Lecture 8

On logical operations -- bitwise shifts, bitwise AND/OR/XOR, and how to execute them in MIPS and C,C++

Bitwise Shifts

- Logical left shift

  `0000 1010` --> *shift left by 1* --> `0001 0100`

  - Leftmost bit is lost, rightmost bit filled with 0
  - It turns out, shifting left by 1 bit is equivalent to multiplying by 2

    To make sense of this, think about it in the decimal number system. Say we have 0123, and we shift it left by 1 bit -- now we have 1230, which is the original number multiplied by 10 (the base of the system)
  - The pattern continues, if we shift left by 2 bits, we are effectively multiplying by 4 (2^2)

- Logical right shift

  `0000 1010` --> *shift right by 1* --> `0000 0101`

  - Rightmost bit is lost, leftmost bit filled with 0
  - It turns out, shifting right by 1 bit is equivalent to dividing by 2
  - Similarly, if we shift right by 2 bits, we'd be dividing by 4, and so on
  - Note that the remainder is discarded when dividing this way

- Shifting can easily overflow, if a number already barely fits in the register, shifting will push significant digits out of the register.

Bitwise Shift Instructions in MIPS

- The bitwise shifts are R-type instructions

6	5	5	5	5	6
opcode	rs	rt	rd	shamt	function

Finally, we'll use the shamt!

`SLL rd, rt, shamt`

- *SLL $t2, $s0, 4*
- Shift $s0 left 4 bits, store in $t2

`SRL rd, rt, shamt`

Bitwise Shift Instructions in C

```
int a = 1;

a = a << 1; //left shift by 1, multiply by 2

a = a >> 2; //right shift by 2, divide by 4
```

Bitwise AND/OR/XOR

Le truth table

x	y	AND	OR	XOR
0	**0**	**0**	**0**	**0**
0	**1**	**0**	**1**	**1**
1	**0**	**0**	**1**	**1**
1	**1**	**1**	**1**	**0**

- As assembly instructions, each of these perform their bitwise operation on every corresponding pair of bits of the two binary bit strings being operated on
- The bitwise AND / OR / XOR instructions are R-type instructions
- *AND $t0, $s0, $s1 (t0 = s0 & t0)*

  OR $t0, $s0, $s1

  XOR $t0, $s0, $s1

Masking

- A common use of the *AND* instruction is for **masking**
- Say we wanted to extract the leftmost 6 bits of an instruction, how would we do that?

Let's do a small experiment first

1010 AND 1111 -- what happens when you AND a bit with a 1?

Result: 1010

Conclusion: ANDing a bit with a 1 does not change its state; since both bits have to be 1 in order for the operation to evaulate to true, ANDing by 1 will keep 1s as 1s and 0s as 0s (think about it)

1010 AND 0000 -- what happens when you AND a bit with a 0?

Result: 0000

Conclusion: ANDing a bit with a 0 'clears' it (sets it to 0). Anything AND 0 has to be 0, cause both operands must be 1 for it to evaluate to true.

```
unsigned int instruction; //what's in the first 6 bits? (opcode)
```

- The way to do it is to make a "mask"
  - Using this discovered property of the bitwise AND, we can create a binary string formatted in such a way that ANDing it with the instruction in question will yield only the bits we want
  - We call this binary string a mask
  - The trick is to make the bits corresponding to positions of bits we don't care about be 0s, and for the ones we do care about be 1s. This way the bits we care about will remain the same while the rest of the binary string will be cleared.

So, to get the first 6 bits, we simply create a mask with 1s in the first 6 bits and 0s everywhere else, and then use that mask

1111 11 *00 0000 0000 0000 0000 0000 0000 (remember that instructions are 32 bits long)*

There's a problem with this: typing out 32 digits is very labor intensive and prone to error. Instead of binary, let's use hexadecimal to make our mask. Simply convert each group of 4 bits to a hexadecimal value.

0x F C 0 0 0 0 0 0

Our example problem is posed in C, not MIPS, so let's continue in C.

unsigned int mask = 0xFC000000;

if (instruction & mask == 0) cout << "OPCODE is 0";

'&' is the bitwise AND symbol, if the entire 32-bit string has the value of 0 after being ANDed with the mask, it means the leftmost 6 bits were all 0, so the opcode is 0

What about checking if the OPCODE is 9? That would mean the leftmost 6 bits have a value of 9 collectively, so like this: 0010 01...

ANDing our mask with the instruction, leaving only the leftmost 6 bits uncleared, should then produce exactly ***0010 01*** *00 0000 0000 0000 0000 0000 0000 (that is, if the OPCODE indeed is 9). We simply need to check if the operation indeed produces this value.*

Problem: we can't write out the entire binary string in the if condition: if(instruction & mask == 00100100000000000000000000000000) -- it's too long and also would be interpreted as an octal number cause it starts with a 0. The solution is to encode your desired result as a hexadecimal number, for brevity and so on."

9 == 0010 0100 0000 0000 0000 0000 0000 0000 == 0x 2 4 0 0 0 0 0 0

if (instruction & mask == 0x24000000) cout << "OPCODE is 9";

That ends this small case study, hopefully that was helpful.

- We know what happens when we AND a binary string with all 1s and 0s -- what happpens when we OR it or XOR it in a similar manner?

1010 OR 1111 == 1111

1010 OR 0000 == 1010

Conclusion: ORing with 1s 'sets' a bit (to 1), while ORing with a 0 preserves a bit's state (doesn't affect it)

1010 XOR 1111 == 0101

1010 XOR 0000 == 1010

Conclusion: XORing with 1s inverts a bit, while XORing with a 0 preserves its value

- We can use these masking tricks to manipulate certain portions of bits (either by inverting with XOR, setting with OR, or clearing with AND)

Motivating Problem (from Leetcode :O)

```
vector<int> nums; // {1, 2, 3, 4, 1, 2, 4}
```

// stores pairs (1 and 1, 2 and 2, etc), and one unique number

// find this unique number

// solutions: brute force (loop through, O(n^2)), fancy unordered map O(n) and linear space, elegant solution XOR O(n) and constant space

```
int result = 0;

for(int num: nums)

    results = result ^ num; //xor

return result;
```

This works because a number XORed with itself results in 0, and 0 XORed with a number results in that number. All of the duplicates cancel each other out, while the unique number ends up getting XORed with 0 to get the final result. Also important is that XOR is commutative, so even though the duplicate pairs are dispersed throughout the array, they'll still cancel each other out, cause order doesn't matter.