Week3 RISCV Assembly

Thursday, March 17, 2022 6:15 PM

1. Registers:

32 registers in RISCV, for 32bit RISCV, width is 32-bits

- Registers numbered from **x0 - x31**
- **x0** is a special register **because it always holds 0**

2. Assembly

2.1. Addition and subtraction

```
a = b + c + d - e;
x10  x11   x12  x13   x14
```

```
1 add x10, x11, x12   # temp = b + c
2 add x10, x10, x13   # temp = temp + d
3 sub x10, x10, x14   # a = temp - e
```

```
f = (g + h) - (i + j);
x10  x11  x12   x13  x14
```

```
1 add x5, x11, x12    # a_temp = g + h
2 add x6, x13, x14    # b_temp = i + j
3 sub x10, x5, x6     # f = a_temp - b_temp
```

2.1.a. Register x0

- As x0 always holds zero, and **does not** require initialization.
- One usage could be move operation

```
f = g           (in C)
add x3,x4,x0  (in RISC-V)
```

2.1.b. Immediate values

- Immediates are used to provide numerical constants

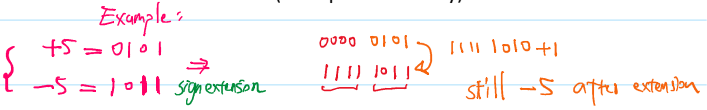
E.g. Add immediate:

```
f = g - 10          (in C)
addi x3,x4,-10    (in RISC-V)
```

- **No subtract** immediate in RISCV as we can achieve **subtraction** by add a **negative** number
- ***addi*** immediate are **limited to 12 bits**
  - Immediate is sign extended to 32-bits

2.1.b.note: **Sign extension** creates the **same** value

- Self-explanatory for positive numbers
- For negative numbers:
  - Recall $-x = \overline{x} + 1$ (x is a positive binary)



2.2 Memory Operations: Pointer Arithmetic to Assembly

2.2.1.a Memory Addressing

- Memory is **Byte addressed**
- For n bytes, $\log_2 n$ bits are required for addressing

2.2.1.b Pointer Arithmetic and Offsets

Problem: accessing the third element of an array pointed by **ptr**

- `ptr[3]`
- `*(ptr + 3)`

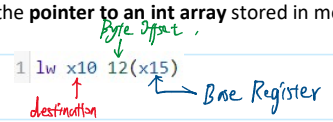
In RISCV: Do the pointer arithmetic **manually**

Example: ***ptr*** points to an int array, then :

```
offset = 3 * sizeof(int) = 3 * 4
```

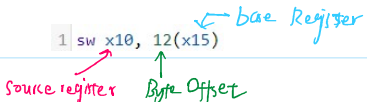
2.2.2.a. **Load data** from memory into Processor Registers: Load Words (LW)

Register **x15** contains the **pointer to an int array** stored in memory, **load** the value located at **arr[3] into x10**



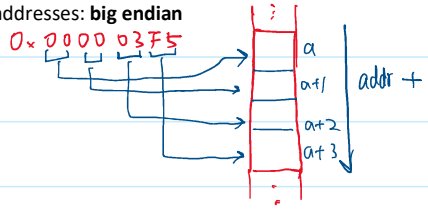
2.2.2.b. **Store data** from register to memory: Store Word (SW)

Register x15 contains the **pointer to an int array** stored in memory, **store** the value at x10 **into** arr[3]

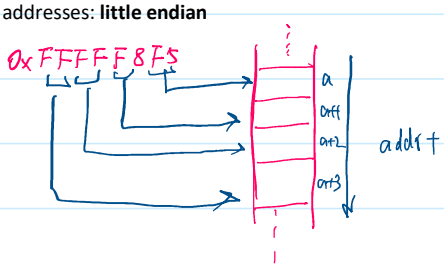


2.2.Notes: **Endianness**

- Decreasing numerical significance with increasing memory addresses: **big endian**



- Increasing numerical significance with Increasing memory addresses: **little endian**



into arr[3]

```
1 sw x10, 12(x15)
```

Source register, Byte Offset, base Register

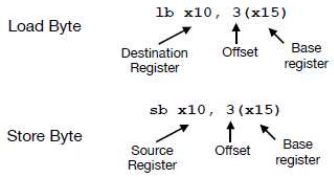
2.2.2.conclusion. LW/SW Example:

Register x15 contains the **pointer to an int array** stored in memory, **adding** the value located at **arr[3] by 4**

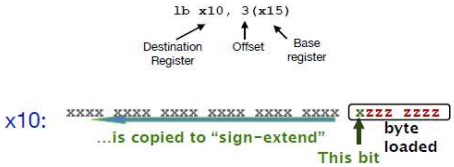
```
1 lw x10 12(x15)
2 addi x10, x10, 4
3 sw x10 12(x15)
```

1. **Load value from memory** into register
2. **ALU** handles adding operation
3. **Store** added value **back to memory**

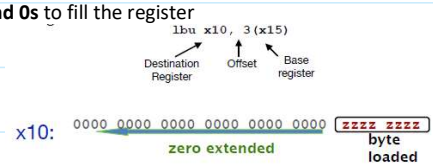
2.2.3. L/S only a part of memory entry: **Load** and **Store bytes**



- When **load byte**, it is **placed to the lowest byte** of the destination register and **sign extended** to 32 bits



- An alternative: ***lbu*** (load byte unsigned) will **not** perform sign extension. Instead, it will **extend 0s** to fill the register



- When **storing the byte**, only the lower 8 bit of the register is copied to memory, so there is **no sign-extension** for ***sb*** operations

2.2.3. Example: Sign Extension Examples

Q: What is the value of x12 after code runs

```
addi x11,x0,0x3F5
sw x11,0(x5)
lb x12,1(x5)
```

x11 = 0x3F5 = 0011 1111 0101
mem[x5] = 0011 1111 0101
x12 = 0s 0011 (extended)

```
addi x11,x0,0x3F5
sw x11,0(x5)
lb x12,0(x5)
```

x12 = 1...1 1111 0101 (extended)

```
addi x11,x0,0x8F5
sw x11,0(x5)
lb x12,1(x5)
```

x11 = 1...1 1000 1111 0101 (extend)
x12 = 1...1 1111 1000

2.3 RISCV logical Instructions

Logical operations	C operators	Java operators	RISC-V instructions
Bitwise AND	&	&	**and**
Bitwise OR	|	|	**or**
Bitwise XOR	^	^	**xor**
Shift left logical	<<	<<	**sll**
Shift right	>>	>>	**srl/sra**

2.3.1 Shifting

- Shift by the **contents of a register**

```
1 sll x10, x11, x12 # x10 = x11 << x12
```

```
sll x10, x11, x12 # x10 = x11 << x12
```

- Shift by a constant value (**immediate** value)

```
slli x10, x11, 2 # x10 = x11 << 2
```

- **Logical** Shift left: Bits on the left fall off, and insert zeros at the end
  - Arithmetic shift left is the same as logical shift right, **redundant and not supported in RISCV**

- **Logical** Shift right: Bits on the right fall off, and **insert zeros** at left **(NO sign extension)**
  - **srl** is **Not** suitable for negative numbers
- **Arithmetic shift right**: Bits on the right fall off, and left bits are **sign extended**

2.3.1.a. Usage of shifting

- Shift left by n is equivalent to multiplying by 2^n

- Shift right a **positive numbe**r by n is equivalent to dividing by 2^n, and **truncate the fractional part**
- Shift right a **negative odd number is equivalent to** dividing by 2^n, and **rounding towards negative infinity**

2.3.1.b. Shifting Example :

Q: Register x15 contains the pointer to an int array stored in memory. How to **store the value** located in register x10 to the index that is stored in x11 of the array?

```
slli x12, x11, 2    # compute offset: x11 * 4
add x12, x12, x15   # compute address: arr + offset
sw x10 0(x12)
```

2.4. Branching: Decision making Instruction

Type of branch instructions

- **Conditional branch**: Only branch if certain condition is met
- **Unconditional branch**: Always branch

Labels:

- **Labels** are used to **give** control flow instruction **places to go**
- You can **place a label** in the assembly **at the place** that you want to **branch** to and then specify that label in your code

2.4.1. Conditional branches

2.4.1.a Branch if equal / not equal

- `beq reg1, reg2, L1`
- If reg1 == reg2, jump to code at the location of label L1, otherwise continue executing the code in sequence

- `bne reg1, reg2, L1`
- If reg1 != reg2, jump to code at the location of label L1, otherwise continue executing the code in sequence

```
    beq x10, x11, Exit
    add x14, x13, x12
Exit:
```

```
    bne x10, x11, Exit
    add x14, x13, x12
Exit:
```

2.4.2.a. Branch if less than / greater or equal than

- `blt reg1, reg2, L1`
- If reg1 < reg2, jump to code at the location of label L1, otherwise continue executing the code in sequence

- `bge reg1, reg2, L1`
- If reg1 >= reg2, jump to code at the location of label L1, otherwise continue executing the code in sequence

```
    blt x10, x11, Exit
    add x14, x13, x12
Exit:
```

```
    bge x10, x11, Exit
    add x14, x13, x12
Exit:
```

2.4.2. UnConditional branches

Jump:

- `j label`
- Always jump to the code located at label

```
    bne x10, x11, else
    add x14, x13, x12
    j done
else: sub x14, x12, x13
done:
```

If-else Statement example

```
if (a == b)
    e = c + d;
else
    e = c - d;
```

x10 = a
x11 = b
x12 = c
x13 = d
x14 = e

```
    bne x10,x11,else
    add x14,x12,x13
    j done
else: sub x14,x12,x13
done:
```

2.4.3 Loop Intro

```
int A[20];
int sum = 0;
for (int i=0; i < 20; i++)
    sum += A[i];
```

```
    add x9, x8, x0  # x9 = &a[0]
    add x10, x0, x0  # sum = 0
    add x11, x0, x0  # i = 0
    addi x13, x0, 20  # x13 = 20 (loop upper bound = 20)
Loop:   bge x11, x13, Done
        lw x12, 0(x9)   # x12 = a[i]
        add x10, x10, x12   # sum = sum + a[i]
        addi x9, x9, 4  # x9 = &a[i+1]
        addi x11, x11, 1    # i++
        j Loop
Done:
```

The pointer = x8

Prologue: Initialization

1. Load base address of the array A to **x9**
2. Initialize sum at **x10**
3. Initialize iterative variable at **x11**
4. Initialize for loop upper bound to 20 at **x13**

Loop Body

1. Branching to **Done** if x11 >= x13
2. Load **memory[x9]** to **x12**
3. **Accumulate** sum at x10, add x10 by x12 at each iteration
4. **Pointer arithmetic,** *add offset to the base address (4 in this case)*
5. Update iterative variable i
6. Jump to next loop

Done

Finish program