CS184a: Computer Architecture (Structure and Organization)

Day 3: January 13, 2003

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

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

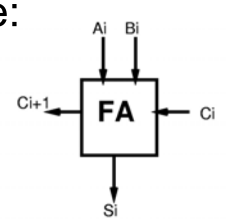
- $A = a_{n-1}*2^{(n-1)}+a_{n-2}*2^{(n-2)}+\ldots a_1*2^1+ a_0*2^0$
  $= \Sigma (a_i*2^i)$
- S=A+B
- s_i=(xor $carry_i$ (xor a_i b_i))
- $carry_i = (a_{i-1} + b_{i-1} + carry_{i-1}) \geq 2$
  = (or (and a_{i-1} b_{i-1}) (and a_{i-1} $carry_{i-1}$)
  (and b_{i-1} $carry_{i-1}$))

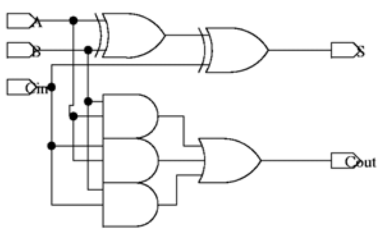
Adder Bit

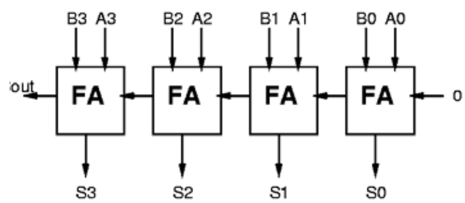
- S=(xor a b carry)
- t=(xor2 a b); s=(xor2 t carry)
- xor2 = (and (not (and2 a b)
  (not (and2 (not a) (not b)))
- carry = (not (and2 (not (and2 a b)) (and2 (not (and2 b carry)) (not (and2 a carry)))))

Ripple Carry Addition

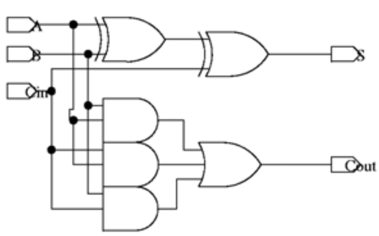
- Shown operation of each bit
- Often convenient to define logic for each bit, then assemble:
  - **bit slice**

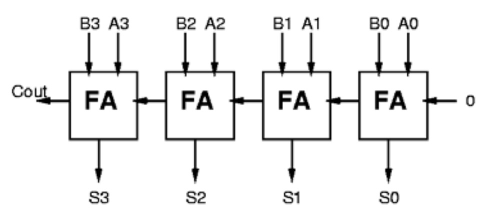






Ripple Carry Analysis





- Area: O(N) [6n]
- Delay: O(N) [2n]

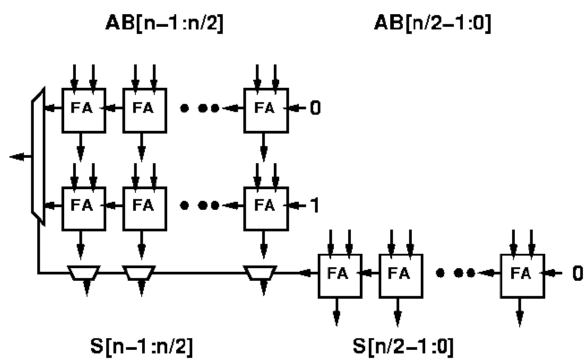
Can we do better?

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



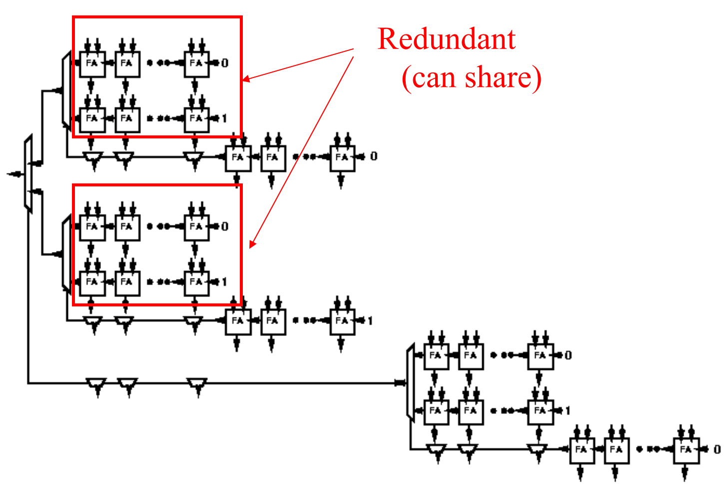
Preliminary Analysis

- DRA--Delay Ripple Adder
- DRA(n) = k*n
- DRA(n) = 2*DRA(n/2)
- DP2A-- Delay Predictive Adder
- DP2A=DRA(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



Recurse

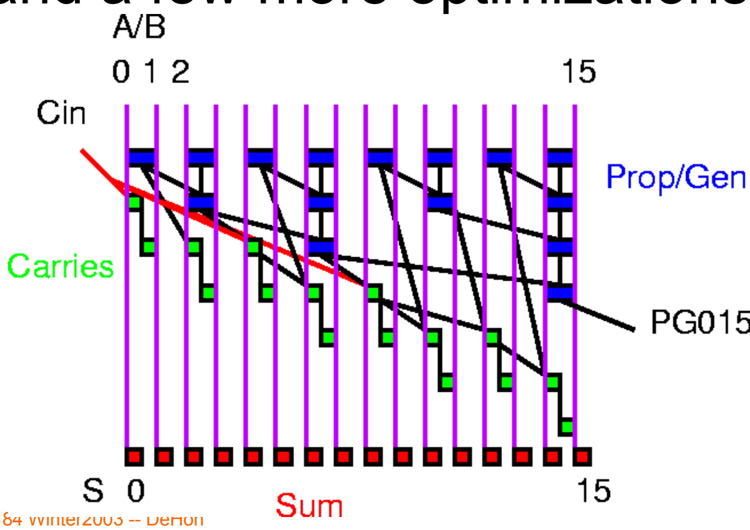
- If something works once, do it again.
- Use the predictive adder to implement the first half of the addition

- DP4A(n)=DRA(n/4) + D(mux2) + D(mux2)
- DP4A(n)=DRA(n/4)+2*D(mux2)

Recurse

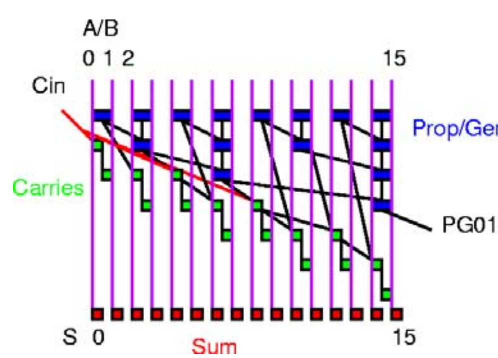
- By know we realize we've been using the wrong recursion
  - should be using the DPA in the recursion
- DPA(n) = DPA(n/2) + D(mux2)
- DPA(n)=$\log_2$(n)*D(mux2)+C

Resulting RPA [and a few more optimizations]



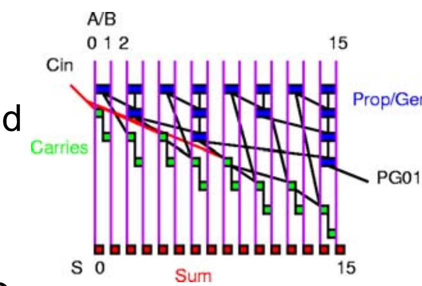
RPA Analysis

- Delay: O(log(n))
- Area: O(n)
  - maybe n log(n) when consider wiring...
- bounded fanout

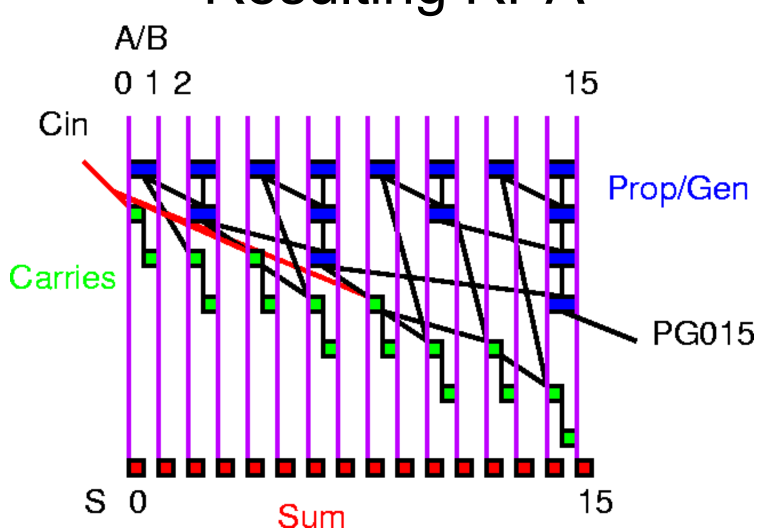


Constructive RPA

- Each block (I,J) may
  - propagate or squash a carry in
  - generate a carry out
  - can compute PG(I,J)
    - in terms of PG(I,K) and PG(K,J) (I<K<J)
- PG(I,J) + carry(I)
  - is enough to calculate carry(J)



Resulting RPA



Note: Constants Matter

- Watch the constants
- Asymptotically this RPA 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

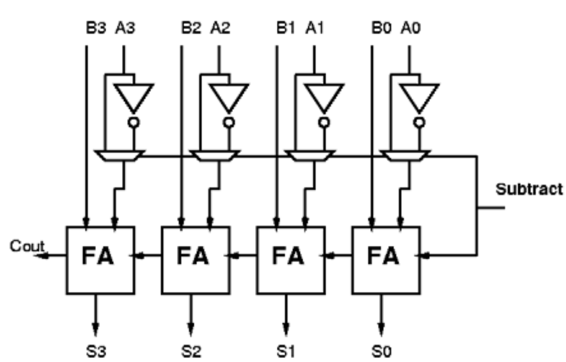
A: 111	A: 110	A: 111	A: 111
B: 001	B: 001	B: 010	B: 110
S: 000	S: 111	S: 001	S: 101

Subtraction

- Negate the subtracted input and use adder
  - which is:
    - invert input and add 1
    - works for both positive and negative input
      - 001 → 110 +1 = 111
      - 111 → 000 +1 = 001
      - 000 → 111 +1 = 000
      - 010 → 101 +1 = 110
      - 110 → 001 +1 = 010

Subtraction (add/sub)

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



Overflow?

A: 111	A: 110	A: 111	A: 111
B: 001	B: 001	B: 010	B: 110
S: 000	S: 111	S: 001	S: 101

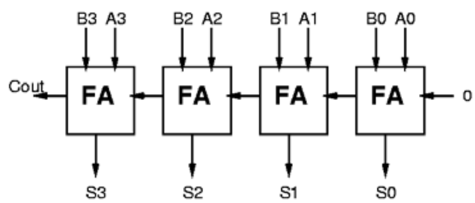
A: 001	A: 011	A: 111
B: 001	B: 001	B: 100
S: 010	S: 100	S: 011

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

Reuse

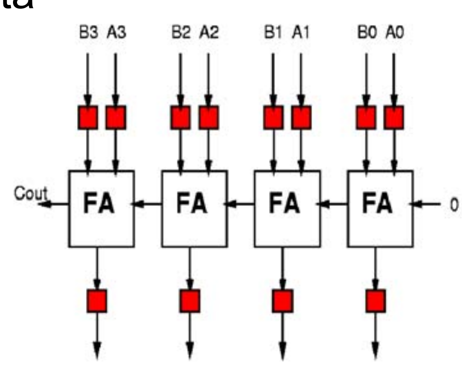
Reuse

- 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

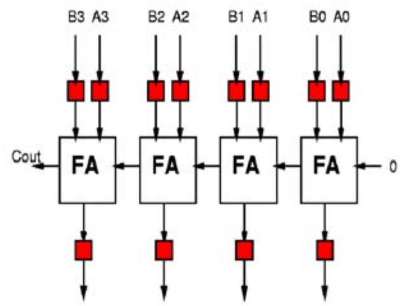


Reuse: "Waiting" Discipline

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

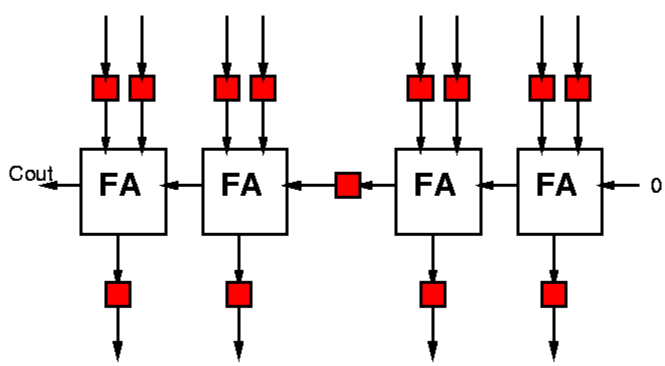


Example: 4b Ripple Adder



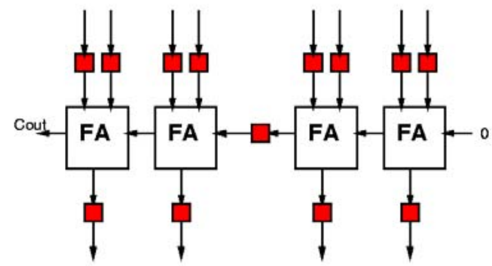
- Recall 2 gates/FA
- Latency: 8 gates to S3
- Throughput: 1 result / 8 gate delays max

Can we do better?



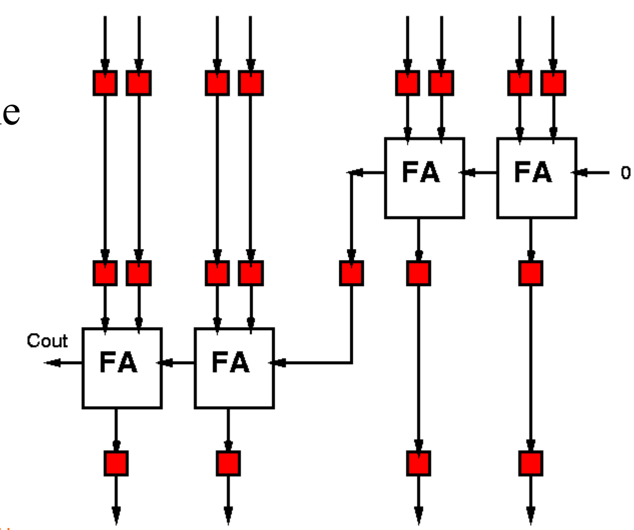
Stagger Inputs

- 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]

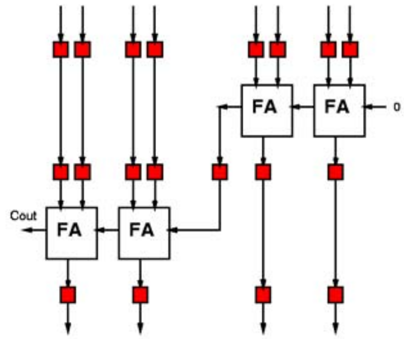


Align Data / Balance Paths

Good discipline to line up pipe stages in diagrams.



Example: 4b RA pipe 2



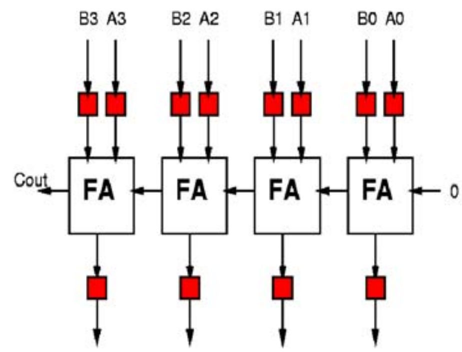
- Recall 2 gates/FA
- Latency: 8 gates to S3
- Throughput: 1 result / 4 gate delays max

Deeper?

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

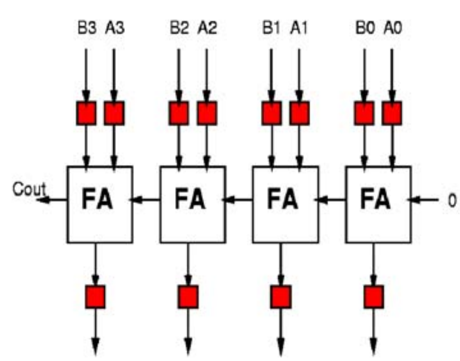
More Reuse

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



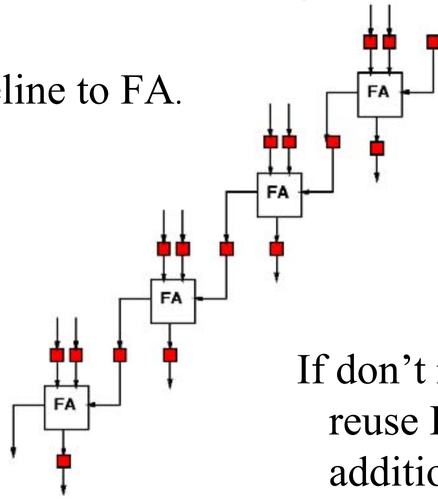
More Reuse (cont.)

- If we're willing to take 8 gate-delay units, do we need 4 FAs?



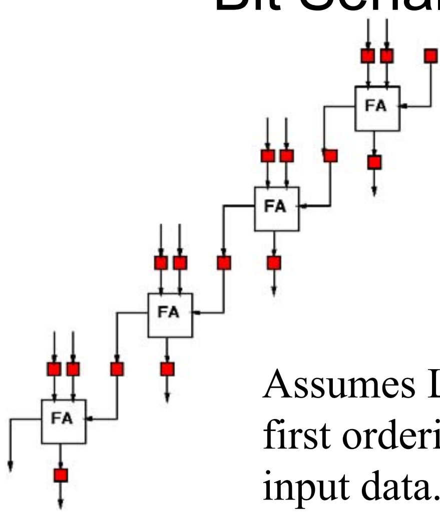
Ripple Add (pipe view)

Can pipeline to FA.

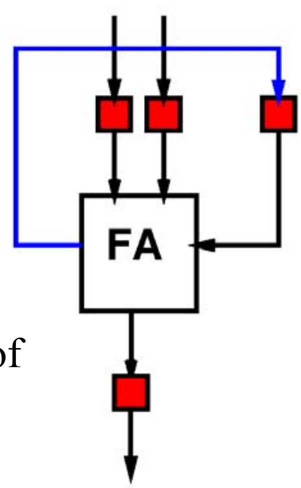


If don't need throughput, reuse FA on **SAME** addition.

Bit Serial Addition

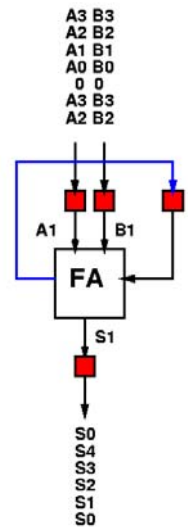


Assumes LSB first ordering of input data.



Bit Serial Addition: Pipelining

- Latency: 8 gate delays
- Throughput: 1 result / 10 gate delays
- Can squash Cout[3] and do in 1 result/8 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

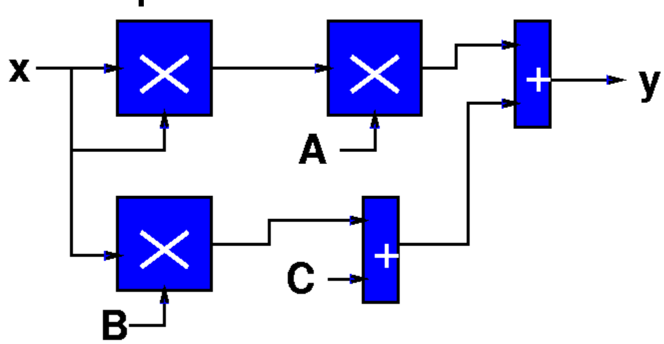
Compute Function

- Compute:

$$y=Ax^2 +Bx +C$$

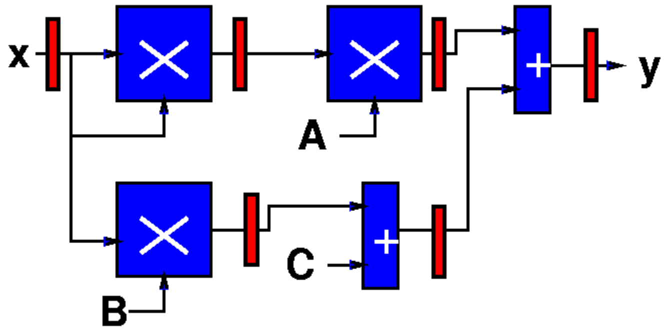
- Assume
  – D(Mpy) > D(Add)
  – A(Mpy) > A(Add)

Spatial Quadratic



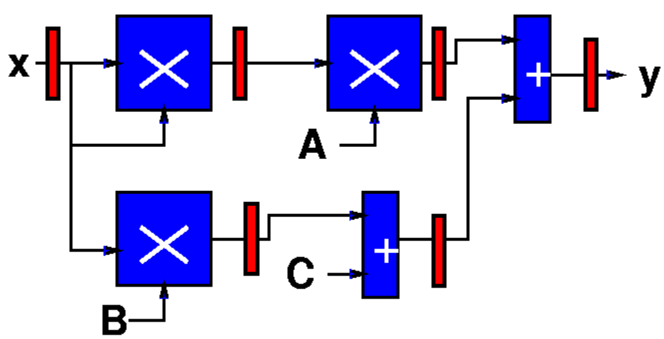
- D(Quad) = 2*D(Mpy)+D(Add)
- Throughput 1/(2*D(Mpy)+D(Add))
- A(Quad) = 3*A(Mpy) + 2*A(Add)

Pipelined Spatial Quadratic



- D(Quad) = 3*D(Mpy)
- Throughput 1/D(Mpy)
- A(Quad) = 3*A(Mpy) + 2*A(Add)+6A(Reg)

Bit Serial Quadratic



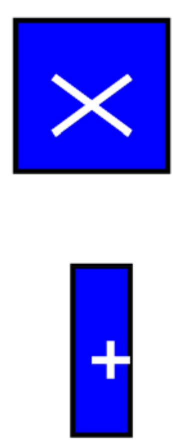
- data width w; one bit per cycle
- roughly 1/w-th the area of pipelined spatial
- roughly 1/w-th the throughput
- latency just a little larger than pipelined

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*x, A*(x*x), B*x
  - also: (Bx)+c, (A*x*x)+(Bx+c)

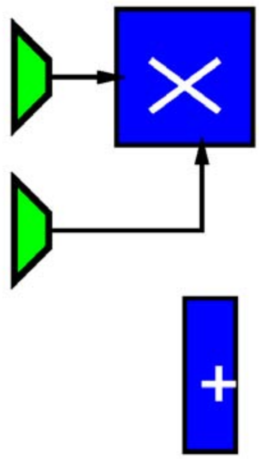
Quadratic Datapath

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



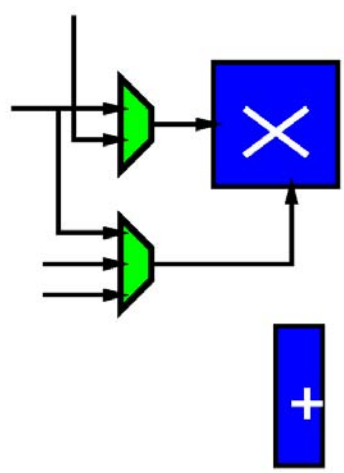
Quadratic Datapath

- Multiplier servers multiple roles
  - x*x
  - A*(x*x)
  - B*x
- Will need to be able to steer data (switch interconnections)



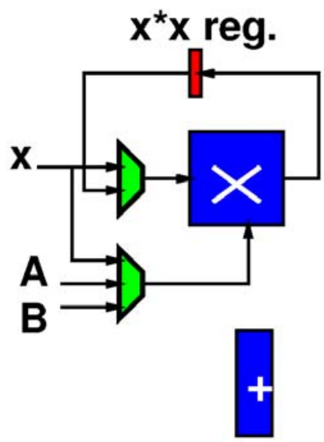
Quadratic Datapath

- Multiplier servers multiple roles
  - x*x
  - A*(x*x)
  - B*x
- x, x*x
- x,A,B



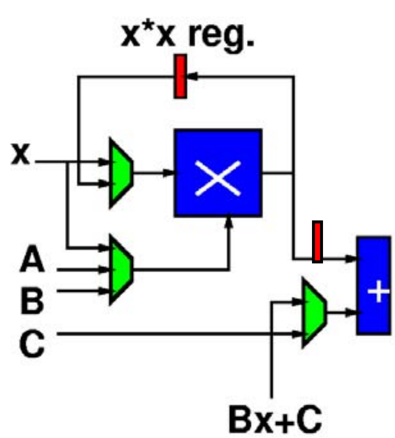
Quadratic Datapath

- Multiplier servers multiple roles
  - x*x
  - A*(x*x)
  - B*x
- x, x*x
- x,A,B

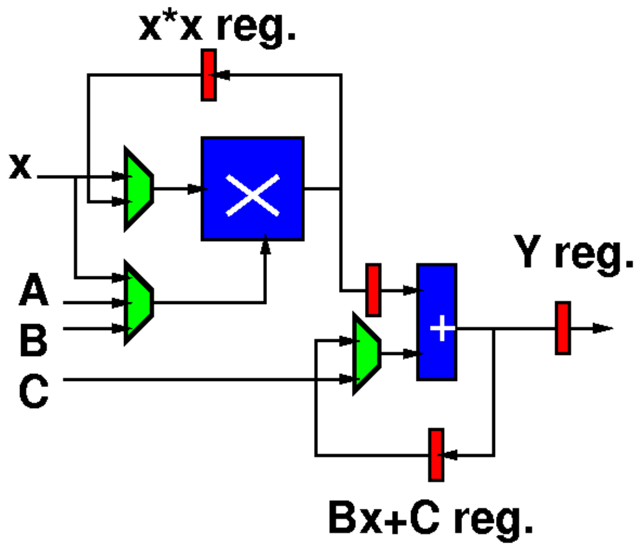


Quadratic Datapath

- Adder servers multiple roles
  - (Bx)+c
  - (A*x*x)+(Bx+c)
- one always mpy output
- C, Bx+C

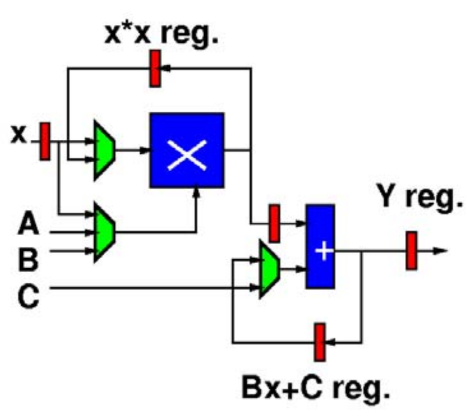


Quadratic Datapath



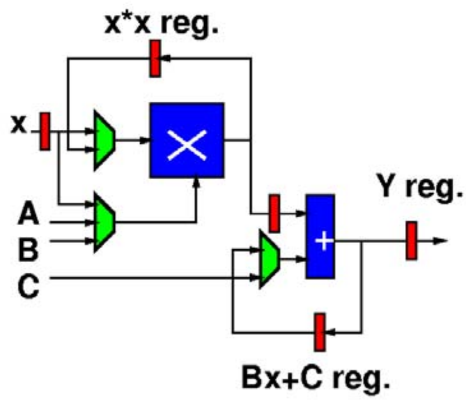
Quadratic Datapath

- Add input register for x



Quadratic Control

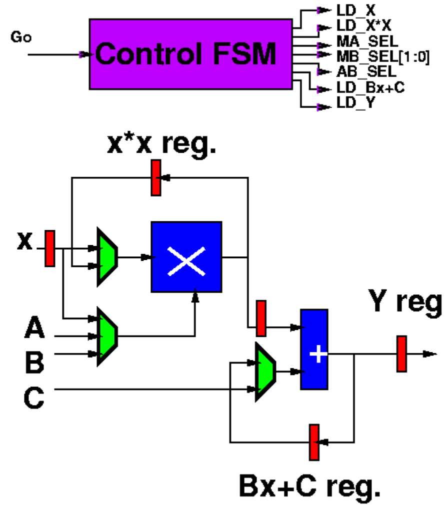
- Now, we just need to control the datapath
- Control:
  - LD x
  - LD x*x
  - MA Select
  - MB Select
  - AB Select
  - LD Bx+C
  - LD Y



FSMD

- FSMD = FSM + Datapath
- Stylization for building controlled datapaths such as this
- 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

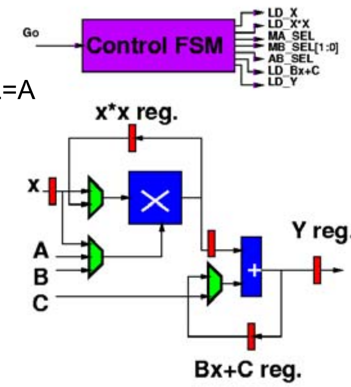


Quadratic FSMD Control

- 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*x, MB_SEL=A
  - goto S4
- S4: AB_SEL=Bx+C, LD_Y
  - goto S0

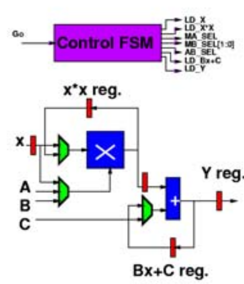
Quadratic FSMD Control

- 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*x, MB_SEL=A
  - goto S4
- S4: AB_SEL=Bx+C, LD_Y
  - goto S0



Quadratic FSM

- 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
[MSB 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
[MSB-1 Ideas]

- Area-Time Tradeoff in Adders
- FSMD control style