Digital Design

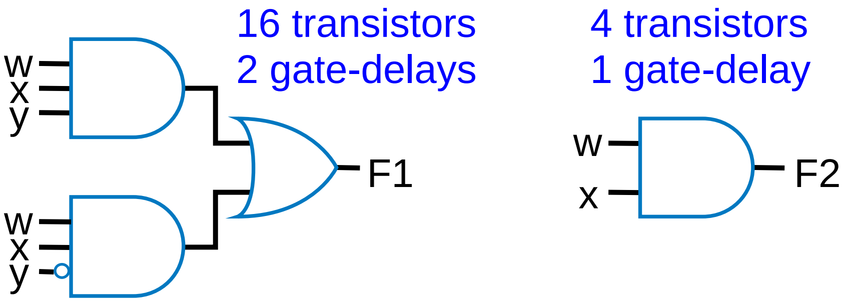
Chapter 6:

Optimizations and Tradeoffs

Introduction

- We now know how to build digital circuits
  - How can we build ***better*** circuits?
- Let's consider two important design criteria
  - ***Delay*** – the time from inputs changing to new correct stable output
  - ***Size*** – the number of transistors
  - For quick estimation, assume
    - Every gate has delay of "1 gate-delay"
    - Every gate *input* requires 2 transistors
    - Ignore inverters

Transforming F1 to F2 represents an ***optimization***: Better in all criteria of interest



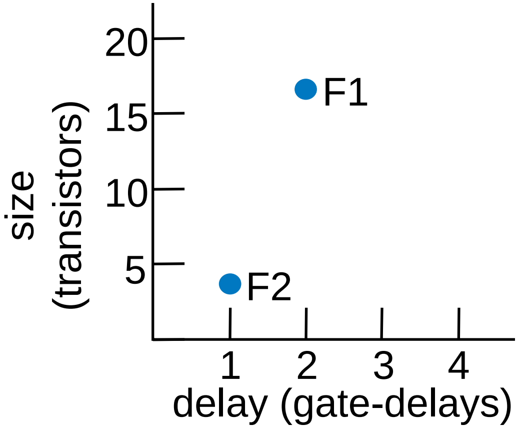
F1 = wxy + wxy'

= wx(y+y') = wx

F2 = wx

(a)

(b)



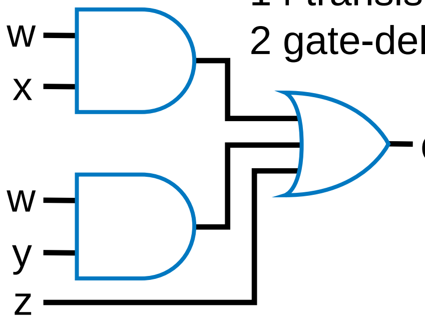
(c)

Introduction

- Tradeoff
  - Improves some, but worsens other, criteria of interest

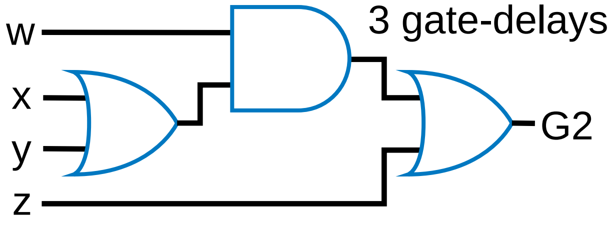
Transforming G1 to G2 represents a ***tradeoff***: Some criteria better, others worse.

14 transistors
2 gate-delays

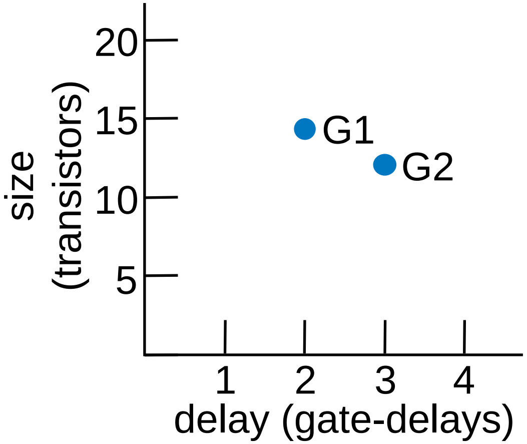


G1 = wx + wy + z

12 transistors
3 gate-delays



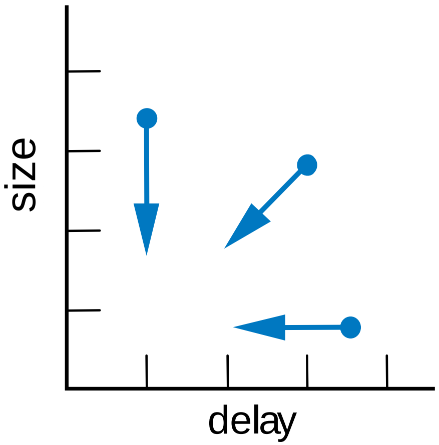
G2 = w(x+y) + z

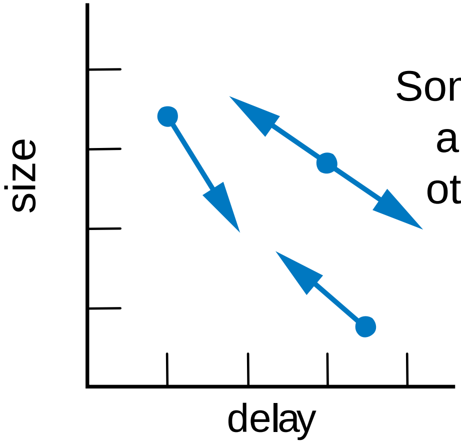


Introduction

Optimizations

All criteria of interest are improved (or at least kept the same)





Tradeoffs

Some criteria of interest are improved, while others are worsened

- We obviously prefer optimizations, but often must accept tradeoffs
  - You can’t build a car that is the most comfortable, and has the best fuel efficiency, and is the fastest – you have to give up something to gain other things.

Combinational Logic Optimization and Tradeoffs

- Two-level size optimization using algebraic methods
  - Goal: circuit with only two levels (ORed AND gates), with minimum transistors
    - Though transistors getting cheaper (Moore's Law), they still cost something
- Define problem algebraically
  - Sum-of-products yields two levels
    - F = abc + abc' is sum-of-products; G = w(xy + z) is not.
  - Transform sum-of-products equation to have fewest literals and terms
    - Each literal and term translates to a gate input, each of which translates to about 2 transistors (see Ch. 2)
    - Ignore inverters for simplicity

Example

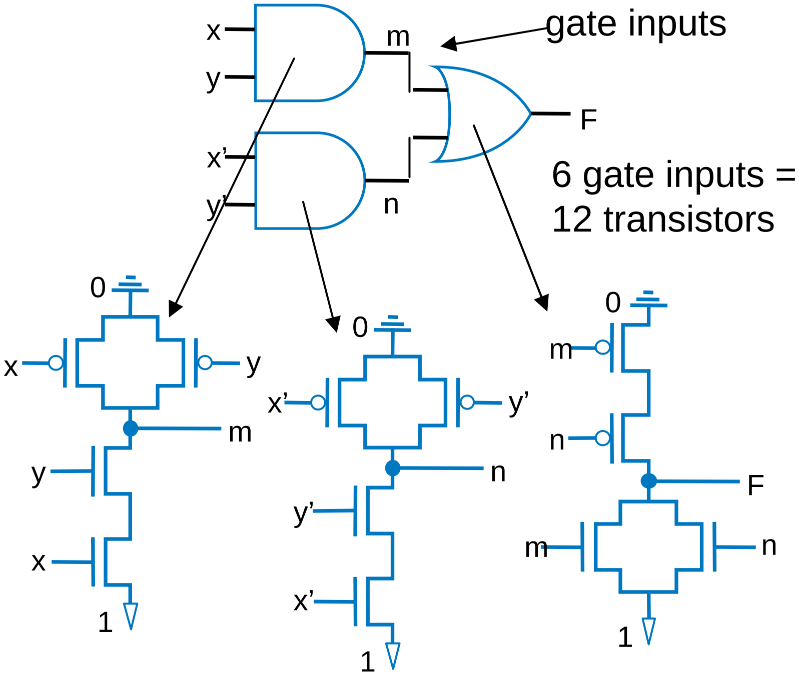
F = xyz + xyz' + x'y'z' + x'y'z

F = xy(z + z') + x'y'(z + z')

F = xy*1 + x'y'*1

F = xy + x'y'

4 literals + 2 terms = 6 gate inputs



6 gate inputs = 12 transistors

Note: Assuming 4-transistor 2-input AND/OR circuits; in reality, only NAND/NOR are so efficient.

Algebraic Two-Level Size Minimization

- Previous example showed common algebraic minimization method
  - (Multiply out to sum-of-products, then)
  - Apply following as much possible
    - $a\mathbf{b} + a\mathbf{b'} = a(\mathbf{b + b'}) = a*\mathbf{1} = a$
    - "Combining terms to eliminate a variable"
      - (Formally called the "Uniting theorem")
  - Duplicating a term sometimes helps
    - Note that doesn't change function
      - c + d = c + d + d = c + d + d + d + d ...
  - Sometimes after combining terms, can combine resulting terms

$F = xy\mathbf{z} + xy\mathbf{z'} + x'y'\mathbf{z'} + x'y'\mathbf{z}$

$F = xy(\mathbf{z + z'}) + x'y'(\mathbf{z + z'})$

$F = xy*\mathbf{1} + x'y'*\mathbf{1}$ a

$F = xy + x'y'$

$F = x'y'z' + \mathbf{x'y'z} + x'yz$

$F = x'y'z' + \mathbf{x'y'z} + \mathbf{x'y'z} + x'yz$

$F = x'y'(z+z') + x'z(y'+y)$ a

$F = x'y' + x'z$

$G = xy'z' + xy'z + xyz + xyz'$

$G = xy'(z'+z) + xy(z+z')$

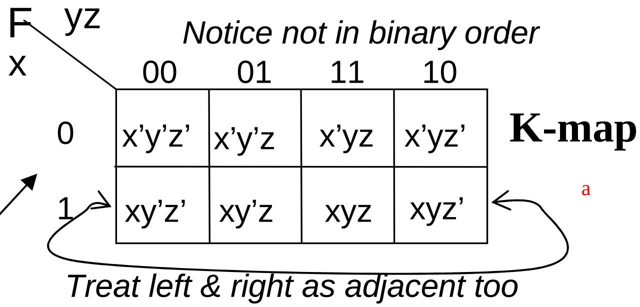
$G = xy' + xy$ *(now do again)* a

$G = x(y'+y)$

$G = x$

Karnaugh Maps for Two-Level Size Minimization

- Easy to miss “seeing” possible opportunities to combine terms
- ***Karnaugh Maps (K-maps)***
  - **Graphical** method to help us find opportunities to combine terms
  - Minterms differing in one variable are *adjacent* in the map
  - Can clearly see opportunities to combine terms – look for adjacent 1s
    - For F, clearly two opportunities
    - Top left circle is *shorthand* for x’y’z’+x’y’z = x’y’(z’+z) = x’y’(1) = x’y’
    - Draw circle, write term that has all the literals except the one that changes in the circle
      - Circle xy, x=1 & y=1 in both cells of the circle, but z changes (z=1 in one cell, 0 in the other)
    - Minimized function: OR the final terms



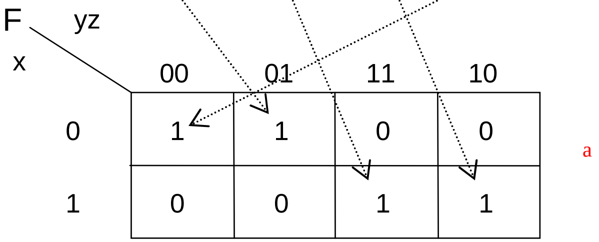
Notice not in binary order

K-map

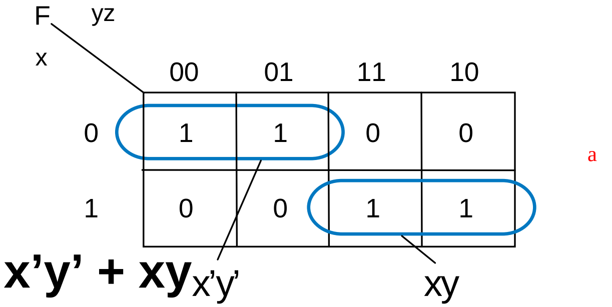
F x \ yz	00	01	11	10
0	x’y’z’	x’y’z	x’yz	x’yz’
1	xy’z’	xy’z	xyz	xyz’

Treat left & right as adjacent too

F = x’y’z + xyz + xyz’ + x’y’z’



F x \ yz	00	01	11	10
0	1	1	0	0
1	0	0	1	1



F x \ yz	00	01	11	10
0	1	1	0	0
1	0	0	1	1

F = x’y’ + xy

Easier than all that algebra:

F = xyz + xyz’ + x’y’z’ + x’y’z
F = xy(z + z’) + x’y’(z + z’)
F = xy*1 + x’y’*1
F = xy + x’y’

K-maps

- Four adjacent 1s means two variables can be eliminated
  - Makes intuitive sense – those two variables appear in all combinations, so one *must* be true
  - Draw one big circle – *shorthand* for the algebraic transformations above

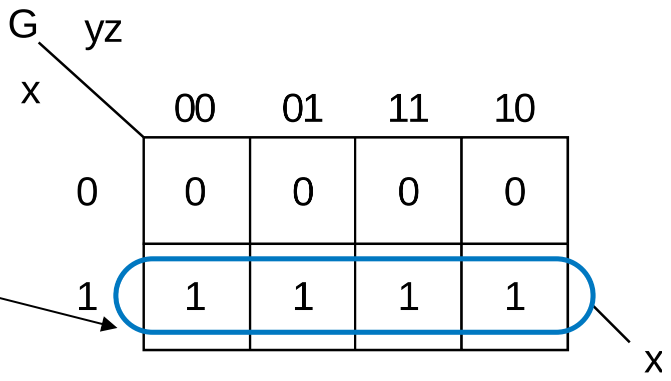
$G = xy'z' + xy'z + xyz + xyz'$

$G = x(y'z' + y'z + yz + yz')$ (must be true)

$G = x(y'(z'+z) + y(z+z'))$

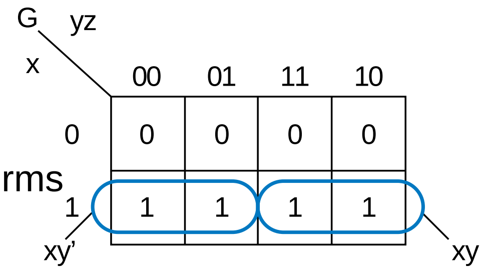
$G = x(y'+y)$

$G = x$



x \ yz	00	01	11	10
0	0	0	0	0
1	1	1	1	1

Draw the biggest circle possible, or you'll have more terms than really needed



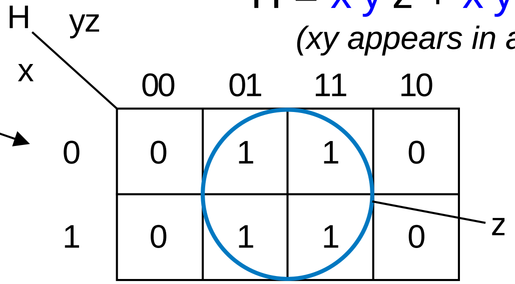
x \ yz	00	01	11	10
0	0	0	0	0
1	1	1	1	1

K-maps

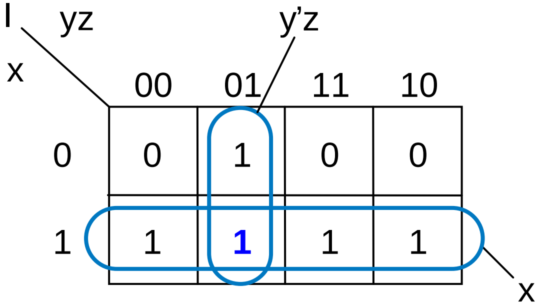
- Four adjacent cells can be in shape of a square
- *OK* to cover a 1 twice
  - Just like duplicating a term
    - Remember, c + d = c + d + d
- No *need* to cover 1s more than once
  - Yields extra terms – not minimized

H = x'y'z + x'yz + xy'z + xyz

(xy appears in all combinations)



x \ yz	00	01	11	10
0	0	1	1	0
1	0	1	1	0



x \ yz	00	01	11	10
0	0	1	0	0
1	1	1	1	1

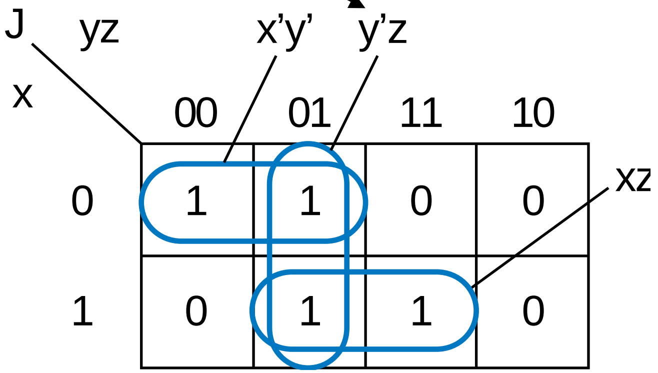
The two circles are shorthand for:

I = x'y'z + xy'z' + xy'z + xyz + xyz'

I = x'y'z + xy'z + xy'z' + xy'z + xyz + xyz'

I = (x'y'z + xy'z) + (xy'z' + xy'z + xyz + xyz')

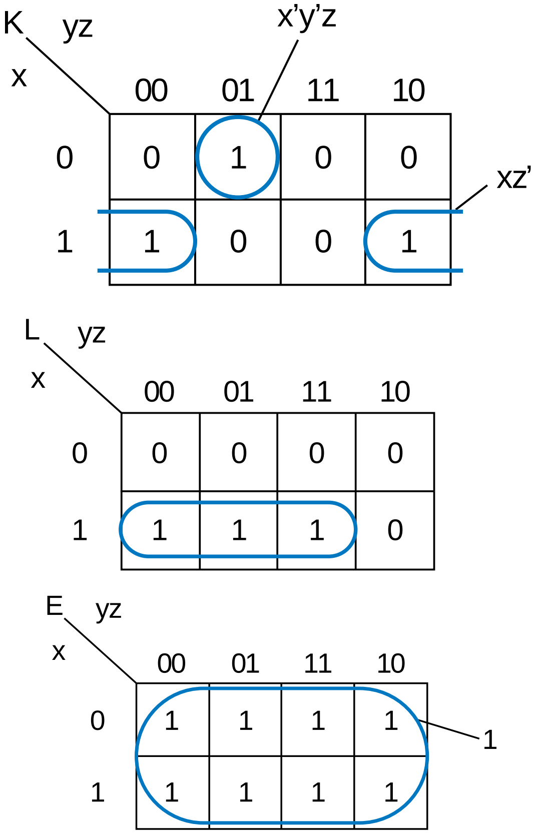
I = (y'z) + (x)



x \ yz	00	01	11	10
0	1	1	0	0
1	0	1	1	0

K-maps

- Circles can cross left/right sides
  - Remember, edges are adjacent
    - Minterms differ in one variable only
- Circles must have 1, 2, 4, or 8 cells – 3, 5, or 7 not allowed
  - 3/5/7 doesn't correspond to algebraic transformations that combine terms to eliminate a variable
- Circling all the cells is OK
  - Function just equals 1



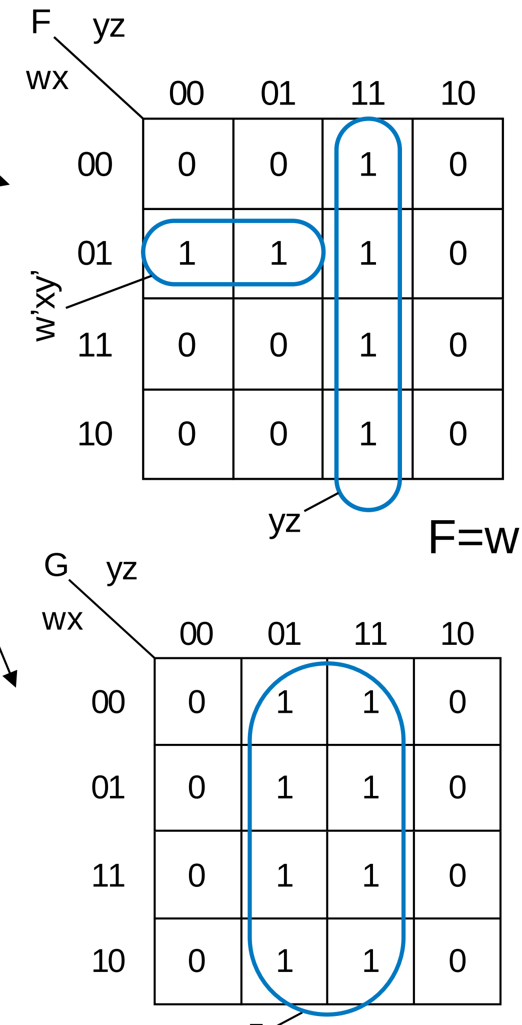
K x \ yz	00	01	11	10
0	0	1	0	0
1	1	0	0	1

L x \ yz	00	01	11	10
0	0	0	0	0
1	1	1	1	0

E x \ yz	00	01	11	10
0	1	1	1	1
1	1	1	1	1

K-maps for Four Variables

- Four-variable K-map follows same principle
  - Adjacent cells differ in one variable
  - Left/right adjacent
  - Top/bottom also adjacent
- 5 and 6 variable maps exist
  - But hard to use
- Two-variable maps exist
  - But not very useful – easy to do algebraically by hand

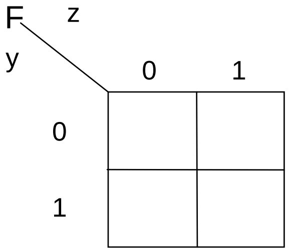


F wx \ yz	00	01	11	10
00	0	0	1	0
01	1	1	1	0
11	0	0	1	0
10	0	0	1	0

F=w'xy'+yz

G wx \ yz	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	0	1	1	0
10	0	1	1	0

G=z



F y \ z	0	1
0		
1		

Two-Level Size Minimization Using K-maps

General K-map method

1. Convert the function's equation into sum-of-products form
2. Place 1s in the appropriate K-map cells for each term
3. Cover all 1s by drawing the fewest largest circles, with every 1 included at least once; write the corresponding term for each circle
4. OR all the resulting terms to create the minimized function.

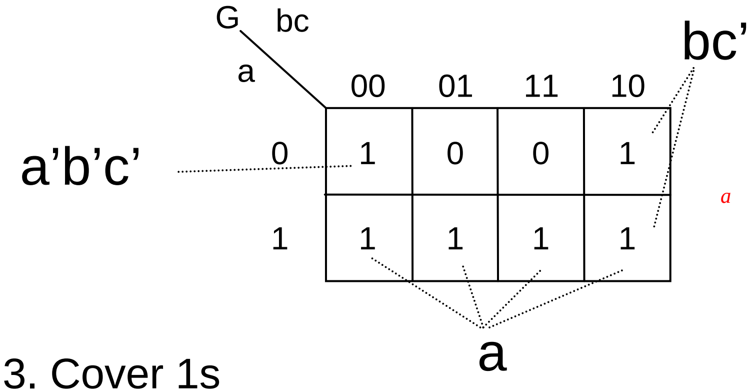
Example: Minimize:

$G = a + a'b'c' + b*(c' + bc')$

1. Convert to sum-of-products

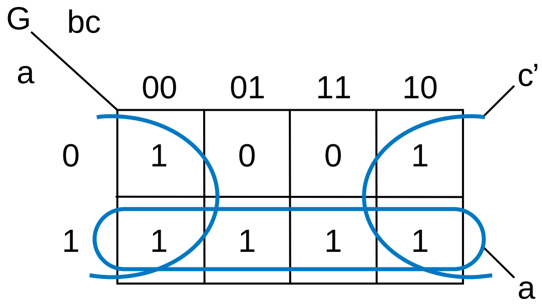
$G = a + a'b'c' + bc' + bc'$

2. Place 1s in appropriate cells



G a \ bc	00	01	11	10
0	1	0	0	1
1	1	1	1	1

3. Cover 1s



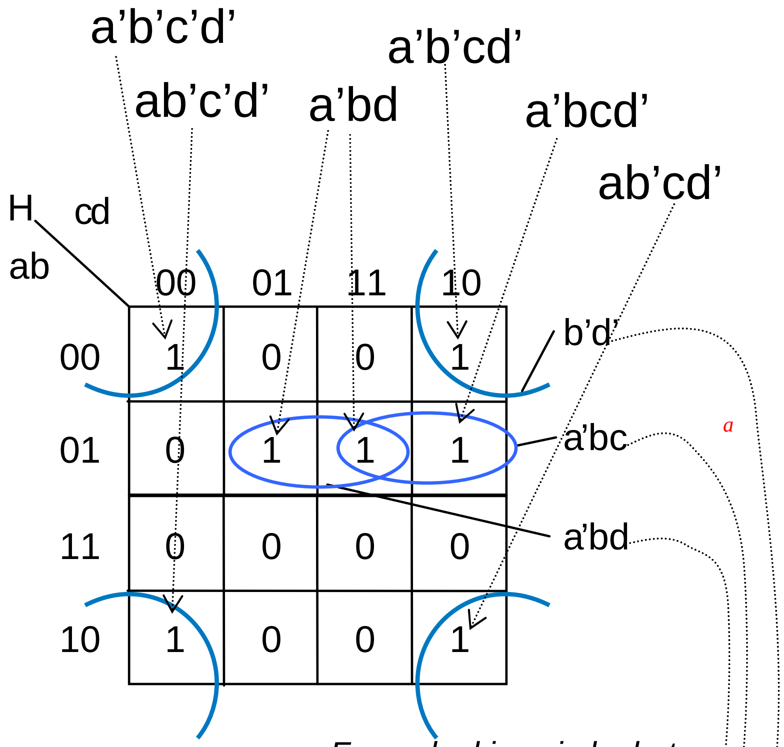
G a \ bc	00	01	11	10
0	1	0	0	1
1	1	1	1	1

4. OR terms: **G = a + c'**

Two-Level Size Minimization Using K-maps
– Four Variable Example

- Minimize:
  - H = a’b’(cd’ + c’d’) + ab’c’d’ + ab’cd’ + a’bd + a’bcd’

1. Convert to sum-of-products:
   - H = a’b’cd’ + a’b’c’d’ + ab’c’d’ + ab’cd’ + a’bd + a’bcd’
2. Place 1s in K-map cells
3. Cover 1s
4. OR resulting terms



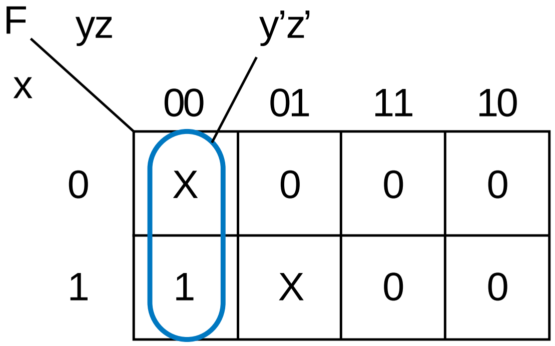
H cd ab	00	01	11	10
00	1	0	0	1
01	0	1	1	1
11	0	0	0	0
10	1	0	0	1

Funny-looking circle, but remember that left/right adjacent, and top/bottom adjacent

H = b’d’ + a’bc + a’bd

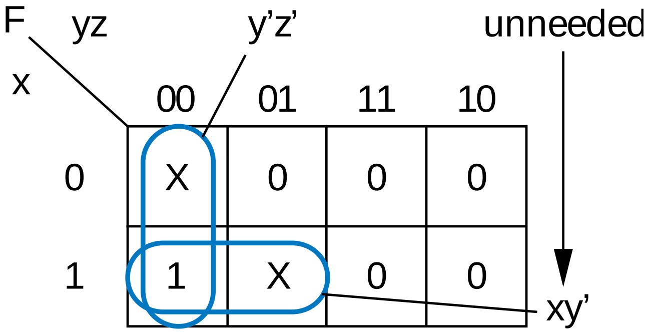
Don’t Care Input Combinations

- What if particular input combinations can never occur?
  - e.g., Minimize F = xy’z’, given that x’y’z’ (xyz=000) can *never* be true, and that xy’z (xyz=101) can *never* be true
  - So it doesn’t matter what F outputs when x’y’z’ or xy’z is true, because those cases *will never occur*
  - Thus, make F be 1 or 0 for those cases *in a way that best minimizes the equation*
- On K-map
  - Draw **X**s for don’t care combinations
    - Include X in circle ONLY if minimizes equation
    - Don’t include other Xs



x \ yz	00	01	11	10
0	X	0	0	0
1	1	X	0	0

Good use of don’t cares



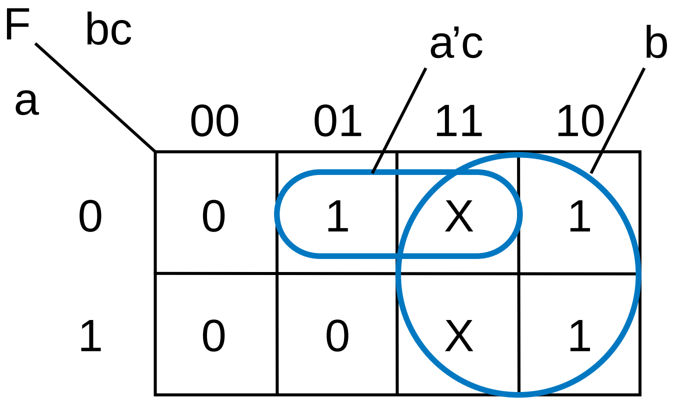
x \ yz	00	01	11	10
0	X	0	0	0
1	1	X	0	0

Unnecessary use of don’t cares; results in extra term

Minimizization Example using Don’t Cares

- Minimize:
  - F = a’bc’ + abc’ + a’b’c
  - Given don’t cares: a’bc, abc

- Note: Use don’t cares with caution
  - Must be *sure* that we really don’t care what the function outputs for that input combination
  - If we do care, even the slightest, then it’s probably safer to set the output to 0



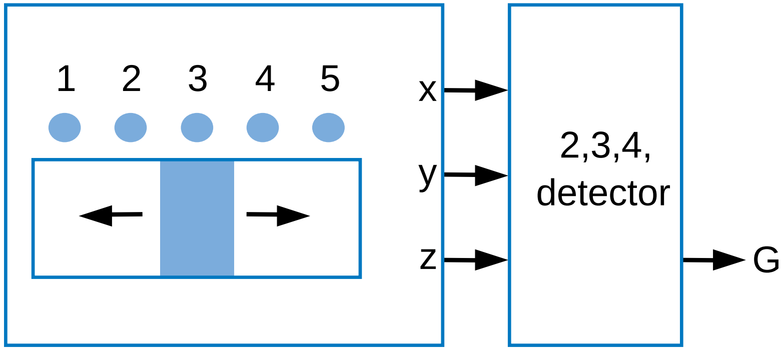
a \ bc	00	01	11	10
0	0	1	X	1
1	0	0	X	1

a

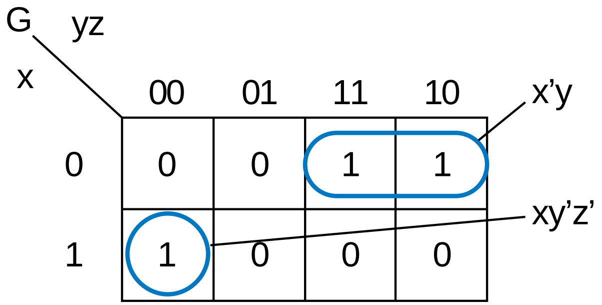
F = a’c + b

Minimization with Don’t Cares Example: Sliding Switch

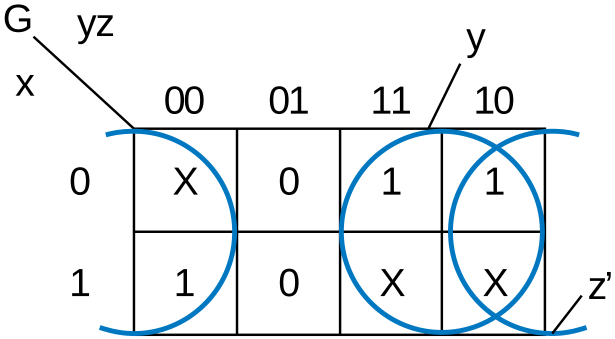
- Switch with 5 positions
  - 3-bit value gives position in binary
- Want circuit that
  - Outputs 1 when switch is in position 2, 3, or 4
  - Outputs 0 when switch is in position 1 or 5
  - Note that the 3-bit input can never output binary 0, 6, or 7
    - Treat as don’t care input combinations



Without don’t cares:
F = x’y + xy’z’



	00	01	11	10
0	0	0	1	1
1	1	0	0	0

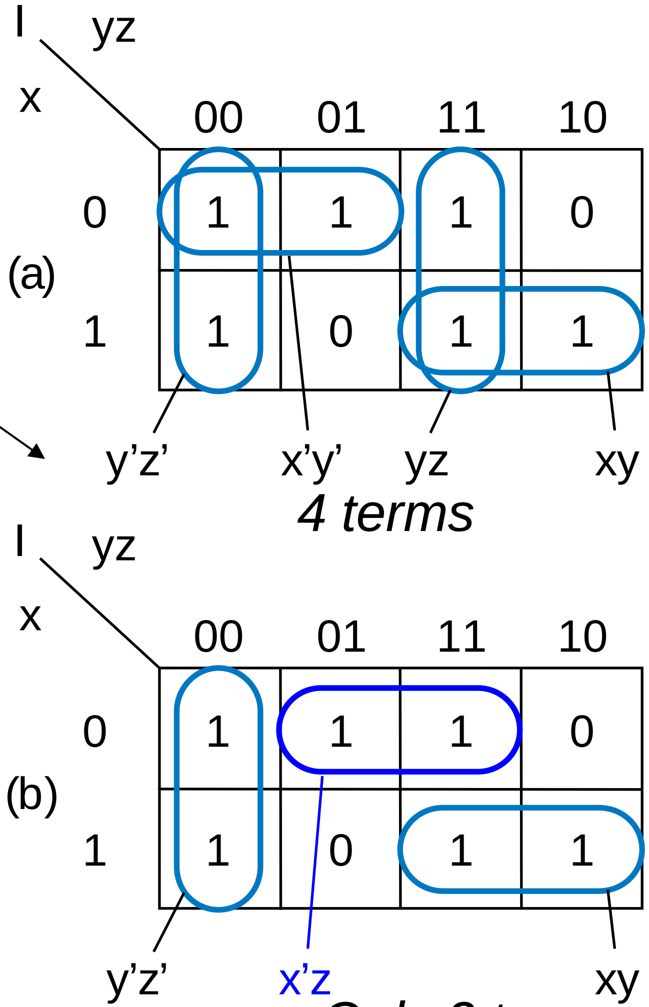


	00	01	11	10
0	X	0	1	1
1	1	0	X	X

With don’t cares:
F = y + z’

Automating Two-Level Logic Size Minimization

- Minimizing by hand
  - Is hard for functions with 5 or more variables
  - May not yield minimum cover depending on order we choose
  - Is error prone
- Minimization thus typically done by automated tools
  - ***Exact algorithm***: finds optimal solution
  - ***Heuristic***: finds good solution, but not necessarily optimal



x \ yz	00	01	11	10
0	1	1	1	0
1	1	0	1	1

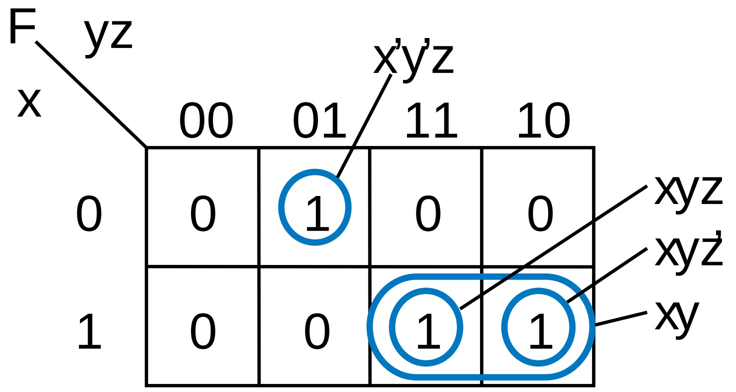
4 terms

x \ yz	00	01	11	10
0	1	1	1	0
1	1	0	1	1

Only 3 terms

Basic Concepts Underlying Automated Two-Level Logic Minimization

- Definitions
  - ***On-set***: All minterms that define when F=1
  - ***Off-set***: All minterms that define when F=0
  - ***Implicant***: Any product term (minterm or other) that when 1 causes F=1
    - On K-map, any legal (but not necessarily largest) circle
    - Cover: Implicant xy ***covers*** minterms xyz and xyz’
  - ***Expanding*** a term: removing a variable (like larger K-map circle)
    - xyz → xy is an expansion of xyz



x \ yz	00	01	11	10
0	0	1	0	0
1	0	0	1	1

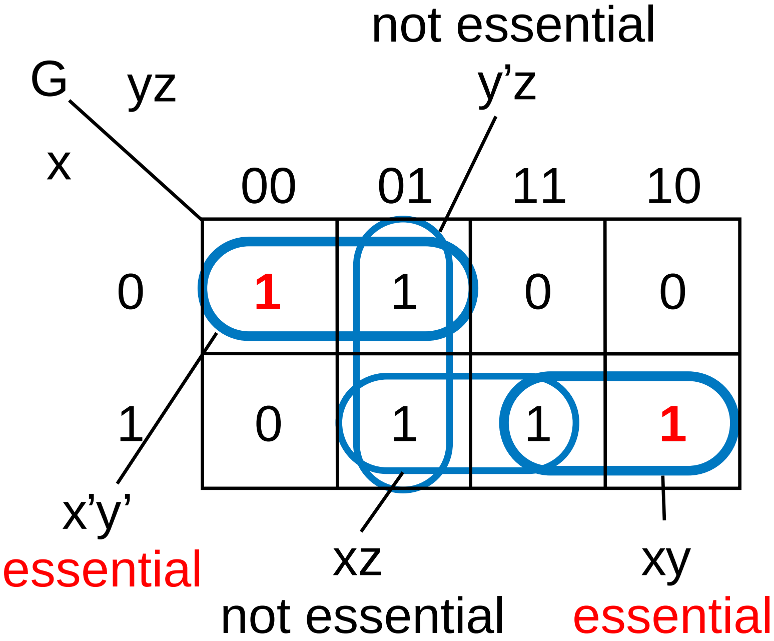
4 implicants of F

*Note: We use K-maps here just for intuitive illustration of concepts; automated tools do **not** use K-maps.*

- ***Prime implicant***: Maximally expanded implicant – any expansion would cover 1s not in on-set
  - x’y’z, and xy, above
  - But not xyz or xyz’ – they can be expanded

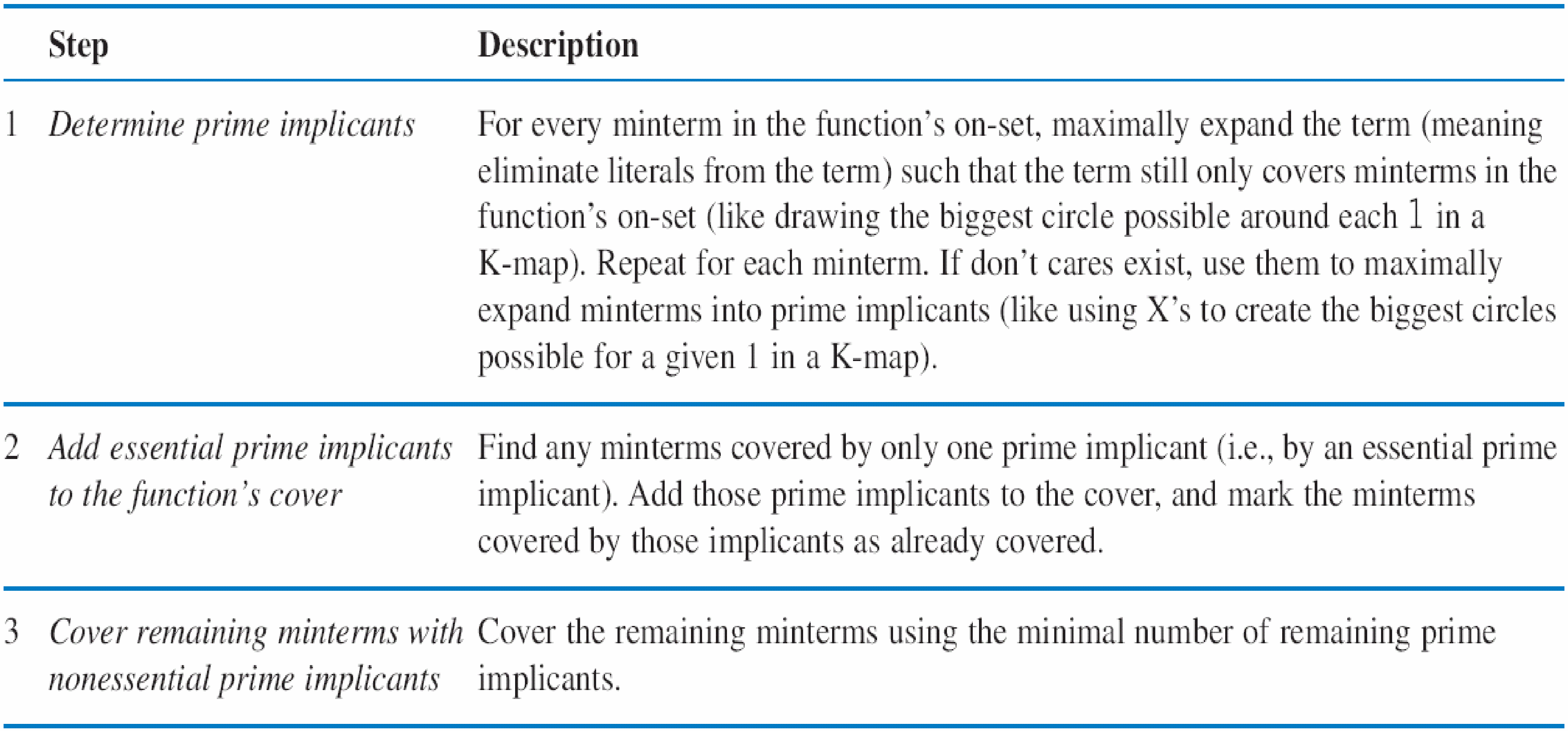
Basic Concepts Underlying Automated Two-Level Logic Minimization

- Definitions (cont)
  - ***Essential prime implicant***: The only prime implicant that covers a particular minterm in a function's on-set
    - Importance: We ***must*** include ***all*** essential PIs in a function's cover
    - In contrast, some, but not all, non-essential PIs will be included



G (x \ yz)	00	01	11	10
0	1	1	0	0
1	0	1	1	1

Automated Two-Level Logic Minimization Method

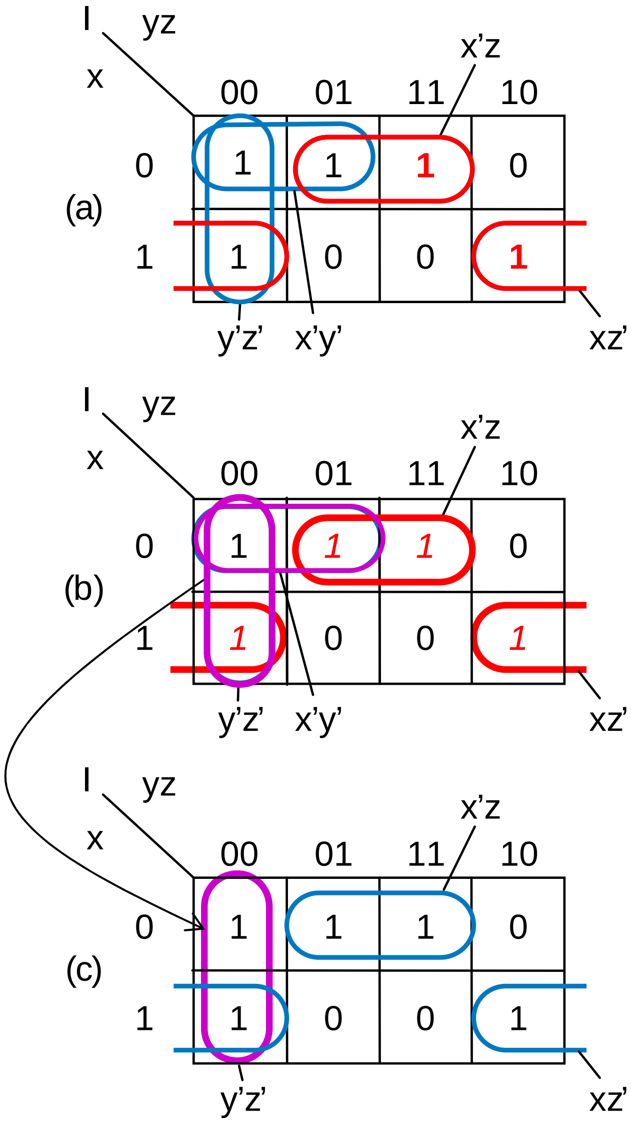


Step		Description
1	*Determine prime implicants*	For every minterm in the function's on-set, maximally expand the term (meaning eliminate literals from the term) such that the term still only covers minterms in the function's on-set (like drawing the biggest circle possible around each 1 in a K-map). Repeat for each minterm. If don't cares exist, use them to maximally expand minterms into prime implicants (like using X's to create the biggest circles possible for a given 1 in a K-map).
2	*Add essential prime implicants to the function's cover*	Find any minterms covered by only one prime implicant (i.e., by an essential prime implicant). Add those prime implicants to the cover, and mark the minterms covered by those implicants as already covered.
3	*Cover remaining minterms with nonessential prime implicants*	Cover the remaining minterms using the minimal number of remaining prime implicants.

- Steps 1 and 2 are exact
- Step 3: Hard. Checking all possibilities: exact, but computationally expensive. Checking some but not all: heuristic.

Example of Automated Two-Level Minimization

- 1. Determine all prime implicants
- 2. Add essential PIs to cover
  - Italicized 1s are thus already covered
  - Only one uncovered 1 remains
- 3. Cover remaining minterms with non-essential PIs
  - Pick among the two possible PIs



(a)

x \ yz	00	01	11	10
0	1	1	1	0
1	1	0	0	1

(b)

x \ yz	00	01	11	10
0	1	*1*	*1*	0
1	*1*	0	0	*1*

(c)

x \ yz	00	01	11	10
0	1	1	1	0
1	1	0	0	1

Problem with Methods that Enumerate all Minterms or Compute all Prime Implicants

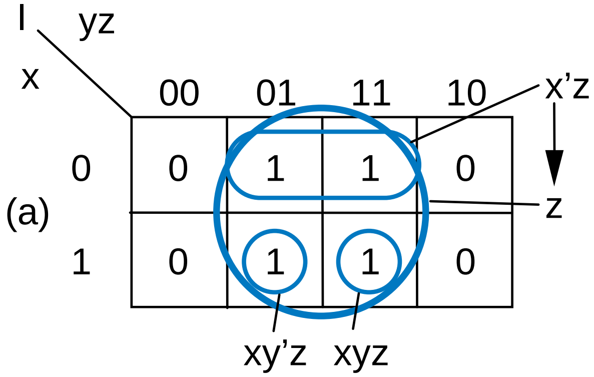
- Too many minterms for functions with many variables
  - Function with 32 variables:
    - 2^{32} = 4 billion possible minterms.
    - Too much compute time/memory
- Too many computations to generate all prime implicants
  - Comparing every minterm with every other minterm, for 32 variables, is $(4 \text{ billion})^2$ = 1 quadrillion computations
  - Functions with many variables could requires days, months, years, or more of computation – unreasonable

Solution to Computation Problem

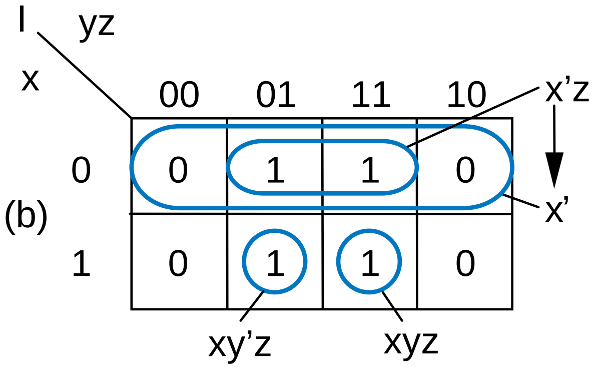
- Solution
  - Don’t generate all minterms or prime implicants
  - Instead, just take input equation, and try to “iteratively” improve it
  - Ex: F = abcdefgh + abcdefgh’+ jklmnop
    - Note: 15 variables, may have thousands of minterms
    - But can minimize just by combining first two terms:
      - F = abcdefg(h+h’) + jklmnop = abcdefg + jklmnop

Two-Level Minimization using Iterative Method

- Method: Randomly apply "expand" operations, see if helps
  - Expand: remove a variable from a term
    - Like expanding circle size on K-map
      - e.g., Expanding x'z to z legal, but expanding x'z to z' not legal, in shown function
      - After expand, remove other terms covered by newly expanded term
  - Keep trying (iterate) until doesn't help



	00	01	11	10
0	0	1	1	0
1	0	1	1	0



	00	01	11	10
0	0	1	1	0
1	0	1	1	0

Ex:

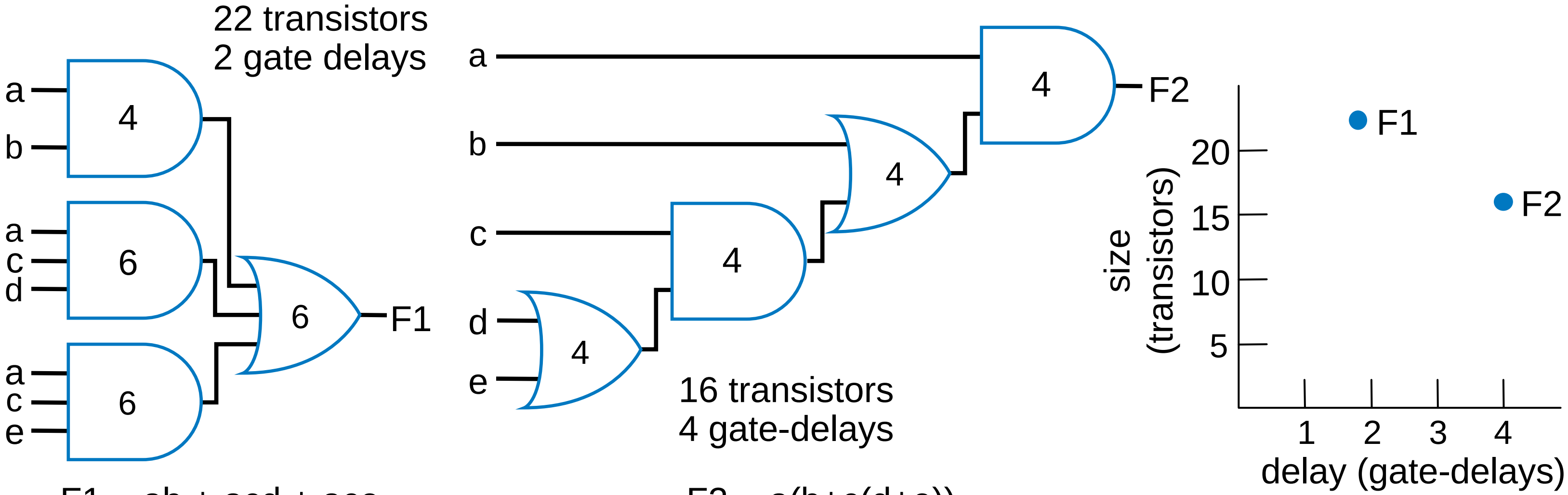
F = abcdefgh + abcdefgh'+ jklmnop

F = abcdefg + abcdefgh' + jklmnop

F = abcdefg + jklmnop

Multi-Level Logic Optimization – Performance/Size Tradeoffs

- We don’t always need the speed of two level logic
  - Multiple levels may yield fewer gates
  - Example
    - F1 = ab + acd + ace → F2 = ab + ac(d + e) = a(b + c(d + e))
    - General technique: Factor out literals – xy + xz = x(y+z)



F1 = ab + acd + ace
(a)

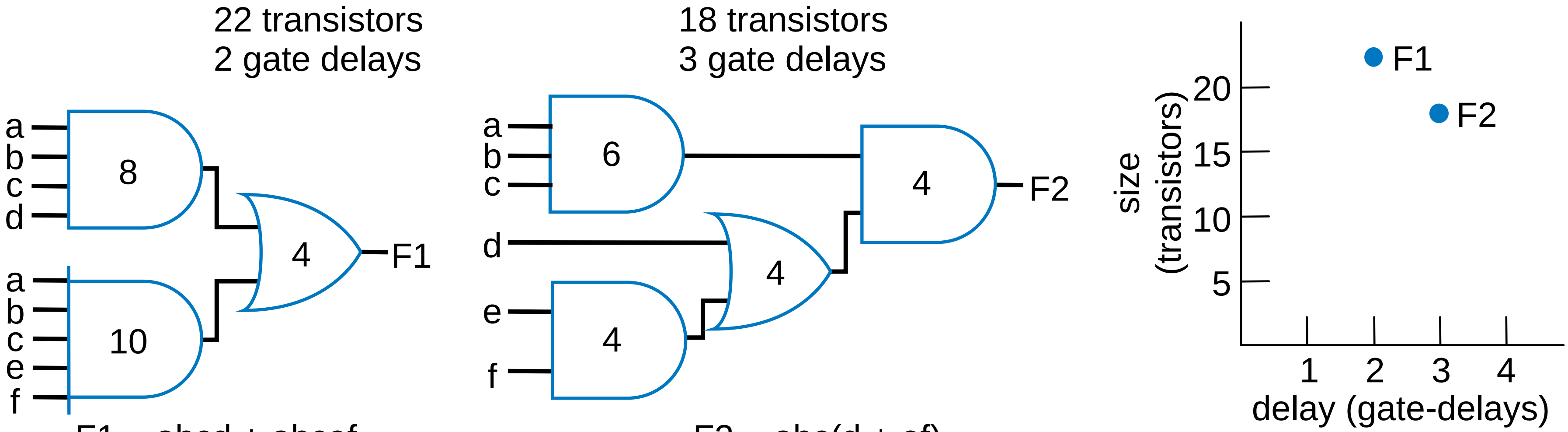
F2 = a(b+c(d+e))
(b)

(c)

Multi-Level Example

- Q: Use multiple levels to reduce number of transistors for
  - F1 = abcd + abcef
- A: abcd + abcef = abc(d + ef)
  - Has fewer gate inputs, thus fewer transistors

a



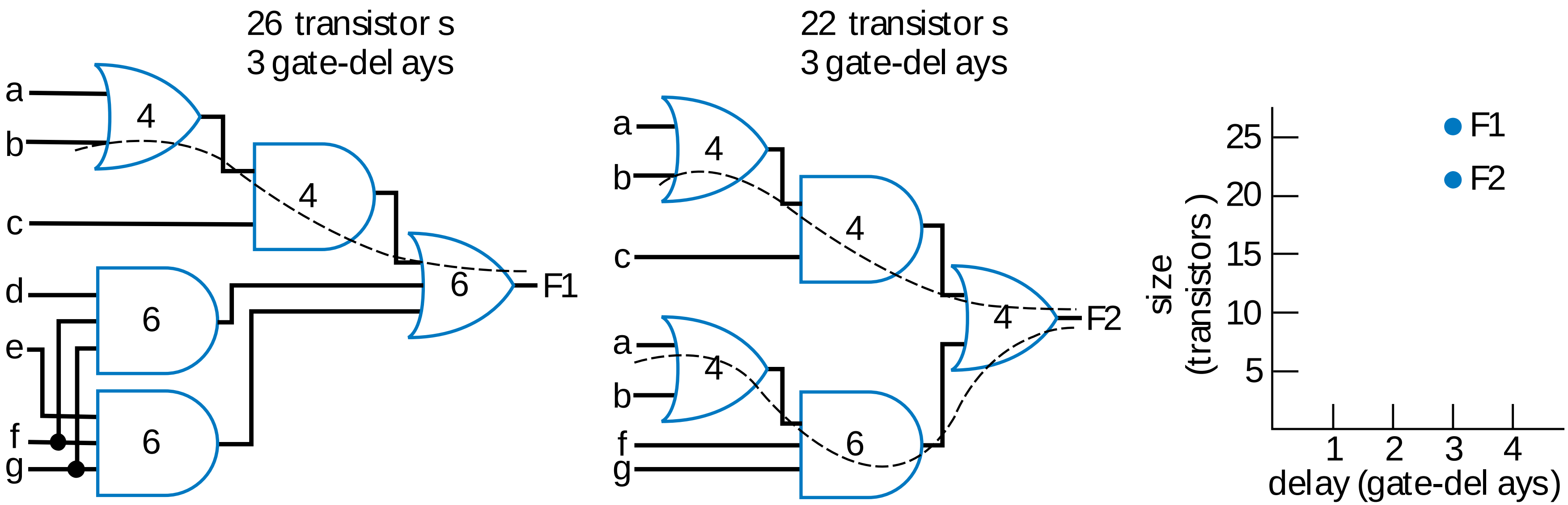
F1 = abcd + abcef
(a)

F2 = abc(d + ef)
(b)

(c)

Multi-Level Example: Non-Critical Path

- Critical path: longest delay path to output
- Optimization: reduce size of logic on non-critical paths by using multiple levels



F1 = (a+b)c + dfg + efg

(a)

F2 = (a+b)c + (d+e)fg

(b)

(c)

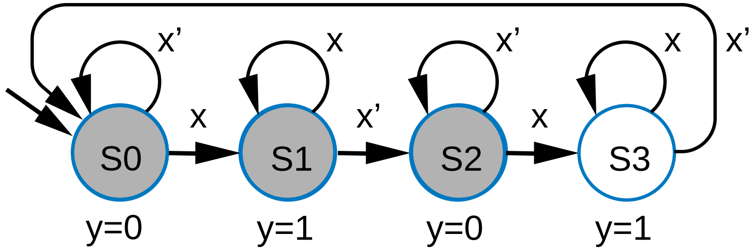
Automated Multi-Level Methods

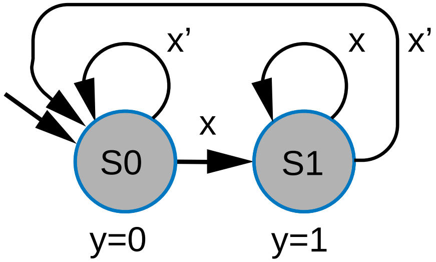
- Main techniques use heuristic iterative methods
  - Define various operations
    - “Factor out”: xy + xz = x(y+z)
    - Expand, and others
  - Randomly apply, see if improves
    - May even accept changes that worsen, in hopes eventually leads to even better equation
    - Keep trying until can’t find further improvement
  - Not guaranteed to find best circuit, but rather a good one

State Reduction (State Minimization)

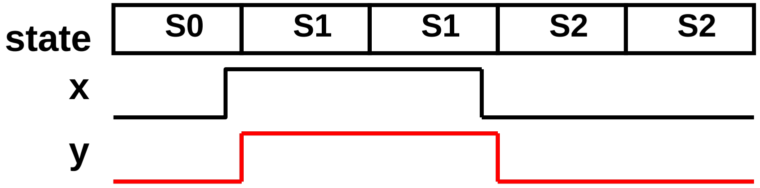
- **Goal**: Reduce number of states in FSM *without* changing behavior
  - – Fewer states potentially reduces size of state register
- Consider the two FSMs below with *x*=1, then 1, then 0, 0

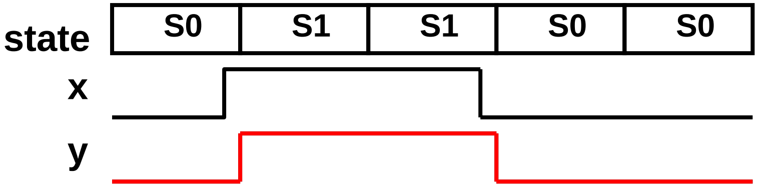
Inputs: x; Outputs: y



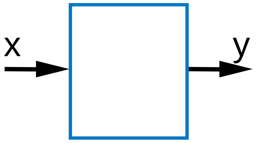


a





For the same sequence of inputs,
the output of the two FSMs is the same



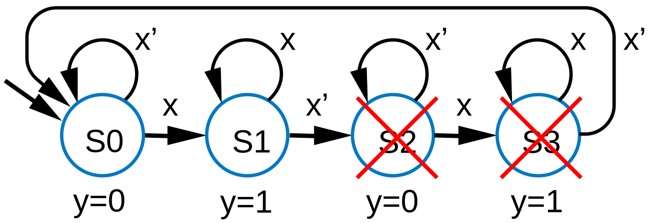
if x = 1,1,0,0
then y = 0,1,1,0,0

State Reduction: Equivalent States

Two states are equivalent if:

1. They assign the same values to outputs
   - e.g. **S0** and **S2** both assign *y* to 0,
   - **S1** and **S3** both assign *y* to 1
2. AND, for all possible sequences of inputs, the FSM outputs will be the same starting from either state
   - e.g. say *x*=1,1,0,0,…
     - starting from **S1**, *y*=1,1,0,0,…
     - starting from **S3**, *y*=1,1,0,0,…

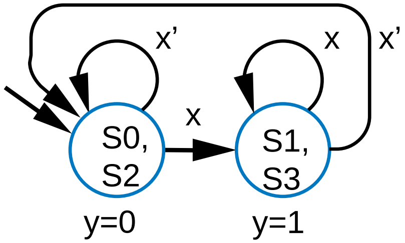
Inputs: x; Outputs: y



States S0 and S2 equivalent

States S1 and S3 equivalent

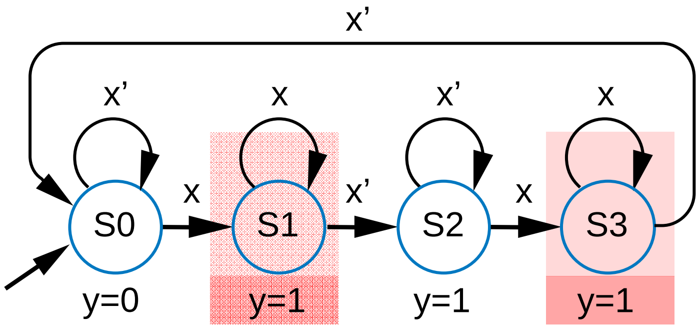
a



State Reduction: Example with no Equivalencies

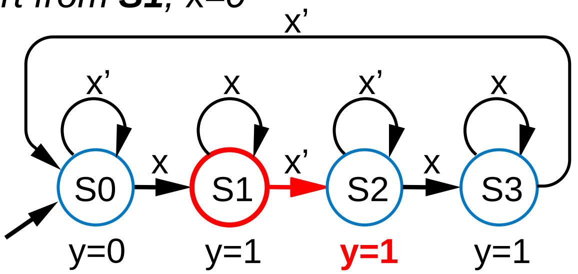
- Another example…
- State **S0** is not equivalent with any other state since its output (*y*=0) differs from other states' output

Inputs: x; Outputs: y

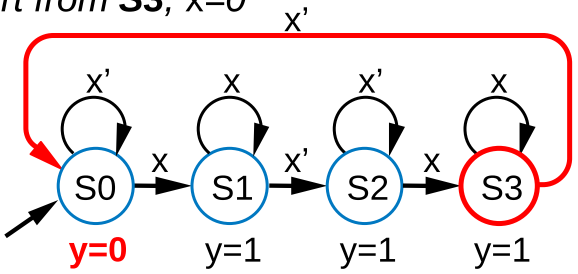


- Consider state **S1** and **S3**
  - Outputs are initially the same (*y*=1)
  - From **S1**, when *x*=0, go to **S2** where *y*=1
  - From **S3**, when *x*=0, go to **S0** where *y*=0
  - Outputs differ, so **S1** and **S3** are *not* equivalent.

*Start from **S1**, x=0*

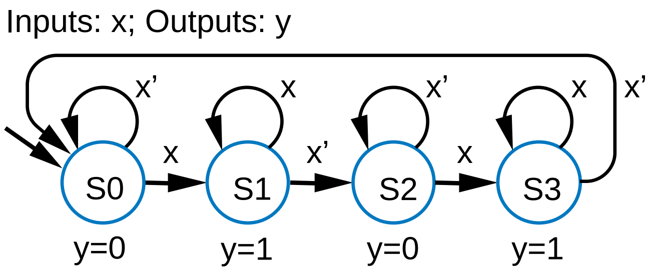


*Start from **S3**, x=0*



State Reduction with Implication Tables

- State reduction through visual inspection (what we did in the last few slides) isn't reliable and cannot be automated – a more methodical approach is needed: **implication tables**
- Example:



S0	■	▨	▨	▨
S1		■	▨	▨
S2			■	▨
S3				■
	S0	S1	S2	S3

Redundant

Diagonal

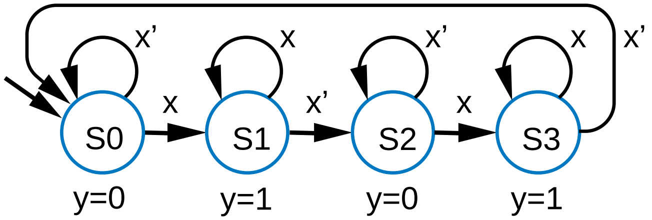
S1			
S2			
S3			
	S0	S1	S2

  - To compare every pair of states, construct a table of *state pairs* (above right)
  - Remove redundant state pairs, and state pairs along the diagonal since a state is equivalent to itself (right)

State Reduction with Implication Tables

- Mark (with an X) state pairs with different outputs as non-equivalent:
  - (**S1**,**S0**): At **S1**, *y*=1 and at **S0**, *y*=0. So **S1** and **S0** are non-equivalent.
  - (**S2**, **S0**): At **S2**, *y*=0 and at **S0**, *y*=0. So we don’t mark **S2** and **S0** now.
  - (**S2**, **S1**): Non-equivalent
  - (**S3**, **S0**): Non-equivalent
  - (**S3**, **S1**): Don’t mark
  - (**S3**, **S2**): Non-equivalent
- We can see that **S2** & **S0** might be equivalent and **S3** & **S1** might be equivalent, but only if their next states are equivalent (remember the example from two slides ago)

Inputs: x; Outputs: y

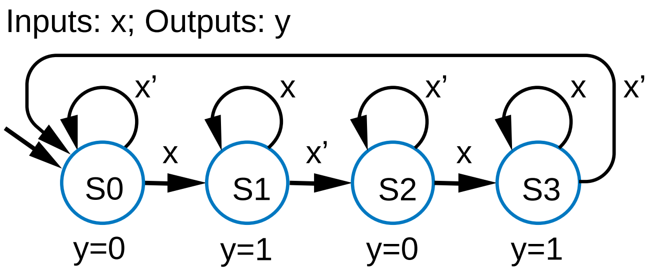


S1	☒		
S2		☒	
S3	☒		☒
	S0	**S1**	**S2**

a

State Reduction with Implication Tables

- We need to check each unmarked state pair’s next states
- We can start by listing what each unmarked state pair’s next states are for every combination of inputs
  - (**S2**, **S0**)
    - From **S2**, when *x*=1 go to **S3**
From **S0**, when *x*=1 go to **S1**
So we add (**S3**, **S1**) as a next state pair
    - From **S2**, when *x*=0 go to **S2**
From **S0**, when *x*=0 go to **S0**
So we add (**S2**, **S0**) as a next state pair
  - (**S3**, **S1**)
    - By a similar process, we add the next state pairs (**S3**, **S1**) and (**S0**, **S2**)

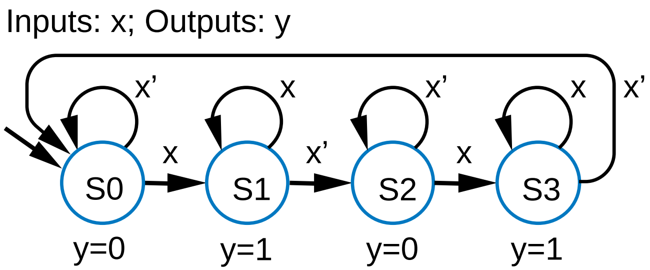


S1	✗		
S2	**(S3, S1)** **(S2, S0)**	✗	
S3	✗	**(S3, S1)** **(S0, S2)**	✗
	S0	**S1**	**S2**

a

State Reduction with Implication Tables

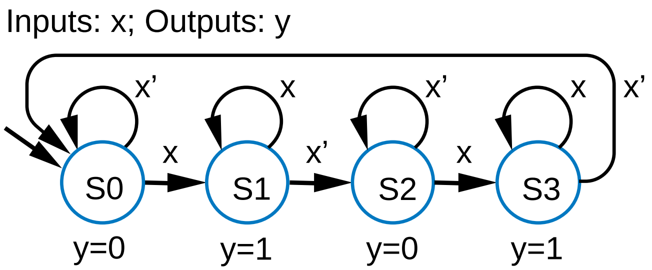
- Next we check every unmarked state pair's next state pairs
- We mark the state pair if one of its next state pairs is marked
  - (**S2**, **S0**)
    - Next state pair (**S3**, **S1**) is not marked
    - Next state pair (**S2**, **S0**) is not marked
    - So we do nothing and move on
  - (**S3**, **S1**)
    - Next state pair (**S3**, **S1**) is not marked
    - Next state pair (**S0**, **S2**) is not marked
    - So we do nothing and move on

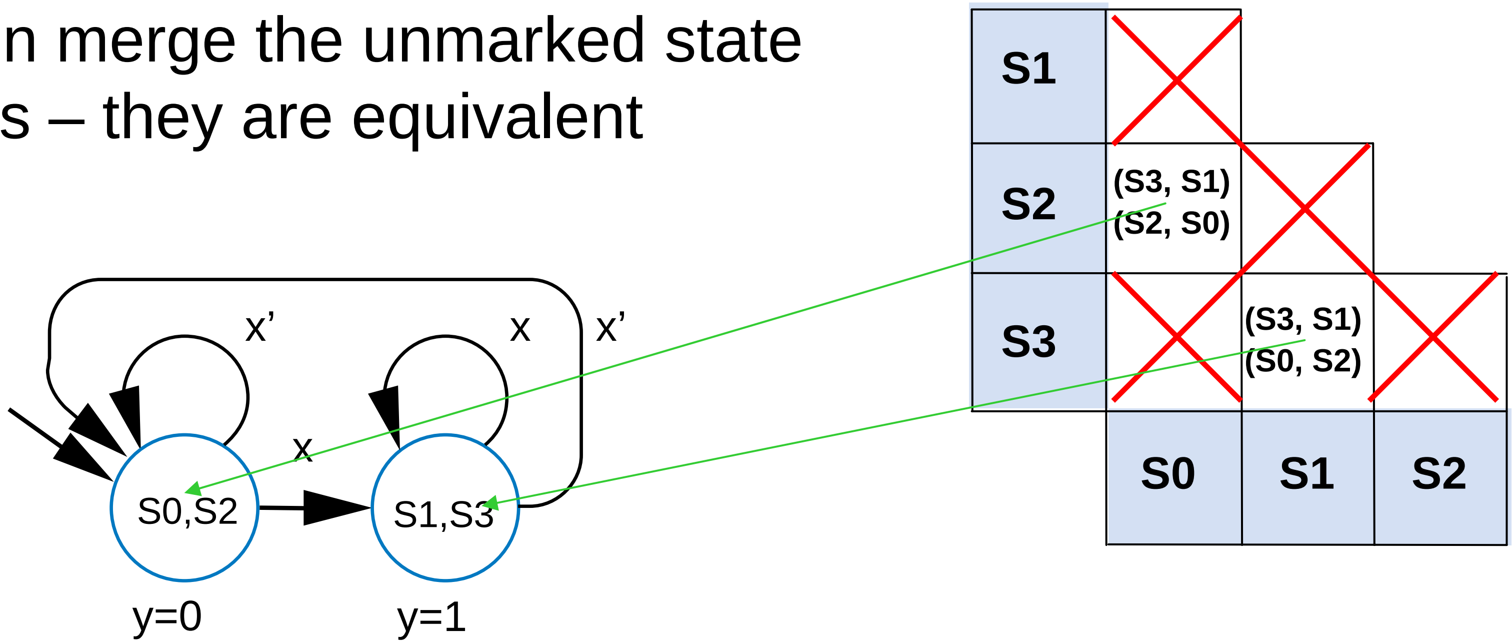


S1	✗		
S2	**(S3, S1)** **(S2, S0)**	✗	
S3	✗	**(S3, S1)** **(S0, S2)**	✗
	S0	**S1**	**S2**

State Reduction with Implication Tables

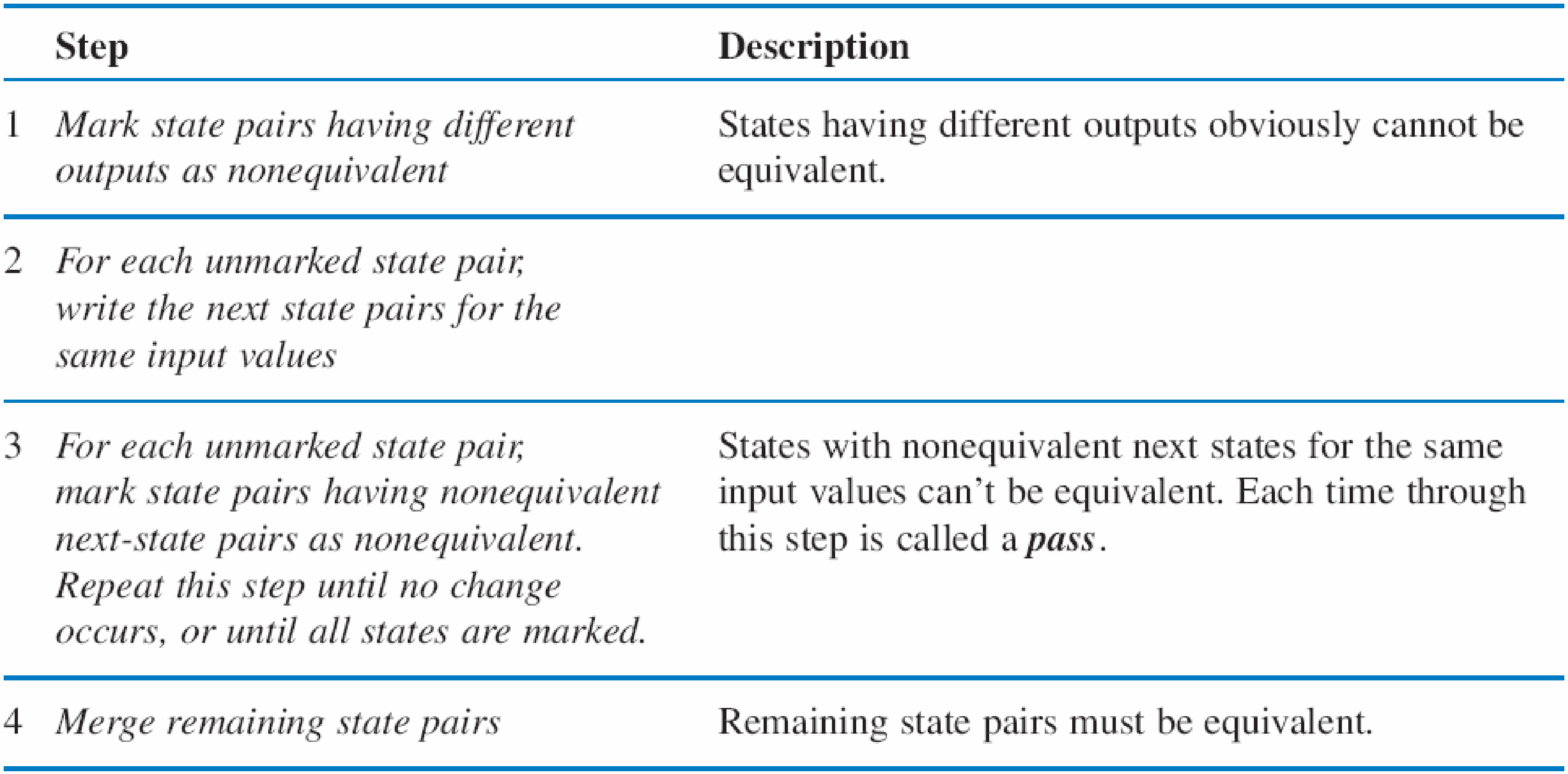
- We just made a *pass* through the implication table
  - Make additional passes until no change occurs
- Then merge the unmarked state pairs – they are equivalent





S1	✗		
S2	(S3, S1) (S2, S0)	✗	
S3	✗	(S3, S1) (S0, S2)	✗
	S0	S1	S2

State Reduction with Implication Tables

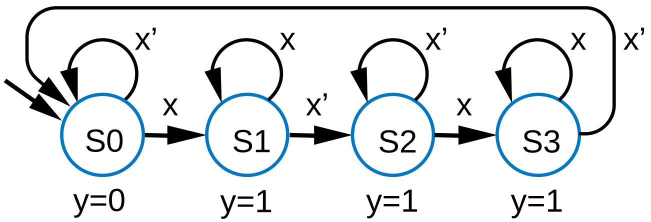


	Step	Description
1	*Mark state pairs having different outputs as nonequivalent*	States having different outputs obviously cannot be equivalent.
2	*For each unmarked state pair, write the next state pairs for the same input values*	
3	*For each unmarked state pair, mark state pairs having nonequivalent next-state pairs as nonequivalent. Repeat this step until no change occurs, or until all states are marked.*	States with nonequivalent next states for the same input values can't be equivalent. Each time through this step is called a ***pass***.
4	*Merge remaining state pairs*	Remaining state pairs must be equivalent.

State Reduction Example

- Given FSM on the right
  - **Step 1:** Mark state pairs having different outputs as nonequivalent

Inputs: x; Outputs: y

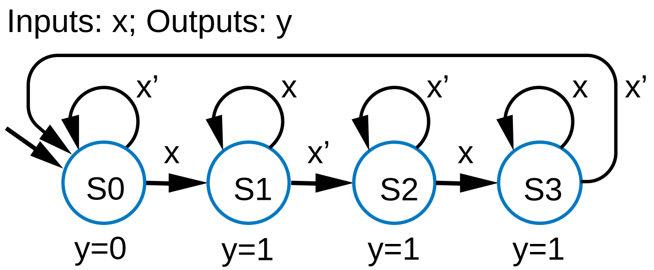


S1	✗		
S2	✗		
S3	✗		
	S0	**S1**	**S2**

a

State Reduction Example

- Given FSM on the right
  - **Step 1:** Mark state pairs having different outputs as nonequivalent
  - **Step 2:** For each unmarked state pair, write the next state pairs for the same input values

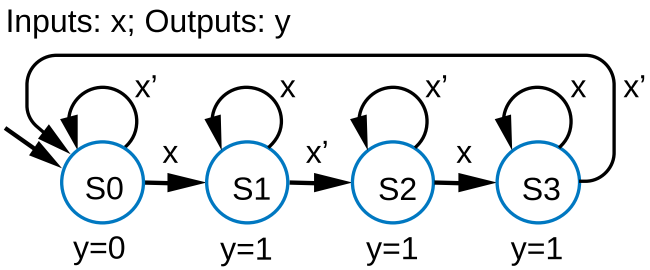


S1	✗		
S2	✗	**(S2, S2)** **(S3, S1)**	
S3	✗	**(S0, S2)** **(S3, S1)**	**(S0, S2)** **(S3, S3)**
	S0	**S1**	**S2**

a

State Reduction Example

- Given FSM on the right
  - **Step 1:** Mark state pairs having different outputs as nonequivalent
  - **Step 2:** For each unmarked state pair, write the next state pairs for the same input values
  - **Step 3:** For each unmarked state pair, mark state pairs having nonequivalent next state pairs as nonequivalent.
    - Repeat this step until no change occurs, or until all states are marked.
  - **Step 4:** Merge remaining state pairs



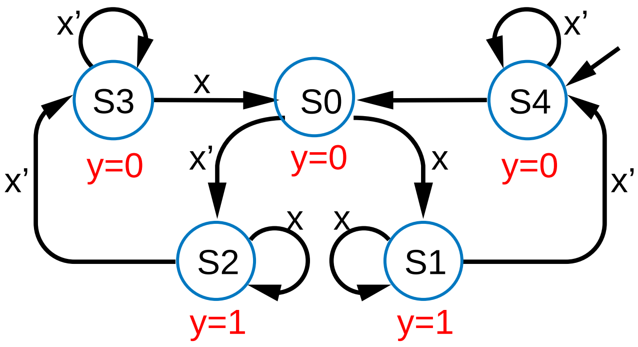
S1	✗		
S2	✗	~~(S2, S2) (S3, S1)~~	
S3	✗	~~(S0, S2) (S3, S1)~~	~~(S0, S2) (S3, S3)~~
	S0	**S1**	**S2**

a

All state pairs are marked – there are no equivalent state pairs to merge

A Larger State Reduction Example

Inputs: x; Outputs: y



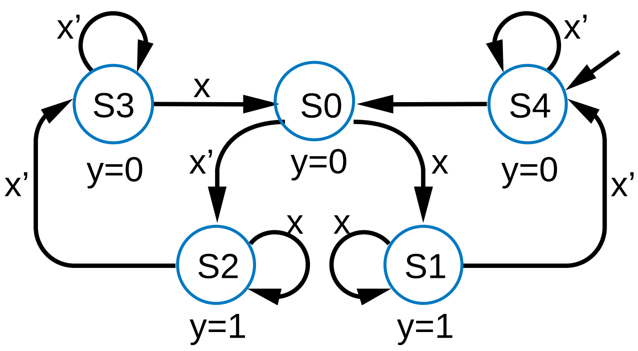
S1	✗			
S2	✗	(S3,S4) (S2,S1)		
S3	(S3,S2) (S0,S1)	✗	✗	
S4	(S4,S2) (S0,S1)	✗	✗	(S4,S3) (S0,S0)
	S0	S1	S2	S3

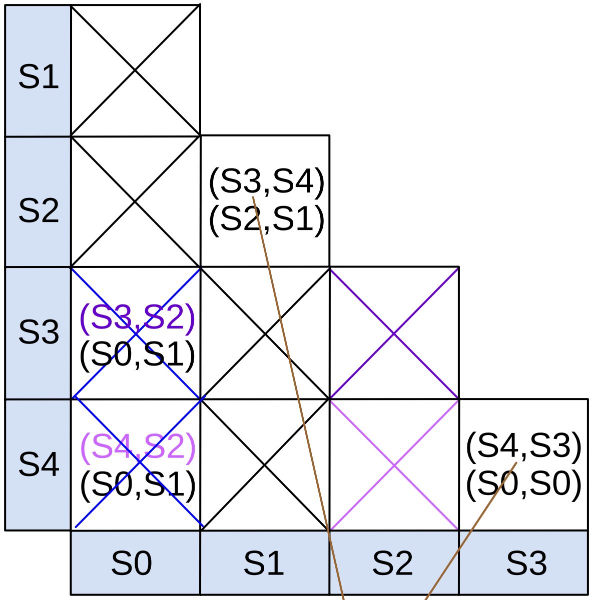
- **Step 1:** Mark state pairs having different outputs as nonequivalent
- **Step 2:** For each unmarked state pair, write the next state pairs for the same input values
- **Step 3:** For each unmarked state pair, mark state pairs having nonequivalent next state pairs as nonequivalent.
  - Repeat this step until no change occurs, or until all states are marked.
- **Step 4:** Merge remaining state pairs

a

A Larger State Reduction Example

Inputs: x; Outputs: y

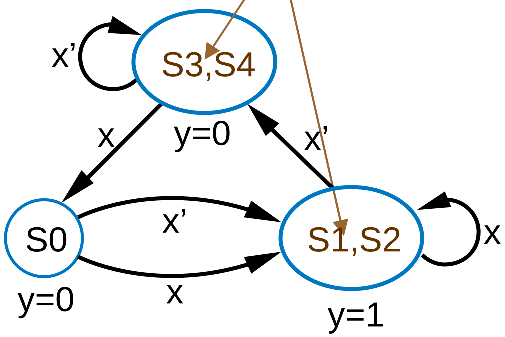




S1	☒			
S2	☒	(S3,S4) (S2,S1)		
S3	☒ (S3,S2) (S0,S1)	☒	☒	
S4	☒ (S4,S2) (S0,S1)	☒	☒	(S4,S3) (S0,S0)
	S0	S1	S2	S3

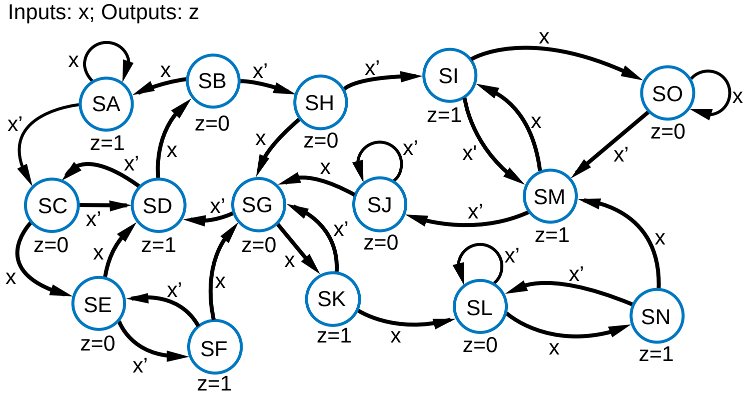
- **Step 1:** Mark state pairs having different outputs as nonequivalent
- **Step 2:** For each unmarked state pair, write the next state pairs for the same input values
- **Step 3:** For each unmarked state pair, mark state pairs having nonequivalent next state pairs as nonequivalent.
  - Repeat this step until no change occurs, or until all states are marked.
- **Step 4:** Merge remaining state pairs

Inputs: x; Outputs: y



Need for Automation

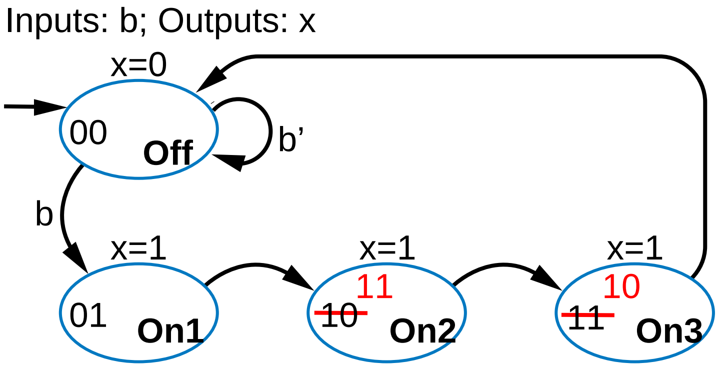
- Automation needed
  - Table for large FSM too big for humans to work with
    - *n* inputs: each state pair can have 2^n next state pairs.
    - 4 inputs → $2^4=16$ next state pairs

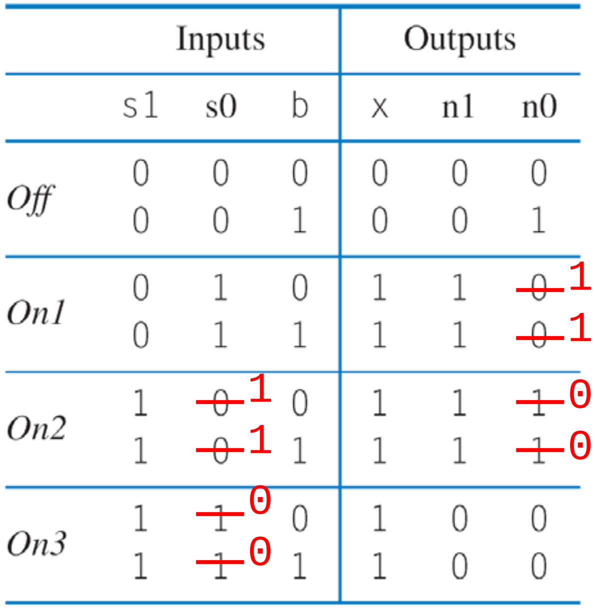


  - 100 states would have table with 100*100=100,000 state pairs cells
  - State reduction typically automated
    - Often using heuristics to reduce compute time

State Encoding

- ***Encoding***: Assigning a unique bit representation to each state
- Different encodings may optimize size, or tradeoff size and performance
- Consider 3-Cycle Laser Timer…
  - Example 3.7's encoding: **15** gate inputs
  - Try alternative encoding
    - x = s1 + s0
    - n1 = s0
    - n0 = s1'b + s1's0
    - Only **8** gate inputs





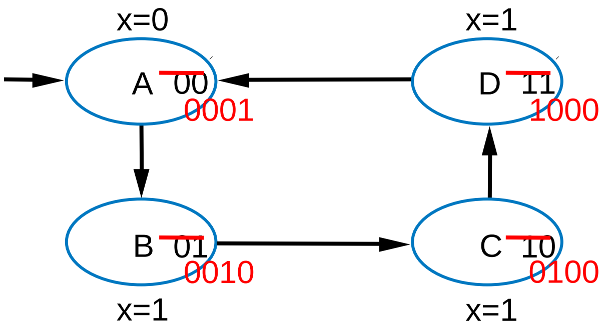
	Inputs			Outputs		
	s1	s0	b	x	n1	n0
Off	0	0	0	0	0	0
	0	0	1	0	0	1
On1	0	1	0	1	1	~~0~~ 1
	0	1	1	1	1	~~0~~ 1
On2	1	~~0~~ 1	0	1	1	~~1~~ 0
	1	~~0~~ 1	1	1	1	~~1~~ 0
On3	1	~~1~~ 0	0	1	0	0
	1	~~1~~ 0	1	1	0	0

a

State Encoding: One-Hot Encoding

- ***One-hot encoding***
  - One bit per state – a bit being ‘1’ corresponds to a particular state
  - Alternative to *minimum bit-width encoding* in previous example
  - For A, B, C, D: A: 0001, B: 0010, C: 0100, D: 1000
- Example: FSM that outputs 0, 1, 1, 1
  - Equations if one-hot encoding:
    - n3 = s2; n2 = s1; n1 = s0; x = s3 + s2 + s1
  - Fewer gates and only one level of logic – less delay than two levels, so faster clock frequency

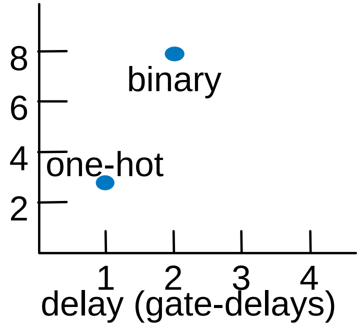
Inputs: none; Outputs: x

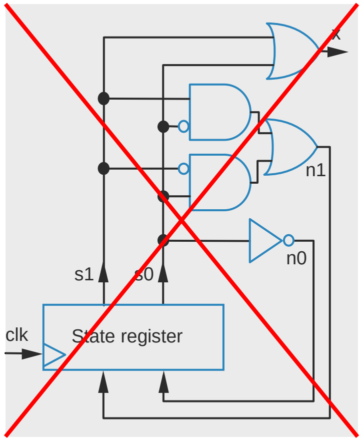


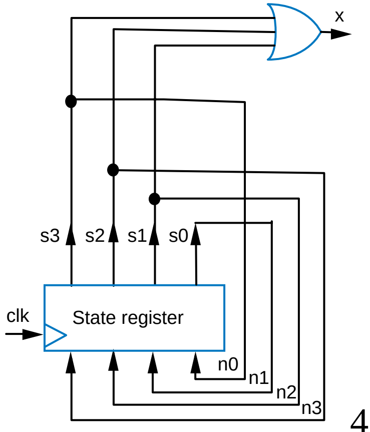
a

	Inputs		Outputs		
	~~s1~~	~~s0~~	~~n1~~	~~n0~~	~~x~~
~~*A*~~	~~0~~	~~0~~	~~0~~	~~1~~	~~0~~
~~*B*~~	~~0~~	~~1~~	~~1~~	~~0~~	~~1~~
~~*C*~~	~~1~~	~~0~~	~~1~~	~~1~~	~~1~~
~~*D*~~	~~1~~	~~1~~	~~0~~	~~0~~	~~1~~

	Inputs				Outputs				
	s3	s2	s1	s0	n3	n2	n1	n0	x
A	0	0	0	1	0	0	1	0	0
B	0	0	1	0	0	1	0	0	1
C	0	1	0	0	1	0	0	0	1
D	1	0	0	0	0	0	0	1	1

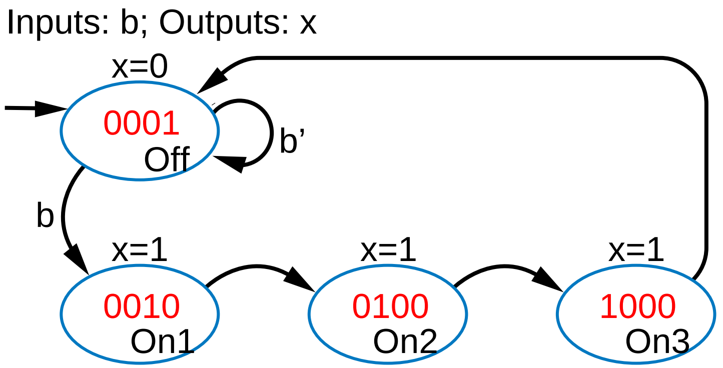


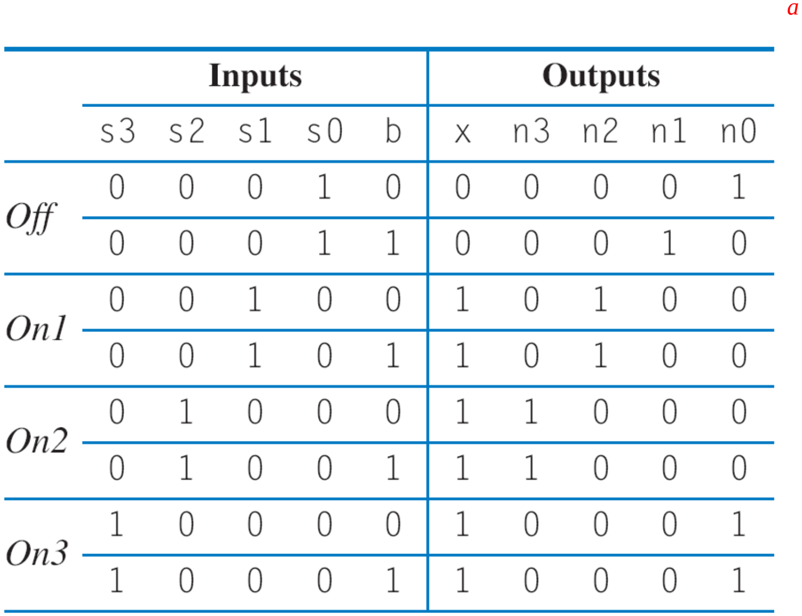




One-Hot Encoding Example: Three-Cycles-High Laser Timer

- Four states – Use four-bit one-hot encoding
  - State table leads to equations:
    - x = s3 + s2 + s1
    - n3 = s2
    - n2 = s1
    - n1 = s0*b
    - n0 = s0*b’ + s3
  - Smaller
    - 3+0+0+2+(2+2) = **9** gate inputs
    - Earlier binary encoding (Ch 3): **15** gate inputs
  - Faster
    - Critical path: n0 = s0*b’ + s3
    - Previously: n0 = s1’s0’b + s1s0’
    - 2-input AND slightly faster than 3-input AND

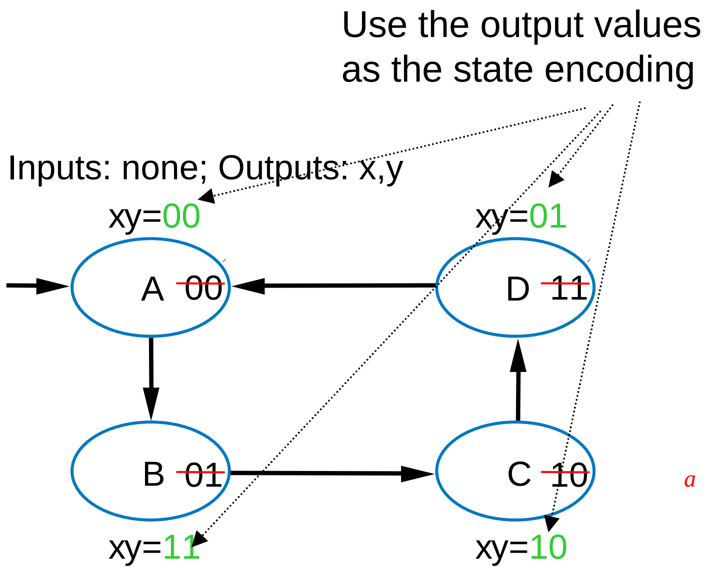




	Inputs					Outputs				
	s3	s2	s1	s0	b	x	n3	n2	n1	n0
Off	0	0	0	1	0	0	0	0	0	1
	0	0	0	1	1	0	0	0	1	0
On1	0	0	1	0	0	1	0	1	0	0
	0	0	1	0	1	1	0	1	0	0
On2	0	1	0	0	0	1	1	0	0	0
	0	1	0	0	1	1	1	0	0	0
On3	1	0	0	0	0	1	0	0	0	1
	1	0	0	0	1	1	0	0	0	1

Output Encoding

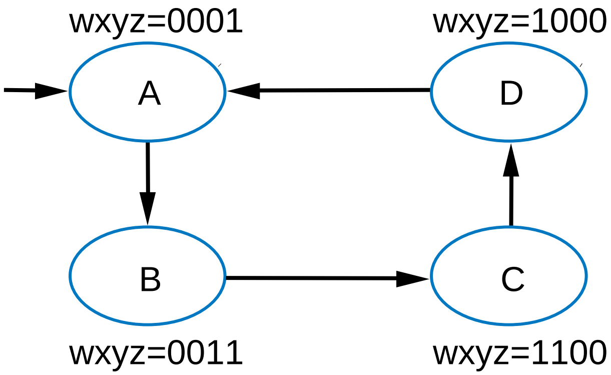
- ***Output encoding***: Encoding method where the state encoding is same as the output values
  - Possible if enough outputs, all states with unique output values

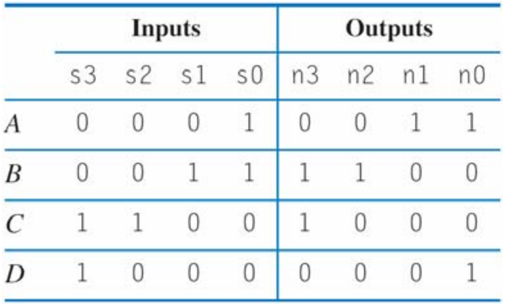


Use the output values as the state encoding
Inputs: none; Outputs: x,y
xy=00
A ~~00~~
xy=01
D ~~11~~
B ~~01~~
C ~~10~~
xy=11
xy=10
a

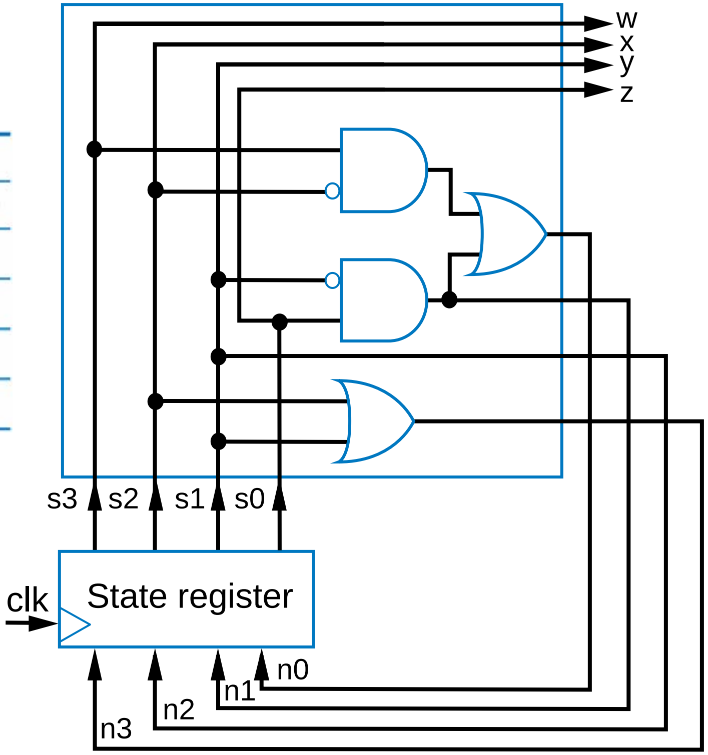
Output Encoding Example: Sequence Generator

Inputs: none; Outputs: w, x, y, z



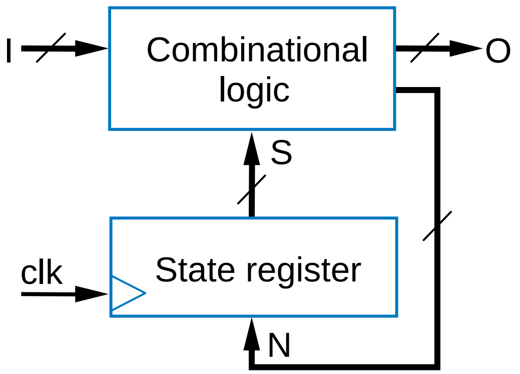


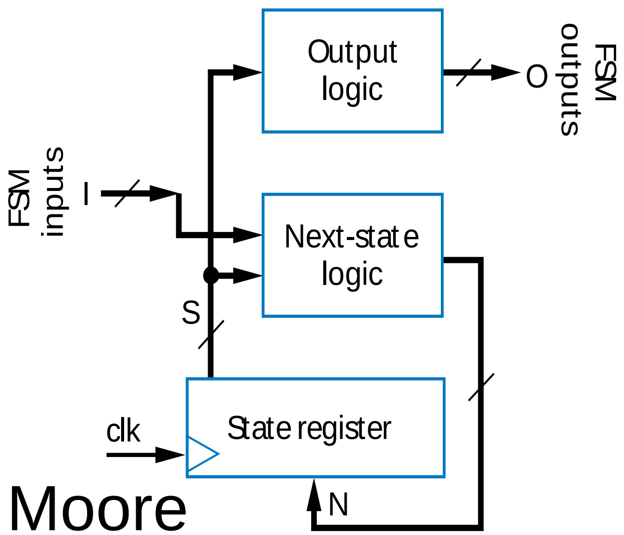
	Inputs				Outputs			
	s3	s2	s1	s0	n3	n2	n1	n0
A	0	0	0	1	0	0	1	1
B	0	0	1	1	1	1	0	0
C	1	1	0	0	1	0	0	0
D	1	0	0	0	0	0	0	1



- Generate sequence 0001, 0011, 1110, 1000, repeat
  - FSM shown
- Use output values as state encoding
- Create state table
- Derive equations for next state
  - n3 = s1 + s2; n2 = s1; n1 = s1’s0; n0 = s1’s0 + s3s2’

Moore vs. Mealy FSMs

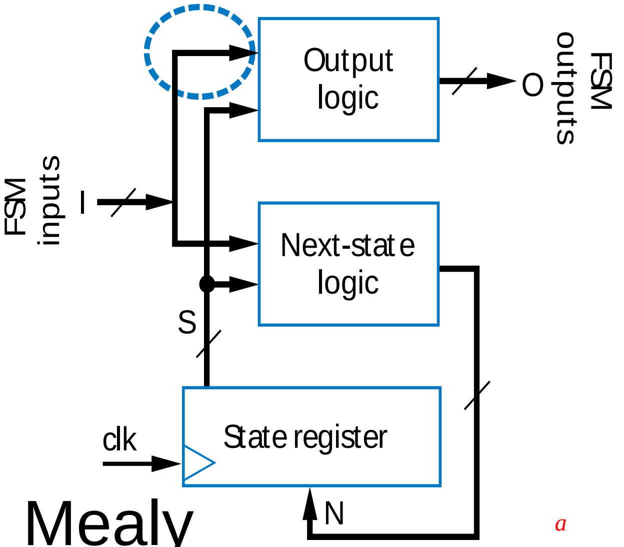




Moore

(a)

Mealy FSM adds this

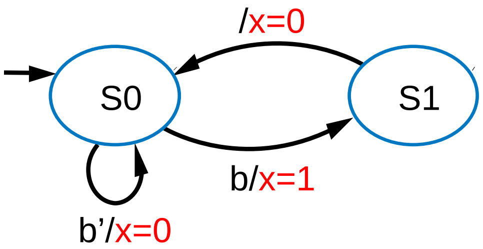


Mealy

(b)

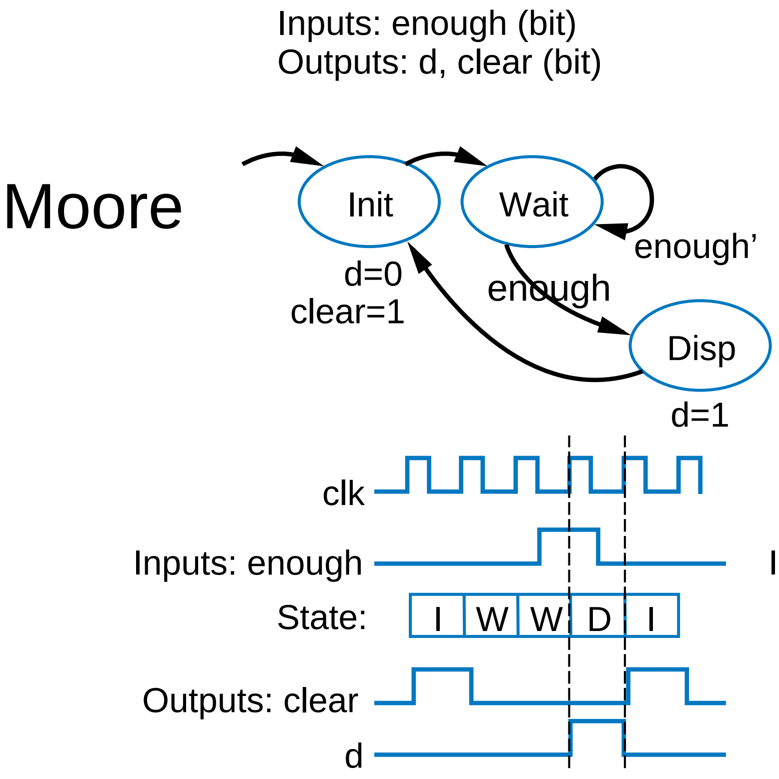
- FSM implementation architecture
  - State register and logic
  - More detailed view
    - Next state logic – function of present state and FSM inputs
    - Output logic
      - If function of present state only – ***Moore FSM***
      - If function of present state and FSM inputs – ***Mealy FSM***

Inputs: b; Outputs: x



Graphically: show outputs with arcs, not with states

Mealy FSMs May Have Fewer States



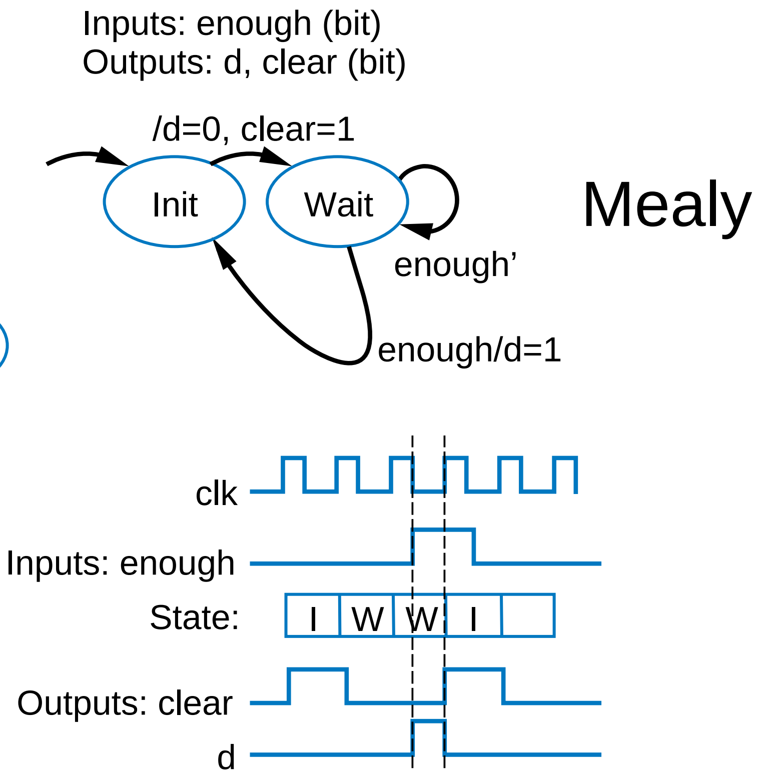
Inputs: enough (bit)
Outputs: d, clear (bit)

Moore

Init
Wait
Disp
enough'
enough
d=0
clear=1
d=1

clk
Inputs: enough
State: I W W D I
Outputs: clear
d

(a)



Inputs: enough (bit)
Outputs: d, clear (bit)

Mealy

/d=0, clear=1
Init
Wait
enough'
enough/d=1

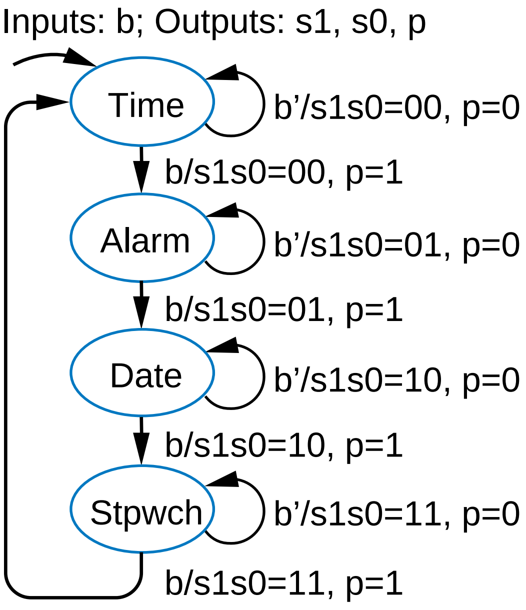
clk
Inputs: enough
State: I W W I
Outputs: clear
d

(b)

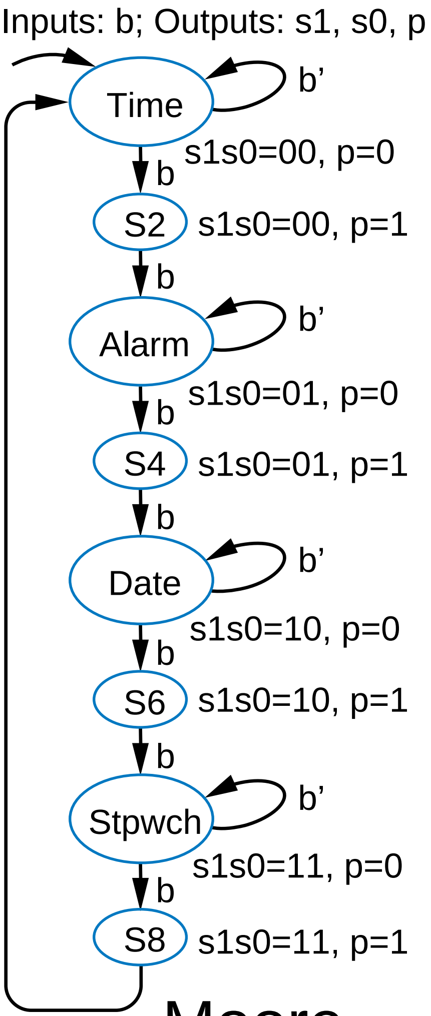
- Soda dispenser example: Initialize, wait until enough, dispense
  - Moore: 3 states; Mealy: 2 states

Mealy vs. Moore

- Q: Which is Moore, and which is Mealy?
- A: Mealy on left, Moore on right
  - Mealy outputs on arcs, meaning outputs are function of state AND INPUTS
  - Moore outputs in states, meaning outputs are function of state only



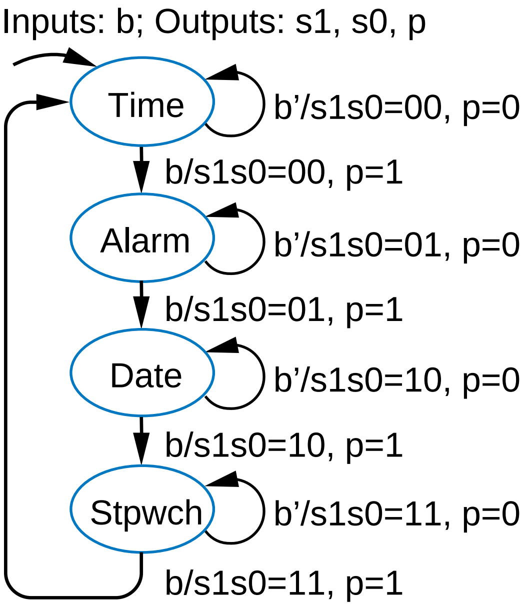
Mealy



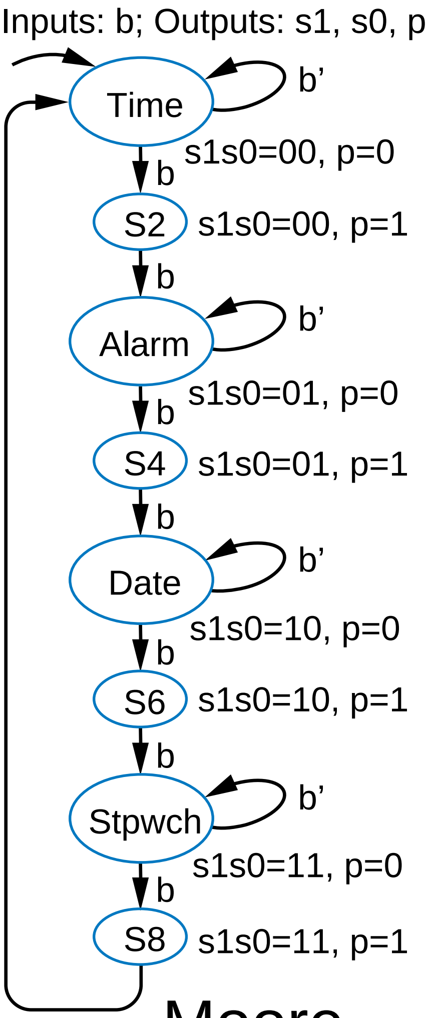
Moore

Mealy vs. Moore Example: Beeping Wristwatch

- Button b
  - Sequences mux select lines *s1s0* through 00, 01, 10, and 11
    - Each value displays different internal register
  - Each unique button press should cause 1-cycle beep, with *p*=1 being beep
- Must wait for button to be released (*b'*) and pushed again (*b*) before sequencing
    - Note that Moore requires unique state to pulse *p*, while Mealy pulses *p* on arc
    - Tradeoff: Mealy's pulse on *p* may not last one full cycle



Mealy



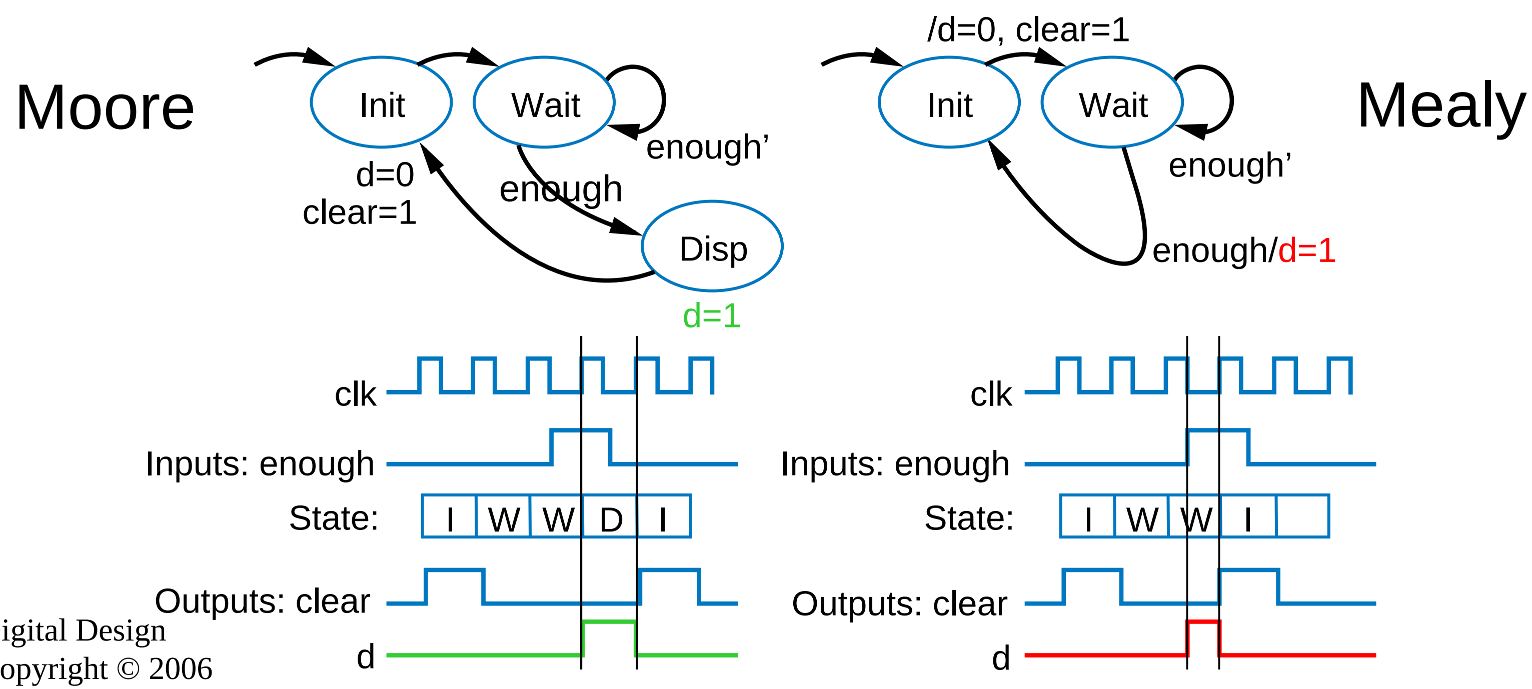
Moore

Mealy vs. Moore Tradeoff

- Mealy outputs change mid-cycle if input changes
  - Note earlier soda dispenser example
    - Mealy had fewer states, but output *d* not 1 for full cycle
  - Represents a type of tradeoff

Inputs: enough (bit)
Outputs: d, clear (bit)

Inputs: enough (bit)
Outputs: d, clear (bit)



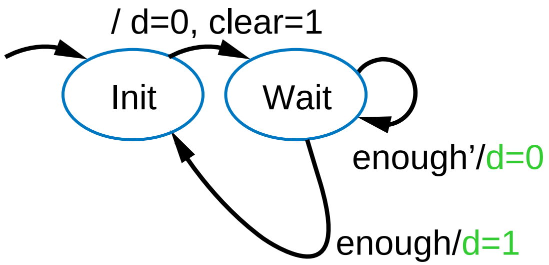
(a)

(b)

Implementing a Mealy FSM

- Straightforward
  - Convert to state table
  - Derive equations for each output
  - Key difference from Moore: External outputs (*d*, *clear*) may have different value in same state, depending on input values

Inputs: enough (bit)
Outputs: d, clear (bit)



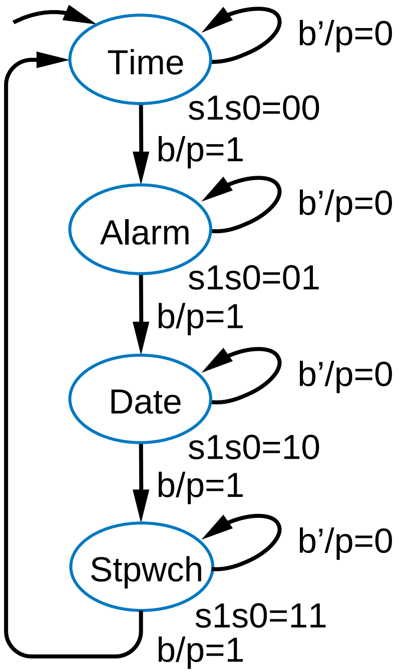
	Inputs		Outputs		
	s0	enough	n0	d	clear
Init	0	0	1	0	1
	0	1	1	0	1
Wait	1	0	1	0	0
	1	1	0	1	0

Digital Design
Copyright © 2006
Frank Vahid

Meały and Moore can be Combined

- Final note on Mealy/Moore
  - May be combined in same FSM

Inputs: b; Outputs: s1, s0, p



Combined Moore/Mealy FSM for beeping wristwatch example

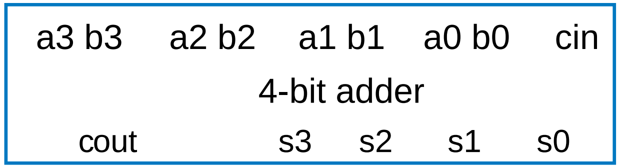
6.4

Datapath Component Tradeoffs

- Can make some components faster (but bigger), or smaller (but slower), than the straightforward components we built in Ch 4
- We’ll build
  - A faster (but bigger) adder than the carry-ripple adder
  - A smaller (but slower) multiplier than the array-based multiplier
- Could also do for the other Ch 4 components

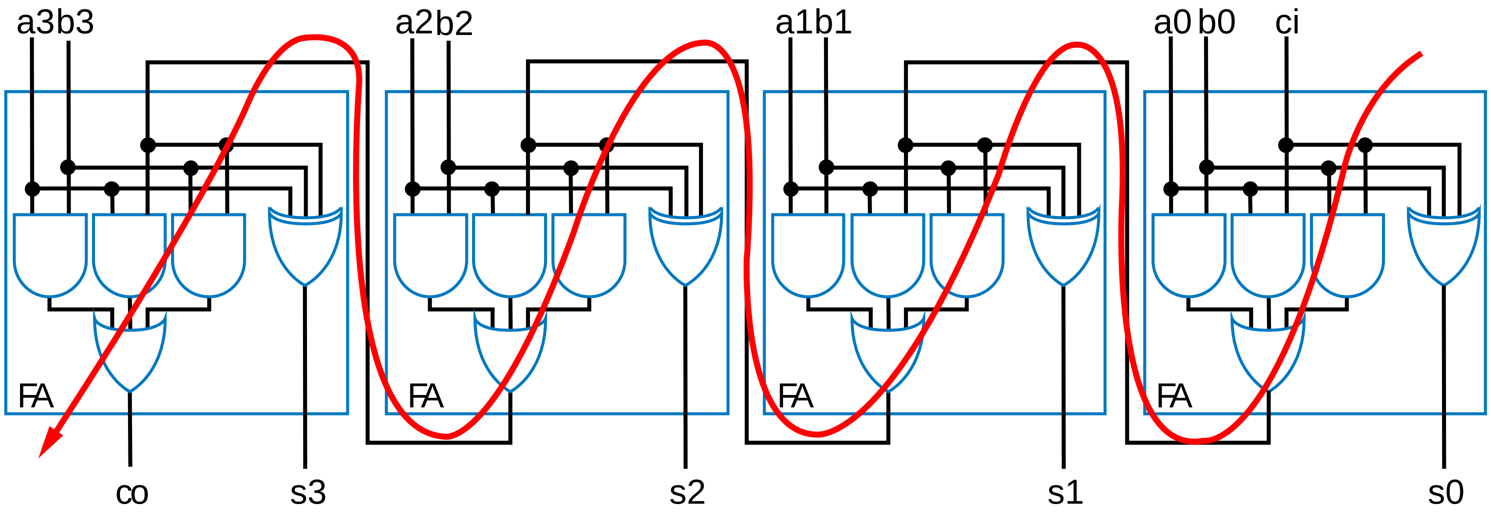
Faster Adder

- Built carry-ripple adder in Ch 4
  - Similar to adding by hand, column by column
  - Con: Slow
    - Output is not correct until the carries have rippled to the left
    - 4-bit carry-ripple adder has 4*2 = 8 gate delays
  - Pro: Small
    - 4-bit carry-ripple adder has just 4*5 = 20 gates



carries:		c3	c2	c1	cin
B:		b3	b2	b1	b0
A:	+	a3	a2	a1	a0
	cout	s3	s2	s1	s0

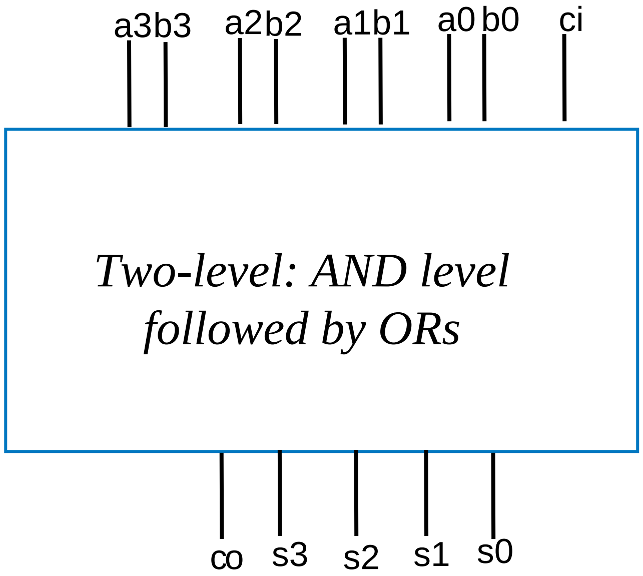
a

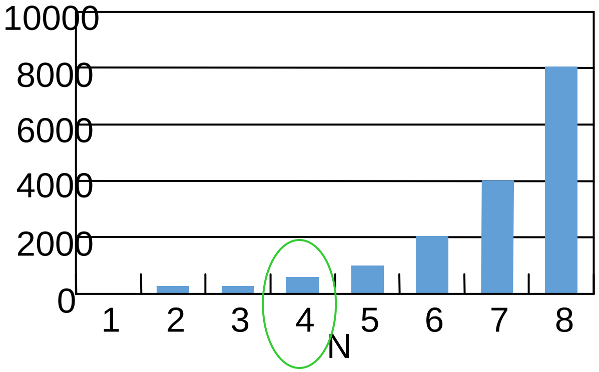


a

Faster Adder

- Faster adder – Use two-level combinational logic design process
  - Recall that 4-bit two-level adder was big
  - Pro: Fast
    - 2 gate delays
  - Con: Large
    - Truth table would have $2^{(4+4)}$ =256 rows
    - Plot shows 4-bit adder would use about 500 gates
- Is there a compromise design?
  - Between 2 and 8 gate delays
  - Between 20 and 500 gates

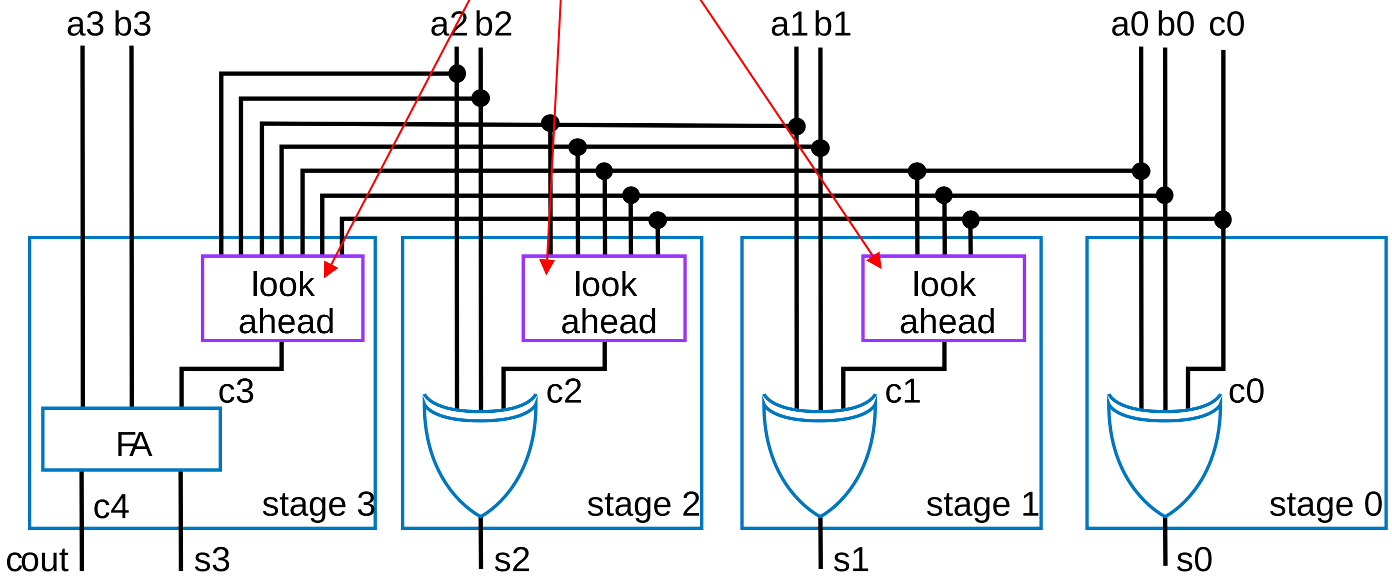




Digital Design
Copyright © 2006
Frank Vahid

Faster Adder – (Bad) Attempt at “Lookahead”

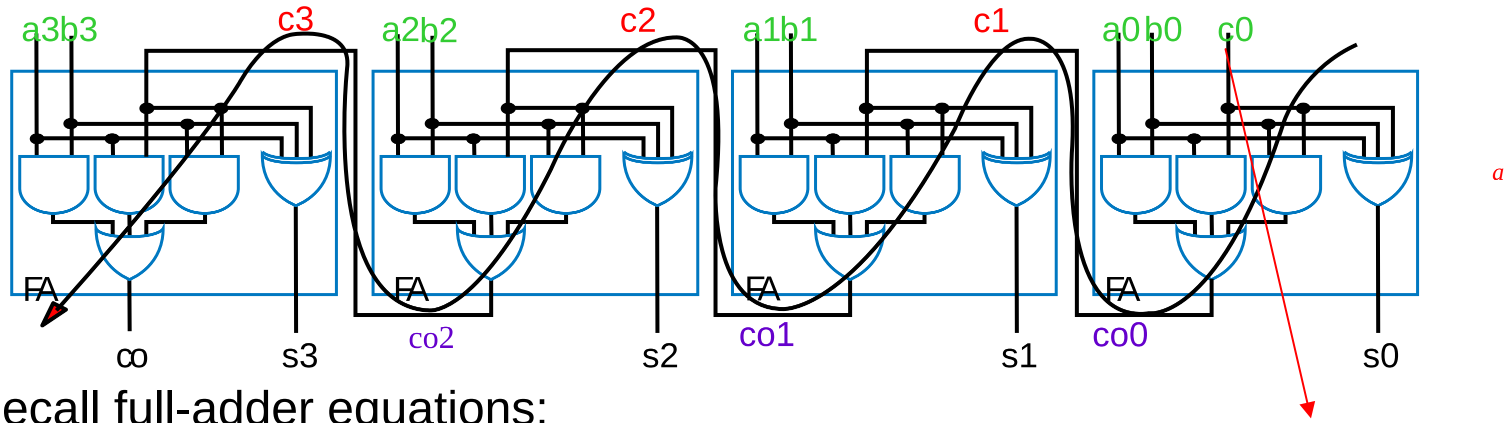
- Idea
  - Modify carry-ripple adder – For a stage’s carry-in, don’t wait for carry to ripple, but rather *directly compute* from inputs of earlier stages
    - Called “lookahead” because current stage “looks ahead” at previous stages rather than waiting for carry to ripple to current stage



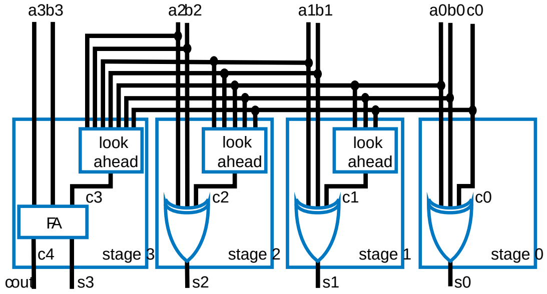
Notice – no rippling of carry

Faster Adder – (Bad) Attempt at “Lookahead”

- Want each stage’s carry-in bit to be function of external inputs only (*a*’s, *b*’s, or *c0*)



- Recall full-adder equations:
  - s = a xor b
  - c = bc + ac + ab



Stage 0: Carry-in is already an external input: **c0**

Stage 1: c1=co0

co0= b0c0 + a0c0 + a0b0

c1 = b0c0 + a0c0 + a0b0

Stage 2: c2=co1

co1 = b1c1 + a1c1 + a1b1

c2 = b1c1 + a1c1 + a1b1

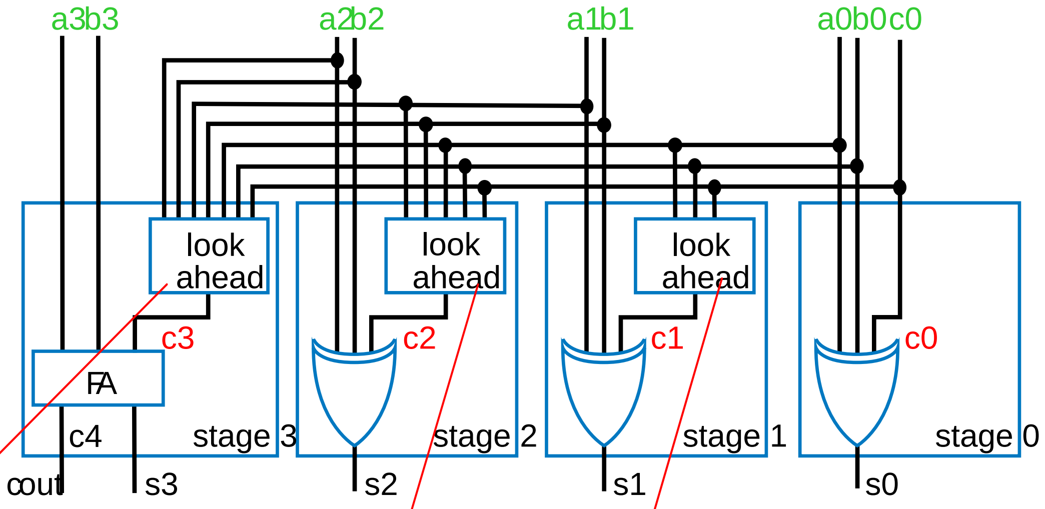
c2 = b1(b0c0 + a0c0 + a0b0) + a1(b0c0 + a0c0 + a0b0) +a1b1

c2 = b1b0c0 + b1a0c0 + b1a0b0 + a1b0c0 + a1a0c0 + a1a0b0 + a1b1

Continue for c3

Faster Adder – (Bad) Attempt at “Lookahead”

- Carry lookahead logic function of external inputs
  - No waiting for ripple
- Problem
  - Equations get too big
  - Not efficient
  - Need a better form of lookahead



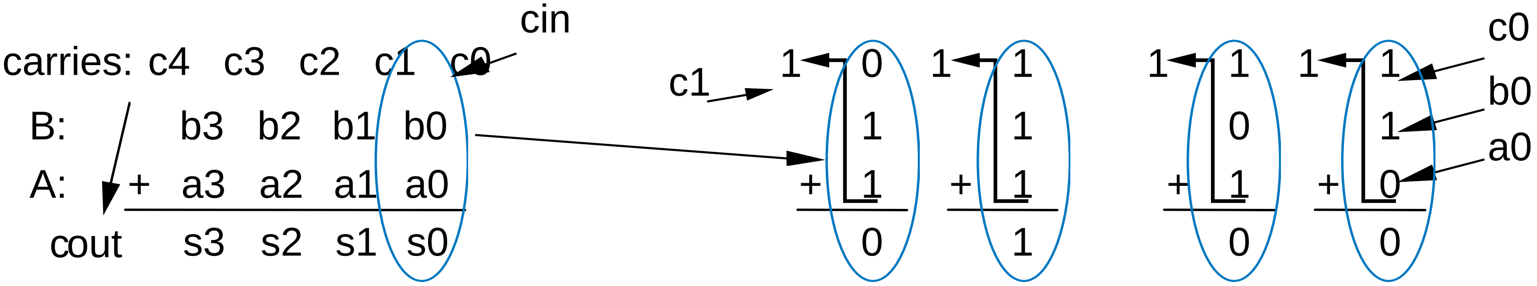
c1 = b0c0 + a0c0 + a0b0

c2 = b1b0c0 + b1a0c0 + b1a0b0 + a1b0c0 + a1a0c0 + a1a0b0 + a1b1

c3 = b2b1b0c0 + b2b1a0c0 + b2b1a0b0 + b2a1b0c0 + b2a1a0c0 + b2a1a0b0 + b2a1b1 + a2b1b0c0 + a2b1a0c0 + a2b1a0b0 + a2a1b0c0 + a2a1a0c0 + a2a1a0b0 + a2a1b1 + a2b2

Better Form of Lookahead

- Have each stage compute two terms
  - ***Propagate***: P = a xor b
  - ***Generate***: G = ab
- Compute lookahead from *P* and *G* terms, *not from external inputs*
  - Why *P* & *G*? Because the logic comes out much simpler
    - Very clever finding; not particularly obvious though
    - Why those names?
      - *G*: If *a* and *b* are 1, carry-out will be 1 – “generate” a carry-out of 1 in this case
      - *P*: If only one of *a* or *b* is 1, then carry-out will equal the carry-in – propagate the carry-in to the carry-out in this case

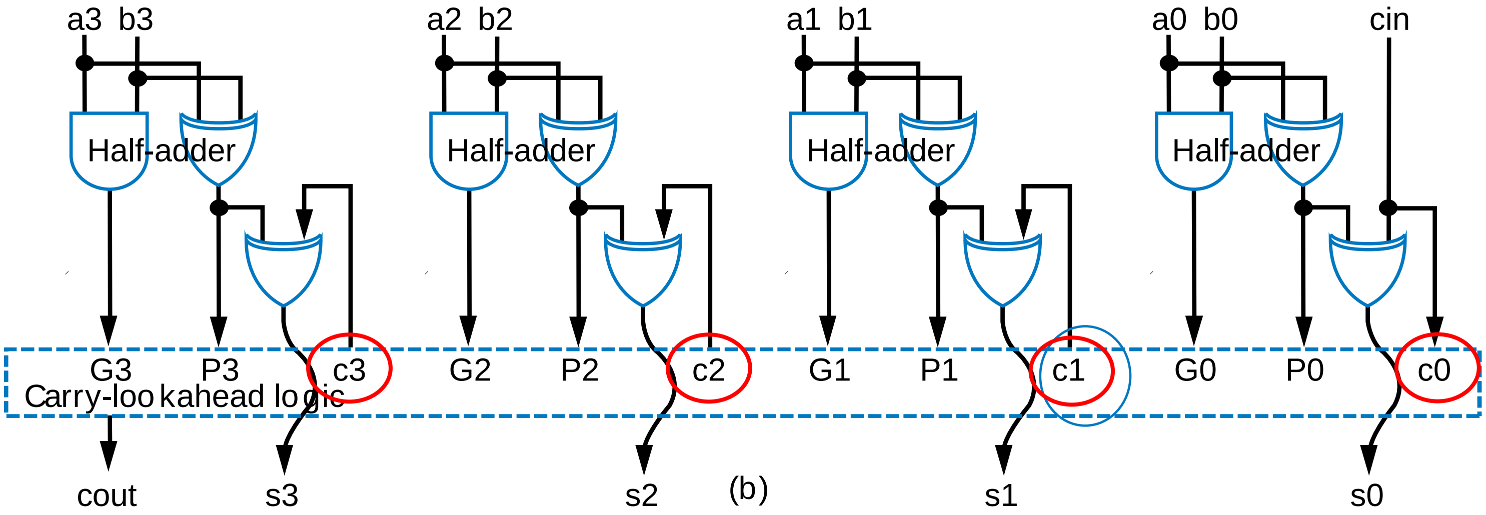


(a)

if a0b0 = 1
then c1 = 1
(call this G:Generate)

if a0xor b0 = 1
then c1 = 1 if c0 = 1
(call this P: Propagate)

Better Form of Lookahead



a

- With *P* & *G*, the carry lookahead equations are much simpler
  - Equations before plugging in
    - c1 = G0 + P0c0
    - c2 = G1 + P1c1
    - c3 = G2 + P2c2
    - cout = G3 + P3c3

After plugging in:

c1 = G0 + P0c0

a

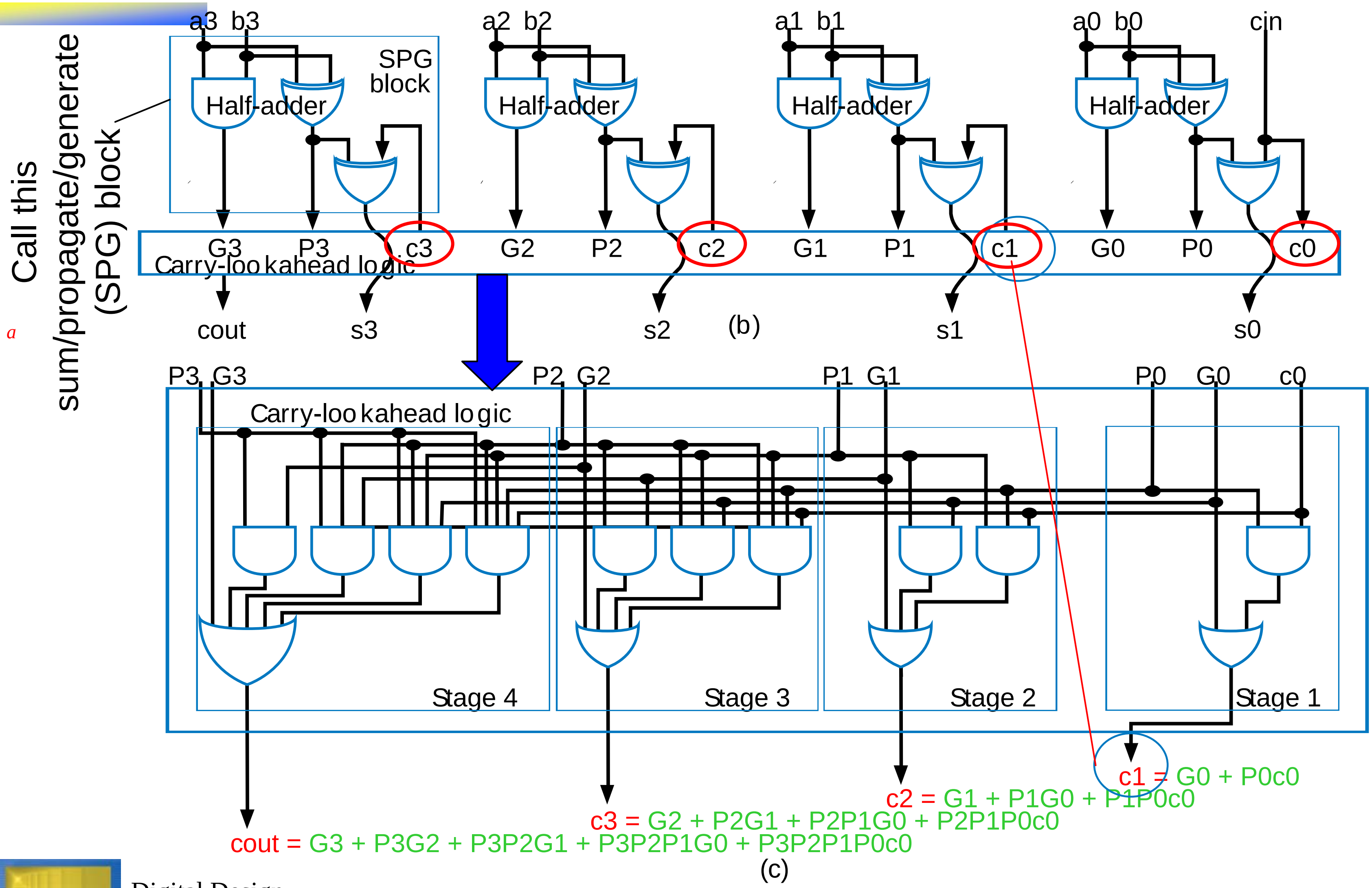
c2 = G1 + P1c1 = G1 + P1(G0 + P0c0)
c2 = G1 + P1G0 + P1P0c0

c3 = G2 + P2c2 = G2 + P2(G1 + P1G0 + P1P0c0)
c3 = G2 + P2G1 + P2P1G0 + P2P1P0c0

cout = G3 + P3G2 + P3P2G1 + P3P2P1G0 + P3P2P1P0c0

Much simpler than the “bad” lookahead

Better Form of Lookahead



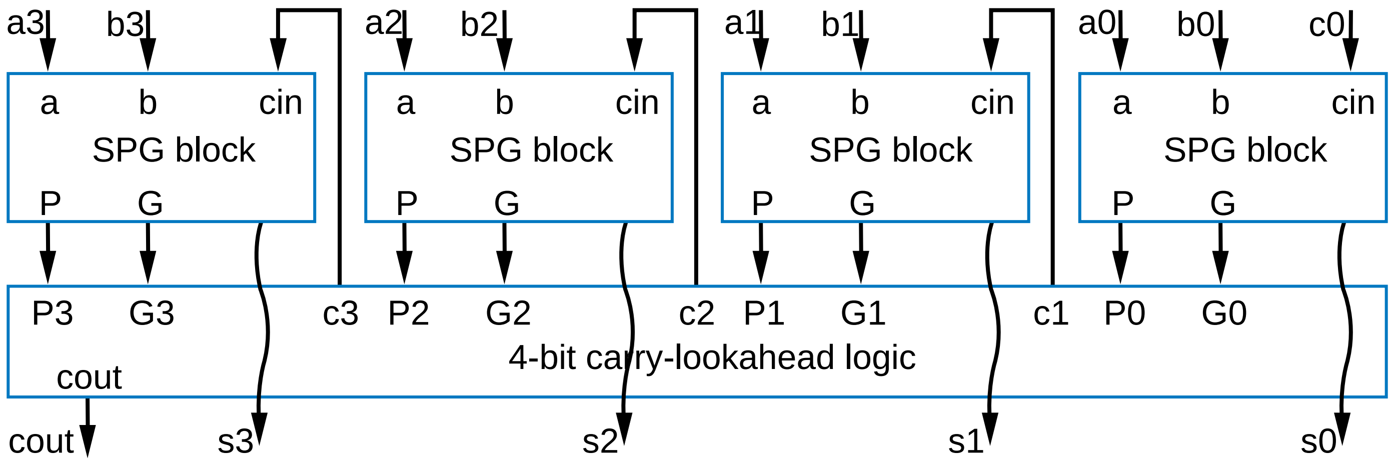
c1 = G0 + P0c0

c2 = G1 + P1G0 + P1P0c0

c3 = G2 + P2G1 + P2P1G0 + P2P1P0c0

cout = G3 + P3G2 + P3P2G1 + P3P2P1G0 + P3P2P1P0c0

Carry-Lookahead Adder -- High-Level View

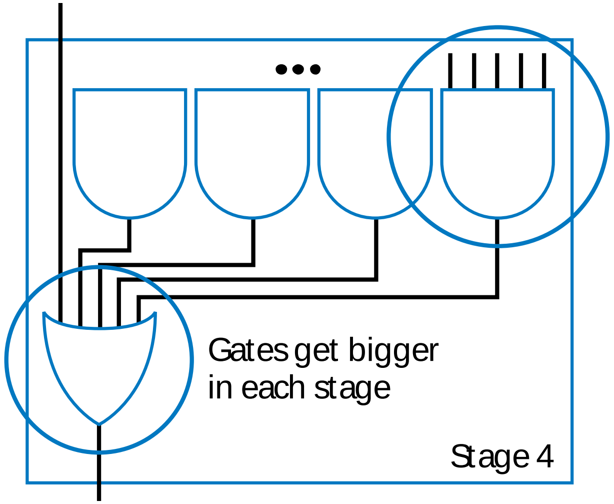


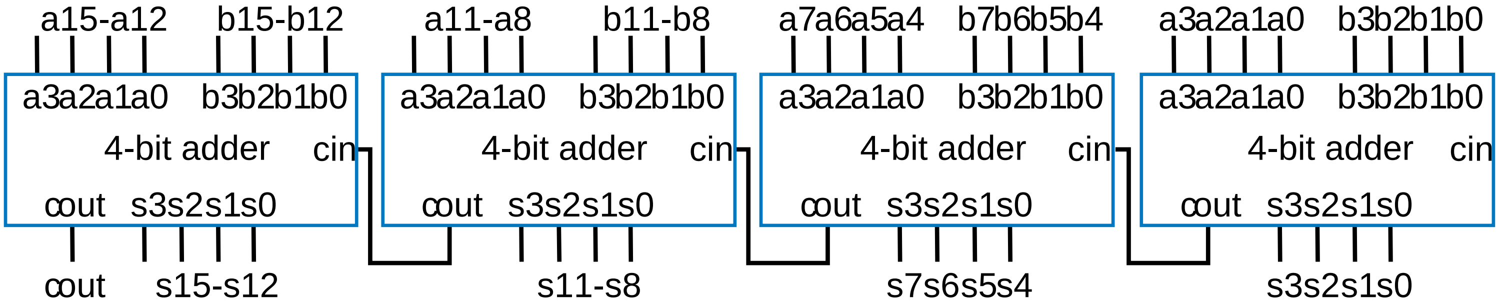
- Fast -- only 4 gate delays
  - Each stage has SPG block with 2 gate levels
  - Carry-lookahead logic quickly computes the carry from the propagate and generate bits using 2 gate levels inside
- Reasonable number of gates -- 4-bit adder has only 26 gates

- 4-bit adder comparison (gate delays, gates)
  - Carry-ripple: (8, 20)
  - Two-level: (2, 500)
  - CLA: (4, 26)
    - o Nice compromise

Carry-Lookahead Adder – 32-bit?

- Problem: Gates get bigger in each stage
  - 4th stage has 5-input gates
  - 32nd stage would have 33-input gates
    - Too many inputs for one gate
    - Would require building from smaller gates, meaning more levels (slower), more gates (bigger)
- One solution: Connect 4-bit CLA adders in ripple manner
  - But slow (4 + 4 + 4 + 4 gate delays)

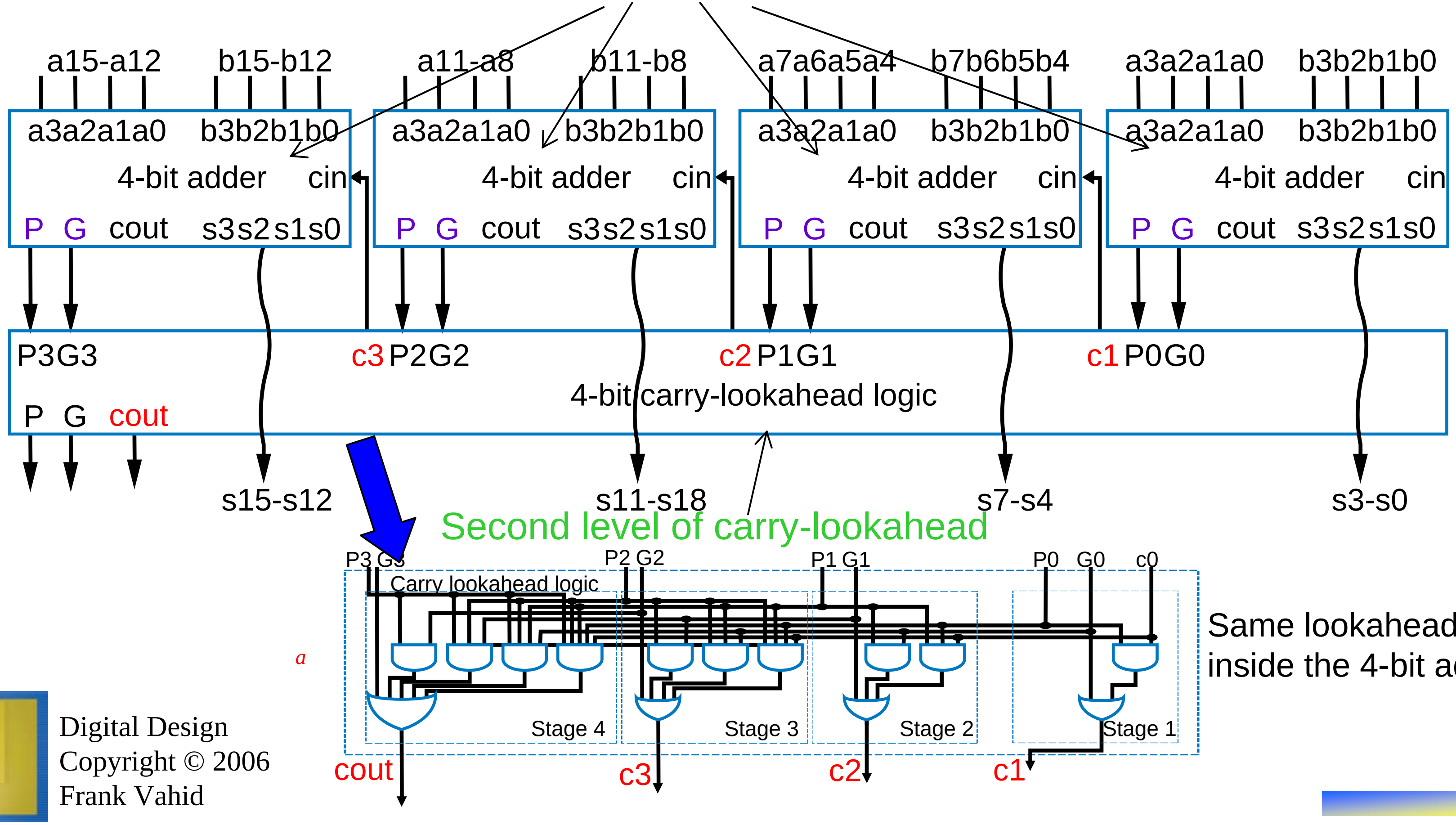




Hierarchical Carry-Lookahead Adders

- Better solution -- Rather than rippling the carries, just *repeat* the carry-lookahead concept
  - Requires minor modification of 4-bit CLA adder to output P and G

These use carry-lookahead internally

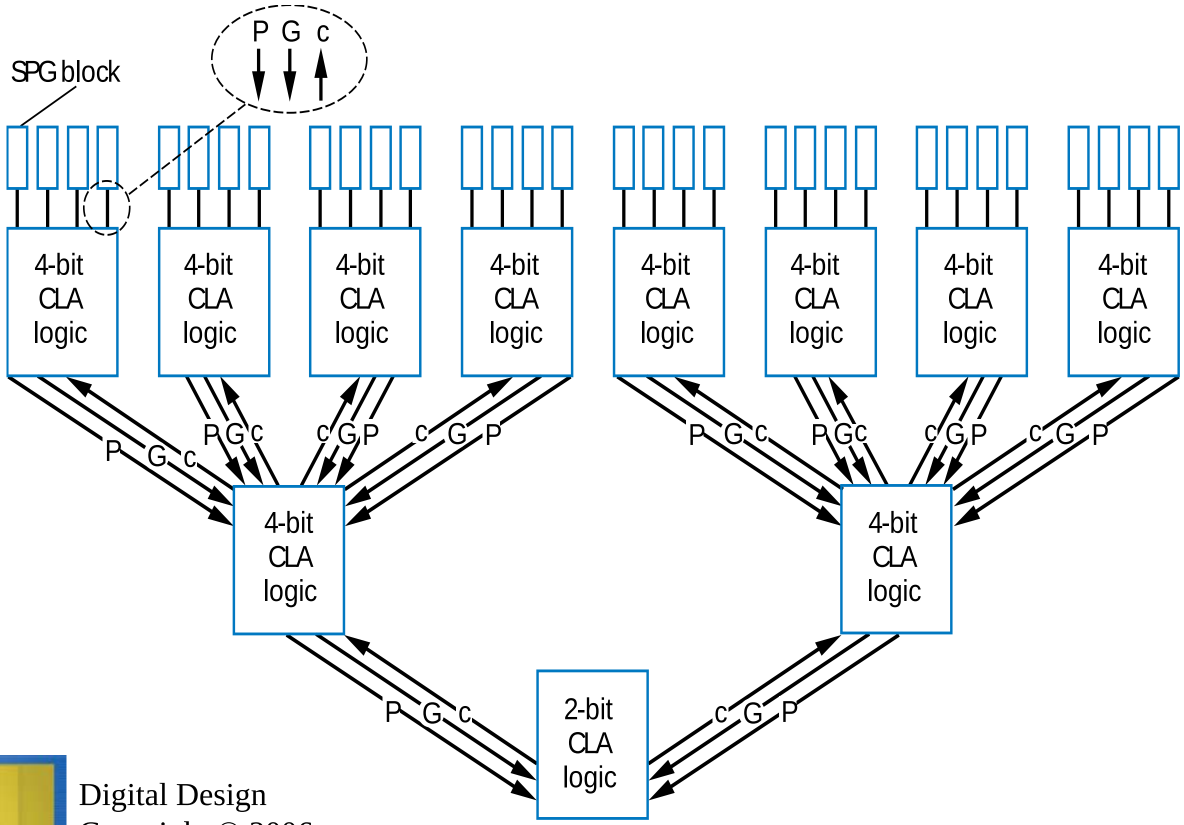


Second level of carry-lookahead

Same lookahead logic as inside the 4-bit adders

Hierarchial Carry-Lookahead Adders

- Hierarchical CLA concept can be applied for larger adders
- 32-bit hierarchical CLA
  - Only about 8 gate delays (2 for SPG block, then 2 per CLA level)
  - Only about 14 gates in each 4-bit CLA logic block



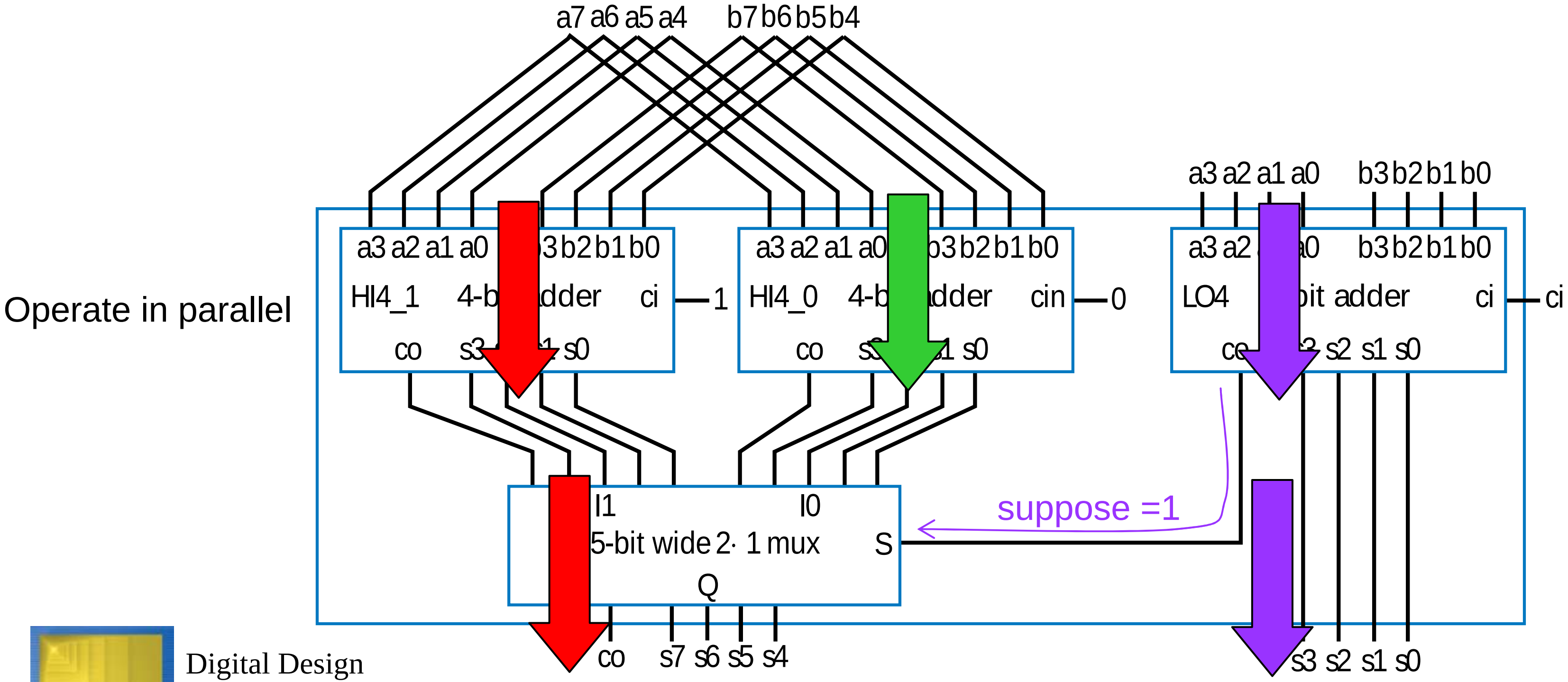
a

Q: How many gate delays for 64-bit hierarchical CLA, using 4-bit CLA logic?

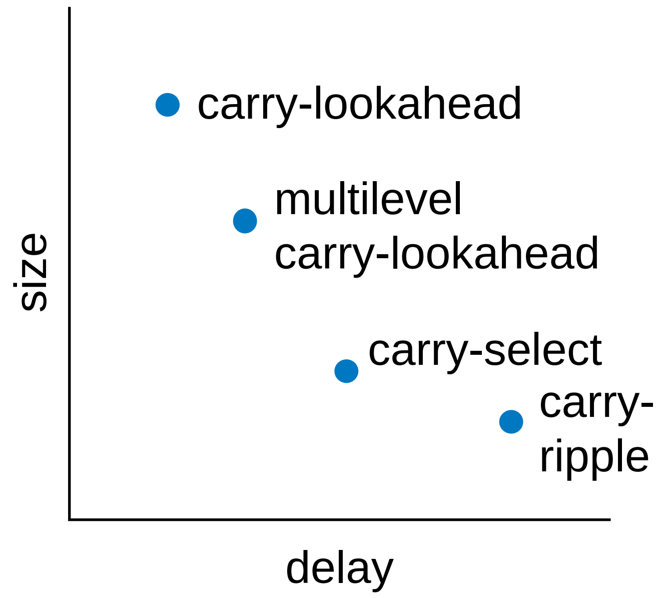
A: 16 CLA-logic blocks in 1st level, 4 in 2nd, 1 in 3rd -- so still just 8 gate delays (2 for SPG, and 2+2+2 for CLA logic). CLA is a very efficient method.

Carry Select Adder

- Another way to compose adders
  - High-order stage -- Compute result for carry in of 1 and of 0
    - Select based on carry-out of low-order stage
    - Faster than pure rippling



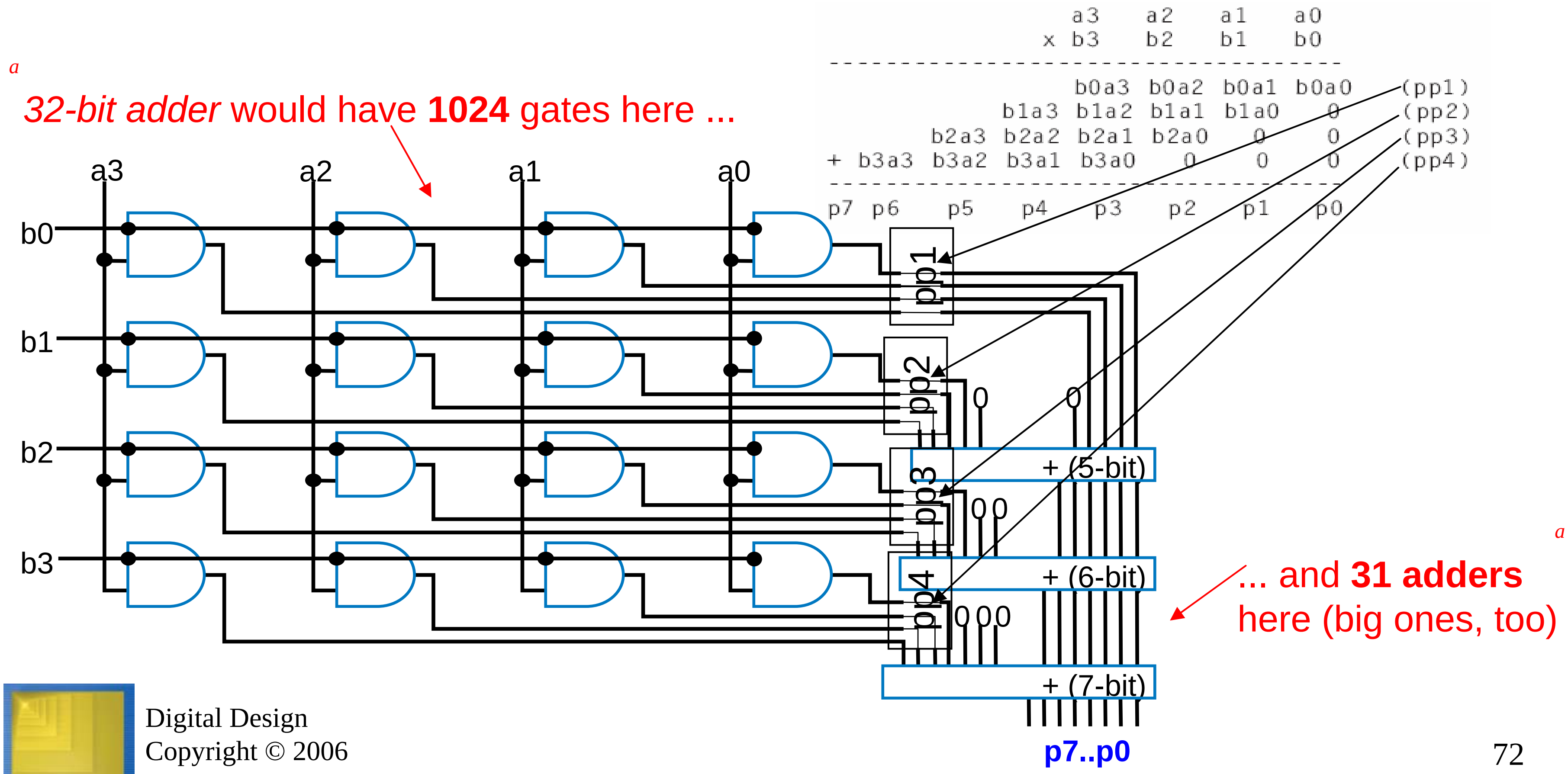
Adder Tradeoffs



- Designer picks the adder that satisfies particular delay and size requirements
  - May use different adder types in different parts of same design
    - Faster adders on critical path, smaller adders on non-critical path

Smaller Multiplier

- Multiplier in Ch 4 was array style
  - Fast, reasonable size for 4-bit: 4*4 = 16 partial product AND terms, 3 adders
  - Rather big for 32-bit: 32*32 = 1024 AND terms, and 31 adders



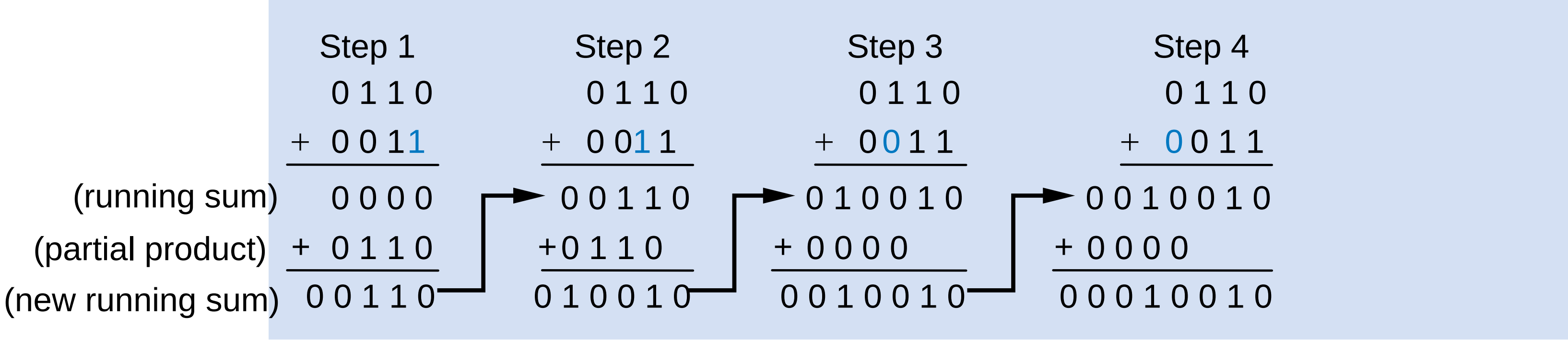
```
                  a3    a2    a1    a0
               x  b3    b2    b1    b0
     --------------------------------------
                  b0a3  b0a2  b0a1  b0a0   (pp1)
            b1a3  b1a2  b1a1  b1a0  0      (pp2)
      b2a3  b2a2  b2a1  b2a0  0     0      (pp3)
+ b3a3  b3a2  b3a1  b3a0  0     0     0    (pp4)
     --------------------------------------
p7  p6    p5    p4    p3    p2    p1    p0
```

32-bit adder would have **1024** gates here ...

... and **31 adders** here (big ones, too)

Smaller Multiplier -- Sequential (Add-and-Shift) Style

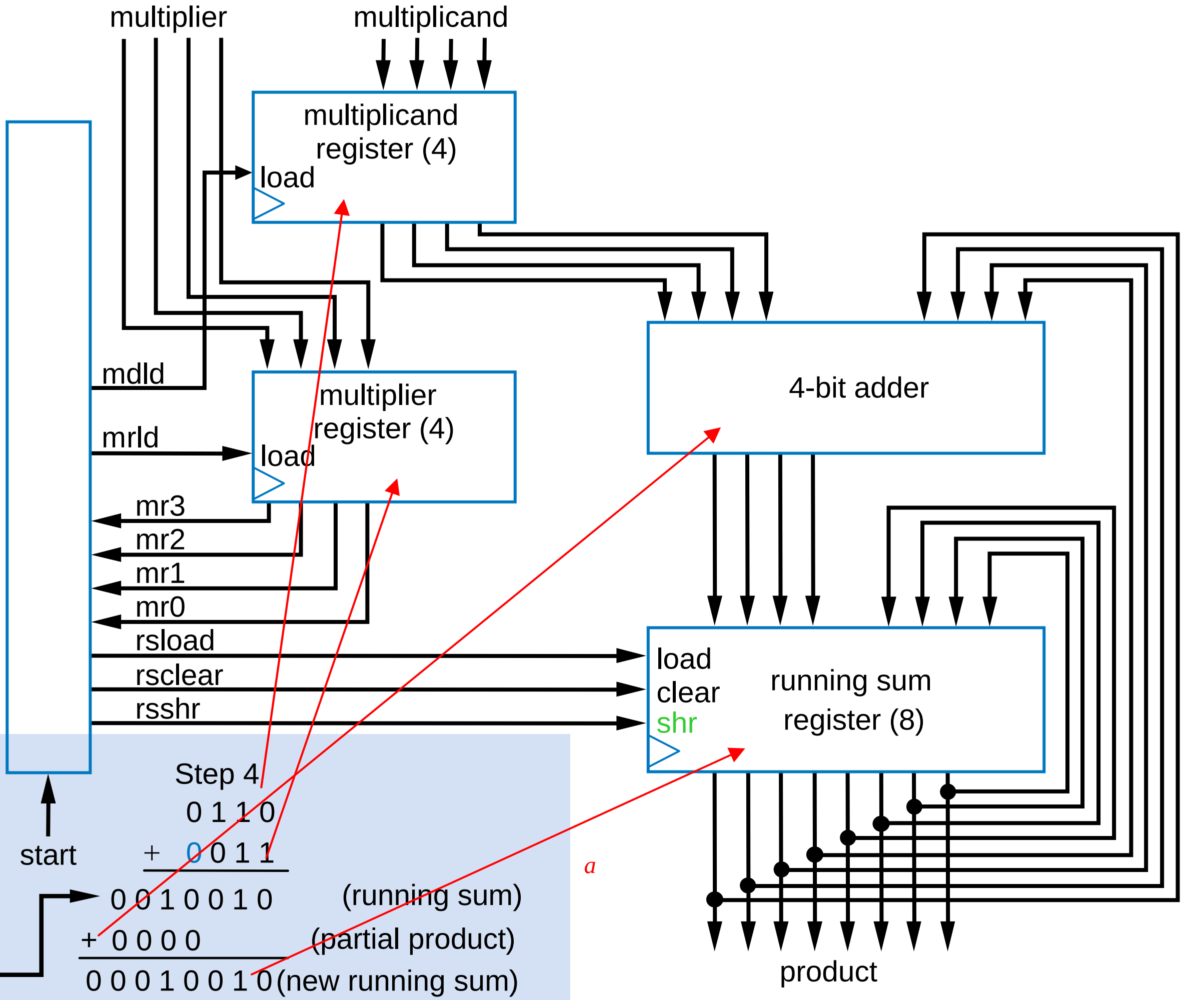
- Smaller multiplier: Basic idea
  - Don’t compute all partial products simultaneously
  - Rather, compute one at a time (similar to by hand), maintain running sum



a

Smaller Multiplier -- Sequential (Add-and-Shift) Style

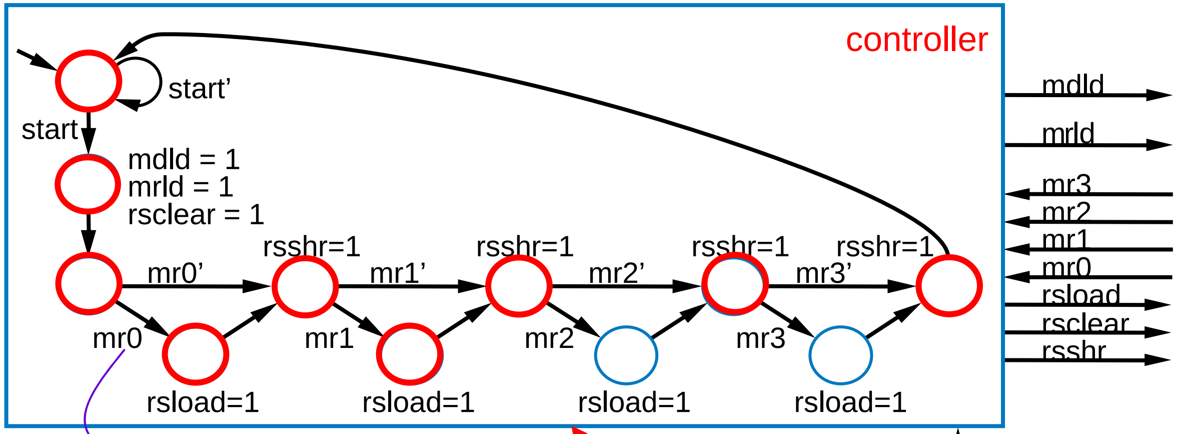
- Design circuit that computes one partial product at a time, adds to running sum
  - Note that shifting running sum right (relative to partial product) after each step ensures partial product added to correct running sum bits



```
Step 1            Step 2            Step 3              Step 4
  0110              0110              0110                 0110
+ 0011            + 0011            + 0011               + 0011
  0000             00110            010010              0010010 (running sum)
+ 0110            +0110             + 0000               + 0000   (partial product)
 00110            010010           0010010             00010010 (new running sum)
```

a

Smaller Multiplier -- Sequential Style: Controller



Vs. array-style:

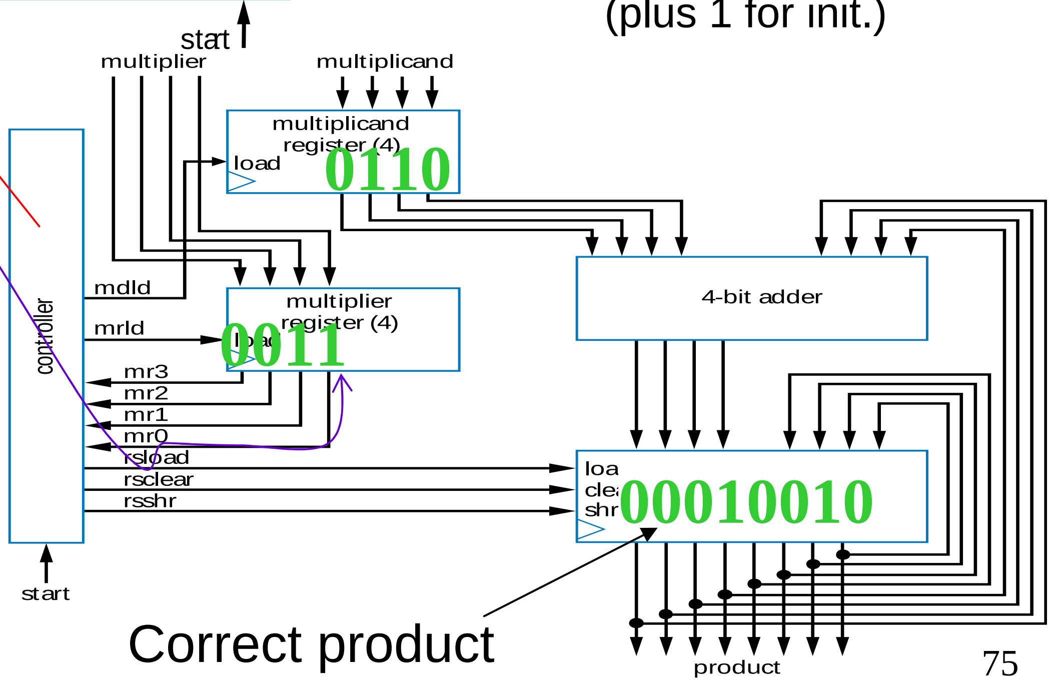
Pro: small

- Just three registers, adder, and controller

Con: slow

- 2 cycles per multiplier bit
- 32-bit: 32*2=64 cycles (plus 1 for init.)

- Wait for *start*=1
- Looks at multiplier one bit at a time
  - Adds partial product (multiplicand) to running sum if present multiplier bit is 1
  - Then shifts running sum right one position



Correct product

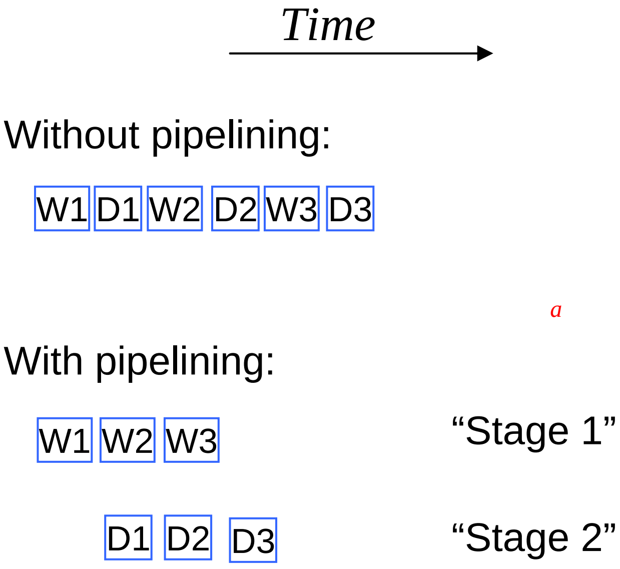
6.5

RTL Design Optimizations and Tradeoffs

- While creating datapath during RTL design, there are several optimizations and tradeoffs, involving
  - Pipelining
  - Concurrency
  - Component allocation
  - Operator binding
  - Operator scheduling
  - Moore vs. Mealy high-level state machines

Pipelining

- Intuitive example: Washing dishes with a friend, you wash, friend dries
  - You wash plate 1
  - Then friend dries plate 1, *while you wash plate 2*
  - Then friend dries plate 2, while you wash plate 3; and so on
  - You don't sit and watch friend dry; you start on the next plate
- ***Pipelining:*** Break task into stages, each stage outputs data for next stage, all stages operate concurrently (if they have data)



Time

Without pipelining:

W1 D1 W2 D2 W3 D3

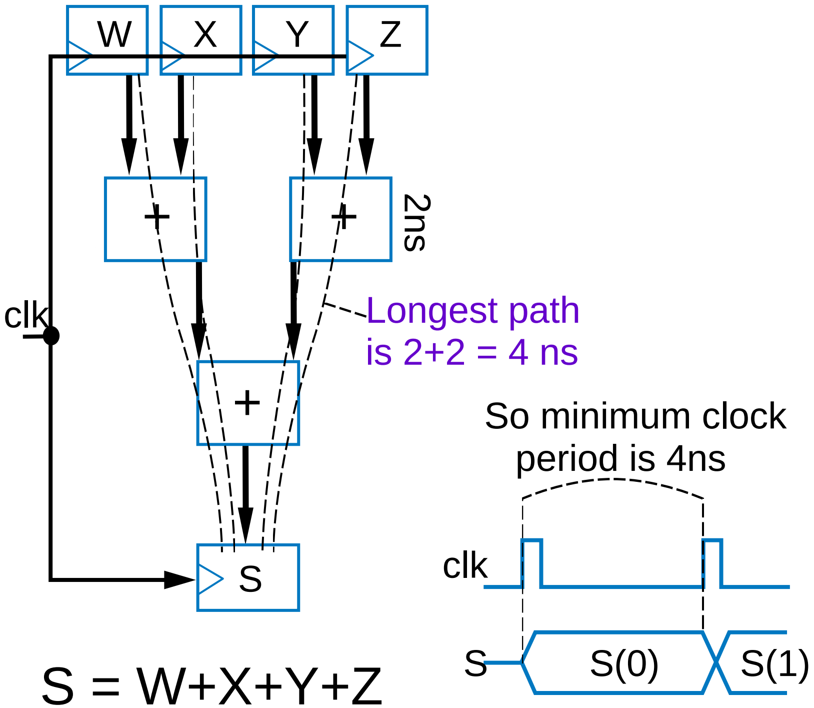
a

With pipelining:

W1 W2 W3 "Stage 1"

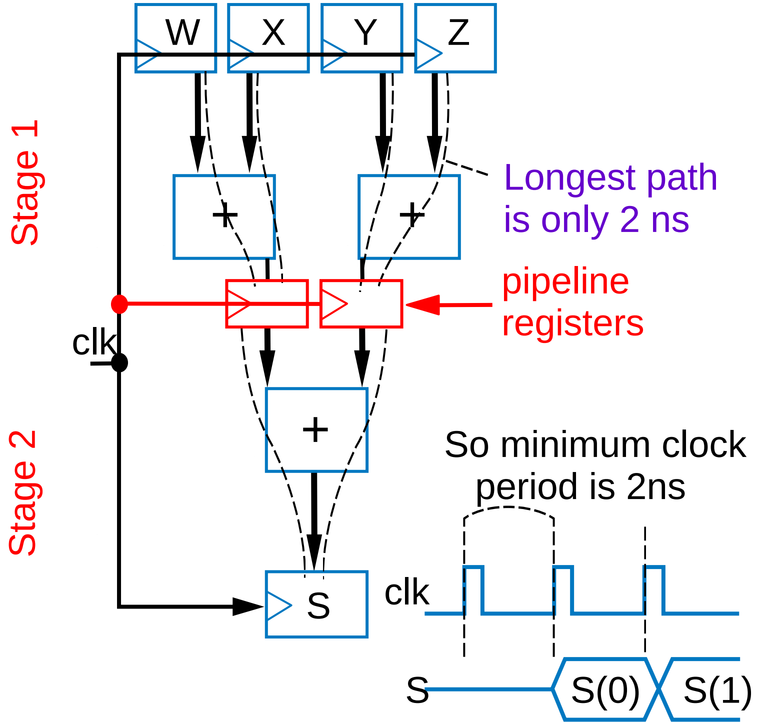
D1 D2 D3 "Stage 2"

Pipelining Example



Longest path is 2+2 = 4 ns

So minimum clock period is 4ns



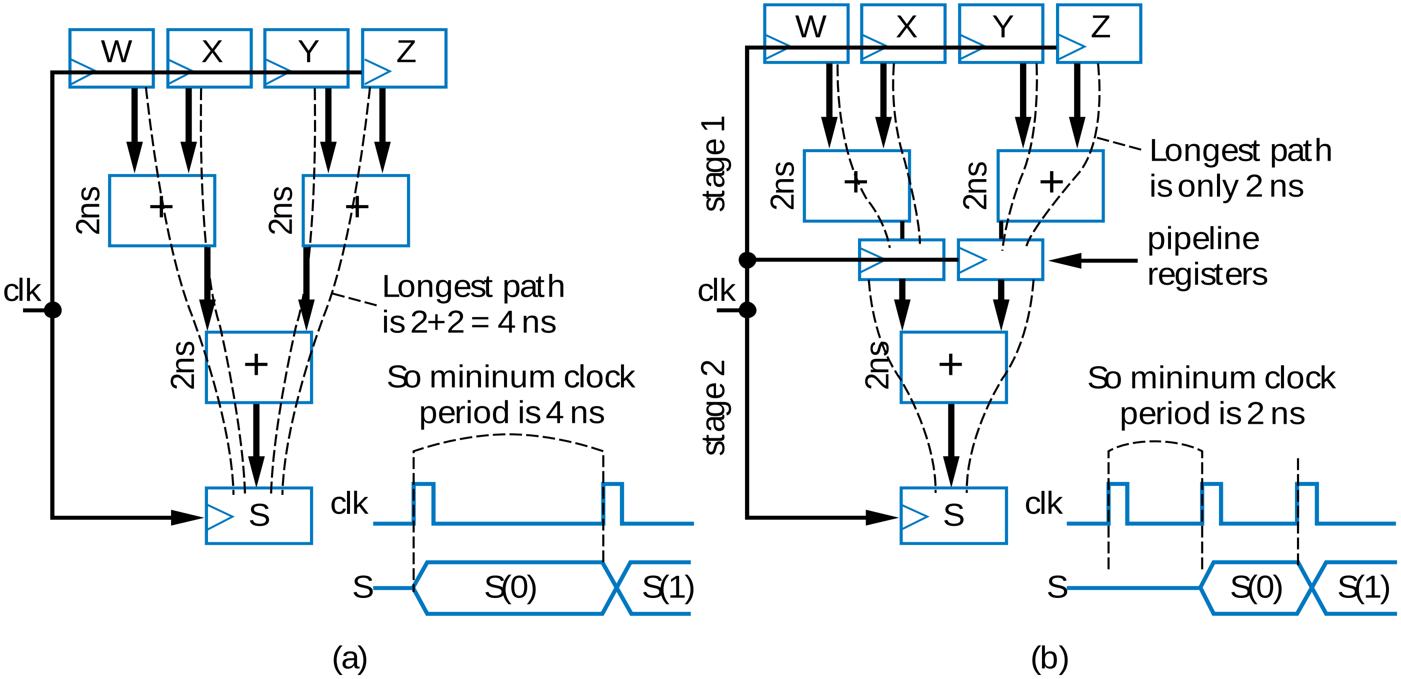
Longest path is only 2 ns

pipeline registers

So minimum clock period is 2ns

- S = W+X+Y+Z
- Datapath on left has critical path of 4 ns, so fastest clock period is 4 ns
  - Can read new data, add, and write result to *S*, every 4 ns
- Datapath on right has critical path of only 2 ns
  - So can read new data every 2 ns – *doubled performance* (sort of...)

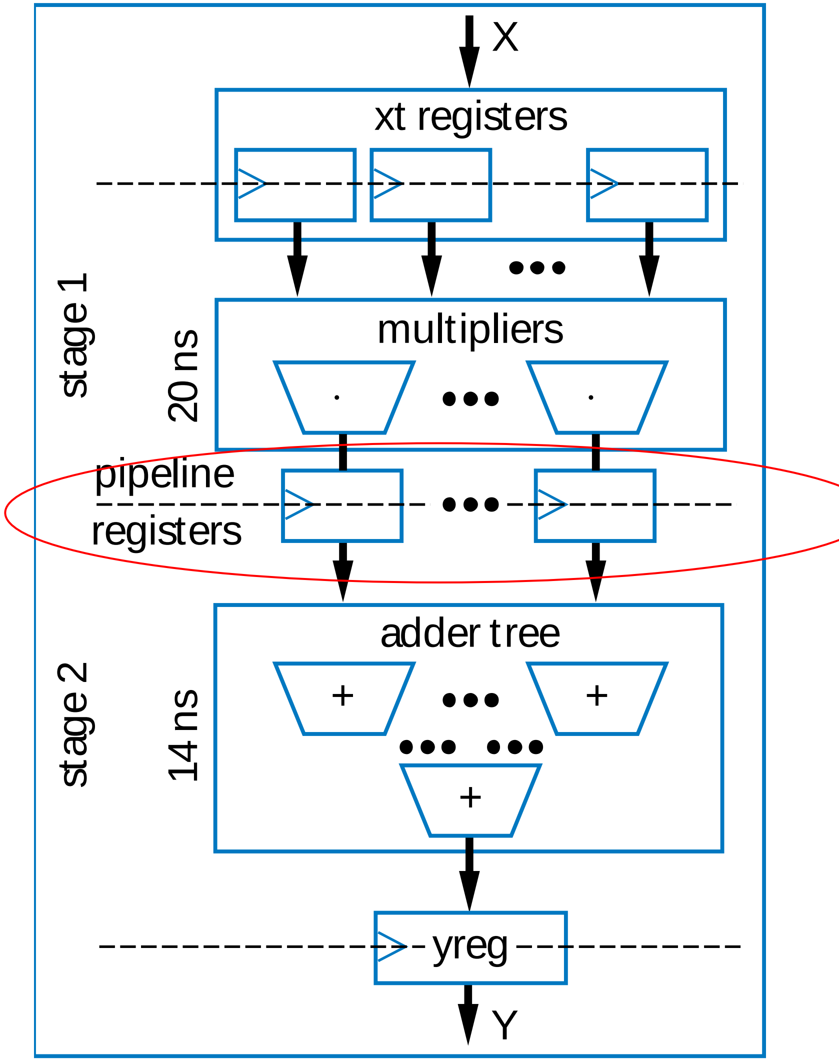
Pipelining Example


W
X
Y
Z
2ns
+
2ns
+
clk
2ns
+
S
Longest path is 2+2 = 4 ns
So mininum clock period is 4 ns
clk
S
S(0)
S(1)
(a)
W
X
Y
Z
stage 1
2ns
+
2ns
+
Longest path is only 2 ns
pipeline registers
clk
stage 2
2ns
+
S
So mininum clock period is 2 ns
clk
S
S(0)
S(1)
(b)

- Pipelining requires refined definition of performance
  - ***Latency:*** Time for new data to result in new output data (seconds)
  - ***Throughput:*** Rate at which new data can be input (items / second)
  - So pipelining above system
    - Doubled the throughput, from 1 item / 4 ns, to 1 item / 2 ns
    - Latency stayed the same: 4 ns

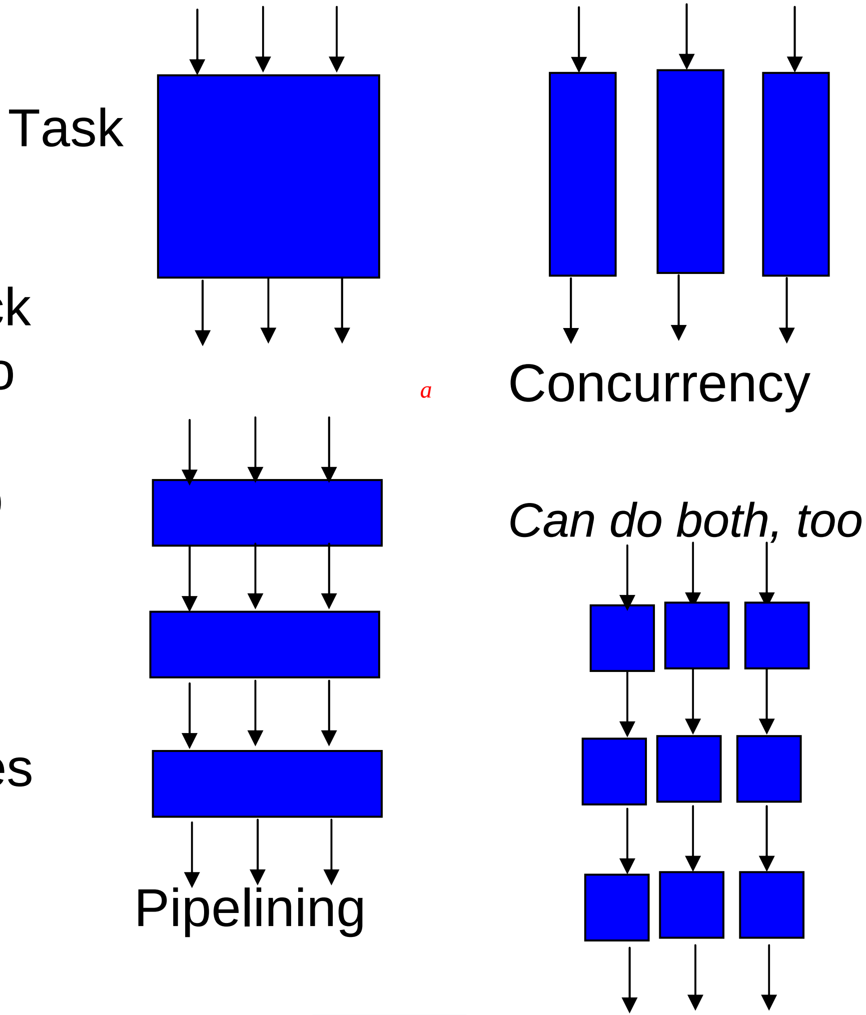
Pipeline Example: FIR Datapath

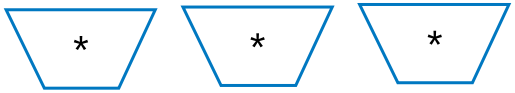
- 100-tap FIR filter: Row of 100 concurrent multipliers, followed by tree of adders
  - Assume 20 ns per multiplier
  - 14 ns for entire adder tree
  - Critical path of 20+14 = 34 ns
- Add pipeline registers
  - Longest path now only 20 ns
  - Clock frequency can be nearly doubled
    - Great speedup with minimal extra hardware



Concurrency

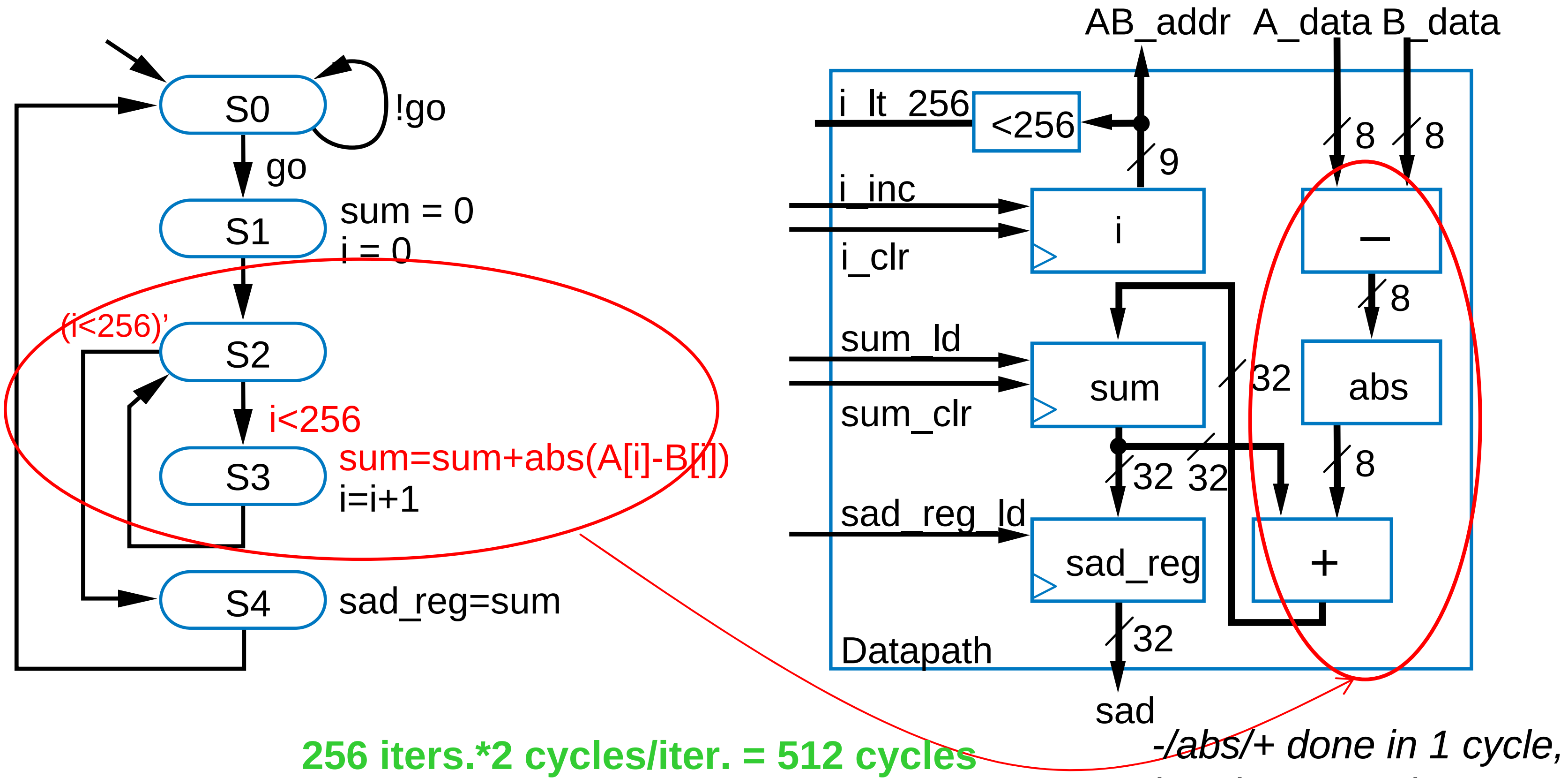
- ***Concurrency***: Divide task into subparts, execute subparts simultaneously
  - Dishwashing example: Divide stack into 3 substacks, give substacks to 3 neighbors, who work simultaneously -- 3 times speedup (ignoring time to move dishes to neighbors' homes)
  - Concurrency does things side-by-side; pipelining instead uses stages (like a factory line)
  - Already used concurrency in FIR filter -- concurrent multiplications





Concurrency Example: SAD Design Revisited

- Sum-of-absolute differences video compression example (Ch 5)
  - Compute sum of absolute differences (SAD) of *256 pairs* of pixels
  - Original : Main loop did 1 sum per iteration, 256 iterations, 2 cycles per iter.

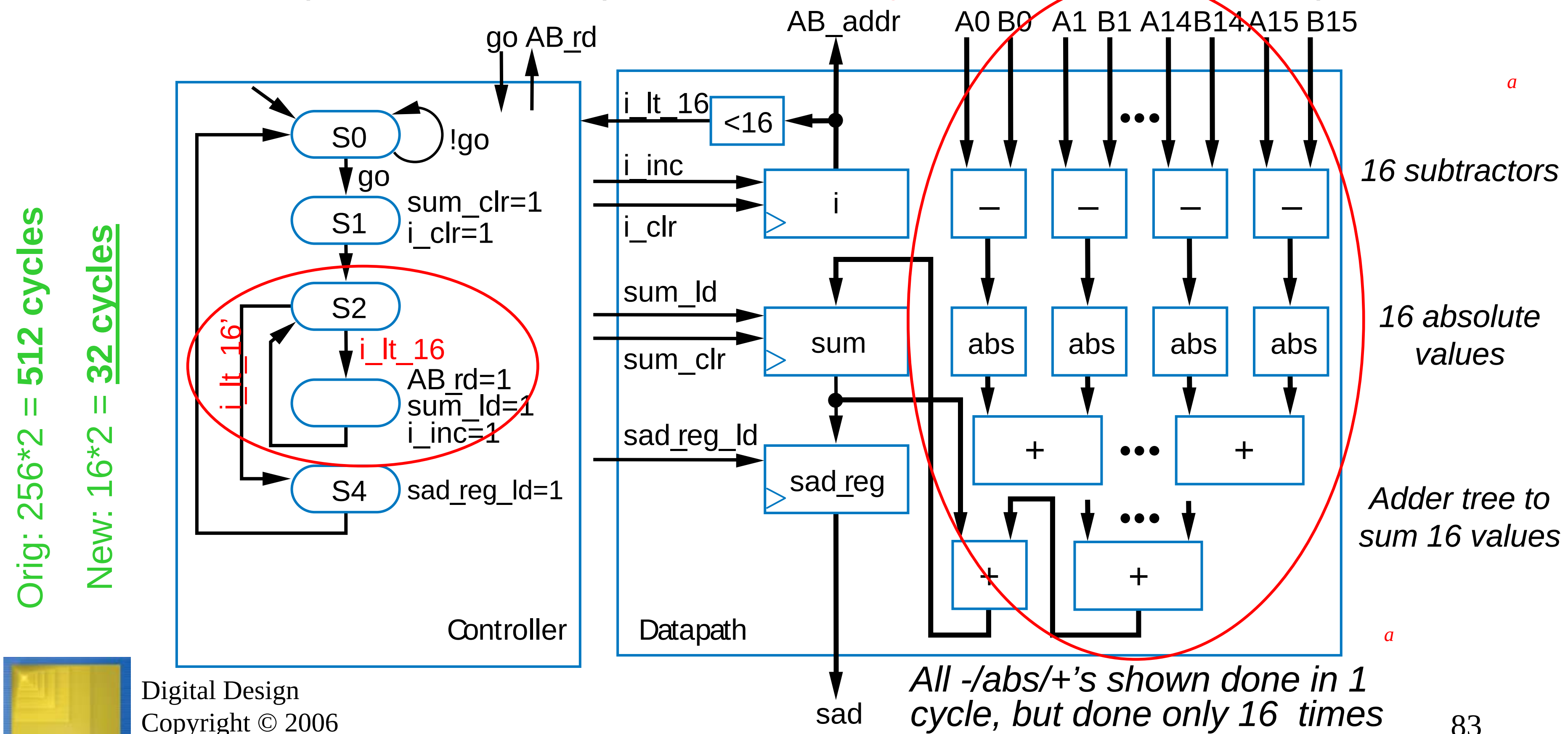


256 iters.*2 cycles/iter. = 512 cycles

-/abs/+ done in 1 cycle, but done 256 times

Concurrency Example: SAD Design Revisited

- More concurrent design
  - Compute SAD for *16 pairs concurrently*, do 16 times to compute all 16*16=256 SADs.
  - Main loop does 16 sums per iteration, only 16 iters., still 2 cycles per iter.



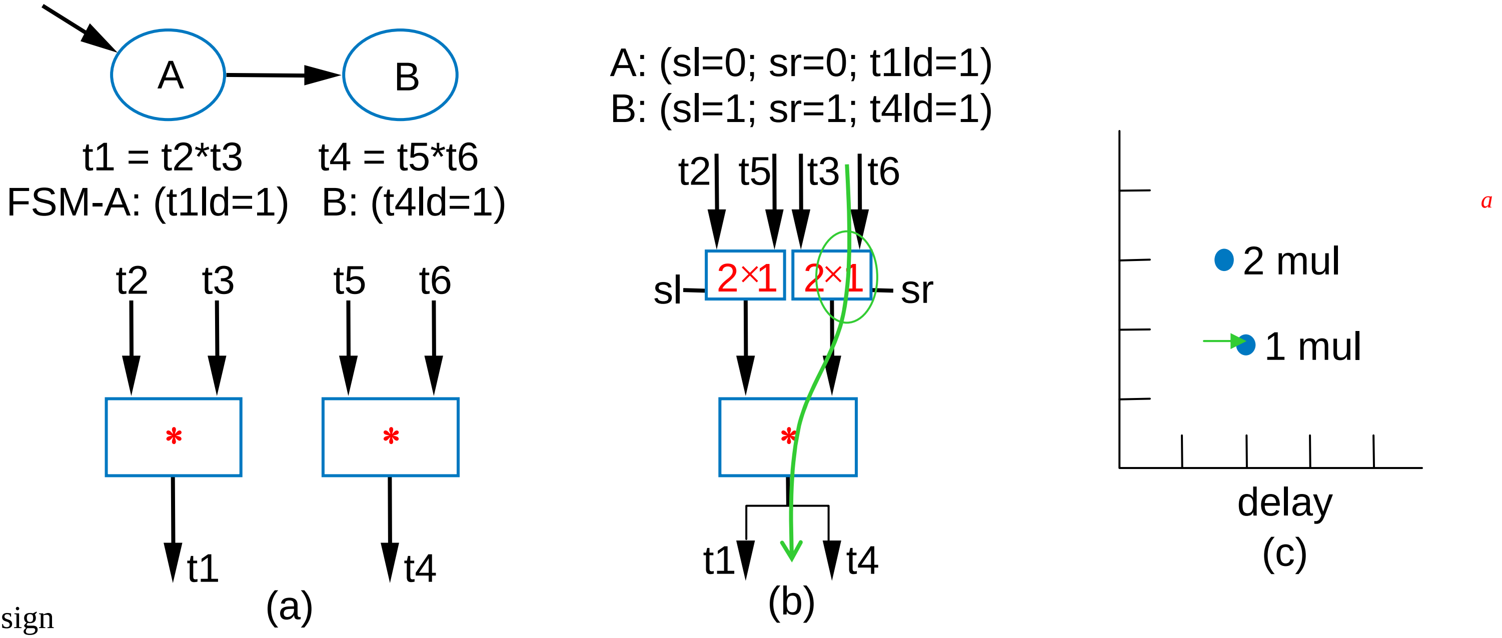
All -/abs/+'s shown done in 1 cycle, but done only 16 times

Concurrency Example: SAD Design Revisited

- Comparing the two designs
  - Original: 256 iterations * 2 cycles/iter = 512 cycles
  - More concurrent: 16 iterations * 2 cycles/iter = 32 cycles
  - Speedup: 512/32 = 16x speedup
- Versus software
  - Recall: Estimated about 6 microprocessor cycles per iteration
    - 256 iterations * 6 cycles per iteration = 1536 cycles
    - Original design speedup vs. software: 1536 / 512 = 3x
      - (assuming cycle lengths are equal)
    - Concurrent design’s speedup vs. software: 1536 / 32 = 48x
  - 48x is very significant – quality of video may be much better

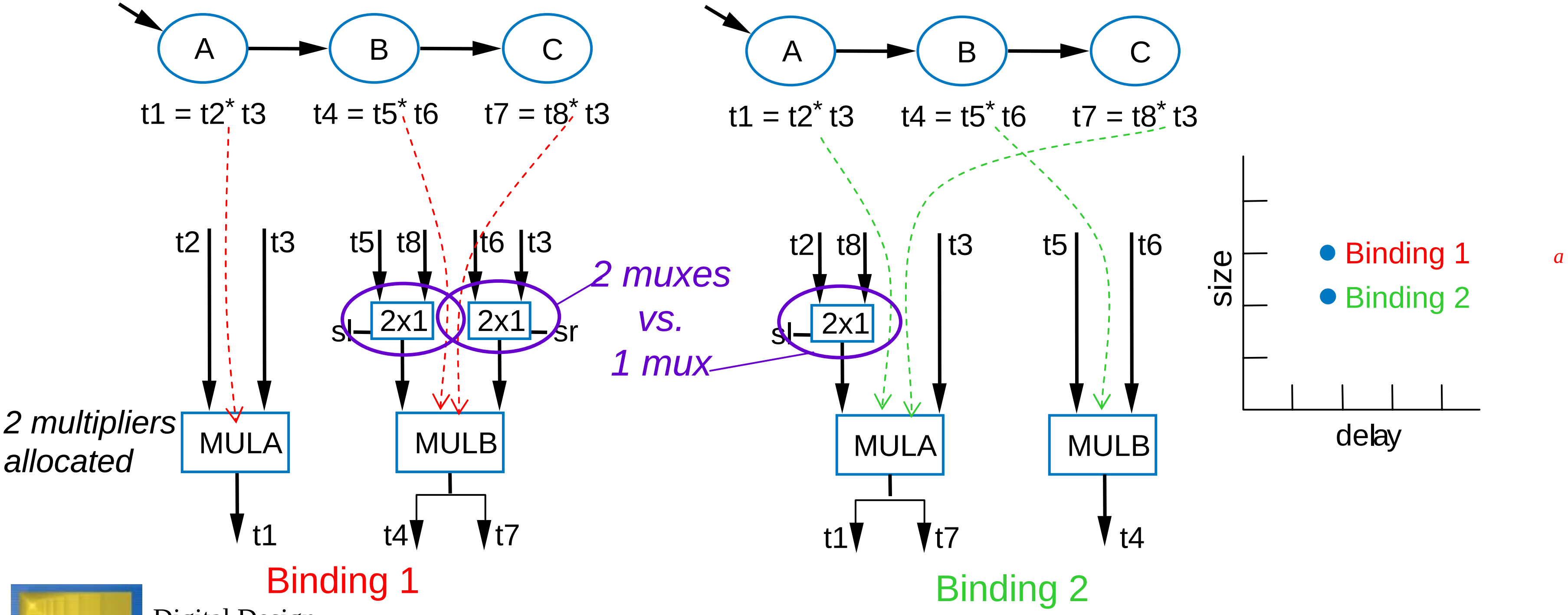
Component Allocation

- Another RTL tradeoff: ***Component allocation*** – Choosing a particular set of functional units to implement a set of operations
  - e.g., given two states, each with multiplication
    - Can use 2 multipliers (*)
    - OR, can instead use 1 multiplier, and 2 muxes
    - Smaller size, but slightly longer delay due to the mux delay



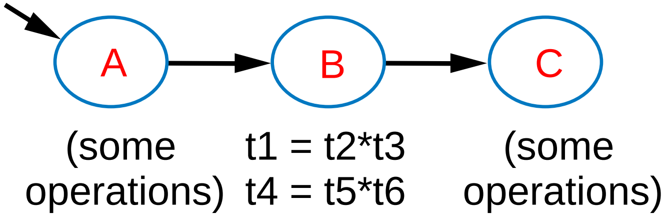
Operator Binding

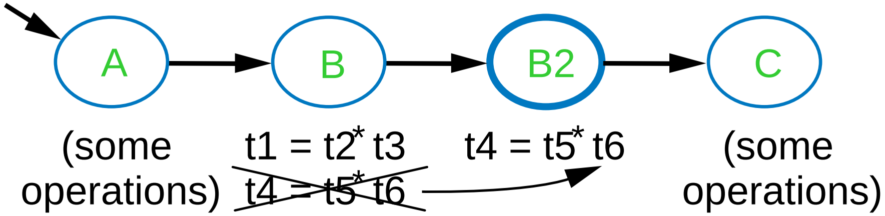
- Another RTL tradeoff: ***Operator binding*** – Mapping a set of operations to a particular component allocation
  - Note: operator/operation mean behavior (multiplication, addition), while component (aka functional unit) means hardware (multiplier, adder)
  - Different bindings may yield different size or delay

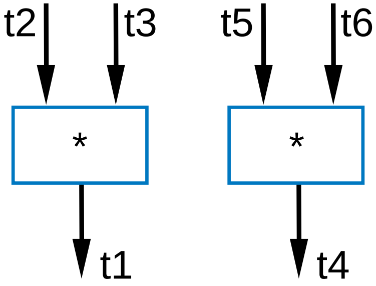


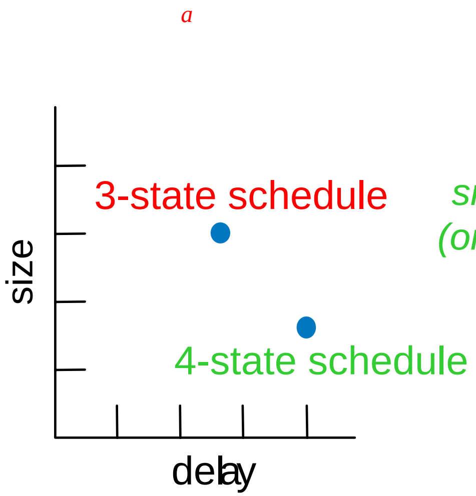
Operator Scheduling

- Yet another RTL tradeoff: ***Operator scheduling*** – Introducing or merging states, and assigning operations to those states.

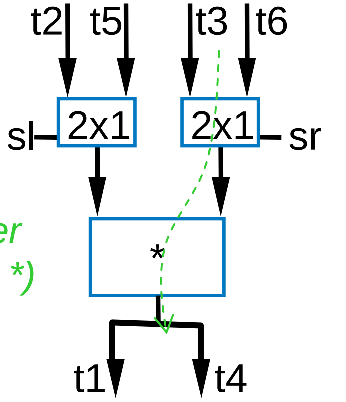








*smaller (only 1 *)*



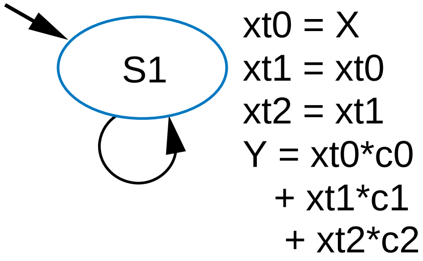
but more delay due to muxes

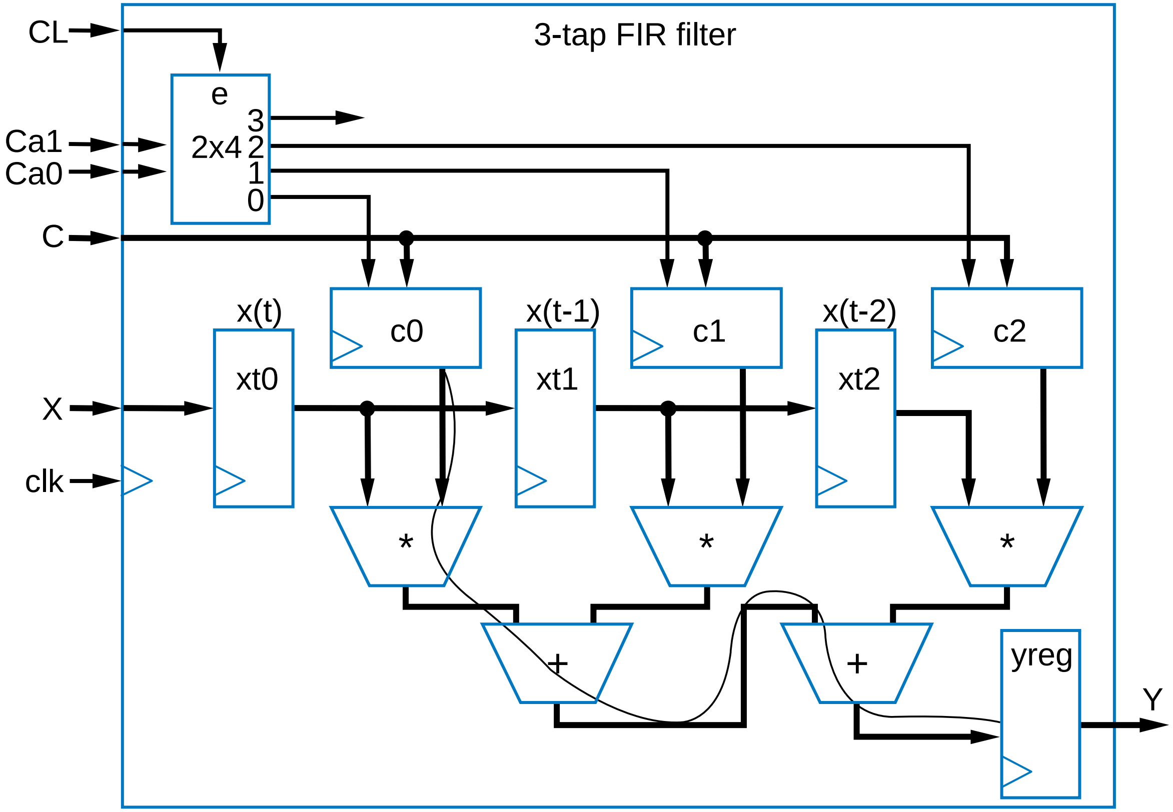
Operator Scheduling Example: Smaller FIR Filter

- 3-tap FIR filter design in Ch 5: Only one state – datapath computes new Y every cycle
  - Used 3 multipliers and 2 adders; can we reduce the design's size?

$$y(t) = c0*x(t) + c1*x(t\text{-}1) + c2*x(t\text{-}2)$$

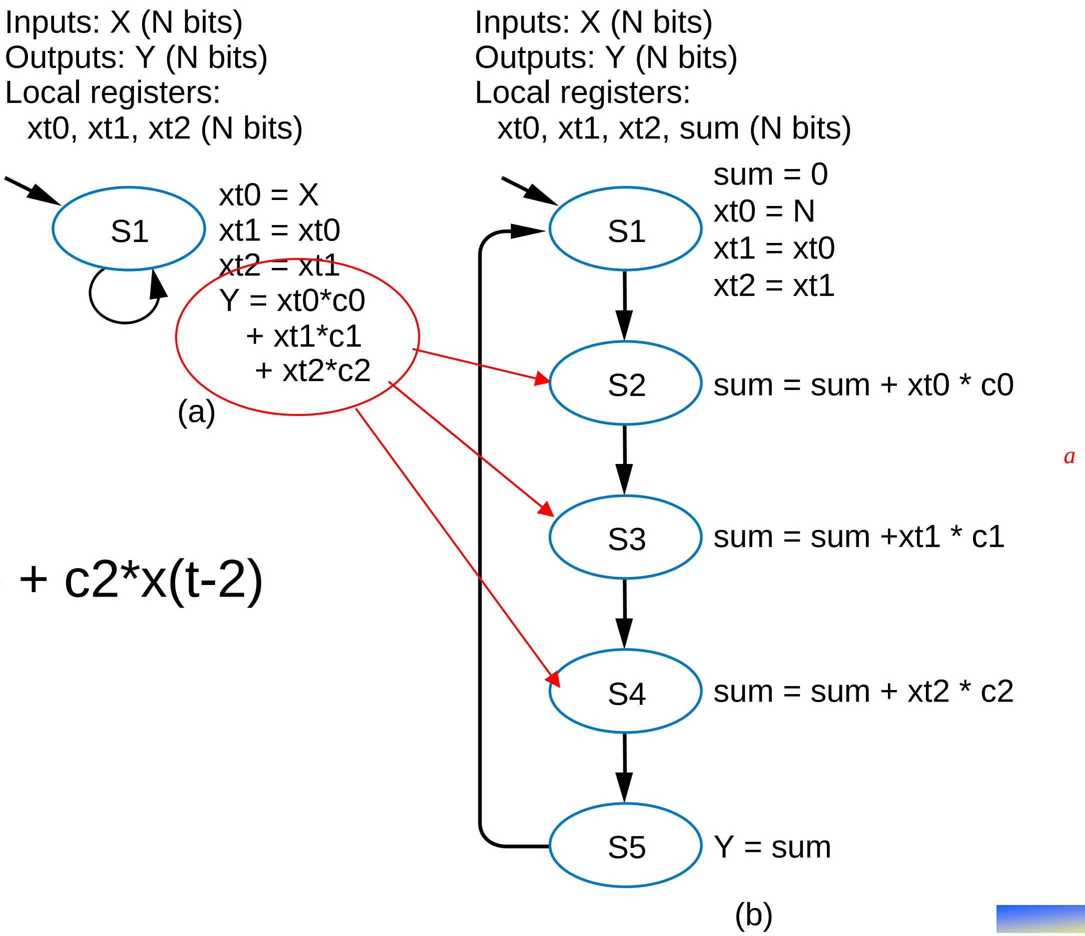
Inputs: X (N bits)
Outputs: Y (N bits)
Local registers:
  xt0, xt1, xt2 (N bits)





Operator Scheduling Example: Smaller FIR Filter

- Reduce the design’s size by re-scheduling the operations
  - Do only one multiplication operation per state

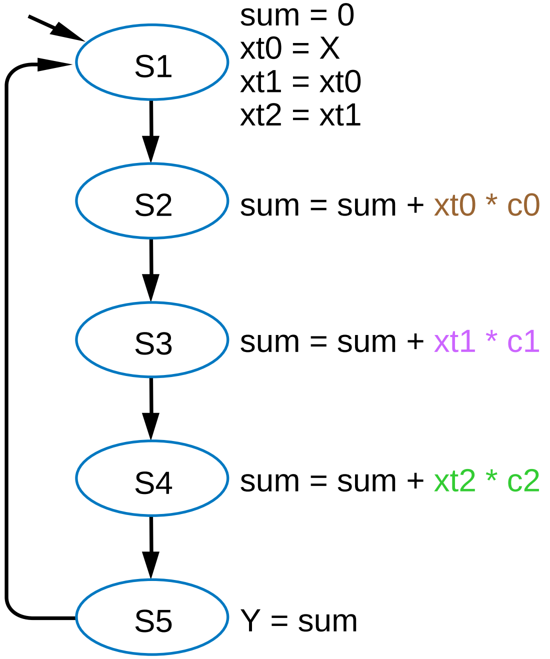


y(t) = c0*x(t) + c1*x(t-1) + c2*x(t-2)

Operator Scheduling Example: Smaller FIR Filter

- Reduce the design’s size by re-scheduling the operations
  - Do only one multiplication (*) operation per state, along with sum (+)

Inputs: X (N bits)
Outputs: Y (N bits)
Local registers:
  xt0, xt1, xt2, sum (N bits)



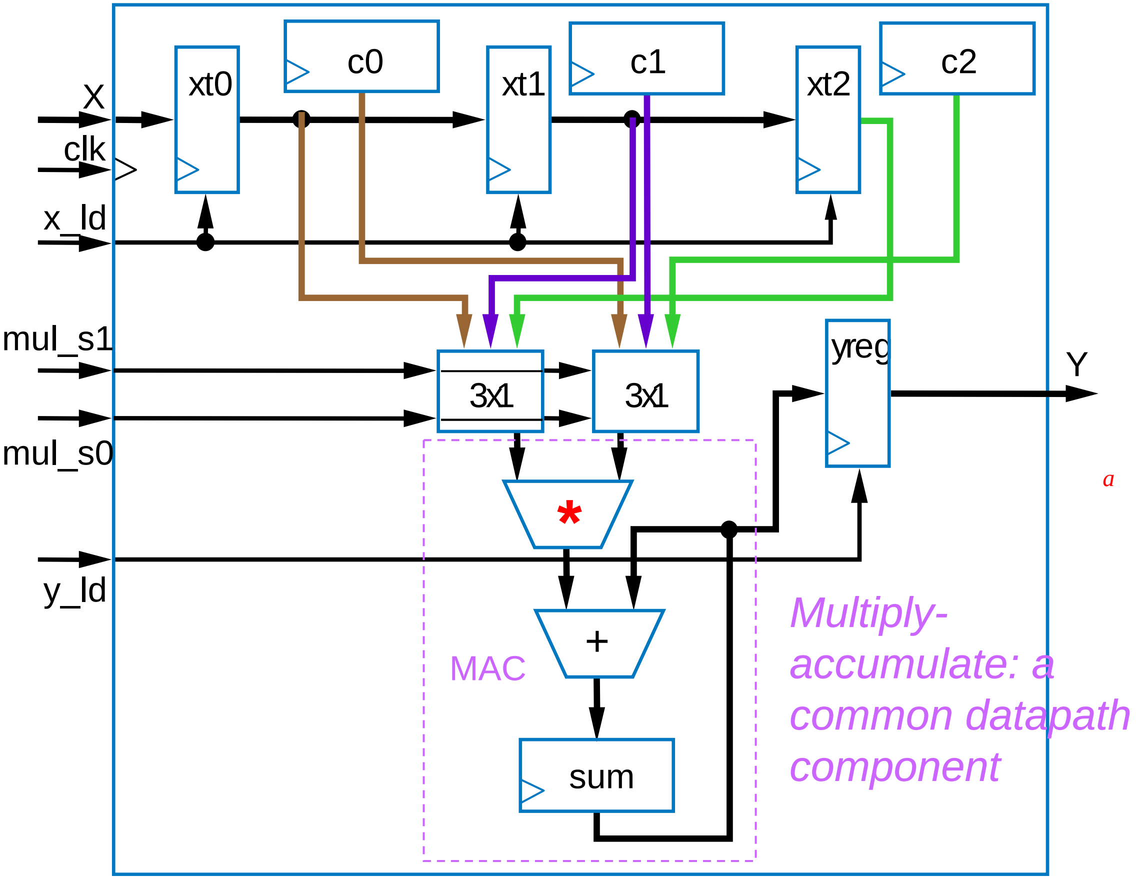
S1: sum = 0
xt0 = X
xt1 = xt0
xt2 = xt1

S2: sum = sum + xt0 * c0

S3: sum = sum + xt1 * c1

S4: sum = sum + xt2 * c2

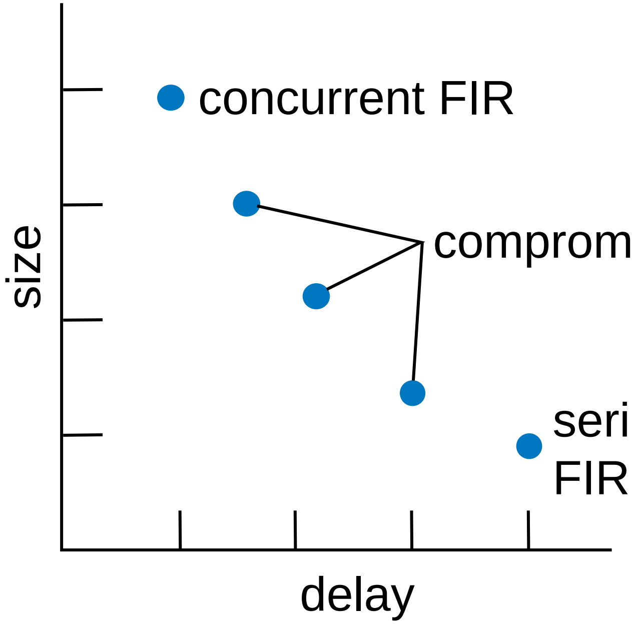
S5: Y = sum



Multiply-accumulate: a common datapath component

Operator Scheduling Example: Smaller FIR Filter

- Many other options exist between fully-concurrent and fully-serialized
  - e.g., for 3-tap FIR, can use 1, 2, or 3 multipliers
  - Can also choose fast array-style multipliers (which are concurrent internally) or slower shift-and-add multipliers (which are serialized internally)
  - Each options represents compromises



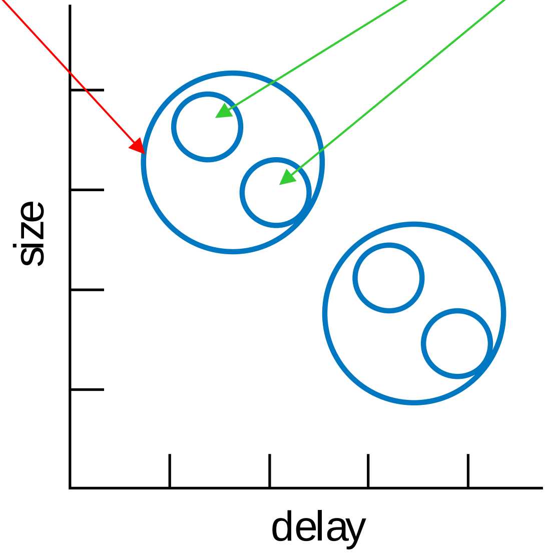
More on Optimizations and Tradeoffs

- **Serial vs. concurrent computation** has been a common tradeoff theme at all levels of design
  - Serial: Perform tasks one at a time
  - Concurrent: Perform multiple tasks simultaneously
- Combinational logic tradeoffs
  - Concurrent: Two-level logic (fast but big)
  - Serial: Multi-level logic (smaller but slower)
    - abc + abd + ef → (ab)(c+d) + ef – essentially computes ab first (serialized)
- Datapath component tradeoffs
  - Serial: Carry-ripple adder (small but slow)
  - Concurrent: Carry-lookahead adder (faster but bigger)
    - Computes the carry-in bits concurrently
  - Also multiplier: concurrent (array-style) vs. serial (shift-and-add)
- RTL design tradeoffs
  - Concurrent: Schedule multiple operations in one state
  - Serial: Schedule one operation per state

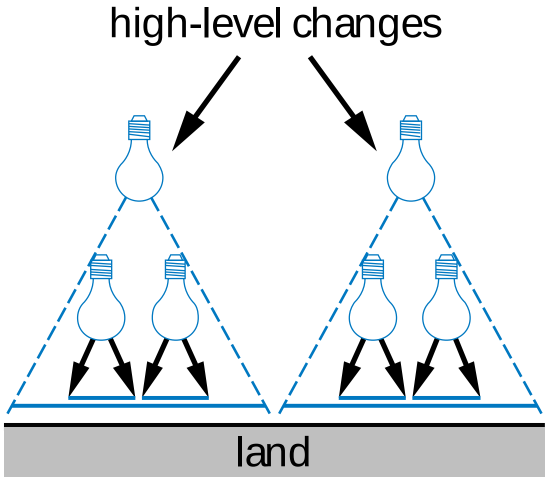
Higher vs. Lower Levels of Design

- Optimizations and tradeoffs at higher levels typically have greater impact than those at lower levels
  - RTL decisions impact size/delay more than gate-level decisions

Spotlight analogy: The lower you are, the less solution landscape is illuminated (meaning possible)



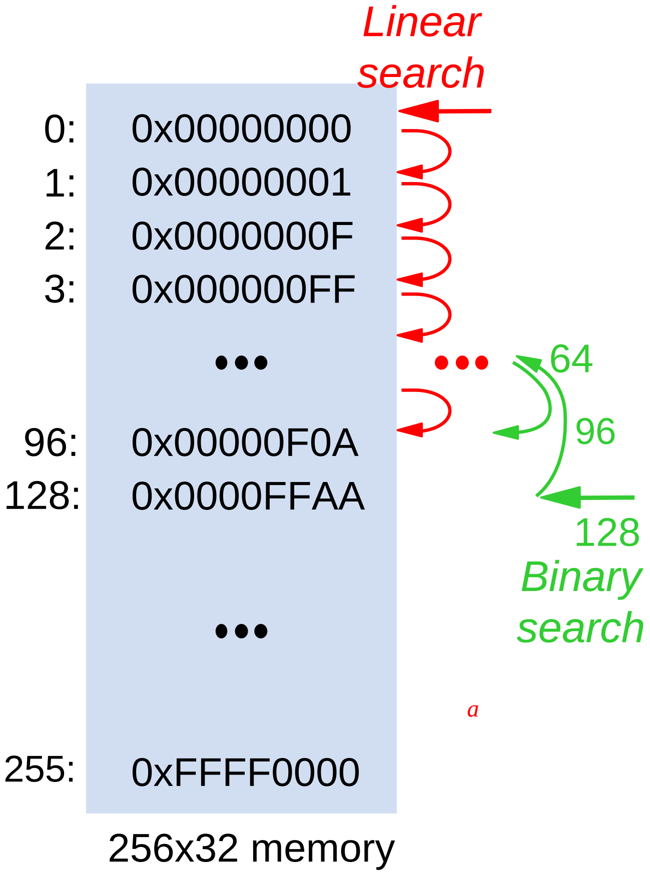
(a)



(b)

Algorithm Selection

- Chosen algorithm can have big impact
  - e.g., which filtering algorithm?
    - FIR is one type, but others require less computation at expense of lower-quality filtering
- Example: Quickly find item's address in 256-word memory
  - One use: data compression. Many others.
  - Algorithm 1: "Linear search"
    - Compare item with M[0], then M[1], M[2], ...
    - 256 comparisons worst case
  - Algorithm 2: "Binary search" (sort memory first)
    - Start considering entire memory range
      - If M[mid]>item, consider lower half of M
      - If M[mid]<item, consider upper half of M
      - Repeat on new smaller range
      - Dividing range by 2 each step; at most 8 such divisions
    - Only 8 comparisons in worst case
- Choice of algorithm has *tremendous* impact
  - Far more impact than say choice of comparator type

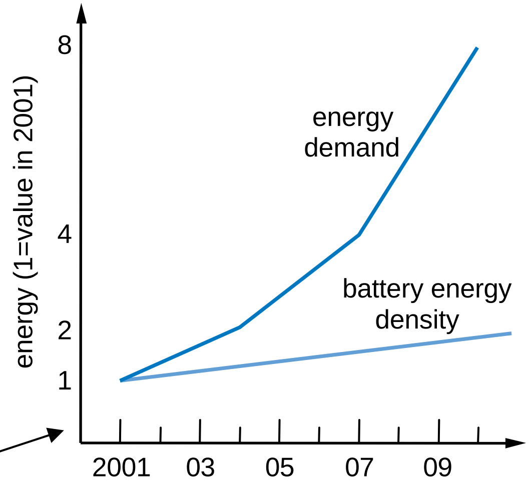


0: 0x00000000
1: 0x00000001
2: 0x0000000F
3: 0x000000FF
•••
96: 0x00000F0A
128: 0x0000FFAA
•••
255: 0xFFFF0000

256x32 memory

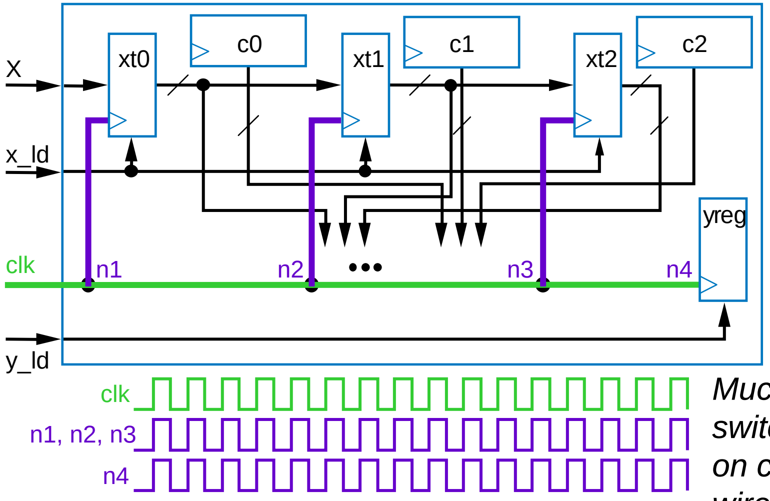
Power Optimization

- Until now, we've focused on size and delay
- **Power** is another important design criteria
  - – Measured in Watts (energy/second)
    - Rate at which energy is consumed
- Increasingly important as more transistors fit on a chip
  - – Power not scaling down at same rate as size
    - Means more heat per unit area – cooling is difficult
    - Coupled with battery's not improving at same rate
      - – Means battery can't supply chip's power for as long
  - – CMOS technology: Switching a wire from 0 to 1 consumes power (known as ***dynamic power***)
    - $P = k * CV^2f$
      - – k: constant; C: capacitance of wires; V: voltage; f: switching frequency
    - Power reduction methods
      - – Reduce voltage: But slower, and there's a limit
      - – What else?



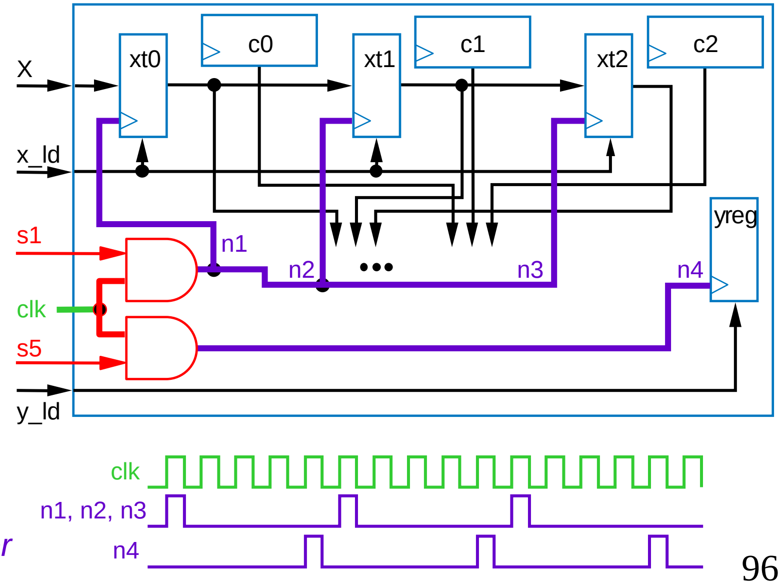
Power Optimization using Clock Gating

- $P = k * CV^2f$
- Much of a chip’s switching *f* (>30%) due to clock signals
  - After all, clock goes to every register
  - Portion of FIR filter shown on right
    - Notice clock signals n1, n2, n3, n4
- Solution: Disable clock switching to registers unused in a particular state
  - Achieve using AND gates
  - FSM only sets 2nd input to AND gate to 1 in those states during which register gets loaded
- Note: Advanced method, usually done by tools, not designers
  - Putting gates on clock wires creates variations in clock signal (***clock skew);*** must be done with great care



Much switching on clock wires

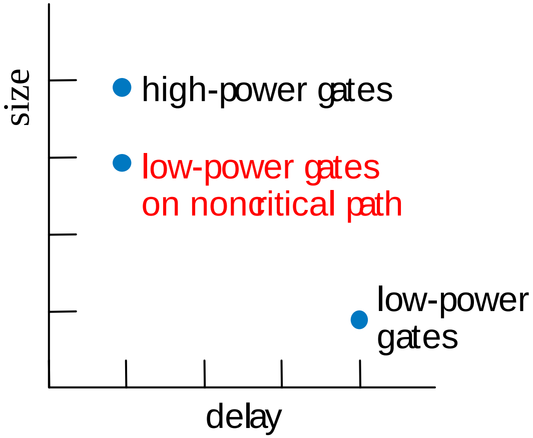
a

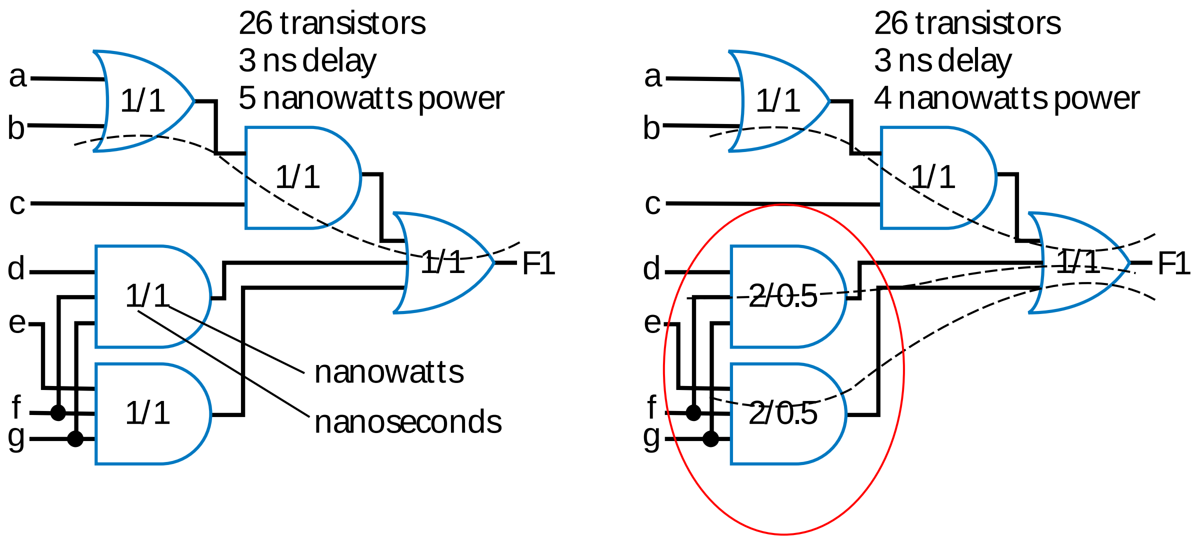


Greatly reduced switching – less power

Power Optimization using Low-Power Gates on Non-Critical Paths

- Another method: Use low-power gates
  - Multiple versions of gates may exist
    - Fast/high-power, and slow/low-power, versions
  - Use slow/low-power gates on non-critical paths
    - Reduces power, without increasing delay


size
high-power gates
low-power gates on noncritical path
low-power gates
delay


26 transistors
3 ns delay
5 nanowatts power
a
b
c
d
e
f
g
1/1
1/1
1/1
1/1
1/1
F1
nanowatts
nanoseconds
26 transistors
3 ns delay
4 nanowatts power
a
b
c
d
e
f
g
1/1
1/1
1/1
2/0.5
2/0.5
F1