Lecture 4 and Lab 4 notes

Mips Addressing

1. R-format => Register Addressing
    1. Using register as the operand
    2. Including arithmetic instructions, logical instructions, setting boolean, shifting...

op(6)	rs(5)	rt(5)	rd(5)	shamt(5)	funct(6)

    op denotes the R-format, i.e. **op = 0**;

    shamt is not 0 for shifting.

    funct denotes the exact type of the instruction.
2. I-format 1 => Immediate Addressing
    1. For instructions including immediate: **addi, andi, lui**...
    2. 16-bits immediate is sufficient

op(6)	rs(5)	rt(5)	Immediate(16)

    op denotes what exact type of instruction here
    3. for the occasional 32-bit constant(What `li` actually do is)

    ```
    lui rt, upper_16bits_constant
    ori rt, lower_16bits_constant
    ```
3. I-format 2 => Base Addressing
    1. Also called displacement addressing
    2. loading and storing data

op(6)	rs(5)	rt(5)	Address(16)

    Notes that Address acutally means the **offset** relative to the current site and it is in the **bytes**, which impies that the range of offset is $[-2^{15},\ 2^{15}-1]$**bytes**. Roughly speaking, $\pm 2^{15} bytes$

    From another view, offset of a address may not be aligned in a **word**. Thus, it's in bytes.
4. PC-relative addressing
    1. branch instructions specify: bne, beq...
    2. PC-relative addressing

    Actually it tells how the PC should go.

We know that an machine instruction take 32 bits. So PC will automatically step forward a word after excuting **any** machine instructions

From an machine-code view, to allow a larger range of offset, it should be in the **words** because the unit of code is **word**.

Target address = (PC+4) + Offset * 4

op(6)	rs(5)	rt(5)	Address(16)

Notes that the address is the offset from the **origin of next instructions** in **words**.

The range of the offset is $[-2^{15},\ 2^{15}-1]words$, or $[-2^{17},\ 2^{17}-4]bytes$. Roughly speaking, $\pm 2^{15}words$ or $\pm 2^{17}bytes$

5. J-format => psedodirect addressing
    1. jump(j and jal) targers could be anywhere in the test segment
    2. solutions to express more details of the **address**
        1. count in words
        2. invent a new format(J-format)
        3. borrow 4 MSBs from PC

op(6)	address(26)

    Target address = PC[31:28] : address * 4

    Only allow target address in the same PC field, i.e. in the range of 4GB / 16 = 256MB. Actually the space is enough.

    address*4 then directly denotes the 28 LSBs of the address of machine code we want to jump.

6. branching far away
    1. We know that the range of branching is about $\pm 2^{17}bytes$, while the range of jumping is about 256MB
    2. When we branch far away, assembler will rewrite the code

```
# original code
beq $s0, $s1, far_label

# rewritten code
bne $s0, $s1, L2
j far_label
L2: ...
```

Running a Program

1. translation vs. Interpretation
    1. difference:

        Interpreter directly execute a program in the source code, but slower

        Translater converts a program from the source language to an equivalent program in another language, and it's more efficient

    2. commonality:

        translation and interpretation helps "hide" the source code from the users

2. Step 1: compiler

    Input: high-level language code, e.g. **x.c**

    Output: low-level language code, e.g. **x.asm**

    The output may contain pseudo-instrucions

3. Step 2: Assembler

    Input: Assembly language code, e.g. **x.asm**

    Output: Object code(binary machine code), information table(The information table contains information such as the names and addresses of symbols used in the program.), e.g. **x.o**

    1. convert the **pseudo-instructions** into actual **hardware instrucions**
    2. convert the **assembly instructions** into **machine instructions**

4. Step 3: Linker

    Input: the object file, e.g. **x.o**

    Output: the executable file, e.g. **a.out**

    1. Stitches different object files into a **single** executable

5. Step 4: Loader

    Input: executable file, e.g. **a.out**

    Output: <program is run>

    1. Executable file is in the disk
    2. loader need to load it into memory

6. Remark

    Compiler optimizations are sensitive to the algorithm

    Java/JIT compoled code is significantly faster than JVM interpreted

RISC-V ISAs

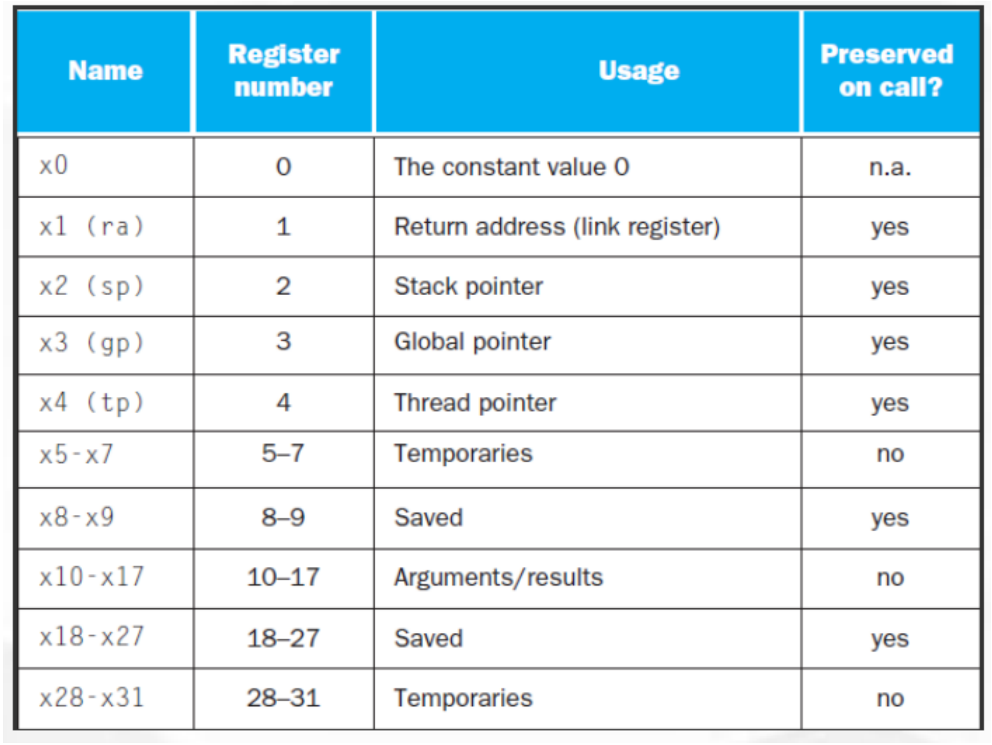
Reference:

> Specifications – RISC-V International (riscv.org)
>
> RISC-V基本介绍_辣椒油li的博客-CSDN博客
>
> 计组学习07——RISC-V Instruction Formats - ZzTzZ - 博客园 (cnblogs.com)
>
> IC知识库 - 个人中心 - 腾讯云开发者社区-腾讯云 (tencent.com)

1. Mips vs. RISC-V

	MIPS32	**RISC-V (RV32)**
License	**Proprietary**	**Open-Source**
Instruction size	32 bits	32 bits
Endianness	**Big-endian**	**Little-endian**
Addressing modes	**5**	**4**
Registers	32 × 32-bit	32 × 32-bit
Conditional branches	slt, sltu + beq, bnq	+blt,bge,bltu,bgeu

2. Registers in RISC-V



Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

remark:

x5 – x7, x28 – x31: temporaries

x8: frame pointer

x9, x18 – x27: saved registers

x10 – x11: function arguments/results

x12 – x17: function arguments

Instructions format

1. Instructions format in RISC-V

R-format	**instructions using 3 register inputs**
I-format	**instructions with immediate, loads**
S-format	**store instructions**
SB-format	**branch instrucions**
U-format	**instructions with upper immediate**
UJ-format	**jump instructions**

2. **R-format** Instructions(10 instructions)

funct7(7)	**rs2(5)**	**rs1(5)**	**funct3(3)**	**rd(5)**	**opcode(7)**

arithmetic, setting a boolean, logical, shifting instructions **without** ant immediate

opcode: distinguish the **format**

funct7 + fucnt3: indicate the specific operation

example: add x9, x20, x21 => add rd, rs1, rs2

arithmetic:

```
add rd, rs1, rs2 # rd = rs1 + rs2
sub rd, rs1, rs2 # rd = rs1 - rs2
```

setting a boolean:

```
slt rd, rs1, rs2 # rd = (rs1 < rs2)
sltu rd, rs1, rs2 # rd = (rs1 < rs2) unsigned
```

logical:

```
and rd, rs1, rs2 # rd = rs1 & rs2
or rd, rs1, rs2 # rd = rs1 & rs2
xor rd, rs1, rs2 # rd = rs1 & rs2
```

shifting:

```
sll rd, rs1, rs2 # rd = (rs1 << rs2)
srl rd, rs1, rs2 # rd = (rs1 >> rs2)
sra rd, rs1, rs2 # rd = (rs1 >> rs2) arithmetic
```

3. **I-format** Instructions(15 instructions)

Imm[11:0]	rs1(5)	funct(3)	rd(5)	opcode(7)

Immediate express **negative number** as 2's complement, with sign extension.

for example: 0000_0000_0001 => 1111_1111_1111

arithmetic, setting a boolean, logical instruction with immediate

```
addi rd, rs1, imm # rd = rs1 + imm
slti rd, rs1, imm # rd = rs1 < imm
sltiu rd, rs1, imm # rd = rs1 < imm (unsigned)
andi rd, rs1, imm # rd = rs1 & imm
ori rd, rs1, imm # rd = rs1 | imm
xori rd, rs1, imm # rd = rs1 ^ imm
```

loading:

```
lb rd, imm(rs1) # load a byte from rs1 + imm
lbu rd, imm(rs1) # load a unsigned byte from rs1 + imm
lh rd, imm(rs1) # load a half-word from rs1 + imm
lhu rd, imm(rs1) # load a unsigned half-word from rs1 + imm
lw rd, imm(rs1) # load a word  from rs1 + imm
```

jump and link register:

```
jalr rd, imm(rs1)
# set RETURN ADDRESS = PC + 4
# set PC = rd + imm

# we want to jump to an arbitrary 32-bit address
# lui x1, <upper 20 bits>
# jalr x1, <lower 12 bits>

# jump and return: a psudo-instruction
jr $x1 # jalr $x1, 0
```

shifting:

Imm[11:5]	Imm[4:0]	rs1(5)	funct(3)	rd(5)	opcode(7)

Imm[11:5] distinguish the **type** of shifting

Imm[4:0] indicate the **shamt**(at most 31)

```
slli rd, rs1, shamt # rd = rs1 << shamt
srli rd, rs1, shamt # rd = rs1 >> shamt
srai rd, rs1, shamt # rd = rs1 >> shamt (arithmetically)
```

4. **S-format** Instructions

Imm[11:5]	rs2(5)	rs1(5)	funct3(3)	Imm[4:0]	opcode(7)

{Imm[11:5], Imm[4:0]} indicates the offset from the base address of the **rs2**

```
sw rs2, imm(rs1)
sh rs2, imm(rs1)
sb rs2, imm(rs1)
```

5. **SB-format** Instructions

Imm[12|10:5]	rs2(5)	rs1(5)	funct3(3)	Imm[4:1|11]	opcode(7)

{Imm[12], Imm[11], Imm[10:5], Imm[4:1]} indicate the offset from the base address of the rs2

```
beq rs1, rs2, label # imm = label - pc
bne rs1, rs2, label # imm = label - pc
blt rs1, rs2, label # imm = label - pc
bltu rs1, rs2, label # imm = label - pc
bge rs1, rs2, label # imm = label - pc
bgeu rs1, rs2, label # imm = label - pc
```

6. **U-format** Instructions

imm[31:12]	rd(5)	opcode(7)

how can we deal with the 32-bit immediate? mostly we can deal with lower 12 bits if we need. Additionally, if we can manipulate the upper 20 bits, then we can deal with 32 bits immediate.

```
# load upper 20 bits
lui x1, 0xdeadc
# addi to load a 32-bit immediate
addi x1, x1, 0xeef

auipc rd, imm # rd = imm << 20 + pc
```

7. **UJ-format** Instructions(1 instruction) (pc-relative addressing)

imm[20|10:1|11|19:12]	rd(5)	opcode(7)

```
jal rd, label
# store (pc+4) to rd, we use $x1 by convention
# imm = label - pc
```

Directives

1. `.macro`, `.endmacro`

   macro is a **text-replacement** that provide a simple mechanism to name a frequently used sequence of instructions.

   syntax:

```
# without parameter
.macro print_hello
.data
    s: .asciiz "hello"
.text
    li $v0, 4
    la $a0, s
    syscall
.end_macro

# with parameter
.macro print_string(%str)
    li $v0, 4
    la $a0, %str
    syscall
.end_macro
# usage
.data
    str: .asciiz "hello"
.text
    print_string(str)

# %str can even be a string
.macro print_string(%str)
.data
    s: .asciiz %str
.text
    li $v0, 4
    la $a0, %str
    syscall
.end_macro
```

2. `.align`

   align **next** data item on specified byte boundary(0=byte, 1=half, 2=word, 3=double), i.e. the address of the next data will be a mutiplication of 1, 2, 4 or 8

```
# distinguished
.data # 1
str1: .ascii "welcome"
.align 2
str2: .ascii "to" str3: .asciiz "MIPS32World"
.text
```

```
la $t0, str2
lw $t1,($t0)

.data # 2
.align 2
str1: .ascii "Welcome" str2: .ascii "to" str3: .asciiz "MIPS32World"
.text
la $t0, str2
lw $t1,($t0)
```

3. `.globl` vs `.extern`

   1. `.include`

      insert the contents of the specified file, put filename in quote

      ```
      .include "example.asm"
      ```

   2. `.globl`

      The **label** can be seen without `include` within the program

      ```
      # in file1.asm
      .globl foo
      foo:

      # in file2.asm
      jal foo
      ```

   3. `.extern`

      when we want to use the **variable** in the **included** file, we should declare the label as extern.

      ```
      # in file1.asm
      .data
          var: .asciiz "abc"

      # in file2.asm
      .include "file2.asm"
      .extern var 4

      # the .extern can be declared in the file1 or file2
      # the bytes specified following the var should not conflict the actual size
      # we can use the "real" var in file2, not its copy
      ```

Memory: Stack vs Heap

1. Stack:

   when the require space is small, we can use the space in the stack directly

```
# for example, store the source registers in the callee
callee:
    addi $sp, $sp, -8
    sw $s0, ($sp)
    sw $s1, ($sp)
    # we can use s0, s1 here

    # stack preservation
    lw $s0, ($sp)
    lw $s1, ($sp)
    addi $sp, $sp, 8
    jr $ra
```

2. heap

   Dynamic memory allocation and deallocation: we should be really careful about the memory leak.

```
# allocate 100 bytes
li $v0, 9 # sbrk syscall
li $a0, 100
syscall
move $s0, $v0 # $s0 store the address of the dynamic space
# we should carefully preserve the $s0, if we want to change the address, we should use another register $t0.
# v0's address = heap-top pointer'address - 100

# deallocation
li $v0, 12 # brk call
move $a0, $s0 # store the address to $s0
syscall
# deallocate the memory whenever it won't be used again
```