Summary of Addressing Modes in MIPS

Introduction

MIPS has only a small number of ways that is computes addresses in memory. The address can be an address of an instruction (for branch and jump instructions) or it can be an address of data (for load and store instructions).

We'll look at the four ways addresses are computed

- **Register Addressing** This is used in the **jr** (jump register) instruction.
- **PC-Relative Addressing** This is used in the **beq** and **bne** (branch equal, branch not equal) instructions.
- **Pseudo-direct Addressing** This is used in the **j** (jump) instruction.
- **Base Addressing** This is used in the **lw** and **sw** (load word, store word) instructions.

We'll also consider *indirect addressing*, a popular addressing mode in CISC ISAs.

Register Addressing

Register addressing is used in the **jr** instruction. Because a register stores 32 bits, and because an address in a MIPS CPU is also 32 bits, you can specify any address in memory.

The typical call is:

```
jr $rs
```

where **$rs** is replaced by any register.

The semantics of this is:

```
PC <- R[s]
```

This means the PC (program counter) is updated with the contents of register **s**. Recall that a jump or branch is updated by modifying the contents of the program counter.

Register addressing gives you the ability to generate any address in memory. An *address exception* occurs if the two low bits are not 00.

PC-Relative Addressing

PC-relative addressing occurs in branch instructions, **beq** and **bne** (and other variations of branch instructions).

These instructions are I-type instructions, with the following format.

Opcode	Register s	Register t	Immediate
B_{31-26}	B_{25-21}	B_{20-16}	B_{15-0}
ooo ooo	sssss	ttttt	iiii iiii iiii iiii

The immediate value is only 16 bits, which means we can't address a large range of memory. The immediate value is in 16 bits 2C, so the range of values is -2^{15} up to $2^{15} - 1$.

Branch instructions are used primarily to implement loops or if-else statements. When you jump to a statement in an if-else or loop, you typically jump to a nearby instruction from the one you jumped from. The instruction you jumped from must have been the one at PC, thus it makes sense to jump *relative* to the PC.

For example, if you had a branch instruction at address 1000, then it's likely you'll jump somewhere near the branch instruction.

A naive (and thus incorrect approach) is to do the following:

$$PC \leftarrow PC + \text{sign-ext}_{32}(IR_{15-0}) \text{ WRONG!!}$$

where **sign-ext$_{32}$(X)** means to sign extend the X to 32 bits.

Why is this naive? We know the following facts:

- MIPS instructions are 32 bits.
- 32 bit quantities in MIPS must be stored in memory at word-aligned addresses.
- Word aligned addresses end in 00 when written in binary.
- Thus, the PC, which stores addresses of instructions must hold a word-aligned address.

If we're going to branch to some address, and PC is already word aligned, then the immediate value has to be word-aligned as well.

However, it makes no sense to make the immediate word-aligned because we're wasting the low two bits by forcing it to be 00.

It makes more sense to allow the 16-bit immediate to be any of the 2^{16} possible immediate values. That way you make full use of the immediate value. It's similar to IEEE 754 floating point where we don't represent the hidden 1 in the fraction for normalized floating point numbers. Since it's always there, why waste a bit to represent it?

The real PC-relative addressing

Here's how the address is really computed.

$$PC \leftarrow PC + \text{sign-ext}_{32}(IR_{15-0}::00) \text{ CORRECT!!}$$

You take the 16 bit immediate value, add two zeroes to the end (which is the same as shifting it logical left 2 bits). This creates a value that's divisible by 4. Then, you sign-extend it to 32 bits, and add it to the PC.

Thus, the range of possible addresses is: **PC - 2^{17}** up to **PC + (2^{17} - 4)**.

2^{17} is 128 K. So you can jump back roughly -128,000 bytes backwards up to about 128,000 bytes forward. That's large, but still a small fraction of memory. Fortunately, for branch instructions, you don't need to jump that far, if you've written reasonably good code.

Pseudo-Direct Addressing

Direct addressing means specifying a complete 32 bit address in the instruction itself. However, since MIPS instructions are 32 bits, we can't do that. In theory, you only need 30 bits to specify the address of an instruction in memory. However, MIPS uses 6 bits for the opcode, so there's still not enough bits to do true direct addressing.

Instead, we can do pseudo-direct addressing. This occurs in **j** instructions.

Opcode	Target

B_{31-26}	B_{25-0}
ooo ooo	tt tttt tttt tttt tttt tttt tttt

26 bits are used for the target. This is how the address for pseudo-direct addressing is computed.

$$PC <- PC_{31-28}::IR_{25-0}::00$$

Take the top 4 bits of the PC, concatenate that with the 26 bits that make up the target, and concatenate that with 00. This produces a 32 bit address. This is the new address of the PC.

This allows you to jump to 1/16 of all possible MIPS addresses (since you don't control the top 4 bits of the PC).

Base Addressing

The other three addressing modes modify the PC. They create addresses for branch/jump instructions.

However, load/store instructions also generate addresses in memory.

Let's consider the following instruction.

```
lw $rt, offset($rs)
```

where **$rs** and **$rt** are any two registers.

The offset is stored in 16 bits 2C. Thus, **lw** and **sw** are I-type instructions.

The address computed is:

$$addr <- R[s] + sign\text{-}ext_{32}(offset)$$

$rs is the base register, which is where the name base addressing comes from.

The **offset** is the 16 bit immediate value from the instruction. Unlike branch instructions, we don't add a 00 to the end of the immediate value, even though we can only load and store are word-aligned addresses.

The reason is because there are other load/store instructions that load/store halfwords and bytes, and it makes sense to compute the addresses the same way, regardless of what you're loading.

So what happens an address is computed that's not word aligned? An address exception occurs.

Indirect Addressing

MIPS does not support indirect addressing. RISC ISAs generally do not support such an instruction. However, it's a popular addressing mode for CISC ISAs.

Let's see how this indirect addressing works. First, recall how **lw** works. It adds the contents of a register to a sign-extended offset. This results in an address. The word stored at that address is loaded into a register.

Let's make up a new instruction called **indlw** which means "indirect load word". This instruction doesn't really exist in MIPS, but pretend it does.

```
indlw $rt, offset($rs)
```

In this case, we'll do the same thing. Add the sign-extended offset to **$rs**. But instead of loading the word at that address into a register, we load the word (say, to some temporary location like $at), and use that word as

an address, then we load that word from memory.

Thus, we go to memory twice. First time, we load a word that represents an address, and second time, we use that address to load a word of data.

The semantics are:

$$R[\ t\ ] \leftarrow M_4[\ M_4[\ R[\ s\ ] + \text{sign-ext}_{32}(\ \text{offset}\ )\ ]\ ]$$

Indirect addressing might seem odd, but if you increment the address stored in memory, it's one way to process an array. That address could be a pointer.

The major reason indirect addressing isn't used in RISC ISAs is because it accesses memory twice. The goal is to make each instruction fast, and accessing memory is considered slow. As it is, even load and store word don't really fully execute in one clock cycle (it takes about two, and that assumes it's in cache).

Still, this kind of addressing mode and far more are seen in CISC ISAs. The goal in those ISAs is to provide a person who programs in assembly language a lot of choices of instructions. CISC ISAs are meant to minimize the amount of memory used for a program by providing a large number of instructions.

The goal of RISC ISAs is minimize instructions to improve performance by making hardware design simpler (fewer instructions means simpler hardware, which means greater opportunities for optimization).

Nevertheless, money and brainpower can compesate for this seemingly good idea. Intel makes CISC processors that are now currently faster than RISC processors. Part of this is because Intel chips do some conversion to RISC-like instructions behind the scenes. Thus, you code in x86 (or, more properly, IA32) and the hardware converts it to RISC like internal instructions which you don't see.

Summary

All assembly languages give you a variety of way to generate addresses in memory. This can be the address of an instruction or the address of data.

We considered the four major ways to compute addresses in MIPS.

- register addressing
- PC-relative addressing
- pseudo-direct addressing
- base addressing

These four ways are much smaller than CISC ISAs, which may provides ten or more addressing modes. We discussed one of the more popular CISC addressing mode: indirect addressing.