# [CSE4152] 고급소프트웨어 실습 I
# Week 12

서강대학교 공과대학 컴퓨터공학과
교수 임 인 성

# 코드 최적화 (Code Optimization)

- **Program optimization (code optimization)**
  - ➢ https://en.wikipedia.org/wiki/Program_optimization#Source_code_level
  - **Goal**
    - Optimize a computer program so that it **executes more rapidly**, or to make it capable of **operating with less memory storage** or **other resources**, or **draw less power**.

- **Levels of otimization**
  - Design level
  - Algorithms and data structures
  - **Source code level**
  - Build level
  - Compile level
  - **Assembly level**
  - Runtime level

# Examples

- **Strength reduction** (https://en.wikipedia.org/wiki/Strength_reduction)
  - Expensive operations are replaced with equivalent but less expensive operations.

```
c = 7;
for (i = 0; i < N; i++)
{
    y[i] = c * i;
}
```

```
c = 7;
k = 0;
for (i = 0; i < N; i++)
{
    y[i] = k;
    k = k + c;
}
```

```
for (i = 0; i < 1048576; i++) {
    A[i] = B[i] / C;
}
```

```
double tmp = 1.0 / C;
for (i = 0; i < 1048576; i++) {
    A[i] = B[i] * tmp;
}
```

| Original calculation | Replacement calculation |
|---|---|
| y = x / 8 | y = x >> 3 |
| y = x * 64 | y = x << 6 |
| y = x * 2 | y = x << 1 |
| y = x * 15 | y = (x << 4) - x |
| y = x / 10 (where x is of type uint16_t) | y = ((uint32_t)x * (uint32_t)0xCCCD) >> 19) |
| y = x / π (where x is of type uint16_t) | y = (((uint32_t)x * (uint32_t)0x45F3) >> 16) + x) >> 2) |

# Cost Comparison of Arithmetic Operations

| | Casio E115 | Casio E115G 168MHz overclocked | Casio E200 | CasioEG800 | Compaq iPAQ 3650 | TouchPC |
|---|---|---|---|---|---|---|
| OS | Windows CE Version 3.0 (build 9348) | Windows CE Version 3.0 (build 9348) | Windows CE Version 3.0 (build 11171) | Windows CE Version 3.0 (build 9348) | Windows CE Version 3.0 (build 9348) | Windows CE Version 3.0 (build 126) |
| Platform | Pocket PC J580 | Pocket PC J582 | Pocket PC J710 | Pocket PC J670 | Compaq iPAQ H3600 | uCErfSA1110 SA-1110 |
| Processor | MIPS R4000 PRId 0x460 | MIPS R4000 PRId 0x460 | StrongArm | MIPS R4000 PRId 0x470 | StrongArm | StrongArm |
| Desktop bits/pixel | 16 | 16 | 16 | 16 | 16 | 8 |
| Desktop resolution | | | 240x320 | | | |
| Performance Counter | 32,768 Hz | 32,768 Hz | 3,686,400 Hz | 32,768 Hz | 1,000,000 Hz | 1,000 Hz |
| Timer speed | 40 Hz | 23 Hz | 468 Hz | 422 Hz | 499 Hz | 40 Hz |
| Integer Add | 18,083,885 Hz | 41,956,466 Hz | 83,364,993 Hz | 37,130,878 Hz | 85,833,226 Hz | Inconsistent results |
| Integer Multiply | 18,649,971 Hz | 20,537,762 Hz | 61,543,084 Hz | 20,390,790 Hz | 61,563,086 Hz | 20,000,000 Hz |
| Integer Divide | 3,494,694 Hz | 4,413,793 Hz | 6,544,954 Hz | 3,873,057 Hz | 7,634,928 Hz | 7,272,727 Hz |
| Float Add | 793,894 Hz | 1,004,537 Hz | 818,766 Hz | 911,361 Hz | 816,876 Hz | 1,000,000 Hz |
| Float Multiply | 586,451 Hz | 1,001,313 Hz | 695,575 Hz | 711,806 Hz | 693,866 Hz | 571,428 Hz |
| Float Divide | 291,984 Hz | 345,581 Hz | 438,665 Hz | 336,582 Hz | 437,620 Hz | 380,952 Hz |

**위 테이블의 내용은 오래 전의 자료임. 하지만 아직도 프로세서에 따라 정도는 다르나 float/double 타입의 연산 특히 곱하기와 나누기 연산은 상대적인 부담이 큼.**

*** NVIDIA GeForce RTX 2080 GPU: Float** 10,068 GFLOPS/ **Double** 314 GFLOPS

# Execution Times on Arduino Mega2560 (16 MHz)

From https://forum.arduino.cc/index.php?topic=196522.0

Unit: nanosecond

| | uint8 | int16 | int32 | int64 | float |
|---|---|---|---|---|---|
| + | 63 | 884 | 1,763 | 8,428 | 10,943 |
| * | 125 | 1,449 | 4,592 | 57,038 | 10,422 |
| / | 15,859<br>(13,714) | 15,969 | 41,866<br>(39,413) | 274,809 | 31,951 |
| sqrt | | 54,251 | 54,448 | 70,884 | 47,127 |

참고: 16 MHz = 62.5 nanoseconds per cycle

- **Loop-invariant code motion (https://en.wikipedia.org/wiki/Loop-invariant_code_motion)**
  - Move statements or expressions which can be moved outside the body of a loop without affecting the semantics of the program.
  - ➢ Mostly done by a complier automatically.

```
int i = 0;
while (i < n) {
    x = y + z;
    a[i] = 6 * i + x * x;
    ++i;
}
```

```
int i = 0;
if (i < n) {
    x = y + z;
    int const t1 = x * x;
    do {
        a[i] = 6 * i + t1;
        ++i;
    } while (i < n);
}
```

# Code Optimization Technique: Loop Unrolling

```
for (int i = 0; i < N; i++) {
    A[i] = B[i] + 2.0 * C[i];
}
```

```
for (int i = 0; i < N; i += 4) {
    A[i] = B[i] + 2.0 * C[i];
    A[i+1] = B[i+1] + 2.0 * C[i+1];
    A[i+2] = B[i+2] + 2.0 * C[i+2];
    A[i+3] = B[i+3] + 2.0 * C[i+3];
}
```

- Increase a program's speed by
  1. reducing or eliminating instructions that control the loop, such as pointer arithmetic and "end of loop" tests on each iteration,
  2. reducing branch penalties, and
  3. hiding latencies, including the delay in reading data from memory.

➢ Undertaken manually by the programmer or by an optimizing compiler.

➢ Optimize the execution speed at the expense of its binary size.

- Often counterproductive, as the increased code size can cause more cache misses.

## Intel® 64 and IA-32 Architectures Optimization Reference Manual

### 3.4.1.7 Loop Unrolling

Benefits of unrolling loops are:

- Unrolling amortizes the branch overhead, since it eliminates branches and some of the code to manage induction variables.
- Unrolling allows one to aggressively schedule (or pipeline) the loop to hide latencies. This is useful if

***Assembly/Compiler Coding Rule 15. (H impact, M generality) Unroll small loops until the overhead of the branch and induction variable accounts (generally) for less than 10% of the execution time of the loop.***

***Assembly/Compiler Coding Rule 16. (H impact, M generality) Avoid unrolling loops excessively; this may thrash the trace cache or instruction cache.***

**Example 3-10. Loop Unrolling**

Before unrolling:

```
do i = 1, 100
      if ( i mod 2 == 0 ) then a( i ) = x
            else a( i ) = y
enddo
```

After unrolling

```
do i = 1, 100, 2
      a( i ) = y
      a( i+1 ) = x
enddo
```

```
for (int i = 0; i < N; i++) {
    A[i] = B[i] + 2.0 * C[i];
}
```

```
for (int i = 0; i < N; i += 4) {
    A[i] = B[i] + 2.0 * C[i];
    A[i+1] = B[i+1] + 2.0 * C[i+1];
    A[i+2] = B[i+2] + 2.0 * C[i+2];
    A[i+3] = B[i+3] + 2.0 * C[i+3];
}
```

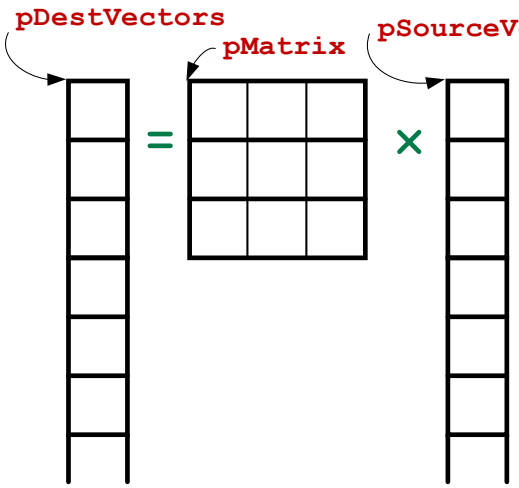
# Loop Unrolling 기법을 통한 성능 향상

강의자료 "1. Loop Unrolling 기법 적용을 통한 성능 향상" 절을 읽을 것.

```
#define NVECTORS 8388608
double *pSVec, *pDVec, pMat[3][3];

// Listing 1
void TransformVectors_L1(double *pDestVectors, const double (*pMatrix)[3],
        const double *pSourceVectors, int NumberOfVectors) {

  int Counter, i, j;
  for (Counter = 0; Counter < NumberOfVectors; Counter++) {
    for (i = 0; i < 3; i++) {
      double Value = 0;
      for(j = 0; j < 3; j++) {
        Value += pMatrix[i][j] * pSourceVectors[j];
      }
      *pDestVectors++ = Value;
    }
    pSourceVectors += 3;
  }
}
```




```
void init_MatVec(void) {
  pSVec = (double *) malloc(sizeof(double)*NVECTORS*3);
  pDVec = (double *) malloc(sizeof(double)*NVECTORS*3);
  // initialize the input vectors pointed by pSVec here.
      :
  // initialize the matrix pointed by pMat here.
      :
}
void main(void) {
  __int64 start, freq, end;
  float resultTime = 0;

  init_MatVec();

  CHECK_TIME_START;
  TransformVectors_L1(pDVec, pMat, pSVec, NVECTORS);
  CHECK_TIME_END(resultTime);
  printf("TransformVectors_L1: %f(s).\n", resultTime);
}
```

C. Hecker, "PowerPC Compilers: Still Not So Hot", Game Developers, 1996.

```
#define NVECTORS 8388608
double *pSVec, *pDVec, pMat[3][3];

// Listing 1
void TransformVectors_L1(double *pDestVectors, const double (*pMatrix)[3],
        const double *pSourceVectors, int NumberOfVectors) {

  int Counter, i, j;
  for (Counter = 0; Counter < NumberOfVectors; Counter++) {
    for (i = 0; i < 3; i++) {
      double Value = 0;
      for(j = 0; j < 3; j++) {
        Value += pMatrix[i][j] * pSourceVectors[j];
      }
      *pDestVectors++ = Value;
    }
    pSourceVectors += 3;
  }
}
```

```
// Listing 5
void TransformVectors_L5(double *pDestVectors, const double (*pMatrix)[3],
        const double *pSourceVectors, int NumberOfVectors) {
  int Counter;
  double Value0, Value1, Value2;

  for (Counter = 0; Counter < NumberOfVectors; Counter++) {
    Value0 = pMatrix[0][0] * pSourceVectors[0];
    Value0 += pMatrix[0][1] * pSourceVectors[1];
    Value0 += pMatrix[0][2] * pSourceVectors[2];
    *pDestVectors++ = Value0;
    Value1 = pMatrix[1][0] * pSourceVectors[0];
    Value1 += pMatrix[1][1] * pSourceVectors[1];
    Value1 += pMatrix[1][2] * pSourceVectors[2];
    *pDestVectors++ = Value1;
    Value2 = pMatrix[2][0] * pSourceVectors[0];
    Value2 += pMatrix[2][1] * pSourceVectors[1];
    Value2 += pMatrix[2][2] * pSourceVectors[2];
    *pDestVectors++ = Value2;
    pSourceVectors += 3;
  }
}
```

**3.4 GHz Intel Core i7 CPU**

0.223184초 → 0.116088초

# 컴파일러와 코드 최적화 정도에 따른 수행 속도 비교

**Table 1. Transform Cycle Counts**

| Compiler | Listing 1 | Listing 2 | KAPed 1 (not shown) | Listing 4 | Listing 5 | Listing 6 |
|---|---|---|---|---|---|---|
| CodeWarrior | 40.7 | 50.5 | 50.9 | 34.3 | 29.7 | 19.6 |
| Symantec C++ | 76.6 | 94.9 | 82.8 | 50.9 | 31.9 | 25.7 |
| Motorola C++ | 34.5 | 47.4 | 39.5 | 33.2 | 30.8 | 20.6 |
| Apple's MrCpp | 52.0 | 65.0 | 56.2 | 36.1 | 28.8 | 19.5 |
| Microsoft VC++ | 41.6 | 49.3 | 42.8 | 31.9 | 21.9 | 22.7 |

**C. Hecker, "PowerPC Compilers: Still Not So Hot", Game Developers, 1996.**

```
// Listing 5
void TransformVectors_L5(double *pDestVectors, const double (*pMatrix)[3],
        const double *pSourceVectors, int NumberOfVectors) {
  int Counter;
  double Value0, Value1, Value2;

  for (Counter = 0; Counter < NumberOfVectors; Counter++) {
    Value0 = pMatrix[0][0] * pSourceVectors[0];
    Value0 += pMatrix[0][1] * pSourceVectors[1];
    Value0 += pMatrix[0][2] * pSourceVectors[2];
    *pDestVectors++ = Value0;
    Value1 = pMatrix[1][0] * pSourceVectors[0];
    Value1 += pMatrix[1][1] * pSourceV
    Value1 += pMatrix[1][2] * pSourceV
    *pDestVectors++ = Value1;
    Value2 = pMatrix[2][0] * pSourceVe
    Value2 += pMatrix[2][1] * pSourceV
    Value2 += pMatrix[2][2] * pSourceV
    *pDestVectors++ = Value2;
    pSourceVectors += 3;
  }
}
```

```
// Listing 6
void TransformVectors_L6(float *pDestVectors, const float (*pMatrix)[3],
        const float *pSourceVectors, int NumberOfVectors) {
  int Counter;
  float Value, _Krr1, _Krr2;

  for (Counter = 0; Counter<NumberOfVectors; Counter++) {
    _Krr1 = pMatrix[0][0] * pSourceVectors[0];
    _Krr2 = pMatrix[1][0] * pSourceVectors[0];
    Value = pMatrix[2][0] * pSourceVectors[0];
    _Krr1 += pMatrix[0][1] * pSourceVectors[1];
    _Krr2 += pMatrix[1][1] * pSourceVectors[1];
    Value += pMatrix[2][1] * pSourceVectors[1];
    _Krr1 += pMatrix[0][2] * pSourceVectors[2];
    _Krr2 += pMatrix[1][2] * pSourceVectors[2];
    Value += pMatrix[2][2] * pSourceVectors[2];
    *pDestVectors++ = _Krr1;
    *pDestVectors++ = _Krr2;
    *pDestVectors++ = Value;
    pSourceVectors += 3;
  }
}
```

| Listing 5 | Listing 6 |
|---|---|
| 29.7 | 19.6 |
| 31.9 | 25.7 |
| 30.8 | 20.6 |
| 28.8 | 19.5 |
| 21.9 | 22.7 |

*Why did it make such a big difference? I have no idea, and the only explanation I can come up with is that you need to hold your compiler's hand on any piece of code you care about. The changes I made for Listings 5 and 6 are very obvious (to a human, if not a compiler). I'm basically just stating explicitly where variables are accessed, where possible aliasing can occur, and which variables are constant throughout a loop iteration. These are all things the compiler is supposed to do for us, so we can work on more important stuff, like design and algorithms, or assembly language code for our most inner loops. We're supposed to trust the compiler will do a respectable job, without having to optimize every line of our code (an impossible task for all but the smallest programs).*

**C. Hecker, "PowerPC Compilers: Still Not So Hot", Game Developers, 1996.**

# 다차원 배열 접근 시 Stride의 크기가 주는 영향

```
#define MATDIM2 8192
double MatA[MATDIM2][MATDIM2], MatB[MATDIM2][MATDIM2], MatC[MATDIM2][MATDIM2];

void MinStride_1(double c[][MATDIM2], double a[][MATDIM2],
                 double b[][MATDIM2]) {
  for (int i = 0; i < MATDIM2; i++)
    for (int j = 0; j < MATDIM2; j++)
      c[i][j] = c[i][j] + a[i][j] * b[i][j];
}

void MinStride_2(double c[][MATDIM2], double a[][MATDIM2],
                 double b[][MATDIM2]) {
  for (int j = 0; j < MATDIM2; j++)
    for (int i = 0; i < MATDIM2; i++)
      c[i][j] = c[i][j] + a[i][j] * b[i][j];
}
```

**3.4 GHz Intel Core i7 CPU**

0.265968초 → 4.862961초

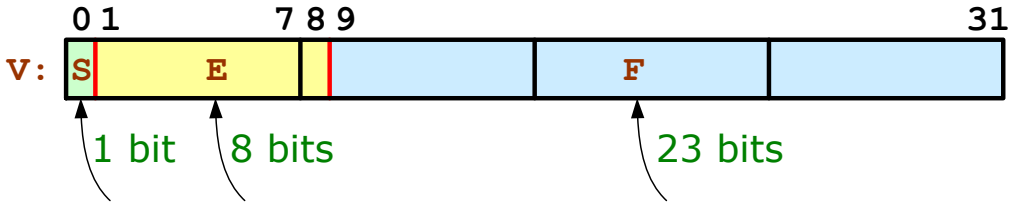
Spatial & temporal locality in memory reference

**강의자료 "2. 다차원 배열 접근 시 Stride의 크기가 주는 영향" 절을 읽을 것.**

http://arstechnica.com/gadgets/2002/07/caching/2

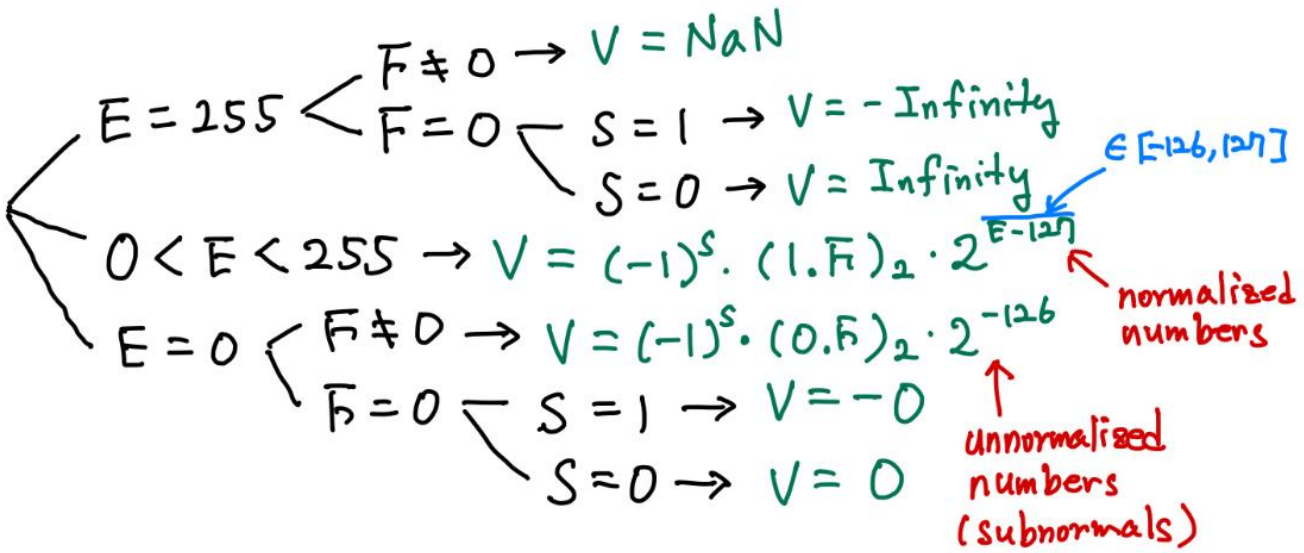
| Level | Access Time | Typical Size | Technology | Managed By |
|---|---|---|---|---|
| **Registers** | 1-3 ns | ?1 KB | Custom CMOS | Compiler |
| **Level 1 Cache** (on-chip) | 2-8 ns | 8 KB-128 KB | SRAM | Hardware |
| **Level 2 Cache** (off-chip) | 5-12 ns | 0.5 MB - 8 MB | SRAM | Hardware |
| **Main Memory** | 10-60 ns | 64 MB - 1 GB | DRAM | Operating System |
| **Hard Disk** | 3,000,000 - 10,000,000 ns | 20 - 100 GB | Magnetic | Operating System/User |

# Single-Precision Floating-Point Number




$$F = f_1 f_2 f_3 \cdots f_{23},\ f_i = 0 \text{ or } 1$$

$$F = 1 \cdot 2^0 + f_1 \cdot 2^{-1} + f_2 \cdot 2^{-2} + f_3 \cdot 2^{-3} + \cdots f_{23} \cdot 2^{-23}$$




$(1.1)_{10}$ = 1 x $(1.000\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100 \ldots)_2$ x $2^0$

(e = 0)

0011 1111 1000 1100 1100 1100 1100 1101

3 f 8 c c c c d

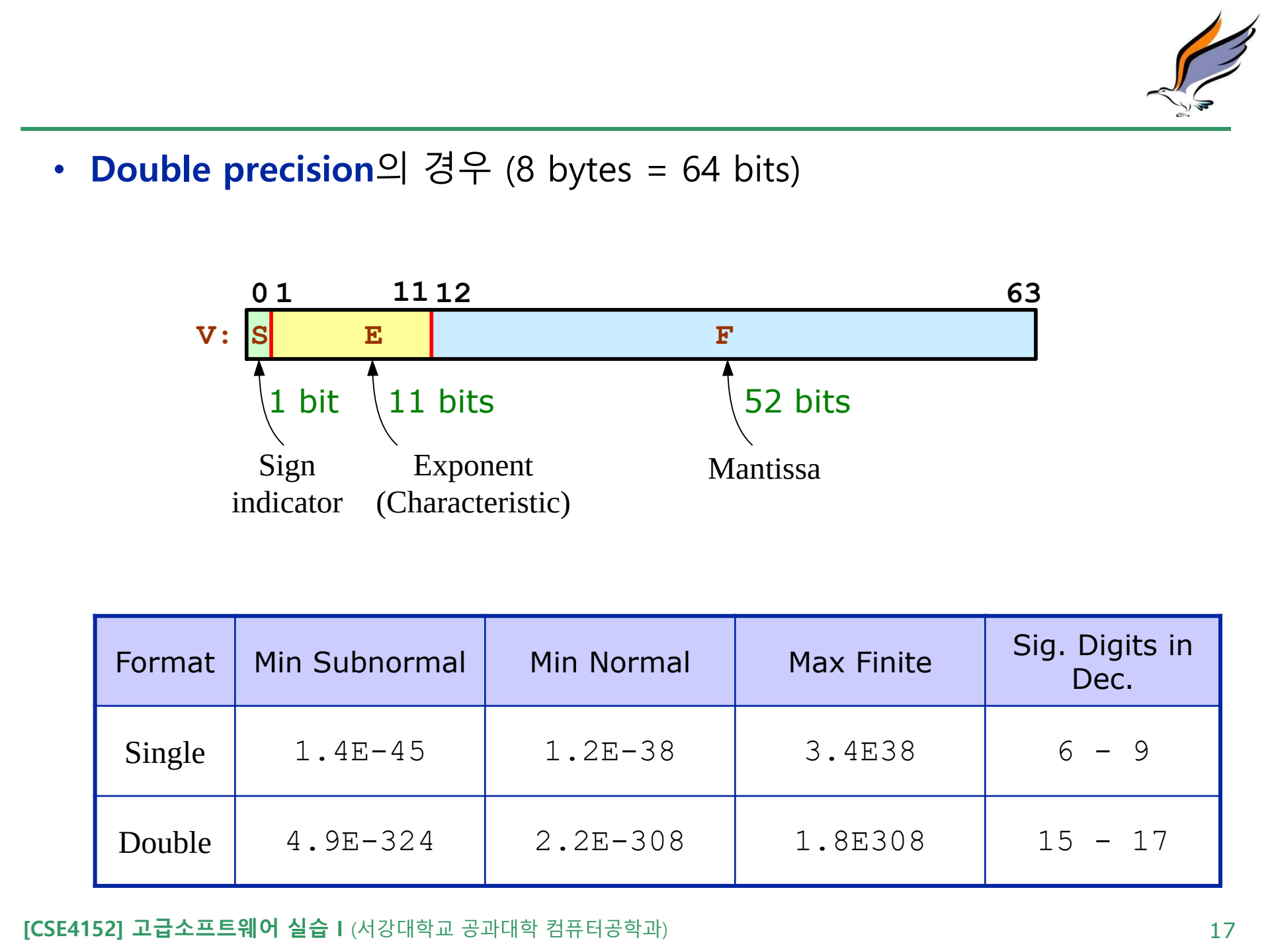
- 특징
  - Can represent only a *finite number* of floating points numbers.
    - At most $2^{32}$ for single precision
    - Is $(1.1)_{10}$ a machine number when this format is used?
  - Can represent only a *limited range* of floating points numbers.

$$\mathrm{MIN}_{single} \le |V| \le \mathrm{MAX}_{single}$$

$$\mathrm{MIN}_{single} = \begin{cases} (1.00\cdots0)_2 \cdot 2^{-126} = 2^{-126} \approx 1.8 \cdot 10^{-38} \ (N) \\ (0.00\cdots1)_2 \cdot 2^{-126} = 2^{-23} \cdot 2^{-126} = 2^{-149} \approx 1.4 \cdot 10^{-45} \ (SUBN) \end{cases}$$

$$\mathrm{MAX}_{single} = (1.11\cdots1)_2 \cdot 2^{127} = \{(10.00\cdots0)_2 - 2^{-23}\} \cdot 2^{127} = (1-2^{-24}) \cdot 2^{128} \approx 3.4 \cdot 10^{38}$$

  - The interval between machine numbers increases as the numbers grow in magnitude.



- **Double precision**의 경우 (8 bytes = 64 bits)




| Format | Min Subnormal | Min Normal | Max Finite | Sig. Digits in Dec. |
|---|---|---|---|---|
| Single | 1.4E-45 | 1.2E-38 | 3.4E38 | 6 - 9 |
| Double | 4.9E-324 | 2.2E-308 | 1.8E308 | 15 - 17 |

# Floating-Point Number를 사용하여 문제를 풀 때

- 다음과 같은 문제 등을 고려해야 함.
  - 컴퓨터가 실수를 정확하게 저장을 못해서 생기는 문제
  - 컴퓨터가 실수 간의 연산을 정확하게 수행을 못해서 생기는 문제
  - 컴퓨터 자체 문제 외에 문제를 푸는 해법, 즉 알고리즘 자체가 본질적으로 unstable (ill-conditioned)해서 생기는 문제
  - 기타

- Floating-Point Arithmetic $x \bullet y \quad (x, y \in \mathrm{R})$
  - On computer,
    ① Store $x$ and $y$ into $fl(x)$ and $fl(y)$, respectively.
    ② Compute $fl(x) \bullet fl(y)$ as correctly as possible.
    ③ Store the result into $fl(\ fl(x) \bullet fl(y)\ )$.

**Normalization and roundoff error**

- A bad example (Base = 10, Num. of sig. dig. = 7)

  `a = 0.1234567E0, b = 0.4711325E4, c = -b`

  Compute `a + b + c`.

**`a + (b + c)`**

```
   ADD r1, b, c
ADD r2, a, r1
```

→ **결과: 0.1234567E0**

**`(a + b) + c`**

```
ADD r1, a, b
ADD r2, r1, c
```

→ **결과: 0.123000E0**

- ✓ Computer에서는 결합 법칙조차 성립 안 함!
- ✓ 부동 소수점 연산을 하면 할 수록 정확도가 점점 나빠질 확률이 높음.

```
float abc, def, ghi;

abc = 1.1e-38;
printf("abc = %20.8e\n", abc);

def = 1.1e-43;
printf("def = %20.8e\n", def);

ghi = 1.1e-46;
printf("ghi = %20.8e\n", ghi);
```

```
abc =       1.09999996e-038
def =       1.09301280e-043
ghi =       0.00000000e+000
Press any key to continue
```

```
float abc, def, ghi;

scanf("%lf %lf", &abc, &def);
printf("abc = %20.8e,   def = %20.8e\n",
abc, def);

ghi = 1.0 - abc*def;
printf("ghi = %20.8e\n", ghi); // 4e-8

ghi = 1.0 - 1.0002*0.9998;
printf("ghi = %20.8e\n", ghi); // 4e-8
```

**Intel Core i7 CPU M620**

```
1.0002 0.9998
abc =       1.00020003e+000,
def =       9.99800026e-001
ghi =      -1.96032914e-008
ghi =       3.99999998e-008
Press any key to continue
```

**A cheap SHARP calculator**

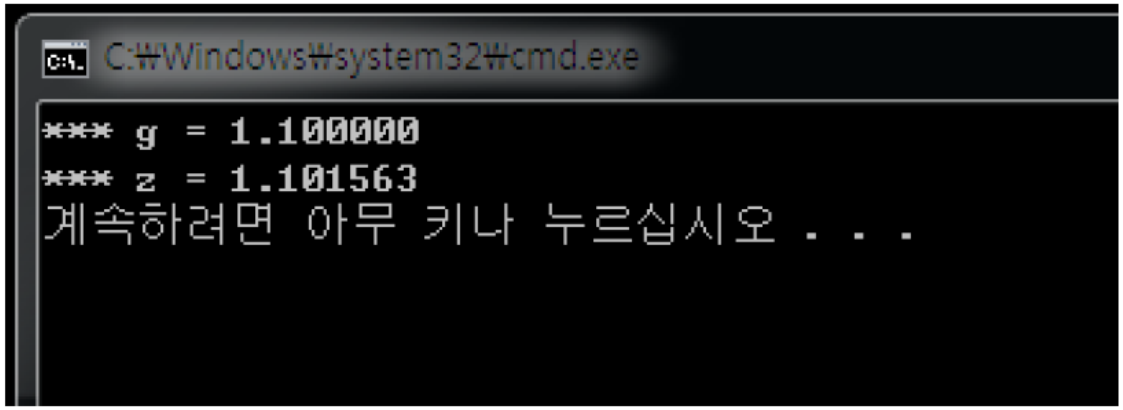
```
0.00000004
```

# 비슷한 숫자 간의 뺄셈

```
void main () {
  float g, x, y, z;


  g = 1.1;
  x = 123456.7890;
  y = x + g;
  z = y - x;
  printf("*** g = %f\n*** z = %f\n", g, z);
}
```

왜 이러한 현상이 발생하였을까?



$$1.1 \approx (-1)^0 \times (1.000\,1100\,1100\,1100\,1100\,1101)_2 \times 2^0$$

$$123456.7890 \approx (-1)^0 \times (1.111\,0001\,0010\,0000\,0110\,0101)_2 \times 2^{16}$$

큰 수와 작은 수와의 덧셈

```
                  (e = 0)
g = 1.1 =         0011 1111 1000 1100 1100 1100 1100 1101
                   3    f    8    c    c    c    c    d

                  (e = 16)
x = 123456.7890 = 0100 0111 1111 0001 0010 0000 0110 0101
                   4    7    f    1    2    0    6    5

x + g =           111 0001 0010 0000 0110 0101
                                1000 1100 1100 1100 1100 1101
                  111 0001 0010 0000 1111 0001 1

y =               0100 0111 1111 0001 0010 0000 1111 0010
x =               0100 0111 1111 0001 0010 0000 0110 0101
y - x =           0100 0111 1000 0000 0000 0000 1000 1101

z =               0011 1111 1000 1101 0000 0000 0000 0000
                   3    f    8    d    0    0    0    0
```

비슷한 수끼리의 뺄셈

# 부동소수점 연산 시 주의할 점

- 비슷한 숫자 간의 뺄셈 (Loss of significance)
  - $-b + \sqrt{b^2 - 4ac} \longrightarrow \frac{-4ac}{b+\sqrt{b^2-4ac}}$ when $b > 0$ and $b^2 \gg |ac|$
  - $\sqrt{x+\delta} - \sqrt{x} \longrightarrow \frac{\delta}{\sqrt{x+\delta}+\sqrt{x}}$ when $x + \delta, x > 0$ and $|\delta| \ll |x|$
  - $\cos(x+\delta) - \cos x \longrightarrow -2 \sin \frac{\delta}{2} \sin(x + \frac{\delta}{2})$ when $|\delta| \ll |x|$
  - $x - \sin x \longrightarrow x - (x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots)$ when $|x| \approx 0$
- 아주 큰 수와 아주 작은 수와의 덧셈/뺄셈
- 아주 작은 수로의 나눗셈
- 기타