National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Lab1: RISC-V Assembly Language Programming

CS10014 Computer Organization

Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao University

Acknowledgements and Disclaimer

- Slides were developed in the reference with
  - RISC-V Programming
    - https://riscv-programming.org/book/riscv-book.html
  - CENG3420, CUHK
    - https://www.cse.cuhk.edu.hk/~byu/CENG3420/2022Spring/slides/lab1-1.pdf

Outline

- RISC-V Assembly Programming
  - Registers
  - Labels
  - Symbols
  - Directives
- Ripes RISC-V Assembly Simulator

Generating native programs

- A native program is a program
  - Encoded using instructions that can be directly executed by the computer hardware

C Code

```
int main ()
{
   int r = func (10);
   return r+1;
}
```

RISC-V assembly code

```
.text
.align 2
main:
   addi sp, sp, -16
   li   a0, 10
   sw   ra, 12(sp)
   jal  func
   lw   ra, 12(sp)
   addi a0, a0, 1
   addi sp, sp, 16
   ret
```

Generating native programs

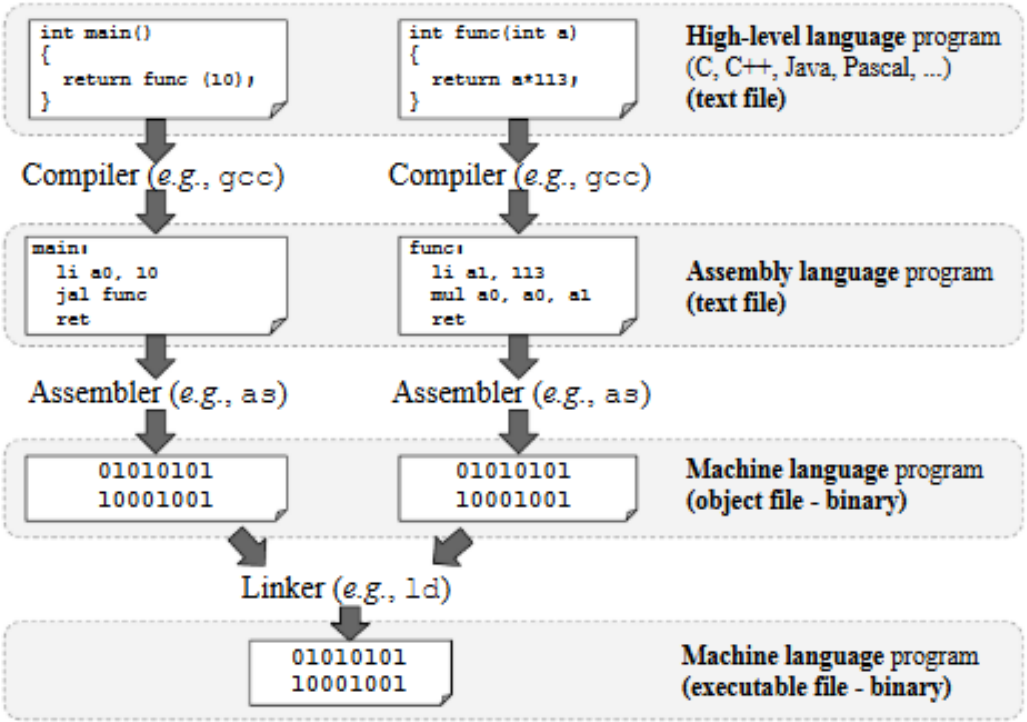
- **A compiler**
  - translate a program from one language to another
  - riscv64-unknown-elf-gcc -mabi=ilp32 -march=rv32i -S main.c -o main.s
  - The RV32I assembly program will be stored on the main.s file
- **An Assembler**
  - A assembler is a tool that translates a program in assembly language into a program in machine language
  - The GNU assembler tool (as) is an assembler
  - The assembler produces object files (.o) that are encoded in binary and contains code in machine language
  - riscv64-unknown-elf-as -mabi=ilp32 -march=rv32i main.s -o main.o

Generating native programs

- **A linker**
  - A tool that ‘links’ together one or more object files
  - Produces an executable file

riscv64-unknown-elf-ld -m elf32lriscv main.o mylib.o -o main.x

main.x is an executable file



Registers

- There are 32 registers in the RISC-V base ISA
  - These registers can be used directly in assembly programming
  - Some of registers have conventional roles
    - e.g. return address, stack pointer
  - RISC-V extensions may introduce additional register files
    - e.g. floating point extension, vector extension

Registers

Table B.4 Register names and numbers

Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0–2	x5–7	Temporary registers
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0–1	x10–11	Function arguments / Return values
a2–7	x12–17	Function arguments
s2–11	x18–27	Saved registers
t3-6	x28–31	Temporary registers

Labels

- A label in assembly language
  - As a marker that represent program location
  - ‘x:’ label identifies a program location that contains a variable, which is allocated and initialized by the directive .word 10
  - ‘sum10:’ label identifies the program location that contains the first instruction of the sum10 routine

RISC-V assembly code

```
x:
   .word 10
sum10:
   lw   a0, x
   addi a0, a0, 10
   ret
```

Program Symbol

- Program symbols
  - “names” that are associated with numerical values
  - The “symbol table” is a data structure that maps each program system to its value
  - “nm” tool helps us to inspect the symbol table of a program

```
$ riscv64-unknown-elf-nm sum10.o
00000004 t .L0
00000004 t sum10
00000000 t x
```

Program Symbol

- Using the “.set” directive
  - Explicitly define symbols
  - The follow example that uses **.set** directive to define a symbol named answer and assign value 42 to it
  - Two symbols: answer, get_answer

```
.set   answer, 42
get_answer
   li   a0, answer
   ret
```

Global vs local Symbols

- Local symbols
  - Only visible on the same file
  - By default, the assembler registers labels as local symbols
- The .global directive
  - Instructs the assembler to register a label as a global symbol

```
.global  exit
exit:
   li   a0, 0
   li   a7, 93
   ecall
```

Program Entry Point

- The entry point is defined by an address
  - The address of the first instruction that must be executed
  - The linker sets the entry point field on the executable file and looks for a symbol named start
  - The linker sets the entry point to a default value (the address of the first instruction of the program) if the linker cannot find “start” symbol

```
.global  start
start:
   li   a0, 10
   li   a1, 20
   jal  exit
```

Program Section

- The assembly program is usually organized in ‘sections’
  - A section may contain data or instructions
  - The contents of each section are mapped to a set of consecutive main memory addresses
  - .text
    - Store the program instructions
  - .data
    - Store initialized global variables
  - .bss
    - Store uninitialized global variables
  - .rodata
    - Store constants

Assembly Language

- Assembly program contains
  - Comment
  - Labels
    - Usually defined by a name ended with the suffix “:”
  - Assembly instructions
    - Converted by the assembler into machine instructions
    - E.g. addi a0, a1, 1
  - Assembly directives
    - Commands used to coordinate the assembling process
    - E.g. .word 10
      - Instruct the assembler to assemble a 32-bit value (10) into the program

Program Structure I

- Plain text file with data declarations
- Data declaration section is followed by program code section

Data Declarations

- Identified with assembler directive **.data**
- Declares variable names used in program
- Storage allocated in main memory (*e.g.*, RAM)
- `<name>:  .<datatype> <value>`

Program Structure II

Code

- placed in section of text identified with assembler directive **.text**
- contains program code (instructions)
- starting point for code e.g. execution given label **start:**

Comments

Anything following # on a line

Program Structure III

The structure of an assembly program looks like this:

Program outline

```
# Comment giving name of program and description
# Template.asm
# Bare-bones outline of RISC-V assembly language program

.globl _start

.data    # variable declarations follow this line
         # ...
.text    # instructions follow this line

_start: # indicates start of code
# ...

# End of program, leave a blank line afterwards is preferred
```

An Example RISC-V Assembly Program

```
.global _start

.data
1 reference
welcome_msg: .asciz "Welcome!"

.text
1 reference
_start:
    # STDOUT = 1
    addi a0, x0, 1
    # Load the address of 'welcome_msg'
    la a1, welcome_msg
    # Length of the string
    addi a2, x0, 8
    # Linux write system call
    addi a7, x0, 64
    # Call Linux service to output the string
    ecall
```

RISC-V Card

- You can search instructions in RISC-V Card

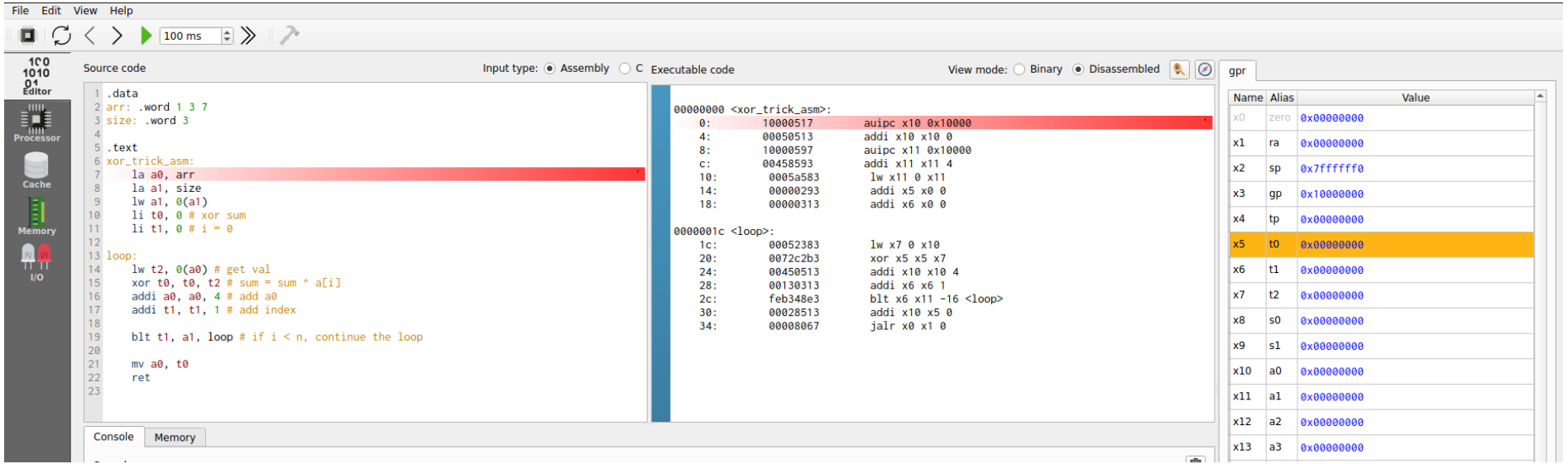
Table B.1 RV32I: RISC-V integer instructions

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	`lb rd, imm(rs1)`	load byte	`rd = SignExt([Address]7:0)`
0000011 (3)	001	–	I	`lh rd, imm(rs1)`	load half	`rd = SignExt([Address]15:0)`
0000011 (3)	010	–	I	`lw rd, imm(rs1)`	load word	`rd = [Address]31:0`
0000011 (3)	100	–	I	`lbu rd, imm(rs1)`	load byte unsigned	`rd = ZeroExt([Address]7:0)`
0000011 (3)	101	–	I	`lhu rd, imm(rs1)`	load half unsigned	`rd = ZeroExt([Address]15:0)`
0010011 (19)	000	–	I	`addi rd, rs1, imm`	add immediate	`rd = rs1 + SignExt(imm)`
0010011 (19)	001	0000000*	I	`slli rd, rs1, uimm`	shift left logical immediate	`rd = rs1 << uimm`
0010011 (19)	010	–	I	`slti rd, rs1, imm`	set less than immediate	`rd = (rs1 < SignExt(imm))`
0010011 (19)	011	–	I	`sltiu rd, rs1, imm`	set less than imm. unsigned	`rd = (rs1 < SignExt(imm))`
0010011 (19)	100	–	I	`xori rd, rs1, imm`	xor immediate	`rd = rs1 ^ SignExt(imm)`
0010011 (19)	101	0000000*	I	`srli rd, rs1, uimm`	shift right logical immediate	`rd = rs1 >> uimm`
0010011 (19)	101	0100000*	I	`srai rd, rs1, uimm`	shift right arithmetic imm.	`rd = rs1 >>> uimm`
0010011 (19)	110	–	I	`ori rd, rs1, imm`	or immediate	`rd = rs1 | SignExt(imm)`
0010011 (19)	111	–	I	`andi rd, rs1, imm`	and immediate	`rd = rs1 & SignExt(imm)`
0010111 (23)	–	–	U	`auipc rd, upimm`	add upper immediate to PC	`rd = {upimm, 12'b0} + PC`
0100011 (35)	000	–	S	`sb rs2, imm(rs1)`	store byte	`[Address]7:0 = rs27:0`

RISC-V ISA Simulator - Ripes

- Ripes
  - The RISC-V visual computer architecture simulator and assembly code editor
  - **Ripes** supports RISC-V IMC ISA base (riscv32 & riscv64)
  - **Ripes** provides multiple processor models (Single Cycle, 5-Stage)
  - Download Ripes
    - https://github.com/mortbopet/Ripes

RISC-V ISA Simulator - Ripes



RISC-V ISA Simulator - Ripes

- You can select different processor model in Ripes.