Introduction to VLSI Design Datapath

Tsung-Chu Huang
Electronics Eng., NCUE
2016/5/10

(I) Adders

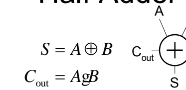
- Single-bit Addition
- Carry-Ripple Adder
- Carry-Skip Adder
- Carry-Lookahead Adder
- Carry-Select Adder
- Carry-Increment Adder
- Tree Adder

Single-Bit Addition

Half Adder

$S = A \oplus B$

$C_{out} = A \cdot B$

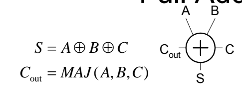


A	B	C_{out}	S
0	0		
0	1		
1	0		
1	1		

Full Adder

$S = A \oplus B \oplus C$

$C_{out} = MAJ(A, B, C)$



A	B	C	C_{out}	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

PGK

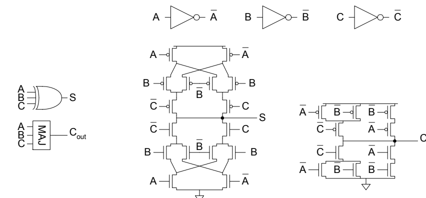
- For a full adder, define what happens to carries
  (in terms of A and B)
  - Generate: C_{out} = 1 independent of C
    - ✓G = A • B
  - Propagate: C_{out} = C
    - ✓P = A ⊕ B
  - Kill: C_{out} = 0 independent of C
    - ✓K = ~A • ~B

Full Adder Design I

- Brute force implementation from eqns

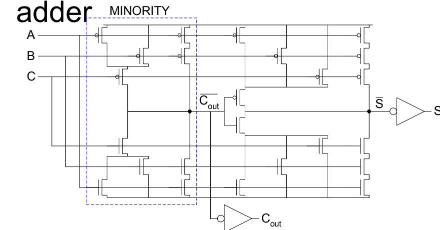
$$S = A \oplus B \oplus C$$

$$C_{out} = MAJ(A, B, C)$$



Full Adder Design II

- Factor S in terms of C_{out}
  S = ABC + (A + B + C)(~C_{out})
- Critical path is usually C to C_{out} in ripple adder

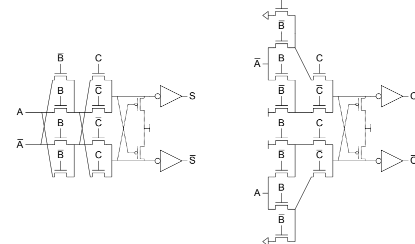


Layout

- Clever layout circumvents usual line of diffusion
  - Use wide transistors on critical path
  - Eliminate output inverters

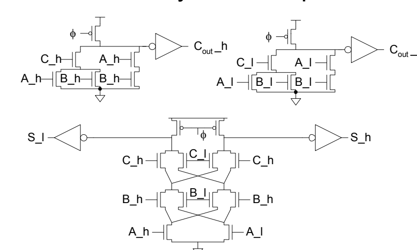
Full Adder Design III

- Complementary Pass Transistor Logic (CPL)
  - Slightly faster, but more area



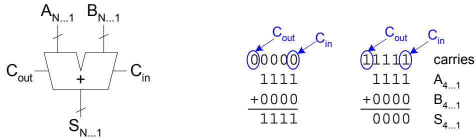
Full Adder Design IV

- Dual-rail domino
  - Very fast, but large and power hungry
  - Used in very fast multipliers



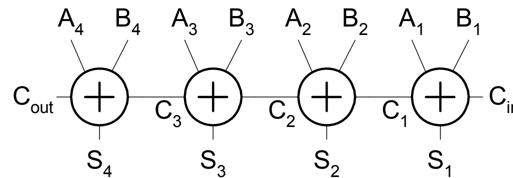
Carry Propagate Adders

- N-bit adder called CPA
  - Each sum bit depends on all previous carries
  - How do we compute all these carries quickly?



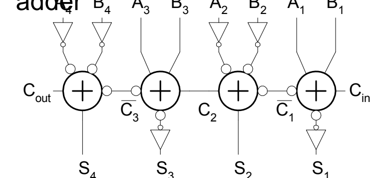
Carry-Ripple Adder

- Simplest design: cascade full adders
  - Critical path goes from C_{in} to C_{out}
  - Design full adder to have fast carry delay



Inversions

- Critical path passes through majority gate
  - Built from minority + inverter
  - Eliminate inverter and use inverting full adder



Generate / Propagate

- Equations often factored into G and P
- Generate and propagate for groups spanning i:j

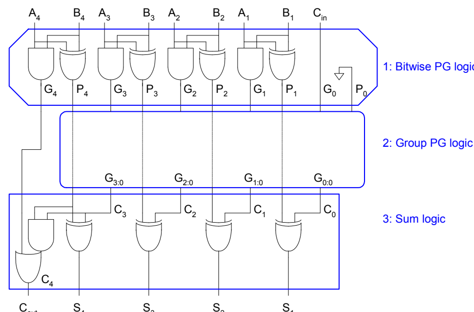
$$G_{i:j} = G_{i:k} + P_{i:k} \cdot G_{k-1:j}$$
$$P_{i:j} = P_{i:k} \cdot P_{k-1:j}$$

- Base case

$$G_{i:i} \equiv G_i = A_i \cdot B_i \qquad G_{0:0} \equiv G_0 = C_{in}$$
$$P_{i:i} \equiv P_i = A_i \oplus B_i \qquad P_{0:0} \equiv P_0 = 0$$

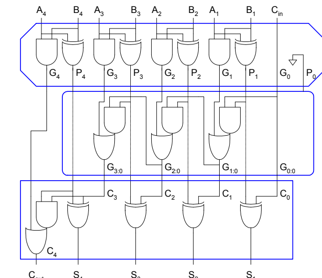
- Sum: $S_i = P_i \oplus G_{i-1:0}$

PG Logic



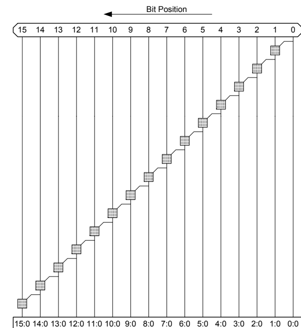
Carry-Ripple Revisited

$$G_{i:0} = G_i + P_i \cdot G_{i-1:0}$$

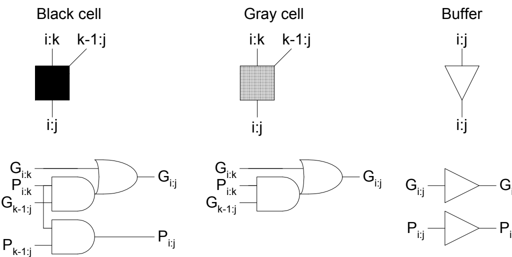


Carry-Ripple PG Diagram

$$t_{ripple} = t_{pg} + (N-1)t_{AO} + t_{xor}$$

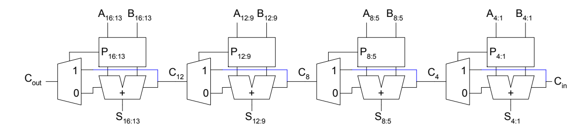


PG Diagram Notation

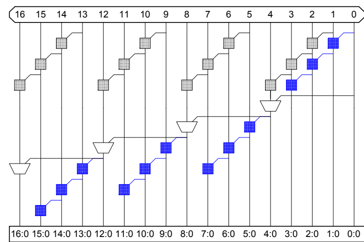


Carry-Skip Adder

- Carry-ripple is slow through all N stages
- Carry-skip allows carry to skip over groups of n bits
  - Decision based on n-bit propagate signal



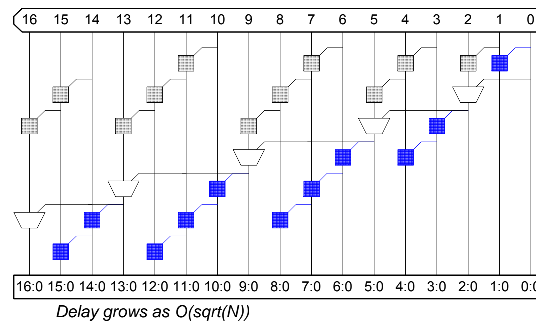
Carry-Skip PG Diagram



$$t_{\text{skip}} = t_{pg} + \left[2(n-1) + (k-1)\right] t_{AO} + t_{\text{xor}}$$

For k n-bit groups (N = nk)

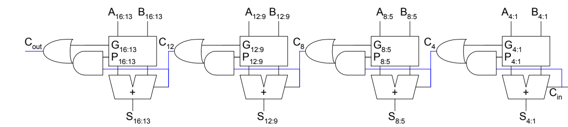
Variable Group Size



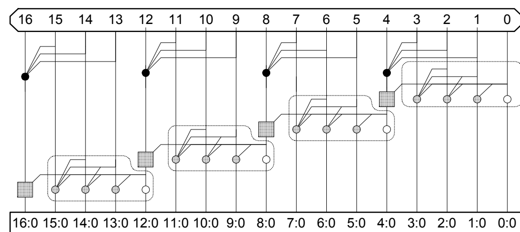
Delay grows as O(sqrt(N))

Carry-Lookahead Adder

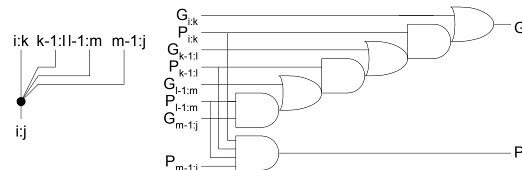
- Carry-lookahead adder computes $G_{i:0}$ for many bits in parallel.
- Uses higher-valency cells with more than two inputs.



CLA PG Diagram

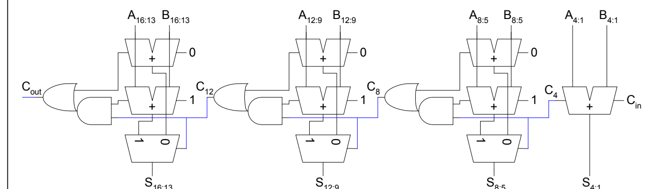


Higher-Valency Cells



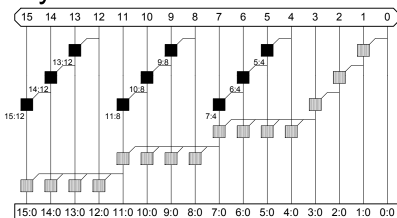
Carry-Select Adder

- Trick for critical paths dependent on late input X
  - Precompute two possible outputs for X = 0, 1
  - Select proper output when X arrives
- Carry-select adder precomputes n-bit sums
  - For both possible carries into n-bit group



Carry-Increment Adder

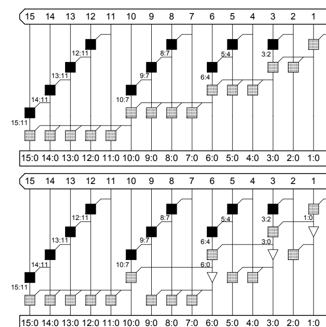
- Factor initial PG and final XOR out of carry-select



$$t_{\text{increment}} = t_{pg} + \left[(n-1) + (k-1)\right] t_{AO} + t_{\text{xor}}$$

Variable Group Size

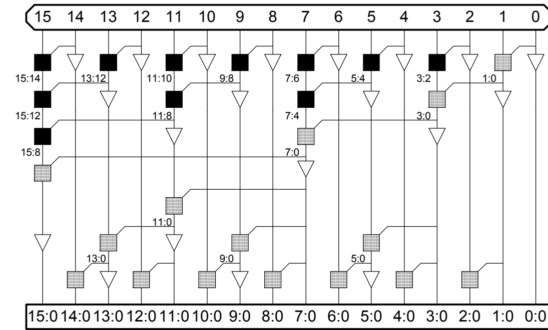
- Also buffer noncritical signals



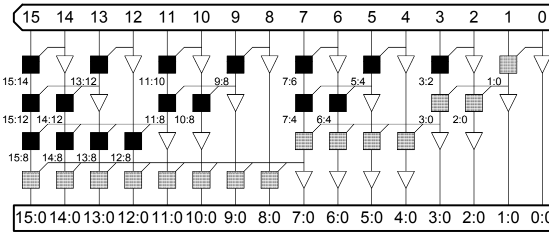
Tree Adder

- If lookahead is good, lookahead across lookahead!
  - Recursive lookahead gives O(log N) delay
- Many variations on tree adders

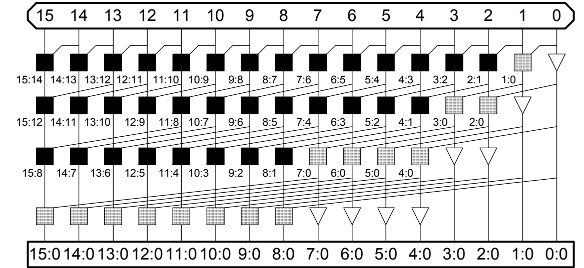
Brent-Kung



Sklansky



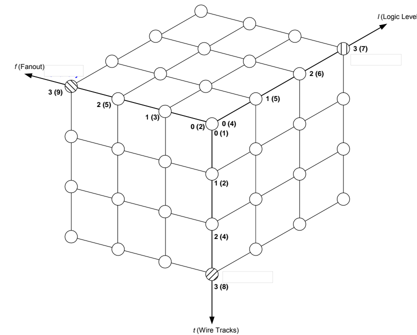
Kogge-Stone



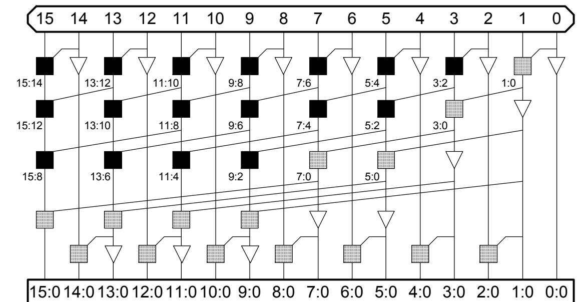
Tree Adder Taxonomy

- Ideal N-bit tree adder would have
  - *L* = log N logic levels
  - Fanout never exceeding 2
  - No more than one wiring track between levels
- Describe adder with 3-D taxonomy (*l*, *f*, *t*)
  - Logic levels: $L + l$
  - Fanout: $2^f + 1$
  - Wiring tracks: 2^t
- Known tree adders sit on plane defined by $l + f + t = L-1$

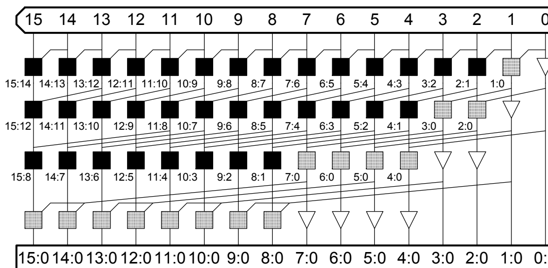
Tree Adder Taxonomy



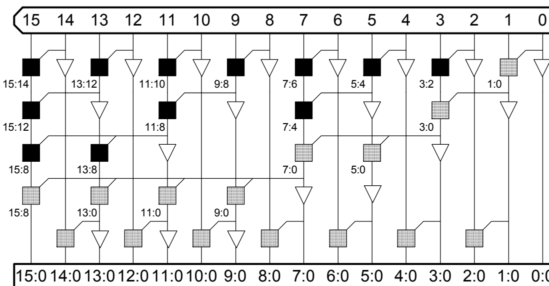
Han-Carlson



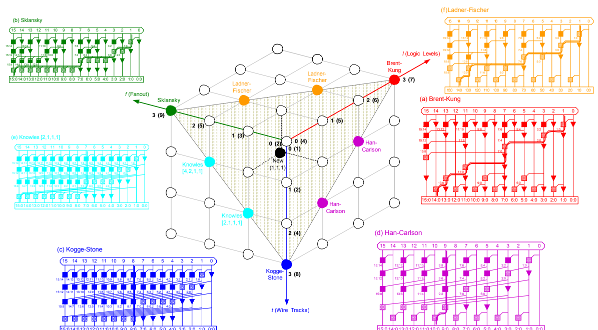
Knowles [2, 1, 1, 1]



Ladner-Fischer



Taxonomy Revisited



Summary

Adder architectures offer area / power / delay tradeoffs.

Choose the best one for your application.

Architecture	Classification	Logic Levels	Max Fanout	Tracks	Cells
Carry-Ripple		N-1	1	1	N
Carry-Skip n=4		N/4 + 5	2	1	1.25N
Carry-Inc. n=4		N/4 + 2	4	1	2N
Brent-Kung	(L-1, 0, 0)	$2\log_2 N - 1$	2	1	2N
Sklansky	(0, L-1, 0)	$\log_2 N$	N/2 + 1	1	$0.5 N\log_2 N$
Kogge-Stone	(0, 0, L-1)	$\log_2 N$	2	N/2	$N\log_2 N$

(II) Basic Functions of Datapath

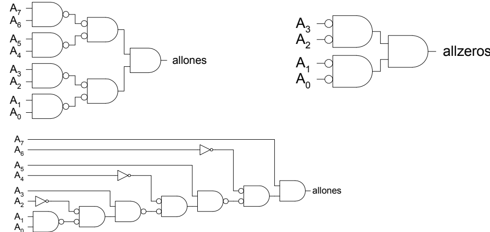
- Comparators
- Shifters
- Multi-input Adders
- Multipliers

Comparators

- 0's detector: A = 00…000
- 1's detector: A = 11…111
- Equality comparator: A = B
- Magnitude comparator: A < B

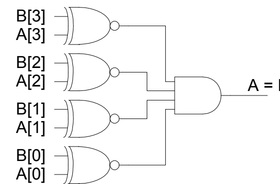
1's & 0's Detectors

- 1's detector: N-input AND gate
- 0's detector: NOTs + 1's detector (N-input NOR)



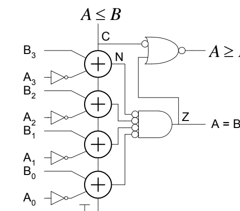
Equality Comparator

- Check if each bit is equal (XNOR, aka equality gate)
- 1's detect on bitwise equality



Magnitude Comparator

- Compute B – A and look at sign
- B – A = B + ~A + 1
- For unsigned numbers, carry out is sign bit



Signed vs. Unsigned

- For signed numbers, comparison is harder
  - C: carry out
  - Z: zero (all bits of B – A are 0)
  - N: negative (MSB of result)
  - V: overflow (inputs had different signs, output sign ≠ B)
  - S: N xor V (sign of result)

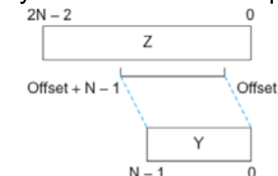
Relation	Unsigned Comparison	Signed Comparison
$A = B$	Z	Z
$A \neq B$	$\bar{Z}$	$\bar{Z}$
$A < B$	$C \cdot \bar{Z}$	$\bar{S} \cdot \bar{Z}$
$A > B$	C	S
$A \leq B$	C	$\bar{S}$
$A \geq B$	$\bar{C} + Z$	$S + Z$

Shifters

- Logical Shift:
  - Shifts number left or right and fills with 0's
    - ✓1011 LSR 1 = 0101 1011 LSL1 = 0110
- Arithmetic Shi
  - Shifts number left or right. Rt shift sign extends
    - ✓1011 ASR1 1011 ASL1
- Rotate:
  - Shifts number left or right and fills with lost bits
    - ✓1011 ROR1 = 1101 1011 ROL1 = 0111

Funnel Shifter

- A funnel shifter can do all six types of shifts
- Selects N-bit field Y from 2N–1-bit input
  - Shift by k bits ($0 \leq k < N$)
  - Logically involves N N:1 multiplexers

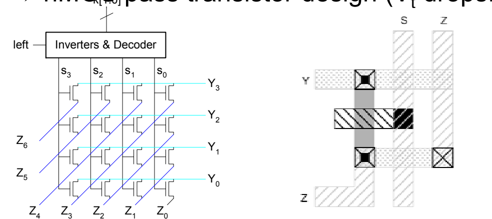


Funnel Source Generator

Shift Type	$Z_{2N-2:N}$	Z_{N-1}	$Z_{N-2:0}$	Offset
Rotate Right	$A_{N-2:0}$	A_{N-1}	$A_{N-2:0}$	k
Logical Right	0	A_{N-1}	$A_{N-2:0}$	k
Arithmetic Right	sign	A_{N-1}	$A_{N-2:0}$	k
Rotate Left	$A_{N-1:1}$	A_0	$A_{N-1:1}$	$\bar{k}$
Logical/Arithmetic Left	$A_{N-1:1}$	A_0	0	$\bar{k}$

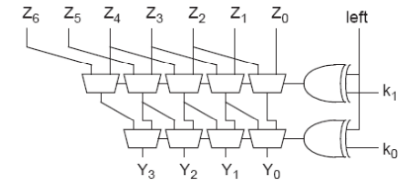
Array Funnel Shifter

- N N-input multiplexers
  - Use 1-of-N hot select signals for shift amount
  - nMOS pass transistor design (V_t drops!)



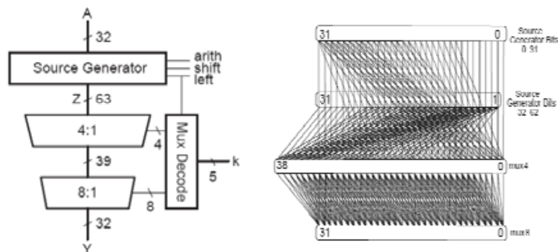
Logarithmic Funnel Shifter

- Log N stages of 2-input muxes
  - No select decoding needed



32-bit Logarithmic Funnel

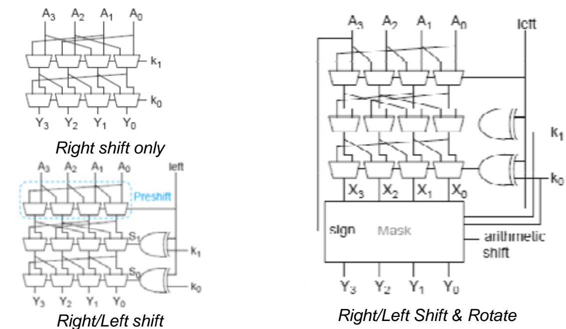
- Wider multiplexers reduce delay and power
- Operands > 32 bits introduce datapath irregularity



Barrel Shifter

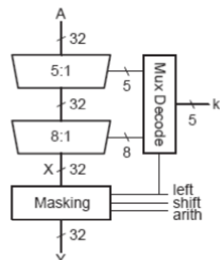
- Barrel shifters perform right rotations using wrap-around wires.
- Left rotations are right rotations by N – k = k + 1 bits.
- Shifts are rotations with the end bits masked off.

Logarithmic Barrel Shifter



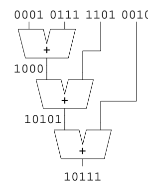
32-bit Logarithmic Barrel

- Datapath never wider than 32 bits
- First stage preshifts by 1 to handle left shifts



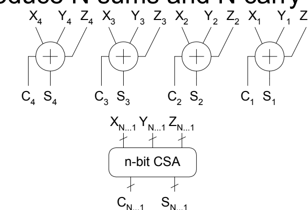
Multi-input Adders

- Suppose we want to add k N-bit words
  - Ex: 0001 + 0111 + 1101 + 0010 = 10111
- Straightforward solution: k-1 N-input CPAs
  - Large and slow



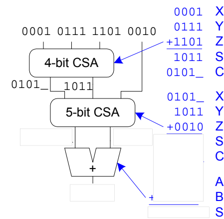
Carry Save Addition

- A full adder sums 3 inputs and produces 2 outputs
  - Carry output has twice *weight* of sum output
- N full adders in parallel are called *carry save adder*
  - Produce N sums and N carry outs



CSA Application

- Use k-2 stages of CSAs
  - Keep result in carry-save redundant form
- Final CPA computes actual result



Multiplication

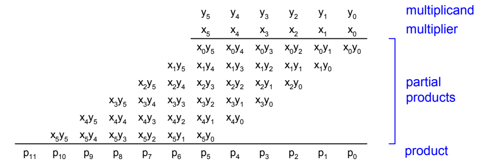
- Example:

```
      1100 : 12_10     multiplicand
      0101 : 5_10      multiplier
      1100
     0000              partial
    1100               products
   0000
  00111100 : 60_10     product
```

- M x N-bit multiplication
  - Produce N M-bit partial products
  - Sum these to produce M+N-bit product

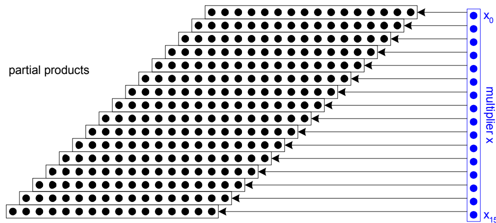
General Form

- Multiplicand: $Y = (y_{M-1}, y_{M-2}, \ldots, y_1, y_0)$
- Multiplier: $X = (x_{N-1}, x_{N-2}, \ldots, x_1, x_0)$
- Product: $P = \left(\sum_{j=0}^{M-1} y_j 2^j\right)\left(\sum_{i=0}^{N-1} x_i 2^i\right) = \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} x_i y_j 2^{i+j}$

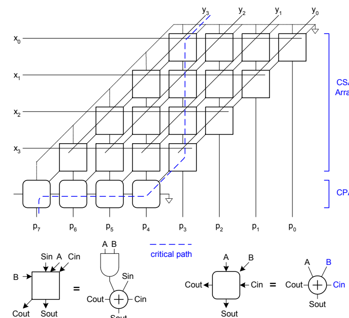


Dot Diagram

- Each dot represents a bit

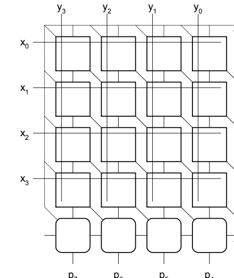


Array Multiplier



Rectangular Array

- Squash array to fit rectangular floorplan



Fewer Partial Products

- Array multiplier requires N partial products
- If we looked at groups of r bits, we could form N/r partial products.
  - Faster and smaller?
  - Called radix-2^r encoding
- Ex: r = 2: look at pairs of bits
  - Form partial products of 0, Y, 2Y, 3Y
  - First three are easy, but 3Y requires adder ☹

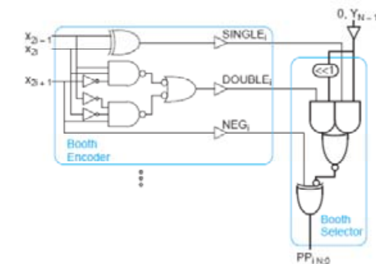
Booth Encoding

- Instead of 3Y, try –Y, then increment next partial product to add 4Y
- Similarly, for 2Y, try –2Y + 4Y in next partial product

Inputs			Partial Product	Booth Selects		
x_{2i+1}	x_{2i}	x_{2i-1}	PP_i	$SINGLE_i$	$DOUBLE_i$	NEG_i
0	0	0	0	0	0	0
0	0	1	Y	1	0	0
0	1	0				
0	1	1				
1	0	0				
1	0	1				
1	1	0				
1	1	1				

Booth Hardware

- Booth encoder generates control lines for each PP
  - Booth selectors choose PP bits

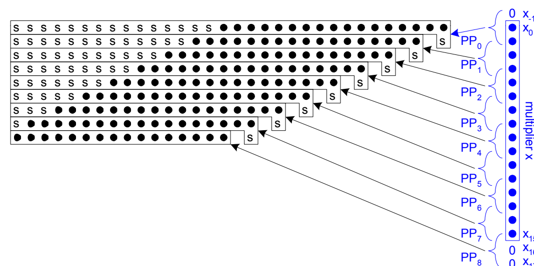


An 8-bit Booth Multiplier in Verilog

```
module multiplier(prod, busy, mc, mp, clk, start);
   output [15:0] prod; output busy;
   input [7:0] mc, mp; input clk, start;
   reg [7:0] A, Q, M; reg Q_1;
   reg [3:0] count;
   wire [7:0] sum, difference;
   always @(posedge clk)
     begin
       if (start) begin A <= 8'b0;
         M <= mc;
         Q <= mp;
         Q_1 <= 1'b0;
         count <= 4'b0;
       end else begin
         case ({Q[0], Q_1})
           2'b0_1 : {A, Q, Q_1} <= {sum[7], sum, Q};
           2'b1_0 : {A, Q, Q_1} <= {difference[7],
                                     difference, Q};
           default: {A, Q, Q_1} <= {A[7], A, Q};
         endcase
         count <= count + 1'b1;
       end
     end
   alu adder (sum, A, M, 1'b0);
   alu subtracter (difference, A, ~M, 1'b1);
   assign prod = {A, Q};
   assign busy = (count < 8);
endmodule
```

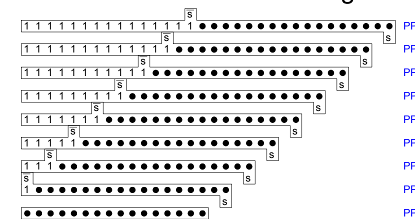

Sign Extension

- Partial products can be negative
  - Require sign extension, which is cumbersome
  - High fanout on most significant bit



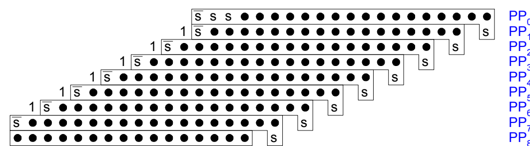
Simplified Sign Ext.

- Sign bits are either all 0's or all 1's
  - Note that all 0's is all 1's + 1 in proper column
  - Use this to reduce loading on MSB



Even Simpler Sign Ext.

- No need to add all the 1's in hardware
  - Precompute the answer!



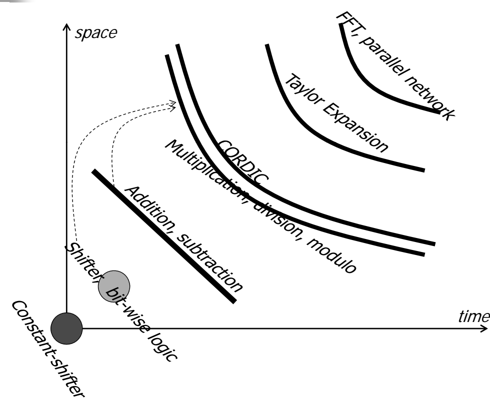
Advanced Multiplication

- Signed vs. unsigned inputs
- Higher radix Booth encoding
- Array vs. tree CSA networks

(III) Advanced Datapaths

- Division (y/x): shift + Add/Sub
- Reciprocal ($1/x$): Newton's Method
- Multiplier: Shifter + Parallel Adder
- Square-Root ($\sqrt{x}$): Babylonian Method
- cos(x), sin(x): CORDIC Algorithm
- exp(x), log(x): CORDIC Algorithm

Time-Space Complexity



General Coordinate Rotation Digital Computation

1. Vector function $f(x) = f(x - x_i) \cdot g(x_i)$
2. $g(x_i) = d_{i0}2^0 + d_{i1}2^1 + d_{i2}2^2 + \cdots + d_{im_i}2^{m_i}$
3. The fewer items the finer!
   - Basic CORDIC fine vector: $\begin{pmatrix} 1 & -2^{m_i} \\ 2^{m_i} & 1 \end{pmatrix}$
   - Fine numbers for exp, log: $2^0 + 2^{m_j}$
4. One end is precisive.
5. Convergence
6. Convergence rate is efficient.
   - ➔ dichotomy

CORDIC-Able Functions

- All standard, inverse, hyperbolic trigonometric functions
- All linear combinations of trigonometric functions are potential "cordicible"

Algorithm ➔ C Code ➔ Verilog Code

```c
#define K 0x26DD3B6A
#define half_pi 0x6487ED51
int Tab[] = {0x3243F6A8, 0x1DAC6705, 0x0FADBAFC, 0x07F56EA6,
0x03FEAB76, 0x01FFD55B,
0x00FFFAAA, 0x007FFF55, 0x003FFFEA, 0x001FFFFD, 0x000FFFFF,
0x0007FFFF, 0x0003FFFF,
0x0001FFFF, 0x0000FFFF, 0x00007FFF, 0x00003FFF, 0x00001FFF,
0x00000FFF, 0x000007FF,
0x000003FF, 0x000001FF, 0x000000FF, 0x0000007F, 0x0000003F,
0x0000001F, 0x0000000F,
0x00000008, 0x00000004, 0x00000002, 0x00000001, 0x00000000, };

void cordic(int theta, int *s, int *c, int n)
{
  int k, d, tx, ty, tz;
  int x=K,y=0,z=theta;
  for (k=0; k<n; ++k)
  {
    d = z>>31;
    tx = x - (((y>>k) ^ d) - d);
    ty = y + (((x>>k) ^ d) - d);
    tz = z - ((Tab[k] ^ d) - d);
    x = tx; y = ty; z = tz;
  }
 *c = x; *s = y;
}
```


Algorithm ➔ C Code ➔ Verilog Code

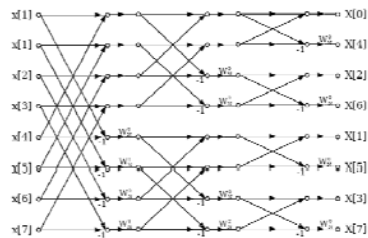
```verilog
module CORDIC32(Clk, Rst, theta, sin, cos);
        input  Clk, Rst;
        input  [31:0] theta; // angle in fixed-point <2.30>
        output [31:0] sin, cos; // sine and cosine in <2.30>

        reg    [31:0] sin, cos, phi, A;
        reg    [5:0]  k;
        wire   [31:0] sin2k, cos2k, delta_sin, delta_cos, delta_A;
        assign sin2k = sin[31] ? ~((~sin)>>k) : sin >> k;
        assign cos2k = cos[31] ? ~((~cos)>>k) : cos >> k;
        assign delta_sin = phi[31] ? (~sin2k + 1) : sin2k;
        assign delta_cos = phi[31] ? (~cos2k + 1) : cos2k;
        assign delta_A = phi[31] ? (~A + 1) : A;

        always@(posedge Clk)
                if(Rst)
                        begin
                                phi <= theta;
                                sin <= 32'h00000000;
                                cos <= 32'h26DD3B6A;
                                k   <= 0;
                        end
                else
                        if(!k[5])
                                begin
 begin
                                        cos <= cos - delta_sin;
                                        sin <= sin + delta_cos;
                                        phi <= phi - delta_A;
                                        k   <= k   + 1;
                                end
```


Algorithm ➔ C Code ➔ Verilog Code

```verilog
        always@(*)
                case(k[4:0])
                        0: A = 32'h3243F6A8;
                        1: A = 32'h1DAC6705;
                        2: A = 32'h0FADBAFC;
                        3: A = 32'h07F56EA6;
                        4: A = 32'h03FEAB76;
                        5: A = 32'h01FFD55B;
                        6: A = 32'h00FFFAAA;
                        7: A = 32'h007FFF55;
                        8: A = 32'h003FFFEA;
                        9: A = 32'h001FFFFD;
                        10: A = 32'h000FFFFF;
                        11: A = 32'h0007FFFF;
                        12: A = 32'h0003FFFF;
                        13: A = 32'h0001FFFF;
                        14: A = 32'h0000FFFF;
                        15: A = 32'h00007FFF;
                        16: A = 32'h00003FFF;
                        17: A = 32'h00001FFF;
                        18: A = 32'h00000FFF;
                        19: A = 32'h000007FF;
                        20: A = 32'h000003FF;
                        21: A = 32'h000001FF;
                        22: A = 32'h000000FF;
                        23: A = 32'h0000007F;
                        24: A = 32'h0000003F;
                        25: A = 32'h0000001F;
                        26: A = 32'h0000000F;
                        27: A = 32'h00000008;
                        28: A = 32'h00000004;
                        29: A = 32'h00000002;
                        30: A = 32'h00000001;
                        31: A = 32'h00000000;
                endcase
endmodule
```

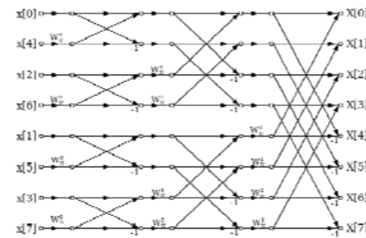

Fast Fourier Transform (FFT)

- Decimation in Frequency (DIF)



Fast Fourier Transform (FFT)

- Decimation in Time (DIT)



Infinite-Impulse-Response (IIR) Filter