计算机组成

—— MIPS汇编

（可参考后附指令集）

第1题

填空

(a) 将10进制数35转换为8位二进制数是 _____

(b) 将二进制数00010101转换为16进制数是 _____

(c) 将10进制数-35转换为8位二进制补码是 _____

(d) 将8进制数 204 转换为10进制数是 _____

(e) 请判断以下两个补码表示的二进制数做二进制加法后是否溢出：__

01101110

00011010

(f) 对8位16进制数0x88做符号扩展成16位数是：0x_____

(g) 下列代码段存储在内存中，起始地址为 0x00012344，分支指令执行后PC的两个可能的值分别是： 0x_____ 和 0x _____。同时，请在注释位置用伪代码形式对每条指令做出描述。

```
loop:   lw      $t0, 0($a0)     # _____
        addi    $a0, $a0, 4     # _____
        andi    $t1, $t0, 1     # _____
        beqz    $t1,  loop      # _____
```

第2题

将下列汇编语言指令翻译成机器语言代码，以16进制表示

```
loop:   addu    $a0, $0, $t0        # _____
        ori     $v0, $0, 4          # _____
        syscall                     # _____
        addi    $t0, $t0, -1        # _____
        bnez    $t0, loop           # _____
        andi    $s0, $s7, 0xffc0    # _____
        or      $a0, $t7, $s0       # _____
        sb      $a0, 4($s6)         # _____
        srl     $s7, $s7, 4         # _____
```

第3题

写一个MIPS汇编程序，要求对内存以“example100”为标签（label）的数据段中前100个字（words）的数据求和，并将结果存入紧跟在这100个字之后的内存中。

第4题

写一段MIPS汇编语言代码，将内存中“SRC”标签开始的100个字的一块数据转移到内存中另一块以“DEST”标签开始的空间中。

第5题

写一个MIPS函数ABS，通过$a0传入一个32位整数，将这个数的绝对值存回$a0。再写一段主程序，调用两次ABS并输出结果，每次传给ABS的数不同。

第6题

写一个函数FIB(N, &array)向内存中的一个数组（array）存入斐波那契数列的前N个元素。N和array的地址分别通过$a0和$a1传递进来。斐波那契数列的前几个元素是：1，1，2，3，5，8，13，.....

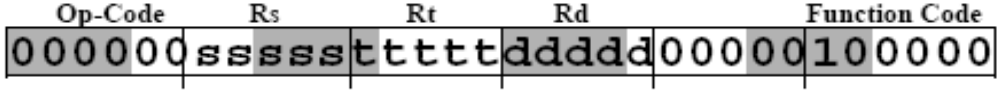
第7题

写一个函数，从$a0，$a1，和$a2中接受传递过来的3个32位整数，按从小到大排序后存回$a0-$a2。

Integer Instruction Set

Add:

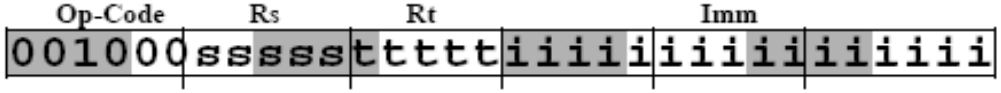
add Rd, Rs, Rt # RF[Rd] = RF[Rs] + RF[Rt]



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:

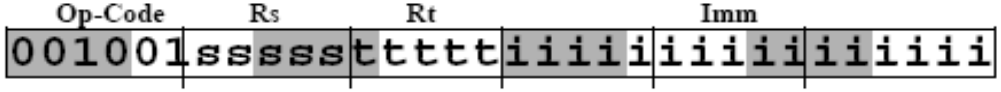
addi Rt, Rs, Imm # RF[Rt] = RF[Rs] + Imm



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:

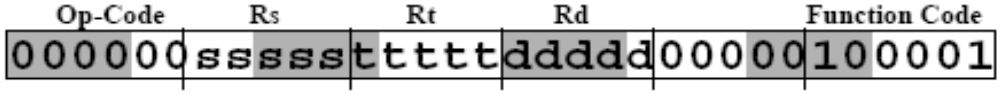
addiu Rt, Rs, Imm # RF[Rt] = RF[Rs] + Imm



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]

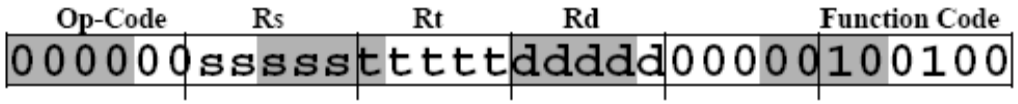


Add contents of Reg.File[Rs] to Reg.File[Rt] and store result in Reg.File [Rd].
No overflow exception is generated.

附录A（续1）

And:

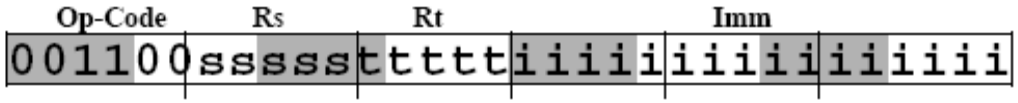
and Rd, Rs, Rt # RF[Rd] = RF[Rs] AND RF[Rt]



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



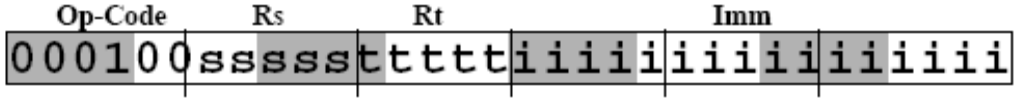
Bitwise logically AND contents of Reg.File[Rs] wih zero-extended Imm value and store result in Reg.File[Rt].

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

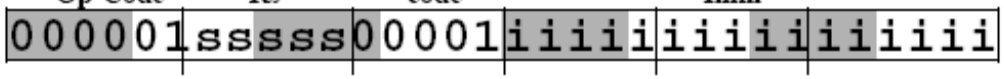


If Reg.File[Rs] is equal to Reg.File[Rt] then branch to label.

附录A（续2）

Branch if Greater Than or Equal to Zero:

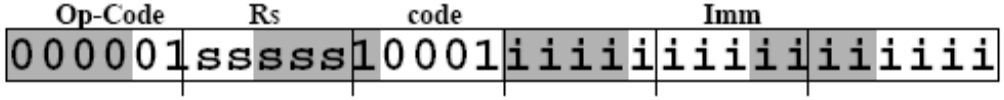
bgez Rs, Label # If (RF[Rs] >= RF[0]) then PC = PC + Imm<< 2



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:

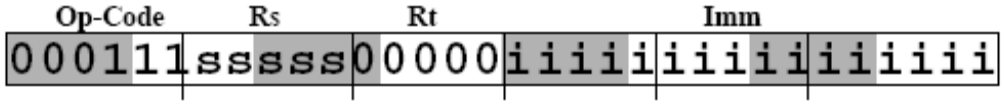
bgezal Rs, Label # If(RF[Rs] >= RF[0])then
{RF[$ra] = PC;
PC = PC + Imm<< 2 }



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:

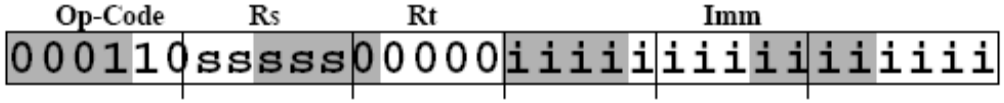
bgtz Rs, Label # If (RF[Rs] > RF[0]) then PC = PC + Imm<< 2



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

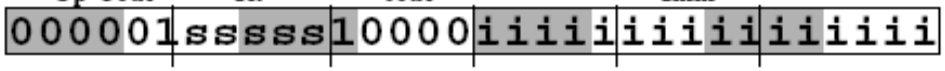


If Reg.File[Rs] is less than or equal to zero, then branch to label.

附录A（续3）

Branch if Less Than Zero and Link:

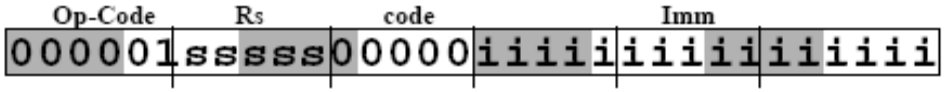
bltzal Rs, Label # If RF[Rs] < RF[0] then {RF[$ra] = PC; PC = PC + Imm<< 2 }



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:

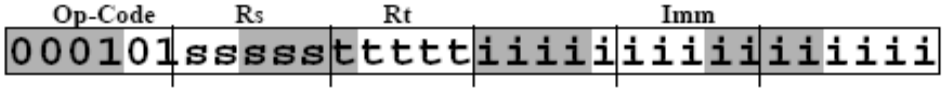
bltz Rs, Label # If RF[Rs] < RF[0] then PC = PC + Imm<< 2



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

Branch if Not Equal:

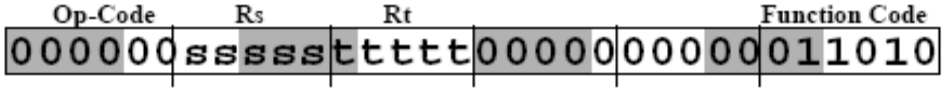
bne Rs, Rt, Label # If RF[Rs] != RF[Rt] then PC = PC + Imm<< 2



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])

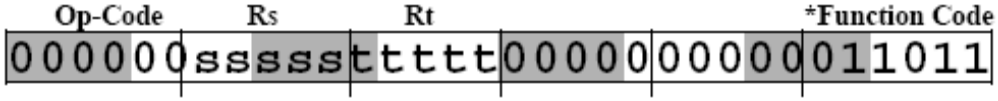


Divide the contents of Reg.File[Rs] by Reg.File[Rt]. Store the quotient in the LOW register, and store the remainder in the HIGH register. The sign of the quotient will be negative if the operands are of opposite signs. The sign of the remainder will be the same as the sign of the numerator, Reg.File[Rs]. 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.

附录A（续4）

Divide Unsigned:

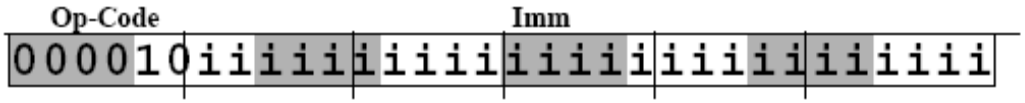
divu Rs, Rt **# Low = Quotient (RF[Rs] / RF[Rt])**
 # High = Remainder (RF[Rs] / RF[Rt])



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:

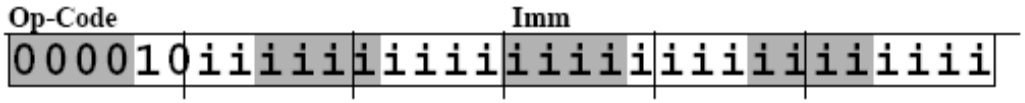
j Label **# PC = PC(31:28) | Imm<< 2**



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**

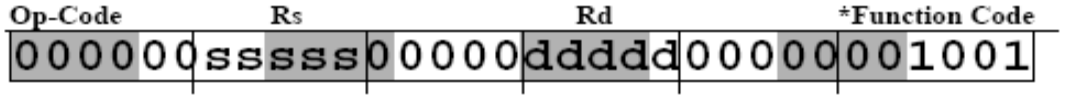


Save the current value of the Program Counter (PC) in Reg.File[$ra], and load the PC with an address formed by concatenating the first 4-bits of the current PC with the value in the 26-bit immediate field shifted left 2-bits.

附录A（续5）

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

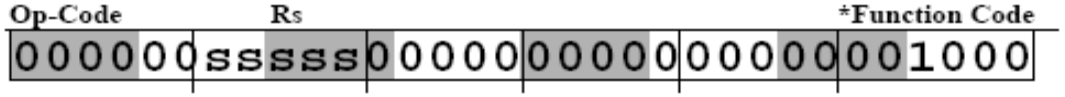
jalr Rd, Rs # RF[Rd] = PC; PC = RF[Rs]



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.)

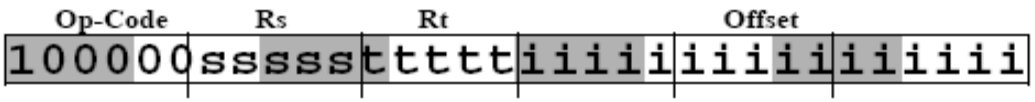
jr Rs # PC = RF[Rs]



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

Load Byte:

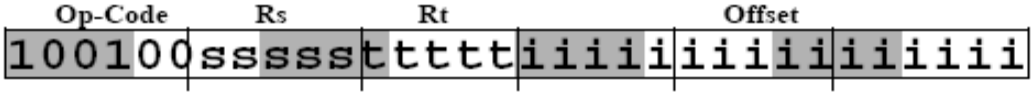
lb Rt, offset(Rs) # RF[Rt] = Mem[RF[Rs] + Offset]



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]

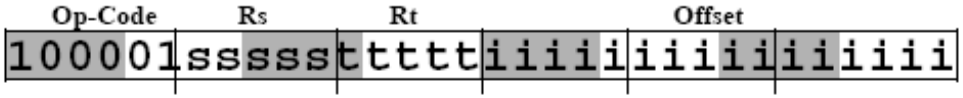


The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. An 8-bit byte is read from memory at the effective address, zero extended and loaded into Reg.File[Rt].

附录A（续6）

Load Halfword:

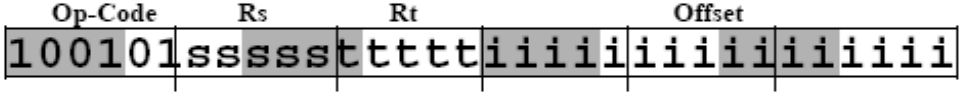
lh Rt, offset(Rs) **# RF[Rt] = Mem[RF[Rs] + Offset]**



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:

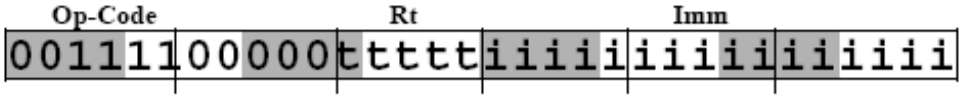
lhu Rt, offset(Rs) **# RF[Rt] = Mem[RF[Rs] + Offset]**



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**

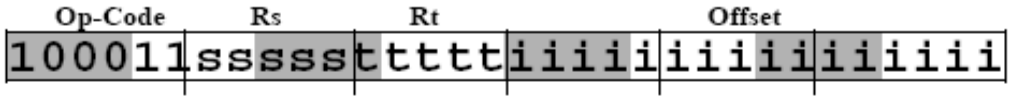


The 16-bit immediate value is shifted left 16-bits concatenated with 16 zeros and loaded into Reg.File[Rt].

附录A（续7）

Load Word:

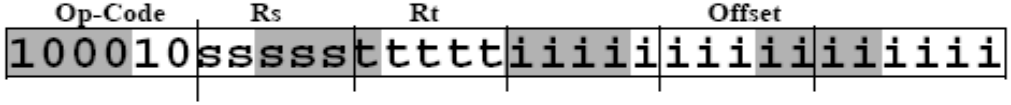
lw Rt, offset(Rs) **# RF[Rt] = Mem[RF[Rs] + Offset]**



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:

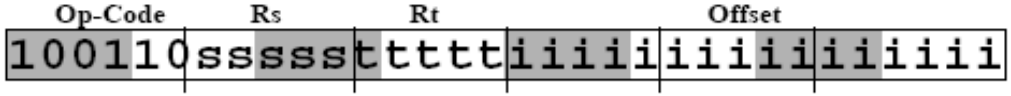
lwl Rt, offset(Rs) **# RF[Rt] = Mem[RF[Rs] + Offset]**



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.

附录A（续8）

Move From High:

mfhi Rd # RF[Rd] = HIGH

Op-Code				Rd		Function Code
000000	00000	00000	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	00000	00000	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	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]

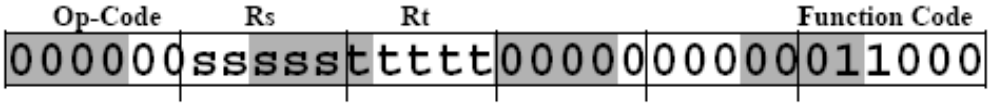
Op-Code	Rs					Function Code
000000	sssss	00000	00000	00000	00000	010011

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

附录A（续9）

Multiply:

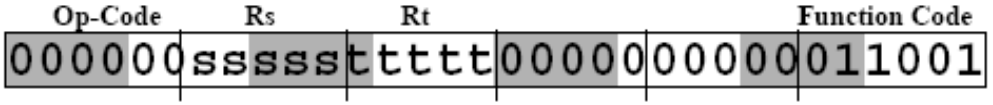
mult Rs, Rt # High |Low = RF[Rs] * RF[Rt]



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:

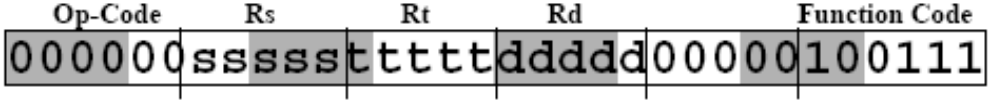
multu Rs, Rt # High |Low = RF[Rs] * RF[Rt]



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]

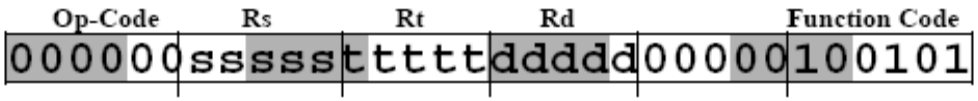


Bit wise logically NOR contents of Register File[Rs] with Reg.File[Rt] and store result in Reg.File[Rd].

附录A（续10）

OR:

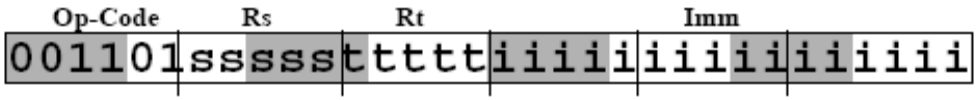
or Rd, Rs, Rt # RF[Rd] = RF[Rs] OR RF[Rt]



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

OR Immediate:

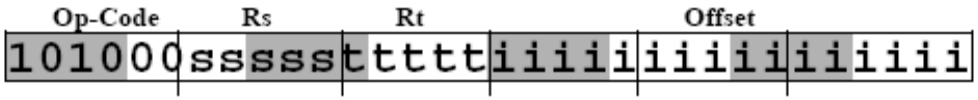
ori Rt, Rs, Imm # RF[Rt] = RF[Rs] OR Imm



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

Store Byte:

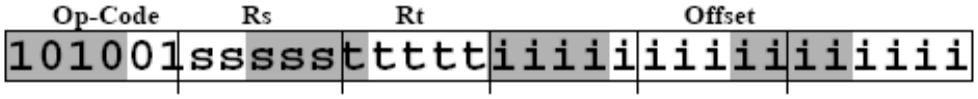
sb Rt, offset(Rs) # Mem[RF[Rs] + Offset] = RF[Rt]



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]

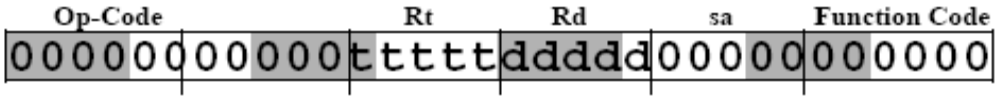


The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. The least significant 16-bits in Reg.File[Rt] are stored in memory at the effective address. If the effective address is an odd number, then an address error exception occurs.

附录A（续11）

Shift Left Logical:

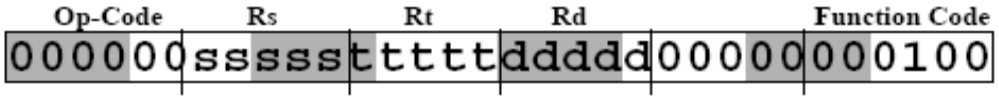
sll Rd, Rt, sa # RF[Rd] = RF[Rt] << sa



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

Shift Left Logical Variable:

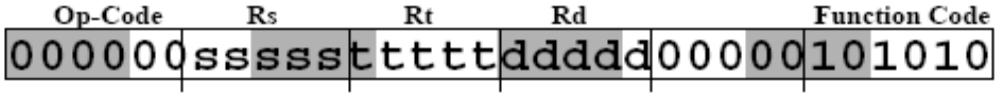
sllv Rd, Rt, Rs # RF[Rd] = RF[Rt] << RF[Rs] amount



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)

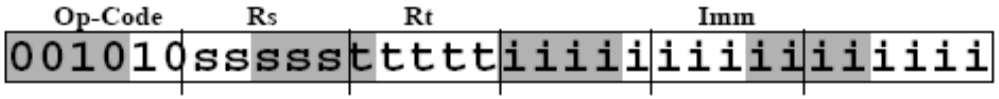
slt Rd, Rs, Rt # if (RF[Rs] < RF[Rt]) then RF[Rd] =1 else RF[Rd] = 0



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

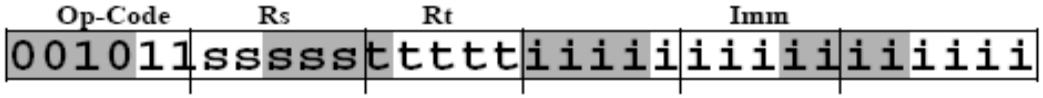


If the contents of Reg.File[Rs] are less than the sign-extended immediate value then Reg.File[Rt] is set to one, otherwise Reg.File[Rt] is set to zero; assuming the two's complement number system representation.

附录A（续12）

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

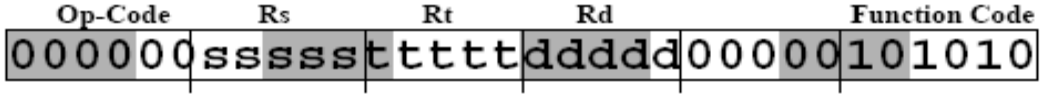
sltiu Rt, Rs, Imm # if (RF[Rs] < Imm) then RF[Rt] =1 else RF[Rt] = 0



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)

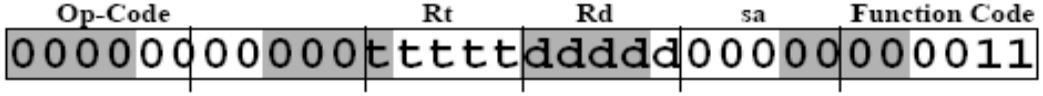
sltu Rd, Rs, Rt # if (RF[Rs] < RF[Rt]) then RF[Rd] =1 else RF[Rd] = 0



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:

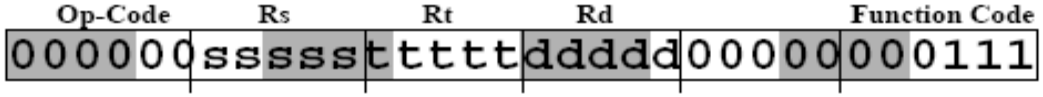
sra Rd, Rt, sa # RF[Rd] = RF[Rt] >> sa



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

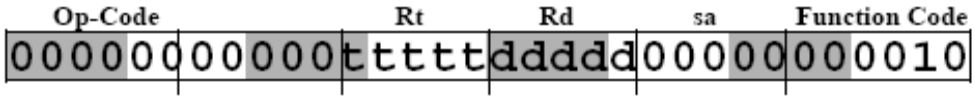


The contents of Reg.File[Rt] are shifted right, sign-extending the high order bits, by the number of bits specified by the low order 5-bits of Reg.File[Rs], and the result is stored in Reg.File[Rd].

附录A（续13）

Shift Right Logical:

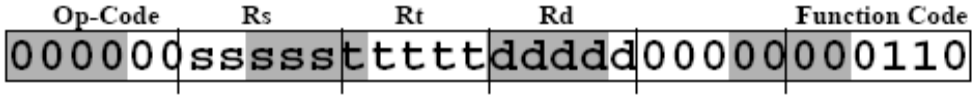
srl Rd, Rt, sa # RF[Rd] = RF[Rt] >> sa



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:

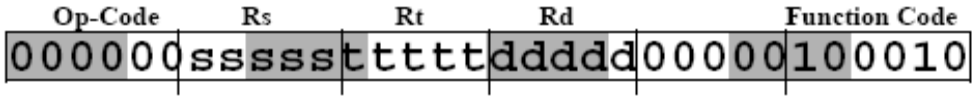
srlv Rd, Rt, Rs # RF[Rd] = RF[Rt] >> RF[Rs] amount



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:

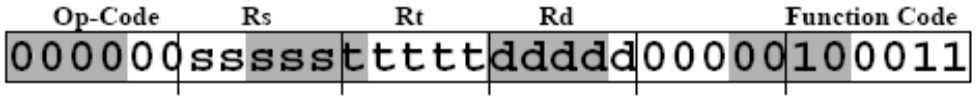
sub Rd, Rs, Rt # RF[Rd] = RF[Rs] - RF[Rt]



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]

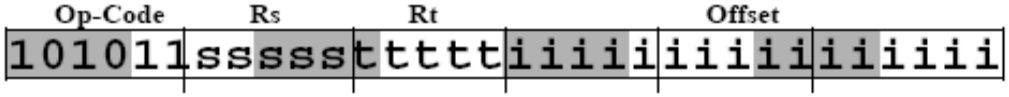


Subtract contents of Reg.File[Rt] from Reg.File[Rs] and store result in Reg.File[Rd]. No overflow exception is generated.

附录A（续14）

Store Word:

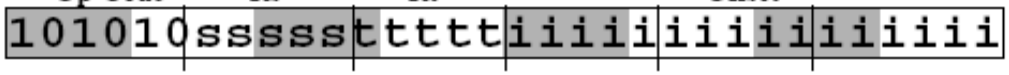
sw Rt, offset(Rs) **# Mem[RF[Rs] + Offset] = RF[Rt]**



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:

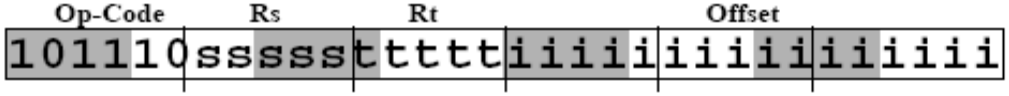
swl Rt, offset(Rs) **# Mem[RF[Rs] + Offset] = RF[Rt]**



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]**

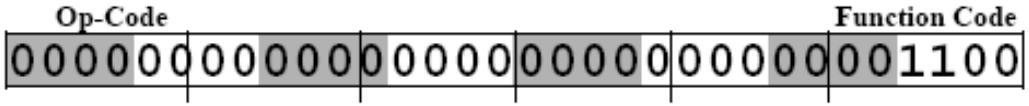


The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. From one to four bytes will be stored right justified into memory beginning with the least significant byte in Reg.File[Rt], then it proceeds toward a higher order byte in memory, until it reaches the highest order 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.

附录A（续15）

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

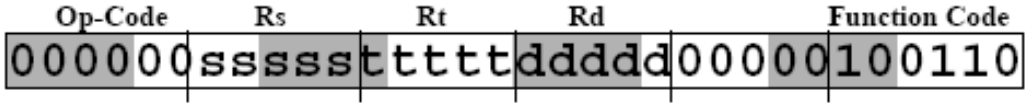
syscall



A user program exception is generated.

Exclusive OR:

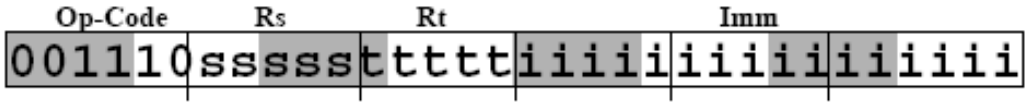
xor Rd, Rs, Rt # RF[Rd] = RF[Rs] XOR RF[Rt]



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]

Macro Instructions

Name	Actual Code	Space/Time
Absolute Value:		

abs Rd, Rs	addu Rd, $0, Rs
	bgez Rs, 1
	sub Rd, $0, Rs
		3/3

Branch if Equal to Zero:		
beqz Rs, Label	beq Rs, $0, Label	**1/1**
Branch if Greater than or Equal :		
bge Rs, Rt, Label	slt $at, Rs, Rt beq $at, $0, Label	**2/2**
If Reg.File[Rs] >= Reg.File[Rt] branch to Label Used to compare values represented in the two's complement number system.		
Branch if Greater than or Equal Unsigned		
bgeu Rs, Rt, Label	sltu $at, Rs, Rt beq $at, $0, Label	**2/2**
If Reg.File[Rs] >= Reg.File[Rt] branch to Label Used to compare addresses (unsigned values).		
Branch if Greater Than:		
bgt Rs, Rt, Label	slt $at, Rt, Rs bne $at, $0, Label	**2/2**
If Reg.File[Rs] > Reg.File[Rt] branch to Label Used to compare values represented in the two's complement number system.		
Branch if Greater Than Unsigned:		
bgtu Rs, Rt, Label	sltu $at, Rt, Rs bne $at, $0, Label	**2/2**
If Reg.File[Rs] > Reg.File[Rt] branch to Label Used to compare addresses (unsigned values).		
Branch if Less Than or Equal:		
ble Rs, Rt, Label	slt $at, Rt, Rs beq $at, $0, Label	**2/2**
If Reg.File[Rs] <= Reg.File[Rt] branch to Label Used to compare values represented in the two's complement number system.		

附录B（续1）

Branch if Less Than or Equal Unsigned:

bleu Rs, Rt, Label		sltu $at, Rt, Rs beq $at, $0, Label	**2/2**

If Reg.File[Rs] <= Reg.File[Rt] branch to Label
Used to compare addresses (unsigned values).

Branch if Less Than:

blt Rs, Rt, Label		slt $at, Rs, Rt bne $at, $0, Label	**2/2**

If Reg.File[Rs] < Reg.File[Rt] branch to Label
Used to compare values represented in the two's complement number system

Branch if Less Than Unsigned:

bltu Rs, Rt, Label		sltu $at, Rs, Rt bne $at, $0, Label	**2/2**

If Reg.File[Rs] < Reg.File[Rt] branch to Label
Used to compare addresses (unsigned values).

Branch if Not Equal to Zero:

bnez Rs, Label		bne Rs, $0, Label	**1/1**

Branch Unconditional

b Label		bgez $0, Label	**1/1**

Divide:

div Rd, Rs, Rt		bne Rt, $0, ok break $0	**4/41**
	ok:	div Rs, Rt mflo Rd	

Divide Unsigned:

divu Rd, Rs, Rt		bne Rt, $0, ok break $0	**4/41**
	ok:	divu Rs, Rt mflo Rd	

Load Address:

la Rd, Label		lui $at, Upper 16-bits of Label ori Rd, $at, Lower 16-bits of Label	**2/2**

Used to initialize pointers.

Load Immediate:

li Rd, value		lui $at, Upper 16-bits of value ori Rd, $at, Lower 16-bits of value	**2/2**

Initialize registers with negative constants and values greater than 32767.

附录B（续2）

Load Immediate:

li Rd, value	ori Rt, $0, value	**1/1**

Initialize registers with positive constants less than 32768.

Move:

move Rd, Rs	addu Rd, $0, Rs	**1/1**
mul Rd, Rs, Rt	mult Rs, Rt mflo Rd	**2/33**

Multiply (with overflow exception):

mulo Rd, Rs, Rt	mult Rs, Rt mfhi $at mflo Rd sra Rd, Rd, 31 beq $at, Rd, ok break $0 ok: mflo Rd	**7/37**

Multiply Unsigned (with overflow exception):

mulou Rd, Rs, Rt	multu Rs, Rt mfhi $at beq $at, $0, ok ok: break $0 mflo Rd	**5/35**

Negate:

neg Rd, Rs	sub Rd, $0, Rs	**1/1**

Two's complement negation. An exception is generated when there is an attempt to negate the most negative value: 2,147,483,648.

Negate Unsigned:

negu Rd, Rs	subu Rd, $0, Rs	**1/1**

Nop:

nop	or $0, $0, $0	**1/1**

Used to solve problems with hazards in the pipeline.

Not:

not Rd, Rs	nor Rd, Rs, $0	**1/1**

A bit-wise Boolean complement.

Remainder:

rem Rd, Rs, Rt	bne Rt, $0, 8 break $0 div Rs, Rt mfhi Rd	**4/40**

附录B（续3）

Remainder Unsigned:

remu Rd, Rs, Rt		bne Rt, $0, ok	4/40
		break $0	
	ok:	divu Rs, Rt	
		mfhi Rd	

Rotate Left Variable:

rol Rd, Rs, Rt		subu $at, $0, Rt	4/4
		srlv $at, Rs, $at	
		sllv Rd, Rs, Rt	
		or Rd, Rd, $at	

The lower 5-bits in Rt specifys the shift amount.

Rotate Right Variable:

ror Rd, Rs, Rt		subu $at, $0, Rt	4/4
		sllv $at, Rs, $at	
		srlv Rd, Rs, Rt	
		or Rd, Rd, $at	

Rotate Left Constant:

rol Rd, Rs, sa		srl $at, Rs, 32-sa	3/3
		sll Rd, Rs, sa	
		or Rd, Rd, $at	

Rotate Right Constant:

ror Rd, Rs, sa		sll $at, Rs, 32-sa	3/3
		srl Rd, Rs, sa	
		or Rd, Rd, $at	

Set if Equal:

seq Rd, Rs, Rt		beq Rt, Rs, yes	4/4
		ori Rd, $0, 0	
		beq $0, $0, skip	
	yes:	ori Rd, $0, 1	
	skip:		

Set if Greater Than or Equal:

sge Rd, Rs, Rt		bne Rt, Rs, yes	4/4
		ori Rd, $0, 1	
		beq $0, $0, skip	
	yes:	slt Rd, Rt, Rs	
	skip:		

Set if Greater Than or Equal Unsigned:

sgeu Rd, Rs, Rt		bne Rt, Rs, yes	4/4
		ori Rd, $0, 1	
		beq $0, $0, skip	
	yes:	sltu Rd, Rt, Rs	
	skip:		

附录B（续4）

Set if Greater Than:		
sgt Rd, Rs, Rt	slt Rd, Rt, Rs	**1/1**
Set if Greater Than Unsigned:		
sgtu Rd, Rs, Rt	sltu Rd, Rt, Rs	**1/1**
Set if Less Than or Equal:		
sle Rd, Rs, Rt	bne Rt, Rs, yes ori Rd, $0, 1 beq $0, $0, skip yes: slt Rd, Rs, Rt skip:	**4/4**
Set if Less Than or Equal Unsigned:		
sleu Rd, Rs, Rt	bne Rt, Rs, yes ori Rd, $0, 1 beq $0, $0, skip yes: sltu Rd, Rs, Rt skip:	**4/4**
Set if Not Equal:		
sne Rd, Rs, Rt	beq Rt, Rs, yes ori Rd, $0, 1 beq $0, $0, skip yes: ori Rd, $0, 0 skip:	**4/4**
Unaligned Load Halfword Unsigned:		
ulh Rd, 3(Rs)	lb Rd, 4(Rs) lbu $at, 3(Rs) sll Rd, Rd, 8 or Rd, Rd, $at	**4/4**
Unaligned Load Halfword:		
ulhu Rd, 3(Rs)	lbu Rd, 4(Rs) lbu $at, 3(Rs) sll Rd, Rd, 8 or Rd, Rd, $at	**4/4**
Unaligned Load Word:		
ulw Rd, 3(Rs	lwl Rd, 6(Rs) lwr Rd, 3(Rs)	**2/2**
Unaligned Store Halfword:		
ush Rd, 3(Rs)	sb Rd, 3(Rs) srl $at, Rd, 8 sb $at, 4(Rs)	**3/3**
Unaligned Store Word:		
usw Rd, 3(Rs)	swl Rd, 6(Rs) swr Rd, 3(Rs)	**2/2**