CSC 371 homework 1
due 26 Sept, 1996

Dr. Bloch

September 30, 1996

1.8 Two instructions are fetched, taking 20 time units.
Four operands are loaded, taking 40 time units.
Two results are stored, taking 20 time units.
Two instructions are decoded, taking 2 time units.
Two operations are executed, taking 2 time units.
The program counter is updated before each of the two operations, taking 2 time units.
Total: 86 time units.

Alternatively, if we think of the program counter update as taking place *concurrently* with the instruction decode, the total time becomes 84 time units.

1.11 A compiler can be written in machine language, but this is such an extremely time-consuming and error-prone process that most compilers are written in high-level languages. This, however, presents a problem of infinite regress: where did the compiler for *that* high-level language come from?

Suppose you want to write a compiler for Machine X. If another compiler (presumably for a different high-level language) is already available for Machine X, you can use that. If not, find a Machine Y for which a good compiler is available, and modify that compiler into a "cross-compiler" for Machine X: that is, the compiler runs on Machine Y, but produces executable code for Machine X. Then you can write your new compiler, compile it on Machine Y, move the executable code onto Machine X, and run it.

1.12 One reason for not delaying the PC update to the end of the unconditional branch instruction is that it destroys some of the symmetry of the implementation: in every instruction *except* the unconditional branch, the PC should be updated to point to the next instruction. It turns out often simpler to build hardware that *always* updates the PC in the same way early on in the instruction, and updates it again later if necessary.

Another reason has to do with interrupts, exceptions, and error-handling. When an interrupt or exception occurs during execution of an instruction, it is handled by a piece of operating system code called a "trap handler" which assumes the PC points to the instruction after the one that caused the error. If unconditional branch instructions didn't update the PC until the end, and if an interrupt occurred in the middle of one, the trap handler might mistakenly think the interrupt had occurred in the middle of the *previous* instruction.

1.13 The simplest reason is that people run programs on their own computers much more often than they write (or test) programs to be used on other computers. It's much more useful to know that all of a certain set of programs (*e.g.* those that run on a less powerful model) will run on this computer than it is to know that all programs that run on this one will run somewhere else as well.

Newly introduced products are generally more powerful than their predecessors, and often upward compatible with them. So another reason for advertising upward compatibility rather than downward compatibility is that it encourages customers to buy the latest, often most expensive, product.

In addition, it is a fact of human nature that people get new stuff more often than they throw away old stuff: for example, people need bigger and bigger homes as they grow older, just to hold all their stuff. Similarly, once you've become accustomed to a particular computer feature, you're more likely to add new features than to throw away old ones, which means your need for computer power will increase over time. As a result, people upgrade their computer systems much more often than they downgrade them, and upward compatibility accomodates these people.

1.14 There is no clearly "right" answer to this question; all of these characteristics are considerations in comparing two computers. If I had to pick one, however, I would probably choose "number of instructions executed per second."

- The number of instructions in a computer's instruction set indicates how flexible its assembler language is, and how much work can be done in one instruction. As such, more instructions are better than fewer instructions, *unless* implementing the additional instructions adds significant cost or significantly decreases the number of instructions executed per second. The RISC philosophy holds that a few simple instructions with an inexpensive, fast implementation are better than a wide variety of powerful instructions that can only be implemented on an expensive, slow processor.
- The size of memory determines how big a program and/or data set the computer can handle at a time. Many modern computers have virtual memory, so exceeding this doesn't crash the computer, it merely causes a degradation in performance.
- The number of instructions executed per second is a reasonably good indicator of the speed of the computer, but not entirely a valid way to compare two computers unless they have the same instruction sets: a RISC with very fast, but simple, instructions might *or might not* solve the problem faster than a CISC with slow but powerful instructions.
- The size of a particular program affects how much memory you'll need to run it. Not terribly important these days, since memory gets less expensive every year and virtual memory is widespread.

- The popularity of the computer architecture tells you how many other people like it, and is therefore an indirect measure of quality. Unfortunately, it also measures marketing skill and business clout, neither of which has much bearing on the quality of the computer itself. Popularity has another benefit, however: a popular computer is likely to have a lot of software available to run on it.

2.3

```
          .data
z:        .word     15           # or whatever
a:        .word
i:        .word
sum:      .word
          .text
          move      i,2
for:      bgt       i,z,endfor
          rem       a,i,2
          bnez      a,incr_i
          add       sum,sum,i
incr_i:   add       i,i,1
          b         for
endfor:
```

2.7

```
          .data
n:        .word
msg:      .asciiz   "How many X's?  "

          .text
__start:  puts      msg
          get       n
          blez      n,finish
again:    put       'X'
          sub       n,n,1
          bgtz      n,again
finish:   put       '\ n'
          done
```

2.12 The "other possible way" given in the text is one instruction shorter, both in program size and in number of instructions executed. There is, however, one very good reason not to use it: if $A = 0$, it causes a divide-by-zero exception, even though in the Pascal code if $A = 0$ no division is performed at all. If you can guarantee that A will not be equal to zero, this alternative code is more efficient.