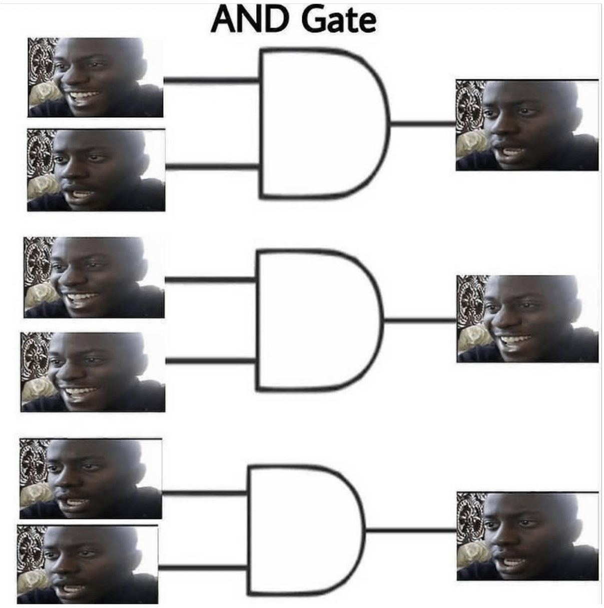


EECS 370 - Lecture 8

Combinational Logic

Announcements

- P1 and 2
  - P1 s+m due today
  - P2 posted by tomorrow
- HW 1
  - Due Monday
- Lab 4 meets Fr/M
  - Don't forget the pre-lab quiz tonight!

What do object files look like?

```
extern int X;
extern void foo();
int Y;

void main() {
   Y = X + 1;
   foo();
}
```

```
extern int Y;
int X;

void foo() {
   Y *= 2;
}
```

"extern" means defined in another file

Compile

```
.main:
LDUR    X1, [XZR, X]
ADDI    X9, X1, #1
STUR    X9, [XZR, Y]
BL      foo
HALT
```

Compile

```
.foo:
LDUR    X1, [XZR, Y]
LSL     X9, X1, #1
STUR    X9, [XZR, Y]
BR      X30
```

Uh-oh!
Don't know address of X, Y, or foo!

Linking

```
.main:
LDUR    X1, [XZR, X]
ADDI    X9, X1, #1
STUR    X9, [XZR, Y]
BL      foo
HALT
```

Assemble

???

```
.foo:
LDUR    X1, [XZR, Y]
LSL     X9, X1, #1
STUR    X9, [XZR, Y]
BR      X30
```

Assemble

???

What needs to go in this intermediate "object file"?

LINK

LINK

NOTE: this will actually be in machine code, not assembly

```
LDUR    X1, [XZR, #40]
ADDI    X9, X1, #1
STUR    X9, [XZR, #36]
BL      #2
HALT
LDUR    X1, [XZR, #36]
LSL     X9, X1, #1
STUR    X9, [XZR, #36]
BR      X30
// Addr #36 starts here
```

Linking

```
.main:
LDUR    X1, [XZR, X]
ADDI    X9, X1, #1
STUR    X9, [XZR, Y]
BL      foo
HALT
```

Assemble

???

LINK

```
LDUR    X1, [XZR, #40]
ADDI    X9, X1, #1
STUR    X9, [XZR, #36]
BL      #2
HALT
LDUR    X1, [XZR, #36]
LSL     X9, X1, #1
STUR    X9, [XZR, #36]
BR      X30
```

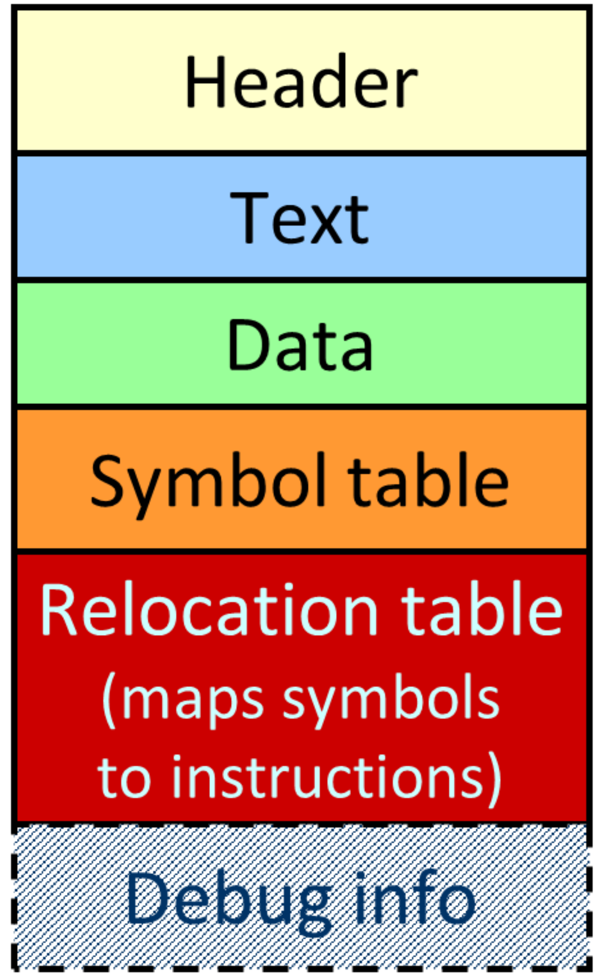
We need:

- the assembled machine code:
- list of instructions that need to be updated once addresses are resolved
- list of symbols to cross-ref

What do object files look like?

- Since we can't make executable, we make an object file
- Basically, includes the machine code that will go in the executable
  - Plus extra information on what we need to modify once we stitch all the other object files together
- Looks like this ->

Object code format



We won't discuss "Debug" much. Get's included when you compile with "-g" in gcc

Assembly → Object file - example

```
extern int G;
extern void B();
int X = 3;
main() {
   int y = G + 1;
   B();
}
```

LDUR	X1, [XZR, G]
ADDI	X9, X1, #1
BL	B

Header	Name	foo	
	Text size	0x0C //probably bigger	
	Data size	0x04 //probably bigger	
Text	Address	Instruction	
	0	LDUR X1, [XZR, G]	
	4	ADDI X9, X1, #1	
	8	BL B	
Data	0	X	3
Symbol table	Label	Address	
	X	0	
	B	-	
	main	0	
	G	-	
Reloc table	Addr	Instruction type	Dependency
	0	LDUR	G
	8	BL	B

Assembly → Object file - example

Header: keeps track of size of each section

```
extern in
extern vo
int X = 3;
main() {
  Y = G + 1;
  B();
}
```

Header	Name	foo	
	Text size	0x0C //probably bigger	
	Data size	0x04 //probably bigger	
Text	Address	Instruction	
	0	LDUR X1, [XZR, G]	
	4	ADDI X9, X1, #1	
	8	BL B	
Data	0	X	3
Symbol table	Label	Address	
	X	0	
	B	-	
	main	0	
	G	-	
Reloc table	Addr	Instruction type	Dependency
	0	LDUR	G
	8	BL	B

```
LDUR    X1, [XZR, G]
ADDI    X9, X1, #1
BL      B
```

Assembly → Object file - example

```
extern int G;
extern void B();
int X = 3
main() {
  Y = G +
  B();
}
```

Text:
machine code

Header	Name	foo	
	Text size	0x0C //probably bigger	
	Data size	0x04 //probably bigger	
Text	Address	Instruction	
	0	LDUR X1, [XZR, G]	
	4	ADDI X9, X1, #1	
	8	BL B	
Data	0	X	3
	Label	Address	
Symbol table	X	0	
	B	-	
	main	0	
	G	-	
Reloc table	Addr	Instruction type	Dependency
	0	LDUR	G
	8	BL	B

```
LDUR    X1, [XZR, G]
ADDI    X9, X1, #1
BL      B
```

Simplifying Assumption for EECS370

All globals and static locals (initialized or not) go in the data segment

Assembly → Object file - example

```
extern int G;
extern void B();
int X = 3;
main() {
   Y = G + 1;
   B();
}
```

Data: initialized globals and static locals

LDUR	X1, [XZR, G]
ADDI	X9, X1, #1
BL	B

Header	Name	foo	
	Text size	0x0C //probably bigger	
	Data size	0x04 //probably bigger	
Text	Address	Instruction	
	0	LDUR X1, [XZR, G]	
	4	ADDI X9, X1, #1	
	8	BL B	
Data	0	X	3
Symbol table	Label	Address	
	X	0	
	B	-	
	main	0	
	G	-	
Reloc table	Addr	Instruction type	Dependency
	0	LDUR	G
	8	BL	B

Assembly → Object file - example

```
extern int G;
extern void B();
int X = 3;
main() {
  Y = G + 1;
  B();
}
```

Symbol table:
Lists all labels visible outside this file (i.e. function names and global variables)

LDUR
ADDI
BL

Header	Name	foo	
	Text size	0x0C //probably bigger	
	Data size	0x04 //probably bigger	
Text	Address	Instruction	
	0	LDUR X1, [XZR, G]	
	4	ADDI X9, X1, #1	
	8	BL B	
Data	0	X	3
Symbol table	Label	Address	
	X	0	
	B	-	
	main	0	
	G	-	
Reloc table	Addr	Instruction type	Dependency
	0	LDUR	G
	8	BL	B

Assembly → Object file - example

```
extern int G;
extern void B();
int X = 3;
main() {
  Y = G + 1;
  B();
}
```

Relocation Table:
list of instructions and data words that must be updated if things are moved in memory

Header	Name	foo	
	Text size	0x0C //probably bigger	
	Data size	0x04 //probably bigger	
Text	Address	Instruction	
	0	LDUR X1, [XZR, G]	
	4	ADDI X9, X1, #1	
	8	BL B	
Data	0	X	3
	Label	Address	
Symbol table	X	0	
	B	-	
	main	0	
	G	-	
Reloc table	Addr	Instruction type	Dependency
	0	LDUR	G
	8	BL	B

Class Problem 1

<u>Poll:</u> Which symbols will be put in the symbol table? (i.e. which "things" should be visible to all files?)

file1.c

```
extern void bar(int);
extern char c[];
int a;
int foo (int x) {
    int b;
    a = c[3] + 1;
    bar(x);
    b = 27;
}
```

file 1 – symbol table

sym	**loc**
a	data
foo	text
c	-
bar	-

file2.c

```
extern int a;
char c[100];
void bar (int y) {
   char e[100];
   a = y;
   c[20] = e[7];
}
```

file 2 – symbol table

sym	**loc**
c	data
bar	text
a	-

<u>Poll:</u> Which lines / instructions are in the relocation table? (i.e. which "things" need to be updated after linking?)

Class Problem 2

<u>file1.c</u>

```
1 extern void bar(int);
2 extern char c[];
3 int a;
4 int foo (int x) {
5     int b;
6     a = c[3] + 1;
7     bar(x);
8     b = 27;
9 }
```

file 1 - relocation table

line	type	dep
6	ldur	c
6	stur	a
7	bl	bar

<u>file2.c</u>

```
1 extern int a;
2 char c[100];
3 void bar (int y) {
4    char e[100];
5    a = y;
6    c[20] = e[7];
7 }
```

file 2 - relocation table

line	type	dep
5	stur	a
6	stur	c

Note: in a real relocation table, the “line” would really be the address in “text” section of the instruction we need to update.

Linker

- Stitches independently created object files into a single executable file (i.e., a.out)
  - Step 1: Take text segment from each .o file and put them together.
  - Step 2: Take data segment from each .o file, put them together, and concatenate this onto end of text segments.
- What about libraries?
  - Libraries are just special object files.
  - You create new libraries by making lots of object files (for the components of the library) and combining them (see ar and ranlib on Unix machines).

  - Step 3: Resolve cross-file references to labels
    - Make sure there are no undefined labels

Linker - Continued

- What kind of instructions get relocated?
- PC-relative branches? (if/else or loops)
  - No – amount we're branching forwards/backwards doesn't change after sliding instructions around

```
B.EQ        endLoop
```

- Local variable accesses?
  - No – memory addresses are relative to stack pointer (SP) value
    - Relative offsets won't change
    - SP value will – but that's a dynamic value anyway, isn't encoded in instruction

```
sub     sp, sp, #16
mov     x0, #42
stur    x0, [sp, 0]
ldur    x1, [sp, 0]
```

- Global / static local variable access?
  - Yes – these use hardcoded addresses which change after linking

```
ldur    x2, [0, MY_VAR]
```

Loader

- Executable file is sitting on the disk
- Loader is software that puts the executable file code image into memory and asks the operating system to schedule it as a new process
  - Creates new address space for program large enough to hold text and data segments, along with a stack segment
  - Copies instructions and data from executable file into the new address space
  - Initializes registers (PC and SP most important)
- Take operating systems class (EECS 482) to learn more!

Summary (for your reference)

- Compiler converts a single source code file into a single assembly language file
- Assembler handles directives (.fill), converts what it can to machine language, and creates a checklist for the linker (relocation table). This changes each .s file into a .o file
- Assembler does 2 passes to resolve addresses, handling internal forward references
- Linker combines several .o files and resolves absolute addresses
- Linker enables separate compilation: Thus unchanged files, including libraries need not be recompiled.
- Linker resolves remaining addresses.
- Loader loads executable into memory and begins execution

Floating Point Arithmetic

See end of slides for bonus material (not covered in HW or exams)

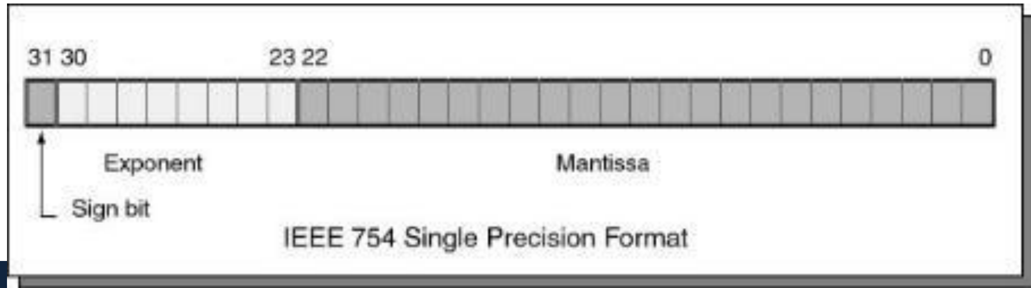
Why floating point

- Have to represent real numbers somehow
- Rational numbers
  - Ok, but can be cumbersome to work with
- Fixed point
  - Do everything in thousandths (or millionths, etc.)
  - Not always easy to pick the right units
  - Different scaling factors for different stages of computation
- **<u>Scientific notation: this is good!</u>**
  - Exponential notation allows HUGE dynamic range
  - Constant (approximately) relative precision across the whole range

IEEE Floating point format (single precision)

- Sign bit: (0 is positive, 1 is negative)
- Significand: (also called the *mantissa*; stores the 23 most significant bits after the decimal point)
- Exponent: used biased base 127 encoding
  - Add 127 to the value of the exponent to encode:
  - -127 → 00000000   1 → 10000000
  - -126 → 00000001   2 → 10000001
  - …   …
  - 0 → 01111111   128 → 11111111
- How do you represent zero ? Special convention:
  - Exponent: -127 (all zeroes), Significand 0 (all zeroes), Sign + or -

Some other exception cases (e.g. NaN) we won't cover



Floating Point Representation

$$10.625_{10} \Rightarrow 1010.101_2$$

- Step 1: convert from decimal to binary
  - 1st bit after "binary" point represents 0.5 (i.e. 2^{-1})
  - 2nd bit represents 0.25 (i.e. 2^{-2})
  - etc.

Floating Point Representation

$$10.625_{10} \Rightarrow 1010.101_2$$

$$1.010101 \times 2^3$$

- Step 2: normalize number by shifting binary point until you get 1.XXX * 2^Y

Floating Point Representation

$$10.625_{10} \Rightarrow 1010.101_{2}$$

$$1.010101 \times 2^{3}$$

This must be a 1!
So don't store it.

+/-	Exponent (3)	Significand (1010101)
1 bit	8 bits	23 bits

- Step 3: store relevant numbers in proper location (ignoring initial 1 of significand)

Floating Point Representation

$10.625_{10} \Rightarrow 1010.101_2$

1.010101×2^3

This must be a 1! So don't store it.

+/-	Exponent (3)	Significand (1010101)
1 bit	8 bits	23 bits

$10.625_{10} = 0 \quad 10000010 \quad 01010100000000000000000$

On Your Own:

- What is the value of the following IEEE 754 floating point encoded number?

1 = -
10000101 = 133 – 127 -> exponent 6
01011001 = mantissa

-1.01011001 x 2^6

-1010110.01
-(2^6+2^4+2^2+2^1+2^-2)
-(64+16+4+2+1/4)

-86.25

1	10000101	01011001000000000000000

What matters to a CS person?

- What happens if you add a big number to a small number?
  - E.g.

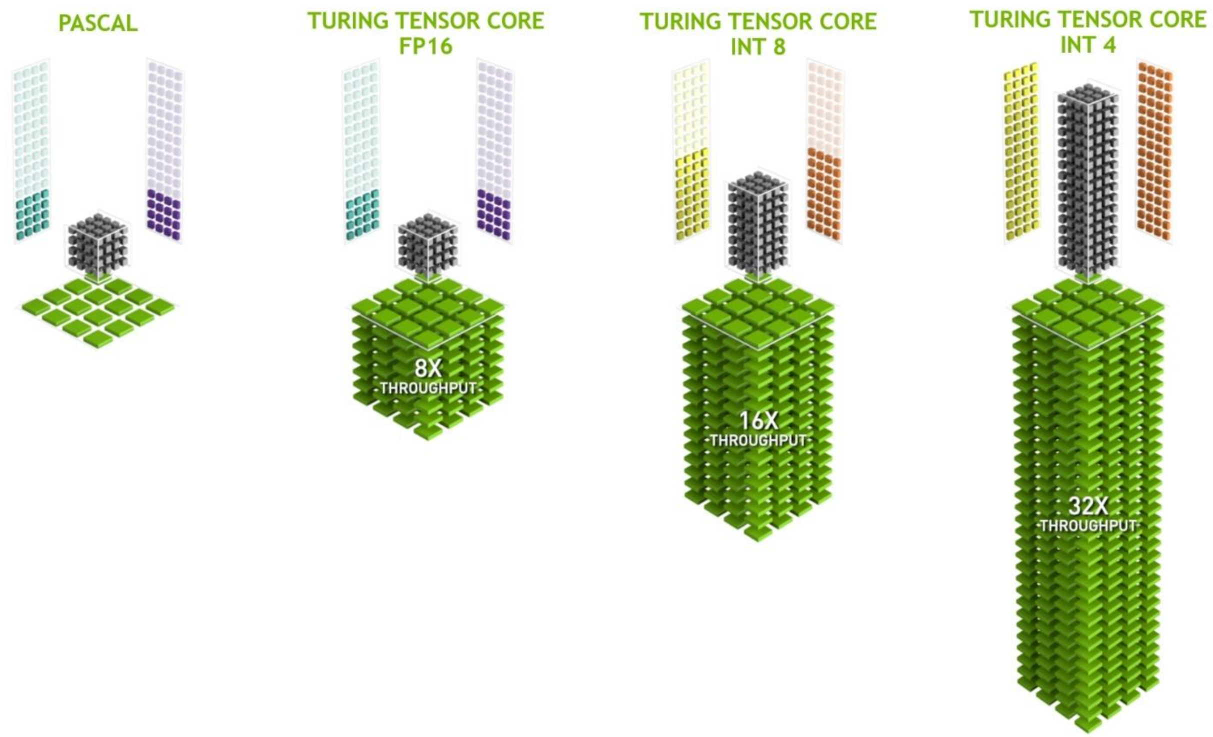
$$1000 + .00001$$

- The larger the exponent, the larger the "gap" between numbers that can be represented
- When the smaller number is added to the larger one, it can't be represented so precisely
  - Rounded down to zero: addition has no effect
  - Imagine having a loop do the above a million times: you'd end up with 1000 instead of 1010
  - Can be a big problem in scientific code
- You need to be aware of the issue.
  - This is why people often use “double” instead of “float”
  - The problem can still exist, it’s just less likely.

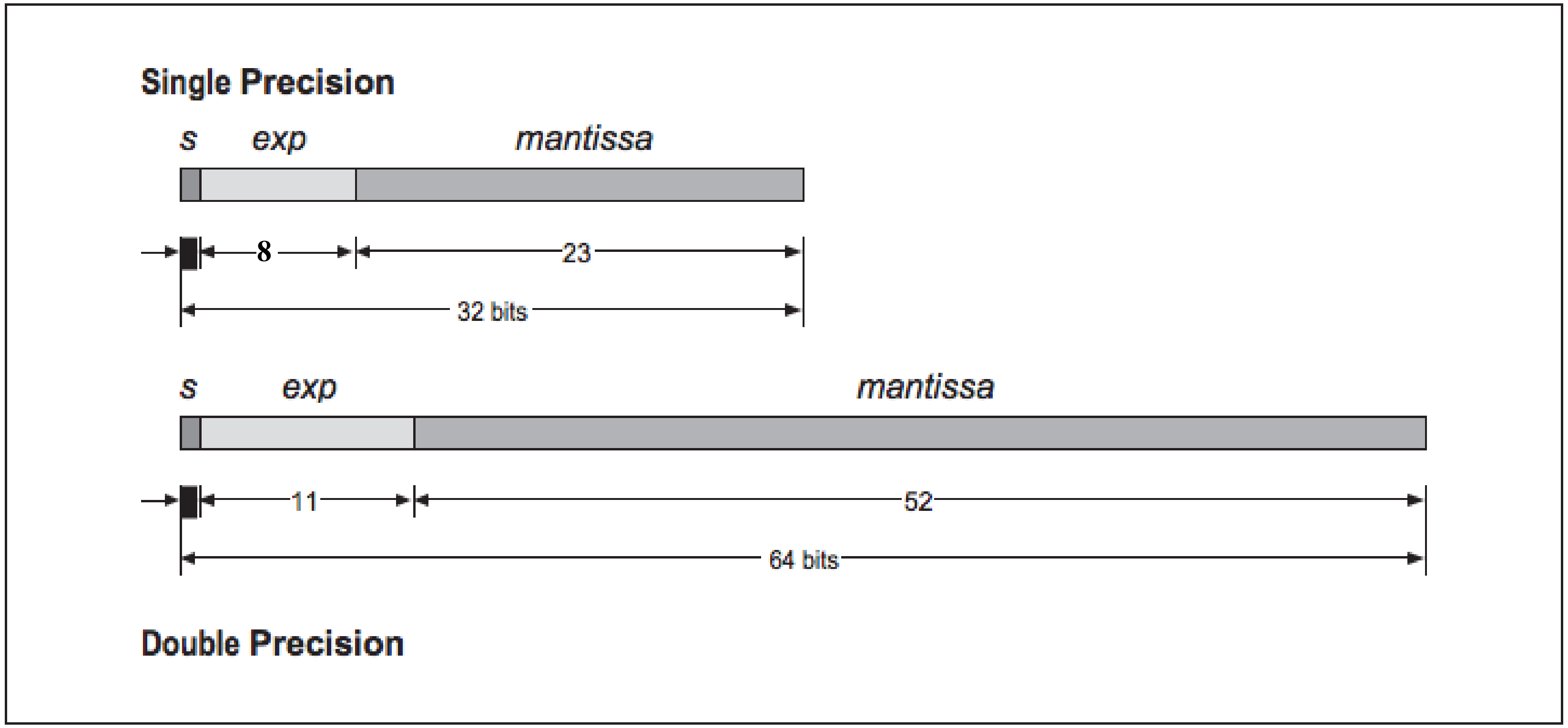
More precision and range

- We've described IEEE-754 binary32 floating point format, i.e. "single precision" ("float" in C/C++)
  - 24 bits precision; equivalent to about 7 decimal digits
  - $3.4 * 10^{38}$ maximum value
  - Good enough for most but not all calculations
- IEEE-754 also defines larger binary64 format, "double precision" ("double" in C/C++)
  - 53 bits precision, equivalent to about 16 decimal digits
  - $1.8 * 10^{308}$ maximum value
  - Most accurate physical values currently known only to about 47 bits precision, about 14 decimal digits
- Interestingly, there's been a surge in popularity for **lower** precision floating point lately. Why?

Low Precision FP Popular in accelerating AI



Single ("float") precision



Next few lectures: Digital Logic

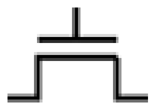
- Lectures 1-7:
  - LC2K and ARMv8/LEGv8 ISAs
  - Converting C to Assembly
  - Function Calls
  - Linking
- **Today:**
  - **Floating Point**
  - **Combinational Logic**
- Next lecture:
  - Sequential Logic

Up Until Now…

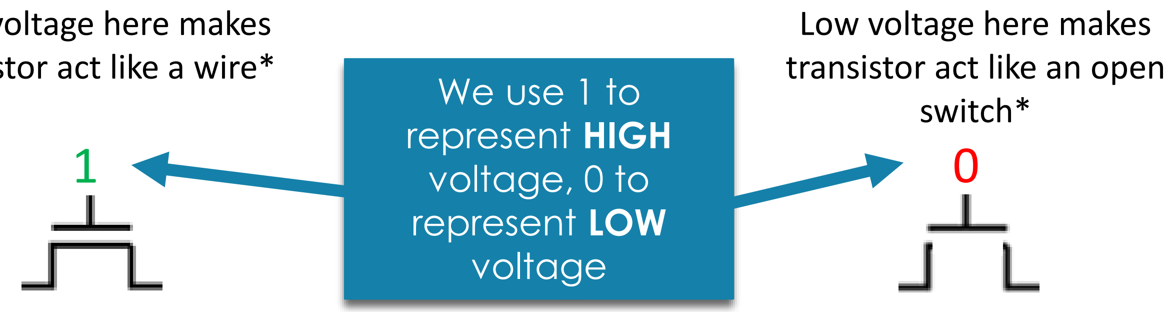
- We've covered high-level C code to an executable
    - Compilation
    - Assembly
    - Linking
    - Loading
- Now, we'll talk about the hardware that runs this code
    - First step: the basics of digital logic

Transistors

- At the heart of digital logic is the transistor
- Electrical engineers draw it like this



- The physics is complicated, but at the end of the day, all it is a **really** small and **really** fast electric switch



Yeah, yeah, circuits people. It's a lot more complicated than that. This abstraction is fine for 370.

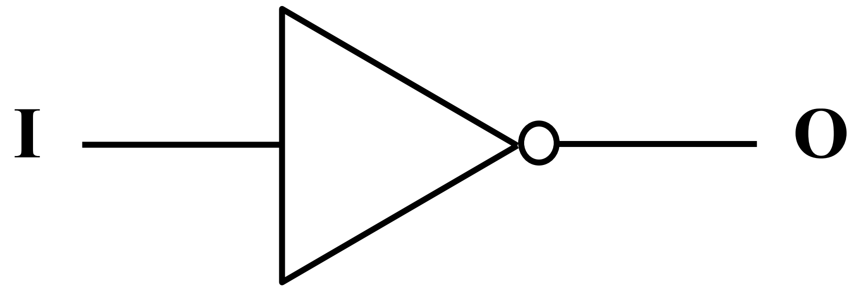
Basic gate: Inverter

CS abstraction
- logic function

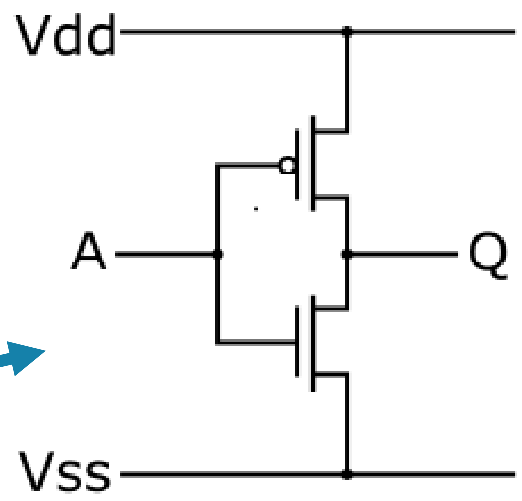
Truth Table

I	O
0	**1**
1	**0**

Schematic symbol (CS/EE)



We won't focus on this part in 370. Take 270 and 312 to learn more

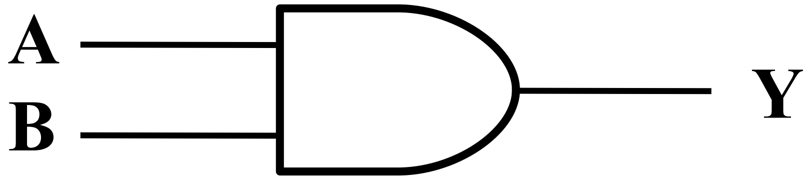


Basic gates: AND and OR

AND

Truth Table

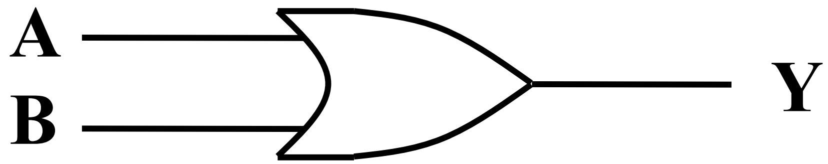
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1



OR

Truth Table

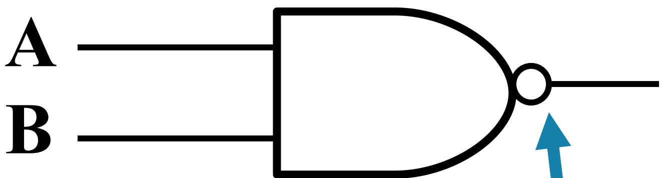
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1



Basic gate: NAND

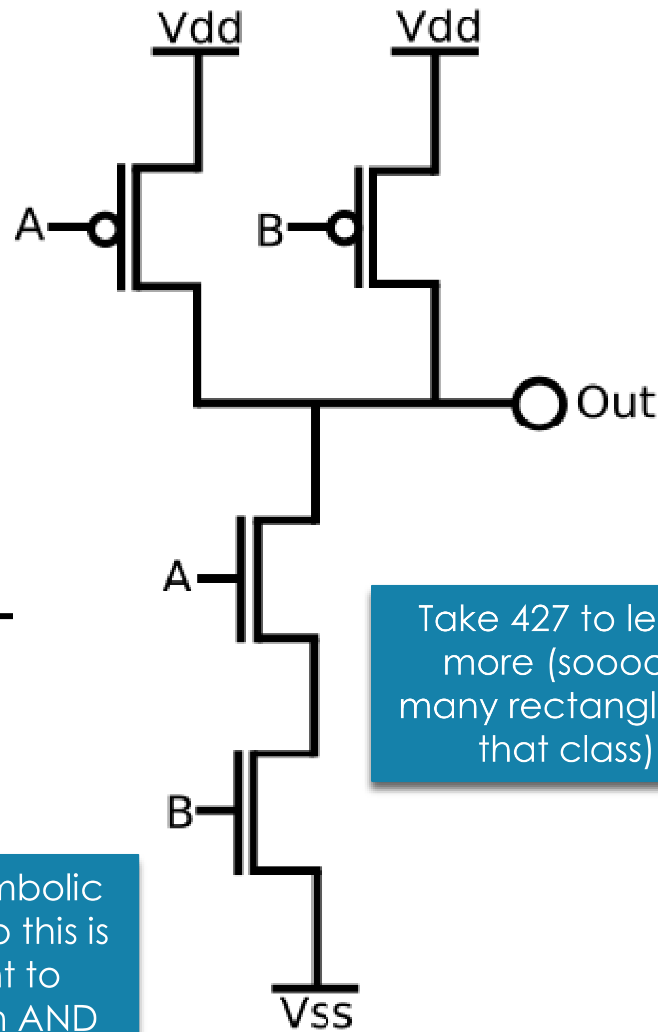
Truth Table

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

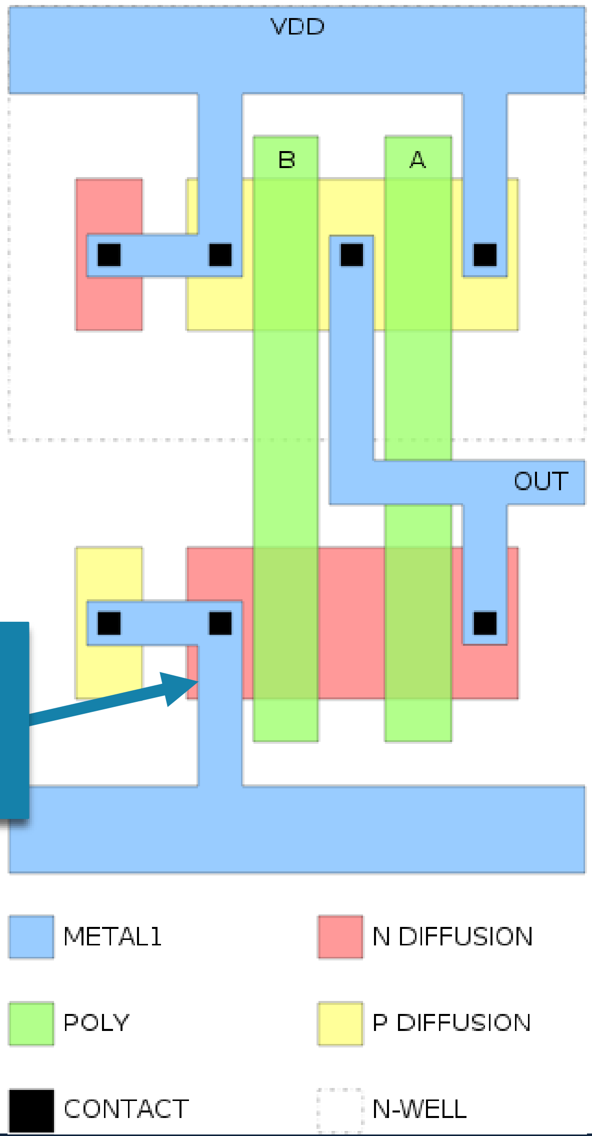


Bubble is symbolic for "invert", so this is equivalent to negating an AND gate

Transistor-level schematic



Layout schematic

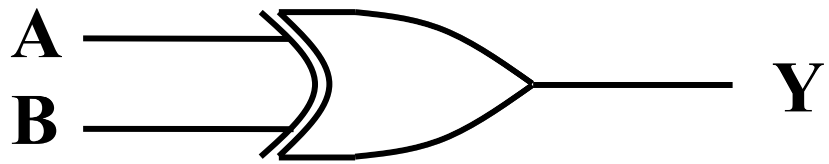


Take 427 to learn more (sooooo many rectangles in that class)

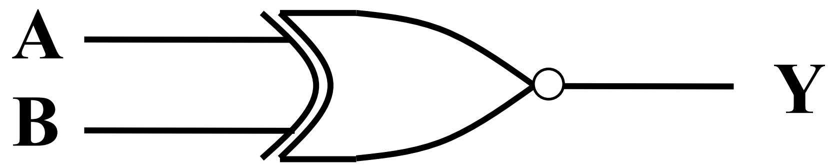
Basic gate: XOR (eXclusive OR)

Truth Table

A	B	Y
0	0	
0	1	
1	0	
1	1	



XNOR Gate



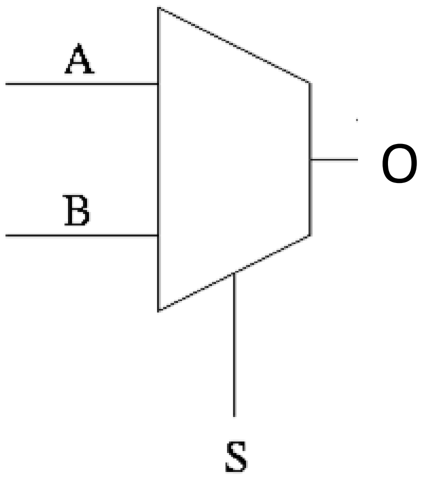
Building Complexity: Selecting

- We want to design a circuit that can select between two inputs (multiplexer or **mux**)
- Let's do a one-bit version
    1. Draw a truth table

A	B	S	O
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Poll: How do we fill in the truth table for this?

Symbol

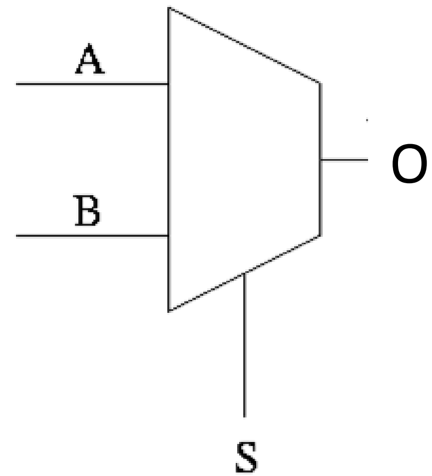


O = S ? B : A

Building Complexity: Selecting

- We want to design a circuit that can select between two inputs (multiplexor or **mux**)
- Let’s do a one-bit version
  1. Draw a truth table

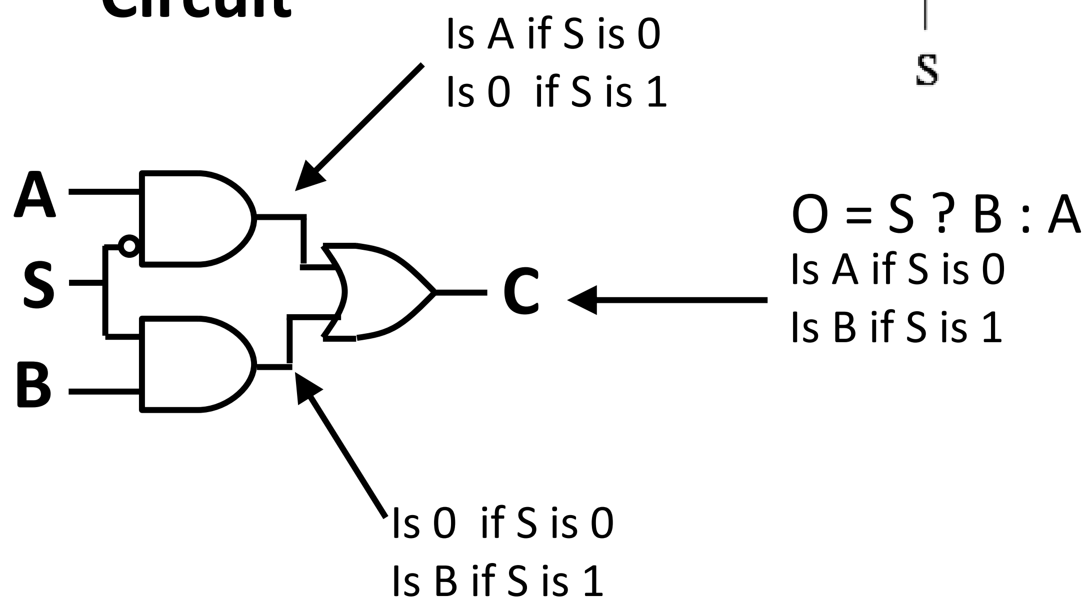
Symbol



A	B	S	O
0	**0**	**0**	**0**
0	**0**	**1**	**0**
0	**1**	**0**	**0**
0	**1**	**1**	**1**
1	**0**	**0**	**1**
1	**0**	**1**	**0**
1	**1**	**0**	**1**
1	**1**	**1**	**1**

Muxes are universal! A 2^N entry truth table can be implemented by passing each ouput value into an input of a 2^N–to-1 mux

Circuit



Building Complexity: Addition

- We want to design a circuit that performs binary addition
- Let's start by adding two bits
  - Design a circuit that takes two bits (A and B) as input
    - Generates a sum and carry bit (S and C)

1. Make a truth table
2. Design a circuit

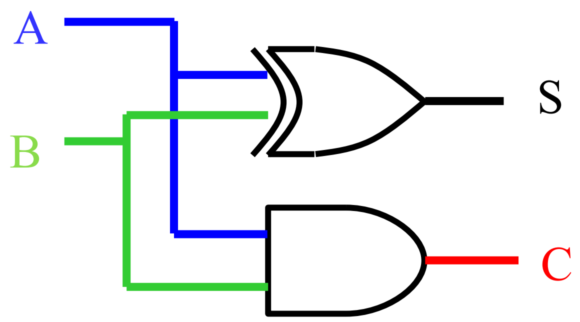
$$
\begin{array}{r}
10011 \\
+\,00110 \\
\hline
\end{array}
$$

A	B	C	S

Building Complexity: Addition

- We want to design a circuit that performs binary addition
- Let's start by adding two bits
  - Design a circuit that takes two bits (A and B) as input
    - Generates a sum and carry bit (S and C)
  1. Make a truth table
  2. Design a circuit

```
 0 1 1 0
  1 0 0 1 1
+ 0 0 1 1 0
-----------
  1 1 0 0 1
```

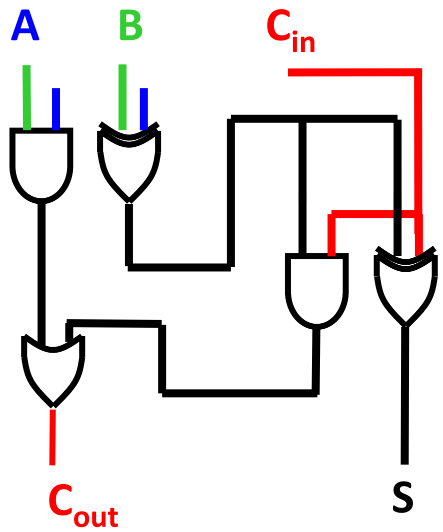


A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Building Complexity: Addition

	0	1	1	0	
	1	0	0	1	1
+	0	0	1	1	0
	1	1	0	0	1

- Now we can add two bits, but how do we deal with carry bits?
- This is a **full adder**
  - We have to design a circuit that can add three bits
    - Inputs: A, B, Cin
    - Outputs: S, Cout
  1. Design a truth table
  2. Circuit
- This is a **full adder**



Cin	A	B	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Next Time

- Logic circuits that "remember"
  - Aka "sequential logic"

BONUS Floating Point Slides

Bonus slides – this material is not testable

Not testable

- This material is here for those folks that may care.
  - **You *may* find it useful when considering the gap between representations**
  - But the material isn't directly testable.

- It is interesting if you are into that kind of thing.

- It can be useful if you are going to do scientific programming for a living.

- So it is provided as a reference, but isn't part of the class (we may cover a bit of it in lecture if we have time)

Not testable

Floating point multiplication

- Add exponents (don’t forget to account for the bias of 127)
- Multiply significands (don’t forget the implicit **1** bits)
- Renormalize if necessary
- Compute sign bit (simple exclusive-or)

Floating point multiply

Not testable

$$10.625_{10} = 1010.101_2$$

$$10_{10} = 1010_2$$

0	10000010	01010100000000000000000
	+	×
0	10000010	01000000000000000000000
	-127	
0	10000101	10101001000000000000000

```
   1 0 1 0 1 0 1
×          1 0 1
-----------------
   1 0 1 0 1 0 1
 1 0 1 0 1 0 1 0 0
-----------------
 1 1 0 1 0 1 0 0 1
```

$$1101010.01_2$$
$$= 106.25_{10}$$

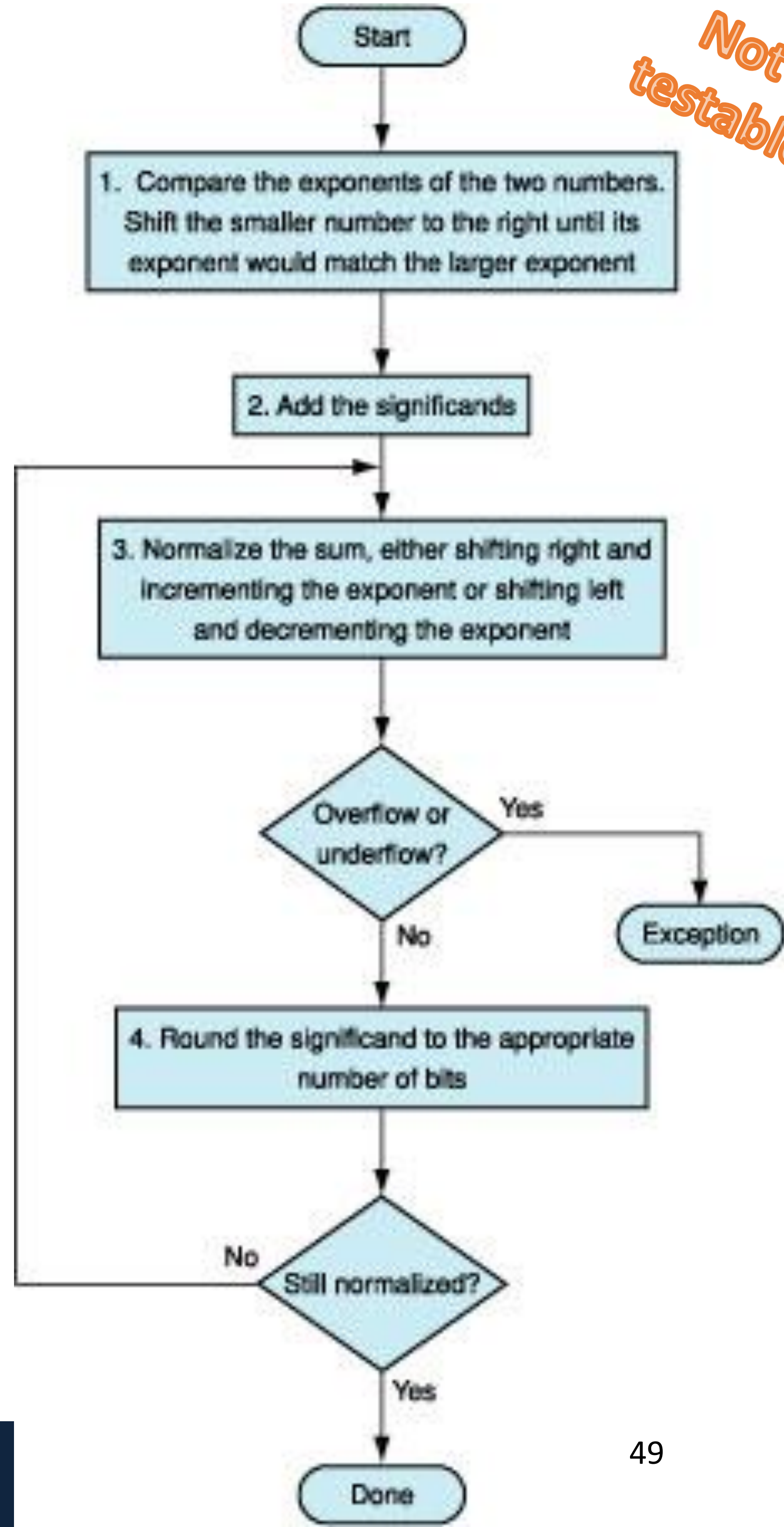
Floating point addition

Not testable

- More complicated than floating point multiplication!
- If exponents are unequal, must shift the significand of the smaller number to the right to align the corresponding place values
- Once numbers are aligned, simple addition (could be subtraction, if one of the numbers is negative)
- Renormalize (which could be messy if the numbers had opposite signs; for example, consider addition of +1.5000 and – 1.4999)
- Added complication: rounding to the correct number of bits to store could denormalize the number, and require one more step

Floating point Addition

1. Shift smaller exponent right to match larger.
2. Add significands
3. Normalize and update exponent
4. Check for “out of range”



Not testable

Class Problem

Not testable

Show how to add the following 2 numbers using IEEE floating point addition: 101.125 + 13.75

Class Problem

Not testable

101.125	0	10000101	10010100100000000000000
13.75	0	10000010	10111000000000000000000

Shift by 6-3 = 3

Shift mantissa by difference in exponent

00110111000000000000000

Note: When shifting to the right, the first shift should put the implicit 1, then 0's

Sum Significands

```
  1 1 0 0 1 0 1 0 0 1
+ 0 0 0 1 1 0 1 1 1 0
---------------------
  1 1 1 0 0 1 0 1 1 1
```

Sum didn't overflow, so no re-normalization needed

0	10000101	11001011100000000000000

= 114.875

Class Problem

Not testable

Show how to add the following 2 numbers using IEEE floating point addition: 117.125 + 13.75

Not testable

Class Problem

117.125	0	10000101	11010100100000000000000
13.75	0	10000010	10111000000000000000000

Shift by 6-3 = 3

Shift mantissa by difference in exponent

00110111000000000000000

Sum Significands

```
  1 1 1 0 1 0 1 0 0 1
+ 0 0 0 1 1 0 1 1 1 0
---------------------
1 0 0 0 0 0 1 0 1 1 1
```

Note: When shifting to the right, the first shift should put the implicit 1, then 0's

0	10000110	00000101110000000000000

= 130.875

Sum overflows, re-normalize by adding one to exponent and shifting mantissa by one