CPSC 261 Sample Midterm 1
February 2016

[8] 1. Short Answers

[2] a) List two techniques for coping with complexity, and briefly describe each of them (one or two sentences suffice).

Solution : Here are the techniques we would accept as answers:

- Modularity
- Abstraction
- Robustness drives independence
- Layering
- Hierarchy
- Indirection
- Iteration

Descriptions can be found in the textbook and will not be rewritten here.

[2] b) Give two reasons why coherency and atomicity are both desirable properties of the *memory* abstraction that are difficult to achieve.

Solution : They are difficult to achieve because of:

- Concurrency
- Values can occupy multiple cells
- Replicated storage
- Performance enhancements

[2] c) When evaluating the performance of a pipelined CPU, should you look at the latency of the pipeline, or at its throughput? Why?

Solution : You should look at the throughput: it tells you how many instructions per second the pipeline can execute. How long one instruction takes isn't all that relevant.

[2] d) What would happen to your program if you ran it on a CPU whose pipeline violates sequential consistency?

Solution : It would produce incorrect results (or crash).

[8] 2. More short answers

[2] a) What technique does the Intel core CPUs utilize when confronted with a very complicated instruction, such as `movq 0x40(%rsi, %rdi, 4), %rbx`, in order to make it easier to execute on its pipeline?

Solution : It breaks every instruction down into micro-instructions (smaller, simpler instructions) and these micro-instructions are what's executed by the pipeline.

[2] b) Core memories and DRAM have one property in common, that relates to the way in which they work. What is it?

Solution : Reading a bit destroys its value, and thus the memory controller needs to rewrite it as soon as its value has been read.

[2] c) Most of the memory in a computer is much slower than the CPU. What property of well-written programs allows us to make it appear as if most of it were actually fast?

Solution : Locality: well written program repeatedly access the same location in memory, or locations near one another.

[2] d) Which area of memory does the memory returned by a call to `malloc` belong to, and who manages it?

Solution : It belongs to the *heap*; the user's program (not the compiler or operating system) manages it, with help from the memory allocation functions (`malloc`, `calloc`, `realloc` and `free`).

[5] 3. Each of the following two pieces of code has a bug that will cause it to *possibly* not behave the way the comments indicate it should, or fail to interact properly with the rest of the program. In each case, explain (1) what the problem is and (2) how you would fix the bug. Recall that registers `%rbx`, `%rbp`, and `%r12` to `%r15` are callee saved. All other registers are caller saved.

[3] a)
```
test1:
  leaq (%rdi, %rdi, 2), %rdi # multiply parameter "n" by 3
  call test3                 # compute something with 3n
  addq %rdi, %rax            # add 3n to test3's return value
  ret
```

Solution : Register `%rdi` is caller saved, and so we can not rely on the fact that function `test3` preserves its value. The value being added to `%rax` could be anything. One solution is to push `%rdi` onto the stack immediately before the `call` instruction, and to pop it back immediately before the `addq` instruction.

[2] b)
```
test2:
  leaq (%rdi, %rdi, 2), %r15 # save 3n into %r15
  movq (%rsi, %r15, 8), %rdi # load array[3n] into %rdi
  call test4                 # compute something with it
  addq %r15, %rax            # add 3n to test4's return value
  ret
```

Solution : Register `%r15` is callee saved. The function that called `test2` expects its value to be preserved by `test2`, so we should push it onto the stack at the beginning of `test2`, and pop it back right before the `ret` statement.

[10] 4. Using what you know about pipelined processors, explain why:

[2] a. conditional branches often decrease throughput.

Solution : They often decrease throughput because, if the CPU guesses the outcome of the branch incorrectly or stalls instead of guessing, then cycles are spent not executing useful instructions.

[2] b. dependencies between instructions may decrease throughput.

Solution : Dependencies will often result in pipeline stalls.

[2] c. a pipeline with more stages is capable of executing more instructions per second than one with fewer stages (under the right conditions).

Solution : There are more instructions executing in parallel (one per pipeline stage under optimal conditions).

[2] d. if we keep adding pipeline stages, we will eventually get to the point where additional stages no longer increase throughput.

Solution : Programs only have a limited amount of concurrency. If we have too many stages, then there may not be enough instructions to use them all, due to dependencies between the instructions.

[2] e. using the pipeline to execute two threads (alternating between them) increases throughput compared to first executing one thread, and then the other.

Solution : Instructions that belong to different threads do not have dependencies, unlike instructions from the same thread.

[7] 5. Consider the following C function:

```
void do_something()
{
    char *mtl = malloc(5 * sizeof(char));
    char *qc;
    char **cdn = &qc;

    mtl[0] = 'Y';
    mtl[4] = '\0';
    qc = mtl + 2;
    *qc = 'A';
    qc[-1] = 'P';
```

```
    *cdn = qc + 1;
    **cdn = 'Z';
    *(qc - 1) = 'E';
    printf("%s\n", mtl);
}
```

[5] a) What characters will the call to `printf` print to the screen? Justify your answer by drawing pictures of the array and of the pointers.

Solution : Here is a line by line explanation of the program's behaviour:

i. `mtl[0] = 'Y'` and `mtl[4] = '\0'` are self-explanatory.
ii. `qc = mtl + 2` assigns to `qc` the address of `mtl[2]`.
iii. `*qc = 'A'` is thus the same as `mtl[2] = 'A'`.
iv. `qc[-1] = 'P'` is the same as `mtl[1] = 'P'`.
v. `*cdn = qc + 1` assigns the address of `mtl[3]` to `*cdn`. Because `cdn` was initialized to point `qc`, this means `qc` now points to `mtl[3]`.
vi. `**cdn = Z` now assigns 'Z' to `mtl[3]`.
vii. `*(qc - 1) = 'E'` finally assigns 'E' to `mtl[2]` (don't forget the statement `*cdn = qc + 1` had modified `qc`).

Therefore the function prints the string "YPEZ".

[2] b) What additional statement(s) should be added at the end of this function, and why? Note: the answer is not "a return statement".

Solution : We should add the statement `free(mtl);` to release the memory used by the string.

[7] 6. Consider the following sequence of instructions written in x86_64 assembly language:

```
1  function:
2    movq   %rdi, %rax
3    subq   %rsi, %rax
4    jle    done
5    movq   %rsi, %rbp
6    movq   $2,   %rax
7    addq   %rdi, %rbp
8    subq   %rax, %rbp
9    movq   0x1000(%rsi), %rax
10   andq   %rax, %rbp
11   pushq  %rdi
12   movq   %rbp, %rdi
13   call   function
14   popq   %rdi
15   call   otherfunction
16 done:
17   popq   %rbp
18   ret
```

List the seven of the eleven hazards that are present in this sequence of instructions. For each hazard:

- If it is a data hazard, write `D`, and both the number of the instruction that uses the data value, and the number of the instruction that computes it. If an instruction uses several values computed by earlier instructions, list only the number of the most recent such instruction. One of the data hazards is fairly hard to find.

Do not worry about the possible existence of data hazards where only one of the two instructions involved is shown here (for instance, if the second instruction is in `otherfunction`).

- If it is a control hazard, write `C` and the number of the instruction where the hazard occurs.

Solution : I will list all eleven hazards. For data hazards, I indicated the register on which the hazard occurs.

1. D, 2, 3 (%rax)
2. C, 4
3. D, 5, 7 (%rbp)
4. D, 7, 8 (%rbp)
5. D, 9, 10 (%rax)
6. D, 10, 12 (%rbp)
7. D, 11, 13 (%rsp)
8. D, 12, 2 (%rdi)
9. D, 14, 15 (%rsp)
10. D, 17, 18 (%rsp)
11. C, 18