JEONBUK NATIONAL UNIVERSITY

컴퓨터 구조

2. Instructions: Language of the Computer

전북대학교
컴퓨터인공지능학부
김찬기

Programming

code/intro/hello.c

```
1  #include <stdio.h>
2
3  int main()
4  {
5      printf("hello, world\n");
6      return 0;
7  }
```

code/intro/hello.c

Figure 1.1 **The `hello` program.** (Source: [60])

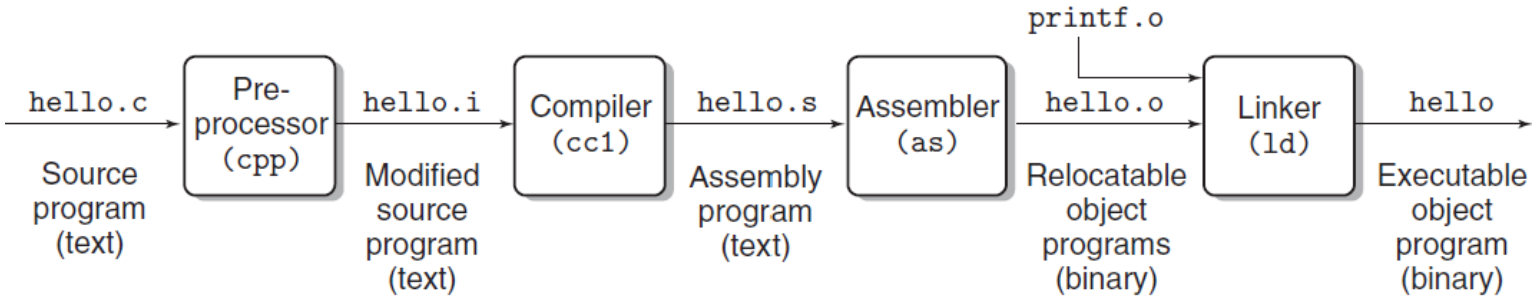


Figure 1.3 **The compilation system.**

```
linux> gcc -o hello hello.c
```

Programming

- *Preprocessing phase.* The preprocessor (cpp) modifies the original C program according to directives that begin with the '#' character.
  - For example, the #include <stdio.h> command in line 1 of hello.c tells the preprocessor to read the contents of the system header file stdio.h and insert it directly into the program text. The result is another C program, typically with the .i suffix.
- *Compilation phase.* The compiler (cc1) translates the text file hello.i into the text file hello.s, which contains an ***assembly-language program***. This program includes the following definition of function

- main:

```
1 main:
2 subq $8, %rsp
3 movl $.LC0, %edi
4 call puts
5 movl $0, %eax
6 addq $8, %rsp
7 ret
```

- Each of lines in this definition describes **one low-level machine language instruction** in a textual form.

Assembly language is useful because it provides a common output language for different compilers for different high-level languages.

- For example, C compilers and Fortran compilers both generate output files in the same assembly language.

Programming

- *Assembly phase.* Next, the assembler (as) translates hello.s into machinelanguage instructions, packages them in a form known as a *relocatable object program*, and stores the result in the object file hello.o.

- This file is a binary file containing 17 bytes to encode the instructions for function main. If we were to view hello.o with a text editor, it would appear to be gibberish.

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

Instruction set

RISC-V operands

Name	Example	Comments
32 registers	x0-x31	Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0.
2^{61} memory words	Memory[0], Memory[8], ..., Memory[18,446,744,073,709,551,608]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential doubleword accesses differ by 8. Memory holds data structures, arrays, and spilled registers.

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	add x5, x6, x7	x5 = x6 + x7	Three register operands; add
	Subtract	sub x5, x6, x7	x5 = x6 - x7	Three register operands; subtract
	Add immediate	addi x5, x6, 20	x5 = x6 + 20	Used to add constants
Data transfer	Load doubleword	ld x5, 40(x6)	x5 = Memory[x6 + 40]	Doubleword from memory to register
	Store doubleword	sd x5, 40(x6)	Memory[x6 + 40] = x5	Doubleword from register to memory
	Load word	lw x5, 40(x6)	x5 = Memory[x6 + 40]	Word from memory to register
	Load word, unsigned	lwu x5, 40(x6)	x5 = Memory[x6 + 40]	Unsigned word from memory to register
	Store word	sw x5, 40(x6)	Memory[x6 + 40] = x5	Word from register to memory
	Load halfword	lh x5, 40(x6)	x5 = Memory[x6 + 40]	Halfword from memory to register
	Load halfword, unsigned	lhu x5, 40(x6)	x5 = Memory[x6 + 40]	Unsigned halfword from memory to register
	Store halfword	sh x5, 40(x6)	Memory[x6 + 40] = x5	Halfword from register to memory
	Load byte	lb x5, 40(x6)	x5 = Memory[x6 + 40]	Byte from memory to register
	Load byte, unsigned	lbu x5, 40(x6)	x5 = Memory[x6 + 40]	Byte unsigned from memory to register
	Store byte	sb x5, 40(x6)	Memory[x6 + 40] = x5	Byte from register to memory
	Load reserved	lr.d x5, (x6)	x5 = Memory[x6]	Load; 1st half of atomic swap
	Store conditional	sc.d x7, x5, (x6)	Memory[x6] = x5; x7 = 0/1	Store; 2nd half of atomic swap
	Load upper immediate	lui x5, 0x12345	x5 = 0x12345000	Loads 20-bit constant shifted left 12 bits
Logical	And	and x5, x6, x7	x5 = x6 & x7	Three reg. operands; bit-by-bit AND
	Inclusive or	or x5, x6, x8	x5 = x6 | x8	Three reg. operands; bit-by-bit OR
	Exclusive or	xor x5, x6, x9	x5 = x6 ^ x9	Three reg. operands; bit-by-bit XOR
	And immediate	andi x5, x6, 20	x5 = x6 & 20	Bit-by-bit AND reg. with constant
	Inclusive or immediate	ori x5, x6, 20	x5 = x6 | 20	Bit-by-bit OR reg. with constant
	Exclusive or immediate	xori x5, x6, 20	x5 = x6 ^ 20	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	sll x5, x6, x7	x5 = x6 << x7	Shift left by register
	Shift right logical	srl x5, x6, x7	x5 = x6 >> x7	Shift right by register
	Shift right arithmetic	sra x5, x6, x7	x5 = x6 >> x7	Arithmetic shift right by register
	Shift left logical immediate	slli x5, x6, 3	x5 = x6 << 3	Shift left by immediate
	Shift right logical immediate	srli x5, x6, 3	x5 = x6 >> 3	Shift right by immediate
	Shift right arithmetic immediate	srai x5, x6, 3	x5 = x6 >> 3	Arithmetic shift right by immediate
Conditional branch	Branch if equal	beq x5, x6, 100	if (x5 == x6) go to PC+100	PC-relative branch if registers equal
	Branch if not equal	bne x5, x6, 100	if (x5 != x6) go to PC+100	PC-relative branch if registers not equal
	Branch if less than	blt x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less
	Branch if greater or equal	bge x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal
	Branch if less, unsigned	bltu x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less, unsigned
	Branch if greater or equal, unsigned	bgeu x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal, unsigned
Unconditional branch	Jump and link	jal x1, 100	x1 = PC+4; go to PC+100	PC-relative procedure call
	Jump and link register	jalr x1, 100(x5)	x1 = PC+4; go to x5+100	Procedure return; indirect call

FIGURE 2.1 RISC-V assembly language revealed in this chapter.
This information is also found in Column 1 of the RISC-V Reference Data Card at the front of this book.

The RISC-V Instruction Set

- Used as the example throughout the book
- Developed at UC Berkeley as open ISA
  - Now managed by the RISC-V Foundation (riscv.org)
- Typical of many modern ISAs
  - See RISC-V Reference Data tear-out card
- Similar ISAs have a large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …

Instruction set

- Add and subtract, three operands
- **Two(or more?) sources** and **one destination**

add a, b, c // a gets b + c

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favours regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Instruction set

- C code:

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

- Compiled RISC-V code:

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

Recall: Our Computer

- **CPU**
- **(Main) memory**
- I/O

현재의 관심사(instruction 관련)

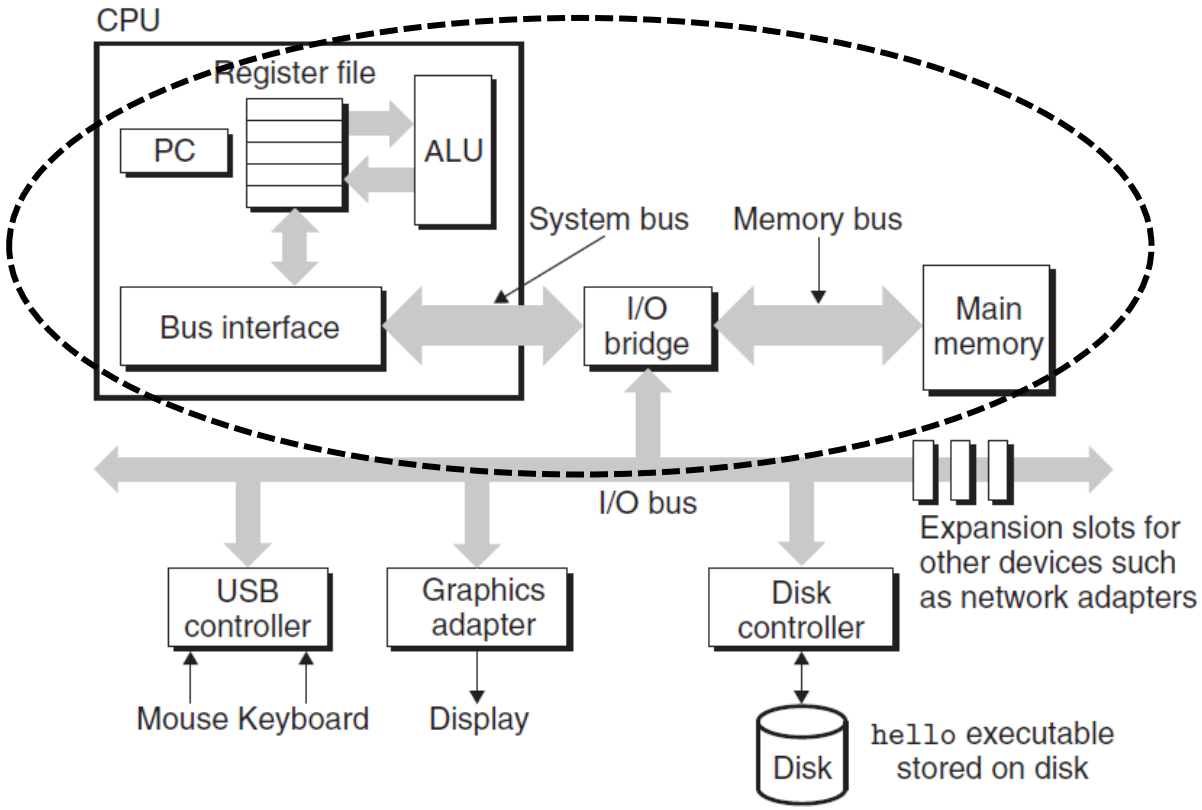


Figure 1.4
Hardware organization of a typical system. CPU: central processing unit, ALU: arithmetic/logic unit, PC: program counter, USB: Universal Serial Bus.

Recall: Our Computer

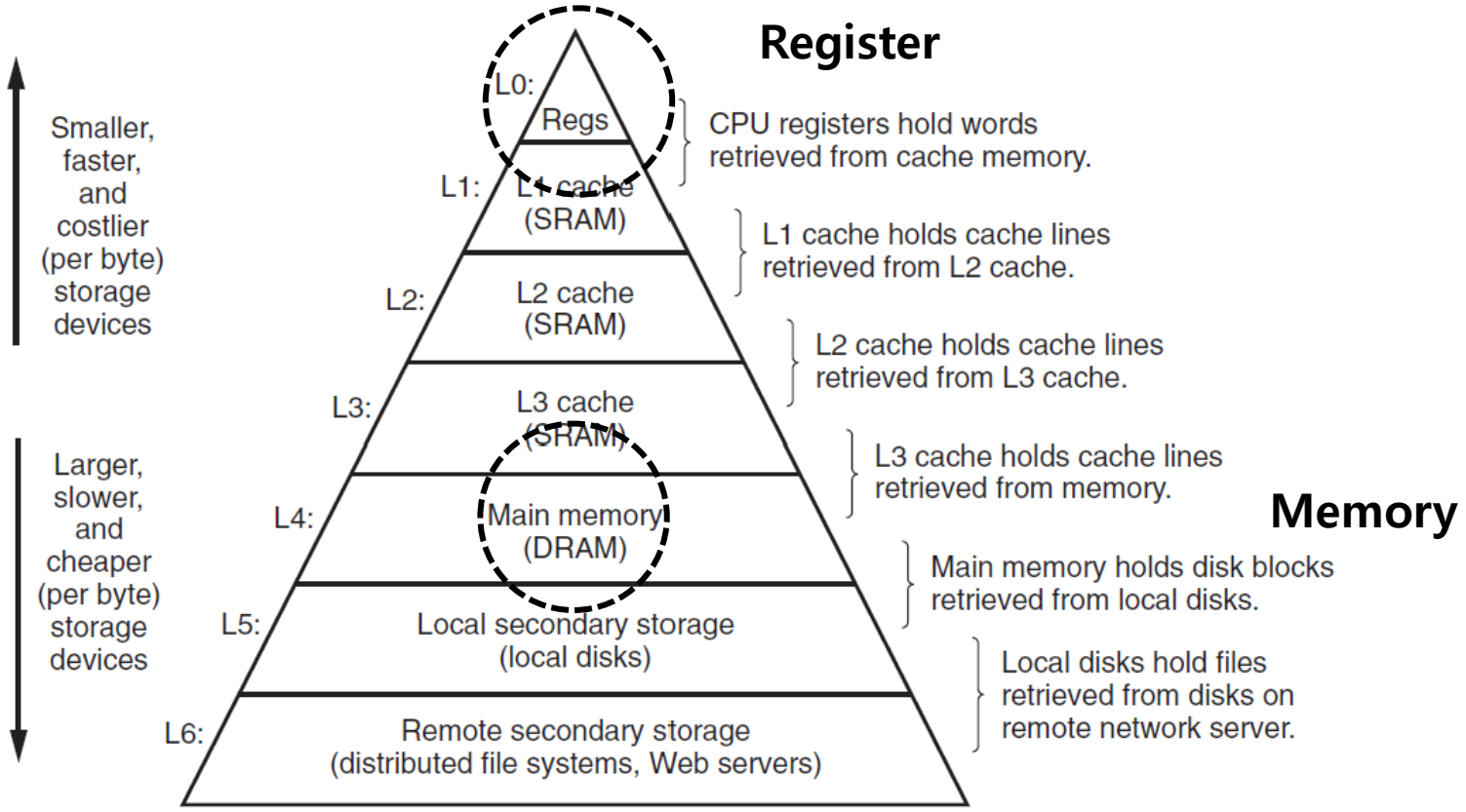


Figure 1.9 **An example of a memory hierarchy.**

Register Operands

- Arithmetic instructions use register operands

- RISC-V has a 32 × 64-bit register file
  - Use for frequently accessed data
  - 64-bit data is called a "doubleword"
  - 32 x 64-bit general purpose registers x0 to x30
  - 32-bit data is called a "word"

- *Design Principle 2:* Smaller is faster(Registers are faster)
- c.f. main memory: millions of locations

Register Operands (Example)

- x0: the constant value 0
- x1: return address
- x2: stack pointer
- x3: global pointer
- x4: thread pointer
- x5 – x7, x28 – x31: temporaries
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments

Register Operands (Example)

Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

FIGURE 2.14 RISC-V register conventions. This information is also found in Column 2 of the RISC-V Reference Data Card at the front of this book.

Register Operands Example

- C code:

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

f, …, j in x19, x20, …, x23

- Compiled RISC-V code:

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

Memory Operands

- Main memory used **for composite data**
  - Arrays, structures, dynamic data
- To apply **arithmetic operations**
  - Load values from memory into 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
  - c.f. Big Endian: most-significant byte at least address
- RISC-V does not require words to be aligned in memory
  - Unlike some other ISAs

Memory Operand Example

- C code:
  - A[12] = h + A[8];
  - h in x21, base address of A in x22
- Compiled RISC-V code:
  - Index 8 requires offset of 64
    - 8 bytes per doubleword

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

Registers vs. Memory

- Registers are faster to access than memory
- 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
  - Immediate operand avoids a load instruction

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
- Example
  - 0000 0000 … 0000 10112
    = 0 + … + 1×23 + 0×22 +1×21 +1×20
    = 0 + … + 8 + 0 + 2 + 1 = 1110
- Using 64 bits: 0 to +18,446,774,073,709,551,615

2s-Complement Signed Integers

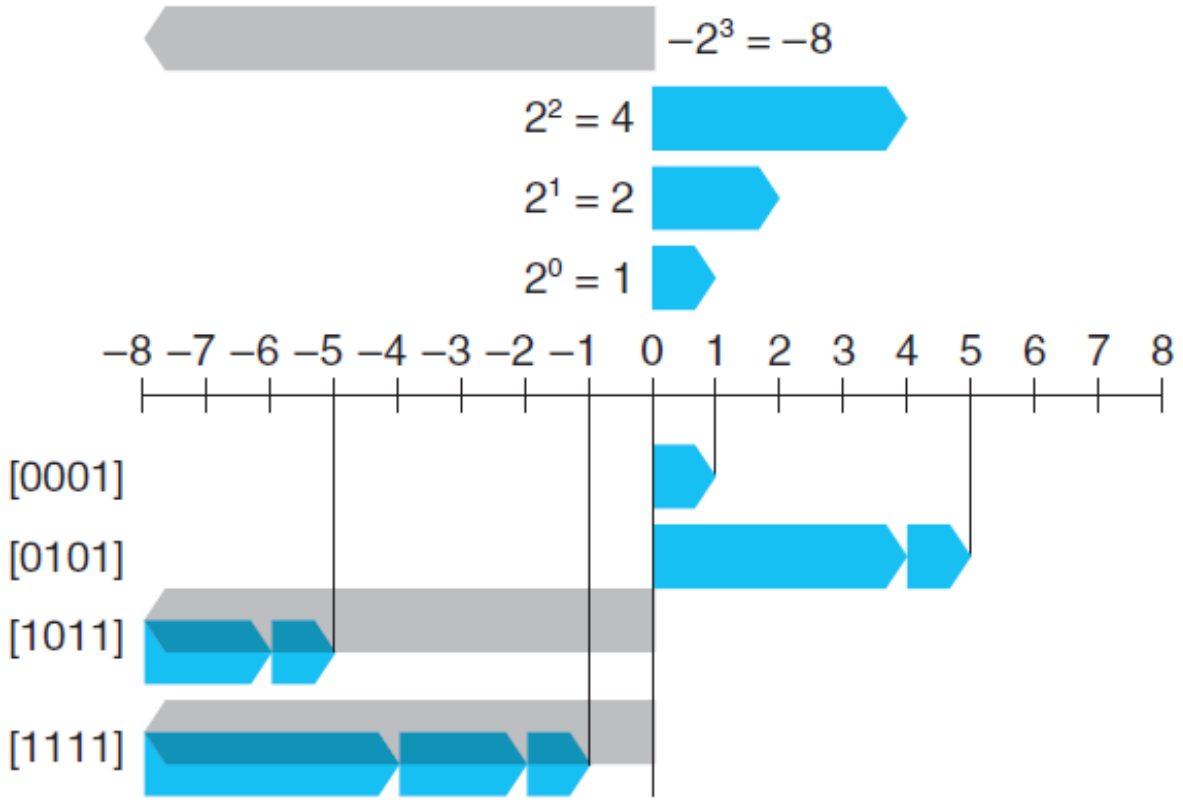
- 2s-Complement Signed Integers

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1}-1$
- Example
  - 1111 1111 … 1111 1100(2)
    $= -1 \times 2^{31} + 1 \times 2^{30} + \ldots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
    = –2,147,483,648 + 2,147,483,644 = –410
- Using 64 bits: −9,223,372,036,854,775,808
  to 9,223,372,036,854,775,807

2s-Complement Signed Integers

Figure 2.13
Two's-complement number examples for $w = 4$. Bit 3 serves as a sign bit; when set to 1, it contributes $-2^3 = -8$ to the value. This weighting is shown as a leftward-pointing gray bar.



2s-Complement Signed Integers

- Bit 63 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

Signed Negation

- Complement and add 1
  - Complement means 1 → 0, 0 → 1

$$x + \bar{x} = 1111\ldots111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate +2
  - +2 = 0000 0000 … 0010_{two}
  - –2 = 1111 1111 … 1101_{two} + 1
    = 1111 1111 … 1110_{two}

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110
- In RISC-V instruction set
  - lb: sign-extend loaded byte
  - lbu: zero-extend loaded byte

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- RISC-V instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!

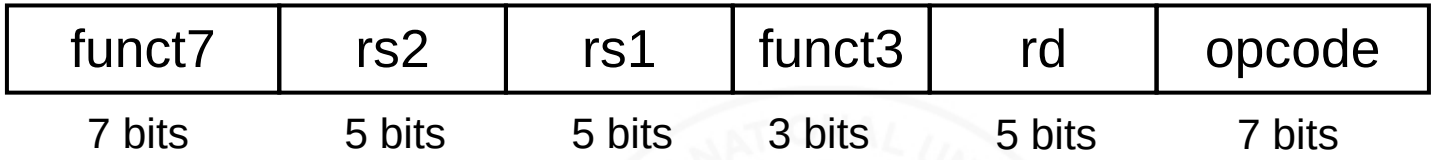
Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

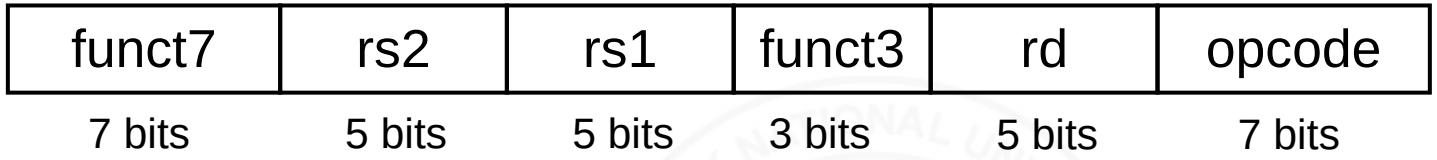
- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

RISC-V R-format Instructions

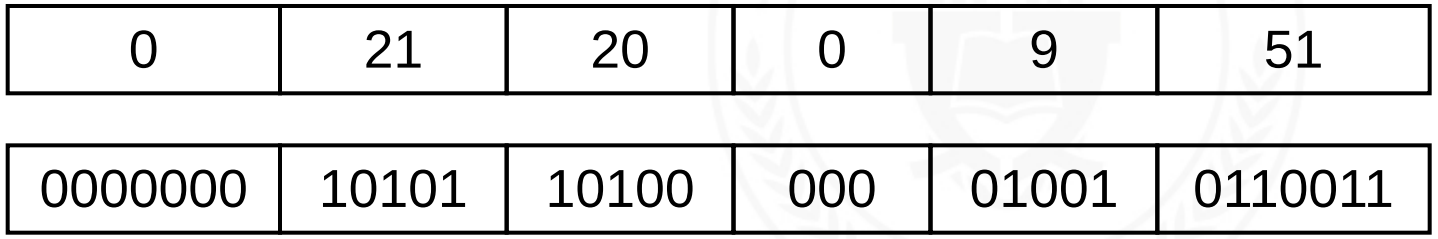


- Instruction fields
  - opcode: operation code
  - rd: destination register number
  - funct3: 3-bit function code (additional opcode)
  - rs1: the first source register number
  - rs2: the second source register number
  - funct7: 7-bit function code (additional opcode)

R-format Example

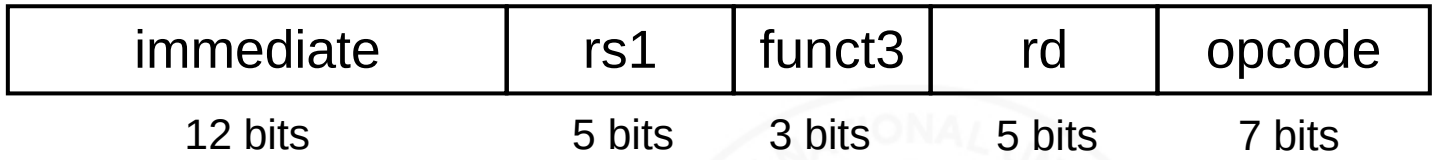



```
add x9,x20,x21
```



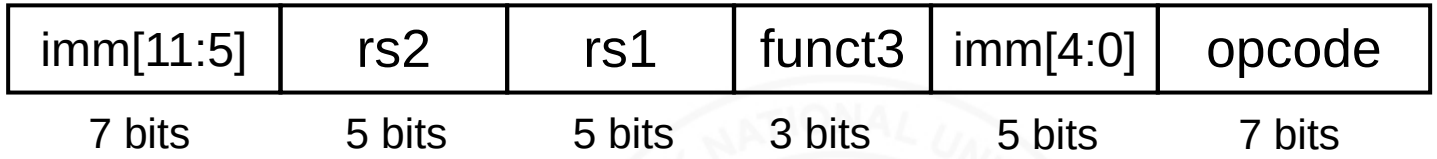
$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_{two} = 015A04B3_{16}$

RISC-V I-format Instructions



- Immediate arithmetic and load instructions
  - rs1: source or base address register number
  - immediate: constant operand, or offset added to base address
    - 2s-complement, sign extended
- Design Principle 3: Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible.

RISC-V S-format Instructions



- Different immediate format for store instructions
  - rs1: base address register number
  - rs2: source operand register number
  - immediate: offset added to base address
    - Split so that rs1 and rs2 fields always in the same place

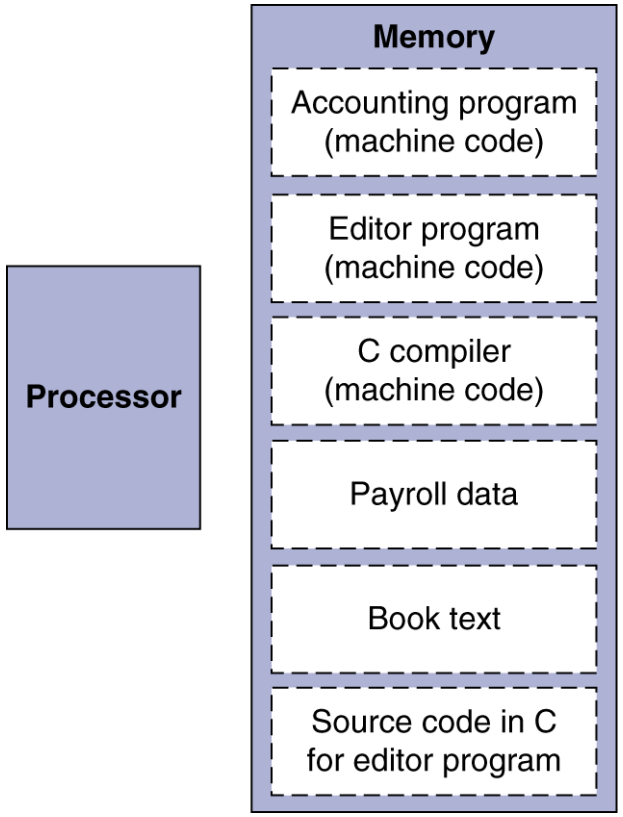
RISC-V S-format Instructions

R-type Instructions	funct7	rs2	rs1	funct3	rd	opcode	Example
add (add)	0000000	00011	00010	000	00001	0110011	add x1, x2, x3
sub (sub)	0100000	00011	00010	000	00001	0110011	sub x1, x2, x3
I-type Instructions	**immediate**		**rs1**	**funct3**	**rd**	**opcode**	**Example**
addi (add immediate)	001111101000		00010	000	00001	0010011	addi x1, x2, 1000
lw (load word)	001111101000		00010	010	00001	0000011	lw x1, 1000 (x2)
S-type Instructions	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**	**Example**
sw (store word)	0011111	00001	00010	010	01000	0100011	sw x1, 1000(x2)

FIGURE 2.6 RISC-V architecture revealed through Section 2.5. The three RISC-V instruction formats so far are R, I, and S. The R-type format has two source register operand and one destination register operand. The I-type format replaces one source register operand with a 12-bit *immediate* field. The S-type format has two source operands and a 12-bit immediate field, but no destination register operand. The S-type immediate field is split into two parts, with bits 11–5 in the leftmost field and bits 4–0 in the second-rightmost field.

Stored Program Computers

The BIG Picture



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

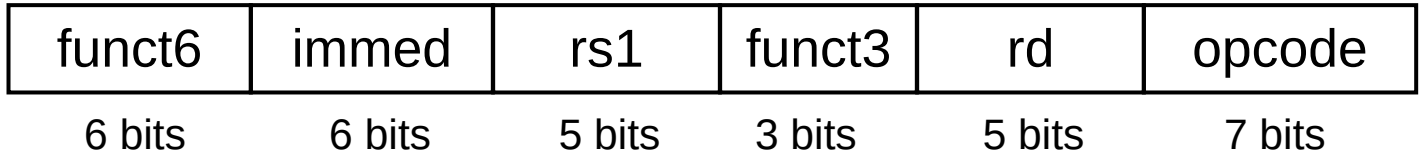
Logical operation

- Instructions for bitwise manipulation

Operation	C	Java	RISC-V
Shift left	<<	<<	`slli`
Shift right	>>	>>>	`srli`
Bit-by-bit AND	&	&	`and, andi`
Bit-by-bit OR	|	|	`or, ori`
Bit-by-bit XOR	^	^	`xor, xori`
Bit-by-bit NOT	~	~	

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

Shift Operations

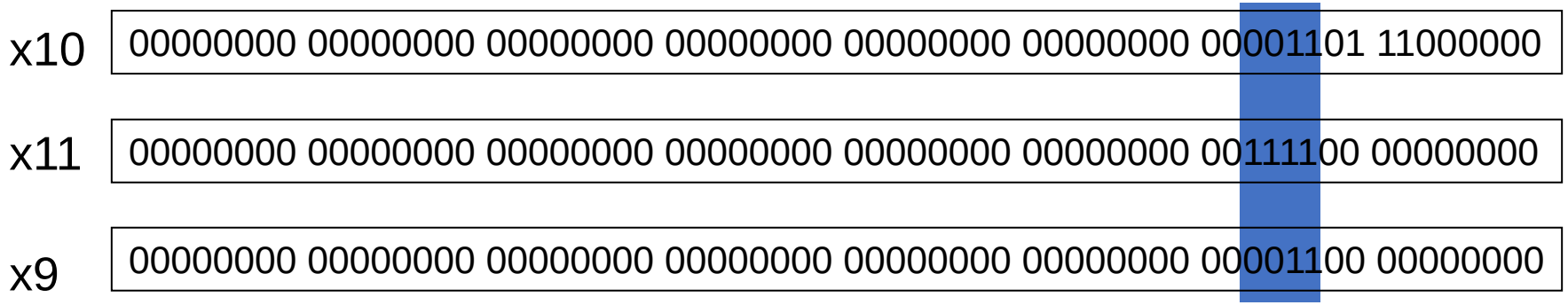


- immed: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - slli by i bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - srli by i bits divides by 2^i (unsigned only)

AND Operations

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

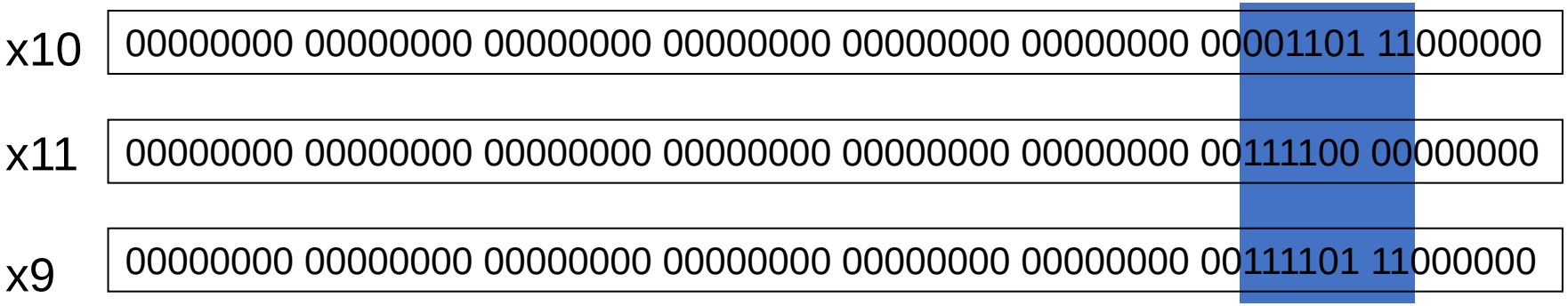
and x9,x10,x11



OR Operations

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

or x9,x10,x11



Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially

- beq rs1, rs2, L1
  - if (rs1 == rs2) branch to instruction labeled L1

- bne rs1, rs2, L1
  - if (rs1 != rs2) branch to instruction labeled L1

Compiling If Statements

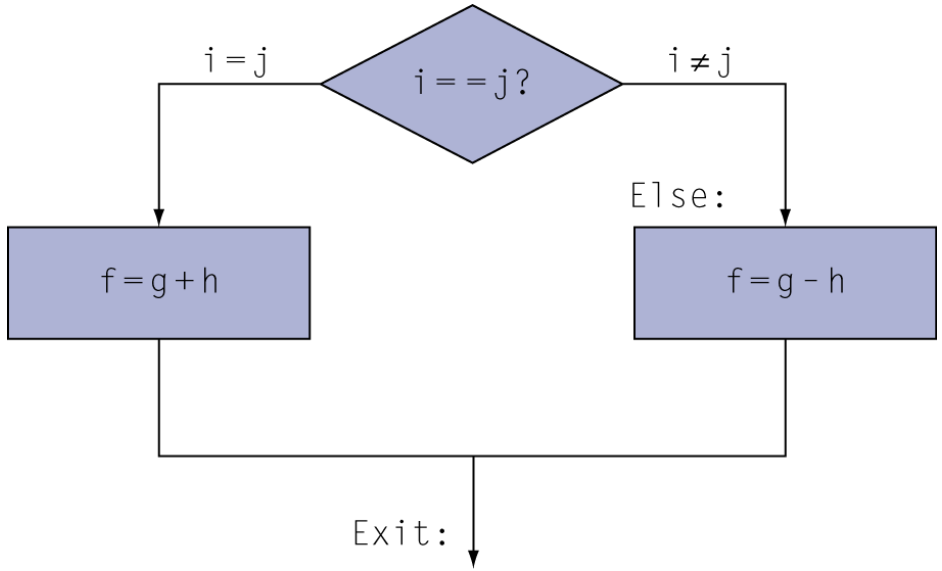
- C code:

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

  - f, g, … in x19, x20, …

- Compiled RISC-V code:

```
    bne x22, x23, Else
     add x19, x20, x21
     beq x0,x0,Exit // unconditional
Else: sub x19, x20, x21
Exit: …
```



Compiling Loop Statements

- C code:

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

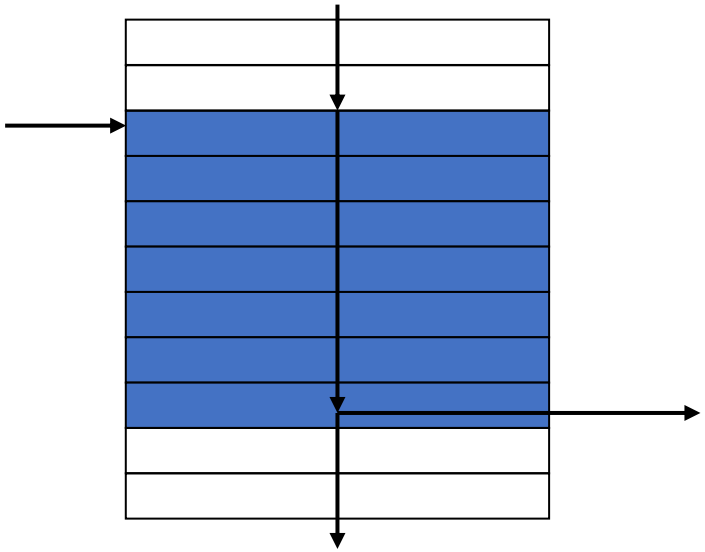
i in x22, k in x24, address of save in x25

- Compiled RISC-V code:

```
Loop: slli x10, x22, 3
      add  x10, x10, x25
      ld   x9, 0(x10)
      bne  x9, x24, Exit
      addi x22, x22, 1
      beq  x0, x0, Loop
Exit: …
```

Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



- A compiler identifies basic bl ocks for optimization
- An advanced processor can a ccelerate execution of basic blocks

Basic Blocks

- blt rs1, rs2, L1
  - if (rs1 < rs2) branch to instruction labeled L1
- bge rs1, rs2, L1
  - if (rs1 >= rs2) branch to instruction labeled L1
- Example
  - if (a > b) a += 1;
  - a in x22, b in x23
    - bge x23, x22, Exit // branch if b >= a
    - addi x22, x22, 1
- Exit:

Signed vs Unsigned

- Signed comparison: blt, bge
- Unsigned comparison: bltu, bgeu
- Example
  - x22 = 1111 1111 1111 1111 1111 1111 1111 1111
  - x23 = 0000 0000 0000 0000 0000 0000 0000 0001
  - x22 < x23 // signed
    - –1 < +1
  - x22 > x23 // unsigned
    - +4,294,967,295 > +1

Procedure Calling

- Steps required
  1. Place parameters in registers x10 to x17
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure's operations
  5. Place result in register for caller
  6. Return to place of call (address in x1)

Procedure Call Instructions

- Procedure call: jump and link
- jal x1, ProcedureLabel
  - Address of following instruction put in x1
  - Jumps to target address
  - Procedure return: jump and link register
- jalr x0, 0(x1)
  - Like jal, but jumps to 0 + address in x1
  - Use x0 as rd (x0 cannot be changed)
  - Can also be used for computed jumps
  - e.g., for case/switch statements

Procedure Call Instructions

- C code:

```
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;
}
```

  - Arguments g, …, j in x10, …, x13
  - f in x20
  - temporaries x5, x6
  - Need to save x5, x6, x20 on stack

Leaf Procedure Example

- RISC-V code(leaf_example):

```
addi sp,sp,-24
sd   x5,16(sp)
sd   x6,8(sp)
sd   x20,0(sp)
add  x5,x10,x11
add  x6,x12,x1
sub  x20,x5,x6
addi x10,x20,0
ld   x20,0(sp)
ld   x6,8(sp)
ld   x5,16(sp)
addi sp,sp,24
jalr x0,0(x1)
```

Save x5, x6, x20 on stack

x5 = g + h

x6 = i + j

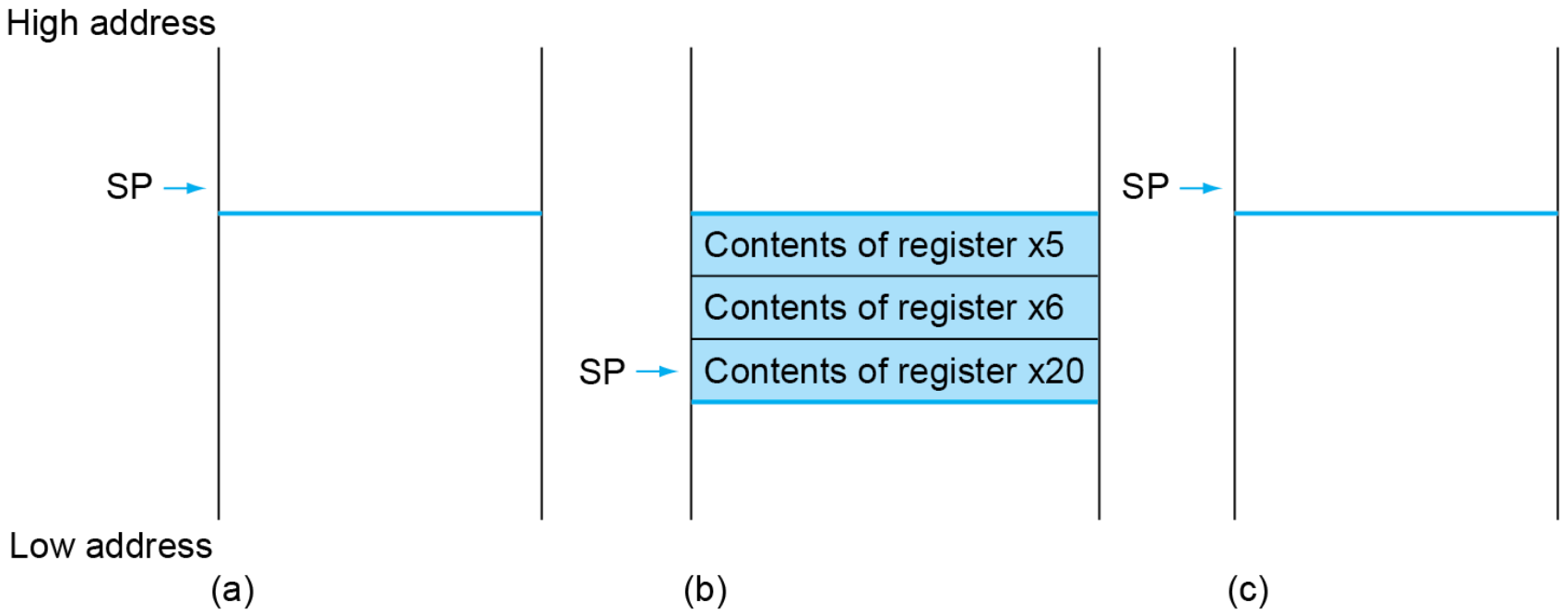
f = x5 – x6

copy f to return register

Resore x5, x6, x20 from stack

Return to caller

Local Data on the Stack



Register Usage

- x5 – x7, x28 – x31: temporary registers
  - Not preserved by the callee
- x8 – x9, x18 – x27: saved registers
  - If used, the callee saves and restores them

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

FIGURE 2.11 What is and what is not preserved across a procedure call. If the software relies on the global pointer register, discussed in the following subsections, it is also preserved.

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure: Example

- C code:

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

- Argument n in x10
- Result in x10

Non-Leaf Procedure: Example

- RISC-V code:

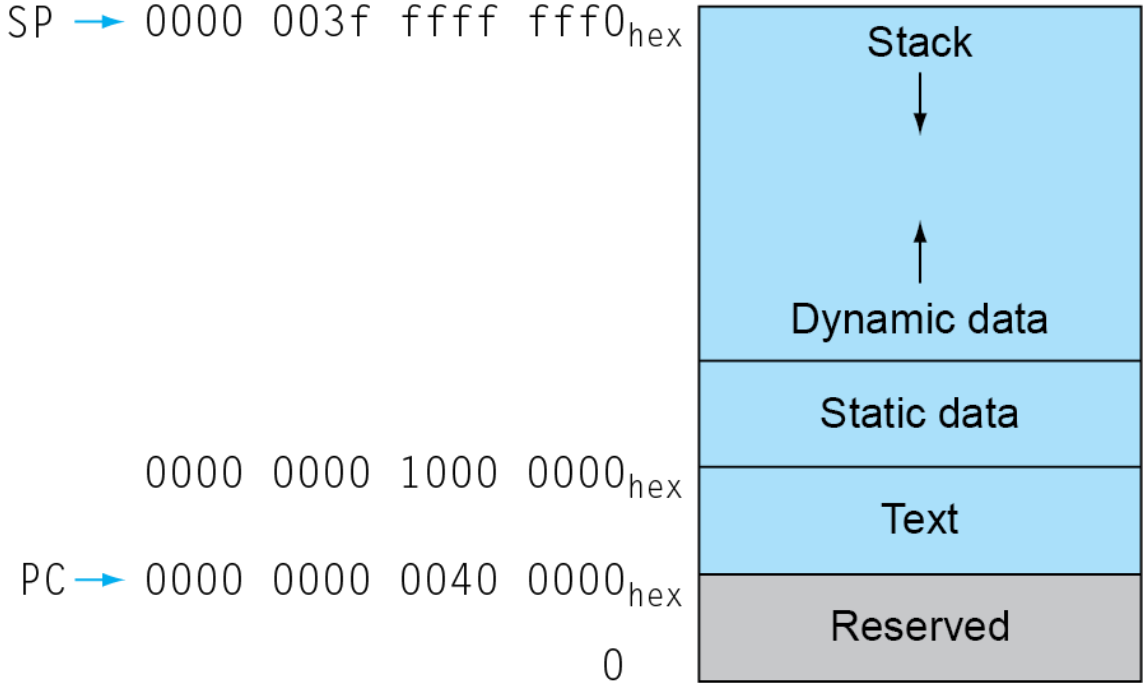
factorial:

```
addi sp,sp,-16      Save return address and n on stack
sd   x1,8(sp)
sd   x10,0(sp)
addi x5,x10,-1      x5 = n - 1
bge  x5,x0,L1       if n >= 1, go to L1
addi x10,x0,1       Pop stack, don't bother restoring values
addi sp,sp,16
jalr x0,0(x1)       Else, set return value to 1
```

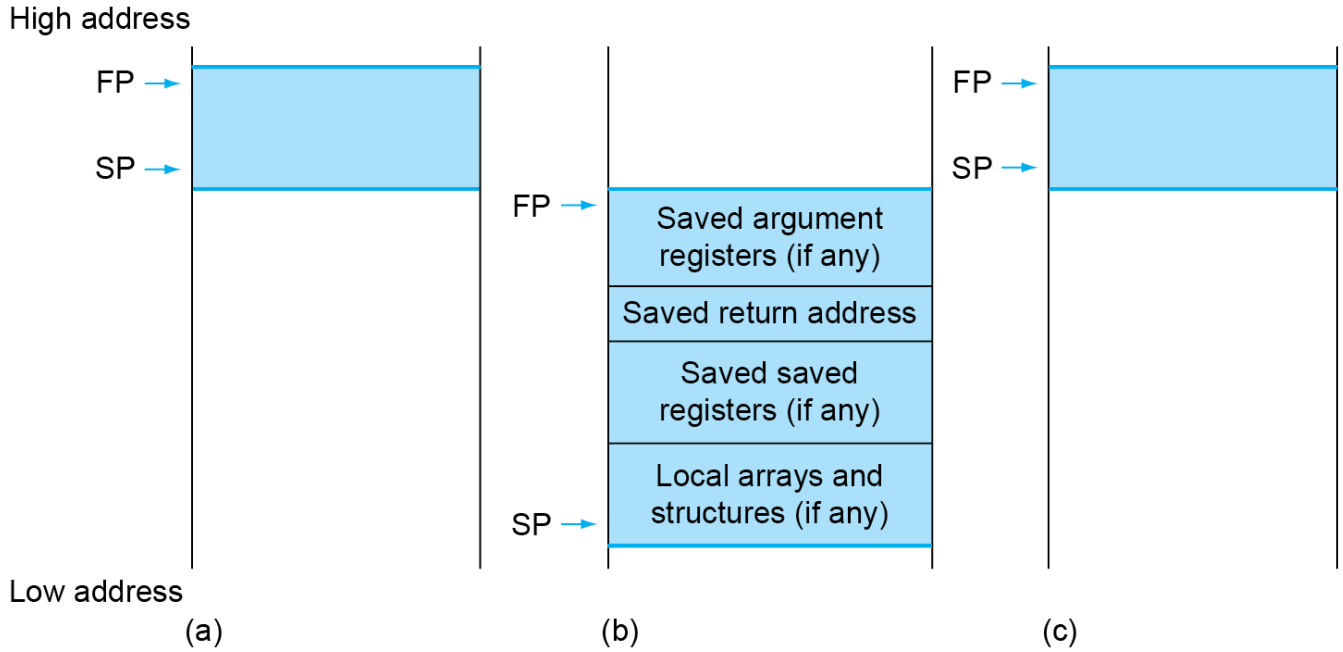
```
L1: addi x10,x10,-1     n = n - 1
    jal  x1,fact        call fact(n-1)
    addi x6,x10,0       move result of fact(n - 1) to x6
    ld   x10,0(sp)      Restore caller's n
    ld   x1,8(sp)       Restore caller's return address
    addi sp,sp,16       Pop stack
    mul  x10,x10,x6     return n * fact(n-1)
    jalr x0,0(x1)       return
```

Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - x3 (global pointer) initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



Memory Layout



- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

Byte/Halfword/Word Operations

- RISC-V byte/halfword/word load/store
  - 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 unsigned: Zero extend to 64 bits 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)

String Copy Example

- C code:
  - Null-terminated string

```
void strcpy (char x[], char y[])
 { size_t i;
 i = 0;
 while ((x[i]=y[i])!='₩0')
   i += 1;
 }
```

String Copy Example

- RISC-V code:

```
strcpy:
        addi sp,sp,-8          // adjust stack for 1 doubleword
        sd   x19,0(sp)       // push x19
        add  x19,x0,x0         // i=0
   L1: add  x5,x19,x10         // x5 = addr of y[i]
        lbu  x6,0(x5)          // x6 = y[i]
        add  x7,x19,x10        // x7 = addr of x[i]
        sb   x6,0(x7)          // x[i] = y[i]
        beq  x6,x0,L2          // if y[i] == 0 then exit
        addi x19,x19,1         // i = i + 1
        jal  x0,L1             // next iteration of loop
   L2: ld   x19,0(sp) // restore saved x19
        addi sp,sp,8           // pop 1 doubleword from stack
        jalr x0,0(x1)          // and return
```

32-bit Constants

- Most constants are small
  - 12-bit immediate is sufficient
- For the occasional 32-bit constant

  lui rd, constant
  - Copies 20-bit constant to bits [31:12] of rd
  - Extends bit 31 to bits [63:32]
  - Clears bits [11:0] of rd to 0

```
lui x19, 976  // 0x003D0
```

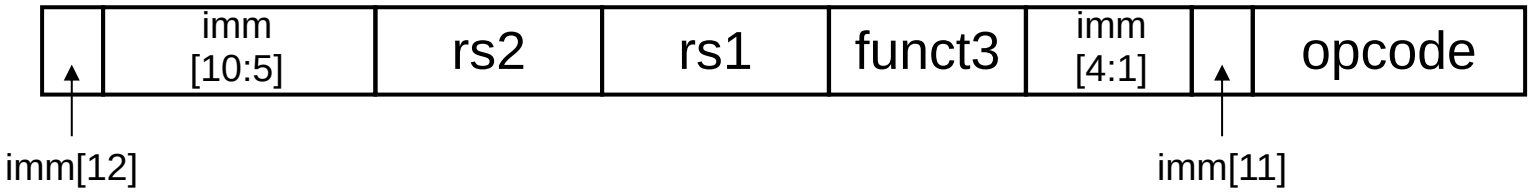
0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0000 0000 0000

```
addi x19,x19,128  // 0x500
```

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

Branch Addressing

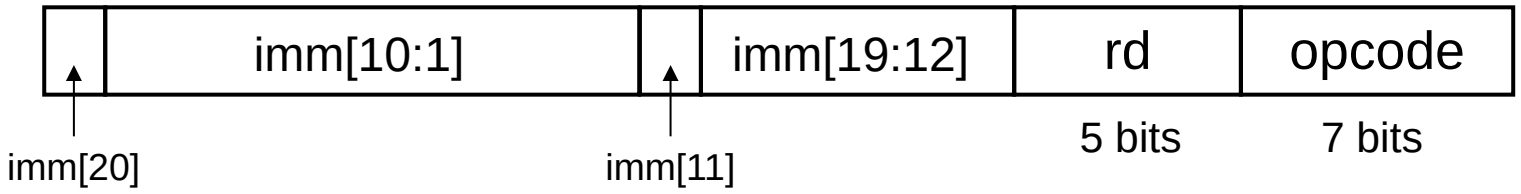
- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward
- SB format:



- PC-relative addressing
  - Target address = PC + immediate × 2

Jump Addressing

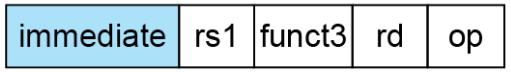
- Jump and link (jal) target uses 20-bit immediate for larger range
- UJ format:



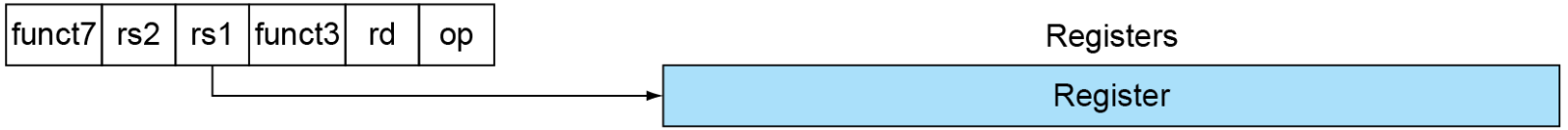
- For long jumps, eg, to 32-bit absolute address
  - lui: load address[31:12] to temp register
  - jalr: add address[11:0] and jump to target

RISC-V Addressing Summary

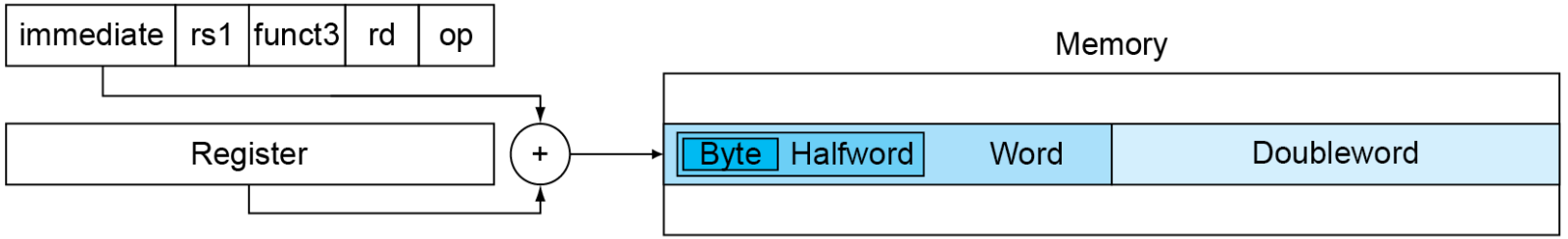
1. Immediate addressing



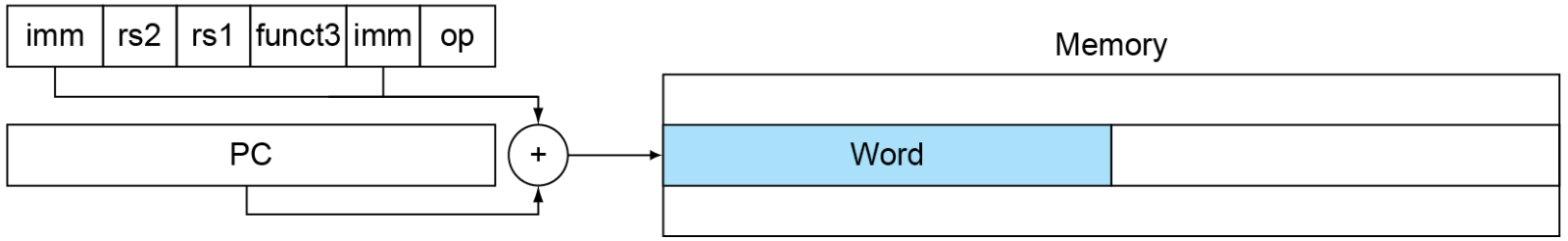
2. Register addressing



3. Base addressing



4. PC-relative addressing



RISC-V Encoding Summary

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

RISC-V Encoding Summary

Format	Instruction	Opcode	Funct3	Funct6/7
R-type	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100
I-type	lb	0000011	000	n.a.
	lh	0000011	001	n.a.
	lw	0000011	010	n.a.
	lbu	0000011	100	n.a.
	lhu	0000011	101	n.a.
	addi	0010011	000	n.a.
	slli	0010011	001	000000
	xori	0010011	100	n.a.
	srli	0010011	101	000000
	srai	0010011	101	010000
	ori	0010011	110	n.a.
	andi	0010011	111	n.a.
	jalr	1100111	000	n.a.
S-type	sb	0100011	000	n.a.
	sh	0100011	001	n.a.
	sw	0100011	010	n.a.
SB-type	beq	1100111	000	n.a.
	bne	1100111	001	n.a.
	blt	1100111	100	n.a.
	bge	1100111	101	n.a.
	bltu	1100111	110	n.a.
	bgeu	1100111	111	n.a.
U-type	lui	0110111	n.a.	n.a.
UJ-type	jal	1101111	n.a.	n.a.

Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

Synchronization in RISC-V

- Load reserved: lr.d rd,(rs1)
  - Load from address in rs1 to rd
  - Place reservation on memory address
- Store conditional: sc.d rd,(rs1),rs2
  - Store from rs2 to address in rs1
  - Succeeds if location not changed since the lr.d
    - Returns 0 in rd
- Fails if location is changed
  - Returns non-zero value in rd

Synchronization in RISC-V

- Example 1: atomic swap (to test/set lock variable)

```
again:  lr.d x10,(x20)
        sc.d x11,(x20),x23 // X11 = status
        bne  x11,x0,again  // branch if store failed
        addi x23,x10,0     // X23 = loaded value
```

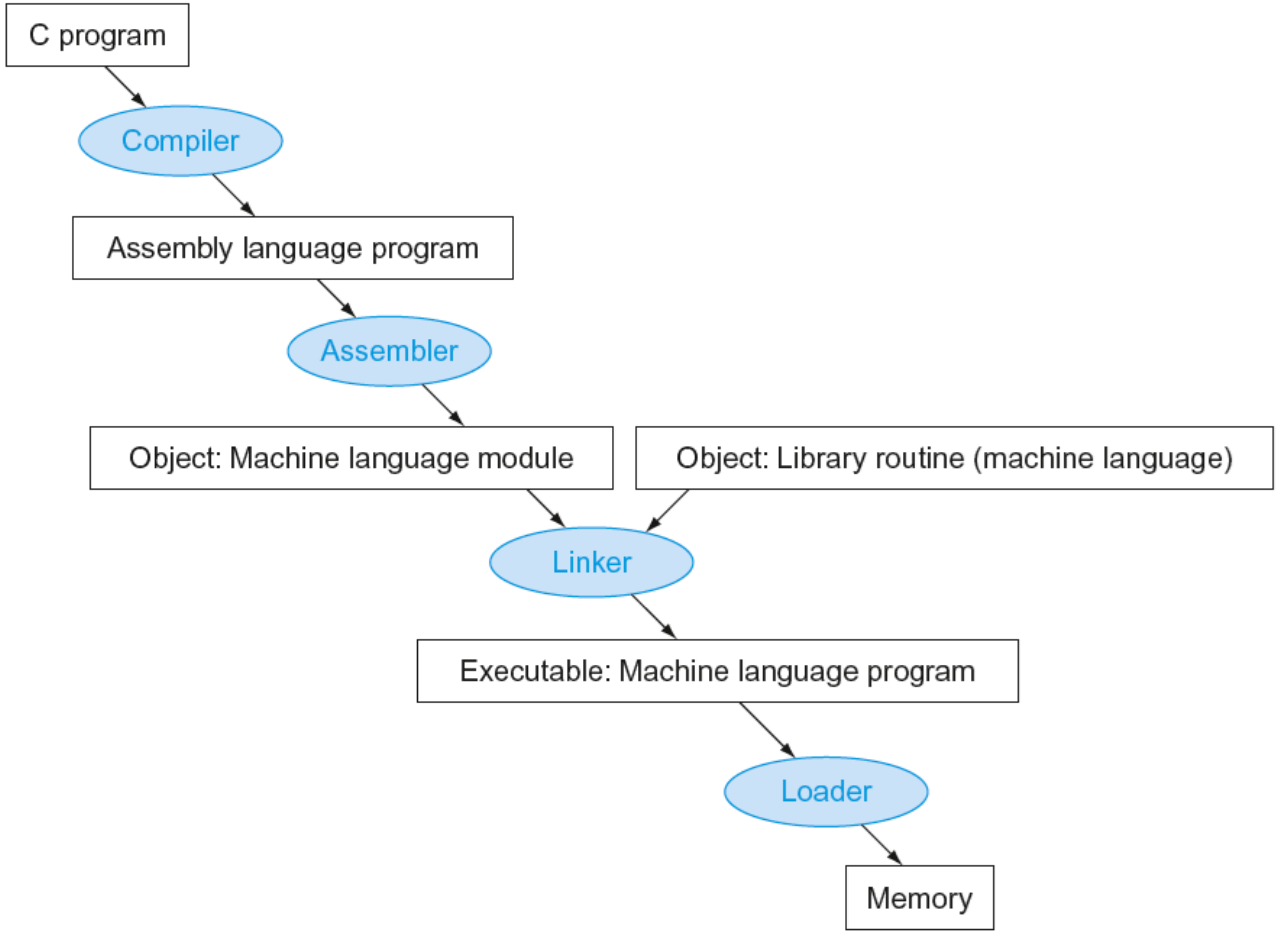
- Example 2: lock

```
        addi x12,x0,1              // copy locked value
again:  lr.d x10,(x20)             // read lock
        bne  x10,x0,again          // check if it is 0 yet
        sc.d x11,(x20),x12         // attempt to store
        bne  x11,x0,again// branch if fails
```

- Unlock:

```
        sd   x0,0(x20)             // free lock
```

Translation and Startup



Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

Linking Object Modules

- Produces an executable image
    1. Merges segments
    2. Resolve labels (determine their addresses)
    3. Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
    - But with virtual memory, no need to do this
    - Program can be loaded into absolute location in virtual memory space

Linking Object Modules

- Load from image file on disk into memory

1. Read header to determine segment sizes
2. Create virtual address space
3. Copy text and initialized data into memory
   - Or set page table entries so they can be faulted in
4. Set up arguments on stack
5. Initialize registers (including sp, fp, gp)
6. Jump to startup routine
   - Copies arguments to x10, … and calls main
   - When main returns, do exit syscall

Linking Object Files (Examples)

Linking Object Files

EXAMPLE

Link the two object files below. Show updated addresses of the first few instructions of the completed executable file. We show the instructions in assembly language just to make the example understandable; in reality, the instructions would be numbers.

Note that in the object files we have highlighted the addresses and symbols that must be updated in the link process: the instructions that refer to the addresses of procedures `A` and `B` and the instructions that refer to the addresses of data words `X` and `Y`.

Procedure `A` needs to find the address for the variable labeled `X` to put in the load instruction and to find the address of procedure `B` to place in the `jal`

Linking Object Files (Examples)

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	`lw x10, 0(x3)`	
	4	`jal x1, 0`	
	...	...	
Data segment	0	(X)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	`lw`	X
	4	`jal`	B
Symbol table	Label	Address	
	X	–	
	B	–	

	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	`sw x11, 0(x3)`	
	4	`jal x1, 0`	
	...	...	
Data segment	0	(Y)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	`sw`	Y
	4	`jal`	A
Symbol table	Label	Address	
	Y	–	
	A	–	

Linking Object Files (Examples)

ANSWER

instruction. Procedure B needs the address of the variable labeled Y for the store instruction and the address of procedure A for its jal instruction.

From Figure 2.14 on page 113, we know that the text segment starts at address $0000\ 0000\ 0040\ 0000_{hex}$ and the data segment at $0000\ 0000\ 1000\ 0000_{hex}$. The text of procedure A is placed at the first address and its data at the second. The object file header for procedure A says that its text is 100_{hex} bytes and its data is 20_{hex} bytes, so the starting address for procedure B text is $40\ 0100_{hex}$, and its data starts at $1000\ 0020_{hex}$.

Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0000\ 0000\ 0040\ 0000_{hex}$	lw x10, 0(x3)
	$0000\ 0000\ 0040\ 0004_{hex}$	jal x1, 252_{ten}
	...	...
	$0000\ 0000\ 0040\ 0100_{hex}$	sw x11, 32(x3)
	$0000\ 0000\ 0040\ 0104_{hex}$	jal x1, -260_{ten}
	...	...
Data segment	Address	
	$0000\ 0000\ 1000\ 0000_{hex}$	(X)
	...	...
	$0000\ 0000\ 1000\ 0020_{hex}$	(Y)
	...	...

Linking Object Files (Examples)

Now the linker updates the address fields of the instructions. It uses the instruction type field to know the format of the address to be edited. We have three types here:

1. The jump and link instructions use PC-relative addressing. Thus, for the `jal` at address $40\ 0004_{hex}$ to go to $40\ 0100_{hex}$ (the address of procedure `B`), it must put ($40\ 0100_{hex} - 40\ 0004_{hex}$) or 252_{ten} in its address field. Similarly, since $40\ 0000_{hex}$ is the address of procedure `A`, the `jal` at $40\ 0104_{hex}$ gets the negative number -260_{ten} ($40\ 0000_{hex} - 40\ 0104_{hex}$) in its address field.

2. The load addresses are harder because they are relative to a base register. This example uses `x3` as the base register, assuming it is initialized to $0000\ 0000\ 1000\ 0000_{hex}$. To get the address $0000\ 0000\ 1000\ 0000_{hex}$ (the address of word `X`), we place 0_{ten} in the address field of `lw` at address $40\ 0000_{hex}$. Similarly, we place 20_{hex} in the address field of `sw` at address $40\ 0100_{hex}$ to get the address $0000\ 0000\ 1000\ 0020_{hex}$ (the address of doubleword `Y`).

3. Store addresses are handled just like load addresses, except that their S-type instruction format represents immediates differently than loads' I-type format. We place 32_{ten} in the address field of `sw` at address $40\ 0100_{hex}$ to get the address $0000\ 0000\ 1000\ 0020_{hex}$ (the address of word `Y`).

Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

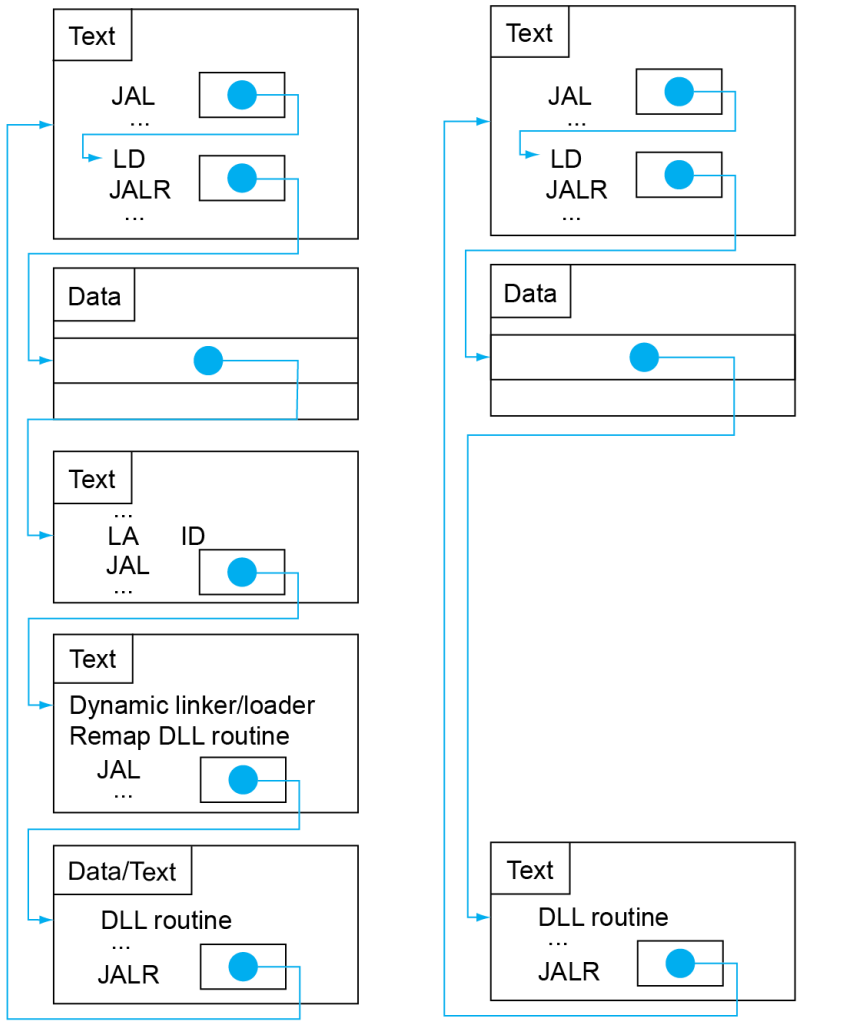
Lazy Linkage

Indirection table

Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

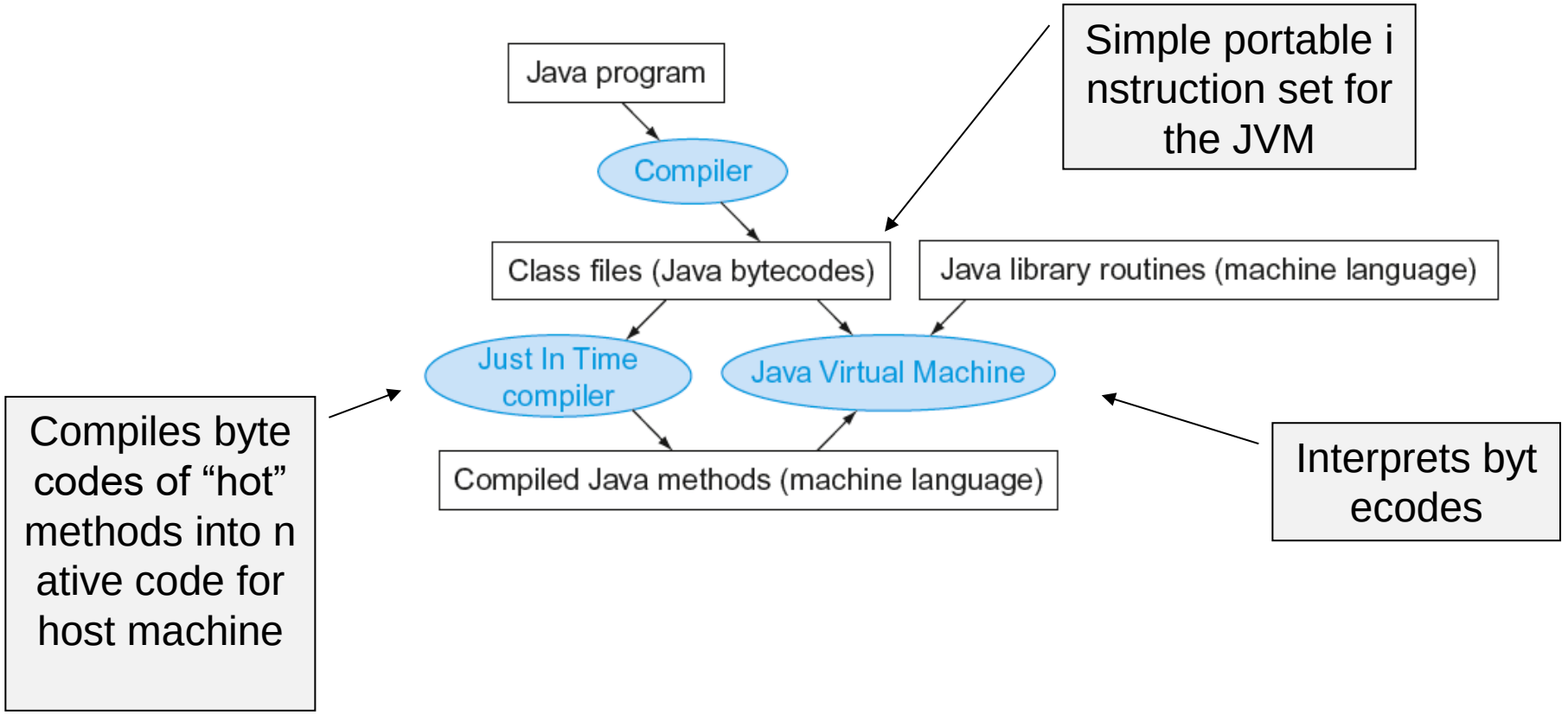
Dynamically
mapped code



(a) First call to DLL routine

(b) Subsequent calls to DLL routine

Starting Java Applications



C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

```
void swap(long long int v[],
          long long int k)
{
  long long int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

v in x10, k in x11, temp in x5

The Procedure Swap

```
swap:
   slli x6,x11,3     // reg x6 = k * 8
   add  x6,x10,x6    // reg x6 = v + (k * 8)
   ld   x5,0(x6)     // reg x5 (temp) = v[k]
   ld   x7,8(x6)     // reg x7 = v[k + 1]
   sd   x7,0(x6)     // v[k] = reg x7
   sd   x5,8(x6)     // v[k+1] = reg x5 (temp)
   jalr x0,0(x1)     // return to calling routine
```

The Sort Procedure in C

Non-leaf (calls swap)

```
void sort (long long int v[], size_t n)
{
  size_t i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i – 1;
         j >= 0 && v[j] > v[j + 1];
         j -= 1) {
      swap(v,j);
    }
  }
}
```

v in x10, n in x11, i in x19, j in x20

The Outer Loop

- Skeleton of outer loop:
  - for (i = 0; i <n; i += 1) {

```
        li   x19,0              // i = 0
```

- for1tst:

```
        bge  x19,x11,exit1   // go to exit1 if x19 ≥ x11 (i≥n)

        (body of outer for-loop)

        addi x19,x19,1              // i += 1
        j    for1tst          // branch to test of outer loop
```

- exit1:

The Inner Loop

Skeleton of inner loop:

```
for (j = i − 1; j >= 0 && v[j] > v[j + 1]; j − = 1) {
                    addi x20,x19,-1     // j = i −1
  for2tst:
      blt  x20,x0,exit2  // go to exit2 if X20 < 0 (j < 0)
      slli x5,x20,3       // reg x5 = j * 8
      add  x5,x10,x5      // reg x5 = v + (j * 8)
      ld   x6,0(x5)       // reg x6 = v[j]
      ld   x7,8(x5)       // reg x7 = v[j + 1]
      ble  x6,x7,exit2    // go to exit2 if x6 ≤ x7
```

The Inner Loop

```
mv   x21, x10      // copy parameter x10 into x21
mv   x22, x11      // copy parameter x11 into x22
mv   x10, x21      // first swap parameter is v
mv   x11, x20      // second swap parameter is j
        jal  x1,swap     // call swap
        addi x20,x20,-1          // j –= 1
        j    for2tst     // branch to test of inner loop
  exit2:
```

Preserving Registers

Preserve saved registers:

```
addi sp,sp,-40  // make room on stack for 5 regs
sd   x1,32(sp)  // save x1 on stack
sd   x22,24(sp) // save x22 on stack
sd   x21,16(sp) // save x21 on stack
sd   x20,8(sp)  // save x20 on stack
sd   x19,0(sp)  // save x19 on stack
```

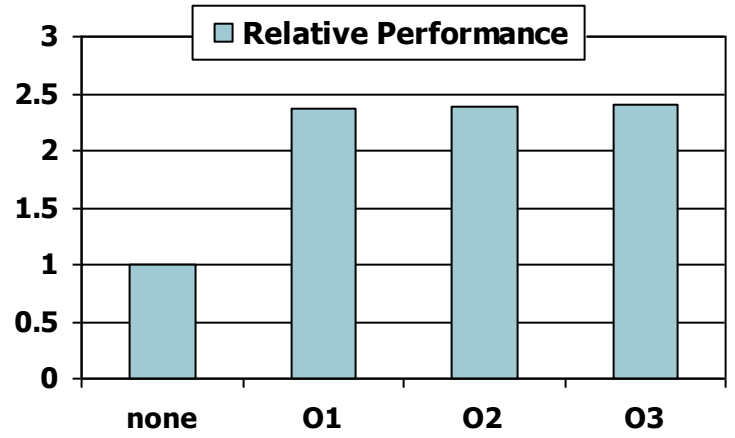
Preserving Registers

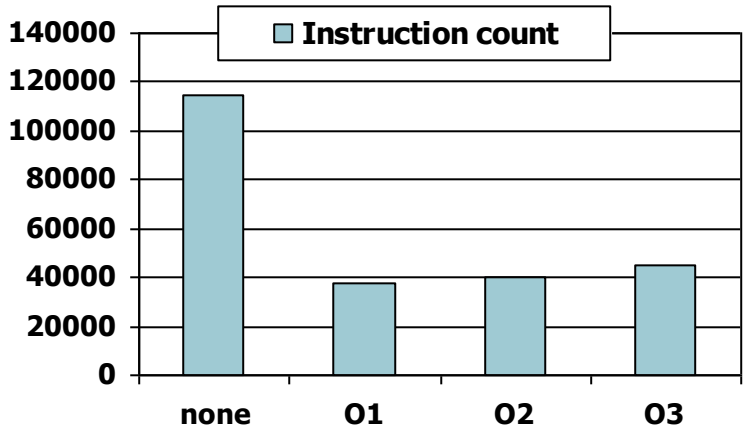
Restore saved registers:

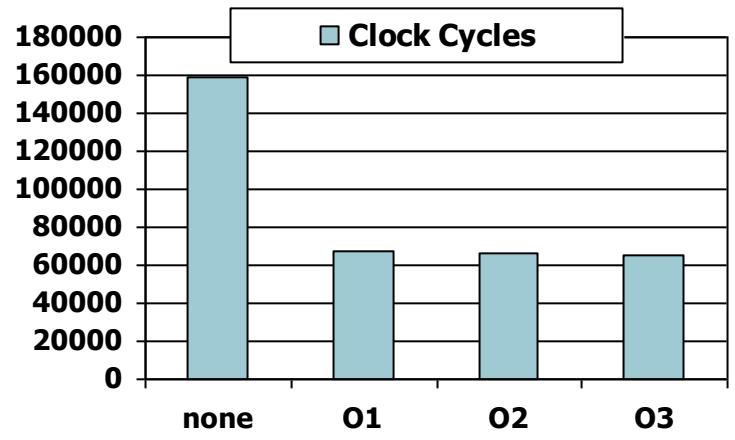
```
exit1:
  sd    x19,0(sp)         // restore x19 from stack
        sd    x20,8(sp)   // restore x20 from stack
        sd    x21,16(sp) // restore x21 from stack
        sd    x22,24(sp) // restore x22 from stack
        sd    x1,32(sp)   // restore x1 from stack
        addi sp,sp, 40   // restore stack pointer
  jalr x0,0(x1)
```

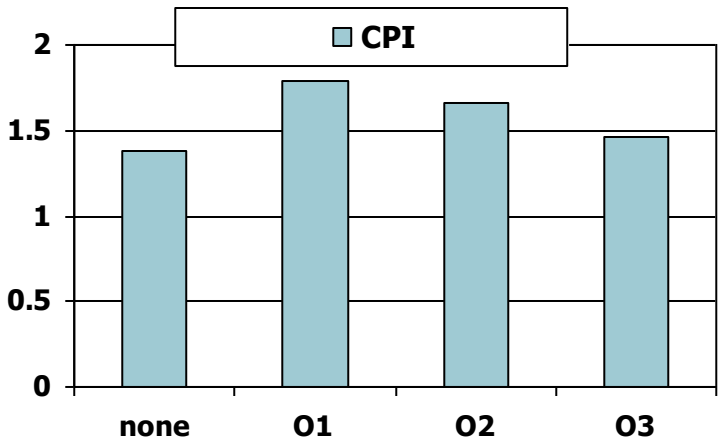
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

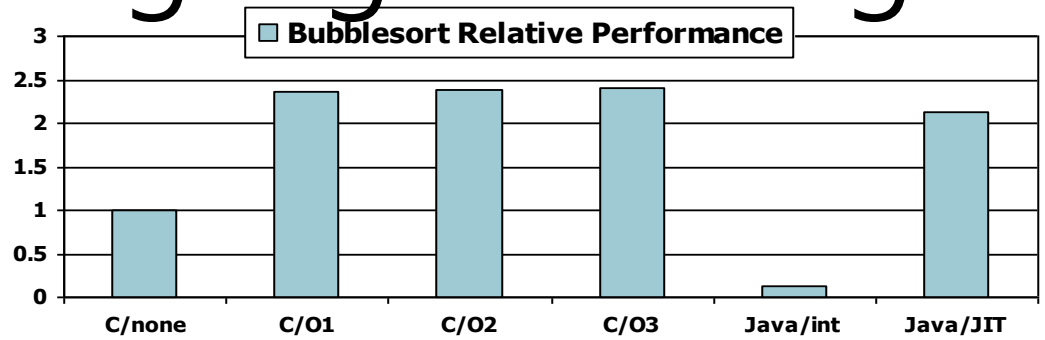


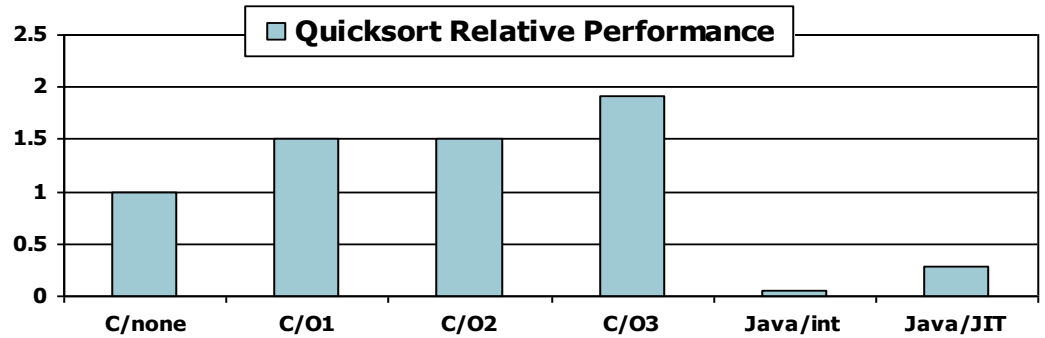


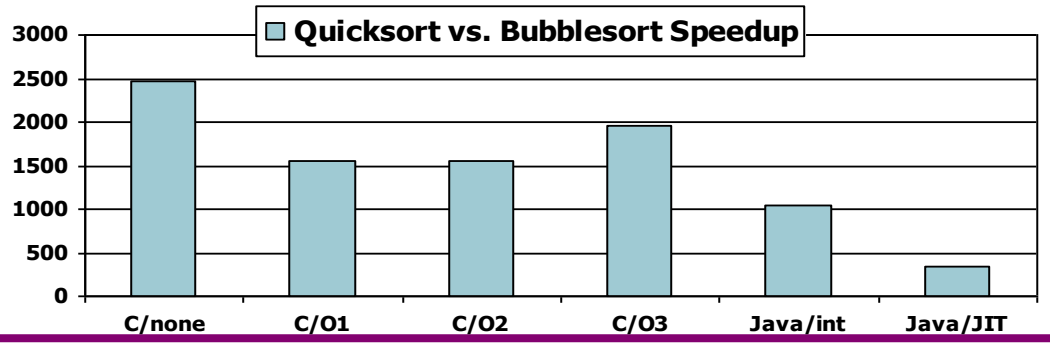




Effect of Language and Algorithm







Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!

Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

Example: Clearing an Array

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```

```
   li   x5,0       // i = 0
loop1:
   slli x6,x5,3    // x6 = i * 8
   add  x7,x10,x6  // x7 = address
                   // of array[i]
   sd   x0,0(x7)   // array[i] = 0
   addi x5,x5,1    // i = i + 1
   blt  x5,x11,loop1  // if (i<size)
                      // go to loop1
```

```
   mv x5,x10       // p = address
                   // of array[0]
   slli x6,x11,3   // x6 = size * 8
   add x7,x10,x6   // x7 = address
                   // of array[size]
loop2:
   sd x0,0(x5)     // Memory[p] = 0
   addi x5,x5,8    // p = p + 8
   bltu x5,x7,loop2
                   // if (p<&array[size])
                   // go to loop2
```

Comparison of Array vs. Ptr

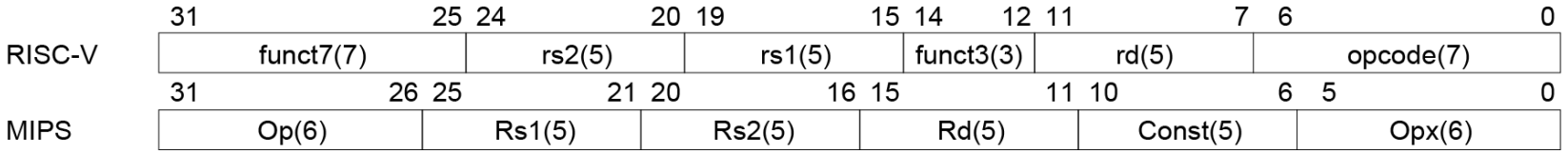
- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer

MIPS Instructions

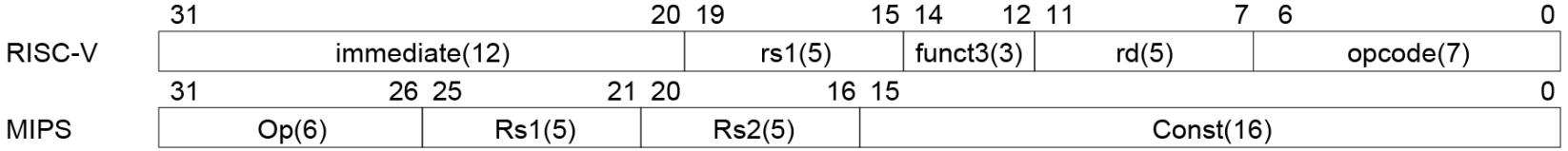
- MIPS: commercial predecessor to RISC-V
- Similar basic set of instructions
  - 32-bit instructions
  - 32 general purpose registers, register 0 is always 0
  - 32 floating-point registers
  - Memory accessed only by load/store instructions
    - Consistent use of addressing modes for all data sizes
- Different conditional branches
  - For <, <=, >, >=
  - RISC-V: blt, bge, bltu, bgeu
  - MIPS: slt, sltu (set less than, result is 0 or 1)
    - Then use beq, bne to complete the branch

Instruction Encoding

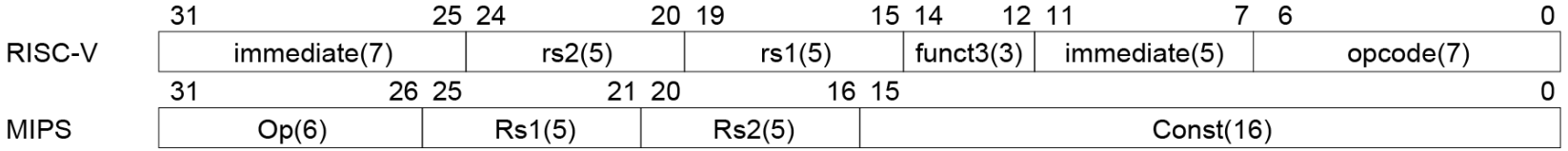
Register-register



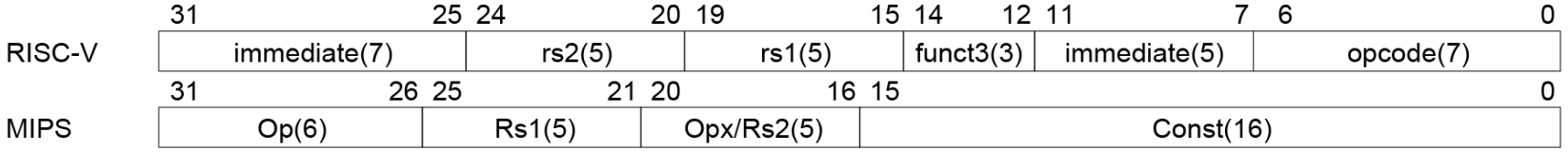
Load



Store



Branch



The Intel x86 ISA

- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

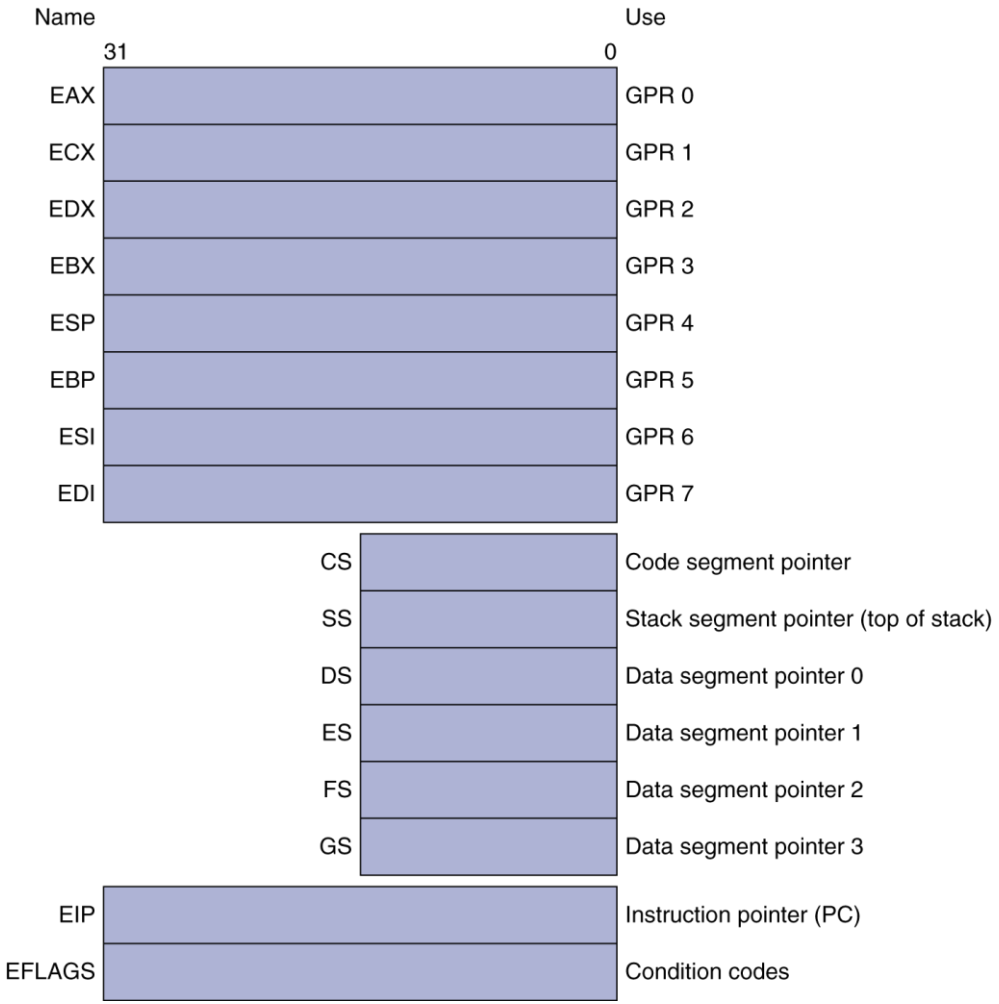
The Intel x86 ISA

- Further evolution…
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, …
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture (see Colwell, The Pentium Chronicles)
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions

The Intel x86 ISA

- And further…
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead…
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
  - Technical elegance ≠ market success

Basic x86 Registers



Basic x86 Addressing Modes

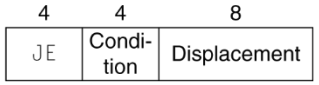
- Two operands per instruction

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

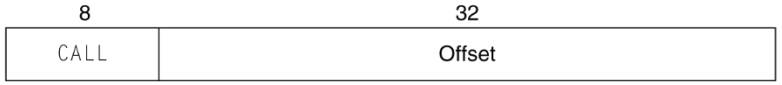
- Memory addressing modes
  - Address in register
  - Address = R_{base} + displacement
  - Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement

x86 Instruction Encoding

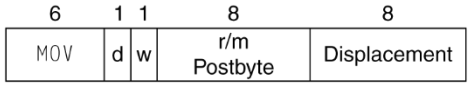
a. JE EIP + displacement



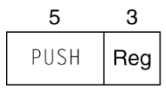
b. CALL



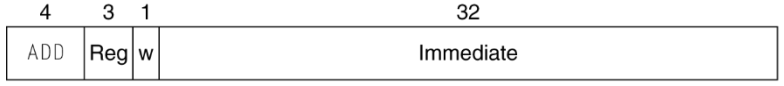
c. MOV EBX, [EDI + 45]



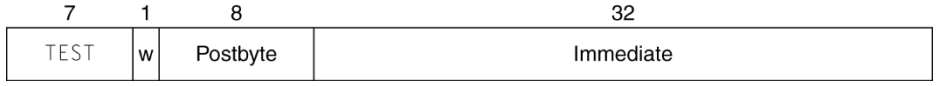
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, …

Implementing IA-32

- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - Compilers avoid complex instructions

Other RISC-V Instructions

- Base integer instructions (RV64I)
  - Those previously described, plus
  - auipc rd, immed // rd = (imm<<12) + pc
    - follow by jalr (adds 12-bit immed) for long jump
  - slt, sltu, slti, sltui: set less than (like MIPS)
  - addw, subw, addiw: 32-bit add/sub
  - sllw, srlw, srlw, slliw, srliw, sraiw: 32-bit shift
- 32-bit variant: RV32I
  - registers are 32-bits wide, 32-bit operations

Instruction Set Extensions

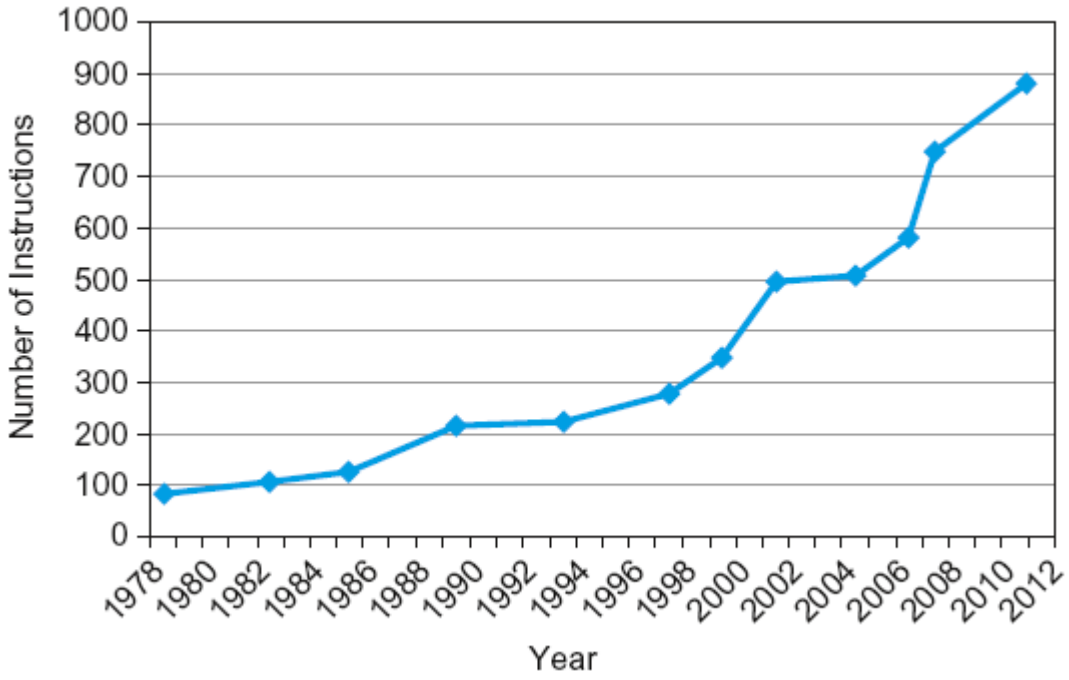
- M: integer multiply, divide, remainder
- A: atomic memory operations
- F: single-precision floating point
- D: double-precision floating point
- C: compressed instructions
  - 16-bit encoding for frequently used instructions

Fallacies

- Powerful instruction → higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code → more errors and less productivity

Fallacies

- Backward compatibility → instruction set doesn't change
  - But they do accrete more instructions



x86 instruction set

Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
  - e.g., passing pointer back via an argument
  - Pointer becomes invalid when stack popped

Concluding Remarks

- Design principles
    1. Simplicity favors regularity
    2. Smaller is faster
    3. Good design demands good compromises
- Make the common case fast
- Layers of software/hardware
    - Compiler, assembler, hardware
- RISC-V: typical of RISC ISAs
    - c.f. x86

감사합니다.