CH-1: Introduction

Contemporary Logic Design

YONSEI UNIVERSITY
Fall 2016

Why Logic Design?

- Main reasons
  - core part of the CS/CE requirements
  - it is the implementation basis for all modern computing devices
    - from small components to large modules (systems)
    - provide a model of how a computer works
    - Basis for computer systems & SOC (system on chip) embedded computing applications for everywhere
  - an interesting counterpoint to software design and therefore useful in expanding our understanding of computation and computing systems

Where can Logic Design be used?

- Major areas
  - All digital devices for control, measurement, & entertainment
    - Embedded systems for *pervasive computing engines*
    - Computing systems for personal/*supercomputing engines*

Applications of Logic Design

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

What to Cover in This Course?

- Basic issues of logic design
  - *Boolean algebra*, *logic minimization*, *state rep, timing, CAD tools*
- <u>Concept of states</u> in digital systems
  - analogous to *variables and program counters* in software systems
- Design principle
  - Design method for combinational circuits
  - Design method for sequential circuits
- How to <u>specify/simulate/compile/realize</u> our designs
  - hardware description languages (Lab)
  - tools to simulate the workings of our designs (Lab)
  - logic compilers to synthesize the hardware blocks of our designs
  - mapping techniques onto programmable hardware

Quick History on LD

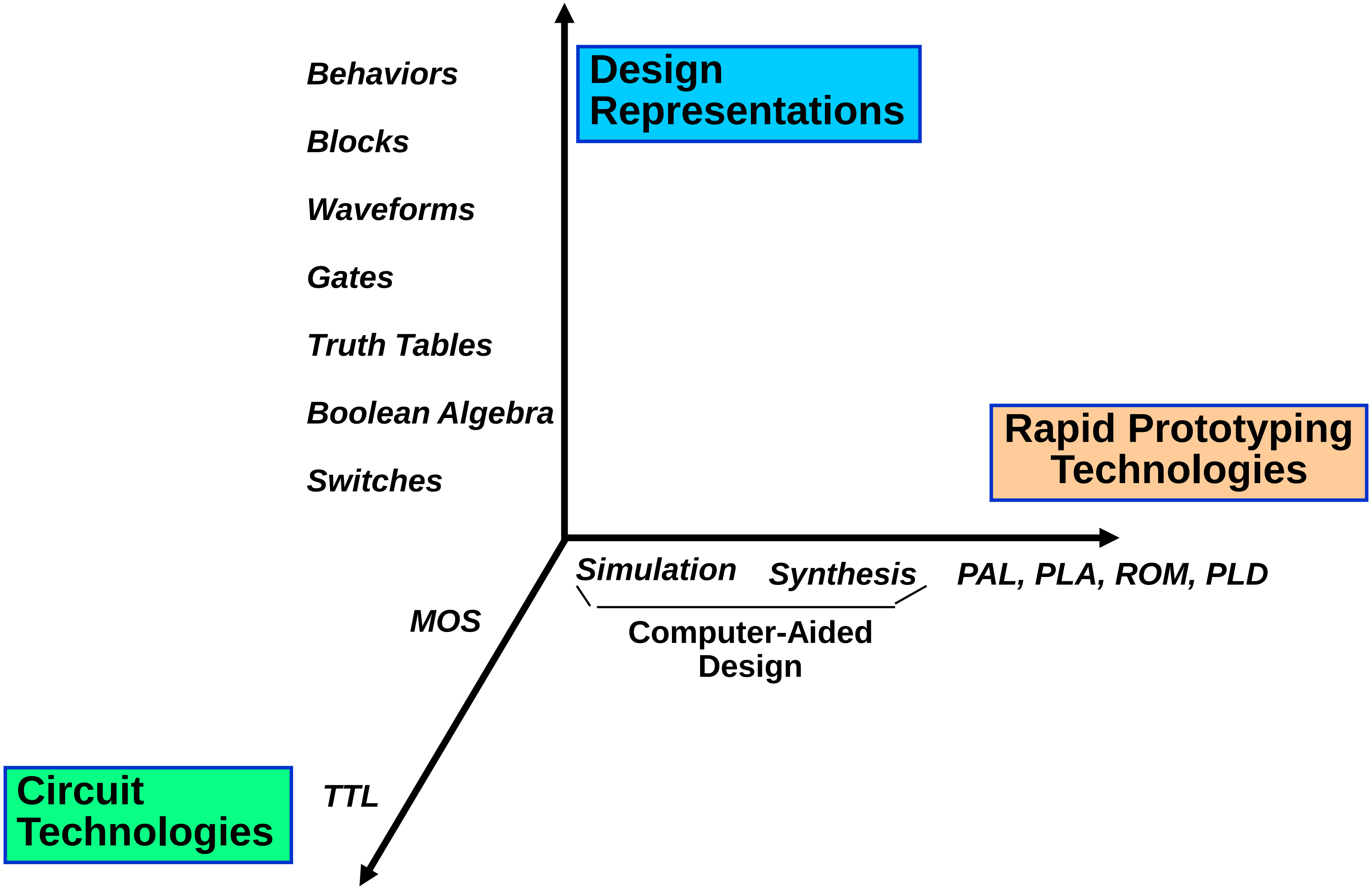
- 1850: George Boole invented Boolean algebra
  - Map logical propositions to symbols
  - Permit manipulation of logic statements using mathematics
- 1938: Claude Shannon links Boolean algebra to switches
  - his Masters' thesis
- 1945: John von Neumann develops the first stored program computer
  - 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 invented the transistor
  - replaced vacuum tubes
  - allowed integration of multiple devices into one package
  - gateway to modern electronics

What is Logic Design?

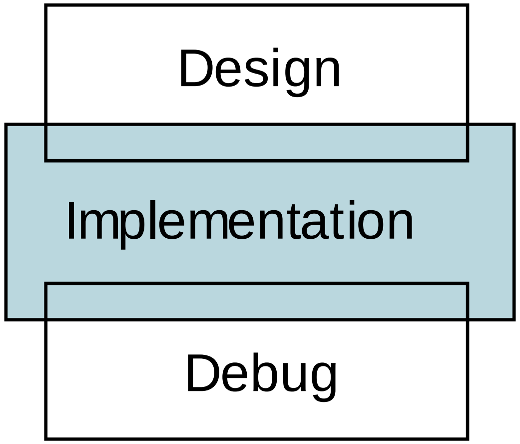
- *What is design?*
  - given a specification of a problem: a way of solving it choosing appropriately from a collection of available components
  - while keeping some criteria for size, cost, power, beauty, elegance, and etc.
- *What is logic design?*
  - determine a <u>collection of digital logic components</u> to perform a specified control, data manipulation, and communication function including the <u>interconnections</u> between them
  - which logic components to choose? – there are <u>many implementation technologies</u> (e.g., off-the-shelf fixed-function components, programmable devices, transistors on a chip, etc.)
  - the design needs to be <u>optimized and/or transformed to meet design constraints</u>

Elements of Modern Design

Representations, Circuit Technologies, & Rapid Prototyping



The Process Of Design



- *Design*

  Initial concept: what is the function performed by the object?
  Constraints: How fast? How much area? How much cost?
  Refine abstract functional blocks into more concrete realizations

- *Implementation*

  Assemble primitives into more complex building blocks
  Composition via wiring
  Choose among alternatives to improve the design

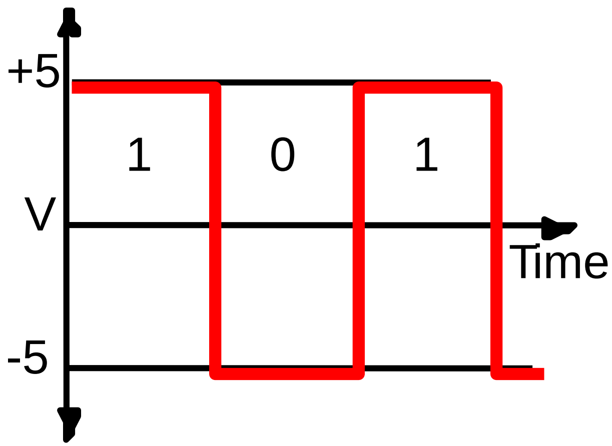
- *Debug*

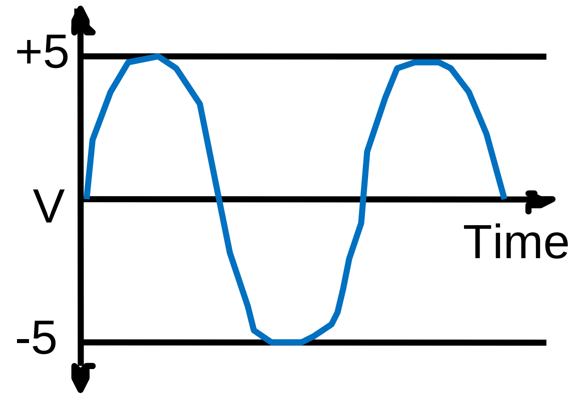
  Faulty systems: design flaws, composition flaws, & component flaws
  Design system to make debugging easier
  Hypothesis formation and troubleshooting skills

Digital Hardware Systems

Digital Systems

Digital vs. Analog Waveforms



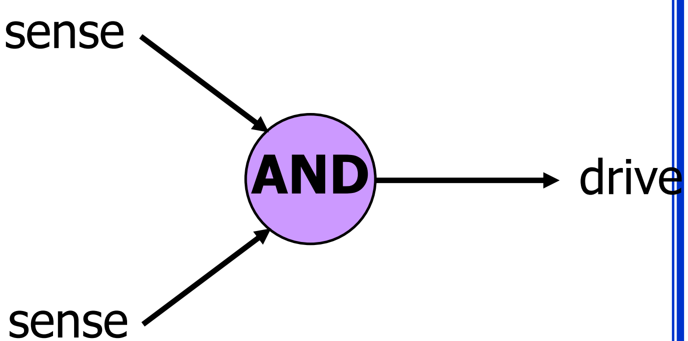


Digital:
only assumes <u>discrete values</u>

Analog:
values vary over a broad range <u>continuously</u>

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")
  - EX: digital logic where voltage < 0.8v is a "0" and > 2.0v is a "1"
  - EX: orientation of magnetization signifies a "0" or a "1"
- ***Primitive digital hardware devices***
  - logic computation devices (sense and drive)
    - two wires both "1" - make another be "1" (***AND***)
    - at least one of two wires "1" - make another be "1" (***OR***)
    - a wire "1" - then make another be "0" (***NOT***)
  - memory devices (store)
    - to store a value
    - recall a previously stored value



Modern Digital Design

- *Important trends* in how industry does hardware design
  - larger and larger designs (complex)
  - shorter and shorter time to market
  - cheaper and cheaper products
- *Scale*
  - pervasive use of computer-aided design tools over hand methods
  - <u>multiple levels of design representation</u>
- *Time*
  - <u>emphasis on abstract design representations</u>
  - programmable devices rather than fixed function components
  - automatic synthesis techniques
  - sound design methodologies
- *Cost*
  - higher levels of integration
  - use of simulation to debug designs
  - simulate and verify before you build

Course Overview

- Understand the basics of logic design (concepts)
- Understand sound design methodologies (concepts)
- Modern specification methods (concepts)
- Familiarity with a full set of CAD tools (skills)
- Realize digital designs in an implementation technology (skills)
- The differences and similarities (abilities) in hardware and software design can be specified

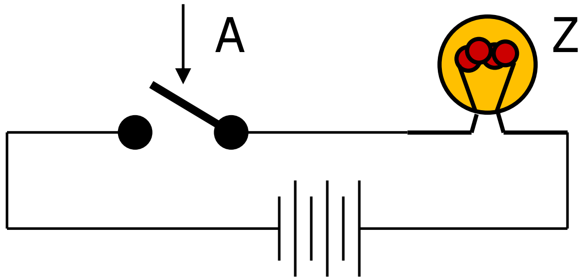
New ability: to accomplish the logic design task with the aid of computer-aided design tools and map a problem description into an implementation with programmable logic devices after validation via simulation and understanding of the advantages/disadvantages as compared to a software implementation

Computation: Abstract /Implementation

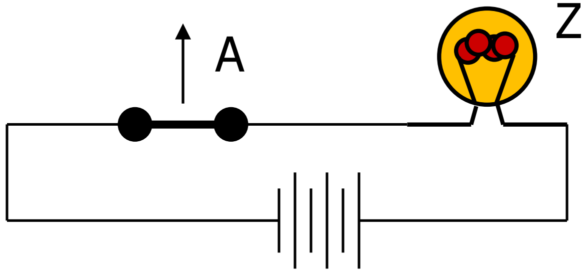
- This course is about physically implementing computation using physical devices that use voltages to represent logical values
- *Basic units of computation* are:
  - representation: "0", "1" on a wire
    set of wires (e.g., for binary ints)
  - assignment: x = y
  - data operations: x + y – 5
  - 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 can be implemented in hardware and composed into computational structures

SW: Basic Element of Implementations

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



open switch (if A is “0” or <u>unasserted</u>)
and turn off light bulb (Z)

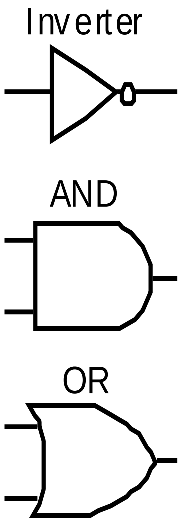


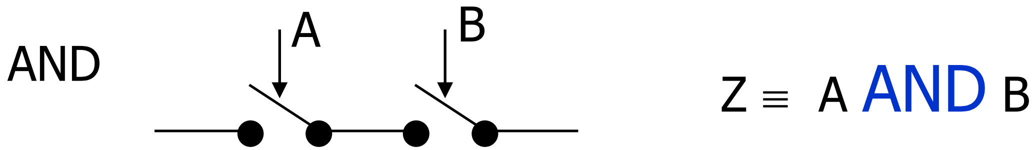
close switch (if A is “1” or <u>asserted</u>)
and turn on light bulb (Z)

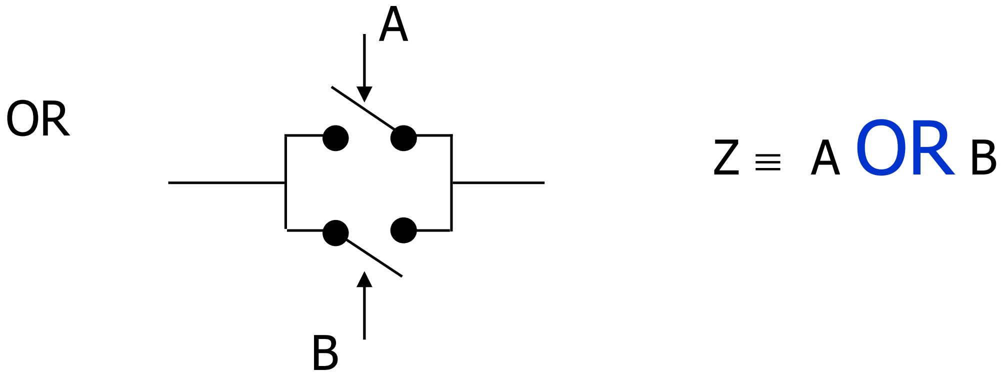
$$Z \equiv A$$

Switches

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







Switching Networks

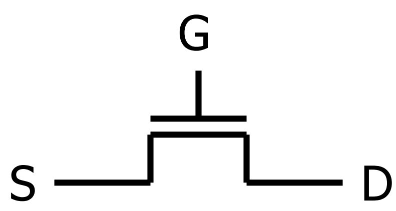
- Digital system: *networks of gates*
  - **Network** implemented from switching elements or logic gates
- Switch settings
  - determine whether or not a conducting path exists to light the light bulb
- To build larger computations
  - Use the output of any network to set other switches (as the inputs to another network)
- Connect together those 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

Transistor Networks

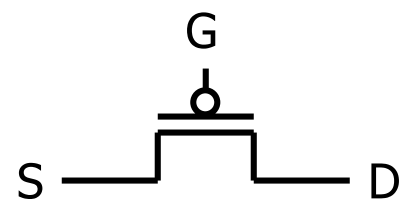
- Modern digital systems are designed in CMOS technology
  - MOS stands for Metal-Oxide on Semiconductor
  - C is for complementary because there are both normally-open and normally-closed switches
- MOS transistors act as voltage-controlled switches

MOS 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

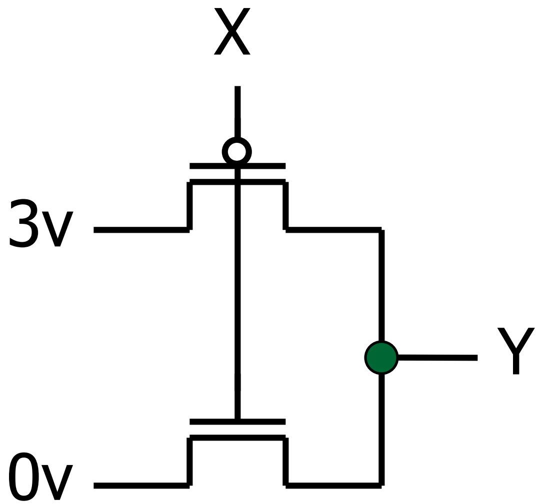


n-channel
open when voltage at G is low
closes when:
voltage(G) > voltage (S) + ε



p-channel
closed when voltage at G is low
opens when:
voltage(G) > voltage (S) – ε

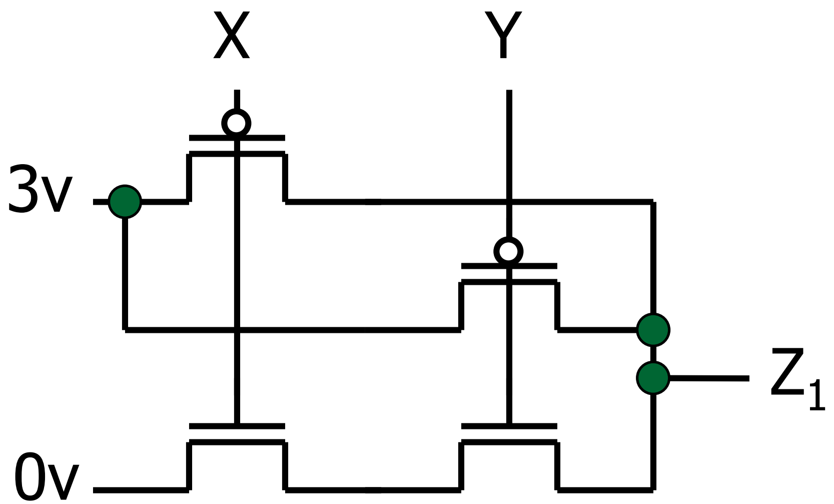
MOS Networks

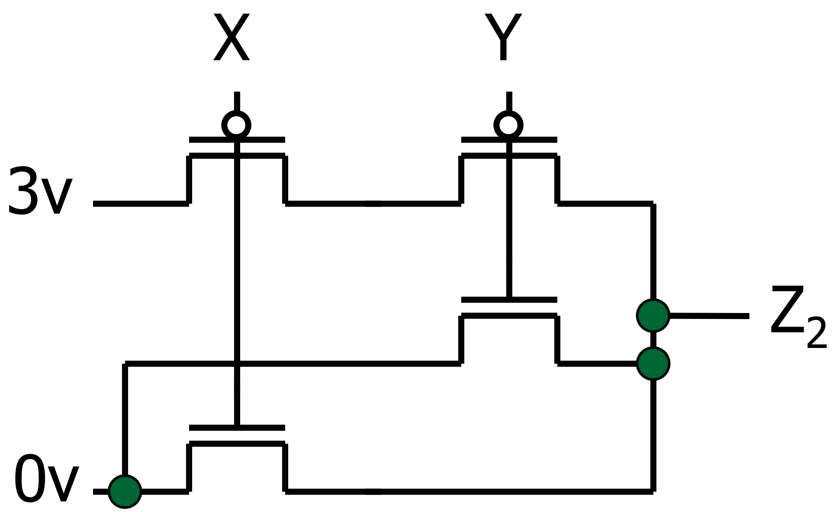


what is the relationship between x and y?

x	y
0 volts	3 volts
3 volts	0 volts

Two Input Networks





what is the relationship between x, y and z?

x	y	z1	z2
0 volts	0 volts	3 volts	3 volts
0 volts	3 volts	3 volts	0 volts
3 volts	0 volts	3 volts	0 volts
3 volts	3 volts	0 volts	0 volts
		NAND	**NOR**

Speed of MOS Networks

- What influences the *speed* of CMOS networks?
  - charging and discharging of voltages on wires and gates of transistors
- Capacitors hold charge
  - capacitance is at gates of transistors and wire material
- Resistors slow movement of electrons
  - resistance mostly due to transistors
- Out of scope: detailed physical/electrical behavior

Representation of Digital Designs

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

Course scope

Principal Representation Method

- ***Truth Tables:***

  tabulate <u>all possible input combinations and their associated output values</u>

***Example:* half adder**
adds two binary digits to form Sum and Carry

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

NOTE: 1 plus 1 is 0 with a carry of 1 in binary

$$
\begin{array}{rcccc}
 & & 1 & 1 & \\
 & & \mathbf{1} & \mathbf{1} & \mathbf{1} \\
\text{FA} & & \mathbf{1} & \mathbf{0} & \mathbf{1} \quad \text{HA} \\
\hline
 & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0}
\end{array}
$$

***Example:* full adder**
adds two binary digits and Carry in to form Sum and Carry Out

A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

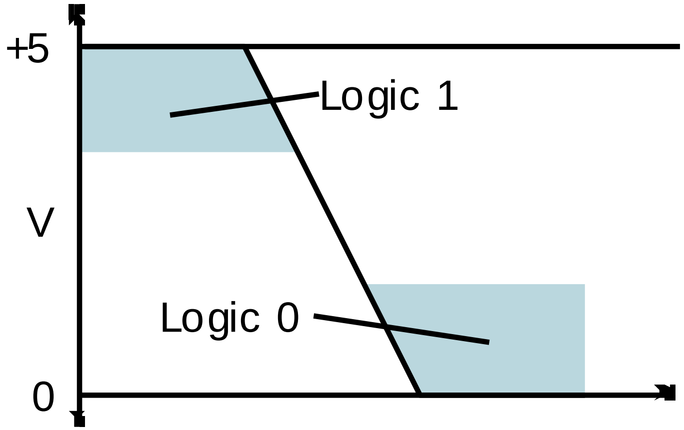
Digital vs. Analog

- 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 the digital abstraction anyway?
  - switches operate in this way
  - easier to think about a small number of discrete values

The Real World

- Physical electronic components are <u>*continuous, not discrete!*</u>
- These are the building blocks of all digital components!



Transition from logic 1 to logic 0 does <u>not take place instantaneously</u> in real digital systems

Intermediate values may be <u>visible for an instant</u>

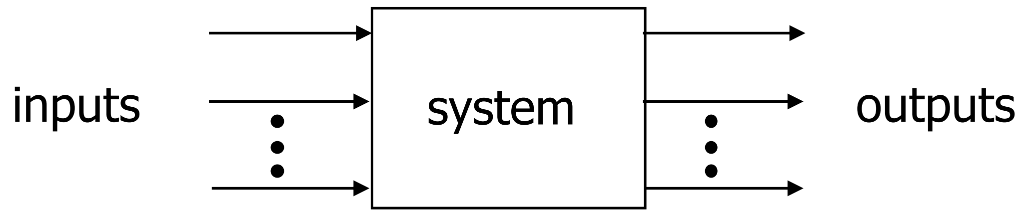
- Digital systems: describe the <u>**steady state behavior**</u>

Mapping: Physical World to Binary World

Technology	State 0 (L)	State 1 (H)
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 Circuits

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



- The presence of feedback distinguishes between *sequential* and *combinational* networks
- Combinational means *"memory-less"*
  - a digital circuit is combinational if its output values only depend on its input values
  - no feedback among inputs and outputs

Combinational Logic Symbols

- 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)

Sequential Logic

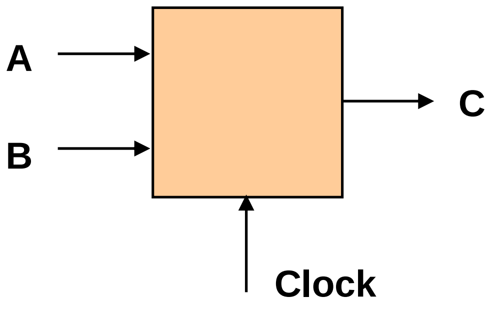
- Sequential systems
  - show behaviors (output values) that depend not only on the *current input* values, but also on *previous input values*
- A fundamental abstraction of digital design is based on (mostly) steady-state behaviors
  - look at the outputs *only after sufficient time has elapsed* for the system to make its required changes and settle down

Synchronous Sequential 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 this form 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

EX: Combinational & 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: <u>may be *different*</u>



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 encoding when more than two logical values are mapped into binary form
  - state diagrams to represent sequential logic
  - hardware description languages to represent digital logic
  - waveforms to represent temporal behavior

An Example

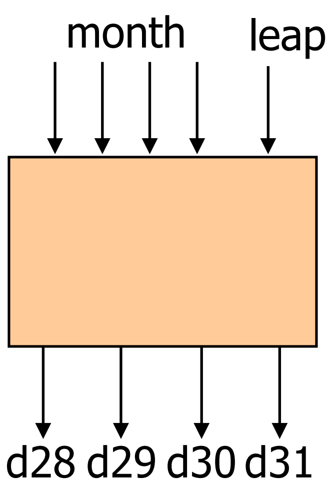
- Calendar subsystem: number of days in a month (to control watch display)
  - used in controlling the display of a wrist-watch LCD screen
  - *inputs*: month, leap year flag
  - *outputs*: number of days

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: Combinational Logic

- 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	d28	d29	d30	d31
0000	–	–	–	–	–
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
0101	–	0	0	0	1
0110	–	0	0	1	0
0111	–	0	0	0	1
1000	–	0	0	0	1
1001	–	0	0	1	0
1010	–	0	0	0	1
1011	–	0	0	1	0
1100	–	0	0	0	1
1101	–	–	–	–	–
111–	–	–	–	–	–

Combinational Logic EX

- Truth-table → logic function → switches/gates
  - d28 = 1 when month=0010 and leap=0
  - d28 = m8'•m4'•m2•m1'•leap'

    Month bit representation : m8m4m2m1
  - d31 = 1 when month=0001 or month=0011 or ... month=1100
  - d31 = (m8'•m4'•m2'•m1) + (m8'•m4'•m2•m1) + ... (m8•m4•m2'•m1')
  - d31 = can we simplify more?

symbol for not

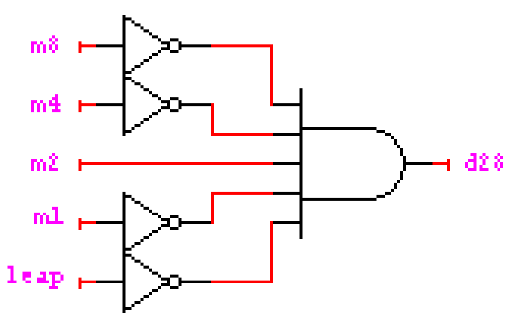
symbol for and

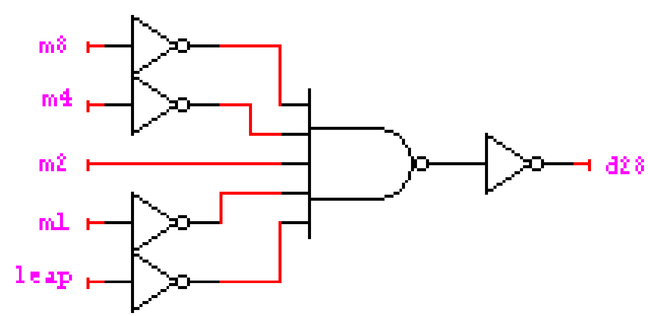
symbol for or

month	**leap**	**d28**	**d29**	**d30**	**d31**
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 Logic EX

- $d28 = m8' \cdot m4' \cdot m2 \cdot m1' \cdot leap'$
- $d29 = m8' \cdot m4' \cdot m2 \cdot m1' \cdot leap$
- $d30 = (m8' \cdot m4 \cdot m2' \cdot m1') + (m8' \cdot m4 \cdot m2 \cdot m1') + (m8 \cdot m4' \cdot m2' \cdot m1) + (m8 \cdot m4' \cdot m2 \cdot m1)$
  $= (m8' \cdot m4 \cdot m1') + (m8 \cdot m4' \cdot m1)$
- $d31 = (m8' \cdot m4' \cdot m2' \cdot m1) + (m8' \cdot m4' \cdot m2 \cdot m1) + (m8' \cdot m4 \cdot m2' \cdot m1) + (m8' \cdot m4 \cdot m2 \cdot m1) + (m8 \cdot m4' \cdot m2' \cdot m1') + (m8 \cdot m4' \cdot m2 \cdot m1') + (m8 \cdot m4 \cdot m2' \cdot m1')$





Activity

- How much can we simplify d31?

Month: m8m4m2m1

d31 is true if: month is 7 or less and odd (1, 3, 5, 7), or
month is 8 or more and even (8, 10, 12, and includes 14)

d31 is true if: m8 is 0 and m1 is 1, or m8 is 1 and m1 is 0

$$d31 = m8'm1 + m8m1'$$

- What if we started the months with 0 instead of 1? (i.e., January is 0000 and December is 1011)

More complex expression (0, 2, 4, 6, 7, 9, 11):

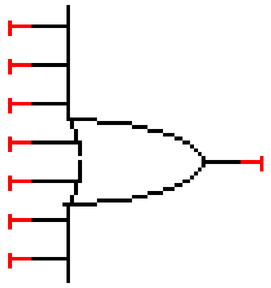
$$d31 = m8'm4'm2'm1' + m8'm4'm2m1' + m8'm4m2'm1' + m8'm4m2m1' + m8'm4m2m1 + m8m4'm2'm1 + m8m4'm2m1$$

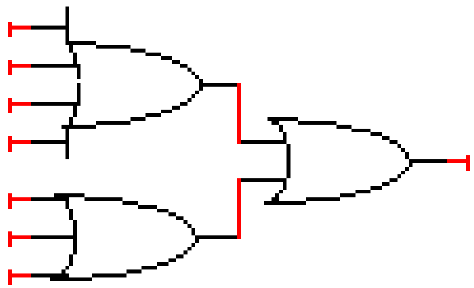
$d31 = m8'm1' + m8'm4m2 + m8m1$ (includes 13 and 15)

$$d31 = (d28 + d29 + d30)'$$

Combinational Logic EX

- d28 = m8'•m4'•m2•m1'•leap'
- d29 = m8'•m4'•m2•m1'•leap
- d30 = (m8'•m4•m2'•m1') + (m8'•m4•m2•m1') +
  (m8•m4'•m2'•m1) + (m8•m4'•m2•m1)
- d31 = (m8'•m4'•m2'•m1) + (m8'•m4'•m2•m1) +
  (m8'•m4•m2'•m1) + (m8'•m4•m2•m1) +
  (m8•m4'•m2'•m4') + (m8•m4'•m2•m1') +
  (m8•m4•m2'•m1')





Another Example

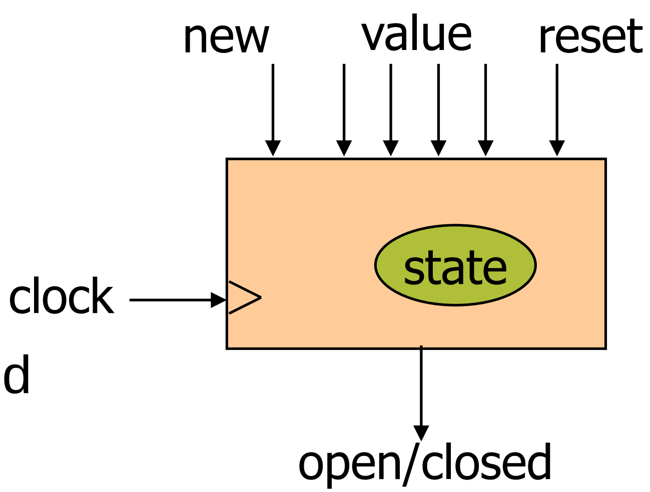
- Door combination lock:
  - punch in 3 values in sequence and the door opens; if there is an error the lock must be reset; once the door opens the lock must be reset
  - *inputs*: sequence of input values, reset
  - *outputs*: door open/close
  - *memory*: must remember combination
    or always have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

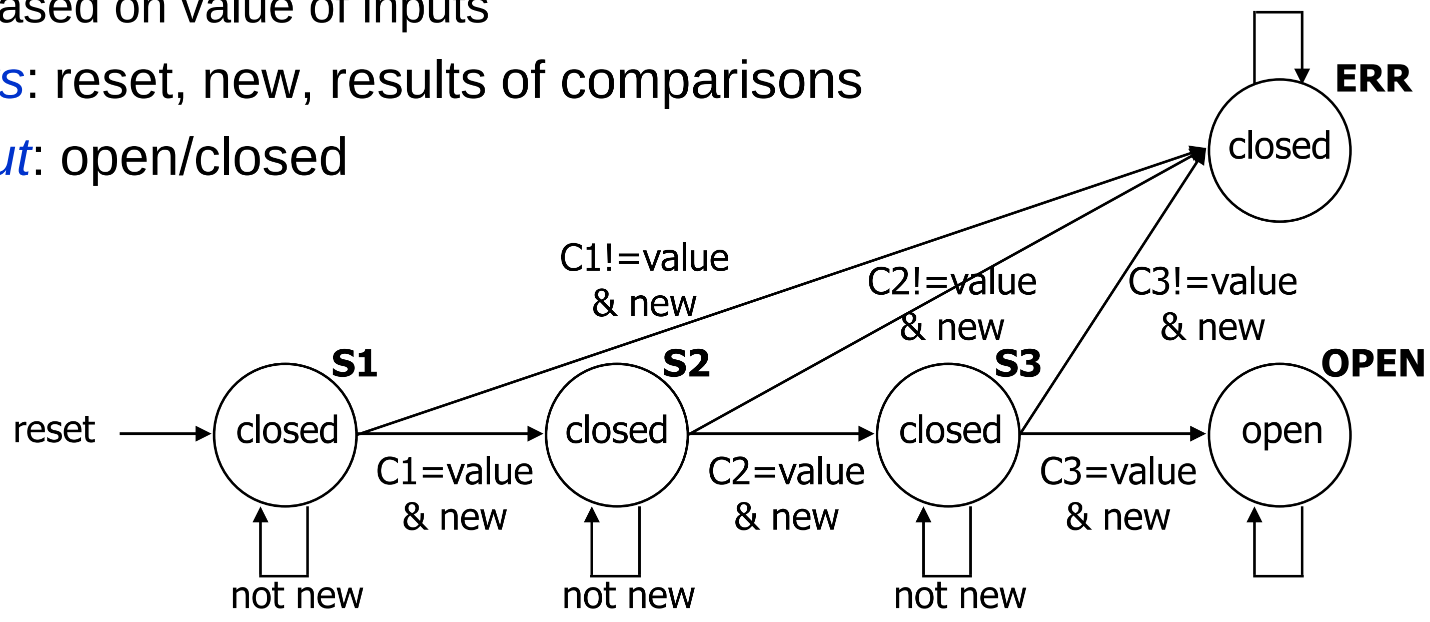
Implementation: Sequential Logic

- *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



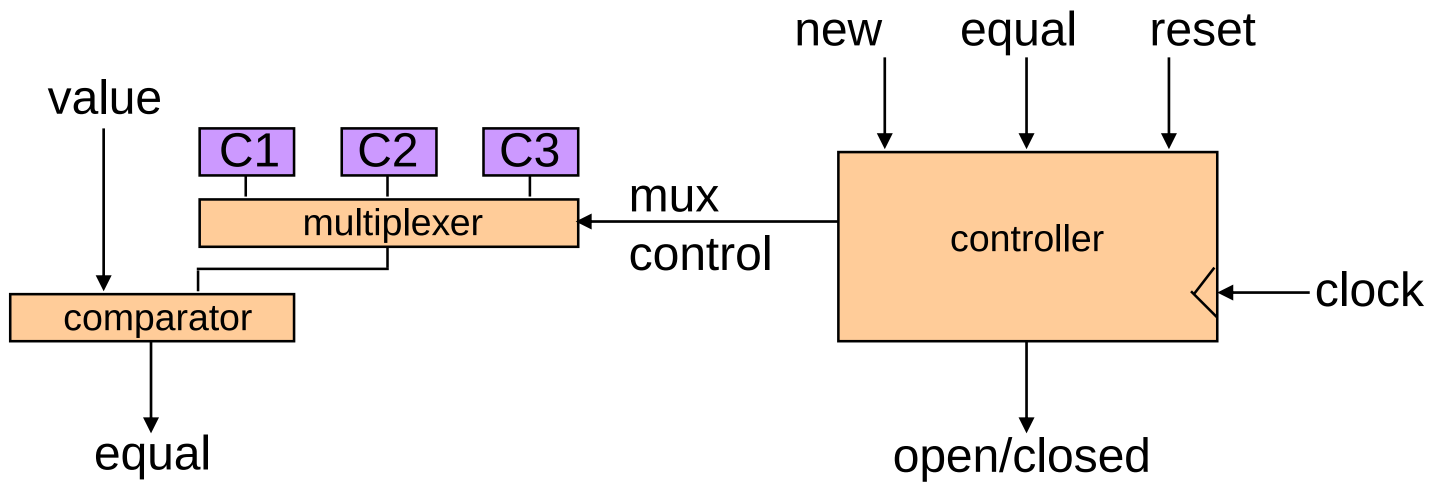
Sequential EX : Abstract Control

- Finite-state diagram
  - *states*: <u>5 states</u>
    - 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



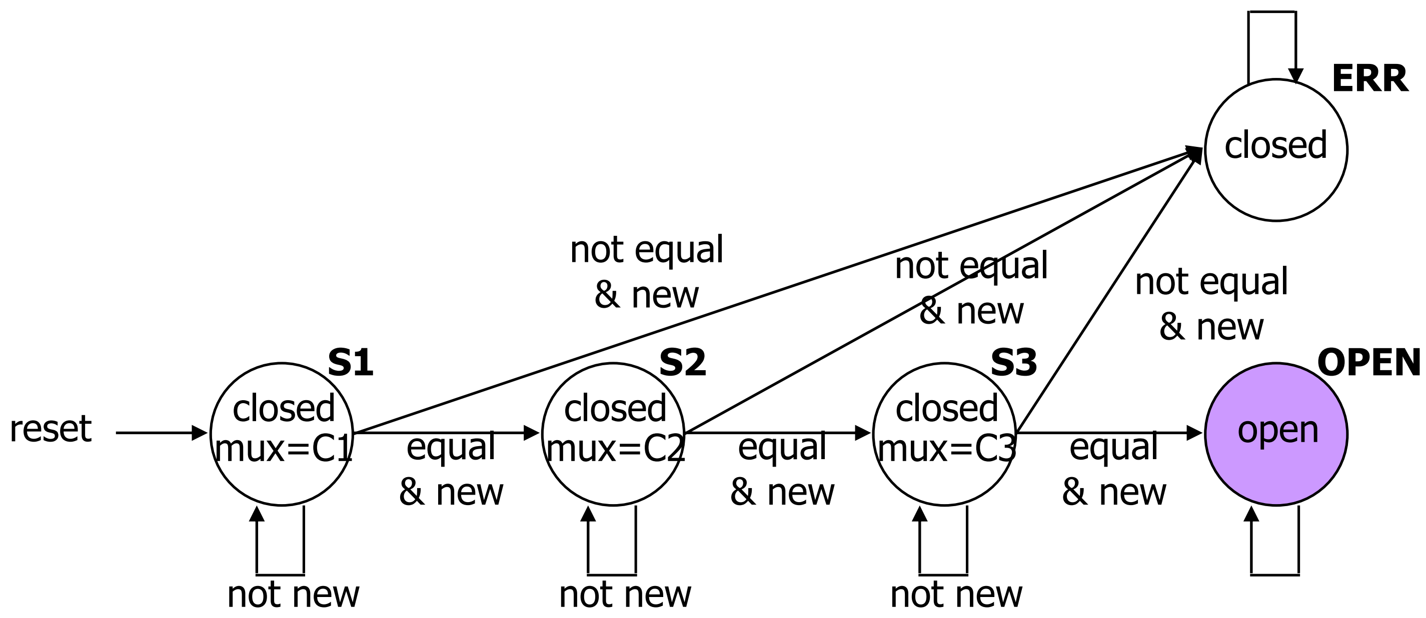
Sequential EX : Data-path vs. Control

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



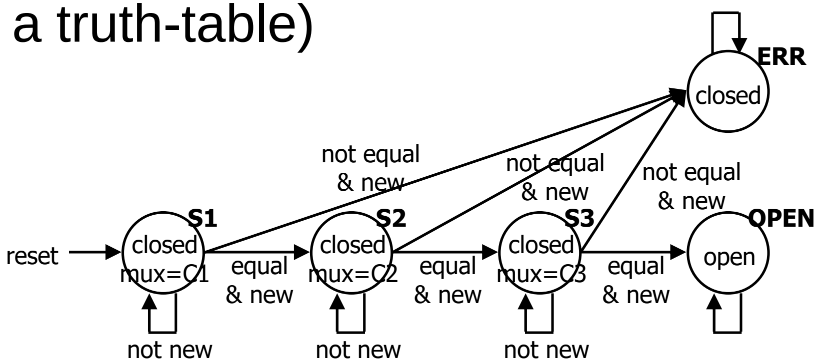
Sequential EX: Finite-State Machine

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



Sequential EX: Finite-State Machine

- Finite-state machine
  - 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	0	–	S2	S2	C2	closed
0	1	0	S2	ERR	–	closed
0	1	1	S2	S3	C3	closed
0	0	–	S3	S3	C3	closed
0	1	0	S3	ERR	–	closed
0	1	1	S3	OPEN	–	open
0	–	–	OPEN	OPEN	–	open
0	–	–	ERR	ERR	–	closed

Sequential EX : Encoding

- Encode state table
  - states can be: S1, S2, S3, OPEN, or ERR
    - needs at least 3 bits to encode: 000, 001, 010, 011, 100
    - or 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
    - need 2 to 3 bits to encode
    - choose 3 bits: 001, 010, 100
  - output open/closed can be: open or closed
    - need 1 or 2 bits to encode
    - choose 1 bits: 1, 0

Sequential EX : Encoding

- Encode state table
  - state can be: S1, S2, S3, OPEN, or ERR
    - choose 4 bits: 0001, 0010, 0100, 1000, 0000
  - output mux can be: C1, C2, or C3
    - choose 3 bits: 001, 010, 100
  - output open/closed can be: open or closed
    - 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	0	–	0010	0010	010	0
0	1	0	0010	0000	–	0
0	1	1	0010	0100	100	0
0	0	–	0100	0100	100	0
0	1	0	0100	0000	–	0
0	1	1	0100	1000	–	1
0	–	–	1000	1000	–	1
0	–	–	0000	0000	–	0

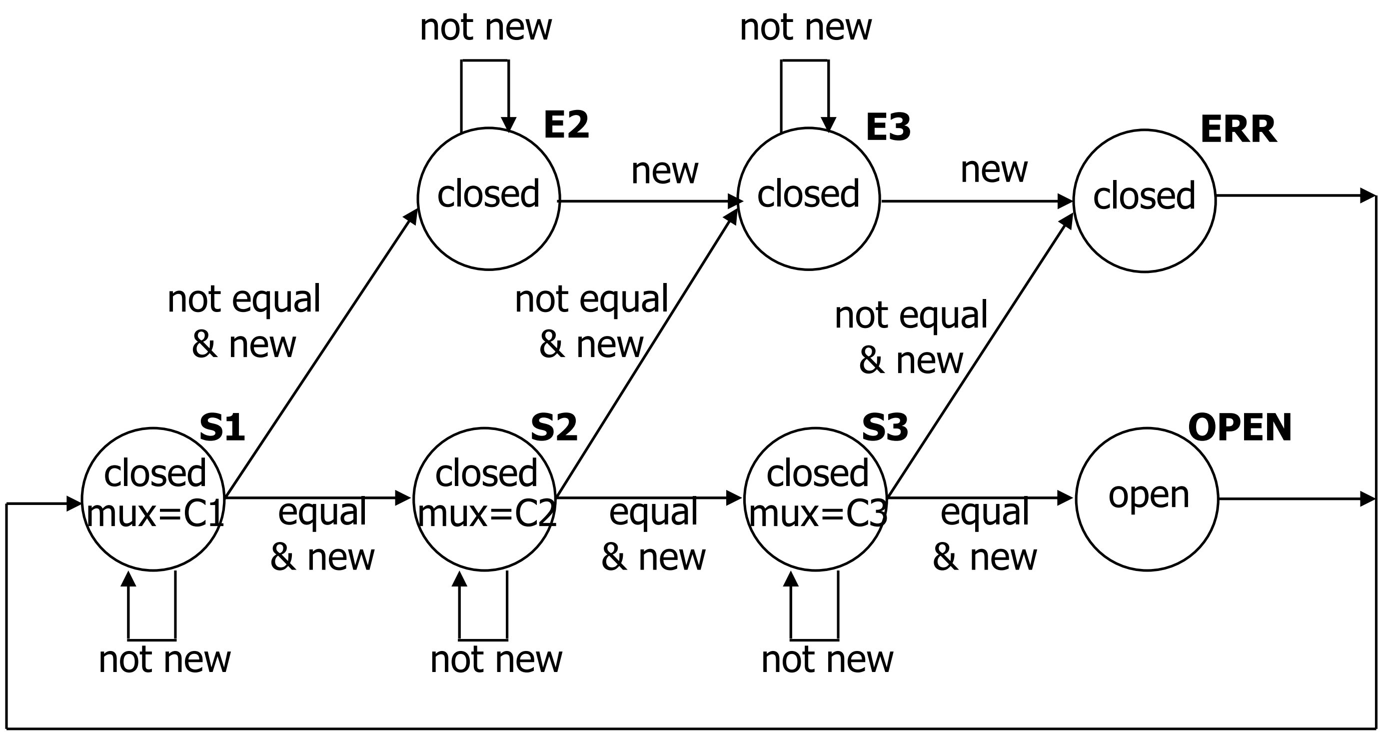
good choice of encoding!

mux is identical to last 3 bits of state

open/closed is identical to first bit of state

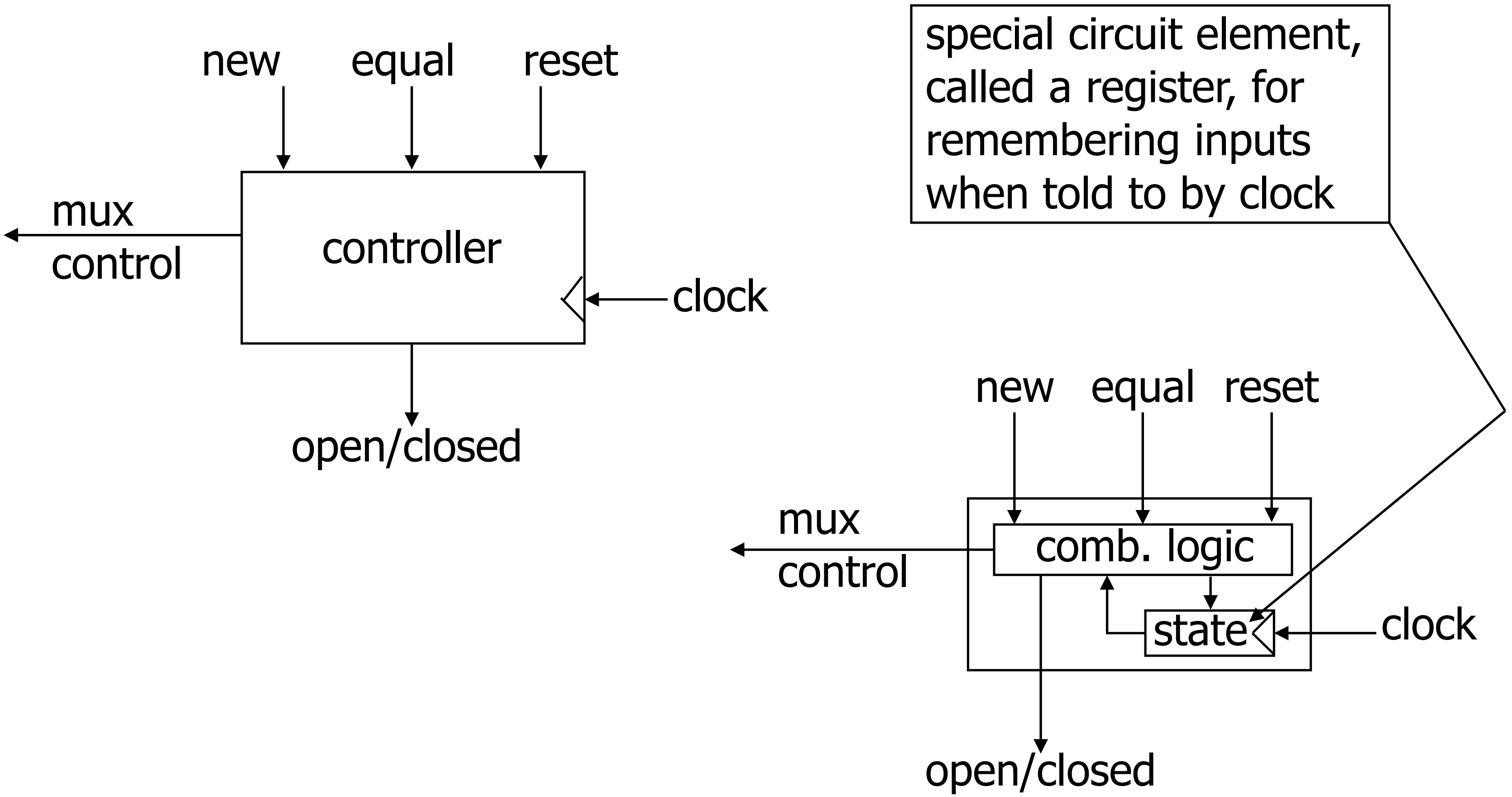
Activity

- Have lock always wait for 3 key presses exactly before making a decision
  - remove reset

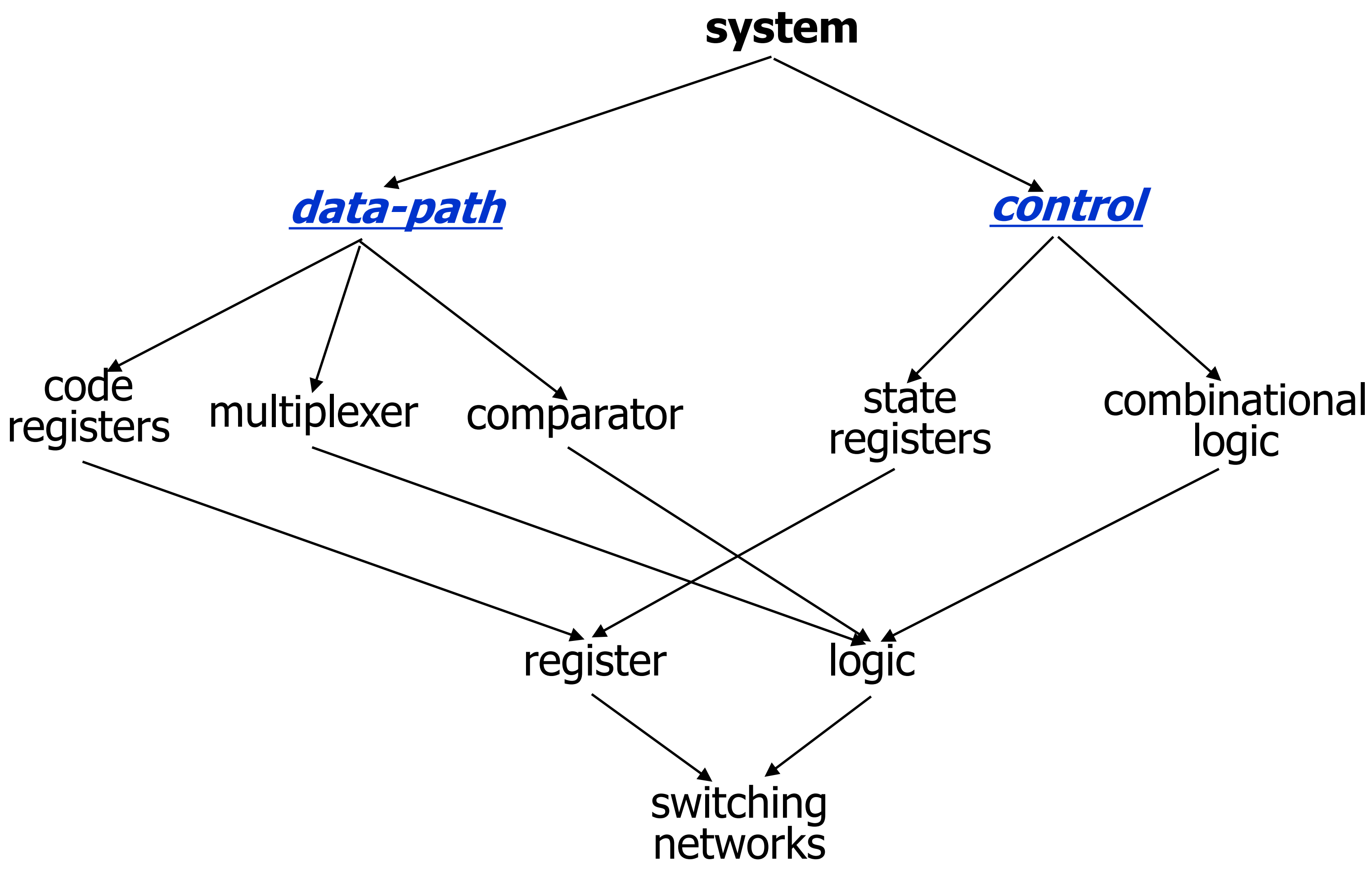


Sequential EX: Controller Implementation

- 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
- Rest of course: learn *one by one* during semester