Chapter 1 sample problems.

1. A quad core processor could speed up a computer by a factor of 4 but this rarely happens. Use Amdahl's Law to compute the percentage of program execution that needs to be distributed across all 4 cores to achieve an overall speedup of 3. Of 2. Of 1.5. Of 1.25.

   Answer: We want to solve for f in S = (1 / (1 – f + f / 4)) where S is 3, 2, 1.5 and 1.25. This involves a little algebra but we wind up with f = 4 / 3 * (1 – 1 / S). For S = 3, f = .889. For S = 2, f = .667. For S = 1.5, f = .444. For S = 1.25, f = .267. So to achieve a speedup of 1.25, all four cores must be in use about 26.7% of the time but to achieve a 3 time speedup, all four cores must be in use 88.9% of the time.

2. A benchmark has a breakdown of the following:
   39% loads, 12% stores, 28% ALU operations other than multiplies/divides, 6% multiplies, 2% divides, 10% conditional branches, 3% unconditional branches
   A processor has a CPI of 5 for loads/stores, 3 for all ALU operations other than multiplies/divides, 6 for multiplies, 15 for divides, and 4 for branches. We are considering making one of several enhancements to the ALU. Which should we make?
   a. Improving the condition tester so that conditional branch CPI is reduced to 2
   b. Improving the multiplier so that the CPI for multiplies is reduced to 3
   c. Improving the divider so that the CPI for divides is reduced to 7
   d. Improving the ALU so non-multiply/divide operations have a CPI of 2

   Answer: We compute the speedup as CPU time old / CPU time new. CPU time = Clock Cycle Time * IC * CPI. Clock Cycle Time does not change, but IC * CPI does for each instruction. So let's compute the old and the new IC * CPI. Old IC * CPI = (.39 + .12) * 5 + .28 * 3 + .06 * 6 + .02 * 15 + (.10 + .03) * 4 = 4.57.
   a. New IC * CPI = (.39 + .12) * 5 + .28 * 3 + .06 * 6 + .02 * 15 + .10 * 2 + .03 * 4 = 4.37. Speedup = 4.57 / 4.37 = 1.046.
   b. New IC * CPI = (.39 + .12) * 5 + .28 * 3 + .06 * 3 + .02 * 15 + (.10 + .03) * 4 = 4.39. Speedup = 4.57 / 4.39 = 1.041.
   c. New IC * CPI = (.39 + .12) * 5 + .28 * 3 + .06 * 6 + .02 * 7 + (.10 + .03) * 4 = 4.43. Speedup = 4.57 / 4.43 = 1.032.
   d. New IC * CPI = (.39 + .12) * 5 + .28 * 2 + .06 * 6 + .02 * 15 + (.10 * .03) * 4 = 4.29. Speedup = 4.57 / 4.29 = 1.065

3. Consider an integer benchmark with a breakdown of instructions of 50% load/store, 40% ALU and 10% branch and a CPI of 5 for loads and stores and 4 for ALU and branch. Given that the benchmark uses only int registers, the FP registers are not used at all. In such a case, imagine that the compiler is able to freely use the FP registers to store integer values, thus reducing the number of loads and stores in the program. The CPU time does not change but the IC * CPI does as some instructions go from load/store to ALU. Assuming the compiler can successfully reduce 20% of the loads/stores, what is the speedup? Use both the CPU Time formula and Amdahl's Law.

   Answer: Using IC * CPI is easier, we'll start with it. Original CPU Time = CPU Clock Cycle Time * IC * CPI = CPU CCT * (.50 * 5 + .50 * 4) = CPU CCT * 4.5. New CPU Time = CPU Clock Cycle Time * (.50 * .80 * 5 + .50 * 1.20 * 4) = CPU CCT * 4.4. Speedup = 4.5 / 4.4 = 1.0227 (2.27 % speedup). Using Amdahl's Law, the speedup k = 5 / 4 = 1.25. But f is not 20% as this does not factor in the actual time the enhancement is in use since it is 20% of the change in clock cycles. So f = .20 * .50 * 5 = .5 clock cycles out of 4.5, or .5 / 4.5 = 11.1%. Now, using Amdahl's Law we have S = 1 / (1 – .111 + .111 / 1.25) = 1.0227.

4. Let's compare a CISC machine versus a RISC machine on a benchmark. Assume the following characteristics of the two machines.

   CISC: CPI of 4 for load/store, 3 for ALU/branch and 10 for call/return, CPU clock rate of 3.5 GHz

   RISC: CPI of 1.3 (the machine is pipelined, the ideal CPI is 1.0, but overhead and stalls make it 1.3) and a CPU clock rate of 1.75 GHz

   Since the CISC machine has more complex instructions, the IC for the CISC machine is 30% smaller than the IC for the RISC machine

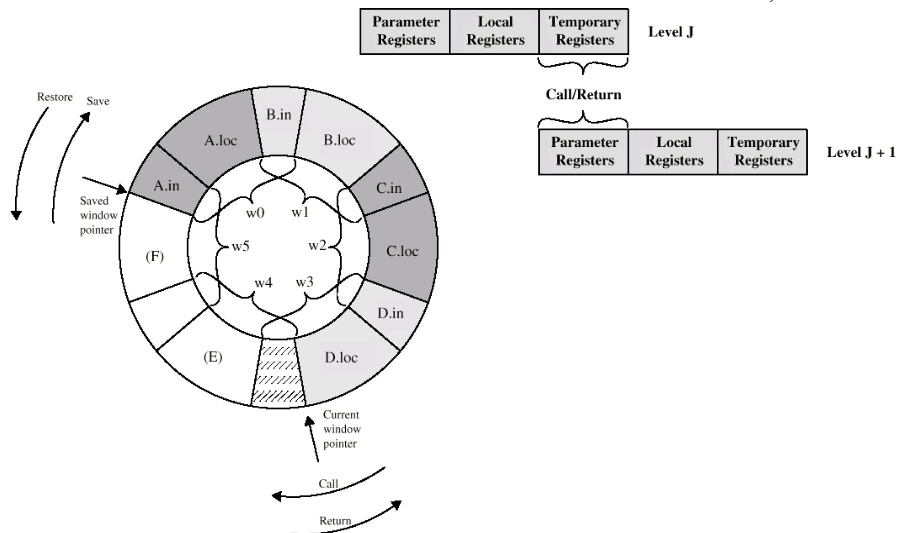
   The benchmark has a breakdown of 38% loads, 10% stores, 35% ALU operations, 3% calls, 3% returns and 11% branches.

   Which machine will run the benchmark in less time and by how much?

   Answer: use CPU time = IC * CPI * Clock cycle time

   RISC: IC_{RISC} * CPI_{RISC} * Clock cycle $time_{RISC}$ = IC_{RISC} * 1.3 * 1 / 1.75GHz = 0.743 * IC_{RISC}
   CISC: IC_{CISC} * CPI_{CISC} * Clock cycle $time_{CISC}$ = IC_{CISC} * (4 * .38 + 4 * .10 + 3 * .35 + 10 * .03 + 10 * .03 + 3 * .11) * 1 / 3.5 GHz = IC_{RISC} * 0.7 * 3.96 / 3.5 GHz = 0.792 * IC_{RISC}
   Thus, the RISC machine is faster on this benchmark by 0.792 / 0.743 = 1.066 or about 6.6%.

5. Most computers pass parameters using a run-time stack stored in memory. This means any function call and return requires potentially several memory accesses. An alternate architecture, Berkeley RISC, uses register windows. With register windows, local variables that are passed as parameters are placed in a set of registers that *overlap* registers between the registers of the calling function and the called function. This "overlap" is a window. It allows parameter passing without the use of memory. See the figure below. Register windows replace memory access operations so IC remains the same but CPI changes as register operations require fewer clock cycles than memory operations. Use for CPI: Loads/stores: 4, ALU and unconditional branches: 2, conditional branches: 3, procedure calls and returns: 15. Architects are trying to decide whether to use additional registers in the CPU for register windows or a larger register file. Enlarging the register file reduces the number of loads and stores by 40% and 30% respectively. Register windows reduces procedure call CPI to 4.5 and returns to 3. Given a benchmark of 40% load, 13% store, 31% ALU, 8% conditional branches, 2% unconditional branches, 3% procedure call and 3% return, should we use the added registers for a register window or a larger register file?



Answer:

CPU Time = IC * CPI * Clock Cycle Time. The last value will not change between the two approaches. If we use register windows, CPI reduces and if we add more registers, IC reduces because of fewer loads & stores.

$CPI_{original}$ = .40 * 4 + .13 * 4 + .31 * 2 + .08 * 3 + .02 * 2 + .03 * 15 + .03 * 15 = 3.92.
$CPI_{regwindows}$ = .40 * 4 + .13 * 4 + .31 * 2 + .08 * 3 + .02 * 2 + .03 * 4.5 + .03 * 3 = 3.245.
In adding the new registers to the register file, we have to recompute the breakdown of instructions as we have 40% fewer loads and 30% fewer stores.

.40 * .40 = .16, so 16% fewer loads
.13 * .30 = .039 so 3.9% fewer stores

This results in .16 + .039 = .199 fewer instructions. Now we recompute the breakdown of instructions given an IC of 1.00 - .199 = .801

Loads = (.40 - .16) / .801 = .300
Stores = (.13 - .039) / .801 = .114
ALU = .31 / .801 = .387
Conditional branches = .08 / .801 = .100
Unconditional branches = .02 / .801 = .025
Procedure calls = .03 / .801 = .037
Returns = .03 / .801 = .037

New CPI = .300 * 4 + .114 * 4 + .387 * 2 + .100 * 3 + .025 * 2 + .037 * 15 + .037 * 15 = 3.89
$IC_{registers}$ = .801 * ICoriginal

CPU Time $_{register\ windows}$ = $IC_{original}$ * 3.245 * CPU clock cycle time = 3.245 * $IC_{original}$ * clock cycle time
CPU $Time_{new\ registers}$ = $IC_{original}$ * .801 * 3.89 * CPU clock cycle time = 3.116 * $IC_{original}$ * clock cycle time

Using the new registers in the register file results in a smaller CPU Time so is faster than using the registers in register windows by 3.245 / 3.116 = 1.041 or a little over 4% faster.

6. In the 1980s and 1990s, architects debated whether the RISC or CISC approach was better. The list below denotes some of the differences in philosophy between the two forms of architecture. For each of the following, explain which is better, RISC or CISC by specifically citing which is improved in the CPU time formula: IC, CPI, Clock Cycle Time (or some combination). Also discuss if other portions of the formula are impacted.
   a. In RISC, there are a greater number of registers available over a CISC processor
   b. In CISC, there can be complex addressing modes such as indirect addressing to obtain data pointed to by pointers
   c. In RISC, instruction pipelines are more effective resulting in fewer stalls
   d. In CISC, variable length instructions are common so that multiple operands can be accessed from memory in one instruction
   e. In RISC, a superpipeline divides the cache access stages (instruction fetch, data access) into multiple stages (for instance, 2 stages for instruction fetch, 3 stages for data access).

Answers:
   a. More registers for a RISC processor means fewer loads and stores, so IC decreases over the corresponding CISC processor. Notice though that in the CISC processor, memory-register operations ultimately can reduce IC as well.

b. The complex addressing modes allow memory accesses in single machine operations whereas in a RISC architecture without complex addressing modes, something like indirect addressing takes multiple operations, therefore this feature lowers IC for CISC processor. More complex addressing modes often requires more time though, so for the CISC processor, CPI might increase.
c. The ideal CPI for a pipelined processor is 1.0. Stalls increase the CPI. Both RISC and CISC processors are able to use pipelines but RISC uses them more successfully meaning fewer stalls and so a lower CPI.
d. By being able to reference more than one memory operand in any instruction reduces IC. The cost though is that CPI increases because multiple memory references (along with a possible ALU operation) takes more time.
e. Since the cache access is the longest of all of the stages, doubling/tripling these stages allows us to increase the clock rate (perhaps by a factor of two). Unfortunately, in doing so, the number of stalls are lengthened, so while clock cycle time decreases, CPI might increase because of longer stalls.

7. Let's see what might happen if we add a register-memory ALU mode to RISC-V (from appendix A). We could replace the two statements

```
lw  x1, 0(x2)
add    x3, x3, x1
```

with

```
add   x3, 0(x2)
```

In order for the new instruction to have enough space in the 32-bit instruction length format to specify the address, we restrict the instruction to being a two-operand instruction where the first operand is a destination register and the second is the memory reference which consists of an offset (limited to 16 bits) and an index register. Assume that to accommodate the memory fetch as part of this instruction, we increase clock cycle time by 15% (because we are lengthening the clock, CPI is not impacted). Using the gcc benchmark (figure A.29, p. A-42), what percentage of loads would have to be eliminated so that this new mode can execute gcc in the same amount of time?

Answer: We want CPU $time_{new}$ = CPU $time_{old}$ or IC_{new} * CPI * CCT_{new} = IC_{old} * CPI * CCT_{old} (CPI does not change). Since CCT_{new} = 1.15 CCT_{old}, we need IC_{new} = IC_{old} / 1.15, so IC_{new} must be 1/1.15 = .87 of IC_{old} (we need to reduce IC by 13%). We are eliminating only load instructions and the benchmark consists of 25.1% loads, so we must remove 13% / 25.1% = 51.8% of the loads! That's a lot of loads.

8. Autoincrement and autodecrement are common addressing modes in CISC computers. These modes are used when accessing array elements by automatically incrementing or decrementing the register storing the offset. The change occurs after the access for the increment, and before the access for the decrement. Let's see what happens in some standard array code with the new mode:

```
for(i=0;i<1000;i++)
        a[i]=b[i]+c[i];
```

Assume that x1, x2, and x3 store the starting addresses arrays a, b, c respectively and that they are all int arrays. If we introduce an autoincrement statement like lwi Xa, 0(Xb) in place of the lw instruction of RISC-V, how will it impact the performance? Below are the two sets of code, without and with the autoincrements. The CPI for our machine is as follows: 5 for loads/stores, 2 for ALU and 3 for branches. The autoincrement load/store allows us to reduce IC. Assuming the new instructions have the same CPI (5) but requires that we lengthen the clock cycle by 25%, is the new mode worth pursuing?

```
addw        x4, x0, x0      // x4 is the loop variable i
```

```
            addiw       x5, x0, 1000     // x5 = 1000
top:        beq         x4, x5, out
            lw          x7, 0(x2)        // x7 = b[i]
            lw          x8, 0(x3)        // x8 = c[i]
            addw        x9, x7, x8       // x9 = b[i] + c[i]
            sw          x9, 0(x1)        // store b[i] + c[i] in a[i]
            addiw       x1, x1, 4
            addiw       x2, x2, 4
            addiw       x3, x3, 4
            addiw       x4, x4, 1
            j           top
out:        ...
```

```
            add         x4, x0, x0       // x4 is the loop variable i
            addi        x5, x0, 1000     // x5 = 1000
top:        beq         x4, x5, out
            lwi         x7, 0(x2)        // x7 = b[i]
            lwi         x8, 0(x3)        // x8 = c[i]
            add         x9, x7, x8       // x9 = b[i] + c[i]
            swi         x9, 0(x1)        // a[i] = b[i] + c[i]
            addi        x4, x4, 1
            j           top
out:        ...
```

Answer:

We compare the two CPU Times = IC * CPI * Clock Cycle Time. The original machine has a shorter Clock Cycle Time while the newer machine has a reduced IC * CPI because we can remove three of the addi instructions. NOTE: we can't just compare the two ICs because instructions have different ICs. The reduction in ALU operations impacts IC * CPI as a total.

$\text{CPU Time}_{original} = \text{IC} * \text{CPI} * \text{Clock Cycle Time}_{original}$
$\text{CPU Time}_{new} = \text{IC}_{new} * \text{CPI}_{new} * \text{Clock Cycle Time}_{new}$

The original code has 2 ALU operations before the loop plus a loop of 6 ALU, 2 branch and 3 load/store. This gives us a total IC * CPI of 2 * 2 + 1000 * (6 * 2 + 2 * 3 + 3 * 5) = 33,004 clock cycles.

The new code has 2 ALU operations before the loop plus a loop of 3 ALU, 2 branch and 3 load/store increment. This gives us a total of IC * CPI = 2 * 2 + 1000 * (3 * 2 + 2 * 3 + 3 * 5) = 27,004.

$\text{Clock Cycle Time}_{new} = \text{Clock Cycle Time}_{old} * 1.25$

$\text{CPU Time}_{old} = 33{,}004 * \text{Clock Cycle Time}_{old}$
$\text{CPU Time}_{new} = 27{,}004 * \text{Clock Cycle Time}_{new} = 27{,}004 * \text{Clock Cycle Time}_{old} * 1.25$

Speedup = $\text{CPU Time}_{old} / \text{CPU Time}_{new}$ = 33,004 / (27,004 * 1.25) = 0.978, so we see a slowdown, not a speedup.

9. As an alternative to slowing down the clock from #8, let's assume that the clock speed does not change, but that the CPI for the lwi and swi is 6 instead of 5. Is the change worth it?

   Answer: We only have to compare IC * CPI for both machines. The old machine's IC * CPI does not change (remains 33,004). The new machine has the following IC * CPI = 2 * 2 + 1000 * (3 * 2 + 2 * 3 + 3 * 6) = 30,004. Since this is a reduction, the new mode would be worth it in this case. The speedup is 33,004 / 30,004 = 1.10.