Introduction to Digital Design
Week 13: Optimizations and Tradeoffs

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

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

Optimizations

- All criteria of interest least kept the same
- Optimizations are improved (or at least better)
- 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.

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

Tradeoffs

Some criteria of interest are improved, while others are worsened.

Example

F = xy + x'y'

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

F = xy + x'y'

Tradeoffs

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 + ab'c' is sum-of-products; G = a(x+y) + 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
### 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
    - \( ab + ab' = a \)  
    - \( a + b + c = a \)  
    - "Combining terms to eliminate a variable"
      - (Formally called the "Uniting theorem")  
    - Duplicating a term sometimes helps
      - Doesn't change function
    - Sometimes after combining terms, can combine resulting terms
      - \( G = xy' + x'y' + xy'z' + x'y'z' \)  
      - \( G = x(y' + y) + z(x' + x) \)  
      - \( G = x \)

### 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 \( x'y' + xy \)
        - Draw circle, write term that has all the literals except one that changes in the circle
        - Circle \( x + y \) & \( y' + y \) in both sides of the circle, but a change (\( y \) or \( y' \) in one cell, 0 in the other)
        - Minimized function: \( F \) is minimized algebraically

### 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

### 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
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:
\[ G = a' + a'b'c + bc' + bc' \]
1. Convert to sum-of-products
2. Place 1s in appropriate cells
3. Cover 1s
4. OR terms: \[ G = a + c' \]

Don’t Care Input Combinations

- What if we know that particular input combinations can never occur?
  - e.g., Minimize \( F = xy'z' \), given that \( x'y'z' \) \( (x'y'z') \) cannot be true, and that \( x'y'z' \) can never be true.
  - So it doesn’t matter what \( F \) outputs when \( x'y'z' \) or \( x'y'z' \) 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

Optimization Example using Don’t Cares

- Minimize:
  - \( F = a'b'c' + a'bc \)
  - 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

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:
\[ F = w'xz + yz + w'xy'z' \]
1. Convert to sum-of-products
2. Draw 1s for each product
3. Example:
   - \( F = \prod (0, 1, 3, 7) \)

Optimization Example Using K-maps

Common to revise (1) and (2):
- Create sum-of-products
- Draw 1s for each product

Two-Level Size Optimization Using K-maps

- Four Variable Example

Don’t Care Input Combinations

- What if we know that particular input combinations can never occur?
  - e.g., Minimize \( F = xy'z' \), given that \( x'y'z' \) \( (x'y'z') \) cannot be true, and that \( x'y'z' \) can never be true.
  - So it doesn’t matter what \( F \) outputs when \( x'y'z' \) or \( x'y'z' \) 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

Optimization Example using Don’t Cares

- Minimize:
  - \( F = a'b'c' + a'bc \)
  - 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
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

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 typically done by automated tools
  - Exact algorithm: finds optimal solution
  - Heuristic: finds good solution, but not necessarily optimal

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 (minterms or other) that when 1 causes F=1
    - On K-map, any legal (but not necessarily largest) circle
      - Cover: Implicant xy covers minterms xz and xy'
        - Expanding a term: removing a variable (like larger K-map circle)
        - xyz → xy is an expansion of xyz
  - Prime implicant: Maximal implicant that covers 1s not in on-set
    - x'y'z', and xy, above
      - BUT not xyz or xyz' — they can be expanded

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.
Tabular Method Step 2: Add Essential PIs to Cover

- Prime implicants (from Step 1): x'y', xy, z
  - (0) x'y'z'
  - (1) x'y'z
  - (3) x'yz
  - (5) xy'z
  - (6) xyz'
  - (7) xyz

Prime implicants

- Minterms
  - x'y'
    - (0, 1)
  - xy
    - (6, 7)
  - z
    - (1, 3, 5, 7)

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

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 = abcd'efg + abcd'efg' + jklmnop
    - Note: 15 variables, may have thousands of minterms
    - But can minimize just by combining first two terms:
      - F = abcd'efg(h+h') + jklmnop = abcd'efg + jklmnop

Ex: Iterative Hueristic for Two-Level Logic Size Optimization

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

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'a to a legal, but expanding x'z to z' not legal, as shown function
    - After expand, remove other terms covered by newly expanded term
    - Keep trying (iterate) until doesn’t help
  - Ex: F = abcd'efg + abcd'efg' + jklmnop
    - Covered by newly expanded term abcd'efg

Illustrated above on K-map, but iterative method is intended for an automated solution (no K-map)
**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
  - \( F_1 = ab + acd + ace \)
    \( \rightarrow \) \( F_2 = ab + ac(d + e) = a(b + c(d + e)) \)
- General technique: Factor out literals – \( xy + xz = x(y+z) \)

**Multi-Level Example**

- Q: Use multiple levels to reduce number of transistors for
  - \( F_1 = abcd + abcef \)
- A: \( abcd + abcef = abc(d + ef) \)
  - Has fewer gate inputs, thus fewer transistors

**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

**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

**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

**State Reduction: Equivalent States**

Two states are equivalent if:
1. They assign the same values to outputs
   - e.g. A and D both assign \( y \) to 0.
2. AND, for all possible sequences of inputs, the FSM outputs will be the same starting from either state
   - e.g. say \( x=1,1,0 \)
     - starting from A, \( y=0,1,1,1,1 \) \( (A,B,C,B) \)
     - starting from D, \( y=0,1,1,1,1 \) \( (D,B,C,B) \)
State Reduction via the Partitioning Method

- First partition states into groups based on outputs
- Then partition based on next states
  - For each possible input, for states in group, if next states in different groups, divide group
  - Repeat until no such states

Example: FSM that outputs 0, 1, 1, 1

- No such states
- Divide groups, divide states in possible input

Inputs: none; Outputs: x

Inputs: b; Outputs: x

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 16 gate inputs
  - Try alternative encoding
    - n0 = s0's1'b + s1s0'
    - n1 = s0b
    - n2 = s1b + s1a
    - Only 8 gate inputs
    - Thus fewer transistors

Example: FSM that outputs 0, 1, 1, 1

- Equations if one-hot encoding:
  - s3 + s2: n0 = s1; s3 + s3 = s1
- Fewer gates and only one level of logic – less delay than two levels, so faster clock frequency
Output Encodings

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

Moore vs. Mealy FSMs

- FSM implementation architecture
  - State register and logic
  - Next state logic — function of present state and FSM inputs
    - Moore FSM: Output logic — if function of present state only
    - Mealy FSM: Output logic — if function of present state and FSM inputs

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
  - \( n_0 = s_1's_0 + s_3s_2' \)
  - \( n_3 = s_1 + s_2 \)
  - \( n_2 = s_1 \)
  - \( n_1 = s_1's_0 \)

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 vs. Moore Tradeoff

- Mealy may have fewer states, but drawback is that its outputs change midcycle if input changes
  - Note earlier soda dispenser example
  - Represents a type of tradeoff

Soda dispenser example: Initialize, wait until enough, dispense

- Moore: 3 states; Mealy: 2 states

Assumes button press is synchronized to occur for one cycle only

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

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0 0 1 1 0 1</td>
<td></td>
</tr>
<tr>
<td>Run 1 1 0 1 0</td>
<td></td>
</tr>
</tbody>
</table>

Mealy and Moore can be Combined

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

Building a Faster Adder

- Carry-ripple is slow but small
  - 8-bit: $8^2 = 64$ gate delays, $8^5 = 40$ gates
  - 16-bit: $16^2 = 32$ gate delays, $16^5 = 80$ gates
  - 32-bit: $32^2 = 64$ gate delays, $32^5 = 160$ gates
- Two-level logic adder (2 gate delays)
  - OK for 4-bit adder: About 160 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: $2^3(2$ gate delays) = 4 gate delays, $2^3(100$ gates)$=300$ gates
  - 32-bit adder: $8^2$ gate delays) = 32 gate delays $8^2(100$ gates)$=800$ gates

<table>
<thead>
<tr>
<th>Can we do better?</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-bit adder</td>
</tr>
</tbody>
</table>

Faster Adder – (Naive 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

Inputs: b; Outputs: s1, s0, p

Notice – no rippling of carry
Faster Adder – (Naïve Inefficient) Attempt at “Lookahead”

- Want each stage’s carry-in bit to be function of external inputs only (a3, b3, or c3)

- Full-adder: s = a xor b, c = bc + ac + ab
  
  c1 = co0 + b0c0 + a0c0 + a0b0
  
  c2 = co1 + b1c1 + a1c1 + a1b1
  
  c2 = b1(b0c0 + a0c0 + a0b0) + a1(b0c0 + a0c0 + a0b0) + a1b1
  
  c3 = co2 + b2c2 + a2c2 + a2b2

(continue plugging in...)

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

Efficient Lookahead

- Substituting as in the naïve scheme:
  
  Gi = aibi (generate)

  Pi = ai XOR bi (propagate)

  
  c1 = G0 + P0c0
  
  c2 = G1 + P1c1
  
  c2 = G1 + P1(G0 + P0c0)
  
  c3 = G2 + P2c2
  
  c3 = G2 + P2(G1 + P1G0 + P1P0c0)
  
  c4 = G3 + P3c4

  c4 = G3 + P3G2 + P3P2G1 + P3P2P1G0 + P3P2P1P0c0

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

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)

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

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

Hierarchical Carry-Lookahead Adders

- Better solution – Rather than ripping 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

Hierarchical Carry-Lookahead Adders

- Carry Select Adder
  - Another way to compose adders
    - High-order stage – Compute result for carry in of 1 and 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

Multiplier in Ch 4 was array style

- But big for 32-bit: $32 \times 32 = 1024$ AND terms, and 31 adders
- 32-bit multiplier would have 1024 gates here...
### 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

### Smaller Multiplier — Sequential Style: Controller

- Wait for start
- 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

### 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**
  - You wash plate 1
  - Then friend rinses plate 1, while you wash plate 2
  - Then friend rinses 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)
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:
- Throughput: 
- Latency:

Main loop does $16 \times 16 = 256$ SADs.

Compute SAD for filter

Already used concurrency in FIR side; pipelining instead uses stages (like a factory line)

Concurrency does things side:
- Longest path now only 20 ns
- Clock frequency can be nearly doubled

Great speedup with minimal extra hardware

Concurrence: Divide task into subparts, execute subparts simultaneously
- Dishwashing example: Divide stack into 3 substacks, give substacks to 3 neighbors, who work simultaneously $\sim 3$ 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

Concurrence Example: SAD Design Revisited
- More concurrent design
  - Compute SAD for 16 pairs concurrently, do 16 times to compute all $16 \times 16$ SADs.
  - Main loop does $16$ sums per iteration, only $16$ items, still $2$ cycles per iter.

Concurrence Example: SAD Design Revisited
- Comparing the two designs
  - Original: $256$ iterations $\times 6$ cycles/iter $= 1536$ cycles
  - More concurrent: $16$ iterations $\times 2$ cycles/iter $= 32$ cycles
  - Speedup: 512/32 $= 16x$ speedup

Versus software
- Recall: Estimated about 6 microprocessor cycles per iteration
  - $256$ iterations $\times 6$ cycles per iteration $= 1536$ cycles
  - Original design speedup vs. software: $1068/512 = 3x$ (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

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

Operator Scheduling

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

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

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

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

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?

```
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
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 option 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: Tap-level logic (fast but big)
  - Serial: **Multi-level logic** (smaller but slower)
    - abc + ab + ef → (abc) + (ab) + ef → essentially computes ab first (serialized)

- **Datapath component tradeoffs**
  - Serial: **Carry-ripple adder** (small but slow)
  - Concurrent: **Carry-lookahead adder** (faster but bigger)
  - 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
  - Ex: **RTL decisions impact size/delay more than gate-level decisions**

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**
      - Finds memory containing item
    - Algorithm 2: **Binary search**
      - Example: Quickly find item’s address in 256-word memory
        - **Start** considering entire memory range
        - **Mid** of range, consider lower half if
        - **Repeat** on new smaller range
        - **Decision** if two steps, at least 8 such divisions

- **Choice of algorithm** has tremendous impact
  - Far more impact than say choice of comparator type

Power Optimization

- Until now, we’ve focused on size and delay
- **Power** is another important design criteria
  - Measured in Watts (energy/second)
  - Ex: rate at which energy is consumed
  - Increasingly important as more transistors fit on a chip
  - Power not scaling down at same rate as size
  - CMOS technology: Switching a wire from 0 to 1 consumes power (known as dynamic power)
  - P = k * CV^2
    - k: constant; C: capacitance of wire; V: voltage; f: switching frequency
  - Power reduction methods
    - Reduce power but slower, and there’s a limit
    - “What else?”

Power Optimization Using Clock Gating

- P = k * CV^2
- Much of a chip’s switching (>30%) due to clock signals
  - After all, clock goes to every register
  - Portion of FIR filter shown on right
    - Reduce clock signals
    - Solution: Disable clock switching to registers unused in a particular state
      - Achieve using AND gates
      - FIM 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
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

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