Computer hardware

The *memory* holds the bits associated with the current state of the program; for example, in word-processing software, the memory would hold the text being edited, along with where you currently are in the document, where you have your windows, where the cursor lies, what's currently on the clipboard, and much more. To get at this data, the computer gives each byte of memory a unique *address*, a number between 0 and the number of bytes in memory.

The memory is typically located in chips outside the CPU. Because electricity takes time to travel, memory is relatively slow. So the CPU also contains a set of *registers*. These are very fast locations on the CPU used for computation and temporary data storage. They are distinguished from memory by their physical location, their quick speed, and their small number. As the CPU executes, it shuttles data between memory and the registers.

Besides data, the memory also holds the program itself. A program is a set of *instructions*. An instruction is a different data type, a sequence of bits that encodes what the machine should do. Different types of processors use different ways to encode instructions; each is called a *machine language*.

The DLX machine language

We examine a particular machine language called *DLX*, designed by John Hennessy and David Patterson for their popular textbook about computer architecture. DLX is a 32-bit machine. The DLX CPU includes 31 registers, labeled r1 through r31, plus r0 which is always 0 and cannot be changed. Each holds 32 bits.

Every DLX instruction fits into four bytes. One example is the instruction

$$001000\ 00010\ 00000\ 0000000000000010\,.$$

When the DLX machine decides to execute an instruction, it reads the instruction from memory and breaks it into pieces. The above 32-bit instruction it would decode as follows.

001000	00010	00000	0000000000000010
op	destination	source	number
code	register	register	
(`addi`)	(r2)	(r0)	(2)

instruction	arguments	what it does
`addi`	rd, ri, n	puts into rd the sum of ri and n
`mul`	rd, ri, rj	puts into rd the product of ri and rj
`mod`	rd, ri, rj	puts into rd the remainder of dividing ri by rj
`slti`	rd, ri, n	puts into rd 1 if ri is less than n, 0 otherwise
`sgt`	rd, ri, rj	puts into rd 1 if ri is greater than rj, 0 otherwise
`lw`	rd, a(ri)	puts into rd the word of memory beginning at address a + ri
`sw`	a(ri), rs	puts the contents of rs into the word of memory beginning at address a + ri
`bnez`	ri, n	if ri is not zero, get the next instruction for execution n bytes beyond the next instruction in memory
`trap`	n	tells DLX to stop executing the program if n is 0

Table 0.1: A subset of DLX operations.

In this case, the first six bits represent the *op code*. This tells what should be done in this instruction. In this case, the op code is 001000, which on DLX represents an `addi` operation, which means that the sum of the number currently in the source register and the number encoded in the instruction should be placed into the destination register. In this case the source register is r0, which is always 0, so the DLX machine will add 2 to 0 and place the sum (2) into the destination register r2.

The DLX machine language specifies a number of different operations with corresponding op codes. We will see a larger sample of DLX operations soon, but first we want to represent instructions in a way that is easier for humans to read.

Assembly language

Although bit sequences are great for computers, they are hard for humans to use. For this reason we typically write low-level programs in *assembly language*, which has a straightforward translation to machine language. The above machine instruction would have an equivalent assembly instruction of “`addi r2,r0,#2`”, where *addi* is short for *ADD I*mmediate. (The number in the instruction is called an *immediate* because it is immediately available from the instruction.)

A human using assembly language would create a file of these instructions and feed them to a program called an *assembler*. The assembler translates each line of assembly language into the equivalent machine language instruction. Then the machine could use this to run the program.

Of course DLX can do many things besides just `addi` instructions. Some of these are tabulated in Table 0.1. DLX operations fall into three categories: arithmetic, memory-access, and branch.

The first category contains the arithmetic instructions you would expect—`addi`, `mul`, `mod`—plus some that you might not expect—`slti` (*S*et if *L*ess *T*han *I*mmediate) and `sgt`. The purpose of these is relatively straightforward; on DLX they always work only with the numbers in registers or contained in the instruction itself.

The second category of operations includes the memory-access operations. The memory-access operations provide the mechanism for using data in memory. They can copy data from memory into registers (like the `lw` (*L*oad *W*ord) operation) or copy data from registers into memory (like the `sw` (*S*tore *W*ord) operation). The memory address is calculated in a slightly funny way: It is computed as the sum of an immediate and a register. This turns out to be

```
lw   r1, 128(r0)  ; load into r1 the number we would like to test (n)
slti r4, r1, #2   ; if the number is less than 2, we want to say it is not prime
bnez r4, #28      ;   in this case, skip 7 instructions to first sw instruction
addi r2, r0, #2   ; let r2 be the divisor we would like to test (i)
mul  r3, r2, r2   ; let r3 be i squared
sgt  r4, r3, r1   ; if this is greater than n, then we can stop; n is prime
bnez r4, #20      ;   in this case, skip 5 instructions to second sw instruction
mod  r3, r1, r2   ; let r3 be the remainder of n divided by i
bnez r3, #-20     ; if it is not zero, step i and go back to mul instruction
addi r2, r2, #1   ;   we add one to i first (this is the delay slot)
sw   132(r31), r0 ; n is not prime; store 0 in memory
trap #0           ; end the program
sw   132(r31), r4 ; n is not prime; store 1 in memory (we got here because r4 was 1)
trap #0           ; end the program
```

Figure 0.1: Prime-Test-All in DLX assembly.

convenient for large programs.

The final category are the *branch operations*, like `bnez` (*B*ranch if *N*ot *E*qual to *Z*ero). Normally, a machine executes instructions in sequence. After doing one instruction, it loads the next instruction in memory, four bytes past the last instruction, and performs it. Then it loads the next instruction and performs it. And so on. Occasionally, however, we want the machine to repeat some instructions or skip some instructions. The `bnez` permits this by telling the DLX machine to load the next instruction n bytes beyond the next instruction in memory if the given register is not equal to zero. (To go backward, we let n be negative.)

A slightly unusual feature of DLX is that whether or not a branch is taken, it always executes the instruction just after the branch instruction. This instruction is in the *delay slot* of the branch. DLX includes this so the CPU can begin executing the instruction without waiting to see whether to take the branch. (Many modern processors go faster by *pipelining*, working on several instructions in advance. Delay slots help.)

How does a real program look in assembly language? Table 0.1 has an assembly-language implementation of Prime-Test-All, which you should puzzle out to see how it works.