CSE370: Introduction to Digital Design

- Course staff
  - Gaetano Borriello, Brian DeRenzi, Firat Kiyak
- Course web
  - www.cs.washington.edu/370/
  - Make sure to subscribe to class mailing list (cse370@cs)
- Course text
  - Contemporary Logic Design, 2e, Katz/Borriello, Prentice-Hall
- Today's agenda
  - Class administration and overview of course web
  - Course objectives and approach
  - Classroom Presenter

Why are you here?

- 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
      - computers = transistors + wires - it's all in how they are interconnected
    - provide a model of how a computer works
- More important reasons
  - the inherent parallelism in hardware is your first exposure to parallel computation
  - it offers an interesting counterpoint to programming and is therefore useful in furthering our understanding of computation

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/realize 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
- Contrast with programming
  - sequential and parallel implementations
  - specify algorithm as well as computing/storage resources it will use

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!

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 components
  - while meeting some 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 logical "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 previously stored value

What is happening now in digital design?

- Important trends in how industry does hardware design
  - larger and larger designs
  - shorter and shorter time to market
  - cheaper and cheaper products
  - design time often dominates cost
- 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
  - simulate and verify before you build

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)
- Realize digital designs in an implementation technology (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

Representation of digital designs

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

scope of CSE 370 (Switches through Register-transfer behavior)

Computation: abstract vs. implementation

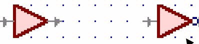
- Up to now, computation has been a mental exercise (paper, programs)
- This class is about physically implementing computation using physical devices that use voltages to represent logical values
- Basic units of computation are:
  - representation: "0", "1" on a wire; set of wires (e.g., for binary 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 are implemented in hardware and composed into computational structures

Class components

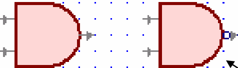
- Combinational logic
  - $output_t = F(input_t)$
- Sequential logic
  - $output_t = F(output_{t-1}, input_t)$
    - output dependent on history
    - concept of a time step (clock)
- Basic computer architecture
  - how a CPU executes instructions

Combinational logic

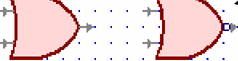
- Common combinational logic elements are called logic gates
  - Buffer, NOT



  - AND, NAND

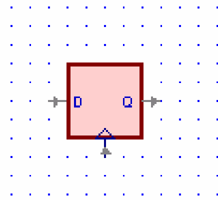


  - OR, NOR



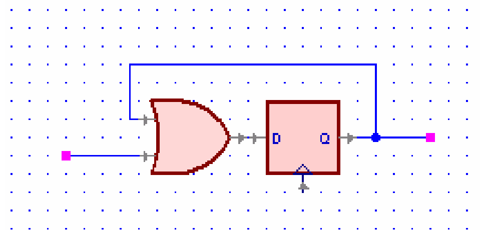
Sequential logic

- Common sequential logic elements are called flip-flops
  - Flip-flops only change their output after a clocking event



Mixing combinational and sequential logic

- What does this very simple circuit do?



Combinational or sequential?

- Circle combinational in red, sequential in blue
  - assignment: x = y;
  - data operations: x + y – 5
  - sequential statements: A; B; C;
  - conditionals: if x == 1 then y;
  - loops: for (i = 1 ; i == 10, i++) {…}
  - procedures/methods: A; proc(...); B;

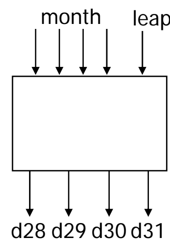
A combinational logic example

- 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 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	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 example (cont'd)

- Truth-table to logic to switches to gates
  - d28 = 1 when month=0010 and leap=0
  - d28 = m8'•m4'•m2•m1'•leap'
  - 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 example (cont'd)

- 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)
  = (m8'•m4•m1') + (m8•m4'•m1)
- d31 = (m8'•m4'•m2'•m1) + (m8'•m4'•m2•m1) +
  (m8'•m4•m2'•m1) + (m8'•m4•m2•m1) +
  (m8•m4'•m2'•m1') + (m8•m4'•m2•m1') +
  (m8•m4•m2'•m1')

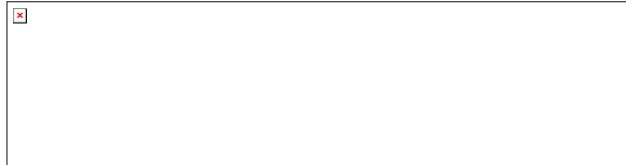
Activity

- How much can we simplify d31?

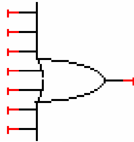
month	d31
0000	–
0001	1
0010	0
0011	1
0100	0
0101	1
0110	0
0111	1
1000	1
1001	0
1010	1
1011	0
1100	1
1101	–
1110	–
1111	–

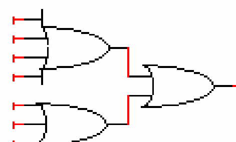
Realization of combinational logic

- d28 = m8'•m4'•m2•m1'•leap'



- 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

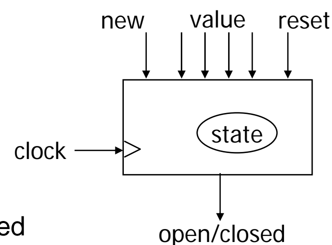
- Door combination lock:
  - punch in 3 values in sequence and the door opens; if there is an error the lock must be reset; once the door opens the lock must be reset
  - inputs: sequence of input values, reset
  - outputs: door open/close
  - memory: must remember combination
    or always have it available 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]) error = 1;

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

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

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

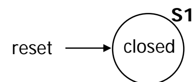

Implementation as a sequential digital system

- 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



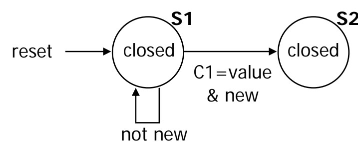
Sequential example: abstract control

- Finite-state diagram
  - states (with outputs: open/closed flag)
    - represent point in execution of machine
  - transitions (based on inputs: reset, new, comparison results)
    - changes of state occur when clock says it's ok



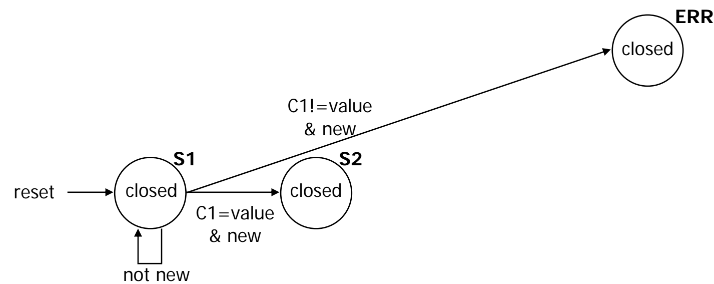
Sequential example: abstract control

- Finite-state diagram
  - states (with outputs: open/closed flag)
    - represent point in execution of machine
  - transitions (based on inputs: reset, new, comparison results)
    - changes of state occur when clock says it's ok



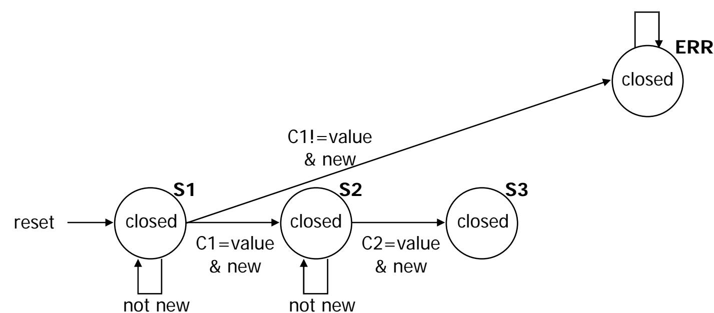
Sequential example: abstract control

- Finite-state diagram
  - states (with outputs: open/closed flag)
    - represent point in execution of machine
  - transitions (based on inputs: reset, new, comparison results)
    - changes of state occur when clock says it's ok



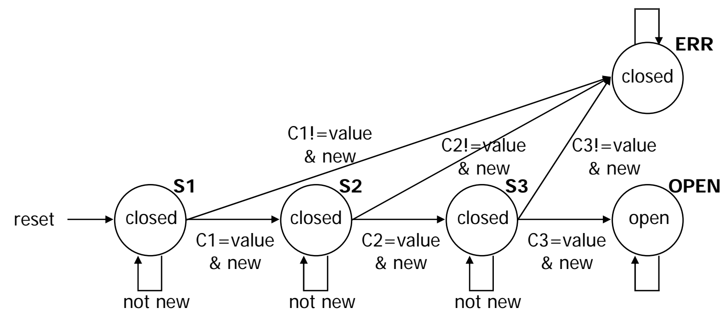
Sequential example: abstract control

- Finite-state diagram
  - states (with outputs: open/closed flag)
    - represent point in execution of machine
  - transitions (based on inputs: reset, new, comparison results)
    - changes of state occur when clock says it's ok



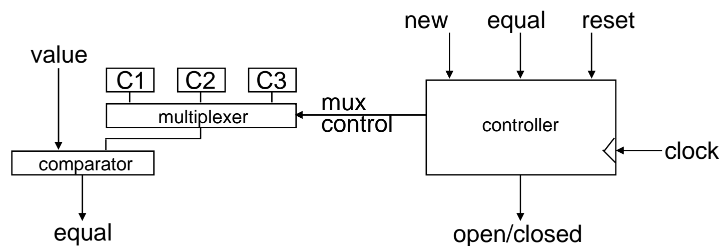
Sequential example: abstract control

- Finite-state diagram
  - states: 5 states
  - transitions: 6 from state to state, 5 self transitions, 1 global
  - inputs: reset, new, results of comparisons
  - output: open/closed



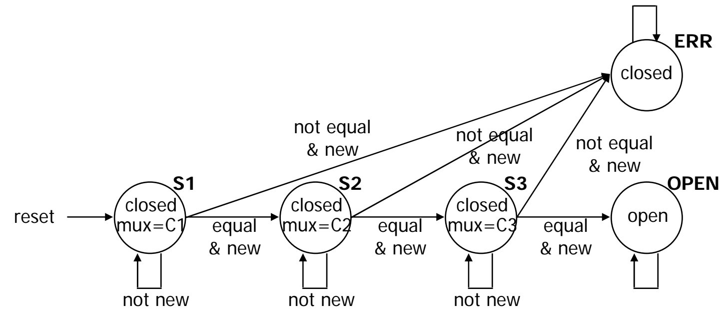
Sequential example: data-path vs. control

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



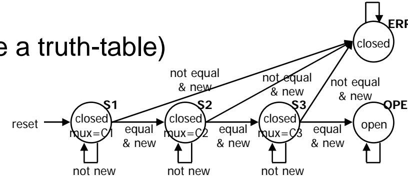
Sequential example: finite-state machine

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



Sequential example: state table

- 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 example: 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

Sequential example: 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

Sequential example (cont'd): 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

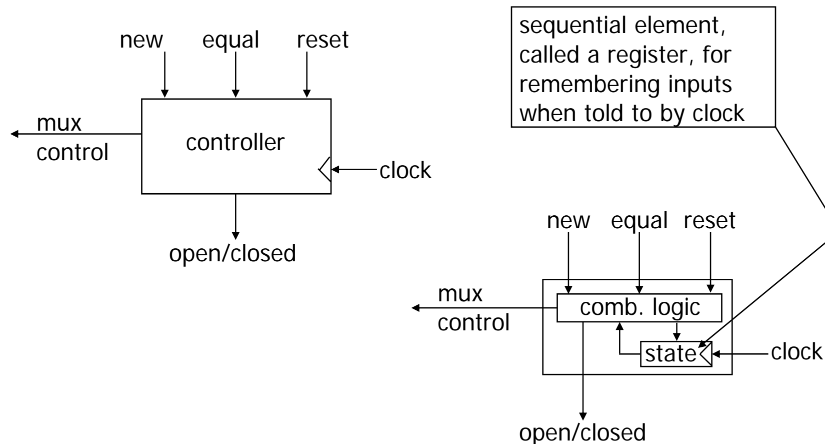
good choice of encoding!

mux is identical to last 3 bits of state

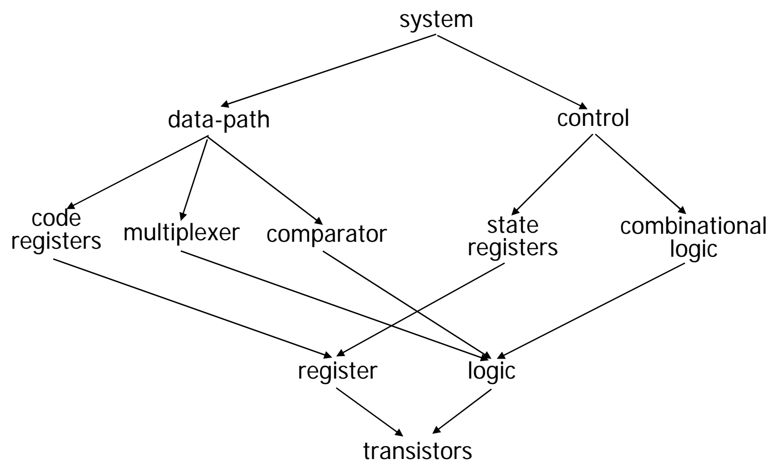
open/closed is identical to first bit of state

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 instead of one