Imperial College London

Central Processing Unit

ELEC40006: 1st Year Electronics Design Project 2020

Bradley Stanley-Clamp	01545990	Electronic and Information Engineering 1st year
Ebby Samson	01737449	Electronic and Information Engineering 1st year
Riccardo El Hassanin	01729427	Electrical and Electronic Engineering 1st year

Dr. Edward Stott
Mrs. Esther Perea

Date of submission: 14/06/2020

Word Count: 10,335

Table of contents

Abstract
The objective of this project was to design a general-purpose CPU using Quartus Prime that is able to execute three algorithms: calculating Fibonacci numbers, calculating pseudo-random integers using a linear congruential generator and traversing a linked list to find an item [1]. The CPU was based on a Harvard architecture design and in order to execute these tasks, the product needs to perform recursion and multiplication. For recursion, a stack was implemented using a register file. Several methods of multiplying were discussed such as parallel and sequential multipliers as well as look up tables. Dual port RAM was used to improve the speed and efficiency of traversing a linked list. In order to make the product more general purpose, register addressing, and computed jump were added, as well as bitwise operations. Another objective was to increase the maximum frequency and decrease execution times of benchmark algorithms. This was done by shortening the longest delays in the system, later in the process. This resulted in a mean maximum clock frequency of 90.08MHz and a minimized geometric mean time of 0.836μs. The design also features a stack with 32 stack frames, each capable of storing an address and the value of register R0. The addition of the stack resulted in a large resource usage and power dissipation of 129.32 mW.

1. Introduction

The goal of this project is to design a general-purpose CPU with an instruction set architecture that is optimised to execute three algorithms efficiently: calculating Fibonacci numbers, calculating pseudo-random integers using a linear congruential generator and traversing a linked list to find an item. This must be done using block schematics and Verilog and simulated using Quartus Prime [1] and compiled for the Cyclone IV E FPGA.

2. Specification

Below is the Product Design Specification:

2.1 Performance

The CPU must successfully run three algorithms: calculating Fibonacci numbers via the process of recursion using a stack, generating pseudo-random integers using a linear congruential generator and finding the address of an item in a linked list [1]. The aim is to also reduce geometric mean time '$(T_1T_2T_3)^{1/3}$' [1, p4], where T_n (n=1..3) are the times of each algorithm. The power consumption, which relates to the total number of transistors used, should also be minimised. The CPU should solve these few commonly occurring computing problems and be general purpose [1].

2.2 Sizes

The CPU must use 16-bit integers. It should also have at least enough memory for 2048 words of instructions and 2048 words of data. The memory can be a single unified memory for instructions and data, or separate memories for instructions and data. Hence the memory units require addresses 12 bits wide if using one memory, or 11 bits wide if using separate memories [1]. The number of registers and logic blocks used should also be minimised.

2.3 Process

The CPU design will be designed using block schematics and Verilog modules, and then simulated using Quartus Prime. However, the design is not permitted to contain pre-existing multiplier blocks or the Verilog multiplication sign[1].

2.4 Testing

The CPU will be tested to ensure it can run the three algorithms. They will be tested using the typical values provided with the C++ algorithms (Appendix A). The CPU will also be tested for speed to find minimum execution times as well as geometric mean time '$(T_1T_2T_3)^{1/3}$' [1, p4], where T_n (n=1..3) are the times of each algorithm. It will also be tested for power consumption via the number of logic gates within the CPU and the clock speed of the CPU[1]. The maximum clock speed will be determined using Quartus's built-in timing analysis feature. The power consumption will be determined using Quartus's Power analysis feature.

2.5 Company constraints

The CPU is to be designed in Quartus Prime. The recursive implementation of the Fibonacci function requires a stack, alternative implementations are not permitted. Fixed hardware constructs cannot be used; hence Verilog IP blocks and Verilog multiply operators cannot be used to implement multipliers in the CPU[1].

2.6 Timescale

This is a 5-week project that has been given a deadline of 14th June 2020[1].

2.7 Documentation

A report and a video are required, the report must be 10,000, words with a 10% margin and cover all work done on the project [2]. The video must be less than 5 minutes and cover the final design and outcomes of the project [2]. In addition, the files of the project are also required for plagiarism checks [1]. It is also required to find the optimal values for the linear congruential generator that will achieve the longest possible sequence.

3. Project Planning

A Gantt chart was used as a guide during the project to ensure correct time management, it is attached in Appendix B. The plan is to start by finding more information on Von Neumann and Harvard architectures, while simultaneously using design criteria to create an ISA. Once completed, the next step is to choose and implement an architecture in Quartus, while finalising the ISA and designing possible implementations for its instructions. After the architecture is completed, the best implementations for the ISA need to be identified and employed. The final steps are then to remove bugs, evaluate the project, create the report and record the video.

The team planned to have online meetings once every 5 days for the first 15 days of the project. Then a meeting every 3 days up until the deadline. These meeting ranged between 15 minutes to 2.5 hours depending on what was discussed and worked on during the meeting. The subject of the meeting was known before so ensure that no time was wasted. In addition, a group chat was used to communicate.

4. CPU Design ideas

4.1 Architecture

Initially, the MU0 CPU was deemed as a good starting point due to its simple design, however, its lack of multiple registers made it very slow due to its dependence on slow memory operations. The ARMish CPU was also considered because of its multiple registers and easily pipelined ARMish instructions providing a faster route.

Next, an architecture has to be chosen between the Von-Newman architecture, where a single memory holds data and instructions, to the Harvard architecture, where separate memories are used for data and instructions. The latter permits data writes and instruction reads to occur simultaneously, which allows all instructions to be pipelined [3].

The memory word size should be 16 bits because the CPU must use integers that are 16 bits wide. So, for a linked list, two words in memory would be needed to store each node. In addition, If Harvard architecture is used addresses would only have to be 11 bits wide, allowing an extra bit for the opcode if necessary.

4.2 Task 1

Task 1 requires that a stack is used for recursion. This requires a form of memory to store the current value of the program counter when a subroutine is called [4]. When the called subroutine has finished, the CPU will jump back to the previous location in the program using the stored address. The memory block can either be in RAM or a dedicated register file. If data memory were used, the stack could have a larger maximum recursion depth but at the cost of speed. This is because the register file can constantly point to the top of the stack, whereas memory would need an extra cycle to read. A stack pointer is used to point to the top of the stack, so the CPU returns to the correct part of the program after recursion [4]. The register file is the better option here because higher speed would better fulfil the design criteria.

Inspired from the 4004 CPU design [4]; for the stack described above, two instructions need to be added to the ISA; JMS and BBL [4]. Firstly, JMS will jump to a subroutine and store the previous value of the PC on the stack. Consequently, when BBL is called the program will branch back to the previous location in the program by popping a value off the stack.

4.3 Task 2

Task 2 requires the implementation of: 'x_{n+1} = (ax_n + b) mod 2^N' [1, p3]. The modulus operation was ignored as *N* = 16 and the data bus in this CPU is 16-bit, therefore the bits of power larger than 2^{15} are cut off [5]. But multiplication and addition both need to be implemented. Addition can be easily done via an adder, whereas for multiplication of two binary numbers there are several possible implementations. These implementations are based on different algorithms.

The first algorithm inspired by information given at [6], where the multiplier is made up of a series of additions, the product is formed by adding the multiplicand to itself (n-1) times, where n is the value of the multiplier. However, before discussing the hardware implementations of this algorithm it is clear this is a very inefficient route. Taking this algorithm to extreme conditions; this CPU uses 16-bit busses therefore if the multiplicand had a value of 1_{10} and the multiplier had large value that would still produce a 16-bit solution such as 2^{16}-1 = 65535_{10}. To obtain the product the CPU would have to do 65534 additions, which using any hardware will take a significantly long time. An improvement to this algorithm could be to make the smaller value the multiplier and the lager value the multiplicand which would work for the extrema cases however for cases where the operands are similar value such as 255_{10} and 250_{10}, which still produce a product within the 16-bits. Whichever way around the multiplier and multiplicand are identified, there is still a very large amount of additions needing to take place. According to design criteria, execution time should be reduced, therefore this algorithm can be disregarded.

Using the information gained from [7], the second algorithm is made up of a series of 'add' and 'logical shift' operations and uses the same process as on paper multiplication. The LSB of the multiplier is individually multiplied with each of the bits of the multiplicand forming a partial product. This is then repeated for the successive bits of the multiplier. Once all the bits of the multiplier have been used to form individual partial products. All the partial products are added together, however, each successive partial product is higher power than the

previous. For example, the 0th bit of the second partial product will be added to the 1st bit of the first partial product and so on [7], as can be seen in Figure 4.1.

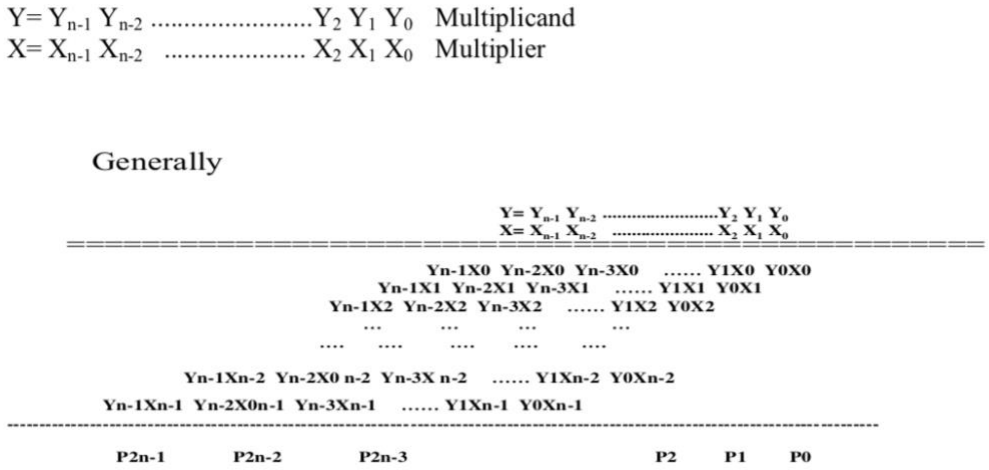


Figure 4. 1:Diagram showing the multiplication of two binary numbers [6]

One hardware implementation of this algorithm is the sequential multiplier, which functions by creating a partial product in each cycle and summing them in a register [7]. The partial product is formed by applying logical AND on the LSB of the multiplier with each bit of the multiplicand. The multiplicand and multiplier are both stored in registers. After each cycle, the multiplicand is shifted left, to increase the bit power of the partial products. The multiplier is right shifted each cycle to ensure the LSB is the correct power for making that partial product [7]. This will take n cycles where n is the number of bits in the multiplier. In this architecture, data words are 16 bits wide, therefore, obtaining a product will take 16 cycles.

The next hardware option is the combinational multiplier which produces the product in one cycle using many adders with AND gates [8]. Logical AND is applied to each bit of the multiplier with all bits of the multiplicand, with the n^{th} bit from the multiplier producing the n^{th} partial product. This is added to the sum of the previous partial products. The LSB of the newly formed sum falls off and is outputted as the nth bit of the product. The remainder of the sum, with the carry out as the MSB is added with the next partial product. This can be seen for a 4x4 bit multiplier in Figure 4.2.

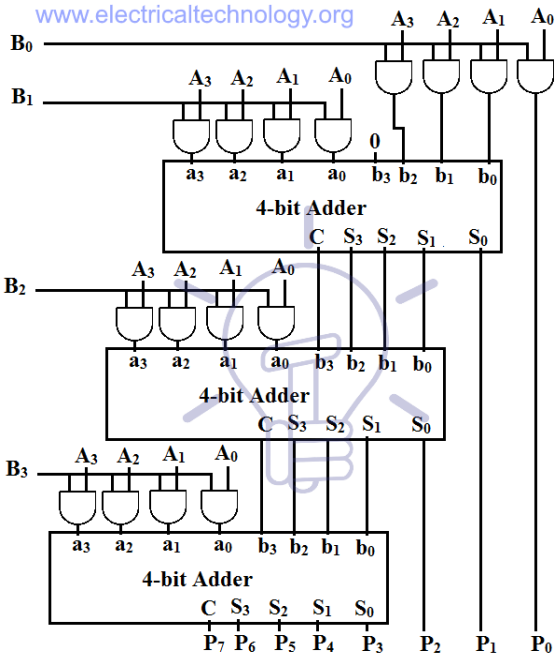


Figure 4. 2: Diagram for a 4x4 binary combinational multiplier[7]

This implementation needs 16 adders, each 16-bit, and 256 AND gates. However this can be optimised by removing the last 16-bit adder and instead using an XOR gate to add the 1^{st} bit of the sum of the previous partial products and the output of an AND gate where the inputs are the 15^{th} bit of the multiplier and the 0^{th} bit of the multiplicand. Reducing to 15 adders, 241 AND gates and 1 XOR gate. This can be done as the as the CPU product is limited size of 16-bits as the data busses are 16-bit wide. And can be seen in Figure 4.3.

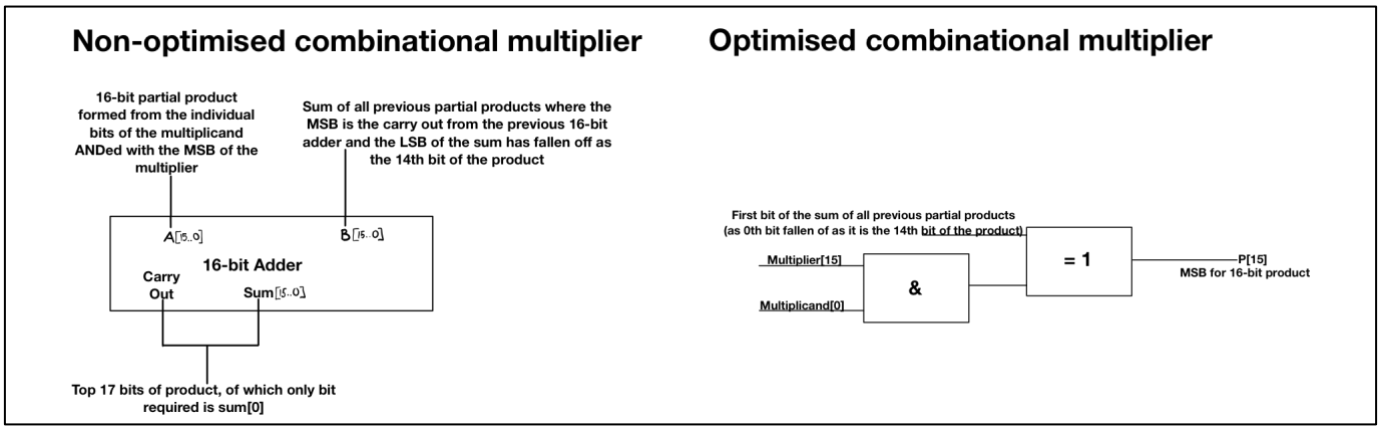


Figure 4. 3: Diagram showing the diffrence between the optimised and non-optimised combinational multiplier.

The sequential multiplier requires significantly less hardware, therefore is optimal for a CPU design where space and low power consumption are a requirement. This comes at the cost of speed, as each multiplication will take a large number of cycles, making multiplication a costly operation [7]. On the other hand, the combinational multiplier has significantly more transistors, therefore it will use more space and have a higher power consumption. This is put aside however as it can potentially complete a multiplication in 1 cycle. However, for larger multiplications with 16-bit data, the single cycle will take a significantly long time due to the delay caused by propagation of the carry through the adders. This can be improved by pipelining the multiplier to use 2 or 3 execution cycles, still significantly less than the sequential multiplier, which would need 16 cycles for each multiplication.

The next option is to implement the multiplication by using look up tables [9]. This would require a form of memory that doesn't need updating such as ROM. The address input is the multiplier followed by the multiplicand. This accesses a location in the memory with a data value equal to the product of the operands. This design works very well for multiplication of small numbers as the available range of products is small and therefore little memory is required. In addition, it uses little hardware so will have low power consumption and take up little space. However, for 16-bit data, the range of products is very large, and would require a very large ROM.

The instruction MUL will be added to the ISA which will use one of the methods described above to multiply two 16-bit numbers together.

4.4 Task 3

The implementation of task 3 depends on how the linked list is represented in memory. One option is for each node of the list to have two memory words. The first of these is to store the data at that node and the next is a pointer to the following node, thereby storing the address of the following nodes data location. A null pointer will terminate the list. The storage of the linked list could be in either single port RAM or true dual port RAM. Dual-port RAM will allow two memory locations to be read in parallel which would reduce the amount of read instructions as both the data of the node and the pointer to the next node could be read at the same time, significantly reducing the number of cycles required [10]. However, this could lead to an increase in power consumption due to increase in ports and may also influence maximum clock speed.

The instruction LDR will be added to the ISA, which will allow register addressing. For a given node, the pointer to the next node can be loaded into a register then using LDR it is possible to access the memory locations of the following node allowing the traversing of a linked list. In addition, if true dual port RAM is used LDR Rd will also load the following memory location into Rd (destination register), which is the pointer to the following node.

4.5 ISA

From the above ideas, the instruction set architecture requires both memory instructions to load/store from memory and register instructions to manipulate register values.

Table 4.1: Memory and jump operations

Assembler	Definition
STA N	Store value from r0 (accumulator) to data memory location N
JMP N	Jump to instruction N
STP	Stop incrementing PC
LDA N	Load value from data memory location N to r0
JMS N	Jump to location N & store PC+1 in stack
BBL	Jump to location at the top of the stack
JEQ Rd, N	Jump if value in R[d] = 0

Table 4.2: Register operations

Assembler	Definition
ADD Rd, Rs	Add value from R[s] to R[d] and stores it in R[d]
SUB Rd, Rs	Subtracts value from R[s] to R[d] and stores it in R[d]
MOV Rd, Rs	Move value of {R[s] + carry-in (Cin)} to R[d]. When the carry-in is 1 (C1) , the instruction can be used to increment as INC Rd, Rs
LSR Rd, Rs	Performs right shift operation to value from R[s] and stores it in R[d]
DEC Rd, Rs	Decrements the value in R[s] by 1 and stores it R[d]
MUL Rd, Rs	Multiplies the values in R[d] and R[s] and the result is stored in R[d]
LDR Rd	Loads R[0] with data taken from the memory location outlined from the value in R[0] (mem [R0]). Then it takes the value from the following data memory location (mem [R0+1]) and stores it R[d].

5. Design Process

5.1 Architecture

Initially, the CPU started based on ARMish design (refer to Appendix D for schematics) because the ARMish design uses registers and does not frequently load data from memory. The CPU was then simplified by removing most of the blocks and converted to Harvard architecture by replacing the single RAM with two separate RAMs. At this point the key parts of the CPU were the register file, program counter, instruction register, state machine and memories as seen in Figure 5.1.

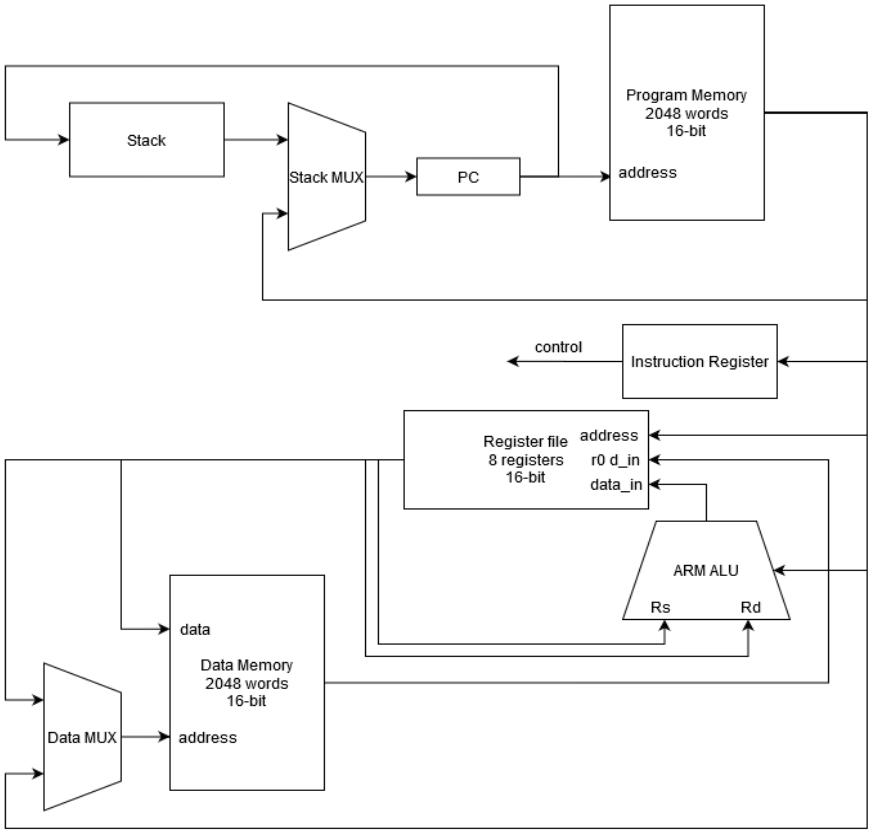


Figure 5,1: Diagram showing the simplified layout of the CPU

5.1.1 Memory

There are two memories, the first memory stores 2048 words of data. This memory contains data and the contents can be changed during the execution of the program. This memory could be single-port or dual-port. Dual port memory will greatly decrease time taken to travel linked lists because the pointer and data of a node can be loaded in parallel, this is explained in further detail in *Design Process/Task 3: Register Addressing*. The second RAM stores 2048 words of instructions (Figure 5.2). It will never be written to, so the write enable port and data input port are grounded. A ROM could be used for this, but it was found to be slower and so limits maximum frequency.

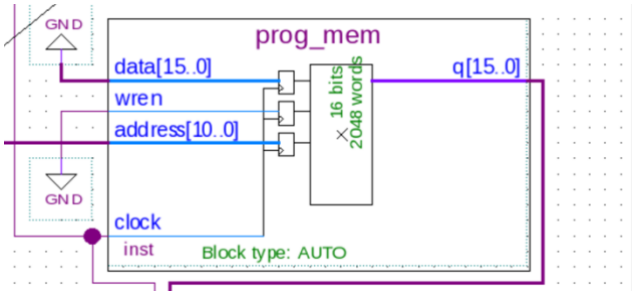


Figure 5.2: Diagram showing the program memory with data input and write enable grounded.

5.1.2 State Machine and Program counter

The state machine has three states: FETCH, EXEC1 and EXEC2. As can be seen from the state diagram in Figure 5.3, the state machine will alternate between FETCH and EXEC1 unless a 3-cycle instruction is executed in which case an EXTRA input (e) determines whether the EXEC2 state is entered after EXEC1.

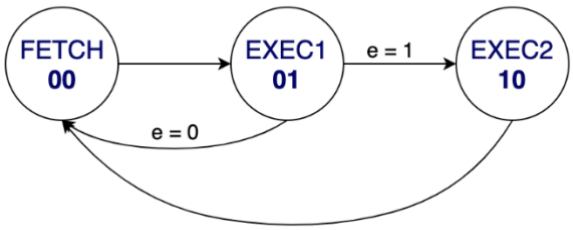


Figure 5. 3: State machine with two execution cycles.

The program counter stores the address of the current instruction. It is incremented in every EXEC1 cycle; if a jump is executed, a new value is written to the program counter.

5.1.3 Decoder

The decoder (Figure 5.4) is used to create control lines for most of the blocks in the design, this includes enable lines to increment (*pc_inc*) or write (*pc_load*) to the program counter, as well as lines to control the state machine (*e* and *m*) and enable writing to r0 (the accumulator) (*acc_load*). It is also used to implement the memory and jump related instructions in the ISA. Jumps are executed by enabling writing to the program counter and choosing a source for the jump address (*jump_mux*). Memory reads can use the operand of the instruction for LDA or the register r0 for LDR. The select signal for this multiplexer are created by the decoder (*data_mux*). In the next execution cycle of the memory read, the write enable for r0 is asserted by the decoder. When writing to memory, the multiplexer selects a source for the address while the write enable of the memory (*WrEn*) is asserted.

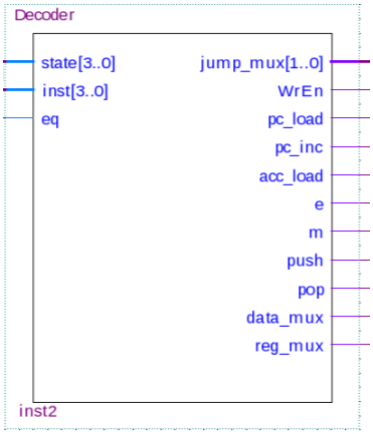


Figure 5.4: Decoder block

5.1.4 Register file

The register file (Figure5.5), contains 4 registers, for MU0 instructions the r0 register is treated as the accumulator. There are three ports for reading and writing to registers. The first port reads and writes to the r0 register only. The second port is a read/write port (Rd) and the final is a read-only port (Rs). The register file uses multiplexers to read from registers. The address is used as the select signal for the multiplexer. A decoder is used to assert the write enable of the required register when writing, using the address as an input.

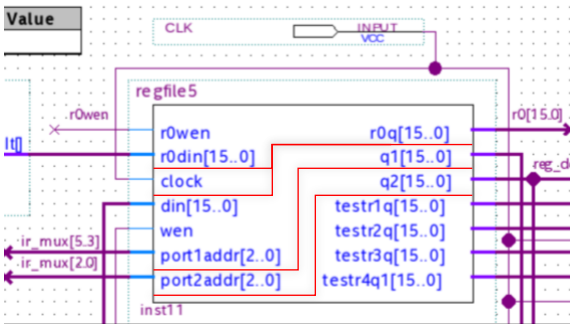


Figure 5.5: Register file block

5.1.5 ALU

The ALU is used for all the register operations in the ISA. An instruction is a register operation if it uses any register in the register file other than r0. It identifies the instruction using the opcode, the MSB of the instruction is high if the instruction is a register operation. The ALU outputs a 16-bit data value to be written to a register if the instruction requires it. For instructions such as MUL and LDR, the data to be written to a register is formed by a different block and input into the ALU to be written to a register.

5.2 Pipelining

Pipelining the CPU will decrease the amount of cycles taken by each instruction by eliminating the idle time of the data path of the CPU during fetch cycles. The CPU was pipelined by modifying the state machine to skip the fetch cycle. The program counter was also changed to increment during fetch and the last EXEC cycle of each instruction. This method of pipelining does not require an adder to selectively increment the program counter. For a state diagram of the new state machine, refer to Figure 5.6.

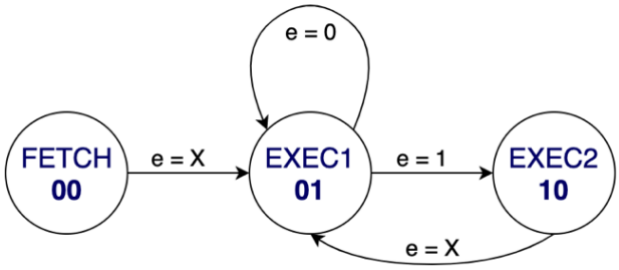


Figure 5.6: Pipelined state machine with two execution cycles.

To pipeline jump instructions, a multiplexer was added to bypass the program counter to allow fetching the next instruction during a jump (Figure 5.7). This also meant that the value written to the PC by the jump had to be incremented before being written, an increment block was used for this.

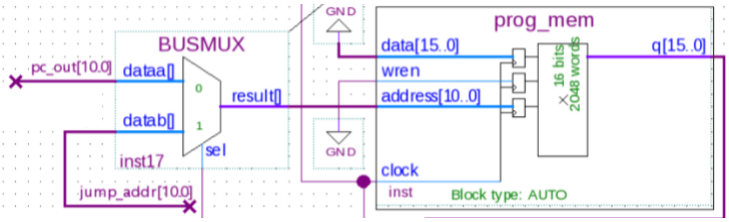


Figure 5.7: Program counter bypass multiplexer.

A problem with this method of pipelining is that STP no longer works normally. The normal implementation of STP works by stopping the program counter from incrementing. However, the program counter now contains the address of the next instruction to be fetched. This implementation of STP will now allow loading the next instruction. To fix this, STP is now treated as a jump instruction, where it unconditionaly jumps to itself, preventing the next instruction from being fetched. This requires that the address of the STP instruction is used as the operand of the STP instruction. This problem with STP can be solved using a block that selectively increments the value of the PC.

5.3 Task1: Stack

Several ways to implement a stack were considered, such as a RAM or a register file. However, it would always need a state machine called a stack pointer. This state machine will point to the next available location on the stack. The stack pointer can be seen in Figure 5.8. It would take the inputs *push* and *pop* and follow: Next state = current state + push – pop, where *push* and *pop* are control lines from the decoder. If *push* is asserted, the value of the program counter is stored at the address corresponding to the current state. In the next cycle, the stack pointer will have a state corresponding to the address of the successive register which will be empty. If *pop* is asserted, the state machine state changes to the previous register and the stack returns a value to the program counter. The read address is taken from the output of the combinational logic of the stack pointer, rather than the register. Doing this removes the 1 cycle delay for the address to update and allows BBL to be executed in 2 cycles rather than 3 cycles.

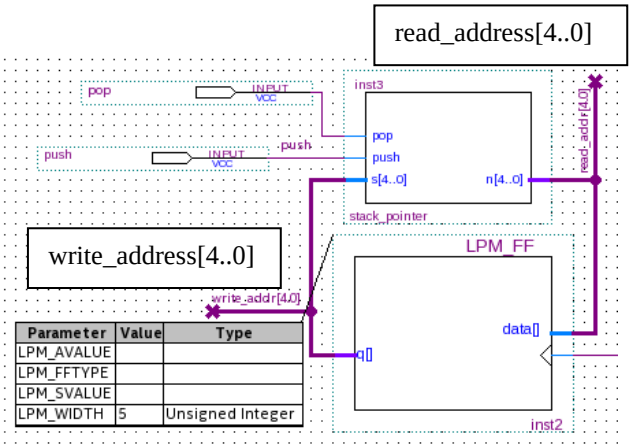


Figure 5.8: Hardware implementation of the stack pointer.

A register file was used as the storage for the stack, refer to Figure 5.5 for a similar register file. The size of this register file determines the maximum recursion depth. The register file was chosen to contain 8 registers because this allows fib(5) to be calculated with some headroom. This register file's *data_input* is the output of the program counter, the *write_address* is formed by the stack pointer and the *write_enable* is the input *push* from the decoder. The read address is formed by the stack pointer. The data output of the register file is connected to the input of the program counter via a mux. The mux selects the stack when executing BBL and selects the instruction register when executing JMS or other jumps.

The instructions JMS and BBL were added to the decoder. During the EXEC1 cycle of JMS, the enable for writing to the program counter is asserted, while the *stack_mux* control line will remain low causing the program counter to be written with the operand from the instruction (*ir_mux*[10:0]). The control line *push* will also be asserted, which asserts the *write_enable* of the register file, storing the incremented value of the program counter into the register that the stack pointer is currently pointing at. This is done by feeding the program counter directly as the input data line into the register file as the CPU is pipelined so the program counter will have already counted up by one. In addition, the asserted *push* will also increment the stack pointer in the next cycle.
During the EXEC1 cycle of BBL, the enable of the program counter and *stack_mux* will both be asserted, thereby writing the value at the top of the stack to the program counter. In this case, the control line *pop* will be asserted, decrementing the stack pointer.

5.4 Task 2: Multiplier

Several possible implementations of the multiplier were created. The first is based on the combinational multiplier spoken about earlier. It was implemented using a Verilog block that takes two 16-bit inputs and outputs a 16-bit result. The multiplier forms 16 partial sums and adds them together, the code can be seen below in Figure 5.9. This will use a lot of transistors and have a very large delay unless pipelined.

```
 8      assign p = ({16{rd_data[0]}} & {rs_data[15:0]}) +
 9                 ({{15{rd_data[1]}} & rs_data[14:0]}, 1'b0}) +
10                 ({{14{rd_data[2]}} & rs_data[13:0]}, 2'b00}) +
11                 ({{13{rd_data[3]}} & rs_data[12:0]}, 3'b000}) +
12                 ({{12{rd_data[4]}} & rs_data[11:0]}, 4'h0}) +
13                 ({{11{rd_data[5]}} & rs_data[10:0]}, 5'b00000}) +
14                 ({{10{rd_data[6]}} & rs_data[9:0]}, 6'b000000}) +
15                 ({{9{rd_data[7]}} & rs_data[8:0]}, 7'b0000000}) +
16                 ({{8{rd_data[8]}} & rs_data[7:0]}, 8'h00}) +
17                 ({{7{rd_data[9]}} & rs_data[6:0]}, 9'b000000000}) +
18                 ({{6{rd_data[10]}} & rs_data[5:0]}, 10'b0000000000}) +
19                 ({{5{rd_data[11]}} & rs_data[4:0]}, 11'b00000000000}) +
20                 ({{4{rd_data[12]}} & rs_data[3:0]}, 12'h000}) +
21                 ({{3{rd_data[13]}} & rs_data[2:0]}, 13'b0000000000000}) +
22                 ({{2{rd_data[14]}} & rs_data[1:0]}, 14'b00000000000000}) +
23                 ({rd_data[015] & rs_data[0]}, 15'b000000000000000});
24
```

Figure 5.9: Section of code from the Verilog ALU which multiplies two 16-bit binary numbers together.

The second implementation used a combination of look up tables and adders. The look up tables multiplied two 4-bit inputs to form an 8-bit output. These partial sums were fed into adders to calculate the product. This method used much less logic and should be much faster; however, this method used more memory. A total of 10 look up tables are needed, using dual-port memories halves the total amount of memory used. There are no dual-port ROMs available, so dual-port RAMs will be used as look up tables. The write enables and data inputs can be grounded as they are not going to be used. This multiplier uses 3 execution cycles because the look up tables have output registers. The first execution cycle is used to send inputs to the look up tables' address ports. In the second execution cycle, results arrive at the output registers of the look up tables. Finally, the signals propagate through adders. This can be seen in Figure 5.10.

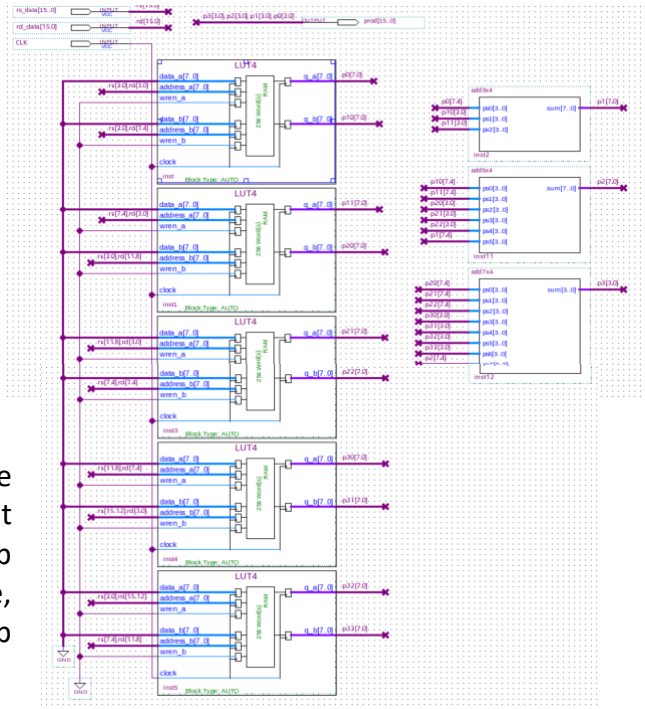


Figure 5.10: Multiplier made using look up tables and adders.

The multipliers were tested using random 16-bit inputs and verified using hand calculations. The test waveforms are provided in the Testing section. The design will use the fully combinational implementation because it requires only one execution cycle.

5.5 Task 3: Register addressing

A true dual-port RAM (Figure 5.11) can be used as the data memory to improve efficiency when traversing a linked list. A multiplexer is connected to the address port *a*. When LDA is executed, the MUX selects the address from the instruction register. When LDR is executed, the MUX selects the address from r0. In addition, the block *increment* is used so that port *b* accesses the following memory address to the value stored in r0. This is to allow the simultaneous reading of both the data and pointer of a node in a linked list, which will be stored in adjacent memory addresses. This second port will never be written to; therefore, the write enable port and data input port are grounded.

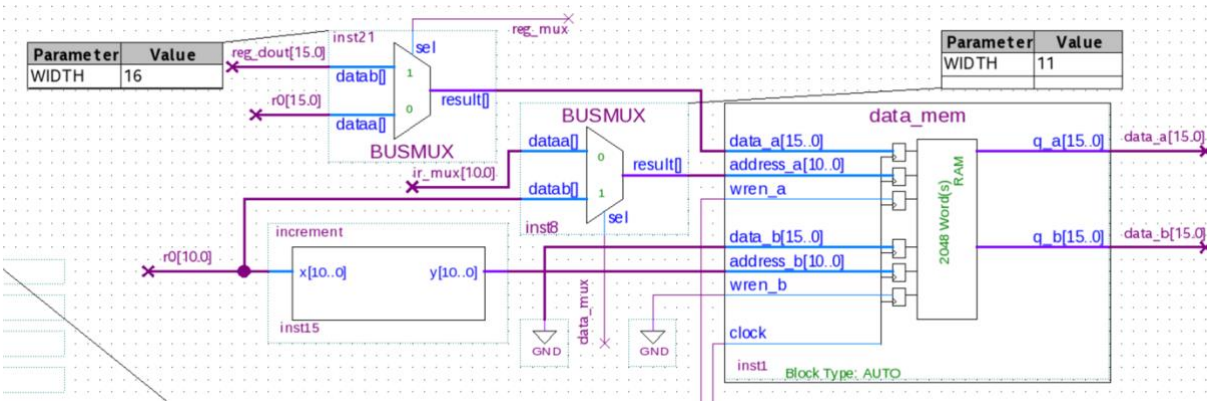


Figure 5.11: Dual port RAM used for Data memory and blocks used to implement register addressing.

5.6 Removal of adder for new register

There were a lot of unused bits in the register operations which could be used for larger register addresses, allowing the size of the register file to be increased. The register ADD operation takes 1 cycle once pipelined, unlike the memory ADD operation which takes 2 cycles. Therefore, the memory ADD operation was removed, with the adder, and replaced with a larger register file of 8 registers.

5.7 JEQ

In order to implement JEQ, the register address from the instruction will have to be sent to the register file. This can be done with a multiplexer. To check if the contents of the register are zero, a NOR gate can be used. The implementation is showed in figure 5.12.

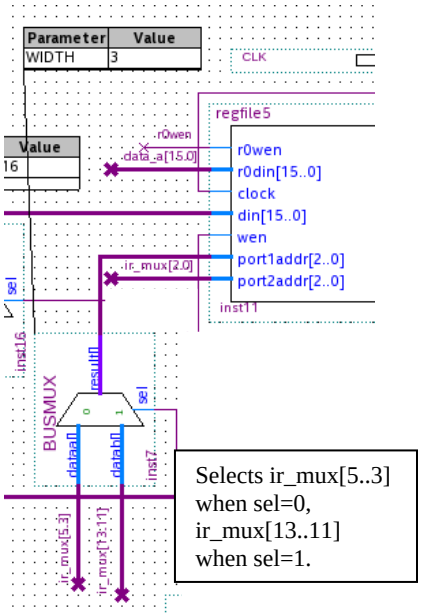


Figure 5.12: Multiplexer to choose source for register file address port.

5.8 Pipelining multiplier

The multiplier will cause the longest delay in the system. To reduce the delay, a pair of registers are added to the input of the multiplier, this makes the whole system faster, but the multiplier will now take two execution cycles rather than a single cycle.

5.9 Memory output register

In order to further reduce the longest delay in the system and increase maximum frequency, a register can be added to the output of the program memory. This means that fetching instructions will take an extra fetch cycle. Once pipelined, all instructions took 2 cycles to execute, this increased the number of cycles for each algorithm significantly. The maximum frequency will be increased slightly; however, the increase in cycles outweighs the speed improvements. The output register was removed to allow easier pipelining.

5.10 Assembler

To speed up the process of converting from assembly code to machine code, a program was created in C++. Initially, this was created for the 4-register design using the instructions in the ISA (Appendix C). To obtain the machine code input the list of instructions into the program, one line per instruction. The program returns a number that corresponds to the memory location the data needs to be planted in. This is followed by the four-digit hex number which corresponds to the machine code of the given instruction. Once the CPU was improved to the 8-register design, the register's address was required to consist of 3 bits with the opcode for all MU0 instructions now being 5 bits. This excluded the JEQ which consisted of 2 bits (Appendix C). Therefore, the machine code was different, and the assembler was updated to work for this new design. Figure 5.13 shows an example of this.

```
ubuntubrad@ubuntubrad-VirtualBox:~/CPU_group_project/GroupProjectGroup$ ./assembler
LDA 0x000
MOV r1, r0
LDA 0x001
MOV r2, r0
LDA 0x002
MOV r3, r0
LDA 0x003
JEQ r3, 0x00D
DEC r3, r3
MUL r0, r1
ADD r0, r2
ADD r4, r0
JMP 0x007
MOV r0, r4
STA 0x004
STP
0: 1800
1: A008
2: 1801
3: A010
4: 1802
5: A018
6: 1803
7: 580D
8: C01B
9: D001
10: 8002
11: 8020
12: 0807
13: A004
14: 0004
15: 100F
```

Figure 5.13 Image showing the input and output of the assembly program

6. **Testing**

Having completed the main blocks for this CPU, it is necessary to test out this design in order to check the completion of each task and its performance. While there are 8 registers in the register file, only 5 of them are used for these tasks and only the first 5 will be displayed on waveforms.

6.1 Multiplier and pipelining

The multiplier blocks were separately designed and tested; Firstly, the Verilog multiplier was tested (Figure 6.1). This multiplier requires only one execution cycle but had a long propagation delay. Next, registers were added to the inputs of the multiplier as described earlier, this delays the output by one cycle but reduces the propagation delay slightly, as seen by the test in Figure 6.2. Finally, the multiplier that uses look up tables and adders was tested (figure 6.3), this multiplier would need 3 execution cycles because of the memory output registers.

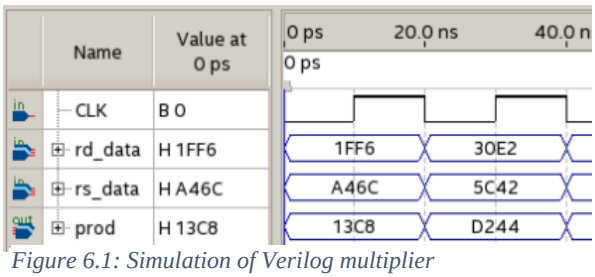


Figure 6.1: Simulation of Verilog multiplier

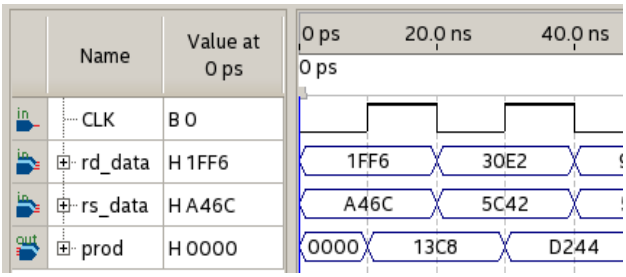


Figure 6.2: Simulation of Verilog multiplier with input registers

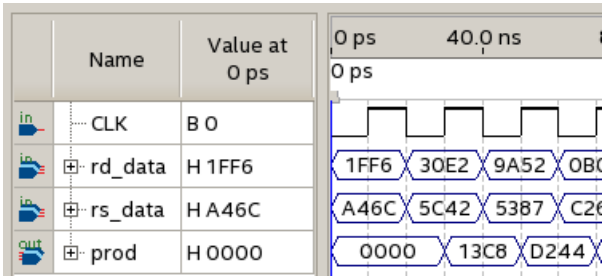


Figure 6.3: Simulation of multiplier made of look up tables and adders.

6.2 Performance

The CPU needs to be tested for its performance, power consumption and size.

The CPU was tested for its performance by calculating its geometric mean time, hence the clock speed of the design and the number of cycles for each algorithm were determined. Firstly, the minimum clock period was found to be 20 ns, this was caused by data propagating from the multiplier's input registers to the register file. This was due to the composition of the multiplier. The multiplier was made up of 15 adders, each adding two 16-bit numbers, as it had to sum 16 signals, each 16 bits wide. The carry propagation through the multiplier caused a large delay, hence the maximum clock frequency (Fmax) was found to be 52.97 MHz.

Next, the three tasks were simulated using test programs shown in Figure 6.4 in order to obtain the number of cycles taken by each algorithm to run. The number of cycles needed were found to be 90, 58, 60 for calculating the value of fib(5), calculating the congruential sum with n = 8 and finding the tenth value in a list, respectively. These waveforms are provided in appendix E (in-depth analysis will be performed for the final design at a later stage in the development process: *Evaluation/Correctness of Tasks*). Thus, by dividing the number of cycles by the clock frequency, the times of each algorithm are: 1.7μs for task 1, 1.095μs for task 2 and 1.133μs for task 3. Hence, the geometric mean time, '$(T_1*T_2*T_3)^{1/3}$' [1, p4] was

found to be 1.282μs. This value can be further reduced significantly by reducing the delay caused by the multiplier.

Task 1	Task 2	Task 3

0: INC R0, R0
INC R0, R0
MOV R3, R0
LDA 0x000
JMS 0x008
MOV R0, R1
STA 0x001
STP
 	
8: LSR R2, R0
JEQ 0x010
DEC R0, R0
JMS 0x008
DEC R0
JMS 0x008
ADD R0, R3
BBL
 	
10: INC R1, R1
BBL
	0: LDA 0x000
	MOV R1, R0
	LDA 0x001
	MOV R2, R0
	LDA 0x002
	MOV R3, R0
	LDA 0x003
	
	7: JEQ r3, 0x00D
	DEC R3, R3
	MUL R0, R1
	ADD R0, R2
	ADD R4, R0
	JMP 0x007
	
	D: MOV r0, r4
	STA 0x004
	STP
		0: LDA 0x001
		MOV R3, R0
		LDA 0x000
		
		3: MOV R2, R0
		LDR R1
		SUB R1, R3
		JEQ R1, 0x008
		JMP 0x003
		
		8: MOV R0, R2
		STA 0x000
		STP

Figure 6.4: Test programs for Task 1, 2, 3.

6.3 Power Consumption and Size

The total thermal power dissipated was found to be 109.41 mW. This can be further broken down into dynamic and static power dissipation. The dynamic thermal power dissipation is the power used caused by transistors switching state within the CPU, it was found to be 9.57mW. The static thermal power dissipation, the power used by the FPGA regardless of the efficiency of the design, is 42.98 mW. The static power is over 4 times larger than the dynamic power because the design uses a small fraction of the resources available in the FPGA, as seen later. Only a small fraction of the logic elements, 767 out of 15,408 available (Appendix F) are used and contribute to dynamic power dissipation. The power dissipation could be reduced slightly by simplifying the design to use less transistors; however, this is not the main goal as the dynamic power dissipation is just a small fraction of the total.

The design's resource usage was also obtained by compiling the CPU, it showed that less than 5% of the total logic elements are being used and only 1% of the total registers available are being used, and the design's memory usage is found to be only 13%. This shows that very little of the FPGA's resources are being used, hence the low dynamic power dissipation.

The initial testing demonstrated that the CPU now fulfils the design requirements. The CPU can execute the three algorithms successfully. The goal now, however, will be to increase the clock frequency. This is because the tests showed that the design only contributes a small fraction of the total power dissipation. The aim is therefore to shorten the longest paths within the CPU, thus reducing the geometric mean time, while trying to keep power dissipation at a similar value.

7. Improvements

In order to better fulfil the design requirements, the maximum clock speed should be increased. This can be done by reducing the size of the longest path that a signal has to travel between registers and reducing the number of total logic units and registers used. In the initial testing, the timing analysis tool was used to find the longest delays in the system. The longest delay was the multiplier, followed by the multiplexer used to implement JEQ.

7.1 Pipelined multiplier

The older multiplier used a lot of adders in order to add 15 signals, each 16 bits wide, in one cycle. This caused a large delay, when measured separately, the maximum frequency of a signal propagating from the input to the output was found to be 56.66 MHz. The delay caused by the multiplier can be reduced if some of the additions are carried out in an extra execution cycle. This will require additional registers to store the partial sums and that the multiplication instruction take three execution cycles. In the first cycle, the 16 signals are added to form 4 signals, then these 4 signals are added together in the new, extra execution cycle, called exec3. The state machine has been modified so that if the new input m is asserted, it transitions from exec2 to exec3 (Figure 7.1). This state was previously unused but necessary to implement the final multiplier. This solution, when tested separately, demonstrated a maximum frequency of 96.26 MHz. However, it is still very expensive in terms of transistors and registers, this could be improved upon if a different type of multiplier is used.

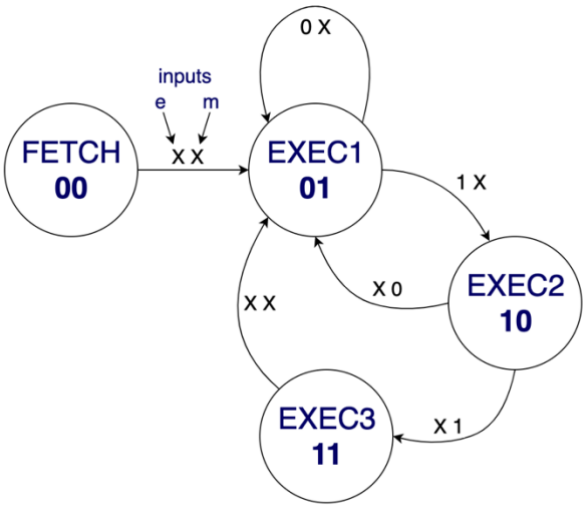


Figure 7.1: State machine with three execution cycles.

7.2 Switch to different multiplier

In order to reduce the delay further, while also significantly reducing the transistor count and total number of registers used, the previous multiplier was replaced with the alternative implementation (Figure 5.10). While using a look up table with 16-bit inputs will require more memory than available, using smaller, 4-bit look up tables followed by adders balances the memory usage with logic block usage. This multiplier increases the memory usage of the design but also increases maximum frequency. This is acceptable as increasing maximum frequency affects the design criteria while memory usage is not. This multiplier demonstrated a maximum frequency of 365.5 MHz when tested separately, this is significantly higher than the previous multiplier implementations.

7.3 Simplifying JEQ

When JEQ was initially implemented, it was required to work with any register, with the address of the register as an operand. This functionality is no longer needed because of the increased number of registers. This functionality was removed, JEQ now only checks the contents of r0. The new implementation of the instruction only needs a NOR gate (Figure 7.2) that inputs the contents of r0 and outputs the *eq* flag, this is far cheaper in terms of delay and transistors. Figure 7.1 shows the new hardware used.

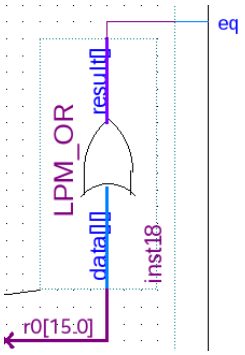


Figure 7.2: A 16-bit logical OR gate, used to implement the new JEQ

7.4 Improved Stack

The stack up to this point contained 8 stack frames, this is sufficient to test that it can calculate fib(5) successfully. In order to make the product more general purpose, the maximum recursion depth should be increased to 32 levels. This value was chosen as 32 stack levels is the typical value for microcontrollers. The stack should also be able to store local variables. This can be done by adding a multiplexer that allows the value of r0 to be stored in stack frames, with the program counter value (Figure 7.3). The register file of the stack was modified to use 32 registers, each 27 bits wide. This is to store the 16-bit value of r0 and the 11-bit value of the program counter. The address bus used by the stack has been increased to 5 bits to allow this. The stack pointer has also been modified to now provide a 5-bit address.

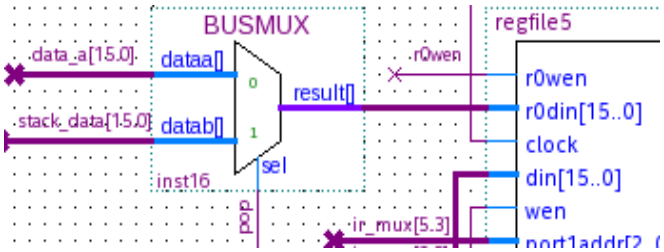


Figure 7.3: Multiplexer allows R0 to be loaded from the stack.

7.5 Turing Complete and general purpose

Finally, in order to make the final product Turing complete, computed jump and register addressing must be fully implemented. Currently, LDR allows data to be loaded from a memory address contained in register r0. To complete the ISA, store with register addressing, STR Rx has to be added. It will allow the value of any one register, specified as an operand, to be stored in the data memory, using the value in register r0 as the address. In order to implement this, the data input to the data memory needs a multiplexer (Figure 7.4) to choose between the r0 port for STA and the register file's read-only port for STR. Another multiplexer is also needed to send the contents of r0 to the address port, this is already in place from implementing LDR.

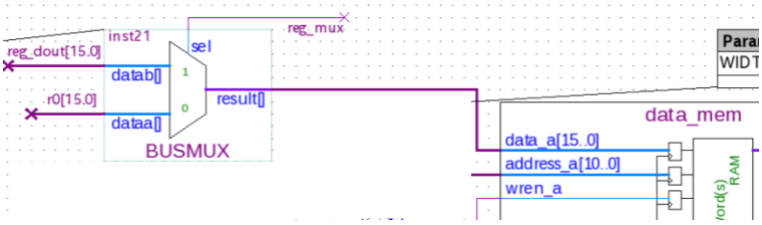


Figure 7.4: Multiplexer to choose data source when writing to memory.

Next, a computed jump is needed. The computed jump works by jumping to the address contained in the register r0. This is done by loading the bottom 11 bits of the register r0 into the program counter. This was implemented by expanding the current jump multiplexer to include the register r0 as a third source for a jump address. The select line will be 2 bits wide, the MSB is asserted for computed jumps while the LSB is asserted for branching back from the stack.

Bitwise instructions will now be added because they make the design more useful as a general-purpose CPU, however they were not required for the benchmark algorithms. The following table shows these instructions.

Assembler	Definition
AND Rd, Rs	Bitwise AND operation between Rd and Rs and stores in result Rd.
ORR Rd, Rs	Bitwise OR operation between Rd and Rs and stores in result Rd.
XOR Rd, Rs	Bitwise XOR operation between Rd and Rs and stores result in Rd.
INC Rd, Rs	Increments the value in Rs and stores in Rd.

Table 7.1: New bitwise operations and INC

Finally, the STP instruction should be fixed. It currently requires the address of the instruction as an operand. Currently, the program counter increments during fetch and during the final execution cycle of an instruction. This means that the value of the program counter is always the address of the next instruction being fetched. The program counter has been modified to increment in exec1 cycles while the current instruction is not STP. In order to keep the pipelining functional, a block has been added to the output of the program counter to selectively increment the output. If enabled, the block increments the output of the program counter. This block is enabled when the PC is being incremented. The STP instruction no longer needs its address as an operand.

7.6 Final ISA

The ISA is now more general purpose as seen from Tables 7.1 and 7.2. JEQ now only operates on r0, therefore it is reverted to a 4-bits opcode. Moreover, all the opcodes for the memory operations were reverted to 4-bits, the opcode is a single hexadecimal digit as this simplifies reading waveforms.

Table 7.2: Memory and jump operations

Assembler	Description	Opcode	Format
STA N	Store value of R0 to memory location N	0000	4'OPCODE + 1'0 + 11'ADDRESS
JMP N	Jump to instruction N	0001	4'OPCODE + 1'0 + 11'ADDRESS
STP	Stop incrementing PC	0010	4'OPCODE + 12'0
LDA N	Load value from memory location N to r0	0011	4'OPCODE + 1'0 + 11'ADDRESS
JMS N	Jump to subroutine, store value of R0	0100	4'OPCODE + 1'0 + 11'ADDRESS
BBL	Branch back from subroutine, load value of R0	0101	4'OPCODE + 12'0
JEQ N	Jump to N if value of R0 is 0	0110	4'OPCODE + 1'0 + 11'ADDRESS
JMC N	Jump to address in R0	0111	4'OPCODE + 1'0 + 11'ADDRESS

Table 7.3: Register operations

Assembler	Description (Result always stored in Rd)	Opcode	Format
ADD Rd, Rs	Add Rs to Rd	10000	5'OP + 5'0 + 3'Rd + 3'Rs
SUB Rd, Rs	Subtract Rs from Rd	10010	5'OP + 5'0 + 3'Rd + 3'Rs
MUL Rd, Rs	Multiply Rd by Rs	11010	5'OP + 5'0 + 3'Rd + 3'Rs
INC Rd, Rs	Increment Rs, store in Rd	10101	5'OP + 5'0 + 3'Rd + 3'Rs
DEC Rd, Rs	Decrement Rs, store in Rd	11000	5'OP + 5'0 + 3'Rd + 3'Rs
AND Rd, Rs	Bitwise AND with Rd and Rs	10001	5'OP + 5'0 + 3'Rd + 3'Rs
ORR Rd, Rs	Bitwise OR with Rd and Rs	10011	5'OP + 5'0 + 3'Rd + 3'Rs
XOR Rd, Rs	Bitwise XOR with Rd and Rs	10111	5'OP + 5'0 + 3'Rd + 3'Rs
MOV Rd, Rs	Move Rs to Rd	10100	5'OP + 5'0 + 3'Rd + 3'Rs

LSR Rd, Rs	Logical shift right Rs, store in Rd	10110	5'OP + 5'0 + 3'Rd + 3'Rs
LDR Rd, Rs	Load value from memory to R0 using address from R0. If operand Rx is not R0, load value at R0+1 to Rx.	11100	5'OP + 5'0 + 3'Rx + 3'0
STR Rd, Rs	Write value of Rx in memory, address taken from R0.	11110	5'OP + 8'0 + 3'Rx

7.6 Final CPU Schematic

The final CPU solution, shown in Figure 7.5, displays how all the blocks are connected in the final schematic.

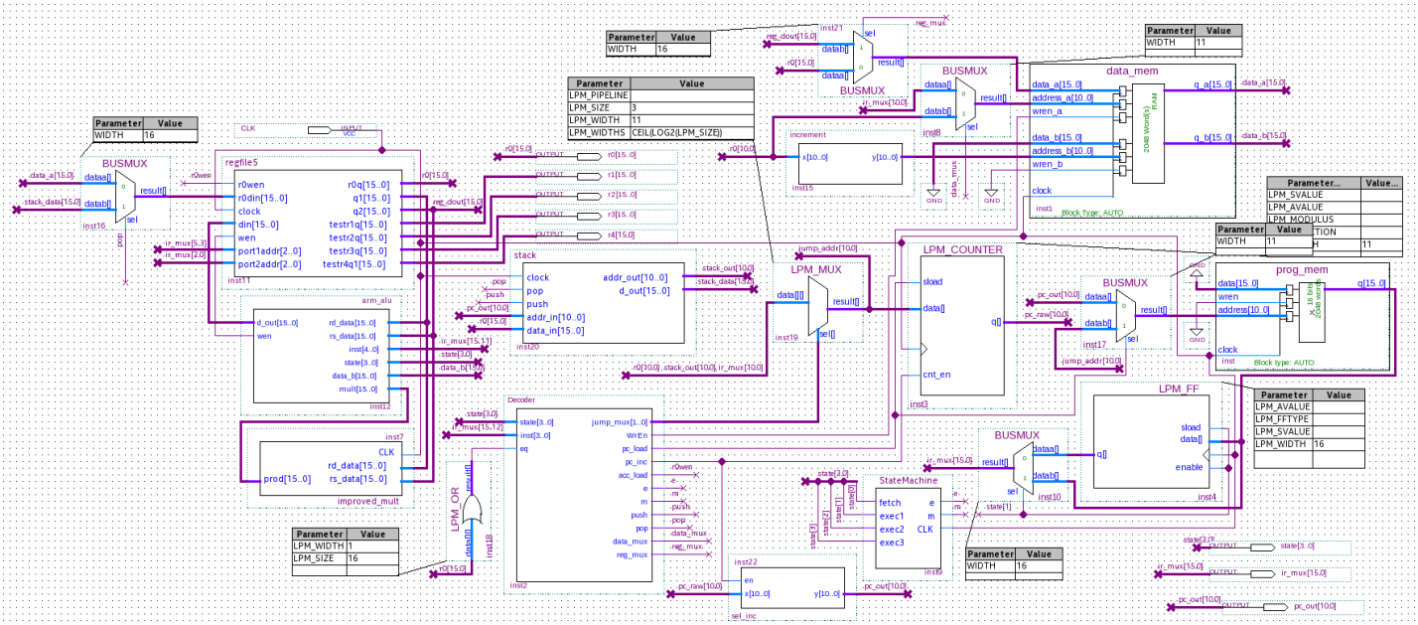


Figure 7.5: Final CPU schematic

8. Evaluation

The final CPU solution has to be evaluated. Firstly, the three benchmark algorithms must be completed successfully. Next, the time taken to complete the benchmarks is to be calculated. The total number of cycles taken for an algorithm to execute have to be measured. This will be determined from simulation waveforms; the cycles are counted starting when the inputs are loaded into registers and end when the outputs are written into memory. The maximum frequency will be measured using the Timing Analysis tool in Quartus Prime. The power consumption and resource usage will also be measured by compiling the design in Quartus for the Cyclone IV E FPGA.

8.1 Correctness of Tasks

The correctness of the three tasks are assessed to ensure the CPU can run the three algorithms successfully. The benchmark programs (Appendix A) are implemented using the ISA and simulated on the CPU, the resulting outputs can be compared to the output of a C++ program.

8.1.1 Task 1

The C++ algorithm to calculate Fibonacci numbers is shown in Figure 8.1. The program was used to provide a benchmark value for calculating the fifth number of the Fibonacci sequence. The result for fib(5) is 8 as shown in Figure 8.2.

```
riccardoelhassanin@ubuntu:~/GroupProjectGroup$ ./fibonacci
input n for fib(n)
5
fib(5) = 8
```

Figure 8.2: Execution of Task 1 C++ code.

Before going through the code for the Fibonacci function in this CPU, the input and output locations need to be chosen in the data memory. The input n=5 is stored in location 0x000, while the result of fib(5) = 8 will be stored in in location 0x001 once the program is executed. Hence the following sequence of instructions was loaded into the program memory in order to simulate the CPU and compare it the above result.

Figure 8.1: Assembly code for Task 1, Fibonacci sequence

```
#include <iostream>

using namespace std;

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

int main()
{
    cerr << "input n for fib(n)" << endl;
    int n;
    cin >> n;

    cout << "fib(" << n << ") = " << fib(n) << endl;

}
```

Table 8.1: Instructions required to calculate fib(5)

Address	Instructions		Description	Equivalent C++
00:	A800	INC R0, R0	Increment value of R0	
01:	A800	INC R0, R0	Increment value of R0	
02:	A018	MOV R3, R0	Moves 2 from R0 to R3, as it is needed later to keep the input constant	
03:	3000	LDA 0x000	Load input from data memory; R0 := 5	int n
04:	A010	MOV R2, R0	Moves 5 to R2, R2 is the input to the subroutine	
05:	4009	JMS 0x009	Jumps to address 0x009 [subroutine starts at 0x009]. Pushes address of next instruction onto stack	fib(n)
06:	A001	MOV R0, R1	Moves result from R1 to R0 so that it can be written to memory	return y;
07:	0001	STA 0x001	Stores value in data memory location 0x001	
08:	2000	STP	Stops the program, PC stops incrementing	
09:	B002	LSR R0, R2	Divides value in R2 by 2 and stores it in R0, to check if initial value in R2 ≤ 1.	if ($n \leq 1$)
0A:	6011	JEQ 0x011	Jumps to address 0x011 if value in R0 is equal to 0. If input ≥ 1, jump fails and moves to next instruction [initially, R0:= 2, hence jump fails and program proceeds to address 0x00B].	
0B:	C012	DEC R2, R2	Decrements value in R2	y = fib(n-1);
0C:	4009	JMS 0x009	Jump to subroutine with n-1 as input	

0D:	C012	DEC R2, R2	Decrements value in R2	y = y + fib(n-2);
0E:	4009	JMS 0x009	Jump to subroutine with n-2 as input	
0F:	8013	ADD R2, R3	2 from R3 to R2 to restore input to its original value	return y;
10:	5000	BBL	Branch back, subroutine ended	
11:	A809	INC R1, R1	The output, R1, is incremented if n <= 1	y = 1;
12:	5000	BBL	Branch back, subroutine ended	return y;

The above program was simulated, and the resulting waveforms can be seen in Figure 8.3. In the waveforms, the dotted lines separate each instruction. The outputs of the state machine, instruction register, and program counter are labelled *state*, *ir_mux* and *pc_out* respectively, while the *stack_* signal is the current address output by the stack.

In the beginning, register R3 is loaded with a value of 2, by using INC instructions to create 2 in R0, then using MOV to move the value to R3. Then, register R0 is loaded with the input which is then moved to register R3. The CPU then jumps to the Fibonacci subroutine using JMS. The subroutine now uses LSR and JEQ operations to check if the input is less than or equal to 1. If the jump passes, the subroutine adds 1 to register R1 and uses BBL to branch back to where the subroutine was called. However, the first JEQ instruction fails as the input is 5, so the input is decremented with DEC and the subroutine is called with n-1, then decremented and called again with n-2. These recursive calls of the subroutine add their results to R1. Finally, R2 is restored to its original value by adding 2, the value of R3. This is done because the input must not be changed by the subroutine, since the input of the C++ algorithm is *const int*, which means the function is not allowed to modify the value of the input. When the subroutine ends and the bottom of the stack is reached, MOV is used to move the result from R1 to R0. This result is then written to memory using STA.

Taking a closer look at the waveforms, the pipelined jumps can be observed, for example, note JMS 0x009 at 70nS, the LSR instruction in 0x009 is fetched during the jump and executed in the next cycle. This is seen as the program counter is directly loaded with 0x00A, instead of loading 0x009 and wasting a cycle to increment the address of the next instruction. This pipelining can also be observed with BBL at 270nS, where the output of the stack is 0x00D but the program counter is loaded with 0x00E while the instruction at 0x00D is being fetched.

The result obtained for an input of 5, is indeed 8, confirming that this CPU performs correctly when compared to the benchmark algorithm for fib(5) and it takes 91 cycles to execute.

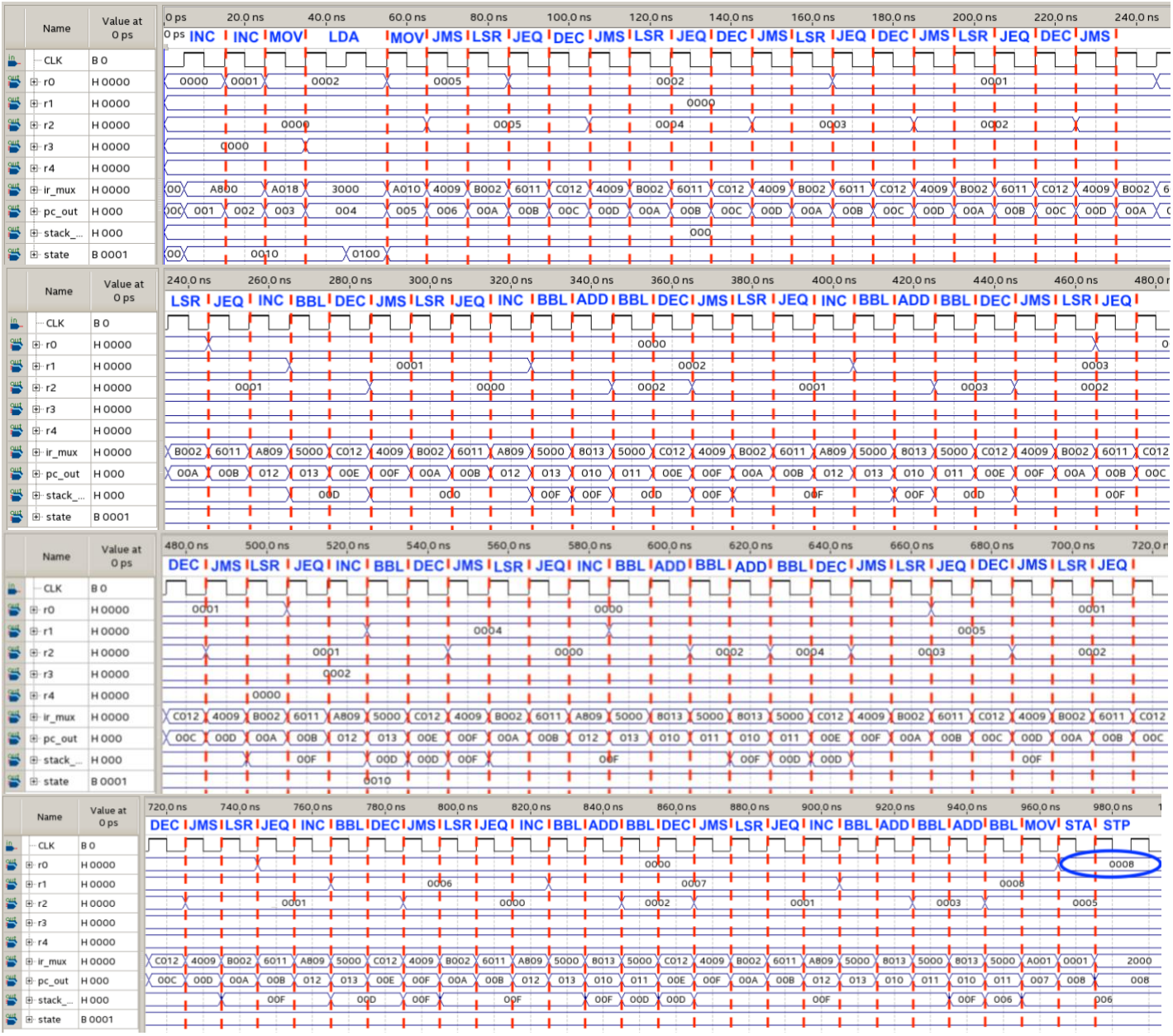


Figure 8.3: Waveforms for the simulation of the program in Table 8.1

8.1.2 Task 2

The program shown in Figure 8.4 implements a linear congruential generator. It was used to provide a benchmark value for $s_{n+1} = (as_n + b) \bmod 2^{16}$ with inputs: a = 25385, b = 3, n = 8, s = 2 as shown in Figure 8.5. For the following tests, the seed (s) which is the starting point of the sequence, is chosen to be 2. The result shows that the eighth congruential sum is 135458716, which in hexadecimal is 812EF9C. After applying the modulo operation, the least significant 16 bits are the result: EF9C.

```
#include <iostream>

using namespace std;

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;
}

int main()
{
int a, b, n, s;
cerr << "input a, b, n, s"<< endl;

cin >> a >> b >> n >> s;

    cout << "sum = " << lcong(a, b, n, s) << endl;
}
```

Figure 8.4: C++ code that calculates pseudo-random integers [1]

```
riccardoelhassanin@ubuntu:~/GroupProjectGroup$ ./LCG
input a, b, n, s
25385 3 8 2
sum = 135458716
```

Figure 8.5: Inputs and corresponding outputs to the C++ program

Firstly, the input values for a, b, n and s are added to the data memory. In location 0x000, a = 6329 is stored, which represents the decimal value 25285 in hex. In locations 0x001, 0x002, 0x003 of the data memory, the values of b = 3, n = 8 and s = 2 are stored accordingly. While in location 0x004 the congruential sum is stored once the program is executed.

The following sequence of instructions was written to the program memory in order to simulate the CPU and compare it the above algorithm.

Table 8.2: Instructions required to calculate pseudo-random integers

Address	Instructions		Description	Extra information
00:	3000	LDA 0x000	Loads value from data memory with location 0x000 in R0; R0 := 6329	
01:	A008	MOV R1, R0	Moves 6329 from R0 to R1; R1 := 6329	
02:	3001	LDA 0x001	Loads value from data memory with location 0x001 in R0; R0 := 3	
03:	A010	MOV R2, R0	Moves 3 from R0 to R2; R2 := 3	
04:	3003	LDA 0x003	Loads value from data memory with location 0x003 in R0; R0 := 2	
05:	A018	MOV R3, R0	Moves 2 from R0 to R3; R3 := 2	
06:	3002	LDA 0x002	Loads value from data memory with location 0x002 in R0; R0 := 8	
07:	600D	JEQ 0x00D	Jumps to address 0x00D if value in R0 is equal to 0 [initially, R0:= 8, hence jump fails and program proceeds to address 0x008]	

08:	C000	DEC R0, R0	Decrements value of R0 by 1 [initially, in R0:= 8 - 1 = 7]	
09:	D019	MUL R3, R1	Multiplies values in R£ and R3 [initially, R3:= R3*R1 = 2*6329 = C652]	$y = y * s$
0A:	801A	ADD R3, R2	Adds values in R3 and R2 [initially, R3:= R3 + R2 = C652 + 3 = C655]	$y = y * s + b$
0B:	8023	ADD R4, R3	Adds values in R4 and R3 [initially, R4:= R4 + R3 = 0 + C655 = C655]	$sum = y * s + b + previus\ sum$
0C:	1007	JMP 0x007	Jumps to address 0x007 [looping repeats until R0:=0]	
0D:	A004	MOV R0, R4	Moves result from R4 to R0 [result is expected to be EF9C]	
0E:	0004	STA 0x004	Stores value in data memory location 0x004	
0F:	2000	STP	Stops the program, PC is stopped	

The following waveforms (figure 8.6) show the simulation of the CPU for the above program. In the waveforms below, the red dotted lines separate each instruction, while the blue dotted lines outline the start and end of the looping of instructions in address 0x007 to 0x00C.

Initially, the program successfully loads values a = 6329, b = 3, n = 8, s = 2 in registers R1, R2, R0 and R3 respectively, in the first 140 ns. The program then runs in loop the instructions from address 7 to C until the value in register R0 is 0 as the new JEQ instruction requires. As seen from the waveforms, every 90 ns, the loops repeat until at 910 ns, the loop stops as the value in register R0 is 0. When the next JEQ instruction is executed, the program successfully jumps to its final instructions of storing the resulting sum and stopping the program. To test if the program is running successfully, it is possible to observe the first loop. At 140 ns, JEQ is executed, and rightfully fails as the value in register R0 is 8 and not 0, hence the following instruction decrements this value by 7. The value 6329 is multiplied by two and incremented by 3 resulting in C655 and the program correctly stores this value in register R3. This value was moved to register four and was added to 0 (the value is 0 because this is the first sum of the program). Thus, the instruction executed so far correctly abide to the algorithm in Figure 8.4. This checking process has been carried out until the end of the execution of the program.

To determine whether this task performs correctly when compared to the benchmark algorithm, the final sum from the waveforms is shown to be 0xEF9C, which is indeed the least 16 bits of the decimal number found with the C++ algorithm: 0x812EF9C. Hence since instruction executed are correct when compared to the given algorithm, the task is correct.

As the multiplier was improved to reduce the delay and transistor count, the MUL instruction affects final the waveforms as it now takes 1 more cycles to execute this instruction. This instruction now requires the values Rs and Rd to be inputted in the address ports of the look up tables (LUT) in EXEC1, their results feed the output registers of the LUT in EXEC2 and go through the adders and to Rd in EXEC3.

When this task was simulated for the input, n=8, it ran the loop 8 times before value in register R0 was 0, therefore it took one more cycle for each loop. As a result, the total number of cycles taken by this program to execute is 66.

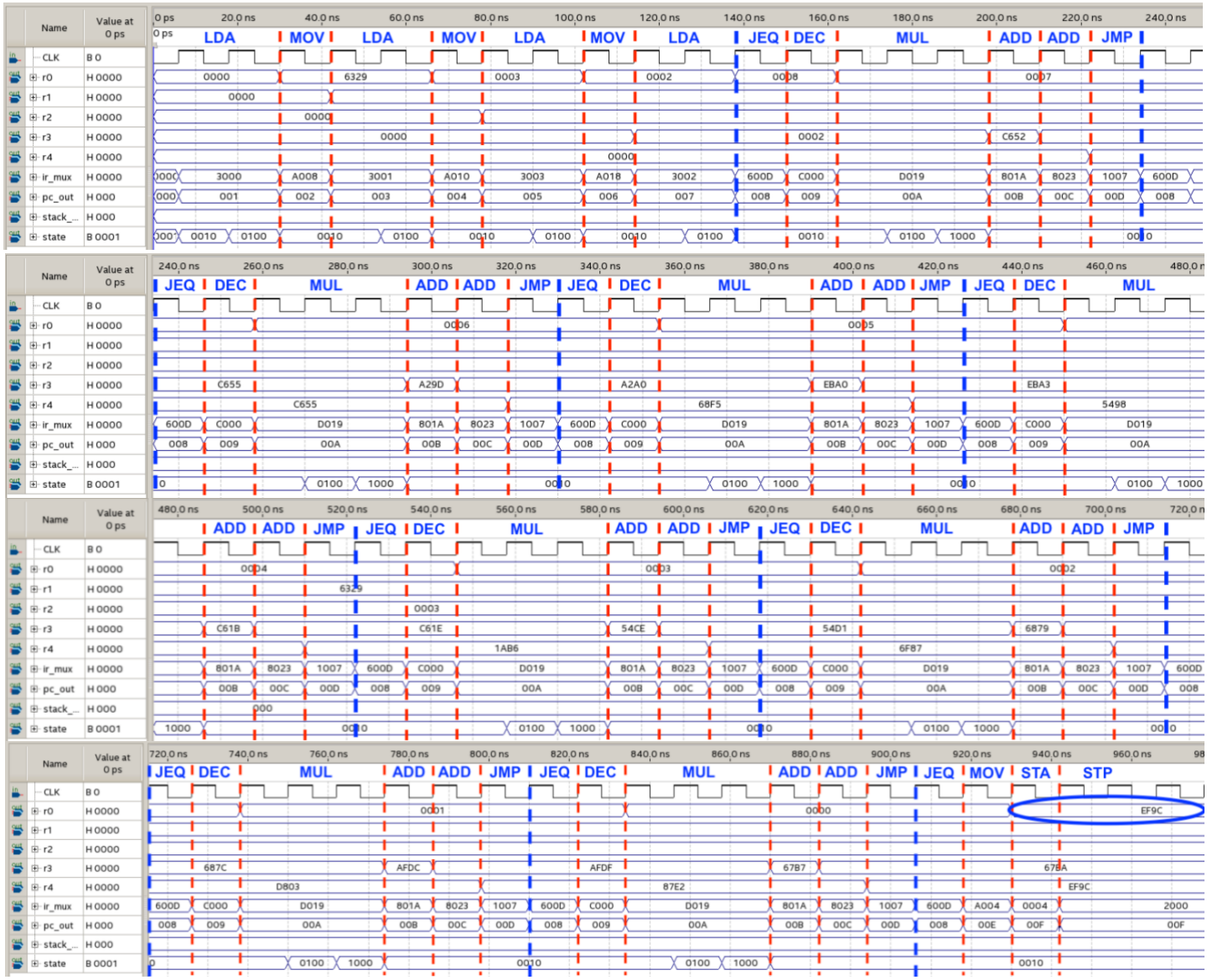


Figure 8.6: Waveforms for the simulation of the program in Table 8.2

8.1.3 Task 3

The C++ program in Figure 8.7 traverses a linked list to find an item. It was used to provide a pointer to a target value in a linked list of length 10 nodes. The following test shows whether the value 9 is in the list or not, and if it is, its pointer is returned as the result.

Firstly, it was needed to compose a list in the data memory, where the tenth value is 9 and its pointer is 0x0212. Moreover, the target value, 0x9, is stored in location 0x001 in the data memory, while the pointer to the head of the list, 0x200, is stored in in location 0x000.

Table 8.3: Table showing memory locations and the corresponding data stored to form a linked list.

000:	0x200	209:	0x20A
001	0x009	20A:	0x005
200:	0x000	20B:	0x20C
201:	0x202	20C:	0x006
202:	0x001	20D:	0x20E
203:	0x204	20E:	0x007
204:	0x002	20F:	0x210
205:	0x206	210:	0x008
206:	0x003	211:	0x212
207:	0x208	212:	0x009
208:	0x004	213:	0x000

Hence the following sequence of instructions was created in order to simulate the CPU and compare it the above algorithm.

```
#include <iostream>
#include <vector>

using namespace std;

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

void initNode(struct item* head, int n){
    head->value = n;
    head->next = NULL;
}

void addNode(struct item *&head, int n){
    struct item *temp = new item;
 temp-> value = n;
 temp -> next = head;
 head = temp;
}

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

int main()
{
    cerr << "input value to find in the list:" << endl;
    int x;
    cin >> x;

    struct item *head = new item;
    initNode(head, 0);
    addNode(head, 1);
    addNode(head, 2);
    addNode(head, 3);
    addNode(head, 4);
    addNode(head, 5);
    addNode(head, 6);
    addNode(head, 7);
    addNode(head, 8);
    addNode(head, 9);

    if (item* p = find(x, head)){
        cout << p->value << " is in the list" << endl;
    }else{
        cout << "inputed value is not in the list" << endl;
    }
}
```

Figure 8.7: C++ code that finds a value in the list

Table 8.4: Instructions required to identify a value by traversing a linked list.

Address	Instructions		Description
00:	3001	LDA 0x001	Loads value from data memory with location 0x001 in R0; R0 := 9
01:	A018	MOV R3, R0	Moves 9 from R0 to R3; R3:= 9
02:	3000	LDA 0x000	Loads pointer from data memory with location 0x000 in R0; R0 = 0x200
03:	A008	MOV R1, R0	Moves pointer from R0 to R1 [initially, 0x200 is moved to R1]
04:	A011	MOV R2, R1	Moves pointer from R1 to R2 [initially, 0x200 is moved to R2]
05:	A001	MOV R0, R1	Moves pointer from R1 to R0 [initially, 0x200 is moved to R0]
06:	E008	LDR R1	Loads R0 with value from data memory with address mem[R0], [initially R0 is loaded with data from mem[0x200], which is 0x000]

			Also loads R1 with value from data memory with address mem[R0 #1], [initially, R1 is loaded with data from mem[0x201], which is 0x202]
07:	9003	SUB R0, R3	Subtracts value in R3 from R0 [initially, R0:= R0 – R3 = 0 - 9 = FFF7]
08:	600A	JEQ 0x00A	Jumps to address 0x00A if value in R0 is equal to 0 [initially, R0:= FFF7, hence jump fails and program proceeds to address 0x009]
09:	1004	JMP 0x004	Jumps to address 0x004 [looping repeats until R0:=0]
0A:	A002	MOV R0, R2	Moves result pointer from R2 to R0 [result is expected to be 0x212]
10:	0002	STA 0x002	Stores pointer in data memory location 0x000
11:	2000	STP	Stops the program, PC is stopped

The following waveforms show the simulation of the CPU for the above program (Figure 8.8). The program successfully loads 9, the value that it is desired to be found in the list, and moves it to register R3, then loads its pointer which is 0x200 and moves it to register R1, in the first 80ns. The program then runs in loop approximately every 80ns as it executes instructions from address 4 to 8 until the value in register R0 is 0 as the outlined earlier. The looping stops at 900ns, as the last JEQ instruction is successfully executed since the value in register one is 0, thus jumping to its final instructions of storing the resulting pointer and stopping the program.

To test if the program is running successfully, it is possible to observe the first loop. At 80 ns, the pointer value is moved to register one and subsequently back to register R0. Then register R0 is loaded with the value 0 through register addressing. This form of addressing successfully works as the value in address 200 in the data memory is indeed 0. Simultaneously, the pointer 0x202, is stored in the next memory location, mem[201], is loaded into register R1. Then the value 0 is subtracted to the input, 9, to check if the result corresponds to 0. If the result is zero a jump would occur in the next instruction as the input would match with the value in the list. However, when JEQ is executed, it rightfully fails as the value in register R0 is FFF7 since 0 – 9 is not equal to 0. Hence the next jump instruction happens, and the loop repeats 9 more times until the input value is found, as seen from the regions enclosed between the blue dotted lines in Figure 8.8.

The final result produced by this program is 0x212, which corresponds to the pointer pointing to the tenth value in the list, 9. Hence, this program can successfully return the pointer that points to the value desired to be found as the benchmark algorithm predicts.

Since the program needs to find the pointer to the tenth value in the list, it loops ten times before the value in register R0 is 0, hence a total of 71 cycles are needed to finish execution and provide the final result.

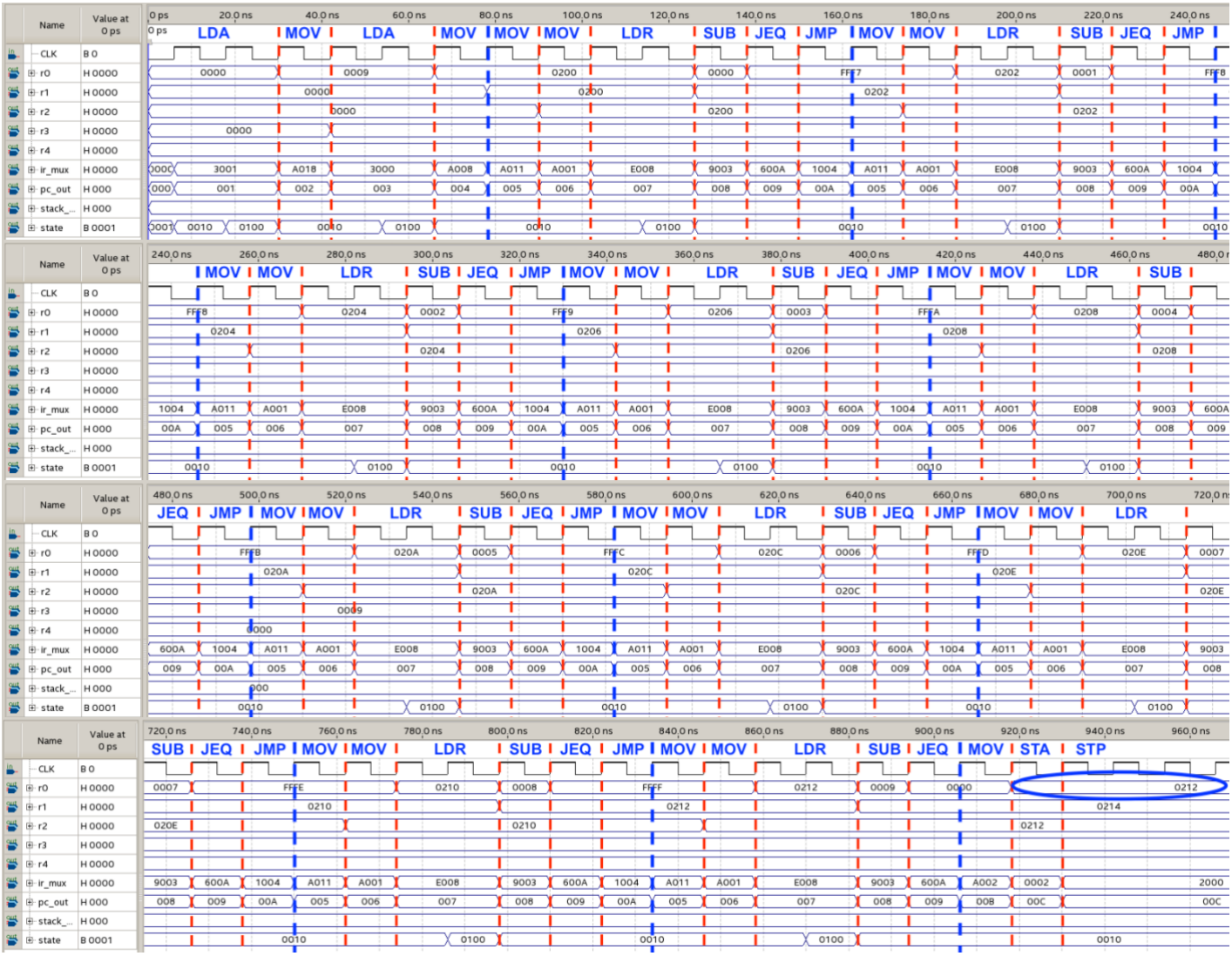


Figure 8.8: Waveforms for the simulation of the program in table 8.4

8.2 Performance

The design performance was greatly enhanced as the clock period of this design has been reduced to 11.6ns, almost doubling the clock speed of the design. This is because of the new multiplier, which has reduced the latency to about 10ns because the long carry propagation, which previously caused delay, is now removed. Hence, the slowest path was shortened and now is from the program memory to the program counter. The delay is now mainly due to the new stack as it is was modified to contain 32 levels.

Additionally, by simplifying JEQ, additional delay reduction was achieved. By reducing the delay and increasing the clock speed, the maximum clock frequency (F_{max}) was increased to 90.08MHz. The higher clock frequency allows for a better performance of the CPU since each task will take less time to execute each algorithm. Thus, by dividing the number of cycles taken by each algorithm (91, 66 and 71 cycles for task 1, 2 and 3 respectively) by the new clock frequency, the execution times of each algorithm are calculated to be: 1.010μs, 0.733μs and 0.788μs respectively for tasks 1, 2 and 3. Therefore, the final geometric mean time was found to be 0.836μs. This value has been minimized by 0.446μs from its previous design, thus increasing its performance by almost 35%. The maximum frequency, at its peak, was

97.1MHz. However, in order to make the design Turing complete the total delay increased due to the implementations of STR and JMC increasing the length of the path from program memory to the program counter.

8.3 Analysing execution time and input size

In Task 1, the execution time varies linearly with the Fibonacci sequence, this means that the execution time increases exponentially as the input grows. This is a very inefficient algorithm but is a good benchmark as it tests the stack. The execution time of Task 2 varies linearly with input n. The execution time is also unaffected by the other inputs. If used to generate a sequence of random numbers by incrementing n, this algorithm will slow down as the sequence grows longer. The execution time of Task 3 depends on the distance of the target node from the head. In the graph for Task 3, the list searched was a sorted list with values of the target nodes, n, going from 0 to 9.

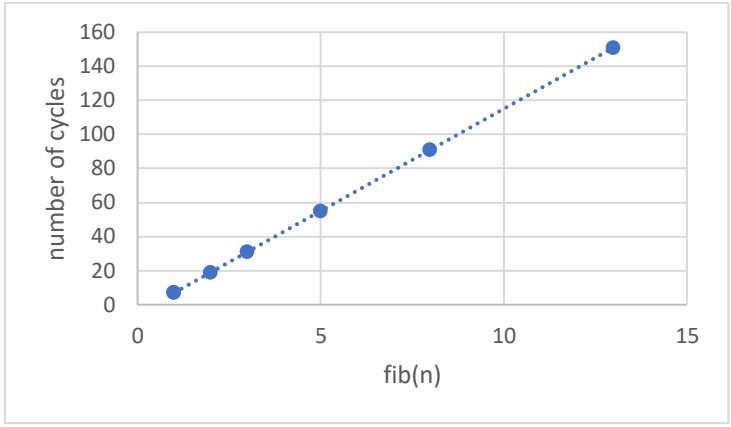


Figure 8.9: Task 1 execution time against output

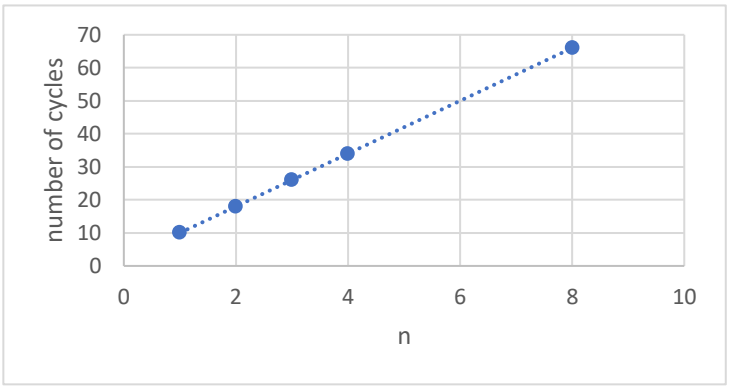


Figure 8.10: Task 2 execution time against input

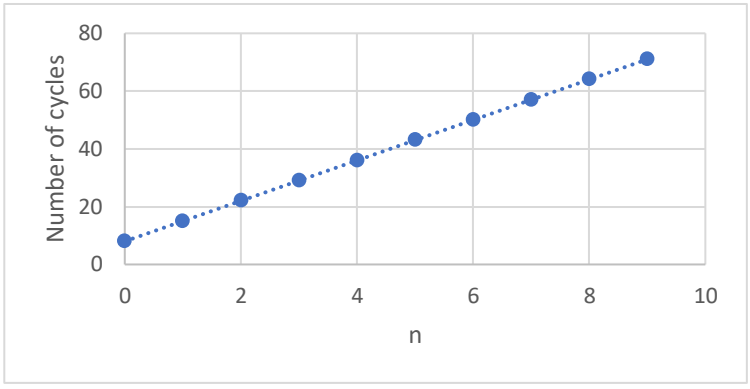


Figure 8.11: Task 3 execution time against distance to node

8.4 Power consumption and Size

Following the improvements to the design, the resource utilization has changed. Firstly, the multiplier reduced the resource usage greatly, as a great number of register and logic gates in the adders were removed and replaced by look up tables thus reducing the transistor count, the logic block usage and register usage were 535 and 200 respectively. However, through the implementation of look up tables, the memory usage of the design has increased to 73,728 bits. However, in the final design, the total logic blocks and registers used are respectively 965 blocks and 466 registers. This increase is mainly due to the full register file implementation and new instructions added to make the design more general purpose. Full resource usage is attached in Appendix G.

The total thermal power dissipated increased to 129.32 mW. This can be further separated into dynamic and static power. The CPU currently uses a bigger fraction of FPGA than the previous design. As a result, the core dynamic thermal power dissipated by the CPU, as transistors switch states, is now 26.36 mW, which is almost triple the value obtained in the

testing section. This is because the design's new total logic elements and register usages are 965 and 466 respectively which are much higher than the 767 and 232 found in the *Testing/Power Consumption and Size*. However, when analysing the overall solution, the fraction of the FPGA being used is still low as the core static thermal power dissipation was found to be 43mW. The static power is still almost double the dynamic because only a small fraction of FPGA is used and contributes to dynamic power as the majority of the logic elements are still not being used since the logic block and register usage is only 15% and 7%. Therefore, the power draw caused by the design is still only a small fraction of the overall power dissipation.

8.5 Optimal values for the Linear Congruential Generator LCG

The specification also requires the LGC to be evaluated to find the optimal values of a and b in '$x_{n+1} = (ax_n + b) \bmod 2^N$' [1, p3] that will lead to the longest sequence. This would not be reasonable to do using this CPU as to obtain such a large set of data to find optimum values would take a very long time, instead research was done. Depending on how the LGC is implemented there are different groups of variable choices. For this CPU the value of $N = 16$ is fixed therefore the 'power-of-two' [5, p1] choice is the best. This choice requires a to be either *3 mod 8* or *5 mod 8*, b=0 and the seed must be odd. This will provide a max sequence length of $2^{16}/4 = 16384$ [5]. This value is significantly larger than the length of sequence required in the typical problem described in the specification.

9. **Conclusion**

During the 5-weeks of this project, the CPU has been researched, conceptualised, designed evaluated and improved. Ending up with a final design that efficiently completes the specification tasks. As well as optimised for speed, power consumption and size. However, moving forward, if there was more time there are improvements that could be done. Firstly, in order to increase the geometric mean time further, the number of cycles taken by Task 2 could be decreased by improving the multiplier further to use only 2 execution cycles. The later part of the design process was focused on making the design more general purpose. The addition of register addressing would allow the design to address a maximum of 64k words of data memory, the computed jump would also allow addressing 64k words of program memory. In order to make the design more general purpose, a few more improvements could be made. A JMI instruction could be added, which checks if the value of R0 is two's complement negative. This will make comparison of numbers much faster. Flags could also be added to allow the register operations to use a carry bit or use conditional execution with a skip flag.

10. Referencing

Bibliography

[1] E. Perea and E. Stott, *ELEC40006: 1st Year Electronics Design Project 2020 Initial specification*. 2020.

[2] E. Perea and E. Stott, "Project deliverables: Final Report | Video", online, 2020.

[3] S. Khillar, "Difference between Von Neumann and Harvard Architecture | Difference Between", *Differencebetween.net*, 2020. [Online]. Available:

http://www.differencebetween.net/technology/difference-between-von-neumann-and-harvard-architecture/. [Accessed: 05- Jun- 2020].

[4] "Intel 4004", *En.wikipedia.org*, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Intel_4004. [Accessed: 02- Jun- 2020].

[5] Linear congruential generator", *En.wikipedia.org*, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Linear_congruential_generator. [Accessed: 07- Jun- 2020].

[6] R. Shirshendu, "Sequential Multiplier - Digital System Design", *Digital System Design*, 2019. [Online]. Available: https://digitalsystemdesign.in/sequential-multiplier/. [Accessed: 01- Jun- 2020].

[7] *Multipliers*. 2019. Available: https://eee.guc.edu.eg/Courses/Electronics/ELCT706%20Microelectronics%20Lab/winter%202014/Multipliers.pdf

[8] "Binary Multiplier - Types & Binary Multiplication Calculator", *ELECTRICAL TECHNOLOGY*, 2020. [Online]. Available: https://www.electricaltechnology.org/2018/05/binary-multiplier-types-binary-multiplication-calculator.html. [Accessed: 29- May- 2020].

[9] [9]P. Andraka, "Multiplication in FPGAs | Andraka Consulting Group", *Andraka.com*, 2020. [Online]. Available: http://www.andraka.com/multipli.php. [Accessed: 05- Jun- 2020].

[10] Intel, *Embedded Memory (RAM: 1-PORT, RAM: 2-PORT, ROM: 1-PORT, and ROM: 2-PORT) User Guide*. Intel, 2020. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_ram_rom.pdf

11. Appendices

11.1 Appendix A: Algorithms

1. C++ algorithms to calculate Fibonacci numbers using recursion. Typically, n=5.

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

[1]

2. C++ algorithms to calculate pseudo-random integers with a linear congruential generator (LCG). Typically, a = 25385, b = 3, n = 8, s = seed.

```
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;
}
```

[1]

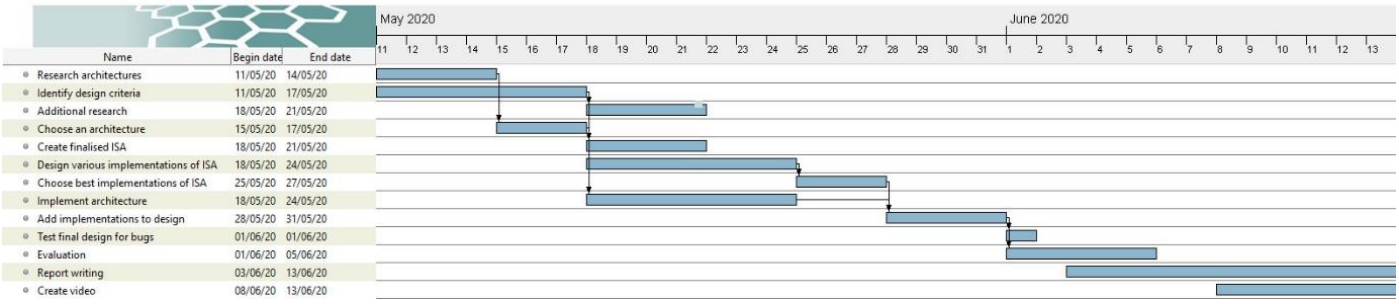
3. C++ algorithms to traverse a linked list to find an item.

```
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;
}
```

[1]

11.2 Appendix B: Gantt chart



11.3 Appendix C: ISA

4-register ISA:

Memory instructions:

Assembler	Opcode	Hex	Format
STA N	0000	0	4'OP + 0 + 11'ADDR
JMP N	0001	1	4'OP + 0 + 11'ADDR

JEQ Rd, N	001	2/3	3'OP + 2'Rx + 11'ADDR
STP	0100	4	4'OP + 0 + 11'ADDR
LDA N	01010	5	5'OP + 11'ADDR
ADD N	01011	5	5'OP + 11'ADDR
JMS N	0110	6	4'OP + 0 + 11'ADDR
BBL	0111	7	4'OP + 12'0

Register instructions:

Assembler	Opcode	Hex	Format
ADD Rd, Rs	1000	8	4'OP + 8'0 + 2'Rd + 2'Rs
SUB Rd, Rs	1001	9	4'OP + 8'0 + 2'Rd + 2'Rs
MOV Rd, Rs	1010	A	4'OP + 1'CIN + 7'0 + 2'Rd + 2'Rs
LSR Rd, Rs	1011	B	4'OP + 8'0 + 2'Rd + 2'Rs
DEC Rd, Rs	1100	C	4'OP + 8'0 + 2'Rd + 2'Rs
MUL Rd, Rs	1101	D	4'OP + 8'0 + 2'Rd + 2'Rs
LDR Rd, Rs	1110	E	4'OP + 8'0 + 2'Rx + 2'0

8-register ISA:

Memory instructions:

Assembler	Opcode	Format
STA N	00000	5'OP + 11'ADDR
JMP N	00001	5'OP + 11'ADDR
STP	00010	5'OP + 11'ADDR
LDA N	00011	5'OP + 11'ADDR
JMS N	00100	5'OP + 11'ADDR
BBL	00101	5'OP + 12'0
JEQ Rd, N	01	2'OP + 3'Rx + 11'ADDR

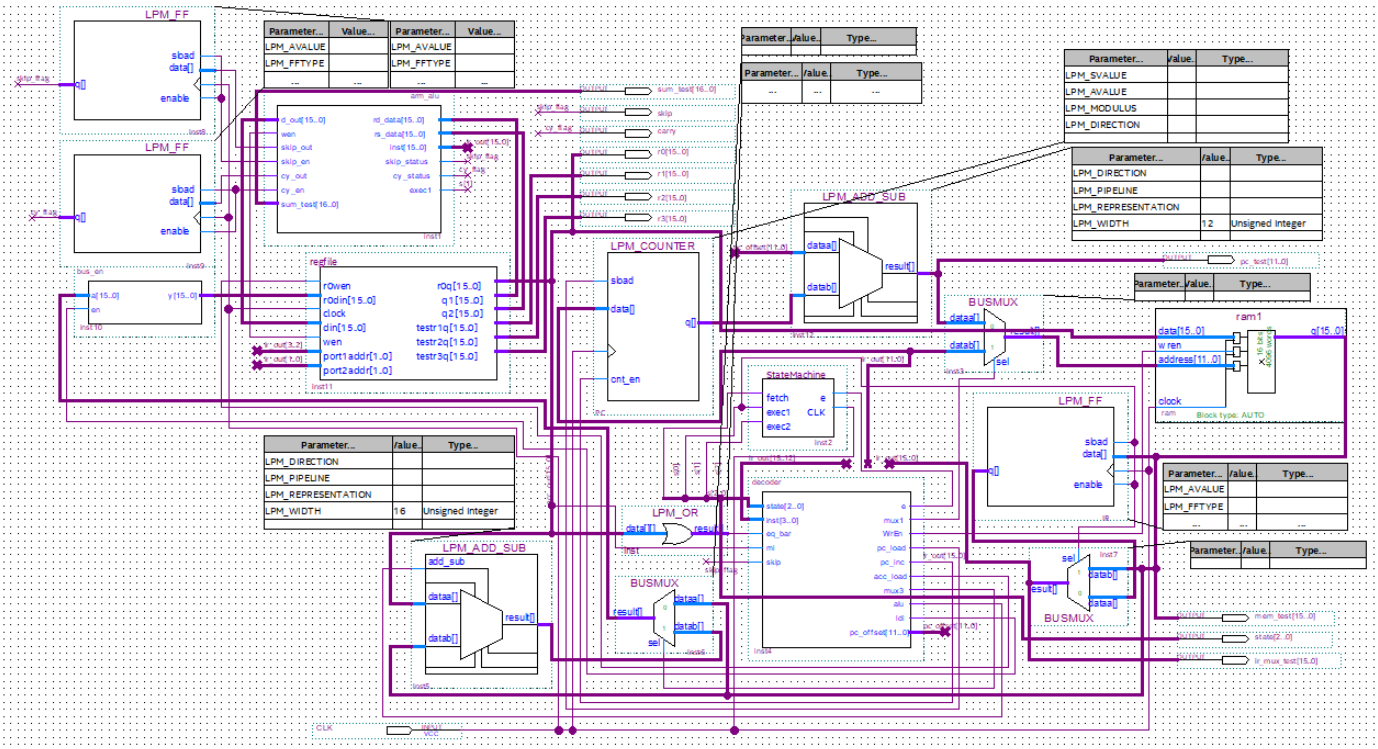
Register instructions:

Assembler	Opcode	Hex	Format
ADD Rd, Rs	1000	8	4'OP + 6'0 + 3'Rd + 3'Rs
SUB Rd, Rs	1001	9	4'OP + 6'0 + 3'Rd + 3'Rs
MOV Rd, Rs	1010	A	4'OP + 1'CIN + 5'0 + 3'Rd + 3'Rs

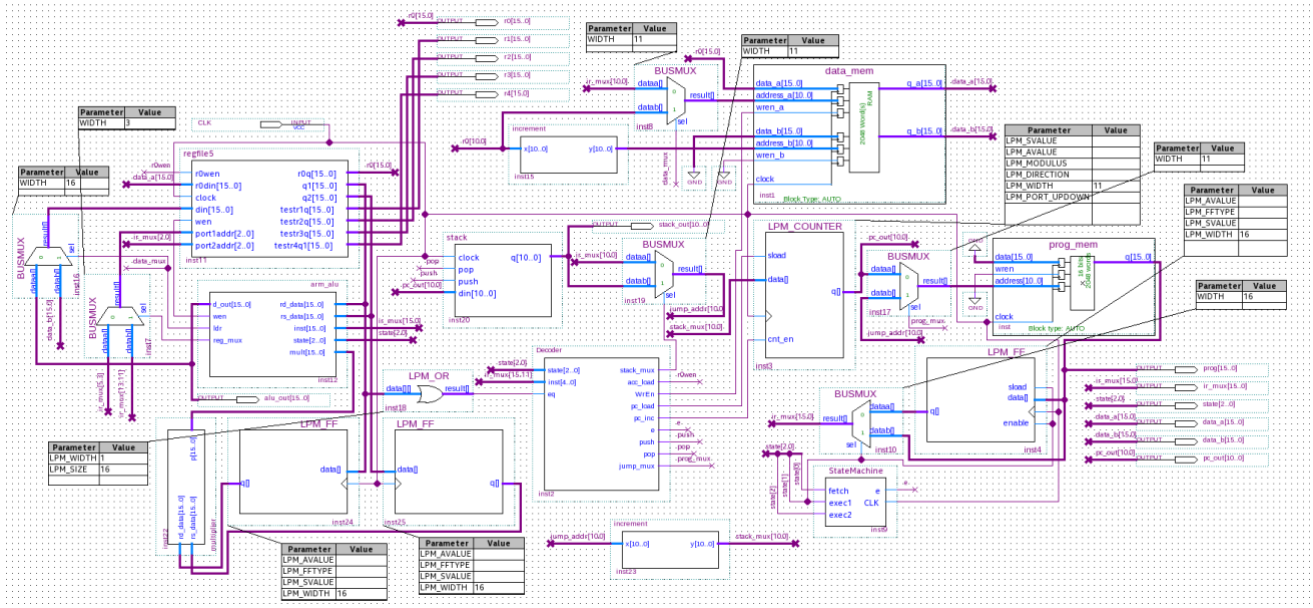
LSR Rd, Rs	1011	B	4'OP + 6'0 + 3'Rd + 3'Rs
DEC Rd, Rs	1100	C	4'OP + 6'0 + 3'Rd + 3'Rs
MUL Rd, Rs	1101	D	4'OP + 6'0 + 3'Rd + 3'Rs
LDR Rd	1110	E	4'OP + 6'0 + 3'Rx + 3'0

11.4 Appendix D: Main block schematics

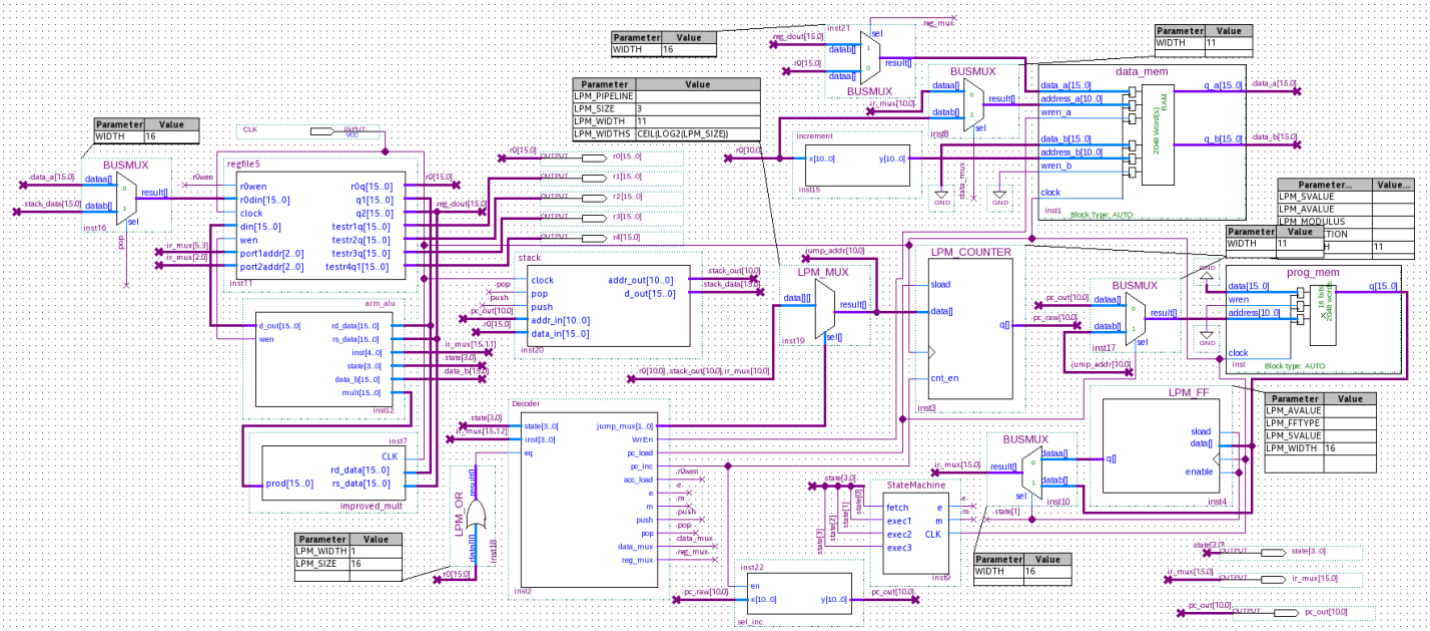
Original ARMish CPU



First Working design



Final design

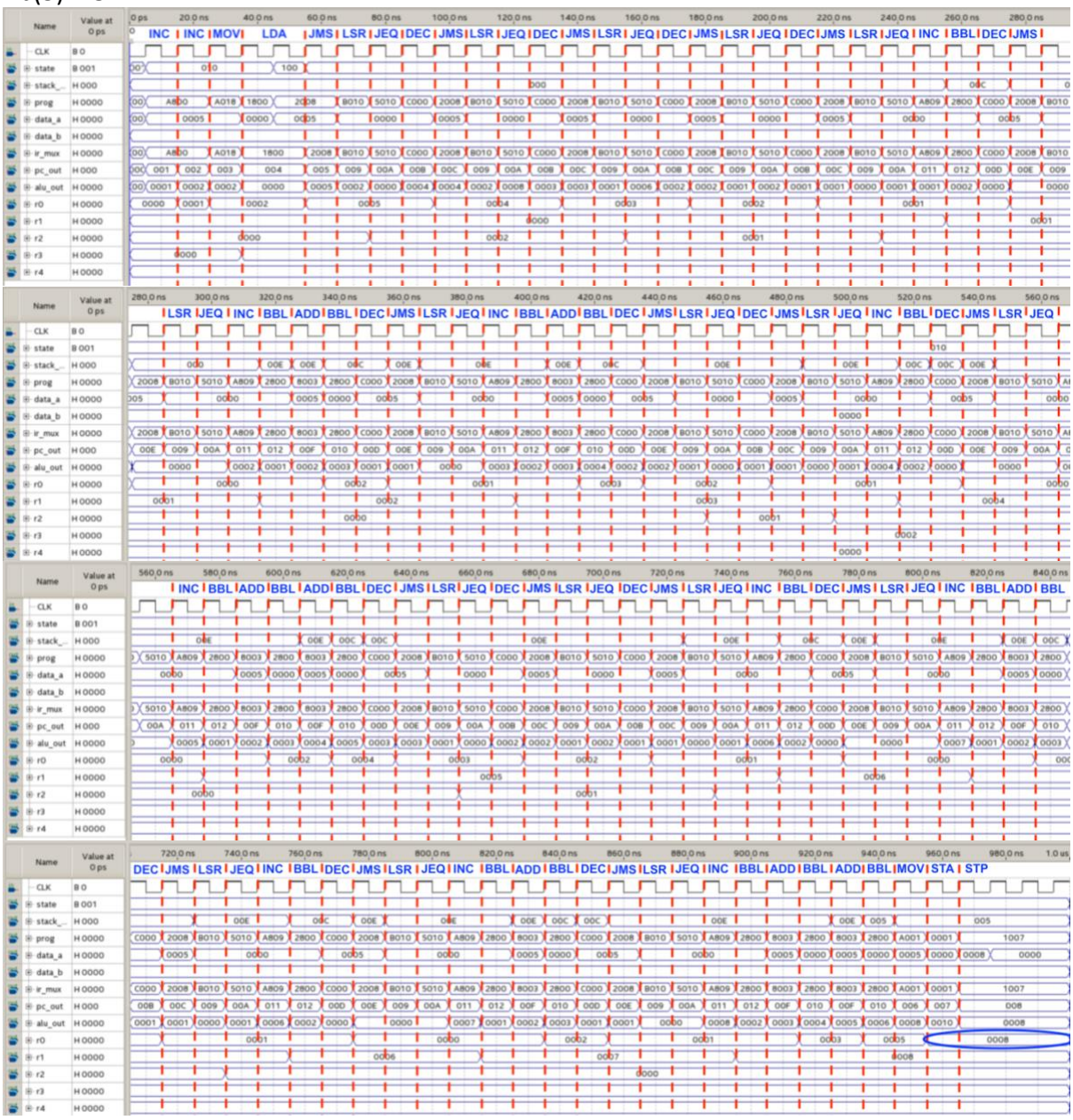


11.5 Appendix E: Waveforms and Cycles for the first working design

Signal	Description
state	state[0] = fetch, state[1] = exec1, state[2] = exec2
stack_	Address output of stack
prog	Data output of program memory
data_a, data_b	Data outputs of data memory
ir_mux	Instruction currently being executed
pc_out, alu_out	Output of program counter/ALU
r0, r1, r2, r3, r4	Values of first 5 registers

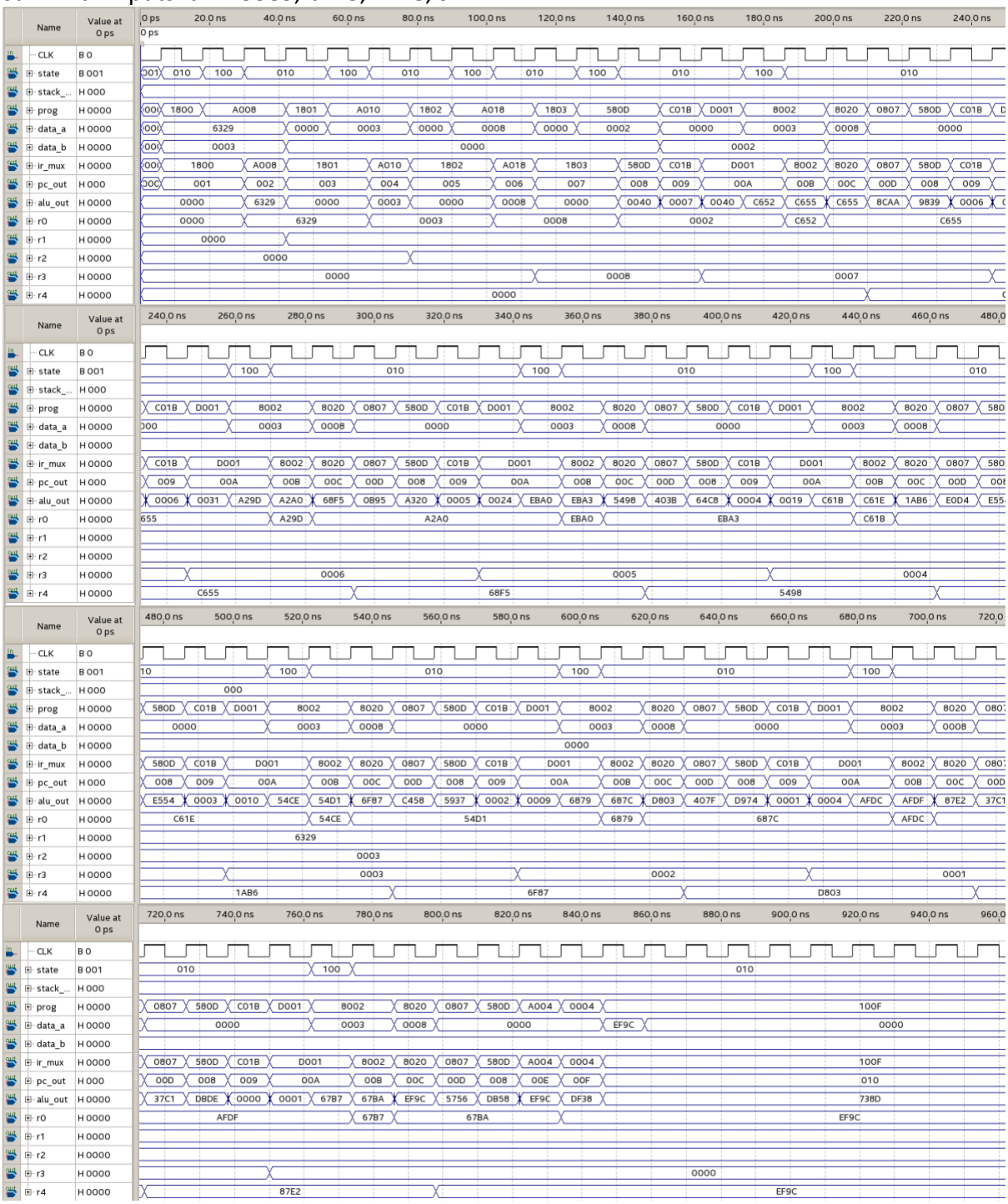
Task 1

As shown by the following waveforms, the program takes 90 cycles to execute successfully fib(5) = 8



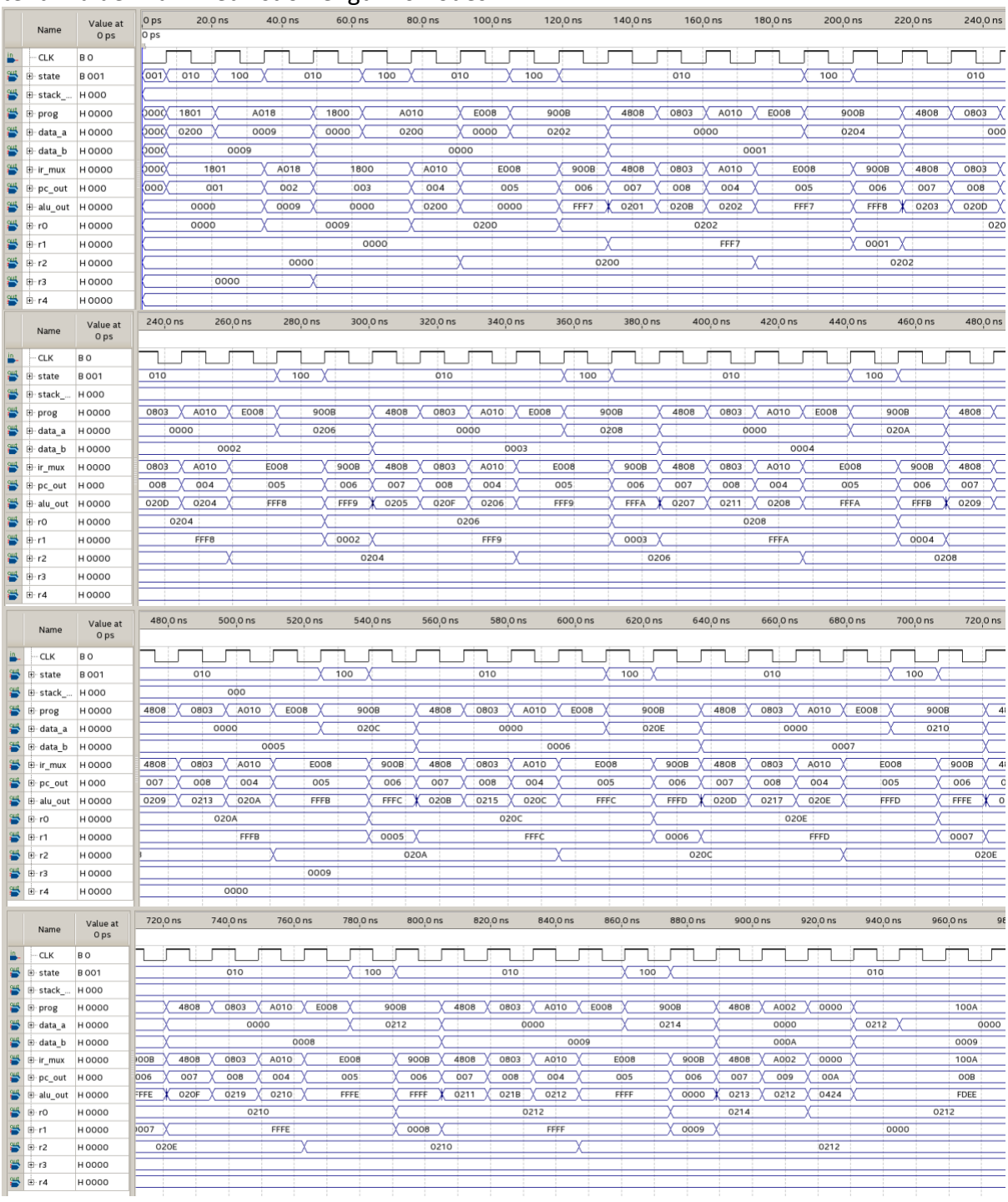
Task 2

As shown by the following waveforms, the CPU takes 58 cycles to calculate the congruential sum with inputs: a = 25385, b = 3, n = 8, s = 2.



Task 3

As shown by the following waveforms, the CPU takes 60 cycles to provide the pointer to the tenth value in a linked list of length 10 nodes



11.5 Appendix F: Resource usage for the first working design

Total logic elements 767 / 15,408 (5 %)
-- Combinational with no register 535
-- Register only 85
-- Combinational with a register 147

Logic element usage by number of LUT inputs
-- 4 input functions 208
-- 3 input functions 258
-- <=2 input functions 216
-- Register only 85

Logic elements by mode
-- normal mode 489
-- arithmetic mode 193

Total registers* 232 / 17,056 (1 %)
-- Dedicated logic registers 232 / 15,408 (2 %)

M9Ks 8 / 56 (14 %)
Total block memory bits 65,536 / 516,096 (13 %)
Total block memory implementation bits 73,728 / 516,096 (14 %)

11.7 Appendix G: Resource usage for the final design

Total logic elements 965 / 6,272 (15 %)
-- Combinational with no register 499
-- Register only 225
-- Combinational with a register 241

Logic element usage by number of LUT inputs
-- 4 input functions 474
-- 3 input functions 177
-- <=2 input functions 89
-- Register only 225

Logic elements by mode
-- normal mode 597
-- arithmetic mode 143

Total registers* 466 / 7,124 (7 %)
-- Dedicated logic registers 466 / 6,272 (7 %)
-- I/O registers 0 / 852 (0 %)

M9Ks 13 / 30 (43 %)
Total block memory bits 73,728 / 276,480 (27 %)
Total block memory implementation bits 119,808 / 276,480 (43 %)