RISC-V ISA

Arithmetic Instructions: Add and compare

Sparsh Mittal

This presentation assumes that learner is familiar with 2's complement representation.

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c  # a gets b + c
```

- All arithmetic operations have this form
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Adding 4 operands?

- Each RISC-V arithmetic instruction performs only one operation and must always have exactly three variables.
- If want to place the sum of four variables b, c, d, and e into variable a.

```
add a, b, c     # The sum of b and c is placed in a
add a, a, d     # The sum of b, c, and d is now in a
add a, a, e     # Sum of b, c, d, and e is now in a
```

Add: overflow are ignored
Sub: overflow and borrow are ignored

Finding whether overflow happened

```
li a0 0x80000000
li a1 0x80000000
add a2 a0 a1
sltu a3 a2 a0
```

Conditions for overflow

A	B	Result
>=0	>=0	<0
<0	<0	>=0

If a3 is 1, then overflow has happened. Else, no overflow

Example on add and sub: all operands in register

- C code:

```
f = (g + h) - (i + j);
```

- Mapping of variables to registers:
- f→ x19, g→ x20, h→ x21 i → x22 j→ x23

- Compiled RISC-V code:

```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

Note that all the 3 operands of ‘add’ and ‘sub’ are stored in the registers

Example on add and sub: one operand is immediate and other is in register

- We can have one of the operands as a constant (called immediate), e.g.,

```
addi x22, x22, 4
```

- This can be used to implement NOP, a pseudo instruction. Below two instructions are equivalent:

```
addi x0, x0, 0
NOP
```

Example on add

- Mapping of variables to registers:
- i→ x19, g→ x20

C codes	C code (simplified)	RISC-V code
i++	i++	addi x19, x19, 1
g = ++i	i++ g=i	addi x19, x19, 1 add x20, x19, x0
g = i++	g=i i++	add x20, x19, x0 addi x19, x19, 1

LI and MV (pseudo) instructions

- Load Immediate (LI) loads register *rd* with an immediate value
- Syntax: li rd, imm
- li t0, 0x4A # Load register t0 with a value
- mv t1, t0 # Copy contents of register t0 to register t1
- Pseudo instructions are similar to macros in C/C++.

Assuming v and r are stored in t0 and t1, respectively.

C codes	RISC-V code
v= 10	li t0, 10
v=r	mv t0, t1

Comparison instructions

Signed vs. Unsigned

- A number stored in a register or memory address can be interpreted as signed (2's complement) or unsigned. Both may lead to different decimal numbers.

  - x22 = 1111 1111 1111 1111 = 65535 (unsigned), -1 (signed)
  - x23 = 0000 0000 0000 0001 = 1 (signed), 1 (unsigned)

  - On comparing them, result depends on whether you take them as signed or unsigned.
  - On signed comparison, x22<x23 because -1 < +1
  - On unsigned comparison, x22 > x23 because +65535 > +1

Signed and unsigned comparison

- To take this into account, RISC-V has two variants of comparison/branching instructions.
- Signed comparison: blt, bge
- Unsigned comparison: bltu, bgeu

SLT (signed comparison) and SLTU (unsigned comparison)

Syntax: slt rd, *rs*1, *rs*2

SLT writes 1 to rd if *rs*1 < *rs*2, 0 otherwise.

Example code:

```
li x5, 39          # x5 ← 39
li x3, 57          # x3 ← 57
slt x1, x5, x3     # x1 ← x5 < x3. So, x1 will store 1
```

Assuming a/b/c are stored in a0/a1/a2 respectively.

C code	RISC-V code

if (a<b)
c=1;
else
c=0;
	slt a2, a0, a1

Comparison Instructions (SLT and SLTU)

SLT and SLTU perform signed and unsigned compares respectively, writing 1 to rd if rs1 < rs2, 0 otherwise.

```
li a0, -4                    # a0 ← 0xFFFFFFFC (in hex)
li a1,  3                    # a1 ← 3
slt a2, a0, a1               # a2 ← 1 because -4<3 is true
sltu a3, a0, a1              # a3 ← 0 because 0xFFFFFFFC <3 is false
```

```
li a0, 4                     # a0 ← 4
li a1, -3                    # a1 ← 0xFFFFFFFD
slt a4, a0, a1               # a4 ← 0 because 4<-3 is false
sltu a5, a0, a1              # a5 ← 1 because 0x4< 0xFFFFFFFD is true
```

Variants of SLT (Pseudo instruction)

SEQZ: Set If Equal to Zero . Syntax: seqz *rd*, *rs*1

Example: seqz x6, x5

If x5 was zero, then x6 is set to 1. Otherwise, x6 is set to 0.

Similar instructions:

SGTZ: Set If Greater Than Zero

SLTZ: Set If Less Than Zero

SNEZ: Set If Not Equal to Zero

Assuming a/b/c are stored in a0/a1/a2 respectively.

C code	RISC-V code

if (a==0)
c=1;
else
c=0;
	seqz a2, a0

Comparison Instructions (SLTZ, SGTZ)

```
li a0, 3          # a0 ← 3
li a1, -2         # a1 ← 0xFFFFFFFE
sltz a2, a0       # a2 ← 0 because 3<0 is false
sltz a3, a1       # a3 ← 1 because -2<0 is true
sgtz a4, a0       # a4 ← 1 because 3>0 is true
sgtz a5, a1       # a5 ← 0 because -2>0 is false

li a1, 0              # a1 ← 0
snez a6, a0       # a6 ← 1 because ‘a0 not equal to 0’ is true
snez a7, a1       # a7 ← 0 because ‘a1 is equal to 0’ is false
```

Solved Question

Add two long 64-bit values stored in <t1; t0> and <t3; t2>. Store the result in <t5; t4>. Assume that the registers contain the 64-bit values already.

- # Goal: <t5 ,t4 > = <t1 ,t0 > + <t3 ,t2 >
- add t4 , t0 , t2 # add lower 32 bits
- add t5 , t1 , t3 # add upper 32 bits
- sltu t6 , t4 , t0 # t6 stores the carry (if any) from adding lower 32-bits.
- add t5 , t5 , t6 # add the carry

RV32I

Integer Computation

- **add** { – / immediate }
- **sub**tract
- { **and** / **or** / exclusive **or** } { – / immediate }
- { shift **l**eft **l**ogical / shift **r**ight **a**rithmetic / shift **r**ight **l**ogical } { – / immediate }
- **l**oad **u**pper **i**mmediate
- **a**dd **u**pper **i**mmediate to **pc**
- **s**et **l**ess **t**han { – / immediate } { – / **u**nsigned }

Control transfer

- **b**ranch { **e**qual / **n**ot **e**qual }
- **b**ranch { **g**reater than or **e**qual / **l**ess **t**han } { – / **u**nsigned }
- **j**ump **a**nd **l**ink { – / **r**egister }

Loads and Stores

- { **l**oad / **s**tore } { **b**yte / **h**alfword / **w**ord }
- **l**oad { **b**yte / **h**alfword } **u**nsigned

Miscellaneous instructions

- **fence** loads & stores
- **fence**.instruction & data
- **e**nvironment { **break** / **call** }
- **c**ontrol **s**tatus **r**egister { **r**ead & **c**lear bit / **r**ead & **s**et bit / **r**ead & **w**rite } { – / **i**mmediate }

Instruction	Type	Example	Meaning
Add	R	`add rd, rs1, rs2`	`R[rd] = R[rs1] + R[rs2]`
Subtract	R	`sub rd, rs1, rs2`	`R[rd] = R[rs1] – R[rs2]`
Add immediate	I	`addi rd, rs1, imm12`	`R[rd] = R[rs1] + SignExt(imm12)`
Set less than	R	`slt rd, rs1, rs2`	`R[rd] = (R[rs1] < R[rs2])? 1 : 0`
Set less than immediate	I	`slti rd, rs1, imm12`	`R[rd] = (R[rs1] < SignExt(imm12))? 1 : 0`
Set less than unsigned	R	`sltu rd, rs1, rs2`	`R[rd] = (R[rs1] <u R[rs2])? 1 : 0`
Set less than immediate unsigned	I	`sltiu rd, rs1, imm12`	`R[rd] = (R[rs1] <u SignExt(imm12))? 1 : 0`
Load upper immediate	U	`lui rd, imm20`	`R[rd] = SignExt(imm20 << 12)`
Add upper immediate to PC	U	`auipc rd, imm20`	`R[rd] = PC + SignExt(imm20 << 12)`