High level language programs versus Machine code

You all have experience programming in a high level language. You know that when you write a program in a language such as C or Java, the program is nothing more than a string of ASCII characters that you type on a keyboard and that are stored in a file. If you are unfamiliar with ASCII, then you should check out `www.asciitable.com`. Note that ASCII is a 7 bit code. The 8th bit of each byte is ignored, although there is something called extended ASCII which uses all 8 bits.

[ASIDE: Another character code which is much more elaborate is UNICODE. I like to think of it as a 16 bit code `www.unicode-table.com`, but in fact it is more complicated than this. `https://en.wikipedia.org/wiki/Unicode`.)]

You also know that in order to run a program that is stored in a file, you need to translate this program into executable instructions (*machine code*, made of 0's and 1's) that specify what your particular computer is supposed to do. Your particular computer has a processor (made by AMD, Intel, etc) and your program must be translated into instructions for this particular machine.

When you write your program in a high level language like Java, you typically don't want to consider which machine you are going to run this program on. This flexibility allows your program to be run on many different machines. Once you decide what machine you will use, though, you need to translate the program into a language for that machine. This translation is done by a *compiler*. A compiler is a program that takes a high level language program (say, in C) and translates it into language that is written in machine code.

In Java, the situation is a bit more subtle. Compiling a .java file gives you a .class file which is machine code. However, this machine code isn't understandable by your computer's processor. Rather it is code that runs on a virtual (abstract) computer called a Java Virtual Machine. This JVM code must be again translated – or more precisely, interpreted[1] – into machine code for your particular processor.

Assembly language

When computers were first invented in the 1940s, programmers had to program in machine code – they had to set 0's and 1's by flipping physical switches. For most people, it is not much fun (and certainly not efficient) to program with 0's and 1's, and so a slightly higher level of code, called *assembly language* was invented. Assembly language is just human readable machine code. The MIPS language we will use in the next few weeks is an example.

An assembly language program is an ASCII file, and is not executable. It needs to be translated into its machine code equivalent. This translation is relatively easy (compared with translating from a high level language like C to machine code). The translation from assembly language into machine code is done by a program called an *assembler*. We are not going to learn in this course

[1] I neglected to mention this in the lecture but I'll mention it here for your interest. The program which interprets the JVM code is called an *interpreter*. The interpreter simulates the JVM. Note that translating and interpreting are not the same thing. Translation happens before the program runs. Interpreting happens while the program is running. This is the same notion of translation versus interpretation that is used in natural languages: people who work as *translators* take written documents in one language and create written documents in another language. People who work as *interpreters* take spoken language in real time and convert it into another language. The two jobs are very different. When politicians travel to foreign countries, they need an interpreter, not a translator.

how exactly this is done.[2] Rather, we will just learn an assembly language and get experience programming in it, and understand what this corresponds to at the level of the actual processor.

MIPS

In the next few weeks, we will look at a particular machine or computer processor: the MIPS R2000 (from here one, just known as MIPS). We will spend the next few weeks getting to know the MIPS "instruction set" (set of MIPS instructions) and you will get some experience programming in the MIPS assembly language.

Before we can introduce MIPS instructions, we need to know that these instructions can refer to two types of memory. The first is the set of 32 MIPS registers that I described in lecture 6 (page 5). The second is a much larger memory that holds all the data and instructions of a user's programs, as well as the instructions and data of *the kernel* or operating system. If you wish to write programs for an operating system for the given computer, then you need to make use of the special instructions and data and memory locations of the kernel. We won't do that in this course. We'll only write user programs.

If you are new to programming, then it may surprise you to learn that both instructions and data sit in memory. This may be counterintuitive since you might think that instructions and data are very different things. As we will see, though, both instructions and data ultimately need to be encoded as 0's and 1's, and so there is no reason why they cannot both sit in memory. Your intuition is correct, though, in that they are kept in separate places in memory, respecting the fact that they are different types of things.

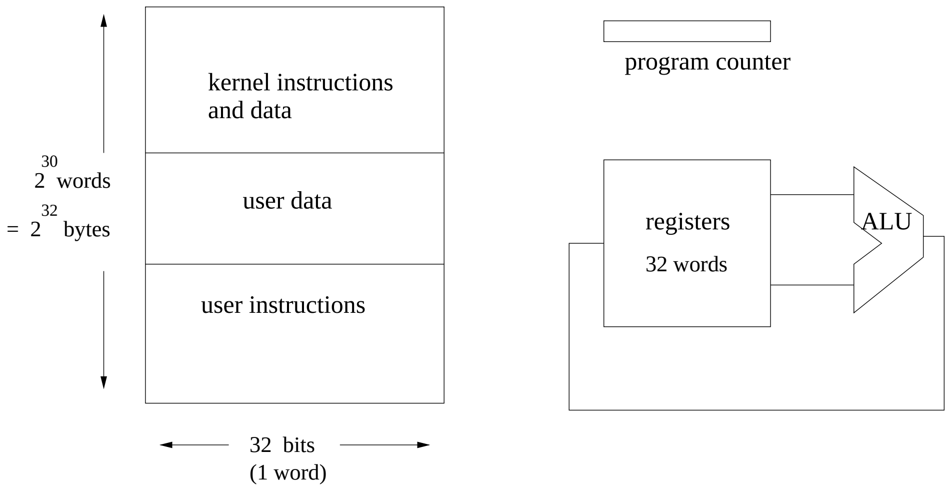
In MIPS, we write Memory (with a capital M) to refer to a set of 2^{32} bytes (or 2^{30} *words*, *i.e.* a word is 4 bytes) that can be addressed by a MIPS program. These 2^{32} bytes do not include the 32 MIPS registers. These 2^{32} bytes should *not* be confused with an address on a physical memory chip or a location on a disk. As we will see later in the course, MIPS Memory addresses need to be translated into physical addresses.[3]

You should have only a simple notion of how a MIPS computer works. This is illustrated in the figure below. You should understand that instructions and data are stored in Memory. You should also understand that operations such as addition and subtraction are implemented by selecting two numbers that are stored in registers, running them through the ALU, and then writing the result back into a new register. The figure above shows another register which is called the *program counter*. It holds the address of the instruction that is currently being executed. This instruction is in Memory and so the address of the instruction is 32 bits.

In this lecture, we will see some examples of instructions for carrying out such arithmetic operations. We will also see instructions for moving words to and from Memory and the registers. Finally, we will see how a program goes from one instruction to the next, and how branches to other instructions can be achieved.

[2] If you want to learn about this topic, then take the Compiler Design course COMP 520.

[3] An analogy here is postal codes versus (latitude,longitude). Postal codes are used by the postal system to sort and direct mail. (Latitude,longitude) coordinates specify where the mail goes in a physical sense. If this is confusing to you, then ask yourself: do you believe that that postal code A0A0A0 is at the tip of Newfoundland and Z9Z9Z9 is at the western tip of Vancouver Island?



Arithmetic instructions

Consider the C instruction:

$$\texttt{c} = \texttt{a} + \texttt{b};$$

In the MIPS assembly language, this instruction might be written:

```
add $16, $17, $18     # register 16 is assigned the sum of registers 17 and 18
```

The `$` symbol marks a register, e.g. `$17` is register 17. The `#` symbol marks a comment. When this symbol appears in a MIPS program, any characters to the right of this symbol are ignored.

Memory transfer instructions

In MIPS, we cannot perform arithmetic or logical operations on numbers that are stored in Memory. The numbers must first be brought from Memory into registers and the operation performed from there. To load a word (4 bytes) from Memory to a particular register, we use the instruction `lw`, or "load word". The `lw` instruction specifies the *register* that we are going to write to. It also specifies the *address* in Memory of the word we want. The Memory address is the sum of a *base address* plus an *offest*. The base address is a 32 bit number which is currently stored in a register, and this register must be specified. The offset also must be specified. Here is an example:

```
lw  $16, 40($17)      #  Bring a word from memory into register 16.
```

In this example, the address of the word in Memory is the value in register 17, plus 40. Allowing for an offset (e.g. 40) gives the programmer more flexibility. If it happens that the address of the word we want is already in `$17`, then then the offset would just be 0.

What about the opposite operation, namely taking a word that is in a register and putting it into Memory? For this, MIPS has `sw`, which stands for "store word".

```
sw $16, 40($17)      # Put the word in $16 into Memory
```

Again, the address of the word in Memory is the value in register 17, plus 40.

In the lecture, I described a few simple examples that combine arithmetic instructions like `add` and Memory transfer instructions. Say you are trying to perform `x = y + z`, and `x` and `y` values happen to be in registers but the value of `z` is not in a register, then you need to load `z` from Memory into a register before you can compute the sum. Similarly, if `y` and `z` are in registers but `x` is in Memory, then you need to compute the sum `y + z` and store it in some other register, and then copy (store) the value in that register to `x`, which is in Memory.

In the slides, I also sketched out a *data path* for the above `lw` and `sw` instructions. We will go into more detail about data paths in the weeks ahead.

Branching instructions

What is the order of instructions executed in a MIPS program? As I mentioned earlier, the *program counter (PC)* register holds the address in Memory of the currently executed instruction. The default is that this counter is incremented after each instruction.

Sometimes, however, it is useful to branch to other instructions. You are familiar with this idea e.g. `if-then-else` statements, `for` loops or `while` loops, `goto` statements. Branching instructions allow the program to execute an instruction other than the one that immediately follows it. For example, consider the following snippet of C code.

```
if (a != b)
   f = g + h;
```

Here you want to execute the sum if `a` is not equal to `b`, and so you don't want to execute the sum if `a` is equal to `b`. In MIPS assembly language, such an instruction could be implemented using `beq`, which stands for "branch if equal", for example:

```
            beq $17, $18, Exit1      # if a is equal to b, goto Exit1
            add $19, $20, $21        # f = g + h
Exit1:                               # Next instruction (following if statement).
```

The `beq` instruction is an example of a *conditional branch.* A condition needs to be satisfied in order for the branch to be taken, that is, for the program to execute an instruction other than the one that directly follows. Let's look at a slightly more complicated C instruction.

```
if (a != b)
   f = g + h;
else g = f + h;
```

This one is more tricky. If the condition `(a != b)` is met, then f is assigned a new value and the program must jump over the else statement. For this, we use a jump instruction `j`.

```
            beq   $17,  $18,   Else      # if a is equal to b, goto Else
            add   $19,  $20,  $21        # f = g + h
            j       Exit                 # goto Exit
Else:       add   $20,  $19,  $21        # g = f + h
Exit:
```

MIPS instruction formats: machine code

Let's now look at how these instructions are coded as 0's and 1's. Each MIPS instruction is one word (1 word = 4 bytes = 32 bits). The upper six bits $(26 - 31)$ of the word always contain an "op code" that specifies what operation the instruction performs. There are $2^6 = 64$ op codes. There are more than 64 operations, though, as we'll see in a moment.

The remaining bits of each instruction are organized into three formats, called R, I, or J.

R-format instruction (R stands for register)

An example of this instruction is `add`. As we saw above, such instructions use three registers. To specify these registers, we need 5 bits each. This leaves 32-15 = 17 bits, 6 of which we saw are for the op code. In general, the R format instruction is as follows:

field	op	rs	rt	rd	shamt	funct
numbits	6	5	5	5	5	6

Here is what the different fields mean:

- "op" stands for "opcode". R-format instructions always have opcode 000000.
- "rs","rt", "rd" are registers.

  The rs field refers to a "source" register. This is one of the two registers we read from (RegRead1). The rd field refers to a "destination" register. This is always the register that is written to (RegWrite). The rt register can serve either as a source register or a destination register.[4] In the case of an R format instruction, it is a source register (RegRead2).

  Notice that the ordering of these three register numbers within the 32 bit machine code of the `add` instruction is not the same as the ordering defined by instruction syntax which you use when programming. (The reason why should be clear in a few weeks.)
- "shamt" stands for "shift amount" (It is not used in `add`. We will discuss it later.)
- "funct" stands for function. It is used to distinguish different R format instructions. For example, the subtraction operation `sub` has the same opcode as `add`, namely 000000, but it has a different funct code.

  Note that there are 64 possible R-format instruction. Why? R-formal instructions always have opcode 000000, and there are $2^6 = 64$ possible funct fields. The instructions like `add`,`sub`, and bitwise operations `and`, `or`, `nor`, etc are all R format and have different funct codes.

[4]It is unclear why the letter "t" is used.

I-format instruction: (I stands for "immediate")

I format instructions use 6 bits for the opcode, 5 bits for each of two registers (rs and rt); and 16 bits for an *immediate value.* For `lw`, `sw`, `beq`, the immediate value is a signed *offset.*

field	op	rs	rt	immediate
numbits	6	5	5	16

For the `lw` instruction, a word is transfered from memory to a register. The address of the word in Memory is determined by the source register rs which contains a 32 bit *base address* plus the 16 bit offset. The offset is a signed number (from -2^{-15} to $2^{15} - 1$). The rt register acts as a destination register.

For the `sw` instruction, a word is transfered from a register to Memory. The address of the word in Memory is determined by the source register rs which contains a 32 bit *base address*) plus the 16 bit signed offset. The register rt contains the 32 bit value to be stored at this address.

For the `beq` instruction, the two registers rs and rt hold 32 bit values that are compared. Depending on the result of the comparison (true or false), the next instruction either is the next instruction in sequence (PC + 4 bytes) or some other instruction (PC + offset). The *relative address* of this other instruction is determined by the 16 bit immediate field i.e. by the offset. As we will see later in the course when we examine data paths, this offset is in words, not bytes. The reason is that instructions are always one word long, so the offset is always a multiple of 4 bytes, since the instruction we are branching to is always a multiple of 4 bytes away.

J format ("jump")

An example of the J format instruction is the jump instruction `j` that we saw earlier. This instruction uses 26 bits of address, rather than 16 bits as we saw with the I format instructions. With 26 bits, we can jump farther than we could with 16 bits. Later when we talk about data paths, I will specify exactly how this is done.

field	op	address
numbits	6	26