



Chapter02_Language of the Computer

Instruction

- **프로세서의 동작을 묘사하는 기본 명령어**이다. 하드웨어와 소프트웨어 사이를 연결하는 인터페이스 역할을 한다.
- 명령어는 Opcode(Operation code)와 Operands (Operation을 위한 데이터)로 이루어져 있다. 예를 들어, 어셈블리어 `add x2 x4 x2` 중 Opcode는 `add`, `x2`, `x4`, `x2` 는 Operands다.

Opcode (operation code)
Operands: data for the operation



Assembly language representation

add \$2, \$4, \$2
opcode operand

Binary language representation

00000000000110000011000001000001

명령어 집합(Instruction Set)

- 명령어 집합(Instruction Set), 또는 명령어 집합 구조(Instruction Set Architecture, ISA)는 주어진 하드웨어에서 실행될 수 있는 모든 명령어들의 집합을 말한다.
- 하드웨어와 소프트웨어 사이에서 Interface 역할을 한다.
- 서로 다른 프로세서라 하더라도, 명령어 집합은 비슷하게 생겼다. 물론 엄밀히 따지자면 다르게 생기긴 했지만, 하드웨어 기술이나 기존의 원리 등이 비슷하기 때문에 각 프로세서마다 명령어 집합도 거의 비슷하게 생겼다. 때로는 고유의 명령어가 추가되기도 한다.
 - 비슷한 하드웨어 기술.
 - 비슷한 규칙을 기반으로 한다.
 - 컴퓨터에서 필요로 하는 모든 기술들의 집합.
- 유효한 명령어 집합(5) : ARMv7, ARMv7, Intel x86, POWER, AMD74

Microarchitecture

- ISA를 프로세서가 실행할 수 있도록 물리적으로 구현한 것을 마이크로 아키텍쳐라고 한다.
- 같은 ISA를 서로 다른 마이크로 아키텍쳐로 구현하기도 한다.
- 마이크로 아키텍쳐의 구현은 ISA를 충족시킨다.
- x86 ISA 구현 마이크로 아키텍처 종류 : 186, 386, 486, 펜티엄 등이 있다.
- 파이프 라이닝(Pipelining), 예측 실행, 메모리 접근 스케줄링 등 소프트웨어에 노출되지 않고 하드웨어에서 수행되는 모든 작업을 정한다.

CISC vs RISC

- CISC(Complex Instruction Set Computer)
 - 오래된 디자인이다.(x86 instruction set is CISC)
 - (Multi-clock) Complex and variable length instructions
 - 명령어가 복잡하며, 길이가 가변적이다.
 - 예를 들어, 두 인자를 더할 때는 add xA, xB 와 인자 세 개를 더할 때는 add xA, xB, xC 라고 할 수 있다. 이 때, 당연히 두 개의 숫자를 더하는 것보다 세 개의 숫자를 더하는 계산이 더 오래 걸릴 것인데, 같은 명령어지만, 인자의 개수에 따라 CPU clock도 달라서 Multi-Clock라고 한다.

- 많은(강력한) 명령어가 ISA에 있다.
 - 하나의 명령어가 다양한 기능을 할 수 있다.
 - 예를 들어, LOAD 와 STORE가 같은 instruction 내 명령어에 병합되어 있다.

CISC의 장점과 단점

장점과 단점에 대해 알아보도록 하자.

장점

- 많은 강력한 명령어 덕분에 어셈블리 프로그래밍이 더 쉽다.
60, 70년대에는 어셈블리 프로그래밍이 많았다.
- 컴파일러 또한 명령어가 강력한 덕분에 **단순하다**.
- 한 프로그램이 적은 메모리를 차지한다.
 - 강력한 명령어가 많기 때문에 프로그램을 구현하는데 있어서, 더 적은 명령어가 사용된다.

단점

- 명령어가 많고 복잡하기 때문에, CPU설계가 끔찍하게 어렵다.
- 따라서 이런 단점에 의해, 구형 설계에서 새로운 필요 사항이나 요구들을 충족 시키기 위해 명령어들을 추가해오며 점점 복잡해지자, 명령어를 짧고 단순하게 만들기 위해 **RISC**가 등장했다.

RISC (Reduced Instruction Set Computer)

- 최근에 등장한 개념이다.
- 명령어가 단순하며, 표준화 되었다.
- ARM, MIPS, POWER, RISC-V 가 RISC에 속한다.
- 명령어 집합이 작으며, CISC에 속하는 명령어 또한 여러 개의 RISC 명령어로 처리가 가능하다.

장점

- CPU 설계가 쉬워졌다.
- **명령어 집합이 작기 때문에 clock 속도가 빠르다.**

- 각 명령어가 한 clock에 실행되도록 고정되어 있다.

단점

- 어셈블리 프로그래밍이 어려워졌다.
 - 컴파일러 설계가 어렵기 때문이다.
- 한 프로그램이 많은 메모리를 차지한다.
 - CISC에서 하나의 명령어로 할 수 있는 일을 RISC의 여러 개의 명령어로 구현해야 하기 때문에, 하나의 프로그램에 많은 명령어가 사용될 수 밖에 없다.
- 대부분의 현대 x86 프로세서들은 RISC 기술로 구현되었다.

$$\text{CPU time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}}$$

CISC Approach: Expecting smaller inst count, but CPI increases.
 RISC Approach: Trying to reduce CPI, but inst count increases.

RISC-V

- RISC-V는 버클리대학에서 개발한 5번째 RISC 기반의 open standard ISA다. 학술 및 산업 분야에서 무료로 사용할 수 있도록 완전히 오픈 돼 있다.
- RISC-V는 단순하며, 80x86 ISA보다 더 강력하다.

Arithmetic Operation

- Notation
 - add a, b, c #a = b+c
- RISC-V 아키텍처는 무조건 3개의 operands를 가져야 한다. 왜? 디자인이 간단하니깐.
- 만약, 4개의 더하는 값을 알려면, b,c,d 그리고 e 를 a에 대입해야 된다.
 - `add a, b, c //a = b+c`
 - `add a, a, d // a = a+d+b`
 - `add a, a, e // a = b+c+d+e`
 - → Three instructions are required
- Design principle

- Simplicity favors regularity.
 - 구현은 점차 쉬워진다.
 - 수행능력의 장점의 잠재적 능력

Arithmetic Example

- C code

```
f = (g+h) - (i+j);
```

- Complied Assembly code

```
add t0, g, h #temp t0 = g+h
add t1, i, j #temp t1 = i+j
sub f, t0, t1 #f = t0-t1
```

- 하드웨어 설계 원칙에는, “간단해지기 위해선, 같은 형태를 사용하는 것이 좋다.”라는 말이 있다. RISC-V의 명령어는 크기가 일정하고, 산술 연산이 동일한 형태를 가지는 등이 이 원칙을 따르는 것이다.

Register Operands

- Operands of arithmetic instructions must be registers(산술 명령어의 피연산자는 레지스터여야 합니다.)
 - Register: storage inside CPU (레지스터: CPU 내부 스토리지)
- Register size of RISC-V : 64 bits (doubleword)
 - Number of registers : 32
- RISC-V notation convention for expressing registers
 - (x + register number) : x0, x1, x2, ...

C code:	Compiled RISC-V code:
<code>f = (g+h)-(i+j);</code>	<code>add x5, x20, x21 add x6, x22, x23 sub x19, x5, x6</code>

Memory Operands

- Registers are limited, and data is usually larger. (레지스터는 제한돼 있지만, data의 크기는 크다.)
 - Arithmetic must be done on registers.
 - Therefore, we need ‘data transfer instructions’
 - To/From memory and registers
- Copying data from memory to register: load
 - Address: Location of data element within a memory array
 - ld : load doubleword
 - format: `ld x2 16(x3)`

Endian

- Big-endian vs little-endian
 - 빅 엔디안에서 가장 중요한 시사점은 어떤 byte를 0x 이후에 처음 나오는 숫자부터 낮은 주소 배열에 넣는 것이다.
 - Number: 0x1234ABCD

Big-endian	12	34	AB	CD
0	1	2	3	

- 리틀 엔디안에서 가장 중요한 시사점은 어떤 byte를 0x 이후에 나오는 숫자에서 맨 뒷 자리 숫자부터 차례대로 낮은 주소에 위치시키는 것이다.



RISC-V는 “little-endian” 방식을 채택하고 있다.

Memory Operands

- Store instruction
 - sd : store doubleword

- format: `sd x9, 96(x22)`
- Base address of the array A is in x22(배열 A의 기본 주소는 x22이다.)
- h는 reg x21 과 연결된다.

C code:

`A[12] = h + A[8];`

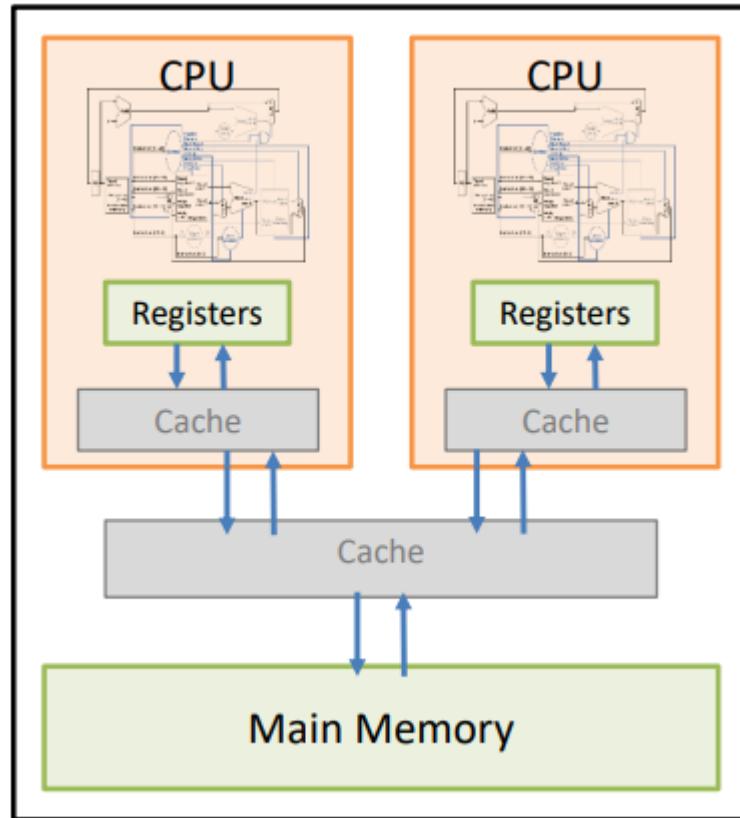


Compiled RISC-V code:

`ld x9, 64(x22)
add x9, x21, x9
sd x9, 96(x22)`

Register vs Memory

- Access speed
 - Register > Memory
- Capacity
 - Register < Memory
- Energy consumption
 - Register < Memory
- 프로그램은 보통 레지스터의 수 (RISC-V에는 32개)보다 많은 수의 변수를 생성한다. 그렇기 때문에 모든 변수를 레지스터에 담을 수 없으므로 **자주 사용하지 않는 데이터는 메모리에 저장을 한다. 이를 Spilling 이라고 한다.**
- 반대로 자주 사용하는 데이터는 레지스터에 저장하는 것이 성능이 좋을 것이다. 이와 같은 것은 컴파일러가 처리된다. 왜냐하면, 메모리 측면에서 레지스터가 더 유리하기 때문이다.(컴파일러 시에 적합한 실행 능력을 시도한다.)



Constant OR Immediate operands

- Arithmetic operation that uses a constant number(산술 연산은 상수 숫자에서 사용된다.)
 - ex. 4라는 변수를 더한다고 가정해 보자.
 - Option1: 메모리가 4를 어딘가에 저장하고 있다. 이것은 반드시 첫번째에 로드된다.

```

var1: .dword 4
...
ld x9, var1(x3)    # x3 + var1's offset
add x22, x22, x9

```

- Option2: 상수 연산자를 사용한다.
 - “add immediate”, addi

```
addi x22, x22, 4
```

- Immediate operand instructions are faster and use less energy
- 4 is included inside the instruction bits

Constant Zero

- Register x0
 - 0으로 값이 고정되어 있는 레지스터이다.
 - Overwritten 할 수가 없다. (modified)
- Useful for common operations
 - E.g., move data between registers
 - add x22, x21, x0 #x22 ← x21
 - E.g., negate the value
 - sub x22, x0, x22 #x22 = -x22

Signed and Unsigned Numbers

- Numbers in base2 (binary numbers)
 - ex) 123 base 10 = 1111011 base 2
- Value of ith digit d: $d \times \text{Base}^i$

$$\begin{aligned}1101_{\text{two}} &= (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \\&= 8 + 4 + 0 + 1 = 13_{\text{ten}}\end{aligned}$$

- 1101 in a doubleword

63	55	47	39	32
0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0
31	23	15	8	0
0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	1 1 0 1

- LSB (least significant bit) the right most bit, bit position 0
- MSB (most significant bit) the left most bit, bit position 63

- LSB: 가장 오른쪽에 있는 bit가 0이다. 즉, 가장 높은 주소자리 부터 값이 채워진다.
- MSB: 가장 왼쪽에 있는 bit가 63번째이다. 즉, 가장 낮은 주소자리 부터 값이 채워진다.

Unsigned Numbers

- RISC-V word can represent 2^{64} bit patterns

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 _{two} = 0 _{ten}
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001 _{two} = 1 _{ten}
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000010 _{two} = 2 _{ten}
...
11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111101 _{two} = 18,446,774,073,709,551,613 _{ten}
11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111110 _{two} = 18,446,744,073,709,551,614 _{ten}
11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 _{two} = 18,446,744,073,709,551,615 _{ten}

- Decimal values calculated using:

$$(x_{63} \times 2^{63}) + (x_{62} \times 2^{62}) + (x_{61} \times 2^{61}) + \dots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

→ representation of the unsigned numbers

- - - - -

- 표현의 한계
 - 컴퓨터는 오로지 bit 배열로만 숫자를 표현 할 수 있다.
 - 만약, 연산 기호 결과가 표현하고자 하는 것보다 크기가 커진다면 overflow 가 발생 한다.

Signed Numbers

- Expressing the sign of a number(숫자의 부호를 표현하는 방법)
 - 양수 또는 음수 기호를 나타내는 1 비트
 - 부호 비트에 어디에 나타나야 하며, 이것이 산술 연산에 어떻게 영향을 미치게 될까?

Naïve solution

- Simply use the MSB as a sign → “*sign and magnitude*” representation

$0000_2 = 0_{10}$	$1000_2 = -0_{10}$
$0001_2 = 1_{10}$	$1001_2 = -1_{10}$
$0010_2 = 2_{10}$	$1010_2 = -2_{10}$
$0011_2 = 3_{10}$	$1011_2 = -3_{10}$
$0100_2 = 4_{10}$	$1100_2 = -4_{10}$
$0101_2 = 5_{10}$	$1101_2 = -5_{10}$
$0110_2 = 6_{10}$	$1110_2 = -6_{10}$
$0111_2 = 7_{10}$	$1111_2 = -7_{10}$

$6 - 3 = 0110 + 1011 = ???$

■ 1's complement

- Invert positive number bits to use as a negative number

$0000_2 = 0_{10}$	$1111_2 = -0_{10}$
$0001_2 = 1_{10}$	$1110_2 = -1_{10}$
$0010_2 = 2_{10}$	$1101_2 = -2_{10}$
$0011_2 = 3_{10}$	$1100_2 = -3_{10}$
$0100_2 = 4_{10}$	$1011_2 = -4_{10}$
$0101_2 = 5_{10}$	$1010_2 = -5_{10}$
$0110_2 = 6_{10}$	$1001_2 = -6_{10}$
$0111_2 = 7_{10}$	$1000_2 = -7_{10}$

$6 - 3 = 0110 + 1100 = 0010 = 2 ???$

carry

■ 2's complement

$0000_2 = 0_{10}$	$1111_2 = -1_{10}$
$0001_2 = 1_{10}$	$1110_2 = -2_{10}$
$0010_2 = 2_{10}$	$\textcolor{red}{1101}_2 = -3_{10}$
$\textcolor{blue}{0011}_2 = 3_{10}$	$1100_2 = -4_{10}$
$0100_2 = 4_{10}$	$1011_2 = -5_{10}$
$0101_2 = 5_{10}$	$1010_2 = -6_{10}$
$0110_2 = 6_{10}$	$1001_2 = -7_{10}$
$0111_2 = 7_{10}$	$1000_2 = -8_{10}$

$6 - 3 = 0110 + \textcolor{red}{1101} = \textcolor{green}{0011} = 3$

- 1의 보수 : 0을 1로, 0을 1로 바꾼다

- 1을 더해줘야 하는 carry 문제가 발생한다.

- 2의 보수: 1의 보수에 1을 더한다.

- 연산 결과에 추가적인 연산을 할 필요가 없고, 공간의 낭비도 없기 때문에 컴퓨터에서는 음수를 표현하기 위해 2의 보수를 사용한다.
- 또한 2의 보수 방법을 사용하면, 모든 음수가 MSB가 1이기 때문에, 컴퓨터가 어떤 수가 음수인지 판단하기 위해 한 비트만 판단하면 된다.
- 2's Complement
 - 모든 음수 표기방법에서 1은 음수를 나타내고 이 비트의 위치는 MSB로 표현된다.
 - 숫자가 양수인지 음수인지 확인하려면 하드웨어에서 단 한 비트만 확인하면 된다.

Computing the decimal value

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Negating 2's complement binary number
 - 1의 보수에 1을 더한다.

$0000_2 = 0_{10}$	$1111_2 = -1_{10}$	$2 \rightarrow -2: 0010 \rightarrow 1101 \rightarrow \text{add } 1 \rightarrow 1110$
$0001_2 = 1_{10}$	$1110_2 = -2_{10}$	
$0010_2 = 2_{10}$	$\textcolor{red}{1101}_2 = -3_{10}$	$-2 \rightarrow 2: 1110 \rightarrow 0001 \rightarrow \text{add } 1 \rightarrow 0010$
$\textcolor{brown}{0011}_2 = 3_{10}$	$1100_2 = -4_{10}$	

- Sign extension(부호의 확장)
 - n 비트 크기의 이진수를 m 비트로 늘리고 싶은 경우, MSB 비트를 가져와 나머지 비트를 MSB 와 같은 비트로 채우면 된다. 예를 들어서,
 - $0000\ 0010 = 2$
 - MSB는 0이다.
 - $0000\ 0000\ 0000\ 0010 = 2$
 - $1111\ 1110 = -2$
 - MSB는 1이다.
 - $1111\ 1111\ 1111\ 1110 = -2$
 - Signed load 가 완벽하게 값을 load 하기 위한 명령어다. 예를 들어, 16비트 숫자를 32비트 레지스터로 로딩하는 경우 자동으로 부호 확장을 해준다.
 - `lb : Load byte`

- lh : load halfword
- lw : load word
- 부호 확장을 원하지 않으면, lbu (load byte unsigned) 을 사용하면 된다.

명령어 표현(Representing instructions)

명령어의 형태에 따라 분류를 할 수 있는데, RISC-V 는 다음과 같이 명령어를 분류하고 있다.

1. RISC-V R-type instruction format(32bit)

a. 아래 그림은 R-type instruction 의 형태이다.

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- opcode : operation의 기반이다. 즉, 더하기냐 빼기이냐를 알려주는 코드다.
- rd: register의 목적지이다.
- funct3: opcode를 더하는 것이다.
- rs1: 첫 번째 레지스터
- rs2: 두 번째 레지스터
- funct7: opcode 를 더하는 것이다.

add x9, x20, x21 있다. 밑에 표는 이것을 나타내기 위해 주어진 값이다 .add와 sub 값이 들어있다. 이것을 참고해서 R-type 형태를 채워보자.

Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
add (add)	R	0000000	reg	reg	000	reg	0110011
sub (sub)	R	0100000	reg	reg	000	reg	0110011

0	21	20	0	9	51
---	----	----	---	---	----

0000000	10101	10100	000	01001	0110011
---------	-------	-------	-----	-------	---------

2. RISC-V I-type instruction format

- immediate: immediate 명령어를 위한 값이다. 2의 보수가 값으로 오기 때문에, 범위가 $-2^{11} \sim 2^{11}$ 이다.
- rs1: 첫 번째 즉, 출발 레지스터이다.
- opcode, funct3: 명령어의 opcode, 두 비트를 모두 합쳐서 사용한다.
- rd: 목적지 (destination) 레지스터이다.
- I-type 명령어의 예시로는 addi, ld 등이 있다.
- 밑에 그림은 I-type의 형태를 나타낸 것이고, addi와 ld가 있는데 각각의 정보를 값으로 표현해 준 표가 있다.

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3bits	5bits	7bits

Instruction encoding information

Instruction	Format	immediate	rs1	funct3	rd	opcode
addi (add immediate)	I	constant	reg	000	reg	0010011
ld (load doubleword)	I	address	reg	011	reg	0000011

ld x14, 8(x2)

000000001000	00010	010	01111	0000011	← Find the error!
--------------	-------	-----	-------	---------	-------------------

- ld x14, 8(x2) 가 있는데, 여기서 error를 찾아 보라고 한다. 우선, funct3의 값을 011로 고쳐야 한다. 그리고 rd값이 14가 되어야 하는데 이진수 01111은 15다. 따라서 14를 이진수로 알맞게 변환한 값이 들어 가야한다.

3. RISC-V S-type instruction format

- 두 개의 레지스터와 한 개의 immediate 영역으로 구성되어 있다.
- store를 해주는 instruction이다.
- 두 개의 immediate 영역: 7bits+5bits
- rs1 그리고 rs2는 항상 모든 instructions에서 같은 위치에 있다.
- 밑에는 RISC-V의 format이다.

immediate	rs2	rs1	funct3	immediate	opcode
7 bits	5 bits	5 bits	3bits	5bits	7bits

sd x14, 8(x2)

0000000		00010	011	01000		← Fill in the blank
---------	--	-------	-----	-------	--	---------------------

- **sd x14, 8(x2)** 를 S-type format으로 나타냈고, 빈칸을 채워보라고 한다.

Instruction encoding information

Instruction	Format	immed-iate	rs2	rs1	funct3	immed-iate	opcode
sd (store doubleword)	S	address	reg	reg	011	address	0100011

위의 표를 이용해 채워보면, rs2가 들어가야 할 즉, 왼쪽에서 첫 번째 빈칸에는 8의 이진수가 들어가고, 오른쪽 첫 번째 빈칸에는 opcode가 들어가야한다.

Logical Operations

Logical operations	C operators	Java operators	RISC-V instructions
Shift left	<<	<<	sll, slli
Shift right	>>	>>	srl, srli
Shift right arithmetic	>>	>>	sra, srai
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR			or, ori
Bit-by-bit XOR	^	^	xor, xorri
Bit-by-bit NOT	~	~	xori

- 논리 연산 기호를 표로 정리해서 나타낸것이다.
- Shift operation
 - 아래는 9의 bit를 왼쪽으로 5만큼 이동시킨 값이다. 이 값을 10진수로 변환한 값이 ?에 들어가게 된다.

00000000 00001001 = 9
shift left 5 00000001 00100000 = ?

- Shift instructions(I-type_)
 - slli, srli → 왼쪽으로, 오른쪽으로 옮겨주는 논리 immediate다.
 - sra, srai → shift, right arithmetic(immediate): 부호 비트(MSB bit)로 빈 칸을 채운다.
 - 즉, 1100 0011 을 sra하면 쉬프트 연산 후 생기는 빈 공간을 1로 채워서 1110 0001이 된다. 그러니깐 요약해 보면, r이 오른쪽이란 뜻이니깐 오른쪽으로 비트를 옮기면 원래는 0110 0001 이 되야하는데 앞의 MSB쪽에서 발생한 0을 다 1로 바꿔준다는 의미인 것이다.

AND/OR Instructions

- and, andi
 - add x9, x10, x11 이라는 식이 있다. 이는, x10 reg와 x11 reg를 and연산을 한 후, reg x9에 저장하는 것이다.

	and x9, x10, x11 // reg x9 = reg x10 & reg x11	
x10	<table border="1"><tr><td>00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000</td></tr></table>	00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000		
x11	<table border="1"><tr><td>00000000 00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000</td></tr></table>	00000000 00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000		
x9	<table border="1"><tr><td>00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000</td></tr></table>	00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000		

- or , ori
 - add x9, x10, x11 이라는 식이 있다. 이는, x10 reg와 x11 reg를 or연산을 한 후, reg x9에 저장하는 것이다.

	or x9, x10, x11 // reg x9 = reg x10 reg x11	
x10	<table border="1"><tr><td>00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000</td></tr></table>	00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000		
x11	<table border="1"><tr><td>00000000 00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000</td></tr></table>	00000000 00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000		
x9	<table border="1"><tr><td>00000000 00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000</td></tr></table>	00000000 00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000		

XOR Instructions

- xor
- xor x9, x10, x12 // 이는 x10 xor x12 → x9 을 의미한다.

xor x9, x10, x12 // reg x9 = reg x10 ^ reg x12

x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x12	11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
x9	11111111 11111111 11111111 11111111 11111111 11111111 11110010 00111111

Truth Table for XOR

Input 1	Input 2	Output
0	1	1
1	1	0
0	0	0
1	0	1

- input1 과 input2가 같은 값이면 0, 서로 다른 값이면 1을 반환한다.

Instructions for Making Decisions

- Branch instructions
 - if statement
 - beq rs1, rs2, L1
 - beq: “branch if equal”
 - Label L1으로 가는 경우는 rs1 == rs2 인 경우다.
 - bne rs1, rs2, L1
 - bne: “branch if not equal”
 - beq, bne: conditional branches (조건 분기문)

Compiling if-then-else into Conditional Branches

- C code segment

```
if (i==j)
    f = g + h;
else
    f = g - h;
```

- Variable to register assignments
 - f : x19, g : x20, h: x21, i : x22, j : x23
- RISC-V assembly code

```
bne x22, x23, Else # go to Else if i ≠ j
add x19, x20, x21 # f = g + h (skipped if i ≠ j)
beq x0, x0, Exit   # if 0 == 0, go to Exit
Else: sub x19, x20, x21 # f = g - h (skipped if i = j)
Exit:
```

Compiling a while loop

- C code segment

```
While (save[i]==k)
    i += 1;
```

- Variable to register assignments
 - i : x22, k : x24, base of save: x25
 - while 반복문에서 만약 save[i] 가 k값과 같으면 i+=1 을 수행한다.

■ RISC-V assembly code

Text after # is comment.

```
Loop: slli x10, x22, 3      # x10 = i × 8
      add x10, x10, x25 # x10 = address of save[i]
      ld x9, 0(x10)      # x9 = save[i]
      bne x9, x24, Exit  # go to Exit if save[i] ≠ k
      addi x22, x22, 1   # i = i+1
      beq x0, x0, Loop   # go to Loop
Exit:
```

- reg x10에 ix8 값을 저장한다. i를 왼쪽으로 3번 옮겼으므로, 2^3 이므로 8이다. 그럼 메모리 주소상에서 한 칸 옮겨진거다.
- 그럼 돌아와서 base of save라는 것이 메모리의 시작 시점에 있다. 당연히 애도 메모리 스택상에 있으므로, 값이 있다. 그렇다면 아까 위에서 ix8 해서 옮겨간 x10이 가지고 있는 값이 있겠지, 이 값을 base of save에 더한다. 그 다음 x10에 그 값을 저장하는 것이다.

Other Conditional Branches

- `blt: "branch if less than"`
 - `blt rs1, rs2, L1`
 - rs1과 rs2의 값을 비교해본다,(2의 보수법을 취한 숫자이다.)
 - Branch if rs1 is smaller , 즉, rs1이 더 작으면 L1으로 간다.
- `bge: "branch if greater than or equal"`
 - `bge rs1, rs2, L1`
 - Branch if rs2 is smaller
- `bltu, bgeu`
 - 부호 없는 숫자들이 레지스터에 저장되어 있다.

Array Index Out-of-bound check

- Array index check

```

int[] age = new int[5];
age[-2] = 12;
age[2] = 4;
age[12] = 4;

```

- Need to check if index > 0 and index ≤ max_length
- bgeu can be used to check both conditions(bgeu는 두 조건을 모두 확인하는데 사용된다.)

Text after # is comment.

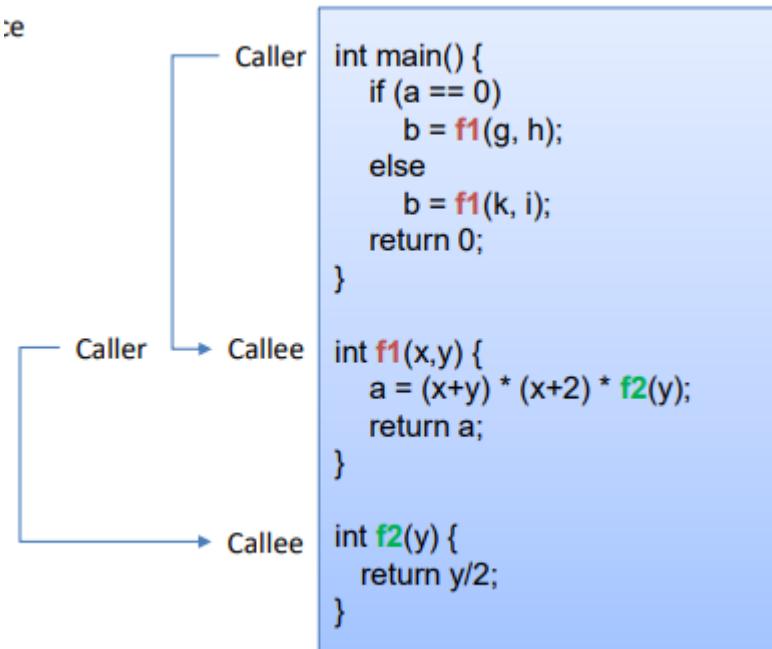
```

# if x20 >= x11 or x20 < 0, goto IndexOutOfBoundsException
bgeu x20, x11, IndexOutOfBoundsException

```

Supporting Procedures

- Procedure (function) calls → 실행하는 것을 불러주는 것(실행이면, 함수를 의미)



아래의 5단계는 함수 호출 과정이다.

1. Pass function parameters : 파라미터(값)을 넘겨주는 함수다
2. Transfer control : 실행 지점을 호출한 함수로 옮긴다.
 - Acquire necessary memory space : 필요한 메모리 공간을 요구한다.
3. Run procedure : 함수를 실행한다.
4. Store return value (result) to predefined location : 저장했던 값을 반환해주고, 반환한 값을 지정된 위치에 저장한다.
5. Return control : 다시 원래 위치로 돌아온다.
 - 이 때, 호출하는 함수를 Caller, 호출 되는 함수를 Callee 라고 하며, Caller-Callee 관계는 항상 바뀔 수 있다.
 - jalr: “jump and link register” instruction(I-type)

jalr x0, 0(x1)

- x1에 저장된 주소로 분기한다. 즉, x0은 무조건 0을 return 하기 때문에, 저렇게 x0이면 x1로 간다는 것은 x1이 분기로 가라는 뜻이다.
- x0은 0으로 연결되어 있다. x0에 쓰는 것은 버리는 효과가 있다고 한다.(예를 들어, return)

Using More Registers

- x10~x17은 값을 저장하고 넘겨주는 레지스터이다. 하지만, 프로그램을 수행하다 보면 이 8개의 레지스터보다 더 필요한 경우가 많다.
 - 그래서 컴파일러는 다른 추가 레지스터를 사용하도록 선택하였다.
 - 레지스터의 값을 메모리로 저장한다. → spill
 - 실행이 끝난 후 값을 저장한다.
- 레지스터를 spilling하기 위한 이상적인 방법에는 “스택”이라는 것이 있다.
 - 메모리 상에서 맨 마지막에 들어온 것을 넣고 빼낼 때는 마지막에 넣은 것을 빼낸다.
 - 메모리 영역은 레지스터를 spilling하기 위한 것이다.
 - 스택은 높은 주소에서 낮은 주소로 자란다.

- 스택 포인터는 8바이트 크기로 맞춰진다. 이것은 레지스터를 저장하고(push) 값을 반환하기 위함이다.(pop)

Compiling a Leaf Procedure

- Leaf procedure
 - Procedure that does not call other procedure(함수가 실행 중일 때는 다른 함수를 실행하지 않는다.)
 - 아래와 같은 코드가 있다. 이를 어셈블리 언어로 바꿔보면 밑의 코드와 같다.

```

long long int
leaf_example(
    long long int g,
    long long int h,
    long long int i,
    long long int j)
{
    long long int f
    f = (g + h) - (i + j);
    return f;
}

// g → x10, h → x11
// i → x12, j → x13
// f → x20

```

```

leaf_example:
    addi sp, sp, -24
    sd x5, 16(sp)
    sd x6, 8(sp)
    sd x20, 0(sp)
    add x5, x10, x11 #g : x10, h: x11
    add x6, x12, x13 # i : x12, j : x13
    sub x20, x5, x6 # g+h : x5, i+j : x6
    addi x10, x20, 0 #x20 -> x10
    ld x20, 0(sp)
    ld x6, 8(sp)
    ld x5, 16(s0)
    addi sp, sp, 24
    jalr x0, 0(x1) # branch back to calling routine

```



- sd 는 doubleword(word $\times 2$)의 byte크기 만큼(스택은 8byte단위로 값을 저장 및 반환해 주기 때문에) store해 주는 것이다. 잘 보면, 16, 8, 0으로 총 세 개의 reg를 저장해 주는데, 값이 점차 작아진다. 즉, 메모리 상에서 High address에서 Low address로 sd 해 주는 것을 볼 수 있다.
- ld 는 load doubleword이다. 잘 보면, 0, 8, 16으로 숫자가 실행할 수록 점점 커지는 것을 볼 수 있다. 스택은 맨 나중에 들어간 순서대로 반환 할 때도 마찬가지로 맨 마지막에 들어간 값부터 반환해 주기 때문이다.

Register Saving Convention(레지스터 저장의 관습)

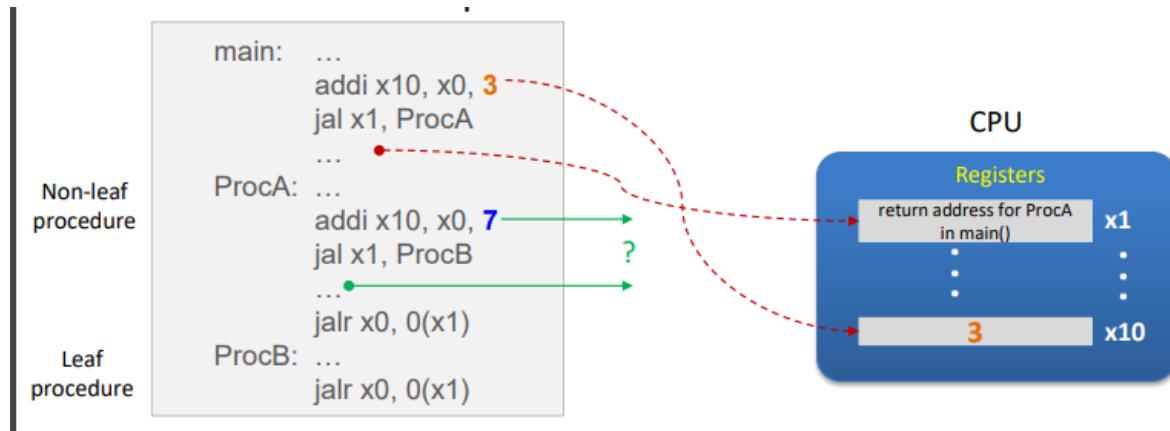
- Two groups of registers in RISC-V(RICS-V의 두 그룹이 있다.)
 - 첫 번째 그룹 : No need for callee to save
 - x5~x7, x28~x31 —> No need store
 - registers for temporary use
 - 두 번째 그룹: Callee must preserve(save, use and restore) if used
 - x8, x9, x18~x27 —> Need store

addi sp, sp, -24	// adjust stack to make room for 3 items
sd x5, 16(sp)	// save register x5 for use afterwards
sd x6, 8(sp)	// save register x6 for use afterwards
sd x20, 0(sp)	// save register x20 for use afterwards
add x5, x10, x11	// register x5 contains g + h
add x6, x12, x13	// register x6 contains i + j
sub x20, x5, x6	// f = x5 - x6, which is (g + h) - (i + j)
addi x10, x20, 0	// returns f (x10 = x20 + 0)
ld x20, 0(sp)	// restore register x20 for caller
ld x6, 8(sp)	// restore register x6 for caller
ld x5, 16(sp)	// restore register x5 for caller
addi sp, sp, 24	// adjust stack to delete 3 items
jalr x0, 0(x1)	// branch back to calling routine

- 첫 번째 그룹과 두 번째 그룹을 나눠 놓았는데, 함수를 사용 할 때, 즉, 보존이 필요 없는 레지스터와 보존이 필요한 레지스터로 나눈 것이다. 즉, 위에서 적었던, x5, x6은 sd, ld라는 레지스터를 보존하는 로직은 불필요했던 것이다.

Nested Procedure

- Issues with nested procedures



- Register Preservation Responsibility

Callee saves these	Caller saves these
Preserved	Not preserved
Saved registers: x8-x9, x18-x27	Temporary registers: x5-x7, x28-x31
Stack pointer register: x2(sp)	Argument/result registers: x10-x17
Frame pointer: x8(fp)	
Return address: x1(ra)	
Stack above the stack pointer	Stack below the stack pointer

Recursive Procedure

Preserved	Not preserved
Saved registers: x8-x9, x18-x27	Temporary registers: x5-x7, x28-x31
Stack pointer register: x2(sp)	Argument/result registers: x10-x17
Frame pointer: x8(fp)	
Return address: x1(ra)	
Stack above the stack pointer	Stack below the stack pointer

▪ Factorial

```
long long int fact (long long int n) {
    if (n < 1) return (1);
    else return (n * fact(n - 1));
}
```

a caller

- 재귀 함수 실행

```

        to the caller }
fact:
    addi sp, sp, -16 // adjust stack for 2 items
    sd x1, 8(sp)    // save the return address
    sd x10, 0(sp)   // save the argument n
    addi x5, x10, -1 // x5 = n - 1
    bge x5, x0, L1  // if(n-1) >= 0, goto L1
    addi x10, x0, 1 // return 1
    addi sp, sp, 16 // pop 2 items off stack
    jalr x0, 0(x1) // return to caller

L1:
    addi x10, x10, -1 // n>=1: argument gets (n-1)
    jal x1, fact      // call fact with (n-1)
    addi x6, x10, 0   // return from jal: move result of fact(n-1) to x6:
    ld x10, 0(sp)    // restore argument n
    ld x1, 8(sp)     // restore the return address
    addi sp, sp, 16 // adjust stack pointer to pop 2 items
    mul x10, x10, x6 // return n*fact(n-1)
    jalr x0, 0(x1) // return to the caller

```

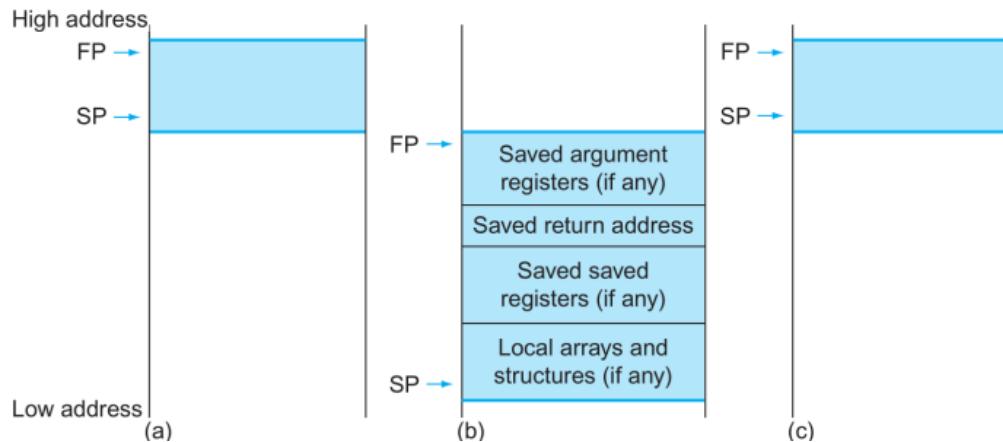
where to?

- fact라는 재귀함수가 있다. 숫자 n이 있을 때, n이 1보다 작으면, 1을 반환하고, n이 1보다 크면 n*fact(n-1)값을 리턴한다.
- 우선, 스택 포인터를 지정해준다. sp라는 스택 포인터를 지정해 준 후, -16을 해 준다. 그 후 return address인 x1을 sd하고, 숫자 n을 저장해 줄 reg x10을 sd한다.
- 그 후, add x10 + (-1) —> x5 하는 데 이건, n-1을 reg x5에 저장해 준다는 뜻이다.
- bge x5, x0, L1 # if (n-1) ≥ 0, goto L1 ##여기서 함수 Caller가 나온다. Callee는 L1이다.
- L1을 살펴보면, addi x10, x10, -1
- addi x10, x0, 1 → return 1
- addi sp, sp, 16 //아까 위에서 두 개 sd 한 것을 다시 더해준다.
- jalr x0, 0(x1) // Caller 함수를 반환해준다.

Procedure's Local Data on the Stack

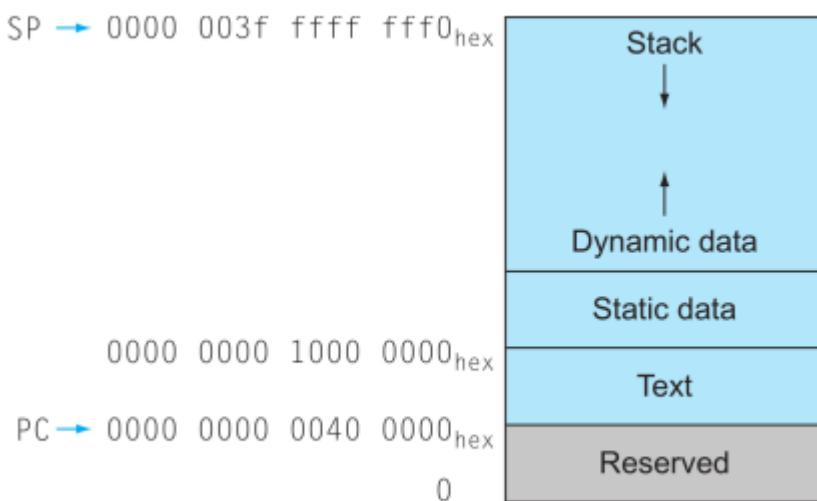
- 지역 변수는 프로시저 안에 포함되어있다.
 - 즉, 스택 안에 위치하고 있다.
- Procedure frame (= activation record)
 - 스택 안에 Segment는 다음과 같은 것들이 포함되어있다. 저장되어 있는 레지스터와 주소를 반환하는 것과 지역 변수들 말이다.

- Frame pointer, FP(x8) —> 프로시저 프레임의 첫 번째 단어를 가리킨다.
- SP 는 실행 중에 유동적으로 변할 수 있다.
- 파란색으로 표시된 부분을 activation record(procedure frame)이라고 부르며, 하나의 함수를 부를 때마다 activation record 가 할당된다. b에 나와있는 값들을 저장한다.



Allocating Memory on the Heap

- Address space convention
 - Address space: 주소의 정렬에는 사용되어지는 것들이 있는데, 할당하고, 참조하는 것들이 있다.
 - 이것은 가상의 메모리이므로 실제 물리적인 메모리는 아니다.



- Text segment

- text : text 영역에는 바이너리, 이진수 값이 들어온다. 우리가 하는 더하기 빼기 곱하기 등등의 모든 것들이 여기에 들어온다. E.g.) instructions, program binaries

- Static data segment

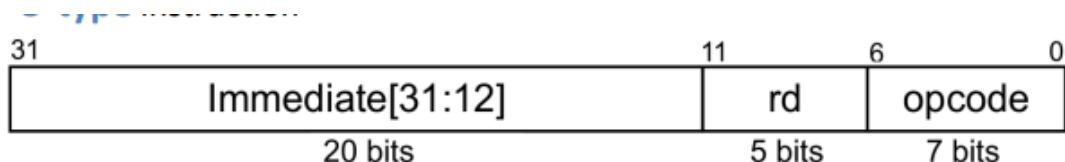
- 상수들 그리고 고정되어 있는 변수들이 온다. 예를 들어, 전역 변수나 static 변수 같은 것들, E.g.) constants and static variables

- Heap(for dynamic data)

- 메모리는 동적으로 할당되는 장소이다. 예를 들면, linked list, tree, ...
- 위로 자라난다.
- malloc(), free() ← C language
- new ← java
- Heap은 위쪽 방향으로 증가하고, Stack은 아래쪽 방향으로 증가한다.

Wide Immediate Operands

- 상수가 너무 12비트에 맞지 않는 경우가 있다.
- 이럴 때, RISC-V는 lui(Load Upper Immediate) instruction을 제공해준다.
 - U-type instruction



- 레지스터에서 비트 위치 31~12에 20비트의 상수를 로드한다.
- 왼쪽의 32비트는 31비트 값으로 확장된다.
- 맨 오른쪽의 12비트는 0으로 채워진다.
- How do we make this value in a register? 어떻게 이 값을 레지스터에 저장할까?

```
00000000 00000000 00000000 00000000 00000000 00111101 00000101 00000000
```

- lui x19, 976 //976 = 0000 0000 0011 1101 0000

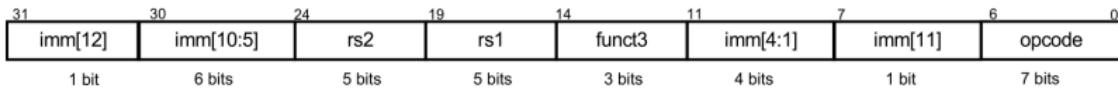
```
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

- addi x19, x19, 1280 //1280 = 0000 0101 0000 0000

```
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

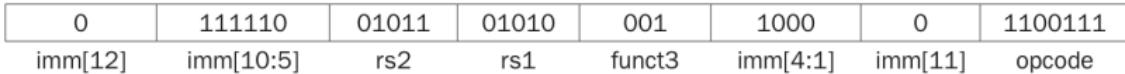
Branch Instruction Format

- SB-type



- Can represent branch addresses from -4096 to 4094 in multiple of 2 (i.e. even address)

```
bne x10, x11, 2000 // if x10 != x11, go to location 2000 ten = 0111 1101 0000 0
```



- SB-type 의 format이다. 분기문의 주소는 -4096부터 4094까지 2의 배수로 나타낼 수 있다.
- beq bge bne 등의 조건 분기 명령어는 SB-type 이다.

- imm 이 imm[12:0] 이 아닌 잘게 잘려 있는 모습을 볼 수 있는데, 이는 다른 명령어들과 형식을 맞추기 위함이다.

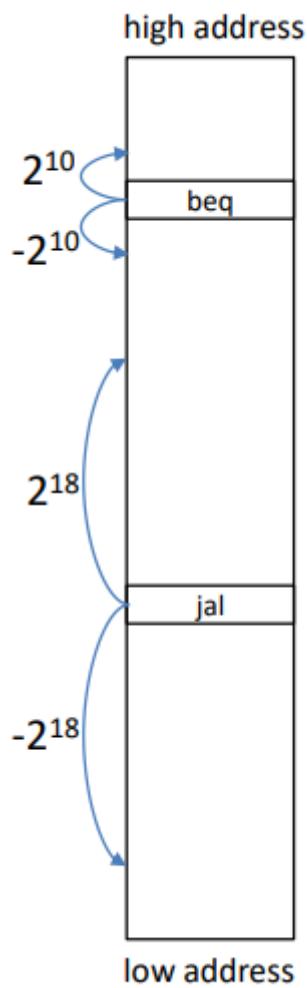
- imm[12] 는 다른 immediate 명령어와 부호 비트 위치를 맞추기 위해 맨 왼쪽에 존해한다.
- rs1, rs2 의 위치를 다른 명령어와 맞추기 위해 imm[10:5] 와 imm[4:1] 로 쪼갰다.
- 그런데 imm[0] 또한 없는 것을 볼 수 있는데, imm[0] 은 항상 0으로 여겨진다. → 2의 배수
- 그래서 immediate field 은 총 13비트다. → 2의 배수 중 -4096 ~ 4096 범위의 분기 주소(branch address)를 표현 할 수 있다.

- UJ-type

imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
1 bit	10 bits	1 bit	8 bits	5 bit	7 bits

- 무조건 분기 명령어 jal (jal은 이 유형의 유일한 명령어다.)
- imm[20:1] : 20비트 주소의 immediate이다.(부호 있는 숫자)
- jal 은 21bit의 주소를 사용한다.
- imm[0]은 항상 0으로 간주됩니다.

PC-relative Addressing



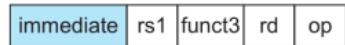
- bne, beq에 사용되는 분기문 용 주소 방식이다. 해당 명령어로부터 거리를 바탕으로 처리한다. E.g.) PC+4(address value x 4) 를 할 경우 분기 대상 위치 주소값이 된다. (x4는 마찬가지로 맨 뒤에 2비트를 생략하고 표현했다.)

- 조건 분기문의 immediate field는 13비트, 무조건 분기문의 immediate field는 21비트다.
- 이는 프로그램이 2^{imm} 보다 클 수 없다는 것을 의미하는데 이는 오늘날의 프로그램에게는 턱없이 부족하다.
- 해결책은 branch instruction의 offset에 추가로 특정 레지스터의 값을 더해 branch 하는 것이다.
 - Program counter = Register + Branch offset
 - 이는 프로그램 크기를 2^{64} 까지 늘려주며, 그대로 branch instruction 들을 사용할 수 있게 해준다.
- 그렇다면, 어떤 reg의 값을 더해야 하는가 → program counter register
- if statement 나 loop에서 conditional branch 들이 사용되는데, 보통 모든 conditional branch 들은 16instruction 이내로 이동하는 경향이 있다.
- 거의 모든 if statement와 loop는 2^{10} word보다 작기 때문에, Program counter 을 사용하는 것이 이상적인 선택이다.
 - 이를 통해 현재 instruction에서 $\pm 2^{10}$ 이내로 branch 할 수 있고, $\pm 2^{18}$ 이내로 jump를 할 수 있다.
- 이러한 addressing 방식을 **PC-relative addressing** 이라고 한다.
- 지금부터는 인터넷 검색을 통해, 공부를 한 내용이므로, 이해를 조금 돋기 위한 정도로만 생각하자.
 - 32bit Immediate Operands: I타입 인스트럭션으로, 뒤의 16비트 내용을, 목표 위치 레지스터 왼쪽부터 16비트에 넣는다.
 - J-type instruction format : 26bit 주소에 값을 넣는다.
 - Direct addressing mode
 - J 10000 #branch address = 10000
 - J 명령어의 경우 opcode 6bit를 제외한 26비트에 주소 값을 포함시켜야 한다.
 - Word addressing
 - Alignment restriction(명령어의 4비트 배수 제한) → Instruction address 는 4비트 단위로 가진다. 즉, LS(맨 왼쪽) 2bits == 00

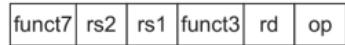
RISC-V Addressing Modes

- Addressing mode: How to locate data needed as operands

1. Immediate addressing



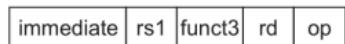
2. Register addressing



Registers

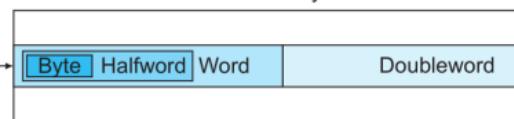
Register

3. Base addressing

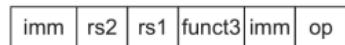


Memory

Register

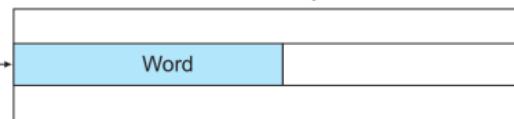


4. PC-relative addressing



Memory

PC



1. Immediate addressing

- a. 상수 사용 경우이다.

- b. 예) addi

2. Register addressing

- a. R-type

3. Base addressing

- a. load/store 와 관련된 명령어

4. PC-relative addressing

- a. branch 명령어

5. Pseudodirect addressing

- a. jump 명령어

Jumping Very Far Away

- RISC-V 는 용량이 매우 높은 점프 거리를 허용해준다.(어느 32비트 주소이든지)
 - lui는 주소의 비트 12~31을 임시 레지스터에 쓴다.
 - jalr은 주소의 하위 12비트를 레지스터에 추가합니다.
그리고 합계로 점프한다.