Assembly Language and Microcomputer Interface

Chapter 1 – Introduction to the Microprocessor and Computer

Ming Cai

cm@zju.edu.cn

College of Computer Science and Technology
Zhejiang University

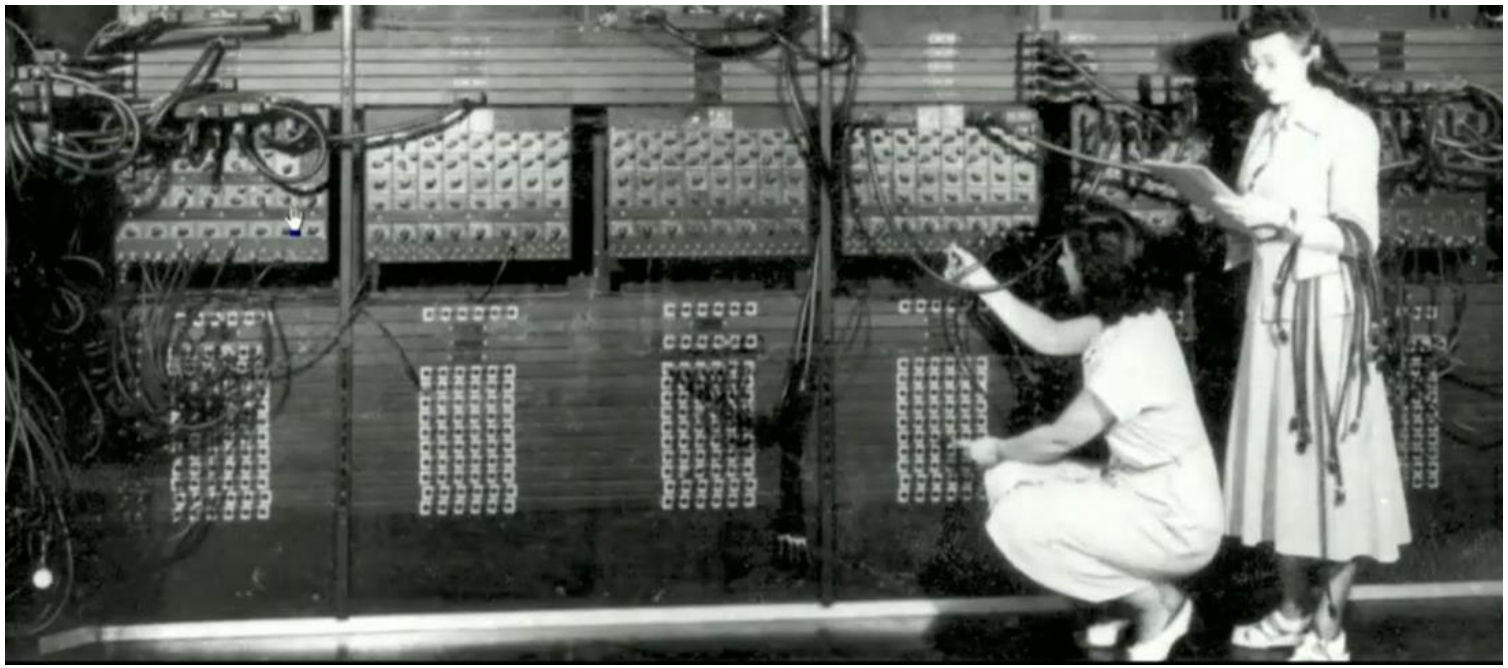
Introduction

- Overview of Intel microprocessors
- Discussion of history of computers
- Function of the microprocessor
- Computer data formats
- Terms and jargon (**computer)**

1–1 A HISTORICAL BACKGROUND

- First general-purpose, programmable electronic computer system ENIAC was developed in 1946.
  - at University of Pennsylvania
  - over 17,000 vacuum tubes
  - 500 miles of wires
  - weighed over 30 tons
  - about 100,000 operations per second

- ENIAC can be programmed by rewiring its circuits.
  - workers changed electrical connections on plug-boards like early telephone switchboards
  - process took many workers several days



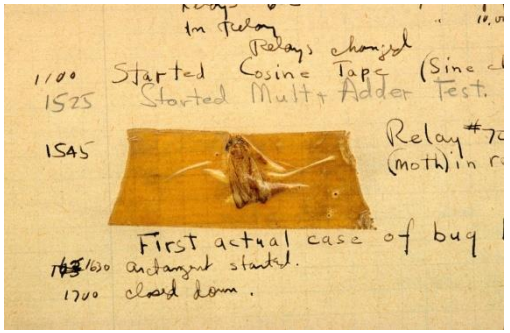
ENIAC 1946

- More efficient than rewiring a machine to program it.
  - still time-consuming to develop a program due to sheer number of program codes required
- Mathematician John von Neumann first modern person to develop a system to accept instructions and store them in memory.
- Once programmable machines developed, programs and programming languages began to appear.

- The first, **machine language**, was constructed of ones and zeros using binary codes.
  - stored in the computer memory system as groups of instructions called a program
- Once systems such as UNIVAC became available in early 1950s, **assembly language** was used to simplify entering binary code.
- In place of a binary number.
  - such as 0100 0111
- Assembler allows programmer to use mnemonic codes…
  - such as ADD for addition
- Assembly language an aid to programming.

- 1957 Grace Hopper developed first high-level programming language called **FLOWMATIC.**
- In same year, IBM developed FORTRAN（**FORmula TRANslator**) for its systems.
  - Allowed programmers to develop programs that used formulas to solve mathematical problems.
- First successful, widespread programming language for business applications was COBOL (**COmputer Business Oriented Language**).

Grace Hopper (1906-1992)



first bug (1947)

The Microprocessor Age——
The Humble Beginnings: Intel 4004 (1971)

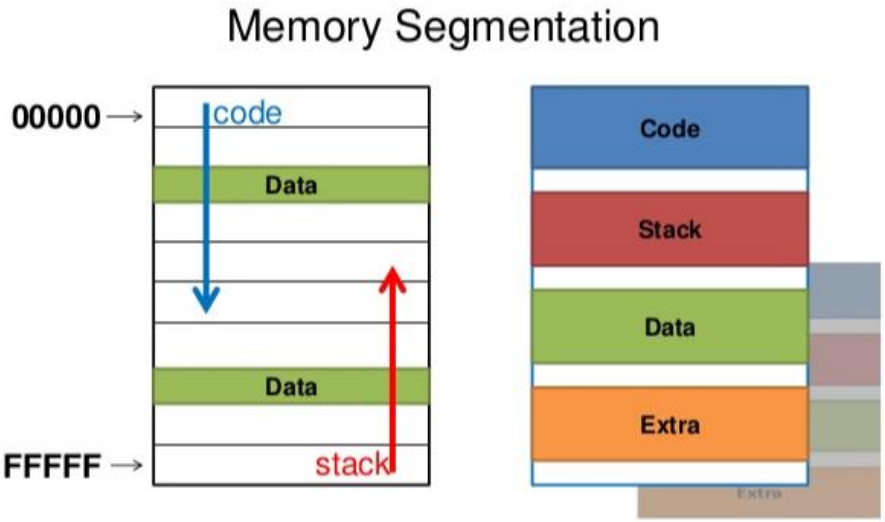
- World's first microprocessor the Intel 4004.
- A 4-bit microprocessor-programmable controller on a chip.
- Addressed 4096, 4-bit-wide memory locations.
  - a **bit** is a binary digit with a value of one or zero
  - 4-bit-wide memory location often called a **nibble**
- The 4004 instruction set contained 45 instructions.

- Fabricated with then-current state-of-the-art P-channel MOSFET technology.
- Executed instructions at 50 KIPs (**kilo-instructions per second**).
  - slow compared to 100,000 instructions per second by 30-ton ENIAC computer in 1946
- Difference was that 4004 weighed less than an ounce.
- 4-bit microprocessor debuted in early game systems and small control systems.
  - early shuffleboard game produced by Bailey

The Modern Microprocessor——
The 8086: The Real Game Changer (1978)

- The IA-32 (Intel Architecture 32) family was preceded by 16-bit processors, the 8086 and 8088.
- In 1978 Intel released the 8086 with 8087 math co-processor; a year or so later, it released the 8088.
- The 8086 has 16-bit registers and a 16-bit external data bus, with 20-bit addressing giving a 1-MByte address space.
- The 8088 is similar to the 8086 except it has an 8-bit external data bus.
- Popularity of Intel ensured in 1981 when IBM chose the 8088 in its personal computer.

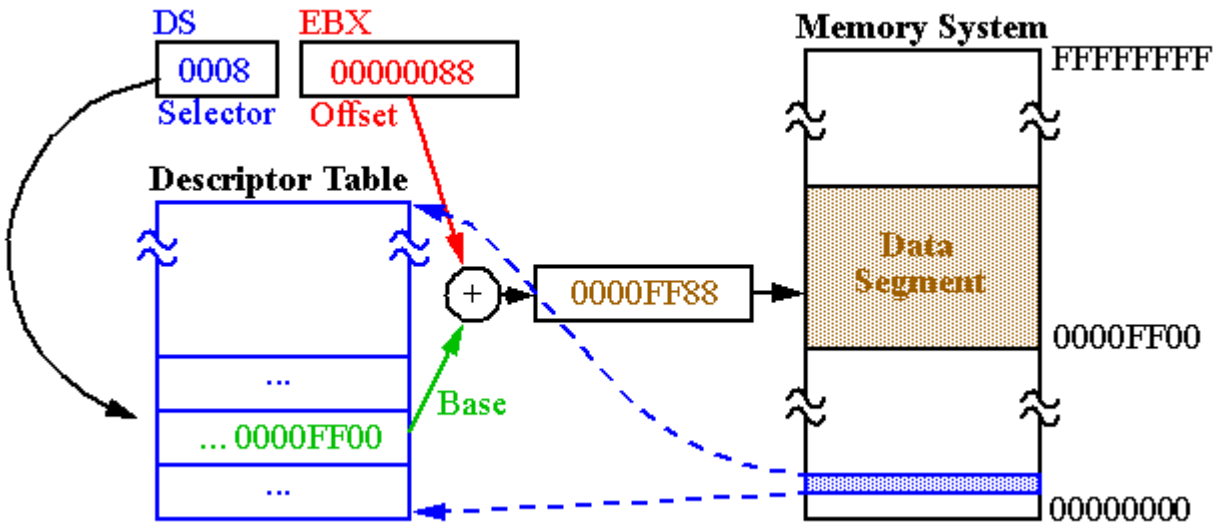
16-bit Processors and Segmentation



- The 8086/8088 introduced segmentation to the IA-32 architecture. With segmentation, a 16-bit segment register contains a pointer to a memory segment of up to 64 KBytes.
- Using four segment registers at a time, 8086/8088 processors are able to address up to 256 KBytes without switching between segments.

The Intel 286 Processor——
The Final 16-bit x86 Processor (1982)

- The Intel 286 processor introduced protected mode operation into the IA-32 architecture.
- Protected mode uses the segment register content as selectors or pointers into descriptor tables.



Protected Mode Memory Addressing

The Intel 286 Processor

- Descriptors provide 24-bit base addresses with a physical memory size of up to 16 MBytes, support for virtual memory management on a segment swapping basis, and a number of protection mechanisms.
- Protection mechanisms include:
  - Four privilege levels
  - Segment limit checking
  - Read-only and execute-only segment options

The Intel 386 Processor——
A Leap in Architecture (1985)

- The Intel 386 processor was the first 32-bit processor in the IA-32 architecture family. It introduced 32-bit registers for use both to hold operands and for addressing.
- The lower half of each 32-bit Intel 386 register retains the properties of the 16-bit registers of earlier generations, permitting backward compatibility.

The Intel 386 Processor

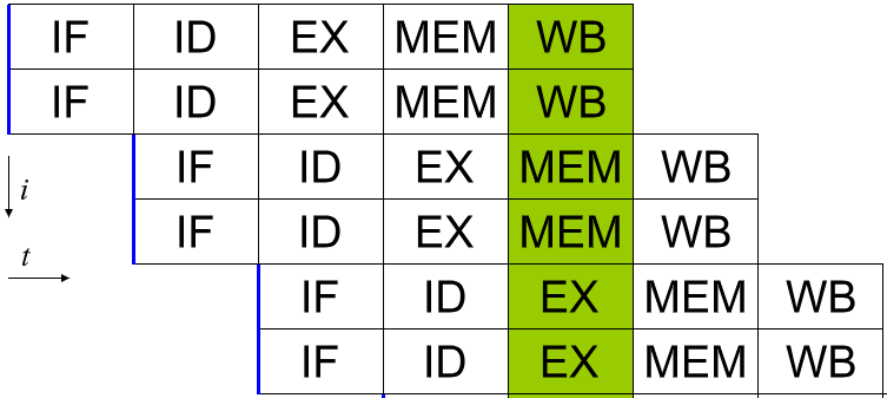
- The processor also provides a virtual-8086 mode that allows for even greater efficiency when executing programs created for 8086/8088 processors.
- The 386 was the first processor in the Intel family to include parallel stages (pipelining) in its execution cycle.
- In addition, the 386 processor has support for paging with a fixed 4-KByte page size providing a method for virtual memory management.

The Intel 486 Processor——
The First x86 CPU with L1 cache (1989)

- The 486 processor added more parallel execution capability by expanding the 386 processor's instruction decode and execution units into five pipelined stages.
- In addition, the 486 added:
  - An 8-KByte on-chip first-level cache that increased the percent of instructions that could execute at the scalar rate of one per clock
  - An integrated x87 FPU (floating point unit)

The Intel Pentium Processor—— Fifth-Generation x86 Chip (1993)

- Pentium processor added a second execution pipeline to achieve superscalar performance (two pipelines, known as u and v, execute two instructions per clock).



superscalar

- The first-level cache doubled, with 8 KBytes devoted to code and another 8 KBytes devoted to data.
- The data cache uses the MESI protocol to support more efficient write-back cache in addition to the previous used write-through cache.
- Branch prediction with an on-chip branch table was added to increase performance in looping constructs.

The P6 Family of Processors——
Better Performance for Multimedia (1995-1999)

- The **Pentium Pro** processor is three-way superscalar.
- The **Pentium II** processor added MMX technology to the P6 family processors along with new packaging and several hardware enhancements.
- The **Pentium III** processor introduced the Streaming SIMD Extensions (SSE) to the IA-32 architecture. SSE extensions expand MMX technology by providing a new set of 128-bit registers and the ability to perform SIMD operations on packed single-precision floating-point values.

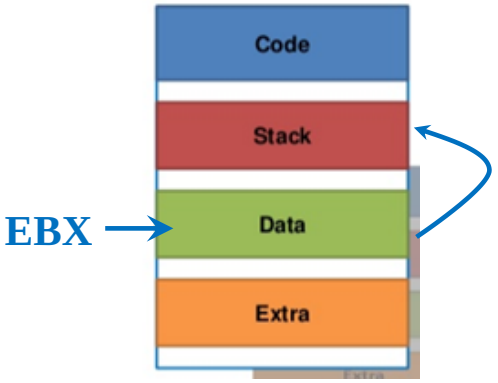
The Pentium 4 Processor Family (2000-2006)

- The Intel Pentium 4 processor is based on NetBurst microarchitecture and introduced:
  - Streaming SIMD Extensions 2 (SSE2)
  - Intel 64 architecture
  - Hyper-Threading Technology
  - Intel Virtualization Technology (Intel VT)

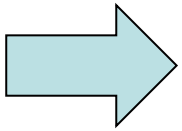
- The Intel Pentium Processor Extreme Edition (2005) introduced dual-core technology. The processor supports SSE, SSE2, SSE3, Hyper-Threading Technology, and Intel 64 architecture.

More Information on Specific Advances

- Starting with Pentium Pro, Intel redesigned it's microprocessors and used internal RISC core under the CISC instructions.
- Since Pentium Pro, all CISC instructions are divided into microcode and then executed by the RISC core.



POP [EBX]



LOAD TMP , [ESP]
STORE [EBX] , TMP
ADD ESP , 4

X86 instruction (CISC)

microcode (RISC)

Out-of-order Execution（1/2）

- Superscalar processor exploits instruction level parallelism. However, only independent subsequent instructions can be executed in parallel. Whereas subsequent instructions are often dependent.
- So the utilization of the second pipeline is often low.
- Solution: Out-of-order Execution
  - Execute instructions based on the data flow graph, rather than program order.
  - Look ahead in a window of instructions and find instructions that are ready to execute.
  - Start instruction execution before execution of a previous instructions.

Out-of-order Execution（2/2）

Example:

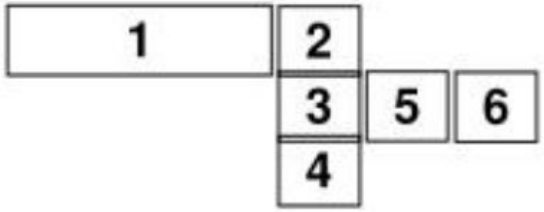
```
(1)  r1 ← r4 / r7 ;
(2)  r8 ← r1 + r2
(3)  r5 ← r5 + 1
(4)  r6 ← r6 - r3 ;
(5)  r4 ← r5 + r6
(6)  r7 ← r8 * r4
```

assume divide takes 20 cycles

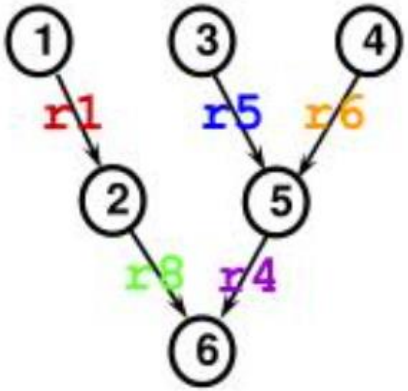
Look ahead window = 4

Three-way superscalar

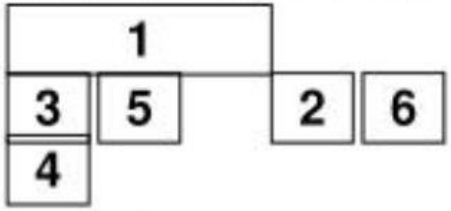
In-order execution



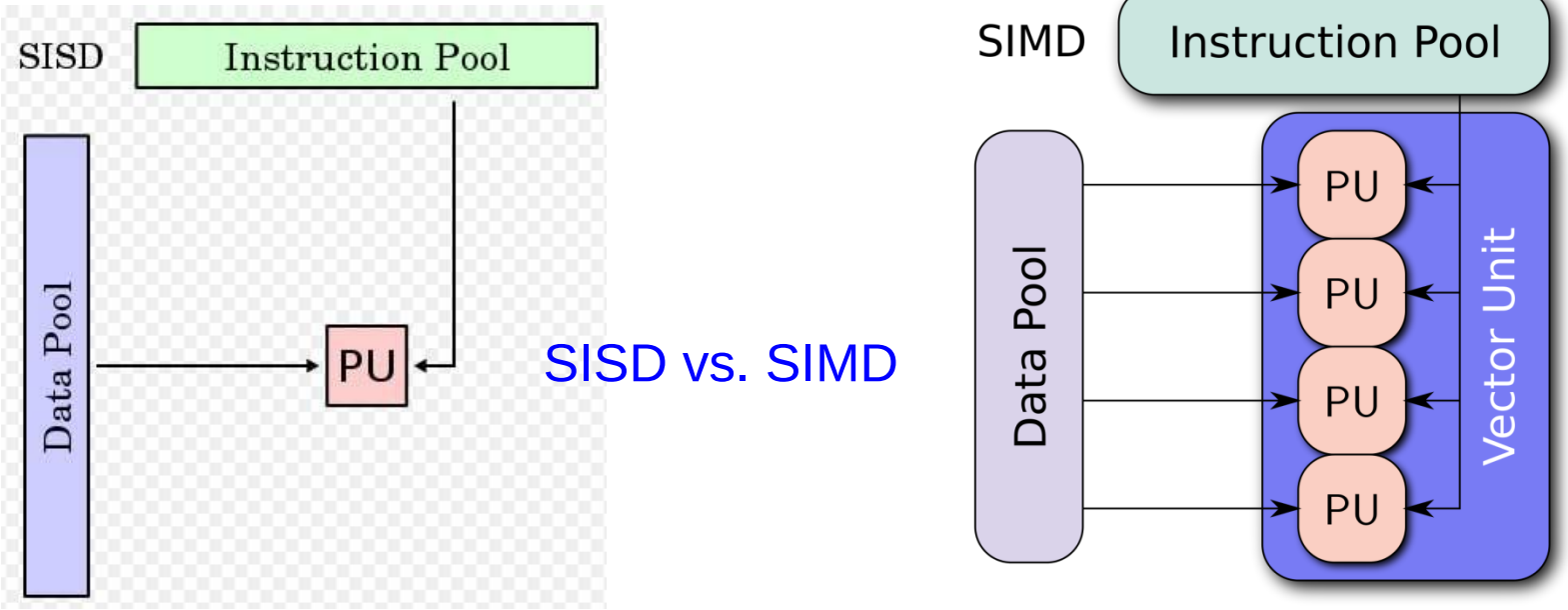
Data Flow Graph



Out-of-order execution

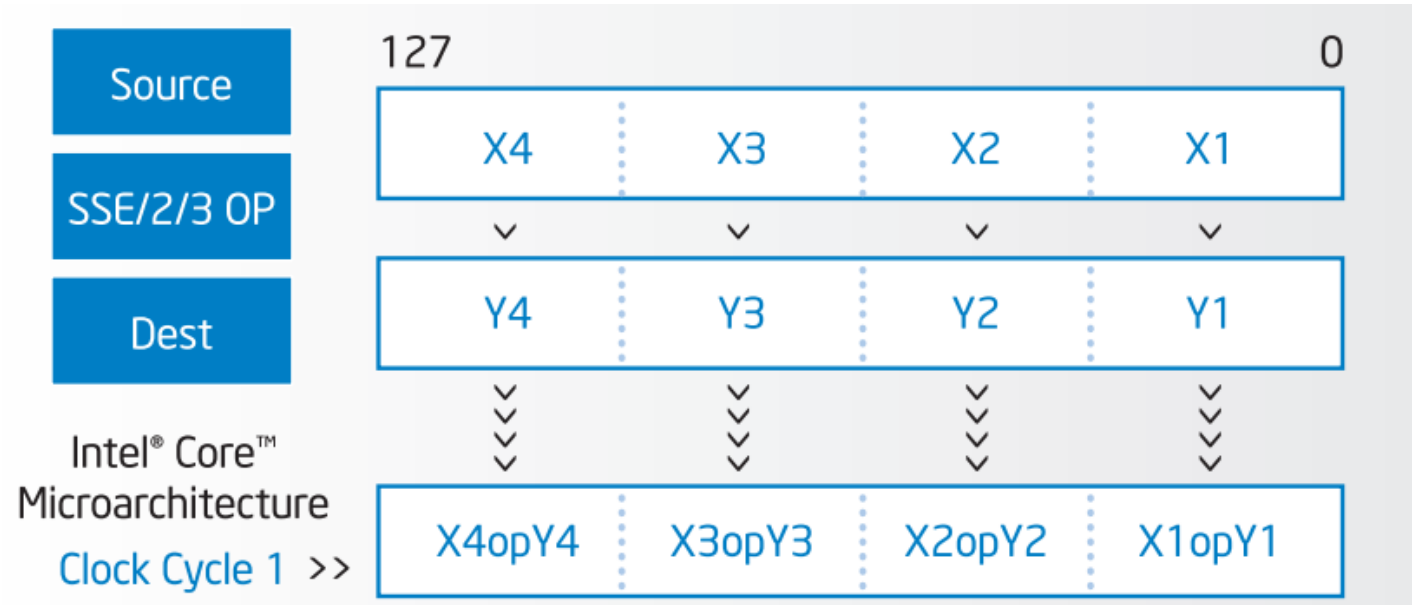


SIMD Instructions（1/3）



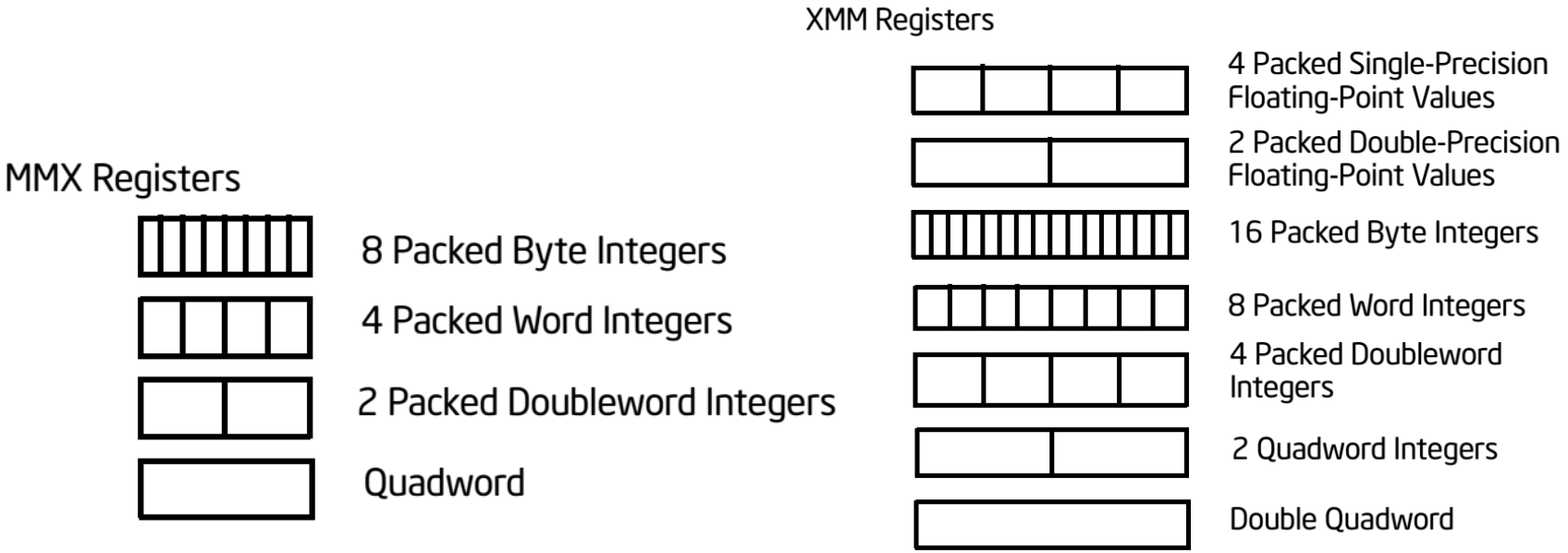
- Compared with SISD, a SIMD computing element performs the same operation on multiple data items simultaneously. Typical SIMD operations include basic arithmetic (addition, subtraction, multiplication, and division), shifts, compares, and data conversions.

SIMD Instructions（2/3）

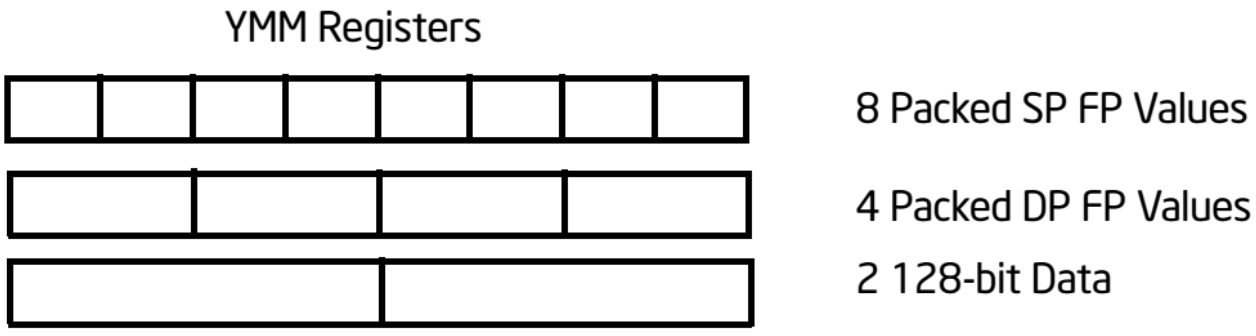


- Since MMX technology in the Pentium II, six extensions (SSE, SSE2,SSE3, SSSE3,SSE4 and AVX) have been introduced into the Intel 64 and IA-32 architectures to perform SIMD operations.

SIMD Instructions（3/3）

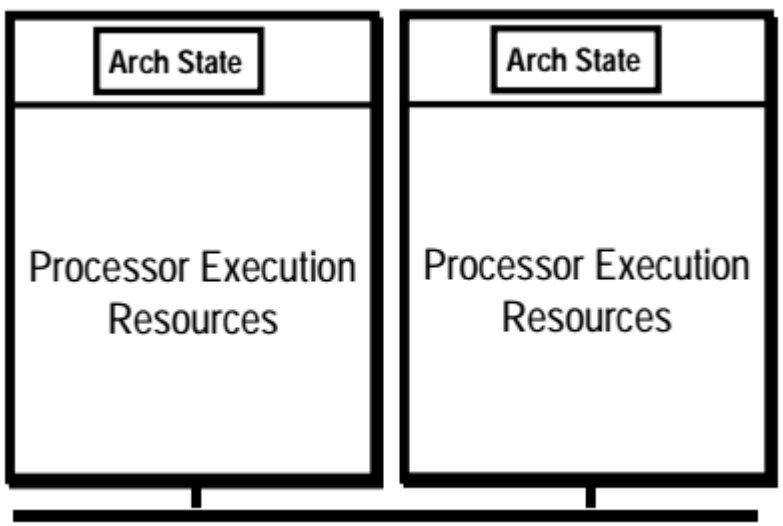


MMX technology　　SSE

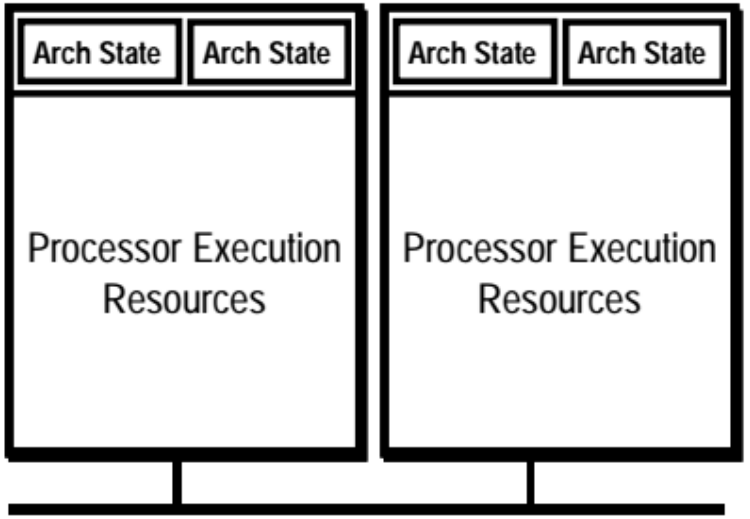


AVX

Hyper-Threading Technology（1/3）



Processors without HT



Processors with HT

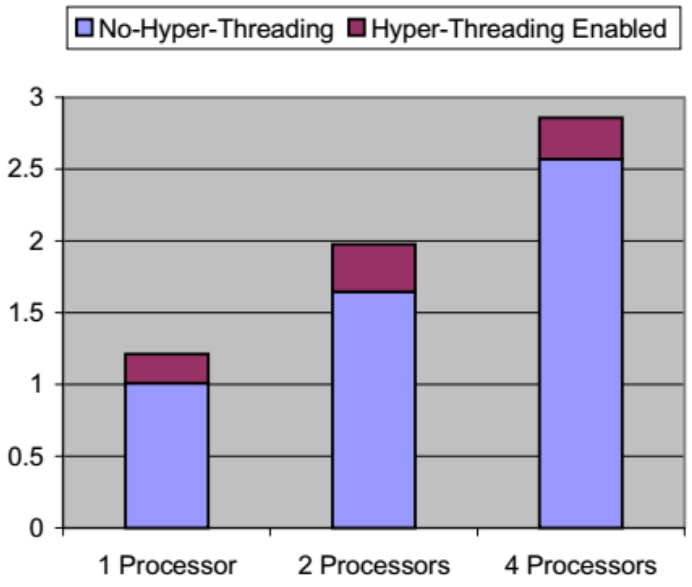
- Three goals of HT technology:
  - To minimize the die area cost of implementing HT technology.
  - When one logical processor is stalled the other could continue.
  - To run at the same speed with HT technology as without this capability when only one software thread is active.

Hyper-Threading Technology（2/3）

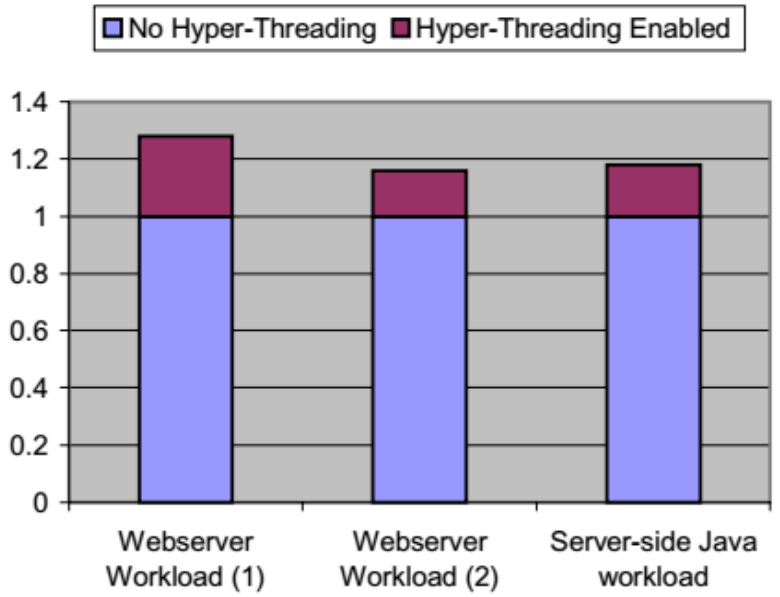
- HT technology provides hardware multi-threading capability with a single physical package by using shared execution resources in a processor core.
- An IA-32 processor that supports HT technology consists of two or more logical processors, each of which has its own IA-32 architectural state.
- The architecture state consists of registers including the general-purpose registers, the control registers, the advanced programmable interrupt controller (APIC) registers, and some machine state registers.
- Logical processors share nearly all other resources on the physical processor, such as caches, execution units, branch predictors, control logic, and buses.

Hyper-Threading Technology（3/3）

- Three goals were achieved through efficient logical processor selection algorithms and the partitioning and recombining algorithms of many key resources with less than 5% of the total die area to obtain up to 30% gain.



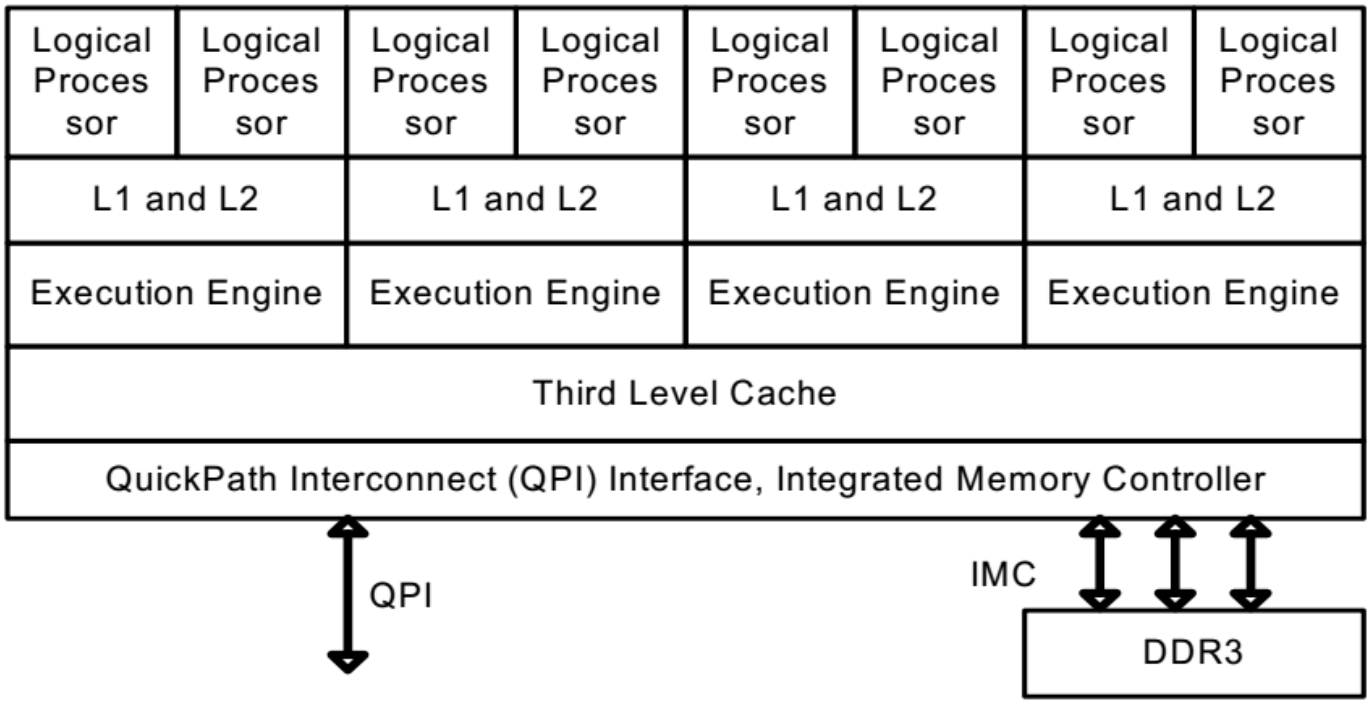
OLTP workload



Web server benchmark performance

Hyper-Threading Technology Architecture and Microarchitecture. Intel Technology Journal, 2002.

Multi-Core Technology



Intel Core i7 Processor (quad-core + Hyper-Threading)

- Multi-core technology enhances hardware multi-threading capability by providing two or more execution cores in a physical package.

Intel 64 Architecture

- Intel 64 architecture increases the linear address space for software to 64 bits and supports physical address space up to 52 bits.
- The technology also introduces a new operating mode referred to as IA-32e mode.
- IA-32e mode operates in one of two sub-modes
  - Compatibility mode enables a 64-bit operating system to run most legacy 32-bit software unmodified.
  - 64-bit mode enables a 64-bit operating system to run applications written to access 64-bit address space.

1–3 NUMBER SYSTEMS

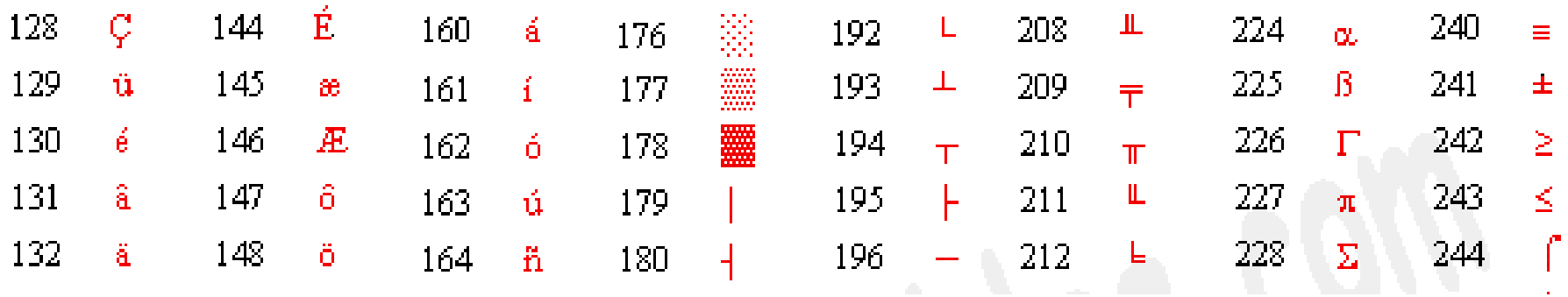
- Use of a microprocessor requires working knowledge of numbering systems.
  - binary, decimal, and hexadecimal
- This section provides a background for these numbering systems.
- Conversions are described.
  - decimal and binary
  - decimal and hexadecimal
  - binary and hexadecimal

1–4 COMPUTER DATA FORMATS

- Successful programming requires a precise understanding of data formats.
- Commonly, data appear as ASCII, Unicode, BCD, signed and unsigned integers, and floating-point numbers (real numbers).
- Other forms are available but are not commonly found.

ASCII and Unicode Data

- ASCII (**American Standard Code for Information Interchange**) data represent alphanumeric characters in computer memory.
- Standard ASCII code is a 7-bit code.
  - eighth and most significant bit used to hold parity
- If used with a printer, most significant bits are 0 for alphanumeric printing; 1 for graphics.
- In PC, an extended ASCII character set is selected by placing 1 in the leftmost bit.



128	Ç	144	É	160	á	176	▒	192	└	208	╨	224	α	240	≡
129	ü	145	æ	161	í	177	▒	193	┴	209	╤	225	ß	241	±
130	é	146	Æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
131	â	147	ô	163	ú	179	│	195	├	211	╙	227	π	243	≤
132	ä	148	ö	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠

- Extended ASCII characters store:
  - some foreign letters and punctuation
  - Greek & mathematical characters
  - box-drawing & other special characters
- Extended characters can vary from one printer to another.
- ASCII control characters perform control functions in a computer system.
  - clear screen, backspace, line feed, etc.

- Many Windows-based applications use the **Unicode** system to store alphanumeric data.
  - stores each character as 16-bit data
- Codes 0000H–00FFH are the same as standard ASCII code.
- Remaining codes, 0100H–FFFFH, store all special characters from many character sets.
- Allows software for Windows to be used in many countries around the world.
- For complete information on Unicode, visit: *http://www.unicode.org*

Binary Coded Decimal (BCD)

- The range of a BCD digit extends from 0000_2 to 1001_2, for 0–9 decimal. BCD can be stored in two forms.
- Stored in packed BCD form:
  - packed BCD data stored as two digits per byte;
  - used for BCD addition and subtraction in the instruction set of the microprocessor or BCD counting
- Stored in unpacked BCD form:
  - unpacked BCD data stored as one digit per byte
  - returned from a keypad or keyboard

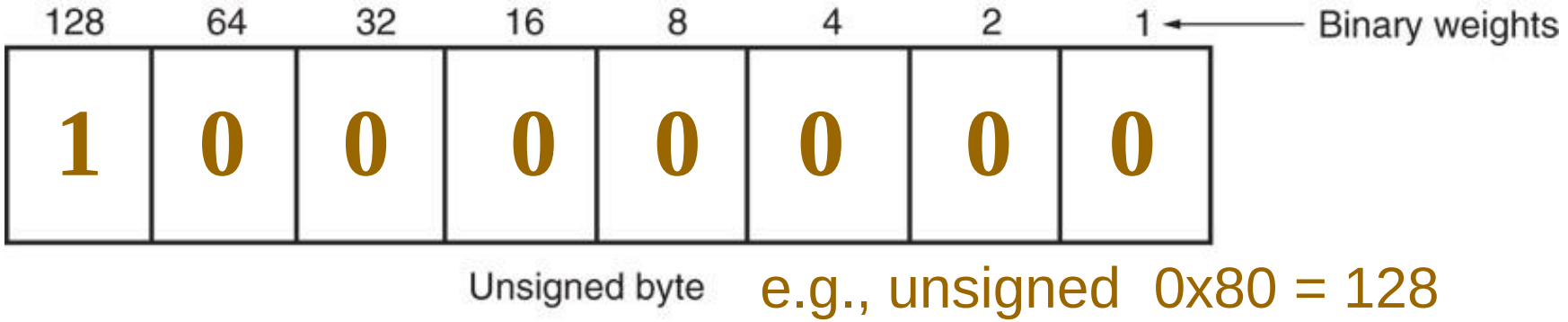
Decimal	packed BCD	unpacked BCD
12	0001 0010	0000 0001 0000 0010
96	1001 0110	0000 1001 0000 0110

- Applications requiring BCD data are point-of-sales terminals.
  - also devices that perform a minimal amount of simple arithmetic
- If a system requires complex arithmetic, BCD data are seldom used.
  - there is no simple and efficient method of performing complex BCD arithmetic

Byte-Sized Data

- Stored as *unsigned* and *signed* integers.
- Difference in these forms is the weight of the leftmost bit position.
  - value 128 for the unsigned integer
  - *minus* 128 for the signed integer
- In signed integer format (signed 2's complement), the leftmost bit represents the sign bit of the number.
  - also a weight of *minus* 128

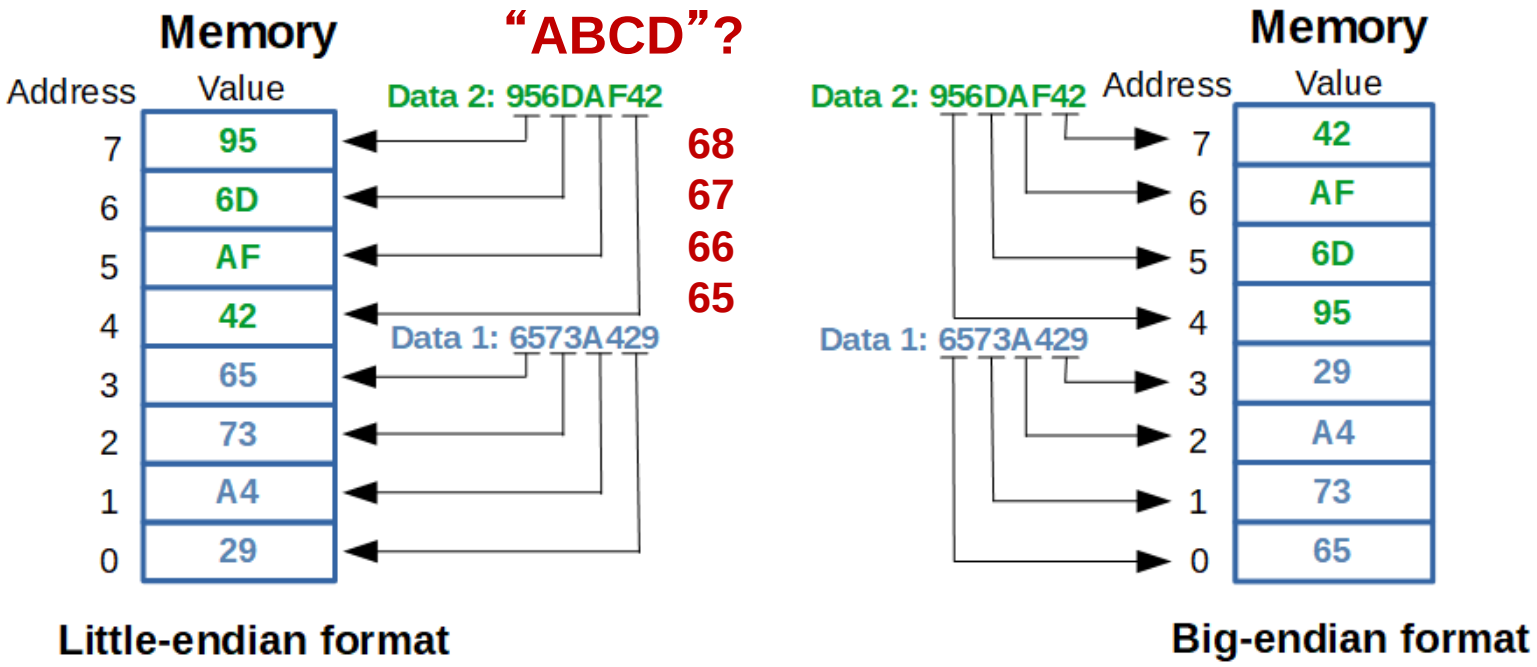
Figure 1–14 The unsigned and signed bytes illustrating the weights of each binary-bit position.



- Unsigned integers range 00H to FFH (0–255)
- Signed integers from –128 to 0 to + 127.
- Negative signed numbers represented in this way are stored in the two's complement form.
- Evaluating a signed number by using weights of each bit position is much easier than the act of two's complementing a number to find its value.
  - especially true in the world of calculators designed for programmers

Word-Sized Data

- A word (16-bits) is formed with two bytes of data. The data is stored in little-endian format:
  - The least significant byte always stored in the lowest-numbered memory location.
  - Most significant byte is stored in the highest.



Little-endian vs Big-endian format

- Alternate method is called the **big endian** format.
- Numbers are stored with the lowest location containing the most significant data.
- Not used with Intel microprocessors.
- The big endian format is used with the Motorola family of microprocessors.

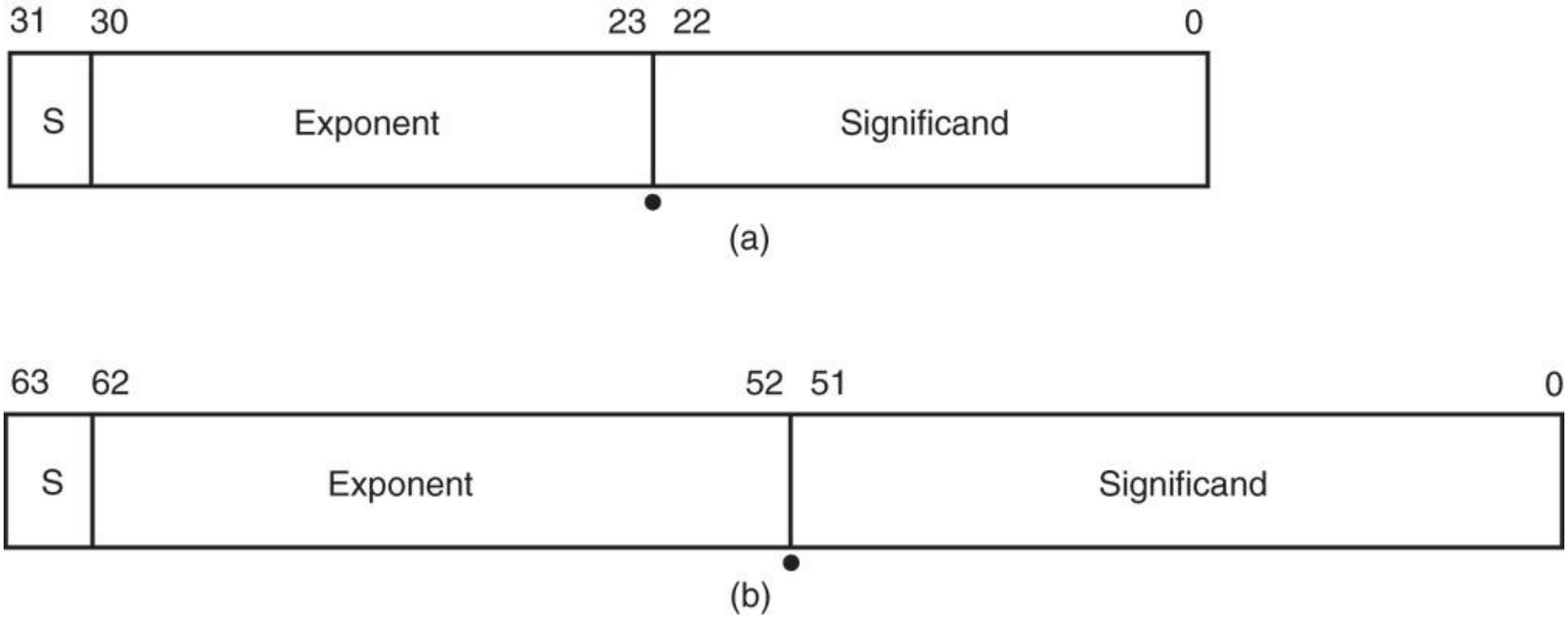
Doubleword-Sized Data

- **Doubleword-sized data** requires four bytes of memory because it is a 32-bit number.
  - appears as a product after a multiplication
  - also as a dividend before a division
- Define using the assembler directive **define doubleword(s)**, or **DD.**
  - also use the DWORD directive in place of DD

Real Numbers

- Since many high-level languages use Intel microprocessors, real numbers are often encountered.
- A real, or a **floating-point number** contains two parts:
  - a mantissa, significand, or fraction
  - an exponent.
- A 4-byte number is called **single-precision.**
- The 8-byte form is called **double-precision.**

Figure 1–17 The floating-point numbers in (a) single-precision using a bias of 7FH and (b) double-precision using a bias of 3FFH.



- The assembler can be used to define real numbers in single- & double-precision forms:
  - use the DD directive for single-precision 32-bit numbers
  - use **define quadword(s)**, or DQ to define 64-bit double-precision real numbers
- Optional directives are REAL4, REAL8, and REAL10.
  - for defining single-, double-, and extended precision real numbers
- The Microsoft Macro Assembler (MASM) is an x86 assembler that uses the Intel syntax.

MASM Assembler Directive for Data Allocation

Directives	Size in bits	Data types
BYTE (DB)	8	Unsigned integer
SBYTE	8	Signed integer
WORD (DW)	16	Unsigned integer
SWORD	16	Signed integer
DWORD (DD)	32	Unsigned integer
SDWORD	32	Signed integer
FDWORD	48	Unsigned integer
QWORD (DQ)	64	Unsigned integer
TBYTE (DT)	80	Unsigned integer
REAL4	32	Single-precision FP (IEEE 754)
REAL8	64	Double-precision FP (IEEE 754)
REAL10	80	Extended FP

The IEEE 754 Format

- In the 1960's and 1970's, each line of computers supported its own range and precision for its floating point numbers, and rounded off arithmetic operations in its own peculiar way.
- IEEE 754 standard was officially adopted in 1985 based on Kahan's design model.

William Morton Kahan
1933-
1989 Turing Prize

- Kahan won the ACM's Turing Award in 1989 for his work in creating IEEE 754 standard, and he has often been called "The Father of Floating Point".

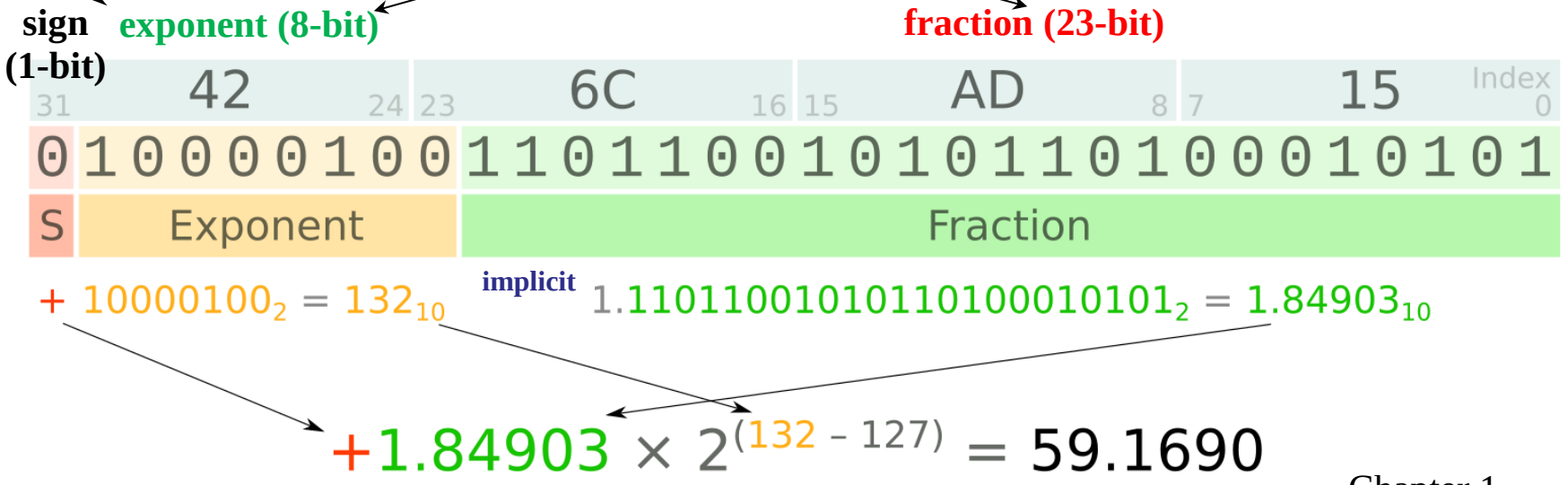
From: An Interview with the Old Man of Floating-Point

The IEEE 754 Format

- IEEE 754 uses the idea of scientific notation:
  - sign bit: 0, the number is positive; 1, the number is negative;
  - biased exponent: a bias is added to the exponent for easy comparisons. All 0s and all 1s denote special values.
  - normalized fraction/mantissa: The mantissa for normalized number has a hidden bit which is not stored.

$$(-1)^{\text{sign bit}}(1 + \text{fraction}) \times 2^{\text{exponent - bias}}$$

E.g., binary32



Why Not 2's Complement?

- The IEEE 754 standard chose sign-magnitude representation, rather than the 2's complement method, for the following reasons:
  - There is no "unsigned" floating point, so there's no benefit (savings) to using 2's complement.
  - 2s complement has major asymmetry problems (there are more representable values <0 than >0) that causes stability issues (e.g., rounding or potential loss of precision).
  - Having separate +/-0 is crucial for approximations. A few arithmetic operations are affected by zero's sign.
  - Some operations will be easy (e.g., comparison).

https://stackoverflow.com/questions/57016843/why-not-use-a-twos-complement-based-floating-point

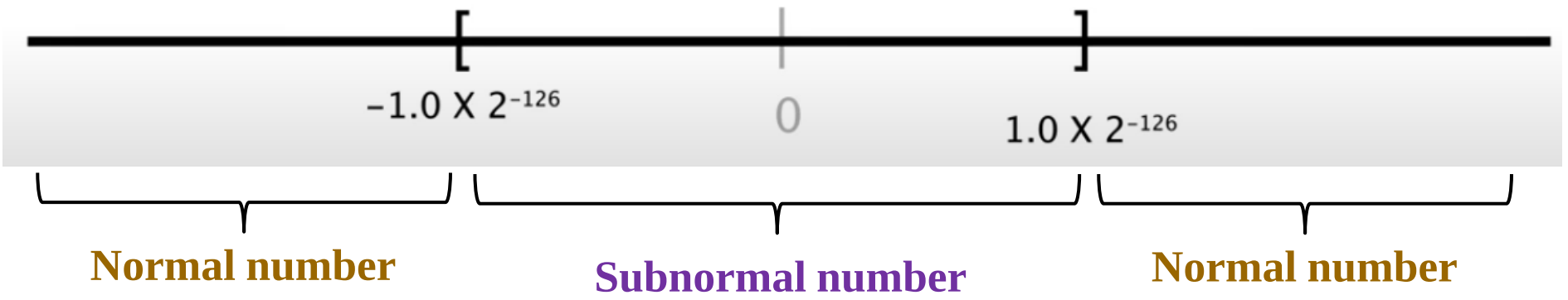
Special Values in IEEE 754

- Special values with all 0s or 1s in the exponent field:
  - Zero: sign bit = 0 and 1 for +0 and -0; biased exponent = all 0 bits; and the fraction = all 0 bits;
  - Infinity: sign bit = 0 and 1 for positive and negative infinity; biased exponent = all 1 bits; and the fraction = all 0 bits;
  - NaN (Not-A-Number): sign bit = 0 or 1; biased exponent = all 1 bits; and the fraction is anything but all 0 bits.

```
0 00000000 00000000000000000000000 = +0
1 00000000 00000000000000000000000 = - 0
0 11111111 00000000000000000000000 = +Infinity
1 11111111 00000000000000000000000 = - Infinity
0 11111111 00000000000001000001000 = NaN
```

Normal and Subnormal Number

- Normal number
  - Exponent field is not all 0s or 1s.



- Subnormal number or denormalized number
  - subnormal is represented by having a zero exponent field with a non-zero significand field.
  - subnormal = $(-1)^{\text{sign}} \times \text{fraction} \times 2^{1-\text{bias}}$, e.g., 0.11×2^{-126}
  - subnormal numbers may dramatically increase latency.

The IEEE 754 Format

Parameter	binary16	binary32	binary64	binary128	binary{k} ($k \geq 128$)
k, storage width in bits	16	32	64	128	multiple of 32
p, precision in bits	11	24	53	113	$k - \text{round}(4 \times \log_2(k)) + 13$
emax, maximum exponent e	15	127	1023	16383	$2^{(k-p-1)} - 1$
Encoding parameters					
bias, $E - e$	15	127	1023	16383	*emax*
sign bit	1	1	1	1	1
w, exponent field width in bits	5	8	11	15	$\text{round}(4 \times \log_2(k)) - 13$
t, trailing significand field width in bits	10	23	52	112	$k - w - 1$
k, storage width in bits	16	32	64	128	$1 + w + t$

Binary interchange format parameters

From IEEE 754-2019

The IEEE 754 Format

Parameter	decimal32	decimal64	decimal 128	decimal{k} ($k \geq 32$)
k, storage width in bits	32	64	128	multiple of 32
p, precision in digits	7	16	34	$9 \times k/32 - 2$
emax	96	384	6144	$3 \times 2^{(k/16+3)}$
Encoding parameters				
bias, $E - q$	101	398	6176	$emax + p - 2$
sign bit	1	1	1	1
w+5, combination field width in bits	11	13	17	$k/16 + 9$
t, trailing significand field width in bits	20	50	110	$15 \times k/16 - 10$
k, storage width in bits	32	64	128	$1 + 5 + w + t$

Decimal interchange format parameters

From IEEE 754-2019

The IEEE 754 Format

- The IEEE 754 includes five rounding modes:
  - default: roundTiesToEven
  - optional: roundTiesToAway、roundTowardZero、roundTowardPositive、roundTowardNegative

Mode / Example Value	+11.5	+12.5	−11.5	−12.5
to nearest, ties to even	+12.0	+12.0	−12.0	−12.0
to nearest, ties away from zero	+12.0	+13.0	−12.0	−13.0
toward 0	+11.0	+12.0	−11.0	−12.0
toward +∞	+12.0	+13.0	−11.0	−12.0
toward −∞	+11.0	+12.0	−12.0	−13.0

Floating-Point Operation—Addition

- For example (using decimal radix with 7 digit precision):

  $123456.7 + 101.7654$

  $= (1.234567 \times 10^5) + (1.017654 \times \mathbf{10^2})$

  $= (1.234567 \times 10^5) + (0.001017654 \times \mathbf{10^5})$ **shifting**

  $= (1.234567 + 0.001017654) \times 10^5$ **addition**

  $= 1.235584654 \times 10^5$ **(true sum)**

  $= 1.235585 \times 10^5$ **rounding**

Accuracy Problem（1/2）

```
int main()
{
    float sum = 0.0;

    float x = 1.0;

    for (int i = 0; i < N; i++) {
        sum += x;
    }
    printf("%1.0f\n", sum);

    return 0;
}
```

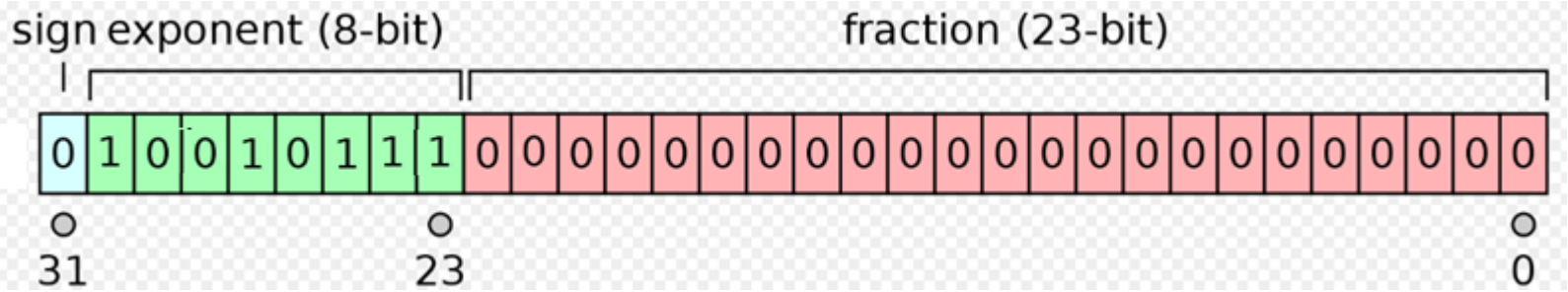
$N = 3 \times 10^6$

$sum = 3 \times 10^6$

$N = 3 \times 10^7$

$sum = 16777216$

Accuracy Problem（2/2）



16777216 + 1

$= 1 \times 2^{24} + 1$

$= 1 \times 2^{24} + (1 \times 2^{-24}) \times 2^{24}$ **shifting**

$= (1 + 2^{-24}) \times 2^{24}$ **addition**

$= \mathbf{1} \times 2^{24}$ **loss of significance**

$= 16777216$

Rewrite Floating-point Expressions

- Herbie rewrites floating point expressions to make floating point arithmetic more accurate. (https://herbie.uwplse.org).
- E.g., $\sqrt{x+1} - \sqrt{x} = \frac{1}{\sqrt{x+1}+\sqrt{x}}$

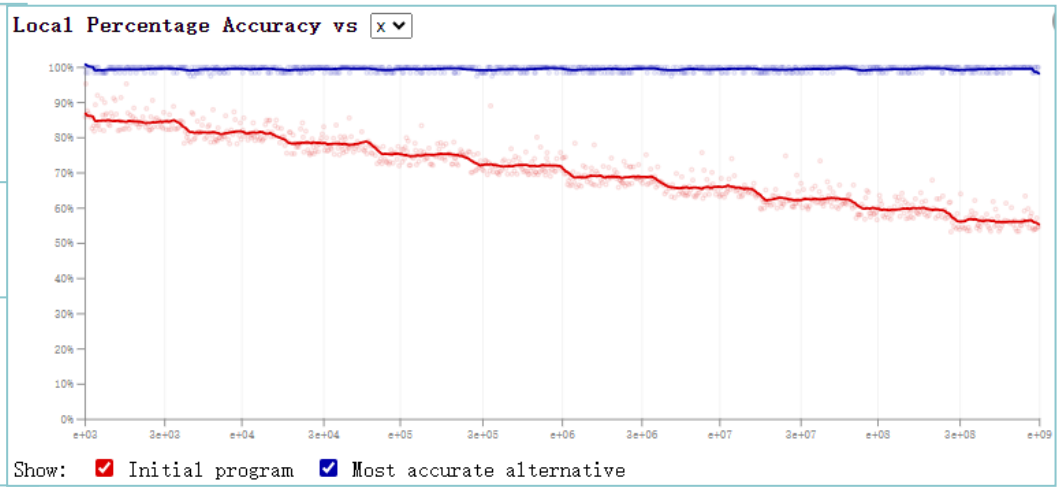
Percentage Accurate:	Time:	Alternatives:	Speedup:
70.6% → 99.5%	4.1s	7	1.0×

Initial Program: 70.6% accurate, 1.0× speedup

$\sqrt{x+1} - \sqrt{x}$

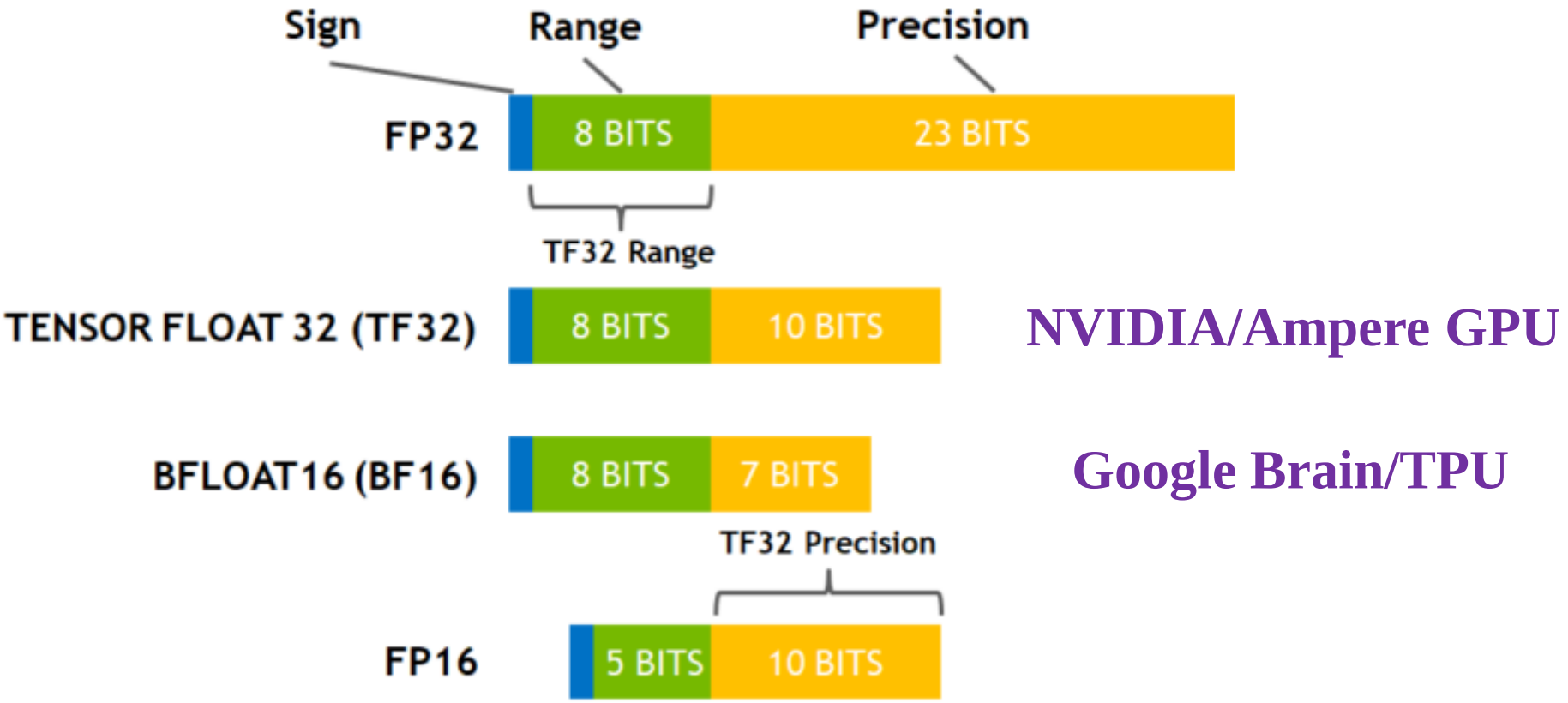
Alternative 2: 99.5% accurate, 0.7× speedup

$\frac{1}{\sqrt{x}+\sqrt{x+1}}$



New Floating-point Format for AI

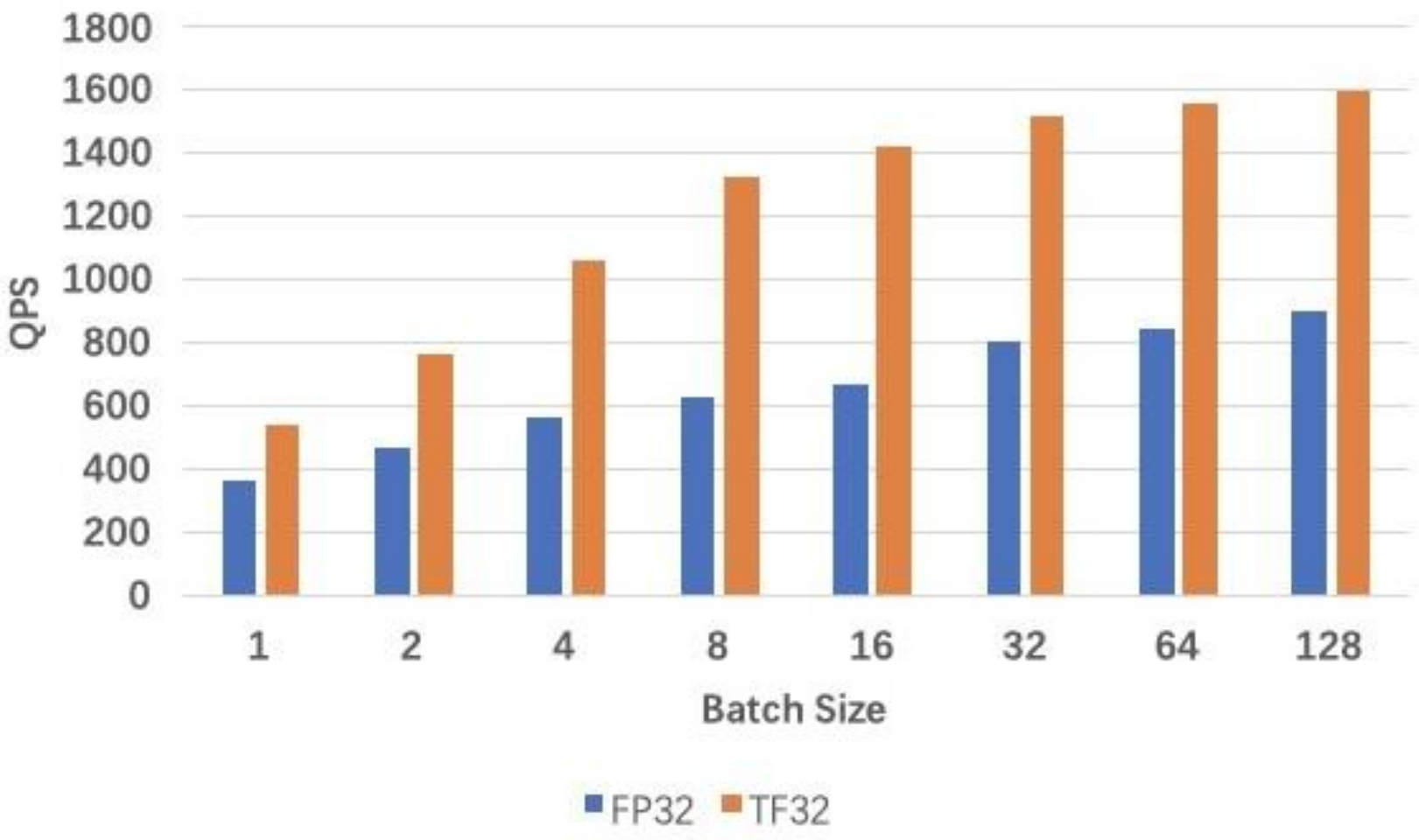
- To accelerate AI training and inference, several new floating-point formats are introduced:



New Floating-point Format for AI

NVIDIA A100 TF32/FP32 VGG16

inference performance



Assignment

1-55, 58, 59, 65, 71, 75, 78, 80, 81, 82

Appendix A: Converting Binary to Decimal

- **To convert to decimal, use decimal arithmetic to form Σ (digit $\times$ respective power of 2).**
- **Example: Convert 11010_2 to N_{10}:**

$11010_2 =>$

$1 * 2^4 = 16$

$+ 1 * 2^3 = 8$

$+ 0 * 2^2 = 0$

$+ 1 * 2^1 = 2$

$+ 0 * 2^0 = 0$

26_{10}

Converting Decimal to Binary

- **To convert from one base to another:**
  1) **Convert the Integer Part**
  2) **Convert the Fraction Part**
  3) **Join the two results with a radix point**

Conversion Details

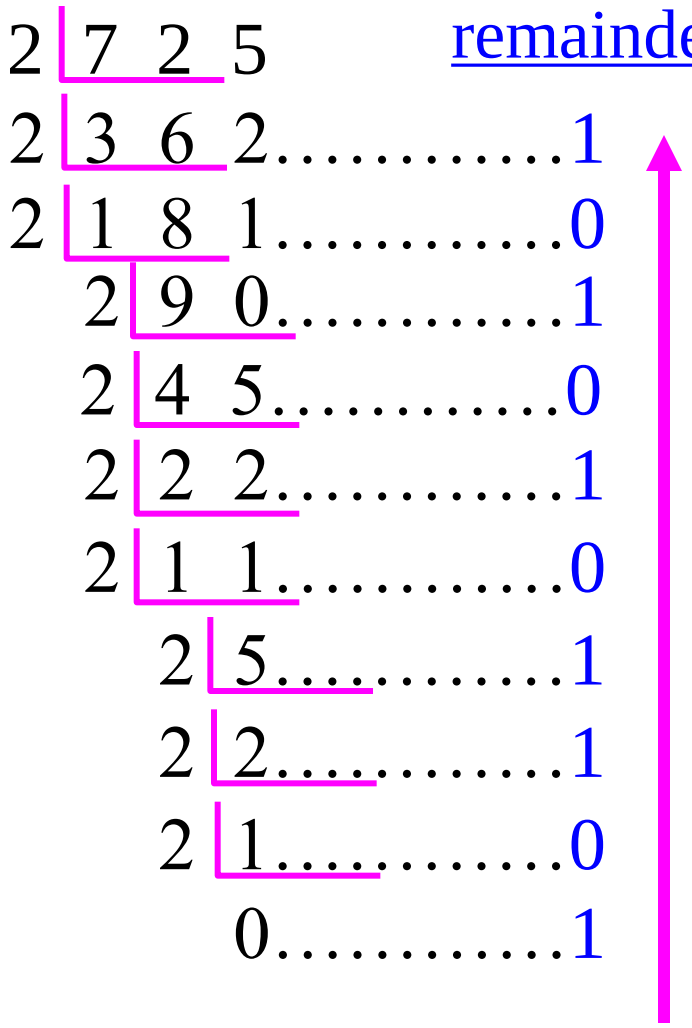
- **To Convert the Integral Part:**

  Repeatedly divide the number by the new radix and save the remainders. The digits for the new radix are the remainders in *reverse order* of their computation. If the new radix is > 10, then convert all remainders > 10 to digits A, B, …

- **To Convert the Fractional Part:**

  Repeatedly multiply the fraction by the new radix and save the integer digits that result. The digits for the new radix are the integer digits in *order* of their computation. If the new radix is > 10, then convert all integers > 10 to digits A, B, …

Example: Convert 725_{10} To Base 2



$(725)_{10} = (10\ \ 1101\ \ 0101)_2$

Example: Convert 0.678_{10} To Base 2

$(0.678)_{10} = (0.1010\ \ 1101\ \ 1001)_2$

overflow

$2 \times 0.678\ldots\ldots = 1.356$

$2 \times 0.356\ldots\ldots = 0.712$

$2 \times 0.712\ldots\ldots = 1.424$

$2 \times 0.424\ldots\ldots = 0.848$

$2 \times 0.848\ldots\ldots = 1.696$

$2 \times 0.696\ldots\ldots = 1.392$

$2 \times 0.392\ldots\ldots = 0.784$

$2 \times 0.784\ldots\ldots = 1.568$

$2 \times 0.568\ldots\ldots = 1.136$

$2 \times 0.136\ldots\ldots = 0.272$

$2 \times 0.272\ldots\ldots = 0.544$

$2 \times 0.544\ldots\ldots = 1.088$

Example: Convert 46.6875_{10} To Base 2

- **Convert 46 to Base 2**

  $46_{10} = 101110_2$

- **Convert 0.6875 to Base 2**

  $0.6875_{10} = 0.1011_2$

- **Join the results together with the radix point:**

  $46.6875_{10} = 101110.1011_2$

Octal (Hexadecimal) to Binary and Back

- **Octal (Hexadecimal) to Binary:**
  - **Restate the octal (hexadecimal) as three (four) binary digits starting at the radix point and going both ways.**
- **Binary to Octal (Hexadecimal):**
  - **Group the binary digits into three (four) bit groups starting at the radix point and going both ways, padding with zeros as needed in the fractional part.**
  - **Convert each group of three bits to an octal (hexadecimal) digit.**

Octal (Hexadecimal) to Binary and Back

- **Example:**

$(67.731)_8$ = （110 111 .111 011 001)$_2$

$(312.64)_8$=(011 001 010 . 110 1)$_2$

(11 111 101 . 010 011 11)$_2$=$(375.236)_8$

(10 110.11)$_2$=$(26.6)_8$

$(3AB4.1)_{16}$ = （0011 1010 1011 0100 .0001)$_2$

$(21A.5)_{16}$=(0010 0001 1010 . 0101)$_2$

$(1001101.01101)_2$= (0100 1101. 0110 1000)$_2$=$(4D.68)_{16}$

(110 0101.101)$_2$=$(65.A)_{16}$

Octal to Hexadecimal via Binary

- **Convert octal to binary.**
- **Use groups of four bits and convert as above to hexadecimal digits.**
- **Example: Octal to Binary to Hexadecimal**

6 3 5 . 1 7 7 $_8$

Restate : 110|011|101 . 001|111|111 $_2$

Regroup: 1|1001|1101 . 0011|1111|1(000)$_2$

Convert : 1 9 D . 3 F 8$_{16}$

- **Why do these conversions work?**

Complements

- At times, data are stored in complement form to represent negative numbers.
- Two systems used to represent negative data:
  - **radix**
  - **radix – 1** complement (earliest)

Complements

- **Two complements:**
  - **Radix Complement**
    - **r's complement for radix r**
    - **2's complement in binary**
    - **Defined as $r^n - N$**
  - **Diminished Radix Complement of N**
    - **(r – 1)'s complement for radix r**
    - **1's complement for radix 2**
    - **Defined as $(r^n - 1) - N$**
- **Subtraction is done by adding the complement of the subtrahend**
- **If the result is negative, takes its 2's complement**

Binary 1's Complement(反码)

- For $r = 2$, $N = 01110011_2$, $n = 8$ (8 digits):

  $(r^n - 1) = 256 - 1 = 255_{10}$ or 11111111_2

- The 1's complement of 01110011_2 is then:

$$\begin{array}{r} 11111111 \\ - \quad \underline{01110011} \\ 10001100 \end{array}$$

- Since the $2^n - 1$ factor consists of all 1's and since 1 – 0 = 1 and 1 – 1 = 0, the one's complement is obtained by complementing each individual bit (bitwise NOT).

Binary 2's Complement(补码)

- For $r = 2$, $N = 01110011_2$, $n = 8$ (8 digits), we have:

  $(r^n) = 256_{10}$ or 100000000_2

- The 2's complement of 01110011 is then:

$$\begin{array}{r} 100000000 \\ -\ \underline{01110011} \\ 10001101 \end{array}$$

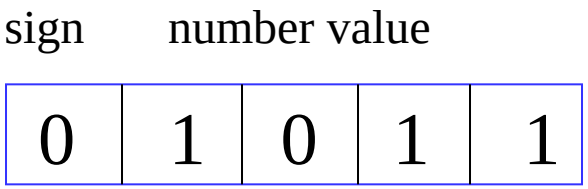
- Note the result is the 1's complement plus 1, a fact that can be used in designing hardware

Signed Integers

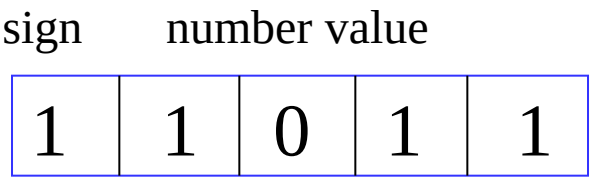
The leftmost bit represents the sign in machine number

Binary Code	Machine number

Example: +1011 →



-1011 →



Signed Integer Representations

- ***Signed-Magnitude*** **– here the n – 1 digits are interpreted as a positive magnitude.**
- ***Signed-Complement*** **– here the digits are interpreted as the rest of the complement of the number. There are two possibilities here:**
  - ***Signed 1's Complement***
    - **Uses 1's Complement Arithmetic**
  - ***Signed 2's Complement***
    - **Uses 2's Complement Arithmetic**

Signed Integer Representation Example

- r =2, n=3

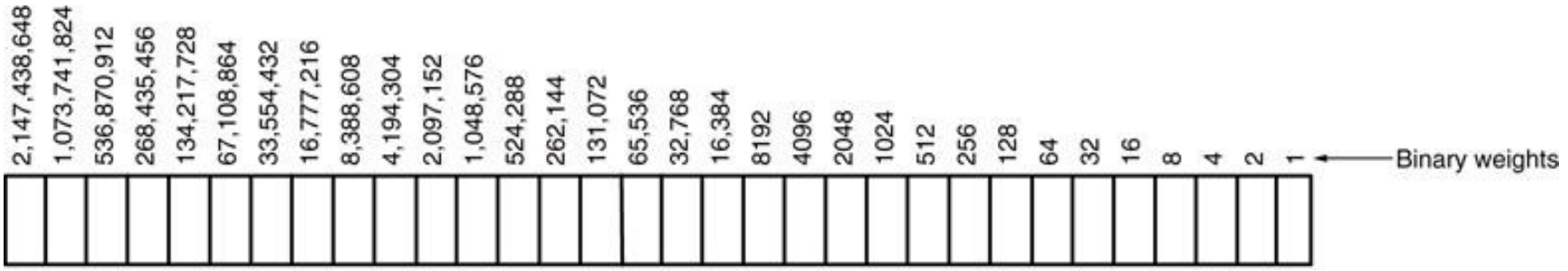
$$S2C(X) = -w_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} w_i \cdot 2^i$$

Number	Sign - Mag.	1's Comp.	2's Comp.
+3	011	011	011
+2	010	010	010
+1	001	001	001
+0	000	000	000
–0	100	111	—
–1	101	110	111
–2	110	101	110
–3	111	100	101
–4	—	—	100

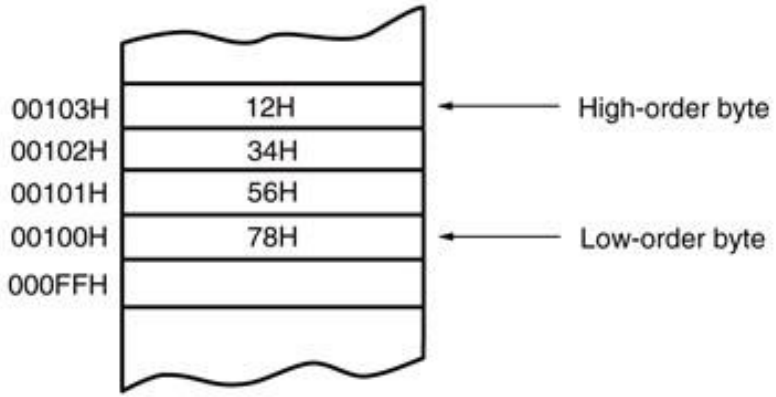
Warning: Conversion or Coding?

- **Do NOT mix up conversion of a decimal number to a binary number with coding a decimal number with a BINARY CODE.**
- **$13_{10} = 1101_2$ (This is conversion)**
- **13 $\Leftrightarrow$ 0001|0011 (This is coding)**

Figure 1–16 The storage format for a 32-bit word in (a) a register and (b) 4 bytes of memory.



(a) Unsigned doubleword



(b) The contents of memory location 00100H–00103H are the doubleword 12345678H.

SIMD and Miscellaneous Data Types

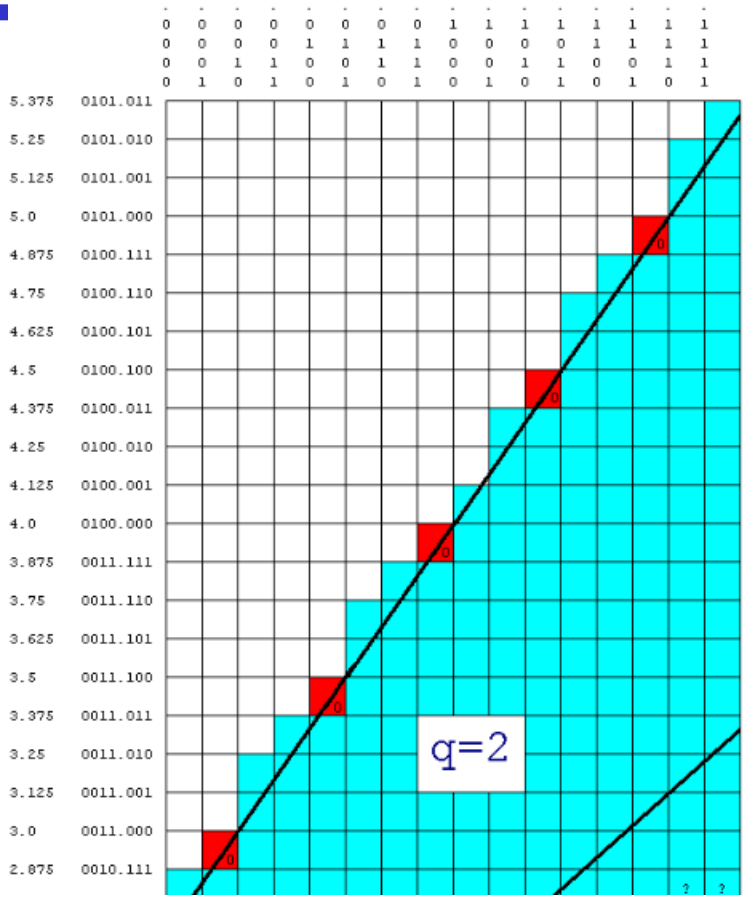
- The x86 supports a variety of packed data types, which are employed to perform SIMD calculations.

Numerical Type	xmmword	ymmword	zmmword
8-bit integer	16	32	64
16-bit integer	8	16	32
32-bit integer	4	8	16
64-bit integer	2	4	8
Single-precision floating-point	4	8	16
Double-precision floating-point	2	4	8

- The x86 platform also supports a number of miscellaneous data types including strings, bit fields, and bit strings.

The Pentium FDIV Flaw

- The Pentium processor was introduced in 1993. A year later it was discovered that in certain very rare instances a division operation returned a result that was slightly incorrect.
- The Pentium uses a SRT division algorithm for floating point divisions. However, there were 5 error cells in lookup table.
- FDIV flaw cost Intel half a billion dollars.



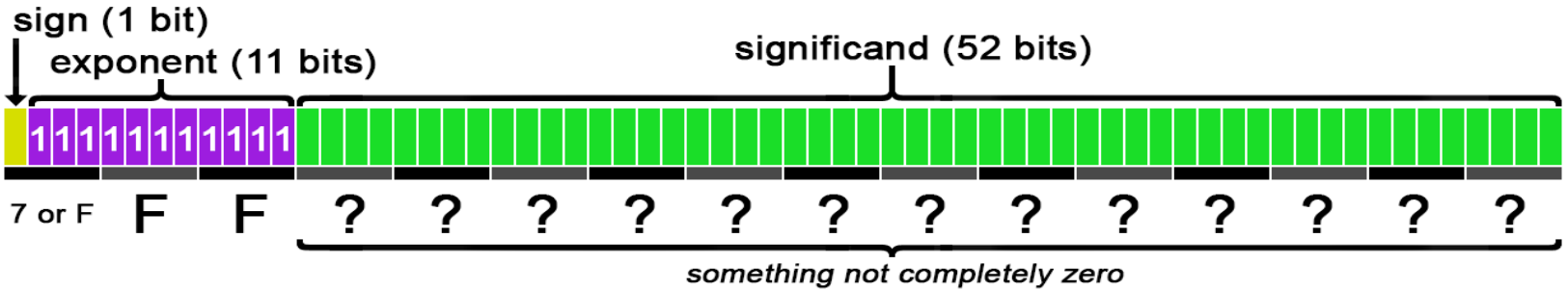
The Pentium lookup table

NVIDIA Single-chip Inference Performance

- Gains from
  - Number representation: TF32, BF16, FP16, INT8
  - Complex instructions: DP4, HMMA, IMMA
  - Process: 28nm, 16nm, 7nm

NaN-Boxing Technology

- In IEEE-754, 52 bit patterns that represent NaN can be encoded to carry data (e.g., 4-bit tag and 48-bit data).



NaN-Boxing technique

```
double ref;

double set_na(){
    if (!ref) {
        ref=0/0.;
        int  *ci = (int *)(&ref);
        char *cr = (char *)(&ref);

        ci[0]=124;
        cr[4]='a';

    }
    return ref;
}
```

markers

set markers in a NaN

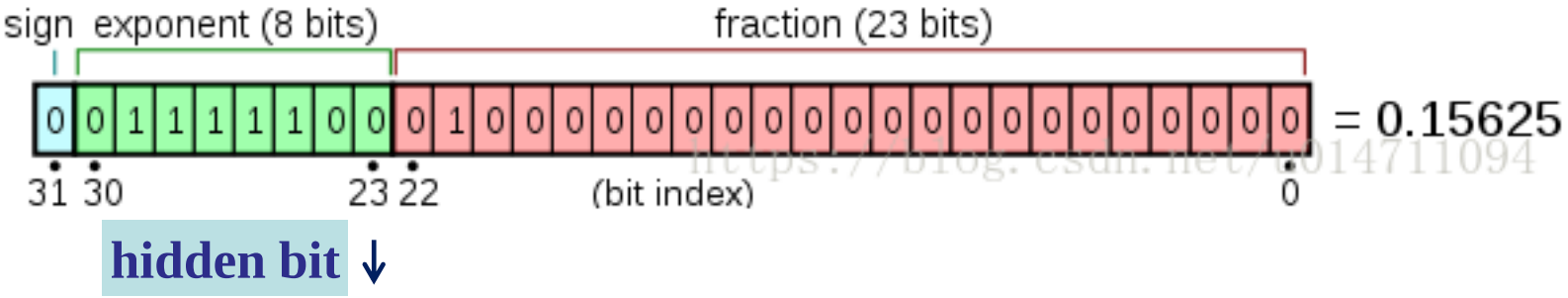
```
void read_from_na(double in){
    if (!isnan(in)) return ;

    char cc;
    char *cr = (char *)(&ref);
    cc = cr[4];

    int  ii;
    int  *ci = (int *)(&ref);
    ii = ci[0];

    printf("the integer is %d\n",ii);
    printf("the character is %c\n",cc);
}
```

read markers in a NaN

The IEEE 754 Format

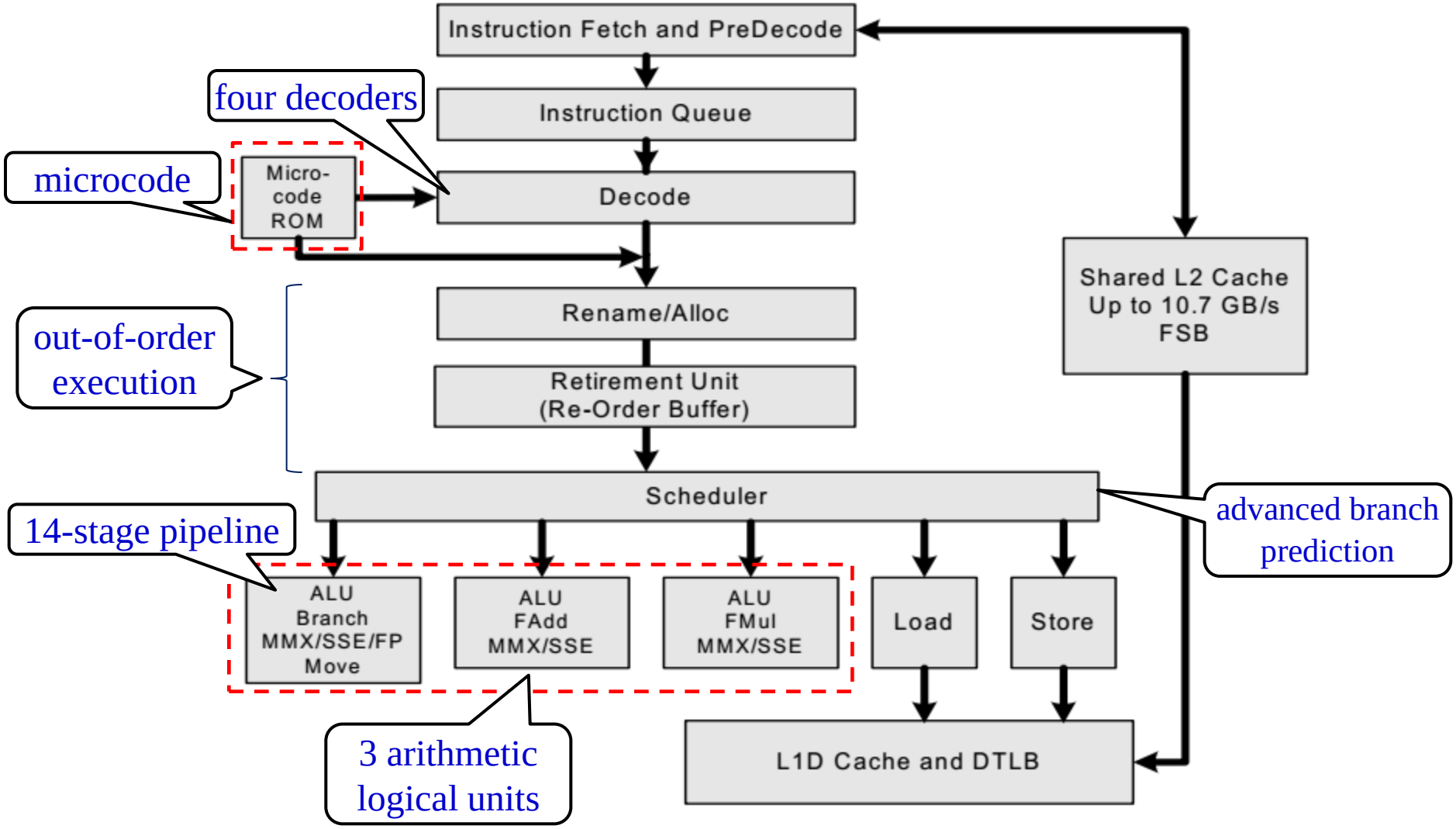
- IEEE 754 uses the idea of scientific notation:
  - sign bit: 0, the number is positive; 1, the number is negative;
  - biased exponent: a bias is added to the actual exponent for high speed comparisons. 2^E-1 and 0 have special meanings.
  - normalized fraction/mantissa: The mantissa for normalized number has a hidden bit which is not stored.

E.g., $(-1)^{\text{sign bit}}(1 + \text{fraction}) \times 2^{\text{exponent - bias}}$



value = $(-1)^0 \times (1+0.25) \times 2^{124-127} = 0.15625$

Intel Core Microarchitecture



Intel Core microarchitecture