Integer Arithmetic:
Multiply, Divide, and Bitwise Operations

Slides revised 3/21/2014 by Patrick Kelley

Overview

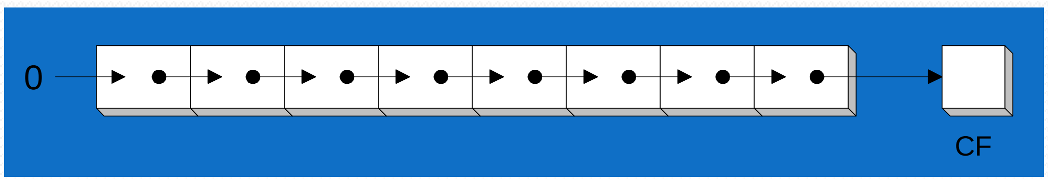
- **Shift and Rotate Instructions**
- Shift and Rotate Applications
- Multiplication and Division Instructions
- Extended Addition and Subtraction

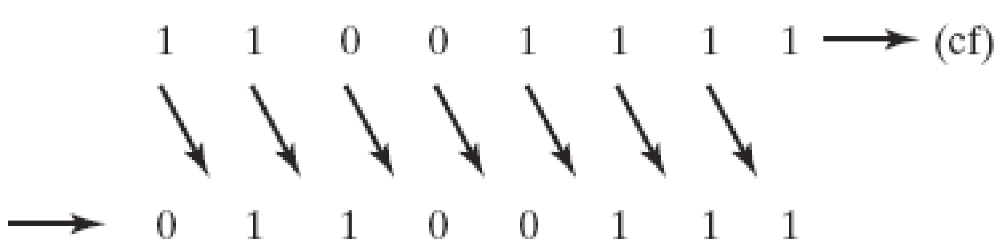
Shift and Rotate Instructions

- Logical vs Arithmetic Shifts
- sll and sllv Instruction
- srl and srlv Instruction
- sra and srav Instructions
- rol Instruction
- ror Instruction

Logical Shift

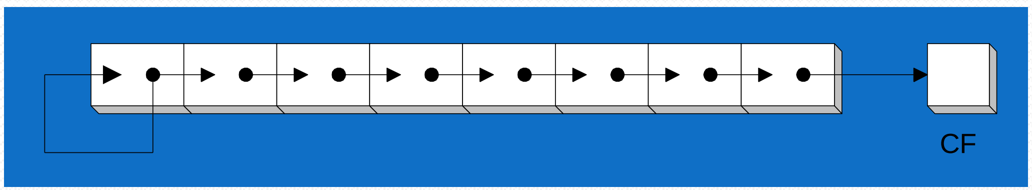
- A logical shift fills the newly created bit position with zero:

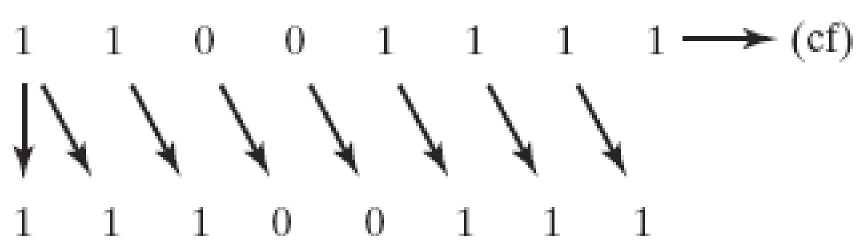




Arithmetic Shift

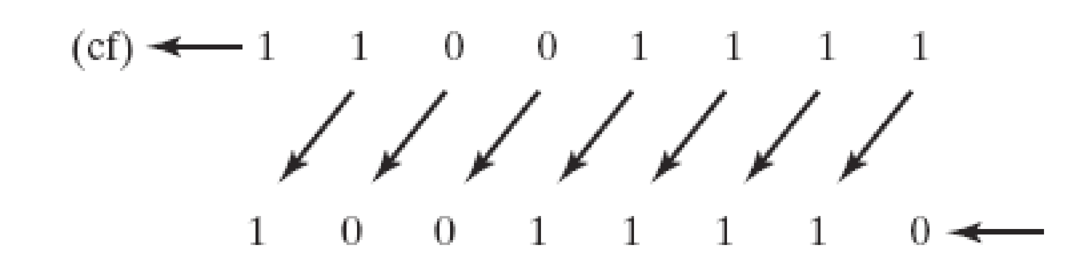
- An arithmetic shift fills the newly created bit position with a copy of the number's sign bit:





sll and sllv Instruction

- The sll (shift left) instruction performs a logical left shift on the destination operand, filling the lowest bit with 0.

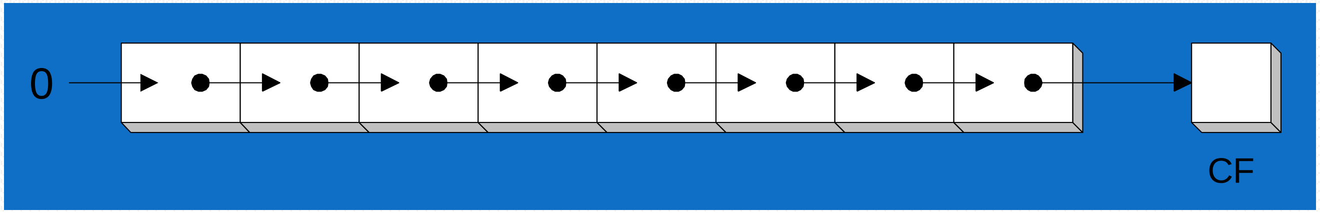



```
# imm = bit positions to shift
   sll Rd, Rt, imm
# low order 5 bits of Rs = positions to shift
   sllv Rd, Rt, Rs
```

- Arithmetic left shift is identical to Logical, so no extra instruction is needed

srl and srlv Instruction

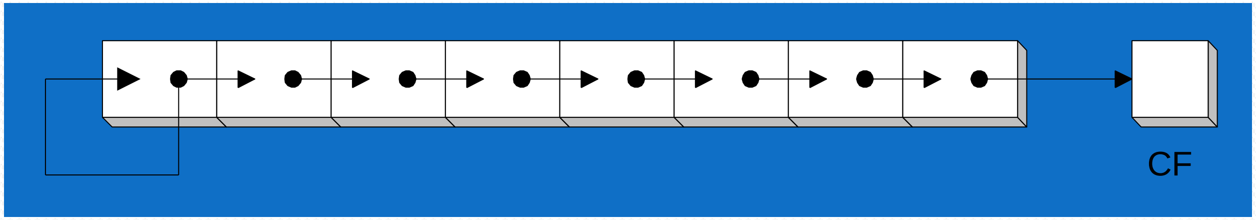
- The srl (shift right) instruction performs a logical right shift on the destination operand. The highest bit position is filled with a zero.




```
# imm = bit positions to shift
    srl Rd, Rt, imm
# low order 5 bits of Rs = positions to shift
    srlv Rd, Rt, Rs
```

sra and srav Instruction

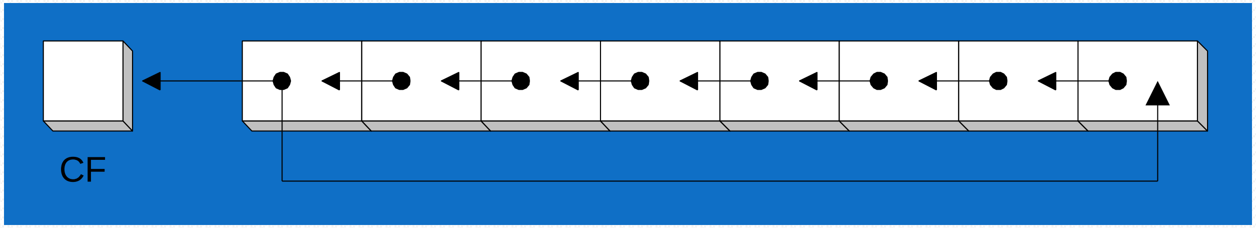
- The sra (arithmetic shift right) instruction performs an arithmetic right shift on the destination operand. The highest bit position is filled with the sign bit.




```
# imm = bit positions to shift
    sra Rd, Rt, imm
# low order 5 bits of Rs = positions to shift
    srav Rd, Rt, Rs
```

rol Instruction

- rol (rotate left) shifts each bit to the left
- The highest bit is copied into the lowest bit
- No bits are lost

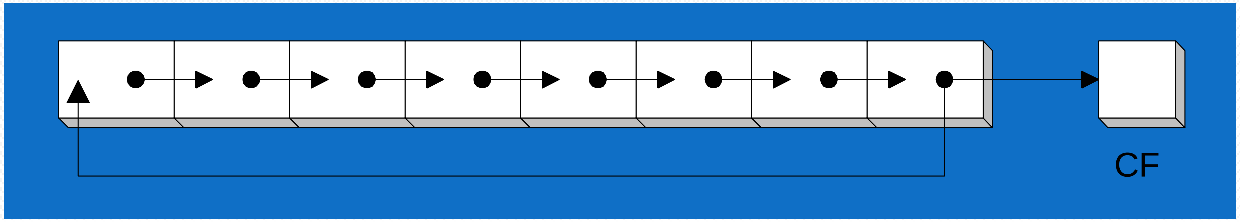


- Image shows an architecture where the highest bit is also copied into a carry flag.

```
# low order 5 bits of Rs = positions to rotate
    rol Rd, Rt, Rs
```

ror Instruction

- ror (rotate left) shifts each bit to the right
- The lowest bit is copied into the highest bit
- No bits are lost



- Image shows an architecture where the lowest bit is also copied into a carry flag.

```
# low order 5 bits of Rs = positions to rotate
    ror Rd, Rt, Rs
```

What's Next

- Shift and Rotate Instructions
- **Shift and Rotate Applications**
- Multiplication and Division Instructions
- Extended Addition and Subtraction

Getting the carry bit

- Many useful shift/rotate apps rely on the carry bit
- Since MIPS does not have status flags, we need a way
- Assume unsigned, store result separately from source
- If result < source, there was a carry.
- sltu compares result and source, setting a register appropriately.

```
# works for shifts, rotates, or addu
    addu $s4, $s1, $s2
    stlu $s3, $s4, $s2    # $s3 now holds carry
```

Shifting Multiple Doublewords

- Programs sometimes need to shift all bits within an array, as one might when moving a bitmapped graphic image from one screen location to another.
- The following shifts an array of 3 words 1 bit to the right:

```
 .data
array: .word 0x99999999h, 0x99999999h, 0x99999999h
 .test
  la   $a0, array          # load array address
  lw   $s0, ($a0)          # load high word into $s0
  srl  $s1, $s0, 1         # shift
  sltu $s2, $s1, $s0       # put carry in s2
  sw   $s1, $a0            # put shifted word back

# continued on next page
```

Shifting Multiple Doublewords

```
# continued from previous page
  addu $a0,$a0, 4             # add 4 bytes for next word
  lw   $s0, ($a0)             # load middle word into $s0
  srl  $s1, $s0, 1            # shift
  sltu $s3, $s1, $s0          # put carry in s3
  ror  $s2, $s2, $s2          # turn prev carry into mask
  addu $s1, $s1, $s2          # add carry mask to word
  sw   $s1, $a0               # put shifted word back

# do last word
  addu $a0,$a0, 4             # add 4 bytes for next word
  lw   $s0, ($a0)             # load low word into $s0
  srl  $s1, $s0, 1            # shift
  ror  $s3, $s3, $s3          # turn prev carry into mask
  addu $s1, $s1, $s3          # add carry mask to word
  sw   $s1, $a0               # put shifted word back
```

Binary Multiplication

- mutiply 123 * 36

	01111011	123
×	00100100	36
	01111011	123 SHL 2
+	01111011	123 SHL 5
	0001000101001100	4428

Binary Multiplication

- We already know that sll performs unsigned multiplication efficiently when the multiplier is a power of 2.
- You can factor any binary number into powers of 2.
  - For example, to multiply $so * 36, factor 36 into 32 + 4 and use the distributive property of multiplication to carry out the operation:

```
$s0 * 36
= $s0 * (32 + 4)
= ($s0 * 32)+(EAX * 4)
```

```
li $s0,123
mov $s0, $s1
sll $s0, $s0, 5  ; mult by 2^5
sll $s1, $s1, 2  ; mult by 2^2
addu $s0,$s0, $s1
```

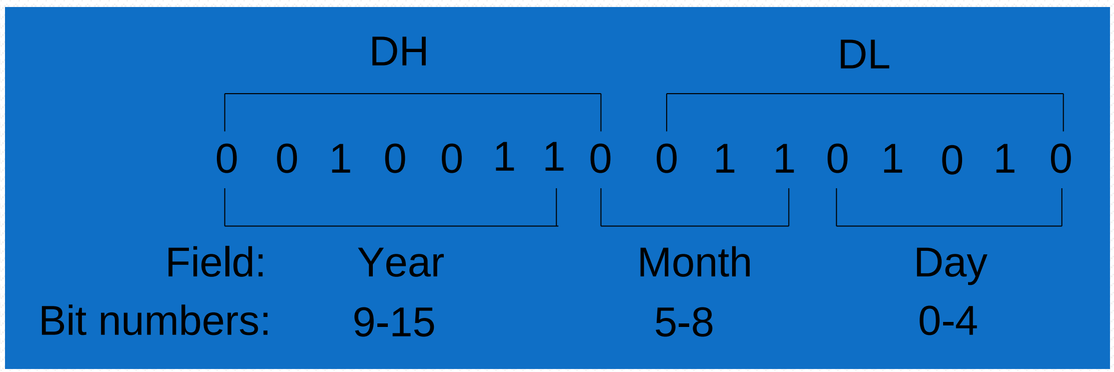
Displaying Binary Bits

Algorithm: Shift MSB into the Carry bit; If CF = 1, append a "1" character to a string; otherwise, append a "0" character. Repeat in a loop for however big your data is.

```
   .data
buffer:   .space 33     # 32 byte string
   .test
    li     $a0, 32      # doing a word
    li     $a2, '0'     # for output
    li     $a3, '1'
    la     $a1, buffer
# word was in $s0
L1: shl    $s1, $s0, 1
    sltu   $s1, $s1, $s0
    sb     $a2, ($a1)   # write '0'
    beqz   $s1, L2
    sb     $a3, ($a1)   # overwrite '1'
L2: addiu  $a1, $a1, 1  # next byte
    addi   $a0, $a0, -1 # next bit
    bgtz   $a0, L1      # loop until 0
```

Isolating a Bit String

- The MS-DOS file date field packs the year, month, and day into 16 bits:



Isolate the Month field:

```
li $a2, 0x0000000F          # mask right 4 bits
lw $a0, date                # load a date
srl $a0, $a0, 5             ; shift right 5 bits
and $a0, $a0, $a2           ; clear bits 4-31
sb $a0, month               ; save in month variable
```

What's Next

- Shift and Rotate Instructions
- Shift and Rotate Applications
- **Multiplication and Division Instructions**
- Extended Addition and Subtraction

Multiply and divide

- It should be apparent that multiplying two 32-bit numbers may produce a result larger than 32-bits
  - In fact, it is possible to get a 64-bit result
  - MIPS uses two special registers ‘high’ and ‘low’ to hold the entire result
- Similarly, an integer division may result in both a quotient and a remainder
  - Division by zero is undefined and you should check for this before you divide
  - The quotient is stored in the ‘low’ register
  - The remainder is stored in the ‘high’ register

Multiply Instructions

```
    mult  Rs, Rt              # remainder in high
    multu Rs, Rt              # quotient in low
```

- No overflow is caught
- Takes 32 cycles to execute (Booth's algorithm)
- There are macro versions with different arguments:

```
    mul   Rd, Rs, Rt          # result in high:low
    mulo  Rd, Rs, Rt          # but low moved to
    mulou Rd, Rs, Rt          # register Rd
```

- 'low' register moved to a specified register
- The latter two operations will also throw an overflow exception

Divide Instructions

```
    div    Rs, Rt                # result in high:low
    divu   Rs, Rt                # result in high:low
```

- No overflow is caught
- Takes 38 cycles to execute
- There are macro versions with different arguments:

```
    div    Rd, Rs, Rt         # result in high:low
    divu   Rd, Rs, Rt         # but low moved to Rd
```

- 'low' register moved to a specified register
- No exceptions thrown; programmer must catch exception cases

What's Next

- Shift and Rotate Instructions
- Shift and Rotate Applications
- Multiplication and Division Instructions
- **Extended Addition and Subtraction**

Extended Precision Addition

- Adding two operands that are longer than the computer's word size (32 bits).
  - Virtually no limit to the size of the operands
- The arithmetic must be performed in steps
  - The Carry value from each step is passed on to the next step.

Extended Addition Example

- Task: Add 1 to $s1:$s0
  - Starting value of $s1:$s0: 0x00000000FFFFFFFF
  - Add the lower 32 bits first, setting the Carry bit in $s2.
  - Add the upper 32 bits.

```
li $s1,0               # set upper half
li $s0,0xFFFFFFFF      # set lower half
addiu $s3, $s0, 1      # add lower half
sltu $s4, $s3, $s0     # check carry
move $s0, $s3          # put lower result in $s0
addu $s1, $s1, $s4     # add carry to upper half
# if both operands are bigger than a word, then
# you could add the upper halves and carry
# checking for a carry out each time.
```

```
$s1:$s0 = 00000001 00000000
```

Extended Subtraction Example

- Task: Subtract 1 from $s1:$s0
  - Starting value of $s1:$s0: 0x0000000100000000
  - Subtract the lower 32 bits first, setting the Carry bit.
  - Subtract the upper 32 bits.

```
li $s1, 1             # set upper half
li $s0, 0             # set lower half
Li $s5, 1             # number to subtract
subu $s3, $s0, $s5    # subtract lower half
sltu $s4, $s3, $s0    # check carry
move $s0, $s3         # put lower result in $s0
subu $s1, $s1, $s4    # subtract carry from upper half
```

```
$s1:$s0 = 00000000 FFFFFFFF
```