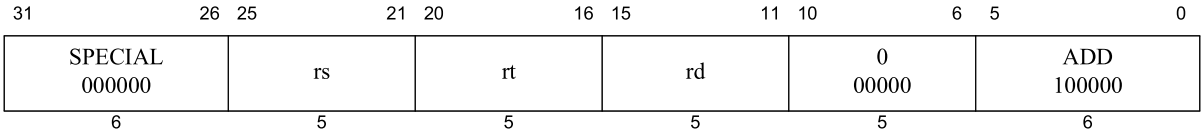


Format: `ADD rd, rs, rt` **MIPS32**

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:

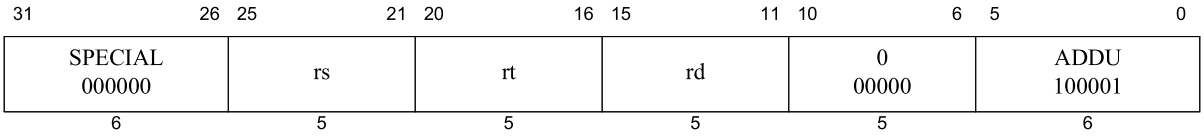
```
temp ← (GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← temp
endif
```

Exceptions:

Integer Overflow

Programming Notes:

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



Format: `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:

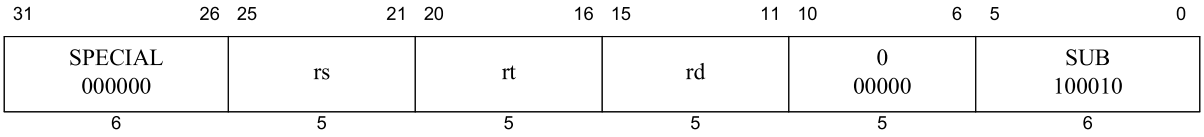
```
temp ← GPR[rs] + GPR[rt]
GPR[rd] ← temp
```

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: `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:

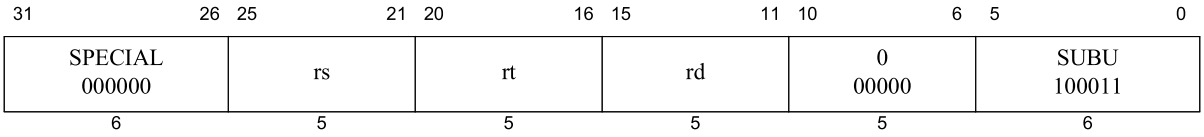
```
temp ← (GPR[rs]31||GPR[rs]31..0) – (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← temp31..0
endif
```

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:

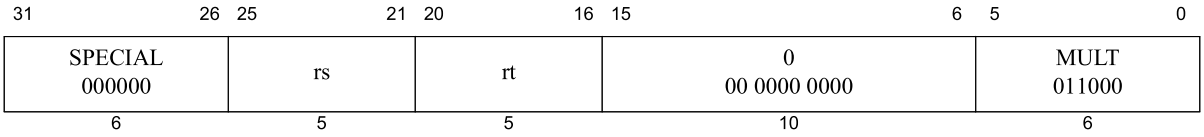
```
temp ← GPR[rs] – GPR[rt]
GPR[rd] ← temp
```

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: `MULT rs, rt` **MIPS32**

Purpose: Multiply Word

To multiply 32-bit signed integers

Description: `(HI, LO) ← GPR[rs] × 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 splaced into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

None

Operation:

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

Exceptions:

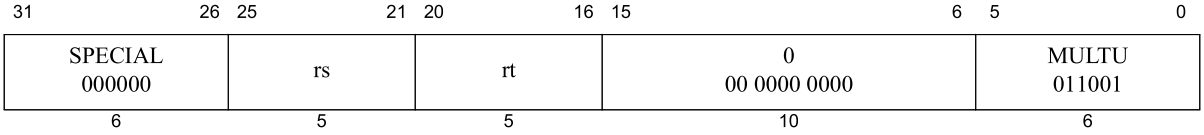
None

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: `MULTU rs, rt` **MIPS32**

Purpose: Multiply Unsigned Word

To multiply 32-bit unsigned integers

Description: `(HI, LO) ← GPR[rs] × 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

Operation:

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

Exceptions:

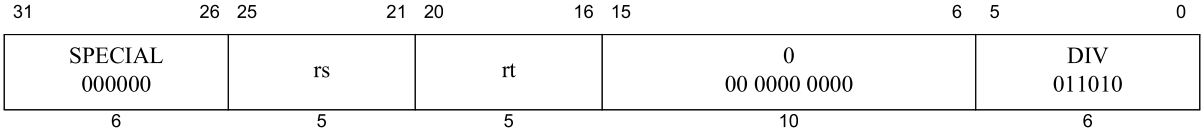
None

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: `DIV rs, rt` **MIPS32**

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**.

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 typically 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 more typically within the system software; one 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.

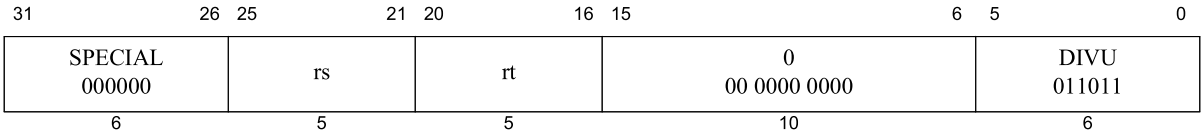
By default, most compilers for the MIPS architecture will emit additional instructions to check for the divide-by-zero and overflow cases when this instruction is used. In many compilers, the assembler mnemonic “DIV r0, rs, rt” can be used to prevent these additional test instructions to be emitted.

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 ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the divide so that other instructions can execute in parallel.

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: `DIVU rs, rt` **MIPS32**

Purpose: Divide Unsigned Word

To divide a 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**.

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:

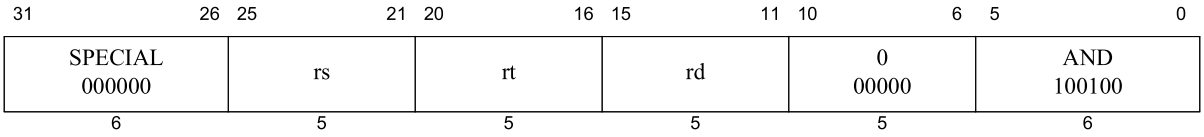
None

Programming Notes:

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: `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:

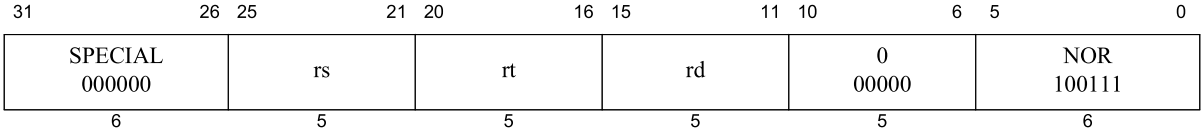
None

Operation:

```
GPR[rd] ← GPR[rs] and GPR[rt]
```

Exceptions:

None



Format: `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:

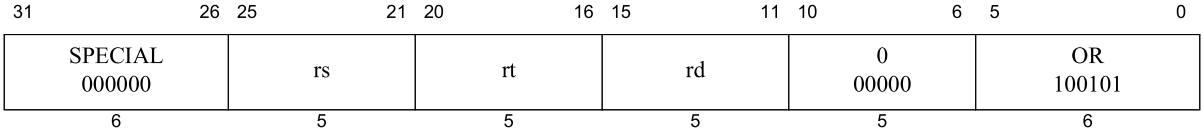
None

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:

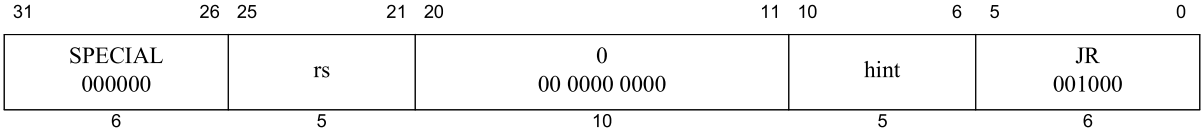
None

Operation:

```
GPR[rd] ← GPR[rs] or GPR[rt]
```

Exceptions:

None



Format: `JR rs` **MIPS32**

Purpose: Jump Register

To execute a branch to an instruction address in a register

Description: `PC ← GPR[rs]`

Jump to the effective target address in GPR *rs*. Execute the instruction following the jump, in the branch delay slot, before jumping.

For processors that implement the MIPS16e ASE or microMIPS32/64 ISA, set the *ISA Mode* bit to the value in GPR *rs* bit 0. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one

Restrictions:

If only one instruction set is implemented, then the effective target address must obey the alignment rules of the instruction set. If multiple instruction sets are implemented, the effective target address must obey the alignment rules of the intended instruction set of the target address as specified by the bit 0 or GPR *rs*.

For processors that do not implement the microMIPS ISA, the effective target address in GPR *rs* must be naturally-aligned. For processors that do not implement the MIPS16e ASE or microMIPS ISA, if either of the two least-significant bits are not zero, an Address Error exception occurs when the branch target is subsequently fetched as an instruction.

For processors that do implement the MIPS16e ASE or microMIPS ISA, if bit 0 is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

In release 1 of the architecture, the only defined hint field value is 0, which sets default handling of JR. In Release 2 of the architecture, bit 10 of the hint field is used to encode an instruction hazard barrier. See the JR.HB instruction description for additional information.

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:

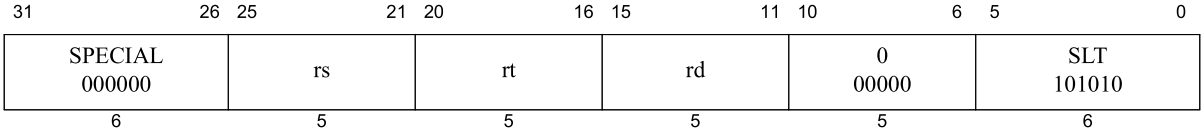
```
I:  temp ← GPR[rs]
I+1:if Config1CA = 0 then
       PC ← temp
    else
       PC ← tempGPRLEN-1..1 || 0
       ISAMode ← temp0
    endif
```

Exceptions:

None

Programming Notes:

Software should use the value 31 for the *rs* field of the instruction word on return from a JAL, JALR, or BGEZAL, and should use a value other than 31 for remaining uses of JR.



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 and 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:

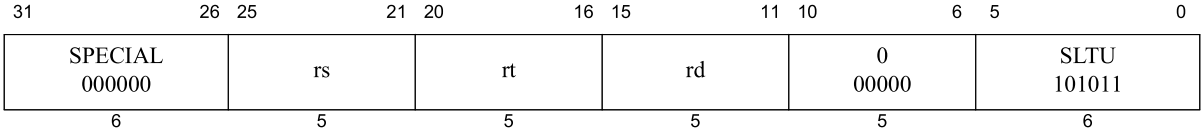
None

Operation:

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

Exceptions:

None



Format: `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 and 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:

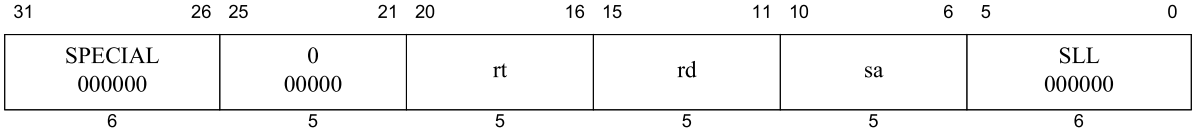
None

Operation:

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

Exceptions:

None



Format: `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
```

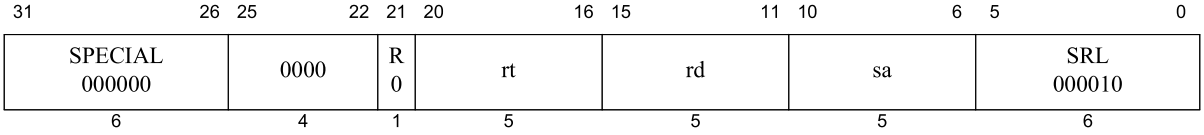
Exceptions:

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: `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:

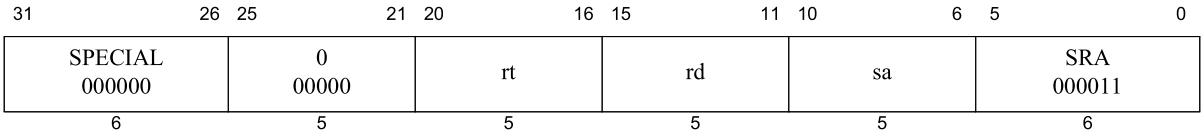
None

Operation:

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

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

Operation:

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

Exceptions:

None