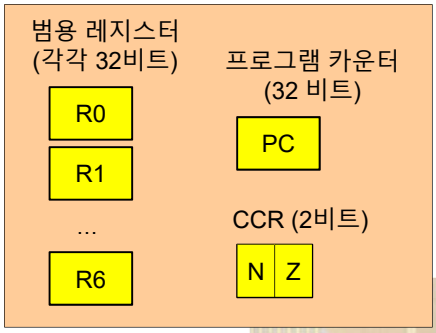
# 제2장
# 어셈블리 언어 (1)




## 프로그래밍 모델

- 프로세서 별로 정의된 명령어 집합 및 레지스터 집합
  - ✓명령어 별로 그 기능 및 처리 대상 데이터 등에 대한 정의
  - ✓레지스터 별로 그 용도 및 비트 수, 비트 별 의미 등에 대한 정의
  - ✓메모리의 주소범위, 예외처리 등 CPU의 동작과정에 관련된 내용
- TOY 프로세서 명령어 집합
  - ✓23개 명령어
  - ✓32비트 정수만 처리
- TOY 프로세서 레지스터 집합
  - ✓범용레지스터: R0~R6
  - ✓PC (R7)
  - ✓CCR: N (Negative), Z (Zero)
- TOY 프로세서 메모리 모델
  - ✓32비트 주소
  - ✓주소당 32비트 데이터



# TOY의 CCR 레지스터

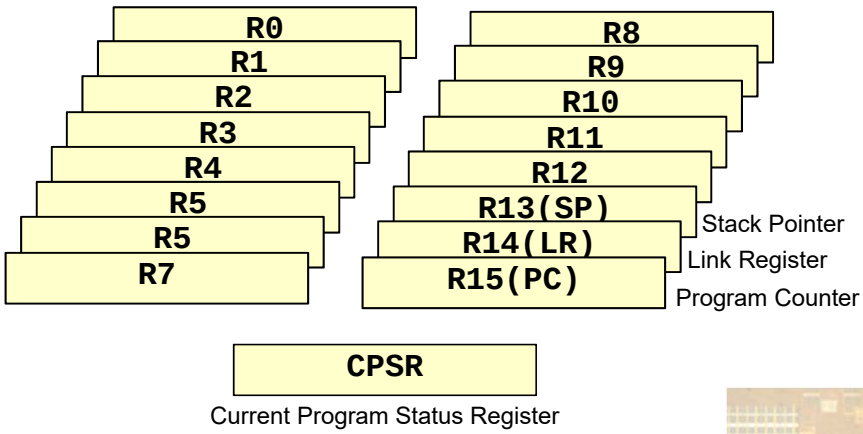
- Condition Code Register
  - ✓N 비트: 직전의 연산 결과가 음수이면 1, 아니면 0
  - ✓Z 비트: 직전의 연산 결과가 영이면 1, 아니면 0
  - ✓(주) Carry 와 Overflow 비트는 회로의 단순화를 위해 제외함

- (참고) 프로세서에 따라서는
  - ✓S (Sign), Z (Zero)로 표기하기도 하고, C (Carry), V (oVerflow)도 사용
  - ✓레지스터 이름을 SR (Status Register), CPSR (Current Program Status Register), FLAGS 등을 사용

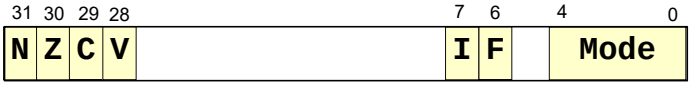




# (참고) ARM 레지스터 집합 (모두 32비트)



# (참고) ARM CPSR의 비트 별 정의




I: Disable Interrupt
F: Disable Fast Interrupt

N: Negative
Z: Zero
C: Carry
V: Overflow

Mode
10000: User
10001: Fast Interrupt
10010: Interrupt
10011: Software Interrupt
11111: System





# TOY 프로세서의 단순 명령어 집합

| 이름 | 어셈블리언어 표현 | 의미 | CCR 갱신 |
|---|---|---|---|
| Add | **ADD** Rd, Rs, Rs2 | Rd ← Rs + Rs2 | O |
| Subtract | **SUB** Rd, Rs, Rs2 | Rd ← Rs - Rs2 | O |
| Compare | **CMP** Rs, Rs2 | Rs - Rs2 | O |
| And | **AND** Rd, Rs, Rs2 | Rd ← Rs AND Rs2 | O |
| Or | **OR** Rd, Rs, Rs2 | Rd ← Rs OR Rs2 | O |
| Exclusive Or | **XOR** Rd, Rs, Rs2 | Rd ← Rs XOR Rs2 | O |
| Not | **NOT** Rd, Rs | Rd ← NOT Rs | O |
| Copy | **COPY** Rd, Rs2 | Rd ← Rs2 | O |
| Load | **LOAD** Rd, addr | Rd ← 메모리 addr 번지의 내용 | X |
| Store | **STORE** Rs, addr | 메모리 addr 번지의 내용 ← Rs | X |

(주) - Rd, Rs : R0부터 R6까지의 레지스터를 지정
- Rs2 : R0부터 R6까지의 레지스터나 16비트로 표현이 가능한 정수

# 명령어의 구성

- 명령어 종류 : 실행 동작을 정의
  - ✓ADD, LOAD, STORE, …
- Operand : 명령어의 처리 대상인 데이터
  - ✓'ADD R3, R1, 9' 의 명령어에서 R1, R3, 9 등
  - ✓여기서 R3는 목적지 (destination) 오퍼랜드, R1과 9는 소스 (source) 오퍼랜드
  - ✓오퍼랜드는 레지스터 일수도 있고, 값일 수도 있고, 메모리 주소(의 메모리 내용)일 수도 있음
- Addressing Mode : 오퍼랜드를 지정하는 방법
  - ✓레지스터 모드 (register mode)
  - ✓즉석 모드 (immediate mode): 명령어 자체에 값이 표현되어 있음
  - ✓직접 모드 (direct mode)
  - ✓레지스터 간접 모드 (register indirect mode)

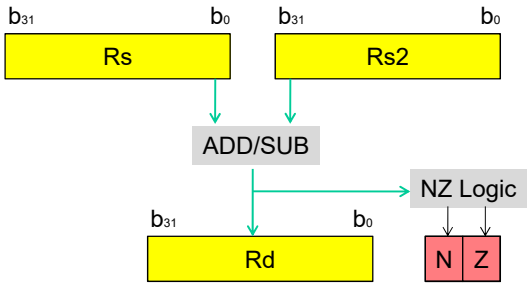




# 산술연산 명령어

- 문법
  - ✓ADD Rd, Rs, Rs2 의미: Rd ← Rs + Rs2
  - ✓SUB Rd, Rs, Rs2 의미: Rd ← Rs – Rs2
  - ✓Rd: destination register, Rs: source register
  - ✓Rs2 는 레지스터 일수도 있고 즉석 값일 수도 있음
  - ✓즉석 값은 16비트로 표현 가능한 정수 ← *기계어 표현의 제약 때문*
- 사용 예
  - ✓ADD R2, R0, R1
  - ✓ADD R3, R1, 9
  - ✓SUB R1, R2, R3
  - ✓SUB R2, R0, 9

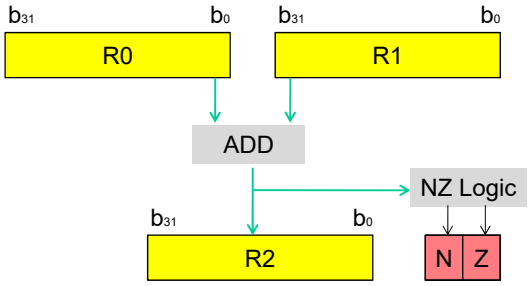
# 레지스터 간의 산술연산








# ADD R2, R0, R1



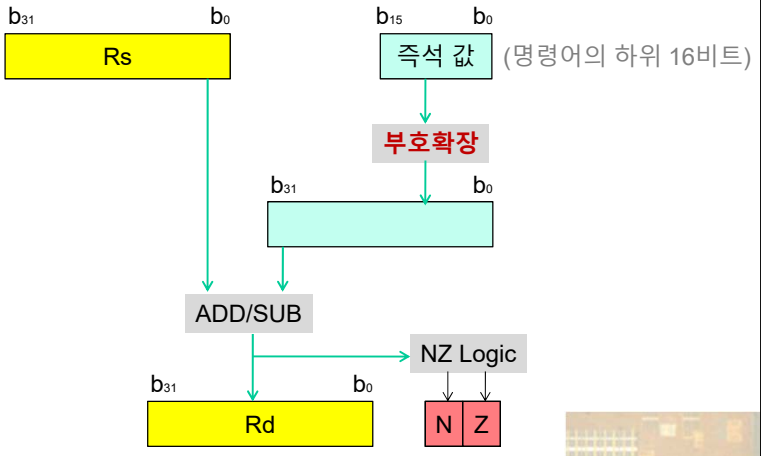
# ‘ADD R2, R0, R1’의 실행에서 CCR의 갱신

| 실행전의 값 | | 실행후의 값 | | |
|---|---|---|---|---|
| R0 | R1 | R2 | CCR | |
| | | | N | Z |
| 5 | 3 | 8 | 0 | 0 |
| 5 | -5 | 0 | 0 | 1 |
| 5 | -7 | -2 | 1 | 0 |

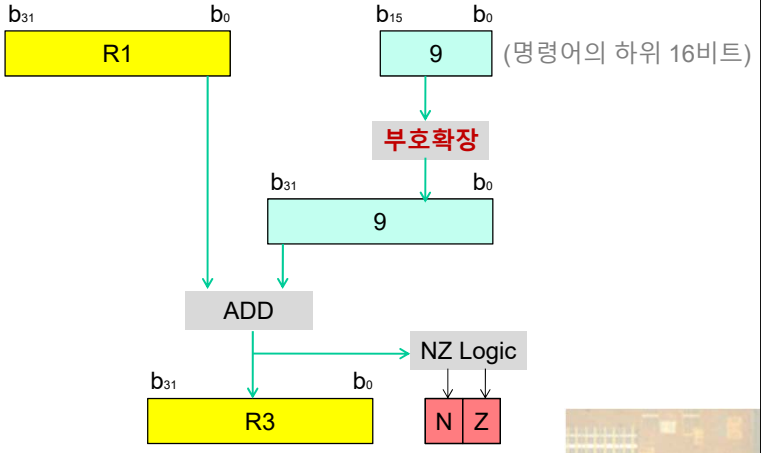




# 레지스터와 즉석 값의 산술연산



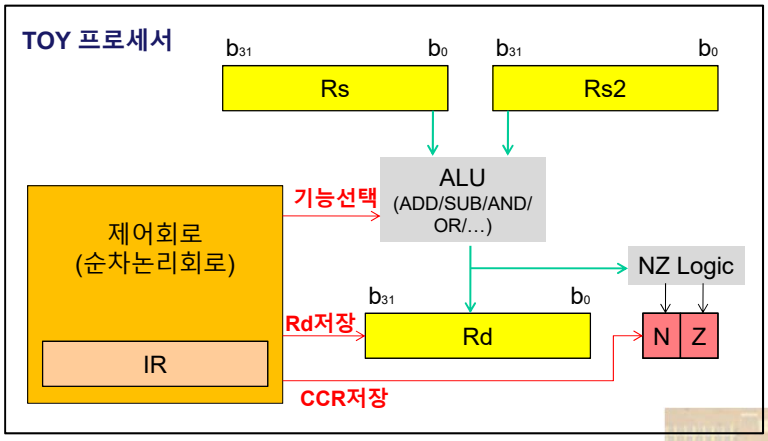
# ADD R3, R1, 9








# (참고) TOY 프로세서의 제어회로




(자세한 내용은 10장 참조)

# 논리연산 명령어

- 대응되는 비트들 간에 논리 연산 적용 (bit-wise operation)
- 문법
  - ✓AND Rd, Rs, Rs2 Rd ← Rs AND Rs2
  - ✓OR Rd, Rs, Rs2 Rd ← Rs OR Rs2
  - ✓XOR Rd, Rs, Rs2 Rd ← Rs XOR Rs2 (exclusive OR)
  - ✓NOT Rd, Rs Rd ← NOT Rs
  - ✓Rs2 는 레지스터 일수도 있고 즉석 값일 수도 있음
  - ✓즉석 값은 16비트로 표현 가능한 정수
- 사용 예
  - ✓OR R1, R1, R2
  - ✓AND R2, R0, 3
  - ✓XOR R1, R2, R3
  - ✓NOT R2, R3





# 비트간의 논리 연산

| R1<br>R2 | 1 1 0 0 0 . . . 0 0 0 1 1<br>0 1 0 1 0 . . . 0 0 1 0 1 | N | Z |
|---|---|---|---|
| R1과 R2의 AND | 0 1 0 0 0 . . . 0 0 0 0 1 | 0 | 0 |
| R1과 R2의 OR | 1 1 0 1 0 . . . 0 0 1 1 1 | 1 | 0 |
| R1과 R2의 XOR | 1 0 0 1 0 . . . 0 0 1 1 0 | 1 | 0 |
| R1의 NOT | 0 0 1 1 1 . . . 1 1 1 0 0 | 0 | 0 |

# 비트간의 논리 연산 (Bit Masking 응용)

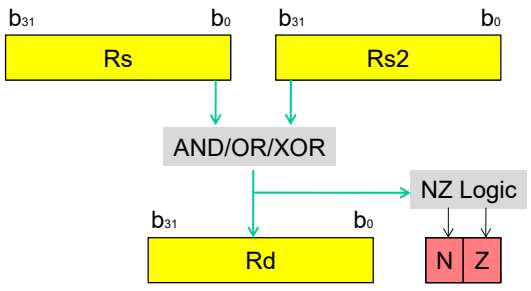
| R0<br>1 | 0 … 0 1 1 0 0 0 0 1 1<br>0 … 0 0 0 0 0 0 0 0 1 | N | Z | 홀수 (Z=0) |
|---|---|---|---|---|
| R0과 1의 AND | 0 … 0 0 0 0 0 0 0 0 1 | 0 | 0 | |

| R0<br>0x20 | 0 … 0 0 1 0 0 0 0 0 1<br>0 … 0 0 0 1 0 0 0 0 0 | N | Z | ‘A’ |
|---|---|---|---|---|
| R0과 0x20의 OR | 0 … 0 0 1 1 0 0 0 0 1 | 0 | 0 | ‘a’ |

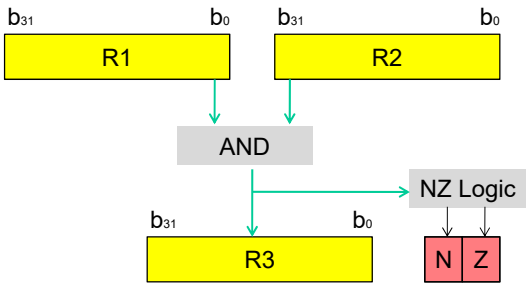




# 레지스터간의 논리연산



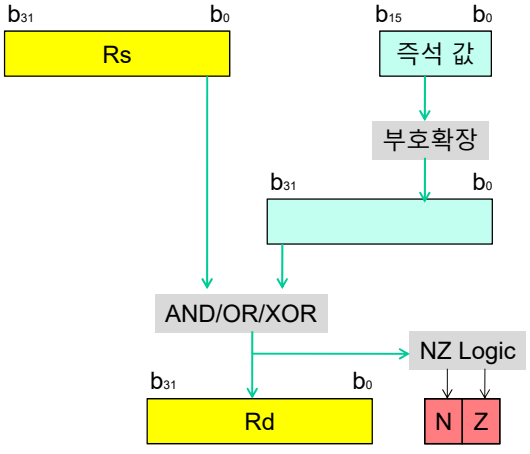
# AND R3, R1, R2



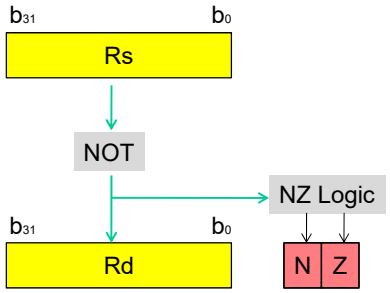





# 레지스터와 즉석 값의 논리연산



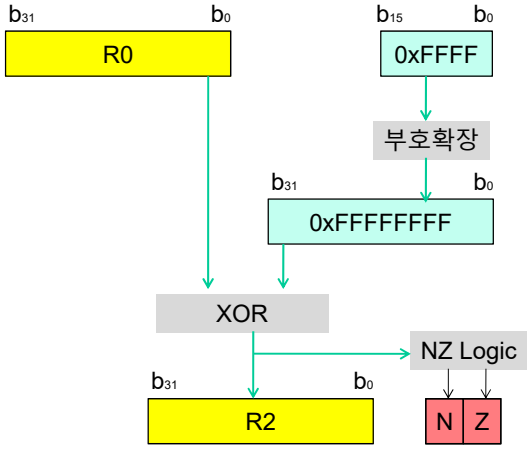
# NOT 연산








# XOR R2, R0, -1



## 직접 값을 이용한 논리연산의 활용예

- AND R2, R0, 1
  - ✓R2 ← (R0의 값) AND (0000…0001)
  - → R0의 끝 비트가 1이면 R2는 1, 아니면 R2는 0
  - → R0의 값이 홀수이면 R2는 1, 아니면 R2는 0
  - → CCR의 Z비트는 R0의 값이 홀수이면 0, 짝수이면 1로 설정
- XOR R1, R0, -1
  - ✓R1 ← (R0의 값) XOR (1111…1111)
  - → 비트 별로 R0 비트가 1이면 0으로, 0이면 1로
  - → NOT R1, R0 와 동일한 결과
  - → TOY 는 NOT 회로가 없고 XOR 회로로 처리함



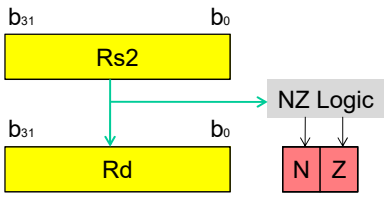


## 복사 명령어

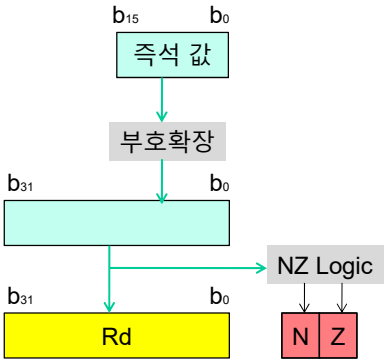
- 문법
  - ✓COPY Rd, Rs2 Rd ← Rs2
  - ✓Rs2 는 레지스터 일수도 있고 즉석 값일 수도 있음
  - ✓즉석 값은 16비트로 표현 가능한 정수
- 사용 예
  - ✓COPY R1, R2 R1 ← R2
  - ✓COPY R1, 15 R1 ← 15
  - ✓COPY R1, -3 R1 ← -3

# COPY 명령어의 실행

레지스터 간의 COPY




즉석 값 COPY








# 메모리 읽기와 쓰기 명령어

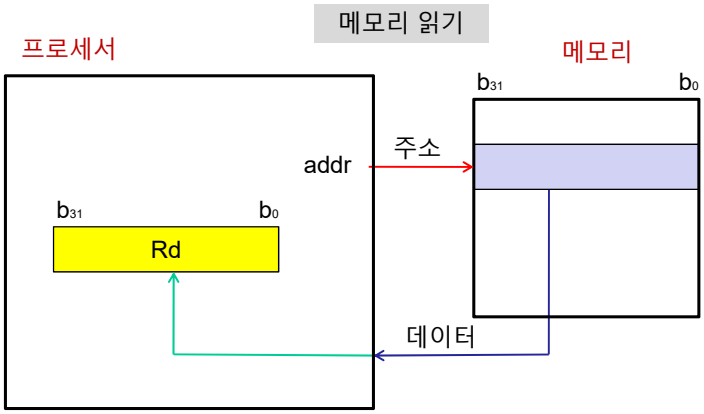
- 문법
  - ✓LOAD Rd, addr — Rd ← 메모리 addr 번지의 내용
  - ✓STORE Rs, addr — 메모리 addr 번지의 내용 ← Rs
- 사용 예 1
  - ✓LOAD R1, 3010 — 메모리 3010 번지의 내용을 R1에 읽어들임
  - ✓STORE R3, 3012 — R3의 내용을 메모리 3012번지에 기록
- 사용 예 2

```
    LOAD R1, value      value 로 표시된 메모리 위치의 내용을
    …                   R1에 읽어 들임 (→ R1은 1234로 갱신)
value:  .FILL 1234
```

- 사용 예 3

```
    STORE R3, sum       sum 으로 표시된 메모리 위치에
    …                   R3의 값을 기록
sum:  .BLOCK 1
```

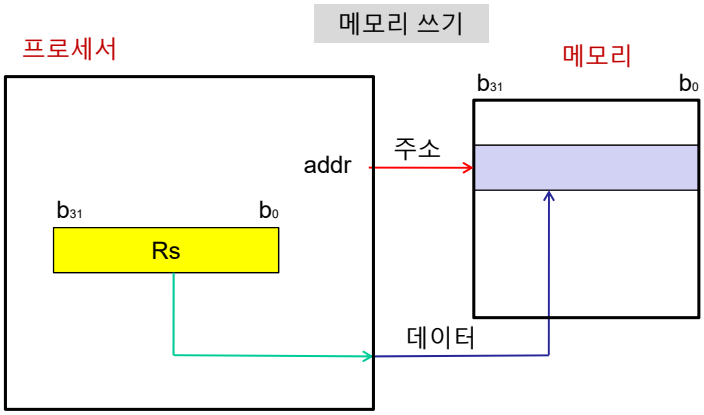
# LOAD 명령어의 실행








# STORE 명령어의 실행



# 레지스터에 32비트 값의 설정

- 16비트로 표현 가능한 정수: COPY 활용
  - ✓COPY R1, 1234
  - ✓COPY R1, -5678

- 16비트로 표현할 수 없는 정수: LOAD 활용

```
        LOAD R1, value
        . . .
value: .FILL 0x12345678
```





# TOY 프로세서 전체 명령어 집합 (1)

| 이름 | 어셈블리언어 표현 | 의미 | CCR 갱신 |
|---|---|---|---|
| Add | **ADD Rd, Rs, Rs2** | Rd ← Rs + Rs2 | O |
| Subtract | **SUB Rd, Rs, Rs2** | Rd ← Rs - Rs2 | O |
| Compare | **CMP Rs, Rs2** | Rs - Rs2 | O |
| And | **AND Rd, Rs, Rs2** | Rd ← Rs AND Rs2 | O |
| Or | **OR Rd, Rs, Rs2** | Rd ← Rs OR Rs2 | O |
| Exclusive Or | **XOR Rd, Rs, Rs2** | Rd ← Rs XOR Rs2 | O |
| Not | **NOT Rd, Rs** | Rd ← NOT Rs | O |
| Logical Shift Left | LSL Rd, Rs, n | Rd ← Rs를 n비트 왼쪽이동 | O |
| Logical Shift Right | LSR Rd, Rs, n | Rd ← Rs를 n비트 오른쪽이동 | O |
| Arithmetic Shift Left | ASL Rd, Rs, n | Rd ← Rs를 n비트 왼쪽이동 | O |
| Arithmetic Shift Right | ASR Rd, Rs, n | Rd ← Rs를 n비트 오른쪽이동 (부호유지) | O |
| Copy | **COPY Rd, Rs2** | Rd ← Rs2 | O |

# TOY 프로세서 전체 명령어 집합 (2)

| 이름 | 어셈블리언어 표현 | 의미 | CCR 갱신 |
|---|---|---|---|
| Load | **LOAD Rd, addr** | Rd ← 메모리의 addr 번지 내용 | X |
| | LDR Rd, Ra, offset | Rd ← 메모리의 Ra+offset 번지 내용 | X |
| Store | **STORE Rs, addr** | 메모리의 addr 번지 내용 ← Rs | X |
| | STR Rs, Ra, offset | 메모리의 Ra+offset 번지 내용 ← Rs | X |
| Branch | BR nzp, addr | 조건에 따라 실행위치를 addr 주소로 이동 | X |
| | BRR nzp, Ra, offset | 조건에 따라 실행위치를 Ra+offset 주소로 이동 | X |
| Load Effective Address | LEA Rd, addr | Rd ← addr 번지 | X |
| Link | LINK | R6 ← PC+1 | X |
| Return | RET | PC ← R6 | X |
| Software Interrupt | SWI n | 운영체제의 n 번 기능 호출 | X |
| Return from Interrupt | RTI | SWI 명령어 다음 위치로 복귀 | X |

(주) - Rd, Rs, Ra : R0부터 R7까지의 레지스터를 지정, R7은 PC에 해당함
- Rs2 : R0부터 R7까지의 레지스터나 16비트로 표현이 가능한 정수 값
- n : 0부터 31까지의 값을 지정
- addr : 프로그램 상의 레이블이나 주소 (현재 PC 값과의 차이가 16비트 정수로 표현이 가능한 범위 이내)
- offset : 16비트로 표현이 가능한 정수 값

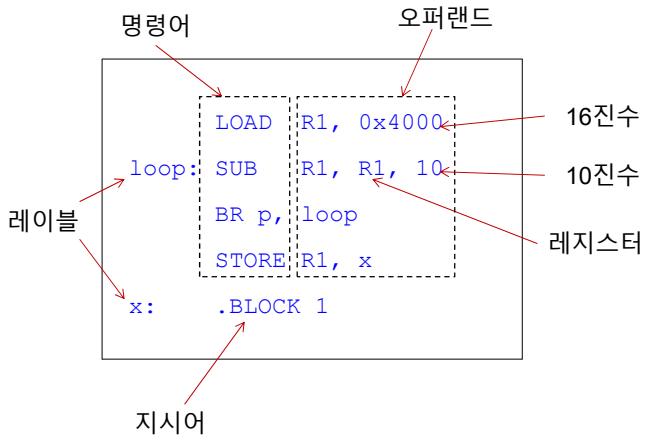




# 어셈블리 언어 문법

- 줄 단위로 표현 (각각 생략 가능, 레이블만 첫 칸부터)

  label: instruction ; comment



# 오퍼랜드 표기 법

| 오퍼랜드 | 표기 | 의미 |
|---|---|---|
| 레지스터 | R0, R1, ..., R7, PC | 특정 레지스터 (PC는 R7과 동일) |
| 수치 | 3, -25, 0x54 | 직접 지정된 수치 (10진수, 16진수) |
| ASCII 코드 | 'A', 'b', '3', '#' | 문자의 ASCII 코드 |
| 주소 | 3000, 0x40A0 | 메모리 상의 주소로서 음이 아닌 수 (10진수, 16진수) |
| 레이블 | x, sum, _next, L2 | 프로그램에 지정된 레이블 위치 |





# 어셈블러의 지시어와 레이블

- 레이블 (label)
  - ✓고정된 주소 대신에 이름으로 메모리 상의 위치 지정
- 지시어 (directive)
  - ✓어셈블링 과정에서 필요한 정보를 제공하기 위한 내용
  - ✓어셈블러가 구별하기 쉽도록 '.' 으로 시작하는 명칭 사용

# TOY 어셈블러의 지시어와 레이블

| 지시어 | 사용 예 | 의미 |
|---|---|---|
| **.ORIGIN** addr | **.ORIGIN** 0x2000 | 프로그램을 0x2000번지부터 시작 |
| **.FILL** value | **.FILL** 2345 | 현재 주소에 1개의 32비트 메모리 영역을 확보하고 값은 2345로 설정 |
| | **.FILL** 'A' | 현재 주소에 1개의 32비트 메모리 영역을 확보하고 값은 문자 A의 ASCII 코드(0x41)로 설정 |
| | **.FILL** L1 | 현재 주소에 1개의 32비트 메모리 영역을 확보하고 값은 레이블 L1에 해당하는 주소로 설정 |
| **.BLOCK** n | **.BLOCK** 5 | 현재의 주소부터 5개의 32비트 메모리 영역 확보 |
| **.STRING** "str" | **.STRING** "yes" | 현재의 주소부터 4개의 32비트 메모리 영역을 확보하고 차례대로 'y', 'e', 's', 및 null (0)의 ASCII 코드를 저장 |
| (레이블의 표시) | **again:**<br>...<br>BR p, **again** | 직전의 실행결과가 양수이면 'again'으로 표시된 레이블 주소로 실행위치를 이동<br>***(데이터 영역의 레이블은 변수 이름)*** |





# 프로그램에서 변수의 사용

C 언어 프로그램

```
int x = 7, y;
...
y = x + 3;
...
```

TOY 어셈블리 언어 프로그램

```
    ...
    LOAD R1, x     ; 변수 x의 값을 읽어서 R1에 저장
    ADD R1, R1, 3  ; R1에 3을 더해서 다시 R1에 저장
    STORE R1, y    ; R1의 값을 변수 y의 값으로 저장
    ...
x:  .FILL 7        ; 레이블 x 의 위치에 32비트 메모리 영역을
                   ; 확보하고 초기 값을 7로 지정
y:  .BLOCK 1       ; 레이블 y 의 위치에 1개의 32비트 크기의
                   ; 메모리 영역확보
    ...
```

# 프로그램: ASCII 대소문자 변환

```
    .ORIGIN 0x2000    ; 프로그램 시작주소를 0x2000 번지로
    LOAD R1, low1     ; 메모리 low1 부분의 값을 R1에 읽어들임
    SUB R1, R1, 0x20  ; 0x20을 빼서 대문자로 변환
    STORE R1, up1     ; 변환한 결과를 메모리의 up1 부분에 저장
    LOAD R1, up2      ; 메모리 up2 부분의 값을 R1에 읽어들임
    ADD R1, R1, 0x20  ; 0x20을 더해서 소문자로 변환
    STORE R1, low2    ; 변환한 결과를 메모리의 low2 부분에 저장
    ...
low1: .FILL 'a'       ; 문자 'a' 의 ASCII 코드 (0x61)
up1: .BLOCK 1         ; 1개의 메모리 영역 확보
up2: .FILL 'B'        ; 문자 'B' 의 ASCII 코드 (0x42)
low2: .BLOCK 1        ; 1개의 메모리 영역 확보
```





# 프로그램: 8을 나눈 나머지 구하기

```
    .ORIGIN 0x2000
    LOAD R1, value    ; 메모리 value 부분의 값을 R1에 읽어들임
    AND R2, R1, 0x07  ; 끝의 3비트만 남기고 나머지는 0으로
    STORE R2, remain  ; 계산 결과를 remain 부분에 저장
    ...
value: .FILL 123456   ; 주어진 숫자
remain: .BLOCK 1      ; 나머지를 저장하기 위한 메모리 영역
```

## 약간 복잡한 예제

- 369 게임
  - ✓프로그램 3.9 (어셈블리 언어 프로그램)
  - ✓프로그램 7.4 c369.c (C 언어 프로그램)
  - ✓프로그램 7.5 c369.s (어셈블리언어 프로그램)
- 간단한 계산기
  - ✓프로그램 7.6 calc.c (C 언어 프로그램)
  - ✓프로그램 7.7 calc.s (어셈블리언어 프로그램)





## 정리

- 프로세서의 프로그래밍 모델
- 명령어 집합
- 어셈블리 언어 문법