CENG 3420
Computer Organization & Design
Lecture 03: Arithmetic Instructions

Textbook: Chapter 2.1-2.6

Zhengrong Wang

CSE Department, CUHK

zhengrongwang@cuhk.edu.hk

RISC-V Extensions

- RISC-V allows/encourage extension for 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, 16-bit instruction encoding)
  - G (General purpose extension, including IMAFD)
- User / Supervisor / Machine level

- In this course we focus on **RV32I**.

RV32I Unprivileged Integer Register

- Return address ra/x1
  - Before function calls, explicitly save the return address in ra (usually pc + 4).
- Stack pointer sp/x2
  - Points to the logical top of the stack (which is to the lowest memory address, since the stack grows downward).
- Global pointer gp/x3
  - Holds the base address of the memory region containing global variables.
- Function arguments a0-a7/x10-x17
  - Holds the arguments of a function call.
  - Extra arguments (>8) are pushed into the stack.

Register	ABI Name	Description	Saver
x0	zero	Zero constant	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	—
x3	gp	Global pointer	—
x4	tp	Thread pointer	Callee
x5	t0-t2	Temporaries	Caller
x8	s0 / fp	Saved / frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Fn args/return values	Caller
x12-x17	a2-a7	Fn args	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP args/return values	Caller
f12-17	fa2-7	FP args	Caller
f18-27	fs2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller

Instruction Encoding

31 27	26 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[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

- 6 base instruction formats: all **32-bit** wide:
  - opcode: 7-bit, specifies the operation.
  - rs1: 5-bit, register file address of the first source operand.
  - rs2: 5-bit, register file address of the second source operand.
  - rd: 5-bit, register file address of the destination.
  - imm: 12-bit or 20-bit, immediate number field.
  - funct: 3-bit or 10-bit, function code augmenting the opcode.
- B/S and U/J only differs in the immediate number encoding.

Arithmetic Instructions

- **R-type**: all three operands are from registers.

```
Assembly:     add x1, x2, x3
Hexadecimal:  003100B3
Binary:       0000 0000 0011 0001 0000    0000 1011 0011
Decode:       0000000  00011 00010 000    00001  0110011
Field:        funct7   rs2   rs1   funct3 rd     opcode
```

- Semantic: destination = source1 op source2

Intermediate Instructions

- Small constants are often used in typical assembly code directly.
- But all operands of arithmetic operations are registers.
- Possible approaches:
  - Put them in memory and load into registers.
  - Special registers to contain specific values (e.g., zero holds constant 0).
  - Directly encode constant number in the instruction as immediate.
- I-type instructions:

```
Assembly:     addi x1, x2, 100
Hexadecimal:  06410093
Binary:       0000 0110 0100  0001 0000    0000 1001 0011
Decode:       000001100100    00010 000    00001  0010011
Field:        imm 64+32+4=100 rs1   funct3 rd     opcode
```

Intermediate Instructions

- What if the constant is beyond the 12-bit range? E.g., 32-bit immediate?
- Two instructions:
- lui to load the upper 20-bit and zero lower 12-bit. (U-type)

```
Assembly:     lui x1, 0x12345
Hexadecimal:  123450b7
Binary:       0001 0010 0011  0100 0101 0000 1011 0111
Decode:       00010010001101000101      00001   011011
Field:        imm 0x12345               rd      opcode
```

- ori to set the lower 12-bit. (I-type)

```
Assembly:     ori x1, x1, 0x678
Hexadecimal:  6780e093
Binary:       0110 0111 1000  0000 1110    0000 1001 0011
Decode:       011001111000    00001 110    00001  0010011
Field:        imm 0x678       rs1   funct3 rd     opcode
```

Shift Instructions

- Logic shift left/right instruction.

sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]

  - Note the 5-bit shift amount can represent up to 2^5 – 1 = 31 bits positions.
  - Logical shift fills with zeros.

- Arithmetic shift right (sra, srai) will replicate the most-significant bit (MSB).
  - E.g., 0xE1234567 >> 4 = 0xFE123456 (Fill with ones).
  - E.g., 0x71234567 >> 4 = 0x07123456 (Fill with zeros).
  - This is to keep the sign of signed integer type (e.g., int).
  - Arithmetic shift left always fill zeros (there is no sla/slai instruction).

Bitwise Logical Instructions

- Bitwise and/or/xor instruction.

xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2

- How to do negate (~x)?
  - The `not` instruction is a pseudo-instruction, not a real instruction.
  - So, what is this `not` instruction?

```
int negate(int v) {
    return ~v;
}

negate(int):
    not     a0,a0
    ret
```

```
Assembly:     not a0, a0
Hexadecimal:  fff54513
Binary: 1111 1111 1111 0101 0100 0101 0001 0011
Decode:   111111111111 01010 100 01010  0010011
Real Inst:    xori a0, a0, 0xFFF (Sign extended to 0xFFFFFFFF)
```

Data Transfer Instructions

- Bitwise and/or/xor instruction.

inst	operation	encoding	opcode	funct3	description (C)
lw	load word (32-bit)	I-type	0000011	0x2	rd = M[rs1+imm][0:31]
sw	store word (32-bit)	S-type	0100011	0x2	M[rs1+imm][0:31] = rs2

  - 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 −2 KB to 2 KB of the address in the base register.

```
Assembly:     lw x1, 8(x2)                                  Suppose x2 = 0x100
Hexadecimal:  00812083                                      Load Mem[0x108] into x1
Binary:       0000 0000 1000  0001 0010 0000 1000 0011
Decode:       000000001000    00010 010    00001  0000011
Field:        imm 8           rs1   funct3 rd     opcode
```

Byte Address

- 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: most significant byte placed in lower address.
  - IBM 360/370, MIPS, SPARC.
- Little endian: least significant byte placed in lower address.
  - RISC-V, x86, ARM (default little endian, can be configured as big endian).

```
Memory Address:                    | 0x100 | 0x101 | 0x102 | 0x103 |
Little endian (value=0x01234567)   | 0x67  | 0x45  | 0x23  | 0x01  |
Bit endian    (value=0x01234567)   | 0x01  | 0x23  | 0x45  | 0x67  |
```

- Why modern architectures prefer little endian?
  - Easier for type casting – e.g., int* → char* -- does not need to shift the address.

Byte Transfer Instruction

- RISC-V provides lb/sb instruction to directly load/store one byte.
- Into the lower 8-bit of rd, leaving other bits unchanged.

- Example: Given the following code sequence and memory state:
  - What is the content of t0 after the load?
  - What word in memory is changed? To what value?
  - What if the system is big-endian?

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

```
Memory:
0x8:   01 00 04 02
0x4:   FF FF FF FF
0x0:   00 90 12 A0
```