Embedded Computer Architecture 2

Riccardo Battistig s1716239, Pietro Pennestrì s2382660

February 8, 2020

Contents

1 Stack Processor (Haskell, ghci)

In order to run the given program argument, all program instructions must be recognizable. Most importantly they must be distinguishable. In order to accomplish this the two data types `Opc` and `Instr` were defined. The function Core was constructed using pattern matching, based on distinguishable cases due to the program's data-types. Based on the pattern-matched argument given to the function, it will compute different results through the function `ALU`. The full code written for this part is reported in listing 1. Furthermore, the result of the simulation through the function `test` are depicted in Figure 1.

```haskell
module CPU_st where
import Data.List

data Opc = Add | Mul
deriving Show

data Instr = Push Int | Calc Opc
deriving Show

-- ---------------------** ALU definition **-----------------------
alu :: Opc -> Int -> Int -> Int
alu Add x y = x + y
alu Mul x y = x * y

-- ---------------------** Core Addition **-----------------------
core :: [Int] -> Instr -> [Int]
core stk (Calc Add) = newstk
where --deletes first 2 and replaces with output of alu
stkt = tail stk
snd = head stkt
frst = head stk
stkrem = delete snd (delete frst stk)
newstk = [alu Add frst snd] ++ stkrem

-- ---------------------** Core Multiplication **-----------------------
--core stk Mul x y = alout ++ stk
-- where -- just appends alu output to top of stack
-- alout = [alu Mul x y]

core stk (Calc Mul) = newstk
where --deletes first 2 and replaces with output of alu
stkt = tail stk
snd = head stkt
frst = head stk
```

```haskell
stkrem = delete snd (delete frst stk)
newstk = [alu Mul frst snd] ++ stkrem


-- ---------------------** Core push **-----------------------
core stk (Push n) = [n] ++ stk

-- ---------------------** Simulation question 1 **-----------------------
program = [ Push 2 , Push 10 , Calc Mul , Push 3 , Push 4 , Push 11 , Calc Add , Calc Mul , Calc Add , Push 12 , Push 5 , Calc
     Add , Calc Mul ]

test = scanl core [] program
```

Listing 1: Haskell Code of Stack Processor

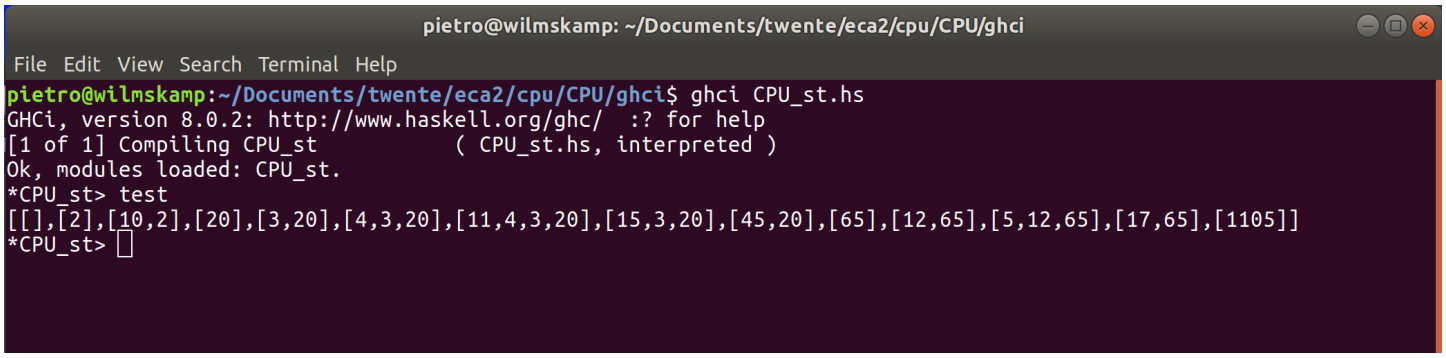

Figure 1: Simulation Results of function Test from listing 1

2 Heap and Stack Processor (Haskell, ghci)

In continuation of the last assignment, a heap was added to the system, resulting in a system with both an argument and output tuple of heap and stack. The heap is added to add an extra layer of complexity, as to mimic local memory. In order to handle this improved functionality the data type `Value` is added, which either is a constant, or it should refer to a value at a specific address in the heap. A function value was written in order to solve this ambiguity, as it takes as argument a type Value and returns the corresponding integer of either a constant or the value stored at a specific address in the heap. The full code is reported in Listing 2, while the results are depicted in Figure 2.

```haskell
module CPU_HpSt where
import Data.List

data Opc = Add | Mul
deriving Show

data Value = Const Int | Addr Int
deriving Show

data Instr = Push Value | Calc Opc
deriving Show

alu :: Opc -> Int -> Int -> Int
alu Add x y = x + y
alu Mul x y = x * y


value :: Value -> [Int]-> Int
value (Addr x) heap = heap !! x
value (Const x) _ = x


core :: ([Int] , [Int]) -> Instr -> ([Int] , [Int])
core ((stkFirst:stkSecond:stkTail) , heap) (Calc Add) = (stk' , heap)
where
operator1 = stkFirst
operator2 = stkSecond
stk' = [alu Add operator1 operator2] ++ stkTail
```

```haskell

core ((stkFirst:stkSecond:stkTail) , heap) (Calc Mul) = (stk' , heap)
where
operator1 = stkFirst
operator2 = stkSecond
stk' = [alu Mul operator1 operator2] ++ stkTail


core (stk , heap) (Push (Const x)) = (stk' , heap)
where
stk' = [x] ++ stk


core (stk , heap) (Push (Addr x)) = (stk' , heap)
where
stk' = [(heap !! x)] ++ stk



program = [ Push (Const 2 ) , Push (Addr 0 ) , Calc Mul , Push (Const 3 ) , Push (Const 4 ) , Push (Addr 1 ) , Calc Add, Calc
     Mul , Calc Add,Push (Const 12) , Push (Const 5 ) , Calc Add, Calc Mul ]

test = scanl core ([],[10,11]) program
```

Listing 2: Haskell Code of Heap and Stack Processor

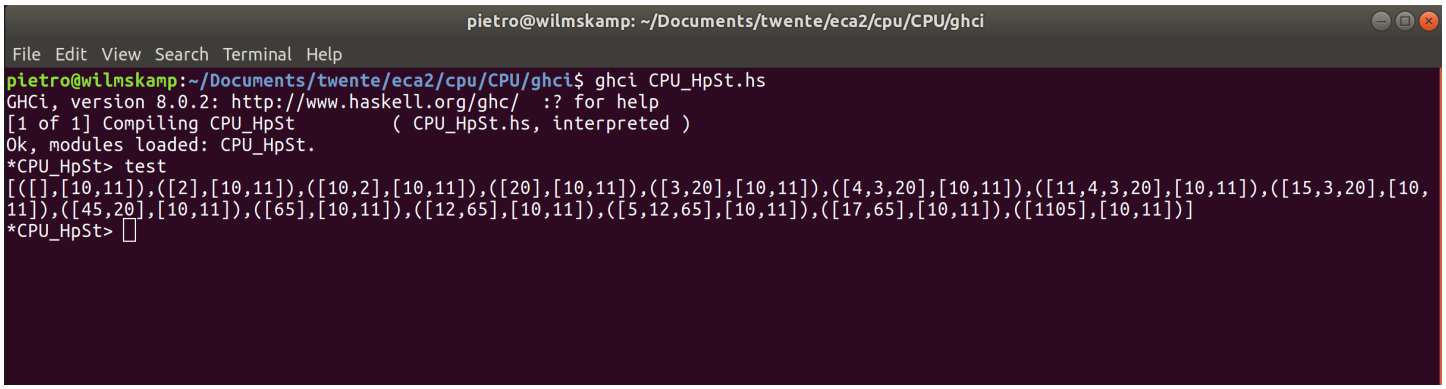

Figure 2: Simulation Results of function Test from listing 2

3 Heap, Stack and Program Counter (Haskell, ghci)

Until now the instructions are given as an argument, however it would be preferable to define the program only once, and have a program counter to navigate the program. In order to implement the program counter efficiently, it was sought to not change the argument definition of the previous 'core' function in order to maintain readability. To do this the function name was changed to execute, which maintains all the behaviors of core (back from listing 2). A new core function was written which reads and increments the program counter, as well as extracting the instruction from the program, and then passes that instruction to the function execute. The full code is reported in listing 3. The result od the simulation is depicted in Figure 3.

```haskell
module CPU_HpStPc where
import Data.List

data Opc = Add | Mul
deriving Show

data Value = Const Int | Addr Int | Top
deriving Show

data Instr = Push Value | Calc Opc | Send Value
deriving Show
program:: [Instr]
program = [ Push (Const 2 ) , Push (Addr 0 ) , Calc Mul , Push (Const 3 ) , Push (Const 4 ) , Push (Addr 1 ) , Calc Add, Calc
     Mul , Calc Add,Push (Const 12) , Push (Const 5 ) , Calc Add, Calc Mul ]

```

```
program2:: [Instr]
program2 = [ Push (Const 2 ) , Push (Addr 0 ) , Calc Mul , Push (Const 3 ) , Push (Const 4 ) , Push (Addr 1 ) , Calc Add, Calc
     Mul , Calc Add,Push (Const 12) , Push (Const 5 ) , Calc Add, Calc Mul, Send Top ]


-- ---------------------** ALU definition **-----------------------
alu :: Opc -> Int -> Int -> Int
alu Add x y = x + y
alu Mul x y = x * y

value :: Value -> [Int]-> Int
value (Addr x) heap = heap !! x
value (Const x) _ = x

execute :: ([Int] , [Int]) -> Instr -> (([Int] , [Int]), Int)
execute ((stkFirst:stkSecond:stkTail) , heap) (Calc Add) = ((stk' , heap),-1)
where
operator1 = stkFirst
operator2 = stkSecond
stk' = [alu Add operator1 operator2] ++ stkTail

execute ((stkFirst:stkSecond:stkTail) , heap) (Calc Mul) = ((stk' , heap),-1)
where
operator1 = stkFirst
operator2 = stkSecond
stk' = [alu Mul operator1 operator2] ++ stkTail

execute (stk , heap) (Push (Const x)) = ((stk' , heap), -1)
where
stk' = [x] ++ stk

execute (stk , heap) (Push (Addr x)) = ((stk' , heap), -1)
where
stk' = [(heap !! x)] ++ stk

execute (stk , heap) (Send (Const x)) = ((stk , heap), x)

execute (stk , heap) (Send (Addr x)) = ((stk , heap), (heap !! x))

execute (stk , heap) (Send Top) = ((stk , heap), (stk !! 0))

-- Simulation Function
sim f s [] = []
sim f s (x:xs) = z:sim f s' xs
where
(s',z) = f s x
test = sim execute ([],[11,10]) program


-- execute function with PC
core :: [Instr] -> (Int , [Int] , [Int]) -> Int -> ((Int, [Int], [Int]), Int)
core prog (pc, stk, heap) tick = ((pc', stk', heap'), out)
where
pc' = tick + pc
((stk' , heap'), out) = execute (stk, heap) (prog !! pc)

test = sim (core program2) (0,[],[10, 11]) $ repeat 1
```

Listing 3: Haskell Code of Heap, Stack and Program Counter Processor

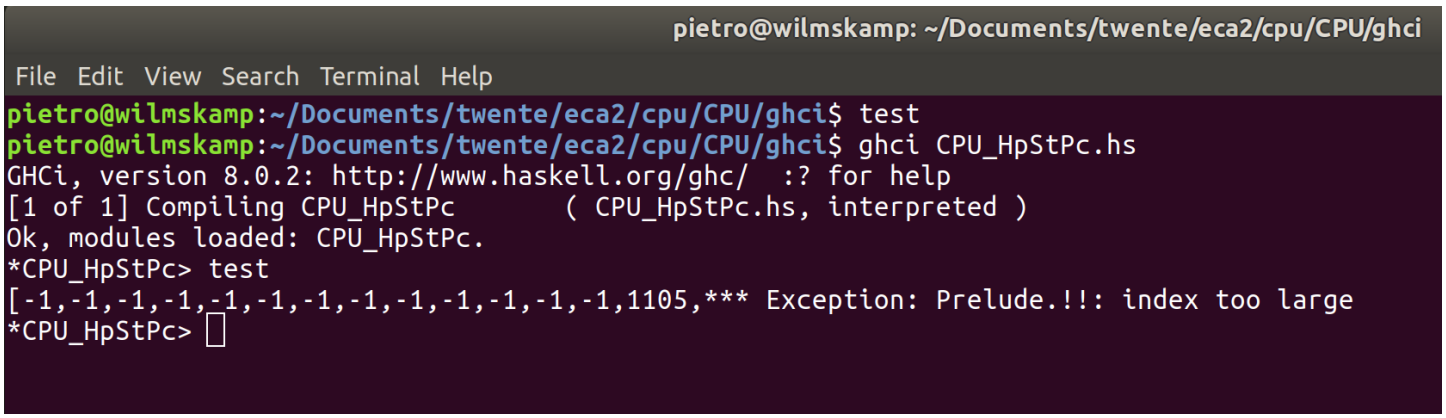

Figure 3: Simulation Results of function Test from listing 3

4 Heap, Stack, Program Counter, and Single Stack Access (Haskell, GHCI)

The last addition to the system is to add single stack access, meaning that instead of accessing two values, the processor treats one value at a time. To facilitate this behavior an extra register is added. This register is seen as being separate from the state, meaning the existing function argument tuples will have to be changed into: (`stack`, `heap`, `register`). The entire code can be found in listing 4. The execute function is then changed to being single access by only passing one value to the `ALU`, and the `ALU` taking its other argument from the previously mentioned register. A value is added to the register by means of the added instruction Pop, which removes the top value of the stack and places it into the register. The resulting code was simulated using the given test function. The result of `test` function are depicted in Figure 4.

```
1  module CPU_HpStPcReg where
2  import Data.List
3
4  data Opc = Add | Mul
5  deriving Show
6
7  data Value = Const Int | Addr Int | Top
8  deriving Show
9
10 data Instr = Push Value | Calc Opc | Send Value | Pop
11 deriving Show
12 program:: [Instr]
13 program = [ Push ( Const 2 ) , Push ( Addr 0 ) , Pop , Calc Mul , Push ( Const 3 ) , Push ( Const 4 ) , Push ( Addr 1 ) , Pop ,
       Calc Add , Pop , Calc Mul , Pop , Calc Add , Push ( Const 12 ) , Push ( Const 5 ) , Pop , Calc Add , Pop , Calc Mul ,
      Send Top , Push ( Const 2 ) , Send Top , Pop , Send Top ]
14 -- ---------------------** ALU definition **-----------------------
15 alu :: Opc -> Int -> Int -> Int
16 alu Add x y = x + y
17 alu Mul x y = x * y
18
19 value :: Value -> [Int]-> Int
20 value (Addr x) heap = heap !! x
21 value (Const x) _ = x
22
23 execute :: ([Int] , [Int] , Int) -> Instr -> (([Int] , [Int], Int), Int)
24 execute ((stkFirst:stkTail) , heap, register) (Calc Add) = ((stk' , heap, register),-1)
25 where
26 operator1 = stkFirst
27 stk' = [alu Add operator1 register] ++ stkTail
28
29
30 execute ((stkFirst:stkTail) , heap, register) (Calc Mul) = ((stk' , heap, register),-1)
31 where
32 operator1 = stkFirst
33 stk' = [alu Mul operator1 register] ++ stkTail
34
35
36 execute ((stkFirst:stkTail) , heap, register) (Pop) = ((stkTail , heap, stkFirst),-1)
37
38
39 execute (stk, heap, register) (Push (Const x)) = ((stk' , heap, register), -1)
40 where
41 stk' = [x] ++ stk
```

```
42
43
44 execute (stk, heap, register) (Push (Addr x)) = ((stk' , heap, register), -1)
45 where
46 stk' = [(heap !! x)] ++ stk
47
48 execute (stk , heap, register) (Send (Const x)) = ((stk , heap, register), x)
49 execute (stk , heap, register) (Send (Addr x)) = ((stk , heap, register), (heap !! x))
50 execute (stk , heap, register) (Send Top) = ((stk , heap, register), (stk !! 0))
51
52
53 -- Simulation Function
54 sim f s [] = []
55 sim f s (x:xs) = z:sim f s' xs
56 where
57 (s',z) = f s x
58
59
60
61 -- execute function with PC
62 core :: [Instr] -> (Int , [Int] , [Int], Int) -> Int -> ((Int, [Int], [Int], Int), Int)
63 core prog (pc, stk, heap, register) tick = ((pc', stk', heap', register'), out)
64 where
65 pc' = tick + pc
66 ((stk' , heap', register'),out) = execute (stk, heap, register) (prog !! pc)
67
68 test = sim (core program) (0,[],[10, 11],0) $ repeat 1
```

Listing 4: Haskell Code of Heap, Stack, Program Counter and Single Stack Access Processor

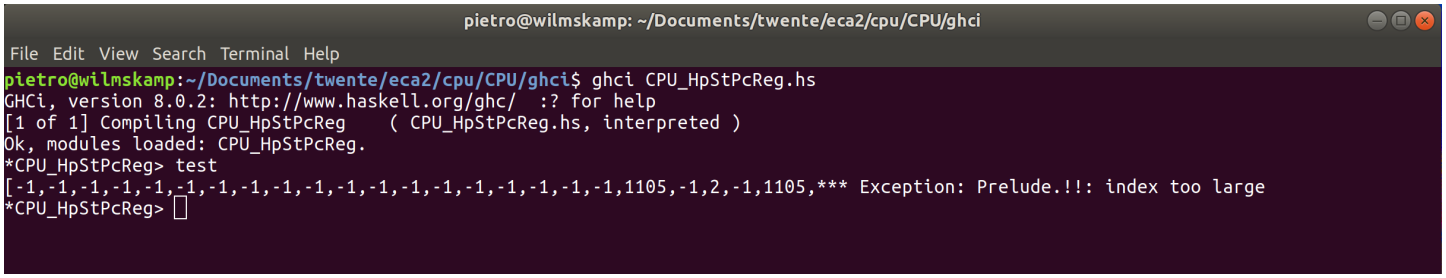

Figure 4: Simulation Results of function Test from listing 4

5 CPU Fixed

Two different implementation of `core` have been implemented: `core` and `core2`. These implementation can be found in listing 5 as function `core` starting at line 75, in the same listing as function `core2` starting at line 104. The differences in the two implementations are listed below:

- The function `core2` is not able to run any other combination of instructions besides the specific order in program. This is a result of currying the argument program into the function call in the top entity.
- The function `core` makes use of an enlarged instruction set, which allows the user to reset or stop the program counter. These tasks are respectively implemented by means of instruction `Idle` and `ResetPc`. The user has also the possibility to upload a new program inside the `heap`.

The difference between the two PC's can be seen in Figures 5 and 6. Additionally, a difference in `core` and `core2` are that `core2` stores the entire program in memory, introducing many registers and selectors. The increased complexity of managing the program instructions in `core`, leads to an increase in complexity of the `RTL` schematic in comparison with `core2`. Both designs however work perfectly and produce the same result as shown in figure 7. Overall, it is also reaffirmed in the flow summaries in figures 8 and 9 that all the resources are quite similar. The only differences being in the data flow paths and memory amount (ALM's as well as pins and registers), which is higher in `core` as it has more complexity in the program and PC changes compared to `core2`.

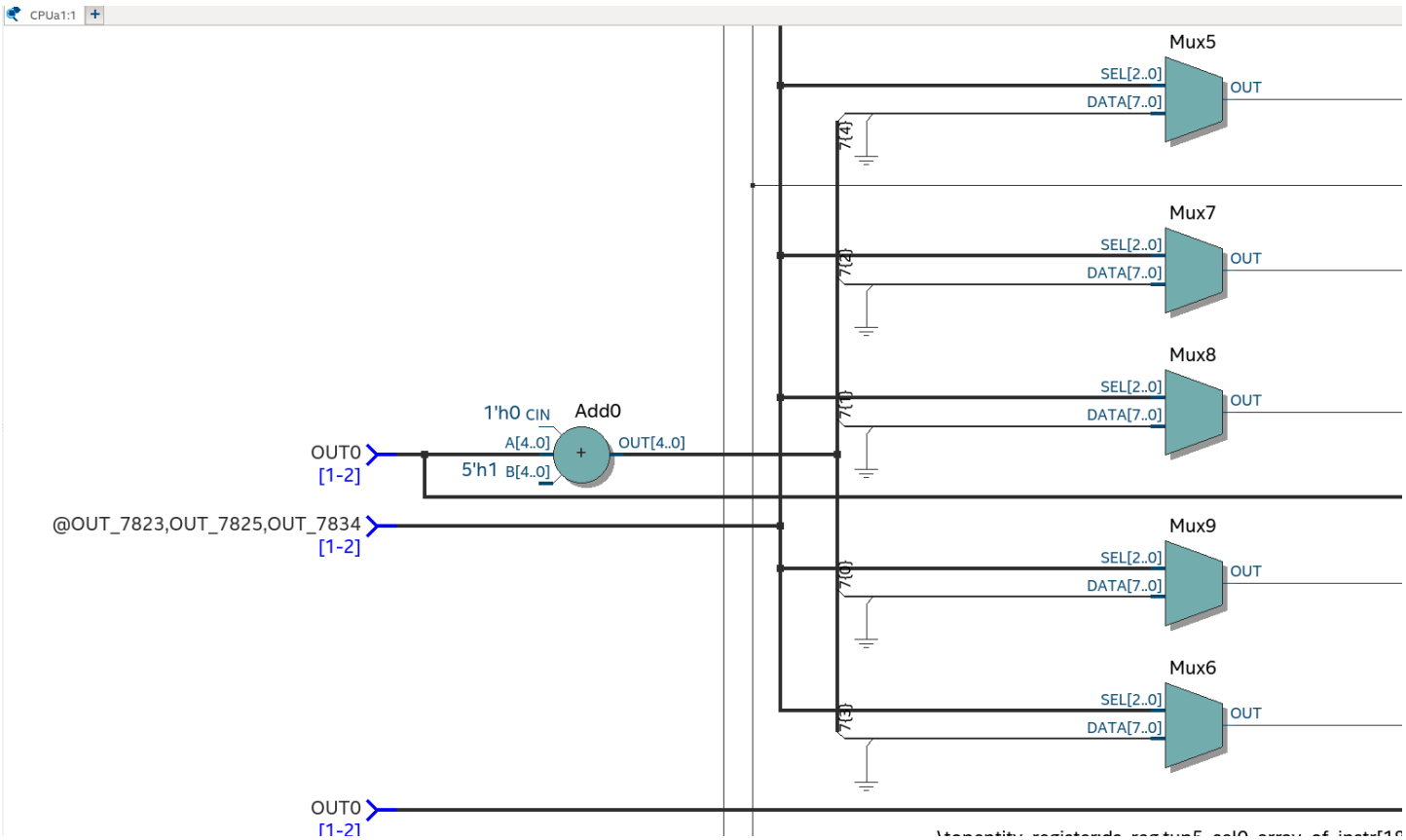

Figure 5: Focused Snapshot of Program counter in produced RTL schematic for the core function 5

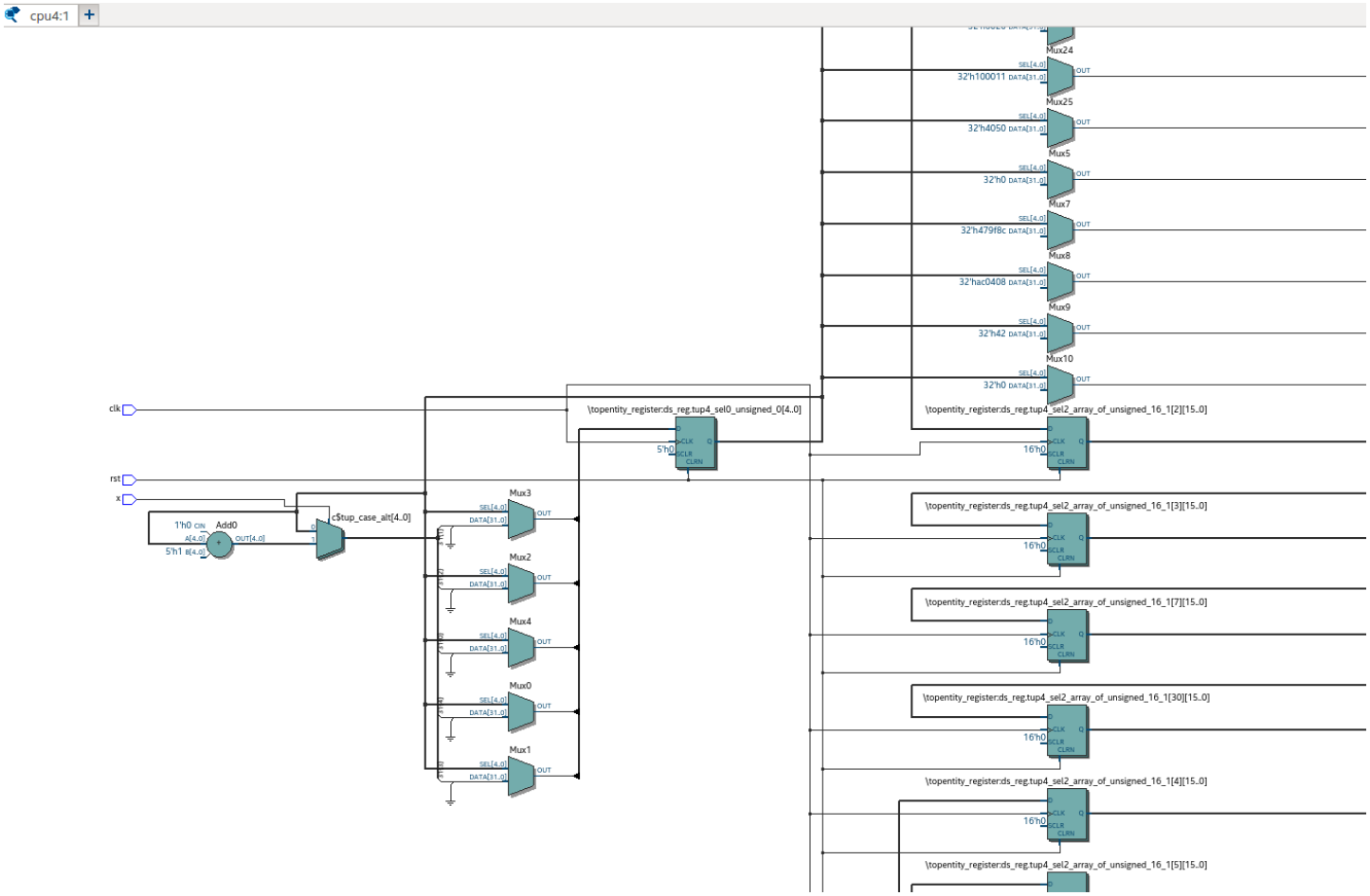

Figure 6: Focused Snapshot of Program counter in produced RTL schematic for the core function 5

```
Clashi, version 1.0.1 (using clash-lib, version 1.0.1):
http://www.clash-lang.org/  :? for help
Ok, one module loaded.
Clash.Prelude CPU_Fixed> simulate @System coreSim testList
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothi
ng,Nothing,Nothing,Nothing,Nothing,Nothing,Just 1105,Nothing,Just 2,Nothing,Just 1105,Nothing,Nothing,Nothing,
Nothing,Nothing,Nothing,Nothing,Nothing,*** Exception: X: finite list
CallStack (from HasCallStack):
  errorX, called at src/Clash/Signal/Internal.hs:1268:57 in clash-prelude-1.0.1-BorHjli5pMX3STUvZInnYc:Clash.S
ignal.Internal
Clash.Prelude CPU_Fixed>
```

Figure 7: Result of Simulation from both Instantiated Clash topEntities.

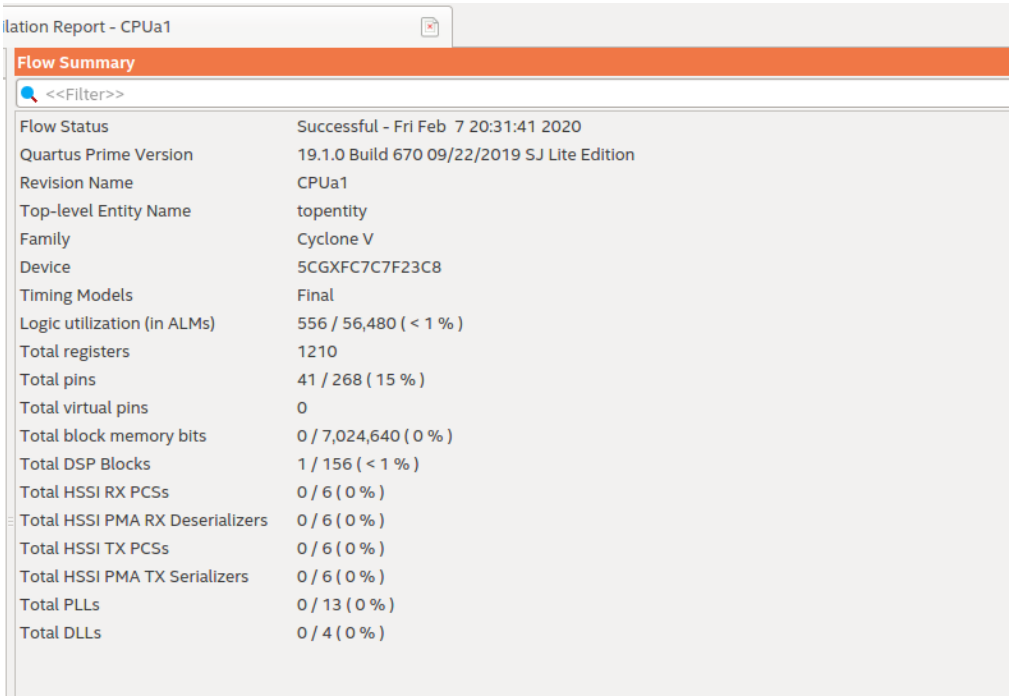

Figure 8: Flow summary from function core

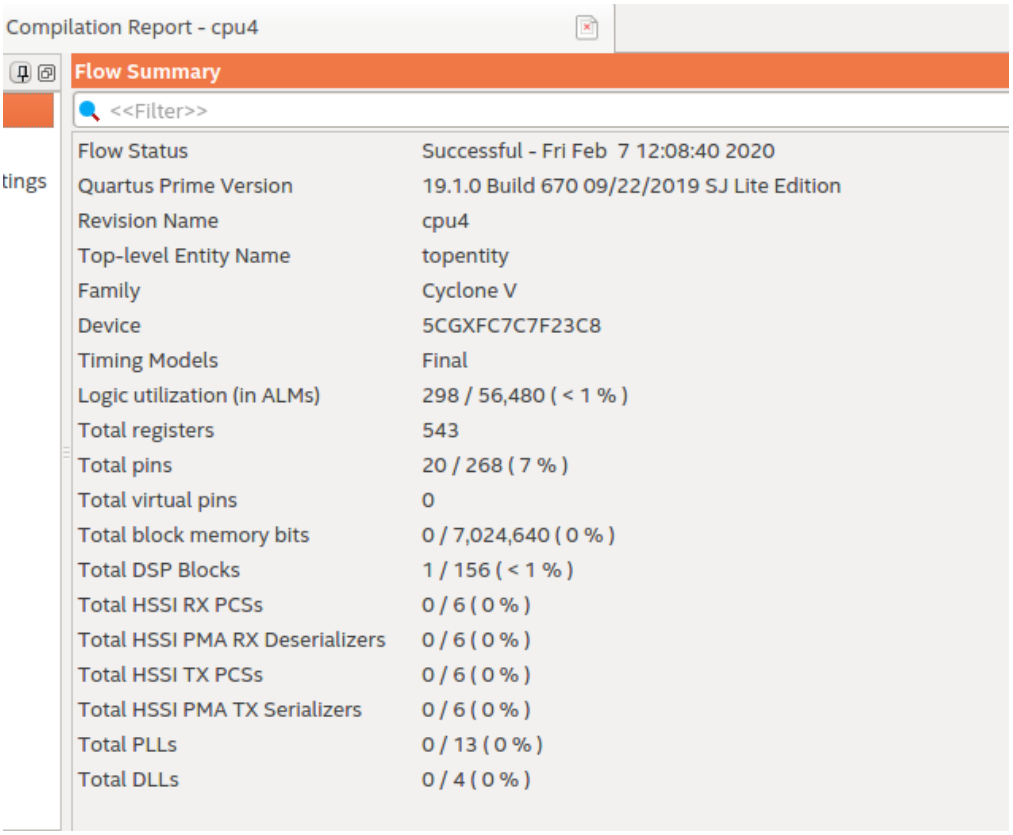

Figure 9: Flow summary from function core2

The implementation of the `CPU` in `clash` was made by turning all lists into vectors. Additionally commands were appropriately changed to suit this transformation. As the program is internal to the function (as is result of currying), the system will create an internal memory to store this program. This can effectively be seen as a sort of read-only memory (`ROM`) which appropriately selects the right memory elements. It is expected due to lines 84 and 111 (in listing 5) that the selection of this program will be made based on an incrementing program counter. This Program counter can be explicitly seen in the figure 6. The `ROM`, which in `VHDL` is seen as simply registers will lead into the execute function, in turn, the execute function

will return values as well, which will be stored in different registers. This is in line with an expected read/write behavior of a CPU, such that we could actually interpret the memory behavior connected to execute function as random access memory (`RAM`). The execute function was isolated in the `RTL` schematic produced in Quartus by using the command `NOINLINE`, and the connections can be seen in Figure 10. Similarly the execute function contains the `ALU` function (as seen in figure 11). The execute function also contains many selectors as it needs to redirect for certain instructions to different memory locations, or load the `ALU`. The full code is reported in listing 5.

```
module CPU_Fixed where
import Clash.Prelude

type Clk = Clock System
type Rst = Reset System
type Sig = Signal System

type Val = Unsigned 16
type Vector = Vec 32 Val

data Opc = Add | Mul
deriving (Show, Generic, NFDataX, Eq)

data Value = Cst Val | Addr Val | Top
deriving (Show, Generic, NFDataX, Eq)

data Instr = Push Value | Calc Opc | Send Value | Pop | Idle | ResetPc
deriving (Show, Generic, NFDataX, Eq)

type ProgMemT = Vec 32 Instr


-- We have considered a 32 instruction memory, because 32 is the nearest power of 2 from 24.
initProgMem :: ProgMemT
initProgMem = (Push ( Cst 2 ) :> Push ( Addr 0 ) :> Pop :> Calc Mul :> Push ( Cst 3 ) :> Push ( Cst 4 ) :> Push ( Addr 1 ) :>
    Pop :> Calc Add :> Pop :> Calc Mul :> Pop :> Calc Add :> Push ( Cst 12 ) :> Push ( Cst 5 ) :> Pop :> Calc Add :> Pop :>
    Calc Mul :> Send Top :> Push ( Cst 2 ) :> Send Top :> Pop :> Send Top:> Idle:>Idle:>Idle:>Idle:>Idle:>Idle:>Idle:>ResetPc
    :> Nil)

-- initProgMem1 contains the original program provided by the assignment
initProgMem1:: Vec 24 Instr
initProgMem1 = (Push ( Cst 2 ) :> Push ( Addr 0 ) :> Pop :> Calc Mul :> Push ( Cst 3 ) :> Push ( Cst 4 ) :> Push ( Addr 1 ) :>
    Pop :> Calc Add :> Pop :> Calc Mul :> Pop :> Calc Add :> Push ( Cst 12 ) :> Push ( Cst 5 ) :> Pop :> Calc Add :> Pop :>
    Calc Mul :> Send Top :> Push ( Cst 2 ) :> Send Top :> Pop :> Send Top :> Nil)

-- initstack contatins the initial stack memory
initstack::Vector
initstack = replicate d32 0

-- initiheap contaons the initial heap
initheap:: Vector
initheap = (10:>Nil) ++ ( (11:>Nil) ++ (replicate d30 0) )

alu :: Opc -> Val -> Val -> Val
alu Add x y = x + y
alu Mul x y = x * y

execute :: (Vector , Vector , Val) -> Instr -> ((Vector , Vector, Val), Maybe(Val))
execute (stk , heap, register) (Calc Add) = ((stk' , heap, register),Nothing)
where
operator1 = head stk
stk' = (alu Add operator1 register) +>> (stk <<+ 0)


execute (stk , heap, register) (Calc Mul) = ((stk' , heap, register),Nothing)
where
operator1 = head stk
stk' = (alu Mul operator1 register) +>> (stk <<+ 0)


execute (stk , heap, register) (Pop) = (( ( (tail stk) ++ (0:>Nil) ) , heap, (head stk)),Nothing)


execute (stk, heap, register) (Push (Cst x)) = ((stk' , heap, register), Nothing)
where
stk' = x +>> stk
```

```
62
63
64 execute (stk, heap, register) (Push (Addr x)) = ((stk' , heap, register), Nothing)
65 where
66 stk' = (heap !! x) +>> stk
67
68
69 execute (stk , heap, register) (Send (Cst x)) = ((stk , heap, register), Just (x) )
70 execute (stk , heap, register) (Send (Addr x)) = ((stk , heap, register), Just (heap !! x))
71 execute (stk , heap, register) (Send Top) = ((stk , heap, register), Just (stk !! 0))
72 execute (stk, heap, register) (Idle) = ((stk , heap, register), Nothing)
73
74
75 core ::  (ProgMemT, Unsigned 5 ,Vector , Vector , Val) -> (Bool, Instr)-> ((ProgMemT, Unsigned 5 ,Vector, Vector, Val), Maybe(
       Val) )
76 core (programMem, pc ,stk, heap, register) (is_write, instr) = ((programMem', pc' ,stk', heap', register'), out )
77 where
78 current_instr = programMem !! pc
79 is_idle = current_instr == Idle
80 is_reset = current_instr == ResetPc
81 pc'
82 |(is_idle || is_write) = pc
83 |is_reset = 0
84 |otherwise = pc + 1
85
86 programMem'
87 |is_write = instr +>> programMem
88 |otherwise = programMem
89 ((stk' , heap', register'), out) = execute (stk, heap, register) (current_instr)
90
91
92 coreSim :: HiddenClockResetEnable dom  => Signal dom ((Bool, Instr)) -> Signal dom (Maybe(Val))
93 coreSim = mealy core (initProgMem, 0, initstack, initheap, 0)
94 testList = toList $ replicate d32 ((False, Idle))
95 -- simulate @System coreSim testList
96
97 {--
98 topEntity :: Clk -> Rst -> Sig (Bool, Instr) -> Sig (Maybe(Val))
99 topEntity clk rst x = withClockResetEnable clk rst enableGen (mealy core (initProgMem, 0, initstack, initheap, 0) ) x
100
101 --}
102
103 -- Alternative implementation of core function:
104 core2 :: Vec 24 Instr ->  (Unsigned 5 ,Vector , Vector , Val) -> (Bool)-> ((Unsigned 5 ,Vector, Vector, Val), Maybe(Val) )
105 core2 programMem (pc ,stk, heap, register) (is_execute) = ((pc' ,stk', heap', register'), out )
106 where
107 current_instr = programMem !! pc
108 is_out_of_bound = pc == 23
109 pc'
110 |is_out_of_bound = 0
111 |is_execute = pc + 1
112 |otherwise = pc
113 --pc' = pc + 1
114 ((stk' , heap', register'), out) = execute (stk, heap, register) (current_instr)
115
116 coreSim2 :: HiddenClockResetEnable dom  => Signal dom (Bool) -> Signal dom (Maybe(Val))
117 coreSim2 = mealy (core2 initProgMem1) (0, initstack, initheap, 0)
118 -- simulate @System coreSim2 testList2
119 testList2 = toList $ replicate d48 (True)
120
121 topEntity :: Clk -> Rst -> Sig (Bool) -> Sig (Maybe(Val))
122 topEntity clk rst x = withClockResetEnable clk rst enableGen ( mealy (core2 initProgMem1) (0, initstack, initheap, 0) ) x
123 {-# NOINLINE alu #-}
124 {-# NOINLINE execute #-}
```

Listing 5: Clash Implementation of Heap, Stack, Program Counter and Stack Access Processor

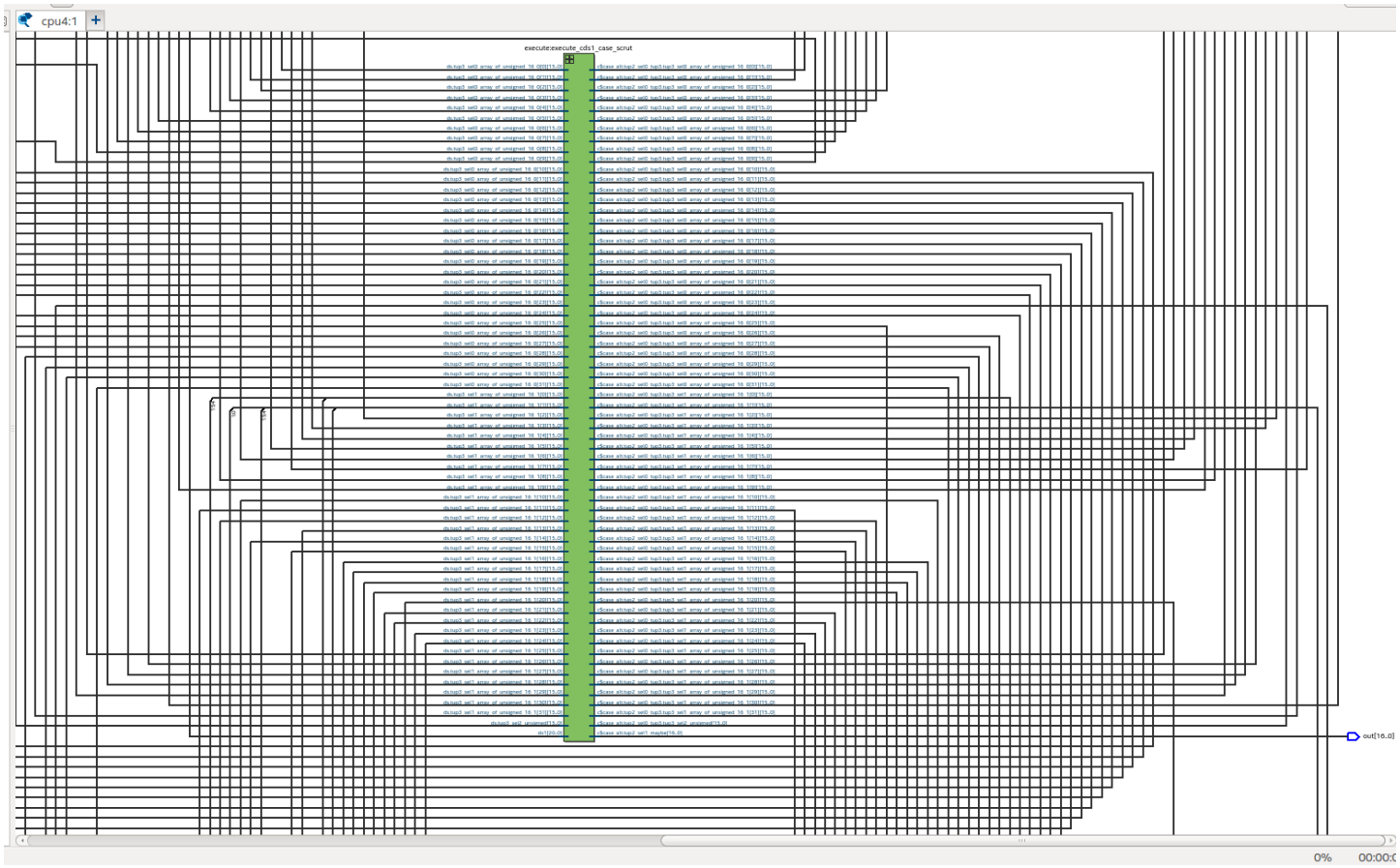

Figure 10: Snapshot of RTL Schematic which Shows the Execute Function Being Fed by the Memory Edges

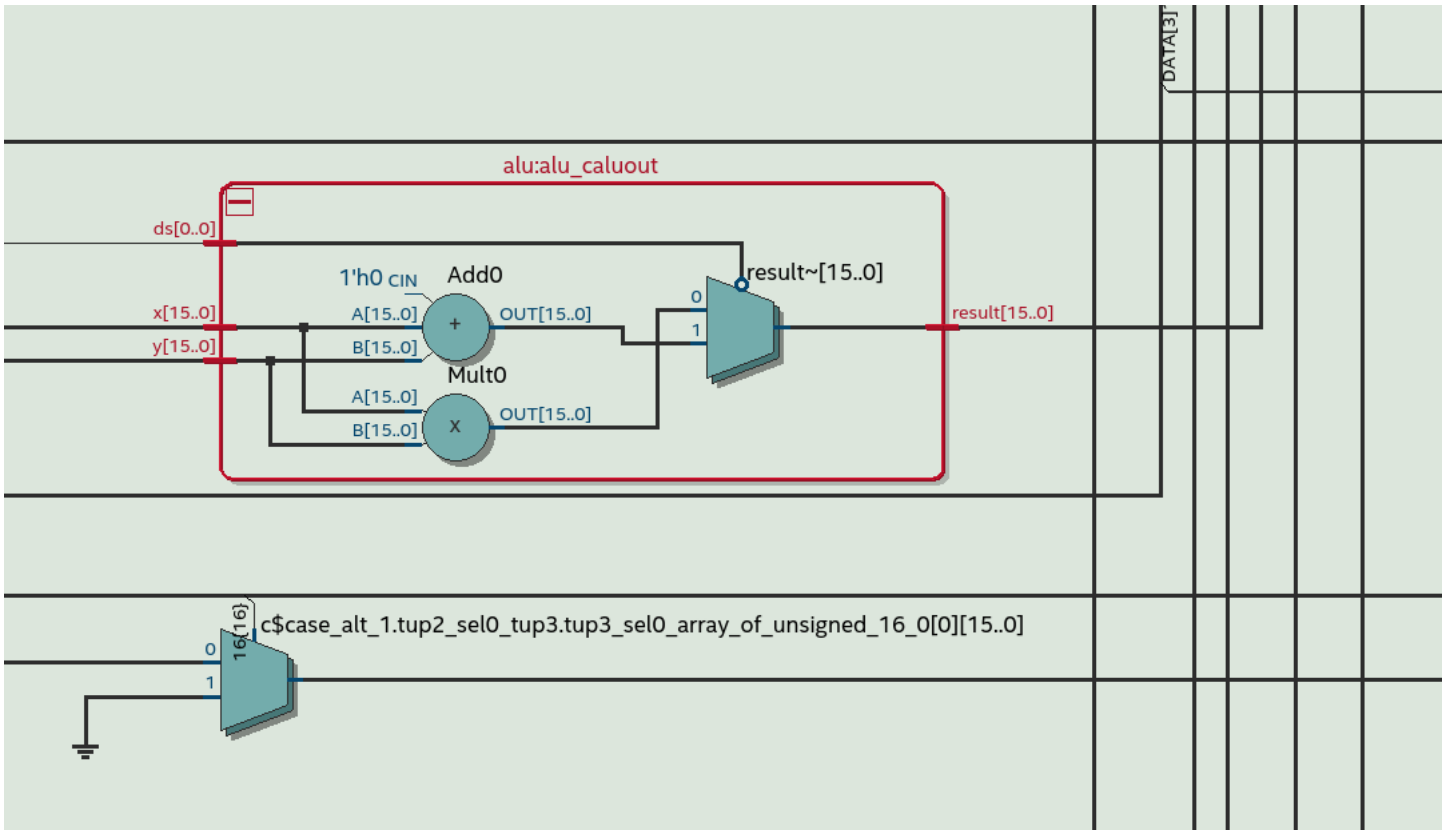

Figure 11: Snapshot of RTL Schematic which Shows the `ALU` Function