Week 3, part D: Booth’s Algorithm

4) 3 x 9 = ?

$$= 3 \times \sqrt{81} = 3\sqrt{81} = 3\overline{)81} = 27$$

$$\begin{array}{r} 27 \\ 3\overline{)81} \\ \underline{6} \\ 21 \\ \underline{21} \\ 0 \end{array}$$

Booth's Algorithm

- In real life we often see sequences of bits: `00111000`
- Designed to take advantage of the fact that in circuits, shifting is cheaper than adding and space is at a premium.
  - Based on the premise that when multiplying by certain values (e.g. `99`), it can be easier to think of this operation as a difference between two products.
- Consider the shortcut method when multiplying a given decimal value `X` by `9999`:
  - `X*9999 = X*10000 - X*1`
- Now consider the equivalent problem in binary:
  - `X*001111 = X*010000 - X*1`

Sign Extension

- We want to subtract 4-bit number from 8-bit number…
- …how do we convert a 4-bit two's complement number to 8-bit, without changing its value?
- Sign extend: replicate most significant bit

`0101`	→	`00000101`	`1001`	→	`11111001`
(5)		(still 5)	(-7)		(still -7)

- Zero extend: pad with zeros: 1001 → 00001001
- Arithmetic shift right: shift right and replicate sign bit

Booth's Example in Decimal

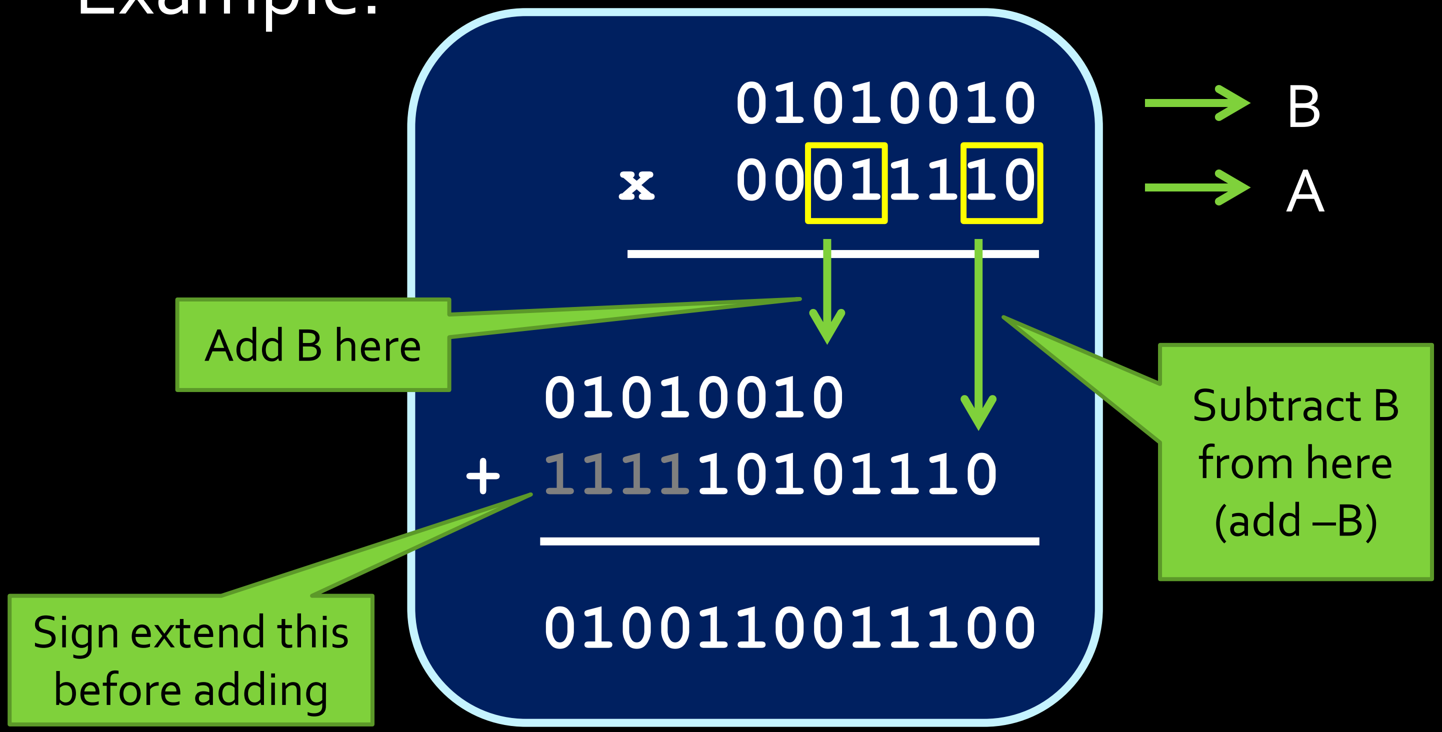
- Compute 999 x 5 →
  - 1000 x 5 – 1 x 5 → 5,000 – 5 = 4,995
- Compute 99,900 x 5 →
  - 100,000 x 5 – 100 x 5 = 500,000 – 500 = 499,500
- Compute 999,099 x 5 →
  - 1,000,000 x 5 – 1,000 x 5
    → 5,000,000 – 5,000 = 4,995,000
  - 100 x 5 – 1 x 5 → 500 – 5 = 495
  - 4,995,000 + 495 = **4,995,495**

Booth's Algorithm

- This idea is applied when two neighboring digits in an operand are different.
- Go through digits from n-1 to 0
  - If digits at `i` and `i-1` are `0` and `1`, the multiplicand is added to the result at position `i`.
  - If digits at `i` and `i-1` are `1` and `0`, the multiplicand is subtracted from the result at position `i`.
- The result is always a value whose size is the sum of the sizes of the two multiplicands.
  - Need n+k bits to multiply n-bit number by k-bit number

Booth's Algorithm

- Example:



Booth's Algorithm

- We need to make this work in hardware.
  - Option #1: Have hardware set up to compare neighbouring bits at every position in A, with adders in place for when the bits don't match.
  - Problem: This is a lot of hardware, which Booth's Algorithm is trying to avoid.
  - Option #2: Have hardware set up to compare two neighbouring bits, and have them move down through A, looking for mismatched pairs.
  - Problem: Hardware doesn't move like that. Oops.

Booth's Algorithm

- Still need to make this work in hardware…
  - Option #3: Have hardware set up to compare two neighbouring bits in the lowest position of `A`, and looking for mismatched pairs in `A` by shifting `A` to the right one bit at a time.
  - Solution! This could work, but the accumulated solution `P` would have to shift one bit at a time as well, so that when `B` is added or subtracted, it's from the correct position.

Booth’s Algorithm

Note: unlike the accumulator, the bits here are being shifted to the right!

- Steps in Booth’s Algorithm:
  1. Designate the two multiplicands as A & B, and the result as some product P.
  2. Add an extra zero bit to the right-most side of A.
  3. Repeat the following for each original bit in A:
     a) If the last two bits of A are the same, do nothing.
     b) If the last two bits of A are `01`, then add B to the highest bits of P.
     c) If the last two bits of A are `10`, then subtract B from the highest bits of P.
     d) Perform one-digit arithmetic right-shift on both P and A.
  4. The result in P is the product of A and B.

Booth's Algorithm Example

- Example: (-5) * 2

- Steps #1 & #2:
  - A = -5 → 11011
    - Add extra zero to the right → A = 11011 0
  - B = 2 → 00010
  - -B = -2 → 11110
  - P = 0 → 00000 00000

Booth's Algorithm Example

```
A =  11011 0
P =  00000 00000
```

- Step #3 (repeat 5 times):
  - Check last two digits of A:

    1101 $\boxed{10}$

  - Since digits are 10, subtract B from the most significant digits of P:

```
   P     00000 00000
  -B    +11110
  ------------------
   P'    11110 00000
```

  - Arithmetic shift P and A one bit to the right:
    - A = 111011   P = 11111 00000

Booth's Algorithm Example

- Step #3 (repeat 4 more times):
  - Check last two digits of A:

    1110 $\boxed{11}$

  - Since digits are 11, do nothing to P.

  - Arithmetic shift P and A one bit to the right:
    - A = 111101   P = 11111 10000

```
A =  11101 1
P =  11111 00000
```

Booth's Algorithm Example

```
A =  11110 1
P =  11111 10000
```

- Step #3 (repeat 3 more times):
  - Check last two digits of A:

    1111 $\boxed{01}$

  - Since digits are 01, add B to the most significant digits of P:

```
     P     11111 10000
    +B    +00010
  ------------------------
     P'    00001 10000
```

  - Arithmetic shift P and A one bit to the right:
    - A = 111110   P = 00000 11000

Booth's Algorithm Example

```
A =  11111 0
P =  00000 11000
```

- Step #3 (repeat 2 more times):
  - Check last two digits of A:

    1111 $\boxed{10}$

  - Since digits are 10, subtract B from the most significant digits of P:

```
 P     00000 11000
-B    +11110
-----------------------
 P'    11110 11000
```

  - Arithmetic shift P and A one bit to the right:
    - A = 111111   P = 11111 01100

Booth’s Algorithm Example

```
A =  11111 1
P =  11111 01100
```

- Step #3 (final time):
  - Check last two digits of A:

    1111 $\boxed{11}$

  - Since digits are 11, do nothing to P:
  - Arithmetic shift P and A one bit to the right:
    - A = 111111   P = 11111 10110

Booth’s Algorithm Example

- Done!

```
A =  11111 1
P =  11111 10110
```

- Final product: $P = 111110110$

  $= -10$

Reflections on multiplication

- A popular version of this algorithm involves copying A into the lower bits of P, so that the testing and shifting only takes place in P.
- Common multiplication and division operations are often powers of 2.
  - We can use a shifter instead of the multiplier circuit.
  - (recall W3 Review)

Reflections on multiplication

- Early CPUs such as Intel 8080 and MOS 6502 did not have a multiplication unit.
- Multiplication was done in software, by using multiple additions, bit shifts, and table lookups.
  - This was very slow.
- Multiplication is less common than addition or subtraction, but is still frequent.
- Hence modern CPUs have multipliers.

Back to the big picture

- We built an ALU
- How do we feed it data? What do we do with the result?
- Move to next part