ALU 개요

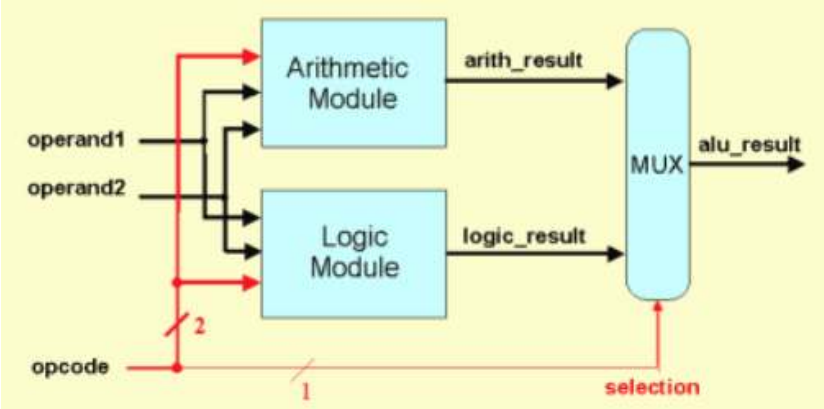
1. ALU 개요

- **ALU**
  - 수치 데이터에 대한 산술적 계산과 논리 데이터에 대한 논리 연산을 수행하는 하드웨어 모듈.
  - ALU는 CPU의 기본 Functional Unit.

- **ALU의 구성요소**
  - 산술 연산 장치 : 사칙 연산(+,-,*,/)을 수행
  - 논리 연산 장치 : 논리 연산(AND, OR, NOT, XOR)을 수행
  - 쉬프트 연산 : 데이터를 좌/우측으로 비트단위로 이동하는 연산을 수행.
  - 보수기 : 데이터에 대한 2의 보수 연산 수행.
  - 상태 레지스터 : 연산 결과의 상태를 나타내는 플래그(Flag)를 저장하고 있는 레지스터

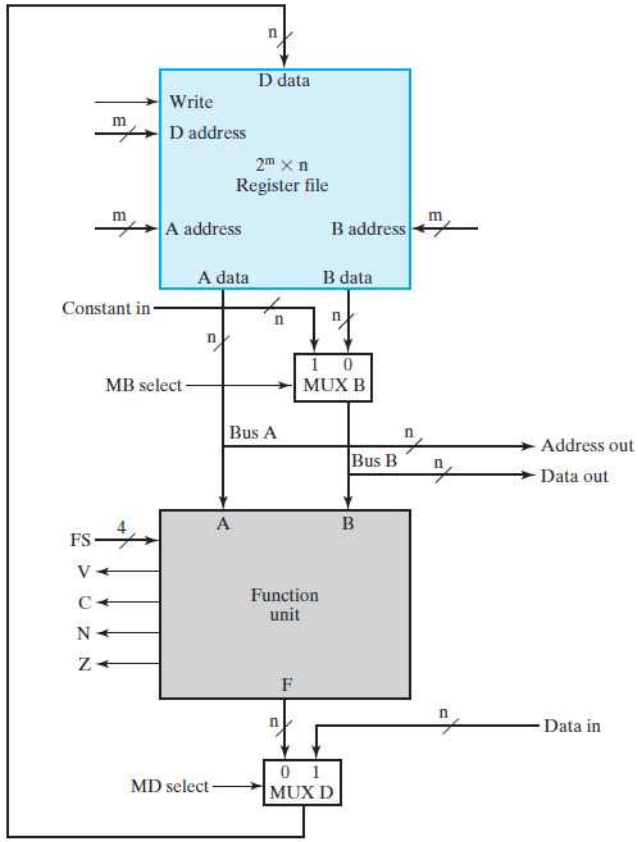
1. ALU 개요

- **ALU 동작**
  - 레지스터 또는 메모리로부터 ALU에서 사용할 오퍼랜드를 입력
  - 입력된 데이터에 대한 연산 유형의 선택과 데이터 흐름 제어에 필요한 제어신호들을 제어 유니트로부터 제공받음.
  - 선택된 연산을 실행
  - 연산 결과에 따라 상태 레지스터의 플래그들을 설정
  - 연산 결과를 지정된 레지스터 또는 메모리에 저장.



1. ALU 개요

- **ALU와 레지스터의 구성 예시**



정수 연산

1. 산술 연산 개요

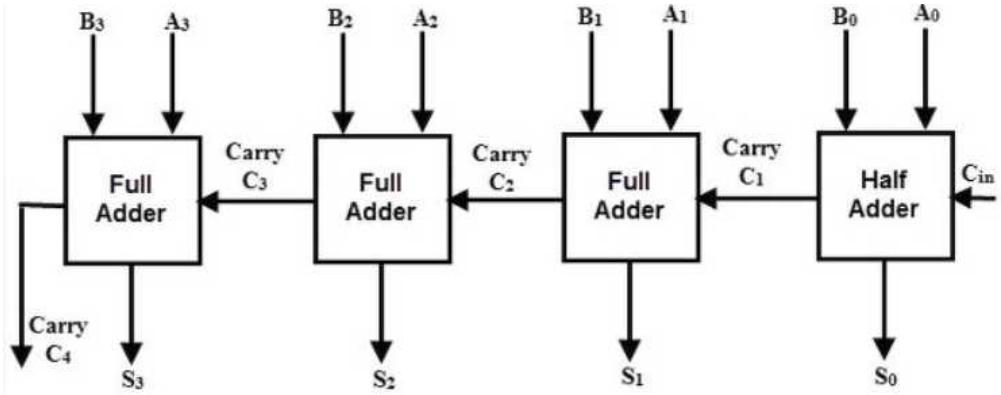
- **데이터에 대한 산술 연산은 정수 연산과 실수 연산으로 구분할 수 있다.**

- **정수 연산**
  - 보수 변환 : 2의 보수 변환
  - 증가/감소(Increment/Decrement) : +/-1
  - 정수 덧셈/뺄셈
  - 정수 곱셈/나눗셈

- **실수 연산**
  - 실수 덧셈/뺄셈
  - 실수 곱셈/나눗셈

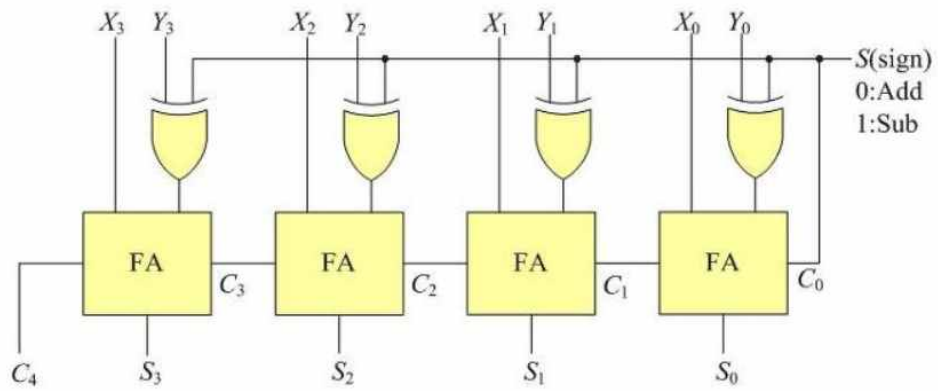
2. 가산기

- **병렬 가산기(Parallel Adder)**
  - 전가산기(Full Adder)를 n개 연결하여 n-bit 가산기 구현
  - 하위 비트 연산결과 발생하는 Carry 를 상위비트 연산에 반영.



3. 가감산기

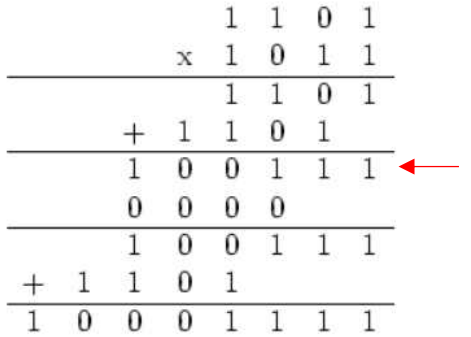
- **병렬 가감산기(Parallel Adder/Subtracter)**
  - 가산기회로에 2의 보수기(Complementer)를 포함시켜 덧셈과 뺄셈을 같이 수행.
  - 제어 비트(S) 를 사용해 덧셈과 뺄셈을 구분 : 0(덧셈), 1(뺄셈)



4. 곱셈기

- **곱셈기**
  - 디지털 신호처리, 통신 시스템 분야에서 곱셈기의 역할이 증가.
    - 컴퓨터 시스템 성능에 큰 영향

이진 곱셈 (Pencil & Paper)



부분 적
(Partial Product)

개선 방법 (한번에 두 오퍼팬드 덧셈)

4.1 곱셈기 특징

- **곱셈기의 특징**
  - n-bit와 n-bit를 곱하면, 결과는 2n 비트가 되기 때문에, 결과를 저장하기 위해서는 2개의 레지스터 또는 메모리 장소를 사용해야 한다.
  - 부호없는 n-bit 곱셈
    - 가산기와 쉬프터로 구현가능.
  - 부호있는 n-bit 곱셈
    - 음수를 양수로 변환 후, 부호 계산
    - 2의 보수 곱셈기

4.2 곱셈기 유형

- **곱셈기 구현 유형**
  - 소프트웨어 곱셈기 : 소프트웨어적으로 구현, 속도 느리다.
  - 하드웨어 곱셈기 : 복잡한 곱셈연산을 하드웨어로 구현하여 곱셈 실행속도를 향상

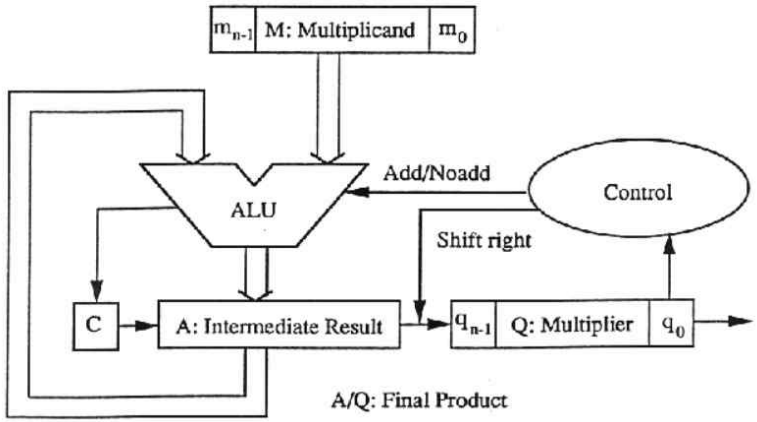
- **하드웨어 곱셈기**
  - 직렬(Serial) 곱셈기
    - Adder 와 Shifter 를 사용한 구현 = 가장 간단한 구현방식
    - 부분 적(Partial Product)를 계산한 후 누적해가는 방식.
  - 병렬(Parallel) 곱셈기
    - 부분 적이 병렬로 계산되는 구조.
    - 고성능 곱셈기 구현가능

4.3 직렬 곱셈기

- **부호없는 곱셈기 회로 경우,**
  - M : 피승수(Multiplicand)
  - Q : 승수(Multiplier)
  - 곱셈 결과 : (A+Q)

n-bit Multiplication 경우, n 번 반복

if ($q_0 = 1$) then $A \leftarrow A + M$;
Shift right (A and Q) by 1 bit



4.4 병렬 곱셈기

- **병렬 곱셈기**
  - 부분 적(Partial Product)을 병렬로 계산해서 계산 속도를 향상
    - 많은 양의 하드웨어가 필요
    - 가산기를 배열(Array) 형태로 배치해서 병렬로 덧셈 수행 → Array Multiplier

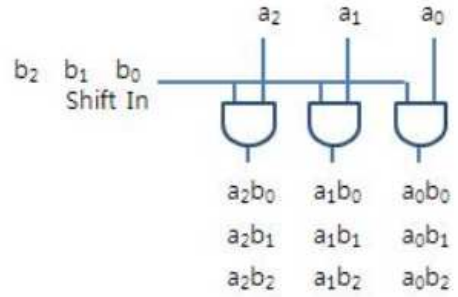
- **Array Multiplier**
  - 장점
    - 가장 단순한 구조 : 동일한 구조를 2차원으로 배열하기 때문에 설계 용이
    - 파이프라이닝 적용과 비트 폭 확장이 용이
  - 단점
    - 많은 칩 면적 필요
    - 소비전력이 증가

4.4.1 Array 곱셈기

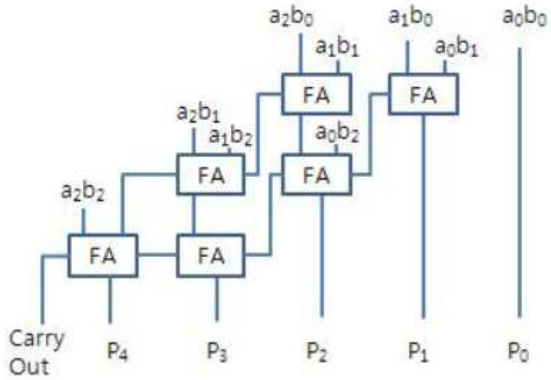
- **3-bit 곱셈**

		a_2	a_1	a_0	피승수(multiplicand)
	x	b_2	b_1	b_0	승수(multiplier)
	a_2b_0	a_1b_0	a_0b_0		부분적(partial products)
a_2b_1	a_1b_1	a_0b_1			
+ a_2b_2	a_1b_2	a_0b_2			
P_4	P_3	P_2	P_1	P_0	곱/적(product)

곱셈 절차



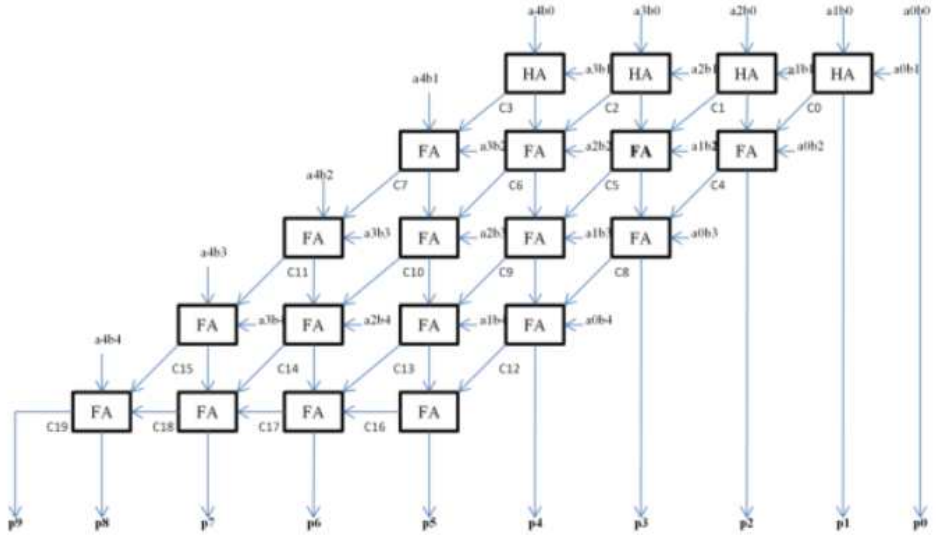
부분 적(Partial Product) 발생회로



3x3 곱셈기 (Array Multiplier)

4.4.1 Array 곱셈기

- 4 x 4 병렬 곱셈기 구조



4.5 고속 병렬 곱셈기

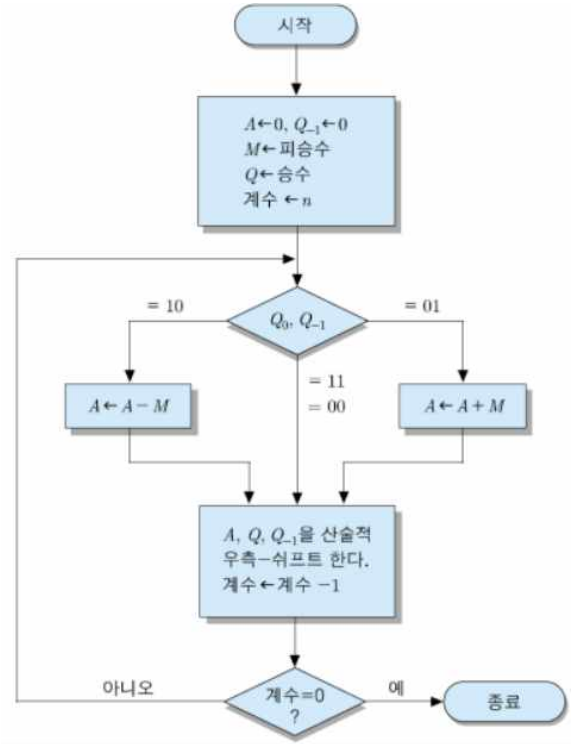
- **고속 병렬 곱셈기**
  - 부분 적 계산 량을 줄이거나 계산 방법을 개선해서 계산 속도를 향상시키거나, 하드웨어 비용을 절약.

- **대표적 고속 병렬 곱셈 알고리즘**
  - Booth 알고리즘
  - 보-울리(Baugh-Wooley)
    - 2의 보수 곱셈기.
  - Wallace Tree
    - 부분 적(Partial product)을 더할 때 스테이지 수를 감소해서 고속화

4.5.1 Booth 곱셈기

- **Booth 알고리즘**
  - 2의 보수 곱셈을 용이하게 수행

- **Booth 알고리즘 순서도**
  - M : 피승수(Multiplicand)
  - Q : 승수(Multiplier)
  - 곱셈 결과 : (A+Q)

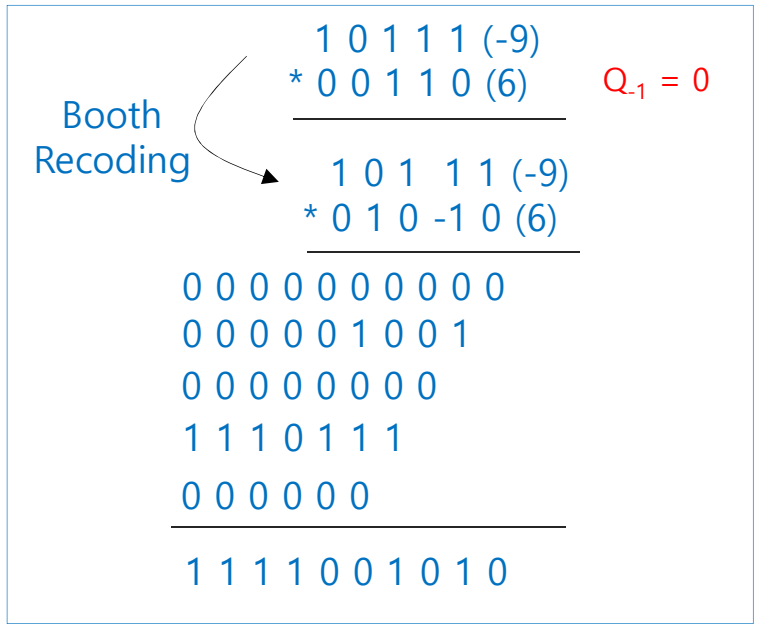


4.5.1 Booth 곱셈기

- **Booth 알고리즘 계산 절차**
  - Multiplier 를 Booth Value로 변환한다. (Boot Recoding)
  - Booth value에 따라 부분 적을 구해서 누적한다.

- **Booth 알고리즘 예제 : -9 * 6**

Q	Q_{-1}	Booth Value
0	0	0
0	1	+1
1	0	-1
1	1	0

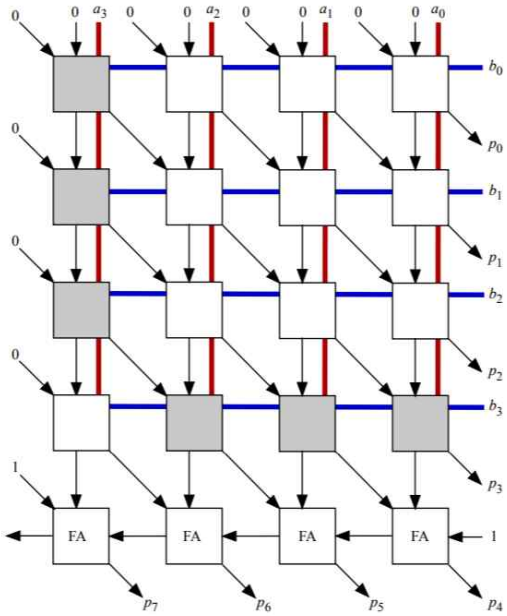


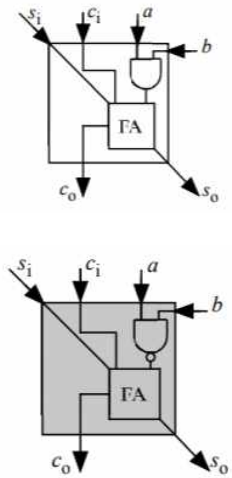
4.5.1 Booth 곱셈기

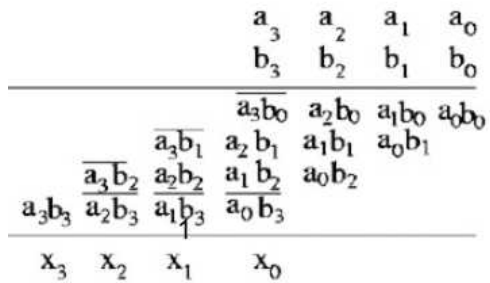
- **Booth 알고리즘 고속화**
  - Bit-Pair Recoding
    - Booth Recoding 된 multiplier 비트를 2비트씩 그룹핑해서 처리
    - 부분 적(Partial Product) 수를 ½로 줄여서, 부분 적 누산과정을 고속화.

4.5.2 Baugh-Wooley 곱셈기

- **4x4 Baugh-Wooley 곱셈기**
  - 2의 보수 곱셈 구조
  - 음수 부분적(partial product)을 마지막 단계에서 수행하게 함으로써 누적 연산 속도 향상.



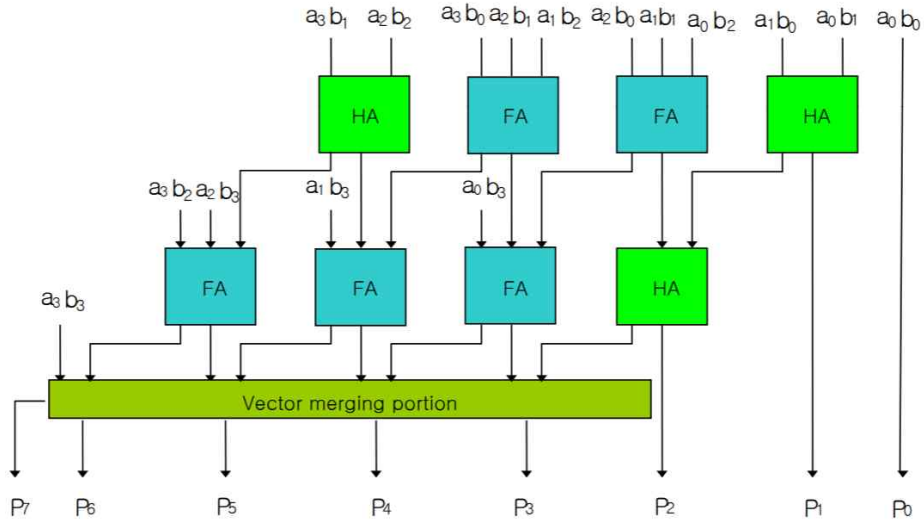




4.5.3 Wallace Tree 곱셈기

- **Array 곱셈기보다 고속**
  - Tree의 높이가 Array 구조보다 낮다.
  - n-bit 곱셈 경우
    - Array 곱셈기의 Array 높이 : n
    - Wallace Tree 높이 : log n
  - 장점 : 고속
  - 단점 : 복잡한 구조

- **4x4 Wallace Tree 곱셈기**



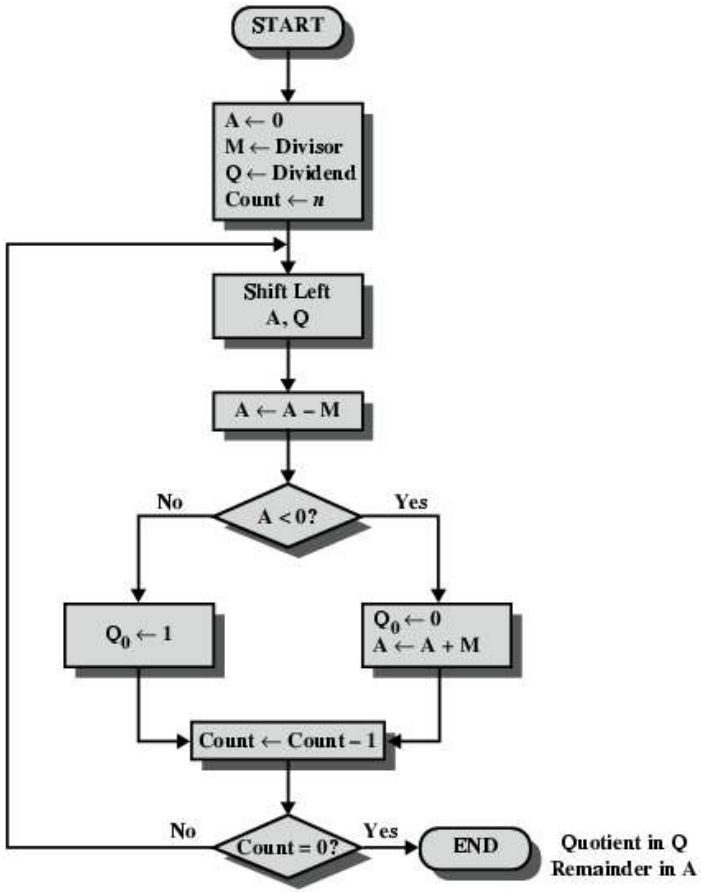
5. 나눗셈기

- **나눗셈**
  - A / B = Q, R
    - A : 피제수(Dividend)
    - B : 제수(Divisor)
    - Q : 몫(Quotient)
    - R : 나머지(Remainder)
  - 나눗셈 중간에 발생하는 나머지 : Partial Remainder

- **부호화 정수에 대한 나눗셈**
  - 부호없는 정수의 나눗셈 수행 후, 보수를 사용하여 부호 결정

5. 나눗셈기

- **나눗셈 실행 흐름도**
  - 사용하여 부호 결정



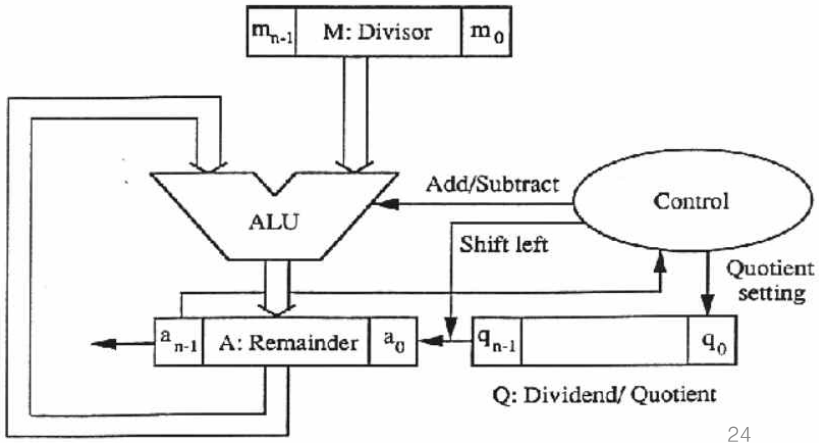
5. 나눗셈기

- **나눗셈 예제**

```
                 000010101   Quotient
Divisor  1101 ) 100010010   Dividend
               -1101
                 10000
                 -1101
                   1110
                   -1101
                       1   Remainder
```

n-bit Division 경우, n 번 반복

{
left shift A and Q by 1 bit
$A \leftarrow A - M$;
if $A < 0$ ($a_{n-1} = 1$), then
$q_0 \leftarrow 0$, $A \leftarrow A + M$; // Restore
else
$q_0 \leftarrow 1$
}



실수 연산

1. 부동소수점 표현

- **부동 소수점 구성**
  - 부호(Sign)
  - 지수(Exponent) : 표현 범위 결정
  - 가수(Mantissa) : 정밀도 결정

Sign	Exponent	Mantissa

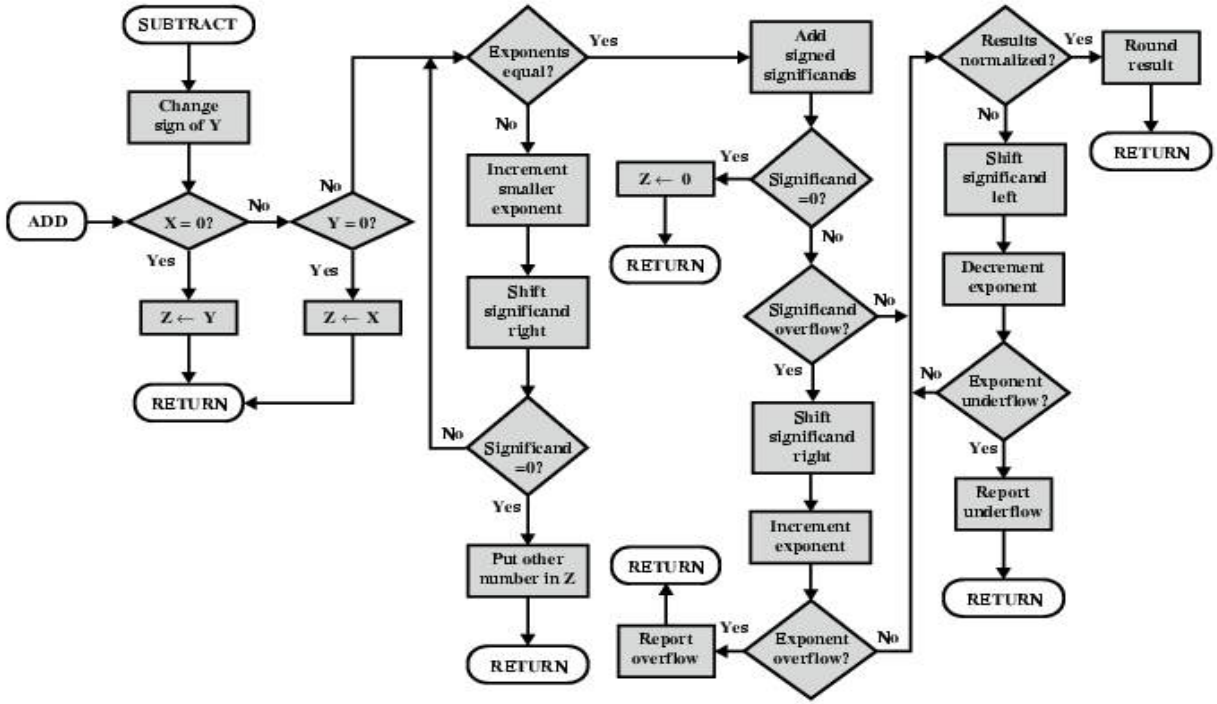
- **표현 범위와 정밀도를 높이는 방법**
  - 많은 비트를 할당
    - 32-bit : 단일-정밀도(Single Precision)
    - 64-bit : 복수-정밀도(Double Precision)

2. 부동소수점 덧셈/뺄셈

- **덧셈/뺄셈 절차**
  - 지수 정렬
    - 지수를 동일하게 맞춘다. 작은 지수를 큰 지수로 일치 시킨다.
  - 가수 덧셈/뺄셈
    - 가수에 대해 덧셈/뺄셈을 수행한다.
  - 정규화(Normalize)
    - 계산 결과에 대해 정규화를 수행한다.
  - 오버플로우 검사
    - 정규화 과정에서 지수가 표현범위를 초과했는지를 검사한다.
    - 초과한 경우 오버플로우 처리한다.
  - 자리 맞춤(Rounding)
    - 유효자리에 맞도록 가수부분을 자리 맞춤한다.
    - 필요에 따라 절삭 또는 반올림을 적용한다.

2. 부동소수점 덧셈/뺄셈

- 부동 소수점 덧셈/뺄셈 흐름도



3. 부동소수점 곱셈/나눗셈

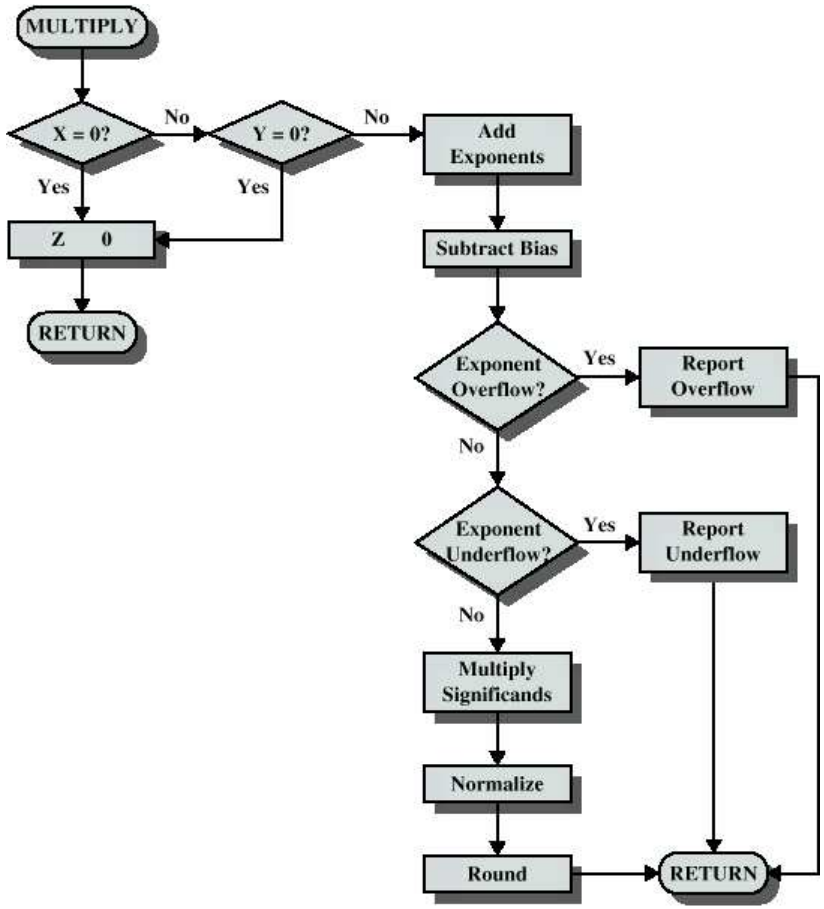
- **곱셈/나눗셈 절차**
  - 지수 덧셈/뺄셈
    - 곱셈/나눗셈에 따라 지수를 더하거나 뺀다.
    - 지수 계산 결과 값을 127-초과 코드로 조정한다.
  - 가수 곱셈/나눗셈
    - 가수에 대해 곱셈/나눗셈을 수행한다.
  - 정규화
    - 계산 결과에 대해 정규화를 수행한다.
    - 정규화 과정에서 발생할 수 있는 데이터 손실을 방지하기 위해 Guard bit를 두어 정확도를 향상시킨다.

3. 부동소수점 곱셈/나눗셈

- **곱셈/나눗셈 절차 (계속)**
  - 오버 플로우 검사
    - 정규화 과정에서 지수가 표현 범위를 초과했는지를 검사한다.
    - 초과한 경우 오버 플로우 처리한다.
  - 자리 맞춤
    - 유효 자리에 맞도록 가수 부분을 자리 맞춤한다.
    - 필요에 따라 절삭 또는 반올림을 적용한다.
  - 부호 결정
    - 오퍼랜드의 부호를 조사하여 동일한 경우 양수로 처리한다.

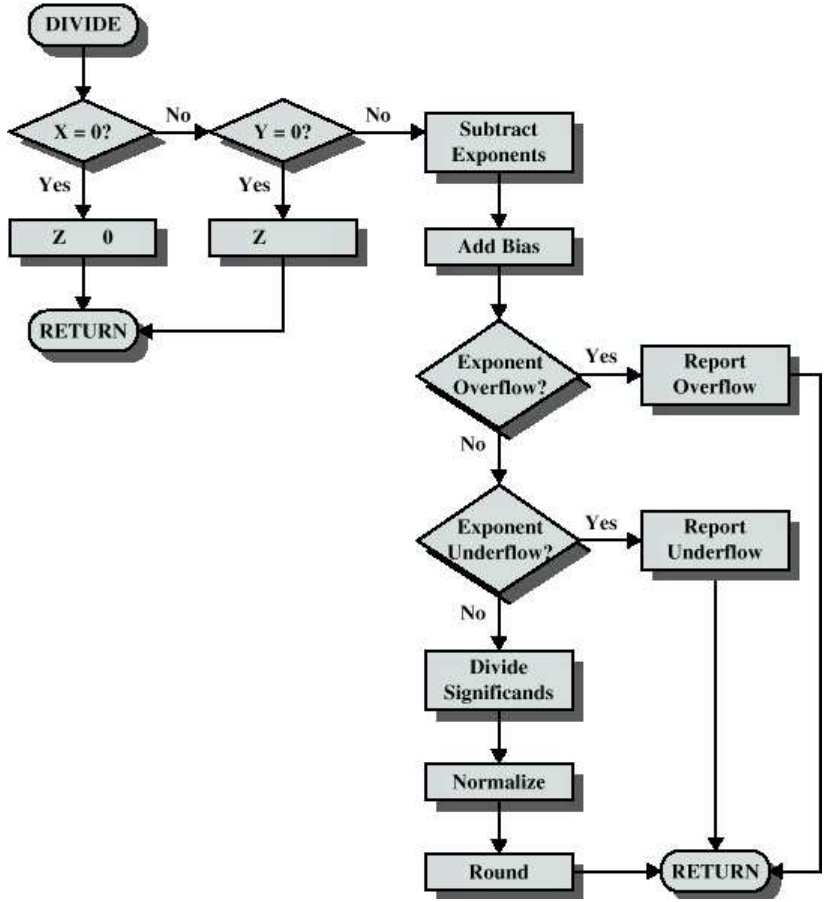
3. 부동소수점 곱셈/나눗셈

- 부동 소수점 곱셈 흐름도



3. 부동소수점 곱셈/나눗셈

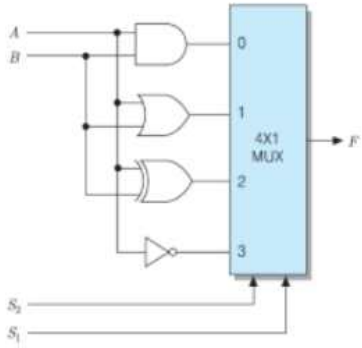
- 부동 소수점 나눗셈 흐름도



논리 연산과 쉬프트

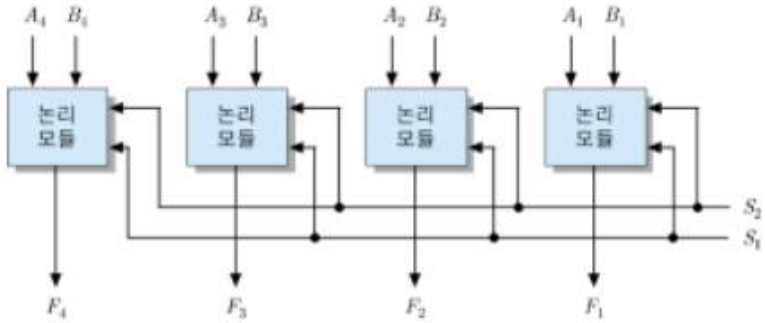
1. 논리 연산

- ALU에서의 논리연산을 비트단위로 수행된다.
- n-비트 논리연산 회로는 1-bit 회로를 병렬로 연결하여 구성



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

1-bit 논리연산 회로



n-bit 논리연산 회로

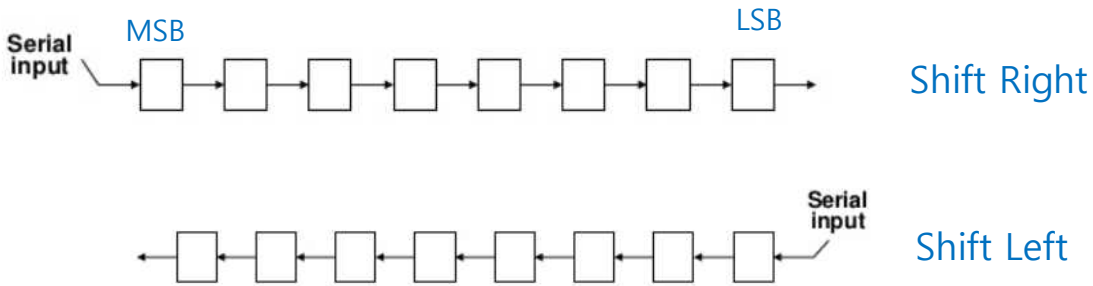
2. 쉬프트 연산

- **쉬프트 연산**
  - 레지스터에 저장된 값을 MSB 또는 LSB쪽으로 1-bit 또는 n-bit 단위로 이동하는 연산
    - 비트단위(Bit-wise) 연산
    - Binary shift 라고도 함.

- **쉬프트시 고려사항**
  - 쉬프트 방향 : Left/Right
  - 부호비트 처리 : 산술/논리 쉬프트
  - Carry Out 처리 : Shift/Rotate

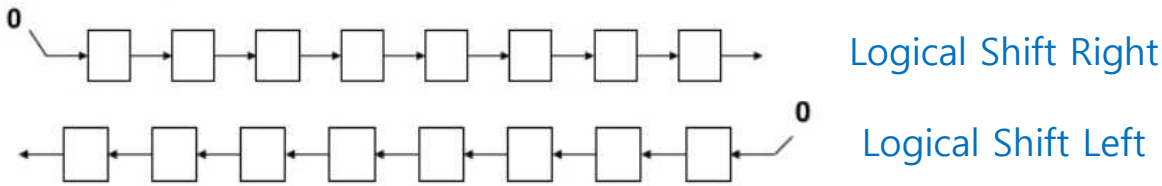
2.1 쉬프트 방향

- **쉬프트 방향**
  - 왼쪽 쉬프트(Shift Left) : MSB 방향으로 이동.
    - 결과값 = 저장값 * 2n , n = 쉬프트 크기
  - 오른쪽 쉬프트(Shift Right) : LSB 방향으로 이동.
    - 결과값 = 저장값 / 2n , n = 쉬프트 크기
  - 쉬프트 연산을 사용하여 곱셈과 나눗셈 구현가능.



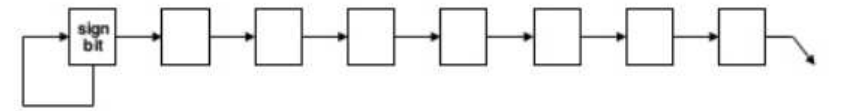
2.2 부호비트 처리

- **부호비트 처리**
  - 쉬프트 연산에서 MSB (부호비트) 처리 방식에 따라 산술 쉬프트, 논리 쉬프트로 구분
    - 산술 쉬프트(Arithmetic Shift) : 부호 연장(Sign-Extension)
    - 논리 쉬프트(Logical Shift) : '0' 추가
    - 왼쪽 산술 쉬프트인 경우 반드시 overflow 검사를 실시해야 한다.

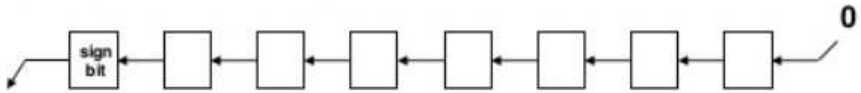


2.2 부호비트 처리

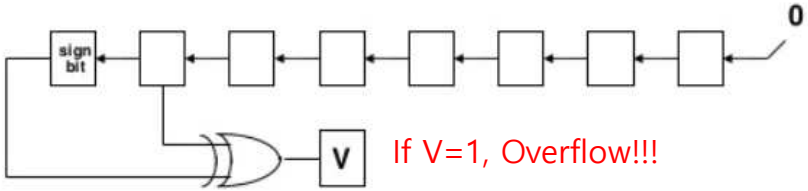
- **부호비트 처리 (계속)**



Arithmetic Shift Right



Arithmetic Shift Left

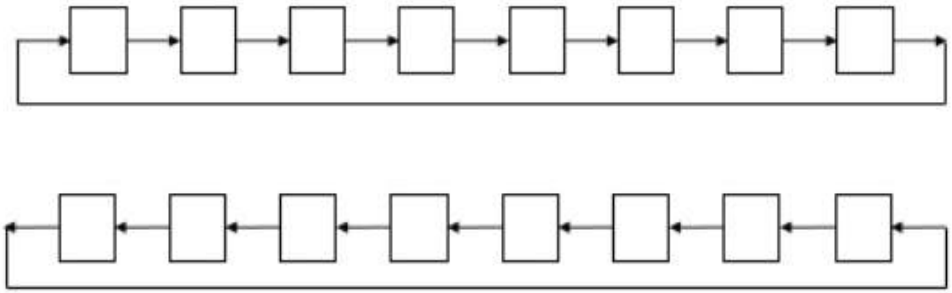


If V=1, Overflow!!!

Overflow Check

2.3 Carry-out 처리

- **Carry-out 처리**
  - 쉬프트 결과 외부로 출력되는 MSB, LSB(Carry-out) 처리 방법에 따라 Shift/Rotate로 구분
    - MSB, LSB 출력을 버림 : Shift 연산
    - MSB, LSB 출력을 각각 LSB, MSB로 연결 : Rotate 연산 (Circular Shift)

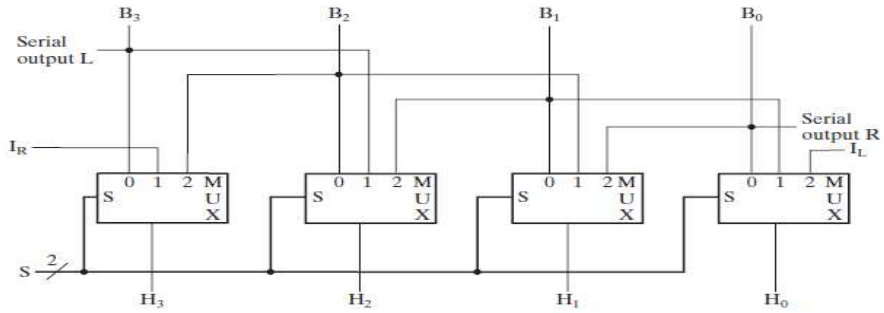


Rotate Right

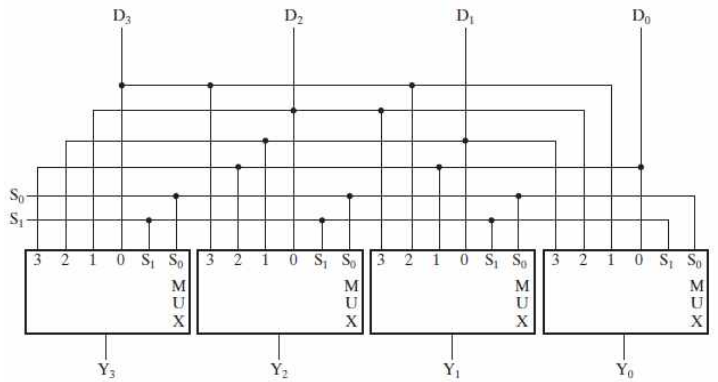
Rotate Left

2.4 쉬프터 회로

- **배럴 쉬프터(Barrel shifter)**
  - 한번에 n-bit 쉬프트가 가능한 쉬프트 회로



4-bit Shifter



4-bit Barrel Shifter

레지스터 개요

1. 레지스터 개요

- **레지스터(Register)**
  - 플립플럽(Flip-Flop) 또는 래치(Latch)를 워드(Word)단위로 병렬 연결한 데이터 저장 회로.
  - 메모리 계층의 최상위에 위치하며, 가장 빠른 속도로 접근 가능.

- **레지스터 길이**
  - 레지스터에 포함된 플립 플럽 수
  - CPU의 워드(word) 길이와 동일

- **레지스터 용도**
  - 데이터의 임시 저장
  - CPU 동작과 관련된 제어 및 상태 정보 저장

1.1 레지스터 특징

- **특징**
  - 가장 빠른 저장 장소
  - CPU와 가장 가까운 데이터 저장 장소.
  - 가장 높은 하드웨어 비용
  - CPU가 보유하고 있는 레지스터 수와 기능은 CPU 종류에 따라 다르다.
    - 프로세서 설계 주요 요소.

1.2 레지스터 분류

- **레지스터 분류**
  - 프로그래머에게 보이는 레지스터 (User-visible Register)
    - 사용자가 프로그램을 통해서 액세스 가능한 레지스터
    - 레지스터를 최적으로 사용함으로써 메모리 액세스를 최소화할 수 있음.

  - 제어 및 상태 레지스터 (Control & Status Register)
    - 프로세서 동작 제어를 위해 제어 유니트가 사용하는 레지스터.
    - 프로그램 실행 제어를 위해, 운영체제에 의해 사용.

2. 프로그래머에게 보이는 레지스터

- **일반 목적용 레지스터**
  - 프로그래머에 의해 다양한 용도로 사용가능.

- **데이터 레지스터**
  - 순수 데이터 저장용 레지스터

- **주소 레지스터**
  - 주소를 저장하고 있는 레지스터
    - 어드레싱 모드에서 사용.
  - 주소 레지스터 유형
    - 세그먼트 포인터 (Segment Pointer) : 세그먼트의 시작 주소 (base address) 저장.
    - 인덱스 레지스터 (Index Register) : 인덱스 주소 지정에서 사용.
    - 스택 포인터 (Stack Pointer) : 스택의 최상위 위치 지정.

2. 프로그래머에게 보이는 레지스터

- **조건 코드 (Condition Codes) 레지스터**
  - 조건 코드 (플래그) 저장.
  - 연산 결과 값에 대한 상태 정보 : 프로세서 H/W 가 설정.
  - 프로그램에 의해 묵시적으로 참조되지만, 설정은 불가.
  - 제어 레지스터의 일부

- **User-visible 레지스터는 함수단위에서 독립적으로 사용된다.**
  - 함수 호출시 User-visible 레지스터에 저장된 내용은 자동적으로 특정 메모리 위치(주로 Stack)에 보관되었다가 Return시 복구된다.

3. 제어 및 상태 레지스터

- **제어 및 상태 레지스터(Control & Status Register)**
  - 사용자(프로그래머)가 직접 접근할 수 없다.
    - 특정 실행 권한을 요구하는 명령어를 사용하여 접근 가능
  - CPU 마다 제어 및 상태 레지스터의 구조 및 명칭이 다름.

- **명령어 실행에 필수 레지스터**
  - 프로세서와 메모리 사이의 데이터 이동에 필요한 레지스터
    - 프로그램 카운터 (PC : program counter) : 다음에 실행할 명령어의 위치 저장.
    - 명령어 레지스터 (IR : instruction register) : 최근에 인출한 명령어 저장
    - 메모리 주소 레지스터 (MAR : memory address register) : 메모리 어드레스 정보 저장
    - 메모리 버퍼 레지스터 (MBR : memory buffer register) : 메모리에 쓰거나 읽은 데이터를 저장.

3. 제어 및 상태 레지스터

- **제어 및 상태 레지스터**
  - 조건 코드 및 상태 정보들을 저장하고 있는 레지스터
    - 플래그(Flag) 또는 필드(Field) 형태로 저장.
  - 컴퓨터에 따라 사용하는 명칭이 상이
    - PSW(Program Status Word), SR(Status Register), …

- **조건코드(Condition Code)**
  - Sign : ALU 연산 결과의 부호 비트
  - Zero : 연산결과가 '0; 인경우, '1'로 설정
  - Carry : 연산과정에서 Carry 가 발생했을 경우, '1'로 설정
  - Overflow : 연산 과정에서 오버플로우가 발생했을 경우 '1'로 설정
  - Equal : 논리 비교 결과, 두 오퍼랜드가 동일했을 경우, '1'로 설정

3. 제어 및 상태 레지스터

- **상태정보**
  - 인터럽트 활성화 비트 : 인터럽트 허용 여부를 설정할 때 사용.
  - Supervisor 비트 : 프로세서의 동작모드(운영체제 모드, 사용자 모드)를 설정.

- **운영체제를 사용하는 경우, 운영체제 목적에 따라 다양한 제어 및 상태 레지스터를 별도로 정의해서 사용할 수 있다.**

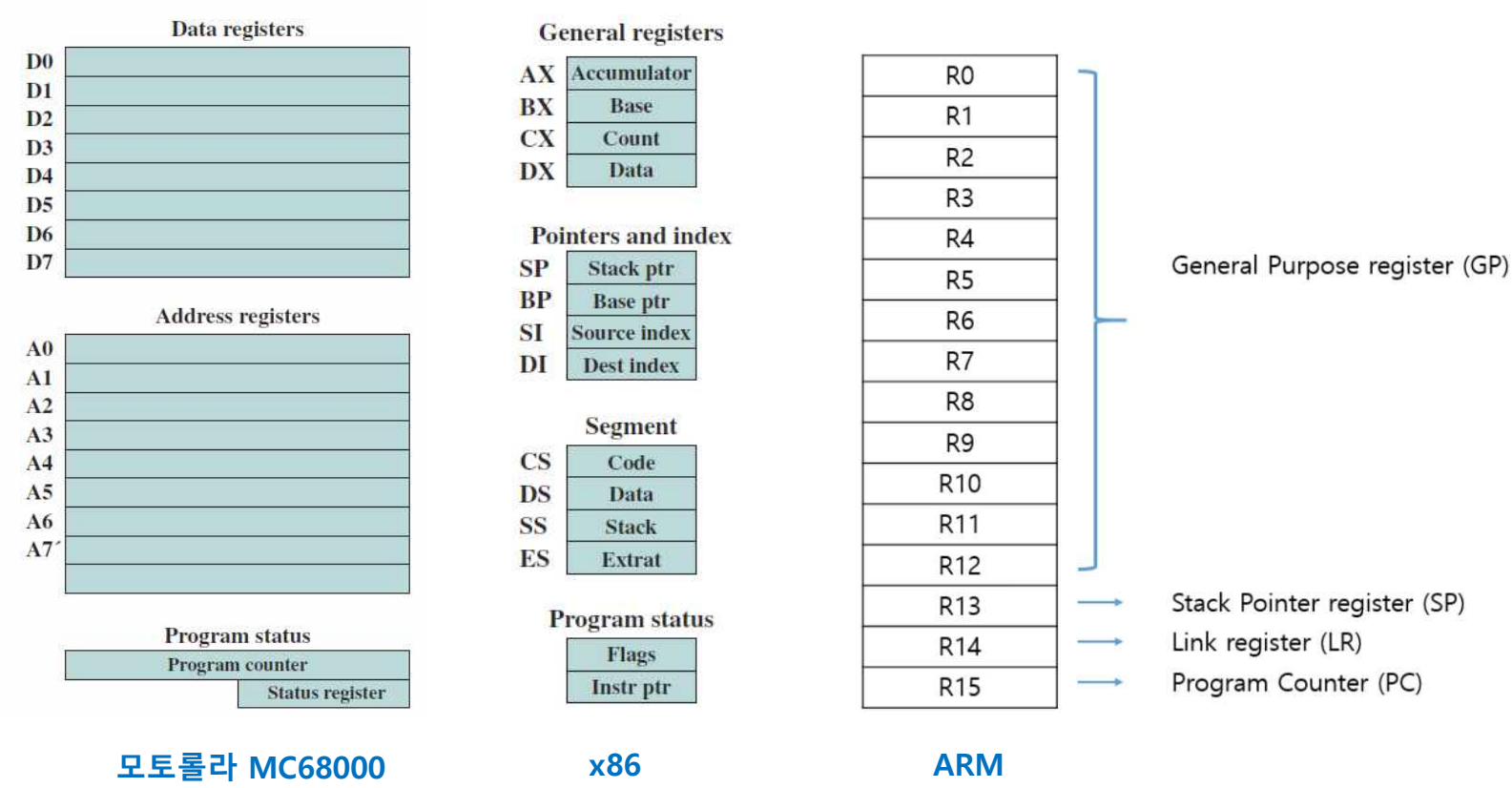
4. 레지스터 설계시 고려사항

- **레지스터 유형 (General vs. Specialized)**
  - 범용(General) : 프로그램의 융통성 (flexibility) 증가. 명령어 크기와 복잡도 증가
  - 특정용도(Specialized) : 명령어 단축가능, 실행속도 향상, 융통성 감소.

- **레지스터 수**
  - 보통 8 – 32
  - 레지스터 수가 적으면, 메모리 참조 횟수가 증가.
  - 많으면 메모리 참조는 줄일 수 있지만, 프로세서 비용증가.

- **레지스터 길이**
  - 어드레스와 워드 (word) 를 저장할 수 있을 정도로 충분하게 커야 함.
  - 여러 개의 레지스터를 연결해서 큰 데이터 저장에 사용할 수 있음.

5. 레지스터 구조 사례

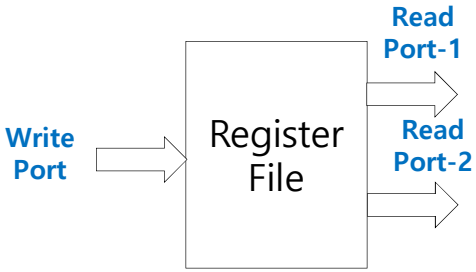


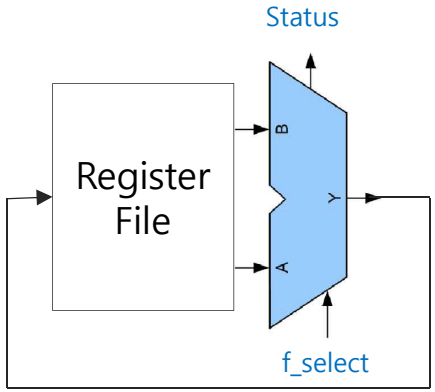
모토롤라 MC68000 x86 ARM

레지스터 파일

1. 레지스터 파일 개요

- **레지스터 파일**
  - 배열 형태로 배치된 레지스터 집합
  - 일반적으로, 다중 포트(Multi-port)를 가진 SRAM으로 구현.





1. 레지스터 파일 개요

- **레지스터 파일을 사용한 성능향상**
  - 자주 사용되는 오퍼랜드를 레지스터에 할당하여 메모리 접근 빈도를 줄인다.

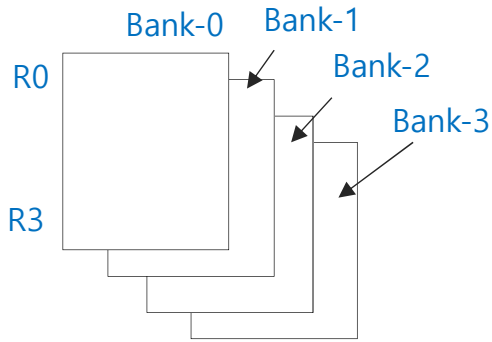
- **소프트웨어적 해결방법**
  - 컴파일러에 의한 레지스터 할당.
    - 자주 사용되는 변수를 레지스터로 할당.
    - 복잡한 프로그램 분석작업이 필요

- **하드웨어적 해결방법**
  - 많은 변수들이 레지스터에 할당될 수 있도록 많은 수의 레지스터를 사용

2. 레지스터 뱅킹

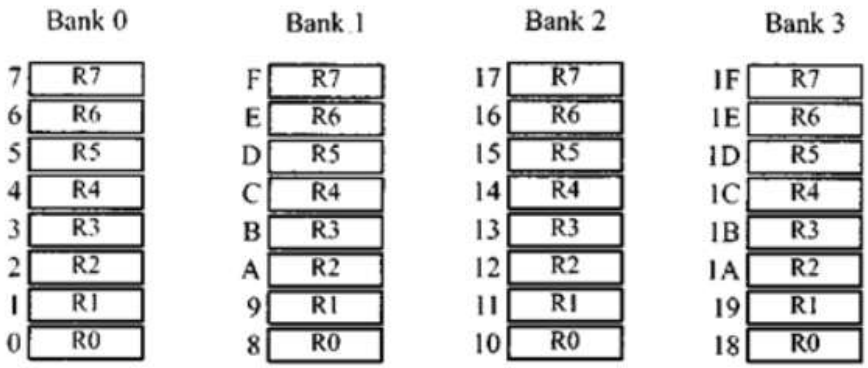
- **레지스터 뱅킹**
  - 레지스터를 그룹으로 나누어서 사용
    - 동일한 이름의 레지스터를 여러 개 보유
    - 동시에 사용할 수 없음

- **레지스터 뱅킹을 사용함으로써, 함수 호출/복귀 또는 인터럽트 처리, CPU 동작모드 변경시 발생하는 Context Switching 부담을 감소시켜, 시스템 성능을 향상시킬 수 있다.**
  - 메모리 스택을 사용한 Context Switching 작업을 수행하는 대신, 간단히 레지스터 뱅킹만 실시.



2.1 8051 레지스터 뱅킹

- **8051 레지스터 뱅킹**
  - 32개의 8-bit 레지스터를 4개의 뱅크로 나누어서 사용



usr	sys	svc	abt	und	irq	fiq
R0						
R1						
R2						
R3						
R4						
R5						
R6						
R7						
R8						R8_fiq
R9						R9_fiq
R10						R10_fiq
R11						R11_fiq
R12						R12_fiq
R13		R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14		R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
R15						
CPSR						
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

2.2 ARM 레지스터 뱅킹

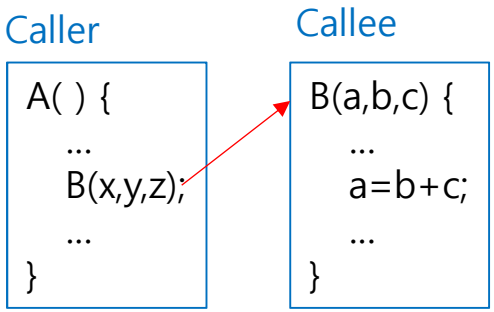
- **ARM 프로세서의 레지스터 뱅킹**
  - 32개의 32-bit 레지스터중 일부를 CPU 동작 모드에 따라 다양하게 뱅킹
    - R13 : Stack Pointer
    - R14 : Link Register
    - R15 : Program Counter

- **Link Register**
  - 함수 호출시 복귀 어드레스를 저장하는 전용 레지스터
    - LR ← PC
  - 복귀 어드레스를 스택에 저장하지 않고 LR에 복사 : 메모리 참조 불필요 → 처리속도 향상

usr	sys	svc	abt	und	irq	fiq
R0						
R1						
R2						
R3						
R4						
R5						
R6						
R7						
R8						R8_fiq
R9						R9_fiq
R10						R10_fiq
R11						R11_fiq
R12						R12_fiq
R13		R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14		R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
R15						
CPSR						
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

2.3 컨텍스트 전환

- **컨텍스트 전환(Context Switching)**
  - CPU 동작 모드의 변환
    - 사용자 모드 → 운영체제 모드 전환, 운영체제 모드 → 사용자 모드 전환
  - Caller 함수에서 Callee 함수 호출
  - 인터럽트 처리



2.4 컨텍스트 전환 부담

- **컨텍스트 전환 부담(Overhead).**
  - 컨텍스트 전환시 수행해야 할 작업
    - 복귀(Return) 어드레스 저장
    - 상태 레지스터 저장
    - 사용중인 레지스터 저장
    - 파라미터 전달
  - 저장장소 : 스택(Stack)

- **컨텍스 전환 부담을 최소화하여 컴퓨터의 성능향상**

2.5 컨텍스트 전환 빈도

- **컨텍스트 전환 빈도**
  - Patterson 분석 결과
    - 함수 호출 12% 정도이지만, 메모리 참조는 45%

	Dynamic Occurrence		Machine-Instruction Weighted		Memory-Reference Weighted	
	Pascal	C	Pascal	C	Pascal	C
ASSIGN	45%	38%	13%	13%	14%	15%
LOOP	5%	3%	42%	32%	33%	26%
CALL	15%	12%	31%	33%	44%	45%
IF	29%	43%	11%	21%	7%	13%
GOTO	–	3%	–	–	–	–
OTHER	6%	1%	3%	1%	2%	1%

레지스터 윈도우

1. 레지스터 윈도우 개요

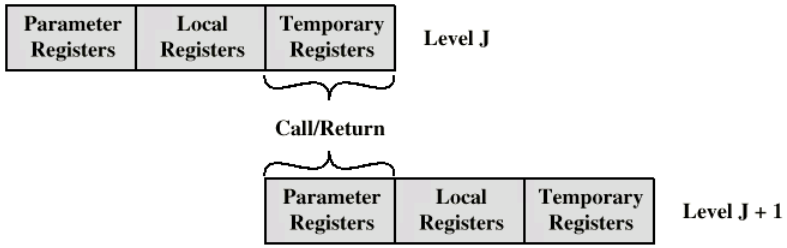
- **레지스터 윈도우(Register Window) 사용 배경**
  - 함수의 파라미터 수와 지역 변수의 갯수가 적다.
  - 함수의 연속호출(Nesting) 정도가 작다.

- **레지스터 윈도우를 정의해서 파라미터와 지역변수를 메모리에 저장하지 않고 레지스터만을 사용해서 성능을 향상시켜 보자.**

- **레지스터 윈도우**
  - 소규모의 레지스터 세트를 여러 개 사용.
  - 함수 호출시 마다 다른 레지스터 세트를 사용
    - 레지스터를 메모리에 저장했다고 복구하는 작업 불필요.
  - 함수 복귀시 이전 레지스터 세트로 전환.

2. 레지스터 윈도우 구성

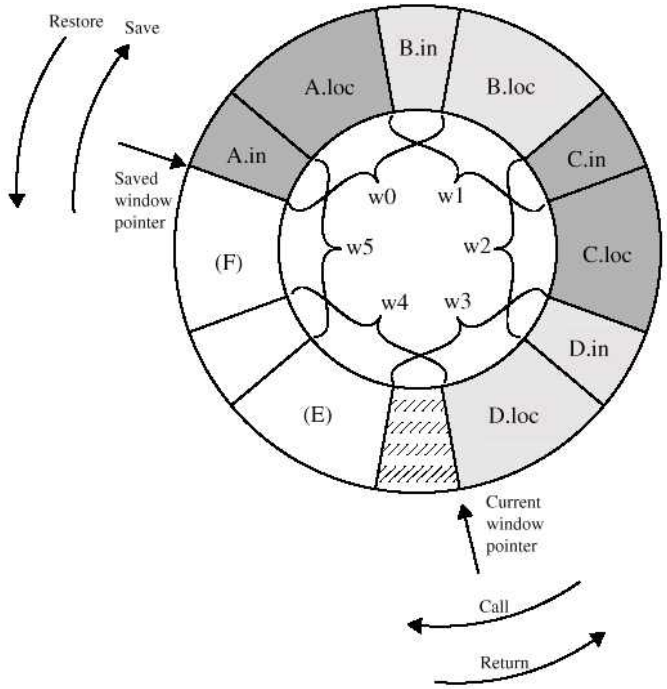
- **레지스터 윈도우 구성**
  - 파라미터 레지스터 (Parameter registers)
    - 함수 호출시 파라미터 저장 및 복귀시 결과 값 저장
  - 지역 레지스터 (Local registers)
    - 지역 변수에 할당
  - 임시 레지스터 (Temporary registers)
    - 함수내에서 다른 함수 호출시 전달할 파라미터와 결과 값 저장.



- **윈도우 중첩 (Overlap)**
  - 현재 레지스터 윈도우의 임시 레지스터와 다음 함수의 파라미터 레지스터의 중첩
    - 데이터 이동없이 파라미터 전달이 가능

3. 순환 버퍼를 사용한 레지스터 윈도우

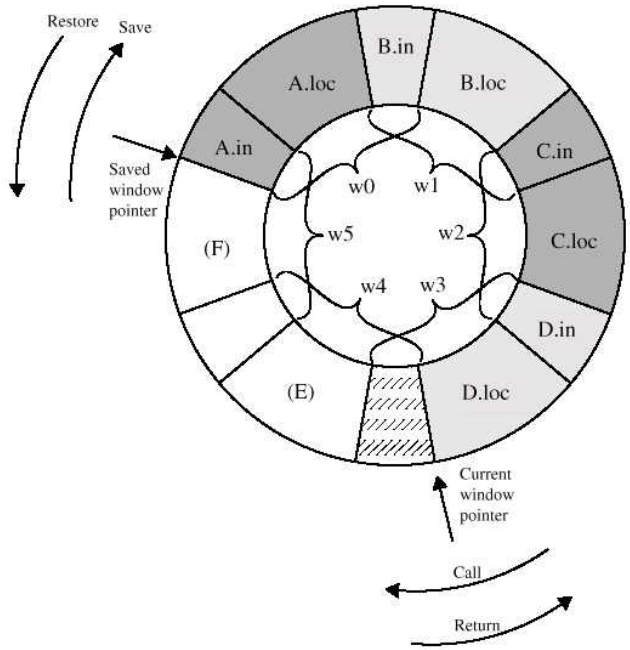
- **순환 버퍼(Circular-buffer)를 사용한 레지스터 윈도우 구성**
  - 레지스터 윈도우를 중첩시켜, circular-buffer 형태로 구현.
  - CWP (current window pointer)
    - 현재 실행하고 있는 함수의 레지스터 윈도우를 포인트한다.
  - SWP (saved window pointer)
    - 가장 최근에 메모리에 저장된 윈도를 포인트한다.
  - N-윈도우 레지스터 파일은 N-1 개의 함수를 지원



3. 순환 버퍼를 사용한 레지스터 윈도우

- **중첩 윈도우의 순환 버퍼 다이어그램**
  - A 함수가 B 호출 → B 가 C 호출 → C 가 D 호출

- **순환 버퍼가 Full 상태가 되면, 인터럽트 발생시켜 가장 오래된 윈도우(A 함수)를 메모리 스택에 저장**



4. 레지스터 윈도우 적용 사례

- **적용 사례**
  - 8개의 윈도우 버퍼를 사용한 경우, 1% 정도만이 메모리 사용.

- **레지스터 윈도우를 사용하는 대표적 프로세서**
  - Sun's SPARC architecture : 8 레지스터의 윈도우 4 set 사용
  - Berkley RISC : 16 레지스터의 윈도우 8 set 사용
  - Pyramid Technology's computer : 32개 레지스터를 가진 레지스터 윈도우 16 set 사용
  - AMD 29000 : Berkley RISC 기반
    - 가변 크기의 윈도우 사용
    - 64 개의 global 레지스터
    - 128 개의 윈도우용 레지스터

5. Global Register

- **전역 변수(Global Variable) 처리 방법**
  - 컴파일러가 메모리의 특정영역을 전역변수로 할당하여 사용
  - 전역변수 전용 레지스터(Global Register) 사용

- **Global 레지스터**
  - 자주 사용하는 global variable 을 전용 레지스터에 할당.
    - 레지스터 어드레싱을 분리하는 부담.
    - 컴파일러와 링커에서 전역변수를 식별하여, 레지스터에 할당해야 하는 부담.

레지스터 최적화

1. 컴파일러기반 레지스터 최적화

- **레지스터 최적화는 컴파일러에서 실시**
  - 가능한 많은 연산을 레지스터상에서 실시할 수 있도록 최적화
  - 가능한 load-store 동작을 최소화할 수 있도록 최적화

- **최적화 절차**
  - 프로그램을 분석해서, 레지스터에 할당할 수 있는 후보 변수를 파악.
  - 각 후보변수에 가상 레지스터(수량 제한없음)를 할당.
  - 가상 레지스터를 실제 레지스터에 매핑.
  - 사용기간이 중복되지 않는 레지스터는 같은 레지스터를 공유.
  - 사용할 수 있는 레지스터가 부족한 경우에는 나머지 변수를 메모리에 할당.

- **레지스터 최적화 과정을 graph coloring 문제로 모델링.**

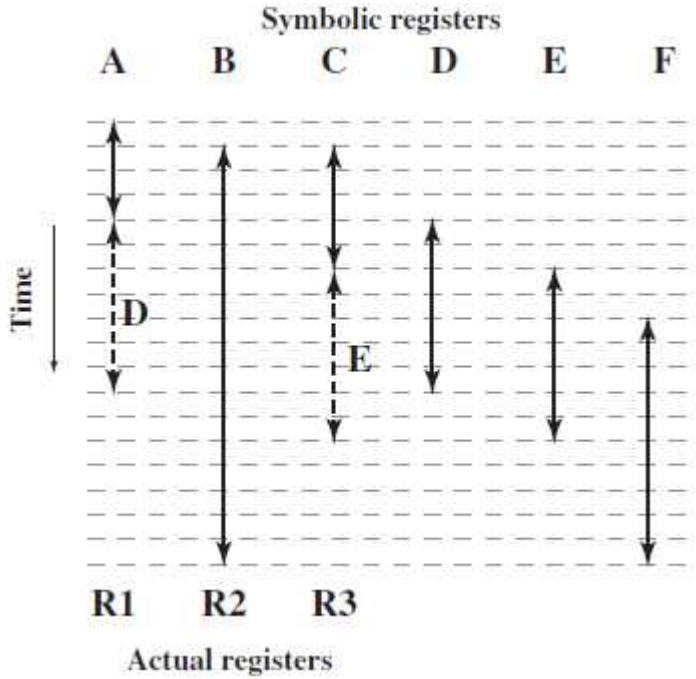
2. Graph Coloring 문제

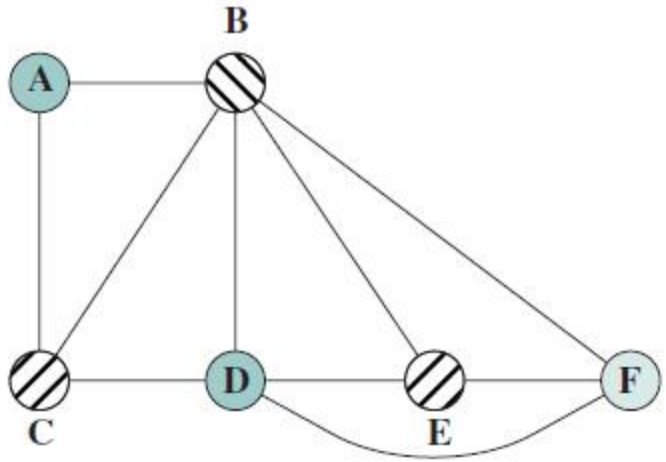
- **Graph coloring 문제**
  - Node와 edge로 구성된 그래프에서 이웃하는 노드가 서로 다른 컬러가 되도록 색칠할 때, 사용해야 하는 컬러의 최소 수를 찾는 문제.

- **레지스터 최적화 문제를 graph coloring 문제로 모델링하는 절차**
  - Graph의 각 node를 가상 레지스터로 표시.
  - 프로그램 실행중 같은 시점에 사용되는 2개의 레지스터들은 edge로 연결.
  - Graph coloring 알고리즘을 적용하여 최소 컬러 수 결정.
    - 최소 컬러 수 = 실제 사용해야 하는 레지스터 수
    - 컬러를 칠할 수 없는 노드 = 메모리에 할당.

3. Graph Coloring 예시

- **Graph coloring 예시**





4 Colors

4. 효율적 레지스터 사용

- **레지스터 설계상의 절충 (Trade-off) 요소**
  - 많은 레지스터를 사용
  - 컴파일러 기반 최적화

- **레지스터 수에 대한 최적화**
  - 간단한 최적화 알고리즘을 적용 했을 경우, 64개 이상의 레지스터를 사용하는 경우, 얻는 성능향상은 미미.
  - 복잡한 최적화 알고리즘을 적용해도, 32 개 이상의 레지스터를 사용하는 경우, 성능향상은 미미.
  - 사용하는 레지스터의 수가 적을 경우, 모든 레지스터를 범용으로 사용하는 것이 범용과 전용으로 분리해서 사용하는 것보다 더 효과적.

end