

Integer Addition, Integer Subtraction

- 32 bit 레지스터로 signed numbers 나타낼 경우 (

양수	0 ~ 2^{30} - 1 까지
음수	-1 ~ -($2^{30} - 1$) 까지

)
- 표현할 수 있는 수의 범위를 넘어가면 overflow 발생
 - 양수 + 음수 → no overflow
 - 양수 + 양수 → overflow if result sign 1
 - 음수 + 음수 → overflow if result sign 0

Overflow

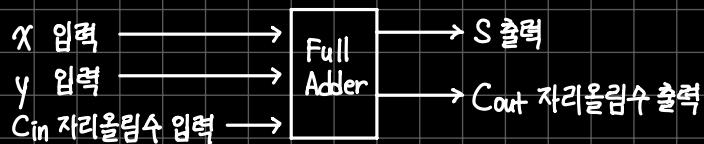
- : 하드웨어가 표현할 수 있는 수의 범위를 넘어서는 경우
- signed
 - (양수끼리 더했는데 음수
음수끼리 더했는데 양수)
- unsigned
 - 원래 나와야하는 결과값보다 작아진 경우
- what to do?
 - ignore 컴퓨터는 overflow 무시하고 연산을 수행
 - call the exception handling routine 예외처리는 개발자의 몫
- 오버플로 발생 조건

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

Arithmetic Logic Unit (ALU) Design

: 모든 명령의 산술 / 논리 계산을 수행

Full Adder (전가산기) 1 Bit ↗ 디노실 범위에서 시험문제 X



Ripple Carry Adder 리플 캐리 가산기

: 전가산기를 여러개 합쳐서 임의의 비트 수 계산 가능

각 비트마다 전가산기 배치

다음 비트의 Carry In (자리올림수 입력) = 이전 비트의 Carry Out (자리올림수 출력)

연산이 느리다 → 이전 가산기의 Carry out 을 기다려야 해서

Hierarchical Carry-Lookahead Adder

리플 캐리 가산기보다 연산이 빠르다

Adder and Subtractor Unit

덧셈과 뺄셈은 기본적으로 같다

단 뺄셈은 한 피연산자의 2의 보수(2's complement)를 구해 이용한다

- 2의 보수 구하기 : XOR 연산으로 1's complement 구하기 $\rightarrow (+1)$ 하면 2's complement

Arithmetic for Multimedia

- Graphics and Media processing - operates on vectors of 8-bit and 16-bit data.

- 64 bit adder 사용, w/ partitioned carry chain

operate on 8×8 bit, 4×16 bit, 2×32 bit vectors

- SIMD 연산 방식 : single instruction, multiple data. 하나의 명령 \rightarrow 여러개의 연산

ex) 32 bit 연산



Multiplication (Unsigned Number)

Multiplicand 피승수 (+14)

Multiplier 곱수 $\times (+11)$

product 결과 $(+154)$

$$\begin{array}{r}
 & 1110 \\
 & + 1011 \\
 \hline
 & 1110 \\
 & + 1110 \\
 \hline
 & 10101 \\
 & + 0000 \\
 \hline
 & 01010 \\
 & + 1110 \\
 \hline
 & 10011010
 \end{array}$$

partial product 0

partial product 1

partial product 2

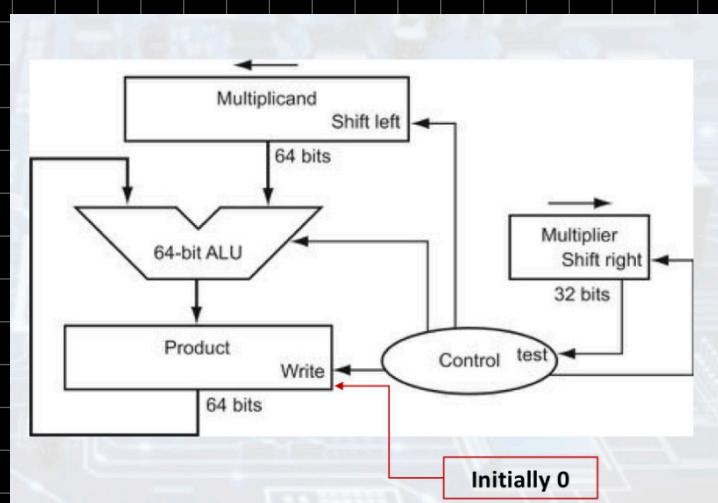
$$\begin{array}{r}
 & 14 \\
 \times & 11 \\
 \hline
 & 14 \\
 & 14 \\
 \hline
 & 154
 \end{array}$$

\leadsto 이런식으로 계산하는거랑 똑같다.

즉. 승수의 자릿수 0 이면 피승수를 해당 위치에 복사한다

승수의 자릿수 1 이면 0을 해당 위치에 복사한다

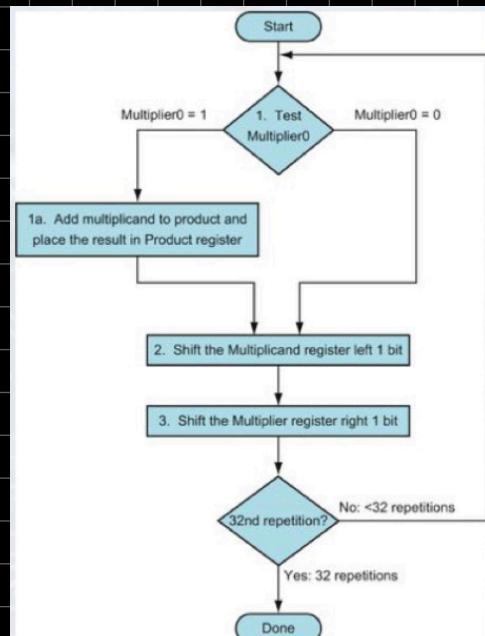
Multiplication Hardware



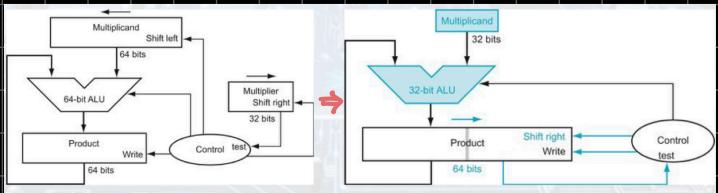
매 단계마다 피승수를 shift left
partial product에 더해준다

승수(Multiplier)는 매 단계마다 shift right

\leadsto 논리구조



Optimized Multiplier



product 결과가 shift right

승수 Multiplier 레지스터가 사라지고 승수를 꼽 product 레지스터의 오른쪽 절반에 배치

Product 레지스터에 사용하지 않는 wasted space 문제
→ product 레지스터의 오른쪽 절반을 multiplier로

Faster Multiplier

: use multiple adders – cost / performance trade off

※ multiply 연산은 대체로 add 연산보다 느리다.

즉, 코드 작성시 add 연산을 사용하는 것이 효율적

RISC-V Multiplication : 4 Instructions

- mul : multiply gives the lower 32 bits of the product
- mulh : multiply high gives the upper 32 bits of the product, assuming the operands are signed
- mulhu : multiply high unsigned gives the upper 32 bits of the product, assuming the operands are unsigned
- mulhsu : multiply high signed / unsigned gives the upper 64 bit of the product,
assuming one operand is signed and the other unsigned

↳ 이 경우는 64bit를 저장할 수 있는 레지스터가 필요

32bit 레지스터가 32개 → multiplication 연산 결과는 64bits 라서 32bits 레지스터에 저장 X

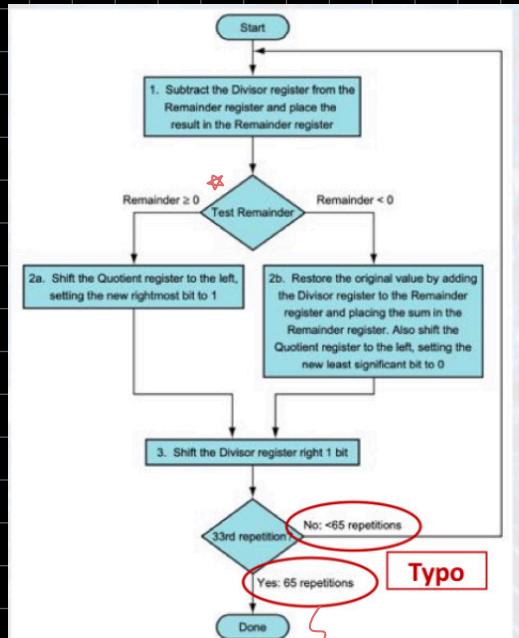
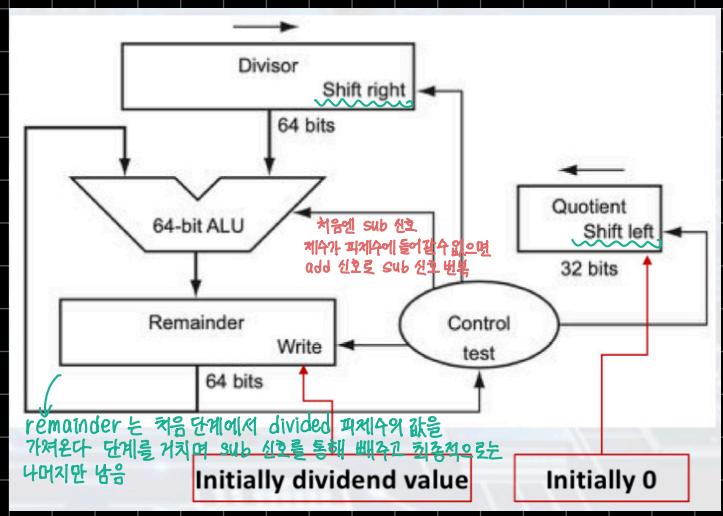
→ 32/32 두 개의 레지스터에 나눠서 product를 저장한다

Division

$$\begin{array}{r}
 & 000/001 \\
 \text{Divisor} \text{ 제수} & 1000 | 100/010 \\
 & -1000 \\
 \hline
 & 000/0 \\
 & 000/01 \\
 & 000/010 \\
 & -1000 \\
 \hline
 & 0010 \text{ Remainder 나머지}
 \end{array}$$

제수가 피제수에 들어갈 수 없으면
Shift right 하고 뒷에 0을 넣어준다

Division Hardware



제수가 피제수에 들어갈 수 있으면 몫에 1 입력
제수가 피제수에 들어갈 수 없으면 몫에 0 입력
+ 뺄셈을 반복한다

32 bit 연산의 경우 $4+1=5$ 번
64 bit 연산이면 $32+1=33$ 번

RISC-V Division : 4 Instructions

- div : signed divide
- rem : signed remainder
- divu : unsigned divide
- remu : unsigned remainder
- Overflow and division-by-zero \Rightarrow doesn't produce errors
 \rightarrow just return defined results
 \rightarrow faster for common case and no error
오버플로와 예외처리는 사람의 몫

Floating Point

- : representation for non-integral numbers
 - including very small ~ very large numbers

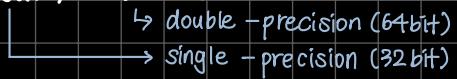
- format of floating point numbers

-2.84×10^{56} : normalized

$\begin{array}{l} +0.002 \times 10^{-4} \\ +987.04 \times 10^9 \end{array} \quad \left. \begin{array}{l} \text{not normalized} \\ \end{array} \right\}$

\Rightarrow In Binary : $\pm 1.xxxxxxx \times 2^{yyyy}$

- C에서 float, double 형



↳ double-precision (64bit)
→ single-precision (32bit)

IEEE Floating Point Format

S	Exponent	Fraction
\downarrow	\downarrow	
Single : fbit double : l1bit	Single : 23bit (+1) double : 52bit (+1)	{ 생략된 1bit 존재 }

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

- S: sign bit (0 : 양수
 1 : 음수)

- Normalize significand : $|1.0 \leq \text{Significand} | < 2.0$

↳ 유효자리에 더 많은 수를 담기 위해 정규화된 이진수의 가장 앞쪽 1bit 생략

Always has a leading pre-binary-point 1 bit,

so no need to represent it explicitly

Significand is Fraction with the 1 restored

- Exponent : excess representation \Rightarrow actual exponent + Bias

- ensures exponent is unsigned

(Single Bias = 127)

Double Bias = 1023

biased notation 을 이용하기 때문 \rightarrow 가장 작은 음수는 000...0 으로 가장 큰 양수는 111...1 을 표기

Single-Precision Range

- Exponents 00000000 and 1111111 reserved

- Smallest value

- Exponent : 00000001 \rightarrow actual exponent = $1 - 127 = -126$

- Fraction : 000...0 \rightarrow significand = 1.0

$\Rightarrow \pm 1.0 \times 2^{-126} \approx 1.2 \times 10^{-38}$

- Largest value

- Exponent : 11111110 \rightarrow actual exponent = $254 - 127 = +127$

- Fraction : 111...1 \rightarrow significand ≈ 2.0

$\Rightarrow \pm 2.0 \times 2^{127} \approx 3.4 \times 10^{38}$

Double - Precision Range

- Exponents 000...0 and 111...1 reserved
- Smallest value
 - Exponent : 000...01 → actual exponent = $1 - 1023 = -1022$
 - Fraction : 000...0 → Significand = 1.0
 $\Rightarrow \pm 1.0 \times 2^{-1022} \approx 2.2 \times 10^{-308}$
- Largest value
 - Exponent : 111...10 → actual exponent = $2046 - 1023 = 1023$
 - Fraction : 000...0 → Significand ≈ 2.0
 $\Rightarrow \pm 2.0 \times 2^{+1023} \approx 1.8 \times 10^{308}$

부동소수점 예제

- represent -0.75

$$-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$$

$$\rightarrow S = 1.$$

$$\rightarrow \text{Fraction} = 1000\cdots0_2$$

$$\rightarrow \text{exponent} = -1 + \text{Bias}$$

$$(\text{Single} : -1 + 127 = 126 = 01111110_2)$$

$$(\text{double} : -1 + 1023 = 1022 = 0111111110_2)$$

$$\therefore \text{Single} : 1011111010000\cdots00_2$$

$$\text{double} : 101111111010000\cdots00_2$$

- 1100 0000 1010 00...00₂ (single precision)

$$\rightarrow S = 1 : \frac{1}{2}$$

$$\rightarrow \text{Exponent} = 1000000_2 = 128$$

$$\rightarrow \text{Fraction} = 010\cdots00_2 \quad 1.01_2 = 1 + 0 \times 2^{-1} + 1 \times 2^{-2}$$

$$x = (-1)^1 \times (1 + 01_2) \times 2^{128-127} = 1 + 0 + 0.25$$

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

$$x = -5.0_{10}$$

Special Encodings of IEEE 754 Floating-point Number (reserved)

Single precision		Double Precision		Object represented
Exp.	Frac.	Exp.	Frac.	
0	0	0	0	0
0	Nonzero	0	Nonzero	\pm denormalized number
1~254	Anything	1~2046	Anything	\pm floating-point number
255	0	2047	0	\pm infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Floating - Point Addition

- Now consider a 4-digit binary example

$$(1.000_2 \times 2^{-1}) + (-1.110_2 \times 2^{-2}) \rightarrow (0.5 + (-0.4375))$$

1. Align Binary Points

- shift number with smaller exponent

$$(1.000_2 \times 2^{-1}) + (-0.111_2 \times 2^{-1}) \text{ 지수부분 통일해준다}$$

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/under flow

$$(1.000_2 \times 2^{-4}) \text{ with no over/under flow}$$

* overflow 는 수가 표현범위보다 커지는 경우

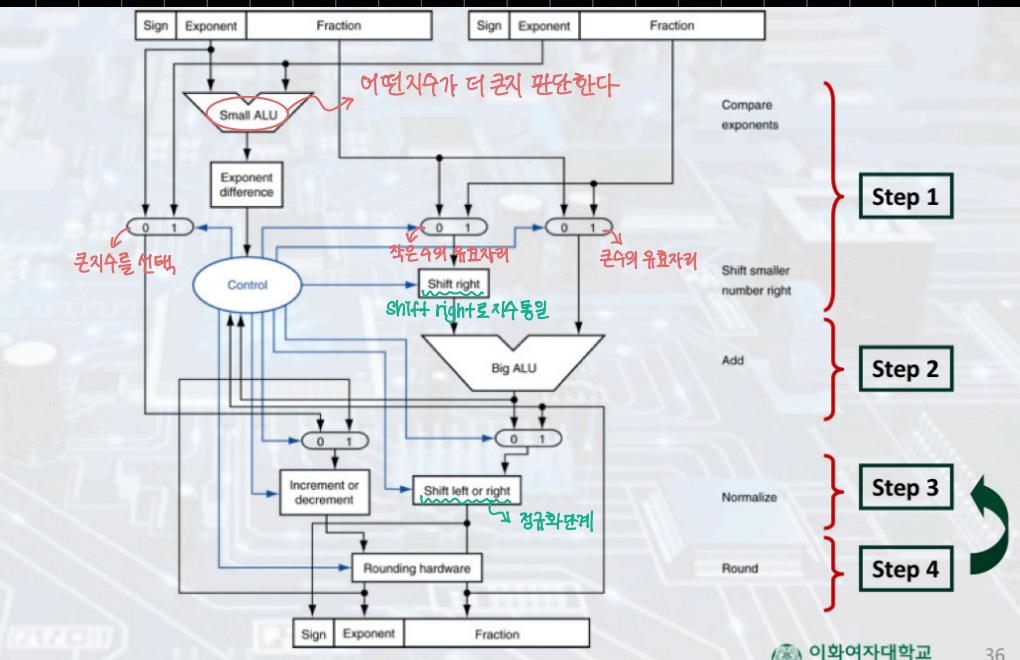
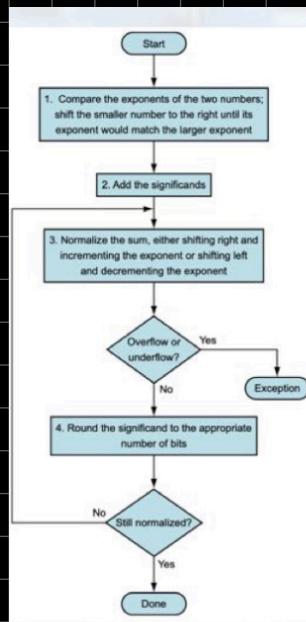
under flow 는 수가 표현범위보다 작아지는 경우

4. Round and renormalize if necessary 유효숫자 맞추기

$$(1.000_2 \times 2^{-4}) = 0.0625$$

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 \rightarrow can be pipelined



Floating - Point Multiplication

- Now consider a 4-digit binary example
 $(1.000_2 \times 2^{-1}) \times (-1.110_2 \times 2^{-2}) \rightarrow (0.5 \times -0.4375)$

1. Add exponents

Unbiased : $(-1) + (-2) = -3$

Biased : $(-1 + 127) + (-2 + 127) = -3 + 254$

$-3 + 254 - 127 = -3 + 127 = 124$

2. Multiply Significands

$1.000_2 \times 1.110_2 = 1.110_2 \rightarrow 1.110_2 \times 2^{-3}$

3. Normalize result

$1.110_2 \times 2^{-3}$ (no change) with no over/underflow

4. Round and renormalize if necessary

$1.110_2 \times 2^{-3}$ (no change)

5. Determine sign : +ve x -ve \rightarrow -ve

$-1.110_2 \times 2^{-3} = -0.21875$

FP Arithmetic Hardware

- FP Multiplier is of similar complexity to FP adder
 - but uses a multiplier for significands instead of an adder
- FP Arithmetic hardware usually does:
 - addition, subtraction, multiplication, division, reciprocal, square-root
 - FP \Leftrightarrow integer 변환
- Operation usually takes several cycles \rightarrow can be pipelined

FP Instructions in RISC-V

RISC-V FP Operands

- 32 floating point registers : f0 - f31

An f-register can hold either a single-precision OR double-precision floating point number

· Single-precision 레지스터는 double-precision 레지스터의 아래쪽 절반이다

· 정수 레지스터 x0 와 달리 부동소수점 레지스터 f0 는 상수 0으로 고정되어 있지 않다

RISC-V FP Assembly Language

fadd.s fadd.d

fsub.s fsub.d

fmul.s fmul.d

fdiv.s fdiv.d

fsqrt.s fsqrt.d

feq.s feq.d equality

flt.s fit.d less than

fle.s fle.d less than or equals

Associativity

- Parallel programs may interleave operations in unexpected orders
→ assumptions of associativity may fail

$$x = -1.50E+38$$

$$y = 1.50E+38$$

$$z = 1.0$$

$$(x+y)+z = 0.00E+00 + 1.0 = 1.00E+00$$

$$x+(y+z) = -1.50E+38 + \underline{1.50E+38} = 0.00E+00$$

↳ $1.50E+38 + 1.0$ 에서 y 에 비해 z 가 너무 작아서 무시된다.

Integer는 정확한 값이 표현되는 반면 FP는 "가장 근사한 값"을 나타내기 때문에 발생하는 문제

→ 큰수부터 연산을 진행하는 것이 해결방법 중 하나

Concluding Remarks

- Bits have no inherent meaning
 - interpretation depends on the instruction applied
- Computer representation of numbers
 - Finite range and precision
 - need to account for this in programs
- ISA supports Arithmetic
 - signed & unsigned integers
 - floating-point approximation to reals
- Bounded range and precision
 - operations can over/underflow