Imperial College London

14th June 2020

Intelligently Built Microprocessors

A 16-bit Instruction Set Architecture & Processor Design

ELEC40006 – Electronics Design Project 1 2019-2020

Design Project Report

Word Count: 10,422

Submitted by:

Edvard J. S. Holen CID: 01719707

Hyunjoon Jeon CID: 01514310

Josiah Mendes CID: 01760165

Submitted to:

Dr Edward Stott

Mrs. Esther Perea

Table of Contents

1. Introduction

In the past decades, the dependency on computers has been growing at an exponential rate. They now play a vital part in all fields of research as well as in most people's lives. As such there is a high demand for efficient processors. In this project, work has been done to meet the demand by designing a CPU that is optimised to carry out several tasks including recursive functions, pseudo-random number generation and iteration through linked lists. This report will present and discuss the stages of building the said CPU in the hopes of giving readers a clear view of how the goals of the project were met. In addition, the report presents simulation and test results which give basis for evaluating the design. Lastly, the report will suggest and assess possible improvements to the design.

2. Problem Outline

The aim of the CPU is to be able to work well and efficiently as a general-purpose CPU, but to be specifically optimised to run a set of three separate algorithms.

The first algorithm calculates the **nth number in the Fibonacci sequence** for a given integer input *n*. The Fibonacci numbers are a sequence of integers where every integer is the sum of the two numbers before, except for the first two, 0 and 1 [1].

$$F_n = \begin{cases} n \leq 1 \rightarrow F_1 = 1 \\ \text{else} \rightarrow F_n = F_{n-1} + F_{n-2} \end{cases}$$

Although literature shows there are ways of implementing this function both iteratively and recursively, the client has chosen to use the latter [1]. A typical use of this benchmark would be finding the 5th Fibonacci number.

The second algorithm is a **pseudo-random integer generator** that produces approximately random numbers in the range $[0..2^N - 1]$ where N is usually set as the computer word length. The generator uses the linear congruential method, computing the following sequence:

$$x_{n+1} = (ax_n + b) \bmod 2^N$$

The third algorithm **searches through a linked list** to find an item. Unlike an array, a linked list's data elements are not given by their physical placement in memory. This structure allows for efficient insertion and removal of nodes at any point in the list. Elements in linked lists holds both data and a reference/link/pointer to the next node in the sequence. A typical linked list for benchmarking would hold 10 elements.

3. Product Design Specification

Two of the most common templates for writing project specifications are the “Product Design Specification” (PDS) and the “Software Requirement Specification” (SRS) [2], [3]. For this CPU project, neither of the formats are a bad choice as the cross-disciplinary nature of developing a CPU, involving both hardware and software, allows both specifications to cover relevant elements. Both specification types also cover elements, which though at the given time are irrelevant, may be useful in the future. After careful consideration, it was concluded that the PDS format would be used based on two observations. Firstly, all the elements of the PDS are potentially relevant for the design and production of CPUs [4]. In contrast, the SRS contains elements that will never be relevant, such as “Software Interfaces” and “Software Quality Attributes” [2]. Secondly, the SRS has a very fixed structure, which while being one of its greatest strengths, is also one of its biggest weaknesses [3]. There is little space for considerations regarding disposal, patents, legal, or manufacturing limitations as the SRS is designed with software in mind [3]. The PDS provides a more modular, expandable specification, which is better suited for this project.

The key points of the final PDS are listed below to demonstrate the goals during the design process.

3.1 Key Points

3.1.1 Performance

The CPU should be able to complete the algorithms described in the problem outline efficiently. Therefore, the instruction set, and hardware have to be designed bearing in mind the need to **minimise the amount of time (clock cycles + clock frequency) needed for each algorithm without compromising on correctness, power consumption and area.**

3.1.2 Testing

In order to ensure the correctness of the CPU and its ability to produce outputs that match the expectations, a **test corresponding to each algorithm** will be created together with their expected outputs. In order to aid CPU diagnostics, **a simulation program** will also be produced using C++ that provides the values stored in CPU registers and memory for every instruction stage in a given program.

3.1.3 Documentation

The user will be provided with a set of assembly instructions that are available for use on this CPU together with the encoding and functionality of each instruction. The documentation will also contain information about how to use the simulator.

3.1.4 Ergonomics

While performance in relation to the number of cycles is important, it is also necessary to consider the general-purpose nature of the CPU, hence the instruction set should be structured in a way that is advantageous for others to use, emphasising **clarity and ease of use.**

3.1.5 Time Scale

The product brief was provided by the customer on the 5th of May. The CPU's source files, documentation, demonstration and CPU simulator will be delivered to the customer on **June 14th.**

3.1.6 Processes

The design specification for manufacturing will be created in a Quartus Prime environment, using both **Quartus Prime Block Diagram files** and **Verilog Hardware Description Language** to describe the wiring and logic specifications.

3.1.7 Size

The number of logic devices used to build the CPU is an important aspect since low power consumption is desired, and the logic gates count will be used to estimate the overall power consumption. **Hence minimising the size is another goal** in design.

4 Project Planning

4.1 Staged Development

A project plan was designed to help the team meet the specification. The design process was split into a stage-by-stage implementation process, to break the problem down into smaller more manageable chunks. As each stage contains its own subtasks and usually overlaps with other stages, a Gantt chart was used to manage the timing and planning for each stage of the project in a visually simple and easy to understand way.

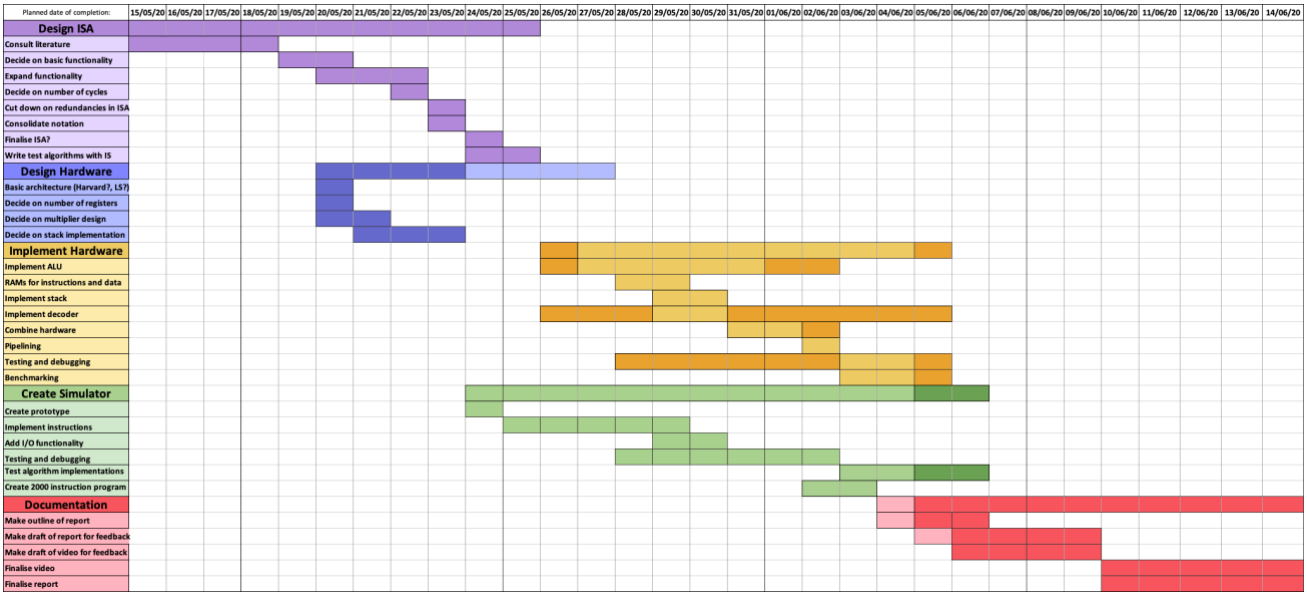


Figure 1 Gantt Chart showing planned and actual time used for each stage and sub-stage of design process. (Darker shades represent time that was used but unplanned, lighter shades represent planned time that was unused.)

The first stage of the design process is to **implement a suitable instruction set architecture (ISA)** for the CPU design. This is important as it lays the groundwork and creates the necessary specifications for hardware design. The design choices here have to be made with reasoning connected to the project brief and client requirements.

The second stage of the design process is to move on to the **hardware design** choices. Overlaps exist between the first and second stage as the two stages are mutually dependent on each other.

The third stage of the design process is to **implement the hardware design** created in the second stage. This stage of the process is the most time-consuming part of the project as it involves a repeated cycle of implementing, testing and debugging.

The fourth stage involves **creating the simulator for the CPU**. This stage runs in parallel to the hardware design and implementation stages. It is crucial for the implementation stage as it provides an accurate reference to the CPU, allowing for quicker debugging and testing by the hardware team.

The final stage of the design process is to **create the formal documentation** for the design to explain the choices made and demonstrate the project's functionality.

4.2 Design Methodology

After evaluating the team dynamic, working styles of each member and the overall goal, the Agile design method was chosen as the overarching design method. The reasoning behind this choice will be discussed in the following sub sections that cover two main values of the Agile methodology.

4.2.1 Iterative Design

The Agile design framework requires commitment to taking a large project and splitting its development into smaller time increments, where a working version of the product is produced at the end of each increment. [5] This working version is then given to the customer for feedback and allows for required changes to be made during the design process. This is especially helpful as it encourages regular reviews of the project specifications and constant striving for improvement.

Although, in this case, a working CPU cannot be developed within the first few days, this approach was suitable for use in each stage of development. For example, the initial ISA was very limited, with only direct load/store and simple arithmetic instructions, but gradually that was extended to include different forms of addressing, offset instructions, stack instructions, bitwise operations and jump instructions.

4.2.2 Regular Feedback and Adaptation

One of the Agile principles is: "At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly."[5] This concept ensures accountability for work done and ensures that every team member is on the same page.

Hence, we had almost daily scrum meetings both morning and evening. The morning meeting usually focused on the goals for the day and any potential issues that could come up. The evening meetings centred around what had been done and what could be done better for the next day. These regular meetings encouraged collaboration and was a time when everyone could discuss ideas and get feedback on work that had been done.

4.3 Team Dynamics

Each team member also took time to self-evaluate on team behaviour, using the Belbin team role methodology. The team was balanced, with Edvard possessing high creativity, imagination and determination traits leading him to take charge of most of the design choices. Hyunjoon's efficient and practical nature, combined with his high programming ability made him the perfect candidate for designing and building the simulator. Josiah shared similar traits, and so worked together with Edvard on the hardware implementation and also individually on the video demonstration. The team's communication skills and high chemistry contributed much to the success of the project.

4.4 Tools

Several different online tools were used to help implement an Agile design methodology and to ensure minimal complications when working together remotely on the same project.

4.4.1 Git and GitHub

Git is an open-source version control system that manages and stores revisions of projects. As it is a distributed version control system, it allows for each user to make changes to a local copy and test locally before committing to a central server. GitHub was used to host the Git repository as it provides an easy to use web-based graphical interface together with several other collaboration features [6]. The branches were useful for working on the same files separately, preventing merge conflicts. GitHub was also used for sharing simulation results and pictures of the design files.

4.4.2 Notion

Notion is a collaboration tool that is marketed as an all-in-one workspace to create wikis, notes, documents and manage projects and tasks [7]. The team used this tool to create the documentation for the instruction set and the details regarding encoding and operation. This tool was chosen over a text document in GitHub for several reasons. Firstly, it syncs automatically, reducing the need for specific commands to update the central copy. Secondly, the user

Figure 2 Screenshot of Notion User Interface and Team Homepage

interface is more intuitive than a text document in GitHub as it has highlighting support and embeddable widgets such as a calendar. Another advantage of Notion was that it keeps page history and lists all changes made so it was easy to view iterations of the instruction set and revert back if necessary. Notion was also used to keep track of deadlines that we had set based on the Gantt Charts and also keep track of notes from each meeting. The Notion page is available for viewing in Appendix A.

4.4.3 Communication Tools

A group chat on WhatsApp was the main point of communication. Meetings were held via Microsoft Teams. These were chosen as all members of the team already had pre-existing accounts and were available in the regions where each team member was located.

5. Design Process

5.1 Instruction Set Design

The implemented ISA contains 21 instructions with variable opcode length to allow for extra functionality for certain instructions. Therefore, while most instructions use 5 bits to identify the operation some only use 3 or 4 bits. Every instruction consists of 16 bits in order to meet the CPU word length requirement. Details on each instruction can be found in Appendix A.

5.1.1 Arithmetic Instructions

Addition and subtraction instructions are fundamental for any ISA. To avoid looping addition instructions, a specialist multiplication instruction was made, specifically with the random number generator in mind.

- **ADR/SBR – Addition/Subtraction between registers**

  Rn = Rn + Rm + Cin / Rn - Rm - Cin

  ADR instruction adds Rn and Rm with carry in and stores the sum in Rn. The user can choose the carry in value and add an offset. SBR instruction performs subtraction rather than addition.

- ADM/SBM – Addition/Subtraction from memory

  Rn = Rn +/- Mem[N]

  ADM instruction adds Rn and the data from location N in the memory and stores the sum in Rn. SBM instruction performs subtraction rather than addition.

- ADI/SBI – Immediate Addition/Subtraction

  Rn = Rn +/- N

  ADI instruction adds Rn and a 9-bit unsigned immediate value N and stores the sum in Rn. SBI instruction performs subtraction rather than addition.

- MLR – Multiplication between registers

  Rn = Rn * Rm + Cin

  MLR instruction multiplies Rn and Rm, adds carry in and stores the result in Rn. The user can choose the value of carry in and add an immediate offset if desired.

5.1.2 Logical Instructions

The following logical instructions are included for advanced operations. For example, shift instructions can be combined with LDI instruction to fill the entire

register as LDI instructions are limited to 11 bits. BBO instruction can be used for complex if-statements involving Boolean operations.

- BBO – Bitwise Boolean Operations

Rn = NOT Rm + N, if NOT operation is chosen

Rn = Rn BBO Rm + N, otherwise

BBO instruction performs a bitwise operation using Rn and Rm, adds an immediate value N and stores the result in Rn. The user can choose between AND, OR, XOR, XNOR, NAND, NOR, and XNOR. The operand N allows for easy inversion in both two's complement and one's complement.

- XSL/XSR – Left shift/Right shift

Rn = Rm XSL N

Rn = Rm XSR N

XSL instruction shifts Rn by N places to the left and stores the result in Rn. The user can choose the value of carry in which will be shifted in during the operation. XSR instruction operates similarly but shifts the value to the right instead.

5.1.3 Load Store Instructions

3 types of addressing are used in the ISA, immediate, direct and register addressing. Indirect addressing is used for LDI, ADI and SBI instructions, being useful for counter operations and loading registers. Direct addressing is used for LDA, STA, ADM and SBM instructions to reduce cycle count. It's inclusion also reduces the effect of the lack of registers, by allowing for algorithms that may need more values to be stored in memory without a large performance loss. Register addressing is used for the rest of the instructions. It provides a more flexible form of addressing as it takes up less bits in each instruction, allowing for space to carry out immediate offsets, register offsets and scaled register offsets. These addressing modes are useful for manipulating data structures like arrays or vectors to keep track of the index and iterate through a set of data. Note that Rb << s means shifting Rb by *s* places to the left.

- LDI – Load Immediate

Rn = N

LDI instruction stores an 11-bit unsigned immediate value N in Rn. Note that N is zero extended to 16 bits before being stored in Rn.

- LDA – Load using direct addressing

Rn = Mem[N]

LDA instruction loads the data at location N in the memory and stores it in Rn.

- LDR – Load using register addressing

  Rn = Mem[Ra + N] or Rn = Mem[Ra + Rb << s]

  The user can choose an addressing mode between “Register with immediate offset” and “Register with scaled register offset”. If the former is chosen, LDR instruction loads the data at location given by [Ra + N] and stores in Rn. If the latter is chosen, the memory location is given by [Ra + Rb << s]. This is detailed in section 5.2.2.

- STA – Store

  Mem[N] = Rn

  STA instruction stores Rn in the memory at location N.

- STI – Store Indirect

  Mem[Ra + N] = Rn or Mem[Ra + Rb << s] = Rn

  The user chooses an addressing mode between “Register with immediate offset” and “Register with scaled register offset”. If the former is chosen, STI instruction stores Rn in the memory at location given by [Ra + N]. If the latter is chosen, the memory location is given by [Ra + Rb << s].

5.1.4 Stack Instructions

STK instruction is included to use the stack. The instruction is necessary for calculating Fibonacci numbers using a recursive implementation.

- STK – Pushing/Popping to the Stack

  PUSH[Rx ± N] or Rx = POP

  The user chooses an operation between push and pop. If push is chosen, STK instruction adds Rx and a 6-bit unsigned immediate value N and pushes the result to the stack. If pop is chosen, the top value from the stack is popped from the stack and stored in Rx.

5.1.5 Jump Instructions

Jump instructions are essential and without self-modifying code they are necessary for Turing completeness [8]. Having powerful jump instructions can severely cut down on execution time as it simplifies the implementation of conditional code elements such as if/else-statements and while loops. The ISA therefore includes a range of conditional jumps in addition to an unconditional jump instruction.

- JMR – Register Addressed Jump

PC = Rm

JMR instruction performs a conditional jump to Rm. The condition is chosen by the user among “if Rn = 0”, “if Rn ≠ 0”, “if Rn = Rx”, “if Rn ≠ Rx”, “if Rn < Rx”, “if Rn ≤ Rx”, “if Rn[X] = 1”, and “if carry = 1”.

- JMP – Direct Addressed Jump

PC = N

JMP instruction performs an unconditional jump to N.

- JEQ/JNQ – Jump if equal to 0/Jump if not equal to 0

PC = N, if R0 =/≠ 0

JEQ instruction performs a conditional jump to N if R0 is equal to 0. JNQ instruction operates similarly but the condition is if R0 is not equal to 0.

STP

Stops the CPU from executing further instructions.

5.2 Hardware Design & Implementations

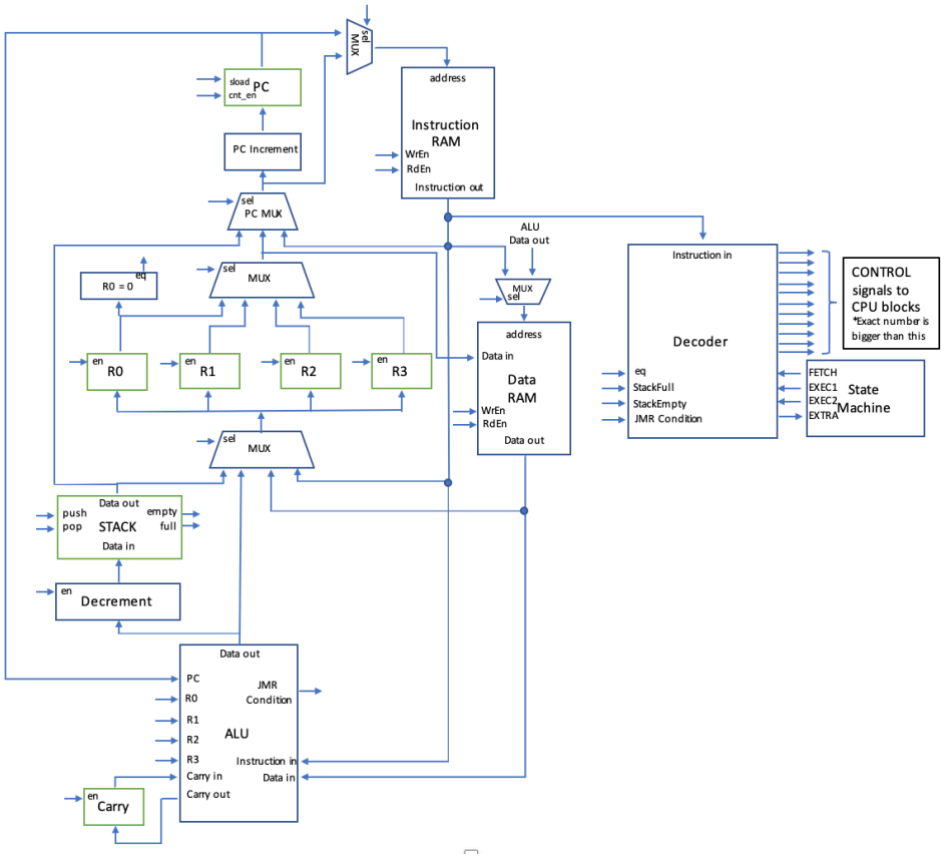


Figure 3: Diagram showing the layout of the CPU.

5.2.1 Overview

The CPU employs a Harvard Architecture [9] where instructions and data are held in separate memory units. For the specification, the Harvard architecture holds significant advantages over the Von Neumann architecture. Firstly, as they operate on separate buses, data writes and instruction reads can be simultaneous. This is important when pipelining, as instructions requiring memory access in EXEC1, such as the STA and STI instructions, can be pipelined with the FETCH cycle of the next instruction, avoiding pipeline stalls. Secondly, as the instruction memory is only accessed during the fetch cycle of an instruction, the current instruction can be kept on the output of the memory for its entire duration, removing the need for instruction registers. Finally, as none of the instructions require access to both the data and instruction memory, fewer bits of the instructions are needed to specify memory locations. This allows for 5-bit opcodes rather than 4 bits, expanding the number of instructions.

The CPU has 4 registers. This number is a compromise between optimal performance for the given algorithms and optimal general performance. The LCG

algorithm is the benchmark which benefits most from more registers and would need 5 registers to have the best performance. Thus, it would seem logical to have 8 registers in the CPU. However, the number of bits needed to address 8 registers in two, or three, operand instructions is significant. Moreover, as the instruction set has a 16-bit architecture the number of bits is limited. If the decision was taken to have 8 registers, this would have to come at the expense of other very useful features such as being able to choose the Carry-in or give operands immediate offsets. Moreover, direct addressed memory operations such as STA and LDA would become completely unfeasible, as if they had to address 8 different registers, there would only be 2 bits left for choosing the operation. It was therefore decided that a CPU with 4 registers would be more advantageous to having 8 registers.

The ISA has a Load/Store structure with most arithmetic instructions exclusively operating between registers. The exceptions to this are the ADM and SBM instructions. The Load/Store structure allows for a shorter average cycle requirement for each arithmetic operation, as no memory read is necessary. The short cycle requirement for arithmetic instructions was important to efficiently exploit the two-cycle pipeline of the CPU. The ADM and SBM instructions which break with the Load/Store structure were added to give better functionality to programs with a high number of variables, such as the LCG algorithm where some variables do not have space in the registers and need to be stored in the memory.

5.2.2 Wallace Multiplier

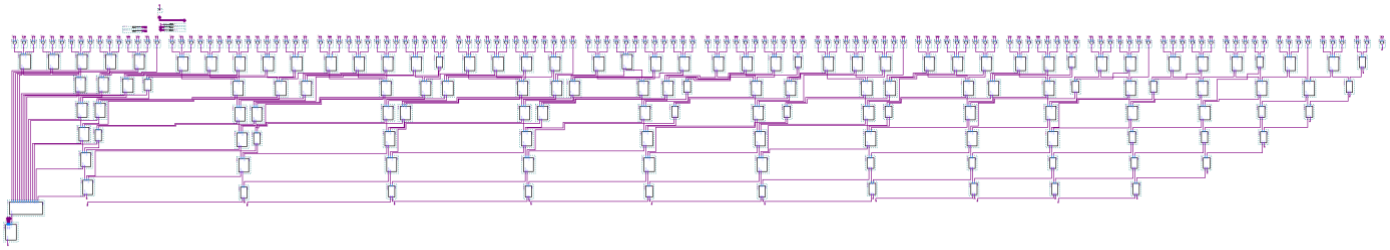


Figure 4: Photo of the Wallace multiplier as implemented in the CPU.

The specification calls for a CPU which is efficient at multiplication. Multiplication can be implemented directly through dedicated hardware, or indirectly through software. The most prominent drawback of using dedicated hardware for multiplication is the large number of transistors and thus area needed to implement a multiplier. In the earlier days of computing it was therefore uncommon to see dedicated hardware implementations of multipliers in processors [10]. However, due to shrinking transistor sizes, it has become feasible to include dedicated hardware multipliers in all but the smallest systems. Software implementations have the benefit of requiring fewer transistors than those in hardware, but they are often very slow. For example, doing 16-bit multiplication on the MU0 architecture can take hundreds of clock cycles.

To get an efficient implementation of the LCG algorithm it was therefore necessary to sacrifice area and power consumption, for the benefit of added speed through dedicated multiplication hardware.

Two multiplier designs were seriously considered for the CPU. Those being the Dadda and Wallace multiplier architectures [11]. The two are closely related and rely on similar algorithms. First partial products are created by AND-ing all two-bit combinations of the input values. The partial products then pass through reduction layers until no bit of the output has more than two weights corresponding to it. Finally, the outputs are added to give the answer. Though similar, the architectures differ in the algorithm used for creating the reduction layers. Wallace multipliers apply an aggressive reduction technique, merging the maximum number of weights every layer. Dadda multipliers have a less aggressive reduction technique which instead tries to minimise the gate count of the circuit. It is debated which of them is better and there are papers stating that the Dadda multiplier is smaller and faster [12] as well as research papers stating the opposite [13]. However, this research is ultimately irrelevant to the CPU design as they only considered multipliers with a full-size output. The given CPU design only requires a 16x16 multiplier with a shortened 16-bit output, not the full 32-bits. Independent research was therefore necessary. The reduction algorithms of the two multipliers were implemented in C++ allowing the adder requirements for each of the multipliers to be calculated. The results for different input sizes are shown in figure 5 below. From the results it was concluded that the Dadda multiplier needs both fewer full adders and half adders than the Wallace multiplier, and thus requires less area to be implemented. Moreover, for 16x16 multipliers the number of reduction layers is the same, meaning that the two designs likely will have very similar latencies.

Multiplier size:	4x4	4x4 (4-bit output)	8x8	8x8 (8-bit output)	16x16	16x16 (16-bit output)	32x32	32x32 (32-bit output)
Wallace Multiplier:								
Total full adders:	4	3	36	18	196	96	900	441
Total half adders:	6	2	25	11	79	34	220	99
# reduction layers	2	2	4	4	6	6	8	7
Dadda Multiplier:								
Total full adders:	3	1	35	15	195	91	899	435
Total half adders:	3	2	7	6	15	14	31	30
# reduction layers	2	2	4	4	6	6	8	8

Figure 5: Table showing the number of adders needed to implement multipliers

However, the determining factor ultimately became ease of implementation. While it was concluded that the Wallace multiplier was less area efficient than the Dadda multiplier, the algorithm used for implementing the reduction layers of a Wallace multiplier is both simpler and more regular than that used for Dadda multipliers. Both multipliers have more than a hundred adders and are rather complex circuits. Thus, the risk of making errors while implementing them is large. A small error such as a missing connection between adders or similar is very hard to detect as the error, might only manifest itself for a small number of input combinations. The simpler reduction algorithm, and consequently easier implementation, of the Wallace

architecture was therefore determined to hold greater importance than the higher area efficiency of the Dadda multiplier. Thus, the Wallace architecture was used for the final CPU design.

5.2.3 Stack

Recursion, the technique of solving a problem by solving smaller instances of the same problem[14], is used by the first benchmark to calculate the Fibonacci value for a particular integer. This leads to the question of how functions are to be implemented in assembly, as this function needs to call itself. It's not a loop, hence simple jump instructions will not suffice. Each call needs to save the current processor state before calling the function again until the base case is reached, and once that base case is reached, the stored processor states are restored and using the return value of the function at each nested call, the algorithm successfully runs.

It can be seen that the first stored processor state is also the last returned, hence it needs to be stored in a specific data structure where the last value that was input is the first value out, otherwise known as LIFO (Last In First Out). This data structure is specifically referred to as a **"stack."**

The stack is used to store temporary variables and for implementing function calls. At assembly level, most instruction sets have the capability to manipulate the stack with direct instructions, namely the push and pop instructions that add data to the top of the stack and remove data from the top of the stack respectively. [15] Regardless of the implementation method, stacks conform to a basic architecture. Initially the size of the stack is zero and it grows in size as more data is "pushed". With each "pop" operation, the size of the stack decreases by 1 unit. The size of the stack is dynamically allocated depending on the amount of data that is stored [16].

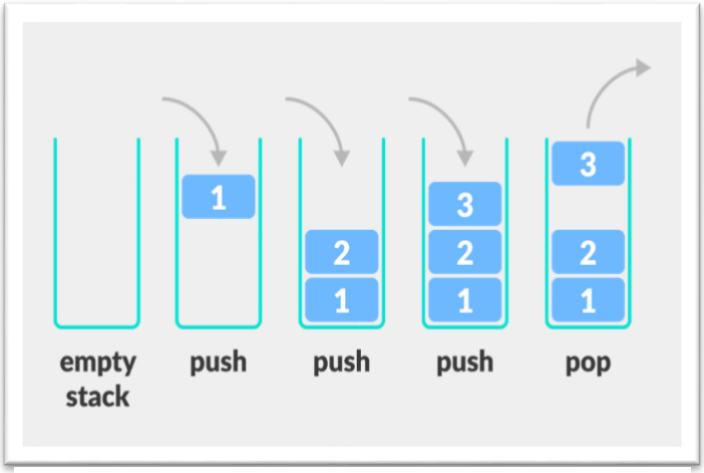


Figure 6 Diagram describing stack operation [20]

At a hardware level, there are two main ways that a stack can be implemented. The first way is to make use of main memory and have a dedicated register that stores a pointer pointing to the top of the stack. This approach is taken by both Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC) CPUs, such as the x86 and ARM based CPUs respectively [17]. The pushing and popping

operations in this case are pseudo-instructions that combine load/store register addressing and immediate addition/subtraction [18]. The second approach taken is to have a dedicated set of registers or memory block for the stack. This is less widely used than the previous approach, with the most widely known CPU that implements this being the x87 architecture CPUs [19].

Both of these were considered in the hardware design process. However, due to the choice to limit the amount of registers to 4, it was decided that keeping a register just to store a pointer to the stack location would be a waste of resources. Another issue that would have arose if the first solution was implemented would be that stack instructions would take extra cycles increasing execution time for the benchmarks. Therefore, a separate block using a set of 256 registers was used to implement the stack memory unit in Verilog.

The block takes in inputs that describe the operation, push or pop, the data that is to be pushed onto the stack and a clock input. There are three outputs, one for reading the value at the top of the stack and two that show if the stack is either completely empty or completely full. The empty and full outputs are important as when the stack is empty, 0 is output onto the data bus. Since the empty and full outputs exist, there is extra peace of mind as the decoder can use the empty output to decide that the pop operation will not be carried out if the stack is empty. The same applies to the push operation that will not be carried out if the stack is full. These empty and full outputs are decided by using a combinational count inside the block that keeps track of how much data has been stored in the stack at any point in time.

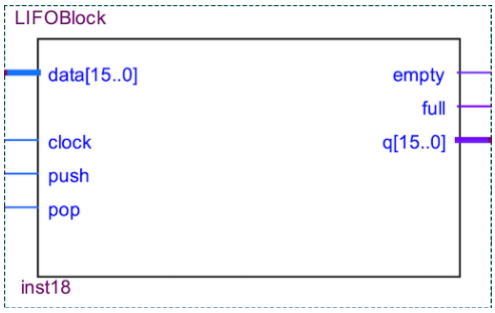


Figure 7 Generated Block Diagram of Stack Memory Unit

This implementation of the stack has its disadvantages such as being limited to only accessing the top stack value and none below. It follows a very strict LIFO data structure, unlike the first solution where the stack pointer can be moved backwards and forwards to read different parts of the stack. However, this implementation is significantly faster and also works considerably better with the current ISA design. If there were more registers in the CPU, there would be a case for implementing a stack that utilised the main memory. This design could also be extended to include an addressing system that allowed for data stored in the stack to be output at any given time, however this may increase the cycle time for a read/pop instruction.

5.2.4 Shifts

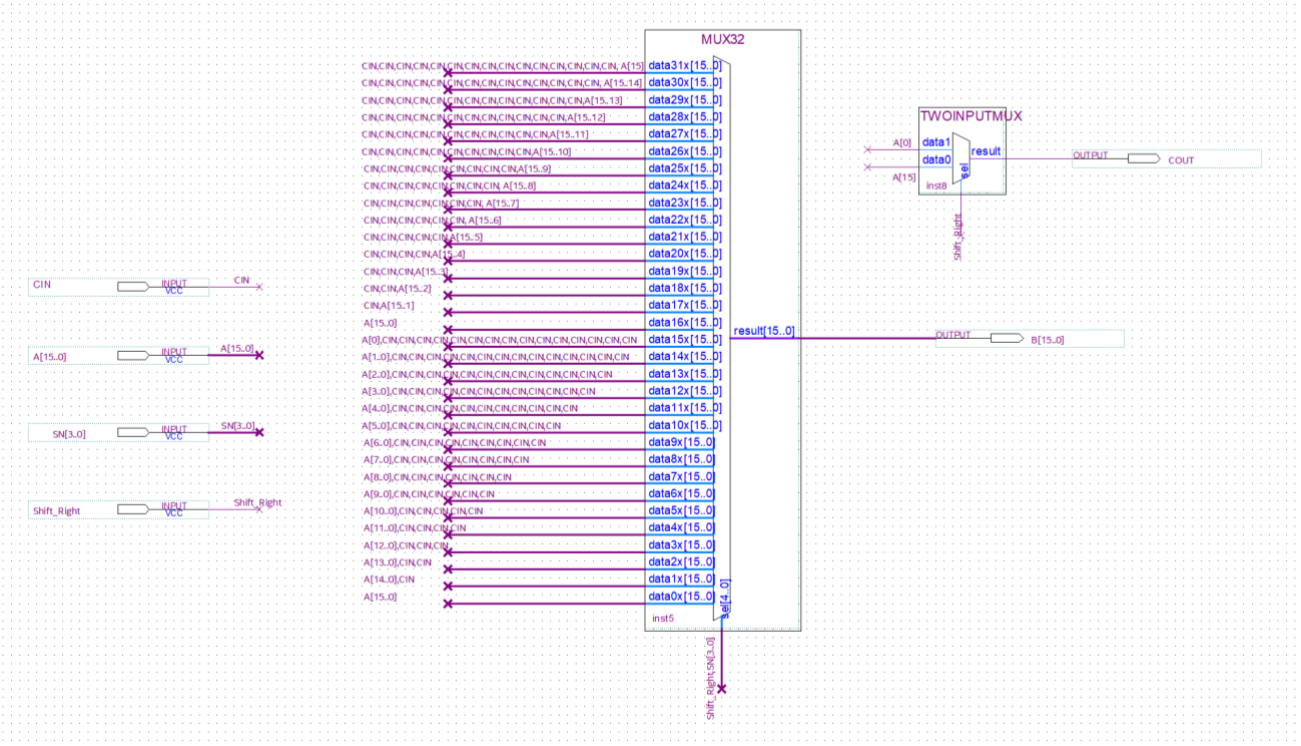


Figure 7: 32-input multiplexer used to implement the shifts in the CPU.

The CPU has a powerful shift mechanism which allows for shifts by up to 15 bits in each direction. The shifts are implemented through a 32-input multiplexer and the value that is shifted in can be set to 0, 1, CARRY or the MSB of the input value. In addition to the output corresponding to the shifted value the shifter has a CARRY output which is equal to the MSB or the LSB of the input value, depending on whether the shift performed is to the left or to the right. The shifter is not the most area efficient and could, for example, have been implemented using only two 2-input multiplexers and a 16-input multiplexer instead. However, several multiplexers in series results in high latency and it was therefore desirable to be perform all shifts through one large multiplexer rather than through several smaller ones.

5.2.5 Instruction pipelining

At all times there will be parts of the CPU which go unused by the current instruction. This is unavoidable, yet highly undesirable. Not only does components going unused mean that processing power is wasted, it is also very energy inefficient as components will have a significant amount of static power dissipation even when they are not being used.

To minimise the problem of unused logic the CPU utilises instruction pipelining [20]. This allows the CPU to start the next instruction while the current instruction is not yet completed. The CPU has a pipeline of length two and separates between the fetching and execution of instructions. This means that the fetching the next instruction from the instruction RAM (FETCH) happens simultaneously with the last execution cycle of the current instruction (EXEC1 or EXEC2). Thus, the number of cycles each instruction needs to be completed is reduced by one. The exception is the first instruction of any program, which does not have any previous instruction to overlap its FETCH cycle with.

The pipeline was chosen to have a length of two as this is the number of cycles the majority of the instructions in the ISA require to be completed from the start of FETCH to the end of execution (EXEC1 or EXEC2). Out of the 21 instructions in the instruction set 16 require two cycles for completion and 5 instructions require three. Assuming every instruction is equally likely to occur, in an unpipelined CPU the average execution time of each instruction would be 2.238 cycles. After pipelining this is cut down to 1.238. As a consequence, the average execution time is lowered by 44.7%, nearly doubling the number of instructions that can be completed per unit time.

There are some weaknesses to pipelining. Firstly, there may be a dependency between the FETCH cycle of an instruction and the execution cycle it is being overlapped with. An example of this is jumps, where the memory location of the next instruction may or may not be the current value of the PC. This problem is however solved through operand forwarding [21]. As such, the memory address of the instruction RAM is fed by a multiplexer, making it possible to choose whether the memory address is given by the PC or the operand of the jump instruction. Some CPUs of the Von Neumann architecture also have a problem with instructions requiring memory access during the execution phase. However, the CPU's Harvard architecture helps avoid the pipeline stalls that otherwise would occur in such situations. Lastly, when a CPU is pipelined the value passed from the PC into the instruction memory address is, at most times, one higher than the current instruction number. This might cause issues when an instruction reads or operates on the value of the PC. To solve "off by one" errors of this nature it was necessary to add an increment block on the PC and a decrement block on the stack. Though the addition of hardware is needed to circumvent these issues, it should be clear that the extra area and power required by these few devices is a downside far outweighed by the advantage of a 44.7% reduction in execution time. Moreover, as each program finishes more quickly the energy spent on static power is severely reduced, giving a net negative change in energy usage.

5.2.6 Optimisations

Once the CPU was functional, and able to complete the tasks, adjustments were made to the design to increase its clock frequency so that the tasks would be able to be completed quicker. These optimisations were done by running timing analyses on Quartus, together with finding the longest paths within the CPU to minimise slack and thus increase clock speed. Each optimisation and its results are summarised in figure 8.

Change	Clock Freq
Initial CPU	42.85Mhz
Pipelined Multiplier	57.36 Mhz
Improved Shifting	63.56Mhz
Removed Multiplexers	69.5Mhz

Figure 8 Optimisation Summary

Initially, without any optimisations, the clock frequency was around 42.85Mhz. The longest path analysis showed what was expected, the Wallace Multiplier was causing a significant delay. After looking at the trade-offs between cycle count and frequency, the decision was made to pipeline the multiplier, increasing multiplication operations by 1 cycle, but increasing clock frequency to 57.36Mhz. By repeating the analysis procedure, it was seen that the longest path was now through the shifts. The original shift block made use of two Quartus mega function shift blocks for both left and right, and after some thinking, the decision was made to just use a 32-input multiplexer as described before. This further improved the clock frequency to 63.56Mhz. Once again, the analysis cycle was repeated, and a few redundant multiplexers were found and by removing them, the clock speed increased to a final value of 69.5Mhz.

5.3 Simulator

In order to aid CPU diagnostics, a simulation program for the CPU was written in C++. The general structure of the simulator is a class which has member variables representing registers, stack, program counter, memory, etc. And every instruction is implemented as a member function of the class which will be called when the corresponding instruction is executed.

5.3.1 Benefits

Ensuring the quality and reliability of the product is very important. For this, a simulator can be employed. It is essential to make sure that the simulator works accurately for every instruction so that it can be an accurate representation of the CPU. This way of testing with the simulator is much faster, accurate, and less prone to human errors. By having a simulator, it was also possible to test a program that has large amount of instructions which would have been tedious if needed to be done by hand.

5.3.2 Capabilities

The main ability of the simulator is to show the values stored in the CPU registers and memory after the execution of each instruction in a given program. Figure 9 shows an example of how the simulator visualises the state of the CPU at a particular instruction stage. Each snapshot of the CPU starts with the Program Counter value immediately after the previous instruction. This is because values are written to the registers after the execution of the instruction is finished and the task of the simulator is to show how the values in each register and memory are changed after executing the instruction. The simulator shows not only the values in each register but the current carry value, the last executed instruction, and the current stack size with stored values. Note that the value placed at the top of the stack list is the top element of the stack. The simulator can express values in both binary and hexadecimal, so the user can read values efficiently without needing to convert between the two. Moreover, the opcode bits of the instruction are translated to the corresponding title of the instruction for better readability. The memory part shows locations where data is modified since the simulation began. Other locations in the memory have values of zero as indicated.

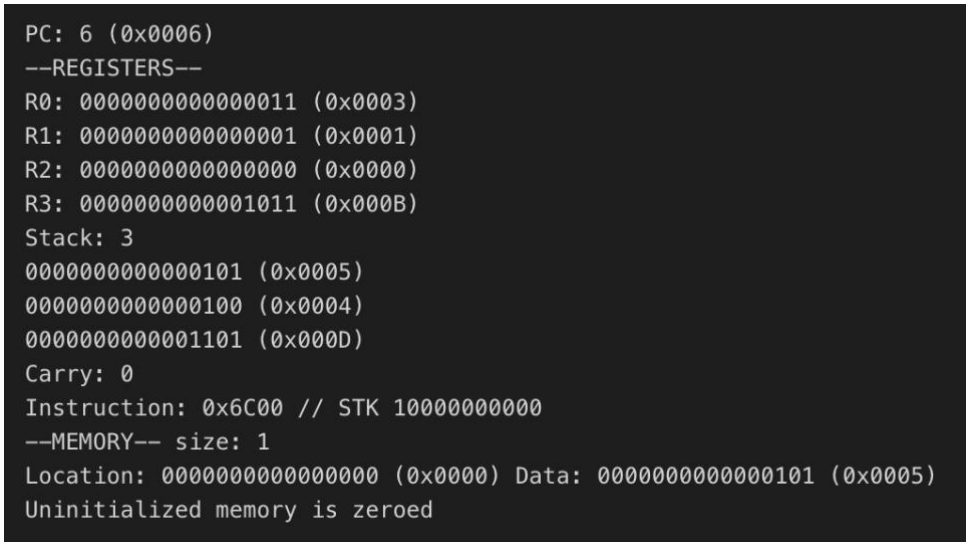


Figure 9: Example of the simulator showing the CPU contents at a given PC value

In order to use the simulator, the user needs to input a text file containing instructions. While creating the input text file, the instructions can be written in two different forms, either instruction name with immediate value or 16-bit hexadecimal number. For example, an ADR instruction R0 := R0 + R3 + 1 can be written either “ADR 00100000011” or “0903”. The first method of writing instructions was used mainly for testing and debugging purposes since it’s more convenient in changing a single bit in the immediate part. The second method was used for running test programs when the simulator result was compared with the CPU.

5.3.3 2000 Instructions Program

As part of testing, a program that contains 2000 instructions was created. The purpose of using the program with such a huge number of instructions is to ensure that the CPU works as intended by checking if the result matches with the expectation calculated by the simulator. Initially, the program was created using a C++ program which generates 2000 random 16-bit hexadecimal numbers. However, since some combinations may not be wanted, some instructions were added more carefully. For example, with STK instruction the user can choose which register is pushed to or popped from the stack with 3 bits. But certain combinations of these 3 bits do not indicate any register therefore, STK instructions with those combinations had to be avoided. In addition, jump instructions can possibly cause unwanted infinite loops and the CPU would never reach a STP instruction. Also, STP instruction in the middle of the program would stop the CPU from executing further instructions which makes having 2000 instructions meaningless. Therefore, these instructions were added manually for example, STP instruction was placed at the end of the program and right after jump instructions to test whether the jump instructions operate correctly. ADM and SBM instructions were paid more attention to as well to test their functionalities by making sure they are provided memory locations where non-zero data is stored.

6. Evaluation

6.1 Correctness

The correctness of the CPU was tested by converting the benchmark algorithms to assembly code and running them on a set of data. The benchmarks were simulated in three different ways: by hand, through Quartus and on the simulator. The results of each simulation are listed in the relevant section below. In addition, all simulator outputs and Quartus waveforms are linked in the appendix. The exception being the Quartus waveform of fib(10) as it was too long to practically store. Do note that the simulator outputs will vary slightly from the waveforms of the Fibonacci and LCG benchmarks since the initial loading of values into registers was done through LDI and shifts rather than LDA. However, the algorithms were not changed and should yield the same results. In addition to the given algorithms the CPU was tested on the previously mentioned 2000 instruction program.

6.1.1 Fibonacci

Below, the Fibonacci benchmark algorithm from the initial specification is shown on the left, while the assembly code equivalent is written in mnemonic form on the right. The assembly code is explained beneath the table.

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

Instructions:

0: LDA R0 = MEM[0x000]

1: LDI R3 = 0x006

2: LDI R2 = 0x001

3: JMR PC = R3 if R2 < R0

4: LDI R1 = 1

5: JMP 0x010

6: STK PSH R0 - 2

7: STK PSH PC + 3

8: SBI R0 - 1

9: JMP PC = 0x003

10: STK POP R0

11: STK PSH R1

12: STK PSH PC + 2

13: JMP PC = 0x002

14: STK POP R2

	15: ADR R1 = R1 + R2 16: STK POP PC 17: STA MEM[0x001] = R1 18: STP Memory data: 0x000 = n

Explanation of code:

0: The value n is stored in register R0.

1, 2: R3 and R2 are loaded with the jump location (0x0006) and a constant (0x0001) used by the JMR instruction.

3: The JMR instruction implements the if-else statement. If the value of R0 (n) is more than R2 (1) the program jumps to R3 (0x006), which is the "else" part of the statement. Otherwise the value of n is <= 1 and the program enters the "if" section.

4, 5: In the "if" part the value of R1 (y) is set to 1. The program then jumps to the STK POP instruction at 16.

In the "else" part the function recursively calls itself.

6, 7: The program stores all relevant values for resumption of execution to the stack. This being the desired value of n (n - 2) and where to continue the function from PC + 3. Notice that it is not necessary to store the value of R1 (y) at this point as R1 so far has not been written to.

8, 9: 1 is subtracted from n and the program jumps to the JMR instruction (if-else statement). This is equivalent to calling the function with the parameter n – 1.

Instructions 6 to 9 will happen recursively until n <= 1. Instructions 4 and 5 will then be executed and the program will jump to 16, where stacked function calls will be resumed.

10: The program "POPs" the stacked value of n into R0, resuming the function call.

11, 12: The program stores the current values of R1 (y) and were to pick up execution, to the stack.

13: The program jumps to 2, which now equals calling fib(n-2). Notice that it is not necessary to subtract 2 from the current value of R0 (n) as this was done previously with an immediate offset when R0 was pushed to the stack.

14, 15: When the fib(n-2) function call is over, the stacked value of y (fib(n-1)) is stored to R2 and is then added to the return value of fib(n-2) stored in R1.

16: The program starts emptying the stack, resuming any unfinished function calls. If the stack is empty the instruction does nothing and continues on to the STA instruction.

17: The final value of y (R1) is stored to MEM[0x001]

18: The program has reached its end and stops.

The simulation results of the benchmark are shown in the table below. The corresponding simulator outputs and Quartus waveforms are found in appendix B.

n	Paper analysis	Quartus output	Simulator output
0	1	1	1
1	1	1	1
2	2	2	2
3	3	3	3
4	5	5	5
5	8	8	8
8	34	34	34
10	89	89	89

Figure 10: Simulation results of the Fibonacci algorithm for different input values.

The table shows that the results from paper analysis and the simulator are identical to the results of the Quartus simulations. The highest input value tested in Quartus was n = 10. The CPU can in theory calculate up to fib(129) before the stack overflows. Though possible, the inefficient nature of the function makes it unfeasible to test input values much higher than 10. Still, the tests performed provide sufficient basis for concluding that the assembly code conversion of the Fibonacci algorithm is working to specification.

Note that the benchmark algorithm as given in the specification does not actually return the nth Fibonacci number, but rather the (n+1)th. For example, the 0_{th} Fibonacci number is 0 and not 1.

6.1.2 Linear Congruential Generator

Below, the benchmark algorithm is shown on the left and the assembly code is shown in mnemonic form on the right. For each test the value of n was varied while the other parameters were kept constant at a = 25385, b = 3 and s = 43861.

	Instructions: 0: LDA R2 = MEM[0x000] 1: LDA R1 = MEM[0x003]

```
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 th
         sum = sum + y // add it to
      }
      return sum;
}
```

2: LDA R0 = MEM[0x002]

3: JEQ PC = 0x009 if R0 == 0

4: MLR R1 = R1 * R2

5: ADM R1 = R1 + MEM[0x001]

6: ADR R3 = R3 + R1

7: SBI R0 = R0 - 1

8: JNQ PC = 0x004 if R0 != 0

9: STA MEM[0x004] = R3

10: STP

Memory data:

0x000 = a

0x001 = b

0x002 = n

0x003 = s

0x004 = sum -> initialised to 0

Explanation of code:

0: The value of a is stored in R2.

1: The initial value of y is stored into R1.

2: The initial value of n/i is stored in R0.

3: If R0 = 0, then the for loop should not be run and the program jumps to 9.

4, 5: Multiply R1 = R1 * R2 (y = y * a), then b is added to the product.

6: The current value of R1 (y) is added to R3 (sum).

7: The value of i is decremented by one.

8: If i ≠ 0 the program jumps back to the top of the for loop, otherwise it continues on to the STA instruction.

9: The final value of R3 (sum) is stored to MEM[0x004].

10: The program is finished and stops.

Note that even though the MLR instruction has an immediate offset option, it was determined that using the ADM instruction to add b was more efficient than using the immediate offset, as otherwise “sum” would have to be stored in memory.

The result of each test is shown in the table below. The corresponding simulator outputs and Quartus waveforms are found in appendix B.

n	Paper analysis	Quartus output	Simulator output
0	0x0000	0x0000	0x0000
1	0x4FA0	0x4FA0	0x4FA0
2	0xF043	0xF043	0xF043
3	0xB361	0xB361	0xB361
5	0x42AE	0x42AE	0x42AE
8	0xFC14	0xFC14	0xFC14
13	0x4E5A	0x4E5A	0x4E5A
21	0x1EC6	0x1EC6	0x1EC6

Figure 11: Simulation results of the LCG algorithm for different input values.

The three test types once again returned identical results. The highest value of n tested was 21, as it is impractical to do paper analysis of the output for any n value much larger than this. From the results it can be concluded that the assembly code conversion of the LCG algorithm is working to specification.

6.1.3 Linked List

Below, the benchmark algorithm is shown on the left and the assembly code is shown in mnemonic form on the right. For the assembly code implementation of the benchmark a null pointer was interpreted to mean an uninitialised pointer. That is a pointer with the value 0x0000. The “item” struct in the benchmark was implemented as two consecutive memory words, where the first word held the integer value of the node and the second word pointed to the first word of the next node in the linked list. For each test the node holding the value x was varied.

	Instructions: 0: LDA R1 = MEM[0x000] 1: LDA R0 = MEM[0x001] 2: LDI R3 = 0x007 3: LDR R2 = MEM[R0] 4: JMR PC = R3 if R1 == R2 5: LDR R0 = MEM[R0 + 1] 6: JNQ PC = 3 if R0 $\neq$ 0 7: STA MEM[0x001] = R0 8: STP

```
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;
}
```

Memory data:

0x000 = x

0x001 = *head

Explanation of code:

0: The value of x is loaded into R1.

1: The value of *head is loaded into R0.

2: The jump location used in the JMR instruction is loaded into R3.

3: The value that *head is pointing to is loaded into R2.

4: If the value that *head is pointing at is equal to x the while-loop is never entered and the program jumps directly to 7.

5: Sets the value R0 (*head) to the “next” pointer of the “item” it is pointing to.

6: If R0 ≠ 0, meaning the head pointer is not a null pointer, jump to the top of the while loop at 3.

7: Stores the value of *head in memory location [0x002].

8: The program is finished and stops

Note that the linked list algorithm from the specification does not work when the *head parameter is a null pointer. As such, the code does not work if the given linked list has length 0. The same is true for the assembly code implementation of the function.

Before testing the program, a linked list of “items” had to be created in the memory. The structure of the linked list is shown in figure 12. Each node has a memory address, an integer value and points to a new memory address. The first node has the memory address 0x010, an integer value not currently specified, and points to the memory address 0x015. For the program the values held by each node is unimportant as long as

they do not hold the value x. When testing the value of x was assigned to one of the given nodes at a time. In the tests x was set to 5 and *head was given the value 0x0010.

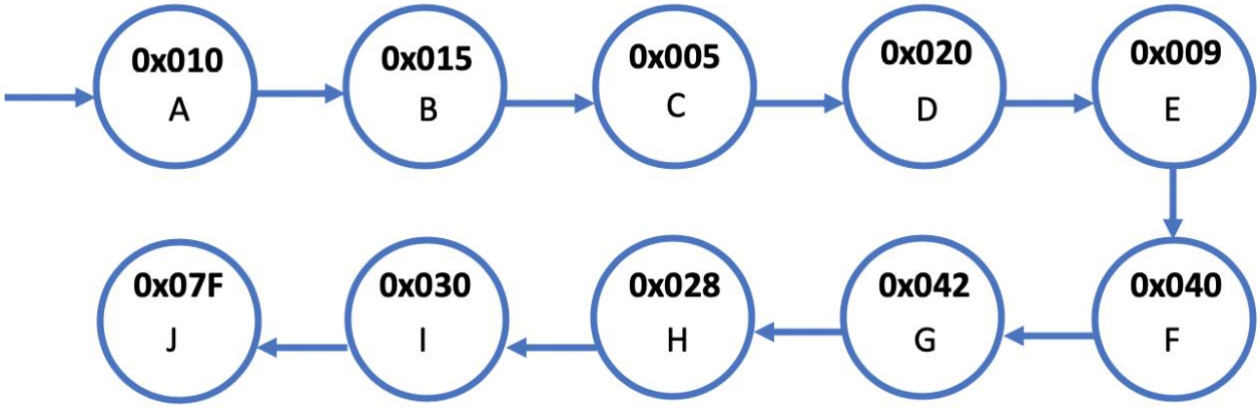


Figure 12: Diagram showing the structure of the linked list. Each node is shown with its memory location and an arrow showing which memory location the "next" pointer points to. The value of the node is omitted.

The result of each test is shown in the table below. The corresponding simulator outputs and Quartus waveforms are found in appendix B.

Node with value x	Paper analysis	Quartus output	Simulator output
No node with x	0x0000	0x0000	0x0000
A	0x0010	0x0010	0x0010
B	0x0015	0x0015	0x0015
C	0x0005	0x0005	0x0005
F	0x0040	0x0040	0x0040
G	0x0042	0x0042	0x0042
H	0x0028	0x0028	0x0028
I	0x0030	0x0030	0x0030
J	0x007F	0x007F	0x007F

Figure 13: Simulation results of the linked list algorithm for different input values.

Similar to the two previous benchmarks, there were no discrepancies between the test results. As such, it can be concluded that the assembly code conversion of the linked list algorithm works to specification, even for the corner case when x is not stored in any node.

6.1.4 Individual Test and 2000 Instruction Program

The above algorithms are all helpful in testing the correctness of the CPU. However, they do not test every instruction in the ISA and their success alone is insufficient to determine whether the CPU is operating correctly. As such, the CPU was tested on the random 2000 instruction program. The Quartus waveform was then compared to the output of the C++ implemented simulator. No discrepancies between the two outputs were found. In addition to the 2000 instruction program every instruction was tested separately and all were found to work to specification.

The success of the three benchmark algorithms confirm that they were correctly implemented in assembly code. When seen in conjunction with the success of the individual tests and the 2000 instruction program, it can with reasonable certainty be concluded that the CPU itself is working as intended.

6.2 Speed

Key figures:

- Maximum clock frequency: 69.5 MHz
- Operating clock frequency: 65.94 MHz
- Benchmark performance:
    - fib(5):
        - Clock cycles: 130
        - Execution time: 1.97 us
    - lcong(8)
        - Clock cycles: 66
        - Execution time: 1.00 us
    - find(10)
        - Clock cycles: 65
        - Execution time: 0.99 us

As previously discussed, the CPU has a fairly low clock speed. The initial simulations showed that the CPU had a maximum clock frequency of 42.85 MHz. Luckily, this figure was later increased to 69.5 MHz through logic optimisation and by pipelining the multiplier. However, it is not optimal to run the CPU at this frequency as doing so adversely affects the CPU's power and area usage. The clock frequency was

therefore limited to 65.94 MHz. Analysis of the slowest path shows that the ALU is the main limiter of the clock speed, having a latency of 10.278 ns. It is not unexpected that the ALU has a high latency. The ISA has several complex instructions and in the worst case an ALU signal would have to pass through three multiplexers, a shifter and two adders, all in one clock cycle. The clock frequency is however not the only factor in a CPU's speed, how many clock cycles needed to complete a program must also be considered. To explore this aspect, the benchmark algorithms were simulated with varying input values and the cycles needed for completion were counted. The results are presented below. The waveforms corresponding to the simulations are linked in appendix B and timing analysis results are linked in appendix C.

6.2.1 Fibonacci

By design, the Fibonacci algorithm is very inefficient. It is written in such a way that the number of function calls is approximately an exponential function of the input value. Thus, the number of clock cycles required to complete the program will also be exponentially dependent upon the input value. The relationship between input value "n" and the number of cycles needed to complete the program is shown in figures 14 and 15.

n	Clock cycles for completion
0	11
1	11
2	28
3	45
4	79
5	130
8	572
10	1507

Figure 15: Table showing the relationship between the input value "n" and number of clock cycles needed to complete the program.

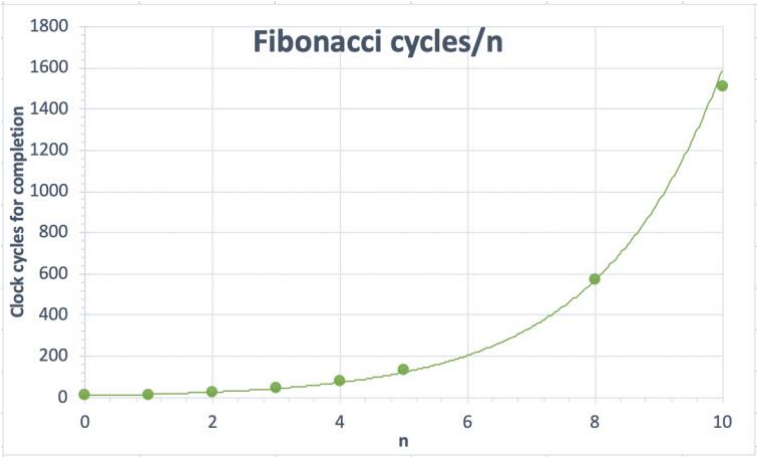


Figure 14: Plot showing the number of cycles needed for execution versus the input value, with an exponential trend line.

The 7th row of figure 15 shows that the CPU needs 130 clock cycles to compute fib(5). With a clock speed of 65.94 MHz this corresponds to an execution time of 1.97 us. The low performance is partly due to the limited clock frequency, but equally important are the design choices that were made in regard to the stack implementation. In the current ISA, any recursive function call of the fib-function requires two stack push instructions and two stack pop instructions. This could have been reduced to one push instruction and one pop instruction by letting the stack act on all registers at the same time. The number of clock cycles needed to compute fib(5) would then be reduced to 102, which would take 1.55 us to execute. This would however drastically lower the maximum

stack depth. In the CPU's current form it is only necessary to store two values for each recursive call of the fib() function: R0/R1 and PC. With 256 memory words in the stack this allows for theoretically computing values as high as fib(129). If every recursive function call instead required the storing of all register values and the carry bit, it would theoretically only be possible to calculate up to fib(43). This is not very important for the given algorithm as calculating fib(129) would take millions of years, but for algorithms in general, trading of memory for speed might be worth it. That being said, the decision on how to implement the stack was taken before the CPU's low clock speed was known, and it is not unlikely that the stack would have been implemented differently in successive iteration of the CPU and its ISA.

6.2.2 Linear Congruential Generator

In comparison to the Fibonacci benchmark algorithm, the LCG benchmark algorithm is far more efficient. It has a linear relationship between the input value and the number of cycles needed for completion given by *Number of Cycles = 10 + 7*n*. This is illustrated below in figures 16 and 17.

n	Clock cycles for completion
0	10
1	17
2	24
3	31
5	45
8	66
13	101
21	157

Figure 17: Table showing the relationship between the input value "n" and number of clock cycles needed to complete the LCG algorithm.

Figure 16: Plot showing the number of cycles needed for execution versus the input value, with a linear trend line

As the 7th row of figure 17 shows that lcong(8) takes 66 clock cycles to complete. This corresponds to an execution time of 1.00 us. The performance of the CPU running the lcong function is far better than when it is running the fib function. However, as mentioned earlier, the program would have functioned even better if 5 registers were available. In that case the ADM instruction would become redundant and each iteration of the for loop would take two cycles less. Computing lcong(8) would then only take 50 clock cycles and have an execution time of 0.76 us. The choice to use 4 registers was supported by the fact that it allows for more powerful instructions as well as making it possible to load and store values using direct addressing. However, as a consequence of the low clock speed, less powerful instructions might give a more powerful CPU

overall. With instructions that do less, the ALU could be significantly simplified, increasing the clock speed and potentially increasing the amount of work done per unit time, even with weaker instructions. Even though the performance of the LCG benchmark is good, different design choices such has having more registers could have led to better performance.

6.2.3 Linked List

Like the LCG algorithm, the linked list algorithm has a linear complexity. The number of clock cycles needed to complete the program is equal to 5 + 6*n where n is the index of the node holding the value x. For example, if x was the value of the third node then n would be 3. If the value that is being searched for is not contained in any of the nodes, then the program takes 8 + 6*linkedList.size() cycles to complete. For the values below the algorithm was given a linked list of length 10.

Node with x	Clock cycles for completion
1	11
2	17
3	23
6	41
7	47
8	53
9	59
10	65
No node with x	68

Figure 18: Table showing the relationship between the number of nodes traversed and the number of clock cycles needed to complete the linked list algorithm.

Figure 19: Plot showing the number of cycles needed for execution versus the input value, with a linear trend line.

The linked list benchmark is executed the most effectively by far. When the value being looked for is in the 10th node the program needs 65 cycles to complete, which corresponds to an execution time of 0.99 us. Unlike the other benchmarks the find() algorithm does not suffer from any shortcomings with the CPU. As such, there is very little which could be done to improve the algorithms performance other than increasing the clock speed.

Speed is not the CPU's biggest strength. It generally suffers from a low maximum clock frequency and there are design choices which would have increased the CPU's performance in both the Fibonacci and LCG benchmark. Though there are optimisations which could be done, the CPU's speed is not unreasonably bad.

6.3 Power Consumption

Key figures:

- Total Logic Elements: 1479 / 6272 (24 %)
- Core Dynamic Thermal Power Dissipation: 13.07 mW
- Core Static Thermal Power Dissipation: 43.12 mW

In addition to the total logic elements the circuit has two RAM-blocks, each with 2048 words and a word length of 16 bits.

The CPU has a maximum clock frequency of 69.5 MHz. However, running the design at this frequency is inefficient as it requires 3296 logic elements and gives a dynamic power of 19.96 mW. By lowering the operating frequency to 65.94 MHz the area and power requirements are significantly reduced to 1479 logic elements and a dynamic power of 13.07 mW. There is no considerable change in static power between the two frequencies. The analysis results at both frequencies are linked in Appendix C. Even though the CPU's low clock speed is its greatest weakness, the benefit of a 34.5 % decrease in dynamic power and a 55.1% decrease in area greatly outweighs the disadvantage of a 5.1% decrease in clock speed. Thus, the clock frequency was set to 65.94 MHz in the final design.

Static power dominates the CPU's power usage. This is mainly due to the CPU design using very few of the FPGA's available logic elements, and it would likely have been more power efficient to implement the CPU on a smaller FPGA. The static power consumption is mostly invariant to the number of logic devices implemented on it, and even without any logic elements the static power was measured to be 42mW (See Appendix C). Little can therefore be done to optimise the CPU's static power consumption.

As dynamic power accounts for less than ¼ of the CPU's total power consumption, its value holds little importance for the CPU's power efficiency. However, the dynamic power may be a good indicator of how well the CPU is utilising its hardware [22]. To get a point of reference on this matter the MU0-armish CPU from the DECA lab was compiled and tested on Cyclone IV with the same target frequency. Simulation showed that the MU0-armish CPU has a similar power consumption, with a dynamic power of 9.33 mW and a static power of 42.97 mW. Even though the MU0-armish CPU only has 309 logic elements and is far simpler, this is 3.89 mW less than the designed CPU. As such, it can be concluded that the new CPU is very power efficient. However, it might also mean that the MU0-armish CPU is better at utilising its hardware. A breakdown of

the power consumption reveals that the RAM consumes 5.38 mW of dynamic power in the MU0-armish CPU and that the two RAM blocks in the new CPU collectively consume 4.13 mW of dynamic power. That leaves 3.95 mW and 8.94 mW of dynamic power for the logic elements in each of the respective circuits. Calculations showed that the MU0-armish CPU has a dynamic power consumption per logic element 2.1 times higher than that of the new CPU. It can therefore be concluded that the new CPU does not use its hardware as effectively as the MU0-armish CPU, matching expectations. The MU0-armish CPU is very simple, having close to the minimum amount of components needed in a CPU. For example, its ALU is only able to do addition, subtraction and shifts by 1. When that is compared to a CPU which is able to addition, subtraction, shifts by up to 15 bits in each direction, multiplication and a wide range of bitwise operations, it is obvious that the MU0-armish architecture at any time will have a smaller part of its logic going unused. Thus, despite the new CPU having a lower degree of hardware utilisation than the MU0-armish CPU, these values actually suggest that the CPU is fairly good at using its hardware.

The CPU design only uses 24% of the available logic elements on the FPGA. Meaning that it would have been possible to put far more logic devices into the CPU's design. However, it is not a goal to use more area than what is necessary. Moreover, using more area inevitably means a higher power consumption, which is unwanted. On the other hand, it was shown that using more of the FPGA's area would allow the CPU to operate at higher clock frequencies. However, this theoretical clock speed increase would likely have little impact on the performance of the CPU in the real world. As a consequence of the higher dynamic power, the FPGA will likely run hotter when it is operating at 69.5 MHz than when it is operating at 65.94 MHz. As Quartus simulations show, a CPU operating at high temperature will have a lower maximum clock frequency than the same CPU operating at a lower temperature. Thus, in the real world it is not unlikely that the CPU implementation having a max frequency at 85 ºC of 65.94 MHz would be able to run at a higher average frequency than the CPU implementation having a max frequency at 85 ºC of 69.5 MHz.

It is clear that neither the power consumption, nor the required area are weaknesses of the CPU. As the CPU only requires 3.89 mW more power than the very simple MU0-armish architecture, power is in fact one of the CPU's greatest strengths. Moreover, though the CPU does not utilise its hardware as well as the MU0-armish CPU, it is still fairly good at using its hardware. Finally, the CPU's area is not too large and if anything, a bit small. However, there is no guarantee that using the free area on the FPGA will make the CPU run faster in the real world. Even if it did, the decrease in power consumption from lowering the max clock frequency to 65.94 MHz outweighs the potentially small increase in speed improvements from running the CPU at 69.5 MHz.

7. Conclusion

The CPU's biggest weakness is its low maximum clock frequency. Moreover, changing the number of registers and the stack implementation could make the Fibonacci and LCG algorithms more efficient. However, such changes would come at the expense of the maximum stack depth, direct addressed loading and storing, and the immediate offset in instructions such as ADR, SBR and MLR. The linked list algorithm is working optimally and would benefit little from any changes besides increasing the maximum clock frequency. One of the CPU's many strengths is its low power usage, which is on par with far simpler CPUs. The CPU also uses few transistors and could fit on FPGA devices considerably smaller than Cyclone IV. Lastly, extensive testing shows that the CPU is in fact functioning as intended.

7.1 Future Work

Though the CPU is currently working to specification, there are still several planned, performance enhancing features, which due to time constraints will not be implemented before the project delivery. Three of the more significant are listed below.

7.1.1 Combining designs

Working remotely, simultaneously in Quartus is challenging as there is a chance of corrupting files or losing progress when merging files. To avoid the need for mergers the CPU was given a modular design with the ALU separated out as its own block, and several small decoders rather than one big. Though this simplified the process of implementing the CPU, it does require more gates than if the decoders were combined and if the ALU could have more shared resources, such as multiplexers, with the top-level design. Moreover, it is not optimal to have clocked logic, such as that used for the pipelining of the multiplier, inside the ALU itself. With more time the ALU logic would therefore have been merged into the CPU, rather than having it as its separate block. These changes are not vast in extent and would with more time have been implemented easily.

7.1.2 Dynamic pipeline length

Currently the CPU has a pipeline with a fixed length of two. As 16 out of the 21 instructions in the ISA are two cycle instructions, the pipeline is efficient for most programs. However, if a program is very dependent upon three cycle instructions the pipeline in its current form is not optimal. The original hardware design therefore

included a dynamic length pipeline, where successive three cycle instructions would be pipelined with a pipeline length of three. The pipeline would then revert to having a length of two at the next two cycle instruction. (See figure 20.) In the LCG implementation given earlier there are two successive three cycle instruction, and this pipeline implementation would speed up the program execution by 2 + n cycles, where n is the number of iterations of the for loop. This is significant, but for programs with an even greater dependence on three cycle instructions the execution time can at best be halved. This pipelining would be done by giving dual outputs to the RAM holding the instruction, such that both the current and next instruction are outputted at the same time. It would then be possible to pipeline the FETCH of the next instruction with EXEC1 of the current instruction. For successive LDA, ADM and SBM instruction there is no dependency between EXEC1 of the next instruction and EXEC2 of the current instruction and they can be pipelined without any significant added complexity. For an MLR instruction following and LDA, ADM and SBM instruction however, EXEC1 might be dependent on the previous instruction's EXEC2. It is then necessary to use operand forwarding to achieve the full three cycle pipeline length. This can easily be done through multiplexing the input signal of the multiplier.

Cycle number	0	1	2	3	4	5	6	7
ADR	FETCH	EXEC1						
ADR		FETCH	EXEC1					
LDA			FETCH	EXEC1	EXEC2			
LDA				FETCH	EXEC1	EXEC2		
LDA					FETCH	EXEC1	EXEC2	
ADR							FETCH	EXEC1

Figure 20: Illustration of variable pipeline length. cycles being executed at the same time are in the same column.

7.1.3 Simulator: Graphical User Interface

For the simulation part, creating a Graphical User Interface to the simulator would be worth considering as a possible improvement. This would be valuable as it would significantly enhance the usability of the simulator. At the moment in order to use the simulator, the user is asked to create a text file containing instructions and run the main file with the created text file as an input. This way of using the simulator may not be user-friendly as it can be difficult and unfamiliar for general users.

8. References

References

[1] A. Dasdan, "Twelve Simple Algorithms to Compute Fibonacci Numbers," 2018. Available: https://www.openaire.eu/search/publication?articleId=od_____18::3f83c67b2b1c638e7f7ced48da8fee95.

[2] (October). *How to Write a Software Requirements Specification (SRS Document)*. Available: https://www.perforce.com/blog/alm/how-write-software-requirements-specification-srs-document.

[3] (July). *Software Requirements Specification document with example*. Available: https://krazytech.com/projects/sample-software-requirements-specificationsrs-report-airline-database.

[4] (May). *WRITING A PRODUCT DESIGN SPECIFICATION*. Available: https://www.jensen-consulting.co.uk/2012/05/16/writing-a-product-design-specification/.

[5] (February 11-13,). *Manifesto for Agile Software Development*. Available: http://agilemanifesto.org/.

[6] (July 14th). *What Exactly Is GitHub Anyway?*. Available: https://techcrunch.com/2012/07/14/what-exactly-is-github-anyway/?guccounter=1.

[7] (). *Notion*. Available: https://www.notion.so/.

[8] R. Rojas, "Conditional Branching is not Necessary for Universal Computation in von Neumann Computers," 1996. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.757&rep=rep1&type=pdf.

[9] D. S. Page, "A practical introduction to computer architecture," in Anonymous London: Springer, 2009, pp. 271.

[10] A. C. Davies and Y. T. Fung, "Microprocessors," in , 7th ed.Anonymous Elsevier B.V., 1977, pp. 425-432.

[11] K. Abbas, "Handbook of digital CMOS technology, circuits, and systems," in Anonymous Cham: Springer, 2020, pp. 456-464.

[12] W. J. Townsend, J. Swartzlander Earl E and J. A. Abraham, "A comparison of dadda and wallace multiplier delays," in Dec 31, 2003, pp. 552-560.

[13] D. Sony, "Comparison of Wallace, Vedic and Dadda Multipliers," 2018. Available: http://www.ijaerd.com/papers/finished_papers/Comparison%20of%20Wallace,%20Vedic%20and%20Dadda%20Multipliers-IJAERDV05I0370711.pdf.

[14] (n.d.). *Recursion*. Available: https://www.khanacademy.org/computing/computer-science/algorithms/recursive-algorithms/a/recursion.

[15] P. B. Russell and Hewlett-Packard, "Exploring a Stack Architecture," .

[16] P. R. Wilson *et al*, "Dynamic Storage Allocation: A Survey and Critical Review ," .

[17] (19 Nov). *x86 Assembly Guide* . Available: https://www.cs.virginia.edu/~evans/cs216/guides/x86.html.

[18] (n.d.). *POP*. Available: http://www.keil.com/support/man/docs/armasm/armasm_dom1361289885303.htm.

[19] Randal E. Bryant and David R. O'Hallaron, "CS:APP2e Web Aside ASM:X87: X87-Based Support for Floating Point," June 5th, 2012.

[20] (Feb). *Lecture 6 – making CPUs faster*. Available: https://intranet.ee.ic.ac.uk/t.clarke/arch/deca/lecs/L6.pdf.

[21] (). *Lecture 19, Pipelining Data Forwarding*. Available: https://www.csee.umbc.edu/~squire/cs411_l19.html.

[22] (June). *Figures of Merit for CPU*. Available: https://piazza.com/class/k9n8clkdzsz3nk?cid=123.

Appendices

Appendix A: Instruction Set Documentation

Detailed ISA Documentation

Appendix B: Outputs and Waveforms

Fibonacci Algorithm

LCG Algorithm

Linked List Algorithm

Appendix C: Timing and Power Results

Slowest path of CPU

CPU at 69.5 MHz

CPU at 65.94 MHz

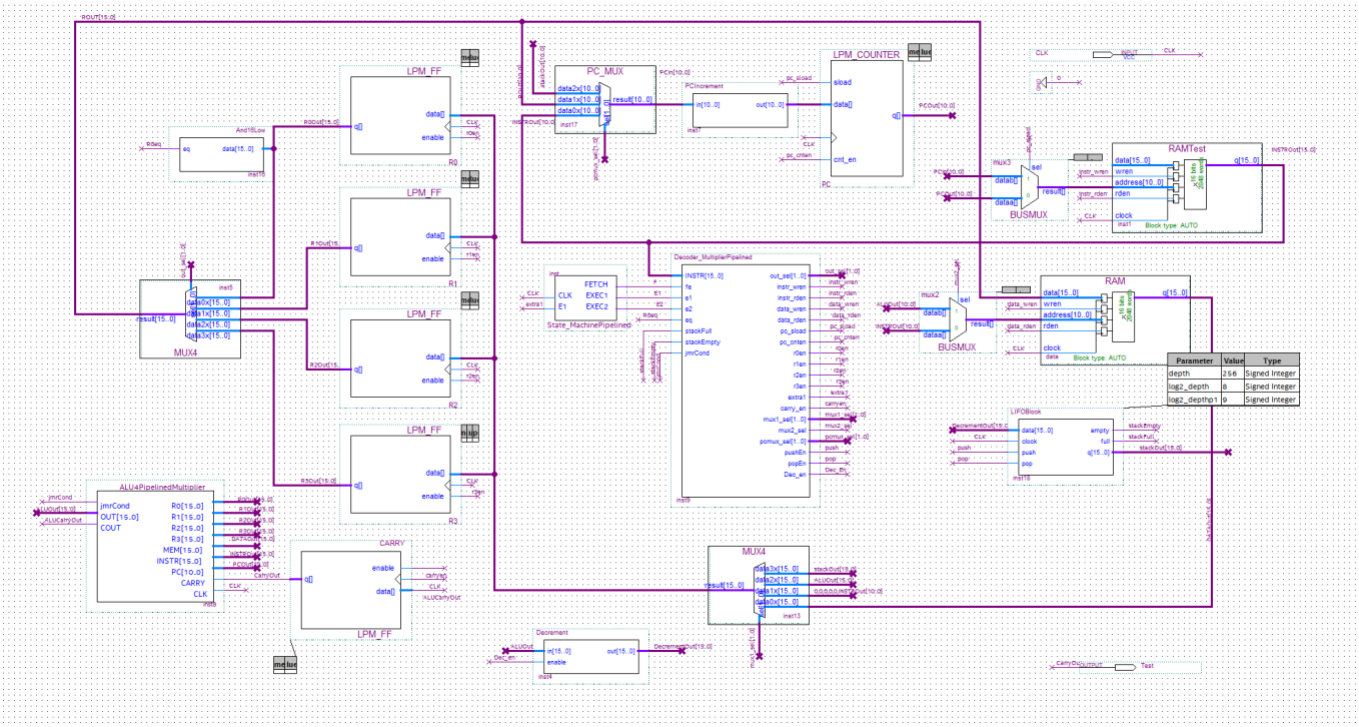
MU0-armish CPU

Cyclone IV FPGA Without Logic

Appendix D: CPU and ALU layout

Higher Resolution Schematics are available here.

Overview of the CPU layout:



Overview of the ALU layout: