Welcome to CSE370: Introduction to Digital Design

- Course staff
  - Martin Dickey, Sorin Lerner, Wanda Hung
- Course web
  - www.cs.washington.edu/education/courses/370/
- This week
  - What is logic design?
  - What is digital hardware?
  - What will we be doing in this class?
  - Class administration, overview of course web, and logistics
  - Preliminaries: number representation systems
  - Fundamental logical operations

Why are we here?

- Fairly obvious reasons
  - this course is part of the CS/CompE requirements
  - it is the implementation basis for all modern computing devices
    - building large things from small components
- Less obvious reasons
  - provide another model of what a computer is
  - the inherent parallelism in hardware is often our first exposure to parallel computation
  - it offers an interesting counterpoint to software design and is therefore useful in furthering our understanding of computation, in general

What will we learn in CSE370?

- The language of logic design
  - Boolean algebra, logic minimization, state, timing, CAD tools
- The concept of state in digital systems
  - analogous to variables and program counters in software systems
- How to specify/simulate/compile our designs
  - hardware description languages
  - tools to simulate the workings of our designs
  - logic compilers to synthesize the hardware blocks of our designs
  - mapping onto programmable hardware (code generation)
- Contrast with software design
  - both map well-posed problems to physical devices
  - both must be flawless… yet hardware and software failure modes are not all the same
    - Is hardware more reliable than software? If so, why?

Applications of logic design

- Conventional computer design
  - CPUs, busses, peripherals
- Networking and communications
  - phones, modems, routers
- Embedded products
  - in cars, toys, appliances, entertainment devices
- Scientific equipment
  - testing, sensing, reporting
- The world of computing is much much bigger than just PCs!

A quick history lesson

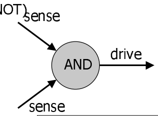
- 1850: George Boole invents Boolean algebra
  - maps logical propositions to symbols
  - permits manipulation of logic statements using mathematics
- 1938: Claude Shannon links Boolean algebra to switches
  - his Masters' thesis
- 1945: John von Neumann develops the first stored program computer
  - its switching elements are vacuum tubes (a big advance from relays)
- 1946: ENIAC . . . The world's first completely electronic computer
  - 18,000 vacuum tubes
  - several hundred multiplications per minute
- 1947: Shockley, Brittain, and Bardeen invent the transistor
  - replaces vacuum tubes
  - enable integration of multiple devices into one package
  - gateway to modern electronics

What is logic design?

- What is design?
  - given a specification of a problem, come up with a way of solving it choosing appropriately from a collection of available techniques and components, while meeting various criteria for size, cost, power, beauty, elegance, etc.
- What is logic design?
  - determining the collection of digital logic components to perform a specified control and/or data manipulation and/or communication function and the interconnections between them
  - which logic components to choose? – there are many implementation technologies (e.g., off-the-shelf fixed-function components, programmable devices, transistors on a chip, etc.)
  - the design may need to be optimized and/or transformed to meet design constraints

What is digital hardware?

- Collection of devices that sense and/or control wires that carry a digital value (i.e., a physical quantity that can be interpreted as a "0" or "1")
  - example: digital logic where voltage < 0.8v is a "0" and > 2.0v is a "1"
  - example: pair of transmission wires where a "0" or "1" is distinguished by which wire has a higher voltage (differential)
  - example: orientation of magnetization signifies a "0" or a "1"
- Primitive digital hardware devices
  - logic computation devices (sense and drive)
    - are two wires both "1" - make another be "1" (AND)
    - is at least one of two wires "1" - make another be "1" (OR)
    - is a wire "1" - then make another be "0" (NOT)
  - memory devices (store)
    - store a value
    - recall a value previously stored



Source: Microsoft Encarta

What is happening now in digital design?

- Big change in the way industry does hardware design over last few years
  - larger and larger designs
  - shorter and shorter time to market
  - cheaper and cheaper products
- Scale
  - pervasive use of computer-aided design tools over hand methods
  - multiple levels of design representation
- Time
  - emphasis on abstract design representations
  - programmable rather than fixed function components
  - automatic synthesis techniques
  - importance of sound design methodologies
- Cost
  - higher levels of integration
  - use of simulation to debug designs

CSE 370: concepts/skills/abilities

- Understanding the basics of logic design (concepts)
- Understanding sound design methodologies (concepts)
- Modern specification methods (concepts)
- Familiarity with a full set of CAD tools (skills)
- Appreciation for the differences and similarities (abilities) in hardware and software design

New ability: *to accomplish the logic design task with the aid of computer-aided design tools and map a problem description into an implementation with programmable logic devices after validation via simulation and understanding of the advantages/disadvantages as compared to a software implementation*

Notes about the course

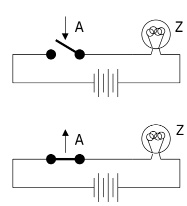
- 3 Lectures, 1 Q. section per week
  - Attendance is expected!
  - Participation is expected!
  - Please read textbook before coming to class
- Homework sets
  - Problems range from mechanical to thought-provoking
  - Good prep for tests
  - OK to work in pairs
- Quizzes and final exams
  - No make-up; can drop one quiz
  - Mostly cover current material
    - some comprehensive questions on Final

Computation: abstract vs. implementation

- Up to now, computation has been a mental exercise (paper, programs)
- This class is about physically implementing computation using physical devices that use voltages to represent logical values
- Basic units of computation are:
  - representation: "0", "1" on a wire; set of wires (e.g., for binary integers)
  - assignment: x = y
  - data operations: x + y – 5
  - control:
    - sequential statements: A; B; C
    - conditionals: if x == 1 then y
    - loops: for (i = 1 ; i == 10, i++)
    - procedures: A; proc(...); B;
- We will study how each of these are implemented in hardware and composed into computational structures

Switches: basic element of physical implementations

- Implementing a simple circuit (arrow shows action if wire changes to "1"):



close switch (if A is "1" or asserted) and turn on light bulb (Z)

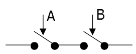
open switch (if A is "0" or unasserted) and turn off light bulb (Z)

$Z \equiv A$

Switches (cont'd)

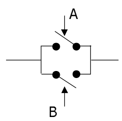
- Compose switches into more complex ones (Boolean functions):

AND



Z ≡ A and B

OR



Z ≡ A or B

Switching networks

- Switch settings
  - determine whether or not a conducting path exists to light the light bulb
- To build larger computations
  - use a light bulb (output of the network) to set other switches (inputs to another network).
- Connect together switching networks
  - to construct larger switching networks, i.e., there is a way to connect outputs of one network to the inputs of the next.

Representation levels of digital designs

- Physical devices (transistors, relays)
- Switches
- Truth tables
- Boolean algebra
- Gates
- Waveforms
- Finite state behavior
- Register-transfer behavior
- Concurrent abstract specifications

scope of CSE 370

Digital vs. analog

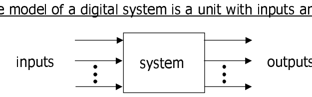
- It is convenient to think of digital systems as having only discrete, digital, input/output values
- In reality, real electronic components exhibit continuous, analog, behavior
- Why do we make this abstraction?
  -
  -
- Why does it work?
  -

Mapping from physical world to binary world

Technology	State 0	State 1
Relay logic	Circuit Open	Circuit Closed
CMOS logic	0.0-1.0 volts	2.0-3.0 volts
Transistor transistor logic (TTL)	0.0-0.8 volts	2.0-5.0 volts
Fiber Optics	Light off	Light on
Dynamic RAM	Discharged capacitor	Charged capacitor
Nonvolatile memory (erasable)	Trapped electrons	No trapped electrons
Programmable ROM	Fuse blown	Fuse intact
Bubble memory	No magnetic bubble	Bubble present
Magnetic disk	No flux reversal	Flux reversal
Compact disc	No pit	Pit

Combinational vs. sequential digital circuits

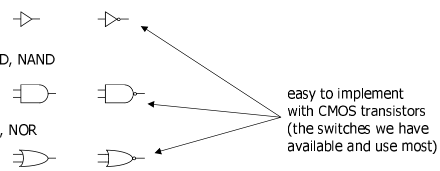
- A simple model of a digital system is a unit with inputs and outputs:



- Combinational means "memory-less"
  - a digital circuit is combinational if its output values only depend on its input values

Combinational logic symbols

- Common combinational logic systems have standard symbols called logic gates
  - Buffer, NOT
  - AND, NAND
  - OR, NOR



Sequential logic

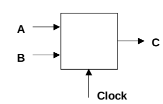
- Sequential systems
  - exhibit behaviors (output values) that depend not only on the current input values, but also on previous input values
- In reality, all real circuits are sequential
  - because the outputs do not change instantaneously after an input change
  - why not, and why is it then sequential?
- A fundamental abstraction of digital design is to reason (mostly) about steady-state behaviors
  - look at the outputs only after sufficient time has elapsed for the system to make its required changes and settle down

Synchronous sequential digital systems

- Outputs of a combinational circuit depend only on current inputs
  - after sufficient time has elapsed
- Sequential circuits have memory
  - even after waiting for the transient activity to finish
- The steady-state abstraction is so useful that most designers use a form of it when constructing sequential circuits:
  - the memory of a system is represented as its state
  - changes in system state are only allowed to occur at specific times controlled by an external periodic clock
  - the clock period is the time that elapses between state changes it must be sufficiently long so that the system reaches a steady-state before the next state change at the end of the period

Example of combinational and sequential logic

- Combinational:
  - input A, B
  - wait for clock edge
  - observe C
  - wait for another clock edge
  - observe C again: will stay the same
- Sequential:
  - input A, B
  - wait for clock edge
  - observe C
  - wait for another clock edge
  - observe C again: may be different



Abstractions

- Some we've seen already
  - digital interpretation of analog values
  - transistors as switches
  - switches as logic gates
  - use of a clock to realize a synchronous sequential circuit
- Some others we will see
  - truth tables and Boolean algebra to represent combinational logic
  - encoding of signals with more than two logical values into binary form
  - state diagrams to represent sequential logic
  - hardware description languages to represent digital logic
  - waveforms to represent temporal behavior

An example

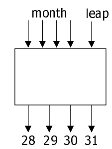
- Calendar subsystem: number of days in a month (to control watch display)
  - used in controlling the display of a wrist-watch LCD screen
  - inputs: month, leap year flag
  - outputs: number of days

Implementation in software

```
integer number_of_days ( month, leap_year_flag) {
    switch (month) {
        case 1: return (31);
        case 2: if (leap_year_flag == 1) then return (29)
          else return (28);
        case 3: return (31);
        ...
        case 12: return (31);
        default: return (0);
    }
}
```


Implementation as a combinational digital system

- Encoding:
  - how many bits for each input/output?
  - binary number for month
  - four wires for 28, 29, 30, and 31
- Behavior:
  - combinational
  - truth table specification



month	leap	28	29	30	31
0001	–	0	0	0	1
0010	0	1	0	0	0
0010	1	0	1	0	0
0011	–	0	0	0	1
0100	–	0	0	1	0
...					
1100	–	0	0	0	1
1101	–	–	–	–	–
111–	–	–	–	–	–
0000	–	–	–	–	–

Combinational example (cont'd)

- Truth-table to logic to switches to gates
  - 28 = 1 when month=0010 and leap=0
  - 28 = m1'•m2'•m3•m4'•leap'
  - 31 = 1 when month=0001 or month=0011 or ... month=1100
  - 31 = (m1'•m2'•m3'•m4) + (m1'•m2'•m3•m4) + ... (m1•m2•m3'•m4')
  - 31 = can we simplify more?

symbol for or — symbol for and — symbol for not

month	leap	28	29	30	31
0001	–	0	0	0	1
0010	0	1	0	0	0
0010	1	0	1	0	0
0011	–	0	0	0	1
0100	–	0	0	1	0
...					
1100	–	0	0	0	1
1101	–	–	–	–	–
111–	–	–	–	–	–
0000	–	–	–	–	–

Another example

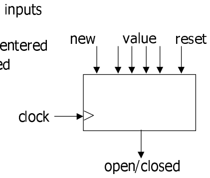
- Door combination lock:
  - punch in 3 values in sequence and the door opens; if there is an error the lock must be reset; once the door opens the lock must be reset
  - inputs: sequence of input values, reset
  - outputs: door open/close
  - memory: must remember combination or always have it available as an input

Implementation in software

```
integer combination_lock ( ) {
    integer v1, v2, v3;
    integer error = 0;
    static integer c[3] = 3, 4, 2;

    while (!new_value( ));
    v1 = read_value( );
    if (v1 != c[1]) then error = 1;

    while (!new_value( ));
    v2 = read_value( );
    if (v2 != c[2]) then error = 1;

    while (!new_value( ));
    v3 = read_value( );
    if (v2 != c[3]) then error = 1;

    if (error == 1) then return(0); else return (1);
}
```

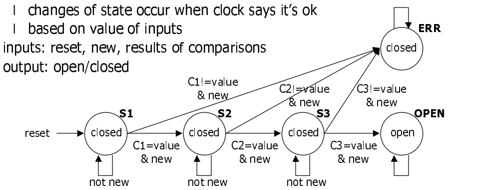

Implementation as a sequential digital system

- Encoding:
  - how many bits per input value?
  - how many values in sequence?
  - how do we know a new input value is entered?
  - how do we represent the states of the system?
- Behavior:
  - clock wire tells us when it's ok to look at inputs (i.e., they have settled after change)
  - sequential: sequence of values must be entered
  - sequential: remember if an error occurred
  - finite-state specification



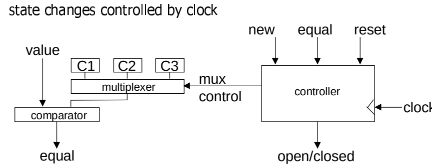
Sequential example (cont'd): abstract control

- Finite-state diagram
  - states: 5 states
    - represent point in execution of machine
    - each state has outputs
  - transitions: 6 from state to state, 5 self transitions, 1 global
    - changes of state occur when clock says it's ok
    - based on value of inputs
  - inputs: reset, new, results of comparisons
  - output: open/closed



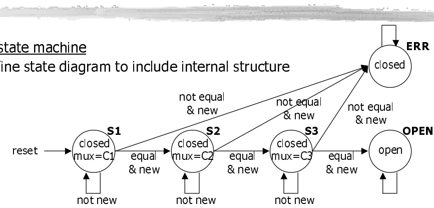
Sequential example (cont'd): data-path vs. control

- Internal structure
  - data-path
    - storage for combination
    - comparators
  - control
    - finite-state machine controller
    - control for data-path
    - state changes controlled by clock



Sequential example (cont'd): finite-state machine

- Finite-state machine
  - refine state diagram to include internal structure



  - generate state table (much like a truth-table)

reset	new	equal	state	next state	mux	open/closed
1	–	–	–	S1	C1	closed
0	0	–	S1	S1	C1	closed
0	1	0	S1	ERR	–	closed
0	1	1	S1	S2	C2	closed
...						
0	1	1	S3	OPEN	–	open
...						

Sequential example (cont'd): encoding

- Encode state table
  - state can be: S1, S2, S3, OPEN, or ERR
    - needs at least 3 bits to encode: 000, 001, 010, 011, 100
    - and as many as 5: 00001, 00010, 00100, 01000, 10000
    - choose 4 bits: 0001, 0010, 0100, 1000, 0000
  - output mux can be: C1, C2, or C3
    - needs 2 to 3 bits to encode
    - choose 3 bits: 001, 010, 100
  - output open/closed can be: open or closed
    - needs 1 or 2 bits to encode
    - choose 1 bits: 1, 0

reset	new	equal	state	next state	mux	open/closed
1	–	–	–	0001	001	0
0	0	–	0001	0001	001	0
0	1	0	0001	0000	–	0
0	1	1	0001	0010	010	0
...						
0	1	1	0100	1000	–	1
...						

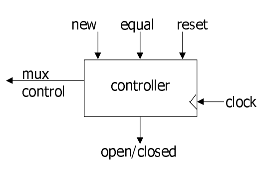
good choice of encoding!

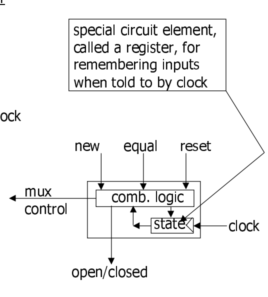
mux is identical to last 3 bits of state

open/closed is identical to first bit of state

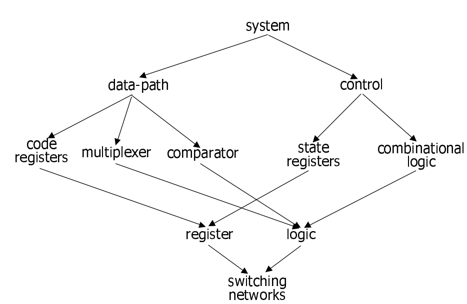
Sequential example (cont'd): controller implementation

- Implementation of the controller





Design hierarchy



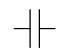
Summary

- That was what the entire course is about
  - converting solutions to problems into combinational and sequential networks effectively organizing the design hierarchically
  - doing so with a modern set of design tools that lets us handle large designs effectively
  - taking advantage of optimization opportunities
- Now lets do it again
  - this time we'll take nine weeks

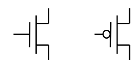
The basics→ electronics

- Resistor
  - Ohm's law→ V=IR
    - V≡voltage, I≡current, R≡resistance

- Capacitor
  - I=C(dV/dt)
    - C≡capacitance
  - No DC current path
  - Voltage cannot change instantaneously



- MOS Transistors
  - Used as switches
  - Pass binary voltages



The basics→ binary numbers

- Base conversion (binary, octal, decimal, hexadecimal)
  - Positional number system
    - $101_2=5_{10}$
    - $101_8=65_{10}$
    - $101_{16}=257_{10}$
  - Conversion between binary/octal/hex
    - Binary: 10011110001
    - Octal: 10 | 011 | 110 | 001=2361_8
    - Hex: 100 | 1111 | 0001=$4F1_{16}$
- Addition and subtraction are trivial, but worth practicing
  - See Katz, appendix A

The basics→ base conversion

- Conversion from decimal to binary/octal/hex

Binary

	Quotient	Remainder
56÷2=	28	0
28÷2=	14	0
14÷2=	7	0
7÷2=	3	1
3÷2=	1	1
1÷2=	0	1

Octal

	Quotient	Remainder
56÷8=	7	0
7÷8=	0	7

$56_{10}=111000_2$

$56_{10}=70_8$

- Why does this work?
  - $N=56_{10}=111000_2$
  - Q=N/2=56/2=111000/2=11100 remainder 0
- Each successive divide liberates an LSB

Number systems

- How do we write negative binary numbers?
- Historically: Three approaches
  - Sign and magnitude
  - Ones complement
  - Twos complement
- Twos complement makes addition and subtraction easy
  - Used almost universally in present-day systems

Approach 1: Sign and magnitude

- The most-significant bit (msb) is the sign digit
  - 0 ≡ positive
  - 1 ≡ negative
- The remaining bits are the number's magnitude
- Problem 1: Two representations for zero
  - 0 = 0000 *and also* –0 = 1000
- Problem 2: Arithmetic is cumbersome

Add		Subtract			Compare and subtract		
4	0100	4	0100	0100	– 4	1100	1100
+ 3	+ 0011	– 3	+ 1011	– 0011	+ 3	+ 0011	– 0011
= 7	= 0111	= 1	≠ 1111	= 0001	– 1	≠ 1111	= 1001

Approach 2: Ones complement

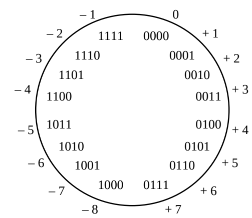
- Negative number: Bitwise complement of positive number
  - 0011 ≡ 3_{10}
  - 1100 ≡ -3_{10}
- Solves the arithmetic problem

Add		Invert, add, add carry		Invert and add	
4	0100	4	0100	− 4	1011
+ 3	+ 0011	− 3	+ 1100	+ 3	+ 0011
= 7	= 0111	= 1	1 0000	− 1	1110
		add carry:	+ 1		
			= 0001		

- Remaining problem: Two representations for zero
  - 0 = 0000 *and also* −0 = 1111

Approach 3: Twos complement

- Negative number: Bitwise complement plus one
  - 0011 ≡ 3_{10}
  - 1101 ≡ -3_{10}
- Number wheel



- Only one zero!
- msb is the sign digit
  - 0 ≡ positive
  - 1 ≡ negative

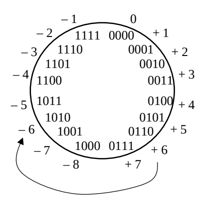
Twos complement (con't)

- Complementing a complement restores the original number
- Arithmetic is easy
  - We ignore the carry
    - Same as a full rotation around the wheel
  - Subtraction = negation and addition
    - Easy to implement in hardware

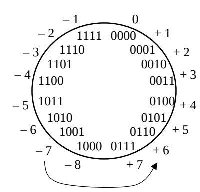
Add		Invert and add		Invert and add	
4	0100	4	0100	− 4	1100
+ 3	+ 0011	− 3	+ 1101	+ 3	+ 0011
= 7	= 0111	= 1	1 0001	− 1	1111
		drop carry	= 0001		

Overflow

- Conditions: Sign bit changes
  - Summing two positive numbers gives a negative result
  - Summing two negative numbers gives a positive result



6 + 4 ⇒ –6



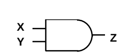
–7 – 3 ⇒ +6

Next subject: Combinational logic

- Logic functions and truth tables
  - AND, OR, Buffer, NAND, NOR, NOT, XOR, XNOR
- Gate logic
  - Networks of Boolean functions
- Axioms and theorems of Boolean algebra
- Canonical forms
  - Sum of products and product of sums
- Simplification
  - Boolean cubes and Karnaugh maps
  - Two-level simplification

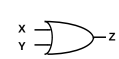
Logic functions and truth tables

- AND X • Y XY



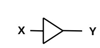
X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

- OR X + Y



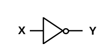
X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

- Buffer X



X	Y
0	0
1	1

- NOT X' $\overline{X}$

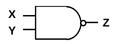


X	Y
0	1
1	0

Logic functions and truth tables (con't)

- NAND

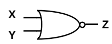
$\overline{X \bullet Y}$ $\overline{XY}$



X	Y	Z
0	0	1
0	1	1
1	0	1
1	1	0

- NOR

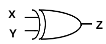
$\overline{X + Y}$



X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	0

- XOR

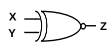
$X \oplus Y$



X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

- XNOR

$\overline{X \oplus Y}$



X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	1

Some notation

- Priorities: $\overline{A} \bullet B + C = ((\overline{A}) \bullet B) + C$
- Variables are called literals
- Definitions
  - *Schematic*: a drawing of interconnected gates
  - *Net*: wires at the same voltage (electrically connected)
  - *Netlist*: a listing of all the I/O (gate and page) in a schematic
  - *Fan-in*: the # of inputs to a gate
  - *Fan-out*: the # of loads the gate drives

Minimal set

- We can implement all logic functions from NOT, NOR, and NAND
  - Example: (X and Y) = not (X nand Y)
- In fact, we can do it with only NOR or only NAND
  - NOT is just NAND or NOR with both inputs tied together

X	Y	X nor Y
0	0	1
1	1	0

X	Y	X nand Y
0	0	1
1	1	0

  - NAND and NOR are duals: We can implement one from the other
    - X nand Y = not ((not X) nor (not Y))
    - X nor Y = not ((not X) nand (not Y))