Design of Datapath elements in Digital Circuits

Debdeep Mukhopadhyay

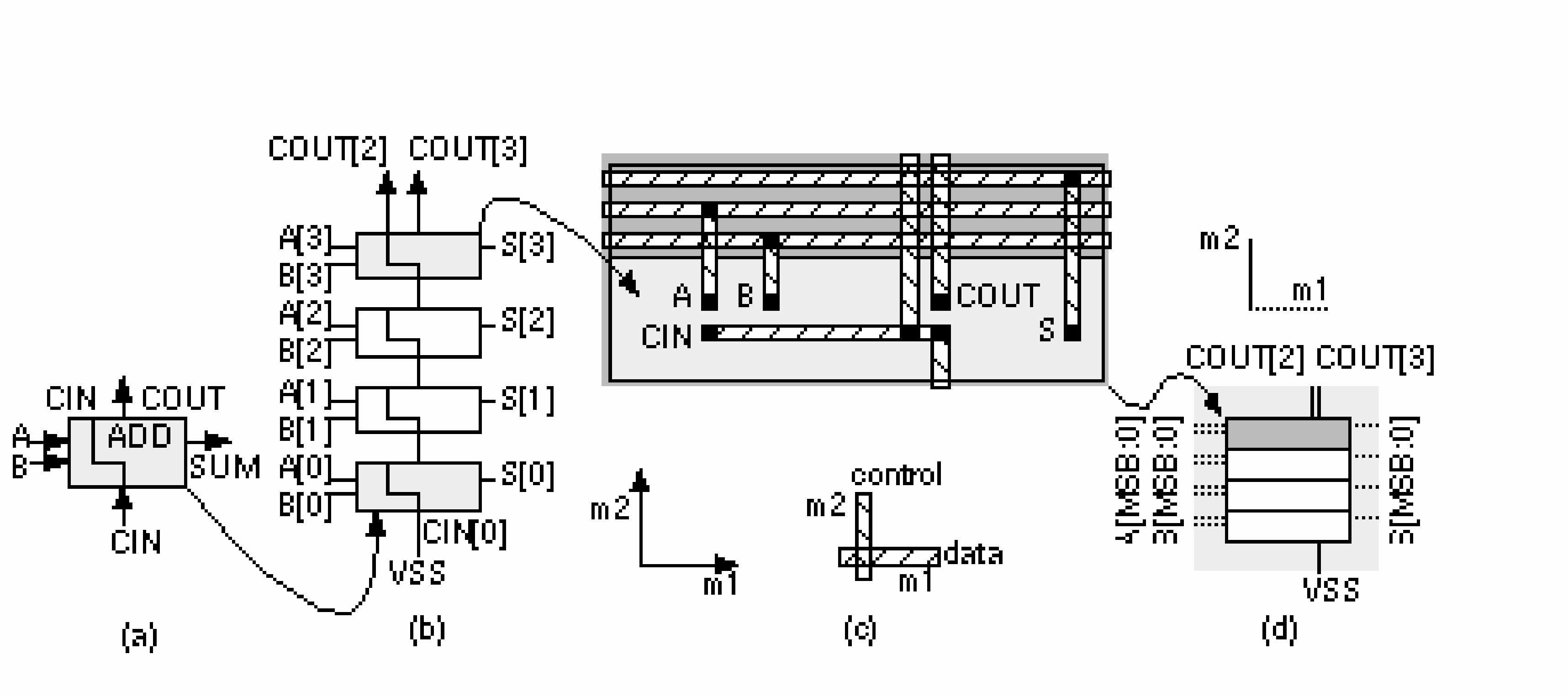
IIT Madras

What is datapath?

- Suppose we want to design a Full Adder (FA):
  - Sum=A ^ B ^ CIN = Parity(A,B,CIN)
  - COUT=AB+ACIN+BCIN=MAJ(A,B,CIN)
- Combine the two functions to a single FA logic cell:

  ADD(A[i],B[i],CIN,S[i],COUT)
- How do we build a 4-bit ripple carry adder?

A 4 bit Adder



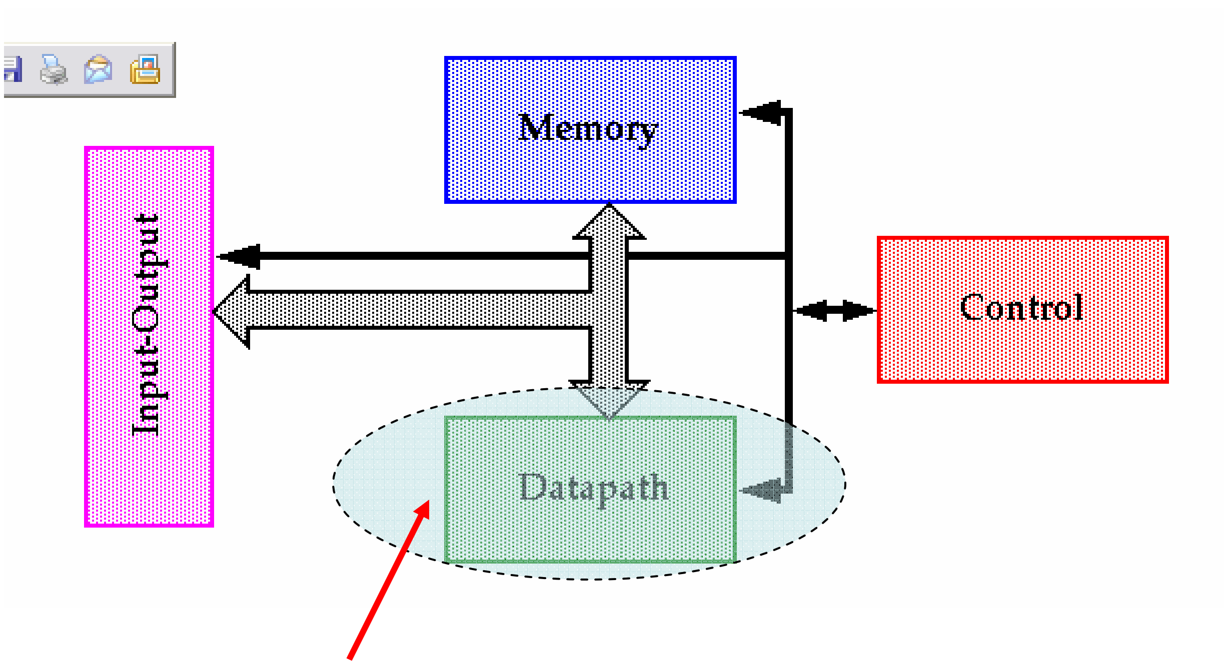
The layout of buswide logic that operates on data signals is called a Datapath.

The module ADD is called a Datapath element.

What is the difference between datapath and standard cells?

- Standard Cell Based Design: Cells are placed together in rows but there is no generally no regularity to the arrangement of the cells within the rows—we let software arrange the cells and complete the interconnect.
- Datapath layout automatically takes care of most of the interconnect between the cells with the following advantages:
  - Regular layout produces predictable and equal delay for each bit.
  - Interconnect between cells can be built into each cell.

Digital Device Components

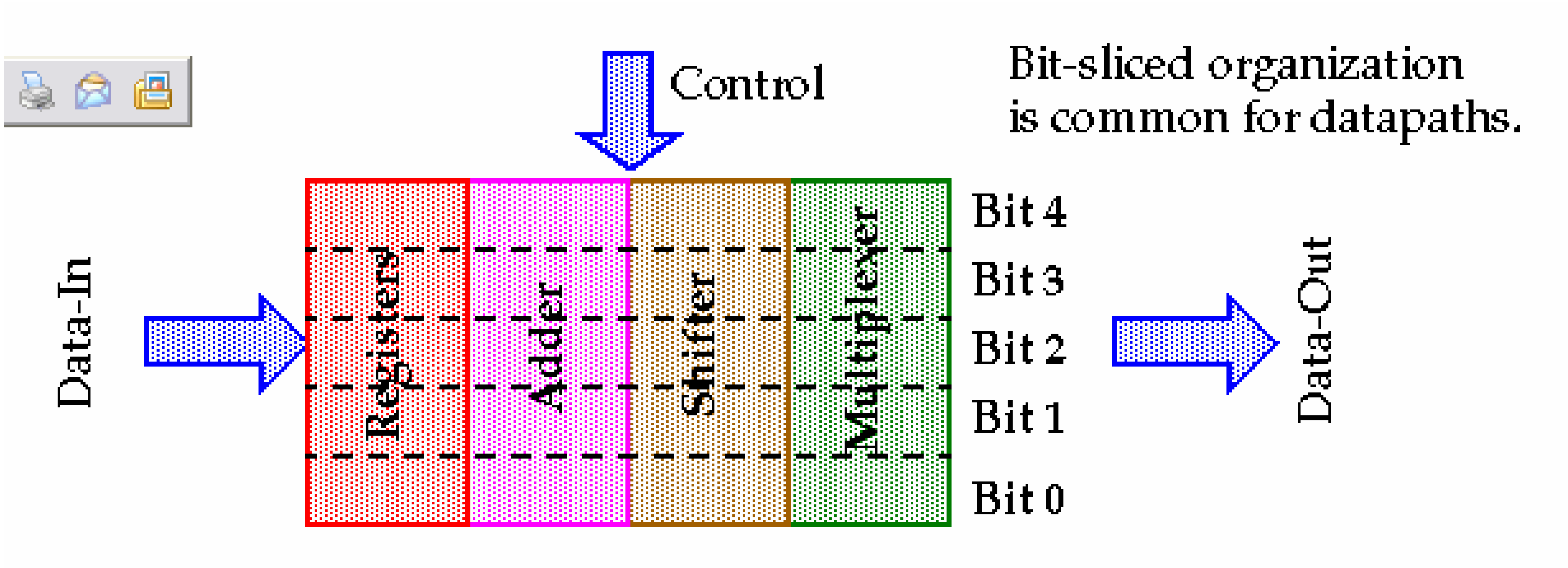


- We shall concentrate first on this.

Why Datapaths?

- *The speed of these elements often dominates the overall system performance so optimization techniques are important.*
- *However, as we will see, the task is non-trivial since there are multiple equivalent logic and circuit topologies to choose from, each with adv./disadv. in terms of speed, power and area.*
- *Datapath elements include shifters, adders, multipliers, etc.*

Bit slicing



How can we develop architectures which are bit sliced?

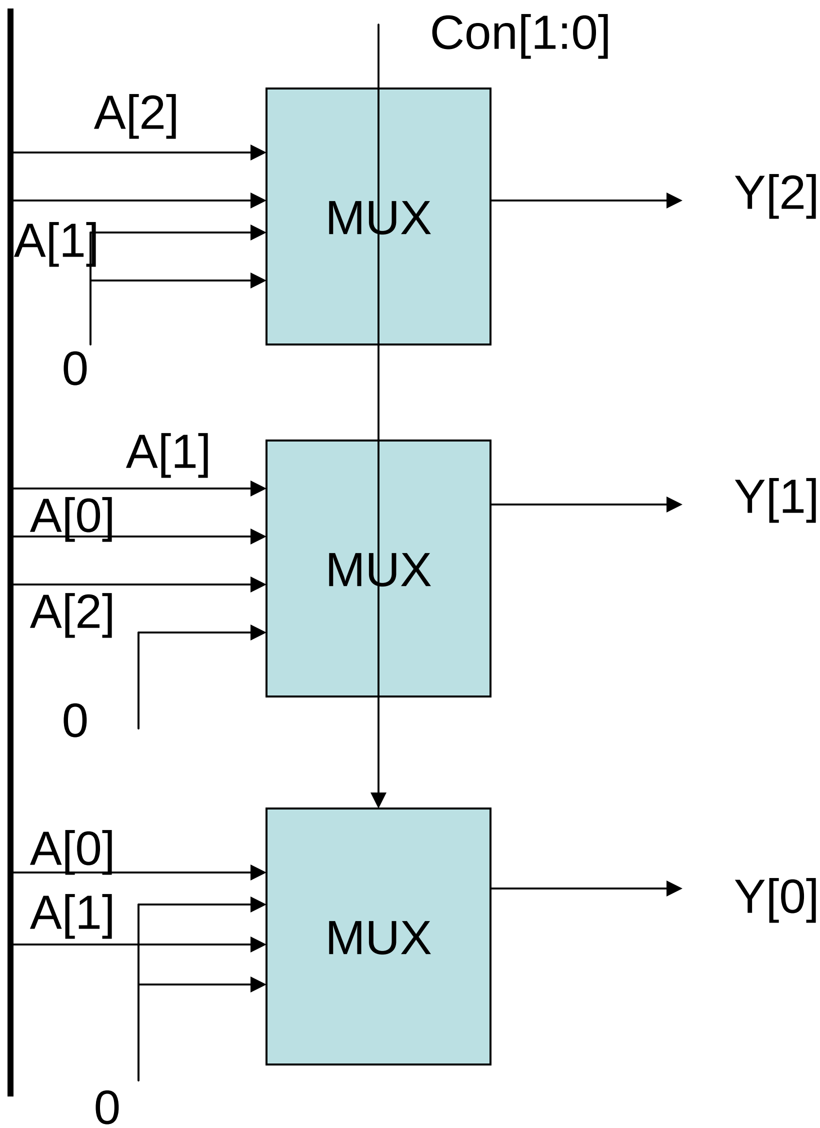
Datapath Elements

Shifters

Sel1	Sel0	Operation	Function
0	0	Y<-A	No shift
0	1	Y<-shlA	Shift left
1	0	Y<-shrA	Shift right
1	1	Y<-0	Zero outputs

What would be a bit sliced architecture of this simple shifter?

Using Muxes



Verilog Code

```
module shifter(Con,A,Y);
        input [1:0] Con;
        input[2:0] A;
        output[2:0] Y;
        reg [2:0] Y;
        always @(A or Con)
        begin
            case(Con)
              0: Y=A;
              1: Y=A<<1;
              2: Y=A>>1;
              default: Y=3’b0;
            endcase
         end
endmodule
```

Combinational logic shifters with shiftin and shiftout

Sel	Operation	Function
0	Y<=A, ShiftLeftOut=0 ShiftRightOut=0	No shift
1	Y<=shl(A), ShiftLeftOut=A[5] ShiftRightOut=0	Shift left
2	Y<=shr(A), ShiftLeftOut=0 ShiftRightOut=A[0]	Shift Right
3	Y<=0, ShiftLeftOut=0 ShiftRightOut=0	Zero Outputs

Verilog Code

```
always@(Sel or A or ShiftLeftIn or ShiftRightIn);
begin
  A_wide={ShiftLeftIn,A,ShiftRightIn};
 case(Sel)
    0: Y_wide=A_wide;
    1: Y_wide=A_wide<<1;
    2: Y_wide=A_wide>>1;
    3:Y_wide=5’b0;
    default: Y=A_wide;
 endcase
ShiftLeftOut=Y_wide[0];
Y=Y_wide[2:0];
ShiftRightOut=Y_wide[4];
end
```

Combinational 6 bit Barrel Shifter

Sel	Operation	Function
0	Y<=A	No shift
1	Y<-A rol 1	Rotate once
2	Y<-A rol 2	Rotate twice
3	Y<- A rol 3	Rotate Thrice
4	Y<-A rol 4	Rotate four times
5	Y<-A rol 5	Rotate five times

Verilog Coding

```
• function [2:0] rotate_left;
   input [5:0] A;
   input [2:0] NumberShifts;
   reg [5:0] Shifting;
   integer N;
   begin
      Shifting = A;
      for(N=1;N<=NumberShifts;N=N+1)
        begin
           Shifting={Shifting[4:0],Shifting[5])};
        end
      rotate_left=Shifting;
   end
  endfunction
```

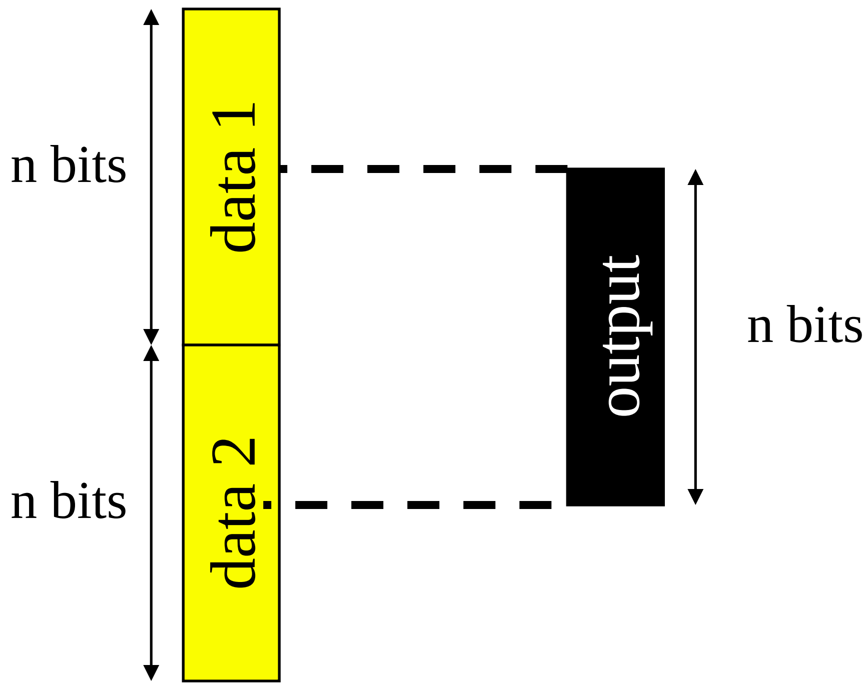
Verilog

- always @(Rotate or A)

```
begin
  case(Rotate)
    0: Y=A;
    1: Y=rotate_left(A,1);
    2: Y=rotate_left(A,2);
    3: Y=rotate_left(A,3);
    4: Y=rotate_left(A,4);
    5: Y=rotate_left(A,5);
    default: Y=6'bx;
   endcase
 end
```

Another Way

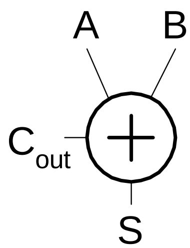
▪



Code is left as an exercise…

Single-Bit Addition

Half Adder

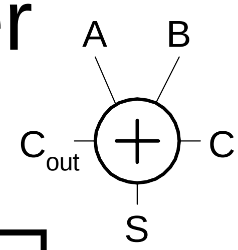


$S =$

$C_{out} =$

A	B	C_o	S
0	0		
0	1		
1	0		
1	1		

Full Adder



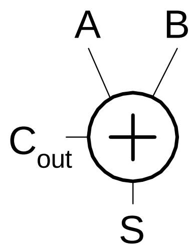
$S =$

$C_{out} =$

A	B	C	C_o	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		

Single-Bit Addition

Half Adder

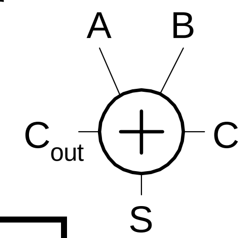


$$S = A \oplus B$$

$$C_{out} = A \bullet B$$

A	B	C_o	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Full Adder



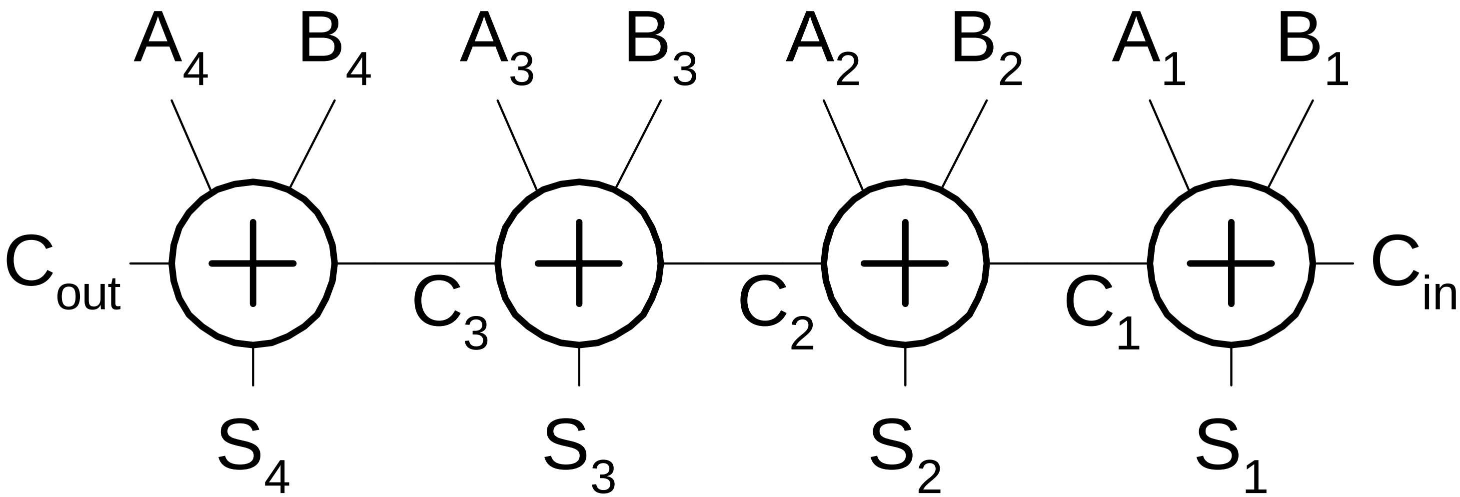
$$S = A \oplus B \oplus C$$

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

A	B	C	C_o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Carry-Ripple Adder

- Simplest design: cascade full adders
  - Critical path goes from Cin to Cout
  - Design full adder to have fast carry delay



Full adder

- Computes one-bit sum, carry:
  - $s_i = a_i \text{ XOR } b_i \text{ XOR } c_i$
  - $c_{i+1} = a_i b_i + a_i c_i + b_i c_i$
- Half adder computes two-bit sum.
- Ripple-carry adder: n-bit adder built from full adders.
- Delay of ripple-carry adder goes through all carry bits.

Verilog for full adder

```
module fulladd(a,b,carryin,sum,carryout);
   input a, b, carryin; /* add these bits*/
   output sum, carryout; /* results */

   assign {carryout, sum} = a + b + carryin;
                /*  compute the sum and carry */
   endmodule
```

Verilog for ripple-carry adder

```
module nbitfulladd(a,b,carryin,sum,carryout)
    input [7:0] a, b; /* add these bits */
    input carryin; /* carry in*/
    output [7:0] sum; /* result */
    output carryout;
    wire [7:1] carry; /* transfers the carry between bits */

    fulladd a0(a[0],b[0],carryin,sum[0],carry[1]);
    fulladd a1(a[1],b[1],carry[1],sum[1],carry[2]);
…
    fulladd a7(a[7],b[7],carry[7],sum[7],carryout]);
endmodule
```

Generate and Propagate

$$G[i] = A[i].B[i]$$
$$P[i] = A[i] \oplus B[i]$$
$$C[i] = G[i] + P[i].C[i-1]$$
$$S[i] = P[i] \oplus C[i-1]$$

$$G[i] = A[i].B[i]$$
$$P[i] = A[i] + B[i]$$
$$C[i] = G[i] + P[i].C[i-1]$$
$$S[i] = A[i] \oplus B[i] \oplus C[i-1]$$

Two methods to develop C[i] and S[i].

Both are correct

- Because, A[i]=1 and B[i]=1 (which may lead to a difference is taken care of by the term A[i]B[i])
- How do we make an n bit adder?
- The delay of the adder chain needs to be optimized.

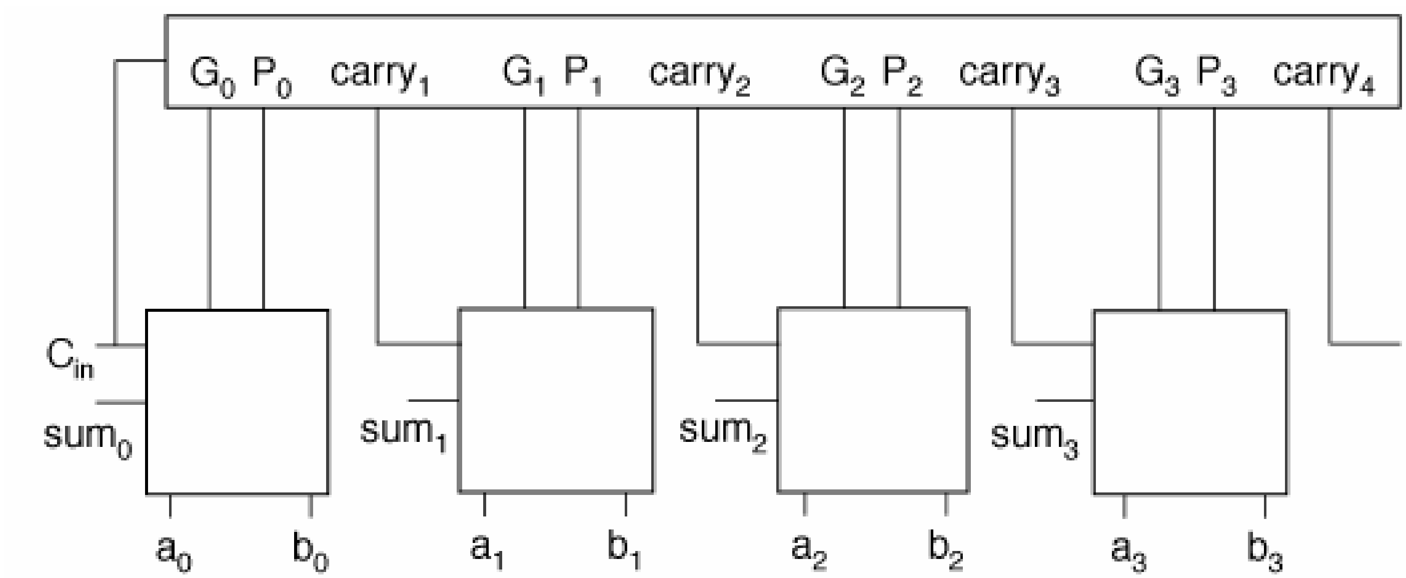
Carry-lookahead adder

- First compute carry propagate, generate:
  - $P_i = a_i + b_i$
  - $G_i = a_i b_i$
- Compute sum and carry from P and G:
  - $s_i = c_i \text{ XOR } P_i \text{ XOR } G_i$
  - $c_{i+1} = G_i + P_i c_i$

Carry-lookahead expansion

- Can recursively expand carry formula:
  - $c_{i+1} = G_i + P_i(G_{i-1} + P_{i-1}c_{i-1})$
  - $c_{i+1} = G_i + P_iG_{i-1} + P_iP_{i-1}(G_{i-2} + P_{i-1}c_{i-2})$
- Expanded formula does not depend on intermerdiate carries.
- Allows carry for each bit to be computed independently.

Depth-4 carry-lookahead



Analysis

- As we look ahead further logic becomes complicated.
- Takes longer to compute
- Becomes less regular.
- There is no similarity of logic structure in each cell.
- We have developed CLA adders, like Brent-Kung adder.

Verilog for carry-lookahead carry block

```
module carry_block(a,b,carryin,carry);
    input [3:0] a, b; /* add these bits*/
    input carryin; /* carry into the block */
    output [3:0] carry; /* carries for each bit in the block */
    wire [3:0] g, p; /* generate and propagate */

    assign g[0] = a[0] & b[0]; /* generate 0 */
    assign p[0] = a[0] ^ b[0]; /* propagate 0 */
    assign g[1] = a[1] & b[1]; /* generate 1 */
    assign p[1] = a[1] ^ b[1]; /* propagate 1 */
…
    assign carry[0] = g[0] | (p[0] & carryin);
    assign carry[1] = g[1] | p[1] & (g[0] | (p[0] & carryin));
    assign carry[2] = g[2] | p[2] &
            (g[1] | p[1] & (g[0] | (p[0] & carryin)));
    assign carry[3] = g[3] | p[3] &
            (g[2] | p[2] & (g[1] | p[1] & (g[0] | (p[0] & carryin))));
```

- endmodule

ci+1 = Gi + Pi(Gi-1 + Pi-1ci-1)

Verilog for carry-lookahead sum unit

```
module sum(a,b,carryin,result);
   input a, b, carryin; /* add these bits*/
   output result; /* sum */

   assign result = a ^ b ^ carryin;
                /*  compute the sum  */
endmodule
```

Verilog for carry-lookahead adder

```
•    module carry_lookahead_adder(a,b,carryin,sum,carryout);
     input [15:0] a, b; /* add these together */
     input carryin;
     output [15:0] sum; /* result */
     output carryout;
     wire [16:1] carry; /* intermediate carries */

     assign carryout = carry[16]; /* for simplicity */
     /* build the carry-lookahead units */
     carry_block b0(a[3:0],b[3:0],carryin,carry[4:1]);
     carry_block b1(a[7:4],b[7:4],carry[4],carry[8:5]);
     carry_block b2(a[11:8],b[11:8],carry[8],carry[12:9]);
     carry_block b3(a[15:12],b[15:12],carry[12],carry[16:13]);
     /* build the sum */
     sum  a0(a[0],b[0],carryin,sum[0]);
     sum  a1(a[1],b[1],carry[1],sum[1]);
…
     sum a15(a[15],b[15],carry[15],sum[15]);
endmodule
```

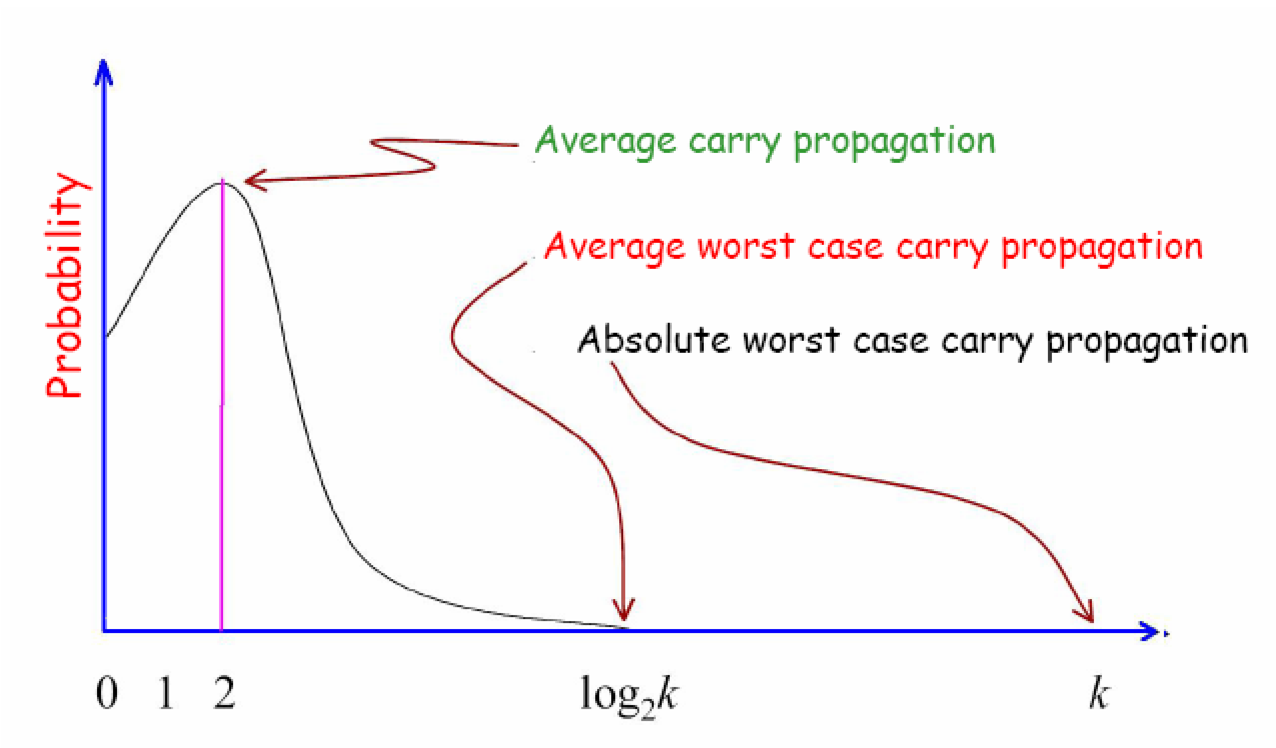
Dealing with the problem of carry propagation

1. Reduce the carry propagation time.

2. To detect the completion of the carry propagation time.

We have seen some ways to do the former.
How do we do the second one?

Motivation



Probability
Average carry propagation
Average worst case carry propagation
Absolute worst case carry propagation
0 1 2
$\log_2 k$
k

Carry Completion Sensing

A=**0** 0 1 1 **1** 0 **1** 1 **0** 1 1 0 1 **1** **0** **1**

B=**0** 1 0 0 **1** 1 **1** 0 **0** 0 0 1 0 **1** **0** **1**

\-

←　　**4**　　←　**1**　　←　　**5**　　←　**1**

```
Bit #  15  14  13  12   11  10   9   8    7   6   5   4    3   2   1   0
       --------------   --------------    --------------   --------------
        1   0   1   1    0   1   1   0    0   1   1   0    1   1   1   0
cout    0   1   0   1    1   0   0   1    1   1   0   0    0   0   1   1   cin
     \_______________/ \_______________________________/   \___________/ \_____/
             4                         6                         3          2
```

Can we compute the average length of carry chain?

- What is the probability that a chain generated at position i terminates at j?
  - It terminates if both the inputs A[j] and B[j] are zero or 1.
  - From i+1 to j-1 the carry has to propagate.
  - $p=(1/2)^{j-I}$
  - So, what is the expected length?
  - Define a random variable L, which denotes the length of the chain.

Expected length

- The chain can terminate at j=i+1 to j=k (the MSB position of the adder)
- Thus L=j-i for a choice of j.
- Thus expected length is: **approximately 2!**

$$\sum_{j=i+1}^{k}(j-i)2^{-(j-i)} = \sum_{j=i+1}^{k}(j-i)2^{-(j-i)} + (k-i)2^{-(k-1-i)}$$

$$= \sum_{l=1}^{k-1-i} l2^{-l} + (k-i)2^{-(k-1-i)} = 2-(k-i+1)2^{-(k-1-i)} + (k-i)2^{-(k-1-i)}$$

$$= 2-2^{-(k-1-i)}$$

$$[\text{Using, } \sum_{l=1}^{p} l2^{-l} = 2-(p+2)2^{-p}]$$

Carry completion sensing adder

A=011101101101101

B=100111000010101

C=000000000000000

N=000000000000000

C=000101000000101

N=000000010000010

A=011101101101101

B=100111000010101

C=000101000000101

N=000000010000010

C=001111000001101

N=000000110000010

Carry completion sensing adder

A=011101101101101

B=100111000010101

C=001111000001101

N=000000110000010

C=011111000011101

N=000000110000010

A=011101101101101

B=100111000010101

C=011111000011101

N=000000110000010

C=111111000111101

N=000000110000010

Carry completion sensing adder

A=0111011011101101

B=1001110000010101

C=1111110000111101

N=0000000110000010

C=1111110011111101

N=0000000110000010

Carry completion sensing adder

- (A[i],B[i])=(0,0)=>(Ci,Ni)=(0,1)
- (A[i],B[i])=(1,1)=>(Ci,Ni)=(1,0)
- (A[i],B[i])=(0,1)=>(Ci,Ni)=(Ci-1,Ni-1)
- (A[i],B[i])=(0,0)=>(Ci,Ni)=(Ci-1,Ni-1)
- Stop, when for all I, Ci V Ni = 1

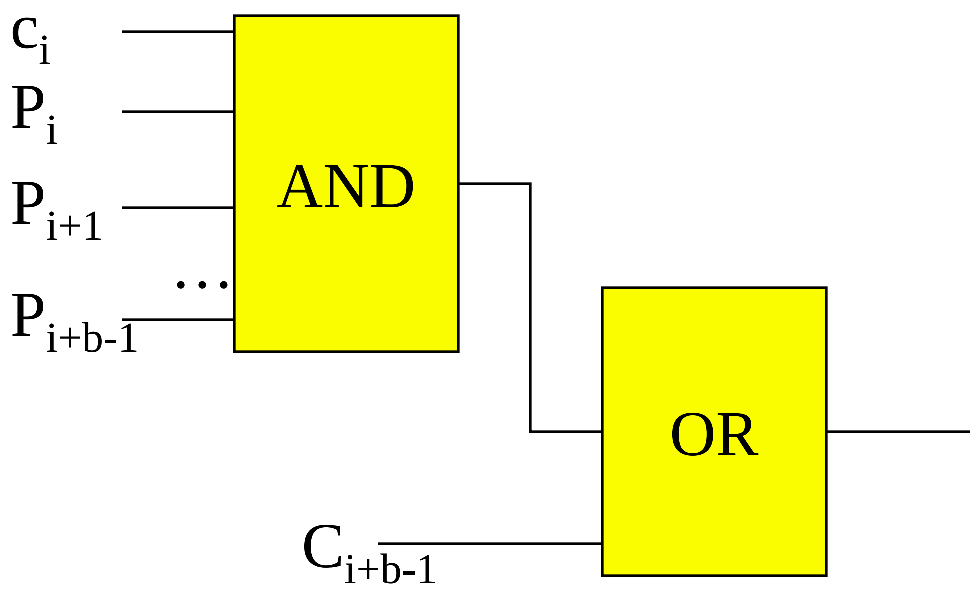
Justification

- Ci and Ni together is a coding for the carry.
- When Ci=1, carry can be computed. Make Ni=0
- When Ci=0 is the final carry, then indicate by Ni=1
- The carry can be surely stated when both Ai and Bi are 1’s or 0’s.

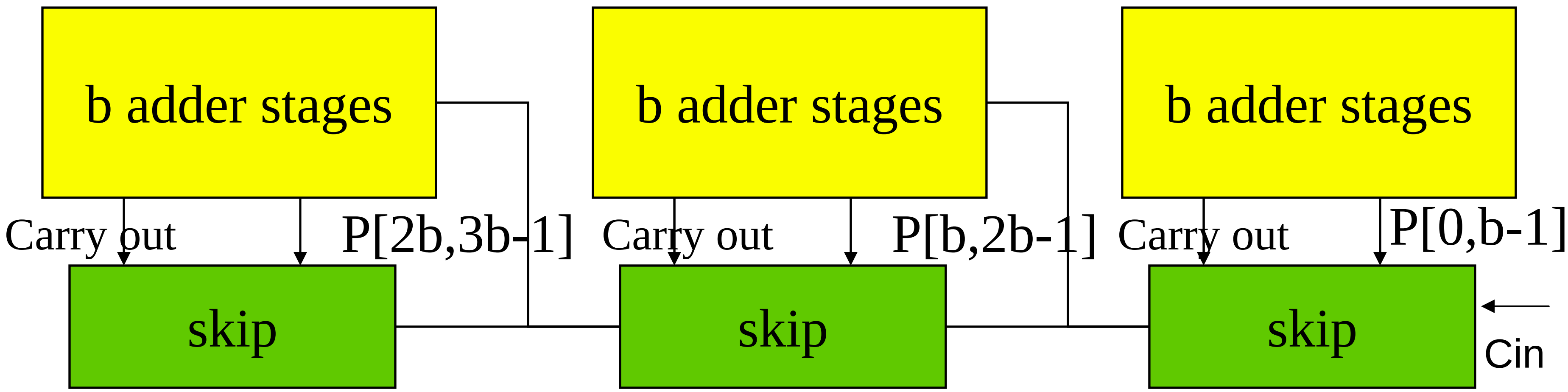
Carry-skip adder

- Looks for cases in which carry out of a set of bits is identical to carry in.
- Typically organized into *b*-bit stages.
- Can bypass carry through all stages in a group when all propagates are true: P_i P_{i+1} … P_{i+b-1}.
  – Carry out of group when carry out of last bit in group or carry is bypassed.

Carry-skip structure

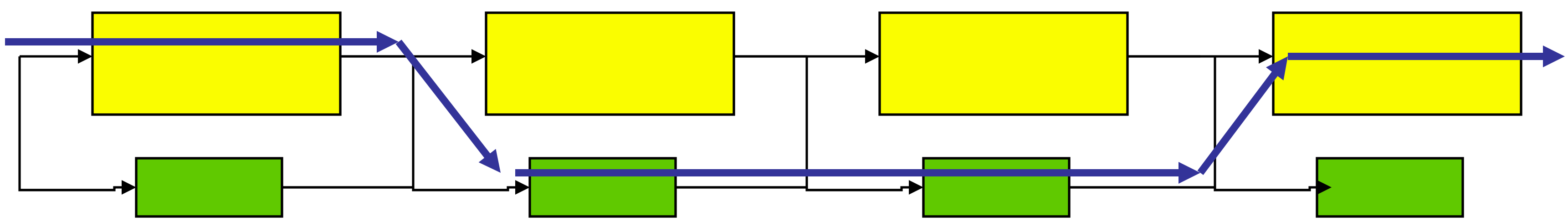


Carry-skip structure



Worst-case carry-skip

- Worst-case carry-propagation path goes through first, last stages:



Verilog for carry-skip add with P

```
module fulladd_p(a,b,carryin,sum,carryout,p);
  input a, b, carryin; /* add these bits*/
  output sum, carryout, p; /* results including
  propagate */

  assign {carryout, sum} = a + b + carryin;
                 /*  compute the sum and carry */
   assign p = a ^ b;
endmodule
```

Want to use ripple carry adder for the blocks

```
module fulladd_p(a,b,carryin,sum,carryout,p);
  input a, b, carryin; /* add these bits*/
  output sum, carryout, p; /* results including
  propagate */
   $rtl_binding="ADD3_RPL";
  assign {carryout, sum} = a + b + carryin;
                /*  compute the sum and carry */
    assign p = a ^ b;
endmodule
```

Directive to a synthesis tool!

Verilog for carry-skip adder

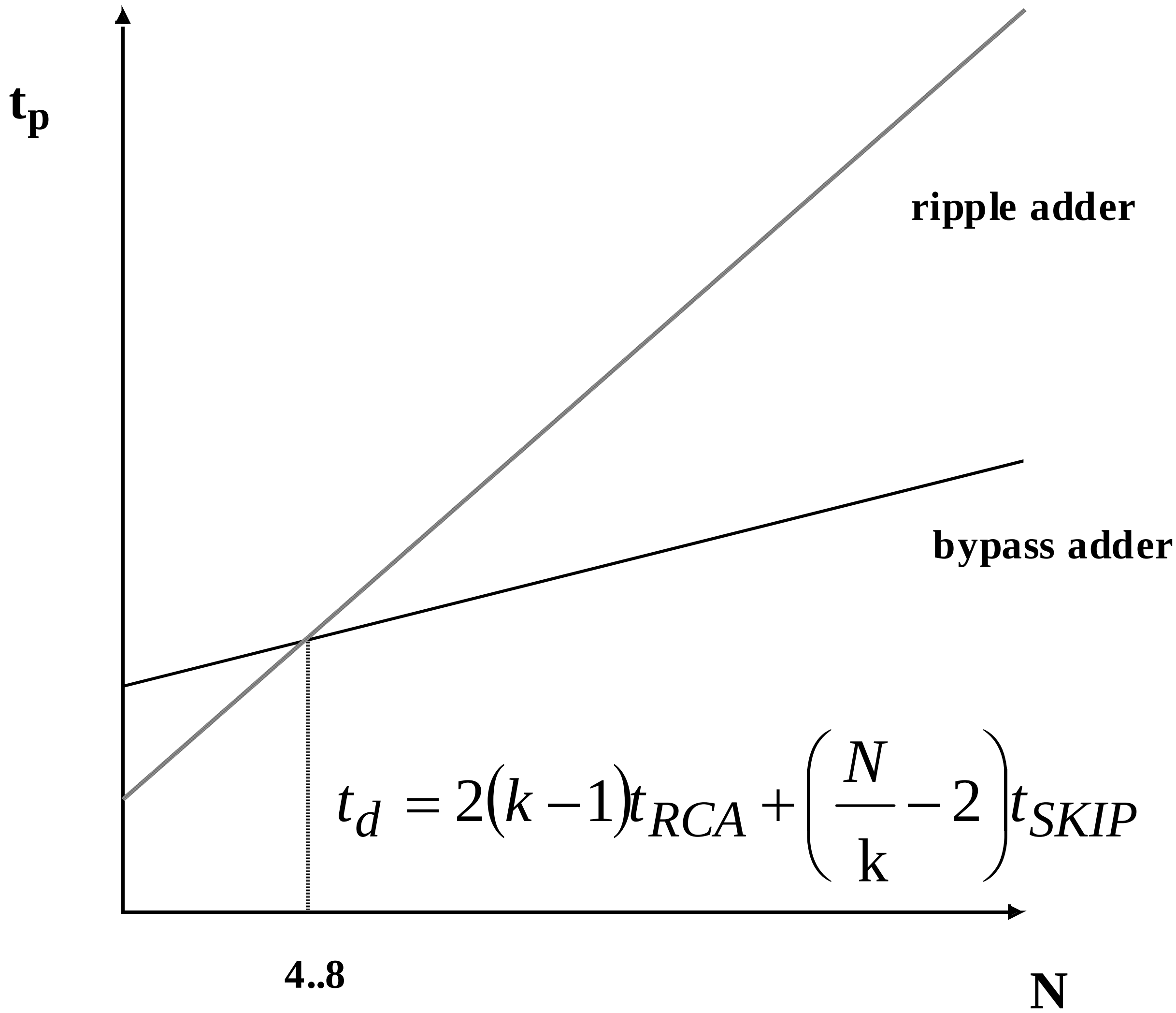
```
module carryskip(a,b,carryin,sum,carryout);
	input [7:0] a, b; /* add these bits */
	input carryin; /* carry in*/
	output [7:0] sum; /* result */
	output carryout;
	wire [8:1] carry; /* transfers the carry between bits */
	wire [7:0] p; /* propagate for each bit */
	wire cs4; /* final carry for first group */

	fulladd_p a0(a[0],b[0],carryin,sum[0],carry[1],p[0]);
	fulladd_p a1(a[1],b[1],carry[1],sum[1],carry[2],p[1]);
	fulladd_p a2(a[2],b[2],carry[2],sum[2],carry[3],p[2]);
	fulladd_p a3(a[3],b[3],carry[3],sum[3],carry[4],p[3]);
	assign cs4 = carry[4] | (p[0] & p[1] & p[2] & p[3] & carryin);
	fulladd_p a4(a[4],b[4],cs4,     sum[4],carry[5],p[4]);
…
	assign carryout = carry[8] | (p[4] & p[5] & p[6] & p[7] & cs4);
endmodule
```

Delay analysis

- Assume that skip delay = 1 bit carry delay.
- Delay of k-bit adder with block size b:
  - T = (b-1) + 0.5 + (k/b –2) + (b-1)

    block 0 OR gate skips   last block
- For equal sized blocks, optimal block size is sqrt(k/2).

Delay of Carry-Skip Adder

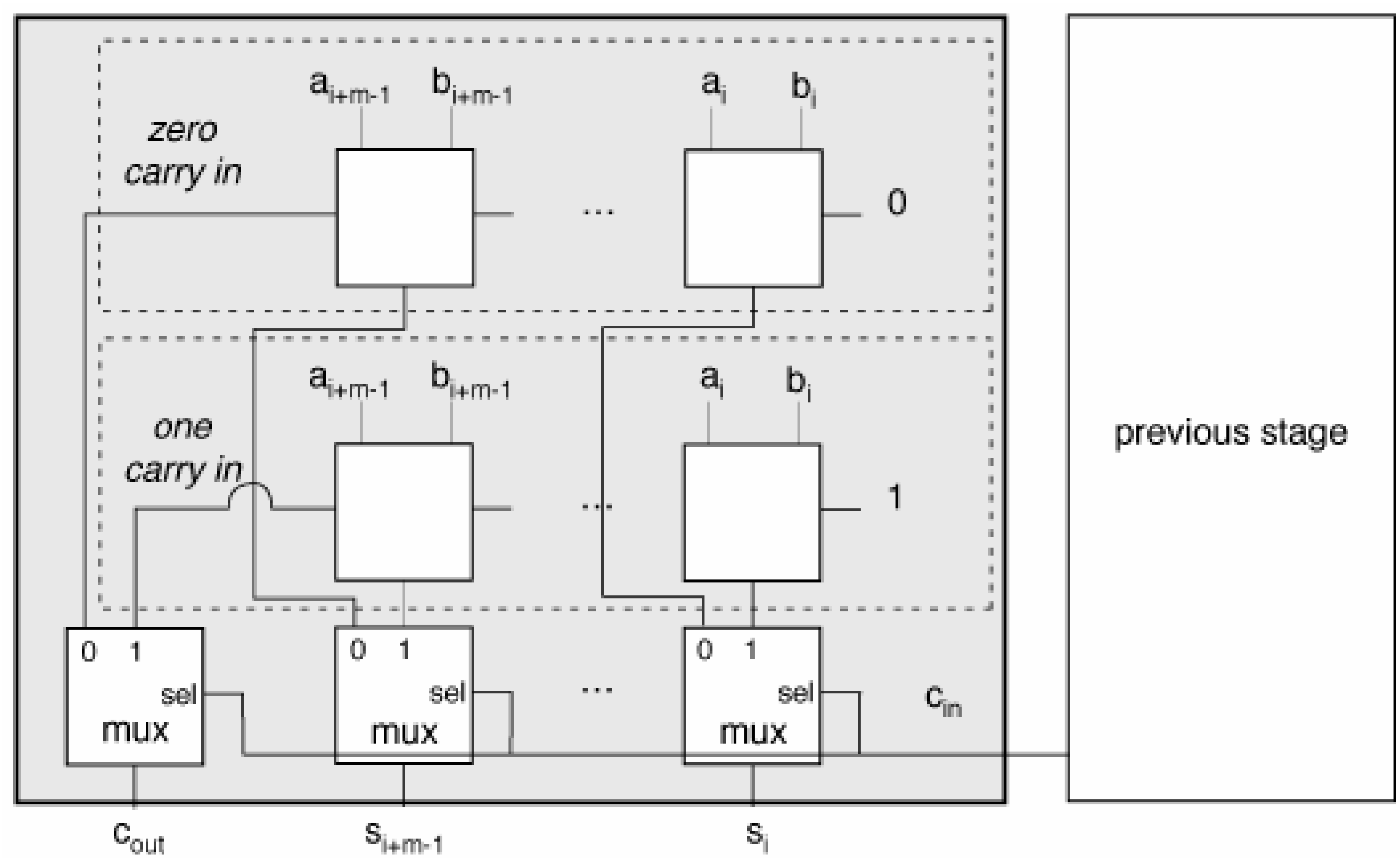


$$t_d = 2(k-1)t_{RCA} + \left(\frac{N}{k} - 2\right)t_{SKIP}$$

Carry-select adder

- Computes two results in parallel, each for different carry input assumptions.
- Uses actual carry in to select correct result.
- Reduces delay to multiplexer.

Carry-select structure



Carry-save adder

- Useful in multiplication.
- Input: 3 n-bit operands.
- Output: n-bit partial sum, n-bit carry.
  - Use carry propagate adder for final sum.
- Operations:
  - $s = (x + y + z) \bmod 2$.
  - $c = [(x + y + z) - 2] / 2$.

Adder comparison

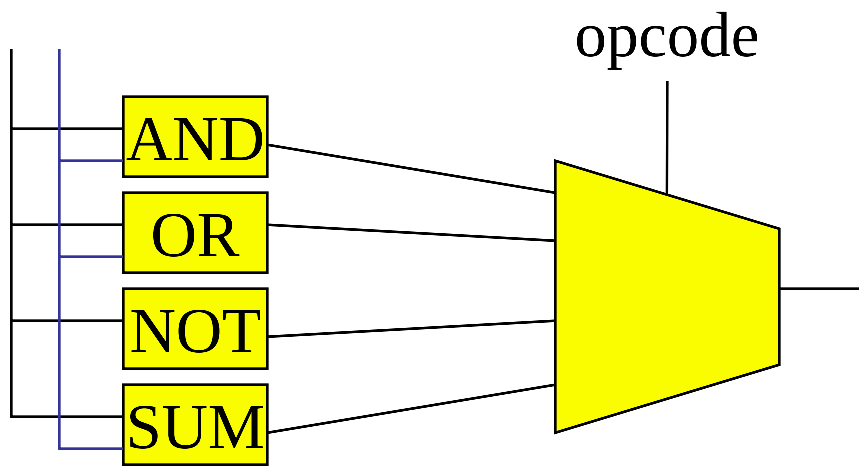
- Ripple-carry adder has highest performance/cost.
- Optimized adders are most effective in very long bit widths (> 48 bits).

ALUs

- ALU computes a variety of logical and arithmetic functions based on **opcode**.
- May offer complete set of functions of two variables or a subset.
- ALU built around adder, since carry chain determines delay.

ALU as multiplexer

- Compute functions then select desired one:



Implementation of the ALU