Task 1: The Fibonacci function

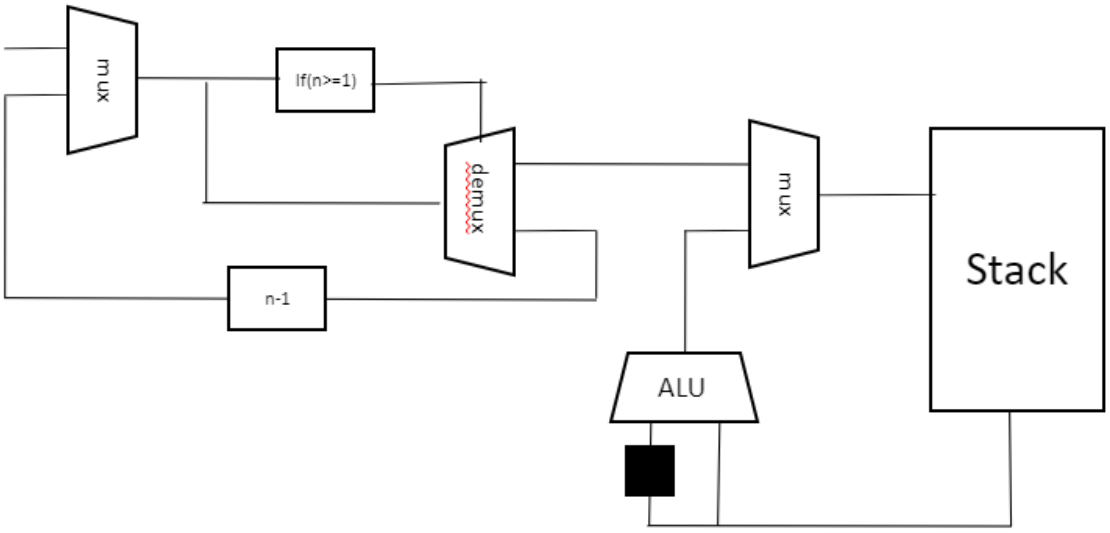
The task was to implement the recursive function shown below which acquires the nth term of the Fibonacci sequence under the requirement that it uses a stack to work with temporary values created in the process. The stack could be implemented with either custom hardware or normal data memory. (Insert reference to the spec)

```
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' (https://gribblelab.org/CBootCamp/7_Memory_Stack_vs_Heap.html). This means whenever a fib() function is called, a spot in the stack is allocated for this variable until it has been fully completed with a returned variable. Then, it's spot in the stack is freed and 'that region of memory becomes available for other stack variables.' (https://gribblelab.org/CBootCamp/7_Memory_Stack_vs_Heap.html). Therefore, stack is usually referred to as 'last in, first out' data structure (https://gribblelab.org/CBootCamp/7_Memory_Stack_vs_Heap.html) due to how the most recent function call is freed up first.

Traversing a stack is possible with a stack pointer, 'a register holding the address for the stack' (https://upscfever.com/upsc-fever/en/gatecse/en-gatecse-chp159.html). A stack pointer makes it possible to choose where new data is inserted and where data must be freed up after a function has been completed.

Early drafts worked around the idea of taking an input n and decrementing it using verilog logic until a 1 is returned; as the value would decrement, signals would be sent around the system to calculate towards the final value. One of these drafts consisted of a system where an input would determine where the returned value 1 is initially stored.

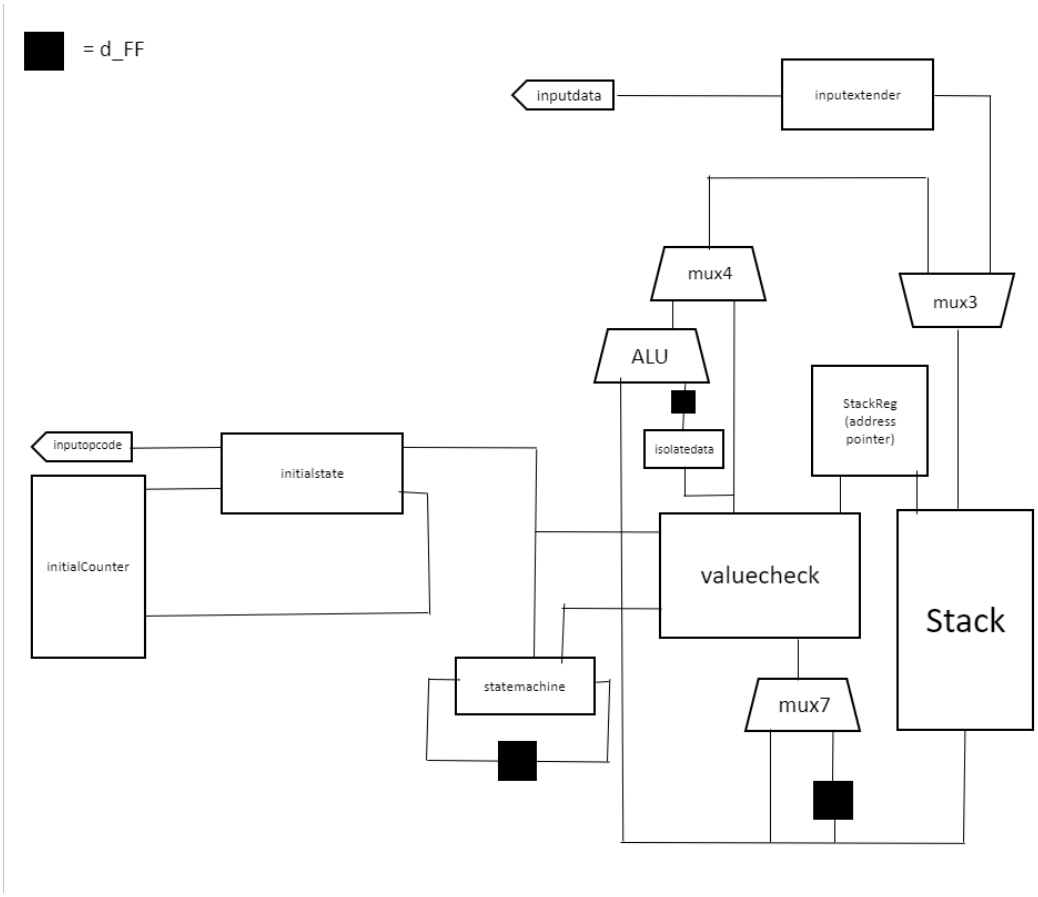


Once the initial returned variable 1 is stored, a recurring addition system where the stack pointer decrements would ultimately lead to the final returned variable being stored in 0^{th} position of the stack. However, it later became clear that this is not a proper representation of the given Fibonacci function implementation despite it providing the desired results; this would have not met the provided specification (especially since it wasn't the proper use of stack memory where the 'last in, first out' concept should be followed). Additionally, despite the system's simplicity, it would not be able be adjusted so that other recursive functions could work with the architecture; this design would only be suitable for functions where repeated addition of the previous two values occurs.

Further research clarified how the stack should properly be implemented. 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. 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.

Paragraph information sourced from https://piazza.com/class/k9n8clkdzsz3nk?cid=37

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.



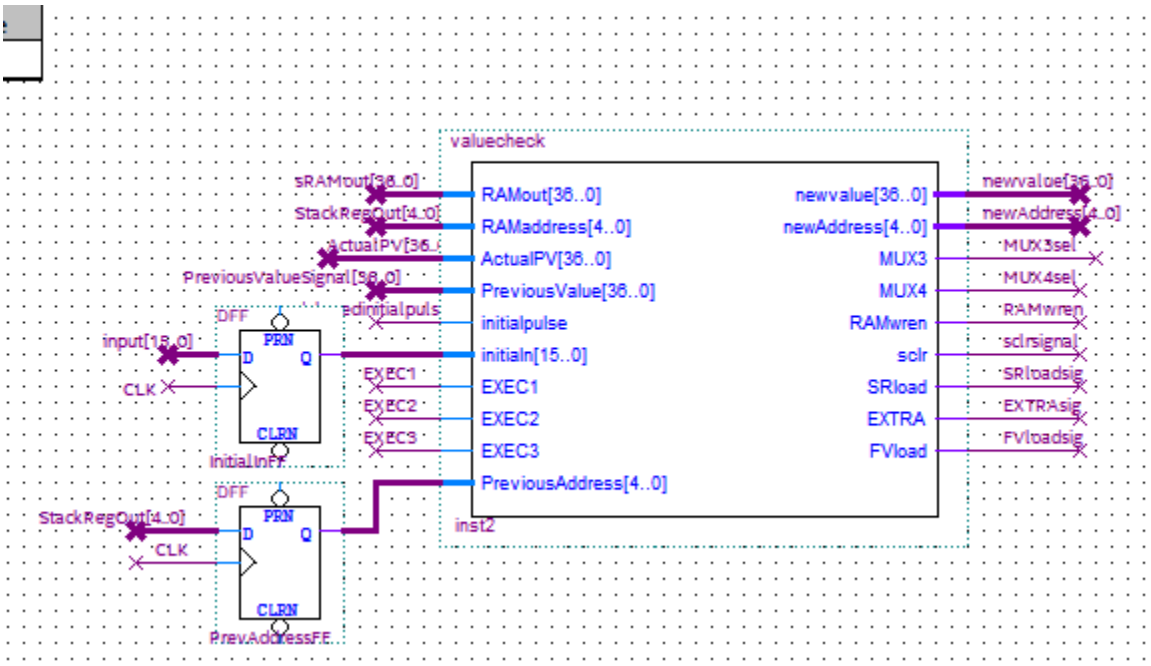
Note that this diagram excludes some registers used to acquire inputs into the valuecheck block.

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 works

The decoder block, which handles the signals around the circuit, called ‘valuecheck’ operates based on different if conditions. These if conditions are triggered depending on various conditions around the circuit; there are 8 if conditions in total: 1, 2, 3, 4, 5, 6, initialising and oneorzero.



Initialising the first value

This architecture is designed so that the operations only begin when the correct opcode is input which is detected by the initialstate block meaning the input data can vary whenever the instruction is not being called without affecting anything. When the correct opcode is detected, the initialstate block outputs a pulse indicating that a value is being loaded in and it increments the counter called ‘InitialCounter’. Once the counter has incremented once, it increments again and causes another pulse to be output from the initialstate block. The counter then increments again but no pulse is sent out at this point and the counter remains at this value of 3 until the final value has been calculated.

The 2-cycle pulse sent out by the initialstate block (called initialpulse) is input into the statemachine block which starts up a 2 or 3 cycle system (depending on the current conditions) used to calculate and write in the next value to be written into the stack.

This 2-cycle pulse is also input into the valuecheck block where it triggers the ‘initialising’ condition in the valuecheck verilog. When this condition is triggered:

- Cycle 1 – MUX3 is made to select 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 pushed into the stack, it must be extended to match the size of the stack using the inputextender block which fills the 21 LSBs with 0s.

Working towards the final value

Once the input variable has been input into the 0^{th} position of the stack, the valuecheck block identifies what set of signals it must output to the rest of the circuit and it calculates the new value that must be stored in the RAM. From the initial value, the resulting action is dependent on the input. For an input of 1 or 0, the valuecheck triggers the 'oneorzero' condition designed specifically for when the input parameter is 1 or 0 and the expected result is 1.

- Cycle 1 – Producing the output data value of 1 and writing 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. Also, the stack's 0^{th} address is overwritten with a null value so that the stack is ready for a new Fibonacci function call.

However, if the input parameter is anything other than a 1 or 0, condition '3' is triggered where fib(n-1) is called.

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

Note that for the following tables, the values are 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 less than or equal to 1, the condition '1' is met where the data value of the return address is incremented once.

- Cycle 1 – A value with the data value set to 1 is generated. The rest of the word excluding this data value is nullified by the isolatedata block which just sets all excess bits to 0. The stack pointer is also 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 data value of 1 from the isolatedata block with the ALU. Additionally, MUX4, the multiplexer controlling the input into the stack is made to take the input from the ALU

- Cycle 3 – This new value is written into the stack to overwrite the return address value; ultimately, the return address has just been incremented.

MUX7 is needed for these 3-cycle conditions so that the input value into the valuecheck block doesn't 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 which is the same value that was input for the previous 2 cycles. This ensures that 3-cycle conditions are fulfilled all the way through.

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 is then able to determine that a fib(n-1) function has just been completed and the fib(n-2) function has yet to be called. The condition '4' then performs the new function call.

- Cycle 1 – Generating the new value where the input variable n has been reduced by 2, the return address has been set to the previous address and the data variable is cleared to be 0. The stack pointer is also incremented.
- Cycle 2 – This new value 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	0
y = fib2	fib2	0001	2	0000	1
y = y + fib(0)	fib0	0002	0	0001	0

Condition '1' is met again here 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 will also know if a fib(n-2) function has just been completed and now the data variable of the original function must be returned to the return address. The set of instructions comes 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 this is added to the previous data value with the ALU. MUX4 is set to take the output from the ALU as the input 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 fib2 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

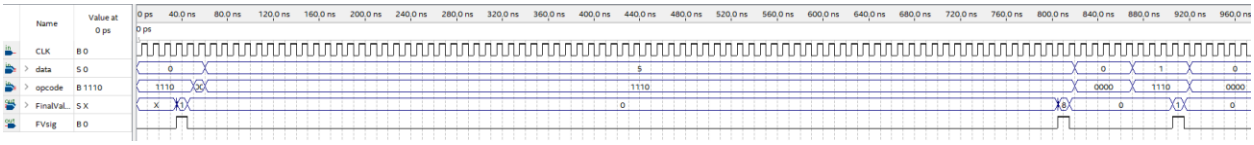
current function line	function calls	stack address	value of n	return address	data value
fib3	fib3	0000	3	(0000) by default	3

Ultimately, the final value is then stored in the 0^{th} address and the 16 relevant bits are stored in a separate register. The valuecheck block detects this through the condition titled '6'.

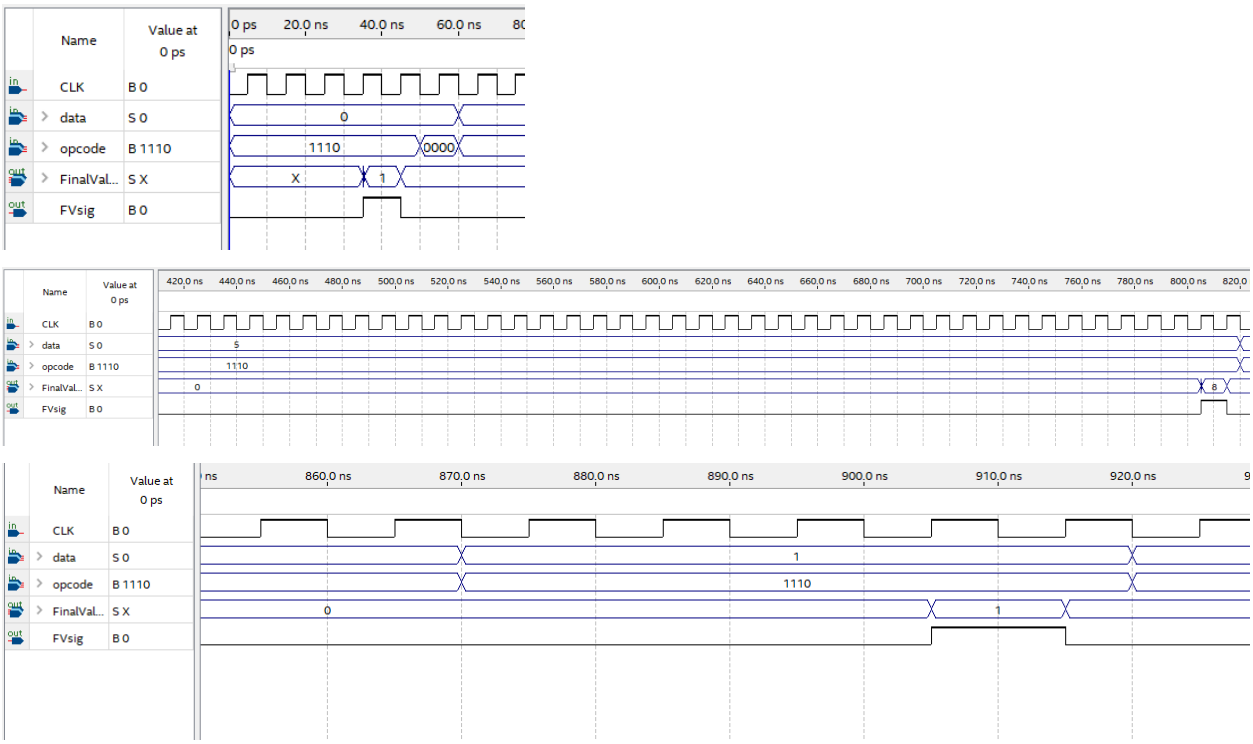
- Cycle 1 – A signal is sent to the sload input of the final value register which allows a value to be loaded in there; this same signal is output from the block to indicate to the rest of the CPU that the final value has been calculated. A null value is also generated so that it can overwrite the 0^{th} address to bring the block into a 'rest' state. Lastly, the counter from the beginning is reset to allow a new input instruction.
- Cycle 2 – This generated null value is written into the stack.

The rest state/condition '2' is used when nothing is happening with the fib block and there should not be any outputs or changes in the circuit. In both cycles, all signals are set to 0 so no values can be written in therefore allowing the circuit to be ready for a new input.

Evidence of functionality



The above waveform tests subsequent inputs of 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.



All the test results are correct and when they are calculated, the FVsig output pulses as desired.

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 during those signals. However, recursive functions that don't solely use addition will need to use more hardware.

Future improvement for the Fibonacci block

With additional time, it may be possible to implement a pipelining method where instead of waiting for the new value to be written into the stack before passing it into the valuecheck block again, we can bypass the stack using a multiplexer that takes the output of mux3 as the input to the valuecheck block. This means we wouldn't need an extra cycle to write in and read out the new value calculated at the end of each instruction. On the other hand, this could possibly create iteration errors when analysing the waveform in Quartus.

The valuecheck's conditions execute 2 or 3 cycle instructions depending on the condition triggered. However, some of these conditions could be completed in a single cycle such as condition '6' where the final value has been acquired and we just need to overwrite the 0^{th} address and the 'initialising' condition where the first value is written into the 0^{th} address. A solution to this would be to have the 2^{nd} cycle be triggered by its own 'extra' signal from the valuecheck block like how the 3^{rd} cycle is therefore allowing 1 cycle processes to occur.

Continuing with reducing the number of cycles, the current condition ‘1’ where the data value of the return address is incremented could be made redundant if increments of 1 are made immediately instead of having to call a new function for input variables of 0 and 1. In the conditions ‘3’ where the next fib(n-1) function is called or ‘4’ where fib(n-2) is called, separate if conditions within these could detect the value of n-1 or n-2 respectively and if they are going to return 1s to the data value. If they will, the increment could be made in possibly just 1 extra cycle instead of 3. This would warrant a large change in the verilog logic for other conditions though to accompany this change.

Concerning the quantity of logic cells, instead of using a separate ALU for the Fibonacci block, addition operations could be completed using a general CPU available to the rest of the CPU. This means an additional addition instruction could be added to the CPU without additional resources except from maybe a few multiplexers to determine the inputs.