香港中文大學
The Chinese University of Hong Kong

CENG3420

Lab 1-1: RISC-V Assembly Language Programing I

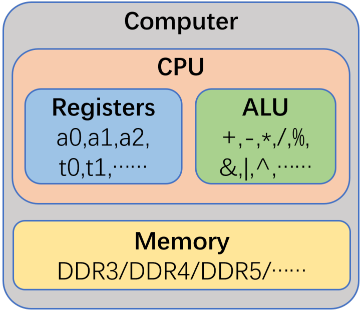
Mingjun LI, Fangzhou LIU
Department of Computer Science & Engineering
Chinese University of Hong Kong
{mjli23, fzliu23}@cse.cuhk.edu.hk

Spring 2025

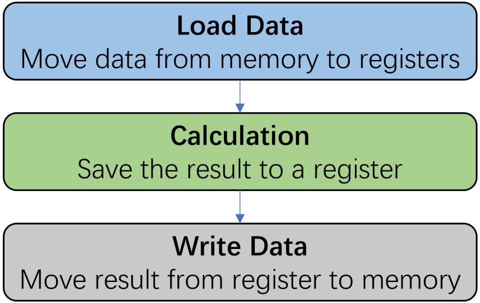
Outline

Introduction to Basic RISC-V Assembly Programing

- **Computer**, oversimplified.



- **Computing**, oversimplified.



What is Assembly Language?

Definition

Assembly language is a low-level programming language that provides a symbolic representation of machine code instructions. It's specific to a particular computer architecture (like RISC-V).

- Direct hardware control
- One-to-one correspondence with machine code
- Architecture-specific syntax

What is an Assembler?

Definition

An assembler is a program that translates assembly language into machine code. It's the tool that converts human-readable assembly instructions into binary code that the computer can execute.

Example

Assembly: `add x1, x2, x3`
↓ (Assembler)
Machine Code: `00000000001100010000000010110011`

What is a RISC-V Simulator?

Definition

A RISC-V simulator is a software tool that emulates RISC-V processor behavior, allowing programmers to run and test RISC-V assembly programs without physical hardware.

- Executes RISC-V instructions virtually
- Provides detailed execution feedback
- Useful for education and debugging

RARS

RARS is an educational simulator based on MARS (MIPS Assembler and Runtime Simulator), specifically adapted for RISC-V architecture.

- Integrated development environment (IDE)
- Built-in text editor
- Assembler and simulator combined
- Java-based (platform independent)

RISC-V Instruction Set

Important Material

The RISC-V Instruction Set Manual Volume I: Unprivileged ISA
`https://riscv.org/technical/specifications/`

In all labs of CENG3420, we focus on RV32I instructions.

An Example Assembly Language Program

- How to compute "C = A + B"

resC = varA + varB => resC = 8 after execution

```
    .globl _start

    .data    # global variable declarations follow this line
    varA: .word 3 # 1 word = 32 bits
    varB: .word 5
    resC: .word 0

    .text    # instructions follow this line
    _start: # a label, marks a position in the code
        la a1, varA # Load varA's address to register a1
        la a2, varB # Load varB's address to register a2
        la a3, resC # Load resC's address to register a3
        lw t1, 0(a1) # Load varA's value to register t1
        lw t2, 0(a2) # Load varB's value to register t2
        add t3, t1, t2 # Register t3 = t1 + t2
        sw t3, 0(a3) # Save register t3 to resC
```

Program Structure I

- Plain text file with data declarations, program code (usually suffixed with *.asm*)
- 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>`
  - `.byte` (1 byte/8 bits), `.2byte`, `.half`, `.short` (2 bytes)
  - `.4byte`, `.word`, `.long` (4 bytes), `.8byte`, `.dword`, `.quad` (8 bytes)
  - `.float`, `.double`, ...

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
```

Data Types and Literals

Data types:

- All instructions are encoding in 32 bits
- Alias: byte (8 bits), halfword (2 bytes), word (4 bytes), double word (8 bytes)

Literals:

- numbers entered as is. *e.g.*, 12 in decimal, and 0xC in hexadecimal
- characters enclosed in single quotes. *e.g.*, 'b'
- strings enclosed in double quotes. *e.g.*, "A string"

Registers

- We can manipulate 32 architectural registers in assembly programming directly.
- We prefer using aliases to indicate registers.
- Instructions category
  - Load and store instructions
  - Bitwise instructions
  - Arithmetic instructions
  - Control transfer instructions
  - Pseudo instructions

Register Names and Descriptions

Table: Register names and descriptions

Register Names	ABI Names	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary / Alternate link register
x6-7	t1 - t2	Temporary register
x8	s0 / fp	Saved register / Frame pointer
x9	s1	Saved register
x10-11	a0-a1	Function argument / Return value registers
x12-17	a2-a7	Function argument registers
x18-27	s2-s11	Saved registers
x28-31	t3-t6	Temporary registers

More Information

For more information about RISC-V instructions and assembly programing you can refer to:

1. Lecture slides and textbook.
2. **RARS** Help: F1
3. `https://github.com/riscv/riscv-asm-manual/blob/master/riscv-asm.md`
4. `https://web.eecs.utk.edu/~smarz1/courses/ece356/notes/assembly/`

System Call in RARS

System Calls in RARS I

RARS provides a small set of operating system-like services through the system call (`ecall`) instruction. Register contents are not affected by a system call, except for result registers in some instructions.

- Load the service number (or number) in register a7.
- Load argument values, if any, in a0, a1, a2 ..., as specified.
- Issue `ecall` instruction.
- Retrieve return values, if any, from result registers as specified.

System Calls in RARS II

Name	Number	Description	Inputs	Outputs
PrintInt	1	Prints an integer	a0 = integer to print	N/A
PrintFloat	2	Prints a float point number	fa0 = float to print	N/A
PrintString	4	Prints a null-terminated string to the console	a0 = the address of the string	N/A
ReadInt	5	Reads an int from input console	a0 = the int	N/A
ReadFloat	6	Reads a float from input console	fa0 = the float	N/A
ReadString	8	Reads a string from the console	a0 = address of input buffer, a1 = maximum number of characters to read	N/A
Open	1024	Opens a file from a path Only supported flags (a1), read-only (0), write-only (1) and write-append (9)	a0 = Null terminated string for the path, a1 = flags	a0 = the file decriptor or -1 if an error occurred
Read	63	Read from a file descriptor into a buffer	a0 = the file descriptor, a1 = address of the buffer, a2 = maximum length to read	a0 = the length read or -1 if error
Write	64	Write to a filedescriptor from a buffer	a0 = the file descriptor, a1 = the buffer address, a2 = the length to write	a0 = the number of charcters written
LSeek	62	Seek to a position in a file	a0 = the file descriptor, a1 = the offset for the base, a2 is the begining of the file (0), the current position (1), or the end of the file (2)}	a0 = the selected position from the beginning of the file or -1 is an error occurred

An Example of System Calls in RARS I

An example shows how to use system calls in RARS

Using system call

```
# Comment giving name of program and description
# sys-call.asm
# Bare-bones outline of RISC-V assembly language program
  .globl _start

.data
msg: .asciz "Hello,␣world!\n"

.text
_start:
li a7, 4     # system call code for PrintString
la a0, msg   # address of string to print
ecall        # Use the system call
# End of program, leave a blank line afterwards is preferred
```

You can check the output in Run/IO of the program information panel.

An Example of System Calls in RARS II

- *li* loads a register with an immediate value given in the instruction.
- *la* loads an address of the specified symbol.
- *.asciz* emits the specified string within double quotes and includes the terminated zero character at the end.

Lab 1-1

Lab1 Overview

We have 3 sub-labs for lab1.

- **Lab1**: RISC-V assembly language programming using **RARS** simulator.
- In lab1, we will practice coding in RISC-V assembly language, and understand how our codes run in a RISC-V CPU.
  - **Lab1-1**: basic operators and system call.
  - **Lab1-2**: function call and simple algorithm implementation.
  - **Lab1-3**: stack data structure, recursive function call, more complex algorithm implementation.

Lab1-1 Requirement

Write a RISC-V assembly program lab1-1.asm step by step:

1. Define three variables `var1`, `var2` and `var3` which will be loaded from **terminal** using syscall.
2. Increase `var1` by 5, multiply `var2` by 4.
3. increase `var3` by `var1` + `var2`.
4. print `var1`, `var2` and `var3` to **terminal** using syscall.

Example:

Input:
1
2
3
Output:
6
8
17

Some Tips

1. Variables should be declared following the `.data` identifier.
2. `<name>:  .<datatype> <value>`
3. Use `la` instruction to access the RAM address of declared data.
4. Use system call to read and print from the terminal.
5. Do not forget `\n`.
6. Do not forget exit system call.
7. You do not need to print "Input:" or "Output:" in the example in the previous page.

Submission

Submission Method:

- Submit the source codes and report **after finishing all the sub-labs** of Lab1.
- The submission window of Lab1 will be opened on **Blackboard**.
- The report template can be found on the homepage of CENG3420: `https://www.cse.cuhk.edu.hk/~byu/CENG3420/2025Spring/doc/lab1-report-template.pdf`

THANK YOU!