CEG 3320 - Digital System Design

Instructor: Travis Doom, Ph.D.
331 Russ Engineering Center
775-5105
travis.doom@wright.edu
http://www.wright.edu/~travis.doom

Lecture slides created by T. Doom for Wright State University's course in Digital System Design. Some slides contain fair use images or material used with permission from textbooks and slides by R. Haggard, F. Vahid, Y. Patt, J. Wakerly, M. Mano, and other sources.

Module IV:
Design, optimization, and complexity

Sequential design optimization

Ad-hoc design

Registers

Register-level devices

Binary Arithmetic & ALUs

ROMs and other PLDs

Sequential Design Optimization

State minimization

State assignment

Mealy and Moore devices

Design Example

State Minimization

- Minimal state table use fewest possible states
  - desirable because it usually means less hardware
- If state table created from a word problem is NOT minimal, then:
  - Identify **equivalent states**:
    - Two states are equivalent if: both give the same current outputs for all inputs **and** all the next states are the same or equivalent for both states
    - e.g.: Two states are equivalent if all future outputs are the same
  - Replace all equivalent states with a single state.
- For small designs, minimization should not be necessary if the designer is careful when creating the state table.
- For larger designs, formal state minimization procedures are sometimes necessary.

State Minimization - Example 1

	X		
S	0	1	Y
A	D	B	0
B	C	D	1
C	A	B	1
D	C	D	1
	S(t+1)		

State Minimization - Example 2

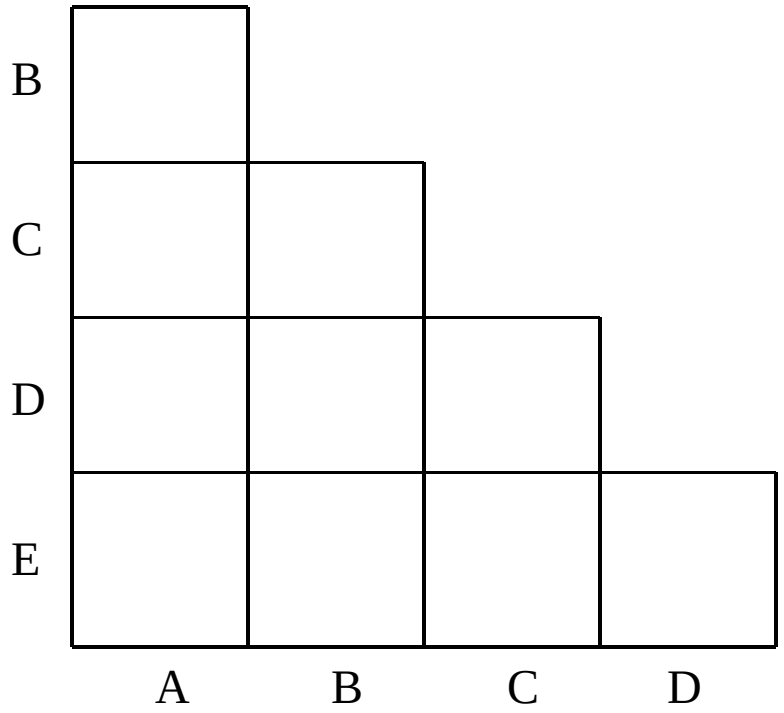
	X		
S	0	1	Y
A	D	B	0
B	C	D	1
C	A	B	1
D	C	B	1
	S(t+1)		

State Minimization - Example 3

	X			
S	0	1	Y	Z
A	D	B	0	0
B	C	D	1	0
C	A	B	1	0
D	C	D	1	1
	S(t+1)			

State Minimization - Example 4

	X		
S	0	1	Y
A	B	C	0
B	C	D	0
C	A	B	1
D	A	E	1
E	C	D	0
	S(t+1)		



Implication Chart

Each cell contains information: e.g. A is equivalent to B iff ((B = C) and (C = D)).

State Minimization - Example 4

S	X = 0	X = 1	Y
A	B	C	0
B	C	D	0
C	A	B	1
D	A	E	1
E	C	D	0

S(t+1)

Implication Chart

	A	B	C	D
B	B = C C = D			
C	≠	≠		
D	≠	≠	A = A B = E	
E	B = C C = D	C = C D = D	≠	≠

	A	B	C	D
B	B = C C = D			
C	≠	≠		
D	≠	≠	A = A B = B	
E	B = C C = D	=	≠	≠

S	X = 0	X = 1	Y
A	B	C	0
B	C	D	0
C	A	B	1
D	A	B	1

S(t+1)

Each cell contains information; e.g., A is equivalent to B iff ((B = C) and (C = D)).

State Minimization - Example 4

S	X=0	X=1	Y
A	B	C	0
B	C	D	0
C	A	B	1
D	A	B	1

S(t+1)

	A	B	C	D
B	B = C C = D			
C	≠	≠		
D	≠	≠	A = A B = B	
E	B = C C = D	=	≠	≠

S	X=0	X=1	Y
A	B	C	0
B	C	C	0
C	A	B	1

S(t+1)

	A	B	C	D
B	B = C C = C			
C	≠	≠		
D	≠	≠	=	
E	B = C C = C	=	≠	≠

State Minimization - Example 5

	X		
S	0	1	Y
A	A	B	1
B	A	C	0
C	A	B	0
	S(t+1)		

State Minimization - Example 6

	X		
S	0	1	Y
A	A	D	0
B	B	D	0
C	C	E	0
D	F	G	1
E	G	F	1
F	D	B	1
G	D	C	1
	S(t+1)		

Mealy machines

- Mealy machine: a finite-state machine whose output is determined by a combination of both its current state AND its current input
  - George Mealy (1955)
- Moore machine: a finite-state machine whose output is determined by ONLY its current state
  - Edward Moore (1956)

- Mealy and Moore machines solve *similar* but *different* problems. That is, the timing behavior of the two types of devices are not identical

State Assignment

- Example 2: Given 5 states: A, B, C, D, E and minimum # of bits (3), How many possible state assignments are there?
  - Assume that the initial state is 'A'
  - Assume that we will use the assignment 000 for the initial state

Minimal Risk/Cost State Assignments

- If extra unused state codes exist (number of available states $2^n > s$), then two choices are possible:

- **Minimal Risk** = most reliable but most expensive
  - Assume it is possible to enter unused state by noise, weird inputs, etc.
  - So include all unused states as present states, but all corresponding next states go to "initial" or "idle"

- **Minimal Cost** = somewhat risky but least expensive
  - Assume unused states NEVER entered accidentally.
  - So the next state and outputs of all unused states = "don't care" to reduce next state and output logic

- Achieving both minimal risk and minimal cost is often possible!

Example: State Assignment Strategies

Alternative Assignments

$Q_3..Q_0$	$Q_4..Q_0$	$Q_2Q_1Q_0$	$Q_2Q_1Q_0$
0000	00001	000	000
0001	00010	000	001
0010	00100	010	010
0100	01000	100	011
1000	10000	101	100
Almost One Hot	**One Hot**	**Decomposed**	**Simplest**

AB

S	00	01	11	10	Z
INIT	A0	A0	A1	A1	0
A0	OK0	OK0	A1	A1	0
A1	A0	A0	OK1	OK1	0
OK0	OK0	OK0	OK1	A1	1
OK1	A0	OK0	OK1	OK1	1

S*

Example decomposition:

- Initial State = all 0's for easy RESET
- Make Q2 follow Z?
- Use simple assignment for remaining state variables
- Make CERTAIN each assignment is used only once**!**

THUS, simpler next state and output logic!

State Assignment Strategies

- Simplest Assignment
  - Straight binary, NOT best; purely arbitrary assignment
- One Hot Assignment
  - Redundant encoding, each flip-flop is assigned a state.
  - Uses the same number of bits as there are states (not useful in large designs)
  - Simple to assign; simple next state logic (no state decoding required)
  - Output logic is simple! One OR gate per Moore output!
- Almost One Hot Assignment
  - Almost same as One Hot, but one less state bit
  - Use all 0's to represent a state (usually INIT)
  - Must now decode state 0 if it is needed
- Decomposed Assignment
  - Use "structure" of the state table to simplify next-state/output logic

State Assignment - Heuristic Guidelines

Starting from the highest priority to the lowest:

- Choose initial coded state that's easy to produce at reset: (all 0's or 1's)
  - This simplifies the initialization circuitry, but is not always wise.
- Freely use any of the 2^n state codes for best assignment

  (i.e.. with s states, don't just use the first s integers 0,1,…,s-1)
- Define specific bits or fields that have meaning with respect to input or output variables (**decomposed codes**)
- Consider using more than minimum number of state variables to allow for decomposed codes
- Minimize number of state variables that change at each transition
  - Prioritized adjacency schemes are often used in the field
- Simplify output logic

State Machine Design Procedure

Most difficult and creative

✓1. Build state/output table (or state diagram) from word description

✓2. Minimize number of states

✓3. Choose state variables and assign bit combinations to named states

Well-defined procedure - can be automated

4. Build transition/output table from state/output table (or state diagram)

5. Choose flip-flop type (D, J-K, etc.)

6. Build excitation table for flip-flop inputs from transition table

7. Derive excitation equations from excitation table

8. Derive output equations from transition/output table

9. Draw logic diagram with excitation logic, output logic, and state memory elements

Ad hoc design

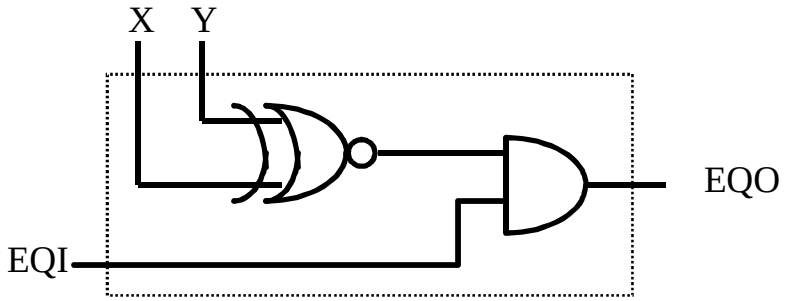
Combinational ad-hoc design

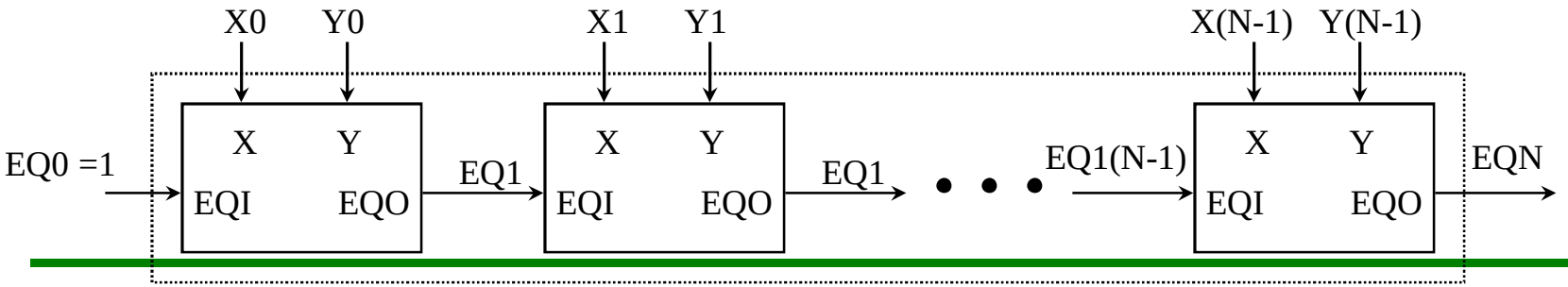
Sequential ad-hoc design

Iterative Combinational Logic

- Iterative logic array: A device consisting of identical sub-circuits connected together in a chain to perform a larger overall function
- Iterative Comparator : cascaded 1-bit comparators
- 1-bit comparator :

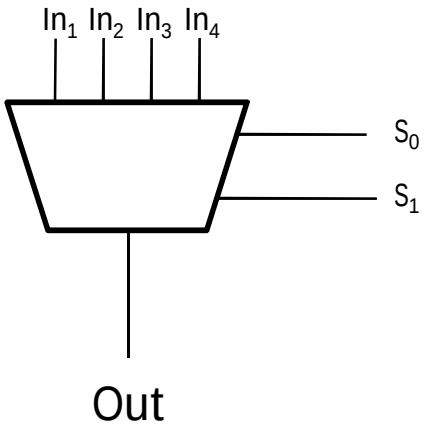
Function Table			
EQI	X	Y	EQO
0	x	x	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



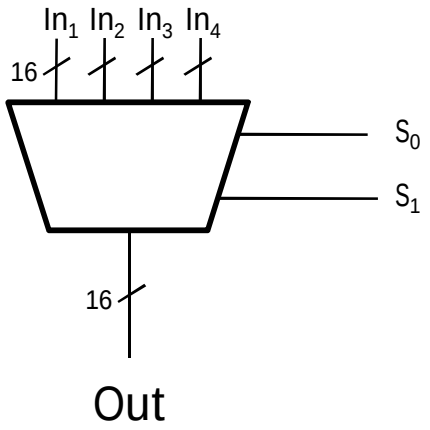


Circuits: a Multiplexor

A four-input MUX



With 16 of these, we can build a 16-bit four input MUX



Ad-hoc maximum value selector

Ad Hoc Design

- Directly translate word description into a circuit
  - Use Flip-Flops to “remember” things
  - Use combinational logic to decode these remembered things
  - Usually simplifies the state assignment process because storage elements remember something meaningful.
- Useful only for small state machines
- May be easier or harder to design
- May require more or less complex hardware
- With an Ad Hoc design, the ease of design and complexity are unknown before the design is attempted
- Larger designs are often decomposed into smaller more easily solved problems and then recombined ad-hoc!
  - e.g.: Control and Data units

Example Ad Hoc Sequential problem

Design a state machine with inputs A, B and output Z. Z=1 if A and B inputs were EQUAL for the last 2 clock ticks OR if B has been 1 ever since the first condition was true. Else, Z=0.

Solve the two conditions separately, then OR the results:

a) 1st Condition - Serial Comparator. Remember last 2 comparison results, then AND together. --> SAMEBOTH output

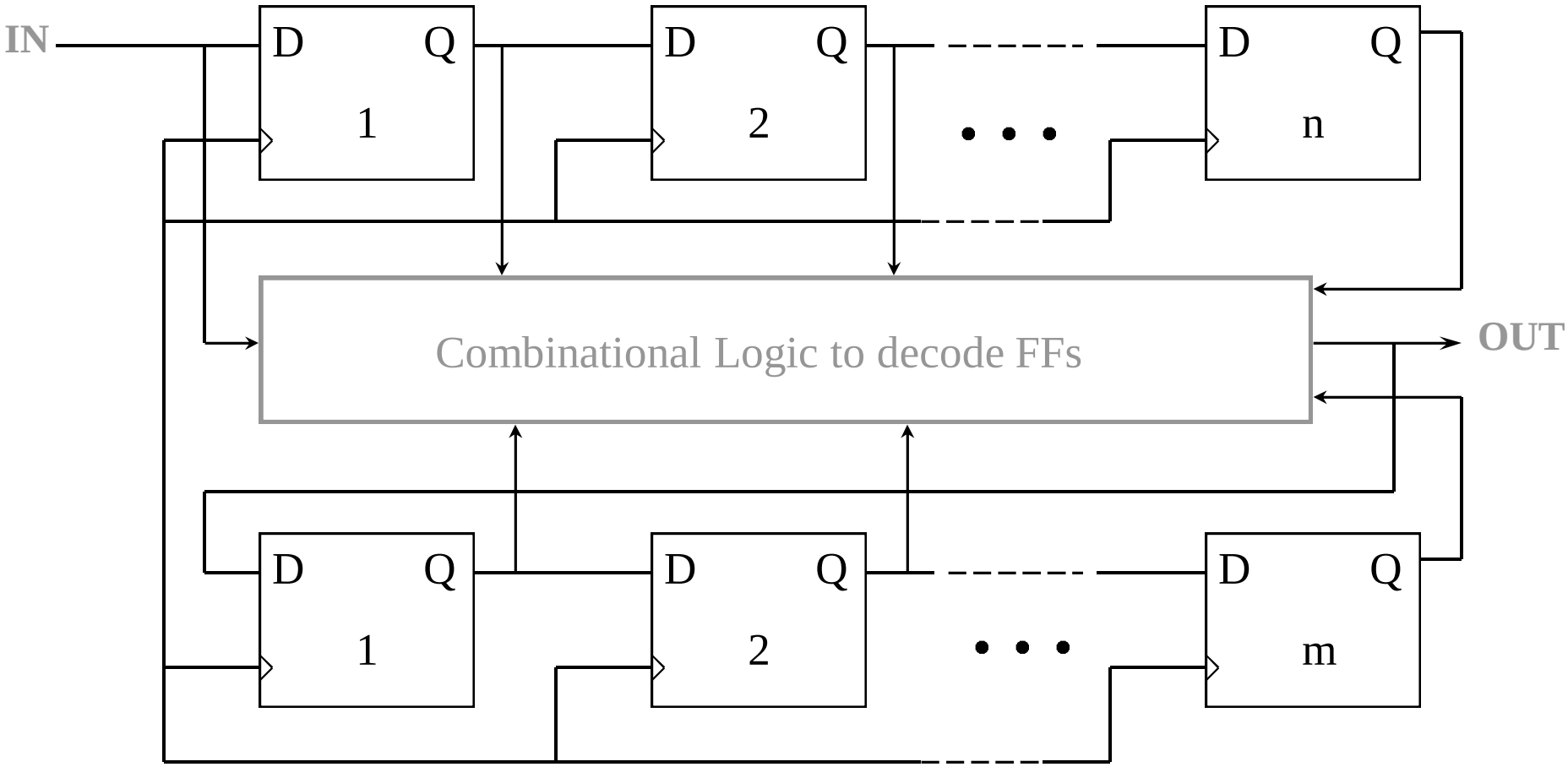
b) 2nd Condition - Remember if B has been 1 since 1st condition true
--> BOK output

Finite Memory Machine

- Finite Memory Machines (FMMs) are one type of Ad Hoc design
- Outputs are determined by:
  - Current inputs
  - and n previous inputs
  - and m previous outputs
- FMMs are a subclass of Finite State Machines, but FSMs are not necessarily FMMs
  - an FSM may depend on all past inputs (forever)
  - a FMM can only depend on a finite number of past inputs

Finite Memory Machine

n flip-flops store previous inputs

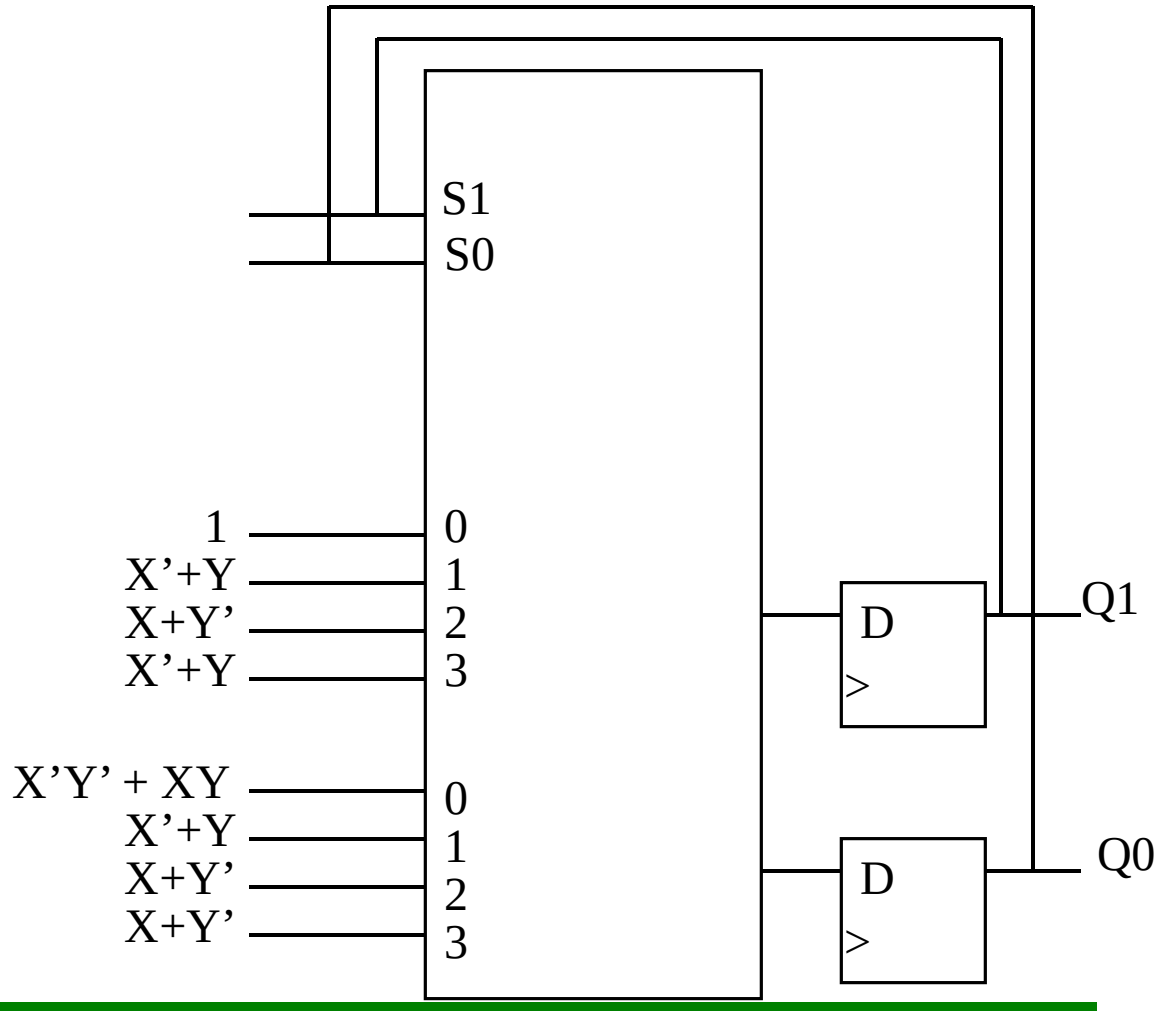


n flip-flops store previous outputs

What if multiple I/O bits are needed?

Prototype Implementation with MUX

Q1	Q0	X	Y	Q1*	Q0*
0	0	0	0	1	1
		0	1	1	0
		1	0	1	0
		1	1	1	1
0	1	0	0	1	1
		0	1	1	1
		1	0	0	0
		1	1	1	1
1	0	0	0	1	1
		0	1	0	0
		1	0	1	1
		1	1	1	1
1	1	0	0	1	1
		0	1	1	0
		1	0	0	1
		1	1	1	1



Computer Arithmetic

Standards all the use of elements for ad-hoc use!

Addition

Subtraction

Twos-complement

Overflow

Devices for Combinational Arithmetic: Adders, ALUs

Combinational/Sequential bit-serial operations

Unsigned Binary Integers

$$Y = \text{“abc”} = a.2^2 + b.2^1 + c.2^0$$

(where the digits a, b, c can each take on the values of 0 or 1 only)

N = number of bits

Range is: $0 \leq i < 2^N - 1$

Umin = 0

Umax = $2^N - 1$

	3-bits	5-bits	8-bits
0	000	00000	00000000
1	001	00001	00000001
2	010	00010	00000010
3	011	00011	00000011
4	100	00100	00000100

Problem:

- How do we represent *negative* numbers?

Manipulating Binary numbers - 01

- Binary to Decimal conversion & vice-versa
  - A 4 bit binary number A = $a_3a_2a_1a_0$ corresponds to:

    $a_3 \times 2^3 + a_2 \times 2^2 + a_1 \times 2^1 + a_0 \times 2^0 = a_3 \times 8 + a_2 \times 4 + a_1 \times 2 + a_0 \times 1$

    (where a_i = 0 or 1 only)
  - A decimal number can be broken down by iteratively determining the highest power of two that "fits" in the number:

    e.g. $(13)_{10}$ =>

    e.g. $(63)_{10}$ =>

    e.g. $(0.75)_{10}$ =>
  - A binary number can be broken down by iteratively adding powers of two:

    e.g. $(00101100)_2$ =>

    e.g. $(10101100)_2$ =>

Binary Arithmetic: Addition

Carries:	0000	101100
Augend:	01100	10110
Addend:	+10001	+10111
	-------	-------
Sum	11101	101101

Binary Arithmetic: Subtraction

	0 0 0 0 0	Borrow
	1 0 1 1 0	Minuend
-	1 0 0 1 0	Subtrahend
	0 0 1 0 0	Difference

	0 0 1 1 0	Borrow
	1 0 1 1 0	Minuend
-	1 0 0 1 1	Subtrahend
	0 0 0 1 1	Difference

Binary Arithmetic: Multiplication

```
        1 0 1 1     Multiplicand
x         1 0 1     Multiplier
-------------------
        1 0 1 1
      0 0 0 0
    1 0 1 1
-------------------
  1 1 0 1 1 1     Product
```

Signed Magnitude

- Leading bit is the sign bit

$$Y = \text{“abc”} = (-1)^a (b.2^1 + c.2^0)$$

$$\text{Range is: } -2^{N-1} + 1 < i < 2^{N-1} - 1$$

$$Smin = -2^{N-1} + 1$$
$$Smax = 2^{N-1} - 1$$

Problems:

- How do we do addition/subtraction?
- We have two numbers for zero (+/-)!

-4	**10100**
-3	**10011**
-2	**10010**
-1	**10001**
-0	**10000**
+0	**00000**
+1	**00001**
+2	**00010**
+3	**00011**
+4	**00100**

Two's Complement

- Transformation
  - To transform a into -a, invert all bits in a and add 1 to the result

$$\text{Range is: } -2^{N-1} < i < 2^{N-1} - 1$$

$$\text{Tmin} = -2^{N-1}$$

$$\text{Tmax} = 2^{N-1} - 1$$

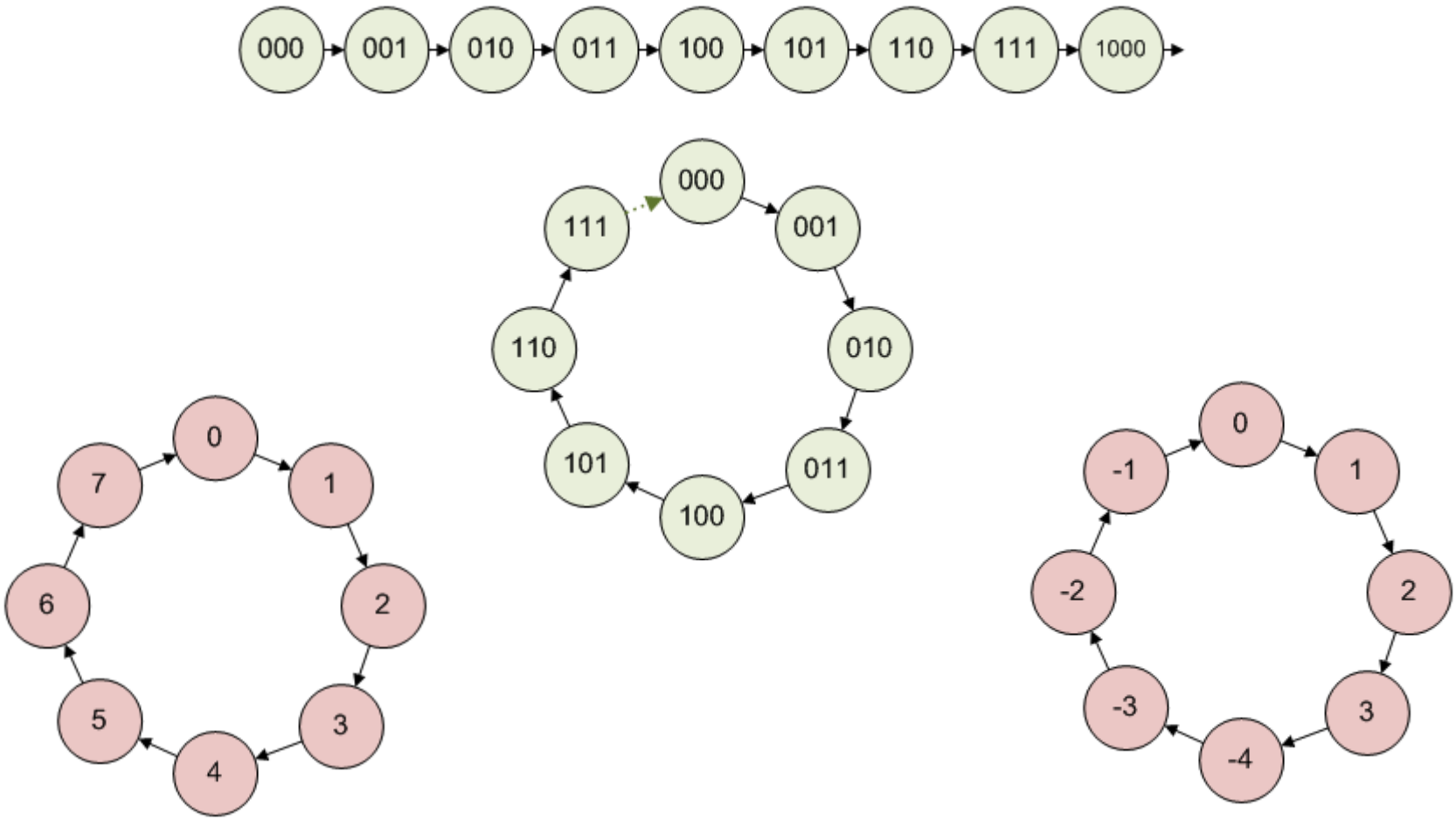
Advantages:

- Operations need not check the sign
- Only one representation for zero
- Efficient use of all the bits
- m + 2's complement n $\leftrightarrow$ m + $(2^n - n)$

  $\leftrightarrow$ m – n + 2^n (ignored carry) $\leftrightarrow$ m – n

-16	**10000**
…	**…**
-3	**11101**
-2	**11110**
-1	**11111**
0	**00000**
+1	**00001**
+2	**00010**
+3	**00011**
…	**…**
+15	**01111**

Binary Integer number lines/rings



Manipulating Binary numbers - 10

- SIGN EXTENTION
  - In the 2's complement representation, leading zeros do not affect the value of a positive binary number, and leading ones do not affect the value of a negative number. So:

    01101 = 00001101 = 13 and 11011 = 11111011 = -5

00001101	x0D
11111011	xFB
00001000 => 8 (as expected!)	x08

Manipulating Binary numbers - 11

- OVERFLOW
  - If we add the two (2's complement) 4 bit numbers representing 7 and 5 we get :

    0111 => +7
    0101 => +5
    1100 => -4 (in 4 bit 2's comp.)

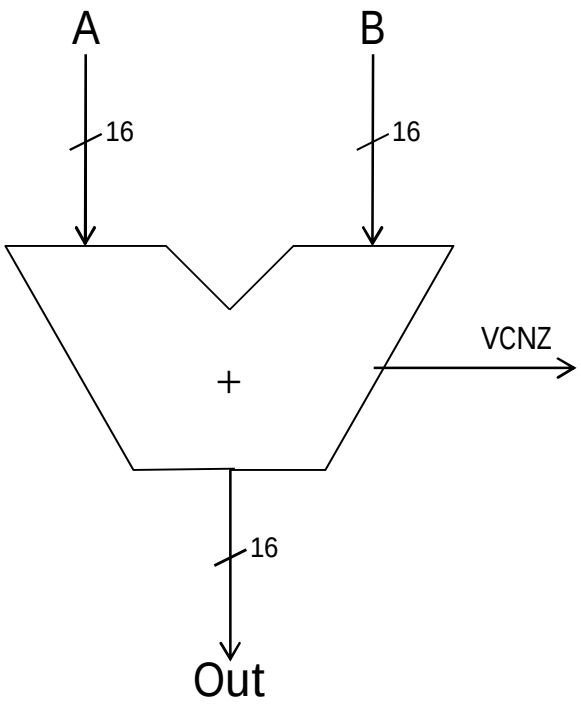
    - We get -4, not +12 as we would expect !!
    - We have *overflowed* the range of 4 bit 2's comp. (-8 to +7), so the result is invalid.
    - Note that if we add 16 to this result we get back 16 - 4 = 12
      - this is like "stepping up" to 5 bit 2's complement representation
  - In general, if the sum of two positive numbers produces a negative result, or vice versa, an overflow has occurred, and the result is invalid in that representation.
  - The sign extension rules help us detect overflow. If the ignored bit is appropriate for sign extension, then no information is lost.

Binary Arithmetic: Subtraction

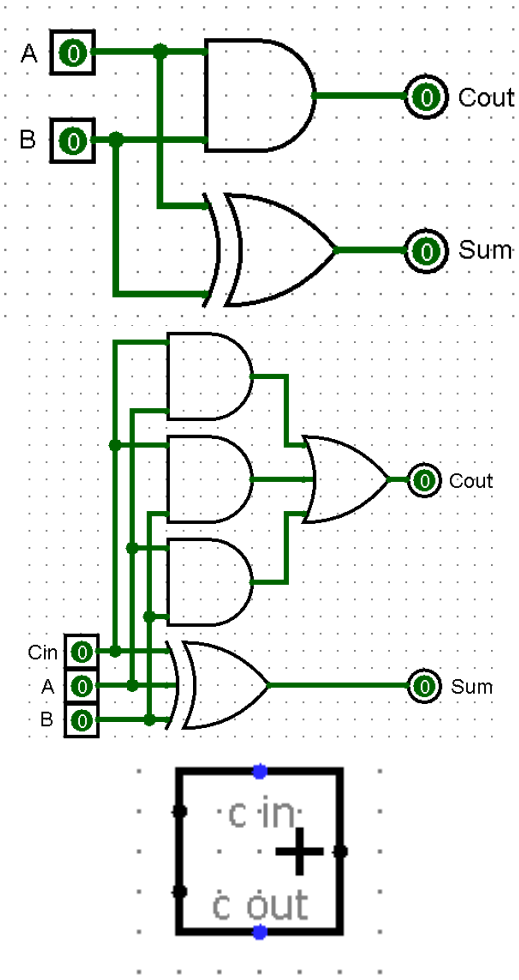
Carries:	0000	101100
Augend:	01100	10110
Addend:	+10001	+10111
	-------	-------
Sum	11101	101101

- two-'s complement subtraction:
  - Ignore carry/overflow
  - $m - n \leftrightarrow m - n + 2^n$ (ignored carry) $\leftrightarrow m + (2^n - n) \leftrightarrow m$ + 2's complement n
  - The easiest way to add 2^n?
    - Flipping all the bits is the same as adding $2^n - 1$ (the one's complement)
    - So then we add one

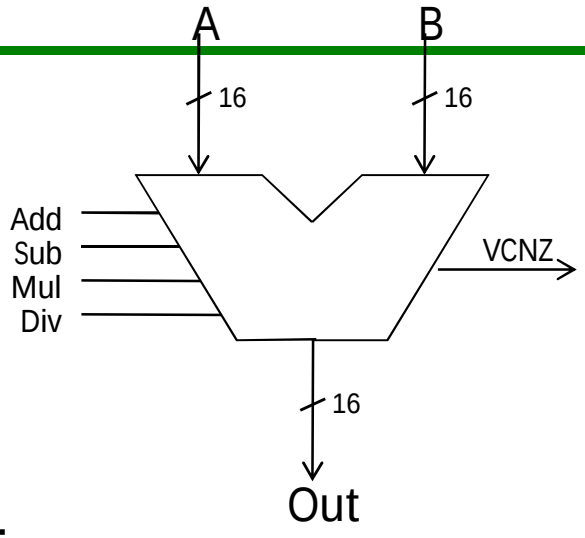
Circuits: an adder

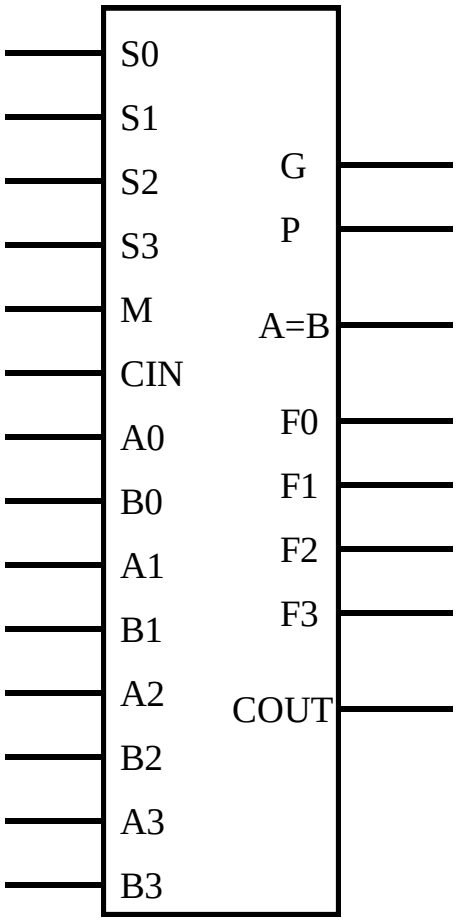


- Half-Adder
  - Two data inputs
  - One data output
  - One carry output
- Full-Adder
  - Two data inputs
  - One Carry input
  - One data output
  - One Carry Output
- Optional Status Outputs
  - V: Overflow
  - C: Carry
  - N: Negative
  - Z: Zero



MSI Arithmetic Logic Units (ALU)



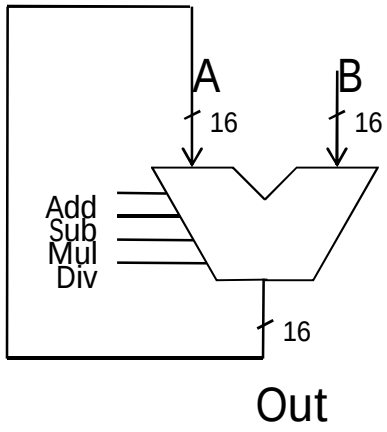


- Example :

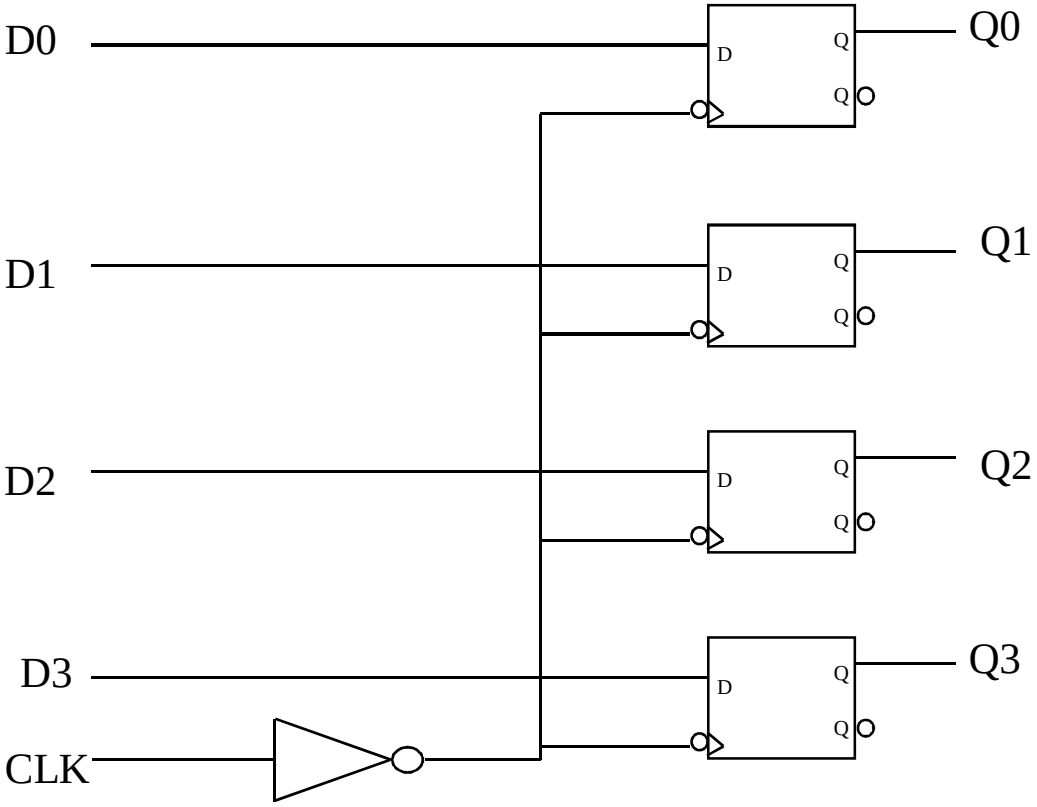
Inputs				Functions	
S3	S2	S1	S0	M=0	M=1
0	0	0	0	F= A-1+CIN	F=A’
0	1	1	0	F= A-B-1+CIN	F=A XOR B’
1	0	0	1	F= A+B+CIN	F=A XOR B
1	0	1	1	F=(A OR B)+ CIN	F=A+B
1	1	0	0	F= A+A+CIN	F= 0000
1	1	1	1	F=A+CIN	F=A

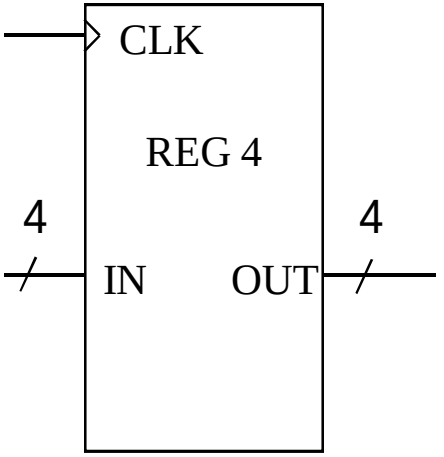
Keeping track of state

- Suppose we perform an addition (A + B) on our ALU.
- Where can we “put” the result?



4-bit (Quad) Register





If

R0 is 0011 (x3)

Then

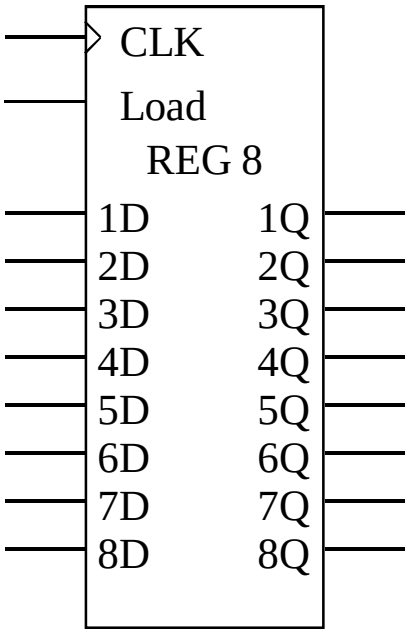
R0[3:0] is 0011

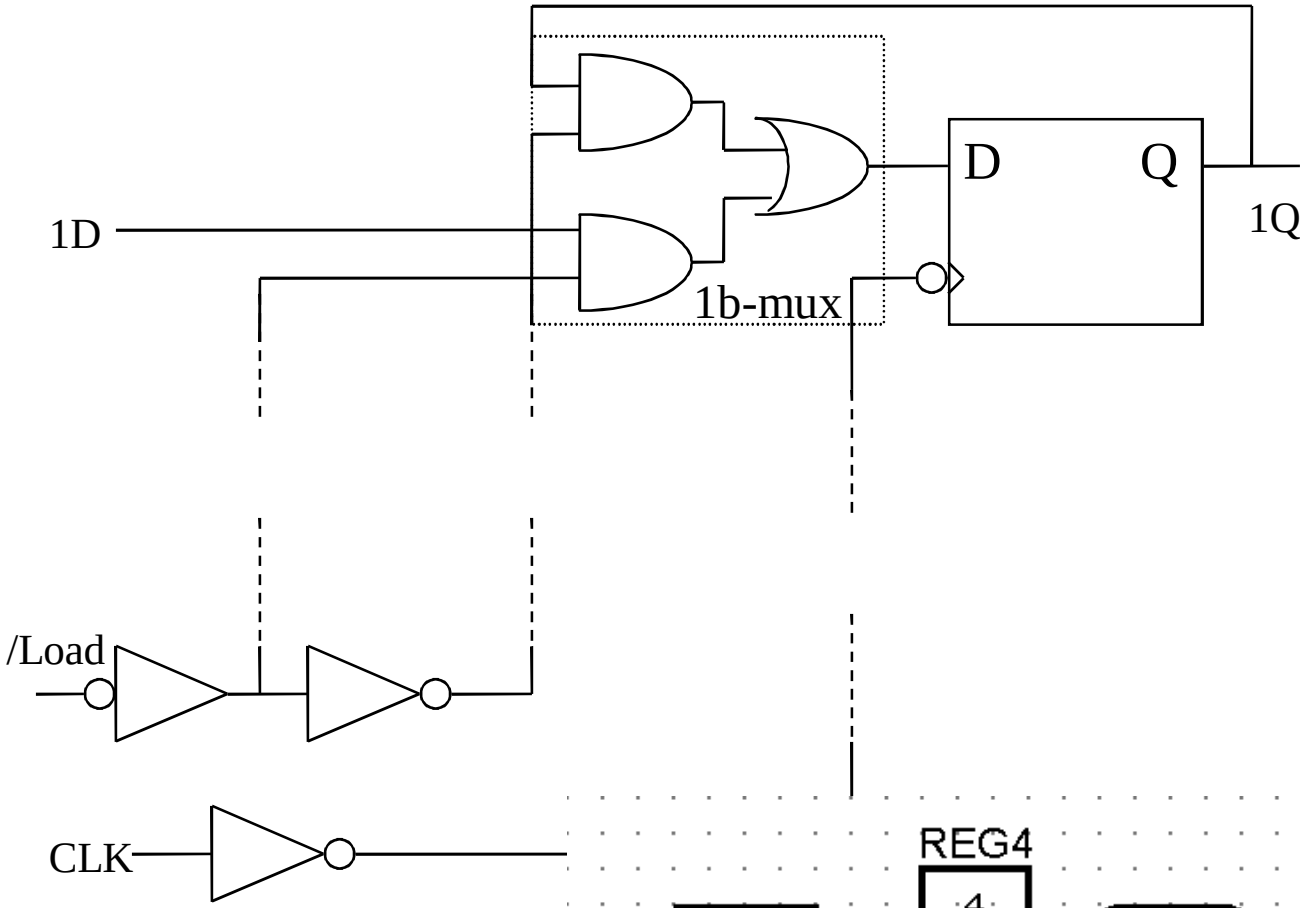
R0[2:0] is 011

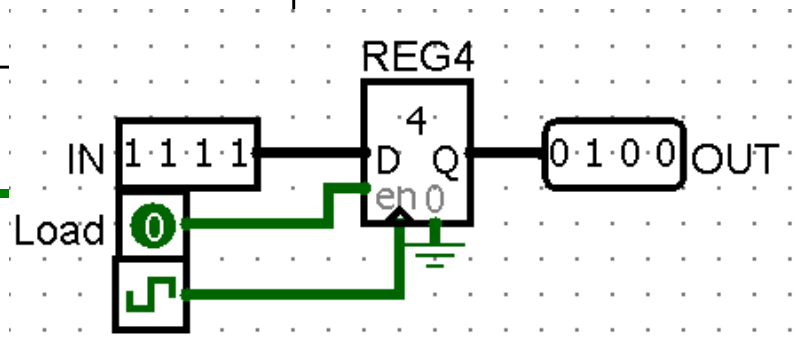
R0[2:1] is 01

R0[0:0] is 1

Register with Parallel Load Enable

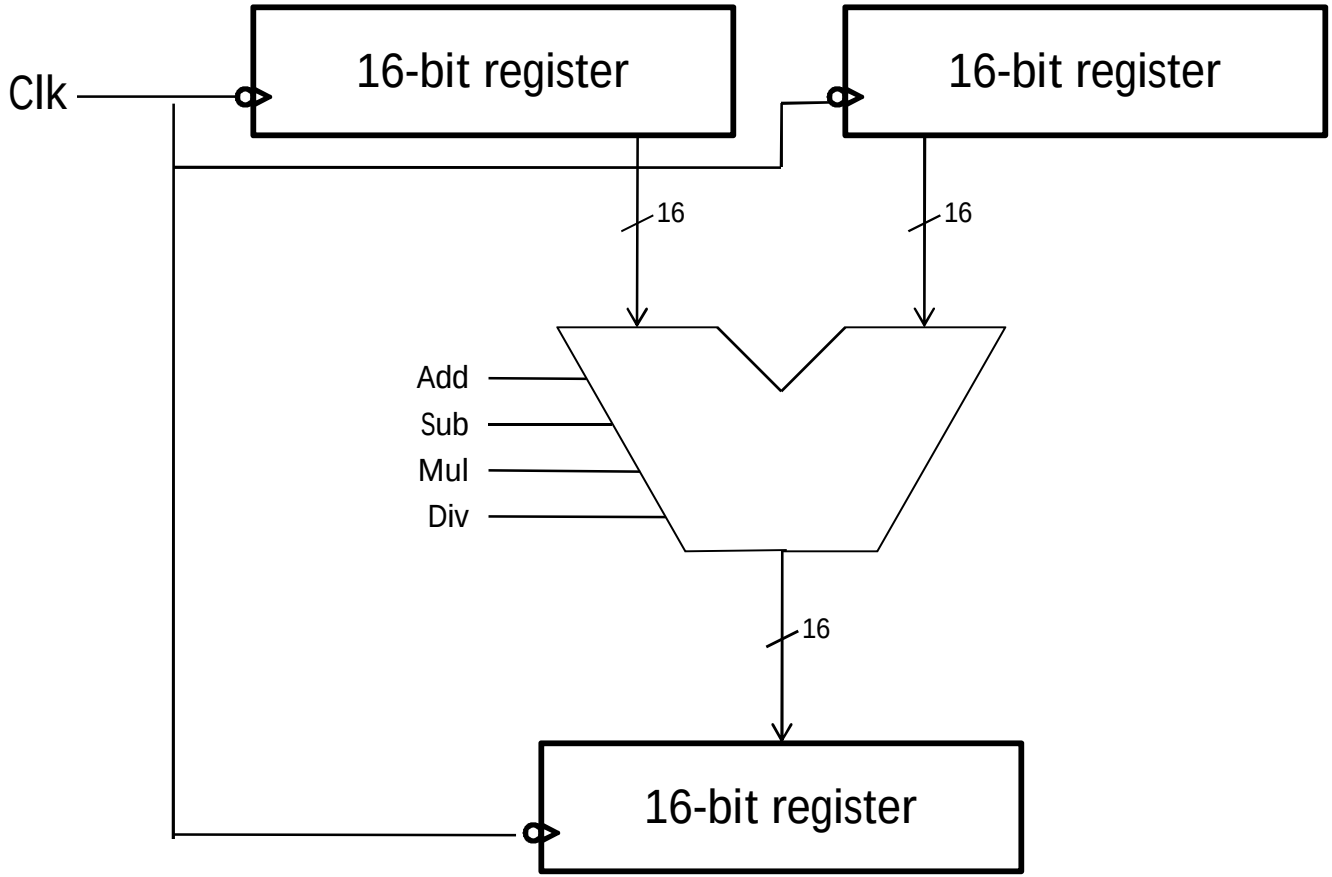




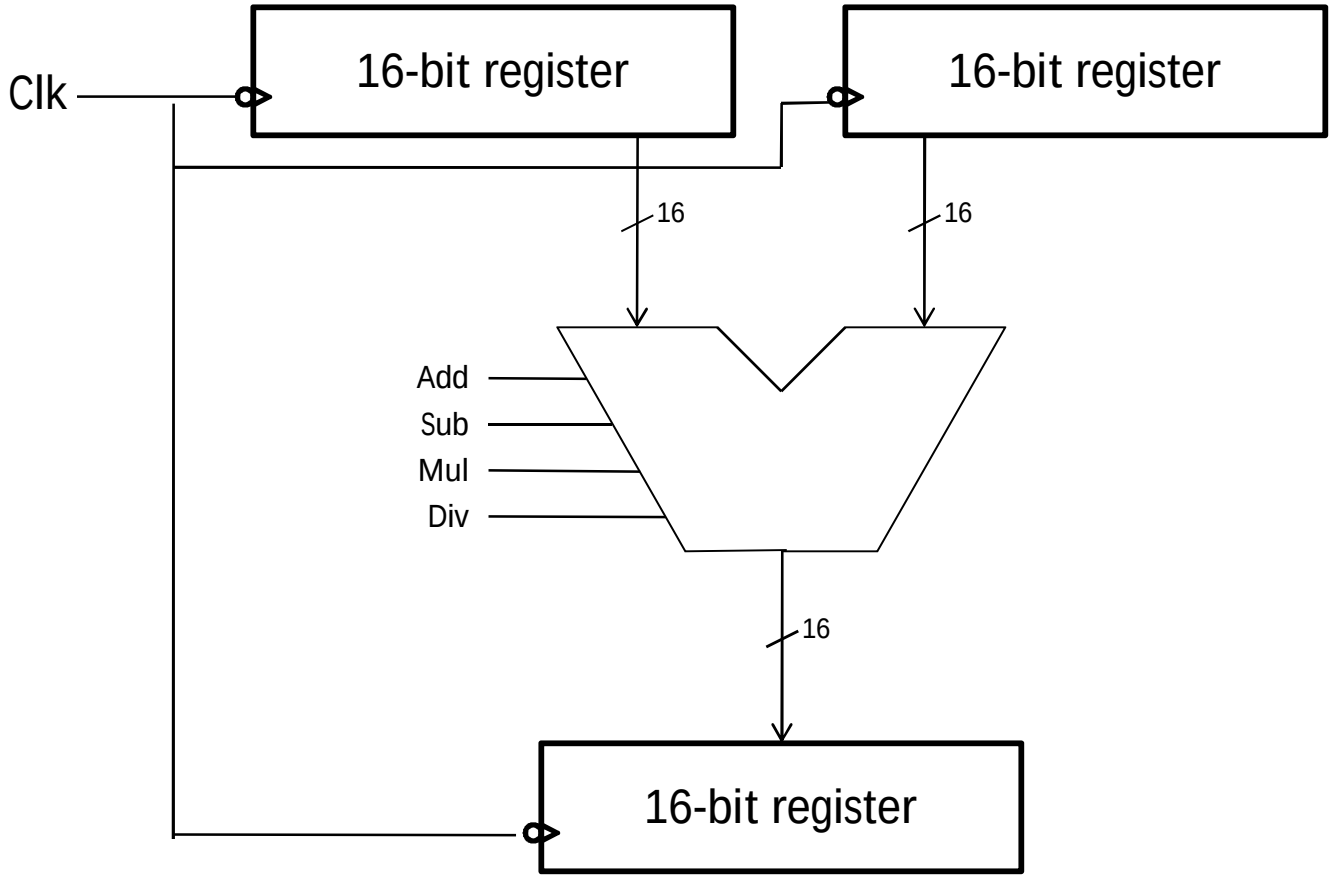


Why don't we gate the clock?

Computation + storage

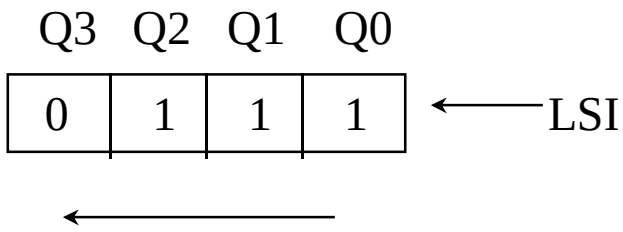


Bit serial operations



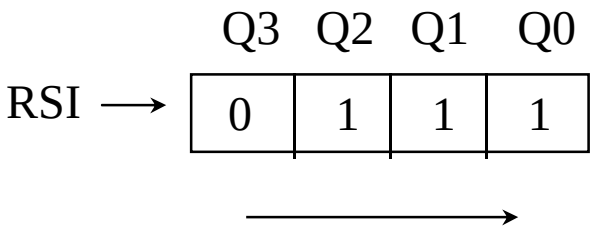
Shifting

- Moving moves data "sideways" left/right
  - Shift Left (or Shift Down) is towards MSB



Q3	Q2	Q1	Q0
1	1	1	LSI

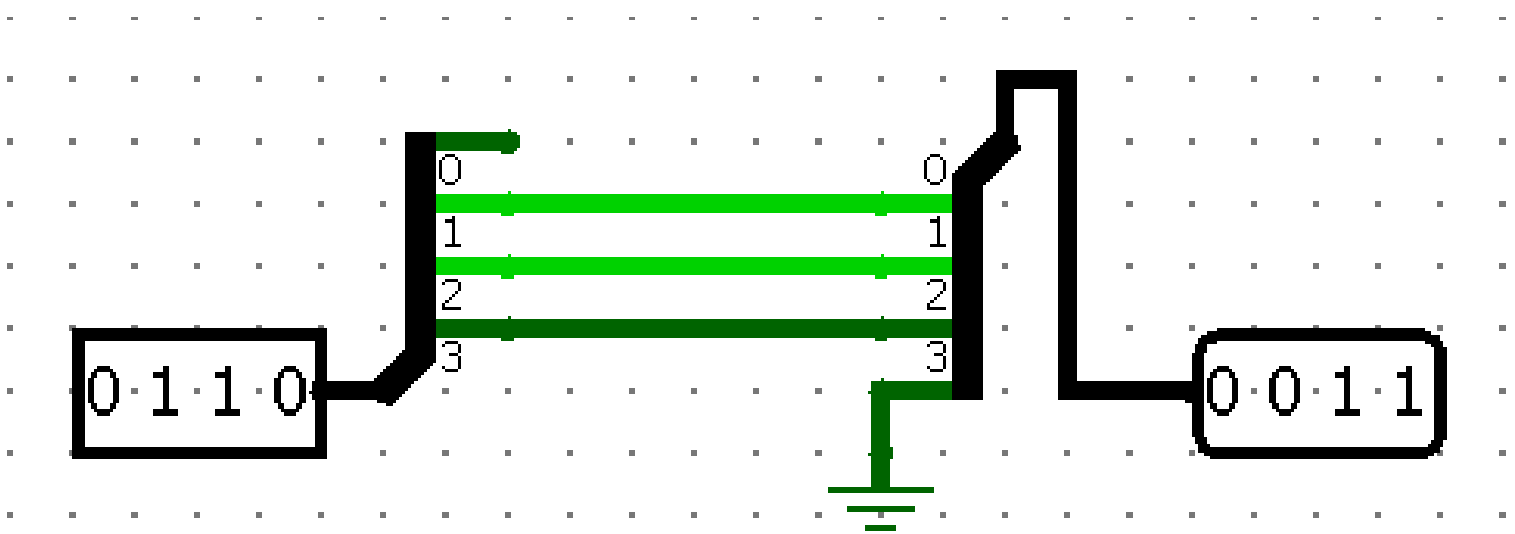
  - Shift Right (or Shift Up) is towards LSB



Q3	Q2	Q1	Q0
RSI	0	1	1

⇨ Often used to rearrange bits or Multiply/Divide by 2

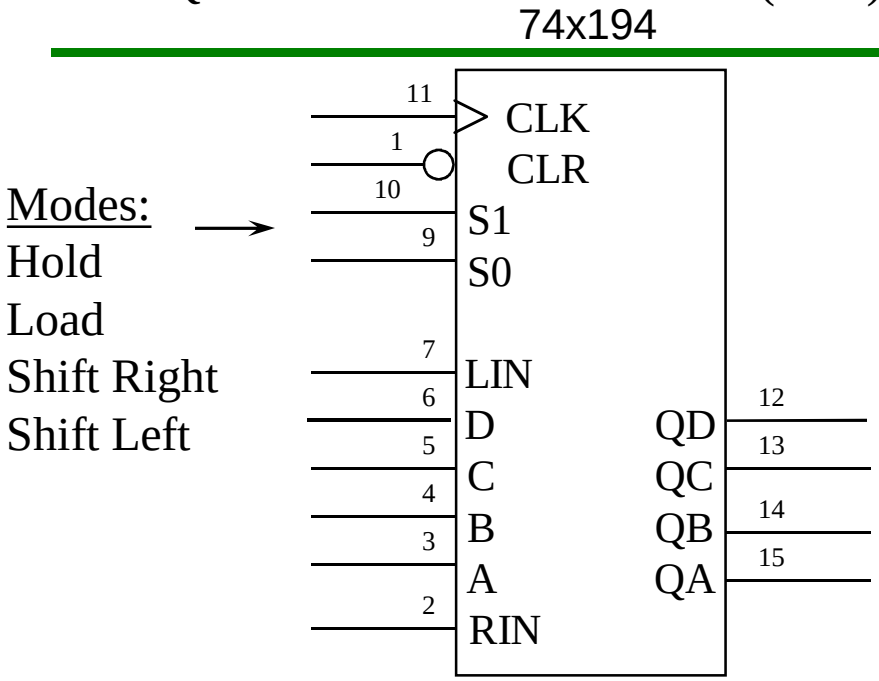
Combinational Barrel Shifter



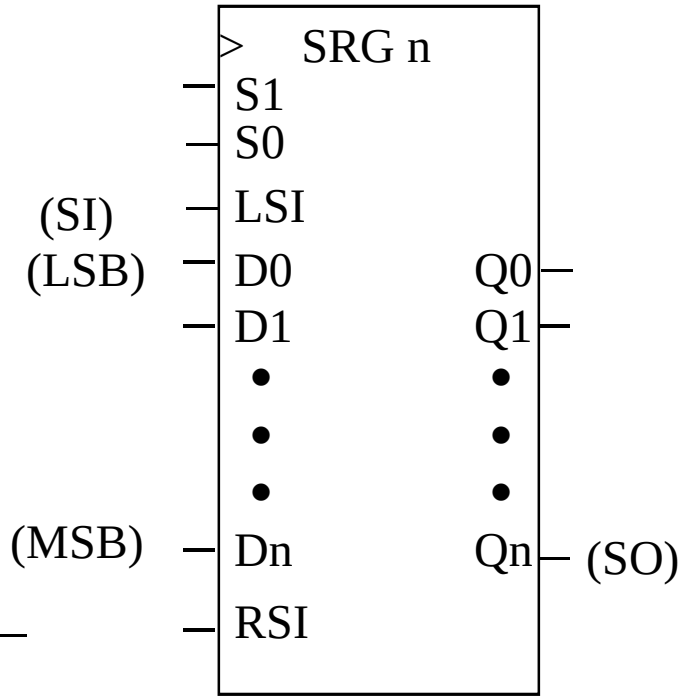
Bi-directional Universal Shift Registers

Quad Bi-directional Universal (4-bit) PIPO

74x194

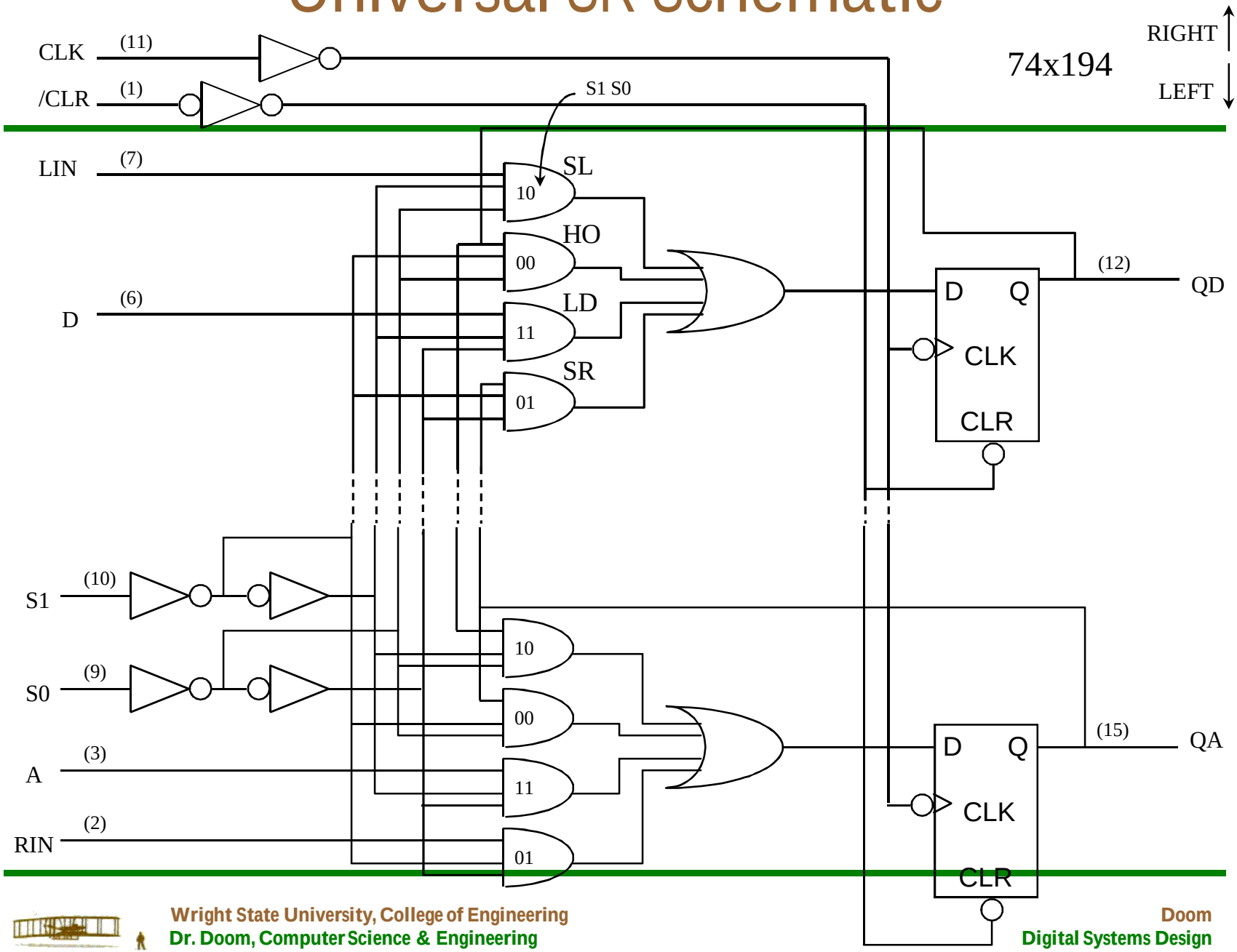


Block symbol

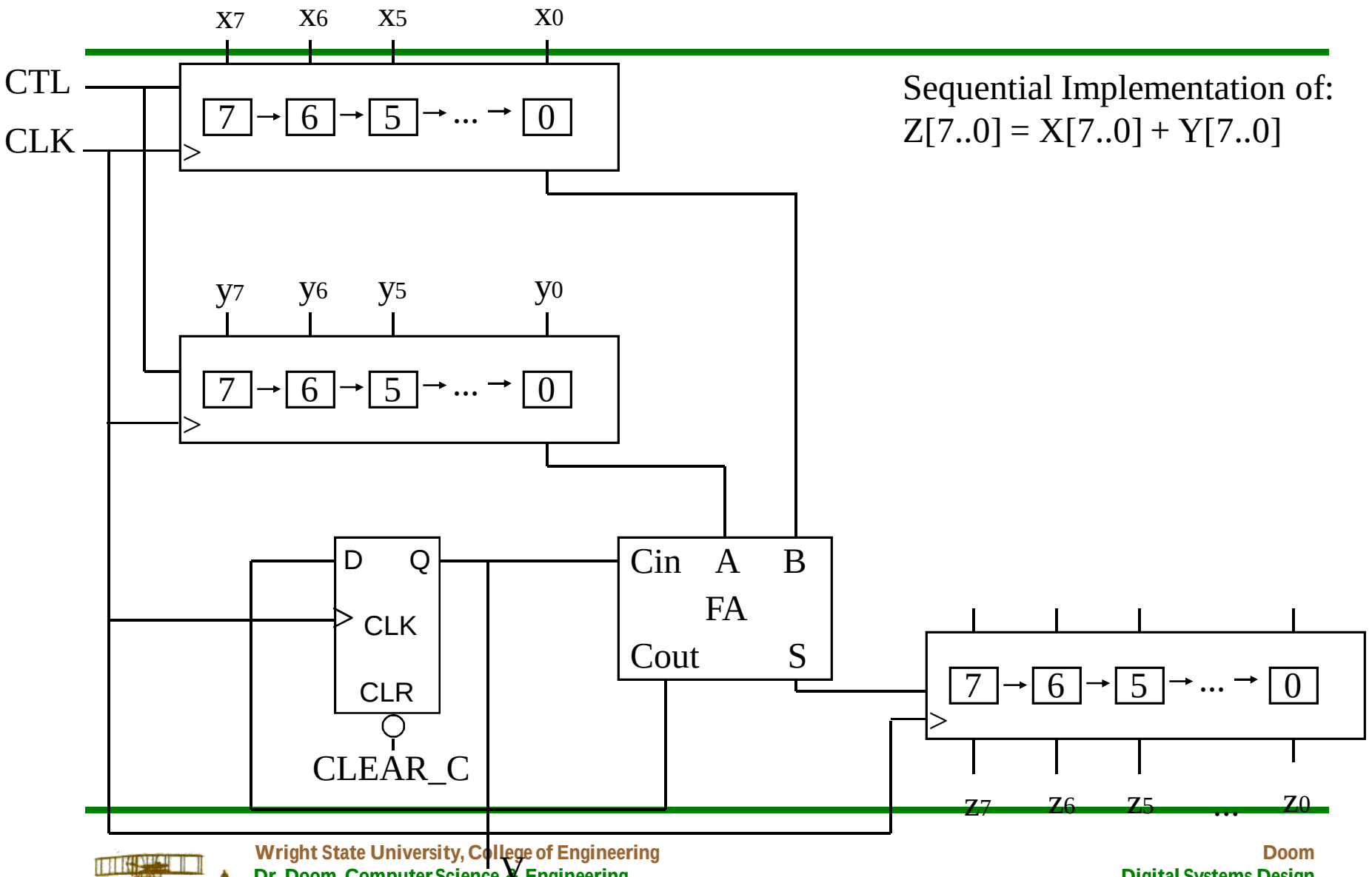


Function	Mode S1	Mode S0	Next state QA*	QB*	QC*	QD*
Hold	0	0	QA	QB	QC	QD
Shift right/up	0	1	RIN	QA	QB	QC
Shift left/down	1	0	QB	QC	QD	LIN
Load	1	1	A	B	C	D

Universal SR Schematic



Shift Register Applications



Sequential Implementation of:
Z[7..0] = X[7..0] + Y[7..0]