Advanced Computer Architecture
COL718 and ELL782
Minor 1 – Fall 2018

- Just writing the answer is not sufficient. You need to thoroughly justify your answer with appropriate proofs.
- When you are creating a mechanism, describe it in great detail – down to the last bit. Note that you are designing hardware. The only structures that you can have in hardware are registers to store int/float variables, arrays of registers, logic gates, state machines, and arithmetic circuits.
- Answers should be very brief and to the point. In every question, we are only looking for the main idea. Do not waste your time in describing things that are irrelevant. Enclose key statements in your answer with a box. Write short answers and use mathematical symbols and terms to reduce the amount of text.
- No pencils, calculators, cell phones, smart watches, or bags allowed.
- No questions will be answered.

90 minutes
Each question is 10 marks – Total 60 marks. This paper contains 2 pages.

1. Assume we have an OOO processor with a PRF (physical register file). Further note that there is no need to put an entry in the instruction window when an instruction does not have any register sources or destinations. Additionally, there are some load instructions that can take a very long time to return from memory. Given that we have 128 physical registers, what is the maximum possible size of the instruction window? In such processors, it is typical to have a large ROB. For example, the ROB in this case (with 128 physical registers) can be sized to contain 160 entries. Why is this the case?

2. How can we detect a loop in hardware? Is it possible for the compiler to do some analysis and help the hardware in predicting the branches in loops? Explain your answer.

3. In an OOO processor, the compare (*cmp*) instruction is used to compare the values in two registers. The result is saved in a *flags* register (not accessible to software). Subsequent branch instructions use the value of the *flags* register. Will the same mechanism work in an OOO pipeline? If not, then how do we augment it to support this functionality in the ISA?

4. Assume that we want to create a scheme where we try to allocate physical registers uniformly. This means that it shall not be the case that after billions of cycles some physical registers have been used many times, and some physical registers have been very infrequently used. How can we modify the free list to support this feature?

5. Do we need bypass and dependence check logic while accessing the register file in the PRF based design? Explain its need with examples (if required). If yes, then provide the details of an implementation.

6. Consider the code for binary search.

```
int bin_search(int arr[], int left, int right, int val){
        /* exit conditions */
        int mid;
        if (right < left) return -1;
        mid = (left + right) / 2;
        if(val == arr[mid])
        return mid;

        /* recursive conditions */
        if(val < arr[mid])
                return bin_search(arr, left, mid - 1, val);
        else
                return bin_search(arr, mid + 1, right, val);
}
int main() {
        ...
        result = bin_search(...);
        ...
}
```

Consider the flow of function calls. We keep calling *bin_search* till we reach an exit condition. After that the value is just returned all the way till we reach the *main* function: the function that originally called the binary search function. The important pattern to note here is that we have a series of returns where a value is just returned. It is not processed. This is known as *tail recursion*.

How can we eliminate tail recursion? This means that we want to pass the return value directly to the *main* function and save it in the *result* variable. We wish to avoid the series of return statements. Note that the overhead of a return statement is not just the extra instruction, we also need to restore registers depending upon the method used to store/restore registers on a function call.

We need some compiler support to do this. Describe it. Subsequently, how do we design our hardware to work in consonance with the compiler to implement this efficiently?