## Decimal Multiplication

- Remember decimal multiplication from 3rd grade?

```
   43    // multiplicand
*  12    // multiplier
   86
+ 430
  516    // product
```

- Start with running total 0, repeat steps until no multiplier digits
  - Multiply multiplicand by least significant multiplier digit
  - Add to total
  - Shift multiplicand one digit to the left (multiply by 10)
  - Shift multiplier one digit to the right (divide by 10)
- Product of N-digit and M-digit numbers potentially has N+M digits



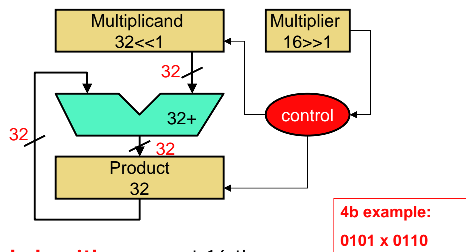
## Binary Multiplication

```
   43 = 00000101011    // multiplicand
*  12 = 00000001100    // multiplier
    0 = 00000000000
    0 = 00000000000
  172 = 00010101100
+ 344 = 00101011000
  516 = 01000000100    // product
```

- Same thing except …
  – There are more individual steps (smaller base)
  + But each step is simpler
  - Multiply multiplicand by **least significant multiplier bit**
    - 0 or 1 → no actual multiplication, just add multiplicand or not
  - Add to total: we know how to do that
  - Shift multiplicand left, multiplier right by one bit: **shift registers**



## Simple 16x16=32bit Multiplier Circuit

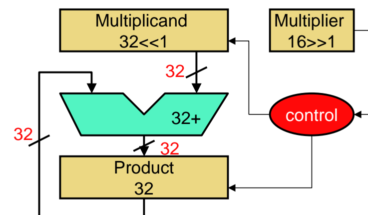



4b example:
0101 x 0110

- **Control algorithm**: repeat 16 times
  - If LSB(multiplier) == 1, then add multiplicand to product
  - Shift multiplicand left by 1
  - Shift multiplier right by 1

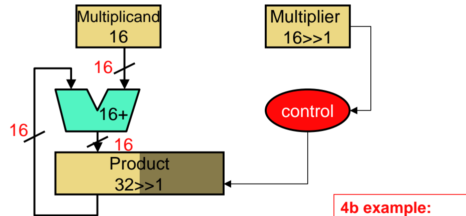


## Inefficiencies with Simple Circuit




- Notice
  – 32-bit addition, but 16 multiplicand bits are always 0
    - And 0 bits are always moving
  - Solution? Instead of shifting multiplicand left, shift product right

## Better 16-bit Multiplier

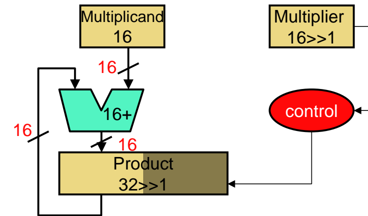



**4b example:**
**0101 x 0110**

- **Control algorithm**: repeat 16 times
  - LSB(multiplier) == 1 ? Add multiplicand to upper half of product
  - Shift multiplier right by 1
  - Shift product right by 1

© 2008 Daniel J. Sorin
from Roth and Lebeck

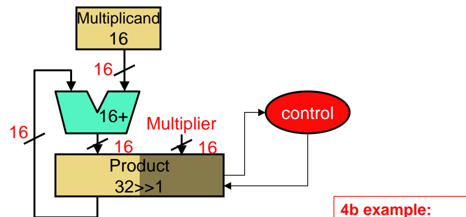

## Another Inefficiency




- Notice one more inefficiency
  - What is initially the lower half of product gets thrown out
  - As useless lower half of product is shifted right, so is multiplier
  - Solution: use lower half of product as multiplier

© 2008 Daniel J. Sorin
from Roth and Lebeck


## Even Better 16-bit Multiplier




**4b example:**
**0101 x 0110**

- **Control algorithm**: repeat 16 times
  - LSB(multiplier) == 1 ? Add multiplicand to upper half of product
  - Shift product right by 1

© 2008 Daniel J. Sorin
from Roth and Lebeck


## Multiplying Negative Numbers

- Just works…
  - As long as right shifts are arithmetic and not logical
  - Try it out for yourself now
- 0101 * 1111 = ?

© 2008 Daniel J. Sorin
from Roth and Lebeck

## Booth's Algorithm

- Notice the following equality (Booth did)
  - $2^J + 2^{J-1} + 2^{J-2} + \dots + 2^K = 2^{J+1} - 2^K$
  - Example: 0111 = 1000 - 0001
  - We can exploit this to create a faster multiplier
- How?
  - Sequence of N 1s in the multiplier yields sequence of N additions
  - Replace with one addition and one subtraction



## Booth In Action

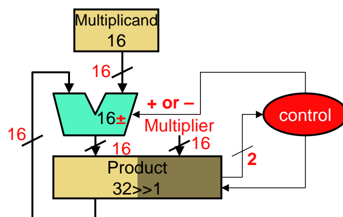
- For each multiplier bit, also examine bit to its right
  - **00**: middle of a run of 0s, do nothing
  - **10**: beginning of a run of 1s, subtract multiplicand
  - **11**: middle of a run of 1s, do nothing
  - **01**: end of a run of 1s, add multiplicand

```
    43 = 00000101011
*   12 = 00000001100
     0 = 00000000000    // multiplier bits 0_ (implicit 0)
+    0 = 00000000000    // multiplier bits 00
-  172 = 11101010100    // multiplier bits 10
+    0 = 00000000000    // multiplier bits 11
+  688 = 01010110000    // multiplier bits 01
   516 = 01000000100
```

**ICQ: so why is Booth better?**



## Booth Hardware




- **Control algorithm**: repeat 16 times
  - Multiplier LSBs == 10? Subtract multiplicand from product
  - Multiplier LSBs == 01? Add multiplicand to product
  - Shift product/multiplier right by 1 (not by 2!)



## Booth in Summary

- Performance/efficiency
  - \+ Good for sequences of 3 or more 1s
    - Replaces 3 (or more) adds with 1 add and 1 subtract
  - Doesn't matter for sequences of 2 1s
    - Replaces 2 adds with 1 add and 1 subtract (add = subtract)
  - – Actually bad for singleton 1s
    - Replaces 1 add with 1 add and 1 subtract
- Bottom line
  - Worst case multiplier (101010) requires N/2 adds + N/2 subs
    - What is the worst case multiplier for straight multiplication?
  - How is this better than normal multiplication?

## Modified Booth's Algorithm

- What if we detect singleton 1s and do the right thing?
- Examine multiplier bits in groups of 2s plus a helper bit on the right (as opposed to 1 bit plus helper bit on right)
  - Means we'll need to shift product/multiplier by 2 (not 1)

- **000**: middle of run of 0s, do nothing
- **100**: beginning of run of 1s, subtract multiplicand<<1 (M*2)
  - Why M*2 instead of M?
- **010**: singleton 1, add multiplicand
- **110**: beginning of run of 1s, subtract multiplicand
- **001**: end of run of 1s, add multiplicand
- **101**: end of run of 1s, beginning of another, subtract multiplicand
  - Why is this? $-2^{J+1} + 2^J = -2^J$
- **011**: end of a run of 1s, add multiplicand<<1 (M*2)
- **111**: middle of run of 1s, do nothing

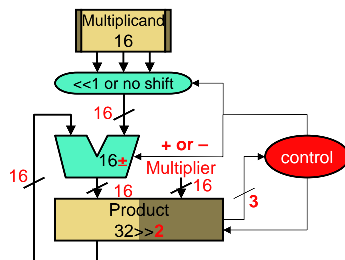


## Modified Booth In Action

```
   43 = 00000101011
*  12 = 00000001100
    0 = 00000000000    // multiplier bits 000
- 172 = 11101010100    // multiplier bits 110
+ 688 = 01010110000    // multiplier bits 001
  516 = 01000000100
```



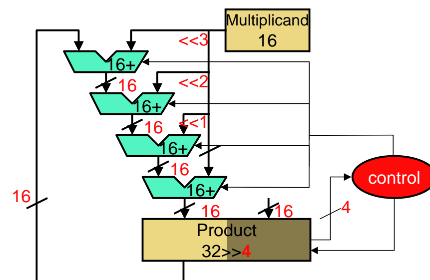
## Modified Booth Hardware




- **Control algorithm**: repeat 8 times
  - Based on 3b groups, add/subtract shifted/unshifted multiplicand
  - Shift product/multiplier right by **2**



## Another Multiplier: Multiple Adders



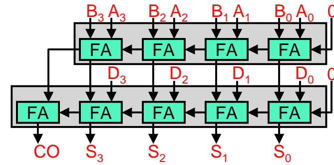

- Can multiply by N bits at a time by using N adders
  - Doesn't help: 4X fewer iterations, each one 4X longer (4*9=**36**)

## Carry Save Addition (CSA)

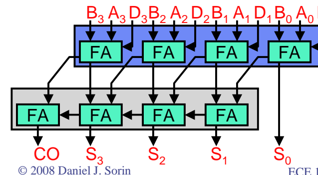
- **Carry save addition (CSA):** d(N adds) < N*d(1 add)
  - Enabling observation: unconventional view of full adder
    - 3 inputs ($A,B,C_{in}$) → 2 outputs ($S,C_{out}$)
  - If adding two numbers, only thing to do is chain $C_{out}$ to $C_{in+1}$
    - But what if we are adding three numbers (A+B+D)?
  - One option: back-to-back conventional adders
    - ($A,B,C_{inT}$) → ($T,C_{outT}$), chain $C_{outT}$ to $C_{inT+1}$
    - ($T,D,C_{inS}$) → ($S,C_{outS}$), chain $C_{outS}$ to $C_{inS+1}$
  - Notice: we have **three independent inputs** to feed first adder
    - ($A,B,D$) → ($T,C_{outT}$), **no chaining (CSA: 2 gate levels)**
    - ($T,C_{outT},C_{inS}$) → ($S,C_{outS}$), chain $C_{outS}$ to $C_{inS+1}$



## Carry Save Addition (CSA)



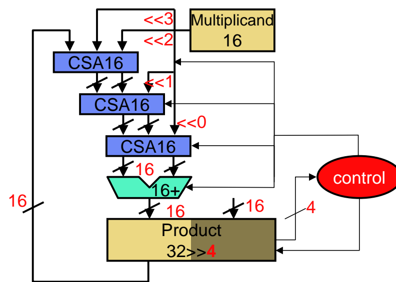

- 2 conventional adders
  - 2 * d(add) gate levels
  - d(add16)=9
  - → d = 18
- k conventional adders
  - k * d(add) gate levels




- CSA+conventional adder
  - 2 + d(add) gate levels
  - → d = 11
- k CSAs+conventional add
  - 2 + d(add) gate levels



## Carry Save Multiplier




- 4-bit at a time multiplier using 3 CSA + 1 normal adder
  - Actually helps: 4X fewer iterations, each only (2+2+2+9=**15**)