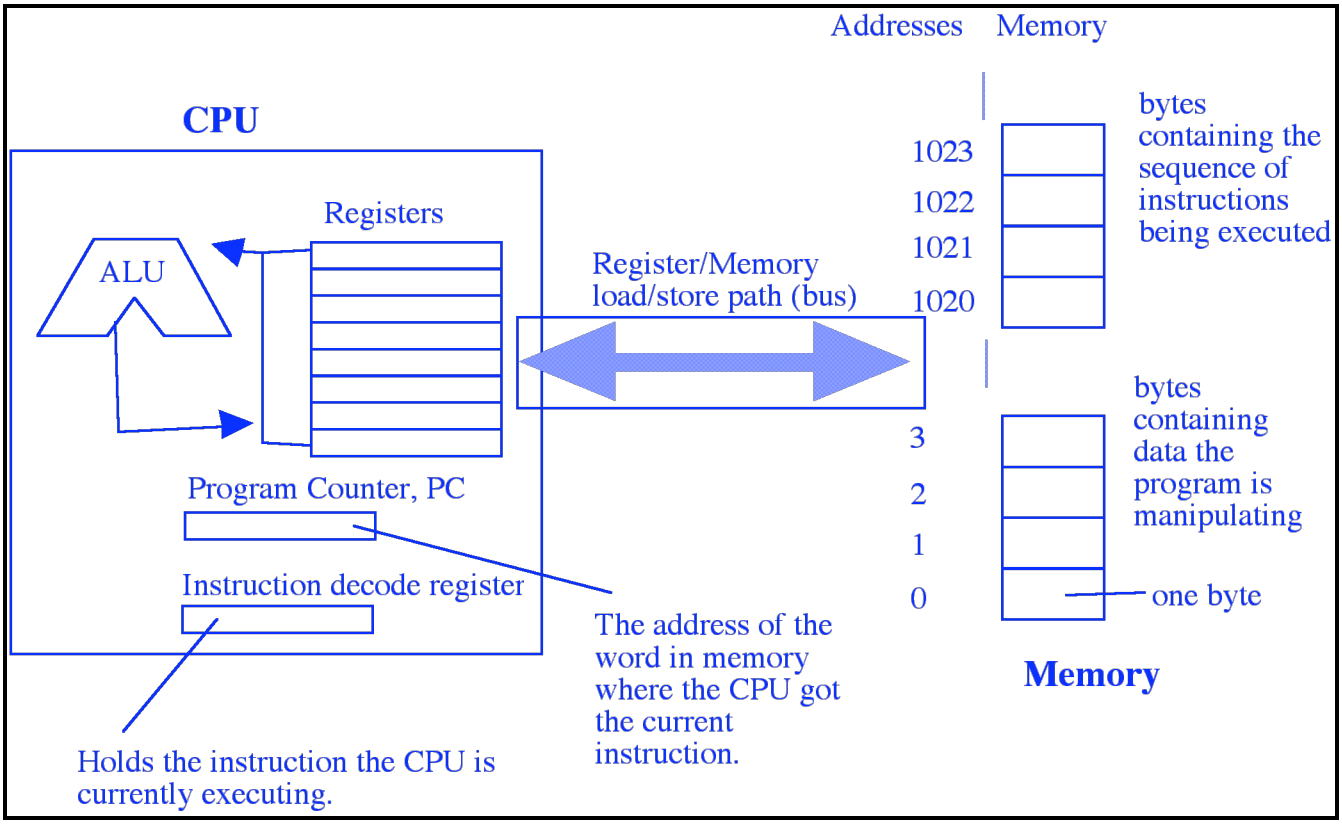
Computer Architecture: Take I

Handout written by Julie Zelenski and Nick Parlante

Computer architecture

A simplified picture with the major features of a computer identified. The CPU is where all the work gets done, the memory is where all the code and data is stored. The path connecting the two is known as the "bus."



CPU

While memory stores the program and the data, the Central Processing Unit does all the work. The CPU has two parts— registers and an Arithmetic Logic Unit (ALU). A register is like the temporary memory in a calculator— each register can store a value that can be used in subsequent computations. Each register can usually hold one word. Sometimes there are separate specialized registers for holding specific types of data— floating point numbers, addresses, etc. The registers can be accessed much more quickly than memory, but the number of registers available is quite small compared to the size of memory. One of the specialized registers is the Program Counter, or PC, which holds the address of which instruction is currently being executed.

The ALU is the part of the CPU that performs the actual computations such as addition and multiplication along with comparison and other logical operations. Most modern, high-performance CPUs actually contain several specialized logic units in addition to the ALU which allow them to work on several computations in parallel. However, that complexity is kept (literally) within the CPU. The abstraction of the CPU is that it executes its instructions one at a time, in the order presented in memory.

For the balance of this handout, we will concentrate on the instruction set of a typical Reduced Instruction Set Computer (RISC) processor. RISC processors are distinguished by a relatively lean instruction set. It's turned out that you can get the best overall performance by giving your processor a simple instruction set, and then concentrating on making the processor's instructions/second performance as high as possible. So RISC processors don't have some of the fancier instructions or addressing modes featured by older Complex Instruction Set (CISC) designs. Thankfully, RISC has the added benefit that it's easy to study since the instruction set is, well, reduced.

Our fictitious processor has 32 registers, each of which can hold a 4-byte word. We will support three types of instructions: Load/Store instructions which move bytes back and forth between registers and memory, ALU instructions which operate on the registers, and Branch/Jump instructions that alter which instruction is executed next.

Load

Load instructions read bytes into a register. The source may be a constant value, another register, or a location in memory. In our simple language, a memory location is expressed Mem[*address*] where *address* may be a constant number, a register, or a register plus a constant offset. Load and Store normally move a whole word at a time starting at the given address. To move less than a whole word at a time, use the variants "=.1" (1 byte) and "=.2" (2 bytes).

```
Load the constant 23 into register 4
R4 = 23

Copy the contents of register 2 into register 3
R3 = R2

Load char (one byte) starting at memory address 244 into register 6
R6 =.1 Mem[244]

Load R5 with the word whose memory address is in R1
R5 = Mem[R1]

Load the word that begins 8 bytes after the address in R1.
This is known as "constant offset" mode and is about the fanciest
addressing mode a RISC processor will support.
R4 = Mem[R1+8]
```

Just to give a sense of how this relates to the "real world", the load instruction in Motorola 68000 assembly looks like this:

```
Move long (4 bytes) constant 15 to register d2
movel #15, d2

Move long at mem address 0x40c to register a0
movel @#0x40c, a0
```

Or in Sparc assembly, it looks like this:

```
Load from mem address at register o0 + constant offset 20 into register o1
ld [ %o0 + 20 ], %o1
```

The syntax of our assembly language is designed to make it easier to read and learn quickly, but the basic functionality is quite similar to any current RISC instruction set.

Store

Store instructions are basically the reverse of load instructions— they move values from registers back out to memory. There is no path in a RISC architecture to move bytes directly from one place in memory to somewhere else in memory. Instead, you need to use loads to get bytes into registers, and then stores to move them back to memory.

```
Store the constant number 37 into the word beginning at 400
Mem[400] = 37

Store the value in R6 into the word whose address is in R1
Mem[R1] = R6

Store lower half-word from R2 into 2 bytes starting at address 1024
Mem[1024] =.2 R2

Store R7 into the word whose address is 12 more than the address in R1
Mem[R1+12] = R7
```

ALU

Arithmetic Logical Unit (ALU) instructions are much like the operation keys of a calculator. ALU operations only work with the registers or constants. Some processors don't even allow constants (i.e. you would need to load the constant into a register first).

```
Add 6 to R3 and store the result in R1
R1 = 6 + R3

Subtract R3 from R2 and store the result in R1
R1 = R2 - R3
```

Although we will use '+' for both indiscriminately, the processor usually has two different versions of the arithmetic operations, one for integers and one for floating point numbers, invoked by two different instructions, for example, the Sparc has `add` and `fadd`. Integer arithmetic is often much more efficient than floating point since the operations are simpler (e.g. require no normalization). Division is by far the most expensive of the arithmetic operations on either type and often is not a single instruction, but a small "micro-coded" routine (think of it as very fast hand-tuned function).

Branching

By default, the CPU fetches and executes instructions from memory in order, working from low memory to high. Branch instructions alter this default order. Branch instructions test a condition and possibly change which instruction should be executed next by changing the value of the PC register. One condition that all processors make heavy use of is testing whether two values are equal or if a value is less (or greater) than some other value. The operands in the test of a branch statement must be in registers or constant values. Branches are used to implement control structures like `if` and `switch` as well as loops like `for` and `while`.

```
Begin executing at address 344 if R1 equals 0
BEQ R1, 0, 344                    "branch if equal"

Begin executing at addr 8 past current instruction if R2 less than R3
BLT R2, R3, PC+8                  "branch if less than"
```

The full set of branch variants:

```
BLT   branch if first argument is less than second
BLE   less than or equal
BGT   greater than
BGE   greater than or equal
BEQ   equal
BNE   not equal
```

Any branch instruction compares its first two arguments (which both must be registers or constants) and then potentially branches to the address given as the third argument. The destination address can be specified as an absolute address, such as 356, or a PC-relative address, such as PC-8 or PC+12. The later allows you to skip over a few instructions or jump to a previous instruction, which are the most common patterns for loops and conditionals.

In addition, there is an unconditional jump that has no test, but just immediately diverts execution to a new address. Like branch, the address can be specified absolute or PC-relative.

```
Begin executing at address 2000 unconditionally- like a goto
Jmp 2000

Begin executing at address 12 before current instruction
Jmp PC-12
```

Data conversion

Two additional instructions are utilities that convert values between integer and floating point formats. Remember a floating point 1.0 has a completely different arrangement of bits than the integer 1 and instructions are required to do those conversions. These

instructions are also used to move values for computers that store floating point and integer values in different sets of registers.

For our purposes, we will have instructions that convert between the 4-byte integer and the 4-byte float value. The destination and source locations for the conversion operations must both be registers.

```
Take bits in R3 that represent integer, convert to float, store in R2
R2 = ItoF R3

Take bits in R4, convert from float to int, and store back in same Note
that converting in this direction loses information, the fractional
component is truncated and lost
R4 = FtoI R4
```

Summary

Although there are a few things we bypassed (the logical and/or/not and some of the operations that support function call/return), this simple set of instructions gives you a pretty good idea of exactly what our average CPU can do in terms of instructions. The richness and complexity of a programming language like C is provided by the compiler which takes something complex like a for-loop, array reference, or function call and translates it into an appropriate sequence of the above simple instructions.