UG Exam Performance Feedback
First Year
2018/2019 Semester 1

COMP15111 Fundamentals of Computer Architecture Jim Miles

Comments Please see the attached report.

COMP15111 feedback 2018-19 exam

Question 1

This question was generally answered well, with high average marks. There were a few errors that were made by several students:

a. Where decimal was converted to binary some students read the binary number in the wrong direction and therefore got the wrong answer, some students omitted the final division (1/2 = 0r1), also some students omitted the hexadecimal conversion.

b. The positive number was in general correct (some minor errors calculations), the negative number had the most errors. Some students tried to do the conversion without thinking of it as a 2s compliment representation, some others used the '1' as the sign bit but didn't use that bit also for the conversion (-128).

c. Although the question states “each colour is represented by 8 bits”, many students divided 8bits between 3 colours leading to the wrong answer of 2 bits per color (R,G,B) and 2 unused bits-> 7 6 (5 4) (3 2) (1 0) -> 2^6 . A few students found the correct answer of 2^8 but then multiplied by 3 or calculated 2^8*2^8 *2^8

d. In general this was well answered except that many students miscalculated 2^{12} (some frequent answers were 2048, 5096). It was expected that students would remember some common powers of 2 and be able to determine 2^{12} from that – e.g. 2^{10} = 1024 = 1k.

Question 2.

a. (i) This was a generally well answered question. Most students knew the basic sequence of operations, although the Von Neumann model and how the PC is used in such architectures was not expressed as clearly as the fetch-decode-execute cycle.
(ii) In this question, many students struggled to give a full answer. The students are used to write code with branch instructions and know how branch instructions work, but many answers did not explain how PC relative addressing is used in ARM specifically for the execution of branches.

b. (i) Most students were aware of the fact that Link Register keeps the return address when a function call happens with the BL instruction and that if the return address is not saved, the program is not able to return to the instruction after the function call. However, some answers did not mention how the Link Register is used to update the PC value in order to return from a function call by simply moving the content of the LR into the PC.
(ii) Although this question was simply asking what needs to be done in order to make another function call inside a function call, most of the students assumed it was a complex question regarding nested function calls. Although an answer as simple as “The LR of the first function needs to be saved somehow” would be sufficient, students mostly answered the question explaining how a stack needs to be used.

c. The most common mistake while answering this question was that students compared use of the stack with registers. The question asked for a comparison between the usage of stack and other memory allocation techniques. Since registers are not a part of memory, this comparison resulted in wrong answers. Another common mistake was that students claimed the stack was faster than memory, whereas the stack resides in memory and in order to push or pop data a memory access is necessary.

Question 3.

a. The majority of the answers were correct. Some had issues with identifying the purpose of the given function, hence lost a few marks but overall the understanding of ARM instructions was good.

b. Most answers struggled to identify different erroneous cases. There were answers which stated negative (BLT/BGT) and zero (BEQ/BNE) values in variables as separate cases, rather than efficiently combining into BLE. Most didn't consider checking for overflow.

c. Overall the answers to this question were good. In some cases marks were lost by trying to change the branching conditions to make the code efficient but creating a more complicated structure/more instructions. Some answers made ADD/SUB conditionally executed but retained B loop so that the number of instructions were the same.

Question 4

a. The marking scheme assumes that the answer will describe a 2-pass assembler process but answers structured around the sequence of lexical analysis, syntactic analysis, semantic analysis and code generation were equally acceptable. Most students gave answers in the 2nd form, and whilst these generally secured good marks the answers were rather succinct and in many cases more of the detail that is in the lecture slides should have been given. Some explanation of the need to construct a symbol table and the consequent need for multiple passes should be included in the answer and this important aspect of the process of assembly was omitted in many answers.

b. The difference between instructions and pseudo instructions was generally not well expressed. Instructions are operations that are a part of the instruction set of the processor, while pseudo-instructions are operations that do not exist in the instruction set of the processor but are a part of the assembly language and can be converted by the assembler into one or more instructions. Many of students wrote that a pseudo instruction is one that would be converted into more than one instruction, which is not necessarily the case. In many cases "compiler" was incorrectly used when describing an assembler. Many people also wrote that an instruction is one that can be assembled directly by the assembler, but that is also true for several pseudo-instructions in that the assembler can convert them directly into an instruction that is part of the instruction set. A few people gave MUL as an example of a pseudo-instruction but in ARM 7 there is a MUL instruction within the instruction set (MUL is not one of the 16 data instructions, which could be the source of confusion, but it is an instruction). A number of people referred to (object code or machine code or binary code) as byte code. This is not strictly correct since an instruction is 32bits. In the Java Virtual Machine the assembled code is called byte code because each operation is a single byte, but that is not the case in the ARM.

c. Part c) was generally answered correctly, although it was expected that usage would include examples of using constants and variables as well as declaring them.

Question 5

There were some excellent answers to this question but the average mark for Q5 was lower than the other questions and the spread of marks was somewhat greater, with a substantial number of very low marks. Although that might be because it was the last question and people ran out of time, it did seem that many students were less well prepared for this question than for Q1-Q4, even though quite similar self-study questions were included in the lecture slides on VMs.

Some general issues: There was quite a lot of confusion around what the JVM does with byte code. It does not routinely translate byte code into native machine code, it is a software emulation of a virtual machine which executes the byte code. When JIT compilation is used then sections of byte code may be compiled into the machine code of the host which does increase performance when used but that is not really an advantage of using a VM, because compiling Java directly into the host machine code without using a VM at all would produce even faster executables than the JVM compiling byte code.

Byte code is compact and so can be fast to download but it is not fast to execute. It is not trivial to theoretically compare the speed of JVM executing byte code with the real processor executing natively compiled code because the relative speed depends upon the usage of registers that can be achieved in an ARM program and the usage of the stack that can be achieved in byte code, but there are 13 general purpose registers in the ARM whereas the JVM only operates on the top two elements of the stack, so it is likely that the JVM executing byte code will need more load/store operations than the ARM with natively compiled code. Consider ADD – where the ARM can often operate on fast registers the JVM always has to load both operands from memory (the stack) and store the result back in memory (the stack), and is more likely to additionally have to load/store from/to the variable space into the stack than the ARM code would be. The JVM is not designed for speed and it is not faster.

In many cases it was not recognised that the stack is in memory and that therefore the stack is not faster than any other memory access. Using the stack does not avoid using memory, it is using memory.

The JVM itself is not compact. Byte code may be, but the JVM is an entire virtual processor and so it is not a small amount of code.

The main value of the JVM is that it allows developers to compile one version of the code (Java Byte Code) and distribute that to users who can run it on any machine or environment that has a JVM installed. This is particularly easy for developers and users where code is embedded in web pages.

One common issue was not answering the specific question that was asked.

a. The specific benefits of the JVM (as opposed to VMs in general) was not well answered by a number of students. The requirements of code for web applications and the match between the characteristics of the JVM and the requirements of web applications was not explained well.
b. Many answers showed a sequence of operations to perform the addition of two variables but few students used specific JVM mnemonics to show how variables were loaded onto the stack (e.g. iload) prior to addition. Few gave a complete explanation of conditional branching in the JVM and very few got that right.
c. The main problem with answers to part c) was that the advantages were not explained in the context of the requirement of portability between different architectures, the range of different architectures that exist and the consequent advantage of the JVM having no registers.