MULTIPLY (unsigned)

- Paper and pencil example (unsigned):
  
  **Multiplicand** → 1000 = 8  
  **Multiplier** → \times 1001 = 9  
  1000  
  0000  
  0000  
  1000  
  
  **Product** → 01001000 = 72

- n bits × n bits = 2n bit product

- Binary makes it easy:
  - 0 => place 0  (0 x multiplicand)
  - 1 => place a copy  (1 x multiplicand)

- 4 versions of multiply hardware & algorithm:
  - successive refinement
Unsigned shift-add multiplier (version 1)

- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg

Multiplier = datapath + control
Multiply Algorithm Version 1

1. Test Multiplier0
   - Multiplier0 = 1
     1a. Add multiplicand to product & place the result in Product register
   - Multiplier0 = 0

2. Shift the Multiplicand register left 1 bit.

3. Shift the Multiplier register right 1 bit.

nth repetition?

Yes: n repetitions

No: < n repetitions

Product | Multiplier | Multiplicand
--------|------------|-----------
0000 0000 | 0011 | 0000 0010
0000 0010 | 0001 | 0000 0100
0000 0110 | 0000 | 0000 1000
0000 0110 | 0000 | 0001 0000
0000 0110 | 0000 | 0010 0000

0000 0110
Observations on Multiply Version 1

- 1/2 bits in multiplicand always 0
  => 64-bit adder is wasted
- 0’s inserted into the least significant bit of multiplicand as shifted => least significant bits of product never changed once formed
- Instead of shifting multiplicand to left, shift product to right.
MULTIPLY HARDWARE Version 2

- 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg, 32-bit Multiplier reg
Multiply Algorithm Version 2

1. Test Multiplier0
   - Multiplier0 = 1
     1a. Add multiplicand to the left half of product & place the result in the left half of Product register
   - Multiplier0 = 0

2. Shift the Product register right 1 bit.

3. Shift the Multiplier register right 1 bit.

n\textsuperscript{th} repetition?
   - No: < n repetitions
   - Yes: n repetitions

Done

<table>
<thead>
<tr>
<th>Product</th>
<th>Multiplier</th>
<th>Multiplicand</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>0011</td>
<td>0010</td>
</tr>
<tr>
<td>1: 0010 0000</td>
<td>0011</td>
<td>0010</td>
</tr>
<tr>
<td>2: 0001 0000</td>
<td>0011</td>
<td>0010</td>
</tr>
<tr>
<td>3: 0001 0000</td>
<td>0001</td>
<td>0010</td>
</tr>
<tr>
<td>1: 0011 0000</td>
<td>0001</td>
<td>0010</td>
</tr>
<tr>
<td>2: 0001 1000</td>
<td>0001</td>
<td>0010</td>
</tr>
<tr>
<td>3: 0001 1000</td>
<td>0000</td>
<td>0010</td>
</tr>
<tr>
<td>1: 0001 1000</td>
<td>0000</td>
<td>0010</td>
</tr>
<tr>
<td>2: 0000 1100</td>
<td>0000</td>
<td>0010</td>
</tr>
<tr>
<td>3: 0000 1100</td>
<td>0000</td>
<td>0010</td>
</tr>
<tr>
<td>1: 0000 1100</td>
<td>0000</td>
<td>0010</td>
</tr>
<tr>
<td>2: 0000 0110</td>
<td>0000</td>
<td>0010</td>
</tr>
<tr>
<td>3: 0000 0110</td>
<td>0000</td>
<td>0010</td>
</tr>
</tbody>
</table>

0000 0110 0000 0010
Still more wasted space!

Start

1. Test Multiplier0

Multiplier0 = 1

1a. Add multiplicand to the left half of product & place the result in the left half of Product register

Multiplier0 = 0

2. Shift the Product register right 1 bit.

3. Shift the Multiplier register right 1 bit.

nth repetition?

Yes: n repetitions

No: < n repetitions

Done

Product  Multiplier  Multiplicand

0000 0000  0011  0010

1: 0010 0000  0011  0010

2: 0001 0000  0011  0010

3: 0001 0000  0001  0010

1: 0011 0000  0001  0010

2: 0001 1000  0001  0010

3: 0001 1000  0000  0010

1: 0001 1000  0000  0010

2: 0000 1100  0000  0010

3: 0000 1100  0000  0010

1: 0000 1100  0000  0010

2: 0000 0110  0000  0010

3: 0000 0110  0000  0010

0000 0110  0000  0010
Observations on Multiply Version 2

- Product register wastes space that exactly matches size of multiplier
- Both Multiplier register and Product register require right shift
- Combine Multiplier register and Product register
MULTIPLY HARDWARE Version 3

- 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg, (0-bit Multiplier reg)
Multiply Algorithm Version 3

**Multiplicand**  **Multiplier**
0010  0011

1. Test Product0

   - **Product0 = 1**
     - 1a. Add multiplicand to the left half of product & place the result in the left half of Product register
   
     | Product        | Multiplicand |
     |----------------|--------------|
     | 0000 0011      | 0010         |
     | 1: 0010 0011   | 0010         |
     | 2: 0001 0001   | 0010         |
     | 1: 0011 0001   | 0010         |
     | 2: 0001 1000   | 0010         |
     | 1: 0001 1000   | 0010         |
     | 2: 0000 1100   | 0010         |
     | 1: 0000 1100   | 0010         |
     | 2: 0000 0110   | 0010         |

   - **Product0 = 0**

2. Shift the Product register right 1 bit.

32nd repetition?

   - **No:** < 32 repetitions
   
   - **Yes:** 32 repetitions

**Done**
Observations on Multiply Version 3

- 2 steps per bit because Multiplier & Product combined
- MIPS registers Hi and Lo are left and right half of Product
- Gives us MIPS instruction MultU
- What about signed multiplication?
  - easiest solution is to make both positive & remember whether to complement product when done (leave out the sign bit, run for 31 steps)
  - Multiply algorithm 3 will work for signed numbers if partial products are sign-extended as shifted
  - Booth’s Algorithm is elegant way to multiply signed numbers using same hardware as before and save cycles
    - can be modified to handle multiple bits at a time
Faster Multiplication

- Whether the multiplicand is to be added or not is known at the beginning of the operation.
- Provide a 32-bit adder for each bit of the multiplier.
- One input is the multiplicand ANDed with a multiplier bit and the other is the output of a prior adder.
- Speed: just the overhead of a clock for each bit of the product. \( \log_2(32) \)
Shifters

Two kinds:

*logical*-- value shifted in is always "0"

"0" → \( \text{msb} \quad \text{lsb} \) ← "0"

*arithmetic*-- on right shifts, sign extend

\[ \text{msb} \quad \text{lsb} \] ← "0"

Note: these are single bit shifts. A given instruction might request 0 to 32 bits to be shifted!
Combinational Shifter from MUXes

Basic Building Block

A B
sel 1 0 2-to-1 Mux
D

8-bit right shifter

A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0

1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0

R_7 R_6 R_5 R_4 R_3 R_2 R_1 R_0

S_2 S_1 S_0

- What comes in the MSBs?
- How many levels for 32-bit shifter?
Unsigned Divide: Paper & Pencil

\[
\begin{array}{c|c|c}
\text{Divisor} & 1000 & \text{Dividend} \\
\hline
\text{Quotient} & 1001 & 1001010 \\
\hline
\text{-Divisor} & \text{-1000} & \text{Remainder (or Modulo result)} \\
\hline
10 & 101 & 1010 \\
\hline
10 & \text{-1000} & 10 \\
\hline
\end{array}
\]

See how big a number can be subtracted, creating quotient bit on each step.

- Binary => 1 * divisor or 0 * divisor
- Dividend = Quotient x Divisor + Remainder
- 3 versions of divide, successive refinement
DIVIDE HARDWARE Version 1

- 64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg
Divide Algorithm Version 1

- Takes \( n+1 \) steps for \( n \)-bit Quotient & Rem.

<table>
<thead>
<tr>
<th>Remainder</th>
<th>Quotient</th>
<th>Divisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0111</td>
<td>0000</td>
<td>0010 0000</td>
</tr>
</tbody>
</table>

1. Subtract the Divisor register from the Remainder register, and place the result in the Remainder register.

2a. Shift the Quotient register to the left setting the new rightmost bit to 1.

2b. Restore the original value by adding the Divisor register to the Remainder register, & place the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0.

3. Shift the Divisor register right 1 bit.

Start: Place Dividend in Remainder

Test Remainder

- Remainder \( \geq 0 \)
- Remainder \( < 0 \)

- Yes: \( n+1 \) repetitions (\( n = 4 \) here)
- No: < \( n+1 \) repetitions

Done
Observations on Divide Version 1

- 1/2 bits in divisor always 0
  => 1/2 of 64-bit adder is wasted
  => 1/2 of divisor is wasted

- Instead of shifting divisor to right, shift remainder to left?

- 1st step cannot produce a 1 in quotient bit (otherwise too big)
  => switch order to shift first and then subtract, can save 1 iteration
Divide: Paper & Pencil

\[
\begin{array}{cccc}
\text{Divisor} & 0001 & \underline{00001010} & \text{Quotient} \\
\text{Dividend} & 0001 & \underline{-0001} & 0000 \\
\underline{-0001} & 0001 & \underline{-0001} & 0 \\
\underline{0000} & 0001 & \underline{-0001} & 00 \\
\end{array}
\]

- Remainder (or Modulo result)

- Notice that there is no way to get a 1 in leading digit!
  (this would be an overflow, since quotient would have n+1 bits)
DIVIDE HARDWARE Version 2

- 32-bit Divisor reg, 32-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg
Divide Algorithm Version 2

Start: Place Dividend in Remainder

1. Shift the Remainder register left 1 bit.

2. Subtract the Divisor register from the left half of the Remainder register, & place the result in the left half of the Remainder register.

Test Remainder

Remainder ≥ 0

3a. Shift the Quotient register to the left setting the new rightmost bit to 1.

Remainder < 0

3b. Restore the original value by adding the Divisor register to the left half of the Remainder register, & place the sum in the left half of the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0.

nth repetition?

No: < n repetitions

Yes: n repetitions (n = 4 here)

Done
Observations on Divide Version 2

- Eliminate Quotient register by combining with Remainder as shifted left
  - Start by shifting the Remainder left as before.
  - Thereafter loop contains only two steps because the shifting of the Remainder register shifts both the remainder in the left half and the quotient in the right half.
  - The consequence of combining the two registers together and the new order of the operations in the loop is that the remainder will shifted left one time too many.
  - Thus the final correction step must shift back only the remainder in the left half of the register.
DIVIDE HARDWARE Version 3

- 32-bit Divisor reg, 32-bit ALU, 64-bit Remainder reg, (0-bit Quotient reg)
Divide Algorithm Version 3

Start: Place Dividend in Remainder

1. Shift the Remainder register left 1 bit.

2. Subtract the Divisor register from the left half of the Remainder register, & place the result in the left half of the Remainder register.

3a. Shift the Remainder register to the left setting the new rightmost bit to 1.

3b. Restore the original value by adding the Divisor register to the left half of the Remainder register, & place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new least significant bit to 0.

Test

Remainder \geq 0

Remainder < 0

Yes: n repetitions (n = 4 here)

No: < n repetitions

nth repetition?

Done. Shift left half of Remainder right 1 bit.

Divisor

0010

Remainder

0000 0111

Remainder

0010
Observations on Divide Version 3

- Same Hardware as Multiply: just need ALU to add or subtract, and 64-bit register to shift left or shift right
- Hi and Lo registers in MIPS combine to act as 64-bit register for multiply and divide
- Signed Divides: Simplest is to remember signs, make positive, and complement quotient and remainder if necessary
  - Note: Dividend and Remainder must have same sign
  - Note: Quotient negated if Divisor sign & Dividend sign disagree e.g., \(-7 \div 2 = -3\), remainder = \(-1\)