Computer Architecture

3. Instructions: Language of the Machine

Young Geun Kim
(younggeun_kim@korea.ac.kr)
Intelligent Computer Architecture & Systems Lab.

Programming Languages

- **There are many programming languages, but they usually fall into two categories.**
  - **High-level Languages**
    - Usually machine independent 어떤 CPU 에서 쓰일지 고민하지 않아도 됨.
    - Statements (or Grammar) are often more expressive
    - C, C++, Java, Python
  - **Low-level Languages**
    - Usually machine specific
    - Offer much finer-grained statements that closely match the machine language of the target processor
    - Assembly languages for x86, ARM, RISC-V

Assembly Languages

- Text representation of the machine language
- One statement represents one machine instruction
- Abstraction-layer between high-level programs and machine code

Machine Languages

- **The native language of the computer**

- **Bit representation of machine operations to be executed by hardware**

  - **Commands (or Instructions)**

– Go one step forward	00
– Go one step backward	01
– Turn left	10
– Turn right	11

matching

Assembly Codes

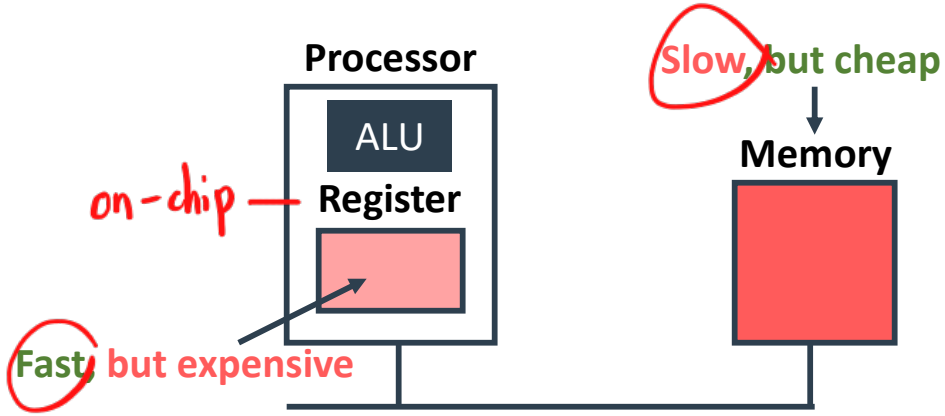
Machine Codes

instructions

Instruction Set Architecture (ISA)

같은 기계어라도 ISA에 따라 대응하는 instruction이 다름.

- **Interface between Hardware and Software**
- **An Abstract Data Type**
  - Objects: Register & Memory
  - Operations: Instructions



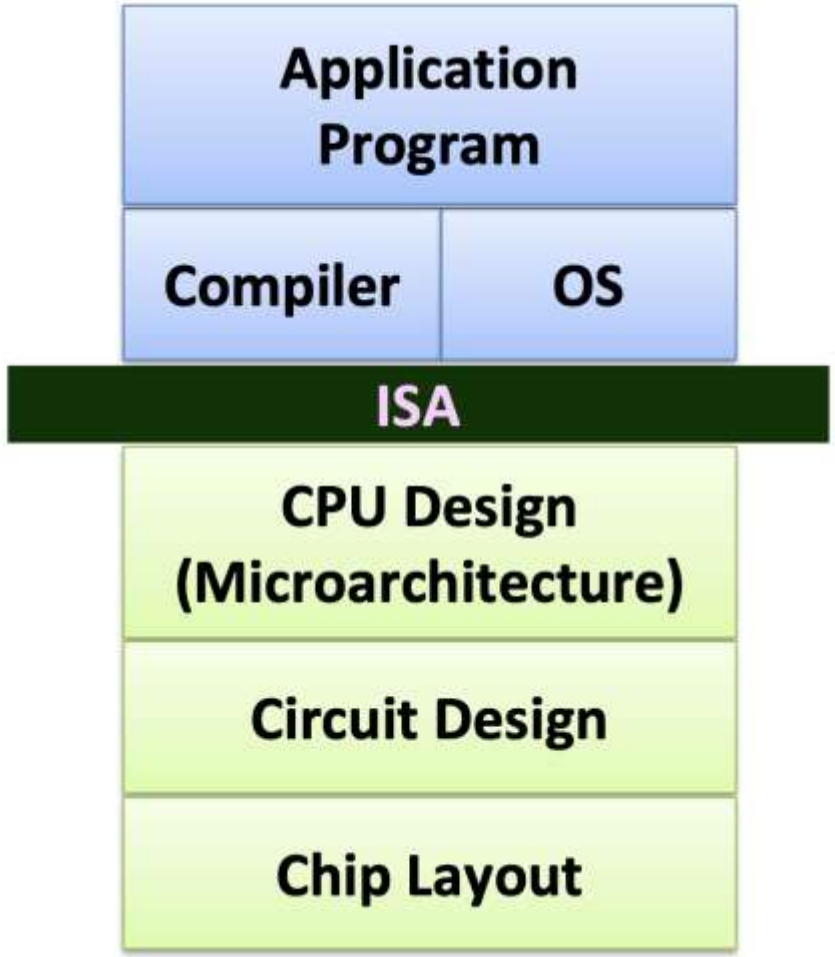
1. Memory -> Register
2. Execute
3. Register -> Memory

\+ Control Sequences

- **Goal of Instruction Set Architecture**
  - To allow high-performance & low-cost hardware implementations while satisfying constraints imposed by software including operating systems and compiler

Instruction Set Architecture (ISA): Assembly ↔ Machine code, register entries usages

- **Above: how to program machine**
  - Processors execute instructions in sequence
- **Below: what needs to be built**
  - Use variety of tricks to make it run fast



- **Instruction set = Operation**
- **Processor registers**
- **Memory addressing modes**
- **Data types and representations**
- **Byte ordering, …**

The RISC-V Instruction Set

- **A completely open ISA that is freely available to academia and industry**
- **Fifth RISC ISA design developed at UC Berkeley**
  - RISC-I (1981), RISC-II (1983), SOAR (1984), SPUR (1989), and RISC-V (2010)

- **Now managed by the RISC-V Foundation (http://riscv.org)**
- **Typical of many modern ISAs**
  - See RISC-V Reference Card (or Green Card)

- **Similar ISAs have a large share of embedded core market**
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …

Why Freely Open ISA?

- **Greater innovation via free-market competition**
  - From many core designers, closed-source and open-source
- **Shared open core designs**
  - Shorter time to market, lower cost from reuse, fewer errors given more eyeballs, transparency makes it difficult for government agencies to add secret trap doors
- **Processors becoming affordable for more devices**
  - Help expand the Internet of Things (IoTs), which could cost as little as $1
- **Software stack survive for long time**
- **Make architectural research and education more real**
  - Fully open hardware and software stacks

RISC-V: Operations

- **Perform an arithmetic or logical function on register data**

- **Transfer data between memory and register**
  - Load data from memory into register
  - Store register data into memory

- **Transfer control**
  - Unconditional jump
  - Conditional branch
  - Procedure call and return

RISC-V: Registers

#	Name	Usage
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporaries (Caller-save registers)
x6	t1	
x7	t2	
x8	s0/fp	Saved register / Frame pointer
x9	s1	Saved register
x10	a0	Function arguments / Return values
x11	a1	
x12	a2	Function arguments
x13	a3	
x14	a4	
x15	a5	

#	Name	Usage
x16	a6	Function arguments
x17	a7	
x18	s2	Saved registers (Callee-save registers)
x19	s3	
x20	s4	
x21	s5	
x22	s6	
x23	s7	
x24	s8	
x25	s9	
x26	s10	
x27	s11	
x28	t3	Temporaries (Caller-save registers)
x29	t4	
x30	t5	
x31	t6	
	pc	Program counter

Registers vs. Memory

- **Registers are faster to access than memory**

  but much smaller (few KB)

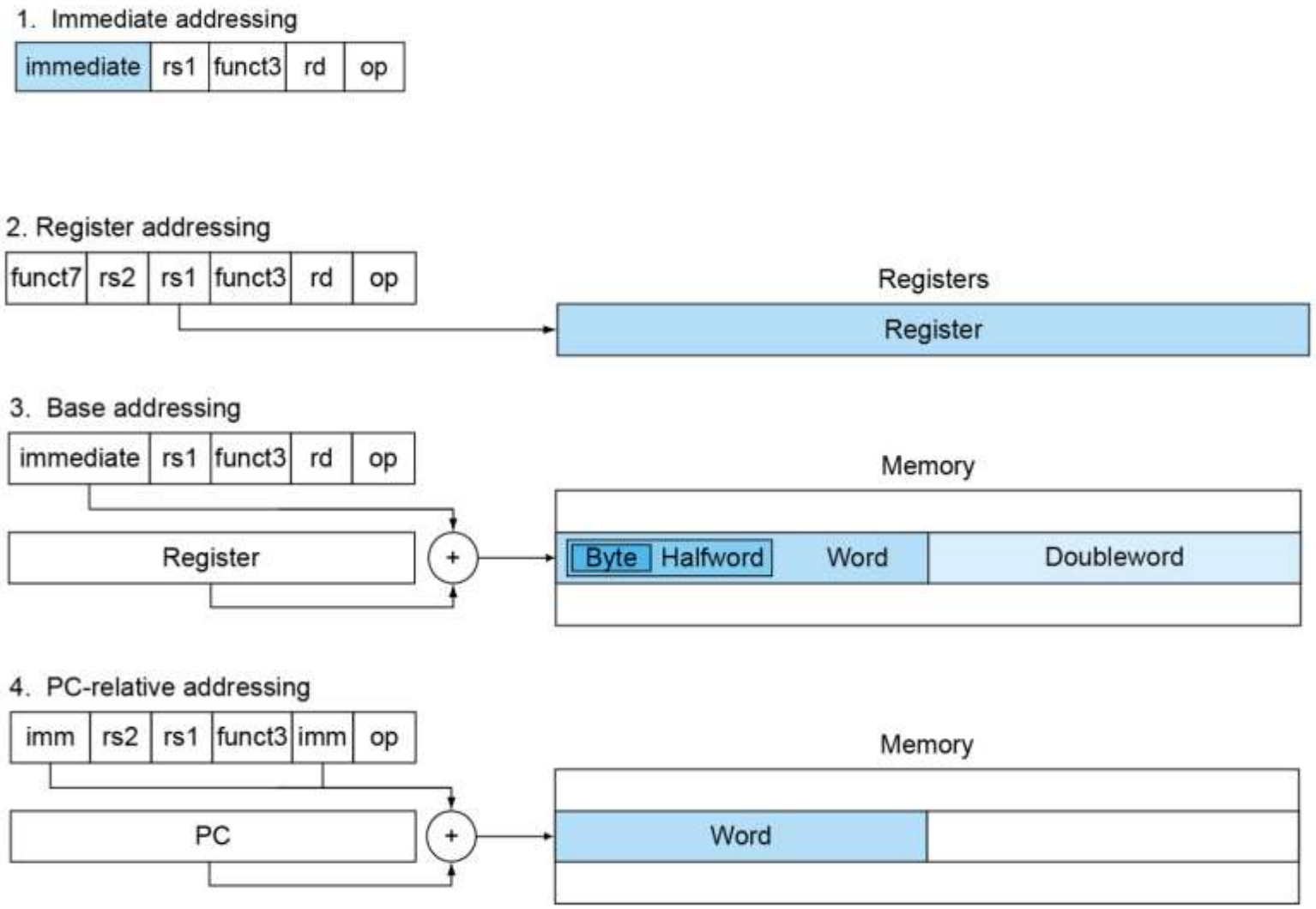
- **In RISC-V, data in memory cannot be directly addressed by arithmetic or logical instructions**

- **Operating on memory data requires data transfer instructions**
  - More instructions to be executed

- **Compiler must use registers for variables as much as possible**
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

RISC-V: Addressing

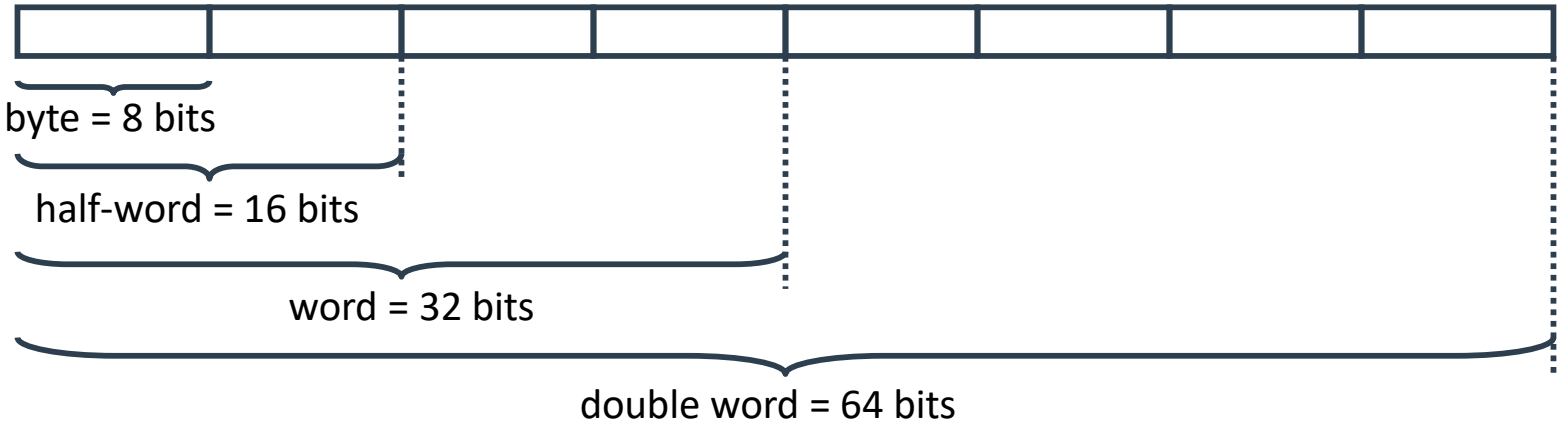
- **How is the data specified?**
  - Immediate value (constant)
  - Value in a register
  - Value in memory
    - Mem[rsl + imm]
  - Address: pc + imm



Data Representation

- **How to represent the value of variables?**
  - Bits: 0 or 1 (binary representation)

- **Bit strings: sequence of bits**



- **Characters: 1 byte, usually using ASCII**
- **Integers: 4 bytes, stored in 2's complement**

RISC-V: Arithmetic & Logical Operations

Arithmetic Operations

- **Add and subtract, three operands**
  - Two sources and one destination

dest sources

```
add a, b, c          // a ← b + c
sub a, b, c          // a ← b - c
```

- **All arithmetic operations have this form**

- **Design Principle 1: Simplicity favors regularity**
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Register Operands

- **Arithmetic instructions use register operands**

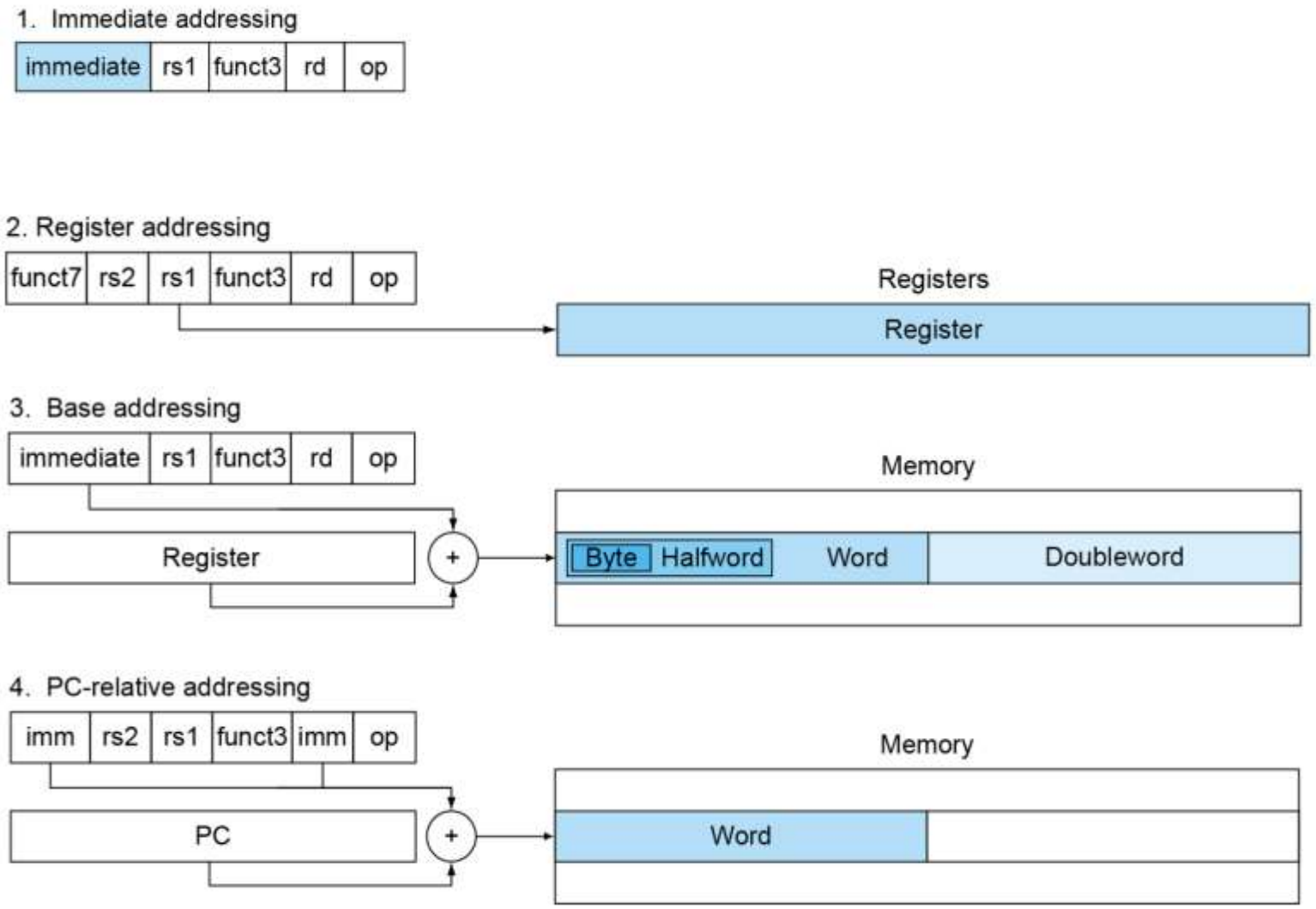
# of entries each entry

- **RISC-V has a 32 * 64-bit register file: x0 ~ x31**
  - Use for frequently accessed data
  - 64-bit data is called a “double word”
  - 32-bit data is called a “word”

- **Design Principle 2: Smaller is faster**
  - Main memory has millions of locations

RISC-V: Addressing

- **How is the data specified?**
  - Immediate value (constant)
  - Value in a register
  - Value in memory
    - Mem[rsl + imm]
  - Address: pc + imm



Register Operand Example

C code:

```
// f in x19
// g in x20
// h in x21
// i in x22
// j in x23

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

add sub add

Compiled RISC-V code:

Temporaries

```
add  x5, x20, x21
add  x6, x22, x23
sub  x19, x5, x6
```

RISC-V: Registers

#	Name	Usage
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporaries (Caller-save registers)
x6	t1	
x7	t2	
x8	s0/fp	Saved register / Frame pointer
x9	s1	Saved register
x10	a0	Function arguments / Return values
x11	a1	
x12	a2	Function arguments
x13	a3	
x14	a4	
x15	a5	

#	Name	Usage
x16	a6	Function arguments
x17	a7	
x18	s2	Saved registers (Callee-save registers)
x19	s3	
x20	s4	
x21	s5	
x22	s6	
x23	s7	
x24	s8	
x25	s9	
x26	s10	
x27	s11	
x28	t3	Temporaries (Caller-save registers)
x29	t4	
x30	t5	
x31	t6	
	pc	Program counter

Immediate Operands

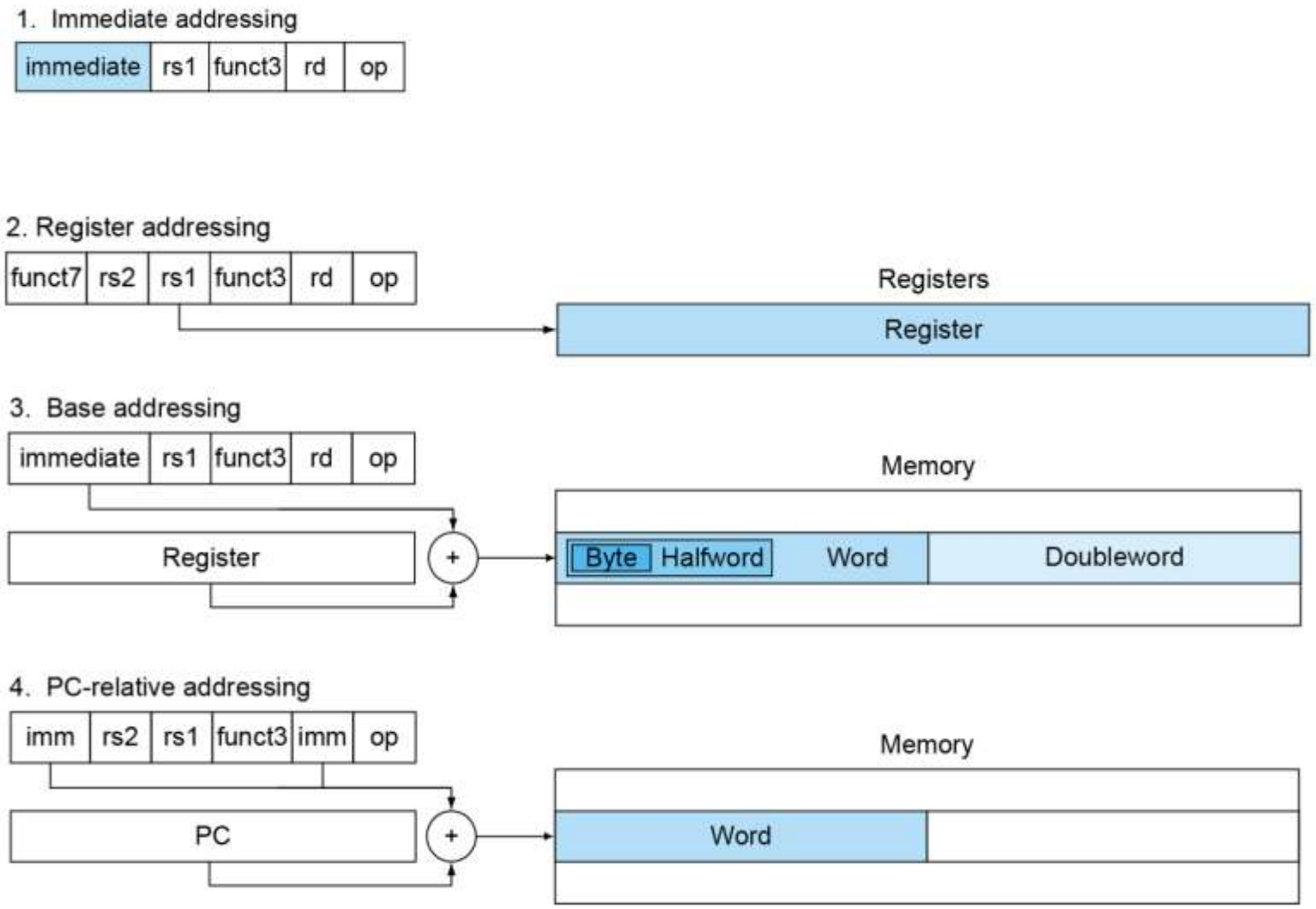
- **Constant data specified in an instruction**

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

- **Why using immediate operands?**
  - Small constants are common (limited to 12 bits)
    - 64-bit register is not really required for all the arithmetic operations
  - Immediate operand avoids a load instruction

RISC-V: Addressing

- **How is the data specified?**
  - Immediate value (constant)
  - Value in a register
  - Value in memory
    - Mem[rsl + imm]
  - Address: pc + imm



Logical Operations

- **Instructions for bitwise manipulation**

Operation	C	Java	RISC-V
Shift left	<<	<<	sll, slli
Shift right (arithmetic)	>>	>>	sra, srai
Shift right (logical)	>>	>>>	srl, srli
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	

sra : 1010 → 1101 (signed extension) / srl : 1010 → 0101 (always 0)

- **Useful for extracting and inserting groups of bits in a word**

AND Operations

- **Useful to mask bits in a word**
  - Select some bits, reset others to 0

```
and x9, x10, x11
```

x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x11	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
x9	00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000

OR Operations

- **Useful to include bits in a word**
  - Set selected bits to 1, leave others unchanged

```
or  x9, x10, x11
```

x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x11	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
x9	00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000

XOR Operations

- **Differencing operations**
  - Reset when bits are the same, set if they are different

```
xor x9, x10, 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

Arithmetic Operations

Instruction	Type	Example	Meaning
Add	R	`add rd, rs1, rs2`	`R[rd] = R[rs1] + R[rs2]`
Subtract	R	`sub rd, rs1, rs2`	`R[rd] = R[rs1] – R[rs2]`
Add immediate	I	`addi rd, rs1, imm12`	`R[rd] = R[rs1] + SignExt(imm12)`
Set less than	R	`slt rd, rs1, rs2`	`R[rd] = (R[rs1] < R[rs2])? 1 : 0`
Set less than immediate	I	`slti rd, rs1, imm12`	`R[rd] = (R[rs1] < SignExt(imm12))? 1 : 0`
Set less than unsigned	R	`sltu rd, rs1, rs2`	`R[rd] = (R[rs1]` $<_u$ `R[rs2])? 1 : 0`
Set less than immediate unsigned	I	`sltiu rd, rs1, imm12`	`R[rd] = (R[rs1]` $<_u$ `SignExt(imm12))? 1 : 0`
Load upper immediate	U	`lui rd, imm20`	`R[rd] = SignExt(imm20 << 12)`
Add upper immediate to PC	U	`auipc rd, imm20`	`R[rd] = PC + SignExt(imm20 << 12)`

Upper Immediate Operations

- **32-bit Constants**
  - When 12-bit immediate is not sufficient

```
lui  rd, constant
```

- **For the occasional 32-bit constant:**
  - Copy 20-bit constant to bits [31:12] of rd
  - Extend bit 31 to bits [63:32]
  - Clear bits [11:0] of rd to 0

- **Example: x19 <- 0x003D0500**

(signed extension)

32-bit 20-bit 12-bit

```
lui  x19, 0x003D0
addi x19,x19,0x500
```

20-bit

12-bit

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0000 0000 0000
0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0101 0000 0000

Logical Operations

Instruction	Type	Example	Meaning
AND	R	`and rd, rs1, rs2`	`R[rd] = R[rs1] & R[rs2]`
OR	R	`or rd, rs1, rs2`	`R[rd] = R[rs1] | R[rs2]`
XOR	R	`xor rd, rs1, rs2`	`R[rd] = R[rs1] ^ R[rs2]`
AND immediate	I	`andi rd, rs1, imm12`	`R[rd] = R[rs1] & SignExt(imm12)`
OR immediate	I	`ori rd, rs1, imm12`	`R[rd] = R[rs1] | SignExt(imm12)`
XOR immediate	I	`xori rd, rs1, imm12`	`R[rd] = R[rs1] ^ SignExt(imm12)`
Shift left logical	R	`sll rd, rs1, rs2`	`R[rd] = R[rs1] << R[rs2]`
Shift right logical	R	`srl rd, rs1, rs2`	`R[rd] = R[rs1] >> R[rs2]` *(logical)*
Shift right arithmetic	R	`sra rd, rs1, rs2`	`R[rd] = R[rs1] >> R[rs2]` *(arithmetic)*
Shift left logical immediate	I	`slli rd, rs1, shamt`	`R[rd] = R[rs1] << shamt`
Shift right logical imm.	I	`srli rd, rs1, shamt`	`R[rd] = R[rs1] >> shamt` *(logical)*
Shift right arithmetic immediate	I	`srai rd, rs1, shamt`	`R[rd] = R[rs1] >> shamt` *(arithmetic)*

Example: Arithmetic Operations

```
long arith (long x,   a0
            long y,   a1
            long z) { a2
    long t1 = x + y;
    long t2 = z + t1;
    long t3 = x + 4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 - t5;
    return rval;
}
```

```
arith:
    add    a5, a0, a1          # a5 = x + y (t1)
    add    a2, a5, a2          # a2 = t1 + z (t2)
    addi   a0, a0, 4           # a0 = x + 4 (t3)
    slli   a5, a1, 1           # a5 = y * 2
    add    a1, a5, a1          # a1 = a5 + y
    slli   a5, a1, 4           # a5 = a1 * 16 (t4)
    add    a0, a0, a5          # a0 = t3 + t4 (t5)
    sub    a0, a2, a0          # a0 = t2 – t5 (rval)
    ret
```

```
x in a0
y in a1
z in a2
```

Example: Logical Operations

```
long logical (long x,
              long y) {
    long t1 = x ^ y;
    long t2 = t1 >> 17;
    long mask = (1 << 8) - 7;
    long rval = t2 & mask;
    return rval;
}
```

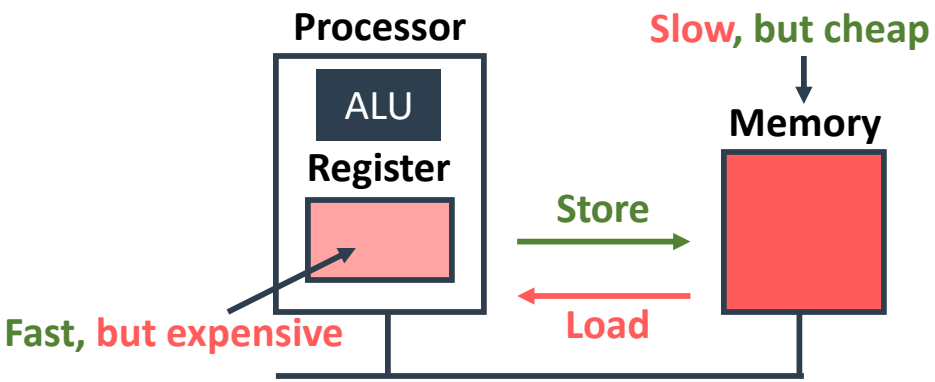
```
logical:
   xor   a0, a0, a1       # a0 = x ^ y (t1)
   srai  a0, a0, 17       # a0 = t1 >> 17 (t2)
   andi  a0, a0, 249      # a0 = t2 & ((1 << 8) - 7)
   ret
```

```
x in a0
y in a1
```

RISC-V: Data Transfer Operations

Memory Operands

- **Main memory used for composite data**
  - Arrays, structures, dynamic data
- **To apply arithmetic operations**



Load

1. **Memory -> Register**
2. **Execute = ALU Ops**
3. **Register -> Memory**

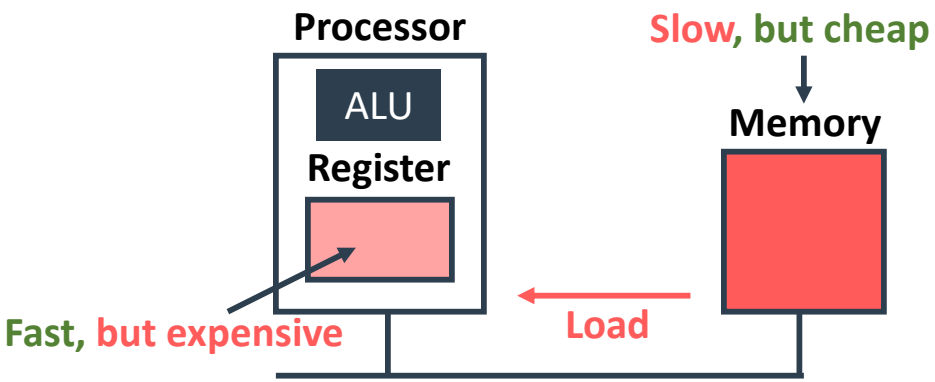
Store

- **Memory is byte addressed: each address identifies an 8-bit byte**

- **RISC-V is Little Endian**
  - Least-significant byte -> Least address of a word
- **RISC-V does not require words to be aligned in memory**

Putting data in Registers

- **Data transfer instructions**



Load

1. **Memory -> Register**
2. **Execute**
3. **Register -> Memory**

\+ **Control Sequences**

- **One double-word is loaded from memory to a register on RISC-V using the ld instruction**
- **Load instructions have three parts**
  - Operator name
  - Destination register (dst)
  - Base register address (base) and constant offset (off)

    ld dst, off(base)
- **Offset value is signed (use uld for unsigned)**

Memory Operand Example

C code:

```
// h in x21
// base address of A in x22

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

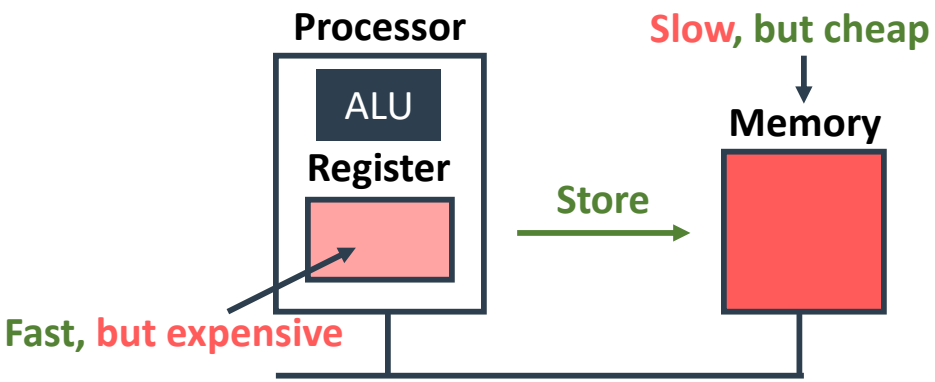
Compiled RISC-V code:

```
// 8 bytes per doubleword
// &A[8] = A + 64

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

Putting data in Memory

- Data transfer instructions



1. Memory -> Register
2. Execute
3. Register -> Memory

Store

- Storing data is just the reverse of "load data" and the instruction is nearly identical
- Use the sd instruction to copy a double-word from the source register to an address in memory

  sd src, off(base)
- Offset value is signed (use usd for unsigned)

Memory Operand Example

C code:

```
// h in x21
// base address of A in x22

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

Compiled RISC-V code:

```
// 8 bytes per doubleword
// &A[8] = A + 64

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

Byte/Halfword/Word Operations

- **Load byte/halfword/word: Sign extend to 64 bits in rd**

```
lb    rd, offset(rs1)
lh    rd, offset(rs1)
lw    rd, offset(rs1)
```

- **Load byte/halfword/word: Zero extend to 64bits in rd**

```
lbu   rd, offset(rs1)
lhu   rd, offset(rs1)
lwu   rd, offset(rs1)
```

- **Store byte/halfword/word: Store rightmost 8/16/32 bits**

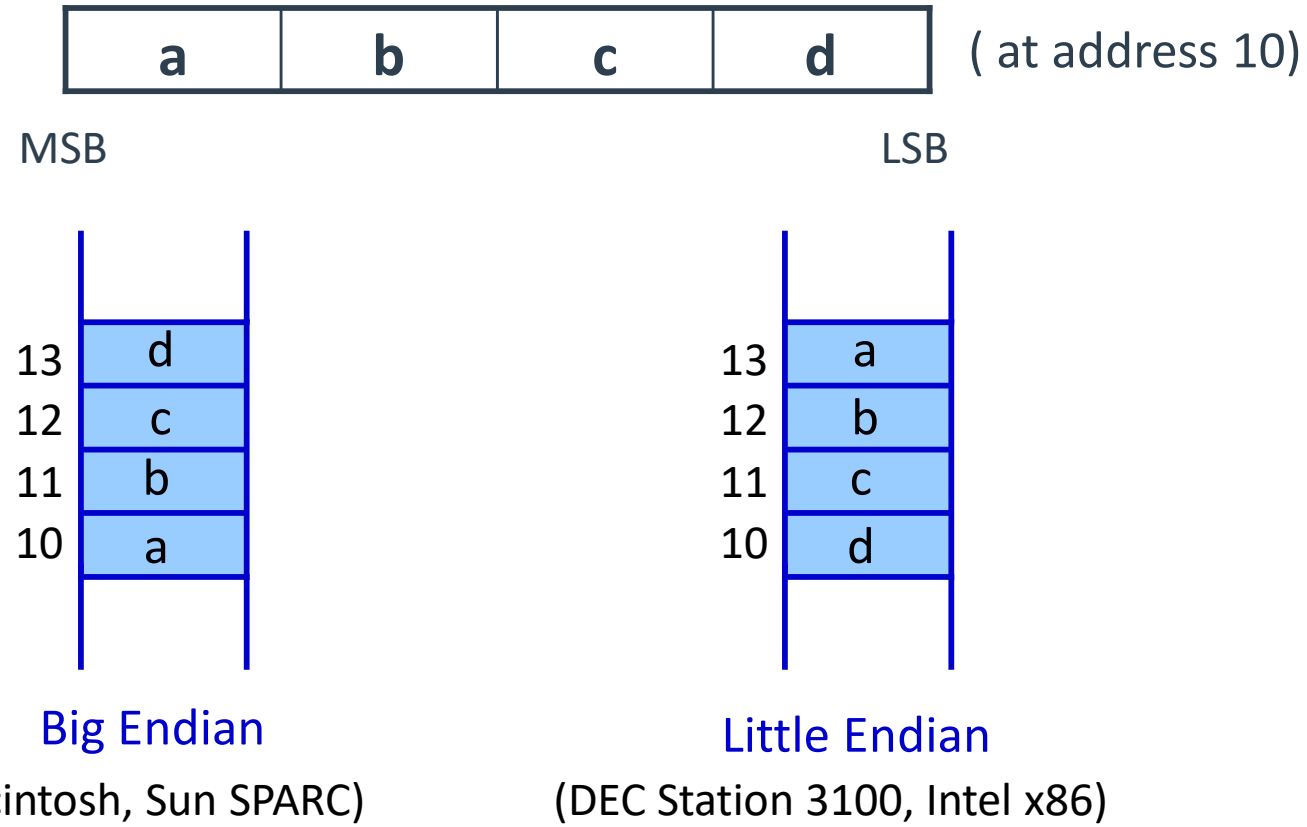
```
sb    rs2, offset(rs1)
sh    rs2, offset(rs1)
sw    rs2, offset(rs1)
```

Data Transfer Operations

Instruction	Type	Example	Meaning
Load doubleword	**I**	`ld rd, imm12(rs1)`	R[rd] = Mem_8[R[rs1] + SignExt(imm12)]
Load word	**I**	`lw rd, imm12(rs1)`	R[rd] = SignExt(Mem_4[R[rs1] + SignExt(imm12)])
Load halfword	**I**	`lh rd, imm12(rs1)`	R[rd] = SignExt(Mem_2[R[rs1] + SignExt(imm12)])
Load byte	**I**	`lb rd, imm12(rs1)`	R[rd] = SignExt(Mem_1[R[rs1] + SignExt(imm12)])
Load word unsigned	**I**	`lwu rd, imm12(rs1)`	R[rd] = ZeroExt(Mem_4[R[rs1] + SignExt(imm12)])
Load halfword unsigned	**I**	`lhu rd, imm12(rs1)`	R[rd] = ZeroExt(Mem_2[R[rs1] + SignExt(imm12)])
Load byte unsigned	**I**	`lbu rd, imm12(rs1)`	R[rd] = ZeroExt(Mem_1[R[rs1] + SignExt(imm12)])
Store doubleword	**S**	`sd rs2, imm12(rs1)`	Mem_8[R[rs1] + SignExt(imm12)] = R[rs2]
Store word	**S**	`sw rs2, imm12(rs1)`	Mem_4[R[rs1] + SignExt(imm12)] = R[rs2](31:0)
Store halfword	**S**	`sh rs2, imm12(rs1)`	Mem_2[R[rs1] + SignExt(imm12)] = R[rs2](15:0)
Store byte	**S**	`sb rs2, imm12(rs1)`	Mem_1[R[rs1] + SignExt(imm12)] = R[rs2](7:0)

Byte Ordering

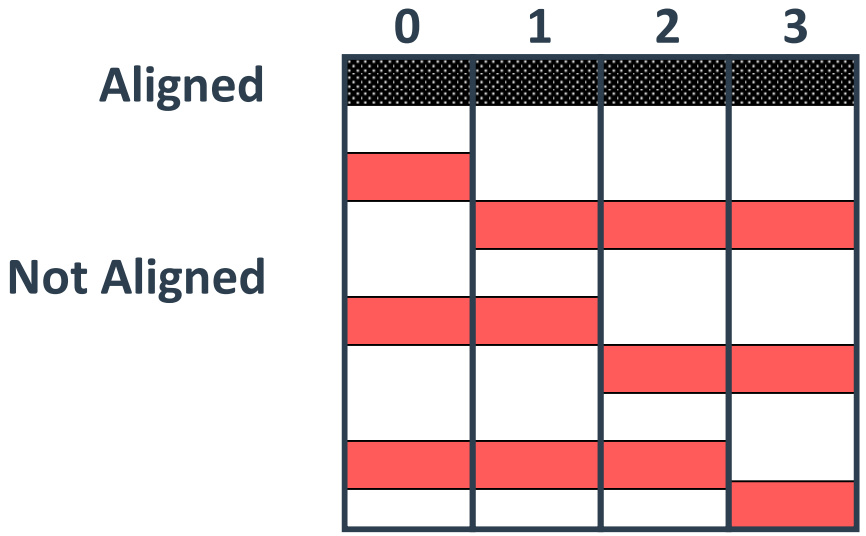
- **Two basic ways of ordering bits**



- **Some machines such as MIPS can do both, but primarily big endian**

Alignment Restrictions

- **For an alignment restricted architecture, data is required to fall on addresses that are even multiples of the data size**
- **Historically**
  - Early machines (IBM360 in 1964) required alignment
  - Removed in 1970s since hard for programmers
  - RISC introduced due to effect on performance
- **Example: word-level alignment**



Swap Example

- Source code in C:

a0 a1

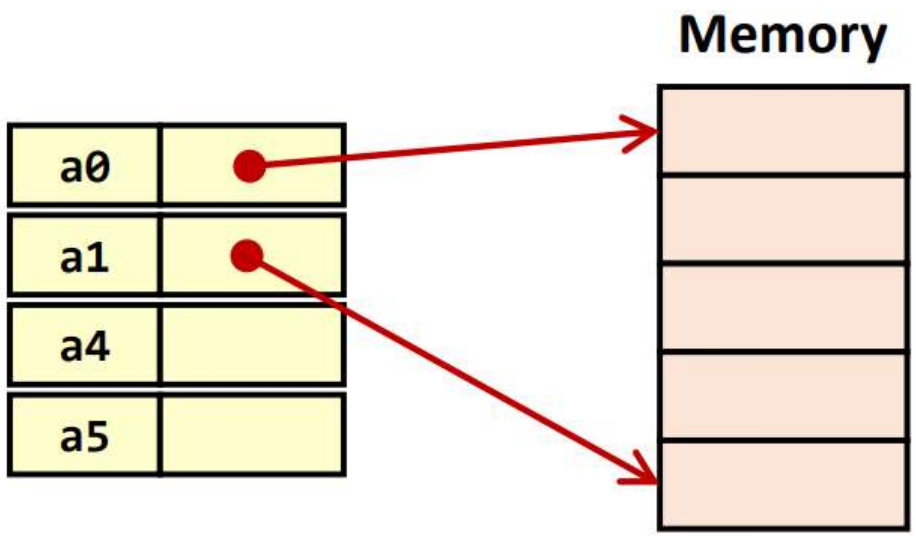
```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

- Corresponding assembly code:

```
swap:
    ld      a4, 0(a0)
    ld      a5, 0(a1)
    sd      a5, 0(a0)
    sd      a4, 0(a1)
    ret
```

Understanding Swap (1)

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```



Register Allocation (By compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

```
swap:
    ld      a4, 0(a0)      # t0 = *xp
    ld      a5, 0(a1)      # t1 = *yp
    sd      a5, 0(a0)      # *xp = t1
    sd      a4, 0(a1)      # *yp = t0
    ret
```

Understanding Swap (2)

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

a0	0x120
a1	0x100
a4	
a5	

Memory

0x120	123
0x118	
0x110	
0x108	
0x100	456

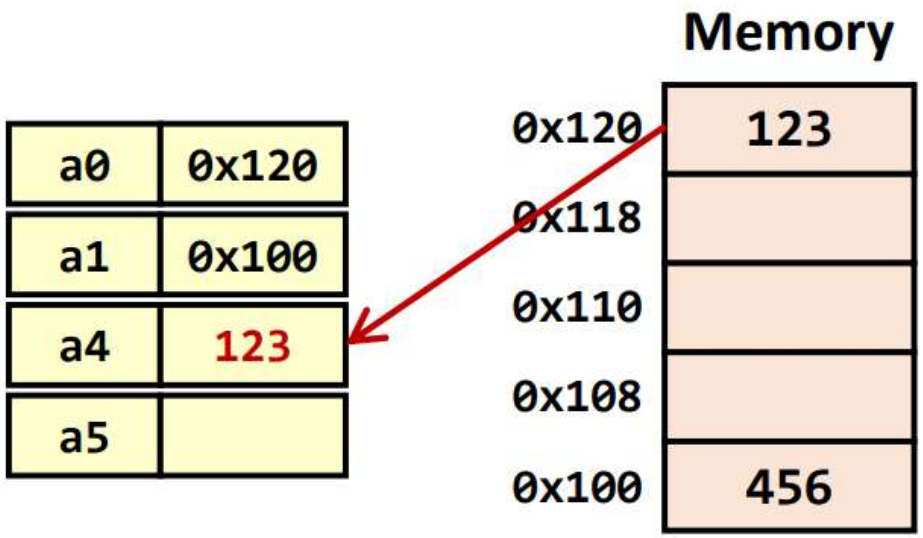
Register Allocation (By compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

```
swap:
    ld      a4, 0(a0)      # t0 = *xp
    ld      a5, 0(a1)      # t1 = *yp
    sd      a5, 0(a0)      # *xp = t1
    sd      a4, 0(a1)      # *yp = t0
    ret
```

Understanding Swap (3)

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```



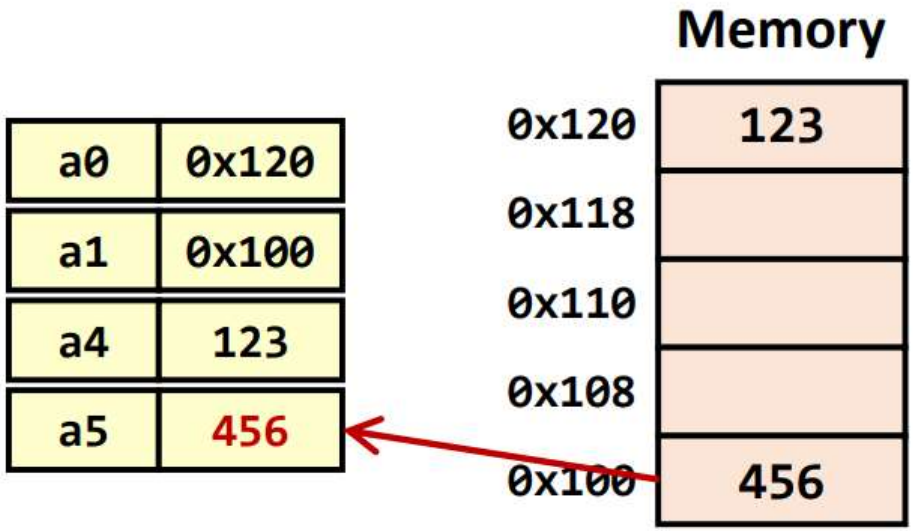
Register Allocation (By compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

```
swap:
    ld      a4, 0(a0)      # t0 = *xp
    ld      a5, 0(a1)      # t1 = *yp
    sd      a5, 0(a0)      # *xp = t1
    sd      a4, 0(a1)      # *yp = t0
    ret
```

Understanding Swap (4)

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```



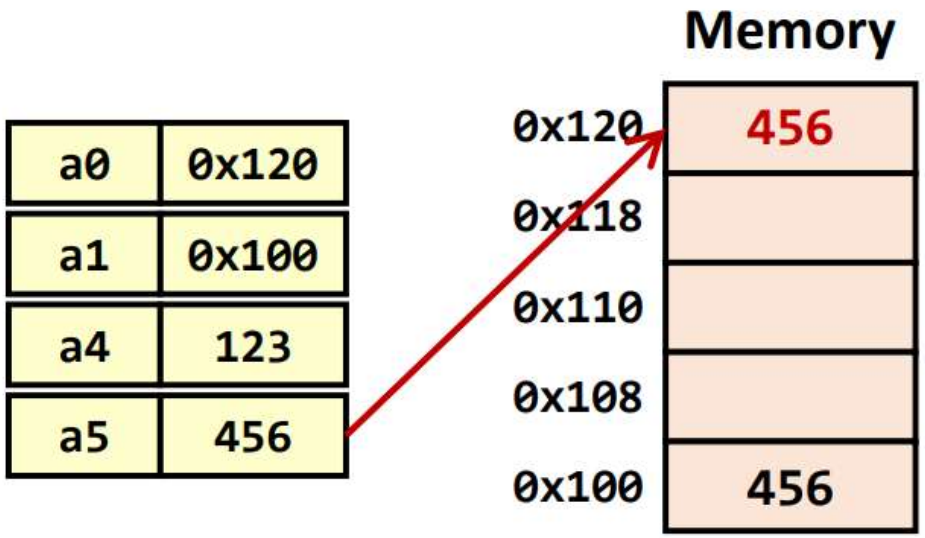
Register Allocation (By compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

```
swap:
    ld      a4, 0(a0)      # t0 = *xp
    ld      a5, 0(a1)      # t1 = *yp
    sd      a5, 0(a0)      # *xp = t1
    sd      a4, 0(a1)      # *yp = t0
    ret
```

Understanding Swap (5)

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```



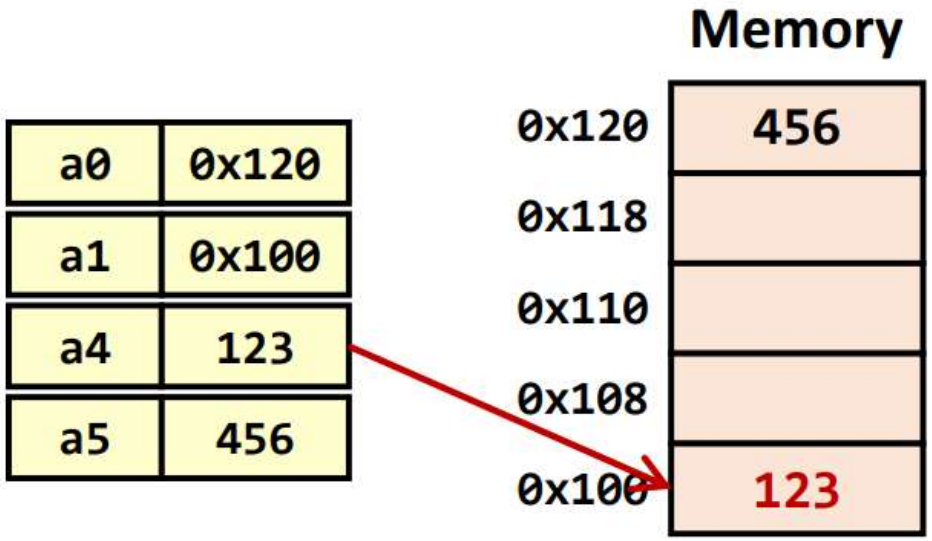
Register Allocation (By compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

```
swap:
    ld      a4, 0(a0)      # t0 = *xp
    ld      a5, 0(a1)      # t1 = *yp
    sd      a5, 0(a0)      # *xp = t1
    sd      a4, 0(a1)      # *yp = t0
    ret
```

Understanding Swap (6)

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```



Register Allocation (By compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

```
swap:
    ld      a4, 0(a0)      # t0 = *xp
    ld      a5, 0(a1)      # t1 = *yp
    sd      a5, 0(a0)      # *xp = t1
    sd      a4, 0(a1)      # *yp = t0
    ret
```