RISC-V Architecture I

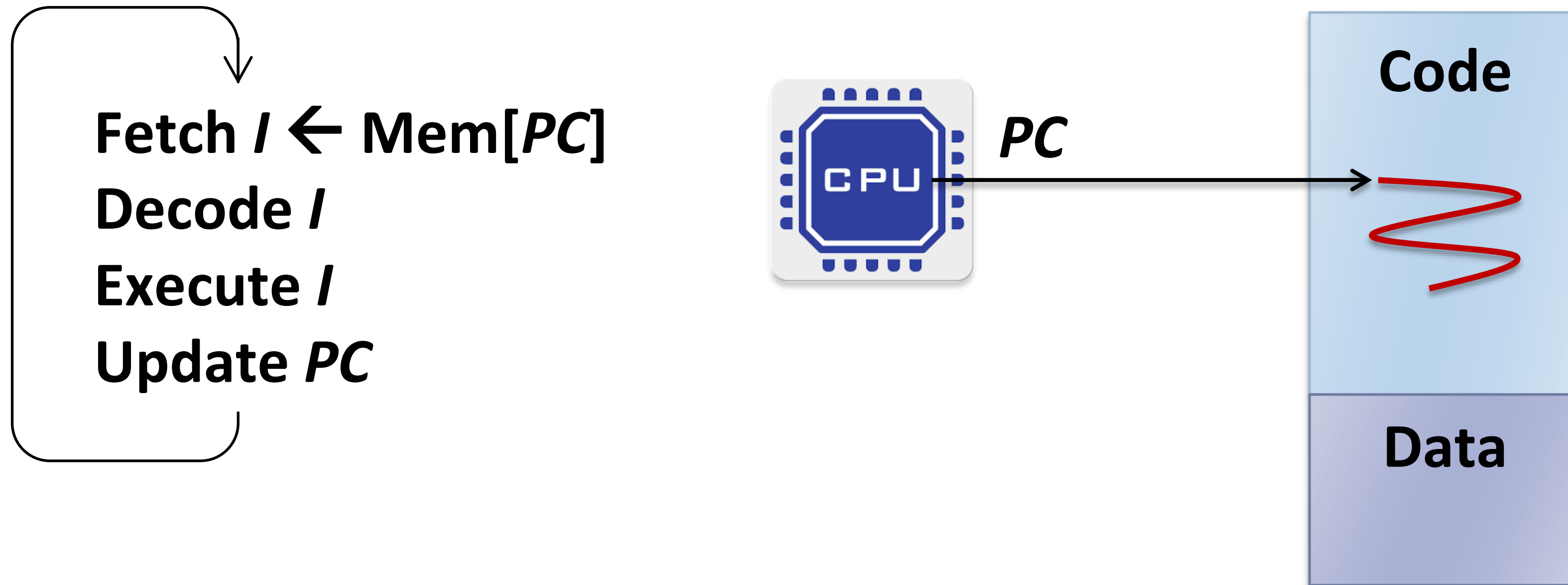
Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)
Systems Software &
Architecture Lab.
Seoul National University

Fall 2022

Introduction to RISC-V

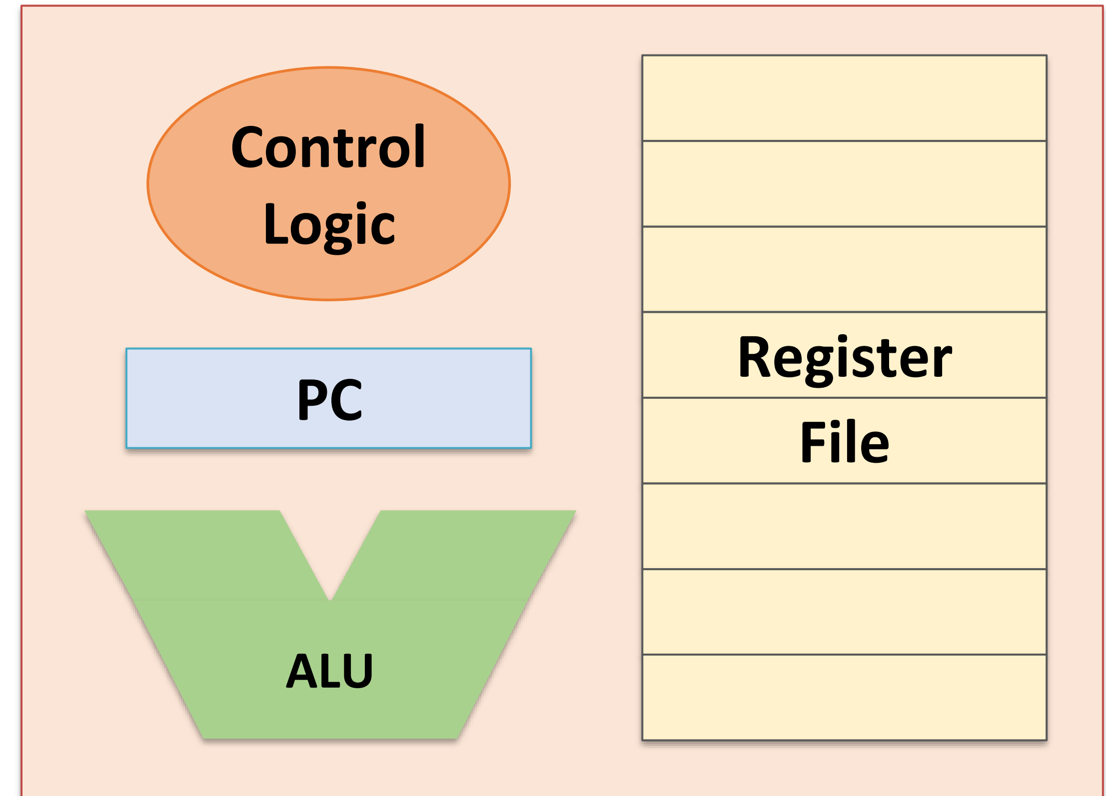
Chap. 2.1

The (Dumb) Life of CPU



CPU

- Central Processing Unit
  - PC (Program Counter)
    - Address of next instruction
  - Register file
    - Temporary storage for heavily-used data
  - ALU (Arithmetic & Logic Unit)
    - Arithmetic operations
    - Logical operations
  - Control logic
    - Control instruction fetch, decoding and execution



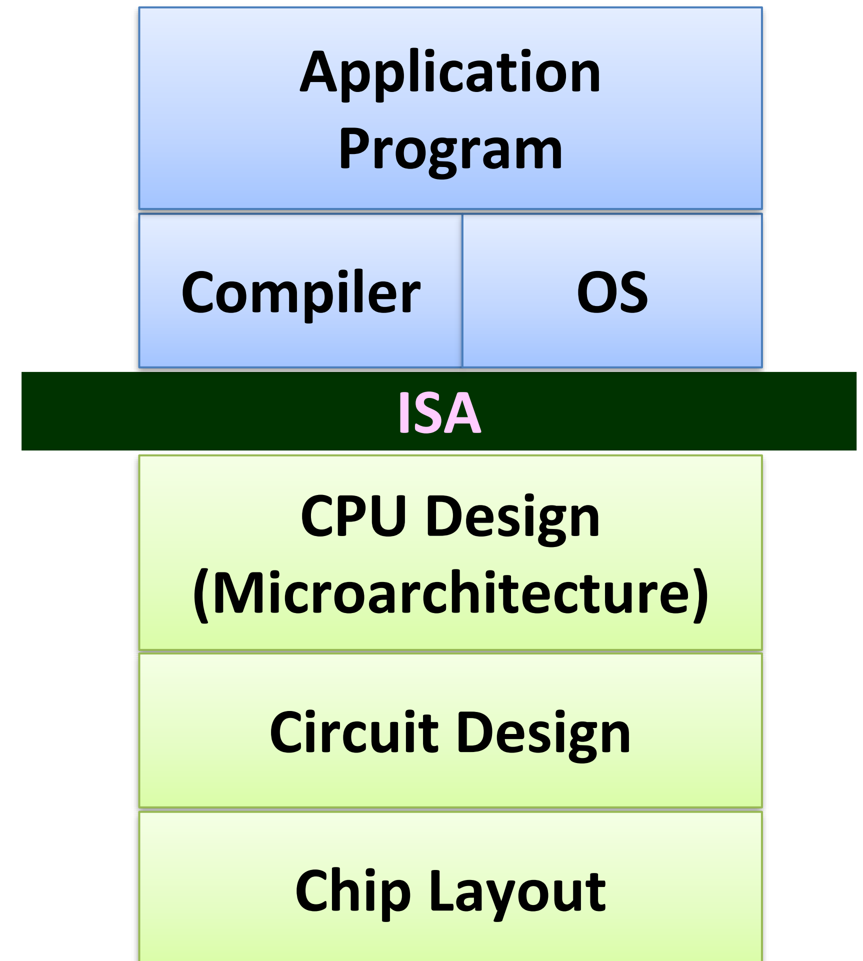
Architecture

> “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”**
>
> *-- Amdahl, Blaauw, and Brooks, Architecture of the IBM System/360, IBM Journal of Research and Development, April 1964.*

- The visible interface between software and hardware
- What the user (OS, compiler, …) needs to know to reason about how the machine behaves
- Abstracted from the details of how it may accomplish its task

Instruction Set Architecture (ISA)

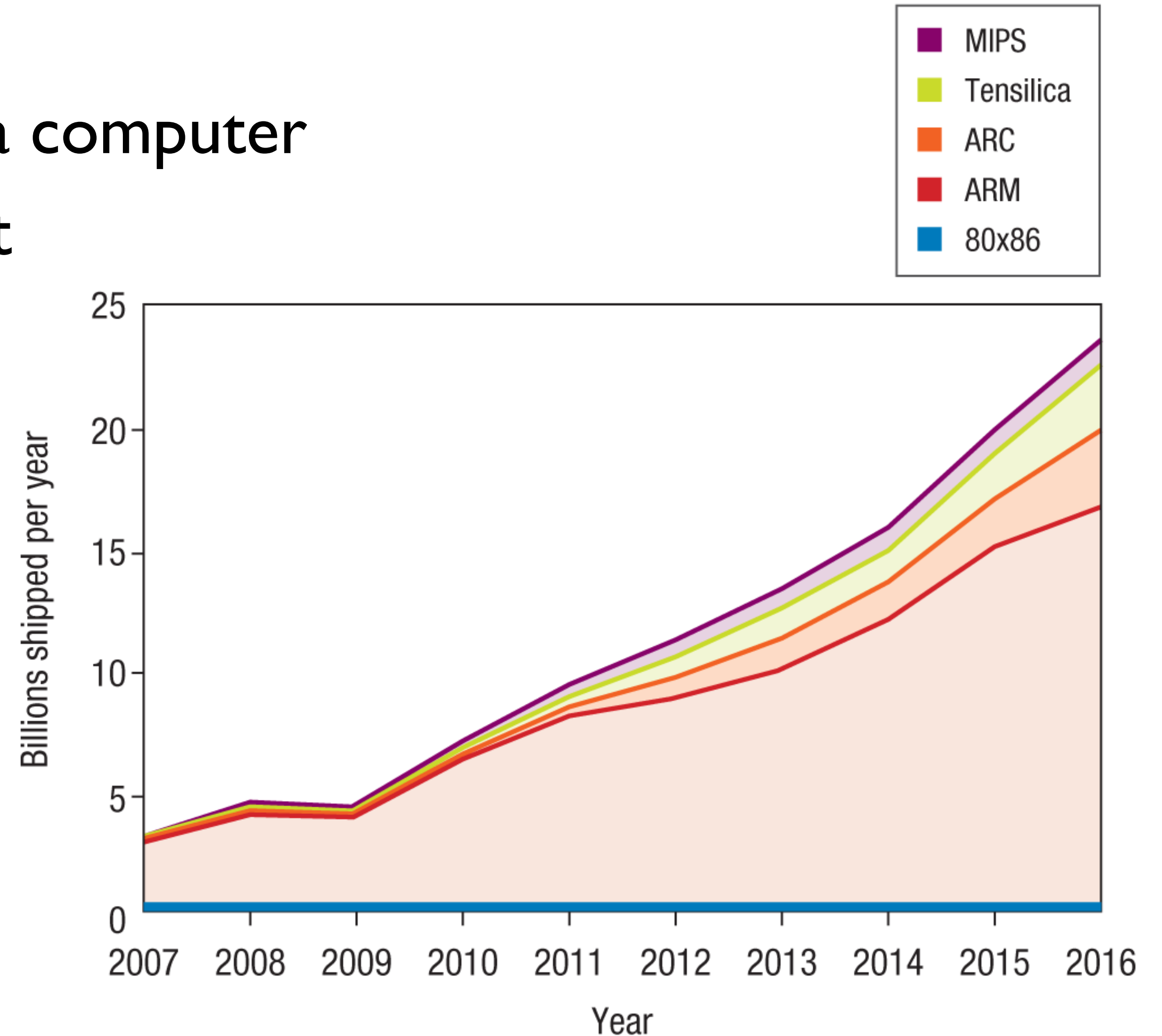
- Above: how to program machine
  - Processors execute instructions in sequence
- Below: what needs to be built
  - Use variety of tricks to make it run fast



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

Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets
  - RISC (Reduced Instruction Set Computer)



Source: David Patterson, "Reduced Instruction Set Computers Then and Now," IEEE Computer, 2017.

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 survives for long time
- Make architectural research and education more real
  - Fully open hardware and software stacks

Source: Krste Asanovic, RISC-V Tutorial at HPCA, 2015.

RISC-V ISAs

- Three base integer ISAs, one per address width
  - RV32I, RV64I, RV128I
  - RV32I: Only 40 instructions defined
  - RV32E: Reduced version of RV32I with 16 registers for embedded systems
- Standard extensions
  - Standard RISC encoding in a fixed 32-bit instruction format
  - C extension offers shorter 16-bit versions of common 32-bit RISC-V instructions (can be intermixed with 32-bit instructions)

Name	Extension
M	Integer Multiply/Divide
A	Atomic Instructions
F	Single-precision FP
D	Double-precision FP
G	General-purpose (= IMAFD)
Q	Quad-precision FP
C	Compressed Instructions

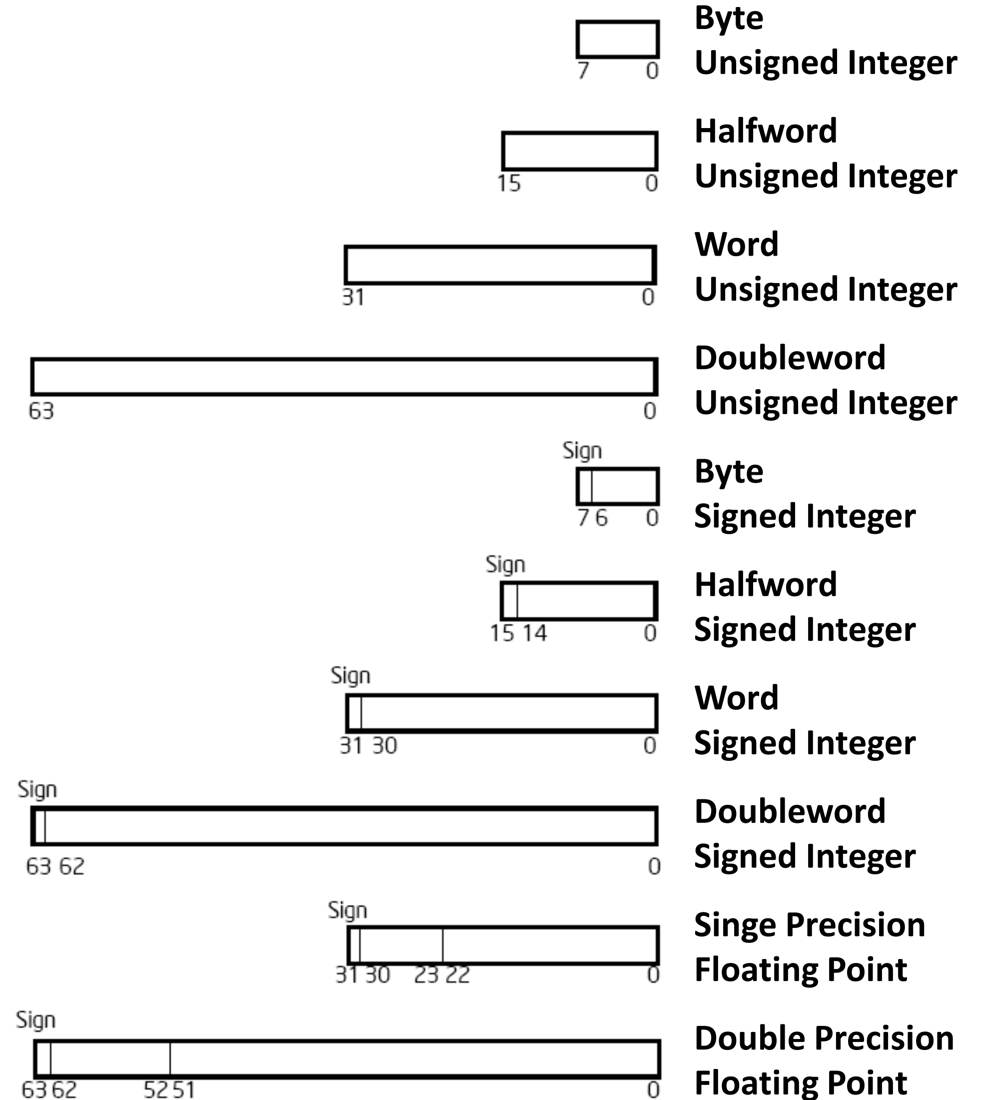
RISC-V Registers

#	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

Data Types

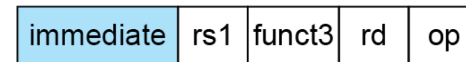
- Integer data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4 or 8 bytes (with F or D extension)
- No aggregated types such as arrays or structures
  - Just contiguously allocated bytes in memory



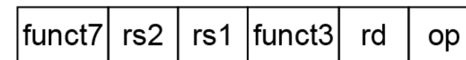
RISC-V Addressing

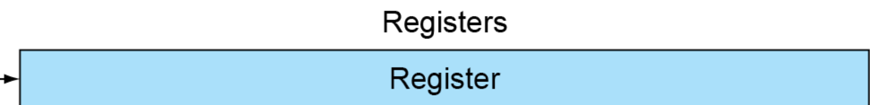
- How is the data specified?
  - Immediate value (constant)
  - Value in a register
  - Value in memory
    - Mem[rs1 + imm]
  - Address: pc + imm

1. Immediate addressing

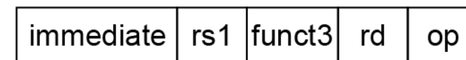


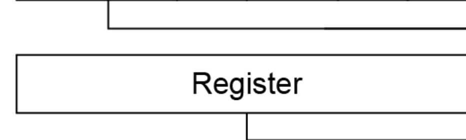
2. Register addressing

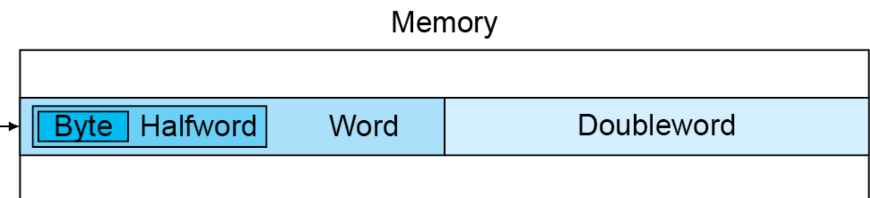




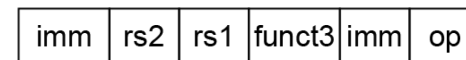
3. Base addressing

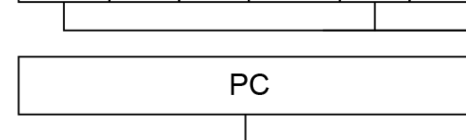


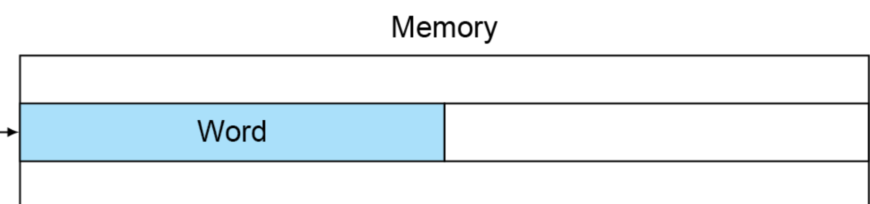




4. PC-relative addressing







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 Tutorial @ HotChips '19

- https://youtu.be/nPXdbm9lc3A

RISC-V:
Arithmetic / Logical Operations

Chap. 2.2–2.3, 2.6

Arithmetic Operations

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

```
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

- RISC-V (RV32I) has a 32 x 32-bit register file: `x0 ~ x31`
  - Use for frequently accessed data
  - 32-bit data is called a "word"

- Design Principle 2: Smaller is faster
  - cf. Main memory: millions of locations

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);
```

Compiled RISC-V code:

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

Registers vs. Memory

- Registers are faster to access than memory
- In RISC-V, data in memory cannot be directly addressed by ALU instructions
- Operating on memory data requires loads and stores
  - 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!

Immediate Operands

- Constant data specified in an instruction

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

- Make the common case fast
  - Small constants are common (limited to 12 bits)
  - Immediate operand avoids a load instruction

Logical Operations

- Instructions for bitwise manipulation

<table>
<tr><th>Operation</th><th>C</th><th>Java</th><th>RISC-V</th></tr>
<tr><td>Shift left</td><td><<</td><td><<</td><td><code>sll, slli</code></td></tr>
<tr><td>Shift right (arithmetic)</td><td>>></td><td>>></td><td><code>sra, srai</code></td></tr>
<tr><td>Shift right (logical)</td><td>>></td><td>>>></td><td><code>srl, srli</code></td></tr>
<tr><td>Bit-by-bit AND</td><td>&</td><td>&</td><td><code>and, andi</code></td></tr>
<tr><td>Bit-by-bit OR</td><td>|</td><td>|</td><td><code>or, ori</code></td></tr>
<tr><td>Bit-by-bit XOR</td><td>^</td><td>^</td><td rowspan="2"><code>xor, xori</code></td></tr>
<tr><td>Bit-by-bit NOT</td><td>~</td><td>~</td></tr>
</table>

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

AND Operations

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

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

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

OR Operation

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

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

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

XOR Operation

- Differencing operation
  - Clear when bits are the same, set if they are different

```
xor x9, x10, x12
```

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

32-bit Constants

- Most constants are small
  - 12-bit immediate is sufficient
- For the occasional 32-bit constant:
  - Copy 20-bit constant to bits [31:12] of rd
  - Clear bits [11:0] of rd to 0

```
lui   rd, constant
```

- Example: `x19 <- 0x003D0500`

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

[31:12]	[11:0]
0000 0000 0011 1101 0000	0000 0000 0000
0000 0000 0011 1101 0000	0101 0000 0000

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)

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: arith

```c
int arith (int x,
           int y,
           int z) {
    int t1 = x + y;
    int t2 = z + t1;
    int t3 = x + 4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int 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

```c
int logical (int x,
             int y) {
    int t1 = x ^ y;
    int t2 = t1 >> 17;
    int mask = (1 << 8) - 7;
    int 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

Chap. 2.2–2.3

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory to registers
  - Store result from register to memory
- Memory is byte addressed: each address identifies an 8-bit byte
- RISC-V is Little Endian
  - Least-significant byte at least address of a word
- RISC-V does not require words to be aligned in memory
  - Unlike some other ISAs

Memory Operand Example

C code:

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

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

Compiled RISC-V code:

```
// 4 bytes per word
// &A[8] = A + 32

lw   x9, 32(x22)
add  x9, x21, x9
sw   x9, 48(x22)
```

Byte/Halfword/Word Operations

- Load byte/halfword/word:
  Sign extend to 32 bits in rd

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

- Load byte/halfword unsigned:
  Zero extend to 32 bits in rd

```
lbu   rd, offset(rs1)
lhu   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 word	**I**	`lw rd, imm12(rs1)`	R[rd] = 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 byte unsigned	**I**	`lbu rd, imm12(rs1)`	R[rd] = ZeroExt(Mem_1[R[rs1] + SignExt(imm12)])
Store word	**S**	`sw rs2, imm12(rs1)`	Mem_4[R[rs1] + SignExt(imm12)] = R[rs2]
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)

Swap Example

- Source code in C:

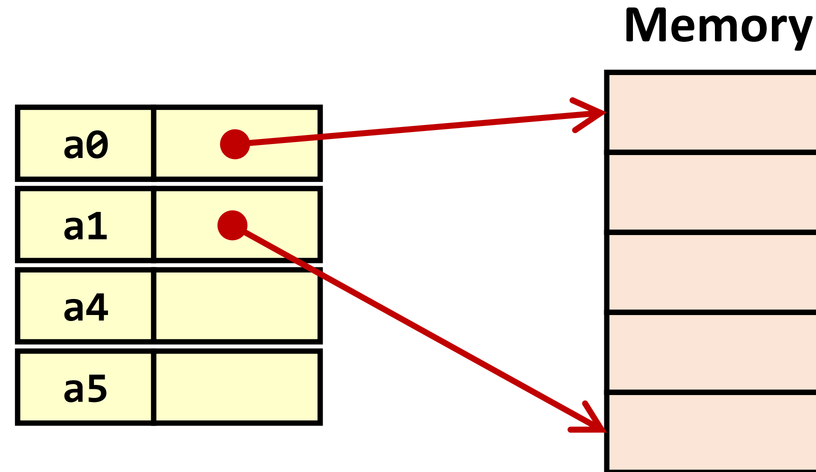
```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

- Corresponding assembly code:

```
swap:
    lw    a4, 0(a0)
    lw    a5, 0(a1)
    sw    a5, 0(a0)
    sw    a4, 0(a1)
    ret
```

Understanding Swap (1)

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



Register Allocation (By compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

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

Understanding Swap (2)

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

Register	Value
a0	0x110
a1	0x100
a4	
a5	

Memory

Address	Value
0x110	123
0x10C	
0x108	
0x104	
0x100	456

Register Allocation (By compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

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

Understanding Swap (3)

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

Register	Value
a0	0x110
a1	0x100
a4	123
a5	

Memory

Address	Value
0x110	123
0x10C	
0x108	
0x104	
0x100	456

Register Allocation (By compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

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

Understanding Swap (4)

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

Register	Value
a0	0x110
a1	0x100
a4	123
a5	456

Memory

Address	Value
0x110	123
0x10C	
0x108	
0x104	
0x100	456

Register Allocation (By compiler)

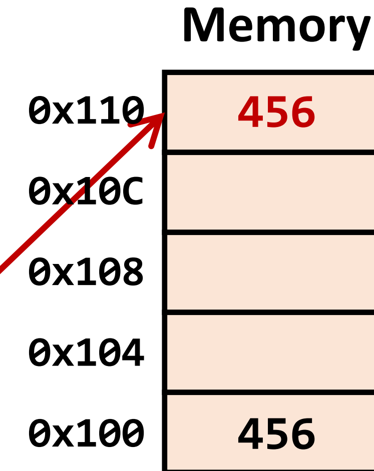
Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

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

Understanding Swap (5)

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

a0	0x110
a1	0x100
a4	123
a5	456



Register Allocation (By compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

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

Understanding Swap (6)

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

a0	0x110
a1	0x100
a4	123
a5	456

Memory

Address	Value
0x110	456
0x10C	
0x108	
0x104	
0x100	123

Register Allocation (By compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

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

String Copy Example

```c
void strcpy(char x[],
            char y[])
{
  int i;

  i = 0;
  while
    ((x[i]=y[i])!='\0')
    i += 1;
}
```

```
strcpy:
      ; x is in a0
      ; y is in a1
      ; i is allocated in t3

      add    t3,zero,zero   ; t3 <- 0
L1:   add    t0,t3,a1       ; t0 <- &y[i]
      lbu    t1,0(t0)       ; t1 <- y[i]
      add    t2,t3,a0       ; t2 <- &x[i]
      sb     t1,0(t2)       ; x[i] <- y[i]
      beq    t1,zero,L2     ; if (t1==0), goto L2
      addi   t3,t3,1        ; t3 += 1
      j      L1             ; goto L1
L2:   ret                   ; return
```