RISC-V ISA

Load/Store Instructions

Sparsh Mittal

Solved question

- Is it a correct RISC-V instruction?
- `sub x5, x6, 10(x7)`
- Answer: No. In RISC-V, only load and store instructions can access the memory directly. For all other instructions, all their operands have to be register values.
- Another way to say: RISC-V is a load-store ISA.
- Correct way to write above instruction is
- `ld x9, 10(x7)`
- `sub x5, x6, x9`

Why zero-extension or sign-extension

- Remember: Memory is byte addressed
  - Each address identifies a single byte (8b).
  - A load from memory or store to memory is in multiple of bytes.
- Assume we are moving data from memory to a register
- In RV64, register is 64b.
- On loading 8b (or 16b or 32b) from memory, how to fill remaining bits?
- Answer: Do either zero-extension or sign-extension.

Load/store instructions in RV64

Command	Full-form	#Bits loaded	Extension?	#bits extended?
ld	Load double-word	64	Not required	N/A
lw	Load word	32	Sign-extension	32
lwu	Load word unsigned	32	Zero-extension	32
lh	Load half-word	16	Sign-extension	48
lhu	Load half-word unsigned	16	Zero-extension	48
lb	Load byte	8	Sign-extension	56
lbu	Load byte unsigned	8	Zero-extension	56

Syntax: `lwu rd, offset(rs1)`

Example: lwu x6, 40(x21)

Command	Full-form	Bits stored in memory
sd	Store double-word	64
sw	Store word	32
sh	Store half-word	16
sb	Store byte	8

Syntax: `sw rs2, offset(rs1)`

Example: sw x7, 40(x21)

Solved example on load-instructions

MEMORY

Byte addresses	
1011	
1012	68
1013	24
1014	E0
1015	AC
1016	09
1017	EF
1018	CD
1019	AB
1020	78
1021	56
1022	34
1023	12
1024	5A
1025	4B
1026	7C

RISC-V is little endian

Assume RV64.

Let x20= 1016

Show the result of

(i) ld x19, 0(x20)

(ii) lw x19, 0(x20)

(iii) lwu x19, 0(x20)

(iv) lh x19, 0(x20)

(v) lhu x19, 0(x20)

(vi) lb x19, 0(x20)

(vii) lbu x19, 0(x20)

Solved example on load-instructions

MEMORY

Byte addresses	
1011	
1012	68
1013	24
1014	E0
1015	AC
1016	09
1017	EF
1018	CD
1019	AB
1020	78
1021	56
1022	34
1023	12
1024	5A
1025	4B
1026	7C

RISC-V is little endian

Assume RV64.
Let x20= 1016
Show the result of
(i) ld x19, 0(x20)
(ii) lw x19, 0(x20)
(iii) lwu x19, 0(x20)
(iv) lh x19, 0(x20)
(v) lhu x19, 0(x20)
(vi) lb x19, 0(x20)
(vii) lbu x19, 0(x20)

Value of x19 after executing instructions

12	34	56	78	AB	CD	EF	09
FF	FF	FF	FF	AB	CD	EF	09
00	00	00	00	AB	CD	EF	09
FF	FF	FF	FF	FF	FF	EF	09
00	00	00	00	00	00	EF	09
00	00	00	00	00	00	00	09
00	00	00	00	00	00	00	09

Selected explanations:
(ii) For A=1010, the sign-bit is 1. Hence, sign-extension of 11…
(iv) For E=1110, the sign-bit is 1. Hence, sign-extension of 11…
(vi) For 0 in 09, the sign-bit is 0. Hence, sign-extension of 00…

Do we need sign/zero extension while storing also?

- Answer: No
- Reason:
- Memory is byte addressed. Register is 64b.
- If we store entire 64b register content to memory, it would take 8 bytes
- If we store 32b register content to memory, it would take 4 bytes
- If we store 16b register content to memory, it would take 2 bytes
- If we store 8b register content to memory, it would take 1 byte
- Summary: extension is required only when storing lower-width data to higher-width storage location. Register is 8B (64b), whereas memory is byte addressable (1B granularity).

Solved example on store-instructions

Initial value of x19

12	34	56	78	AB	CD	EF	09
MSB							LSB

Assume RV64.
Let x20= 1016

Show the result of
(i) sd x19, 0(x20)
(ii) sw x19, 0(x20)
(iii) sh x19, 0(x20)
(iv) sb x19, 0(x20)

RISC-V is little endian

Solved example on store-instructions

x20= 1016

MSB				Initial value of x19			LSB
12	34	56	78	AB	CD	EF	09

sd x19, 0(x20)

1012	
1013	
1014	
1015	
1016	09
1017	EF
1018	CD
1019	AB
1020	78
1021	56
1022	34
1023	12
1024	
1025	

sw x19, 0(x20)

1012	
1013	
1014	
1015	
1016	09
1017	EF
1018	CD
1019	AB
1020	
1021	
1022	
1023	
1024	
1025	

sh x19, 0(x20)

1012	
1013	
1014	
1015	
1016	09
1017	EF
1018	
1019	
1020	
1021	
1022	
1023	
1024	
1025	

sb x19, 0(x20)

1012	
1013	
1014	
1015	
1016	09
1017	
1018	
1019	
1020	
1021	
1022	
1023	
1024	
1025	

Load/store Instruction Examples

```
li a0, 305419912          # data (0x12345688)
li a1, 0x12000000         # address
sw a0, 0(a1)              # store at addr.
lw a2, 0(a1)              # a2 <- 0x12345688
lh a3,0(a1)               # a3 <- 0x00005688
lhu a4,0(a1)              # a4 <- 0x00005688
lb a5,0(a1)               # a5 <- 0xffffff88
lbu a6,0(a1)              # a6 <- 0x00000088
```

```
li a0, 305419896          # data (0x12345678)
li a1, 0x12000000         # address
sw a0, 0(a1)              # store at addr.
lw a2, 0(a1)              # a2 <- 0x12348678
lh a3,0(a1)               # a3 <- 0xffff8678
lhu a4,0(a1)              # a4 <- 0x00008678
lb a5,0(a1)               # a5 <- 0x00000078
lbu a6,0(a1)              # a6 <- 0x00000078
```

Solved example

1. A register x19 in RV64 stores the value AB BC CD DE EF F1 12 23. Let x20= 1024. Show the memory state after implementing
   (i) sd x19, 0(x20)
   (ii) sh x19, 0(x20)

Byte Address:	Contents:
1024	23
1025	12
1026	F1
1027	EF
1028	DE
1029	CD
1030	BC
1031	AB

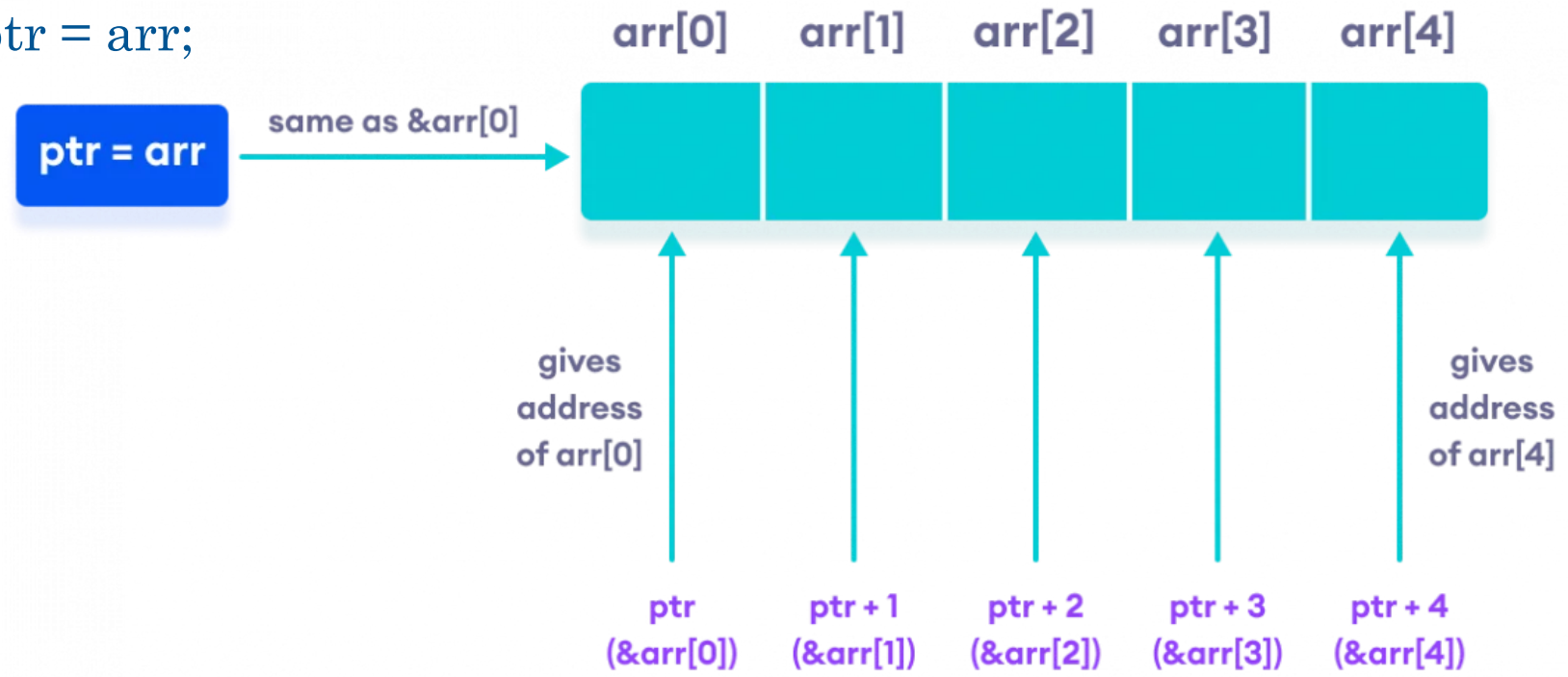
Byte Address:	Contents:
1024	23
1025	12

Addresses and pointers in C/C++

(This will be useful for doing load-store on array operands in RISC-V)

Understanding pointers

```
int *ptr;
int arr[5];
ptr = arr;
```



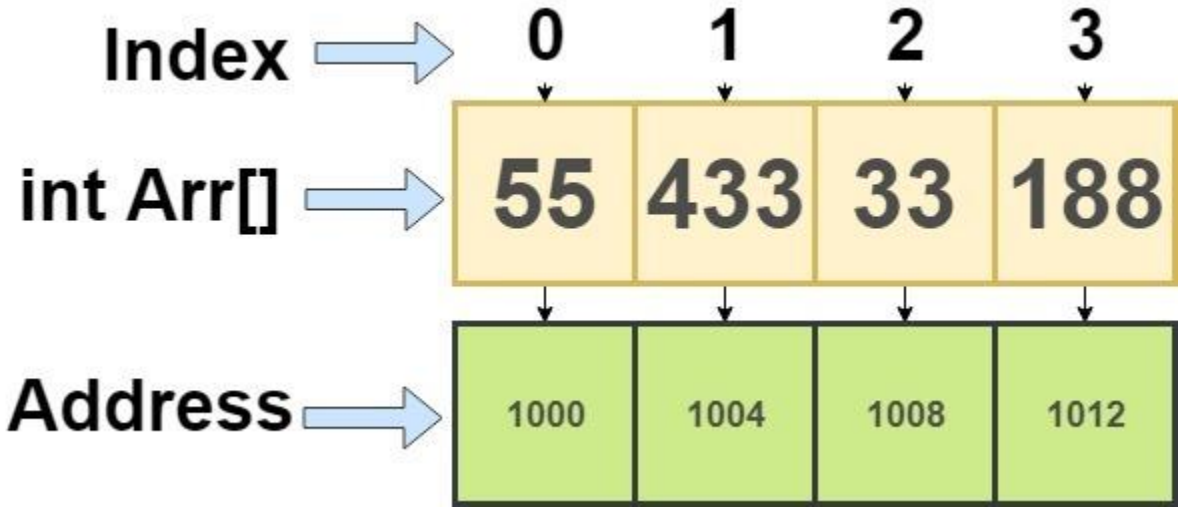
The address between ptr and ptr + 1 differs by 4 bytes.
Reason: ptr points to int, which is 4B

Displaying values

- Displaying address using arrays:
- &arr[0] = 0x61fef0
- &arr[1] = 0x61fef4
- &arr[2] = 0x61fef8

- Displaying address using pointers:
- ptr + 0 = 0x61fef0
- ptr + 1 = 0x61fef4
- ptr + 2 = 0x61fef8

Finding address of variable



Base address of Arr is 1000
Arr[2] starts from 1008

Question: Let the base address of an array Arr be BaseAddress. The size of each variable is K bytes, find the address of the element Arr[i]
Answer: BaseAddress+K*i

Solved example

Assume “long int” variables (8B). Base address of Arr is in a0. i is in a1. val is in a2

C code	Simplified C code	RISC-V code
Arr[i] =5;	temp=5 offset = i*8 addr = Arr+offset Memory[addr] = temp	li a3, 5 slli a4, a1, 3 add a5, a4, a0 sd a3, 0(a5)
val = Arr[8]		ld a2, 64(a0)

Arr[i]++	offset = i*8
	addr = Arr+offset
	temp = Memory[addr]
	temp++;
	Memory[addr] = temp
		slli a4, a1, 3
		add a5, a4, a0
		ld a6, 0(a5)
		addi a6, a6, 1
		sd a6, 0(a5)

Common mistake: ld a2, a3(a4)

→ This offset has to be an immediate, not a register value

Solved example

Convert *Aptr = *Bptr into RISC-V.
Aptr in a0, Bptr in a1. 4B variables

Answer:
Above is same as Aptr[0] = Bptr[0]

```
lw a2, 0(a1)
sw a2, 0(a0)
```

Solved example: Convert array accesses

- C code:

```
A[12] = h + A[7];
```

  - h in x21, base address of A in x22

- Corresponding RISC-V code: Every doubleword has 8B, hence, index 7 requires offset of 7*8(=56). Index 12 requires offset of 12*8(=96).

C/C++ code (with simple operations)

```
addr = A+7*8; val = Memory[addr]
temp = val +h;
addr = A+12*8; Memory[addr] = temp
```

RISC-V code

```
ld   x9, 56(x22)
add  x9, x21, x9
sd   x9, 96(x22)
```

Solved example

- Convert below code to RISC-V (RV32). The base address of A is in x22.

A[11] = A[5]*A[6];

Solution:

```
lw x9, 20(x22)
lw x10, 24(x22)
mul x9, x9, x10
sw x9, 44(x22)
```

Solved example: Create a copy of a 10-element array

The starting address of the original array is stored in a1

The starting address of the destination array is stored in a2.

```
. main :
        li t1 , 0        # counter (t1) = 0
        li t2 , 10       # number of iterations
. loop :
        lw t0 , 0( a1)  # load an element from the source array
        sw t0 , 0( a2)  # store an element in the destination
array
        addi a1 , a1 , 4 # get address of next element : src array
        addi a2 , a2 , 4 # do the same for destination array
        addi t1 , t1 , 1 # increment the counter
        bne t1 , t2 , .loop # loop back
```

Instruction	Type	Example	Meaning
Load doubleword	I	`ld rd, imm12(rs1)`	R[rd] = Mem_8[R[rs1] + SignExt(imm12)]
Load word	I	`lw rd, imm12(rs1)`	R[rd] = SignExt(Mem_4[R[rs1] + SignExt(imm12)])
Load halfword	I	`lh rd, imm12(rs1)`	R[rd] = SignExt(Mem_2[R[rs1] + SignExt(imm12)])
Load byte	I	`lb rd, imm12(rs1)`	R[rd] = SignExt(Mem_1[R[rs1] + SignExt(imm12)])
Load word unsigned	I	`lwu rd, imm12(rs1)`	R[rd] = ZeroExt(Mem_4[R[rs1] + SignExt(imm12)])
Load halfword unsigned	I	`lhu rd, imm12(rs1)`	R[rd] = ZeroExt(Mem_2[R[rs1] + SignExt(imm12)])
Load byte unsigned	I	`lbu rd, imm12(rs1)`	R[rd] = ZeroExt(Mem_1[R[rs1] + SignExt(imm12)])
Store doubleword	S	`sd rs2, imm12(rs1)`	Mem_8[R[rs1] + SignExt(imm12)] = R[rs2]
Store word	S	`sw rs2, imm12(rs1)`	Mem_4[R[rs1] + SignExt(imm12)] = R[rs2](31:0)
Store halfword	S	`sh rs2, imm12(rs1)`	Mem_2[R[rs1] + SignExt(imm12)] = R[rs2](15:0)
Store byte	S	`sb rs2, imm12(rs1)`	Mem_1[R[rs1] + SignExt(imm12)] = R[rs2](7:0)

Courtesy: Jin-Soo Kim

Load address (LA) instruction

The Load Address (LA) loads the location address of the specified SYMBOL.

Syntax: la rd, SYMBOL

Example:

```
.data
NumElements: .byte 6
.text
la x5, NumElements          #x5 ← addr[NumElements]
```

If 'NumElements' has a location address '10010074', then after above instruction, 10010074' is loaded into register x5.

Solved example on “la”

- Dene a constant val that is initialized to 17. Store its value in register s0 after loading it from memory.

```
val: .word 17

la t1 , val
lw s0 , 0( t1)
```