Comments

COMP22111 Exam Feedback 2015/16

Q1 set by Paul Nutter – 100% attempted
Q2 set by Paul Nutter – 100% attempted
Q3 set by Jim Garside – 89% attempted
Q4 set by Jim Garside – 11% attempted

Overall exam average 53%.

Q1 On the whole this question was answered well. The average mark was 67%.

a) Most got this correct. The key things I was looking for were:
•Stump does not operate on data held in memory, instead it operates on data held in local registers and the result is written back to a local register
•To allow data from memory to be used, Stump provides load and store instructions
•To support this a register bank is provided.
Any marks lost were largely as a result of a lack of detail in the answer.

b)A simple discussion of the difference between Type 1 and Type 2 instruction was required, mainly focusing on where data comes from. Many answers simply stated that the instruction specifies registers – but failed to discuss what these were used for, i.e. the source of the operands. A fair number of answers failed to provide example instructions, as the question clearly asks for.

In the case of Type 3 instructions, a number of answers failed to discuss that the branch instruction are conditional. In addition, not mentioning that the offset is from the current value of PC.

c)On the whole answered well. In some cases the approach was mentioned, i.e. use R0 for the MOV and add the contents of a register to itself for a shift, but without giving code examples – which is what was asked for. I was expecting the answer to explain why using R0 for a MOV works, i.e. since R0 is hardwired to 0, and why adding the contents of a register to itself is the same as a shift, i.e. multiplying by two.

d)The answers to this question were mixed. A lot of answers failed to state correct instructions that would correctly load data from memory into registers, and also to store the result back to memory. Even though the Stump instruction formats were provided. A common instruction stated for load was

LD R1, [myData, #1]

which is an invalid instruction since the first parameter in the brackets must be a register.

A number of answers failed to set the flags (using ADDS) when performing the addition of the lower word, i.e. bits 15:0. Also, a number of answers failed to use ADC to take into account he carry from the lower word when performing the addition of the upper work, i.e. bits 31:16. In some case branches were used to decide whether to add 1 or not depending on whether the carry had been set – this is what ADC does for you (in effect)!

Q2 Again, everyone attempted this question. However, overall the answers were not very good, embarrassing even, considering anyone who attempted the lab should be able to make a good attempt at this question. The average mark was 55%.

a)A relatively simple question asking what type of circuits you typically associate with blocking and non-blocking assignments. However, some very strange answers were provided, including synchronous and asynchronous, really? Typically, blocking assignments are associated with combinatorial circuits, and non-blocking with sequential circuits.

b)A discussion was required to explain how the simulator treats a list of non-blocking assignments. I was expecting some example code and a diagram of the active events and nonblocking events queue covered in lectures. The discussion (which was often not provided) should discuss how the rhs hand side the expression is placed on the active events queue, and the lhs assignment to variables are placed on the nonblocking events queue. Once the active events queue has been executed, then the nonblocking queue is executed resulting in the assignments being made. The “discussion” should also mention that events go on the events queue when an event in the sensitivyt list of the always sensitivity list occurs, such as posedge clock. At this time the simulator stops time and only restarts time once the queues are empty.

Some mistakes/misunderstandings:
•Not providing a suitable diagram
•Not providing a discussion
•Not discussing what happens as a function of time

c)The aim of this question was to explain the highlighted parts of the code given. In all cases, the answer should relate to the example code.

i)reg declaration – the answer is that state needs to declared as reg simply because we are using non-blocking assignments. reg does not necessarily mean a register will be implemented, although in this example it probably does.

ii)always block – a block of code that is executed depending on the conditions of the sensitivity list (not always discussed). In this example the assignments are made at the rising edge of the signal clock.
iii)default – when do we use default, and why? When? To cover cases not listed in the case statement – it doesn't always relate to don't care signals! Why? For simulation purposes, so you can trap any invalid states you shouldn't be in because you will observe don't care values, i.e. X.

In the case of the state transition diagram for the FSM, this was relatively simple. However, in this example the reset is synchronous (since there is no dependency on reset in the always block sensitivity list), which many did not spot (or even include a reset action). Since when have state transition diagrams shown "posedge clock" on a transition?

d)This was an example in producing a Verilog description for an FSM module. The question itself presented the full design of the complete system, and in some respect this discussion was unnecessary for answering the question – all you need is the state transition diagram. However, this was given as an example of a complete design.

You have produced FSM descriptions in both 121 and 221, where we have used an approach where the FSM is broken down into three always blocks:
•A combinatorial block for producing the next state from the current state and the inputs, sensitivity list of *.
•One for performing the next state to current state assignment (a register!), including and reset action.
•A combinatorial block for setting the output signals in each state, sensitivity list of *.

What was I looking for?

•A correct module definition
•Correct implementation of an asynchronous reset action
•Properly defining variables when assigning by specifying the bus width and base, i.e. 3'b000
•Defining internal variables, such as the state, correctly and using the correct number of bits
•Use of default (as case statements are the most efficient approach) and assigning variables X in the default
•Correct state transitions
•Correct output assignment for each state

Some general mistakes/misunderstandings:
•Implementing the design as a single always block, or two always blocks … often resulted in the outputs being out of sync with the current state as
•Not implementing an asynchronous reset – posedge reset should be in the sensitivity list for the next state to current state assignment block.
•Not defining outputs as reg when blocking/non-blocking assignments are used.
•Not setting all variables to a value in every state. For example, in quite a few of the answers provided outputs were only assigned when they were required to be set to 1. They were never set to 0, and as a result they were always set to 1 once set.
•Performing the assignment to a single variable in more than one always block – Verilog-101 – one should never assign a variable in more than one always block.
•Putting comments such as – "all other outputs are 0" – you are asked to write the code, would you do this in your code?

Q3 Most candidates attempted this question.

Marking this question may have been more depressing than trying to answer it. It is depressing to see how little -understanding- of the issues there is, this year.

The intent of the question was to connect some of the microarchitectural principles from the lectures with the practical experience from the lab; almost all students have completed the latter so it might reasonably be expected that at least the starting point was familiar.

To give some 'progression' through the question it begins with an unpipelined implementation, as in the lab. This has the more difficult 'problem' elements.

(a) The key here is the (emphasised) minimum number of cycles. The Stump in the lab. executes operations in 2 or 3 cycles and this is not minimised. The commonest answer (for the first part) was 'five cycles', yet this is noticeably more than the two (or, indeed, three) which have been experienced with a very similar problem! It seems that 'functional unit' is somehow equated to 'clock cycle', yet in the lab we had {register read, shift, ALU, register write} in a single cycle without demur. Additionally, several people decided that there would be two register reads (reasonable) so these would be done successively (certainly not these days) despite having seen plenty of pictures of parallel read ports.

Another common issue was to discard operations such as register read when performing memory operations. Did Stump do this? How about 'store' operations?

The key to this first part is the "single memory bus" which is the architectural bottleneck, forcing loads and stores to require two cycles (one for fetch, one for memory transfer) and thus the execution time can be partitioned differently. This was almost universally overlooked.

It was common to find the 'cycle time' specified as the sum of the functional unit times, even when the timing supposedly took (say) 5 cycles. This is missing something fundamental about synchronous operation.

(b) This was intended as the 'challenging' part of the question, and so it proved. Significant flexibility in answers was allowed to accommodate answers from part (a). Few candidates apparently thought about the implications of the instruction mix on overall performance - a fundamental issue in microarchitectural choices; credit was given for

attempts to do so.

(c) Most candidates could describe pipelining to some extent. This is, of course, straightforward 'bookwork'.

(d) This 'problem' is more straightforward than the earlier task and was better addressed. However it was disturbing to see this interpreted as 'pick four stages of your choice' (and discard ones you don't like) etc.

There are several issues to think about in this question but it was intended to test how several of the topics in the module are connected: architecture, critical paths, pipelining, timing and clocking... Apparently the answer is this didn't 'happen' for most people this year. We will look at the presentation of the material again in this light.

Q4 Not much to say here. Very few candidates attempted this question; although part of the syllabus and necessary background for a hardware implementer it is less integrated with the majority of material, especially the practicals.

Answers which were submitted were mostly 'okay'.