## Quote

*I do not like $\times$ as a symbol for multiplication, as it is easily confounded with $x$; often I simply relate two quantities by an interposed dot and indicate multiplication by $ZC \cdot LM$.*

*– Leibniz*

- Goal: develop a circuit for integer multiplication which
  1. functions correctly, and
  2. is efficient (wrt. number of gates and critical path).
- This is important in that
  1. there is a *vast* design space of possible approaches, because
  2. multiplication represents one of the more complex operations in an ALU, and hence a potential bottleneck.

# COMS12200 lecture: week #6



## MUL—Unsigned Multiply

| Opcode | Instruction | Op/En | 64-Bit Mode | Compat/Leg Mode | Description |
|---|---|---|---|---|---|
| F6 /4 | MUL *r/m8* | A | Valid | Valid | Unsigned multiply (AX ← AL ∗ *r/m8*). |
| REX + F6 /4 | MUL *r/m8** | A | Valid | N.E. | Unsigned multiply (AX ← AL ∗ *r/m8*). |
| F7 /4 | MUL *r/m16* | A | Valid | Valid | Unsigned multiply (DX:AX ← AX ∗ *r/m16*). |
| F7 /4 | MUL *r/m32* | A | Valid | Valid | Unsigned multiply (EDX:EAX ← EAX ∗ *r/m32*). |
| REX.W + F7 /4 | MUL *r/m64* | A | Valid | N.E. | Unsigned multiply (RDX:RAX ← RAX ∗ *r/m64*. |

**NOTES:**

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (r) | NA | NA | NA |

### Description

Performs an unsigned multiplication of the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand is an implied operand located in register AL, AX or EAX (depending on the size of the operand); the source operand is located in a general-purpose register or a memory location. The action of this instruction and the location of the result depends on the opcode and the operand size as shown in Table 3-61.

The result is stored in register AX, register pair DX:AX, or register pair EDX:EAX (depending on the operand size), with the high-order bits of the product contained in register AH, DX, or EDX, respectively. If the high-order bits of the product are 0, the CF and OF flags are cleared; otherwise, the flags are set.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

See the summary chart at the beginning of this section for encoding data and limits.





**Table 3-61. MUL Results**

| Operand Size | Source 1 | Source 2 | Destination |
|---|---|---|---|
| Byte | AL | r/m8 | AX |
| Word | AX | r/m16 | DX:AX |
| Doubleword | EAX | r/m32 | EDX:EAX |
| Quadword | RAX | r/m64 | RDX:RAX |

### Operation

```
IF (Byte operation)
    THEN
        AX ← AL ∗ SRC;
    ELSE (* Word or doubleword operation *)
        IF OperandSize = 16
            THEN
                DX:AX ← AX ∗ SRC;
            ELSE IF OperandSize = 32
                THEN EDX:EAX ← EAX ∗ SRC; FI;
            ELSE (* OperandSize = 64 *)
                RDX:RAX ← RAX ∗ SRC;
        FI;
FI;
```

### Flags Affected

The OF and CF flags are set to 0 if the upper half of the result is 0; otherwise, they are set to 1. The SF, ZF, AF, and PF flags are undefined.

### Protected Mode Exceptions

| | |
|---|---|
| # GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| # SS(0) | If a memory operand effective address is outside the SS segment limit. |
| # PF(fault-code) | If a page fault occurs. |
| # AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| # UD | If the LOCK prefix is used. |





### Real-Address Mode Exceptions

| | |
|---|---|
| # GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| # SS | If a memory operand effective address is outside the SS segment limit. |
| # UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| # GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| # SS(0) | If a memory operand effective address is outside the SS segment limit. |
| # PF(fault-code) | If a page fault occurs. |
| # AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| # UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| # SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| # GP(0) | If the memory address is in a non-canonical form. |
| # PF(fault-code) | If a page fault occurs. |
| # AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

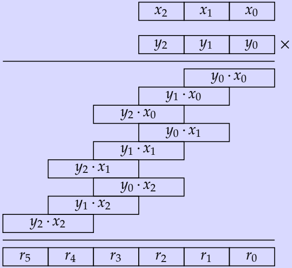
# Basic Concepts (1)

- An unsigned, integer multiplier takes two $n$-bit inputs
  1. $x$, the **multiplicand** that is multiplied, and
  2. $y$, the **multiplier** that does the multiplying

  and computes their $2n$-bit **product** $r$ as output.

# Basic Concepts (2)

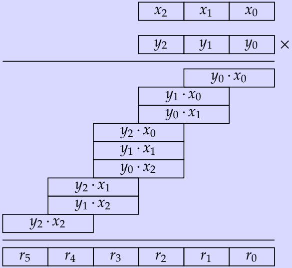
## Example (operand scanning, $|x| = |y| = 3$)




Notice that

1. an outer-loop steps through limbs of $y$, say $y_i$,
2. an inner-loop steps through limbs of $x$, say $x_j$.

## Example (product scanning, $|x| = |y| = 3$)




Notice that

1. an outer-loop steps through limbs of $r$, say $r_i$,
2. two inner-loops step through matching limbs of $x$ and $y$, say $x_j$ and $x_i$.

# Basic Concepts (3)

- Scalar multiplication of $x$ by $y$ is simply repeated addition, i.e.,

$$x \cdot y = \underbrace{x + x + \cdots + x + x}_{y \text{ terms}},$$

  so if we select $y = 14_{(10)}$, then we obviously have

$$x \cdot 14 = x + x + x + x + x + x + x + x + x + x + x + x + x + x.$$

- Another way of look it, is inclusion of another "weight" to the digits that describe $y$; if we write $y$ out in binary then

$$x \cdot y \equiv x \cdot \sum_{i=0}^{n-1} y_i \cdot 2^i \equiv \sum_{i=0}^{n-1} y_i \cdot x \cdot 2^i$$

  and, as such, if $y = 14_{(10)} = 1110_{(2)}$ then

$$\begin{array}{lcccccccc} y \cdot x & = & y_0 \cdot x \cdot 2^0 & + & y_1 \cdot x \cdot 2^1 & + & y_2 \cdot x \cdot 2^2 & + & y_3 \cdot x \cdot 2^3 \\ & = & 0 \cdot x \cdot 2^0 & + & 1 \cdot x \cdot 2^1 & + & 1 \cdot x \cdot 2^2 & + & 1 \cdot x \cdot 2^3 \\ & = & 0 \cdot x & + & 2 \cdot x & + & 4 \cdot x & + & 8 \cdot x \\ & = & 14 \cdot x & & & & & & \end{array}$$

# Basic Concepts (4)

- If we bracket this in a nicer way, we can accumulate the result rather than express it as separate terms ...
- ... applying **Horner's rule**; for $y = 14_{(10)} = 1110_{(2)}$ we write

$$
\begin{array}{lcl}
y \cdot x & = & y_0 \cdot x + 2 \cdot ( y_1 \cdot x + 2 \cdot ( y_2 \cdot x + 2 \cdot ( y_3 \cdot x + 2 \cdot ( 0 ) ) ) ) \\
& = & 0 \cdot x + 2 \cdot ( 1 \cdot x + 2 \cdot ( 1 \cdot x + 2 \cdot ( 1 \cdot x + 2 \cdot ( 0 ) ) ) ) \\
& = & 0 \cdot x + 2 \cdot ( 1 \cdot x + 2 \cdot ( 1 \cdot x + 2 \cdot ( 1 \cdot x + 0 ) ) ) \\
& = & 0 \cdot x + 2 \cdot ( 1 \cdot x + 2 \cdot ( 1 \cdot x + 2 \cdot ( 1 \cdot x ) ) ) \\
& = & 0 \cdot x + 2 \cdot ( 1 \cdot x + 2 \cdot ( 1 \cdot x + 2 \cdot x ) ) \\
& = & 0 \cdot x + 2 \cdot ( 1 \cdot x + 2 \cdot ( 3 \cdot x ) ) \\
& = & 0 \cdot x + 2 \cdot ( 1 \cdot x + 6 \cdot x ) \\
& = & 0 \cdot x + 2 \cdot ( 7 \cdot x ) \\
& = & 0 \cdot x + 14 \cdot x \\
& = & 14 \cdot x
\end{array}
$$

# Example Designs (1)

- Idea:
  - reading the result of applying the Horner Rule "inside-out" hints at a loop based on an accumulator $t$,
  - that is, if $y_i = 1$ we set $t$ to $2 \cdot t + x$ or $2 \cdot t$ otherwise.

## Algorithm (MULTIPLY)

**Input**: Two unsigned, $n$-bit, base-2 integers $x$ and $y$
**Output**: An unsigned, $2n$-bit, base-2 integer $r = y \cdot x$

```
t ← 0
for i = n − 1 downto 0 step −1 do
    t ← 2 · t
    if y_i = 1 then
        t ← t + x
    end
end
return t
```

## Example

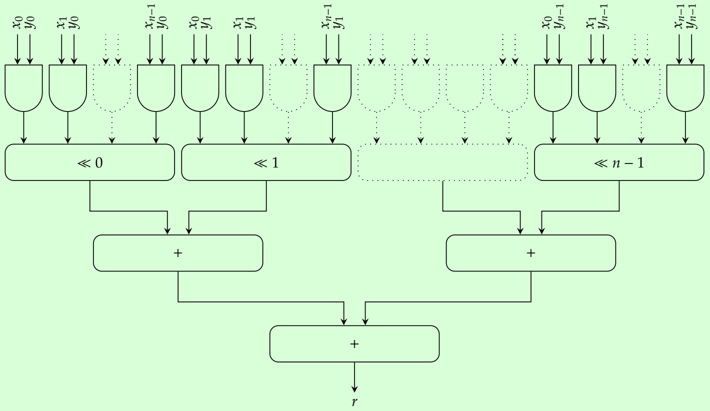
Setting $n = 4$ and $y = 14_{(10)} = 1110_{(2)}$, the algorithm performs

| $i$ | $t$ | $y_i$ | $t'$ | |
|---|---|---|---|---|
| | $0$ | | | |
| $3$ | $0$ | $1$ | $x$ | $t' \leftarrow 2 \cdot t + x$ |
| $2$ | $x$ | $1$ | $3 \cdot x$ | $t' \leftarrow 2 \cdot t + x$ |
| $1$ | $3 \cdot x$ | $1$ | $7 \cdot x$ | $t' \leftarrow 2 \cdot t + x$ |
| $0$ | $7 \cdot x$ | $0$ | $14 \cdot x$ | $t' \leftarrow 2 \cdot t$ |
| | $14 \cdot x$ | | | |

- Note that $n$ is *fixed*, so we can either
  1. unroll the loop and produce a combinatorial circuit often termed a **tree multiplier**, or
  2. keep the loop and produce an iterative circuit often termed a **bit-serial multiplier**.

# Example Designs (2)

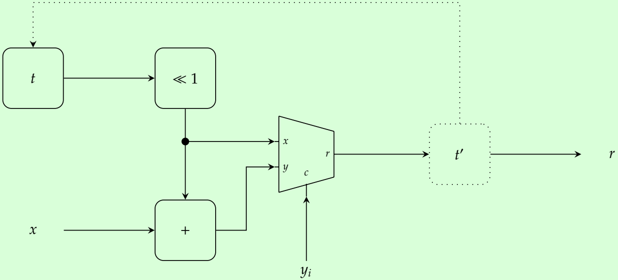
## Circuit




- Bad: requires a larger data-path (although less of a control-path).
- Good: requires less steps (i.e., 1) to compute a result, meaning shorter latency.

# Example Designs (3)

## Circuit




- Bad: requires more steps (i.e., $n$) to compute a result, meaning longer latency.
- Good: requires a smaller data-path (although more of a control-path).

# Booth Recoding (1)

- Question: what's the most efficient way to compute $15 \cdot x$?

# Booth Recoding (1)

- Question: what's the most efficient way to compute $15 \cdot x$?
- Answer #1: shifts by fixed distances are "free", so

$$\begin{array}{rcllllllll} 15 \cdot x & = & 1 \cdot x & + & 2 \cdot x & + & 4 \cdot x & + & 8 \cdot x \\ & = & x & + & x \ll 1 & + & x \ll 2 & + & x \ll 3 \end{array}$$

- Answer #2: remember we can do subtraction more or less for the same cost as addition, so

$$\begin{array}{rclll} 15 \cdot x & = & 16 \cdot x & - & 1 \cdot x \\ & = & x \ll 4 & - & x \end{array}$$

- **Booth recoding** is basically a generalisation of the second approach
  1. spend some effort *before* multiplication to **recode** $y$ into some $y'$, then
  2. be more efficient *during* multiplication by using $y'$.

# Booth Recoding (2)

- Idea:
  - take advantage of the fact that addition *and* subtraction are possible by
  - recoding $y$ to eliminate "runs" of 1 or 0 bits.
- Example: given the 8-bit multiplier

$$30_{(10)} = 00011110_{(2)}$$

  our strategy is as follows:
  - For a sub-sequence of 1 bits between $i$ and $j$, we treat the sub-sequence as a single digit of weight $2^{j+1} - 2^i$.
  - In this case $i = 1$ to $j = 4$, so we treat

$$2^4 + 2^3 + 2^2 + 2^1 = 30$$

    as a single digit of weight

$$2^{4+1} - 2^1 = 2^5 - 2^1 = 30$$

  - Now, instead of accumulating 4 partial products we just need to accumulate 2; this can clearly takes less steps.

# Booth Recoding (3)

## Example

Consider setting $x = 6_{(10)} = 00000110_{(2)}$ and $y = 30_{(10)} = 00011110_{(2)}$; using the iterative multiplier, we get

$$
\begin{array}{llrrc|cccccccccccccccc|c}
x & = & & 6_{(10)} & \mapsto & & & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & \\
y & = & & 30_{(10)} & \mapsto & & & & & & & & & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & \times \\
\hline
p_0 & = & 0 \cdot x \cdot 2^0 = & 0_{(10)} & \mapsto & & & & & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
p_1 & = & +1 \cdot x \cdot 2^1 = & +12_{(10)} & \mapsto & & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & \\
p_2 & = & +1 \cdot x \cdot 2^2 = & +24_{(10)} & \mapsto & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & & \\
p_3 & = & +1 \cdot x \cdot 2^3 = & +48_{(10)} & \mapsto & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & & & \\
p_4 & = & +1 \cdot x \cdot 2^4 = & +96_{(10)} & \mapsto & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & & & & \\
p_5 & = & 0 \cdot x \cdot 2^5 = & 0_{(10)} & \mapsto & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & \\
p_6 & = & 0 \cdot x \cdot 2^6 = & 0_{(10)} & \mapsto & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & \\
p_7 & = & 0 \cdot x \cdot 2^7 = & 0_{(10)} & \mapsto & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & & \\
\hline
r & = & & 180_{(10)} & \mapsto & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & \\
\hline
\end{array}
$$

which requires accumulation of 4 non-zero partial products.

# Booth Recoding (4)

## Example

Consider setting $x = 6_{(10)} = 00000110_{(2)}$ and $y = 30_{(10)} = 00011110_{(2)}$; by first recoding $y$ into $y'$, we get

| | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $x$ | $=$ | | $6_{(10)}$ | $\mapsto$ | | | | | | | | | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | |
| $y$ | $=$ | | $30_{(10)}$ | $\mapsto$ | | | | | | | | | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | $\times$ |
| $y'_{(2)}$ | $=$ | | $30_{(10)}$ | $\mapsto$ | | | | | | | | | 0 | 0 | +1 | 0 | 0 | 0 | −1 | 0 | |
| $p_0$ | $=$ | $0 \cdot x \cdot 2^0 =$ | $0_{(10)}$ | $\mapsto$ | | | | | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| $p_1$ | $=$ | $-1 \cdot x \cdot 2^1 =$ | $-12_{(10)}$ | $\mapsto$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | | |
| $p_2$ | $=$ | $0 \cdot x \cdot 2^2 =$ | $0_{(10)}$ | $\mapsto$ | | | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | |
| $p_3$ | $=$ | $0 \cdot x \cdot 2^3 =$ | $0_{(10)}$ | $\mapsto$ | | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | |
| $p_4$ | $=$ | $0 \cdot x \cdot 2^4 =$ | $0_{(10)}$ | $\mapsto$ | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | |
| $p_5$ | $=$ | $+1 \cdot x \cdot 2^5 =$ | $+192_{(10)}$ | $\mapsto$ | | | | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | | | | | | |
| $p_6$ | $=$ | $0 \cdot x \cdot 2^6 =$ | $0_{(10)}$ | $\mapsto$ | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | | | |
| $p_7$ | $=$ | $0 \cdot x \cdot 2^7 =$ | $0_{(10)}$ | $\mapsto$ | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | | | | |
| $r$ | $=$ | | $180_{(10)}$ | $\mapsto$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | |

which requires accumulation of less non-zero partial products, i.e., 2 rather than 4.

# Booth Recoding (5)

## Example

Consider setting $x = 6_{(10)} = 00000110_{(2)}$ and $y = 5_{(10)} = 00000101_{(2)}$; by first recoding $y$ into $y'$, we get

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $x$ | $=$ | $6_{(10)} \mapsto$ | | | | | | | | | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | |
| $y$ | $=$ | $5_{(10)} \mapsto$ | | | | | | | | | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | $\times$ |
| $y'_{(2)}$ | $=$ | $5_{(10)} \mapsto$ | | | | | | | | | 0 | 0 | 0 | 0 | +1 | −1 | +1 | −1 | |
| $p_0$ | $= +1 \cdot x \cdot 2^0 =$ | $-6_{(10)} \mapsto$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | |
| $p_1$ | $= -1 \cdot x \cdot 2^1 =$ | $+12_{(10)} \mapsto$ | | | | | | | | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | | |
| $p_2$ | $= +1 \cdot x \cdot 2^2 =$ | $-24_{(10)} \mapsto$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | | | |
| $p_3$ | $= -1 \cdot x \cdot 2^3 =$ | $+48_{(10)} \mapsto$ | | | | | | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | | | | |
| $p_4$ | $= 0 \cdot x \cdot 2^4 =$ | $0_{(10)} \mapsto$ | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | |
| $p_5$ | $= 0 \cdot x \cdot 2^5 =$ | $0_{(10)} \mapsto$ | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | | |
| $p_6$ | $= 0 \cdot x \cdot 2^6 =$ | $0_{(10)} \mapsto$ | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | | | |
| $p_7$ | $= 0 \cdot x \cdot 2^7 =$ | $0_{(10)} \mapsto$ | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | | | | |
| $r$ | $=$ | $30_{(10)} \mapsto$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | |

which (still) requires accumulation of 4 non-zero partial products.

- Problem:
  1. in the worst-case, the recoding doesn't produce *less* non-zero partial products than in the standard iterative multiplier, and
  2. we need a way to translate any theoretical advantage into a concrete improvement.

# Booth Recoding (6)

- Solution: modified, radix-4 Booth recoding.
  1. reading right-to-left, group the recoded digits into pairs of the form $(y'_i, y'_{i+1})$, then
  2. treat each pair as a single digit whose value is $y'_i + 2 \cdot y'_{i+1}$ per

$$
\begin{array}{lclclcl}
y'_i & = & 0 & y'_{i+1} & = & 0 & \mapsto & 0 \\
y'_i & = & +1 & y'_{i+1} & = & 0 & \mapsto & +1 \\
y'_i & = & -1 & y'_{i+1} & = & 0 & \mapsto & -1 \\
y'_i & = & 0 & y'_{i+1} & = & +1 & \mapsto & +2 \\
y'_i & = & +1 & y'_{i+1} & = & +1 & \mapsto & \text{not possible} \\
y'_i & = & -1 & y'_{i+1} & = & +1 & \mapsto & +1 \\
y'_i & = & 0 & y'_{i+1} & = & -1 & \mapsto & -2 \\
y'_i & = & +1 & y'_{i+1} & = & -1 & \mapsto & -1 \\
y'_i & = & -1 & y'_{i+1} & = & -1 & \mapsto & \text{not possible}
\end{array}
$$

  meaning that $y'_{i+1}$ has twice the weight of $y'_i$.

- Example:
  - the pair $(-1, +1)$ represents $-1 + 2 \cdot +1 = +1$, and
  - the pair $(+1, -1)$ represents $+1 + 2 \cdot -1 = -1$.

# Booth Recoding (7)

## Example

Consider setting $x = 6_{(10)} = 00000110_{(2)}$ and $y = 5_{(10)} = 00000101_{(2)}$; by first recoding $y$ into $y'$ with the modified method, we get

| | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $x$ | $=$ | $6_{(10)} \mapsto$ | | | | | | | | | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| $y$ | $=$ | $5_{(10)} \mapsto$ | | | | | | | | | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 $\times$ |
| $y'_{(2)}$ | $=$ | $5_{(10)} \mapsto$ | | | | | | | | | 0 | 0 | +1 | 0 | +1 | −1 | +1 | −1 |
| $y'_{(4)}$ | $=$ | $5_{(10)} \mapsto$ | | | | | | | | | | | | | | +1 | | +1 |
| $p_0$ | $= +1 \cdot x \cdot 2^0 =$ | $+6_{(10)} \mapsto$ | | | | | | | | | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| $p_2$ | $= +1 \cdot x \cdot 2^2 =$ | $+24_{(10)} \mapsto$ | | | | | | | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | | |
| $p_4$ | $= 0 \cdot x \cdot 2^4 =$ | $0_{(10)} \mapsto$ | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | |
| $p_6$ | $= 0 \cdot x \cdot 2^6 =$ | $0_{(10)} \mapsto$ | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | | |
| $r$ | $=$ | $30_{(10)} \mapsto$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |

which

1. has a more reasonable number of non-zero partial products, and
2. makes the reduction in iterations clear as a result of considering digit pairs rather than individual digits

but means we now need to pre-shift partial products to cope with ±2 digits.

# Booth Recoding (8)

## Algorithm (Multiply-Booth-Modified)

**Input**: An unsigned, $n$-bit, base-2 integer $x$, and a radix-4 Booth recoding $y'_{(4)}$ of some integer $y$

**Output**: An unsigned, $2n$-bit, base-2 integer $r = y \cdot x$

```
t ← 0
for i = |y'| − 1 downto 0 step −1 do
    t ← t · 2^2
    if y'_i = −2 then
        t ← t − 2 · x
    end
    else if y'_i = −1 then
        t ← t − x
    end
    else if y'_i = +1 then
        t ← t + x
    end
    else if y'_i = +2 then
        t ← t + 2 · x
    end
end
return t
```

## Example

Setting $n = 4$ and $y = 14_{(10)} = 1110_{(2)}$, the algorithm recodes $y$ first into the radix-2 Booth encoding

$$\langle 0, -1, 0, 0, +1 \rangle$$

then the modified, radix-4 Booth encoding

$$\langle -2, 0, +1 \rangle$$

used as $y'$; then it performs

| $i$ | $y'_i$ | $t$ | $t'$ | |
|---|---|---|---|---|
| | | $0$ | | |
| $2$ | $+1$ | $0$ | $x$ | $t' \leftarrow 2^2 \cdot t + 1 \cdot x$ |
| $1$ | $0$ | $x$ | $4 \cdot x$ | $t' \leftarrow 2^2 \cdot t$ |
| $0$ | $-2$ | $4 \cdot x$ | $14 \cdot x$ | $t' \leftarrow 2^2 \cdot t - 2 \cdot x$ |
| | | $14 \cdot x$ | | |

noting that

1. $|y'| \simeq \frac{|y|}{2}$, and
2. in each step we multiply $t$ by $2^2 = 4$ (not $2^1 = 2$ as before).

# Conclusions

- Take away points:
  1. Computer arithmetic is a broad topic with a rich history; there is usually a large design space of potential approaches.
  2. Even if you don't care about arithmetic circuits, understanding their design can be useful since you use them all the time ...
  3. ... this needn't be too hard: most of the time, switching from $b = 10$ to $b = 2$ is the hardest step.
  4. For multiplication, the important steps are to
     - understand representation of numbers,
     - construct and/or translate algorithms (and algorithmic optimisations) into concrete designs, and
     - find a trade-off between various quality metrics (e.g., efficiency and area) for the given use-case

     plus, obviously, compute the correct result!

# References and Further Reading


[1] Wikipedia: Arithmetic Logic Unit (ALU).
http://en.wikipedia.org/wiki/Arithmetic_logic_unit.

[2] Wikipedia: Computer arithmetic.
http://en.wikipedia.org/wiki/Category:Computer_arithmetic.

[3] A.D. Booth.
A signed binary multiplication technique.
*Quarterly Journal of Mechanics and Applied Mathematics*, 4:236–240, 1951.

[4] L. Dadda.
Some schemes for parallel multipliers.
*Alta Frequenza*, 34:349–356, 1965.

[5] A. Karatsuba and Y. Ofman.
Multiplication of many-digital numbers by automatic computers.
*Physics-Doklady*, 7:595–596, 1963.

[6] D. Page.
Chapter 7: Arithmetic and logic.
In *A Practical Introduction to Computer Architecture*. Springer-Verlag, 1st edition, 2009.

[7] B. Parhami.
Part 3: Multiplication.
In *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, 1st edition, 2000.

# References and Further Reading


[8] W. Stallings.
Chapter 10: Computer arithmetic.
In *Computer Organisation and Architecture*. Prentice-Hall, 9th edition, 2013.

[9] C.S. Wallace.
A suggestion for fast multipliers.
*IEEE Transactions on Computers*, 13:14–17, 1964.