

# CENG 3420

# Computer Organization & Design



## Lecture 03: Arithmetic Instructions

Bei Yu

CSE Department, CUHK

[byu@cse.cuhk.edu.hk](mailto:byu@cse.cuhk.edu.hk)

(Textbook: Chapters 2.1 – 2.7)

Spring 2023

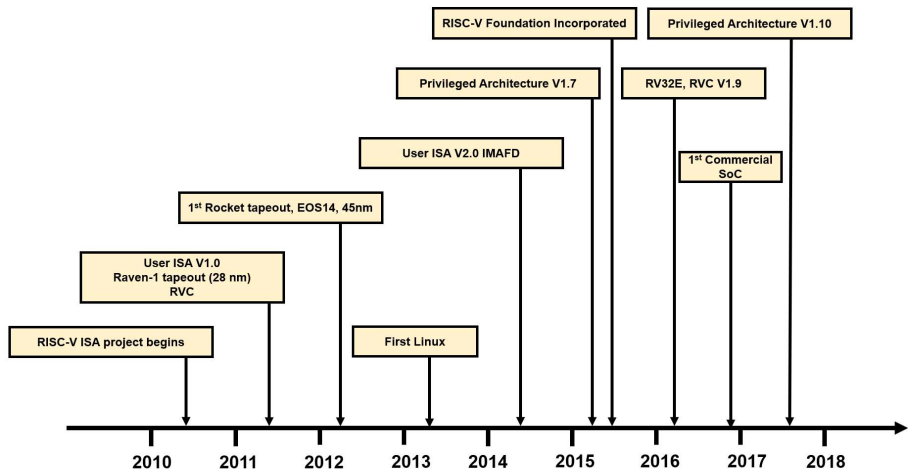


# Introduction



## RISC-V

- 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



## Specification of RISC-V

- Allow / Encourage custom extension
- Emphasize flexibility
- Standard extensions
  - I (Integer-related extension)
  - M (Standard integer multiply and divide extension)
  - A (Atomic extension)
  - F (Floating-point extension)
  - D (double-precision extension)
  - C (Compressed instruction extension)
  - G (General purpose extension, including IMAFD)
- G extension in RV32I encodes in 32-bit, C extension encodes in 16-bit
- User / Supervisor / Machine level

## Notice

Our Labs will focus on **RV32I**



Table: RV32I Unprivileged Integer Register

Register Name	ABI Name	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-x7	t1-t2	
x8	s0/fp	Saved Register (Frame Pointer)
x9	s1	Saved Register
x10-x11	a0-a1	Function Argument/Return Value Registers
x12-x17	a2-a7	
x18-x27	s2-s11	Saved Registers
x28-x31	t3-t6	Temporary Registers



## Stack pointer register

In RISC-V architecture, x2 register is use as Stack Pointer *sp0* and holds the base address of the stack.

Stack base address must aligne to 4-bytes, if not, a load / store alignment fault may arise.



## Global pointer register

Data is allocated to the memory when it is globally declared in an application. Using pc-relative or absolute addressing mode leads to utilization of extra instructions, thus increasing the code size.

In order to decrease the code size, RISC-V places all the global variables in a particular area which is pointed to, using the x3 *gp* register. The x3 register will hold the base address of the location where the global variables reside.





## Thread pointer register

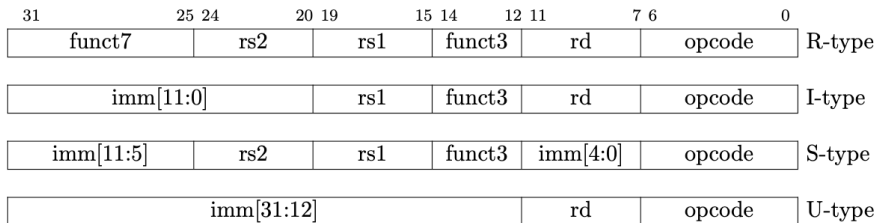
The x1 *ra* register is used to save the subroutine / function return addresses. Before a subroutine call is performed, x1 is explicitly set to the subroutine return address which is usually  $pc + 4$ .

The standard software calling convention uses x1 register to hold the return address on a function call.



## Argument register

In RISC-V, 8 argument registers, namely, x10 to x17 are used to pass arguments in a subroutine / function. Before a subroutine call is made, the arguments to the subroutine are copied to the argument registers. The stack is used in case the number of arguments exceeds 8.



**opcode** 6-bits, opcode that specifies the operation

**rs1** 5-bits, register file address of the first source operand

**rs2** 5-bits, register file address of the second source operand

**rd** 5-bits, register file address of the result's destination

**imm** 12-bits / 20-bits, immediate number field

**funct** 3-bits / 10-bits, function code augmenting the opcode



## Four RV32I Encodes

- Immediate Encoding Variants, *e.g., slti, addi, lui, and etc.*
- Integer Computational Instructions, *e.g., sll, sub, or, and etc.*
- Control Transfer Instructions, *e.g., jal, jalr, beq, and etc.*
- Load and Store Instructions, *e.g., lb, ld, sh, and etc.*

## Notice

We will be detailed in Lab 1-1



# Arithmetic & Logical Instructions



- RISC-V assembly language arithmetic statement

```
add    t0, a1, a2  
sub    t0, a1, a2
```

- Each arithmetic instruction performs **one** operation
- Each specifies exactly three operands that are all contained in the datapath's register file (t0, s1, s2)

```
destination = source1 op source2
```

- Instruction Format (**R** format)

0x0 / 0x40	0xc	0xb	0	0x5	0x33
------------	-----	-----	---	-----	------



- Small constants are often used in typical assembly code directly

## Possible approaches?

- put “typical constants” in memory and load them
- create hard-wired registers (like `zero`) for constants like 1
- have special instructions that contain constants

```
addi sp, sp, 4      # sp = sp + 4
slti t0, s2, 15     # t0 = 1 if s2 < 15
```

- Machine format (I format)
- The constant is kept inside the instruction itself!
- Immediate format limits values to the range  $-2^{11}$  to  $+2^{11} - 1$



```
1  .globl _start
2
3  .text
4  _start:
5      li a1, 20
6      li a2, 23
7      add t0, a1, a2
8      sub t1, a1, a2
```

RARS example:  $t0 = 0x2b$ ,  $t1 = 0xffffffff$





- We'd also like to be able to load a 32 bit constant into a register
  - For this we must use two instructions
- ① A new “load upper immediate” instruction (U-type format, load top 20bits)

```
lui t0, 1010101010101010
```

- ② Then must get the lower order bits right, use

```
ori t0, t0, 1010101010101010
```

## Aside: How About Larger Constants?



- We'd also like to be able to load a 32 bit constant into a register
  - For this we must use two instructions
- ① A new “load upper immediate” instruction (U-type format, load top 20bits)

```
lui t0, 1010101010101010
```

- ② Then must get the lower order bits right, use

```
ori t0, t0, 1010101010101010
```

1010101010101010	0000000000000000
------------------	------------------

0000000000000000	1010101010101010
------------------	------------------

---

1010101010101010	1010101010101010
------------------	------------------



- Need operations to **pack** and **unpack** 8-bit characters into 32-bit words
- Shifts move all the bits in a word left or right

```
slli t2, s0, 8    # t2 = s0 << 8 bits  
srli t2, s0, 8    # t2 = s0 >> 8 bits
```

- Instruction Format (**I** format)
- Such shifts are called **logical** because they fill with **zeros**
- Notice that a 5-bit shamt field is enough to shift a 32-bit value  $2^5 - 1$  or **31 bit positions**



```
1  .globl _start
2
3  .text
4  _start:
5      li a1, 20
6      li a2, 23
7      slli t0, a1, 2
8      srli t1, a2, 1
```

RARS example: t0 = 0x50, t1 = 0x0b



There are a number of **bit-wise** logical operations in the RISC-V ISA

## R Format

```
and t0, t1, t2      #  $t0 = t1 \& t2$   
or  t0, t1, t2      #  $t0 = t1 | t2$   
xor t0, t1, t2      #  $t0 = t1 \oplus t2$ 
```

## I Format

```
andi t0, t1, 0xFF00  #  $t0 = t1 \& 0xFF00$   
ori  t0, t1, 0xFF00  #  $t0 = t1 | 0xFF00$ 
```



```
1  .globl _start
2
3  .text
4  _start:
5      li a1, 20
6      li a2, 23
7      and t0, a1, a2
8      or t1, a1, a2
9      xor t2, a1, a2
10     andi t3, a1, 0x12
11     ori t4, a2, 0x21
```

RARS example: t0 = 0x14, t1 = 0x17, t2 = 0x03, t3 = 0x10, t4 = 37



# Data Transfer Instructions



- Two basic **data transfer** instructions for accessing memory

```
lw   t0, 4(s3)  # load word from memory  
sw   t0, 8(s3)  # store word to memory
```

- The data is loaded into (**lw**) or stored from (**sw**) a register in the register file – a 5 bit address
- The memory address – a 32 bit address – is formed by adding the contents of the base address register to the offset value
- A 12-bit field in RV32I meaning access is limited to memory locations within a region from -4 KB to 4 KB of the address in the base register





```
1  .globl _start
2
3  .data
4  a: .word 1 2 3 4 5 # .word = 32 bits
5
6  .text
7  _start:
8      la a1, a
9      lw t0, 0(a1)
10     lw t1, 4(a1)
11     lw t2, 8(a1)
12     lw t3, 12(a1)
13     lw t4, 16(a1)
14     addi t4, t4, 1
15     sw t4, 20(a1)
16     lw t5, 20(a1)
```

RARS example: t0 = 0x01, t1 = 0x02, t2 = 0x03, t3 = 0x04, t4 = 0x06, t5 = 0x06



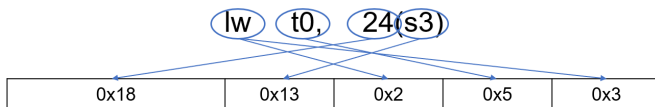
## Load/Store Instruction Format (I format):

lw t0, 24(s3)

0x18	0x13	0x2	0x5	0x3
------	------	-----	-----	-----

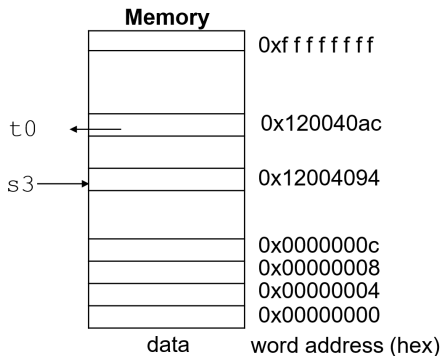


## Load/Store Instruction Format (I format):



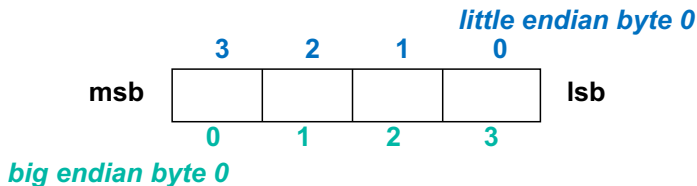
$$24_{10} + s3 =$$

$$\begin{array}{r} \dots 0001\ 1000 \\ + \dots 1001\ 0100 \\ \hline \dots 1010\ 1100 = \\ \quad 0x120040ac \end{array}$$





- Since 8-bit bytes are so useful, most architectures address individual **bytes** in memory
- **Alignment restriction** – the memory address of a word must be on natural word boundaries (a multiple of 4 in RV32I)
- **Big Endian**: leftmost byte is word address
  - IBM 360/370, Motorola 68k, **MIPS**, Sparc, HP PA
- **Little Endian**: rightmost byte is word address
  - RISC-V, Intel 80x86, DEC Vax, DEC Alpha (Windows NT)





### RISC-V provides special instructions to move bytes

```
lb    t0, 1(s3)    # load byte from memory  
sb    t0, 6(s3)    # store byte to memory
```

- What 8 bits get loaded and stored?
- Load byte places the byte from memory in the **rightmost** 8 bits to the destination register
- Store byte takes the byte from the **rightmost** 8 bits of a register and writes it to a byte in memory



## EX-1:

Given following code sequence and memory state:

```
add    s3, zero, zero
lb     t0, 1(s3)
sb     t0, 6(s3)
```

Memory	
0x 0 0 0 0 0 0 0 0	24
0x 0 0 0 0 0 0 0 0	20
0x 0 0 0 0 0 0 0 0	16
0x 1 0 0 0 0 0 1 0	12
0x 0 1 0 0 0 4 0 2	8
0x F F F F F F F F	4
0x 0 0 9 0 1 2 A 0	0
Data	Word Address (Decimal)

- 1 What value is left in `t0`?
- 2 What word is changed in Memory and to what?
- 3 What if the machine was **little Endian**?