Computer Architecture Detailed Assessment Specification

Overview

This document gives the detailed specification for the tasks for the Computer Architecture coursework. This includes development of a Logisim CPU simulation, amending a MIPS assembly language program and written work. All practical tasks must be demonstrated using images in your report as described in this detailed specification. Your final code files from the practical work must be included in appropriate appendices to your report.

The full report should include your answers to the discussion questions, detailed below, as well as being structured appropriately with a title page, contents page, introduction and conclusion. You should use the report structure document from the assignment pack on Moodle as the basis for your report. There are videos available from the assignment pack covering how to get started on each part of the assessment, including using the report structure document.

The Logisim simulation development described in detail below form the following tasks:

- Test the existing Logisim CPU using digits from your student id.
- Amendments to the existing Logisim CPU:
  1. Implement a reset button
  2. Add functionality for the halt instruction
  3. Add functionality to the ALU
  4. Add functionality for branch instructions
  5. Incorporate input and/or output devices into the CPU simulation.

The MIPS Assembly language details cover the following tasks:

- Assemble and test the existing MIPS Assembly language program in the MARS/MIPS simulator.
- Amendments to the Assembly Language program:
  1. Improve the messages the program outputs to the user
  2. Use a subroutine
  3. Enhance the functionality of the program
  4. Incorporate a loop
  5. Use an array to store the digits. This is a more advanced task.

Part 1 Logisim CPU Simulation development

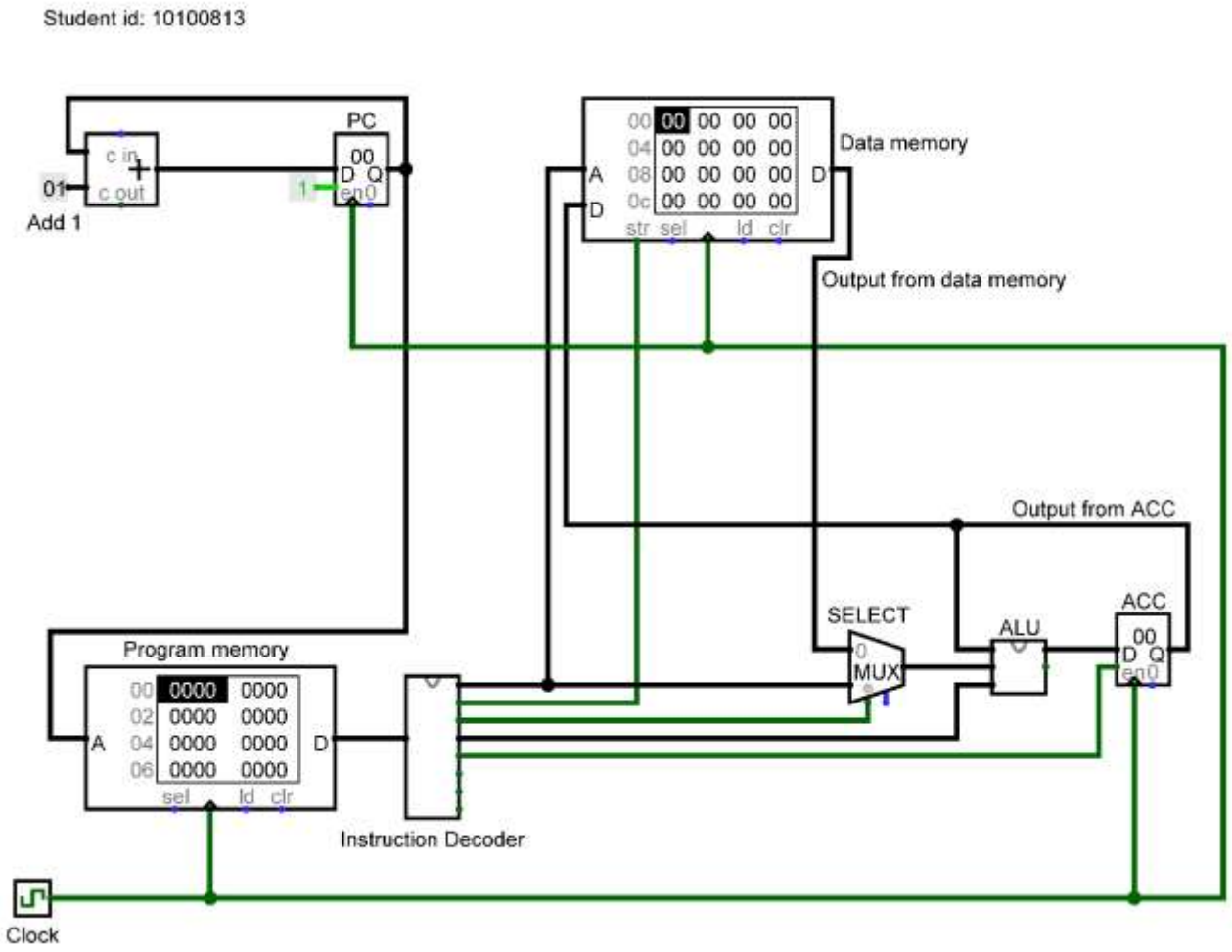
Download the following resources from the assignment pack Moodle and store them together in a folder.

- CPU circuit (Logisim CPU starter.circ)
- Sample program(s) (program.dat, program-fib.dat, program-fib-loop.dat, program-fib-loop-cond.dat)
- Sample data (data.dat)

You will need to add evidence to your report based on the report structure document from the assignment pack.

In the assignment pack on Moodle, you will find a separate document and video with background information about how this Logisim CPU works. There is also a video showing the first tasks being carried out and evidence gathered for a report.

When you open the CPU circuit in Logisim, it will look like this:



You are strongly advised to save a copy of your circuit after each task (not just the images for your report) as you will need to include the full Logisim code of your final version of the CPU in an appendix to your report. In the assignment pack you will find a word document that you can use to structure your report.

The starter CPU has spaces where you might want to add more components and wires, but you will need to change some of the connections to complete the tasks and may move components, as long as the

original features still work. You need to take care when moving wires as it is easy for components to become disconnected or connected to the wrong items, so you should keep a previous version to go back to in case of mistakes.

Each task asks you to add images to your report to show the development and testing. For each task, there should be a statement as to whether you consider that task complete or attempted, the report structure document from the assignment pack initially states that all tasks are not complete. These statements need to be amended. For some tasks, you are asked to calculate a decimal answer separately (using a calculator if needed) and to convert that answer to hexadecimal. You may use any suitable online tool to make that conversion and you do not need to reference the source for conversion between decimal, hex and binary.

The first few of these tasks have very detailed instructions, but later tasks expect you to apply problem solving and logical thinking to complete them. The final task, to incorporate input and output devices is much more open with many possible approaches.

Testing the CPU with your student id

Open the circuit provided (Logisim CPU starter.circ) in Logisim. Change the student id shown in the text in the circuit to your own student id. To do this, click on the Edit Text icon (an "A") and then click to edit that student id.

Edit the data.dat file in notepad++ (or another tool for editing plain text). Change the values to be the last four digits of your MMU student id, each separated by spaces. Save the file. Make sure that the circuit is in simulation mode (the hand symbol is selected) and load the program file from program.dat to the program memory and your amended data to the data memory. To load data to memory, right click on the memory and choose "Load image…", navigate to the appropriate file and click open.

Add an image to your report showing the CPU with the data and program loaded. To create an image of a Logisim circuit, go to File and Export Image. Make sure that you uncheck Printer View. Choose a suitable name and save the image so that you can paste it into your report, see the video on getting started with the Logisim tasks if you are unsure about this.

The program (program.dat) provided should add up the four digits that you have placed in the data memory and put the result at address 4 in the data memory. Run the program by clicking on the clock signal multiple times. Each time the clock goes to 1 (it will show as a brighter green), one operation will be carried out. Stop clicking when the instruction f000 is highlighted in the program memory. Take an image at this point and add to your report. The instruction f000 should stop the program but does not work in the initial version.

In your report, give the sum of the digits in decimal and converted to hexadecimal (use a calculator or other tools as necessary) so that you can check the result of the program.

Amendments to the Logisim CPU

1. Implement a reset button

The initial version of the CPU has no means of resetting the computer. Your task is to create a reset button that will reset the PC, ACC, data and program memories to contain only zeros, as follows.

With the Logisim CPU open and in edit mode (the arrow highlighted at the top left in Logisim), in the folders of tools, expand Input/Output and click on the Button to highlight it. Choose somewhere to add the button and which direction you want it to face. Use wires to link from the button to the "0" inputs on both the PC and the ACC and to the "clr" input on the data and program memories. There should be only one button to reset all those components. Make sure that you connect the wires from the correct position to the correct positions. Add a suitable label to your reset button.

Test your reset button. Go back to simulation mode and load a program and data as before (if not already loaded). Click the clock to change the values showing in the PC and ACC from 00 (if they are zero). Take an image of your circuit to show the situation before pressing reset, press your reset button and take another image. Add the before and after images to your report.

2. Add functionality for the halt instruction

As you saw when testing the program, instruction f000 should stop the program but does not work. The description below gives a mechanism to get the halt instruction to work.

If you are in edit mode (the arrow highlighted) and on the CPU main circuit, put your mouse over the bottom output of the instruction decoder, you will see that it says Halt. This signal will be on (1) when the op code is "f" and the program needs to stop.

In brief, your task is to amend the circuit so that the halt instruction being set to 1 means that the clock signal is not passed to the components and so the registers (PC and ACC) and the memory stores do not update. In other words, we need what was the clock signal going round the CPU to be on (1) when the clock signal is 1 AND the halt signal is 0. We introduce logic gates to achieve this, as follows.

In the existing circuit, the clock signal comes in and passes to other components. You should remove the first wire coming out of the clock as you will need to replace it with wires and logic gates. Removing the wire will make the remaining clock wires turn blue as they are no longer connected.

We know that AND will output 1 when both inputs are 1. We want a signal that is 1 when the clock is 1 AND the halt signal is NOT 1. To do this, take a wire from the halt output from the instruction decoder and pass it through a NOT gate, then pass this into an AND gate with the other input to the AND gate being from the clock. The output from the AND gate needs to go where the clock signal previously joined the rest of the circuit. You should decide where to place the NOT and AND gates to enable the wiring described. Take care with the direction that your components are facing, if your wires go red you have probably made an error in the connections.

Test your program as with the initial testing and click the clock until the instruction f000 is highlighted in the program memory. Click the clock a few more times to make sure that the program does not continue. Take an image to add to your report where the instruction f000 is highlighted and the clock signal is 1 (bright green) to show that the clock signal is not passed to the other components.

3. Add functionality to the ALU

The ALU is built as a subcircuit in Logisim. To view the internal structure of the ALU, double click on the ALU in the project view (so that the magnifying glass is over the ALU). Add an image of the ALU after your changes to your report, as well as the images required of the main CPU to show your testing as described below. Double click on CPU main to go back to the normal view of the CPU when you want to test a change.

The ALU has a 4-bit input labelled as ALU Operation that indicate which of the outputs should be passed to the accumulator. A multiplexer (MUX) is used to select the signal from one of the components according to the value of ALU operation. Note that the inputs to the MUX start with 0 at the top.

Add the following features:

i. Operation 3 should multiply the two inputs.

   Before doing this task, study carefully how the adder (+) works in the existing ALU. When you highlight the adder, you will see that the data bits setting is set to 8. The inputs to the adder come from the two data inputs to the ALU (ACC to ALU and Inp to ALU). The output from the adder goes to the multiplexer (MUX) input corresponding to number 1.

   The multiply operation should be built in the same way as the adder. Select the Logisim built-in multiplier from the Arithmetic tools. The inputs to the multiplier should be the same as for the adder and the output needs to go to the MUX input corresponding to number 3 (remember that numbering starts from 0).

   To test operation 3, take a copy of the program used previously that added four numbers and change it to multiply those four numbers together. Use the guidance from the assignment pack on Moodle to make sure that you know which of the hex digits in the instruction are for the ALU operation. Note that starting from zero and multiplying will always give zero. You could add the first number (from data address 0) to the accumulator and then multiply by the remaining digits.

   You should have at least three tests, one that contains a zero in the four digits (because anything multiplied by zero is zero), one that has four numbers that when multiplied together give an answer that can fit into the data memory and one where multiplying the digits gives a number that is too big for the storage available. One of those tests must be the digits from your student id. For each test, create an image of the CPU that shows the result of the program running (with the final instruction f000 highlighted). If your halt instruction works, you might want to run the program continuously by going to Simulation and choosing Ticks enabled. Choose a low frequency so you can watch the simulation working. To stop the simulation, go back to Simulation and clicking Ticks enabled again.

   For the tests without a zero, multiply the numbers separately in decimal (on a calculator) and add the results to your report. Convert those results to hexadecimal and show that result as well so that you can check that the output of the CPU is correct (remember that the CPU output will be shown in hex at address 4 in the data memory). Give a statement as to whether the result in the simulation is what you expected from your calculations.

ii. Operation 4 should invert the value from the accumulator.

To make this amendment inside the ALU sub-circuit, remember that a NOT gate is also called an inverter and has only one input which in this case should come from ACC to ALU. The data bits will need to be set appropriately for the NOT gate and its output connected to the appropriate input to the MUX in a similar way to above.

To test operation 4, you should make a copy of the program that you used for the multiplication task and add an additional command to the program to invert the value from the accumulator after the multiplication is complete but before the final result is copied to address 4 in the data memory. This operation does not need other data from the instruction or data memory, but you should set the CPU operation to 1 and the immediate value to 00.

Test your circuit with one of the inputs that you used for the multiplication test above that did not give a result of zero and take an image of your simulation when the final instruction f000 is highlighted, as before. In your report, give the hex value of the previous output, followed by that value converted to binary. Invert the binary value (change all ones to zeros and all zeros to ones) and convert back to hex to check the result given by running the program. Give a statement in your report indicating whether the result of the simulation was what you expected.

Remember to add an image of your completed ALU to your report after doing these tasks (in addition to the image of the CPU to show testing).

4. Add functionality for branch instructions.

i. Cater for a branch always instruction.

Being able to repeat instructions is important in programming, a branch always instruction will set the program counter to the address given in the instruction (rather than just using the value after adding 1 to the previous program counter value). The branch always feature should be implemented as described below.

You will see that one of the unused outputs from the Instruction Decoder is labelled Branch Always. This is a 1-bit signal and when it is 1 the program counter should receive the address from the instruction instead of the output from the “Add 1” component. When the Branch Always signal is zero, the program counter should still receive the output from the Add 1 as now.

Notice that the address from the instruction already goes into the address (A) input of the Data Memory. The data (D) input to the PC register needs to come from the Add 1 component when Branch Always is zero, but from the signal that goes into the address (A) of the Data memory when Branch Always is one. This can be achieved as follows.

Remove the existing connection from the Add 1 component to the PC register, you will replace this with other wires and components. You should use a multiplexer to select which of two inputs to pass to the program counter depending on the value of a single bit, from the branch always signal. Multiplexers are in the Plexers set of tools in Logisim and you will have to set the Select Bits and Data Bits attributes and wire it up appropriately.

To test your work, use the file program-fib-loop.dat from the Assignment pack on Moodle. Use your reset button to clear everything and then load this program to the program memory. This program calculates the Fibonacci sequence. The instruction 4003 in the program tells it to go back to the instruction at address 3.

Run the program until it reaches the 4003 instruction the sixth time. You might want to run the program continuously by going to Simulation and choosing Ticks enabled but use a low frequency so that you can count the number of times it reaches the 4003 instruction. You can stop the simulation by going back to Simulation and clicking Ticks enabled again. Take an image of the CPU at this point to add to the relevant part of your report. Remember to reset everything if you need to run the program again.

Check the output against an online listing of Fibonacci numbers (you do need to reference the source you used in this case). Remember that most websites will show the Fibonacci sequence in decimal, but the data in the data memory is in hexadecimal. You will need to do some conversion to show that the program has worked correctly.

ii. Cater for a branch when not equal to zero instruction

The previous simulation never stopped because it kept going back to the same instruction. This task asks you to amend the circuit so that it branches to the address given in the instruction if the value on the accumulator is not zero. The instructions for this task are less detailed than previously, you are expected to apply problem solving.

The ALU has an unused output labelled “Not Equal Zero”. You will need to go into the ALU subcircuit to make that functionality work. In the starter version, the output of “Not Equal Zero” is always 0. Remove the constant 0 and the wire from that constant to the Not Equal Zero indicator, ready to add components. You need to make it work so that the output at “Not Equal Zero” is 1 only when the accumulator is not showing zero. That means that “Not Equal Zero” will give a 0 output when the accumulator is zero. You may use the Logisim comparator component (from the arithmetic tools) and/or basic gates with wiring and a constant to solve this.

You can test your Not Equal Zero functionality before completing the full branch when not zero task. When in the CPU main circuit with the hand icon is highlighted, click on the value in the ACC to change it and check the Not Equal Zero output from the ALU. Remember that the Not Equal Zero output should be 1 (bright green) when the ACC is not zero, and 0 (dark green) when the ACC is zero. Take images for your report to show these two situations.

If you think that the “Not Equal Zero” is working correctly, then you will need to add the functionality to pass the address to the program counter (PC). There is an output from the instruction decoder that is labelled “Branch NE zero” (short for branch when the accumulator is not zero). When this signal is 1 AND the Not Equal Zero from the ALU is 1, then the program counter should be set from the address in the instruction. In summary, after this task the address from the instruction should be used to set the program counter when branch always is set or (branch NE zero AND Not Equal Zero) are set.

When you have all this prepared, there is a program (program-fib-loop-cond.dat) in the assignment pack, that you can use to test your circuit. This program uses a counter stored at data address 4 to count down from 10 (decimal) to 0. The loop continues if the counter is not zero. The program reaches the Halt instruction when the counter gets to zero.

Reset your circuit and load that program. Run the simulation with Ticks enabled, it should stop when the counter gets to zero and the program gets to the final instruction (f000) as long as the tasks have been competed appropriately. Take an image of the circuit at the end of that simulation and add it to your report. Add a statement in your report to explain how the values in the data memory related to your research into Fibonacci sequence values.

5. Incorporate input and/or output devices

One way in which the Logisim CPU is not realistic is that we have to put data in or read it out directly from the data memory rather than using input or output devices. Investigate how to use input and/or output components in Logisim and add to the CPU simulator to make simulation more versatile. You do not need to include both input and output.

You will need to add additional instructions to the instruction set and make appropriate changes to the instruction decoder. You will need to create programs to demonstrate your work. These could include amending one of the previous programs, for example adding numbers that are input from the keyboard or creating an output from one of the programs. You could start with some simpler programs that just demonstrate a new feature, like taking input from the keyboard or sending a value to an output display.

This is a much more open-ended task than the others and you are advised to try this after you have done the other tasks. To gain credit for this task, your input and/or output devices need to be incorporated into the CPU, using the instruction decoder and accumulator, data memory or other components as appropriate. Be very careful when changing the instruction decoder as, when you add a new output, the decoder will move on the layout and no longer be connected correctly.

As it is more flexible and there are many ways of achieving this, you need to include a short description of the work along with suitable images in your report. Explain what you have done and why including the program you have used for testing and the components you added. This explanation should be up to 200 words. Make sure that you include images of all circuits you have changed and show a range of testing.

You will find online materials including YouTube videos that use Logisim input and/or output devices. If you use any online materials, make sure that you cite them in your report with appropriate references.

Adding the Logisim file to your report

When you have completed your Logisim work, paste your Logisim CPU circuit containing all the parts you have attempted to the appropriate appendix in your report.

To do this, open your Logisim .circ file in Notepad++ (or another plain text editor) highlight and copy all the code and paste into your appendix. The report structure document contains a style called Code listings that should be used for code, both the Logisim and MIPS Assembly language files. See the supporting video in the assignment pack on Moodle if you are unsure about this.

Part 2 MIPS Assembly Language programming

This assembly program development is based on amending and testing an existing assembly program that runs in the Mars MIPS simulator. Download **MIPS program starter.asm** from the assignment pack on Moodle.

You need to include your final version of the program containing all the tasks you have attempted in the appropriate appendix to your report. That final code should assemble and run. If you have started a task but not been able to complete it, you may submit the attempt commented out so that the remaining code works. If you comment out a section that you want the marker to see, add an additional comment explaining what you were trying to do.

As you improve the program, the previous features need to still work and that should be evident through your testing. You might want to make sure that you have a previous version of the program available as you make changes in case you need to revert to a previous version.

The points below suggest a suitable order in which to do the tasks, but it is not essential to follow them in this order. Some of the tasks have only brief specifications so that you can use problem solving to complete them.

Test the existing program

Open the program in the Mars MIPS simulator. Assemble the program and run it. You will need to enter the last four digits of your student id one at a time pressing enter between each one. The program will output the sum of those digits.

Make sure that you can see the full interaction in the Run I/O and take a screenshot of the Mars simulator and add it to your report. Crop your screenshot to show only the Run I/O part.

Amending the program

1. Improve the messages that the program outputs

There is a message in the data area of the program telling the user to press enter between each digit, but that message is not output by the original version of the program.

Amend the program so that the message telling the user to press enter between each digit is output after the initial message and before they have entered their first input. Your code should go at the first position that has ##. Test the program.

Add another message to the data area to say, “The total of the digits is: ”, and change the program so that it outputs this message after all the input has been received and before the total is output. Test the program and add a screenshot of the Run I/O to your report.

Read the program to make sure you can understand how it works before trying the next tasks. It may help to add additional comments explaining what each part does.

2. Using a subroutine

After having looked at the code, you should have found that there were blocks of repeated code. Your task is to create a subroutine to hold that code. You will need to use an appropriate label for the subroutine and put all the repeated code there. You will be able to remove the repeated code from the main body of the program and call the subroutine using jal (jump and link) followed by the label name. The subroutine code should finish with a jr $ra (jump return to address stored in $ra).

Make sure that you can see the code of the subroutine itself as well as the Run I/O showing your testing of this feature and take a screenshot to add to your report leaving the code and Run I/O visible. The program output should be identical to the version without a subroutine.

3. Enhancing the functions of the program

One of the Logisim tasks was to change a program from adding the four digits to multiplying them. For the MIPS assembly programming, you are required to multiply the digits as well as adding them. You will need to choose a register to hold the result of multiplying, amend the value stored as appropriate and output a suitable message as well as the final value. Remember that, when multiplying, you will need to start by adding the first number and multiplying the later digits.

You should show at least two tests for this task. One test needs to have a zero in the input digits, so that the result of the multiplication is zero. The other test should give a result that is not zero. One of those tests must be using the last four digits of your student id.

For the testing screenshots to add to your report, one test should show the code that outputs the result of the multiplication to the console as well as the Run I/O. The other test can show the Run I/O only.

4. Adding a loop

After amending the code to include a subroutine, you will find that there is still some repeated code in the main body of the program. This task requires you to use a loop to repeat that code. In particular, to align with the loop used in the Logisim task, you are required to use a counter that counts down and use a branch if not equal zero (bnez) instruction to control the loop.

You will need to choose a register to contain your counter and store an appropriate initial value in that register. You will need to create a label at the position of the start of your loop, subtract 1 from the counter value and use a bnez instruction to branch to your label.

For your screenshot showing testing, make sure that the code for the loop can be seen as well as the Run I/O for a full run of the program. Add the screenshot to your report.

5. Storing the digits in an array

The program as supplied and amended up to now does not store the digits that have been entered. Each digit is added to the total and then the input value is lost from the program (it is overwritten when the next digit is read in). For this task, you should store each number into an array in the data memory when it is read in. When all the digits have been read in, you should loop through the array and output them to the console before the results of the calculations are output. As well as considering how to use an array, you will need to think about how the console output will appear.

This is a more advanced task than the others and so, if you try this task, you should add a written description of how your code works (or should work) to your report in addition to appropriate evidence

of testing, where you have to determine what is appropriate. The written description should be up to 200 words and cite any resources you have used for the task.

Adding the assembly language code to your report.

When you have finished your attempt at the MIPS assembly language programming, make sure that you have included your final code in the appropriate appendix in the report using the Code Listing style as for the Logisim circuit code.

You should make sure that your final code is set out well and has suitable comments. Although you have changed the program several times in following these steps, the comments should relate to the final version that should contain all the features you have been able to complete. The code should be able to be pasted into the MARS MIPS Assembly Language simulator and assembled and run with no changes made to the code.

Part 3 Written Discussion

To make the most of the discussion part of the report, you should make use of items from the assignment pack; the understanding you gained from working on the assignment; materials and exercises from during the unit as well as using additional research. Any resources that you use, that were not resources created for this unit and available on Moodle, should be referenced appropriately using MMU Harvard referencing. Even when referenced appropriately, using too many direct quotes will limit your mark as explaining concepts in your own words will show that you have understood the work.

When approaching the following discussion topics, you should not repeat any information that you have given earlier in your report but may refer the reader back to any previous parts. You may include additional sub-headings, beyond those included in the report structure document. You may use additional images or code snippets to aid your explanations or refer back to your images from the practical tasks of this assignment. If you use images that are not your own or created for the unit and available on Moodle, make sure they are referenced appropriately.

The three parts of the written discussion should be up to 400 words each.

1. Explanation of how the Logisim CPU runs a program

This should explain how the Logisim CPU works to run a program after completing the tasks.

You should cover the following aspects:

- Flow of signals and data through the circuit
- Program and data memory (referring to bits and bytes)
- Instructions and instruction decoder
- ALU
- The reason for a clock signal
- accumulator (labelled ACC)
- program counter (labelled PC)
- The multiplexer labelled SELECT

You may include other topics to aid your explanation of how the Logisim CPU runs a program. The explanations need to be in your own words and relate directly to the Logisim CPU simulation used in this assessment. This CPU simulation has features in common with other CPU simulations, but there are several differences, so not everything you research will directly apply to this simulation. If you have not completed all the practical tasks, you may still describe the Logisim CPU as if all the features worked.

2. Comparing programming the Logisim CPU simulation to MIPS Assembly language

You should describe the similarities and differences between writing and testing a program on the Logisim CPU simulation used in this assignment and in MIPS Assembly language using the MARS simulator.

Aspects that you might consider are:

- The instruction set and operations
- Planning, reading and writing a program
- Assembling and machine code
- Registers and memory
- Input/Output or Syscalls in MIPS

You are not expected to cover all of these, but to cover a small number in detail. You may choose other areas to describe and should draw on the experience from this coursework, for the Logisim CPU simulation, this means understanding, creating or amending the files that you loaded to the program memory. Some of the MIPS tasks were similar to some Logisim tasks to aid with this comparison.

3. Links between programming a high-level language and the CPU simulation and/or Assembly language

For this part you should critically analyse the links between programming in a high-level language (e.g., Java) and using the Logisim CPU simulation and/or MIPS Assembly language (through the MARS simulator).

This is a flexible task and there are many different approaches that could all gain high marks.

You could consider any of the following or use some ideas of your own:

- How developing in Java relates to the CPU simulation and/or MIPS Assembly language
- Professional practice between the approaches, what is good quality code in Java compared to MIPS Assembly language
- How understanding of the CPU simulation and/or Assembly languages helps with coding in a high-level language
- Data representation
- Tools available to support programming in Java compared to MIPS Assembly language, bearing in mind that MARS is a simulator

You are expected to choose one idea to explore in depth rather than giving brief answers to several ideas. Although are expected to incorporate research into your answer, make sure that you relate your answer to the Logisim CPU simulation used in this assignment and/or MIPS Assembly language rather than giving a very general response. As with the other written work, you should reference any materials you use.

Completing your report

In addition to the content described in this detailed specification, your final report should have a short introduction and conclusion of up to 200 words each as indicated in the report structure document. Note that you are not expected to include an abstract for this report.

To complete the report, you will need to fill in the details on the title page by choosing an appropriate title and subtitle and including your name and student id. You will need to select the contents page and click “Update Table” to get the section headings and page numbers correct. When you have completed the assignment, please check every part of your report. You should not submit any text that was a guideline to using the report structure (in italics).

You should also evaluate your own work and complete a marking grid by highlighting the level that you have achieved on each of the elements. You may also add some comments about your evaluation of your own work.