Computer Systems Principles

Data Representation in C

UMASSCS
SCHOOL OF COMPUTER SCIENCE

Today

- Data representation in C
- Conversion between data types
  - Type casting explicit and implicit
  - Sign extension
  - Overflows
- Bit Manipulation
  - bitwise *and*, *or*, *exclusive-or*, and *not*

Ariane 5

Exploded 37 seconds after lift-off with cargo worth 500 million

Why..

- Computed horizontal velocity as floating-point number
- Converted to 16-bit integer
- Worked for Ariane 4
- Overflowed for Ariane 5

Representation Matters!

- Why do we care so much about bits and how to manipulate them?
- Bits are important for low-level systems programming tasks
  - representing sets in compiler analyses,
  - accessing systems resources,
  - processing, reading, and writing in terms of streams of bits (such as processing packets in network programming),
  - cryptography (encoding or decoding data with complex bit-manipulation), etc.

What do these binary sequences represent?

- 1101
  - unsigned integer: 11 ?
  - signed integer: -5?
- 01000001
  - unsigned integer: 65?
  - character A?
- C data types: unsigned integers, signed integers, characters, …..floats, doubles.

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

Representation Matters!

- No self-identifying data
  - Looking at a sequence of bits doesn't tell you what they mean
  - Could be signed, unsigned integer
  - Could be floating-point number
  - Could be part of a string
- The machine interprets what those bits mean!

A simple C program..1

```
# include <stdio.h> // This is needed to run the
                    // printf() function
int main()
{
   // displays the content inside the quotes
   printf("C Programming is Fun!");
   return 0;
}
```

A simple C program..2

```
# include <stdio.h> // This is needed to run the
                    // printf() function
int main()
{
   int var = 10;
   int c = 69;
// displays the content inside the quotes
   printf("Number = %d",var);
   printf("Character of ASCII value 69: %c",c);
   printf("Character of ASCII value 69: %d",c);
   return 0;
}
```

A simple C program..3

```
# include <stdio.h> // This is needed to run printf()
int main()
{
    int a;
    short int b;
    unsigned int c;
    char d;
    // size-of displays the size of the data type
    printf("Size of int=%d bytes\n",sizeof(a));
    printf("Size of short int=%d bytes\n",sizeof(b));
    printf("Size of unsigned int=%d bytes\n",sizeof(c));
    printf("Size of char=%d bytes\n",sizeof(d));
    return 0;
}
```

Data types in C

```
int x;
```

"typically reflecting the natural size of integers on the host machine" [K&R]

- first IBM PC: `int` [16bits]
- today's PC: `int` [32bit]
  (even on 64-bit PCs – but be careful!)

Data types in C (for gcc)

C Data Type	Typical 32-bit	Intel IA 32	x86-64
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	4	8
`long long`	8	8	8
`float`	4	4	4
`double`	8	8	8
`long double`	8	10/12	10/16
`pointer`	4	4	8

Code Portability?

Notice that `long` and `pointer` data types are different on different processors (and maybe compilers).

Casting Signed to Unsigned

C allows conversions from Signed (two's complement) to Unsigned.

```
short int          x =  15213;
unsigned short int ux = (unsigned short) x;
short int          y  = -15213;
unsigned short int uy = (unsigned short) y;
```

Resulting Value

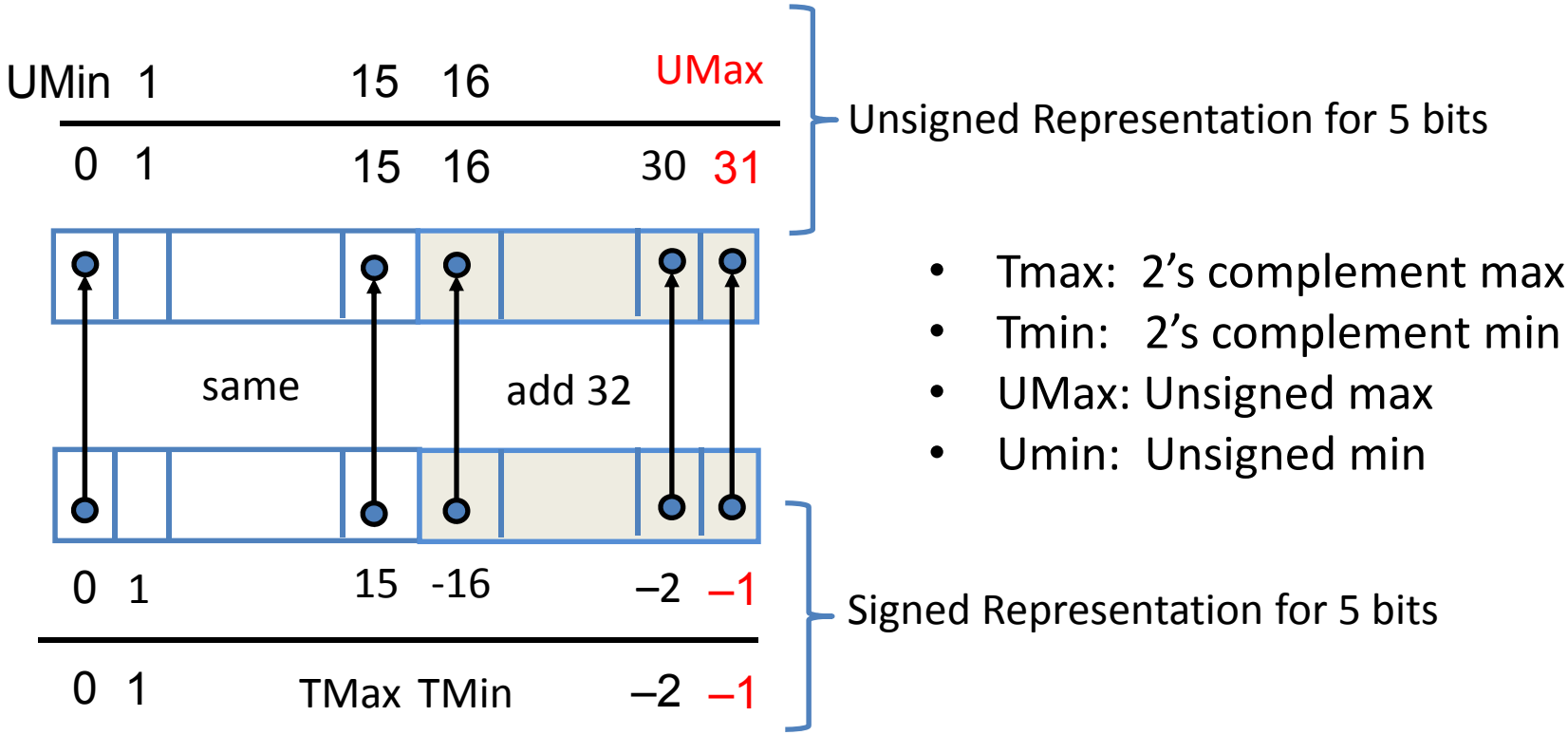
- No change in bit representation!
- Non-negative values unchanged
- Negative values change into (large) positive values

Signed vs. Unsigned in C

- Declaration for two signed and unsigned integers
  - ```
    int tx, ty;       // signed
    unsigned ux, uy; // unsigned
    ```
- Explicit casting between signed & unsigned
  - ```
    tx  = (int) ux;
    uy  = (unsigned) ty;
    ```
- Implicit casting also occurs via assignments and procedure calls
  - ```
    tx  =  ux;
    uy  =  ty;
    ```
    ```

Explanation of Casting Surprises

- 2's Complement to Unsigned: (Same number of bits)
  - A small negative value maps to a large positive value!
  - (e.g., Signed representation of -1 maps to unsigned representation of 31! (5-bit integers))



- Tmax: 2's complement max
- Tmin: 2's complement min
- UMax: Unsigned max
- Umin: Unsigned min

Expanding Bit representation : Sign Extension

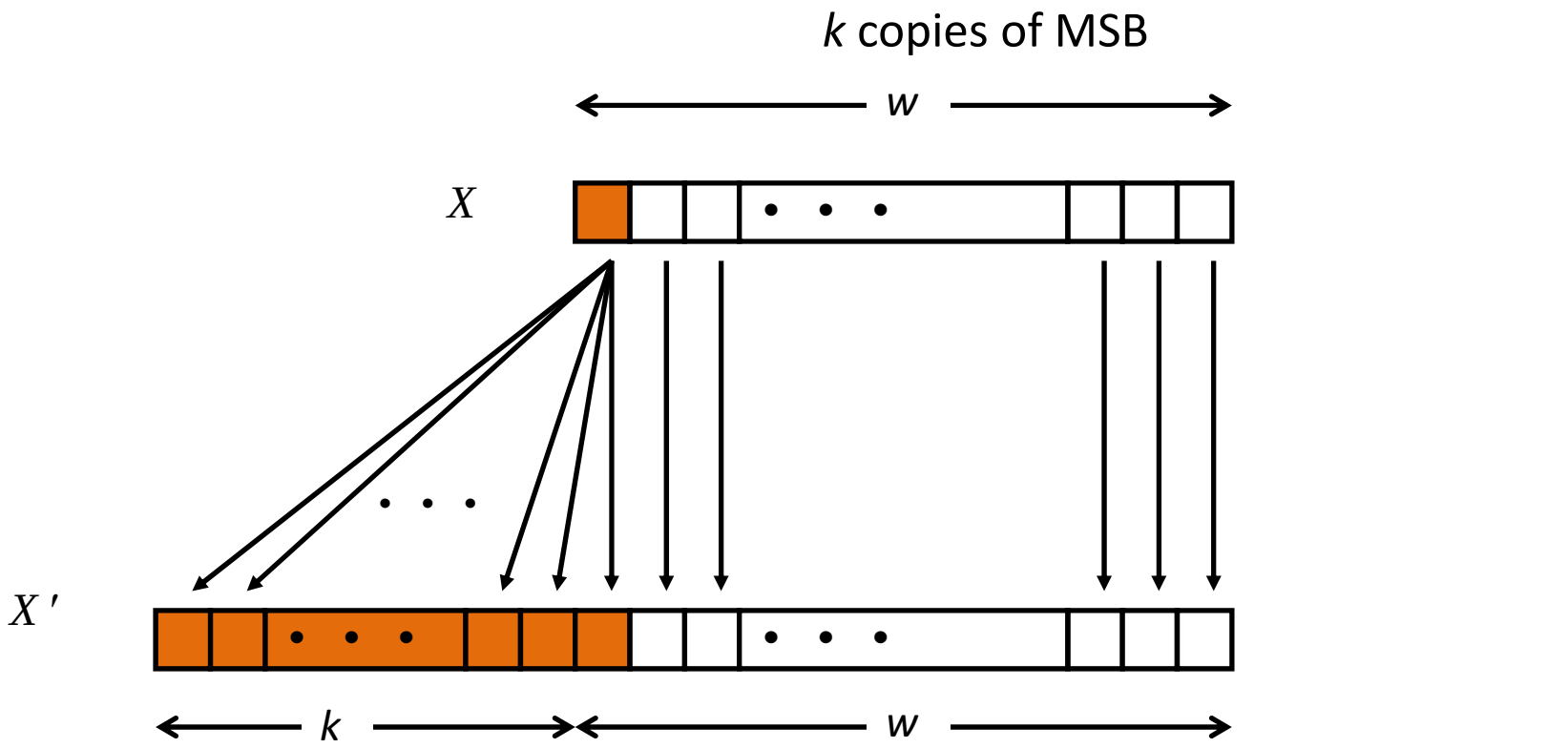
- Converting from smaller to larger integer data type
- C automatically performs sign- or zero- extension

```
short int sx =  -12345;
int    x  = sx;
unsigned short int usx = sx;
unsigned int ux = usx;
```

Variables	Value	Hexadecimal representation	Binary representation
sx	-12345	cf c7	11001111 11000111
usx	53191	cf c7	11001111 11000111
x	-12345	ff ff cf c7	11111111 11111111 11001111 11000111
ux	53191	00 00 cf c7	00000000 00000000 11001111 11000111

Expanding Bit representation : Sign Extension

- Given *w*-bit signed integer *x:*
  - Convert it to *w*+*k*-bit integer with same value
  - Make *k* copies of sign bit: $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$



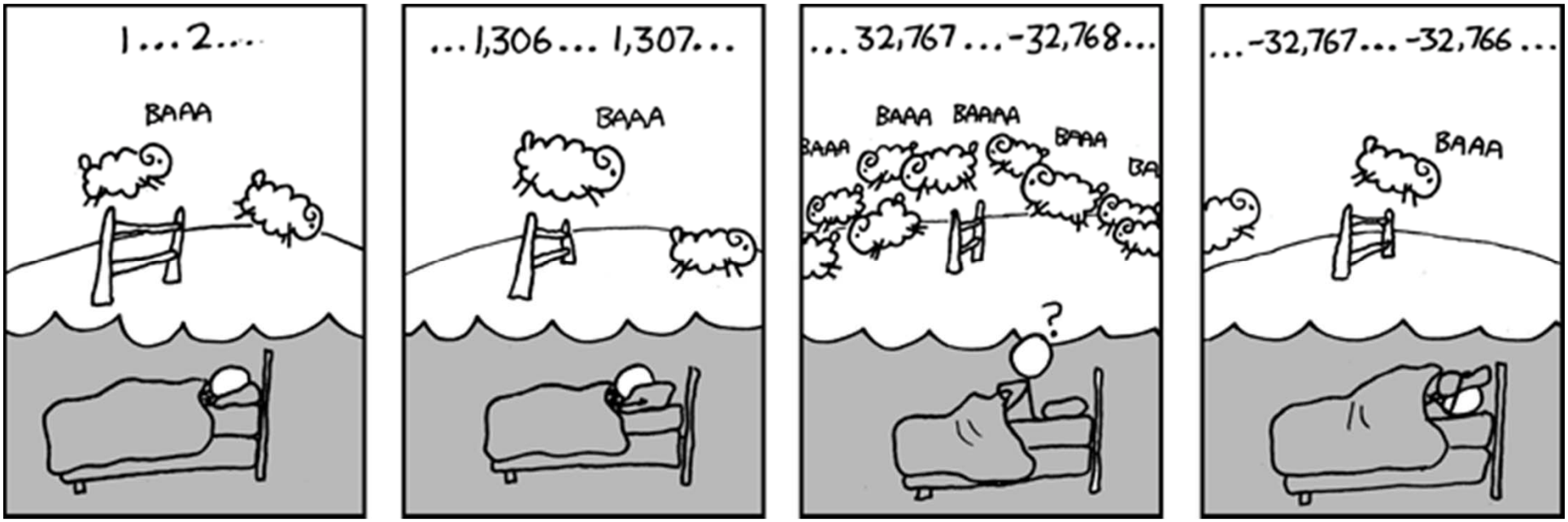
Expanding, Truncating: Basic Rules

- Expanding (e.g., `short int` to `int`)
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected results
- Truncating (e.g., `unsigned short` to `short`)
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small numbers can yield unexpected behavior

Casting Surprises: Expression evaluation

- If there is a mix of unsigned and signed in a single expression, signed values are implicitly cast to unsigned!
- Including comparison operations <,>,==,<=,>=
- E.g.: *W* = 32 TMIN = -2,147,483,648 (2^31) TMAX = 2,147,483,647 (2^31-1)

Constant-1	Relation	Constant-2	Evaluation
0	==	0u	unsigned
-1	<	0	signed
-1	>	0u	unsigned
2147483647	>	-2147483648	signed
2147483647u	<	-2147483648	unsigned
-1	>	-2	signed
(unsigned) -1	>	-2	unsigned
2147483647	<	2147483648u	unsigned
2147483647	>	(int) 2147483648u	signed



Is this dynamic ram??

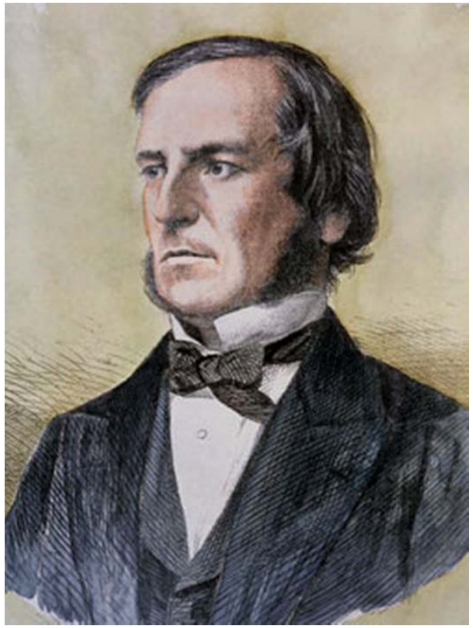
Source: http://xkcd.com/571/

Bit-Manipulations

Boolean Algebra:

- Developed by George Boole in the 19th Century and applied to Digital Systems by Claude Shannon

Operators	Operator Definition
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right



"Laws of Thought"
image source: Wikipedia

Bit-Manipulations

Boolean Algebra:

- Encode "True" as 1 and "False" as 0

Not (~A)

~	
0	1
1	0

And (A & B)

&	0	1
0	0	0
1	0	1

Or (A|B)

|	0	1
0	0	1
1	1	1

Xor A^ B

^	0	1
0	0	1
1	1	0

Bit-Manipulations

Boolean operations are applied *bitwise* on the bit sequences (i.e., by columns)

Not (~A)	And (A & B)	Or (A|B)	Xor A^ B
	0 1 1 0	0 1 1 0	0 1 1 0
~ 1 0 1 0	& 1 0 1 0	| 1 0 1 0	^ 1 0 1 0
0 1 0 1	0 0 1 0	1 1 1 0	1 1 0 0

Bit Manipulations

Boolean algebra obeys some of the properties of integer algebra.. but not all!

Boolean	Boolean	Integer
Sum and product identities	A|0 = A A&1 = A	A+0 = A A*1 = A
Zero is product annihilator	A & 0 = 0	A *0 = 0
Cancellation of negation	~(~A) = A	-(-A) = A
Laws of Complements	A | ~ A = 1	A + -A ≠ 1
Every element has an additive inverse	A | ~A ≠ 0	A + -A = 0

Bit Manipulations: shift operators

- Left shift: x << k : Shift bit-vector x left k positions
  - Throw away extra bits on the left
  - Fill with 0's on the right.
  - 1001 0001<< 2 = 0100 0100
  - x << k is equivalent to x * 2^k

Bit Manipulations: shift operators

- Right shift: x >> k : Shift bit-vector x right k positions.
  - Throw away extra bits on the right
  - x >> k corresponds to $x/2^k$ for rounded down.

TWO KINDS:

- Logical Shift: Fill with 0's on the left. Applies to C *unsigned*.
  - 1001 0001 >> 3 = 0001 0010
    - 1001 0001 >> 3 in decimal : 145 / 2^3 = 18.125
    - 0001 0010 in decimal : 18
- Arithmetic Shift : Replicate with most significant bit on the left.
  - Copies the sign bit – applies to C *signed* numbers
  - Arithmetic shift is equivalent to logical shift for positive numbers
  - 1001 0001 >> 3 = 1111 0010
    - 1001 0001 >> 3 in decimal: (-111) / 2^3 = -13.875
    - 1111 0010 in decimal: -14

Comparison with shifting in Java

C:

- Has *signed* and *unsigned* types
- >> operates according to the *type* of the operand

Java:

- Has only *signed* types
- >> is arithmetic shift, >>> is logical shift

Bit Manipulations in C

```
# include <stdio.h> // This is needed to run printf()
function
int main()
{
    int a;
    short int b;
    unsigned int c;
    char d;
    // size-of displays the size of the data type
    printf("Size of int=%d bytes\n",sizeof(a));
    printf("Size of short int=%d bytes\n",sizeof(b));
    printf("Size of unsigned int=%d bytes\n",sizeof(c));
    printf("Size of char=%d bytes\n",sizeof(d));
    return 0;
}
```

iClicker Question

Compute this *arithmetic* right shift:

1001 0001 >> 2

a) 1111 0010
b) 1110 0100
c) 1110 0101
d) 0010 0100

Ariane 5

Exploded 37 seconds after lift-off with cargo worth 500 million

Why..

- Computed horizontal velocity as floating-point number
- Converted to 16-bit integer
- Worked for Ariane 4
- Overflowed for Ariane 5

Ariane 5: Spot the problem..

```
L_M_BV_32 := TBD.T_ENTIER_32S ((1.0/C_M_LSB_BV) * G_M_INFO_DERIVE(T_ALG.E_BV));

if L_M_BV_32 > 32767 then
      P_M_DERIVE(T_ALG.E_BV) := 16#7fff#;
elsif L_M_BV_32 < -32768 then
      P_M_DERIVE(T_ALG.E_BV) := 16#8000#;
else
      P_M_DERIVE(T_ALG.E_BV) := UC_16S_EN_16NS(TBD.T_ENTER_16S(L_M_BV_32));
end if;
```

Checks for integer range for vertical velocity of rocket

```
P_M_DERIVE(T_ALG.E_BH) :=
UC_16S_EN_16NS (TDB.T_ENTIER_16S ((1.0/C_M_LSB_BH) *
G_M_INFO_DERIVE(T_ALG.E_BH)));
```

Horizontal velocity..

Ariane 5: Problem Fix

```
L_M_BV_32 := TBD.T_ENTIER_32S ((1.0/C_M_LSB_BV) * G_M_INFO_DERIVE(T_ALG.E_BV));

if  L_M_BV_32 > 32767 then
      P_M_DERIVE(T_ALG.E_BV) := 16#7fff#;
elsif L_M_BV_32 < -32768 then
      P_M_DERIVE(T_ALG.E_BV) := 16#8000#;
else
      P_M_DERIVE(T_ALG.E_BV) := UC_16S_EN_16NS(TBD.T_ENTER_16S(L_M_BV_32));
end if;
```

Checks for integer range for vertical velocity of rocket

```
L_M_BH_32 := TBD.T_ENTIER_32S ((1.0/C_M_LSB_BV) * G_M_INFO_DERIVE(T_ALG.E_BH));

if L_M_BH_32 > 32767 then
      P_M_DERIVE(T_ALG.E_BH) := 16#7fff#;
elsif L_M_BH_32 < -32768 then
      P_M_DERIVE(T_ALG.E_BH) := 16#8000#;
else
      P_M_DERIVE(T_ALG.E_BH) := UC_16S_EN_16NS(TBD.T_ENTER_16S(L_M_BH_32));
end if;
```

Checks for integer range for horizontal velocity of rocket!

Summary

- Bit representation and manipulation is extremely useful in a wide variety of applications like compiler analyses, network programming, cryptography and many more!
- The same binary sequence can be used to represent ASCII characters, unsigned binary, and two's complement integers. Their interpretation is based on the context in which they are defined!
- C has different data types to store integers and floating point numbers that have different memory sizes on different operating systems.
- Typecasting operations between two different data types can be explicit or implicit.
  - Casting surprises when changing between data types can change the numeric value.
  - Casting surprises also occur if we use arithmetic and relational operators on two different data types.
- Boolean algebra includes {not, and, or and x-or} operations and left and right shifts.
  - Not to be confused with conditional operators !
- Next class we will cover more programming in C!