Logical and Shift Operations

Up Until Now

- Up until now, we've done
  - Arithmetic: `add`, `sub`, `addi`
  - Memory access: `lw` and `sw`
  - branches and jumps: j, jr, jal, beq, bne

- These instructions view contents of register as a **single quantity** (such as a signed or unsigned integer)

Bitwise Operations

- View contents of register as 32 **independent bits**
  - Since registers are composed of 32 bits, we may want to access individual bits (or groups of bits) rather than the whole.

- Two new classes of MIPS instructions for bitwise operations:
  - Logical Operators
  - Shift Operators

Bitwise Operations

- Truth Table: lists all combinations of inputs and outputs

A	B	AND	OR	NOR
0	0	0	0	1
0	1	0	1	0
1	0	0	1	0
1	1	1	1	0

AND: outputs 1 if both inputs are 1
OR : outputs 1 if at least one input is 1
NOR: outputs 1 if both inputs are 0

Bitwise Operations

- Bitwise result **applies function to each bit independently**
- The i^{th} bit of inputs produce the i^{th} bit of outputs

A	B	AND	OR	NOR
0	0	0	0	1
0	1	0	1	0
1	0	0	1	0
1	1	1	1	0

Bit-wise AND

```
AND( 01101001,
     11001100 )
   = 01001000
```

Bit-wise OR

```
OR ( 01101001,
     11001100 )
   = 11101101
```

Boolean function applied at each bit position

MIPS Logical Operators

MIPS Logical Operations

- MIPS Logical Operators are ***bitwise operations***
- Basic MIPS logical operators

Syntax

```
and TargetReg, SourceReg1, SourceReg2
or  TargetReg, SourceReg1, SourceReg2
nor TargetReg, SourceReg1, SourceReg2
```

- Like many MIPS instructions, logical operations accept exactly 2 inputs and produce 1 output

Use Logical Operator in Conditional Statement

- Conditional statements

```
// C
if ( A < 0 && B < 0 ) {
     ...
}
```

```
$t0 = 0000 0000 0000 0000 0000 0000 0000 0001
$t1 = 0000 0000 0000 0000 0000 0000 0000 0000
$t0 and $t1 = 0000 0000 0000 0000 0000 0000 0000 0000
```

```
                # MIPS code
slt $t0, $s0, $zero   # $t0 = $s0 < 0?
slt $t1, $s1, $zero   # $t1 = $s1 < 0?
and $t2, $t0, $t1     # $t2 = ( $s0 < 0 && $s1 < 0)?
```

Use Logical Operator in Conditional Statement

- Conditional statements

```
// C
if ( A < 0 || B < 0 ) {
     ...
}
```

```
          $t0 = 0000 0000 0000 0000 0000 0000 0000 0001
          $t1 = 0000 0000 0000 0000 0000 0000 0000 0000
$t0 or $t1 = 0000 0000 0000 0000 0000 0000 0000 0001
```

```
                # MIPS code
slt $t0, $s0, $zero   # $t0 = $s0 < 0?
slt $t1, $s1, $zero   # $t1 = $s1 < 0?
or  $t2, $t0, $t1     # $t2 = ( $s0 < 0 || $s1 < 0)?
```

Logical Operators with Immediate

- Similar to and, or, but the third argument is an immediate

```
                    Syntax
andi  TargetReg, SourceReg, Immediate
ori   TargetReg, SourceReg, Immediate
```

NOR for NOT

- Boolean expressions are made with AND and OR and NOT
- Why is NOT not a MIPS instruction?
  - NOT takes one operand and produces one result, which is not in keeping with the three-operand format of other instructions
  - How do we do NOT with NOR?

```
nor $t1, $t0, $zero
```

$zero	Input	NOR
0	0	1
0	1	0

Mask

An Application of Logical Operators

No, not this one

What is a Mask?

- A **mask help us:**
  - Query a bit to find out if it is a 1 or a 0
  - Change the value of a bit to 1 or a 0

- Based on observation of **AND** & **OR** logical operation properties.

Use AND for Mask

- Any bit `and` 0 produces an output 0

0	Input	AND
0	0	0
0	1	0

This can be used to create a *mask*.

- Any bit `and` 1 produces the original bit

1	Input	AND
1	0	0
1	1	1

Use AND for Mask

Example:

```
      A = 1011 0110 1010 0100 0011 1101 1001 1010
      B = 0000 0000 0000 0000 0000 0000 1111 1111
A and B= 0000 0000 0000 0000 0000 0000 1001 1010
```

- In this example, **B** is called a **mask**.
- **B** is used to isolate the rightmost 8 bits of **A** by *masking* out the rest of the string (e.g., setting it to all 0s)
- Thus, the `and` operator can be used to set certain portions of a bitstring to 0s, while leaving the rest alone.

Use AND for Mask

Example: If A = 0xB6A43D9A is saved in $t0, then what is $t1 and $t2 after the following instructions?

```
andi $t1, $t0, 0xFF
```

```
  $t0 = 1011 0110 1010 0100 0011 1101 1001 1010
0xFF = 0000 0000 0000 0000 0000 0000 1111 1111
  $t1 = 0000 0000 0000 0000 0000 0000 1001 1010   = 0x9A
```

```
andi $t2, $t0, 0x000000FF
```

```
  $t0 = 1011 0110 1010 0100 0011 1101 1001 1010
0x000000FF = 0000 0000 0000 0000 0000 0000 1111 1111
  $t2 = 0000 0000 0000 0000 0000 0000 1001 1010   = 0x9A
```

Use AND for Mask

Example: If A = 0xB6A43D9A is saved in $t0, then what is $t1 and $t2 after the following instructions?

```
andi $t1, $t0, 0xFF
```

```
  $t0 = 1011 0110 1010 0100 0011 1101 1001 1010
0xFF = 0000 0000 0000 0000 0000 0000 1111 1111
  $t1 = 0000 0000 0000 0000 0000 0000 1001 1010   = 0x9A
```

```
andi $t2, $t0, 0x000000FF
```

```
  $t0 = 1011 0110 1010 0100 0011 1101 1001 1010
0x000000FF = 0000 0000 0000 0000 0000 0000 1111 1111
  $t2 = 0000 0000 0000 0000 0000 0000 1001 1010   = 0x9A
```

Uses OR for Mask

- Any bit `or` 0 produces the original bit

0	Input	OR
0	0	0
0	1	1

- Any bit `or` 1 produces 1

0	Input	OR
1	0	1
1	1	1

This can also be used to create a *mask*.

Uses OR for Mask

- Can be used to force certain bits of a string to 1s.

  Example: if $t0 contains 0x12345678, then after

```
ori $t1, $t0, 0xFFFF
```

  $t1 contains 0x1234FFFF (high-order 16 bits are untouched, low-order 16 bits are forced to 1s).

```
   $t0 = 0001 0010 0011 0100 0101 0110 0111 1000
0xFFFF = 0000 0000 0000 0000 1111 1111 1111 1111
   $t1 = 0001 0010 0011 0100 1111 1111 1111 1111
```

Why it's important

Controller:

- A card that interfaces with a peripheral
- It uses three special purpose registers
  - Status – bit flags, 1 – true, 0 – false
  - Command – bit flags or integers to send commands

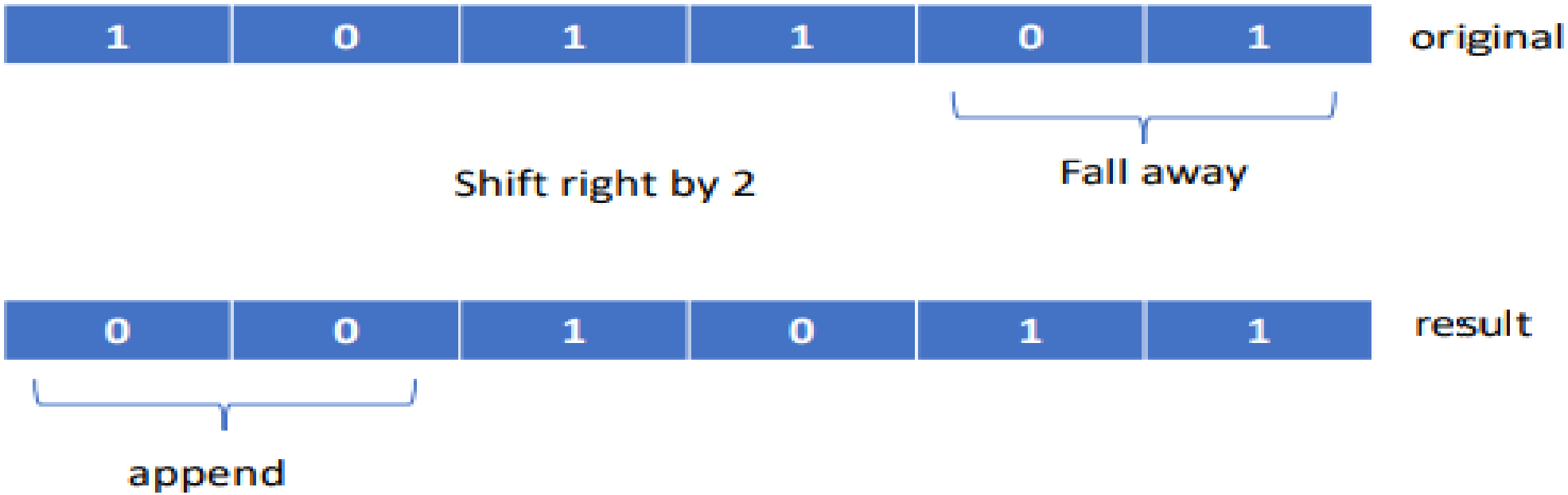
- Check the individual bits in ALU status register

Shift Operators

<<<<<<<<<<

What is shifting?

- Moving the bits in a register a certain number of positions to the left or right.



Shift Logical Instructions

- Shift Instruction Syntax:

```
Operation TargetReg, SourceReg, ShiftAmount
```

1) **`Operation:`** operation name
2) **`TargetReg:`** register that will receive value
3) **`SourceReg:`** register that contains the original value
4) **`ShiftAmount:`** shift amount (non-negative constant < 32)

Shift Logical Instructions

- Shift left logical **`sll`**
  - shifts left and fills emptied bits with 0s

```
sll TargetReg, SourceReg, ShiftAmount
```

- Shift right logical **`srl`**
  - shifts right and fills emptied bits with 0s

```
srl TargetReg, SourceReg, ShiftAmount
```

Shift Logical Instructions

Example:

Assume $t0 contains 0001 0010 0011 0100 0101 0110 0111 1000

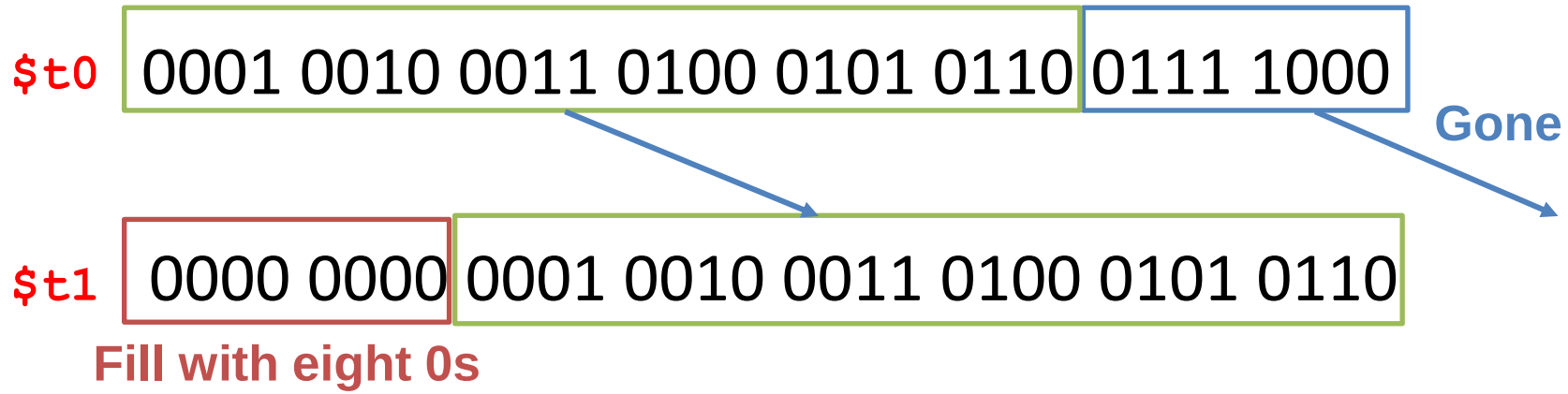
What are $t1 and $t2

```
# shift right
srl $t1, $t0, 8
```

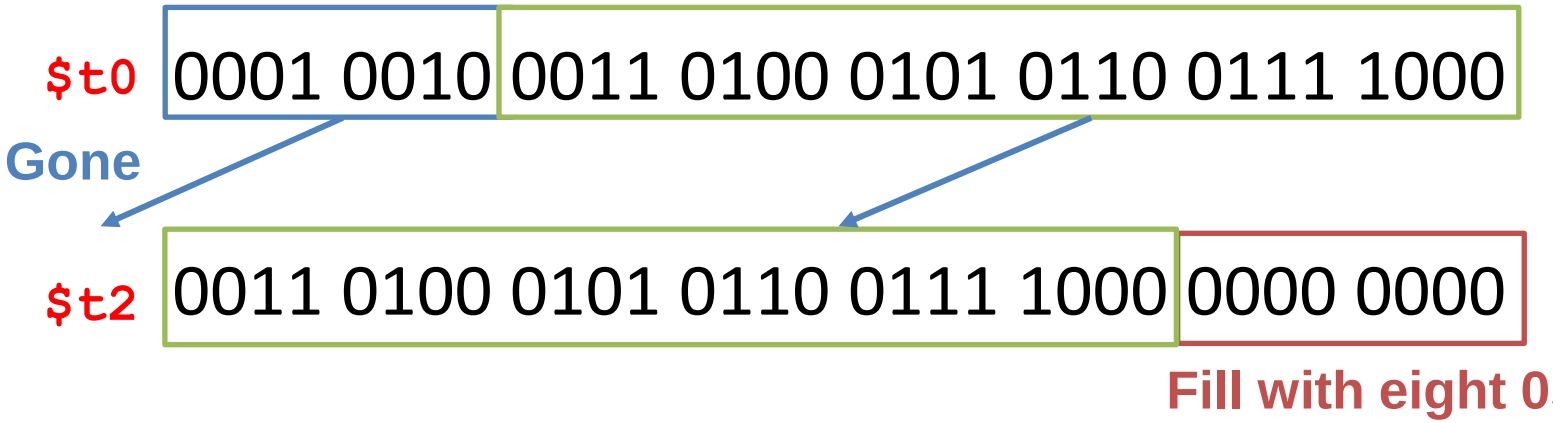
```
# shift left
sll $t2, $t0, 8
```

Shift Logical Instructions

```
# shift right
srl $t1, $t0, 8
```




```
# shift left
sll $t2, $t0, 8
```



Shift Arithmetic Instructions

- Shift right arithmetic **`sra`**
  - Shifts right and fills emptied bits by sign extending (sensitive to sign bit)

```
sra TargetReg, SourceReg, ShiftAmount
```

- Why? A negative number should stay negative after shifting
  - If MSB = 0, shift and fill the new bits with 0s
  - If MSB = 1, shift and fill the new bits with 1s

Shift Arithmetic Instructions

Example: SRA (shift right arithmetic) by 8 bits

$t3 = 0001 0010 0011 0100 0101 0110 0111 1000

$t4 = 1001 0010 0011 0100 0101 0110 0111 1000

What happen after shift right arithmetic by 8 bits?

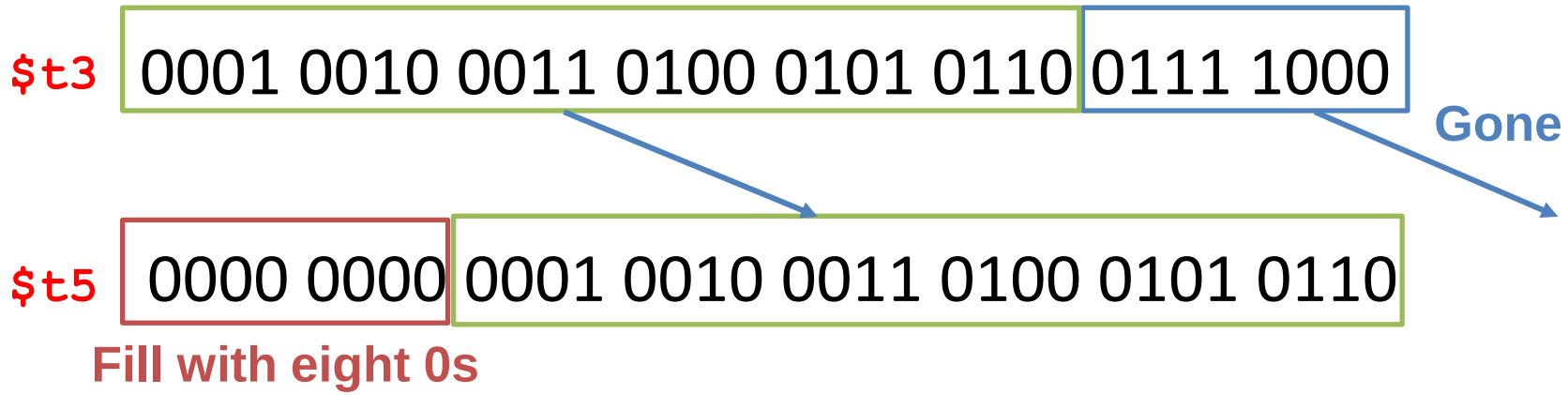
```
sra $t5, $t3, 8
```

```
sra $t5, $t4, 8
```

Shift Arithmetic Instructions

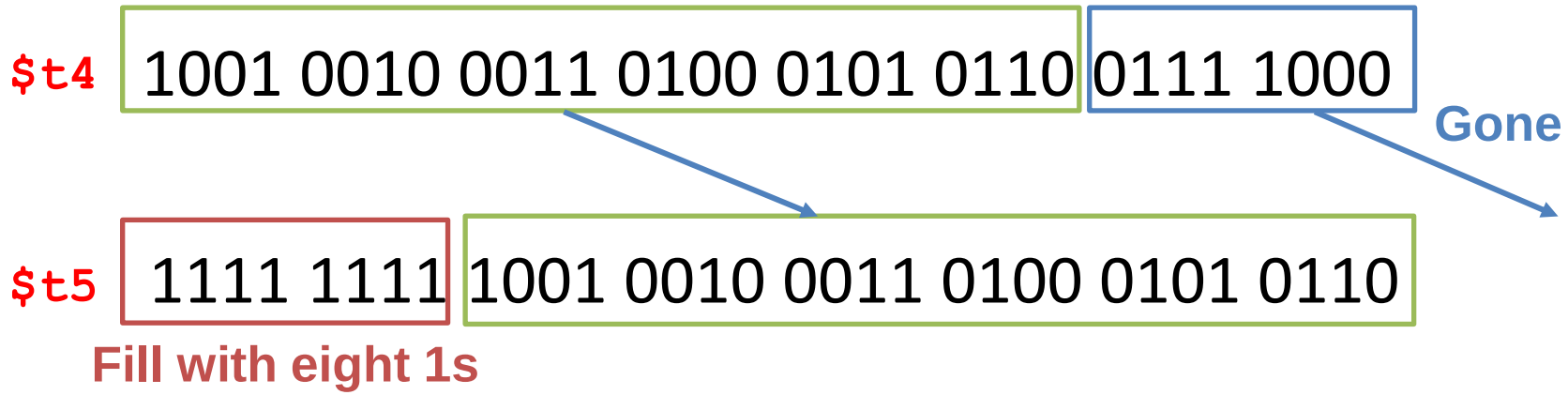
```
# shift right
sra $t5, $t3, 8
```

If MSB = 0, the new bit after shifting = 0




```
# shift right
sra $t5, $t4, 8
```

If MSB = 1, the new bit after shifting = 1



Use Shift Instructions in Multiplication

- In decimal:
  - Multiplying by 10 = shifting left by 1: $714_{10} \times 10_{10} = 7140_{10}$
  - Multiplying by 100 =shifting left by 2: $714_{10} \times 100_{10} = 71400_{10}$
  - Multiplying by 10^n = shifting left by n
- In binary:
  - Multiplying by 2 = shifting left by 1: $11_2 \times 10_2 = 110_2$
  - Multiplying by 4 = shifting left by 2: $11_2 \times 100_2 = 1100_2$
  - Multiplying by 2^n = shifting left by n

Use Shift Instructions in Multiplication

- Shifting maybe faster than multiplication!
  - a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:

```
a = a * 8 ; // C
```

→

```
sll $s0, $s0, 3 # MIPS
```

- Likewise, shift right to divide by powers of 2
  - Use `sra` but watch out for negative numbers as the result is rounded down

```
b = b / 2 ; // C
```

→

```
sra $s1, $s1, 1 # MIPS
```

with $s1 = -3 answer is -2

Why it's important

Fast multiplication by 2

Fast division by 2

Regular multiplication and division require n clock cycles, where n is the number of steps.

Shifting runs in a single clock cycle.

Shift Arithmetic Instructions

Example: `$t0 = 1001_2` What happen after the operations?

```
sra $t1, $t0, 1
srl $t2, $t0, 1
sll $t3, $t0, 1
```

Shift Arithmetic Instructions

Example: $t0 = 1001_2 What happen after the operations?

```
sra $t1, $t0, 1
srl $t2, $t0, 1
sll $t3, $t0, 1
```

$t0 = 1001_2 = -7_10

$t1 = 1100_2 = -4_10

$t2 = 0100_2 = 4_10

$t3 = 0010_2 = 2_10

Use Shift to Extract Information

- Suppose we want to isolate byte 0 (rightmost 8 bits) of a word stored in $t0

```
andi $t0, $t0, 0xFF
```

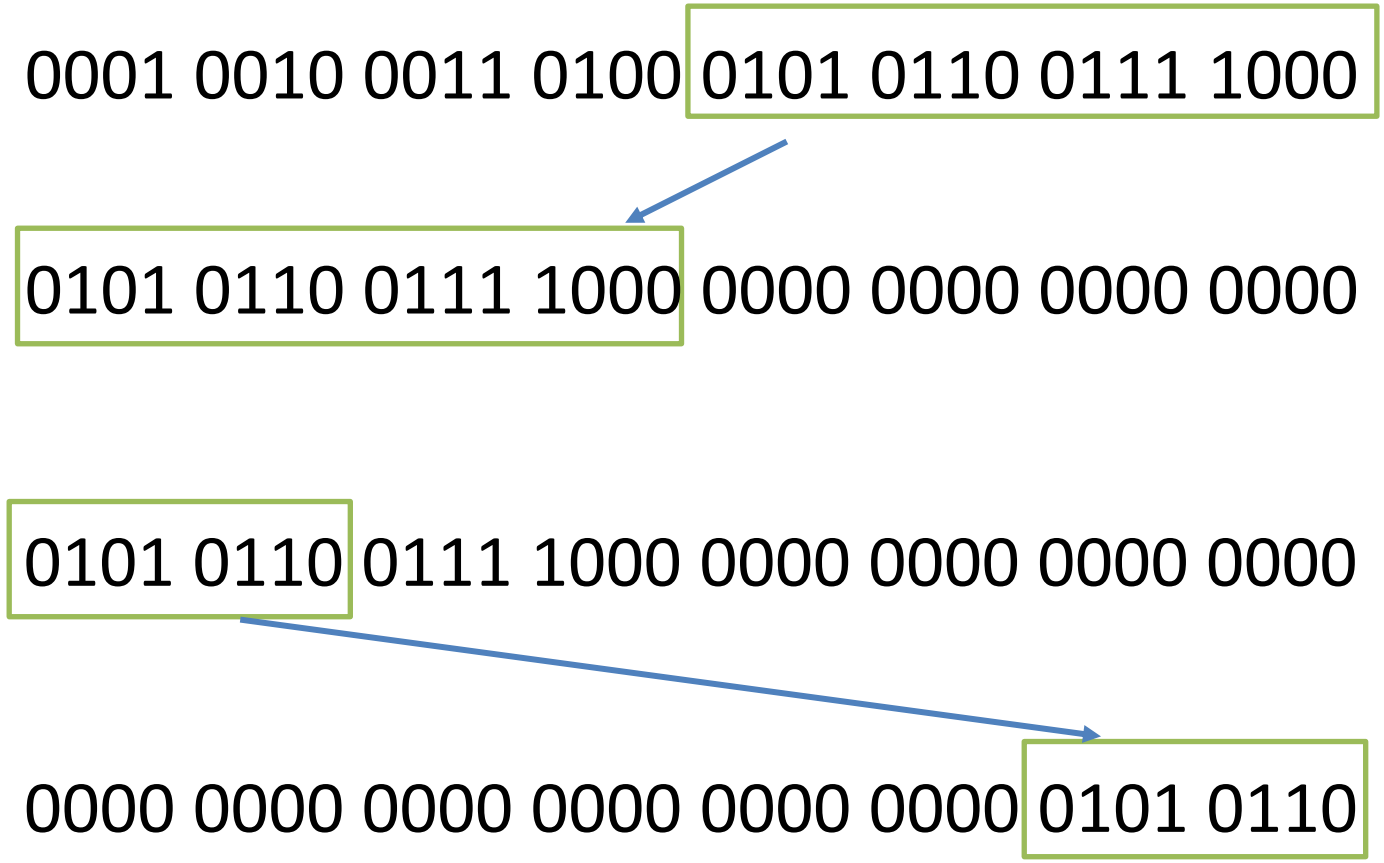
- Suppose we want to isolate byte 1 (bit 15 to bit 8) of a word stored in $t0.

```
andi $t0, $t0, 0xFF00
```

- How do we “extract” the information?

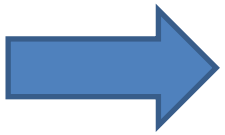
Use Shift to Extract Information

```
sll $t0, $t0, 16
srl $t0, $t0, 24
```



Application: Pixel Data for Images

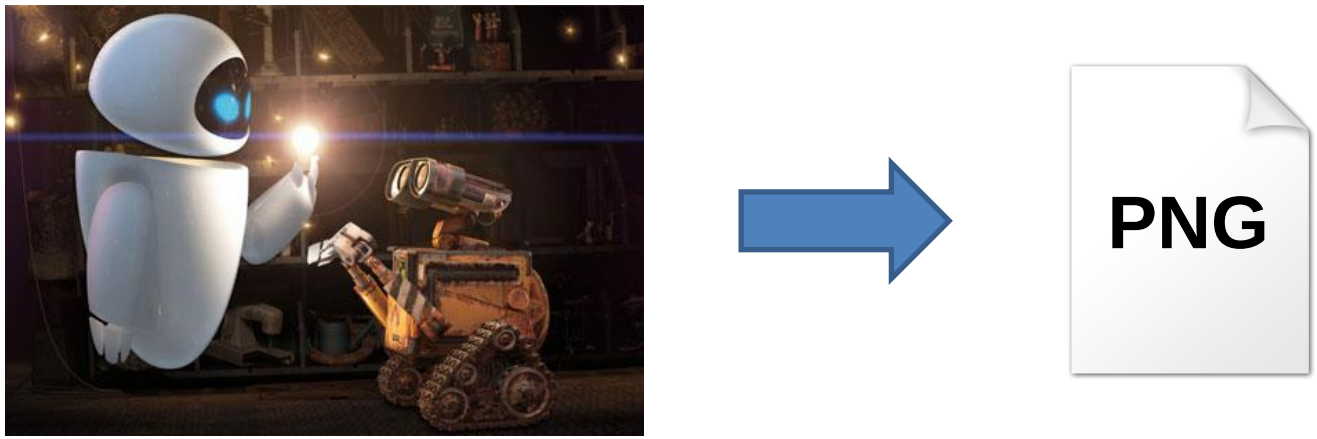
Example: Packing pixel data for images



- Suppose each pixel of an image has
  - *Red r* [0 ~ 255]
  - *Green g* [0 ~ 255]
  - *Blue b* [0 ~ 255]
  - *Alpha a*, transparency value [0 ~ 255]

Each of them can be represented by a byte (8-bit)

Example: Packing pixel data for images



- Instead of using 4 registers for r, g, b, and a, we **pack** each of 8 bits into a 32-bit integer

```
// C code
int packARGB ( int r, int g, int b, int a ) {
    return a << 24 | r << 16 | g << 8 | b ;
}
```

Example: Packing pixel data for images

```
// C code
int packARGB ( int r, int g, int b, int a ) {
    return a << 24 | r << 16 | g << 8 | b ;
}
```

```
# MIPS
sll   $t0, $a3, 24      # shift a to the left
or    $v0, $t0, $zero   # combine a with $v0
sll   $t0, $a0, 16      # shift r to the left
or    $v0, $t0, $v0     # combine r with $v0
sll   $t0, $a1, 8       # shift g to the left
or    $v0, $t0, $v0     # combine g with $v0
or    $v0, $a2, $v0     # combine b with $v0
jr    $ra               # return to $ra
```

Review

- Logical and Shift Instructions operate on bits individually, unlike arithmetic, which operate on entire word
- Use Logical and Shift Instructions to isolate fields, either by masking or by shifting back and forth
- New Instructions:
  - `Logical: and, andi, or, ori, nor`
  - `Shift:  sll, srl, sra`
- Practice: try writing MIPS functions
- Textbook 2.6