18-100 Lecture 25: More Finite State Machines

James C. Hoe
Dept of ECE, CMU
April 23, 2015

Today's Goal: Let's make it a wrap!

Announcements: Exam 3 on April 30
Final Exam, Fri., May 8, 8:30~11:30, GHC 4401
HW 10 due on Tuesday

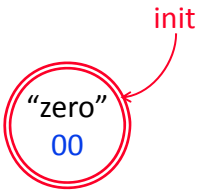
Handouts:

Another Example: Up-down Counter

- Input: 1-bit **(U)**p
  1-bit **(D)**own
- Output: 2-bit unsigned binary value: 0, 1, 2, or 3

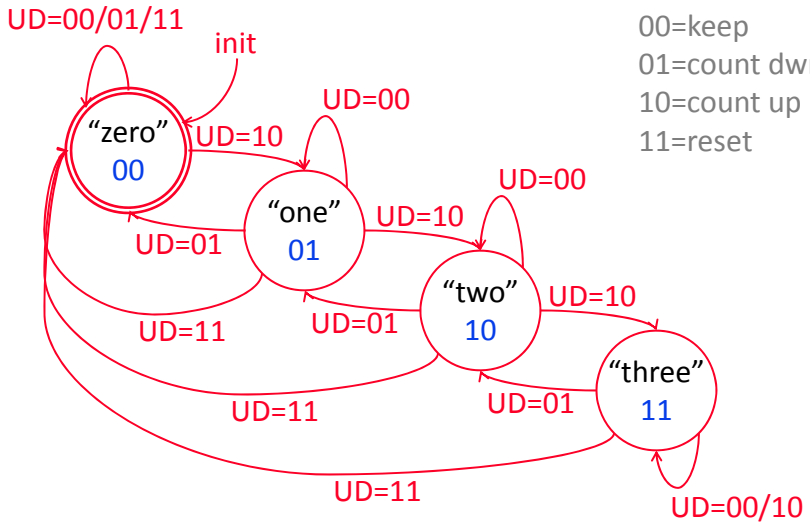
- What to do:
  - output should initialized to 0
  - if U and D are both deasserted, output should stay at the same value (after the next clock edge)
  - if U and D are both asserted, output should reset to 0
  - if only U is asserted, output should count up; if output is already 3, output stays at 3
  - if only D is asserted, output should count down; if output is already 0, output stays at 0

Up-down Counter



UD
00=keep
01=count dwn
10=count up
11=reset

Up-down Counter



UD
00=keep
01=count dwn
10=count up
11=reset

What state encoding would you choose?

My All-Time Favorite FSM

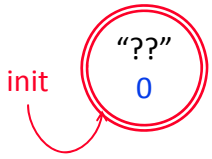
- Input: 1-bit **I**
- Output: 1-bit **O**

- What to do:
  - output O should be 1 each time after observing the sequence 1011 on input I
  - output O should be 0 at all other times

  - e.g.,

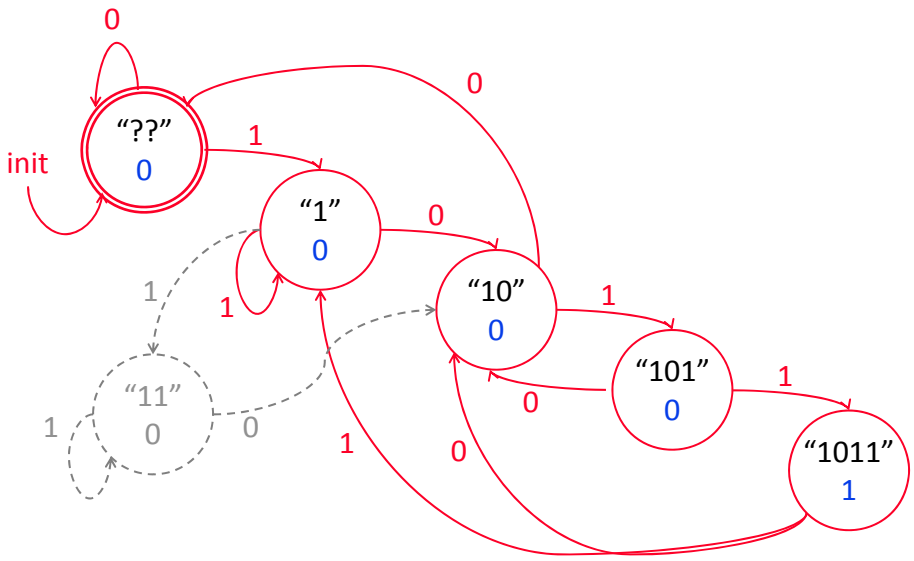
    I: 10101101001011011111 . . .
    O: 00000010000000100100 . . .

Looking for 1011



Picking good state names is very important

Looking for 1011



Picking good state names is very important

Next-State and Output Logic

State	Q_2	Q_1	Q_0	I	D_2	D_1	D_0
"??"	0	0	0	0			
				1			
"1"	0	0	1	0			
				1			
"10"	0	1	0	0			
				1			
"101"	0	1	1	0			
				1			
"1011"	1	0	0	0			
				1			

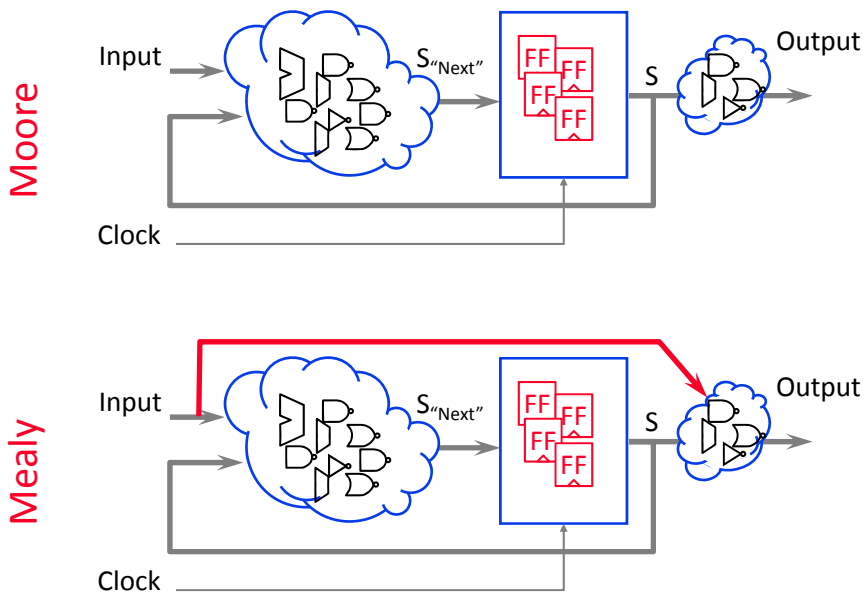
State	Q_2	Q_1	Q_0	O
"??"	0	0	0	
"1"	0	0	1	
"10"	0	1	0	
"101"	0	1	1	
"1011"	1	0	0	

Next-State and Output Logic

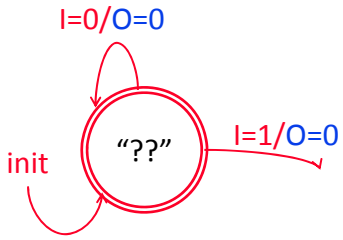
State	Q_2	Q_1	Q_0	I	D_2	D_1	D_0	
"??"	0	0	0	0	**0**	**0**	**0**	"??"
				1	**0**	**0**	**1**	"1"
"1"	0	0	1	0	**0**	**1**	**0**	"10"
				1	**0**	**0**	**1**	"1"
"10"	0	1	0	0	**0**	**0**	**0**	"??"
				1	**0**	**1**	**1**	"101"
"101"	0	1	1	0	**0**	**1**	**0**	"10"
				1	**1**	**0**	**0**	"1011"
"1011"	1	0	0	0	**0**	**1**	**0**	"10"
				1	**0**	**0**	**1**	"1"
	1	0	1	*	*	*	*	
	1	1	0	*	*	*	*	
	1	1	1	*	*	*	*	

State	Q_2	Q_1	Q_0	**O**
"??"	0	0	0	**0**
"1"	0	0	1	**0**
"10"	0	1	0	**0**
"101"	0	1	1	**0**
"1011"	1	0	0	**1**
	1	0	1	*
	1	1	0	*
	1	1	1	*

Mealy vs. Moore

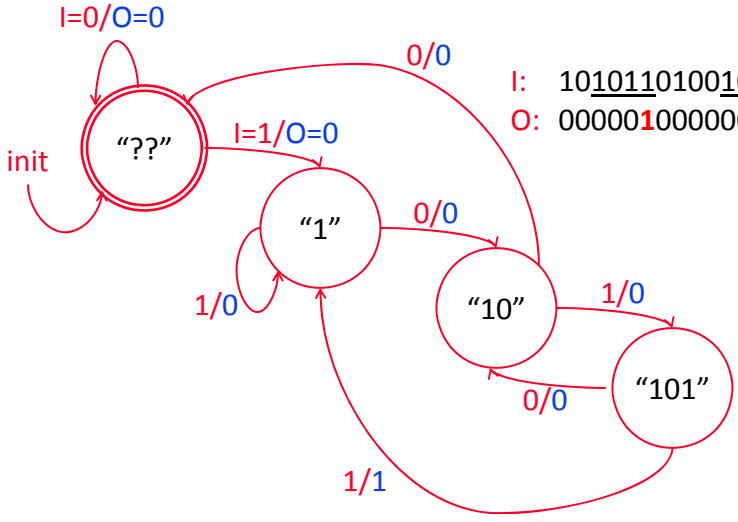


Looking for 1011 (Mealy)



The output in a given state also depends on current input

Looking for 1011 (Mealy)



The output in a given state also depends on current input

Mealy FSM Logic

next-state logic

State	Q_1	Q_0	I	D_1	D_0
"??"	0	0	0	**0**	**0**
			1	**0**	**1**
"1"	0	1	0	**1**	**0**
			1	**0**	**1**
"10"	1	0	0	**0**	**0**
			1	**1**	**1**
"101"	1	1	0	**1**	**0**
			1	**0**	**1**

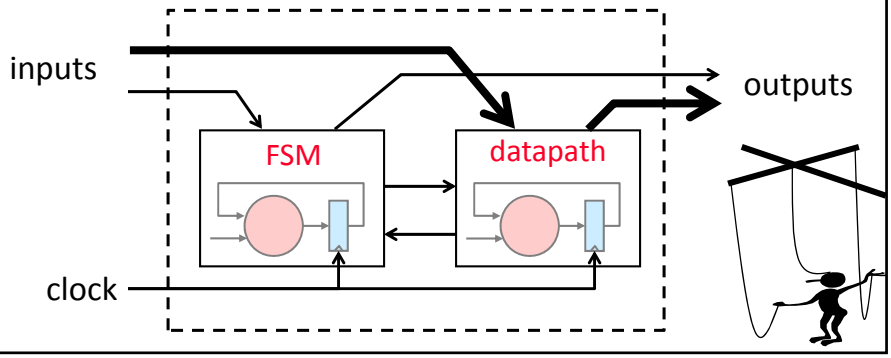
output logic

State	Q_1	Q_0	I	O
"??"	0	0	0	**0**
			1	**0**
"1"	0	1	0	**0**
			1	**0**
"10"	1	0	0	**0**
			1	**0**
"101"	1	1	0	**0**
			1	**1**

We are done! Time for some fun!

FSM-D

- datapath = combinational logic and registers to carry out computation (puppet)
- FSM = combinational logic and registers for control and sequencing (puppeteer)

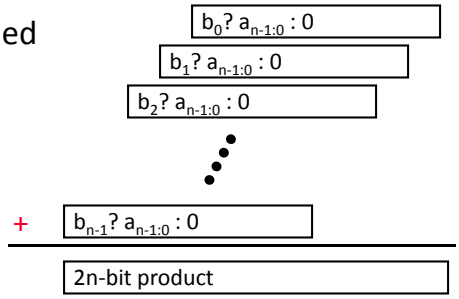


Multiply 2 n-bit numbers: a x b

- Given unsigned numbers $a_{n-1}a_{n-2}\ldots a_2a_1a_0$ and $b_{n-1}b_{n-2}\ldots b_2b_1b_0$

$$a \cdot b = \sum_{j=0}^{n-1}\left(2^j b_j\left(\sum_{i=0}^{n-1} 2^i a_i\right)\right)$$

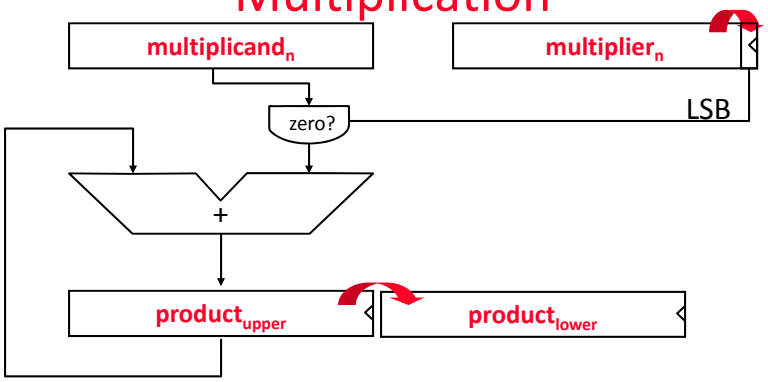
- Construct a full adder array where the summand ($a_{n-1:0} \times 2^i$) can be conditionally zero'ed according to b_i of $b_{n-1:0}$



- 2n bits are required to represent all possible products without overflow

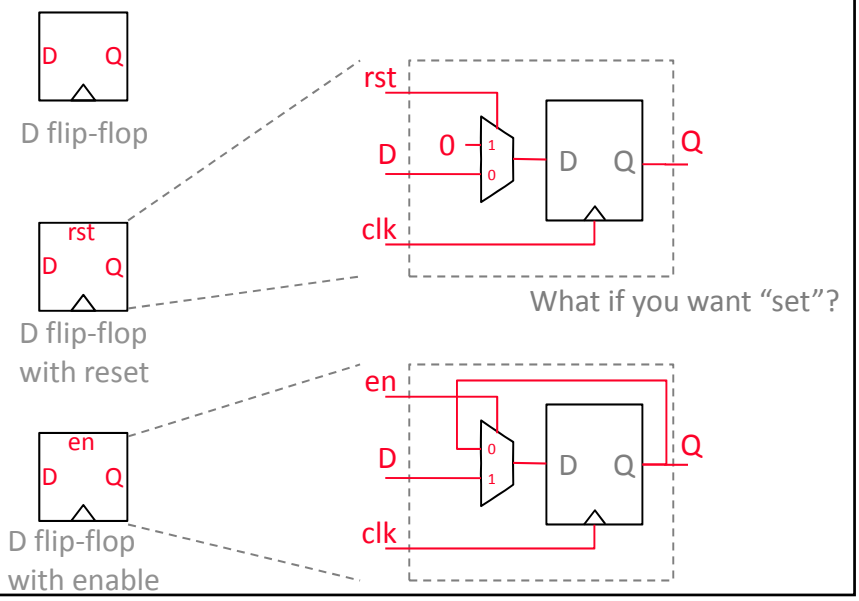
2's complement?

Iterative Shift-and-Add Multiplication

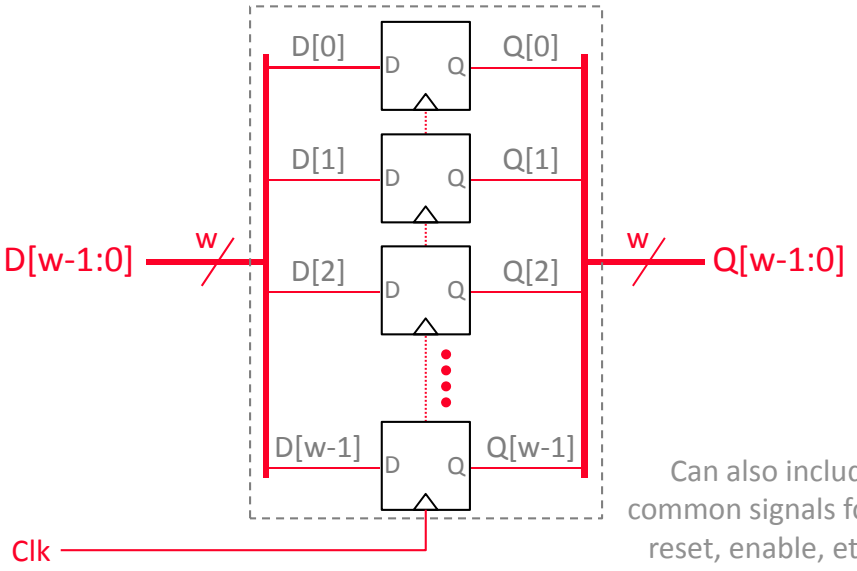


- To start, set multiplier/multiplicand and zero product
- Require n iterations of "shift-and-add"
  - add either 0 or multiplicand depending on LSB of multiplier
  - shift both multiplier and product

DFF Varieties

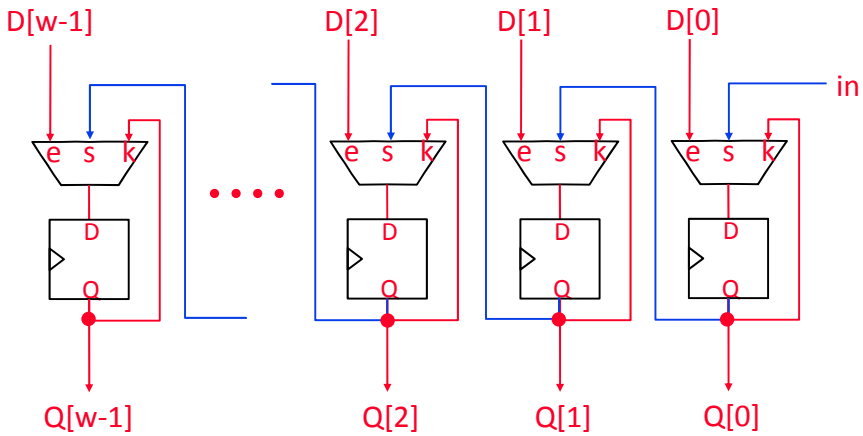


Register
(DFFs with common control)



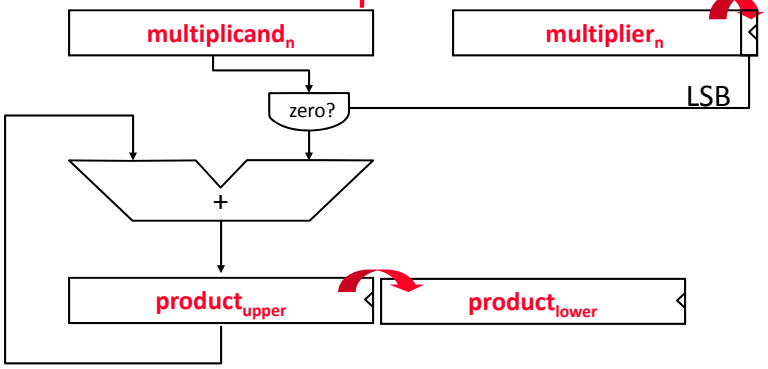
Can also include common signals for reset, enable, etc.

Shift Register



Set mux to **(e)**nable to latch new value
(s)hift to shift current value by 1 position
(k)eep to remember the current value

Iterative Shift-and-Add Multiplication

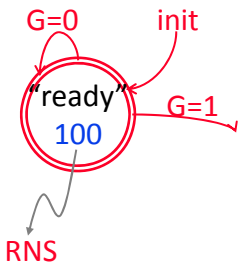


- To start, set multiplier/multiplicand and zero product
- Require n iterations of "shift-and-add"
  - add either 0 or multiplicand depending on LSB of multiplier
  - shift both multiplier and product

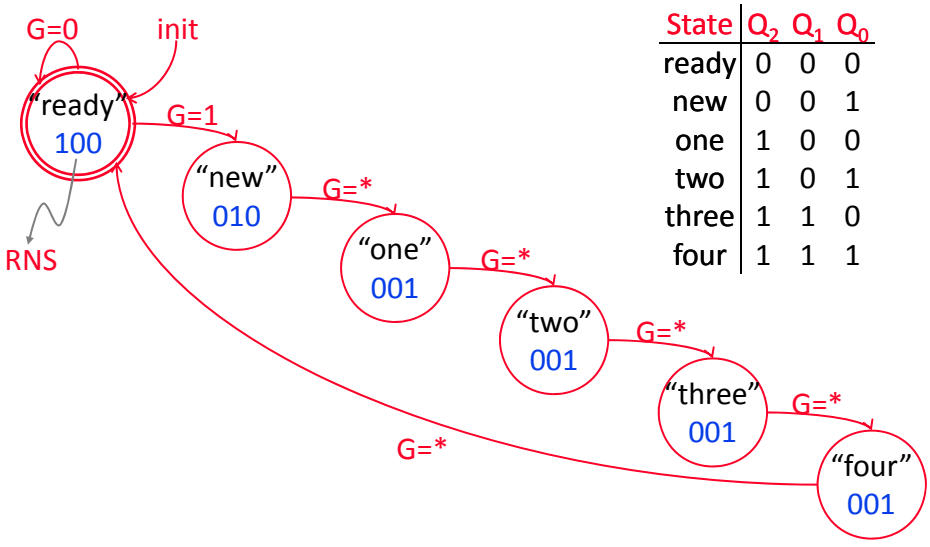
FSM Controller

- User input: 1-bit **(G)**o
- User output: 1-bit **(R)**eady
- Control output (to datapath):
  - 1-bit **(N)**ew: - latch new multiplier and multiplicand
    - zero the product register
  - 1-bit **(S)**hift: - shift the multiplier register
    - latch-and-shift the product register
- What to do
  - the user can assert G (for 1 clock edge) whenever R is asserted to begin a new multiplication
  - controller asserts N to latch new multiplicand and multiplier; R should be de-asserted until finished
  - controller should assert S for n cycles, after which
  - controller should assert R, de-assert S and wait for the next G trigger to start over

State Transition Diagram (say n=4)



State Transition Diagram (say n=4)



State	Q_2	Q_1	Q_0
ready	0	0	0
new	0	0	1
one	1	0	0
two	1	0	1
three	1	1	0
four	1	1	1

Next-State Logic

next-state logic

Q_2	Q_1	Q_0	G	D_2	D_1	D_0
0	0	0	0			
0	0	0	1			
0	0	1	0			
0	0	1	1			
0	1	0	0			
0	1	0	1			
0	1	1	0			
0	1	1	1			
1	0	0	0			
1	0	0	1			
1	0	1	0			
1	0	1	1			
1	1	0	0			
1	1	0	1			
1	1	1	0			
1	1	1	1			

D_0

G \ Q_1Q_0	00	01	11	10
0				
1				

Q_2

D_2

G \ Q_1Q_0	00	01	11	10
0				
1				

Q_2

D_1

G \ Q_1Q_0	00	01	11	10
0				
1				

Q_2

D_0=
D_1=
D_2=

Next-State Logic

next-state logic

Q_2	Q_1	Q_0	G	D_2	D_1	D_0
0	0	0	0	**0**	**0**	**0**
0	0	0	1	**0**	**0**	**1**
0	0	1	0	**1**	**0**	**0**
0	0	1	1	**1**	**0**	**0**
0	1	0	0	**x**	**x**	**x**
0	1	0	1	**x**	**x**	**x**
0	1	1	0	**x**	**x**	**x**
0	1	1	1	**x**	**x**	**x**
1	0	0	0	**1**	**0**	**1**
1	0	0	1	**1**	**0**	**1**
1	0	1	0	**1**	**1**	**0**
1	0	1	1	**1**	**1**	**0**
1	1	0	0	**1**	**1**	**1**
1	1	0	1	**1**	**1**	**1**
1	1	1	0	**0**	**0**	**0**
1	1	1	1	**0**	**0**	**0**

D_0

G \ Q_1Q_0	00	01	11	10
0	0	x	1	1
1	1	x	1	1
0	0	x	0	0
0	0	x	0	0

Q_2

D_2

G \ Q_1Q_0	00	01	11	10
0	0	x	1	1
0	0	x	1	1
1	1	x	0	1
1	1	x	0	1

Q_2

D_1

G \ Q_1Q_0	00	01	11	10
0	0	x	1	0
0	0	x	1	0
0	0	x	0	1
0	0	x	0	1

Q_2

$D_0=Q_2Q_0'+Q_0'G$
$D_1=Q_1Q_0'+Q_2Q_1'Q_0$
$D_2=Q_2Q_0'+Q_2Q_1'+Q_1'Q_0$

The choice of state encoding helped here

Output Logic

State	Q_2	Q_1	Q_0	R	N	S
ready	0	0	0	1	0	0
new	0	0	1	0	1	0
unused	0	1	0	x	x	x
unused	0	1	1	x	x	x
one	1	0	0	0	0	1
two	1	0	1	0	0	1
three	1	1	0	0	0	1
four	1	1	1	0	0	1

$R=Q_2'Q_0'$
$N=Q_2'Q_0$
$S=Q_2$

The choice of state encoding made a REALLY BIG difference here

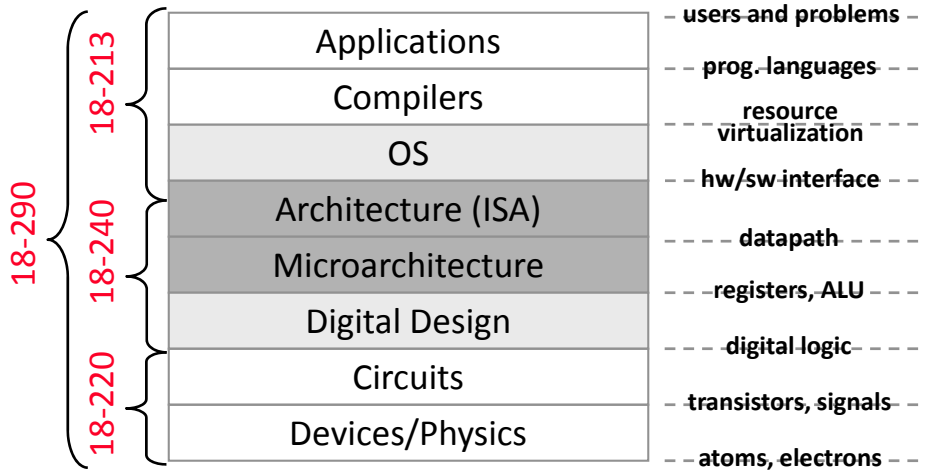
(Try repeating this example with a random state assignment on your own)

What we didn't talk about

- Cooperating Finite State Machines (CFSM)
  - many real designs have too many state bits to be designed as FSMs directly (e.g., a microprocessor)
  - must decompose by functionality and structure into manageable chunks (we saw FSM-D decomposition)
- Other timing disciplines, e.g.,
  - latch-based designs
  - dynamic logic
  - "asynchronous" logic
- Array memory structures and technologies
- Programmable logic devices

18-240 is waiting for you

Computer System Abstraction Layers



To use an abstraction properly you must understand the limits of the abstraction