Summary of Cavium Networks-Specific cnMIPS Core Instructions

cnMIPS cores support the standard instruction set for Release 2 of the MIPS64 architecture as well as an expanded set of instructions. This document describes the latter instructions. It is intended for open source software developers who want to use the Cavium Networks' OCTEON Multi-Core MIPS64 processors. For details on the standard MIPS64 Release 2 instructions and programming please refer to documentation and resources published by MIPS Inc. (www.MIPS.com).

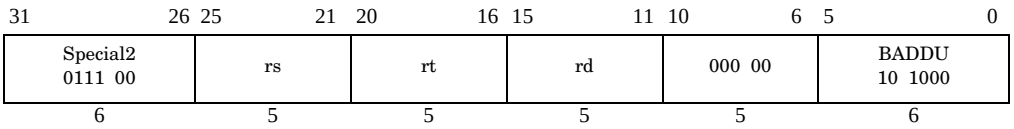
This document covers:

- List of Cavium Networks-Specific Core Instructions
- Description of Cavium Networks-Specific Instructions

1.1 List of Cavium Networks-Specific Core Instructions

1.2 Description of Cavium Networks-Specific Instructions

Unsigned Byte Add **BADDU**



Format: BADDU rd, rs, rt **CVM**

Purpose:
To do an unsigned byte (8-bit) add.

Description: The 8-bit byte value in GPR rt is added to the 8-bit byte value in GPR rs producing an 8-bit result that is zero- extended.

No integer overflow or any other exception occurs under any circumstance.

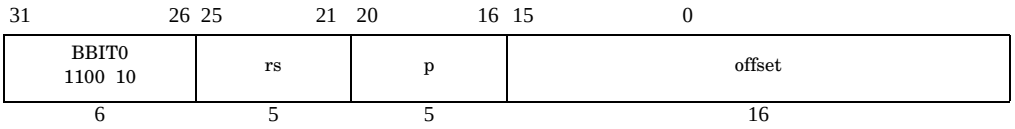
Operation:

```
GPR[rd] = (GPR[rs] + GPR[rt]) & 0xFF
```

Exceptions:

None.

Branch on Bit Clear BBIT0



Format: BBIT0 rs, p, offset **CVM**

Purpose:

To do a PC-relative conditional branch if one of the lower 32-bits is clear

Description: `if !rs<p> then branch`

An 18-bit signed offset (the 16-bit offset field shifted left 2-bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If bit rs<p> is clear, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

Processor operation is unpredictable if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

```
I:    target_offset <- sign_extend(offset || 00)
      condition <- !GPR[rs]<p>
I+1:  if condition then
         PC <- PC + target_offset
     endif
```

Exceptions:

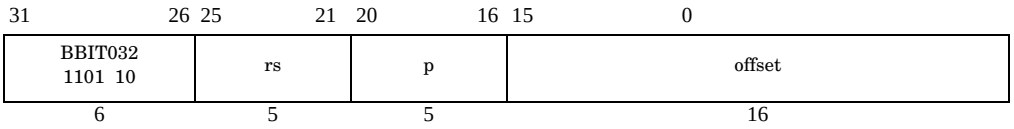
None.

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

BBIT0 consumes the major opcode of the MIPS LWC2 instruction. BBIT0 is not considered a COP2 instruction so does not take an exception when COP2 is not enabled.

Branch on Bit Clear Plus 32 — BBIT032



Format: BBIT032 rs, p, offset **CVM**

Purpose:

To do a PC-relative conditional branch if one of the upper 32 bits is clear

Description: `if !rs<p+32> then branch`

An 18-bit signed offset (the 16-bit offset field shifted left 2-bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If bit rs<p+32> is clear, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

Processor operation is unpredictable if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

```
I:    target_offset <- sign_extend(offset || 00)
      condition <- !GPR[rs]<p+32>
I+1:  if condition then
         PC <- PC + target_offset
     endif
```

Exceptions:

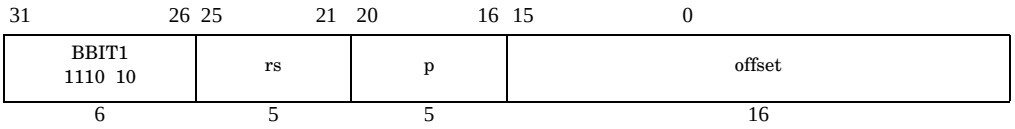
None.

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

BBIT032 consumes the major opcode of the MIPS LDC2 instruction. BBIT032 is not considered a COP2 instruction so does not take an exception when COP2 is not enabled.

Branch on Bit Set — BBIT1



Format: BBIT1 rs, p, offset **CVM**

Purpose:

To do a PC-relative conditional branch if one of the lower 32 bits is set

Description: `if rs<p> then branch`

An 18-bit signed offset (the 16-bit offset field shifted left 2-bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If-bit rs<p> is set, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

Processor operation is unpredictable if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

```
I:    target_offset <- sign_extend(offset || 00)
      condition <- GPR[rs]<p>
I+1:  if condition then
         PC <- PC + target_offset
     endif
```

Exceptions:

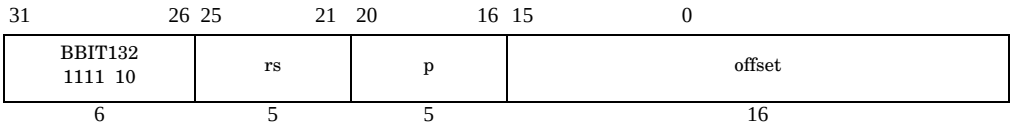
None.

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

BBIT1 consumes the major opcode of the MIPS SWC2 instruction. BBIT1 is not considered a COP2 instruction so does not take an exception when COP2 is not enabled.

Branch on Bit Set Plus 32 — BBIT132



Format: BBIT132 rs, p, offset **CVM**

Purpose:

To do a PC-relative conditional branch if one of the upper 32 bits is set

Description: `if rs<p+32> then branch`

An 18-bit signed offset (the 16-bit offset field shifted left 2-bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If bit rs<p+32> is set, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

Processor operation is unpredictable if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

```
I:    target_offset <- sign_extend(offset || 00)
      condition <- GPR[rs]<p+32>
I+1:  if condition then
         PC <- PC + target_offset
      endif
```

Exceptions:

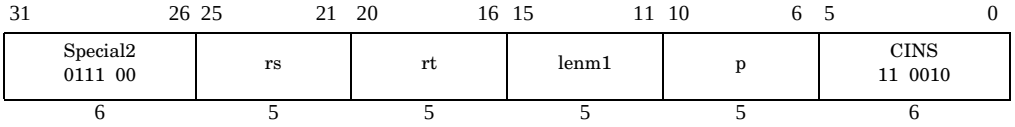
None.

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

BBIT132 consumes the major opcode of the MIPS SDC2 instruction. BBIT132 is not considered a COP2 instruction so does not take an exception when COP2 is not enabled.

Clear and Insert a Bit Field — CINS



Format: CINS rt, rs, p, lenm1 **CVM**

Purpose:

To insert a bit field that starts in the lower 32 bits of a register and clear the rest of the register.

Description: `rt = rs<lenm1:0> << p`
The contents of general-purpose register rt from bit location p + lenm1 to bit location p are filled with the contents of general-purpose register rs from bit location lenm1 to bit location 0. The remaining bits in rt are zeroed by this instruction.

Restrictions:

The p and lenm1 fields are only 5-bits, so the widest allowed bit field is 32-bits.

A Reserved Instruction Exception is signaled if access to 64-bit operations is not enabled.

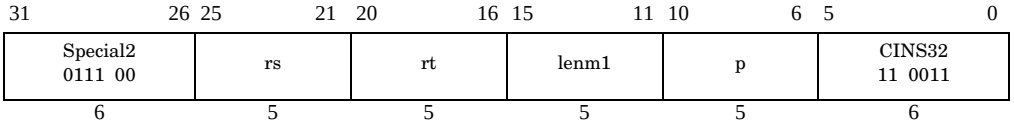
Operation:

```
if Are64bitOperationsEnabled() then
   GPR[rt] <- GPR[rs]<lenm1:0> << p
else
   SignalException(ReservedInstruction)
endif
```

Exceptions:

Reserved Instruction.

Clear and Insert a Bit Field Plus 32 — CINS32



Format: CINS32 rt, rs, p, lenm1 **CVM**

Purpose:

To insert a bit field that starts in the upper 32 bits of a register and clear the rest of the register.

Description: `rt = rs<lenm1:0> << (p + 32)`

The contents of general-purpose register rt from bit location p + 32 + lenm1 to bit location p + 32 are filled with the contents of general-purpose register rs from bit location lenm1 to bit location 0. The remaining bits in rt are zeroed by this instruction.

Restrictions:

The p and lenm1 fields are only 5-bits, so the widest allowed bit field is 32-bits.

The result register GPR[rt] is unpredictable when p+32+lenm1 is larger than 63.

A Reserved Instruction Exception is signaled if access to 64-bit operations is not enabled.

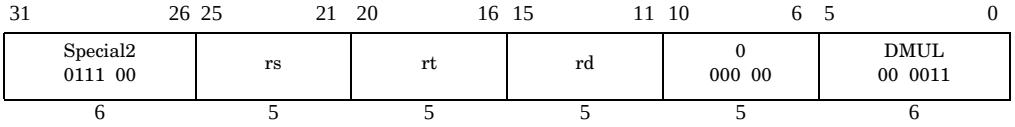
Operation:

```
if Are64bitOperationsEnabled() then
   GPR[rt] <- GPR[rs]<lenm1:0> << (p + 32)
else
   SignalException(ReservedInstruction)
endif
```

Exceptions:

Reserved Instruction.

Multiply Doubleword to GPR — DMUL



Format: DMUL rd, rs, rt **CVM**

Purpose:

To multiply 64-bit signed integers and write the result to a GPR.

Description: `rd = rs × rt`

The 64-bit double-word value in GPR rt is multiplied by the 64-bit value in GPR rs, treating both operands as signed values, to produce a 128-bit result. The low-order 64-bit double word of the result is written to GPR rd. The contents of HI, LO, P0, P1, and P2 are unpredictable after the operation.

No arithmetic exception occurs under any circumstances.

Restrictions:

A Reserved Instruction Exception is signaled if access to 64-bit operations is not enabled.

Operation:

```
if Are64bitOperationsEnabled() then
   prod = GPR[rs] * GPR[rt]
   GPR[rd] = prod<63:0>
else
   SignalException(ReservedInstruction)
endif
```

Exceptions:

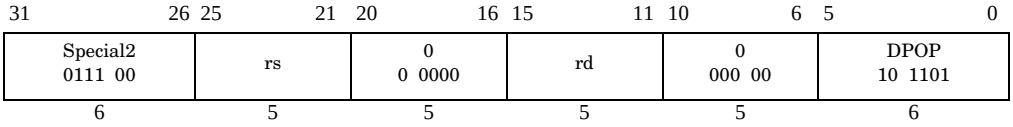
Reserved Instruction

Programming Notes:

The integer multiply operation proceeds asynchronously and other CPU instructions can execute before it is complete. An attempt to read GPR rd before the results are written interlocks until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

Programs that require overflow detection must check for it explicitly.

Count Ones in a Doubleword — DPOP



Format: DPOP rd, rs **CVM**

Purpose

Count the number of ones in a double word

Description: `rd = count_ones(rs)`

Bits 63..0 of GPR rs are scanned. The number of ones is counted and the result is written to GPR rd.

Restrictions:

A Reserved Instruction Exception is signaled if access to 64-bit operations is not enabled.

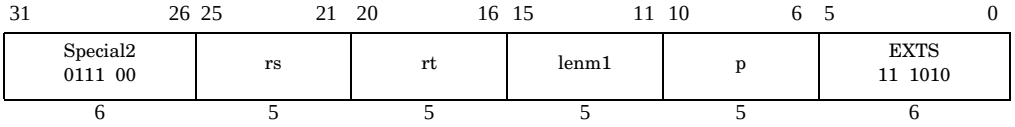
Operation:

```
if Are64bitOperationsEnabled() then
   temp = 0
   for i in 63 .. 0
      if GPR[rs]<i> = 1 then
         temp++
      endif
   endfor
   GPR[rd] = temp
else
   SignalException(ReservedInstruction)
endif
```

Exceptions:

Reserved Instruction.

Extract a Signed Bit Field — EXTS



Format: EXTS rt, rs, p, lenm1 **CVM**

Purpose:

To extract and sign-extend a bit field that starts from the lower 32 bits of a register.

Description: `rt = sign-extend(rs<p+lenm1:p>, lenm1)`

Bit locations p + lenm1 to p are extracted from rs and the result is written into the lowest bits of destination register rt. The remaining bits in rt are a sign-extension of the most-significant bit of the bit field (i.e. rt<63:lenm1> are all duplicates of the source-register bit rs<p+lenm1>).

Restrictions:

The p and lenm1 fields are only 5-bits, so the widest allowed bit field is 32-bits.

A Reserved Instruction Exception is signaled if access to 64-bit operations is not enabled.

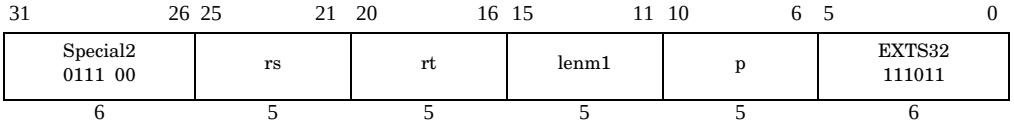
Operation:

```
if Are64bitOperationsEnabled() then
   GPR[rt] <- sign-extend(GPR[rs]<p+lenm1:p>, lenm1)
else
   SignalException(ReservedInstruction)
endif
```

Exceptions:

Reserved Instruction.

Extract a Signed Bit Field Plus 32 — EXTS32



Format: EXTS32 rt, rs, p, lenm1 **CVM**

Purpose:

To extract and sign-extend a bit field that starts from the upper 32 bits of a register.

Description: `rt = sign-extend(rs<p+32+lenm1:p+32>, lenm1)`

Bit locations p + 32 + lenm1 to p + 32 are extracted from rs and the result is written into the lowest bits of destination register rt. The remaining bits in rt are a sign-extension of the most-significant bit of the bit field (i.e. rt<63:lenm1> are all duplicates of the source-register bit rs<p+32+lenm1>).

Restrictions:

The p and lenm1 fields are only 5-bits, so the widest allowed bit field is 32-bits.

The result register GPR[rt] is unpredictable when p+32+lenm1 is larger than 63.

A Reserved Instruction Exception is signaled if access to 64-bit operations is not enabled.

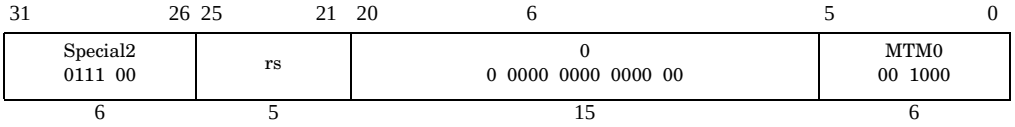
Operation:

```
if Are64bitOperationsEnabled() then
   GPR[rt] <- sign-extend(GPR[rs]<p+32+lenm1:p+32>, lenm1)
else
   SignalException(ReservedInstruction)
endif
```

Exceptions:

Reserved Instruction.

Load Multiplier Register MPL0 — MTM0



Format: MTM0 rs **CVM**

Purpose:

To load the multiplier register MPL0 (and zero P0, P1, P2).

Description: `MPL0 = rs; P0, P1, P2 = 0`

The 64-bit double-word value in GPR rs is loaded into the multiplier register MPL0 and P0-P2 are zeroed.

Restrictions:

A Reserved Instruction Exception is signaled if access to 64-bit operations is not enabled.

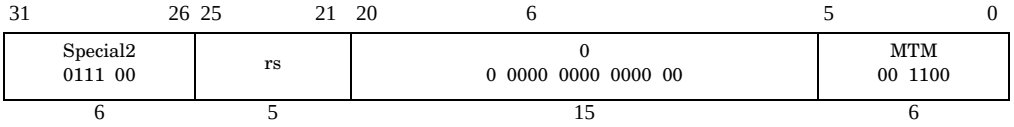
Operation:

```
if Are64bitOperationsEnabled() and !CvmCtl[NOMUL] then
   MPL0<63:0> = GPR[rs]
   P0 = 0
   P1 = 0
   P2 = 0
else
   SignalException(ReservedInstruction)
endif
```

Exceptions:

Reserved Instruction

Load Multiplier Register MPL1 — MTM1



Format: MTM1 rs **CVM**

Purpose:

To load the multiplier register MPL1 (and zero P0, P1, P2).

Description: `MPL1 = rs; P0, P1, P2 = 0`

The 64-bit doubleword value in GPR rs is loaded into the multiplier register MPL1 and P0-P2 are zeroed.

Restrictions:

A Reserved Instruction Exception is signaled if access to 64-bit operations is not enabled.

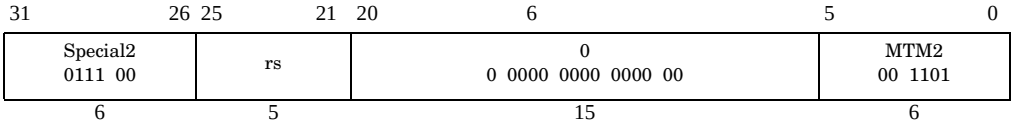
Operation:

```
if Are64bitOperationsEnabled() and !CvmCtl[NOMUL] then
   MPL1<63:0> = GPR[rs]
   P0 = 0
   P1 = 0
   P2 = 0
else
   SignalException(ReservedInstruction)
endif
```

Exceptions:

Reserved Instruction

Load Multiplier Register MPL2 — MTM2



Format: MTM2 rs — **CVM**

Purpose:

To load the multiplier register MPL2 (and zero P0, P1, P2).

Description: `MPL2 = rs; P0, P1, P2 = 0`

The 64-bit doubleword value in GPR rs is loaded into the multiplier register MPL2 and P0-P2 are zeroed.

Restrictions:

A Reserved Instruction Exception is signaled if access to 64-bit operations is not enabled.

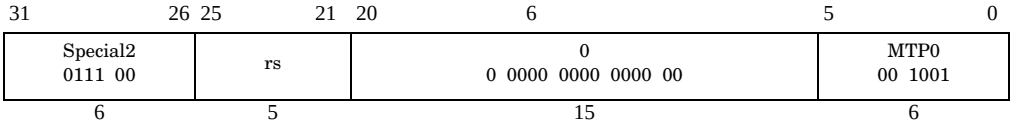
Operation:

```
if Are64bitOperationsEnabled() and !CvmCtl[NOMUL] then
   MPL2<63:0> = GPR[rs]
   P0 = 0
   P1 = 0
   P2 = 0
else
   SignalException(ReservedInstruction)
endif
```

Exceptions:

Reserved Instruction

Load Multiplier Register P0 — MTP0



Format: MTP0 rs **CVM**

Purpose:

To load the product register P0.

Description: `P0 = rs`

The 64-bit doubleword value in GPR rs is loaded into the product register P0.

Restrictions:

A Reserved Instruction Exception is signaled if access to 64-bit operations is not enabled.

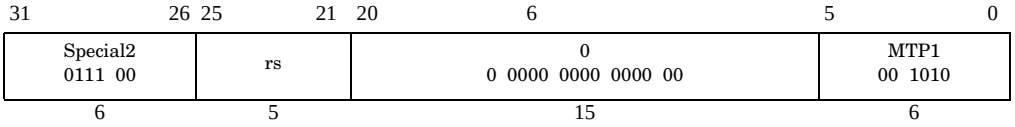
Operation:

```
if Are64bitOperationsEnabled() and !CvmCtl[NOMUL] then
   P0 = GPR[rs]
else
   SignalException(ReservedInstruction)
endif
```

Exceptions:

Reserved Instruction

Load Multiplier Register P1 — MTP1



Format: MTP1 rs **CVM**

Purpose:

To load the product register P1.

Description: `P1 = rs`

The 64-bit doubleword value in GPR rs is loaded into the product register P1.

Restrictions:

A Reserved Instruction Exception is signaled if access to 64-bit operations is not enabled.

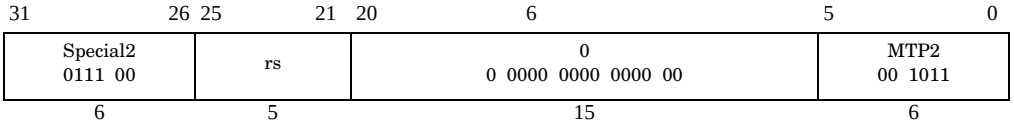
Operation:

```
if Are64bitOperationsEnabled() and !CvmCtl[NOMUL] then
   P1 = GPR[rs]
else
   SignalException(ReservedInstruction)
endif
```

Exceptions:

Reserved Instruction

Load Multiplier Register P2 — MTP2



Format: MTP2 rs **CVM**

Purpose:

To load the product register P2.

Description: `P2 = rs`

The 64-bit doubleword value in GPR rs is loaded into the product register P2.

Restrictions:

A Reserved Instruction Exception is signaled if access to 64-bit operations is not enabled.

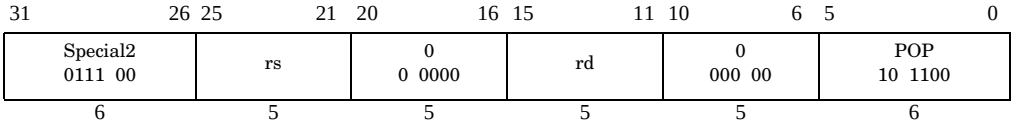
Operation:

```
if Are64bitOperationsEnabled() and !CvmCtl[NOMUL] then
   P2 = GPR[rs]
else
   SignalException(ReservedInstruction)
endif
```

Exceptions:

Reserved Instruction

Count Ones in a Word — POP



Format: POP rd, rs **CVM**

Purpose

Count the number of ones in a word

Description: `rd = count_ones(rs)`

Bits 31..0 of GPR rs are scanned. The number of ones is counted and the result is written to GPR rd.

Restrictions:

None.

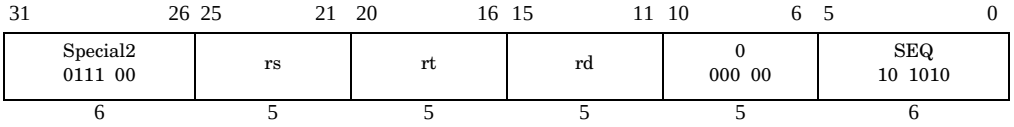
Operation:

```
temp = 0
for i in 31 .. 0
   if GPR[rs]<i> = 1 then
      temp++
   endif
endfor
GPR[rd] = temp
```

Exceptions:

None.

Set on Equal — SEQ



Format: SEQ rd, rs, rt **CVM**

Purpose:

To record the result of an equals comparison

Description: `rd = (rs == rt)`

Compare the contents of GPR rs and GPR rt and record the Boolean result of the comparison in GPR rd. If GPR rs equals GPR rt, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None.

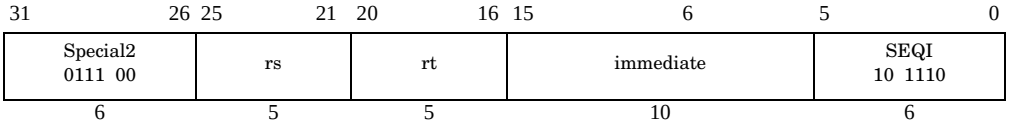
Operation:

```
if GPR[rs] == GPR[rt] then
   GPR[rd] = 0^63 || 1
else
   GPR[rd] = 0^64
endif
```

Exceptions:

None.

Set on Equal Immediate — SEQI



Format: SEQI rt, rs, immediate **CVM**

The immediate is a 10-bit sign-extended value contained in <15:6> of the instruction.

Purpose: To record the result of an equals comparison with a constant

Description: `rt = (rs == immediate)`

Compare the contents of GPR rs and the 10-bit signed immediate and record the Boolean result of the comparison in GPR rt. If GPR rs equals immediate, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None.

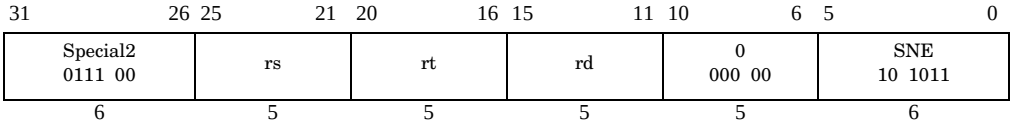
Operation:

```
if GPR[rs] == sign_extend(immediate) then
   GPR[rt] = 0^63 || 1
else
   GPR[rt] = 0^64
endif
```

Exceptions:

None.

Set on Not Equal — SNE



Format: SNE rd, rs, rt **CVM**

Purpose:

To record the result of a not equals comparison

Description: `rd = (rs != rt)`

Compare the contents of GPR rs and GPR rt and record the Boolean result of the comparison in GPR rd. If GPR rs equals GPR rt, the result is 0 (false); otherwise, it is 1 (true).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None.

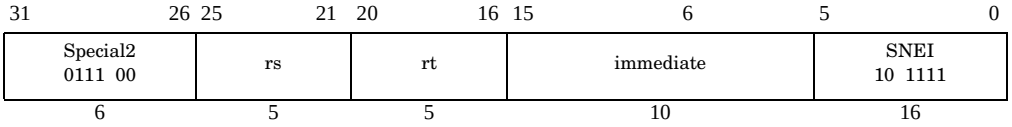
Operation:

```
if GPR[rs] != GPR[rt] then
   GPR[rd] = 0^63 || 1
else
   GPR[rd] = 0^64
endif
```

Exceptions:

None.

Set on Not Equal Immediate — SNEI



Format: SNEI rt, rs, immediate **CVM**

The immediate is a 10-bit sign-extended value contained in <15:6> of the instruction.

Purpose:

To record the result of a not equals comparison with a constant

Description: `rt = (rs != immediate)`

Compare the contents of GPR rs and the 10-bit signed immediate and record the Boolean result of the comparison in GPR rt. If GPR rs equals immediate, the result is 0 (false); otherwise, it is 1 (true).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None.

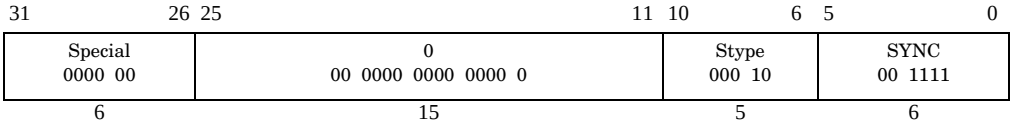
Operation:

```
if GPR[rs] != sign_extend(immediate) then
   GPR[rt] = 0^63 || 1
else
   GPR[rt] = 0^64
endif
```

Exceptions:

None.

Synchronize IOBDMAs — SYNCIOBDMA



Format: SYNCIOBDMA **CVM**

Purpose:

Complete all outstanding asynchronous I/O load operations.

Restrictions:

None.

Operation:

```
SyncOperation(stype)
```

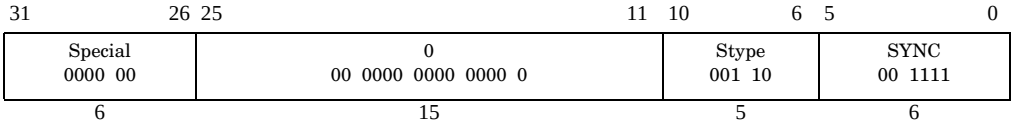
Exceptions:

None.

Programming Notes:

This instruction is not compliant with the MIPS64® Architecture specification and is incompatible with the future architecture definition and other MIPS64® compliant processors. Software should use this instruction only in situations in which the software is known to be running on this Cavium processor.

Synchronize Special — SYNCS



Format: SYNCS **CVM**

Purpose:

Similar to SYNC except that, depending on the cache-coherency attribute, certain L2/DRAM load/store operations are not ordered.

Restrictions:

None.

Operation:

```
SyncOperation(stype)
```

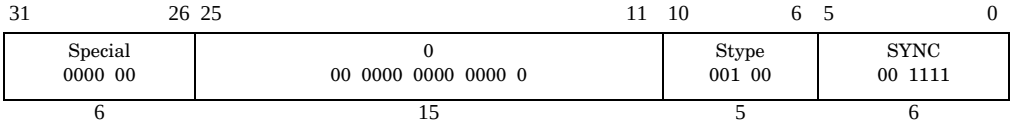
Exceptions:

None.

Programming Notes:

This instruction is not compliant with the MIPS64® Architecture specification and is incompatible with the future architecture definition and other MIPS64® compliant processors. Software should use this instruction only in situations in which the software is known to be running on this Cavium processor.

Synchronize Stores — SYNCW



Format: SYNCW **CVM**

Purpose:

To order stores. SYNCW is similar to SYNCWS, but SYNCW orders more stores.

Restrictions:

None.

Operation:

```
SyncOperation(stype)
```

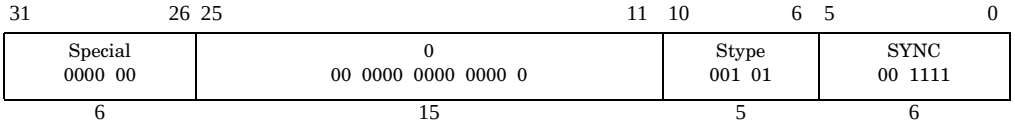
Exceptions:

None.

Programming Notes:

This instruction is not compliant with the MIPS64® Architecture specification and is incompatible with the future architecture definition and other MIPS64® compliant processors. Software should use this instruction only in situations in which the software is known to be running on this Cavium processor.

Synchronize Stores Special — SYNCWS



Format: SYNCWS **CVM**

Purpose:

Similar to SYNCW except that depending on the cache-coherency attribute certain L2/DRAM store operations are not orderred.

Restrictions:

None.

Operation:

```
SyncOperation(stype)
```

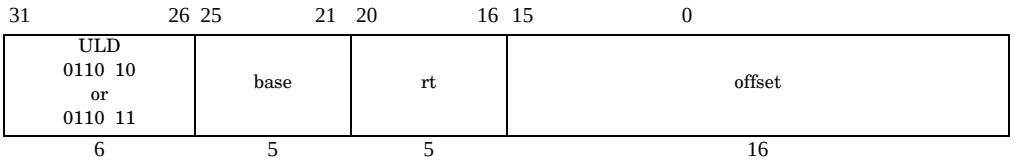
Exceptions:

None.

Programming Notes:

This instruction is not compliant with the MIPS64® Architecture specification and is incompatible with the future architecture definition and other MIPS64® compliant processors. Software should use this instruction only in situations in which the software is known to be running on this Cavium processor.

Unaligned Load Doubleword ULD



Format: ULD rt, offset(base) **CVM**

The ULD instruction does not exist when CvmCtl[USEUN] is clear. (The MIPS LDL and LDR instructions do not exist when CvmCtl[USEUN] is set.)

The ULD instruction (when enabled by CvmCtl[USEUN]) consumes either the MIPS LDL or LDR major opcodes according to the following table:

CvmCtl[USELONLY]	BigEndianCPU (!CvmCtl[LE])	MIPS LDL opcode is:	MIPS LDR opcode is:
0	0	NOP	ULD
0	1	ULD	NOP
1	X	ULD	NOP

When CvmCtl[USEUN] is set, one of MIPS LDL and LDR major opcodes execute as NOPs, as specified in the table above.

CvmCtl[USELY], CvmCtl[LE] and CvmCtl[USEUN] refer to bit 0, 1 and 12 of the Cavium-specific COP0 register #9, select = 7 respectively.

Purpose:

To load a doubleword from a (potentially-unaligned) memory location

Description: `rt = memory[base+offset]`

The contents of the 64-bit doubleword at the memory location specified by the effective address are fetched and placed in GPR rt. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error, Reserved Instruction, Watch, Breakpoint

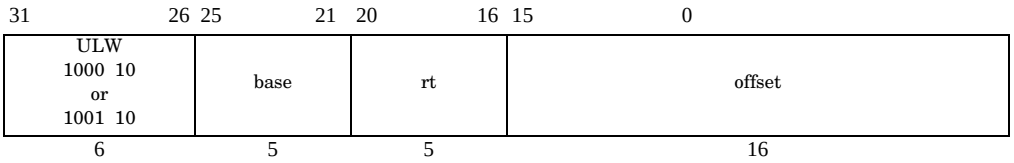
Programming Notes:

OCTEON Cores execute naturally-aligned LDs one cycle faster than naturally-aligned ULDs, so the LD instructions should be used rather than ULD when an address is known to be naturally-aligned.

ULD is an assembler macro that converts to MIPS LDL/LDR sequences on most MIPS assemblers.

This mode and the instruction functional changes that it controls is not compliant with the MIPS64® Architecture specification and may not be compatible with the future architecture definition and other MIPS64® compliant processors. Software should use this mode only in situations in which the software is known to be running on this Cavium processor.

Unaligned Load Word ULW



Format: ULW rt, offset(base) **CVM**

The ULW instruction does not exist when CvmCtl[USEUN] is clear. (The MIPS LWL and LWR instructions do not exist when CvmCtl[USEUN] is set.)

The ULW instruction (when enabled by CvmCtl[USEUN]) consumes either the MIPS LWL or LWR major opcodes according to the following table:

CvmCtl[USELONLY]	BigEndianCPU (!CvmCtl[LE])	MIPS LWL opcode is:	MIPS LWR opcode is:
0	0	NOP	ULW
0	1	ULW	NOP
1	X	ULW	NOP

When CvmCtl[USEUN] is set, one of MIPS LWL and LWR major opcodes execute as NOPs, as specified in the table above.

CvmCtl[USELY], CvmCtl[LE] and CvmCtl[USEUN] refer to bit 0, 1 and 12 of the Cavium-specific COP0 register #9, select = 7 respectively.

Purpose:

To load a word from a (potentially-unaligned) memory location

Description: `rt = memory[base+offset]`

The contents of the 32-bit word at the memory location specified by the effective address are fetched, sign-extended to 64-bits, and placed in GPR rt. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error, Watch, Breakpoint

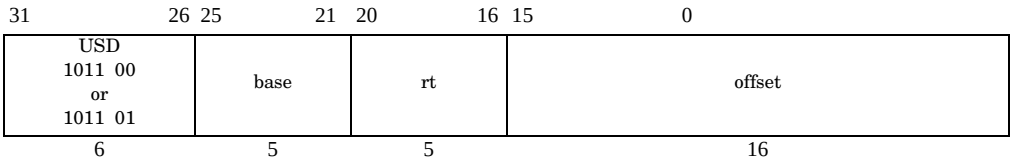
Programming Notes:

OCTEON Cores execute naturally-aligned LWs one cycle faster than naturally-aligned ULWs, so the LW instructions should be used rather than ULW when an address is known to be naturally-aligned.

ULW is an assembler macro that converts to MIPS LWL/LWR sequences on most MIPS assemblers.

This mode and the instruction functional changes that it controls is not compliant with the MIPS64® Architecture specification and may not be compatible with the future architecture definition and other MIPS64® compliant processors. Software should use this mode only in situations in which the software is known to be running on this Cavium processor.

Unaligned Store Doubleword USD



Format: USD rt, offset(base) **CVM**

The USD instruction does not exist when CvmCtl[USEUN] is clear. (The MIPS SDL and SDR instructions do not exist when CvmCtl[USEUN] is set.)

The USD instruction (when enabled by CvmCtl[USEUN]) consumes either the MIPS SDL or SDR major opcodes according to the following table:

CvmCtl[USELY]	BigEndianCPU (!CvmCtl[LE])	MIPS SDL opcode is:	MIPS SDR opcode is:
0	0	NOP	USD
0	1	USD	NOP
1	X	USD	NOP

When CvmCtl[USEUN] is set, one of MIPS SDL and SDR major opcodes execute as NOPs, as specified in the table above.

CvmCtl[USELY], CvmCtl[LE] and CvmCtl[USEUN] refer to bit 0, 1 and 12 of the Cavium-specific COP0 register #9, select = 7 respectively.

Purpose:

To store a doubleword to a (potentially-unaligned) memory location

Description: `memory[base+offset] = rt`

The 64-bit doubleword in GPR rt is stored in memory at the location specified by the effective address. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

If the effective address is not naturally-aligned and the second of the two memory stores required to complete the unaligned reference encounters an Address Error, TLB Refill, TLB Invalid, or TLB Modified exception, the first of the two memory references will still complete. Effectively, the USD may partially complete.

If the effective address is not naturally-aligned, the two memory stores required to complete the unaligned reference may complete in any order (as seen by other OCTEON Cores and I/O devices).

Exceptions:

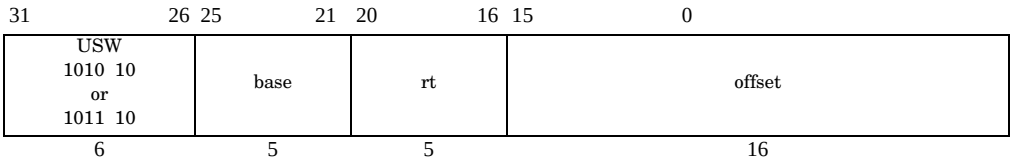
TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error, Reserved Instruction, Watch, Breakpoint

Programming Notes:

OCTEON Cores execute naturally-aligned SDs one cycle faster than naturally-aligned USDs, so the SD instructions should be used rather than USD when an address is known to be naturally-aligned.

USD is an assembler macro that converts to MIPS SDL/SDR sequences on most MIPS assemblers.

Unaligned Store Word USW



Format: USW rt, offset(base) **CVM**

The USW instruction does not exist when CvmCtl[USEUN] is clear. (The MIPS SWL and SWR instructions do not exist when CvmCtl[USEUN] is set.)

The USW instruction (when enabled by CvmCtl[USEUN]) consumes either the MIPS SWL or SWR major opcodes according to the following table:

CvmCtl[USELY]	BigEndianCPU (!CvmCtl[LE])	MIPS SWL opcode is:	MIPS SWR opcode is:
0	0	NOP	USW
0	1	USW	NOP
1	X	USW	NOP

When CvmCtl[USEUN] is set, one of MIPS SWL and SWR major opcodes execute as NOPs, as specified in the table above.

CvmCtl[USELY], CvmCtl[LE] and CvmCtl[USEUN] refer to bit 0, 1 and 12 of the Cavium-specific COP0 register #9, select = 7 respectively.

Purpose:

To store a word to a (potentially-unaligned) memory location

Description: `memory[base+offset] = rt`

The 64-bit word in GPR rt is stored in memory at the location specified by the effective address. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

If the effective address is not naturally-aligned and two memory stores are required to complete the unaligned reference and the second of the two memory stores encounters an Address Error, TLB Refill, TLB Invalid, or TLB Modified exception, the first of the two memory references will still complete. Effectively, the USW may partially complete.

If the effective address is not naturally-aligned and two memory stores are required to complete the unaligned reference, the two memory stores may complete in any order (as seen by other OCTEON Cores and I/O devices).

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error, Watch, Breakpoint

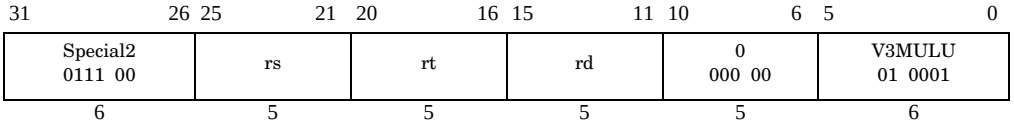
Programming Notes:

OCTEON Cores execute naturally-aligned SWs one cycle faster than naturally-aligned USWs, so the SW instructions should be used rather than USW when an address is known to be naturally-aligned.

USW is an assembler macro that converts to MIPS SWL/SWR sequences on most MIPS assemblers.

This mode and the instruction functional changes that it controls is not compliant with the MIPS64® Architecture specification and may not be compatible with the future architecture definition and other MIPS64® compliant processors. Software should use this mode only in situations in which the software is known to be running on this Cavium processor.

192-bit × 64-bit Unsigned Multiply and Add — V3MULU



Format: V3MULU rd, rs, rt **CVM**

Purpose:

To perform a 192-bit x 64-bit unsigned multiply and add yielding a 256-bit result.

Description: `(P2 || P1 || P0 || rd) = (0^64 || P2 || P1 || P0) + (0^192 || rt) + rs × (MPL2 || MPL1 || MPL0)`

The 64-bit doubleword value in GPR rs is multiplied by the 192-bit product registers MPL0-MPL2, treating both as unsigned values, producing a 256-bit result. The 64-bit doubleword value in GPR rt and the 64-bit product registers P0-P2 are zero-extended and added to the 256-bit result. The least-significant 64 bits of the result is placed in GPR rd. The most-significant 192 bits of the result is placed into the product registers P0-P2.

P0-P2 are 64-bit product registers. MPL0-MPL2 are 64-bit multiplier registers.

Restrictions:

A Reserved Instruction Exception is signaled if access to 64-bit operations is not enabled.

If a V3MULU is preceded by an MTP* instruction (without an intervening MTM*/VMM0, VMULU/V3MULU instruction), the MTP* instruction must be preceded by an MTM* or VMM0 instruction (without any intervening VMULU/V3MULU instructions).

If a V3MULU precedes a VMULU/VMM0, there must be an intervening MTM*/VMM0 between the two.

No overflow or other arithmetic exception occurs under any circumstances.

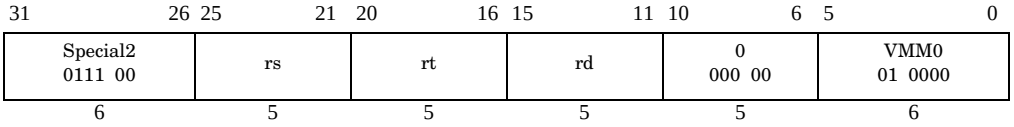
Operation:

```
if Are64bitOperationsEnabled() and !CvmCtl[NOMUL] then
   product_register<191:0> = MPL2<63:0> || MPL1<63:0> || MPL0<63:0>
   product<255:0> = GPR[rs]<63:0> * product_register<191:0>
   Pext<255:0> = 0^64 || P2<63:0> || P1<63:0> || P0<63:0>
   rtext<127:0> = 0^192 || GPR[rt]<63:0>
   sum<255:0> = product<255:0> + Pext<255:0> + rtext<255:0>
   GPR[rd] = sum<63:0>
   P2 = sum<255:192>
   P1 = sum<191:128>
   P0 = sum<127:64>
else
   SignalException(ReservedInstruction)
endif
```

Exceptions:

Reserved Instruction

64-bit Unsigned Multiply and Add Move — VMM0



Format: VMM0 rd, rs, rt **CVM**

Purpose:

To perform a 64-bit x 64-bit unsigned multiply and add yielding a 128-bit result.

Description:

```
rd = P0 + rt + rs × MPL0
MPL0 = rd
P0, P1, P2 = 0
```

The 64-bit doubleword value in GPR rs is multiplied by the 64-bit product register MPL0, treating both as unsigned values. The 64-bit doubleword value in GPR rt and the 64-bit product register P0 are added to the result. The 64-bit result is placed in GPR rd and the multiplier register MPL0. The product registers P0-P2 are zeroed.

MPL1 and MPL2 are unpredictable after this operation executes.

P0-P2 are 64-bit product registers. MPL0-MPL2 are 64-bit multiplier registers.

Restrictions:

A Reserved Instruction Exception is signaled if access to 64-bit operations is not enabled.

If a VMM0 instruction is preceded by an MTP* instruction (without an intervening MTM*/VMM0, VMULU/V3MULU instruction), the MTP* instruction must be preceded by an MTM* or VMM0 instruction (without any intervening VMULU/V3MULU instructions).

If a VMM0 is preceded by a V3MULU, there must be an intervening MTM*/ VMM0 between the two.

No overflow or other arithmetic exception occurs under any circumstances.

Operation:

```
if Are64bitOperationsEnabled() and !CvmCtl[NOMUL] then
   product<127:0> = GPR[rs]<63:0> * MPL0<63:0>
   Pext<127:0> = 0^64 || P0<63:0>
   rtext<127:0> = 0^64 || GPR[rt]<63:0>
   sum<127:0> = product<127:0> + Pext<127:0> + rtext<127:0>
   GPR[rd] = sum<63:0>
   MPL0 = sum<63:0>
   MPL1 = unpredictable
   MPL2 = unpredictable
   P0 = 0
   P1 = 0
   P2 = 0
else
   SignalException(ReservedInstruction)
```

```
endif
```

Exceptions:

Reserved Instruction

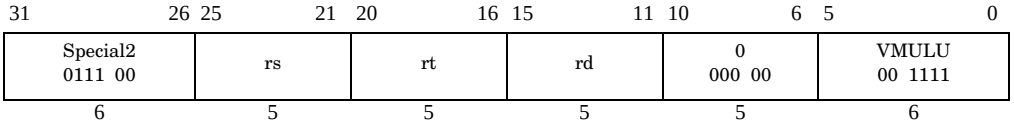
Programming Notes:

VMM0 rd, rs, rt is functionally identical to the two-instruction sequence:

VMULU rd, rs, rt

MTM0 rd

64-bit Unsigned Multiply and Add — VMULU



Format: VMULU rd, rs, rt **CVM**

Purpose:

To perform a 64-bit x 64-bit unsigned multiply and add yielding a 128-bit result.

Description: `(P0 || rd) = (0^64 || P0) + (0^64 || rt) + rs×  MPL0`

The 64-bit doubleword value in GPR rs is multiplied by the 64-bit product register MPL0, treating both as unsigned values, producing a 128-bit result. The 64-bit doubleword value in GPR rt and the 64-bit product register P0 are zero-extended and added to the 128-bit result. The least-significant 64 bits of the result is placed in GPR rd. The most-significant 64 bits of the result is placed into the product register P0.

P1, P2, MPL1, and MPL2 are unpredictable after this operation executes.

P0-P2 are 64-bit product registers. MPL0-MPL2 are 64-bit multiplier registers.

Restrictions:

A Reserved Instruction Exception is signaled if access to 64-bit operations is not enabled.

If a VMULU is preceded by an MTP* instruction (without an intervening MTM*/ VMM0, VMULU/V3MULU instruction), the MTP* instruction must be preceded by an MTM* or VMM0 instruction (without any intervening VMULU/V3MULU instructions).

If a VMULU is preceded by a V3MULU, there must be an intervening MTM*/ VMM0 between the two.

No overflow or other arithmetic exception occurs under any circumstances.

Operation:

```
if Are64bitOperationsEnabled() and !CvmCtl[NOMUL] then
   product<127:0> = GPR[rs]<63:0> * MPL0<63:0>
   Pext<127:0> = 0^64 || P0<63:0>
   rtext<127:0> = 0^64 || GPR[rt]<63:0>
   sum<127:0> = product<127:0> + P2ext<127:0> + rtext<127:0>
   GPR[rd] = sum<63:0>
   P0 = sum<127:64>
   P1 = unpredictable
   P2 = unpredictable
   MPL1 = unpredictable
   MPL2 = unpredictable
else
   SignalException(ReservedInstruction)
endif
```

Exceptions:

Reserved Instruction

1.2.1 Note:

If you need to ensure OCTEON code runs on other MIPS64 Release 2 cores please:

1. Set the following control bits to zero:

   CvmMemCtl[CMVSEGENAU]

   CvmMemCtl[CMVSEGENAS]

   CvmMemCtl[XKMEMENAU]

   CvmMemCtl[XKIOENAU]

   CvmMemCtl[XKMEMENAS]

   CvmMemCtl[XKIOENAS]

   CvmCtl[USEUN]

   CvmCtl[USELY]

   CvmCtl[REPUN]

   CvmCtl[DISIOCACHE]

2. Software which makes use of OCTEON-specific cache coherence attributes, the SYNCIOBDMA, SYNCS, SYNCW, SYNCWS instructions or data cache operations may not be used.