제3장
어셈블리 언어 (2)

Youtube 주소
[3-1] http://youtu.be/MzmgiTSd8_c
[3-2] http://youtu.be/dKORKv8CiuQ
[3-3] http://youtu.be/wfsyi4utkf8

분기 조건의 설정

- 프로그램의 분기
  - ✓직전의 실행결과에 따라서 다음 작업을 선택해야 할 때
  - ✓(예) C 나 Java 언어의 if 문, while 문 등에 필요
- 분기조건 (CCR의 조건 비트들 활용)
  - ✓'직전의 연산결과가 음수이면' → 'CCR의 N 비트가 1이면'
  - ✓'직전의 연산결과가 영이면' → 'CCR의 Z 비트가 1이면'
  - ✓…
  - ✓일반프로세서에서는 C 비트, V 비트도 활용함

	b_3	b_2	b_1	b_0
CCR	C	V	N	Z

CCR 의 비트 설정

- 명령어에 따라 CCR 비트들의 갱신여부가 다름 (표 2.5)
  - ✓산술/논리 연산 관련 명령어 들은 CCR 갱신
  - ✓그 외 명령어들은 CCR 값이 변하지 않음
- N (Negative)
  - ✓연산결과 최상위 비트 (정수에서 음수를 나타내는 비트)와 동일하게 설정
- Z (Zero)
  - ✓연산결과의 모든 비트가 0이면 1, 아니면 0
- C (Carry)
  - ✓연산과정에서 캐리가 발생하면 1, 아니면 0
  - ✓TOY는 지원하지 않음 ← 정수형만 사용함
- V (oVerflow)
  - ✓연산 과정에서 오버플로가 발생하면 1, 아니면 0
  - ✓TOY는 지원하지 않음 ← 프로그램 작성자 몫

분기 명령어 BR (Branch)

- 문법: BR 조건, addr
  - ✓조건: BR 명령어의 조건으로서 n,z,p 의 일부 또는 모두를 지정가능
- 사용 예: BR nz, 4000
  - → 이전 실행결과가 음수이거나 영이면 4000 번지로 실행위치 이동
  - →CCR의 N 비트가 1이거나 Z 비트가 1이면 …
- (주) 4000번지로 실행위치 이동은
  - ✓PC 에 4000 이 저장되는 것임

분기 명령어의 활용

- 조건문 프로그램
  - ✓(예) 변수 x의 절대값 구하기

```
if (x < 0)
   x = -x;
...
```

- BR (Branch) 명령어를 이용한 조건문 처리

```
   ...
   LOAD R1, x      ; 변수 x의 값을 R1로 읽어 들임
   CMP R1, 0       ; 0과 비교하여
   BR zp, next     ; 그 결과가 0이거나 양이면 next 위치로 건너 뜀
   COPY R2, 0      ; R2의 값을 0으로 설정
   SUB R1, R2, R1  ; R2의 값 0에다 R1의 값을 빼서 반대부호로
   STORE R1, x     ; 결과인 R1 값을 변수 x에 저장
next:
   ...
x: .FILL -23       ; 지정된 임의의 값
```


조건 처리 예

- if (x < 7) ➔ if (x-7 < 0)
  - ✓'x - 7' 을 먼저 실행한 후에 'N 비트가 1이면'
- if (x == 7) ➔ if (x-7 == 0)
  - ✓'x - 7'을 먼저 실행한 후에 'Z 비트가 1이면'
- if (x > 7) ➔ if (x-7 > 0)
  - ✓'x-7'을 먼저 실행한 후에 ???
- if (x <= 7) ➔ if (x-7 <= 0)
  - ✓'x - 7'을 먼저 실행한 후에 ???
- if (x >= 7) ➔ if (x-7 >= 0)
  - ✓'x - 7'을 먼저 실행한 후에 ???

분기조건 표시

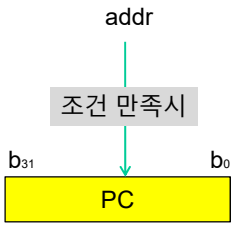
조건	조건 표시	CCR의 N,Z 비트 조건 (회로설계에 반영)
작다	**n**	N=1 이면 (이때 Z=0임)
같다	**z**	Z=1 이면 (이때 N=0임)
크다	**p**	N과 Z 모두 0이면
작거나 같다	**nz**	n과 z중 하나에 해당하면
같지 않다	**np**	n과 p중 하나에 해당하면
크거나 같다	**zp**	z과 p중 하나에 해당하면
무조건	**nzp**	n, z, p중 하나에 해당하면 (항상 참)

㈜ TOY에서는 단순한 회로로 설계하기 위해서 C와 V를 제외함
(표 11.3 참조: ARM 프로세서 에서 C와 V의 사용)

분기 명령어의 개념적 실행

- 조건이 만족되면 PC 레지스터에 대상 주소 (addr)를 기록

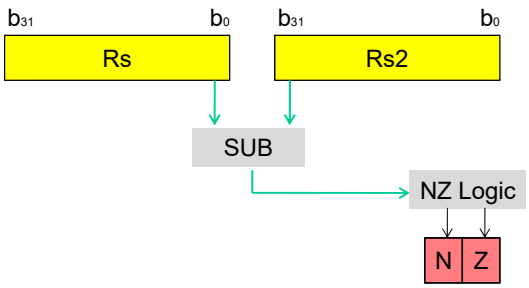
→ 다음 명령어는 메모리 주소 addr에 있는 명령어를 읽어들임



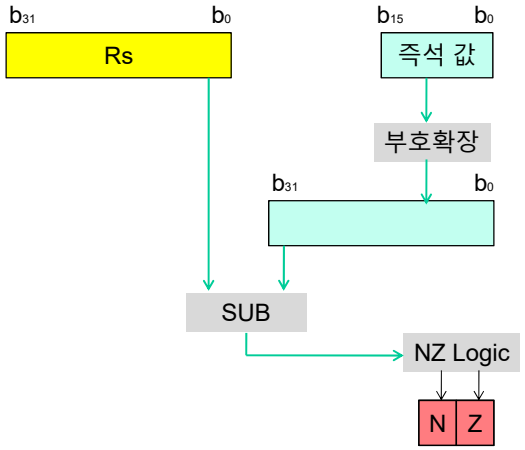
비교 명령어 CMP (Compare)

- 문법: CMP Rs, Rs2
  - ✓Rs – Rs2 의 계산 과정에서 CCR 비트들을 갱신
  - ✓계산 결과는 저장하지 않음
- 사용 예: CMP R1, 99
  - ✓현재의 R1의 값에서 99를 빼는 과정에서 CCR 비트들만 적절히 갱신

레지스터 간의 비교 연산



레지스터와 즉석 값의 비교연산



CMP를 이용한 조건문 처리 예

- 조건문 프로그램 y ← min(x, 15)

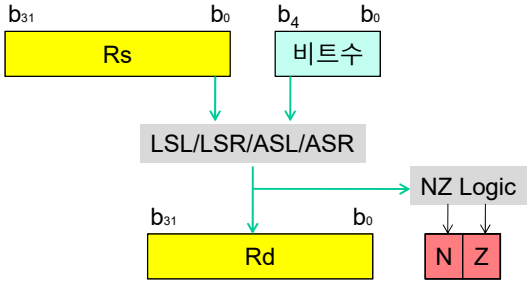
```
if (x >= 15)
   y = 15;
else
   y = x;
...
```

- BR (Branch) 명령어를 이용한 조건문 처리

```
   LOAD R1, x
   CMP R1, 15
   BR n, else    ; 음수이면
   COPY R2, 15
   STORE R2, y
   BR nzp, addr  ; 무조건
else:
   COPY R2, R1
   STORE R2, y
addr:
   ...
```

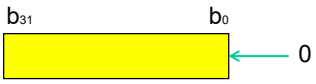
시프트 연산 명령어

- 시프트 연산: Rs의 비트들을 지정된 비트 수 만큼 왼쪽이나 오른쪽으로 이동
  - ✓LSL (Logical Shift Left)
  - ✓LSR (Logical Shift Right)
  - ✓ASL (Arithmetic Shift Left)
  - ✓ASR (Arithmetic Shift Right)

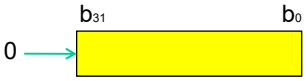


시프트 동작

- LSL (Logical Shift Left)



- LSR (Logical Shift Right)



- ASL (Arithmetic Shift Left): LSL과 동일함
- ASR (Arithmetic Shift Right): → 정수에 대해 부호가 바뀌지 않음

시프트 연산과 곱셈/나눗셈 효과

명령어	레지스터 값	비부호 정수값
실행전 ①	R1: **00**00 0000 ... 0000 1100	12
LSL R2,R1,2	R2: 0000 0000 ... 0011 00**00**	48
LSR R2,R1,2	R2: **00**00 0000 ... 0000 0011	3
실행전 ②	R1: **01**00 0000 ... 0000 0000	2^{30}
LSL R2,R1,2	R2: 0000 0000 ... 0000 00**00**	0(오버플로우)
LSR R2,R1,2	R2: **00**01 0000 ... 0000 0000	2^{28}

시프트 연산 결과의 비교

명령어	레지스터 값	정수값
실행 전 ①	R1: **0**000 0000 ... 0000 1100	12
ASR R2,R1,2	R2: **000**0 0000 ... 0000 0011	3
ASL R2,R1,2	R2: 0000 0000 ... 0011 00**00**	48
실행 전 ②	R1: **1**111 1111 ... 1111 1100	-4
ASR R2,R1,2	R2: **111**1 1111 ... 1111 1111	-1
ASL R2,R1,2	R2: 1111 1111 ... 1111 00**00**	-16
실행 전 ③	R1: **0**010 0000 ... 0000 0000	2^{29}
ASR R2,R1,2	R2: **000**0 1000 ... 0000 0000	2^{27}
ASL R2,R1,2	R2: 1000 0000 ... 0000 00**00**	-2^{31}(오버플로)

C 언어와 시프트 연산

C 언어 표현	어셈블리 언어 표현
`int x;` `unsigned y;`	`x: .BLOCK 1` `y: .BLOCK 1`

`x = (x >> 1);`	`LOAD R0, x`
	`ASR R0, R0, 1`
	`STORE R0, x`

`y = (y >> 2);`	`LOAD R0, y`
	`LSR R0, R0, 2`
	`STORE R0, y`

`x = (x << 3);`	`LOAD R0, x`
	`ASL R0, R0, 3; or LSL`
	`STORE R0, x`

`y = (y << 4);`	`LOAD R0, y`
	`LSL R0, R0, 4`
	`STORE R0, y`

시프트 연산을 활용한 곱하기 10 의 처리

- value x 10 = value x (8 + 2) = (value x 8) + (value x 2)

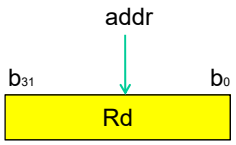
```
    .ORIGIN 0x2000
    LOAD R1, value
    LSL R2, R1, 3     ; R2 ← R1 x 8
    LSL R3, R1, 1     ; R3 ← R1 x 2
    ADD R1, R2, R3    ; R1 ← (R1 x 8) + (R1 x 2) = R1 x 10
    STORE R1, result
halt: BR nzp, halt
value: .FILL 123456
result: .BLOCK 1
```

정수를 8로 나눈 몫과 나머지 구하기

```
    .ORIGIN 0x2000
    LOAD R1, value
    ASR R2, R1, 3      ; R2 ← R1 / 8
    STORE R2, quot
    AND R2, R1, 0x07   ; R2 ← R1 % 8
    STORE R2, remain
halt:
    BR nzp, halt
value: .FILL 0x12345
quot: .BLOCK 1
remain: .BLOCK 1
```


유효주소 저장 명령어

- LEA (Load Effective Address)
- 특정 주소를 저장해 놓고 이후에 계속 사용할 때 편리
- 특정 구조체나 배열의 각 항목을 차례대로 처리할 때 유용함
- 사용 예: LEA R1, addr



LEA와 LOAD 의 비교

```
        .ORIGIN 0x2000
        LOAD R1, value   ; value 부분의 값을 R1에 읽어들임
        LEA R2, value    ; value에 해당하는 주소를 R2에 저장
halt:   BR nzp, halt
value:  .FILL 0x1234
```

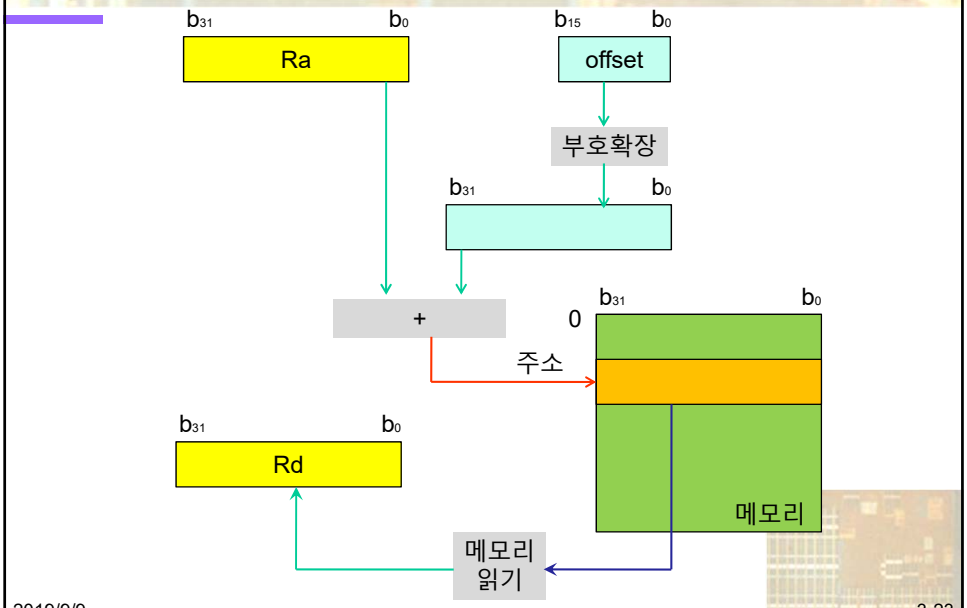
- 실행 후에
  - R1 = ?
  - R2 = ?

레지스터 간접 어드레싱 모드

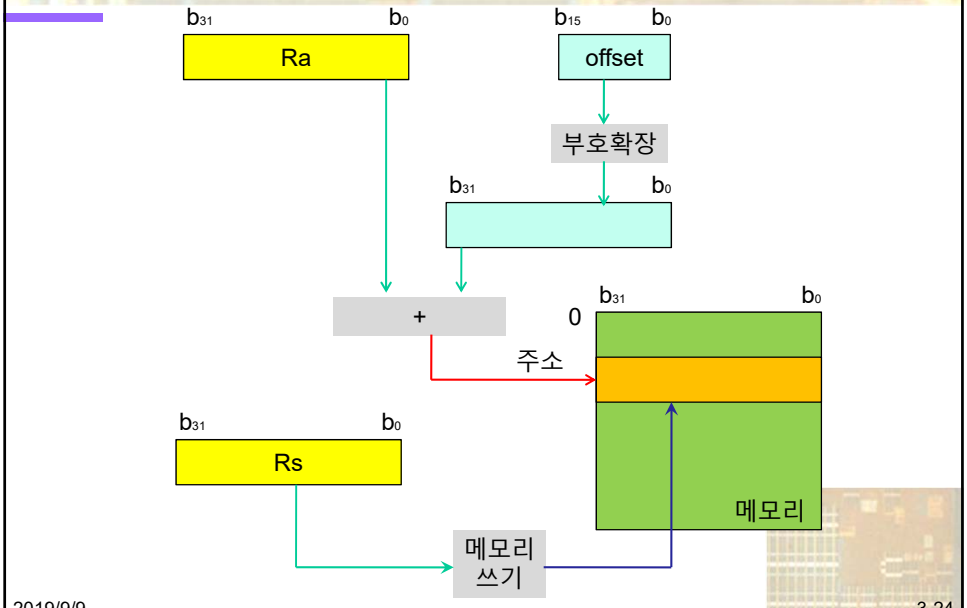
- 지정한 레지스터의 값에 offset 을 더한 것을 메모리 주소로 사용
  - LDR R1, R0, 3 ; R1 ← M[R0+3]
- (비교) 직접 어드레싱 모드는 메모리 주소를 수치나 레이블로 직접 지정 (메모리 주소가 기계어 코드에 들어 있음)
  - LOAD R1, 5000 ; R1 ← M[5000]

이름	어셈블리언어 표현	의미
Load Register Indirect	`LDR Rd, Ra, offset`	Rd ← 메모리의 Ra + offset 번지 내용
Store Register Indirect	`STR Rs, Ra, offset`	메모리의 Ra + offset 번지 내용 ← Rs
Branch Register Indirect	`BRR nzp, Ra, offset`	조건에 따라 Ra + offset 번지로 다음 실행위치를 이동

LDR 명령어의 실행



STR 명령어의 실행



레지스터 간접 어드레싱 모드의 활용

- **목적하는 주소가 고정되지 않은 경우**:
- 배열의 처리
  - ✓배열 A의 i번째 변수 접근 (예: x = A[i];)
  - ✓A의 주소에 i값을 더한 값을 레지스터에 저장 후 LDR이나 STR 실행
- 지역변수 접근
  - ✓지역변수는 스택에 할당되므로 스택 부분의 주소를 저장하는 레지스터를 기준으로 LDR이나 STR 실행
- 포인터의 처리
- **직접 어드레싱에서 주소표현이 불가능한 경우**
  - ✓주소는 현재 주소로부터 16비트로 표현 가능한 범위 내만 가능
  - ✓그 이상이면 그 주소를 레지스터에 LOAD 후에 LDR이나 STR로 처리

프로그램 예: 배열에 26개의 ASCII 문자 넣기

```
// C나 Java 표현

char table[26];
int i;
char ascii = 'A';
...
i = 0;
while (i < 26) {
    table[i] = ascii;
    i++;
    ascii++;
}
```

배열에 26개의 ASCII 문자 넣기 (STR, LEA 활용)

```
    .ORIGIN 0x2000

    LEA R1, table     ; 배열 table 의 주소
    COPY R2, 'A'      ; (0x41)
    COPY R0, 26       ; 반복할 회수
loop: STR R2, R1, 0 ; R2의 문자를 R1이 가리키는 주소에 기록
    ADD R1, R1, 1
    ADD R2, R2, 1
    SUB R0, R0, 1     ; 회수를 1만큼 감소
    BR p, loop        ; 남은 회수가 양수이면

halt: BR nzp, halt

table: .BLOCK 26
```

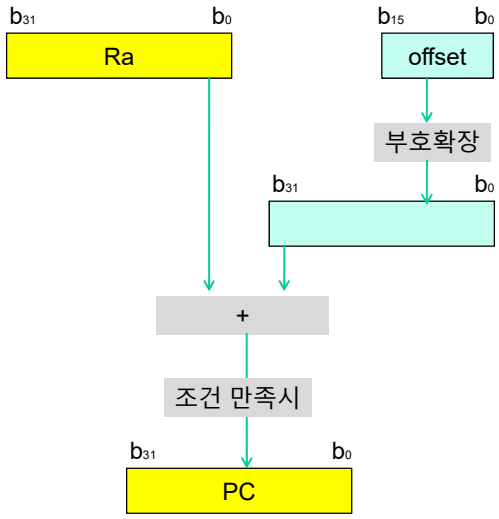

(참고) C 의 포인터와 레지스터 간접 어드레싱 모드

C 언어 표현	어셈블리 언어 표현	의미

`int x;`
`int y;`
`int* p;`
	`x: .BLOCK 1`
	`y: .BLOCK 1`
	`p: .BLOCK 1`
		변수들의 선언 *(어셈블리 언어에서는 메모리 확보에 대한 것만 있을 뿐이지 데이터 유형의 구별이 없음)*

`p = &x;`	**LEA** `R0, x` `STORE R0,` **p**	변수 x의 주소를 변수 p에 저장
`*p = 1234;`	`COPY R1, 1234` `LOAD R0,` **p** **STR** `R1, R0, 0`	변수 p에는 주소가 저장되어 있고, 이 주소에 해당하는 메모리에 값 1234를 저장
`y = *p;`	`LOAD R0,` **p** **LDR** `R1, R0, 0` `STORE R1, y`	변수 p에는 주소가 저장되어 있고, 이 주소에 해당하는 메모리의 내용을 읽어서 변수 y에 저장

BRR 명령어의 실행



(참고) 직접 어드레싱 모드의 지정 가능한 주소범위

- LOAD, STORE, BR 및 LEA는 특정 주소를 직접 지정 (주소나 레이블로 지정)
- 현재의 PC 값에서 16비트 정수로 표현 가능한 범위 이내만 가능
  - ✓PC – 16비트 최소 정수 ≤ addr ≤ PC + 16비트 최대 정수
  - ✓PC – 2^{15} ≤ addr ≤ PC + 2^{15} - 1
  - ← TOY 프로세서의 설계 제약: 기계어 코드에 offset을 표현하기 위한 영역이 16비트 밖에 없음
- Offset이 16비트 정수 범위를 벗어나면
  - ✓LDR, STR, 또는 BRR을 사용해야 함

문자 입출력 및 프로그램 종료

- 운영체제의 기능 호출
  - ✓파일 입출력, 다른 프로세스 생성, …
  - ✓키보드 입력, 화면 출력, …
- SWI (Software Interrupt) 명령어
  - ✓운영체제 기능을 호출할 때 사용 (SWI 명령어를 실행하면 운영체제 모드로 바뀜)
  - ✓특정 기능마다 지정된 번호로 호출
  - ✓운영체제 SW 설계자가 기능별로 번호 결정
- 적용 예
  - ✓SWI 0 : 키보드에서 문자 1자 읽기
  - ✓SWI 1 : 화면에 문자 1자 출력
  - ✓…

TOY 시뮬레이터의 가상적인 운영체제 기능

이름	어셈블리언어 표현	의미
문자 읽기	**SWI 0**	키보드에서 한 문자를 읽어서 R0에 기록
문자 쓰기	**SWI 1**	R0에 저장된 문자를 화면에 출력
문자열 읽기	**SWI 2**	키보드에서 공백으로 구분되는 단어를 읽어서 R0가 가리키는 메모리 주소부터 차례대로 저장; 문자열의 끝은 0(null) 로 처리
문자열 쓰기	**SWI 3**	R0가 가리키는 메모리 주소로부터 시작되는 문자열을 화면에 출력; 문자열의 끝은 0(null) 문자로 가정
에코모드 설정	**SWI 4**	키보드 입력시의 에코모드 설정: R0의 값이 1이면 에코 실행, R0의 값이 0이면 에코 없음 (기본 모드는 에코 있음)
프로그램 종료	**SWI 255**	실행중인 프로그램의 종료

예제: 단순한 369 게임

```
     .ORIGIN 0x2000
loop: SWI 0           ; 키보드 문자 읽기
     CMP R0, '3'
     BR z, match
     CMP R0, '6'
     BR z, match
     CMP R0, '9'
     BR np, print
match: COPY R0, '*'
print: SWI 1          ; 문자를 화면에 출력
     BR nzp, loop
```


예제: 데이터 합 구하기

```
; 0x50000 번지부터 1000개의 데이터를 더한 결과를 sum에 저장
; (참고) 0x50000 번지는 현재 주소로 부터 16비트 offset 초과
   .ORIGIN 0x2000

   COPY R0, 0         ; 합
   LOAD R1, addr      ; 데이터 주소 ('COPY R1, 0x50000' ??)
   COPY R2, 1000      ; 회수
loop: LDR R3, R1, 0
   ADD R0, R0, R3
   ADD R1, R1, 1
   SUB R2, R2, 1
   BR p, loop
   STORE R0, sum
   SWI 255            ; 프로그램 종료

addr:  .FILL 0x50000 ; 데이터가 있는 메모리 주소
sum:   .BLOCK 1
```