2
ORI	OR Immediate	2
XOR	Exclusive OR	2
XORI	Exclusive OR Immediate	2
PAND	Parallel AND	3
PNOR	Parallel NOR	3
POR	Parallel OR	3
PXOR	Parallel XOR	3
VIAND	Integer AND	VPU0
VIOR	Integer OR	VPU0

6.1.9. Comparison Operation

Inst.	Description	Ref.
SLT	Set on Less Than	2
SLTI	Set on Less Than on Immediate	2
SLTIU	Set on Less Than on Immediate Unsigned	2
SLTU	Set on Less Than Unsigned	2
PCEQB	Parallel Compare for Equal Byte	3
PCEQH	Parallel Compare for Equal Halfword	3
PCEQW	Parallel Compare for Equal Word	3
PCGTB	Parallel Compare for Greater Than Byte	3
PCGTH	Parallel Compare for Greater Than Halfword	3
PCGTW	Parallel Compare for Greater Than Word	3
C.EQ.S	Single Floating Point Compare	5
C.F.S	Single Floating Point Compare	5
C.LE.S	Single Floating Point Compare	5
C.LT.S	Single Floating Point Compare	5
VCLIP	Clipping	VPU0

6.1.10. Maximum / Minimum Value

Inst.	Description	Ref.
PMAXH	Parallel Maximum Halfword	3
PMAXW	Parallel Maximum Word	3
PMINH	Parallel Minimum Halfword	3
PMINW	Parallel Minimum Word	3
MAX.S	Single Floating Point Maximum	5
MIN.S	Single Floating Point Minimum	5
VMAX	Maximum	VPU0
VMAXbc	Maximum broadcast bc field	VPU0
VMAXi	Maximum broadcast I register	VPU0
VMINI	Minimum	VPU0
VMINIbc	Minimum broadcast bc field	VPU0
VMINIi	Minimum broadcast I register	VPU0

6.1.11. Data Format Conversion

Inst.	Description	Ref.
PEXT5	Parallel Extend from 5 bits	3
PPAC5	Parallel Pack to 5 bits	3
CVT.S.W	32-bit Fixed Point Floating Point Convert to Single Floating Point	5
CVT.W.S	Single Floating Point Convert to 32-bit Fixed Point	5
VFTOI0	Float to Integer, fixed point 0 bit	VPU0
VFTOI12	Float to Integer, fixed point 12 bits	VPU0
VFTOI15	Float to Integer, fixed point 15 bits	VPU0
VFTOI4	Float to Integer, fixed point 4 bits	VPU0
VITOF0	Integer to Float, fixed point 0 bit	VPU0
VITOF12	Integer to Float, fixed point 12 bits	VPU0
VITOF15	Integer to Float, fixed point 15 bits	VPU0
VITOF4	Integer to Float, fixed point 4 bits	VPU0
VMR32	Rotate right 32 bits	VPU0

6.1.12. Exchange

Inst.	Description	Ref.
PCPYH	Parallel Copy Halfword	3
PCPYLD	Parallel Copy Lower Doubleword	3
PCPYUD	Parallel Copy Upper Doubleword	3
PEXCH	Parallel Exchange Center Halfword	3
PEXCW	Parallel Exchange Center Word	3
PEXEH	Parallel Exchange Even Halfword	3
PEXEW	Parallel Exchange Even Word	3
PEXTLB	Parallel Extend Lower From Byte	3
PEXTLH	Parallel Extend Lower From Halfword	3
PEXTLW	Parallel Extend Lower From Word	3
PEXTUB	Parallel Extend Upper From Byte	3
PEXTUH	Parallel Extend Upper From Halfword	3
PEXTUW	Parallel Extend Upper From Word	3
PINTEH	Parallel Interleave Even Halfword	3
PINTH	Parallel Interleave Halfword	3
PPACB	Parallel Pack To Byte	3
PPACH	Parallel Pack To Halfword	3
PPACW	Parallel Pack To Word	3
PREVH	Parallel Reverse Halfword	3
PROT3W	Parallel Rotate 3 Word	3

6.1.13. Random Number

Inst.	Description	Ref.
VRGET	Random-unit get R register	VPU0
VRINIT	Random-unit init R register	VPU0
VRNEXT	Random-unit next M sequence	VPU0
VRXOR	Random-unit XOR R register	VPU0

6.1.14. Other Operations

Inst.	Description	Ref.
PABSH	Parallel Absolute Halfword	3
PABSW	Parallel Absolute Word	3
PLZCW	Parallel Leading Zero Count Word	3
ABS.S	Single Floating Point Absolute	5
NEG.S	Single Floating Point Negate	5
RSQRT.S	Single Floating Point Reciprocal Square Root	5
SQRT.S	Single Floating Point Square Root	5
VABS	Absolute	VPU0
VOPMSUB	Outer product MSUB	VPU0
VOPMULA	Outer product MULA	VPU0
VRSQRT	Floating reciprocal Square-root	VPU0
VSQRT	Floating reciprocal Square	VPU0

6.2. Data Transfer Instructions

6.2.1. Instructions for Transferring between Registers

Inst.	Description	Ref.
MFHI	Move From HI	2
MFLO	Move From LO	2
MOVN	Move on Register Not Equal to Zero	2
MOVZ	Move on Register Equal to Zero	2
MTHI	Move To HI	2
MTLO	Move To LO	2
MFHI1	Move From HI1	3
MFLO1	Move From LO1	3
MTHI1	Move To HI1	3
MTLO1	Move To LO1	3
PMFHI	Parallel Move From HI	3
PMFHL	Parallel Move From HI / LO	3
PMFLO	Parallel Move From LO	3
PMTHI	Parallel Move To HI	3
PMTHL	Parallel Move To HI / LO	3
PMTLO	Parallel Move To LO	3
MFC1	Move Word from FPR	5
MOV.S	Single Floating Point Move	5
MTC1	Move Word to FCR	5
CFC2	Move Control From COP2	VPU0
CTC2	Move Control To COP2	VPU0
LQC2	Load Quadword to COP2	VPU0
QMFC2	Quadword Move From COP2	VPU0
QMTTC2	Quadword Move To COP2	VPU0
SQC2	Store Quadword from COP2	VPU0
VMFIR	Move From integer register	VPU0
VMOVE	Move Floating register	VPU0
VMTIR	Move To integer register	VPU0

6.2.2. Load

Inst.	Description	Ref.
LB	Load Byte	2
LBU	Load Byte Unsigned	2
LD	Load Doubleword	2
LDL	Load Doubleword Left	2
LDR	Load Doubleword Right	2
LH	Load Halfword	2
LHU	Load Halfword Unsigned	2
LUI	Load Upper Immediate	2
LW	Load Word	2
LWL	Load Word Left	2
LWR	Load Word Right	2
LWU	Load Word Right Unsigned	2
LQ	Load Quadword	3
LWC1	Load Word to FPR	5
VILWR	Integer load word register	VPU0
VLQD	Load Quadword with pre-decrement	VPU0
VLQI	Load Quadword with post-increment	VPU0

6.2.3. Store

Inst.	Description	Ref.
SB	Store Byte	2
SD	Store Doubleword	2
SDL	Store Doubleword Left	2
SDR	Store Doubleword Right	2
SH	Store Halfword	2
SW	Store Word	2
SWL	Store Word Left	2
SWR	Store Word Right	2
SQ	Store Quadword	3
SWC1	Store Word from FPR	5
VISWR	Integer store word register	VPU0
VSQD	Store Quadword with pre-decrement	VPU0
VSQI	Store Quadword with post-increment	VPU0

6.2.4. Special Data Transfer

Inst.	Description	Ref.
MFSA	Move From SA Register	3
MTSA	Move To SA Register	3
MTSAB	Move Byte Count to SA Register	3
MTSAH	Move Halfword Count to SA Register	3
MFBPC	Move From Breakpoint Control	4
MFC0	Move From COP0	4
MFDAB	Move From Data Address Breakpoint	4
MFDABM	Move From Data Address Breakpoint Mask	4
MFDVB	Move From Data Value Breakpoint	4
MFDVBM	Move From Data Value Breakpoint Mask	4
MFIAB	Move From Instruction Address Breakpoint	4
MFIABM	Move From Instruction Address Breakpoint Mask	4
MFPC	Move From Performance Counter	4
MFPS	Move From Performance Event Specifier	4
MTBPC	Move To Breakpoint Control	4
MTC0	Move To COP0	4
MTDAB	Move To Data Address Breakpoint	4
MTDABM	Move To Data Address Breakpoint Mask	4
MTDVB	Move To Data Value Breakpoint	4
MTDVBM	Move To Data Value Breakpoint Mask	4
MTIAB	Move To Instruction Address Breakpoint	4
MTIABM	Move To Instruction Address Breakpoint Mask	4
MTPC	Move From Performance Counter	4
MTPS	Move From Performance Event Specifier	4
CFC1	Move Control Word from FCR	5
CTC1	Move Control Word to FCR	5

6.3. Program Control Instructions

6.3.1. Conditional Branch

Inst.	Description	Ref.
BEQ	Branch on Equal	2
BEQL	Branch on Equal Likely	2
BGEZ	Branch on Greater Than or Equal to Zero	2
BGEZL	Branch on Greater Than or Equal to Zero And Link	2
BGTZ	Branch on Greater Than Zero	2
BGTZL	Branch on Greater Than Zero Likely	2
BLEZ	Branch on Less Than or Equal to Zero	2
BLEZL	Branch on Less Than or Equal to Zero Likely	2
BLTZ	Branch on Less Than Zero	2
BLTZL	Branch on Less Than Zero Likely	2
BNE	Branch on Not Equal	2
BNEL	Branch on Not Equal Likely	2
BC0F	Branch on COP0 False	4
BC0FL	Branch on COP0 False Likely	4
BC0T	Branch on COP0 True	4
BC0TL	Branch on COP0 True Likely	4
BC1F	Branch on FPU False	5
BC1FL	Branch on FPU False Likely	5
BC1T	Branch on FPU True	5
BC1TL	Branch on FPU True Likely	5
BC2F	Branch on COP2 False	VPU0
BC2FL	Branch on COP2 False Likely	VPU0
BC2T	Branch on COP2 True	VPU0
BC2TL	Branch on COP2 True Likely	VPU0

6.3.2. Jump

Inst.	Description	Ref.
J	Jump	2
JR	Jump Register	2

6.3.3. Subroutine Call

Inst.	Description	Ref.
BGEZAL	Branch on Greater Than or Equal to Zero And Link	2
BGEZALL	Branch on Greater Than or Equal to Zero And Link Likely	2
BLTZAL	Branch on Less Than Zero and Link	2
BLTZALL	Branch on Less Than Zero And Link Likely	2
JAL	Jump And Link	2
JALR	Jump And Link Register	2
VCALLMS	Call micro sub-routine	VPU0
VCALLMSR	Call micro sub-routine register	VPU0

6.3.4. Break / Trap

Inst.	Description	Ref.
BREAK	Break	2
SYSCALL	System Call	2
TEQ	Trap if Equal	2
TEQI	Trap if Equal Immediate	2
TGE	Trap if Greater Than or Equal	2
TGEI	Trap if Greater Than or Equal Immediate	2
TGEIU	Trap if Greater Than or Equal Immediate Unsigned	2
TGEU	Trap if Greater Than or Equal Immediate Unsigned	2
TLT	Trap if Less Than	2
TLTI	Trap if Less Than Immediate	2
TLTIU	Trap if Less Than Immediate Unsigned	2
TLTU	Trap if Less Than Unsigned	2
TNE	Trap if Not Equal	2
TNEI	Trap if Not Equal Immediate	2
ERET	Exception Return	4

6.4. Other Instructions

Inst.	Description	Ref.
SYNC.stype	Synchronization	2
VWAITQ	wait Q register	VPU0
PREF	Prefetch	2
DI	Disable Interrupt	4
EI	Enable Interrupt	4
VNOP	no operation	VPU0

(This page is left blank intentionally)

7. Appendix OpCode Encoding

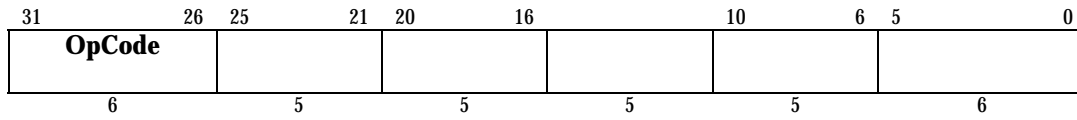
This section shows the instructions of the EE Core, COP0 and COP1 (FPU) according to the bit pattern of the instruction codes. The characters shown in a bold italic type are instruction classes and their details are shown in the attached tables.

Explanation of terms:

reserved	Undefined instruction. When executing, generates undefined instruction exception.
undefined	Undefined instruction. When executing, the operation is undefined.
unsupported	MIPS IV instructions that the EE Core does not support. If executing them, an undefined instruction exception occurs.
*	EE Core-specific instructions (in the table of the CPU instruction)

7.1. CPU Instructions

7.1.1. Instructions encoded by OpCode field

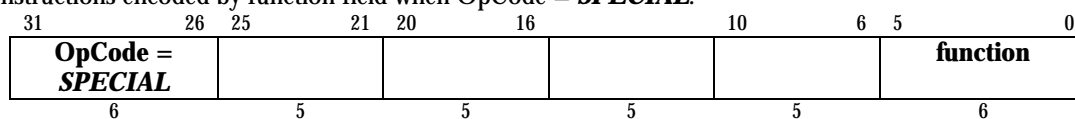


	Bits 28 - 26							
Bits 31 - 29	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111
0 000	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1 001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2 010	COP0	COP1	COP2	reserved	BEQL	BNEL	BLEZL	BGTZL
3 011	DADDI	DADDIU	LDL	LDR	MMI *	reserved	LQ *	SQ *
4 100	LB	LH	LWL	LW	LBU	LHU	LWR	LWU
5 101	SB	SH	SWL	SW	SDL	SDR	SWR	CACHE
6 110	unsupported	LWC1	unsupported	PREF	unsupported	unsupported	LQC2 **	LD
7 111	unsupported	SWC1	unsupported	reserved	unsupported	unsupported	SQC2 **	SD

** LQC2 and SQC2 are COP2 Instructions.

7.1.2. SPECIAL Instruction Class

Instructions encoded by function field when OpCode = **SPECIAL**.



	Bits 2 - 0							
Bits 5 - 3	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111
0 000	SLL	reserved	SRL	SRA	SLLV	reserved	SRLV	SRAV
1 001	JR	JALR	MOVZ	MOVN	SYSCALL	BREAK	reserved	SYNC
2 010	MFHI	MTHI	MFLO	MTLO	DSLLV	reserved	DSRLV	DSRAV
3 011	MULT	MULTU	DIV	DIVU	unsupported	unsupported	unsupported	unsupported
4 100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5 101	MFSA *	MTSA *	SLT	SLTU	DADD	DADDU	DSUB	DSUBU
6 110	TGE	TGEU	TLT	TLTU	TEQ	reserved	TNE	reserved
7 111	DSLL	reserved	DSRL	DSRA	DSLL32	reserved	DSRL32	DSRA32

7.1.3. REGIMM Instruction Class

Instructions encoded by rt field when OpCode = **REGIMM**.

31	26	25	21	20	16	10	6	5	0
OpCode = REGIMM						rt			
6						5	5	5	6

	Bits 18 - 16							
Bits 20 - 19	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 00	BLTZ	BGEZ	BLTZL	BGEZL	reserved	reserved	reserved	reserved
0 01	TGEI	TGEIU	TLTI	TLTIU	TEQI	reserved	TNEI	reserved
2 10	BLTZAL	BGEZAL	BLTZALL	BGEZALL	reserved	reserved	reserved	reserved
3 11	MTSAB *	MTSAH *	reserved	reserved	reserved	reserved	reserved	reserved

7.2. EE Core-Specific Instructions

7.2.1. MMI Instruction Class

Instructions encoded by function field when OpCode = **MMI**.

31	26	25	21	20	16	10	6	5	0
OpCode = MMI						function			
6						6			

Bits 5 - 3	Bits 2 - 0							
	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111
0 000	MADD	MADDU	reserved	reserved	PLZCW	reserved	reserved	reserved
1 001	MMI0	MMI2	reserved	reserved	reserved	reserved	reserved	reserved
2 010	MFHI1	MTHI1	MFLO1	MTLO1	reserved	reserved	reserved	reserved
3 011	MULT1	MULTU1	DIV1	DIVU1	reserved	reserved	reserved	reserved
4 100	MADD1	MADDU1	reserved	reserved	reserved	reserved	reserved	reserved
5 101	MMI1	MMI3	reserved	reserved	reserved	reserved	reserved	reserved
6 110	PMFHL	PMTHL	reserved	reserved	PSLLH	reserved	PSRLH	PSRAH
7 111	reserved	reserved	reserved	reserved	PSLLW	reserved	PSRLW	PSRAW

7.2.2. MMIO Instruction Class

Instructions encoded by function field when OpCode = **MMI** and bits 5..0 = **MMIO**

31	26	25	21	20	16	10	6	5	0
OpCode = MMI						function		MMIO	
6						5		6	

	Bits 7 - 6			
Bits 10 - 8	0	1	2	3
	00	01	10	11
0 000	PADDW	PSUBW	PCGTW	PMAW
1 001	PADDH	PSUBH	PCGTH	PMAH
2 010	PADDDB	PSUBDB	PCGTDB	reserved
3 011	reserved	reserved	reserved	reserved
4 100	PADDSW	PSUBSW	PEXTLW	PPACW
5 101	PADDSH	PSUBSH	PEXTLH	PPACH
6 110	PADDDB	PSUBDB	PEXTLB	PPACB
7 111	reserved	reserved	PEXT5	PPAC5

7.2.3. MMI1 Instruction Class

Instructions encoded by function field when OpCode = **MMI** and bits 5..0 = **MMI1**.

31	26	25	21	20	16	10	6	5	0
OpCode = MMI						function			
						MMI1			
6						6			

	Bits 7 - 6			
Bits 10 - 8	0	1	2	3
	00	01	10	11
0 000	reserved	PABSW	PCEQW	PMINW
1 001	PADSBH	PABSH	PCEQH	PMINH
2 010	reserved	reserved	PCEQB	reserved
3 011	reserved	reserved	reserved	reserved
4 100	PADDUW	PSUBUW	PEXTUW	reserved
5 101	PADDUH	PSUBUH	PEXTUH	reserved
6 110	PADDUB	PSUBUB	PEXTUB	QFSRV
7 111	reserved	reserved	reserved	reserved

7.2.4. MMI2 Instruction Class

Instructions encoded by function field when OpCode = **MMI** and bits 5..0 = **MMI2**.

31	26	25	21	20	16	10	6	5	0
OpCode = MMI						function		MMI2	
6						5		6	

	Bits 7 - 6			
Bits 10 - 8	0	1	2	3
	00	01	10	11
0 000	PMADDW	reserved	PSLLVW	PSRLVW
1 001	PMSUBW	reserved	reserved	reserved
2 010	PMFHI	PMFLO	PINTH	reserved
3 011	PMULTW	PDIVW	PCPYLD	reserved
4 100	PMADDH	PHMADH	PAND	PXOR
5 101	PMSUBH	PHMSBH	reserved	reserved
6 110	reserved	reserved	PEXEH	PREVH
7 111	PMULTH	PDIVBW	PEXEW	PROT3W

7.2.5. MMI3 Instruction Class

Instructions encoded by function field when OpCode = **MMI** and bits 5..0 = **MMI3**

31	26	25	21	20	16	10	6	5	0
OpCode = MMI						function		MMI3	
6						5		6	

	Bits 7 - 6			
Bits 10 - 8	0	1	2	3
	00	01	10	11
0 000	PMADDUW	reserved	reserved	PSRAVW
1 001	reserved	reserved	reserved	reserved
2 010	PMTHI	PMTLO	PINTEH	reserved
3 011	PMULTUW	PDIVUW	PCPYUD	reserved
4 100	reserved	reserved	POR	PNOR
5 101	reserved	reserved	reserved	reserved
6 110	reserved	reserved	PEXCH	PCPYH
7 111	reserved	reserved	PEXCW	reserved

7.3. COP0 Instructions

7.3.1. COP0 Instruction Class

Instructions encoded by rs field when OpCode = **COP0**

31	26	25	21	20	16	10	6	5	0
OpCode = COP0						rs			
6						5		5	

Bits 23 - 21								
Bits 25 - 24	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111
0 00	MF0	reserved	reserved	reserved	MT0	reserved	reserved	reserved
1 01	BC0	reserved	reserved	reserved	reserved	reserved	reserved	reserved
2 10	CO	reserved	reserved	reserved	reserved	reserved	reserved	reserved
3 11	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved

7.3.2. BC0 Instruction Class

Instructions encoded by rt field when OpCode field = **COP0** and rs field = **BC0**

31	26	25	21	20	16	10	6	5	0
OpCode = COP0						rs = BC0		rt	
6						5		5	

Bits 18 - 16								
Bits 20 - 19	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111
0 00	BC0F	BC0T	BC0FL	BC0TL	reserved	reserved	reserved	reserved
1 01	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved
2 10	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved
3 11	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved

7.3.3. C0 Instruction Class

Instructions encoded by function field when OpCode = **COP0** and rs = **C0**

31	26	25	21	20	16	10	6	5	0
OpCode = COP0						rs = C0			
						function			
6						6			

	Bits 2 - 0							
Bits 5 - 3	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111
0 000	undefined	TLBR	TLBWI	undefined	undefined	undefined	TLBWR	undefined
1 001	TLBP	undefined	undefined	undefined	undefined	undefined	undefined	undefined
2 010	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined
3 011	ERET	undefined	undefined	undefined	undefined	undefined	undefined	undefined
4 100	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined
5 101	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined
6 110	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined
7 111	EI	DI	undefined	undefined	undefined	undefined	undefined	undefined

7.4. COP1 Instructions

7.4.1. COP1 Instruction Class

Instructions encoded by rs field when OpCode = **COP1**.

31	26	25	21	20	16	10	6	5	0
OpCode = COP0						rs			
6						5		5	

	Bits 23 - 21							
Bits 25 - 24	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111
0 00	MFC1	reserved	CFC1	reserved	MTC1	reserved	CTC1	reserved
1 01	BC1	reserved	reserved	reserved	reserved	reserved	reserved	reserved
2 10	<i>S</i>	reserved	reserved	reserved	<i>W</i>	reserved	reserved	reserved
3 11	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved

7.4.2. BC1 Instruction Class

Instructions encoded by rt field when OpCode field = **COP1** and rs = **BC1**.

31	26	25	21	20	16	10	6	5	0
OpCode = COP0						rs = BC1		rt	
6						5		5	

	Bits 18 - 16							
Bits 20 - 19	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111
0 00	BC1F	BC1T	BC1FL	BC1TL	reserved	reserved	reserved	reserved
1 01	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved
2 10	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved
3 11	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved

7.4.3. S Instruction Class

Instructions encoded by function field when OpCode = **COP1** and rs = **S**.

31	26	25	21	20	16	10	6	5	0
OpCode = COP0						rs = S			
						function			
6						6			

	Bits 2 - 0							
Bits 5 - 3	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111
0 000	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1 001	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined
2 010	undefined	undefined	undefined	undefined	undefined	undefined	RSQRT	undefined
3 011	ADDA	SUBA	MULA	undefined	MADD	MSUB	MADDA	MSUBA
4 100	undefined	undefined	undefined	undefined	CVTW	undefined	undefined	undefined
5 101	MAX	MIN	undefined	undefined	undefined	undefined	undefined	undefined
6 110	C.F	undefined	C.EQ	undefined	C.LT	undefined	C.LE	undefined
7 111	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined

7.4.4. W Instruction Class

Instructions encoded by function field when OpCode = **COP1** and rs = **W**.

31	26	25	21	20	16	10	6	5	0
OpCode = COP0						rs = W			
						function			
6						5			

	Bits 2~0							
Bits 5 - 3	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111
0 000	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined
1 001	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined
2 010	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined
3 011	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined
4 100	CVTS	undefined	undefined	undefined	undefined	undefined	undefined	undefined
5 101	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined
6 110	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined
7 111	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined