Ch.3 Arithmetic for Computers

▼ 3.2 덧셈, 뺄셈

▼ Arithmetic for computers

- operations on integers → 사칙연산, overflow
- floating-point real numbers → 표현방법, 연산

- Number Representations
    - 32-bit signed number(2의 보수)

```
0000 0000 0000 0000 0000 0000 0000 0000two = 0ten
0000 0000 0000 0000 0000 0000 0000 0001two = + 1ten
...
0111 1111 1111 1111 1111 1111 1111 1110two = + 2,147,483,646ten
0111 1111 1111 1111 1111 1111 1111 1111two = + 2,147,483,647ten
1000 0000 0000 0000 0000 0000 0000 0000two = – 2,147,483,648ten
1000 0000 0000 0000 0000 0000 0000 0001two = – 2,147,483,647ten
...
1111 1111 1111 1111 1111 1111 1111 1110two = – 2ten
1111 1111 1111 1111 1111 1111 1111 1111two = – 1ten
```

maxint

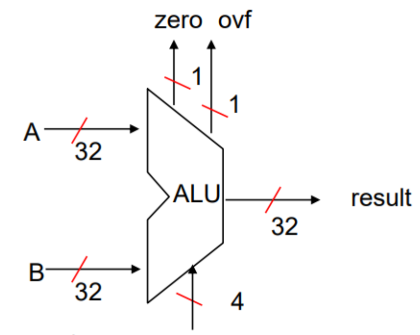
minint

MSB

LSB

- 2의 보수
    - ① 음수의 2의 보수 → 모든 bit complement 후 add 1

      ↳ 첫번째 1이 나온 이후 모두 바꾸기
    - ② n bit짜리 수를 n bit보다 더 많은 bit로 바꾸기 (부호확장)
        - MIPS : 16 bit를 32 bit로 채움
        - 부호 비트를 copy해서 확장할 것 → sign extension
            - lbu (0으로 확장, unsigned) 0100 → 0000 0100
            - lb (1로 확장 → signed) 1001 → 1111 1001
- MIPS Arithmetic Logic Unit ⇒ ALU
    - ISA 대부분의 연산을 지원



xori

- arithmetic → sign extend.
  - *add, addi, addiu, addu*
  - *sub, subu* → overflow detection.
  - *mult, multu, div, divu* → overflow(X)
  - *sqrt*
- logic — zero extend.
  - *and, andi, nor, or, ori, xor, xori*
  - *beq, bne, slt, slti, sltiu, sltu*

◦ spcial handling

- ▶ sign extend – `addi, addiu, slti, sltiu`
- ▶ zero extend – `andi, ori, xori`
- ▶ overflow detection – `add, addi, sub`

▼ Dealing with overflow

① Integer addition

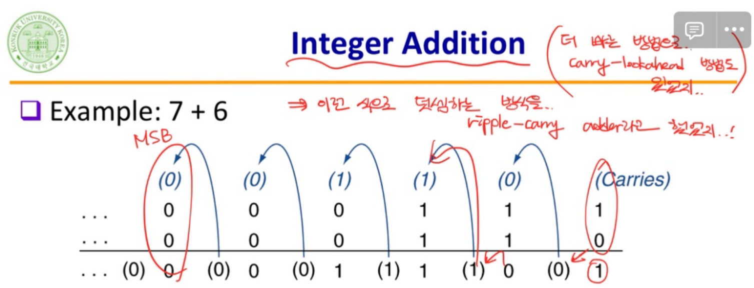
ex) 7 + 6

7 : 0 0 0 1 1 1

6 : 0 0 0 1 1 0

13 : 0 0 1 1 0 1

⇒ overflow



↳ overflow

- positive + negative → no overflow
- positive + positive = sign bit이 1 → overflow

- negative + negative = sign bit이 0 → overflow

sign : 0 + 0 = 1 → overflow
sign : 1 + 1 = 0 → overflow

2. Integer Subtraction
   - 7 - 6 = 7 + (-6)

+7:	0000 0000 … 0000 0111
–6:	1111 1111 … 1111 1010
+1:	0000 0000 … 0000 0001

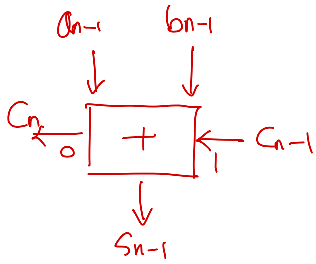
   ⇒ overflow

sign : 0 − 1 = 1 → overflow
sign : 1 − 0 = 0 → overflow

   - overflow → 표현가능한 최소수보다 작거나 최대보다 크면.
     - positive - positive / negative - negative → no overflow
     - positive - negative = sign bit이 1 → overflow
     - negative - positive = sign bit이 0 → overflow

- Dealing with overflow ⇒ 계산결과가 일관성을 유지해야 함.
   - 32 bit 내에서 표현할 수 없을 때 일어남
     - sign bit이 결과의 value bit 포함 and sign bit 불일치
   - detecting overflow
     - 2의 보수 덧셈 → 두 수의 부호가 같은데 결과는 다를 때 발생

$$
\begin{array}{rcccc}
 & a_{n-1} & a_{n-2} \cdots & a_1 & a_0 \\
+ & b_{n-1} & b_{n-2} \cdots & b_1 & b_0 \\
\hline
= & s_{n-1} & s_{n-2} \cdots & s_1 & s_0
\end{array}
$$



↳ 여기만 확인하면 over/underflow 알 수 있음.

음수 음수 양수 양수 양수 음수.

$$V = a_{n-1} \times b_{n-1} \times \overline{s_{n-1}} + \overline{a_{n-1}} \times \overline{b_{n-1}} \times s_{n-1}$$

$$V = c_n \otimes c_{n-1}$$

↳ carry in, carry out 서로 다를 때에도 overflow 발생.

- carry in, carry out → MSB

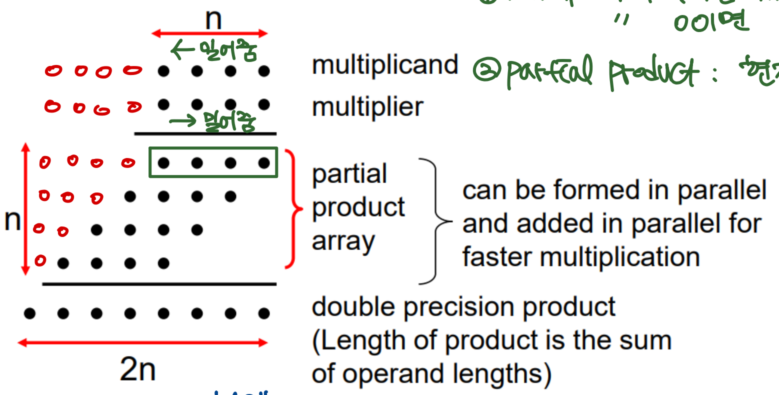
   - C : ignore overflow
   - Fortran : exception 발생

→ exception handler의 유무에 따라서 처리 방식 다름.

▼ 3.3 곱셈

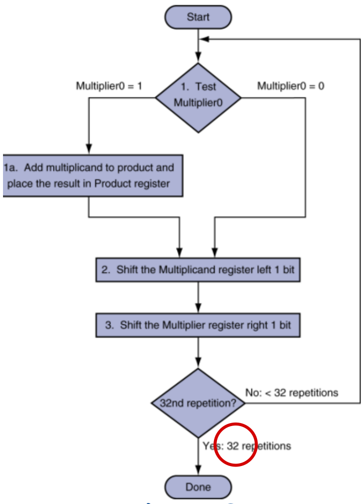
▼ Binary multiplication : right shift → add

① multiplier가 1이면 multiplicand 그대로
〃 0이면 〃 0으로
② partial product : 현재 자리보다 왼쪽으로 한칸씩

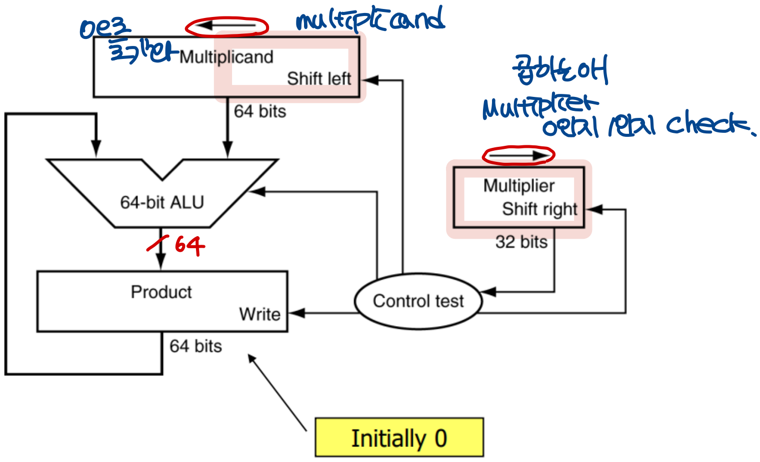


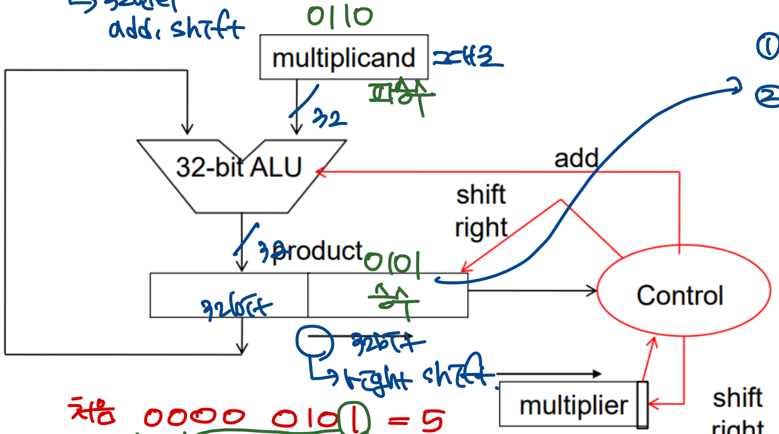
32비트계산 → 64비트까지 가능.

▼ Multiplication hw



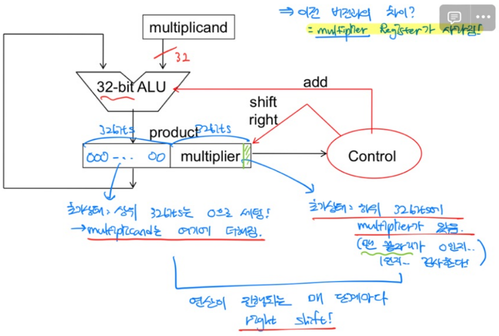
32번의 add, shift





① 초기상태 → 모두 0
② 끝자리가 1이면 값을 더함.

처음 0000 0101 = 5
+) 0110
add 0110 0101
shift → 0011 0010
add 0011 0010
shift → 0001 1001
+) 0110
add 0111 1001
shift → 0011 1100
add 0011 1100
shift → 0001 1110



▼ MIPS Mulitply instruction

```
mult $s0, $s1  # hi||lo = $s0 * $s1
```

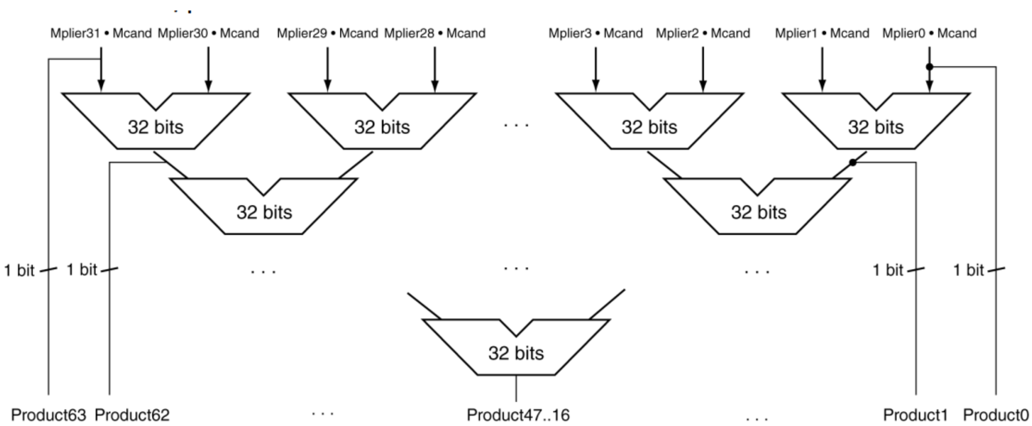
(or multu)

0	16	17	0	0	0x18

- 32-bit registers for product
  - *hi* : MSB → 32bits 상위 32 bit 저장
  - *lo* : LSB → 32bits 하위 32 bit 저장
- instructions : register에 product 옮기는 명령어
  - *mfhi rd* : move from lo
  - *mflo rd* : mover form hi

▼ Fast Multiplier ⇒ adder보다 빠름.

- 32번의 adder를 병렬구조로 시행 → log32 = 5 정도로 걸림, 비용도 절약



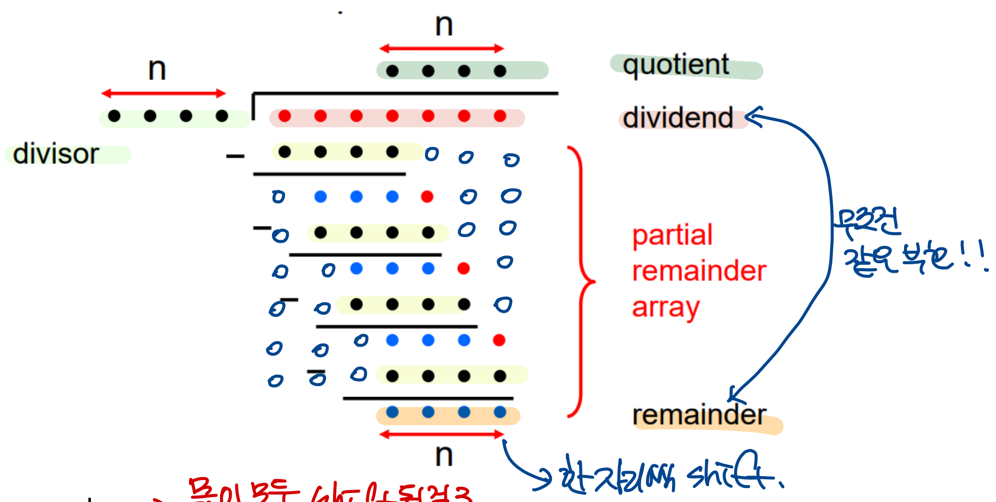
- 파이프라이닝 가능 → 다수의 곱셈도 병렬적으로 수행 가능 ⇒ 더 빨리 가능

▼ 3.4 나눗셈

▼ Binary Division :

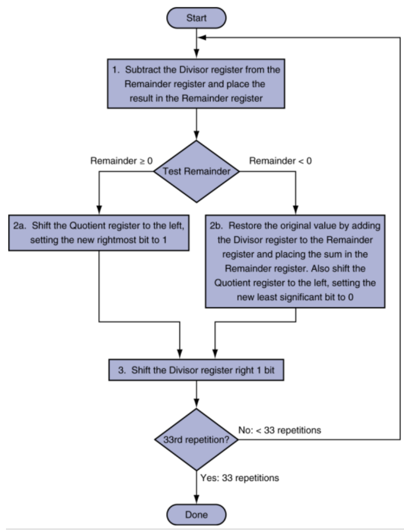
- dividend(나눠지는 수) = quotient(몫) X divisor(나눌 수) + remainder

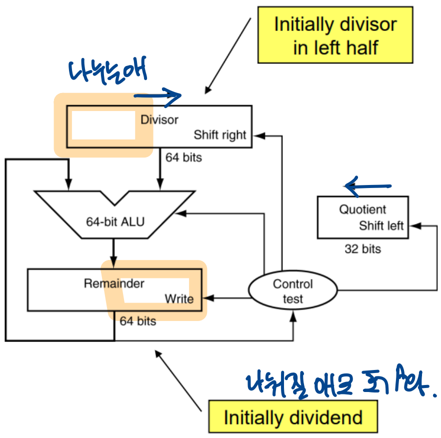
연산 반복 ⇒ 음수되면 복구해줌.



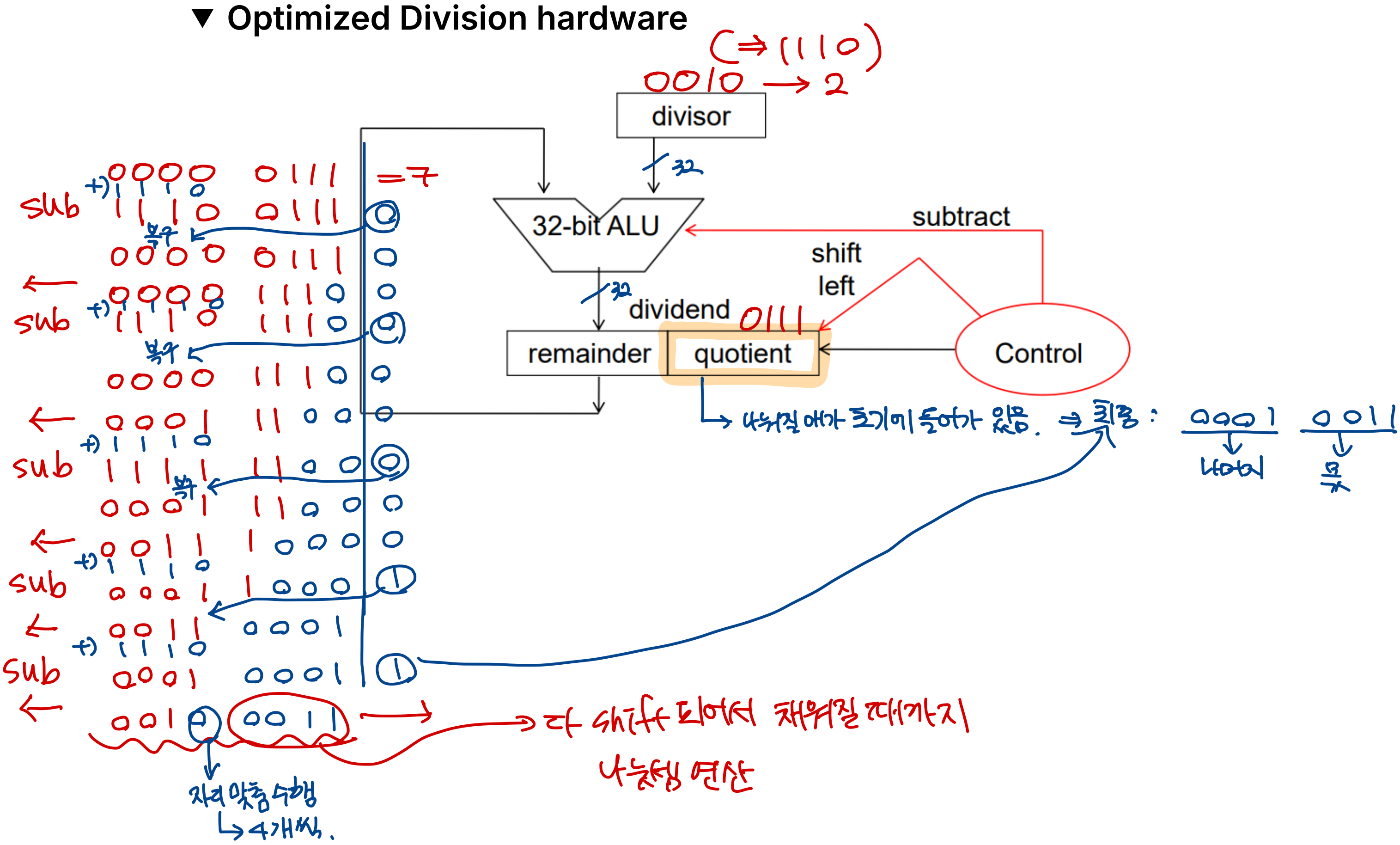
- division approach → 몫이 모두 shift 될걸로
채워질 때까지 연산!
    1. 나누는 애가 0인가?
    2. long division
        a. 나누는 애 bit ≤ 나눠지는 애의 bit
            i. 몫 = 1 , 나눠지는 애에서 빼줌
        b. otherwise
            i. 몫 = 0, 다음 나눠지는 애에서 가져옴
    3. restoring division 몫 [0보다 ↑ → 1 / 0보다 ↓ → 0
        a. 뺄셈 후 남은 놈이 0보다 작다 → 다시 나누는 애 더해줌 ★
        restore.
    4. signed division
        a. 절대값으로 나눠야 함
        b. MIPS : 나눠지는 애랑 나눠지고 있는 남은 애들은 다 같은 부호여야 함

▼ Division hardware





▼ Optimized Division hardware



▼ MIPS Divide Instruction

```
div $s0, $s1 # lo = $s0 / $s1
             # hi = $s0 mod $1
```

0	16	17	0	0	0x1A

- ***mfhi rd, mflo rd***
  - 결과에 접근 가능 → register 안에 있는 몫, reminder 옮길 수 있음
- 몫이 너무 크면 divide는 oveflow 무시함.
  - sw: 0으로 나눠지는 것을 방지하기 위해 반드시 check함.

▼ 3.5 부동 소수점

▼ Floating point

- 너무 큰 수, 너무 작은 수 32 bit로 표현 → 부동 소수점 사용
- notation

- normalized : 소수점 앞에 한 자리만 + 0이 아닌 숫자로 표현
  - ex) -2.34 X 10^56
  - 유효 숫자 몇 개까지 있는지 표현 ☆
- not normalized : $\pm 1.xxxxxxx_2 \times 2^{yyyy}$ frac → exp
  - ex) 0.23 X 10^9 , 234.3 X 10^7

- type : float, double in C

E-Bias = e
0010. 2^0.

▼ Floating point representation

→ 2진수 표현법

F → 유효숫자.

s	E (exponent)	F (fraction)

⇒ 어떻게 표현하는지에 따라서 표현의 범위가 달라짐.

- fraction(F)와 exponent(E) 크기 사이에서 타협점 찾아야 함
  - 정밀도/표현 범위 타협
- floating point standard → IEEE, single/double 기준 ☆
  - IEEE 754 FP Standard → bias 필요함

unsigned

$\pm 1.xxxx_2 \times 2^{yyyy}$

	s	E (exponent)	F (fraction)
single	1 bit	8 bits	23 bits 32
double	1 bit	11 bits	52 bits 64

$$(-1)^{sign} \times (1+F) \times 2^{e(=E\text{-bias})}$$

- F : normalized format ⇒ hidden bit를 더해줘야 함 ☆

$$1.xxxx_2 \times 2^{yyyy}$$

- E : excess (biased) notation 표현 ⇒ FP 구분 단순화
  - single : Bias = 127 ⇒ -126 ~ 127 사용 ⇒ Smallest Value : 1
  - double : Bias = 1023 ⇒ -1022 ~ 1023 사용

- floating point example
  - -0.75 $= \frac{-3}{2^2}{}_{(10)} = -11 \times 2^{-2}{}_{(2)} = -1.1 \times 2^{-1}{}_{(2)}$

S = 1
F = 1.1000···0 (hidden bit)
Exponent = e + Bias
single : -1 + 127 = 126
Double : -1 + 1024 = 1023

	1 bit	8bit / 11bit	23bit / 54bit	
⇒ single	1	01111110	10000 ···· 000	32bit
Double	1	01111111110	10000 ··· 000	64bit

single

- 1|1000|0001|01000 ... 00
  ↳ 129

  $s = 1$
  $e = E - Bias = 2$
  $F = .0100 \cdots 0$

  $\Rightarrow (-1)^1 \times 1.01 \times 2^2 = (-1)^1 \times 101_{(2)} = -5$

- IEEE 754 FP Normalized Form
  - Normailized form (single) with a hidden bit
    - E : 0000 0001 ~ 1111 1110 (−126 ~ 127)
    - F : Any
  - example ⇒ normalized format

	sign	E (exp) 8bit	hidden bit	F (frac) 23bit	
양수중 가장작은수 smallest+	0	0000 0001 ⇒1	1.	0000 ..... 00000 ⇒1	$= 1 \times 2^{1-127}$
양수 중 가장 큰수 largest+	0	1111 1110 ⇒254	1.	1111 ..... 111111 ⇒2^{-23}	$= 2 - 2^{-23} \times 2^{254-127}$

ex) $1.0_2 \times 2^{-1} = 0\ 01111110\ (1)\ \underbrace{0000 \cdots 0}_{23}$

$0.75_{10} \times 2^4 = \frac{3}{4}_{10} \times 2^4 = 3 \times 2^2 = 11 \times 2^2 = 1.1 \times 2^3 = 0\ 10000010\ 1.100 \cdots 0$

- IEEE 754 FP Standard Encoding
  - special encodings : unusual event 표현
    - 0 : 모든 bit가 0
    - 무한대 : 어떤 수를 0으로 나눔
    - NAN : 0/0과 같은 invaild operations의 결과

Single Precision		Double Precision		Object Represented
E (8)	F (23)	E (11)	F (52)	
0000 0000	0	0000 … 0000	0	true zero (0) ⇒ 모든것이다 0임.
0000 0000	nonzero	0000 … 0000	nonzero	± denormalized (0.xxx) number → 정수부분이 1로시작X (hidden 적용X)
0000 0001 to 1111 1110 (−126 ~ 127)	anything	0000 …0001 to 1111 …1110	anything	± floating point number ⇒ normalized
1111 1111	+ 0	1111 … 1111	- 0	± infinity
1111 1111	nonzero	1111 … 1111	nonzero	not a number (NaN)

Denormal Numbers

- Exponent = 000 ... 0 → hidden bit is 0
  ↑ 0인걸 의미

$0.\text{XXX}\cdots$

single -> -126
double -> -1022

$$x = (-1)^S \times (0 + \text{Fraction}) \times 2^{1-\text{Bias}}$$ ★

- smaller than normal numbers ⇒ normal한 숫자보다 더 작은 숫자로 표현
  - precision점점 작아져서 0으로 다가감 → 0으로 부호 표현 가능
- Fraction = 000... 0

$$x = (-1)^S \times (0 + 0) \times 2^{1-\text{Bias}} = \pm 0.0$$

Two representations of 0.0!

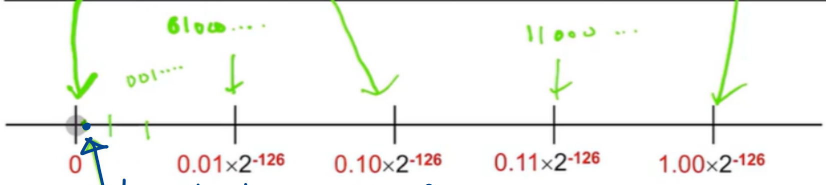
- Denormalization

→ exponent = 0

▶ True zero is the bit string all zero. ★

Single Precision		Double Precision		Object Represented
E (8)	F (23)	E (11)	F (52)	
0000 0001	0000....0000	00000000001	0000....0000	Smallest Normalized No
0000 0000	Nonzero	00000000000	Nonzero	±Denormalized No.
0000 0000	0000....0000	00000000000	0000....0000	true zero

→ Normalized 중가장 ↓



denormalized 가장 작은수 ⇒ frac = 000...1

$= 0.00\cdots1 \times 2^{-126}$

특수 case

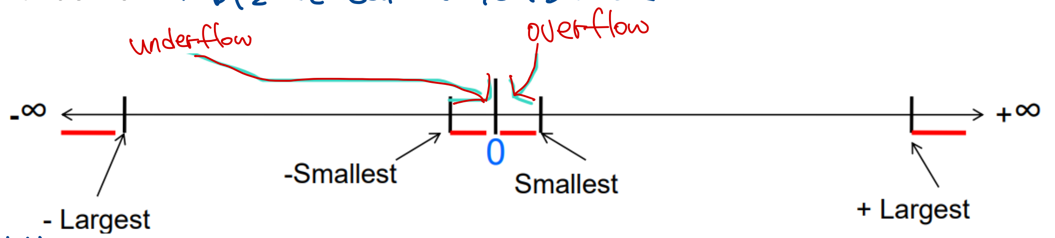
▼ Infinites and NaNs

- Exp = 111 ... 1, Frac = 000...0
  - +/- infinity
  - subsequent 계산에 사용 가능 →overflow check 필요 x
- Exp = 111 .... 1 , Frac ≠ 0000..00
  - Nan
  - undefiend result
  - subsequent 계산에 사용 가능

▼ Exception Events in FP

- overflow : 절댓값이 아주 클 때 발생

- underflow : 음수를 표현하는데 너무 작은 수를 표현할 때 ⇒ exp이 절댓값 너무 커짐!



*해결방안.

⇒ 다른 format이 더 큰 exp field 가지고 있으면 방지 가능

Double precision – takes two MIPS words

s	E (exponent)	F (fraction)
1 bit	11 bits	20 bits

F (fraction continued)
32 bits

▼ Support for accurate arithmetic

- IEEE 754 rounding modes(자리맞춤)
  - round up (항상 올림), round down (항상 내림), turncate (버림), round to nearest even (반올림)
- rounding : extra F bit 필요함
  - Guard bit : result normalize → shift left 1bit
  - Round bit : rounding accuracy 향상
  - Sticky bit : round to nearest even(가장 가까운 짝수로의 자리맞춤)
    - 1 bit shift 할 때마다 1로 setting

F = 1 . xxxxxxxxxxxxxxxxxxxxxxx G R S

G=1 ─ R=1 : 올림
G=0 : 내림
R=0 ─ S=1 : 올림
S=0
- 마지막 : 0 → 버림
- 마지막 : 1 → 올림

⇒ 가장 가까운 짝수 만들기

▼ Floating point addition

- addition(subtraction)
  1. F1, F2 → hidden bit restore
  2. 둘 중 작은 수의 E를 큰 것과 일치할 때까지 shift right
     큰거에 맞춰줌!! ⇒ shift.
     ↳ shift → bit수 넘어가면 Rne 반올림.
  3. F1 + F2 = F3
  4. F3 normalize ⇒ 1, xxx
     a. F1, F2 같은 부호 + 0<F3<4 : 1 bit right shift F3 + increment E3 (overflow) ☆
     b. F1, F2 다른 부호 : many left shifts each time decrementing E3 (underflow) ☆

→ 표현할 수 없을만큼 넘어가면 반올림

5. Round F3 , normalize F3
6. rehide the MSB ⇒ 표현 변경
   - example

$0.5 = (1.0000 \times 2^{-1}) + (-0.4375 = -1.1100 \times 2^{-2})$

복원완료

0) add the hidden bit

1) -1.1100 → shift right ⇒ 2^{-1}로 맞추기

⇒ -0.111×2^{-1}

2) add significands : $1.0000 + (-0.111) = 0.001$

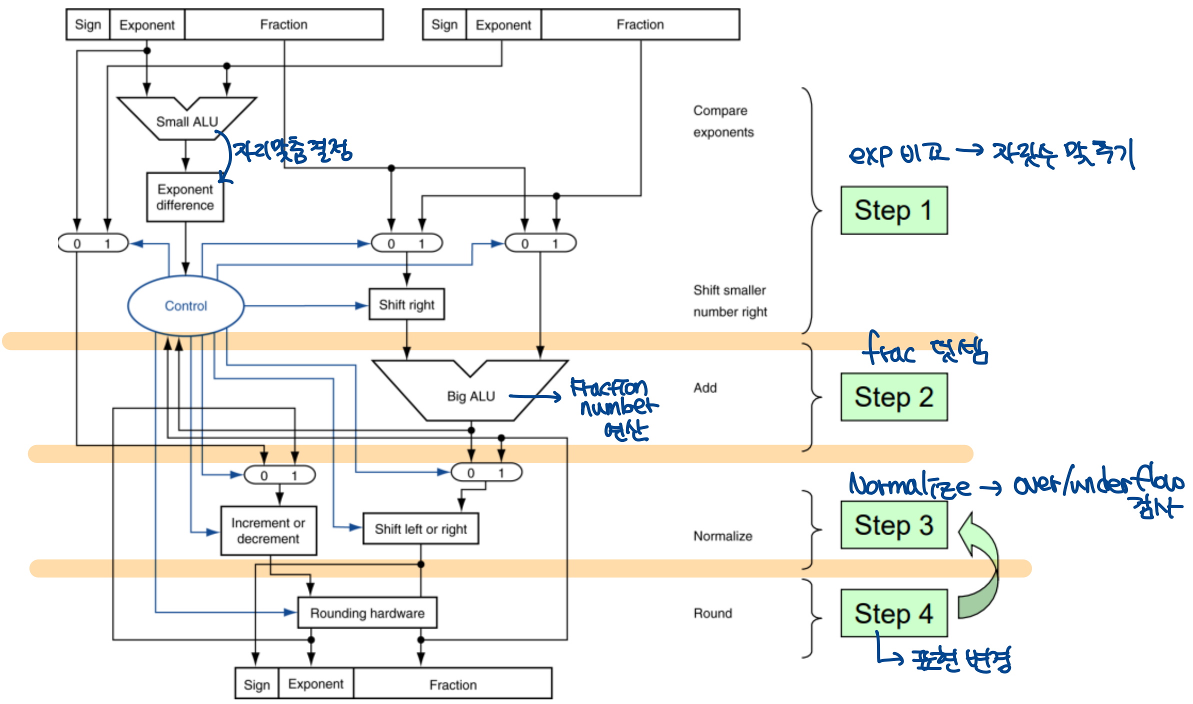
3) normalize the sum, checking exp over/underflow → $-4+127=123$ overflow/underflow X

$0.001 \times 2^{-1} = 0.010 \times 2^{-2} = \dots = 1.000 \times 2^{-4}$

4) 1.000×2^{-4} ⇒ shift left 4칸 ⇒ $0.0001.000$ ⇒ $0.0001000_{(2)}$ $= \frac{1}{16}$

5) rehide the hidden bit before storing

- FP adder hw



   - integer adder보다 훨씬 복잡
   - 한 clock cycle에서 수행 → integer operations보다 오래 걸림
     - slower clock : 모든 instructions 병렬적으로 수행
   - FP adder : 여러 cycle에 걸쳐 수행 → 병렬적으로 처리 가능

→ 할 수 있으면 integer로.

Floating point Multiplication

- multiplication

1. F1, F2 → hidden bit restore
2. biased 수를 더한 뒤 bias 빼기 ⇒ 지수계산, 부호결정 (sign bit 같으면 양수)
   a. E1 + E2 - 127 = E3 = $e_1 + e_2 + 127$
      (E1 = $127+e_1$, E2 = $127+e_2$)
3. F1 * F2 = F3(double precision)
4. F3 normalize ⇒ 1, xxx
   a. F1, F2 normalized + 0<F3<4 : 1 bit right shift F3 + increment E3 (overflow)
   b. overflow, underflow check
5. Round F3 , normalize F3 (반올림)
6. rehide the MSB

- example $0.5 = (1.0000 \times 2^{-1}) \times (-0.4375 = -1.1100 \times 2^{-2})$
  복원완료

0) add the hidden bit

1) add the two (biased) exponents and subtract the bias from the sum,

$E_1 = -1$, $E_2 = -2$

E_3 (biased) $= -1 - 2 + 127 = 124$

2) multiplication $F_1 \times F_2 \longrightarrow 1.110000 \times 2^{-3}$

3) normalization → multiplication 완료

over, underflow → $-126 \leq -3 \leq 127$

4) 1.110×2^{-3} ⇒ 자리맞춤.

5) 부호가 다름 ⇒ 결과가 음수 ⟶ -1.110×2^{-3}

⇒ 10진수로 변환 $-1.110 \times 2^{-3} = 0.001110 = \frac{1}{8} + \frac{1}{16} + \frac{1}{32} = -\frac{7}{32}$

```
  1.000
  1.110
-------
   0000
  1000
 1000
1000
-------
1110000
```
GRS
내림

- FP Arithmetic hw
  - FP multiplier : FP adder와 복잡도가 유사
  - FP arithmetic hw → 사칙연산, reciprocol(역수), square-root ↔ integer conversion
  - operation → 여러 사이클에 걸쳐 병렬적으로 수행됨

FP instructions in MIPS

- FP h/w : coprocessor 1 (ISA 확장)
- MIPS : 분리된 FP register file 가지고 있음
  - 32 single-precision : \$f0, \$f1, \$f2, ... \$f31
  - paired double-precision : \$f0/\$f1, \$f2/\$f3, ... → register 두개씩 사용.
- FP instructions : FP register에서만 operate

- MIPS FP instructions
  - load and store

```
lwcl  $f1,54($s2)   #$f1 = Memory[$s2+54]
swcl  $f1,58($s4)   #Memory[$s4+58] = $f1
```

  - supports IEEE 754
    - single

```
add.s $f2,$f4,$f6  #$f2 = $f4 + $f6
```

    - double

```
add.d $f2,$f4,$f6  #$f2||$f3 =
                         $f4||$f5 + $f6||$f7
```

  - similarly for

similarly for `sub.s, sub.d, mul.s, mul.d, div.s, div.d`

  - floating point insrtuctions

```
c.x.s $f2,$f4       #if($f2 < $f4) cond=1;
                        else cond=0
```

  - double precision comparison

```
c.x.d $f2,$f4       #$f2||$f3 < $f4||$f5
                        cond=1; else cond=0
```

  - floating point branch

```
bclt  25            #if(cond==1)
                        go to PC+4+25

bclf  25            #if(cond==0)
                        go to PC+4+25
```

▼ 3.5 Subword Parallellism

▼ subword parallellism

- 그래픽, 오디오 어플리케이션 : 데이터의 벡터에 같은 연산을 반복 수행
  - example : 128-bit adder
    - 8비트 연산자 16개 or 16비트 연산자 8개 or 32bit 연산자 4개 동시 연산 가능
- 데이터 수준 병렬성, 벡터 병렬성, SIMD라고도 함

▼ 3.9 오류 및 함정

▼ Fallacy : 1bit shift left = 2를 곱해준 것 ⇒ 1 bit shift right = 2로 나눈 것

⇒ 부호 없는 정수에서만 가능

⇒ 부호 있으면 오류 발생

e.g., –5 / 4

- $11111011_2 >> 2 = 11111110_2 = -2$
- Rounds toward –∞

c.f. $11111011_2 >> 2 = 00111110_2 = +62$ logical shift 경우.

▼ Pitfall : floating-point addition → 결합법칙 성립 x (c +(a+b) = (c+a)+b) ⇒ FP에서는 성립 X

- 병렬프로그램 → 예상되지 않은 순서로 interleabe operations 결합법칙 성립 X.

▼ Fallacy : integer type에서 사용되는 병렬 수행 방식은 FP type에도 적용

- 결합 법칙이 성립하지 않으므로 옳지 않은 가정

▼ Pitfalls : MIPS addiu → 16비트의 immediate filed를 부호확장하여 사용 X sign extends ★.