Penn ESE680-002 Spring2007 -- DeHon

ESE680-002: Computer Organization

Day 3: January 17, 2007
Arithmetic and Pipelining

Last Time

• Boolean logic ⇒ computing any finite function
• Sequential logic ⇒ computing any finite automata
  – included some functions of unbounded size
• Saw gates and registers
  – …and a few properties of logic

Today

• Addition
  – organization
  – design space
  – area, time
• Pipelining
• Temporal Reuse
  – area-time tradeoffs

Why?

• Start getting a handle on
  – Complexity
    • Area and time
    • Area-time tradeoffs
  – Parallelism
  – Regularity
• Arithmetic underlies much computation
  – grounds out complexity

Example: Bit Level Addition

• Addition
  – (everyone knows how to do addition base 2, right?)
C: 11011010000
A: 01101101010
B: 01100101100
S: 11110010110

Addition Base 2

• $A = a_{n-1} \cdot 2^{(n-1)} + a_{n-2} \cdot 2^{(n-2)} + \ldots + a_1 \cdot 2^1 + a_0 \cdot 2^0$
  $= \Sigma (a_i \cdot 2^i)$
• $S = A + B$
• What is the function for $s_i$ … carry? $s_i = (\text{xor carry}_{i-1} \cdot \text{xor a}_{i-1} \cdot b_i) + (a_{i-1} + b_{i-1} + \text{carry}_{i-1}) \geq 2$
  $= (\text{or} (\text{and} a_{i-1} \cdot b_{i-1}) \cdot (\text{and} a_{i-1} \cdot \text{carry}_{i-1})$
  $\cdot \text{and} b_{i-1} \cdot \text{carry}_{i-1})$
Adder Bit

• S = (xor a b carry)
• t = (xor2 a b); s = (xor2 t carry)
• xor2 = (and (not (and2 a b))
  (not (and2 (not a) (not b)))
• carry = (not (and2 (not (and2 a b))
  (and2 (not (and2 b carry)) (not (and2 a carry)))))

Ripple Carry Addition

• Shown operation of each bit
• Often convenient to define logic for each bit, then assemble:
  – bit slice

Ripple Carry Analysis

• Shown operation of each bit
• Often convenient to define logic for each bit, then assemble:
  – bit slice
• Area: O(N) [6n]
• Delay: O(N) [n+3]

Can we do better?

• Compute both possible values and
  select correct result when we know the answer

Important Observation

• Do we have to wait for the carry to show up to begin doing useful work?
  – We do have to know the carry to get the right answer.
  – But, it can only take on two values

Idea

• Compute both possible values and
  select correct result when we know the answer
Preliminary Analysis

- Delay(RA) -- Delay Ripple Adder
  - Delay(RA(n)) = kn
  - Delay(RA(n)) = 2*(kn/2) = 2*Delay(RA(n/2))

- Delay(P2A) -- Delay Predictive Adder
  - Delay(P2A) = Delay(RA(n/2)) + D(mux2)
  - ...almost half the delay!

Recurse

- If something works once, do it again.
- Use the predictive adder to implement the first half of the addition

Recurse

- If something works once, do it again.
- Use the predictive adder to implement the first half of the addition
  - Delay(P4A(n)) = Delay(RA(n/4)) + 2*D(mux2)

Recurse

- By know we realize we've been using the wrong recursion
  - should be using the Predictive Adder in the recursion
  - Delay(PA(n)) = Delay(PA(n/2)) + D(mux2)
  - Every time cut in half...?
  - How many times cut in half?
  - Delay(PA(n)) = \log_2(n) * D(mux2) + C

Another Way

(Parallel Prefix)
• Think about each adder bit as a computing a function on the carry in
  – \( C[i] = g(c[i-1]) \)
  – Particular function \( f \) will depend on \( a[i], b[i] \)
  – \( G = f(a, b) \)

• What functions can \( g(c[i-1]) \) be?
  – \( g(x) = 1 \)
    • \( a[i] = b[i] = 1 \)
  – \( g(x) = x \)
    • \( a[i] \ XOR b[i] = 1 \)
  – \( g(x) = 0 \)
    • \( a[i] = b[i] = 0 \)

• Want to combine functions
  – Compute \( c[i] = g_i(g_{i-1}(c[i-2])) \)
  – Compute compose of two functions
  • What functions will the compose of two of these functions be?
    – Same as before
      • Propagate, generate, squash

• \( GG = G \)
• \( GP = G \)
• \( GS = S \)
• \( PG = G \)
• \( PP = P \)
• \( PS = S \)
Combining

- Do it again...
- Combine $g[i-3,i-2]$ and $g[i-1,i]$  
- What do we get?

Reduce Tree

Prefix Tree

Parallel Prefix

- Important Pattern
- Applicable any time operation is associative
- Examples of associative functions?
  - Non-associative?
- Function Composition is always associative

Note: Constants Matter

- Watch the constants
- Asymptotically this Carry-Lookahead Adder (CLA) is great
- For small adders can be smaller with 
  - fast ripple carry  
  - larger combining than 2-ary tree
  - mix of techniques
- ...will depend on the technology primitives and cost functions

Two’s Complement

- Everyone seemed to know Two’s complement
- 2’s complement:
  - positive numbers in binary
  - negative numbers
    - subtract 1 and invert
    - (or invert and add 1)
Two’s Complement

- $2 = 010$
- $1 = 001$
- $0 = 000$
- $-1 = 111$
- $-2 = 110$

Addition of Negative Numbers?

- ...just works

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>010</td>
</tr>
<tr>
<td>000</td>
<td>111</td>
<td>001</td>
</tr>
</tbody>
</table>

Subtraction

- Negate the subtracted input and use adder
  - which is:
    - invert input and add 1
    - works for both positive and negative input

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>001</td>
<td>000</td>
</tr>
<tr>
<td>001</td>
<td>010</td>
<td>110</td>
</tr>
<tr>
<td>111</td>
<td>110</td>
<td>010</td>
</tr>
</tbody>
</table>

Overflow?

<table>
<thead>
<tr>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>110</td>
<td>111</td>
<td>111</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>010</td>
<td>110</td>
</tr>
<tr>
<td>000</td>
<td>111</td>
<td>001</td>
<td>101</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>S</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>111</td>
<td>001</td>
</tr>
<tr>
<td>010</td>
<td>100</td>
<td>011</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>011</td>
<td>111</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>100</td>
</tr>
<tr>
<td>010</td>
<td>100</td>
<td>011</td>
</tr>
</tbody>
</table>

- Overflow=(A.s==B.s)*(A.s!=S.s)

Subtraction (add/sub)

- **Note:** you can use the “unused” carry input at the LSB to perform the “add 1”

Reuse
Reuse

- In general, we want to reuse our components in time
  - not disposable logic
- How do we do that?
  - Wait until done, someone’s used output

Reuse: “Waiting” Discipline

- Use registers and timing (or acknowledgements) for orderly progression of data

Example: 4b Ripple Adder

- Recall 1 gates/F
- Latency and throughput?
- Latency: 4 gates to S3
- Throughput: 1 result / 4 gate delays max

Can we do better?

Stagger Inputs

- Correct if expecting A,B[3:2] to be staggered one cycle behind A,B[1:0]
- …and succeeding stage expects S[3:2] staggered from S[1:0]

Align Data / Balance Paths

Good discipline to line up pipe stages in diagrams.
Example: 4b RA pipe 2

- Recall 1 gates/FA
- Latency and Throughput?
- Latency: 4 gates to S3
- Throughput: 1 result / 2 gate delays max

Deeper?

- Can we do it again?
- What's our limit?
- Why would we stop?

More Reuse

- Saw could pipeline and reuse FA more frequently
- Suggests we're wasting the FA part of the time in non-pipelined

More Reuse (cont.)

- If we're willing to take 4 gate-delay units, do we need 4 FAs?

Ripple Add (pipe view)

Can pipeline to FA.

What if don't need the throughput?

If don't need throughput, reuse FA on SAME addition.

Bit Serial Addition

Assumes LSB first ordering of input data.
Bit Serial Addition: Pipelining

- Latency and throughput?
- Latency: 4 gate delays
- Throughput: 1 result / 5 gate delays
- Can squash Cout[3] and do in 1 result/4 gate delays
- Registers do have time overhead
  - setup, hold time, clock jitter

Multiplication

- Can be defined in terms of addition
- Ask you to play with implementations and tradeoffs in homework 2
  - Out today
  - Pickup from syllabus page on web

Compute Function

- Compute:
  \[ y = Ax^2 + Bx + C \]
- Assume
  - \( D(\text{Mpy}) > D(\text{Add}) \)
  - \( A(\text{Mpy}) > A(\text{Add}) \)

Spatial Quadratic

- \( D(\text{Quad}) = 2D(\text{Mpy}) + D(\text{Add}) \)
- Throughput \( 1/(2D(\text{Mpy}) + D(\text{Add})) \)
- \( A(\text{Quad}) = 3A(\text{Mpy}) + 2A(\text{Add}) \)

Pipelined Spatial Quadratic

- \( D(\text{Quad}) = 3D(\text{Mpy}) \)
- Throughput \( 1/D(\text{Mpy}) \)
- \( A(\text{Quad}) = 3A(\text{Mpy}) + 2A(\text{Add}) + 6A(\text{Reg}) \)

Bit Serial Quadratic

- Data width \( w \); one bit per cycle
- Roughly \( 1/w \)-th the area of pipelined spatial
- Roughly \( 1/w \)-th the throughput
- Latency just a little larger than pipelined
Quadratic with Single Multiplier and Adder?

- We've seen reuse to perform the **same** operation
  - pipelining
  - bit-serial, homogeneous datapath
- We can also reuse a resource in time to perform a different role.
  - Here: \(x^2, A(x^2), Bx\)
  - also: \((Bx)+c, (A^2x^2)+(Bx+c)\)

Quadratic Datapath

- Start with one of each operation
- (alternatives where build multiply from adds...e.g. homework)

Quadratic Datapath

- Multiplier servers multiple roles
  - \(x^2\)
  - \(A(x^2)\)
  - \(Bx\)
- Will need to be able to steer data (switch interconnections)

Quadratic Datapath

- Multiplier servers multiple roles
  - \(x^2\)
  - \(A(x^2)\)
  - \(Bx\)
- \(x, x^2, x\)
- \(x, A, B\)

Quadratic Datapath

- Adder servers multiple roles
  - \((Bx)+c\)
  - \((A^2x^2)+(Bx+c)\)
- one always mpy output
- \(C, Bx+C\)
Quadratic Datapath

• Add input register for x

Quadratic Control

• Now, we just need to control the datapath
• What control?
• Control:
  – LD x
  – LD x\times x
  – MA Select
  – MB Select
  – AB Select
  – LD Bx+C
  – LD Y

FSMD

• FSMD = FSM + Datapath
• Stylization for building controlled datapaths such as this (a pattern)
• Of course, an FSMD is just an FSM
  – it's often easier to think about as a datapath
  – synthesis, AP&R tools have been notoriously bad about discovering/exploiting datapath structure

Quadratic FSMD Control

• S0: if (go) LD_X; goto S1
  – else goto S0
• S1: MA_SEL=x, MB_SEL[1:0]=x, LD_x\times x
  – goto S2
• S2: MA_SEL=x, MB_SEL[1:0]=B
  – goto S3
• S3: AB_SEL=C, MA_SEL=x\times x, MB_SEL=A
  – goto S4
• S4: AB_SEL=Bx+C, LD_Y
  – goto S0
Quadratic FSM Control

- S0: if (go) LD_X; goto S1
  - else goto S0
- S1: MA_SEL=x, MB_SEL[1:0]=x, LD_x*x
  - goto S2
- S2: MA_SEL=x, MB_SEL[1:0]=B
  - goto S3
- S3: AB_SEL=C, MA_SEL=x, MB_SEL=A
  - goto S4
- S4: AB_SEL=Bx+C, LD_Y
  - goto S0

Quadratic FSM

- Latency/Throughput/Area?
- Latency: 5*(D(MPY)+D(mux3))
- Throughput: 1/Latency
- Area: A(Mpy)+A(Add)+5*A(Reg) +2*A(Mux2)+A(Mux3)+A(QFSM)

Big Ideas

- Can build arithmetic out of logic
- Pipelining:
  - increases parallelism
  - allows reuse in time (same function)
- Control and Sequencing
  - reuse in time for different functions
- Can tradeoff Area and Time

Big Ideas

- Area-Time Tradeoff in Adders
- Parallel Prefix
- FSMD control style