CENG3420

Lab 2-1: RISC-V Assembler

Chen BAI

Department of Computer Science and Engineering
The Chinese University of Hong Kong

cbai@cse.cuhk.edu.hk

Spring 2021

香 港 中 文 大 學
The Chinese University of Hong Kong

Overview

Overview

Assembler & Simulator

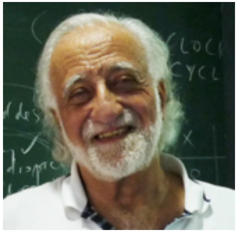
- Assembly language – symbolic (RISC-V, MIPS, ARM, X86, LC-3b, ...)
- Machine language – binary

- Assembler is a program that
  - turns symbols into machine instructions.
  - EX: lc3b_asm, riscv64-unknown-elf-as, ...

- Simulator is a program that
  - mimics the behavior of a processor
  - usually in high-level language
  - EX: lc3b_sim, spike, ...

LC-3b

- LC-3b: **Little Computer 3, b** version.
- Relatively simple instruction set
- Most used in teaching for CS & CE
- Developed by Yale Patt@UT & Sanjay J. Patel@UIUC



RISCV-LC

- RISC-V 32 general-purpose registers
- 32-bit data and address
- 28+ instructions (including pseudo instructions)

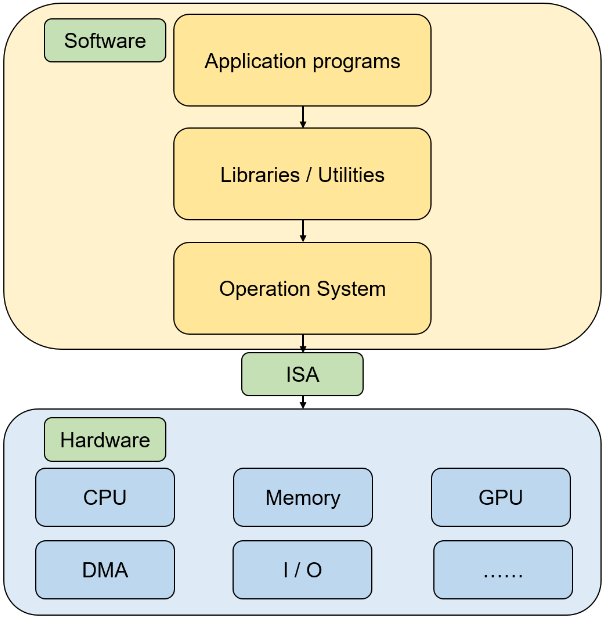
Plus 4 special-purpose registers:

- Program Counter (PC)
- Instruction Register (IR)
- Memory Access Register (MAR)
- Memory Data Register (MDR)

In order to make labs easy, I have modified some definitions of instructions, and they do not oberseve RISC-V specifications strictly!

Overview

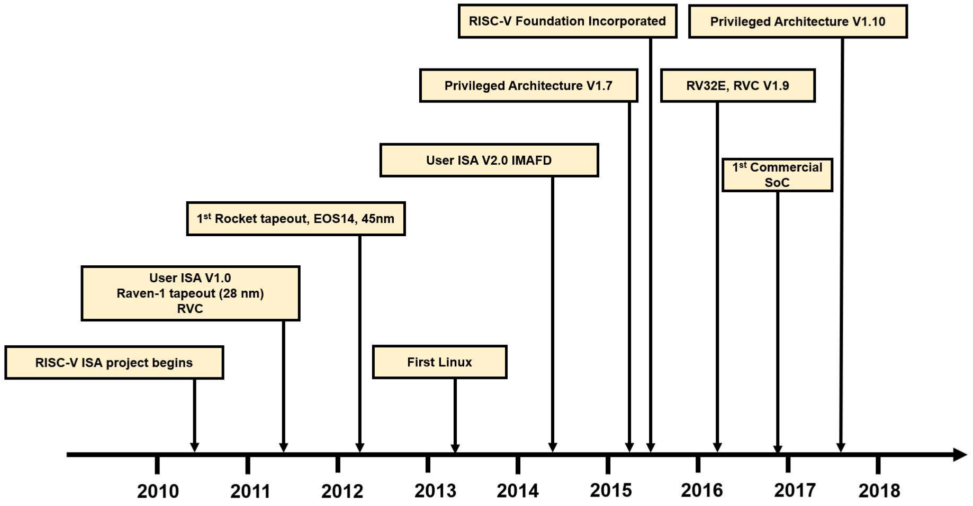
Computer architecture



RISC-V ISA

- An open standard instruction set architecture (ISA)
- A clean break from the earlier MIPS-inspired designs
- Modular ISA organization
- Open standards, numerous proprietary and open-source cores
- Managed by RISC-V Foundation

RISC-V ISA


RISC-V Foundation Incorporated
Privileged Architecture V1.10
Privileged Architecture V1.7
RV32E, RVC V1.9
User ISA V2.0 IMAFD
1st Commercial SoC
1st Rocket tapeout, EOS14, 45nm
User ISA V1.0
Raven-1 tapeout (28 nm)
RVC
RISC-V ISA project begins
First Linux
2010
2011
2012
2013
2014
2015
2016
2017
2018

RISC-V ISA

- Allow / Encourage custom extension
- Emphasize flexibility
- Standard extensions
  - I (Integer-related module)
  - M (Multiply and divide module)
  - A (Atomic-related module)
  - F (Floating point number calculation module)
  - D (Double point number calculation module)
  - C (Compressed module)
  - G (General purpose module, including IMAFD)
- 32-bit instruction encoding in G module, 16-bit instruction encoding in C module
- User / Supervisor / Machine level

RV32I

- Instruction encoding width is 32-bit
- Align on a four-byte boundary in memory
- Manipulate integer data
- Our lab is based on RV32I version 2.0

RV32I

- Four core instruction formats

31–25	24–20	19–15	14–12	11–7	6–0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[31:12]				rd	opcode	U-type

- Two variants of the instruction formats

31	30–25	24–21	20	19–15	14–12	11–8	7	6–0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

Why does the RISC-V specification add two variant instructions B/J ?

RV32I

- Integer Computational Instructions
- Control Transfer Instructions
- Load and Store Instructions
- Memory Odering Instructions
- Environment Call and Breakpoints
- HINT Instructions

Integer Computational Instructions

- Integer Register-Immediate Instructions
- Integer Register-Register Instructions
- NOP Instructions

Integer Register-Immediate Instructions

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM

31 12	11 7	6 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC

Integer Register-Register Instructions

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

NOP Instructions

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
0	0	ADDI	0	OP-IMM

Control Transfer Instructions

- Unconditional Jumps
- Conditional Branches

Unconditional Jumps

31	30 … 21	20	19 … 12	11 … 7	6 … 0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
1	10	1	8	5	7
offset[20:1]				dest	JAL

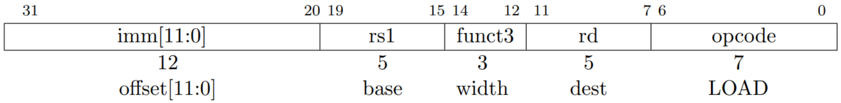
31 … 20	19 … 15	14 … 12	11 … 7	6 … 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	0	dest	JALR

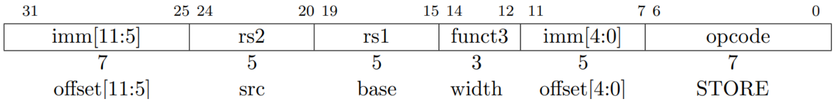
Conditional Branches

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12|10:5]		src2	src1	BEQ/BNE	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BLT[U]	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BGE[U]	offset[11|4:1]		BRANCH

The difference between RISC-V ISA and other ISA like MIPS, X86, ARM, SPARC?

Load and Store Instructions





The EEI will define whether the memory system is little-endian or big-endian. In RISC-V, endianness is byte-address invariant.

Memory Ordering Instructions

31 28	27	26	25	24	23	22	21	20	19 15	14 12	11 7	6 0
fm	PI	PO	PR	PW	SI	SO	SR	SW	rs1	funct3	rd	opcode
4	1	1	1	1	1	1	1	1	5	3	5	7
FM	predecessor				successor				0	FENCE	0	MISC-MEM

Environment Call and Breakpoints

31 20	19 15	14 12	11 7	6 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
ECALL	0	PRIV	0	SYSTEM
EBREAK	0	PRIV	0	SYSTEM

HINT Instructions

HINTs are encoded as integer computational instructions with rd=x0

Instruction	Constraints	Code Points	Purpose
LUI	*rd*=x0	2^{20}	*Reserved for future standard use*
AUIPC	*rd*=x0	2^{20}	
ADDI	*rd*=x0, and either *rs1*≠x0 or *imm*≠0	$2^{17}-1$	
ANDI	*rd*=x0	2^{17}	
ORI	*rd*=x0	2^{17}	
XORI	*rd*=x0	2^{17}	
ADD	*rd*=x0	2^{10}	
SUB	*rd*=x0	2^{10}	
AND	*rd*=x0	2^{10}	
OR	*rd*=x0	2^{10}	
XOR	*rd*=x0	2^{10}	
SLL	*rd*=x0	2^{10}	
SRL	*rd*=x0	2^{10}	
SRA	*rd*=x0	2^{10}	
FENCE	*pred*=0 or *succ*=0	$2^{5}-1$	
SLTI	*rd*=x0	2^{17}	*Reserved for custom use*
SLTIU	*rd*=x0	2^{17}	
SLLI	*rd*=x0	2^{10}	
SRLI	*rd*=x0	2^{10}	
SRAI	*rd*=x0	2^{10}	
SLT	*rd*=x0	2^{10}	
SLTU	*rd*=x0	2^{10}	

Overview

Specifications

In order to make labs easy, we have different requirements, and they do not oberseve RISC-V specifications strictly!

- Branch offset values are re-defined
- Shift offset values are re-defined
- Label specification: Do not support the colon and labels should be on the same line with RV32I instructions
- No .data and .text directives
- Add one pseudo instruction: LA
- Add one self-customized instruction: .FILL

Branch offset values

The address offset is encoded in the instruction directly!

- Unconditional jump

31	30 … 21	20	19 … 12	11 … 7	6 … 0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
1	10	1	8	5	7
offset[20:1]				dest	JAL

31 … 20	19 … 15	14 … 12	11 … 7	6 … 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	0	dest	JALR

- Conditional branches

31	30 … 25	24 … 20	19 … 15	14 … 12	11 … 8	7	6 … 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12|10:5]		src2	src1	BEQ/BNE	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BLT[U]	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BGE[U]	offset[11|4:1]		BRANCH

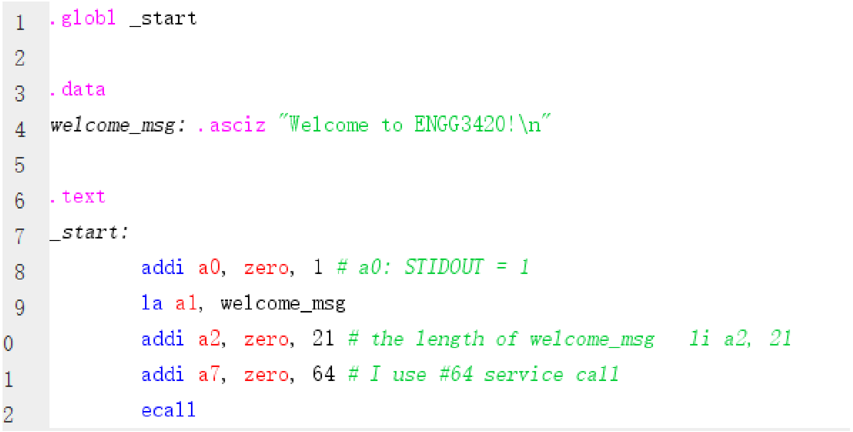
Shift offset values

The shift value is between 0 and $2^5 - 1$

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM

Label specification

Do not support the colon and labels should be on the same line with RV32I instructions



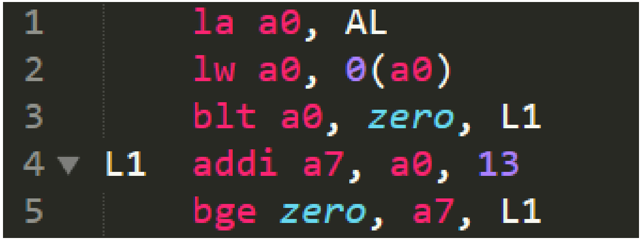
```
.globl _start

.data
welcome_msg: .asciz "Welcome to ENGG3420!\n"

.text
_start:
        addi a0, zero, 1 # a0: STIDOUT = 1
        la a1, welcome_msg
        addi a2, zero, 21 # the length of welcome_msg   li a2, 21
        addi a7, zero, 64 # I use #64 service call
        ecall
```

Label specification

Do not support the colon and labels should be on the same line with RV32I instructions

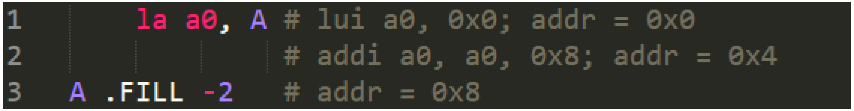


```
        la a0, AL
        lw a0, 0(a0)
        blt a0, zero, L1
L1      addi a7, a0, 13
        bge zero, a7, L1
```

The code snippet under our specifications

LA pseudo instruction

LA is translated to two RV32I instructions: lui and addi.

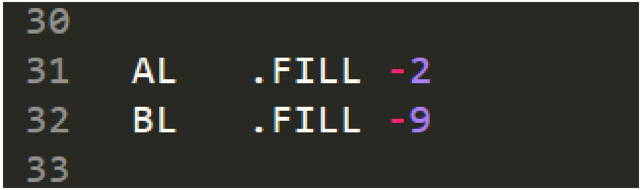


```
        la a0, A # lui a0, 0x0; addr = 0x0
                 # addi a0, a0, 0x8; addr = 0x4
A .FILL -2       # addr = 0x8
```

Translate la to lui and addi

FILL self-customized instruction

.FILL is similar to .byte, .word etc.



```
AL     .FILL -2
BL     .FILL -9
```

.FILL instruction

Overview

Lab2.1 assignment

A framework of the naive RV32I assembler (Released on Mar. 11)

- git clone https://github.com/baichen318/ceng3420.git
- git checkout lab2.1

Compile (Linux environment is suggested)

- make

Run the assembler

- ./asm benchmarks/isa.asm isa.bin # you can check the output machine code: isa.bin if you have implemented the assembler

Lab2.1 assignment

Finish the RV32I assembler including 26+ instructions as follows

- Pseudo instruction: la
- Integer Register-Immediate Instructions: slli, xori, srli, srai, ori, andi, lui
- Integer Register-Register Operations: sub, sll, xor, srl, sra, or, and
- Unconditional Jumps: jalr, jal
- Conditional Branches: bne, blt, bge
- Load and Store Instructions: lb, lh, lw, sb, sh, sw

These unimplemented codes are annotated with Lab2-1 assignment

Lab2.1 assignment

Verify your codes with benchmarks and the true binary machine codes suffixed with .bin

- isa.asm
- count10.asm
- swap.asm

Submission Method:

Submit the source code and report after the whole Lab2, onto blackboard.