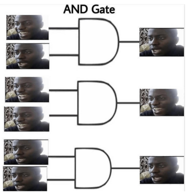


EECS 370 - Lecture 8

Combinational Logic

Announcements

- P1
  - Project 1 s + m due ~~tonight~~ **tomorrow at 11:55 pm**

- HW 1
  - Due next Monday
- Lab 4 meets Fr/M
- Get exam conflicts and SSD accommodations sent to us **ASAP**
- My group office hours moved to 3941 BBB for this week only

Reminder: Object File

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

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

Header	Name Text size Data size	foo 0x0C //probably bigger 0x04 //probably bigger	

Text	Address
	0
	4
	8
		Instruction
		LDUR X1, [XZR, G]
		ADDI X9, X1, #1
		BL B	

Data	0	X	3
Symbol table	Label X B main G	Address 0 - 0 -	
Reloc table	Addr 0 8	Instruction type LDUR BL	Dependency G B

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

- Determine the memory locations the code and data of each file will occupy
  - Each function could be assembled on its own
  - Thus, the relative placement of code/data is not known up to this point
  - Must relocate absolute references to reflect placement by the linker
    - PC-Relative Addressing (beq, bne): never relocate
    - Absolute Address (mov 27, #X): always relocate
    - External Reference (usually bl): always relocate
    - Data Reference (often movz/movk): always relocate
- Executable file contains no relocation info or symbol table these just used by assembler/linker

Loader

- Executable file is sitting on the disk
- 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

- 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)

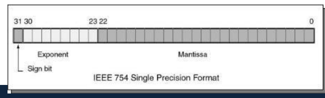
Why floating point

- 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
- **Scientific notation: this is good!**
  - 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

$10.625_{10} \Rightarrow 1010.101_2$

8 2 2^{-1} 2^{-2}

0.5 + 0.125 = 0.625

- 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

$10.625_{10} \Rightarrow 1010.101_2$

1.010101×2^3

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

Floating Point Representation

$10.625_{10} \Rightarrow 1010.101_2$

1.010101×2^3

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

2^7 2

10.625_{10} = 0 10000010 01010100000000000000000

Class Problem

Poll

- 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?

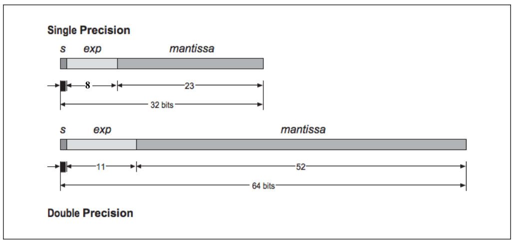
- What happens if you add a big number to a small number?
  - E.g.

    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
- It will be rounded down to zero: we end up with the same number
- This can be a real problem when writing scientific code.
  - For the above example, imagine you did that addition a million times
  - You'd still have 1000 when the answer should be 1,010
- So you need to be aware of the issue.
  - This is why most people use "double" instead of "float"
  - The problem can still exist, it's just less likely.

More precision and range

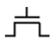
- We've described IEEE-754 binary32 floating point format, i.e. "single precision" ("float" in C/C++)
  - 24 bits precision; equivalent to about 7 decimal digits
  - $3.4 * 10^{38}$ maximum value
  - Good enough for most but not all calculations
- IEEE-754 also defines larger binary64 format, "double 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

Single ("float") precision

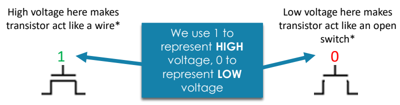


Transistors

- 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.

(Digital Logic Part : start .)

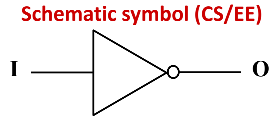
Basic gate: Inverter

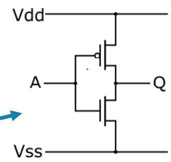
CS abstraction - logic function

Truth Table

I	O
0	1
1	0

Schematic symbol (CS/EE)





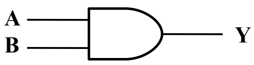
We won't focus on this part in 370. Take 270 and 312 to learn more

Basic gates: AND and OR

AND

Truth Table

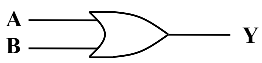
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1



OR

Truth Table

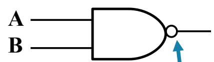
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1



Basic gate: NAND

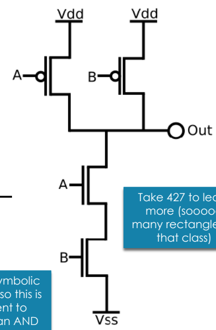
Truth Table

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

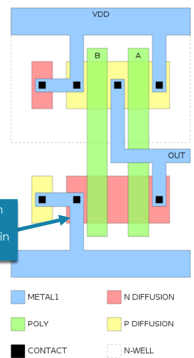


Bubble is symbolic for "invert", so this is equivalent to negating an AND gate

Transistor-level schematic



Layout schematic

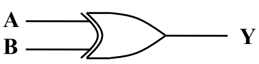


Take 427 to learn more (sooooo many rectangles in that class)

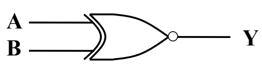
Basic gate: XOR (eXclusive OR)

Truth Table

A	B	Y
0	0	
0	1	
1	0	
1	1	



XNOR Gate



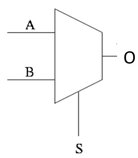
Building Complexity: Selecting

- 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	

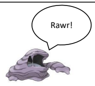
Poll: How do we fill in the truth table for this?

Symbol



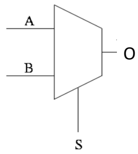
O = S ? B : A

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

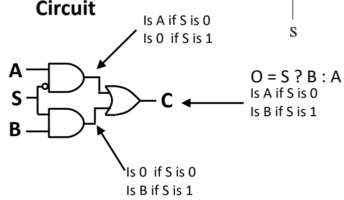
Symbol



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

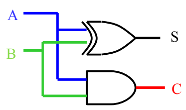
```
  10011
 +00110
```

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

```
  0110
  10011
 +00110
  11001
```



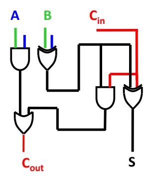
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Building Complexity: Addition

- 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**

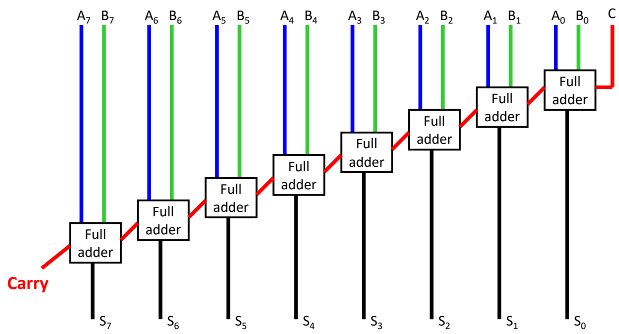
```
  0110
  10011
 +00110
  11001
```

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



8-bit Ripple Carry Adder

If we invert B's bits and set C to 1, we also have a subtractor! Why?

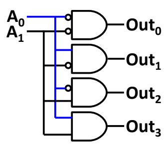


This will be very slow for 32 or 64 bit adds, but is sufficient for our needs

Building Complexity: Decoding

- Another common device is a decoder
  - Input: N-bit binary number
  - Output: 2^N bits, exactly one of which will be high
  - Allows us to **index** into things (like a register file)

Decoder



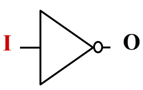
Poll: What will be the output for 101?

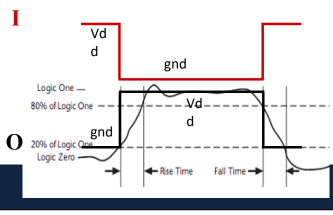
Live Poll + Q&A: slido.com #eecs370

Propagation delay in combinational gates

- Gate outputs do not change exactly when inputs do.
  - Transmission time over wires (~speed of light)
  - Saturation time to make transistor gate switch

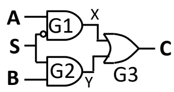
⇒ Every combinatorial circuit has a propagation delay (time between input and output stabilization)

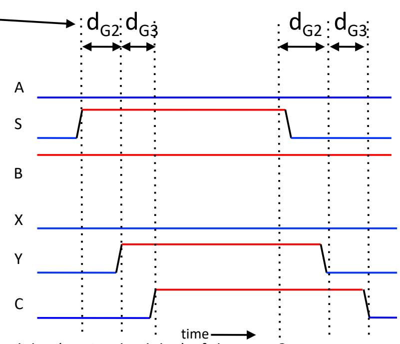




Timing in Combinational Circuits

MUX



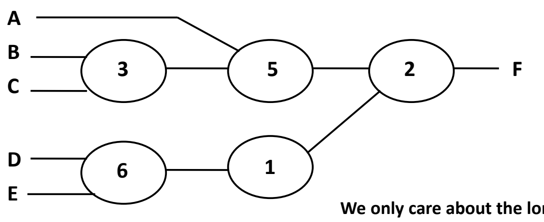


What is the input/output delay (or simply, delay) of the MUX?

What is the delay of this Circuit?

Each oval represents one gate, the type does not matter

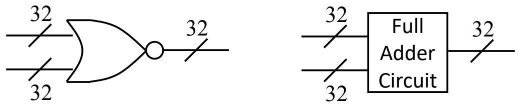
Poll : What is the delay?



We only care about the longest path, or critical path

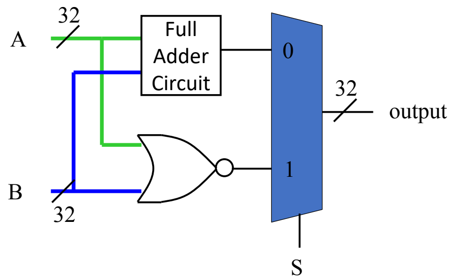
Exercise

- Use the blocks we have learned about so far (full adder, NOR, mux) to build this circuit
  - Input A, 32 bits
  - Input B, 32 bits
  - Input S, 1 bit
  - Output, 32 bits
  - When S is low, the output is A+B, when S is high, the output is NOR(a,b)
- Hint: you can express multi-bit gates like this:



Exercise

- This is a basic ALU (Arithmetic Logic Unit)
- It is the heart of a computer processor!



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)

Floating point multiplication

Not testable

- 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

$10.625_{10} = 1010.101_2$ → 0 | 10000010 | 01010100000000000000000

+ ×

$10_{10} = 1010_2$ → 0 | 10000010 | 01000000000000000000000

-127

0 10000101 10101001000000000000000

```
  1010101
×     101
  1010101
101010100
110101001
```

1101010.01_2

$= 106.25_{10}$

Floating point addition

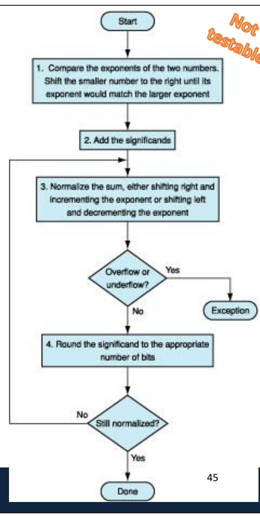
Not testable

- 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



Not testable

Class Problem

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

Not testable

Class Problem

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

Not testable

Class Problem

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

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

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
```

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

0 | 10000110 | 00000101110000000000000

= 130.875