Computer Architecture
Chapter 2

Instructions: Language of the computer
3

Joonho Kong
School of EE, KNU

RISC-V Addressing for Wide Immediate and Address

Load upper immediate (lui) instruction (U-type)

- lui instruction is a solution for the problem of the immediate data size.
- This instruction loads a 20-bit constants into bits 12~31 of a register.

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

Figure from Computer Organization and Design: RISC-V Edition

RISC-V Addressing for Wide Immediate and Address

Example of lui instruction

- What is the RISC-V assembly code to load this 64-bit constant into register x19?

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

Sol)

- First, load bits 12~31 with that bit pattern using lui instruction

  lui x19, 976

- Then, the value of x19 will be:

```
0000000 00000000 00000000 00000000 00000000 00111101 00000000 00000000
```

- The next step is to add in the lowest 12 bits:

  addi x19, x19, 1280 // 1280 = 00000101 00000000

- Then, the result of the loaded value in x19 is:

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

RISC-V Addressing for Wide Immediate and Address

Addressing in branches

SB-type

- It is only possible to branch to even addresses
- The LSB of instruction memory address is always 0

UJ-type

- The unconditional jump-and-link instruction (jal) is the UJ-type
- Like SB-type, it cannot encode odd addresses

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

- In SB and UJtype, we cannot see immediate[0] (immed[0]) because it will be always filled with 0

Figure from Computer Organization and Design: RISC-V Edition

RISC-V Addressing for Wide Immediate and Address

How to solve the branch address size problem?

Using PC-relative addressing

Program Counter = Register + Branch offset

- This can be a solution for the problem of the branch address size
- Conditional branches tend to branch to a nearby instruction
- We can branch within $\pm 2^{10}$ words of the current instruction address

  or jump within $\pm 2^{18}$ words of the current instruction address

Exploits common behaviors of workloads

RISC-V Addressing for Wide Immediate and Address

RISC-V addressing mode summary

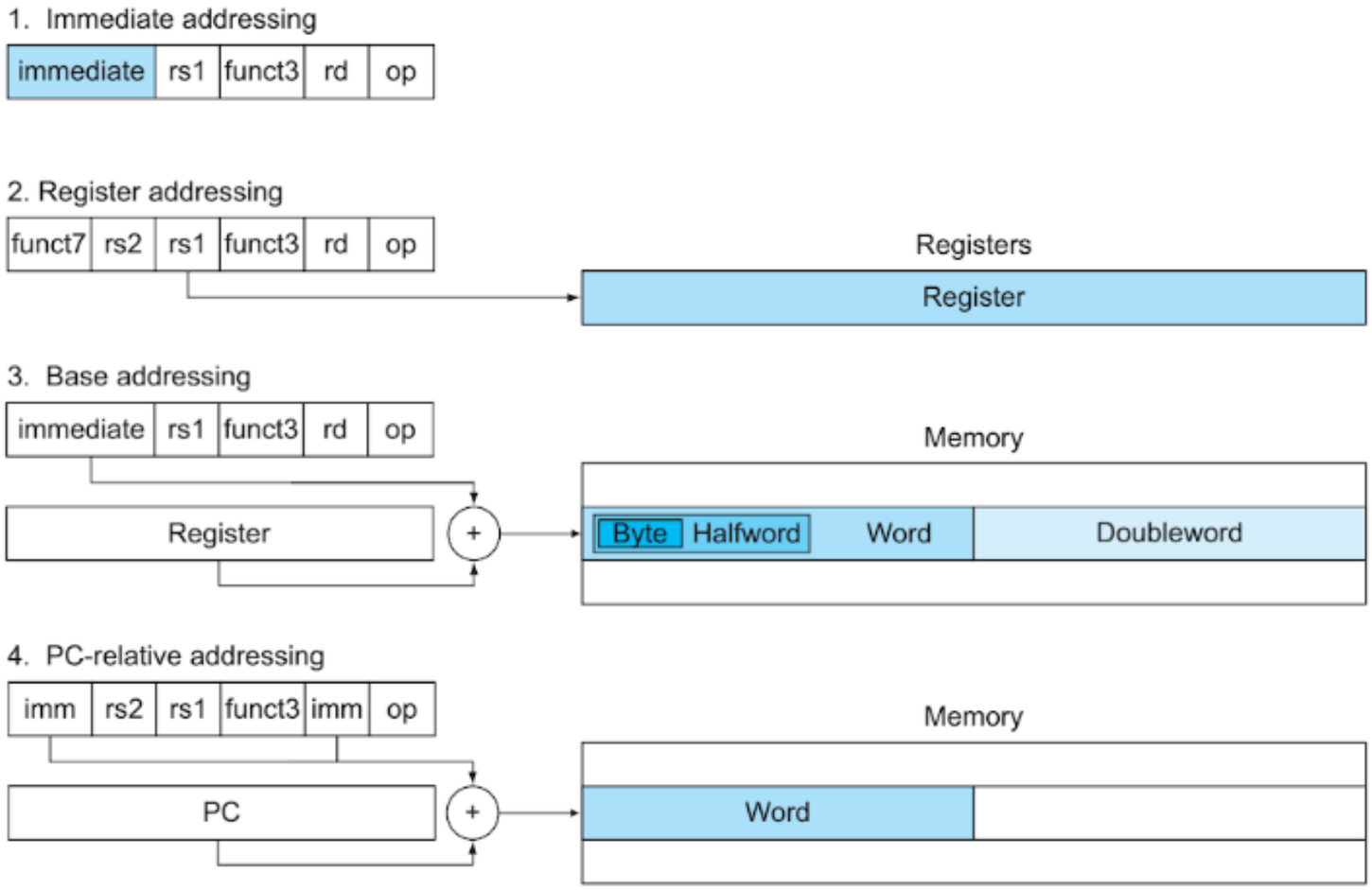


Figure from Computer Organization and Design: RISC-V Edition

RISC-V Addressing for Wide Immediate and Address

RISC-V addressing mode summary

Immediate addressing

: the operand is a constant within the instruction itself (immediate value)

Register addressing

: the operand is from a register

Base addressing

: the operand is at the memory location whose address is the sum of a register and offset (immediate value) in the instruction.

PC-relative addressing

: the branch address is the sum of the current PC and a offset (immediate value) in the instruction

Translating and Starting a Program

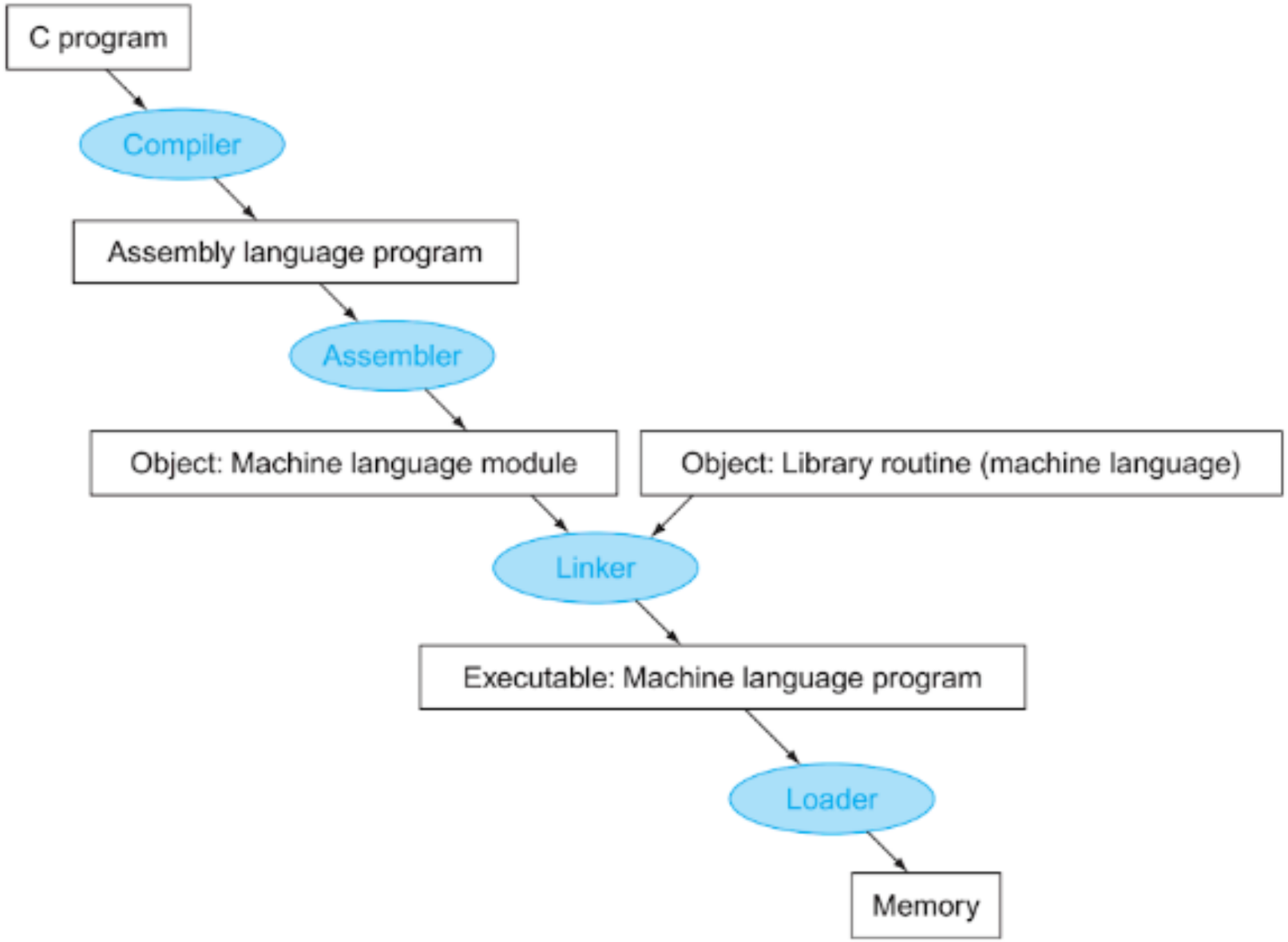


Figure from Computer Organization and Design: RISC-V Edition

Translating and Starting a Program

Compiler

- Translates the from C program to assembly language program

Assembler

- From assembly codes to machine codes (binary)
- When we code using pseudo instructions, e.g., mv (move) instruction, like:

```
mv  x10, x11    // register x10 gets the value of register x11.
```

  Assembler translates it into:

```
addi  x10, x11, 0      // register x10 gets register x11 + 0.
```

- Symbol table which saves labels used in branch and jump instructions.

Translating and Starting a Program

Linker

- The linker produces an executable file like object file that can be run on a computer
- Complete retranslation is a terrible waste of computing resources
- Computer compiles and assembles each procedure independently

→ A change to one line would require compiling and assembling only one procedure

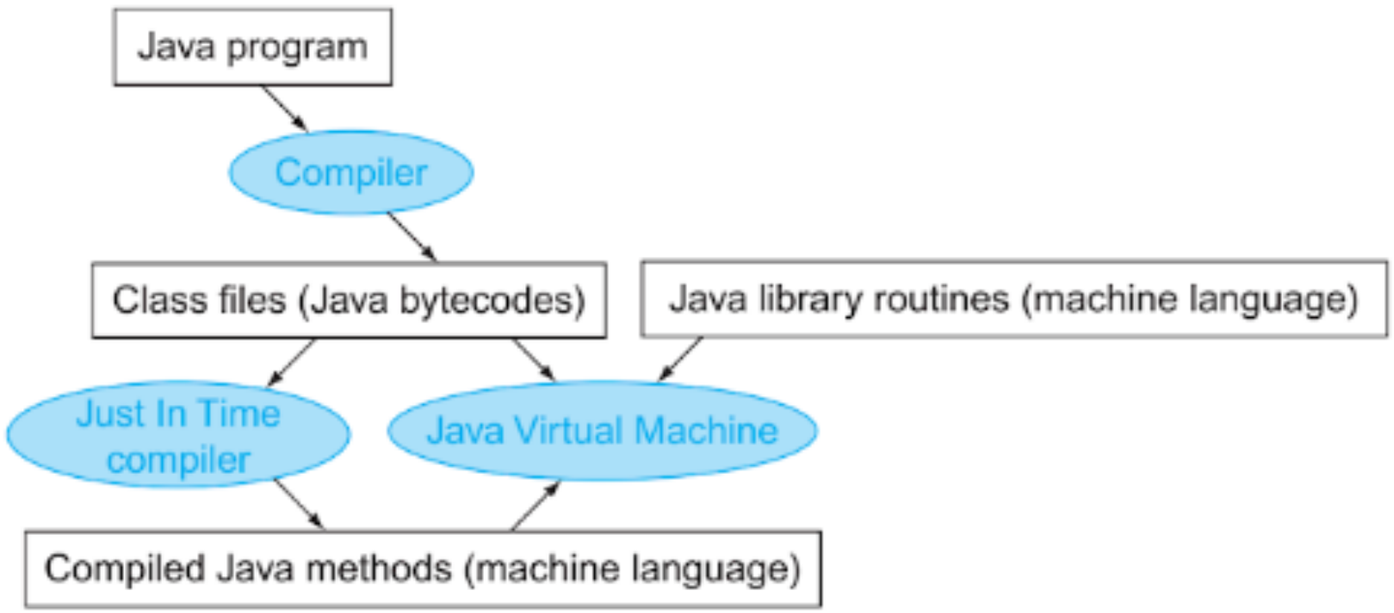
→ When recompile the program, compiler only compiles the modified procedure or source files

Loader

- After getting the executable file on permanent storage, the OS can reads it to memory and starts it

Translating and Starting a Program

Starting a java program



- Java bytecode instruction set
  : Java code is compiled to bytecodes that are easy to interpret
- Java Virtual Machine (JVM)
  : The JVM which is a software interpreter can execute Java bytecode
- Upside of interpretation: Portability
- Downside of interpretation: Lower performance

Figure from Computer Organization and Design: RISC-V Edition

Translating and Starting a Program

Starting a java program

- Java In Time compilers (JIT)
  - Maintains the native codes (i.e., binary) of hot methods (frequently used methods)
  - Removes re-interpretation overheads of frequently used methods
  - The balance of interpretation and compilation
  - **Make common case fast**

- The performance gap between Java and C or C++ is now very small
- JVM overheads would be negligible in high-performance machines