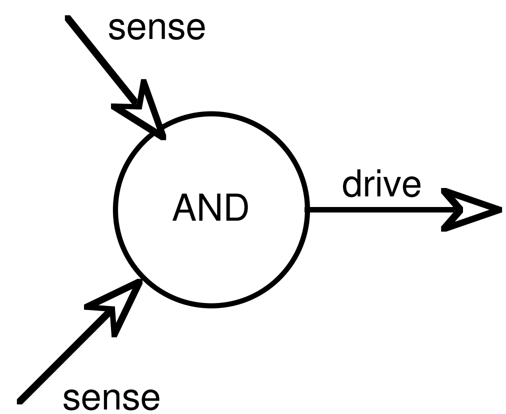
- Welcome to CSE 370
- Course staff: Gaetano Borriello and Corey Anderson
- What is logic design?
- What is digital hardware?
- What will we be doing in CSE 370?
- Class administration and logistics

- **What is design?**
  - given a specification of a problem, come up with a way of solving it using choosing appropriately from a collection of available components
  - while meeting some criteria for size, cost, robustness, 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 < .5v is a "0" and > 2.4v is a "1"
  – example: pair of transmission wires where a "0" or "1" is distinguished by which wire has a higher voltage (differential)

- **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



Why do we study logic design?

- **Obvious reasons**
  - this course is part of the CS/CompE requirements
  - It is the implementation basis for all modern computing devices

- **More important reasons**
  - The inherent parallelism in hardware if 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

How dynamic is this field?

- **Didn't Boole start doing this stuff in a previous century?**
  - Boole, George (1815-64), British mathematician and logician, professor of mathematics at Queen's College (now University College) in Cork, Ireland. Noted for the development of the algebraic system known as Boolean algebra, in which logical propositions are denoted by symbols and subjected to abstract mathematical operations, corresponding to the laws of logic. Boolean algebra is of prime importance to the study of pure mathematics and the design of modern computers. Among his works is *An Investigation of the Laws of Thought* (1854).

- **Didn't most new design developments take place earlier this century?**
  - The rapidly advancing field of electronics led to construction of the first all-electronic computer in 1946 at the University of Pennsylvania by the American engineer John Presper Eckert, Jr. (1919-) and the American physicist John William Mauchly (1907-80). ENIAC (Electronic Numerical Integrator And Computer) contained 18,000 vacuum tubes and had a speed of several hundred multiplications per minute. Its program was wired into the processor and had to be manually altered. In 1945, John von Neumann had developed the first stored program computer.

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

What is CSE 370 about?

- **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 for implementation with programmable logic devices after validation via simulation and understand the advantages/disadvantages as compared to a software implementation

Computation: Abstract vs. Implementation

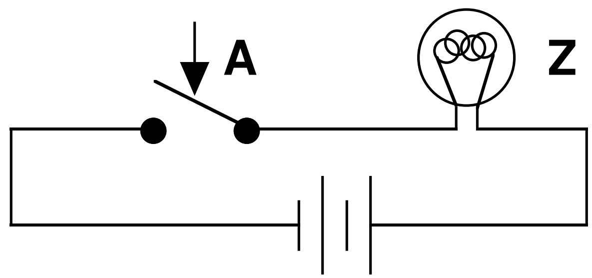
- **Up to now, computation has been a mental exercise (paper, programs)**
- **This class is about physically implementing computation using physical devices that use voltages to represent logical values**
- **Basic units of computation are:**

– 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 as the basic element of physical implementations

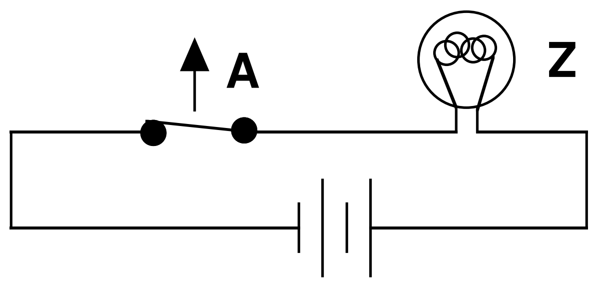
- **Implementing a simple circuit (arrow show action if wire changes to "1"):**



close switch (if x is "1" or asserted)

and turn on light bulb (y)

Z ≡ A



open switch (if x is "1" or asserted)

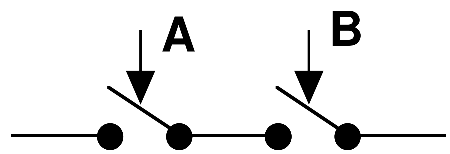
and turn on light bulb (y)

Z ≡ (not A)

single-pole-single-throw (SPST) switches are very much like transistors of integrated circuits – either open or closed

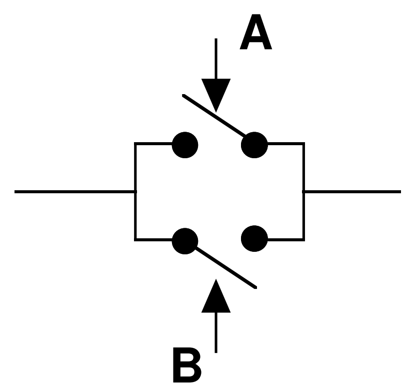
- **Compose switches into more complex ones (Boolean functions):**

AND



$Z \equiv A$ and B

OR



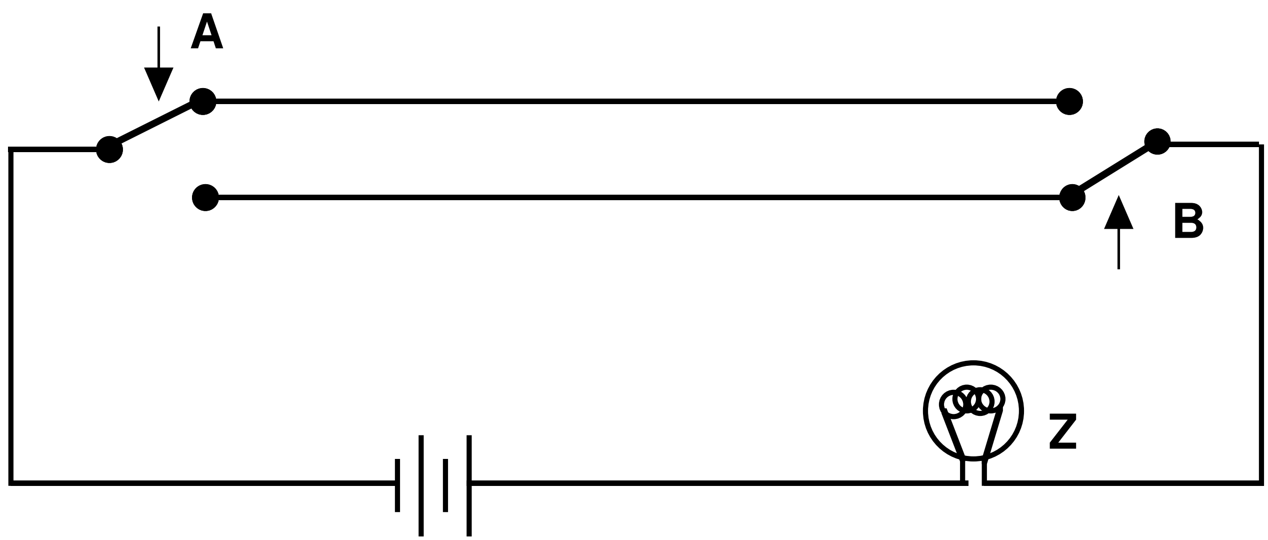
$Z \equiv A$ or B

Switches as the basic element of physical implementations (cont'd)

- A more interesting Boolean function: EXCLUSIVE OR (XOR)

$$Z \equiv A \text{ xor } B$$

$$Z \equiv (A \text{ and } (\text{not } B)) \text{ or } (B \text{ and } (\text{not } A))$$



single-pole-double-throw (SPDT) switches are very convenient but
can not be implemented directly with transistors of integrated circuits

Switching networks

- Switch settings determine whether or not a conducting path exists to light the light bulb

- Inorder to use these networks to build larger computations, there must be some way to use light bulb (output of the network) to set other switches (inputs to another network).

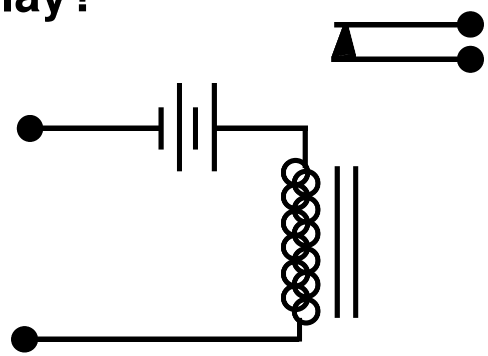
- Now we can connect together switching networks to construct larger switching networks, i.e., there is a way to connect outputs of one network to the inputs of the next.

Relay networks

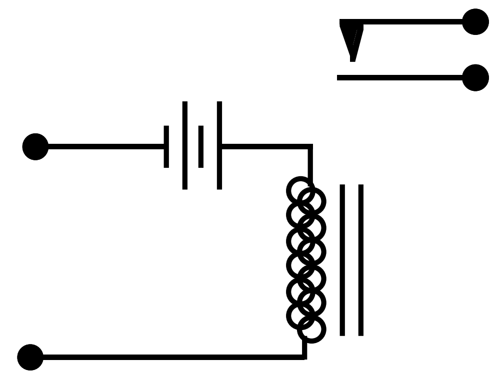
- A simple way to convert between conducting paths and switch settings is to use (electro-mechanical) relays.

- What is a relay?

conducting
path composed
of switches
closes circuit



current flowing through coil
magnetizes core and causes normally
closed (nc) contact to be pulled open



current flowing through coil
magnetizes core and causes normally
open (no) contact to be pulled closed

when no current flows, a spring in the contacts
returns them to their normal positions

- What determines the switching speed of a relay network?

CMOS transistor networks

- **Relays aren't used much anymore**
  - some traffic light controllers are still electro-mechanical

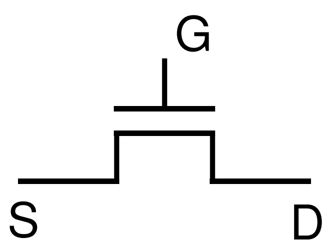
- **Modern digital systems are designed in CMOS technology (Complementary MOS)**
  - MOS stands for Metal-Oxide on Semiconductor
  - it is called complementary because there are both normally-open and normally-closed switches

- **MOS transistors act as voltage-controlled switches**
  - similar, though easier to work with than relays.

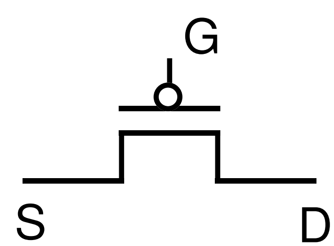
CMOS transistors

- **MOS transistors have three terminals: drain, gate, and source**
  - they act as switches in the following way:

    if the voltage on the gate terminal is (some amount)
        higher/lower than the source terminal
    then a conducting path will be established between
        the drain and source terminals

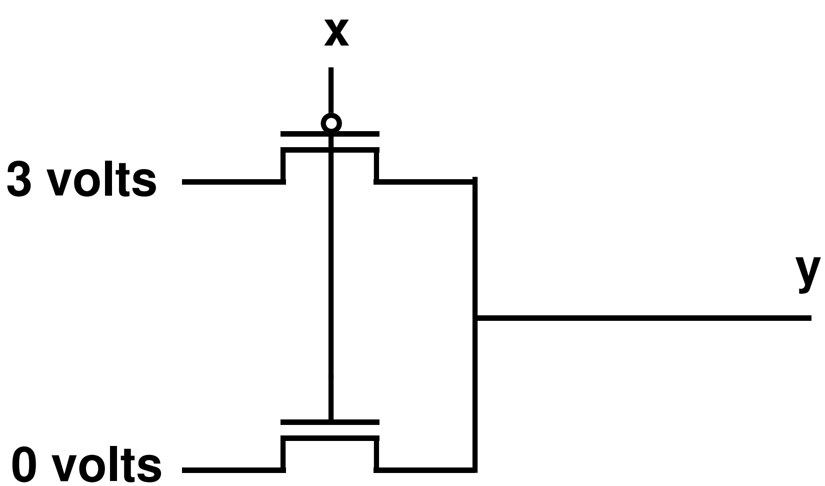


n-channel
normally-open
closes when:
voltage(G) > voltage (S) + ε



p-channel
normally-closed
opens when:
voltage(G) < voltage (S) – ε

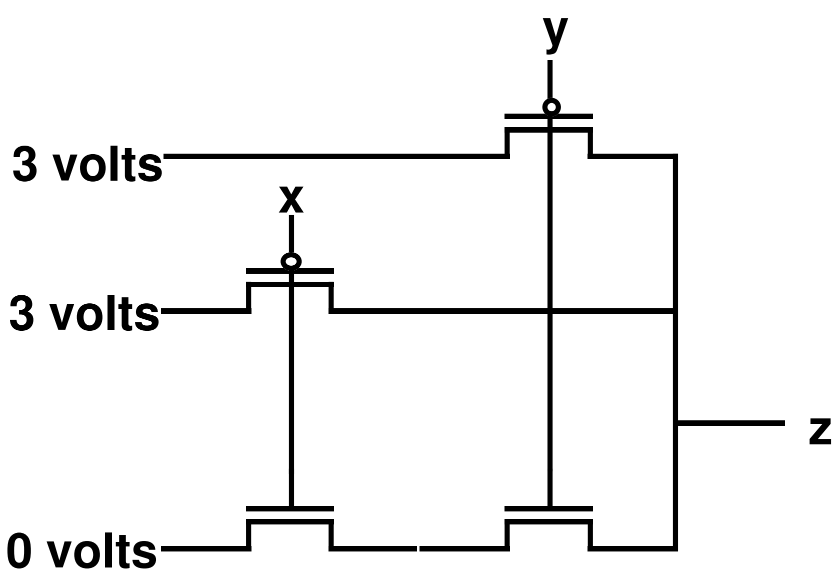
CMOS networks



what is the
relationship
between x and y?

x	y
0 volts	
3 volts	

Two input networks



what is the relationship between x, y and z?

x	y	z
0 volts	0 volts	
0 volts	3 volts	
3 volts	0 volts	
3 volts	3 volts	

Speed of CMOS networks

- **What influences the speed of CMOS networks?**

Representations 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

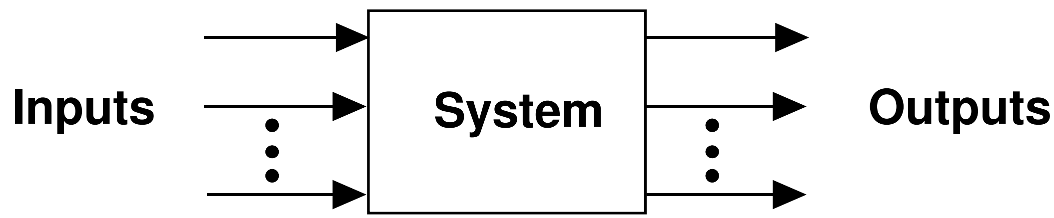
- It is convenient to think of digital systems as having only discrete, digital, input/output values
- In reality, real electronic components exhibit continuous, analog, behavior
- Why do we make this abstraction?
  - –
  - –
- Why does it work?
  - –
  - –

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	2.5-5.0 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:**



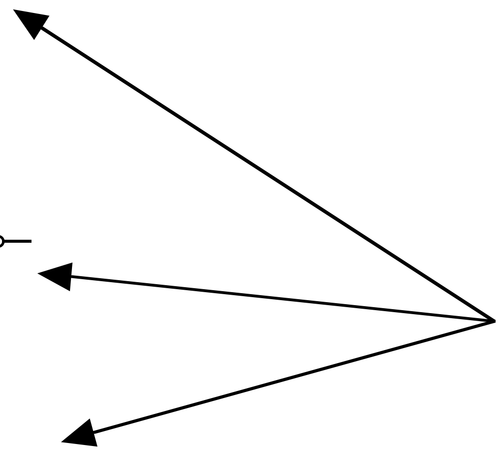
- **Combinational means "memory-less"**
  – a digital circuit is combinational if its output values only depend on its input values

- **Common combinational logic systems have standard symbols called logic gates**

  – Buffer, NOT

  – AND, NAND

  – OR, NOR



easy to implement
with CMOS transistors
(the switches we have
available and use most)

Combinational vs. sequential digital circuits (cont'd)

- **Sequential systems exhibit behavoirs (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 instananeously after an input change**
  - why, and why is it then sequential?
- **A fundamental abstraction of digital design is to reason (mostly) about steady-state behaviors**
  - what are the outputs after sufficient time has elapsed for the system to make its required changes and settle down

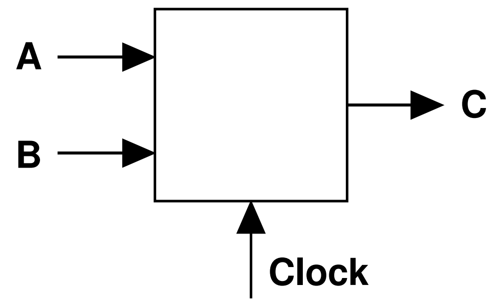
Synchronous sequential digital systems

- **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 <u>memory</u> of a system is represented as its <u>state</u>
  - changes in system state are only allowed to occur at specific times controlled by an external periodic <u>clock</u>
  - the <u>clock period</u> 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

- **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
  - State diagrams to represent sequential logic
  - Hardware description languages to represent 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

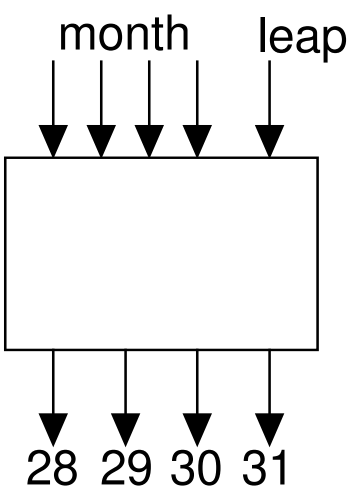
  - inputs: month, leap year flag
  - outputs: number of days

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)

symbol for or

symbol for and

symbol for not

- **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?

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

Implementation in software

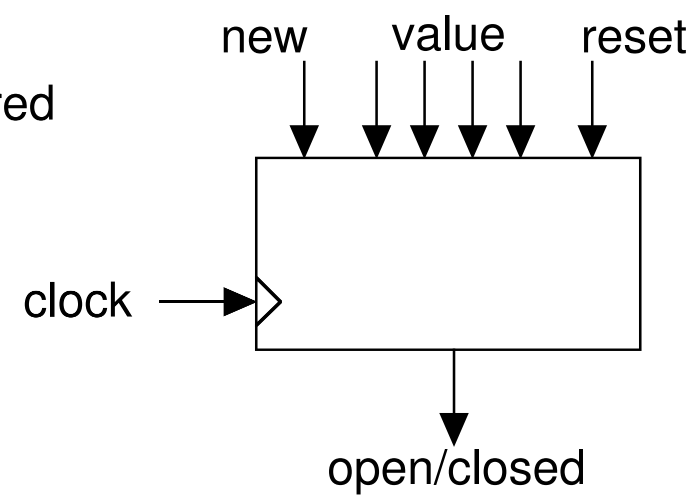
```
integer combination_lock ( ) {
    integer v1, v2, v3;
    integer error = 0;
    static c[3] = 3, 4, 2;

    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?

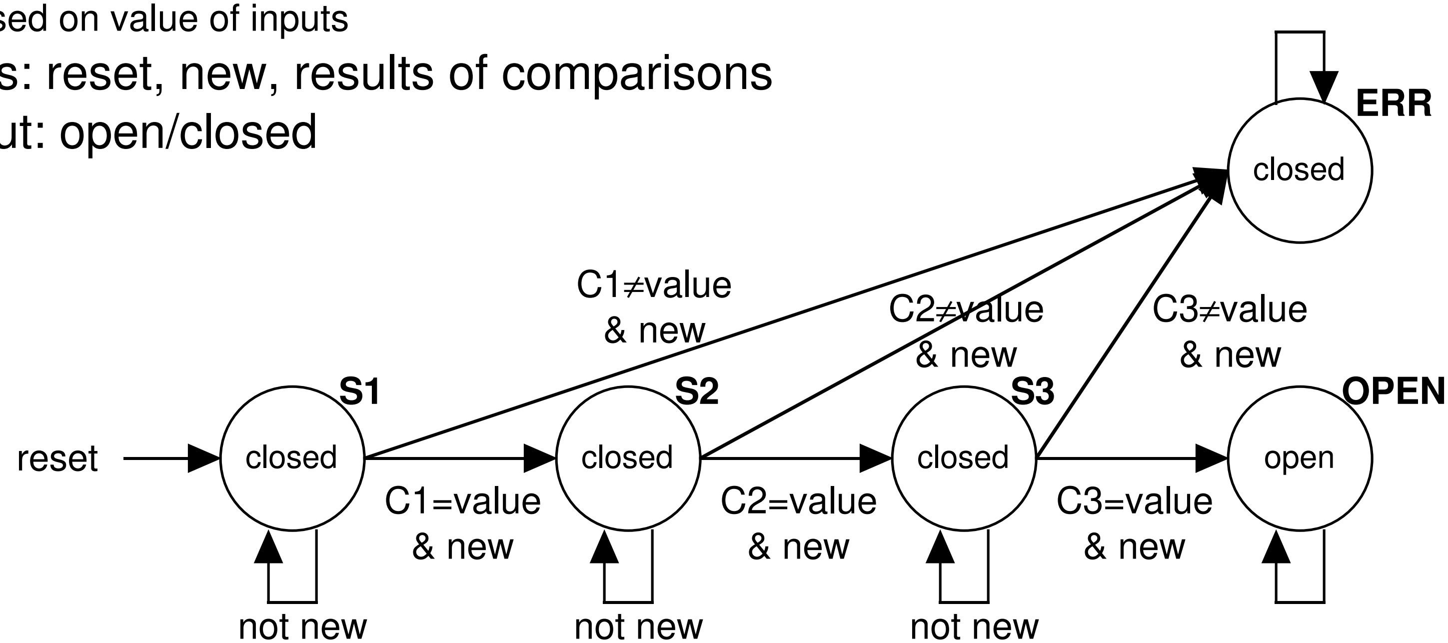
- **Behavior:**
  - clock wire tells us when its ok to look at inputs (i.e., they have settled after change)

  - sequential: sequence of values must be entered
  - sequential: remember if an error occured
  - finite-state specification



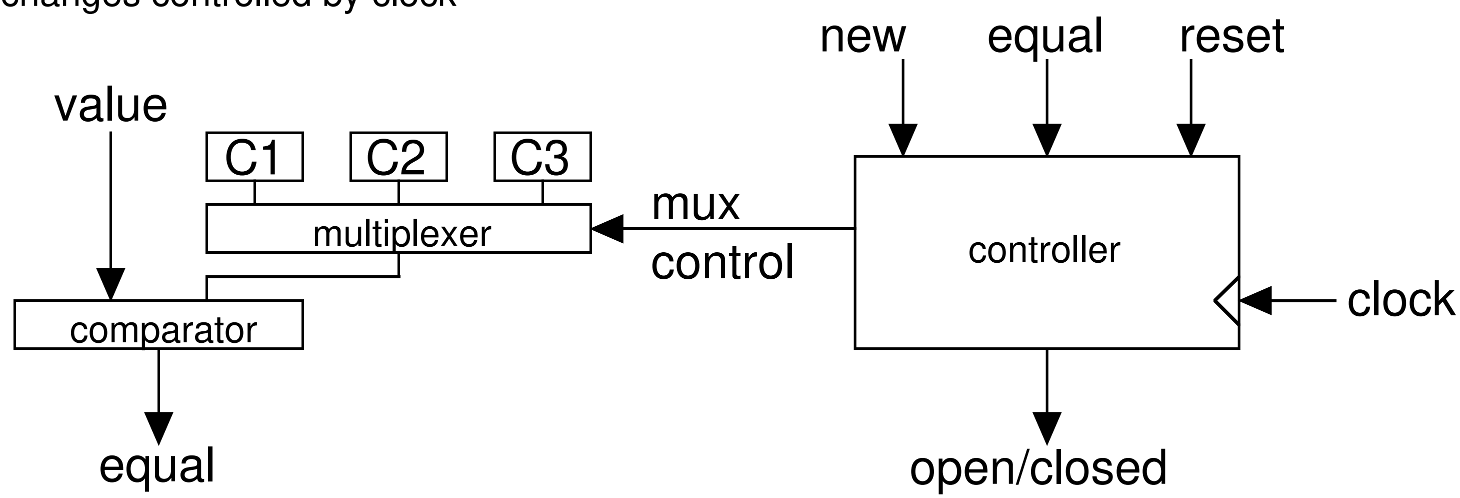
Sequential example (cont'd): abstract control

- **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 its ok
    - based on value of inputs
  - Inputs: reset, new, results of comparisons
  - Output: open/closed



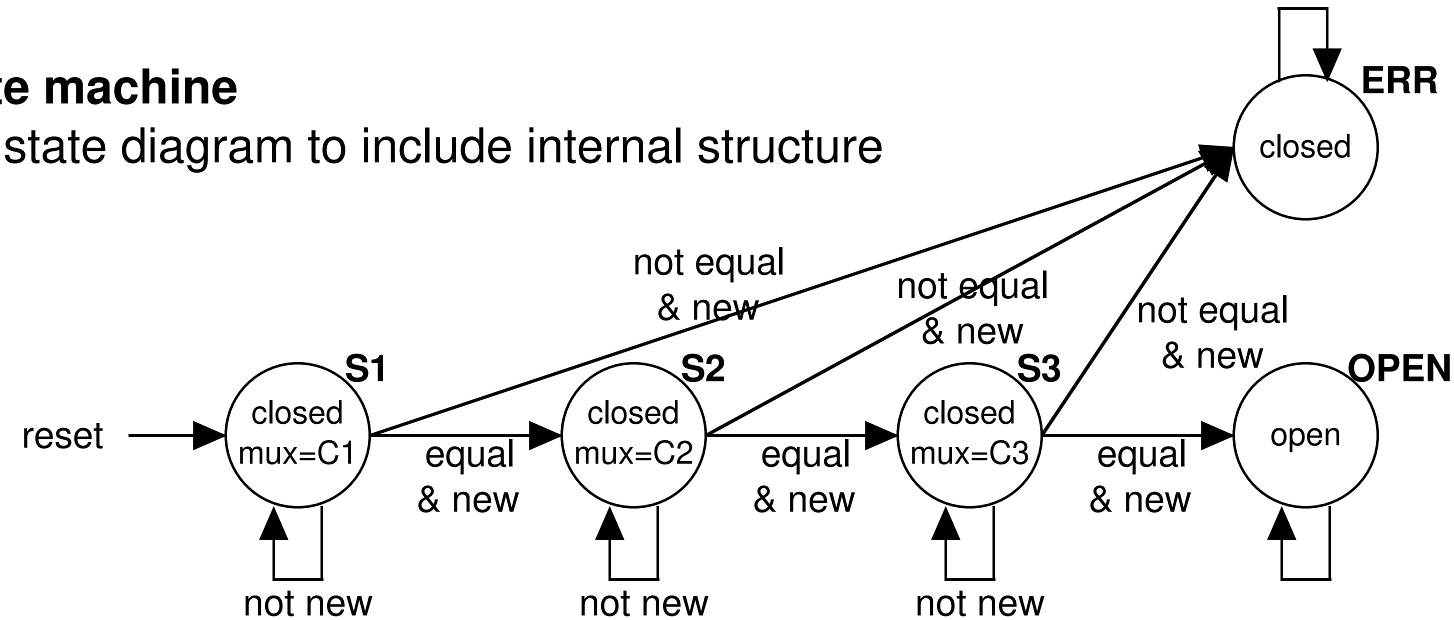
Sequential example (cont'd): data-path vs. control

- **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): FSM

- **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: <u>0001, 0010, 0100, 1000, 0000</u>
  - output mux can be: C1, C2, or C3
    - needs 2 to 3 bits to encode
    - choose 3 bits: <u>001, 010, 100</u>
  - output open/closed can be: open or closed
    - needs 1 or 2 bits to encode
    - choose 1 bits: <u>1, 0</u>

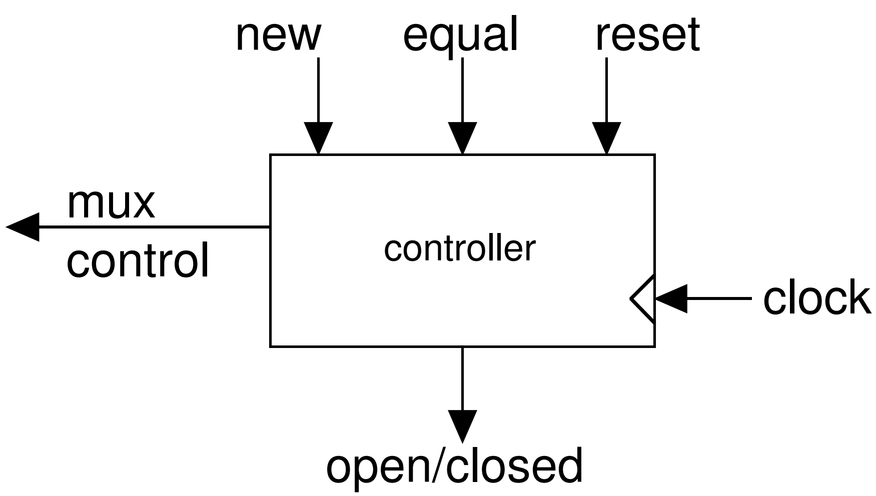
reset	new	equal	state	next state	mux	open/closed
1	–	–	–	0001	001	0
0	0	–	0001	0001	001	0
0	1	0	0001	0000	–	0
0	1	1	0001	0010	010	0
...						
0	1	1	0100	1000	–	1
...						

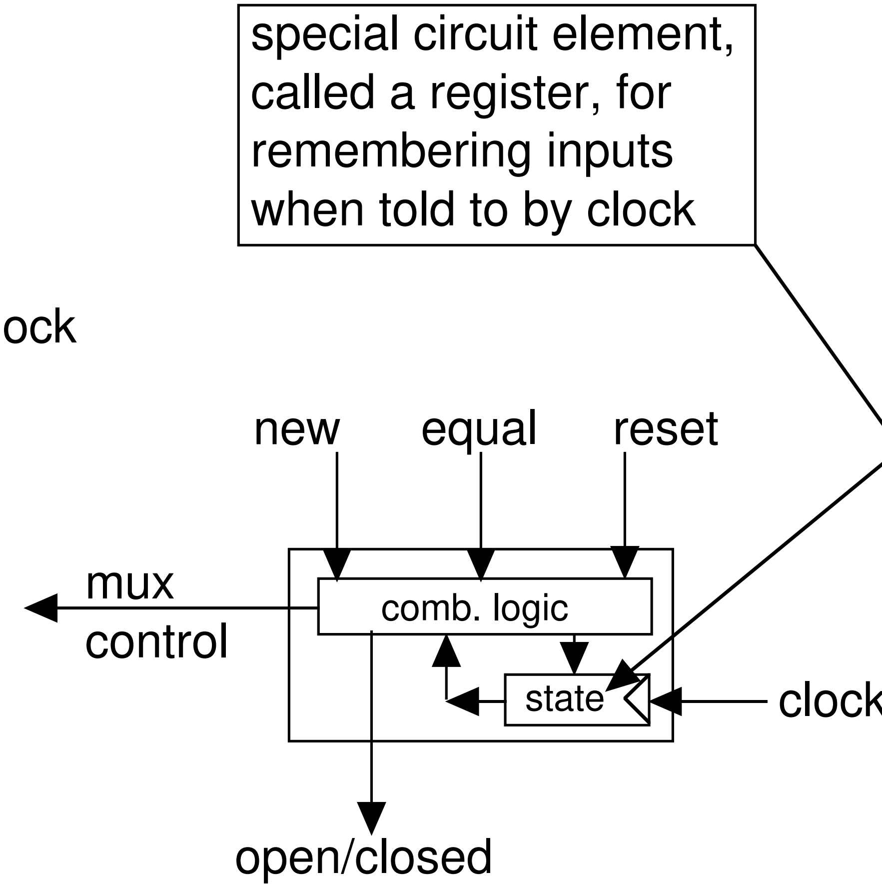
good choice of encoding!

mux is identical to last 3 bits of state

open/closed is identical to first bit of state

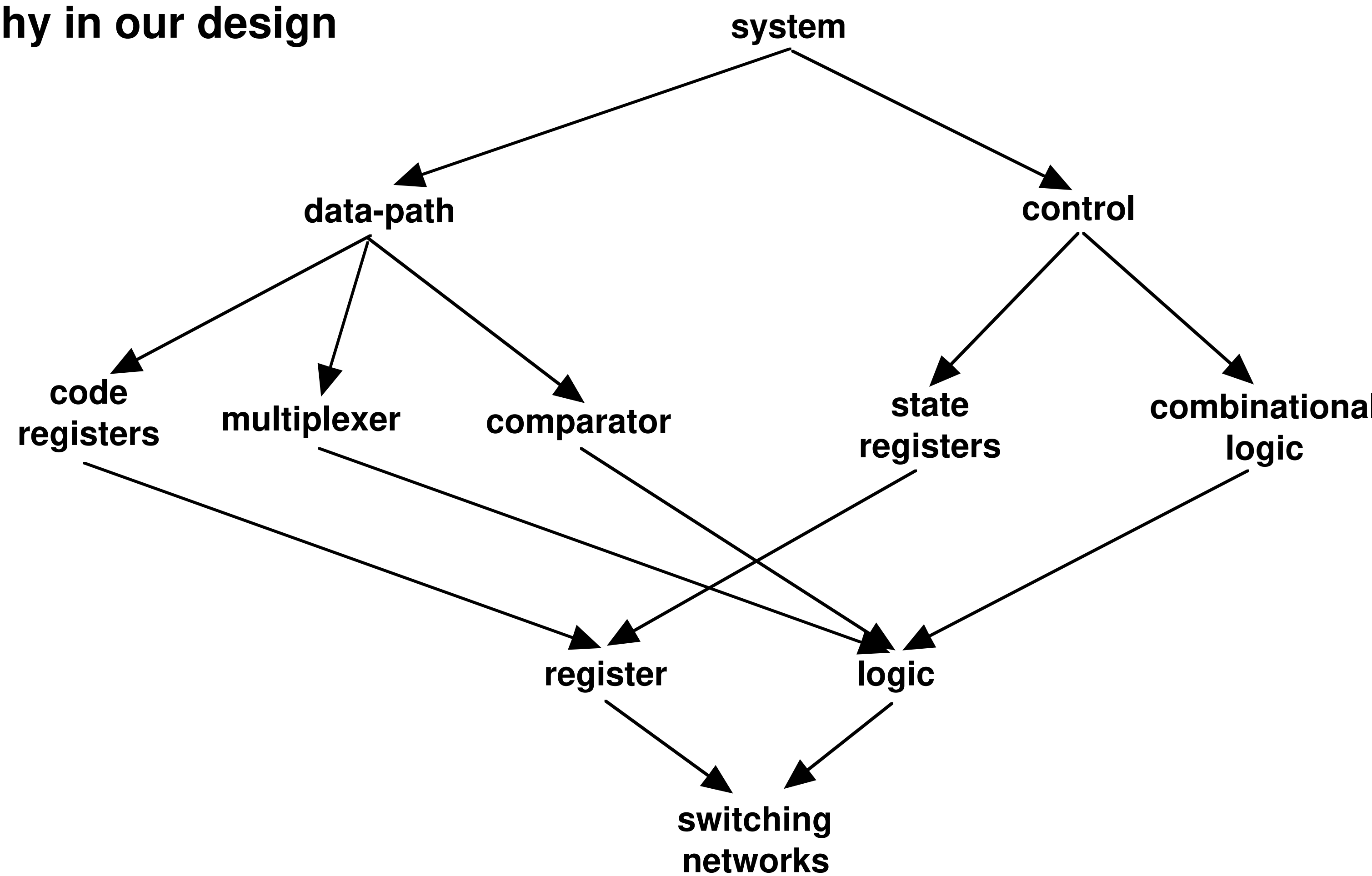
- **Implementation of the controller**





Sequential example (cont'd): hierarchy

- **Hierarchy in our design**



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