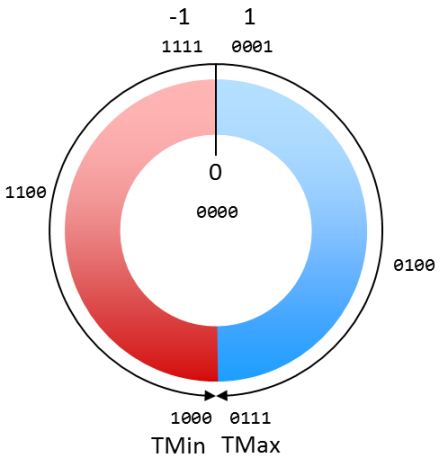
Data Representations

Integers



Module Outline

- Integer Representations
- Integer Conversion and Casting
- Integer Expansion and Truncating
- Manipulating Integers
- Module Summary

$$B2T(X) = -2^{w-1} \cdot x_{w-1} + \sum_{i=0}^{w-2} 2^i \cdot x_i$$

Integer Representations

Encoding Integers

Unsigned (usual Binary system)

$$B2U(X) = \sum_{i=0}^{w-1} 2^i \cdot x_i$$

$X =$

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	bit position
x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	bit value
2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	weight (unsigned)

$-X = X \,\hat{}\, 11111111_{(2)}$: Xor for Sign conversion ex) X $61_{10} = 00111101_{(2)}$

$$\begin{array}{r} 11111111 \\ 00111101 \\ \hline -61_{10} = 11000010_{(2)} \end{array}$$

- Weight of bit position $i = 2^i$ is multiplied by value x_i

Sign & Magnitude (1's complement)

Represent sign { 0 : positive, 1 : Negative } ⇒ problem : 2 type of Zero, Extra Operation for signing

Encoding Integers

By Neumann Again

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} 2^i \cdot x_i$$

Signed (Two's Complement)

$$B2T(X) = -2^{w-1} \cdot x_{w-1} + \sum_{i=0}^{w-2} 2^i \cdot x_i$$

$$= -1 - \sum_{i=0}^{w-2} 2^i \cdot !x_i$$

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	bit position
x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	bit value
2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	weight (unsigned)
-2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	weight (signed)

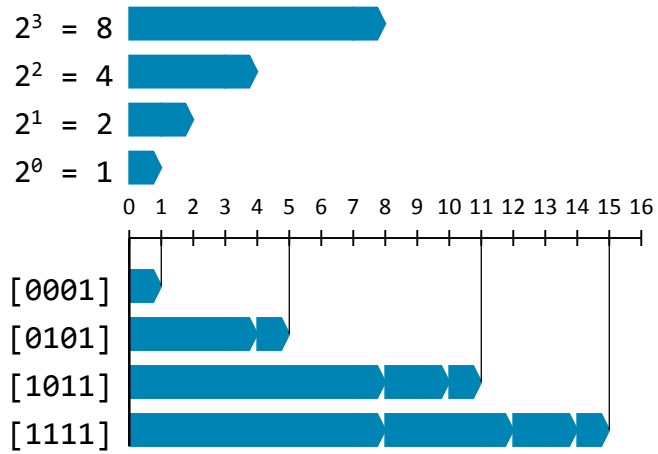
We can use same hardware to calculate sign/unsign number and addition/subtraction either

- Weight of bit position $i = 2^i$ is multiplied by value x_i
- For signed integers, weight of most significant bit is negative: -2^{w-1}
  - most significant bit can be considered to be the "sign bit"

Integer Encoding Visualized

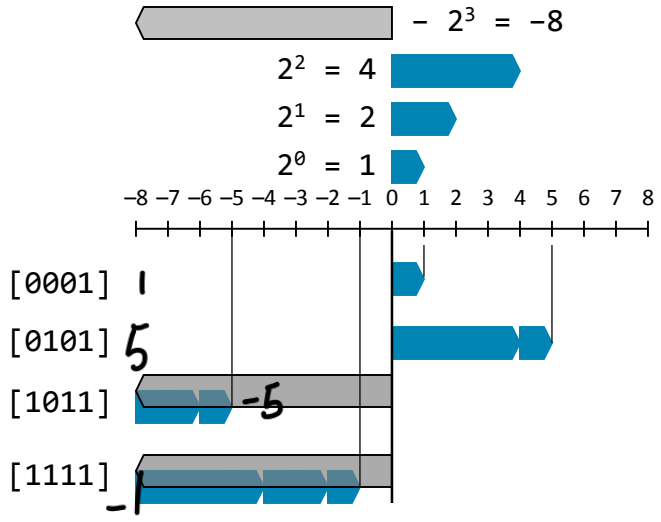
Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} 2^i \cdot x_i$$



Signed (Two's Complement)

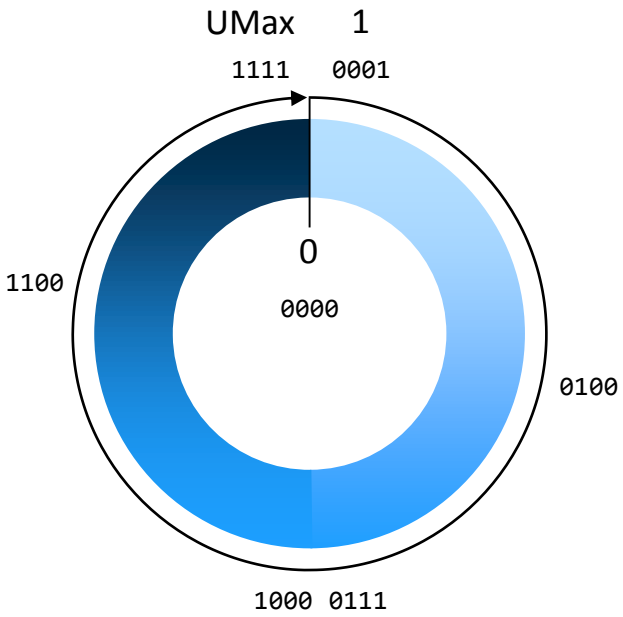
$$B2T(X) = -2^{w-1} \cdot x_{w-1} + \sum_{i=0}^{w-2} 2^i \cdot x_i$$



An (Easier) Way To Look At It

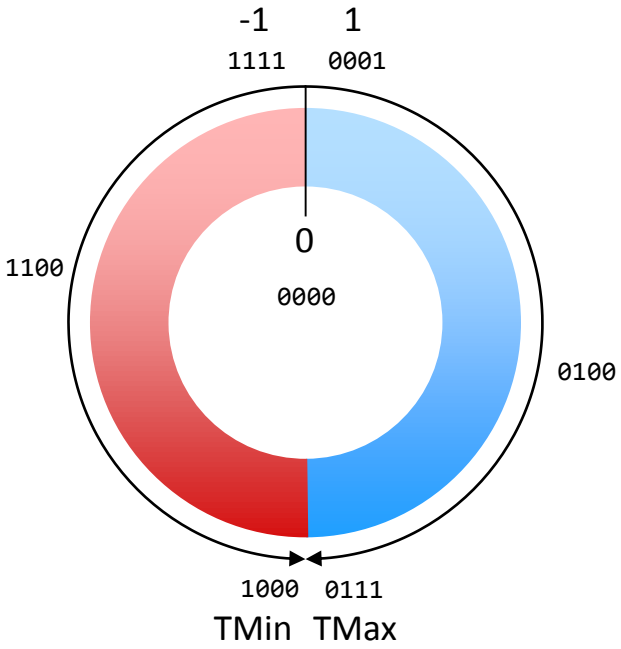
Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} 2^i \cdot x_i$$



Signed (Two's Complement)

$$B2T(X) = -2^{w-1} \cdot x_{w-1} + \sum_{i=0}^{w-2} 2^i \cdot x_i$$



Encoding Example

Declaration	Decimal	Hex	Binary
`short int x =  4190`	4190	10 5e	00010000 01011110
`short int y = -4190`	-4190	ef a2	11101111 10100010

- short int x = 4190

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	0	1	0	1	1	1	1	0
2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}

4190

- short int y = -4190

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	0	1	0	0	0	1	0
-2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}

-4190

Numeric Ranges

- **Unsigned values**
  - UMin = 0
    - 0000…0000
  - UMax = $2^w - 1$
    - 1111…1111
- **Two's complement values**
  - TMin = -2^{w-1}
    - 1000…0000
  - TMax = $2^{w-1} - 1$
    - 0111…1111
- Other values
  - Minus 1
    - 1111…1111
- Values for w=16

Value	Decimal	Hex	Binary
UMax	65535	ff ff	11111111 11111111
0 (=UMin)	0	00 00	00000000 00000000
TMax	32767	7f ff	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	ff ff	11111111 11111111

Values for Different Word Sizes

	w			
	8	**16**	**32**	**64**
Umax	255	65,536	4,294,967,295	18,446,744,073,709,551,615
Tmax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

- Observations
  - |TMin | = TMax + 1
    - Asymmetric range
  - UMax = 2 * TMax + 1

- C Programming
  - `#include <limits.h>`
  - Declares constants, e.g.,
    - `ULONG_MAX`
    - `LONG_MAX`
    - `LONG_MIN`
  - Values platform specific

Unsigned & Signed Numeric Values

x	B2U(x)	B2T(x)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

- Equivalence
  - Same encodings for nonnegative values
- Uniqueness
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding
- $\Rightarrow$ Can invert mappings
  - U2B(x) = $B2U^{-1}(x)$
    - Bit pattern for unsigned integer
  - T2B(x) = $B2T^{-1}(x)$
    - Bit pattern for two's comp integer

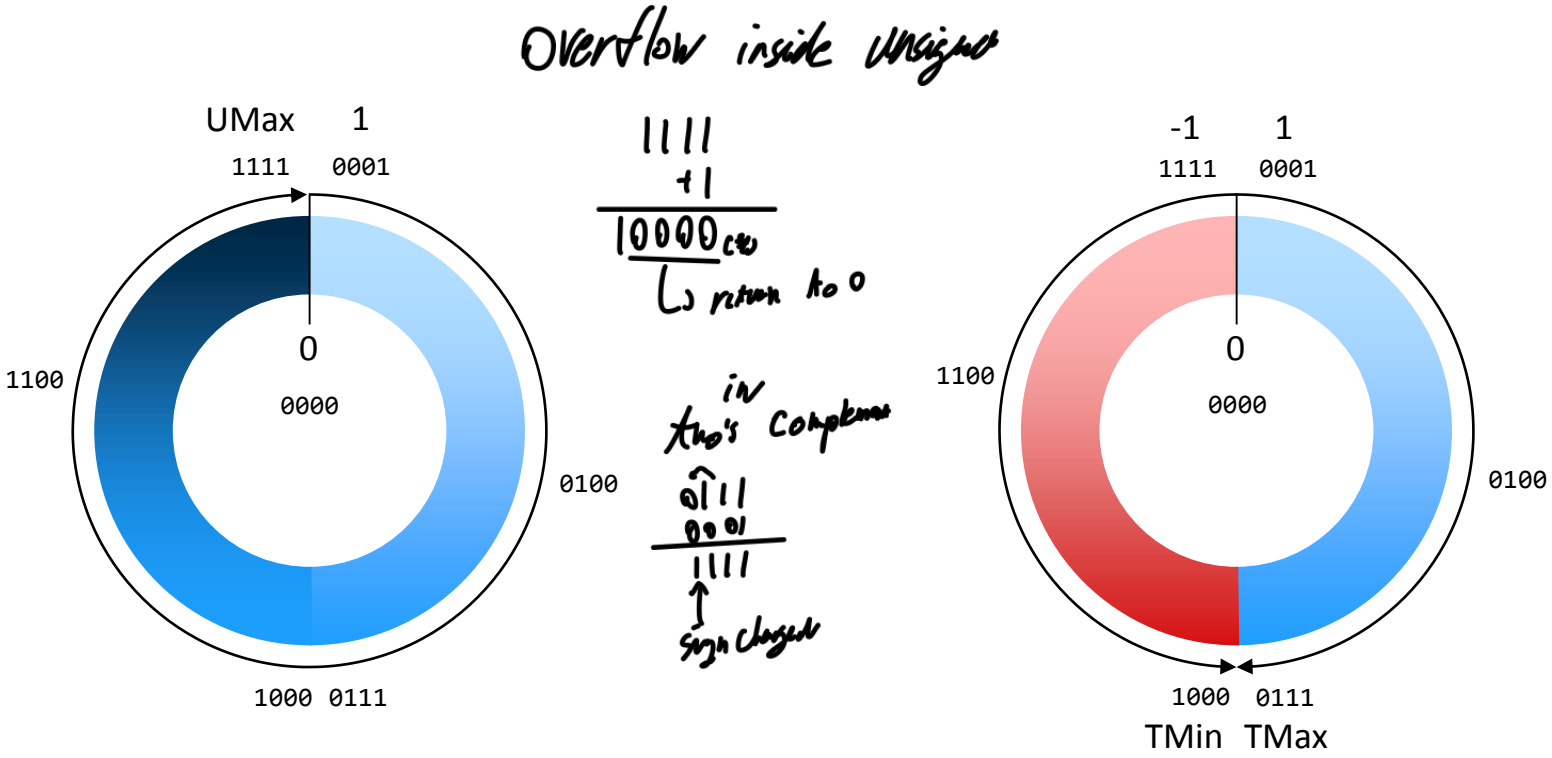
An (Easier) Way To Look At It

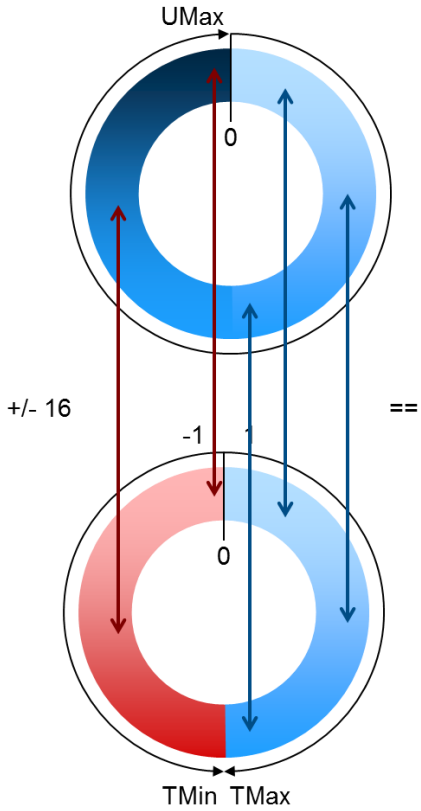
Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} 2^i \cdot x_i$$

Signed (Two's Complement)

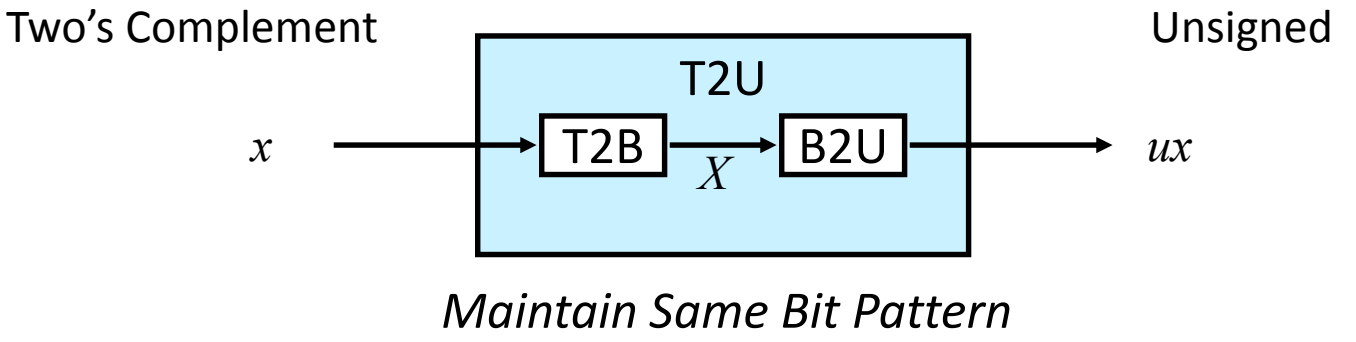
$$B2T(X) = -2^{w-1} \cdot x_{w-1} + \sum_{i=0}^{w-2} 2^i \cdot x_i$$





Integer Conversion and Casting

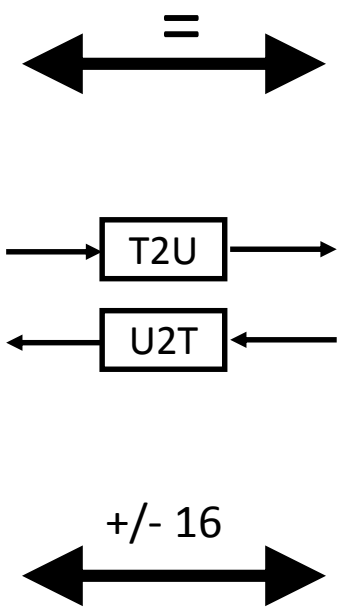
Mapping Between Signed & Unsigned



- **Mappings between unsigned and two's complement numbers:**
  Keep bit representations and reinterpret
  (Remember: Information = Bits + Context!)

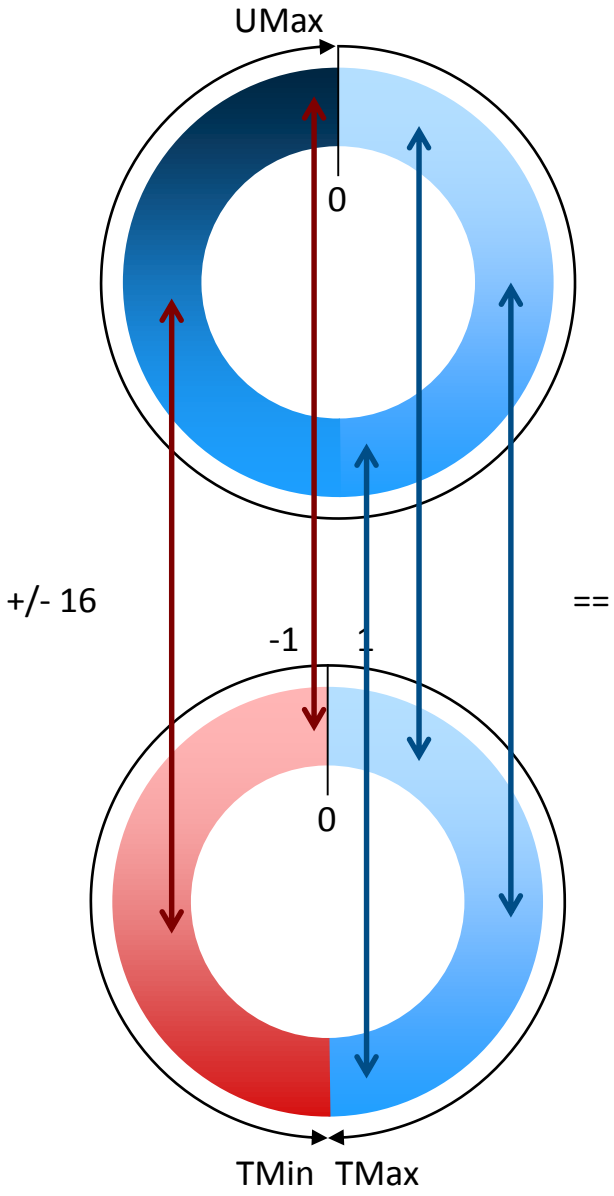
Mapping Signed ↔ Unsigned

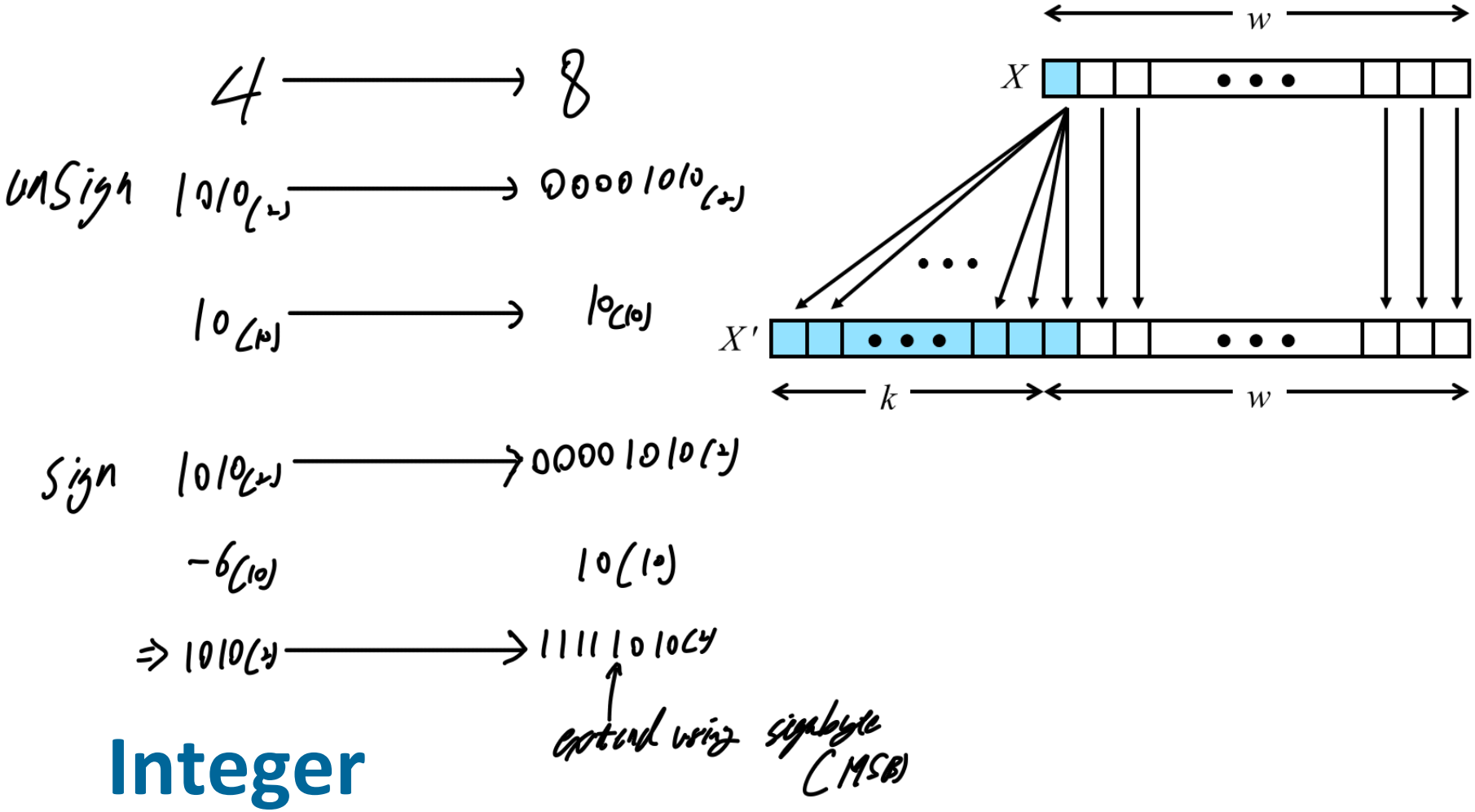
Bits	Signed	Unsigned
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15



Conversion Visualized

- **2's Comp. → Unsigned**
  - Ordering Inversion
  - Negative → Big Positive
- **Unsigned → 2's Comp.**
  - Ordering Inversion
  - Big Positive → Negative : OK
- **Conversion**
  - Number <= TMax: Unsigned == 2's Comp
  - Otherwise: Unsigned = 2's Comp + 2^w





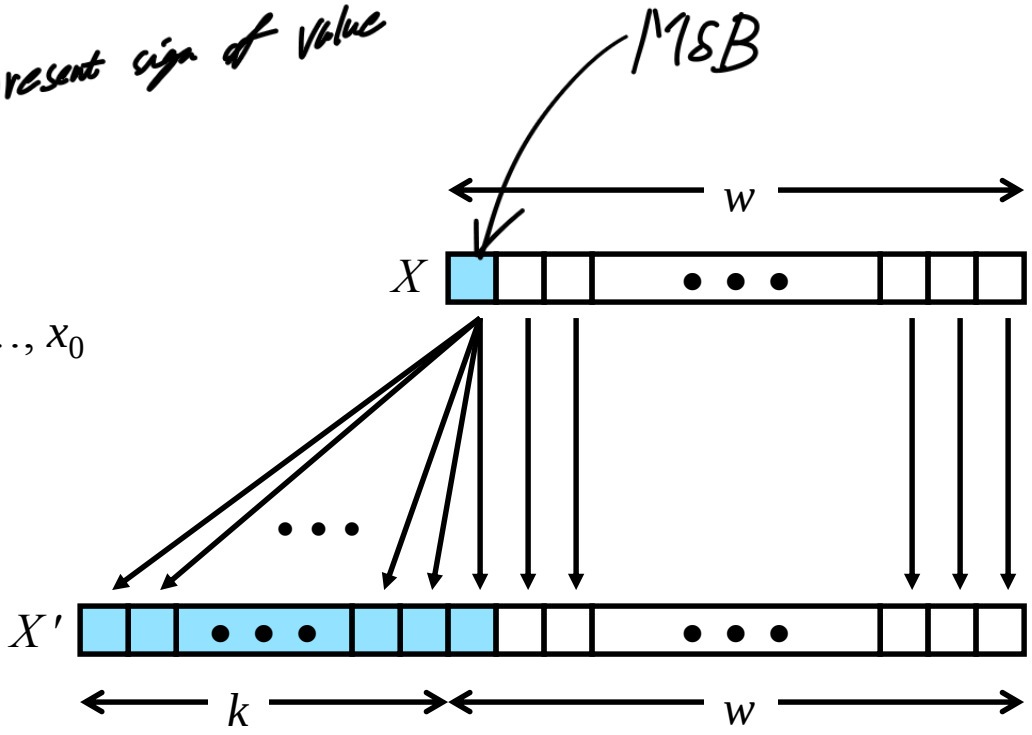
Integer Expansion and Truncating

Expansion (Sign Extension)

- Task:
  - Expand *w*-bit integer *x* to *w*+*k*-bit integer

- **Zero expand** (does not preserve value for negative signed values)
  - Add k 0s in front of MSB

- **Sign expand**: preserves value
  - Make *k* copies of MSB:
  - $X = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$
  - Works for both unsigned and signed (2's complement) values



Truncating

- Shrink w bits to w-k bits
- Drop k MSB

Sign Extension Example

```
short int x =   7375;
int      ix = (int)x;
short int y =  -7375;
int      iy = (int)y;
```

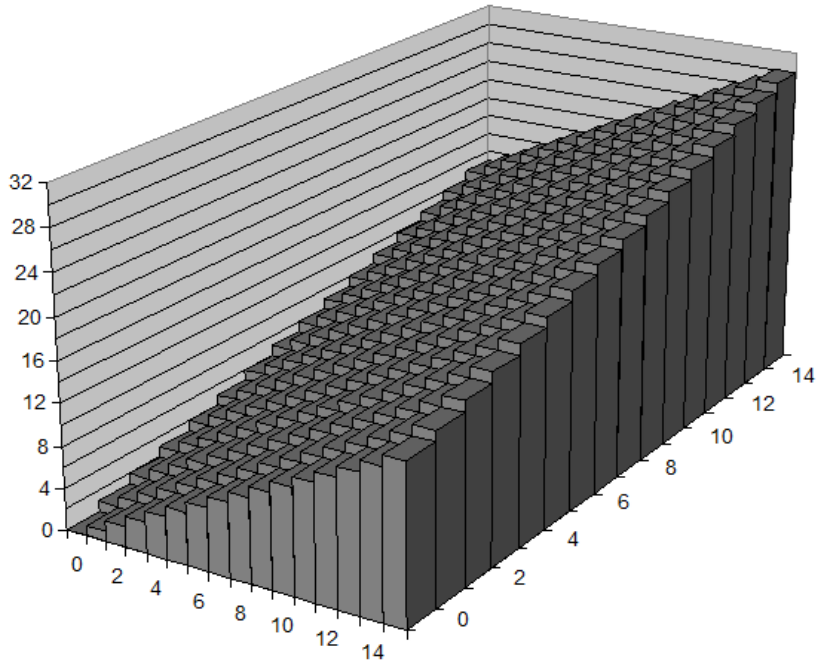
	Decimal	Hex	Binary
x	7375	1c cf	0001 1100 1100 1111
ix	7375	00 00 1c cf	0000 0000 0000 0000 0001 1100 1100 1111
y	-7375	e3 31	1110 0011 0011 0001
iy	-7375	ff ff e3 31	1111 1111 1111 1111 1110 0011 0011 0001

- Converting from smaller to larger integer data type
- C automatically performs sign extension

Zero Extension Example

```
short int x =   7375;
int       ix = (int)(unsigned short)x;
short int y =  -7375;
int       iy = (int)(unsigned short)y;
```

	Decimal	Hex	Binary
x	7375	1c cf	0001 1100 1100 1111
ix	7375	00 00 1c cf	0000 0000 0000 0000 0001 1100 1100 1111
y	-7375	e3 31	1110 0011 0011 0001
iy	58161	00 00 e3 31	0000 0000 0000 0000 1110 0011 0011 0001



Manipulating Integers

Bit-Level Operations in C

- **C: bit-wise AND, OR, NOT, XOR: &, |, ~, ^**
  - Applicable to any “integral” data type
    - (unsigned) char, short, int, long, long long
  - Arguments viewed as bit vectors, operator applied bit-wise

- **Examples (char data type)**

Hex			Binary		
~0x41	=	0xbe	$\sim 01000001_2$	=	10111110_2
~0x00	=	0xff	$\sim 00000000_2$	=	11111111_2
0x64 & 0x55	=	0x41	01101001_2 & 01010101_2	=	01000001_2
0x64 | 0x55	=	0x7d	01101001_2 | 01010101_2	=	01111101_2
0x64 ^ 0x55	=	0x3c	01101001_2 ^ 01010101_2	=	00111100_2

not ≠ Sign flip

Logic Operations in C

- **Contrast to Bit-level Operators**
  - `&&`, `||`, `!`
    - View 0 as "False", *anything nonzero* as "True" (not just 1!)
    - Always return 0 or 1
    - Early termination

- Examples (char data type)

Hex			Remarks
!0x41	=	0x00	
!0x00	=	0x01	
0x64 && 0x55	=	0x01	
0x64 || 0x55	=	0x01	early termination 0x55 not looked at
if (p && *p) …			if ((p != NULL) && (*p != 0)) …

Shift Operations

- **Left shift:** `x << y` = $x * 2^y$
  - Shift bit-vector x left y positions
    - Throw away extra bits on left
    - Fill with 0's on right
- **Right shift:** `x >> y` = $x / 2^y$
  - Shift bit-vector x right y positions
    - Throw away extra bits on right
  - *Logical* shift
    - Fill with 0's on left
  - *Arithmetic* shift
    - Replicate most significant bit on right
- Undefined behavior
  - Shift amount < 0 or ≥ word size

Sign

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

always fill Zero

Argument x	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 2	11101000

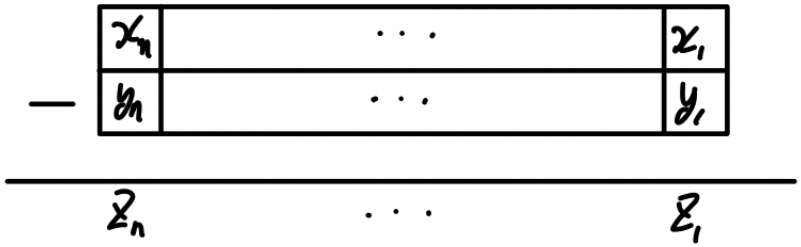
fill Zero

fill MSB

Negation: Complement & Increment

- Claim: the following holds for 2's complement

  -x = ~x + 1



- Complement
  - Observation: **~x + x == 1111…111 == -1**

$x - y = x + (\sim y + 1)$

	x	1	0	0	1	1	1	0	1
+	~x	0	1	1	0	0	0	1	0
	-1	1	1	1	1	1	1	1	1

Complement & Increment Examples

- **Negate x = 7375**

	Decimal	Hex	Binary
x	7375	1c cf	0001 1100 1100 1111
~x	-7376	e3 30	1110 0011 0011 0000
~x+1	-7375	e3 31	1110 0011 0011 0001
y	-7375	e3 31	1110 0011 0011 0001

- **Negate x = 0**

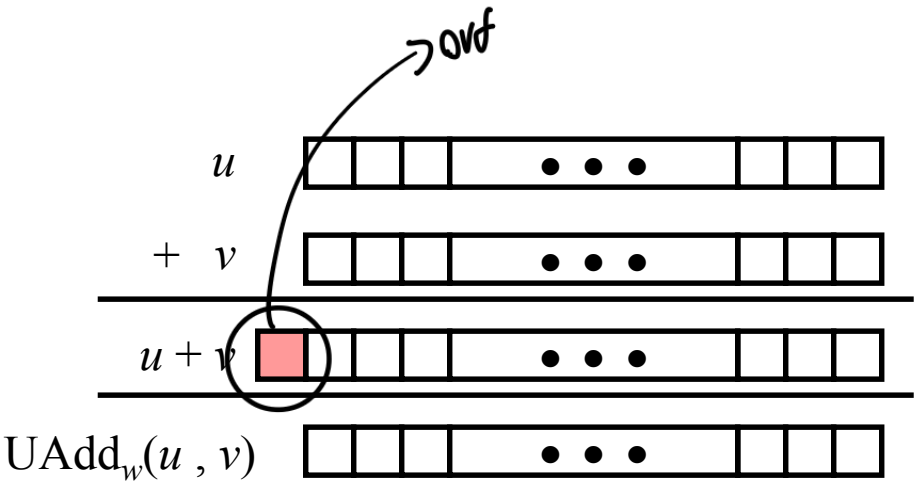
	Decimal	Hex	Binary
x	0	00 00	0000 0000 0000 0000
~x	-1	ff ff	1111 1111 1111 1111
~x+1	0	00 00	0000 0000 0000 0000

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



- Standard addition function
  - Ignores carry output
  - Implements modular arithmetic

    $s = \text{UAdd}_w(u , v) = u + v \pmod{2^w}$

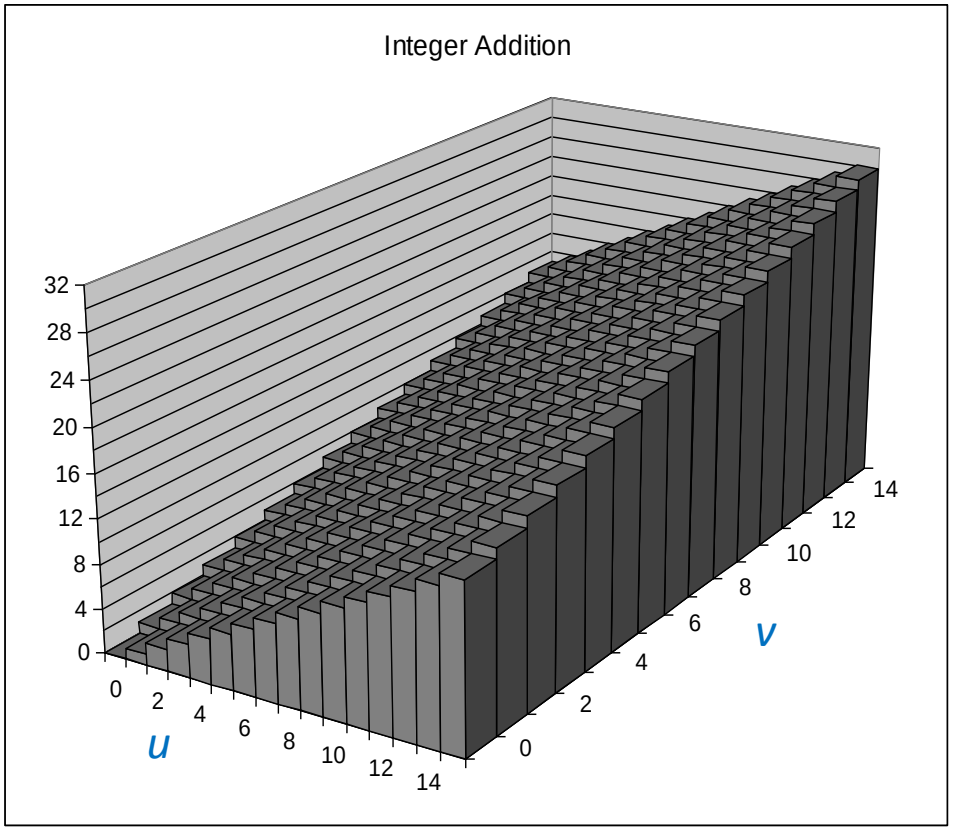
due to ovf

$$UAdd_w(u,v) = \begin{cases} u + v & u + v < 2^w \\ u + v - 2^w & u + v \geq 2^w \end{cases}$$

Visualizing (Mathematical) Integer Addition

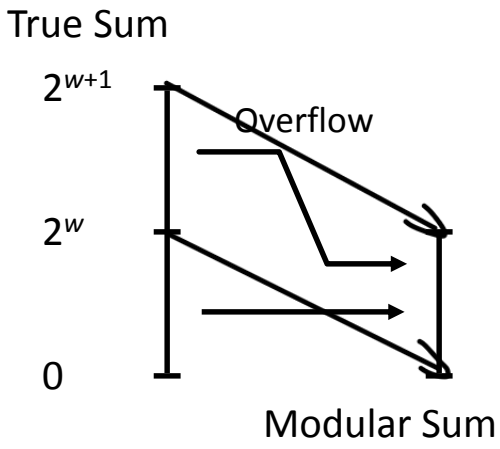
- **Integer addition**
  - 4-bit integers u, v
  - Compute true sum Add4(u , v)
  - Values increase linearly with u and v
  - Forms planar surface

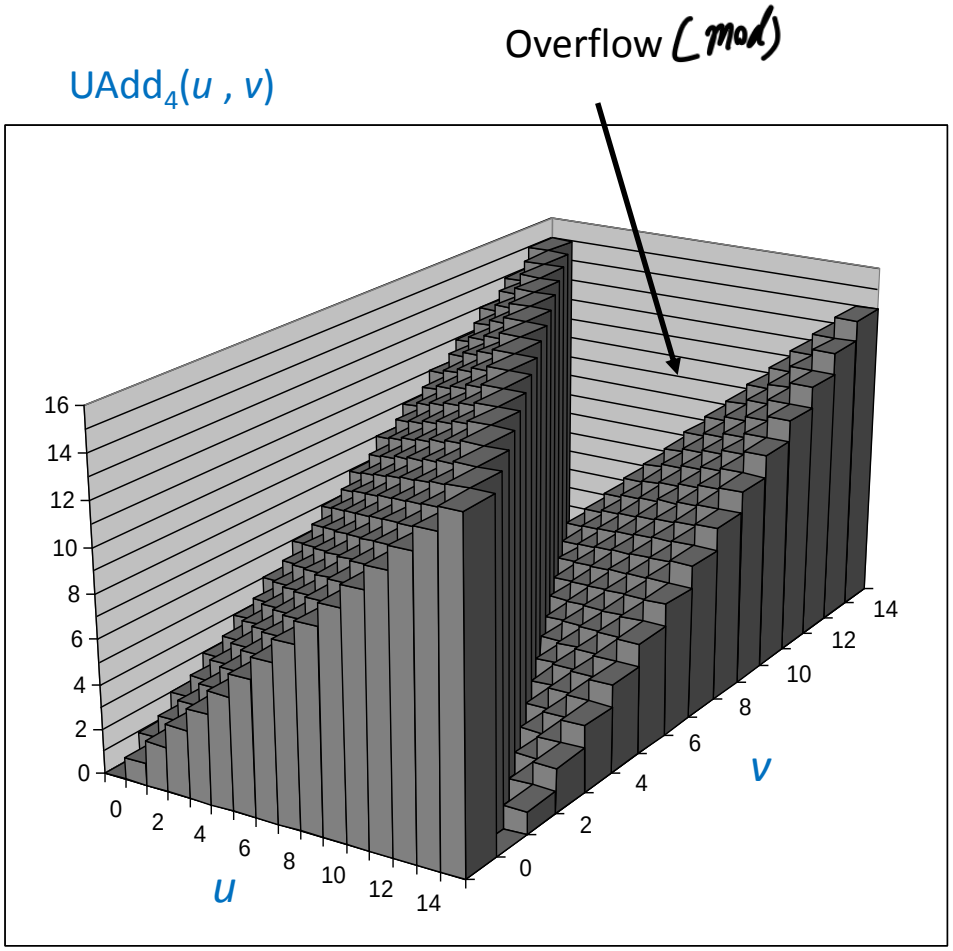
$Add_4(u , v)$



Visualizing Unsigned Addition

- Wraps around
  - If true sum ≥ 2^w
  - At most once



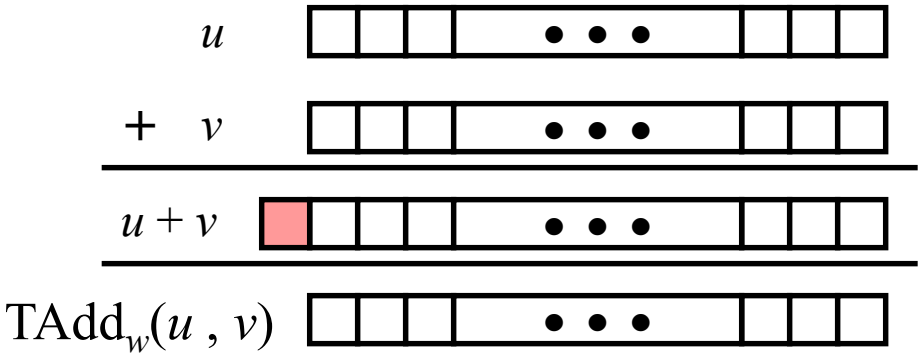


Two's Complement Addition : Same as Unsigned Number

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



- **TAdd and UAdd have identical bit-level behavior**
  - Signed vs. unsigned addition in C:

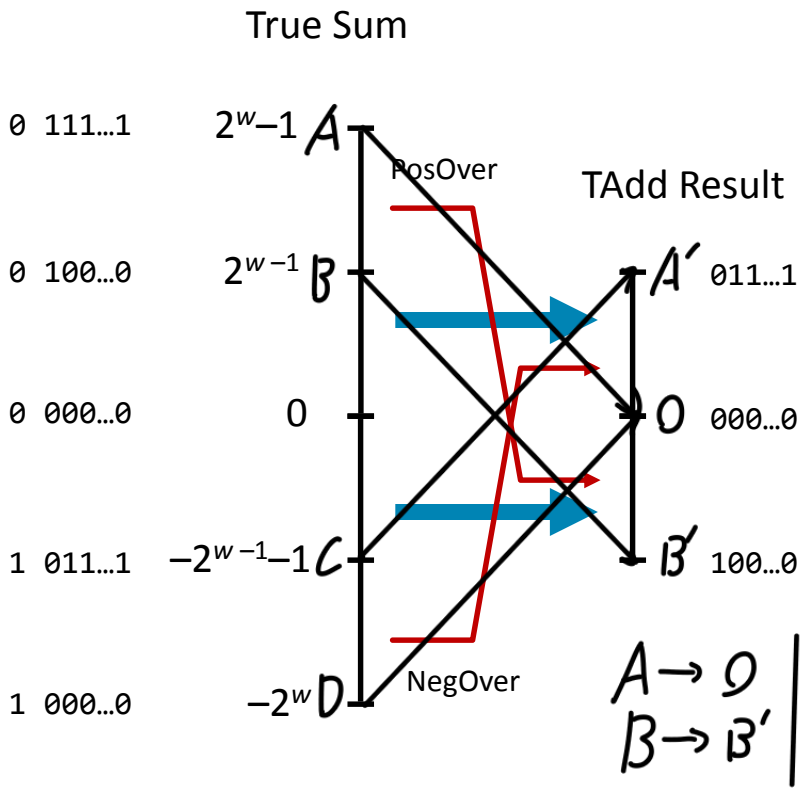
    int s, t, u, v;

    s = (int) ((unsigned) u + (unsigned) v);

    t = u + v
  - Will give s == t

Visualizing 2's Complement Addition

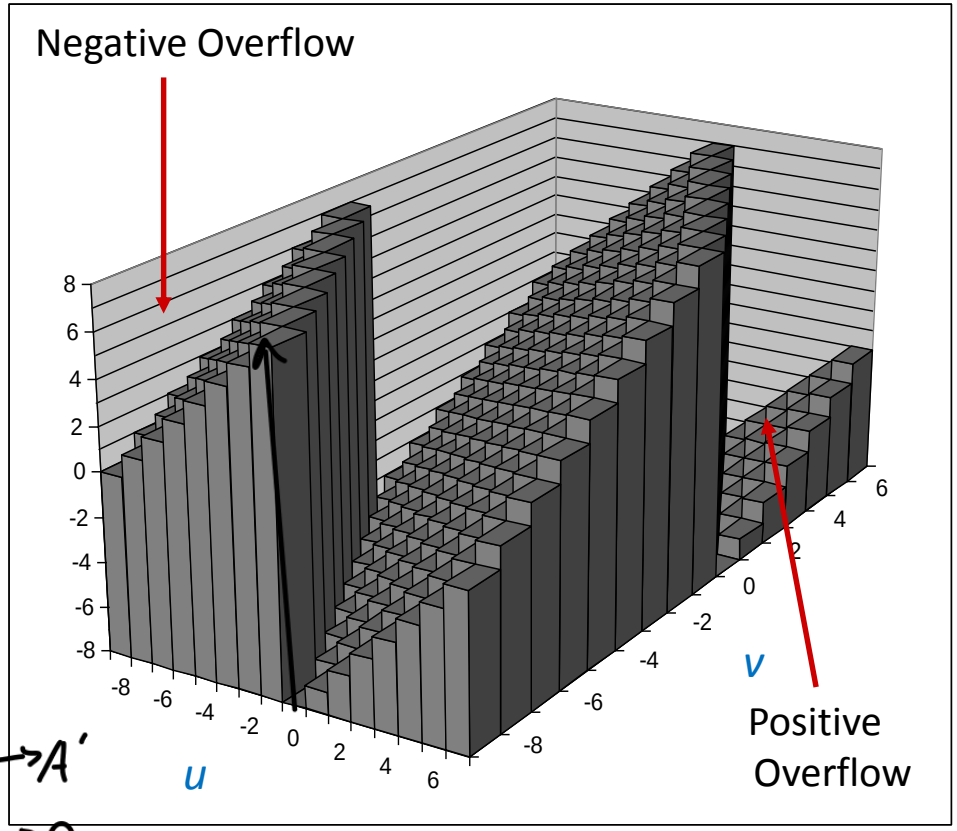
- Wrap around
  - Becomes negative if sum $\geq 2^{w-1}$ (at most once)
  - Becomes positive if sum $< -2^{w-1}$ (at most once)



$A \to 0 \quad C \to A'$

$B \to B' \quad D \to 0$

$TAdd_4(u, v)$

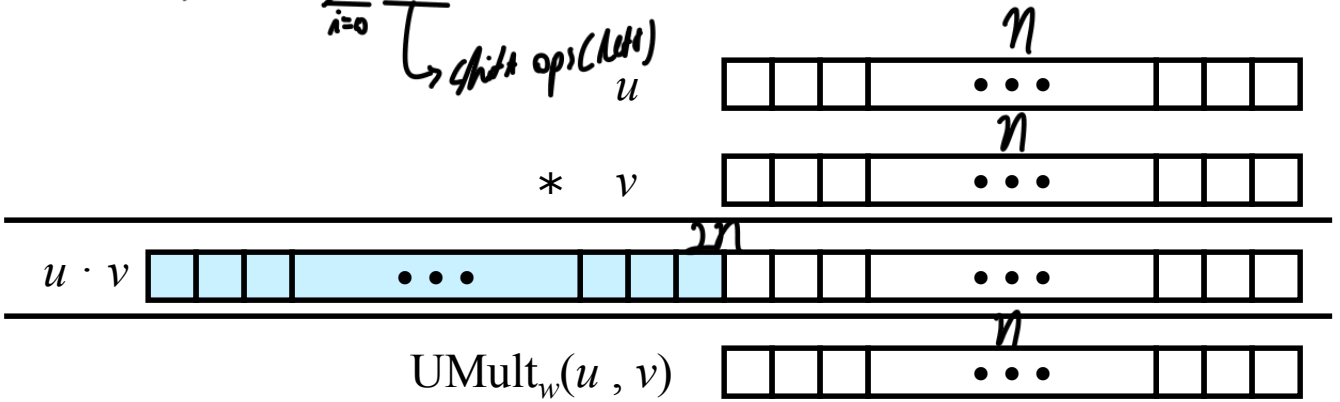


Unsigned Multiplication

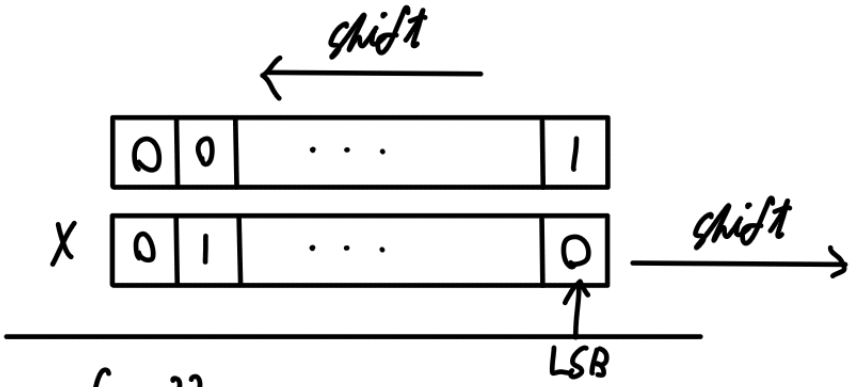
Operands: *w* bits

True Product: 2*w* bits

Discard upper *w* bits

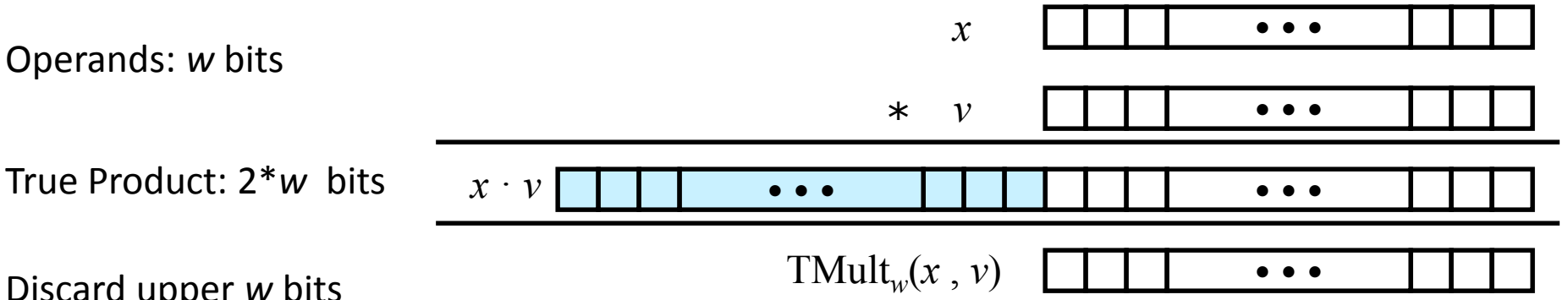


- Standard multiplication function
  - Ignores high order w bits
  - Implements modular arithmetic
    - $\text{UMult}_w(u , v) = u \cdot v \mod 2^w$




```
for 32
  if(LSB)
    sum += target
  target<<1 : left
  sum >>1 : right
```

Signed Multiplication



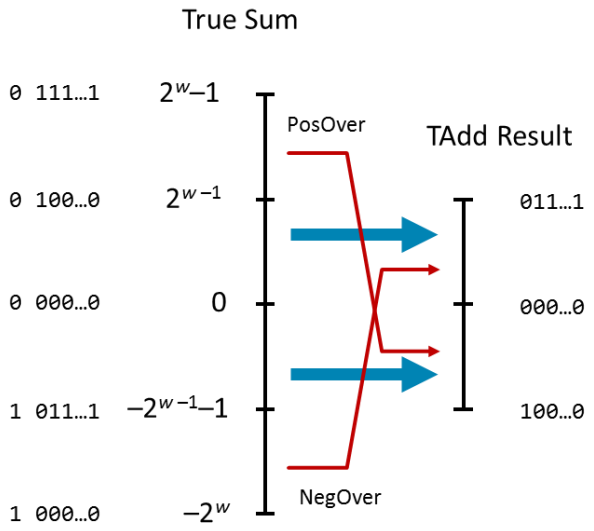
- Standard multiplication function
  - Ignores high order w bits
  - Some of which are different for signed vs. unsigned multiplication
  - Lower bits are the same

Multiplication: Signed vs. Unsigned

- Signed multiplication in C
  - Ignores high order ***w*** bits
  - The low-order ***w*** bits are identical to unsigned multiplication

Examples for 3-bit integer

Mode	x	y	x · y	Truncated x · y
Unsigned	5 [101]	3 [011]	15 [001111]	7 [111]
Two's comp.	-3 [101]	3 [011]	-9 [110111]	-1 [111]
Unsigned	4 [100]	7 [111]	28 [011100]	4 [100]
Two's comp.	-4 [100]	-1 [111]	4 [000100]	-4 [100]
Unsigned	3 [011]	3 [011]	9 [001001]	1 [001]
Two's comp.	3 [011]	3 [011]	9 [001001]	1 [001]



Module Summary

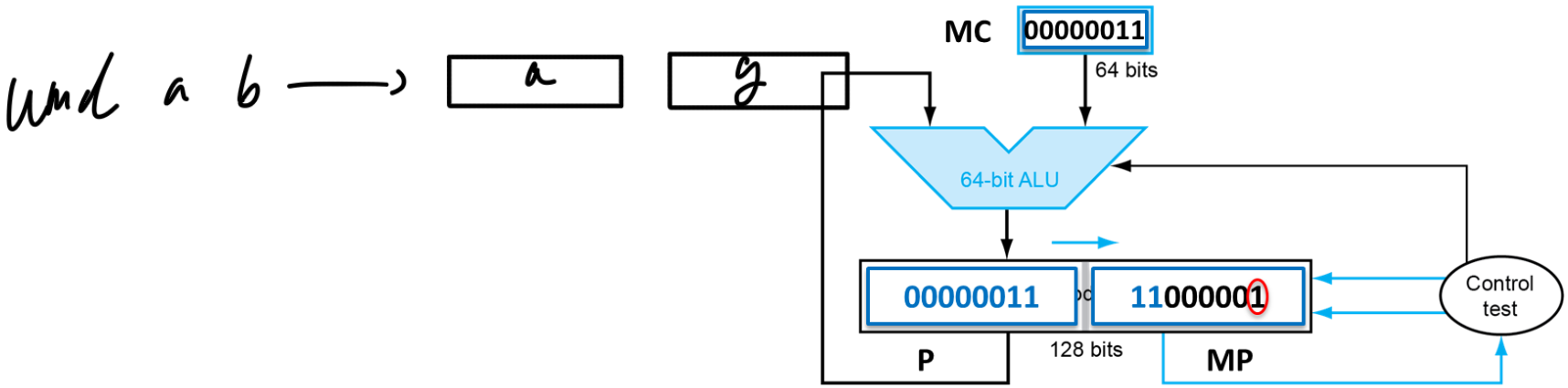
Module Summary

- **Integers encoded as unsigned and signed**
  - Two's complement encoding used for signed integers
    - -x = ~x + 1
  - Conversion changes only the context, but not the bits

- **Operations**
  - Expansion, shift right of signed integers: keep sign bit to avoid surprises
  - Distinction between bit-level and logical operators

- **Single-precision Addition/Subtraction/Multiplication**
  - identical bit-level operations for signed/unsigned in
  - overflow can occur

- **Full-precision Multiplication/Division**
  - different operations for signed/unsigned data types

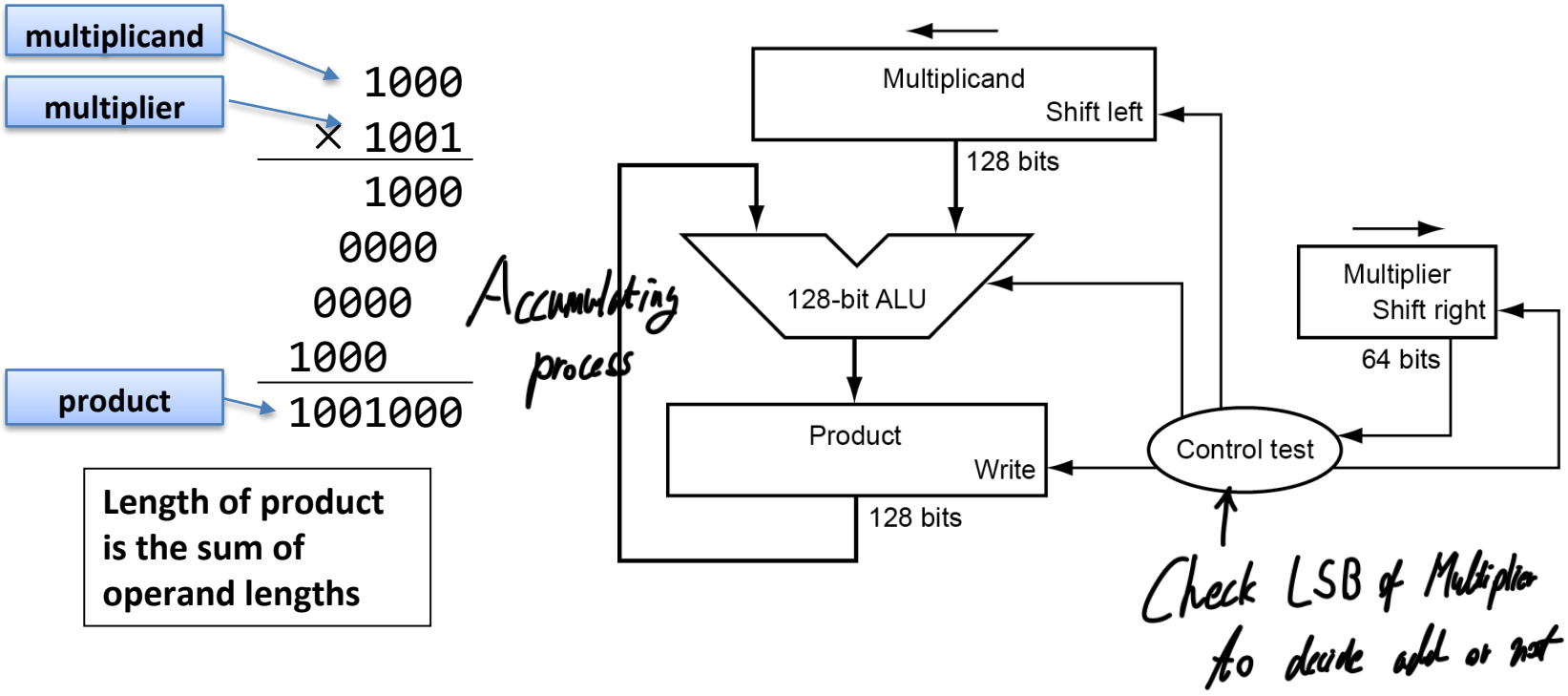


Appendix

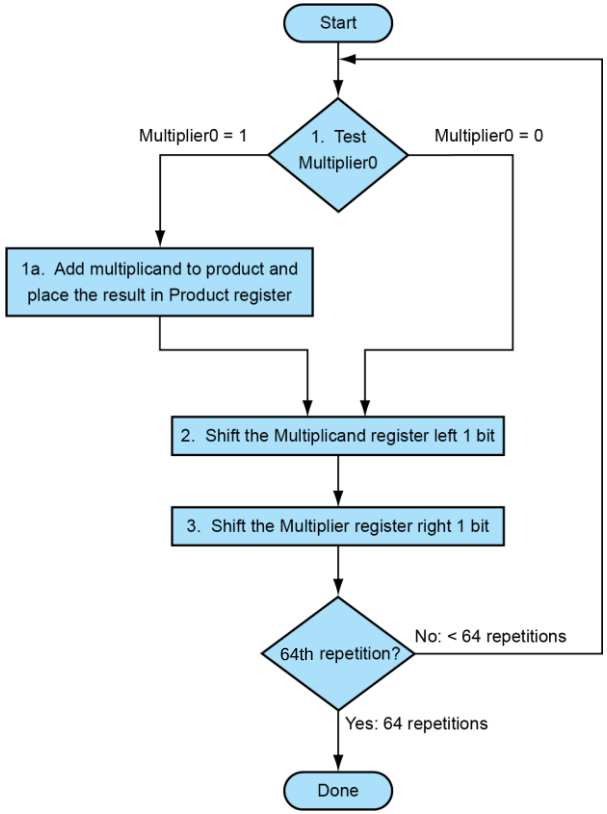
Multiplication and Division in Hardware

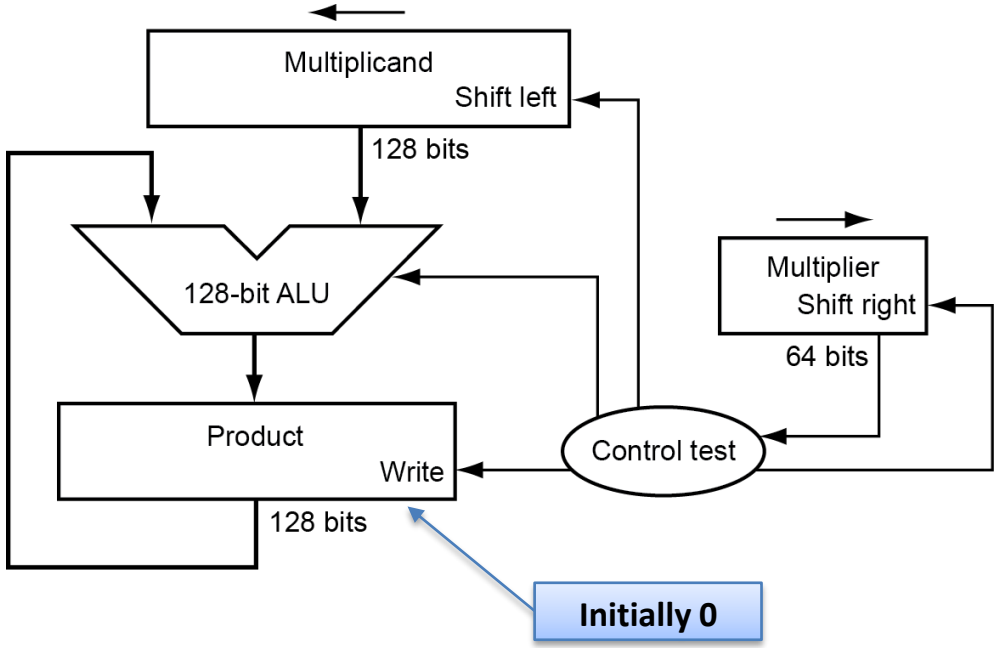
Multiplication

- Long-multiplication approach



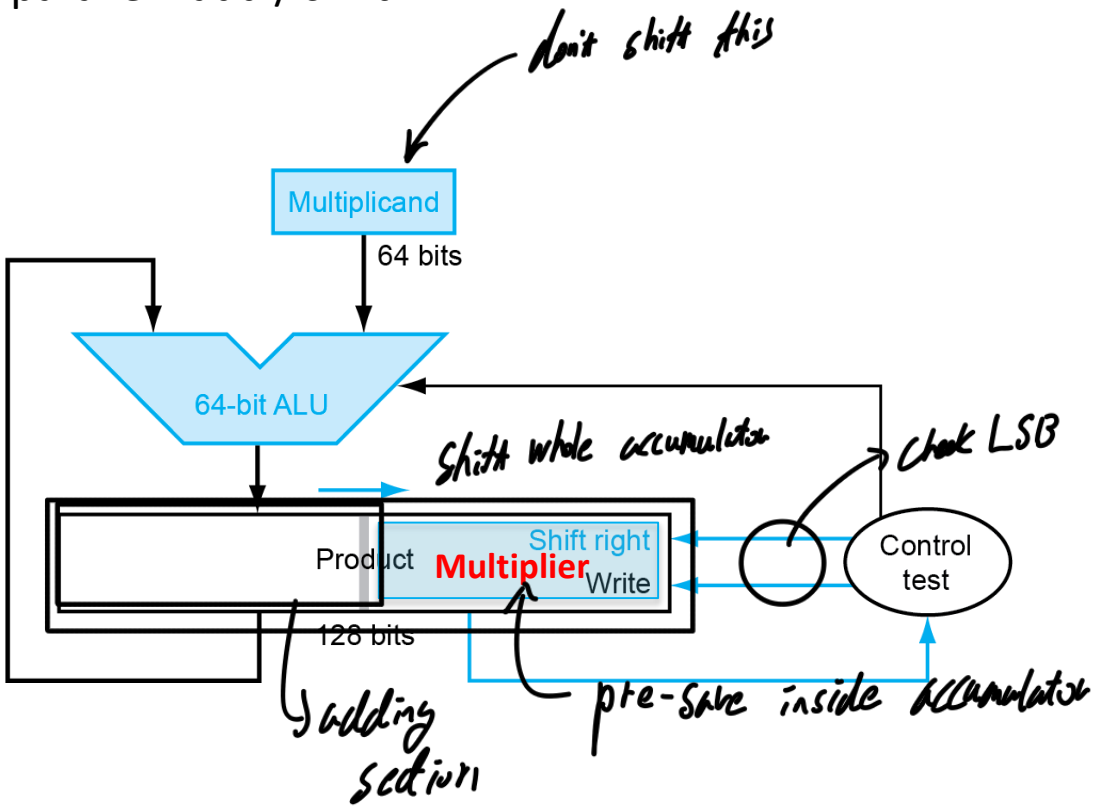
Multiplication Hardware





Optimized Multiplier

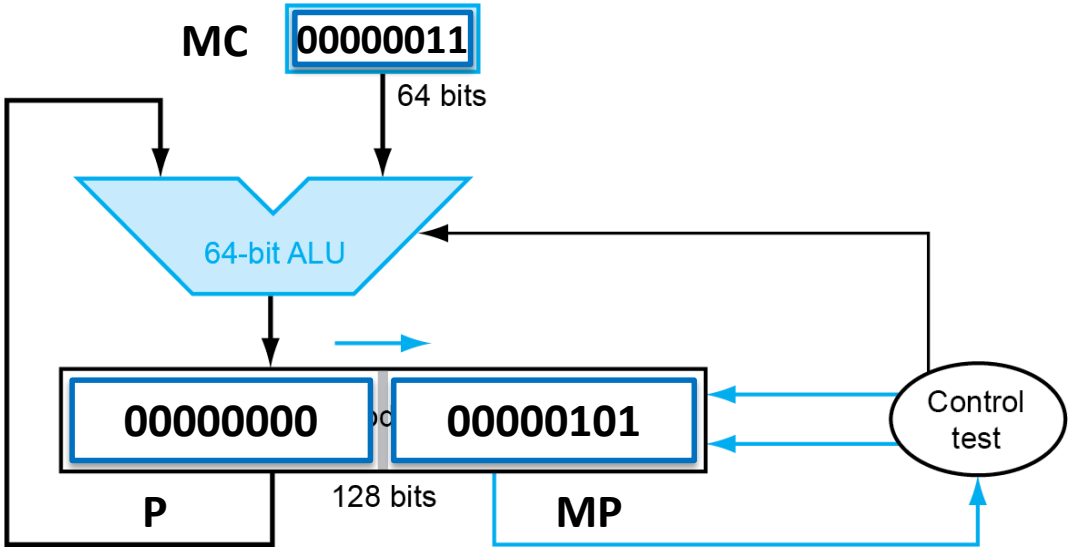
- Perform steps in parallel: add / shift



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

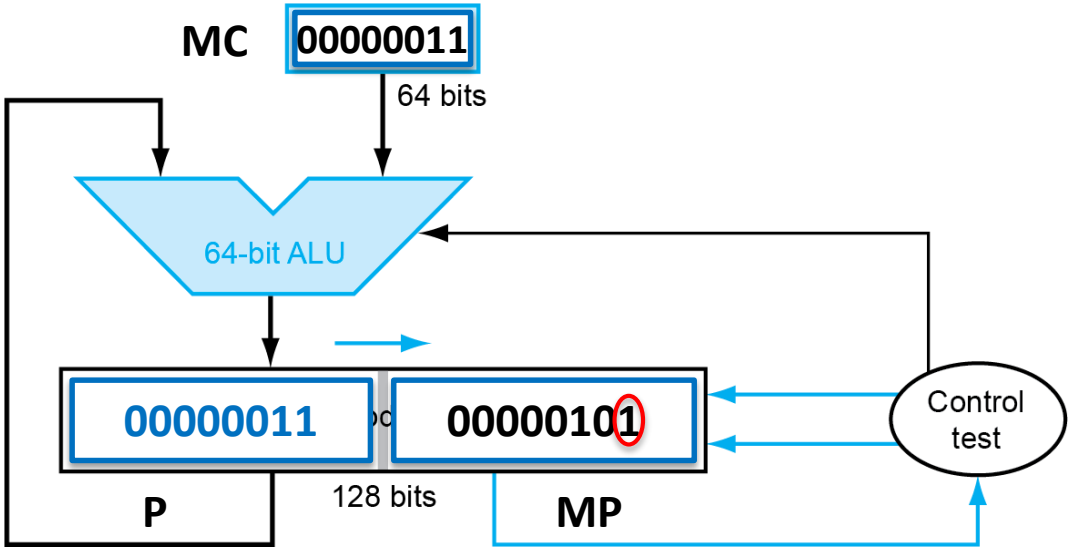
Multiplication Example (1)

- $3_{10} \times 5_{10} = 00000011_2 \times 00000101_2$



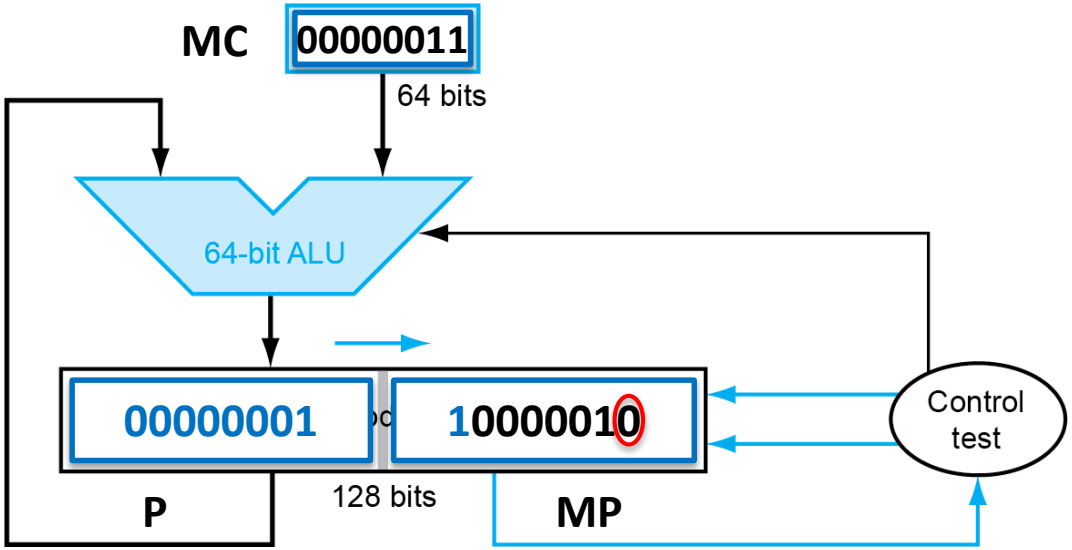
Multiplication Example (2)

- MP_0 = 1: Add MC to P, shift right P and MP by 1 bit



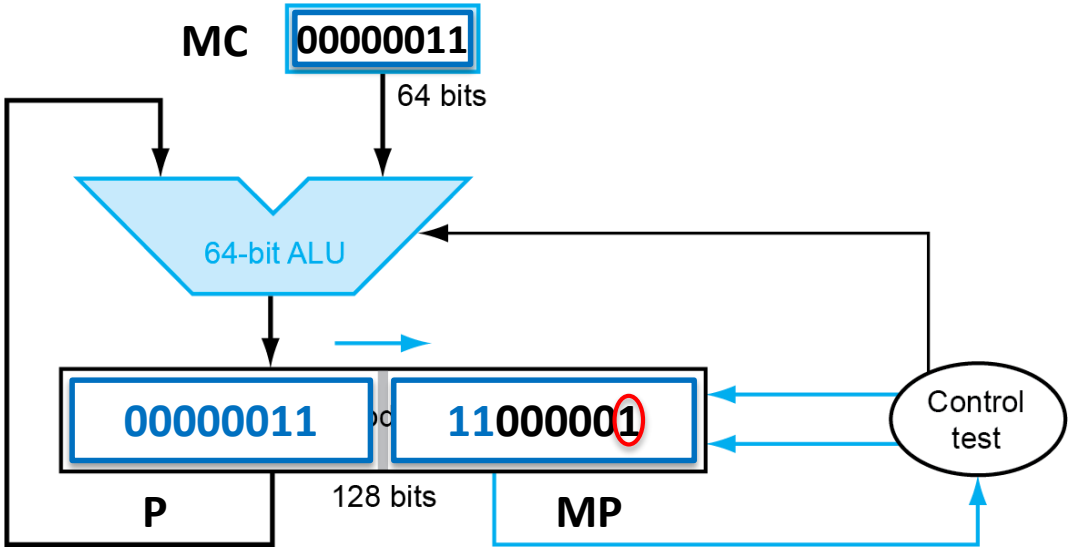
Multiplication Example (3)

- $MP_0 = 0$: Shift right P and MP by 1 bit



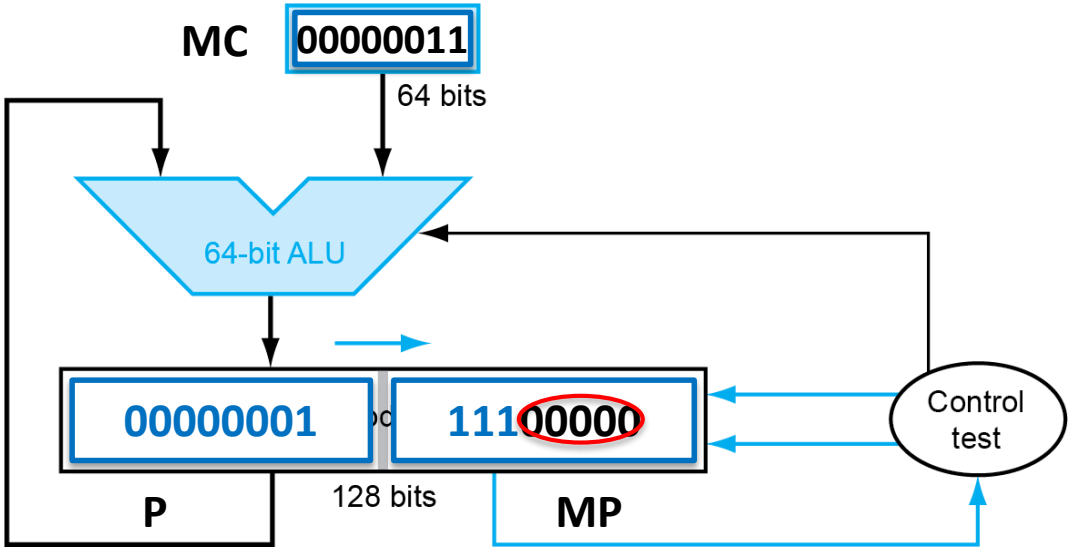
Multiplication Example (4)

- MP_0 = 1: Add MC to P, shift right P and MP by 1 bit



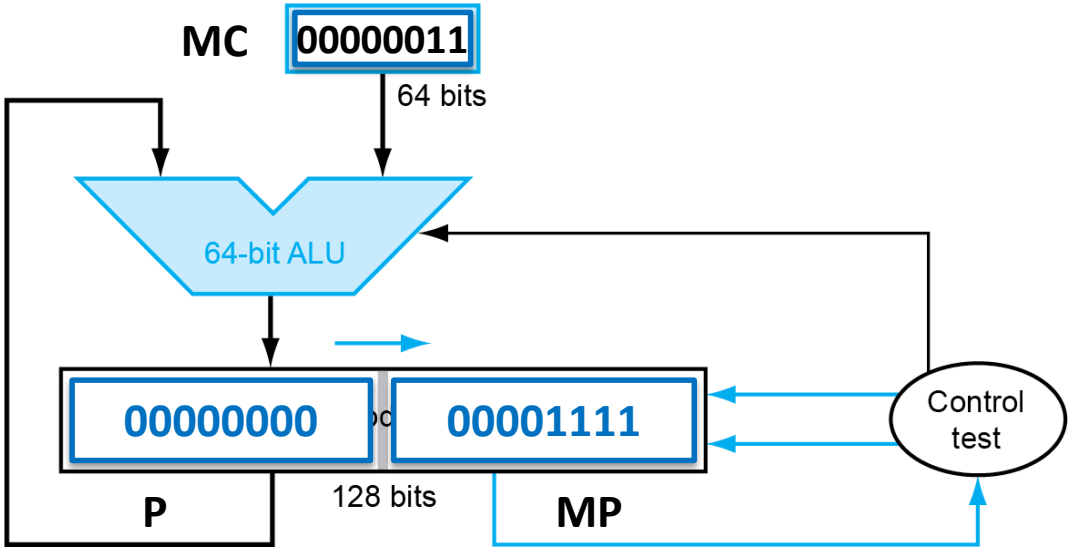
Multiplication Example (5)

- MP_0 = 0 (5 times): Shift right P and MP by 1 bit (5 times)



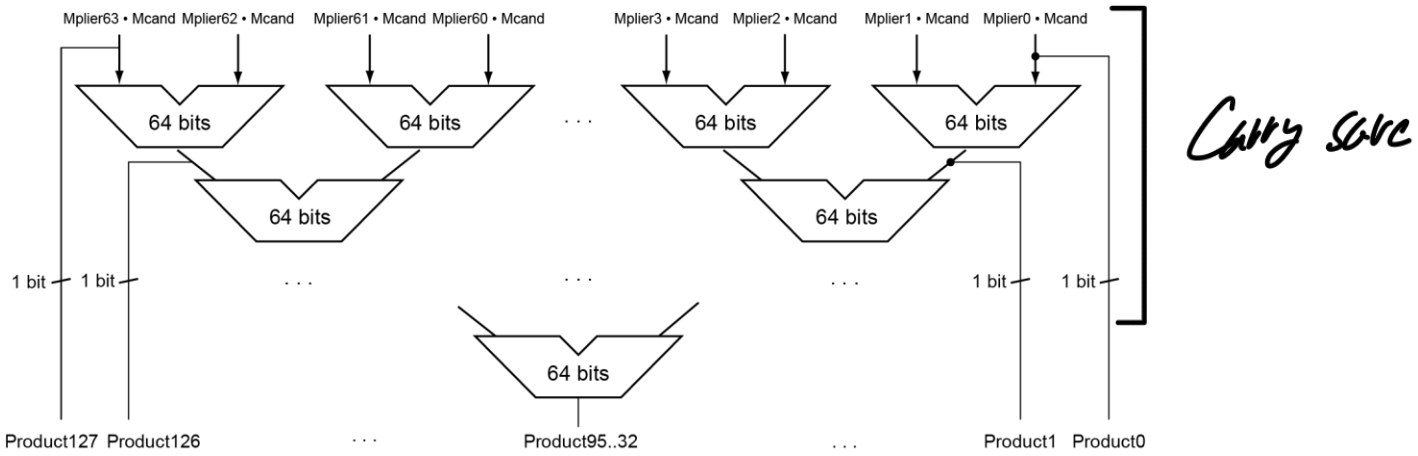
Multiplication Example (6)

- Final product available in P + MP



Faster Multiplier

- "Adder Tree": use multiple adders
  - Cost/performance tradeoff



- Can be pipelined
  - Several multiplication performed in parallel

Division

- Anatomy of a division

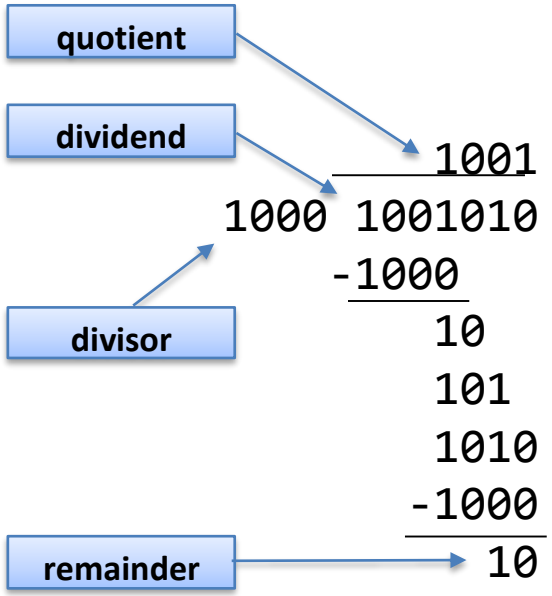
$$\frac{12}{5} = 2,\ rem.2$$

$$\frac{dividend}{divisor} = quotient, remainder$$

Division

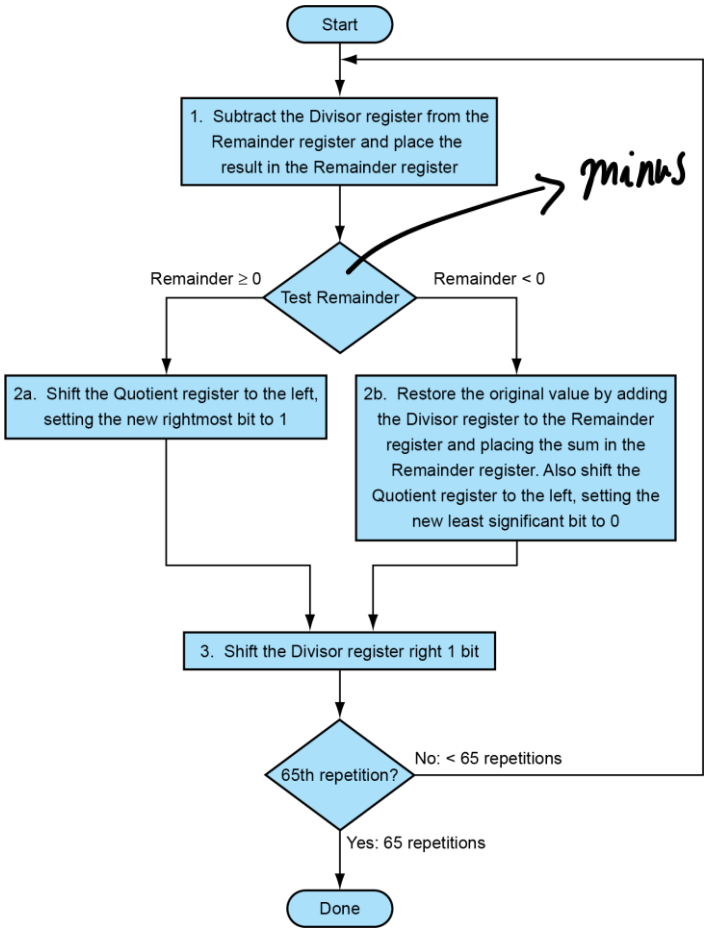
- Check for 0 divisor

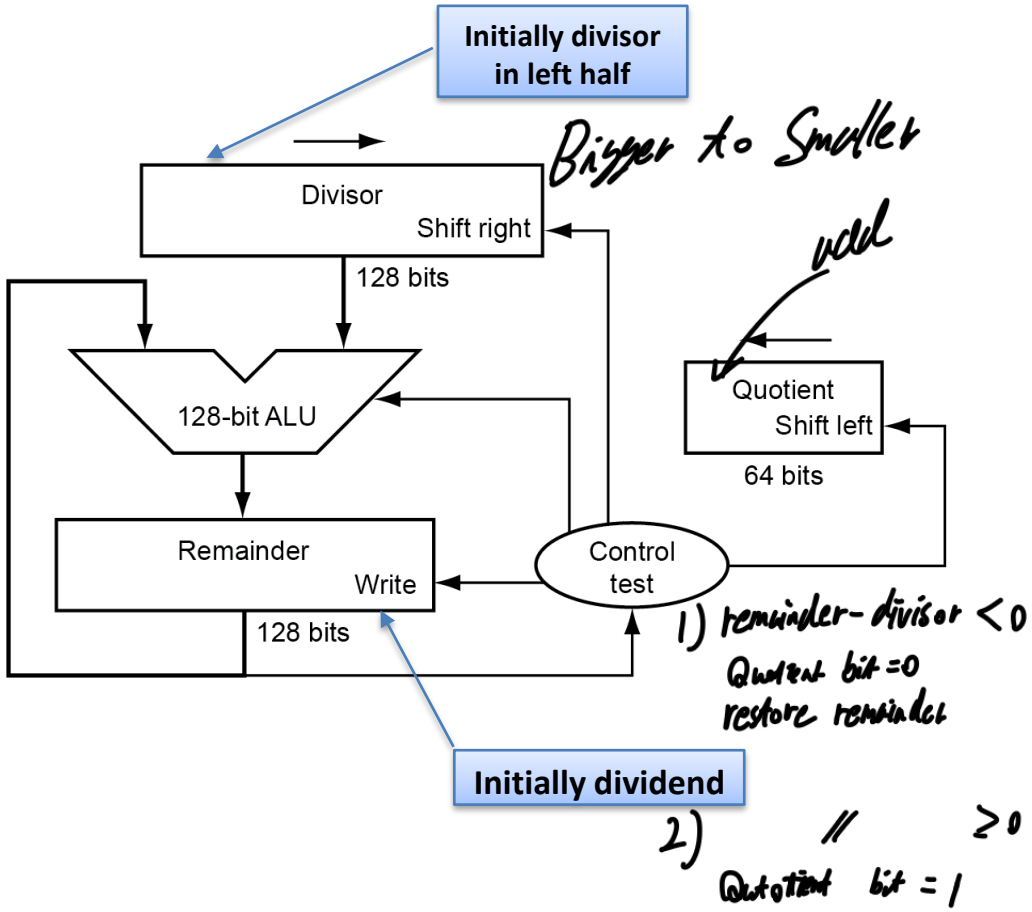
- If divisor ≤ dividend bits:
  1 bit in quotient, subtract
- Otherwise: 0 bit in quotient,
  bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back



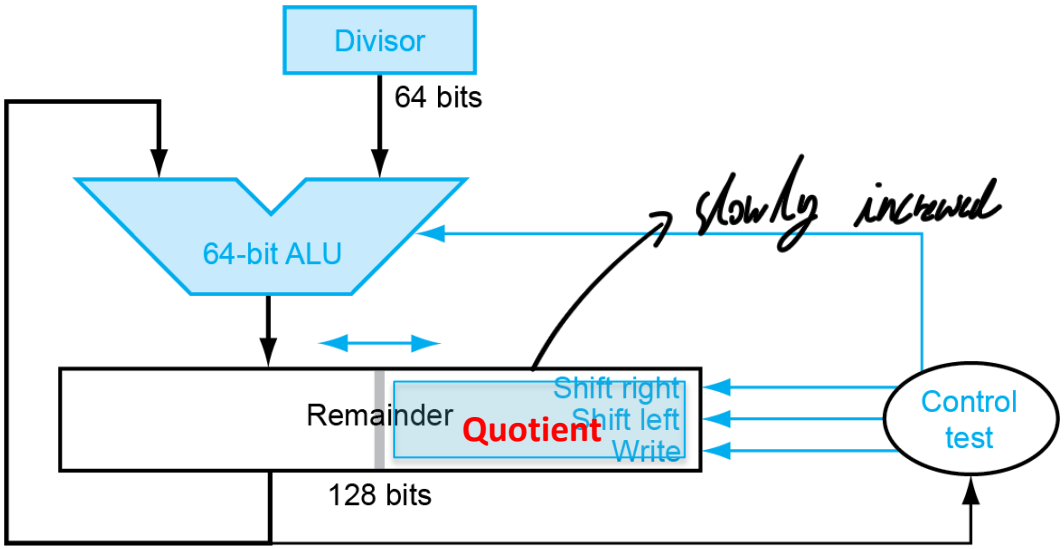
***n*-bit operands yield *n*-bit quotient and remainder**

Division Hardware





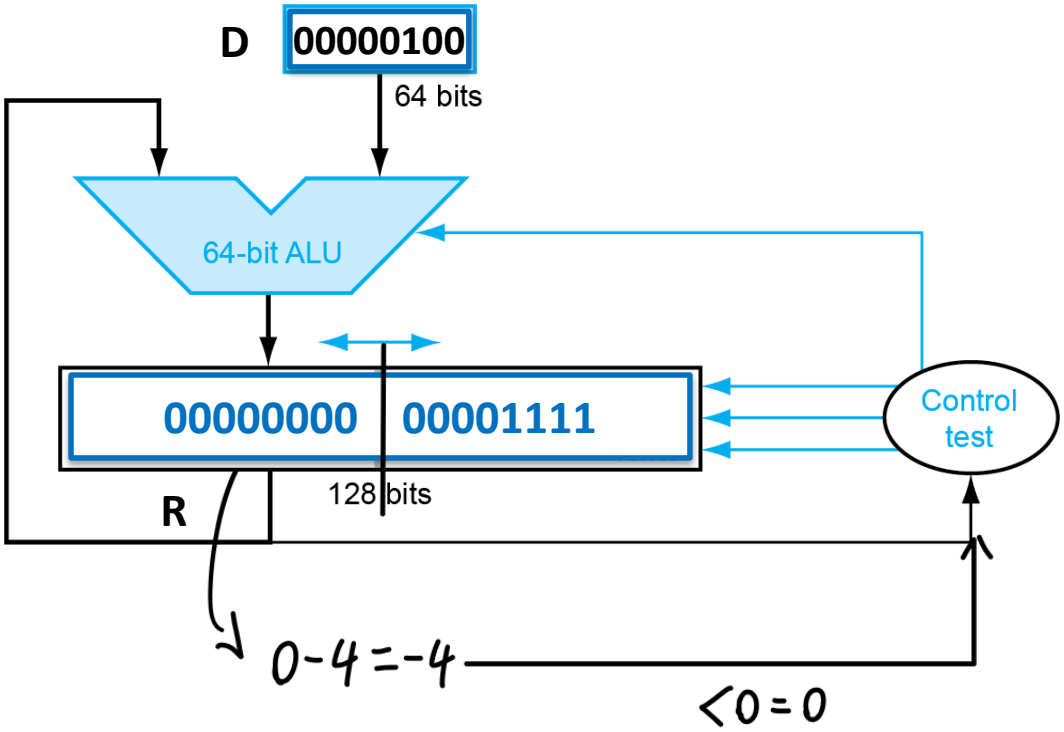
Optimized Divider



- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both

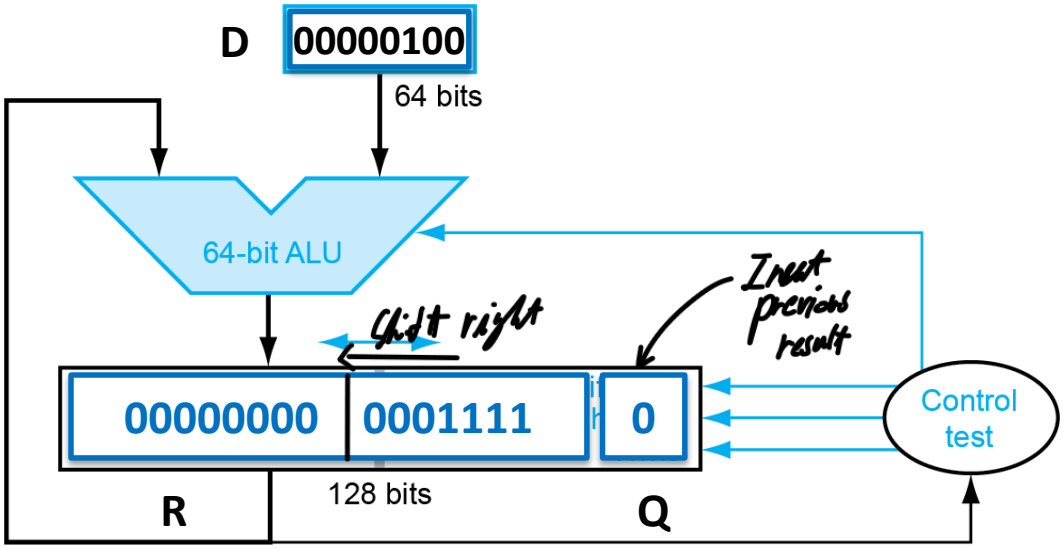
Division Example (1)

- $15_{10} / 4_{10} = 00001111_2 / 00000100_2$



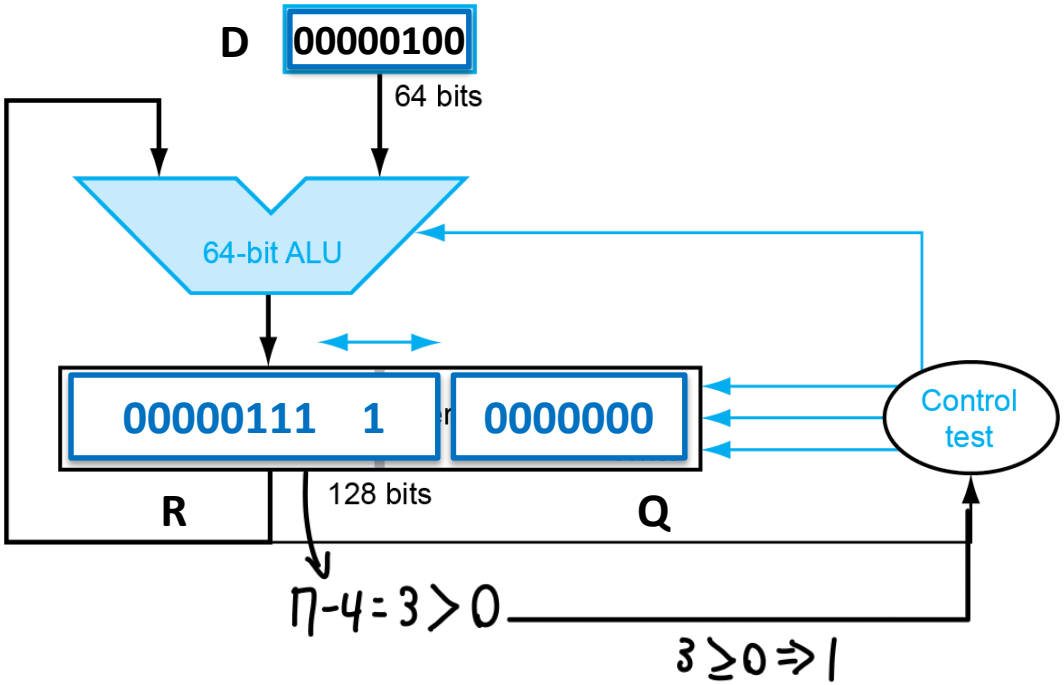
Division Example (2)

- R – D < 0: Shift left R and Q by 1 bit, $Q_0 = 0$



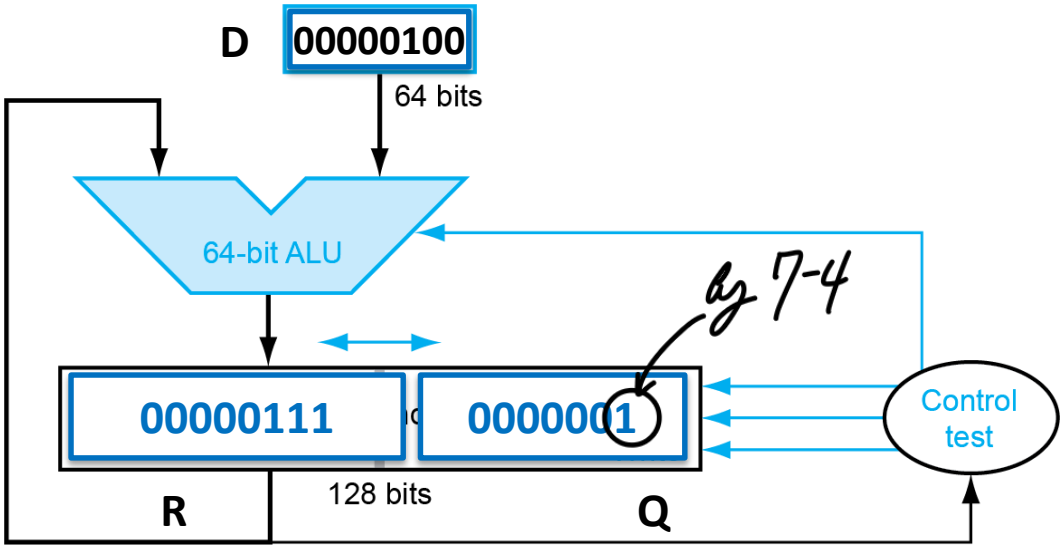
Division Example (3)

- R – D < 0: Shift left R and Q by 1 bit, $Q_0 = 0$ (6 times)



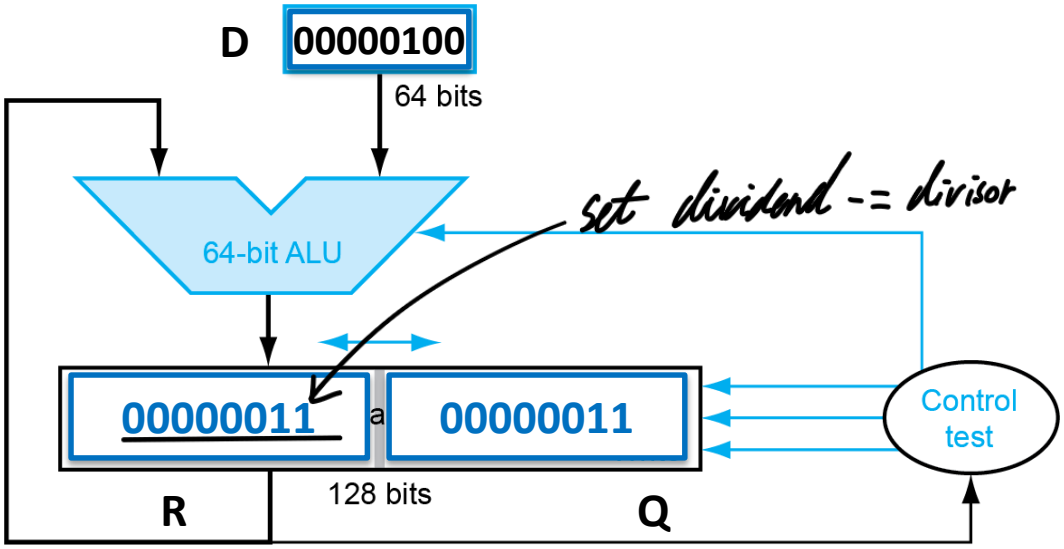
Division Example (4)

- R – D >= 0: R = R – D, shift left R and Q by 1 bit, $Q_0 = 1$



Division Example (5)

- R – D >= 0: R = R – D, shift left Q by 1 bit, $Q_0 = 1$



More on Division

$$-\left(\frac{x}{y}\right) == \frac{(-x)}{y}$$

$$Dividend = Quotient \times Divisor + Remainder$$

- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required
    - (e.g.) 7 / 2: Q = 3, R = 1
      (-7) / 2: Q = -3, R = -1
      7 / (-2): Q = -3, R = 1
      (-7) / (-2): Q = 3, R = -1

- Faster division
  - Can't use parallel hardware as in multiplier – Subtraction is conditional on sign of remainder
  - Faster dividers (e.g. SRT division) generate multiple quotient bits per step, but still require multiple steps