3장. 컴퓨터 산술과 논리 연산

목차

- 3.1 ALU의 구성 요소
- 3.2 정수의 표현
- 3.3 논리 연산
- 3.4 쉬프트 연산
- 3.5 정수의 산술 연산
- 3.6 부동소수점 수의 표현
- 3.7 부동소수점 산술 연산

컴퓨터의 기본 기능

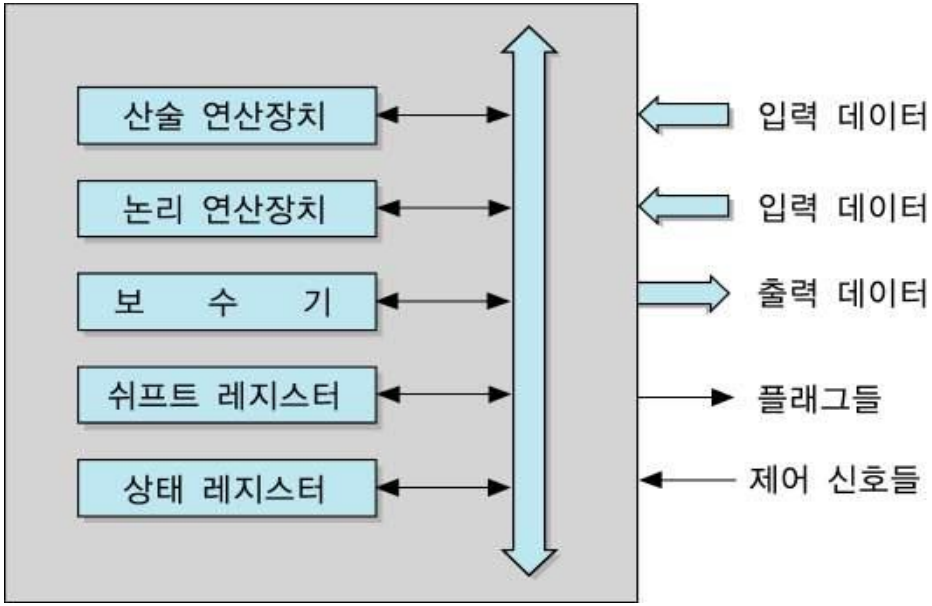
- 산술적 계산
- 논리 데이터에 대한 연산
- → 산술논리연산장치 (ALU: Arithmetic and Logical Unit)
- 산술적 계산
  - 정수, 부동소수점 수
- 논리 연산
  - 2진 데이터

3.1 ALU의 구성 요소

- ALU
  - CPU 내부 구성요소 중 하나
  - 컴퓨터 시스템의 다른 요소(제어유니트, 레지스터, 주기억장치, I/O장치)는 ALU가 처리하는 데이터를 가져오고 결과 저장, 출력
- 산술 연산
  - 산술 연산들(+, -, ×, ÷)을 수행
- 논리 연산
  - 논리 연산들(AND, OR, XOR, NOT 등)을 수행
- 쉬프트 레지스터(shift register)
  - 비트들을 좌측 혹은 우측으로 이동시키는 기능을 가진 레지스터
- 보수기(complementer)
  - 2진 데이터를 2의 보수로 변환(음수화)
- 상태 레지스터(status register)
  - 연산 결과의 상태를 나타내는 플래그(flag)들을 저장하는 레지스터

ALU의 동작

- ALU가 처리하는 데이터
  - 입력: 레지스터/주기억장치 → ALU
  - 저장: ALU→ 레지스터
- 상태 레지스터
  - ALU가 연산의 결과에 따라 상태 레지스터의 플래그 값을 세트
  - 플래그 값은 조건 분기 명령, 산술명령에 의해 사용됨
- 제어유니트
  - 입력 데이터에 대한 연산을 수행할 내부 요소 선택, ALU 내외로 데이터 이동 제어신호 발생



3.2 정수의 표현

- 2진수 : 0, 1, 부호, 소수점으로 표현

  $-13.625_{10} = -1101.101_2$

- 컴퓨터의 데이터 저장/처리
  - 부호, 소수점 사용 안 함
  - 0과 1만 사용
- 부호 없는 정수 표현의 예 (양수, 8비트)

  00111001 = 57

  00000000 = 0

  00000001 = 1

  10000000 = 128

  11111111 = 255

- n-비트 2진수를 부호 없는 정수 A로 변환하는 방법

  $A = a_{n-1} \times 2^{n-1} + a_{n-2} \times 2^{n-2} + \ldots + a_1 \times 2^1 + a_0 \times 2^0$

소수와 음수의 표현

- 최상위 비트인 a_{n-1}의 좌측에 소수점이 있는 소수의 10진수 변환방법
  - $A = a_{n-1} \times 2^{-1} + a_{n-2} \times 2^{-2} + \ldots + a_1 \times 2^{-(n-1)} + a_0 \times 2^{-n}$
  - $$\begin{array}{cccccccc} 2^3 & 2^2 & 2^1 & 2^0 & & 2^{-1} & 2^{-2} & 2^{-3} \\ 1 & 1 & 0 & 1 & . & 1 & 0 & 1 \end{array}$$

    $= 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0 + 1\times2^{-1} + 0\times2^{-2} + 1\times2^{-3}$

    $= 8 + 4 + 1 + 0.5 + 0.125 = 13.625$
- 음수 표현 방법 (부호 비트 사용: 0 or 1, 왼쪽 첫 비트)
  - 부호-크기 표현 (signed-magnitude representation)
  - 1의 보수 표현 (1's complement representation)
  - 2의 보수 표현 (2's complement representation)

3.2.1 부호-크기 표현

- 부호: 맨 좌측 비트
- 크기: 나머지 n-1 개의 비트
- 부호-크기(magnitude) 표현 방식

  [예] + 9 = 0 0001001 + 35 = 0 0100011

  - 9 = 1 0001001 - 35 = 1 0100011

- 부호-크기로 표현된 2진수(a_{n-1} a_{n-2} … a_1 a_0)를 10진수로 변환
  - $A = (-1)^{a_{n-1}} (a_{n-2} \times 2^{n-2} + a_{n-3} \times 2^{n-3} + \ldots + a_1 \times 2^1 + a_0 \times 2^0)$

    0 0100011 $= (-1)^0 (0\times2^6+1\times2^5+0\times2^4+0\times2^3+0\times2^2+1\times2^1+1\times2^0)$

    $= (32 + 2 + 1) = 35$

    1 0001001 $= (-1)^1 (0\times2^6+0\times2^5+0\times2^4+1\times2^3+0\times2^2+0\times2^1+1\times2^0)$

    $= - (8 + 1) = - 9$

국립 한경대학교 HANKYONG NATIONAL UNIV.

부호-크기 표현 (계속)

- 단점
  - 덧셈과 뺄셈을 수행하기 위해서는 부호비트와 크기 부분을 별도로 처리
  - 0 표현이 두 개 존재

    ➔ n-비트 단어로 표현할 수 있는 수들이 2^n 개가 아닌, $(2^n - 1)$개로 감소

    0 0000000 = + 0

    1 0000000 = - 0

3.2.2 보수 표현

- 1의 보수(1's complement) 표현
  - 모든 비트들을 반전 (0 → 1, 1→ 0)
- 2의 보수(2's complement) 표현
  - 모든 비트들을 반전하고, 결과값에 1을 더함
- 양수는 1의 보수 2의 보수가 동일
- 음수의 표현
  - [예]
    - \+ 9 = 0 0001001   + 35 = 0 0100011
    - \- 9 = 1 1110110 (1의 보수)   - 35 = 1 1011100 (1의 보수)
    - \- 9 = 1 1110111 (2의 보수)   - 35 = 1 1011101 (2의 보수)

8-비트 보수로 표현된 정수들

- 8-비트 2진수로 표현할 수 있는 10진수의 범위
  - 1의 보수 : $-(2^7 - 1) \sim +(2^7 - 1)$
  - 2의 보수 : $-2^7 \sim +(2^7 - 1)$

10진수	1의 보수	2의 보수
127	01111111	01111111
126	01111110	01111110
⋮	⋮	⋮
1	00000001	00000001
+ 0	00000000	00000000
− 0	11111111	−
− 1	11111110	11111111
− 2	11111101	11111110
⋮	⋮	⋮
− 126	10000001	10000010
− 127	10000000	10000001
− 128	−	10000000

2의 보수 → 10진수 변환

- 2의 보수로 표현된 양수(a_{n-1} = 0)를 10진수로 변환하는 방법
  - $A = a_{n-2} \times 2^{n-2} + a_{n-3} \times 2^{n-3} + \ldots a_1 \times 2^1 + a_0 \times 2^0$
- 2의 보수로 표현된 음수(a_{n-1} = 1)를 10진수로 변환하는 방법
  - $A = -2^{n-1} + (a_{n-2} \times 2^{n-2} + a_{n-3} \times 2^{n-3} + \ldots a_1 \times 2^1 + a_0 \times 2^0)$

  [예] $10101110 = -128 + (1 \times 2^5 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1)$

  $= -128 + (32 + 8 + 4 + 2) = -82$

  [다른 방법]

  10101110 → 01010010(대응하는 양수)으로 먼저 변환한 후, - 부호 추가

  $01010010 = -(1 \times 2^6 + 1 \times 2^4 + 1 \times 2^1)$

  $= -(64 + 16 + 2) = -82$

3.2.3 비트 확장 (Bit Extension)

- 데이터의 길이(비트 수)를 늘리는 방법
  - 목적: 데이터를 더 많은 비트의 레지스터에 저장하거나 더 긴 데이터와의 연산 수행

예제 3-7

10진수 '21'과 '-21'에 대한 8-비트 길이의 부호화-크기 표현을 16-비트 길이로 확장하라.

풀이

$+21=$	00010101	(8-비트 부호화-크기 표현)
$+21=$	0000000000010101	(16-비트 부호화-크기 표현)
$-21=$	10010101	(8-비트 부호화-크기 표현)
$-21=$	1000000000010101	(16-비트 부호화-크기 표현)

비트 확장 (계속)

- 2의 보수 표현의 경우
  - 확장되는 상위 비트들을 부호 비트와 같은 값으로 세트
  - = 부호 비트 확장(sign-bit extension)

예제 3-8

10진수 '21'과 '−21'에 대한 8−비트 길이의 2의 보수 표현을 16−비트 길이로 확장하라.

풀이

$+21 = 00010101$ (8-비트 2의 보수)

$+21 = 0000000000010101$ (16-비트 2의 보수)

$-21 = 11101011$ (8-비트 2의 보수)

$-21 = 1111111111101011$ (16-비트 2의 보수)

3.3 논리 연산

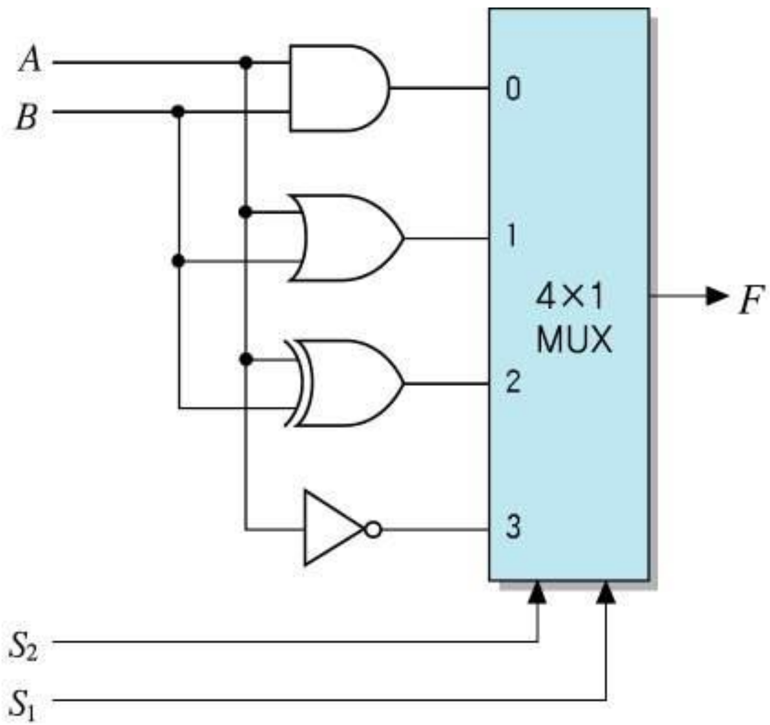
- 수를 나타내는 데이터: 단어 단위로 취급
- 논리 데이터: 각 비트 단위로 취급

- 기본적인 논리 연산들

A	*B*	NOT *A*	NOT *B*	*A* AND *B*	*A* OR *B*	*A* XOR *B*
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	1	0

논리 연산을 위한 하드웨어 모듈

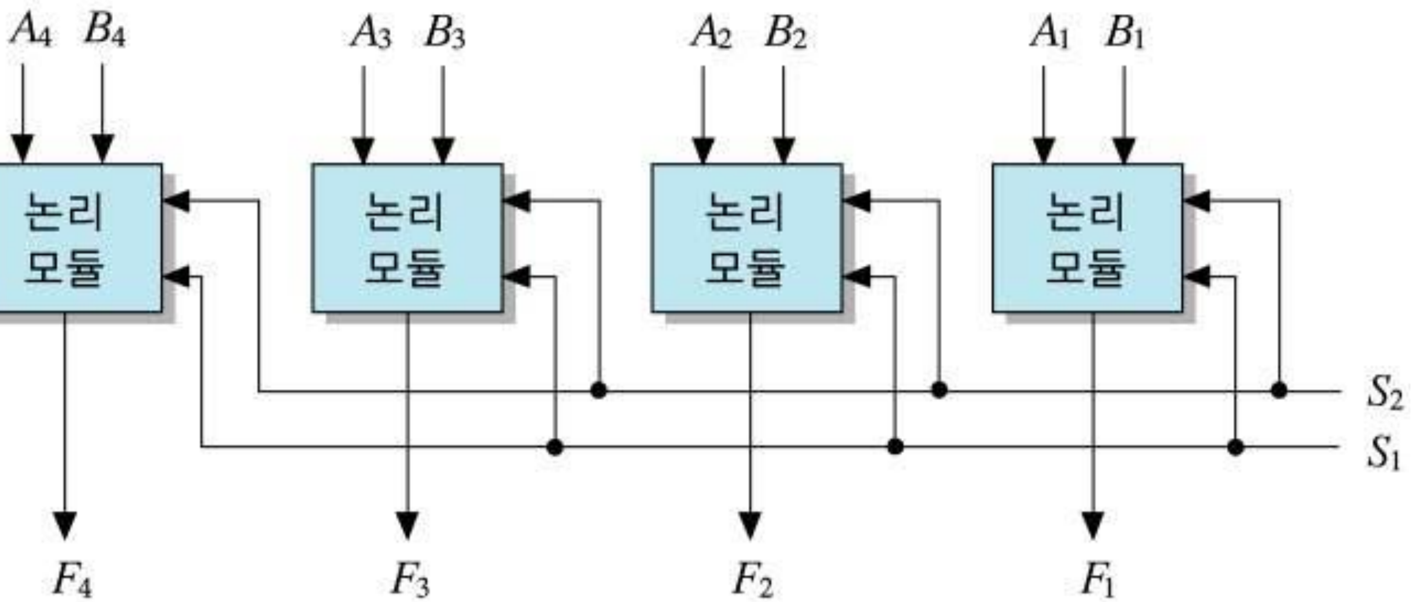
- 하드웨어의 구성
  - 입력 비트들은 모든 논리 게이트들을 통과
  - 선택 신호(S1,S2)에 의하여 멀티플렉서의 네 입력들 중의 하나를 출력
  - 멀티플렉서(Multiplexer: MUX): 여러 개 입력→1개 출력



S_2	S_1	출력	연산
0	0	$F = A \wedge B$	AND
0	1	$F = A \vee B$	OR
1	0	$F = A \oplus B$	XOR
1	1	$F = \overline{A}$	NOT

n-비트 논리 연산장치

- n-비트 데이터들을 위한 논리 연산장치
  - 기본 논리 모듈들을 병렬로 n개 접속
- 4-비트 논리 연산장치



- 논리 연산의 활용
  - 레지스터에 저장된 데이터 단어의 특정 비트 값을 변경
  - 일부 비트들을 삽입

AND 연산 / OR 연산

- AND 연산
  - 두 데이터 단어들의 대응되는 비트들 간에 AND 연산을 수행

```
    A  =  1 0 1 1   0 1 0 1
    B  =  0 0 1 1   1 0 1 1
  -------------------
          0 0 1 1   0 0 0 1    (연산 결과)
```

- OR 연산
  - 두 데이터 단어들의 대응되는 비트들 간에 OR 연산을 수행

```
    A  =  1 0 0 1   0 1 0 1
    B  =  0 0 1 1   1 0 1 1
  -------------------
          1 0 1 1   1 1 1 1   (연산 결과)
```

XOR 연산 / NOT 연산

- XOR 연산
  – 두 데이터 단어들의 대응되는 비트들 간에 exclusive-OR 연산을 수행

```
    A  =  1 0 0 1   0 1 0 1
    B  =  0 0 1 1   1 0 1 1
  -------------------
          1 0 1 0   1 1 1 0    (연산 결과)
```

- NOT 연산
  – 데이터 단어의 모든 비트들을 반전(invert)

```
    A  =  1 0 0 1   0 1 0 1    (연산 전)
  -------------------------------
          0 1 1 0   1 0 1 0    (연산 후)
```

데이터를 변경하기 위한 논리 연산

- 선택적-세트(selective-set) 연산
- 선택적-보수(selective-complement) 연산
- 마스크(mask) 연산
- 삽입(insert) 연산
- 비교(compare) 연산

선택적-세트 연산 / 선택적-보수 연산

- 선택적-세트(selective-set) 연산
  - B 레지스터의 비트들 중에서 1로 세트된 비트들과 같은 위치에 있는 A 레지스터의 비트들을 1로 세트 <OR 연산 이용>

```
A  =  1 0 0 1   0 0 1 0    (연산 전)
B  =  0 0 0 0   1 1 1 1
--------------------
A  =  1 0 0 1   1 1 1 1    (연산 후)
```

- 선택적-보수(selective-complement) 연산
  - B 레지스터의 비트들 중에서 1로 세트된 비트들에 대응되는 A 레지스터의 비트들을 보수로 변환 <XOR 연산 이용>

```
A  =  1 0 0 1   0 1 0 1    (연산 전)
B  =  0 0 0 0   1 1 1 1
--------------------
A  =  1 0 0 1   1 0 1 0    (연산 후)
```

마스크 연산

- 마스크(mask) 연산
  - B 레지스터의 비트들 중에서 값이 0인 비트들과 같은 위치에 있는 A 레지스터의 비트들을 0으로 바꾸는(clear) 연산 <AND 연산 이용>
  - 용도 : 단어내의 원하는 비트들을 선택적으로 clear하는 데 사용
  - [예]

```
  A  =  1 1 0 1   0 1 0 1    (연산 전)
  B  =  0 0 0 0   1 1 1 1
-------------------
  A  =  0 0 0 0   0 1 0 1    (연산 후)
```

삽입 연산

- 삽입(insert) 연산
  - 새로운 비트 값들을 데이터 단어내의 특정 위치에 삽입
  - 방법 : ① 삽입할 비트 위치들에 대하여 마스크(AND) 연산 수행
    ② 새로이 삽입할 비트들과 OR 연산을 수행

```
 A  =  1 0 0 1   0 1 0 1
 B  =  0 0 0 0   1 1 1 1    마스크 (AND 연산)
-------------------
 A  =  0 0 0 0   0 1 0 1    첫 단계 결과
 B  =  1 1 1 0   0 0 0 0    삽입 (OR 연산)
-------------------
 A  =  1 1 1 0   0 1 0 1    최종(삽입) 결과
```

비교 연산

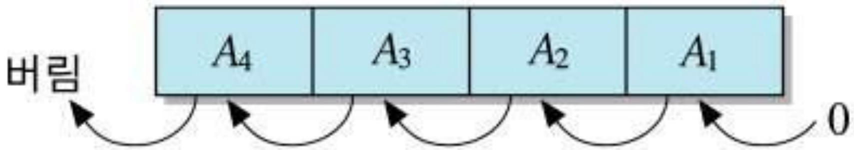
- 비교(compare) 연산
  - A와 B 레지스터의 내용을 비교 → XOR 연산
    - 같으면 0, 다르면 1로 세트
    - 결과는 A레지스터에 저장
    - 모든 비트들이 같으면→모든 비트가 0 → 상태레지스터 Z 플래그: 1 로 세트

```
  A  =  1 1 0 1   0 1 0 1
  B  =  1 0 0 1   0 1 1 0
-------------------
  A  =  0 1 0 0   0 0 1 1    (연산 결과)
```

3.4 쉬프트(shift) 연산

- 논리적 쉬프트 (logical shift)
  - 레지스터내의 데이터 비트들을 왼쪽 혹은 오른쪽으로 한 칸씩 이동
  - 좌측 쉬프트(left shift)
    - 모든 비트들을 좌측으로 한 칸씩 이동
    - 최하위 비트(A1)로는 0 이 들어오고, 최상위 비트(A4)는 버림

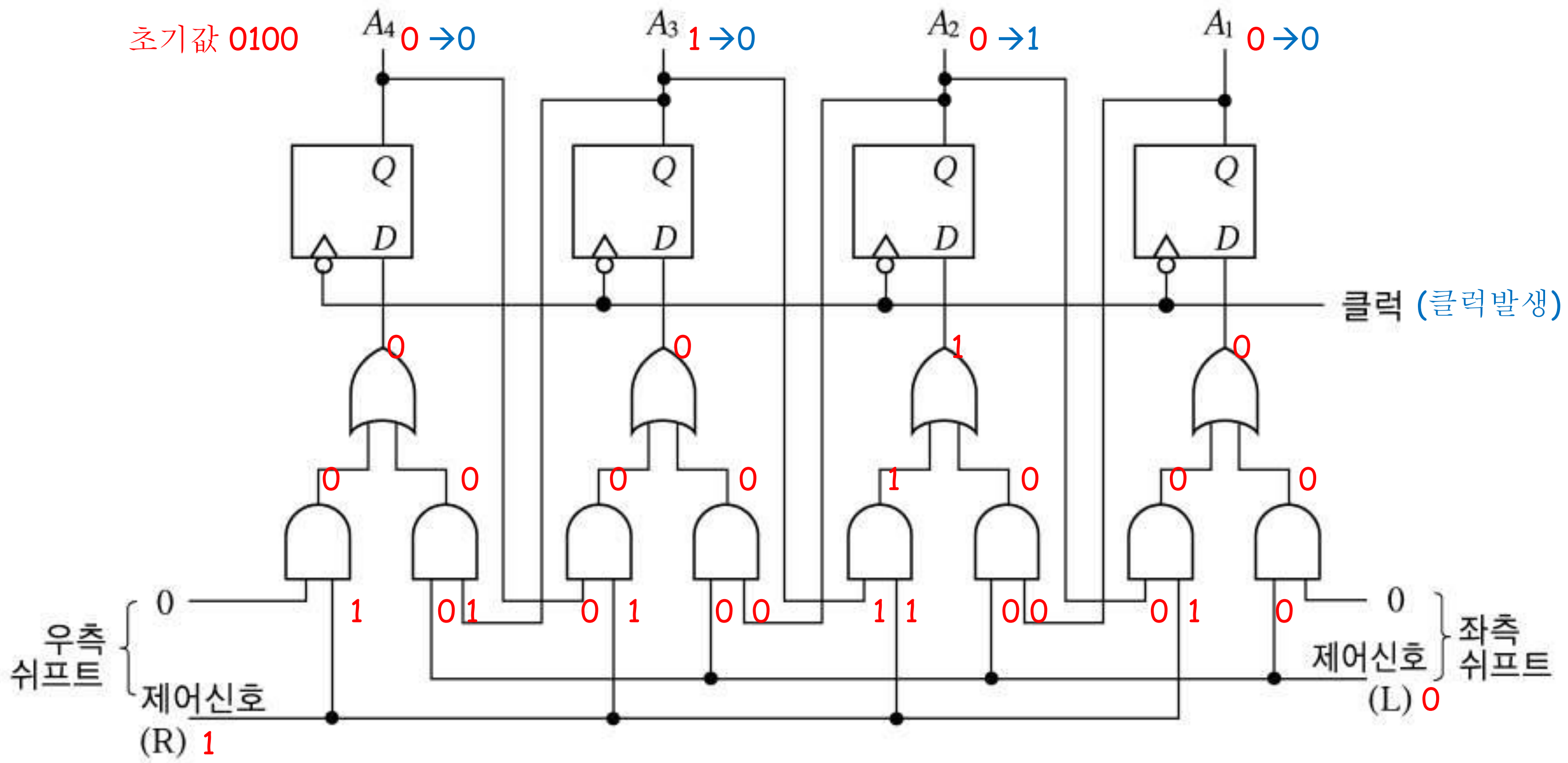
$(A_4 \leftarrow A_3,\ A_3 \leftarrow A_2,\ A_2 \leftarrow A_1,\ A_1 \leftarrow A_0)$



  - 우측 쉬프트(right shift)
    - 모든 비트들이 우측으로 한 칸씩 이동
    - 최상위 비트(A4)로 0이 들어오고, 최하위 비트(A1)는 버림
  - 논리적 쉬프트의 사용 예
    - 좌측 쉬프트: x2, 0100 → 1000
    - 우측 쉬프트: /2, 0100 → 0010

쉬프트 레지스터 (shift register)

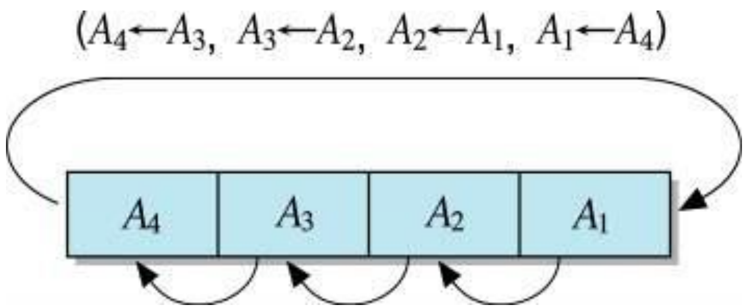
- D-플립플롭을 이용한 쉬프트 연산 기능 레지스터



  - D 플립플롭: 클럭이 발생하는 경우, D→Q (입력이 출력으로 전달)
  - 동작: 우측쉬프트 동작 (제어신호 R←1, L←0)

순환 쉬프트(circular shift)

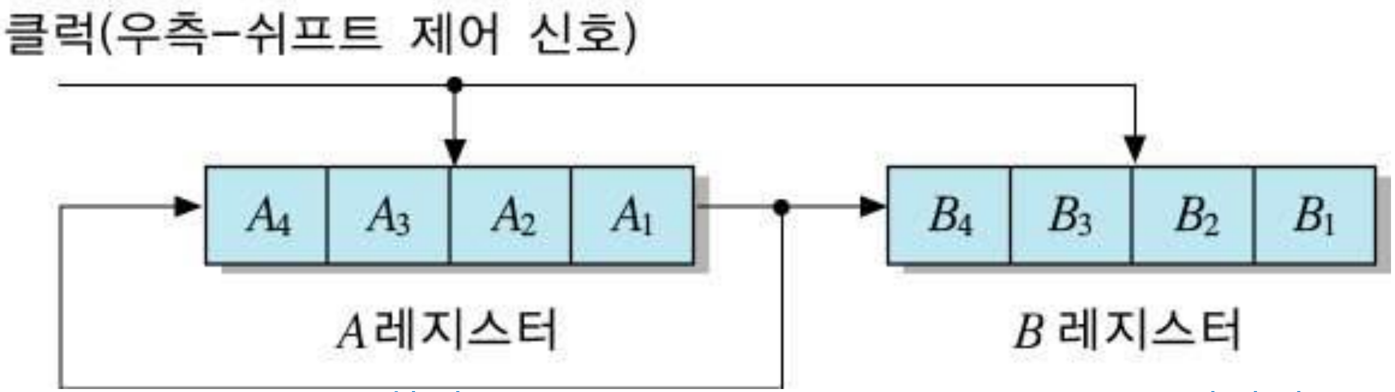
- 순환 쉬프트(circular shift)
  - 회전(rotate)이라고도 부르며, 최상위 혹은 최하위에 있는 비트를 버리지 않고 반대편 끝에 있는 비트 위치로 이동
  - 순환 좌측-쉬프트(circular shift-left)
    - 최상위 비트인 A4가 최하위 비트 위치인 A1으로 이동



  - 순환 우측-쉬프트(circular shift-right)
    - A4 → A3, A3 → A2, A2 → A1, A1 → A4

직렬 데이터 전송 (serial data transfer)

- 직렬 데이터 전송
  - 쉬프트 연산을 데이터 비트 수만큼 연속적으로 수행함으로써
  - 두 레지스터들 사이에 한 개의 선을 통하여 전체 데이터를 이동하는 동작



순환쉬프트 논리적쉬프트

	A_4	A_3	A_2	A_1	B_4	B_3	B_2	B_1
초기상태	1	0	1	1	0	0	0	0
t_1	1	1	0	1	1	0	0	0
t_2	1	1	1	0	1	1	0	0
t_3	0	1	1	1	0	1	1	0
t_4	1	0	1	1	1	0	1	1

산술적 쉬프트 (arithmetic shift)

- 수를 나타내는 데이터(부호를 가진 정수)에 대한 쉬프트
- 방법
  - 부호 비트: 유지
  - 수의 크기: 크기 비트들만 쉬프트
  - (1) 산술적 좌측-쉬프트(arithmetic shift-left)
    - A4 (불변), A3 ← A2, A2 ← A1, A1 ← 0
  - (2) 산술적 우측-쉬프트(arithmetic shift-right)
    - A4 (불변), A4 → A3, A3 → A2, A2 → A1

[예] A = 1 1 1 0 (-2) ; 초기 상태
1 1 0 0 (-4) ; 산술적 좌측-쉬프트 결과
1 1 1 0 (-2) ; 산술적 우측-쉬프트 결과
1 1 1 1 (-1) ; 산술적 우측-쉬프트 결과

3.5 정수의 산술 연산

- 기본적인 산술 연산들 (마이크로-연산)

$$
\begin{array}{llll}
A & \leftarrow & \overline{A} + 1 & \text{; 보수화(2의 보수 변환)} \\
A & \leftarrow & A + B & \text{; 덧셈} \\
A & \leftarrow & A - B & \text{; 뺄셈} \\
A & \leftarrow & A \times B & \text{; 곱셈} \\
A & \leftarrow & A \div B & \text{; 나눗셈} \\
A & \leftarrow & A + 1 & \text{; 증가(increment)} \\
A & \leftarrow & A - 1 & \text{; 감소(decrement)}
\end{array}
$$

3.5.1 덧셈

- 2의 보수로 표현된 수들의 덧셈 방법
  - 두 수를 더하고, 만약 올림수가 발생하면 버림

(a) $(+3) + (+4) = +7$

$$\begin{array}{r} 0011 \\ +\,0100 \\ \hline 0111 = +7 \end{array}$$

(b) $(-3) + (+3) = 0$

$$\begin{array}{rr} & 1101 \\ & +\,0011 \\ \hline \text{버림}\ 1 & 0000 = 0 \end{array}$$

(c) $(-6) + (+2) = -4$

$$\begin{array}{r} 1010 \\ +\,0010 \\ \hline 1100 = -4 \end{array}$$

(d) $(-4) + (-1) = -5$

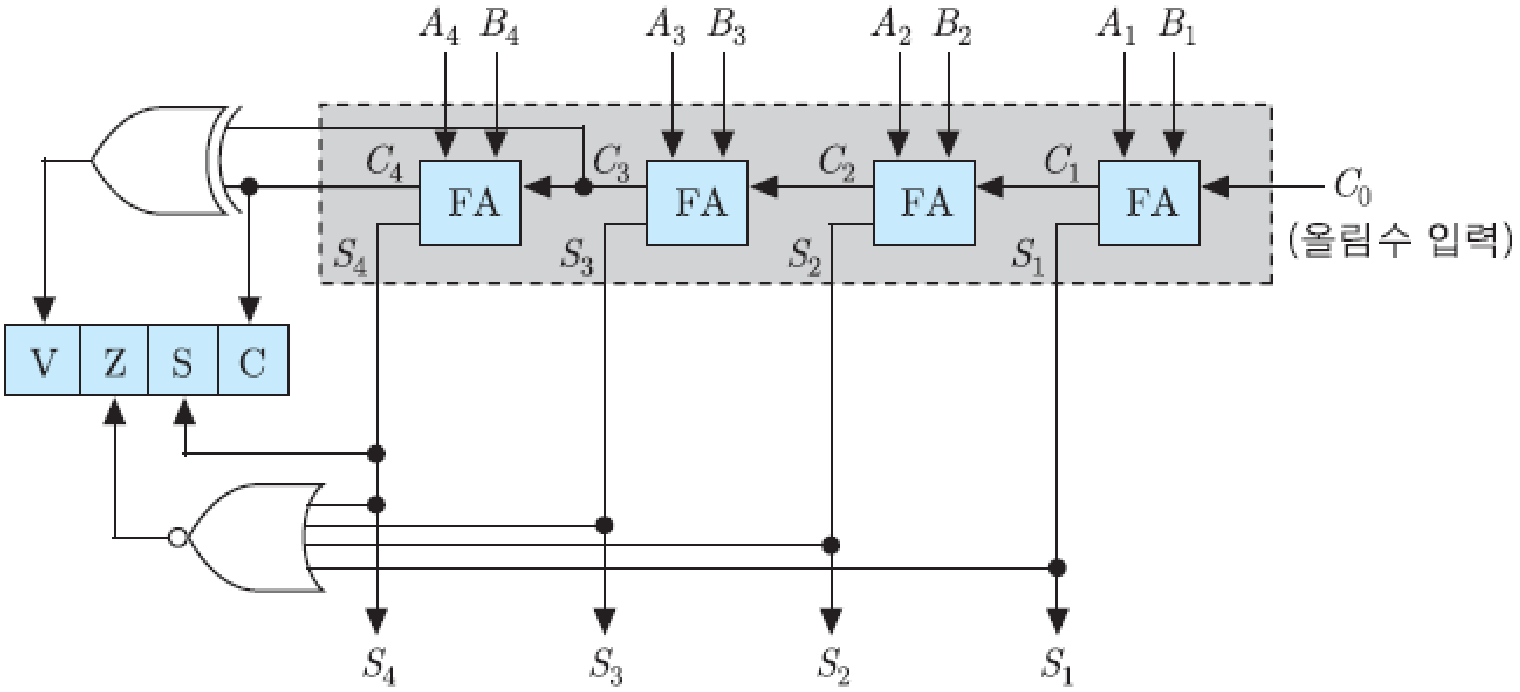
$$\begin{array}{rr} & 1100 \\ & +\,1111 \\ \hline 1 & 1011 = -5 \end{array}$$

병렬 가산기(parallel adder)

- 덧셈을 수행하는 하드웨어 모듈
- 비트 수만큼의 전가산기(full-adder)들을 연결하여 구성
- 덧셈 연산 결과에 따라 해당 조건 플래그들(condition flags)을 세트
  - C 플래그 : 올림수(carry)
  - S 플래그 : 부호(sign)
  - Z 플래그 : 0(zero)
  - V 플래그 : 오버플로우(overflow)

4-비트 병렬 가산기와 상태 비트 제어회로

- (a4 a3 a2 a1)+(b4 b3 b2 b1)→(S4 S3 S2 S1)



덧셈 오버플로우

- 덧셈 결과가 그 범위를 초과하여 결과값이 틀리게 되는 상태
- 검출 방법
  - 두 올림수(carry)들 간의 exclusive-OR를 이용

$V = C_4 \oplus C_3$

  - V가 1 → CPU는 결과값을 다른 연산에 사용하지 않도록 조치
- 덧셈에서 오버플로우가 발생하는 예

(a) (+6) + (+3) = +9

$$\begin{array}{r} 0110 \\ +\,0011 \\ \hline 1001 \end{array} = -7 \text{ (오버플로우)}$$

(b) (−7) + (−6) = −13

$$\begin{array}{r} 1001 \\ +\,1010 \\ \hline 1\ \ 0011 \end{array} = +3 \text{ (오버플로우)}$$

3.5.2 뺄셈

- 덧셈을 이용하여 수행

  A - (+B) = A + (-B)

  A - (-B) = A + (+B)

  단, A : 피감수(minuend), B : 감수(subtrahend)

(a) $(+2) - (+6) = (+2) + (-6) = -4$

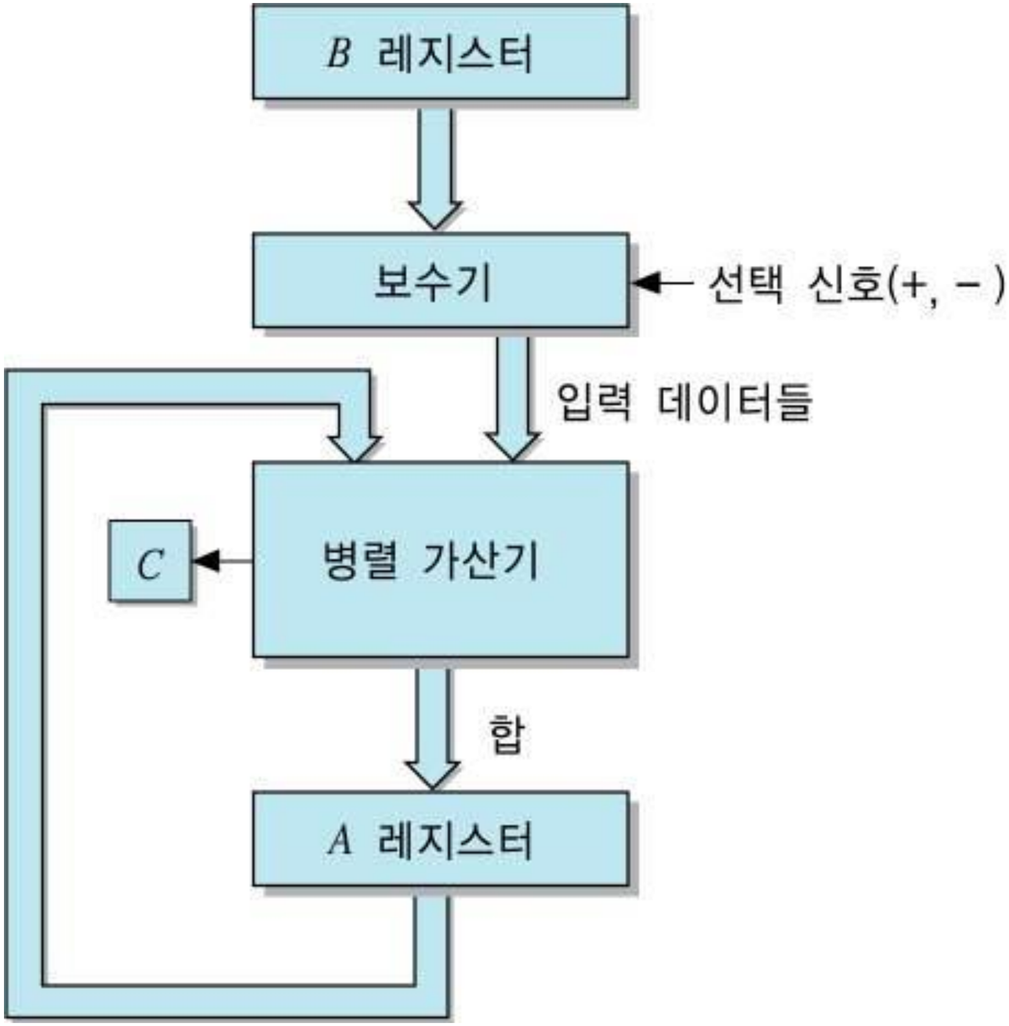
$$
\begin{array}{r}
0010 \\
+\ 1010 \\
\hline
1100 = -4
\end{array}
$$

(b) $(+5) - (+2) = (+5) + (-2) = +3$

$$
\begin{array}{r}
0101 \\
+\ 1110 \\
\hline
1\ 0011 = +3
\end{array}
$$

덧셈과 뺄셈 겸용 하드웨어의 블록 구성도

- 가산기+보수기



뺄셈 오버플로우

- 뺄셈 결과가 그 범위를 초과하여 결과값이 틀리게 되는 상태
- 검출 방법 : 덧셈과 동일 ($V = C_4 \oplus C_3$)

(a) (+7) − (−5) = (+7) + (+5) = +12

$$\begin{array}{r} 0111 \\ +\ 0101 \\ \hline 1100 \end{array} = -4 \ \text{(오버플로우)}$$

(b) (−6) − (+4) = (−6) + (−4) = −10

$$\begin{array}{r} 1010 \\ +\ 1100 \\ \hline 1\ \ 0110 \end{array} = +6 \ \text{(오버플로우)}$$

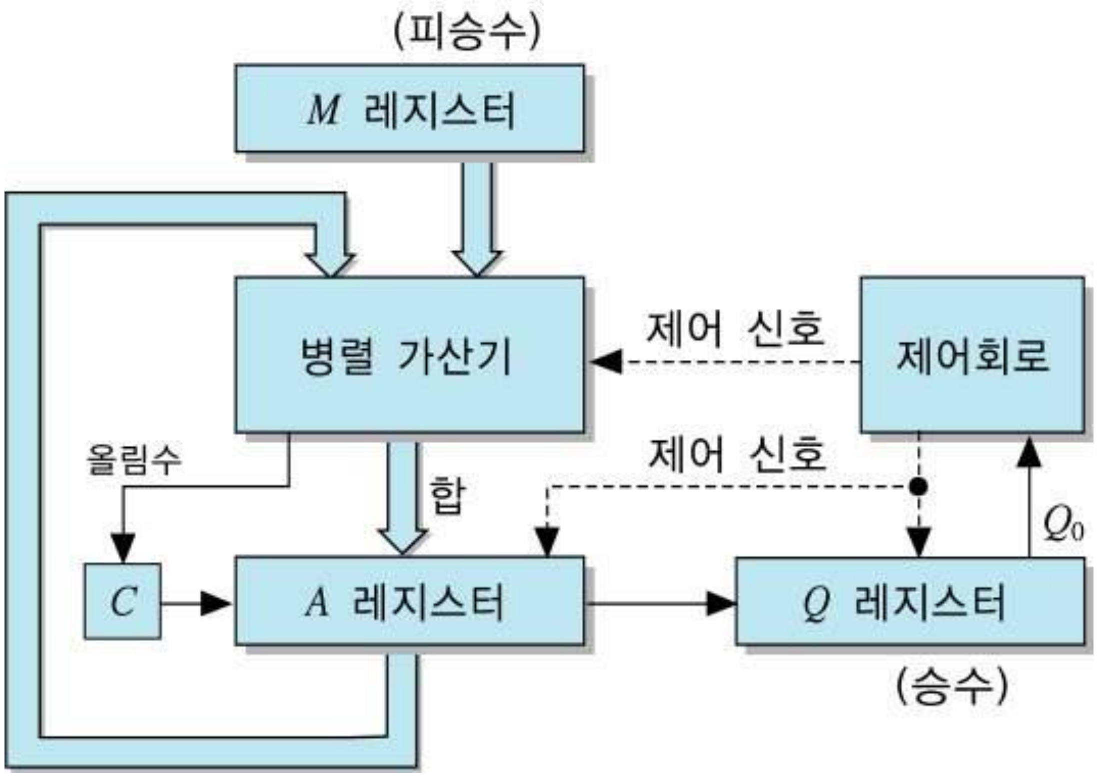
부호 없는 정수의 곱셈

- 부호 없는 정수의 곱셈
  - 각 비트에 대하여 부분 적(partial product) 계산
  - 부분적들을 모두 더하여 최종 결과를 얻음
  - 2개의 n비트 2진 정수의 곱셈 → 최대 2n 비트 결과

1101	(피승수=13)
× 1011	(승수=11)
1101	부분 적들(partial products)
1101	
0000	
1101	
10001111	(최종 결과=143)

부호 없는 정수 승산기의 하드웨어 구성도

- M 레지스터 : 피승수(multiplicand) 저장
- Q 레지스터 : 승수(multiplier) 저장
- 두 배 길이의 결과값은 A 레지스터와 Q 레지스터에 저장



곱셈이 수행되는 과정에서의 레지스터 내용들

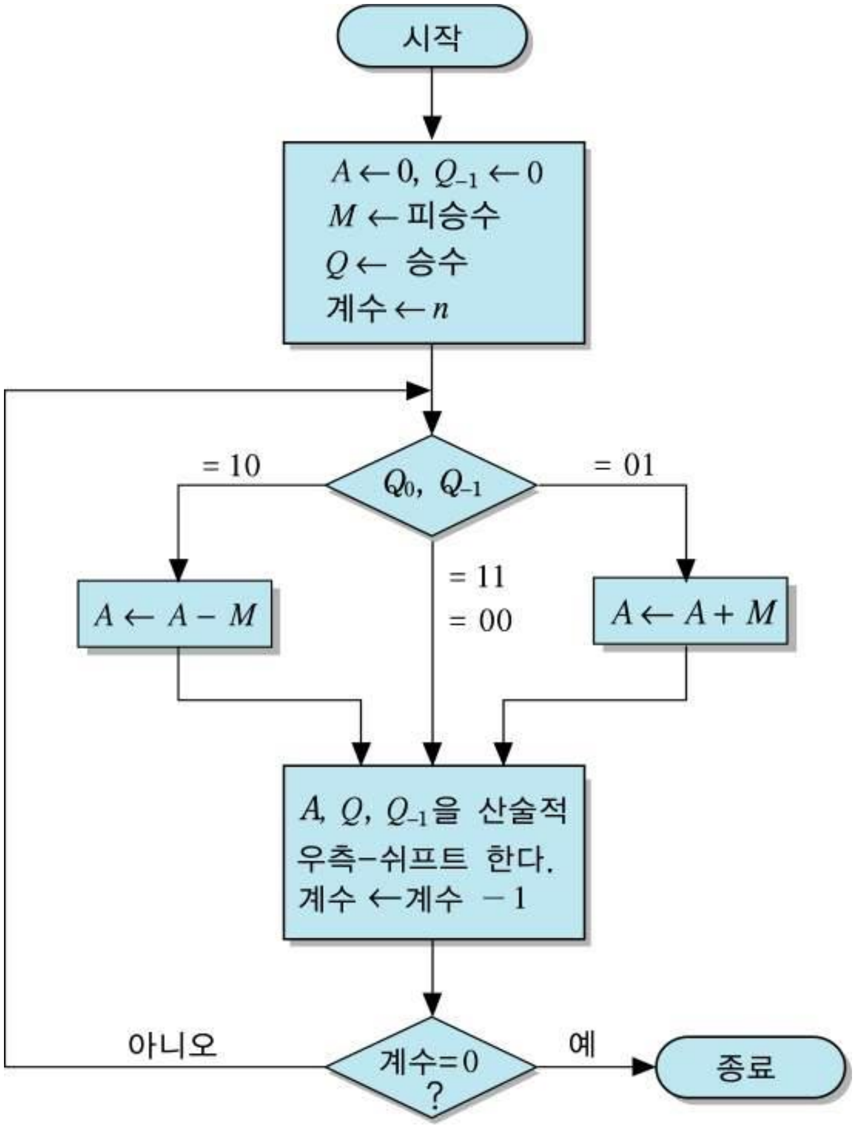
- 1011x1101
- M=1011, Q=1101

	C	A	Q	
[초기 상태]	0	0000	1101	
[사이클 1]	0	1011	1101	; $Q_0 = 1$이므로, $A \leftarrow A+M$.
	0	0101	1110	; 우측 쉬프트(C-A-Q)
[사이클 2]	0	0010	1111	; $Q_0 = 0$이므로, 쉬프트(C-A-Q)만 한다.
[사이클 3]	0	1101	1111	; $Q_0 = 1$이므로, $A \leftarrow A+M$.
	0	0110	1111	; 우측 쉬프트(C-A-Q)
[사이클 4]	1	0001	1111	; $Q_0 = 1$이므로, $A \leftarrow A+M$.
	0	1000	1111	; 우측 쉬프트(C-A-Q)

2의 보수들 간의 곱셈

- Booth 알고리즘(Booth's algorithm) 사용
- 하드웨어 구성
  - 부호 없는 정수 승산기의 하드웨어에 다음 부분을 추가
    - M 레지스터와 병렬 가산기 사이에 보수기(complementer) 추가
    - Q 레지스터의 우측에 Q_{-1} 이라고 부르는 1-비트 레지스터를 추가하고, 출력을 Q_0와 함께 제어 회로로 입력

Booth 알고리즘의 흐름도



Booth 알고리즘을 이용한 곱셈의 예 (-7x3)

(M)		1001			; 초깃값 : $A=0000$, $Q_{-1}=0$, 계수$(n)=4$
(Q)		$\times$ 0011	Q_{-1}	계수	
$(A\ Q)$	0111	0011	0		; $Q_0Q_{-1}=$'10'이므로, A로부터 피승수(1001)를 뺀다(실제로는 그 보수인 0111을 더한다).
	0011	1001	1	3	; A-Q-Q_{-1}에 대하여 산술적 우측-시프트를 수행하고, 계수에서 1을 뺀다.
	0001	1100	1	2	; $Q_0Q_{-1}=$'11'이므로, A-Q-Q_{-1}에 대하여 산술적 우측-시프트만 수행하고, 계수에서 1을 뺀다.
	1010	1100	1		; $Q_0Q_{-1}=$'01'이므로, A에 피승수(1001)를 더한다.
	1101	0110	0	1	; A-Q-Q_{-1}에 대하여 산술적 우측—시프트를 수행하고, 계수에서 1을 뺀다.
	1110	1011	0	0	; $Q_0Q_{-1}=$'00'이므로, A-Q-Q_{-1}에 대하여 산술적 우측—시프트를 수행한다. 계수에서 1을 빼면 0이므로, 계산을 종료한다.

1110 1011 → −21 (곱셈 결과)

3.5.4 나눗셈

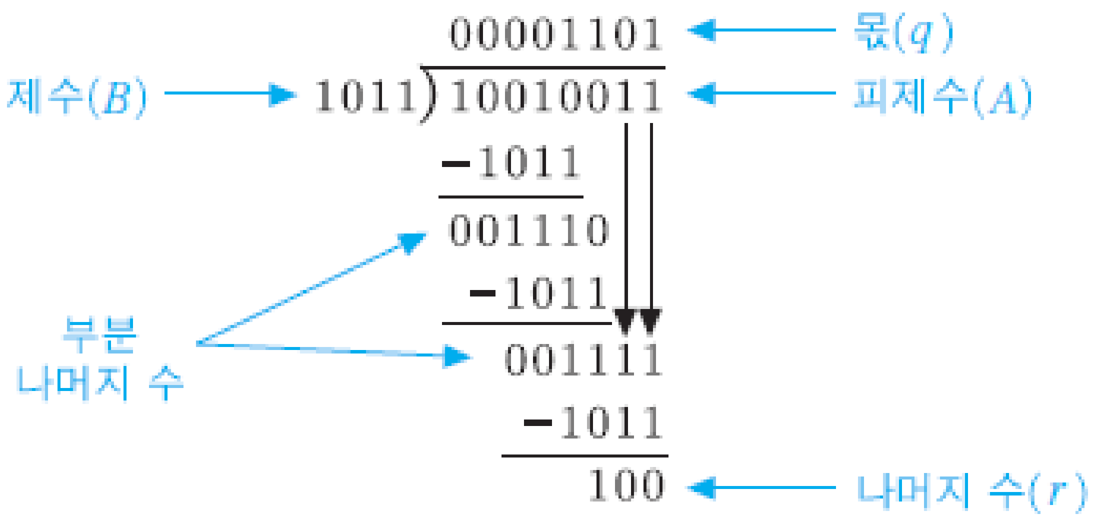
- 나눗셈의 수식 표현

  A ÷ B = q ····· r

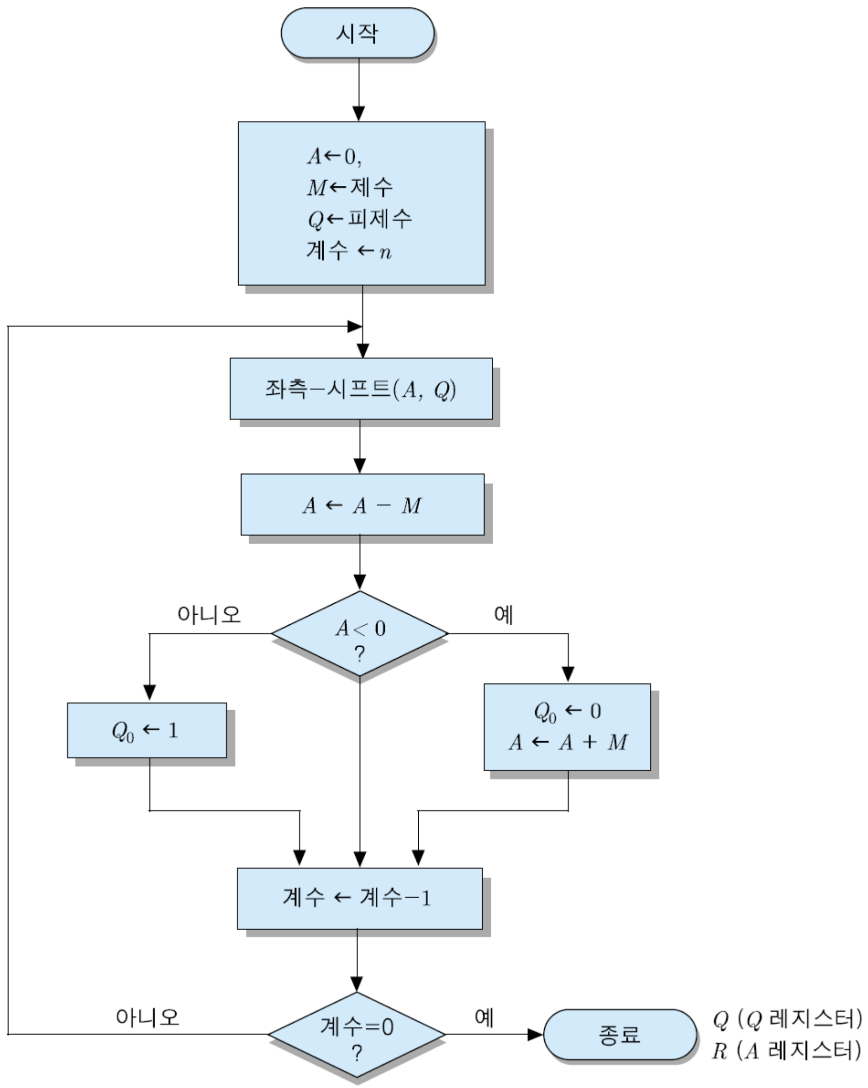
  단, A : 피제수(dividend), B : 제수(divisor)

  q : 몫(quotient) r : 나머지 수(remainder)

- 부호 없는 2진 나눗셈



부호 없는 2진 나눗셈 알고리즘의 흐름도



2의 보수 나눗셈 과정

- [초기 상태]
  - 젯수: M 레지스터, 피젯수: A와 Q 레지스터에 저장
  - 각 레지스터가 n 비트일 때, 피젯수는 2n 비트 길이의 2의 보수로 표시
- [사이클 1] A와 Q 레지스터를 좌측으로 한 비트씩 쉬프트
- [사이클 2] 만약 M과 A의 부호가 같으면 A ← A - M,
  다르면 A ← A + M을 수행한다.
- [사이클 3] 연산 전과 후의 A의 부호가 같으면 위의 연산은 성공
  - 연산이 성공이거나 A = 0 이면, Q0 ← 1로 세트
  - 연산이 실패이고 A ≠ 0 이면, Q0 ← 0으로 하고 A를 이전의 값으로 복구
- [사이클 4] Q에 비트 자리 수가 남아있다면, 단계 1에서 4까지를 반복
- [사이클 5]
  - 나머지 수는 A
  - 만약 젯수와 피젯수의 부호가 같으면 몫은 Q의 값
  - 부호가 다르면 Q값의 2의 보수가 몫

2의 보수 나눗셈의 예 (7 ÷ (-3))

A	Q	M=1101 (−3)
0000	0111	; 초기 상태
0000	1110	; 좌측−시프트($A-Q$)
1101		; A와 M의 부호가 서로 다르므로, $A \leftarrow A+M$
0000	1110	; A의 부호가 바뀌었으므로, A의 원래값을 복구
0001	1100	; 좌측−시프트($A-Q$)
1110		; A와 M의 부호가 서로 다르므로, $A \leftarrow A+M$
0001	1100	; A의 부호가 바뀌었으므로, $Q_0 \leftarrow 0$으로 세트하고 A의 원래값을 복구
0011	1000	; 좌측−시프트($A-Q$)
0000		; A와 M의 부호가 서로 다르므로, $A \leftarrow A+M$
0000	1001	; $A=0$이므로, $Q_0 \leftarrow 1$로 세트
0001	0010	; 좌측−시프트($A-Q$).
1110		; A와 M의 부호가 서로 다르므로, $A \leftarrow A+M$
0001	0010	; A의 부호가 바뀌었으므로, $Q_0 \leftarrow 0$으로 세트하고 A의 원래값을 복구

연산 결과 : 몫 = Q의 내용(0010)에 대한 2의 보수인 '1110' (−2)

나머지 = A 레지스터의 내용인 '0001' (+1)

3.6 부동소수점 수의 표현

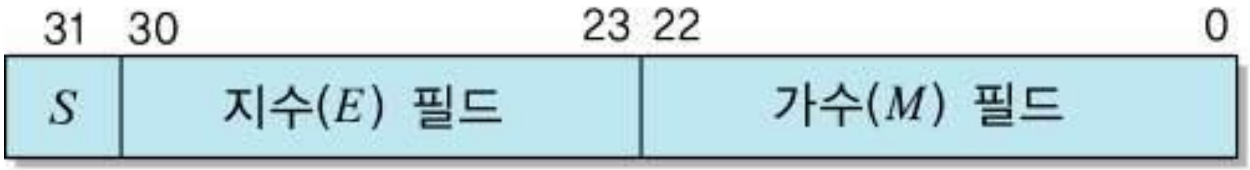
- 정수의 표현(3.2절)의 방법에서 2진 소수점이 있다고 가정
  - 소수 표현이 가능
- 소수 표현의 가능한 수의 범위에 한계
  - 아주 작은 수, 큰 수는 표현이 불가능
- 10진수의 과학적 표기
  - $274{,}000{,}000{,}000{,}000 = 2.74 \times 10^{14}$
  - $0.00000000000274 = 2.74 \times 10^{-12}$
  - 소수점의 위치를 적절히 이동
  - 소수점의 위치는 지수(exponent) 사용

3.6 부동소수점 수의 표현

- 부동소수점 표현(floating-point representation)
  - 소수점의 위치를 이동시킬 수 있는 표현 방법
  - 매우 큰 수, 작은 수의 표현 가능
- 부동소수점 수(floating-point number)의 일반적인 형태
  - $N = (-1)^S M \times B^E$
  - 여기서, S: 부호(sign), M: 가수(mantissa), B: 기수(base), E: 지수(exponent)
- 2진 부동소수점 수(binary floating-point number)
  - 기수 B = 2
  - 단일-정밀도(single-precision) 부동소수점 수 : 32 비트
  - 복수-정밀도(double-precision) 부동소수점 수 : 64 비트

단일-정밀도 부동소수점 수 형식의 예

- S: 1 비트,　　E: 8 비트,　　M: 23 비트
- 지수(E) 필드의 비트 수가 늘어나면 → 표현 가능한 수의 범위 확장
- 가수(M) 필드의 비트 수가 늘어나면 → 정밀도(precision) 증가
- 제한된 비트
  - 가수와 지수의 trade-off 존재



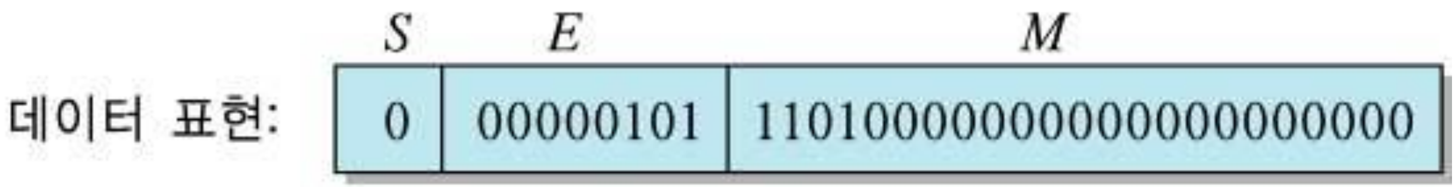
같은 수에 대한 부동소수점 표현

- 같은 수에 대한 부동소수점 표현이 여러 가지가 존재
  - **0.1101×2^5**
  - **110.1×2^2**
  - **0.01101×2^6**
- 정규화된 표현(normalized representation)
  - 수에 대한 표현을 한 가지로 통일하기 위한 방법

    $\pm\ 0.1bbb...b \times 2^E$
  - 위의 예에서 정규화된 표현은 0.1101×2^5

비트 배열의 예 (0.1101 × 2^5)

- 부호(S) 비트 = 0
- 지수(E) = 00000101
- 가수(M) = 1101 0000 0000 0000 0000 000
- 소수점 아래 첫 번째 비트는 항상 1이므로, 저장할 필요가 없음
  → 가수 23비트를 이용하여 소수점 아래 24 자리 수까지 표현 가능



바이어스된 지수 (biased exponent)

- 부동소수점 표현의 0 표현 문제
  - 가수: 0
  - 지수
    - 어떠한 수라도 상관 없음
    - E의 값이 매우 큰 음수의 경우
      - $|2^E| = 0$ (근사)
  - 0 검사를 위해 0에 대한 부동소수점 표현: 가수와 지수가 모두=0
- 지수: 바이어스된 수(biased number)로 표현
  - 예) 바이어스 3인 경우: 1000 → 1011
- 해결 방법: 바이어스된 지수 사용
  - 가수: 모든 비트가 0
  - 지수(바이어스 127): 00000000 → $M \times 2^{-127}$
  - → 가수 M과 상관없이 0에 근접한 작은 수
  - 모든 비트= 0 이므로 0-검사(zero-test)가 용이함

  - 참고) 모든 비트가 0이면 실제 0을 의미???

국립 한경대학교 HANKYONG NATIONAL UNIV.

8-비트 바이어스된 지수값들

지수 비트 패턴	절대값	실제 지수값 바이어스=127	실제 지수값 바이어스=128
11111111	255	+128	+127
11111110	254	+127	+126
⋮	⋮	⋮	⋮
10000001	129	+2	+1
10000000	128	+1	0
01111111	127	0	−1
01111110	126	−1	−2
⋮	⋮	⋮	⋮
00000001	1	−126	−127
00000000	0	−127	−128

바이어스된 지수를 사용한 부동소수점 표현의 예

- 바이어스값 = 128일 때, N = - 13.625에 대한 부동소수점 표현

  $13.625_{10} = 1101.101_2 = 0.1101101 \times 2^4$

  부호(S) 비트 = 1 (-)

  지수(E) = 00000100 + 10000000 = 10000100

  (바이어스 128을 더한다)

  가수(M) = 10110100000000000000000

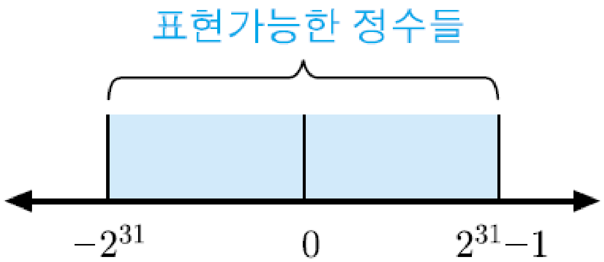
  (소수점 우측의 첫 번째 1은 제외)

S	*E*	*M*
1	10000100	10110100000000000000000

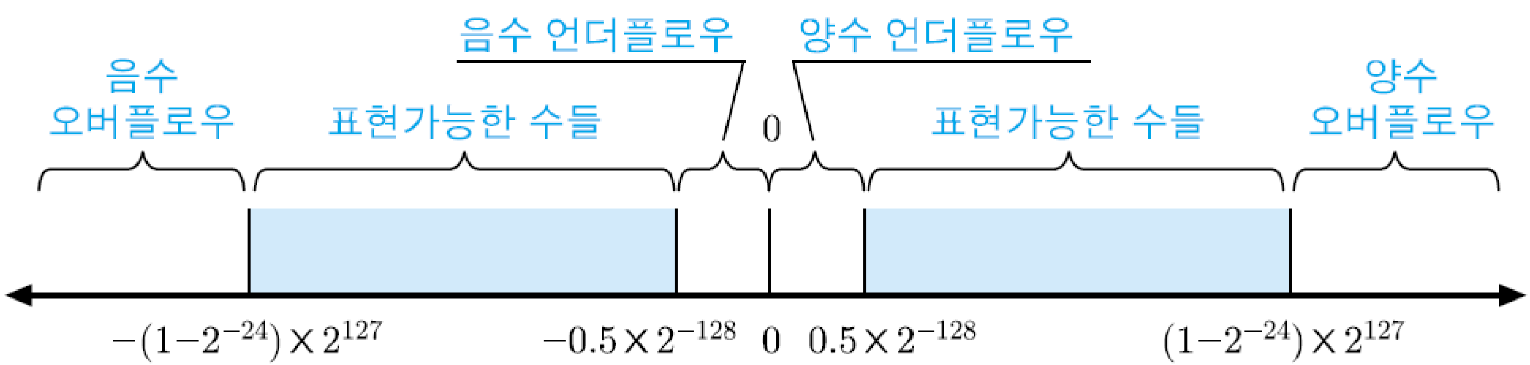
부동소수점 수의 표현 범위

- 부동소수점 수의 표현 범위
  - 양수: 0.5×2^{-128} ~ $(1 - 2^{-24}) \times 2^{127}$ // $0.100... \times 2^{-(0000...)}$ ~ $0.111... \times 2^{-(1111...)}$
    (대략 1.47×10^{-39} ~ 1.7×10^{38})
  - 음수: $-(1 - 2^{-24}) \times 2^{127}$ ~ -0.5×2^{-128}
- 제외되는 범위
  - $(1 - 2^{-24}) \times 2^{127}$보다 작은 음수 → 음수 오버플로우(negative overflow)
  - 0.5×2^{-128} 보다 큰 음수 → 음수 언더플로우(negative underflow)
  - 0
  - 0.5×2^{-128} 보다 작은 양수 → 양수 언더플로우(positive underflow)
  - $(1 - 2^{-24}) \times 2^{127}$ 보다 큰 양수→ 양수 오버플로우(positive overflow)

32-비트 데이터 형식의 표현 가능한 수의 범위



(a) 2의 보수 정수의 표현 범위

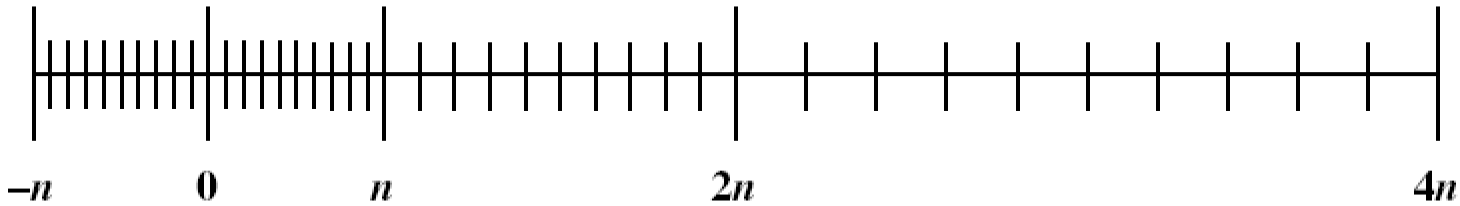


(b) 부동소수점 수의 표현 범위

국립 한경대학교 HANKYONG NATIONAL UNIV.

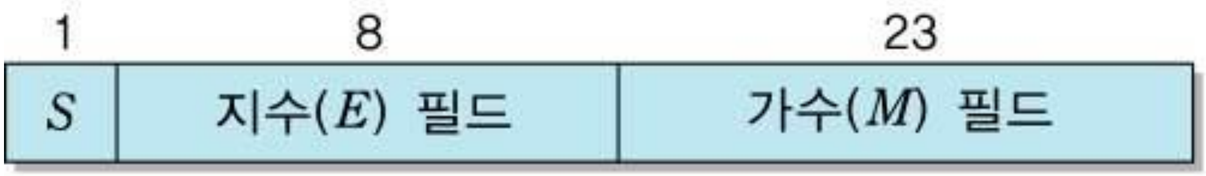
부동소수점 수의 밀도

- 부동-소수점을 사용하더라도, 표현 가능한 수의 개수가 늘어나지 않음
- 고정(fixed point)-소수점의 경우와 달리, 수의 선(number line) 상에 균등한 간격으로 나열되지 않음
- “0”점 근처에서 수들이 더 밀집, 멀리 떨어질수록 간격이 더 커짐



IEEE 754 표준 부동소수점 수의 형식

- 부동소수점 수의 표현 방식의 통일을 위하여 미국전기전자공학회(IEEE)에서 정의한 표준



(a) 단일-정밀도 형식(single-precision format)

(b) 복수-정밀도 형식(double-precision format)

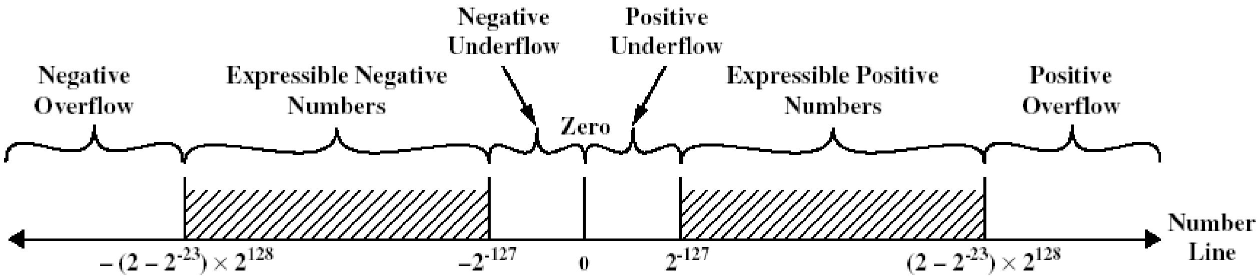
IEEE 754 표준 부동소수점 수의 형식

- 표현 방법

$$N = (-1)^S\ 2^{E-127}\ (1.M)$$

  – 부호+가수 : 부호화-크기 표현 사용
  – 지수 필드 : 바이어스 127 사용
  – $1.M \times 2^E$의 형태를 가지며, 소수점 아래의 M 부분만 가수 필드에 저장 (소수점 왼쪽의 표현되지 않는 1을 hidden bit라고 지칭)
  – 64-비트 복수-정밀도 부동소수점 형식을 사용하는 경우

$$N = (-1)^S\ 2^{E-1023}\ (1.M)$$



IEEE 754 표현 예 (N = - 13.625)

- $13.625_{10} = 1101.101_2 = 1.101101 \times 2^3$

  부호(S) 비트 = 1 (-)

  지수 E = 00000011 + 01111111 = 10000010

  (바이어스 127을 더한다)

  가수 M = 10110100000000000000000

  (소수점 좌측의 1은 제외한다)

S	*E*	*M*
1	10000010	10110100000000000000000

예외 경우를 포함한 IEEE 754 표준

- 예외 경우를 포함한 정의 (32-비트 형식)
  - 만약 $E = 255$이고 $M \neq 0$이면, $N = \text{NaN}$ (0으로 나누기, 음수의 루트 계산)
  - 만약 $E = 255$이고 $M = 0$이면, $N = (-1)^S \infty$ (오버플로우)
  - 만약 $0 < E < 255$ 이면, $N = (-1)^S 2^{E-127} (1.M)$
  - 만약 $E = 0$이고 $M \neq 0$이면, $N = (-1)^S 2^{-126} (0.M)$ (언더플로우)
  - 만약 $E = 0$이고 $M = 0$이면, $N = (-1)^S 0$ (0)
- 예외 경우를 포함한 정의 (64-비트 형식)
  - 만약 $E = 2047$이고 $M \neq 0$이면, $N = \text{NaN}$
  - 만약 $E = 2047$이고 $M = 0$이면, $N = (-1)^S \infty$
  - 만약 $0 < E < 2047$ 이면, $N = (-1)^S 2^{E-1023} (1.M)$
  - 만약 $E = 0$이고 $M \neq 0$이면, $N = (-1)^S 2^{-1022} (0.M)$
  - 만약 $E = 0$이고 $M = 0$이면, $N = (-1)^S 0$

부동소수점 덧셈 / 뺄셈

- 덧셈과 뺄셈
  – 지수들이 일치되도록 조정 (alignment)
  – 가수들 간의 연산(더하기 혹은 빼기) 수행
  – 결과를 정규화 (normalization)

- [10진 부동소수점 산술의 예]

$$(135 \times 10^{-5}) + (246 \times 10^{-3}) \rightarrow \begin{array}{rr} 1.35 & \times 10^{-3} \\ +\,246 & \times 10^{-3} \\ \hline 247.35 & \times 10^{-3} \end{array}$$

예제 3-29

부동소수점 수들 간의 덧셈 ($0.110100 \times 2^3 + 0.111100 \times 2^5$)을 수행하라.

풀이

이 덧셈은 아래와 같은 세 단계를 통하여 수행될 수 있다.

(1) 지수 조정

$$\begin{array}{r} 0.110100 \times 2^3 \\ +\ \ 0.111100 \times 2^5 \\ \hline \end{array} \rightarrow \begin{array}{r} 0.001101 \times 2^5 \\ +\ \ 0.111100 \times 2^5 \\ \hline 1.001001 \times 2^5 \end{array} \xrightarrow{\text{(3) 정규화}} 0.1001001 \times 2^6$$

(2) 더하기

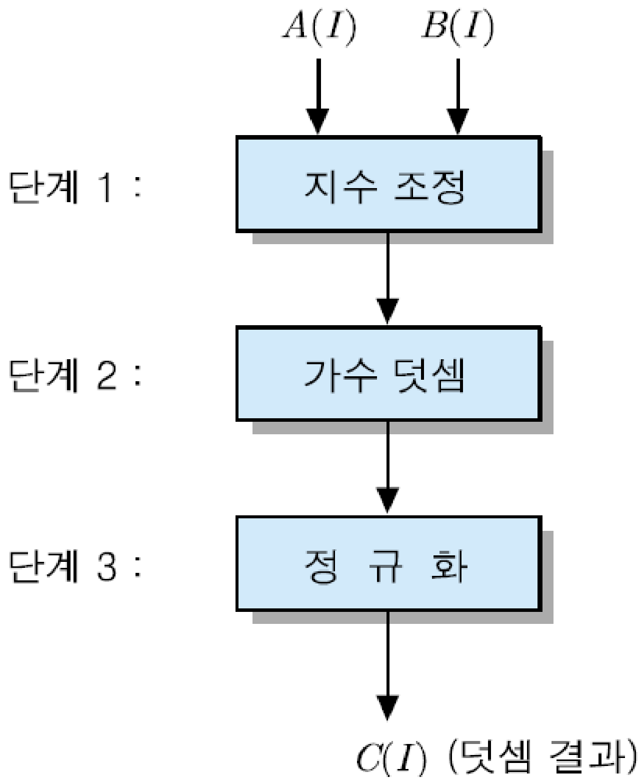
(최종 결과)

부동소수점 산술의 파이프라이닝

- 연산 과정을 독립적 단계들로 분리 가능
- 단계 수만큼의 속도 향상
- 대규모의 부동소수점 계산을 처리하는 거의 모든 슈퍼컴퓨터들에서 채택

[예] 수 배열(number array)들 간의 덧셈

$$C(I) = A(I) + B(I)$$



부동소수점 곱셈 / 나눗셈

- 2진수 부동소수점 곱셈 과정
  - 가수들을 곱한다
  - 지수들을 더한다
  - 결과값을 정규화
- 2진수 부동소수점 나눗셈 과정
  - 가수들을 나눈다
  - 피젯수의 지수에서 젯수의 지수를 뺀다
  - 결과값을 정규화

[부동소수점 곱셈의 예]

$(0.1011 \times 2^3) \times (0.1001 \times 2^5)$

<가수 곱하기>

$1011 \times 1001 = 01100011$

<지수 더하기>

$3 + 5 = 8$

<정규화>

0.01100011×2^8

$= 0.1100011 \times 2^7$ (결과값)

부동소수점 연산 과정에서 발생 가능한 문제들

- 지수 오버플로우(exponent overflow)
  - 양의 지수값이 최대 지수값을 초과
  - 수가 너무 커서 표현될 수 없는 상태이므로, +∞ 또는 -∞로 세트
- 지수 언더플로우(exponent underflow)
  - 음의 지수값이 최대 지수값을 초과
  - 수가 너무 작아서 표현될 수 없는 상태이므로, 0으로 세트
- 가수 언더플로우(mantissa underflow)
  - 가수의 소수점 위치 조정 과정에서 비트들이 가수의 우측 편으로 넘치는 경우

    ➔ 반올림(rounding) 적용
- 가수 오버플로우(mantissa overflow)
  - 같은 부호를 가진 두 가수들을 덧셈하였을 때, MSB에서 올림수가 발생하는 경우

    ➔ 재조정(realignment) 과정을 통하여 정규화

Assignment #1

- 문제 1
  - 그림 3-12 (p.171)을 수정하여 Booth 곱셈기를 설계하라.
  - 2의 보수로 표현된 -5x-3의 곱셈을 예제 3-25와 같이 과정을 전개하여 15가 결과로 나오는지 확인하라.
    - M레지스터: -5
    - Q레지스터: -3
- 문제 2
  - -1.625를 IEEE 754 표준의 32비트 부동소수점 형식으로 나타내어라.