Chapter 3

Arithmetic for Computers

Multiplication

- Start with long-multiplication approach

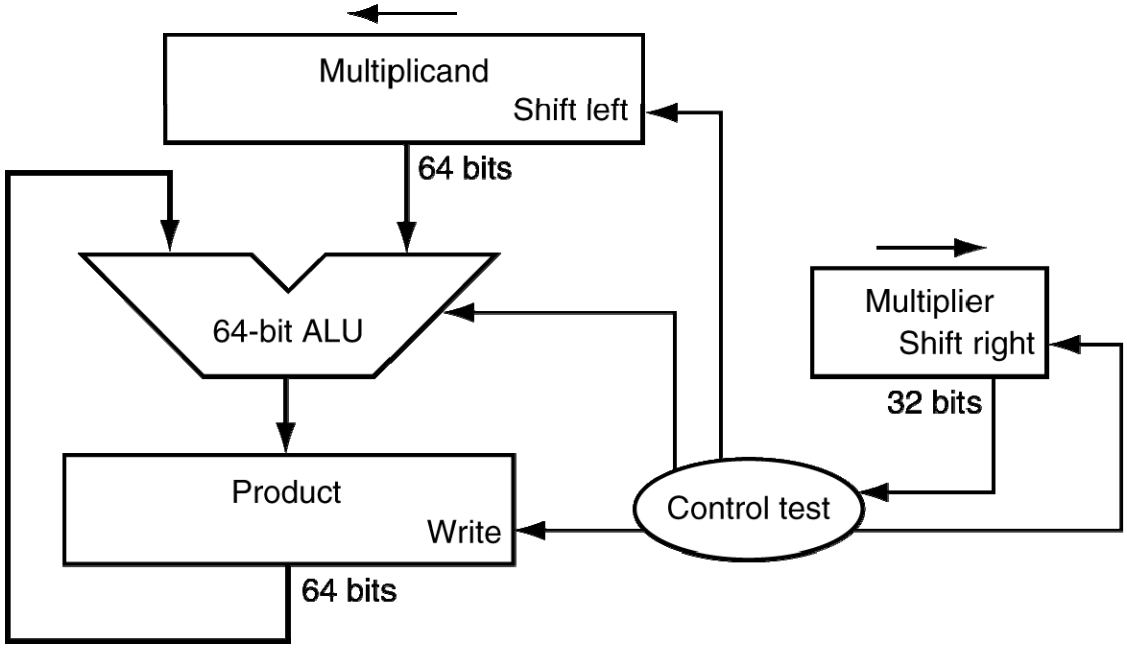
multiplicand

multiplier

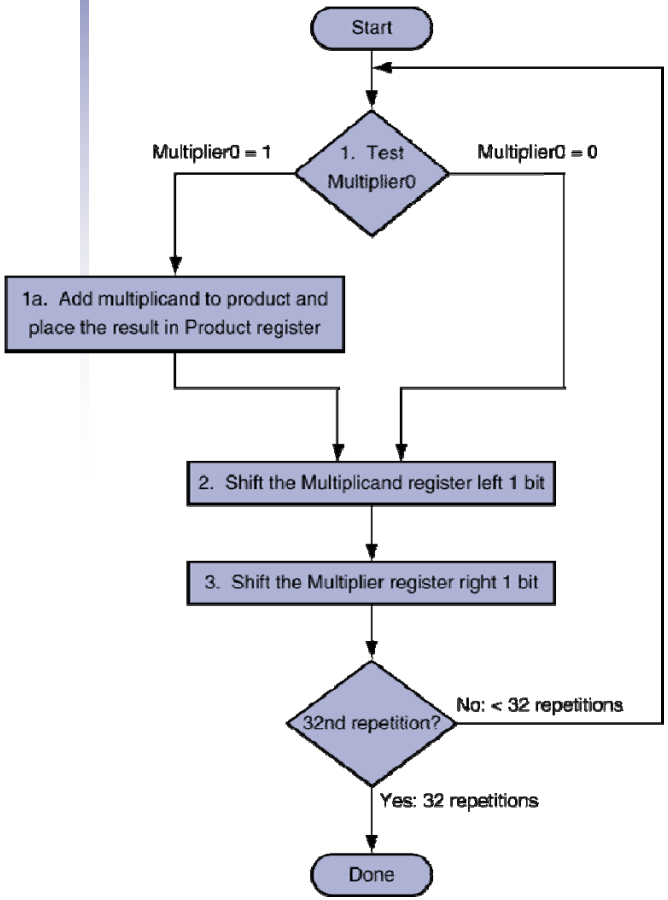
product

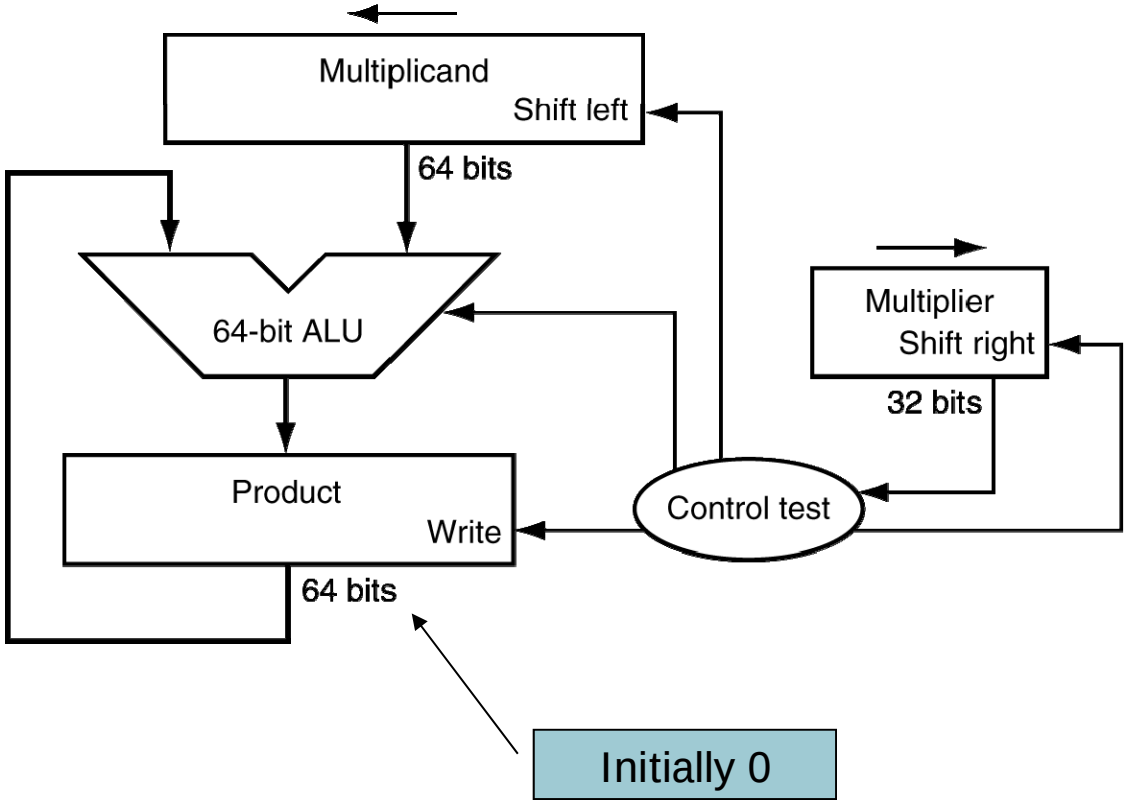
```
  1000
× 1001
------
  1000
 0000
0000
1000
------
1001000
```

Length of product is the sum of operand lengths



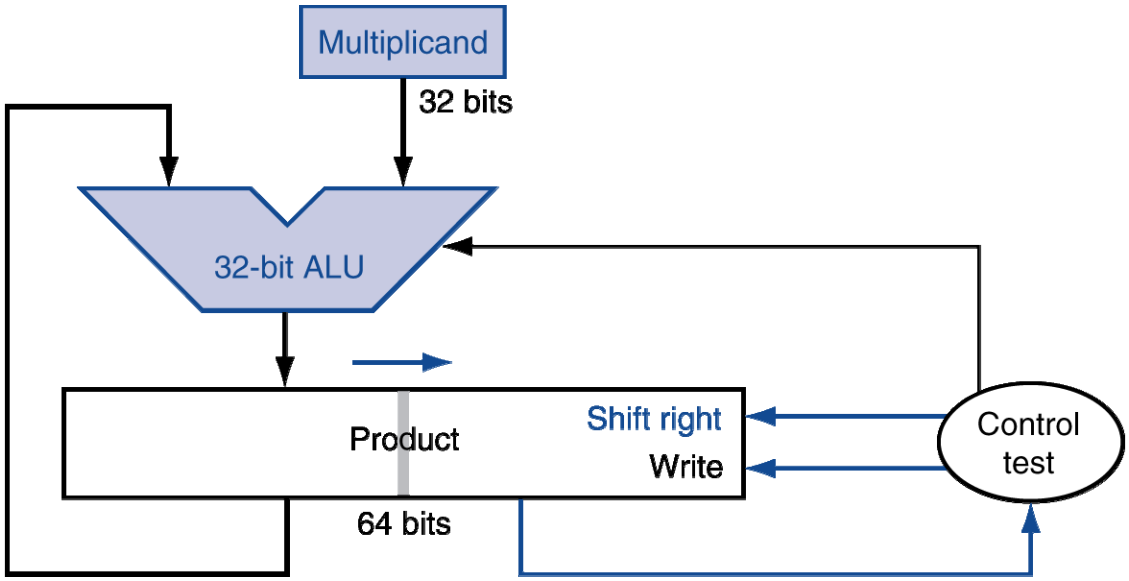
Multiplication Hardware





Optimized Multiplier

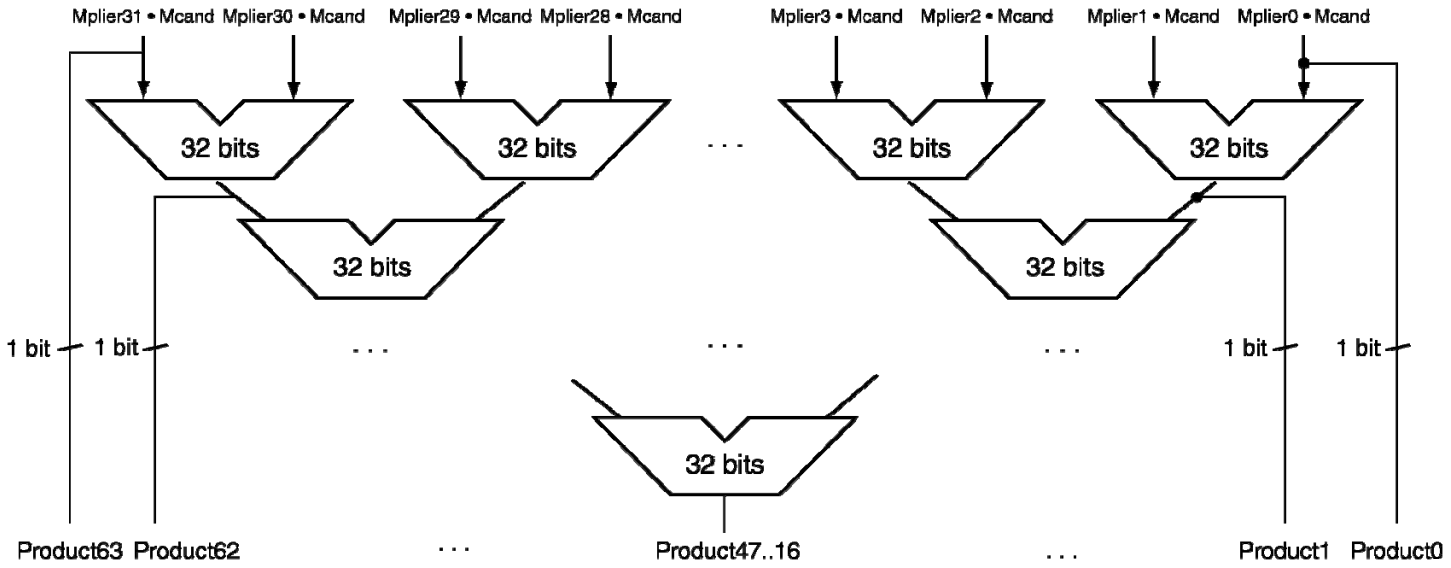
- Perform steps in parallel: add/shift



- One cycle per partial-product addition
  - That's ok, if frequency of multiplications is low

Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff



- Can be pipelined
  - Several multiplication performed in parallel

MIPS Multiplication

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Instructions
  - `mult rs, rt  /  multu rs, rt`
    - 64-bit product in HI/LO
  - `mfhi rd  /  mflo rd`
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - `mul rd, rs, rt`
    - Least-significant 32 bits of product –> rd

Chapter 3

Arithmetic for Computers

Signed Multiplication?

- Make both positive
  - remember whether to complement product when done
- Apply definition of 2's complement
  - need to sign-extend partial products and subtract at the end
- Booth's Algorithm
  - elegant way to multiply signed numbers
    - using same hardware as before and save cycles

Motivation for Booth's Algorithm

- Example 2 x 6 = 0010 x 0110:

```
                0010
        x       0110
        ---------------
        +       0000    shift (0 in multiplier)
        +      0010     add (1 in multiplier)
        +     0010      add (1 in multiplier)
        +    0000       shift (0 in multiplier)
        ---------------
        00001100
```

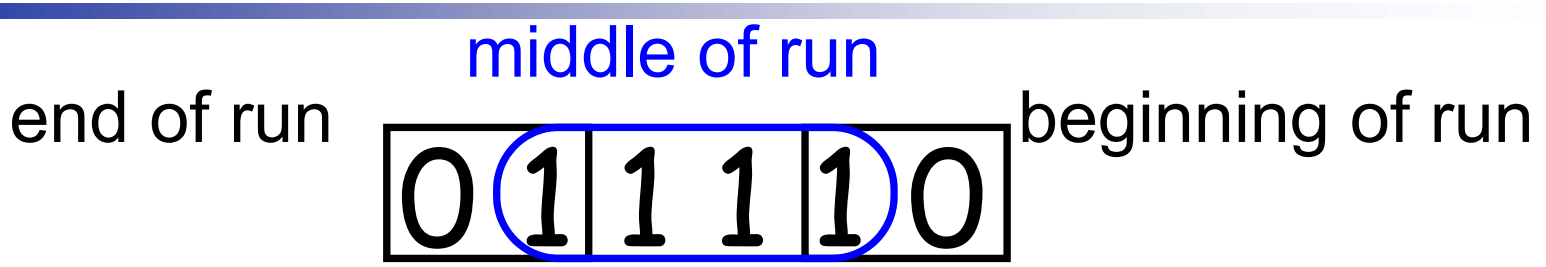
- ALU with add or subtract gets same result in more than one way:

```
14     = 2 + 4 + 8
14     = – 2 + 16
001110 = – 000010 + 010000 = 111110 + 010000
```

- For example

```
                0010
        x       0110
        ---------------
                0000 shift (0 in multiplier)
        –       0010 sub (first 1 in multpl.) .
               0000  shift (mid string of 1s) .
        +     0010    add (prior step had last 1)
        ---------------
        00001100
```

Booth's Algorithm



Current Bit	Bit to the Right	Explanation	Example	Op
1	0	Begins run of 1s	0001111000	sub
1	1	Middle of run of 1s	0001111000	none
0	1	End of run of 1s	0001111000	add
0	0	Middle of run of 0s	0001111000	none

Originally for Speed (when shift was faster than add)

- Replace a string of 1s in multiplier with an initial subtract when we first see a one and then later add for the bit after the last one

```
    -1
+ 10000
-------
 01111
```

Chapter 3

Arithmetic for Computers

IEEE Floating-Point Format

single: 8 bits double: 11 bits	single: 23 bits double: 52 bits

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: 1.0 ≤ |significand| < 2.0
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the “1.” restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

Floating-Point Addition

- Consider a 4-digit decimal example
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$
- 1. Align decimal points
  - Shift number with smaller exponent
  - $9.999 \times 10^{1} + 0.016 \times 10^{1}$
- 2. Add significands
  - $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$
- 3. Normalize result & check for over/underflow
  - 1.0015×10^{2}
- 4. Round and renormalize if necessary
  - 1.002×10^{2}

FP Adder Hardware