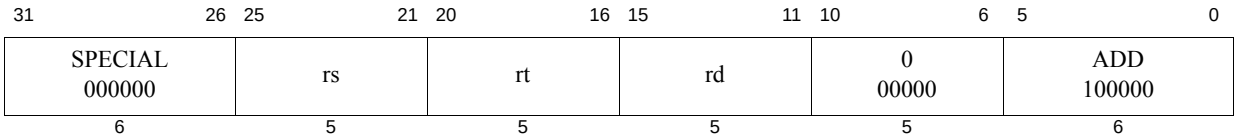


Format: `ADD rd, rs, rt` **MIPS32**

Purpose: Add Word

To add 32-bit integers. If an overflow occurs, then trap.

Description: `GPR[rd] ← GPR[rs] + GPR[rt]`

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* to produce a 32-bit result.

- If the addition results in 32-bit 2's complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs.
- If the addition does not overflow, the 32-bit result is placed into GPR *rd*.

Restrictions:

None

Operation:

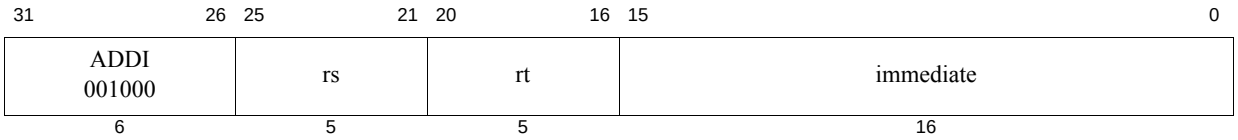
```
temp ← (GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← temp
endif
```

Exceptions:

Integer Overflow

Programming Notes:

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



Format: `ADDI rt, rs, immediate` **MIPS32, removed in Release 6**

Purpose: Add Immediate Word

To add a constant to a 32-bit integer. If overflow occurs, then trap**.**

Description: `GPR[rt] ← GPR[rs] + immediate`

The 16-bit signed *immediate* is added to the 32-bit value in GPR *rs* to produce a 32-bit result.

- If the addition results in 32-bit 2's complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs.
- If the addition does not overflow, the 32-bit result is placed into GPR *rt*.

Restrictions:

Availability and Compatibility:

This instruction has been removed in Release 6. The encoding has been reused for other instructions introduced by Release 6.

Operation:

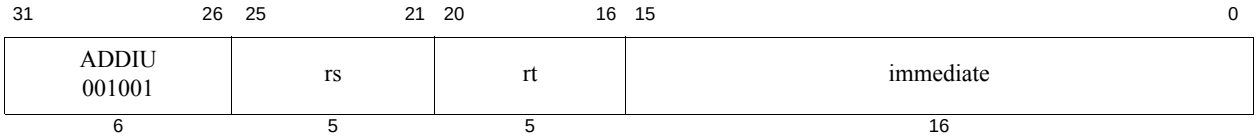
```
temp ← (GPR[rs]31||GPR[rs]31..0) + sign_extend(immediate)
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rt] ← temp
endif
```

Exceptions:

Integer Overflow

Programming Notes:

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



Format: `ADDIU rt, rs, immediate` **MIPS32**

Purpose: Add Immediate Unsigned Word

To add a constant to a 32-bit integer.

Description: `GPR[rt] ← GPR[rs] + immediate`

The 16-bit signed *immediate* is added to the 32-bit value in GPR *rs* and the 32-bit arithmetic result is placed into GPR *rt*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

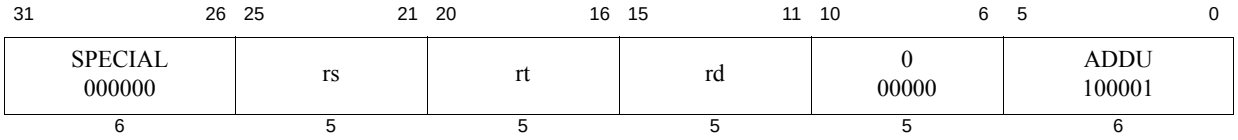
```
temp ← GPR[rs] + sign_extend(immediate)
GPR[rt] ← temp
```

Exceptions:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. This instruction is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.



Format: `ADDU rd, rs, rt` **MIPS32**

Purpose: Add Unsigned Word

To add 32-bit integers.

Description: `GPR[rd] ← GPR[rs] + GPR[rt]`

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* and the 32-bit arithmetic result is placed into GPR *rd*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

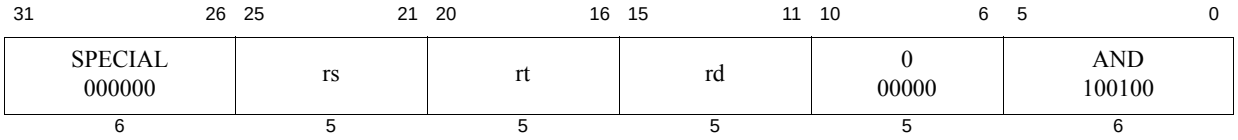
```
temp ← GPR[rs] + GPR[rt]
GPR[rd] ← temp
```

Exceptions:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. This instruction is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.



Format: `AND rd, rs, rt` **MIPS32**

Purpose: and

To do a bitwise logical AND.

Description: `GPR[rd] ← GPR[rs] and GPR[rt]`

The contents of GPR *rs* are combined with the contents of GPR *rt* in a bitwise logical AND operation. The result is placed into GPR *rd*.

Restrictions:

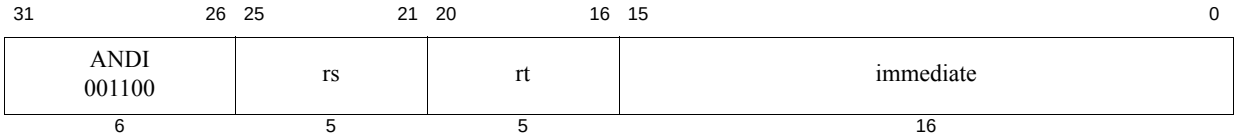
None

Operation:

```
GPR[rd] ← GPR[rs] and GPR[rt]
```

Exceptions:

None



Format: `ANDI rt, rs, immediate` **MIPS32**

Purpose: and immediate

To do a bitwise logical AND with a constant

Description: `GPR[rt] ← GPR[rs] and zero_extend(immediate)`

The 16-bit immediate is zero-extended to the left and combined with the contents of GPR rs in a bitwise logical AND operation. The result is placed into GPR *rt*.

Restrictions:

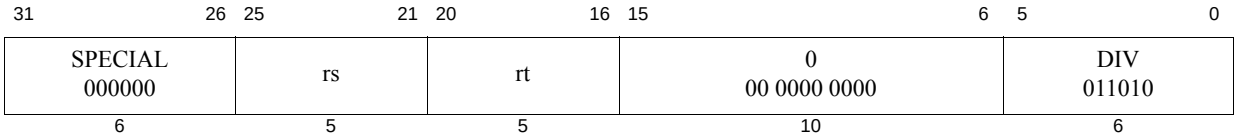
None

Operation:

```
GPR[rt] ← GPR[rs] and zero_extend(immediate)
```

Exceptions:

None



Format: `DIV rs, rt` **MIPS32, removed in Release 6**

Purpose: Divide Word

To divide a 32-bit signed integers.

Description: `(HI, LO) ← GPR[rs] / GPR[rt]`

The 32-bit word value in GPR *rs* is divided by the 32-bit value in GPR *rt,* treating both operands as signed values. The 32-bit quotient is placed into special register *LO* and the 32-bit remainder isplaced into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

If the divisor in GPR *rt* is zero, the arithmetic result value is **UNPREDICTABLE**.

Availability and Compatibility:

DIV has been removed in Release 6 and has been replaced by DIV and MOD instructions that produce only quotient and remainder, respectively. Refer to the Release 6 introduced ‘DIV’ and ‘MOD’ instructions in this manual for more information. This instruction remains current for all release levels lower than Release 6 of the MIPS architecture.

Operation:

```
q  ← GPR[rs]31..0 div GPR[rt]31..0
LO ← q
r  ← GPR[rs]31..0 mod GPR[rt]31..0
HI ← r
```

Exceptions:

None

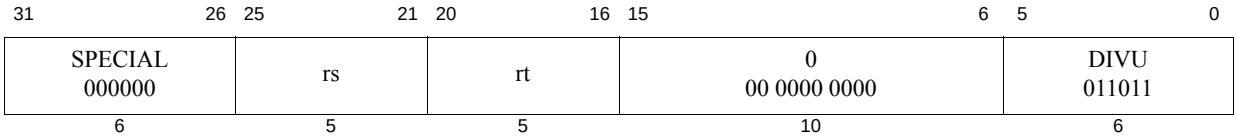
Programming Notes:

No arithmetic exception occurs under any circumstances. If divide-by-zero or overflow conditions are detected and some action taken, then the divide instruction is followed by additional instructions to check for a zero divisor and/or for overflow. If the divide is asynchronous then the zero-divisor check can execute in parallel with the divide. The action taken on either divide-by-zero or overflow is either a convention within the program itself, or within the system software. A possibility is to take a BREAK exception with a *code* field value to signal the problem to the system software.

As an example, the C programming language in a UNIX® environment expects division by zero to either terminate the program or execute a program-specified signal handler. C does not expect overflow to cause any exceptional condition. If the C compiler uses a divide instruction, it also emits code to test for a zero divisor and execute a BREAK instruction to inform the operating system if a zero is detected.

By default, most compilers for the MIPS architecture emits additional instructions to check for the divide-by-zero and overflow cases when this instruction is used. In many compilers, the assembler mnemonic “DIV r0, rs, rt” can be used to prevent these additional test instructions to be emitted.

In some processors the integer divide operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read *LO* or *HI* before the results are written interlocks until the results are



Format: `DIVU rs, rt` **MIPS32, removed in Release 6**

Purpose: Divide Unsigned Word

To divide 32-bit unsigned integers

Description: `(HI, LO) ← GPR[rs] / GPR[rt]`

The 32-bit word value in GPR *rs* is divided by the 32-bit value in GPR *rt*, treating both operands as unsigned values. The 32-bit quotient is placed into special register *LO* and the 32-bit remainder is placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

If the divisor in GPR *rt* is zero, the arithmetic result value is **UNPREDICTABLE**.

Availability and Compatibility:

This instruction has been removed in Release 6.

Operation:

```
q  ← (0 || GPR[rs]31..0) div (0 || GPR[rt]31..0)
r  ← (0 || GPR[rs]31..0) mod (0 || GPR[rt]31..0)
LO ← sign_extend(q31..0)
HI ← sign_extend(r31..0)
```

Exceptions:

None

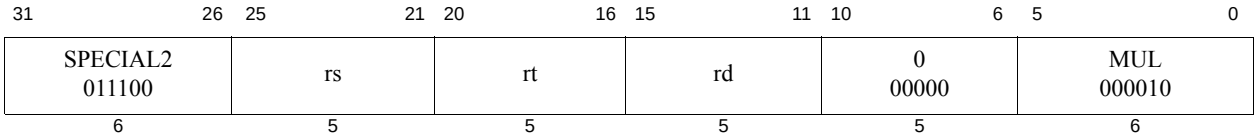
Programming Notes:

Pre-Release 6 instruction DIV has been removed in Release 6 and has been replaced by DIV and MOD instructions that produce only quotient and remainder, respectively. Refer to the Release 6 introduced ‘DIV’ and ‘MOD’ instructions in this manual for more information. This instruction remains current for all release levels lower than Release 6 of the MIPS architecture.

See “Programming Notes” for the DIV instruction.

Historical Perspective:

In MIPS 1 through MIPS III, if either of the two instructions preceding the divide is an MFHI or MFLO, the result of the MFHI or MFLO is UNPREDICTABLE. Reads of the *HI* or *LO* special register must be separated from subsequent instructions that write to them by two or more instructions. This restriction was removed in MIPS IV and MIPS32 and all subsequent levels of the architecture.



Format: `MUL rd, rs, rt` **MIPS32, removed in Release 6**

Purpose: Multiply Word to GPR

To multiply two words and write the result to a GPR.

Description: `GPR[rd] ← GPR[rs] x GPR[rt]`

The 32-bit word value in GPR *rs* is multiplied by the 32-bit value in GPR *rt*, treating both operands as signed values, to produce a 64-bit result. The least significant 32 bits of the product are written to GPR *rd*. The contents of *HI* and *LO* are **UNPREDICTABLE** after the operation. No arithmetic exception occurs under any circumstances.

Restrictions:

Note that this instruction does not provide the capability of writing the result to the *HI* and *LO* registers.

Availability and Compatibility:

The pre-Release 6 MUL instruction has been removed in Release 6. It has been replaced by a similar instruction of the same mnemonic, MUL, but different encoding, which is a member of a family of single-width multiply instructions. Refer to the ‘MUL’ and ‘MUH’ instructions in this manual for more information.

Operation:

```
temp ← GPR[rs] x GPR[rt]
GPR[rd] ← temp31..0
HI ← UNPREDICTABLE
LO ← UNPREDICTABLE
```

Exceptions:

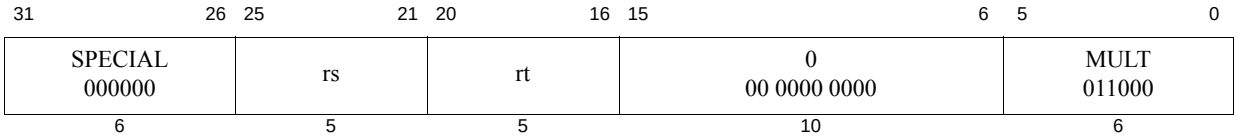
None

Programming Notes:

In some processors the integer multiply operation may proceed asynchronously and allow other CPU instructions to 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.

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.



Format: `MULT rs, rt` **MIPS32, removed in Release 6**

Purpose: Multiply Word

To multiply 32-bit signed integers.

Description: `(HI, LO) ← GPR[rs] x GPR[rt]`

The 32-bit word value in GPR *rt* is multiplied by the 32-bit value in GPR *rs*, treating both operands as signed values, to produce a 64-bit result. The low-order 32-bit word of the result is placed into special register *LO*, and the high-order 32-bit word is placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

None

Availability and Compatibility:

The MULT instruction has been removed in Release 6. It has been replaced by the Multiply Low (MUL) and Multiply High (MUH) instructions, whose output is written to a single GPR. Refer to the 'MUL' and 'MUH' instructions in this manual for more information.

Operation:

```
prod ← GPR[rs]31..0 x GPR[rt]31..0
LO ← prod31..0
HI ← prod63..32
```

Exceptions:

None

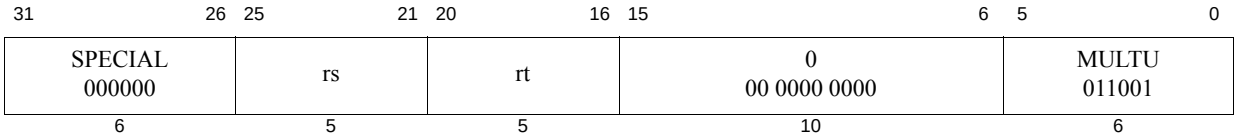
Programming Notes:

In some processors the integer multiply operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read *LO* or *HI* 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.

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

Implementation Note:



Format: `MULTU rs, rt` **MIPS32, removed in Release 6**

Purpose: Multiply Unsigned Word

To multiply 32-bit unsigned integers.

Description: `(HI, LO) ← GPR[rs] x GPR[rt]`

The 32-bit word value in GPR *rt* is multiplied by the 32-bit value in GPR *rs*, treating both operands as unsigned values, to produce a 64-bit result. The low-order 32-bit word of the result is placed into special register *LO*, and the high-order 32-bit word is placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

None

Availability and Compatibility:

The MULTU instruction has been removed in Release 6. It has been replaced by the Multiply Low (MULU) and Multiply High (MUHU) instructions, whose output is written to a single GPR. Refer to the ‘MULU’ and ‘MUHU’ instructions in this manual for more information.

Operation:

```
prod ← (0 || GPR[rs]31..0) x (0 || GPR[rt]31..0)
LO ← prod31..0
HI ← prod63..32
```

Exceptions:

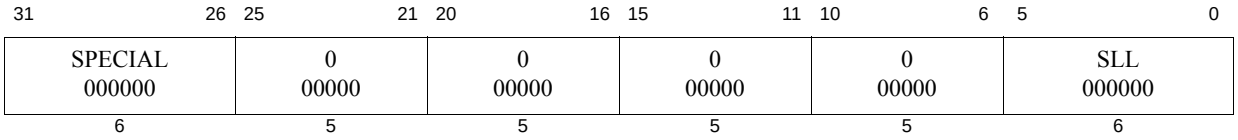
None

Programming Notes:

In some processors the integer multiply operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read *LO* or *HI* 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.

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.



Format: `NOP` **Assembly Idiom**

Purpose: No Operation

To perform no operation.

Description:

NOP is the assembly idiom used to denote no operation. The actual instruction is interpreted by the hardware as SLL r0, r0, 0.

Restrictions:

None

Operations:

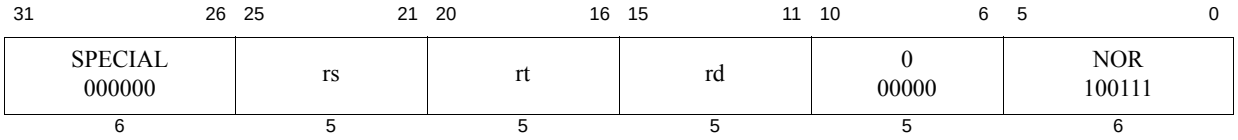
None

Exceptions:

None

Programming Notes:

The zero instruction word, which represents SLL, r0, r0, 0, is the preferred NOP for software to use to fill branch and jump delay slots and to pad out alignment sequences.



Format: `NOR rd, rs, rt` **MIPS32**

Purpose: Not Or

To do a bitwise logical NOT OR.

Description: `GPR[rd] ← GPR[rs] nor GPR[rt]`

The contents of GPR *rs* are combined with the contents of GPR *rt* in a bitwise logical NOR operation. The result is placed into GPR *rd*.

Restrictions:

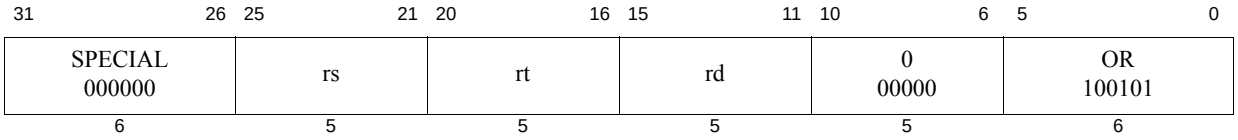
None

Operation:

```
GPR[rd] ← GPR[rs] nor GPR[rt]
```

Exceptions:

None



Format: `OR rd, rs, rt` **MIPS32**

Purpose: Or

To do a bitwise logical OR.

Description: `GPR[rd] ← GPR[rs] or GPR[rt]`

The contents of GPR *rs* are combined with the contents of GPR *rt* in a bitwise logical OR operation. The result is placed into GPR *rd*.

Restrictions:

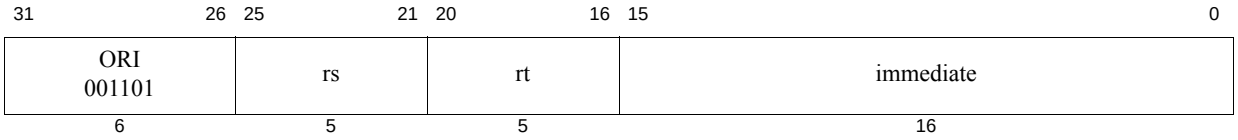
None

Operations:

```
GPR[rd] ← GPR[rs] or GPR[rt]
```

Exceptions:

None



Format: `ORI rt, rs, immediate` **MIPS32**

Purpose: Or Immediate

To do a bitwise logical OR with a constant.

Description: `GPR[rt] ← GPR[rs] or immediate`

The 16-bit *immediate* is zero-extended to the left and combined with the contents of GPR *rs* in a bitwise logical OR operation. The result is placed into GPR *rt*.

Restrictions:

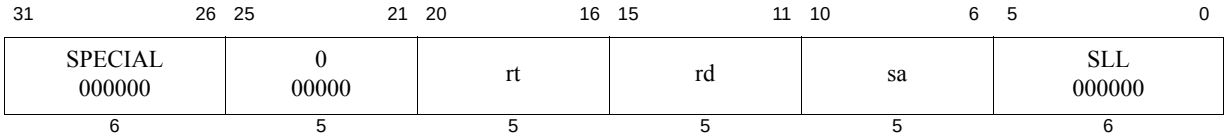
None

Operations:

```
GPR[rt] ← GPR[rs] or zero_extend(immediate)
```

Exceptions:

None



Format: `SLL rd, rt, sa` **MIPS32**

Purpose: Shift Word Left Logical

To left-shift a word by a fixed number of bits.

Description: `GPR[rd] ← GPR[rt] << sa`

The contents of the low-order 32-bit word of GPR *rt* are shifted left, inserting zeros into the emptied bits. The word result is placed in GPR *rd*. The bit-shift amount is specified by *sa*.

Restrictions:

None

Operation:

```
s ← sa
temp ← GPR[rt](31-s)..0 || 0^s
GPR[rd] ← temp
```

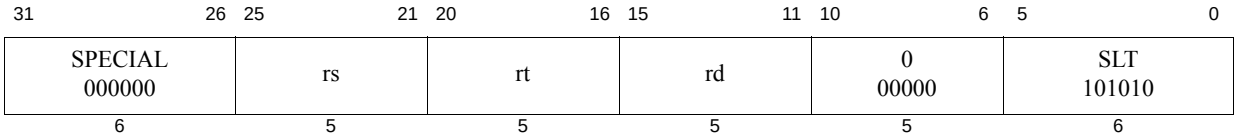
Exceptions:

None

Programming Notes:

SLL r0, r0, 0, expressed as NOP, is the assembly idiom used to denote no operation.

SLL r0, r0, 1, expressed as SSNOP, is the assembly idiom used to denote no operation that causes an issue break on superscalar processors.



Format: `SLT rd, rs, rt` **MIPS32**

Purpose: Set on Less Than

To record the result of a less-than comparison.

Description: `GPR[rd] ← (GPR[rs] < GPR[rt])`

Compare the contents of GPR *rs* and GPR *rt* as signed integers; record the Boolean result of the comparison in GPR *rd*. If GPR *rs* is less than GPR *rt*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

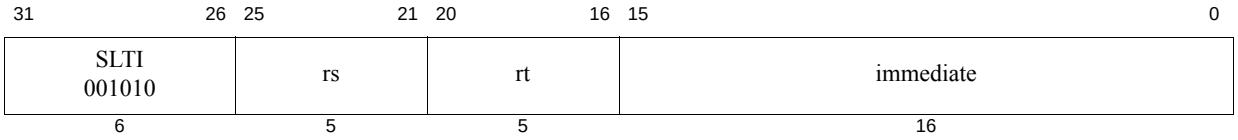
None

Operation:

```
if GPR[rs] < GPR[rt] then
   GPR[rd] ← 0^(GPRLEN-1) || 1
else
   GPR[rd] ← 0^GPRLEN
endif
```

Exceptions:

None



Format: `SLTI rt, rs, immediate` **MIPS32**

Purpose: Set on Less Than Immediate

To record the result of a less-than comparison with a constant.

Description: `GPR[rt] ← (GPR[rs] < sign_extend(immediate) )`

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

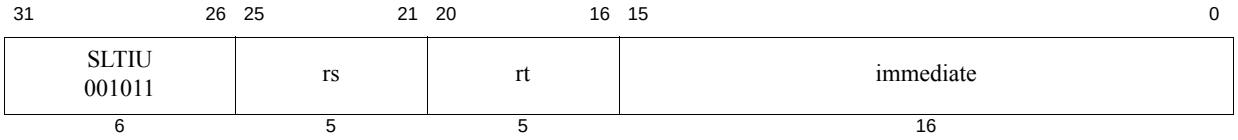
None

Operation:

```
if GPR[rs] < sign_extend(immediate) then
   GPR[rt] ← 0^(GPRLEN-1) || 1
else
   GPR[rt] ← 0^GPRLEN
endif
```

Exceptions:

None



Format: `SLTIU rt, rs, immediate` **MIPS32**

Purpose: Set on Less Than Immediate Unsigned

To record the result of an unsigned less-than comparison with a constant.

Description: `GPR[rt] ← (GPR[rs] < sign_extend(immediate))`

Compare the contents of GPR *rs* and the sign-extended 16-bit *immediate* as unsigned integers; record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate*, the result is 1 (true); otherwise, it is 0 (false).

Because the 16-bit *immediate* is sign-extended before comparison, the instruction can represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

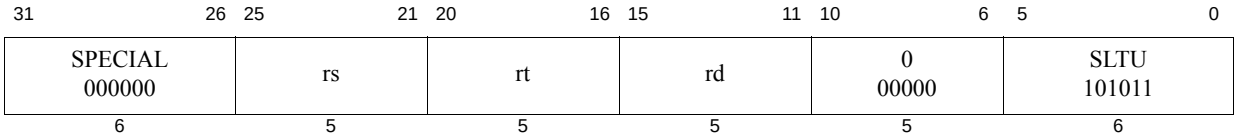
None

Operation:

```
if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
   GPR[rt] ← 0^(GPRLEN-1) || 1
else
   GPR[rt] ← 0^GPRLEN
endif
```

Exceptions:

None



Format: `SLTU rd, rs, rt` **MIPS32**

Purpose: Set on Less Than Unsigned

To record the result of an unsigned less-than comparison.

Description: `GPR[rd] ← (GPR[rs] < GPR[rt])`

Compare the contents of GPR *rs* and GPR *rt* as unsigned integers; record the Boolean result of the comparison in GPR *rd*. If GPR *rs* is less than GPR *rt*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

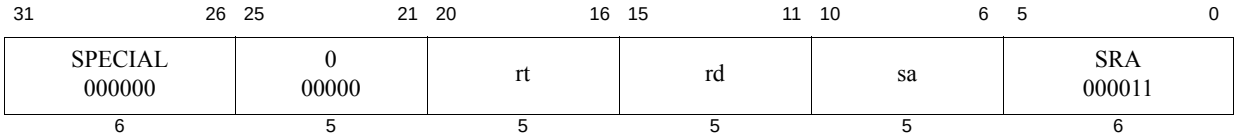
None

Operation:

```
if (0 || GPR[rs]) < (0 || GPR[rt]) then
   GPR[rd] ← 0^(GPRLEN-1) || 1
else
   GPR[rd] ← 0^GPRLEN
endif
```

Exceptions:

None



Format: `SRA rd, rt, sa` **MIPS32**

Purpose: Shift Word Right Arithmetic

To execute an arithmetic right-shift of a word by a fixed number of bits.

Description: `GPR[rd] ← GPR[rt] >> sa (arithmetic)`

The contents of the low-order 32-bit word of GPR *rt* are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by *sa*.

Restrictions:

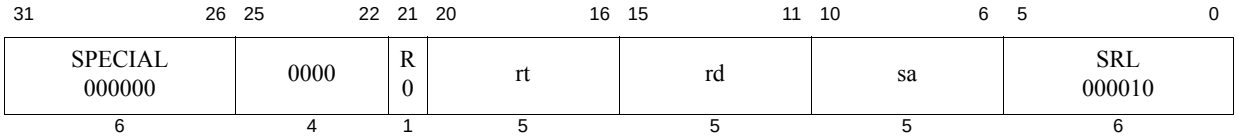
None

***s*Operation:**

```
ss ← sa
temp ← GPR[rt]31)s || GPR[rt]31..s
GPR[rd] ← temp
```

Exceptions:

None



Format: `SRL rd, rt, sa` **MIPS32**

Purpose: Shift Word Right Logical

To execute a logical right-shift of a word by a fixed number of bits.

Description: `GPR[rd] ← GPR[rt] >> sa (logical)`

The contents of the low-order 32-bit word of GPR *rt* are shifted right, inserting zeros into the emptied bits. The word result is placed in GPR *rd*. The bit-shift amount is specified by *sa*.

Restrictions:

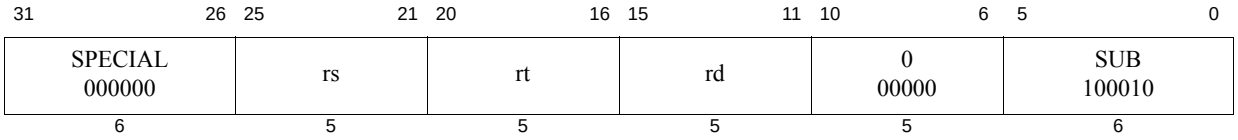
None

***s*Operation:**

```
ss ← sa
temp ← 0^s || GPR[rt]_31..s
GPR[rd] ← temp
```

Exceptions:

None



Format: `SUB rd, rs, rt` **MIPS32**

Purpose: Subtract Word

To subtract 32-bit integers. If overflow occurs, then trap.

Description: `GPR[rd] ← GPR[rs] – GPR[rt]`

The 32-bit word value in GPR *rt* is subtracted from the 32-bit value in GPR *rs* to produce a 32-bit result. If the subtraction results in 32-bit 2's complement arithmetic overflow, then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into GPR *rd*.

Restrictions:

None

Operation:

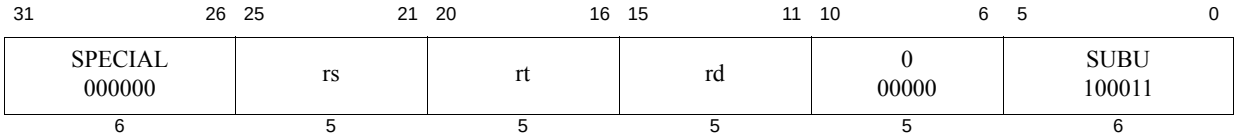
```
temp ← (GPR[rs]31||GPR[rs]31..0) – (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← temp31..0
endif
```

Exceptions:

Integer Overflow

Programming Notes:

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



Format: `SUBU rd, rs, rt` **MIPS32**

Purpose: Subtract Unsigned Word

To subtract 32-bit integers.

Description: `GPR[rd] ← GPR[rs] – GPR[rt]`

The 32-bit word value in GPR *rt* is subtracted from the 32-bit value in GPR *rs* and the 32-bit arithmetic result is and placed into GPR *rd*.

No integer overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

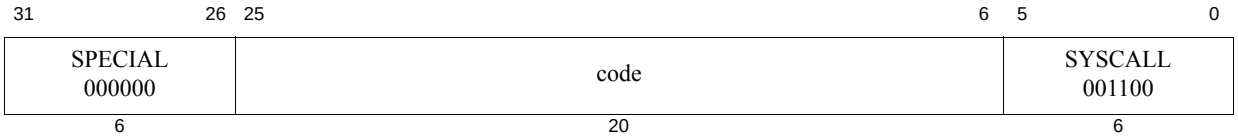
```
temp ← GPR[rs] – GPR[rt]
GPR[rd] ← temp
```

Exceptions:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. It is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.



Format: `SYSCALL` **MIPS32**

Purpose: System Call

To cause a System Call exception.

Description:

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

The *code* field is available for use as software parameters, but may be retrieved by the exception handler by loading the contents of the memory word containing the instruction. Alternatively, if CP0 *BadInstr* is implemented, the *code* field may be obtained from *BadInstr*.

Restrictions:

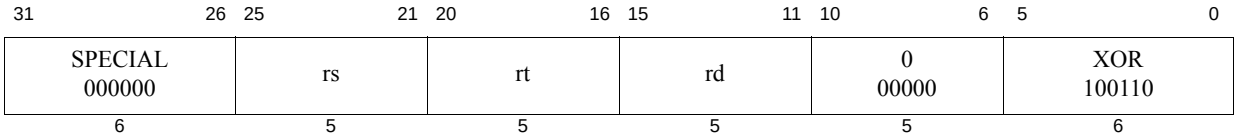
None

Operation:

```
SignalException(SystemCall)
```

Exceptions:

System Call



Format: `XOR rd, rs, rt` **MIPS32**

Purpose: Exclusive OR

To do a bitwise logical Exclusive OR.

Description: `GPR[rd] ← GPR[rs] XOR GPR[rt]`

Combine the contents of GPR *rs* and GPR *rt* in a bitwise logical Exclusive OR operation and place the result into GPR *rd*.

Restrictions:

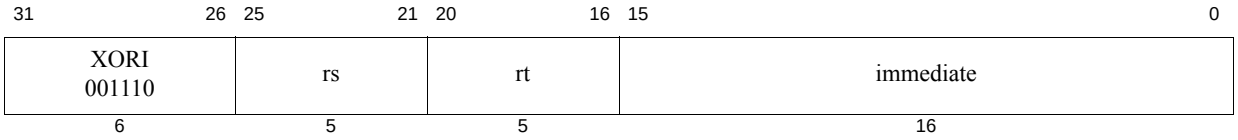
None

Operation:

```
GPR[rd] ← GPR[rs] xor GPR[rt]
```

Exceptions:

None



Format: `XORI rt, rs, immediate` **MIPS32**

Purpose: Exclusive OR Immediate

To do a bitwise logical Exclusive OR with a constant.

Description: `GPR[rt] ← GPR[rs] XOR immediate`

Combine the contents of GPR *rs* and the 16-bit zero-extended *immediate* in a bitwise logical Exclusive OR operation and place the result into GPR *rt*.

Restrictions:

None

Operation:

```
GPR[rt] ← GPR[rs] xor zero_extend(immediate)
```

Exceptions:

None