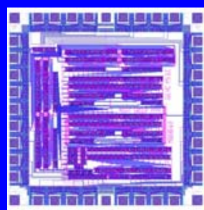
Computer Arithmetic

Multiplication & Shift

Chapter 3.4

EEC170 FQ 2005



Layout 8-bit Pipelined Multiplier

Multiply

- We will start with unsigned multiply and contrast how humans and computers multiply

How Humans Multiply

- We first generate all partial product terms

```
  1010   ← Multiplicand
x 1101   ← Multiplier
======

  1010   ← Partial Product
```


How Humans Multiply

- We first generate all partial product terms

```
  1010   ← Multiplicand
x 1101   ← Multiplier
======

  1010
 0000    ← Partial Product
```


How Humans Multiply

- We first generate all partial product terms

```
  1010   ← Multiplicand
x 1101   ← Multiplier
======

  1010
 0000
1010     ← Partial Product
```


How Humans Multiply

- We first generate all partial product terms

```
   1010   ← Multiplicand
 x 1101   ← Multiplier
 ======

   1010
  0000
 1010
1010      ← Partial Product
```

How Humans Multiply

- Then add column by column, right to left

```
   1010
 x 1101
 ======
   1010
  0000
 1010
1010
========
      0    ← Product
```


How Humans Multiply

- Then add column by column, right to left

```
   1010
 x 1101
 ======
   1010
  0000
 1010
1010
========
     10    ← Product
```


How Humans Multiply

- Then add column by column, right to left

```
   1010
 x 1101
 ======
   1010
  0000
 1010
1010
========
    010    ← Product
```


How Humans Multiply

- Sometimes with one or more carry digits

```
   1010
 x 1101
 ======
  1
   1010
  0000
 1010
1010
========
   0010    ← Product
```


How Humans Multiply

- Sometimes with one or more carry digits

```
   1010
 x 1101
 ======
 11
   1010
  0000
 1010
1010
========
  00010    ← Product
```


How Humans Multiply

- Sometimes with one or more carry digits

```
   1010
 x 1101
 ======
111
   1010
  0000
 1010
1010
========
 000010    ← Product
```

How Humans Multiply

- Sometimes with one or more carry digits

```
    1010
  x 1101
  ======
1111
    1010
   0000
  1010
 1010
========
 0000010     ← Product
```


How Humans Multiply

- Sometimes with one or more carry digits

```
    1010
  x 1101
  ======
1111
    1010
   0000
  1010
 1010
========
10000010     ← Product
```


How Humans Multiply

- Sometimes with one or more carry digits

```
    1010
  x 1101
  ======
1111
    1010
   0000
  1010
 1010
========
10000010     ← Product
```


Human Method Not Best for Computers

- Each partial product must be stored ⇨ extra hardware
- Columns vary in size ⇨ complexity
- Multiple-digit carries ⇨ complexity
- Need a simpler method for low-cost multipliers

Shift & Add Multiply

- Simplest computer multiply is to shift Multiplicand left one bit per iteration to generate partial product
- Each iteration if corresponding Multiplier bit is 1:
  - Product = Product + Multiplicand
- NxN bit multiply takes N iterations (N clock cycles)

Shift & Add Multiply

```
    1010     ← Multiplicand
  x 1101     ← Multiplier
  ======
00000000     ← Old Product
00001010     ← New Product
```

Shift & Add Multiply

```
    1010    ← Multiplicand
  x 1101    ← Multiplier
  ======
00001010    ← Old Product
00001010    ← New Product
```


Shift & Add Multiply

```
    1010    ← Multiplicand
  x 1101    ← Multiplier
  ======
00001010    ← Old Product
00110010    ← New Product
```


Shift & Add Multiply

```
 1010       ← Multiplicand
  x 1101    ← Multiplier
  ======
00110010    ← Old Product
10000010    ← New Product
```


Shift & Add Multiply

- Computer multiply also shifts multiplier right so current multiplier bit is at a fixed position, the least significant bit (LSB)

Shift & Add Multiply

```
    1010    ← Multiplicand
  x 1101    ← Multiplier
  ======
00000000    ← Old Product
00001010    ← New Product
```


Shift & Add Multiply

```
    1010    ← Multiplicand
  x  110    ← Multiplier
  ======
00001010    ← Old Product
00001010    ← New Product
```

Shift & Add Multiply

```
  1010    ← Multiplicand
x   11    ← Multiplier
======
00001010  ← Old Product
00110010  ← New Product
```

25

Shift & Add Multiply

```
  1010    ← Multiplicand
x    1    ← Multiplier
======
00110010  ← Old Product
10000010  ← New Product
```

26

Software Multiple

- Very simple processors don't have hardware, implement shift & add in software

Multiply in C:

```
int product = 0;
for(int i = 0; i<32; i++)
   if( ( multiplier>>i % 2 ) == 1)
       product = product + multiplicand<<i;
```

27

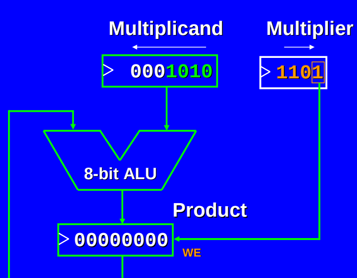
MIPS Assembly Multiply

```
# multiplicand is in $a0, need not save
# multiplier is in $a1, need not save
#
       move $v0,0          #$v0 is the product
Loop:  andi $t1,$a1,1      #get multiplier bit
       beq $t1,$0,Next     #test bit
       add $v0,$v0,$a0     #add partial product
Next:  sll $a0,$a0,1       #get next partial product
       slr $a1,$a1,1       #position multiplier bit
       bne $a1,$0,Loop     #got any bits left?
```

28

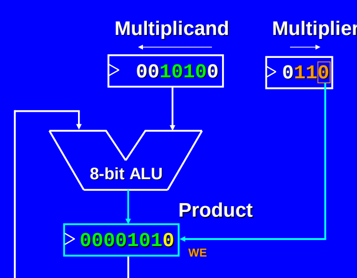
Simple Hardware Multiply

- Like shift & add we have already seen
  - Multiplier LSB is write enable for Product latch



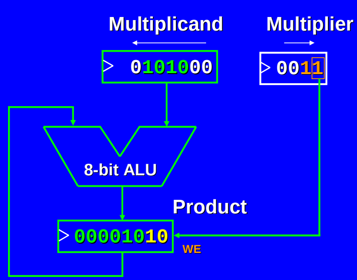
29

Simple Hardware Multiply

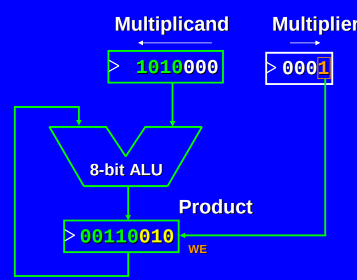


30

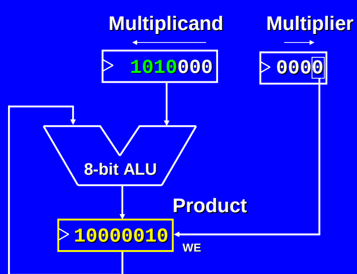
Simple Hardware Multiply



Simple Hardware Multiply



Simple Hardware Multiply



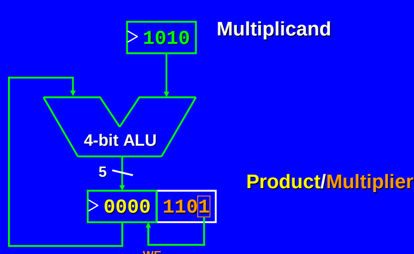
Refined Multiply Hardware

- Notice the following about the simple shift & add hardware:
  - Only N significant bits are being summed each cycle, but we are using a 2N-bit adder, a waste
  - Each cycle one new bit of the product is resolved, while one old bit of the multiplier is discarded
  - Simple multiply shifts Multiplicand left and keeps Product stationary. It is equivalent to keep Multiplicand stationary and shift Product right (same relative motion).

Refined Hardware Multiply

- ALU input is accept/not accept based on WE

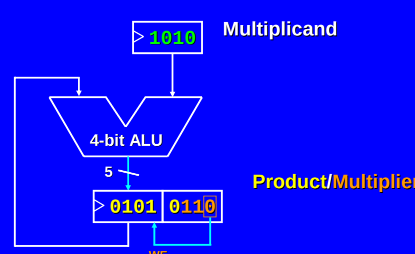
Start of 1st Iteration



Refined Hardware Multiply

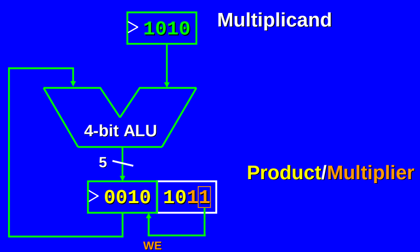
- When WE=0, shift but no ALU input

Start of 2nd Iteration



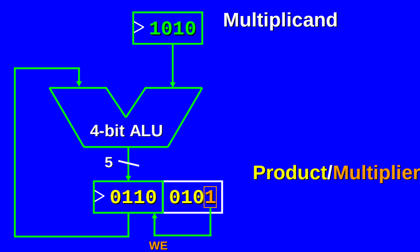
Refined Hardware Multiply

Start of 3rd Iteration



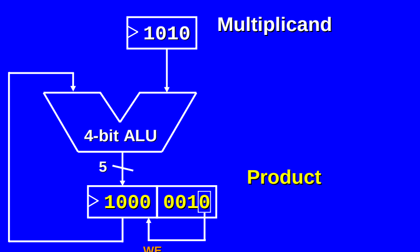
Refined Hardware Multiply

Start of 4th Iteration



Refined Hardware Multiply

Final Result: 10 x 13 = 130



Signed Multiplication

- Shift and Add only works for positive numbers
  To include negative numbers must:
  - Save XOR of sign bits to get product sign bit
  - Convert multiplier/multiplicand to positive
  - Do shift and add algorithm
  - Negate result if product sign bit is 1
- Booth's Algorithm handles positive /negative numbers uniformly
  - Based on observation that ...011...1110... = ...100...0000... - ...000...0010...

    E.g., 01110_2 (14_{10}) = 10000_2 (16_{10}) - 00010_2 (2_{10})
    = 10000_2 - 00010_2
  - I.e. convert string of 1s into leading +1 and a trialing -1

Booth's Algorithm

- Identify leading +1s and trailing –1s in Multiplier bit position i by looking at Multiplier bit i and bit i-1:

  i i-1
  - -1 for 1 0
  - +1 for 0 1
  - 0 for 0 0
  - 0 for 1 1
- Examples:

  1_{ten} = 0001 ⇨ 1

Booth's Algorithm

- Identify leading +1s and trailing –1s in Multiplier bit position i by looking at Multiplier bit i and bit i-1:

  i i-1
  - -1 for 1 0
  - +1 for 0 1
  - 0 for 0 0
  - 0 for 1 1
- Examples:

  1_{ten} = 0001 ⇨ 11

Booth's Algorithm

- Identify leading +1s and trailing –1s in Multiplier bit position i by looking at Multiplier bit i and bit i-1:

  i i-1

  - -1 for 1 0
  - +1 for 0 1
  - 0 for 0 0
  - 0 for 1 1
- Examples:

  1_{ten} = 0001 ⇨ 011

Booth's Algorithm

- Identify leading +1s and trailing –1s in Multiplier bit position i by looking at Multiplier bit i and bit i-1:

  i i-1

  - -1 for 1 0
  - +1 for 0 1
  - 0 for 0 0
  - 0 for 1 1
- Examples:

  1_{ten} = 0001 ⇨ 0011 = 2-1 = 1

Booth's Algorithm

- Identify leading +1s and trailing –1s in Multiplier bit position i by looking at Multiplier bit i and bit i-1:

  i i-1

  - -1 for 1 0
  - +1 for 0 1
  - 0 for 0 0
  - 0 for 1 1
- Examples:

  1_{ten} = 0001 ⇨ 0011 = 2-1 = 1

  -1_{ten} = 1111 ⇨ 1

Booth's Algorithm

- Identify leading +1s and trailing –1s in Multiplier bit position i by looking at Multiplier bit i and bit i-1:

  i i-1

  - -1 for 1 0
  - +1 for 0 1
  - 0 for 0 0
  - 0 for 1 1
- Examples:

  1_{ten} = 0001 ⇨ 0011 = 2-1 = 1

  -1_{ten} = 1111 ⇨ 01

Booth's Algorithm

- Identify leading +1s and trailing –1s in Multiplier bit position i by looking at Multiplier bit i and bit i-1:

  i i-1

  - -1 for 1 0
  - +1 for 0 1
  - 0 for 0 0
  - 0 for 1 1
- Examples:

  1_{ten} = 0001 ⇨ 0011 = 2-1 = 1

  -1_{ten} = 1111 ⇨ 001

Booth's Algorithm

- Identify leading +1s and trailing –1s in Multiplier bit position i by looking at Multiplier bit i and bit i-1:

  i i-1

  - -1 for 1 0
  - +1 for 0 1
  - 0 for 0 0
  - 0 for 1 1
- Examples:

  1_{ten} = 0001 ⇨ 0011 = 2-1 = 1

  -1_{ten} = 1111 ⇨ 0001 = -1

Booth's Algorithm

- Identify leading +1s and trailing –1s in Multiplier bit position i by looking at Multiplier bit i and bit i-1:

  i i-1
  - -1 for 1 0
  - +1 for 0 1
  - 0 for 0 0
  - 0 for 1 1
- Examples:

  1_{ten} = 0001 ⇨ 0011 = 2-1 = 1

  -1_{ten} = 1111 ⇨ 0001 = -1

  -6_{ten} = 1010 ⇨ 0

49

Booth's Algorithm

- Identify leading +1s and trailing –1s in Multiplier bit position i by looking at Multiplier bit i and bit i-1:

  i i-1
  - -1 for 1 0
  - +1 for 0 1
  - 0 for 0 0
  - 0 for 1 1
- Examples:

  1_{ten} = 0001 ⇨ 0011 = 2-1 = 1

  -1_{ten} = 1111 ⇨ 0001 = -1

  -6_{ten} = 1010 ⇨ 10

50

Booth's Algorithm

- Identify leading +1s and trailing –1s in Multiplier bit position i by looking at Multiplier bit i and bit i-1:

  i i-1
  - -1 for 1 0
  - +1 for 0 1
  - 0 for 0 0
  - 0 for 1 1
- Examples:

  1_{ten} = 0001 ⇨ 0011 = 2-1 = 1

  -1_{ten} = 1111 ⇨ 0001 = -1

  -6_{ten} = 1010 ⇨ 110

51

Booth's Algorithm

- Identify leading +1s and trailing –1s in Multiplier bit position i by looking at Multiplier bit i and bit i-1:

  i i-1
  - -1 for 1 0
  - +1 for 0 1
  - 0 for 0 0
  - 0 for 1 1
- Examples:

  1_{ten} = 0001 ⇨ 0011 = 2-1 = 1

  -1_{ten} = 1111 ⇨ 0001 = -1

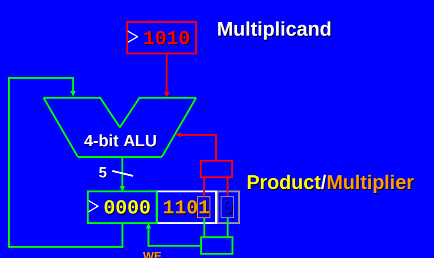
  -6_{ten} = 1010 ⇨ 1110 = -8+4-2= -6

52

Booth Hardware Implementation

- Use ALU to ADD or SUB based on the trailing –1s, leading 1s from the Multiplier

Start of 1^{st} iteration

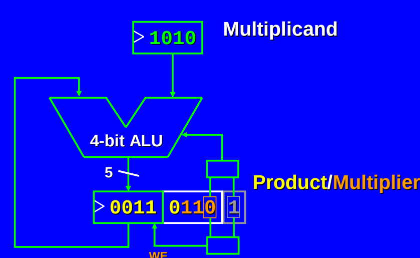


53

Booth Hardware Implementation

- Product shift is arithmetic shift, sign bit does not change

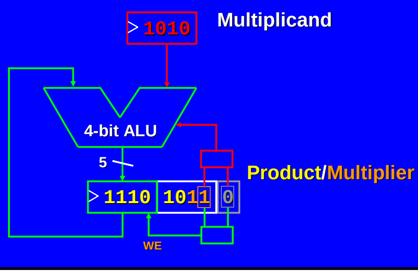
Start of 2^{nd} iteration



54

Booth Hardware Implementation

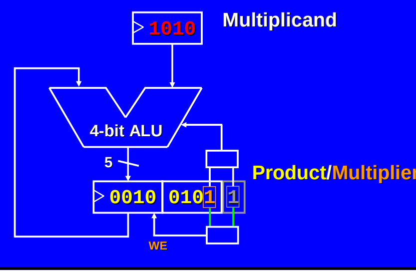
Start of 3rd iteration



55

Booth Hardware Implementation

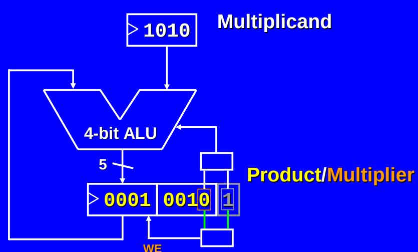
Start of 4th iteration



56

Booth Hardware Implementation

Final Result: -6 x –3 = 18



57

Fast Multiply

- Shift & Add is slow: 32x slower than addition
- Fast processors have fast multiply hardware using more hardware than shift & add
- Basic example is a Wallace tree adder, which uses an array of full adder cells

58

Wallace Tree Stage 0

- Partial products produced sequentially (slow)

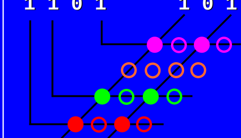
```
  1010   Multiplicand
x 1101   Multiplier
======

  1010
 0000
1010
1010
```

Partial Products

- Partial products produced using array of AND gates (fast)

Multiplier Multiplicand

1 1 0 1   1 0 1 0



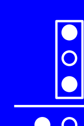
Partial Products

59

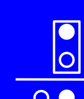
Wallace Tree Reducers

- Full adder cells used to reduce column segment of height 3 to row of width 2
  - sum and carry out
- Half adders used selectively for height-3 column that does not need full reduction

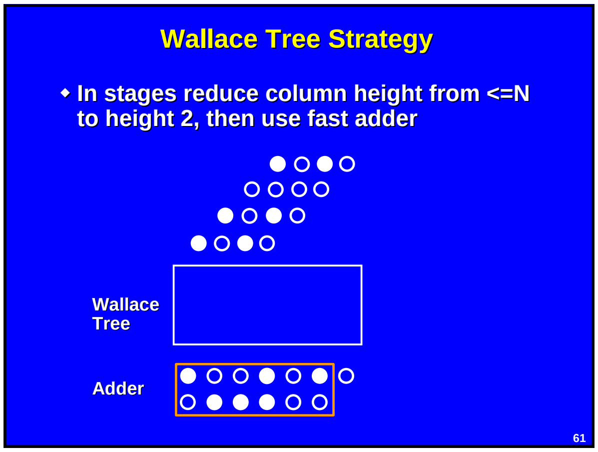
Full Adder

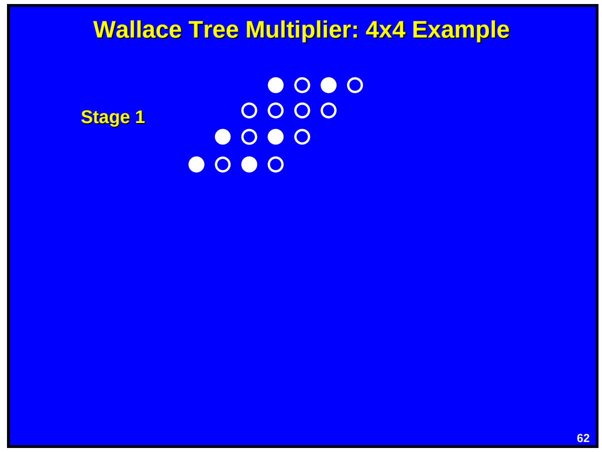


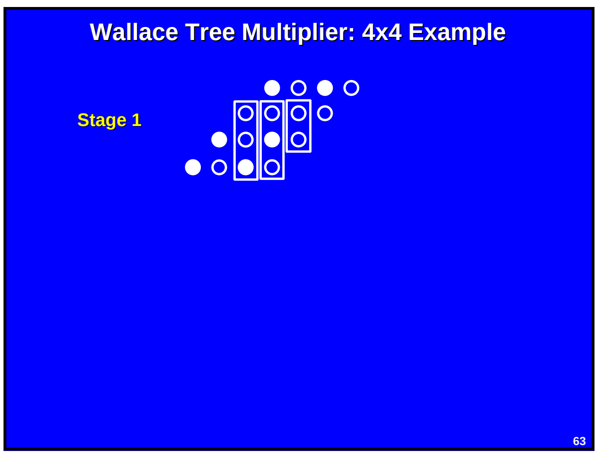
Half Adder

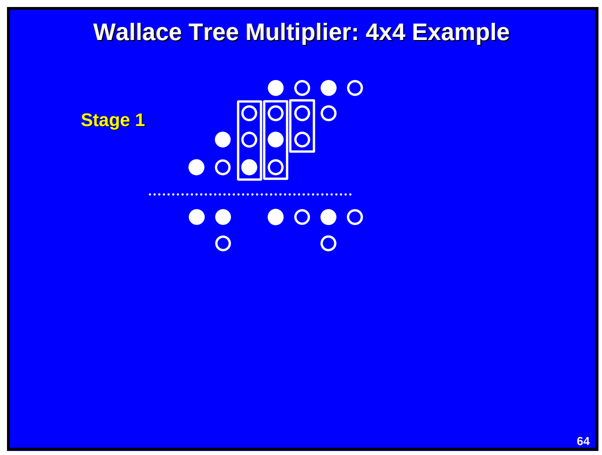


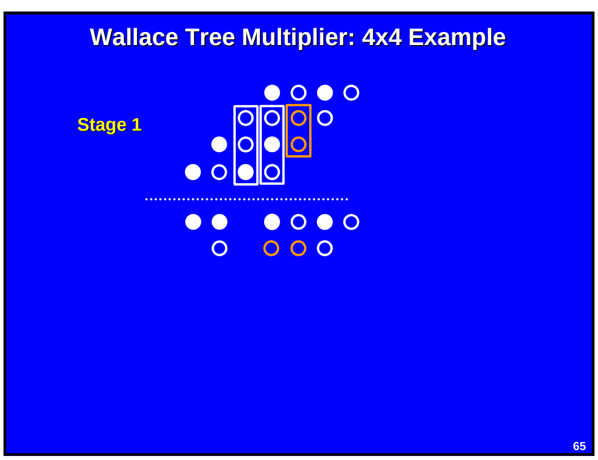
60

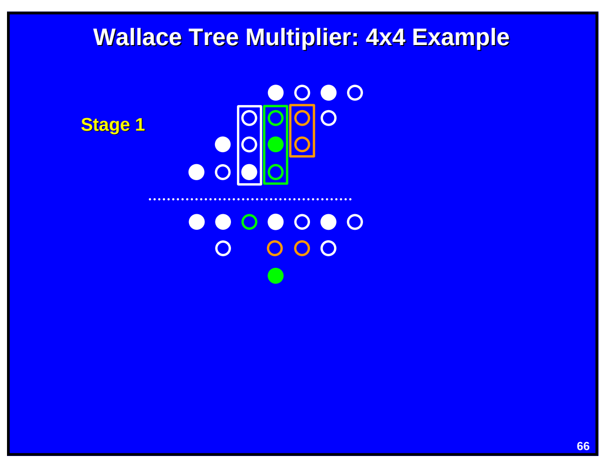

Wallace Tree Strategy
In stages reduce column height from <=N to height 2, then use fast adder
Wallace Tree
Adder
61

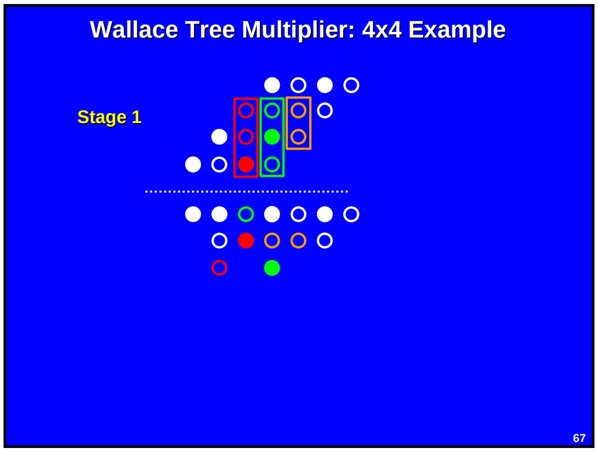

Wallace Tree Multiplier: 4x4 Example
Stage 1
62

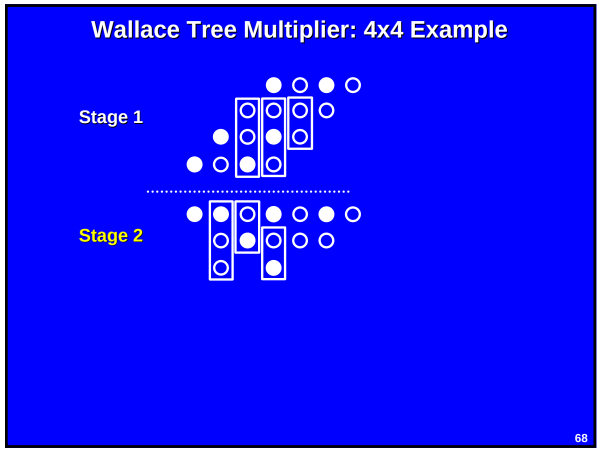

Wallace Tree Multiplier: 4x4 Example
Stage 1
63

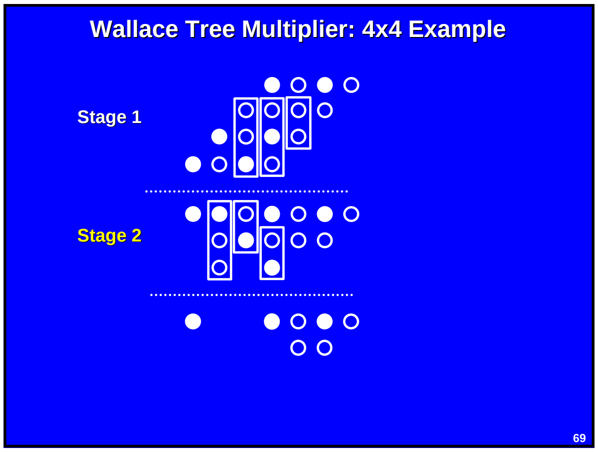

Wallace Tree Multiplier: 4x4 Example
Stage 1
64

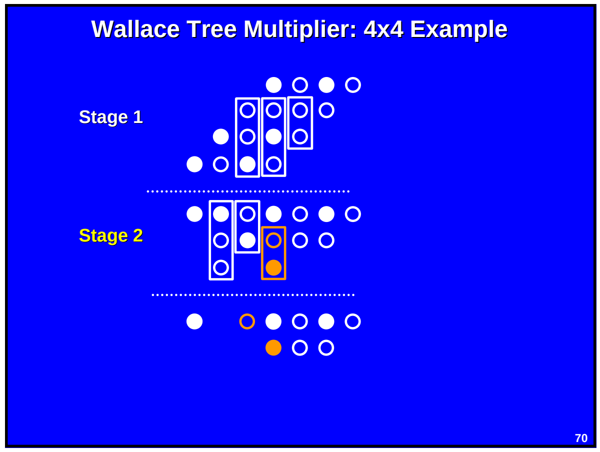

Wallace Tree Multiplier: 4x4 Example
Stage 1
65

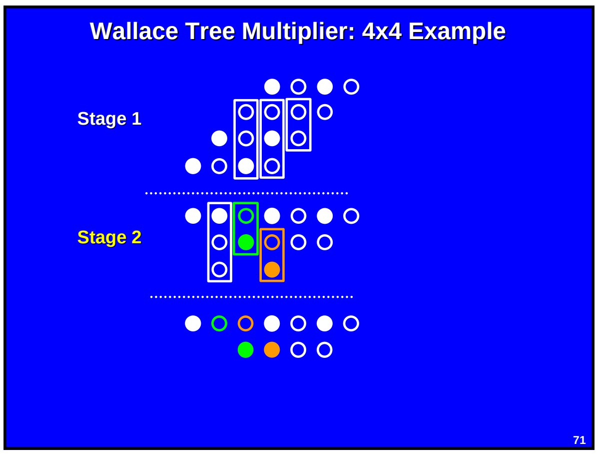

Wallace Tree Multiplier: 4x4 Example
Stage 1
66

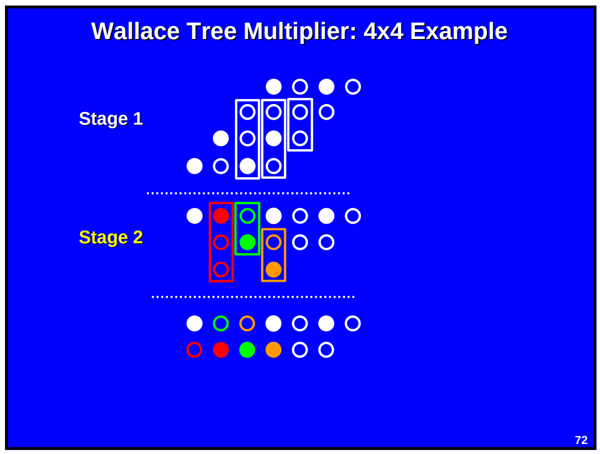

Wallace Tree Multiplier: 4x4 Example
Stage 1
67


Wallace Tree Multiplier: 4x4 Example
Stage 1
Stage 2
68

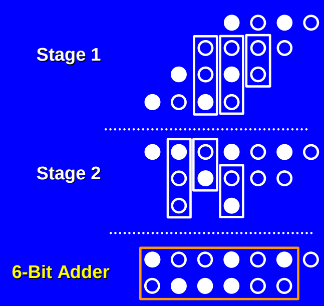

Wallace Tree Multiplier: 4x4 Example
Stage 1
Stage 2
69


Wallace Tree Multiplier: 4x4 Example
Stage 1
Stage 2
70

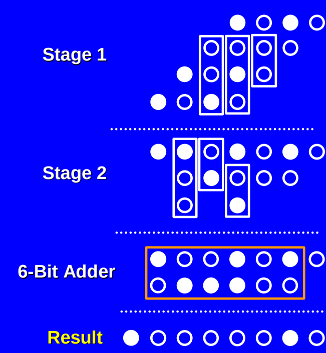

Wallace Tree Multiplier: 4x4 Example
Stage 1
Stage 2
71


Wallace Tree Multiplier: 4x4 Example
Stage 1
Stage 2
72

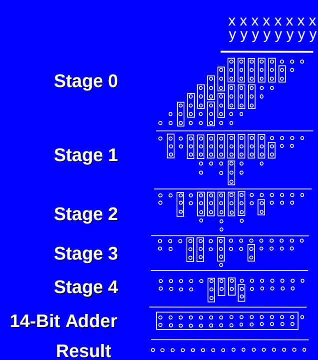
Wallace Tree Multiplier: 4x4 Example



Wallace Tree Multiplier: 4x4 Example



Wallace Tree Multiplier: 8x8 Example



Wallace Tree Mult (cont.)

- Column height is reduced by about 2/3 during each level
- For NxN multiply, an estimate for number of levels:

$N \times (2/3)^x = 2$

$(2/3)^x = 2/N$

$x \log_2(2/3) = \log_2(2/N)$

$x = \log_2(2/N) / \log_2(2/3)$

$x = [\log(2) - \log(N)] / -0.6$

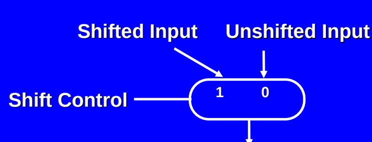
$x = -1.7 \times [1 - \log_2(N)]$

$x = 1.7 * \log_2(N) - 1.7$

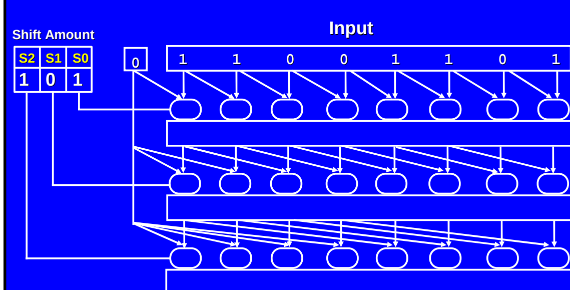
2 Levels for N=4, 4 Levels for N = 8, 7 levels for N = 32

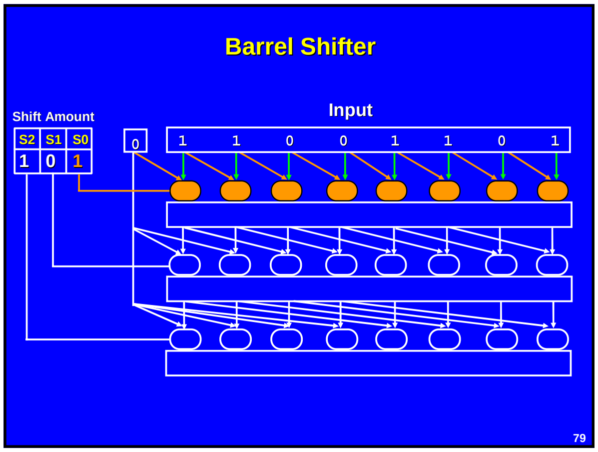
Barrel Shifter

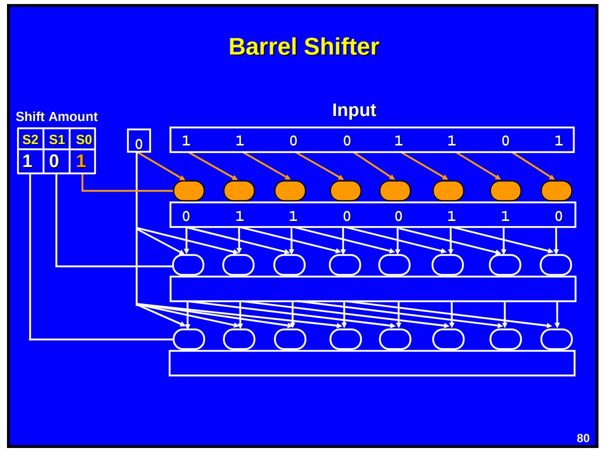
- Want shifter that is fast, not shift one bit position at a time
- Can be done in N-1=31 shifts in $\log_2 N$=5 stages using an array of N $\log_2 N$= 160 multiplexers

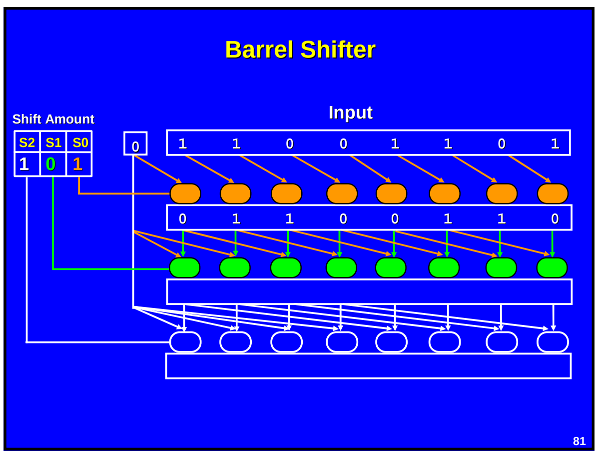


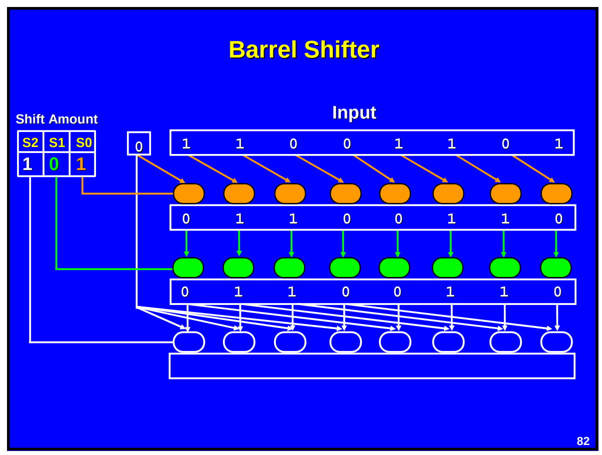
Barrel Shifter

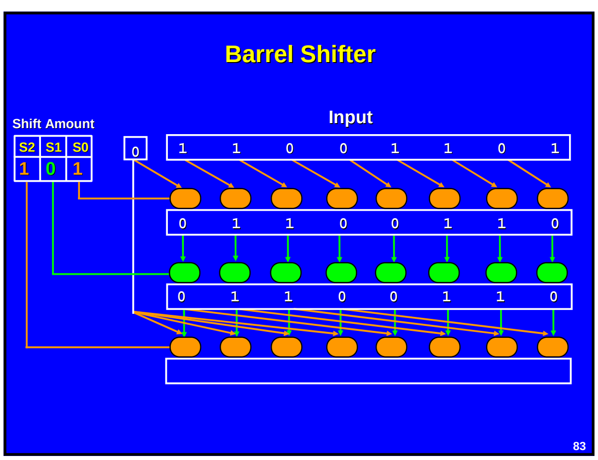


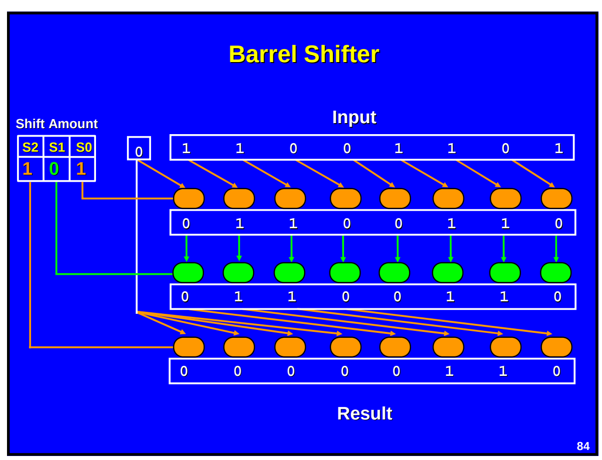

Barrel Shifter
Shift Amount
S2 S1 S0
1 0 1
Input
0
1 1 0 0 1 1 0 1
79


Barrel Shifter
Shift Amount
S2 S1 S0
1 0 1
Input
0
1 1 0 0 1 1 0 1
0 1 1 0 0 1 1 0
80


Barrel Shifter
Shift Amount
S2 S1 S0
1 0 1
Input
0
1 1 0 0 1 1 0 1
0 1 1 0 0 1 1 0
81


Barrel Shifter
Shift Amount
S2 S1 S0
1 0 1
Input
0
1 1 0 0 1 1 0 1
0 1 1 0 0 1 1 0
0 1 1 0 0 1 1 0
82


Barrel Shifter
Shift Amount
S2 S1 S0
1 0 1
Input
0
1 1 0 0 1 1 0 1
0 1 1 0 0 1 1 0
0 1 1 0 0 1 1 0
83


Barrel Shifter
Shift Amount
S2 S1 S0
1 0 1
Input
0
1 1 0 0 1 1 0 1
0 1 1 0 0 1 1 0
0 1 1 0 0 1 1 0
0 0 0 0 0 1 1 0
Result
84