

CSE 3302, Summer 2019

Module 05

Target Machine Architecture



Architecture vs Implementation

- A machine's *architecture* is the *formal* specification of the capabilities and operation of the machine in the abstract.
 - It's what one needs to know to generate machine code for the *bare* machine (that is, no other software present).
- An *implementation* of an architecture is how the architecture is realized as a specific machine.
 - Thus we speak generally of the x86 (or ARM) architecture, but we have specific implementations such as the 8086 (1978) or the Ryzen 7 1800X (2017).



Implications for Programming Languages

- An aspect of a machine's architecture is the *machine language* it accepts.
- As there are very few architectures in general use, there are very few machine languages.
 - Different *implementations* of the *same* architecture can have machine language *differences* (e.g., because of options).
- Even those few machine languages are quite *similar* to each other.
- This contrasts with the plethora of *higher level programming languages* supported on those machines.
 - And how different all of those programming languages can be from each other.



Implications for Programming Languages

- Generally, only the *compilers* (and *interpreters*) have to care about the actual machine language.
 - Part of their job is to insulate the user from specific machine dependencies.
- *Good* compilers take into account the subtle differences between different implementations of the same architecture.
 - This is necessary to get the *best possible* performance.
 - Might include code matched to the exact version of the processor. (E.g., to take advantage of special instructions.)



Macro Architecture

- A machine's *macro architecture* is how it is structured as a collection of discrete units.
 - For example, a processor, some memory cards, a GPU card, an HDD and/or SSD, and so forth.
- Almost all modern machines at the macro level are a collection of *Devices* connected by a *Bus*.
 - A machine might have *multiple* busses to improve performance.
- The most important device is the *processor* itself.



Processor Architecture

- Almost all processors use the (von Neumann) Stored Program concept.
 - A program is a series of *bits* kept in memory that the processor *interprets* as *instructions* rather than as *data*.
 - The processor performs this loop repeatedly,
 - Fetch some bits from memory.
 - Decode the bits as an instruction.
 - Fetch any required data from memory.
 - Execute the required operation.
 - Store any generated results to memory.



<https://commons.wikimedia.org/wiki/File:JohnvonNeumann-LosAlamos.gif>



Processor Architecture

- The loop is known as the *fetch-execute* cycle.
- Every processor performs this loop as fast as it can *forever*.
 - Even when there is nothing useful to do, the processor has to be doing something, so useless instructions are executed.
 - Known as the *null job* or *idle processing*.
- Even if the processor is told to fetch *garbage*, it will try to execute the garbage as an instruction.
 - *Garbage* might be *data* that was not intended to be used as instructions.
 - Maybe the data won't decode properly, maybe it will. Who knows?
 - *Something* will happen. It might not be useful, but it will happen.
 - The *something* might be an *Unable to Decode Instruction* error.



[*Getting Use out of Idle Processing*]

- Generally speaking, as long as a processor is powered, it is (trying to) execute instructions.
- Most of the time, the processor is waiting for the *user* to do something and so the processor is *idling* or *running the null job*.
- To make use of this otherwise wasted execution time, there are a number of *distributed computing projects* to which a user can donate idle time.
- The user installs software which takes the place of the *null job* so whenever the processor is not otherwise working, it contributes to the project.
- The Folding@home project is an example of such a project, now consisting of 8,355,996 processors, running at a combined rate of 98,747 x86 TFLOPs.

As of 2019 June 19.

<http://fah-web.stanford.edu/cgi-bin/main.py?qttype=osstats2>



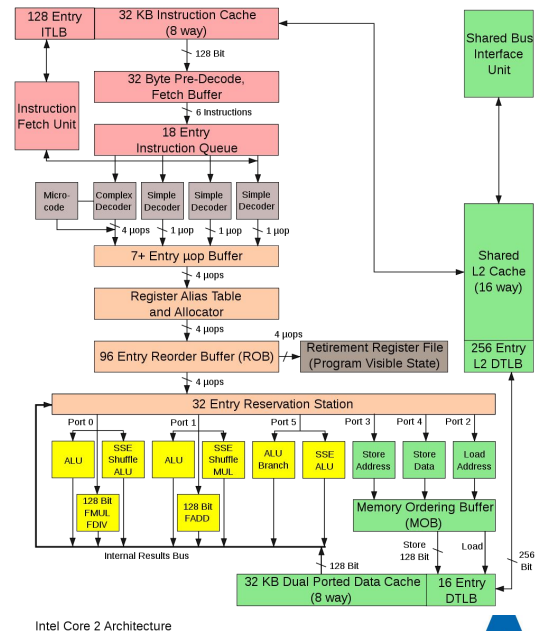
Processor Architecture

- The processor internally comprises a collection of *functional units*, each of which has a particular purpose. For example,
 - Instruction Fetch
 - Instruction Decode
 - Arithmetic or Logical Operation
 - Memory
 - Registers
 - L1 (and maybe L2) Memory Caching
 - Overall Processor Control
 - Internal / External Communication Control



Machine Organization

- A fairly simple set of *functional units* inside a processor
(Intel Core 2, 2006).
- Much of the complexity concerns the fetching of operands and the caching of results.
- The actual ALU operations might be comparatively simple.



Intel Core 2 Architecture

"Appaloosa" (WikiMedia Commons)
https://commons.wikimedia.org/wiki/File:Intel_Core2_arch.svg



The Memory Hierarchy

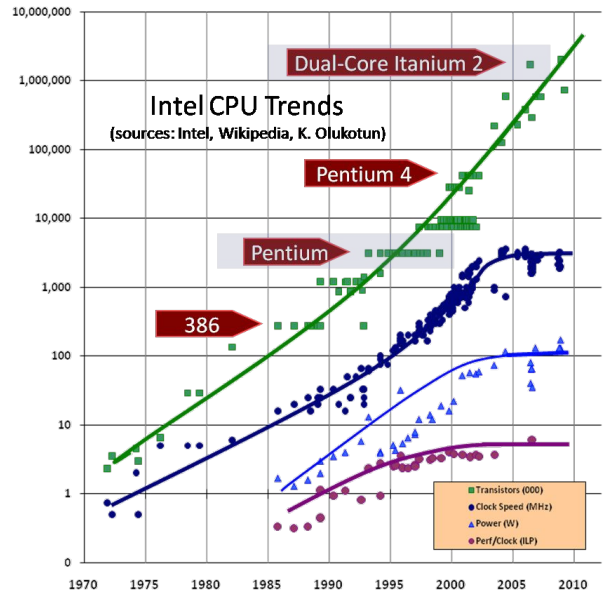
- Historically, the *processor's* speed increased faster than the *memory* speed.
 - Memory speed is catching up today as processor speed growth rate slows (or stops).
- Also, the *total amount* of memory available to the machine grew significantly as larger problems were considered.
 - On the order of 10^6 to 10^7 times as much today.
- The need for *storage* as well encouraged a *hierarchy* of memory technologies, each level of which with its own characteristics.



[“Recent” Trends in Processor Development]

- Processor *clock speed* generally plateaued in about 2004.
 - Even in 2019, the top processor speed is only about 5.0 GHz.
 - Using the slope from the 1970s though 2004, it would have been $\gg 10$ GHz.
- Performance improvement has been because of increased numbers of functional units, cores, etc., as shown by the increasing numbers of *transistors*.

<http://www.gotw.ca/publications/concurrency-ddj.htm>
<http://www.gotw.ca/images/CPU.png>
<https://github.com/karlrupp/microprocessor-trend-data>



The Memory Hierarchy

<i>Memory</i>	<i>Access Time (Typical)</i>	<i>Capacity</i>
Registers	0.2 – 0.5 ns	256 B – 1024 B
L1 (primary) cache	0.4 – 1 ns	32 kB – 256 kB
L2, L3 (on-chip) cache	4 – 30 ns	1 MB – 32 MB
Off-chip cache	10 – 50 ns	up to 128 MB
Main memory	50 – 200 ns	256 MB – 16 GB
Flash / SSD memory	40 – 400 μ s	4 GB – 16 TB
HDD storage	5 – 15 ms	500 GB – 12 TB
Tape storage	1 – 50 s	Effectively unlimited

circa 2015-2017



The Memory Hierarchy

- *Registers* — On the processor chip, directly accessible through instruction. Fastest access. Very limited size.
- *L1 Cache* — On chip. Very fast access. Limited size.
- *L2, L3 Cache* — On chip, possibly shared between cores. Very fast access. Bigger than L1, but still limited size.
- *Main Memory* — Off chip and across the bus. Fast access. Size limited by architecture: 4 GB (for 32-bit), ≥ 256 TB (for 64-bit) [but limited by OS concerns to about 1 TB to 16 TB].
- *"Disks"* — Across the bus. Slow, but SSD can be 10x HDD. Size limited by partition methods. Non-volatile.
- *Tape* — Across the bus. Very, very slow. Essentially unlimited size. Non-volatile.



The Memory Hierarchy

- The programmer generally can distinguish *Registers* from *Main Memory* from *Disks/Tapes*.
 - Registers and Main Memory have specific instruction support.
 - Disks/Tapes are handled as external devices or *peripherals*.
- The L1, L2, L3, ... caches are generally *invisible* from a functional point of view, but can cause observable *delays*.
 - The L1 cache is almost as fast as the registers.
 - The L2, L3 caches are on chip yet usually about 10x - 60x slower than the registers.
 - Any off-chip caches might be 100x slower than the registers.



The Memory Hierarchy

- 100x slower sounds pretty bad, but Main Memory is even worse, at *100x - 400x* slower than the registers.
- Now you know why compilers work to keep as much as possible in the registers and why designing algorithms that don't spill out of the cache is so important.
- For some problems, this is just not possible and one has to put up with the slower access times.
 - Heaven forbid you have to spend much time looking at data on disks or — *shudder* — tapes.



Organization of Memory

- Most architectures (though not all) have *byte-addressable* memory.
 - That is, each *byte* in memory has its own address.
- Integers (signed *or* unsigned) come in various sizes.
 - Often 2, 4, or 8 bytes, the typical **short**, **int**, and **long** data types.
 - 16 byte (and larger) integers also exist.
- Floating Point numbers also come in various sizes.
 - 4 and 8 bytes are the typical **float** and **double** data types.
 - 16 byte (and larger) FP values also exist, but are rare.



Big Endian vs. Little Endian

- There are *two* ways a multi-byte value can be placed in memory.
- Big Endian* means the *Most Significant Byte* gets the lowest address.
- Little Endian* means the *Least Significant Byte* gets the lowest address.
- The values 0x12345678 and 0x11223344 are both four bytes long.
- The tables show how they would be stored at addresses 1000 and 1004 in either Big or Little Endian order.

Big Endian

Address	Value
1000	11
1001	22
1002	33
1003	44
1004	12
1005	34
1006	56
1007	78

Little Endian

Address	Value
1000	44
1001	33
1002	22
1003	11
1004	78
1005	56
1006	34
1007	12

Address		MSB			LSB
1000	0x	11	22	33	44
1004	0x	12	34	56	78



Big Endian vs. Little Endian

- The x86 architecture is Little Endian.
- IBM's z/Architecture and Motorola 68000 are Big Endian.
- Many common processors can be either.
 - The Operating System sets the Endian mode.
 - Alpha, ARM, IA-64, MIPS, PA-RISC, PowerPC, SPARC, ...
- Most Network Protocols express data in Big Endian format.
 - IPv4, IPv6, TCP, UDP, ...



Interpretation of Memory

- Values stored in memory locations have *no* format or type, only a *size*.
- *No* interpretation is placed on the value in and of itself.
- Values have a type only in how they are *used*.
 - That is, which instructions are used to manipulate them.

<i>Element</i>	<i>Size</i>
byte	8 bits
half-word (word)	16 bits
word (double word)	32 bits
long	64 bits
long long	128 bits



Interpretation of Memory

- A single 32 bit word of memory with three different interpretations.
 - `unsigned int`
 - `int`
 - `float`
- It's how the memory is *used* that determines its semantic *value*. The bits are the *same*.

```
union {
    unsigned int asUInt;
    int         asInt;
    float       asFloat;
} oneWord;

int main() {
    oneWord.asInt = 1;
    printf( "Assigning 1 to asInt results in:\n" );
    printf( "0x%08X: int %11d, uint %10u, float %9f, %e\n\n",
        oneWord.asUInt, oneWord.asInt, oneWord.asUInt,
        oneWord.asFloat, oneWord.asFloat );

    oneWord.asUInt = 0x3F << 24;
    printf( "Assigning 0x3F << 24 to asUInt results in:\n" );
    printf( "0x%08X: int %11d, uint %10u, float %9f, %e\n\n",
        oneWord.asUInt, oneWord.asInt, oneWord.asUInt,
        oneWord.asFloat, oneWord.asFloat );

    oneWord.asFloat = -1.0;
    printf( "Assigning -1.0 to asFloat results in:\n" );
    printf( "0x%08X: int %11d, uint %10u, float %9f, %e\n",
        oneWord.asUInt, oneWord.asInt, oneWord.asUInt,
        oneWord.asFloat, oneWord.asFloat );
}
```



Interpretation of Memory

```
Assigning 1 to asInt results in:
0x00000001: int      1, uint      1, float  0.000000, 1.401298e-45

Assigning 0 to asInt results in:
0x00000000: int      0, uint      0, float  0.000000, 0.000000e+00

Assigning -1 to asInt results in:
0xFFFFFFFF: int     -1, uint 4294967295, float  -nan, -nan

Assigning 0x3F << 24 to asUInt results in:
0x3F000000: int 1056964608, uint 1056964608, float  0.500000, 5.000000e-01

Assigning 1.0 to asFloat results in:
0x3F800000: int 1065353216, uint 1065353216, float  1.000000, 1.000000e+00

Assigning 0.0 to asFloat results in:
0x00000000: int      0, uint      0, float  0.000000, 0.000000e+00

Assigning -0.0 to asFloat results in:
0x80000000: int -2147483648, uint 2147483648, float -0.000000, -0.000000e+00

Assigning -1.0 to asFloat results in:
0xBF800000: int -1082130432, uint 3212836864, float -1.000000, -1.000000e+00
```



Signed and Unsigned Integers

- Integers are a fundamental data type.
- Almost all processors implement two forms of integers.
 - *Unsigned* and *Signed* Integers
- *Unsigned* integers (usually) range from 0 to 2^n-1 , where n is the number of bits used to represent the integer.
 - The actual bits are (usually) the binary form of the value.
- *Signed* integers (usually) range from -2^{n-1} to $2^{n-1}-1$.
 - The actual bits can be in several different formats.



Signed and Unsigned Integers ...

<i>Bits</i>	<i>Unsigned Byte</i>	<i>Two's Complement Byte</i>
0111 1111	127	127
0111 1110	126	126
0000 0010	2	2
0000 0001	1	1
0000 0000	0	0
1111 1111	255	-1
1111 1110	254	-2
1000 0010	130	-126
1000 0001	129	-127
1000 0000	128	-128



Two's Complement for Signed Integer

- Positive numbers are represented by themselves.
 - $1 \rightarrow 0000\ 0001$, $2 \rightarrow 0000\ 0010$, etc., in eight bits.
- Negative numbers are represented by 2^n plus the number.
 - $-1 \rightarrow 1111\ 1111$, $-2 \rightarrow 1111\ 1110$, etc., in eight bits.
- Negative numbers always have the high bit set.
- The range of representable numbers in n bits is -2^{n-1} to $2^{n-1}-1$. So, -128 to 127 in eight bits.
- There is *one more* negative number than positive number.
 - We need a representation for zero!



Computing “Minus” of a Two’s Complement ...

- To compute $-n$, invert the bits and add 1. Simple!
- For example, for “minus” 1 in eight bits we have,
 - 1 in eight bits is 0000 0001
 - 0000 0001 inverted is 1111 1110
 - 1111 1110 + 1 is 1111 1111, which is -1 two’s complement.
- For “minus” -127 in eight bits we have,
 - -127 two’s complement in eight bits is 1000 0001
 - 1000 0001 inverted is 0111 1110
 - 0111 1110 + 1 is 0111 1111, which is 127 two’s complement.



Two’s Complement Observations ...

- Why use Two’s Complement?
 - Forming the negative is *easy*!
 - Inverting bits and adding 1 are both trivial operations.
 - Recognizing a negative number is *easy*! The high bit is 1.
 - It makes adding and subtracting *the same operation*.
 - For example, to compute $2 + 1$, we have (in eight bits)
 - 0000 0010 + 0000 0001 \rightarrow 0000 0011 = 3
 - To compute $2 - 1$, we compute $2 + -1$ (in eight bits)
 - 0000 0010 + 1111 1111 \rightarrow 0000 0001 = 1
 - The rest of the “addition” just flowed off the left side.



Floating Point Representation

- Similar to *Scientific Notation*,
 - which includes a *Sign*, a *Mantissa*, and an *Exponent*.
 - The *Sign* is Plus or Minus.
 - The *Mantissa* is (normally) a number between 1.0 and 9.99...
 - The *Exponent* is an integer.

A diagram illustrating the components of a floating point number. The expression is -1.234567×10^4 . The minus sign is circled in orange and labeled "Sign" with an arrow. The digits "1.234567" are enclosed in an orange box and labeled "Mantissa" with an arrow. The "4" in the exponent is circled in orange and labeled "Exponent" with an arrow.



Floating Point Representation

- Floating Point (FP) representation is a property of the underlying *hardware* as FP *operations* are generally built in.
 - If FP is implemented by *software*, any / multiple representations may be used on the same machine.
 - Have to agree when *exchanging* FP values, though.
- In the early days (pre-1985), every company had its own idea of what the “best” representation was.
- This caused much grief for anyone going from machine to machine.



Floating Point Representation

- Eventually we got to a cross-company standard, IEEE 754.
 - Originally issued in 1985.
 - Latest version is IEEE 754 -2008.
- Specifies representations, operations, rounding rules, possible exceptions, etc.
- Solved a lot of headaches, even though there will be arguments forever about the “best” way to represent floating-point numbers.




Floating Point Representation

- In memory, Floating Point numbers are represented as,
 - a *Sign*, an *Exponent*, and a *Significand*.
- The *Sign* is Plus or Minus, same as in Scientific Notation.
- The *Exponent* is a power of 2, an unsigned integer.
 - But represented as a *biased* value ...
- The *Significand* is a binary *fraction*, interpreted depending on the value of the exponent.
 - *Similar to but not the same* as the *Mantissa* of Scientific Notation!



Floating Point Representation Examples ...

- IEEE 754 **binary32** representation has
 - 1 sign bit, 8 exponent bits, 23 significand bits, 1 *hidden* bit.
- The *exponent* is *biased* by 127 (with two special cases).
 - Exponent values 0 and 255 are handled specially.
- The *significand* is a *binary fraction*.
 - The *hidden* bit is not explicitly represented; depends on *exponent*.
 - For exponent = 0, the significand value (SV) is 0. *b b b ...*
 - For exponent non-zero, the SV is 1. *b b b ...* 
- The final interpreted value is $-1^{\text{sign}} \times 2^{\text{exponent}-127} \times \text{SV}$.



FP Representation Exponent Special Cases

- *Exponent is 0.*
 - *Significand is 0:* Value is ± 0 , depending on the sign bit.
 - $+0$ and -0 are *different*, but IEEE 754 requires that they compare equal to each other. Useful for some numerical procedures.
 - *Significand is non-zero:* Value is a *denormalized* FP value and is computed specially. (See later examples.)
- *Exponent is $2^n - 1$, where n is the number of exponent bits.*
 - *Significand is 0:* Value is $\pm \infty$, depending on the sign bit.
 - *Significand is non-zero:* Value is $\pm \text{NaN}$, *Not A Number*



Floating Point Representation Examples ...

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Sign (bit 31) = 1
- Exponent (bits 30-23) = $1000\ 0000 = 128$
 - Unbiased exponent = $128 - 127 = 1$
- Hidden bit = 1, since the biased exponent is $\neq 0$.
- Significand value = $1 + 0.111000\dots_2$ (bits 22-0)
 - $= 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} = 1.875_{10}$
- Interpreted value is $-1^1 \times 2^1 \times 1.875 = -3.75$



Floating Point Representation Examples ...

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Sign (bit 31) = 0
- Exponent (bits 30-23) = $0111\ 1110 = 126$
 - Unbiased exponent = $126 - 127 = -1$
- Hidden bit = 1, since the biased exponent is $\neq 0$.
- Significand value = $1 + 0.000000\dots_2$ (bits 22-0)
- Interpreted value is $-1^0 \times 2^{-1} \times 1.0 = 0.5$



Floating Point Representation Examples ...

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

- Sign (bit 31) = 0
- Exponent (bits 30-23) = 0000 0000 = 0
 - Unbiased exponent = -126 (*special case!*)
- Hidden bit = 0, since the biased exponent is = 0.
- Significand value = $0 + 0.000000000000000000000001_2$ (bits 22-0)
 - = 2^{-23}
- Interpreted value is $-1^0 \times 2^{-126} \times 2^{-23} = 2^{-149} \approx 1.401298 \times 10^{-45}$



Floating Point Representation Examples ...

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Sign (bit 31) = 1
- Exponent (bits 30-23) = 0000 0000 = 0
 - Unbiased exponent = -126 (*special case!*)
- Hidden bit = 0, since the biased exponent is = 0.
- Significand value = $0 + 0.000000000000000000000000_2$ (bits 22-0)
 - = 0
- Interpreted value is -0.0, which is *different* from +0.0.



Floating Point Representation Examples ...

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Sign (bit 31) = 0
- Exponent (bits 30-23) = 1111 1111 = 255
 - *Special case!*
- Significand value = 0.
- Interpreted value is *+Infinity*.
 - If the sign bit had been set, the value would be *-Infinity*.



Floating Point Representation Examples ...

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0

- Sign (bit 31) = 1
- Exponent (bits 30-23) = 1111 1111 = 255
 - *Special case!*
- Significand value $\neq 0$ (bits 10, 11, 12 are all 1).
- Interpreted value is *-NaN*.
 - If the sign bit had not been set, the value would be *+NaN*.



IEEE 754 Floating Point Sizes

<i>Size</i>	<i>Sign</i>	<i>Exponent (bias)</i>	<i>Significand</i>
binary32 (single)	1 bit	8 bits (127)	23+1 bits
binary64 (double)	1 bit	11 bits (1,023)	52+1 bits
binary128 (quadruple)	1 bit	15 bits (16,383)	112+1 bits
binary256 (octuple)	1 bit	19 bits (262,143)	236+1 bits

- While the *quadruple* and *octuple* floating point sizes are defined by the IEEE 754 standard, they are rarely supported in hardware (or even software, for *octuple*).



GCC Floating Point Ranges

<i>Size</i>	<i>Decimal Digits</i>	<i>Value Range</i>
<code>float</code> (32 bits)	~7	$\pm 3.402823\text{e}+38$
<code>double</code> (64 bits)	~16	$\pm 1.797693\text{e}+308$
<code>long double</code> (??? bits)	???	???
<code>__float128</code> (128 bits)	~34	$\pm 1.189731\text{e}+4932$

- `long double` can depend on platform / version / etc. Be careful!
- GCC does not directly support IEEE 754 *octuple* size floating point values.
- If they were supported, they would be ~71 decimal digits and have a value range of $\pm 1.6113\text{e}+78913$.



[*How big is +1.6113e+78913?*]

- The *Observable Universe* is estimated ...
 - ... to be 8.8×10^{26} meters in diameter.
 - ... to have a volume of 3.58×10^{80} cubic meters.
 - ... to contain 4.5×10^{51} kilograms of ordinary matter.
 - ... to contain about 10^{80} hydrogen atoms.
- The Bohr radius of a hydrogen atom is about 53 picometers.
 - 5.29×10^{-11} meters, so a volume of about $620 \times 10^{-33} \text{ m}^3$.
- Even if the Universe were packed full of hydrogen atoms, that would be only about 5.77×10^{110} atoms.

https://en.wikipedia.org/wiki/Observable_universe



[*How big is +1.6113e+78913?*]

- So do we really “need” numbers as big as 10^{78913} ?
 - Well, maybe that’s why no one has yet felt a need to implement this format in hardware.
- Perhaps it’s the ~ 71 decimal digits of precision that is attractive.
- Who knows?

https://en.wikipedia.org/wiki/Observable_universe



Review Questions (1)

1. Explain how to compute the additive inverse (negative) of a two's complement number.
2. Explain how to detect overflow in two's complement addition.
3. Do two's complement numbers use a bit to indicate their sign? Explain.
4. Summarize the key features of IEEE 754 floating-point arithmetic.
5. What is the approximate range of single- and double-precision floating-point values? What is the precision (in bits) of each?
6. What is a floating-point NaN?



Review Questions (2)

9. What is the difference between big-endian and little-endian addressing?
10. What is the purpose of a cache?
11. Why do many machines have more than one level of cache?
12. How many processor cycles does it typically take to access primary (level-1) cache? How many cycles does it typically take to access main memory?
14. List four common formats (interpretations) for bits in memory.
15. What is IEEE standard number 754? Why is it important?

