Computer Architecture

Additional hints and advice for the assessment

Elaine Duffin and Elysia Barker

2023

[1. General FAQ 3](#_Toc155090236)

[1.1 Using the report structure template 3](#_Toc155090237)

[1.1.1 We were told the report would be about 2,000 words, but it seems way more than that? 3](#_Toc155090238)

[1.1.2 What do I need to do in the report for tasks that I have not attempted? 3](#_Toc155090239)

[1.1.3 How do I show that I have attempted a task? 3](#_Toc155090240)

[1.1.4 Do I have to do the tasks in the order specified? 3](#_Toc155090241)

[1.1.5 Won’t everyone’s code be the same? 3](#_Toc155090242)

[1.1.6 What do I need to include in the Introduction and Conclusion? 4](#_Toc155090243)

[1.1.7 What should I reference? 4](#_Toc155090244)

[1.1.8 Should the written parts be in first or third person? 4](#_Toc155090245)

[2. CPU Simulation 5](#_Toc155090246)

[General Hints 5](#_Toc155090247)

[Getting the software 5](#_Toc155090248)

[The circuit 5](#_Toc155090249)

[I went back to one of the tasks and now my images for an earlier task include a later feature, will that matter? 6](#_Toc155090250)

[Please show how the code needs to be added to the report 6](#_Toc155090251)

[2.1 Calculating the sum of data stored in data memory 6](#_Toc155090252)

[2.1.1 Presenting programs and data files in the report 6](#_Toc155090253)

[2.1.2 Changing the student id in the circuit 6](#_Toc155090254)

[2.1.3 Saving a circuit in Logisim Evolution 6](#_Toc155090255)

[2.1.4 Loading data to memory in the CPU simulation 7](#_Toc155090256)

[2.1.5 Clearing data from memory and registers in the CPU simulation 7](#_Toc155090257)

[2.1.6 Running the CPU simulation 7](#_Toc155090258)

[2.1.7 Understanding the program 7](#_Toc155090259)

[2.1.8 Creating an image of the CPU simulation 7](#_Toc155090260)

[2.2 Calculating the sum of Immediate values 7](#_Toc155090261)

[2.2.1 Showing the expected results 7](#_Toc155090262)

[2.3 Bitwise OR 8](#_Toc155090263)

[2.3.1 Showing the expected results 8](#_Toc155090264)

[2.4 Completing the stop instruction 8](#_Toc155090265)

[2.4.1 Changing the CPU simulation wiring 8](#_Toc155090266)

[2.5 Branch always 8](#_Toc155090267)

[2.5.1 Changing the CPU simulation wiring for the Branch always task 8](#_Toc155090268)

[2.5.2 Writing the expected results 8](#_Toc155090269)

[2.5.3 Creating the program with a loop 9](#_Toc155090270)

[2.6 Amend the ALU 9](#_Toc155090271)

[2.6.1 Changing the ALU sub-circuit 9](#_Toc155090272)

[2.6.2 Showing the expected results 9](#_Toc155090273)

[2.6.3 Writing the program 10](#_Toc155090274)

[2.7 Conditional branch instruction 10](#_Toc155090275)

[2.7.1 Identifying powers of 2 10](#_Toc155090276)

[3. RISC-V Assembly Language Programming 11](#_Toc155090277)

[Getting the software 11](#_Toc155090278)

[3.1 Test the existing program with minor adjustments 11](#_Toc155090279)

[3.1.1 Understanding the program 11](#_Toc155090280)

[3.1.2 Additional hints 11](#_Toc155090281)

[3.1.2.1 Presenting data in the report 11](#_Toc155090282)

[3.1.2.2 Making use of the help in RARS 11](#_Toc155090283)

[3.1.2.3 Saving your work 11](#_Toc155090284)

[3.2 Improving the messages that the program outputs 11](#_Toc155090285)

[3.3 Adding a subroutine 11](#_Toc155090286)

[3.3.1 Creating the subroutine 11](#_Toc155090287)

[3.3.2 Benefits of a subroutine 12](#_Toc155090288)

[3.4 Enhancing the functionality of the program 12](#_Toc155090289)

[3.4.1 Adding multiplication to the program 12](#_Toc155090290)

[3.5 Splitting Numbers 12](#_Toc155090291)

[3.5.1 Getting started with the pseudocode 12](#_Toc155090292)

[3.5.2 Calculating the remainder 13](#_Toc155090293)

[3.5.3 Outputs 13](#_Toc155090294)

[3.6 Splitting numbers with a loop 13](#_Toc155090295)

[3.6.1 Looping 13](#_Toc155090296)

[3.6.2 Extending to 5 digits 13](#_Toc155090297)

[3.7 Making use of arrays 14](#_Toc155090298)

[3.7.1 Additional Hints 14](#_Toc155090299)

[3.8 Designing a console game 15](#_Toc155090300)

[3.8.1 Game suggestions 15](#_Toc155090301)

[3.8.2 Random numbers 16](#_Toc155090302)

# General FAQ

## Using the report structure template

### We were told the report would be about 2,000 words, but it seems way more than that?

The report will seem very long in terms of the number of pages, but it will be filled with images of your work. The word count that you are expected to write does not include the title and contents pages, section headings, text in a table, program code or pseudocode, your list of references or any of the appendices.

The CPU simulation and Assembly language sections each have 8 tasks. Six of those should have about 80 words each, the seventh one about 120 words and the final task 200-250 words. Make sure that you only explain the specific aspects required for each explanation. That gives 850 words for each of the two practical aspects with 300 words for the introduction and conclusion.

For most tasks, you should stick to very short text for the Expected Result and Result with up to five sentences for the Explanation. You do not have to explain every aspect of the task and part of the marks for the report aspect are based on explaining the aspects required rather than just explaining everything. The last two tasks in both practical areas will need more writing.

### What do I need to do in the report for tasks that I have not attempted?

For tasks that you have not tried, you can do either of the following: keep the task headings and add text saying this task is not complete ***or*** you can remove the headings for those tasks. As long as the contents page has been updated, then those approaches both count as using the report structure template correctly.

### How do I show that I have attempted a task?

You can get credit for tasks you have attempted but are not complete. For example, to get between 70 and 85% you need to have six tasks complete and an attempt at one more. You will need to have enough evidence to convince the marker that you have made a reasonable attempt by trying something out. This will usually need images to show what you tried as well as a short description of what you tried and what happened. The final code that you paste in your appendices needs to be a working version, so the images you add to demonstrate attempts will be important to show what you tried. For the final tasks that require research, you would need more written description than for the other tasks.

### Do I have to do the tasks in the order specified?

The CPU simulation and Assembly language tasks are presented in an order that was expected to be increasing in difficulty. However, different people will find different aspects easier than others. You do not have to do all the tasks in the order given.

It can be worth reading a later task when you are stuck as you should be able to do at least enough to show an attempt of a later task. For example, the Amend the ALU task does not require you to have completed the Branch always task for the CPU simulation. The research based final tasks for both the CPU simulation and Assembly language work can be done without earlier tasks being complete.

### Won’t everyone’s code be the same?

The starter code and detailed instructions for the tasks were provided and you will be using a standard structure for your reports, you are expected to have similarity in those areas. This will be considered when the markers view the similarity levels given by Turnitin. Although there will be that similarity, the exact position of wires and components and the assembly language code should not be the same between different students except using the Road to 40 videos for the early tasks. The written descriptions need to be in your own words. Even one paragraph that is identical could be misconduct.

### What do I need to include in the Introduction and Conclusion?

The introduction can be very short, perhaps as little as 3 sentences but needs to be relevant to the work included in the report. The better introductions will not refer explicitly to tasks from a coursework so that the work would make more sense generally.

If you go to [Academic Study Skills](https://moodle.mmu.ac.uk/course/view.php?id=98810&sectionid=1033909) on Moodle, you will find a video on Writing Good Introductions and Conclusions. That might be more than is needed for this coursework but could be very useful for future reports. Even though the video refers to writing an essay, most of the points made apply equally well to scientific reports.

The video mentioned above explains that a conclusion should bring together different sections of the work and should show a more complex understanding of the subject of the work. For your conclusion, you should be using your own experience from the coursework to compare the two approaches shown in the report as a way of investigating and understanding computer architecture. The conclusion can include first person writing where you give an overview of your own understanding gained from the topics.

### What should I reference?

You should know from your Graduate Skills unit that all material that is not your own should be referenced even when you are writing it in your own words. For this coursework, you don’t need to reference any materials that were created for the unit and available through Moodle. You also don’t need to reference any tool that you have used to help you convert between binary, hexadecimal and decimal.

Your references list should include all the items that are cited in the main body of the report.

Any websites, videos, books etc. that you have used for background learning but don’t need to cite in the main body should be placed in a separate list with the heading **Bibliography.** Create the new heading and make sure that it is the same style as the existing References heading (the same level of heading). Format the list in the same way as your existing references list. Make sure that you update the contents page so that it includes your new heading.

Remember that all sources that you use towards the coursework need to be included in the references list or bibliography. If it is clear that you have used other resources, but they are not included in those lists, then your mark can be penalised for not using the template correctly. You could be called to a meeting about potential academic misconduct. This could be when your work clearly goes beyond what was covered in the unit (which would be expected for the hardest tasks).

This unit and coursework is designed so that you should be able to complete it using only what is available on Moodle and created for the unit, the software Logisim Evolution and RARS (and the help available in them) and a tool to convert between binary, decimal and hexadecimal. If those are the only materials you have used then you don’t need to include any references at all.

### Should the written parts be in first or third person?

The report should mostly be in third person. That is, you should try to avoid using “I”, except in the conclusion. However, this will not be marked strictly for this report.

# CPU Simulation

## General Hints

### Getting the software

Logisim Evolution is available from here <https://github.com/logisim-evolution/logisim-evolution>

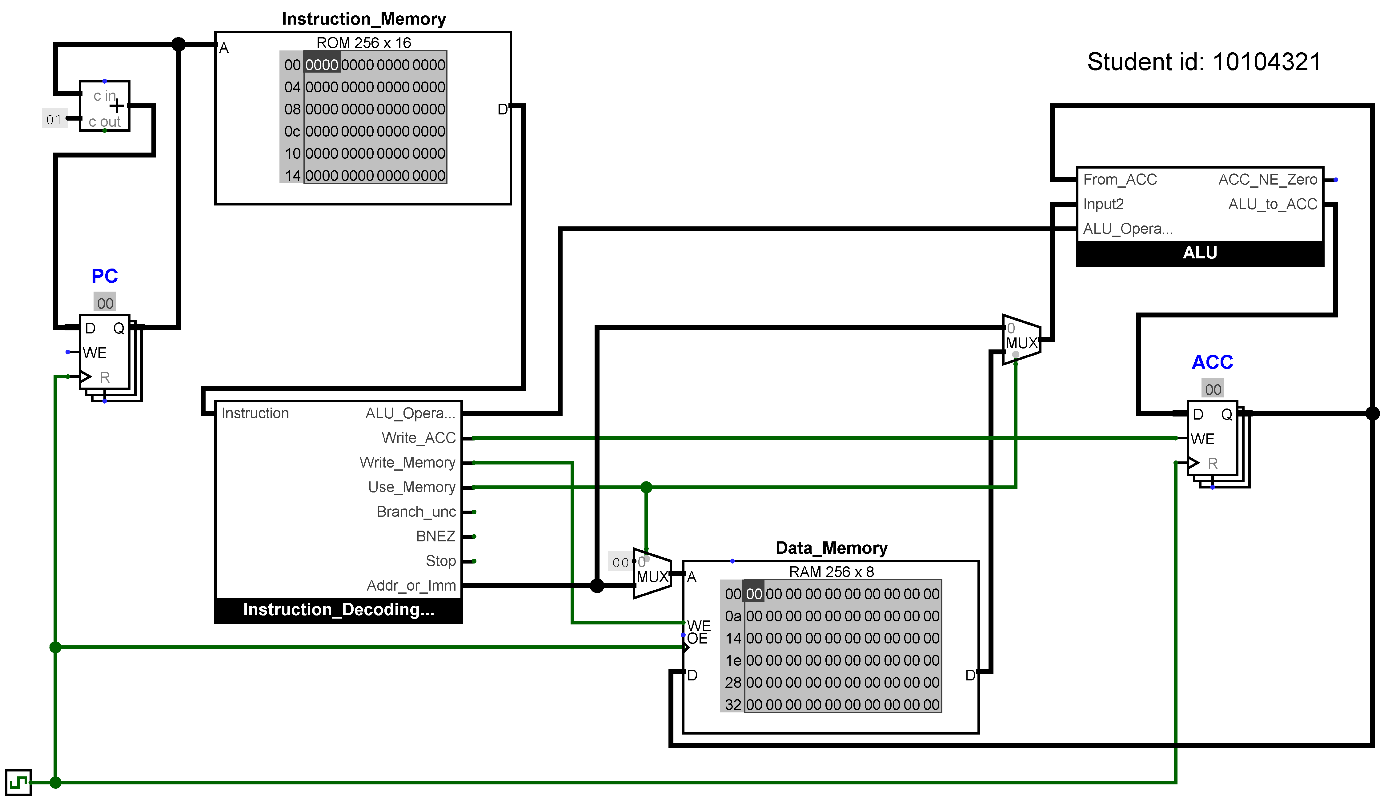
Scroll down to the section headed Download and find out which type of file you need for your system. There are further notes below that for macOS users. Then click download in compiled form and find the correct file for your system.

Some people have been getting an error when trying to install on Windows. It is fine to use version 3.7.2, click on Assets to view the related files.

Another option for mac users is to install a Linux virtual machine and then install Logisim Evolution in there. I have checked that it works fine in an Ubuntu virtual machine. There's a LinkedIn Learning video that covers setting up VirtualBox and Ubuntu.

### The circuit

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



Each task asks you to add images and text to your report. Some of the CPU circuit simulation tasks need calculation of a decimal answer (using a calculator if needed) and conversion to hexadecimal. You may use any suitable calculator or online tool and you do not need to use references for conversion between decimal, hex and binary.

You will need to change connections and move components to complete the tasks and ensure that the original features still work. Please take care when moving wires as it is easy for components to become disconnected or connected to the wrong points, so you should save your circuit after each task so you can go back to a previous version in case of mistakes. If you use your university OneDrive to save your work, you can get previous files back from the web interface.

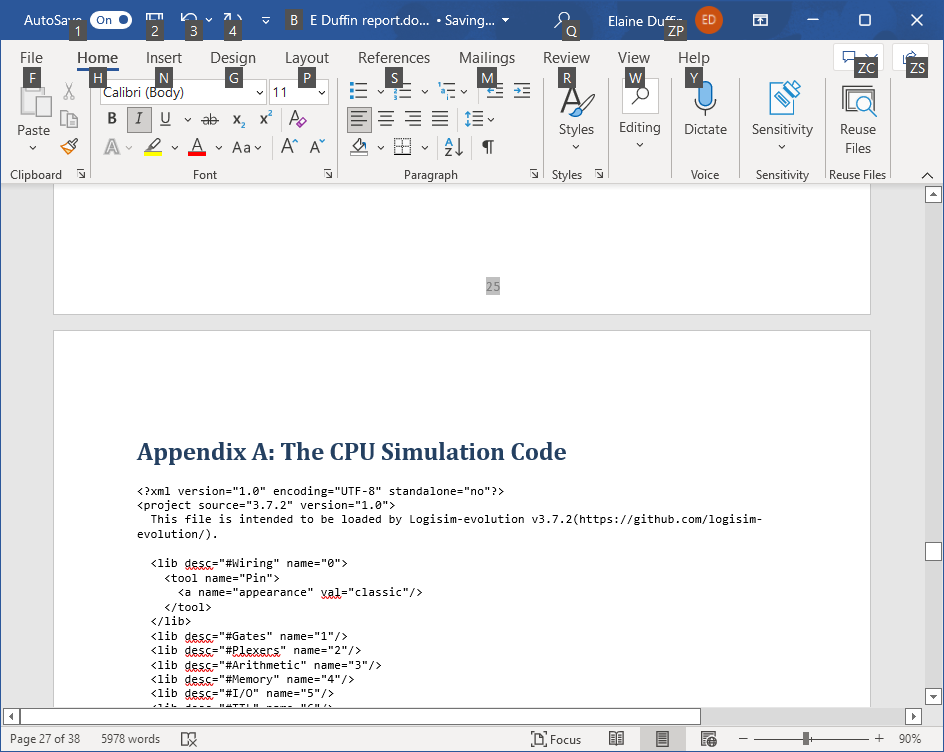
### I went back to one of the tasks and now my images for an earlier task include a later feature, will that matter?

No that won’t make any difference to the marking. However, your final version will need all features that you consider to be complete to work. Features that don’t work in the final version will be considered as an attempt rather than complete even if your report gives clear evidence that they previously worked.

### Please show how the code needs to be added to the report

The full code for both the Logisim Evolution CPU Simulation and RISC-V Assembly language tasks needs to be added to the report. For the Logisim Evolution code you should go to the folder where you have stored your work and open the .circ file in a text editor (like notepad++). You would normally need to right-click on the item to do this.

You should paste in the code, highlight it in the report and select Code Listing in the Styles settings in the Home ribbon in MS Word (from the report structure document), or that you have selected a fixed-width (monospaced) font, ideally use font Consolas with size 9 pt. The image below shows my report as shown in the video on getting started with the CPU Simulation tasks.



## Calculating the sum of data stored in data memory

### Presenting programs and data files in the report

To open a .dat file you will need to right-click and choose Edit with notepad++ or Open with. Paste the contents directly into your report from notepad++ (or similar tool). For a professional look, the code should be formatted using a monospaced (fixed width) font, such as Consolas or Courier New. For a student with the last four digits 4321, the data would look like this:

v2.0 raw

4 3 2 1

### Changing the student id in the circuit

There are two possible approaches:

1. Click on the Edit Text icon (an “A” on the toolbar) and then click to edit that student id.
2. With the circuit in edit mode (an arrow pointer on the toolbar), highlight the text and change the Text in the properties on the left. Make sure you press enter when you have changed the text to get it to actually apply.

### Saving a circuit in Logisim Evolution

Files are not saved automatically in Logisim Evolution. You can go to the File menu and select Save. In Windows you can use Ctrl+S. You are advised to use your university OneDrive so that the files are backed up.

### Loading data to memory in the CPU simulation

Make sure that the circuit is in simulation mode (the hand symbol is selected). To load data to memory, right click on the memory and choose “Load image…”, navigate to the appropriate file and click open. You might need to zoom in or out to see the loaded data clearly.

### Clearing data from memory and registers in the CPU simulation

To clear the Data\_Memory and all registers at the same time, press Ctrl+R. If you just want to clear a specific memory, you can right-click on a memory component and choose clear contents. To clear the registers (or data at specific addresses in the data memory), with the circuit in simulation mode (the hand symbol selected) you can highlight specific data (e.g., above a register) and just overtype the value from your keyboard.

### Running the CPU simulation

Make sure that the circuit is in simulation mode (the hand symbol is selected) and that the Instruction memory and data memory contain values required for the task and that the PC and ACC are both zero. Run the program by clicking on the clock signal. Each time the clock goes to 1 (it will show as a brighter green), one operation will be carried out.

### Understanding the program

See the CPU simulation background information document and video for a run-though of how the simulation works to run a program.

### Creating an image of the CPU simulation

You could use print screen or a snipping tool, but you should be careful that the quality of the image is good and that your report will be readable without the marker having to zoom in.

The images are generally better quality if you use the feature provided in Logisim Evolution. To create an image of a circuit in Logisim Evolution, go to File and Export Image. Make sure that you **uncheck Printer View** and use the scroll to 400% to increase the scaling (which makes the details clearer). Choose a suitable file name and save the image so that you can paste it into your report. In Windows, you can open a file explorer and right click on the file to copy and then paste into the report.

## Calculating the sum of Immediate values

### Showing the expected results

You are asked to shown the running total at each step in both decimal and hexadecimal. For the example of a student id finishing 4321, the table would appear as follows.

|  |  |  |
| --- | --- | --- |
| Input Data | Running Total | |
| Decimal | Hexadecimal |
| 4 | 4 | 04 |
| 3 | 7 | 07 |
| 2 | 9 | 09 |
| 1 | 10 | 0a |

Make sure that the table in your report includes the last four digits of **your** student id.

## Bitwise OR

### Showing the expected results

For the example of a student id finishing 4321, the table would appear as follows.

|  |  |  |  |
| --- | --- | --- | --- |
| Input data | | Expected Results | |
| Decimal | Binary | Binary | Hexadecimal |
| 4 | 00000100 | 00000100 | 04 |
| 3 | 00000011 | 00000111 | 07 |
| 2 | 00000010 | 00000111 | 07 |
| 1 | 00000001 | 00000111 | 07 |

## Completing the stop instruction

### Changing the CPU simulation wiring

When using gates and other components in Logisim Evolution, remember that you have to connect your wires where there are connection points on those components. Wires that touch a component aren’t necessarily connected.

For this task, the clock signal input to the components will be on (1) when the clock signal is 1 AND the Stop signal is 0 as follows.

In the existing circuit, the clock signal comes in and passes to inputs labelled “>” on other components. Make sure that you are in edit mode (with the arrow highlighted) and 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 wires to the clock inputs turn blue as they are no longer connected.

An AND gate will output 1 when both inputs are 1. We want a signal that is 1 when the clock is 1 AND the Stop signal is NOT 1. To do this, take a wire from the Stop 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 join the wires that previously joined clock signal the rest of the circuit. You should decide where to place the NOT and AND gates to enable the wiring described as neatly as you can. Take care with the direction that your components are facing, if your wires go red you have probably made an error in the connections.

## Branch always

### Changing the CPU simulation wiring for the Branch always task

The data (D) input to the PC register needs to come from the existing Adder component when Branch\_unc is zero, but from the Addr\_or\_Imm value when Branch\_unc is one. Remove the existing connection from the Adder 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 data (D) input to the PC with the select signal being the 1-bit Branch\_unc control signal. Multiplexers are in the Plexers set of tools in Logisim Evolution and you will have to make sure that the Select Bits and Data Bits properties are set appropriately.

### Writing the expected results

You do not need to include a table for your expected results for this task (although you can do so if you wish). When your wiring is complete, you will be writing a program (based on program1.dat) so that when it has added up the four digits, it goes back and adds them up again and allow it to do so five times.

For a student ID that ends in 4321, the program should do 4 + 3 + 2 + 1 + 4 + 3 + 2 + 1 + 4 + 3 + 2 + 1 + 4 + 3 + 2 + 1 + 4 + 3 + 2 + 1 by going back and repeating the instructions (rather than just adding lots of instructions). You will be running the program by clicking the clock signal so that this has happened 5 times. For a student ID ending in 4321, calculating the total gives 4 + 3 + 2 + 1 = 10 in decimal, so after looping five times the total would be 5 times that.

You can make a short statement (one or two sentences) that state the expected result in decimal and hexadecimal (referring back to the first task if appropriate).

### Creating the program with a loop

When you take a copy of program1.dat, the f000 instruction can be replaced by your new instruction. Your new program will form an infinite loop and never reach stop. The program needs to do the instructions from program1.dat, but instead of stopping, it should go back to add the first digit from memory (from address 0 of the data memory) again and continue from there.

CPU operation 4 is for the unconditional branch instruction, so the first digit of the new instruction needs to be 4. The ALU will not be used, so the second digit of the instruction should be zero. The last two digits of the new instruction need to be the address of the instruction that adds the data from address zero of the data memory. That is the instruction that adds the first digit from the data memory. Remember that the addresses of the instructions are consecutive numbers starting from zero so you can work out the address by counting.

When testing the program, make sure that your program adds from the previous values and doesn’t start from zero each time the loop runs.

## Amend the ALU

### Changing the ALU sub-circuit

The ALU is built as a subcircuit in Logisim Evolution. To view the internal structure of the ALU, double click on the ALU in the project view on the left (so that the magnifying glass is over the ALU). 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 to indicate which of the outputs should be passed to the accumulator. A multiplexer (MUX) selects the signal from one of the components according to the value of ALU\_Operation.

Check that you know how the existing addition and bitwise OR work inside the existing ALU. When you highlight the adder, you will see that data bits property is set to 8. The inputs to the adder come from the two data inputs to the ALU (labelled From\_ACC and Input2). The output from the adder goes to the multiplexer (MUX) input corresponding to number 1.

The bitwise AND operation and subtraction operations should be built in the same way as the existing addition and bitwise OR (having the same two inputs) but should connect to the required inputs to the MUX. You will need to set the Data Bits property of the components used.

### Showing the expected results

|  |  |  |  |
| --- | --- | --- | --- |
|  | Decimal | Binary | Hexadecimal |
| Initial Value |  |  |  |
| After subtracting 1 |  |  |  |
| After Bitwise AND | n/a |  |  |

### Writing the program

The specification asks you to write a program to do the following:

1. Load an immediate value to ACC – that value should be the total of the last four digits of your student ID.
2. Store the value from the ACC to address 01 in data memory.
3. Subtract 1 from the value from the ACC (use an immediate value).
4. Store the result from the ACC to address 00 in data memory.
5. Do a bitwise AND between value from accumulator and value from address 01 in data memory.
6. Stop the program.

Each of those points should be one instruction (four hex digits). Use the document CPU simulation background information from the Assessment Video Guides on Moodle for a full list of the CPU and ALU operations. You will probably want to write your code into notepad++ (or other text editor), possibly using one of the previous programs as a starting point.

For each instruction, work out what the CPU instruction should be, the ALU instruction (or zero if that is not needed) and the last two digits will be an address or immediate value.

The programs that were used for tasks 2.1, 2.2 and 2.3 give you examples for loading an immediate value, storing to data memory, and using addition and bitwise OR in the ALU. The instructions you need for this task are very similar but you will need to use the ALU operation for subtraction and bitwise AND.

## Conditional branch instruction

### Identifying powers of 2

A number is a power of 2 if it is a number of the form 2*n* where n is a non-negative integer. The first few powers of two are 1, 2, 4, 8, 16, 32, 64. Because binary is base 2, a power of two will always have just one 1 and the rest zeros in its binary representation.

One method to find out if a binary number (*x*) is a power of two is to do a bitwise AND between the number and one less than the number. If the result of that AND is zero, then the original number must have been a power of two. If the result was any other value, the output would not be zero. This bitwise AND between a number and that number minus 1 is exactly the final value that was calculated by your program for the ALU amendment task so you can copy that program as a starting point for this task.

You will need to use both conditional and unconditional branch instructions in your program. The best submissions will output 00 when the input is value is zero rather than including zero as a power of two.

The logic for using the branch instructions is the same as you would use in assembly language.

# RISC-V Assembly Language Programming

## Getting the software

The RISC-V emulator RARS is available here [GitHub - TheThirdOne/rars: RARS -- RISC-V Assembler and Runtime Simulator](https://github.com/TheThirdOne/rars)

You download a jar file and need to have java available on your computer to run it. Some people find that they need to run the jar file from a command line. If you need to run it from a command line, make sure the file rars1\_6.jar is in the same folder as the command prompt shows and then type “java -jar rars1\_6.jar” and press enter. Depending on the operating system you are using, you may need to give permissions to allow your computer to run the file.

## Test the existing program with minor adjustments

### Understanding the program

The program we have here initially starts off with the .data section which allows us to define our messages which will be stored in memory. This helps us when we assemble and run the code so we can be prompted for what to do next.

The main body of the program is written in the .text section which initially outputs a message to tell us what to do, followed by us entering the last 4 digits of our student ID separately. Each individual digit is stored in the register a0 when entered and is immediately added to the value in the register s0 to perform a running total, adding the digits together as we go. This is why we can copy and paste a section of text without the need for any alterations.

### Additional hints

#### Presenting data in the report

Remember to be professional about your presentation. Ensure the screenshots of the Run I/O are of just the Run I/O and not the code itself unless the question specifically asks for a screenshot of the code. Remember to adequately crop and size the screenshots so they are easily readable and well presented.

#### Making use of the help in RARS

In RARS there is a handy help button at the top of the screen which gives you a list of all the possible values to store in a7 and what they do. This will help you right from the start if you are unsure how to perform different actions. Click on the question mark and then open the tab called “syscalls” to find the list of commands.

#### Saving your work

Remember to save regularly.

## Improving the messages that the program outputs

## Adding a subroutine

### Creating the subroutine

Make sure that you place your subroutine at the end of the code after the system call to exit the program.

Make sure that you ensure that the name (label) you use for the subroutine is professional and relevant to the task. Also, ensure you are formatting the subroutine correctly with the appropriate names, statements, and syntax.

You might find it makes more sense to use more than one subroutine and it is fine to do so. Bear in mind that you might not have the skills be able to call a subroutine from another subroutine.

### Benefits of a subroutine

Remember that a subroutine has the same benefits as using a procedure, function or method in programming.

## Enhancing the functionality of the program

### Adding multiplication to the program

The existing program uses register s0 to keep the running total as each new digit is entered. Study the existing code and make sure that you find all the places where the value of s0 changes. The changes for multiplication can be just a few lines of code based on the existing addition. You should **not** require the digits to be entered twice or store all the input digits in separate registers as those would change the original features of the program.

Choose a register to hold the result of multiplying all four numbers (any register that is s or t followed by a number and is not already used in the program would be fine). Amend the value stored in that register as appropriate based in a similar way to you found for s0 but using multiplication when needed. Remember that, when multiplying, starting from zero and multiplying always gives zero, so you might want to add the first digit rather than multiplying or set your register to have an initial value of 1.

Set up a message and output the message and the value from your register in a similar way to the addition.

## Splitting Numbers

### Getting started with the pseudocode

Each line of the provided pseudocode becomes one assembly language instruction. If you are unsure about continuing the pseudocode, you may want to code, assemble, test and provide evidence for just that part of the program to show that you have attempted the task.

You are not expected to use a loop in this task, so your pseudocode and assembly language will have a lot of repetition that is (nearly) identical.

The first part of the program can be built by copying and amending parts of the program used in the previous tasks, i.e., to request the last 4 digits of the id, get the input from the user and store that input in a register (in this case s0)

The remaining parts of the pseudocode provided will be explained using user input of 4321 as an example. Note that the integer result of division is also called the quotient.

Set register s0 to have the value from a0 (the user input)

Set register s1 to 10 (this will be used for division later)

Set register t0 to the sum of 1000 plus zero

After these: s0=4321, s1=10, t0=1000

Divide s0 by t0 and store the integer result in t1

After this: s0=4321, s1=10, t0=1000, t1=4

Update s0 to be the remainder of s0 divided by t0

After this: s0=321, s1=10, t0=1000, t1=4

Divide t0 by s1 storing the integer result back in t0

After this: s0=321, s1=10, t0=100, t1=4

Divide new s0 by new t0 and store the integer result in t2

After this: s0=321, s1=10, t0=100, t1=4, t2=3

Update s0 to be the remainder of s0 divided by t0

After this: s0=21, s1=10, t0=100, t1=4

### Calculating the remainder

Note that the instruction **rem** can be used to calculate the remainder after division.

### Outputs

Remember to make sure there is a distinct separation between the digits. Add a space, new line, or other character between each digit to make sure they are visibly separate.

## Splitting numbers with a loop

### Looping

The code you created for task 3.5 would have a lot of repeated identical or nearly identical code. You might have reduced the repetition a little by including a subroutine, but still have repeated code.

You should first define your message asking for the student ID numbers in the .data section above the loop as in the previous version of the program. In the .text section, you should also define any necessary information outside of the loop (e.g. the number of times you would like the loop to repeat).

Think about what needs to go inside the loop and what needs to go outside the loop. Is there anything that should have a value before the loop begins?

Remember, you can have the loop counting up or counting down but you need to make sure to define the correct number of iterations through the loop. You can use any of the available branch instructions in RISC-V assembly language to control your loop.

Think about what values need to change inside the loop. Are there things in the previous task that are changed by a fixed amount each time?

### Extending to 5 digits

When you extend to 5 digits you don’t need to keep your program flexible so that it can still handle four digits, your final version can handle 5 digits only.

In the previous task you added 1000 to zero to store a value of 1000 in a register. You will find that this approach doesn’t work when you want to extend the code to cater for a 5-digit number.

There are multiple ways to cater for setting a value of 10000. For one approach, look at the help in RARS for the pseudo-instruction li. You will find that there are two uses of li shown one of which will cater for what you need.

There is an additional section heading Conclusion but nothing in the specification telling you what to include. You can either write a one sentence summary of what your program has achieved under the heading “Conclusion” or you can remove the heading entirely.

## Making use of arrays

### Additional Hints

If you have completed task 3.6, you should be able to store the individual digits in the array inside the loop you have created. If you have not completed 3.6, you can store the digits as they are worked out and adjust the memory address to move to the next position in the array.

The task on using arrays in Assembly language is designed to be harder than the others and need some research to build on the lecture from week 10. There is a concept video that has an additional example from those given in the week 10 lecture in the [Week 10 Concept Videos](https://moodle.mmu.ac.uk/mod/page/view.php?id=4172086). In the week 8 Useful Links on Moodle there is a link to a YouTube playlist that includes using arrays as well as an online book that has code examples including using arrays and memory.

If you are doing the version that uses loops, you should consider using two loops when using arrays. In your existing loop, you should store the value to consecutive addresses in memory (but not print it). You can then test this and see the values in the data segment (this would be a strong attempt at the task). You can then create a second loop to print out the values from the array. You should be able to find plenty of examples online of code to print out from an array.

Remember that you need to use citations and references for any materials other than those created for this unit and made available on Moodle.

## Designing a console game

### Game suggestions

Listed below are three suggestions of a game to design in RISC-V assembly language. You should challenge yourself and use techniques from throughout the course. The game is ultimately your decision and you are free to create a new game not in the list of suggestions if you wish. Remember to copy and paste your commented program into the corresponding appendix.

Suggestions list:

* Higher or Lower
  + A random number is generated by the program
  + User guesses whether the next number will be higher or lower
  + A further random number is generated
  + If the user input is correct, continue playing
  + If the user is incorrect, end the game and output a list of previous guesses
  + Continue playing for a set amount of rounds
  + If the user completes all rounds successfully, a congratulatory message is displayed along with a list of their previous guesses
* Guess the number
  + A random number is generated by the program
  + User guesses the number
  + The user gets a clue as to whether their number was too high, too low or correct.
  + The user has multiple chances to guess the number
* Noughts and crosses
  + Represent the board using arrays
  + This will be a 2-player game so you will have to play as 2 people
  + Retrieve a user prompt to input where you will place your X or O by specifying the row and column
  + Check to see if the move is valid (if there is already an X or O on that place, it is invalid)
  + Check to see if the move has won (3 in a row either horizontal, vertical, or diagonal)
  + Check to see if there are empty spaces (if there has been no winner and there are no longer empty spaces available, the game is a draw)
  + Output appropriate messages depending on a win or draw
* Short multiple choice quiz
  + Create all the questions and set of answers in the .text area of your program
  + Consider how to get the response from the user and check whether they got the answer correct.
  + Consider how to keep a score when a player takes multiple questions.
* Snakes and Ladders
  + Represent the board using arrays (You do not have to have the same number of squares on the board as a real game of snakes and ladders)
  + This should be a 2 player game but you may also program this for just 1 player for ease
  + Place a “snake” or a “ladder” on random tiles by specifying their row and column (where there is a snake you may use “S” and ladder you may use “L”). Make sure there is an appropriate amount of snakes and ladders on the board.
  + At your turn you “roll” dice by randomly generating a number between 2-12 (for 2 dice).
  + Your counter (use an appropriate symbol e.g. “Y” for You) will move the number of positions on the dice.
  + If you land on a ladder you will climb up by a full row or two
  + If you land on a snake, you will slide down by a full row or two.
  + This game may be much more difficult to write than the previous suggestions but will be a good challenge.

### Random numbers

Several of the game ideas require random number generation. The RARS emulator has Syscalls available for this. In particular, you should look at the descriptions for RandSeed, RandInt and RandIntRange.

Other than this, you are not expected to use features of RISC-V assembly language beyond those covered in the course. In particular, you should not use floating point numbers.