Class Topics (클래스 홈페이지 참조)

- Part 1: Fundamental concepts and principles
- Part 2: 빠른 컴퓨터를 위한 ISA design
  - Topic 1 Computer performance and ISA design (Ch. 1)
  - Topic 2 RISC (MIPS) instruction set (Chapter 2)

    2-1 ALU and data transfer instructions

    2-2 Branch instructions

    2-3 Supporting program execution
  - Topic 3 Computer arithmetic and ALU (Chapter 3)
- Part 3: ISA 의 효율적인 구현 (pipelining, cache memory)

Chapter 3

Arithmetic for Computers (ALU)

Some of authors' slides are modified

Arithmetic for Computers

- We have build a 32-bit ALU in “Part 1” (concept only)
- Can you imagine many algorithms for addition?
  - Multiplication, division, FP operations as well
- Focus of designing computers in 1950s
  - Algorithms and implementation for faster ALU
- Field of “Computer Arithmetic”
  - Matured in 1950s and 1960s; improved since then
- Today, can buy ALU as IP (Intellectual Property)
- We do not study the design of fast ALU in detail

Instead, we focus on

1) Operations on integers (concept only, skip performance)
   - Addition and subtraction
     - Dealing with overflow
   - Multiplication and division
2) Floating-point real numbers
   - Representation and operations
3) Arithmetic for multimedia

❑ Impact of these operations on ISA

- This is why Chapter 3 belongs to “Part 2”

1) Integer Addition & Subtraction

Overflow (표현범위 벗어남),

Issue of Finite Precision

Representation of Integers

- Unsigned integers

000 = +0

001 = +1

010 = +2

011 = +3

100 = +4

101 = +5

110 = +6

111 = +7

Representing Signed Integers

Sign Magnitude	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- Issues: balance, number of zeros, ease of operations
- Which one is best? Why?

Representing Signed Integers

Biased Notation:	Biased Notation:
000 = -3	000 = -4
001 = -2	001 = -3
010 = -1	010 = -2
011 = 0	011 = -1
100 = +1	100 = 0
101 = +2	101 = +1
110 = +3	110 = +2
111 = +4	111 = +3
(Bias 3)	(Bias 4)

Integer Addition

- Addition (unsigned and signed)
  - Example: 9 + 12

$$
\begin{array}{rcl}
 & & 0\,0\,0\,1\;\;0\,0\,0 \qquad \text{carry} \\
9_{10} & = & 0\,0\,0\,0\;\;1\,0\,0\,1 \\
12_{10} & = & 0\,0\,0\,0\;\;1\,1\,0\,0 \\
\hline
 & & 0\,0\,0\,1\;\;0\,1\,0\,1 \quad = 21_{10}
\end{array}
$$

  - Subtraction for signed integers (add 2's complement)
- Subtraction for unsigned integers

C/B in Unsigned Arithmetic

au = bu + cu // au, bu, cu are unsigned number

- Carry and borrow (표현범위 벗어남)

	1 ←	
192	1100	0000
65	0100	0001
1(?)	0000	0001

	1 →	
128	1000	0000
-192	1100	0000
192(?)	1100	0000

- Programmers are entirely responsible (processor 개입 불가)
  - If problem, add correctional code or use long data type
    - If (carry bit), then … // error handling
  - Otherwise, ignore (e.g., address arithmetic by compiler)

Overflow in Signed Arithmetic

a = b + c // a, b, c are signed numbers

- (2's complement) arithmetic overflow (표현범위 벗어남)
  - Two patterns: can detect using a single XOR gate

	carry: 0 1			carry: 1 0		
70	0100	0110		-70	1011	1010
+ 80	0101	0000		-80	1011	0000
150(?)	1001	0110		-150(?)	0110	1010

- Overflow occur when adding two positives yields a negative
  - Or, adding two negatives gives a positive

Overflow in Signed Arithmetic

- If problem, add correctional code or use long data type
  - If (overflow), then … // error handling
- Otherwise, ignore
  - But 2's C. arithmetic overflow almost always a problem!
    - Processor 개입 여지 (should it detect overflow?)

- If we ignore 2's complement arithmetic overflow
  - Programmers responsible for everything
    - Consistent with the case of unsigned arithmetic

Overflow in Signed Arithmetic

- C language: leave all to programmers
  - Compilers use "**addu, addiu, subu, …**"
- Fortran (& Ada): catch 2's complement overflow (exception)
  - Use "**add, addi, sub, …**" for signed numbers
  - Use "**addu, addiu, subu, …**" for unsigned numbers
- MIPS solution: two kinds of arithmetic instructions

addu, addiu, subu, … // ignore overflow

// "u" in "addu" does not mean unsigned

add, addi, sub, … // detect overflow (exception)

Overflow in Signed Arithmetic

- Detecting overflow means invoking exception handler
  - Save PC in exception program counter (EPC) register
  - Jump to predefined handler address
- What can an exception handler do?

Overflow in Signed Arithmetic

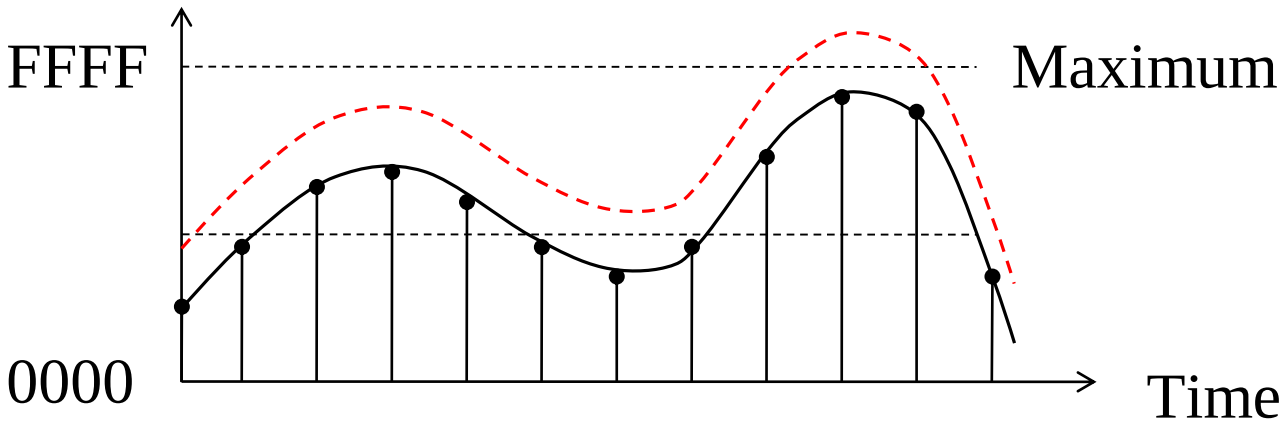
- 32 bit signed numbers:

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +\ 2_{ten}$
...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$ ***maxint***
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$ ***minint***
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2{,}147{,}483{,}646_{ten}$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

- Finite precision; how can we represent big numbers?
  - Double precision, floating point

Arithmetic for Multimedia

- Saturating operations
  - On overflow, result is largest (or smallest) representable value
    - e.g., clipping in audio, saturation in video



Integer Multiplication & Division

(가볍게)

(concept only; not for performance)

Multiplication

multiplicand

multiplier

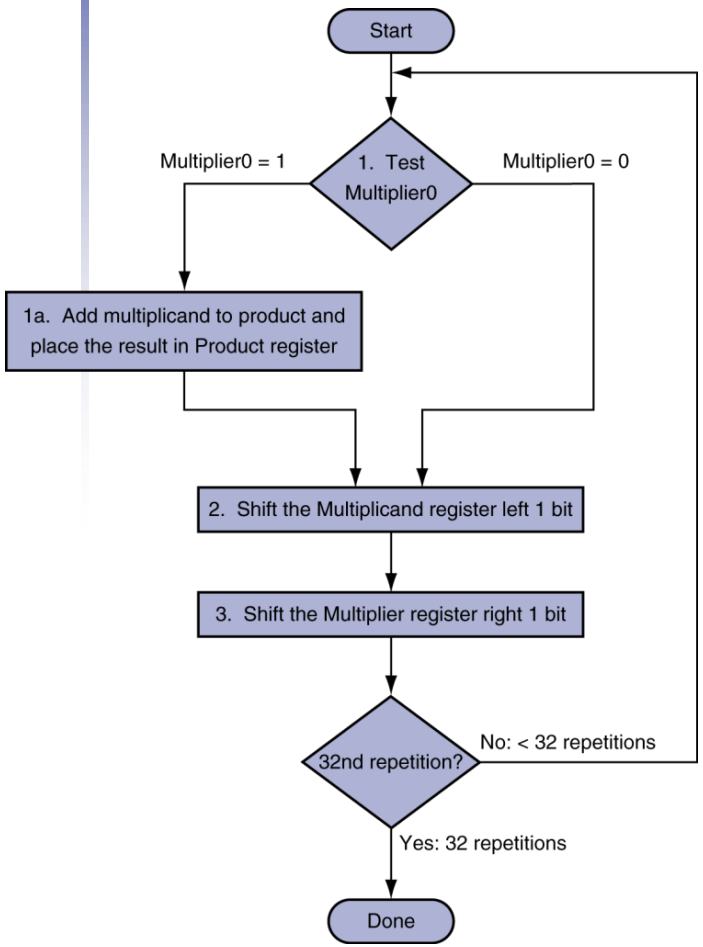
product

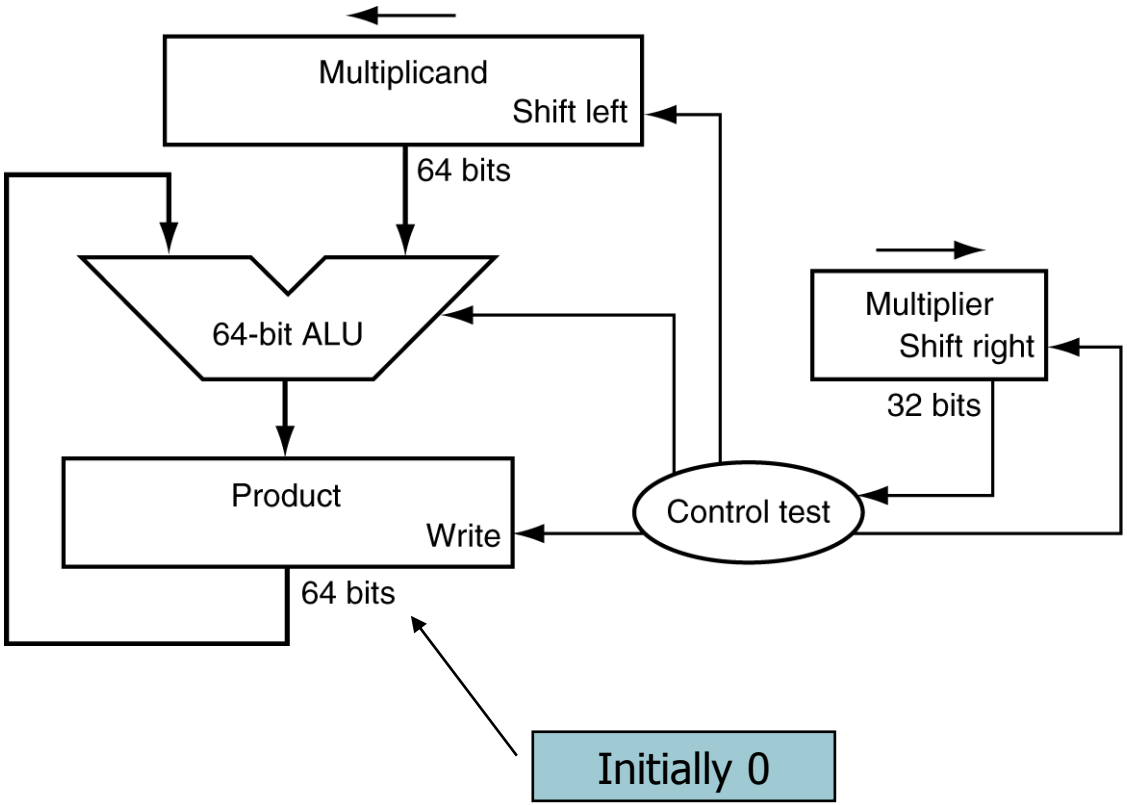
```
      1000
  ×   1001
  --------
      1000
     0000
    0000
   1000
  --------
   1001000
```

Length of product is the sum of operand lengths

- More complicated than addition
  - Accomplished via shifting and addition
- More time and more area
- Negative numbers: convert and multiply
  - There are better techniques

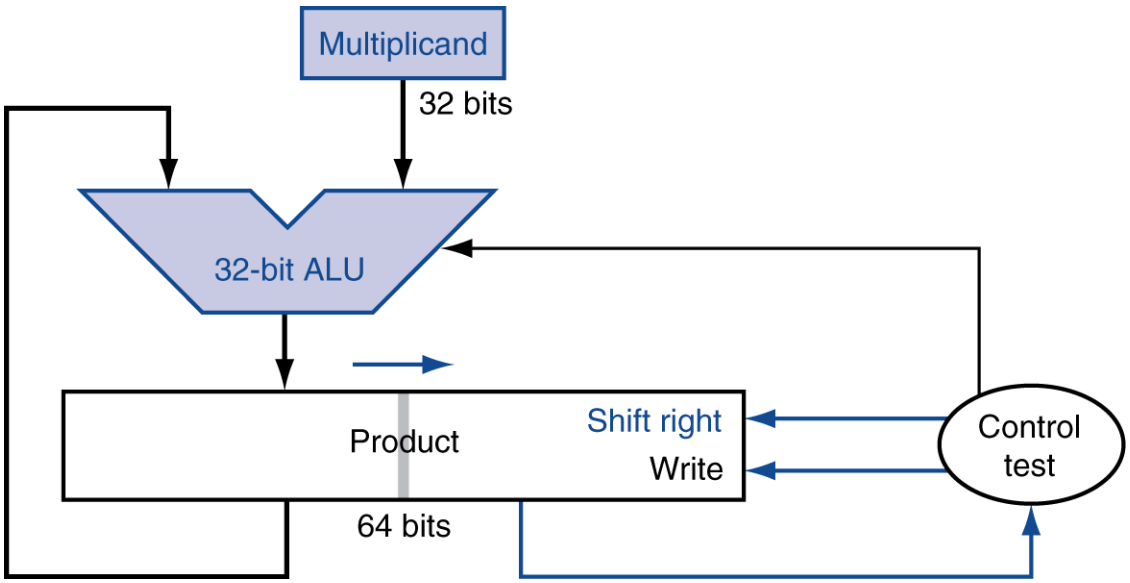
Multiplication Hardware (참고)





Optimized Multiplier (참고)

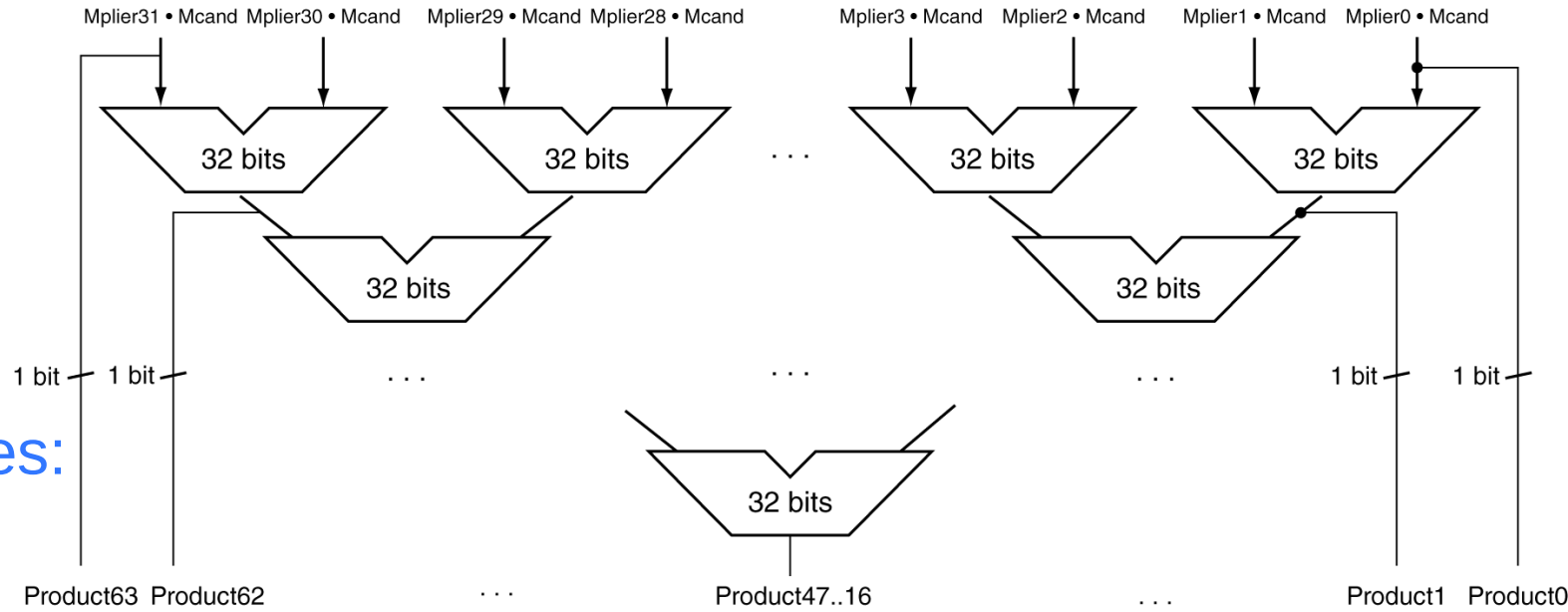
- Perform steps in parallel: add/shift



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

Faster Multiplier (참고)

- Uses multiple adders (Moore's law)
  - Cost/performance tradeoff



No. of stages:

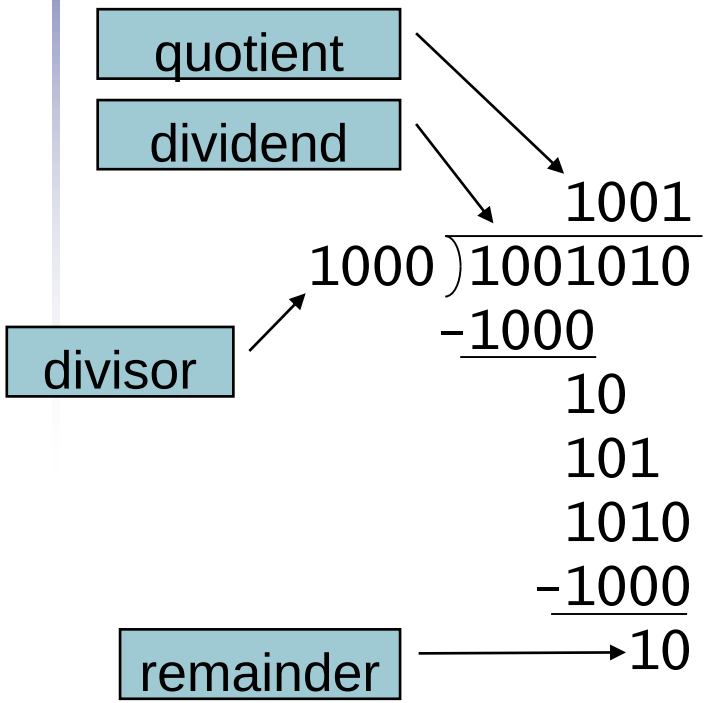
$\text{Log}_2 32 = 5$

- Can be pipelined
  - Several multiplication performed in parallel

MIPS Multiplication

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Instructions
  - **`mult rs, rt / multu rs, rt`**
    - 64-bit product in HI/LO
  - **`mfhi rd / mflo rd`**
    - Move from HI/LO to rd
  - **`mul rd, rs, rt`**
    - Least-significant 32 bits of product –> rd

Division (slower than mult)



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

- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

MIPS Division

- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient
- Instructions
  - `div rs, rt / divu rs, rt`
  - Use `mfhi`, `mflo` to access result
  - No overflow or divide-by-0 checking
    - Software must perform checks if required

2) Floating-Point Real Numbers

Floating Point

- Representation for non-integral numbers
  - Including very small and very big numbers
- Scientific numbers

$$-2.34 \times 10^{56}$$

$$+0.002 \times 10^{-94}$$

$+9.5$ // small numbers, no problem

- 지수표현
  - Significand and exponent

-2.34	56

32-bit

Floating Point

- Floating point Representation not unique
  - Use normalized form

$$-2.34 \times 10^{56}$$ ← normalized

$$-0.234 \times 10^{57}$$ ← not normalized

$$-23.4 \times 10^{55}$$ ← not normalized

- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$

$\pm$ *1.xxxxxxx*	*yyyyy*

32-bit

  - Implicit 1 and fraction

Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit): type **float** in C
  - Double precision (64-bit): type **double** in C

IEEE Floating-Point Format

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
S	exponent								fraction																						

1 bit 8 bits 23 bits

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
S	exponent											fraction																			

1 bit 11 bits 20 bits

fraction (continued)

32 bits

Floating-Point Example

```
float  x = -0.75;
double  x = -0.75;
```

- Represent –0.75
  - $-0.75 = -0.11_2 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias // bias 127 or 1023
    - Single: –1 + 127 = 126 = 01111110_2
    - Double: –1 + 1023 = 1022 = 01111111110_2
- Single: 10111111 01000000 00000000 00000000
- Double: 10111111 11101000 00000000 00000000 …00

Floating-Point Example

- What number is represented by the single-precision float

  11000000 10100000 00000000 00000000

  - S = 1
  - Fraction = $01000\ldots00_2$
  - Exponent = $10000001_2 = 129$

- $x = (-1)^1 \times (1 + .01_2) \times 2^{(129 - 127)}$ // bias 127

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

IEEE Floating-Point Format (부연)

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: 1.0 ≤ |significand| < 2.0
  - Implicit 1; Significand is Fraction + 1
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1023

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating Point Standard

- Members of the committee
  - 과학 계산 전문가
  - Computer arithmetic (ALU 설계/구현) 전문가
- 과학계산 issues (표현 범위, 유효숫자)
  - Total number of bits
  - Number of bits for significand/exponent
    - More bits for significand gives more accuracy
    - More bits for exponent increases range
- The rest are determined by ALU specialists
  - For efficient implementation

Floating Point Standard

- When you do C, Java programming
  - When do you use **float**?
  - When do you use **double**?
- FP numbers and FP ALU (or FP coprocessor)
  - 근사계산
    - Too many bits for fraction: rounded
  - No beauty of integers
    - Not accurate from mathematics perspective
  - Think about overflow, underflow

Floating Point Standard

- How do we represent 0?

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- Exponents 0000…00 and 1111…11 reserved
  - Exp = 255, sig != 0 → NaN (not a number)
  - Exp = 255, sig = 0 → +, - infinity
  - Exp = 0, sig = 0 → 0
  - Exp =0, sig != 0 → even smaller numbers
- IEEE 754-2008
  - Binary 32, 64, 128, (16, 256)

Associativity

- Order of computation in "x + y + z" (problem?)

		(x+y)+z	x+(y+z)
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

Y: 1.50 E+38

Z: 0.00000 00000 00000 00000 00000 00000 00000 001 E+38

Y+Z: 1.50000 00000 00000 00000 00000 00000 00000 001 E+38

// too many bits in fraction: rounded

Y+Z = 1.50 E38

Floating Point Comparison

- Floating-point "beq"? What will happen? Why?

```
float  x, y;
x = 1.5;  y = 2.0;
if ((x * y) == 3.0)  printf("equal\n");

x = 0.1;  y = 10.0;                    // no. of fraction bits
if ((x * y) == 1.0)  printf("equal\n");
```

- Floating point equality check: use " | x – constant | < ϵ "

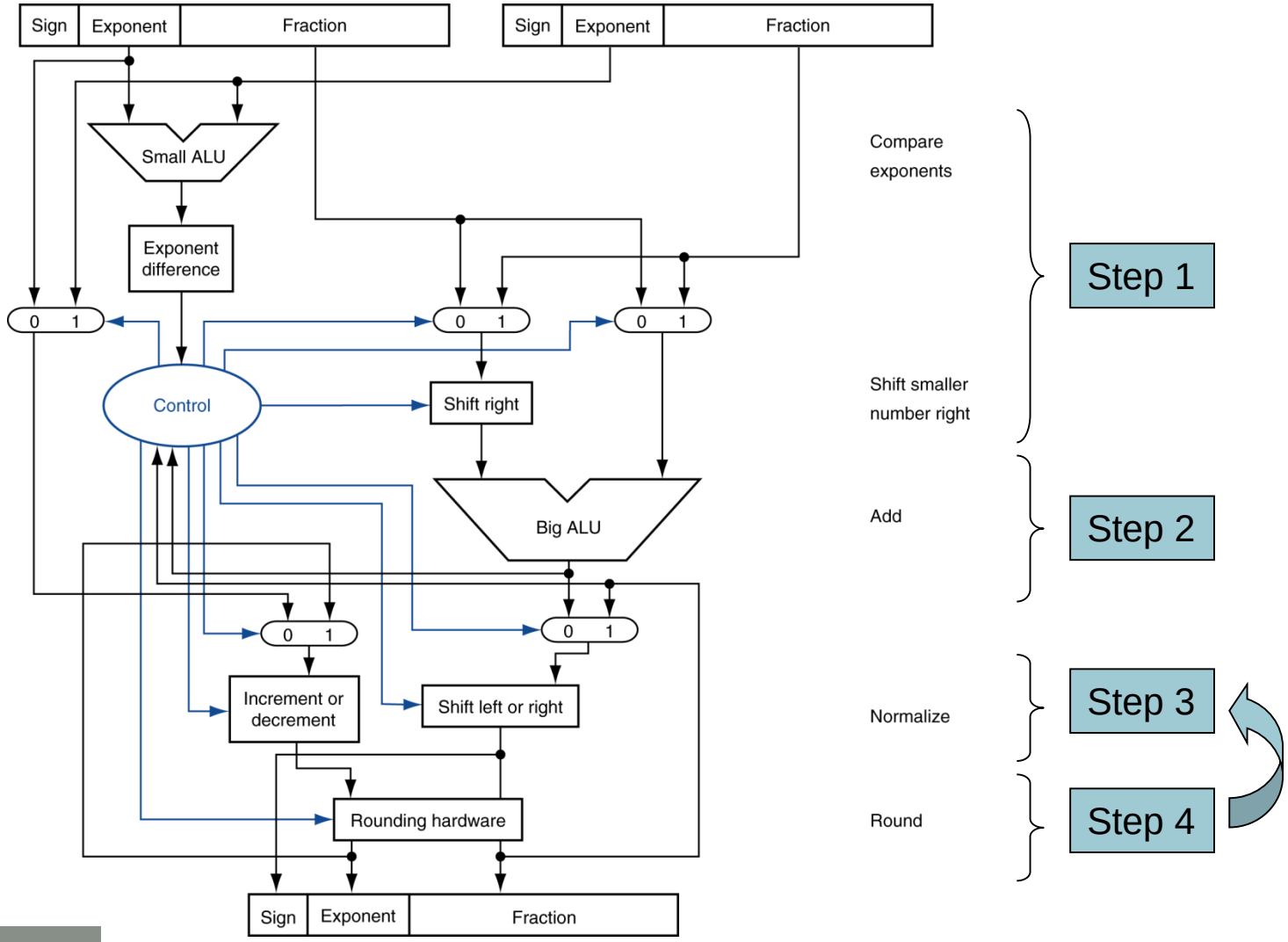
Floating-Point Addition

- Now consider a 4-digit binary example

  $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ // 0.5 + –0.4375

  1. Align binary points
     - Shift number with smaller exponent
     - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
  2. Add significands
     - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-}1 = 0.001_2 \times 2^{-1}$
  3. Normalize result & check for over/underflow
     - $1.000_2 \times 2^{-4}$, with no over/underflow
  4. Round and renormalize if necessary
     - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

FP Adder Hardware (참고)



FP Adder Hardware

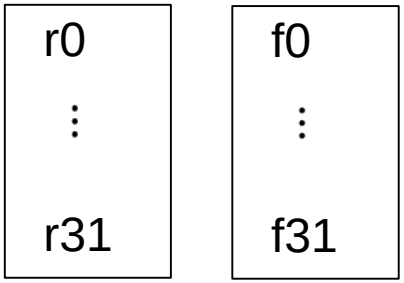
- Much more complex than integer adder
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined

FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP $\leftrightarrow$ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

FP Instructions in MIPS

- FP hardware is adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: $f0, $f1, … $f31
  - Paired for double-precision: $f0/$f1, $f2/$f3, …
- FP instructions operate only on FP registers
  - Generally no integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - `lwc1`, `ldc1`, `swc1`, `sdc1`
    - e.g., `ldc1 $f8, 32($sp)`



FP Instructions in MIPS

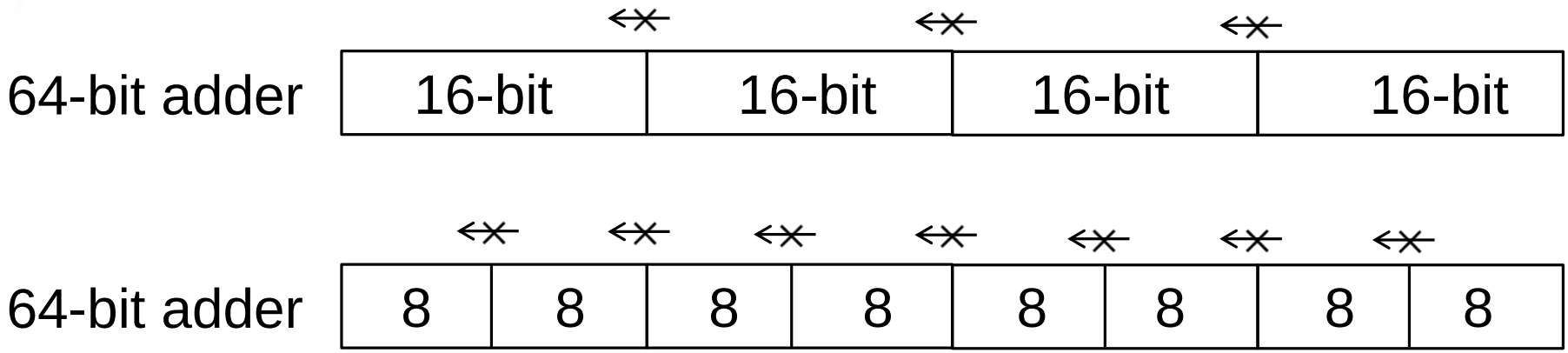
- Single-precision arithmetic: `add.s`, `sub.s`, `mul.s`, div.s
  - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic: `add.d`, `sub.d`, `mul.d`, `div.d`
  - e.g., `mul.d $f4, $f4, $f6`
- Single- and double-precision comparison
  - `c.`*xx*`.s`, `c.`*xx*`.d` (*xx* is `lt`, `le`, …)
  - Sets or clears FP condition-code bit
    - e.g. `c.lt.s $f3, $f4`
- Branch on FP condition code true or false: `bc1t`, `bc1f`
  - e.g., `bc1t TargetLabel`

3) Arithmetic for Multimedia (and Scientific Computing)

(Textbook Sections 3.6 – 3.8)

Arithmetic for Multimedia

- Graphics and media processing operates on vectors of 8-bit and 16-bit data
  - Use 64-bit adder, with partitioned carry chain
    - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors



Subword Parallelism

- Also called data-level parallelism (DLP), vector parallelism, or Single Instruction Multiple Data (SIMD)
  - Instructions operate on them simultaneously
- Moore's law for graphics and audio applications
  - 128-bit adder and wide registers
    - 16×8-bit adds, 8×16-bit adds, 4×32-bit adds

MMX and SSE (Intel 사례)

- MMX (1993) // multimedia
  - Use (64 bits of) existing FP registers
  - Small integer data types (8bit color, 16bit audio)
    - (8×8), (4×16), (2×32), (1×64)
- SSE (1999) // scientific computing (Moore's law)
  - Add new 8×128-bit FP registers (XMM0-XMM7)

32-bit FP	32-bit FP	32-bit FP	32-bit FP

  - Packed single precision FP (4×32) with more FP units

SSE2, …, AVX (Intel 사례)

- SSE2 (2001)
  - Generalize SSE (8 × 128-bit registers)
  - Can be used for multiple FP/INT operands
    - 2 × 64-bit FP (addpd %xmm0, %xmm4)
    - 4 × 32-bit FP (addps %xmm0, %xmm4)
    - 2 × 64-bit, 4 × 32-bit, 8 ×16-bit,16×8-bit INT
- AVX (advanced vector extension; 2008)
  - Double the width of registers
    - e.g., 4 × 64-bit double precision FP

Going Faster

- DGEMM (Double Precision General Matrix Multiply) with Intel Core i7
  - AVX version 3.85 times fast
    - 4 double precision FP operations in parallel
- Matrix computation (과학 계산)
  - Differential equations
  - Deep learning (vectorization with GPU)

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ & \cdots & \end{bmatrix} \begin{bmatrix} b_{11} & \cdots & \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & \end{bmatrix} = \begin{bmatrix} * & \cdots & \\ \vdots & \ddots & \vdots \\ & \cdots & \end{bmatrix}$$

Matrix Multiply, DGEMM (참고)

- Unoptimized code:

```
void dgemm (int n, double* A, double* B, double* C)
{
 for (int i = 0; i < n; ++i)
   for (int j = 0; j < n; ++j)
   {
    double cij = C[i+j*n]; /* cij = C[i][j] */
    for(int k = 0; k < n; k++ )
     cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
    C[i+j*n] = cij; /* C[i][j] = cij */
   }
}
```

Matrix Multiply, DGEMM (참고)

- x86 assembly code:

```
1. vmovsd (%r10),%xmm0   # Load 1 element of C into %xmm0
2. mov %rsi,%rcx         # register %rcx = %rsi
3. xor %eax,%eax         # register %eax = 0
4. vmovsd (%rcx),%xmm1   # Load 1 element of B into %xmm1
5. add %r9,%rcx          # register %rcx = %rcx + %r9
6. vmulsd (%r8,%rax,8),%xmm1,%xmm1 # Multiply %xmm1,
   element of A
7. add $0x1,%rax         # register %rax = %rax + 1
8. cmp %eax,%edi         # compare %eax to %edi
9. vaddsd %xmm1,%xmm0,%xmm0 # Add %xmm1, %xmm0
10. jg 30 <dgemm+0x30>   # jump if %eax > %edi
11. add $0x1,%r11d       # register %r11 = %r11 + 1
12. vmovsd %xmm0,(%r10) # Store %xmm0 into C element
```

Matrix Multiply, DGEMM (참고)

- Optimized C code:

```
#include <x86intrin.h>
void dgemm (int n, double* A, double* B, double* C)
{
 for ( int i = 0; i < n; i+=4 )
  for ( int j = 0; j < n; j++ ) {
   __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
   for( int k = 0; k < n; k++ )
    c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
            _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
            _mm256_broadcast_sd(B+k+j*n)));
   _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
  }
}
```

Matrix Multiply, DGEMM (참고)

- Optimized x86 assembly code:

```
1. vmovapd (%r11),%ymm0        # Load 4 elements of C into %ymm0
2. mov %rbx,%rcx               # register %rcx = %rbx
3. xor %eax,%eax               # register %eax = 0
4. vbroadcastsd (%rax,%r8,1),%ymm1 # Make 4 copies of B element
5. add $0x8,%rax               # register %rax = %rax + 8
6. vmulpd (%rcx),%ymm1,%ymm1 # Parallel mul %ymm1,4 A elements
7. add %r9,%rcx                # register %rcx = %rcx + %r9
8. cmp %r10,%rax               # compare %r10 to %rax
9. vaddpd %ymm1,%ymm0,%ymm0    # Parallel add %ymm1, %ymm0
10. jne 50 <dgemm+0x50>        # jump if not %r10 != %rax
11. add $0x1,%esi              # register % esi = % esi + 1
12. vmovapd %ymm0,(%r11)       # Store %ymm0 into 4 C elements
```

C Intrinsics

- Intrinsic functions
  - Functions available for use in a given programming language whose implementation is specially handled by compiler
  - Often used to explicitly implement vectorization and parallelization in languages which do not address such constructs
    - e.g., MMX, SSE, OpenMP
- Compiler target?
- Cost-performance-energy tradeoff

SIMD Parallelism

- Three variations (Chapter 6) of DLP
  - SIMD extensions
  - Vector architectures
  - Graphics Processing Units (GPUs)
    - 3D video games since 2000, extensive parallelism
    - Also used by scientific computing
- Deep learning and GPU computing

4) Summary of MIPS ISA

(Textbook section 3.10)

Concluding Remarks

- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow
- MIPS ISA
  - Core instructions: 56 most frequently used
    - 100% of SPECINT, 97% of SPECFP
  - Other instructions: less frequent

MIPS Core, MIPS-32, Pseudo MIPS

MIPS core instructions	Name	Format	MIPS arithmetic core	Name	Format
add	add	R	multiply	mult	R
add immediate	addi	I	multiply unsigned	multu	R
add unsigned	addu	R	divide	div	R
add immediate unsigned	addiu	I	divide unsigned	divu	R
subtract	sub	R	move from Hi	mfhi	R
subtract unsigned	subu	R	move from Lo	mflo	R
AND	AND	R	move from system control (EPC)	mfc0	R
AND immediate	ANDi	I	floating-point add single	add.s	R
OR	OR	R	floating-point add double	add.d	R
OR immediate	ORi	I	floating-point subtract single	sub.s	R
NOR	NOR	R	floating-point subtract double	sub.d	R
shift left logical	sll	R	floating-point multiply single	mul.s	R
shift right logical	srl	R	floating-point multiply double	mul.d	R
load upper immediate	lui	I	floating-point divide single	div.s	R
load word	lw	I	floating-point divide double	div.d	R
store word	sw	I	load word to floating-point single	lwc1	I
load halfword unsigned	lhu	I	store word to floating-point single	swc1	I
store halfword	sh	I	load word to floating-point double	ldc1	I
load byte unsigned	lbu	I	store word to floating-point double	sdc1	I
store byte	sb	I	branch on floating-point true	bc1t	I
load linked (*atomic update*)	ll	I	branch on floating-point false	bc1f	I
store cond. (*atomic update*)	sc	I	floating-point compare single	c.x.s	R
branch on equal	beq	I	(x = eq, neq, lt, le, gt, ge)		
branch on not equal	bne	I	floating-point compare double	c.x.d	R
jump	j	J	(x = eq, neq, lt, le, gt, ge)		
jump and link	jal	J			
jump register	jr	R			
set less than	slt	R			
set less than immediate	slti	I			
set less than unsigned	sltu	R			
set less than immediate unsigned	sltiu	I			

Figure 3.26

MIPS Core, MIPS-32, Pseudo MIPS

Remaining MIPS-32	Name	Format
exclusive or (*rs* ⊕ *rt*)	xor	R
exclusive or immediate	xori	I
shift right arithmetic	sra	R
shift left logical variable	sllv	R
shift right logical variable	srlv	R
shift right arithmetic variable	srav	R
move to Hi	mthi	R
move to Lo	mtlo	R
load halfword	lh	I
load byte	lb	I
load word left *(unaligned)*	lwl	I
load word right *(unaligned)*	lwr	I
store word left *(unaligned)*	swl	I
store word right *(unaligned)*	swr	I
load linked (*atomic update*)	ll	I
store cond. (*atomic update*)	sc	I
move if zero	movz	R
move if not zero	movn	R
multiply and add (S or *uns.*)	madd*s*	R
multiply and subtract (S or *uns.*)	msub*s*	I
branch on ≥ zero and link	bgezal	I
branch on < zero and link	bltzal	I
jump and link register	jalr	R
branch compare to zero	bxz	I
branch compare to zero likely	bxzl	I
(x = lt, le, gt, ge)		
branch compare reg likely	bxl	I
trap if compare reg	tx	R
trap if compare immediate	txi	I
(x = eq, neq, lt, le, gt, ge)		
return from exception	rfe	R
system call	syscall	I
break *(cause exception)*	break	I
move from FP to integer	mfc1	R
move to FP from integer	mtc1	R
FP move (*s* or *d*)	mov.*f*	R
FP move if zero (*s* or *d*)	movz.*f*	R
FP move if not zero (*s* or *d*)	movn.*f*	R
FP square root (*s* or *d*)	sqrt.*f*	R
FP absolute value (*s* or *d*)	abs.*f*	R
FP negate (*s* or *d*)	neg.*f*	R
FP convert (*w*, *s*, or *d*)	cvt.*f*.*f*	R
FP compare un (*s* or *d*)	c.xn.*f*	R

Pseudo MIPS	Name	Format
absolute value	abs	rd,rs
negate *(signed or unsigned)*	neg*s*	rd,rs
rotate left	rol	rd,rs,rt
rotate right	ror	rd,rs,rt
multiply and don't check oflw *(signed or uns.)*	mul*s*	rd,rs,rt
multiply and check oflw *(signed or uns.)*	mulo*s*	rd,rs,rt
divide and check overflow	div	rd,rs,rt
divide and don't check overflow	divu	rd,rs,rt
remainder *(signed or unsigned)*	rem*s*	rd,rs,rt
load immediate	li	rd,imm
load address	la	rd,addr
load double	ld	rd,addr
store double	sd	rd,addr
unaligned load word	ulw	rd,addr
unaligned store word	usw	rd,addr
unaligned load halfword *(signed or uns.)*	ulh*s*	rd,addr
unaligned store halfword	ush	rd,addr
branch	b	Label
branch on equal zero	beqz	rs,L
branch on compare *(signed or unsigned)*	bx*s*	rs,rt,L
(x = lt, le, gt, ge)		
set equal	seq	rd,rs,rt
set not equal	sne	rd,rs,rt
set on compare *(signed or unsigned)*	sx*s*	rd,rs,rt
(x = lt, le, gt, ge)		
load to floating point (*s* or *d*)	l.*f*	rd,addr
store from floating point (*s* or *d*)	s.*f*	rd,addr

Figure 3.27

MIPS Core, MIPS-32, Pseudo MIPS

- Frequency of use in SPEC CPU2006 benchmark

Instruction subset	Integer	Fl. pt.
MIPS core	98%	31%
MIPS arithmetic core	2%	66%
Remaining MIPS-32	0%	3%

- For the rest of book, focus on MIPS core (integer instruction set excluding multiply and divide)
  - To make the explanation of computer design easier

Homework #10 (see Class Homepage)

1) Write a report summarizing the materials discussed in Topic 3
2) Write a report summarizing the materials discussed in Topic 4-1 (이번 주 수업에서 공부하는 부분만 요약함)

** 문장으로 써도 좋고 파워포인트 형태의 개조식 정리도 좋음

- Due: see Blackboard
  - Submit electronically to Blackboard

Class Topics (클래스 홈페이지 참조)

- Part 1: Fundamental concepts and principles
- Part 2: 빠른 컴퓨터를 위한 ISA design
  - Topic 1 Computer performance and ISA design (Ch. 1)
  - Topic 2 RISC (MIPS) instruction set (Chapter 2)

    2-1 ALU and data transfer instructions

    2-2 Branch instructions

    2-3 Supporting program execution
  - Topic 3 Computer arithmetic and ALU (Chapter 3)
- Part 3: ISA 의 효율적인 구현 (pipelining, cache memory)