ELEC40006 - Electronics Design Project 1 2019-2020

Option 2: Central Processing Unit

Group name: “this is fine”

- Bernard Benz
  - bfb19
  - Electronic and Electrical Engineering, Year 1
- Matilde Piccoli
  - mp2419
  - Electronic and Electrical Engineering, Year 1
- Nelson Da Silva
  - nmd19
  - Electronic and Electrical Engineering, Year 1

Word Count: 10414

Submission date: 14th June 2020

Imperial College

Index

Introduction

This report covers the results and design process of team “this is fine” ‘s EEE1 end of year project, detailing the methodology and planning behind the project as well as the implementation into Quartus and the errors encountered leading up to the final complete CPU and its operation/performance.

Outline

The task is to use Quartus Prime to design a CPU with an ISA that performs the following tasks: calculates Fibonacci numbers using recursion, calculates pseudo-random integers with a linear congruential generator and traverses a linked list to find an item. It must be designed with features ‘chosen carefully to achieve the best performance in the greatest number of applications for the smallest number of transistors.

From the specification **[2]** the following key elements were identified: A stack to keep track of intermediate variables of a recursive function, the implementation of multiplication for two sixteen-bit integers, and traversing a linked list (stored in RAM) using indirect addressing.

Additionally, the instruction words had to be encoded to provide an assembly line translation of them. No specific requirement on the architecture of the CPU was required, but it was agreed that an efficient and fast circuit was needed, but one could still be easily modifier to compute more general instructions, such as loading or storing, and more complex operations than the ones in the specification **[2]**.

Design Criteria

Approaching this task was done with Product Design Specification in mind as the CPU, despite being a digital design, emulates a real-world product and will therefore have specifications more akin to hardware rather than software.

Performance. It is essential that after the functional requirements for the CPU have been achieved, a lot of optimisation is needed to ensure this functionality is provided with minimal components, minimal power consumption and a high clock frequency.

Size. Whilst there are no restrictions on the physical size of the CPU, it is essential that the number of components is as limited as possible, as detailed by the spec which highlights the necessity of optimisation.

Quality and Reliability. The CPU must provide consistent results and should operate correctly for corner cases. It is important that after an instruction has been completed, the functionality is retained and can perform further operations with consistent outputs.

Timescale. The project was set mid-May with a deadline of June 14th. Proper planning must be implemented to ensure the CPU is functioning early on so that later, more time can be allocated to optimising and writing up the final report

Testing. A necessity that should be done regularly as it aids in identifying errors throughout the design process. It should be done thoroughly to ensures the CPU works for all possible input values.

Documentation. Proper documentation is important as half of the final grade comes from the report. Additionally, it is useful for personal use to better recount how past errors were solved. Keeping note of why changes are made help members understand why design aspects, made by others, exist.

Design Process

The plan for the design process was for individual members to be assigned one of the 3 “tasks” (operations required by the specification) to design a block which would execute it. Once tested, these blocks would be compiled into 2 similar but distinct general architectures that would run these instructions. One version was aimed to be a more flexible CPU able to run other instructions as well as more easily incorporate new instructions which used features introduced by the 3 tasks blocks, e.g. the stack or indirect addressing. This general CPU would then be converted into an optimized CPU which prioritises performance and removed all functions not related to the 3 required tasks, trading flexibility for performance. Though both versions would undergo functional analysis, the non-general CPU (performance-oriented version) would be the focus of optimizing after analysis, aiming for a higher clock speed and lower power consumption than the general CPU.

Task 1 – Fibonacci Sequence

The task was to implement the recursive function in *Figure 1* which calculates the nth term of the Fibonacci sequence using a stack which stores temporary variables created in the process. The stack could be implemented with either custom hardware or data memory.

Figure 1 – Fibonacci function in C++

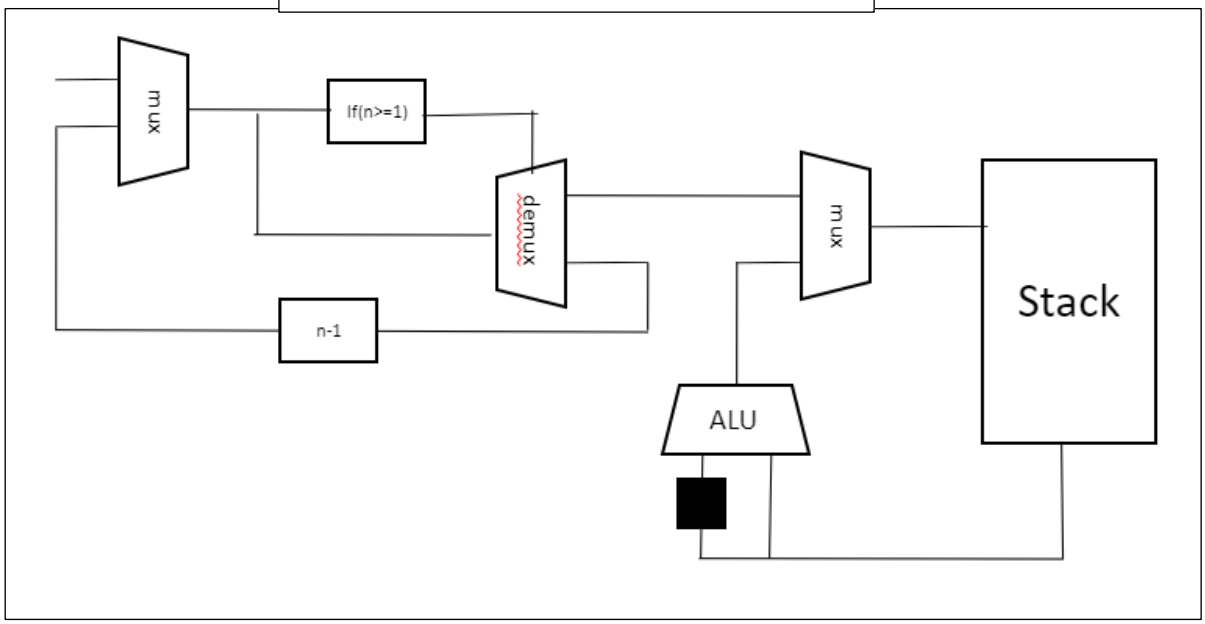
```
int fib(const int n){
     int y;
     if (n <= 1) y = 1;
     else {
          y = fib(n-1)
          y = y + fib(n-2);
     return y;
}
```

Stack memory is “a special region of your computer’s memory that stores temporary variables created by each function” **[3]**. This means whenever a fib() function is called, a location stack is allocated for this function, along with a return address, until it is completed. Then, this location in stack is freed and “that region of memory becomes available for other stack variables” **[3]**. Consequently, stack is usually referred to as a ‘last in, first out’ **[3]** data structure.

Traversing the stack requires a stack pointer, 'a register holding the address for the stack' **[1]**. A stack pointer makes it possible to choose where new data is inserted and where data must be freed up after a function is completed.

Early version used the idea of taking an input (n) and decrementing it using verilog logic until a 1 is returned. *Figure 2* shows a system where an input would determine where the returned value 1 is initially stored.

Figure 2 – a draft diagram of the stack pointer



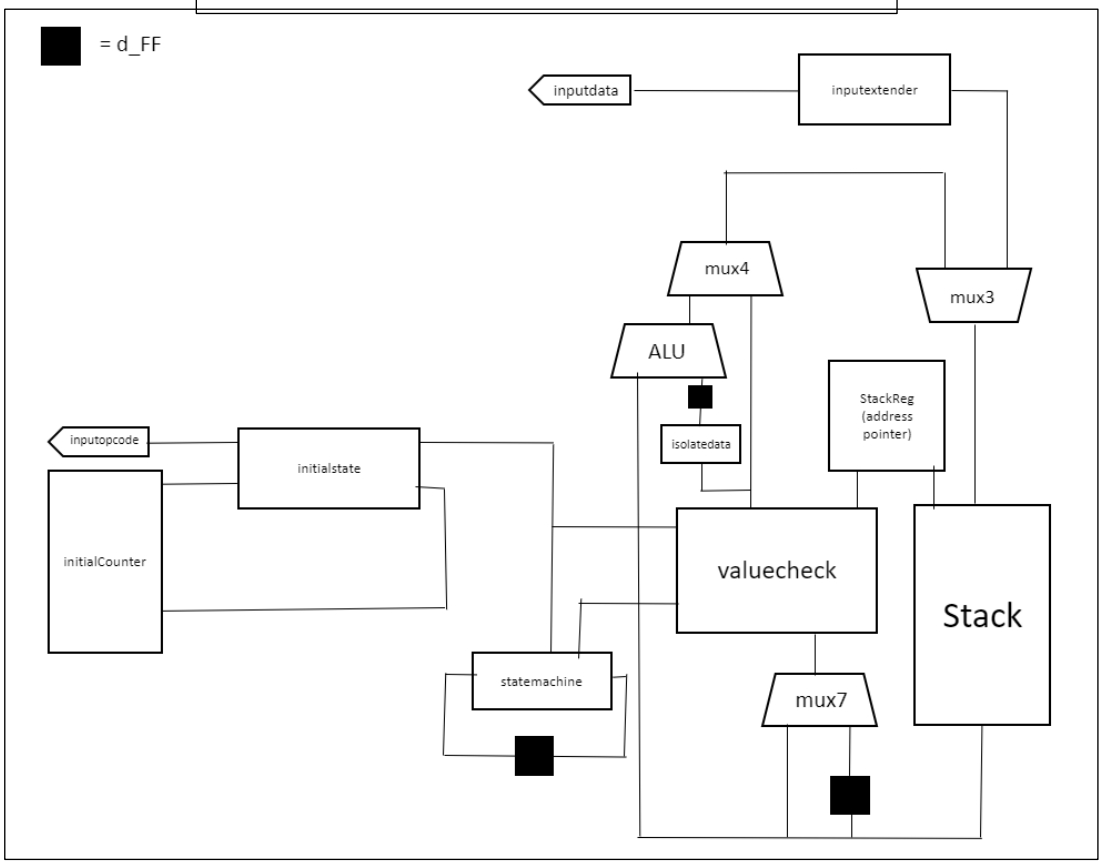
Once the initial returned variable 1 is stored, a recurring addition system where the stack pointer decrements would lead to the final returned variable being stored in 0^{th} position in stack. However, it became clear that this is not a correct representation of the given Fibonacci function (*Figure 1*), as well as not being the correct use of stack memory, despite it providing the desired results.

Additionally, despite the system's simplicity, it would not be usable for other recursive functions that could work with the architecture; this design would only be suitable for functions where repeated addition of the previous two values occurs.

Rather than storing just the individual data value for each variable, the input parameter (n) and the return address of the function, which this local variable will be returned to, should also be stored **[4]**. Despite forcing an increase in the stack word size, making this change will allow the architecture to work properly for other recursive functions with the proper adjustments.

Using this information as well as reinforcing the necessity of stack's 'last in, first out' principles created a more flexible architecture that better suited the specification (*Figure 3*)

Figure 3 – diagram for final version of the stack pointer



Note that *Figure 3* excludes some registers used to acquire inputs into the valuecheck block.

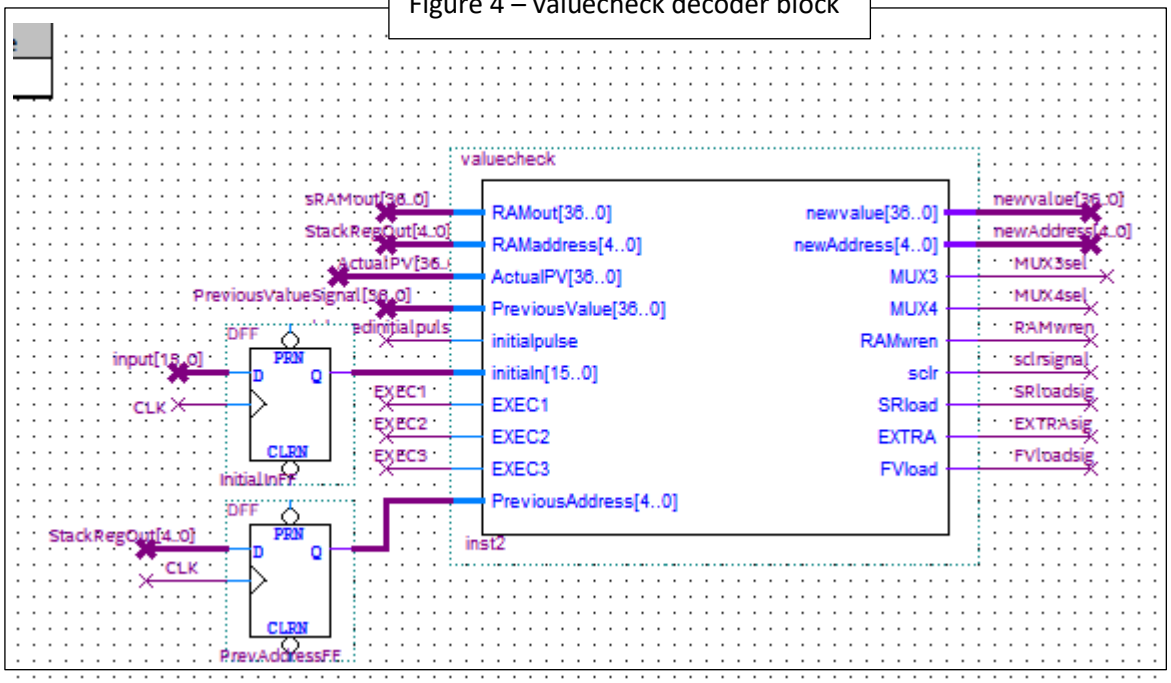
Word sizes

- Stack instruction words (48 bits)
  - Input variable n (bits 47-32)
  - Return address (bits 31-16)
  - Returned variable y (bits 15-0)
- Inputs
  - Opcode (4 bits)
  - Data (16 bits)
- Outputs
  - Final returned value (16 bits)
  - Final value pulse (1 bit)

How it worked

The decoder block called ‘valuecheck’ (*Figure 4*) handles the signals around the circuit and operates based on different ‘if’ conditions. These conditions are triggered depending on inputs from around the circuit; there are eight ‘if’ conditions in total: 1, 2, 3, 4, 5, 6, ‘initializing’, and ‘oneorzero’.

Figure 4 – valuecheck decoder block



Initialising the first value

This architecture is designed so that the operations only begin when the correct opcode is detected by the ‘initialstate’ block meaning the input data can vary whenever the instruction is not being called without affecting anything. When the opcode is detected, the ‘initialstate’ block outputs a pulse indicating that a value is being loaded in subsequently incrementing the counter called ‘InitialCounter’. The counter increments again causing another pulse to output from the ‘initialstate’ block. The counter increments again but no pulse is sent out at this point, remaining at a value of three until the final value has been calculated.

The two-cycle pulse sent out by the ‘initialstate’ block (called ‘initialpulse’) is input into the ‘statemachine’ block which starts a two or three cycle system (depending on current conditions) used to calculate and write in the next value into the stack.

This two-cycle pulse is also input to ‘valuecheck’, triggering the ‘initialising’ condition in the ‘valuecheck’ block, causing the following:

- Cycle 1 – MUX3 selects the input parameter as the input data into the stack.
- Cycle 2 – This value is written into the stack’s 0^{th} address

Before the input variable can be written to stack, it is extended to match the world length of the stack using the ‘inputextender’ block which fills the 21 LSBs with 0s.

Working towards the final value

Once this variable has been input into the 0^{th} position of stack, the ‘valuecheck’ block identifies what set of signals to output to the circuit and calculates the new value to store in the RAM. From the initial value onward, the resulting actions are dependent on the input. For an input of one or zero, the ‘valuecheck’ triggers the ‘oneorzero’ condition designed for when the input parameter (n) is one or zero and the expected result is one:

- Cycle 1 – Produces the output data value of 1 and writes this into the stack’s 0^{th} address.
- Cycle 2 – Storing the previous output’s data value of 1 into the register that stores the final calculated value. The stack’s 0^{th} address is then overwritten with a null value so that the stack is ready for a new Fibonacci function.

However, if the input parameter (n) is not one or zero, condition ‘3’ is triggered calling fib(n - 1):

- Cycle 1 – The stack pointer is incremented since calling a new function means another spot in stack is occupied.
- Cycle 2 – The new output value is generated; this is the previous value with the input variable decremented and the return address adjusted to call the previous function. This new value is written into the RAM in the same cycle.

Note that for the following tables, all values are in hexadecimal

current function line	**function calls**	**stack address**	**value of n**	**return address**	**data value**
fib3	fib3	0000	3	(0000) by default	0000
y = fib2	fib2	0001	2	0000	0000
y = fib1	fib1	0002	0001	0001	0001

Once a point has been reached where the input variable (n) is one ore zero, the condition ‘1’ is met where the data value of the return address is incremented once:

- Cycle 1 – A data value set to 1 is generated. The ‘isolatedata’ block (*Figure 3*) sets the rest of the word excluding this data value to zero. The stack pointer is updated with the return address so that this value will be read out in the next cycle.
- Cycle 2 – The value read out from the return address is added to the output of one, from the ‘isolatedata’ block, using the ALU. Additionally, MUX4, the multiplexer controlling inputs to stack is made to take inputs from the ALU
- Cycle 3 – This new value is written into the stack to overwrite the return address value.

MUX7 is needed for these 3-cycle conditions so that the input to the ‘valuecheck’ block does not change at the 3rd cycle. Once the 3rd cycle has been reached, the signal EXEC3 is used as a select line for MUX7 so that it switches to a delayed value of the stack output. This ensures that 3-cycle conditions are fulfilled.

current function line	**function calls**	**stack address**	**value of n**	**return address**	**data value**
fib3	fib3	0000	3	(0000) by default	0
y = fib2	fib2	0001	2	0000	1

The ‘valuecheck’ block then determines that a fib(n-1) function has just been completed and the fib(n-2) function has yet to be called. Condition ‘4’ performs the new function call:

- Cycle 1 – Generating the new value where n has been reduced by 2, the return address has been set to the previous address and the data variable is cleared. The stack pointer is incremented.
- Cycle 2 – This new value is written to stack.

current function line	**function calls**	**stack address**	**value of n**	**return address**	**data value**
fib3	fib3	0000	3	(0000) by default	0
y = fib2	fib2	0001	2	0000	1
y = y + fib(0)	fib0	0002	0	0001	0

Condition ‘1’ is met again in this transition.

current function line	function calls	stack address	value of n	return address	data value
fib3	fib3	0000	3	(0000) by default	0
y = fib2	fib2	0001	2	0000	2

The ‘valuecheck’ block also knows if a fib(n-2) function has just been completed, now the data variable of the original function must be input to the return address. These instructions come under the if condition ‘5’:

- Cycle 1 – Data value is isolated using the ‘isolatedata’ block. The stack pointer also updates with the return address
- Cycle 2 – The value from the return address is read out and added to the previous data value using the ALU. MUX4 is set to direct the ALU output into to the stack.
- Cycle 3 – The ALU output is written into the stack.

current function line	function calls	stack address	value of n	return address	data value
fib3	fib3	0000	3	(0000) by default	2

Since fib(2) was the fib(n - 1) function of fib(3), the condition where fib(n - 2) must be completed is triggered again (‘4’).

current function line	function calls	stack address	value of n	return address	data value
fib3	fib3	0000	3	(0000) by default	2
y = y + fib1	fib1	0001	1	0000	0

current function line	function calls	stack address	value of n	return address	data value
fib3	fib3	0000	3	(0000) by default	3

The final value is stored in the 0th address and the 16 relevant bits are stored in a separate register. The 'valuecheck' block detects this through the condition '6':

- Cycle 1 – A signal is sent to the 'sload' input of the 'final value' register (*Figure 3*) which allows a value to be loaded in. This same signal is output from the block to indicate that the final value has been calculated. A null value is also generated to overwrite the 0th address to bring the block into a 'rest' state. The counter from the beginning is reset to allow for a new input instruction.
- Cycle 2 – This null value is written into the stack.

The rest state/condition '2' is used when nothing is happening with the fib block thus, there are no outputs or changes in the circuit. In both cycles, all signals are set to zero so no values can be written in making the circuit ready for a new input.

Evidence of functionality

Figure 5 – test waveform

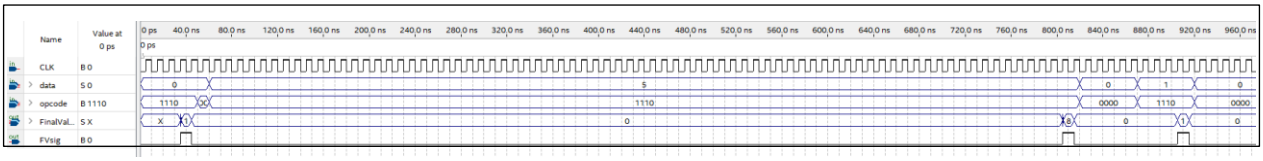
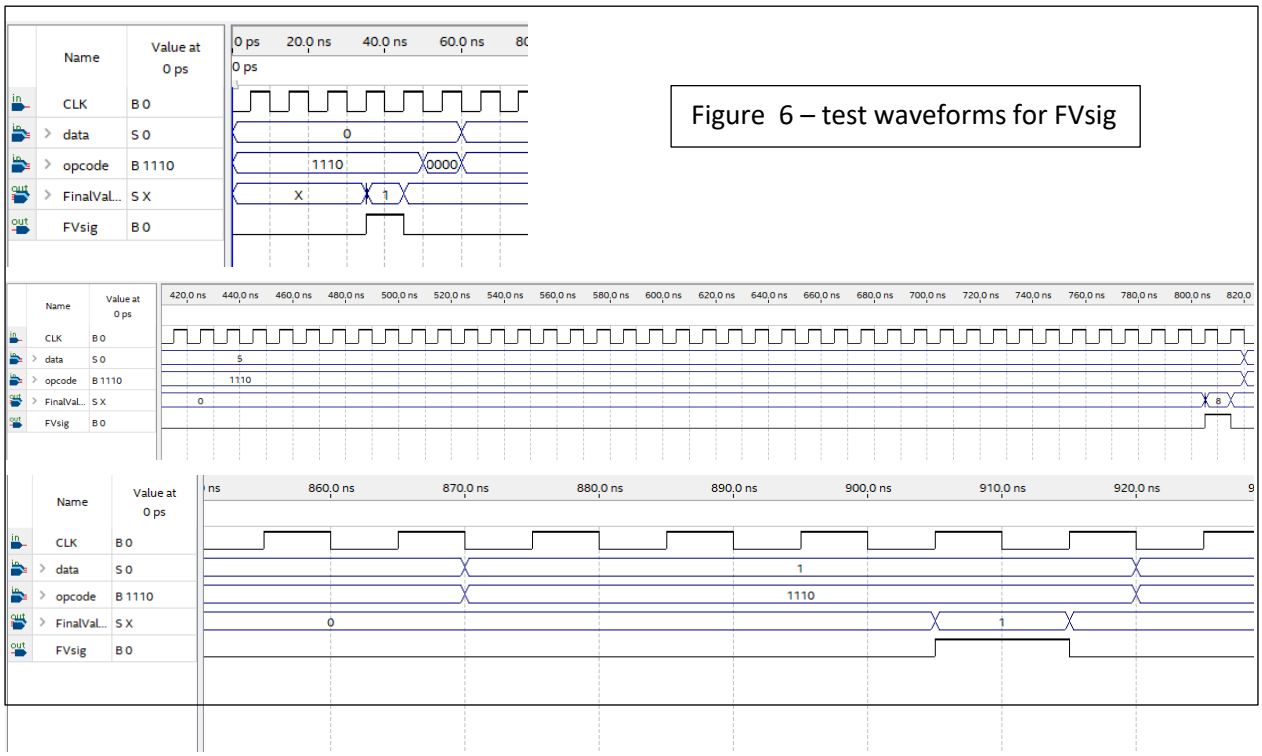


Figure 5 test inputs of n: 0, 5, and 1. Note that once the final value has been calculated, the opcode switches; this switch is causes by the rest of the CPU.

Figure 6 – test waveforms for FVsig



All the test results are correct (*Figure 6*) and when they are calculated the FVsig output pulses correctly.

The current architecture allows flexibility when implementing other recursive functions due to only needing to make modifications to the 'valuecheck' verilog such as the conditions and the signals sent out. However, recursive functions that do not solely use addition will need additional hardware.

Optimising Task 1 – Fibonacci Sequence

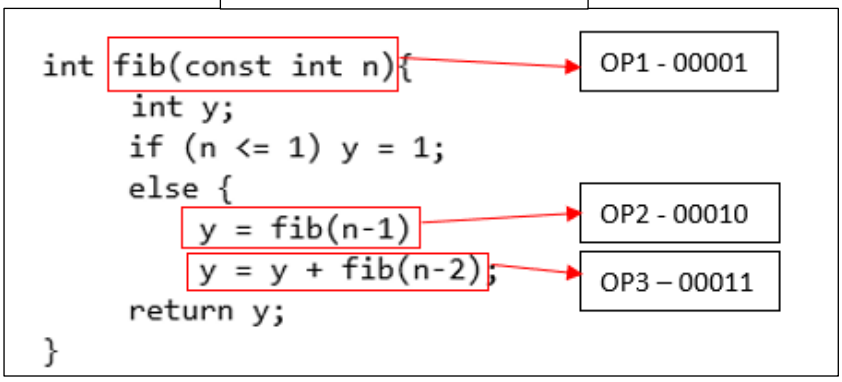
Planning

The implementation of the Fibonacci task was done in 2 different ways. The method detailed above was used in the general CPU whilst the other was optimised to be more task specific and less flexible. This version will be the discussed in the following section.

The process of designing the block, similarly to the other Fibonacci block design process, began with research about the stack and how it typically works as this was the main feature this task introduced **[1]**.

In the case of the recursive function from the specification (*Figure 1*), the stack would need to contain the current value of n as well as the current value of y, both of which are sixteen-bit integers.

Figure 7.1 – Opcode format



Once this was established, the focus was directed to how the stack would look like during operation, including the values contained and word size, and how the stack pointer changes during operation. The initial conclusion was that the C++ function given (*Figure 1*) had 3 possible outcomes each time a new n was inputted: either n was one or zero in which case the return value is one, n is greater than one and the function calls on itself with the new input n being one less than the current n, or 'y = fib(n - 1)' has been performed and now 'fib(n - 2)' is being added on. From this came the first alterations to the general implementation of stack to make it more optimized to solely run this task, the removal of return address', using instead an opcode to indicate which of the three positions the function was currently in (Figure 7.1). Figure 7.2 briefly illustrates how the opcodes would interact during runtime.

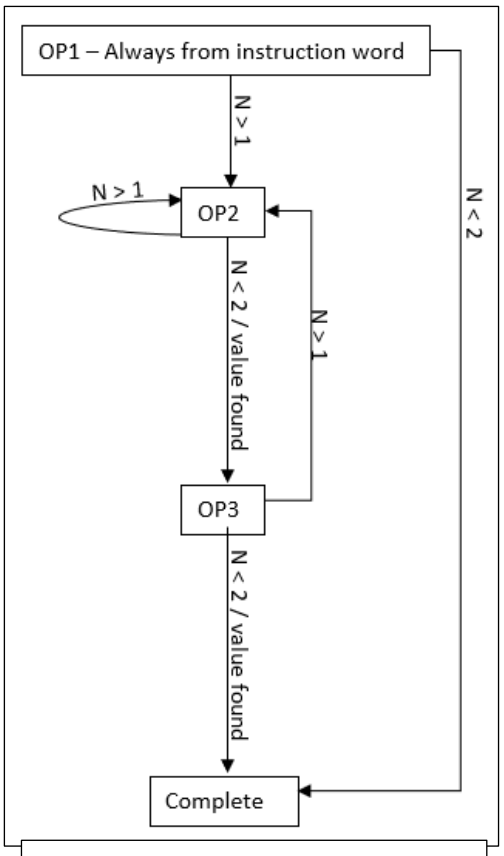
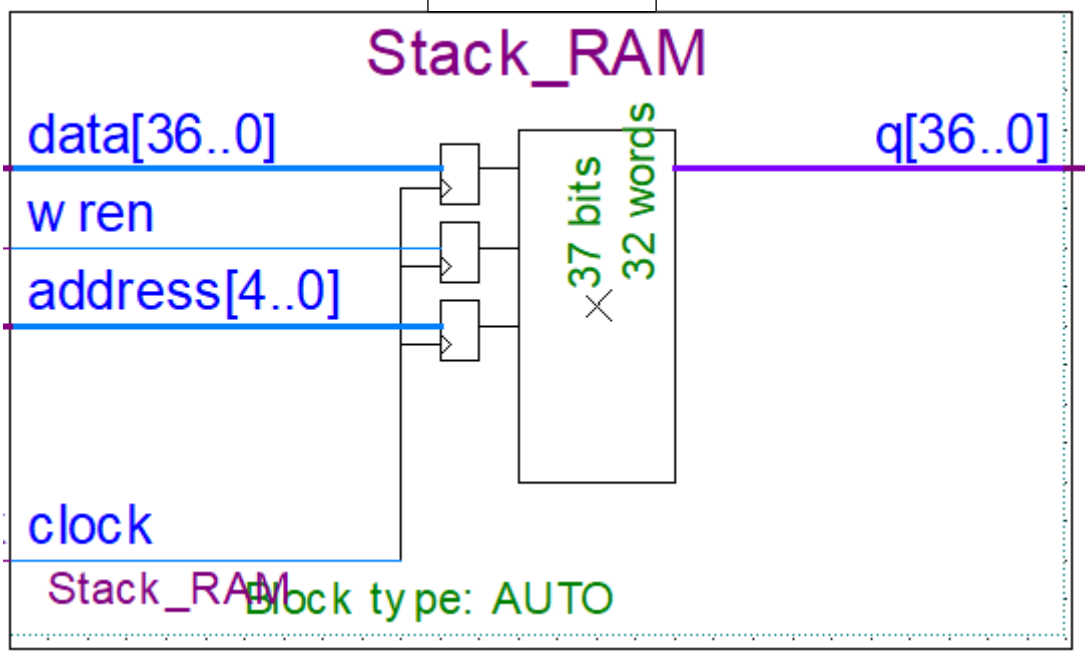


Figure 7.2 – diagram of opcode use

The word length of the opcode is five bits so that the first operand could come directly from the instruction word, which is separated into five bits of opcode and eleven bits which contained the address of the starting value of n stored in the data RAM. From this the stack word size would have to be thirty-seven bits to accommodate for the two sixteen-bit integers (n and y) and the five-bit opcode. This meant it would be better to keep the stack separate to the data RAM, which had a sixteen-bit word length **[2]**, so that all the variables (OP, n, and y) could be fetched in 1 cycle. The stack was implemented using a 1-port RAM with 32 words (*Figure 8*).

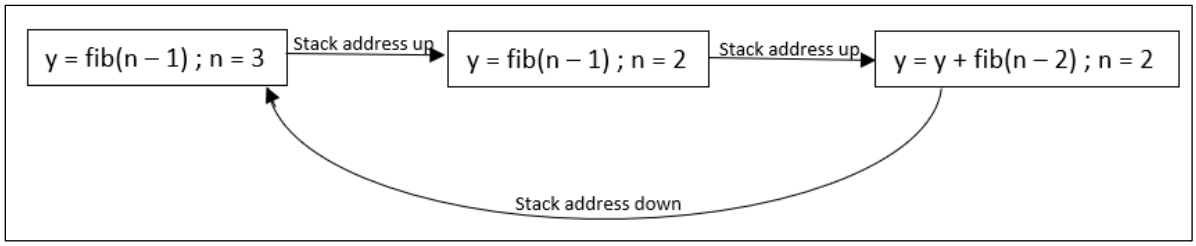
Figure 8 – Stack



After it was clear what the stack would contain during operation, the new focus was how the stack pointer would traverse this information. The address contained in the stack pointer could either increase or decrease; in the case of it increasing it would only ever increase by 1 when new items were added to the stack; This is easily implemented by incrementing the value of the stack pointer. The stack pointer would decrease only in a specific case where it completes the following set of recursions shown in *Figure 9*. From this it is inferred that in cases where the stack pointer decreases in value, it does so by two. From here all the possible stack pointer operations needed were an increase by one or decrease by two, and a reset function all tasks are complete, effectively resetting the stack size to zero. At this stage, implementing and testing in Quartus began.

implementation

Figure 9 – Situation needed for stack address



The stack pointer (*Figure 10*) was implemented using a counter and an adder; The counter would increase when the ‘up’ input was high as it was connected to the count enable of the counter. Decreasing the address of the counter by 2 was done by connected the ‘down_two’ input to the ‘sload’ port of the counter and having the input address as always equal to the current address minus 2.

Figure 10 – Stack Pointer

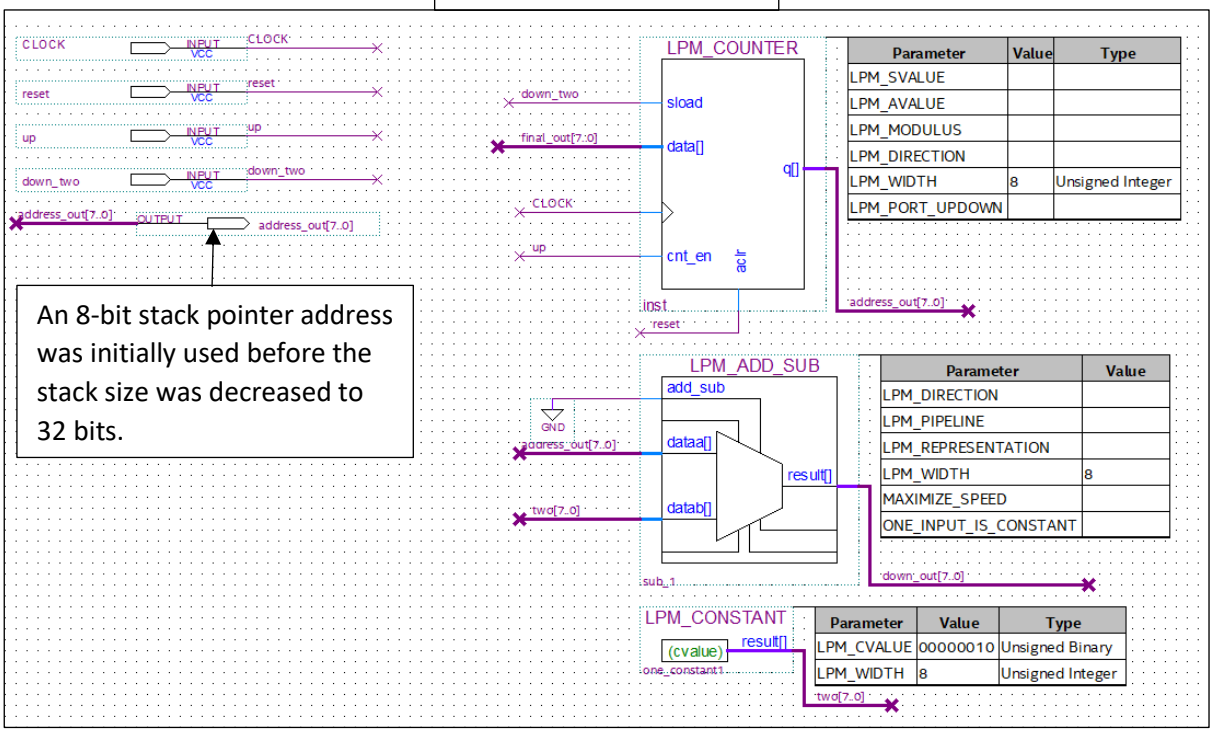
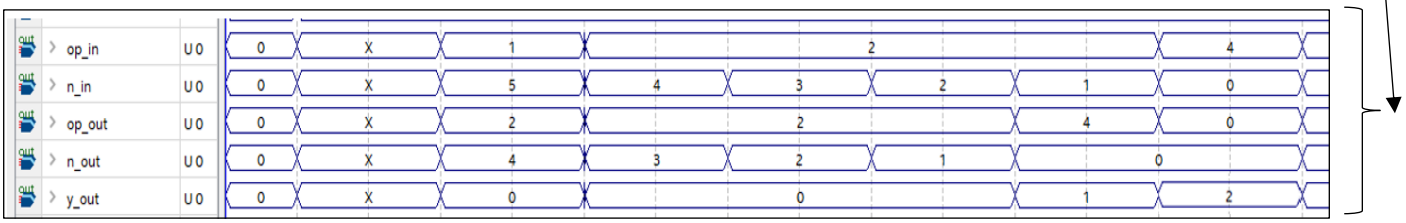


Figure 11 – Incorrect Waveform for fib(5)

Output was X here onward



The first error encountered is seen in *Figure 11*. The problem was that the function was not returning to previous recursions correctly, it would only continue to perform OP2's (*Figure 7.1*) until the current value of 'n' was less than two, at which point it performed an OP3 (*Figure 7.1*) meaning that any initial input of 'n' yielded an output y of two. It was clear that there needed to be a way to distinguish when an item in the stack had been completed and the stack pointers value had decreased. This was done by inserting in a delayed input of the 'down_two' signal back into the main operating block, resulting in different operations when there was a 'down_two' output last clock cycle.

Figure 12 – Verilog when down_in is high

```
else if ( (op_in[4:0] == 5'b00010) && !ONE && !down_in) // y = fib n - 1 where n - 1 !<= 1 without previous down
   begin
   OP_OUT[4:0] = 5'b00100;
   N_OUT[15:0] = n_in[15:0] - 16'b0000000000000001;
   Y_OUT[15:0] = carry_in[15:0];
   UP = 1;
   end
```

Figure 12 is the Verilog code for the situation depicted in *Figure 9* after the stack address is decreased by two. It also meant that somehow when the items were complete, the y value needed to be stored and inserted back into the old instruction the stack pointer is pointing at, which resulted in the use of a carry in and carry out mechanism.

Figure 13 – carry_out delay block

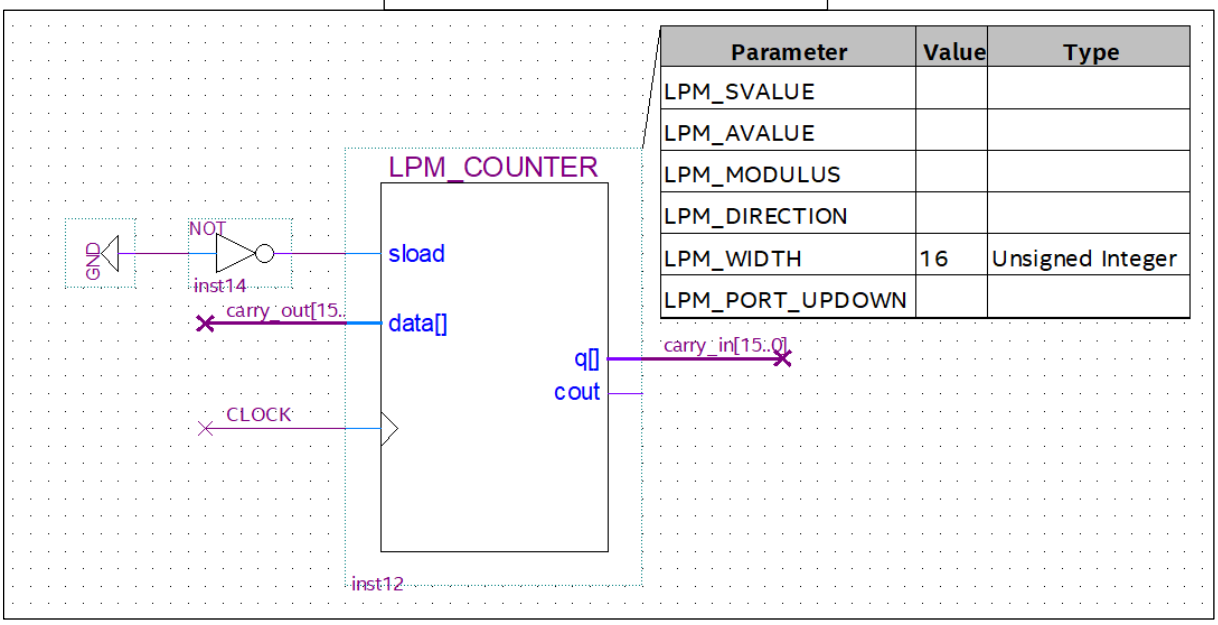


Figure 13 works by taking the y output that needed to be re-inputted into the main operating block (carry_out) and inputting it into a counter which would then update at the next clock cycle. This meant that the 'carry_out' value would be inputted as the new 'carry_in' value one cycle later when the stack address is lowered. This value would then be used as seen in *Figure 12*.

At this point in time some changes to the original opcode format for the task were made this new format is explained in *Figure 14*.

Figure 14 – New opcode format

Opcode name	Binary value	Significance
OP1	00001	Used for the initial fib(N) instruction directly from the instruction RAM
OP2	00010	Used for the first iteration of y = fib(n – 1) where n = N
OP3	00011	Used for the first iteration of y = y + fib(n – 2) where n = N
OP4	00100	Used for the all other iterations of y = fib(n – 1) where n ≠ N
OP5	00101	Used for the all other iterations of y = fib(n – 1) where n ≠ N

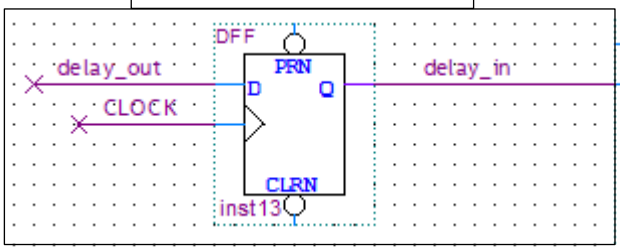
Once these changes were made a timing issue was spotted. There was a delay between the changing of the stack pointer address and when the correct set of data would be outputted. This was previously unnoticed since the settings of the RAM used for the stack were such that when the address is changed and the stack is being written to, the output of the stack would always be the new

value that was being written to it meaning there was no delay between when the value was inputted and when the stack outputted the value. This however meant that when the address of the stack pointer was changed and the stack was not being written to, the stack would take an extra cycle to output the correct values.

This was solved by implementing a general delay which meant that the main operating block would only change its outputs every other cycle to ensure it was receiving the correct outputs from stack. *Figure 15* shows how this delay was implemented. When 'delay_in' is high, the main operational block does nothing and 'delay_out' is set to low; when 'delay_in' is low the block would operate normally and 'delay_out' would be set to high.

This meant that several input variables as well as write enables also had to be delayed for everything to operate at the correct timing. This resulted in a doubling of clock cycles needed to complete a fib(n) instruction.

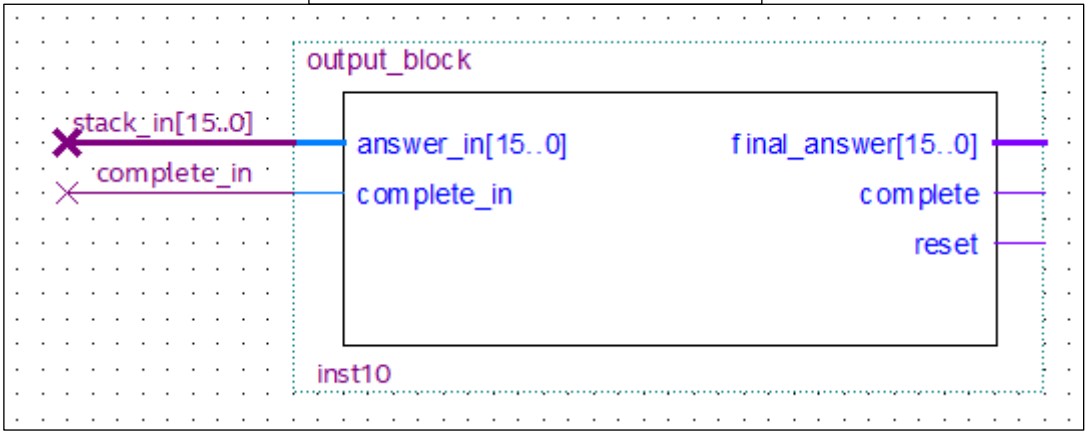
Figure 15 – delay flip flop



Now operation was fine for a single execution of the Fibonacci function**.** Additions to the circuit were made to allow for multiple executions of the function to be made in a row.

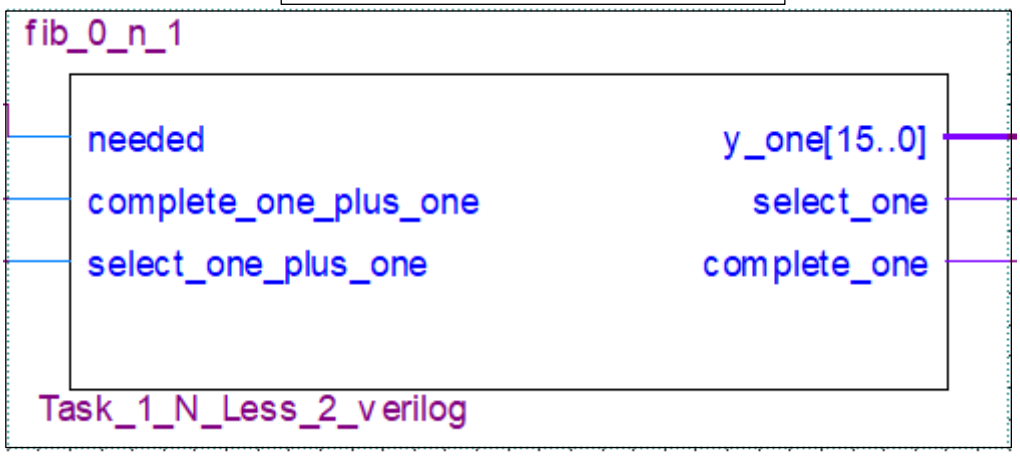
Figure 16 shows an output block for the Fibonacci task which takes inputs from the main operating block and had outputs that included: a 'complete' output signal that indicates when the main operating block finishes executing its current instruction, the 'final_answer' to the current instruction, and a 'reset' output so the stack pointer points to address 0. This was implemented using a verilog file.

Figure 16 – Fibonacci output block



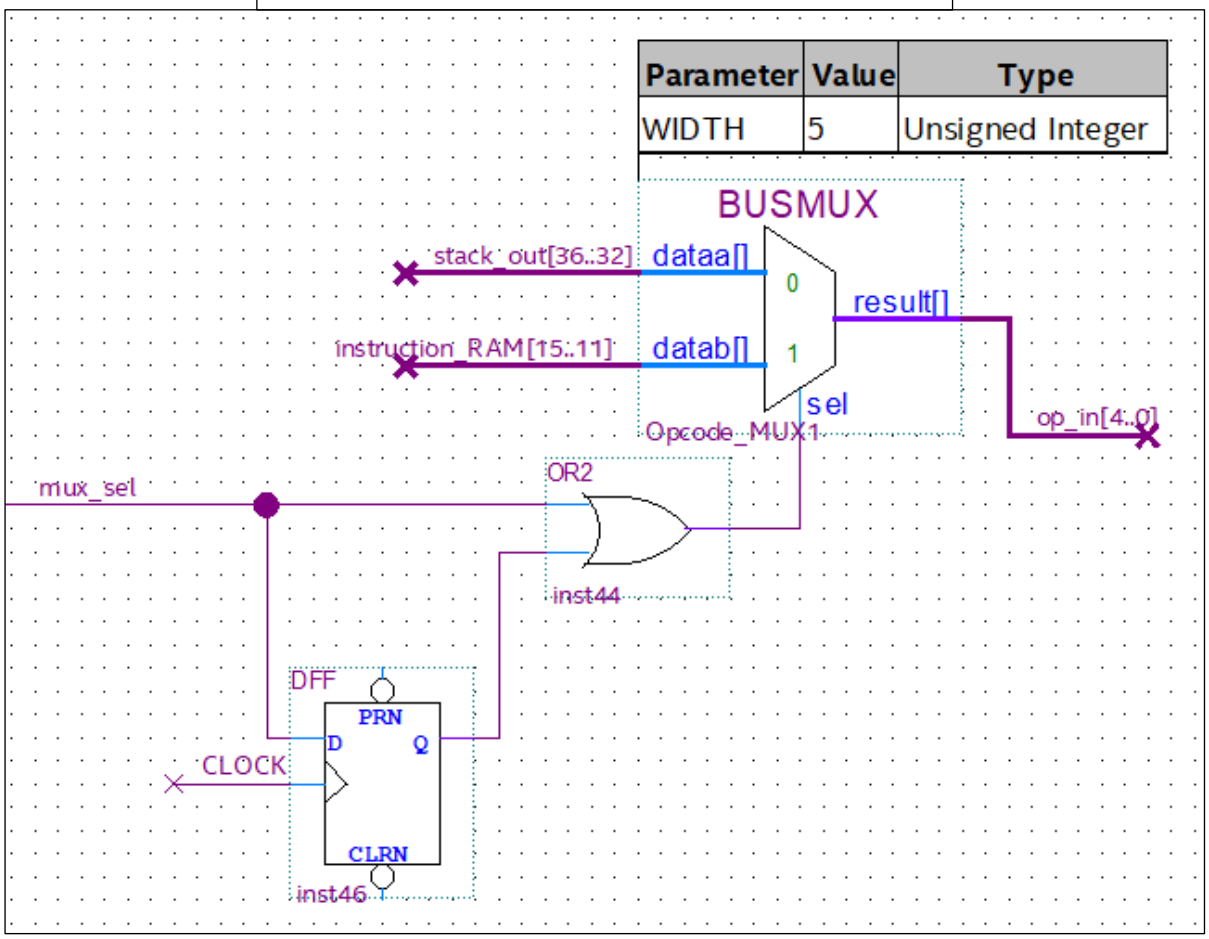
A problem arose when automating the fetch cycle of the Fibonacci block (this is discussed later in the section regarding the optimized complete architecture) for it to take multiple inputs in a row. The error was that it would not work for initial values of ‘n’ that were less than two, which did work prior to automation. This was solved by introducing a separate operating block for cases where the initial ‘n’ was one or zero (*Figure 17*) which had an enable input, that was high when an ‘n’ coming from the data RAM was less than two, and had output: ‘y_one’ (which was always equal to one in sixteen-bit binary), ‘select_one’, and ‘complete_one’. As a result, BUSMUX’s had to be connected to the inputs of the output block (*Figure 16*) in order change whether it took ‘answer_in’ from the main operating block or the block in *Figure 17*.

Figure 17 – Block for fib(n) where n < 2



After testing another problem was identified which was a consequence of the delay mechanism (*Figure 15*); The block was in effect just a clock with half the cycle speed of the actual clock. Consequently, if the main operating block would receive the initial fib(n) instruction, from the instruction RAM, when ‘delay_in’ was high, the whole Fibonacci block would not work. This was solved simply by extending the time in which the initial inputs of the main operating block come from the instructions register (*Figure 18*).

Figure 18 – BUSMUX for op_in of main operating block



Final operation

Figures 19 explain the operation of the complete Fibonacci block.

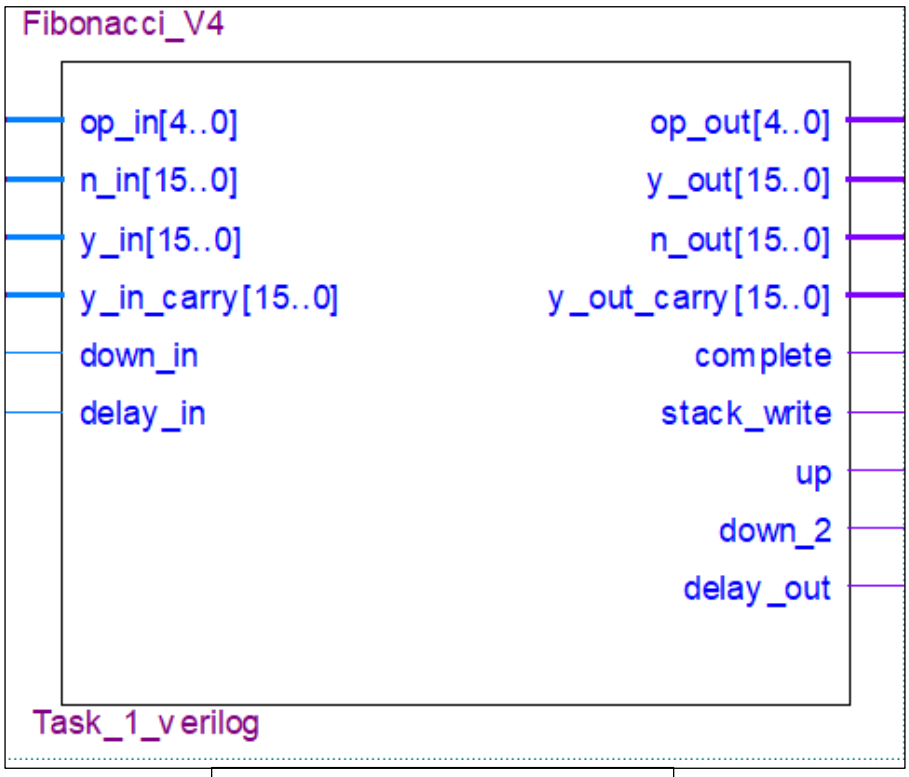


Figure 19.1 – main operating block

Figure 19.2 – Simplified diagram of Fibonacci block

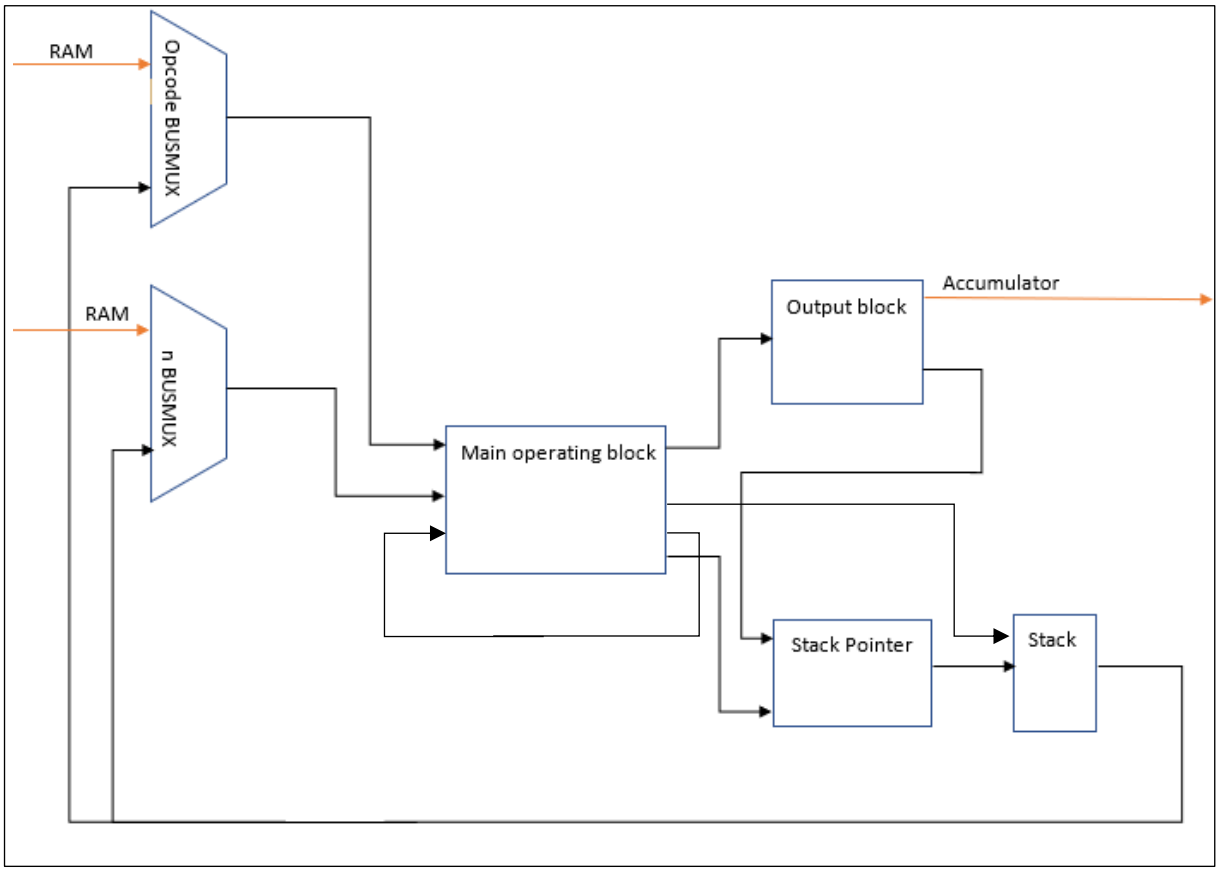
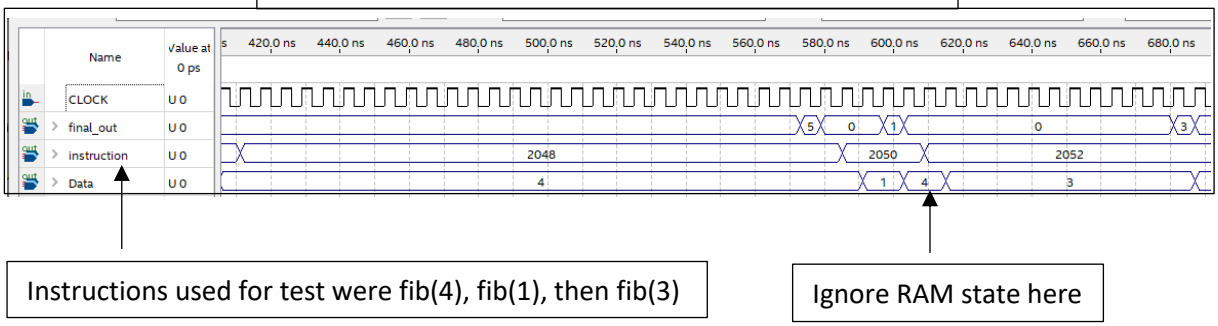


Figure 19.3 – Table of operation for Figure 19.2

Component Name	Reference Figure Number	Function
Opcode BUSMUX	13	Used to change where the main operating block gets op_in from
n BUSMUX	13	Used to change where the main operating block gets n_in from
Main operating block	14.1 & 12	Describes figures 12 and 14.1 combined and is used to alter the value of the stack pointer. Also outputs the value to be written to stack. And controls the output block.
Output block	11	Outputs the final answer and a “complete” signal the cycle the fib(n) instruction is complete
Stack Pointer	5	Contains the current address to stack
Stack	3	Used to store intermediate values during operation

Final tests were conducted to illustrate that the block worked for single and multiple instructions in a row, as shown in *Figure 20*, and was now ready to be implemented into the complete architecture. The main cause of problems throughout the design process of the optimized Fibonacci block came from the physical restrictions of hardware components, either in the form of delays of clocked components or the difference between the theorised operation and actual operation of certain blocks. Overall, no major problems arose due to time spent planning the process and designing on paper before implementing anything into Quartus.

Figure 20 – waveform of working Fibonacci instructions



Instructions used for test were fib(4), fib(1), then fib(3)

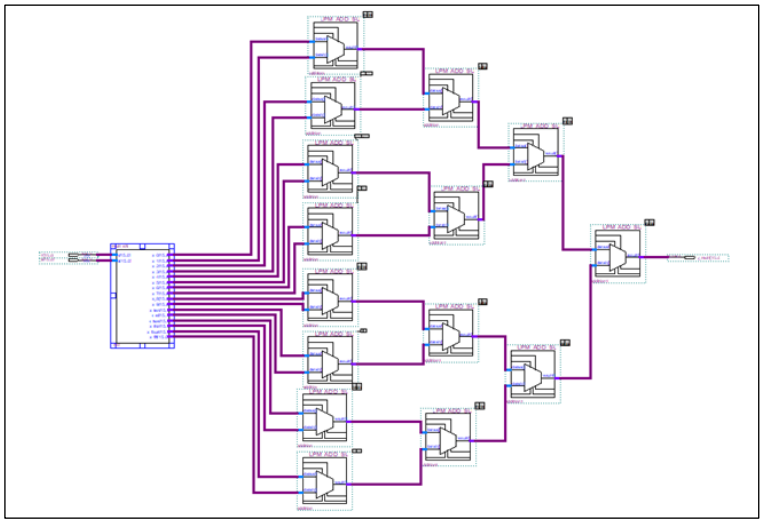
Ignore RAM state here

Task 2 – Random Number Generator

The Multiplication Block

This block computes binary multiplication by performing series of shifts and additions. This method can perform multiplication of sixteen-bit integers in a single cycle as it pipelines the adders. The resulting circuit (*Figure 21*) is composed of two parts. The first a Verilog file performing the shift; One of the two multiplicand word is taken as argument (A) and the other (B) is used to decide when to perform the shift on the argument. Every time a bit of B is high, a shift left is performed 'N' times on a copy of 'A', 'N' being the position of the high bit in 'B' (if the 'N^{th}' bit of B is 0, the copy will just be set to zero. The second part consists of a pipelined series of adders adding two of the shifted copies of A at time. Since the Verilog block and the adders are not clocked, the process is not subject to any delay and the result will be ready in the next cycle. Since word length of the integer variables used is limited, if the resulting number exceeds sixteen bits, the result will be different from the calculated expected.

Figure 21 – multiplication block



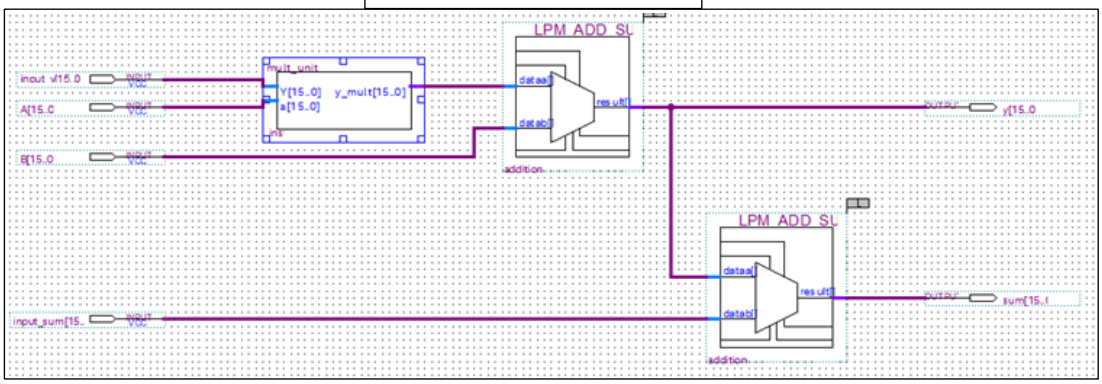
Random number generation logic

Figure 22 – code from the specification

```
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; }
```

At first, the linear congruential generator was designed manually, then implemented on Quartus. The circuit was designed to execute the above code (*Figure 22*) using the multiplication unit and two addition units (*Figure* 23) one implementing 'y = y*a + b' and the other 'sum = sum + y'. A block was created to implement these additions and was implemented in the final task 2 circuit.

Figure 23 – addition unit



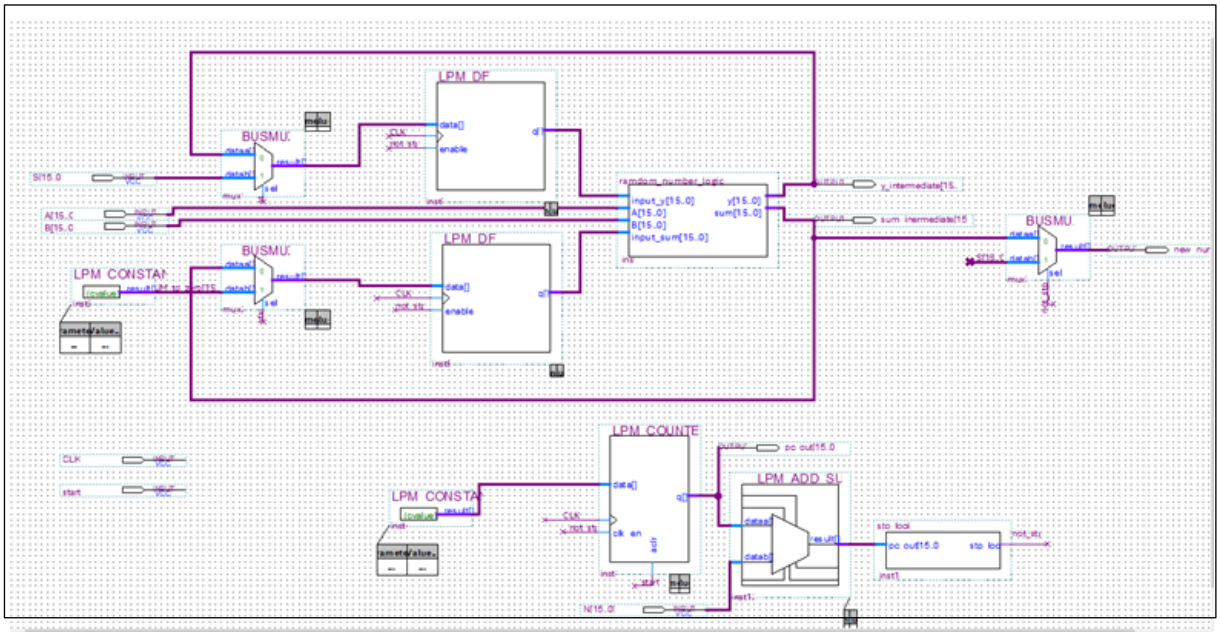
Complete Task 2 Circuit

The final circuit used a loop connecting the outputs ‘sum' and ‘y' to the corresponding inputs. Two multiplexers were placed to choose the starting values of y (input) and sum (zero when initialised) for the first cycle of the loop, when a start input for the whole task is high. Another MUX was used for the output stage selecting zero unless the complete output of the task is high, in which case the MUX outputs the final value of ‘sum’.

An additional block was included in the complete task 2 circuit (*Figure* 24) that sets the maximum number of loops allowed (given by the input ‘N’). The final circuit was made with a Counter and an Adder that subtracted ‘N’ from the output of the counter. When the difference between the two values is zero, a complete output is produced by a Verilog block, disabling the counter, and enabling the output stage of the task. The start output is also used to clear the counter so that is set again to zero at the beginning of the next instruction.

After testing the circuit noticed that the main operating block had undefined behaviour. This was solved by clocking the inputs to this block using DFFs ensuring that the block would only change outputs once a clock cycle.

Figure 24- complete task 2 circuit



Final Testing

The task was compiled into a single symbol and was tested to check the functionality with given inputs. The inputs required by the circuit are the values of ‘A’, ‘B’, and ‘S’ (from the Data RAM) and ‘N’ (set to 16 according to the specification **[2]**), the clock input and the start input (later automated using a state machine). The outputs produced are the final ‘sum’ value and a ‘complete’ output, that will be used increment program counter. Other additional test outputs were made to check intermediate values during operation.

After testing the circuit with different inputs, it was confirmed that when the start input high for one cycle (excluding the first cycle) instruction starts to be executed and the loop continues (*Figures 25*). This is shown by the changes in the intermediate values and stops when counter reaches the maximum value (sixteen cycles). Comparing the final output with the manually calculated answer confirmed the circuit is working as expected. *Figures 25* are the two most relevant tests results which confirm the behaviour previously explained. The task 2 circuit was now ready to be implemented into both CPUs since it already optimised.

Figure 25.1 – test waveform

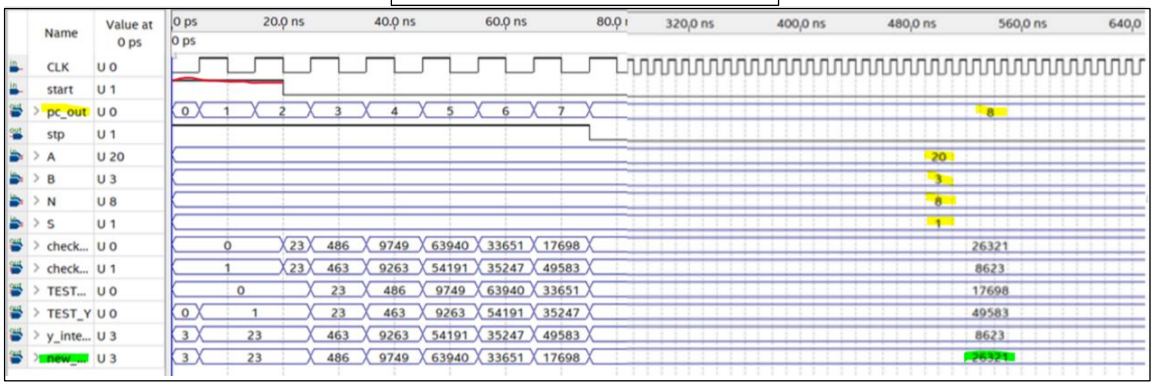
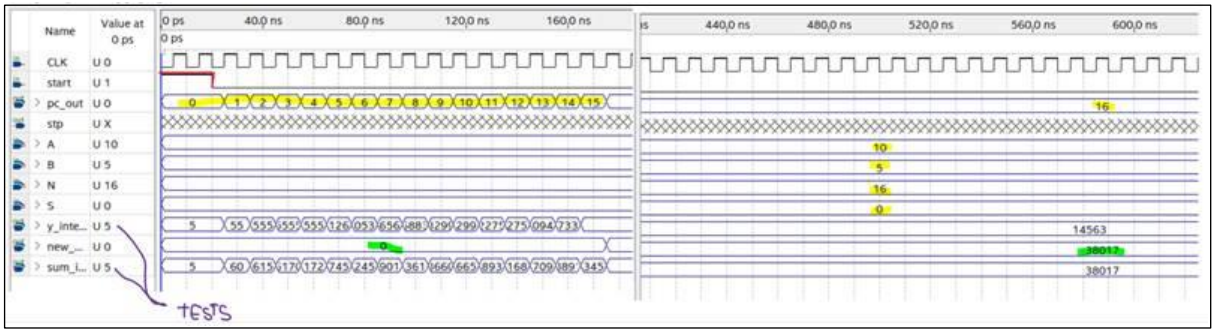


Figure 25.1 – test waveform



Task 3 – Linked List Search

Selection an Implementation Method

Multiple implementations were initially considered before settling on the following method. The linked list was implemented by storing both elements (value and next pointer) of an item into the same memory location using a twenty-eight-bit wide data RAM. The 12 MSBs of the word contain the address of the next pointer and the 16 LSBs contain the value. This method to be simpler in terms of implementation and more effective since it does not require any new piece of hardware.

Indirect Addressing and Registers

A register was needed to store the memory content and the current address while traversing the linked list. It was decided that an indirect addressing block would be in the final general CPU which allows for different operation on linked list (e.g. inserting a value). It was noticed that the task itself does not require indirect addressing since a comparator can be used to know if the search value is not found, the 12 most significant bit of the data out can be directly fed into RAM. Consequently, the task used a register to control when the address input into the RAM changes, and a comparator that produce ‘complete’ output if the value coming out of the RAM is the search value.

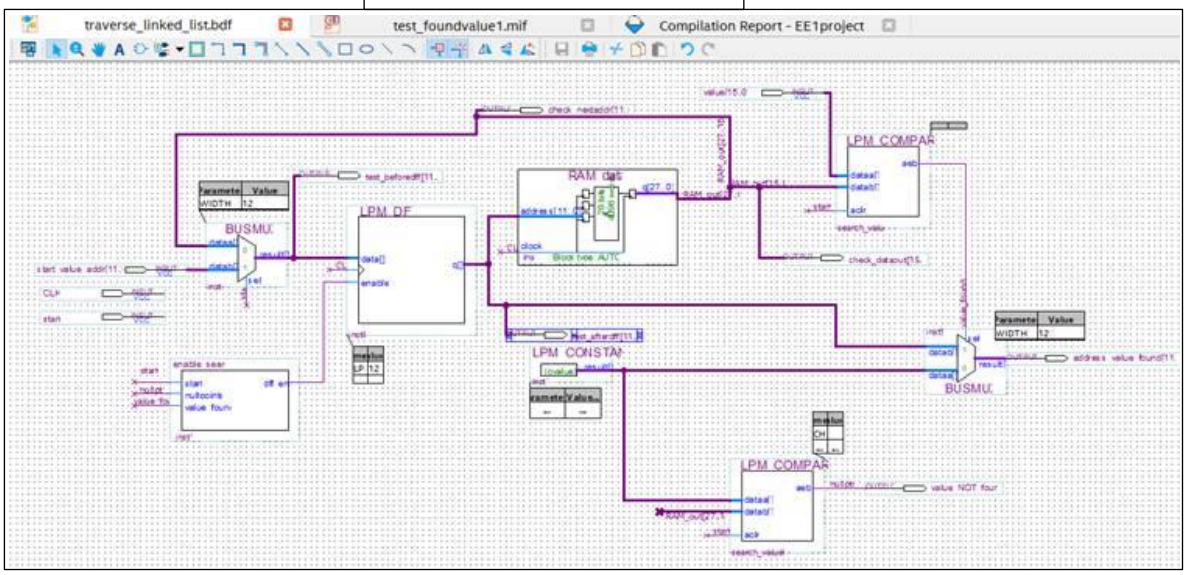
Control inputs

The output stage consisted of a multiplexer that outputs the current address, if the value at that address corresponded to the search number, or a series of zeros otherwise. If the address of the found value is zero, it could be misinterpreted if the value were found or not. To solve this a signal was made indicating when the task is done so that the CPU can recognise when the task is and the found address is zero.

A block was made to identify when the next address is a null pointer, this produces an output (value_not_found) that stops the search by disabling changes in the register containing the address and enables the CPU to execute the next instruction. The ‘complete’ signal comprised of both the ‘value_found’ and the ‘value_not_found’ signals.

After running initial tests, it was identified that very first cycle of the instruction the complete output signals was high. To avoid the block ending the instruction before even beginning to execute, a verilog block was implemented that when the enable is high whenever start is high or, both ‘found’ outputs are low. The start input is also used to clear the comparators. The final circuit is seen in *Figure 26*.

Figure 26 – complete output



Final Testing

The circuit was compiled into a single symbol file and connected data RAM. The initial inputs needed by task block are the starting address and the search value a clock signal and the start value. The final outputs are the memory address where the value can be found and the two ‘complete’ signals.

A MIF file was made to test the case in which the value is found (*Figure* 27), and one that test a list that leads to the break condition/null pointer (*Figure* 28). Additional outputs were added to check the entire functionality. The two simulations showed the expected outputs both in terms of the final address and the behaviour during operation.

Figure 27 – test MIF file “value exists” and corresponding waveform

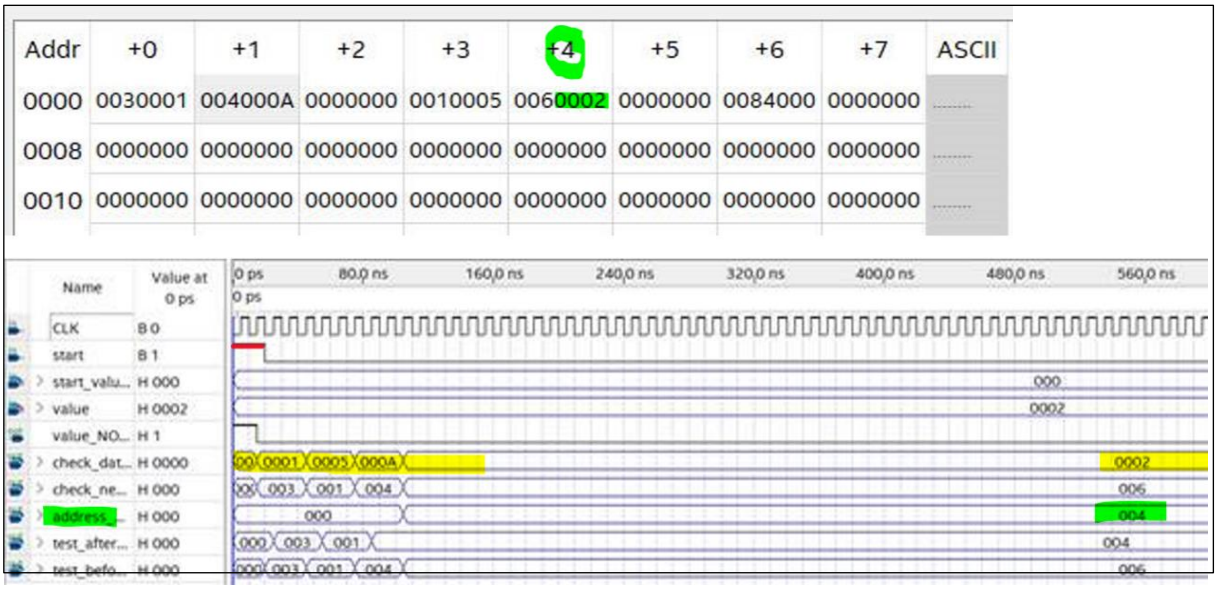
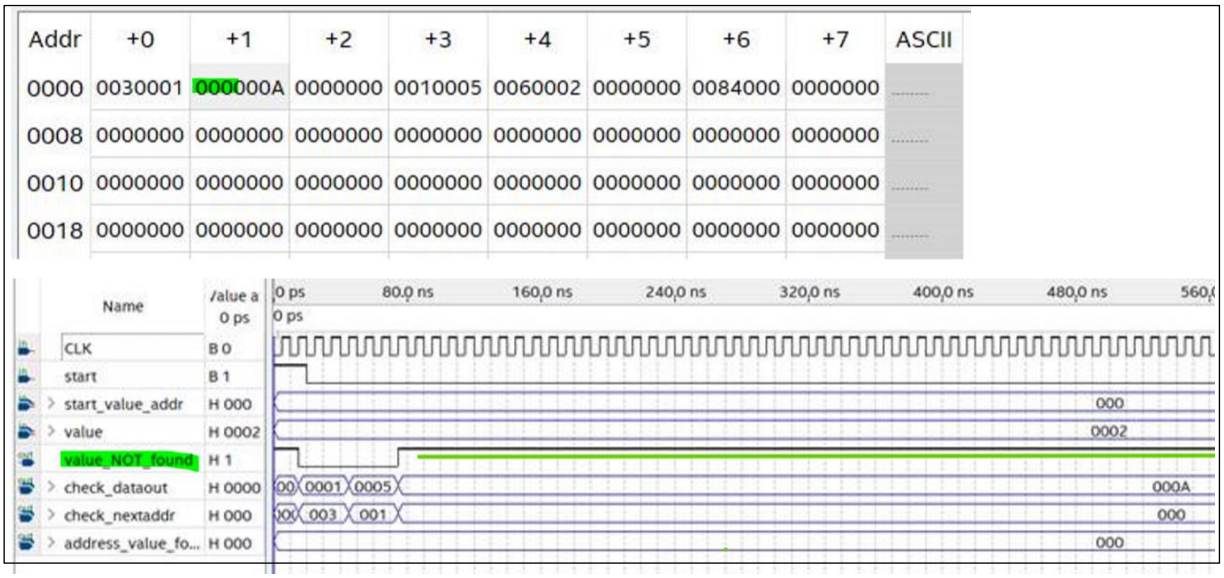


Figure 28 – test MIF file “value does not exist” and corresponding waveform



The final version of this task takes as many cycles as the number of nodes traversed due to final optimisations made prior to implementing in the general circuit.

Optimizing Task 3 – Linked List Search

To form the optimized block for task 3, alterations were made to the original block so that it would operate with a sixteen-bit memory word length. Changes that were made involved changing bus widths as well as introducing a block connecting the next pointer address input to the search address output, these changes can be seen in *Figures 29*.

Figure 29.1 – unchanged task 3

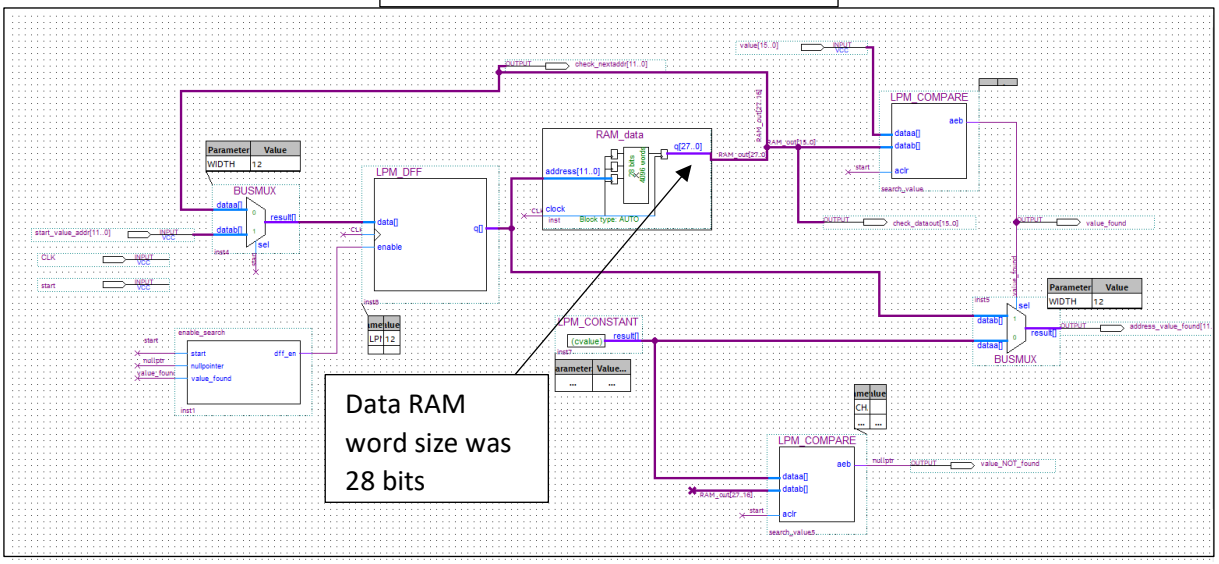
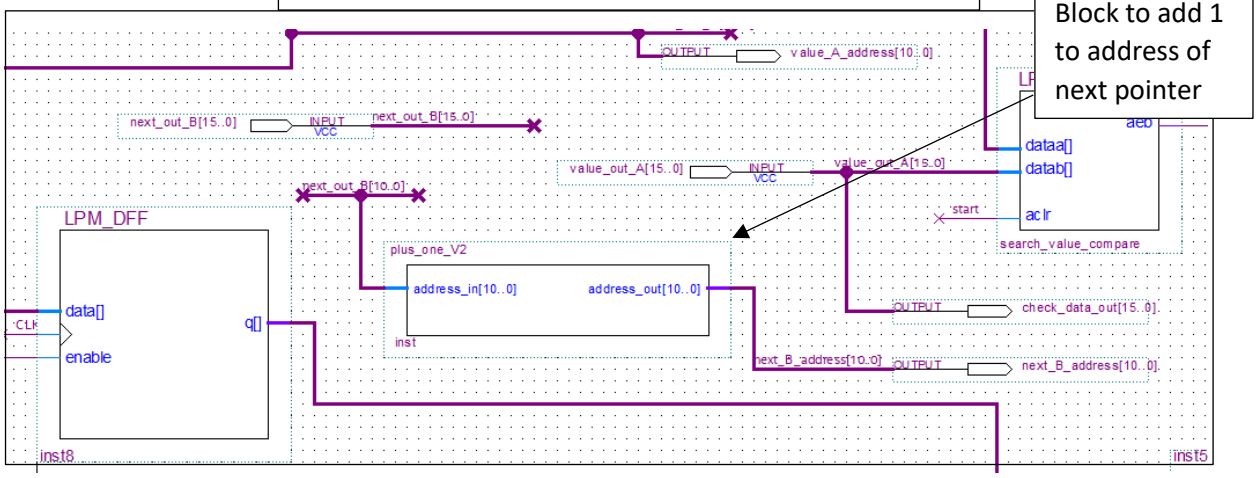


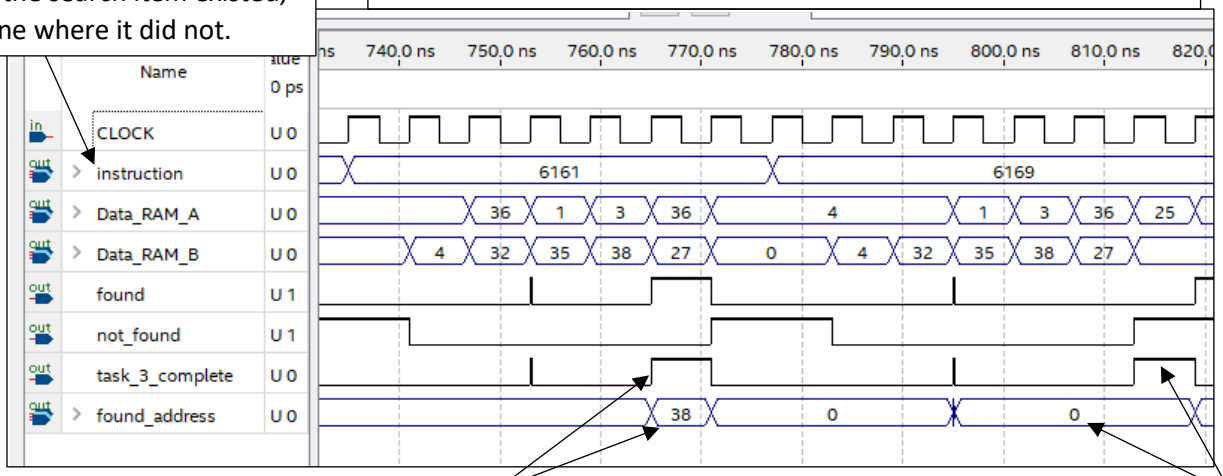
Figure 29.2 – highlight of changes made to Figure 16.1



Outputs value_A_address and next_B_address are the address inputs for port A and B of the data RAM meaning that item value and next pointer can be fetched at the same time. These changes were implemented with the assumption that the “value” and “next” pointer of each item in the linked list differed by 1 in their memory location and were made to take advantage of the RAM being a 2-port RAM instead of the 1-port in *Figure 29.1*. This was then tested for single and multiple inputs to ensure outputs were still correct (*Figure 30*).

Instructions were a linked list where the search item existed, then one where it did not.

Figure 30 – working waveform for 2 linked list search instructions



Task complete and address was found.

Task complete and item was not found (defaults to 0 as the ouput).

Figure 31 – logic for task 3 “complete” signal

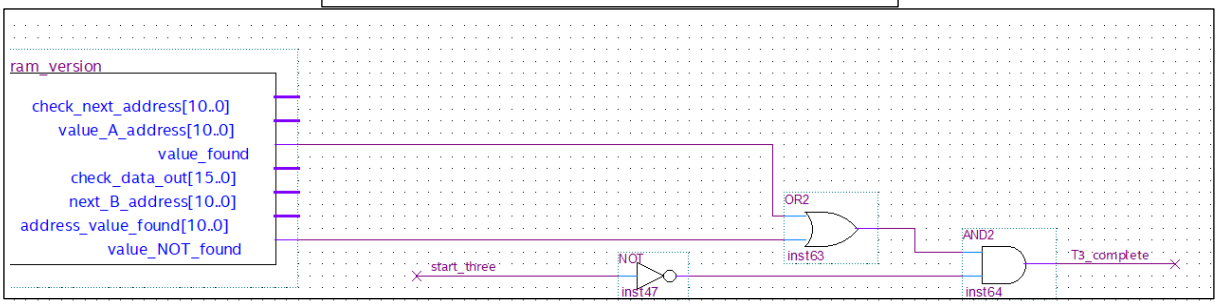


Figure 31 is a complete output signal that was formed by ANDing the ‘found’ OR ‘not_found’ signal (which indicate if the search value does/does not exist in the linked list) with NOT ‘start’ (which was set to low during operation and high otherwise). An assumption was made with the implementation of this task for the non-general CPU, the “Next” pointer value for any item in the linked list could never be zero (unless it was a null pointer).

The Complete CPUs

Initial decisions

The CPU is characterized by a flexible design that can adapt the task blocks to execute other instructions relating to the task as well as additional unrelated instructions such as load and store.

Both CPUs have the following general characteristics: two separate RAMs (Data and Instruction), the use of 16bits word for data path, the use of one only instruction that take multiple cycles to execute a specific task, following the CISC model of architecture **[5]** since found it to be faster in terms of clock cycles than executing multiple simple instructions.

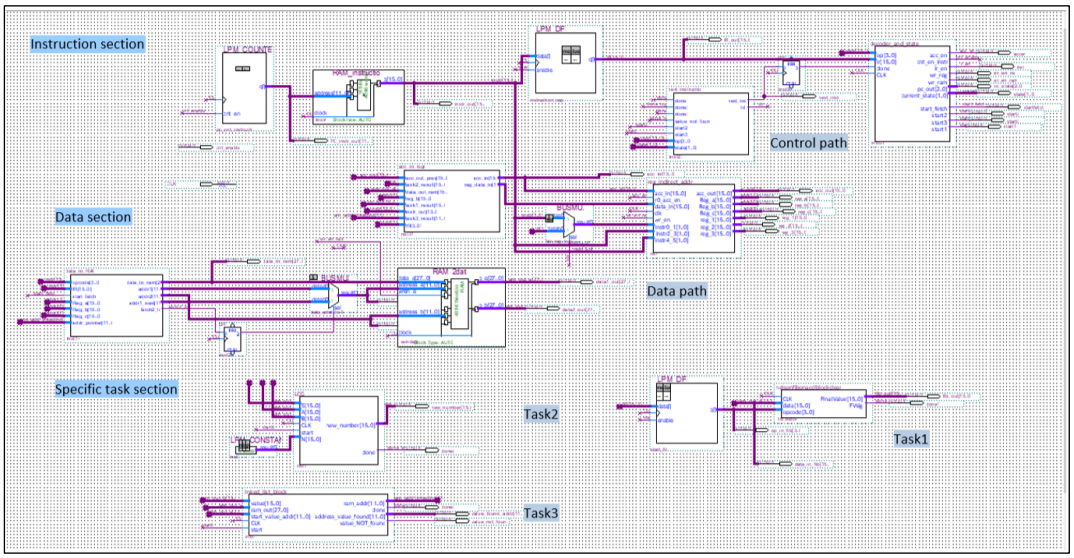
The General CPU

Overview

This CPU is characterized by a flexible design that can adapt the task blocks to execute other instructions relating to the task as well as additional unrelated instructions such as load and store.

The CPU was first designed on paper, listing all the elements needed to perform the most basic tasks (load, store etc.). Then it was built in Quartus (*Figure 32*) and after testing the functionality of the control and data paths, the specific tasks' blocks were added. Adjustments to the general circuit were made to produce the needed inputs. A block was created and to implement the option to fetch the inputs for task 2 and 3 by either indirect addressing or by fetching from a register.

Figure 32 – General CPU design file



ISA

Since the tasks performed by the CPU require different information in the instruction word, every instruction is divided differently. If the two most significant bits [15:14] are both 1, the opcode will be four bits long [15:12] otherwise the opcode is only 2 bits. Depending on the instruction, the rest of the word contain either a RAM address or a registers address in which the address/value is stored (*Figure 33*) explains the instruction word structure and provides more details on the functionality of each operation.

Figure 33 – ISA for the general CPU

instruction	opcode	Requirements info included in the instruction (corresponding bit word)	Enables required	Addressing mode and use of registers	functionality
LDR	00	Address (13-2), Ra (destination reg) (1-0)	Wr_en_reg	Indirect addressing	Load value from ram to register
STR	01	Address, Ra (source reg) (1-0)	Wr_en_ram	Indirect addressing	Store value from register into ram
LDI	10	value N (13-0)	Acc_en	Direct addressing	Load value N into accumulator
MOV	1100	Ra (destination reg) (1-0), Rb (source reg) (2-3)	Wr_en_reg	Indirect addressing	Move value from one register to another
LCG	1101	Ra (A) (1-0), Rb (B) (2-3), Rc (S)(4-5), mode of addressing (11)	Acc_en, start 2	Indirect addressing or value in a register	Generate a random number using a linear congruential generator
FIB	1110	address (11-0)	Acc_en, start 1	Direct addressing	Calculate the Fibonacci function of an input
LIS	1111	Ra (starting address) (1-0), Rb (value to be found)(2-3), mode of addressing (11)	Acc_en, start 3	Indirect addressing of value in a register	Search a value traversing a linked list

Instruction section and control path

This first section (*Figure 34*) of the CPU is composed by:

- Program Counter: incremented by the decoder at the end of each instruction it contains the instruction RAM address.
- Instruction RAM: initialized to have a test program.
- Instruction Register: updated with the contents of the instruction RAM at the beginning of each fetch state.
- Decoder: enables control lines based on the current state and instruction
- 'Next instruction' Block: defines when a task is complete, and the next instruction can be fetched
- 'Accumulator input' Block: that defines which sixteen-bit word is input into one of the four registers
- Accumulator and "indirect addressing" block: four register that can be written to and read from (three at time: Ra, Rb, Rc). Register 0 can be used as an accumulator

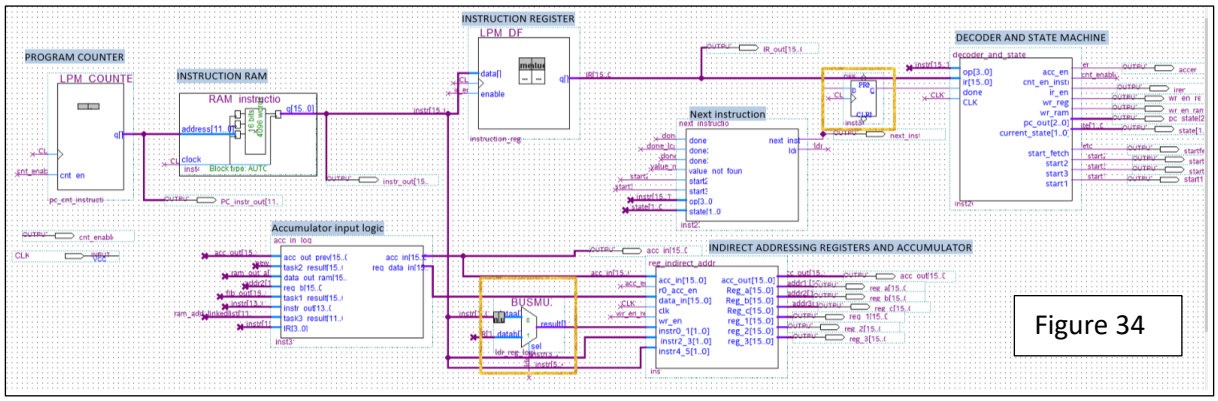


Figure 34

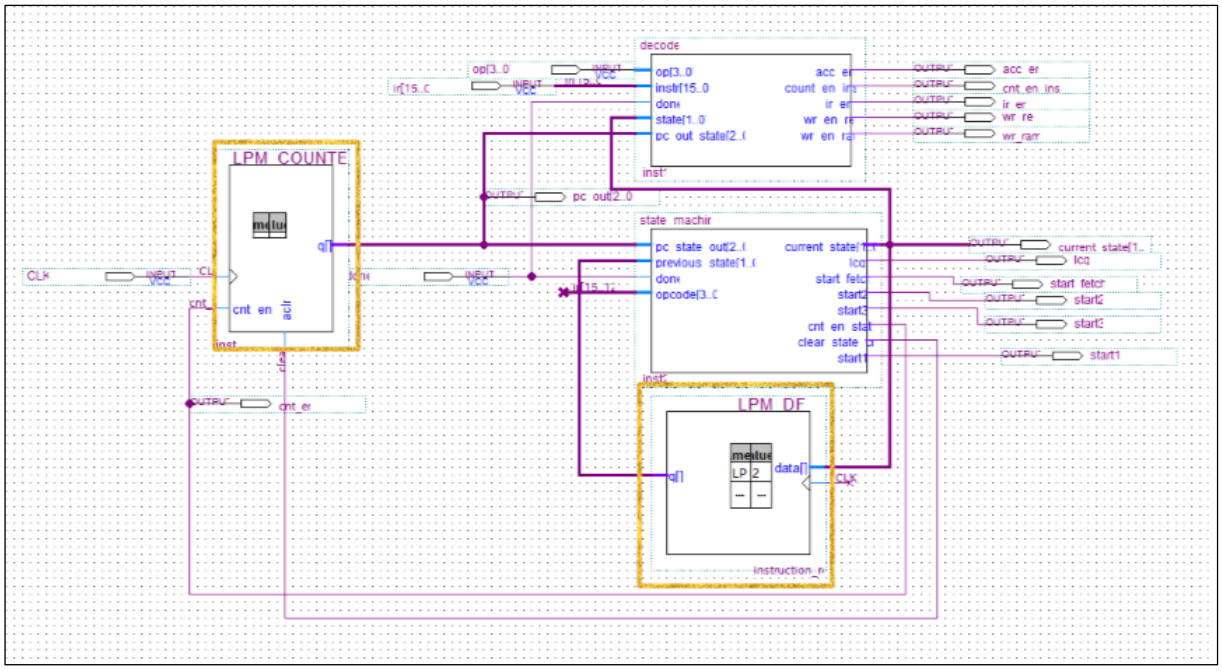
Decoder and state machine block

Both are controlled by a counter (PC) (*Figure 35*) that counts the number of cycles passed, resetting at every new instruction:

- PC 0,1,2 state: “fetch instruction”
- PC 3, 4, (5 for LCG) state: “fetch data”
- PC 5 (6 for LCG) state: “execution”

The counter stops during the execution and it is cleared when the instruction is done. After testing, a co-dependency between the state machine- decoder block and the next instruction logic was identified. Since they are both combinational logics, if one changes, the other does, resulting in a loop causing an error in the waveform. A multi-bit wide DFF was added (*Figure* 34) giving a one cycle delay between the changes of the two blocks, avoiding loop. For the same reason, a DFF was inserted between the output of the current state of the state machine and the inputs of the previous stat (*Figure 35*).

Figure 35



State machine

There are three possible states:

- “Fetch instruction”: the program counter increases by one, the new instruction is fetched from the RAM during cycle 1 and 2 of the state machine counter (considered two cycles for the ram to output the new value since the first implementation involved a register at its output stage).
- “Fetch data”: cycles 3 and 4, the address of the data RAM is updated, and it outputs the variables needed for the task to execute. since two values are fetched at time and LCG needs three input values, this state will extend until cycle 5. At cycle 3 the IR is updated.
- “Execution” of the current task: this state continues until the “next instruction” input is high.

The state machine takes as inputs the output of the state machine counter, the previous state, the current state, and the output of the next instruction block delayed by one cycle. The outputs include the current state which is equal to 10 during the execution state, 01 if in fetch data, and 00 in any other case; Along with all the start inputs for fetching data and enabling execution of a specific task. The “start fetch” output is high at the first cycle of fetch data. The start2 (LCG) and start3 (LIS) outputs are high the first cycle of execution state and since FIB (task1) works slightly differently than the other two, the start1 output is high during all execution cycles.

Pipelining these states would only be partially possible since:

- It cannot be forecast when certain instructions will end their execution given their variable execution times.
- Fetching data cannot be done until the instruction is fetch (the addresses are not known in advance since there are contained in the instruction word)
- The execution cannot happen before fetching the data input

In the implementation given the only pipelined stages taking places are the execution of LDI, STR, MOV and LDR during the fetching state of the next instruction since these operations require only one execution cycle (LDR takes two and has one cycle pipelined) and involve only the content of the data RAM and the registers. Further optimizations have been done to the non-general of the CPU, more relevant than this one in terms of performance.

Decoder

Here most of the control inputs are defined based on the state, the current cycle number, the opcode coming from the instruction register and the “next instruction” input. The Verilog code implements the following logic:

- The accumulator is enabled whenever one of the tasks requires to store a result in it (FIB, LCG, LIS, LDI) and is loaded in the instruction register and the “done” output is high.
- The program counter fetching the next instruction address is enabled whenever the “next instruction” output is high.
- The instruction register changes its content whenever the fetch_data state starts.
- The write enable for the registers of the indirect addressing block is set to 1 whenever one of the instructions that requires to write in a register (LDR, MOV) is complete (*Figure 36*).
- The write enable of the ram is high during an STR instruction when it has been completed (*Figure 36*).

Next Instruction Block

This block outputs when an instruction is completed. Since there are different timings/execution length for each instruction, each task block has a “done output” that is high when the instruction is complete. Both the program counter and accumulator enable depend on this. Depending on the current instruction, the ‘next_instr’ output (*Figure 36*) follows different logic:

- STR, MOV and LDI do not need execution cycles since their execution happens between the ‘fetch_data’ state and the ‘fetch_instruction’ of the next instruction (pipelined); Since the ‘done’ output is delayed by one cycle “next instruction” output will be high already in first cycle of ‘fetch data’.
- For LDR an execution cycle is needed (using the ram requires an addition cycle). consequently, the output will be set high at the first cycle of execution.
- For the specific tasks, they have been implemented so that there is one done output for LCG and for FIB, while for LIS there are two. Therefore, the ‘next instruction’ output will be high whenever one of those instruction are completed.

Figure 36 – next_instr code

```
assign next_instr = (done2&lcg&!start2&exec)|((done3|value_not_found)&lis&!start3&exec)
                    |(exec&ldr)|(fetch_data&(ldi|str|mov))|(done1&fib);
```

While running tests, an additional condition was added as an assertion after incurring in errors; Sometimes the task would produce a done output while fetching the input at the start and would not execute. Consequently, stated that any done output can be high only when start input is low. Having the state machine be reset by the complete output of the current task makes the control path very flexible in implementing additional instructions with undefined number of cycles.

Testing the Control Path

Some tests were done at this point of the implementation to check the functionality of the control path, focusing on the state machine timing. As can be observed in *Figure 37* the state machine and decoder outputs behave as expected. In *Figure 38* it can be noted how the entire control path works.

Figure 37 – state machine and decoder test waveform

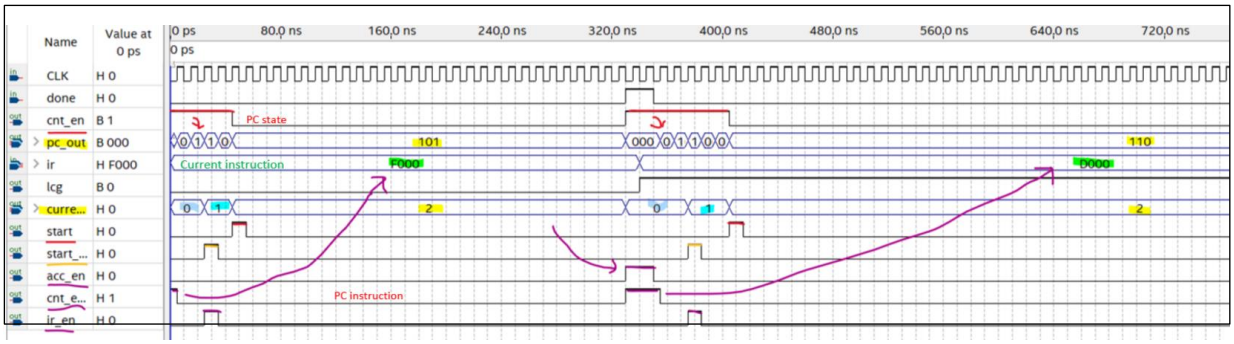
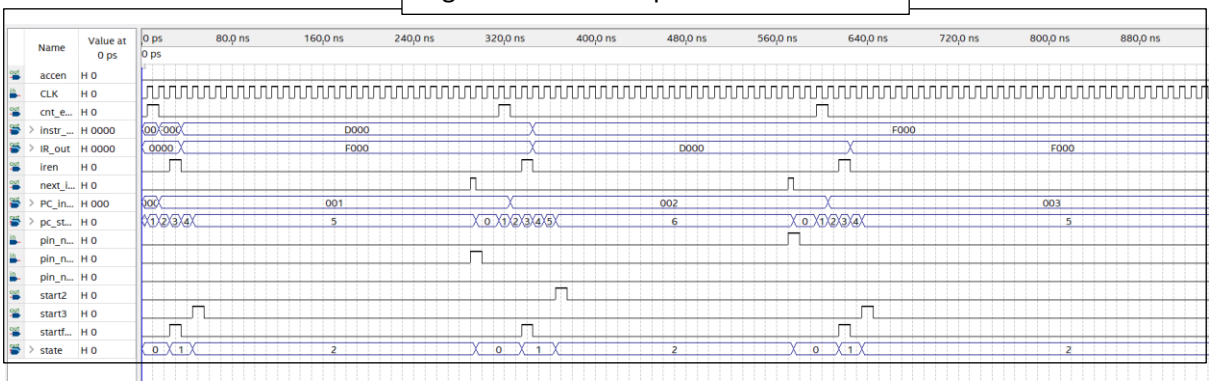


Figure 38 – control path test waveform



Data section

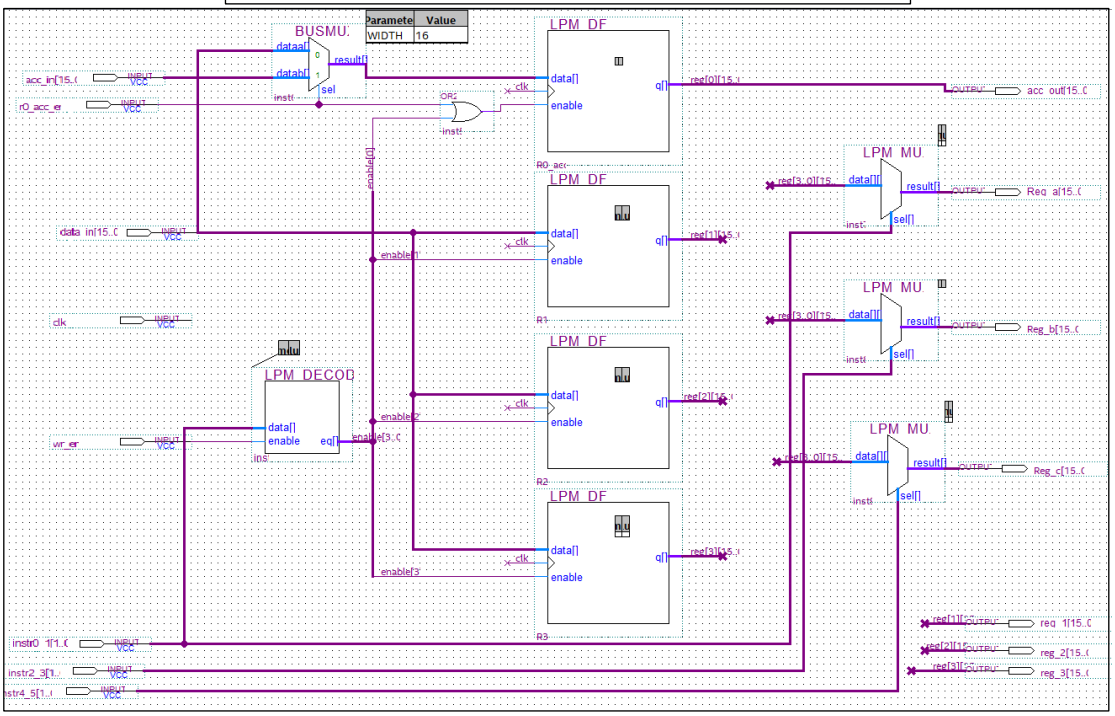
This section is composed of the data RAM, the logic block defining which data to be input in the RAM (current address and value to be written), the accumulator, and the indirect addressing block with the logic block defining the data that's loaded into the registers.

Indirect Addressing and the Accumulator Block

This block (*Figure* 39) consists of four registers (r0, i.e. the accumulator, r1, r2, and r3) and can perform different and more complex operations from the ones required by the specification **[2]** thanks to the two modes of operation it uses:

- R0 can be used as an accumulator, enabled by the input defined in the decoder and loaded with the data defined by the logic block “accumulator input”. Using an LDI instruction, the accumulator can be loaded directly with a value provided in the instruction word’s 14 least significant bits.
- All four registers can be loaded either with a value coming from the data RAM output (LDR) or a value contained in another one of the registers (MOV). The lpm_decoder defines which of the four registers is written by enabling only one of them using a two bit select line (the two least significant bits of the instruction word) and loading the data from the “data_in” port. Simultaneously to the write operation, three register can be read from. Those register are selected by three multiplexers with control inputs respectively bits 0 and 1 (Ra), 2 and 3 (Rb), 4 and 5 (Rc) of the instruction word. The data outputs do not need to all come from different registers, but one will always be the same as the one written (Ra).

Figure 39 – indirect addressing and accumulator block



Accumulator Input Logic Block

This Verilog file defines which data word should load in the accumulator/one of the registers:

- Accumulator: depending on the current instruction (LCG, FIB, LIS or LDI) the “data_in” input will be set to the result of task1, 2, 3 or the least 14 significant bits of the LDI instruction word
- Register: depending on the current instruction the “data_in” input will be the output of the RAM (STR) or the content of the source register (MOV) and will be written into the register corresponding to the two least significant bit of the instruction word (Ra)

Additional Observations on the Timing of the Load Operation

Since for STR and MOV the execution of the instruction happens during fetch states to reduce the number of cycles of those instruction, thus, the execution state never occurs, the written register will be loaded before the instruction register actually changes. Therefore, a need to use the output of the instruction RAM, instead of the output of the IR, for all the control input of those two instructions was necessary. At the same time, after testing LDR, it was evident that this is the only instruction that writes to registers after the execution, when the instruction RAM output cannot be taken as it is already changing to fetch the next instruction, but the IR keeps LDR. Consequently, a bus mux has been added before the indirect addressing block input port to choose the register based on the instruction word coming from the RAM or the IR, depending the last instruction fetched.

Data RAM and Input Data Logic

Since there were no restrictions on the width of the data word in the memory **[2]**, it was set at twenty-eight-bits wide; The twelve most significant bits are not be used if not dealing with a linked list. It was also decided that a two port RAM be used so that two values can be read at the same time, halving the time needed to fetch data.

An additional Verilog file defines which address and data are input into the data RAM, given the control input “start_fetch” is high and the current instruction is:

- LIS (indirect addressing): addr1 from the register ‘A’ (starting address) and addr2 from Rb (value to find) (0-1 and 2-3 bits of instruction word).
- FIB (direct addressing): address is directly in the twelve least significant bits of the instruction word (argument of Fibonacci function).
- LCG: since this instruction needs three inputs, it cannot be fetched in one cycle. The first cycle, when start fetch is high, addr1 (content of Ra) fetches ‘A’ and add2 (content of Rb) fetches ‘B’. On the second cycle addr 1 (content of Rc) fetches value of ‘S’. The ‘start fetch’ input is delayed by one cycle with a flip-flop and is used as select line (fetch2_lcg) of a mux that, when high, selects the address of ‘S’ from the first port (add1_next, second cycle of fetch data state). Therefore, the fetch data state for LCG instruction needs one additional cycle.
- STR or LDR: address found in bits thirteen to twelve of the instruction word.

The “data_in_ram” input (twenty-eight bits) is needed only with store instructions, in which case the sixteen least significant bits are equal to the output of ‘Ra’ with the twelve most significant bits set to zero**.**

Implementation of the specific Tasks and Testing

Finally, the blocks of the specific tasks were implemented and connected to the required input and outputs. An additional option was introduced: if the 11th bit of the instruction word of LIS or LCG is high, instead of fetching the input values through indirect addressing the registers addressed by the instruction word directly contain the input value. Several tests were ran at this point, to check the functionality of each of the seven instructions, comparing the outputs with the predicted ones calculated on paper, and multiple instructions in series, verifying the CPU worked with various combination of operations in sequence. Below (*Figures 40* to *43*) are two of the most relevant waveforms generated by the CPU using different MIF files both for the data and the instruction RAM.

TEST 1

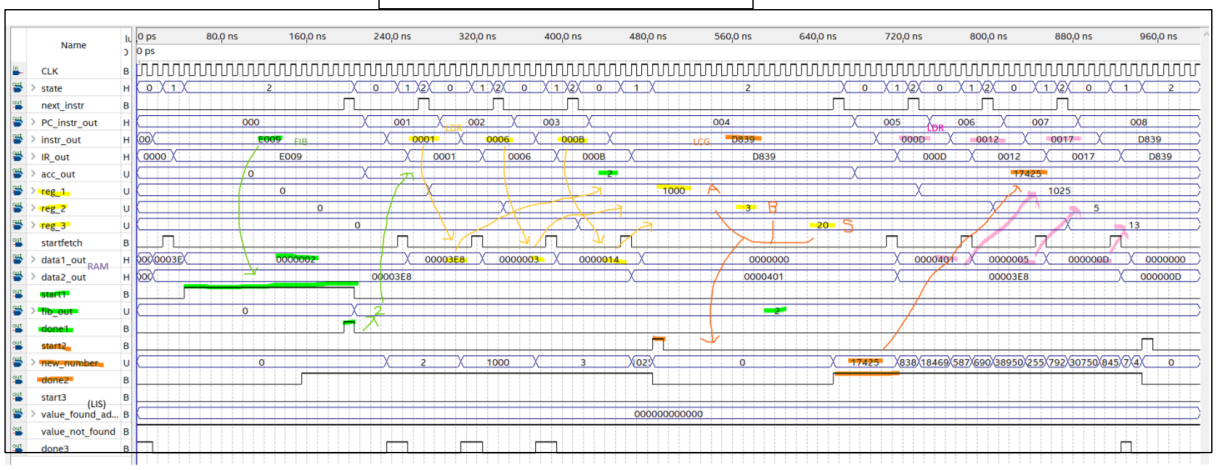
Given the following content of the data RAM (address in decimal, data in hex)

Figure 40 – test 1 MIF file

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	00003E8	0000003	0000014	0000401	0000005	000000D	0000000	0000000	.....
8	0000001	0000002	0000003	0000004	0000000	0000000	0000000	0000000	.....
16	0000000	0000000	0000000	0000000	0000000	0000000	0000000	0000000	.....
24	0000024	0000020	0000000	0000000	0000000	0000000	0000000	0000000	$ .....
32	0230001	0000000	0000000	0260003	0000000	0000000	01B0024	0000000	.....

FIB 0x009 (green); LDR 0x000 R1; LDR 0x001 R2; LDR 0x002 R3 (yellow); LCG R3 R2 R1 (orange); LDR 0x003 R1; LDR 0x004 R2; LDR 0x005 R3 (pink)

Figure 41 – test 1 waveform



TEST 2

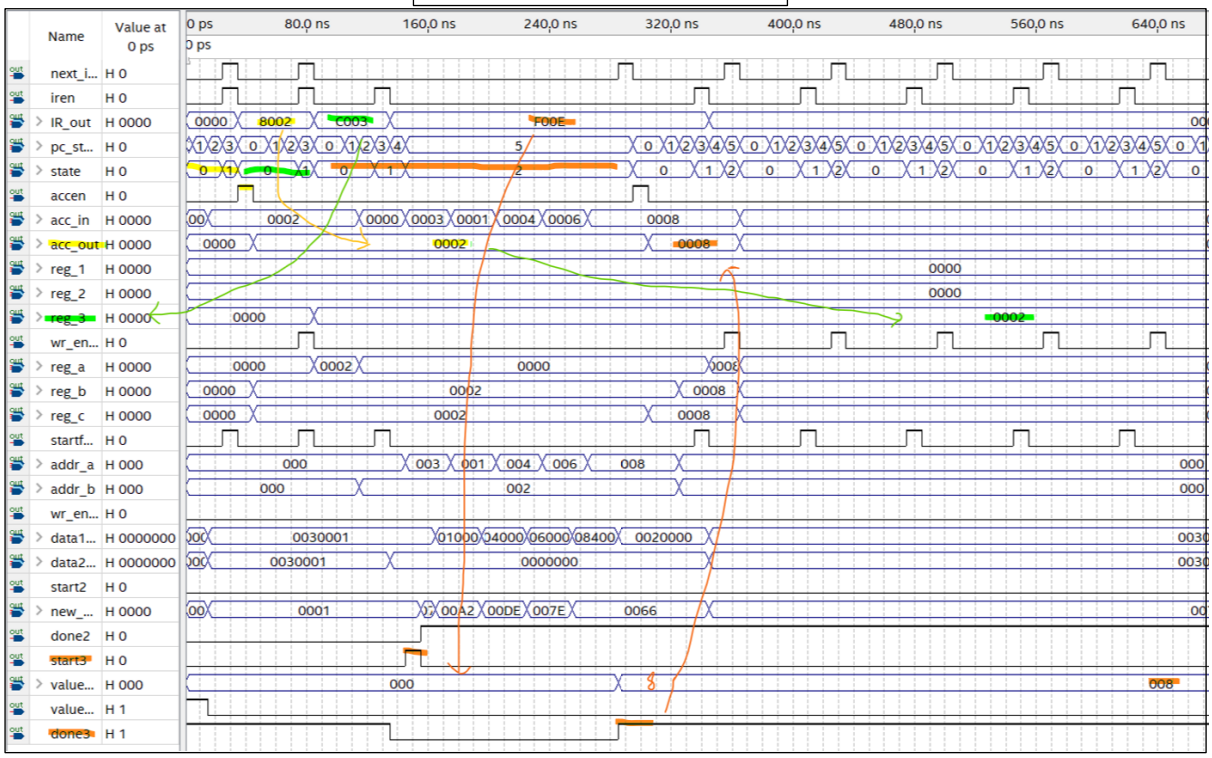
Given the following content of the data RAM (linked list)

Figure 42 – test 2 MIF file

Addr	+0	+1	+2	+3	+4	+5	+6	+7
0000	0030001	004000A	0000000	0010005	0060002	0000000	0084000	0000000
0008	0020000	0000000	0000000	0000000	0000000	0000000	0000000	0000000
0010	0000000	0000000	0000000	0000000	0000000	0000000	0000000	0000000

LDI 2 (yellow); MOV R0 R3 (green); LIS R3 (address of the value to be found = 0000) R2 (address of the starting address= 001) (orange)

Figure 43 – test 2 waveform



During the optimization process, the register present at the output inside the RAMs were removed, speeding up the fetch process by one cycle. Some possible improvements to this circuit would be to add a jump instruction or an arithmetic one. Both these ideas would be fairly easy to implement: the first one would require adding an ‘if’ statement in decoder (if the opcode of JMP is present the state machine returns to the first cycle of fetch data and program counter output is equal to the address in the jump instruction). While for the second suggestion, an arithmetic unit (ALU for addition or subtraction, same multiplication unit used in task 2 for multiplication or slightly modified one for division) at the output of the accumulator would be added, making minimal changes in the data_in logic block. Another instruction that could be implemented is stop, by adding a Verilog block at the output of the instruction RAM with the following logic: if JMP opcode, all enable set to low, included the one of the program counter of the state machine.

Optimized CPU

The optimized complete architecture uses the altered Fibonacci and Linked List blocks as well as the unaltered LCG block as the basis for its main computational unit. For these blocks to operate together, a common instruction register/decoder (IR) had to be created. To minimize the amount of code in the Verilog block for the IR, it was split into four units. One task specific decoder for each of the 3 possible operations, and a general IR whose sole purpose was to act as an enable line for each of the instruction specific decoders, and ensured that no operations were taking place when the address of the instruction RAM, which comes from the program counter (PC), was changing (*Figures 44*).

Figure 44.1 – input into general IR to disable operations during PC address changes

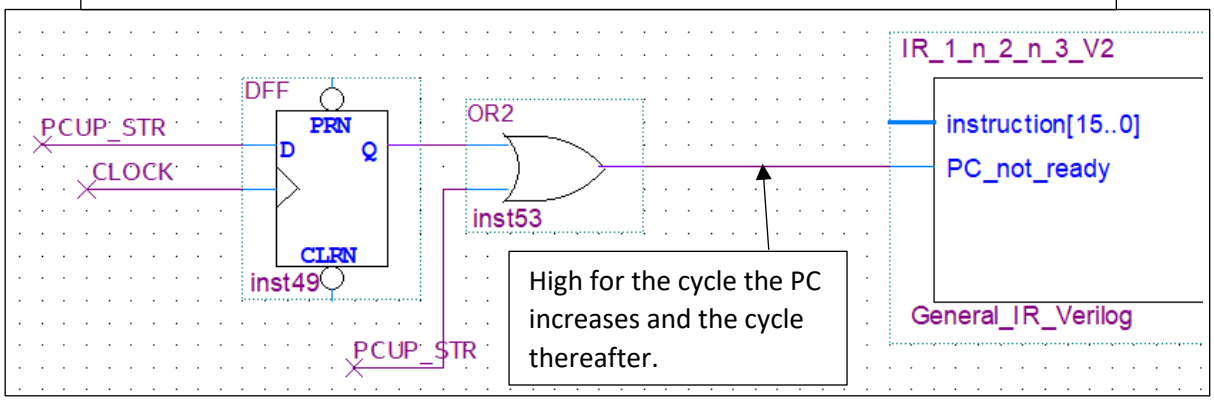
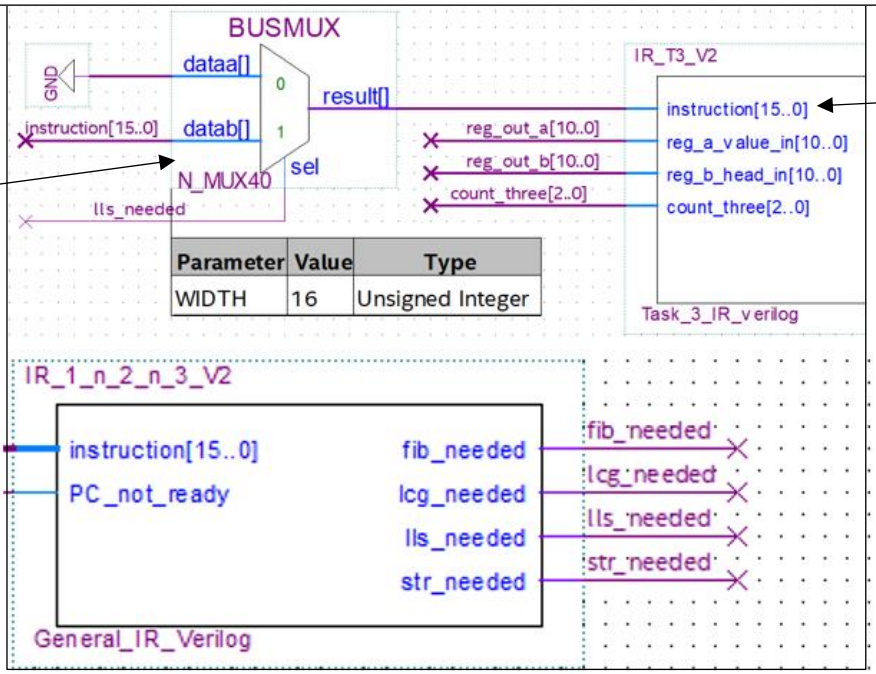


Figure 44.2 – Verilog code for general IR

```
1   module IR_1_n_2_n_3_V2
2
3   (
4       input [15:0] instruction,
5       input PC_not_ready,
6
7       output fib_needed,
8       output lcg_needed,
9       output lls_needed,
10      output str_needed
11  );
12
13  wire [4:0] OP;
14
15  assign OP = instruction[15:11];
16
17  assign fib_needed = (OP == 1) & !PC_not_ready;
18  assign lcg_needed = (OP == 2) & !PC_not_ready;
19  assign lls_needed = (OP == 3) & !PC_not_ready;
20  assign str_needed = (OP > 7) & !PC_not_ready;
21
22  endmodule
```

Figure 45 illustrates how the control lines from the general IR acts as an enable line for the task specific decoders.

Figure 45 – Output of general IR and how it enables the task 3 (linked list) IR



Same situation for all other task IR's.

Does not operate when instruction input is 0.

When a task specific decoder receives the instruction data it performs its own fetch cycles to ensure the main task specific operating block has all the correct input before it starts operating; Each main operating block has some form of a direct or indirect enable line which stops it from operating. This required a different number of cycles, which were timed using a counter for each task. The implementation of the counter and each of the task's IR operation is depicted in *Figures 46*.

Figure 46.1 – Task 2 (random number generator) IR and its counter

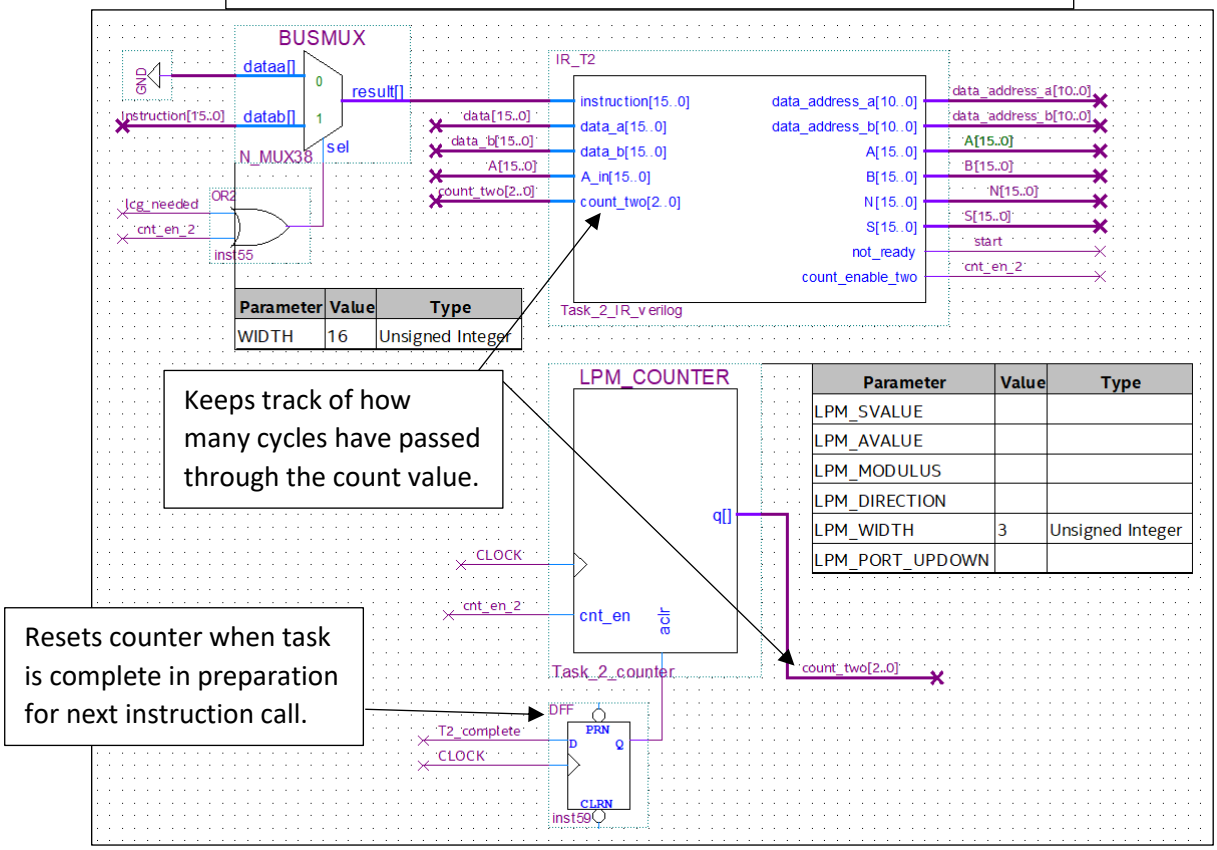
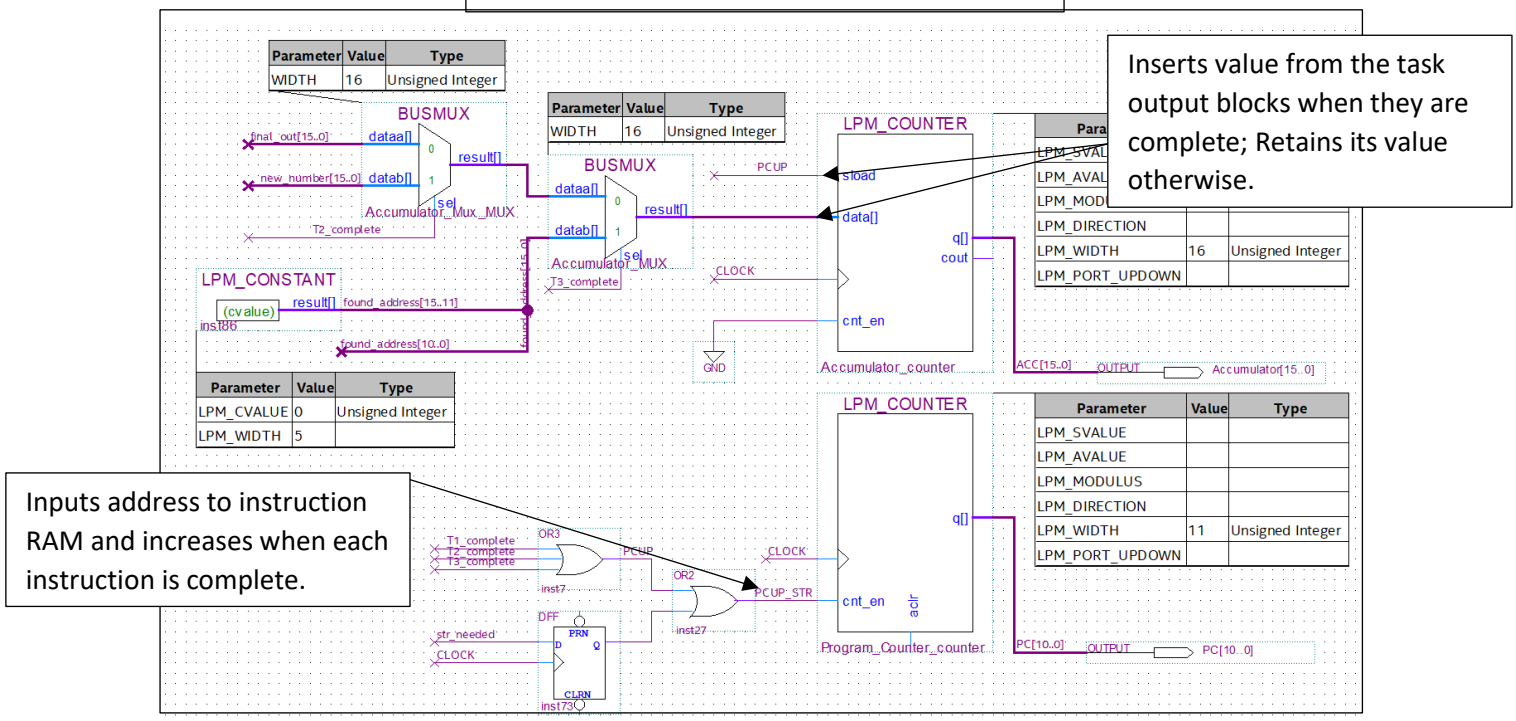


Figure 46.2 – Fetch process for each task

Task	Counter Value	Operation
Fibonacci	0	Fetches ‘n’ from Data RAM
	1	Inputs n into main operating block and enables it
Random Number Generator	0	Fetches A and B
	1	Fetches S
	2	Inputs variables into main operating block and enables it
Linked List Search	0	Fetches ‘head address’ address and search value address from registers
	1	Fetches head address and search value from the Data RAM using address’ stored in registers
	2	Inputs these values into the main operating block and enables it

Other components added into the general architecture included a PC and an Accumulator (*Figure 47*), both implemented using counters with different settings and use the “complete” outputs of each task block to alter their current value.

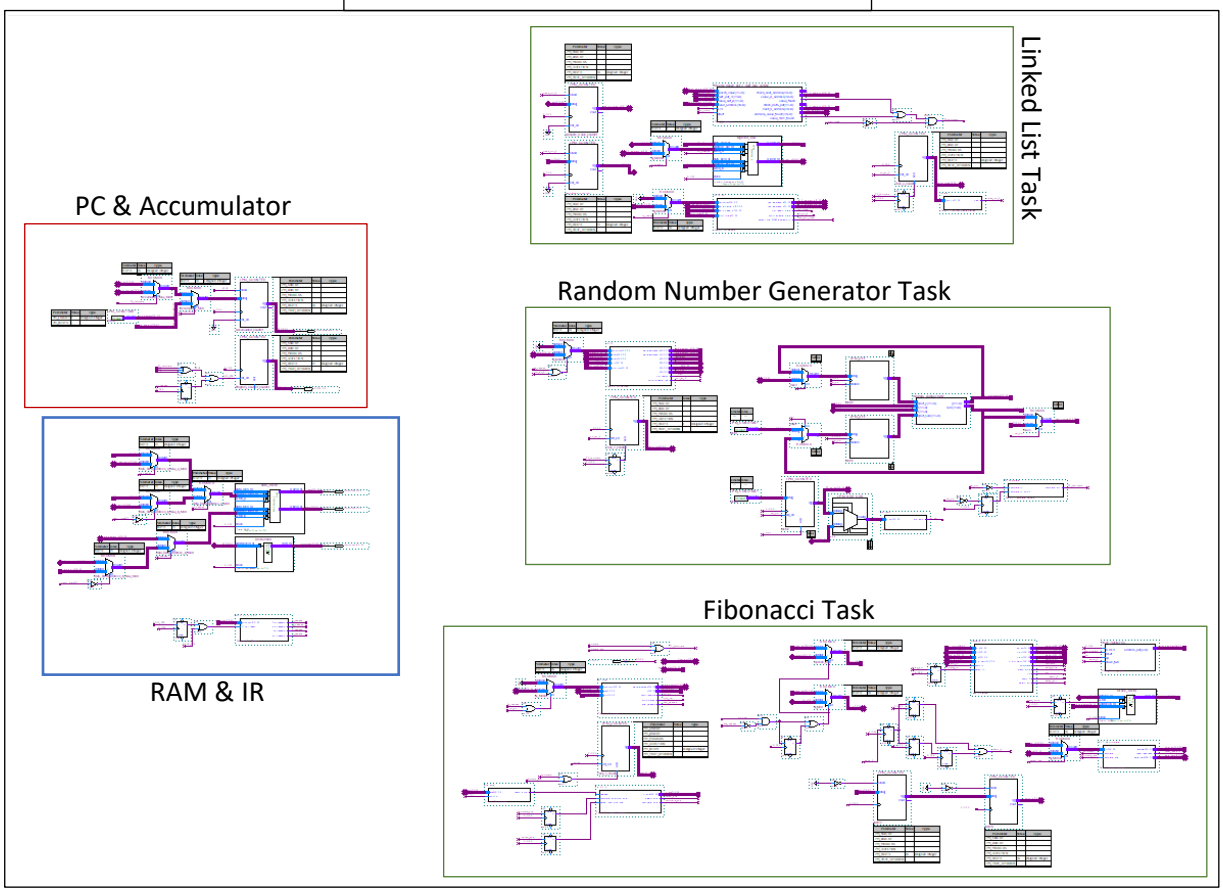
Figure 47 – Program counter and Accumulator



A 2-port RAM block (with eight, eleven-bit words), which was to be used for the indirect addressing of task 3, was added. To initialise variables into this RAM, a 4th instruction was needed which loaded addresses into the RAM directly. The execution of this instruction was implemented and tested to ensure the correct values were written to the correct address.

The final complete circuit can be split into distinct sections as seen in *Figure 48*. At the start of each new instruction the general IR selects which task specific block is needed for the task. Once the task-specific decoder performs its fetch cycle and the main operational block executes the instruction, the output is automatically stored into the Accumulator and the Program Counter is incremented by 1.

Figure 48 – Complete non-general circuit



The CPU was tested with *Figure 49* being the format for the assembly line code used. To confirm that all the components correctly operated, the CPU was tested with *Figure 50*, this format is unique to the non-general CPU.

Figure 49 – Table translating the binary instructions to an assembly line code

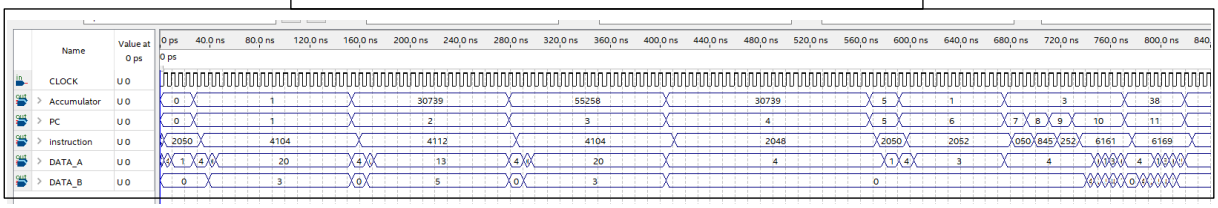
Assembly Line			Binary		
Fibonacci Task					
Operand	**Address of n**		**Opcode [15:11]**	**Address of n [10:0]**	
FIB	Data RAM		00001	Data RAM	
Random Number Generator Task					
Operand	**Address of a**		**Opcode [15:11]**	**Address of a [10:0]**	
LCG	Data RAM		00010	Data RAM	
Linked List Search Task					
Operand	**Search value address**	**Head address**	**Opcode [15:11]**	**Search value address [5:3]**	**Head address [2:0]**
LLS	Task 3 RAM		00011	Task 3 RAM	
Loading Values to Task 3 RAM					
Operand	**Address**	**Value**	**Opcode [15:14]**	**Address [13:11]**	**Value [10:0]**
LDR	Task 3 RAM	Integer <= 2K	01	Task 3 RAM	Integer <= 2K

Figure 50 – CPU test code

```
FIB  2  // n = 1
LCG  8  // a = 1000, b = 3, s = 20
LCG  16  // a = 1025, b = 5, s = 13
LCG  8  //  a = 1000, b = 3, s = 20
FIB  0  // n = 4
FIB  2  // n = 1
FIB  4  // n = 3
LDR  2  24
LDR  1  25
LDR  3  0
LLS  2  1 // search value = 36, head = 32
LLS  3  1 // search value = 4, head = 32
```

Figure 51 is a waveform that shows the instructions executed correctly through the changing Accumulator value, the default for an LLS value when the search value does not exist in the linked list is to update the accumulator to 0.

Figure 51 – waveform corresponding to Figure 50



Now the complete circuit was ready for functional tests to find maximum clock speed and power consumption along with the main bottlenecks of the circuit to further optimize the performance. Since all the task specific blocks were already tested no noteworthy errors were encountered during the creation of the general architecture, the implementation of task specific blocks was made easier through communication between the members who originally designed them.

Functional Analysis and Optimisations

When analysing the block, the main parameters that we are looking to improve upon are:

- Maximum clock frequency (Fmax). Ideally, we want a clock frequency of 100MHz.
- Power dissipation
    - Total Thermal power dissipation (excluding I/O power dissipation)
    - Dynamic Thermal power dissipation – Power dissipated from the switching of signals
    - Static Thermal power dissipation - Power dissipated from the circuit being on
- Total number of logic elements

Initial CPU Comparison

Slack – “the margin by which a timing requirement was met or not met. A positive slack value, displayed in black, indicates the margin by which a requirement was met. A negative slack value, displayed in red, indicates the margin by which a requirement was not met **[6].**

	General	**Non-General**
Fmax (MHz)	52.71	91.94
Largest slack	-8.970	-0.877
Total power dissipation (mW)	297.43	68.67
Dynamic power dissipation (mW)	197.56	25.66
Static power dissipation (mW)	99.87	43.01
Total logic elements	1737	895
Total registers	453	170
Total block memory bits	3,325,953	66,808

Initially, the non-general CPU was displaying much more desirable values and the slack was very low compared to the general CPU. Additionally, the power consumption is about 4-5 times lower with the number of logic elements being about half. A significant value here is the number of memory bits which requires attention to.

Optimising the General CPU

The performance of the general CPU was limited by the Fibonacci block which made it the focus of optimization.

Initial analysis

Properties	**Values**
Max clock frequency (MHz)	49.78
Total thermal power dissipation (mW)	237.4
Dynamic thermal power dissipation (mW)	138.14
Static thermal power dissipation (mW)	99.26
Total number of logic elements	1085

The main issue causing these undesirable values came from the extreme stack size of 65536 addresses and 3,145,728 bits (due to the word length being 48 bits). Further research into how stack space would be used up when calling a function showed that for an input parameter of n, n address spaces would be needed. This means that the current configuration allows for a maximum input of 65536 which is unnecessarily high; due to the how the expected input parameter is supposed to be quite low. The stack size was reduced to 32 addresses and as the address length could be reduced to 5 bits, the word length was reduced to 37 bits. As a result, the stack now uses 1,184 memory bits.
As stack traditionally uses space in the main memory (in this case the data RAM), it was appropriate to evaluate the effect of moving the stack into the RAM. Currently the number of memory bits used by the data RAM is 114,688 and the address length is 12. To accompany this address length, the word length used with the Fibonacci block would have to be extended to 44 which consequently extends the RAM's word length from 28. This would increase the total number of memory bits by 64,352 which is much higher than the increase 1,184 memory bits when the stack is separate from the data RAM. Consequently, the stack was kept separate from the data RAM.

There was also a plethora of minor changes such as:

- Replacing the register that stored the previous value in addition conditions with a d-type flip flop. This removed the need of a load signal to be output from the ‘valuecheck’ block.
- Reducing the bit count of values used in the state machine and the ‘InitialCounter’. The state machine only used 4 states and the counter’s maximum value was 3 meaning both needed a minimum of 2 bits to function properly.
- Deciding to use a counter to store the final value instead of a register.

Properties	**Counter**	**Register**
Fmax (MHz)	93.48	92.48
Total Thermal Power Dissipation (mW)	61.87	60.34
Total logic elements	682	625

Despite the register providing a lower power dissipation and reducing the area of this block, the maximum clock frequency was prioritized to get closer to the target frequency. Therefore, a counter was used clearing up unnecessary inputs into the ‘valuecheck’ block and unnecessary code in its verilog.

Final evaluation

Properties	**Initial Values**	**Final Values**
Max clock frequency (MHz)	49.78	93.11
Total thermal power dissipation (mW)	237.4	57.84
Dynamic thermal power dissipation (mW)	138.14	16.66
Static thermal power dissipation (mW)	99.26	42.94
Total number of logic elements	1085	618
Total block memory bits	3,145,728	1184

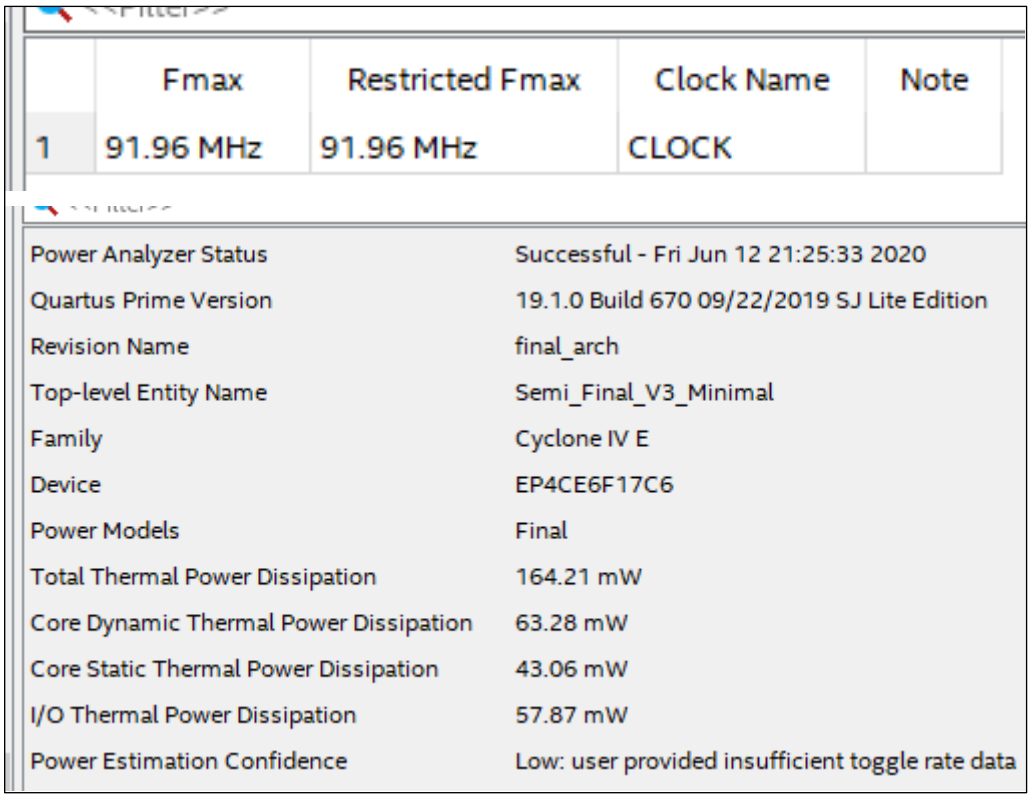
Optimizing the Non-general CPU

Optimizing the non-general circuit can come in the form of lower power consumption or higher clock speed. The initial power analyser summary showed that total power consumption was low, relative to that of the general circuit, at around 105 milli watts, most of which was a result of the RAM's used in the design meaning that little change could come from power oriented optimizations.

The main design improvements would likely be in the clock speed. A lot of the slack came from either connection to the stack pointer or to the task specific counters. Form here the first plan was to replace all task counters with a single task counter. However, that resulted in tasks 2 and 3 not operating correctly. An attempt was made to alter the instruction format for the Fibonacci instruction by replacing the address part of the instruction word with the direct value of n; This resulted in a significant decrease in the max clock speed to around 75MHz.

In the end, no significant improvements were made to the non-general CPU. Task counter port widths were reduced, and some test outputs were removed resulting in the final maximum clock speed of 91.96 MHz and a power dissipation of 106.34 as seen in *Figure 52*.

Figure 52 – results from final timing and power analysis



	Fmax	Restricted Fmax	Clock Name	Note
1	91.96 MHz	91.96 MHz	CLOCK	

Power Analyzer Status	Successful - Fri Jun 12 21:25:33 2020
Quartus Prime Version	19.1.0 Build 670 09/22/2019 SJ Lite Edition
Revision Name	final_arch
Top-level Entity Name	Semi_Final_V3_Minimal
Family	Cyclone IV E
Device	EP4CE6F17C6
Power Models	Final
Total Thermal Power Dissipation	164.21 mW
Core Dynamic Thermal Power Dissipation	63.28 mW
Core Static Thermal Power Dissipation	43.06 mW
I/O Thermal Power Dissipation	57.87 mW
Power Estimation Confidence	Low: user provided insufficient toggle rate data

Final CPU Comparison

	General		Non-General	
	Initial	**Final**	**Initial**	**Final**
Fmax (MHz)	52.71	64.7	91.94	91.96
Largest slack	-8.970	-5.456	-0.877	-6.874
Total power dissipation (mW)	297.43	97.5	68.67	106.34
Dynamic power dissipation (mW)	197.56	54.38	25.66	63.28
Static power dissipation (mW)	99.87	43.12	43.01	43.06
Total logic elements	1737	1282	895	878
Total registers	453	376	170	168
Total block memory bits	3,325,953	181,408	66,808	66,308

The general CPU has shown a significant improvement in every aspect but most of the non-general CPU's properties have worsened. Despite the higher power total power dissipation, the non-general CPU still has the better clock frequency, logic element and register count. This increased total power dissipation comes from the higher dynamic power dissipation meaning the non-general CPU has more frequent changes in the signals.

For practical use, the non-general CPU would be more effective due to its significantly better maximum clock frequency potential and its much lower area. The difference in power dissipation is negligible given its advantages.

Project Planning and Management

Early on during the project each of the group members were assigned Belbin roles based on a survey that was filled out.

Figure 53 – Team members' top predicted Belbin

Name	**Top Two Roles**	
Matilde	Implementer	Shaper
Nelson	Completer Finisher	Implementer
Bernard	Co-ordinator	Plant

The predicted Belbin roles (*Figure 53*) were quite representative of how the team worked together. Bernard was the social leader whilst Matilde made sure that all the deliverables were ready on time and Nelson was continuously working to polish the final product. Frequent and clear communication between members was key and resulted in an organised work flow where each member was aware of what the others were working on at any given point which made it simple to implement design files made by different members.

The meeting frequency was initially weekly, after the first 2 weeks of the project this was increased to every Thursday, Saturday, and Tuesday to accommodate for the increased workflow.

Figure 54 – Initial Gantt chart

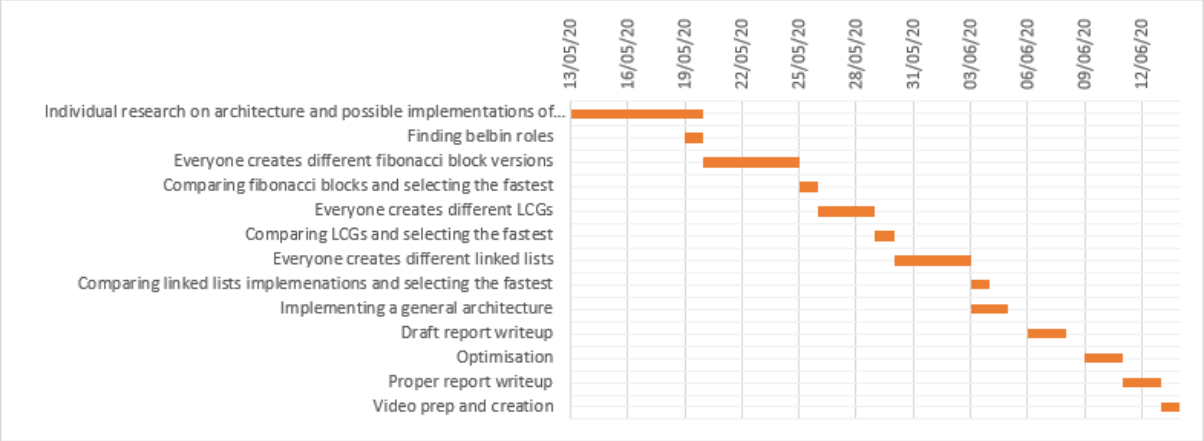
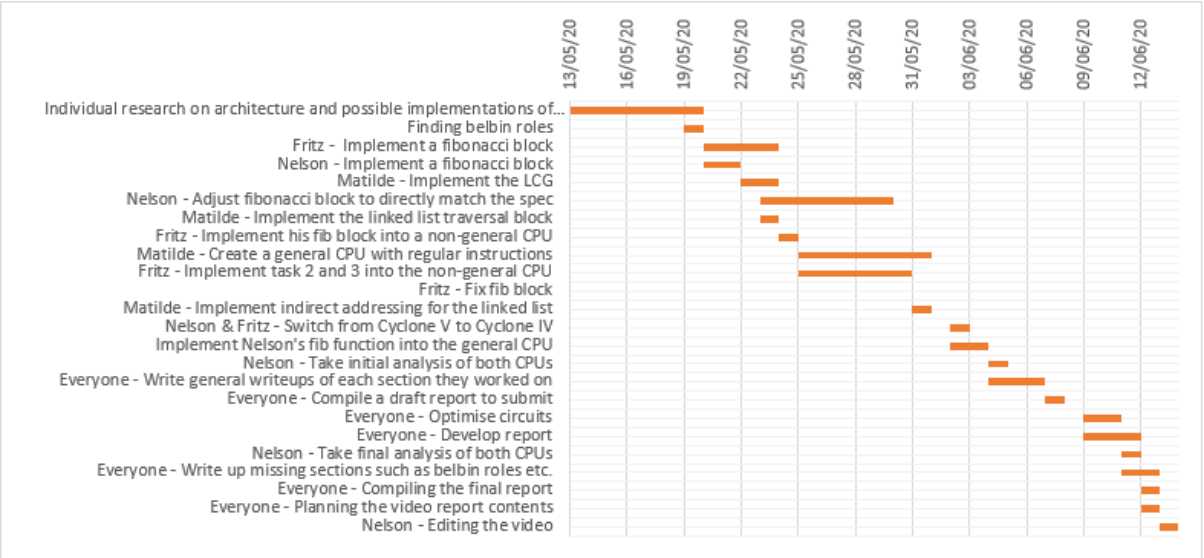


Figure 54 depicts the originally planned series of events/milestones whilst *Figure 55* show how the workflow turned out to be.

Figure 55 – Final Gantt chart



Final Conclusions

The final deliverables were two complete working CPUs, one designed to have a larger range of operations and flexibility; The other an optimisation based on the first which achieves a similar power consumption but at a higher maximum clock speed, sacrificing flexibility for performance. The more general CPU gives the opportunity to easily implement new and more complex tasks in the future whilst the optimised will be credited for its superior performance (relatively speaking).

If given more time, additional tasks would have been implemented into the general CPU. These could include traversing the linked list and instead of searching for a value, a logic block could be implemented to add a node in a sorted linked list. This would be done by finding the first value (B) bigger/smaller than a given value (N). Then storing N with a pointer to the address of B (STR N setting first 12 bits to address at which the search has ended). Subsequently, change the value of pointers of previous elements to their new correct values (this could be done by storing intermediate values in a register whilst the list is being altered).

The initial plan for the project ended was followed through to a satisfactory standard. This largely due to successful communication between members and proper organisation as well as continued documentation from the beginning.

References

[1] *Computer Organisation and Architecture,* UPSC Fever. Accessed on Jun. 13, 2020. [Online]. Available: https://upscfever.com/upsc-fever/en/gatecse/en-gatecse-chp159.html

[2] E. Perea and E. Stott. *ELEC40006: 1st Year Electronics Design Project 2020.* Accessed on Jun. 13, 2020. [Online]. Available: https://bb.imperial.ac.uk/bbcswebdav/pid-1764926-dt-content-rid-6221241_1/courses/13390.201910/EEE1%20Project%202020.pdf

[3] P. Gribble**.** *Memory: Stack vs Heap*. Accessed on Jun. 13, 2020. [Online]. Available: https://gribblelab.org/CBootCamp/7_Memory_Stack_vs_Heap.html

[4] Piazza. Accessed on Jun. 13, 2020. [Online]. Available: https://piazza.com/class/k9n8clkdzsz3nk?cid=37

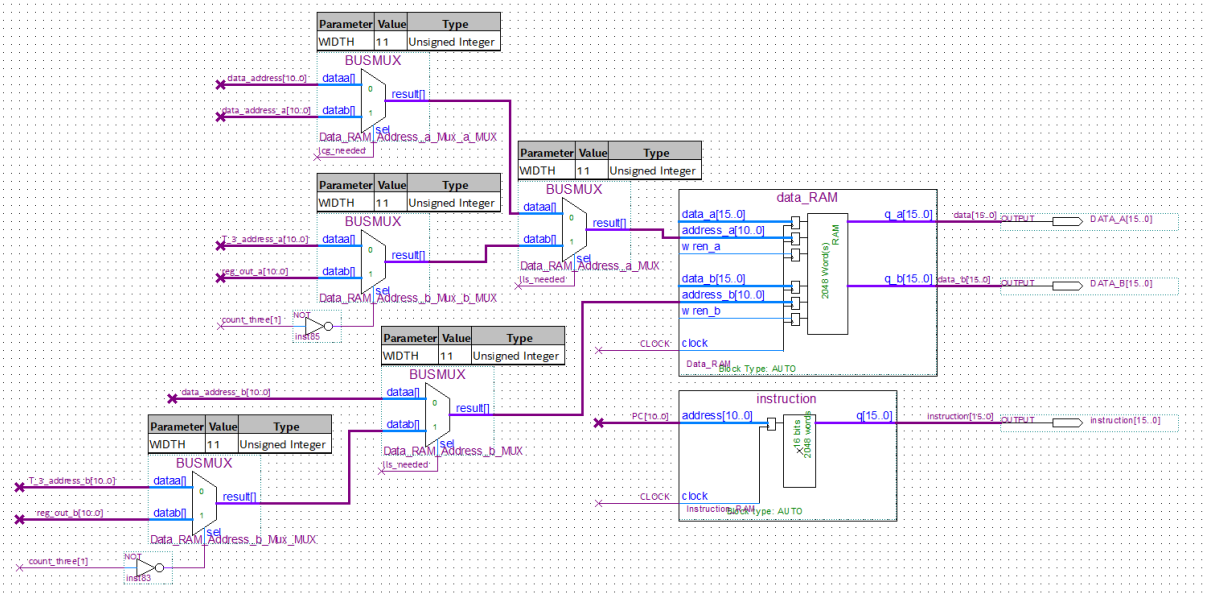
[5] *Difference Between RISC and CISC,* Tech Difference*.* Accessed on Jun. 13, 2020. [Online]. Available: https://techdifferences.com/difference-between-risc-and-cisc.html

[6] *Quartus II Help v13.0,* Intel. Accessed on Jun. 13, 2020. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/quartushelp/13.0/mergedProjects/reference/glossary/def_slack.htm

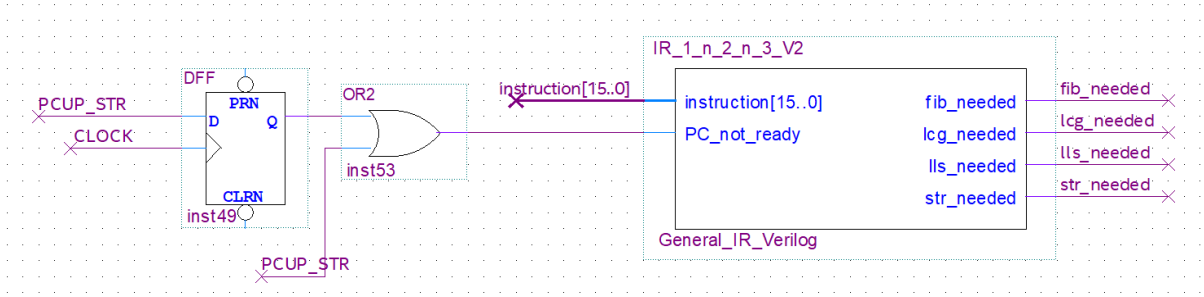
Appendix

Screenshots of complete non-general CPU:

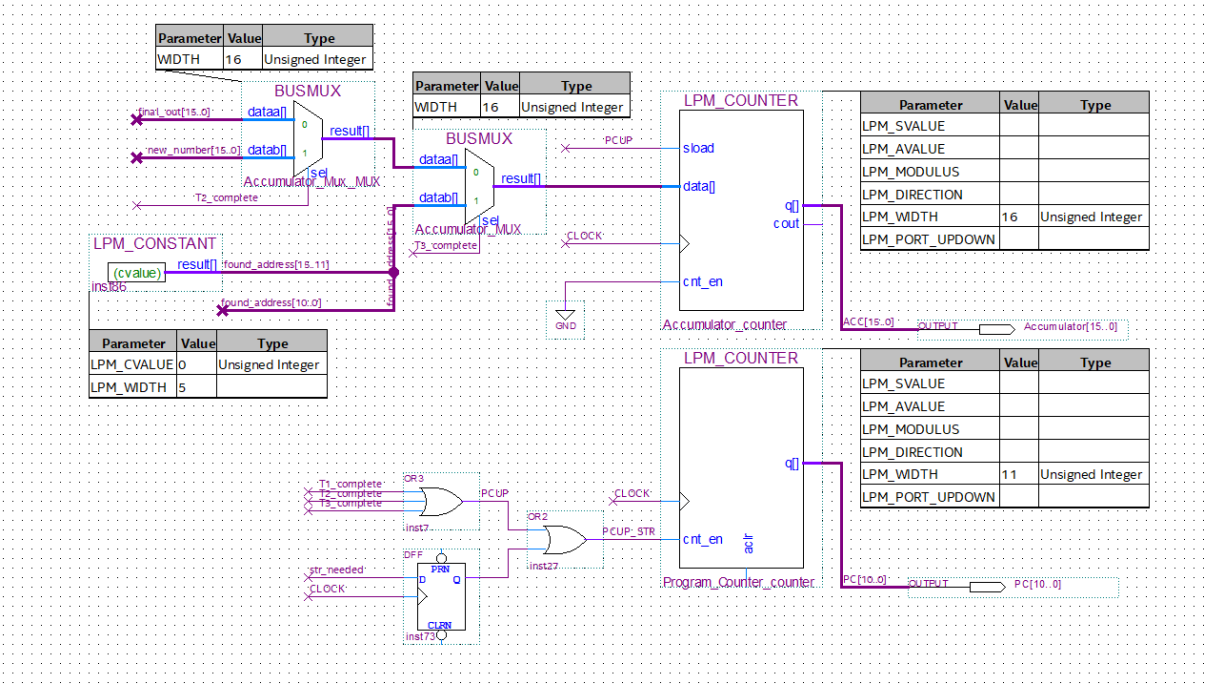
Instruction and Data RAM



General IR



Program Counter and Accumulator



Parameter	Value	Type
WIDTH	16	Unsigned Integer

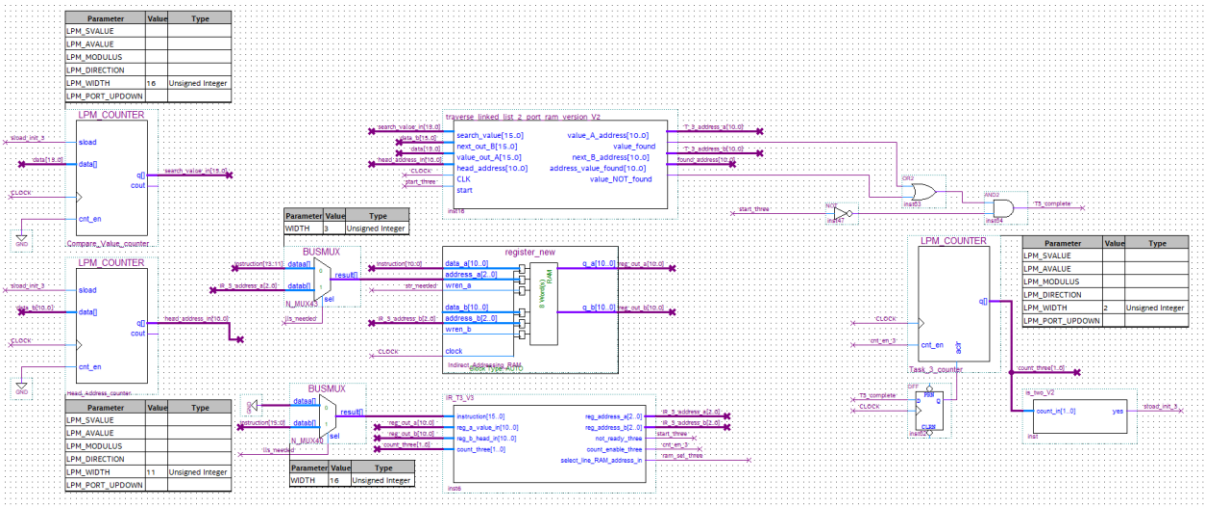
Parameter	Value	Type
WIDTH	16	Unsigned Integer

Parameter	Value	Type
LPM_SVALUE		
LPM_AVALUE		
LPM_MODULUS		
LPM_DIRECTION		
LPM_WIDTH	16	Unsigned Integer
LPM_PORT_UPDOWN		

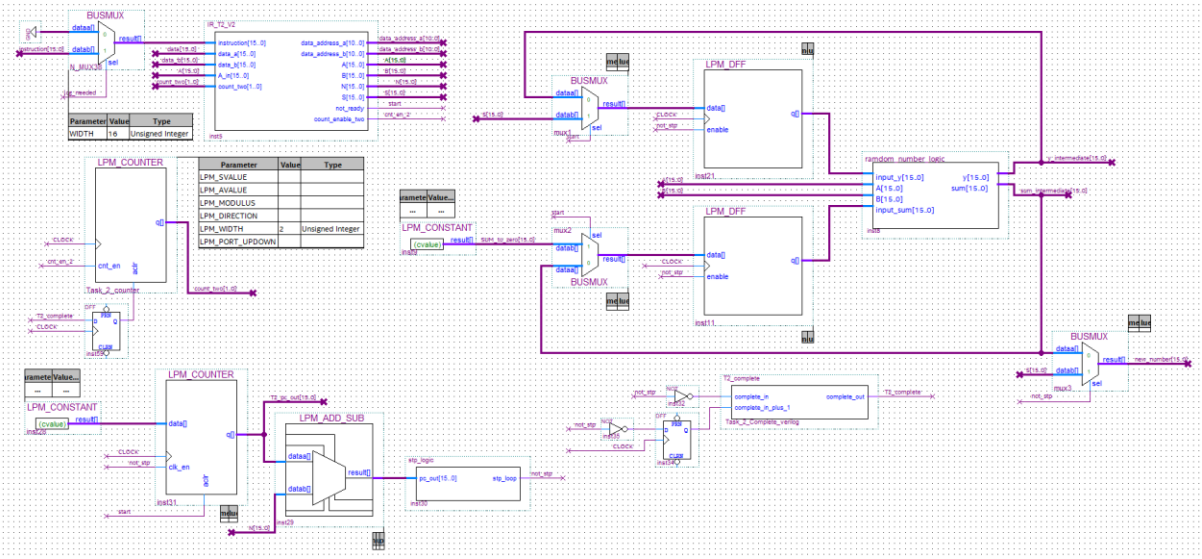
Parameter	Value	Type
LPM_CVALUE	0	Unsigned Integer
LPM_WIDTH	5	

Parameter	Value	Type
LPM_SVALUE		
LPM_AVALUE		
LPM_MODULUS		
LPM_DIRECTION		
LPM_WIDTH	11	Unsigned Integer
LPM_PORT_UPDOWN		

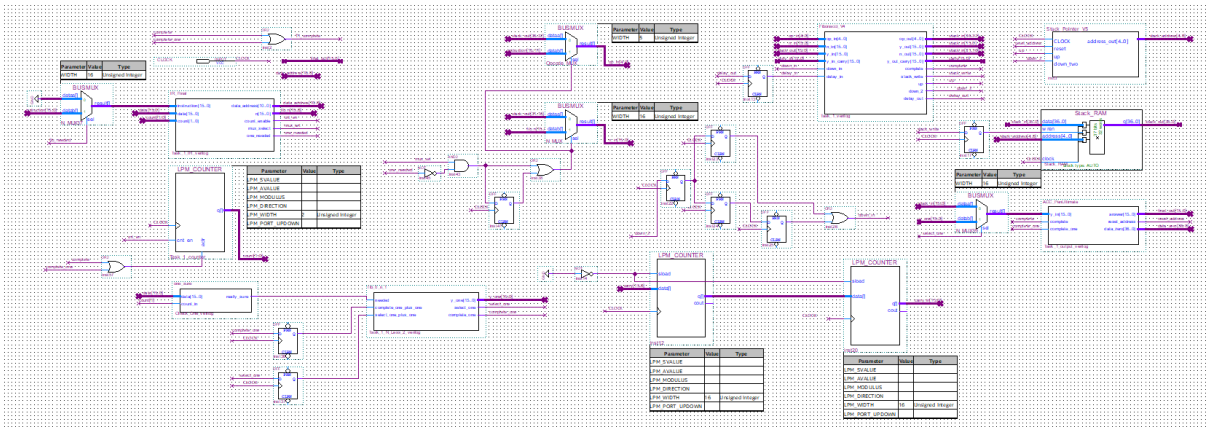
Blocks for Linked List Search Instruction



Blocks for Random Number Instruction



Blocks for Fibonacci Instruction



Verilog Code from the Non-general IR

General IR

```
module IR_1_n_2_n_3_V2

(
    input [15:0] instruction,
    input PC_not_ready,

    output fib_needed,
    output lcg_needed,
    output lls_needed,
    output str_needed
);

wire [4:0] OP;

assign OP = instruction[15:11];

assign fib_needed = (OP == 1) & !PC_not_ready;
assign lcg_needed = (OP == 2) & !PC_not_ready;
assign lls_needed = (OP == 3) & !PC_not_ready;
assign str_needed = (OP > 7) & !PC_not_ready;

endmodule
```

Fibonacci Task IR

```
module IR_Final
(
    input [15:0] instruction,
    input [15:0] data,
    input [1:0] count,

    output [10:0] data_address,
    output [15:0] n,
    output count_enable,
    output mux_select,
);

wire OP1;
wire C0;
wire C1;

assign OP1 = !instruction[15] & !instruction[14] & !instruction[13] & !instruction[12] & instruction[11];
assign C0 = !count[1] & !count[0];
assign C1 = !count[1] & count[0];

assign count_enable = (OP1 &  C0) | (C1);
assign mux_select = C1;
assign n[15:0] = data[15:0];
assign data_address[10:0] = instruction[10:0];

endmodule
```

Fibonacci Task – block to check if n < 2 at when the task counter is not equal to zero

```
module one_sure

(
    input [15:0] data,
    input count_in,

    output really_sure
);

assign really_sure = (data < 2) & count_in;

endmodule
```

Fibonacci Task – block used to output correct signals to the task output register when n < 2

```
module fib_0_n_1

(
    input needed,
    input complete_one_plus_one,
    input select_one_plus_one,

    output [15:0] y_one,
    output select_one,
    output complete_one
);

assign y_one = 16'b0000000000000001;
assign select_one = needed & !select_one_plus_one;
assign complete_one = needed & !complete_one_plus_one;

endmodule
```

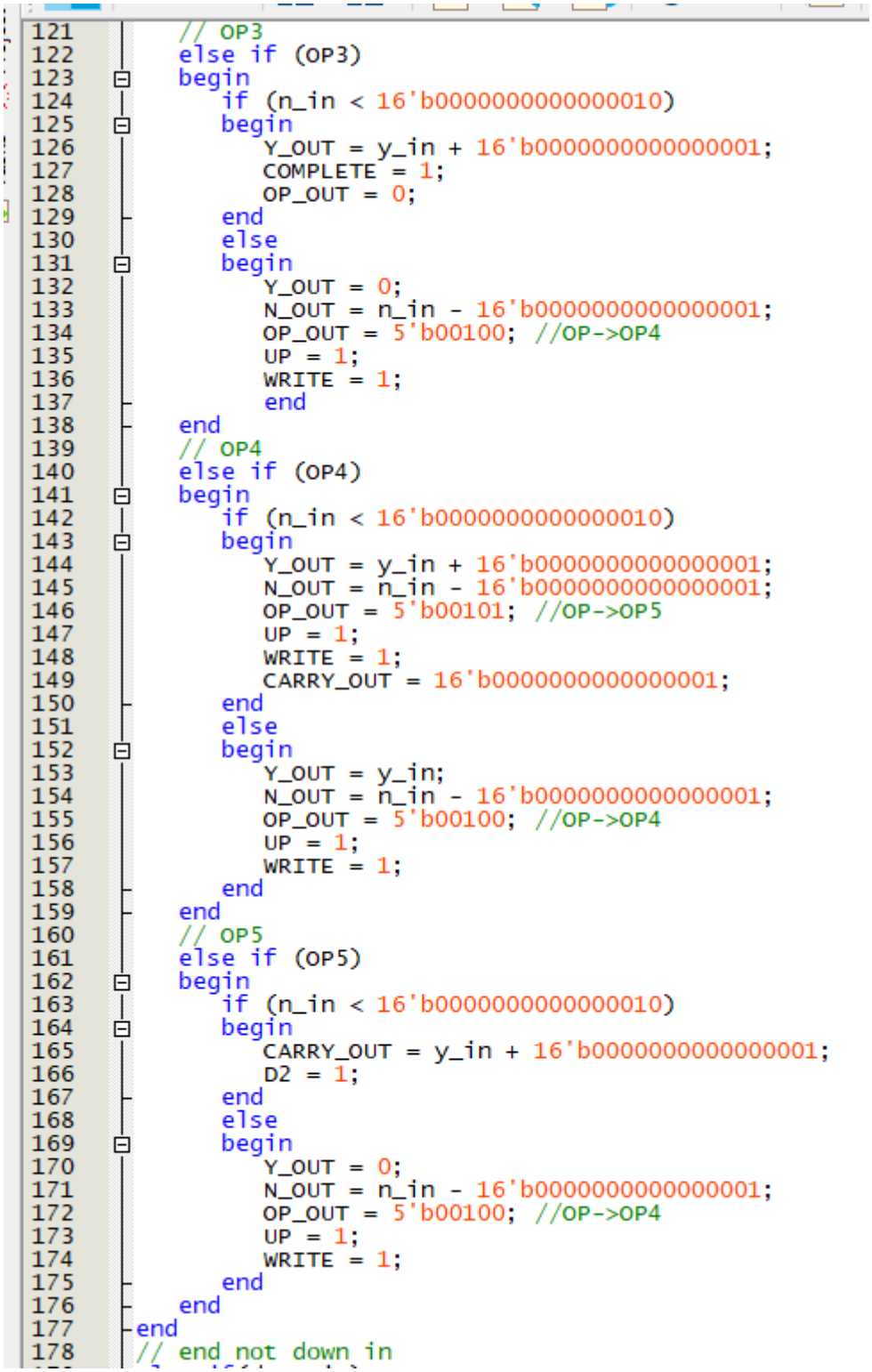
Main Fibonacci Operating Block – 1/5

```
module Fibonacci_V4

(
    // input fib_ready,
    input [4:0] op_in,
    input [15:0] n_in,
    input [15:0] y_in,

    input [15:0] y_in_carry,
    input down_in,

    input delay_in,

    //input complete_in,

    output [4:0] op_out,
    output [15:0] y_out,
    output [15:0] n_out,

    output [15:0] y_out_carry,
    output complete,
    output stack_write,
    output up,
    //output delay_start,
    output down_2,

    output delay_out
);

wire OP0;
wire OP1;// fib(N) from instruction register
wire OP2;// fib(n-1) from instruction register
wire OP3;// fib(n-2) from instruction register
wire OP4;// fib(n-1) from a recursion
wire OP5;// fib(n-2) from a recursion

assign OP0 = (op_in[4:0] == 5'b00000);
assign OP1 = (op_in[4:0] == 5'b00001);
assign OP2 = (op_in[4:0] == 5'b00010);
assign OP3 = (op_in[4:0] == 5'b00011);
assign OP4 = (op_in[4:0] == 5'b00100);
assign OP5 = (op_in[4:0] == 5'b00101);

reg COMPLETE;
reg D1;
reg D2;
reg UP;
reg WRITE;
reg DO;

// reg DOP;

reg [15:0] Y_OUT;
reg [15:0] CARRY_OUT;
reg [15:0] N_OUT;
reg [4:0] OP_OUT;
```

Main Fibonacci Operating Block – 2/5

```
58    // start main function
59    always@(*)
60    begin
61
62    COMPLETE = 0;
63    D1 = 0;
64    D2 = 0;
65    UP = 0;
66    WRITE = 0;
67    //Y_OUT = 0;
68    //N_OUT = 0;
69    //OP_OUT = 0;
70    DO = 0;
71    //DOP = 0;
72
73    if (!delay_in)
74    // start not delay
75    begin
76
77    DO = 1;
78
79    // start not down in
80    if (!down_in)
81    begin
82       // OP1
83       if (OP1)
84       begin
85          if (n_in < 16'b0000000000000010)
86          begin
87             Y_OUT = 16'b0000000000000001;
88             COMPLETE = 1;
89             OP_OUT = 0;
90          end
91          else
92          begin
93             Y_OUT = 16'b0000000000000;
94             N_OUT = n_in - 16'b0000000000000001;
95             OP_OUT = 5'b00010; //OP->OP2
96             UP = 1;
97             WRITE = 1;
98             CARRY_OUT = 0;
99          end
100      end
101      // OP2
102      else if (OP2)
103      begin
104         if (n_in < 16'b0000000000000010)
105         begin
106            Y_OUT = 16'b0000000000000001;
107            N_OUT = n_in - 16'b0000000000000001;
108            OP_OUT = 5'b00011; //OP->OP3
109            UP = 1;
110            WRITE = 1;
111         end
112         else
113         begin
114            Y_OUT = 16'b0000000000000000;
115            N_OUT = n_in - 16'b0000000000000001;
116            OP_OUT = 5'b00100; //OP->OP4
117            UP = 1;
118            WRITE = 1;
119         end
120      end
```

Main Fibonacci Operating Block – 3/5



```
121     // OP3
122     else if (OP3)
123     begin
124         if (n_in < 16'b0000000000000010)
125         begin
126             Y_OUT = y_in + 16'b0000000000000001;
127             COMPLETE = 1;
128             OP_OUT = 0;
129         end
130         else
131         begin
132             Y_OUT = 0;
133             N_OUT = n_in - 16'b0000000000000001;
134             OP_OUT = 5'b00100; //OP->OP4
135             UP = 1;
136             WRITE = 1;
137             end
138     end
139     // OP4
140     else if (OP4)
141     begin
142         if (n_in < 16'b0000000000000010)
143         begin
144             Y_OUT = y_in + 16'b0000000000000001;
145             N_OUT = n_in - 16'b0000000000000001;
146             OP_OUT = 5'b00101; //OP->OP5
147             UP = 1;
148             WRITE = 1;
149             CARRY_OUT = 16'b0000000000000001;
150         end
151         else
152         begin
153             Y_OUT = y_in;
154             N_OUT = n_in - 16'b0000000000000001;
155             OP_OUT = 5'b00100; //OP->OP4
156             UP = 1;
157             WRITE = 1;
158         end
159     end
160     // OP5
161     else if (OP5)
162     begin
163         if (n_in < 16'b0000000000000010)
164         begin
165             CARRY_OUT = y_in + 16'b0000000000000001;
166             D2 = 1;
167         end
168         else
169         begin
170             Y_OUT = 0;
171             N_OUT = n_in - 16'b0000000000000001;
172             OP_OUT = 5'b00100; //OP->OP4
173             UP = 1;
174             WRITE = 1;
175         end
176     end
177 end
178 // end not down in
```

Main Fibonacci Operating Block – 4/5

```
179     else if(down_in)
180     // start down in
181     begin
182         // OP2
183         if (OP2)
184         begin
185             Y_OUT = y_in_carry;
186             N_OUT = n_in - 16'b0000000000000001;
187             OP_OUT  = 5'b00011; //OP->OP3
188             UP = 1;
189             WRITE = 1;
190             CARRY_OUT = 0;
191         end
192         // OP3
193         else if (OP3)
194         begin
195             Y_OUT = y_in + y_in_carry;
196             COMPLETE = 1;
197             OP_OUT = 0;
198         end
199         // OP4
200         else if (OP4)
201         begin
202             Y_OUT = y_in + y_in_carry;
203             N_OUT = n_in - 16'b0000000000000001;
204             OP_OUT = 5'b00101; //OP->OP5
205             UP = 1;
206             WRITE = 1;
207             CARRY_OUT = y_in_carry;
208         end
209         // OP5
210         else if (OP5)
211         begin
212             CARRY_OUT = y_in + y_in_carry;
213             D2 = 1;
214         end
215     end
216     // end down in
217     end
218     // end not delay
219
220     else if (delay_in)
221     // start delay
222     begin
223     // CARRY_OUT = y_in_carry;
224     DO = 0;
225
226     //if (OP1)
227     //begin
228     // DOP = 1;
229     //end
230
231     //Y_OUT = y_in;
232     //N_OUT = n_in;
233     //OP_OUT = op_in;
234
235     end
236     // end delay
237
238     end
239     // end function
240
```

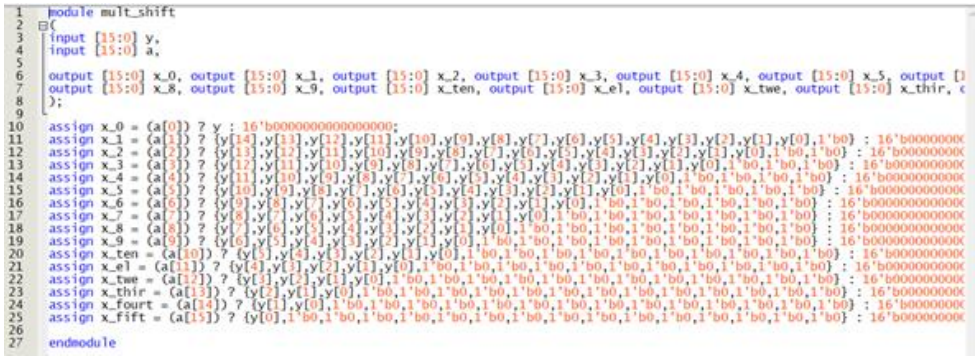
Main Fibonacci Operating Block – 5/5

```
241    assign op_out = OP_OUT;
242    assign y_out = Y_OUT;
243    assign n_out = N_OUT;
244    assign y_out_carry = CARRY_OUT;
245    assign complete = COMPLETE;
246    assign stack_write = WRITE;
247    assign up = UP;
248    //assign delay_start = 0;
249    assign down_2 = D2;
250    assign delay_out = DO;
251
252    endmodule
```

Fibonacci Task – output block

```
1   module ACC_PenUltimate
2
3   (
4       input [15:0] y_in,
5       input complete,
6       input complete_one,
7
8       output [15:0] answer,
9       output reset_address,
10      output [36:0] data_zero
11  );
12
13  reg [15:0] OUT;
14
15  always @(*)
16  begin
17
18  if (complete || complete_one)
19  begin
20      OUT = y_in;
21
22  end
23  else
24  begin
25  OUT = 0;
26  end
27
28  end
29
30  assign answer = OUT;
31  assign reset_address = complete | complete_one;
32  assign data_zero = 37'b0000000000000000000000000000000000000;
33
34  endmodule
```

Task 2 multiplication

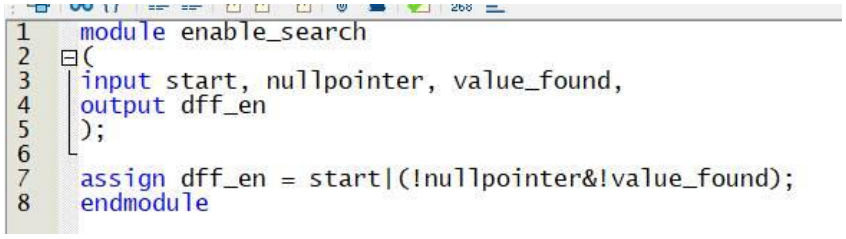


```
module mult_shift
(
input [15:0] y,
input [15:0] a,

output [15:0] x_0, output [15:0] x_1, output [15:0] x_2, output [15:0] x_3, output [15:0] x_4, output [15:0] x_5, output [1
output [15:0] x_8, output [15:0] x_9, output [15:0] x_ten, output [15:0] x_el, output [15:0] x_twe, output [15:0] x_thir, o
);

assign x_0 = (a[0]) ? y : 16'b0000000000000000;
assign x_1 = (a[1]) ? {y[14],y[13],y[12],y[11],y[10],y[9],y[8],y[7],y[6],y[5],y[4],y[3],y[2],y[1],y[0],1'b0} : 16'b0000000
assign x_2 = (a[2]) ? {y[13],y[12],y[11],y[10],y[9],y[8],y[7],y[6],y[5],y[4],y[3],y[2],y[1],y[0],1'b0,1'b0} : 16'b0000000
assign x_3 = (a[3]) ? {y[12],y[11],y[10],y[9],y[8],y[7],y[6],y[5],y[4],y[3],y[2],y[1],y[0],1'b0,1'b0,1'b0} : 16'b0000000
assign x_4 = (a[4]) ? {y[11],y[10],y[9],y[8],y[7],y[6],y[5],y[4],y[3],y[2],y[1],y[0],1'b0,1'b0,1'b0,1'b0} : 16'b0000000
assign x_5 = (a[5]) ? {y[10],y[9],y[8],y[7],y[6],y[5],y[4],y[3],y[2],y[1],y[0],1'b0,1'b0,1'b0,1'b0,1'b0} : 16'b0000000
assign x_6 = (a[6]) ? {y[9],y[8],y[7],y[6],y[5],y[4],y[3],y[2],y[1],y[0],1'b0,1'b0,1'b0,1'b0,1'b0,1'b0} : 16'b0000000
assign x_7 = (a[7]) ? {y[8],y[7],y[6],y[5],y[4],y[3],y[2],y[1],y[0],1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0} : 16'b0000000
assign x_8 = (a[8]) ? {y[7],y[6],y[5],y[4],y[3],y[2],y[1],y[0],1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0} : 16'b0000000
assign x_9 = (a[9]) ? {y[6],y[5],y[4],y[3],y[2],y[1],y[0],1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0} : 16'b0000000
assign x_ten = (a[10]) ? {y[5],y[4],y[3],y[2],y[1],y[0],1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0} : 16'b0000000
assign x_el = (a[11]) ? {y[4],y[3],y[2],y[1],y[0],1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0} : 16'b0000000
assign x_twe = (a[12]) ? {y[3],y[2],y[1],y[0],1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0} : 16'b0000000
assign x_thir = (a[13]) ? {y[2],y[1],y[0],1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0} : 16'b0000000
assign x_fourt = (a[14]) ? {y[1],y[0],1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0} : 16'b0000000
assign x_fift = (a[15]) ? {y[0],1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0} : 16'b0000000

endmodule
```

stp lcg

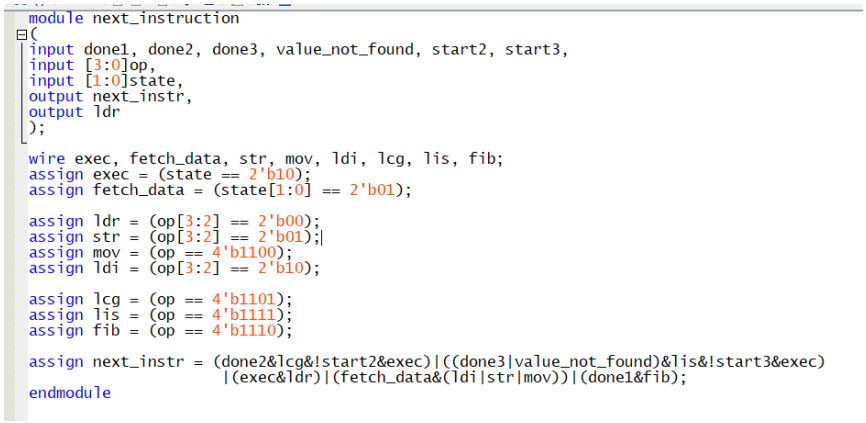
```
module stp_logic
(
input [15:0] pc_out,
output stp_loop
);

assign stp_loop = !(!pc_out[15]&!pc_out[14]&!pc_out[13]&!pc_out[12]&!pc_out[11]&!pc_out[10]&!pc_out[9]&!pc_out[8]&!pc_out[7]
endmodule
```

Enable LIS



```
module enable_search
(
input start, nullpointer, value_found,
output dff_en
);

assign dff_en = start|(!nullpointer&!value_found);
endmodule
```

Next instruction



```
module next_instruction
(
input done1, done2, done3, value_not_found, start2, start3,
input [3:0]op,
input [1:0]state,
output next_instr,
output ldr
);

wire exec, fetch_data, str, mov, ldi, lcg, lis, fib;
assign exec = (state == 2'b10);
assign fetch_data = (state[1:0] == 2'b01);

assign ldr = (op[3:2] == 2'b00);
assign str = (op[3:2] == 2'b01);
assign mov = (op == 4'b1100);
assign ldi = (op[3:2] == 2'b10);

assign lcg = (op == 4'b1101);
assign lis = (op == 4'b1111);
assign fib = (op == 4'b1110);

assign next_instr = (done2&lcg&!start2&exec)|((done3|value_not_found)&lis&!start3&exec)
                    |(exec&ldr)|(fetch_data&(ldi|str|mov))|(done1&fib);
endmodule
```

State machine

```
module state_machine
(
input [2:0]pc_state_out,
input [1:0] previous_state,
input done,
input [3:0]opcode,
output reg [1:0] current_state,
output lcg, start_fetch, start2, start3, cnt_en_state, clear_state_cnt, start1
);

wire fetch_data, fetch_instr, strart, exec;

assign lcg = (opcode == 4'b1101);

assign fetch_instr = ((previous_state == 2'b00)&((pc_state_out == 3'b001)|(pc_state_out == 3'b010)))|done;
assign fetch_data = ((previous_state == 2'b00)&(pc_state_out == 3'b011))|((previous_state == 2'b01)&((pc_state_out == 3'b100)|((pc_state_out == 3'b101)&lcg)));
assign exec = (previous_state == 2'b01)&((pc_state_out == 3'b101)&!lcg)|((pc_state_out == 3'b110)&lcg))|((previous_state == 2'b10)&(!done));

assign start_fetch = ((previous_state == 2'b00)&fetch_data);
assign start = ((previous_state == 2'b01)&exec);

assign start1 = exec&(opcode == 4'b1110);
assign start2 = start&lcg;
assign start3 = start&(opcode == 4'b1111);

//set current state: 00 op fetch instr, 01 op fetch data, 10 op exec
always @ (current_state or 2'b00 or 2'b01 or 2'b10)
begin
      if (exec)
         current_state = 2'b10;
      else if (fetch_data)
         current_state = 2'b01;
      else
         current_state =  2'b00;
end

assign cnt_en_state = ((current_state == 2'b00)|(current_state == 2'b01));
assign clear_state_cnt = done;

endmodule
```

Decoder

```
module decoder
(
input [3:0] op,
input [15:0]instr,
input done,
input [1:0]state,
input [2:0]pc_out_state,//? 00fetch_data, 01 exec
output acc_en, count_en_instr, ir_en, wr_en_reg, wr_en_ram
);

wire lcg, lis, mov, ldr, str, fib;

assign ldr = (op[3:2] == 2'b00);
assign str = (op[3:2] == 2'b01);
assign mov = (op[3:0] == 4'b1100);
assign ldi = (op[3:2] == 2'b10);
assign fib = (instr[15:12] == 4'b1110);
assign lcg = (instr[15:12] == 4'b1101);
assign lis = (instr[15:12] == 4'b1111);

wire fetch_instr, fetch_data, exec;

assign fetch_instr = (state[1:0] == 2'b00);
assign fetch_data = (state[1:0] == 2'b01);
assign exec = (state[1:0] == 2'b10);

assign acc_en = (lcg&done)|(lis&done)|(ldi&done)|(fib&done);//|(lda&exec);
assign count_en_instr = done;//previous version: didn't work for first instr 0000 :fetch_instr&(pc_out_state == 3'b001);
assign ir_en = fetch_data&(pc_out_state == 3'b011);//first cycle of fetch data
assign wr_en_reg = (exec&(ldr))|(fetch_data&(mov));//timing corresponding to done for ldr, ldi and mov
assign wr_en_ram = (fetch_data&(str));

endmodule
```

Accumulator input logic 5

```
module data_in_RAM
(
input [11:0] addr_prev,
input [3:0]opcode,
input [15:0] IR,
input start_fetch,
input [15:0] Reg_a, Reg_b, Reg_c,
input [11:0] addr_pointer,
//input [1:0] state,
output [27:0]data_in_ram,
output reg [11:0] addr1,
output [11:0]addr2, addr1_next,
output fetch2_lcg
);

wire lcg, lis, ldr, str, fib;
assign lcg = (opcode[3:0] == 4'b1101);
assign lis = (opcode[3:0] == 4'b1111);
assign ldr = (opcode[3:2] == 2'b00);
assign str = (opcode[3:2] == 2'b01);
assign fib = (opcode[3:0] == 4'b1110);
//assign fetch_data = (state == 2'b01);
//str write ram
assign data_in_ram = {12'b000000000000, Reg_a};
//lcg: Reg_a = A (1), Reg_b= B(2), Reg_c = S(1_next) and lis: Reg_a = starting addr(1), Reg_b = value to be found(2)

//assign addr1 = (start&(lcg|lis)? Reg_a[11:0] : 12'b000000000000); if else lis and not start fetch: addr from pointer
 always @ (addr1 or Reg_a[11:0] or addr_pointer or addr_prev or IR[13:2] or IR[11:0])
begin
    //if (lda|sta)
    // addr1 = IR[11:0];
    //else begin
      if (start_fetch&(lcg|lis))
         addr1 = Reg_a[11:0];
      else if (lis&(!start_fetch))
        addr1 = addr_pointer[11:0];
      else if (str|ldr)//start fetch ?
         addr1 = IR[13:2];
      else if (fib&start_fetch)
         addr1 = IR[11:0];
      else
         addr1 =  addr_prev;
    //end
  end
//lda and sta implement addr from 12 least significant bits IR and write from acc and task1 implement
assign addr2 = ((lcg|lis)? Reg_b[11:0] : 12'b000000000000);
assign addr1_next = (lcg ? Reg_c[11:0] : 12'b000000000000);
assign fetch2_lcg = (lcg&start_fetch ? 1'b1 : 1'b0);
```

Data in ram

```
module acc_in_logic
(
input [15:0] acc_out_prev, task2_result, data_out_ram, reg_b, task1_result,
input [3:0] instr_out,
input [11:0] task3_result,
input [3:0] IR,
output [15:0] acc_in,
output [15:0] reg_data_in
);
//task2, task3, ldi
always @ (acc_in or {4'b0000, task3_result} or task2_result or acc_out_prev or task1_result)
 begin
       if (IR[3:0]== 4'b1111)//task 2
          acc_in = {4'b0000, task3_result};
       else if (IR[3:0]== 4'b1101)//task2
          acc_in = task2_result;
       else if (IR[3:2]== 2'b10)//ldi
          acc_in = {2'b00, instr_out};
       else if (IR[3:0] == 4'b1110)
          acc_in = task1_result;
       else
          acc_in =  acc_out_prev;
end

//mov and lda
always @ (reg_data_in or data_out_ram or reg_b or 16'b0000000000000000)
 begin
       if (IR[3:0]== 4'b1100)//mov
          reg_data_in = reg_b;
       else if (IR[3:2]== 2'b00)//ldr
         reg_data_in = data_out_ram;
       else
          reg_data_in = 16'b0000000000000000;
end

endmodule
```

Nelson's main Fibonacci block code

<u>Valuecheck if conditions</u>

If condition '1'

```
//1
//when we get to a value of n <= 1 where a 1 is returned and must add to the return address
if (n <= 1 & ((ActualPV != 0 & EXEC1) | (PreviousValue != 0 & (EXEC2 | EXEC3))))
begin
    if (EXEC1 == 1)
    begin
        newdata = 1;
        //newraddress = 0;
        //newn = 1;
        newAddress = raddress;
        RAMwren = 0;
    end
    else if (EXEC2 == 1)
    begin
        RAMwren = 0;
    end
    else if (EXEC3 == 1)
    begin
        RAMwren = 1;
    end

    MUX3 = 0;
    MUX4 = EXEC2 | EXEC3;

    sclr = 0;
    SRload = 1;

    EXTRA = 1;

    FVload = 0;
end
```

If condition ‘2’

```
//2
//when we are at the start of the stack register
//stops any operations occuring without data first being input
else if (RAMout == 0 & PreviousValue == 0 & initialpulse == 0)
begin
    if (EXEC1 == 1)
        begin
        newdata = 0;
        newraddress = 0;
        newn = 0;

        newAddress = 0;
        end

    MUX3 = 0;
    MUX4 = 0;

    sclr = 0;
    SRload = 0;

    RAMwren = 0;

    EXTRA = 0;

    FVload = 0;
end
```

If condition ‘3’

```
//3
//when initialising the first couple of values starting from n
//no confliciton errors when doing the addition since this block gets skipped

else if ((n > 1) & data == 0 & (initialpulse == 0))
begin
    if (EXEC1 == 1)
        begin
        newAddress = RAMaddress + {4'b0,1'b1};
        end

    else if (EXEC2 == 1)
    begin
        newdata = data;

        if (PreviousAddress == 0)
            begin
            newraddress = 0;
            end
        else if (PreviousAddress != 0)
            begin
            newraddress = PreviousAddress;
            end

        newn = (n - {15'b0, 1'b1});

        newAddress = RAMaddress; //so that the address is unchanged
    end

    SRload = 1;

    RAMwren = EXEC2;

    MUX3 = 0;
    MUX4 = 0;

    sclr = 0;

    EXTRA = 0;

    FVload = 0;
end
```

If condition ‘4’

```
//4
//when we have already done fib(n-1) and are now doing fib(n-2)
//previous address would be the address just below this
else if ((n > 1) & data != 0 & (previousn == n - 1) ) //last AND condition used to differ from next if
begin

   if (EXEC1 == 1)
   begin
      newdata = 0;

      if (raddress == 0 & (n == initialn))
      begin
         newraddress = 0;
      end
      else
      begin
         newraddress = (raddress + {4'b0, 1'b1});
      end


      newn = (n - {14'b0, 1'b1, 1'b0});

      newAddress = (RAMaddress + {4'b0, 1'b1});
   end
   else if (EXEC2 == 1)
   begin
      newAddress = RAMaddress;
   end

   RAMwren = EXEC2;

   MUX3 = 0;
   MUX4 = 0;

   sclr = 0;
   SRload = 1;

   EXTRA = 0;

   FVload = 0;
end
```

If condition ‘5’

```
//5
//when we have already done fib(n-2) and now need to add to the return address
//for values at an address below the return address
else if ((n > 1) & data != 0 & (previousn == n - 2) & !(n == initialn))
   begin
   if (EXEC1 == 1)
   begin
      newAddress = raddress;
      newdata = data;
   end

      RAMwren = EXEC3;

      MUX3 = 0;
      MUX4 = EXEC2 | EXEC3;

      sclr = 0;
      SRload = 1;

      EXTRA = 1;

      FVload = 0;
   end
```

If condition ‘6’

```
//6
//when we've completed everything and the final value is stored in 0000
else if ((n > 1) & data != 0 & (previousn == n - 2) & (RAMaddress == 0) & (n == initialn))
   begin
   if (EXEC1 == 1)
   begin
      newdata = 0;
      newraddress = 0;
      newn = 0;
      newAddress = 0;
   end

      RAMwren = 1;

      MUX3 = 0;
      MUX4 = 0;

      sclr = EXEC1;
      SRload = 1;

      EXTRA = 0;

      FVload = 1;
   end
```

If condition ‘initialising’

```
//initially loading the first value
   else if ((initialpulse == 1) & initialn != 0 & initialn != 1)
      begin
      if (EXEC1 == 1)
      begin
         newdata = data;
         newraddress = raddress;
         newn = n;
         newAddress = RAMaddress;
      end
      else if (EXEC2 == 1)
      begin
         newdata = data;
         newraddress = raddress;
         newn = n;

         newAddress = RAMaddress;
      end

         RAMwren = EXEC2;

         MUX3 = 1;
         MUX4= 0;

         sclr = 0;
         SRload = 1;

         EXTRA = 0;

         FVload = 0;
      end
```

If condition ‘oneorzero’

```
//oneorzero
else if ((initialn == 1 | initialn == 0 ) & initialpulse == 1 & RAMaddress == 0 & ActualPV == 0)
   begin
   if (EXEC1 == 1)
      begin
      newdata = 1;

      RAMwren = 1;
      end
   else if (EXEC2 == 1)
      begin
      newdata = 0;
      newraddress = 0;
      newn = 0;

      RAMwren = 1;
      end

      sclr = EXEC2;
      SRload = 1;

      MUX3 = 0;

      FVload = EXEC2;

      EXTRA = 0;
   end
```