숭실대학교
Soongsil University

Computer Architecture

강의 #3: Instructions: Language of the Machine (1)

2021년 1학기

Young Geun Kim (김영근)

컴퓨터의 언어

- 컴퓨터도 언어가 필요함
- 프로그래밍 언어는 데이터 처리에 대한 상징적인 의미를 제공함

프로그래밍 언어

- **많은 종류의 프로그래밍 언어가 있지만, 크게 2개의 카테고리로 분류할 수 있음**
  - High-level Languages
    - 보통 하드웨어에 종속적이지 않음 (= Machine Independent)
    - 하나의 문장이 여러 개의 명령어를 포함함
    - C, C++, Python, Java, etc.
  - Low-level Languages
    - 하드웨어에 종속적임 (= Machine Specific)
    - 하나의 문장이 프로세서의 기계어 하나로 대응되는 경우가 많음
    - X86, ARM, MIPS, 등 프로세서를 위한 Assembly 언어

- **이번 강의에서는 High-level Language보다는 Low-level Language와 그 둘 사이의 관계에 집중함**

Assembly Languages

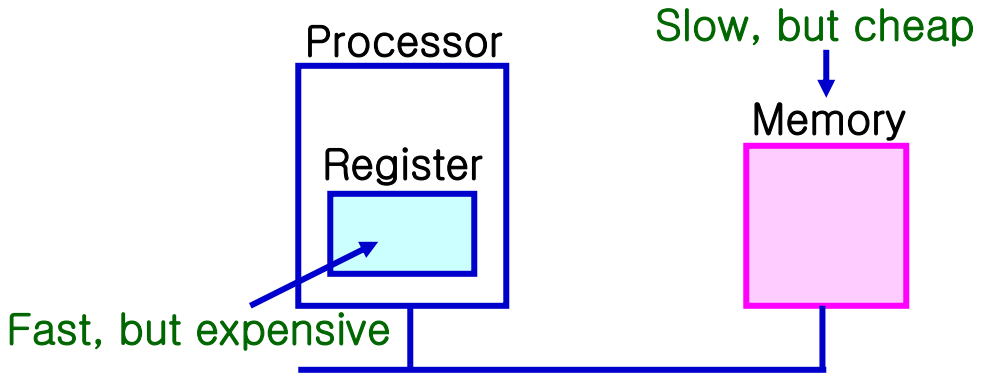
- 기계어에 대한 Text Representation (문자형)
- 보통 하나의 문장 (Statement)이 하나의 기계어에 대응됨
- High-level 프로그램과 기계어 코드 사이의 Abstraction Layer임

Machine Languages (기계어)

- 컴퓨터의 Native 언어
- 하드웨어에서 실행되는 동작을 bits로 표현
- 이번 강의에서는 MIPS 명령어를 살펴볼 예정

Instruction Set Architecture (명령어 집합 구조)

- 하드웨어와 소프트웨어 사이의 Interface
- 일반적인 데이터 처리 방식
  - 저장 대상: Registers & Memory
  - 동작: Instructions (명령어)



- 명령어 집합 구조 Design의 목표
  - 고성능/저비용의 구현이 가능하도록 함
  - 동시에, 상위 Layer (e.g., 응용, 운영체제, 등)의 요구조건을 만족함

Assembly 명령어 구성

- **어떤 정보가 필요한지에 따라, 명령어는 다양하게 구성될 수 있음**

- **MIPS Architecture는 3가지 명령어 Format으로 구성됨**
  - 모든 명령어는 32 bits로 구성되어 있음

- **각 명령어의 첫 6 bits는 이 명령어가 어떤 동작을 하는지를 나타내는 Opcode로 구성되어 있음**
  - 프로세서가 첫 6 bits를 보고, 어떤 종류의 명령어인지를 알 수 있도록 하기 위함임

Arithmetic Operations (산술 연산)

- 덧셈, 뺄셈을 위한 C 언어 문장

  a = b + c

  a = b - c

- 덧셈을 위한 MIPS의 Assembly 명령어

  – add a, b, c

- 뺄셈을 위한 MIPS의 Assembly 명령어

  – sub a, b, c

복잡한 연산

- **좀 더 복잡한 연산은?**

  a = b + c + d - e

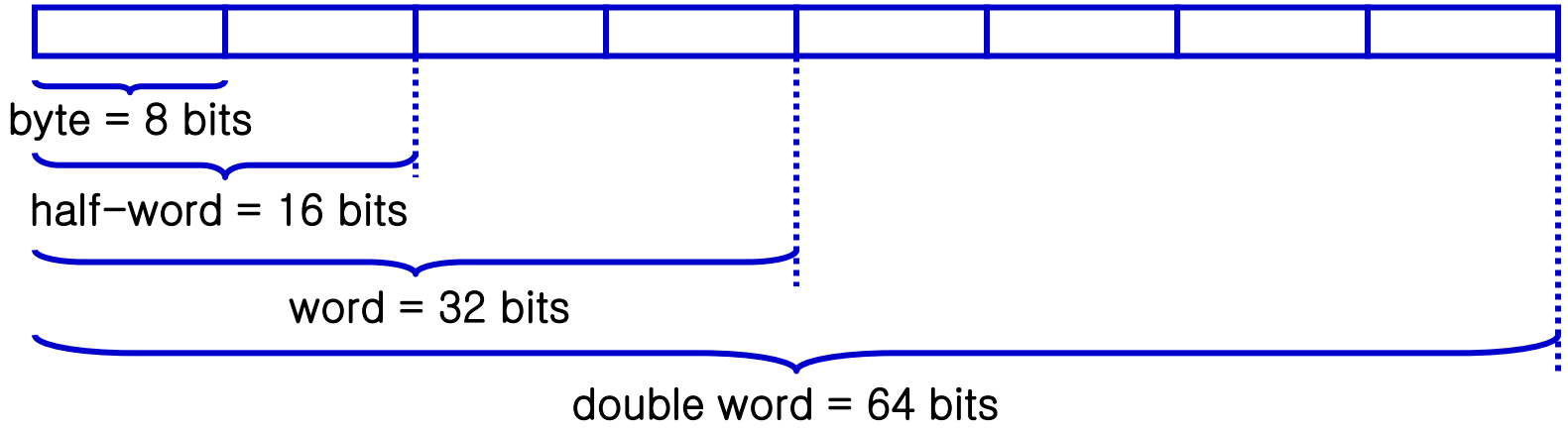
- **여러 개의 명령어로 표현 가능**

```
add t0, b, c        # t0 = b + c
add t1, t0, d       # t1 = t0 + d (= b + c + d)
sub a, t1, e        # a = t1 – e (= b + c + d – e)
```

- **Compiler가 High-level 언어를 Assembly 언어 (및 기계어)로 변환할 때, 임시 변수를 보통 사용함**

Data 표현

- **변수는 어떻게 표현할까?**
  - bits: 0 or 1 (이진법 표현)
- **Bit Strings: 연속적인 Bit들**

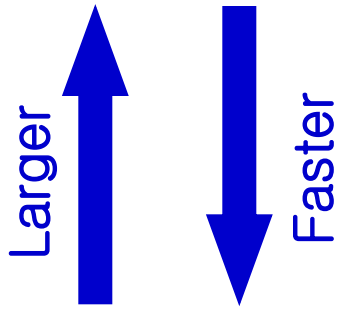


- **Characters: 1 Byte**
- **Integer: 4 Byte (= 32 bits)**

Data의 저장

- **High-level 언어에서는 변수/배열을 선언해서 데이터를 저장함**
- **실제로는 해당 데이터가 어디에 저장될까?**

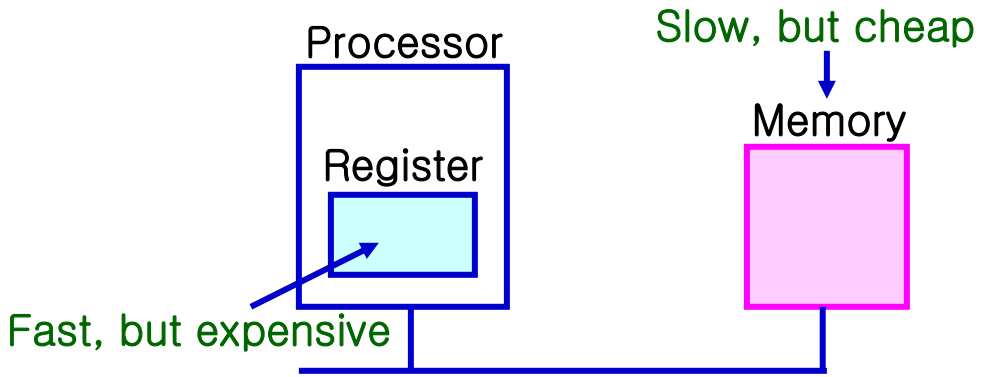
- **데이터는 다양한 장소에 저장됨**
  - Disk
  - RAM
  - Cache
  - Registers



- **크기와 성능 사이의 Trade-Off가 필요함**
  - 조금 더 자세한 내용은 Memory Hierarchy Chapter에서 설명될 예정

Register

- **정의**
  - 작지만 고성능의 메모리



- **Register를 구현하는 세가지 방법**
  - Accumulator
  - Stack
  - General Purpose Register (GPR)
    - 한정된 수의 Register가 Data를 (임시로) 저장하기 위해 사용됨 (현재 거의 대부분의 System)

Accumulator의 예

- **Accumulator는?**
  - 하나의 Register가 연산을 위한 Data의 Source와 Destination으로 사용됨

- **아래와 같은 연산의 경우**

  a = b + c

- **Accumulator 기반 구조에서는**

  load address B

  add address C

  store address A

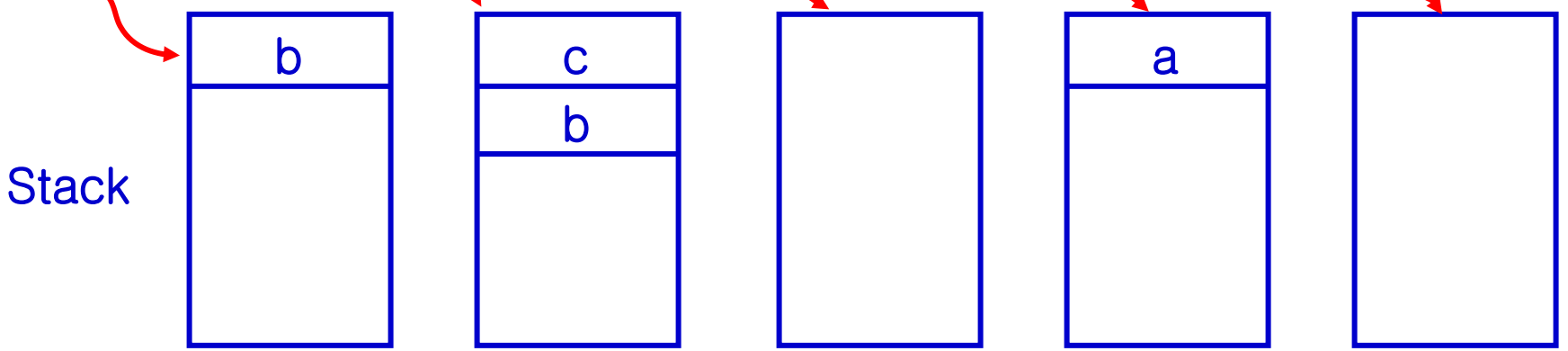
Stack의 예

- **Stack은?**
  - 후입선출 (Last In First Out) 방식의 메모리
- **이전 연산을 다시 살펴보면**

a = b + c

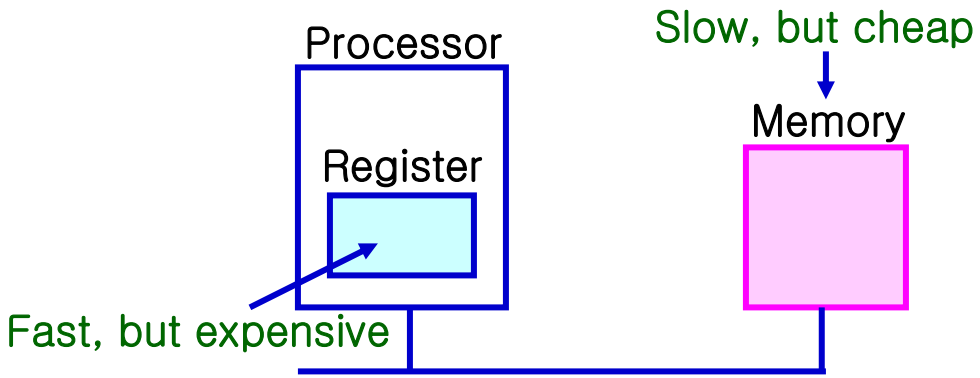
❑ **In Java bytecode**

```
iload_1      # Pushes b onto the stack
iload_2      # Pushes c onto the stack
iadd         # Adds and Pushes the result onto the stack
istore_0     # Stores into a
```



General Purpose Registers

- **대표적인 구현 방식**
  - Load-Store (L/S) 방식
    - 동작
      1. Data를 Memory에서 Register로 "Load"함
      2. 연산 작업 (e.g., 덧셈, 뺄셈, 등)을 수행함
      3. 연산 작업의 결과물을 Register에서 Memory로 "Store"함
    - 대부분의 RISC 방식의 하드웨어는 이 방식을 사용함
      - RISC vs. CISC

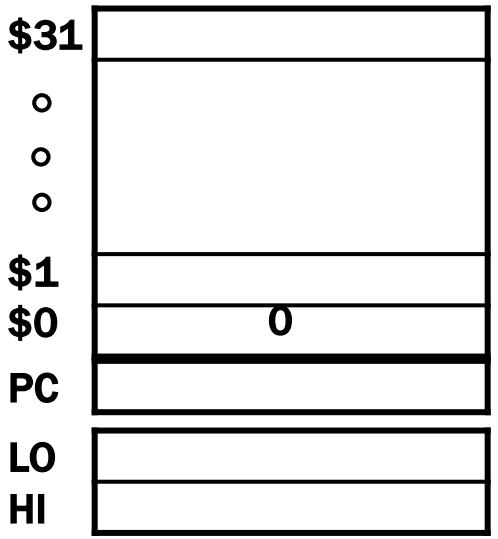


MIPS Architecture

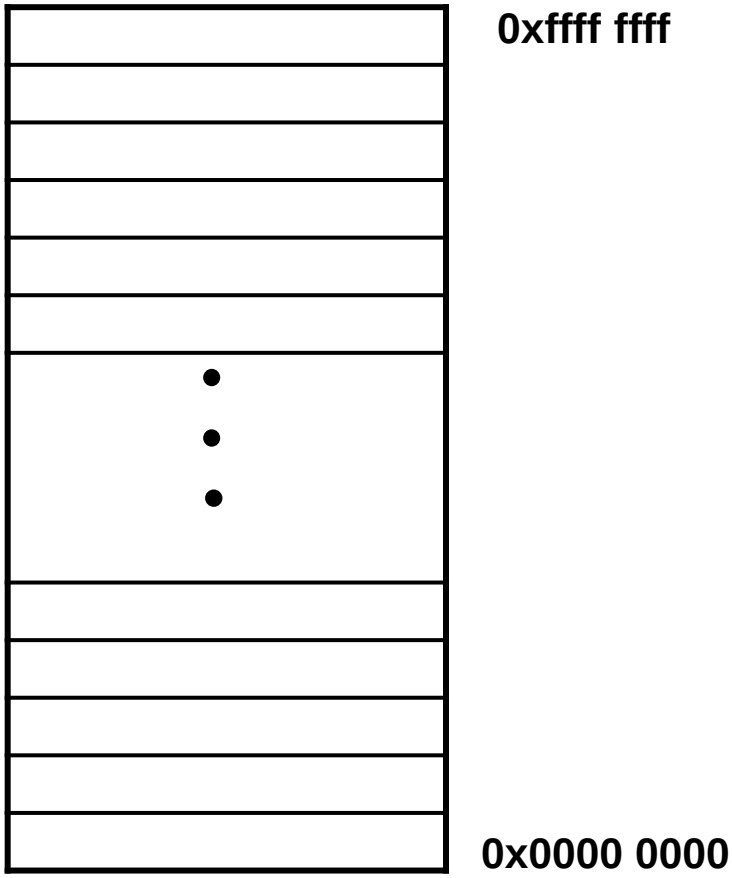
- **A Load-Store Architecture**
  - Register Size: 32 bits
  - GPR의 개수: 32

MIPS Register and Memory Model

Register



Memory



32 GPR = 128 bytes

access time: 1 cycle

2^{30} words = 2^{32} bytes = 4GB

access time: multiple cycles

Register Naming

- **Register 0-31은 $<번호>로 이름을 붙임**
- **용도에 따라 아래와 같이 이름을 붙이고 구분할 수 있음**
  - $zero: 0으로 된 32 bits가 저장되어 있음
  - $s0 - $s7: 변수들을 저장 (Save)하기 위한 Register들
    - High-level 언어의 변수들에 대응되도록 사용함
  - $t0 - $t9: 임시로 사용되는 변수 (= Temporary Variables)들을 위한 Register들

- **변수와 달리, Register의 수는 일반적으로 정해져있음**
  - 크기와 수가 작으면 작을수록 빠르기 때문임
  - 왜?
    - 크기와 개수가 크면, 원하는 데이터를 찾는데 시간이 더 걸림

Using Registers

- **목적**
  - Data를 최대한 Register에서 들고 있도록 함
  - 가능하면 Register에 저장되어 있는 Data를 우선적으로 사용함
    - 왜? Register가 Memory보다 훨씬 빠르기 때문
- **Issues**
  - 제한적인 Register만 사용 가능함
    - 모든 Register가 사용 중이면, 그 중 일부에 저장된 Data를 Memory로 보내야 함
    - Function (= Procedure)을 부르는 경우, Function에서만 사용하는 변수들에 대해 처리할 수 있어야 함
  - 배열
    - Register에 저장하기에는 크기가 너무 큼
    - 배열 내 데이터에 접근하기 위해서 Index 계산이 추가적으로 필요함
  - Dynamic Memory Allocation (e.g., malloc)
    - Runtime에 어느 정도 크기의 Memory가 할당될지 모르지만, Register는 32 Bit만 저장할 수 있음

Three Instruction Formats in MIPS

Name	Fields						Comments
Field size	6bits	5bits	5bits	5bits	5bits	6bits	All MIPS insturctions 32 bits
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

Instruction Format (I)

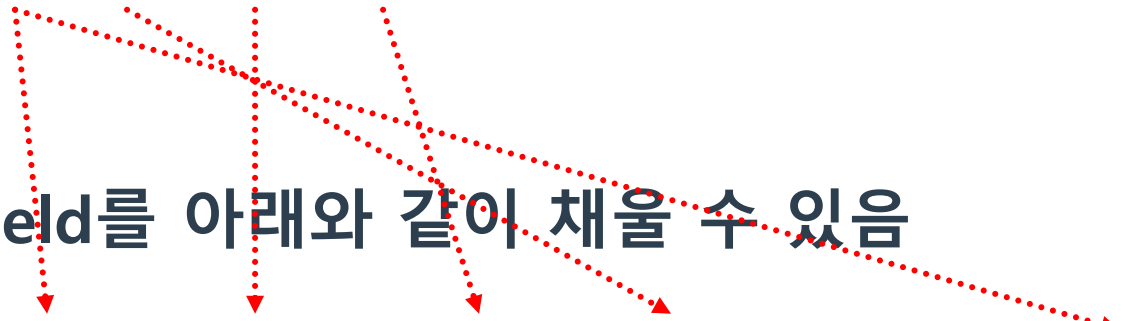
- **R-Format**
  - ALU (Arithmetic and Logical Unit) 명령어에서 주로 사용됨
  - 이름에서 확인할 수 있는 것처럼, 대부분 산술 연산 및 논리 연산을 수행함
  - **3개의 Register를 사용함**
    - 1개는 Destination (연산의 결과물이 저장됨)
    - 2개는 Source (연산의 대상이 저장되어 있던 곳)

Fields					
6bits	5bits	5bits	5bits	5bits	6bits
op	rs	rt	rd	shamt	funct
	1st source register	2nd source register	result register	shift amount	function code

Instruction Format (II)

- 예제
  - Add 명령어의 예

    add $t0, $s1, $s2



- 각 Field를 아래와 같이 채울 수 있음

Fields					
6bits	5bits	5bits	5bits	5bits	6bits
op	rs	rt	rd	shamt	funct
	1st source register	2nd source register	result register	shift amount	function code

000000	10001	10010	01000	00000	100000

Instruction Format (III)

- **R-Format 명령어의 한계**
  - ALU-type의 연산 (2개의 연산자를 활용해 결과물을 만드는 연산)에는 잘 맞지만, 다른 동작에는 잘 맞지 않음
  - 예를 들어, Load/Store관련 동작에는?
    - R-Format을 사용하면, 오직 5개의 bit만 메모리 Offset을 위해 활용할 수 있음
    - 배열의 크기가 2^5 = 32 보다 큰 경우에는?

- **좋은 Design은 다양한 형식의 프로그램에 대해서 사용될 수 있어야 하기 때문에, 한 개의 Instruction Format으로는 부족함**

Instruction Format (IV)

- **I-Format (Immediate Instruction Format)**
  - 각 명령어를 위해 다른 Opcode를 사용함
  - Immediate Field는 부호가 있음 (양수/음수 모두 표현 가능)
  - 주로 Load 및 Store연산에 활용되고,
    직접 입력된 값 (Immediate Value)에 대한 산술/논리 연산에도 활용됨
  - Branch Destination이 PC 값에 Offset 값을 더해 계산이 되므로 (= PC Relative), Branch 명령어에도 활용됨

Name	Fields					
Field size	6bits	5bits	5bits	5bits	5bits	6bits
I–format	op	rs	rt	address/immediate		

1st source register

2nd source register

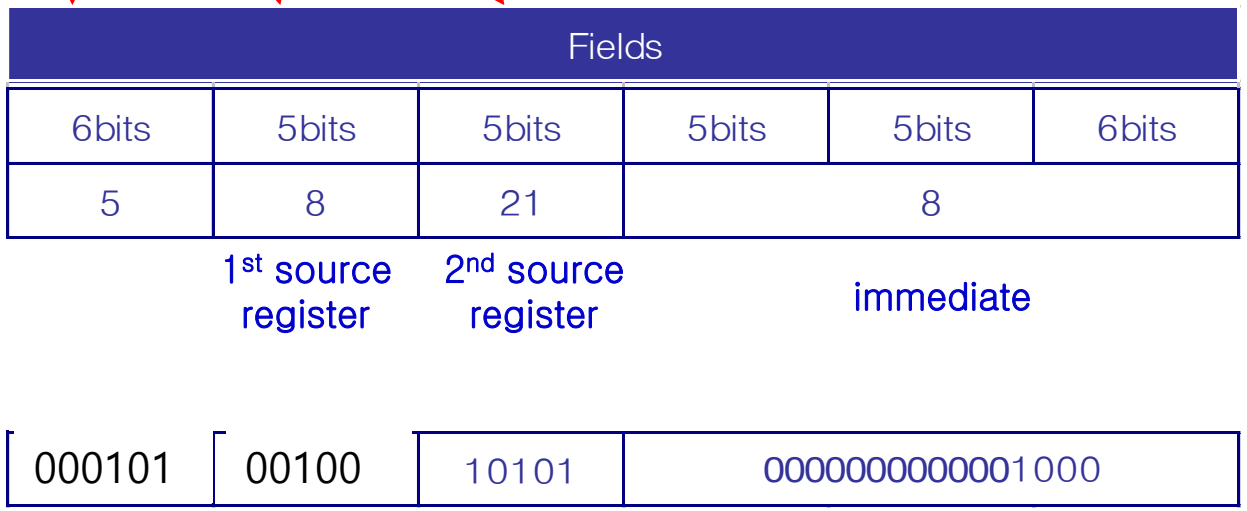
immediate

Instruction Format (V)

- **I-Format 예제**
  - 아래와 같은 bne 명령어가 있을 수 있음

```
bne $t0, $s5, Exit          # goto Exit if $t0 != $s5
```

- **각 Field를 아래와 같이 채울 수 있음**



Instruction Format (VI)

- **J-Format**
  - 각 명령어를 위해 다른 Opcode를 사용함
  - j와 jal 명령어를 위해 사용됨
  - 먼 지점으로의 jump를 위해 절대 주소가 포함되어 있음
  - 절대 주소는 Word단위로 되어 있음 (Target * 4)

Name	Fields					
Field size	6bits	5bits	5bits	5bits	5bits	6bits
J-format	op	target address				

MIPS 명령어

- Arithmetic 명령어
- Data Transfer (Load/Store) 명령어
- Conditional Branch 명령어
- Unconditional Jump 명령어
- Logical 명령어

MIPS 명령어 Format

Name	Fields						Comments
Field size	6bits	5bits	5bits	5bits	5bits	6bits	All MIPS insturctions 32 bits
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

이전의 예를 다시 살펴보자 (산술 연산)

- **같은 코드를 살펴보자**

  a = b + c

  – 변수들은 각각 $s0, $s1, $s2에 저장되어 있다고 가정

- **Register를 활용하는 Add 명령어**

  – add $s0, $s1, $s2 # a = b + c

- **a = b – c를 계산하기 위한 Sub 명령어**

  – sub $s0, $s1, $s2 # a = b - c

Fields					
6bits	5bits	5bits	5bits	5bits	6bits
op	rs	rt	rd	shamt	funct
	1st source register	2nd source register	result register	shift amount	function code

이전의 예를 다시 살펴보자 (복잡한 산술 연산)

- **좀 더 복잡한 연산은?**

  a = b + c + d - e

- **여러 개의 명령어로 표현 가능**

```
add $t0, $s1, $s2          # t0 = b + c
add $t1, $t0, $s3          # t1 = t0 + d (= b + c + d)
sub $s0, $t1, $s4          # a = t1 – e (= b + c + d – e)
```

MIPS 정수형 산술연산 명령어

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1, $s2, $s3	$s1 = $s2 + $s3	3 operands; exception possible
	add immediate	addi $s1, $s2, 100	$s1 = $s2 + 100	+ constant; exception possible
	add unsigned	addu $s1, $s2, $s3	$s1 = $s2 + $s3	3 operands; no exceptions
	add immediate unsigned	addiu $s1, $s2, 100	$s1 = $s2 + 100	+ constant; no exceptions
	subtract	sub $s1, $s2, $s3	$s1 = $s2 – $s3	3 operands; exception possible
	subtract unsigned	subu $s1, $s2, $s3	$s1 = $s2 – $s3	3 operands; no exceptions
	set less than	slt $s1, $s2, $s3	$s1 = ($s2 < $s3)	compare signed <
	set less than immediate	slti $s1, $s2, 100	$s1 = ($s2 < 100)	compare signed < constant
	set less than unsigned	sltu $s1, $s2, $s3	$s1 = ($s2 < $s3)	compare unsigned <
	set less than immediate unsigned	sltiu $s1, $s2, 100	$s1 = ($s2 < 100)	compare unsigned < constant
	multiply	mult $s2, $s3	HI, LO ← $s2 x $s3	64 bit signed product
	multiply unsigned	multu $s2, $s3	HI, LO ← $s2 x $s3	64 bit unsigned product
	divide	div $s2, $s3	LO ← $s2 ÷ $s3,	LO ← quotient
			HI ← $s2 mod $s3	HI ← remainder
	divide unsigned	divu $s2, $s3	LO ← $s2 ÷ $s3,	Unsigned quotient
			HI ← $s2 mod $s3	Unsigned remainder
	move from HI	mfhi $s1	$s1 ← HI	Used to get copy of HI
	move from LO	mflo $s1	$s1 ← LO	Used to get copy of LO