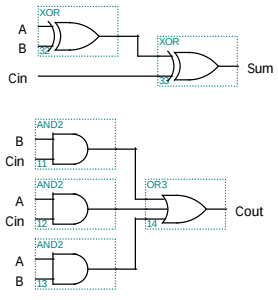
Lecture 15

- Logistics
  - HW4 is due today
  - HW5 posted today
  - Exam questions: to me
  - Class feedback
- Last lecture
  - Adders
- Today
  - More on Adder timing issues (hard!)
  - Summary of Combinational Logic
  - Introduction to Sequential Logic
    - The basic concepts
    - An example

Binary full adder

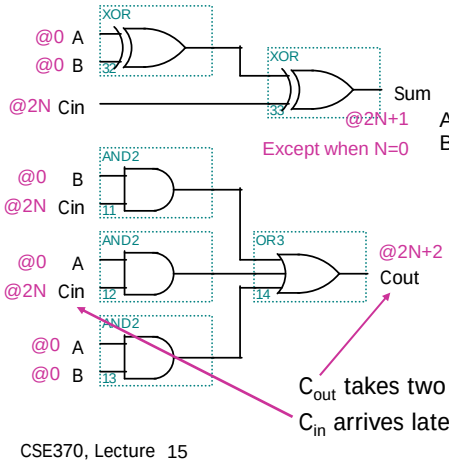
- 1-bit full adder
  - Computes sum, carry-out
    - Carry-in allows cascaded adders
  - Sum = Cin xor A xor B
  - Cout = ACin + BCin + AB

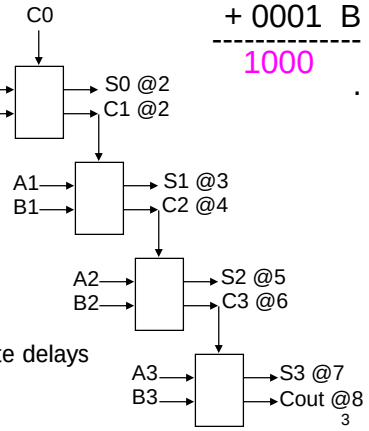
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Problem: Ripple-carry delay

- Carry propagation limits adder speed






```
  0111  A
+ 0001  B
----------
  1000
```


Speeding up the adder

- Need to find a way to "predict" Cout for all bits
- Without knowing what Cin is

Cout is always 0

```
   0
 + 0
--------
Predict Cout
```

Cout is 0 if Cin is 0
Cout is 1 if Cin is 1

```
   0
 + 1
--------
Predict Cout
```

Call this PROPAGATE

- Let's try all cases:
- A = 0, B = 0 but not sure of Cin
- A = 0, B = 1 but not sure of Cin
- A = 1, B = 0 but not sure of Cin
- A = 1, B = 1 but not sure of Cin

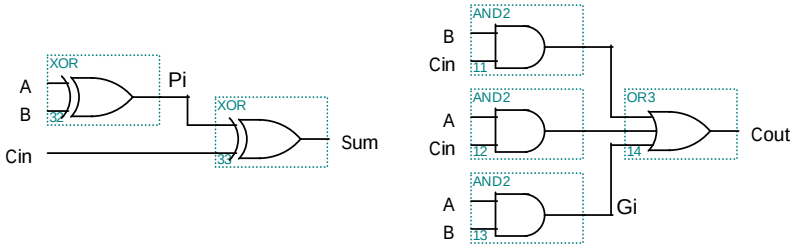
Cout is 0 if Cin is 0
Cout is 1 if Cin is 1

```
   1
 + 0
--------
Predict Cout
```

Cout is always 1
Call this GENERATE

```
   1
 + 1
--------
Predict Cout
```

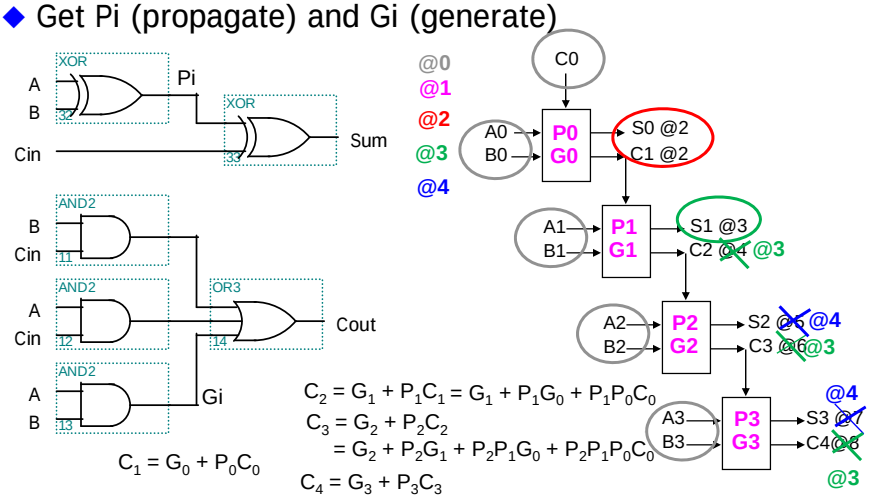
Solution: Create a carry lookahead logic Getting Pi an Gi



- Carry generate: $G_i = A_iB_i$ for i-th bit
  - Generate Cout when A = B = 1
- Carry propagate: $P_i = A_i$ xor B_i for i-th bit
  - Propagate Cin to Cout when (A xor B) = 1
- So, Cout = G + PCin $\qquad C_{i+1} = G_i + P_iC_i$

One Solution: Carry lookahead logic

- Get Pi (propagate) and Gi (generate)



$C_1 = G_0 + P_0C_0$

$C_2 = G_1 + P_1C_1 = G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2C_2$
$= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$

$C_4 = G_3 + P_3C_3$
$= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$

We've finished combinational logic…

- Negative numbers in binary
- Truth tables
- Basic logic gates
- Schematic diagrams
- Minterm and maxterm expansions (canonical, minimized)
- de Morgan's theorem
- AND/OR to NAND/NOR logic conversion
- K-maps, logic minimization, don't cares
- Multiplexers/demultiplexers
- PLAs/PALs
- ROMs
- Multi-level logics
- Timing diagrams
- Hazards
- Adders

We had no way to store memory:
When the input changed, the output changed

Next: Sequential logic can store memory…

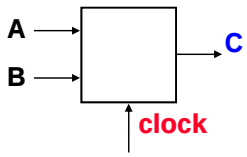
Sequential Logic (next 5 weeks!)

- We learn the details
  - Latches, flip-flops, registers (storage)
  - Shift registers, counters (we can count now!)
  - State machines (when we can store, we have states)
  - Moore and Mealy machines (types of state machines)
  - Timing and timing diagrams
    - timing more important than combinational logic
  - Synchronous and asynchronous inputs
    - Metastability (problem!)

The “WHY” slide

- Learning sequential logic
  - Having the ability to hold memory is important. If you couldn’t use your prior knowledge stored in the memory, you wouldn’t be very smart (and same goes for a computer).

Sequential versus combinational

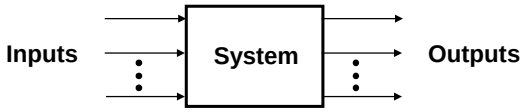


Apply fixed inputs A, B
When the clock ticks, the output becomes available
Observe C
Wait for another clock tick
Observe C again

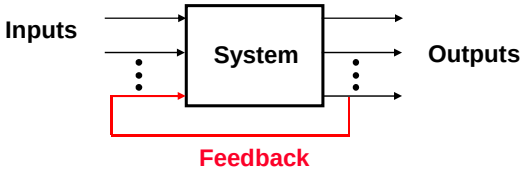
Combinational: C will stay the same
Sequential: C may be different

Sequential versus combinational

- Combinational systems are memoryless
  - Outputs depend only on the present inputs

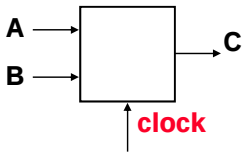


- Sequential systems have memory
  - Outputs depend on the present and the previous inputs

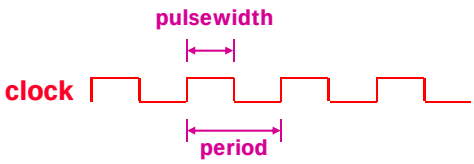


Synchronous sequential systems

- Memory holds a system's *state*
  - Changes in state occur at specific times
  - A periodic signal times or clocks the state changes
  - The clock period is the time between state changes



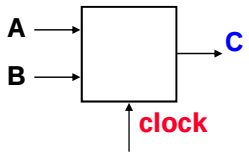
State changes occur at rising edge of clock



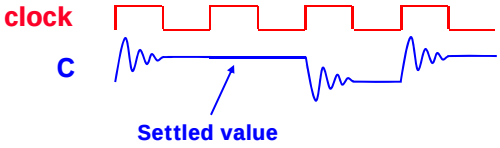
duty cycle = pulsewidth / period (here it is 50%)

Steady-state abstraction

- Outputs retain their *settled values*
  - The clock period must be long enough for all voltages to settle to a steady state before the next state change



Clock hides transient behavior



What did I just say about sequential logic?

- Has clock
  - Synchronous = clocked
  - Exception: Asynchronous
- Has state
  - State = memory
- Employs feedback
- Assumes steady-state signals
  - Signals are valid after they have settled
  - State elements hold their settled output values

Example: A sequential system

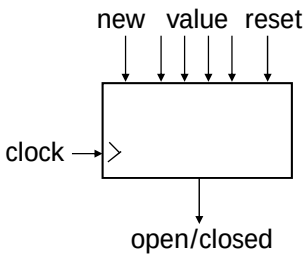
- Door combination lock
  - Enter three numbers in sequence and the door opens
  - When one number is entered, press 'enter'
  - If there is an error the lock must be reset
  - After the door opens the lock must be reset
  - Inputs: Sequence of numbers, reset, enter
  - Outputs: Door open/close
  - Memory: Must remember the combination

We will go through the motion of designing a real system

We will teach details of "how" to do these steps in the next few weeks

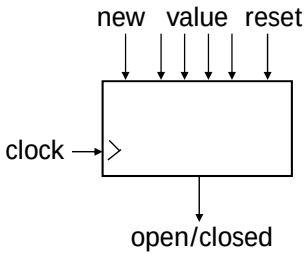
Understand the problem

- Consider I/O and unknowns
  - How many bits per input?
  - How many inputs in sequence?
  - How do we know a new input is entered?
  - How do we represent the system states?



Implement using sequential logic

- Behavior
  - Clock tells us when to look at inputs
    - After inputs have settled
  - Sequential: Enter sequence of numbers
  - Sequential: Remember if error occurred
- A diagram may be helpful
  - Assume synchronous inputs
  - State sequence
    - Enter 3 numbers serially
    - Remember if error occurred
  - All states have outputs
    - Lock open or closed

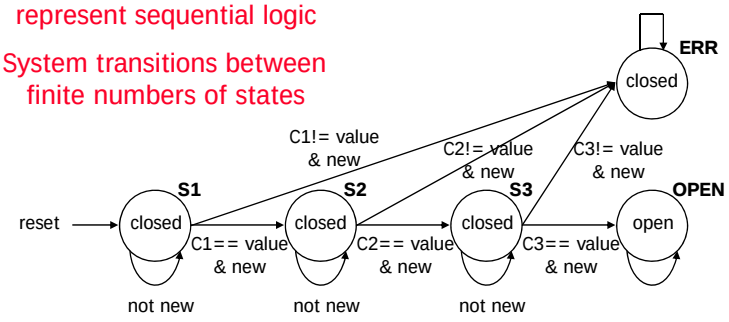


A diagram (called finite-state diagram)

- States: 5
  - Each state has outputs
- Outputs: open/closed
- Inputs: reset, new, results of comparisons
  - Assume synchronous inputs

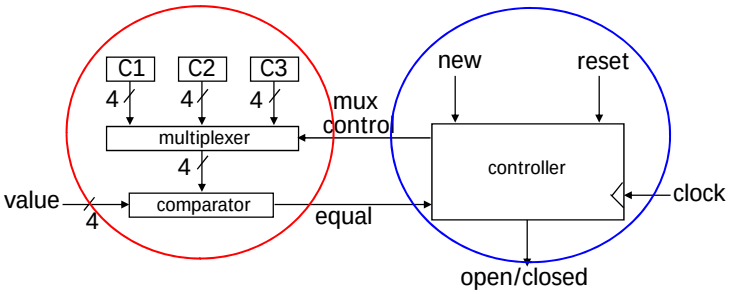
We use state diagrams to represent sequential logic

System transitions between finite numbers of states



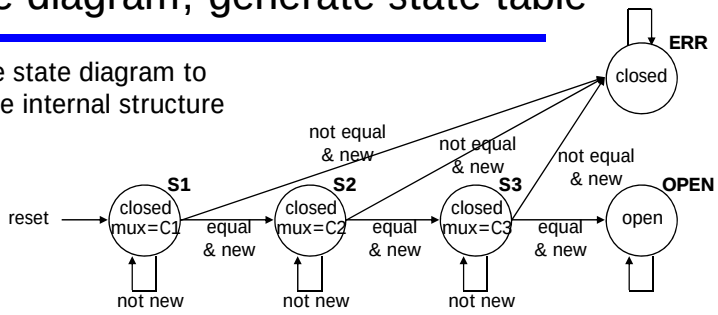
Separate data path and control

- Data path
  - Stores combination
  - Compares inputs with combination
- Control
  - Finite state-machine controller
  - Control for data path
  - State changes clocked



Refine diagram; generate state table

- Refine state diagram to include internal structure



- Generate state 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
…						

Encode state table

- State can be: S1, S2, S3, OPEN, or ERR
  - Need at least 3 bits to encode: 000, 001, 010, 011, 100
  - Can use 5 bits: 00001, 00010, 00100, 01000, 10000
  - Choose 4 bits: 0001, 0010, 0100, 1000, 0000
- Output to mux can be: C1, C2, or C3
  - Need 2 or 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 bit: 1, 0

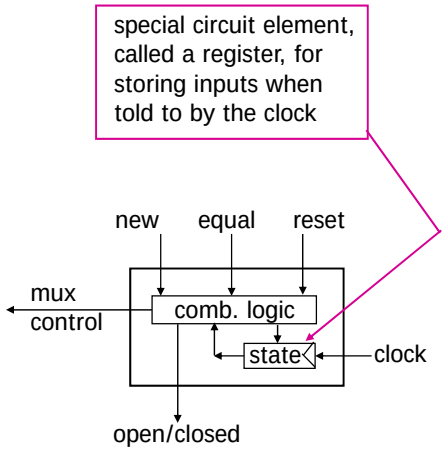
Encode state table (con't)

- Good encoding choice!
  - Mux control is identical to last 3 state bits
  - Open/closed is identical to first state bit
  - Output encoding ⇒ the outputs and state bits are the same

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
…						

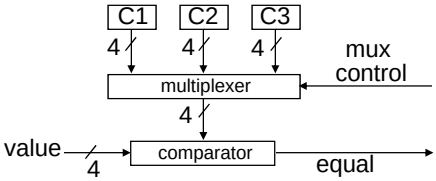
Implementing the controller

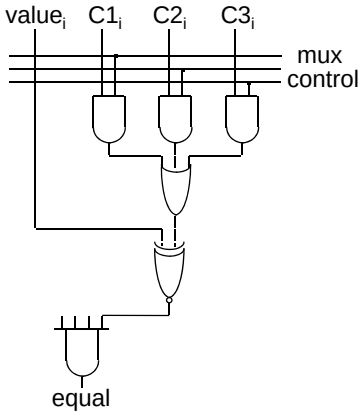
- We will learn how to design the controller given the encoded state-transition table



Designing the datapath

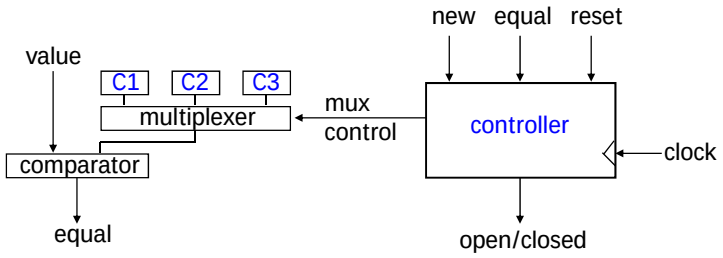
- Four multiplexers
  - 2-input ANDs and 3-input OR
- Four single-bit comparators
  - 2-input XNORs
- 4-input AND





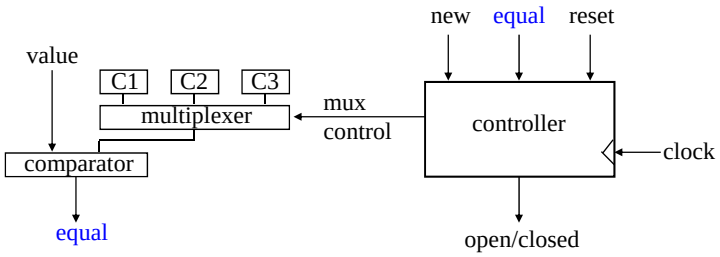
Where did we use memory?

- Memory: Stored combination, state (errors or successes in past inputs)



Where did we use feedback?

- Feedback: Comparator output ("equal" signal)



Where did we use clock?

- Clock synchronizes the inputs
  - Accept inputs when clock goes high
- Controller is clocked
  - Mux-control and open/closed signals change on the clock edge