Addressing 32-bit addresses and constants

Till now, ***only 16 bits have been addressed or used for specifying a constant.*** Often, more than the 16-bit field is needed.

32-bit Immediate Operands

How can we do a, say, **`addi` operation with the constant being more than 2^{16}** (65536)?
We **set a register to this constant**, and do the `add` operation instead of `addi`.

`lui` loads the 16 bits of the constant into the high bits of a register:

∴, to load 4000000 in register `$s0`
4000000_{dec} = 0000 0000 0011 1101 0000 1001 0000 0000_{binary}

We load the higher 16 bits in `$s0`'s higher 16:
`lui $s0, 61` # 0000 0000 0011 1101_{binary} = $61_{decimal}$

Contents of register `$s2` after above:
0000 0000 0011 1101 0000 0000 0000 0000

Then, OR `s2` to get the lower bits in:
`ori $s2, 2304` # or with $2304_{decimal}$ = 0000 1001 0000 0000_{binary}

So now **the operand is converted into a register** by transfering it to one. ∴ now the reg. version can be used.

So if we wanted to do an `addi $s0, $t0, 66000`, we can first load 66000 in a reg., say. `$s2` by using the operations above, and then do `add $s0, $t0, $s2.`

32-bit addresses (in jump and branches)

Jump to address allows for 26-bit addresses (jump to register of course allows for 32-bit addresses), which is pretty decent.

Therefore, `j 10000 # go to location 10000`
will be encoded as:

2	10000
6 bits	26 bits

However, for branches, we only have 16 bits for addresses:

Therefore, `bne $s0, $s1, Exit  # Exit if s0 ≠ s1`
will be encoded as:

5	16	17	Exit
6 bits	5 bits	5 bits	16 bits

16 bits is not much at all, since we then can't have a program bigger than 2^{16} bytes (may want to branch to a procedure stored at a very large addr. or may be the program is huge already, say PC holds 68K, ∴ even a branch to two instructions ahead will be too big unless we use *relative addressing*). In any case, being able to branch to addresses less than 2^{16} is unrealistic.

What if, we can add these 16 bits to another register? Then actual address = $2^{16} + 2^{32}$ which allows us to have address of length 2^{32}.

Therefore, PC = branch add. + the register value

Which register? Usually jumps and branches are taken as part of loops and if() statements, so will be close to the PC.
What if we consider the beanch add. as an offset from the current instruction?
Then PC can serve as the register.
∴, we can address $\pm 2^{15}$ ($2 * 2^{15} = 2^{16}$ since we are doing both directions) words from the current instruction. Actually, the hardware increments PC quickly, so the relative address is not from current instruction, but from next instruction (PC + 4).
This is called **PC–relative addressing**.

Important: Usually the address in instructions is the address of a byte. For e.g., in `lw $s2, 2000`, we want the word starting from the *byte* 2000 to be loaded. However, in relative addressing as above, **the offset is in terms of no. of *words*** from next instructions. So in `beq $s1, $s2, 64`, we want to jump 64 words away from next instruction, ∴ 64*4 bytes away.

Similarly, **both the 26-bit address in jump and jump-and-link instructions is for a word** and to get to the actual address, we will have to multiply it by 4 to get an actual **28-bit byte address**.