Logical and Shift Operations

Overview

- Logical Instructions
- Shifts

Bitwise Operations (1/2)

- Up until now, we've done arithmetic (`add`, `sub`,`addi`), memory access (`lw` and `sw`), and branches and jumps.
- All of these instructions view contents of register as a single quantity (such as a signed or unsigned integer)
- **New Perspective**: View contents of register as 32 bits rather than as a single 32-bit number

Bitwise Operations (2/2)

- Since registers are composed of 32 bits, we may want to access individual bits (or groups of bits) rather than the whole.
- Introduce two new classes of instructions:
  - Logical Operators
  - Shift Instructions (we've seen `sll` already)

Logical Operators (1/4)

- Two basic logical operators:
  - AND: outputs 1 only if both inputs are 1
  - OR: outputs 1 if at least one input is 1
- In general, can define them to accept >2 inputs, but in the case of MIPS assembly, both of these accept exactly 2 inputs and produce 1 output
  - Again, rigid syntax, simpler hardware

Logical Operators (2/4)

- Truth Table: standard table listing all possible combinations of inputs and resultant output for each
- Truth Table for AND and OR

A	B	AND	OR
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Logical Operators (3/4)

- **Logical Instruction Syntax:**

  1 2,3,4

  - **where**

    1) operation name

    2) register that will receive value

    3) first operand (register)

    4) second operand (register) or immediate (numerical constant)

Logical Operators (4/4)

- **Instruction Names:**
  - `and`, `or`: Both of these expect the third argument to be a register
  - `andi`, `ori`: Both of these expect the third argument to be an immediate
- **MIPS Logical Operators are all bitwise, meaning that bit 0 of the output is produced by the respective bit 0's of the inputs, bit 1 by the bit 1's, etc.**

Uses for Logical Operators (1/3)

- Note that `anding` a bit with 0 produces a 0 at the output while `anding` a bit with 1 produces the original bit.
- This can be used to create a **mask**.
  - Example:

    1011 0110 1010 0100 0011 1101 1001 1010

    0000 0000 0000 0000 0000 1111 1111 1111
  - The result of `anding` these two is:

    0000 0000 0000 0000 0000 1101 1001 1010

Uses for Logical Operators (2/3)

- The second bitstring in the example is called a **mask**. It is used to isolate the rightmost 12 bits of the first bitstring 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.
  - In particular, if the first bitstring in the above example were in $t0, then the following instruction would mask it:

```
andi   $t0,$t0,0xFFF
```

Uses for Logical Operators (3/3)

- Similarly, note that `oring` a bit with 1 produces a 1 at the output while `oring` a bit with 0 produces the original bit.
- This can be used to force certain bits of a string to 1s.
  - For example, if $t0 contains 0x12345678, then after this instruction:

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

  - … $t0 contains 0x1234FFFF (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are forced to 1s).

Shift Instructions (1/4)

- Move (shift) all the bits in a word to the left or right by a number of bits.
  - Example: shift right by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

  - Example: shift left by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0011 0100 0101 0110 0111 1000 0000 0000

Shift Instructions (2/4)

- **Shift Instruction Syntax:**

  1 2,3,4

  - **where**

    1) operation name

    2) register that will receive value

    3) first operand (register)

    4) shift amount (constant <= 32)

Shift Instructions (3/4)

- MIPS shift instructions:
  1. `sll` (shift left logical): shifts left and fills emptied bits with 0s
  2. `srl` (shift right logical): shifts right and fills emptied bits with 0s
  3. `sra` (shift right arithmetic): shifts right and fills emptied bits by sign extending

Shift Instructions (4/4)

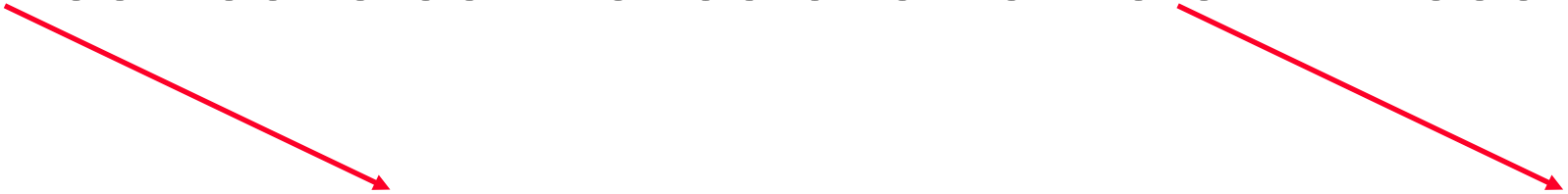
- Example: shift right arith by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

- Example: shift right arith by 8 bits

1001 0010 0011 0100 0101 0110 0111 1000



1111 1111 1001 0010 0011 0100 0101 0110

Uses for Shift Instructions (1/5)

- Suppose we want to isolate byte 0 (rightmost 8 bits) of a word in $t0. Simply use:

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

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

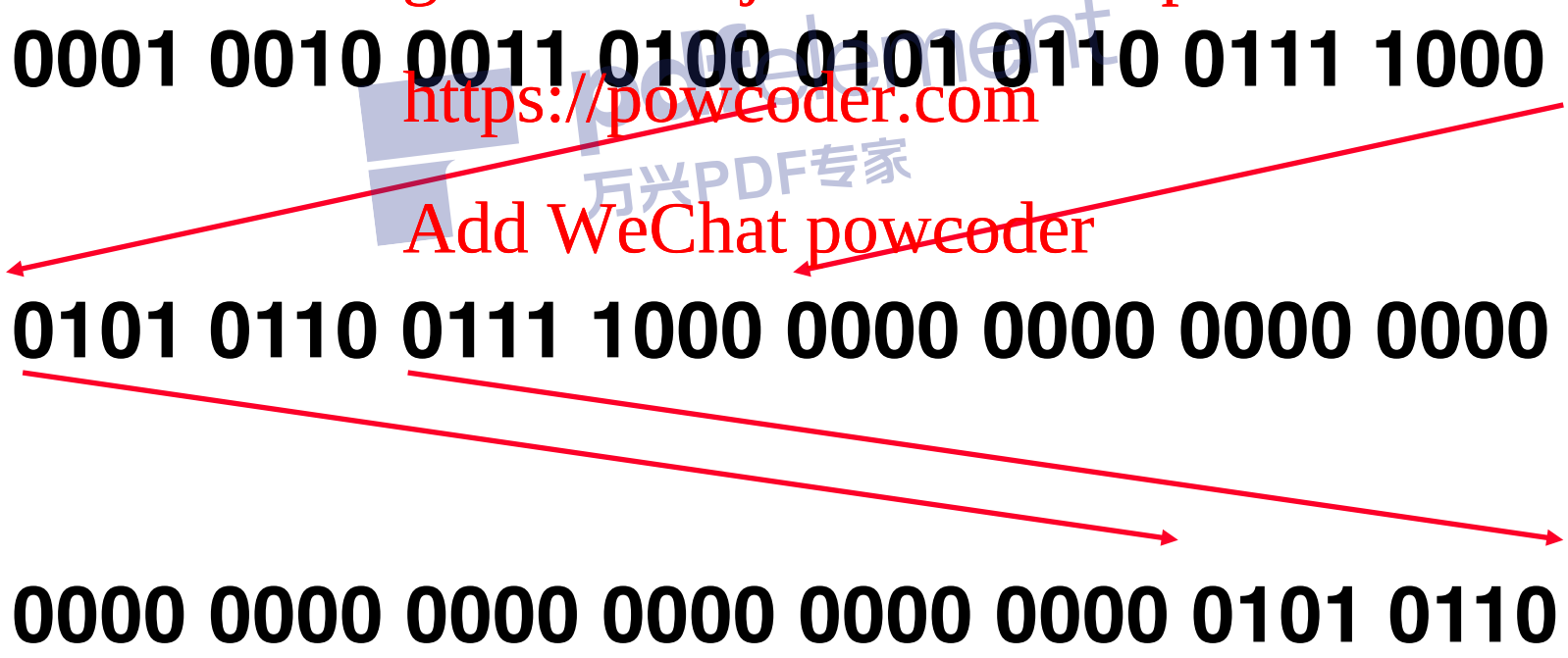
```
andi    $t0,$t0,0xFF00
```

but then we still need to shift to the right by 8 bits...

Uses for Shift Instructions (2/5)

- Could use instead:

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

Uses for Shift Instructions (3/5)

- In decimal:
  - Multiplying by 10 is same as shifting left by 1:
    - $714_{10} \times 10_{10} = 7140_{10}$
    - $56_{10} \times 10_{10} = 560_{10}$
  - Multiplying by 100 is same as shifting left by 2:
    - $714_{10} \times 100_{10} = 71400_{10}$
    - $56_{10} \times 100_{10} = 5600_{10}$
  - Multiplying by 10^n is same as shifting left by n

Uses for Shift Instructions (4/5)

- In binary:
  - Multiplying by 2 is same as shifting left by 1:
    - $11_2 \times 10_2 = 110_2$
    - $1010_2 \times 10_2 = 10100_2$
  - Multiplying by 4 is same as shifting left by 2:
    - $11_2 \times 100_2 = 1100_2$
    - $1010_2 \times 100_2 = 101000_2$
  - Multiplying by 2^n is same as shifting left by n

Uses for Shift Instructions (5/5)

- Since 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 *= 8;` (in C)

  would compile to:

  `sll $s0,$s0,3` (in MIPS)

- Likewise, shift right to divide by powers of 2
  - remember to use `sra`

Things to Remember (1/2)

- 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.

Things to Remember (2/2)

- **New Instructions:**

```
and, andi, or, ori
sll, srl, sra
```