컴퓨터 구조

3장 컴퓨터 산술과 논리 연산

안형태

anten@kumoh.ac.kr

디지털관 139호

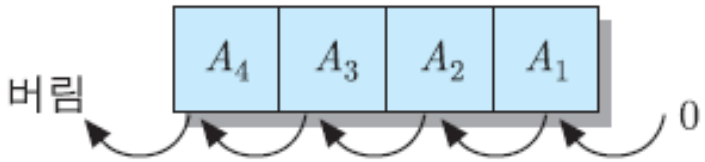
컴퓨터 산술과 논리 연산

4. 시프트 연산자

시프트(shift) 연산

- **논리적 시프트 (logical shift):** 레지스터 내의 데이터 비트들을 왼쪽 혹은 오른쪽으로 한 칸씩 이동
  - **좌측 시프트(Logical Shift Left, LSL)**
    - 모든 비트들을 좌측으로 한 칸씩 이동
    - 최하위 비트(A_1)로는 '0'이 들어오고, 최상위 비트(A_4)는 버림

$$A_4 \leftarrow A_3,\ A_3 \leftarrow A_2,\ A_2 \leftarrow A_1,\ A_1 \leftarrow 0$$

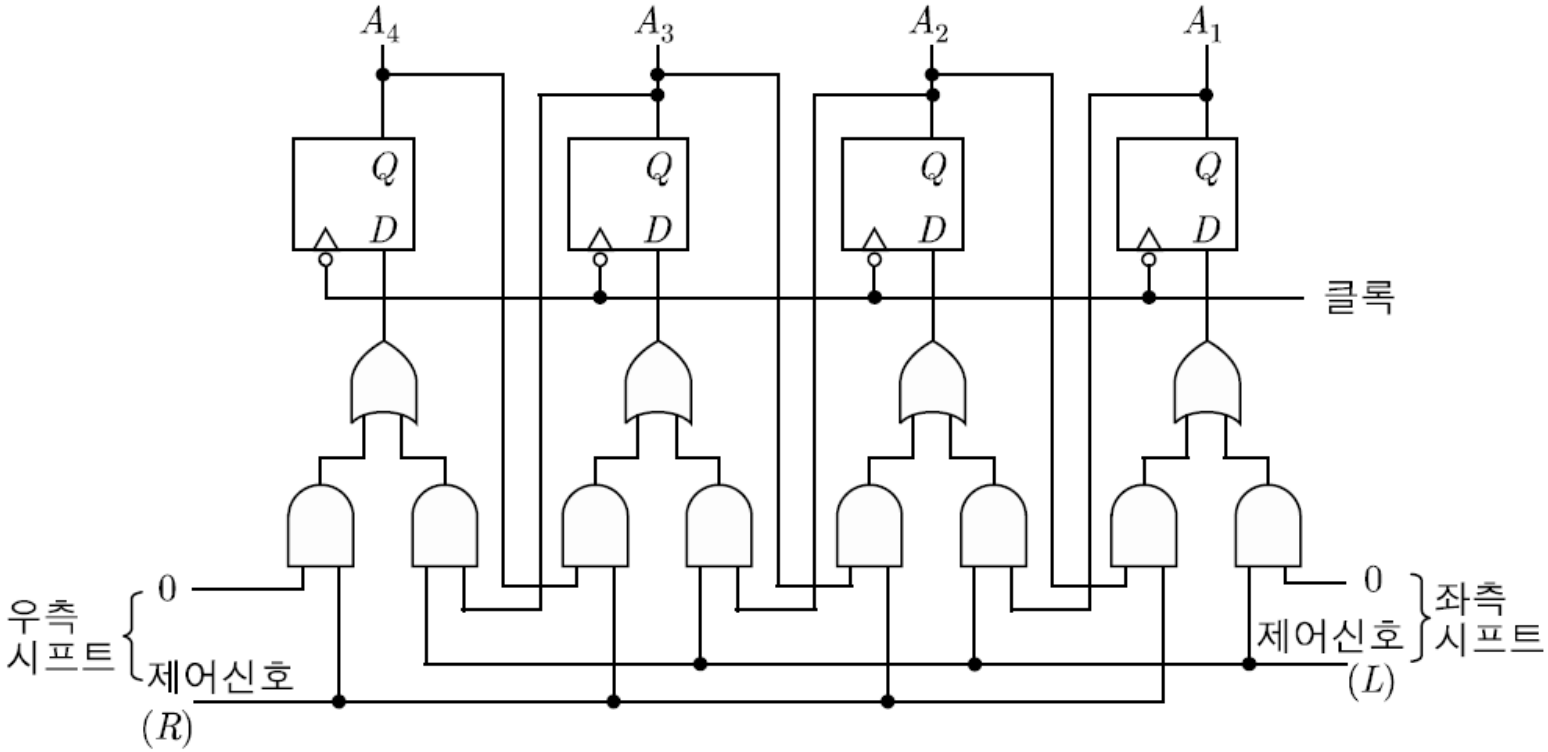


  - **우측 시프트 (Logical Shift Right, LSR)**
    - 모든 비트들이 우측으로 한 칸씩 이동
    - 최상위 비트(A_4)로 '0'이 들어오고, 최하위 비트(A_0)는 버림

[TMI] 시프트 레지스터(shift register)

시프트 기능을 가진 레지스터의 내부 회로

- **D 플립플롭(flip-flop):** 1bit의 정보를 저장하는 논리 소자(logic element)
  - 클록 펄스가 입력되면 기존의 저장 값이 Q로 출력되고 D의 입력이 D 플립플롭에 저장됨

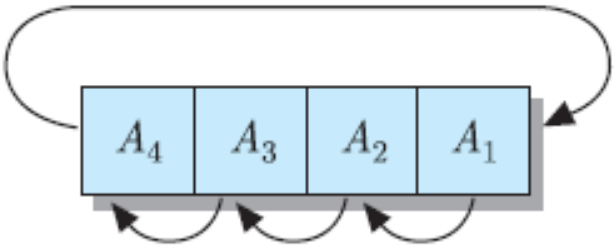


시프트(shift) 연산

순환 시프트(circular shift)

- 회전(rotate)이라고도 부르며, 최상위 혹은 최하위에 있는 비트를 버리지 않고 반대편 끝에 있는 비트 위치로 이동
- **순환 좌측-시프트(circular shift-left):** 최상위 비트인 A_4가 최하위 비트 위치인 A_1으로 이동

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



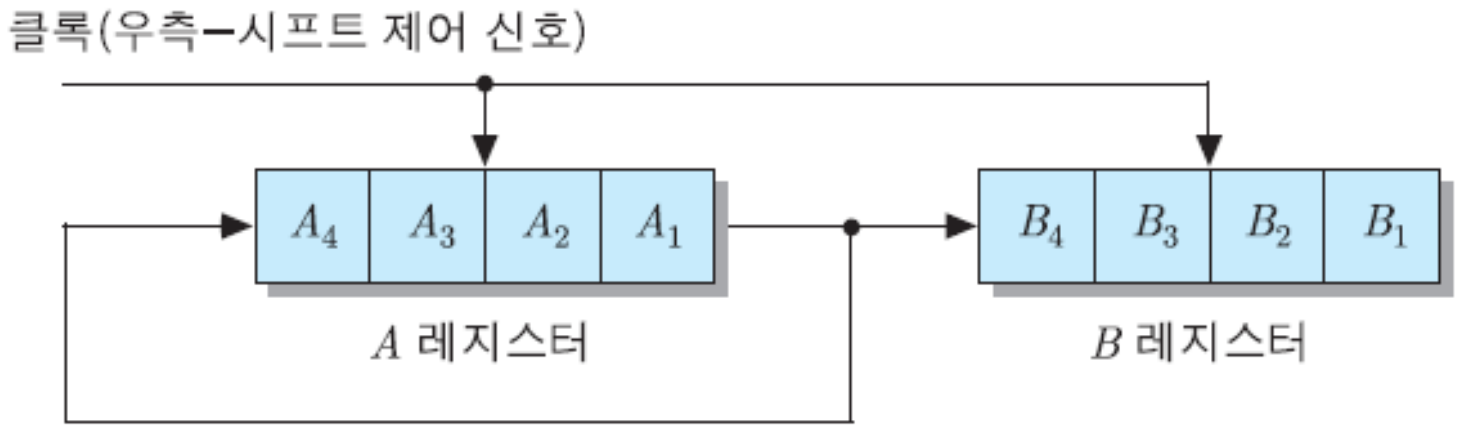
- **순환 우측-시프트(circular shift-right)**

$$A_4 \rightarrow A_3,\ A_3 \rightarrow A_2,\ A_2 \rightarrow A_1,\ A_1 \rightarrow A_4$$

직렬 데이터 전송

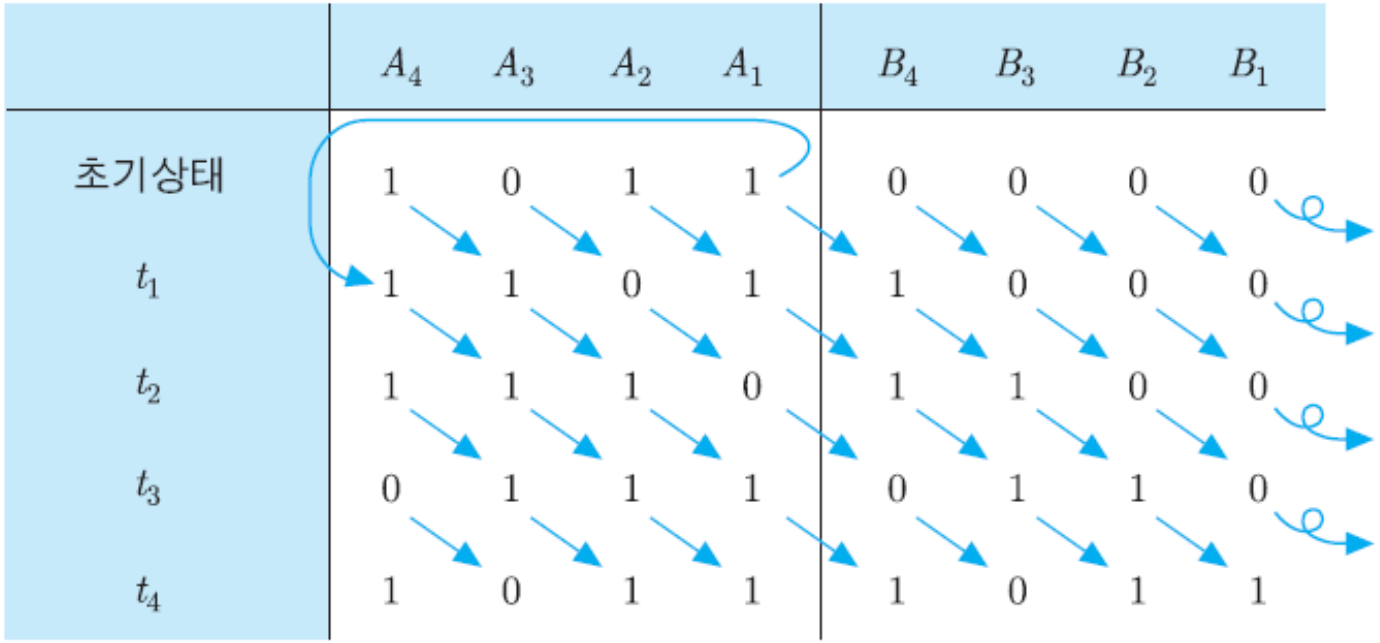
직렬 데이터 전송(serial data transfer)

- 시프트 연산을 데이터 비트 수만큼 연속적으로 수행함으로써 두 레지스터들 사이에 한 개의 선을 통하여 전체 데이터를 이동하는 동작
  - 순환 시프트 연산과 논리적 시프트 연산을 활용



직렬 데이터 전송

- **[예제]** 두 개의 4-비트 레지스터들이 직렬로 접속된 상태에서 A 레지스터에 저장되어 있는 데이터인 '1011'을 네 클록 주기 후에 모두 B 레지스터로 전송하고, A 레지스터는 원래의 데이터를 그대로 유지하게 되는 과정을 표시하라.(레지스터 B의 초기 값은 '0000'으로 가정)
- **[풀이]**



	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 시프트

- **산술적 시프트(arithmetic shift):** 수(number)를 나타내는 데이터(부호를 가진 데이터)에 대한 시프트 동작
  - 시프트 과정에서 부호 비트는 그대로 유지시키고, 수의 크기를 나타내는 비트들만 시프트
  - **산술적 좌측-시프트(arithmetic shift-left)**

$$A4\ (\text{불변}),\ A3 \leftarrow A2,\ A2 \leftarrow A1,\ A1 \leftarrow 0$$

  - **산술적 우측-시프트(arithmetic shift-right)**
    - 부호 비트가 그 우측의 비트로 복사: **부호-비트 확장**

$$A4\ (\text{불변}),\ A4 \rightarrow A3,\ A3 \rightarrow A2,\ A2 \rightarrow A1$$

산술적 시프트

- **[예제]** 10진수 -2에 대한 4-비트 2진수에 대하여 산술적 좌측-시프트를 수행한 후, 그 결과 값에 대하여 두번의 산술적 우측-시프트를 수행하라.

- **[풀이]**

A = 1 1 1 0 (-2) : 초기상태

1 1 0 0 (-4) : 산술적 좌측-시프트 결과

1 1 1 0 (-2) : 첫 번째 산술적 우측-시프트 결과

1 1 1 1 (-1) : 두 번째 산술적 우측-시프트 결과

시프트의 주요 용도

- 시프트 연산은 특정 산술 연산을 빠르고 효율적으로 수행하기 위해 사용됨
  - 특히, 2의 거듭제곱수의 곱셈이나 나눗셈을 매우 빠르게 수행할 수 있음
    - 양의 정수를 왼쪽으로 k 비트 시프트(LSL): 오버플로우가 없다면, 원래 수 × 2^k
    - 양의 정수를 오른쪽으로 k 비트 시프트(LSR): 데이터 손실이 없다면 원래 수 ÷ 2^k
  - **[예]** LSL은 2를 곱하는 효과(0010 → 0100), LSR은 2로 나누는 효과(0010 → 0001)가 발생
  - **[예]** 양의 정수 n에 대해 $24 \times n$을 계산
    - $24 \times n = (16 + 8) \times n = 2^4 \times n + 2^3 \times n$ 이므로,
    - n을 4비트 왼쪽으로 시프트하면 $16 \times n$이 되고,
    - n을 왼쪽으로 3비트 시프트하면 $8 \times n$
    - 두 값의 합이 $24 \times n$
    - 즉, 시프트 두 번 + 덧셈 한 번으로 곱셈 연산을 대체 가능
  - ※ 곱셈 연산기보다 시프트·덧셈 조합이 하드웨어적으로 단순하고 빠르므로, 성능 최적화에 널리 활용

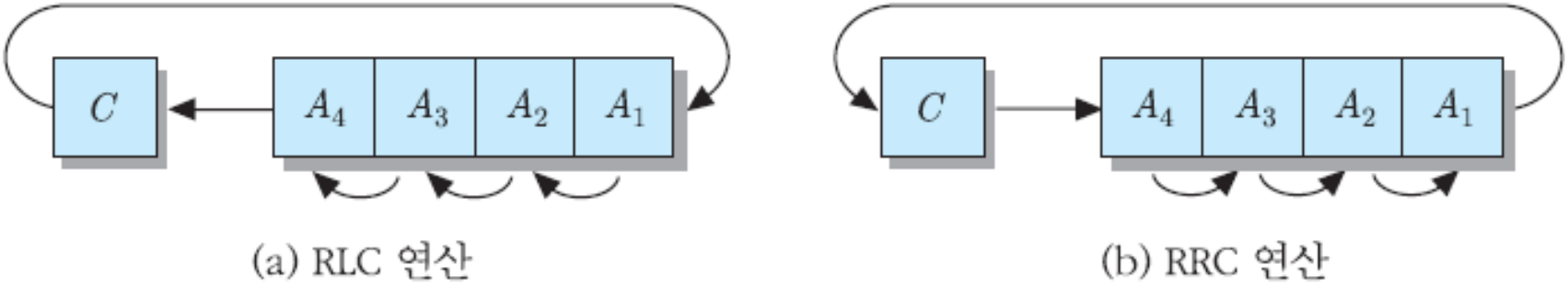
C 플래그를 포함한 시프트 연산

- 실제 CPU에서는 일반적으로 시프트 연산에 올림수(C) 플래그가 포함
  - **SHLC(shift left with carry):** C 플래그를 포함한 좌측-시프트
  - **SHRC(shift right with carry):** C 플래그를 포함한 우측-시프트

(a) SHLC 연산 (b) SHRC 연산

C 플래그를 포함한 시프트 연산

- **RLC(rotate left with carry)**: C 플래그를 포함하는 좌측 순환 시프트(회전) 연산
- **RRC(rotate right with carry)**: C 플래그를 포함하는 우측 순환 시프트(회전) 연산



(a) RLC 연산 (b) RRC 연산

- 산술적 우측 시프트에서 C 플래그의 값이 포함되면,
  - 데이터의 부호가 변경되는 걸 방지해야 됨
    - 오버플로우(V) 플래그 세트해서 알림
    - C 플래그가 레지스터로 들어오지 못하게 막음

컴퓨터 산술과 논리 연산

5. 정수의 산술 연산

기본적인 산술 연산들

$A \leftarrow \overline{A} + 1$; 보수화(2의 보수 변환)

$A \leftarrow A + B$; 덧셈

$A \leftarrow A - B$; 뺄셈

$A \leftarrow A \times B$; 곱셈

$A \leftarrow A \div B$; 나눗셈

$A \leftarrow A + 1$; 증가(increment)

$A \leftarrow A - 1$; 감소(decrement)

덧셈

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

- **[예]** 2진수 덧셈과 10진수 덧셈 비교

(a) (+3)+(+4)=

(b) (-3)+(+3)=

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

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

반가산기

- **반가산기(Half-Adder, HA):** 1bit 2진수 2개를 입력하여 합(Sum, **S**)과 캐리(Carry, **C**)를 출력하는 조합 논리 회로

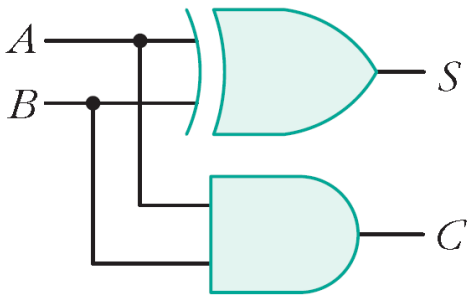
$$\begin{array}{r} A \\ +\ B \\ \hline C\ S \end{array} \qquad \begin{array}{r} 0 \\ +\ 0 \\ \hline 0\ 0 \end{array} \qquad \begin{array}{r} 0 \\ +\ 1 \\ \hline 0\ 1 \end{array} \qquad \begin{array}{r} 1 \\ +\ 0 \\ \hline 0\ 1 \end{array} \qquad \begin{array}{r} 1 \\ +\ 1 \\ \hline 1\ 0 \end{array}$$

입력		출력	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

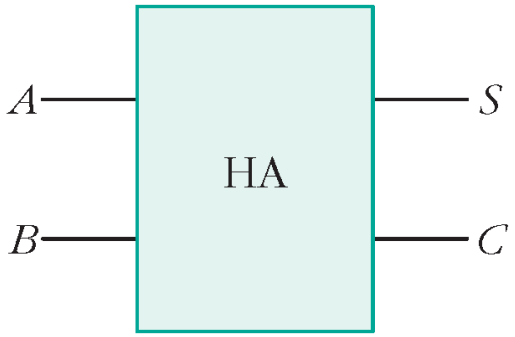
$S = \overline{A}B + A\overline{B} = A \oplus B$

$C = AB$

(a) 진리표와 논리식



(b) 논리 회로



(c) 논리 기호

전가산기

- **전가산기(Full-Adder, FA):** 2진수 입력 A, B와 아랫자리에서 올라온 캐리 C_i를 포함하여 1bit 2진수 3개를 더하는 조합 논리 회로

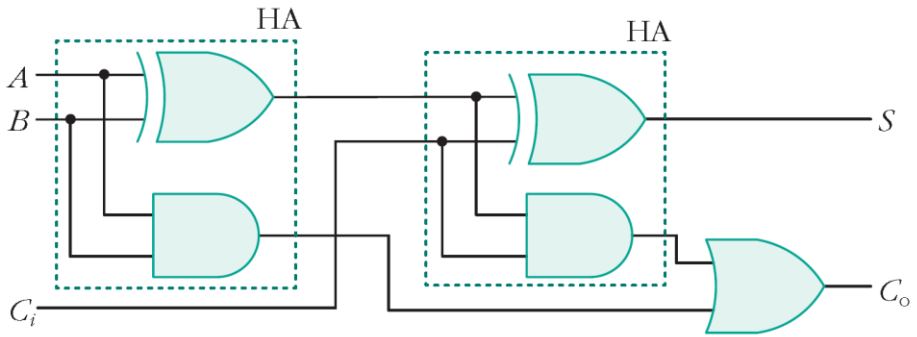
입력			출력	
A	B	C_i	S	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(a) 진리표

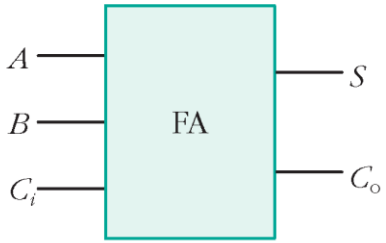
$$
\begin{aligned}
S &= \overline{A}\,\overline{B}C_i + \overline{A}B\overline{C_i} + A\overline{B}\,\overline{C_i} + ABC_i \\
&= \overline{A}(\overline{B}C_i + B\overline{C_i}) + A(\overline{B}\,\overline{C_i} + BC_i) \\
&= \overline{A}(B \oplus C_i) + A(\overline{B \oplus C_i}) \\
&= A \oplus (B \oplus C_i) = (A \oplus B) \oplus C_i
\end{aligned}
$$

$$
\begin{aligned}
C_o &= \overline{A}BC_i + A\overline{B}C_i + AB\overline{C_i} + ABC_i \\
&= C_i(\overline{A}B + A\overline{B}) + AB(\overline{C_i} + C_i) \\
&= C_i(A \oplus B) + AB
\end{aligned}
$$

(b) 논리식의 간소화



(c) 논리 회로

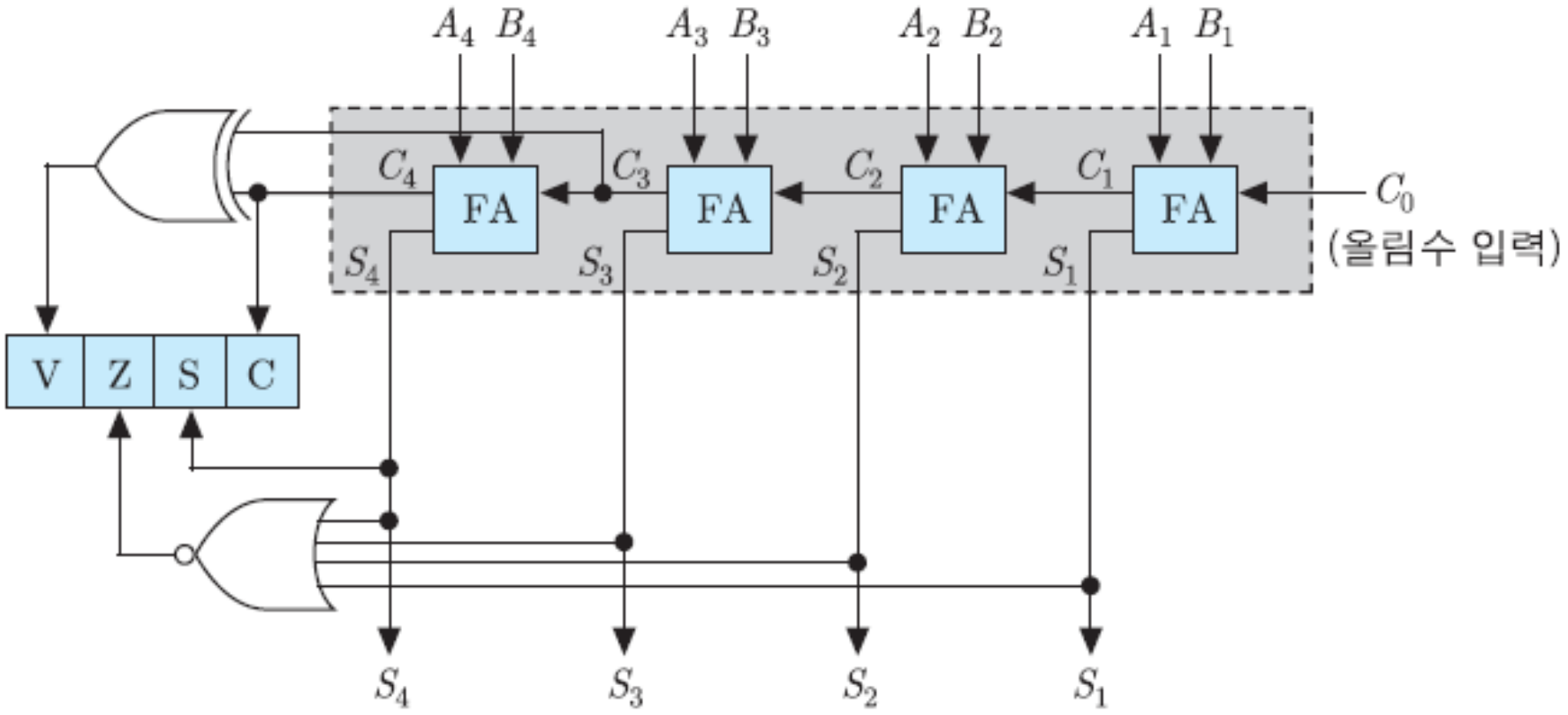


(d) 논리 기호

병렬 가산기(parallel adder)

- 덧셈을 수행하는 하드웨어 모듈
  - 비트 수만큼의 전가산기(FA)들로 구성
- 덧셈 연산 결과에 따라 해당 조건 플래그들(condition flags)을 세트
  - **V** 플래그: **오버플로우(overflow)**
    - 산술 연산의 결과 데이터가 표현 범위를 벗어났을 때 1로 설정
    - 오버플로우 발생: 에러 루틴 및 수정 조치
  - **Z** 플래그: **0(zero)**
    - 루프 및 기타 여러 용도 유용
    - 1이 하나라도 들어 있는지를 나타내는 비트를 제공하기 위해 OR 회로를 사용
    - Z 비트는 ALU의 모든 출력 비트를 OR한 후 반전(NOR)
  - **S** 플래그: **부호(sign)**
  - **C** 플래그: **올림수(carry)**
    - 맨 왼쪽 비트에서 데이터가 넘칠 때 세트
    - 가장 왼쪽 비트의 캐리는 정상 연산에서도 발생하므로 오버플로와 혼동하면 안 됨

4-비트 병렬 가산기와 상태 비트 제어회로



덧셈 오버플로우

- 덧셈 오버플로우: 덧셈 결과가 수의 표현 범위를 초과하여, 결과값이 틀리게 되는 상태
  - 검출 방법: 최상위 올림수와 차상위 올림수 간의 XOR를 이용

$$V = C_4 \ \mathbf{XOR} \ C_3$$

- **[예]** 2의 보수를 이용하여 4bit 덧셈에서, 오버플로우가 발생하는지 확인하라.

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

(b) (-7)+(-6)=

[TMI] 오버플로우

❑ 오버플로우 발생 조건

- A의 부호 비트와 B의 부호 비트가 다르면, 수의 표현 범위를 벗어나지 않기 때문에 오버플로우가 발생하지 않음
- 부호 비트가 둘다 0인데, 차상위 올림수가 1이면 오버플로우가 발생 → 양수를 더했는데, 결과값이 음수
- 부호 비트가 둘다 1인데, 차상위 올림수가 0이면 오버플로우가 발생 → 음수를 더했는데, 결과값이 양수

Input			Output		
A_{sign}	B_{sign}	C_{in} (차상위 올림수)	C_{out} (최상위 올림수)	Sum_{sign}	Overflow
0	0	0	0	0	0
0	0	**1**	**0**	1	**1**
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	1	1	0	0
1	1	**0**	**1**	0	**1**
1	1	1	1	1	0

뺄셈

- 덧셈을 이용하여 수행: 감수에 2의 보수를 취한 후, 피감수와 덧셈을 수행

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

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

A: 피감수(minuend), B: 감수(subtrahend)

- **[예]** 2의 보수를 이용하여 2진수 뺄셈을 수행하라.

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

(b) (7)-(+4)=

뺄셈 오버플로우

- 뺄셈 오버플로우: 뺄셈 결과가 수의 표현 범위를 초과하여, 결과값이 틀리게 되는 상태

- 검출 방법: 덧셈과 동일

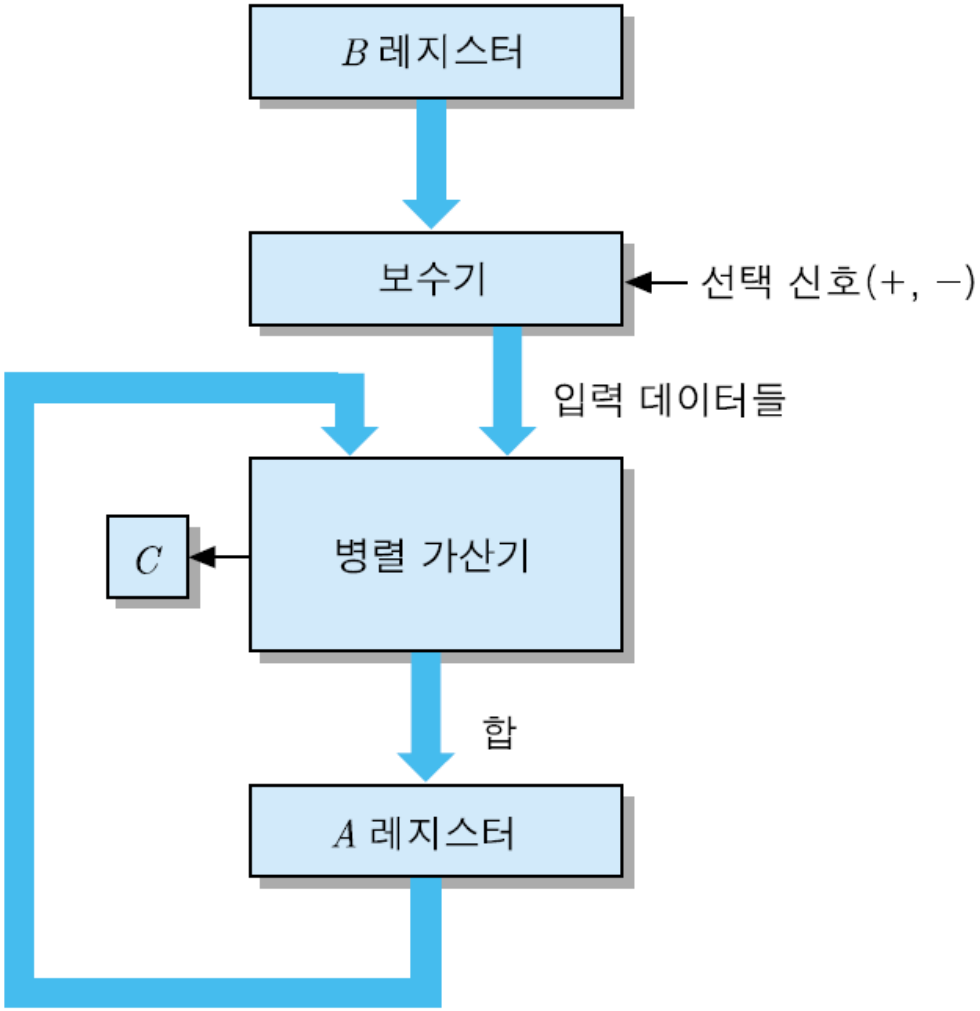
$$V = C_4 \ \mathbf{XOR}\ C_3$$

- **[예]** 2의 보수를 이용하여 4bit 뺄셈에서, 오버플로우가 발생하는지 확인하라.

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

(b) (-7)-(+6)=

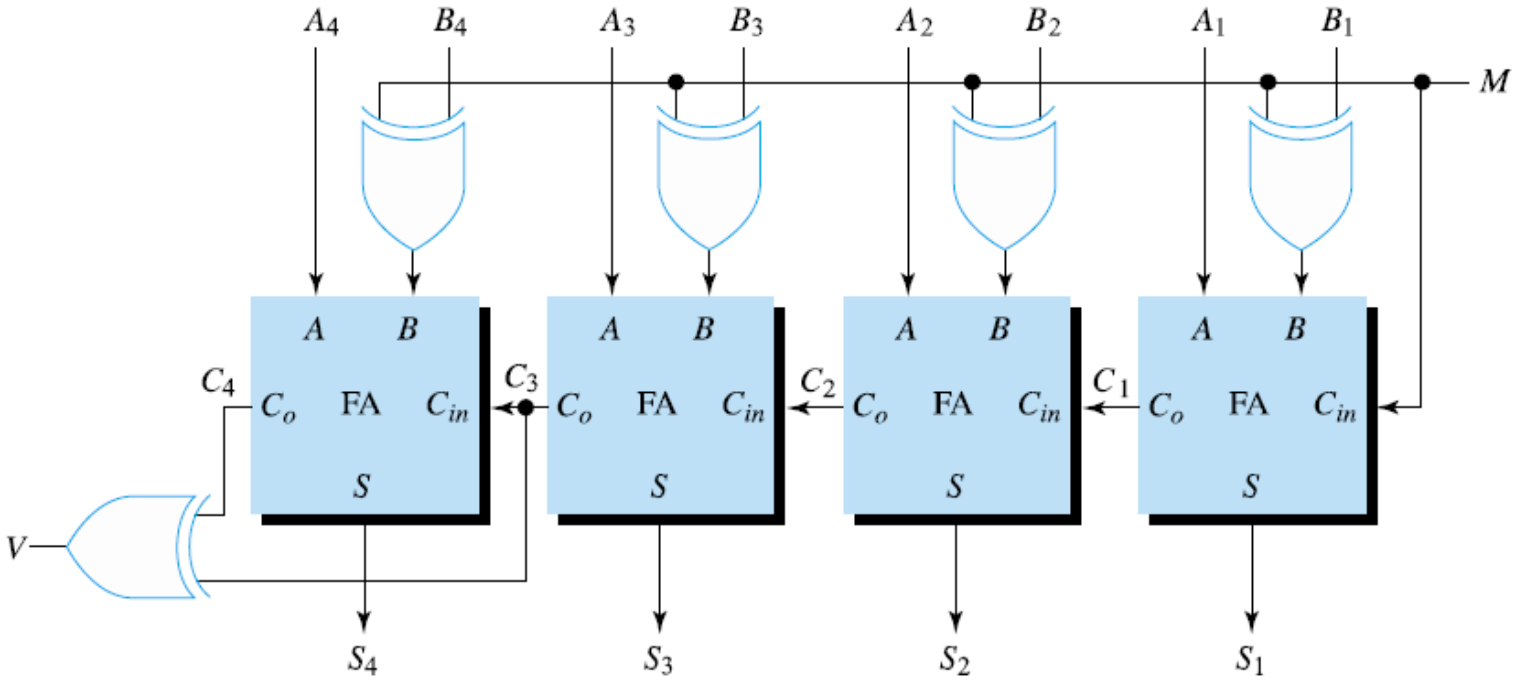
덧셈과 뺄셈 겸용 하드웨어의 블록 구성도



4-비트 병렬 가감산기

4-비트 병렬 가감산기(4-bit parallel adder/subtractor)

- 4-비트 데이터들 간의 덧셈(A+B) 및 뺄셈(A-B)을 모두 수행하는 조합회로
- 제어신호 M = 0이면, 덧셈을 수행
- 제어신호 M = 1이면, 뺄셈을 수행
  - 입력 B의 비트들을 반전하고, 최하위 올림수(C_0)로서 M을 입력



부호 없는 정수의 곱셈

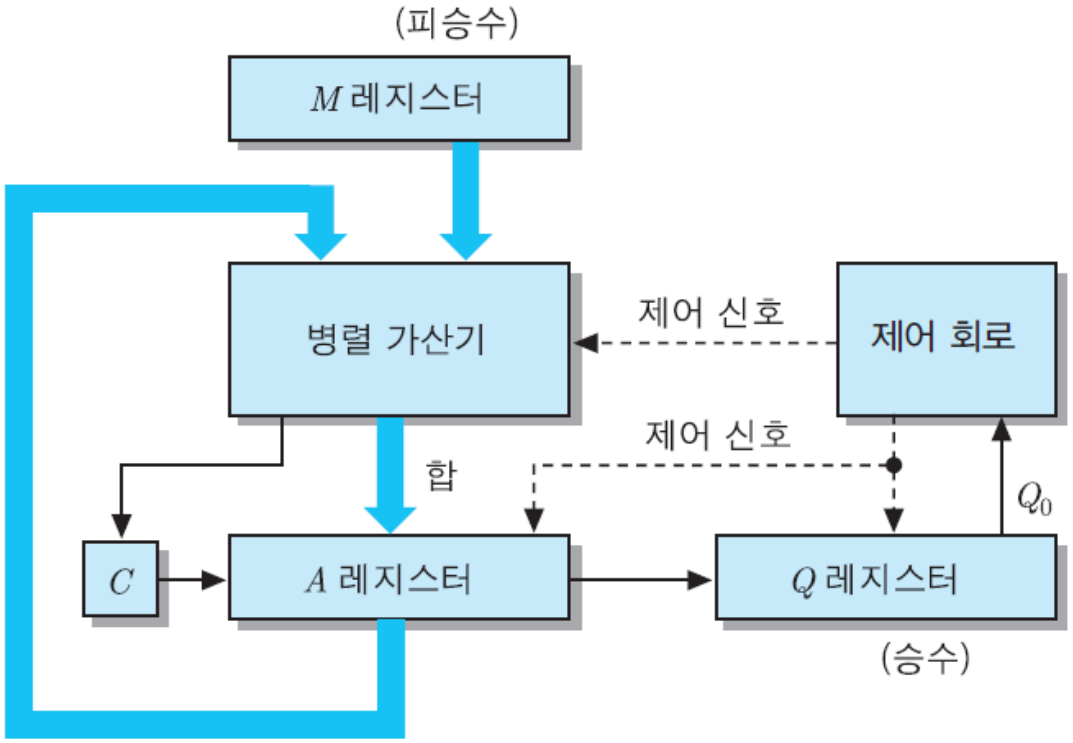
- 부호 없는 정수의 곱셈
  - 각 비트에 대하여 **부분 적(partial product)** 계산
  - 부분 적들을 모두 더하여 최종 결과를 얻음

- **[예제]** 2진수 곱셈(1101×1011)을 수행하고, 결과값이 10진수 곱셈의 결과(13×11=143)과 일치하는가?
- **[풀이]**

```
    1101      (피승수 = 13)
  × 1011      (승수=11)
  ------
    1101
   1101       부분 적들 (partial products)
  0000
 1101
 -------
 1000111      (최종 결과= 143)
```

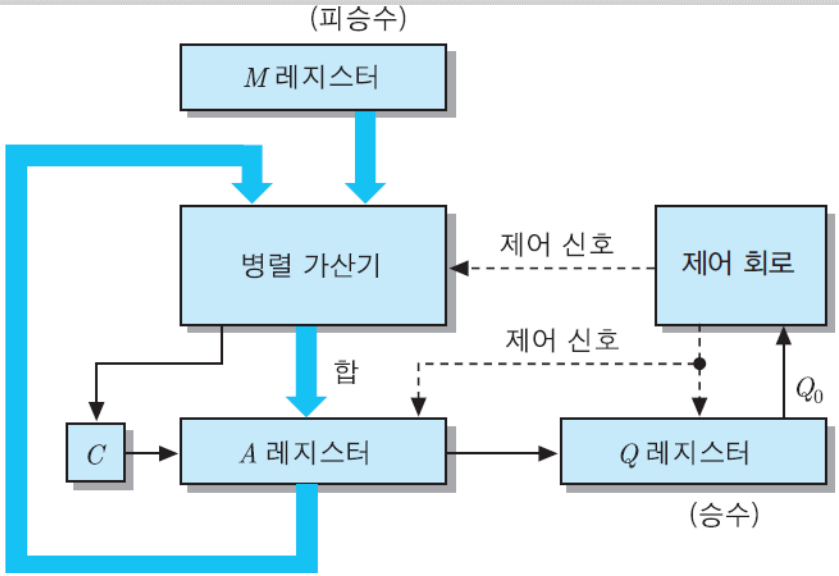
부호 없는 2진수의 곱셈

- 부호 없는 2진수 곱셈기의 하드웨어 구성도
  - **M 레지스터**: 피승수(multiplicand) 저장
  - **Q 레지스터**: 승수(multiplier) 저장
  - 두 배 길이의 결과값은 A 레지스터와 Q 레지스터에 저장
  - Q_0 비트가 1이면 피승수 덧셈 후 C-A-Q 레지스터 우측 시프트, Q_0 비트가 0이면 바로 C-A-Q 레지스터 우측 시프트



곱셈이 수행되는 과정에서의 레지스터 내용들

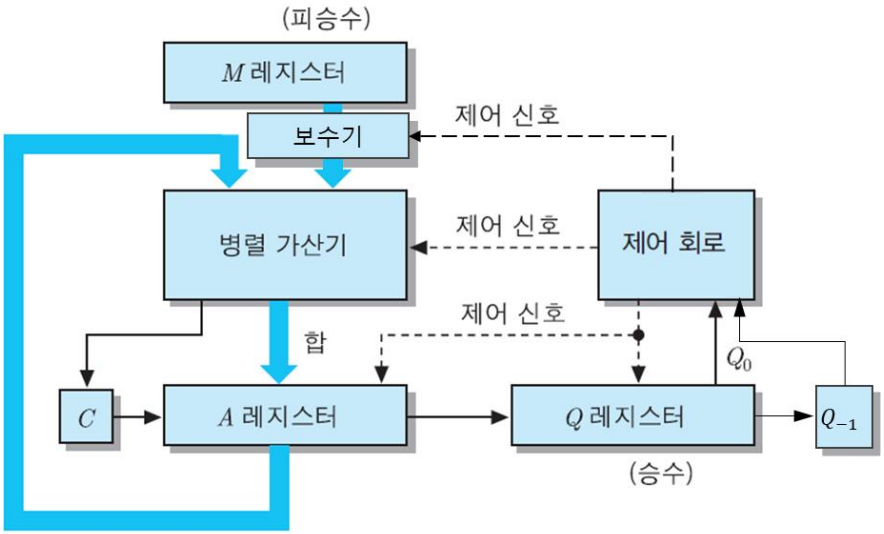
```
    1101
  × 1011
  ------
    1101
   1101
  0000
 1101
--------
10001111
```



	C	A	Q	
[초기 상태]	0	0000	1011	
[사이클 1]	0	1101	1011	; Q_0=1이므로, $A \leftarrow A+M$
	0	0110	1101	; 우측 시프트($C-A-Q$)
[사이클 2]	1	0011	1101	; Q_0=1이므로, $A \leftarrow A+M$
	0	1001	1110	; 우측 시프트($C-A-$Q)
[사이클 3]	0	0100	1111	; Q_0=0이므로, 우측 시프트($C-A-Q$)만 수행
[사이클 4]	1	0001	1111	; Q_0=1이므로, $A \leftarrow A+M$
	0	1000	1111	; 우측 시프트($C-A-Q$)

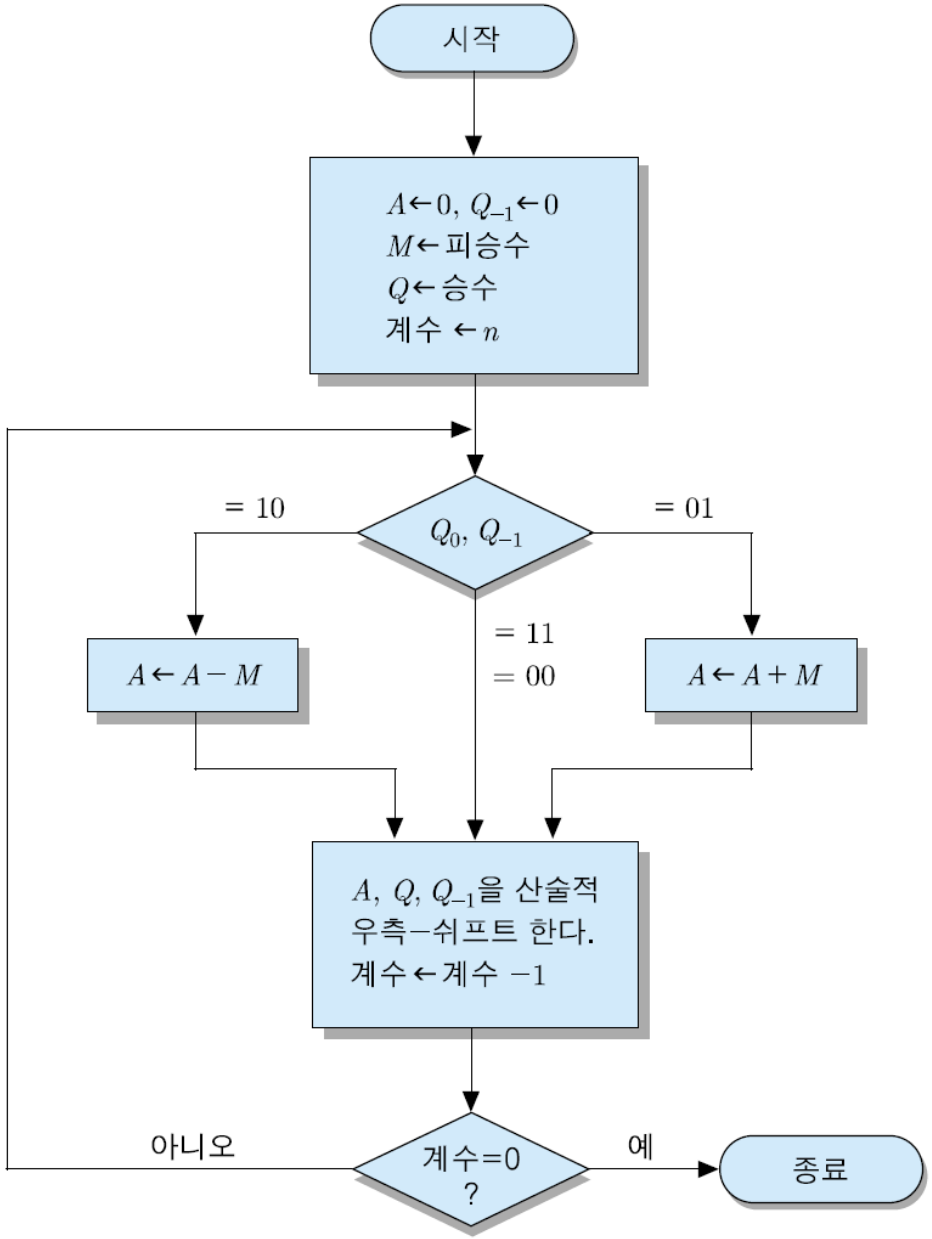
2의 보수들 간의 곱셈

❑ Booth 알고리즘(Booth's algorithm) 사용



- M 레지스터와 병렬 가산기 사이에 **보수기(complementer)** 추가
- Q 레지스터의 우측에 1-비트 레지스터($\boldsymbol{Q_{-1}}$)를 추가하고, 출력을 Q_0와 함께 제어 회로로 입력
- Q_0Q_{-1}비트가 00 또는 11이면, A, Q, Q_{-1} 산술적 우측 시프트
- Q_0Q_{-1}비트가 01이면, A=A+M, 연산 후 A, Q, Q_{-1} 산술적 우측 시프트
- Q_0Q_{-1}비트가 10이면, A=A-M, 연산 후 A, Q, Q_{-1} 산술적 우측 시프트

Booth 알고리즘의 흐름도



[예] Booth 알고리즘을 이용한 곱셈

❑ **[예]** -7×3=?

Q_0Q_{-1}	연산	시프트
00	None	산술적 우측 시프트
11	None	
01	A=A+M	
10	A=A-M	

(M) 1001
(Q) × 0011

	A	Q	Q_{-1}	계수	
초기값:	0000	0011	0	4	
	0111	0011	0		; $(Q_0\,Q_{-1})$ = (1 0)이므로, A로부터 피승수(1001)을 뺀다 (실제로는 그 보수인 0111을 더한다).
	0011	1001	1	3	; AQQ_{-1}을 산술적 우측-쉬프트 하고, 계수에서 1을 뺀다.
	0001	1100	1	2	; $(Q_0\,Q_{-1})$ = (1 1)이므로, AQQ_{-1}에 대한 산술적 우측-쉬프트만 하고, 계수에서 1을 뺀다.
	1010	1100	1		; $(Q_0\,Q_{-1})$ = (0 1)이므로, A에 피승수(1001)를 더하고, 계수에서 1를 뺀다.
	1101	0110	0	1	; AQQ_{-1}을 산술적 우측-쉬프트 하고, 계수에서 1을 뺀다.
	1110	1011	0	0	; $(Q_0\,Q_{-1})$ = (0 0)이므로, AQQ_{-1}을 산술적 우측-쉬프트 한다. 계수에서 1을 빼면 0이므로 계산이 종료되었다.

→ −21 (곱셈 결과)

[TMI] Booth 알고리즘 원리

- 0으로 둘러싸인 1의 블록을 가진 이진수와 피승수 M의 곱
  - **[예]** $M \times 00111110 = M \times (2^5 + 2^4 + 2^3 + 2^2 + 2^1) = M \times 62$
    - 괄호 안 5번의 연산을 2번으로 감소할 수 있음
    - $M \times (01000000 - 00000010) = M \times (2^6 - 2^1) = M \times 62$
  - 이를 일반화시키면, (…01…10…) = (…10…00…) – (…00…10…)로 나타낼 수 있음
    - 한 개의 1로 구성된 블록(010)도 가능
  - **[예]** $M \times 00111010 = M \times (2^5 + 2^4 + 2^3 + 2^1) = M \times 58$
    - $00111000 = 01000000 - 00001000 = +2^6 - 2^3$
    - $00000010 = 00000100 - 00000010 = +2^2 - 2^1$
    - $M \times (2^6 - 2^3 + 2^2 - 2^1) = M \times 58$

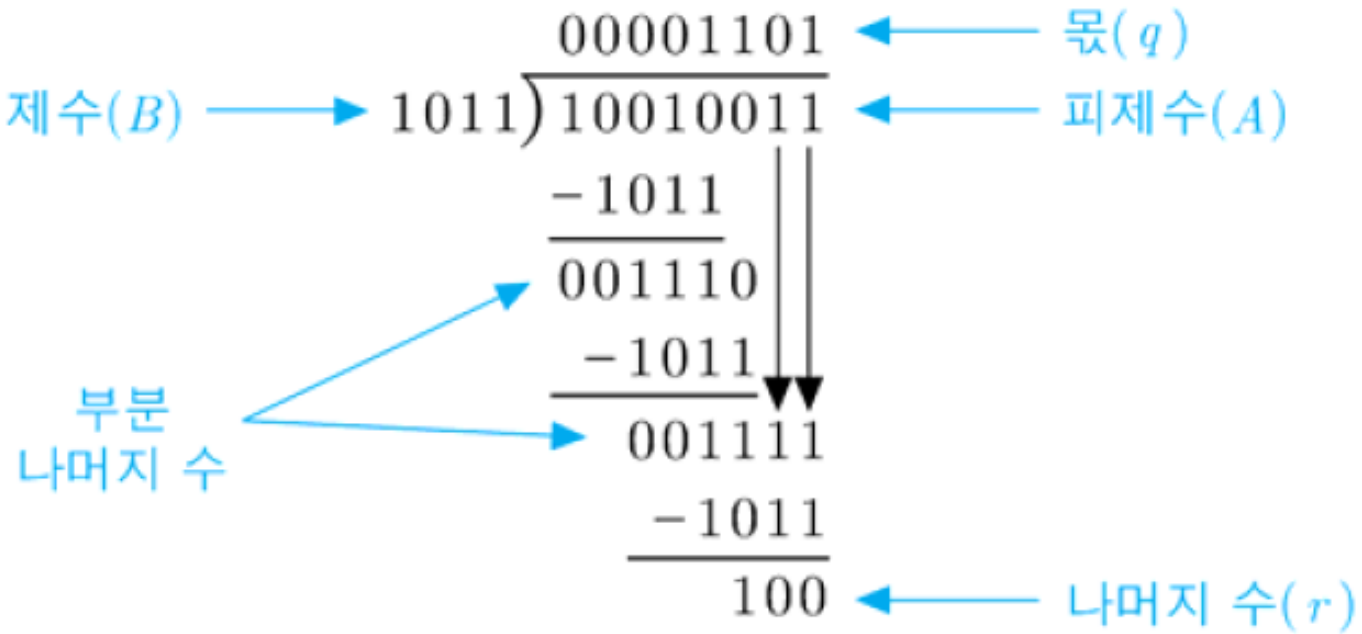
- Booth 알고리즘에서 1의 블록을 처음 만났을 때(01) 덧셈을 수행하고, 1의 블록이 끝날 때(10) 뺄셈을 수행
  - 1의 블록이 길어질수록 연산 횟수는 감소

나눗셈

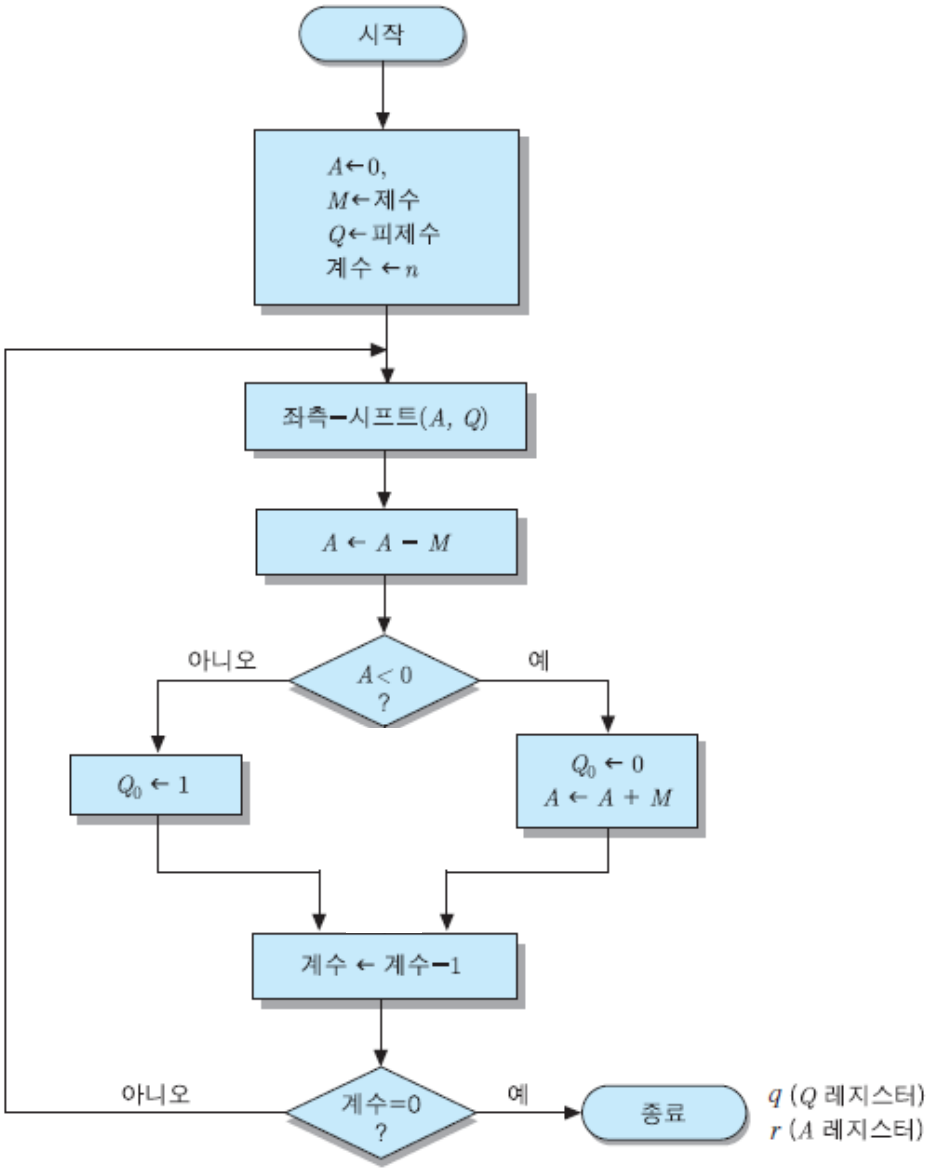
나눗셈의 수식 표현

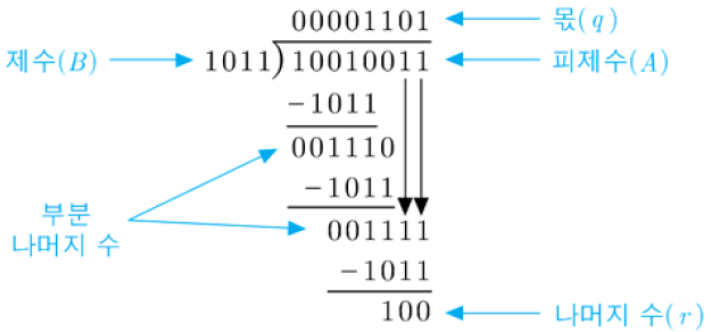
- **A ÷ B = q ····· r**
  - **A**: 피제수(dividend), **B**: 제수(divisor)
  - **q**: 몫(quotient) **r**: 나머지 수(remainder)

부호 없는 2진 나눗셈



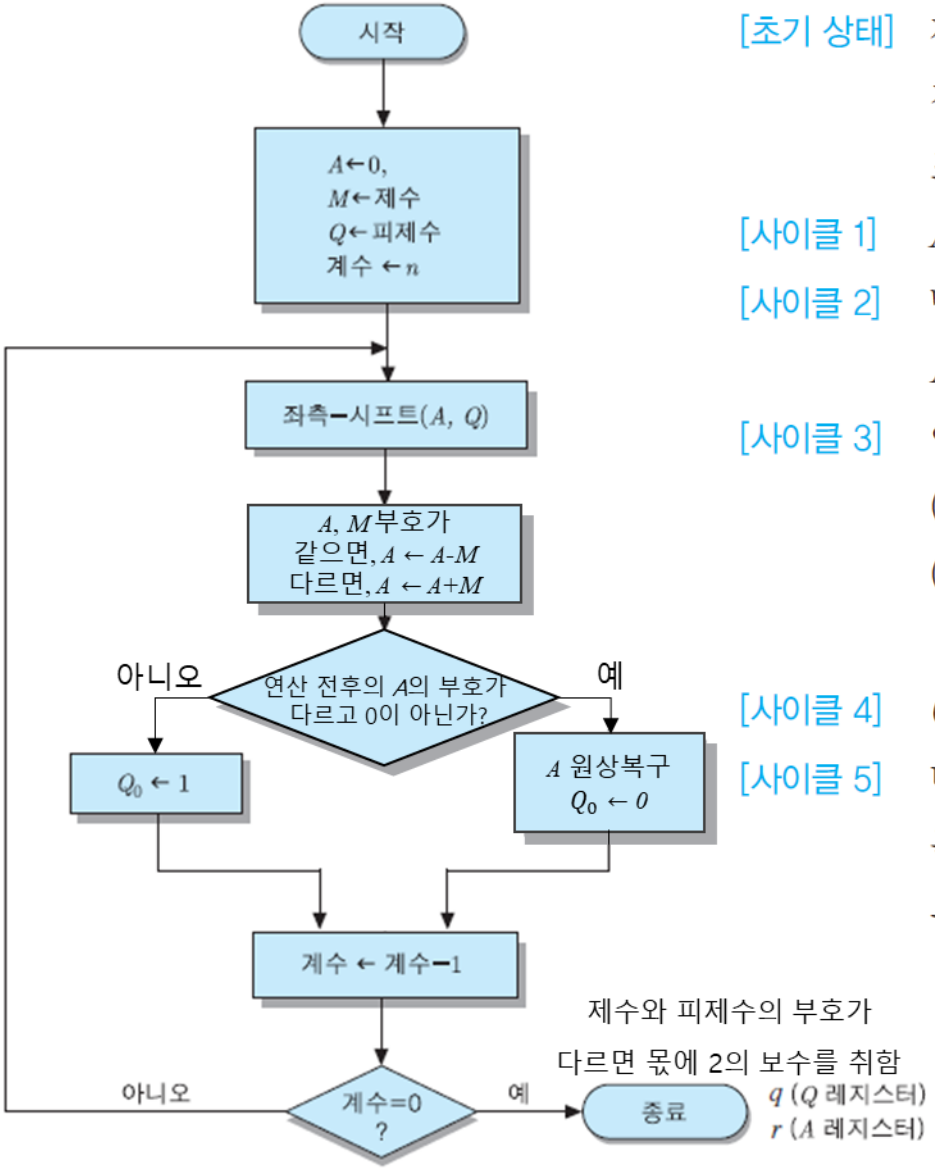
부호 없는 2진 나눗셈 알고리즘의 흐름도





A	Q	비고
00000000	10010011	초기값
00000001	00100110	좌측 시프트, $Q_0 \leftarrow 0$
00000010	01001100	좌측 시프트, $Q_0 \leftarrow 0$
00000100	10011000	좌측 시프트, $Q_0 \leftarrow 0$
00001001	00110000	좌측 시프트, $Q_0 \leftarrow 0$
00010010 -1011 ---------- 00000111	01100000 ---------- 01100001	좌측 시프트 ---------- 제수 뺄셈, $Q_0 \leftarrow 1$
00001110 -1011 ---------- 00000011	11000010 ---------- 11000011	좌측 시프트 ---------- 제수 뺄셈, $Q_0 \leftarrow 1$
00000111	10000110	좌측 시프트, $Q_0 \leftarrow 0$
00001111 -1011 ---------- 00000100	000001100 ---------- 000001101	좌측 시프트 ---------- 제수 뺄셈, $Q_0 \leftarrow 1$

2의 보수 나눗셈 방법



[초기 상태] 제수는 M 레지스터에, 피제수는 A와 Q 레지스터에 저장한다. 각 레지스터가 n비트일 때, 피제수는 $2n$비트 길이의 2의 보수로 표현한다.

[사이클 1] A와 Q 레지스터를 좌측으로 한 비트씩 시프트 한다.

[사이클 2] 만약 M과 A의 부호가 같다면 $A \leftarrow A-M$을 수행하고, 다르다면 $A \leftarrow A+M$을 수행한다.

[사이클 3] 연산 전과 후에서 A의 부호가 같다면, 위의 연산은 성공이다.

(a) 연산이 성공이거나 $A=0$이면, $Q_0 \leftarrow 1$로 세트한다.

(b) 연산이 실패이고 $A \neq 0$이면, $Q_0 \leftarrow 0$으로 세트하고, A를 이전의 값으로 복구한다.

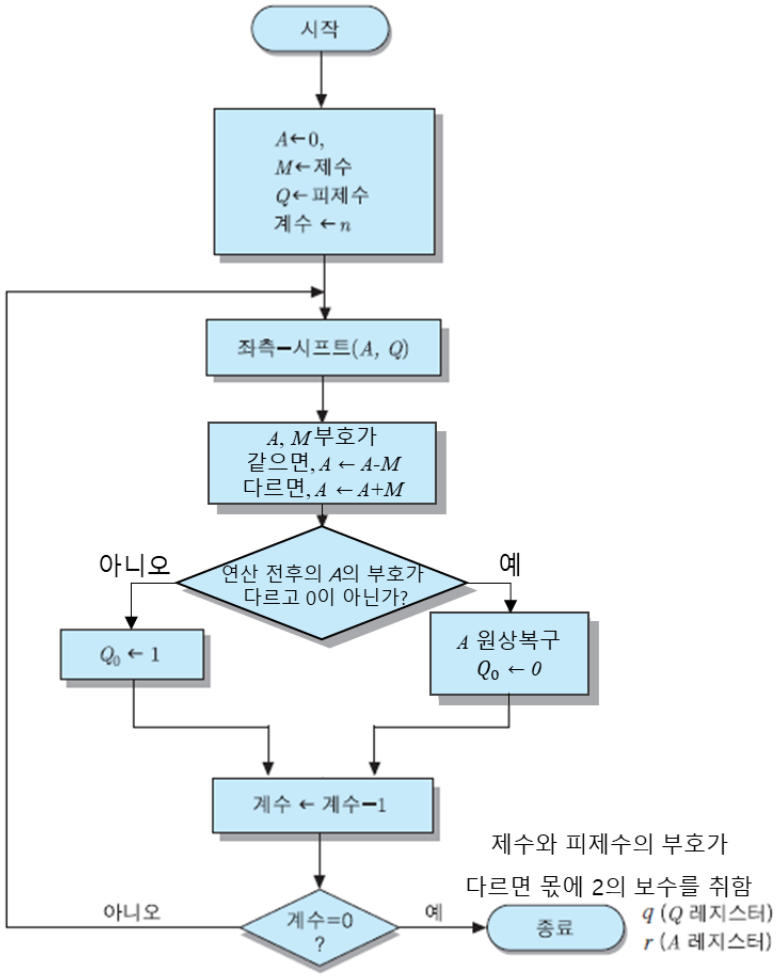
[사이클 4] Q 레지스터에 비트가 남아있다면, 단계 2에서 4까지를 반복한다.

[사이클 5] 나머지 수는 A에 남는다. 만약 제수와 피제수의 부호가 같다면 몫은 Q 레지스터의 값이 되고, 그렇지 않다면 Q 레지스터의 내용에 대한 2의 보수가 몫이 된다.

2의 보수 나눗셈 방법

❑ [예] 7÷(-3)

- 제수 (-3)에 대한 2의 보수 표현인 '1101'을 M 레지스터에 저장하고, 아래의 연산을 순차적으로 수행



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

연산 결과 : 몫=Q의 내용(0010)에 대한 2의 보수인 '1110' (−2)
나머지=A 레지스터의 내용인 '0001' (+1)

곱셈 정리

- 부호 없는 2진수의 곱셈
  - Q_0 비트가 1이면 $A \leftarrow A+M$ (피승수), 연산 후 우측 시프트
  - Q_0 비트가 0이면, 연산 없이 우측 시프트

- 부호 있는 2진수의 곱셈(Booth 알고리즘)
  - Q_0Q_{-1}가 00 또는 11이면, 연산 없이 산술적 우측 시프트
  - Q_0Q_{-1}가 01이면, $A \leftarrow A+M$, 연산 후 산술적 우측 시프트
  - Q_0Q_{-1}가 10이면, $A \leftarrow A-M$, 연산 후 산술적 우측 시프트

- $2N$ 비트의 연산 결과는 A 레지스터와 Q 레지스터에 저장

나눗셈 정리

- 부호 없는 2진수의 나눗셈
  - 좌측 시프트 수행
  - 현재 A 레지스터의 값이 M(제수)보다 크면, $A \leftarrow A - M$ 연산 후 $Q_0 \leftarrow 1$
- 부호 있는 2진수의 나눗셈
  - 좌측 시프트 수행
  - A, M의 부호가 같으면, $A \leftarrow A - M$ 수행
  - A, M의 부호가 다르면, $A \leftarrow A + M$ 수행
  - 만약 연산 전후 A 레지스터의 데이터 부호가 다르고 0이 아니면, A 레지스터는 연산 전 데이터로 복구
  - 그렇지 않다면, $Q_0 \leftarrow 1$
- A 레지스터에는 나머지, Q 레지스터에 몫이 저장
  - 부호가 있는 2진수의 나눗셈인 경우, 피제수와 제수의 부호가 다르면, Q 레지스터의 데이터에 2의 보수를 취함

Q & A

End!