Imperial College London

CPU Project Report

Aadi Desai, Benjamin Ramhorst, Kacper Neumann

ELEC40006 – Electronics Design Project 1 2019-2020

Dr Edward Stott, Mrs Esther Perea

Imperial College London

Submission date: Sunday, 14th June 2020

Real word count: ~10,862

TABLE OF CONTENTS

1. Introduction

A Central Processing Unit (CPU) is at the backbone of electronic engineering and computing. State of the art CPUs are capable of performing many different operations at very high speeds. There are many different CPU implementations, ranging from basic to very specialised, but the most commonly used CPUs are found in desktop computers and mobiles phones. We chose a project through which we would design our own CPU, since this involves both creativity, to ensure a wide range of capabilities, as well as technical and analytical skills, to ensure correctness and efficiency. Furthermore, we thought this project would give us some insight into how a modern computer works.

The project aim was to build a CPU capable of performing a wide range of operations, such as arithmetic, logic and memory operations. The final design was based on several modern CPU architectures and mathematical algorithms to ensure correctness and maximise efficiency.

This report describes our project. The report starts with a detailed set of requirements for the final CPU. The next two sections describe planning, teamwork and project progression. Finally, the last three sections describe the final design, alongside a critical analysis and a reflection on the overall project.

2. Problem definition

2.1. Project outline

The following section describes the project outline – technical problem definition, team members, project length and some constraints and requirements.

Project title: Design of a general-purpose CPU

Project description: The main goal of this project is to build a general-purpose CPU, capable of performing a wide range of operations including arithmetic operations such as addition, subtraction and multiplication; logic operations such as bitwise operations and conditionals; memory operations such as load and store. The CPU should be tested with a wide range of mathematical functions and programming algorithms, including but not limited to calculating a Fibonacci number, generating pseudo-random numbers, and traversing a linked list. The CPU is expected to complete these operations efficiently and correctly, using only built-in hardware and instructions.

Team members: Aadi Desai, Benjamin Ramhorst and Kacper Neumann.

Project length: Five weeks, starting from 11th May with a submission deadline set for 14th June.

Project budget: No set budget.

Technologies used: The CPU should be designed and tested using simulation tools and hardware-design languages. The required software for this project is Quartus Prime. For testing, programming languages such as C++ or bash can be used to automate specific tasks. There are no set requirements for communication services, but it is highly recommended to use some form of version control, preferably GitHub, for sharing project files. It is not required to build the final circuit; project files will suffice.

Technical requirements and constraints:

- The instruction length must be 16 bits, but the minimum number of instructions is not set.
- Efficient multiplication must be built-into the CPU.
- The CPU must have enough memory to store 2K words of data and 2K words of instructions, each word being 16 bits long.
- The CPU must have a stack, but there are no other hardware requirements.
- The CPU should minimise power consumption.
- The CPU should have a low geometric mean time of executing the given testbench tasks.

Verification: The CPU should be tested with three algorithms, as described below. It must produce correct results, but also be efficient, both in terms of power and speed. The algorithms used for testing should be:

1. Calculating the n^{th} Fibonacci number using recursion.
2. Calculating pseudo-random integers.
3. Traversing a linked list of integers and searching for a specific element in the list.

The source code for these algorithms can be found in Appendix A.

2.2. Project specification

The following section describes the project specification, which was used to verify the final design. The specification used for this project was based on the Software Requirements Specification (SRS). We decided to use SRS, rather than Product Design Specification (PDS) since the project will be completed using simulation tools. Some features from a PDS, such as shipping, packing, and manufacturing facilities do not apply to this project. Furthermore, a PDS is more useful when the project is built using physical components since the PDS document includes features such as weight, materials, and shelf storage. Finally, a PDS includes features such as legal, political, and social implications, which would require significant market research. The main drawback of SRS is the lack of user interface requirements for this project. Based on the initial project specification and the project outline, the following functional and non-functional requirements were set for this project:

Functional requirements:

- The CPU must have a 16-bit instruction length.
- The CPU must have enough memory to store 2K, 16-bit words of data and instructions, each.
- The CPU must be able to perform the following arithmetic instructions:
    - Addition
    - Multiplication
- The CPU must be able to perform frequent comparisons, such as equal, greater than (or equal) and less than (or equal).
- The CPU must have a stack.

Non-functional requirements:

- No instruction should take longer than three cycles.
- The CPU must be pipelined to improve performance.
- The maximum clock frequency should be at least 100 MHz at 0 degrees Celsius.
- The CPU dynamic power consumption should not be above 50 mW.

3. Project planning and management

3.1. Project aims and milestones

Based on the project specification, the following aims were set. These aims were later used to verify the final project and the overall functionality and performance of the CPU.

1. Design a CPU capable of performing a wide range of basic instructions (arithmetic, logic, jumps, load/store), with a 16-bit instruction length and enough memory for 2K of instructions and data each. Avoid creating very specialised instructions, such as an instruction for calculating a Fibonacci number or a pseudo-random number, unless necessary.
2. The CPU should be able to perform both arithmetic instructions (addition, subtraction, and multiplication) as well as bitwise logical operations, such as AND, OR, NOT, XOR etc. Division and modular arithmetic are not required, but an efficient implementation of multiplication is. Multiplication must not be implemented using the multiply megafunctions or the Verilog multiply operator.
3. The CPU must have at least four registers to enable work with more data concurrently. Preferably eight, if instruction length allows.
4. The instruction set architecture (ISA) should support a wide range of jumps, due to many different conditionals used in modern programming languages.
5. Implement a last-in-first-out (LIFO) buffer to be used as a stack. This can be done using Verilog or block diagrams.
6. Make the CPU as power-efficient as possible, mainly through minimising the number of components.
7. Pipeline the CPU to make it more efficient.
8. Optional: Create a program for generating .mif files from instructions. This will make converting instructions to hexadecimal easier and speed up testing.
9. Optional: Create a program for simulating the CPU based on the instructions. This is unlikely to be completed, as it requires quite a bit of work and is not necessary as most instructions can be tested quite quickly by hand simulation.

From the above aims, a list of milestones was created to make sure we were on the right track and completed the project on time:

1. Create the basic project and enable collaboration through GitHub.
2. Design a suitable instruction set with corresponding opcodes and meanings.
3. Design a basic CPU with a control and data path and multiple registers.

4. Implement an arithmetic-logic unit (ALU) with the functionality described above. The ALU should be able to perform both logic and arithmetic, but not multiplication. Multiplication needs to be implemented separately.
5. Implement a multiplier and test it independently from the rest of the CPU.
6. Implement a stack.
7. Test with the required tests (Fibonacci, number generator and linked list).
8. Pipeline the CPU and complete the project.

3.2. Team roles

In the following section, the role of each team member is described, alongside some overall team responsibilities. The team consists of three first-year Electrical and Electronic Engineering students: Aadi Desai, Benjamin Ramhorst and Kacper Neumann. This role split was done based on the project requirements as well as individual skills, such as Verilog coding, git usage, report writing etc. Some of these changed as the project progressed.

Aadi Desai (Belbin roles – Implementer and Monitor evaluator): Aadi has a good understanding of Verilog, so he worked on the ALU and the stack. These blocks were implemented in Verilog to avoid large block schematics which are hard to debug. Furthermore, Aadi was responsible for setting up and maintaining collaboration and version control of project files via GitHub, as well as organising and updating the instruction set.

Benjamin Ramhorst (Belbin roles – Implementer and Completer finisher): Benjamin implemented the multiplier circuits and worked on C++ programs to simplify testing. He also acted as the overall report editor, keeping detailed track of sources used and the project progression.

Kacper Neumann (Belbin roles – Implementer and Completer finisher): Kacper implemented the overall CPU, having designed the decoder and the data and control path. Kacper also acted as the overall project manager, keeping detailed track of meeting schedules, pending tasks and current progress.

Every team member also researched different CPU architectures and possible implementations of a multiplier and a stack during the early stages of the project. The final instruction set was agreed on collectively after researching different ISAs. Finally, every team member was required to work on sections of this project report for parts they implemented.

3.3. Communication

For communication and file sharing, we used a wide range of tools and services, as outlined below:

- WhatsApp – Used for planning meetings and asking questions. Also used for posting updates that do not require immediate phone calls or large files.
- Microsoft Teams – Used for scheduling meetings and sharing files to be edited collectively. This service also supports drawing and resource sharing in real-time, which was used while creating the instruction or the CPU design.
- OneDrive – Used for sharing larger files and backup.
- GitHub – Used for sharing technical files, such as block schematics or C++ and Verilog code.

3.4. Planning

Based on the previously mentioned requirements and team roles, the project was broken down into stages, each of which is described below. The project started on 11th May and needed to be submitted by 14th June. Detailed meeting structures and project progression can be seen in the following section.

1. Initial research and project setup (to be completed by 19th May):
   - Two meetings were allocated for this stage, which occurred on 12th May and 16th May.
   - Understand the specification and make an initial plan.
   - Start a new Quartus project and enable collaboration though GitHub, OneDrive and Microsoft Teams.
   - Research modern CPU architectures, their advantages and drawbacks.
2. Conceptual designs (to be completed by 22nd May):
   - One meeting was allocated for this stage, which took place on 20th May.
   - Decide on the instruction set and CPU capabilities.
   - Design a basic CPU, consisting of a control and data path, multiple registers and an ALU. No work in Quartus is required during this stage.
   - Research possible implementations of multipliers and stacks.
3. Design of individual parts (to be completed by 2nd June):
   - Three meetings were allocated for this stage, which occurred on 24th May, 26th May and 31st May.
   - Implement the control and data paths in Quartus.
   - Implement the ALU using Verilog.
   - Implement and test the multiplier.

- Implement a stack, either with Verilog or block schematics.
- Write programs for converting assembly code into hexadecimal instructions.
- Write documentation for individual parts - to be used in the final report.

4. Initial testing and integration of various parts (to be completed by 6th June):
   - One meeting was allocated for this stage, which took place on 4th June.
   - Start writing a draft of the report.
   - Integrate the various parts (ALU, multiplier, stack) with the overall CPU.
   - Start basic testing by checking individual instructions. Testing the three algorithms outlined by the specification is not required during this stage. However, the three source codes (provided in Appendix A) need to be translated to assembly code.
5. Testing and optimisation (to be completed by 12th June):
   - Five meetings were allocated for this stage, which occurred on 8th June, 9th June, 10th June, 11th June and 12th June.
   - Fix any bugs and incorrect instructions.
   - Pipeline the CPU.
   - Test with the required algorithms.
   - Remove any unnecessary components and reduce power consumption.
   - Finish writing a draft of the report.
6. Submission (to be completed by 14th June):
   - Two meetings were set for this stage, which occurred on 13th June and 14th June.
   - Fix any remaining bugs.
   - Complete the final report and create a video showcasing the CPU.

This plan can be summarised with the following table (Fig. 3.4a), which can further be used to develop a Gantt chart, as shown in Fig. 3.4b. The table shows each activity, alongside the stage during which the activity occurs and how long each stage should last, as set out in the plan.

Stage	Description	Start date	End date	Stage duration (days)
1	Setup Quartus and create a GitHub repostiory	Monday, 11 May 2020	Tuesday, 19 May 2020	8
1	Research modern CPU architectures	Monday, 11 May 2020	Tuesday, 19 May 2020	8
2	Decide on an instruction set and hardware capabilities	Tuesday, 19 May 2020	Friday, 22 May 2020	3
3	Implement control path	Friday, 22 May 2020	Tuesday, 2 June 2020	11
3	Implement data path	Friday, 22 May 2020	Tuesday, 2 June 2020	11
3	Implement an ALU	Friday, 22 May 2020	Tuesday, 2 June 2020	11
3	Implement a multiplier	Friday, 22 May 2020	Tuesday, 2 June 2020	11
3	Implement a stack	Friday, 22 May 2020	Tuesday, 2 June 2020	11
3	Write software for converting assembly to hexadecimal	Friday, 22 May 2020	Tuesday, 2 June 2020	11
4	Integration of various parts and initial testing	Tuesday, 2 June 2020	Saturday, 6 June 2020	4
5	Fix any remaining bugs	Saturday, 6 June 2020	Monday, 8 June 2020	2
5	Pipeline the CPU	Monday, 8 June 2020	Wednesday, 10 June 2020	2
5	Test with required algorithms and verification	Wednesday, 10 June 2020	Friday, 12 June 2020	2
5 and 6	Report writing	Tuesday, 2 June 2020	Sunday, 14 June 2020	12
6	Final review and submission	Friday, 12 June 2020	Sunday, 14 June 2020	2

Fig. 3.4a: Task summary

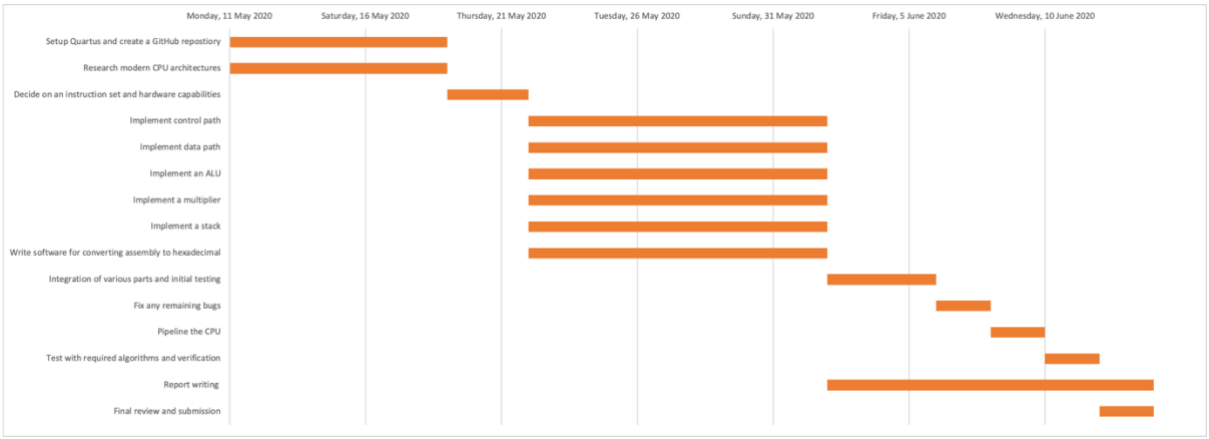


Fig. 3.4b: Gantt chart

Activity networks are another useful way of visualising project plans. The table shown in Fig. 3.4a was adjusted to include task dependencies and the expected duration of each task, rather than overall stage duration. Furthermore, the table includes the earliest day a task can start, based on previous tasks, as well as the latest day a task can be completed, as defined in the project plan. The adjusted table can be seen in Fig. 3.4c and the activity network is shown in Fig. 3.4d. The network consists of connected boxes; each box represents a single task. The values within each box are the earliest start day, expected duration and latest end day, from left to right.

Task	Description	Earliest start day	Latest end day	Expected duration	Depends on
A	Setup Quartus and create a GitHub repostiory	0	8	2	
B	Research modern CPU architectures	0	8	6	
C	Decide on an instruction set and hardware capabilities	6	11	3	B
D	Implement control path	9	22	4	A, B, C
E	Implement data path	9	22	4	A, B, C
F	Implement an ALU	9	22	4	A, C
G	Implement a multiplier	6	22	4	A, B
H	Implement a stack	2	22	3	A
I	Write software for converting assembly to hexadecimal	9	22	3	C
J	Integration of various parts and initial testing	13	26	4	D, E, F, G, H, I
K	Fixing any remaining bugs	17	28	2	J
L	Pipeline the CPU	19	30	2	K
M	Test with required algorithms and verification	21	32	2	L
N	Report writing	0	34	10	
O	Final review and submission	23	34	2	M, N

Fig. 3.4.c: Activity network table

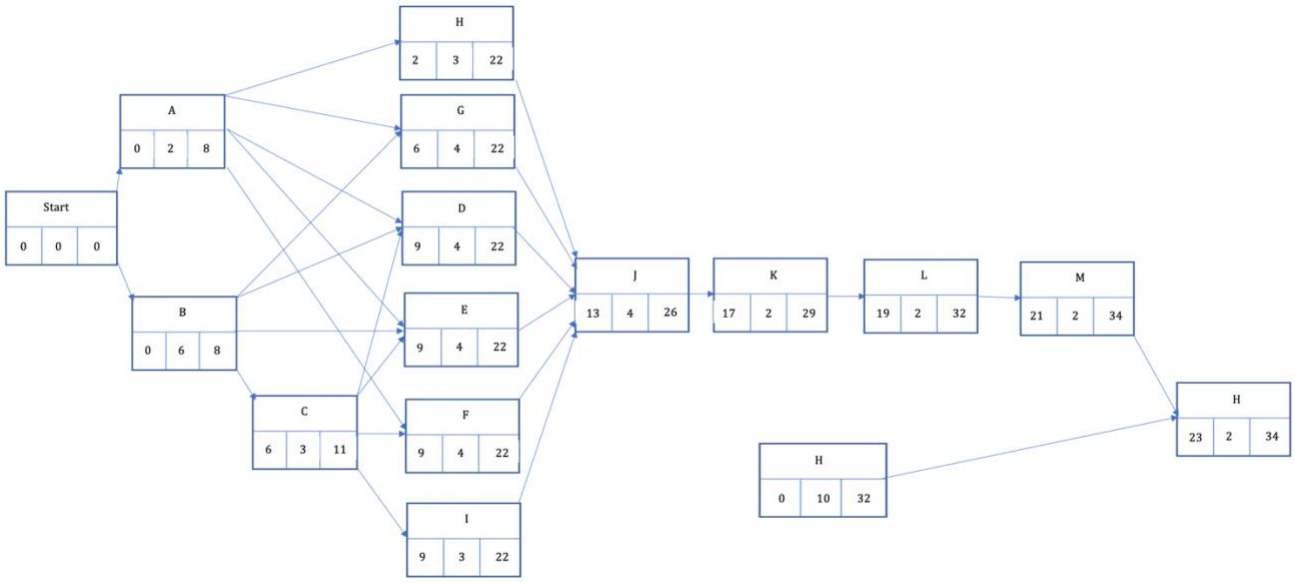


Fig. 3.4.d: Activity network

This network is useful for visualising dependencies between various parts of the project. However, the earliest start dates can be misleading: some tasks such as tasks D through I need to be implemented concurrently, which would increase the duration of each task. Therefore, while unlikely, it is possible to start initial testing and overall CPU integration (activity J) by the 13th day.

4. Project timeline

The following section describes the project progression, including meeting descriptions, tasks assigned to each team member, progress and any problems encountered along the way.

First meeting (12th May):

- Understand the specification, discuss future meetings, deadlines and team roles.
- We agreed on collaboration methods (described in the previous section), but GitHub was still not set up.
- The main goal before the next meeting was to research different ISAs and possible implementations of a multiplier.
- Every team member needed to set up Quartus by the next meeting.
- Next meeting was set for 16th May at 2 pm BST.

Second meeting (16th May):

- GitHub was still not set up due to issues with collaboration using Quartus; this needed to be completed as soon as possible.
- We decided to use separate memory units for instructions and data, based on the Harvard architecture, as it would make debugging easier and reduce power consumption.
- A possible implementation of multiplication was found – an array multiplier, built using partial products which are shifted and added. This was an acceptable approach at the time but was later replaced with a more elegant design (for details see subsection 5.10. 'Multiplier').
- Before the next meeting, each of the team members needed to read about the following topics:
    - Benjamin – AVR architecture, research implementations of a multiplication.
    - Aadi – SPARC and MIPS instruction set architecture, collaboration with Quartus files through GitHub.
    - Kacper – ARM architecture, pseudo-random number generator, brainstorm the initial design of the CPU.
- Next meeting was set for 20th May at 5 pm BST.

Third meeting (20th May):

- Aadi created a GitHub repository and added Benjamin and Kacper to it.
- Belbin roles needed to be completed by 20th May.
- The instruction set was finalised (see subsection 5.2. 'Instruction set'):
    - To allow working with more data, as described in project aims, eight registers would be used in the final CPU.
    - Two separate memory units would be used, as discussed in the previous meeting.
    - For non-memory instructions:
        - MSB is always 0, to distinguish from memory operations.
        - The following 6 bits determine the opcode of the given instruction.
        - The last 9 bits determine the three registers required for that instruction – 3 bits for each of the registers, those being the destination register and two source registers.
        - Note: this was later amended to include an indirect load and store instruction (see subsection 5.2. 'Load/store operations').
    - For memory instructions:
        - MSB is always 1, to distinguish from non-memory operations.
        - The next bit is used to distinguish between load and store, the only two possible instructions in this category.
        - The next 3 bits determine the register from which data is read or to which data is written
        - The next 11 bits are used for determining the memory location needed for this operation.
- Tasks to be completed before the next meeting:
    - Benjamin – further investigate possible implementations of multiplication circuits.
    - Aadi – document the instruction set with a description of each instruction and their opcodes.
    - Kacper – design the initial CPU.
- Next meeting was set for 24th May at 3 pm BST.

Fourth Meeting (24th May):

- Kacper finalised the initial CPU design.
- The instruction set was completed with preliminary documentation available in a Freehand diagram (included in Appendix B).
- Tasks to be completed before the next meeting:
    - Benjamin – implement and test the multiplier circuit.
    - Aadi – finish documenting the ISA and start working on the ALU.
    - Kacper – translate the initial design from the diagram into Quartus and ensure the control path works without the ALU and multiplier.
- Next meeting was set for 26th May at 3 pm BST.

Fifth meeting (26th May):

- Benjamin read about the multiplier: a possible implementation was through the use of lookup tables.
- Aadi implemented the ALU; however, it still needed to be integrated with the multiplier block.
- Initial CPU was almost implemented; the control path still needed some work.
- Tasks to be completed before the next meeting:
    - Benjamin – finish implementing the multiplier circuit, investigate possible implementations of a stack.
    - Aadi – finish the ALU, investigate possible implementations of a stack.
    - Kacper – finish implementing the control path, investigate possible implementations of a stack.
- Next meeting was set for 31st May at 4:30 pm BST.

Sixth Meeting (31st May):

- Multiplier was implemented.
- Started thinking about improving efficiency, mainly through pipelining.
- With the current instruction set, linked list traversal would take five cycles per item, provided the CPU is pipelined. We needed to investigate possible alternatives.
- Tasks to be completed before the next meeting:
    - Kacper – adjust the state machine to account for instructions that take three cycles, incorporate the ALU with the rest of the CPU, write a draft report for the decoder, state machine and the overall CPU.

    - Aadi – cooperate with Benjamin to finish the ALU and integrate it with the multiplier block, write a draft report of the ALU and start working on the stack.
    - Benjamin – cooperate with Aadi to finish ALU and integrate it with the multiplier block and write a draft report of the multiplier.
    - Everyone – brainstorm ideas for optimising the linked list.
- Started writing a draft version of the report.
- Next meeting was set for 4th June at 4:30 pm BST.

Seventh Meeting (4th June):

- The stack still needed to be implemented, implementations using Verilog were considered.
- Multiply block and ALU were completed and integrated but not tested.
- Benjamin created an assembly to MIF generator, written in C++.
- Tasks to be completed before the next meeting:
    - Benjamin – edit the report, translate pseudo-random generator source code into assembly code, figure out the shortest clock period of the design.
    - Kacper – edit the report, translate the linked list and Fibonacci source code into assembly, modify the decoder to integrate stack operations.
    - Aadi – implement and document the stack, start analysis for component power consumption.
- Next meeting was set for 7th June at 4:30 pm BST.

Eighth meeting (8th June):

- Kacper started testing the CPU. Multiplication did not work when integrated with the rest of the CPU, but it worked as a separate block.
- Benjamin completed the following parts of the draft: project management and progression.
- Aadi implemented the stack.
- Tasks to be completed before the next meeting:
    - Kacper – finish technical documentation and send it to Benjamin by June 8th, debug and test the rest of the CPU instructions, excluding multiplication.
    - Benjamin – finalise the draft, send it to Mrs Perea, and work on fixing the ALU-multiplier integration bugs.
    - Aadi – document the stack, debug the other instructions and work on fixing the ALU-multiplier integration bugs.
- Next meetings were set for 9th June at 4:30 pm BST and 10th June at 4 pm BST.

Ninth Meeting (9th June):

- Aadi was still trying to fix the multiplier block.
- Kacper tested the rest of the CPU. All the instructions were functional, except for multiplier commands.
- Tasks to be completed before the next meeting:
    - Kacper – document the ISA, apply fixes to the test code.
    - Benjamin – ensure the ALU and multiplier are fully functional.
    - Aadi – ensure the ALU and multiplier are fully functional.
- Next meeting was set for 10th June at 4 pm BST.

Tenth Meeting (10th June):

- Aadi managed to test the CPU, including the multiplier. Everything worked.
- The next stage was pipelining the CPU to increase speed.
- Benjamin started translating the C++ test codes into instructions executable by the CPU.
- The team started brainstorming ideas for the video and how to structure it.
- Indirect addressing was added as it is needed for traversing a linked list. This error was not spotted earlier, but the ISA is versatile and there were lots of opcodes left, so having to add a new instruction was not a significant setback.
- Tasks to be completed before the next meeting:
    - Kacper – pipeline the CPU and document it.
    - Ben – finish translating the C++ code into assembly code.
    - Aadi – plan out the video.
- Next meeting was set for 11th June at 4 pm BST.

Eleventh Meeting (11th June):

- Kacper pipelined and almost completed the CPU.
- Ben finished translating the C++ code into instructions executable by the CPU.
- Tasks to be completed before the next meeting:
    - Kacper – test the CPU with the translated C++ tests.
    - Benjamin– write the report introduction.
    - Aadi – test the CPU with the translated C++ tests.
- Next meeting was set for 12th June at 4 pm BST.

Twelfth and Last Meeting (12th June):

- Kacper added a JMA instruction for immediate jumps to a given address. This eliminated the need to use a load operation before a jump but is limited to addresses up to 0x1FF.
- Kacper added the CLL and RTN instructions to ease the implementation of the Fibonacci test code.
- Tasks to be completed before the next meeting:
    - Benjamin – test the pipelined CPU with the random number generator algorithm, work on translating the Fibonacci code and write an introduction and conclusion, perform final edits of the report.
    - Kacper – test the pipelined CPU with the linked list code, work on translating the Fibonacci code, plan, record and upload the video.
    - Aadi – finish analysing the CPU performance and work on the Appendix, perform final edits of the report.
- Submission and final edits were coordinated informally and not documented.

5. Design process

5.1. Overview of the CPU

The final design was based on multiple architectures. After researching ARM, MIPS, SPARC and AVR, we chose to include the following in our design:

- General features of a RISC machine.
- An ARMv8 load/store architecture and some instructions from this architecture such as multiply and add/subtract (MLA/MLS), no operation (NOP) and various shifts (ROR and RRC) [1].
- Splitting instructions into different formats, like the MIPS architecture [2]; for our design, this meant splitting instructions into two categories, memory and non-memory operations (see subsection 5.2. 'Instruction set').

The main goal of this approach was to make each instruction take as few cycles as possible, ideally just one, while avoiding any specific instructions, such as an instruction for calculating the n^{th} Fibonacci number. The diagram in Fig 5.1 shows a block diagram of the CPU. Signals such as the input clock and control lines are omitted from this diagram to make it simpler. The data path is shown in blue and the control path is purple. Green blocks represent memory units/registers, while grey ones are asynchronous logic (except the multiplier located inside the ALU). The CPU consists of the following components which are described in the following subsections (for detailed block diagrams and Verilog code, see Appendix C):

- State Machine
- Decoder block
- Instruction memory unit
- Data memory unit
- Register file
- Add 1 logic block
- Arithmetic Logic Unit (ALU)
    - General ALU
    - Multiplier
- Last-in First-out (LIFO) stack buffer
- Multiplexers

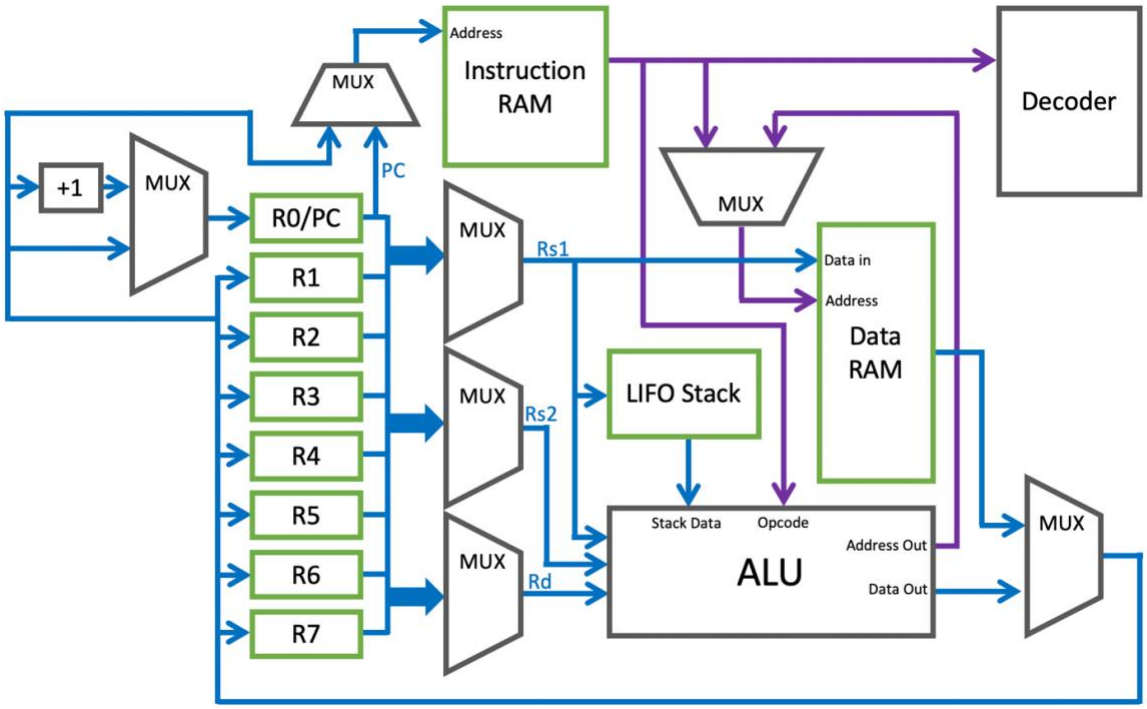


Fig. 5.1: Simplified CPU block diagram

5.2. Instruction set

5.2.1. Overview and initial ideas

One of the most crucial early-stage design decisions was the creation of an instruction set architecture (ISA) which the CPU's implementation would be based on. Upon researching different ISAs, the decision was made to proceed with an ISA similar to ARMv8 [1] which would have two types of encoding for instructions – load/store encoding and three operand instruction encoding. This general format (like many other RISC architectures) was chosen as it fits the functional requirement of 16-bit long instruction words and provided space for up to 8 registers in the CPU.

Generally, all instructions can be classified into one of these categories:

- Load/store operations
- Arithmetic
- Logical
- Shifts
- Jumps
- Stack manipulators
- Others

The CPU can accommodate up to 64 different commands due to the size of the opcode field and therefore can be expanded by adding new instructions and hardware when the current design does not meet a specific need or if a high degree of specialisation is required for a given implementation. For instructions other than the memory load/store operations, the encoding can be summarised in terms of bits as:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	Opcode						Rd: destination register			Rs1: source register 1			Rs2: source register 2		

5.2.2. Load/store operations

These instructions are used to manipulate the data RAM to retrieve values or save new values into it from one of the CPU's registers. The LDA and STA instructions require 11 bits in the instruction word for the memory address and due to this need, their format is different from all other instructions in the ISA. The most significant bit (MSB) of the instruction word is 1 which distinguishes them from all other commands. The other two available commands follow the standard pattern and start with 0. The instructions available in this category are:

- **LDA (load direct address):** load Rx with the value found at the specified memory address in the data RAM (Rx = Mem[Memory Address])
- **STA (store direct address):** store the value of Rx at the specified memory address in the data RAM (Mem[Memory Address] = Rx)
- **LDR (load indirect address):** load Rd with the value found at the memory address specified by Rs1 in the data RAM (Rd = Mem[Rs1])
- **STR (store indirect address):** store the value of Rs1 at the memory address specified by Rd in the data RAM (Mem[Rd] = Rs1)

The encoding of these instructions is summarised below. Rx represents the register on which the load/store operation is performed in the case of LDA and STA. In the case of LDR and STR, Rd is the register to which the value will be read from/stored to (respectively) and Rs1 is the register which contains the memory address.

Instruction	**Bits of Instruction Word**															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LDA	1	0	Rx			Memory Address										
STA		1														
LDR	0	1	0	1	0	1	0	Rd			Rs1			Unused		
STR						1	1									

5.2.3. Arithmetic operations

The CPU's Arithmetic Logic Unit (ALU) can perform basic arithmetic operations on 16-bit numbers like addition, subtraction and multiplication (division was not implemented due to complex high hardware requirements). The implementation of the MLA command was essential since it combines two operations into one command which shortens the operation A = A*B+C from taking 3 clock cycles (multiply then add) to just 2. This instruction was included in the ISA as this specific operation was required to complete the Linear Congruential Generator C++ code test (see subsection 2.2. 'Project outline' and Appendix A). The ADC and SBC operations (with carry) were implemented as additional operations to account for the use of 32-bit integers by the user if desired, even though this is strongly discouraged. The available operations are:

- **ADD (add):** Rd = Rs1 + Rs2
- **ADC (add with carry):** Rd = Rs1 + Rs2 + CARRY
- **ADO (add 1):** Rd = Rs1 + 1
- **SUB (subtract):** Rd = Rs1 – Rs2
- **SBC (subtract with carry):** Rd = Rs1 – Rs2 + CARRY – 1
- **SBO (subtract 1):** Rd = Rs1 – 1
- **MUL (multiply):** Rd = Rs1 * Rs2
- **MLA (multiply and add):** Rd = (Rd * Rs1) + Rs2
- **MLS (multiply and subtract):** Rd = Rs2 – (Rd * Rs1)

The format of each of these instructions is summarised in the following table. Rd represents the destination register while Rs1 and Rs2 represent the source registers 1 and 2, respectively.

<table>
<tr><th rowspan="2">Instruction</th><th colspan="16">Bits of Instruction Word</th></tr>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>ADD</td><td rowspan="10">0</td><td rowspan="3">0</td><td rowspan="3">1</td><td rowspan="3">0</td><td rowspan="3">1</td><td>0</td><td>0</td><td rowspan="10" colspan="4">Rd</td><td rowspan="10" colspan="4">Rs1</td><td rowspan="3" colspan="3">Rs2</td></tr>
<tr><td>ADC</td><td>0</td><td>1</td></tr>
<tr><td>ADO</td><td>1</td><td>0</td><td colspan="3">Unused</td></tr>
<tr><td>SUB</td><td rowspan="3">0</td><td rowspan="3">1</td><td rowspan="3">1</td><td rowspan="3">0</td><td>0</td><td>0</td><td rowspan="3" colspan="3">Rs2</td></tr>
<tr><td>SBC</td><td>0</td><td>1</td></tr>
<tr><td>SBO</td><td>1</td><td>0</td><td colspan="3">Unused</td></tr>
<tr><td>MUL</td><td rowspan="3">0</td><td rowspan="3">1</td><td rowspan="3">1</td><td rowspan="3">1</td><td>0</td><td>0</td><td rowspan="3" colspan="3">Rs2</td></tr>
<tr><td>MLA</td><td>0</td><td>1</td></tr>
<tr><td>MLS</td><td>1</td><td>0</td></tr>
</table>

5.2.4. Logical operations

The CPU is also capable of performing some basic logical operations. Although these are not needed to execute the specified tasks outlined in the three test codes (see subsection 2.1. 'Project outline' and Appendix A), they are included in the ISA for the sake of completeness and versatility. The operations that fall under this category are:

- **AND:** Rd = Rs1 AND Rs2
- **OR:** Rd = Rs1 OR Rs2
- **XOR:** Rd = Rs1 XOR Rs2
- **NOT:** Rd = NOT Rs1
- **NAND (NND):** Rd = NOT (Rs1 AND Rs2)
- **NOR:** Rd = NOT (Rs1 OR Rs2)
- **XNOR (XNR):** Rd = NOT (Rs1 XOR Rs2)

The format of each of these instructions is summarised in the following table. Rd represents the destination register while Rs1 and Rs2 represent the source registers 1 and 2, respectively.

<table>
<tr><th rowspan="2">Instruction</th><th colspan="16">Bits of Instruction Word</th></tr>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>AND</td><td rowspan="8">0</td><td rowspan="4">0</td><td rowspan="4">0</td><td rowspan="4">1</td><td rowspan="4">1</td><td>0</td><td>0</td><td rowspan="8" colspan="4">Rd</td><td rowspan="8" colspan="4">Rs1</td><td rowspan="3" colspan="3">Rs2</td></tr>
<tr><td>OR</td><td>0</td><td>1</td></tr>
<tr><td>XOR</td><td>1</td><td>0</td></tr>
<tr><td>NOT</td><td>1</td><td>1</td><td colspan="3">Unused</td></tr>
<tr><td>NND</td><td rowspan="3">0</td><td rowspan="3">1</td><td rowspan="3">0</td><td rowspan="3">0</td><td>0</td><td>0</td><td rowspan="3" colspan="3">Rs2</td></tr>
<tr><td>NOR</td><td>0</td><td>1</td></tr>
<tr><td>XNR</td><td>1</td><td>1</td></tr>
</table>

5.2.5. Shift operations

The ALU can perform asynchronous shifts by a specified number of places. These are not directly required for any of the testbench codes (see subsection 2.1. 'Project outline' and Appendix A), but they provide a useful addition to the CPU. Shifts can be used to perform quick multiplication and division by powers of 2. For a detailed description of the functionality of the ASR, ROR, and RRC instructions, see Appendix D. The available operations in this category are:

- **LSL (logical shift left):** Rd = Rs1 shifted left by Rs2 number of places
- **LSR (logical shift right):** Rd = Rs1 shifted right by Rs2 number of places without copying the most significant bit

- **ASR (arithmetic shift right):** Rd = Rs1 shifted right by Rs2 number of places with the sign of Rs1 preserved through sign extension (shifting in the most significant bit of Rs1 for every shift)
- **ROR (shift right loop):** Rd = Rs1 shifted right by Rs2 number of places with the value shifted in being the least significant bit of Rs1 with every shift
- [DEPRECATED] **RRC (shift right loop with carry):** Rd = Rs1 shifted right by Rs2 number of places with the value of the carry flip-flop of the ALU shifted in and the least significant bit of Rs1 saved into the carry flip-flop with every shift

The RRC instruction was removed after a timing and resource analysis of the CPU (see section 6. 'Final analysis of design'). The format of each of these instructions is summarised in the following table. Rd represents the destination register while Rs1 and Rs2 represent the source registers 1 and 2, respectively.

<table>
<tr><th rowspan="2">Instruction</th><th colspan="16">Bits of Instruction Word</th></tr>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>LSL</td><td rowspan="5">0</td><td rowspan="3">1</td><td rowspan="3">0</td><td rowspan="3">0</td><td rowspan="3">0</td><td>0</td><td>0</td><td rowspan="5" colspan="3">Rd</td><td rowspan="5" colspan="4">Rs1</td><td rowspan="5" colspan="4">Rs2</td></tr>
<tr><td>LSR</td><td>0</td><td>1</td></tr>
<tr><td>ASR</td><td>1</td><td>0</td></tr>
<tr><td>ROR</td><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">1</td><td>0</td><td>0</td></tr>
<tr><td>RRC</td><td>0</td><td>1</td></tr>
</table>

5.2.6. Jump operations

The jump operations allow the user to directly manipulate the value of the program counter (PC) to skip certain instructions or create conditional loops. Both conditional jumps and an unconditional jump are implemented to allow a high degree of flexibility. The conditional jumps also provide comparisons of two numbers. JMA was added as an alternative to JMP, without needing a load operation beforehand; however, it is important to mention that JMA cannot reach all addresses in the instruction RAM. The available jump instructions are:

- **JMP (unconditional jump):** unconditional jump to instruction with address Rd
- **JMA (unconditional jump to address):** jump to instruction at the address specified (note: maximum address available is 0x1FF due to the size of the 'Jump Address' field)
- **JC1 (conditional jump 1):** jump to instruction with address Rd if Rs1 < Rs2
- **JC2 (conditional jump 2):** jump to instruction with address Rd if Rs1 > Rs2
- **JC3 (conditional jump 3):** jump to instruction with address Rd if Rs1 = Rs2
- **JC4 (conditional jump 4):** jump to instruction with index Rd if Rs1 = 0

- **JC5 (conditional jump 5):** jump to instruction with index Rd if Rs1 ≥ Rs2
- **JC6 (conditional jump 6):** jump to instruction with index Rd if Rs1 ≤ Rs2
- **JC7 (conditional jump 7):** jump to instruction with index Rd if Rs1 ≠ Rs2
- **JC8 (conditional jump 8):** jump to instruction with index Rd if Rs1 < 0

The format of each of these instructions is summarised in the following table. Rd represents the destination register (in this case, the destination of the jump) while Rs1 and Rs2 represent the source registers 1 and 2, respectively.

<table>
<tr><th rowspan="2">Instruction</th><th colspan="17">Bits of Instruction Word</th></tr>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>JMP</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td colspan="3">Rd</td><td colspan="7">Unused</td></tr>
<tr><td>JMA</td><td></td><td></td><td></td><td></td><td></td><td>0</td><td>1</td><td colspan="10">Jump Address</td></tr>
<tr><td>JC1</td><td></td><td rowspan="4">0</td><td rowspan="4">0</td><td rowspan="4">0</td><td rowspan="4">1</td><td>0</td><td>0</td><td rowspan="8" colspan="3">Rd</td><td rowspan="8" colspan="5">Rs1</td><td rowspan="3" colspan="2">Rs2</td></tr>
<tr><td>JC2</td><td></td><td>0</td><td>1</td></tr>
<tr><td>JC3</td><td></td><td>1</td><td>0</td></tr>
<tr><td>JC4</td><td></td><td>1</td><td>1</td><td colspan="2">Unused</td></tr>
<tr><td>JC5</td><td></td><td rowspan="4">0</td><td rowspan="4">0</td><td rowspan="4">1</td><td rowspan="4">0</td><td>0</td><td>0</td><td rowspan="3" colspan="2">Rs2</td></tr>
<tr><td>JC6</td><td></td><td>0</td><td>1</td></tr>
<tr><td>JC7</td><td></td><td>1</td><td>0</td></tr>
<tr><td>JC8</td><td></td><td>1</td><td>1</td><td colspan="2">Unused</td></tr>
</table>

5.2.7. Stack operations

These operations manipulate the stack, either by saving (pushing) a new value to it from a specified register or by retrieving (popping) the latest value pushed onto the stack and saving it in a specified register. These simple operations allow for primitive subroutines to be constructed in assembly language (CLL and RTN commands can also be used; for details see subsection 5.2.8. 'Other operations'); however, they require the user to think more intensively about their layout in comparison to the simple-to-use BX and BL instructions of the ARMv8 architecture. The stack is a last-in-first-out buffer, meaning that the value available for retrieval is the latest pushed to it. For a detailed description of the stack's operation and the reasoning behind its implementation, refer to subsection 5.11. 'Stack'. The operations in this category are:

- **PSH (push):** push the value of a register (Rs1) onto the stack
- **POP (pop):** save the latest value pushed onto the stack in a register (Rd)

The format of these instructions is summarised in the following table. Rd represents the destination register while Rs1 represents the source register 1. These instructions only take one register operand each. Their register fields are different to make it clear what functionality they perform (PSH requires a source, POP requires a destination).

Instruction	Bits of Instruction Word															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PSH	0	1	0	1	0	0	0	Unused			Rs1			Unused		
POP						0	1	Rd			Unused					

5.2.8. Other operations

The operations in this category cannot be classified into any of the categories mentioned previously. Nevertheless, their importance in the correct and efficient operation of the CPU is crucial. The CLL and RTN instructions were added to allow the user to easily create basic subroutines and nested functions in assembly language (for a detailed description of their operation see Appendix E). The operations in this category include:

- **MOV (move):** Rd = Rs1
- **CLL (make call):** save the PC's value to the stack and jump to the instruction at the memory address specified in Rd
- **RTN (return from call):** load the latest value from the stack to the PC and jump to the instruction at the memory address of that value
- **NOP (no operation):** do nothing for a clock cycle/wait
- **STP (stop):** halts all CPU operations, clears the stack, and signifies the end of the list of instructions/program

The format of these instructions is summarised in the following table. Rd represents the destination register while Rs1 represents the source register 1.

Instruction	Bits of Instruction Word															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOV	0	0	1	0	0	1	1	Rd			Rs1			Unused		
CLL		1	0	1	0	1	0	Rd			Unused					
RTN						1	1	Unused								
NOP		1	1	1	1	1	0									
STP						1	1									

5.3. State machine

A crucial element of any CPU is the state machine. This design includes a simple state machine with three possible states:

- FETCH (001) – the cycle during which the instruction memory is read and the instruction that needs to be executed appears at its output.
- EXEC1 (010) – the execution cycle during which all necessary operations are performed, as specified by the current instruction.
- EXEC2 (100) – an optional third cycle needed for the execution of some instructions, such as multiplication and load instructions.

The following Moore state diagram shows the transition between the three states, dependent on the inputs, RST and E2. The input RST resets the state machine to FETCH. The input signal E2 determines whether a third cycle is needed, depending on the current instruction. The 000 state (not shown) that occurs during the initialisation of the state machine leads directly to the FETCH state on the next rising edge of the clock without considering any of the inputs.

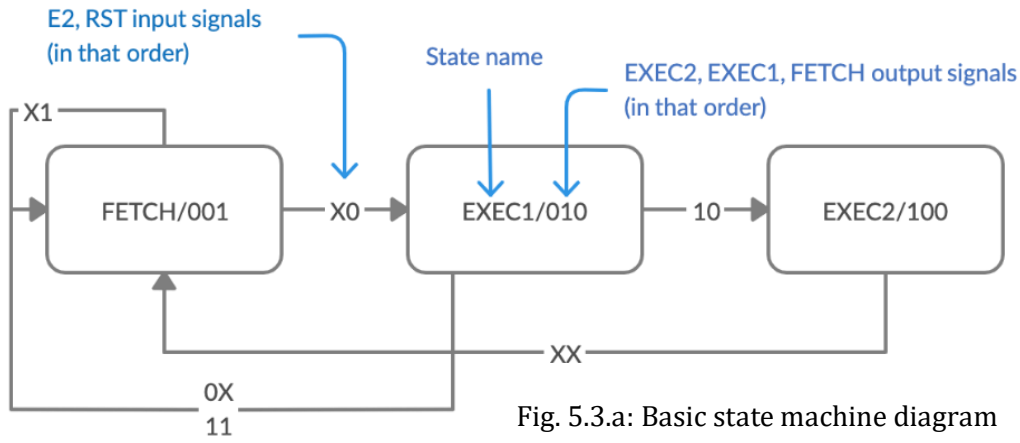


Fig. 5.3.a: Basic state machine diagram

Pipelining is a simple way of optimising digital circuits. This means that the CPU would stay in the EXEC1 cycle all the time, unless an instruction required an extra cycle to complete, in which case the EXEC2 cycle would occur. The following changes were made to the state machine to achieve pipelining:

- The internal state numbers were changed to the ones shown in brackets:
  - FETCH (00) – initial state; it serves the same function as previously but needed to be kept in for the very first cycle after initialising the CPU, i.e. the first command still needs to be fetched.
  - EXEC1 (01) – same function as previously.
  - EXEC2 (10) – same function as previously.

The new Moore state diagram of the pipelined state machine is shown below.

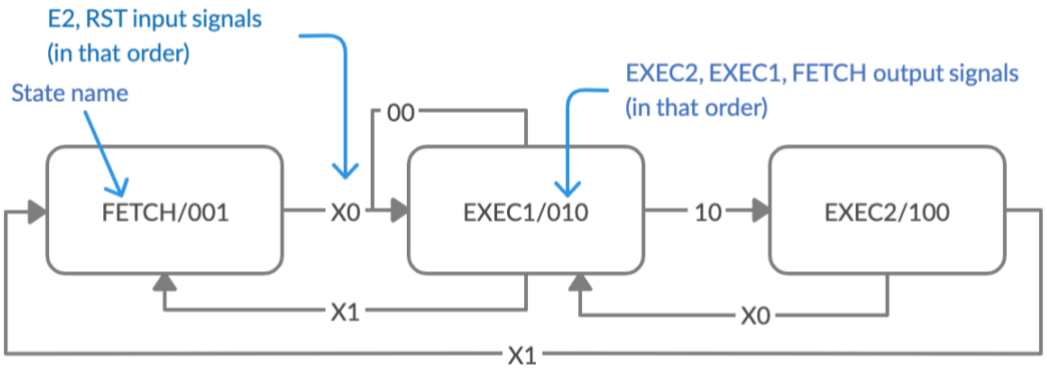


Fig. 5.3.b: Pipelined state machine diagram

5.4. Instruction memory unit

The design includes a 16-bit word length RAM unit that can hold up to 2048 memory words, as required by the specification. This RAM unit only holds the instructions. A separate RAM unit is used to store data. This separation was chosen to decrease the number of components needed for the CPU to operate, as explained below.

This RAM unit is mostly independent of the rest of the CPU and it acts as read-only memory. This implementation was chosen over a ROM unit because using a RAM gives the CPU versatility as an element of a larger machine, such as a computer or smartphone since new instructions can be loaded if required to do so.

The RAM also has a clock enable port. This was added to turn off the RAM when not in use, but more importantly, to ensure correct timing of the instructions that appear at its output. When a new instruction is read, this control port becomes de-asserted (active high) until the execution of the new instruction is completed. Only then is the next instruction read by enabling the control input. This means that there is no need for an instruction register to hold the currently executed instruction. This decreases the number of components as well as power consumption.

5.5. Data memory unit

The second RAM unit is identical to the instruction RAM; however, it is only used for storing data such as 16-bit C++ integers specified in the project specification. Like the instruction RAM, the capacity is also 2048 words of data, each 16 bits long. This RAM also has a clock enable port, but in this case, it was only added to reduce power consumption of the CPU. The data RAM unit can store new data at request, using a STORE command, with its data input port connected to the destination register multiplexer (see subsection 5.2. 'Instruction set' and Appendix C).

5.6. Register file

The register file contains eight 16-bit registers, which are implemented using D-type flip-flops. The first register, R0, performs a specific function: it acts as a program counter (PC). This allows direct manipulation of the PC during jump commands without any additional commands or hardware.

The final decision was to include eight registers in the CPU as eight registers provide a reasonable degree of versatility while being able to fit in with the 16-bit instruction word. The register file concept was mainly based on the ARM architecture [3].

5.7. Add 1 logic block

The "+1" logic block is a simple adder. Its only function is to add 1 to the current value at its input. This output is then passed to the multiplexer at the input of the PC and is saved to it during the next rising clock edge. This block was needed to pipeline the jumps and thus achieve a completely pipelined CPU. During a jump, the address of the next instruction is specified by one of the registers. The PC needs to load this value and also count up by one in the same cycle, which is not possible (the Quartus lpm_counter megafunction used for the PC is not able to count up and load at the same time). The solution to this problem was simple – increment the required value by one before loading it into the PC. This Add 1 logic block is used to achieve this.

5.8. Decoder

The decoder block is an integral part of the design. It controls many operations of the CPU and ensures correct execution of all commands: it is responsible for controlling different components and registers based on the current instruction. The internal logic is complex (annotated code is provided in Appendix F), but it can be summarised in terms of output signals, which are active high unless stated otherwise:

- ***R0_count*** – causes the program counter (PC) to increment its value by 1 when asserted.
- ***R0_en through R7_en*** – enable signals for the eight registers. The registers save the value at their inputs when these signals are asserted.
- ***s1 through s6*** – select signals for the six multiplexers in the CPU (note: the multiplexer at the input of the PC is controlled by the *ADD1_en* signal; see below).
- ***RAMd_wren*** – enables writing to the data RAM when asserted; otherwise, data is read from this memory unit.

- ***RAMd_en*** and ***RAMi_en*** – clock enable signals for the two RAM units. These were added to conserve power (prevent switching when the blocks are not in use) and remove the need for an instruction register. When not asserted, the units are disabled.
- ***ALU_en*** – an enable signal for the ALU (active low), that was added to ensure that no undefined behaviour exists during LOAD and STORE operations and to conserve power.
- ***E2*** – a control signal for the state machine. When asserted it causes an extra cycle to occur, which is needed for some instructions.
- ***Stack_en*** – an enable signal for the stack; added to conserve power.
- ***Stack_rst*** – a signal that clears the stack when asserted.
- ***Stack_rw*** – a signal that causes the stack to save the value at its input when asserted. Otherwise, a read operation is performed.
- ***ADD1_en*** – an enable signal that controls the operation of the "+1" block. When asserted, the multiplexer at the input of the PC chooses this block's output to be fed into the PC.

5.9. Arithmetic logic unit (ALU)

The goal of the Verilog Arithmetic Logic Unit (ALU) is to complete any logical or arithmetic operations efficiently so that the result can be saved back to the required register. Hence, the ALU must be asynchronous to avoid the need for extra cycles. The only delay between the inputs to the ALU and the result is due to the propagation delay of the logic within the block itself.

The ALU is designed and implemented as a Verilog HDL block. This is achieved using an 'always' block with an automatic sensitivity list. Inside this block, a 'switch' in the form of a 'case' statement is used, which, based on the opcode and the register inputs, performs the correct operation. The output is updated whenever the opcode changes. Therefore, a clock is not required for the ALU. Using the *enable* input (active low) can disable the ALU during load/store operations to conserve power and avoid any unintended operations.

The inputs and outputs are given the "signed" flag, which indicates the use of 2's complement. This allows for easy comparison between Rs1 and Rs2, especially for conditional jump instructions.

Fig. 5.9 shows a snippet of Verilog code (for a more detailed description, see Appendix G).

```
always @(opcode, mulresult)
   begin
      if(!enable) begin
         case (opcode)
            6'b000000: alusum = {1'b1, Rd}; // JMP Unconditional Jump, first bit high to indicate jump and passes through Rd

            6'b000100: alusum = {JC1, Rd}; // JC1 Conditional Jump A < B
            6'b000101: alusum = {JC2, Rd}; // JC2 Conditional Jump A > B
            6'b000110: alusum = {JC3, Rd}; // JC3 Conditional Jump A = B
            6'b000111: alusum = {JC4, Rd}; // JC4 Conditional Jump A = 0

            6'b001000: alusum = {JC5, Rd}; // JC5 Conditional Jump A >= B / A !< B
            6'b001001: alusum = {JC6, Rd}; // JC6 Conditional Jump A <= B / A !> B
            6'b001010: alusum = {JC7, Rd}; // JC7 Conditional Jump A != B
            6'b001011: alusum = {JC8, Rd}; // JC8 Conditional Jump A < 0

            6'b001100: alusum = {1'b0, Rs1 & Rs2}; // AND Bitwise AND
            6'b001101: alusum = {1'b0, Rs1 | Rs2}; // OR Bitwise OR
            6'b001110: alusum = {1'b0, Rs1 ^ Rs2}; // XOR Bitwise XOR
            6'b001111: alusum = {1'b0, ~Rs1}; // NOT Bitwise NOT

            6'b010000: alusum = {1'b0, ~Rs1 | ~Rs2}; // NND Bitwise NAND
            6'b010001: alusum = {1'b0, ~Rs1 & ~Rs2}; // NOR Bitwise NOR
            6'b010010: alusum = {1'b0, Rs1 ~^ Rs2}; // XNR Bitwise XNOR
            6'b010011: alusum = {1'b0, Rs1}; // MOV Move (Rd = Rs1)

            6'b010100: begin
                  alusum = {1'b0, Rs1} + {1'b0, Rs2}; // ADD Add (Rd = Rs1 + Rs2)
                  carry = alusum[16];
               end
```

Fig. 5.9: Snippet of the ALU's Verilog HDL code

5.10. Multiplier

5.10.1. Initial ideas

The multiplier is an integral part of any ALU, which meant that the final design needed to be both power- and speed-efficient. Considering these requirements, initial ideas came from some basic multiplier designs: an array multiplier or one built using the Ancient Egyptian/Russian peasant algorithm. However, neither of these were suitable for the project requirements:

An array multiplier is built using binary multiplication rules, forming partial products using bits of the two numbers being multiplied. The partial products are shifted and added, as explained in the lecture slides from MIT [4]. The main advantage of this design is that only combinational logic is used, meaning multiplication can be completed in one cycle. As pointed out by M. Moeng and J. Wei [5], implementing an N-bit array multiplier would require N^2 full adders and N^2 AND gates. Therefore, this implementation comes with two significant drawbacks: high power consumption and more importantly, complex debugging. For a 16-bit multiplier, this implementation would require 256 full adders and 256 AND gates (debugging it would be close to impossible). Another potential drawback could be a high propagation delay, due to the high number of logic gates, however this was not investigated in detail. Numerous optimisations of the basic array multiplier exist, the most notable being the Wallace multiplier [6], but the final decision was to try a different approach.

Ancient Egyptian/Russian peasant algorithm [7]: A reasonably straightforward algorithm, where one of the multiplied numbers is repeatedly divided by 2, while the other is repeatedly multiplied by 2. Initially, the product is set to 0. If the result of the division is odd, add the multiplied number to the result. This process is repeated until repeated division by 2 produces 1. Implementing this would be relatively easy since the algorithm requires two shift registers and a

32-bit adder. The biggest drawback of this implementation is that multiplication could take up to 16 cycles! Furthermore, this implementation would make it very hard to pipeline the CPU. This design would be a good fit for circuits where speed does not matter as much as it does in a CPU since it is easy to implement and requires very few components. An application of this algorithm which comes to mind would be a basic calculator, which acts as an ALU with only one arithmetic instruction being executed at any given time.

5.10.2. Lookup table and Karatsuba's algorithm

The final implementation was based on the circuit designed by M. Moeng and J. Wei [5]. This circuit provides a nice balance between the number of cycles to complete the multiplication and the complexity of the circuit. The main difference between the design proposed in [5] and our implementation is the lack of pipelining. It was not necessary to pipeline the multiplier, as the overall CPU was pipelined.

The circuit is built using a lookup table, the Karatsuba algorithm and a multiplication property observed by Ling [8] and Vinnakota [9].

Multiplication property: If two numbers, A and B (where B is smaller than or equal to A) are being multiplied, then the product P=AB, is given by the algorithm below; for proof and examples see publications from Ling [8] and Vinnakota [9].

$$\text{Let } x = \left\lfloor \frac{A+B}{2} \right\rfloor \text{ and } y = \left\lfloor \frac{A-B}{2} \right\rfloor; \qquad \text{then P is given by:}$$

$P = x^2 - y^2$, if the numbers A and B have the same parity and
$P = x^2 - y^2 + B$, otherwise.

The Karatsuba algorithm: Let X be an N-bit binary number: then define X as $\{X_1, X_0\}$, where X_1 represents the first N/2 bits and X_0 is the second group of N/2 bits. The algorithm states:

1. For A and B, define $\{A_1, A_0\}$ and $\{B_1, B_0\}$, where A and B are N-bit long binary numbers
2. Form the following products using an N/2-bit multiplier: $P_1=A_1B_1$, $P_2=A_1B_0$, $P_3=A_0B_1$ and $P_4=A_0B_0$.
3. Let X << K, denote a left shift of X by K bits. Then the final product of A and B is given by $P = (P_1 << N) + (P_2 << N/2) + (P_3 << N/2) + P_4$

Lookup table: The purpose of the lookup table is to store all the squares of integers up to and including 255 so that they are available at run-time. These are used for 8-bit multiplication, using the property described above. The lookup table was implemented using a ROM unit with 256 words, with each word 16 bits long. The ROM is used to store the squares of 8-bit numbers: with

8-bits, there are 256 possible numbers and, at most, the square of such a number will require 16 bits. The .mif file was generated using a C++ program.

5.10.2. Final implementation and verification

The final implementation consisted of four 8-bit multipliers and three 32-bit adders. First, the 8-bit multiplier was designed and tested.

The 8-bit multiplier was implemented using the previously mentioned multiplication property and a lookup table. Firstly, two arithmetic components and a right shift (which is equivalent to division by 2 and the floor function) are used to calculate $x = \left\lfloor \frac{A+B}{2} \right\rfloor$ and $y = \left\lfloor \frac{A-B}{2} \right\rfloor$. Values x and y are then fed to the ROM unit as addresses to the lookup table, to obtain their squares. Address N in the lookup table contains the square of number N. To determine whether to add B or not to the final product, a multiplexer was used with a select line obtained from an XOR gate of the least significant bits of the two multiplied numbers (checking the parity of the two numbers in accordance with the multiplication property).

While designing this circuit, a timing problem was encountered - the ROM unit could not be made asynchronous within the Cyclone IV family. The initial plan was to have the lookup table asynchronous so that that multiplication could be completed in one cycle. However, this was not possible, so the decision was to proceed with a synchronous implementation. Even though three cycles would make it slightly harder to pipeline the overall CPU, the CPU already needed to support three-cycle, such as loading instructions.

To complete the 16-bit multiplier, the final circuit made use of the 8-bit multiplier and the Karatsuba algorithm, as described above. The partial products are shifted left and added using three 32-bit arithmetic components. The final circuit can be seen in Appendix I.

5.11. Stack

A major advantage of this CPU is the ability to run subroutines and nested functions. One way of handling these is to store the values needed after the end of each subroutine to the data RAM and load them back into the registers once the call is completed. However, this is not a practical implementation as there is only a limited number of registers available. The alternative is to have a temporary storage block for these variables, which would be quicker to access than a RAM and would allow for nested subroutines. A stack is usually implemented either using specific branch opcodes (such as in ARMv8 [1]) or by using a stack block which is similar to a RAM unit but without control over which memory locations are written or read, hence requiring fewer logic components and a smaller chip area.

All the memory megafunction blocks provided in Quartus are of the FIFO (First-in First-out) type, which are usually used for a queue or list where the order of the data must be maintained. However, when executing a subroutine, the data from previous subroutines must be returned to the registers as the program steps out of each subroutine in order. This required a LIFO (Last-in First-out) buffer block.

The LIFO stack buffer was implemented as a Verilog HDL block based on code by an anonymous author [10] as shown below.

```
module LIFOstack (Din, clk, en, rst, rw, Dout, empty, full);

input [15:0] Din; // Data being fed to stack
input clk; // clock signal input
input en; // disable stack when not in use
input rst; // reset pin to clear and reinitialise stack (active high)
input rw; // 0: read, 1: write

output reg [15:0] Dout; // Data being pulled from stack
output reg empty; // goes high to indicate SP is at 0
output reg full; // goes high to indicate SP is at (slots)

reg [5:0] SP; // Points to slot to save next value to
integer i;
reg [15:0] mem [31:0];

always @ (posedge clk) begin
      if (!en); // if not enabled, ignore this cycle
      else begin
            if (rst) begin // if rst is high, clear memory and reset pointers/outputs
                  Dout = 16'h0000;
                  SP = 6'b000000;
                  empty = 1'b1;
                  for (i = 0; i < 32; i = i + 1) begin
                     mem[i] = 16'h0000;
                     end
               end
            else begin
                  if (!full && rw) begin // Write when NOT full & Writing
                        mem[SP] = Din; // Store data into current slot
                        SP = SP + 1'b1; // Increment stack pointer to next empty slot
                        full = (SP == 6'b100000) ? 1 : 0; // Stack is full if SP is (slots)
                        empty = 1'b0; // Stack is never empty after a push
                     end
                  else if (!empty && !rw) begin // Read when NOT empty & reading
                        SP = SP - 1'b1; // Decrement stack pointer to last filled slot
                        Dout = mem[SP]; // Output data from last filled slot
                        mem[SP] = 16'h0000; // Clear slot after setting output
                        full = 1'b0; // Stack is never full after a pop
                        empty = (SP == 6'b000000) ? 1 : 0; // Stack is empty if SP is 0
                     end
               end
         end
   end

endmodule
```

Fig 5.11: Stack Verilog code

5.12. Multiplexers

Multiplexers are simple asynchronous logic devices. They were custom made in Verilog as this improved readability over using built-in megafunctions. Their initial design was a block diagram that contained some smaller multiplexers chained together. This was later changed to a more elegant Verilog implementation since it provided a more efficient circuit. The multiplexers simply select which input to pass to their output based on their select input line(s) (see Appendix J).

6. Final analysis of design

The following section describes some test results, as well as an analysis on power consumption and clock frequency of the CPU. Many tests were performed on the two versions (original and pipelined) of the CPU to quantise its performance. The initial maximum frequency tests were done with both the original design as well as the pipelined design to see the performance impact of pipelining. Further tests were only performed on the pipelined version as it performed better than the original design.

6.1. Benchmark tests

Before performing the three benchmark tests outlined in the project specification, every command was tested individually in a custom-made program. The three tests were then translated into code executable by the CPU and tested using the pipelined version of the CPU. The waveform simulation results were recorded and can be seen in Appendix L. Multiple scenarios were considered for each test so that the CPU's functionality was thoroughly verified. The real time taken for each of the tests to complete was calculated using the value of 9.091 ns as the true clock period. This value was obtained from running timing simulations (see subsection 6.2. 'Maximum clock frequency tests' for more details). The geometric mean time was calculated using $T_G = (T_1 T_2 T_3)^{1/3}$ where T_1, T_2, and T_3 represent the time taken for each test to complete. Fig 6.1 shows a summary of these results.

Test	Number of Cycles	Real Time Taken	Geometric Mean Time
Fibonacci (n=5)	221	2.0091 µs	0.685 µs
LCG	44 (average)	0.400 µs (average)	
Linked List (10 elements)	44 (average)	0.400 µs (average)	

Fig. 6.1.a: Summary of the results of the three benchmark tests

6.2. Maximum clock frequency tests

```
create_clock -name {CLK} -period 9.1 [get_ports {CLK}]
```

The line above shows the Synopsys Design Constraints file used to set up timing tests, setting the clock input to a target period of 9.1ns giving a frequency of approximately 110MHz.

The three different models shown in Fig. 6.2 have different delays and gate transition speeds which affect the total propagation delay of signals within the CPU. A longer delay means that a block has longer setup and hold times to produce outputs, for given inputs. A longer delay results in a lower maximum clock frequency. The limiting factor of the maximum clock frequency was

usually due to the data delay between the instruction RAM and the stack. Initially, this seemed to be a false path as there is no instruction for which data flows directly to the stack.

Further analysis using the TimeQuest tool (shown in Appendix K) revealed that the limiting path was the control path for the stack block which travels from the instruction RAM through the decoder (which provides control signals for the stack) to the stack itself. This results in a delay of between 9 and 9.5ns on the Slow 85C model and around 5ns on the Fast 0C model.

	Maximum Frequency		
Model	Original	Pipelined	Pipelined -RRC
Slow 1.2V 85C	102.57 MHz	106.08 MHz	109.51 MHz
Slow 1.2V 0C	112.79 MHz	117.44 MHz	121.29 MHz
Fast 1.2V 0C	184.84 MHz	193.57 MHz	199.20 MHz

Fig. 6.2: Results of the maximum clock frequency with different test cases.
Note: -RRC indicates the removal of the RRC instruction from the ALU. Original stands for the CPU design before pipelining.

The timing tests were first done using the original design, with each instruction taking between two and three cycles, averaging ~0.4 instructions per cycle (IPC). On the pipelined version, each instruction takes between one and two cycles, averaging an IPC of ~0.86. Along with increased maximum clock frequency, pipelining caused an increase of ~125% in instructions executed per cycle over the original design.

The third column shows the tests repeated with the pipelined version but with the RRC instruction removed from the ALU, which further improved the clock frequency.

6.3. FPGA area and utilisation tests

When each version of the CPU was compiled fully, the Fitter report showed how much of the selected FPGA was utilised as well as a breakdown of the resources that each instance within the design required. The breakdowns listed the number of logic cells used within each instance as well as the number of registers and the amount of memory in bits.

	Resource Usage			
Version	Logic Cells	ALU Logic Cells	Logic Registers	Memory (Bits)
Original	3186	1868	667	73728
Pipelined	3212	1868	666	73728
Pipelined -RRC	2786	1438	666	73728

Fig. 6.3.a: Resource usage within the FPGA as determined by the Fitter

The most resource-intensive blocks were the 'LIFOstack' and the ALU. Within the ALU there was an instance named "Mod0" which contributed to almost 1/6th of the logic cells used by the ALU (as shown in Fig 6.3.b). This was not an instance we created. The 'lpm_divide' block was automatically added to the symbol due to the use of the modulus function ("%" symbol) used for the RRC instruction. After commenting out the RRC case, logic cell utilisation of the ALU fell by approximately 15%, while also increasing the maximum clock frequency. We decided to continue with this instruction removed as it has very similar functionality to the ROR instruction and is also difficult to use since the carry bit would have to be set by the previous instruction. However, our instruction definitions do not include information on how to use a carry bit or whether to ignore it, compared to the ARM architecture. The difference in utilisation is fully shown in Appendix M.

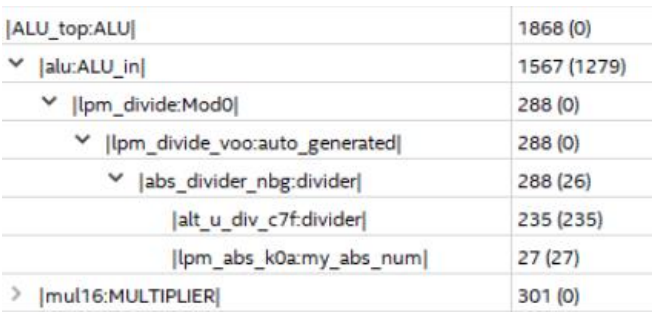


Fig. 6.3.b: Resource utilisation breakdown of the ALU

6.4. Power analysis tests

All the power analysis tests were done with the target clock period set to 9.091ns (clock frequency set to ~110 MHz) as this is close to the maximum for the original design using the Slow 1200mV 0C Model.

	Dynamic Power	Static Power	Confidence
Original	58.87 mW	43.53 mW	Low: insufficient toggle rate data
Pipelined	35.00 mW	43.46 mW	Low: insufficient toggle rate data
Pipelined -RRC	33.14 mW	43.40 mW	Low: insufficient toggle rate data

Fig. 6.4.a: Thermal Power Dissipation of the three versions of the CPU

The confidence level remained low during all the tests as we did not have a Value Change Dump (VCD) file which provides the Power Analyser with information about how often the nodes within the device change state. Without this information, the Power Analyser guesses the states either from a previous simulation or using a conservative estimate that the nodes change state at around 12.5% of the frequency of the input clock signal. The static power remained similar through all three tests since the Fitter picked the same device and the design had roughly the same number of static devices such as RAM units and registers. Pipelining had a significant impact on the dynamic power consumption of the circuit, causing a reduction of ~41%. This is primarily due to reduced signal switching between different states. The FETCH state that existed in the original design sets many control lines to low during the FETCH phase and then back to high during EXEC1. However, in the pipelined version the FETCH cycle is removed. In this version, the lines

can remain high, significantly reducing the amount of switching between cycles and thus, decreasing dynamic power consumption.

7. Reflection

The following section summarises our team's evaluation of the success of the final design, as well as providing some personal thoughts on the project progression and outcome.

7.1 Project success

Before starting this project, a list of functional and non-functional requirements has been made (see subsection to 2.2 'Project specification'). Most of those requirements have been met, but some required more work than expected.

Implementing a multiplier circuit proved to be a challenging task, which required researching many different implementations. The biggest drawback of our multiplier is the fact that it is synchronous. Therefore, it takes two cycles to calculate the product. Initially, the aim was to use one of the many asynchronous memory blocks available in Quartus. This would make the multiplier block asynchronous and more efficient. However, asynchronous memory blocks are not available within Cyclone IV.

Pipelining turned out to be a handy add-on to the final design. Not only did pipelining reduce the number of cycles needed to complete a particular instruction, but it also reduced the overall power consumption and increased the maximum clock frequency. It is important to notice that without pipelining the last non-functional requirement would not have been met. A non-pipelined circuit was consuming just over 50mW of dynamic power, but a pipelined circuit stayed well below the threshold of 50mW (for more information see section 6. 'Final analysis of design).

Using the Harvard architecture with separate instruction and data memory proved to be the right choice. Due to the nature of the architecture, the occurrence of self-modifying code is virtually impossible. Furthermore, using the Harvard architecture prevents the pipeline stall during a store instruction as seen in MU0. Finally, we believed that using this architecture would reduce power consumption.

The CPU successfully executed the three required tests, as set out in the specification (see subsection 6.1 'Benchmark tests'). There were no specialised instructions and the source codes could be implemented directly into the CPU just by using basic instructions such as loading, jumping and arithmetic. As expected, the Fibonacci test took the longest time to complete. This is not due to the CPU architecture, but due to the nature of the recursive algorithm. Even though

there are many different ways to calculate Fibonacci numbers, some much faster than recursion, implementing recursion into the CPU turned out to be a challenging, but worthwhile task. The CPU is capable of performing a wide range of recursive functions, rather than just being limited to calculating the n^{th} Fibonacci number.

7.2 Future work

Even though the final CPU is capable of performing a wide range of operations, there are a few features that might be considered in the future. The instruction set is quite versatile and with 64 opcodes available, more instructions could be added.

An instruction that comes to mind is an immediate load. This was not implemented initially because the instruction word length is not long enough to support an 11-bit data memory location with the chosen template for instructions in the ISA.

Secondly, indirect memory addressing might be improved, to take into account register offsets. ARMv8 supports many different types of indirect addressing [1], including those with offsets. While this was not a requirement of the project, such an instruction might be useful when the CPU is considered as a part of a larger system.

Finally, many algorithms require floating point numbers and division. With these two features added and the current arithmetic capabilities, the CPU would be able to deal with many advanced mathematical concepts such as Maclaurin series, trigonometric functions and exponentiation.

7.3 Final thoughts

This project was quite challenging but also rewarding. We learned quite a lot about modern CPU architectures and implementations of certain digital circuits, such as ALUs, multipliers and memory buffers. Furthermore, we obtained key skills around project management, planning and writing documentation. The project went mostly according to plan with few minor issues along the way and was submitted on time.

8. Bibliography

[1] ARM, "ARM Information Center".
Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204j/Cjafgdih.html [Accessed: May 18, 2020].

[2] E. De Vries, "Introduction to the MIPS Processor", 2016.
Available: https://www.scss.tcd.ie/jeremy.jones/vivio/dlx/dlxtutorial.htm [Accessed: May 17, 2020].

[3] Arm Limited, "Assembler User Guide: ARM registers".
Available: http://www.keil.com/support/man/docs/armasm/armasm_dom1359731128950.htm [Accessed: May 18, 2020].

[4] Massachusetts Institute of Technology, "Arithmetic Circuits & Multipliers", 2016. Available: http://web.mit.edu/6.111/www/f2016/handouts/L08.pdf [Accessed: May 23, 2020].

[5] M. Moeng and J. Wei, "Optimising Multipliers for the CPU: A ROM based approach", 2007.
Available: https://people.eecs.berkeley.edu/~kubitron/courses/cs252-S07/projects/reports/project6_report_ver2.pdf [Accessed: May 25, 2020].

[6] C. S. Wallace, "A Suggestion for a Fast Multiplier", *IEEE Transactions on Electronic Computers,* vol. EC-13, *(1),* pp. 14-17, 1964. DOI: 10.1109/PGEC.1964.263830 [Accessed: May 23, 2020].

[7] Wikipedia contributors, "Ancient Egyptian multiplication", 2020.
Available: https://en.wikipedia.org/w/index.php?title=Ancient_Egyptian_multiplication&oldid=957721403 [Accessed: May 23, 2020].

[8] H. Ling, "An approach to implementing multiplication with small tables", *IEEE Transactions on Computers,* vol. 39, *(5),* pp. 717-718, 1990. DOI: 10.1109/12.53588 [Accessed: May 25, 2020].

[9] B. Vinnakota, "Implementing multiplication with split read-only memory", *IEEE Transactions on Computers,* vol. 44, *(11),* pp. 1352-1356, 1995. DOI: 10.1109/12.475134 [Accessed: May 25, 2020].

[10] Anonymous, "Verilog for Beginners: Last-In-First-Out Buffer". Available: https://esrd2014.blogspot.com/p/last-in-first-out-buffer.html [Accessed: Jun 04, 2020].

[11] AVR Microcontrollers, "AVR Instruction Set Manual," 2016. Available: http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf [Accessed: May 20, 2020].

[12] P. Johnson, "LIFO", 2009. Available: https://www.beyond-circuits.com/wordpress/2009/10/lifo/ [Accessed: Jun 04, 2020].

9. Appendix

A. C++ test codes

A.1. Calculate Fibonacci numbers using recursion

```
int fib(const int n){
     int y;
     if (n <= 1) y = 1;
     else {
          y = fib(n-1)
          y = y + fib(n-2);
     return y;
}
```

```
Comments:
  R1 holds n in the current function call; used to communicate between calls (input)
  R2 holds y of the last call; used to communicate between calls (output)
  R3 holds y of the current function call
  R4 holds the number 0x2, the value to subtract for the fib(n-2) call
  R5 holds the number 0x7, the address of the beginning of the fib function
  R6 holds the number 0xD, the address of the beginning of the 'else' part of the fib function
  R7 holds the number 0x1 for the comparison 'n<=1'

(0x0)  JMA 2                                          (skips the STP command)
(0x1)  STP                                            (program end)
(0x2)  LDA R1 ##MEMORY_LOCATION_OF_n##                (load master n into R1)
(0x3)  LDA R7 ##MEMORY_LOCATION_CONTAINING_0x1##      (load 0x1, number needed for n<=1 comparison, to R7)
(0x4)  LDA R6 ##MEMORY_LOCATION_CONTAINING_0xD##      (load 0xD, memory location of the 'else' part of the 'fib' function, to R6)
(0x5)  LDA R5 ##MEMORY_LOCATION_CONTAINING_0x7##      (load 0x7, memory location of the beginning of the 'fib' function, to R5)
(0x6)  LDA R4 ##MEMORY_LOCATION_CONTAINING_0x2##      (load 0x2, number needed for fib(n-2) call, to R4)
(0x7)  JC2 R6 R1 R7                                   (jump to 'else' part if n>1)
(0x8)  MOV R2 R7                                      (make y = 1)
(0x9)  LDA R7 ##MEMORY_LOCATION_OF_n##                (load master n to R7)
(0xA)  JC3 R2 R1 R7                                   (if current n = master n, jump to stop)
(0xB)  MOV R7 R2                                      (set R7 to 0x1 again)
(0xC)  RTN                                            (return from call)
(0xD)  PSH R1                                         (beginning of 'else' part; push current n to stack)
(0xE)  SBO R1 R1                                      (decrease n by 1)
(0xF)  CLL R5                                         (call fib(n-1), i.e. jump to 0x7 and save the current head)
(0x10) POP R1                                         (retrieve current n from stack)
(0x11) MOV R3 R2                                      (make output y = current y, i.e. y = fib(n-1))
(0x12) PSH R3                                         (push current y onto stack)
(0x13) PSH R1                                         (push current n onto stack)
(0x14) SUB R1 R1 R4                                   (decrease n by 2)
(0x15) CLL R5                                         (call fib(n-2), i.e. jump to 0x7 and save the current head)
(0x16) POP R1                                         (retrieve current n)
(0x17) POP R3                                         (retrieve current y)
(0x18) ADD R2 R2 R3                                   (make output y = output y + current y, i.e. y = y + fib(n-2))
(0x19) LDA R3 ##MEMORY_LOCATION_OF_n##                (load master n to R3)
(0x1A) JC3 R7 R1 R3                                   (if current n = master n, jump to stop)
(0x1B) RTN                                            (if master n hasn't been reached, return from call)


Example that can be used with the instruction generator program:
JMA 2
STP
LDA R1 0
LDA R7 1
LDA R6 2
LDA R5 3
LDA R4 4
JC2 R6 R1 R7
MOV R2 R7
LDA R7 0
JC3 R2 R1 R7
MOV R7 R2
RTN
PSH R1
SBO R1 R1
CLL R5
POP R1
MOV R3 R2
PSH R3
PSH R1
SUB R1 R1 R4
CLL R5
POP R1
POP R3
ADD R2 R2 R3
LDA R3 0
JC3 R7 R1 R3
RTN

Requires setting the following data memory locations:
Set location '0x0' to value of n.
Set location '0x1' to value of 0x1 (used for n<=1 comparison).
Set location '0x2' to value of 0xD (beginning of 'else' part of 'fib' function).
Set location '0x3' to value of 0x7 (beginning of 'fib' function).
Set location '0x4' to value of 0x2 (used for fib(n-2) call).
```

A.2. Calculate pseudo-random integers with a linear congruential generator (LCG)

```
int lcong(
      const unsigned int a,
      const unsigned int b,
      const int n,
      const unsigned int s)
{
      unsigned int y = s;
      unsigned int sum = 0;
      for (int i = n ; i > 0; i--){
         y = y*a + b // calculate the new pseudo-random number
         sum = sum + y // add it to the total
      }
      return sum;
}
```

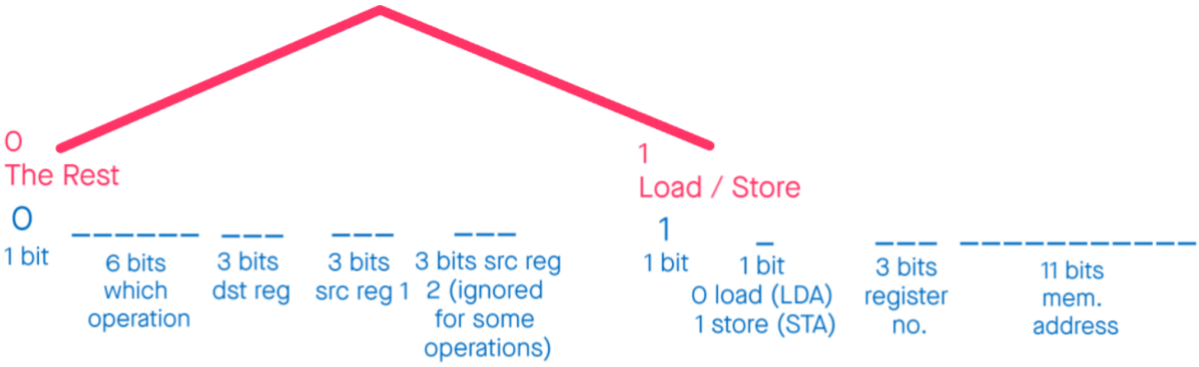
```
Comments:
  R1 holds y, as described in the source code
  R2 holds a, as described in the source code
  R3 holds b, as described in the source code
  R4 holds n, as described in the source code
  R5 holds the output (sum), as described in the source code
  R6 holds the number 0xB (for jumping to the end of the program)
  R7 holds the number 0x6 (for jumping back)

(0x0) LDA R1 ##MEMORY_LOCATION_OF_s##                (loads the seed, s, into R1, which then becomes y as in the source code)
(0x1) LDA R2 ##MEMORY_LOCATION_OF_a##                (loads a into R2)
(0x2) LDA R3 ##MEMORY_LOCATION_OF_b##                (loads b into R3)
(0x3) LDA R4 ##MEMORY_LOCATION_OF_n##                (loads n into R4)
(0x4) LDA R6 ##MEMORY_LOCATION_CONTAINING_0xA##      (loads 0xB, memory location of program end to R6)
(0x5) LDA R7 ##MEMORY_LOCATION_CONTAINING_0x6##      (loads 0x6, memory location for jumping back)
(0x6) JC4 R6 R4                                      (if R4, in this case, n, is equal to 0, then jump to end, STP)
(0x7) MLA R1 R2 R3                                   (multiply and add, same as in the source code y=y*a+b)
(0x8) ADD R5 R5 R1                                   (adds y to the sum, so that sum+=y, as in the source code)
(0x9) SBO R4 R4                                      (decrease R4, i.e. n by one, so no infinite loops occur)
(0xA) JMP R7                                         (jumps back to check whether n>0 and if so, repeats again)
(0xB) STP


Example that can be used with the instruction generator program:
LDA R1 0
LDA R2 1
LDA R3 2
LDA R4 3
LDA R6 4
LDA R7 5
JC4 R6 R4
MLA R1 R2 R3
ADD R5 R5 R1
SBO R4 R4
JMP R7 R6
STP

Requires setting the following data memory location:
Set location '0' to value of s;
Set location '1' to value of a;
Set location '2' to value of b;
Set location '3' to value of n;
Set location '4' to value of 0xB;
Set location '5' to value of 0x6;
```

A.3. Traverse a linked list to find an item

```
typedef struct item{
     int value;
     struct item *next;
} item_t;

item_t* find(const int x, item_t* head){
     while (head->value != x){
           head = head->next;
           if (head == NULL) break;
     return head;
}
```

```
Comments:
  R1 holds x, value to be found
  R2 holds head, the memory location of the first element
  R3 holds the number END = 16b'0000100000000000'=0x800 (for comparing, this is the end of the linked list)
  R4 holds the current element in the traversed list
  R5 holds the number 0xB (for jumping to the end)
  R6 holds the number 0x5 (for jumping back)
  A linked list pair is stored as men[N]=value and mem[N+1] = address of next element

(0x0) LDA R1 ##MEMORY_LOCATION_OF_x##
    (loads the value to be found, x, into R1)
(0x1) LDA R2 ##MEMORY_LOCATION_OF_head##
    (loads the address of the first element into R2)
(0x2) LDA R3 ##MEMORY_LOCATION_OF_END##
    (loads a number (0x800) to R3 which is our definition of an end of a linked list)
(0x3) LDA R5 ####MEMORY_LOCATION_CONTAINING_0xB##
    (loads a number (0xB) to R5 which is used for jumping to STP, when the element is found or no elements left)
(0x4) LDA R6 ####MEMORY_LOCATION_CONTAINING_0x5##
    (loads a number (0x5) to R6 which is used for jumping to back, if current element is not the required element and there are still elements left)
(0x5) LDR R4 R2
    (loads R4 using data from the memory address which is the value in R4)
(0x6) JC3 R5 R4 R1
    (jumps to the end ('STP') if R4 (current element) is equal to R1 (required element))
(0x7) ADO R2
    (adds one to R2, which when completed contains the address of the address of next element in the linked list)
(0x8) LDR R2 R2
    (loads the address of the next element into R2, thus updating the current head)
(0x9) JC3 R5 R2 R3
    (jumps to the end ('STP') if R2 (current head) is equal to R3 (end of list)
(0xA) JMP R6
    (jumps to value stored in R4 () which repeats this process)
(0xB) STP
    (program end)


Example that can be used with the instruction generator program:
LDA R1 0
LDA R2 1
LDA R3 2
LDA R5 3
LDA R6 4
LDR R4 R2
JC3 R5 R4 R1
ADO R2
LDR R2 R2
JC3 R5 R2 R3
JMP R6
STP

Requires setting the following data memory locations:
Set location '0' to value of x;
Set location '1' to value of head;
Set location '2' to value of 0x800 (end of linked list)
Set location '3' to value of 0xB (end of program, STP);
Set location '4' to value of 0x5; (going back)
Create a linked list that starts at 'head' memory location
```

B. Freehand ISA map



00 JMP 0000 00- Unconditional Jump
01 JMA 0000 01 - Unconditional Jump to address

For all jumps → A = Rs1, B = Rs2, Rd is the destination of jump

04 JC1 0001 00 - Conditional Jump (A < B)
05 JC2 0001 01 - Conditional Jump (A > B)
06 JC3 0001 10 - Conditional Jump (A = B)
07 JC4 0001 11 - Conditional Jump (A = 0)

08 JC5 0010 00 - Conditional Jump (A >= B / A !< B)
09 JC6 0010 01 - Conditional Jump (A <= B / A !> B)
0A JC7 0010 10 - Conditional Jump (A != B)
0B JC8 0010 11 - Conditional Jump (A < 0)

0C AND 0011 00 - Bitwise AND (Rd = Rs1 && Rs2)
0D OR 0011 01 - Bitwise OR (Rd = Rs1 || Rs2)
0E XOR 0011 10 - Bitwise XOR (Rd = Rs1 XOR Rs2)
0F NOT 0011 11 - Bitwise NOT (Rd = !Rs1)

10 NND 0100 00 - Bitwise NAND (Rd = !(Rs1 && Rs2))
11 NOR 0100 01 - Bitwise NOR (Rd = !(Rs1 || Rs2))
12 XNR 0100 10 - Bitwise XNOR (Rd = !(Rs1 XOR Rs2))
13 MOV 0100 11 - Move (Rd = Rs1)

14 ADD 0101 00 - Add (Rd = Rs1 + Rs2)
15 ADC 0101 01 - Add w/ Carry (Rd = Rs1 + Rs2 + C)
16 ADO 0101 10 - Add 1 (Rd = Rs1 + 1)
0101 11 -

18 SUB 0110 00 - Subtract (Rd = Rs1 - Rs2)
19 SBC 0110 01 - Subtract w/ Carry (Rd = Rs1 - Rs2 + C - 1)
1A SBO 0110 10 - Subtract 1 (Rd = Rs1 - 1)
0110 11 -

1C MUL 0111 00 - Multiply (Rd = Rs1 * Rs2)
1D MLA 0111 01 - Multiply and Add (Rd = [Rd*Rs1] + Rs2)
1E MLS 0111 10 - Multiply and Subtract (Rd = Rs2 - [Rd*Rs1])
1F MRT 0111 11 - Retrieve Multiply MSBs (Rd = MSBs)

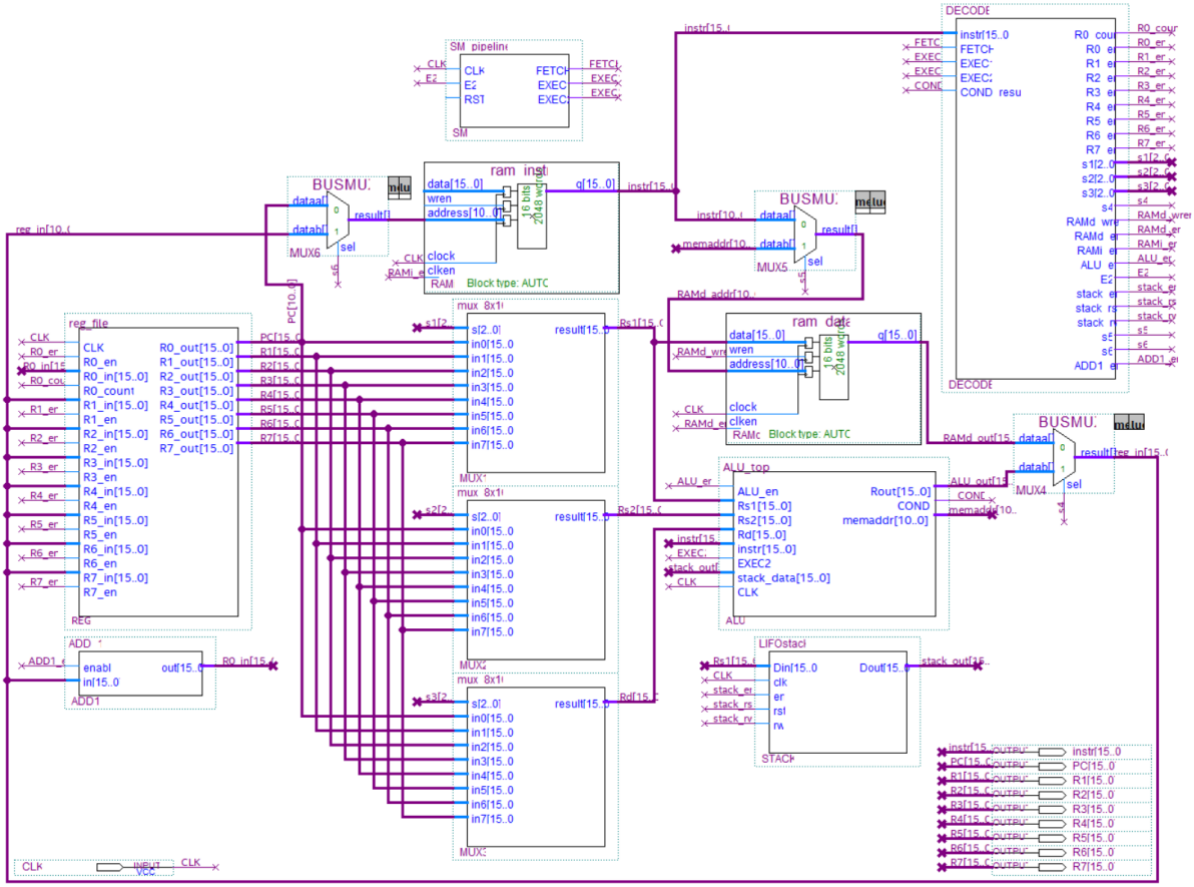
20 LSL 1000 00 - Logical Shift Left (Rd = Rs1 shifted by value of Rs2)
21 LSR 1000 01 - Logical Shift Right (Rd = Rs1 shifted by Rs2)
22 ASR 1000 10 - Arithmetic Shift Right (Rd = Rs1 shifted by Rs2, maintaining sign bit)
1000 11 -

24 ROR 1001 00 - Shift Right Loop (Rd = Rs1 shifted by Rs2, but Rs1[0]→Rs1[15])
25 RRC 1001 01 - Shift Right Loop w/ Carry (Above but Rs1[0]→Carry & Carry→Rs1[15])
26 CLL 1001 10 - Call (jumps to a given instruction while saving the current index to stack)
27 RTN 1001 11 - Return (retrieves a value from stack, loads into PC and fetches the instruction)

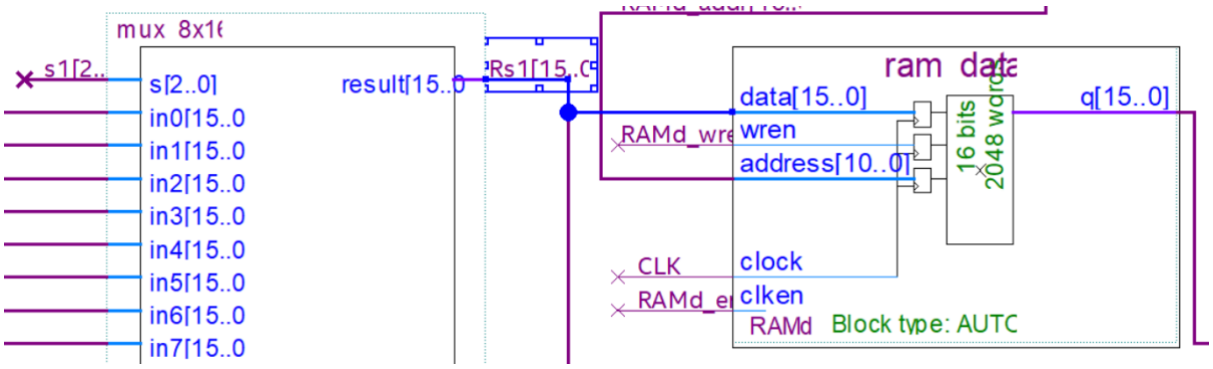
28 PSH 1010 00 - Push onto stack (Stack = Rs1)
29 POP 1010 01 - Pop from stack (Rd = Stack)
2A LDR 1010 10 - Indirect Load (Rd = Mem[Rs1])
2B STR 1010 11 - Indirect Store (Mem[Rd] = Rs1)

3E NOP 1111 10 - No Operation (Do nothing for a cycle)
3F STP 1111 11 - Stop (Program Ends)

C. Complete Block Diagram File



Register Rs1 also feeds into RAMd so the STORE instruction can set RS1 and save the value without enabling the ALU, shown closer below:



D. ASR, ROR and RRC explained

```
6'b100010: alusum = {Rs1[15], Rs1 >>> Rs2};
```

Arithmetic Shift Right uses the >>> bitwise operator, which shifts "RS1" right by the value of "RS2" while shifting in the old MSB. Then appending the original MSB to the front to create a 17-bit number to fit in the 'alusum' register.

```
6'b100100: alusum = {1'b0, (Rs1 >> Rs2[3:0]) | (Rs1 << (16 - Rs2[3:0]))};
```

ROR uses both logical shift operators along with an OR to simplify a rotational shift. As RS1 is 16 bits long, every 16 shifts the result is the same as no shift and so only the last 4 bits of RS2 are needed to determine the result. RS1 is shifted right by the value and left by 16, minus the value to represent the LSB is shifted into the MSB with each shift. The two values are then OR'd together.

```
6'b100101: alusum = ({Rs1, carry} >> (Rs2 % 17)) | ({Rs1, carry} << (17 - (Rs2 % 17)));
```

RRC is like ROR, but the LSB is shifted into the carry slot, and the previous carry value is shifted into the MSB. As the size of the value being rotated is 17 bits rather than 16, the remainder of RS2 divided by 17 is needed. (RS2 mod 17) This results in a large lpm_divide block being added to the ALU.

E. CLL and RTN explained

The CLL and RTN instructions allow for an operation similar to a C++ function call and return:

- CLL Rd – saves the value of the PC to the top of the stack and loads the value of Rd into the PC, effectively jumping to the instruction at the memory address specified in Rd.
- RTN – retrieves a value from the stack then (in EXEC2) saves this value into the PC (also reads the instruction at that location in the pipelined version).

Due to the usage of one single stack by both the user and these instructions, the use of these instructions needs to be given special care. A recommended call and return can be structured in the following way:

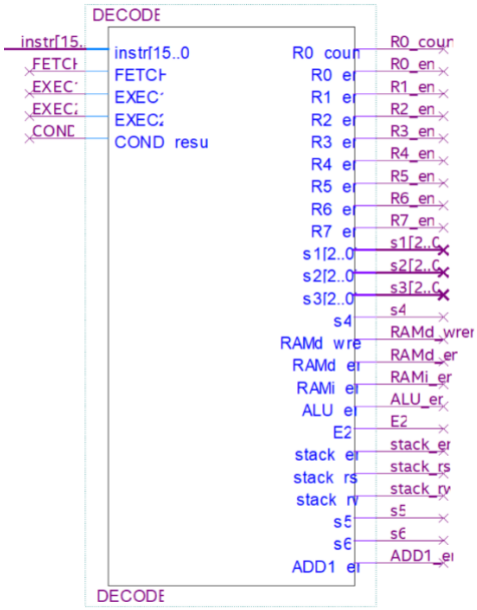
```
    ...
0.  PSH R2
1.  PSH R1      save the variables that need to be kept track of
2.  CLL R7      make a call (R7 contains the value 0x7)
3.  POP R1      retrieve the variables saved before the call was made
4.  POP R2
    ...
7.  Some operation
8.  RTN
    ...
```

F. Decoder block

The decoder is an asynchronous block which determines the current instruction from the opcode, setting the corresponding wire (lines 44-59) high and the rest low. Next, the control lines are set (lines 61-90) depending on which controls are needed for the current instruction as well as the multiplexers for selecting registers Rd, Rs1 & Rs2.

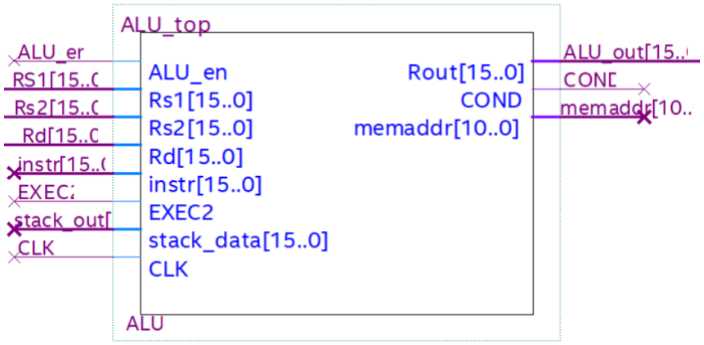
Decoder.v and the symbol created from the Verilog:

```
1  module DECODE
2  (
3      input [15:0] instr,
4      input FETCH,
5      input EXEC1,
6      input EXEC2,
7      input COND_result,
8      output R0_count,
9      output R0_en,
10     output R1_en,
11     output R2_en,
12     output R3_en,
13     output R4_en,
14     output R5_en,
15     output R6_en,
16     output R7_en,
17     output [2:0] s1,
18     output [2:0] s2,
19     output [2:0] s3,
20     output s4,
21     output RAMd_wren,
22     output RAMd_en,
23     output RAMi_en,
24     output ALU_en,
25     output E2,
26     output stack_en,
27     output stack_rst,
28     output stack_rw,
29     output s5,
30     output s6,
31     output ADD1_en
32 );
33
34     wire msb = instr[15];              //MSB of the instruction word
35     wire ls = instr[14];               //LDA or STA bit
36     wire [2:0] Rls = instr[13:11];     //Register in the LDA/STA operation
37     wire [10:0] addr = instr[10:0];    //Memory address in the LDA/STA operation
38     wire [5:0] op = instr[14:9];       //Opcode in regular instructions
39     wire [2:0] Rd = instr[8:6];        //Destination register in command
40     wire [2:0] Rs1 = instr[5:3];       //Source register 1 in command
41     wire [2:0] Rs2 = instr[2:0];       //Source register 2 in command
42
43     //Different opcodes (refer to documentation):
44     wire LDA =  msb & ~ls;
45     wire STA =  msb &  ls;
46     wire JMP = ~msb & ~op[5] & ~op[4] & ~op[3] & ~op[2] & ~op[1] & ~op[0];
47     wire JMA = ~msb & ~op[5] & ~op[4] & ~op[3] & ~op[2] & ~op[1] &  op[0];
48     wire JCX = ~msb & ((~op[5] & ~op[4] & ~op[3] & op[2]) | (~op[5] & ~op[4] & op[3] & ~op[2]));
49     wire MUL = ~msb & ~op[5] &  op[4] &  op[3] &  op[2] & ~op[1] & ~op[0];
50     wire MLA = ~msb & ~op[5] &  op[4] &  op[3] &  op[2] & ~op[1] &  op[0];
51     wire MLS = ~msb & ~op[5] &  op[4] &  op[3] &  op[2] &  op[1] & ~op[0];
52     wire PSH = ~msb &  op[5] & ~op[4] &  op[3] & ~op[2] & ~op[1] & ~op[0];
53     wire POP = ~msb &  op[5] & ~op[4] &  op[3] & ~op[2] & ~op[1] &  op[0];
54     wire LDR = ~msb &  op[5] & ~op[4] &  op[3] & ~op[2] &  op[1] & ~op[0];
55     wire STR = ~msb &  op[5] & ~op[4] &  op[3] & ~op[2] &  op[1] &  op[0];
56     wire CLL = ~msb &  op[5] & ~op[4] & ~op[3] &  op[2] &  op[1] & ~op[0];
57     wire RTN = ~msb &  op[5] & ~op[4] & ~op[3] &  op[2] &  op[1] &  op[0];
58     wire NOP = ~msb &  op[5] &  op[4] &  op[3] &  op[2] &  op[1] & ~op[0];
59     wire STP = ~msb &  op[5] &  op[4] &  op[3] &  op[2] &  op[1] &  op[0];
60
61     assign R0_count = (FETCH & ~STP) | (EXEC1 & ~(JMP | JMA | (JCX & COND_result) | STP | LDR | LDA | MUL | MLA | MLS | POP | RTN | CLL)) | (EXEC2 & (
       LDR | LDA | MUL | MLA | MLS | POP));
62     assign R0_en = (EXEC1 & (~(STA | NOP | STP | LDA | PSH | LDR | CLL | RTN) & ~Rd[2] & ~Rd[1] & ~Rd[0] | JMP | (JCX & COND_result) | JMA)) | (EXEC2
       & LDA & ~Rls[2] & ~Rls[1] & ~Rls[0]) | (EXEC2 & (MUL | MLA | MLS | POP | STR | LDR) & ~Rd[2] & ~Rd[1] & ~Rd[0]) | (EXEC2 & RTN) | (EXEC1 & CLL);
63     assign R1_en = (EXEC1 & ~(JMP | JMA | JCX | STA | LDA | MUL | MLA | MLS | NOP | STP | POP | PSH | LDR | CLL | RTN) & ~Rd[2] & ~Rd[1] &  Rd[0]) | (
       EXEC2 & LDA & ~Rls[2] & ~Rls[1] &  Rls[0]) | (EXEC2 & (MUL | MLA | MLS | POP | LDR) & ~Rd[2] & ~Rd[1] &  Rd[0]);
64     assign R2_en = (EXEC1 & ~(JMP | JMA | JCX | STA | LDA | MUL | MLA | MLS | NOP | STP | POP | PSH | LDR | CLL | RTN) & ~Rd[2] &  Rd[1] & ~Rd[0]) | (
       EXEC2 & LDA & ~Rls[2] &  Rls[1] & ~Rls[0]) | (EXEC2 & (MUL | MLA | MLS | POP | LDR) & ~Rd[2] &  Rd[1] & ~Rd[0]);
65     assign R3_en = (EXEC1 & ~(JMP | JMA | JCX | STA | LDA | MUL | MLA | MLS | NOP | STP | POP | PSH | LDR | CLL | RTN) & ~Rd[2] &  Rd[1] &  Rd[0]) | (
       EXEC2 & LDA & ~Rls[2] &  Rls[1] &  Rls[0]) | (EXEC2 & (MUL | MLA | MLS | POP | LDR) & ~Rd[2] &  Rd[1] &  Rd[0]);
66     assign R4_en = (EXEC1 & ~(JMP | JMA | JCX | STA | LDA | MUL | MLA | MLS | NOP | STP | POP | PSH | LDR | CLL | RTN) &  Rd[2] & ~Rd[1] & ~Rd[0]) | (
       EXEC2 & LDA &  Rls[2] & ~Rls[1] & ~Rls[0]) | (EXEC2 & (MUL | MLA | MLS | POP | LDR) &  Rd[2] & ~Rd[1] & ~Rd[0]);
67     assign R5_en = (EXEC1 & ~(JMP | JMA | JCX | STA | LDA | MUL | MLA | MLS | NOP | STP | POP | PSH | LDR | CLL | RTN) &  Rd[2] & ~Rd[1] &  Rd[0]) | (
       EXEC2 & LDA &  Rls[2] & ~Rls[1] &  Rls[0]) | (EXEC2 & (MUL | MLA | MLS | POP | LDR) &  Rd[2] & ~Rd[1] &  Rd[0]);
68     assign R6_en = (EXEC1 & ~(JMP | JMA | JCX | STA | LDA | MUL | MLA | MLS | NOP | STP | POP | PSH | LDR | CLL | RTN) &  Rd[2] &  Rd[1] & ~Rd[0]) | (
       EXEC2 & LDA &  Rls[2] &  Rls[1] & ~Rls[0]) | (EXEC2 & (MUL | MLA | MLS | POP | LDR) &  Rd[2] &  Rd[1] & ~Rd[0]);
69     assign R7_en = (EXEC1 & ~(JMP | JMA | JCX | STA | LDA | MUL | MLA | MLS | NOP | STP | POP | PSH | LDR | CLL | RTN) &  Rd[2] &  Rd[1] &  Rd[0]) | (
       EXEC2 & LDA &  Rls[2] &  Rls[1] &  Rls[0]) | (EXEC2 & (MUL | MLA | MLS | POP | LDR) &  Rd[2] &  Rd[1] &  Rd[0]);
70     assign s1[2] = (~(JMP | JMA | STA | LDA | NOP | STP | POP | CLL | RTN) & Rs1[2]) | (STA & Rls[2]);
71     assign s1[1] = (~(JMP | JMA | STA | LDA | NOP | STP | POP | CLL | RTN) & Rs1[1]) | (STA & Rls[1]);
72     assign s1[0] = (~(JMP | JMA | STA | LDA | NOP | STP | POP | CLL | RTN) & Rs1[0]) | (STA & Rls[0]);
73     assign s2[2] = (~(JMP | JMA | STA | LDA | NOP | STP | POP | PSH | LDR | STR | CLL | RTN) & Rs2[2]);
74     assign s2[1] = (~(JMP | JMA | STA | LDA | NOP | STP | POP | PSH | LDR | STR | CLL | RTN) & Rs2[1]);
75     assign s2[0] = (~(JMP | JMA | STA | LDA | NOP | STP | POP | PSH | LDR | STR | CLL | RTN) & Rs2[0]);
76     assign s3[2] = (~(STA | LDA | NOP | STP | PSH | POP | RTN) & Rd[2]);
77     assign s3[1] = (~(STA | LDA | NOP | STP | PSH | POP | RTN) & Rd[1]);
78     assign s3[0] = (~(STA | LDA | NOP | STP | PSH | POP | RTN) & Rd[0]);
79     assign s4 = ~(LDA | LDR);
80     assign RAMd_wren = EXEC1 & (STA | STR);
81     assign RAMd_en = EXEC1 & (STA | LDA | STR | LDR);
82     assign RAMi_en = (FETCH & ~STP) | (EXEC1 & ~(LDA | LDR | MUL | MLA | MLS | POP | STP | RTN)) | (EXEC2 & (LDA | LDR | MUL | MLA | MLS | POP | RTN));
83     assign ALU_en = LDA | STA;
84     assign E2 = EXEC1 & (LDA | MUL | MLA | MLS | POP | LDR | RTN);
85     assign stack_en = (EXEC1 & (PSH | CLL | RTN | POP));
86     assign stack_rst = STP;
87     assign stack_rw = EXEC1 & (PSH | CLL);
88     assign s5 = EXEC1 & (STR | LDR);
89     assign s6 = (EXEC1 & (JMP | JMA | (JCX & COND_result) | CLL)) | (EXEC2 & RTN);
90     assign ADD1_en = (EXEC1 & (JMP | JMA | (JCX & COND_result) | CLL)) | (EXEC2 & RTN);
91
92 endmodule
```

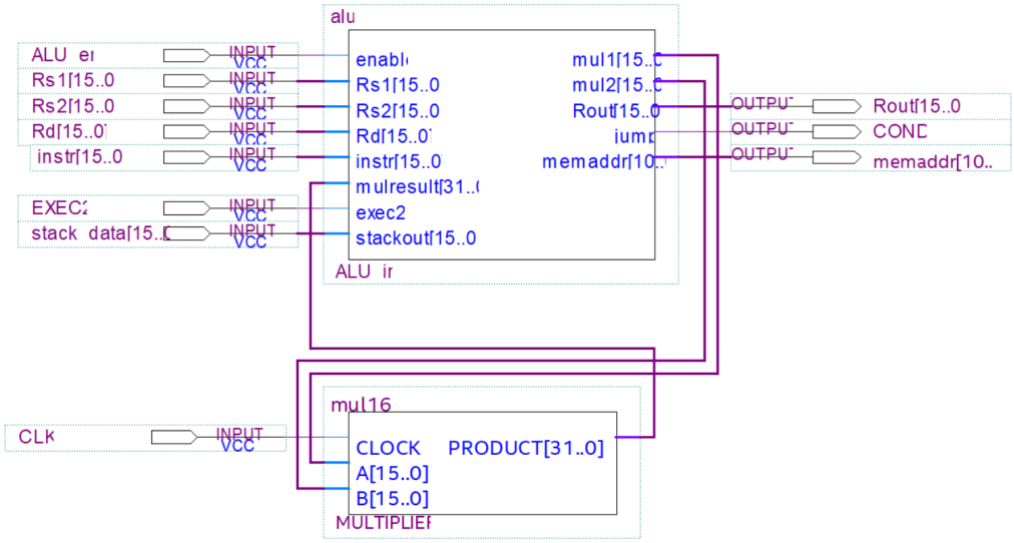


G. Arithmetic Logic Unit block

ALU symbol:



ALU_top.bdf:



alu.v:

```
1   module alu (enable, Rs1, Rs2, Rd, instr, mulresult, exec2, stackout, mul1, mul2, Rout, jump, memaddr);
2
3   input enable; // active LOW, disables the ALU during load/store operations so that undefined behaviour does not occur
4   input signed [15:0] Rs1; // input source register 1
5   input signed [15:0] Rs2; // input source register 2
6   input signed [15:0] Rd; // input destination register
7   input [15:0] instr; // current instruction being executed
8   input signed [31:0] mulresult; // 32-bit result from multiplier
9   input exec2; // Input from state machine to indicate when to take in result from multiplication
10  input [15:0] stackout; // input from stack to be fed back to registers
11
12  output reg signed [15:0] mul1; // first number to be multiplied
13  output reg signed [15:0] mul2; // second number to be multiplied
14  output signed [15:0] Rout; // value to be saved to destination register
15  output jump; // tells decoder whether Jump condition is true
16  reg carry; // Internal carry register that is updated during appropriate opcodes
17  output reg [10:0] memaddr; // address to load data from / store data to RAMd
18
19  wire [5:0]opcode = instr[14:9]; // opcode of current instruction
20  reg signed [16:0] alusum; // extra bit to hold carry from operations other than Multiply
21  assign Rout = alusum [15:0];
22  assign jump = (alusum[16] && ((opcode[5:2] == 4'b0000) | (opcode[5:2] == 4'b0001) | (opcode[5:2] == 4'b0010)));
23  reg [15:0] mulextra;
24
25  //Jump Conditionals:
26  wire JC1, JC2, JC3, JC4, JC5, JC6, JC7, JC8;
27  assign JC1 = (Rs1 < Rs2);
28  assign JC2 = (Rs1 > Rs2);
29  assign JC3 = (Rs1 == Rs2);
30  assign JC4 = (Rs1 == 0);
31  assign JC5 = (Rs1 >= Rs2);
32  assign JC6 = (Rs1 <= Rs2);
33  assign JC7 = (Rs1 != Rs2);
34  assign JC8 = (Rs1 < 0);
35
```

```
always @(opcode, mulresult)
   begin
      if(!enable) begin
         case (opcode)
            6'b000000: alusum = {1'b1, Rd}; // JMP Unconditional Jump, first bit high to indicate jump and passes through Rd
            6'b000001: alusum = {8'b10000000, instr[8:0]};  //JMA unconditional jump to address, MSB high indicates jump, [8:0] is destination

            6'b000100: alusum = {JC1, Rd}; // JC1 Conditional Jump A < B
            6'b000101: alusum = {JC2, Rd}; // JC2 Conditional Jump A > B
            6'b000110: alusum = {JC3, Rd}; // JC3 Conditional Jump A = B
            6'b000111: alusum = {JC4, Rd}; // JC4 Conditional Jump A = 0

            6'b001000: alusum = {JC5, Rd}; // JC5 Conditional Jump A >= B / A !< B
            6'b001001: alusum = {JC6, Rd}; // JC6 Conditional Jump A <= B / A !> B
            6'b001010: alusum = {JC7, Rd}; // JC7 Conditional Jump A != B
            6'b001011: alusum = {JC8, Rd}; // JC8 Conditional Jump A < 0

            6'b001100: alusum = {1'b0, Rs1 & Rs2}; // AND Bitwise AND
            6'b001101: alusum = {1'b0, Rs1 | Rs2}; // OR Bitwise OR
            6'b001110: alusum = {1'b0, Rs1 ^ Rs2}; // XOR Bitwise XOR
            6'b001111: alusum = {1'b0, ~Rs1}; // NOT Bitwise NOT

            6'b010000: alusum = {1'b0, ~Rs1 | ~Rs2}; // NND Bitwise NAND
            6'b010001: alusum = {1'b0, ~Rs1 & ~Rs2}; // NOR Bitwise NOR
            6'b010010: alusum = {1'b0, Rs1 ~^ Rs2}; // XNR Bitwise XNOR
            6'b010011: alusum = {1'b0, Rs1}; // MOV Move (Rd = Rs1)

            6'b010100: begin
                  alusum = {1'b0, Rs1} + {1'b0, Rs2}; // ADD Add (Rd = Rs1 + Rs2)
                  carry = alusum[16];
               end
            6'b010101: begin
                  alusum = {1'b0, Rs1} + {1'b0, Rs2} + carry; // ADC Add w/ Carry (Rd = Rs1 + Rs2 + C)
                  carry = alusum[16];
               end
            6'b010110: begin
                  alusum = {1'b0, Rs1} + {17'b00000000000000001}; // ADO Add 1 (Rd = Rd + 1)
                  carry = alusum[16];
               end
            6'b010111: ;

            6'b011000: begin
                  alusum = {1'b0, Rs1} - {1'b0, Rs2}; // SUB Subtract (Rd = Rs1 - Rs2)
                  carry = alusum[16];
               end
            6'b011001: begin
                  alusum = {1'b0, Rs1} - {1'b0, Rs2} + carry - {17'b00000000000000001}; // SBC Subtract w/ Carry (Rd = Rs1 - Rs2 + C - 1)
                  carry = alusum[16];
               end
            6'b011010: begin
                  alusum = {1'b0, Rs1} - {17'b00000000000000001}; // SBO Subtract 1 (Rd = Rd - 1)
                  carry = alusum[16];
               end
            6'b011011: ;

            6'b011100: begin // MUL Multiply (Rd = Rs1 * Rs2)
                  if(!exec2) begin
                        if(Rs1[15]) begin
                              mul1 = ~Rs1 + {16'h0001};
                           end
                        else begin
                              mul1 = Rs1;
                           end
                        if(Rs2[15]) begin
                              mul2 = ~Rs2 + {16'h0001};
                           end
                        else begin
                              mul2 = Rs2;
                           end
                        alusum = 17'b00000000000000000;
                        carry = (Rs1[15]^Rs2[15]) ? 1'b1 : 1'b0;
                     end
                  else begin
                        {mulextra, alusum[15:0]} = (carry) ? ~mulresult + 32'h00000001 : mulresult;
                     end
               end
            6'b011101: begin // MLA Multiply and Add  (Rd = Rs2 + (Rd * Rs1))
                  if(!exec2) begin
                        if(Rd[15]) begin
                              mul1 = ~Rd + {16'h0001};
                           end
                        else begin
                              mul1 = Rd;
                           end
                        if(Rs1[15]) begin
                              mul2 = ~Rs1 + {16'h0001};
                           end
                        else begin
                              mul2 = Rs1;
                           end
                        alusum = 17'b00000000000000000;
                        carry = (Rs1[15]^Rs2[15]) ? 1'b1 : 1'b0;
                     end
                  else begin
                        {mulextra, alusum[15:0]} = (carry) ? ~mulresult + 32'h00000001 + {16'h0000, Rs2} : mulresult + {16'h0000, Rs2};
                     end
               end
            6'b011110: begin // MLS Multiply and Subtract (Rd = Rs2 - (Rd * Rs1)[15:0])
                  if(!exec2) begin
                        if(Rd[15]) begin
                              mul1 = ~Rd + {16'h0001};
                           end
                        else begin
                              mul1 = Rd;
                           end
                        if(Rs1[15]) begin
                              mul2 = ~Rs1 + {16'h0001};
                           end
                        else begin
                              mul2 = Rs1;
                           end
                        alusum = 17'b00000000000000000;
                        carry = (Rs1[15]^Rs2[15]) ? 1'b1 : 1'b0;
                     end
                  else begin
                        alusum = (carry) ? {1'b0, Rs2 - (~mulresult[15:0] + 16'h0001)} : {1'b0, Rs2 - mulresult[15:0]};
                     end
               end
            6'b011111: alusum = mulextra; // MRT Retrieve Multiply MSBs (Rd = MSBs)

            6'b100000: alusum = {1'b0, Rs1 << Rs2}; // LSL Logical Shift Left (Rd = Rs1 shifted left by value of Rs2)
            6'b100001: alusum = {1'b0, Rs1 >> Rs2}; // LSR Logical Shift Right (Rd = Rs1 shifted right by value of Rs2)
            6'b100010: alusum = {Rs1[15], Rs1 >>> Rs2}; // ASR Arithmetic Shift Right (Rd = Rs1 shifted right by value of Rs2, maintaining sign bit)
            6'b100011: ;

            6'b100100: alusum = {1'b0, (Rs1 >> Rs2[3:0]) | (Rs1 << (16 - Rs2[3:0]))}; // ROR Shift Right Loop (Rd = Rs1 shifted right by Rs2, but Rs1[0] -> Rs1[15])
//          6'b100101: alusum = {{Rs1, carry} >> (Rs2 % 17)) | ({Rs1, carry} << (17 - (Rs2 % 17)));
//                                                                RRC Shift Right Loop w/ Carry (Rd = Rs1 shifted right by Rs2, but Rs1[0] -> Carry & Carry -> Rs1[15])
            6'b100110: alusum = {1'b1, Rd};  //CLL function call
            6'b100111: begin  //RTN return to prev call
                  if(exec2) begin
                        alusum = {1'b0, stackout};
                     end
               end

            6'b101000: alusum = {1'b0, Rs1}; // PSH Push value to stack (Stack = Rs1)
            6'b101001: alusum = {1'b0, stackout}; // POP Pop value from stack (Rd = Stack)
            6'b101010: begin // LDR Indirect Load (Rd = Mem[Rs1])
                  if(!exec2) begin
                        memaddr = Rs1[10:0];
                     end
               end
            6'b101011: begin // STR Indirect Store (Mem[Rd] = Rs1)
                  memaddr = Rd[10:0];
               end

            6'b111110: ; // NOP No Operation (Do Nothing for a cycle)
            6'b111111: alusum = {1'b0, 16'h0000}; // STP Stop (Program Ends)

            default: ; // During Load & Store as well as undefined opcodes
         endcase;
      end
      else begin
         alusum = {1'b0, 16'h0000}; // Bring output low during Load/Store so it does not interfere
      end
   end
endmodule
```

More detail on the multiplier (mul16) is given in Appendix I.

H. C++ code for generating .mif files

```cpp
/*
  Source code for generating MIF files from instructions
  Input format: Text file with each instruction on a seperate line (see below, please follow this new lines breeak this code);
  Output format: Use stdout to redirect to filename.mif or filename.txt and then convert to .mif
  Single instruction format and example:
    1. INSTRUCTION RD RS1 RS2 (all in capitals, seperated by whitespace (this is okay if not exact, code removes whitespace anyway),
       BUT MUST INCLUDE R for registers)
       example: AND R2 R4 R5
                MUL R0 R4 R7
                JMP R2
                STP
       For instructions that use only two registers, example MOV: MOV R0 R1 (do not enter third register, just proceed to next instruction)
       For instructions that use one register, example JMP: JMP R0 (similar to before, just proceed to next instruction)
       For insturctions that use no registers, example STP: STP (and just proceed to next line)
    2. LDA/STA RN MEMORY_LOCATION(DECIMAL) (please make sure memory location is in DECIMAL, otherwise bad things happen with the code)
      example: LDA R3 1546
               STA R6 909
*/

/*
  IMPORTANT NOTE: For the OR instruction, enter it as "_OR", otherwise the code breaks :(
  -Kacper
*/

#include <iostream>
#include <string>
#include <vector>
#include <cassert>
#include <algorithm>

using namespace std;

#define pb push_back

#define endl "\n"
#define IOS ios_base::sync_with_stdio(false); cin.tie(NULL);

const unsigned int RAM_SIZE = 2048;
const unsigned int INSTRUCTION_LENGTH = 16;

string convertBinaryToHex(string binary4){
  if(binary4=="0000"){
    return "0";
  }else if(binary4=="0001"){
    return "1";
  }else if(binary4=="0010"){
    return "2";
  }else if(binary4=="0011"){
    return "3";
  }else if(binary4=="0100"){
    return "4";
  }else if(binary4=="0101"){
    return "5";
  }else if(binary4=="0110"){
    return "6";
  }else if(binary4=="0111"){
    return "7";
  }else if(binary4=="1000"){
    return "8";
  }else if(binary4=="1001"){
    return "9";
  }else if(binary4=="1010"){
    return "A";
  }else if(binary4=="1011"){
    return "B";
  }else if(binary4=="1100"){
    return "C";
  }else if(binary4=="1101"){
    return "D";
  }else if(binary4=="1110"){
    return "E";
  }else if(binary4=="1111"){
    return "F";
  }else{
    cerr << "Invalide binary quartet, cannot convert to HEX (line 78 in .cpp file)" << endl;
    assert(0);
  }
}

string convertDecimalToBinary(int decimal, int digits){
  if (decimal>2047){
    cerr << "Too large memory location, we don't have that much memory for LDA/STA" << endl;
    assert(0);
  }
  string ans="";
  while(decimal>0){
    int rem = decimal%2;
    ans+=to_string(rem);
    decimal/=2;
  }
  while(ans.size()<digits){
    ans+="0";
  }
  reverse(ans.begin(), ans.end());
  return ans;
}
```

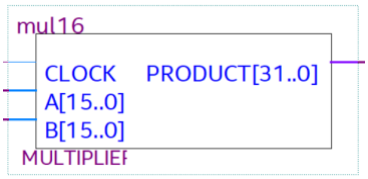
```
97
98  string convertInstructionToHex(string binaryInstruction){
99    string ans="";
100   string temp="";
101   if (binaryInstruction.size()!=16){
102     cerr << "Instruction needs to be exactly 16 bits long, crash in line 105" << endl;
103     assert(0);
104   }
105   for (int i=0; i<16; i++){
106     temp+=binaryInstruction.at(i);
107     if(i==3 || i==7 || i==11 || i==15){
108       ans+=convertBinaryToHex(temp);
109       temp="";
110     }
111   }
112   return ans;
113 }
114
115 string getRegisterBinary(string reg){
116   if (reg.size()!=2){
117     cerr << "Invalid register format, please use form RN, example R0, R3,...; crash in line 120" << endl;
118     assert(0);
119   }
120   if(reg.at(1)=='0'){
121     return "000";
122   }else if(reg.at(1)=='1'){
123     return "001";
124   }else if(reg.at(1)=='2'){
125     return "010";
126   }else if(reg.at(1)=='3'){
127     return "011";
128   }else if(reg.at(1)=='4'){
129     return "100";
130   }else if(reg.at(1)=='5'){
131     return "101";
132   }else if(reg.at(1)=='6'){
133     return "110";
134   }else if(reg.at(1)=='7'){
135     return "111";
136   }else{
137     cerr << "Unknown register input (not between 0 and 7), crash in line 140" << endl;
138     assert(0);
139   }
140 }
141
```

```
142 string getInstructionHex(string instruction){
143   string opcode = instruction.substr(0, 3);
144   string rd = instruction.substr(3, 2);
145   string binary;
146
147   if (opcode=="LDA" || opcode=="STA"){
148     if (instruction.size()<6){
149       cerr << "Instruction format not valid, crash at line 155" << endl;
150       assert(0);
151     }
152     binary="1";
153     if(opcode=="LDA"){
154         binary+="0";
155     }else if (opcode=="STA"){
156         binary+="1";
157     }else {
158       cerr << "Unknown instruction, I think you wanted LDA or STA, crash in line 164" << endl;
159       assert(0);
160     }
161     binary+=getRegisterBinary(rd);
162     int lengthOfAddress = instruction.size()-5;
163     binary+=convertDecimalToBinary(stoi(instruction.substr(5, lengthOfAddress)), 11);
164   }else{
165     binary="0";
166     string rs1, rs2;
167
168     if(instruction.size()>=7){
169       rs1 = instruction.substr(5, 2);
170     }
171     if(instruction.size()>=9){
172       rs2 = instruction.substr(7, 2);
173     }
174     if(opcode=="JMP"){
175       binary+="000000";
176       rs1="R0";
177       rs2="R0";
178     }else if(opcode=="JC1"){
179       binary+="000100";
180     }else if(opcode=="JC2"){
181       binary+="000101";
182     }else if(opcode=="JC3"){
183       binary+="000110";
184     }else if(opcode=="JC4"){
185       binary+="000111";
186       rs2="R0";
187     }else if(opcode=="JC5"){
188       binary+="001000";
189     }else if(opcode=="JC6"){
190       binary+="001001";
191     }else if(opcode=="JC7"){
192       binary+="001010";
193     }else if(opcode=="JC8"){
194       binary+="001011";
195       rs2="R0";
196     }else if(opcode=="AND"){
197       binary+="001100";
198     }else if(opcode=="_OR"){
199       binary+="001101";
200     }else if(opcode=="XOR"){
201       binary+="001110";
202     }else if(opcode=="NOT"){
203       binary+="001111";
204       rs2="R0";
205     }else if(opcode=="NND"){
206       binary+="010000";
207     }else if(opcode=="NOR"){
208       binary+="010001";
209     }else if(opcode=="XNR"){
210       binary+="010010";
```

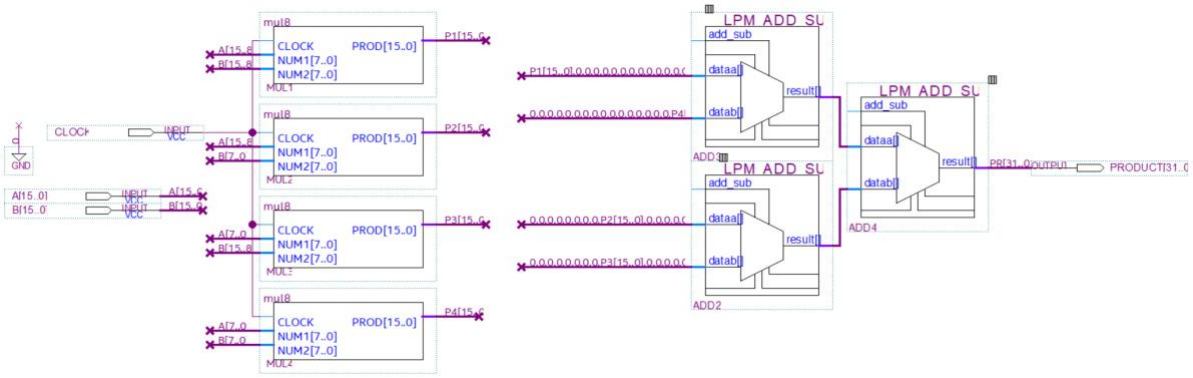
```
    }else if(opcode=="MOV"){
      binary+="010011";
      rs2="R0";
    }else if(opcode=="ADD"){
      binary+="010100";
    }else if(opcode=="ADC"){
      binary+="010101";
    }else if(opcode=="ADO"){
      binary+="010110";
      rs2="R0";
    }else if(opcode=="SUB"){
      binary+="011000";
    }else if(opcode=="SBC"){
      binary+="011001";
    }else if(opcode=="SBO"){
      binary+="011010";
      rs2="R0";
    }else if(opcode=="MUL"){
      binary+="011100";
    }else if(opcode=="MLA"){
      binary+="011101";
    }else if(opcode=="MLS"){
      binary+="011110";
    }else if(opcode=="MRT"){
      binary+="011111";
      rs1="R0";
      rs2="R0";
    }else if(opcode=="LSL"){
      binary+="100000";
    }else if(opcode=="LSR"){
      binary+="100001";
    }else if(opcode=="ASR"){
      binary+="100010";
    }else if(opcode=="ROR"){
      binary+="100100";
    }else if(opcode=="RRC"){
      binary+="100101";
    }else if(opcode=="PSH"){
      binary+="101000";
      rs1=rd;
      rd="R0";
      rs2="R0";
    }else if(opcode=="POP"){
      binary+="101001";
      rs1="R0";
      rs2="R0";
    }else if(opcode=="LDR"){
      binary+="101010";
      rs2="R0";
    }else if(opcode=="STR"){
      binary+="101011";
      rs2="R0";
    }else if(opcode=="NOP"){
      binary+="111110";
      rd="R0";
      rs1="R0";
      rs2="R0";
    }else if(opcode=="STP"){
      binary+="111111";
      rd="R0";
      rs1="R0";
      rs2="R0";
    }else{
      assert(0);
    }
    binary+=getRegisterBinary(rd);
    binary+=getRegisterBinary(rs1);
    binary+=getRegisterBinary(rs2);
  }
  return convertInstructionToHex(binary);
}

void generateMIF(vector<string> instructions){
  cout << "DEPTH = " << RAM_SIZE << ";" << endl;
  cout << "WIDTH = " << INSTRUCTION_LENGTH << ";" << endl;
  cout << "ADDRESS_RADIX = DEC;" << endl;
  cout << "DATA_RADIX = HEX;" << endl;
  cout << "CONTENT" << endl;
  cout << "BEGIN" << endl;
  int i=0;
  for (i; i < instructions.size(); i++){
    cout << i << " : " << instructions.at(i) << ";" << endl;
  }
  cout << "[" << i << "..2047]: 0;" << endl;
  cout << "END;" << endl;
}

int main(){
  IOS;
  string temp;
  vector<string> hexCodes;
  while(getline(cin, temp)){
    auto it = remove(temp.begin(), temp.end(), ' ');
    temp.erase(it, temp.end());
    if (temp.size()>=3){
      hexCodes.pb(getInstructionHex(temp));
    }
  }
  generateMIF(hexCodes);
}
```

I. 1 cycle multiplier block

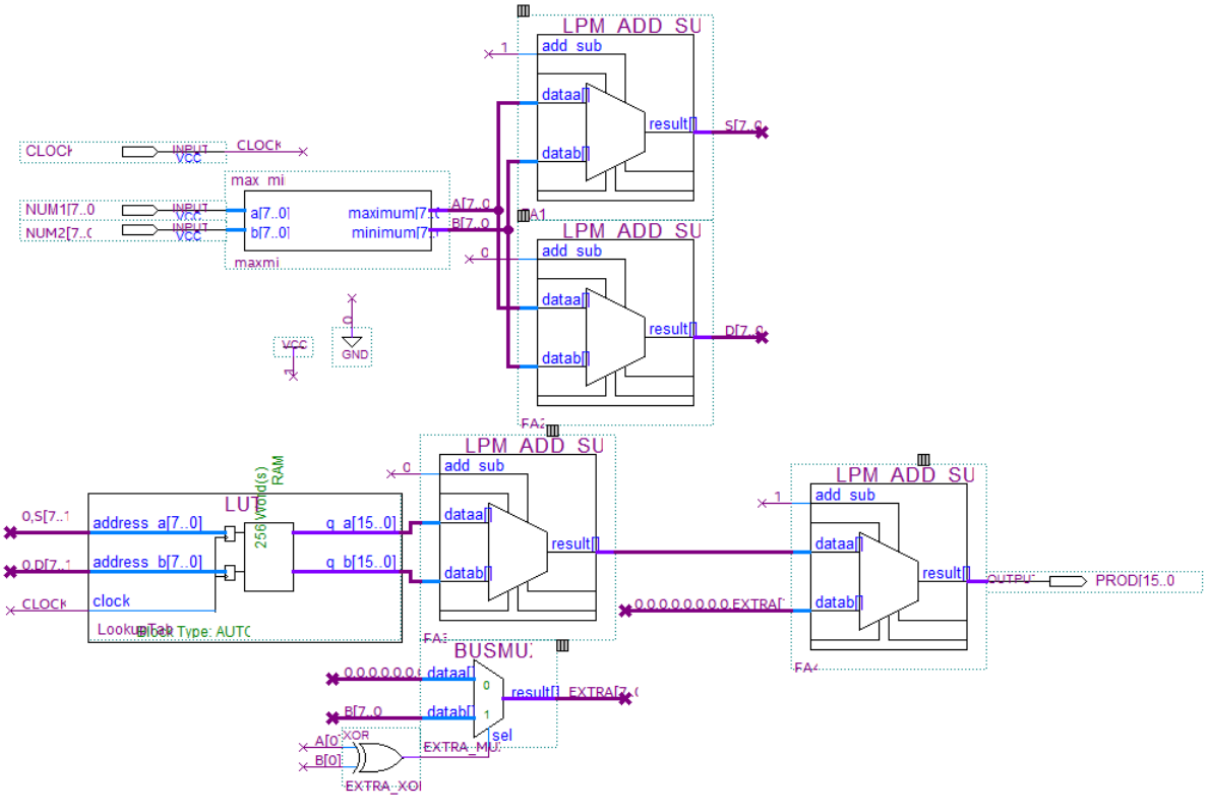
Multiplier symbol:



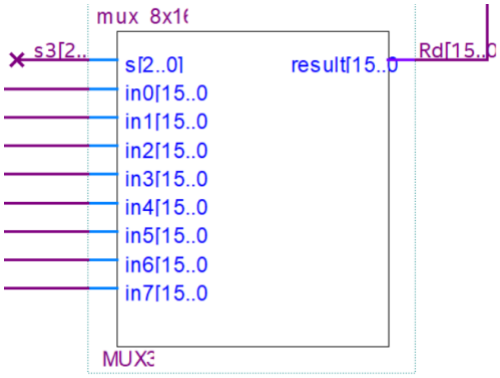
mul16.bdf:



mul8.bdf:

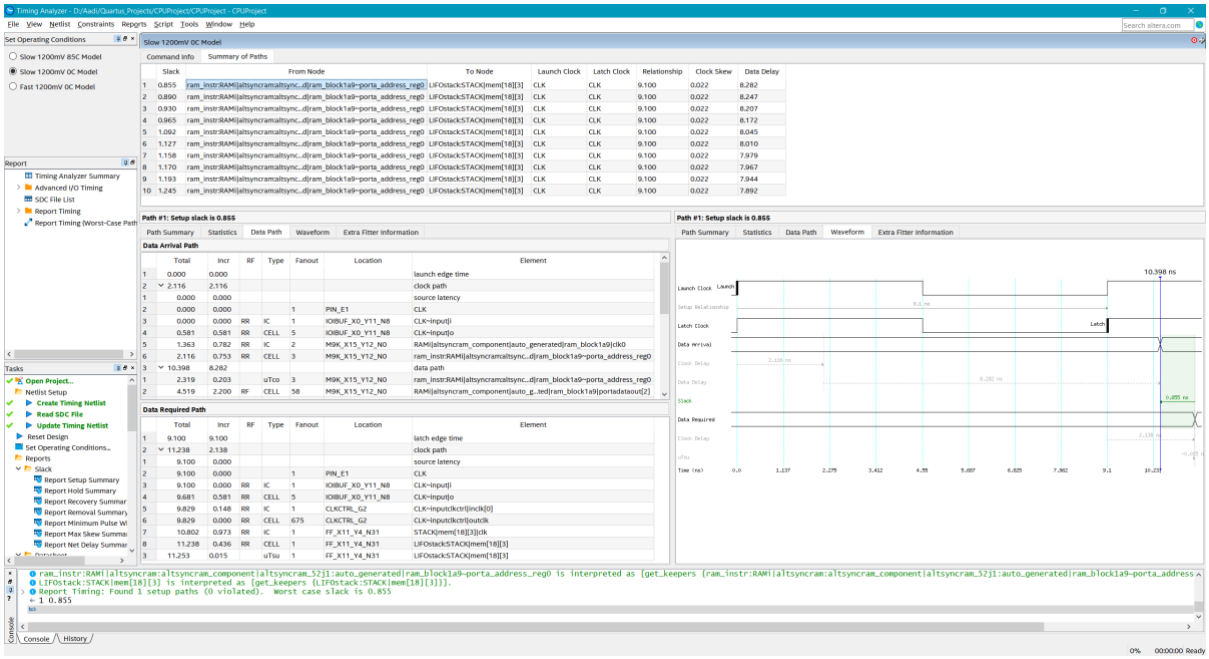


J. Register Multiplexer




```
module mux_8x16 (s, in0, in1, in2, in3, in4, in5, in6, in7, result);

input [2:0]s;
input [15:0]in0;
input [15:0]in1;
input [15:0]in2;
input [15:0]in3;
input [15:0]in4;
input [15:0]in5;
input [15:0]in6;
input [15:0]in7;

output reg [15:0]result;

always @(*) begin
    case(s)
        3'b000: result = in0;
        3'b001: result = in1;
        3'b010: result = in2;
        3'b011: result = in3;
        3'b100: result = in4;
        3'b101: result = in5;
        3'b110: result = in6;
        3'b111: result = in7;
        default: result = in0;
    endcase
end

endmodule
```

K. TimeQuest timing analyser

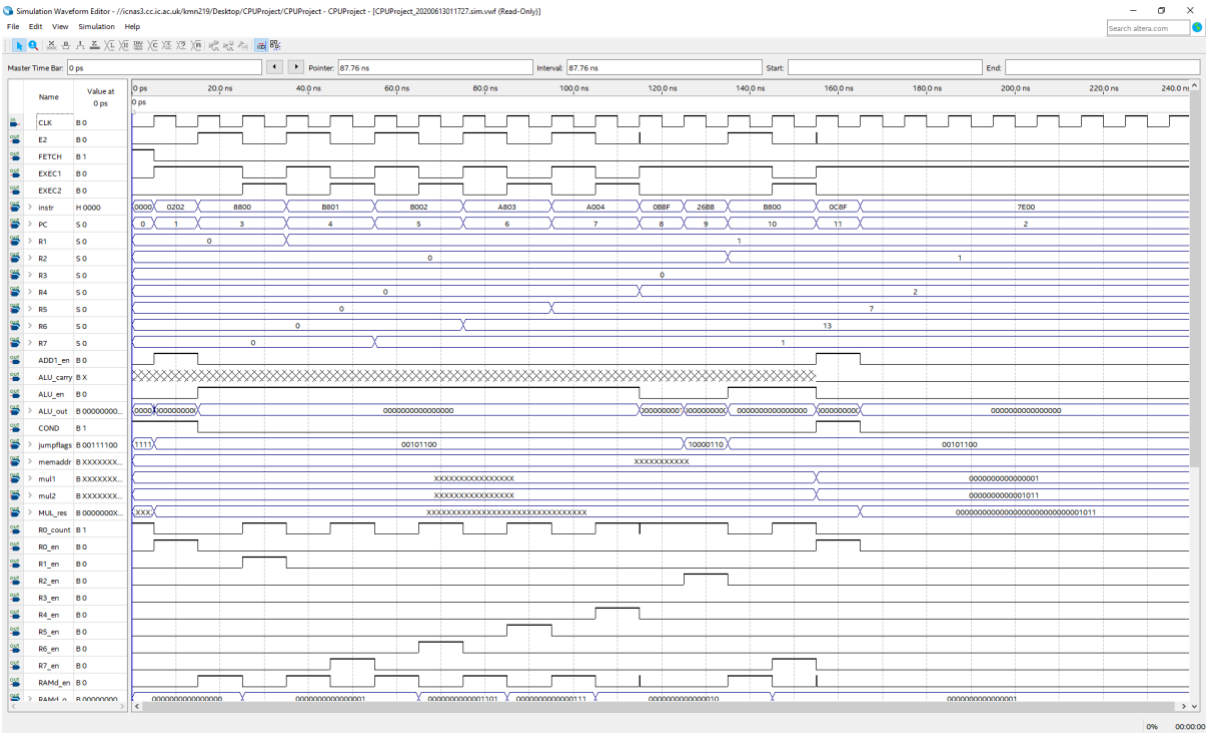


The top panel lists all the paths between nodes in order of least slack to most slack. In the case of the desired clock not being achievable, the worst path is shown first and is red. The lower panels show details about the selected path. The lower left panel shows the route and timing of the signal and the lower right panel is a visual representation of the two clock cycles as well as the time taken for the data to reach the end node, and when the data needs to reach the end node for the result to be valid. In the image, the data delay is 0.855ns shorter than the data required time resulting in the 0.855ns slack.

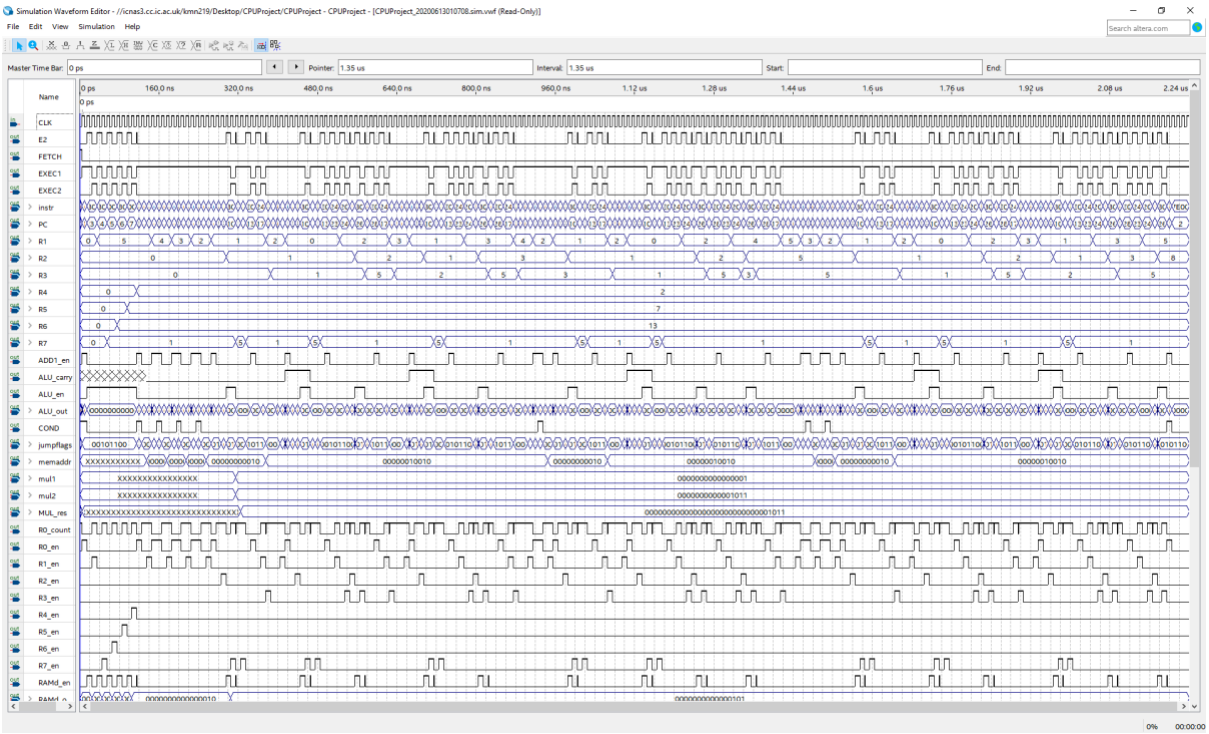
L. Test Waveforms

L.1. Calculate Fibonacci numbers using recursion

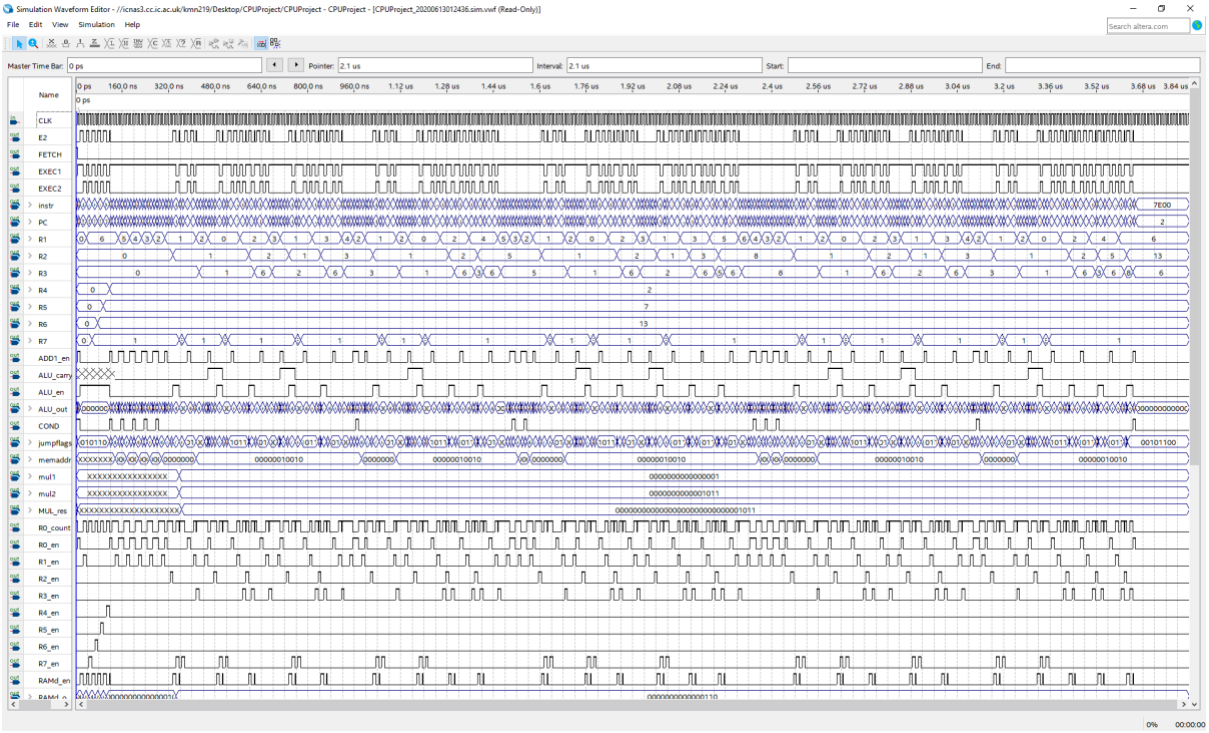
Fibonacci Test (N=1)



Fibonacci Test (N=5) - this is the benchmark test from the project brief



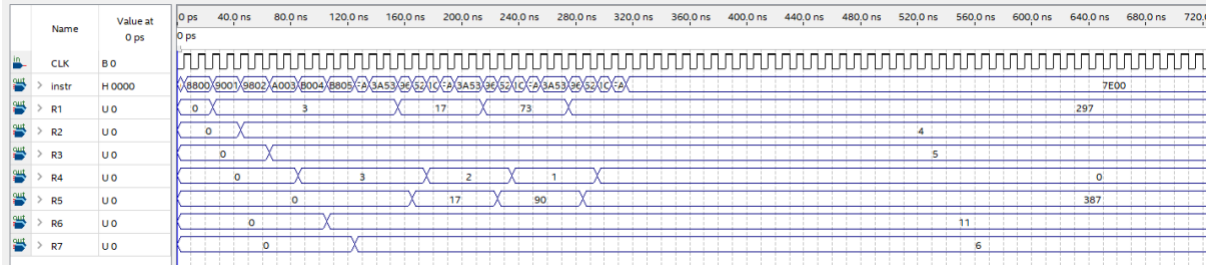
Fibonacci Test (N=6)



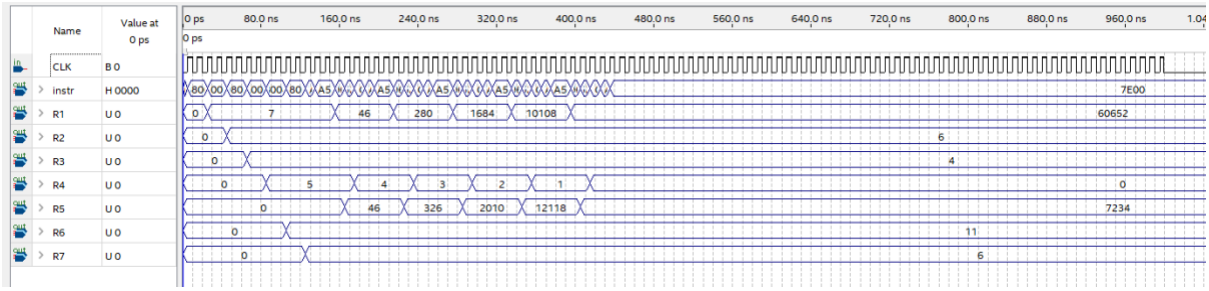
L.2. Calculate pseudo-random integers with a linear congruential generator (LCG)

```
Code used:          Requires setting the following data memory locations:
LDA R1 0            Set location '0' to value of s;
LDA R2 1            Set location '1' to value of a;
LDA R3 2            Set location '2' to value of b;
LDA R4 3            Set location '3' to value of n;
LDA R6 4            Set location '4' to value of 0xB;
LDA R7 5            Set location '5' to value of 0x6;
JC4 R6 R4
MLA R1 R2 R3
ADD R5 R5 R1
SBO R4 R4
JMP R7 R6
STP
```

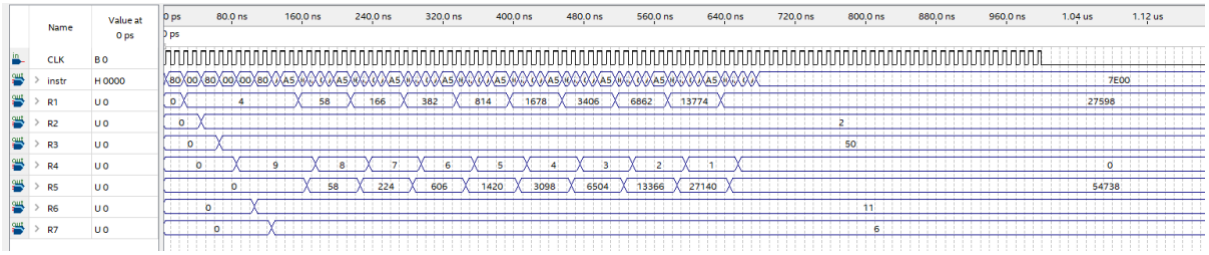
Test 1: (s = 3, a = 4, b = 5, n = 3) Expected 387, Result 387, Took 28 cycles from start to result.



Test 2: (s = 7, a = 6, b = 4, n = 5) Expected 7234, Result 7234, 40 cycles from start to result

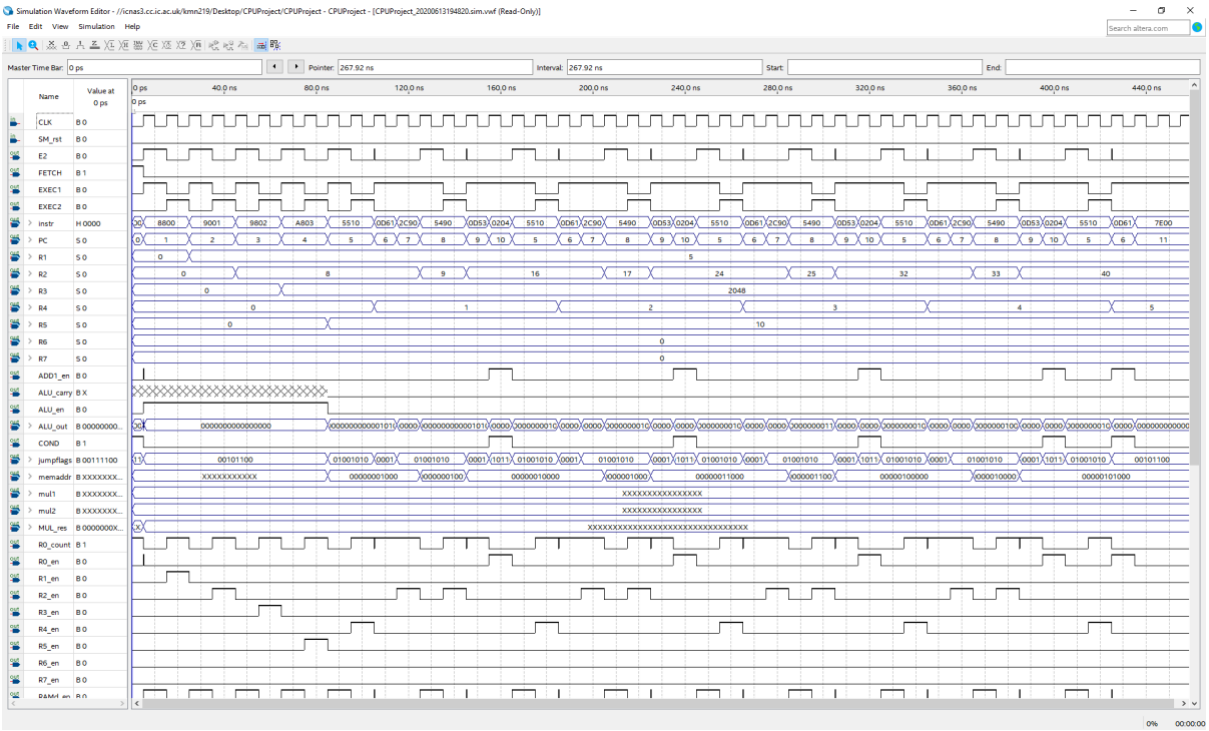


Test 3: (s = 4, a = 2, b = 50, n = 9) Expected 54738, Result 54738, 64 cycles from start to result.

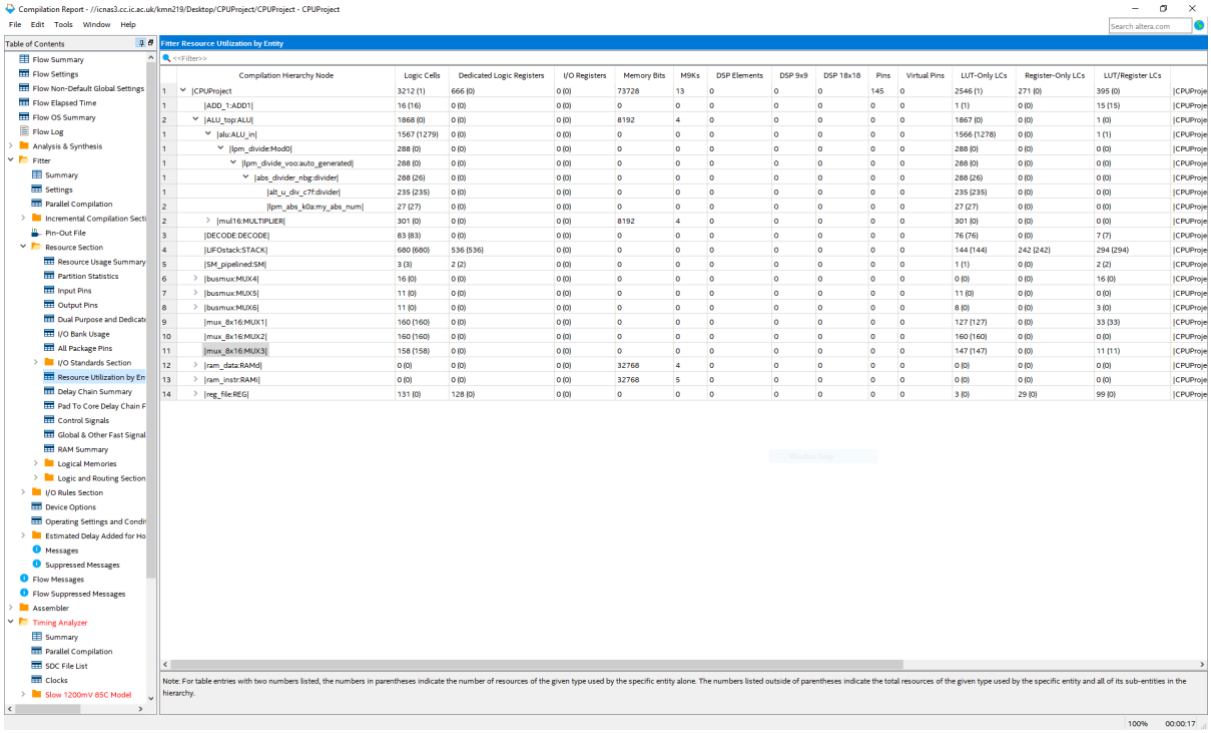


L.3. Traverse a linked list to find an item

The following waveform shows traversal of a linked list to find the value 5.



M. Resource utilisation report



The image above is the resource utilisation of the Pipelined CPU before the RRC instruction was removed. Within the 'ALU_top' instance, there is a lpm_divide block which adds a significant resource use to the ALU without adding much functionality. The image below shows the resulting report where the utilisation is much lower.