Ch.2-1 Instructions: Language of the Computer

2.1 MIPS instruction set

instruction set

컴퓨터의 명령어 집합 ⇒ 컴퓨터가 다르면 다른 instruction set 가짐

- *초기 컴퓨터* : 매우 간단한 instruction set → simplified implementation ex. 한 cycle당 하나의 명령어 (순차적으로)
- *현대 컴퓨터* : 복잡한 instruction set(CISC) + 간단한 instruction sets ex. ,, 많은명령어 ↵ HW수
  - CISC : 모든 명령어가 memory 필요하진 않음, 연산은 register에서만
  - reduced instruction sets(RISC) ⇒ 단순화, 정규화

MIPS instruction Set ⇒ embedded core 사용

↳ 요즘!

↳RISC ★

- stanford MIPS commercialized by MIPS tech.
- Design Principle

복잡. ↑ CISC → complexed
RISC → reduced

1. 정규화 → 단순화
   a. hw 측면에서 instruction 정규화 → 단순화
   b. 단순화 : lower cost에서 높은 성능을 보장
2. 작은 것이 더 빠르다 (한정된 ISA → 한정된 register → 한정된 addressing mode)
   - 레지스터가 많아짐 → 전기 신호 전달 더 멀리 → clock cycle 길어짐
3. 흔한 경우는 더 빠르게 만들기
   - 작은 상수는 많이 쓰인다
   - 산술 피연산자는 load instruction을 하지 않는다

4. 좋은 디자인 → 좋은 compromises 존재 (각각의 instruction format)

2.2 Arithmetic Operations

① Simplicity → Regularity 선택.

- two sources + one destination

```
add t0, g, h
add t1, i, j
sub g, t0, t1
```

dst, src

temp variable

$t_0 = g + h$

$t_1 = i + j$

$g = t_0 - t_1 = (g+h) - (i+j)$

⇒ compiler 변환과정

2.3 Operands

상위 수준 언어와 달리 피연산자에는 제약이 생김 → 레지스터에서만 사용 (memory 사용 X)

Register Operands(register에게 할당) : R-R to architecture

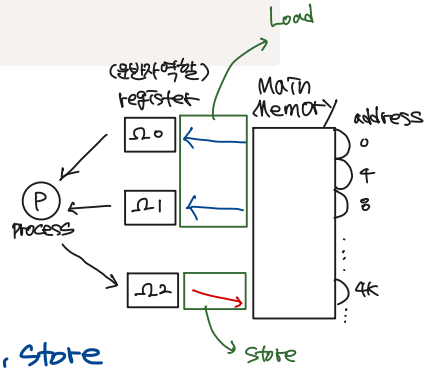
- arithmetic instructions ⇒ register operands

↳ Register : processor 내부적으로 관리하는 저장공간

- **MIPS** : 32 X 32 bit의 register file
  - word : 32비트 단위를 말함(0 to 31)
- assembler name
  - 변수 해당 레지스터 → $s0, $s1, $s2 ...
  - 컴파일 과정에서 필요한 임시 레지스터 → $t0. $t1, $t2 ...

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

▼ Memory Operands



- processor와 memory
  - *processor* : 소량의 데이터만 레지스터에 저장
  - *memory* : 수십억 개의 데이터 저장 → 배열, 구조체 같은 자료구조

→ MIPS의 arithmetic operations → register에서만 실행

⇒ data transfer instruction 필요(register에서 memory로 전달) → load, store

  - memory address(32bit) : base address register의 offset value

- **data transfer transition**

```
memory
lw  $t0,  4($s3)   #load word from memory
sw  $t0,  8($s3)   #store word to memory
```

  - ***lw :*** load value (memory→register)
  - ***sw :*** store result (register→memory)

```
lb   $t0, 1($s3)   #load byte from memory
sb   $t0, 6($s3)   #store byte to  memory
```

0x28	19	8	16 bit offset

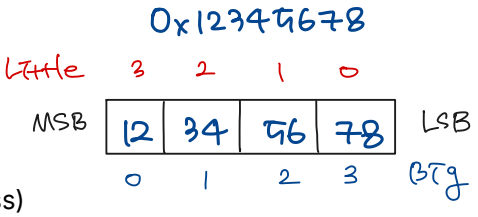
  - ***lb :*** load byte(memory → register)
    - destination register의 가장 오른쪽 8bit를 memory로부터 읽어옴
  - ***sb :*** store byte (register → memory)
    - register로부터 가장 오른쪽 8bit 읽어오고 memory에 byte만큼 write

- Byte address
  - memory address 지정 필요함 → byte addressed(8 bit byte)

- word address → 4bit

⇒ MIPS의 시작 주소는 항상 4의 배수(alignment restriction)

- MIPS: Big Endian(최상위 byte address를 word address)
- cf) liitel Endian



- example

```
#c code : g = h + A[8];
lw $t0, 32($s3)
add $s1, $s2, $t0

#c code : A[12] = h + A[8];
lw $t0, 32($s3) # load word
add $t0, $s2, $t0
sw $t0, 48($s3)
```

- registers vs memory
  - ① register : memory보다 접근이 빠름
    - 단위 시간 당 훨씬 많은 명령어 실행
    - register numbers

$t0 – $t7 are reg's 8 – 15
$t8 – $t9 are reg's 24 – 25
$s0 – $s7 are reg's 16 – 23

  - ② memory : data 연산을 하기 위해 load, store 필요 → 더 많은 instructions

⇒ complier : register를 가능한 많은 변수에 대해서 사용해야 한다

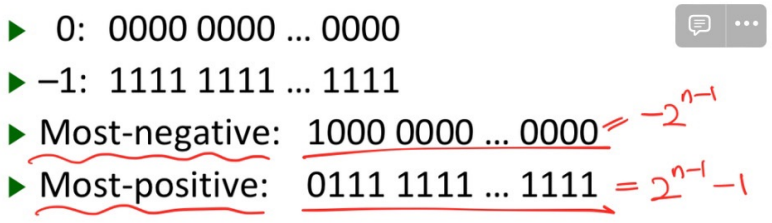
▼ immediate operands(수치 피연산자)

메모리에 저장 명령을 사용하지 않고 산술 연산 가능한 명령어

```
addi $s2, $s1, -1
뺄셈 연산이 없다 → 그냥 음수 상수 써도 된다!
```

- 연산이 훨씬 빨라지고 energy 소모 감소
- 상수 0 : register $zero로 사용 → cannot be overwritten
  - 자주 쓰이는 케이스로 시간 단축

```
add $t2, $s1, $zero
```



2.4 binary Integers

Unsigned Binary Integers

- **n** bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- **0** to **2^n -1**

2s-Complement Signed Integers

- **n** bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- **-2^(n-1)** to **2^(n-1)-1**
- Signbit(MSB)
  - **1** → 음수
  - **0** → 양수 ⇒ 2의 보수 표현이랑 unsigned 표현이랑 같음

Signed Negation

- complement and add 1
  - complement : 비트 반대로 바꾸기

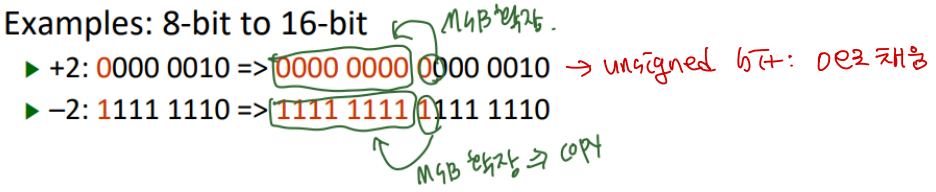
$$x + \bar{x} = 1111\ldots111_2 = -1$$
$$\bar{x} + 1 = -x$$

Sign Extension

부호 있는 적재의 경우 레지스터의 남는 곳을 채우기 위해 부호를 반복 → 확장

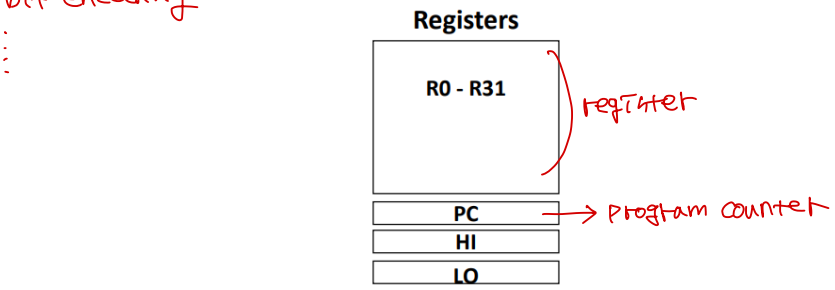
(cf : 부호 없는 수 → 0으로 확장)

- MIPS instruction set
  - ***addi*** : extend immediate value
  - ***lb*** : 바이트를 부호 있는 수로 간주 → 24비트를 부호 확장하여 채움
  - ***lh*** : 12비트 부호 확장하여 채움
  - ***beq, bne*** : extend the displacement

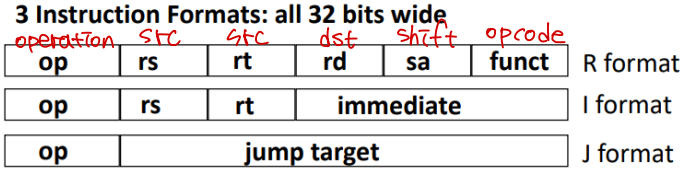


▼ 2.5 명령어의 컴퓨터 내용 표현

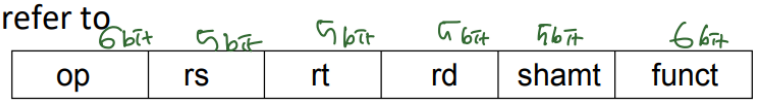
▼ MIPS-32 ISA



- Instruction categories
  - computational
  - load / store
  - jump and branch
  - floating point → coprocessor
  - memory management
  - special



- MIPS - **R instructions format**
  - MIPS field → 32 bit

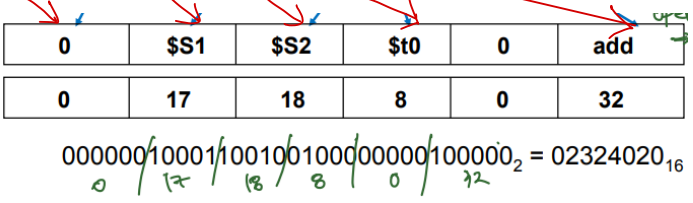


- op : 명령어가 실행할 연산의 종류 → 연산자
- rs : 첫번째 근원지 피연산자 레지스터
- rt : 두번째 근원지 피연산자 레지스터
- rd : destination register → 연산 결과
- shamt : 자리 이동 shift
- funct : 기능 → op 필드에서 정해진 연산을 구체화

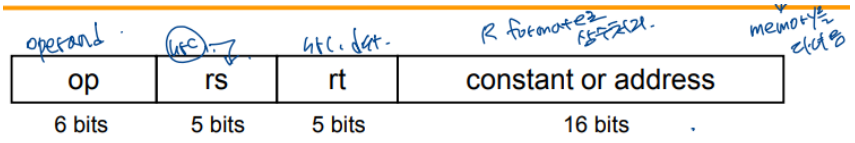
- machine code : 명령어를 숫자로 표현한 것

```
add $t0, $s1, $s2
(destination <- source1 op source2)
```

  - opeation → 1개, operand → 3개(register file의 경로도 모두 포함)



- MIPS - **I intructions fomat** ⇒ Immediate arithmetic, load/store instruction 시에 사용



  - rt : destination or source reigster number
  - constant : -2^15 to 2^15 -1
  - address : offset added to base address in rs

좋은 설계 → 타협점 찾기

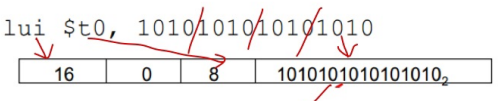
- *Design Principle 4 : good design → good compromises에 따라 달림*
  - 다른 format : decoding하는데 오래 걸림
  - 가능한 유사한 format → 단순한 format ⇒ 간단하게 !

- Load instruction example

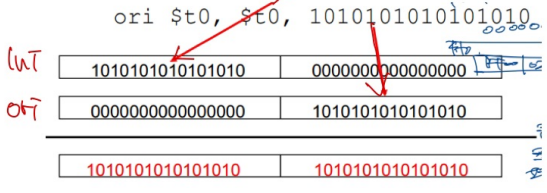
① Larger Constants ⇒

  - register에 32 bit로 로드해야 함

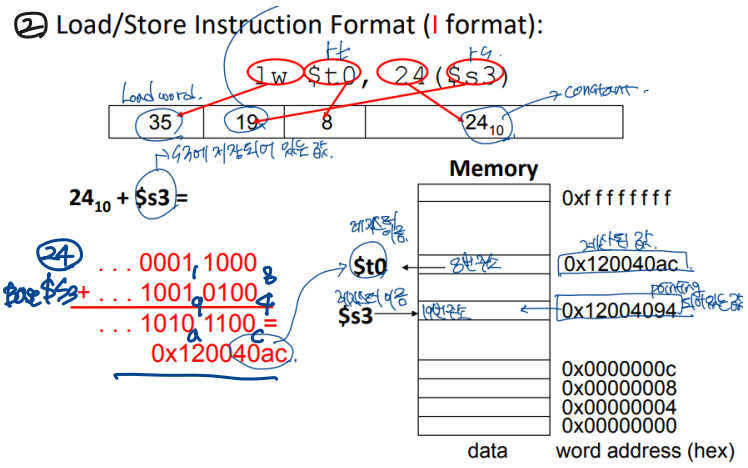
1. load upper immediate instruction ⇒ 메모리 손실 적게



2. lower order bits



② Load/Store Instruction Format (I format):

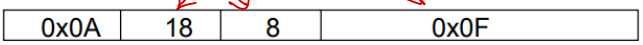


③ immediate instructions example → 상수 사용, 바로 메모리에 적정 (Load)

```
addi $sp, $sp, 4      #$sp = $sp + 4
slti $t0, $s2, 15     #$t0 = 1 if $s2<15
```

⇒ 비교연산자

Machine format (I format):



Load word 느낌 format

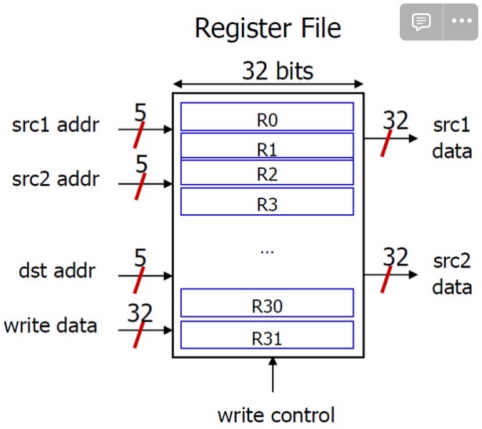
The constant is kept inside the instruction itself!

⇒ 대부분 명령에서 상수를 사용함. (큰 값 x)

- Immediate format limits values to the range -2^{15} to $+2^{15}-1$

→ 즉 16bit 까지 제한.

▼ MIPS register file



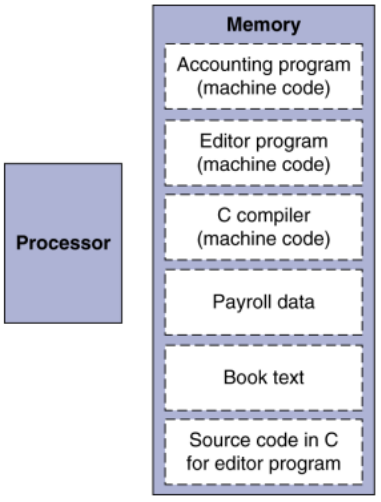
- holds 32개의 32bit registers
  - 두 개의 read port
  - 하나의 write port
- register
  - ① main memory보다 빠름
    - But, 더 많은 위치를 location의 register file → 느림
    - r/w port가 속도를 매우 빠르게 증가시킴
  - ② compiler가 사용하기에 편함
  - ③ 변수를 가지고 있을 수 있음(코드 향상)

- register가 메모리 location보다 더 적은 bit를 가짐)

▼ 요점 정리

- 컴퓨터의 두 가지 중요한 원리
  1. 명령어는 숫자로 표현 ⇒ 데이터와 구별X
  2. 프로그램은 메모리에 기억 → 데이터처럼 읽고 쓸 수 있다

⇒ 내장 프로그램(stored-program)의 개념



- stored program
  - program : binary number file 형태로 판매됨 → binary compatibility (이진호환성)
  - binary compatibility : 다른 컴퓨터의 sw를 물려받을 수 있음 ⇒ ISAs

▼ 2.6 논리 연산 명령어

▼ MIPS Logical operations

- Logical Operations

Operation	C	Java	MIPS
Shift left	<<	<<	sll → shift left
Shift right	>>	>>	srl
Bitwise AND	& (&&→논리연산)	& →비트연산	and, andi
Bitwise OR	|	|	or, ori
Bitwise NOT	~	~	nor

- MIPS ISA : 수많은 bit-wise logical operation 존재

```
and $t0, $t1, $t2     #$t0 = $t1 & $t2
or  $t0, $t1, $t2     #$t0 = $t1 | $t2
nor $t0, $t1, $t2     #$t0 = not($t1 | $t2)
```

- instruction Format(R format)

0	9	10	8	0	0x24

```
andi $t0, $t1, 0xFF00 #$t0 = $t1 & ff00
ori  $t0, $t1, 0xFF00 #$t0 = $t1 | ff00
```

- instruction Format(I format)

0x0D	9	8	0xFF00

▼ MIPS shift Operations

Instruction Format (R format)

shift ⇒ 얼만큼 shift?

op		rt	rd	shamt	funct

Shifts move all the bits in a word left or right

```
sll $t2, $s0, 8      #$t2 = $s0 << 8 bits
srl $t2, $s0, 8      #$t2 = $s0 >> 8 bits
```

- logical → fill with zeros
  - 5-bit shamt ⇒ 32bit shift하기에 충분함
- ***sll*** : shift left logical ⇒ 가리키는 왼쪽으로 이동 ($\times 2^n$)
- ***srl*** : shift right logica l (부호 없는 것만 가능) ⇒ 가리키는 오른쪽으로 이동 ($\div 2^n$) ⇒ unsigned

sra : arithmetic
srl : logical
구동 방식이 차이 존재

▼ MIPS AND Operations

- 둘 다 1인 곳만 1이 됨
- 0의 위치에 해당하는 비트들 강제로 0으로 만듦 → mask

```
and $t0, $t1, $t2
```

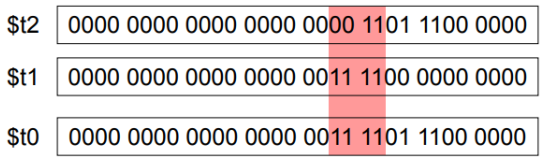
$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0000 1100 0000 0000

→ 나머지 0으로 표시

▼ MIPS OR Operations

- 둘 중 하나만 1이면 1이 됨

```
or $t0, $t1, $t2
```



▼ MIPS NOT Operations

- **NOT** → 0이면 1로, 1이면 0으로
- MIPS : NOR : ㅋ – 피연산자 형식을 유지하기 위해 **NOT OR** 사용
  - A NOT 0 = NOT (A OR 0) = NOT (A)

```
nor $t0, $t1, $zero
```