COMP 122/L Lecture 2

Kyle Dewey

Outline

- Operations on binary values
  - AND, OR, XOR, NOT
  - Bit shifting (left, two forms of right)
  - Addition
  - Subtraction
- Twos complement

Bitwise Operations

Bitwise AND

- Similar to logical AND (`&&`), except it works on a bit-by-bit manner
- Denoted by a single ampersand: `&`

```
(1001 &
 0101)=
 0001
```

Bitwise OR

- Similar to logical OR (`||`), except it works on a bit-by-bit manner
- Denoted by a single pipe character: `|`

```
(1001 |
 0101)=
 1101
```

Bitwise XOR

- Exclusive OR, denoted by a carat: `^`
- Similar to bitwise OR, except that if both inputs are `1` then the result is `0`

```
(1001 ^
 0101)=
 1100
```

Bitwise NOT

- Similar to logical NOT (`!`), except it works on a bit-by-bit manner
- Denoted by a tilde character: ~

```
~1001 =
 0110
```

Shift Left

- Move all the bits `N` positions to the left, subbing in `N` `0`s on the right

Shift Left

- Move all the bits `N` positions to the left, subbing in `N` `0`s on the right

```
1001
```

Shift Left

- Move all the bits `N` positions to the left, subbing in `N` `0`s on the right

```
1001 << 2 =
100100
```

Shift Left

- Useful as a restricted form of multiplication
- Question: how?

```
1001 << 2 =
100100
```

Shift Left as Multiplication

- Equivalent decimal operation:

234

Shift Left as Multiplication

- Equivalent decimal operation:

```
234 << 1 =
2340
```

Shift Left as Multiplication

- Equivalent decimal operation:

```
234 << 1 =
2340

234 << 2 =
23400
```

Multiplication

- Shifting left `N` positions multiplies by `(base)`^{N}
- Multiplying by 2 or 4 is often necessary (shift left 1 or 2 positions, respectively)
- Often a whooole lot faster than telling the processor to multiply
- Compilers try hard to do this

```
234 << 2 =
23400
```

Shift Right

- Move all the bits `N` positions to the right, subbing in **either** `N` `0`s or `N` `1`s on the left
  - Two different forms

Shift Right

- Move all the bits `N` positions to the right, subbing in **either** `N` `0`s or `N` (whatever the leftmost bit is)s on the left
  - Two different forms

```
1001 >> 2 =
either 0010 or 1110
```

Shift Right Trick

- Question: If shifting left multiplies, what does shift right do?

Shift Right Trick

- Question: If shifting left multiplies, what does shift right do?
    - Answer: divides in a similar way, but truncates result

Shift Right Trick

- Question: If shifting left multiplies, what does shift right do?
  - Answer: divides in a similar way, but truncates result

234

Shift Right Trick

- Question: If shifting left multiplies, what does shift right do?
  - Answer: divides in a similar way, but truncates result

```
234 >> 1 =
23
```

Two Forms of Shift Right

- Subbing in `0`s makes sense
- What about subbing in the leftmost bit?
  - And why is this called "arithmetic" shift right?

```
1100 (arithmetic)>> 1 =
1110
```

Answer...Sort of

- Arithmetic form is intended for numbers in *twos complement*, whereas the non-arithmetic form is intended for *unsigned* numbers

Twos Complement

Problem

- Binary representation so far makes it easy to represent positive numbers and zero
- Question: What about representing negative numbers?

Twos Complement

- Way to represent positive integers, negative integers, and zero
- If `1` is in the *most significant bit* (generally leftmost bit in this class), then it is negative

Decimal to Twos Complement

- Example: -5 decimal to binary (twos complement)

Decimal to Twos Complement

- Example: -5 decimal to binary (twos complement)
- First, convert the magnitude to an unsigned representation

Decimal to Twos Complement

- Example: -5 decimal to binary (twos complement)
- First, convert the magnitude to an unsigned representation

`5` (decimal) = `0101` (binary)

Decimal to Twos Complement

- Then, take the bits, and negate them

Decimal to Twos Complement

- Then, take the bits, and negate them

```
0101
```

Decimal to Twos Complement

- Then, take the bits, and negate them

```
~0101 =
 1010
```

Decimal to Twos Complement

- Finally, add one:

Decimal to Twos Complement

- Finally, add one:

```
1010
```

Decimal to Twos Complement

- Finally, add one:

```
1010 + 1 =
1011
```

Twos Complement to Decimal

- Same operation: negate the bits, and add one

Twos Complement to Decimal

- Same operation: negate the bits, and add one

```
1011
```

Twos Complement to Decimal

- Same operation: negate the bits, and add one

```
~1011 =
 0100
```

Twos Complement to Decimal

- Same operation: negate the bits, and add one

`0100`

Twos Complement to Decimal

- Same operation: negate the bits, and add one

```
0100 + 1 =
0101
```

Twos Complement to Decimal

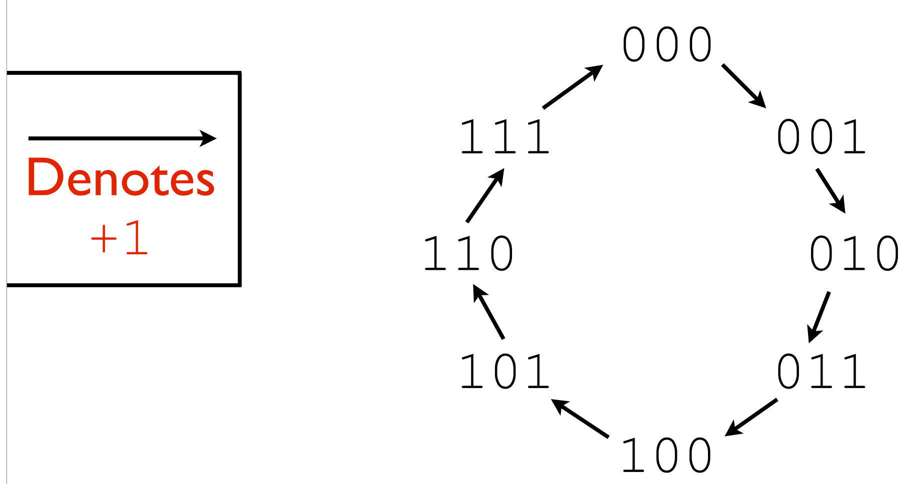
- Same operation: negate the bits, and add one

```
0100 + 1 =
0101 =
-5
```

We started with `1011` - negative

Intuition

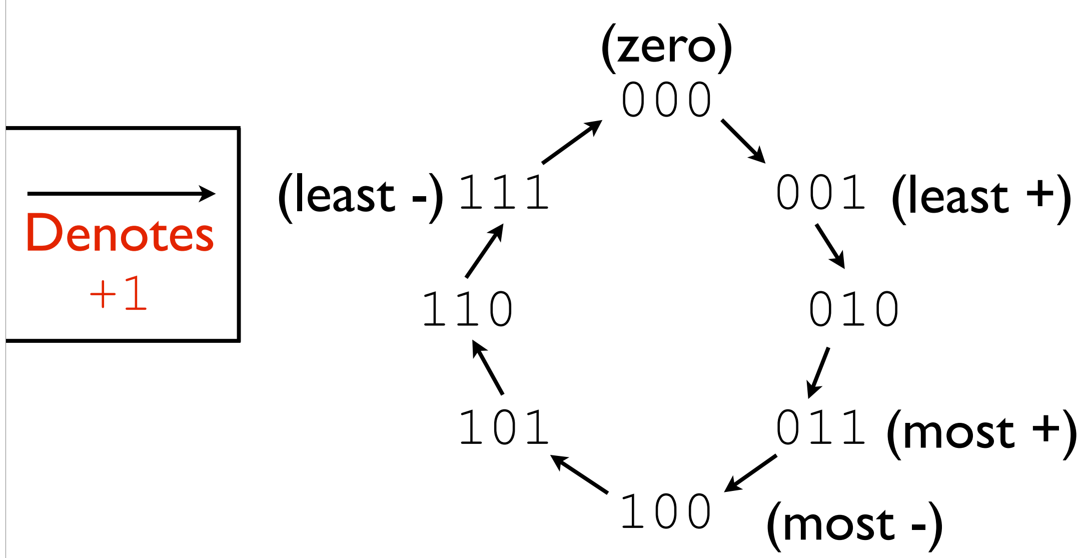
- Modular arithmetic, with the convention that a leading 1 bit means negative



-This is the intuition from Wikipedia, which makes a whole lot more sense

Intuition

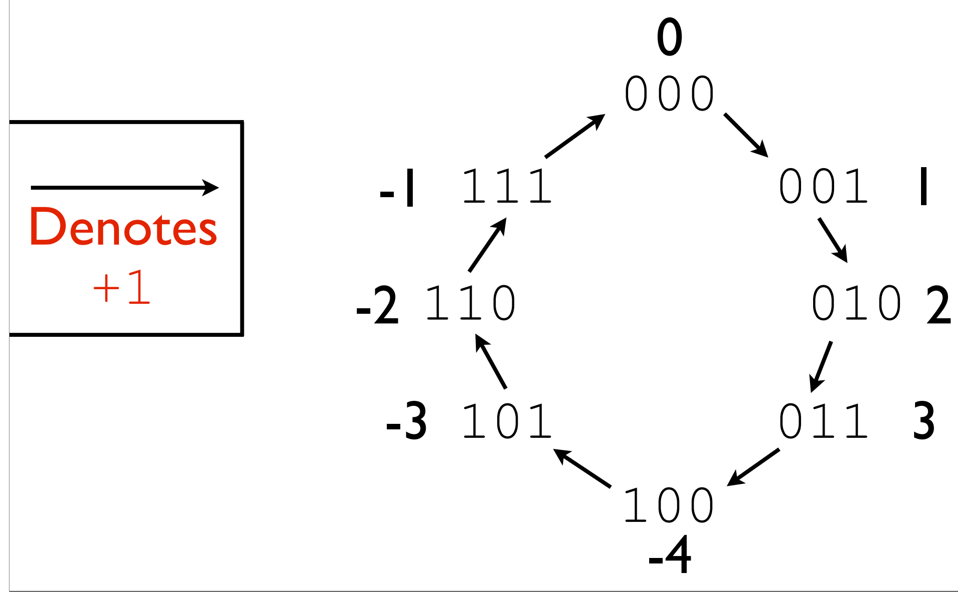
- Modular arithmetic, with the convention that a leading `1` bit means negative



-This is the intuition from Wikipedia, which makes a whole lot more sense
-There is still a lot of detail missing here - it's not necessary to understand in order to work with this. There is actually quite a bit of mathematics behind why this works

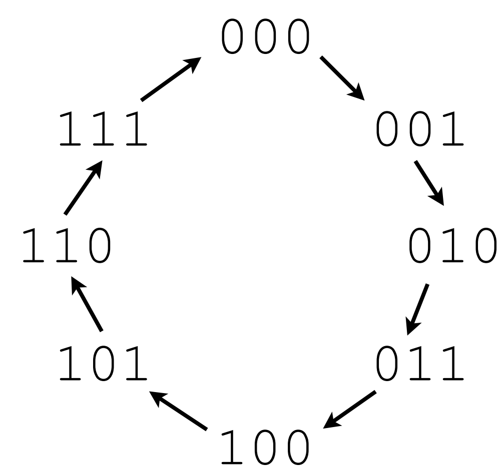
Intuition

- Modular arithmetic, with the convention that a leading `1` bit means negative



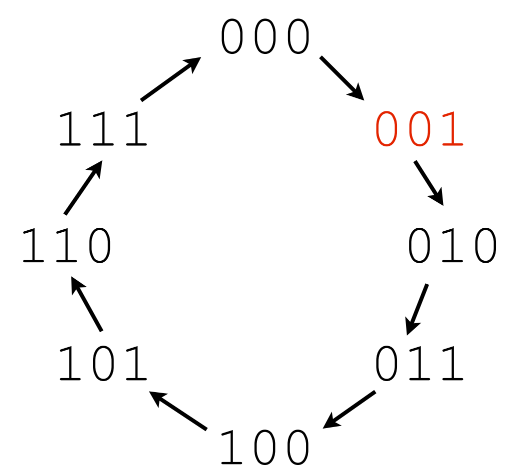
-This is the intuition from Wikipedia, which makes a whole lot more sense
-There is still a lot of detail missing here – it's not necessary to understand in order to work with this. There is actually quite a bit of mathematics behind why this works

Negation of 1



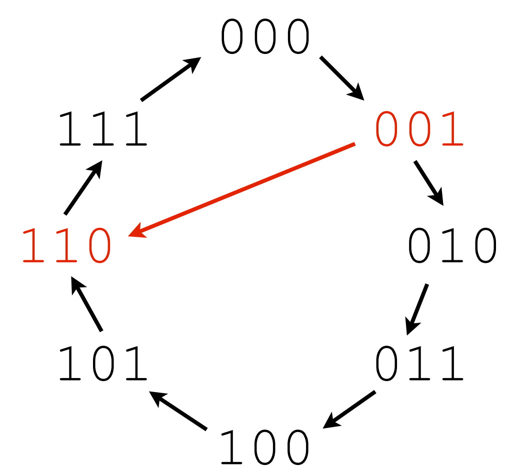
–Take our wheel from before

Negation of 1



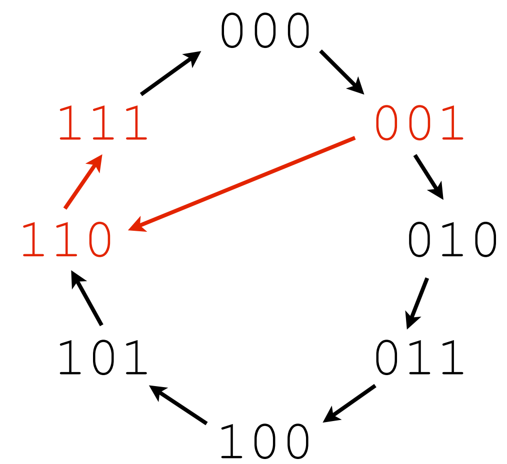
–This is 1

Negation of 1



–Inverted bits

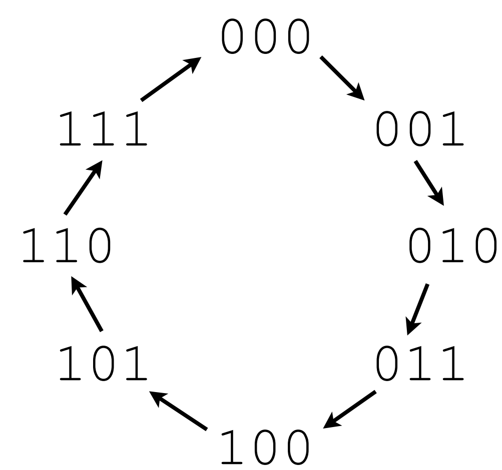
Negation of 1



–Add 1
–This is exactly what we expected – binary 111 represents decimal –1

Consequences

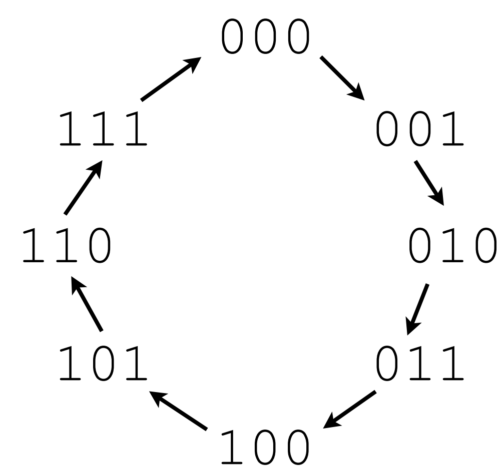
- What is the negation of `000`?



-Negate all bits: 000 -> 111
-Add one: 000
-Technically, adding one resulted in 1000, but that got cut off

Consequences

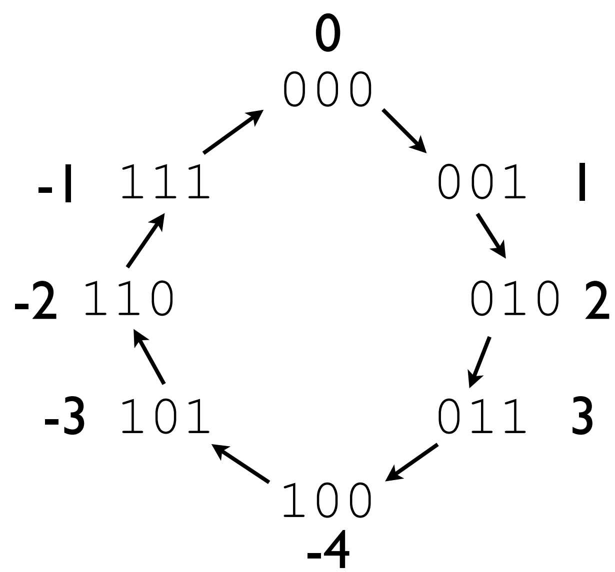
- What is the negation of `100`?



-Negate all bits: 100 -> 011
-Add one: 100
-Uh oh...this states that the negation of -4 is -4.
-Underlying problem is that we don't have a representation for 4 with just three bits

Arithmetic Shift Right

- **Not exactly** division by a power of two
- Consider -3 / 2



–101 (–3) shifted right yields 110 (–2), NOT 111 (–1) as expected from typical integer division
–Integer division rounds towards zero, whereas shift right rounds towards negative infinity
–This means they work _identically_ for positive values, but not for negative values (also meaning they are always the same for _unsigned_ values)

Addition

Building Up Addition

- Question: how might we add the following, in decimal?

```
 986
+123
----
   ?
```

Building Up Addition

- Question: how might we add the following, in decimal?

```
  986
 +123
 ----
    ?
```

			`6`
			`+3`
			`--`
			`?`

Building Up Addition

- Question: how might we add the following, in decimal?

```
 986
+123
----
   ?
```

		8 +2 -- ?	6 +3 -- 9

Building Up Addition

- Question: how might we add the following, in decimal?

```
 986
+123
----
   ?
```

	Carry: 1	8 +2 -- 0	6 +3 -- 9

Building Up Addition

- Question: how might we add the following, in decimal?

```
 986
+123
----
   ?
```

	1 9 +1 -- ?	8 +2 -- 0	6 +3 -- 9

Building Up Addition

- Question: how might we add the following, in decimal?

```
 986
+123
----
   ?
```

Carry: 1			
	1 9 +1 -- 1	8 +2 -- 0	6 +3 -- 9

Building Up Addition

- Question: how might we add the following, in decimal?

```
 986
+123
----
   ?
```

1 +0 -- 1	1 9 +1 -- 1	8 +2 -- 0	6 +3 -- 9

Core Concepts

- We have a “primitive” notion of adding single digits, along with an idea of *carrying* digits
- We can build on this notion to add numbers together that are more than one digit long

Now in Binary

- Arguably simpler - fewer one-bit possibilities

0 +0 -- ?	0 +1 -- ?	1 +0 -- ?	1 +1 -- ?

Now in Binary

- Arguably simpler - fewer one-bit possibilities

0 +0 -- 0	0 +1 -- 1	1 +0 -- 1	1 +1 -- 0 Carry: 1

Chaining the Carry

- Also need to account for any input carry

0 0 +0 -- 0	0 0 +1 -- 1	0 1 +0 -- 1	0 1 +1 -- 0 Carry: 1
1 0 +0 -- 1	1 0 +1 -- 0 Carry: 1	1 1 +0 -- 0 Carry: 1	1 1 +1 -- 1 Carry: 1

Adding Multiple Bits

- How might we add the numbers below?

```
  011
 +001
------
```

Adding Multiple Bits

- How might we add the numbers below?

```
    0
  011
 +001
------
```

-Need an initial carry-in of zero

Adding Multiple Bits

- How might we add the numbers below?

```
   10
  011
 +001
------
    0
```

-Need an initial carry-in of zero

Adding Multiple Bits

- How might we add the numbers below?

```
  110
  011
 +001
------
   00
```

-Need an initial carry-in of zero

Adding Multiple Bits

- How might we add the numbers below?

```
 0110
  011
 +001
------
  100
```

-Need an initial carry-in of zero

Adding Multiple Bits

- How might we add the numbers below?

```
 0110
  011
 +001
------
  100
```

Output Carry Bit

Result Bits

-Need an initial carry-in of zero

Another Example

```
  111
 +001
------
```

Another Example

```
    0
  111
 +001
------
```

Another Example

```
  10
 111
+001
------
   0
```

Another Example

```
  110
  111
 +001
------
   00
```

Another Example

```
 1110
  111
 +001
------
  000
```

Output Carry Bit

Result Bits

-Now we have an output carry bit of 1. What does this mean?

Output Carry Bit Significance

- For unsigned numbers, it indicates if the result did not fit all the way into the number of bits allotted
- May be an error condition for software

Signed Addition

- Question: what is the result of the following operation?

```
 011
+011
----
   ?
```

Signed Addition

- Question: what is the result of the following operation?

```
 011
+011
----
0110
```

–If these are treated as signed numbers in two's complement, then we need a leading 0 to indicate that this is a positive number
–Truncated to three bits, the result is a negative number!

Overflow

- In this situation, *overflow* occurred: this means that both the operands had the same sign, and the result's sign differed

```
 011
+011
----
 110
```

- Possibly a software error

Overflow vs. Carry

- These are **different ideas**
  - Carry is relevant to **unsigned** values
  - Overflow is relevant to **signed** values

`111`
`+001`
`----`
`000`
	`011`
	`+011`
	`----`
	`110`
		`111`
		`+100`
		`----`
		`011`
			`001`
			`+001`
			`----`
			`010`

No Overflow; Carry	Overflow; No Carry	Overflow; Carry	No Overflow; No Carry

-As to when is it a problem, this all depends on exactly what it is you're doing

Subtraction

Subtraction

- Have been saying to invert bits and add one to second operand
- Could do it this way in hardware, but there is a trick

```
 001
-001
----
   ?
```

Hint: these two questions are equivalent

```
 001
+111
----
   ?
```

Subtraction Trick

- Assume we can cheaply invert bits, but we want to avoid adding twice (once to add 1 and once to add the other result)
- How can we do this easily?

Subtraction Trick

- Assume we can cheaply invert bits, but we want to avoid adding twice (once to add 1 and once to add the other result)
- How can we do this easily?
  - Set the initial carry to `1` instead of `0`

Subtraction Example

```
 0101
-0011
 ----
```

Subtraction Example

```
  0101
 -0011
  ----
```

Invert `0011` $\longrightarrow$

Subtraction Example

```
  0101      Invert 0011
 -0011     ------------->  1100
  ----
```

Subtraction Example

```
  0101
 -0011
  ----
```

$\xrightarrow{\text{Invert } \texttt{0011}}$ `1100` $\xrightarrow{\text{Equivalent to}}$

Subtraction Example

```
                                              1
  0101   Invert 0011        Equivalent to   0101
 -0011   ----------->  1100 ------------>  +1100
  ----                                      ----
```

-An initial carry-in of 1 is equivalent to adding 1 and then adding the other operand

Subtraction Example

```
                                                   [1]1011
  0101     Invert 0011            Equivalent to       0101
 -0011   --------------->  1100  --------------->    +1100
 ----                                                 ----
                                                      0010
```

-An initial carry-in of 1 is equivalent to adding 1 and then adding the other operand