



Introduction to Digital Design

Week 13: Optimizations and Tradeoffs

Yao Zheng
University of Hawai'i at Mānoa
Department of Electrical and Computer Engineering

1

Overview

- Optimization (improve criteria without loss) vs. tradeoff (improve criteria at expense of another)
- Combinational logic
 - K-maps and tabular method for two-level logic size optimization
 - Iterative improvement heuristics for two-level optimization and multi-level too
- Sequential logic
 - State minimization, state encoding, Moore vs. Mealy
- Datapath components
 - Faster adder using carry-lookahead
 - Smaller multiplier using sequential multiplication
- RTL
 - Pipelining, concurrency, component allocation, operator binding, operator scheduling
- (Optional) Serial vs. concurrent, efficient algorithms, power reduction methods (clock gating, low-power gates)
- (Optional) Multibillion dollar EDA industry (electronic design automation) creates tools for automated optimization/tradeoffs

2

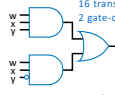
Introduction

6.1

- 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

16 transistors
2 gate-delays



F1 = $wx + wxy'$

(a)

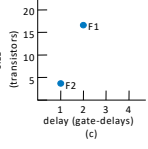
4 transistors
1 gate-delay



F2 = wx

(b)

(a) $\Rightarrow wx(y+y') = wx$ (b)



(c)

Note: Slides with animation are denoted with a small red "a" near the animated items

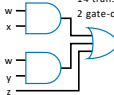
3

Introduction

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

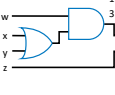
Transforming G1 to G2 represents a **tradeoff**: Some criteria better, others worse.

14 transistors
2 gate-delays

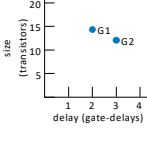


G1 = $wx + wy + z$

12 transistors
3 gate-delays



G2 = $w(x+y) + z$

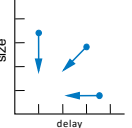


4

Introduction

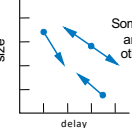
Optimizations

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.

5

Combinational Logic Optimization and Tradeoffs

6.2

- Two-level size optimization using algebraic methods
 - Goal: Two-level circuit (ORed AND gates) with fewest transistors
 - Though transistors getting cheaper (Moore's Law), 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)
 - For simplicity, ignore inverters

Example

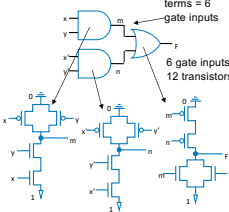
$$F = xyz + xyz' + x'y'z' + x'y'z$$

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

$$F = xy \cdot 1 + x'y' \cdot 1$$

$$F = xy + x'y'$$

4 literals + 2 terms = 6
6 gate inputs = 12 transistors



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

6

Algebraic Two-Level Size Optimization

- Previous example showed common algebraic minimization method
 - (Multiply out to sum-of-products, then...)
- Apply following as much as possible
 - $ab + ab' = a(b + b') = a \cdot 1 = a$
 - "Combining terms to eliminate a variable"
 - (Formally called the "Uniting theorem")
- Duplicating a term sometimes helps
 - Doesn't change function
 - $c + d = c + d + d = c + d + d + d + \dots$
- Sometimes after combining terms, can combine resulting terms
 - $G = xy'z' + xy'z + xyz + xyz'$
 - $G = xy'(z' + z) + xy(z + z')$
 - $G = xy' + xy$ (now do again)
 - $G = x(y' + y)$
 - $G = x$

7

7

Karnaugh Maps for Two-Level Size Optimization

- Easy to miss possible opportunities to combine terms when doing algebraically
- 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: $xy'z' + xy'z = xy'(z' + z) = xy'(1) = xy'$
 - 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 $F = xy' + xy$

Easier than algebraically:

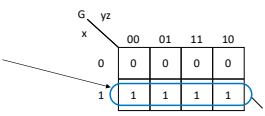
8

8

K-maps

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

$$\begin{aligned} G &= xy'z' + xy'z + xyz + xyz' \\ G &= x(y'z' + y'z + yz + yz') \text{ (must be true)} \\ G &= x(y'(z' + z) + y(z + z')) \\ G &= x(y' + y) \\ G &= x \end{aligned}$$



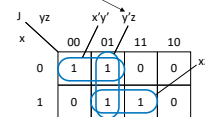
Draw the biggest circle possible, or you'll have more terms than really needed

9

9

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



$$H = xy'z' + xy'z + xy'z + xyz$$

(xy appears in all combinations)

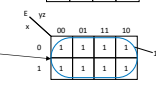
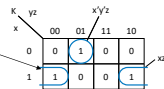
The two circles are shorthand for:
 $l = xy'z' + xy'z + xy'z + xyz + xyz'$
 $l = x(y'z' + y'z + yz + yz') + (xy'z' + xy'z + xy'z + xyz)$
 $l = (x(y' + y) + (xy'z' + xy'z + xy'z + xyz))$
 $l = (y'z' + x)$

10

10

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

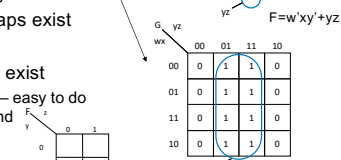
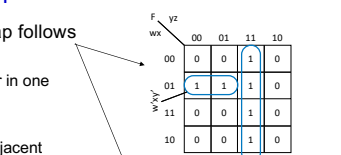


11

11

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



12

12

Two-Level Size Optimization Using K-maps

General K-map method

1. Convert the function's equation into sum-of-minterms form
2. Place 1s in the appropriate K-map cells for each minterm
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.

13

13

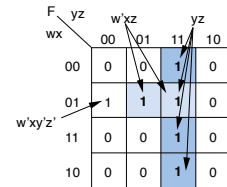
Two-Level Size Optimization Using K-maps

General K-map method

1. Convert the function's equation into sum-of-minterms form
2. Place 1s in the appropriate K-map cells for each minterm

Common to revise (1) and (2):

- Create *sum-of-products*
- Draw 1s for each product

Ex: $F = w'xz + yz + w'xy'z'$ 

14

14

Two-Level Size Optimization Using K-maps

General K-map method

1. Convert the function's equation into sum-of-minterms form
2. Place 1s in the appropriate K-map cells for each minterm
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.

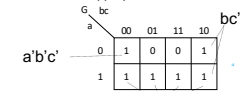
Example: Minimize:

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

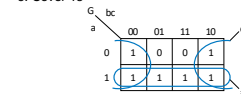
1. Convert to sum-of-products

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

2. Place 1s in appropriate cells



3. Cover 1s

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

15

15

Two-Level Size Optimization Using K-maps

– Four Variable Example

Minimize:

$$H = a'b'(cd' + c'd) + ab'c'd' + ab'cd' + a'bd + a'bcd'$$

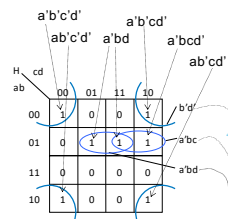
1. Convert to sum-of-products:

$$H = a'b'cd' + a'b'c'd' + ab'c'd' + ab'cd' + a'bd + a'bcd'$$

2. Place 1s in K-map cells

3. Cover 1s

4. OR resulting terms



$$H = b'd' + a'bc + a'bd$$

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

16

16

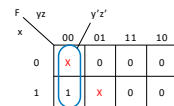
Don't Care Input Combinations

- What if we know that particular input combinations can never occur?

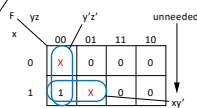
- e.g., Minimize $F = xy'z'$, given that $x'y'z'$ ($xyz=000$) can never be true, and that $xy'z$ ($xyz=101$) can never be true
- So it doesn't matter what F outputs when $x'y'z'$ or $xy'z$ is true, because those cases will never occur
- Thus, make F be 1 or 0 for those cases in a way that best minimizes the equation

On K-map

- Draw Xs for don't care combinations
 - Include X in circle ONLY if minimizes equation
 - Don't include other Xs



Good use of don't cares



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

17

17

Optimization Example using Don't Cares

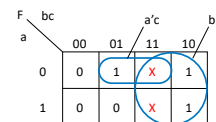
Minimize:

$$F = a'bc' + abc' + a'b'c$$

– Given don't cares: $a'bc, abc$

Note: Introduce 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



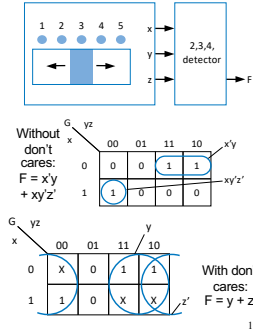
$$F = a'c + b$$

18

18

Optimization with Don't Cares Example: Sliding Switch

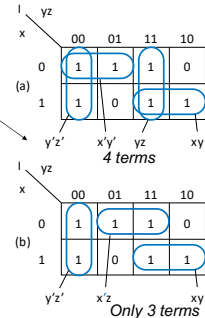
- Switch with 5 positions
 - 3-bit value gives position in binary
- Want circuit that
 - Outputs 1 when switch is in position 2, 3, or 4
 - Outputs 0 when switch is in position 1 or 5
 - Note that the 3-bit input can never output binary 0, 6, or 7
 - Treat as don't care input combinations



19

Automating Two-Level Logic Size Optimization

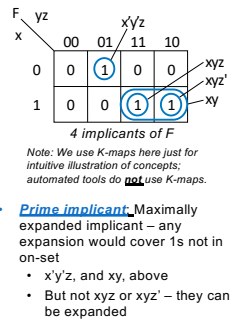
- Minimizing by hand
 - Is hard for functions with 5 or more variables
 - May not yield minimum cover depending on order we choose
 - Is error prone
- Minimization thus typically done by automated tools
 - Exact algorithm:** finds optimal solution
 - Heuristic:** finds good solution, but not necessarily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



20

Basic Concepts Underlying Automated Two-Level Logic Size Optimization

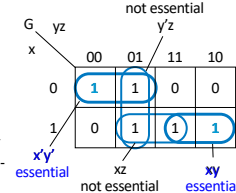
- Definitions
 - On-set:** All minterms that define when $F=1$
 - Off-set:** All minterms that define when $F=0$
 - Implicant:** Any product term (minterm or other) that when 1 causes $F=1$
 - On K-map, any legal (but not necessarily largest) circle
 - Cover: Implicant xy covers minterms xyz and xyz'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



21

Basic Concepts Underlying Automated Two-Level Logic Size Optimization

- 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



22

Automated Two-Level Logic Size Optimization Method

TABLE 6.1 Automatable tabular method for two-level logic size optimization.

Step	Description
1 Determine prime implicants	Starting with minterm implicants, methodically compare all pairs (actually, all pairs whose numbers of uncomplemented literals differ by one) to find opportunities to combine terms to eliminate a variable, yielding new implicants with one less literal. Repeat for new implicants. Stop when no implicants can be combined. All implicants not covered by a new implicant are prime implicants.
2 Add essential prime implicants to the function's cover	Find every minterm covered by only one prime implicant, and denote that prime implicant as essential. Add essential prime implicants to the cover, and mark all 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.

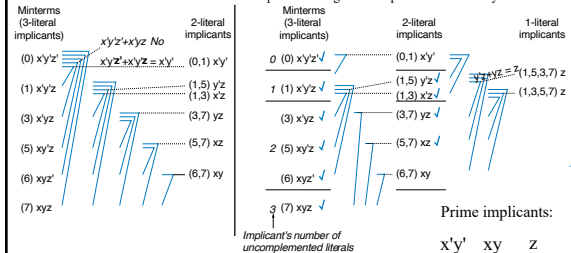
- Steps 1 and 2 are exact
- Step 3: Hard. Checking all possibilities: exact, but computationally expensive. Checking some but not all: heuristic.

23

Tabular Method Step 1: Determine Prime Implicants Methodically Compare All Implicant Pairs, Try to Combine

- Example function: $F = x'y'z' + x'y'z + x'yz + xy'z + xyz' + xyz$

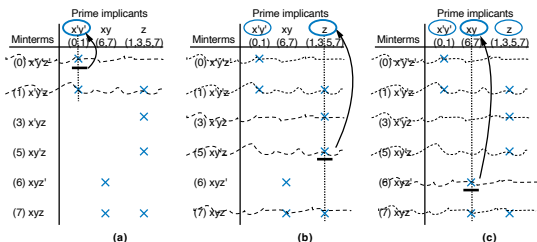
Actually, comparing ALL pairs isn't necessary—just pairs differing in uncomplemented literals by one.



24

Tabular Method Step 2: Add Essential PIs to Cover

- Prime implicants (from Step 1): $x'y'$, xy , z



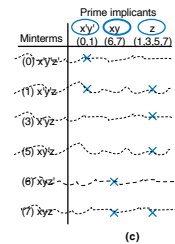
If only one X in row, then that PI is essential—it's the only PI that covers that row's minterm.

25

25

Tabular Method Step 3: Use Fewest Remaining PIs to Cover Remaining Minterms

- Essential PIs (from Step 2): $x'y'$, xy , z
 - Covered all minterms, thus nothing to do in step 3
- Final minimized equation: $F = x'y' + xy + z$



26

26

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 require days, months, years, or more of computation – unreasonable

27

27

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:
 - $F = abcdefg(h+h') + jklmnop = abcdefg + jklmnop$

28

28

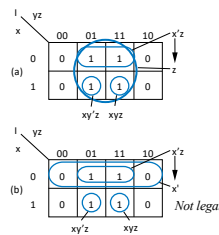
Two-Level Optimization using Iterative Method

- 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 not legal, in shown function
 - After expand, **remove other terms covered** by newly expanded term
 - Keep trying (iterate) until doesn't help

Ex:

$$\begin{aligned} F &= abcdefgh + abcdefgh' + jklmnop \\ F &= abcdefg + abcdefgh' + jklmnop \\ F &= abcdefg + jklmnop \end{aligned}$$

Covered by newly expanded term abcdefg



Illustrated above on K-map, but iterative method is intended for an automated solution (no K-map)

29

29

Ex: Iterative Heuristic for Two-Level Logic Size Optimization

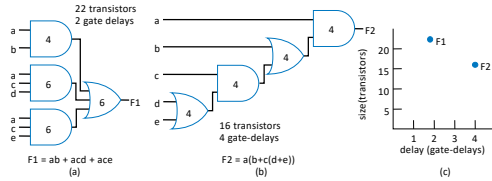
- $F = xyz + xyz' + x'y'z' + x'y'z$ (minterms in on-set)
- Random expand: $F = xy\cancel{x} + xyz' + x'y'z' + x'y'z$
 - Legal: Covers xyz' and xyz , both in on-set
 - Any implicant covered by xy ? Yes, xyz .
- $F = xy + \cancel{xyz'} + x'y'z' + x'y'z$
- Random expand: $F = xy + \cancel{x'y'z'} + x'y'z$
 - Not legal (x covers $xy'z'$, $xy'z$, xyz' , xyz : two not in on-set)
- Random expand: $F = xy + x'y'\cancel{z} + x'y'z$
 - Legal
 - Implicant covered by $x'y'$: $x'y'z$
- $F = xy + x'y'z' + \cancel{x'y'z}$

30

30

Multi-Level Logic Optimization – Performance/Size Tradeoffs

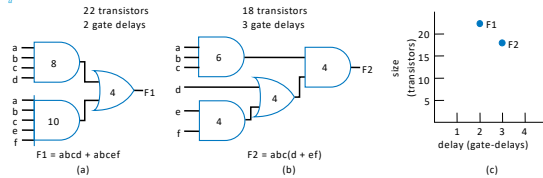
- We don't always need the speed of two-level logic
 - Multiple levels may yield fewer gates
 - Example
 - $F1 = ab + acd + ace \rightarrow F2 = ab + ac(d + e) = a(b + c(d + e))$
 - General technique: Factor out literals – $xy + xz = x(y+z)$



31

Multi-Level Example

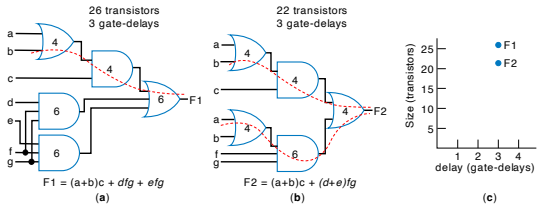
- Q: Use multiple levels to reduce number of transistors for
 - $F1 = abcd + abcef$
- A: $abcd + abcef = abc(d + ef)$
 - Has fewer gate inputs, thus fewer transistors



32

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



33

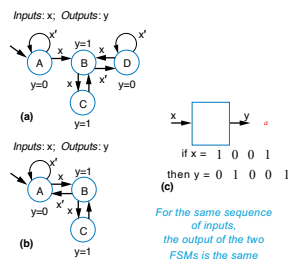
Automated Multi-Level Methods

- Main techniques use heuristic iterative methods
 - Define various operations
 - "Factoring": $abc + abd = ab(c+d)$
 - Plus other transformations similar to two-level iterative improvement

34

Sequential Logic Optimization and Tradeoffs

- State Reduction:** Reduce number of states in FSM *without* changing behavior
 - Fewer states potentially reduces size of state register and combinational logic

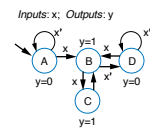


35

State Reduction: Equivalent States

Two states are equivalent if:

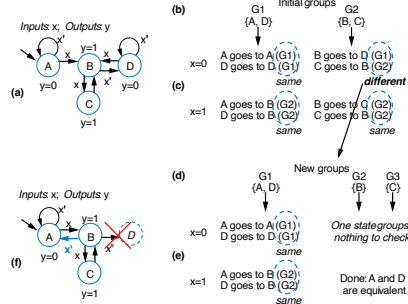
- They assign the same values to outputs
 - e.g. A and D both assign y to 0,
- AND, for *all* possible sequences of inputs, the FSM outputs will be the same starting from either state
 - e.g. say $x=1, 1, 0$
 - starting from A, $y=0, 1, 1, \dots$ (A,B,C,B)
 - starting from D, $y=0, 1, 1, \dots$ (D,B,C,B)



36

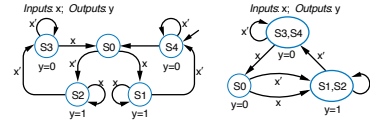
State Reduction via the Partitioning Method

- First partition states into groups based on outputs
- Then partition based on next states



37

Ex: Minimizing States using the Partitioning Method

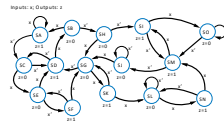


- Initial grouping based on outputs: $G1=\{S3, S0, S4\}$ and $G2=\{S2, S1\}$
- Group G1:
 - for $x=0$, $S3 \rightarrow S3$ (G1), $S0 \rightarrow S2$ (G2), $S4 \rightarrow S0$ (G1)
 - for $x=1$, $S3 \rightarrow S0$ (G1), $S0 \rightarrow S1$ (G2), $S4 \rightarrow S0$ (G1)
 - Divide G1. New groups: $G1=\{S3, S4\}$, $G2=\{S2, S1\}$, and $G3=\{S0\}$
- Repeat for G1, then G2. (G3 only has one state). No further divisions, so done.
 - S3 and S4 are equivalent
 - S2 and S1 are equivalent

38

Need for Automation

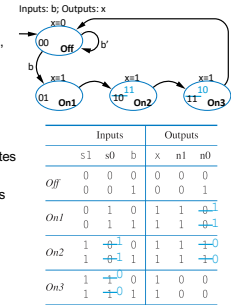
- Partitioning for large FSM too complex for humans
- State reduction typically automated
 - Often using heuristics to reduce compute time



39

State Encoding

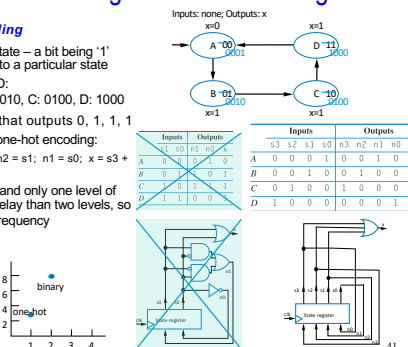
- State encoding:** Assigning unique bit representation to each state
- Different encodings may reduce circuit size, or trade off size and performance
- Minimum-bitwidth binary encoding:** Uses fewest bits
 - Alternatives still exist, such as
 - A:00, B:01, C:10, D:11
 - A:00, B:01, C:11, D:10
 - $4! = 24$ alternatives for 4 states, $N!$ for N states
- Consider Three-Cycle Laser Timer...
 - Example 3.7's encoding led to 15 gate inputs
 - Try **alternative encoding**
 - $x = s1 + s0$
 - $n1 = s0$
 - $n0 = s1'b + s1's0$
 - Only 8 gate inputs
 - Thus fewer transistors



40

State Encoding: One-Hot Encoding

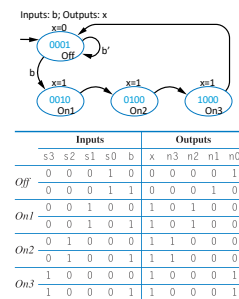
- One-hot encoding**
 - One bit per state – a bit being '1' corresponds to a particular state
 - For A, B, C, D: A: 0001, B: 0010, C: 0100, D: 1000
- Example: FSM that outputs 0, 1, 1, 1
 - Equations if one-hot encoding:
 - $n3 = s2$; $n2 = s1$; $n1 = s0$; $x = s3 + s2 + s1$
 - Fewer gates and only one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



41

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

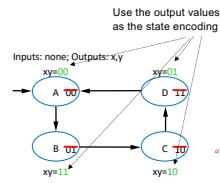
- Four states – Use four-bit one-hot encoding
 - State table leads to equations:
 - $x = s3 + s2 + s1$
 - $n3 = s2$
 - $n2 = s1$
 - $n1 = s0'b$
 - $n0 = s0'b' + s3$
 - Smaller
 - $3+0+0+2+(2+2) = 9$ gate inputs
 - Earlier binary encoding (Ch 3): 15 gate inputs
 - Faster
 - Critical path: $n0 = s0'b'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



42

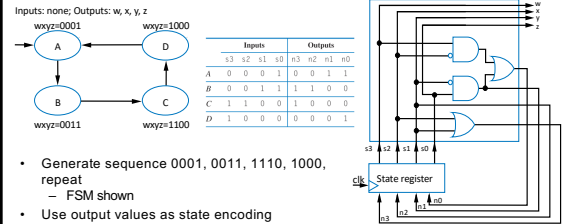
Output Encoding

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



43

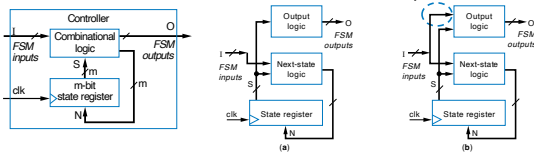
Output Encoding Example: Sequence Generator



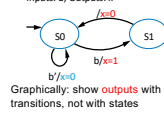
- Generate sequence 0001, 0011, 1110, 1000, repeat
 - FSM shown
- Use output values as state encoding
- Create state table
- Derive equations for next state
 - $n3 = s1 + s2$; $n2 = s1$; $n1 = s1's0$; $n0 = s1's0 + s3s2'$

44

Moore vs. Mealy FSMs

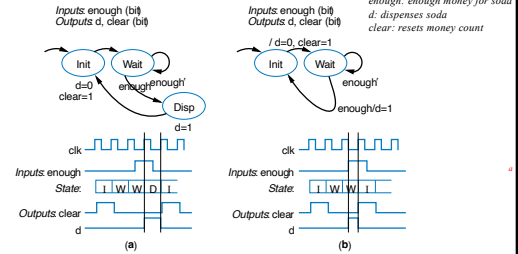


- FSM implementation architecture
 - State register and logic
 - More detailed view
 - Next state logic – function of present state and FSM inputs
 - Output logic
 - If function of present state only – **Moore FSM**
 - If function of present state and FSM inputs – **Mealy FSM**



45

Mealy FSMs May Have Fewer States

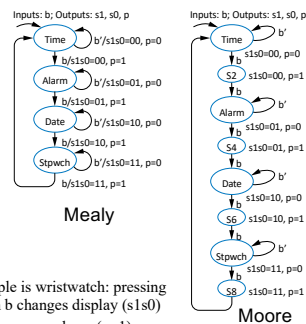


- Soda dispenser example: Initialize, wait until enough, dispense
 - Moore: 3 states; Mealy: 2 states

46

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



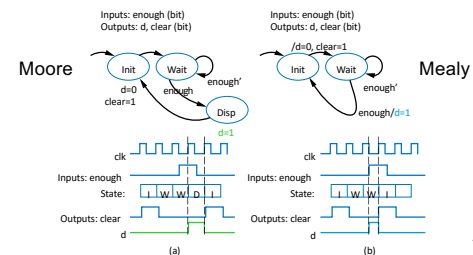
Example is wristwatch: pressing button b changes display (s1s0) and also causes beep (p=1)

Assumes button press is synchronized to occur for one cycle only

47

Mealy vs. Moore Tradeoff

- Mealy may have fewer states, but drawback is that its 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



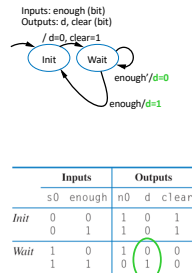
48

47

48

Implementing a Mealy FSM

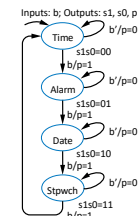
- Straightforward
 - Convert to state table
 - Derive equations for each output
 - Key difference from Moore: External outputs (d , $clear$) may have **different value** in same state, depending on input values



49

49

Mealy and Moore can be Combined



Combined Moore/Mealy FSM for beeping wristwatch example

Easier to comprehend due to clearly associating s1s0 assignments to each state, and not duplicating s1s0 assignments on outgoing transitions

50

50

Datapath Component Tradeoffs

- Can make some components faster (but bigger), or smaller (but slower), than the straightforward components in Ch 4
- This chapter builds:
 - 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

6.4

51

51

Building a Faster Adder

- Built carry-ripple adder in Ch 4

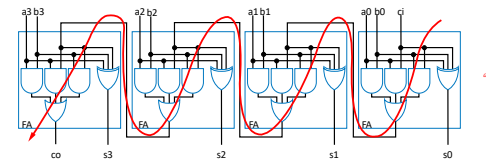
- Similar to adding by hand, column by column

- Con: Slow

- Output is not correct until the carries have rippled to the left – **critical path**
- 4-bit carry-ripple adder has $4 \times 2 = 8$ gate delays

- Pro: Small

- 4-bit carry-ripple adder has just $4 \times 5 = 20$ gates



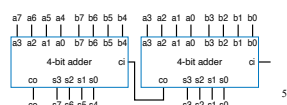
52

52

Building a Faster Adder

- Carry-ripple is slow but small
 - 8-bit: $8 \times 2 = 16$ gate delays, $8 \times 5 = 40$ gates
 - 16-bit: $16 \times 2 = 32$ gate delays, $16 \times 5 = 80$ gates
 - 32-bit: $32 \times 2 = 64$ gate delays, $32 \times 5 = 160$ gates
- Two-level logic adder (2 gate delays)
 - OK for 4-bit adder: About 100 gates
 - 8-bit: 8,000 transistors / 16-bit: 2 M / 32-bit: 100 billion
 - N-bit two-level adder uses *absurd* number of gates for N much beyond 4
- Compromise
 - Build 4-bit adder using two-level logic, compose to build N-bit adder
 - 8-bit adder: $2 \times (2 \text{ gate delays}) = 4$ gate delays, $2 \times (100 \text{ gates}) = 200$ gates
 - 32-bit adder: $8 \times (2 \text{ gate delays}) = 32$ gate delays, $8 \times (100 \text{ gates}) = 800$ gates

Can we do better?



53

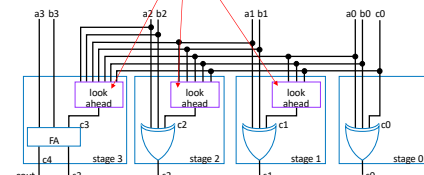
53

Faster Adder – (Naïve Inefficient) 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



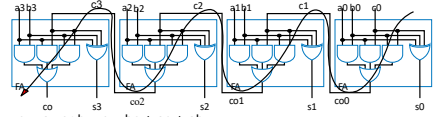
Notice – no rippling of carry

54

54

Faster Adder – (Naïve Inefficient) Attempt at “Lookahead”

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



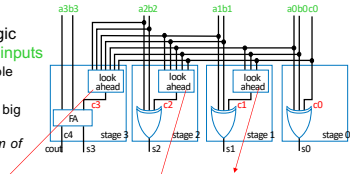
- Full-adder: $s = a \text{ xor } b$ $c = bc + ac + ab$
 $c1 = c0 = b0c0 + a0c0 + a0b0$
 $c2 = c0 = b1c1 + a1c1 + a1b1$
 $c2 = b1(b0c0 + a0c0 + a0b0) + a1(b0c0 + a0c0 + a0b0) + a1b1$
 $c2 = b1b0c0 + b1a0c0 + b1a0b0 + a1b0c0 + a1a0c0 + a1a0b0 + a1b1$
 $c3 = c0 = b2c2 + a2c2 + a2b2$
 (continue plugging in...)

55

55

Faster Adder – (Naïve Inefficient) 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



$$c1 = b0c0 + a0c0 + a0b0$$

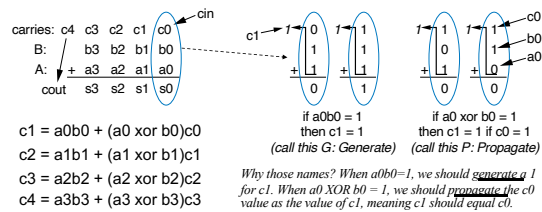
$$c2 = b1b0c0 + b1a0c0 + b1a0b0 + a1b0c0 + a1a0c0 + a1a0b0 + a1b1$$

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

56

56

Efficient Lookahead



$$c1 = a0b0 + (a0 \text{ xor } b0)c0$$

$$c2 = a1b1 + (a1 \text{ xor } b1)c1$$

$$c3 = a2b2 + (a2 \text{ xor } b2)c2$$

$$c4 = a3b3 + (a3 \text{ xor } b3)c3$$

$$c1 = G0 + P0c0$$

$$c2 = G1 + P1c1$$

$$c3 = G2 + P2c2$$

$$c4 = G3 + P3c3$$

Why those names? When $a0b0=1$, we should generate a 1 for $c1$. When $a0 \text{ xor } b0 = 1$, we should propagate the $c0$ value as the value of $c1$, meaning $c1$ should equal $c0$.

$$Gi = ai bi \text{ (generate)}$$

$$Pi = ai \text{ xor } bi \text{ (propagate)}$$

57

57

Efficient Lookahead

- Substituting as in the naïve scheme:

$$c1 = G0 + P0c0$$

$$c2 = G1 + P1c1$$

$$c2 = G1 + P1(G0 + P0c0)$$

$$c2 = G1 + P1G0 + P1P0c0$$

$$c3 = G2 + P2c2$$

$$c3 = G2 + P2(G1 + P1G0 + P1P0c0)$$

$$c3 = G2 + P2G1 + P2P1G0 + P2P1P0c0$$

$$c4 = G3 + P3G2 + P3P2G1 + P3P2P1G0 + P3P2P1P0c0$$

$$Gi = ai bi \text{ (generate)}$$

$$Pi = ai \text{ xor } bi \text{ (propagate)}$$

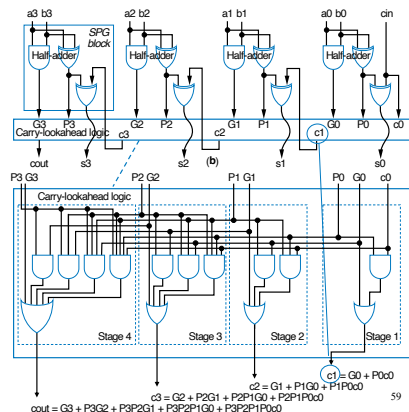
Gi/Pi function of inputs only

58

58

CLA

- Each stage:
 - HA for G and P
 - Another XOR for s
 - Call SPG block
- Create carry-lookahead logic from equations
- More efficient than naïve scheme, at expense of one extra gate delay



$$c1 = G0 + P0c0$$

$$c2 = G1 + P1G0 + P1P0c0$$

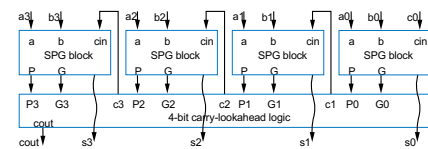
$$c3 = G2 + P2G1 + P2P1G0 + P2P1P0c0$$

$$c4 = G3 + P3G2 + P3P2G1 + P3P2P1G0 + P3P2P1P0c0$$

59

59

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

60

60

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 carry-ripple manner
 - Ex: 16-bit adder: $4 + 4 + 4 + 4 = 16$ gate delays. Can we do better?

61

61

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

62

62

Hierarchical 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

63

63

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

64

64

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

65

65

Smaller Multiplier

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

66

66

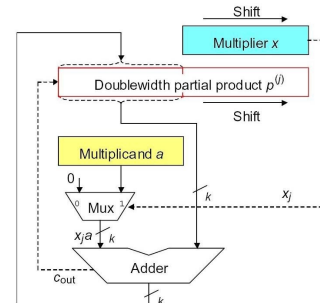
Smaller Multiplier -- Sequential (Add-and-Shift) Style

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

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

67

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



68

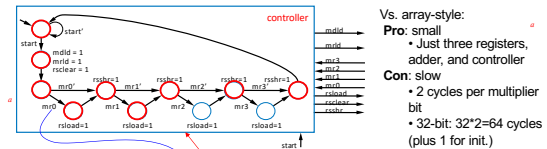
Smaller Multiplier -- Sequential (Add-and-Shift) Style

- 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



69

Smaller Multiplier -- Sequential Style: Controller



- 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

70

RTL Design Optimizations and Tradeoffs

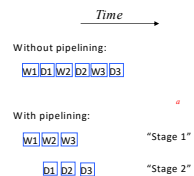
- While creating datapath during RTL design, there are several optimizations and tradeoffs, involving
 - Pipelining
 - Concurrency
 - Component allocation
 - Operator binding
 - Operator scheduling
 - Moore vs. Mealy high-level state machines

6.5

71

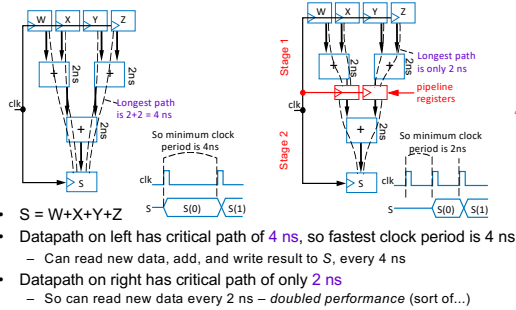
Pipelining

- 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)



72

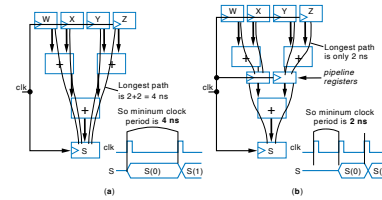
Pipelining Example



73

73

Pipelining Example

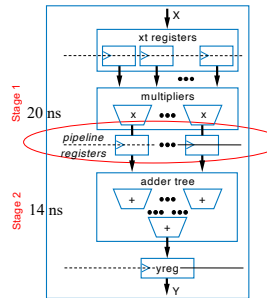


74

74

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

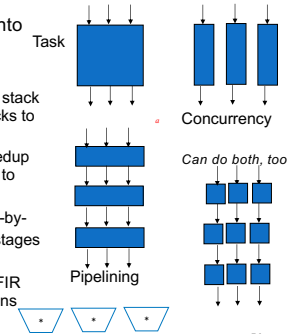


75

75

Concurrency

- Concurrency:** Divide task into subtasks, execute subtasks 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

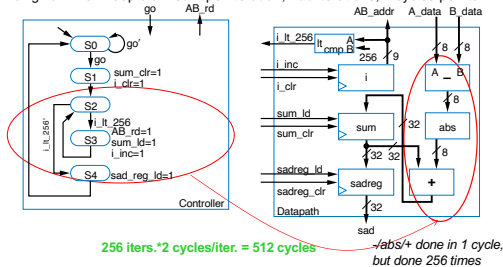


76

76

Concurrency Example: SAD Design Revisited

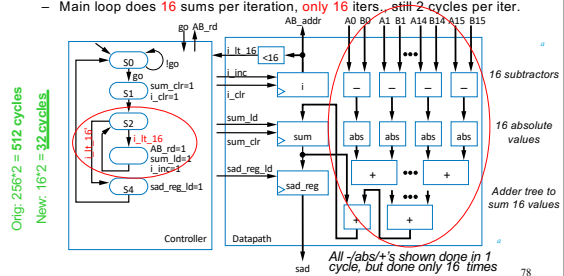
- Sum-of-absolute differences video compression example (Ch 5)
 - Compute sum of absolute differences (SAD) of 256 pairs of pixels
 - Original: Main loop did 1 sum per iteration, 256 iterations, 2 cycles per iter.



77

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.



78

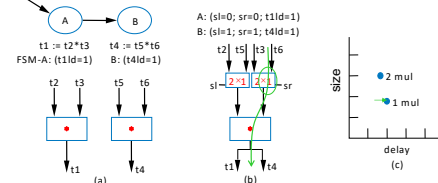
Concurrency Example: SAD Design Revisited

- 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

79

Component Allocation

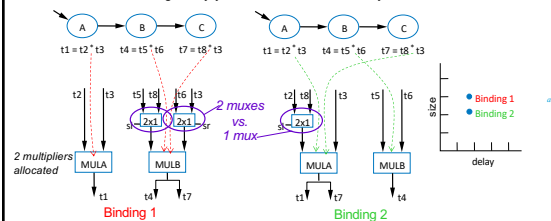
- Another RTL tradeoff: **Component allocation** – Choosing a particular set of functional units to implement a set of operations
 - e.g., given two states, each with multiplication
 - Can use **2 multipliers (*)**
 - OR, can instead use **1 multiplier, and 2 muxes**
 - Smaller size, but slightly longer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



80

Operator Binding

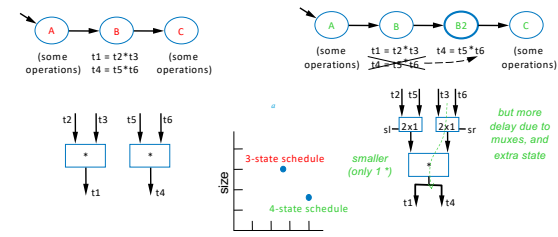
- Another RTL tradeoff: **Operator binding** – Mapping a set of operations to a particular component allocation
 - Note: operator/operation mean behavior (multiplication, addition), while component (aka functional unit) means hardware (multiplier, adder)
 - Different bindings may yield different size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



81

Operator Scheduling

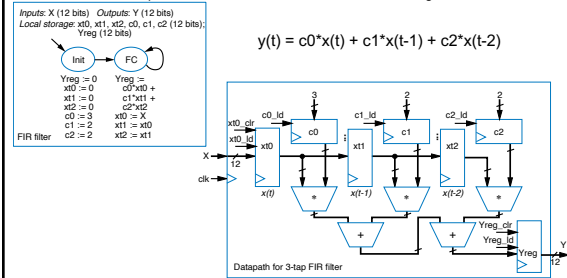
- Yet another RTL tradeoff: **Operator scheduling** – Introducing or merging states, and assigning operations to those states.



82

Operator Scheduling Example: Smaller FIR Filter

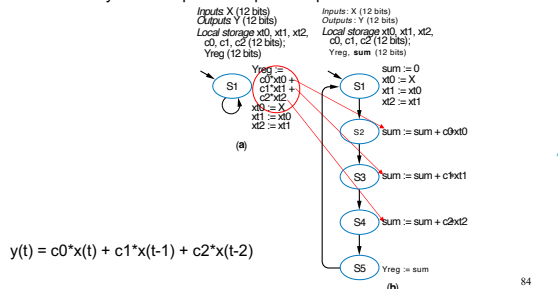
- 3-tap FIR filter of Ch 5: Two states – DP computes new Y every cycle
 - Used 3 multipliers and 2 adders; can we reduce the design's size?



83

Operator Scheduling Example: Smaller FIR Filter

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



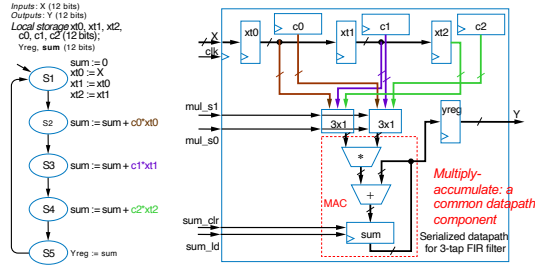
84

83

84

Operator Scheduling Example: Smaller FIR Filter

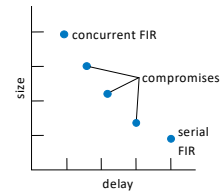
- Reduce the design's size by re-scheduling the operations
 - Do only one multiplication (*) operation per state, along with sum (+)



85

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



86

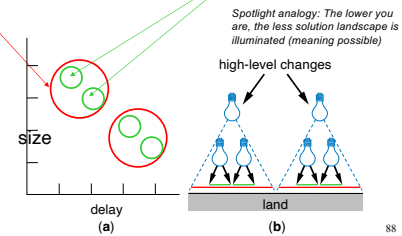
More on Optimizations and Tradeoffs

- 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)
 - $abc + abd + ef \rightarrow (ab)(c+d) + ef$ – 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**

87

Higher vs. Lower Levels of Design

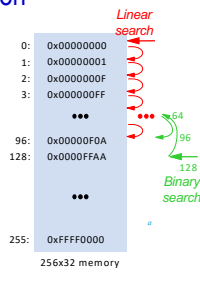
- Optimizations and tradeoffs at higher levels typically have greater impact than those at lower levels
 - Ex: **RTL** decisions impact size/delay more than **gate-level** decisions



88

Algorithm Selection

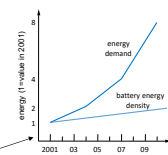
- Chosen algorithm can have big impact
 - e.g., which filtering algorithm?
 - FIR is one type, but others require less computation at expense of lower-quality filtering
- Example: Quickly find item's address in 256-word memory
 - One use: data compression. Many others.
 - Algorithm 1: **"Linear search"**
 - 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 $M[mid] > \text{item}$, consider lower half of M
 - If $M[mid] < \text{item}$, 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



89

Power Optimization

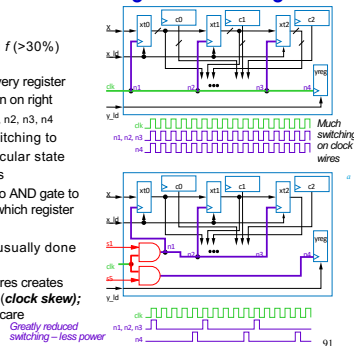
- Until now, we've focused on size and delay
- Power** is another important design criteria
 - Measured in Watts (energy/second)
 - Rate at which energy is consumed
- Increasingly important as more transistors fit on a chip
 - Power not scaling down at same rate as size
 - Means more heat per unit area – 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**)
 - $P = k \cdot C V^2 f$
 - 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?



90

Power Optimization using Clock Gating

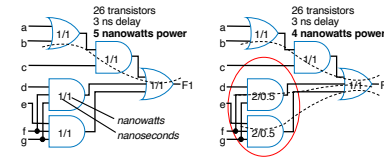
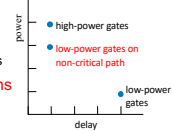
- $P = k \cdot CV^2f$
- Much of a chip's switching f (>30%) due to clock signals
 - After all, clock goes to every register
 - Portion of FIR filter shown on right
 - Notice clock signals $n1, n2, n3, n4$
- Solution: Disable clock switching to registers unused in a particular state
 - Achieve using AND gates
 - FSM only sets 2^N 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



91

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



92

92

Summary

- Optimization (improve criteria without loss) vs. tradeoff (improve criteria at expense of another)
- Combinational logic
 - K-maps and tabular method for two-level logic size optimization
 - Iterative improvement heuristics for two-level optimization and multi-level too
- Sequential logic
 - State minimization, state encoding, Moore vs. Mealy
- Datapath components
 - Faster adder using carry-lookahead
 - Smaller multiplier using sequential multiplication
- RTL
 - Pipelining, concurrency, component allocation, operator binding, operator scheduling
- (Optional) Serial vs. concurrent, efficient algorithms, power reduction methods (clock gating, low-power gates)
- (Optional) Multibillion dollar EDA industry (electronic design automation) creates tools for automated optimization/tradeoffs

93

93