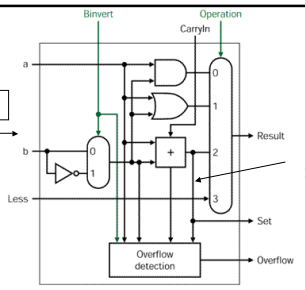
Overflow Logic

MSB logic block

Note that A, B are sign bits of operand.



Sign bit of result for addition, subtraction. Call this 'Nf'

Overflow logic depends on whether doing an addition or subtraction:

if (addition) overflow = (A and B and (not Nf)) or
((not A) and (not B) and Nf)

i.e. For addition, if sign bits of operands are the same, but the result sign bit is different, then OVERFLOW has occurred

Overflow Logic (cont).

What about subtraction? (A – B)

If operands have different sign bits, and the result sign bit is DIFFERENT from the sign of the A operand, then overflow!

If (subtraction) OF = (A and (not B) and (not Nf)) or
((not A) and B and Nf)

Recall that for subtraction, binvert = 1.

VHDL code:

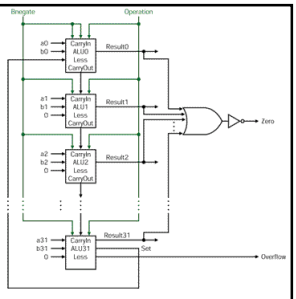
```
if (binvert = '0') then
   OF <= (A and B and (not Nf ) ) or ( ( not A) and (not B) and Nf) ;
else
   OF <= (A and (not B) and (not Nf) ) or ((not A) and B and Nf) ;
end if;
```

Set Logic

Support operation
slt rX, rA, rB

rX = '1' if rA < rB

rX = '0' if rA >= rB



From previous slides, note that 'Set' of MSB slice is simply the sign of the result.
If we do a 'SLT' operation, we want Bnegate = 1 so that ALU does A- B operation. 'Set' of MSB slice becomes LSB output value.

Set Logic (continued).

For ‘slt’ operation, ALU does A- B.

According to slides, the LSB of the result will be EQUAL to the SIGN bit of the A-B result, other bits will be zero.

Does this work?

A = 0x7F, B = 0x01. A-B = 0x7F – 0x01 = 0x7D.

Sign of result = ‘0’, result of SLT = 0x00 (A is NOT less than B)

Assume an 8-bit ALU:

A = 0x80, B= 0xFF. A-B = 0x80 – 0xFF = 0x81
Sign of result = ‘1’, result of SLT = 0x01.
?? (A= -128, B = -1, so A is less than B, so result correct!!!)

Set Logic (cont.)

Another Example:

A = 0x7F, B = 0xFF. Comparing +127 to -1.

A is NOT LESS than B, so result should be 0x00.

Lets see:

A-B = 0x7F - 0xFF = 0x80. Sign bit = 1, result = 0x01!!!!!

WRONG!!!

Overflow occurred – this means that just using the sign bit by itself is not good enough.

We have to consider the overflow flag.

Set Logic (cont.)

Set Logic modified to include Overflow flag is (exercise 4.23).

Let Nf = sign bit of result

SET = ((not OF) and Nf) or (OF and (not Nf))

If A < B, then result of A-B should be Negative if overflow did not occur.

However, if Overflow did occur, the Negative flag will be ‘0’ in the case of A < B.

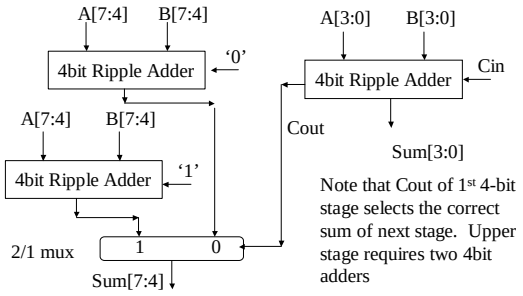
What about SLTU? Set Less than UNSIGNED?

Need to use the Carry Flag in some manner to determine the 'Set' bit (you figure it out).

We also need another control line to tell us if the operation is an UNSIGNED operation or SIGNED operation since the logic for the 'set' bit will be different.

Carry-Select Adder

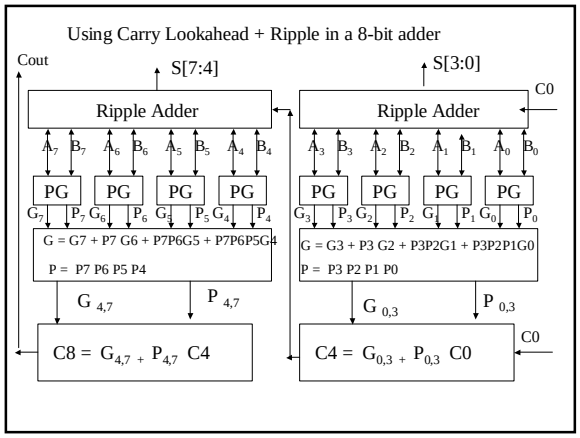
The Carry path is the slowest path in the ripple carry adder. We can speed it up with the following scheme (8-bit adder):

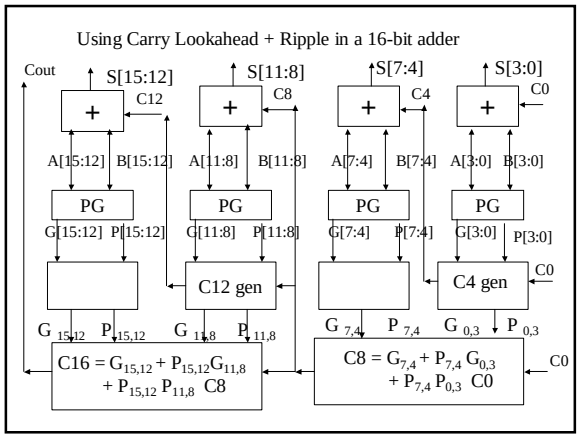


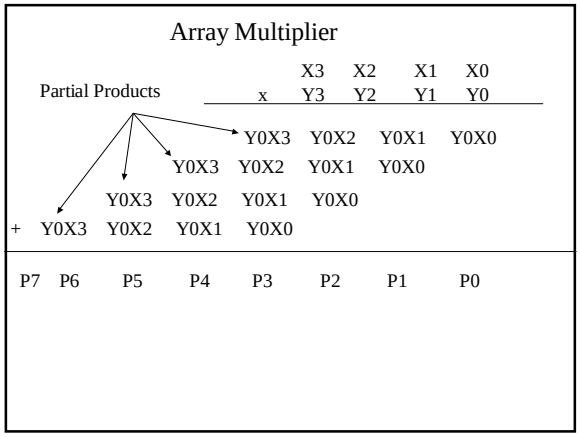
Note that Cout of 1st 4-bit stage selects the correct sum of next stage. Upper stage requires two 4bit adders

Speeding Up Addition

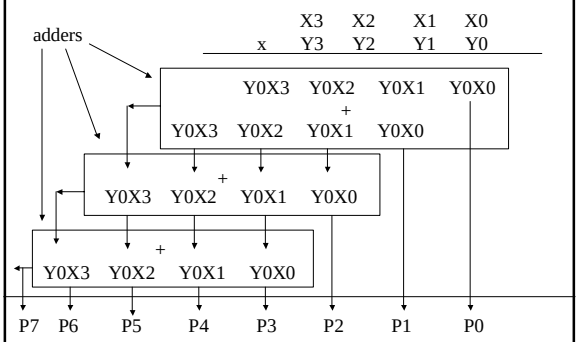
- Ripple Carry adder – slowest, but least gates
- Carry Select Adder
  - ⇒Breaks addition into groups of bits
  - ⇒Each group consists of two ripple carry adders – one has the CIN = 0, the other has CIN = 1
  - ⇒A 2/1 MUX that uses the Cout from the previous group is used to select the correct sum.
  - ⇒Speedup comes from carry not having to ripple through all groups.
  - ⇒Requires about 2x number of gates over Ripple Carry
- Carry LookAhead Adder
  - ⇒Expensive in terms of number of gates
  - ⇒Very fast since carries are generated without rippling.

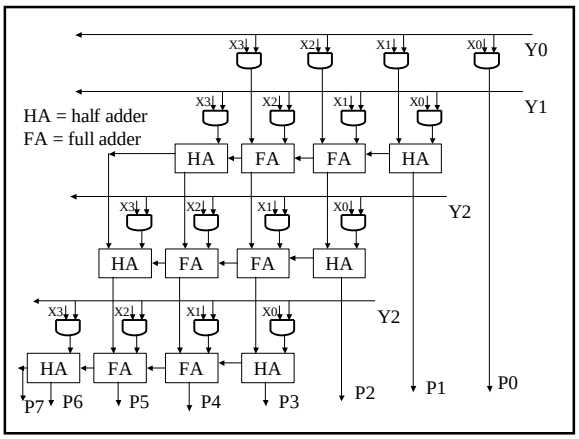

Using Carry Lookahead + Ripple in a 8-bit adder
Cout
S[7:4]
S[3:0]
C0
Ripple Adder
Ripple Adder
PG
G = G7 + P7 G6 + P7P6G5 + P7P6P5G4
P = P7 P6 P5 P4
G = G3 + P3 G2 + P3P2G1 + P3P2P1G0
P = P3 P2 P1 P0
C8 = G4,7 + P4,7 C4
C4 = G0,3 + P0,3 C0


Using Carry Lookahead + Ripple in a 16-bit adder
Cout
S[15:12]
S[11:8]
S[7:4]
S[3:0]
C12
C8
C4
C0
PG
C12 gen
C4 gen
C16 = G15,12 + P15,12 G11,8 + P15,12 P11,8 C8
C8 = G7,4 + P7,4 G0,3 + P7,4 P0,3 C0


Array Multiplier
Partial Products
X3 X2 X1 X0
x Y3 Y2 Y1 Y0
Y0X3 Y0X2 Y0X1 Y0X0
Y0X3 Y0X2 Y0X1 Y0X0
Y0X3 Y0X2 Y0X1 Y0X0
+ Y0X3 Y0X2 Y0X1 Y0X0
P7 P6 P5 P4 P3 P2 P1 P0

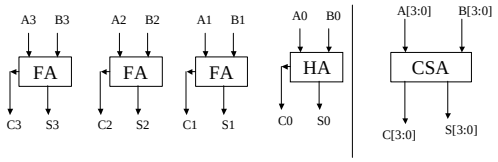
Array Multiplier

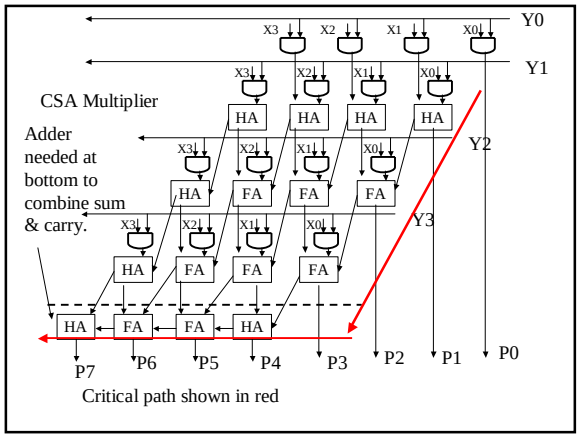




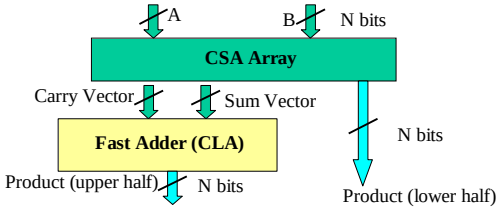
Array Multiplier – Can we do better?

- One problem with the array multiplier on the previous slide is that there are multiple critical paths
  ⇒ It is tough to speed up because there are multiple 'longest' paths – we would have to speedup all of these paths.
- A better design uses Carry-Save-Adders (CSA)
  ⇒Carry and Sum brought out separately, carry does not ripple





Why is the CSA Multiplier Better?



Using a fast adder to combine Sum + Carry will speed up multiplier!!!

Many other schemes are used for speeding up multiplication, most involve variations on using CSAs.