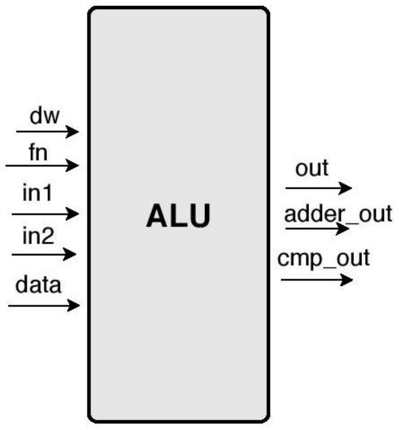
Arithmetic Logic Unit (ALU).

Block Diagram:



Where

dw=datawidth which can be 32 or 64 bit
fn=ALUop bit as input
in1=Operand B as input,32 or 64 bit
in2=Operand A as input,32 or 64 bit
out=ALU result output for logical operators
adder_out=ALU result output for mathematical operations
cmp_out=Branch result output

DEEP DIVE INTO CODE

Explanation (By Means of Flowcharts) :

```
object ALU
{
  val SZ_ALU_FN = 4
  def FN_X     = BitPat("b????")
  def FN_ADD   = UInt(0)
  def FN_SL    = UInt(1)
  def FN_SEQ   = UInt(2)
  def FN_SNE   = UInt(3)
  def FN_XOR   = UInt(4)
  def FN_SR    = UInt(5)
  def FN_OR    = UInt(6)
  def FN_AND   = UInt(7)
  def FN_SUB   = UInt(10)
  def FN_SRA   = UInt(11)
```

explanation

Object of ALU is created.

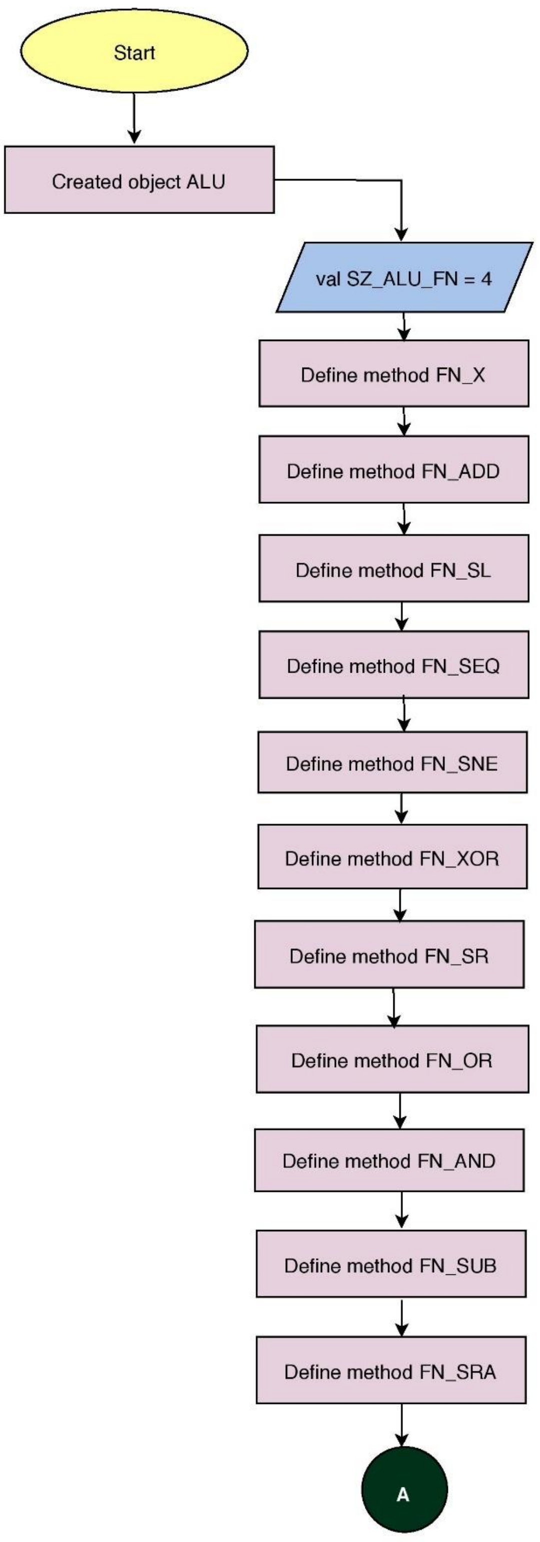
This object have different methods of different operation ALU has to perform

Define SZ_ALU_FN as 4 which means 2^4=16 possible ways in which it performs operation on ALU.This variable is defining Aluop bits which in this case is 4.Aluop bits can be changed if more operations in ALU is added

Define FN_X which uses BitPat command in order to enable representation of don't cares.

Define functions of different operation which returns different unsigned integer values. Uses these unsigned integer values later in this code in order to compare Mux conditions

Functions	Operation function performs
FN_ADD	This function will perform Add operation
FN_SL	This function will perform Shift left operation
FN_SEQ	This function will perform Set equals to operation
FN_SNE	This function will perform Set not equals to operation
FN_XOR	This function will perform XOR operation
FN_SR	This function will perform shift right operation
FN_OR	This function will perform OR operation
FN_AND	This function will perform AND operation
FN_SUB	This function will perform subtraction operation
FN_SRA	This function will perform shift right arithmetic operation
FN_SRA	This function will perform shift right arithmetic operation


Start
Created object ALU
val SZ_ALU_FN = 4
Define method FN_X
Define method FN_ADD
Define method FN_SL
Define method FN_SEQ
Define method FN_SNE
Define method FN_XOR
Define method FN_SR
Define method FN_OR
Define method FN_AND
Define method FN_SUB
Define method FN_SRA
A

```
def FN_SLT  = UInt(12)
def FN_SGE  = UInt(13)
def FN_SLTU = UInt(14)
def FN_SGEU = UInt(15)
def FN_DIV  = FN_XOR
def FN_DIVU = FN_SR
def FN_REM  = FN_OR
def FN_REMU = FN_AND
def FN_MUL    = FN_ADD
def FN_MULH   = FN_SL
def FN_MULHSU = FN_SEQ
def FN_MULHU  = FN_SNE
def isMulFN(fn: UInt, cmp: UInt) = fn(1,0) === cmp(1,0)
def isSub(cmd: UInt) = cmd(3)
```

explanation

Define methods of M extension version of RISCV and these methods includes different methods of Iformat instruction of RSICV

We define these methods which is later used in Multiplier module

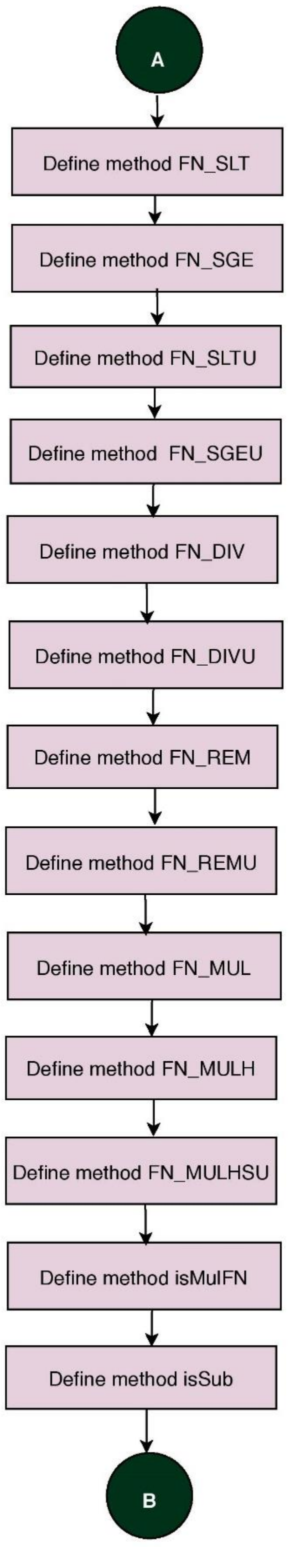
Functions	Operation function performs
FN_SLT	This function will perform set less than operation
FN_SGE	This function will perform set greater than operation
FN_SLTU	This function will perform Set less than unsigned operation
FN_SGEU	This function will perform Set greater than equals to unsigned operation
FN_DIV	This function will perform division operation
FN_DIVU	This function will perform division unsigned operation
FN_REM	This function will perform remainder operation
FN_REMU	This function will perform remainder unsigned operation
FN_MUL	This function will perform multiplication operation
FN_MULH	This function will perform multiplication higher bits operation

FN_MULHSU	This function will perform multiplication higher singed bits unsigned operation
FN_MULHU	This function will perform multiplication higher bits unsigned operation

Define different methods to perform ALU operations

isSub method used to perform subtraction operation

isMULFN takes two parameter fn and cmp as input where fn is ALUop bits and cmp is compare bits.


A
Define method FN_SLT
Define method FN_SGE
Define method FN_SLTU
Define method FN_SGEU
Define method FN_DIV
Define method FN_DIVU
Define method FN_REM
Define method FN_REMU
Define method FN_MUL
Define method FN_MULH
Define method FN_MULHSU
Define method isMulFN
Define method isSub
B

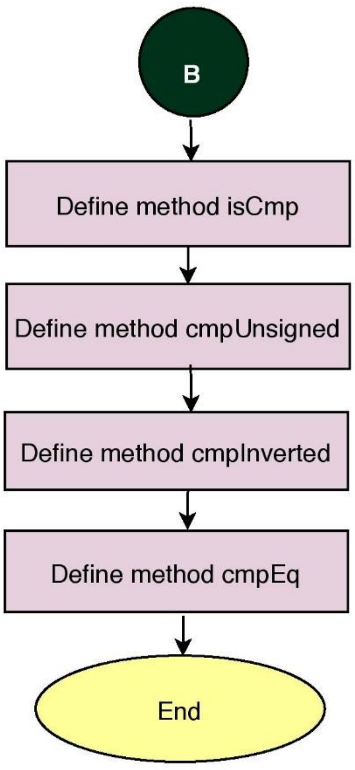
```
def isCmp(cmd: UInt) = cmd >= FN_SLT
  def cmpUnsigned(cmd: UInt) = cmd(1)
  def cmpInverted(cmd: UInt) = cmd(0)
  def cmpEq(cmd: UInt) = !cmd(3)
}
```

explanation

isCmp and cmpUnsigned method is used to help in compare operations of ALU

--- No cmpInverted ----

cmpEq(compare Equals) and cmpInverted(compare Inverted) method is used to check equality between operands of ALU



```
class ALU(implicit p: Parameters) extends CoreModule()(p) {
  val io = new Bundle {
    val dw = Bits(INPUT, SZ_DW)
    val fn = Bits(INPUT, SZ_ALU_FN)
    val in2 = UInt(INPUT, xLen)
    val in1 = UInt(INPUT, xLen)
    val out = UInt(OUTPUT, xLen)
    val adder_out = UInt(OUTPUT, xLen)
    val cmp_out = Bool(OUTPUT)
  }
  // ADD, SUB
  val in2_inv = Mux(isSub(io.fn), ~io.in2, io.in2)
  val in1_xor_in2 = io.in1 ^ in2_inv
  io.adder_out := io.in1 + in2_inv + isSub(io.fn)
  // SLT, SLTU
  val slt =
    Mux(io.in1(xLen-1) === io.in2(xLen-1), io.adder_out(xLen-1),
    Mux(cmpUnsigned(io.fn), io.in2(xLen-1), io.in1(xLen-1)))
  io.cmp_out := cmpInverted(io.fn) ^ Mux(cmpEq(io.fn), in1_xor_in2 ===
UInt(0), slt)
```

explanation

Creates a class ALU and takes implicit parameter. This class ALU is extend by main class Core Module

Class ALU has three inputs and three outputs

ALU I/O	Description
dw	Data width which can be 32bit or 64 bit
fn	Takes Aluop bits as input, width is predefined by 4
in2	Operand B input, here xLen can be changed to 32 or 64bits
in1	Operand A input, here xLen can be changed to 32 or 64bits
out	It is the ALU result output for logical operators, generated by the ALU
adder_out	It is the ALU result output for mathematical operators, generated by the ALU
cmp_out	Branch output weather branch condition True or False

The Next chunk of code will perform addition and subtraction operation of ALU.

Creates a variable in2_inv which have MUX condition. Mux Condition has 1st parameter as Bool in which it has isSub method as input with fn as a parameter in it.If the condition true, get the complement output of ALU second operand input otherwise ALU second operand input will be selected as it is.

In the next step takes XOR operation between operand (in1) and in2_inv(MUX selection output)

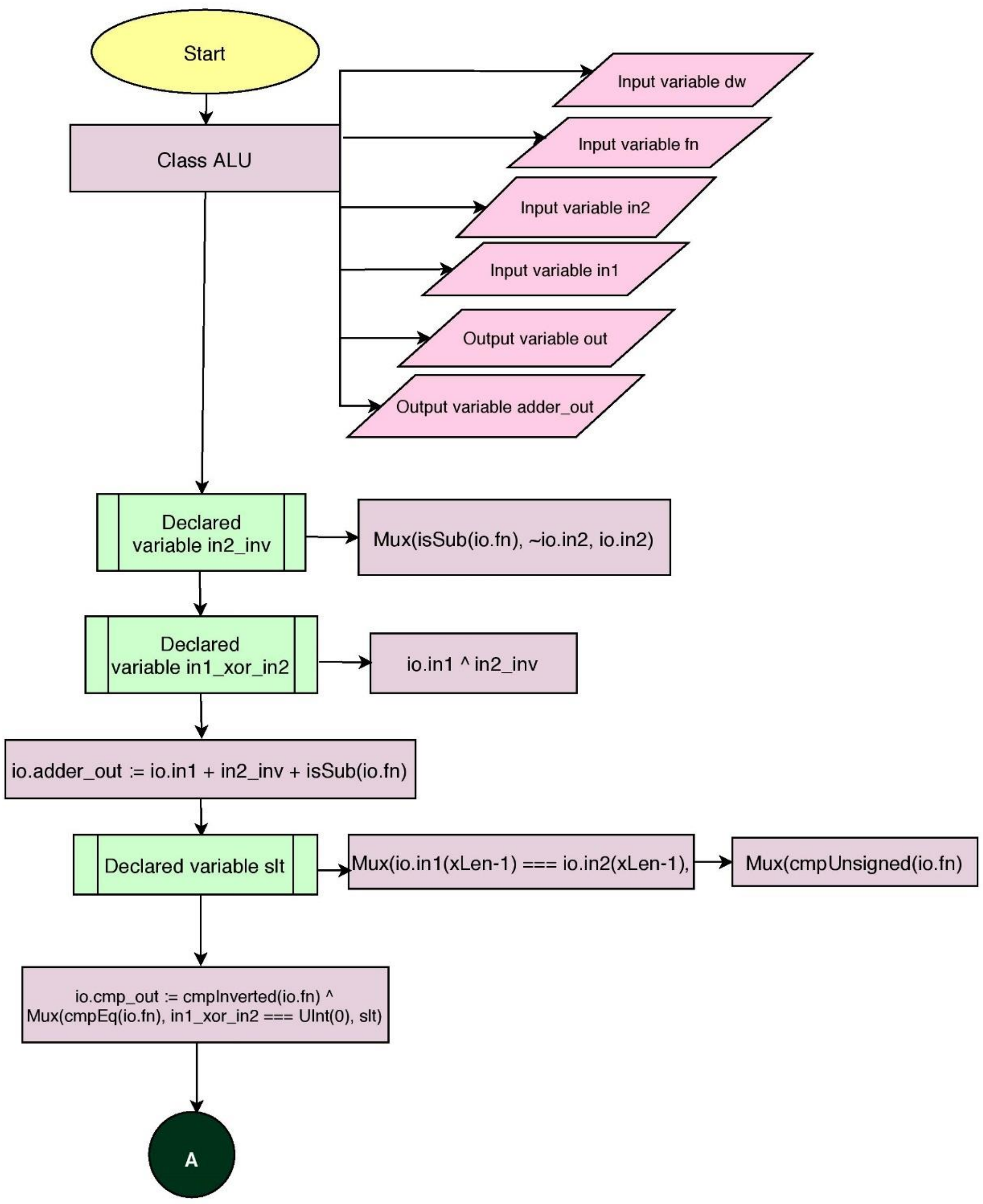
Add in1(operand A of ALU), in2_inv(MUX selection output) and isSub(io.fn) if we have subtraction operation it will be 1 .Wired that result with wire adder_out

This chunk of code will perform set less than (SLT) and set less than unsigned (SLTU)

Creates a variable slt to perform Set less than and set less than unsigned operation on ALU. It contains MUX in which it has Bool condition as parameter which describes if the width of the in1 and in2 are same after subtracting from each width it has true condition and width with subtraction of 1 would be wired with adder_out.

If 1st MUX condition didn't satisfy, 2nd MUX which have function cmpUnsigned as input parameter of Bool with fn as its parameter if condition satisfies, select in2 with its width xlen -1 otherwise in1 will be selected with its width decrement by 1

Use method cmpInverted and takes fn as parameter. Use MUX in order to compare operands and takes cmpEq as Bool parameter Mux input if condition satisfies, select xor result between in1 and in2 and compare it with UINT(0) if result is 0 it outputs 1 otherwise 0,thus if Mux condition did not satisfies slt will be selected and then take XOR with the condition result of MUX with cmpInverted result


Start
Class ALU
Input variable dw
Input variable fn
Input variable in2
Input variable in1
Output variable out
Output variable adder_out
Declared variable in2_inv
Mux(isSub(io.fn), ~io.in2, io.in2)
Declared variable in1_xor_in2
io.in1 ^ in2_inv
io.adder_out := io.in1 + in2_inv + isSub(io.fn)
Declared variable slt
Mux(io.in1(xLen-1) === io.in2(xLen-1),
Mux(cmpUnsigned(io.fn)
io.cmp_out := cmpInverted(io.fn) ^ Mux(cmpEq(io.fn), in1_xor_in2 === UInt(0), slt)
A

```
  // SLL, SRL, SRA
  val (shamt, shin_r) =
    if (xLen == 32) (io.in2(4,0), io.in1)
    else {
      require(xLen == 64)
      val shin_hi_32 = Fill(32, isSub(io.fn) && io.in1(31))
      val shin_hi = Mux(io.dw === DW_64, io.in1(63,32), shin_hi_32)
      val shamt = Cat(io.in2(5) & (io.dw === DW_64), io.in2(4,0))
      (shamt, Cat(shin_hi, io.in1(31,0)))
    }
  val shin = Mux(io.fn === FN_SR  || io.fn === FN_SRA, shin_r,
Reverse(shin_r))
  val shout_r = (Cat(isSub(io.fn) & shin(xLen-1), shin).asSInt >>
shamt)(xLen-1,0)
  val shout_l = Reverse(shout_r)
  val shout = Mux(io.fn === FN_SR || io.fn === FN_SRA, shout_r, UInt(0)) |
              Mux(io.fn === FN_SL,                     shout_l, UInt(0))
```

explanation

This chunk of code will perform shift left logical (SLL), shift right logical (SRL) and shift right arithmetic (SRA)

Creates variables shamt (shift amount) and shin_r (shift right) and assign them values using if condition.

xLen is the parameter which will decide whether we have 32 bit or 64 bit instruction xLen=32

If xLen is equals to 32 shamt will be equals to in2 (operand B 5 bits) and shin_r is equals in1 (operand A)

xLen=64

If xLen is not equals to 32 else condition and execute require method which will compare xLen with 64 and if it equals

Creates variable shin_hi_32 (variable which fills MSB 32 times weather with zeros or ones depending upon the AND condition)

Create variable shin_hi and uses Mux condition if dw (data width) equals to DW_64 in1 (63,32) bits will be extracted otherwise shin_hi_32 variable bits will be selected

Create variable shamt (shift amount) and do concatenations of bits. Shamt variable is used for shift instructions. Used with the shift and rotate instructions, this is the amount by which the source operand rs is rotated/shifted. First performs and operation between in2 (operand B 5th bit) and compare condition of DW_64 (data width 64).

Bits that will concatenate are the result of AND condition between in2 (operand B of ALU 5th bit) and data width condition of 64 bits and the other bits that will concatenate are in2 (operand B bits from 0 to 4)

For shin_r concatenate bits of shin_hi variable and in1 (operand A of ALU) bits from 0 to 31.This ends up our else loop for data width =64

Create variable shin in which use Mux which compares function bits with FN_SR (function shift right) OR FN_SRA (function shift right arithmetic) as Bool parameter if condition satisfies shin_r(shift right) variable will be selected otherwise it will reverse the sequence of shift right variable and select it as Mux result.

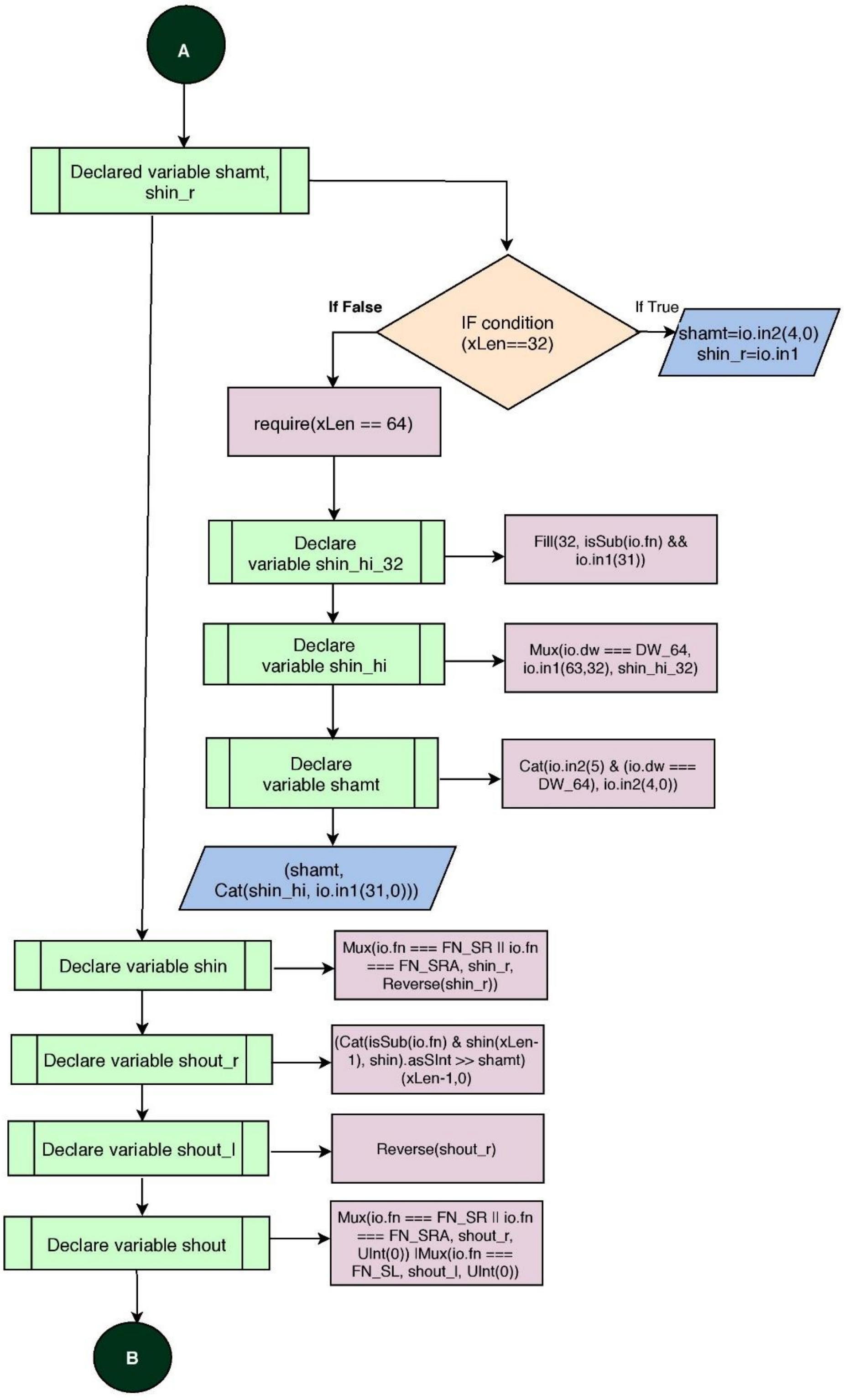
Creates variable shout_r (shift right output).This variable will output of shift right operation by concatenating bits, converts them to signed integer and then performs right shift operation with shift amount (shamt) variable

Variable shout_l will give the output of shift left operation by reverse the shout_r variable result using reverse function

shout variable will give the output of shift operations by comparing functions bits with shift right, shift arithmetic and shift left functions

shout variable which uses MUX and compare fn with FN_SR (shift right) OR fn with FN_SRA (shift right arithmetic) if condition of Mux satisfies shout_r variable selected

Otherwise UInt(0) will be selected, then uses 2nd Mux which compares fn with FN_SL (shift left) if this condition satisfies shout_l will be selected otherwise UInt(0) will be chosen as output


A
Declared variable shamt, shin_r
IF condition (xLen==32)
If False
If True
shamt=io.in2(4,0) shin_r=io.in1
require(xLen == 64)
Declare variable shin_hi_32
Fill(32, isSub(io.fn) && io.in1(31))
Declare variable shin_hi
Mux(io.dw === DW_64, io.in1(63,32), shin_hi_32)
Declare variable shamt
Cat(io.in2(5) & (io.dw === DW_64), io.in2(4,0))
(shamt, Cat(shin_hi, io.in1(31,0)))
Declare variable shin
Mux(io.fn === FN_SR || io.fn === FN_SRA, shin_r, Reverse(shin_r))
Declare variable shout_r
(Cat(isSub(io.fn) & shin(xLen-1), shin).asSInt >> shamt)(xLen-1,0)
Declare variable shout_l
Reverse(shout_r)
Declare variable shout
Mux(io.fn === FN_SR || io.fn === FN_SRA, shout_r, UInt(0)) |Mux(io.fn === FN_SL, shout_l, UInt(0))
B

```
  // AND, OR, XOR

  val logic = Mux(io.fn === FN_XOR || io.fn === FN_OR, in1_xor_in2,
UInt(0)) |

              Mux(io.fn === FN_OR || io.fn === FN_AND, io.in1 & io.in2,
UInt(0))

  val shift_logic = (isCmp(io.fn) && slt) | logic | shout

  val out = Mux(io.fn === FN_ADD || io.fn === FN_SUB, io.adder_out,
shift_logic)

  io.out := out

  if (xLen > 32) {

    require(xLen == 64)

    when (io.dw === DW_32) { io.out := Cat(Fill(32, out(31)), out(31,0)) }

  }

}
```

explanation

This chunk of code will perform logical operation in ALU which includes AND, OR and XOR

variable logic, shift logic and out variable is used to calculate AND, OR and XOR operations in ALU.

logic variable in which it uses two Muxes and between them it perform OR operations.1st Mux uses Bool parameter condition if fn (ALUop bits) is equals to FN_XOR (ALUop bits of XOR) or fn is equals to FN_OR (ALUop bits of OR) if this MUX condition satisfies, XOR operation result between in1 (operand A of ALU) and in2 (Operand B of ALU) as output otherwise UInt (0) will be selected if MUX Bool condition did not satisfies.

In logic variable the 2nd Mux which has fn (ALUop) as Bool parameter if fn is equals to FN_OR or fn is equals to FN_AND, perform logical AND operation between in1(operand A of ALU) and in2 (Operand B of ALU) as output otherwise UInt(0) will be selected if MUX Bool condition did not satisfies.

shift logic variable which takes fn (ALUop bits) as input parameter of method isCmp and perform AND operation with slt(set less than) variable .The result of AND condition will than perform OR bit operations with logic and shout variable

out variable which uses Mux in order to compare fn with FN_ADD and FN_SUB if condition of MUX satisfies adder_out will be output from variable out otherwise shift_logic variable will be output

Variable out output is then wired with ALU out output

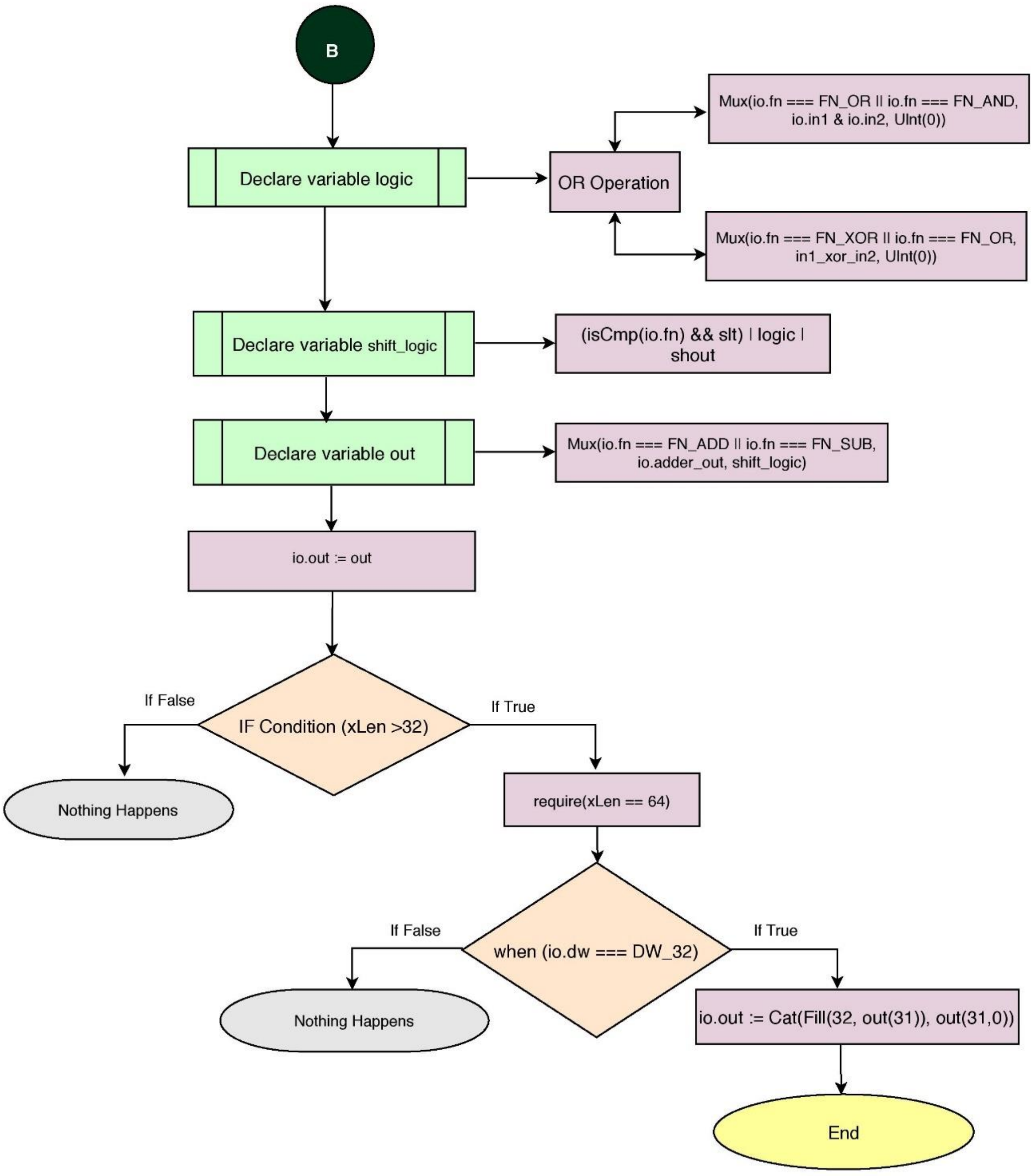
Variable out is used to output mathematical operations result

If xLen parameter is greater than 32, execute if condition and in that have require function which ensures that if xLen is equals to 64 rest of the code below will execute otherwise it will generate an exceptional error.

If require condition satisfies when condition will execute if dw (data width) is equals to 32 concatenate bits and uses Fill command to make 64 bit output

When loop will make sure that if our data width length is 32 make it 64 bit compatible by using Fill command

Fill command used to replicate bits. In this case it uses 31th bit and Fill it by 32 times to make 64 bit output


B
Declare variable logic
OR Operation
Mux(io.fn === FN_OR || io.fn === FN_AND, io.in1 & io.in2, UInt(0))
Mux(io.fn === FN_XOR || io.fn === FN_OR, in1_xor_in2, UInt(0))
Declare variable shift_logic
(isCmp(io.fn) && slt) | logic | shout
Declare variable out
Mux(io.fn === FN_ADD || io.fn === FN_SUB, io.adder_out, shift_logic)
io.out := out
IF Condition (xLen >32)
If False
If True
Nothing Happens
require(xLen == 64)
when (io.dw === DW_32)
If False
If True
Nothing Happens
io.out := Cat(Fill(32, out(31)), out(31,0))
End