Introduction

Computer design is set to achieve a common goal, find a language that makes it easy to build the hardware and the compiler while maximizing performance and minimizing cost and energy.

The "simplicity of the equipment" is a valuable part of designing a machine, as the more simplistic the instructions to be interpreted, the better the overall performance gains will be. Section 2.15 shows how the examples will change with object-oriented languages.

Operations of the Computer Hardware

Notation for performing arithmetic in RISC-V assembly language:

add a, b, c

This gives the computer instructions to add the two variables b and c and store the value of that in the variable a. This notation can only use three variables at a time, due to the concept of simplicity. The less variables the computer has to deal with at a time, the faster the operation will be and for this reason, arithmetic is done on multiple lines.

For example, take the equation f = a + b + c + d, this would have to be written as,

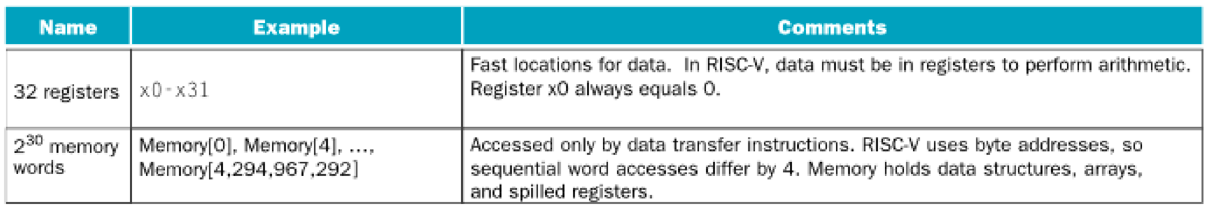
add f, a, b // The sum of a and b is stored in f

add f, f, c // The sum of a, b, and c is stored in f

add f, f, d // The sum of a, b, c, and d is stored in f

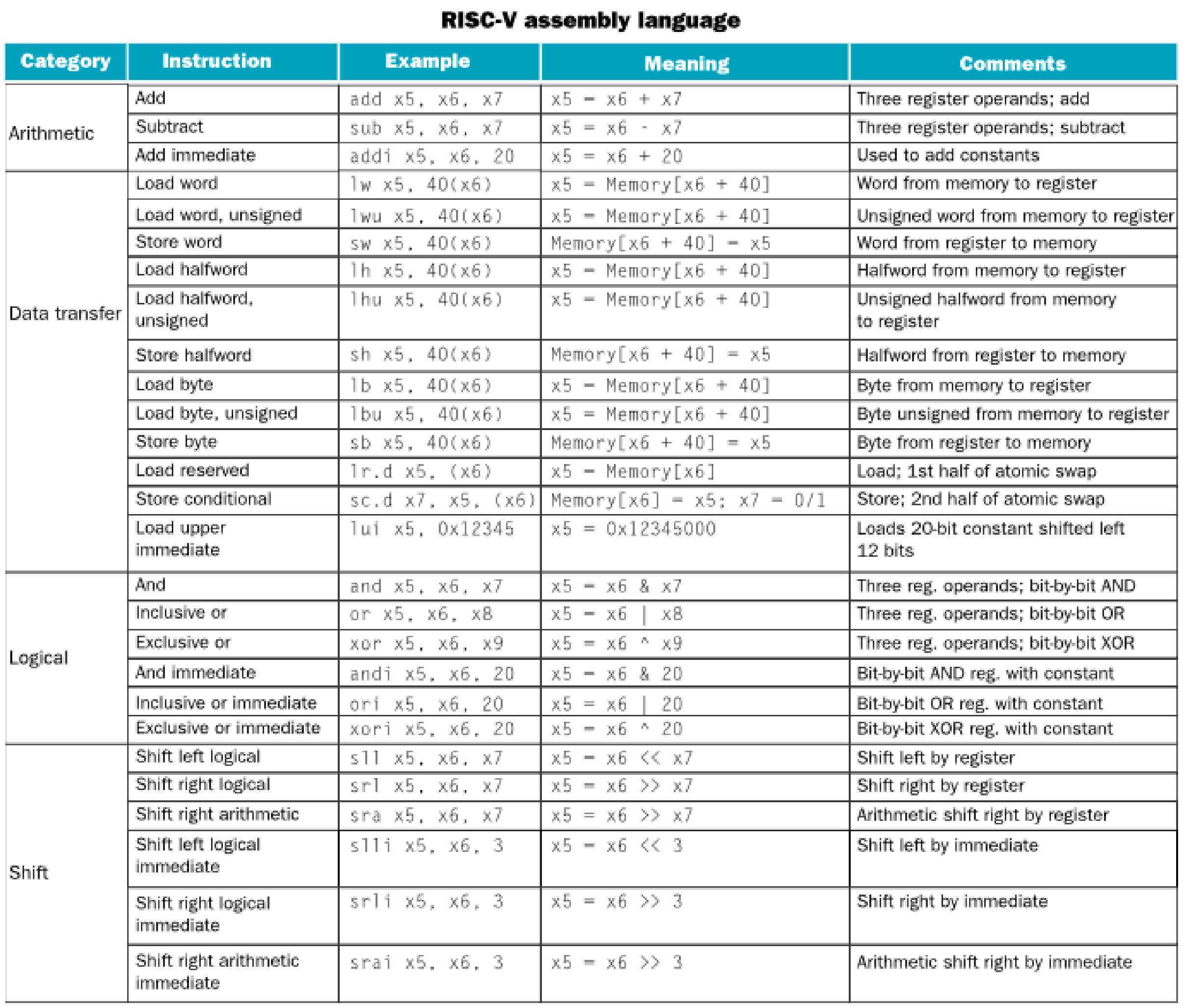
This gives the sum of the four variables and stores its value into f. The two slash marks indicate a comment and explain what is happening at each of the steps.

RISC-V operands



Name	Example	Comments
32 registers	x0-x31	Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4,294,967,292]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential word accesses differ by 4. Memory holds data structures, arrays, and spilled registers.

This table explains what the "variables" in the examples are supposed to be. According to the standards of RISC-V, in order to perform arithmetic on some data, the data must be in the CPU registers. This makes accessing them easier than parsing through memory, making computations faster and therefore improving the overall performance of the machine. There are 32 32-bit registers, ranging from x0 to x31. The number of operands for addition is 3, the two numbers being added and a place to put the sum.



RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	`add x5, x6, x7`	`x5 = x6 + x7`	Three register operands; add
	Subtract	`sub x5, x6, x7`	`x5 = x6 - x7`	Three register operands; subtract
	Add immediate	`addi x5, x6, 20`	`x5 = x6 + 20`	Used to add constants
Data transfer	Load word	`lw x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Word from memory to register
	Load word, unsigned	`lwu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned word from memory to register
	Store word	`sw x5, 40(x6)`	`Memory[x6 + 40] = x5`	Word from register to memory
	Load halfword	`lh x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Halfword from memory to register
	Load halfword, unsigned	`lhu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned halfword from memory to register
	Store halfword	`sh x5, 40(x6)`	`Memory[x6 + 40] = x5`	Halfword from register to memory
	Load byte	`lb x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte from memory to register
	Load byte, unsigned	`lbu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte unsigned from memory to register
	Store byte	`sb x5, 40(x6)`	`Memory[x6 + 40] = x5`	Byte from register to memory
	Load reserved	`lr.d x5, (x6)`	`x5 = Memory[x6]`	Load; 1st half of atomic swap
	Store conditional	`sc.d x7, x5, (x6)`	`Memory[x6] = x5; x7 = 0/1`	Store; 2nd half of atomic swap
	Load upper immediate	`lui x5, 0x12345`	`x5 = 0x12345000`	Loads 20-bit constant shifted left 12 bits
Logical	And	`and x5, x6, x7`	`x5 = x6 & x7`	Three reg. operands; bit-by-bit AND
	Inclusive or	`or x5, x6, x8`	`x5 = x6 | x8`	Three reg. operands; bit-by-bit OR
	Exclusive or	`xor x5, x6, x9`	`x5 = x6 ^ x9`	Three reg. operands; bit-by-bit XOR
	And immediate	`andi x5, x6, 20`	`x5 = x6 & 20`	Bit-by-bit AND reg. with constant
	Inclusive or immediate	`ori x5, x6, 20`	`x5 = x6 | 20`	Bit-by-bit OR reg. with constant
	Exclusive or immediate	`xori x5, x6, 20`	`x5 = x6 ^ 20`	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	`sll x5, x6, x7`	`x5 = x6 << x7`	Shift left by register
	Shift right logical	`srl x5, x6, x7`	`x5 = x6 >> x7`	Shift right by register
	Shift right arithmetic	`sra x5, x6, x7`	`x5 = x6 >> x7`	Arithmetic shift right by register
	Shift left logical immediate	`slli x5, x6, 3`	`x5 = x6 << 3`	Shift left by immediate
	Shift right logical immediate	`srli x5, x6, 3`	`x5 = x6 >> 3`	Shift right by immediate
	Shift right arithmetic immediate	`srai x5, x6, 3`	`x5 = x6 >> 3`	Arithmetic shift right by immediate

These are the list of built in functions that can be used by the assembly language to give instructions. It can be seen that many of these commands have only three operands. This is to serve the purpose of simplicity. **Design Principle 1: Simplicity favors regularity.**

Example:

Take for example these operations written in high-level languages such as C,

$$a = b + c;$$

$$d = a - e;$$

The compiler will translate this from the C language to RISC-V and create the assembly instructions,

add a, b, c

sub d, a, e

Example:

Take the equation f = (g + h) - (i + j); what would the C compiler produce? There are multiple variables here and we know that the assembly language can only make use of three at a time.

```
add t0, g, h     // Temp variable t0 takes on the value of g + h
add t1, i, j     // Temp variable t1 takes on the value of i + j
sub f, t0, t1    // f takes on the value of (g + h) - (i + j)
```

It can be seen here that this process takes three lines of code where as in the high level language the equation was merely one line. First it adds the two variables that need adding together, g and h, and i and j, and then takes the values of these sums and subtracts them to get the desired output for f.

Operands of the Computer Hardware

Register Operands

The operands in assembly languages are restricted in that they must come from a specified place. These places are known as *registers* and the reason for this, as stated before, is for performance enhancement. The closer the objects are to the CPU and the easier they can be accessed, the faster the program will be able to run.

Word – corresponds to a unit of access in a computer, a group of 32 bits. RV32 refers to the size of a RISC-V register which is 32 bits.

Doubleword – corresponds to a unit of access in a computer, a group of 64 bits. RV64 refers to the size of a RISC-V register which is 64 bits.

Most computers use 32 registers and the RV32 architecture will be most often used. In both cases, the restriction on the number of registers to 32 is due to the principle that smaller is faster. **Design Principle 2: Smaller is faster.** A large number of registers may increase the clock cycle time since the signals inside the chip will have to travel a farther distance. The registers 0 through 31 are indicated with the letter x and the number of the register following.

Taking the same example as before and fixing it so that it uses registers instead of arbitrary variables, let's say that f, g, h, i, and j are assigned to the registers x19, x20, x21, x22, and x23. We will also have to have two registers open to store the temporary values, let's call that x5 and x6.

```
add x5, x20, x21    // Register x5 takes on the value of g + h
add x6, x22, x23    // Register x6 takes on the value of i + j
add x19, x5, x6     // Register x19 takes on the value of (g + h) - (i + j)
```

This is the correct RISC-V assembly language code as it uses the registers to store and retrieve the values needed for calculation.

Memory Operands

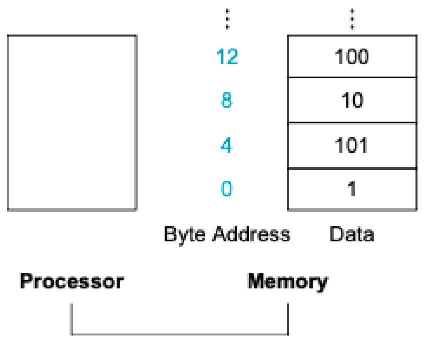
Programming languages can have simple single data elements, as shown in the examples above, but they also can have more complex data such as arrays and structures. How can these be represented in the assembly language?

The processor can only hold a small amount of data in its registers, however, the computer memory has billions of data cells. Data structures for this reason are stored within the computer memory.

Since arithmetic operations can only occur within the registers of the processor, there must be some method by which RISC-V can transfer data from memory to the registers.

Data Transfer Instruction – a command that moves data between the memory and the CPU registers.

Address – a value used to define the location of a specific data element within the memory array.



Memory can be thought of as a large single dimensional array, where the address is like a specific index of the array, starting from 0. From the picture above it can be seen that the data in index [2], which has a byte address of 8 in the memory, would be 10.

The data transfer instruction is usually referred to as load. To load data into a register, the "load word" instruction would be used.

Example:

Assume that there is an array A which consists of 100 words and is assigned to the register x22, and that the variables g and h are associated to the registers x20 and x21. If the code is C is g = h + A[8], what is the compiled code in RISC-V?

Since one of the operands, A[8], resides in the memory, it must first be transferred to a register, for example register x9. This is done by the following notation,

```
lw x9, 32(x22)      // Temp reg x9 takes value of A[8] from memory
```

Here, the word is being loaded into the register x9 and it is important to realize the notation that A[8] takes in the code. The notation for the array is defined as,
Byte Address(Array Register). The Byte Address can be determined by taking the desired index and multiplying it by 4, and adding on the base Byte Address,

$$\text{Byte Address} = \text{Index} \times 4 + \text{Base Byte Address}$$

This is where the 32 comes from in the notation. The Base Byte Address refers to the first Byte Address, and in this case, since it is 0, the Byte Address for A[8] will simply be, $8 \times 4 = 32$.

Now that the data from memory is stored in a register, x9, the rest of the arithmetic can be carried through. The full code would be,

```
lw x9, 32(x22)      // Loads A[8] into register x9

add x20, x21, x9      // Places sum of h + A[8] into g
```

With that, the statement is completely compiled from C to RISC-V.

Example:

Now, take the same example above, and instead of storing the value into g, store it into the array at index 12, A[12].

Here, there is not much different from the previous problem other than the fact that now we need to store the value into memory as well. The first steps will be the same, except the addition will go to the temp register instead of one for the variable g.

```
lw x9, 32(x22)      // Loads A[8] into register x9

add x9, x21, x9      // Places sum of h + A[8] into register x9
```

The last thing to do is store the value of register x9 into A[12]. Using the technique to define the operand for A[8], the same can be done for A[12], $12 \times 4 = 48$, A[12] $\implies$ 48(x22). The value can then be stored using the "store word" instruction, sw. The full code would then be:

```
lw x9, 32(x22)      // Loads A[8] into register x9

add x9, x21, x9      // Places sum of h + A[8] into register x9

sw x9, 48(x22)      // Places x9, h + A[8], into A[12]
```

Many programs have more variables than the computer has registers and for this reason, the computer must move variables around between registers and memory. The most important variables are kept in the registers and the rest are stored in memory. The processes of keeping less frequently used variables in memory is called *spilling registers.*

Constant or Immediate Operands

In many cases, there will be a constant that is being used within a program and in fact, this is the most common case which means, it must be made as fast as possible. It would not make sense to load the constant into a register of its own just for operation, that would take too long instead, immediate operations are used.

These operations are used in the case that there is a content which is easy to add on and does not require its own dedicated register. If say we are adding 4 to the contents of the register x22, the code would simply be,

```
addi x22, x22, 4    // x22 = x22 + 4
```

By including constants as operands to arithmetic instructions, the operations are much faster and require less energy than if they were being loaded in from memory.

The constant 0 has a specific role as it is used to simplify the instructions by offering useful variations. For example, a number can be negated by using the subtraction instruction with 0 as the first operand. For this reason RISC-V dedicates the first register, x0, to always equal 0. This is another example of making the common case fast.

Representing Instructions in the Computer