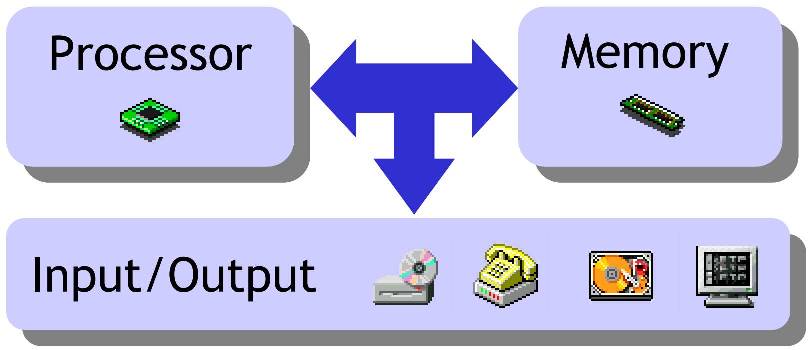
CS231: Computer Architecture I

Summer 2003

What is computer architecture about?

- Computer architecture is the study of building entire computer systems.



- There are numerous factors to consider, many of which are conflicting.
  - *Performance*, *price* and *reliability* are obviously vital concerns.
  - Systems should be *expandable* to accommodate future developments, but must also be *compatible* with existing technology.
  - *Power consumption* is especially important in the growing market of portable devices such as cell phones, PDAs, and MP3 players.

An example of architects hard at work

- Processor!
- Input!
- Output!
- Storage!
- Compatibility!
- Networking!
- Power consumption!

Why should you care?

- Computer science majors are often expected to know something about hardware and computer architecture.
  - What are caches, DDR SDRAMs, and AGPs?
  - Is a 3.0GHz processor or a 7200RPM hard disk worth it?

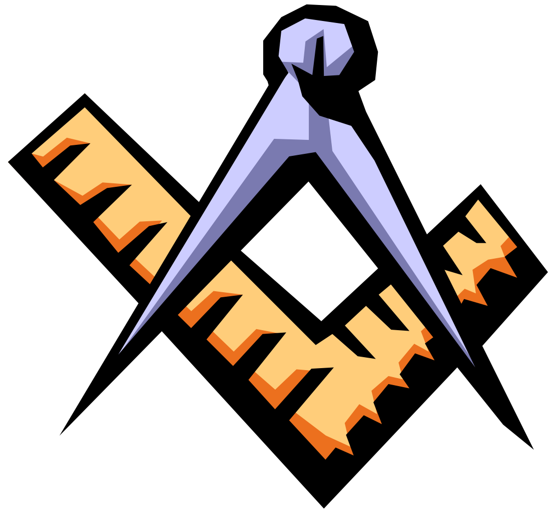
Architecture and programming

- Understanding architecture helps to explain why programming languages are designed the way they are.
  - What happens when we compile our source code?
  - Why is computer arithmetic sometimes wrong?
  - What is a bus error or segmentation fault?
- You can also learn how to make your code run faster.
  - Where and how you store your data makes a big difference.
  - Just rearranging the order of statements can sometimes help!
- A lot of software development requires knowledge of architecture.
  - Compilers generate optimized code for specific processors.
  - Operating systems manage hardware resources for applications.
  - Good I/O systems are important for databases and networking.

What is CS231 about?

- There's a lot of stuff to cover, and it takes more than one semester!
- In CS231 and CS232, we learn architecture bottom-up, from the simplest bits and binary operations all the way up to complete systems.
- CS231 is divided into roughly three parts.
  - We start with combinational circuits, which can compute relatively simple functions. Boolean algebra is the mathematical foundation upon which we build and analyze circuits.
  - Sequential circuits are more complex because they have memory. We'll see additional analysis and design techniques, based on state machines.
  - Finally, we will use both combinational and sequential circuits to build a simple, but complete, processor.

Important themes in CS231



- Choosing a good data representation can increase system performance, lower resource utilization and improve accuracy.
- We rely on mathematical techniques to describe and analyze circuits.
- Abstraction and hierarchical designs are critical to control complexity.
- There are often many design tradeoffs to consider.
  - Simplicity and low cost usually lead to low performance.
  - Higher performance comes with higher cost and greater complexity.
- These themes also pervade software development, and every other area of engineering.

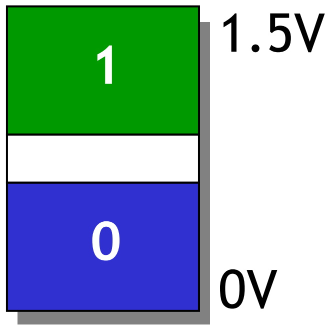
Helpful hints for CS231

- **Remember the big picture.**
  What are we trying to accomplish, and why?
- **Read the textbook.**
  Not everybody likes it, but it covers everything we talk about in class and has additional examples. Try it out if you have difficulty with any of the course material.
- **Talk to each other.**
  You can learn a lot from other students, both by asking and answering questions. Find some good partners for the assignments, and make sure you all understand what's going on.
- **Help us help you.**
  Come to lectures, sections and office hours. Send email or post on the newsgroup. Ask lots of questions! Check out the smashing web page:

http://www-courses.cs.uiuc.edu/~cs231

Representing information

- For the rest of the day, we'll discuss how computers use voltages to represent information.
  - In modern desktop processors the voltage is limited to around 1.5V to reduce power consumption.
  - However, it's hard to measure voltages precisely.
- It's more convenient for hardware designers to interpret analog voltages as just two discrete, or digital, values.
- How can two lousy values be useful for anything?
  - We can represent arbitrary numbers with sequences of just 0s and 1s.
  - We can also interpret voltages as "false" and "true" instead, and work with logical operations.



Decimal review

- Decimal numbers consist of digits from 0 to 9, each with a weight.

1	6	2	.	3	7	5	digits
100	10	1		1/10	1/100	1/1000	weights

- Notice that the weights are all powers of the base, which is 10.

1	6	2	.	3	7	5	digits
10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}	weights

- To find the decimal value of a number, you can multiply each digit by its weight and sum the products:

$$(1 \times 10^2) + (6 \times 10^1) + (2 \times 10^0) + (3 \times 10^{-1}) + (7 \times 10^{-2}) + (5 \times 10^{-3}) = 162.375$$

Binary numbers

- Binary, or base 2, numbers consist of only the digits 0 and 1. The weights are now powers of 2.
- For example, consider the binary number 1101.01:

1	1	0	1	.	0	1	binary digits, or bits
2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	weights in decimal

- The decimal value of 1101.01 is computed just like before:

$$(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) =$$
$$8 + 4 + 0 + 1 + 0 + 0.25 = 13.25$$

Some powers of 2		
$2^0 = 1$	$2^4 = 16$	$2^8 = 256$
$2^1 = 2$	$2^5 = 32$	$2^9 = 512$
$2^2 = 4$	$2^6 = 64$	$2^{10} = 1024$
$2^3 = 8$	$2^7 = 128$	

Converting decimal to binary

- To convert a decimal integer into binary, keep dividing by two until the quotient is 0. Then collect the remainders in reverse order.
- To convert a decimal fraction into binary, keep multiplying the fractional part by two until it becomes 0. Collect the integers in forward order.
- An example will make it all clear. Let’s convert 162.375 to binary.

162 / 2	=	81	rem 0		
81 / 2	=	40	rem 1		
40 / 2	=	20	rem 0		
20 / 2	=	10	rem 0		
10 / 2	=	5	rem 0		
5 / 2	=	2	rem 1		
2 / 2	=	1	rem 0		
1 / 2	=	0	rem 1		

0.375×2	=	0.750
0.750×2	=	1.500
0.500×2	=	1.000

- So $162.375_{10} = 10100010.011_2$

Why does this work?

- This same idea works for converting from decimal to any other base.
- Think about "converting" 162 from decimal to decimal:

$$\begin{aligned} 162 / 10 &= 16 \quad \text{rem } 2 \\ 16 / 10 &= 1 \quad \text{rem } 6 \\ 1 / 10 &= 0 \quad \text{rem } 1 \end{aligned}$$

- After each division, the remainder contains the rightmost digit of the dividend, while the quotient holds the remaining digits.
- Similarly when converting fractions, each multiplication strips off the leftmost digit as the integer result, leaving the remaining digits in the fractional part.

$$\begin{aligned} 0.375 \times 10 &= 3.750 \\ 0.750 \times 10 &= 7.500 \\ 0.500 \times 10 &= 5.000 \end{aligned}$$

Base 16 is useful too

- The hexadecimal system uses 16 digits:

  0 1 2 3 4 5 6 7 8 9 A B C D E F

- Hexadecimal is useful as a shorthand for binary numbers.
  – Since 16 = 2^4, one hex digit is equivalent to four bits (including leading 0s).
  – It's often easier to work with numbers like "B4" instead of "10110100".
- Hex shows up in many different contexts.
  – IP addresses, such as "80.AE.05.27".
  – RGB color triplets, like "C0C0FF".
- You can convert between base 10 and base 16 using the same method as for converting from decimal to binary.

Decimal	Binary	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Binary and hexadecimal conversions

- Converting from hexadecimal to binary is easy: replace each hex digit with its equivalent four-bit binary value.

$$261.A5_{16} = 2 \quad 6 \quad 1 \; . \; A \quad 5_{16}$$
$$= 0010 \quad 0110 \quad 0001 \; . \; 1010 \quad 0101_2$$

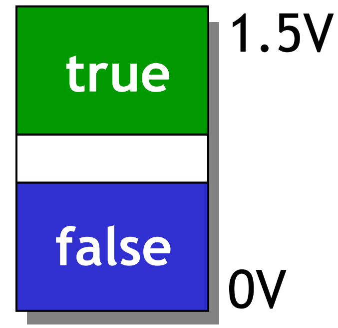
- To convert from binary to hexadecimal, partition the binary number into groups of four bits, starting from the point. (Add 0s to the ends if needed.) Then replace each four-bit group by the corresponding hex digit.

$$10110100.001011_2 = 1011 \quad 0100 \; . \; 0010 \quad 1100_2$$
$$B \quad 4 \; . \; 2 \quad C_{16}$$

Binary	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Boolean values

- It's also possible to think of voltages as representing the discrete logical values true and false.
- For various reasons that we'll see later, people often keep using digits instead.
  - 0 is false
  - 1 is true
- Many of you may have seen Boolean logic before, but we'll focus on its connection to computer hardware.
- Today we discuss functions on logical values, and show how those functions can be implemented in hardware.



How can we describe functions?

- Computers take inputs and produce outputs—just like functions.
- We can express mathematical functions in two ways.

An expression is finite, but not unique.

$$f(x,y) = 2x + 4x + 4y/2$$
$$= 6x + (4/2)y$$
$$= 6x + 2y$$
$$= \ldots$$

A function table is unique, but infinite.

x	y	f(x,y)
0	0	0
...	...	...
2	2	16
...	...	...
23	45	228
...	...	...

- We can represent logical functions in two analogous ways.
  - A Boolean expression is finite but not unique.
  - A truth table turns out to be unique *and* finite.

Basic Boolean operations

- Boolean expressions are created from three basic operations.

Operation:	AND (product) of two inputs	OR (sum) of two inputs	NOT (complement) of one input
Expression:	xy or x•y	x + y	x’ or $\overline{x}$

Truth table:

x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x + y
0	0	0
0	1	1
1	0	1
1	1	1

x	x’
0	1
1	0

Boolean operations are special

- The AND and OR operations are similar to multiplication and addition.
  - AND yields the same results as multiplication for the values 0 and 1.
  - OR is almost the same as addition, except for the case 1 + 1.

x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x + y
0	0	0
0	1	1
1	0	1
1	1	1

- This explains why we borrow the arithmetic symbols •, +, 0 and 1 for Boolean operations.
- But there are important differences too.
  - There are a finite number of Boolean values—just 0 and 1.
  - OR is not quite the same as addition, and NOT is a new operation.

Boolean expressions

- Using the basic operations, we can form more complex expressions.

$$f(x, y, z) = (x + y')z + x'$$

- Some terminology and notation:
  - f is the name of the function.
  - x, y and z are input variables, which range over 0 and 1.
  - A literal is any occurrence of an input variable or its complement. The function above has four literals: x, y', z and x'.
- Precedences are important, but not too difficult.
  - NOT has the highest precedence, followed by AND, and then OR.
  - Fully parenthesized, the expression above would be written:

$$f(x, y, z) = (((x + (y'))z) + x')$$

Truth tables

- A truth table shows all possible inputs and outputs of a Boolean function.
- Remember that each input variable ranges over just 0 and 1.
  - A function with *n* variables has 2^n possible combinations of inputs.
  - Since there are a finite number of values, truth tables themselves are finite.
- Inputs are listed in binary order—here in this example, from xyz=000 (0_{10}) to xyz=111 (7_{10}).
- You can find the output values by plugging the various input combinations into an expression.

x	y	z	f(x,y,z)
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

$f(x,y,z) = (x + y')z + x'$

Primitive logic gates

- Each basic operation can be implemented in hardware with a logic gate.

Operation:
- AND (product) of two inputs
- OR (sum) of two inputs
- NOT (complement) of one input

Expression:
- AND: xy or x•y
- OR: x + y
- NOT: x’ or $\overline{x}$

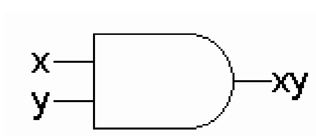
Truth table:

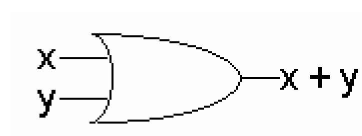
x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

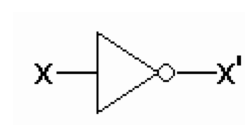
x	y	x + y
0	0	0
0	1	1
1	0	1
1	1	1

x	x’
0	1
1	0

Logic gate symbol:

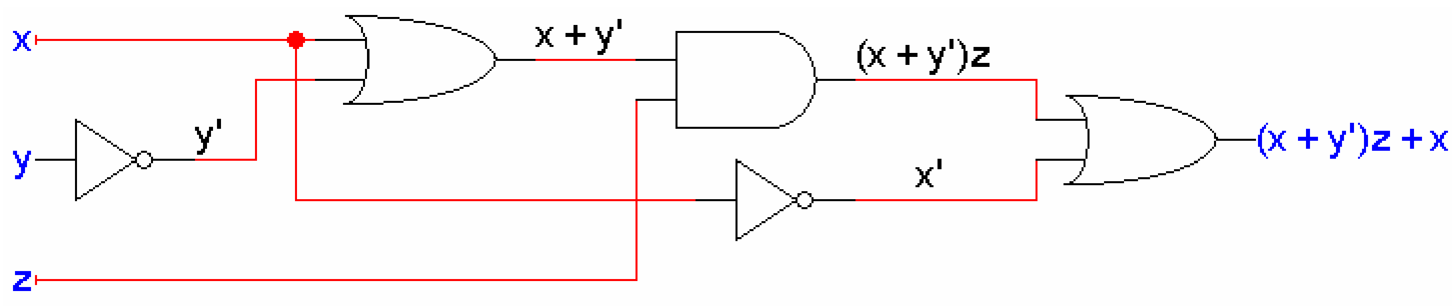






Expressions and circuits

- We can build a circuit for any Boolean expression by connecting primitive logic gates in the correct order.
- The example circuit below accepts input values x, y and z, and produces the output (x + y')z + x'.



- Notice that the order of operations is explicit in the circuit.

Summary

- One of the fundamental concepts of digital circuit design is that deep down inside, computers work with just 0s and 1s.
  - The discrete values 0 and 1 are abstractions for analog voltages.
  - They can represent either arbitrary numbers or Boolean values.
- Boolean logic is especially important in computer architecture.
  - We can build functions from the Boolean values true and false, and the basic operations AND, OR and NOT.
  - Any Boolean function can be implemented by a circuit, built using primitive logic gates to compute products, sums and complements.
- Tomorrow we'll introduce Boolean algebra, which will help us simplify our circuits!