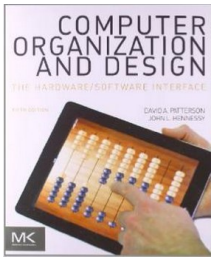
Lecture 3:
Instruction Set Architecture (ISA)

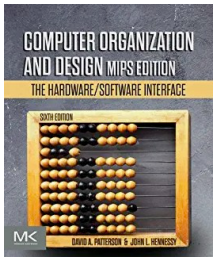
Hunjun Lee
hunjunlee@hanyang.ac.kr

About Textbook

- Textbook



vs.



5th edition (OK)

6th edition (recommended)

We use this ISA version!

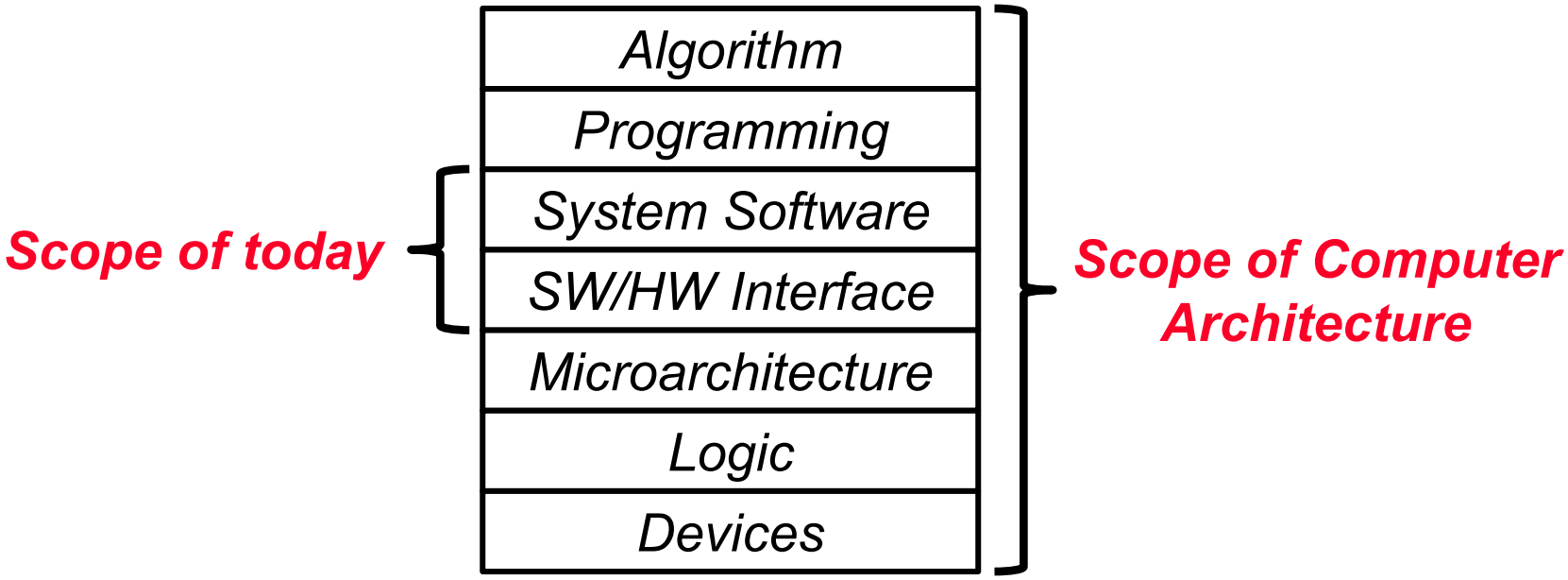
P&H, 6e // three versions: ARMS, RISC-V & **MIPS**

(computer organization & design by Patterson & Hennessy, 6th edition)

More importantly, "Lecture slides" are the most important readings in this class

Computing Problem Hierarchy

- Let's point out how your program run in computer!



OK, very basics first.

“How does your program run on computer?”

Basic components of a computer

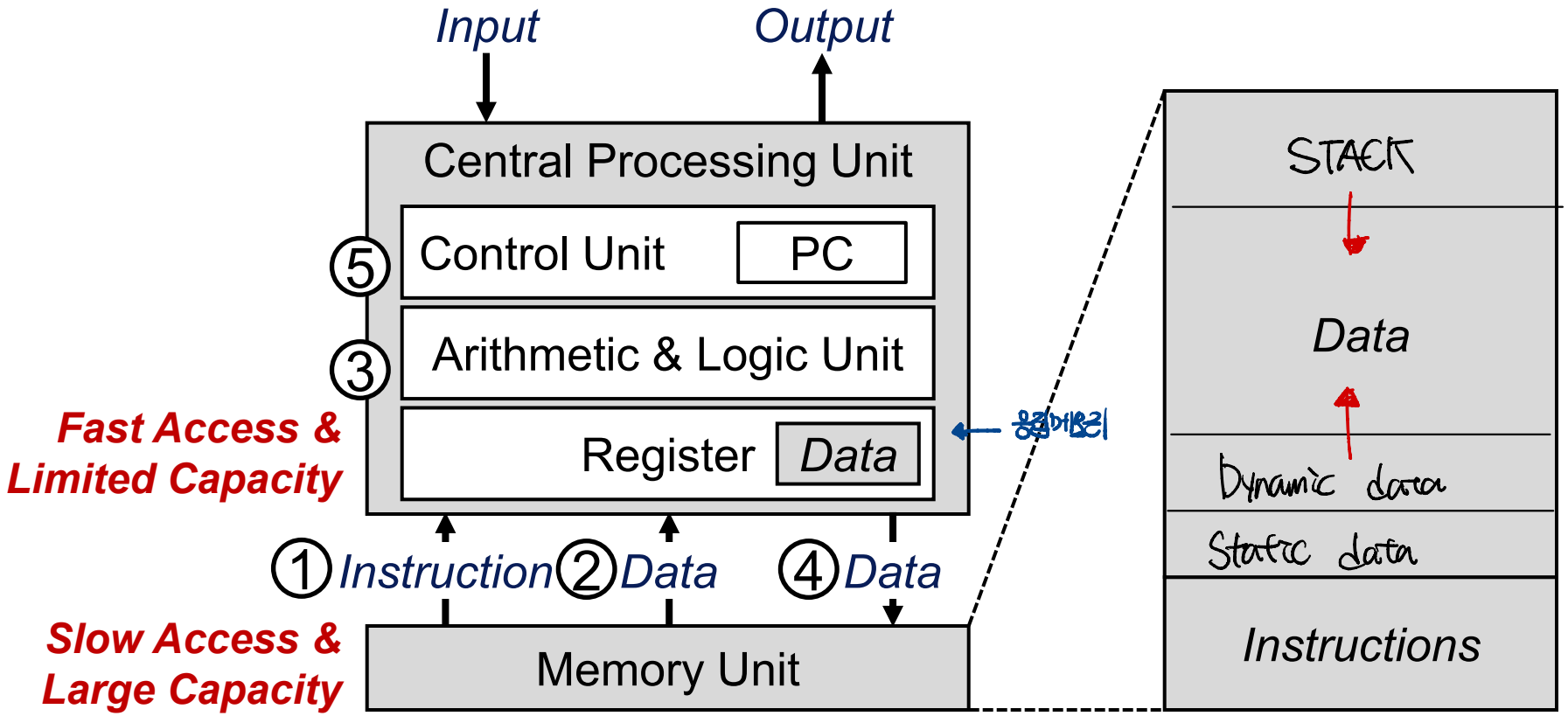
- To get a task done by a general-purpose computer, we need
  - A computer program
    - Specifies what computers must do!
  - The computer itself
    - To carry out the specified task

- Program: A sequence of instructions
  - Instruction: the smallest piece of specified work that the computer can carry out

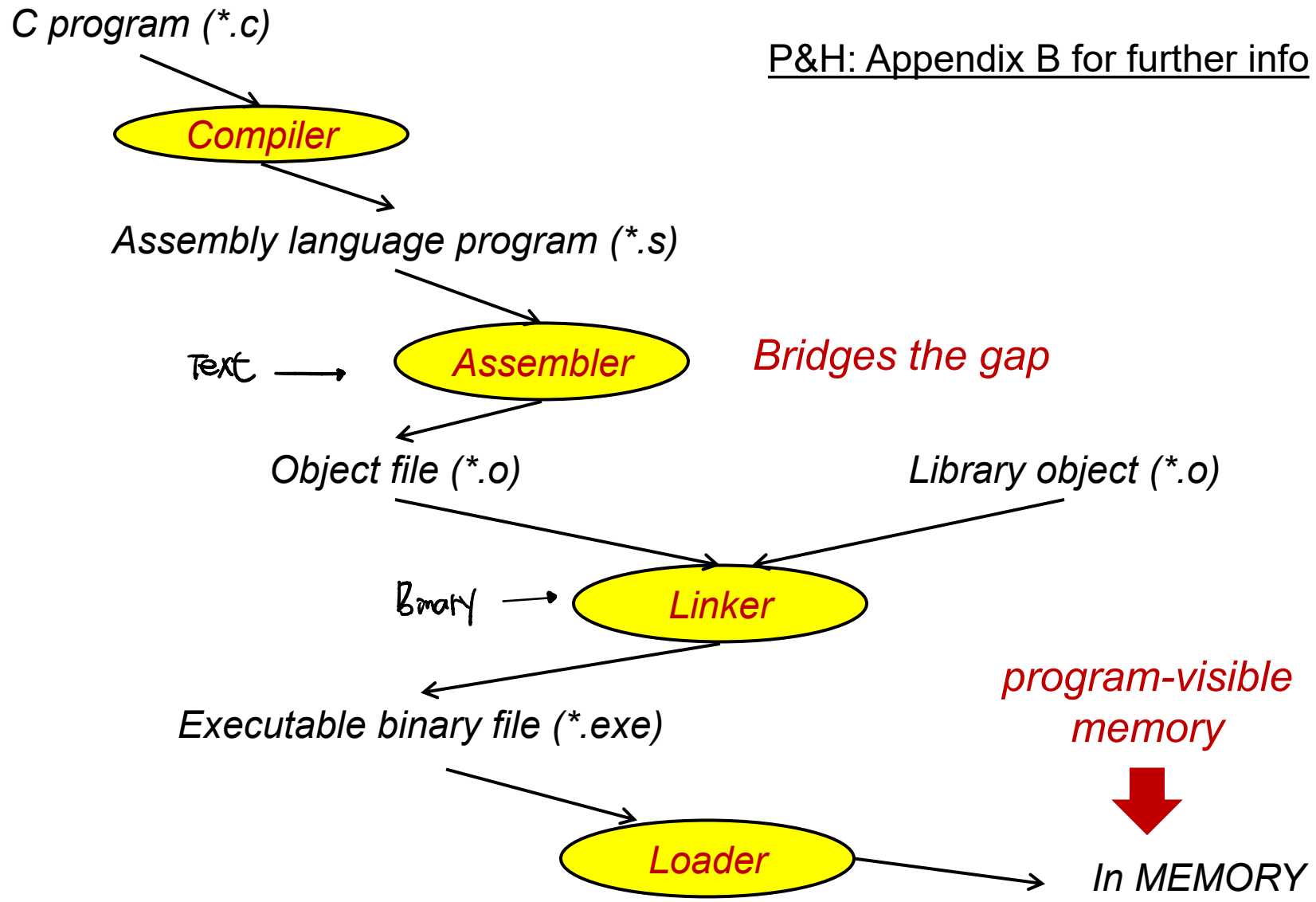
- Instruction set: All possible instructions that a computer is designed to be able to carry out

Von Neumann Architecture

- Both instructions and data are stored in the memory
- Instructions dictate (1) which and how data are manipulated and (2) which instruction should be next



How to load a program to your computer?

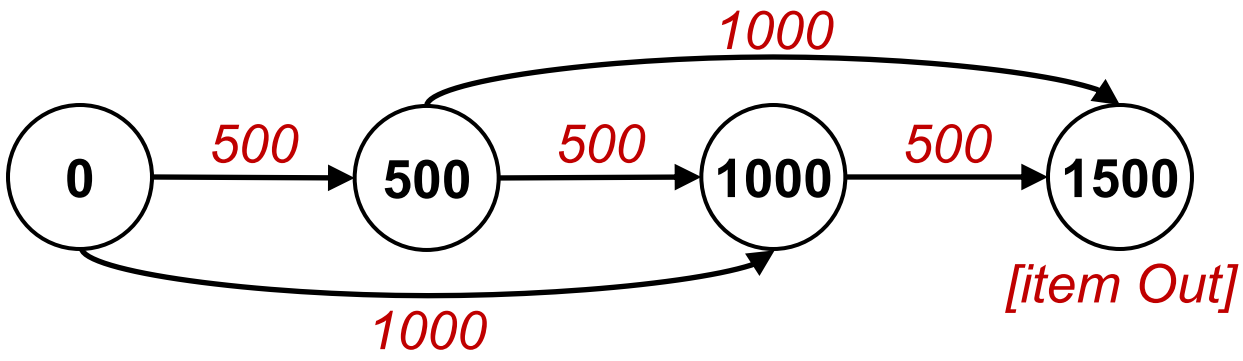


P&H: Appendix B for further info

Recall: State machine

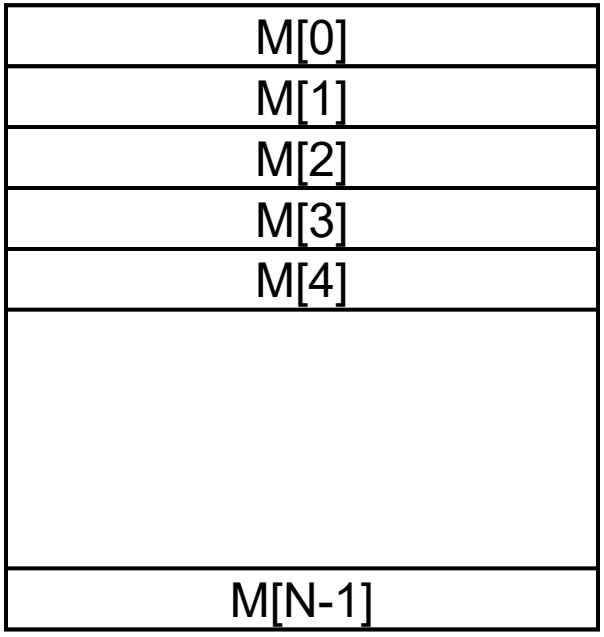
- Computers are essentially a complex state machine

- State machine (Ex. vending machine) 자판기
  - Condition: When a user inputs a total of 1500 ₩, the machine outputs a Coke
  - Input type: a user can input 500 ₩ coin and 1000 ₩ bill)
  - States: an amount of cash a user has inputted
    → 낸 금액



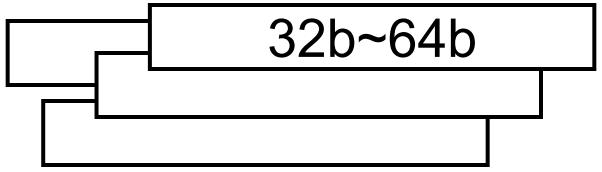
States in Computer
: Memory, Register, PC 가
정의된 값 집

Programmer visible state
(a.k.a. architectural state)



Memory

Array of storage locations indexed by an address



Registers

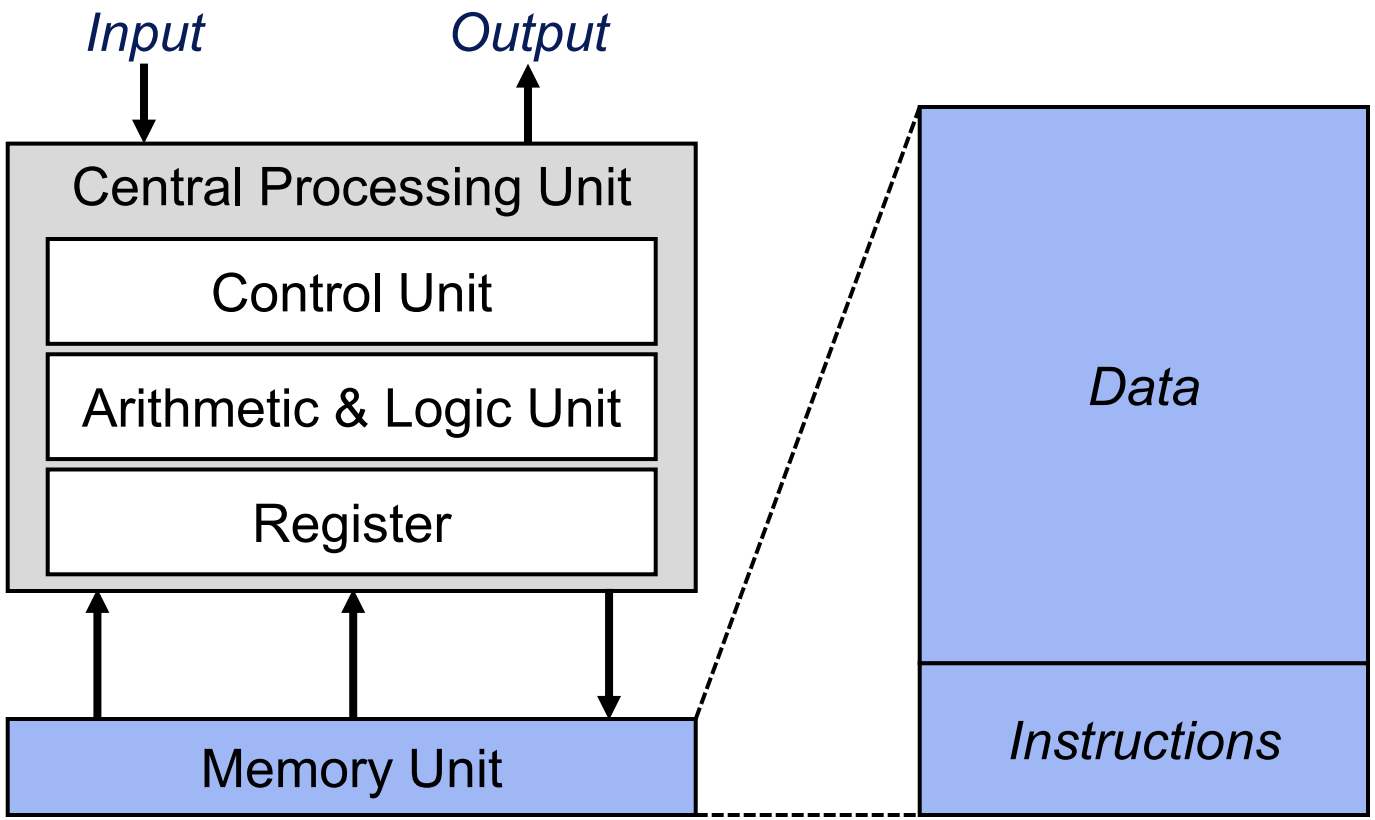
- Given special names in the ISA (as opposed to addresses)
- General vs. special purpose

Program Counter (32b~64b)

Memory address of the current instruction

Instructions (and programs) specify how to transform the values of programmer visible state

Von Neumann Architecture



Runtime storage organization

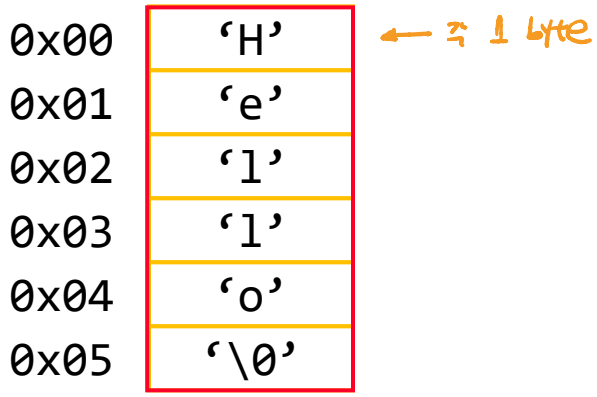
- The memory stores the following:
  - (1) program (instructions) and (2) data required in program execution
- Memory contains bits
  - Logically grouped into bytes (8 bits) and words (e.g., 8, 16, 32 bits)
  - The word size determines the instruction width, register size, …
- Address space: Total number of uniquely identifiable locations in memory (differs depending on the architecture)
  - In LC-3, the address space is 2^{16} (16-bit addresses)
  - In MIPS, the address space is 2^{32} (32-bit addresses)
  - In x86-64, the address space is (up to) 2^{48} (48-bit addresses)

Basics on how the data is stored

Memory addressing

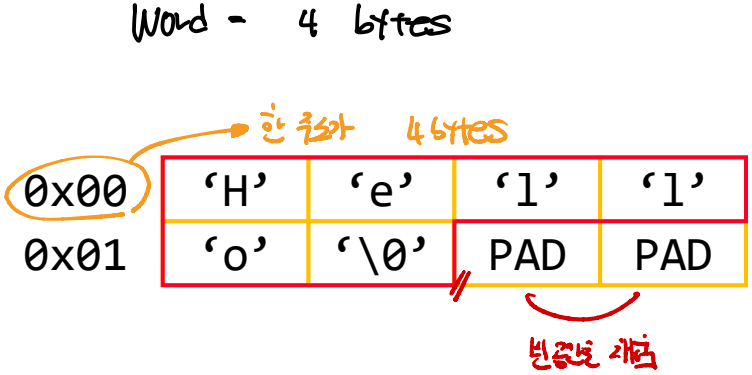
- Machines are either byte or word addressable
  - The addressing format determines how the data is aligned
  - A word addressable memory keeps the data word-aligned

Adopted in memory



Byte Addressing

Adopted in register file

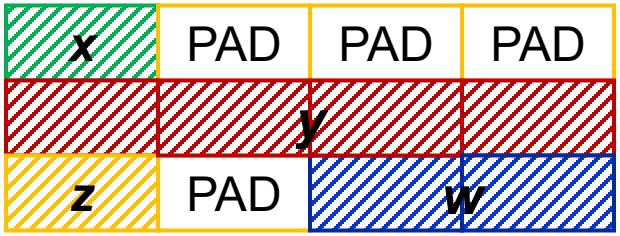


Word Addressing

Memory alignment

- **An address of a variable is aligned based on the size of the variable**
  - Char is byte aligned (any addr is fine)
  - Short is halfword aligned (LSB of byte addr must be 0)
  - Int is word aligned (2 LSBs of byte addr must be 0)

```
char x;          // size 1 byte
int y;           // size 4 byte
char z;          // size 1 byte
short w;         // size 2 byte
```



Memory Alignment 란 무엇인가?

CPU가 기본적으로 4-byte read 하기 때문이다.

4 byte Chunk (word 단위)

- Memory bus 는 4 byte 기준
- Cache Line 도 한번에 여러 byte (32 Byte ~ 64 Byte)를 읽어온다 하지만 내부적으로 4 byte 씩 처리한다.

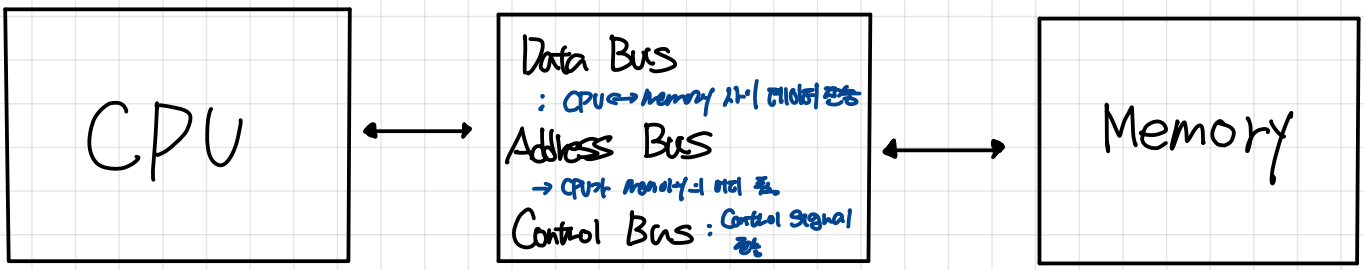
만약 Memory가 Align 되어 있지 않다면?

Bus와 Cache가 하나의 데이터가 여러개의 Bus, Cache line 에 걸치게 될 가능성이 있어 성능이 저하된다.

※ 가장 중요하게, MIPS architecture는 위 같은 상황을 용납하지 않는다.
→ 트라이앵글 발생시킴

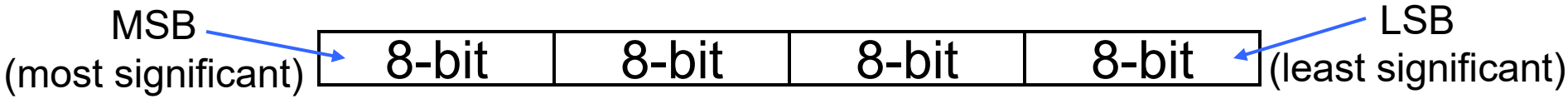
LW (Load) 는 32-bit를 가리키며, Alignment 되어있을 때 성능이 가장 좋다.

추가) MIPS는 LW, Lh, Lb 모두 관해해 성능 및 효율↑
2 byte 1 byte



How are the data stored?
Big Endian vs. Little Endian

- 32-bit signed or unsigned integer comprises 4 bytes



- On a byte-addressable machine . . . . . .
  - When storing 0x12345678

Big Endian

MSB Addr → LSB

12	34	56	78
byte 7	byte 6	byte 5	byte 4
byte 11	byte 10	byte 9	byte 8
byte 15	byte 14	byte 13	byte 12
byte 19	byte 18	byte 17	byte 16

pointer points to the big end

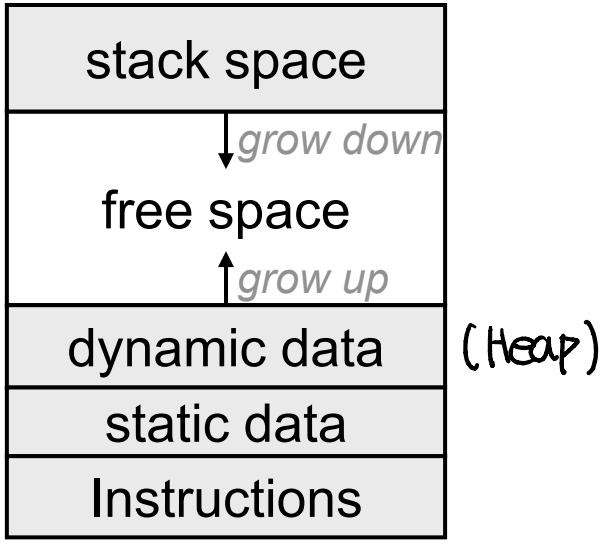
Little Endian

LSB Addr → MSB

78	56	34	12
byte 4	byte 5	byte 6	byte 7
byte 8	byte 9	byte 10	byte 11
byte 12	byte 13	byte 14	byte 15
byte 16	byte 17	byte 18	byte 19

pointer points to the little end

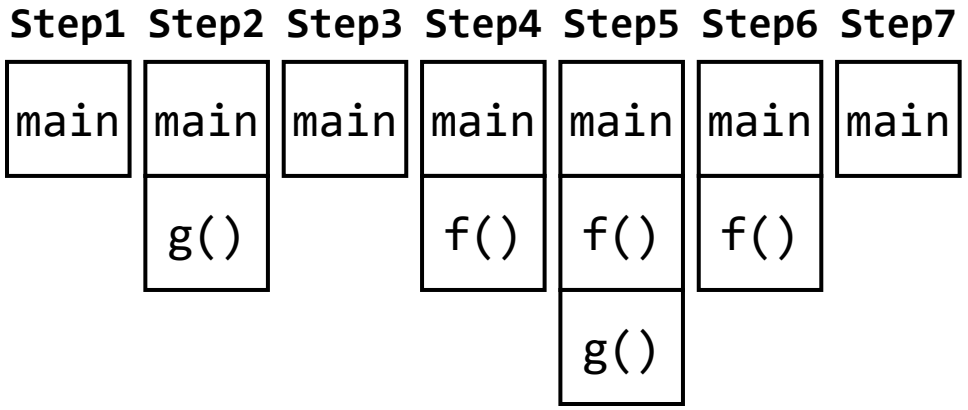
There are four different parts in the storage!



How to maintain data for the program? – using stack

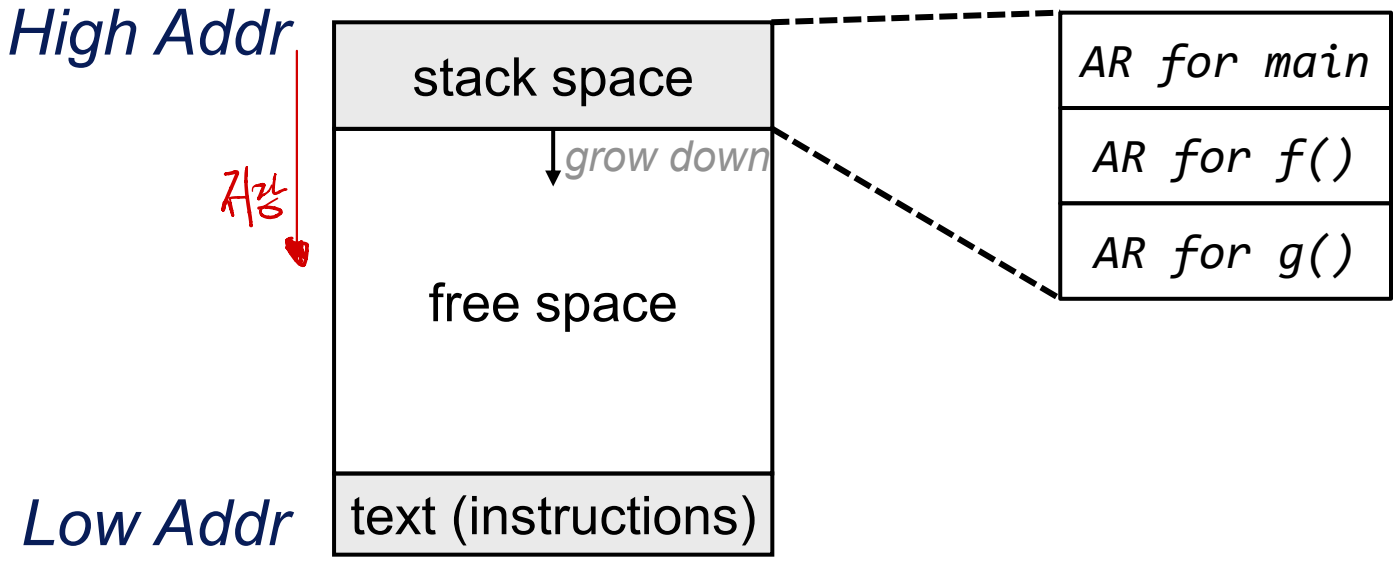
- We can utilize a "stack" to manage data for the active procedures (or functions)
- When P calls Q, then Q returns before P returns
  - Lifetimes of procedure activations (required data for the procedure) are properly nested
  - The activation depends on run-time behavior

```
int g() { return 1; }
int f() { return g(); }
void main() {
        g();
        f();
}
```



Stack management

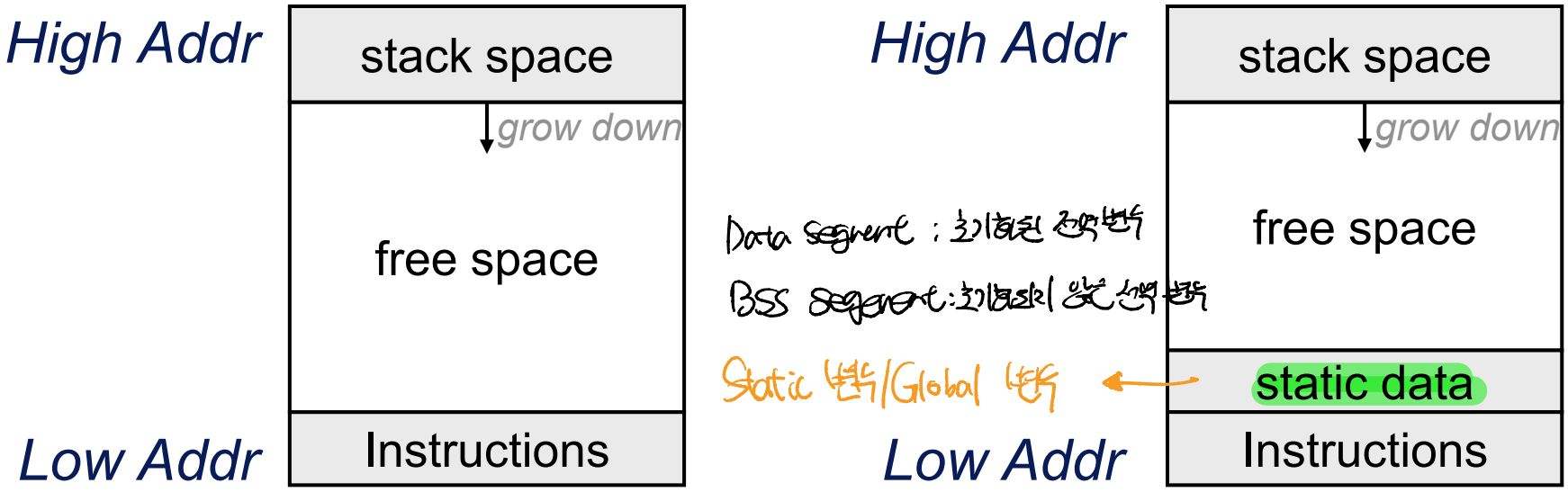
- Stack data is stored starting from the low address, which grows downwards
- The information to manage one function call is called activation record (AR) or frame



Global variables

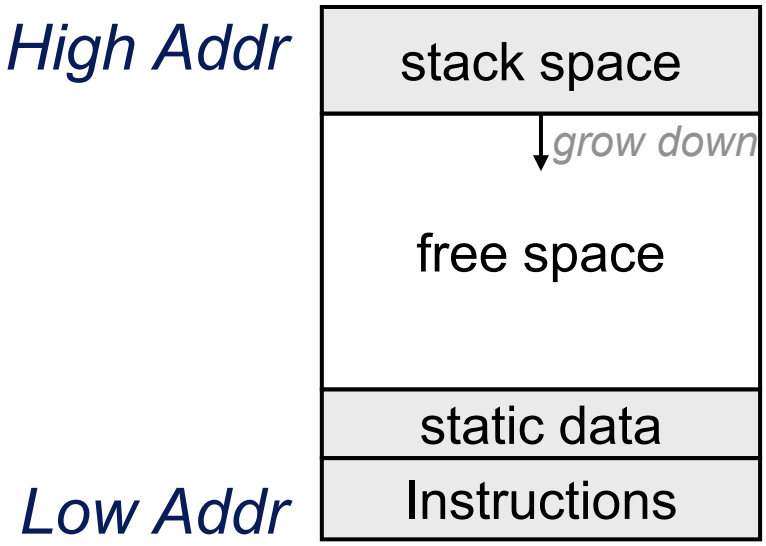
- All references to a global variable point to the same object
  - It would be impossible (or inefficient) to store a global activation in an activation record
- Global variables are assigned a fixed address once (statically allocated)

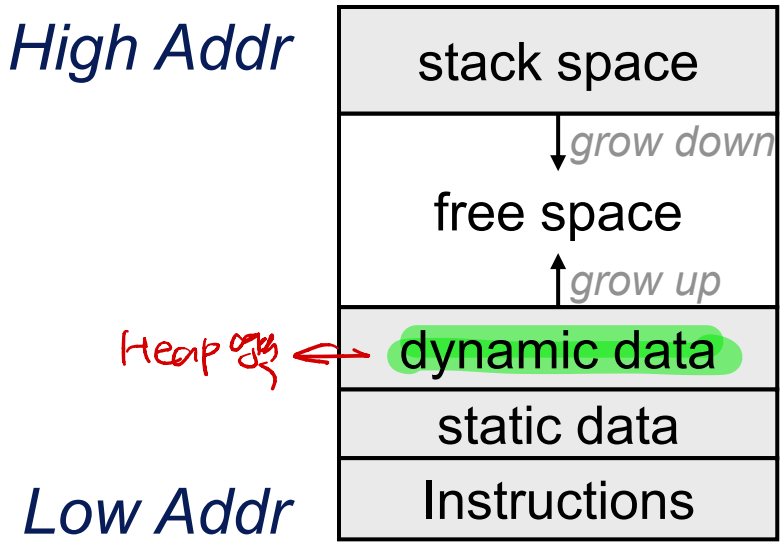
Data 영역(static data)에 Compile-time에 결정



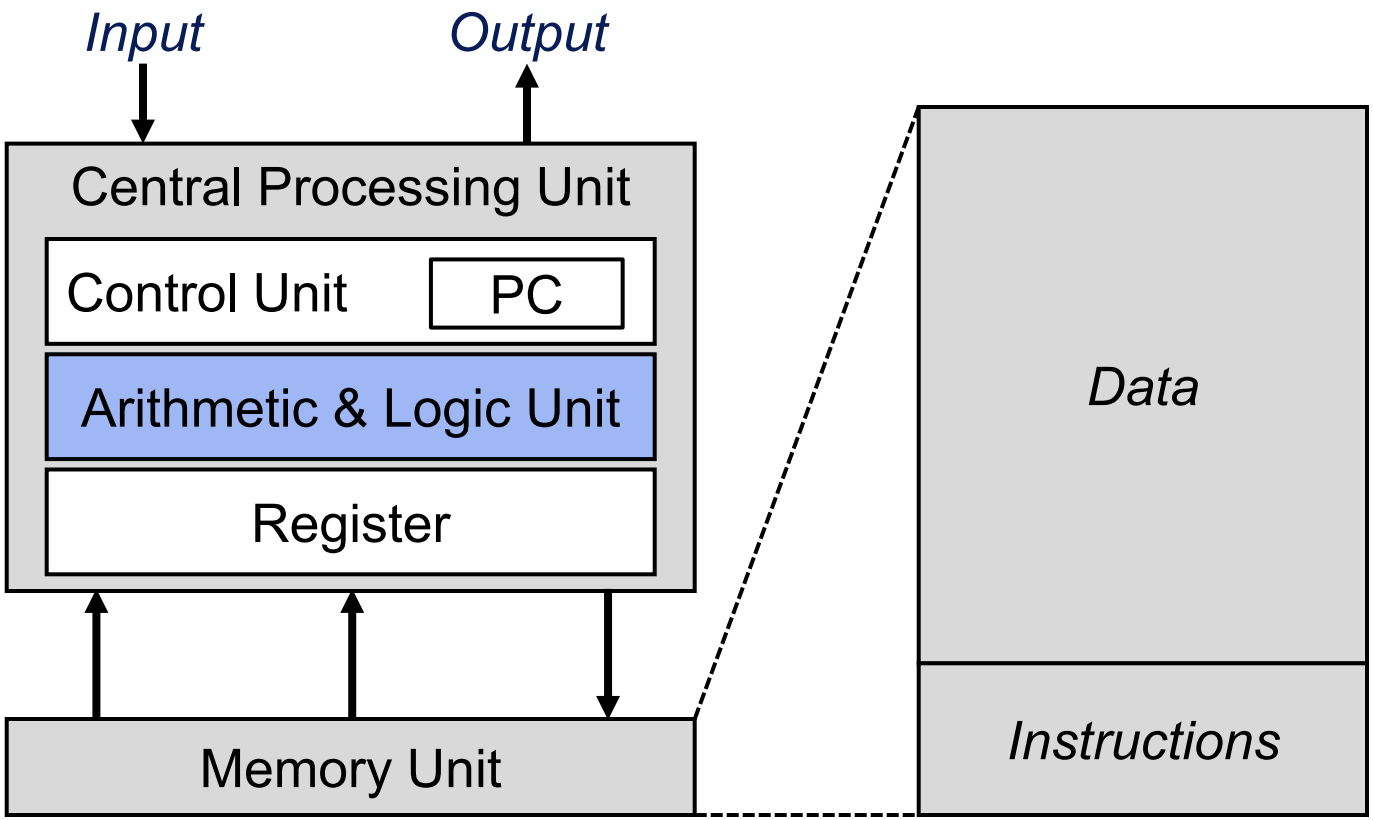
Dynamic allocation

- The dynamically allocated value outlives the procedure that creates it (unless deleted beforehand)
- We rely on heap to store the dynamically allocated data





Von Neumann Architecture



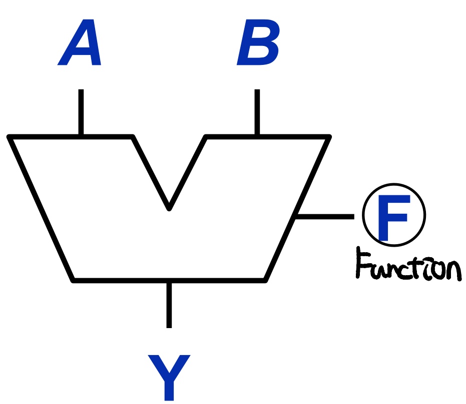
Processing units

- Performs the actual computation(s)
- The processing unit can consist of many functional units
- We start with a simple Arithmetic and Logic Unit (ALU), which executes computation and logic operations
  - MIPS: add, sub, mult, and, nor, …
- The ALU processes quantities that are referred to as words
  - Word length in MIPS is 32 bits (4 bytes)

ALU (Arithmetic logical unit)

- Combines a variety of arithmetic and logical operations into a single unit (that performs only one function at a time)

- Usually denoted with this symbol:



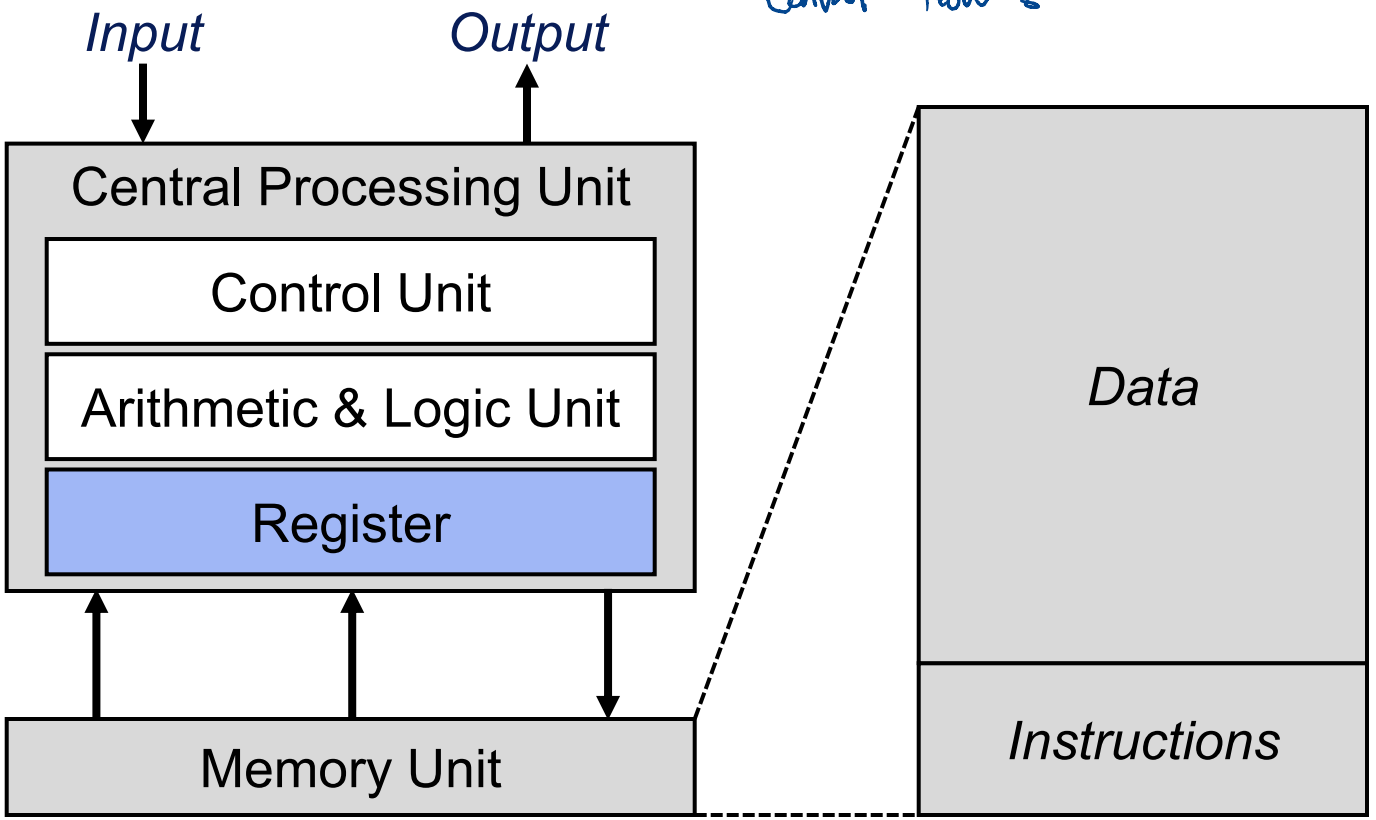
F[2:0]	Function
000	Y = A and B
001	Y = A or B
010	A + B
011	Not used
100	A – B
101	A * B
110	A / B
111	SLT

One of the examples

Von Neumann Architecture

- Both instructions and data are stored in the memory
- Each instruction dictates (1) which and how data are manipulated and (2) which instruction should be next

Control flow



Registers: fast storage

Synchronous write
→ 모든 write가 rising clock 때만 → System 전체를 예측하기 쉬움

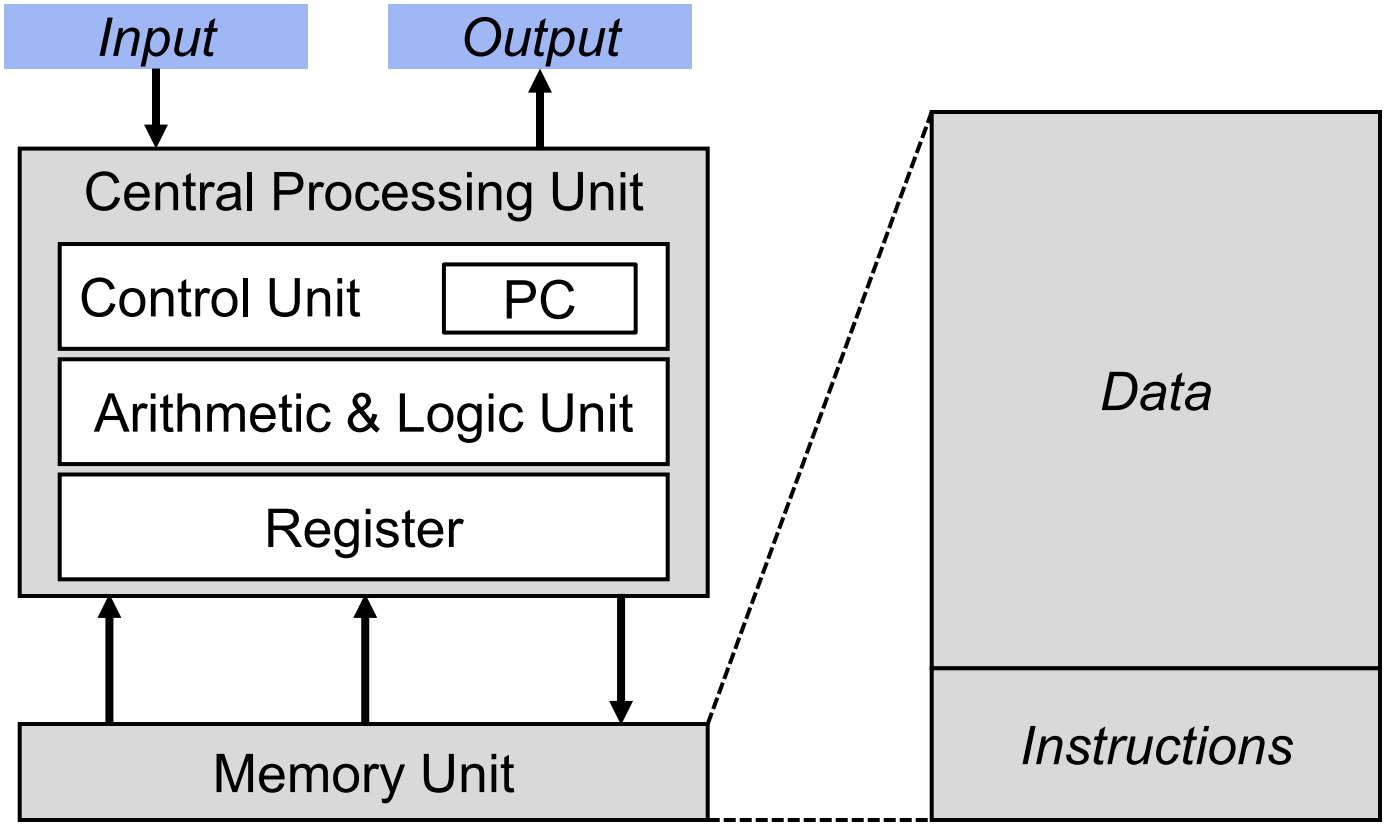
Q) Reader write가 동시에 존재되면??

- **Memory** is large but slow while **registers** are fast and small

- Processing unit utilizes registers during operation
  - Ensure fast access to values to be processed in the ALU (stores temporary values)
    ALU 연산, 데이터 load 시 중간 결과는 일시적으로 register에 저장
  - Combinational read & synchronous write ➔ can execute read + ALU + write in a single cycle
    ALU + register 사용 (시뮬레이터 다르기함)
    Clock Signal 없이 동작
    read+ALU : Clock 없이 가능
    write: Clock 때만
    → 한 cycle 안에 모두 가능
    Clock 상승 Edge 때만 write가능

- Register Set or Register File (Single register 하나에 32개 register)
  - Set of registers that can be manipulated by instructions
  - MIPS has 32 registers
    → 31 = 11111 (5-bit)
    - R0 to R31: 5-bit register number (or Register ID)
    - Register size = Word length = 32 bits
  - There are some special-purpose registers (e.g., `$fp`, `$sp`, …)

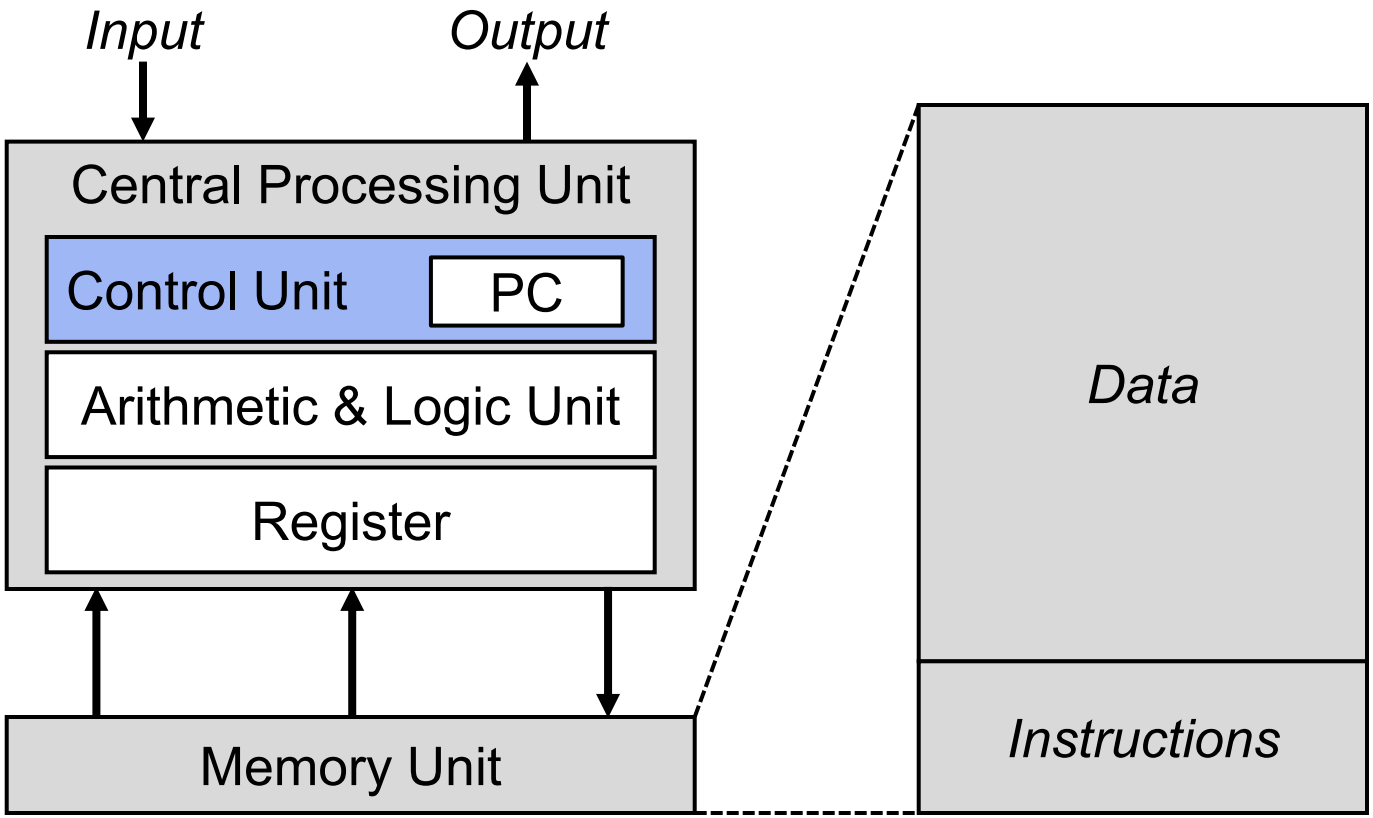
Von Neumann Architecture



Input and output

- Enable information to get into and out of a computer
- There are many input and output devices
- Input examples
  - keyboard, mouse, scanner, disks, …
- Output examples
  - monitor, printer, disks, …

Von Neumann Architecture



Control unit

- Enables a step-by-step execution of a program
  - Proceeds through each instruction in a program in sequence

- Keeps track of which instruction is being processed, via an instruction register

- Keeps track of which instruction to process next, using a program counter (PC) and determines which instruction to process next

Wrap up

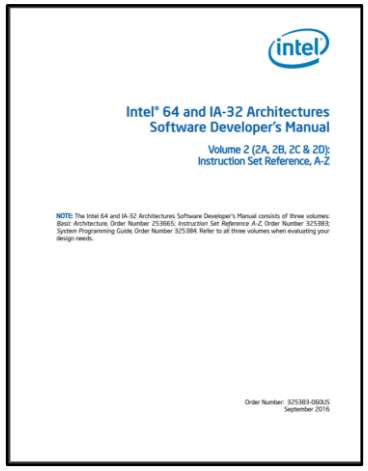
- **Memory Unit:** stores (1) program (instructions) and (2) data required in program execution
- **Arithmetic & Logical Unit:** performs the actual computation(s)
- **Input & Output:** enables information to get in and out of the computer
- **Control Unit:** enables a step-by-step execution of a program

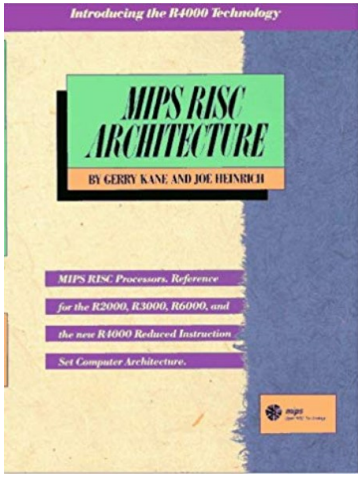
Instruction Set Architecture (ISA)

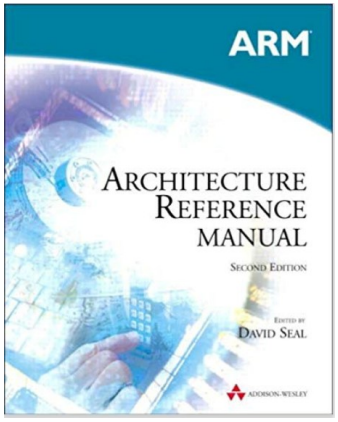
“User’s manual for the computer”

– 사용자의 매뉴얼

Architecture Manuals







Each vendor specifies the instruction sets in a different way

Architecture*

- "The term *architecture* is used here to describe the attributes of a system **as seen by the programmer**, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation."

--- footnote on page 1, Architecture of the IBM System/360,
Amdahl, Blaauw and Brooks, 1964.

How to specify what a computer does?

- **Architecture** Level : User-level에 가까움
  - Car: driving manual & operation manual

    // you don't have to be a car mechanic to drive a car.
  - Computer : program manual

    // you don't have to be a circuit designer to program a computer.
  - 변하면 변하지 않음
- **Microarchitecture** (implementation) Level : Back에서 돌아가는 원리 Architecture의 구현 (설계서)
  - A particular car design has a certain configuration of electrical/mechanical components (e.g., v8 engine vs. v4 engine)
  - A particular computer design has a certain configuration of datapath and control logic units (e.g., adder type, cache, …)
  - 변한다 – 성능 측면에서는 Architecture보다 중요하다.

What are specified/decided in an ISA?

- Data format and size
  - character, binary, decimal, floating point, …
- "Programmer Visible State" **(a.k.a. architectural state)**
  - memory, registers, program counter (PC), etc.
- Instructions: how to **"change"** the programmer visible state?
  - What to perform and what to perform next
  - Where the operands are
- How to interface with the outside world? ISA의 기능적인 측면
- Protection and privileged operations
- Software conventions

Often, you compromise performance for future scalability and compatibility

General Instruction Classes

- **Arithmetic and logical operations** (e.g., add, sub, and, or)
  1) Load operands from specified locations
  2) Compute a result as a function of the operands
  3) Store result to a specified location
  4) Update PC to the next sequential instruction
- **Data movement operations** (e.g., load, store)
  1) Fetch operands from specified locations
  2) Store operand values to specified locations
  3) Update PC to the next sequential instruction
- **Control flow operations** (e.g., branch, jump)
  1) Fetch operands from specified locations
  2) Compute a branch condition and a target address
  3) If "branch condition is true" then PC ← target address
     else PC ← next seq. instruction

Generally defined to be atomic

Instructions for operations

- Different ISAs utilize different instruction for operations
- Number of Operands
  - **Monadic** `OP in2`
  - **Binatic** `OP inout, in2`
    - Save memory (smaller instruction size)
  - **Triadic** `OP out, in1, in2` ← MIPS에서 사용
    - → 보통 ALU operand는 Memory에 있을 필요없이 항상 Register에 저장되있는 값을 이용한다.
- Can ALU operands be in memory? MIPS는 Register file에서 가져온 것이 전제! Memory에서 가져올 수 없다.
  - **No!** `ADD r1 r2 r3 (r1~r3 are registers)`
    - You should load memory to the register before ALU operations
  - **Yes!** `ADD r2, r1, [2000]`
    - → MIPS는 허용 X, 타 ISA에서는 가능할 수도

Different methods depending on the ISA

Instructions for memory addressing

lw $rt, offset($rs) : rt에 [rs+offset] 저장

- Absolute — `lw rt, 10000`
  - `lw`: load word use immediate value as address
- Register Indirect — `lw rt, (r_base)`
  - use Register[r_{base}] as address
- Displaced or based — `lw rt, offset(r_base)`
  - use offset+ Register[r_{base}] as address
- Indexed — `lw rt, (r_base, r_index)`
  - use Register[r_{base}]+ Register[r_{index}] as address
- Memory Indirect — `lw rt ((r_base))`
  - use value at M[Register[r_{base}]] as address

MIPS 지원 X

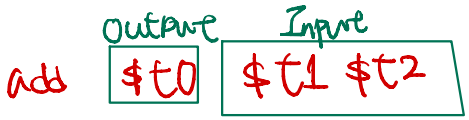
→ Register Indirect 2번 사용과 동일

→ r_{base}에 저장된 값을 읽고 그 값을 주소로 하여 데이터를 가져온다

Complicated memory addressing modes simplify program

MIPS RISC

- Simple operations
  - 2-input, 1-output arithmetic and logical operations
  - Only few alternatives exist to do the same thing



- Simple data movements
  - ALU ops are register-to-register (need a large register file)
  - Memory can be accessed by only load and store instructions
    → **"Load-store architecture"**
- Simple branches
  - Limited varieties of branch conditions and targets
- Simple instruction encoding
  - All instructions encoded **in the same number of bits**
  - Only a few formats

4 byte

Such ISA intended for compiler advances rather than assembly programmers

Evolution of ISA

- Why were the earlier ISAs so simple?
  - Technology limitation
  - Inexperience, lack of precedence
- Why did it get so complicated later?
  - Complex instruction set architecture (CISC)
  - Ease of assembly programming
  - Lack of memory size and performance (in 1970s~80s)
  - Micro-programmed implementation
- Why did it become simple again?
  - Reduced instruction set architecture (RISC)
  - Memory size and speed (cache!)
  - **Compilers**

What about x86 (Intel and AMD)?

RISC vs. CISC

- RISC (Reduced instruction set architecture)
  - The hardware exposes only basic operations as an ISA
  - Simpler instruction (+ simple decoding)
  - Fixed-width instruction (one word)

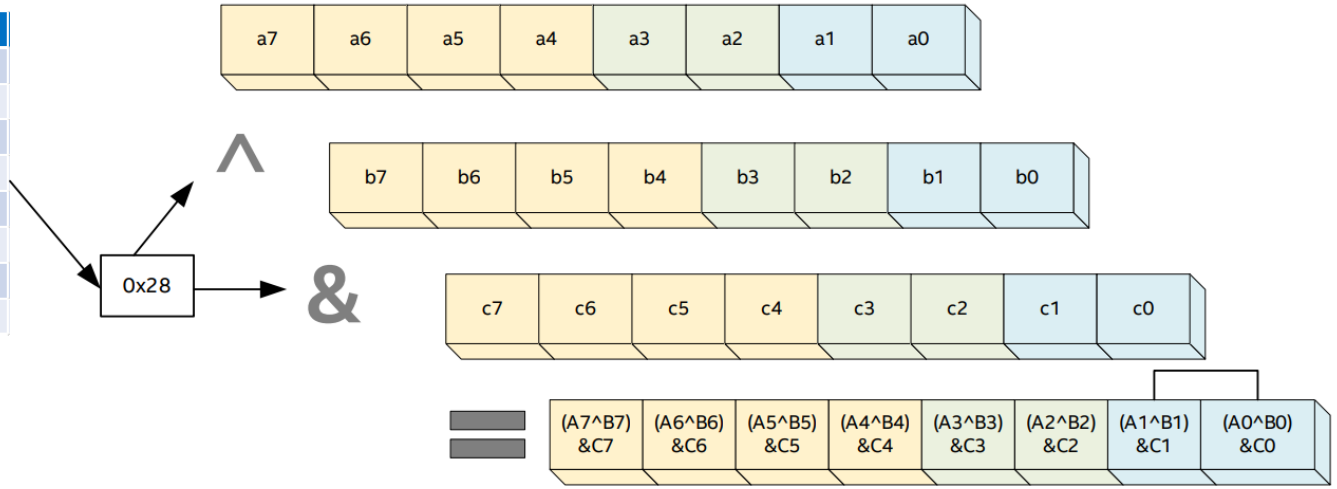
- CISC (Complex instruction set architecture)
  - Exposes more complex operations (combination of multiple RISC-style operations)
  - E.g., A single instruction may … load the data from both memory and register, perform addition, and write back the result
    - Remember `ADD r2, r1, [2000]`

Modern CPUs are RISC, but x86 (Intel + AMD) are CISC
Why??

ISA extension in modern processors - 1

- Intel AVX enables vector operations by defining an Intel AVX-512 ISA

Bit	A	B	C	(A ^ B) & C
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	0

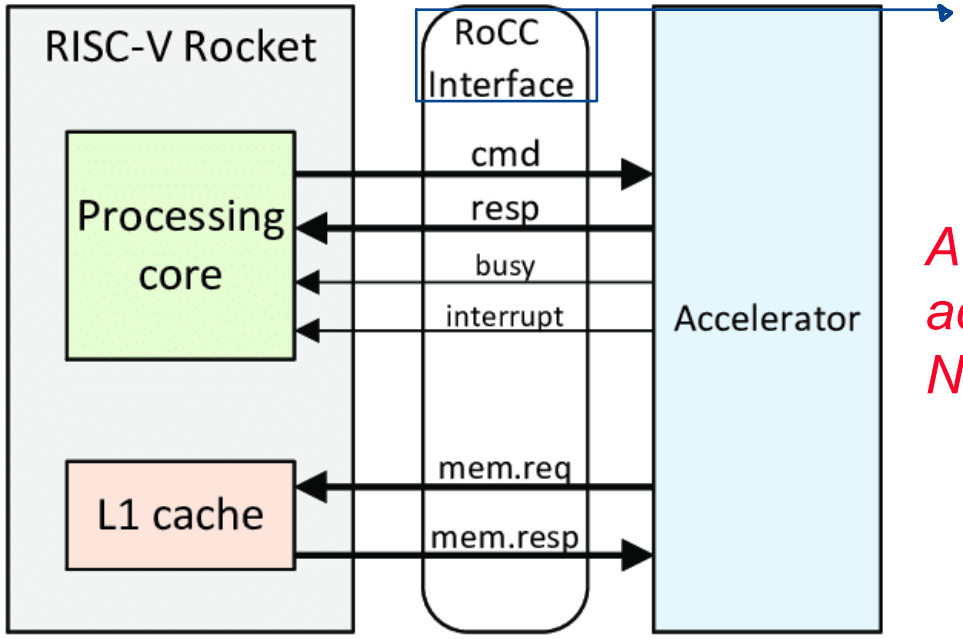


INSTRUCTION SET	C INTRINSIC FORM OF INSTRUCTION
AVX-512	__m512i _mm512_ternarylogic_epi64 (__m512i a, __m512i b, __m512i c, int imm8)
AVX-512	__m512i _mm512_mask_ternarylogic_epi64 (__m512i src, __mmask8 k, __m512i a, __m512i b, int imm8)
AVX-512	__m512i _mm512_maskz_ternarylogic_epi64 (__mmask8 k, __m512i a, __m512i b, __m512i c, int imm8)

ISA extension in modern processors - 2

- Rocket custom coprocessor (RoCC) interface in RISC-V
  - The computer architects can add a custom coprocessor
  - The RoCC interface controls the coprocessor

Controls the accelerator using RoCC



RISC-V to Accelerator : Request
Accelerator to RISC-V : Response

Add a custom accelerator (e.g., NPU)

```
1 #define ROCC_INSTRUCTION_0_R_R(x, rs1, rs2, func7){
2       asm volatile(
3       ".insn r " STR(CAT(CUSTOM_, x)) ", "
4       STR(0x3) ", " STR(func7) ", x0, %0, %1”
5       :
6       : "r"(rs1), "r"(rs2));
7   }
```

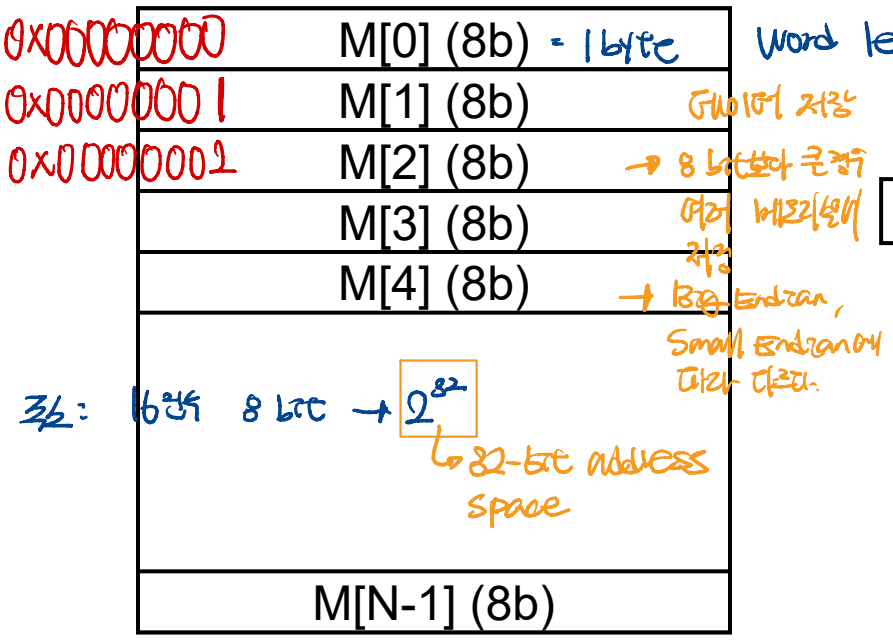
Wrap-up: Terminologies

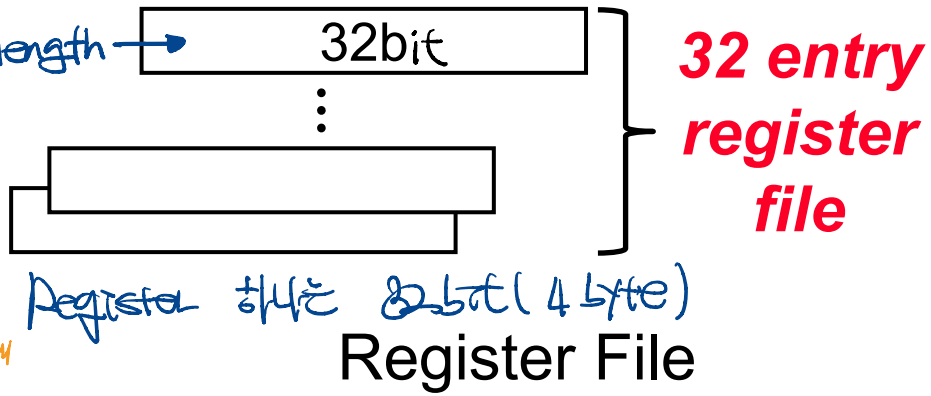
- Instruction Set Architecture (ISA)
  - The machine behavior as observable and controllable by the programmer
- Instruction Set
  - The set of commands understood by the computer
- Assembly Code
  - A collection of instructions expressed in “textual” format
    e.g. Add r1, r2, r3
  - Converted to machine code by an assembler
  - One-to-one correspondence with machine code
- Machine Code
  - A collection of instructions encoded in binary format 0101000…
  - Directly consumable by the hardware

Let’s dive into a concrete example

MIPS ISA (32bit)

MIPS architectural state





Program Counter (32b)

32bit address space

+ Byte addressable

32bit instruction

MIPS architecture has (1) 32-bit word size and (2) a 32-entry register file

MIPS Data format

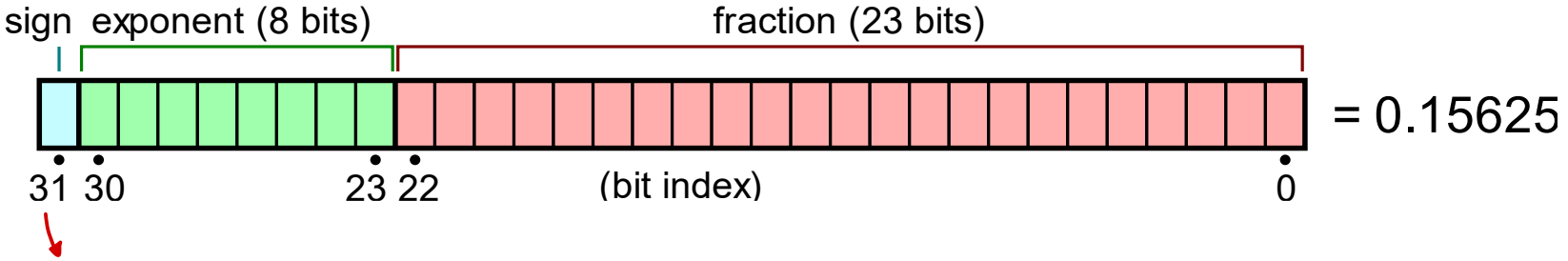
- Most of the data are in 32 bits (= word)
  - Instruction and data addresses
  - Signed and unsigned integers
- Unsigned data format (all values are positive)

$$(b_{31}b_{30}\ldots b_0)_2 = 2^{31}\cdot b_{31} + 2^{30}\cdot b_{30} + \ldots + 2^0\cdot b_0$$

- Signed data format (There exists negative values)
  - 1's complement: change sign by flipping bits 전체 bit를 flip
    - Ex) neg(0111) → 1000, neg(1100) → 0011, …
    - Complicated zero value (i.e., 1111 and 0000 are all 0)
  - 2's complement: change sign by inverting bits and add 1 (Adopted in MIPS)
    - Ex) neg(0111) -> 1000 + 1 → 1001, neg(1100) → 0011 + 1 → 0100
    - Sign extension

Data format - 2

- Floating-point numbers
  - Single precision:
    1-bit sign, 8-bit exponent, 23-bit fraction (Adopted in MIPS)
  - Double precision:
    1-bit sign, 11-bit exponent, 52-bit fraction



$$(-1)^{b_{31}} \times 2^{(b_{30}b_{29}\ldots b_{23})_2 - 127} \times (1.b_{22}b_{21}\ldots b_0)_2$$

지수 필드의 값에서 bias(127)를 뺀다.

There can be precision errors (A floating point numbers may not represent exact value!)

Data format (Sidenote)

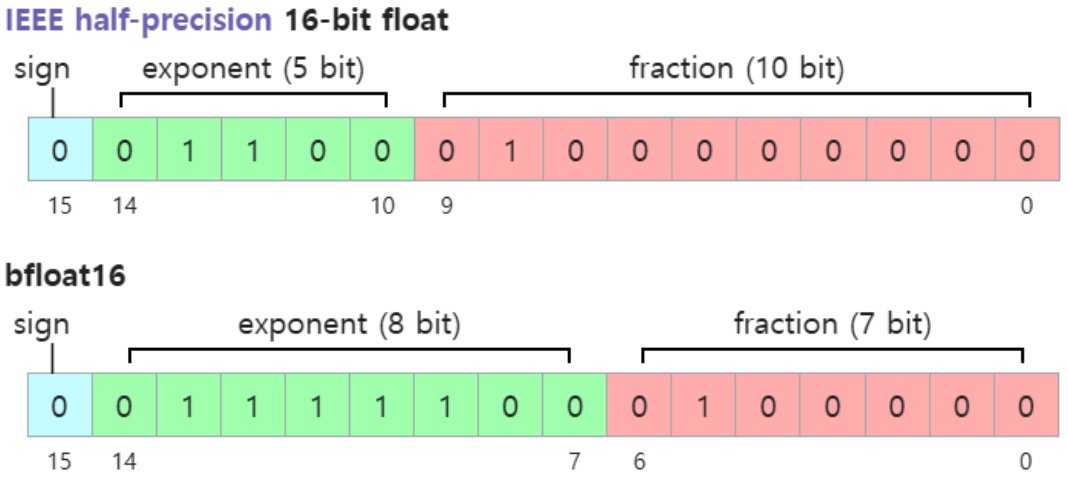
- We can further reduce the precision in some fault tolerant applications (e.g., neural networks)
  - IEEE 754 half-precision

    1-bit sign, 5-bit exponent, 10-bit fraction

    ➔ Suffers from high accuracy drop due to small range
  - bfloat (brain floating point)

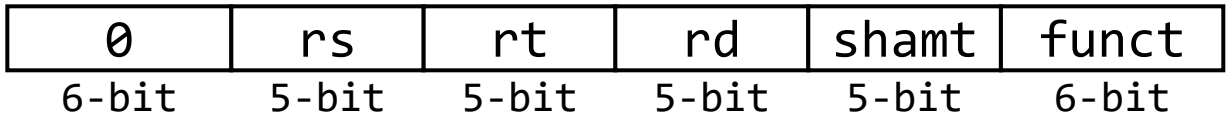
    1-bit sign, 8-bit exponent, 7-bit fraction

    ➔ **Improves accuracy** by maintaining the range



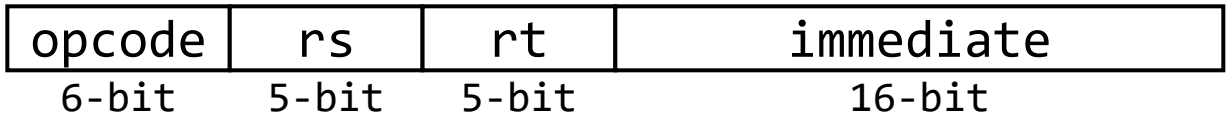
MIPS instruction formats

- Three simple formats
  - R-type, 3 register operands



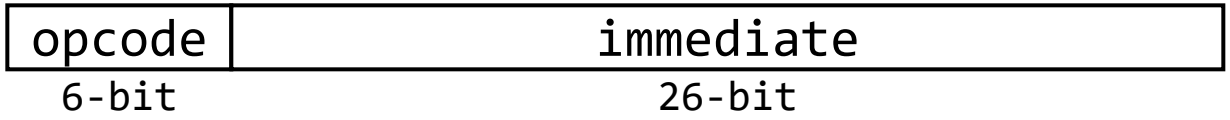
R-type

  - I-type, 2 register operands and 16-bit immediate operand



I-type

  - J-type, 26-bit immediate operand

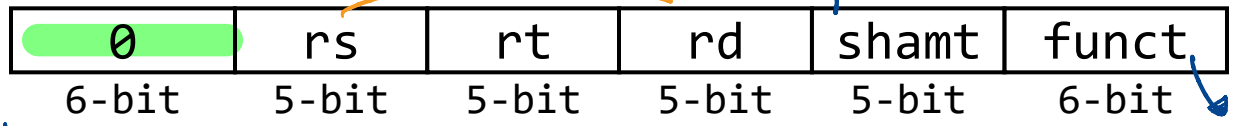


J-type

- Simple Decoding
  - 4 bytes per instruction, regardless of format (fixed size)
  - Must be 4-byte aligned (2 LSB of PC must be 2b'00)
  - Format and fields readily extractable

→ 고정된 형식때에 쉽게 추출이 가능하다.

R-type Instructions



R-type

- `funct`: type of ALU operation
  - Arithmetic: {signed, unsigned} x {ADD, SUB, MULT, DIV, …}
  - Logical: {AND, OR, XOR, NOR, …}
  - Shift: {Left, Right-Logical, Right-Arithmetic}
- `shamt`: shift amount (only used for shift operation)

- Assembly and semantics
  - Corresponding assembly: **`opcode rd rs rt`**
    - `GPR[rd] = GPR[rs] op GPR[rt]`
    - `PC = PC + 4` (use the next instruction)

I-type Instructions – ALU ver.

opcode	rs	rt	immediate
6-bit	5-bit	5-bit	16-bit

I-type

- opcode: there are immediate ALU instructions
  - addi, addiu, andi, ori, xori, slti, sltiu, lui, …
- Assembly and semantics
  - Corresponding assembly: **opcode rt rs immediate**
    - GPR[rt] = GPR[rs] op sign-extend(immediate)
      - sign-extend(1000…000) = concat(11…1,1000…000)
    - PC = PC + 4 (use the next instruction)
- What if you need to perform 32-bit immediate?
  - lui at 0xABCD // store the upper 16 bits
  - ori at 0x1234 // store the lower 16 bits

Assembly Programming - ALU

- Break down high-level program constructs into a sequence of elemental operations

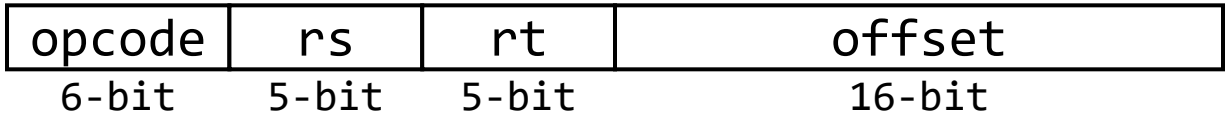
- E.g. High-level Code

```
f = ( g + h ) - ( i + 100 )
```

- Assembly Code
  - suppose f, g, h, i are in r_f, r_g, r_h, r_i
  - suppose r_{temp} is a free (available) register

```
add r_temp r_g r_h      # temp = g+h
addi r_f r_i 100        # f = i+100
sub r_f r_temp r_f      # f = temp-f
```

I-type Instructions – Memory ver.



I-type

- opcode: there are immediate memory instructions
  - lw, lh, lhu, lb, lbu, sw, sh, sb, …
  - // lw indicates load, sw indicates store

- Assembly and semantics
  - Corresponding assembler: **load/store** **rt offset(rs)**
    - GPR[rt] = MEM[GPR[rs] + sign-extend(offset)] // load
    - MEM[GPR[rs] + sign-extend(offset)]= GPR[rt] // store
    - PC = PC + 4 (use the next instruction)

Assembly Programming - Memory

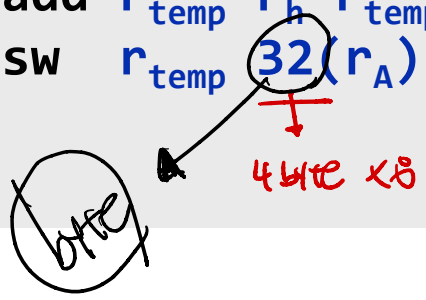
- E.g. High-level Code

```
A[ 8 ] = h + A[ 0 ]
```

where **A** is an array of integers (4–byte each)

- Assembly Code
  - suppose &A, h are in r_A, r_h
  - suppose r_{temp} is a free (available) register

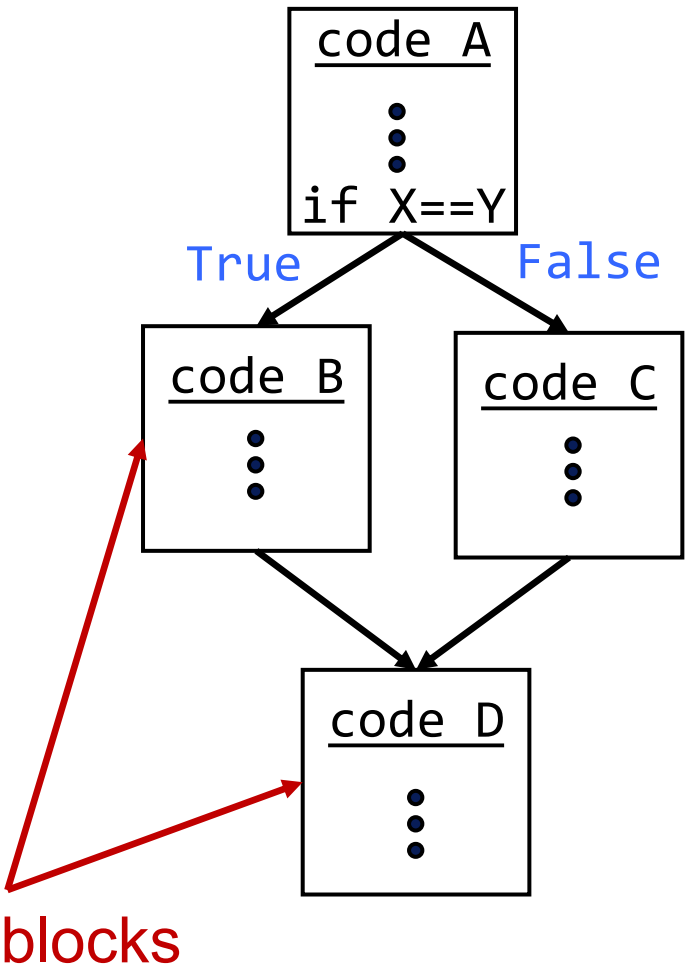
```
lw  r_temp 0(r_A)      # temp = A[0]
add r_temp r_h r_temp  # temp = h + A[0]
sw  r_temp 32(r_A)     # A[8] = temp
                       # note A[8] is 32 bytes
                       #      from A[0]
```

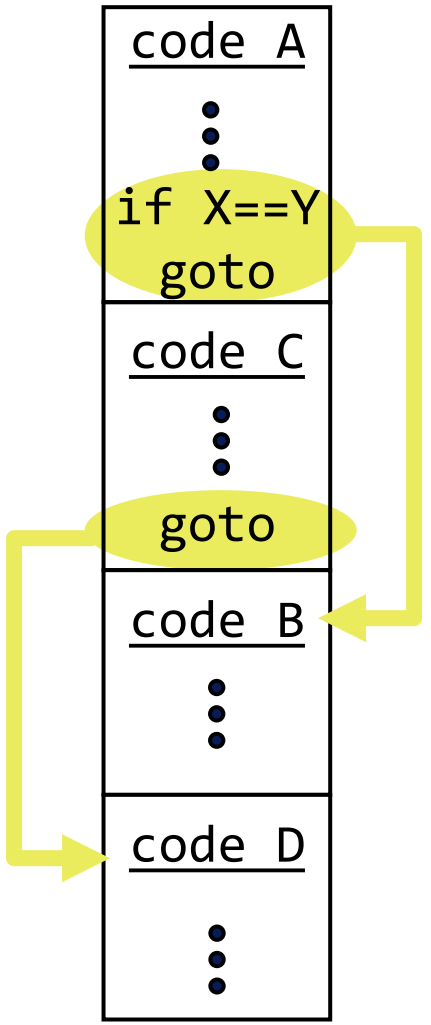


Control Flow Instructions

Example code:

```
code A
if (X == Y){
        code B
}
else {
        code C
}
code D
```





I-type은 현재 PC에서 상대적인 이동 → 두 피연산자 외에 Sign-Extend

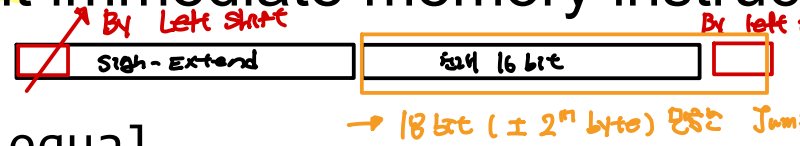
I-type Instructions – Control ver.

opcode	rs	rt	label
6-bit	5-bit	5-bit	16-bit

I-type

→ Jump 할 Instruction의 개수 → 나중에 4를 곱해 byte 단위로 환산

- opcode: there are eight immediate memory instructions
  - bne, beq, …
  - branch equal or not equal …
    - ↳ rs와 rt값 비교



→ 18 bit (± 2^{17} byte) 만큼 Jump 가능

- Assembler and semantics
  - Corresponding assembly: **beq rs rt label**
    - target = (PC + 4) + sign-extend(label) x 4 // word-aligned
    - if (GPR[rs] == GPR[rt]) PC = target else PC = PC + 4
    - ↳ beq
  - If you want to jump more than 18 bits (i.e., 16 bit + 2 bit) ➔ Utilize J-type (in the following slides)

byte → MIPS 명령어 공통 4-byte씩을 4byte로 구분해준다. Left Shift

rs == rt일 때

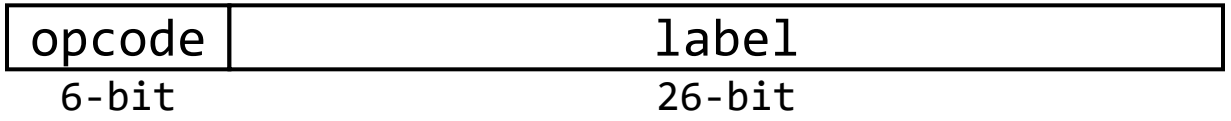
Instruction은 word-aligned → 4의 배수 맞추기

offset, label이 현재 명령어의 다음 명령어 기준으로 몇 개의 명령어가 떨어져 있는지

16-bit → Sign-extend → 32-bit → Shift → 18-bit 환산

Left Shift (2-bit)

J-type Instructions

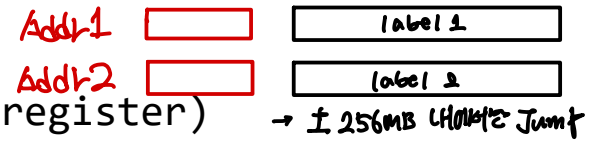


- opcode: there are eight immediate memory instructions
  - j, jal

MIPS는 J-type으로 32-bit 전체 주소 표현하지 못한다.
→ (PC+4)의 상위 4bit를 이용한다.
현재 PC위치로 2^{28} byte = 256MB 범위 내에서만 Jump 가능

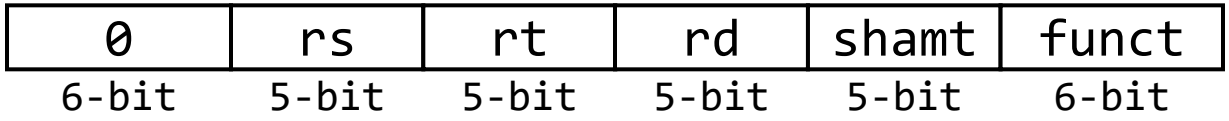
- Assembler and semantics
  - target = (PC+4)[31:28]x2^{28} |$_{\text{bitwise-or}}$ zero-extend(label) x 4
  - // use the first four bits of PC and append label x 4
  - Corresponding assembly: **j label**
    - PC = target
  - Corresponding assembly: **jal label**
    - GPR[ra] = PC + 4
      - save the next PC to ra (a dedicated register)
    - PC = target

현재 PC에서 상위 4bit 고정
→ 현재 PC위치 256MB 범위에서 Jump



ZERO extend를 사용하는 이유
→ 절대 주소 점프, 즉 음수일 리가 없다.

R-type Instructions – Control ver.



R-type

- There exists an R-type control instruction
  - `jr`

- Generally used along with `jal label` (upon a function call)
- Assembler and semantics
  - `Corresponding assembly: jr rs`
    - `PC = GPR[rs]`

Assembly Programming - Control

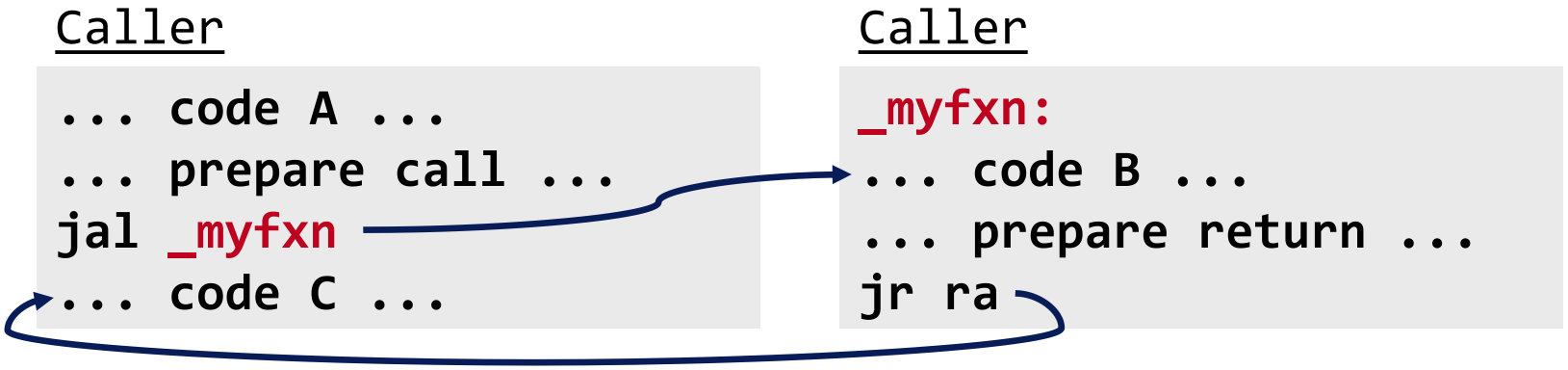
- E.g. High-level Code

```
if (i == j) then
        e = g;
else
        e = h;
f = e;
```

- Assembly Code
  - suppose `e, f, g, h, i, j` are in `re, rf, rg, rh, ri, rj`

```
        bne ri rj L1          # L1 and L2 are addr labels
                              # assembler computes offset
        add re rg r0 # e = g
        j L2
L1:     add re rh r0 # e = h
L2:     add rf re r0 # f = e
```

Assembly Programming – Function Call



- sequence: **A** → call **B** → return **C** …
- There is a prologue and epilogue before and after a function call
  - How to pass an argument to the caller?
  - If **A** set r10 to 1, what is the value of r10 when **B** returns to **C**?
  - What registers can **B** use?
  - What happens to ra if **B** calls another function

Let's dive into the register file usages in MIPS

R2000 Register Usage Convention

Register	Functionality	Alias
r0	Always 0	$zero
r1	Reserved for the assembler	$at
r2, r3	Function return values	$v0, $v1
r4 ~ r7	Function call arguments	$a0 ~ $a3
r8 ~ r15	“Caller-saved” temporaries	$t0 ~ $t7
r16 ~ r23	“Callee-saved” temporaries	$s0 ~ $s7
r24 ~ r25	“Caller-saved” temporaries	$t8 ~ $t9
r26, r27	Reserved for the OS (e.g., interrupts)	$k0 ~ $k1
r28	Global pointer	$gp
r29	Stack pointer	$sp
r30	Frame pointer	$fp
r31	Return address	$ra

R2000 Register Usage Convention

Register	Functionality	Alias
r0	Always 0	$zero
r1	Reserved for the assembler	$at
r2, r3		
r4 ~ r7	Function call arguments	$a0 ~ $a3
r8 ~ r15	“Caller-saved” temporaries	$t0 ~ $t7
r16 ~ r23	“Callee-saved” temporaries	$s0 ~ $s7
r24 ~ r25	“Caller-saved” temporaries	$t8 ~ $t9
r26, r27	Reserved for the OS (e.g., interrupts)	$k0 ~ $k1
r28	Global pointer	$gp
r29	Stack pointer	$sp
r30	Frame pointer	$fp
r31	Return address	$ra

Stores 32-bit zero value

R2000 Register Usage Convention

Register	Functionality	Alias
r0	Always 0	$zero
r1	Reserved for the assembler	$at
r2, r3	Function return values	$v0, $v1
r4 ~ r7		
r8 ~ r15		
r16 ~ r23	“Callee-saved” temporaries	$s0 ~ $s7
r24 ~ r25	“Caller-saved” temporaries	$t8 ~ $t9
r26, r27	Reserved for the OS (e.g., interrupts)	$k0 ~ $k1
r28	Global pointer	$gp
r29	Stack pointer	$sp
r30	Frame pointer	$fp
r31	Return address	$ra

Temporary register to enable 32-bit immediate operation

R2000 Register Usage Convention

Register	Functionality	Alias
r0	Always 0	$zero
r1	Reserved for the assembler	$at
r2, r3	Function return values	$v0, $v1
r4 ~ r7	Function call arguments	$a0 ~ $a3
r8 ~ r15	"Caller saved" temporaries	$t0 ~ $t7
r16 ~ r23		
r24 ~ r25		
r26, r27		
r28		
r29	Stack pointer	$sp
r30	Frame pointer	$fp
r31	Return address	$ra

Used to pass values between function calls!

Q1. We need to pass more than 4 values

Q2. The function returns non-primitive data (e.g., struct)

R2000 Register Usage Convention

Register	Functionality	Alias
r0	Always 0	$zero
r1	Reserved for the assembler	$at
r2, r3	Function return values	$v0, $v1
r4 ~ r7	Function call arguments	$a0 ~ $a3
r8 ~ r15	“Caller-saved” temporaries	$t0 ~ $t7
r16 ~ r23	“Callee-saved” temporaries	$s0 ~ $s7
r24 ~ r25	“Caller-saved” temporaries	$t8 ~ $t9
r26, r27	Reserved for the OS (e.g. interrupts)	$k0 ~ $k1
r28		
r29		
r30		
r31		

Temporary values are stored in the register file, instead of using memory

↳ Register 대신 메모리에 저장하게 됨

Q. The # of temporaries exceeds 18

Q. What are caller-saved and callee-saved?

R2000 Register Usage Convention

Register	Functionality	Alias
r0	Always 0	$zero
r1	Reserved for the assembler	$at
r2, r3	Function return values	$v0, $v1
r4 ~ r7	Function call arguments	$a0 ~ $a3
r8 ~ r15	"Caller-saved" temporaries	$t0 ~ $t7
r16 ~ r23	"Callee-saved" temporaries	$s0 ~ $s7
r24 ~ r25	"Caller-saved" temporaries	$t8 ~ $t9
r26, r27	Reserved for the OS (e.g., interrupts)	$k0 ~ $k1
r28	Global pointer	$gp
r29	Stack pointer	$sp
r30	Frame pointer	$fp
r31	Return address	$ra

Determine which functions to apply on syscall
e.g., print, put … (which to apply)?

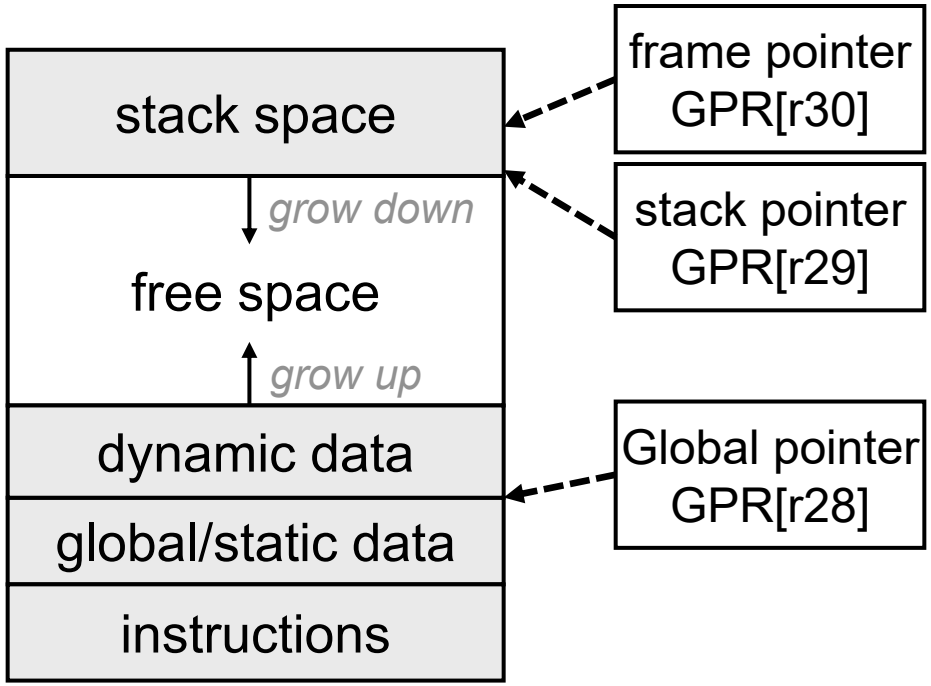
↳ 여러 syscall을 한 경우

→ $k0~$k1에 저장된 값을 확인

R2000 Register

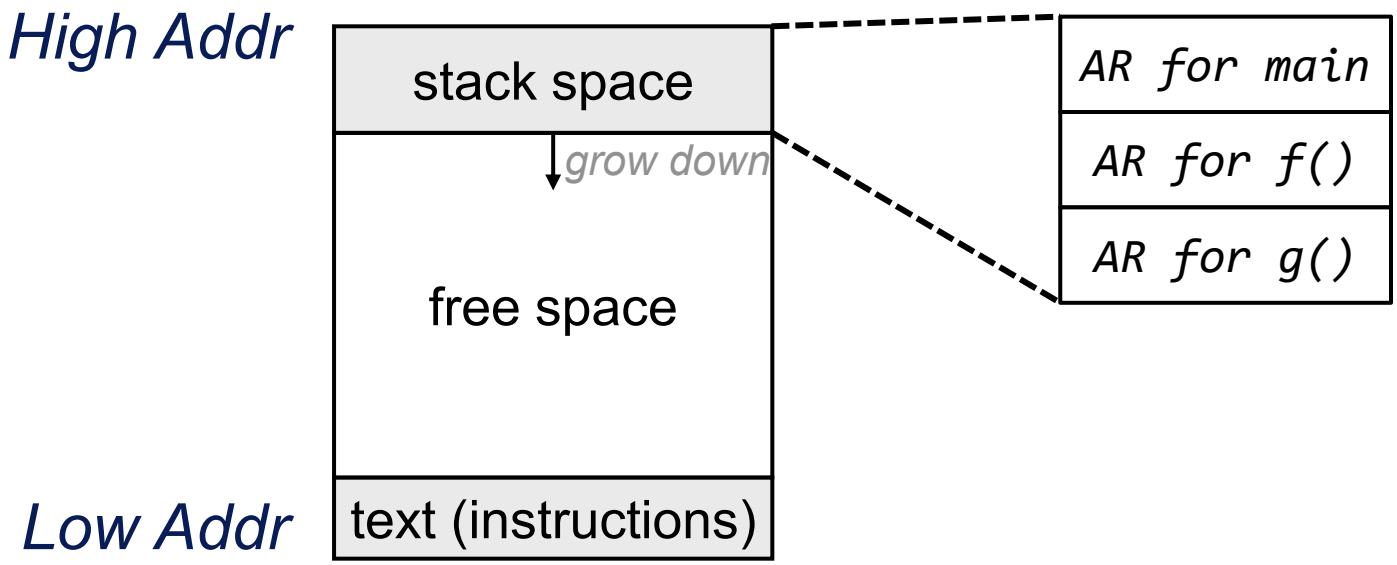
Register	F	
r0		
r1	Reserve	
r2, r3	Funct	
r4 ~ r7	Functi	
r8 ~ r15	“Caller-	
r16 ~ r23	“Callee-	
r24 ~ r25	“Caller-	
r26, r27	Reserved for the OS (e.g., interrupts)	$k0 ~ $k1
r28	Global pointer	$gp
r29	Stack pointer → Stack	$sp
r30	Frame pointer	$fp
r31	Return address	$ra

Pointers to address main memory



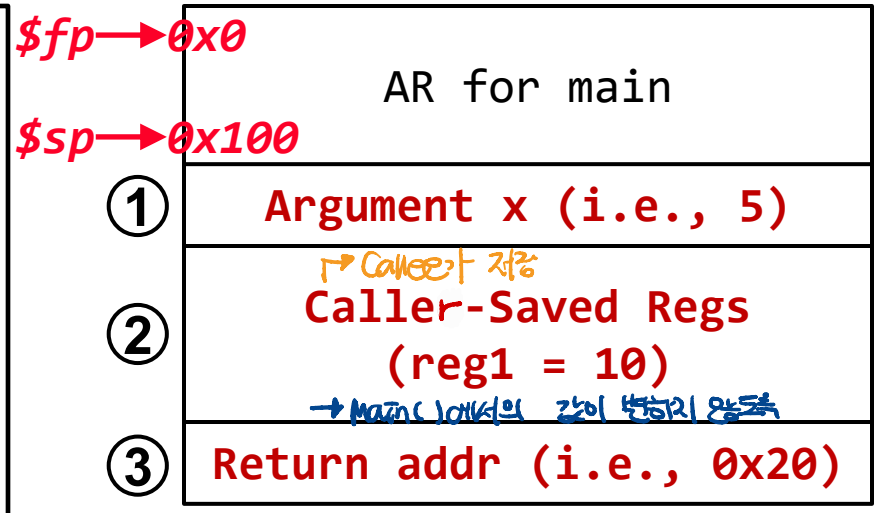
Recall: Stack management

- Stack data is stored starting from the low address, which grows downwards
- The information to manage one function call is called activation record (AR) or frame

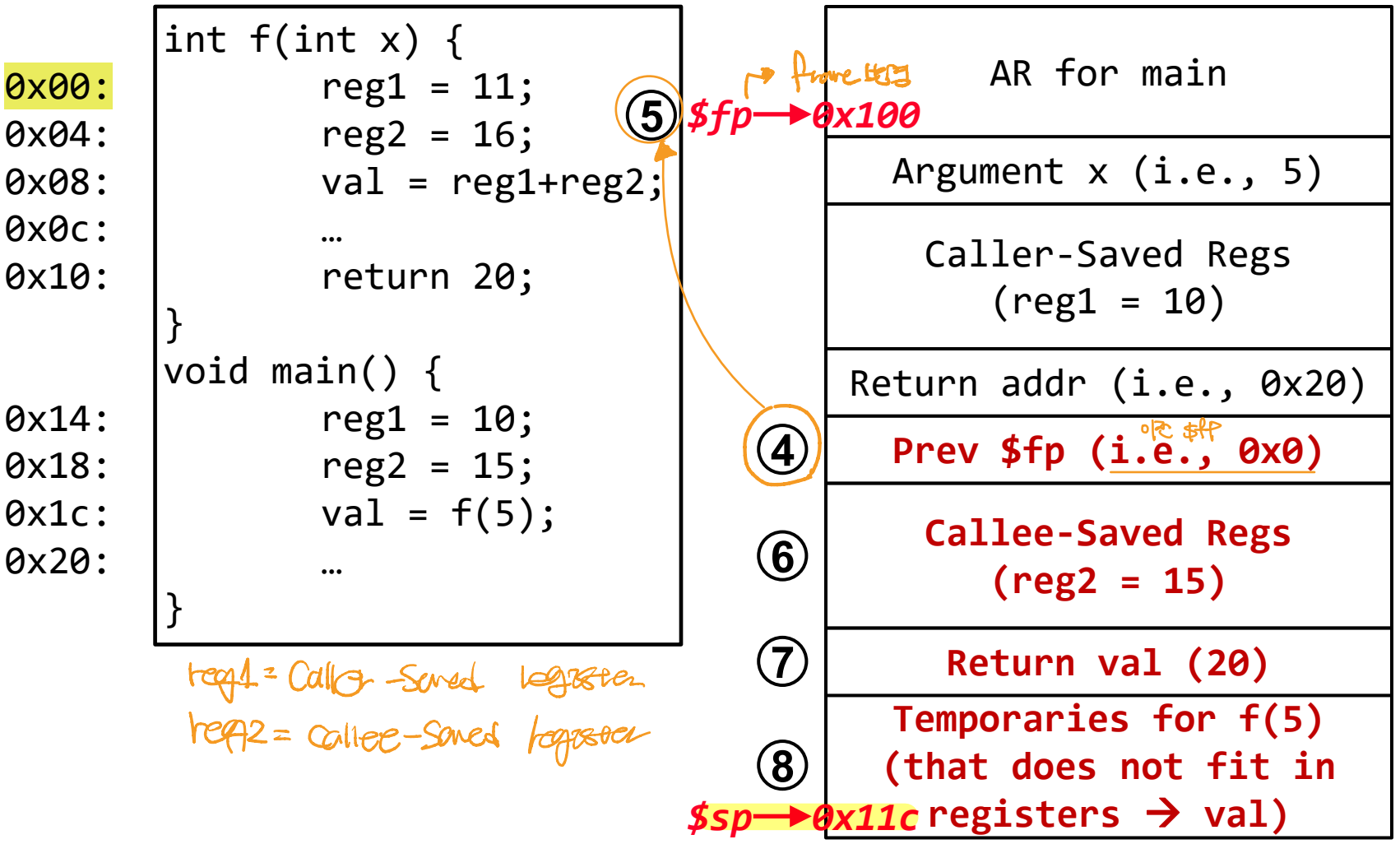


Calling a function - 1

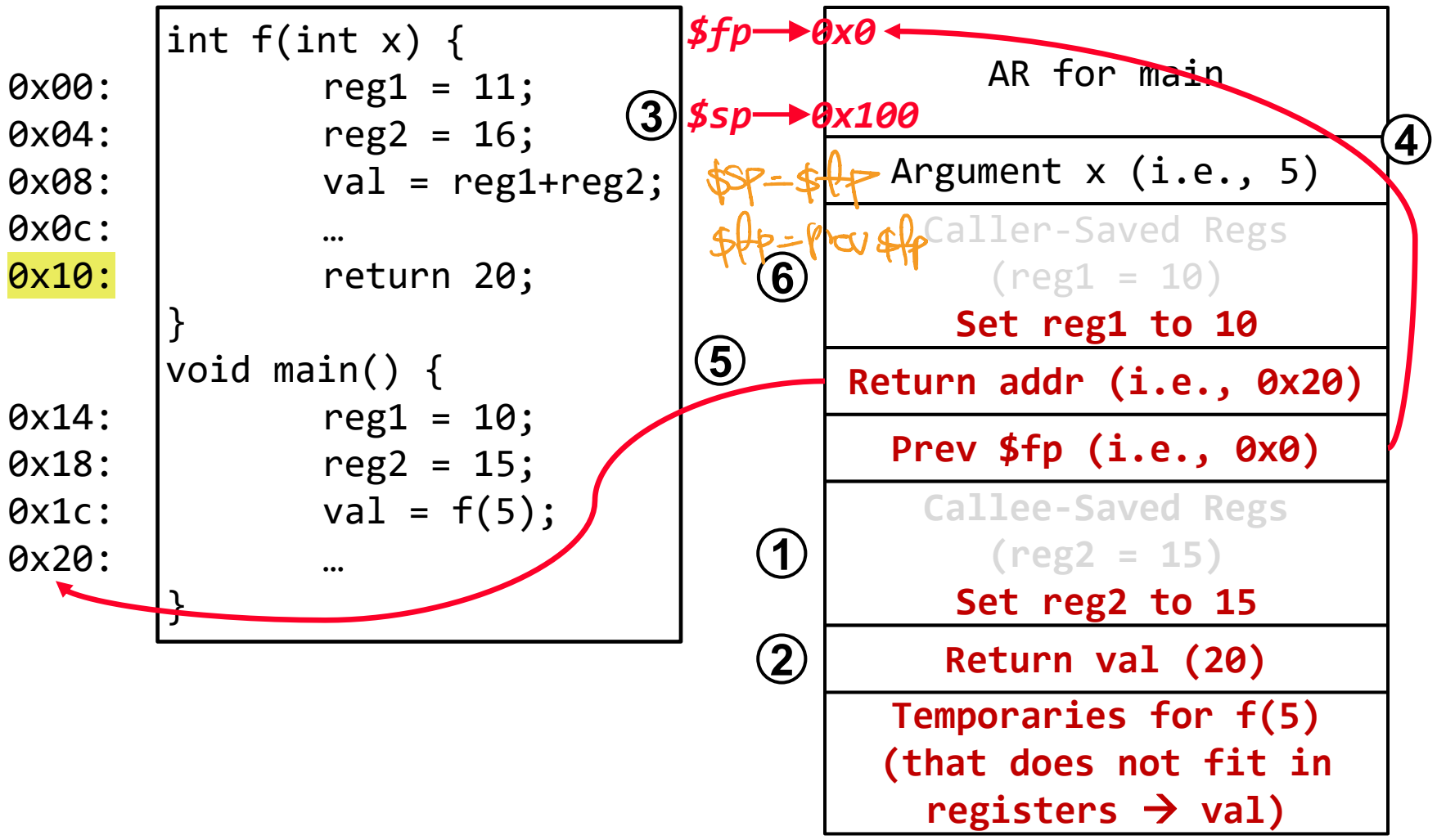
```
0x00:   int f(int x) {
0x00:           reg1 = 11;
0x04:           reg2 = 16;
0x08:           val = reg1+reg2;
0x0c:           …
0x10:           return 20;
        }
        void main() {
0x14:           reg1 = 10;
0x18:           reg2 = 15;
0x1c:           val = f(5);
0x20:           …
        }
```



Calling a function - 2

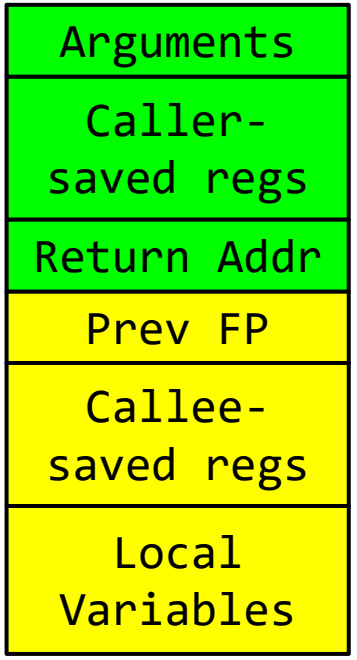


Returning a function



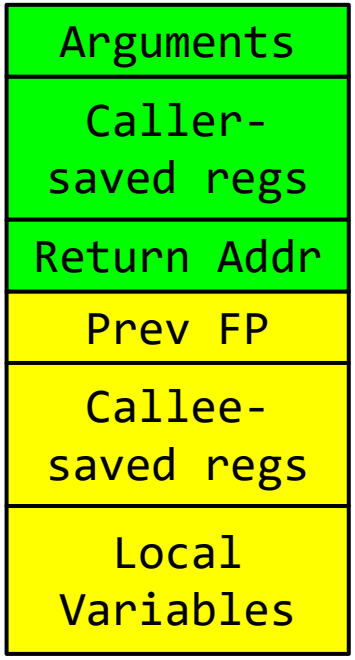
Wrap up - 1

- Code before call instruction
  - Push each actual parameter
  - Push caller-saved registers
  - Push return address (current PC) and jump to callee code
- Prologue = code at function entry
  - Push dynamic link (i.e., FP)
  - Old stack pointer becomes new frame pointer
  - Push callee-saved registers
  - Push local variables



Wrap up - 2

- Epilogue = code at return instruction
  - Pop (restore) callee-saved registers
  - Store return value at appropriate place
  - Restore old stack pointer (pop callee frame)
  - Pop old frame pointer
  - Pop return address and jump to that address
- Code after call
  - Pop (restore) caller-saved registers
  - Use return value



You will learn further details in the compiler class (next semester)

Something to think about!

- Why do we need to store return value to the stack? Why not use registers (e.g., r2, r3)?
  - A. The program may return complex values

    r2, r3은 복잡한 값을 return하여 못하기 때문이다.
- Why do we store arguments to the stack? Why not use registers (e.g., r4~r7)?
  - A. The function may need multiple arguments
  - A. The function may call another function (which will overwrite r4~r7)

- Why are some registers caller-saved while others are callee-saved?

Caller and Callee Saved Registers

→ function call 에서 Register 값이 변하지 않았다는 것을 의미

- **Caller-saved registers (AKA volatile registers, or call-clobbered):** used to hold temporary quantities that need not be preserved across calls
  - It's the caller's responsibility to save / restore the registers (if the caller wants)

- **Callee-saved registers (AKA non-volatile registers, or call-preserved):** used to hold long-lived values that should be preserved across calls
  - It's the callee's responsibility to save / restore the registers (necessary)

Question?

Announcements: *Will skip chapter 3 (arithmetic)*
and jump to Chapter 4 (edition 6) in the next lecture

Reading: ***P&H Ch 2***

Handouts: *None*