APPENDIX C

Integer Instruction Set

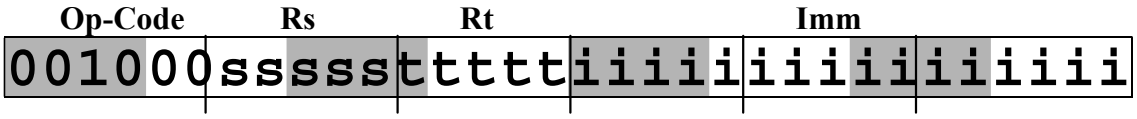
Add:

add Rd, Rs, Rt **# RF[Rd] = RF[Rs] + RF[Rt]**

Add contents of Reg.File[Rs] to Reg.File[Rt] and store result in Reg.File[Rd].
If overflow occurs in the two's complement number system, an exception is generated.

Add Immediate:

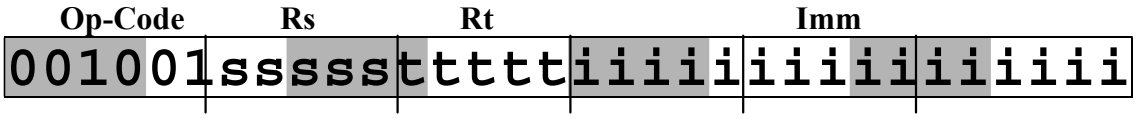
addi Rt, Rs, Imm **# RF[Rt] = RF[Rs] + Imm**



Add contents of Reg.File[Rs] to sign extended Imm value, store result in Reg.File [Rt].
If overflow occurs in the two's complement number system, an exception is generated.

Add Immediate Unsigned:

addiu Rt, Rs, Imm **# RF[Rt] = RF[Rs] + Imm**



Add contents of Reg.File[Rs] to sign extended Imm value, store result in Reg.File[Rt].
No overflow exception is generated.

Add Unsigned:

addu Rd, Rs, Rt **# RF[Rd] = RF[Rs] + RF[Rt]**

Add contents of Reg.File[Rs] to Reg.File[Rt] and store result in Reg.File [Rd].
No overflow exception is generated.

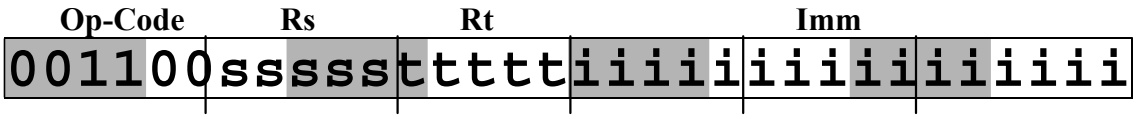
And:

and Rd, Rs, Rt **# RF[Rd] = RF[Rs] AND RF[Rt]**

Bitwise logically AND contents of Register File[Rs] with Reg.File[Rt] and store result in Reg.File[Rd].

And Immediate:

andi Rt, Rs, Imm **# RF[Rt] = RF[Rs] AND Imm**



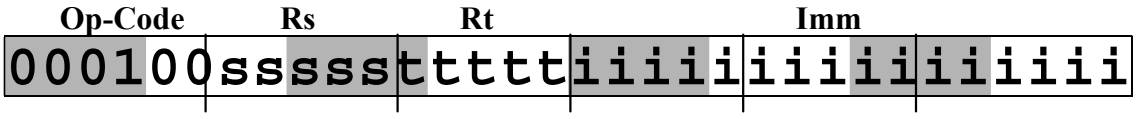
Bitwise logically AND contents of Reg.File[Rs] wih zero-extended Imm value and store result in Reg.File[Rt].

Branch Instructions

The immediate field contains a signed 16-bit value specifying the number of words away from the current program counter address to the location symbolically specified by the label. Since MIPS uses byte addressing, this word offset value in the immediate field is shifted left by two bits and added to the current contents of the program counter when a branch is taken. The SPIM assembler generates the offset from the address of the branch instruction. Whereas the assembler for an actual MIPS processor will generate the offset from the address of the instruction following the branch instruction since the program counter will have already been incremented by the time the branch instruction is executed.

Branch if Equal:

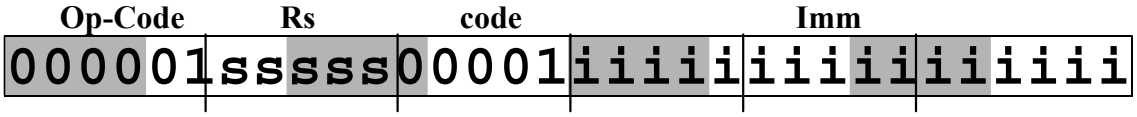
Beq Rs, Rt, Label **# If (RF[Rs] == RF[Rt])then PC = PC + Imm<< 2**



If Reg.File[Rs] is equal to Reg.File[Rt] then branch to label.

Branch if Greater Than or Equal to Zero:

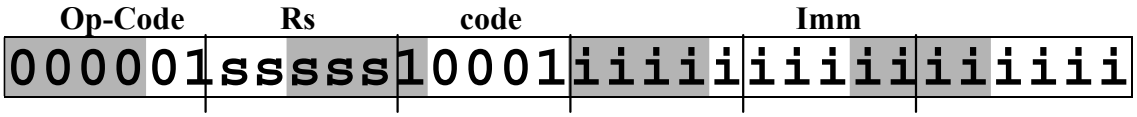
bgez Rs, Label **# If (RF[Rs] >= RF[0]) then PC = PC + Imm<< 2**



If Reg.File[Rs] is greater than or equal to zero, then branch to label.

Branch if Greater Than or Equal to Zero and Link:

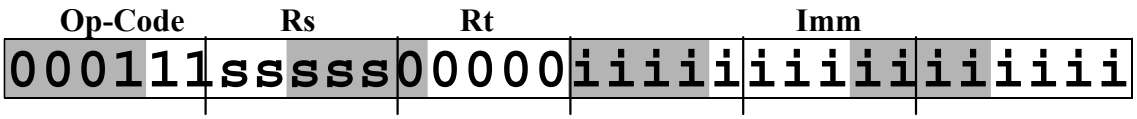
bgezal Rs, Label **# If(RF[Rs] >= RF[0])then**
{RF[$ra] = PC;
PC = PC + Imm<< 2 }



If Reg.File[Rs] is greater than or equal to zero, then save the return address in Reg.File[$rs] and branch to label. (Used to make conditional function calls)

Branch if Greater Than Zero:

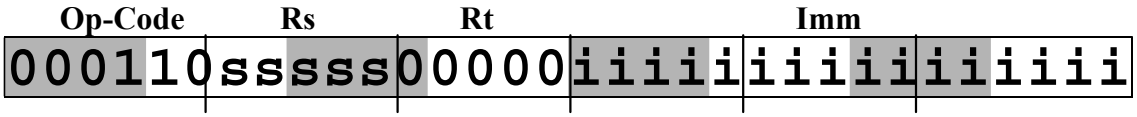
bgtz Rs, Label **# If (RF[Rs] > RF[0]) then PC = PC + Imm<< 2**



If Reg.File[Rs] is greater than zero, then branch to label.

Branch if Less Than or Equal to Zero:

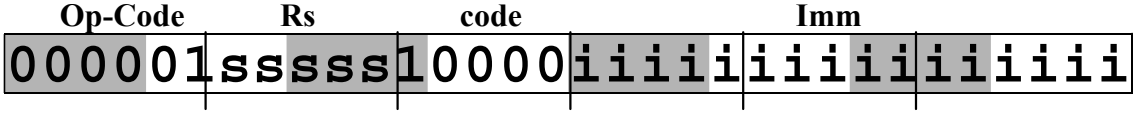
blez Rs, Label **# If (RF[Rs] <= RF[0]) then PC = PC + Imm<< 2**



If Reg.File[Rs] is less than or equal to zero, then branch to label.

Branch if Less Than Zero and Link:

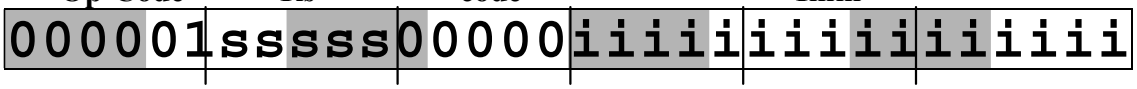
bltzal Rs, Label **# If RF[Rs] < RF[0] then**
{RF[$ra] = PC;
PC = PC + Imm<< 2 }



If Reg.File[Rs] is less than zero then save the return address in Reg.File[$rs] and branch to label.

Branch if Less Than Zero:

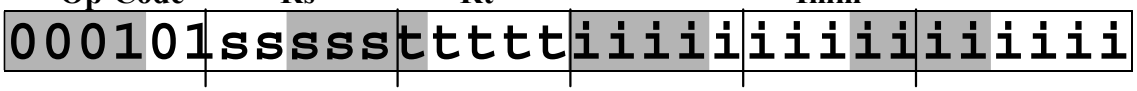
bltz Rs, Label **# If RF[Rs] < RF[0] then PC = PC + Imm<< 2**



If Reg.File[Rs] is less than zero then branch to label.

Branch if Not Equal:

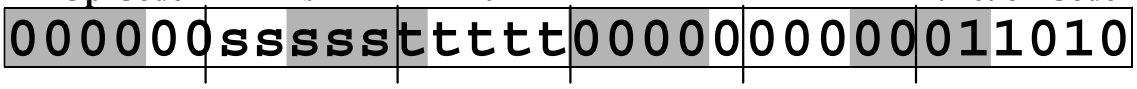
bne Rs, Rt, Label **# If RF[Rs] != RF[Rt] then PC = PC + Imm<< 2**



If Reg.File[Rs] is not equal to Reg.File[Rt] then branch to label.

Divide:

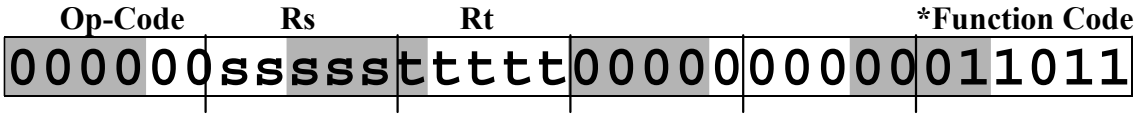
div Rs, Rt **# Low = Quotient (RF[Rs] / RF[Rt])**
 # High = Remainder (RF[Rs] / RF[Rt])



Divide the contents of Reg.File[Rs] by Reg.File[Rt]. Store the quotient in the LOW register, and store the remainder in the HIGH register. The sign of the quotient will be negative if the operands are of opposite signs. The sign of the remainder will be the same as the sign of the numerator, Reg.File[Rs]. No overflow exception occurs under any circumstances. It is the programmer's responsibility to test if the divisor is zero before executing this instruction, because the results are undefined when the divisor is zero. For some implementations of the MIPS architecture, it takes 38 clock cycles to execute the divide instruction.

Divide Unsigned:

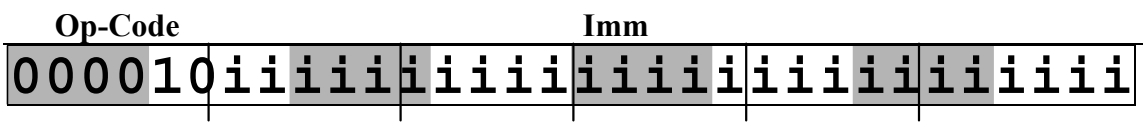
divu Rs, Rt **# Low = Quotient (RF[Rs] / RF[Rt])**
 # High = Remainder (RF[Rs] / RF[Rt])



Divide the contents of Reg.File[Rs] by Reg.File[Rt], treating both operands as unsigned values. Store the quotient in the LOW register, and store the remainder in the HIGH register. The quotient and remainder will always be positive values. No overflow exception occurs under any circumstances. It is the programmer's responsibility to test if the divisor is zero before executing this instruction, because the results are undefined when the divisor is zero. For some implementations of the MIPS architecture, it takes 38 clock cycles to execute the divide instruction.

Jump:

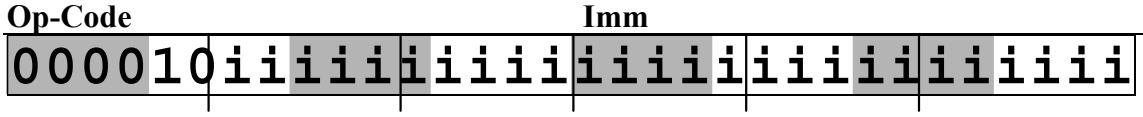
j Label **# PC = PC(31:28) | Imm<< 2**



Load the PC with an address formed by concatenating the first 4-bits of the current PC with the value in the 26-bit immediate field shifted left 2-bits.

Jump and Link: (Use this instructions to make function calls.

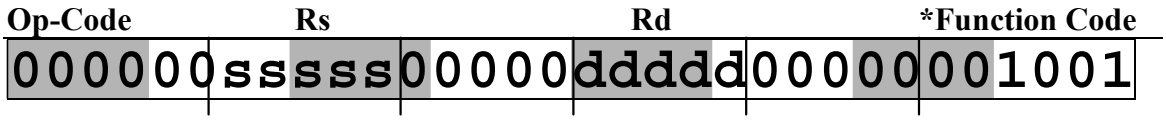
jal Label **# RF[$ra] = PC; PC = PC(31:28) | Imm<< 2**



Save the current value of the Program Counter (PC) in Reg.File[$ra], and load the PC with an address formed by concatenating the first 4-bits of the current PC with the value in the 26-bit immediate field shifted left 2-bits.

Jump and Link Register: (Use this instructions to make function calls.

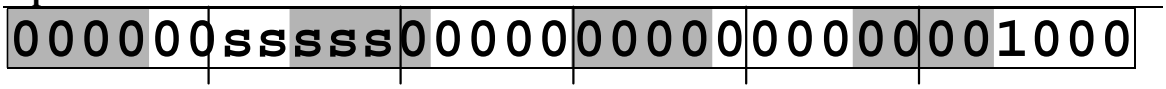
jalr Rd, Rs **# RF[Rd] = PC; PC = RF[Rs]**



Save the current value of the Program Counter (PC) in Reg.File[Rd] and load the PC with the address that is in Reg.File[Rs]. A programmer must insure a valid address has been loaded into Reg.File[Rs] before executing this instruction.

Jump Register: (Use this instructions to return from a function call.)

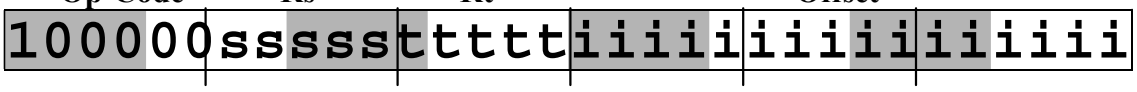
jr Rs **# PC = RF[Rs]**



Load the PC with an the address that is in Reg.File[Rs].

Load Byte:

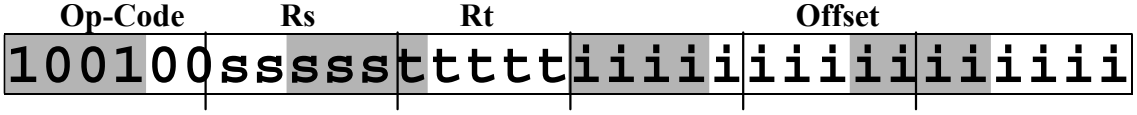
lb Rt, offset(Rs) **# RF[Rt] = Mem[RF[Rs] + Offset]**



The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. An 8-bit byte is read from memory at the effective address, sign extended and loaded into Reg.File[Rt].

Load Byte Unsigned:

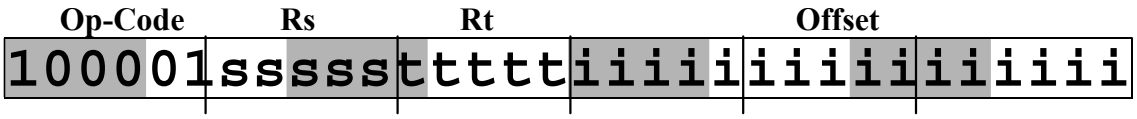
lbu Rt, offset(Rs) **# RF[Rt] = Mem[RF[Rs] + Offset]**



The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. An 8-bit byte is read from memory at the effective address, zero extended and loaded into Reg.File[Rt].

Load Halfword:

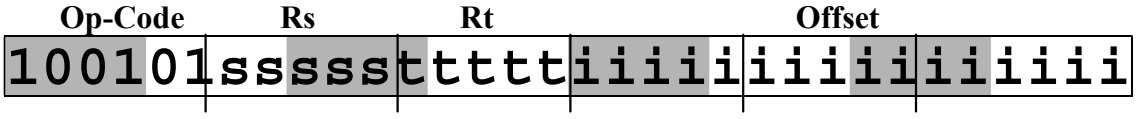
lh Rt, offset(Rs) **# RF[Rt] = Mem[RF[Rs] + Offset]**



The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. A 16-bit half word is read from memory at the effective address, sign extended and loaded into Reg.File[Rt]. If the effective address is an odd number, an address error exception occurs.

Load Halfword Unsigned:

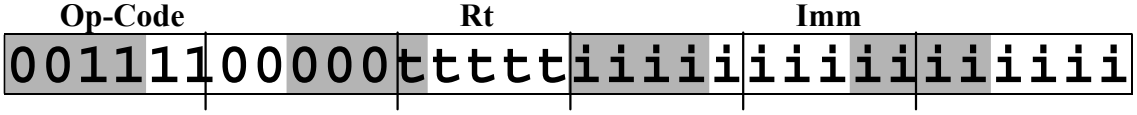
lhu Rt, offset(Rs) **# RF[Rt] = Mem[RF[Rs] + Offset]**



The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. A 16-bit half word is read from memory at the effective address, zero extended and loaded into Reg.File[Rt]. If the effective address is an odd number, an address error exception occurs.

Load Upper Immediate: (This instruction in conjunction with an OR immediate instruction is used to implement the Load Address pseudo instruction - la Label)

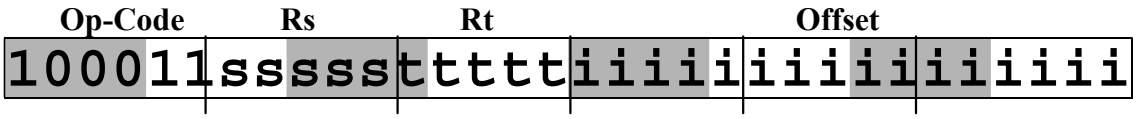
lui Rt, Imm **# RF[Rt] = Imm<<16 | 0x0000**



The 16-bit immediate value is shifted left 16-bits concatenated with 16 zeros and loaded into Reg.File[Rt].

Load Word:

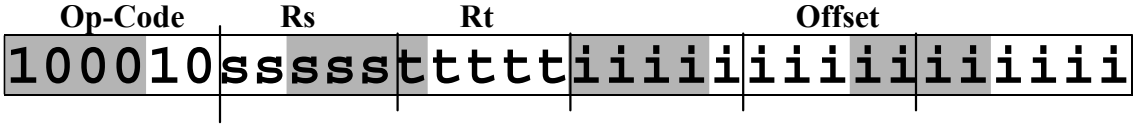
lw Rt, offset(Rs) **# RF[Rt] = Mem[RF[Rs] + Offset]**



The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. A 32-bit word is read from memory at the effective address and loaded into Reg.File[Rt]. If the least two significant bits of the effective address are not zero, an address error exception occurs. There are four bytes in a word, so word addresses must be binary numbers that are a multiple of four, otherwise an address error exception occurs.

Load Word Left:

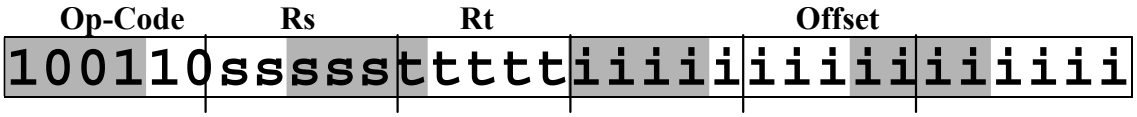
lwl Rt, offset(Rs) **# RF[Rt] = Mem[RF[Rs] + Offset]**



The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective byte address. From one to four bytes will be loaded left justified into Reg.File[Rt] beginning with the effective byte address then it proceeds toward a lower order byte in memory, until it reaches the lowest order byte of the word in memory. This instruction can be used in combination with the LWR instruction to load a register with four consecutive bytes from memory, when the bytes cross a boundary between two words.

Load Word Right:

lwr Rt, offset(Rs) **# RF[Rt] = Mem[RF[Rs] + Offset]**



The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective byte address. From one to four bytes will be loaded right justified into Reg.File[Rt] beginning with the effective byte address then it proceeds toward a higher order byte in memory, until it reaches the high order byte of the word in memory. This instruction can be used in combination with the LWL instruction to load a register with four consecutive bytes from memory, when the bytes cross a boundary between two words.

Move From High:

mfhi Rd **# RF[Rd] = HIGH**

Op-Code			Rd		Function Code
000000	0000000	00000	ddddd	00000	010000

Load Reg.File[Rd] with a copy of the value currently in special register HIGH.

Move From Low:

mflo Rd **# RF[Rd] = LOW**

Op-Code			Rd		Function Code
000000	0000000	00000	ddddd	00000	010010

Load Reg.File[Rd] with a copy of the value currently in special register LOW.

Move to High:

mthi Rs **# HIGH = RF[Rs]**

Op-Code	Rs				Function Code
000000	sssss	00000	00000	00000	010001

Load special register HIGH with a copy of the value currently in Reg.File[Rs].

Move to Low:

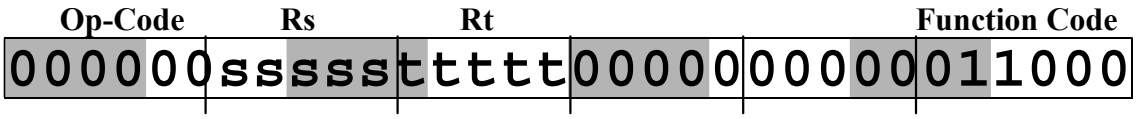
mtlo Rs **# LOW = RF[Rs]**

Op-Code	Rs				Function Code
000000	sssss	00000	00000	00000	010011

Load special register LOW with a copy of the value currently in Reg.File[Rs].

Multiply:

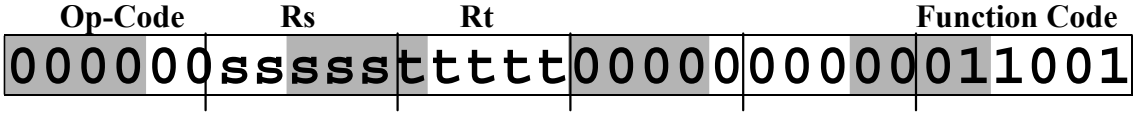
mult Rs, Rt **# High |Low = RF[Rs] * RF[Rt]**



Multiply the contents of Reg.File[Rs] by Reg.File[Rt] and store the lower 32-bits of the product in the LOW register, and store the upper 32-bits of the product in the HIGH register. The two operands are treated as two's complement numbers, the 64-bit product is negative if the signs of the two operands are different. No overflow exception occurs under any circumstances. For some implementations of the MIPS architecture it takes 32 clock cycles to execute the multiply instruction.

Multiply Unsigned:

multu Rs, Rt **# High |Low = RF[Rs] * RF[Rt]**



Multiply the contents of Reg.File[Rs] by Reg.File[Rt] and store the lower 32-bits of the product in the LOW register, and store the upper 32-bits of the product in the HIGH register. The two operands are treated as unsigned positive values. No overflow exception occurs under any circumstances. For some implementations of the MIPS architecture it takes 32 clock cycles to execute the multiply instruction.

NOR:

nor Rd, Rs, Rt **# RF[Rd] = RF[Rs] NOR RF[Rt]**

Bit wise logically NOR contents of Register File[Rs] with Reg.File[Rt] and store result in Reg.File[Rd].

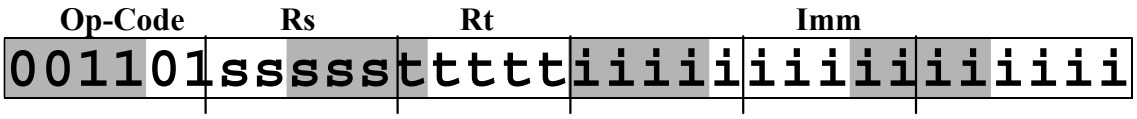
OR:

or Rd, Rs, Rt **# RF[Rd] = RF[Rs] OR RF[Rt]**

Bit wise logically OR contents of Register File[Rs] with Reg.File[Rt] and store result in Reg.File[Rd].

OR Immediate:

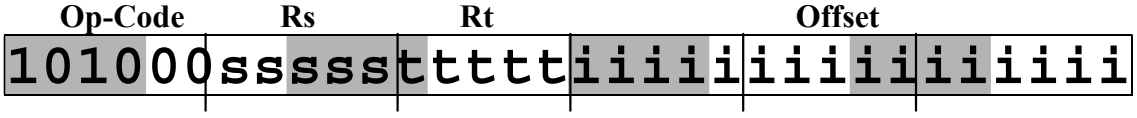
ori Rt, Rs, Imm **# RF[Rt] = RF[Rs] OR Imm**



Bit wise logically OR contents of Reg.File[Rs] wih zero extended Imm value and store result in Reg.File[Rt].

Store Byte:

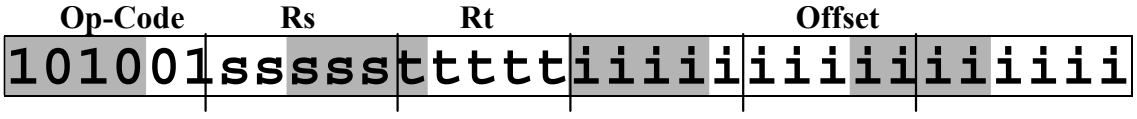
sb Rt, offset(Rs) **# Mem[RF[Rs] + Offset] = RF[Rt]**



The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. The least significant 8-bit byte in Reg.File[Rt] are stored in memory at the effective address.

Store Halfword:

sh Rt, offset(Rs) **# Mem[RF[Rs] + Offset] = RF[Rt]**



The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. The least significant 16-bits in Reg.File[Rt] are stored in memory at the effective address. If the effective address is an odd number, then an address error exception occurs.

Shift Left Logical:

sll Rd, Rt, sa **# RF[Rd] = RF[Rt] << sa**

The contents of Reg.File[Rt] are shifted left sa-bits & the result is stored in Reg.File[Rd].

Shift Left Logical Variable:

sllv Rd, Rt, Rs **# RF[Rd] = RF[Rt] << RF[Rs] amount**

The contents of Reg.File[Rt] are shifted left by the number of bits specified by the low order 5-bits of Reg.File[Rs], and the result is stored in Reg.File[Rd].

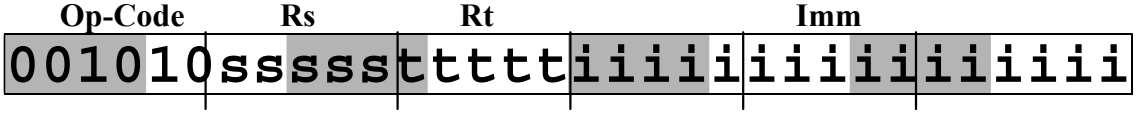
Set on Less Than: (Used in branch macro instructions)

slt Rd, Rs, Rt **# if (RF[Rs] < RF[Rt]) then RF[Rd] =1 else RF[Rd] = 0**

If the contents of Reg.File[Rs] are less than the contents of Reg.File[Rt], then Reg.File[Rd] is set to one, otherwise Reg.File[Rd] is set to zero; assuming the two's complement number system representation.

Set on Less Than Immediate: (Used in branch macro instructions)

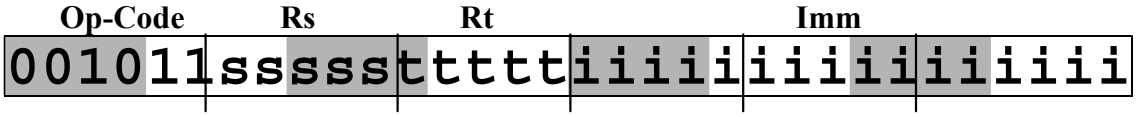
slti Rt, Rs, Imm **# if (RF[Rs] < Imm) then RF[Rt] =1 else RF[Rt] = 0**



If the contents of Reg.File[Rs] are less than the sign-extended immediate value then Reg.File[Rt] is set to one, otherwise Reg.File[Rt] is set to zero; assuming the two's complement number system representation.

Set on Less Than Immediate Unsigned: (Used in branch macro instructions)

sltiu Rt, Rs, Imm **# if (RF[Rs] < Imm) then RF[Rt] =1 else RF[Rt] = 0**



If the contents of Reg.File[Rs] are less than the sign-extended immediate value, then Reg.File[Rt] is set to one, otherwise Reg.File[Rt] is set to zero; assuming an unsigned number representation (only positive values).

Set on Less Than Unsigned: (Used in branch macroinstructions)

sltu Rd, Rs, Rt **# if (RF[Rs] < RF[Rt]) then RF[Rd] =1 else RF[Rd] = 0**

If the contents of Reg.File[Rs] are less than the contents of Reg.File[Rt], then Reg.File[Rd] is set to one, otherwise Reg.File[Rd] is set to zero; assuming an unsigned number representation (only positive values).

Shift Right Arithmetic:

sra Rd, Rt, sa **# RF[Rd] = RF[Rt] >> sa**

The contents of Reg.File[Rt] are shifted right sa-bits, sign-extending the high order bits, and the result is stored in Reg.File[Rd].

Shift Right Arithmetic Variable:

srav Rd, Rt, Rs **# RF[Rd] = RF[Rt] >> RF[Rs] amount**

The contents of Reg.File[Rt] are shifted right, sign-extending the high order bits, by the number of bits specified by the low order 5-bits of Reg.File[Rs], and the result is stored in Reg.File[Rd].

Shift Right Logical:

srl Rd, Rt, sa **# RF[Rd] = RF[Rt] >> sa**

The contents of Reg.File[Rt] are shifted right sa-bits, inserting zeros into the high order bits, the result is stored in Reg.File[Rd].

Shift Right Logical Variable:

srlv Rd, Rt, Rs **# RF[Rd] = RF[Rt] >> RF[Rs] amount**

The contents of Reg.File[Rt] are shifted right, inserting zeros into the high order bits, by the number of bits specified by the low order 5-bits of Reg.File[Rs], and the result is stored in Reg.File[Rd].

Subtract:

sub Rd, Rs, Rt **# RF[Rd] = RF[Rs] - RF[Rt]**

Subtract contents of Reg.File[Rt] from Reg.File[Rs] and store result in Reg.File[Rd]. If overflow occurs in the two's complement number system, an exception is generated.

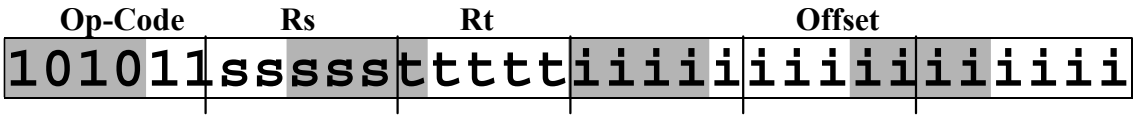
Subtract Unsigned:

subu Rd, Rs, Rt **# RF[Rd] = RF[Rs] - RF[Rt]**

Subtract contents of Reg.File[Rt] from Reg.File[Rs] and store result in Reg.File[Rd]. No overflow exception is generated.

Store Word:

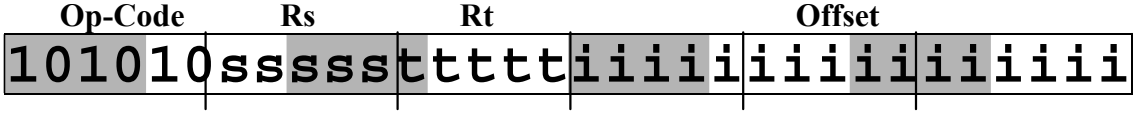
sw Rt, offset(Rs) **# Mem[RF[Rs] + Offset] = RF[Rt]**



The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. The contents of Reg.File[Rt] are stored in memory at the effective address. If the least two significant bits of the effective address are not zero, an address error exception occurs. There are four bytes in a word, so word addresses must be binary numbers that are a multiple of four, otherwise an address error exception occurs.

Store Word Left:

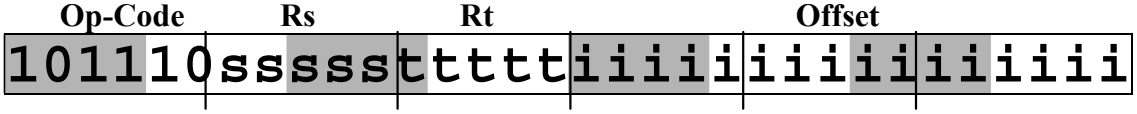
swl Rt, offset(Rs) **# Mem[RF[Rs] + Offset] = RF[Rt]**



The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. From one to four bytes will be stored left justified into memory beginning with the most significant byte in Reg.File[Rt], then it proceeds toward a lower order byte in memory, until it reaches the lowest order byte of the word in memory. This instruction can be used in combination with the SWR instruction, to store the contents of a register into four consecutive bytes of memory, when the bytes cross a boundary between two words.

Store Word Right:

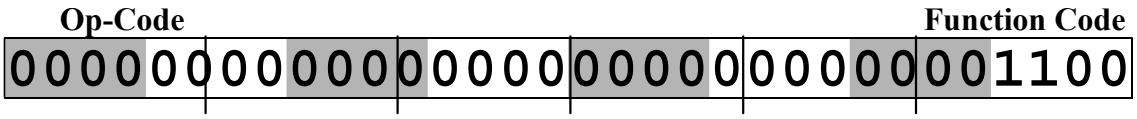
swr Rt, offset(Rs) **# Mem[RF[Rs] + Offset] = RF[Rt]**



The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. From one to four bytes will be stored right justified into memory beginning with the least significant byte in Reg.File[Rt], then it proceeds toward a higher order byte in memory, until it reaches the highest order byte of the word in memory. This instruction can be used in combination with the SWL instruction, to store the contents of a register into four consecutive bytes of memory, when the bytes cross a boundary between two words.

System Call: (Used to call system services to perform I/O)

syscall



A user program exception is generated.

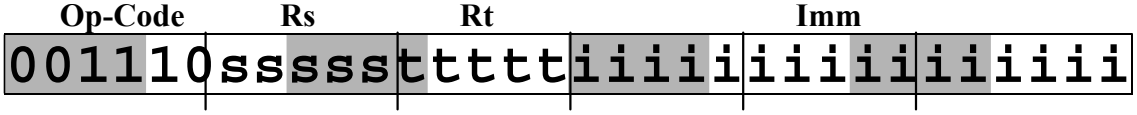
Exclusive OR:

xor Rd, Rs, Rt **# RF[Rd] = RF[Rs] XOR RF[Rt]**

Bit wise logically Exclusive-OR contents of Register File[Rs] with Reg.File[Rt] and store result in Reg.File[Rd].

Exclusive OR Immediate:

xori Rt, Rs, Imm **# RF[Rt] = RF[Rs] XOR Imm**



Bit wise logically Exclusive-OR contents of Reg.File[Rs] with zero extended Imm value and store result in Reg.File[Rt]
