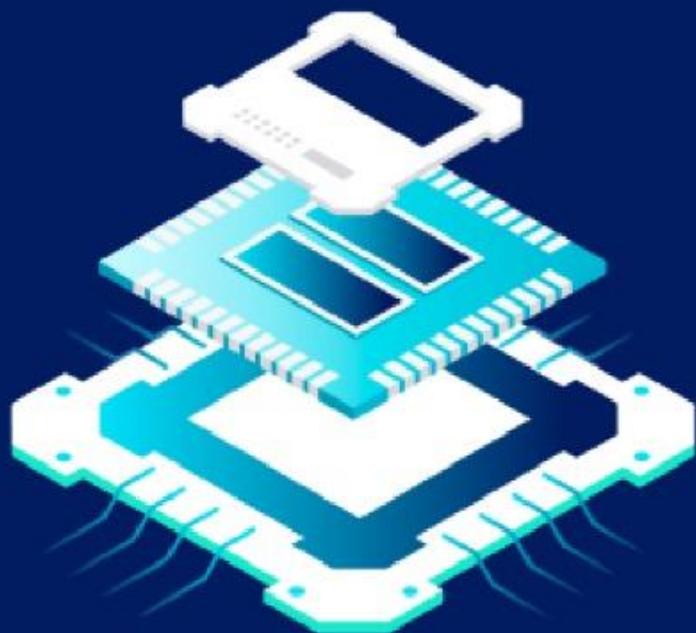


Micro Architecture Specification (MAS) Documentation



Rocket Chip
NPC-GEN

Preface

This MAS Document provides in depth understanding of Modules of Rocket-Chip SoC

Explanation of Modules is achieved by means of Block Diagrams, Class Diagrams, Flowcharts, and explanation of all the keywords found in the code.

This MAS Document is essential for those who have basic knowledge of Scala, CHISEL and SoC hardware.

By going through this Document one can get the proper understanding of Module, also the external dependencies and interconnections required to operate that specific module.

In explanation only the name of keywords used but their definition is mentioned in the Appendix – Scala & CHISEL keywords.

Acknowledgment

This Document was made possible with the counselling, guidance and support of Dr.Roomi Naqvi, Dr Ali Ahmed Ansari, and Farhan Ahmed Karim. Without their encouragement and guidance, this could not be possible in any way.

This Document was created under the Lead of Uzair Khan. along with the team of collaborators/ interns under him that gave their level best under the supervision of such experienced mentors.

Team Includes:

- Shahzaib Kashif
- Talha Ahmed
- Mohsin Raza
- Muhammad Shahzaib
- Fizza Jaffery
- Almas Ibrahim
- Ubab Nadeem

Table of Contents

.....	1
Preface	2
Acknowledgment	3
NPC-GEN Stage	6
Page Table Walker (PTW)	7
Block Diagram:	7
DEEP DIVE INTO CODE:	8
Class Diagram:	8
Explanation	9
Priority Encoder	87
Block Diagram:	87
DEEP DIVE INTO CODE	88
Explanation	88
EPC	111
Block Diagram:	111
DEEP DIVE INTO CODE	112
Explanation	112
NPC CHECK	134
DEEP DIVE INTO CODE	134
Explanation	134
FETCH STAGE	140
Instruction Cache (ICache)	141
Block Diagram:	141
DEEP DIVE INTO CODE	142
Explanation	142
Branch Target Buffer (BTB)	208
Block Diagram:	208
DEEP DIVE INTO CODE	209
Explanation	209
TLB	253
Block Diagram:	253
DEEP DIVE INTO CODE	254
Explanation	254

ROCKET-CHIP Micro Architecture Specification Document

TLB Permissions.....	364
DEEP DIVE INTO CODE	364
Explanation	364
DECODE STAGE.....	385
REG FILE	386
Block Diagram:.....	386
DEEP DIVE INTO CODE	387
Explanation	387
Immediate Generation	393
Block Diagram:.....	393
DEEP DIVE INTO CODE	394
Explanation	394
Instruction Decode (IDecode)	401
DEEP DIVE INTO CODE	401
Explanation	401
RVC	475
DEEP DIVE INTO CODE	475
Scoreboard	500
DEEP DIVE INTO CODE	500
APPENDIX.....	508
Scala & CHISEL keywords:	508
References	509

NPC-GEN Stage

NPC-GEN is the acronym of New PC Generation. The Primary task of NPC-GEN is to predict the Next PC value based on the instruction in further stages of the Rocket-Chip.

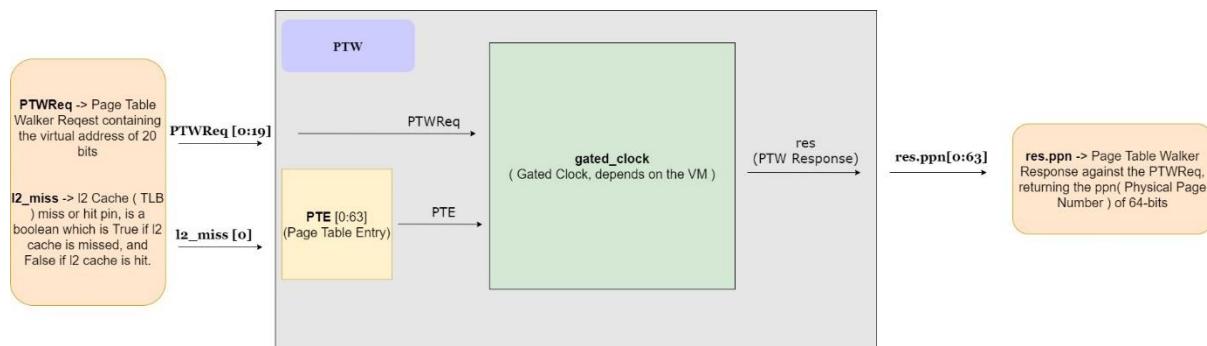
The NPC-GEN Stage contains the following Modules:

- Page Table Walker (PTW)
- Priority Encoder
- NPC Check
- VPC
- EPC

Page Table Walker (PTW)

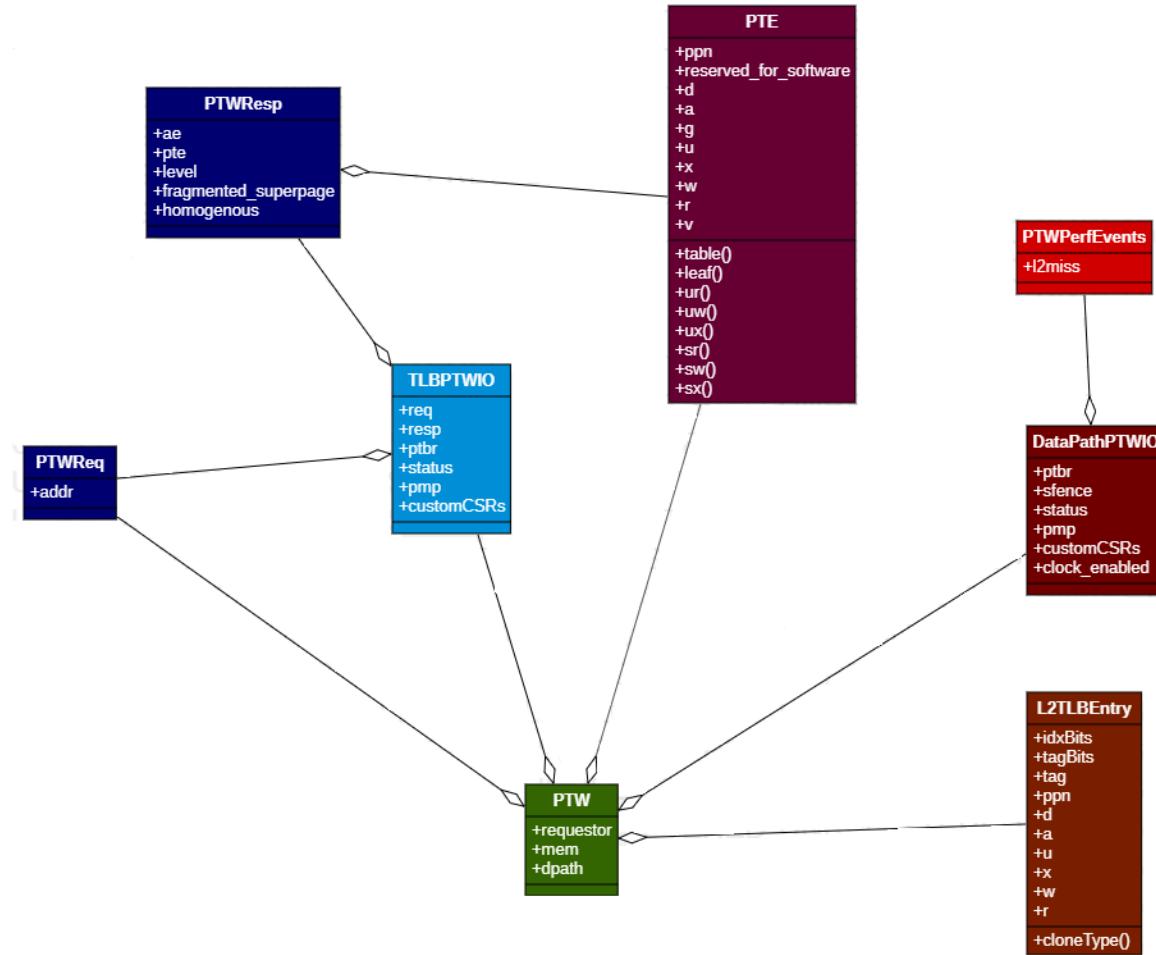
Page Table Walker is a standalone Module in NPC-GEN . It is used to access Virtual Memory. NPC-GEN Module can work without it. The only reason it is placed in this stage is to achieve fast accessing of virtual memory.

Block Diagram:



DEEP DIVE INTO CODE:

Class Diagram:



Explanation (By Means of Flowcharts) :

➤ **Class PTWReq :**

```
class PTWReq (implicit p: Parameters) extends CoreBundle () (p) {  
    val addr = UInt (width = vpnBits)  
}
```

PTWReq -> Page Table Walker Request is initialized with a class.

Where,

addr : address

➤ **Class PTWResp :**

```
class PTWResp(implicit p: Parameters) extends CoreBundle() (p) {  
    val ae = Bool()  
    val pte = new PTE  
    val level = UInt(width = log2Ceil(pgLevels))  
    val fragmented_superpage = Bool()  
    val homogeneous = Bool()  
}
```

PTWResp -> Page Table Walker Response is initialized with a class.

Where,

ae: Boolean
pte: Page Table Entry (seprate class)
level: → log2Ceil: returns (n-1) for integers; pgLevels is the number of page tables coming from some other module (maybe Main File)
fragmented_superpage: Boolean
homogeneous: Boolean

➤ Class TLBPTWIO :

```
class TLBPTWIO(implicit p: Parameters) extends CoreBundle()(p)
  with HasCoreParameters {
  val req = Decoupled(Valid(new PTWReq))
  val resp = Valid(new PTWResp).flip
  val ptbr = new PTBR().asInput
  val status = new MStatus().asInput
  val pmp = Vec(nPMPs, new PMP).asInput
  val customCSRs = coreParams.customCSRs.asInput
}
```

TLBTWIO-> TLB(Translation Lookaside Buffer)
PTW(Page Table Walker) IO(Input Output)

where,

CoreBundle: is the bundle which is extended by almost every other class, contains the basic code that is needed by all of them
HasCoreParameter: is a trait also in some other file.

req: → Decoupled: adds a decoupled handshaking protocol; **PTWReq:** ptw req class.

resp: → Valid: adds a ‘valid’ bit to some data; **PTWRes** object is instantiated.

ptbr: → PTBR object is instantiated as input.
status: → MStatus object is instantiated as input.

pmp: → PMP object is instantiated as input.

customCSRs: → coreParams.customCSRs object is instantiated as input.

➤ Class PTWPerfEvents :

```
class PTWPerfEvents extends Bundle {
  val l2miss = Bool()
}
```

PTW(Page Table Walker) Perf(Performed) Events

where,

l2miss: → boolean for when TLB L2 Cache is missed.

➤ **Class DatapathPTWIO :**

```
class DatapathPTWIO(implicit p: Parameters) extends CoreBundle()(p)
  with HasCoreParameters {
  val ptbr = new PTBR().asInput
  val sfence = Valid(new SFenceReq).flip
  val status = new MStatus().asInput
  val pmp = Vec(nPMPs, new PMP).asInput
  val perf = new PTWPerfEvents().asOutput
  val customCSRs = coreParams.customCSRs.asInput
  val clock_enabled = Bool(OUTPUT)
}
```

Datapath PTW(Page Table Walker)
IO(Input Output)

where,

ptbr: → PTBR object is instantiated as input.

sfence: → SFenceReq object is instantiated .

status: → MStatus object is instantiated as input.

pmp: → PMP object is instantiated as input.

perf : → PTWPerfEvents object is instantiated as output.

CustomCSRs: → coreParams.customCSRs object is instantiated as input.

clock_enabled: → Boolean

➤ **Class PTE :**

```
class PTE(implicit p: Parameters) extends CoreBundle()(p) {
    val ppn = UInt(width = 54)
    val reserved_for_software = Bits(width = 2)
    val d = Bool()
    val a = Bool()
    val g = Bool()
    val u = Bool()
    val x = Bool()
    val w = Bool()
    val r = Bool()
    val v = Bool()

    def table(dummy: Int = 0) = v && !r && !w && !x
    def leaf(dummy: Int = 0) = v && (r || (x && !w)) && a
    def ur(dummy: Int = 0) = sr() && u
    def uw(dummy: Int = 0) = sw() && u
    def ux(dummy: Int = 0) = sx() && u
    def sr(dummy: Int = 0) = leaf() && r
    def sw(dummy: Int = 0) = leaf() && w && d
    def sx(dummy: Int = 0) = leaf() && x
}
```

PTE->Page Table Entry

where,

ppn → 54 bits
reserved_for_software → 2 bits
other variables → 8 bits
TOTAL : 64 bits

➤ **Class L2TLBEntry :**

```
class L2TLBEntry(implicit p: Parameters) extends CoreBundle()(p)
  with HasCoreParameters {
  val idxBits = log2Ceil(coreParams.nL2TLBEntries)
  val tagBits = vpnBits - idxBits
  val tag = UInt(width = tagBits)
  val ppn = UInt(width = ppnBits)
  val d = Bool()
  val a = Bool()
  val u = Bool()
  val x = Bool()
  val w = Bool()
  val r = Bool()
  override def cloneType = new L2TLBEntry().asInstanceOf[this.type]
}
```

➤ Class PTW :

```

@chiselName

class PTW(n: Int) (implicit edge: TLEdgeOut, p: Parameters) extends
CoreModule() (p) {

    val io = new Bundle {

        val requestor = Vec(n, new TLBPTWIO).flip

        val mem = new HellaCacheIO

        val dpath = new DatapathPTWIO

    }

    val s_ready :: s_req :: s_wait1 :: s_dummy1 :: s_wait2 :: s_wait3 :: 
    s_dummy2 :: s_fragment_superpage :: Nil = Enum(UInt(), 8)

    val state = Reg(init=s_ready)
}

```

 explanation

chiselName decorator is used before creation of Class PTW

class PTW is defined with parameter n as an Int, and implicit edge and p extended by CoreModule.

io is defined with Bundle for defining input outputs.

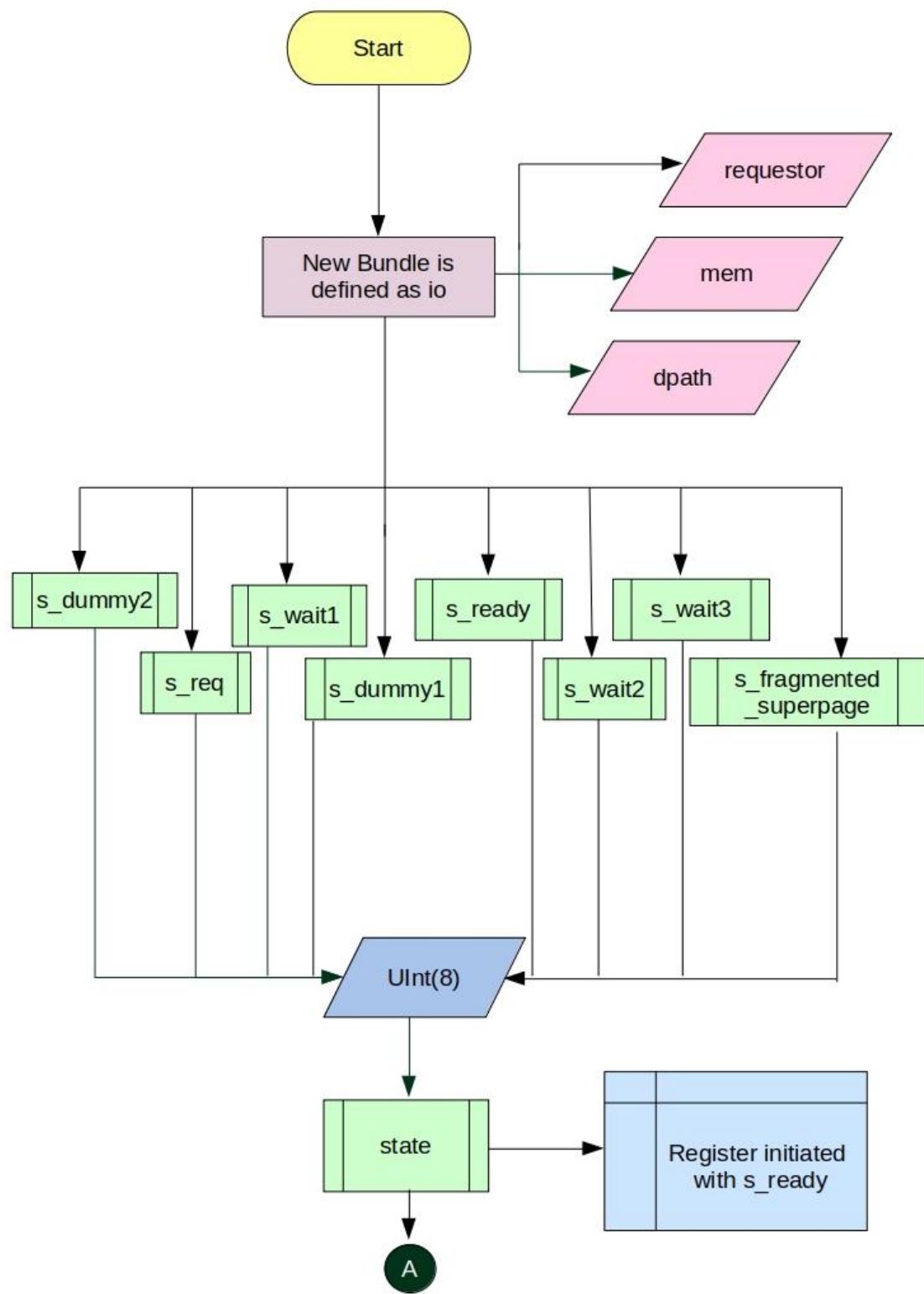
requestor is defined as Vector of length n and TLBPTWIO object with flipped bits on every index.

mem is defined as HellaCacheIO object

dpath is defined as DatapathPTWIO object

then 9 variables are instantiated with Enum as UInt of 8(0-8)

state is defined with Reg initiated with s_ready.



```
val arb = Module(new Arbiter(Valid(new PTWReq), n))

arb.io.in <> io.requestor.map(_.req)

arb.io.out.ready := state === s_ready

val resp_valid = Reg(next=Vec.fill(io.requestor.size) (Bool(false))
```

explanation

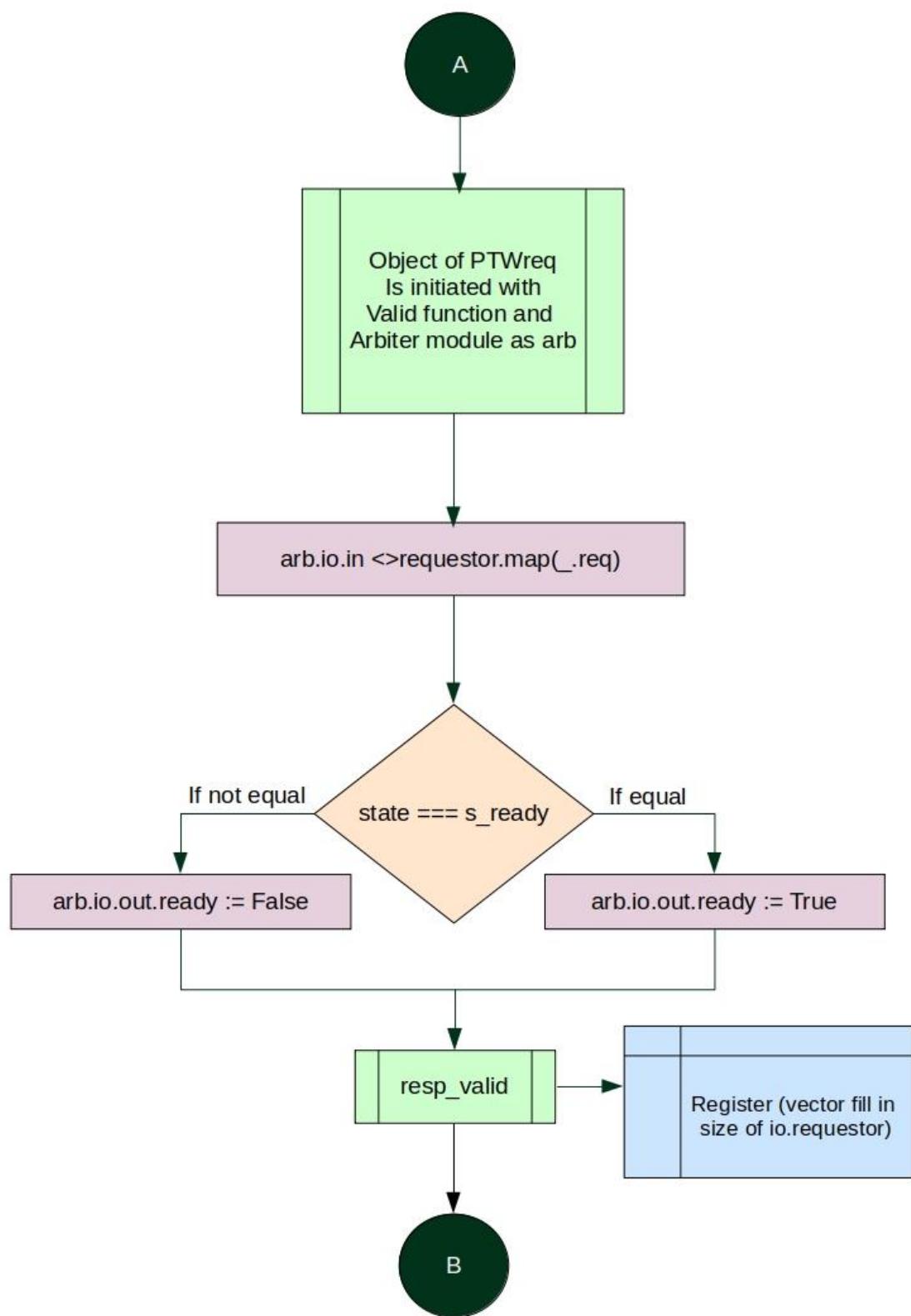
arb is defined with Arbiter object with PTWRrq object with Valid bit and n as a parameter.

arb.io.in calls <> method of Arbiter class with io.requestor mapped with _.req, as a parameter.

[In Scala's syntax the Method notation is in such a way that object.method(parameter) can be written separately like, object method parameter]

arb.io.out.ready calls := method of Arbiter Class with Bool value on condition; state === s_ready.

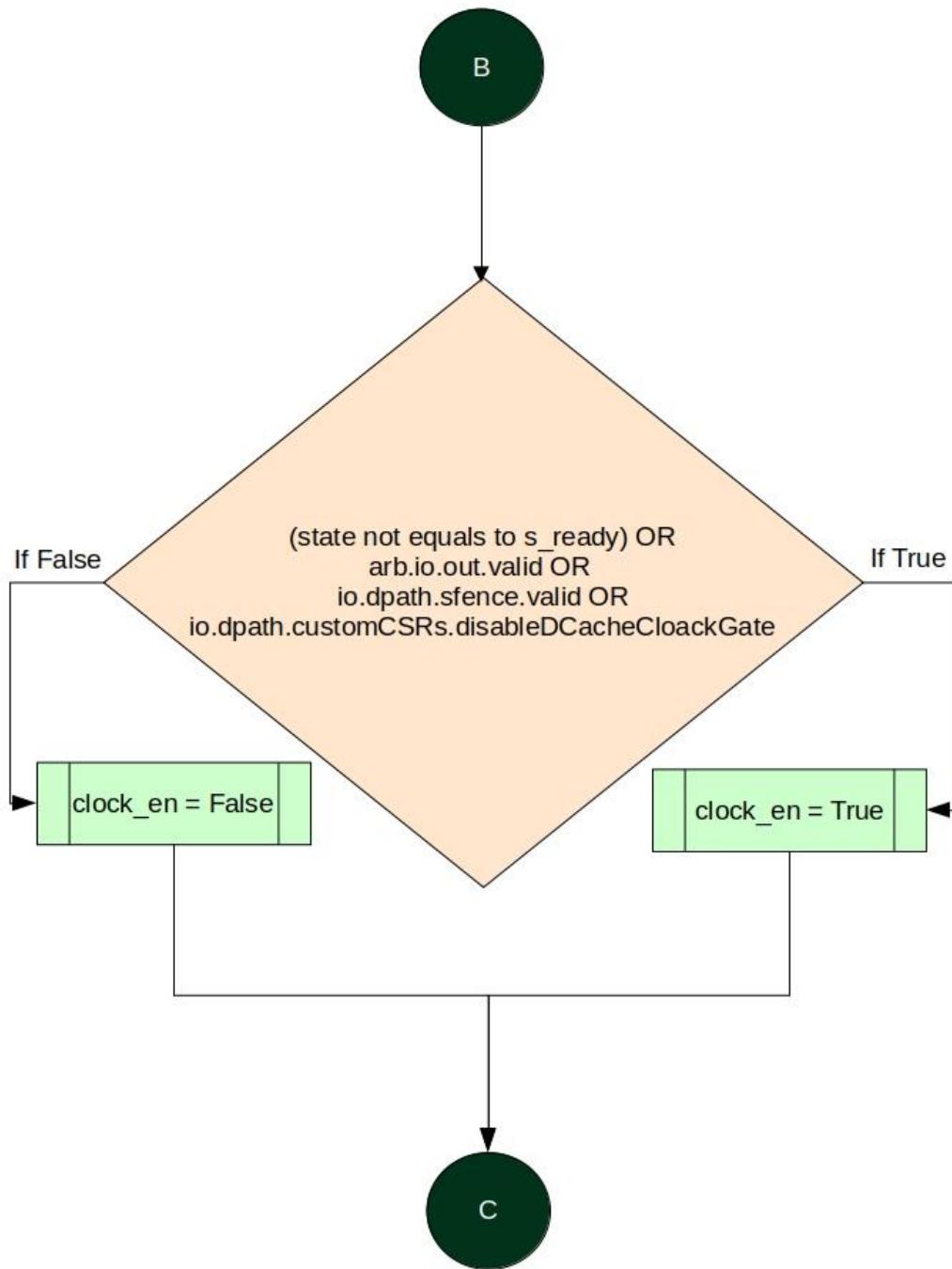
resp_valid variable is defined with Reg initialized with next value as Vec of length io.requestor.size and filled with Boolean false on every index.



```
val clock_en = state /= s_ready || arb.io.out.valid || io.dpath.sfence.valid || io.dpath.customCSRs.disableDCacheClockGate
```

explanation

clock_en variable is initialized as Bool depending upon the given condition.



```

io.dpath.clock_enabled := usingVM && clock_en
val gated_clock =
  if (!usingVM || !tileParams.dcache.get.clockGate) clock
  else ClockGate(clock, clock_en, "ptw_clock_gate")

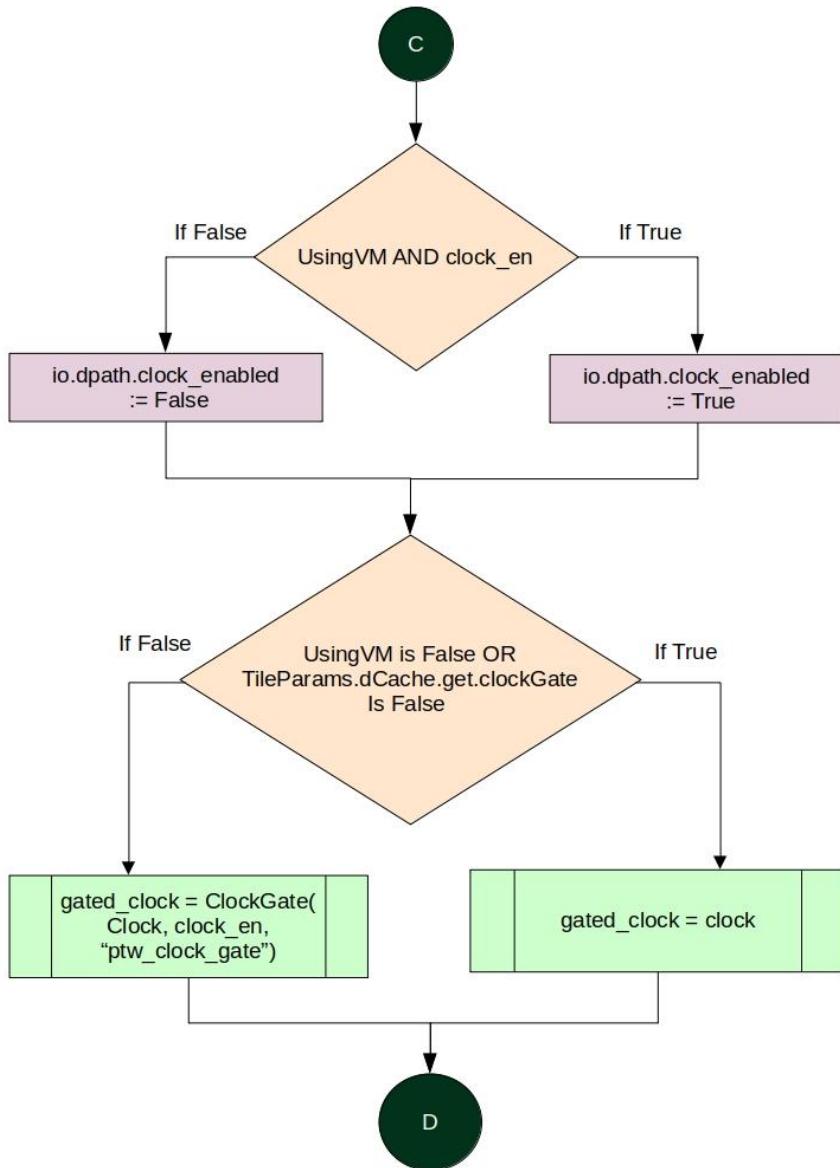
```

explanation

dpah clock_enabled input is wired as Bool on condition; usingVM AND'ed with clock_en

means if the Virtual Memory is in use and clock is also enable then dpath clock will be enabled (true)

gated_clock variable is initialized in such a way that if VM will not be in use and tileParams won't have a clock gate as well then normal Clock will be its value otherwise a Clock Gate will be formed for it.



ROCKET-CHIP Micro Architecture Specification Document

```
withClock (gated_clock) { // entering gated-clock domain
  val invalidated = Reg(Bool())
  val count = Reg(UInt(width = log2Up(pgLevels)))
  val resp_ae = RegNext(false.B)
  val resp_fragmented_superpage = RegNext(false.B)
  val r_req = Reg(new PTWReq)
```

— explanation —

withClock receives a clock parameter and creates a scope based on that received clock.

gated_clock is passed as a parameter for withClock, to get into the scope of gated_clock.

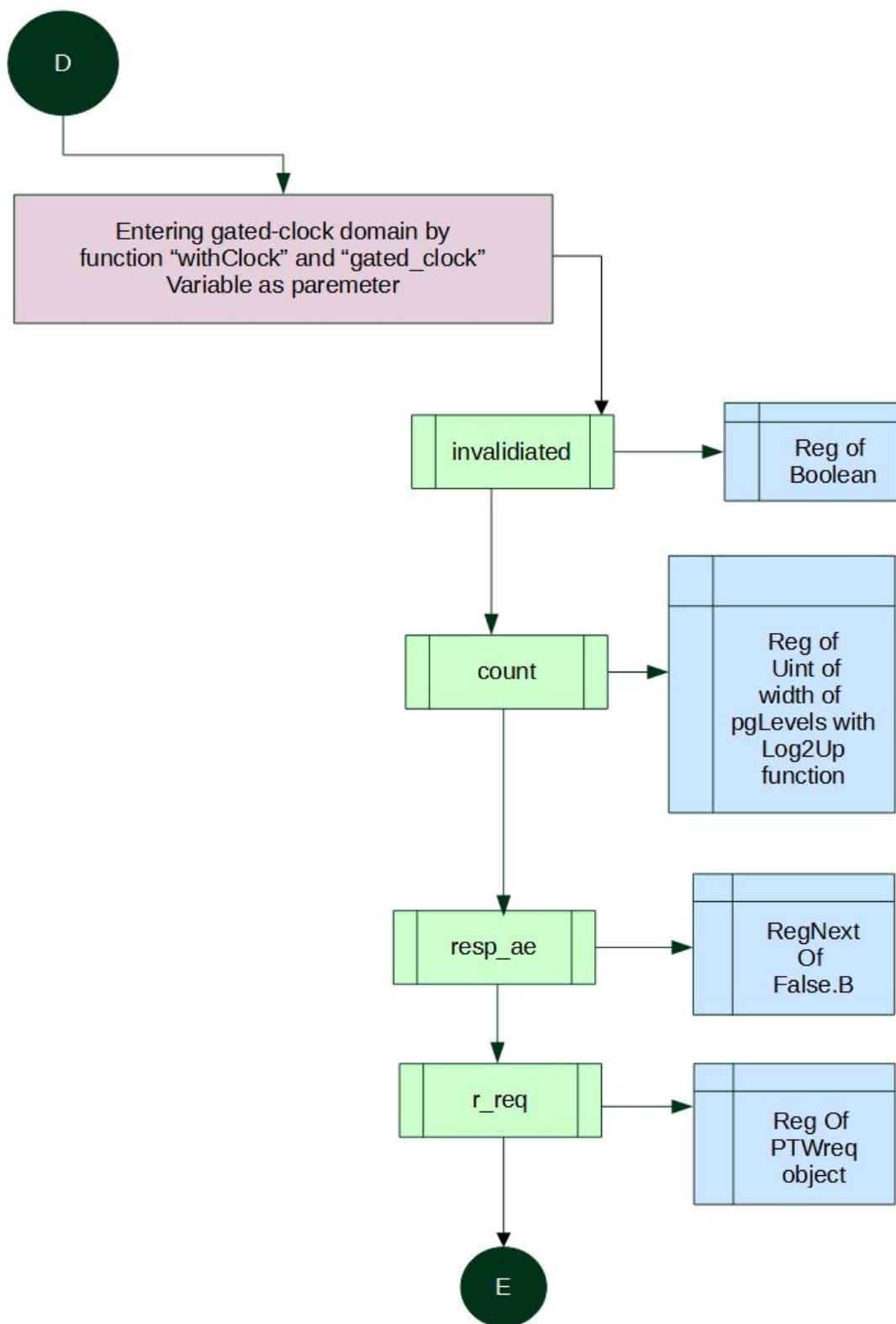
Invalidated is defined with a n Reg having a Boolean value

Count is defined with Reg having UInt of width pgLevels after log2Up.

resp.ae is defined with RegNext having false.B

resp_fragmented_superpage is defined with RegNext having false.B

r_req is defined with Reg having PTWReq object



```

val r_req_dest = Reg(Bits())  

val r_pte = Reg(new PTE)  

val mem_resp_valid = RegNext(io.mem.resp.valid)  

val mem_resp_data = RegNext(io.mem.resp.bits.data)

```

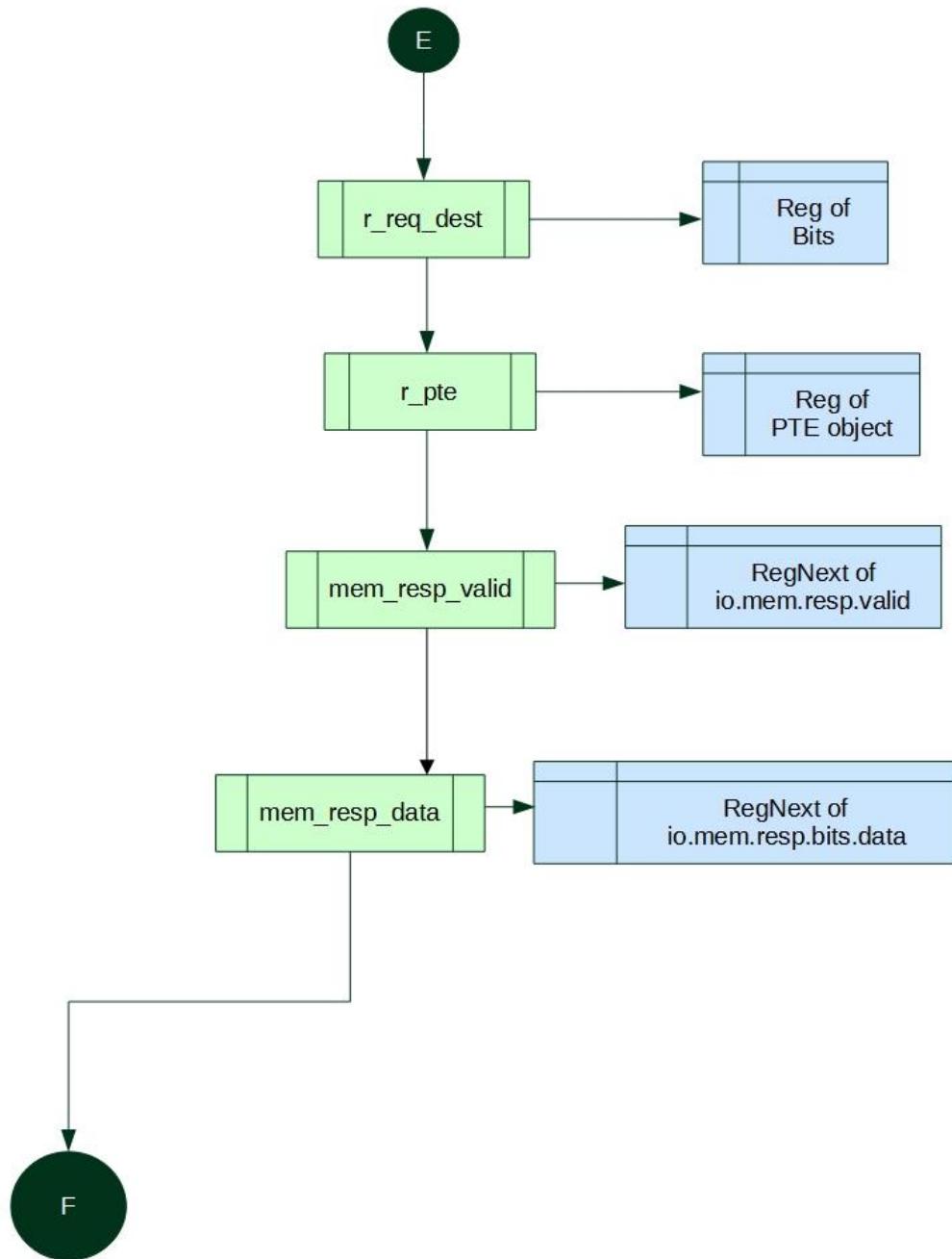
 explanation

r_req_dest is defined with Reg having Bits value

r_pte is defined with Reg having PTE object

mem_resp_valid is defined with RegNext of io.mem.resp.valid

mem_resp_data is defined with RegNext of io.mem.resp.bits.data



```
io.mem.uncached_resp.map { resp =>
    assert(! (resp.valid && io.mem.resp.valid))
    resp.ready := true
    when (resp.valid) {
        mem_resp_valid := true
        mem_resp_data := resp.bits.data
    }
}
```

— explanation —

mem unchahed_resp is mapped with in such a way,

assert function is called with the given condition

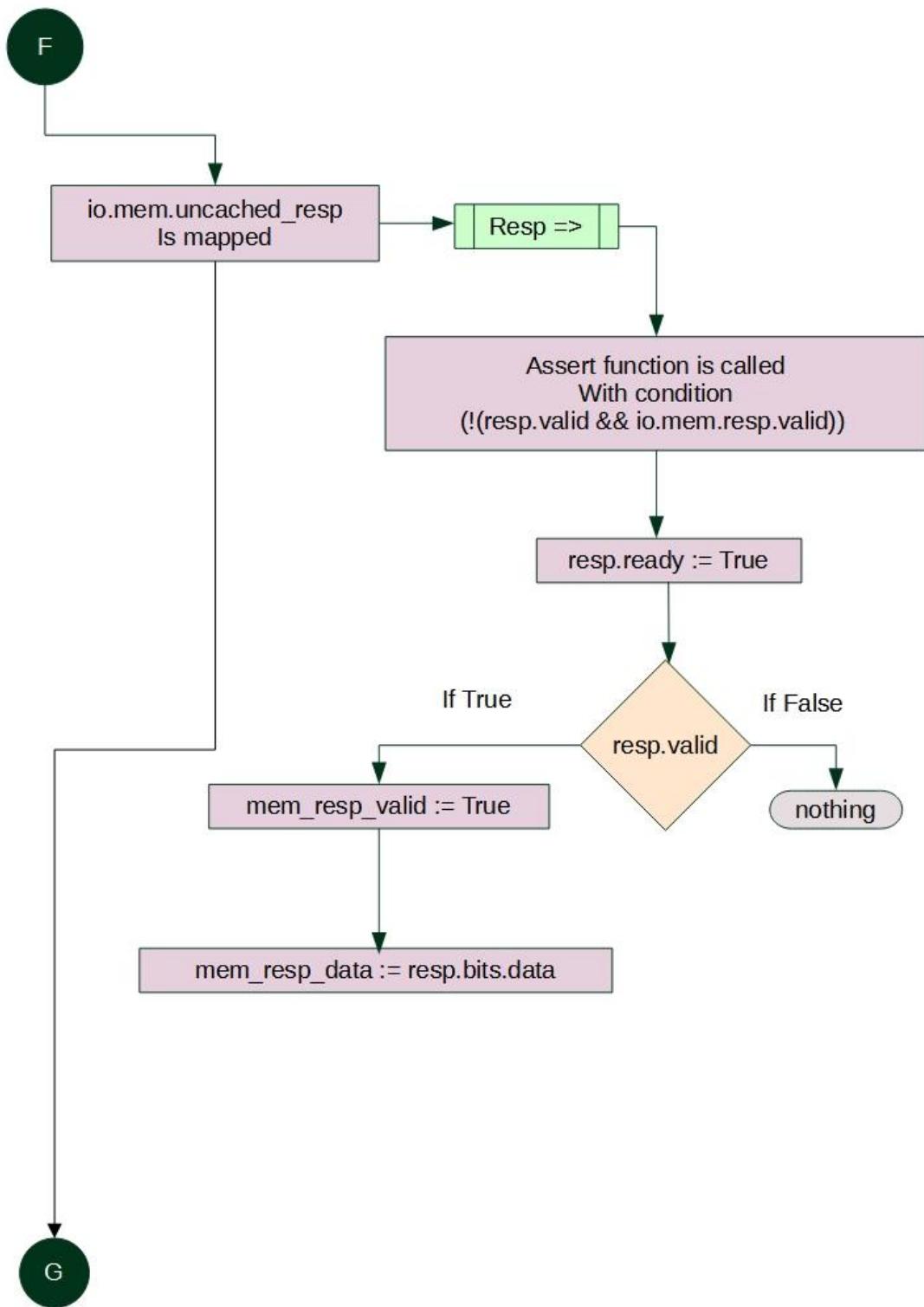
resp.ready is wired true

when resp.valid is true, then

mem_resp_valid is wired true

mem_resp_data is wired with resp.bits.data

what happened here is that uncached_resp in mem is iterated, and then is resp is already valid along with mem resp is valid too then by use of assert an exception is thrown, then resp is indicated that it is ready, then mem_resp_valid, means resp coming from memory is valid and inside mem_resp_data, the data in resp is saved.



```

val (pte, invalid_paddr) = {
  val tmp = new PTE().fromBits(mem_resp_data)
  val res = Wire(init = tmp)
  res.ppn := tmp.ppn(ppnBits-1, 0)

```

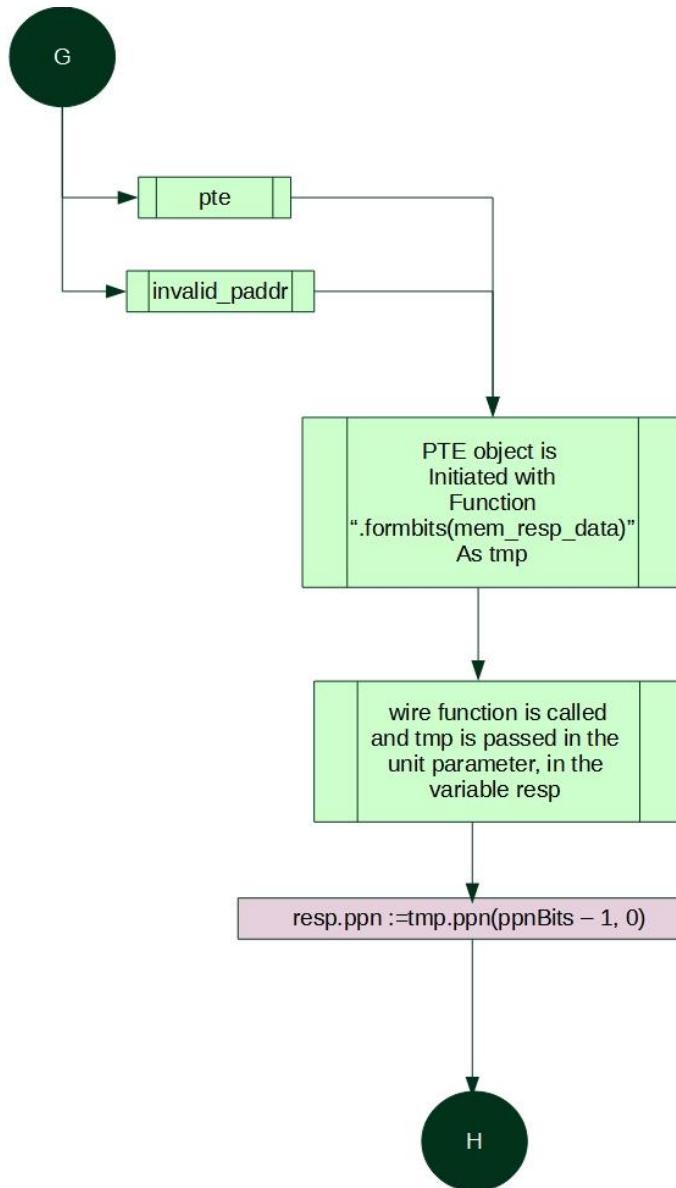
 explanation

pte, and invalid_paddr are initiated inside a code block, in which
tmp is defined with a PTE object instantiated from the bits coming from
mem_resp_data

res has the tmp with Wire function

res.ppn is wired with tmp.ppn(ppnBits-1 to)

res.ppn got the same ppn but reversed.



```
when (tmp.r || tmp.w || tmp.x) {
    // for superpage mappings, make sure PPN LSBs are zero
    for (i <- 0 until pgLevels-1)
        when (count <= i && tmp.ppn((pgLevels-1-i)*pgLevelBits-1,
(pgLevels-2-i)*pgLevelBits) /= 0) { res.v := false }
    }
    (res, (tmp.ppn >> ppnBits) /= 0)
}
```

explanation

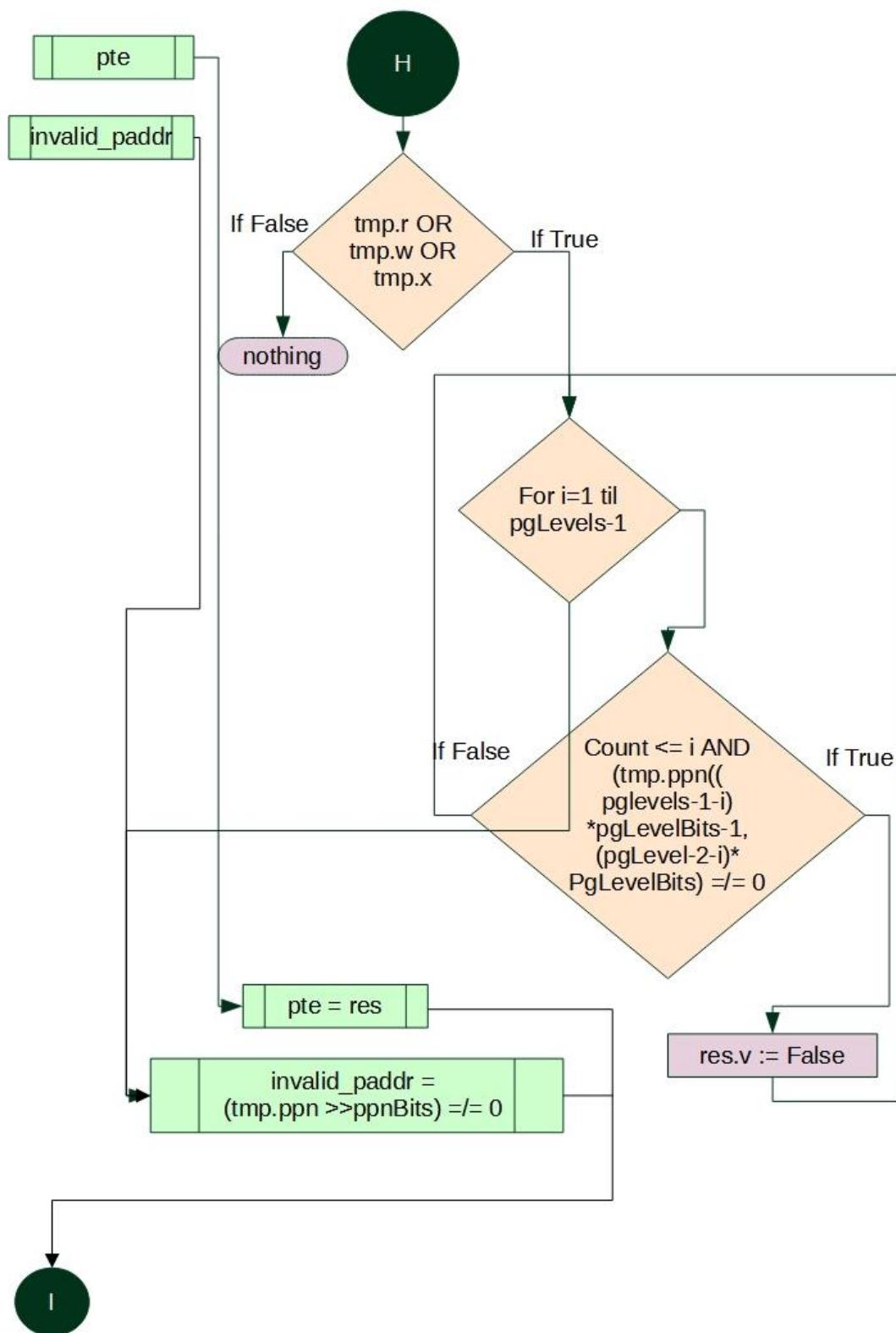
when tmp.r w and x will be true then

a loop will be iterated from 0 to pgLevels-1, in which

if the count variable will be <= current value index AND the calculation isn't equal to 0, then res.v will be wired false

and pte will get the whole res variable

and invalid paddr will get the Boolean based on the condition that tmp.ppn right shifted by ppnBits is equals to zero or not.



```

val traverse = pte.table() && !invalid_paddr && count < pgLevels-1
val pte_addr = if (!usingVM) 0.U else {
    val vpn_idxs = (0 until pgLevels).map(i => (r_req.addr >> (pgLevels-i-1)*pgLevelBits) (pgLevelBits-1,0))
    val vpn_idx = vpn_idxs(count)
    Cat(r_pte.ppn, vpn_idx) << log2Ceil(xLen/8)
}

```

———— explanation —————

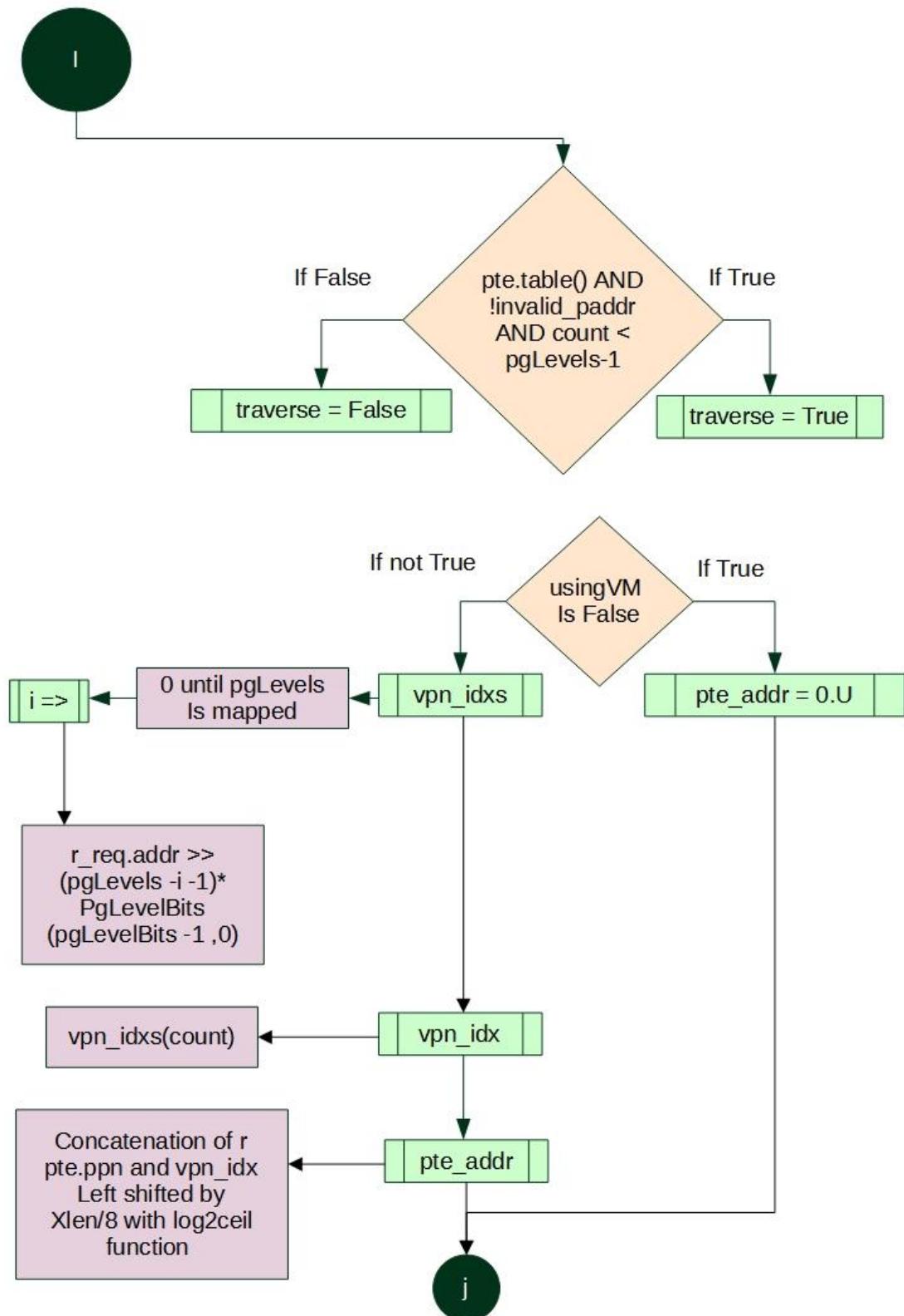
traverse is defined as Boolean upon the given condition.

pte_addr will be 0 if VM won't be in use, otherwise,

loop will be iterated over 0 to pgLevels-1, in which every iteration will have a different binary data based on the iteration, and the whole list will be saved in vpn_idxs

from that list, the only value that is on the index count(log2Up of pgLevels) will be saved in vpn_idx

pte_addr is Concatenation of original PTE ppns with that specific index bits which are left shifted by log2Ceil of xLen/8



```

val fragmented_superpage_ppn = {
    val choices = (pgLevels-1 until 0 by -1).map(i => Cat(r_pte.ppn >>
(pgLevelBits*i), r_req.addr(((pgLevelBits*i) min vpnBits)-1,
0).padTo(pgLevelBits*i)))
    choices(count)
}
when (arb.io.out.fire()) {
    r_req := arb.io.out.bits.bits
    r_req_dest := arb.io.chosen
}

```

— explanation —

fragmented superpage is being defined in such a way,

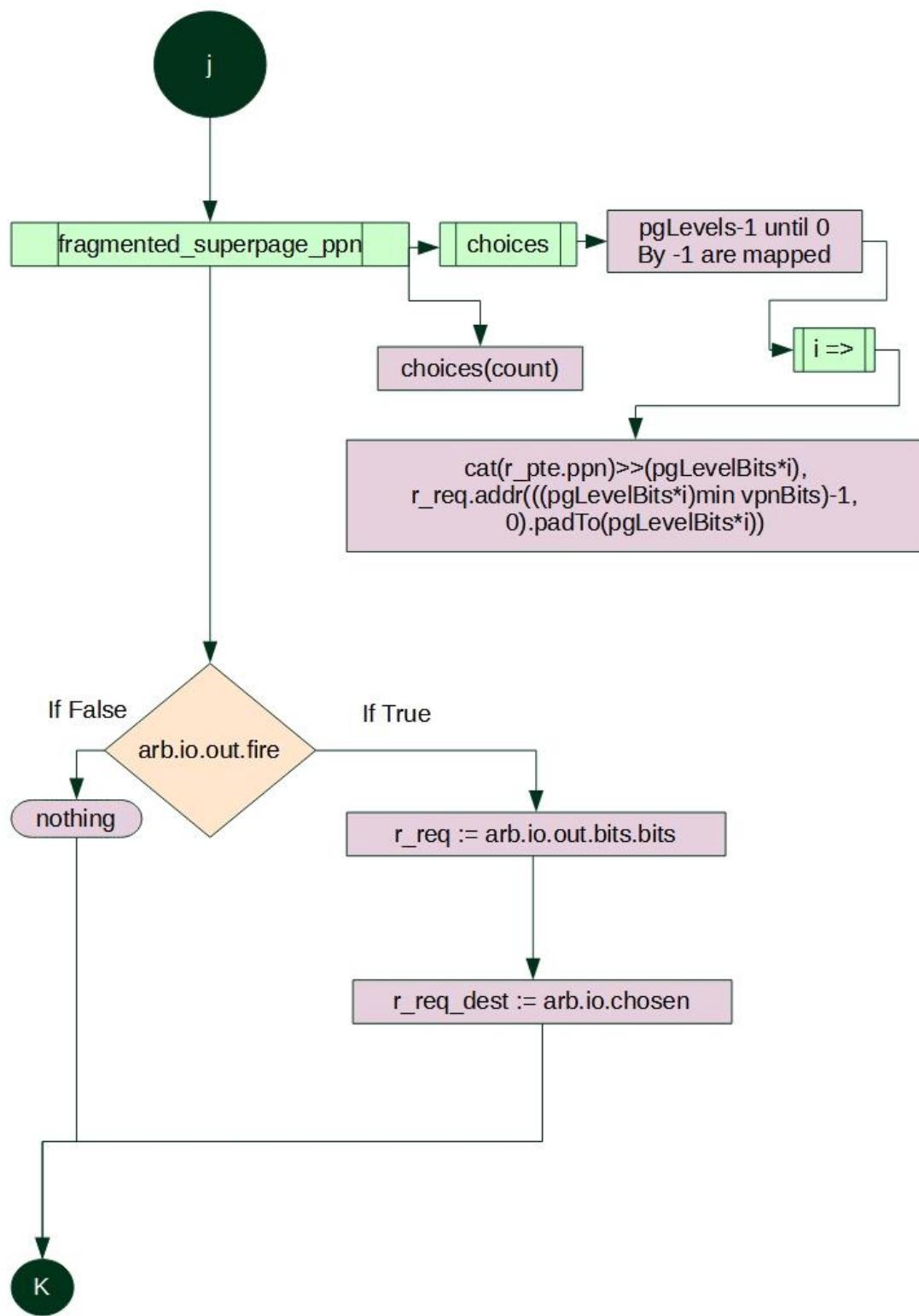
the list is iterated in reverse here from pgLevels-1 to 0 with certain concatenation on every iteration, saved in choices

fragmented_superpage has the specific index of count(log2Up os pgLevels)

when arb.io.out.fire will be true then

r_req will be wired to arb.io.out.bits.bits

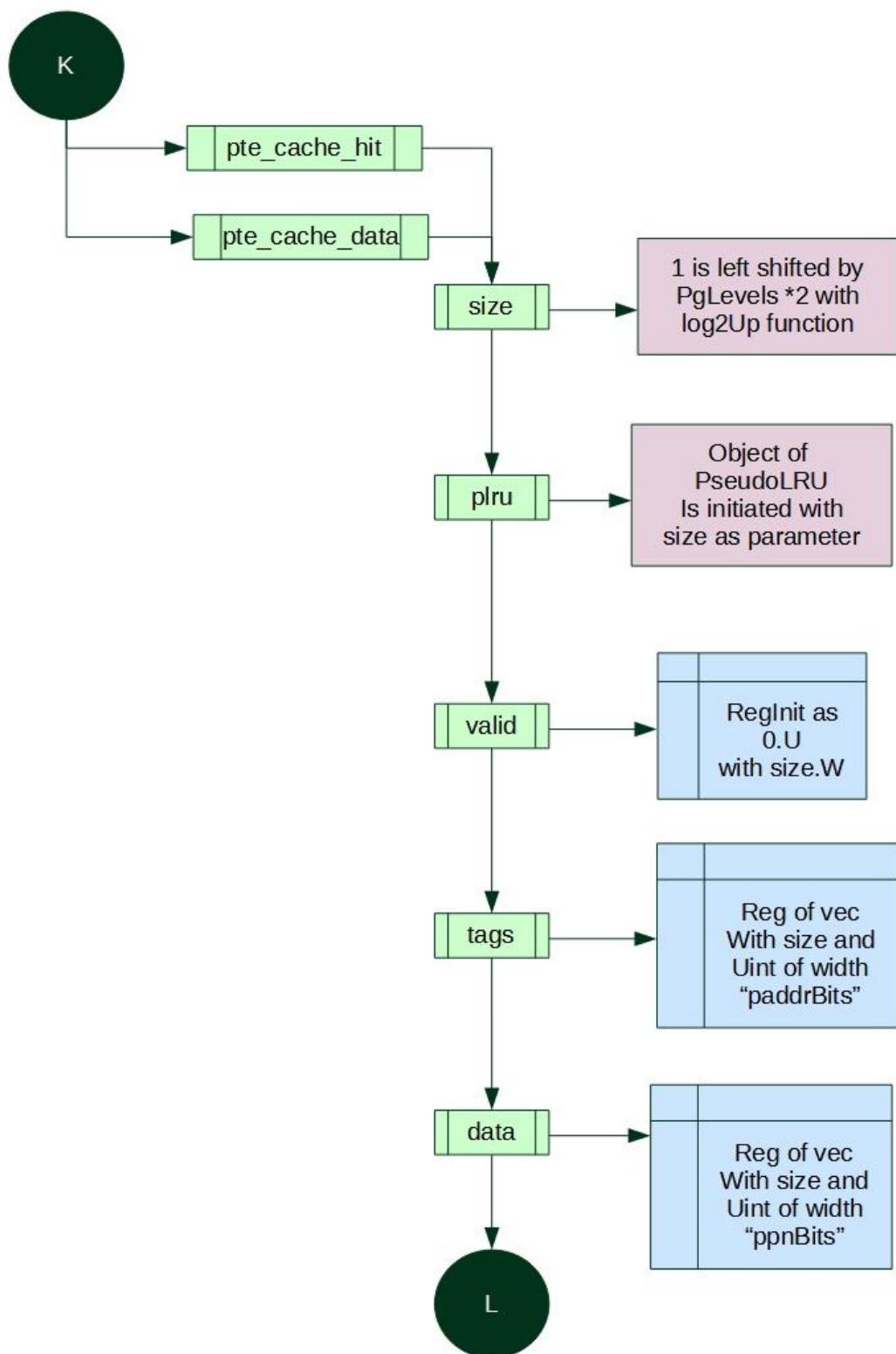
r_req_dest will be wired to arb.io.chosen



```
val (pte_cache_hit, pte_cache_data) = {
    val size = 1 << log2Up(pgLevels * 2)
    val plru = new PseudoLRU(size)
    val valid = RegInit(0.U(size.W))
    val tags = Reg(Vec(size, UInt(width = paddrBits)))
    val data = Reg(Vec(size, UInt(width = ppnBits)))
}
```

— explanation —

pte_cache_hit and data are defined,
size is initialized with 1 left shifted with log2Up of pgLevels * 2
plru is initialized as object of PseudoLRU with size as the parameter
valid has RegInit og 0.U with width of Size
tags has a Reg of Vec of length size and values are UInt of width paddrBits
data has a Reg of Vec of length size and values are UInt of width ppnBits



```

val hits = tags.map(_ === pte_addr).asUInt & valid
val hit = hits.orR
when (mem_resp_valid && traverse && !hit && !invalidated) {
    val r = Mux(valid.andR, plru.way, PriorityEncoder(~valid))
    valid := valid | UIntToOH(r)
    tags(r) := pte_addr
    data(r) := pte.ppn
}

```

— explanation —

hits has tags iterated with pte_addr===_ bit AND'ed with valid bits

All the bits of hits OR'ed with each other into hit with .orR

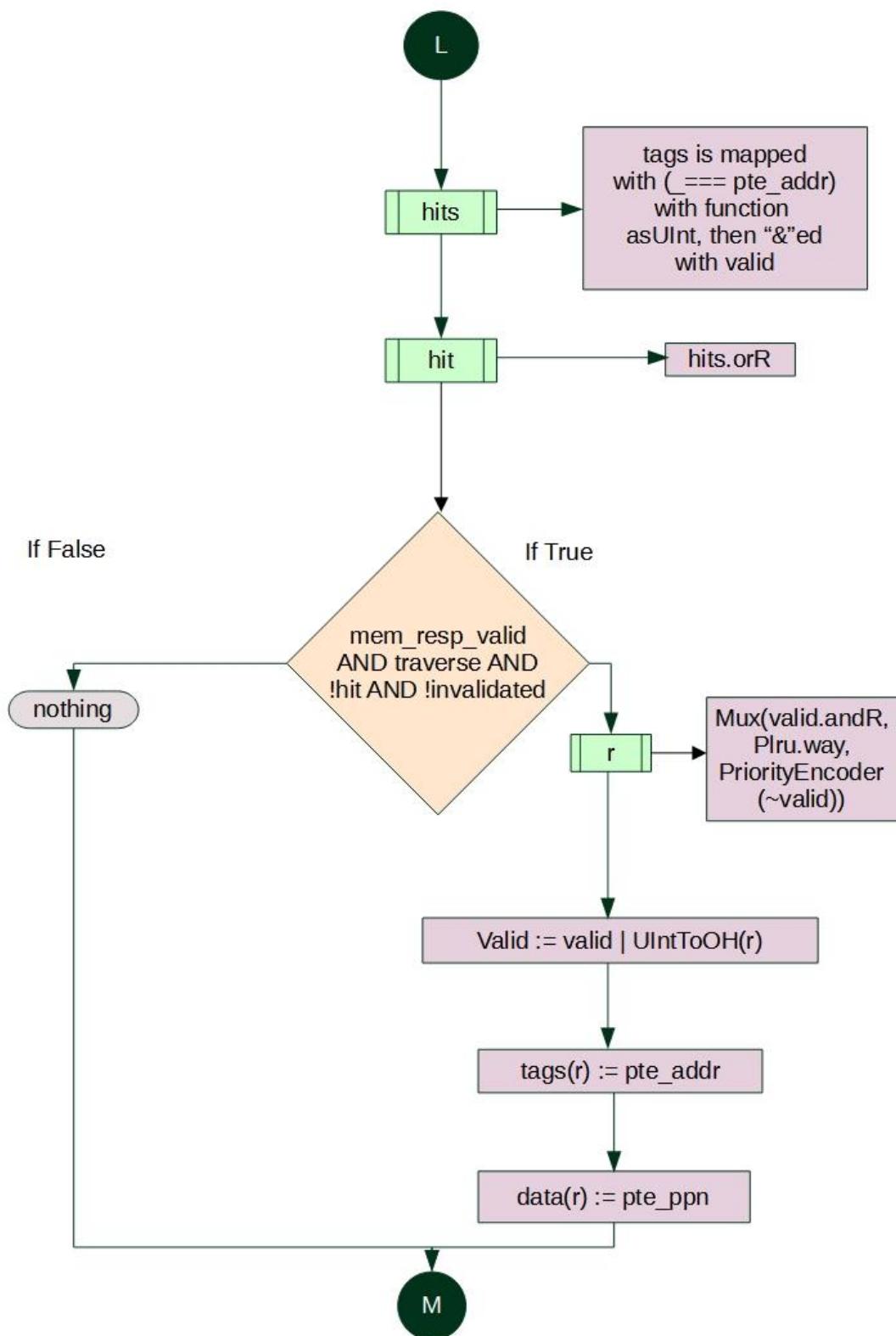
when mem_resp is valid AND traverse is true AND hit is false(0) AND invalidated is false, then,

r = Mux with cond as valid bits AND'ed with each other and options as plru.way and PriorityEncoder of valid bits reversed

valid wired with valid existing bits OR'ed with value of r converted into binary by UIntToOH()

tags value at index r is wired with pte_addr

data value at index r wired with pte.ppn



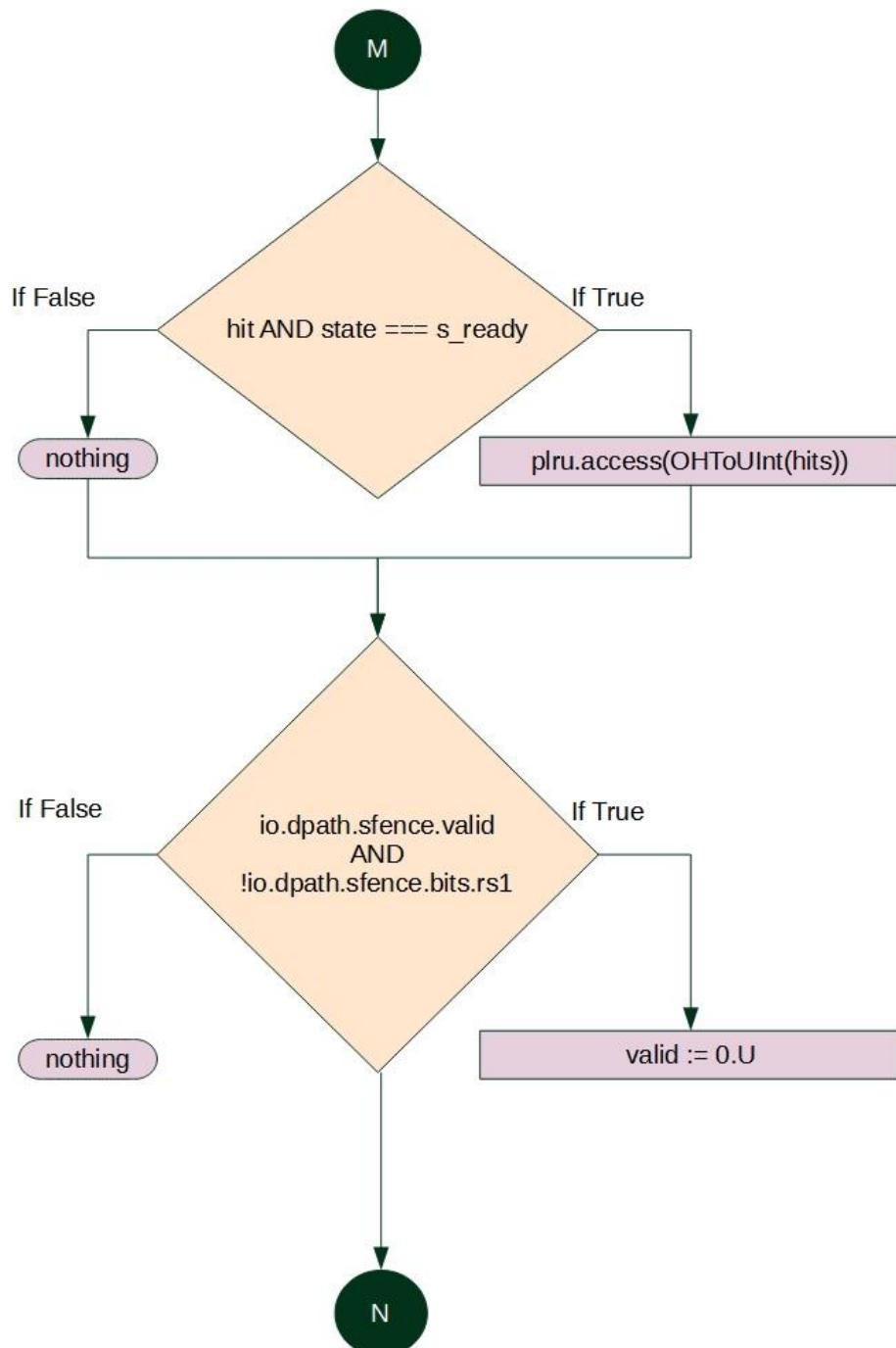
```

when (hit && state === s_req) { plru.access(OHToUInt(hits)) }
when (io.dpath.sfence.valid && !io.dpath.sfence.bits.rs1) { valid := 0.U }

```

 explanation

when hit is true and state == s_req then,
plru.access receives hits value in UInt.
when sfence is valid and rs1 isn't, then valid is wired 0



```

for (i <- 0 until pgLevels-1)
    ccover(hit && state === s_req && count === i, s"PTE_CACHE_HIT_L$i",
s"PTE cache hit, level $i")

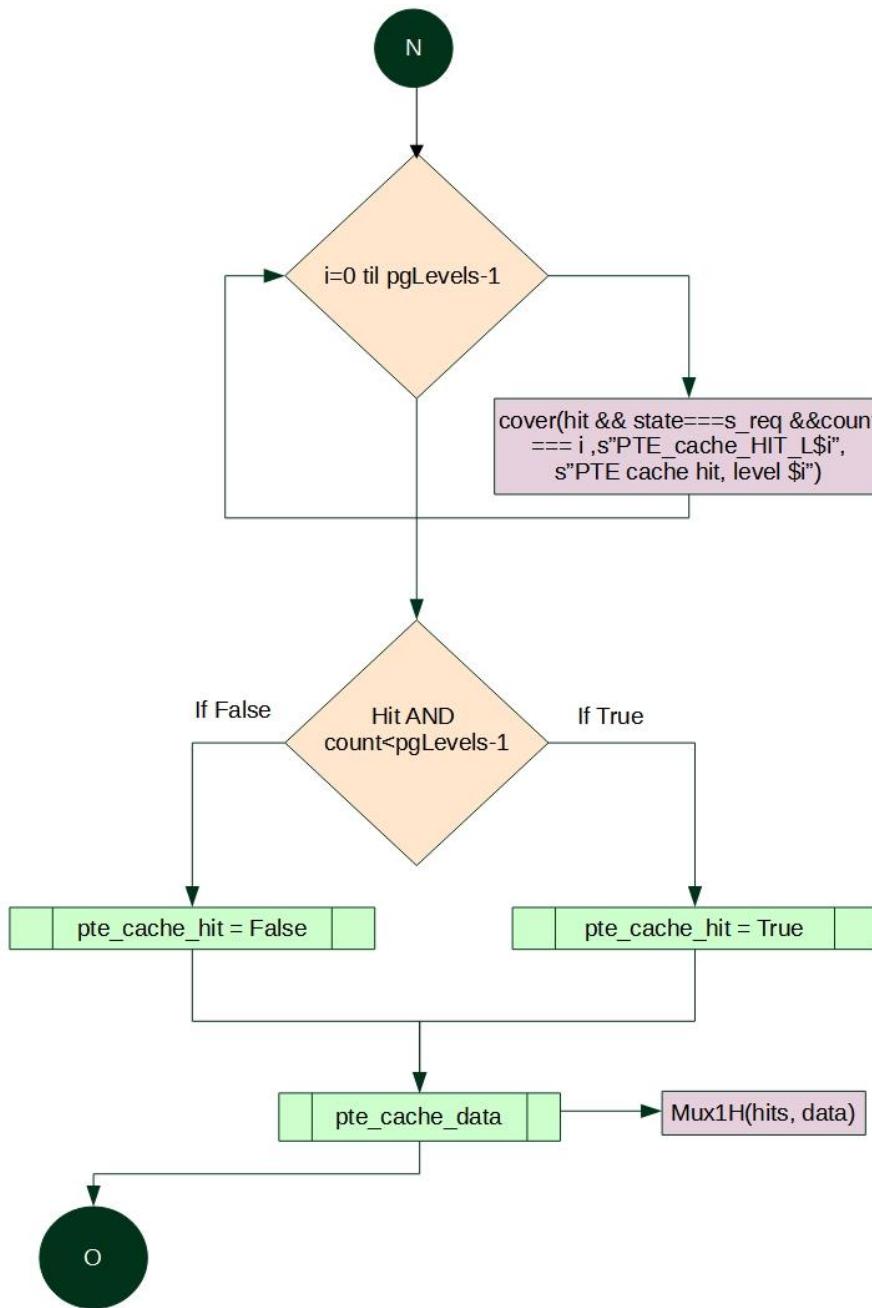
    (hit && count < pgLevels-1, Mux1H(hits, data))
}

```

— explanation —

loop iterated from 0 to pgLevels-1 calling ccover method of PTW class

pte_cache_hit gets true if hit is true and count is smaller than pgLevels-1,
pte_cache_data gets the value on index hits of data converted into binary.



```

val l2_refill = RegNext(false.B)
io.dpath.perf.l2miss := false
val (l2_hit, l2_error, l2_pte, l2_tlb_ram)

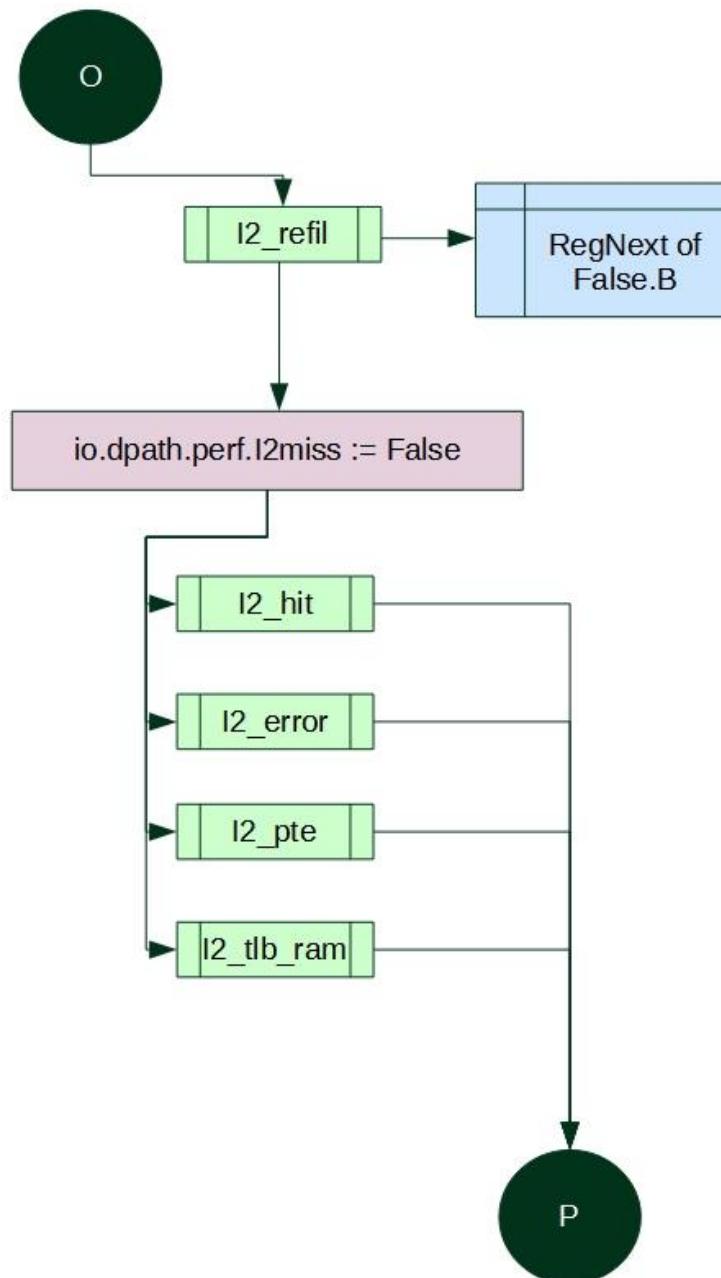
```

explanation

l2_refill has next Reg value as false in hardware

l2_refill_wire wired with 0

l2_hit, l2_error, l2_pte, and l2_tlb_ram are initialized inside a code block.



```

val (l2_hit, l2_error, l2_pte, l2_tlb_ram) = if (coreParams.nL2TLBEntries
== 0) (false.B, false.B, Wire(new PTE), None) else {
    val code = new ParityCode
    require(isPow2(coreParams.nL2TLBEntries))
    val idxBits = log2Ceil(coreParams.nL2TLBEntries)
    val (ram, omSRAM) = DescribedSRAM(
        name = "l2_tlb_ram",
        desc = "L2 TLB",
        size = coreParams.nL2TLBEntries,
        data = UInt(width = code.width(new L2TLBEntry().getWidth))
    )
}

```

— explanation —

when L2 entries are zero then L2 didn't hit, no error, l2_pte is fresh PTE object, and has no RAM.

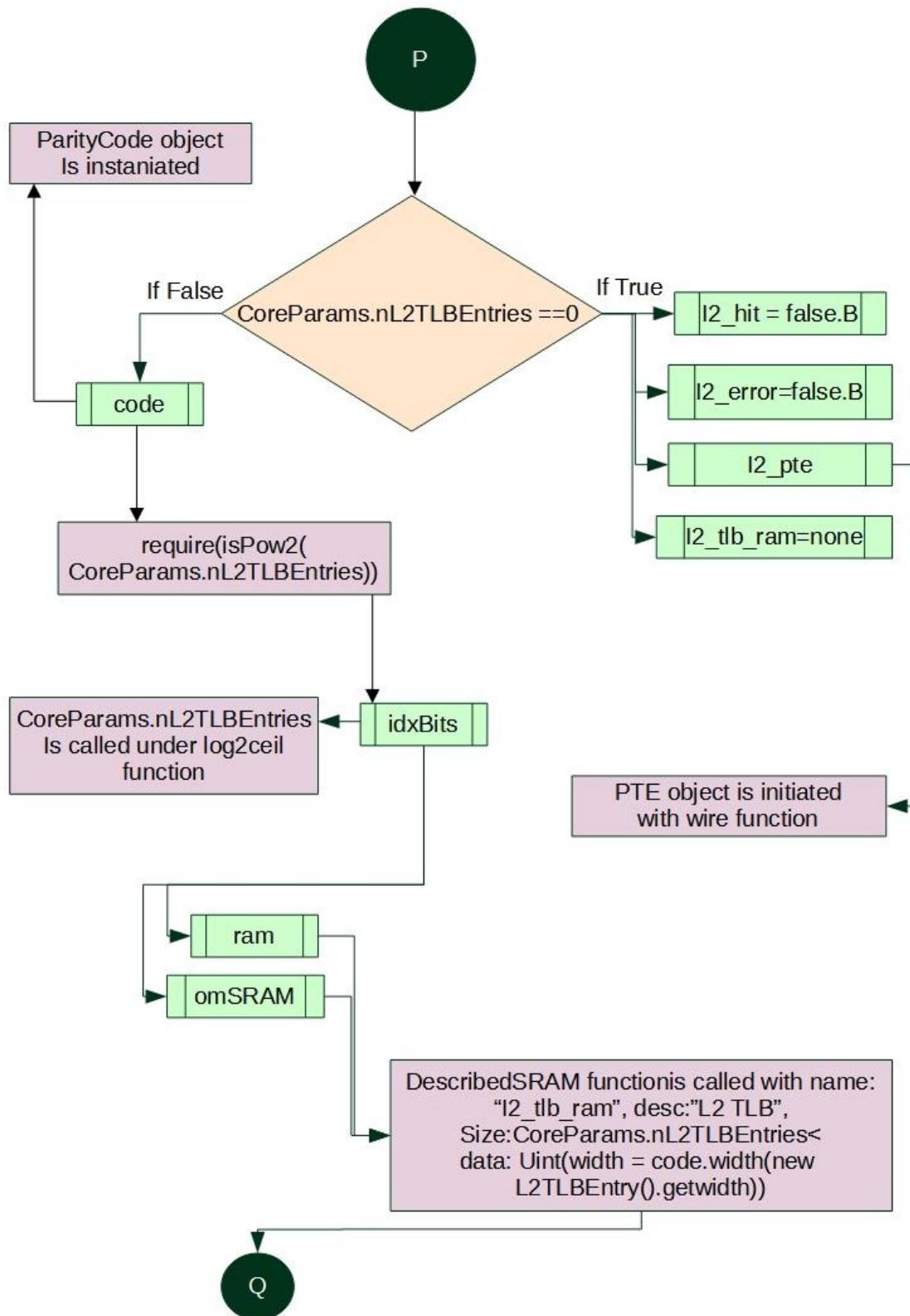
ParityCode object is instantiated in code variable.

Else if L2 entries are not zero then,

require takes a condition isPow2, means value Powered to 2, if given false then it'll return an exception.

idxBits has log2Ceil of length of TLB Entries

ram and omSRAM defined with DescribedSRAM() with the given values.



```

val g = Reg(UInt(width = coreParams.nL2TLBEntries))
val valid = RegInit(UInt(0, coreParams.nL2TLBEntries))
val (r_tag, r_idx) = Split(r_req.addr, idxBits)

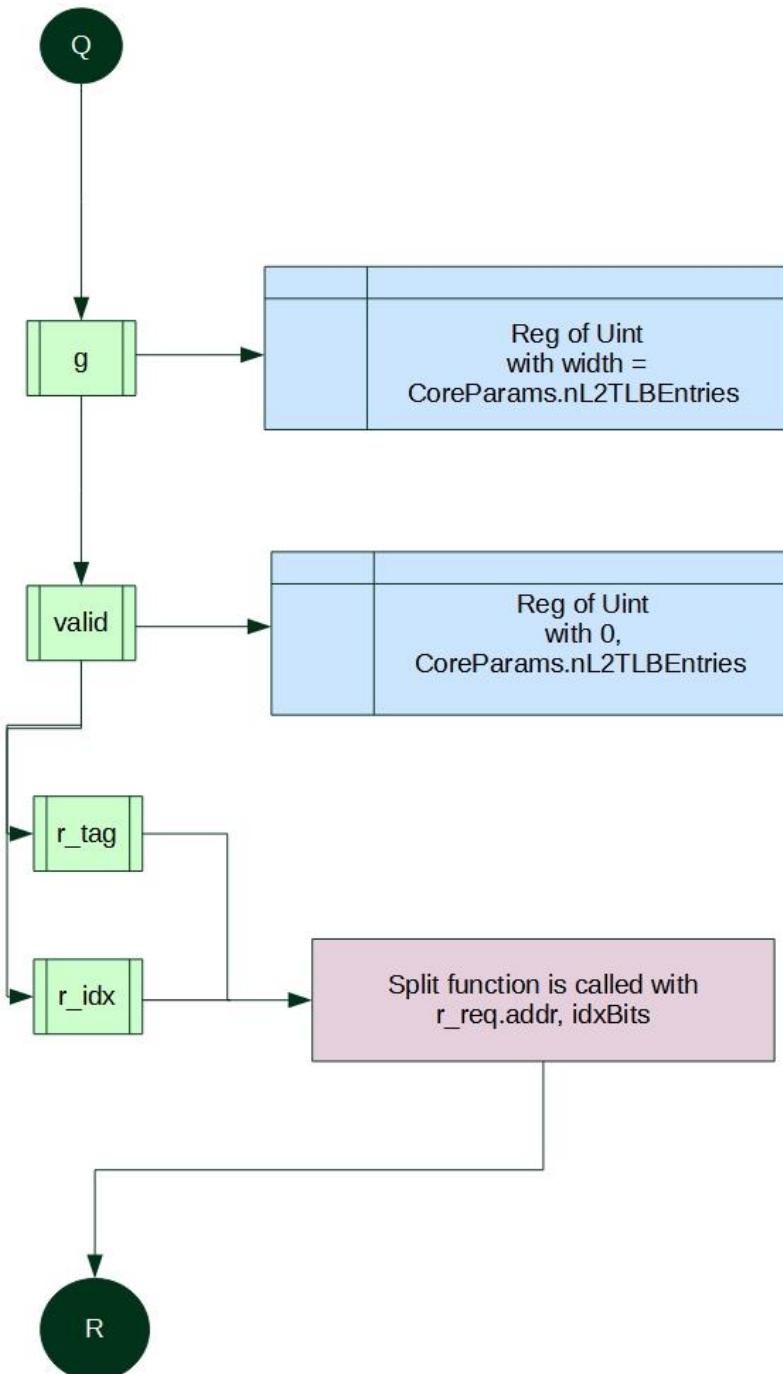
```

explanation

g has Reg defined with UInt of width TLB Entries

valid has RegInit of UInt with 0 and nL2TLBEntries

r_tag and r_idx has r_req.addr, splitted on basis of number of TLB Entries; before part is r_tag, after part is r_idx.



```

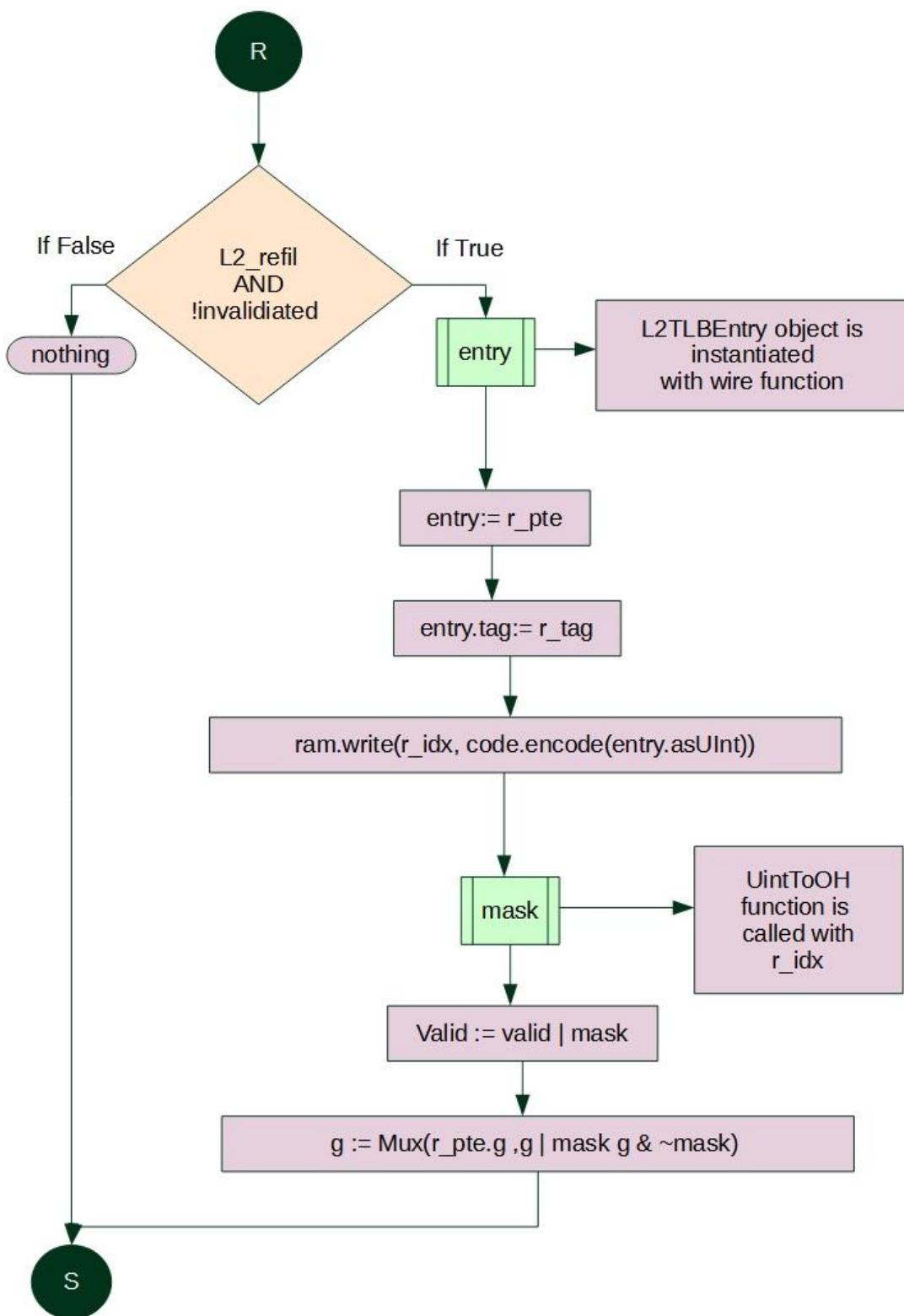
when (l2_refill && !invalidated) {
    val entry = Wire(new L2TLBEntry)
    entry := r_pte
    entry.tag := r_tag
    ram.write(r_idx, code.encode(entry.asUInt))

    val mask = UIntToOH(r_idx)
    valid := valid | mask
    g := Mux(r_pte.g, g | mask, g & ~mask)
}

```

explanation

when l2_refill is true and invalidated is false,
 L2TLBEnrty object is hardware wired in entry variable
 entry gets PTE by r_pte
 entry tag gets r_tag
 writes on ram r_idx line, the whole entry
 mask has r_idx in binary by UIntToOH
 valid bits are OR'ed with mask bits
 g is wired to MUX in which r_pte.g decides whther g gets Or'ed with mask bits OR g
 gets AND'ed with mask inverted bits.

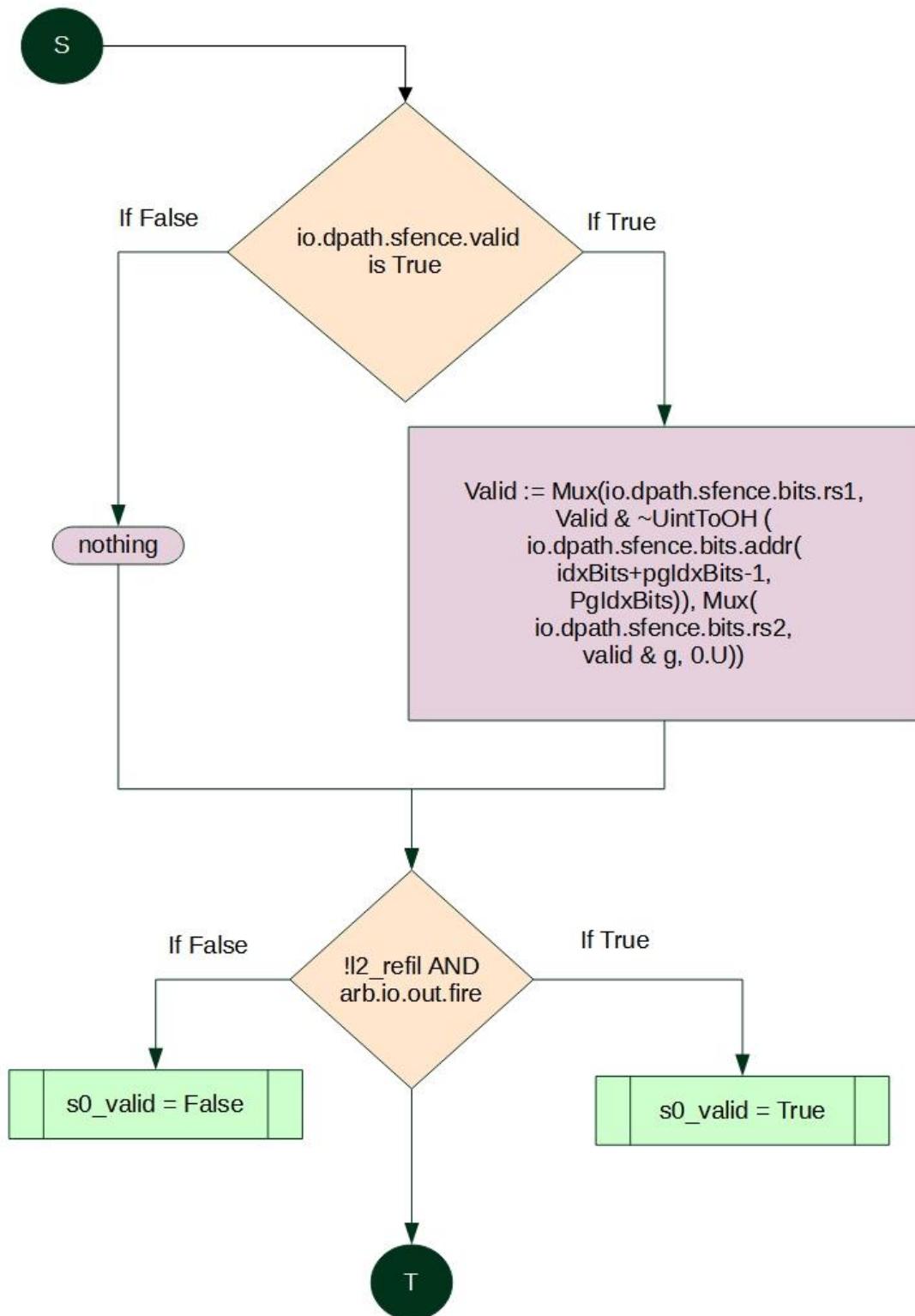


ROCKET-CHIP Micro Architecture Specification Document

```
when (io.dpath.sfence.valid) {  
    valid := Mux(io.dpath.sfence.bits.rs1, valid &  
~UIntToOH(io.dpath.sfence.bits.addr(idxBits+pgIdxBits-1, pgIdxBits)),  
        Mux(io.dpath.sfence.bits.rs2, valid & g, 0.U))  
}  
val s0_valid = !l2_refill && arb.io.out.fire()
```

— explanation —

```
if sfence is valid  
valid gets muxed like this  
s0_valid is true when l2_refill is false AND arb.io.out.fire() is true  
means L2 shouldn't refill
```



```

val s1_valid = RegNext(s0_valid && arb.io.out.bits.valid)
val s2_valid = RegNext(s1_valid)
val s1_rdata = ram.read(arb.io.out.bits.bits.addr(idxBits-1, 0),
s0_valid)
val s2_rdata = code.decode(RegEnable(s1_rdata, s1_valid))
val s2_valid_bit = RegEnable(valid(r_idx), s1_valid)
val s2_g = RegEnable(g(r_idx), s1_valid)

```

 explanation

s1_valid is RegNext with true when s0_valid is true AND arb.io.out.bits.valid brings true

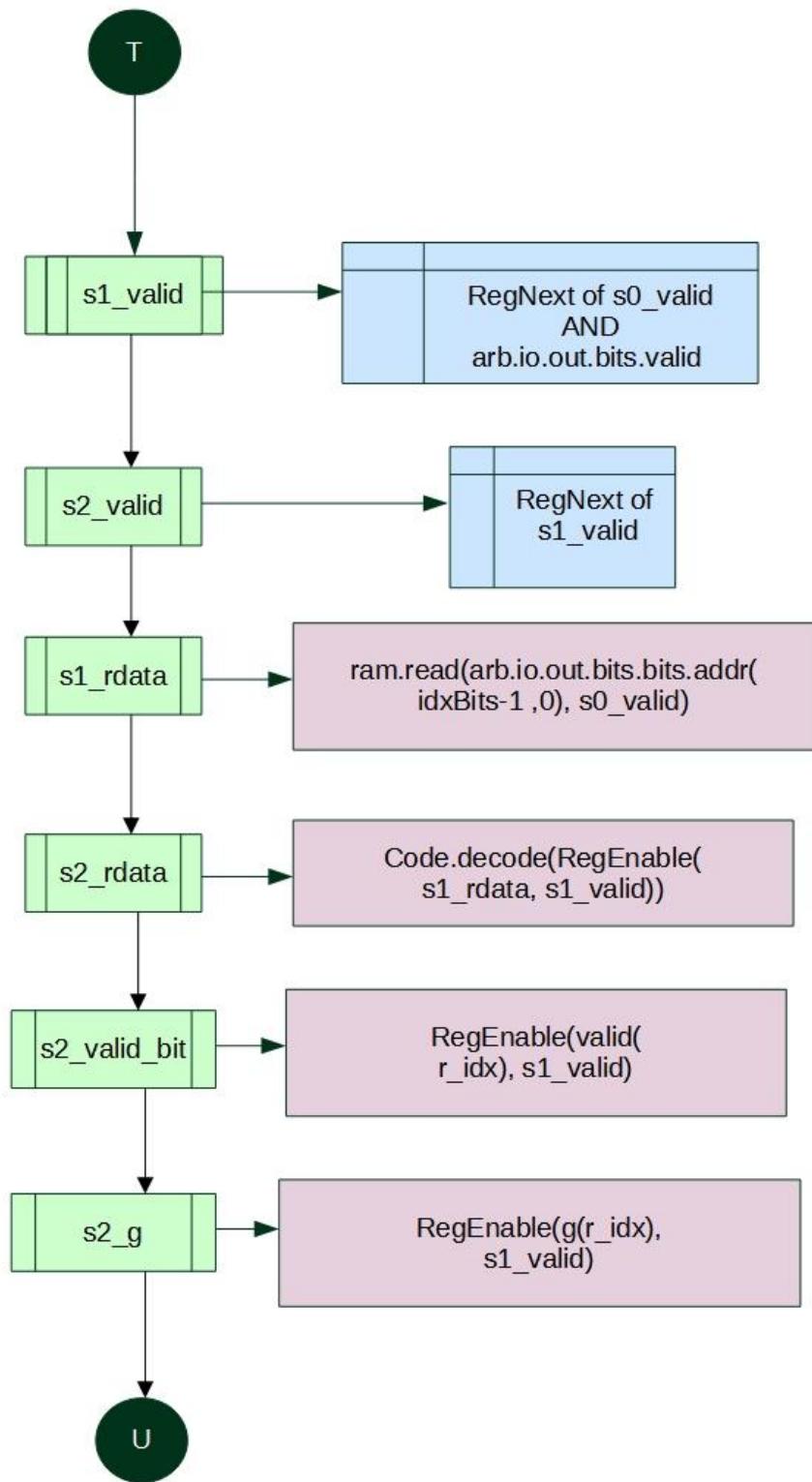
s2_valid is true in the next cycle if s1_valid in prev

s1_rdata had data read from RAM from arb.io.out.bits.bits.addr on idxBits-1, to 0, on index s0_valid

s2_rdata has decoded code from RegEnable of s1_rdata if s1_valid is true

s2_valid_bit also has a RegEnable of valid(r_idx) if s1_valid is true

s2_g also has RegEnable of g(r_idx) if s1_valid is true



```
when (s2_valid && s2_valid_bit && s2_rdata.error) { valid := 0.U }

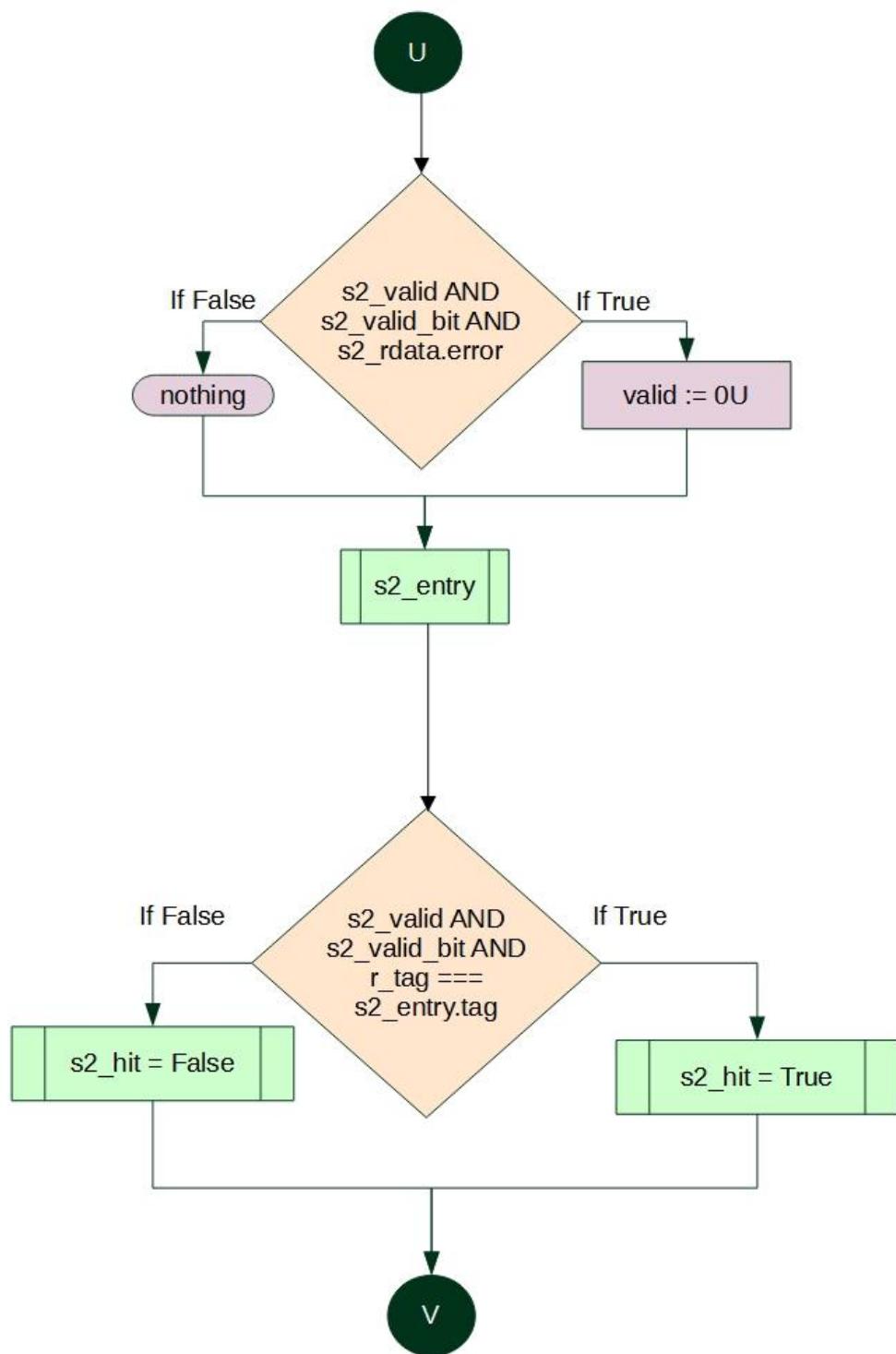
val s2_entry = s2_rdata.uncorrected.asTypeOf(new L2TLBEntry)
val s2_hit = s2_valid && s2_valid_bit && r_tag === s2_entry.tag
```

— explanation —

when s2_valid is true (last cycle's s1_Valid was true) AND s2_valid_bit (which is enabled by this cycle's s1_valid) AND s2.rdata.error is True so valid will be wired 0

s2_entry has bits of s2_rdata in the format provided by L2TLBEntry; as a L2TLBEntry object hardware wired

s2_hit is true if s2_valid is True AND s2_valid_bit is true AND r_tag === s2_entry.tag



```
    io.dpath.perf.l2miss := s2_valid && !(s2_valid_bit && r_tag ===  
s2_entry.tag)  
  
    val s2_pte = Wire(new PTE)  
    s2_pte := s2_entry  
    s2_pte.g := s2_g  
    s2_pte.v := true  
  
    ccover(s2_hit, "L2_TLB_HIT", "L2 TLB hit")
```

— explanation —

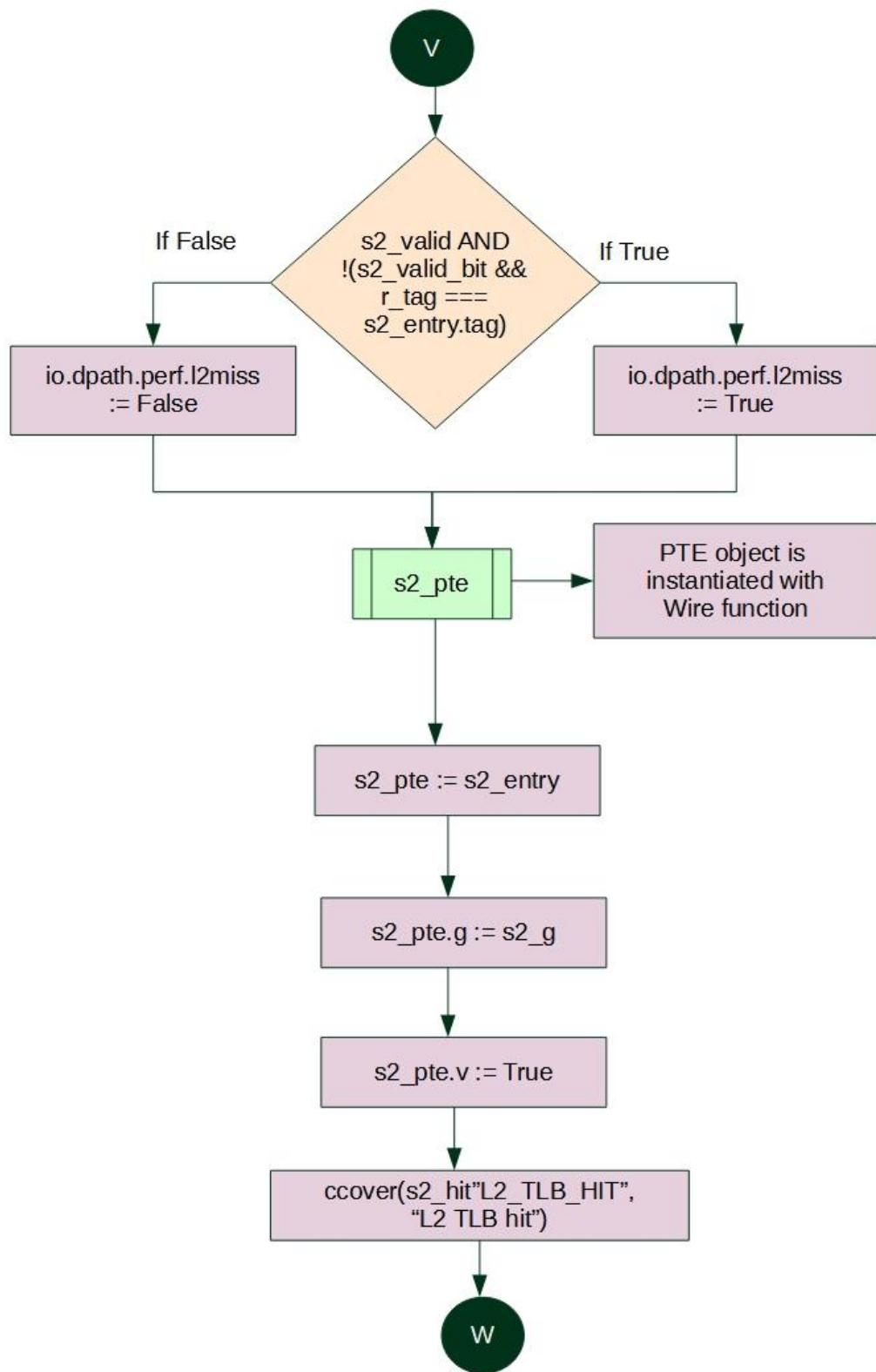
dpath will get l2miss status when s2_valid is true AND s2_valid_bit is fasle AND r_tag != s2_entry.tag

s2_pte has PTE object hardware wired

s2_pte is wired with s2_entry bits

s2_pte.g is wired to s2_g

s2_pte.v is now true

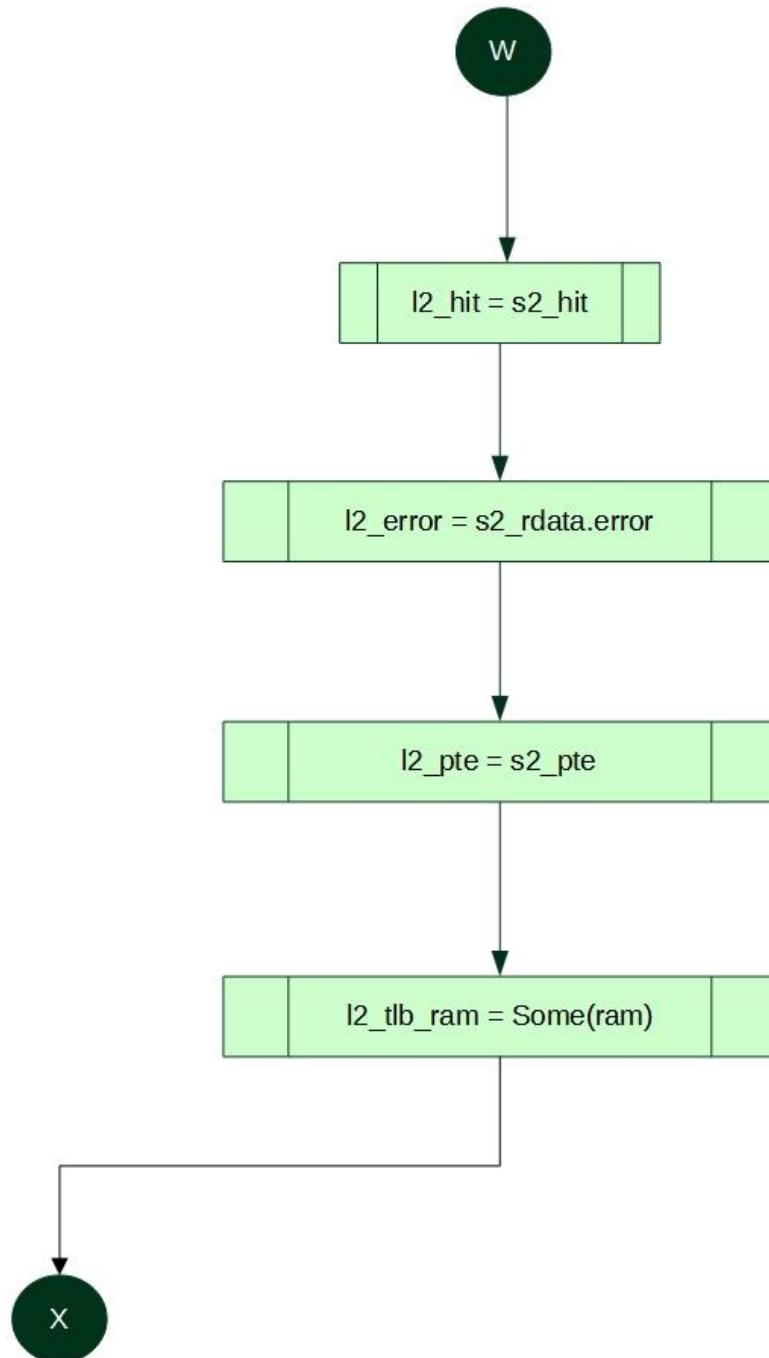


```
(s2_hit, s2_rdata.error, s2_pte, Some(ram))
}
```

— explanation —

I2_hit has s2_hit, I2_error has s2_error, I2_pte has s2_pte, I2_ram has some(ram)

Some represents the existing values of the ram.

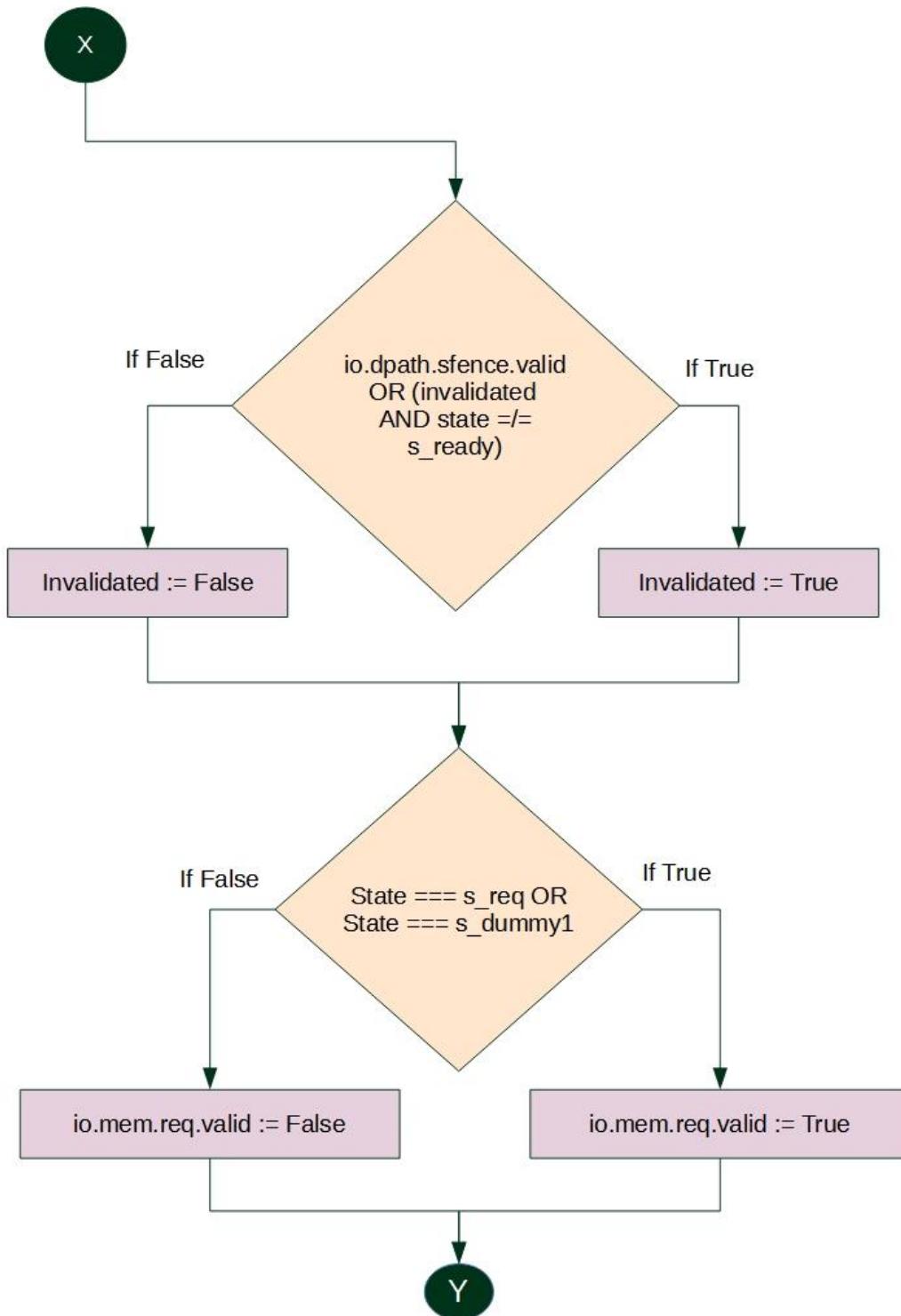


```
invalidated := io.dpath.sfence.valid || (invalidated && state == s_ready)
io.mem.req.valid := state == s_req || state == s_dummy1
```

explanation

invalidated is wired to validity of sfence in dpath OR invalidated is true AND state == s_ready

mem_req_valid is true if state == s_req OR state == s_dummy1



```
io.mem.req.bits.phys := Bool(true)
io.mem.req.bits.cmd  := M_XRD
io.mem.req.bits.size := log2Ceil(xLen/8)
io.mem.req.bits.signed := false
io.mem.req.bits.addr := pte_addr
io.mem.req.bits.dprv := PRV.S.U
```

— explanation —

bitsphys is hardware True

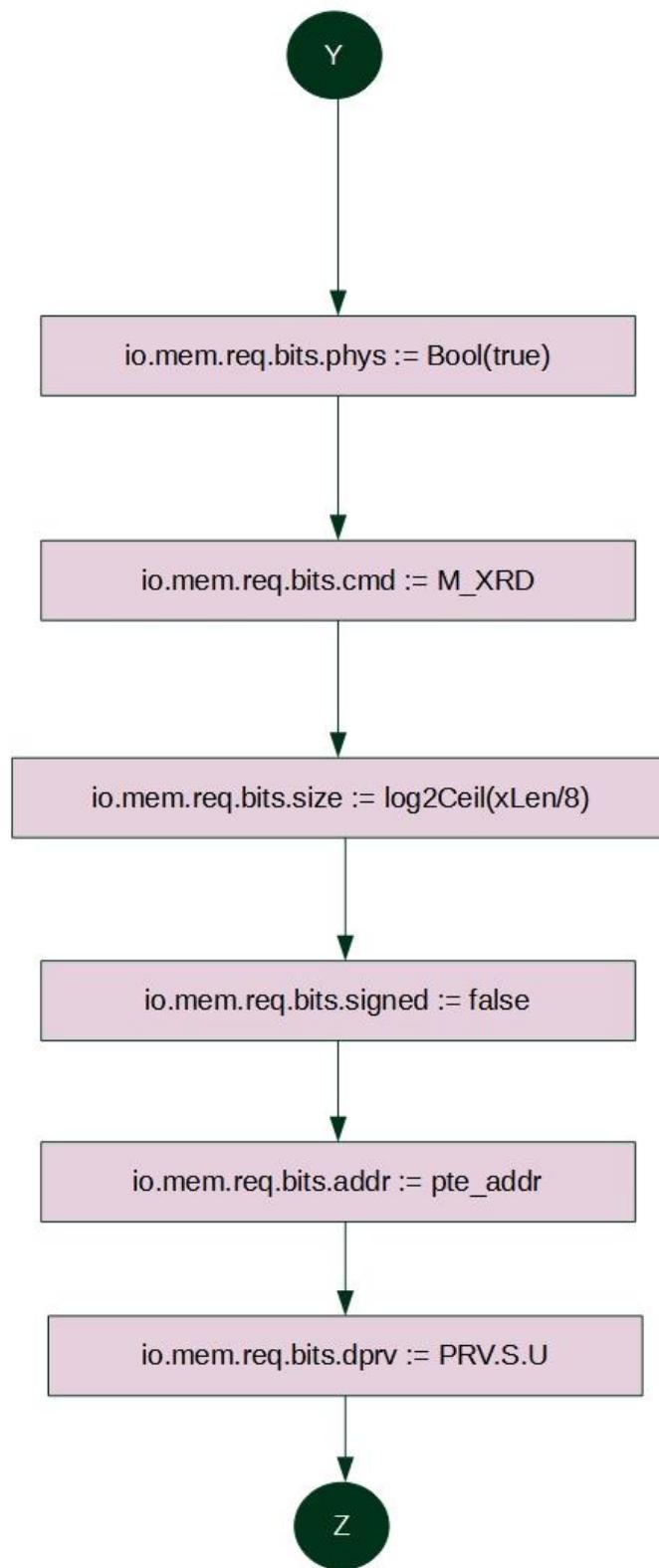
MXRD is 0 so cmd is wired 0

size has log2 Ceiled of xLen/8

signed is false bcuz we provide unsigned bits

pte_addr is the addr

PTW accesses are S-mode by definition



```
io.mem.s1_kill := l2_hit || state /= s_wait1
io.mem.s2_kill := Bool(false)
val pageGranularityPMPs = pmpGranularity >= (1 << pgIdxBits)
val pmaPgLevelHomogeneous
```

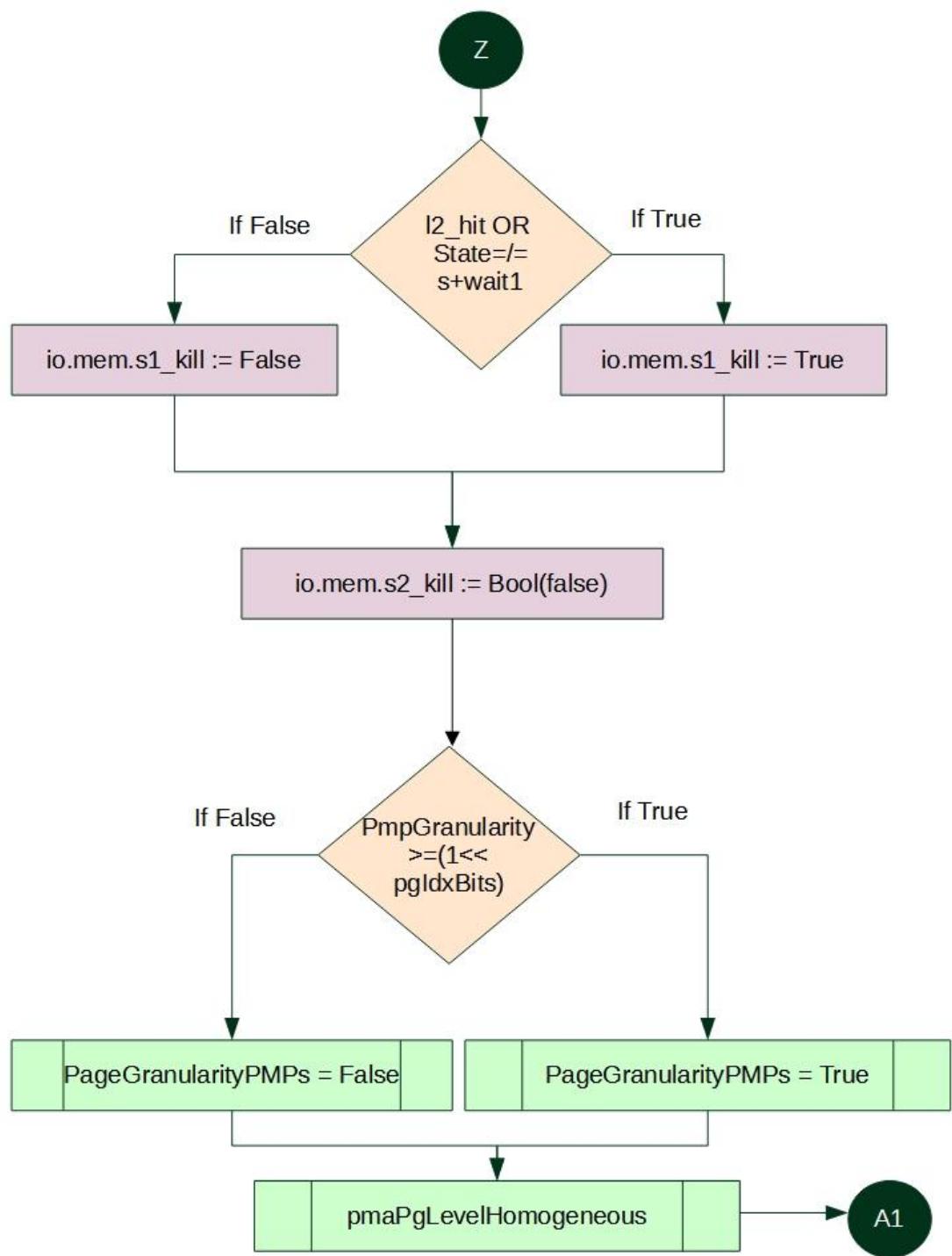
explanation

s1_kill is true if l2_hit is true and state != s_ready

s2_kill is false in hardware

pmpG.. is 4 >= 1 left shifted to pgIdxBits saved in pageGranularityPMPs

pmaPgLevelHomogeneous is defined



```

val pmaPgLevelHomogeneous = (0 until pgLevels) map { i =>
    val pgSize = BigInt(1) << (pgIdxBits + ((pgLevels - 1 - i) * pgLevelBits))
    if (pageGranularityPMPs && i == pgLevels - 1) {
        require(TLBPageLookup.homogeneous(edge.manager.managers, pgSize),
               s"All memory regions must be $pgSize-byte aligned")
        true.B
    } else {
        TLBPageLookup(edge.manager.managers, xLen, p(CacheBlockBytes),
                      pgSize)(pte_addr).homogeneous
    }
}

```

— explanation —

pmaPgLevelHomogeneous has iteration from 0 to pgLevels-1 mapped into,

pgSize is 1 left shifted with (pgIdxBits + (pgLevels-1-i * pgLevelBits)

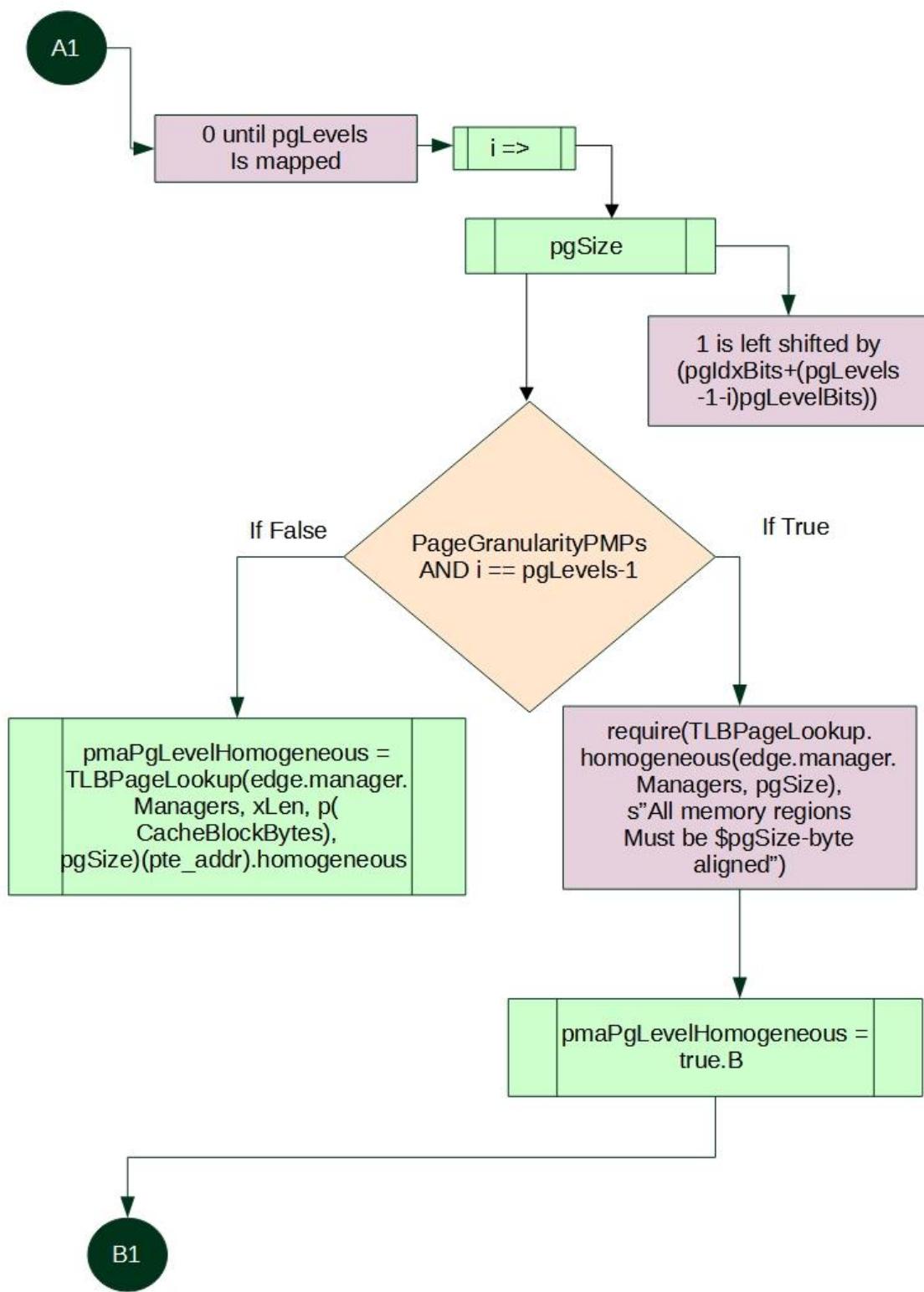
if pageGranularityPMPs is true and i == pgLevels-1 (last iteration)

require will be called

Homogeneity will be true in hw, iffffff require doesn't bring any exceptions

else.. in all other iterations

TLBPageLookup will be called



ROCKET-CHIP Micro Architecture Specification Document

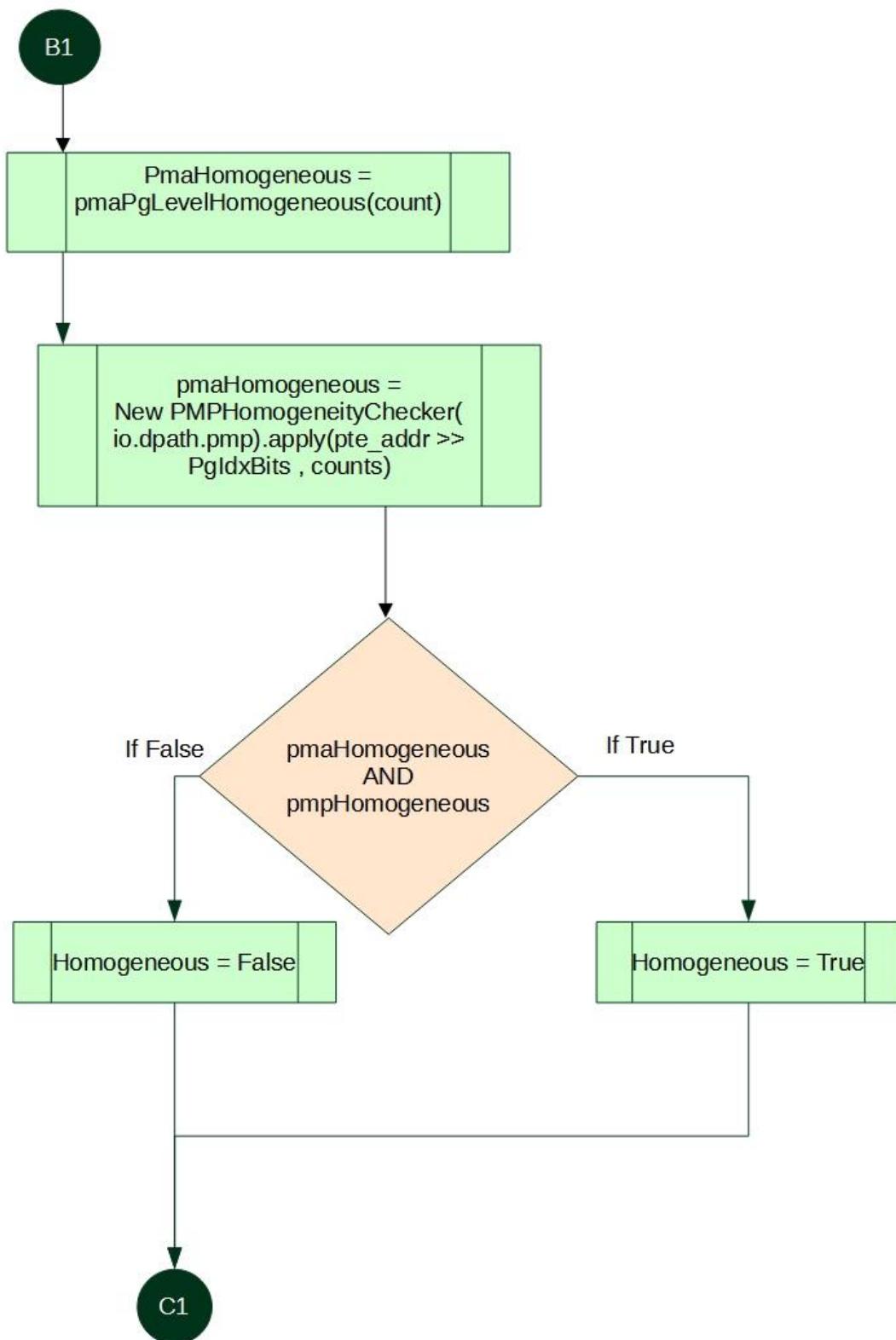
```
val pmaHomogeneous = pmaPgLevelHomogeneous(count)
val pmpHomogeneous = new
PMPHomogeneityChecker(io.dpath.pmp).apply(pte_addr >> pgIdxBits <<
pgIdxBits, count)
val homogeneous = pmaHomogeneous && pmpHomogeneous
```

explanation

count index of pmaPgLevelHomogenous in pmaHomogeneous

pmpHomogenous -- PMPhomogenityChecker called with original and suzilary constructor

homogenous -- AND of pmaHomogenous and pmpHomogenous



```

for (i <- 0 until io.requestor.size) {
    io.requestor(i).resp.valid := resp_valid(i)
    io.requestor(i).resp.bits.ae := resp_ae
    io.requestor(i).resp.bits.pte := r_pte
    io.requestor(i).resp.bits.level := count
    io.requestor(i).resp.bits.homogeneous := homogeneous || pageGranularityPMPs
    io.requestor(i).resp.bits.fragmented_superpage := resp_fragmented_superpage && pageGranularityPMPs
    io.requestor(i).ptbr := io.dpath.ptbr
    io.requestor(i).customCSRs := io.dpath.customCSRs
    io.requestor(i).status := io.dpath.status
    io.requestor(i).pmp := io.dpath.pmp
}

```

— explanation —

loop iterated for 0 to requestor size -1

requestor i resp valid has resp_valid of i

req resp bits ae has resp_ae

req resp bits pte has r_pte

req resp bits level has count - pgLevels -1

req resp bits homogenous has homogeneous true AND pageGranularityPMPs true

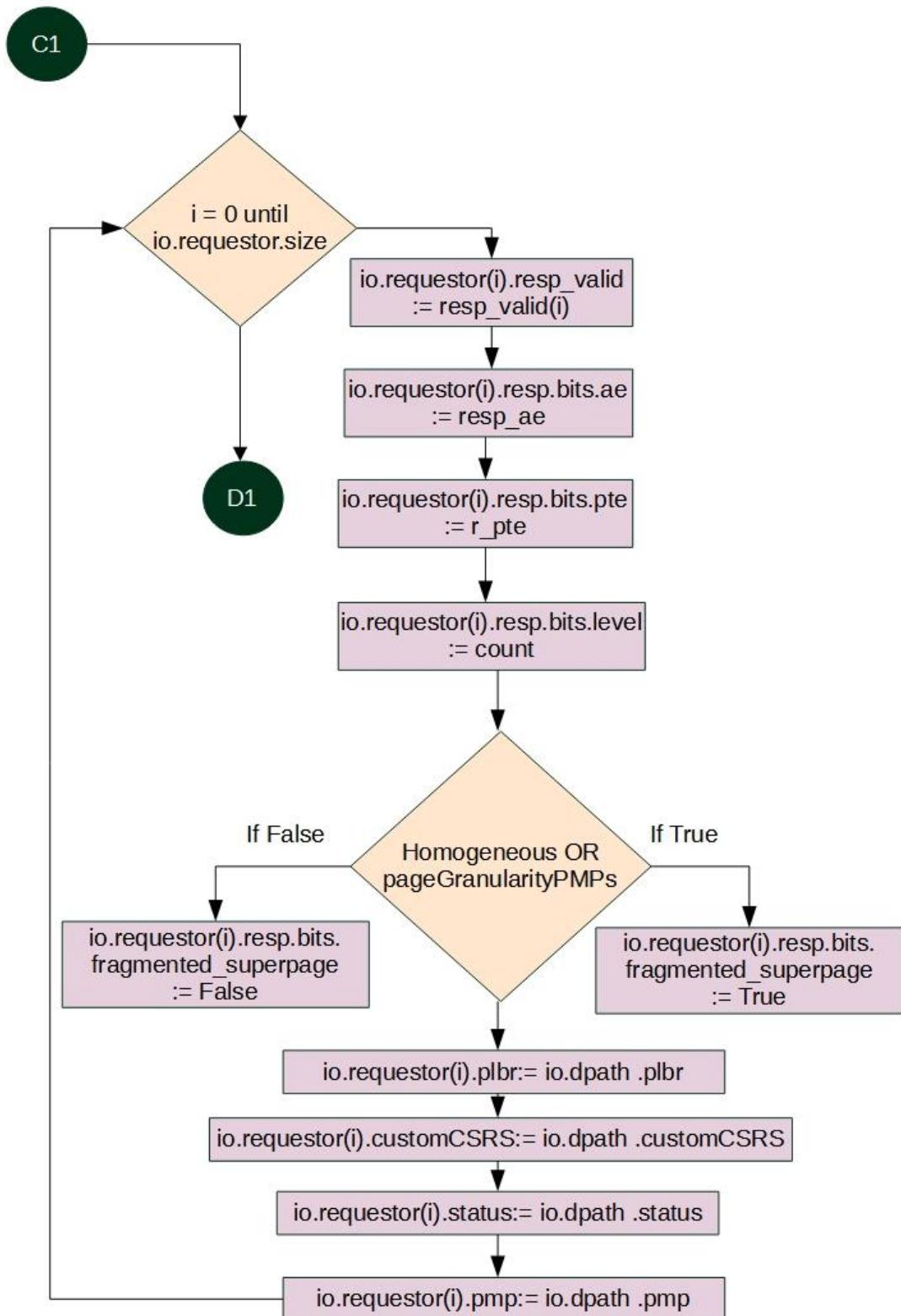
req resp bits fragmented_dsuperpage has resp_frag_superpg aND pageGRanularityPMPs

req ptbr has io.dpath.ptbr

req customCSRs has dpath customCSR

req status has dpath status

req pmp has dpath pmp

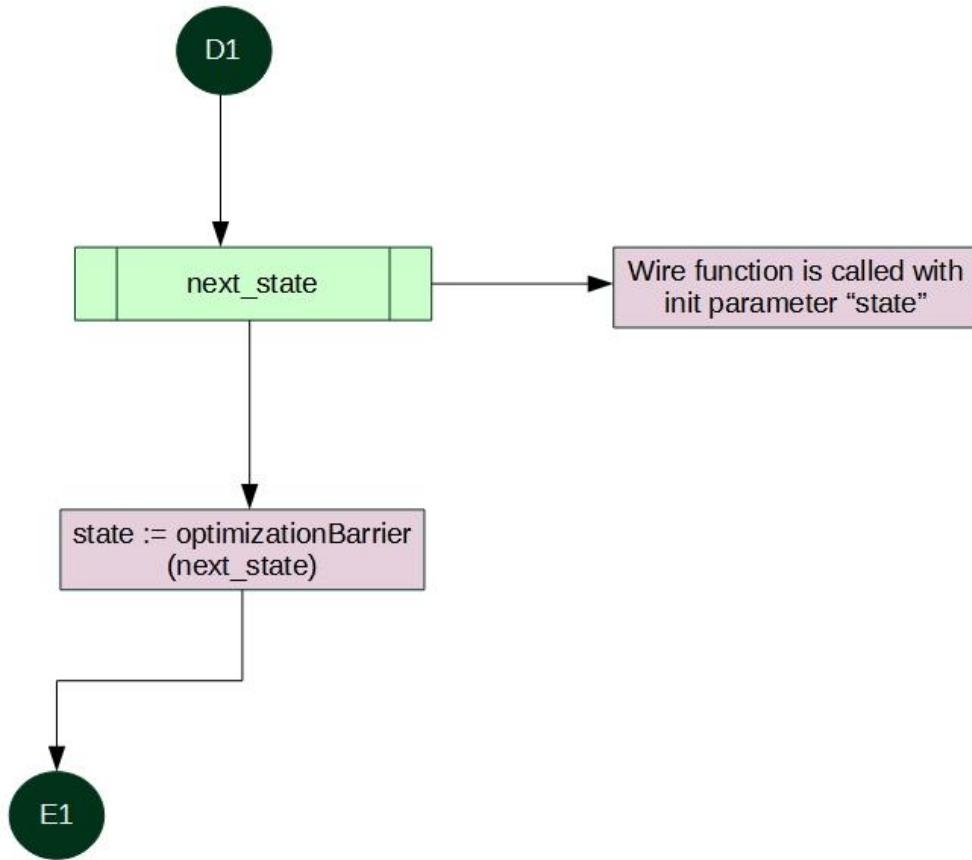


```
val next_state = Wire(init = state)
state := OptimizationBarrier(next_state)
```

explanation

next_state has Wire initiated with state

OptimizationBarrier has nextState



```

switch (state) {
    is (s_ready) {
        when (arb.io.out.fire()) {
            next_state := Mux(arb.io.out.bits.valid, s_req, s_ready)
        }
        count := pgLevels - minPgLevels - io.dpath.ptbr.additionalPgLevels
    }
    is (s_req) switch (state) {
        is (s_ready) {
            when (arb.io.out.fire()) {
                next_state := Mux(arb.io.out.bits.valid, s_req, s_ready)
            }
            count := pgLevels - minPgLevels - io.dpath.ptbr.additionalPgLevels
        }
    }
}
is (s_req)

```

— explanation —

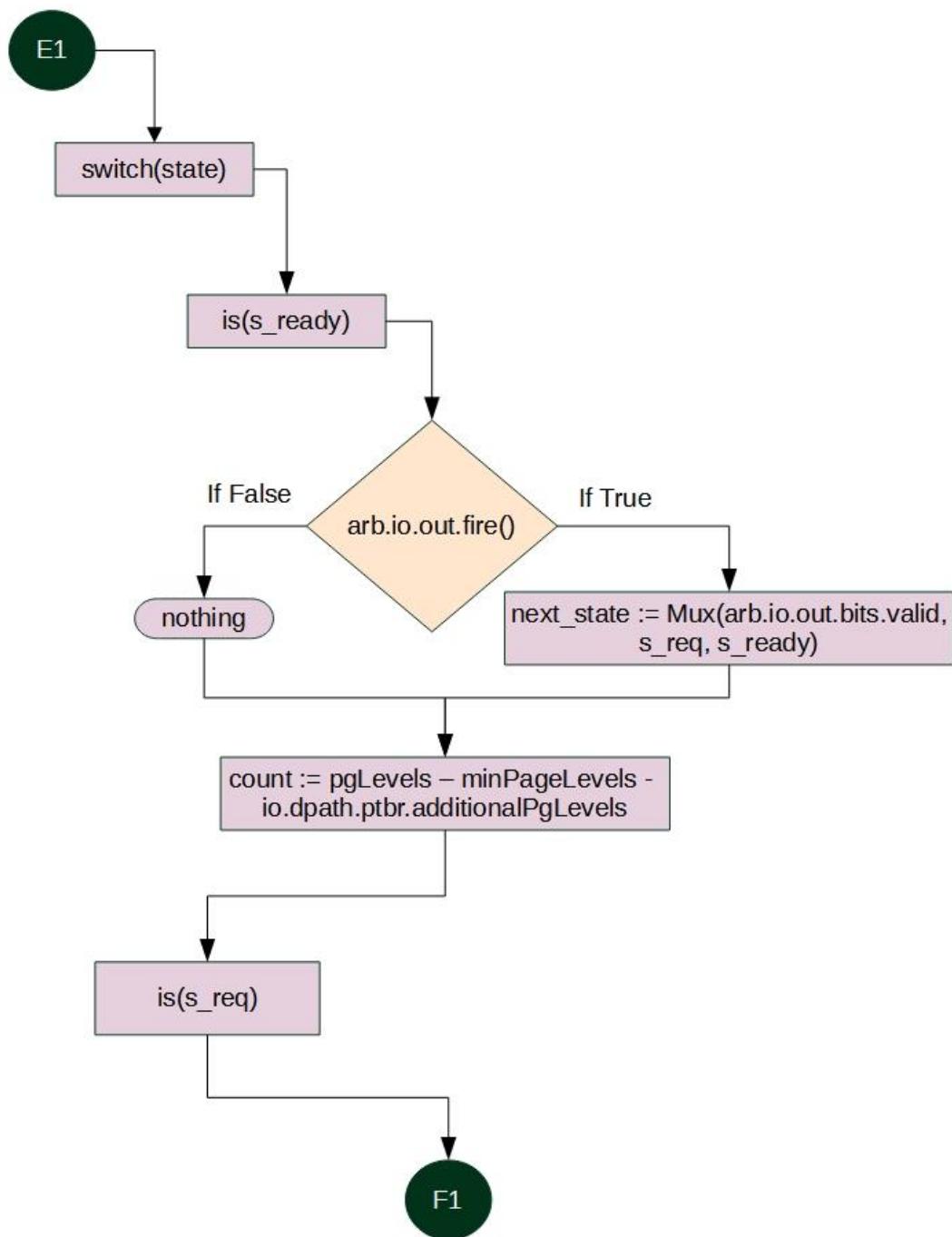
switch ---- state === conds

if state == s_ready , then....

when arb.io.out.fire() is true

next_state is wired to a Mux with sel - arb.io.out.bits.valid with s_req and s_ready

count will be changed to pgLEvels - minPgLevels - io.dpath/ptbr.additionalPgLEvels



```

is (s_req) {
    when (pte_cache_hit) {
        count := count + 1
    }.otherwise {
        next_state := Mux(io.mem.req.ready, s_wait1, s_req)
    }
}
is (s_wait1) {
    // This Mux is for the l2_error case; the l2_hit && !l2_error case is
    overriden below
    next_state := Mux(l2_hit, s_req, s_wait2)
}
is (s_wait2) {
    next_state := s_wait3
}

```

— explanation —

state == s_req

when pte_cache_hit is true

count will have +1

else

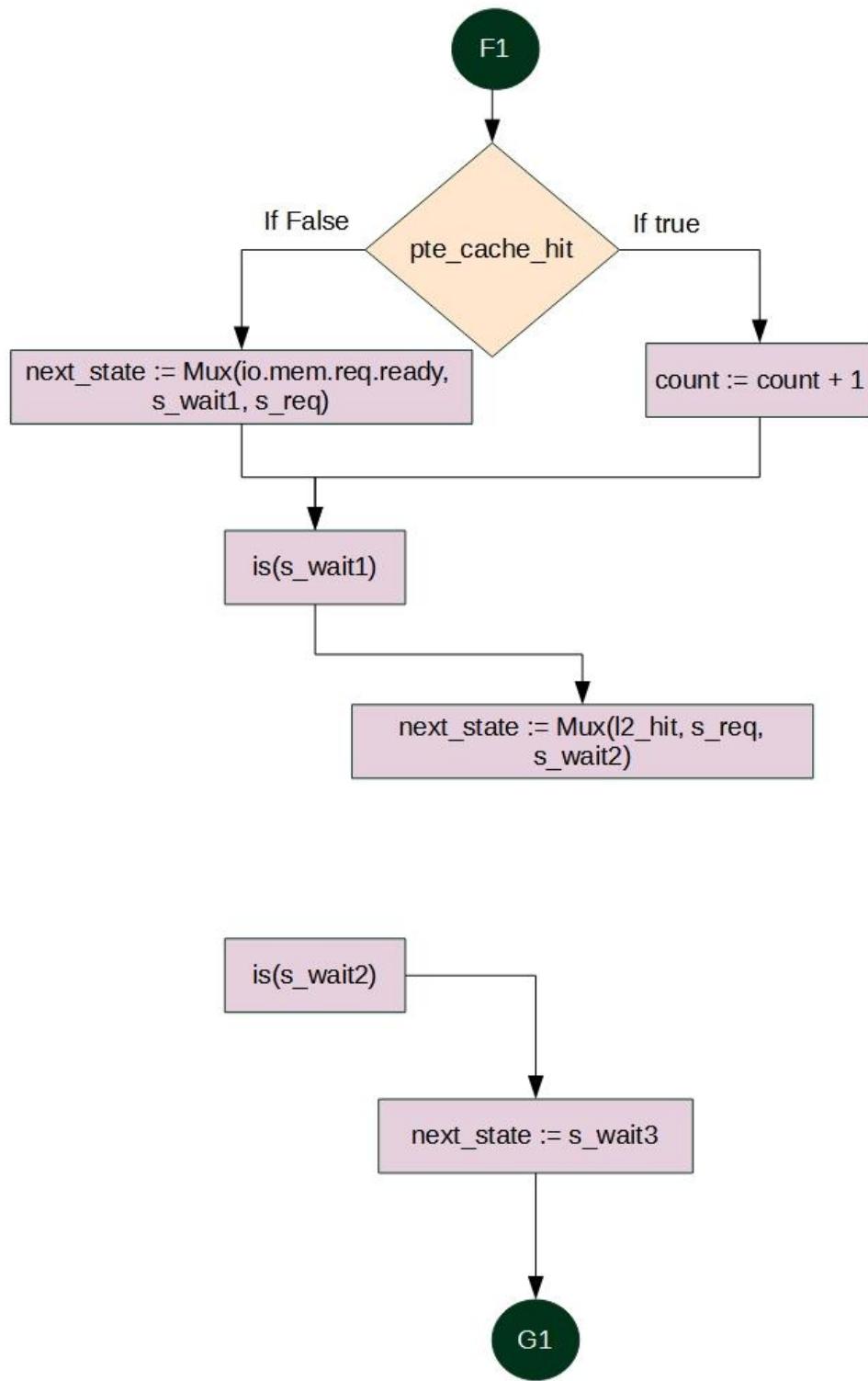
next_state will be Mux withs sel -- mem req ready with s_wait1 and s_req

state == s_wait1

next_state will be Mux with sel -- l2_hit with s_req and s_wait2

state == s_wait2

next_state will be s_wait3



```
when (io.mem.s2_xcpt.ae.ld) {  
    resp_ae := true  
    next_state := s_ready  
    resp_valid(r_req_dest) := true  
}  
}  
is (s_fragment_superpage) {  
    next_state := s_ready  
    resp_valid(r_req_dest)
```

explanation

when mem.s2_xcpt.ae.ld (tho idk wtf this is -_-) is true...

resp_ae is true

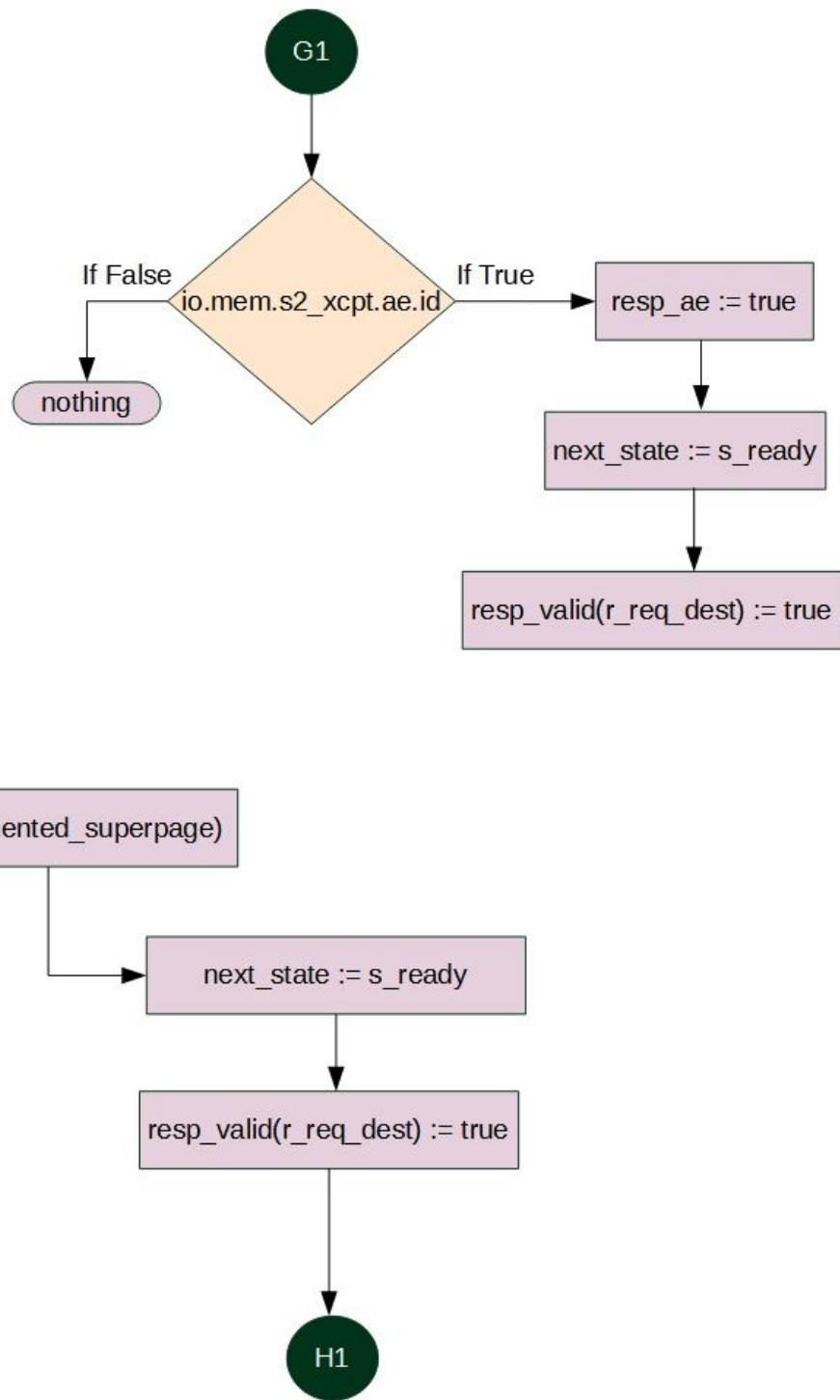
next_state will be s_ready

resp_valid index r_req_dest as true

state === s_fragmented_superpage

next_state will eb s_ready

resp_valid index r_req_dest will be true



```
resp_ae := false
    when (!homogeneous) {
        count := pgLevels-1
        resp_fragmented_superpage := true
    }
}
```

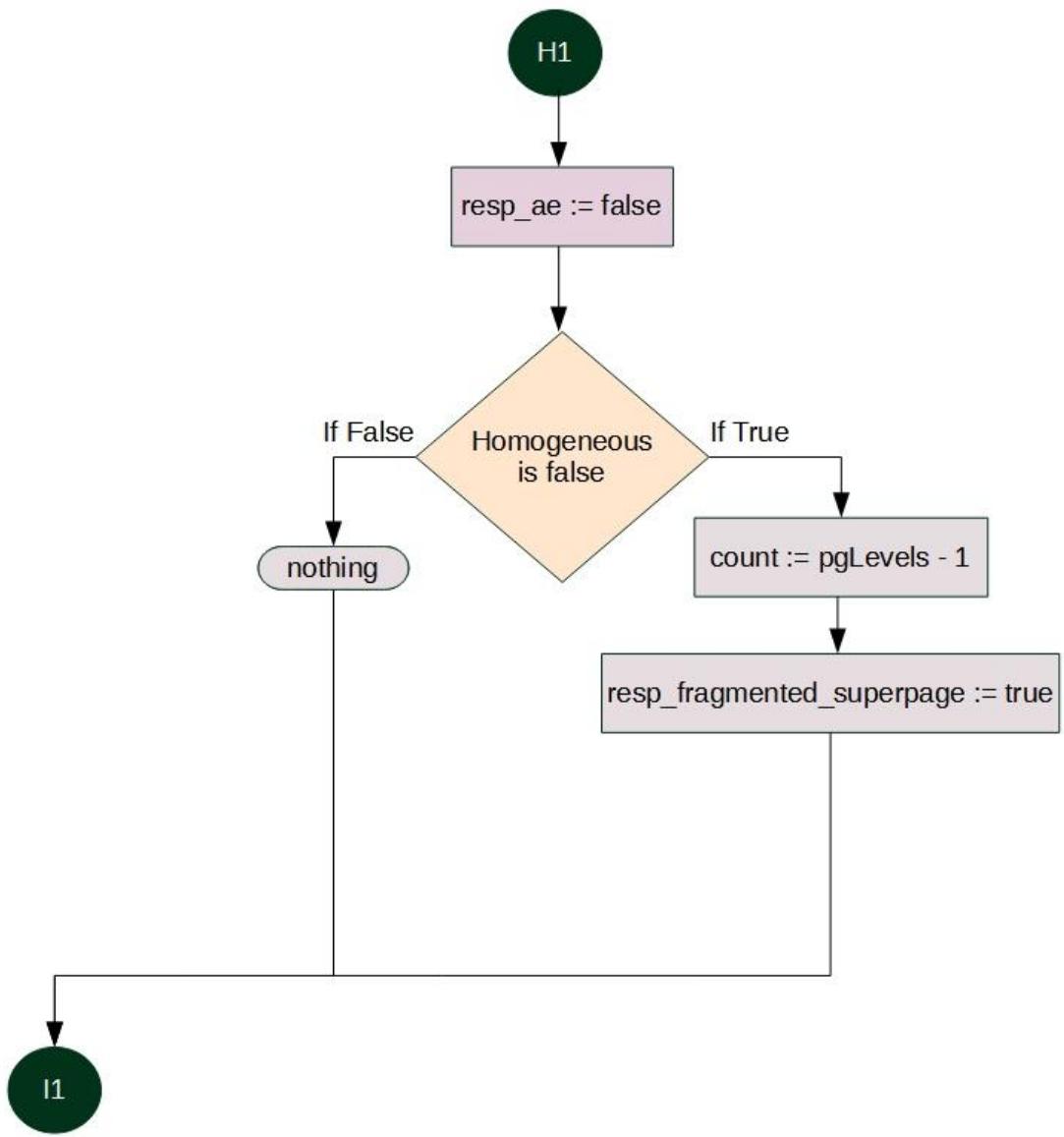
explanation

resp_ae is true

when homogenous is false

count will again be pgLevels -1

resp_fragmented_superpage will be true

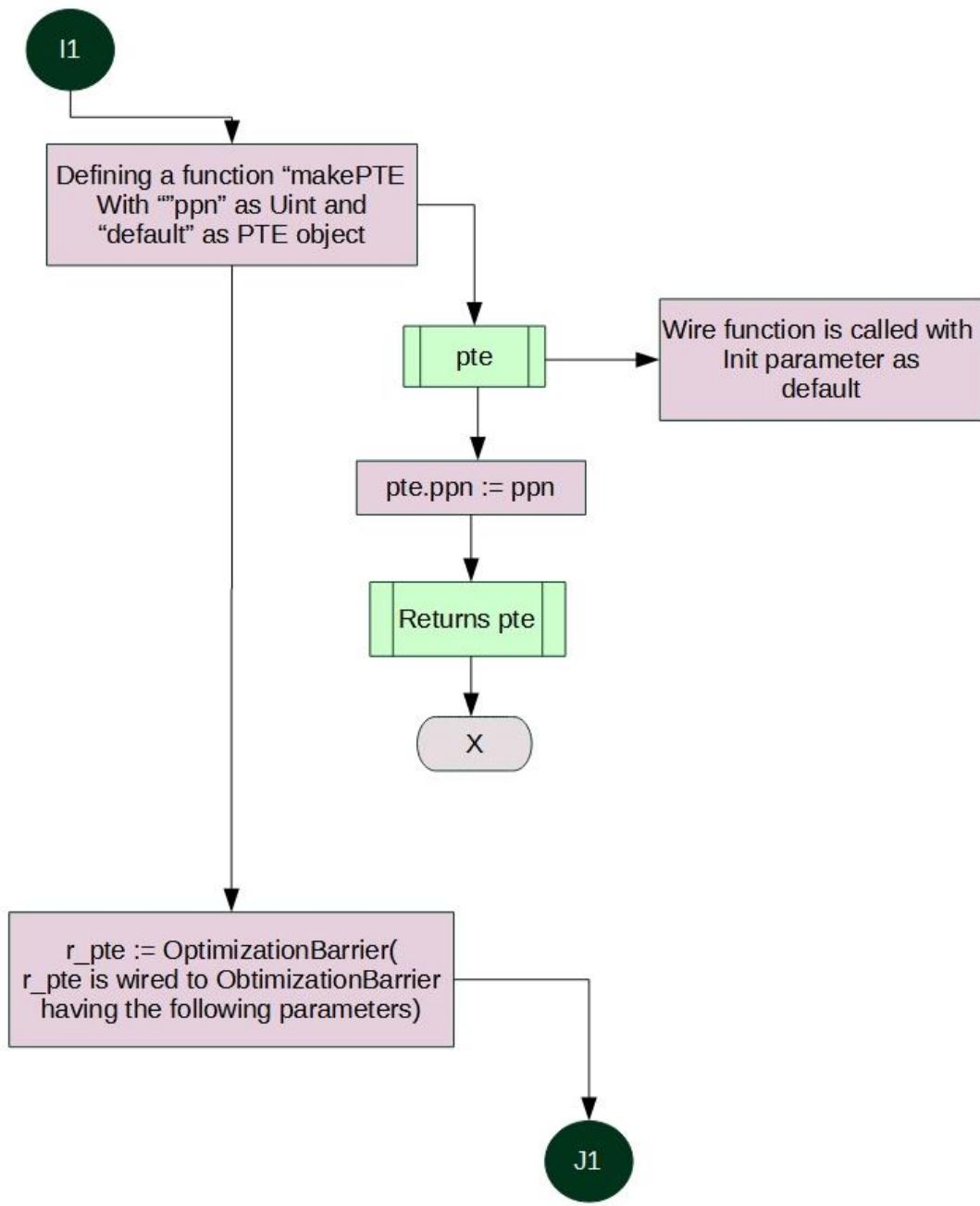


```
def makePTE(ppn: UInt, default: PTE) = {
    val pte = Wire(init = default)
    pte.ppn := ppn
    pte
}
r_pte := OptimizationBarrier
```

— explanation —

makePTE takes ppn as UInt and default PTE

r_pte has OptBar



```
Mux(mem_resp_valid, pte,
Mux(l2_hit && !l2_error, l2_pte,
Mux(state === s_fragment_superpage && !homogeneous,
makePTE(fragmented_superpage_ppn, r_pte),
Mux(state === s_req && pte_cache_hit, makePTE(pte_cache_data, l2_pte),
Mux(arb.io.out.fire(), makePTE(io.dpath.ptbr.ppn, r_pte),
r_pte))))))
```

— explanation —

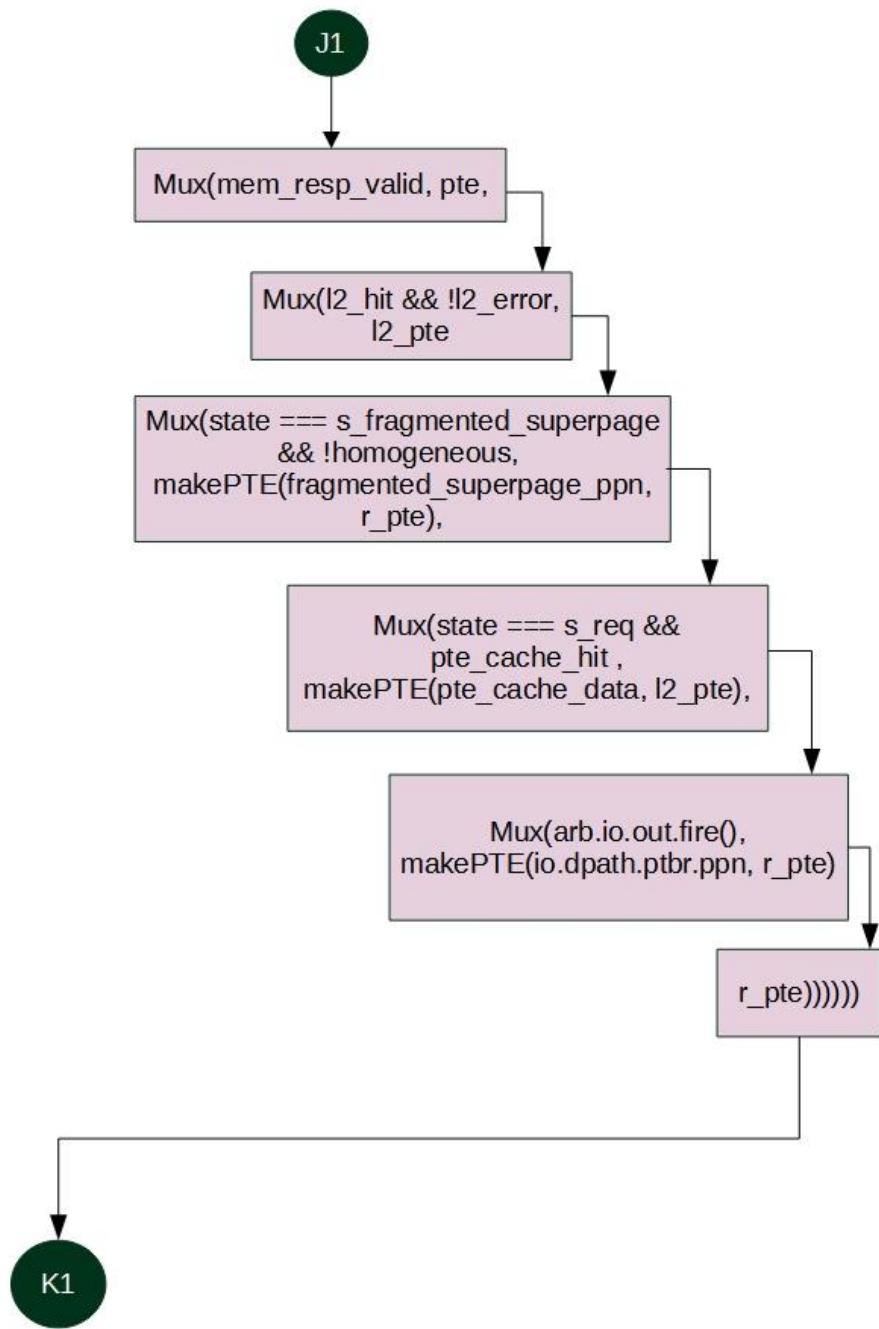
Mux1 has sel mem_resp_valid with pte and

Mux2 has sel l2_hit ANDed with l2_error false with l2_pte and

Mux3 has sel state === s_fragmented_superpage AND homogenous is false with
makePTE of superpagge ppn and r_pte and

Mux4 has sel state == s_req AND pte_cache_hit is true with makePTE of
pte_cache_data as ppn and l2_pte and

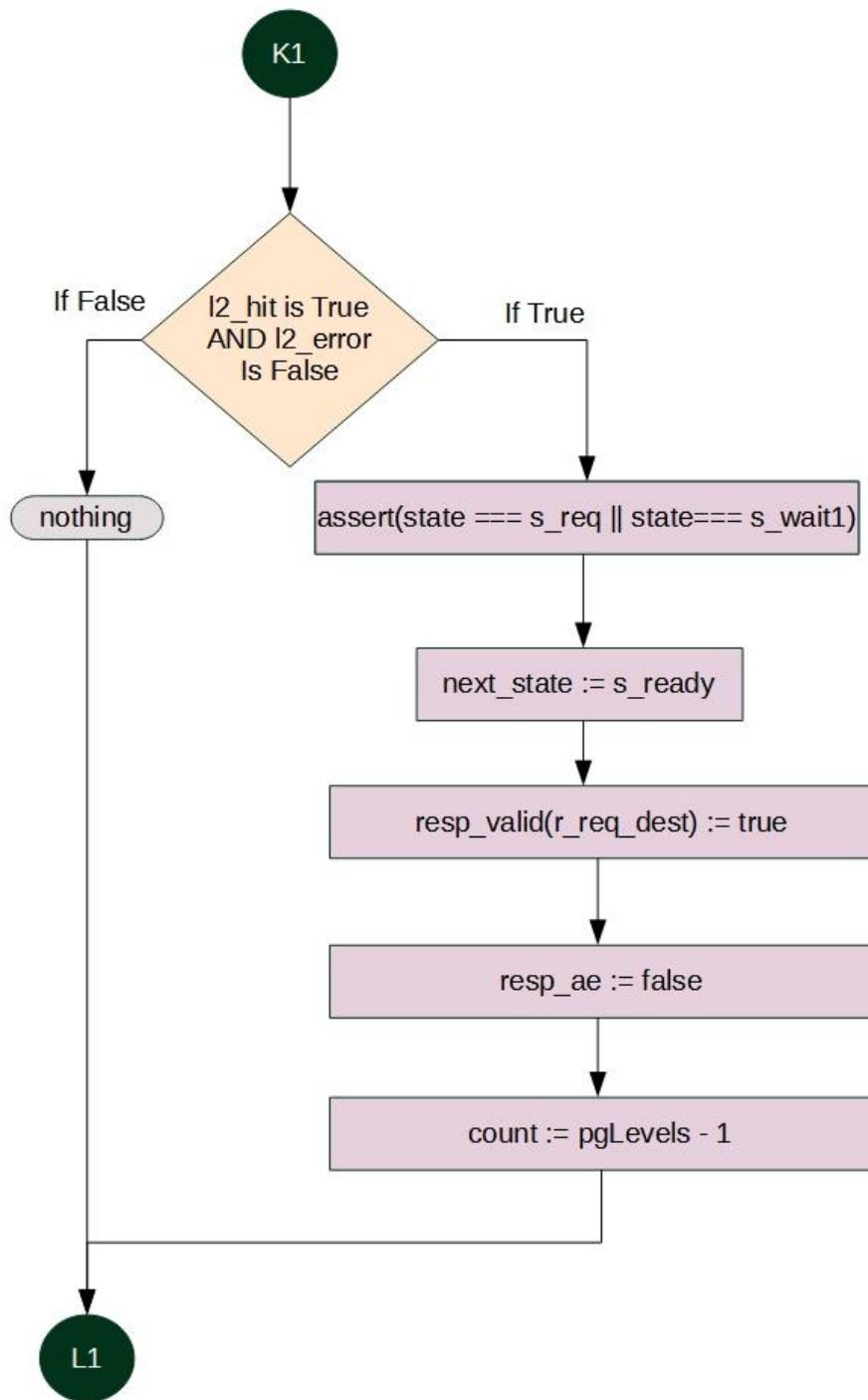
Mux5 has sel arb.io.out.fire() with makePTE of iodpath.ptbr.pnn as ppn and r_pte and
--- r_pte



```
when (l2_hit && !l2_error) {  
    assert(state === s_req || state === s_wait1)  
    next_state := s_ready  
    resp_valid(r_req_dest) := true  
    resp_ae := false  
    count := pgLevels-1  
}
```

— explanation —

when l2_hit AND l2_error is false
conds asserted
next_state becomes s_ready
resp_valid index r_req_dest becomes true
resp_ae becomes false
count again becoems pgLevels -1



```

when (mem_resp_valid) {
    assert(state === s_wait3)
when (traverse) {
    next_state := s_req
    count := count + 1
}.otherwise {
    l2_refill := pte.v && !invalid_paddr && count === pgLevels-1
    val ae = pte.v && invalid_paddr
    resp_ae := ae
}

```

 explanation

when mem_resp_valid is true

conditions are asserted

when traverse is true

next state becomes s_req

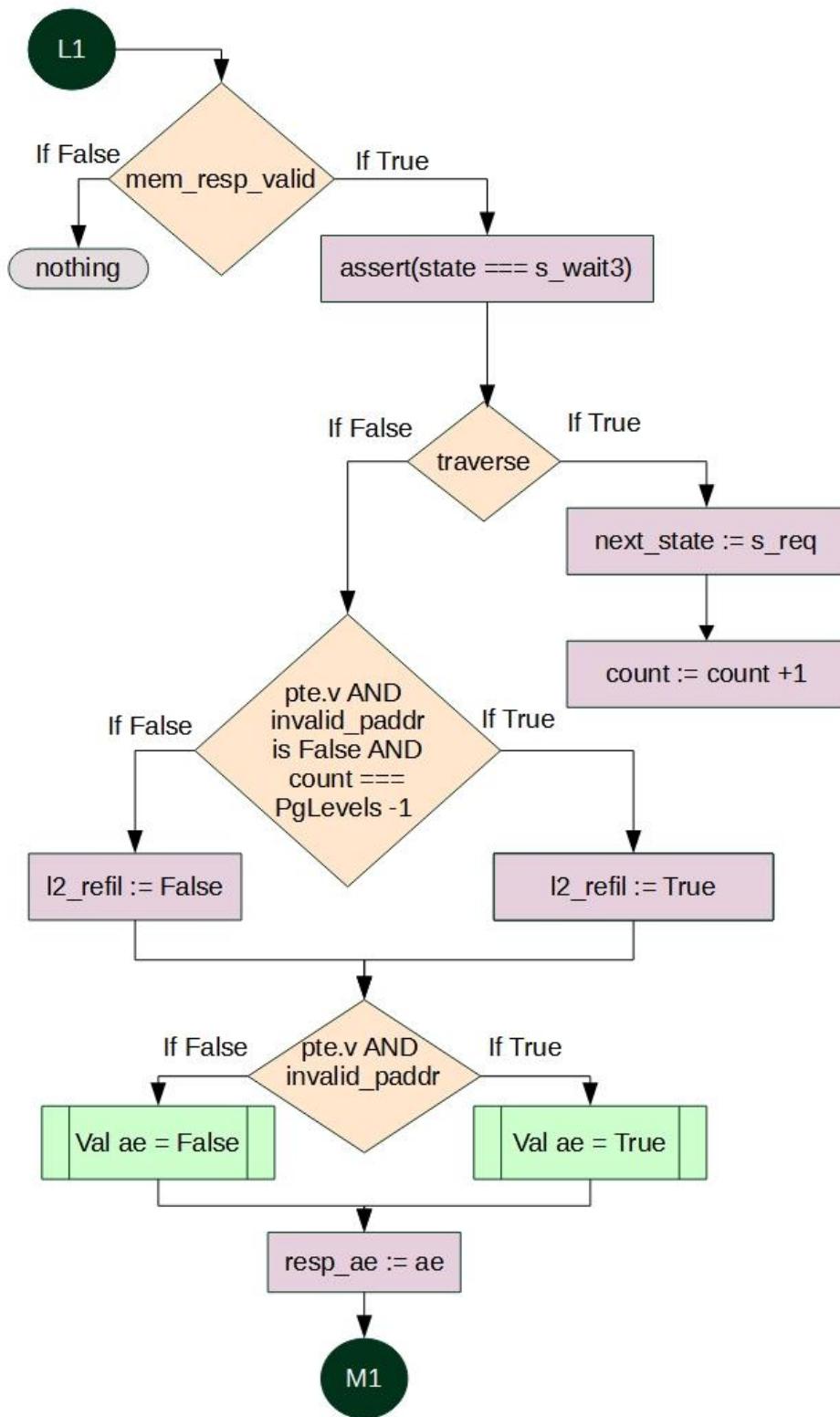
count +=1

else

l2_refill is true when pte.v is true AND invalid_paddr is false AND count == pgLevels-1

ae is pte.v AND invalid_paddr

esp_ae becomes ae (pte.v AND invalid_paddr)



ROCKET-CHIP Micro Architecture Specification Document

```
when (pageGranularityPMPs && count == pgLevels-1 && !ae) {
    next_state := s_fragment_superpage
}.otherwise {
    next_state := s_ready
    resp_valid(r_req_dest) := true
}
}

when (io.mem.s2_nack) {
    assert(state == s_wait2)
    next_state := s_req
}
```

— explanation —

when pageGranPMPs AND count != pgLevels-1 AND ae is false

next state becomes frag superpage

else

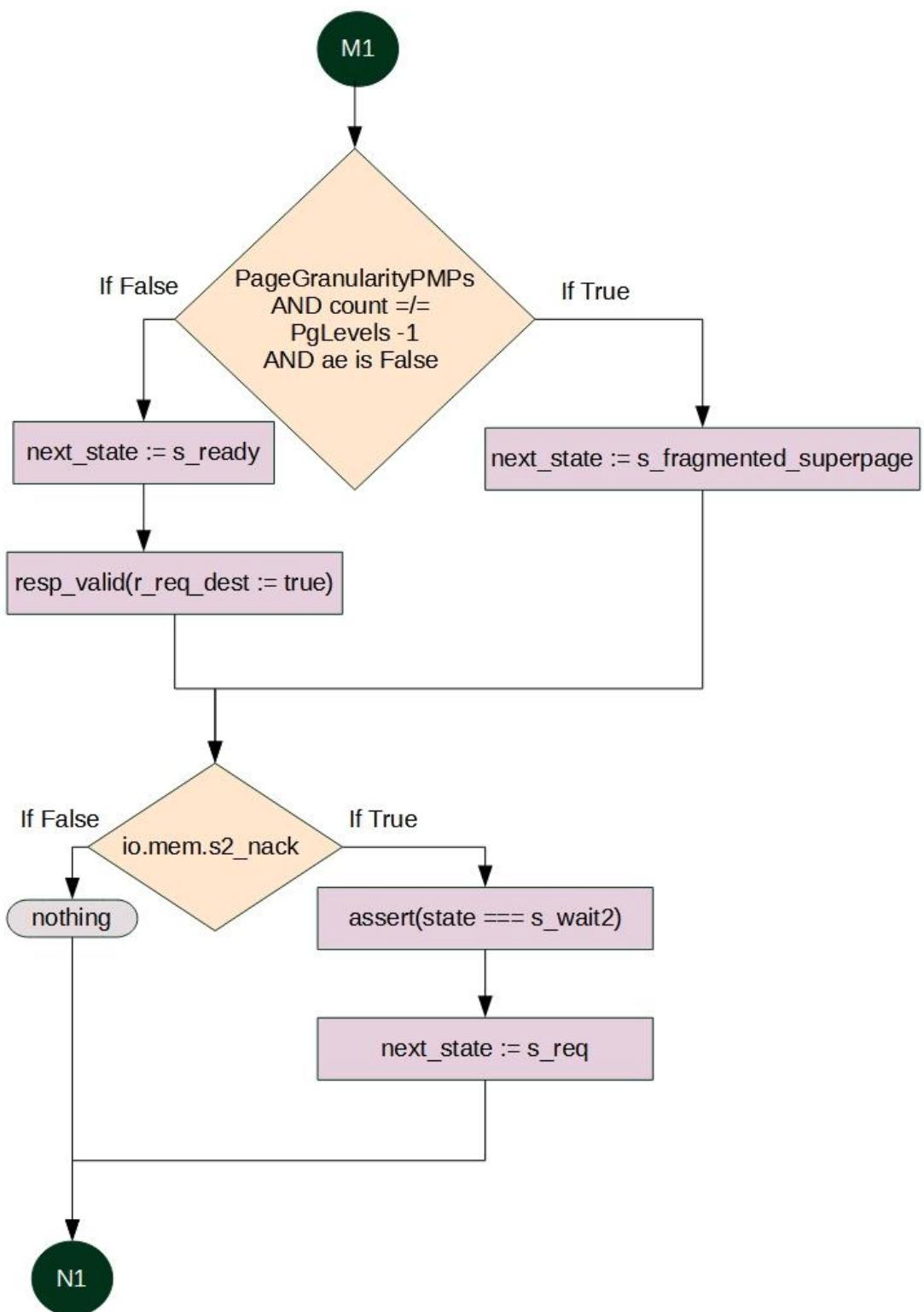
next state is s_ready

resp_valid index r_req_dest is true

when mem s2_nack is true

condition asserted

next state becomes s_req



```

for (i <- 0 until pgLevels) {
    val leaf = mem_resp_valid && !traverse && count === i
    ccover(leaf && pte.v && !invalid_paddr, s"L${i}", s"successful page-table
access, level ${i}")
    ccover(leaf && pte.v && invalid_paddr, s"L${i}_BAD_PPN_MSB", s"PPN too
large, level ${i}")
    ccover(leaf && !mem_resp_data(0), s"L${i}_INVALID_PTE", s"page not
present, level ${i}")
    if (i != pgLevels-1)
        ccover(leaf && !pte.v && mem_resp_data(0), s"L${i}_BAD_PPN_LSB",
s"PPN LSBS not zero, level ${i}")
}

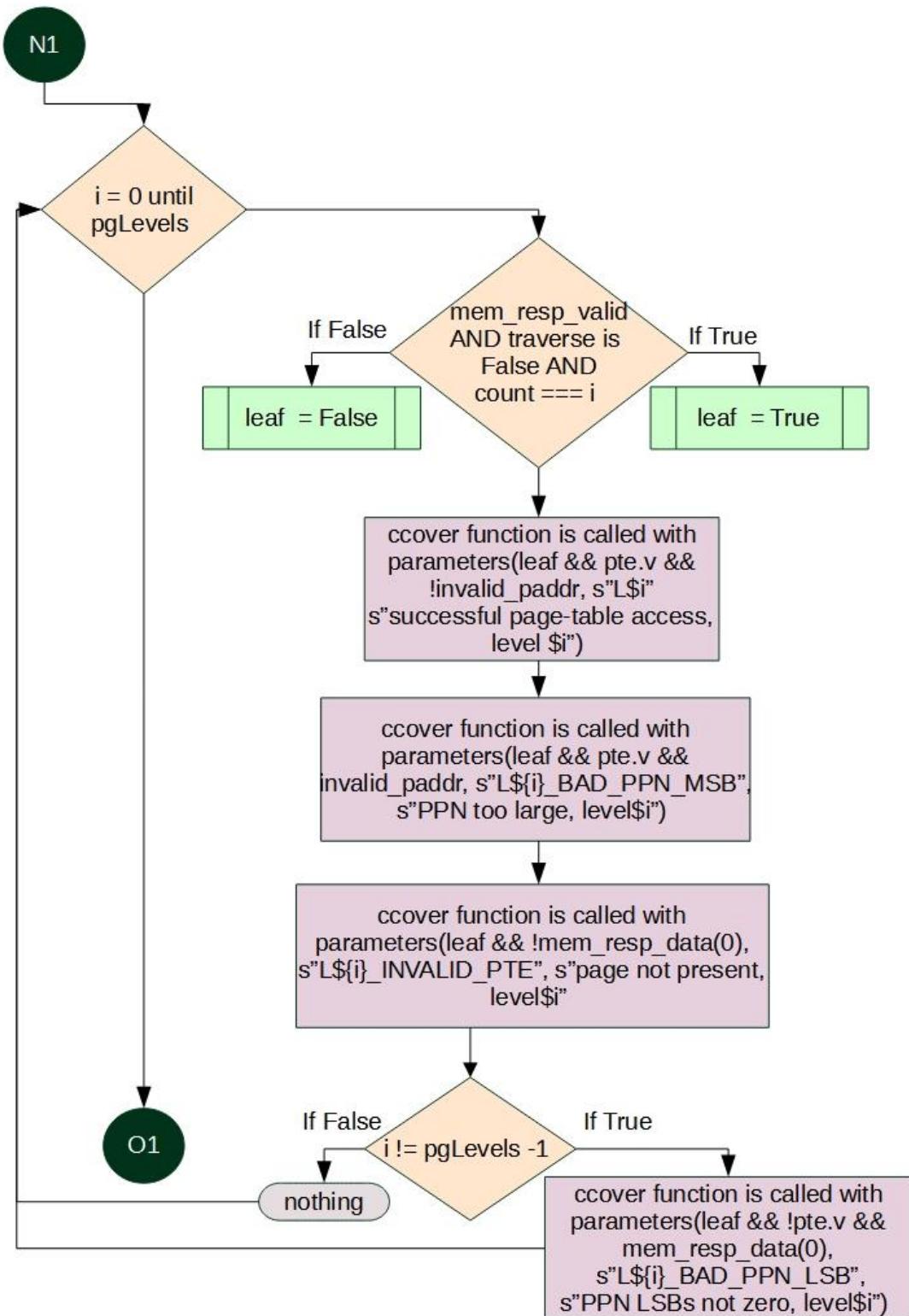
```

 explanation

loop iterating from 0 to pgLevels -1

leaf is mem_resp_valid AND trasvers is false AND count === i (last iteration) , it'll be true only then

ccover methods are called with the given parameters.



ROCKET-CHIP Micro Architecture Specification Document

```
ccover(mem_resp_valid && count === pgLevels-1 && pte.table(), s"TOO_DEEP",
      s"page table too deep")
  ccover(io.mem.s2_nack, "NACK", "D$ nacked page-table access")
  ccover(state === s_wait2 && io.mem.s2_xcpt.ae.ld, "AE", "access exception
while walking page table")
} // leaving gated-clock domain

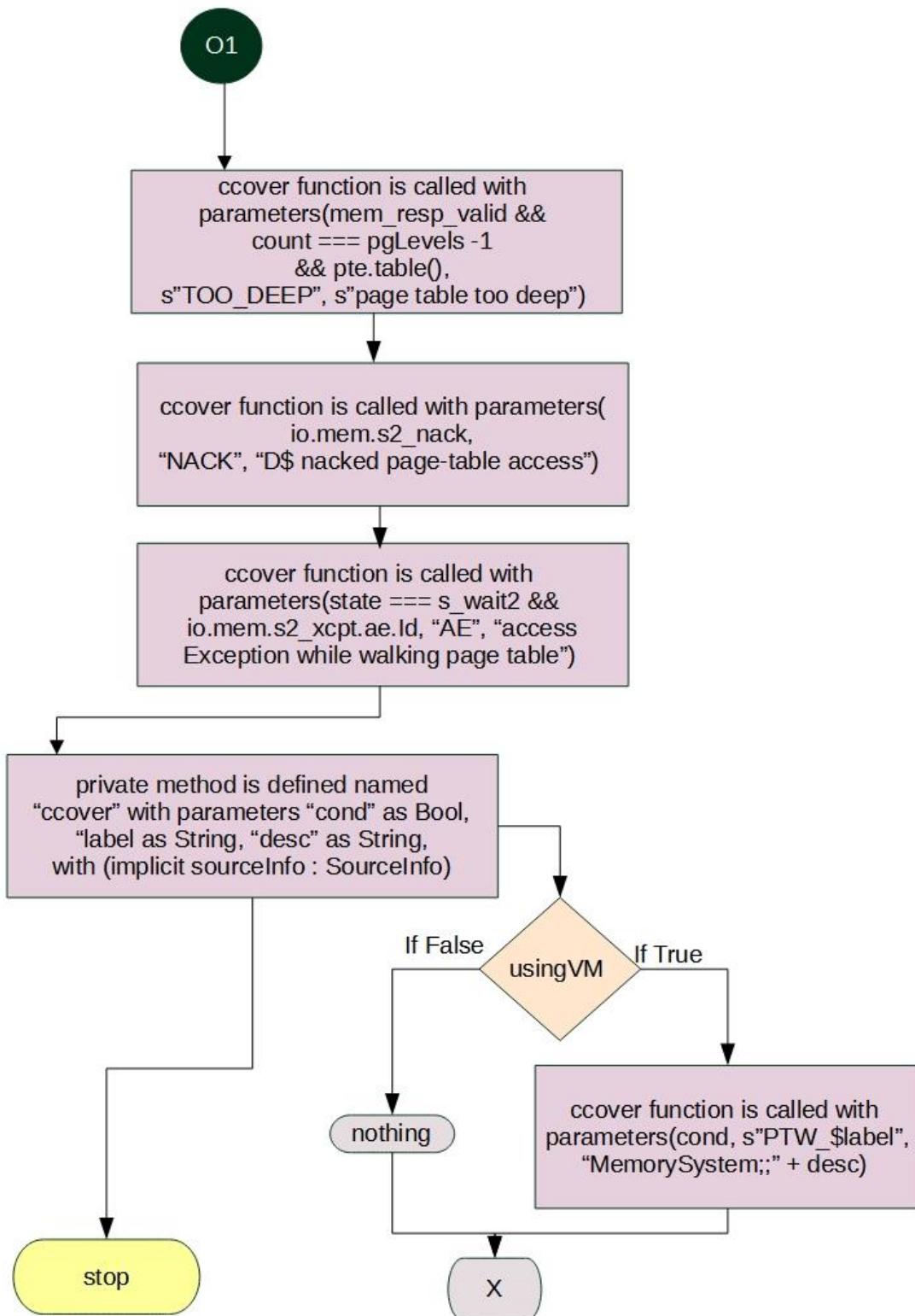
private def ccover(cond: Bool, label: String, desc: String) (implicit
  sourceInfo: SourceInfo) =
  if (usingVM) cover(cond, s"PTW_$label", "MemorySystem;" + desc)
}
```

explanation

ccover methods called with given parameters

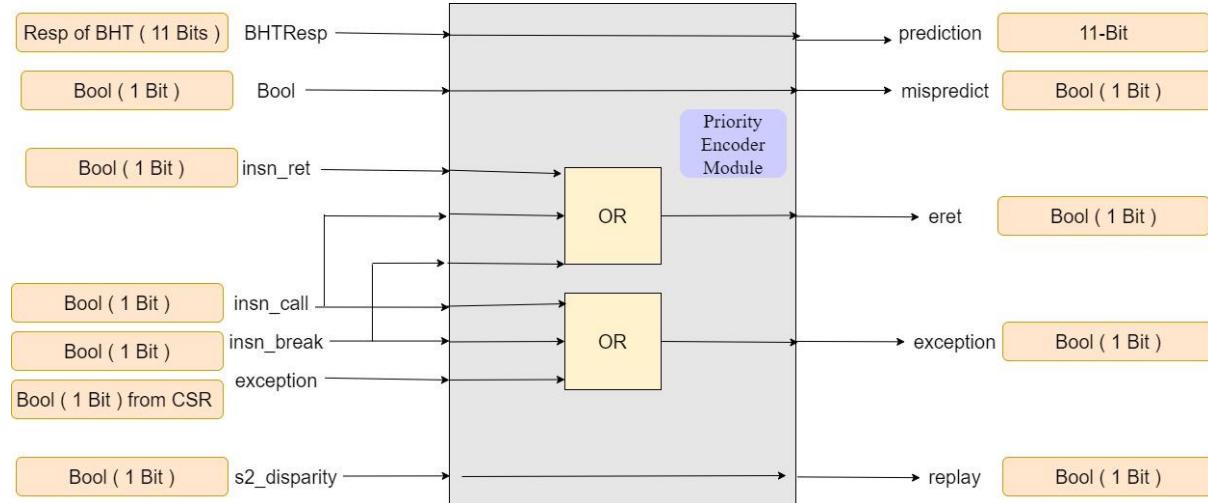
private method ccover is defined.

If Virtual Memory is in use then and only then cover method will be called.



Priority Encoder

Block Diagram:



DEEP DIVE INTO CODE

Explanation (By Means of Flowcharts) :

```
class BTBResp(implicit p: Parameters) extends BtbBundle()(p) {
    val cfiType = CFIType()
    val taken = Bool()
    val mask = Bits(width = fetchWidth)
    val bidx = Bits(width = log2Up(fetchWidth))
    val target = UInt(width = vaddrBits)
    val entry = UInt(width = log2Up(entries + 1))
    val bht = new BHTResp
}
```

— explanation —

First BTBResp class is defined which is extended with BtbBundle.

CFIType() object is instantiated in cfiType variable.

```
object CFIType {
    def SZ = 2
    def apply() = UInt(width = SZ)
    def branch = 0.U
    def jump = 1.U
    def call = 2.U
    def ret = 3.U
}
```

the taken variable is Bool

mask variable has Bits of width as fetchWidth, taken from the RocketCore.scala file whereas the line of code is

```
val fetchWidth: Int = if (useCompressed) 2 else 1
```

bidx variable has Bits of width log2Up of fetchwidth.

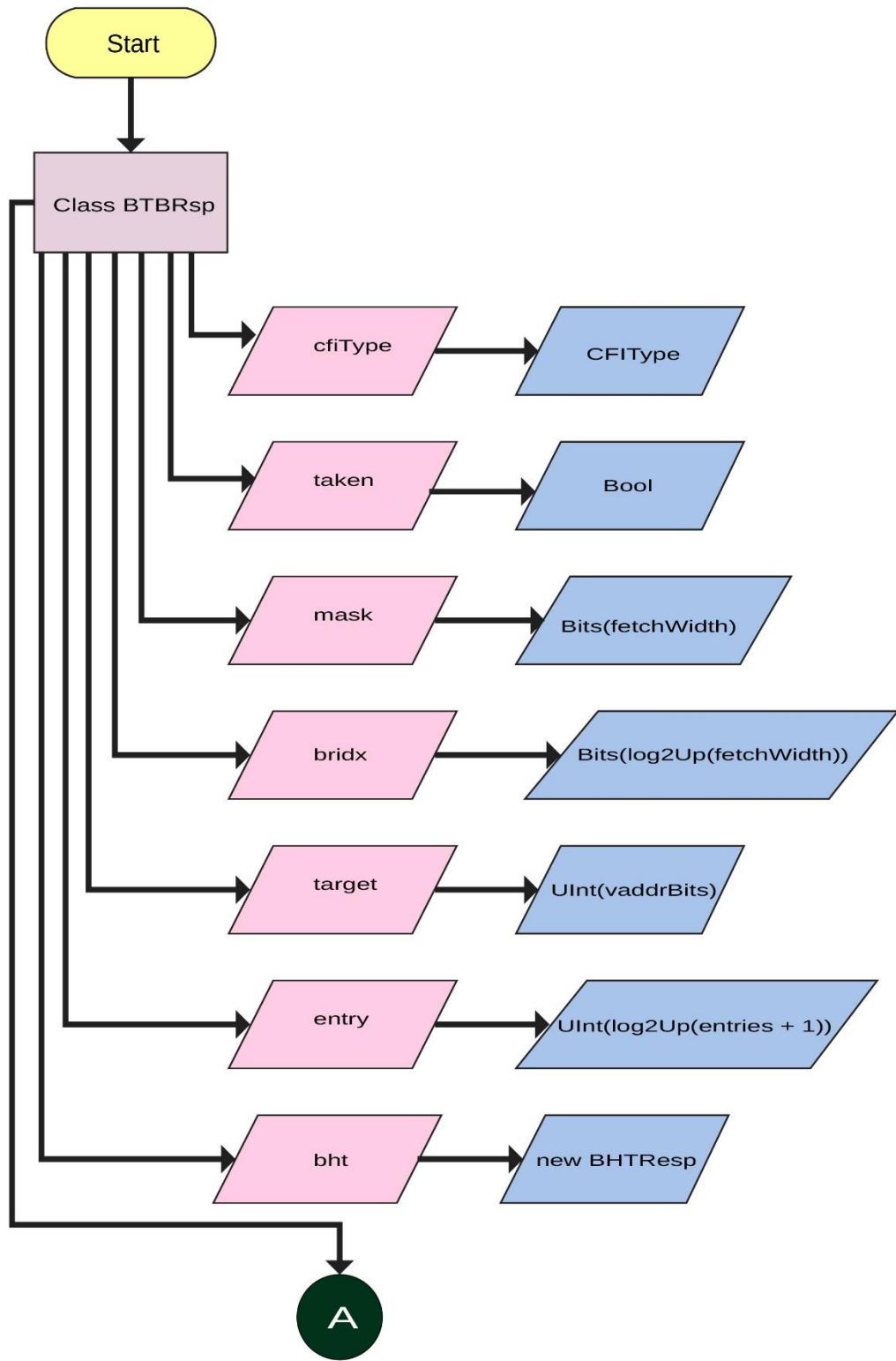
target variable has [UInt](#) of width vaddrBits(Virtual Address Bits), taken from BaseTile.scala file whereas the code is

```
def vaddrBits: Int =
    if (usingVM) {
        val v = maxSVAddrBits
        require(v == xLen || xLen > v && v > paddrBits)
    } else {
        // since virtual addresses sign-extend but physical addresses
        // zero-extend, make room for a zero sign bit for physical addresses
        (paddrBits + 1) min xLen
    }
def maxSVAddrBits: Int = pgIdxBits + pgLevels * pgLevelBits
def pgIdxBits: Int = 12
def pgLevels: Int = p(PgLevels)
```

```
case object PgLevels extends Field[Int] (2)
def pgLevelBits: Int = 10 - log2Ceil(xLen / 32)
```

entry variable has UInt of width [log2Up](#) of entries + 1, which is the parameter of abstract trait of btb whereas the line of code is

```
trait HasBtbParameters extends HasCoreParameters { this: InstanceId =>
  val btbParams = tileParams.btb.getOrElse(BTBParams(nEntries = 0))
  val matchBits = btbParams.nMatchBits max log2Ceil(p(CacheBlockBytes)
tileParams.icache.get.nSets)
  val entries = btbParams.nEntries
  val updatesOutOfOrder = btbParams.updatesOutOfOrder
  val nPages = (btbParams.nPages + 1) / 2 * 2 // control logic assumes 2
divides pages
}
```



```

class BTBUpdate(implicit p: Parameters) extends BtbBundle(p) {
  val prediction = new BTBResp
  val pc = UInt(width = vaddrBits)
  val target = UInt(width = vaddrBits)
  val taken = Bool()
  val isValid = Bool()
  val br_pc = UInt(width = vaddrBits)
  val cfiType = CFIType()
}

```

— explanation —

Class BTBUpdate is initialized extended by BtbBundle.

prediction variable has BTBResp object

pc variable has UInt of width vaddrBits

```

def vaddrBits: Int =
  if (usingVM) {
    val v = maxSVAddrBits
    require(v == xLen || xLen > v && v > paddrBits)
    v
  } else {
    // since virtual addresses sign-extend but physical addresses
    // zero-extend, make room for a zero sign bit for physical addresses
    (paddrBits + 1) min xLen
  }
def maxSVAddrBits: Int = pgIdxBits + pgLevels * pgLevelBits
def pgIdxBits: Int = 12
def pgLevels: Int = p(PgLevels)
case object PgLevels extends Field[Int](2)
def pgLevelBits: Int = 10 - log2Ceil(xLen / 32)

```

target variable has UInt of width vaddrBits

the taken variable has Bool

isValid variable has Bool

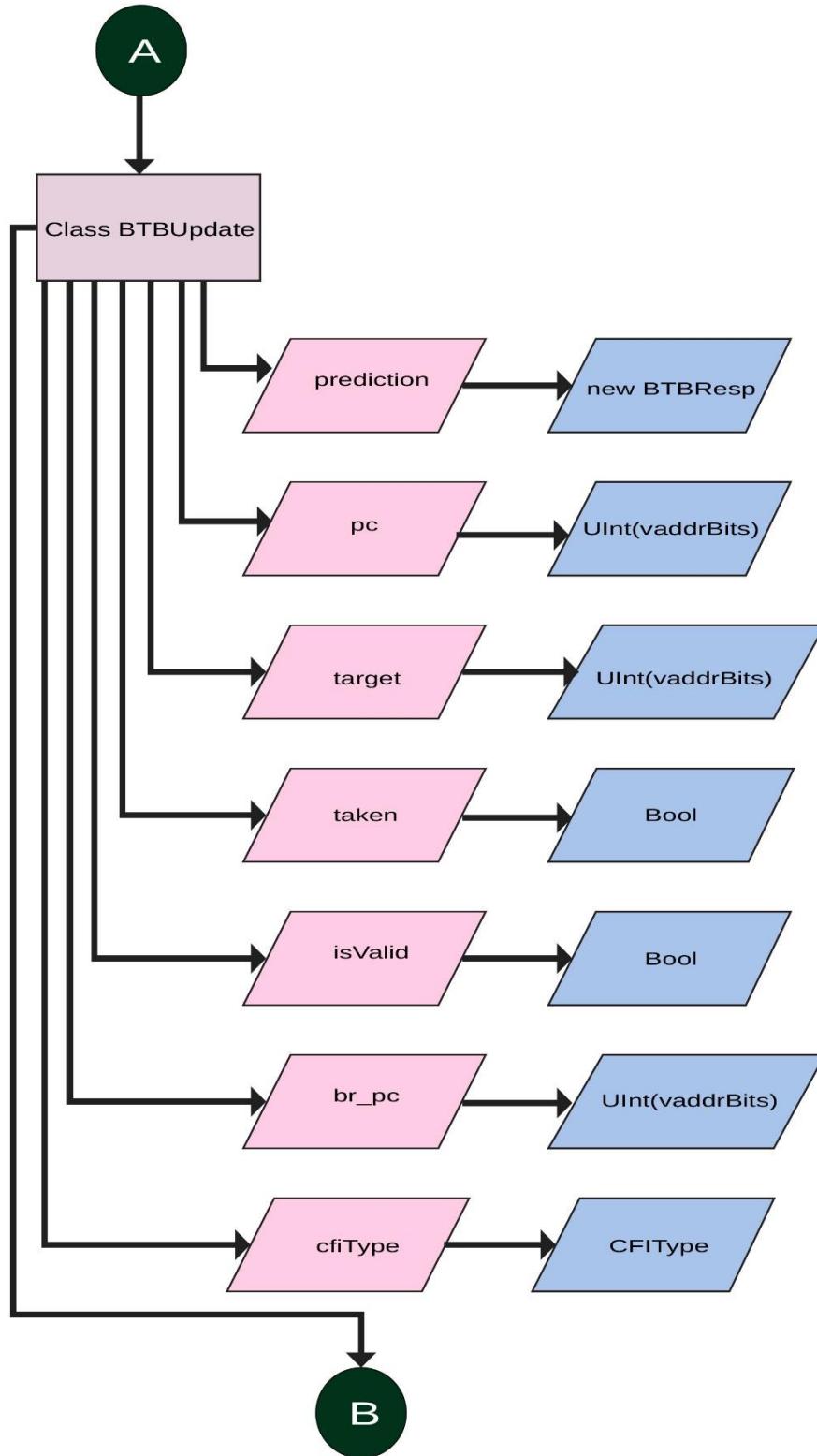
br_pc variable has UInt of width vaddrBits

cfiType variable has the object of CFIType

```

object CFIType {
  def SZ = 2
  def apply() = UInt(width = SZ)
  def branch = 0.U
  def jump = 1.U
  def call = 2.U
  def ret = 3.U
}

```



```

class BHTResp(implicit p: Parameters) extends BtbBundle()(p) {
    val history = UInt(width =
btbParams.bhtParams.map(_.historyLength).getOrElse(1))
    val value = UInt(width =
btbParams.bhtParams.map(_.counterLength).getOrElse(1))
    def taken = value(0)
    def strongly_taken = value === 1
}

```

explanation

BHTResp class is defined which is extended with BtbBundle.

```

abstract class BtbBundle(implicit val p: Parameters) extends Bundle with
HasBtbParameters
trait HasBtbParameters extends HasCoreParameters { this: InstanceId =>
    val btbParams = tileParams.btb.getOrElse(BTBParams(nEntries = 0))
    val matchBits = btbParams.nMatchBits max log2Ceil(p(CacheBlockBytes))
tileParams.icache.get.nSets)
    val entries = btbParams.nEntries
    val updatesOutOfOrder = btbParams.updatesOutOfOrder
    val nPages = (btbParams.nPages + 1) / 2 * 2 // control logic assumes 2
divides pages
}

```

history variable has UInt of width of a value coming from btbParams.bhtParams mapped with _.historyLength with function getOrElse as 1

value variable has UInt of width of a value coming from btbParams.bhtParams mapped with _.counterLength with function getOrElse as 1

historyLength and counterLength is initialized in the case class named BHTParams chunk of code given below.

```

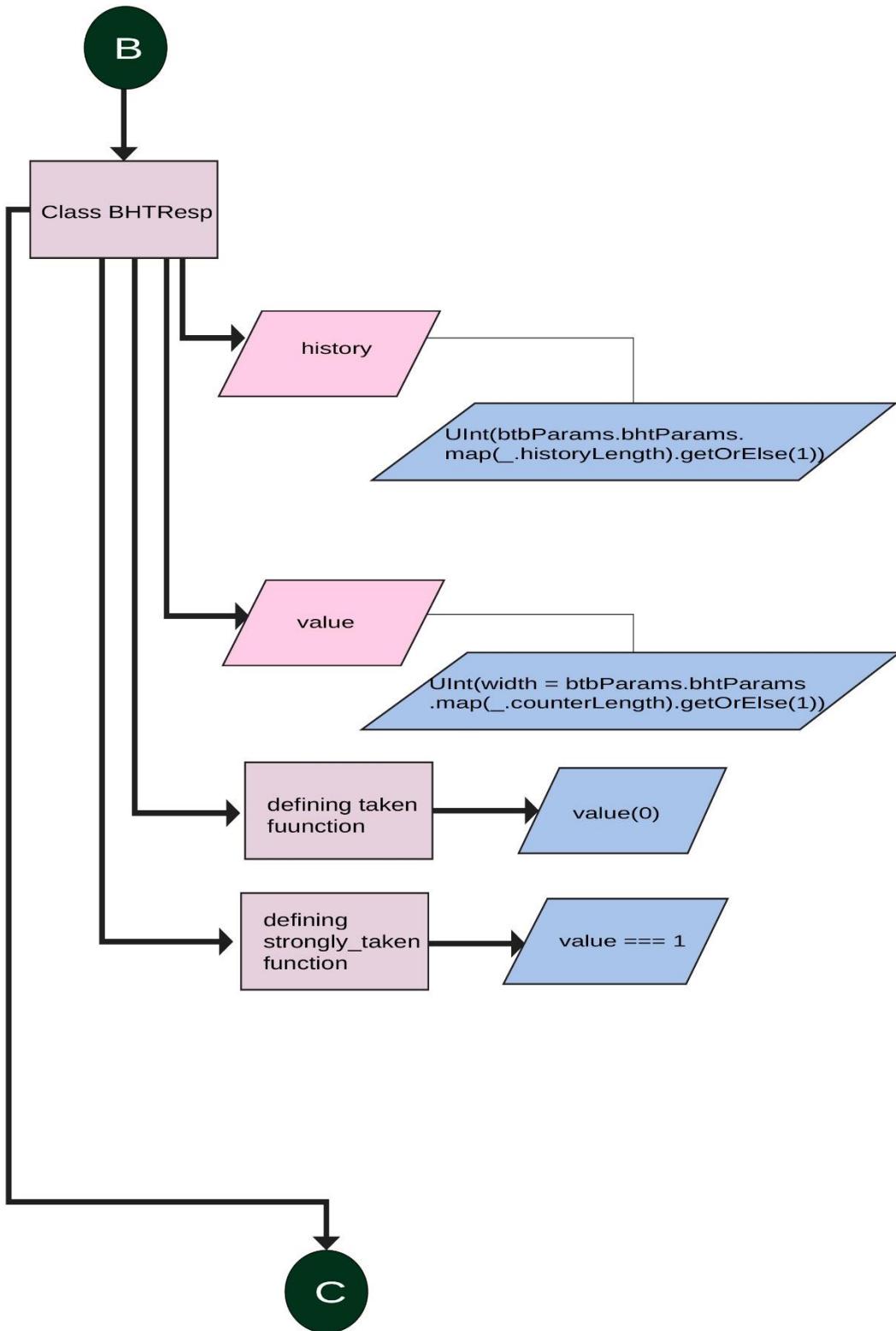
case class BHTParams(
    nEntries: Int = 512,
    counterLength: Int = 1,
    historyLength: Int = 8,
    historyBits: Int = 3)

```

btbParams.bhtParams.map is a map in which _.historyLength is provided as key to find the value of key if the value of given key is None then getOrElse(1) function will save 1 in history attribute, and in _.counterLength, _.counterLength is provided as key to find the value of key if the value of given key is None then getOrElse(1) function will save 1 in value attribute.

Taken method is defined and 0 key is send into the value parameter to get the value of 0 key.

Strongly_taken method is defined with Boolean value if value = 1 it becomes True except False.



```

class BHTUpdate(implicit p: Parameters) extends BtbBundle()(p) {
    val prediction = new BHTResp
    val pc = UInt(width = vaddrBits)
    val branch = Bool()
    val taken = Bool()
    val mispredict = Bool()
}
class BTBReq(implicit p: Parameters) extends BtbBundle()(p) {
    val addr = UInt(width = vaddrBits)
}

```

— explanation —

Class BHTUpdate is initialized extended by BtbBundle.

Prediction variable has BHTResp object.

Pc variable has UInt of width vaddrBits

```

def vaddrBits: Int =
    if (usingVM) {
        val v = maxSVAddrBits
        require(v == xLen || xLen > v && v > paddrBits)
        v
    } else {
        // since virtual addresses sign-extend but physical addresses
        // zero-extend, make room for a zero sign bit for physical addresses
        (paddrBits + 1) min xLen
    }
def maxSVAddrBits: Int = pgIdxBits + pgLevels * pgLevelBits
def pgIdxBits: Int = 12
def pgLevels: Int = p(PgLevels)
case object PgLevels extends Field[Int](2)
def pgLevelBits: Int = 10 - log2Ceil(xLen / 32)

```

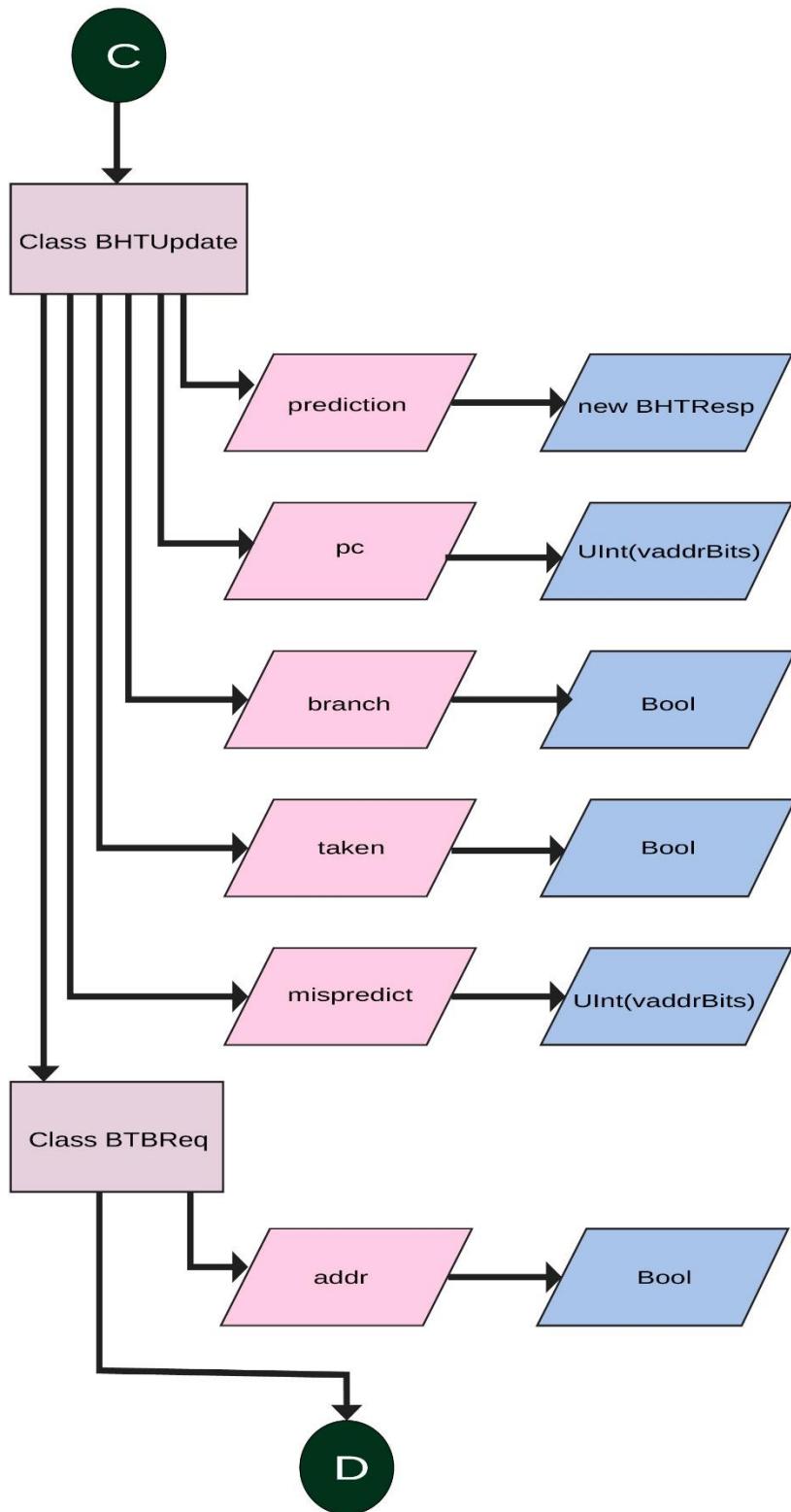
branch variable has Bool

taken variable has Bool

mispredict variable has Bool

Class BTBReq is initialized extended by BtbBundle

addr variable has UInt of vaddr



```

class BTB(implicit p: Parameters) extends BtbModule {
  val io = new Bundle {
    val req = Valid(new BTBReq).flip
    val resp = Valid(new BTBResp)
    val btb_update = Valid(new BTBUpdate).flip
    val bht_update = Valid(new BHTUpdate).flip
    val bht_advance = Valid(new BTBResp).flip
    val ras_update = Valid(new RASUpdate).flip
    val ras_head = Valid(UInt(width = vaddrBits))
    val flush = Bool().asInput
  }
}

```

 explanation

Class BTB is initialized extended by BtbModule

io is defined to be input/output;

req has BTBReq object with Valid function and the .flip, flips all its bits.

resp has BTBResp object with Valid function

btb_update has BTBUpdate object with Valid function and the .flip, flips all its bits

bht_update has BHTUpdate object with Valid function and the .flip, flips all its bits

bht_advance has BTBResp object with Valid function and the .flip, flips all its bits

ras_update has RASUpdate object with Valid function and the .flip, flips all its bits

```

class RASUpdate(implicit p: Parameters) extends BtbBundle()(p) {
  val cfiType = CFIType()
  val returnAddr = UInt(width = vaddrBits)
}

```

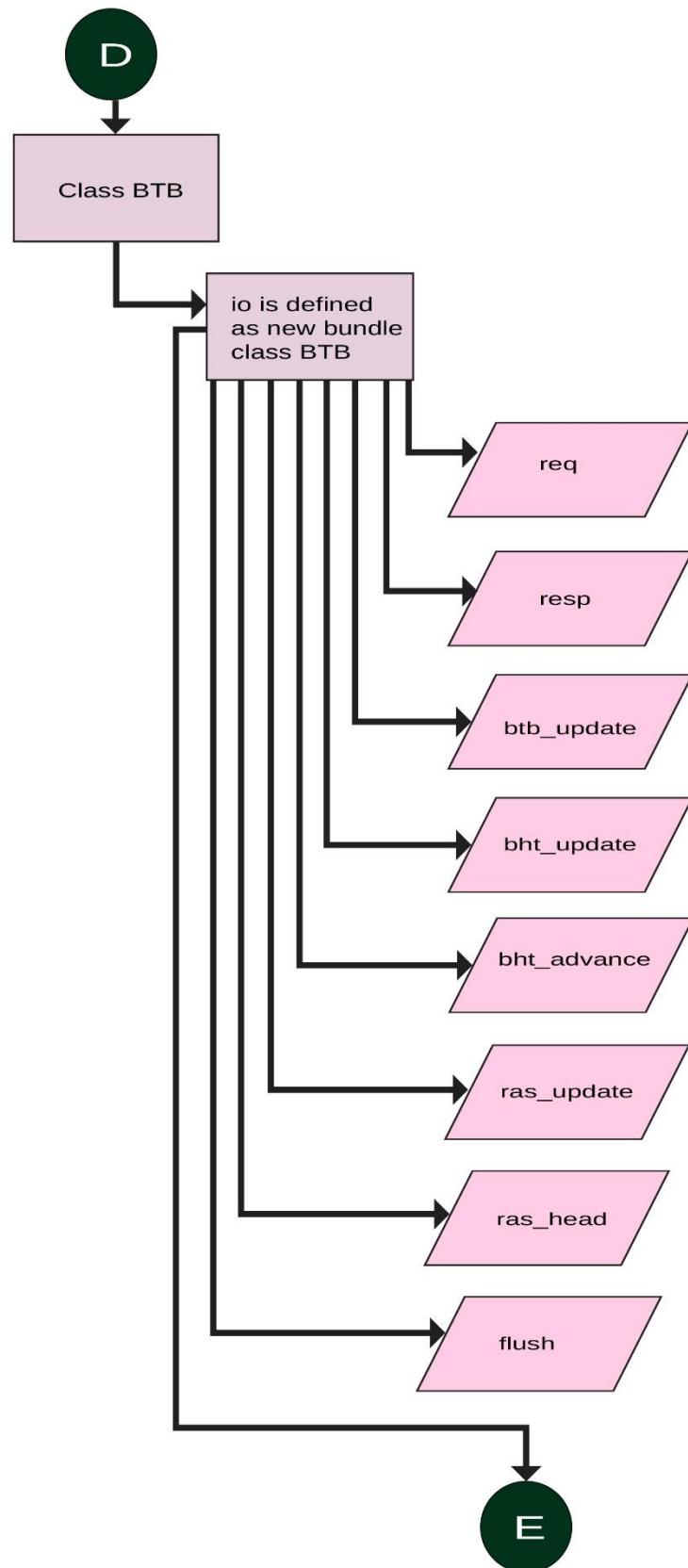
ras_head has UInt of width vaddrBits with valid function

```

def vaddrBits: Int =
  if (usingVM) {
    val v = maxSVAddrBits
    require(v == xLen || xLen > v && v > paddrBits)
    v
  } else {
    // since virtual addresses sign-extend but physical addresses
    // zero-extend, make room for a zero sign bit for physical addresses
    (paddrBits + 1) min xLen
  }
def maxSVAddrBits: Int = pgIdxBits + pgLevels * pgLevelBits
def pgIdxBits: Int = 12
def pgLevels: Int = p(PgLevels)
case object PgLevels extends Field[Int](2)
def pgLevelBits: Int = 10 - log2Ceil(xLen / 32)

```

flush has Bool as Input

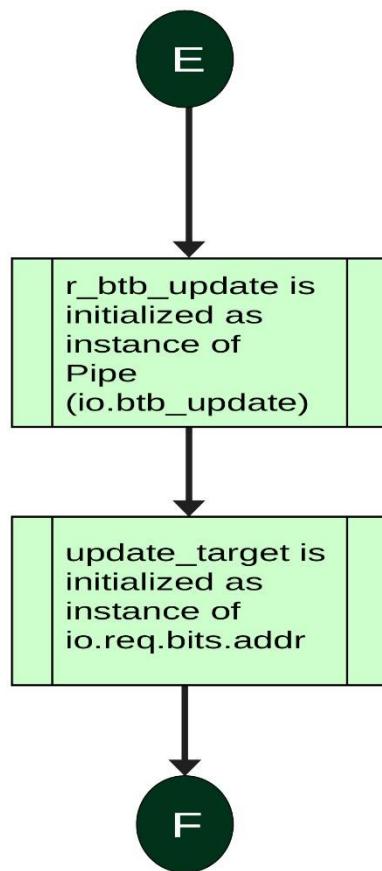


```
val r_btb_update = Pipe(io.btb_update)
val update_target = io.req.bits.addr
```

— explanation —

Btb_update is the attribute of BTB class which is piped.

req is also the attribute of BTB class and the address of bits in req attribute is save as instance in update_target



```

val (updateHit, updateHitAddr) =
  if (updatesOutOfOrder) {
    val updateHits = (pageHit << 1) (Mux1H(idxMatch(r_btb_update.bits.pc),
idxPages))
    (updateHits.orR, OHToUInt(updateHits))
  } else (r_btb_update.bits.prediction.entry < entries,
r_btb_update.bits.prediction.entry)

  if (btbParams.bhtParams.nonEmpty) {
    val bht = new BHT(Annotated).params(this, btbParams.bhtParams.get))
    val isBranch = (idxHit & cfiType.map(_ === CFIType.branch).asUInt).orR
    val res = bht.get(io.req.bits.addr)
  }
  when (io.bht_advance.valid) {
    bht.advanceHistory(io.bht_advance.bits.bht.taken)
  }
}

```

— explanation —

values updateHit and updateHitAddr will be initialized by the return value of,

```

trait HasBtbParameters extends HasCoreParameters { this: InstanceId =>
  val btbParams = tileParams.btb.getOrElse(BTBParams(nEntries = 0))
  val matchBits = btbParams.nMatchBits max log2Ceil(p(CacheBlockBytes)
tileParams.icache.get.nSets)
  val entries = btbParams.nEntries
  val updatesOutOfOrder = btbParams.updatesOutOfOrder
  val nPages = (btbParams.nPages + 1) / 2 * 2 // control logic assumes 2
divides pages
}
case class BTBParams(
  nEntries: Int = 28,
  nMatchBits: Int = 14,
  nPages: Int = 6,
  nRAS: Int = 6,
  bhtParams: Option[BHTParams] = Some(BHTParams()),
  updatesOutOfOrder: Boolean = false)

```

In the trait HasBtbParameters, updatesOutOfOrder is initialized as btbParams.updatesOutOfOrder, where btbParams is initialized in the same class as an instance of case class BTBParams that contains the value of updatesOutOfOrder

So, if updatesOutOfOrder is true then, updateHits variable has, pageHit left shifted by 1 and value from Mux1H with cond as idxMatch(r_btb_update.bits.pc) and idxPages.

```

val pageHit = pageMatch(io.req.bits.addr)
private def pageMatch(addr: UInt) = {
  val p = page(addr)
  pageValid & pages.map(_ === p).asUInt
}
val page = 4096
val pageValid = Reg(init = UInt(0, nPages))
nPages: Int = 6, (case class BTBParams)
private def idxMatch(addr: UInt) = {
  val idx = addr(matchBits-1, log2Up(coreInstBytes))
  idxs.map(_ === idx).asUInt & isVali
}

```

```

        }
    val coreInstBytes = coreInstBits/8
    val coreInstBits = coreParams.instBits
    val isValid = Reg(init = UInt(0, entries))
    val idxPages = Reg(Vec(entries, UInt(width=log2Up(nPages))))
}

```

pageHit and pageMatch private method is defined in the BTB Class as well, then nPages is the attribute of case classs BTBParams, as well as the idxMatch private method is in the BTB Class as well.

Next, the btbParams.bhtParams.nonEmpty will be true if the bhtPrams object that is initialized as a variable in btbParams case class is not None, if the condition is True then,

bht is initialized as an instance of class BHT passing as a parameter the values of bhtParams object in btbParams.

isBranch has idxHit bits AND'ed with cfiType.mapped with _ === CFIType.branch, visualized as UInt , and the whole AND'ed result bits are all OR'ed with each other.

res is initialized as bht.get passing io.req.bits.addr as parameter,

```

def get(addr: UInt): BHTResp = {
    val res = Wire(new BHTResp)
    res.value := table(index(addr, history))
    res.history := history
    res
}

```

This method returns a BHTResp object hardware wired, into the res variable

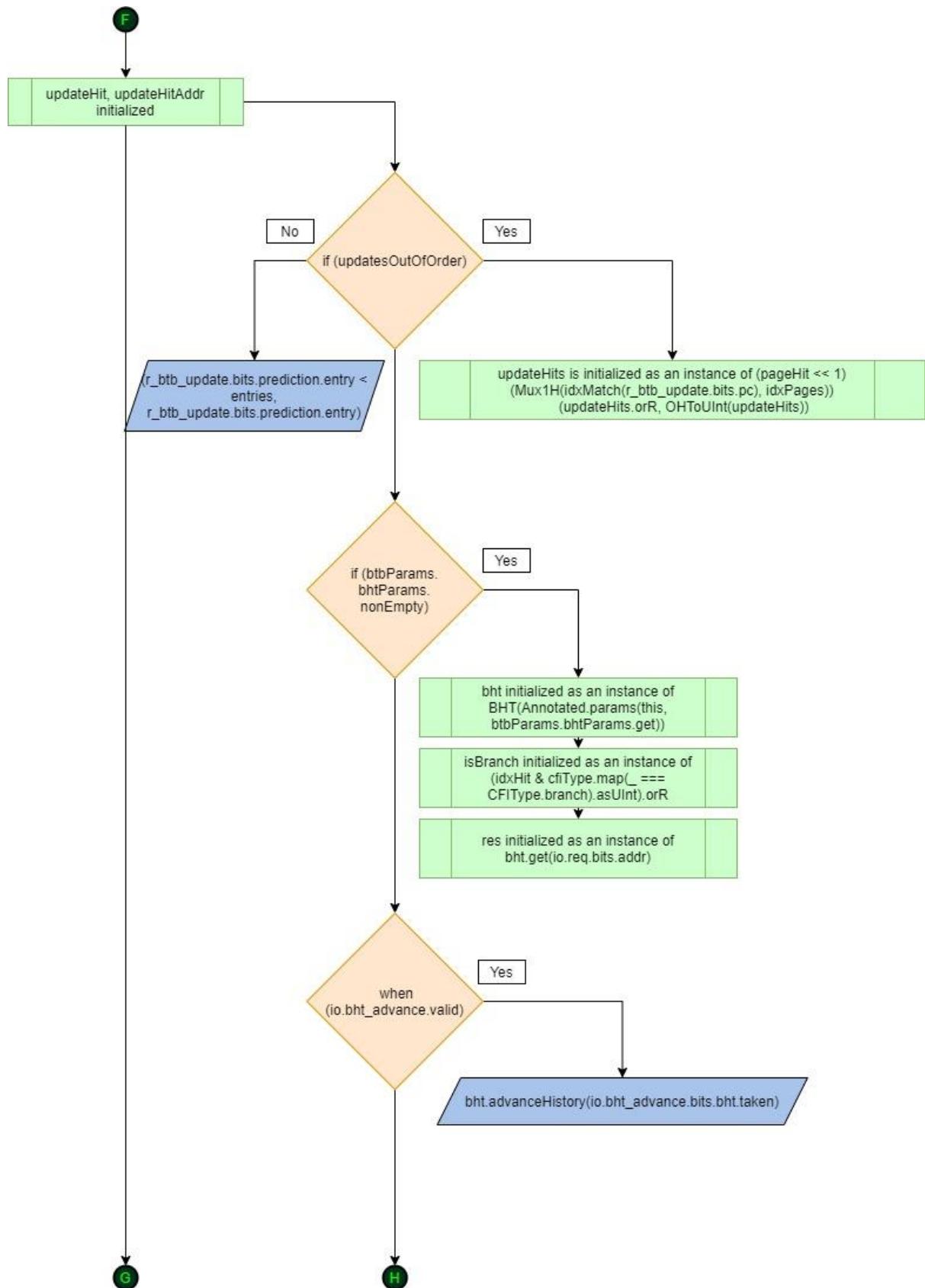
bht_advance is initialized as an instance of BTBResp.

bht.advanceHistory, advanceHistory is a function defined in the class bht which takes as a parameter the value of taken variable from the class BTBResp.

```

def advanceHistory(taken: Bool): Unit = {
    history := Cat(taken, history >> 1)
}

```



Now the code that contains the **prediction** and **mispredict** io pins is,

```

when (io.bht_update.valid) {
    when (io.bht_update.bits.branch) {
        bht.updateTable(io.bht_update.bits.pc,
io.bht_update.bits.prediction, io.bht_update.bits.taken)
            when (io.bht_update.bits.mispredict) {
                bht.updateHistory(io.bht_update.bits.pc,
io.bht_update.bits.prediction, io.bht_update.bits.taken)
            }
        }.elsewhen (io.bht_update.bits.mispredict) {
            bht.resetHistory(io.bht_update.bits.prediction)
        }
    }
    when (!res.taken && isBranch) { io.resp.bits.taken := false }
    io.resp.bits.bht := res
}

```

explanation

bht_update is initialized in the io bundle as an instance of class BHTUpdate, when its valid is true then,

bht_update.bits.branch is 1 (true) then,

The updateTable function in class BHT takes 3 arguments, which are passed in as the values from BHTUpdate class instance,

```

def updateTable(addr: UInt, d: BHTResp, taken: Bool): Unit = {
    table(index(addr, d.history)) := (params.counterLength match {
        case 1 => taken
        case 2 => Cat(taken ^ d.value(0), d.value === 1 || d.value(1) &&
taken)
    })
}

```

and When bhr_update.misprediction is 1 (true), then,

updateHistory is another function defined in the class BHT that requires arguments passed in from the values of bht_update object,

```

def updateHistory(addr: UInt, d: BHTResp, taken: Bool): Unit = {
    history := Cat(taken, d.history >> 1)
}

```

elsewhen, only bht_update.bits.mispredict is true and branch is false, then,

resetHistory is also a function defined in class the BHT

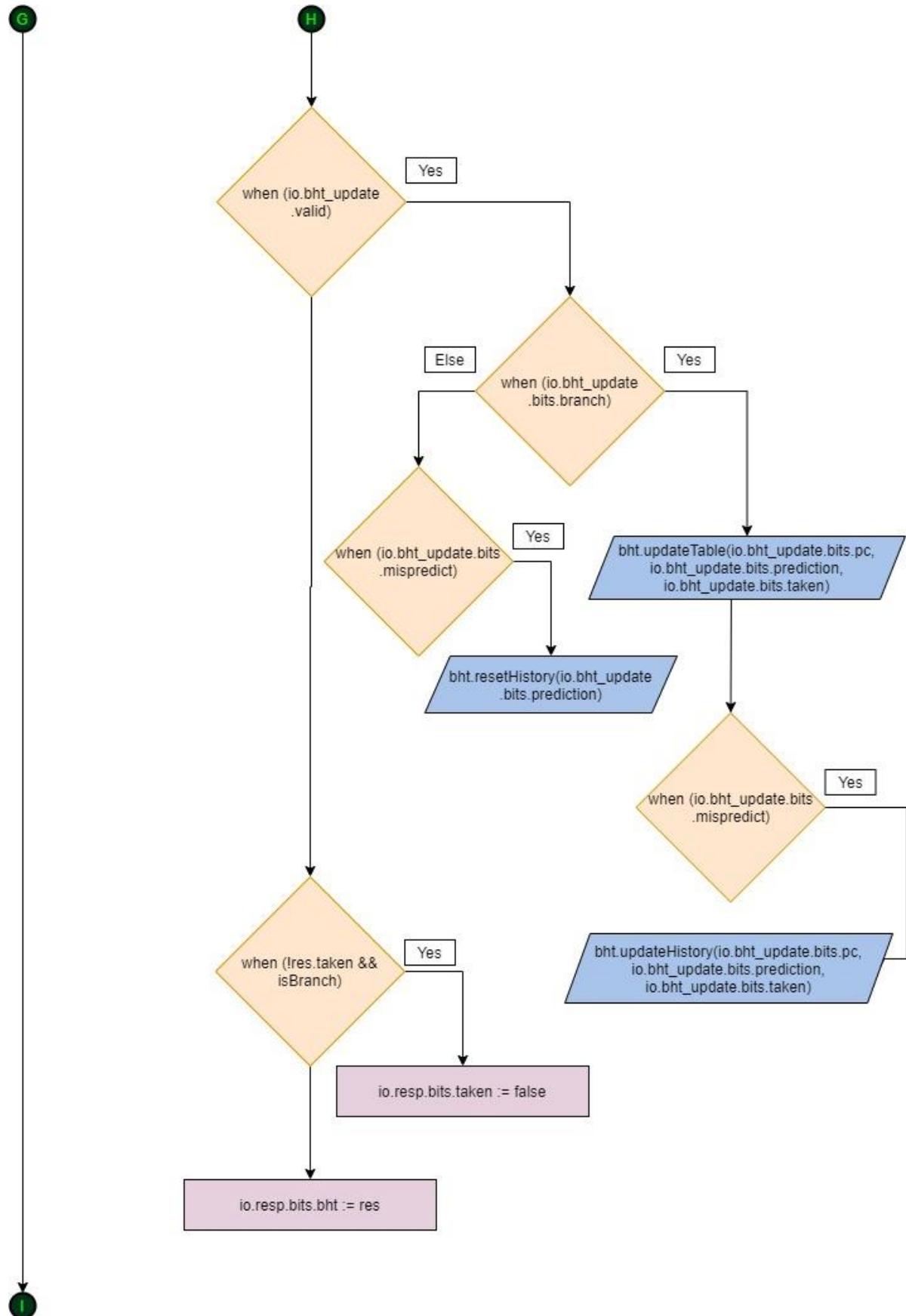
```

def resetHistory(d: BHTResp): Unit = {
    history := d.history
}

```

When res.taken is false and isBranch is true, then,

BHBResp is initialized as bht in the class BTBResp, which is then wired to res(the variable holding the get value or the primary BHT class)



```

val eret = Bool(OUTPUT)

val insn_call :: insn_break :: insn_ret :: insn_cease :: insn_wfi ::  

insn_sfence :: Nil =  
  

DecodeLogic(io.rw.addr << 20, decode_table(0)._2.map(x=>X),  

decode_table).map(system_insn && _.asBool  
  

io.eret := insn_call || insn_break || insn_ret  
  

class TracedInstruction(implicit p: Parameters) extends CoreBundle {  

  val valid = Bool()  

  val iaddr = UInt(width = coreMaxAddrBits)  

  val insn = UInt(width = iLen)  

  val priv = UInt(width = 3)  

  val exception = Bool()  

  val interrupt = Bool()  

  val cause = UInt(width = log2Ceil(1 + CSR.busErrorIntCause))  

  val tval = UInt(width = coreMaxAddrBits max iLen)  

}  
  

val exception = insn_call || insn_break || io.exception  
  

assert(PopCount(insn_ret :: insn_call :: insn_break :: io.exception ::  

Nil) <= 1, "these conditions must be mutually exclusive")

```

 explanation

value eret(**ERET PIN**) is initialized as a Boolean output in class CSRFileIO which extends CoreBundle.

io.eret is wired to the OR product of insn_call, insn_break and insn_ret, under the class CSRFile, in which io is initialized as an instance of CSRFileIO.

Here insn_call, insn_break and insn_ret are initialed with DecodeLogic, rw is initialized in CSRFileIO as a bundle containing addr as a UInt,

```

val rw = new Bundle {  

  val addr = UInt(INPUT, CSR.ADDRSZ)  

  val cmd = Bits(INPUT, CSR.SZ)  

  val rdata = Bits(OUTPUT, xLen)  

  val wdata = Bits(INPUT, xLen)  

}

```

ADDRSZ is defined as a value of CSR object

```

object CSR  

{ val ADDRSZ = 12 }

```

Decode Logic is an object that returns a seq of values.

decode_table is initialized as a seq in main CSRFile class

```
val decode_table = Seq(  
    SCALL-> List(Y,N,N,N,N,N),  
    SBREAK-> List(N,Y,N,N,N,N),  
    MRET-> List(N,N,Y,N,N,N),  
    CEASE-> List(N,N,N,Y,N,N),  
    WFI-> List(N,N,N,N,Y,N))
```

And system_insn holds a Boolean value for the result,

```
val system_insn = io.rw.cmd === CSR.I
```

in which io.rw.cmd links to,

```
val cmd = Bits(INPUT, CSR.SZ)
```

where SZ holds a value in CSR object

```
val SZ = 3
```

And CSR.I is defined as,

```
def I = UInt(4, SZ)
```

In CSR.scala value exception(**EXCEPTION PIN**) is initialized as an OR product of insn_call, insn_break and io.exception under the class CSRFile.

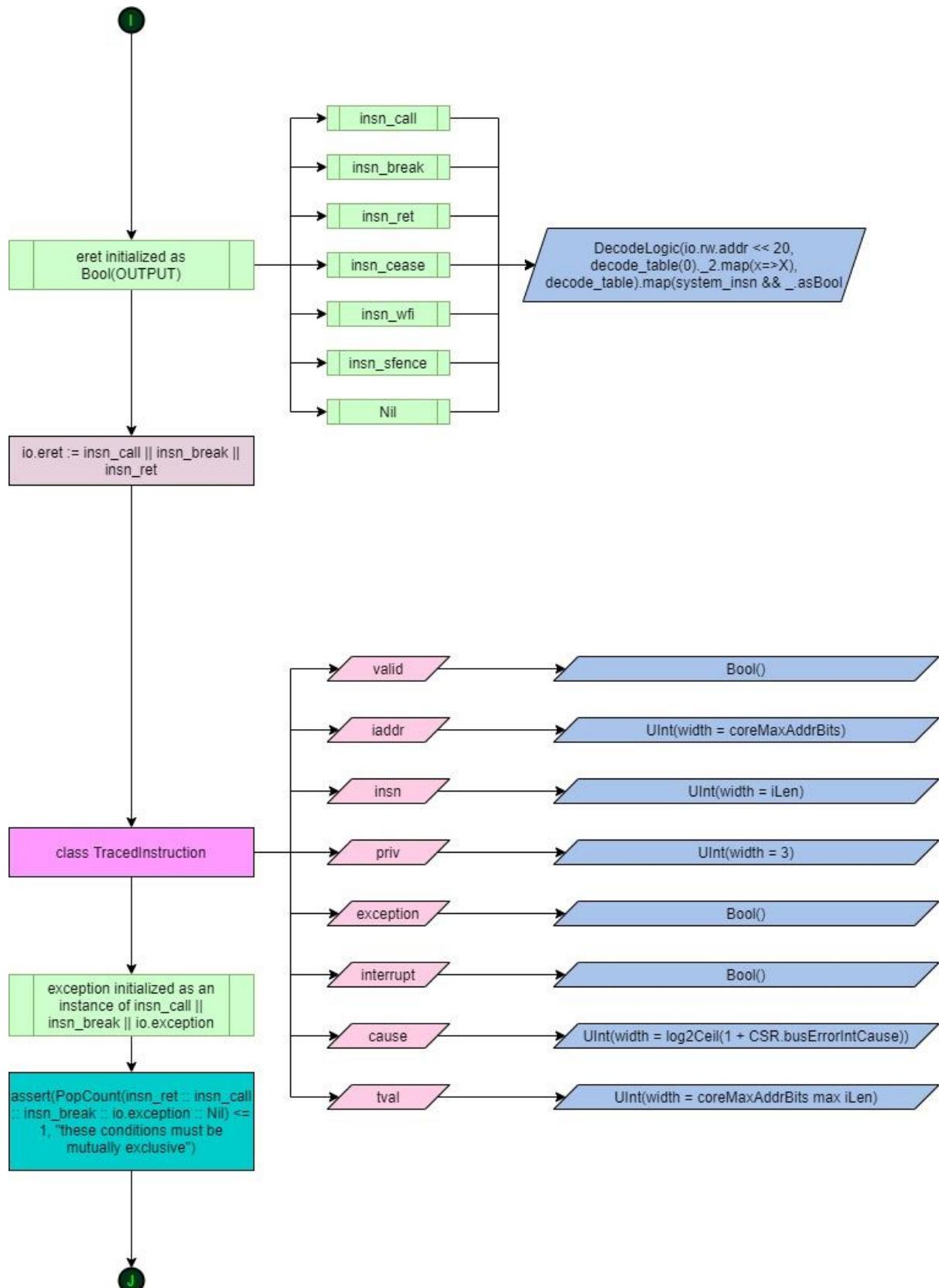
in which io is initialized as an instance of CSRFileIO

```
val io = new CSRFileIO {  
    val customCSRs = Vec(CSRFile.this.customCSRs.size, new  
CustomCSRIO).asOutput  
}
```

in Class CSRFileIO exception is initialized as a Boolean Input,

```
class CSRFileIO(implicit p: Parameters) extends CoreBundle  
    with HasCoreParameters {  
val exception = Bool(INPUT)}
```

in the last line of code PopCount returns the count of 1s in the list of boolean values, which in this case should be less than or equals to 1



```
class ICacheResp(outer: ICache) extends Bundle {
    val data = UInt(width = outer.icacheParams.fetchBytes*8)
    val replay = Bool()
    val ae = Bool()

    io.resp.bit.replay := s2_diparity
```

- explanation -

Class ICacheResp contains a variable replay (**REPLAY PIN**) initialized as Boolean,

Class ICacheBundle contains a value resp that is initialized as a valid outer of class ICacheResp,

```
class ICacheBundle(val outer: ICache) extends CoreBundle()(outer.p) {  
    val resp = Valid(new ICacheResp(outer))
```

In the main class `ICacheModule`, `io` is defined as the input/outputs of class `ICacheBundle`,

```
class ICacheModule(outer: ICache) extends LazyModuleImp(outer)
    with HasL1ICacheParameters {
  val io = IO(new ICacheBundle(outer))
```

Now inside the ICacheModule, This replay is wired to s2_disparity which is a variable initialized in this class,

```
io.resp.bits.replay := s2_disparity  
val s2_disparity = s2.tag.disparity || s2.data.decoded.error
```

This s2_disparity is the or product of s2_tag_disparity and s2_data_decoded,

s2_tag_disparity:

```
val s2 tag disparity = RegEnable(s1 tag disparity, s1 clk en).asUInt.orR
```

s2_tag_disparity is initialized as a register value which takes 2 arguments(1st is the next value which in this case is set s1_tag_disparity and the 2nd argument is the initialized value once the register resets, which in this case is set to s1_clk_en.

`s1_tag_disparity` is wired to a `vec(array)` that contains `nWays` and an instance of `bool`.

```
val s1_tag_disparity = Wire(Vec(nWays, Bool()))
```

Now nWays is a variable initialized with a value of int(4) in ICacheParams which is overridden and initialized in this class (ICacheModule)

While $s1_{clk_en}$ is the OR product of $s1_{valid}$ and $s1_{slaveValid}$

```
val s1_clk_en = s1_valid || s1_slaveValid
```

Where s1_valid is initialized as a register and wired to s0_valid,

```
val s1_valid = Reg(init=Bool(false))
s1_valid := s0_valid
```

s0_valid is initialized as an io related to the variable req initialized in ICacheBundle as a decoupled instance of ICacheReq which extends CoreBundle

```
val s0_valid = io.req.fire()
class ICacheBundle(val outer: ICache) extends CoreBundle()(outer.p) {
    val req = Decoupled(new ICacheReq).flip
class ICacheReq(implicit p: Parameters) extends CoreBundle()(p) with
HasL1ICacheParameters {
    val addr = UInt(width = vaddrBits)
}
```

CoreBundle extends ParameterizedBundle which is in the file Misc.scala and contains the function fire().

s1_slavevalid is an initialized register storing the value of s0_slavevalid

```
val s1_slaveValid = RegNext(s0_slaveValid, false.B)
val s0_slaveValid = tl_in.map(_ .a.fire()).getOrElse(false.B)
val (tl_in, edge_in) = outer.slaveNode.in.headOption.unzip
val slaveNode =
    TLManagerNode(icacheParams.itimAddr.toSeq.map { itimAddr =>
TLSlavePortParameters.v1(
    Seq(TLSlateParameters.v1(
        address      = Seq(AddressSet(itimAddr, size-1)),
        resources     = device.reg("mem"),
        regionType   = RegionType.IDEMPOTENT,
        executable    = true,
        supportsPutFull = TransferSizes(1, wordBytes),
        supportsPutPartial = TransferSizes(1, wordBytes),
        supportsGet     = TransferSizes(1, wordBytes),
        fifoId        = Some(0))), // requests handled in FIFO order
beatBytes = wordBytes,
minLatency = 1))})
}
s2_data_decoded:
val s2_data_decoded = dECC.decode(s2_way_mux)
val dECC = cacheParams.dataCode
case class ICACHEParams(
    def dataCode: Code = Code.fromString(dataECC)
    dataECC: Option[String] = None,
```

s2_way_mux is a mux that takes two arguments which are s2_tag_hit and s2_dout respectively,

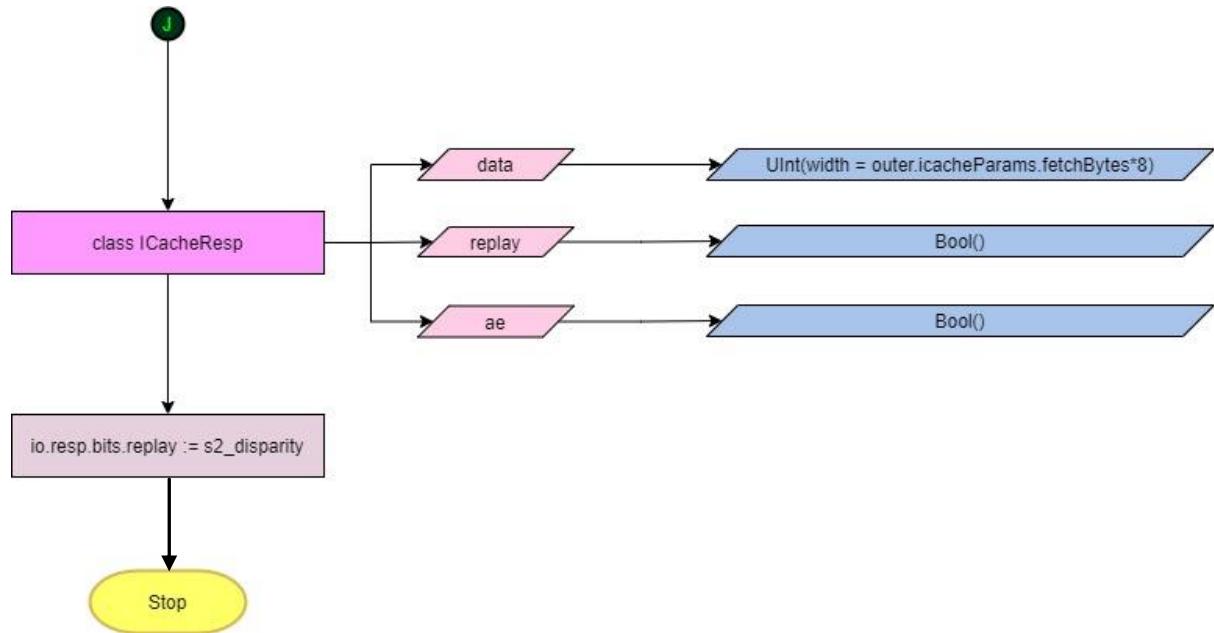
```
val s2_way_mux = Mux1H(s2_tag_hit, s2_dout)
```

s2_tag_hit is initialized as a register taking the arguments s1_tag_hit for its next value and s1_clk_en (tracked above) as reset initial.

```
val s2_tag_hit = RegEnable(s1_tag_hit, s1_clk_en)
```

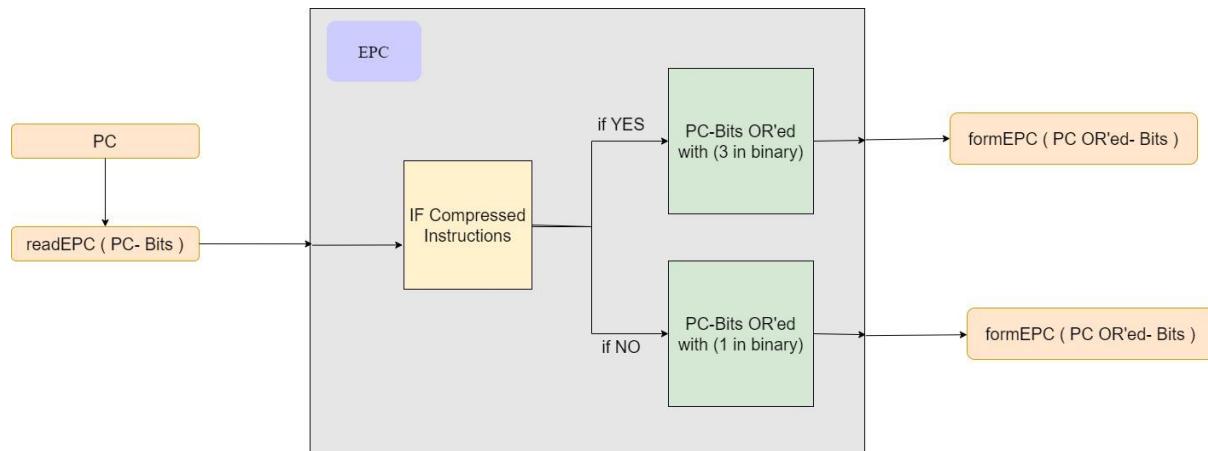
s1_tag_hit is initialized with the same value as s1_tag_disparity (mentioned above),

```
val s1_tag_hit = Wire(Vec(nWays, Bool()))
```



EPC

Block Diagram:



DEEP DIVE INTO CODE

Explanation (By Means of Flowcharts) :

```

val reg_misa = Reg(init=UInt(isaMax))

def readEPC(x: UInt) = ~(~x | Mux(reg_misa('c' - 'a'), 1.U, 3.U))
def formEPC(x: UInt) = ~(~x | (if (usingCompressed) 1.U else 3.U))

val reg_mepc = Reg(UInt(width = vaddrBitsExtended))
val reg_sepc = Reg(UInt(width = vaddrBitsExtended))
val reg_dpc = Reg(UInt(width = vaddrBitsExtended))

val read_mapping = LinkedHashMap[Int, Bits](
    CSRs.tselect -> reg_tselect,
    CSRs.tdata1 -> reg_bp(reg_tselect).control.asUInt,
    CSRs.tdata2 -> reg_bp(reg_tselect).address.sexTo(xLen),
    CSRs.misa -> reg_misa,
    CSRs.mstatus -> read_mstatus,
    CSRs.mtvec -> read_mtvec,
    CSRs.mip -> read_mip,
    CSRs.mie -> reg_mie,
    CSRs.mscratch -> reg_mscratch,
    CSRs.mepc -> readEPC(reg_mepc).sexTo(xLen),
    CSRs.mtval -> reg_mtval.sexTo(xLen),
    CSRs.mcause -> reg_mcause,
    CSRs.mhartid -> io.hartid)

```

explanation

immutable variable named reg_misa initialized register along with variable initialized above in CSR.scala with variable isaMax

```

val isaMax = (BigInt(log2Ceil(xLen) - 4) << (xLen-2)) |
isaStringToMask(isaString)
def isaStringToMask(s: String) = s.map(x => 1 << (x -
'A')).foldLeft(0)(_ | _)
val isaString = (if (coreParams.useRVE) "E" else "I") +
isaMaskString +
"X" + // Custom extensions always present (e.g. CEASE instruction)
(if (usingSupervisor) "S" else "") +
(if (usingUser) "U" else "")

```

isaMax has bits from BigInt of xLen log2Ceil'ed – 4, left shifter by xLen-2, OR'ed with isaStringToMask func with isaString variable.

function readEPC is created in which binary OR operation is performed between the parameter x and Mux with given values Unsigned Int format.

function formEPC is created in which binary OR operation is performed between the parameter x and condition along with given values in Unsigned Int format.

immutable variable named reg_mepc initialized register with UInt of width vaddrBitsExtended

```
def vaddrBitsExtended: Int = vpnBitsExtended + pgIdxBits
immutable variable named reg_sepc initialized register with UInt of width
vaddrBitsExtended
```

immutable variable named reg_dpc initialized register with UInt of width
vaddrBitsExtended

LinkedHashMap is a collection of key and value pairs which are stored internally
using Hash Table data structure. But iterating through the elements is done in order.

The -> operator is a shorthand for associating a key with a value when used in the
context of a map.

immutable variable named read_mapping is initialized and the Built-in function
Linked Hash Map is used. Associating the keys:

CSRs.tselect , CSRs.tdata1 , CSRs.tdata2 , CSRs.misa

CSRs.mstatus , CSRs.mtvec , CSRs.mip , CSRs.mie

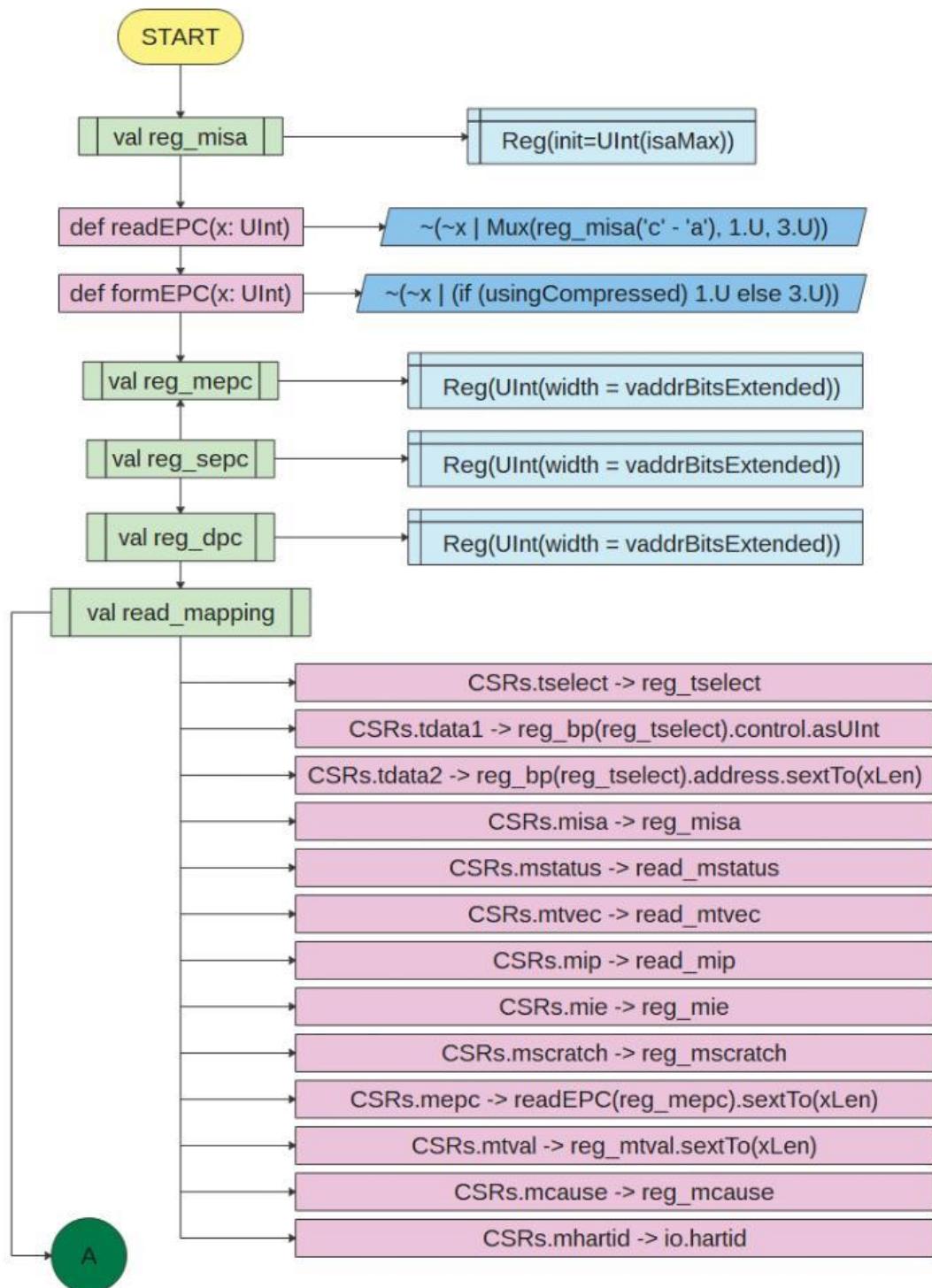
CSRs.mscratch , CSRs.mepc , CSRs.mtval , CSRs.mcause

CSRs.mhartid

with their values:

```
reg_tselect , reg_bp(reg_tselect).control.asUInt
reg_bp(reg_tselect).address.sexTo(xLen)
reg_misa , read_mstatus , read_mtvec
read_mip , reg_mie , reg_mscratch
readEPC(reg_mepc).sexTo(xLen)
reg_mtval.sexTo(xLen)
reg_mcause , io.hartid
```

respectively to use in the context of a map.



```

val debug_csrs = if (!usingDebug) LinkedHashMap() else
LinkedHashMap[Int, Bits] (
    CSRs.dcsr -> reg_dcsr.asUInt,
    CSRs.dpc -> readEPC(reg_dpc).sextTo(xLen),
    CSRs.dscratch -> reg_dscratch.asUInt)

if (usingSupervisor) {
    val read_sie = reg_mie & read_mideleg
    val read_sip = read_mip & read_mideleg
    val read_sstatus = Wire(init = 0.U.asTypeOf(new MStatus))
    read_sstatus.sd := io.status.sd
    read_sstatus.uxl := io.status.uxl
    read_sstatus.sd_rv32 := io.status.sd_rv32
    read_sstatus.mxr := io.status.mxr
    read_sstatus.sum := io.status.sum
    read_sstatus.xs := io.status_xs
    read_sstatus.fs := io.status_fs
    read_sstatus.vs := io.status_vs
    read_sstatus.spp := io.status_spp
    read_sstatus.spie := io.status_spie
    read_sstatus.sie := io.status_sie
}

```

 explanation

immutable variable named debug_csrs is initialized. if not using Debug occur, LinkedHashMap is deployed. else LinkedHashMap is deployed but with given int value and bits.

Associating the keys: CSRs.dcsr, CSRs.dpc, CSRs.dscratch

with their values: reg_dcsr.asUInt, readEPC(reg_dpc).sextTo(xLen), reg_dscratch.asUInt); respectively to use in the context of a map.

If condition with usingSupervisor. Using supervisor is a hierarchy means that the parental chain of components are responsible for detecting and correcting failures

```
def usingSupervisor: Boolean = tileParams.core.hasSupervisorMode
```

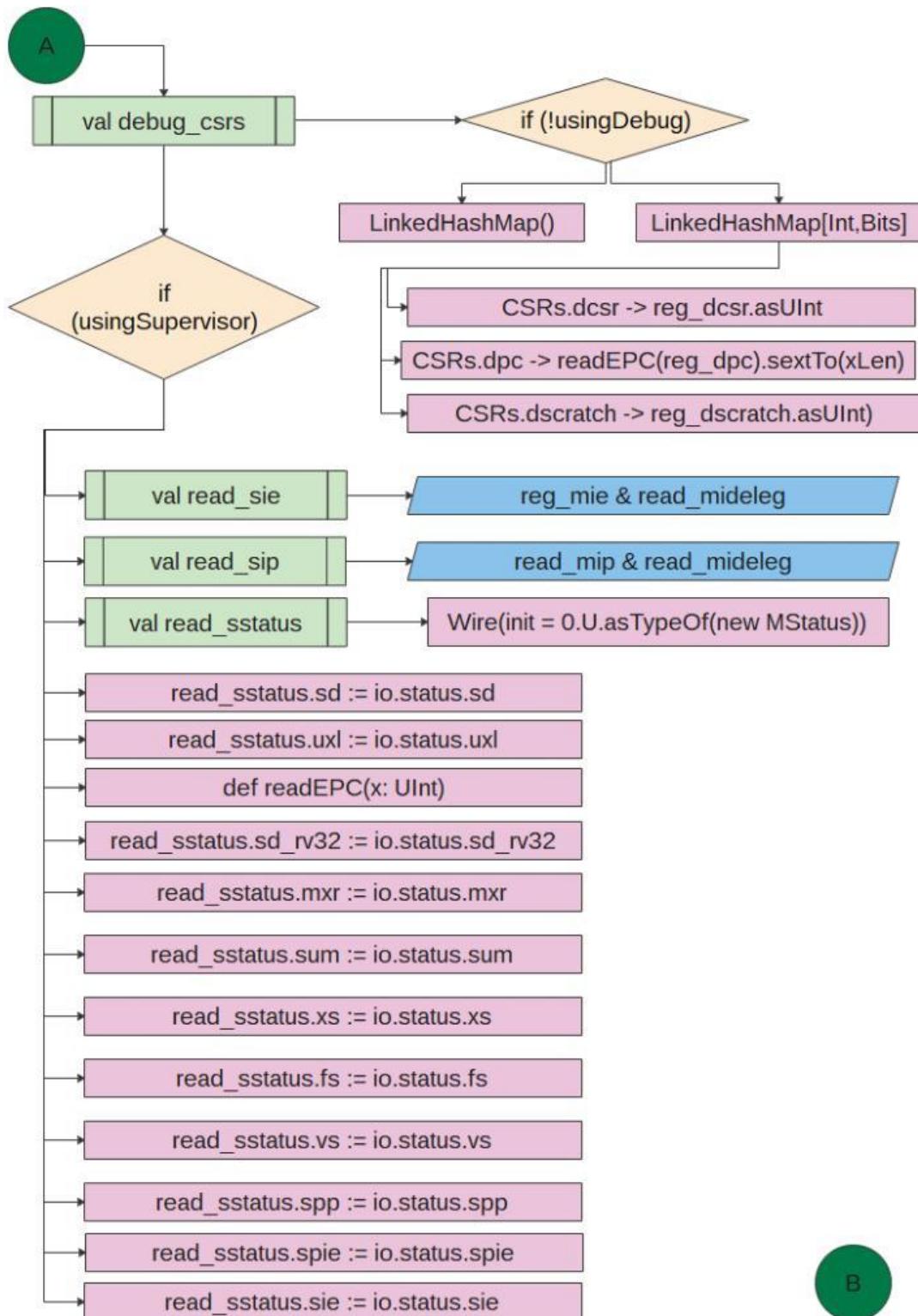
immutable variable named read_sie is initialized in which Binary AND operation is performed between reg_mie and read_mideleg

immutable variable named read_sip is initialized in which Binary AND operation is performed between read_mip and read_mideleg

immutable variable named read_sstatus is initialized with Wire initiated as 0.U as a type of MStatus object

Assign (connect) wires: read_sstatus.sd ,read_sstatus.uxl, read_sstatus.sd_rv32, read_sstatus.mxr, read_sstatus.sum, read_sstatus.xs, read_sstatus.fs, read_sstatus.vs, read_sstatus.spp, read_sstatus.spie, read_sstatus.sie

To wires: io.status.sd, io.status.uxl, io.status_sd_rv32, io.status.mxr, io.status.sum, io.status_xs, io.status_fs, io.status_vs, io.status_spp, io.status_spie, io.status_sie; Respectively.



```

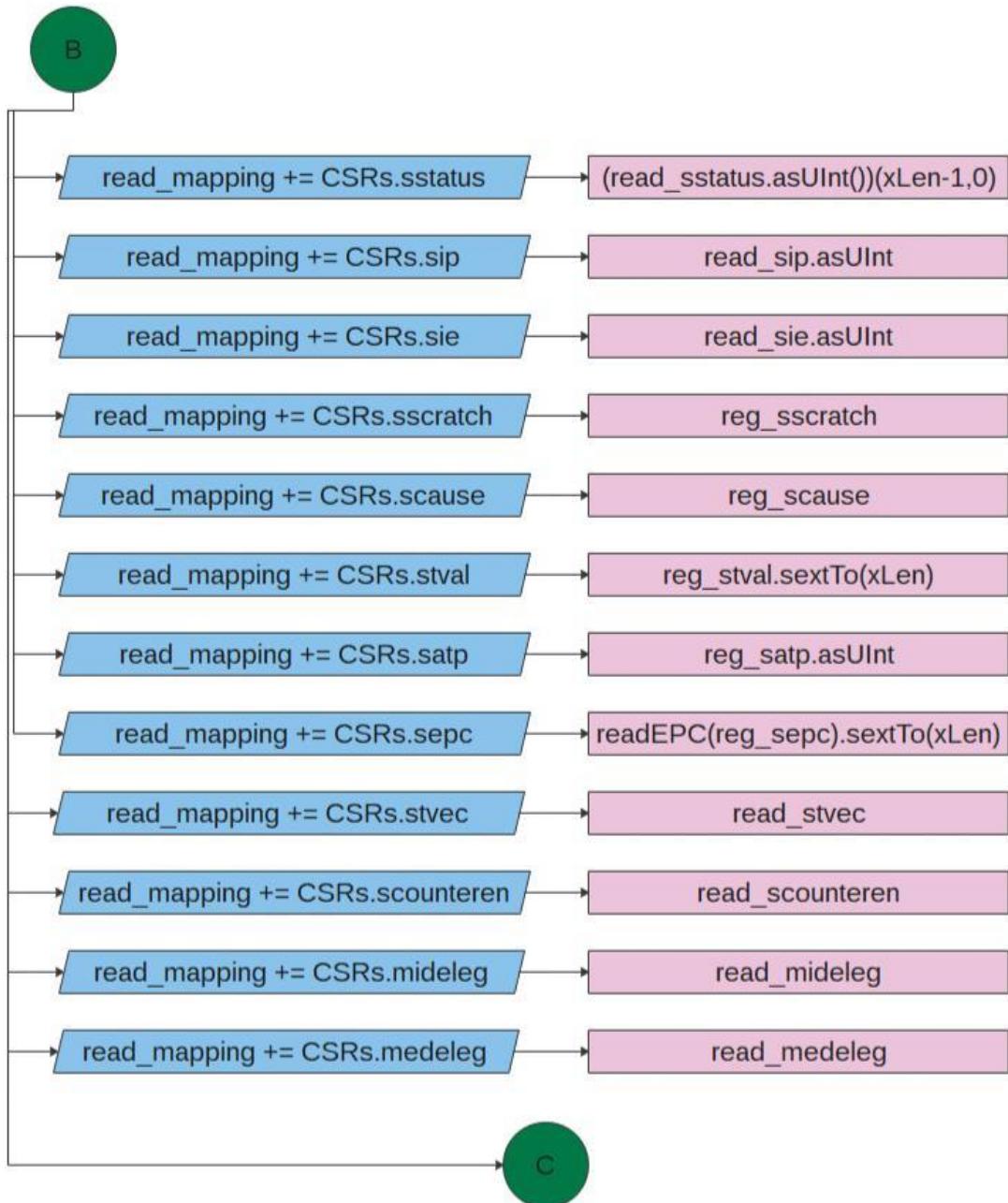
read_mapping += CSRs.sstatus -> (read_sstatus.asUInt())(xLen-1,0)
read_mapping += CSRs.sip -> read_sip.asUInt
read_mapping += CSRs.sie -> read_sie.asUInt
read_mapping += CSRs.scratch -> reg_sscratch
read_mapping += CSRs.scause -> reg_scause
read_mapping += CSRs.stval -> reg_stval.sexTo(xLen)
read_mapping += CSRs.satp -> reg_satp.asUInt
read_mapping += CSRs.sepc -> readEPC(reg_sepc).sexTo(xLen)
read_mapping += CSRs.stvec -> read_stvec
read_mapping += CSRs.scounteren -> read_scounteren
read_mapping += CSRs.mideleg -> read_mideleg
read_mapping += CSRs.medeleg -> read_medeleg
}

```

 explanation

read_mapping adds to the CSRs.sstatus and later assigns the same, associating this output (key) with its value (read_sstatus.asUInt())(xLen-1,0) to use in the context of a map.

- read_mapping adds to the CSRs.sip and later assigns the same, associating this output (key) with its value read_sip.asUInt to use in the context of a map.
 - read_mapping adds to the CSRs.sie and later assigns the same, associating this output (key) with its value read_sie.asUInt to use in the context of a map.
 - read_mapping adds to the CSRs.scratch and later assigns the same, associating this output (key) with its value reg_sscratch to use in the context of a map.
 - read_mapping adds to the CSRs.scause and later assigns the same, associating this output (key) with its value reg_scause to use in the context of a map.
 - read_mapping adds to the CSRs.stval and later assigns the same, associating this output (key) with its value reg_stval.sexTo(xLen) to use in the context of a map.
 - read_mapping adds to the CSRs.satp and later assigns the same, associating this output (key) with its value reg_satp.asUInt to use in the context of a map.
 - read_mapping adds to the CSRs.sepc and later assigns the same, associating this output (key) with its value readEPC(reg_sepc).sexTo(xLen) to use in the context of a map.
 - read_mapping adds to the CSRs.stvec and later assigns the same, associating this output (key) with its value read_stvec to use in the context of a map.
 - read_mapping adds to the CSRs.scounteren and later assigns the same, associating this output (key) with its value read_scounteren to use in the context of a map.
 - read_mapping adds to the CSRs.mideleg and later assigns the same, associating this output (key) with its value read_mideleg to use in the context of a map.
 - read_mapping adds to the CSRs.medeleg and later assigns the same, associating this output (key) with its value read_medeleg to use in the context of a map
-



```

val pc = UInt(INPUT, vaddrBitsExtended)
val epc = formEPC(io.pc)

class TracedInstruction(implicit p: Parameters) extends CoreBundle {
  val valid = Bool()
  val iaddr = UInt(width = coreMaxAddrBits)
  val insn = UInt(width = iLen)
  val priv = UInt(width = 3)
  val exception = Bool()
  val interrupt = Bool()
  val cause = UInt(width = log2Ceil(1 + CSR.busErrorIntCause))
  val tval = UInt(width = coreMaxAddrBits max iLen)
}

```

 explanation

immutable variable named pc is initialized taking input as vaddrBitsExtended in Unsigned Integer format

immutable variable named epc is initialized connecting pc to the function defined before i.e formEPC

NEW CLASS IS INITIALIZED named TracedInstruction containing implicit parameter, Scala will look if it can get an implicit value of the correct type, and if it can, pass it automatically.

immutable variable named valid is initialized contains Data types indexed by name Bool()

immutable variable named iaddr is initialized declare unassigned width

immutable variable named insn is initialized declare unassigned width

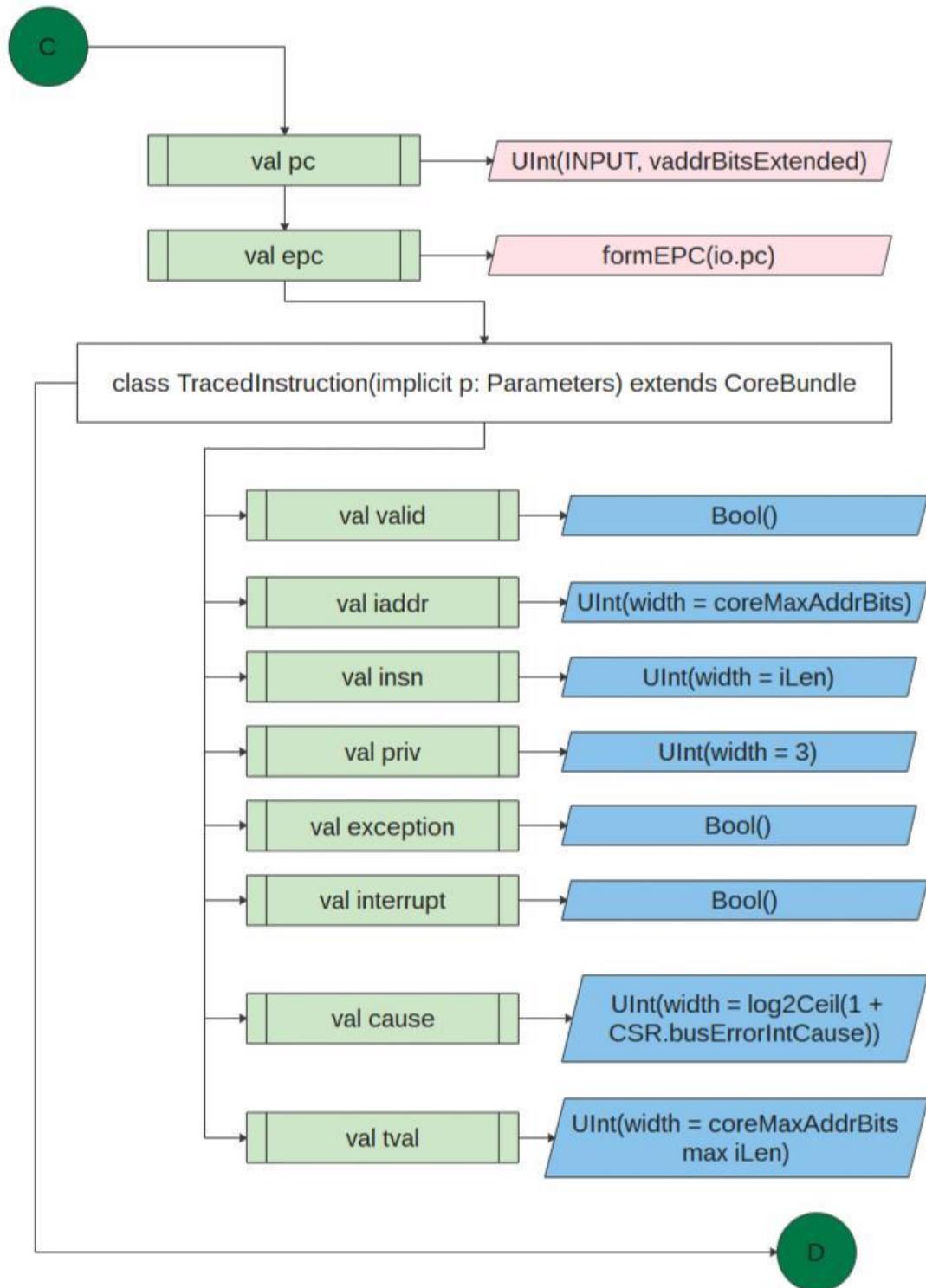
immutable variable named priv is initialized declare unassigned bit width 3

immutable variable named exception is initialized contains Data types indexed by name Bool()

immutable variable named interrupt is initialized contains Data types indexed by name Bool()

immutable variable named cause is initialized declare unassigned width. log2Ceil Compute the log2 of a Scala integer, rounded up. To get the number of bits needed to represent , using log2Ceil(1 + CSR.busErrorIntCause).

immutable variable named tval is initialized declare unassigned width



```

val exception = Bool(INPUT)

val insn_call :: insn_break :: insn_ret :: insn_cease :: insn_wfi ::  

insn_sfence :: Nil =  

  DecodeLogic(io.rw.addr << 20, decode_table(0)._2.map(x=>X),  

decode_table).map(system_insn && _.asBool)

val exception = insn_call || insn_break || io.exception

val dprv = UInt(width = PRV.SZ) // effective privilege for data accesses
val prv = UInt(width = PRV.SZ) // not truly part of mstatus, but convenient

val reg_mstatus = Reg(init=reset_mstatus)
val new_prv = Wire(init = reg_mstatus.prv)
reg_mstatus.prv := legalizePrivilege(new_prv)

val cause = UInt(INPUT, xLen)
val cause_lsbs = cause(io.trace.head.cause.getWidth-1, 0)

val delegate = Bool(usingSupervisor) && reg_mstatus.prv <= PRV.S &&  

Mux(cause(xLen-1), read_mideleg(cause_lsbs), read_medeleg(cause_lsbs))

```

 explanation

Exception variable of Boolean type is initialized as input

8 variables defined with DecodeLogic, which returns a list of 8 values.

Then value of exception is reassigned by applying OR operation between insn_call,insn_break and io.exception .

dprv variable is initialized with UInt of width PRV.SZ

prv variable is initialized with UInt of width PRV.SZ

PRV.SZ is come from PRV object which is created in CSR.scala file.

```

object PRV
{
  val SZ = 2
  val U = 0
  val S = 1
  val H = 2
  val M = 3
}

```

Reg_mstatus is a register initialized with reset_mstatus value.

```
val reset_mstatus = Wire(init=new MStatus().fromBits(0))
```

In reset_mstatus , wire function is called which is chisel function for wiring hardware.
In this function, object of MStatus class is created with all bits 0.

In new_prv variable Wire function is called and reg_mstatus .prv is passed as a parameter. As reg_mstatus becomes an object of class MStatus so we can access its attributes by object name.attribute_name .

Then, reg-mstatus is wired with legalizePrivilege function with new-prv as a parameter.

```
def legalizePrivilege(priv: UInt) : UInt =
  if (usingSupervisor) Mux(priv === PRV.H, PRV.U, priv)
  else if (usingUser) Fill(2, priv(0))
  else PRV.M
```

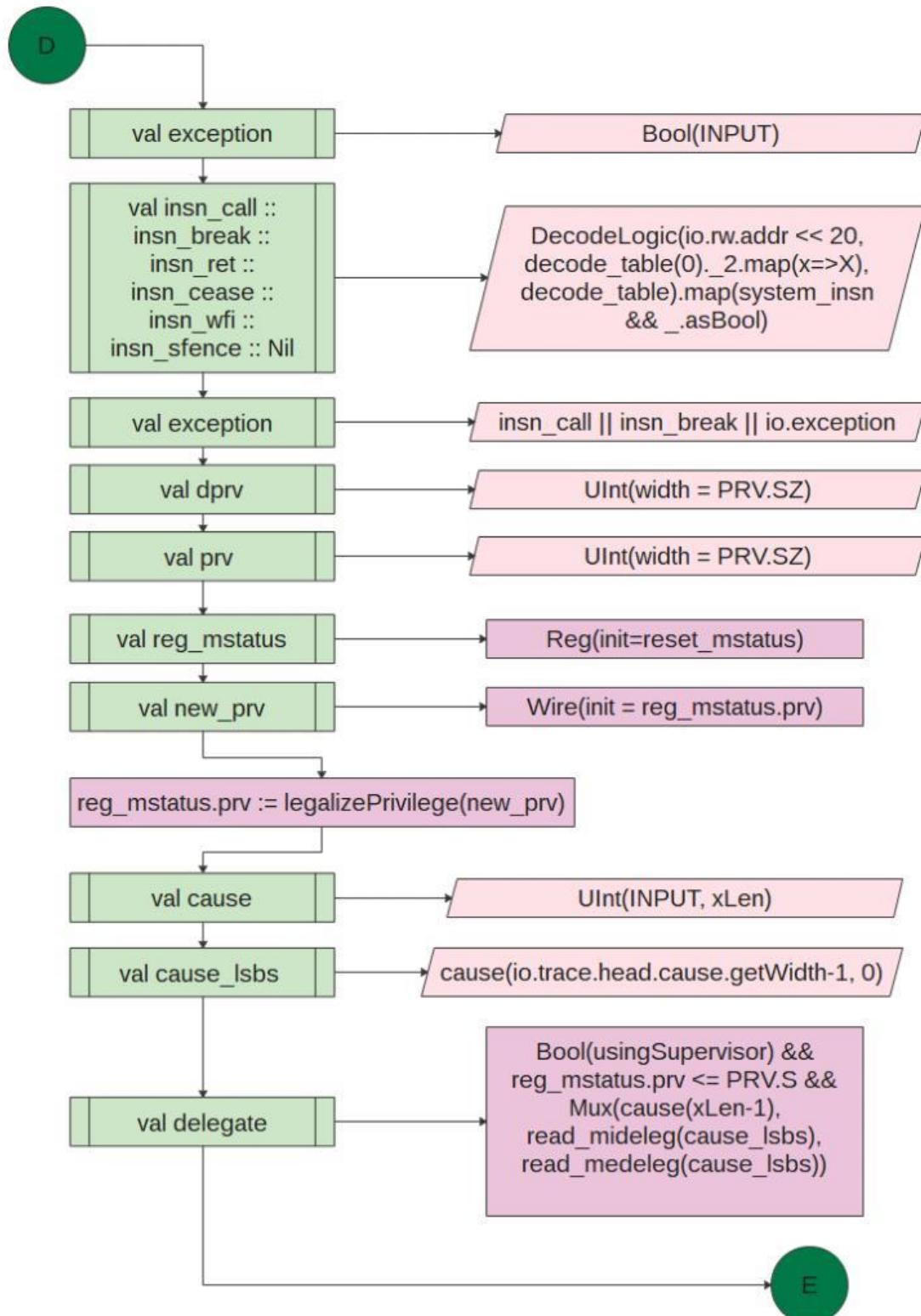
Priv is initialized in TracedInstruction class

Unsigned Integer cause is initialized with 2 parameters.

In cause_lbs, pass two values to cause variable input and xlen.

In delegate, And operation is performed between usingSupervisor, reg_mstatus.prv <= PRV.S && Mux(cause(xLen-1), read_mideleg(cause_lsb), read_medeleg(cause_lsb)). If all are True then True value is assigned to delegate.

```
val (reg_mideleg, read_mideleg) = {
  val reg = Reg(UInt(xLen.W))
  (reg, Mux(usingSupervisor, reg & delegable_interrupts, 0.U))
}
```



```

when (exception) {
    when (trapToDebug) {
        when (!reg_debug) {
            reg_debug := true
            reg_dpc := epc
            reg_dcsr.cause := Mux(reg_singleStepped, 4, Mux(causeIsDebugInt, 3,
Mux[UInt](causeIsDebugTrigger, 2, 1)))
            reg_dcsr.prv := trimPrivilege(reg_mstatus.prv)
            new_prv := PRV.M
        }
    }.elsewhen (delegate) {

```

 explanation

Exception, trapdebug and reg_debug all are initialized as bool type.

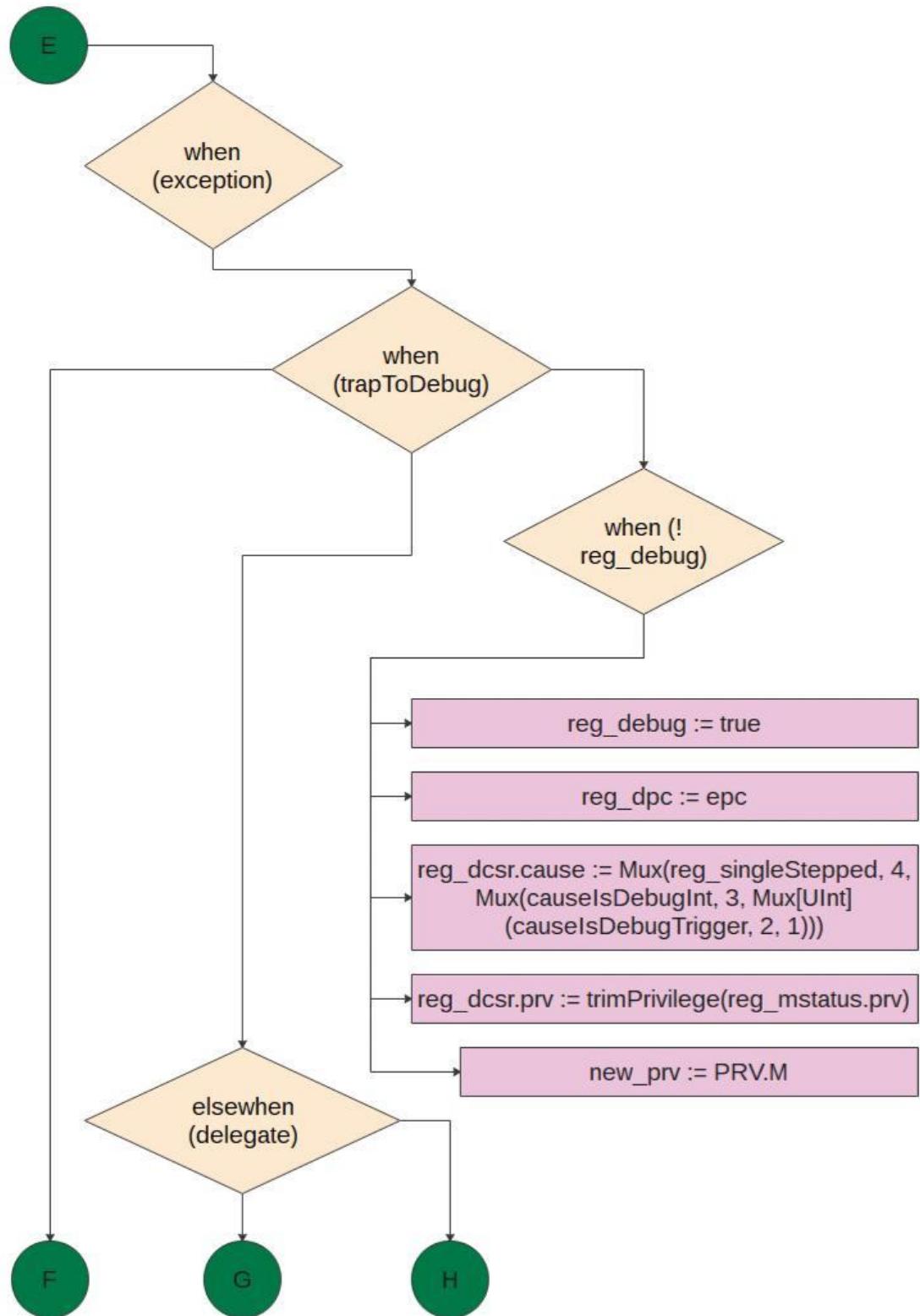
If exception, trapdebug are true and reg_debug is false then reg_debug wired as true, reg_pc is wired with epc, reg_dcse.cause is wired with MUX, reg_dcr.prv is wired with function trimPrivilege and new_prv wired with PRV.M.

```

def trimPrivilege(priv: UInt) : UInt =
    if (usingSupervisor) priv
    else legalizePrivilege(priv)

```

If above condition is false and delegate which is also a bool type is true then conditions under it execute.



```

reg_sepc := epc
reg_scause := cause
xcause_dest := sCause
reg_stval := io.tval
reg_mstatus.spie := reg_mstatus.sie
reg_mstatus.spp := reg_mstatus.prv
reg_mstatus.sie := false
new_prv := PRV.S
}.otherwise {
    reg_mepc := epc
    reg_mcause := cause
    xcause_dest := mCause
    reg_mtval := io.tval
    reg_mstatus.mpie := reg_mstatus.mie
    reg_mstatus.mpp := trimPrivilege(reg_mstatus.prv)
    reg_mstatus.mie := false
    new_prv := PRV.M
}
}

```

 explanation

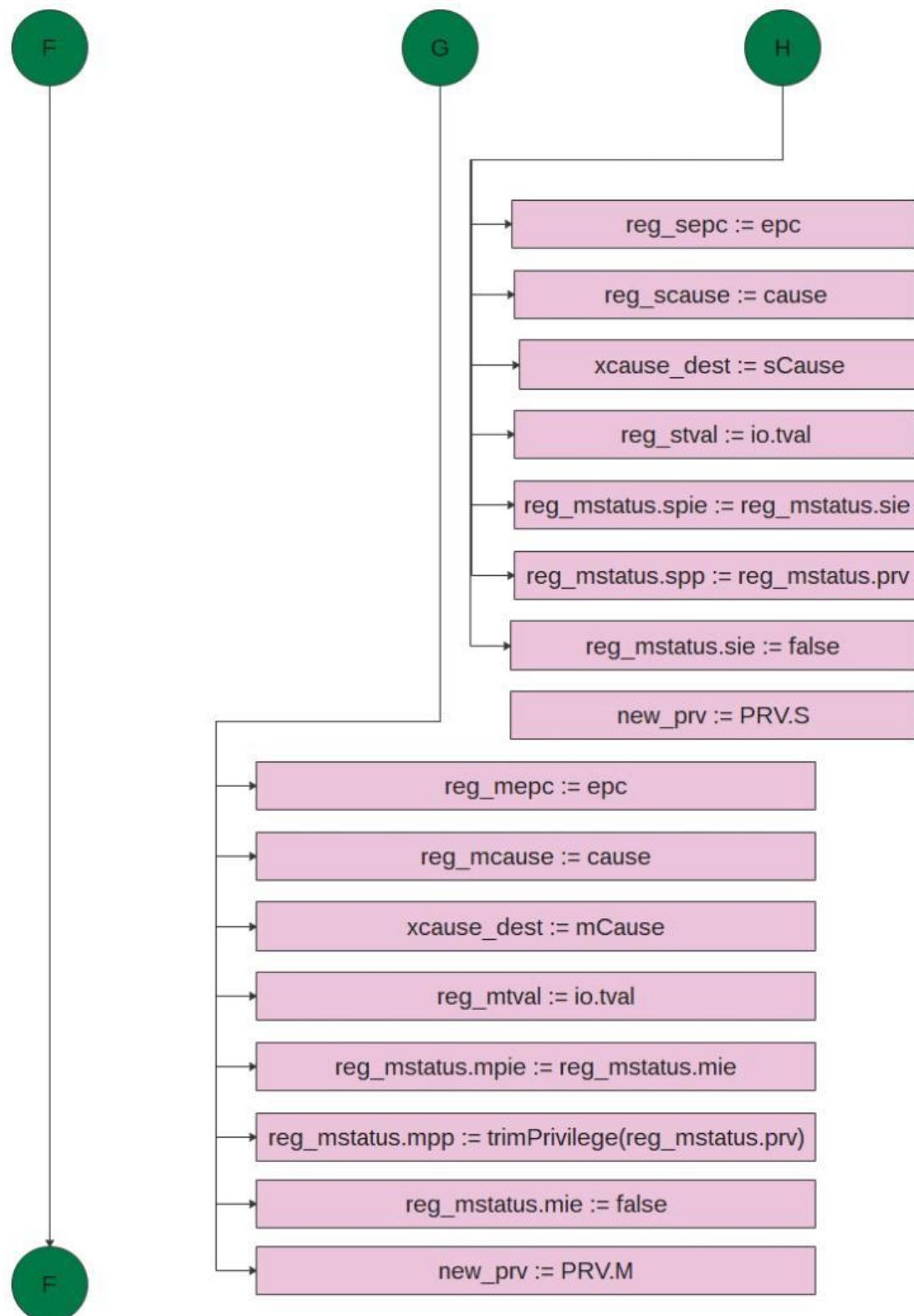
If delegate is true then reg_sepc, reg_scause,xcause_dest,reg_stval,reg, mstatus.spie, reg_mstatus.spp, reg_mstatus.sie, new_prv are wired with epc.cause.sCause, io.tval, reg_mstatus.sie, reg_mstatus.prv, false, PRV.S respectively.

If above both conditions are False then otherwise condition is execute

In otherwise condition, reg_mepc, reg_mcause, xcause_dest, reg_mtval, reg_mstatus.mpie, reg_mstatus.mpp, reg_mstatus.mie, new_prv is wired with epc, cause, mCause ,io.tval ,reg_mstatus.mie, trimPrivilege(reg_mstatus.prv) ,false,PRV.M respectively.

io.tval is unsigned integer initialized in class TracedInstruction.

```
val tval = UInt(width = coreMaxAddrBits max iLen)
```



```

val evec = UIInt(OUTPUT, vaddrBitsExtended)

when (insn_ret) {
    when (Bool(usingSupervisor) && !io.rw.addr(9)) {
        reg_mstatus.sie := reg_mstatus.spie
        reg_mstatus.spie := true
        reg_mstatus.spp := PRV.U
        new_prv := reg_mstatus.spp
        io.evec := readEPC(reg_sepc)
    }.elsewhen (Bool(usingDebug) && io.rw.addr(10)) {

```

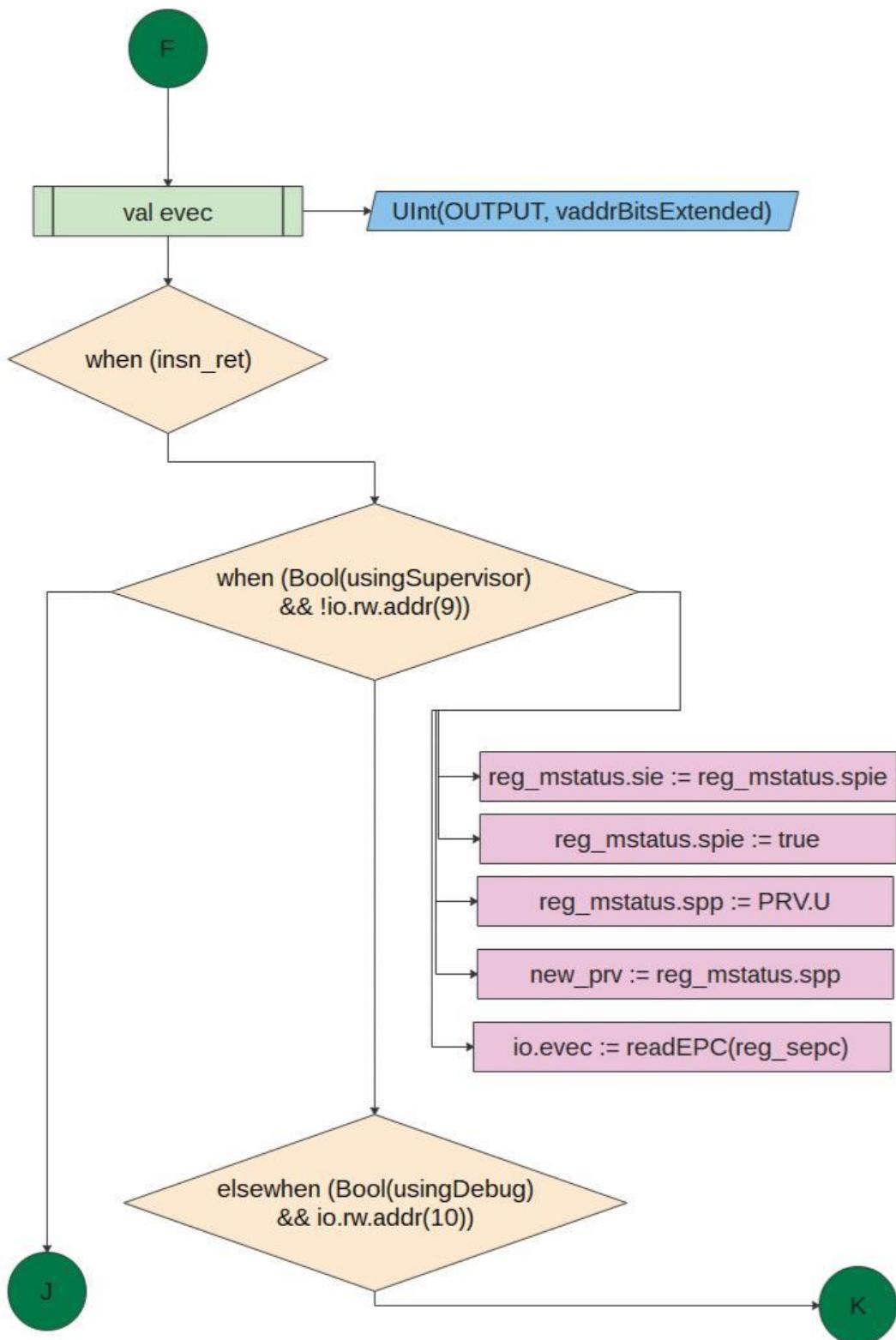
explanation

val evec is initialized as unsigned integer.

Insn_ret variable is initialized as bool type

If insn_ret is true and usingsupervisor and rw.adder is not equal to 9 then each variable is wired with respective one as shown above.

If above condition is False and using debug is ture and io.re.adder is 10 then each variable is wired with respective one as shown below.



```

new_prv := reg_dcsr.prv
reg_debug := false
io.evec := readEPC(reg_dpc)
}.otherwise {
    reg_mstatus.mie := reg_mstatus.mpie
    reg_mstatus.mpie := true
    reg_mstatus.mpp := legalizePrivilege(PRV.U)
    new_prv := reg_mstatus.mpp
    io.evec := readEPC(reg_mepc)
}

val decoded_addr = read_mapping map { case (k, v) => k -> (io.rw.addr ===
k) }

when (decoded_addr(CSRs.mepc)) { reg_mepc := formEPC(wdata) }

```

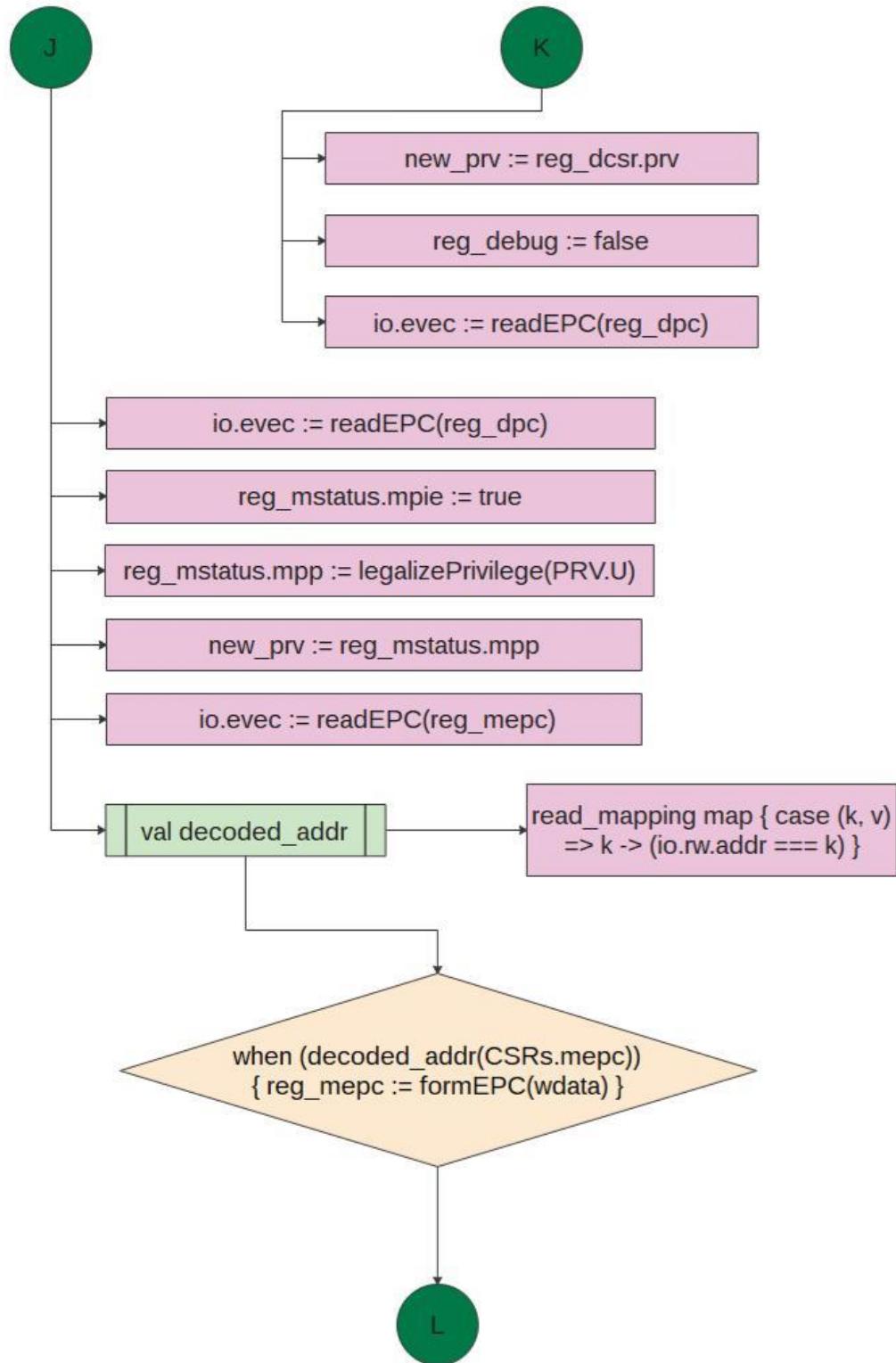
explanation

If above both conditions are False then otherwise condition is execute. In otherwise condition, each variable is wired with respective one as shown above.

In decoded_Addr, map is read and found the variable which have k as a key and value is equal to True is io.rw.addr and k are UInt and if both are not Unit then it will be false.

decoded_addr have a bool value.

If decoded_addr is True when CSRs.mepc is read from map then reg_mepc is wired with function formEPC with wdata as a parameter.



```

when (decoded_addr(CSRs.dpc))      { reg_dpc := formEPC(wdata) }

when (decoded_addr(CSRs.sepc))      { reg_sepc := formEPC(wdata) }

val wdata = Bits(INPUT, xLen)
val wdata = readModifyWriteCSR(io.rw.cmd, io.rw.rdata, io.rw.wdata)

val uepc = 0x41
val sepc = 0x141
val vsepc = 0x241
val mepc = 0x341

val all = {
    val res = collection.mutable.ArrayBuffer[Int]()
}

res += uepc
res += sepc
res += vsepc
res += mepc

```

— explanation —

If decoded_addr is True when CSRs.dpc is read from map then reg_dpc is wired with function formEPC with wdata as a parameter.

If decoded_addr is True when CSRs.sepc is read from map then reg_sepc is wired with function formEPC with wdata as a parameter.

wdata is initialized as Bits data type of chisel.

In wdata, readModifyWriteCSR function is called which have 3 parameters of unsigned integer, io.rw.cmd, io.rw.rdata, io.rw.wdata.

```

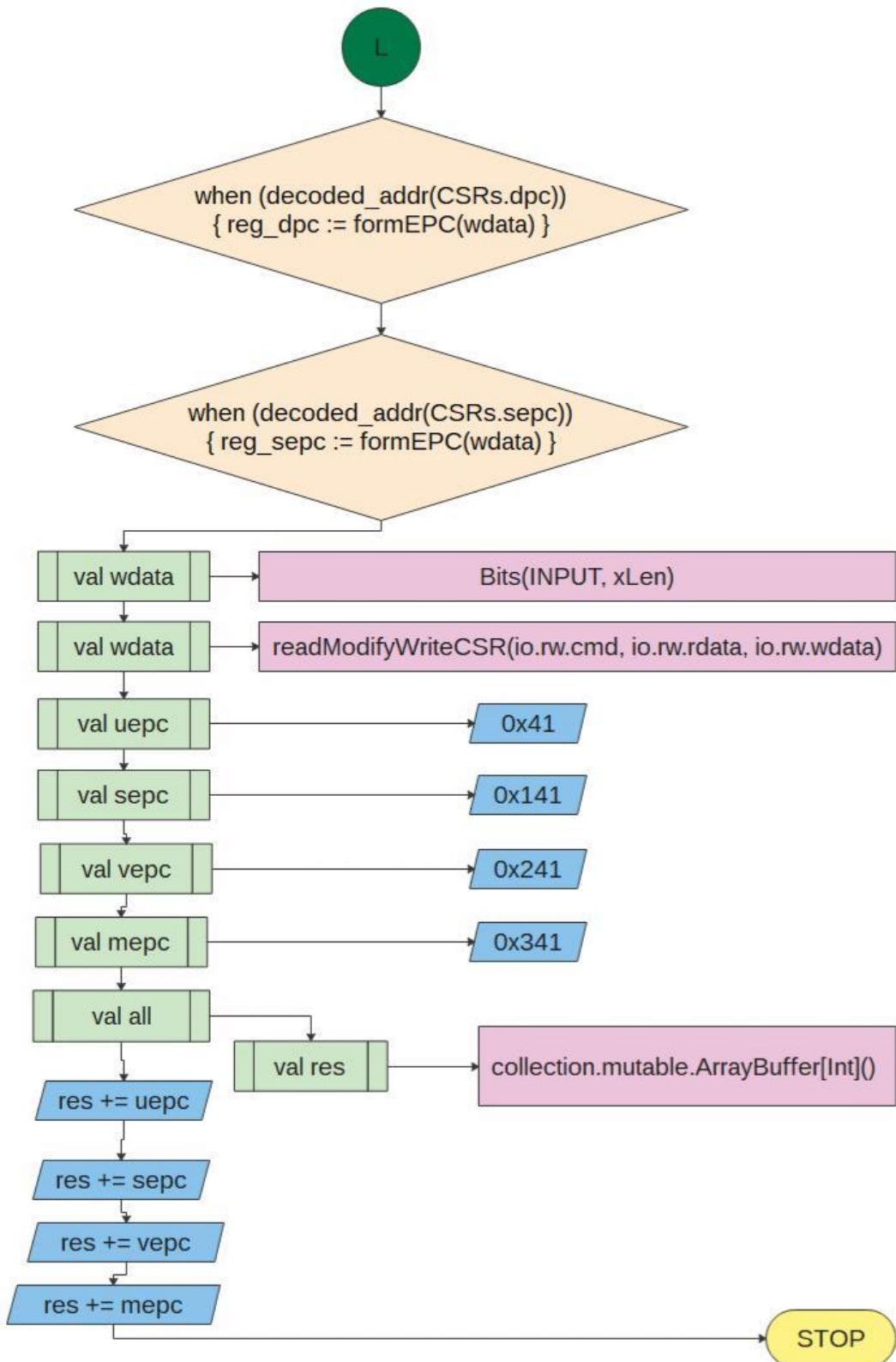
def readModifyWriteCSR(cmd: UInt, rdata: UInt, wdata: UInt) = {
    Mux(cmd(1), rdata, UInt(0) | wdata) & ~Mux(cmd(1,0).andR, wdata,
    UInt(0))
}

```

Upec, sepc, vsepc and mepc are assigned values 0*41, 0*141, 0*241, 0*341 respectively.

Res is an array in which uepc, sepc, vsepc and mepc are append.

This res array is placed in variable all.



NPC CHECK

DEEP DIVE INTO CODE

Explanation (By Means of Flowcharts) :

```
val npc = UInt(INPUT, width = vaddrBitsExtended)
val predicted_npc = Wire(init = ntpc)
val npc = Mux(s2_replay, s2_pc, predicted_npc)
s1_pc := io.cpu.npc
```

explanation

In the first line of code value of npc is being declared as a virtual address which consists of two parts a page and an offset into that page

In the second line of code value of predicted npc is being declared which is wired to ntpc. Ntpc is calculated in the upper code block by adding s1 base pc and fetched bytes. Fetchbytes is a function created in file Core.scala (<https://github.com/chipsalliance/rocket-chip/blob/master/src/main/scala/tile/Core.scala>) as:

```
def fetchBytes: Int = fetchWidth * instBytes
```

According to which the fetched width is summed up with instruction bytes

```
def instBytes: Int = instBits / 8
```

The instruction bytes in the fetch bytes function comes by dividing instruction bits by 8. Where instruction bits are originally declared as 16 when the instruction is compressed, else 32

```
val instBits: Int = if (useCompressed) 16 else 32
val instBits: Int
```

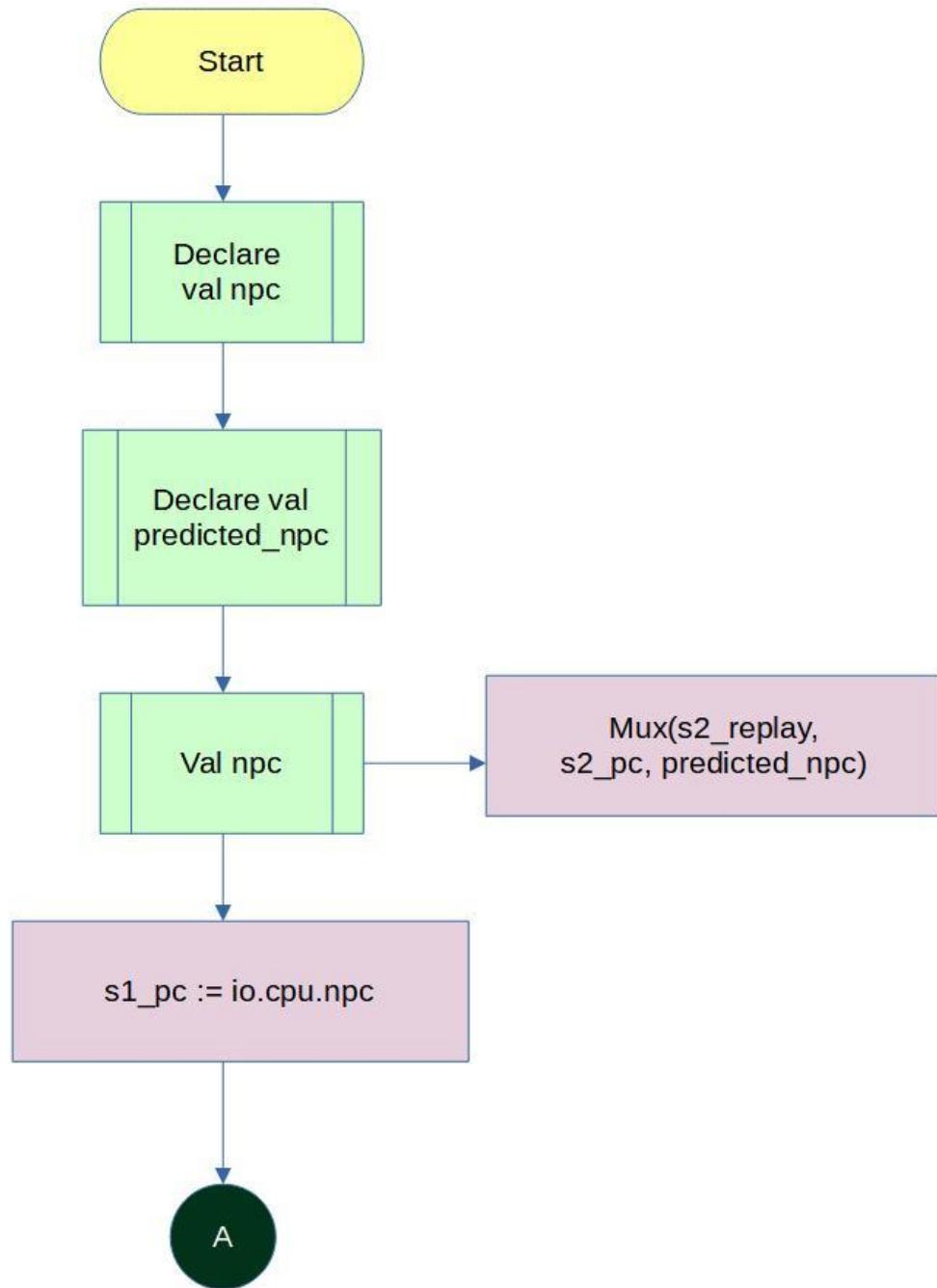
fetched width in the fetched byte function is set as 2 if instruction is compressed else 1.

```
val fetchWidth: Int = if (useCompressed) 2 else 1
//fetchWidth doubled, but coreInstBytes halved, for RVC:
val decodeWidth: Int = fetchWidth / (if (useCompresses) 2 else 1)
```

Value of npc is determined by a mux in the third line of code, the first parameter is the select pin which is declared as Boolean in the start. If branch prediction is taken then npc=predicted targeted address caculated as predicted_npc else npc= s2_pc(s2_pc is wired with s1_pc which has the original value of npc referenced via cpu)

```
s2_pc := s1_pc
```

In the fourth line of code, s1_pc has been assigned the value of npc referenced via cpu



```
icache.io.req.bits.addr := io.cpu.npc
io.cpu.npc := alignPC(Mux(io.cpu.req.valid, io.cpu.req.bits.pc, npc))

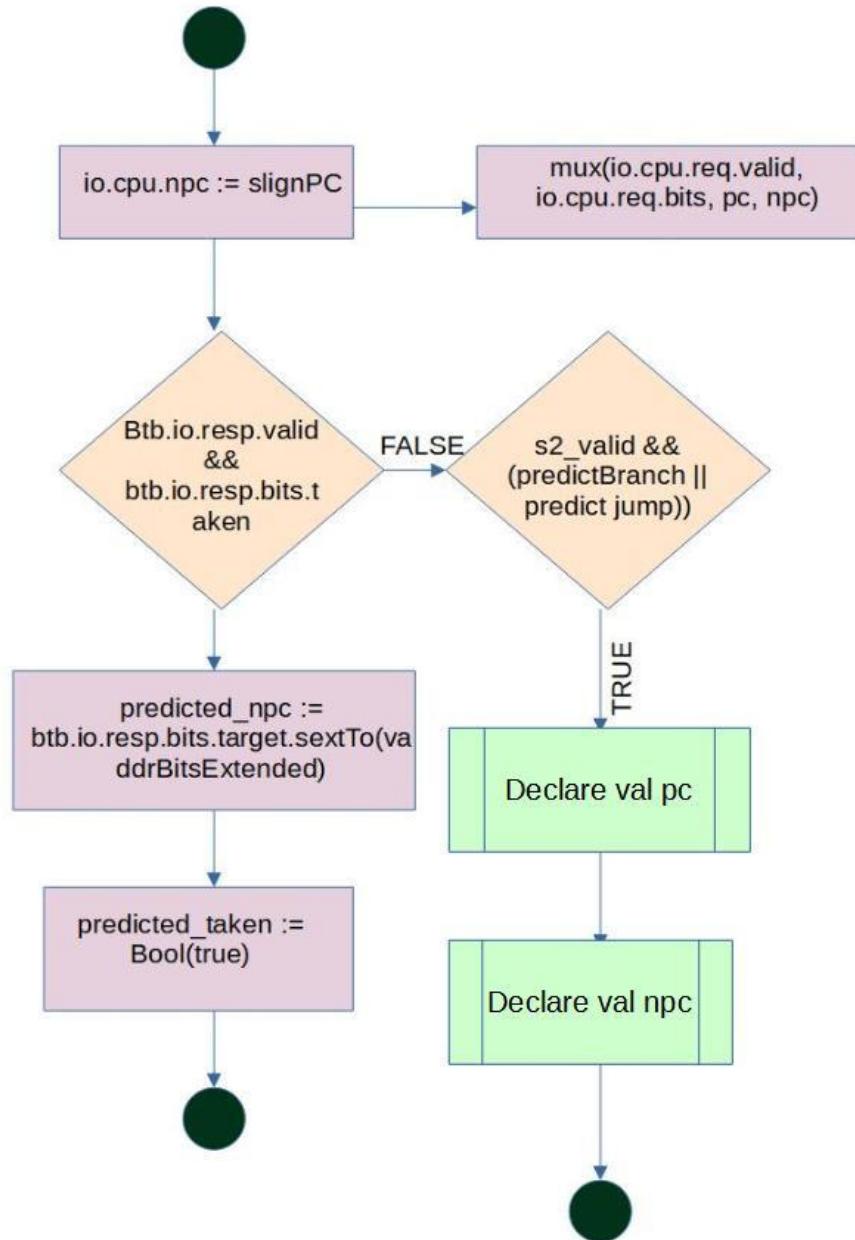
when (btb.io.resp.valid && btb.io.resp.bits.taken) {
    predicted_npc := btb.io.resp.bits.target.sexTo(vaddrBitsExtended)
    predicted_taken := Bool(true)
}
```

— explanation —

Address bits that have been requested from CPU for instruction cache, are being assigned the value of npc referenced through cpu

Io.cpu.npc is being assigned the aligned value via a mux. The value that can be assigned include either request valid referenced with cpu or requested pc bits referenced with cpu or npc itself

If the response of branch table buffer is valid and response bits are taken, values have been assigned to predicted npc also predicted_taken pin is turned on



```

when (s2_valid && (predictBranch || predictJump)) {
    val pc = s2_base_pc | (idx*coreInstBytes)
    val npc =
        if (idx == 0) pc.assInt + Mux(prevRVI, rviImm -& 2.S, rvcImm)
        else Mux(prevRVI, pc - coreInstBytes, pc).assInt +
    Mux(prevRVI, rviImm, rvcImm)
    predicted_npc := npc.asUInt
}

when (useRAS) {
    predicted_npc := btb.io.ras_head.bits
}

```

 explanation

when s2 is valid that is the pin is true, and either predicted branch or predicted jump is taken the values of pc, npc and predicted_npc has been assigned

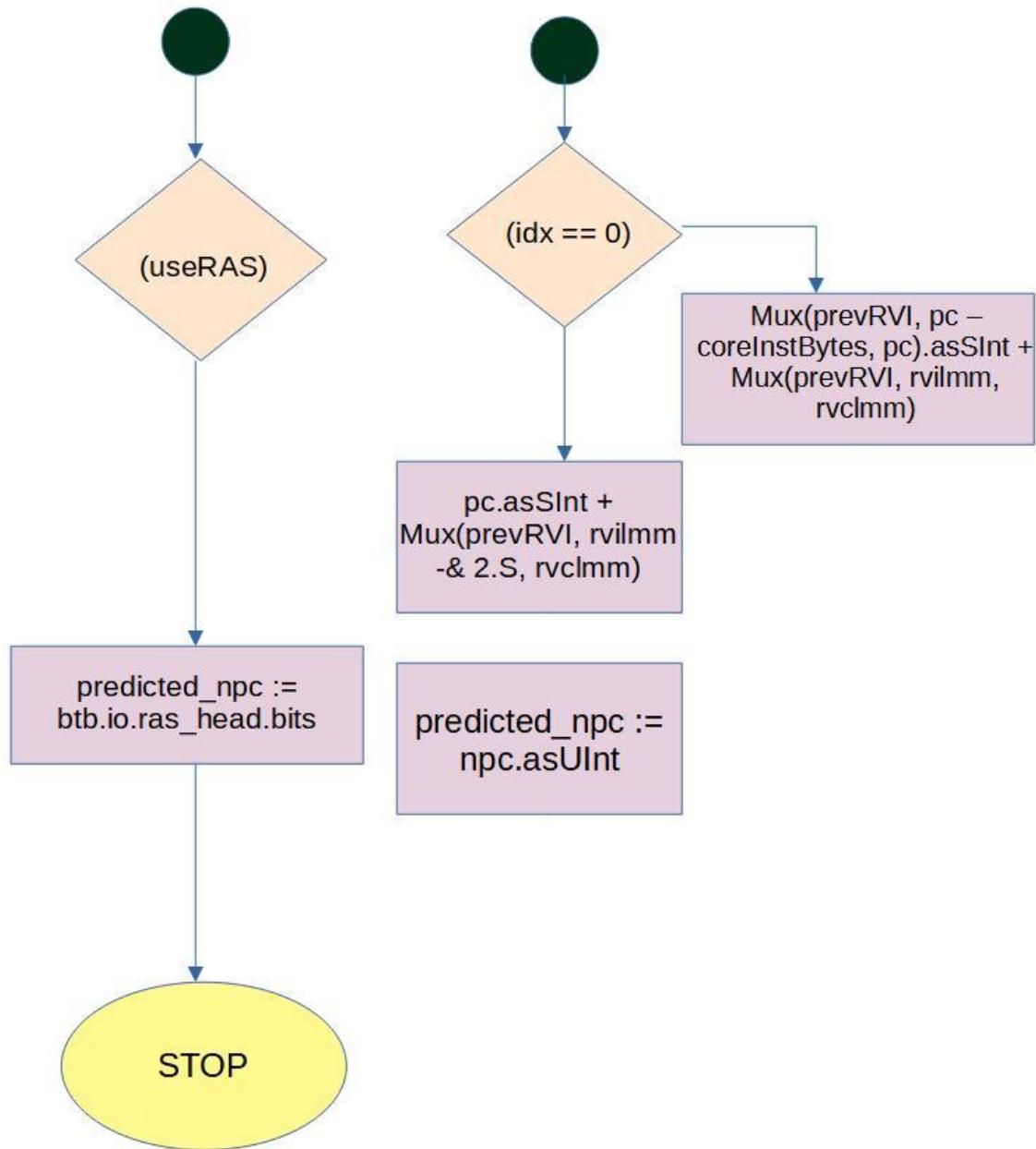
```

when (useRAS) {
    predicted_npc := btb.io.ras_head.bits
}

```

RAS: Return Address Stack

During the instruction fetch, the NLP takes the current Fetch-PC (the current program counter that fetches the fetch packet of instruction) as input and works together with a branch history table (BHT), a branch target buffer (BTB) and a return address stack (RAS) depending on what kind of instruction is being speculated (conditional, unconditional or a return instruction).



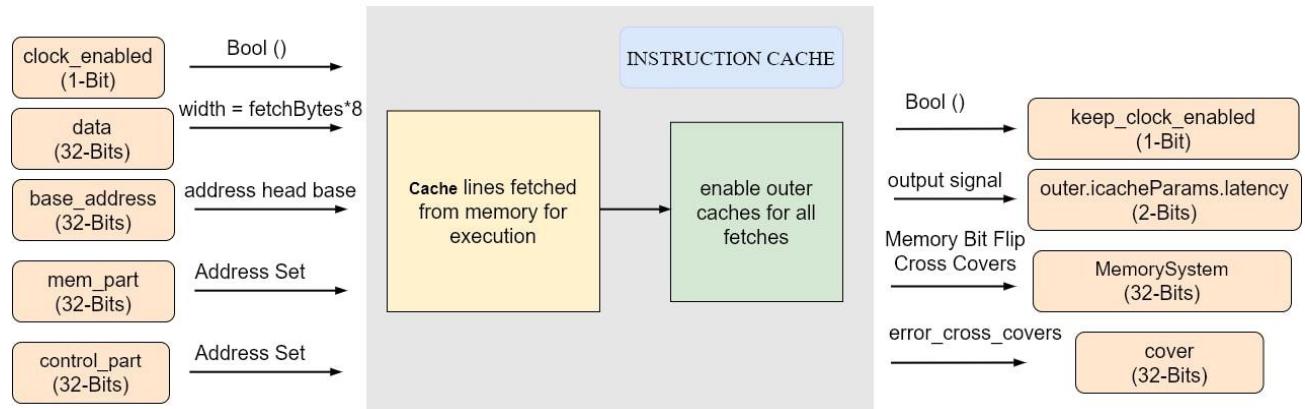
FETCH STAGE

The Fetch Stage contains the following Modules.

- ICache
- Branch Target Buffer
- TLB
- TLB Permissions
- Hella Cache IF

Instruction Cache (ICache)

Block Diagram:



DEEP DIVE INTO CODE

Explanation (By Means of Flowcharts) :

```
case class ICacheParams (
    nSets: Int = 64,
    nWays: Int = 4,
    rowBits: Int = 128,
    nTLBSets: Int = 1,
    nTLBWays: Int = 32,
    nTLBBasePageSectors: Int = 4,
    nTLBSuperpages: Int = 4,
    cacheIdBits: Int = 0,
    tagECC: Option[String] = None,
    dataECC: Option[String] = None,
    itimAddr: Option[BigInt] = None,
    prefetch: Boolean = false,
    blockBytes: Int = 64,
    latency: Int = 2,
    fetchBytes: Int = 4)
```

— explanation —

Scala case classes are just regular classes which are immutable by default and decomposable through pattern matching.

In case class ICacheParams, different parameters with different data types and values are passed.

nsets has data type integer and has value 64.

nWays has data type integer and has value 4.

rowBits has data type integer and has value 128.

nTLBSets has data type integer and has value 1.

nTLBWays has data type integer and has value 32.

nTLBBasePageSectors has data type integer and has value 4.

nTLBSuperpages has data type integer and has value 4.

cacheldBits has data type integer and has value 0.

tagECC is defined as option[string] which means it may have string value or otherwise it is equal to none.

dataECC is defined as option[string] which means it may have string value or otherwise it is equal to none.

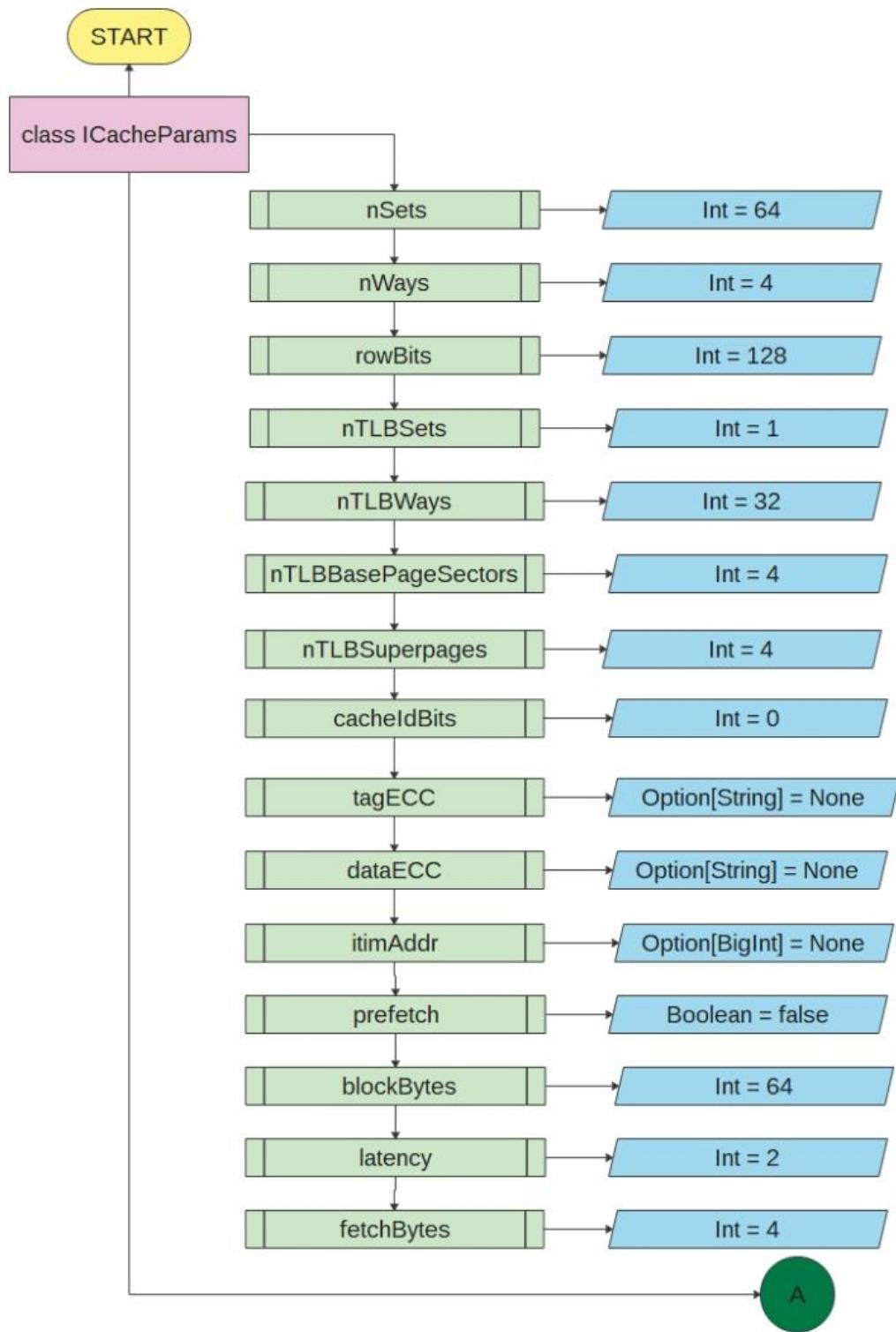
itimAddr is defined as option[BigInt] which means it may have integer value or otherwise it is equal to none.

Prefetch has data type Boolean and has value False.

blockBytes has data type integer and has value 64.

latency has data type integer and has value 2.

fetchBytes has data type integer and has value 4.



```

trait HasL1ICacheParameters extends HasL1CacheParameters with
HasCoreParameters {
    val cacheParams = tileParams.icache.get
}
class ICacheReq(implicit p: Parameters) extends CoreBundle()(p) with
HasL1ICacheParameters {
    val addr = UInt(width = vaddrBits)
}
class ICacheErrors(implicit p: Parameters) extends CoreBundle()(p)
with HasL1ICacheParameters
with CanHaveErrors {
    val correctable = (cacheParams.tagCode.canDetect ||
cacheParams.dataCode.canDetect).option(Valid(UInt(width = paddrBits)))
}

```

 explanation

trait class is just like a abstract class in java.

Trait class name HasL1ICacheParameters extends with its own parameters with HasCoreParameters which is trait use in other files.

cacheParams is initialized wit the value tileParams.icache.get.

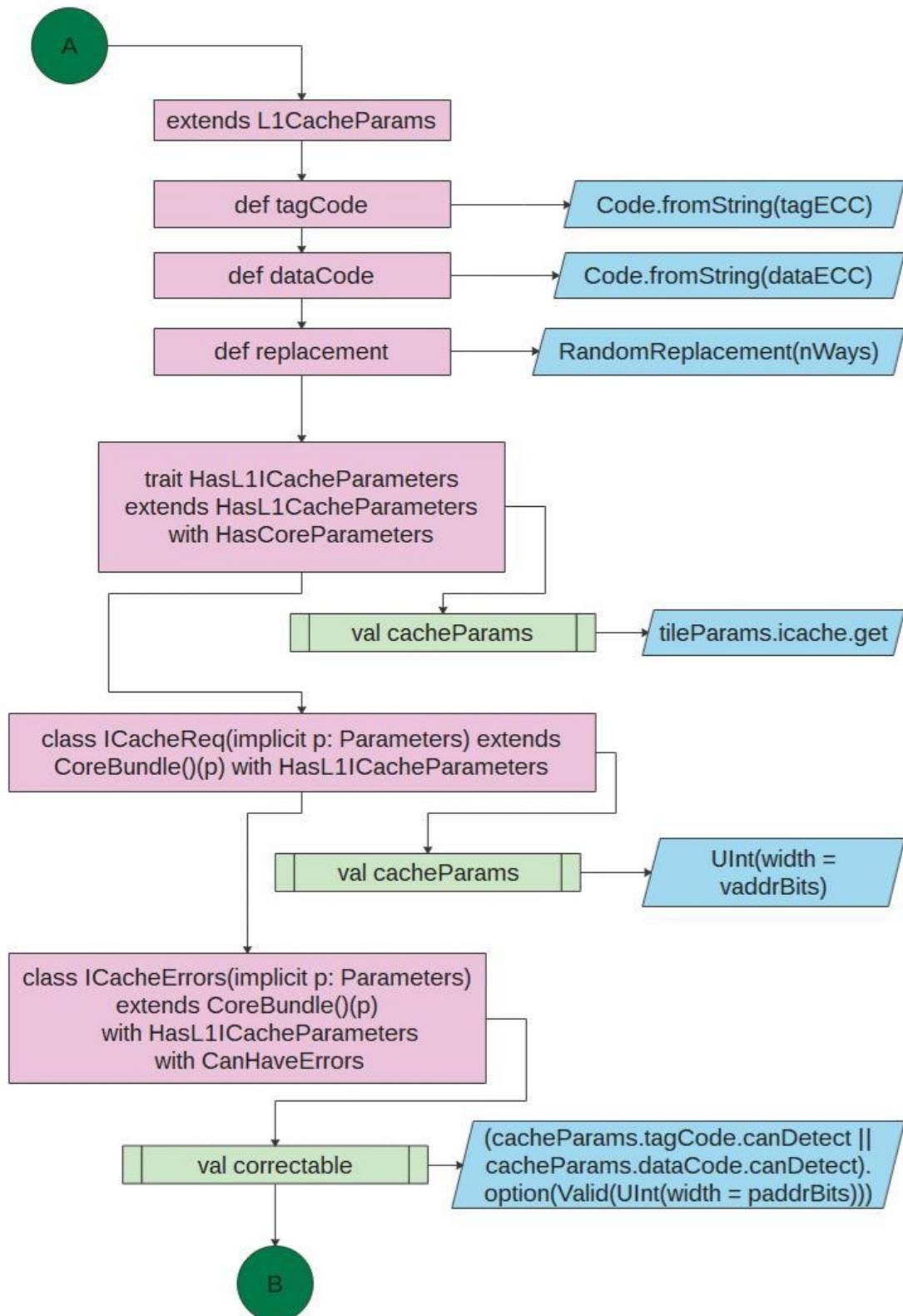
TileParams is a scala build-in library which import at the starting of ICache file from class tileParams which get value of icache which is assigned to variable cacheParams.

Class ICacheReg extends with core bundle and HasL1ICacheParameters class.

In ICacheReg class, addr variable is initialized as unsigned integer with width of vaddrBits.

Class ICacheErrors extends with core bundle and with HasL1ICacheParameters and CanHaveErrors class.

Assigned value to correctable variable by performing OR operation between cacheParams.tagCode.canDetect and cacheParams.dataCode.canDetect with width of paddrBits.



```

val uncorrectable = (cacheParams.itimAddr.nonEmpty &&
cacheParams.dataCode.canDetect).option(Valid(UInt(width = paddrBits)))
val bus = Valid(UInt(width = paddrBits))
}
class ICache(val icacheParams: ICacheParams, val
staticIdForMetadataUseOnly: Int)(implicit p: Parameters) extends LazyModule
{
  lazy val module = new ICacheModule(this)
  val hartIdSinkNodeOpt = icacheParams.itimAddr.map(_ =>
BundleBridgeSink[UInt]())
  val mmioAddressPrefixSinkNodeOpt = icacheParams.itimAddr.map(_ =>
BundleBridgeSink[UInt]())
  val useVM = p(TileKey).core.useVM
  val masterNode = TLClientNode(Seq(TLMasterPortParameters.v1(
    clients = Seq(TLMasterParameters.v1(
      sourceId = IdRange(0, 1 + icacheParams.prefetch.toInt), // 0=refill,
      1=hint
      name = s"Core ${staticIdForMetadataUseOnly} ICache"),
      requestFields = useVM.option(Seq()).getOrElse(Seq(AMBAProtField()))))))
```

explanation

Assigned value to uncorrectable variable by performing AND operation between cacheParams.itimAddr.nonEmpty and cacheParams.dataCode.canDetect with width of paddrBits.

Declared bus variable as unsigned integer of width paddrBits.

Class ICache initialized with two parameters icacheParams: ICacheParams, staticIdForMetadataUseOnly:Int and extends with LazyModule.

lazy keyword changes the val to get lazily initialized. Lazy initialization means that whenever an object creation seems expensive, the lazy keyword can be stuck before val.

Module variable get lazy initialized. In module variable, create the object of ICache class with parameter this. "this" keyword is used to fetch the variable of the current class.

hartIdSinkNodeOpt variable map with BundleBridgeSInk, which is to be broadcast to units within the tile.

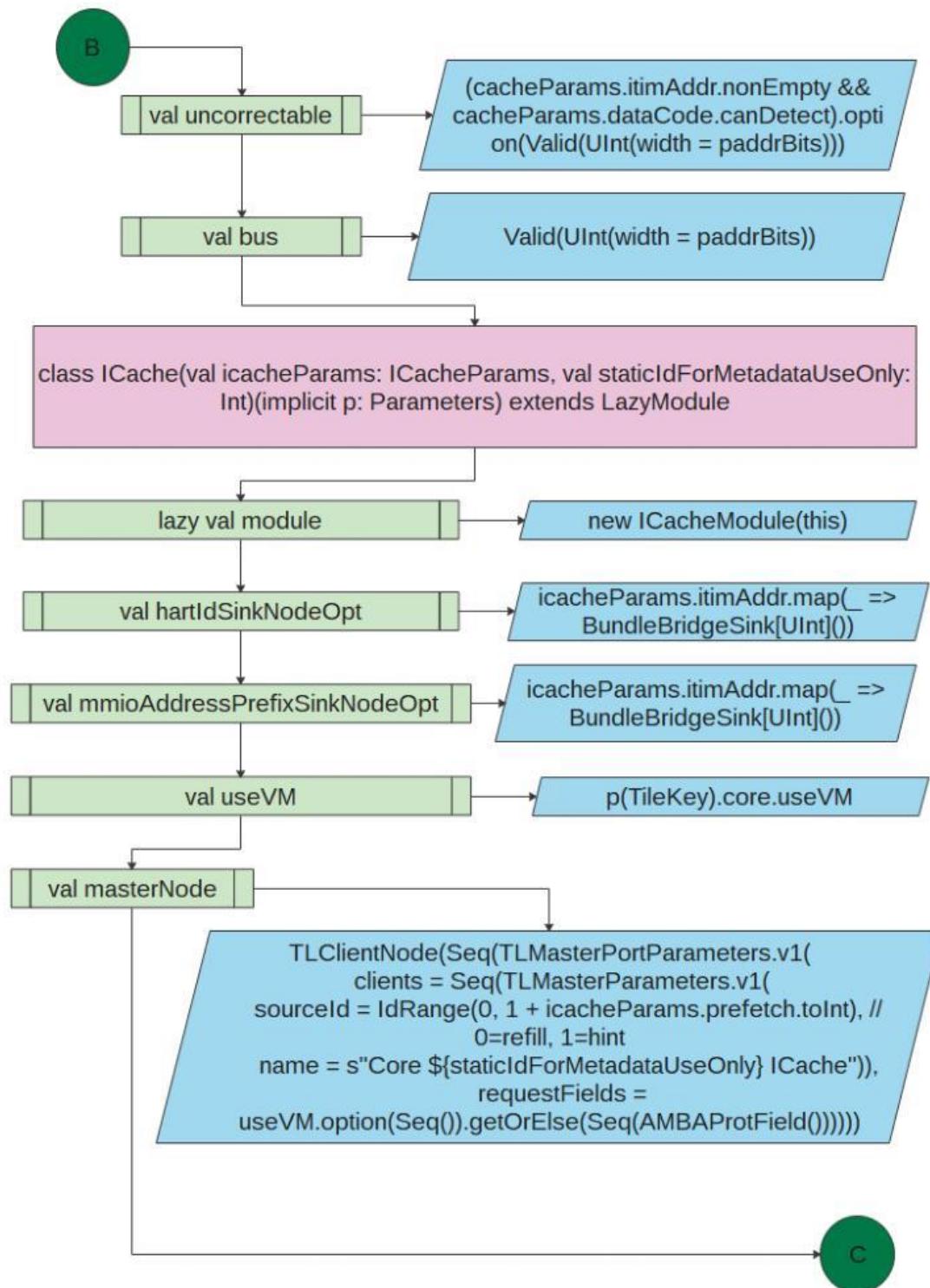
BundleBridgeSInk is tile-layer Chisel logic.

mmioAddressPrefixSinkNodeOpt variable map with BundleBridgeSInk, which is to be broadcast to units within the tile.

Value of useVM is initialized in tilelink class in chisel logic.

Seq is a trait which represents indexed sequences that are guaranteed immutable. You can access elements by using their indexes. It maintains insertion order of elements.

In masterNode variable, seq is created.



```

val size = icacheParams.nSets * icacheParams.nWays *
icacheParams.blockBytes
val itim_control_offset = size - icacheParams.nSets *
icacheParams.blockBytes

val device = new SimpleDevice("itim", Seq("sifive,itim0")) {
  override def describe(resources: ResourceBindings): Description = {
    val Description(name, mapping) = super.describe(resources)
    val Seq(Binding(_, ResourceAddress(address, perms))) =
resources("reg/mem")
    val base_address = address.head.base
    val mem_part = AddressSet.misaligned(base_address,
itim_control_offset)
    val control_part = AddressSet.misaligned(base_address +
itim_control_offset, size - itim_control_offset)
    val extra = Map(
      "reg-names" -> Seq(ResourceString("mem"),
ResourceString("control")),
      "reg" -> Seq(ResourceAddress(mem_part, perms),
ResourceAddress(control_part, perms)))
    Description(name, mapping ++ extra)
  }
}

```

Explanation

In size variable, performing multiplication between three parameters of icacheParams class that is nSets, nWays and blockBytes which is mentioned at the starting of icache file.

In itim_control_offset, size will subtracting from nSets multiple by blockBytes.

In device, SimpleDevice function is called and pass itim and sequence.

Override the describe function by passing resourceBindings. Description of this Function includes description(name, mapping) variable to which super.describe(resources) is assigned.

Then, make seq by binding resourceaddress to resources("reg/mem").

Declared base-address as address.head.base.

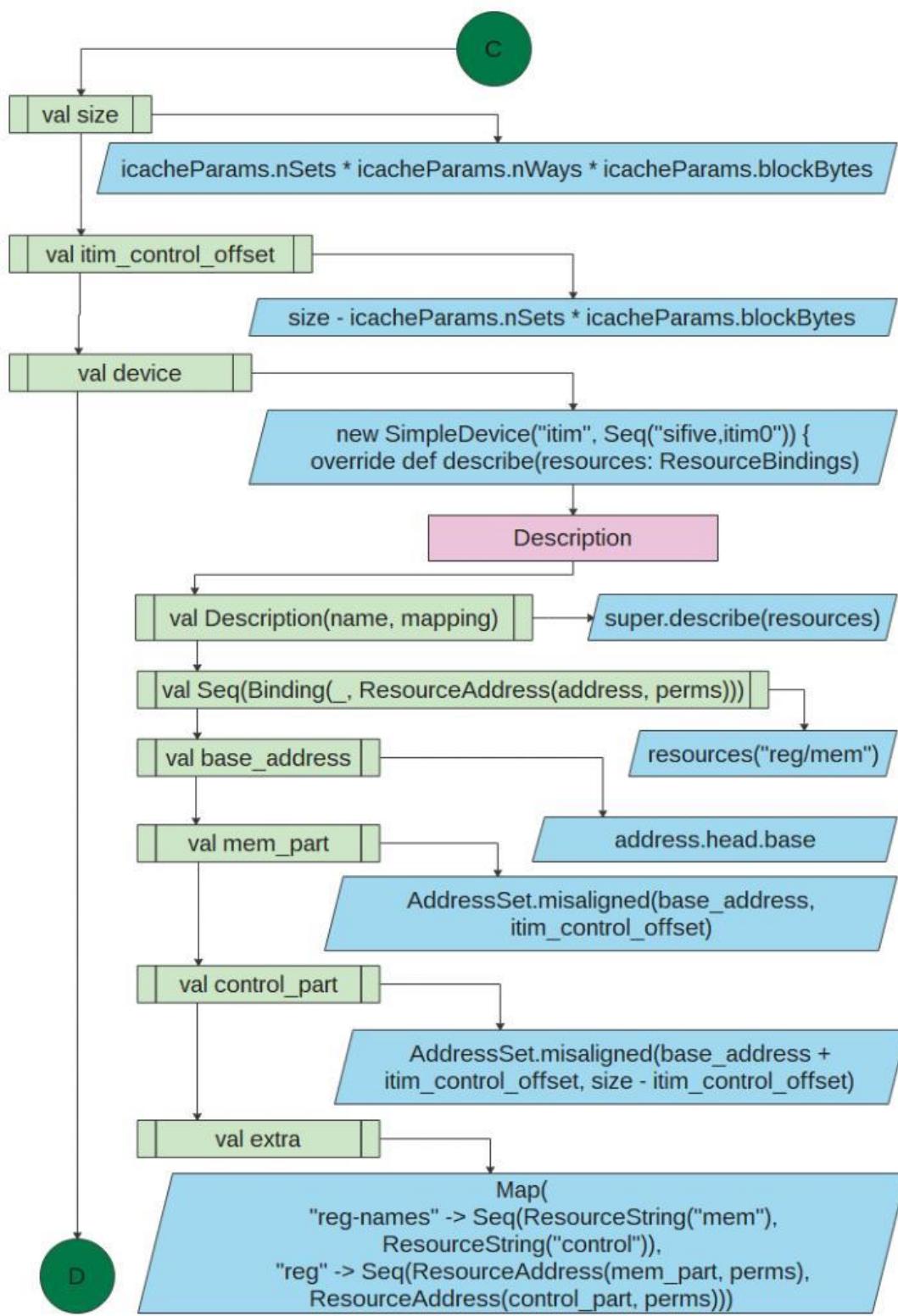
In mem_part, AddressSet.misaligned is called by passing base_address and itim_control_offset as parameter.

In control_part, AddressSet.misaligned is called by passing base_address + itim_control_offset and size - itim_control_offset as parameters.

ROCKET-CHIP Micro Architecture Specification Document

In extra variable, "reg-names" and "reg" are mapped with
Seq(ResourceString("mem"), ResourceString("control")) and
Seq(ResourceAddress(mem_part, perms), ResourceAddress(control_part, perms)))
respectively.

Then pass name and mapping ++ extra to decription variable.



```

def itimProperty: Option[Seq[ResourceValue]] = icacheParams.itimAddr.map(_
=> device.asProperty)

private val wordBytes = icacheParams.fetchBytes
val slaveNode =
  TLManagerNode(icacheParams.itimAddr.toSeq.map { itimAddr =>
TLSlavePortParameters.v1(
  Seq(TLSlavePortParameters.v1 (
    address      = Seq(AddressSet(itimAddr, size-1)),
    resources    = device.reg("mem"),
    regionType   = RegionType.IDEMPOTENT,
    executable   = true,
    supportsPutFull = TransferSizes(1, wordBytes),
    supportsPutPartial = TransferSizes(1, wordBytes),
    supportsGet    = TransferSizes(1, wordBytes),
    fifoId        = Some(0))), // requests handled in FIFO order
beatBytes = wordBytes,
minLatency = 1)})}
}

```

 explanation

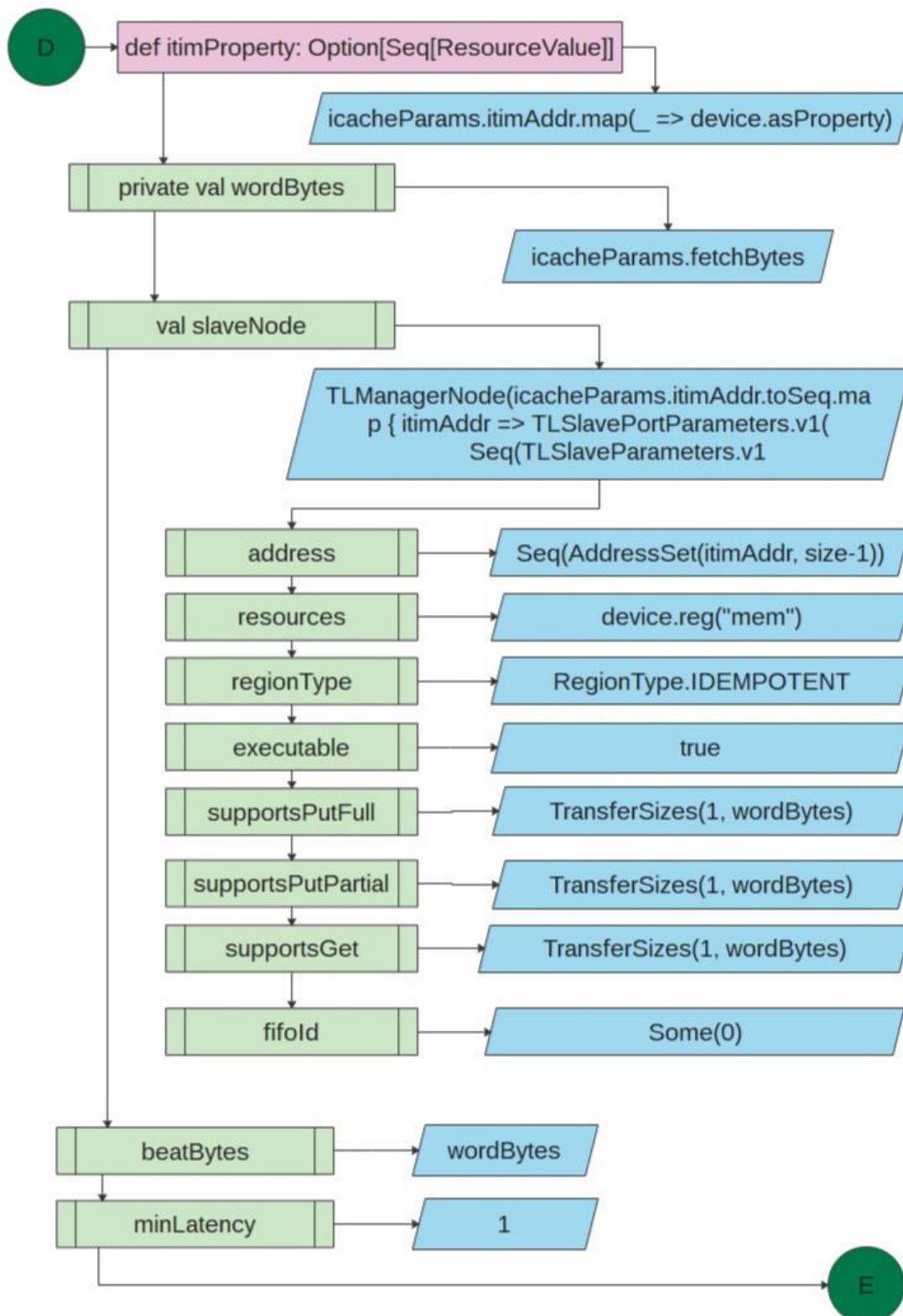
create a function called itimProperty in which option of resourcevalue sequence is map with device.asProperty.

Make wordBytes as a private variable of class and as value of fetchBytes variable which is a parameter of icacheParams class.

In slaveNode variable, first TLManagerNode is called in which itimAddr map with TLSlavePortParameters.v1 .

Then in TLSlavePortParameters.v1 again make a sequence in which TLSlavePortParameters.v1 is again called and then assigned values to variables which include address, resources, regionType, executable, supportsPutFull, supportsPutPartial ,supportsGet, fifold, beatBytes, minLatency.

Values of these variables are Seq(AddressSet(itimAddr, size-1)), device.reg("mem"), RegionType.IDEMPOTENT, true, TransferSizes(1, wordBytes), TransferSizes(1, wordBytes), TransferSizes(1, wordBytes), Some(0))), wordBytes, 1 respectively.



```

class ICacheResp(outer: ICache) extends Bundle {
    val data = UInt(width = outer.icacheParams.fetchBytes*8)
    val replay = Bool()
    val ae = Bool()
    override def cloneType = new ICacheResp(outer).asInstanceOf[this.type]
}
class ICachePerfEvents extends Bundle {
    val acquire = Bool()
}
class ICacheBundle(val outer: ICache) extends CoreBundle()(outer.p) {
    val req = Decoupled(new ICacheReq).flip
    val s1_paddr = UInt(INPUT, paddrBits) // delayed one cycle w.r.t. req
    val s2_vaddr = UInt(INPUT, vaddrBits) // delayed two cycles w.r.t. req
    val s1_kill = Bool(INPUT) // delayed one cycle w.r.t. req
    val s2_kill = Bool(INPUT) // delayed two cycles; prevents I$ miss
emission
    val s2_prefetch = Bool(INPUT) // should I$ prefetch next line on a miss?
    val resp = Valid(new ICacheResp(outer))

```

explanation

class ICacheResp with Icache as a outer is declared and extends with bundle.

Variable data is declared as unsigned integer with width equal to fetchBytes which is parameter of icacheParams class multiple by 8.

Replay and ae variables declared as bool type.

Override the cloneType function in which make a object of same class which is ICacheResp that's why use this keyword because "this" keyword is used to fetch the variable of the current class.

Then make ICachePerfEvents class and extends it with bundle.

Variable acquire is declared as bool type in ICachePerfEvents class.

class ICacheBundle with Icache as a outer is declared and extends with Corebundle.

In req variable, Decoupled is called in which make a object of ICacheReq class then value return from ICacheReq class is flipped and assigned to req variable

Decoupled is a helper to construct DecoupledIO from some other type.

DecoupledIO is a ready/valid interface type with members ready, valid, and bits

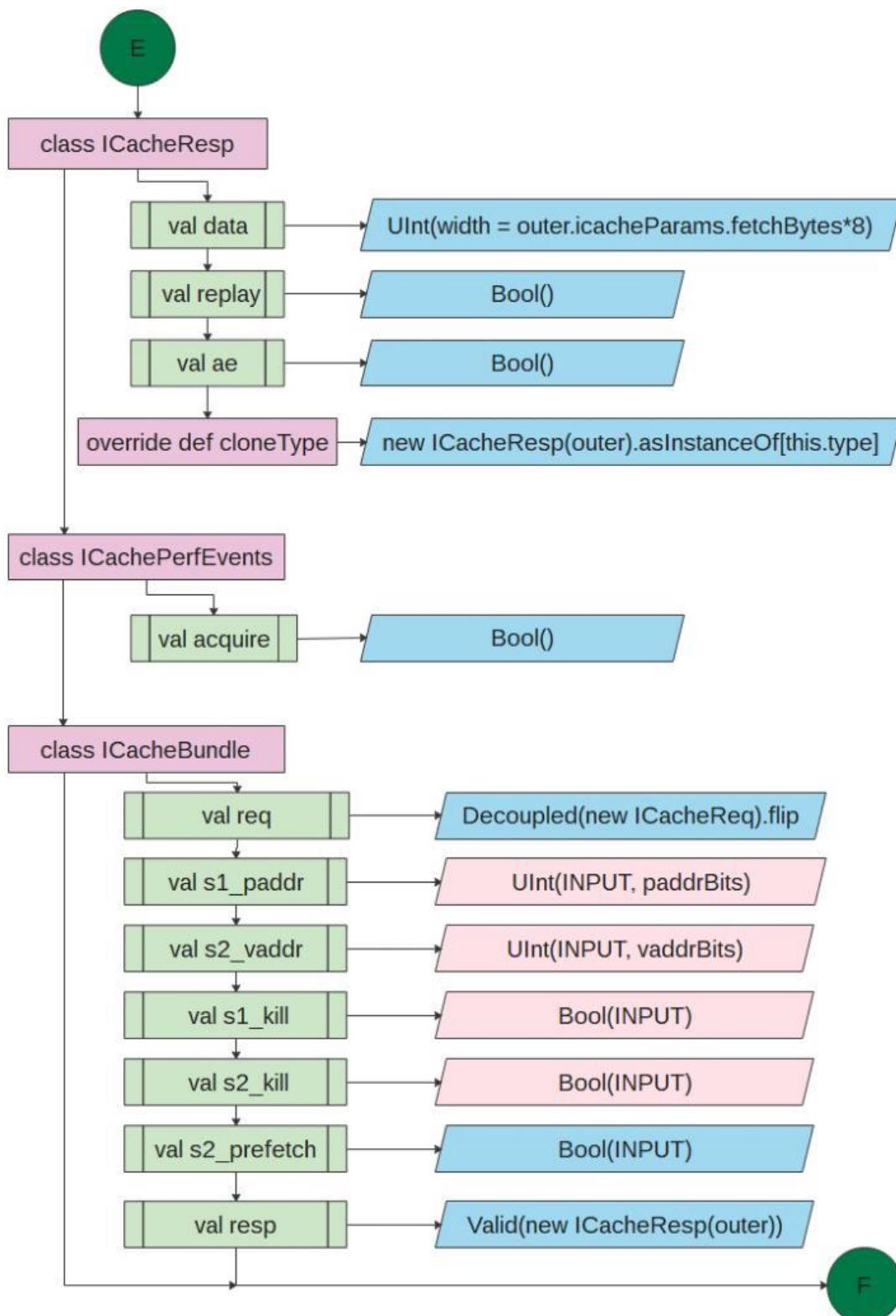
s1_paddr variable is declared with unsigned integer of width paddrBitis.

s2_vaddr variable is declared with unsigned integer of width vaddrBitis.

s1_kill, s2_kill and s2_prefetch are declared as bool input.

In resp variable, Valid is called in which make a object of ICacheReq class with outer as a parameter.

Valid is similar to Decoupled for constructing ValidIOs. ValidIO is similar to DecoupledIO except that it only has valid and bits



```

val invalidate = Bool(INPUT)
val errors = new ICacheErrors
val perf = new ICachePerfEvents().asOutput
val clock_enabled = Bool(INPUT)
val keep_clock_enabled = Bool(OUTPUT)
}
class ICacheModule(outer: ICache) extends LazyModuleImp(outer)
  with HasL1ICacheParameters {
  override val cacheParams = outer.icacheParams // Use the local parameters
  val io = IO(new ICacheBundle(outer))
  val (tl_out, edge_out) = outer.masterNode.out(0)
  // Option.unzip does not exist :-( 
  val (tl_in, edge_in) = outer.slaveNode.in.headOption.unzip
  val tECC = cacheParams.tagCode
  val dECC = cacheParams.dataCode
  require(isPow2(nSets) && isPow2(nWays))
  require(!usingVM || outer.icacheParams.itimAddr.isEmpty || pgIdxBits >=
untagBits,
    s"When VM and ITIM are enabled, I$$ set size must not exceed
${1<<(pgIdxBits-10)} KiB; got ${(outer.size/nWays)>>10} KiB")
}

```

 explanation

invalidate variable is declared as bool type input.

In errors make an object of ICacheErrors class.

In perf make an object of ICachePerfEvents as output.

clock_enabled and keep_clock_enabled are declared as bool type input.

Make a class of ICacheModule with icache as outer and extends I with LazyModuleImp with HasL1ICacheParameters class.

Override cacheParams variable to outer.icacheParams which means parameters of icacheParams class.

Declared val io as input output variable and make object of ICacheBundle class.

Assigned outer.masterNode.out(0) to (tl_out, edge_out) variable.

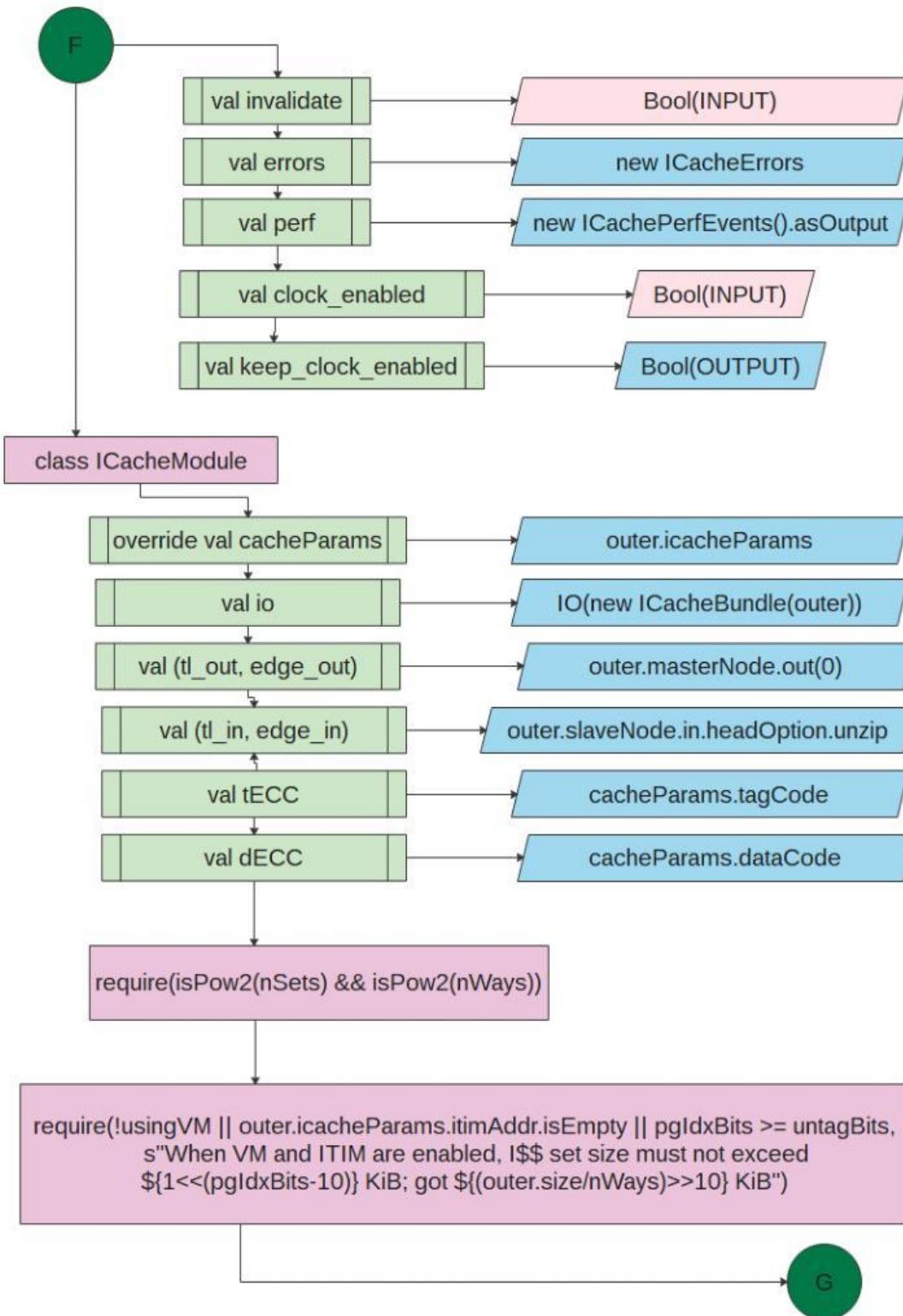
In tECC and dECC called the functions of ICacheParams class which tagCode and dataCode.

Require is scala build-in function. require means that the caller of the method is at fault and should fix its call.

First call require and pass parameter which is result generate after performing AND operation between nSets and nWays of power of 2.

Then calls require function again and pass parameter which is result generate after performing OR operation between not usingVM, outer.icacheParams.itimAddr.isEmpty ,and pgIdxBits >= untagBits.

Then notation "s" means: Prepending s to any string literal allows the usage of variables directly in the string.



```

val io_hartid = outer.hartIdSinkNodeOpt.map(_.bundle)
val io_mmio_address_prefix =
outer.mmioAddressPrefixSinkNodeOpt.map(_.bundle)
val scratchpadOn = RegInit(false.B)
val scratchpadMax = tl_in.map(tl => Reg(UInt(width = log2Ceil(nSets * (nWays - 1)))))
def lineInScratchpad(line: UInt) = scratchpadMax.map(scratchpadOn && line
<= _).getOrElse(false.B)
val scratchpadBase = outer.icacheParams.itimAddr.map { dummy =>
  p(LookupByHartId) (_ .icache.flatMap(_ .itimAddr.map(_ .U)), io_hartid.get)
| io_mmio_address_prefix.get
}
def addrMaybeInScratchpad(addr: UInt) = scratchpadBase.map(base => addr
>= base && addr < base + outer.size).getOrElse(false.B)

```

explanation

In io_hartid variable, outer.hartIdSinkNodeOpt is map with bundle.

In io_mmio_address_prefix, outer.mmioAddressPrefixSinkNodeOpt is map with bundle.

In scratchpadOn, register is initialized with value false.B.

In scratchpadMax, tl is map with register of value unsigned integer with width of log2Ceil(nSets * (nWays - 1)) where nSets and nWays are parameters of ICacheparams class.

Create a function lineInScratchpad with parameter line as unsigned integer. scratchpadMax is map with the result of AND operation between scratchpadOn and line.

Line is map with the getOrElse function of scala with False.B as a parameter.

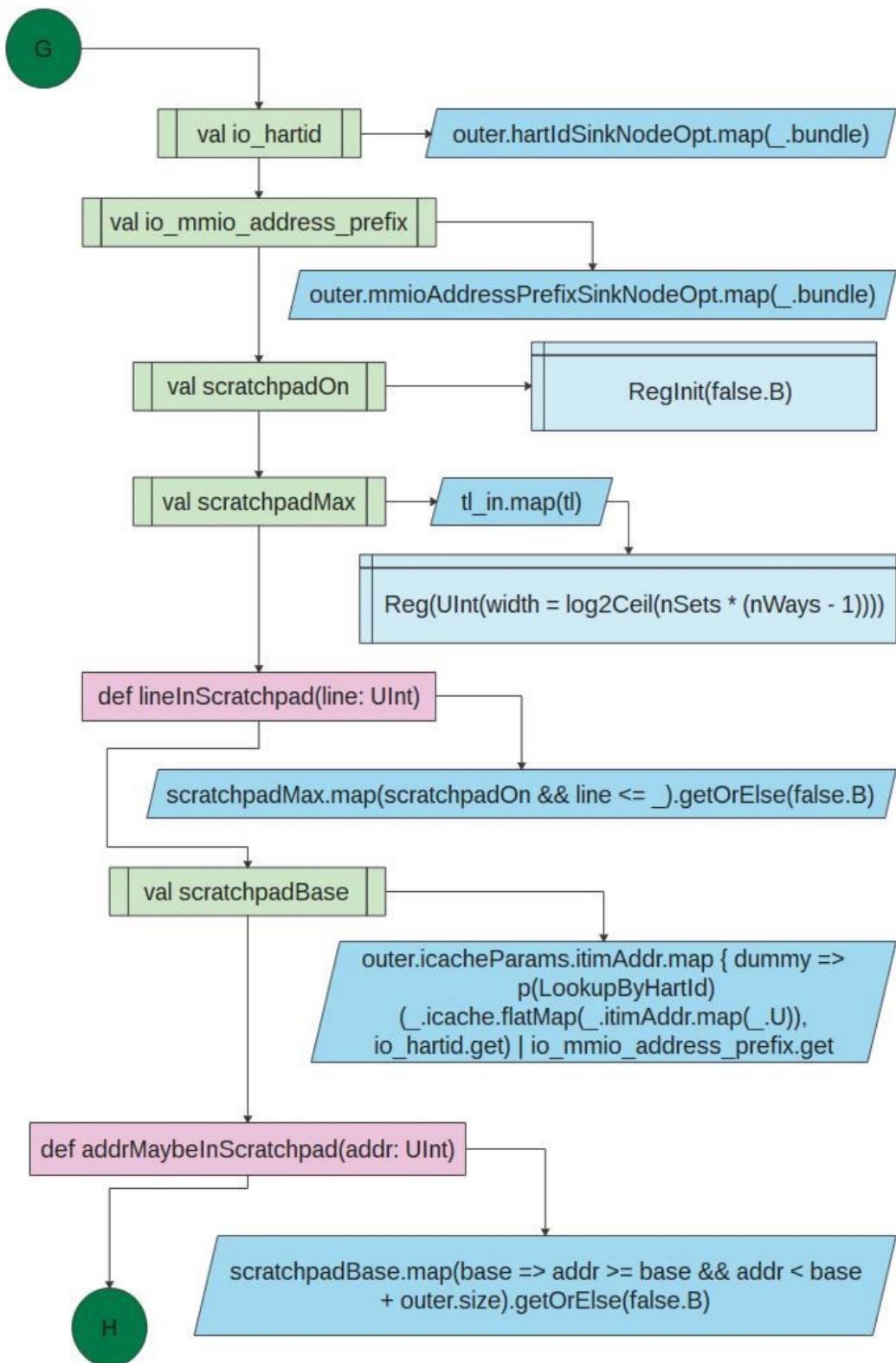
getOrElse() method of scala is used to access a value or a default when no value is present.

In scratchpadBase variable, itimAddr which is parameter of ICacheparams class is map with dummy and then dummy is map with LookupByHartId.

Then icache.flatMap function is called in which itimAddr is map with result of bitwiseOR operation between io_hartid.get and io_mmio_address_prefix.get.

Create a function addrMaybeInScratchpad with unsigned variable addr as a parameter.

In addrMaybeInScratchpad function, = scratchpadBase is map with base then base is map with result of following operation: addr is greater than equal to result of And operation between base and addr which is less than outer.size. Also call getOrElse function of scala with false.B as parameter



```

def addrInScratchpad(addr: UInt) = addrMaybeInScratchpad(addr) &&
lineInScratchpad(addr(untagBits+log2Ceil(nWays)-1, blockOffBits))
def scratchpadWay(addr: UInt) = addr.extract(untagBits+log2Ceil(nWays)-1,
untagBits)
def scratchpadWayValid(way: UInt) = way < nWays - 1
def scratchpadLine(addr: UInt) = addr(untagBits+log2Ceil(nWays)-1,
blockOffBits)
val s0_slaveValid = tl_in.map(_.a.fire()).getOrElse(false.B)
val s1_slaveValid = RegNext(s0_slaveValid, false.B)
val s2_slaveValid = RegNext(s1_slaveValid, false.B)
val s3_slaveValid = RegNext(false.B)
val s0_valid = io.req.fire()
val s0_vaddr = io.req.bits.addr

```

explanation

Create a function of addrInScratchpad with unsigned variable addr as parameter where addr is initialized in ICacheReq class.

In addrInScratchpad function, call addrMaybeInScratchpad function with parameter addr and also call lineInScratchpad function with parameter addr(untagBits+log2Ceil(nWays)-1, blockOffBits). Then perform And operation between these two functions.

Create function scratchpadWay with unsigned variable addr as parameter where addr is initialized in ICacheReq class.

In this addr.extract function is called with two parameters untagBits+log2Ceil(nWays)-1 and untagBits where nWays and untagBits are parameters ICacheParams class.

log2Ceil is useful for getting the number of bits needed to represent some number of states (in - 1).

Create a function scratchpadWayValid with unsigned variable way as parameter. In this function, check that way is less than nWays-1 or not where nWays is a parameter of ICacheParams class.

Create a function of scratchpadLine with unsigned variable addr as parameter where addr is initialized in ICacheReq class. In this function addr variable is called with value equal to (untagBits+log2Ceil(nWays)-1, blockOffBits).

In s0_slaveValid variable, tl is map with a.fire function with getOrElse function of scala has parameter false.B.

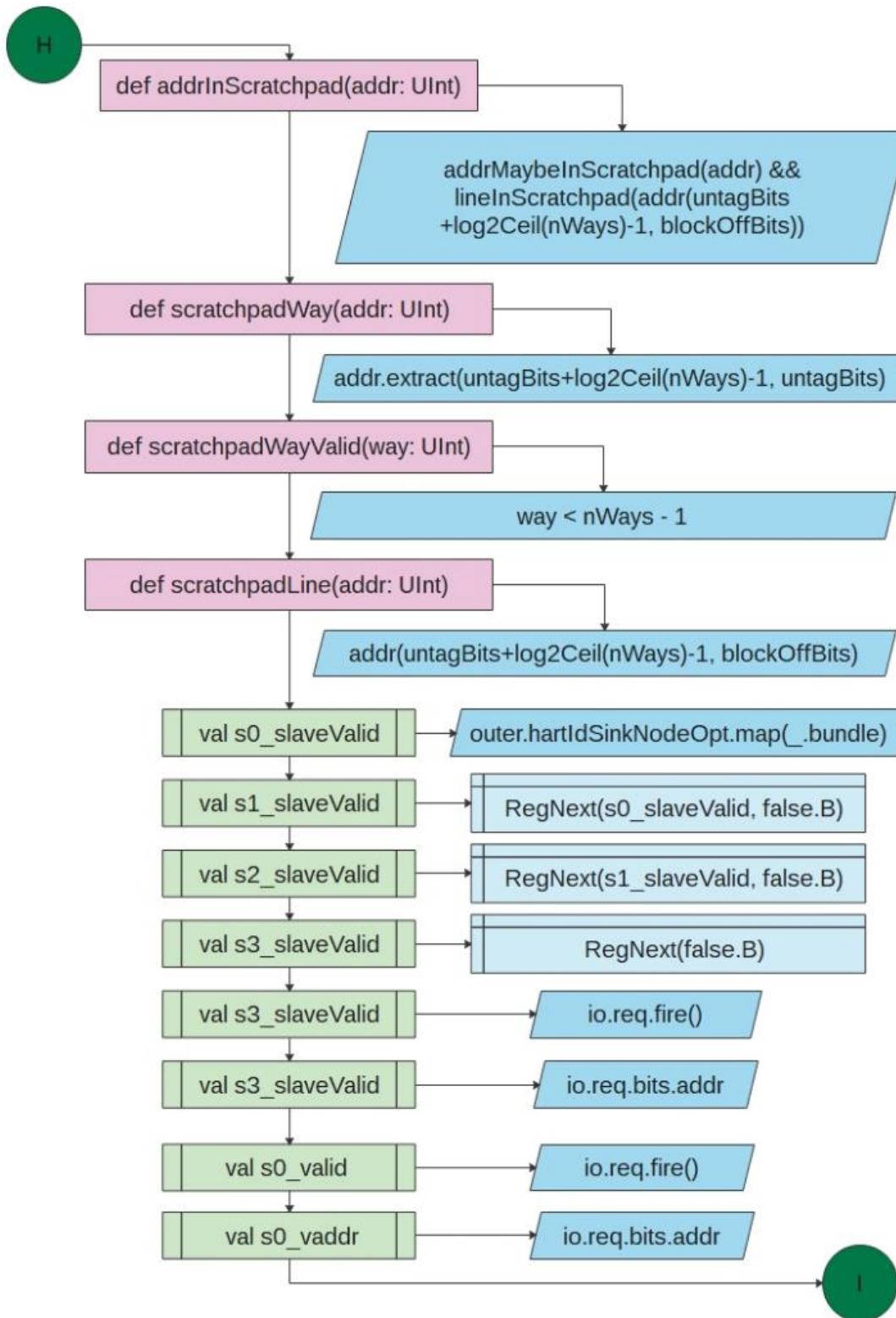
In s1_slaveValid, create a register using regNext with values s0_slaveValid and false.B .

In s2_slaveValid, create a register using regNext with values s1_slaveValid and false.B .

In s3_slaveValid, create a register using regNext with value false.B .

s0_valid variable is declared and io.req.fire() is assigned as value.

s0_vaddr variable is declared and io.req.bits.addr is assigned as value



```

val s1_valid = Reg(init=Bool(false))
val s1_vaddr = RegEnable(s0_vaddr, s0_valid)
val s1_tag_hit = Wire(Vec(nWays, Bool()))
val s1_hit = s1_tag_hit.reduce(_||_) || Mux(s1_slaveValid, true.B,
addrMaybeInScratchpad(io.s1_paddr))
don'tTouch(s1_hit)
val s2_valid = RegNext(s1_valid && !io.s1_kill, Bool(false))
val s2_hit = RegNext(s1_hit)
val invalidated = Reg(Bool())
val refill_valid = RegInit(false.B)
val send_hint = RegInit(false.B)
val refill_fire = tl_out.a.fire() && !send_hint
val hint_outstanding = RegInit(false.B)
val s2_miss = s2_valid && !s2_hit && !io.s2_kill
val s1_can_request_refill = !(s2_miss || refill_valid)

```

 explanation

s1_valid variable is declared in which initialized a register with value False of bool type.

In s1_vaddr, Regenable function is called which chisel build-in function to enable register and pass two parameters s0_vaddr and s0_valid.

In s1_tag_hit, wire function is called which is chisel build-in function use for hardware wiring. In wire function, vector is create with nWays and bool type values.

In s1_hit, Or operation is performed between s1_tag_hit.reduce and Mux. In Mux, s1_slaveValid and true.B passed as input of Mux and for selected pin addrMaybeInScratchpad is called with parameter (io.s1_paddr)).

don'tTouch is chisel function which is import from chisel in starting of Icache class. don'tTouch function marks that a signal should not be removed by Chisel and Firrtl optimization passes.

RegNext is Utility for constructing one-cycle delayed versions of signals.

In s2_valid , register is create with RegNext and pass the value which is get after performing AND operation between s1_valid and not io.s1_kill, Bool(false)).

Invalidated variable is declared as a Reg with bool value.

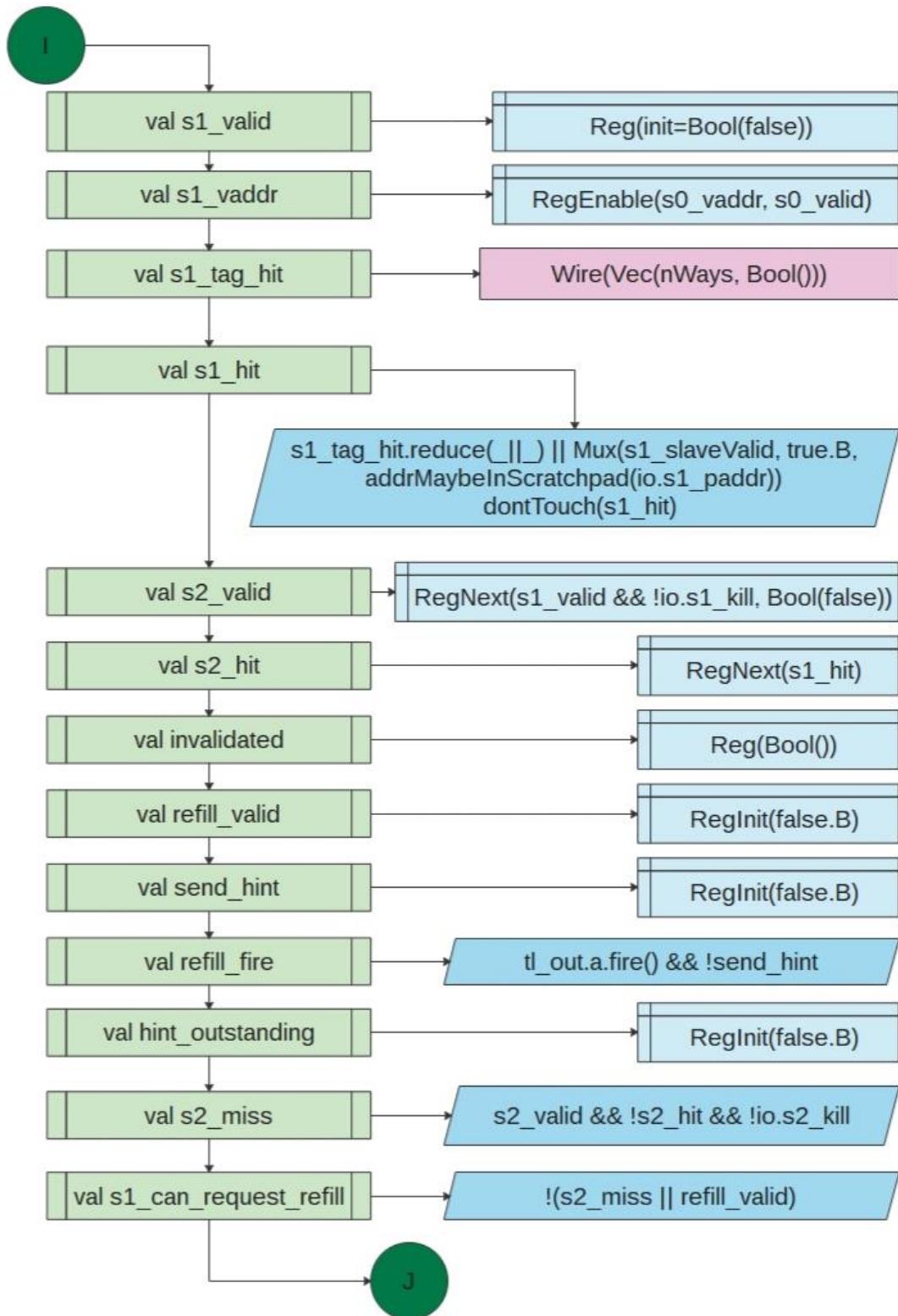
In refill_valid and send_hint, register is initialized with value false.B.

In refill_fire, And operation is performed between tl_out.a.fire() and not send_hint.

In hint_outstanding, register is initialized with value false.B.

In s2_miss, And operation is performed between s2_valid, not s2_hit and not io.s2_kill.

In s1_can_request_refill , Or operation is performed between s2_miss and refill_valid which is become reverse as true become false and vice versa and then assigned to s1_can_request_refill variable.



```

val s2_request_refill = s2_miss && RegNext(s1_can_request_refill)
val refill_paddr = RegEnable(io.s1_paddr, s1_valid &&
s1_can_request_refill)
val refill_vaddr = RegEnable(s1_vaddr, s1_valid && s1_can_request_refill)
val refill_tag = refill_paddr >> pgUntagBits
val refill_idx = index(refill_vaddr, refill_paddr)
val refill_one_beat = tl_out.d.fire() && edge_out.hasData(tl_out.d.bits)
io.req.ready := !(refill_one_beat || s0_slaveValid || s3_slaveValid)
s1_valid := s0_valid
val (_ , _ , d_done, refill_cnt) = edge_out.count(tl_out.d)
val refill_done = refill_one_beat && d_done
tl_out.d.ready := !s3_slaveValid
require (edge_out.manager.minLatency > 0)

```

 explanation

In s2_request_refill, AND operation is performed between s2_miss and RegNext(s1_can_request_refill).

In refill_paddr, calls RegEnable function which is chisel build-in function. Then pass two parameters io.s1_paddr and result of AND operation between s1_valid and s1_can_request_refill.

In refill_vaddr, calls RegEnable function which is chisel build-in function. Then pass two parameters s1_paddr and result of AND operation between s1_valid and s1_can_request_refill.

In refill_tag, Binary Right Shift Operator is used between refill_paddr and pgUntagBits. The Bit positions of the left operand value is moved right by the number of bits specified by the right operand i.e Bit positions of refill_paddr is moved right by number of bits in pgUntagBits. pgUntagBits is function describe in tile file which is import at the starting of icache file.

```
def pgUntagBits = if (usingVM) untagBits min pgIdxBits else untagBits
```

In refill_idx, call index function of scala in which pass two parameters refill_vaddr and refill_paddr.

In refill_one_beat, AND operation is performed between tl_out.d.fire() and edge_out.hasData(tl_out.d.bits).

Then io.req.ready is wired with inverse of OR operatio result which is performed between refill_one_beat, s0_slaveValid and s3_slaveValid.

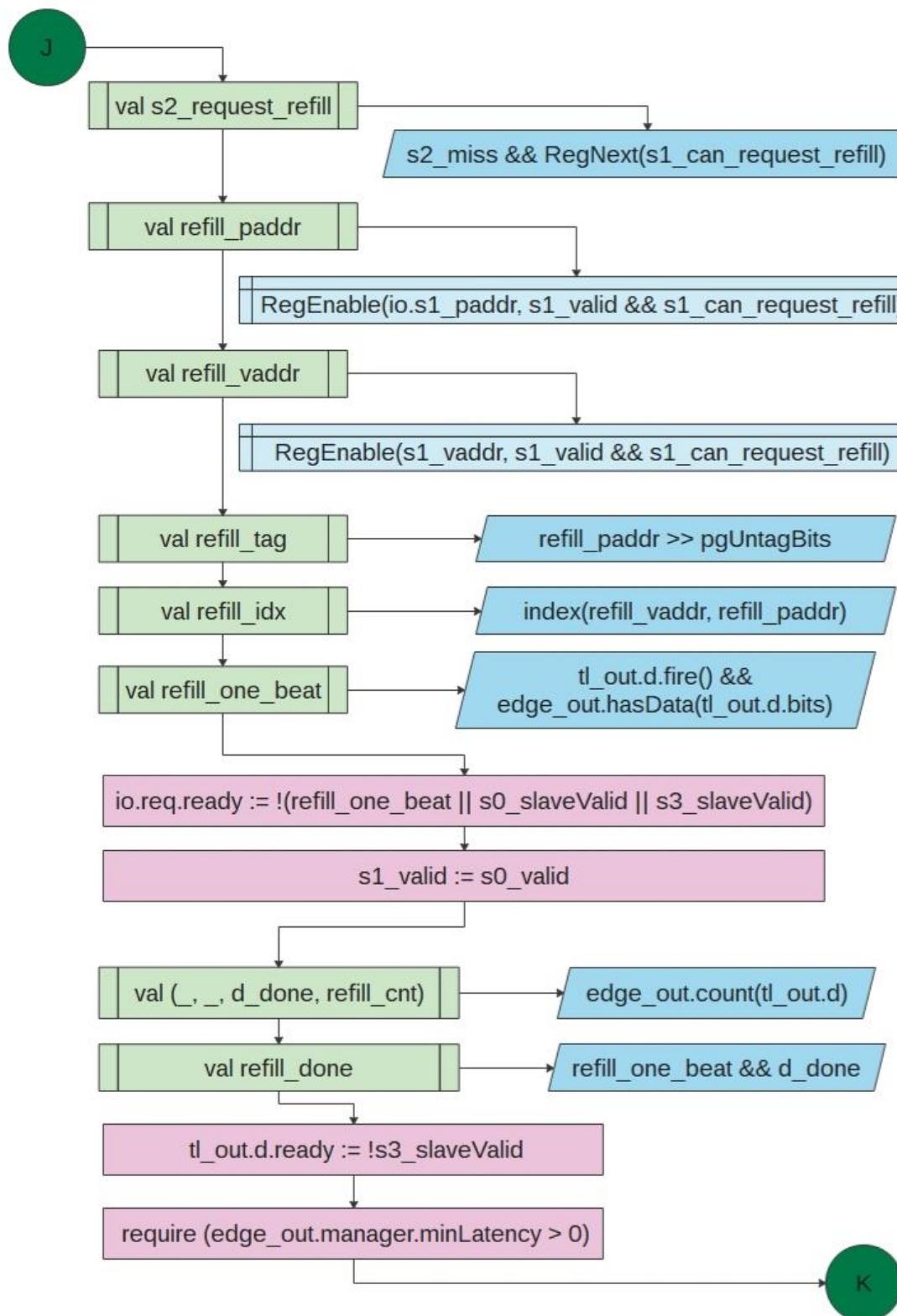
s1_valid is wired with s0_valid.

val (_ , _ , d_done, refill_cnt) assigned a value which is edge_out.count(tl_out.d).

In refill_done, AND operation is performed between refill_one_beat and d_done.

tl_out.d.ready is wired with reverse of s3_slaveValid value.

Then require function is called which is scala build-in function and pass edge_out.manager.minLatency > 0 as a parameter.



```

val repl_way = if (isDM) UInt(0) else {
    // pick a way that is not used by the scratchpad
    val v0 = LFSR(16, refill_fire)(log2Up(nWays)-1,0)
    var v = v0
    for (i <- log2Ceil(nWays) - 1 to 0 by -1) {
        val mask = nWays - (BigInt(1) << (i + 1))
        v = v | (lineInScratchpad(Cat(v0 | mask.U, refill_idx)) << i)
    }
    assert(!lineInScratchpad(Cat(v, refill_idx)))
    v
}

```

 explanation

In repl_way, if isDm is True then assigned unsigned 0 to repl_way.

In else condition of repl-way:

First declared a variable v0 in which LFSR function is call which is chisel build-in function with parameters 16 and refill_fire, and again call LFSR with parameters log2Up(nWays)-1 and 0.

LFSR is utilities related to psuedorandom number generation using Linear Feedback Shift Registers (LFSRs).

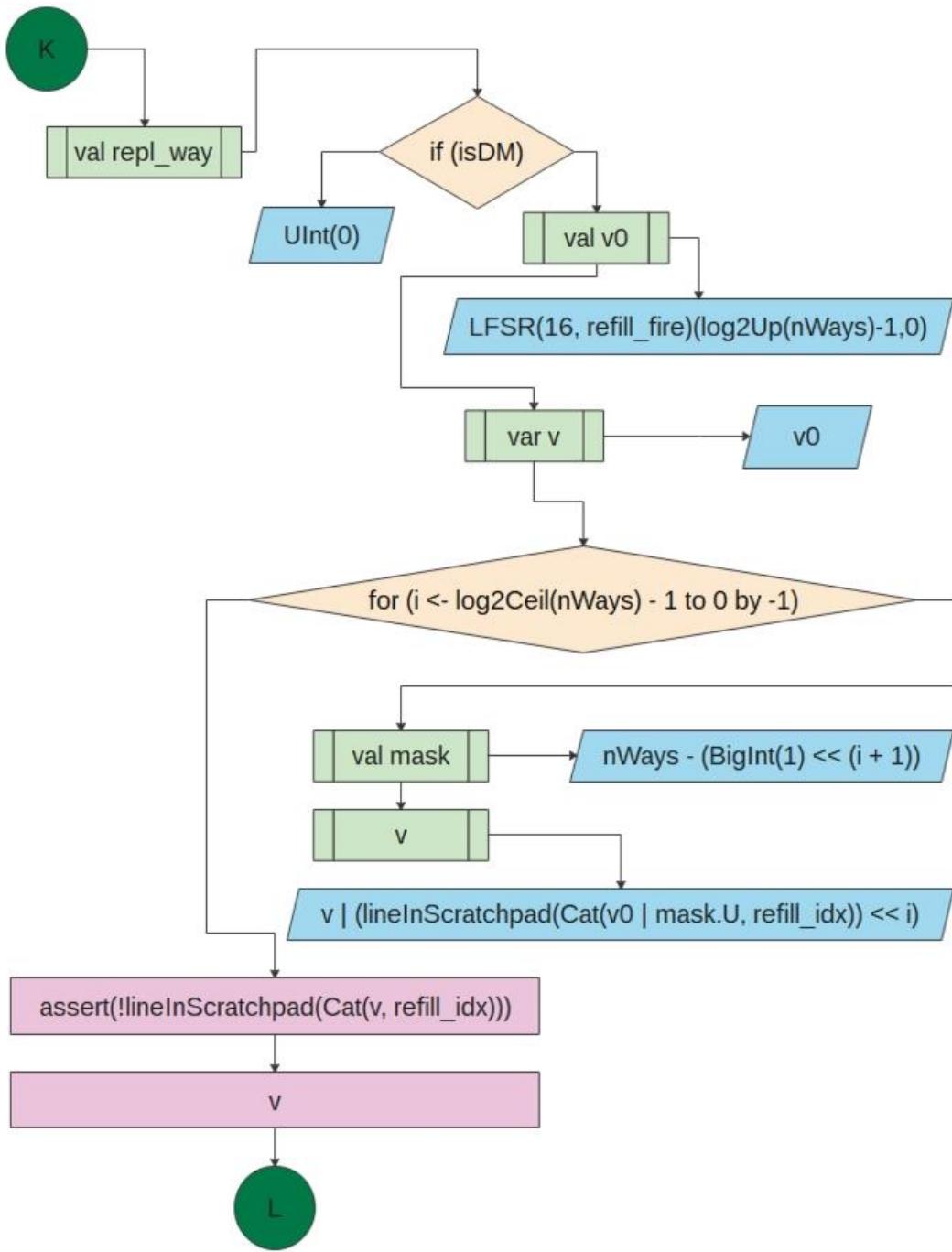
Then v is initialized as var which means it can be change later. V0 is assigned to v.

Then for loop will start from log2Ceil(nWays) - 1 and iterate in reverse order as it shows “by -1” .

In this for loop, mask variable is assigned with value of nWays subtracts from (BigInt(1) Binary Left Shift (i + 1)) i.e. BigInt(1) is left shift by number of i+1 bits. Then in variable v, perform bitwise OR operation between v and lineInScratchpad function with parameter which is get after performing bitwise Or operation between v0 and mask.U, then concatenate it with refill_idx and left shift this by i number of bits.

Then assert function is call which is scala build-in function. In assert function, lineInScratchpad function is call with a parameter create by concatenate v with refill_idx then pass. Value return from lineInScratchpad function is invert and pass as a parameter of assert function.

Then value of v is set.



```

val (tag_array, omSRAM) = DescribedSRAM(
    name = "tag_array",
    desc = "ICache Tag Array",
    size = nSets,
    data = Vec(nWays, UInt(width = tECC.width(1 + tagBits)))
)
val tag_rdata = tag_array.read(s0_vaddr(untagBits-1,blockOffBits),
!refill_done && s0_valid)
val accruedRefillError = Reg(Bool())
val refillError = tl_out.d.bits.corrupt || (refill_cnt > 0 &&
accruedRefillError)
when (refill_done) {
    // For AccessAckData, denied => corrupt
    val enc_tag = tECC.encode(Cat(refillError, refill_tag))
    tag_array.write(refill_idx, Vec.fill(nWays)(enc_tag),
Seq.tabulate(nWays)(repl_way === _))
    ccover(tl_out.d.bits.corrupt, "D_CORRUPT", "I$ D-channel corrupt")
}

```

 explanation

In val (tag_array, omSRAM), DescribedSRAM function is call which initialized somewhere in rocketchip.

In DescribedSRAM function, pass name, desc, size and data as a parameters with values "tag_array", "ICache Tag Array", nSets, Vec(nWays, UInt(width = tECC.width(1 + tagBits))) respectively.

In tag_rdata, tag_array reads value of s0_vaddr variable by passing untagBits-1 and blockOffBits, and also value of result get after performing AND operation between not refill_done and s0_valid.

accruedRefillError declared as register with bool.

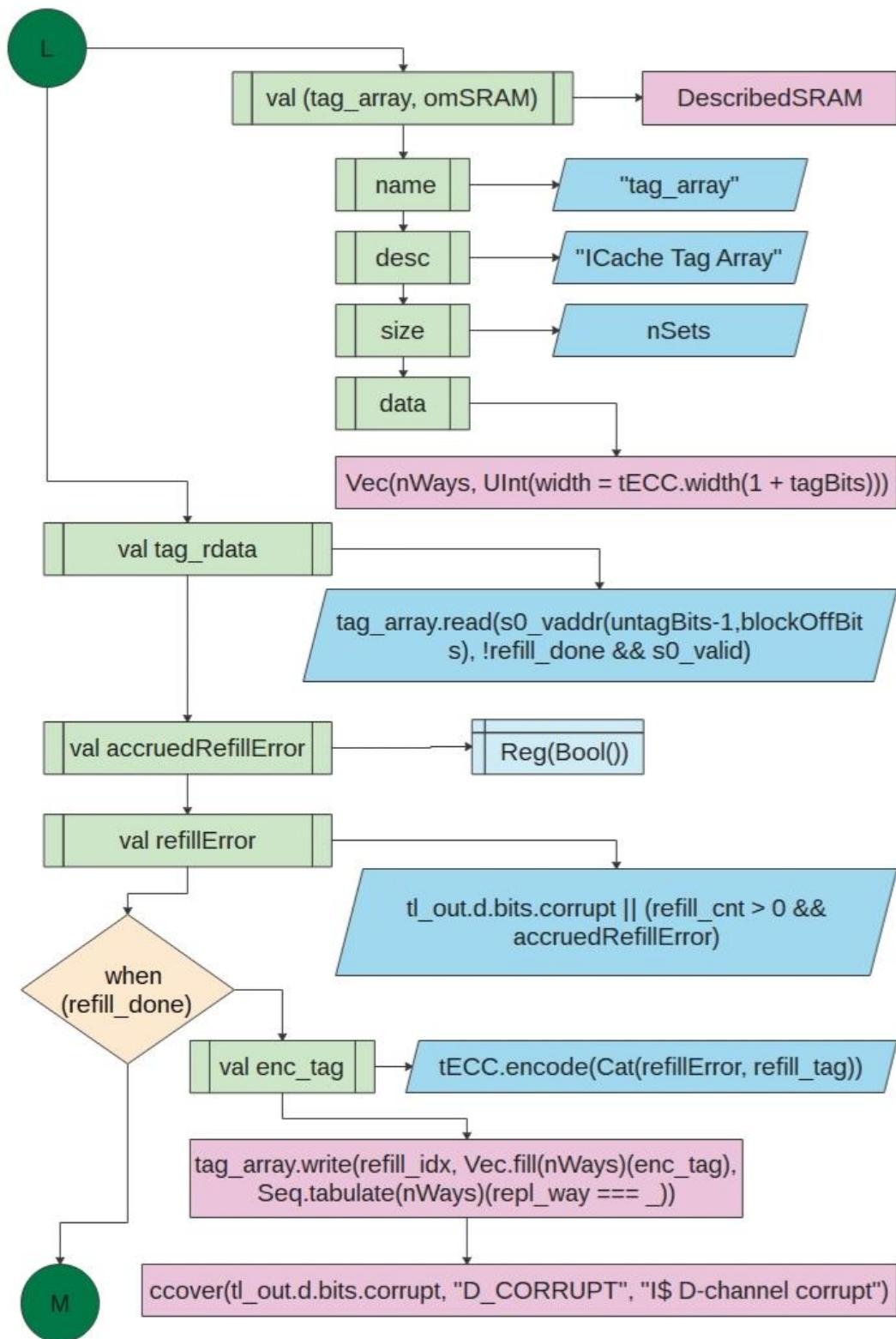
In refillError, OR operation is perform between tl_out.d.bits.corrupt and result of AND operation between refill_cnt > 0 and accruedRefillError.

When refill_done is True.

Then enc_tag assigned as tECC.encode with concatenation of refillError and refill_tag.

Then tag_array will write in sequence tabulate form.

Then ccover function is call, in which tl_out.d.bits.corrupt, "D_CORRUPT", "I\$ D-channel corrupt" are pass as a parameter



```

io.errors.bus.valid := tl_out.d.fire() && (tl_out.d.bits.denied || tl_out.d.bits.corrupt)
io.errors.bus.bits  := (refill_paddr >> blockOffBits) << blockOffBits
val vb_array = Reg(init=Bits(0, nSets*nWays))
when (refill_one_beat) {
    accruedRefillError := refillError
    // clear bit when refill starts so hit-under-miss doesn't fetch bad data
    vb_array := vb_array.bitSet(Cat(repl_way, refill_idx), refill_done && !invalidated)
}
val invalidate = Wire(init = io.invalidate)
when (invalidate) {
    vb_array := Bits(0)
    invalidated := Bool(true)
}
val s1_tag_disparity = Wire(Vec(nWays, Bool()))
val s1_tl_error = Wire(Vec(nWays, Bool()))
val wordBits = outer.icacheParams.fetchBytes*8
val s1_dout = Wire(Vec(nWays, UInt(width = dECC.width(wordBits))))

```

explanation

io.errors.bus.valid is wired with Boolean result which get by performing AND operation between tl_out.d.fire() and result of OR operation between tl_out.d.bits.denied and tl_out.d.bits.corrupt.

io.errors.bus.bits is wired with refill_paddr which is right shift by number of blockOffBits then again left shift by number of blockOffBits.

In vb_array , register is initialized by Bits(0, nSets*nWays) where nSets and nWays are parameter of ICacheParams class.

When refill_one_beat is True .

Then accruedRefillError is wired with refillError.

vb_array is wired with vb_array.bitSet(with the concatenation of repl_way and refill_idx), and result of AND operation between refill_done and not invalidated).

In invalidate variable , Wire function is call which chisel build-in function which initialized as io.invalidate.

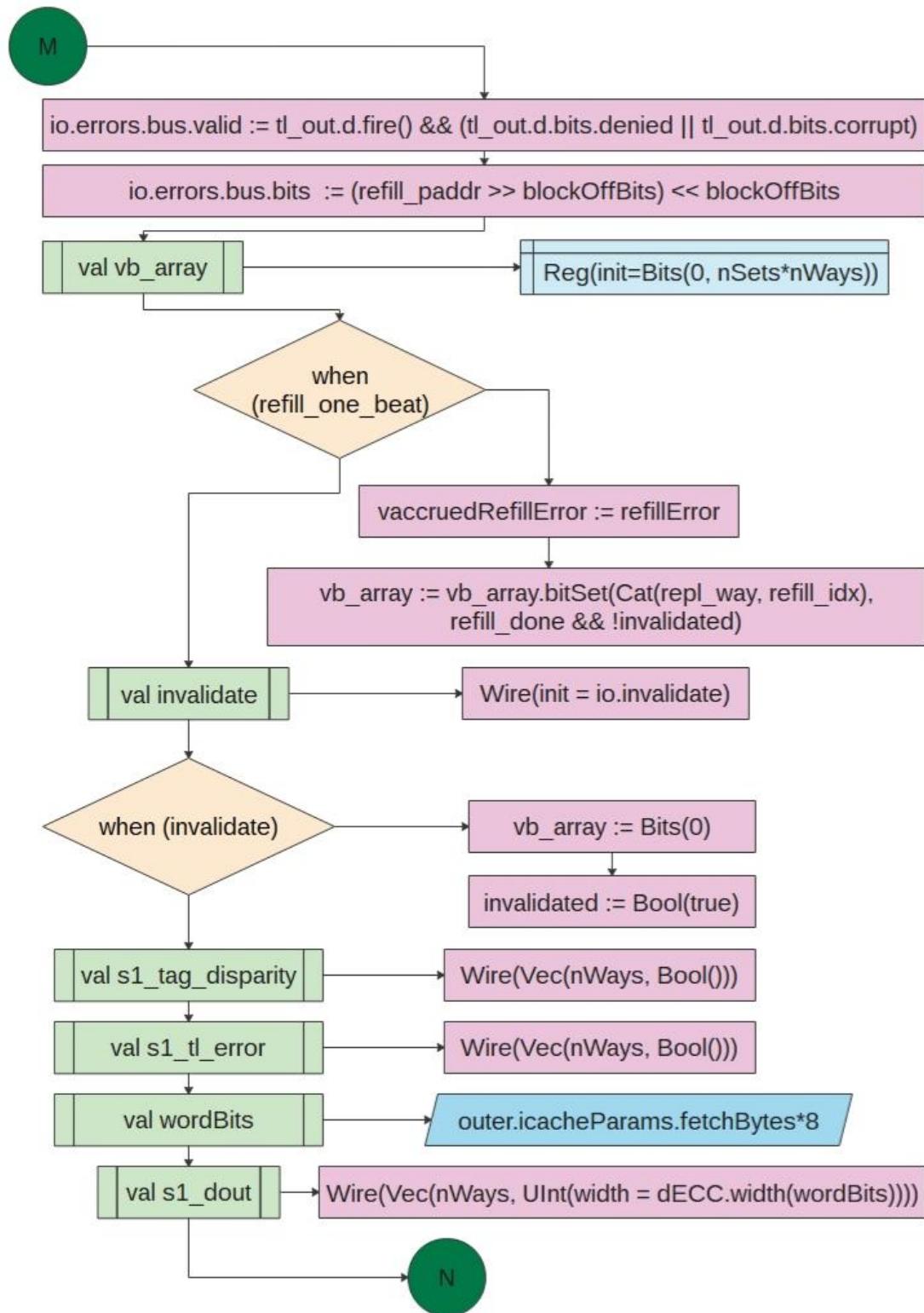
when invalidate is True then vb_array is wired with Bits(0) and invalidated is wired with Bool(true).

In s1_tag_disparity , Wire function is call which chisel build-in function which initialized as Vector of nWays and Bool().

In s1_tl_error, Wire function is call which chisel build-in function which initialized as Vector of nWays and Bool().

In wordBits , outer.icacheParams.fetchBytes*8 where fetchBytes is a parameter of ICache Params class.

In s1_dout, Wire function is call which chisel build-in function which initialized as Vector of nWays and Unsigned interger of width dECC.width(wordBits))).



```

val s0_slaveAddr = tl_in.map(_.a.bits.address).getOrElse(0.U)
val s1s3_slaveAddr = Reg(UInt(width = log2Ceil(outer.size)))
val s1s3_slaveData = Reg(UInt(width = wordBits))

for (i <- 0 until nWays) {
  val s1_idx = index(s1_vaddr, io.s1_paddr)
  val s1_tag = io.s1_paddr >> pgUntagBits
  val scratchpadHit = scratchpadWayValid(i) &&
    Mux(s1_slaveValid,
        lineInScratchpad(scratchpadLine(s1s3_slaveAddr)) &&
        scratchpadWay(s1s3_slaveAddr) === i,
        addrInScratchpad(io.s1_paddr) && scratchpadWay(io.s1_paddr) === i)
  val s1_vb = vb_array(Cat(UInt(i), s1_idx)) && !s1_slaveValid
  val enc_tag = tECC.decode(tag_rdata(i))
  val (tl_error, tag) = Split(enc_tag.uncorrected, tagBits)
  val tagMatch = s1_vb && tag === s1_tag
  s1_tag_disparity(i) := s1_vb && enc_tag.error

```

explanation

In s0_slaveAddr, tl is map with (_.a.bits.address).getOrElse function with unsigned 0

In s1s3_slaveAddr, Register of unsigned integer with width of log2Ceil(outer.size) is declared.

In s1s3_slaveData, Register of unsigned integer with width of wordBits.

For loop is iterate from zero to nWays, value of nWays is declared in ICacheParams class parameter.

Until for loop is iterated :

In s1_idx variable, index function of scala is call, s1_vaddr and io.s1_paddr are pass as parameters.

In s1_tag, io.s1_paddr is right shift by number of pgUntagBits.

In scratchpadHit variable, AND operation is perform between scratchpadWayValid function with i as a parameter and Mux.

In Mux following inputs are passed s1_slaveValid,
 lineInScratchpad(scratchpadLine(s1s3_slaveAddr)) &&
 scratchpadWay(s1s3_slaveAddr) === i and addrInScratchpad(io.s1_paddr) &&
 scratchpadWay(io.s1_paddr) === i)

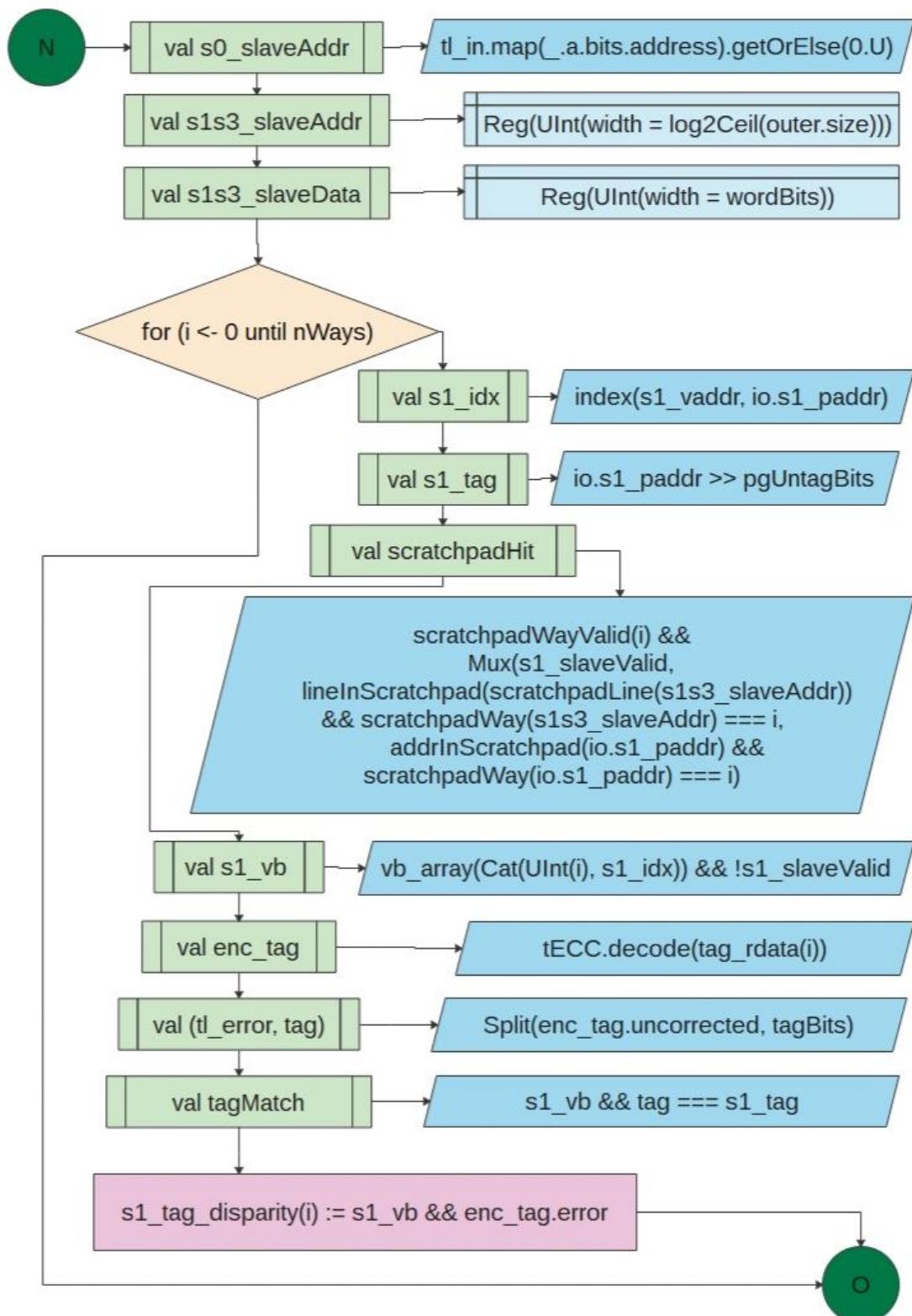
In s1_vb, AND operation between vb_array concatenate with unsigned integer i and s1_idx and not s1_slaveValid.

enc_tag is assigned as tECC.decode(tag_rdata(i)).

In val (tl_error, tag), split function scala is call which is used for splitting. Here enc_tag.uncorrected and tagBits are pass as split function parameters.

In tagMatch , AND operation is perform between s1_vb and tag === s1_tag. tag === s1_tag is True when both have same type i.e. UInt.

s1_tag_disparity(i) is wired with the result of AND operation between s1_vb and enc_tag.error.



```

s1_tl_error(i) := tagMatch && tl_error.asBool
    s1_tag_hit(i) := tagMatch || scratchpadHit
}
assert(!(s1_valid || s1_slaveValid) || PopCount(s1_tag_hit zip
s1_tag_disparity map { case (h, d) => h && !d }) <= 1)

require(tl_out.d.bits.data.getWidth % wordBits == 0)

val data_arrays = Seq.tabulate(tl_out.d.bits.data.getWidth / wordBits) {
  i =>
  DescribedSRAM(
    name = s"data_arrays_${i}",
    desc = "ICache Data Array",
    size = nSets * refillCycles,
    data = Vec(nWays, UInt(width = dECC.width(wordBits)))
  )
}

for (((data_array, omSRAM), i) <- data_arrays zipWithIndex)

```

— explanation —

`s1_tl_error(i)` is wired with result of AND operation between `tagMatch` and `tl_error.asBool`.

`s1_tag_hit(i)` is wired with result of OR operation between `tagMatch` and `scratchpadHit`.

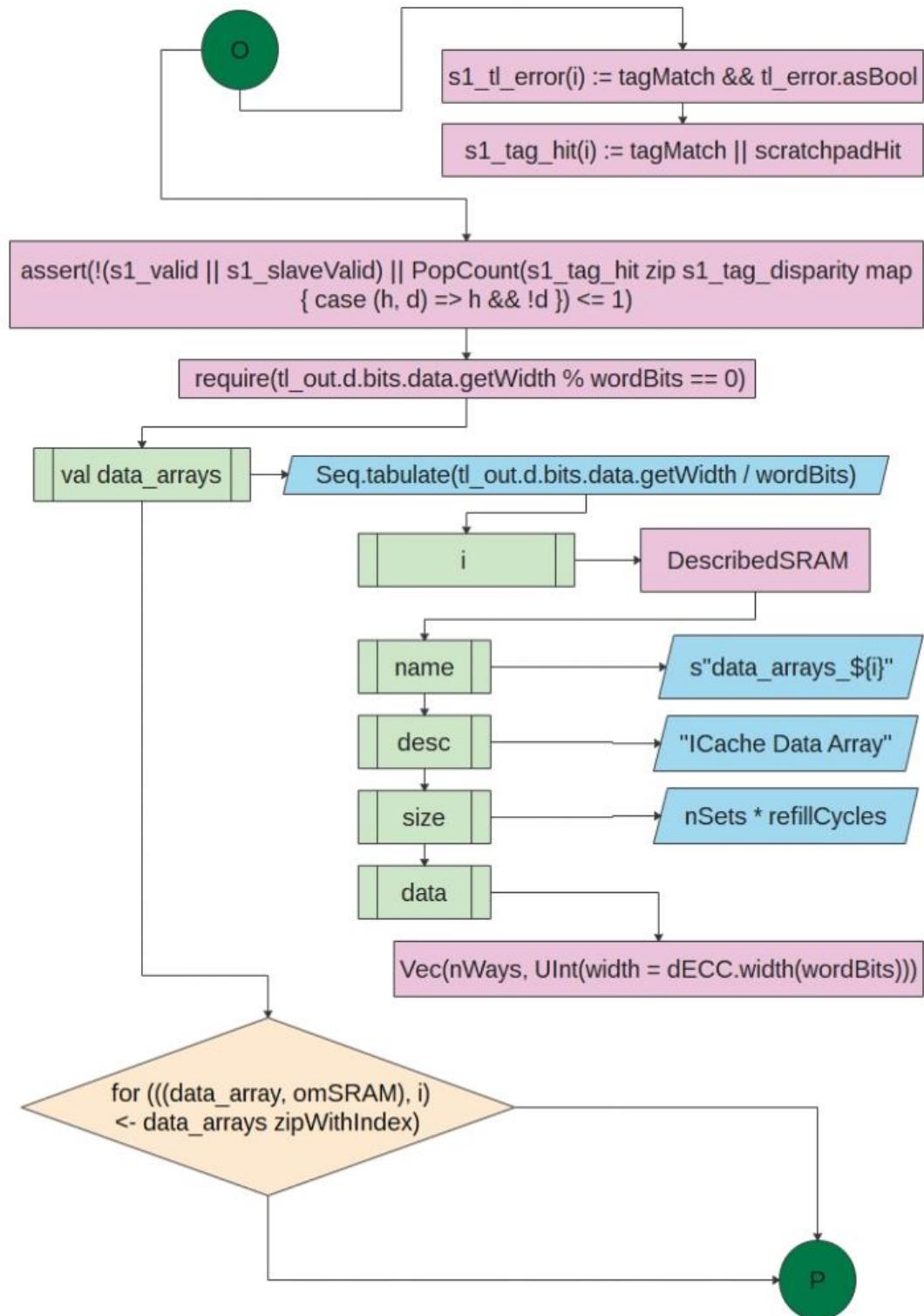
In assert function, OR operation is perform between `s1_valid`, `s1_slaveValid`) and `PopCount`, then invert the result of OR operation and pass a parameter in assert function.

In require function, modulus of `tl_out.d.bits.data.getWidth` and `wordBits` is equal to zero then `bool(True)` is pass as a parameter of require function otherwise `bool(False)` is a parameter of require function.

In `data_arrays`, Sequence of tabulate form with size equal to division of `tl_out.d.bits.data.getWidth` and `wordBits`.

Then I is map with `DescribedSRAM` function.

In `DescribedSRAM` function, pass name, desc, size and data as a parameters with values `s"data_arrays_${i}"`, "ICache Tag Array", `nSets * refillCycles`, `Vec(nWays, UInt(width = dECC.width(wordBits)))` respectively.



```

def wordMatch(addr: UInt) =
addr.extract(log2Ceil(tl_out.d.bits.data.getWidth/8)-1,
log2Ceil(wordBits/8)) === i
def row(addr: UInt) = addr(untagBits-1, blockOffBits-
log2Ceil(refillCycles))
val s0_ren = (s0_valid && wordMatch(s0_vaddr)) || (s0_slaveValid &&
wordMatch(s0_slaveAddr))
val wen = (refill_one_beat && !invalidated) || (s3_slaveValid &&
wordMatch(s1s3_slaveAddr))
val mem_idx = Mux(refill_one_beat, (refill_idx <<
log2Ceil(refillCycles)) | refill_cnt,
Mux(s3_slaveValid, row(s1s3_slaveAddr),
Mux(s0_slaveValid, row(s0_slaveAddr),
row(s0_vaddr))))
when (wen) {
    val data = Mux(s3_slaveValid, s1s3_slaveData,
tl_out.d.bits.data(wordBits*(i+1)-1, wordBits*i))
    val way = Mux(s3_slaveValid, scratchpadWay(s1s3_slaveAddr), repl_way)
    data_array.write(mem_idx, Vec.fill(nWays)(dECC.encode(data)), (0
until nWays).map(way == _))
}

```

Explanation

Create wordMatch function and declared addr as unsigned integer.

Then in addr variable, addr.extract function is call log2Ceil(tl_out.d.bits.data.getWidth/8)-1 and log2Ceil(wordBits/8)) === I are pass as parameters.

Create row function and declared addr as unsigned integer.

Then in addr variable, addr function is call untagBits-1 and blockOffBits-log2Ceil(refillCycles) are pass as parameters.

In s0_ren, OR operation is perform between the result of AND operations which is perform between s0_valid and wordMatch(s0_vaddr) and s0_slaveValid && wordMatch(s0_slaveAddr) respectively.

In wen, OR operation is perform between the result of AND operations which is perform between refill_one_beat and not invalidated and s3_slaveValid and wordMatch(s1s3_slaveAddr) respectively.

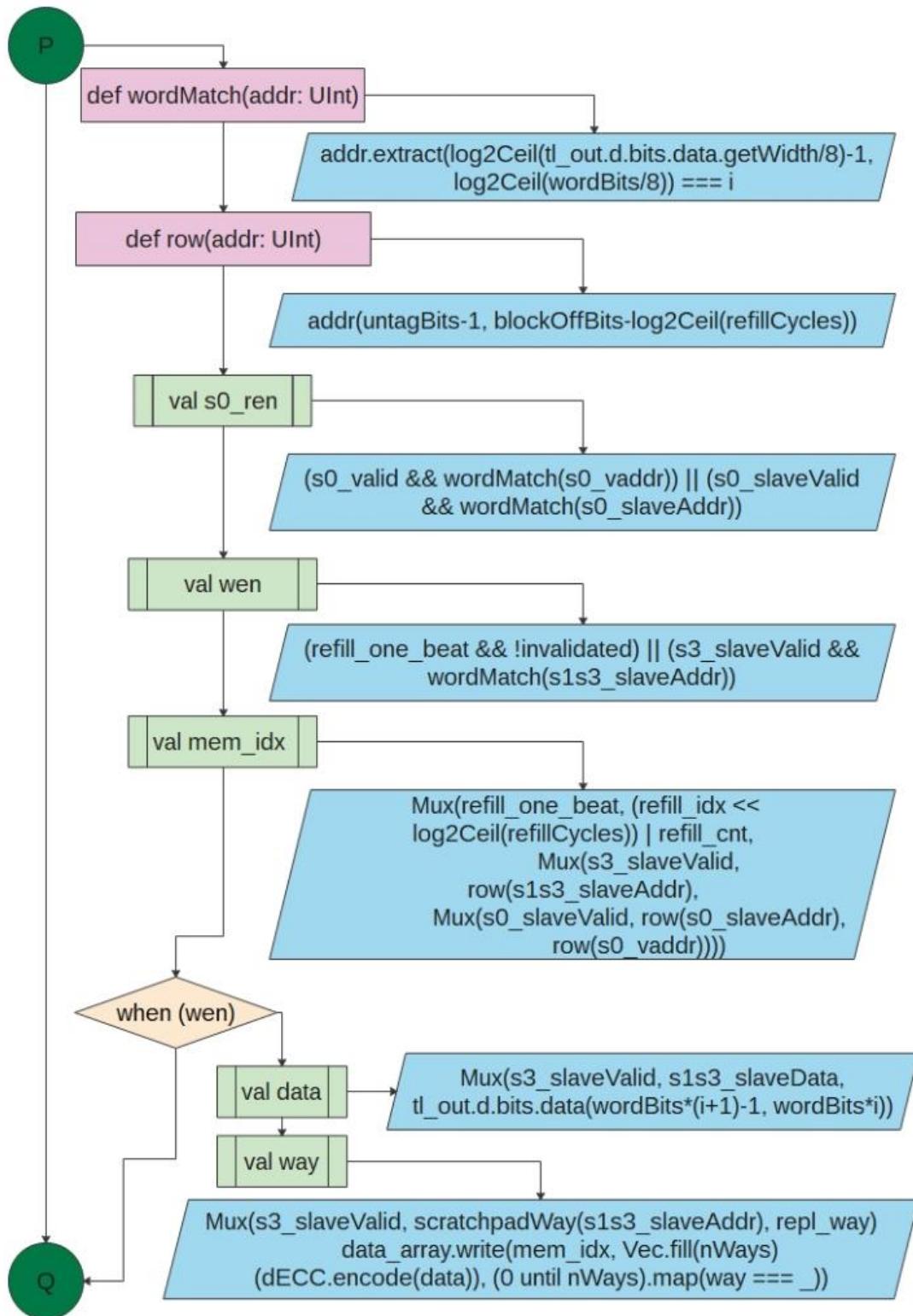
In mem_idx, Bitwise OR operation is perform between Mux(refill_one_beat, (refill_idx << log2Ceil(refillCycles)) and refill_cnt,Mux(s3_slaveValid, row(s1s3_slaveAddr),Mux(s0_slaveValid, row(s0_slaveAddr),row(s0_vaddr)))).

When wen is True then:

In data variable, create a MUX with s3_slaveValid, s1s3_slaveData, and tl_out.d.bits.data(wordBits*(i+1)-1 as a input of Mux and wordBits*I as a select pin of Mux.

In way, create a MUX with s3_slaveValid and scratchpadWay(s1s3_slaveAddr as a input of Mux and repl_way as a select pin of Mux.

Then data_array will write



```

val dout = data_array.read(mem_idx, !wen && s0_ren)
when (wordMatch(Mux(s1_slaveValid, s1s3_slaveAddr, io.s1_paddr))) {
    s1_dout := dout
}
}

val s1_clk_en = s1_valid || s1_slaveValid
val s2_tag_hit = RegEnable(s1_tag_hit, s1_clk_en)
val s2_hit_way = OHToUInt(s2_tag_hit)
val s2_scratchpad_word_addr = Cat(s2_hit_way, Mux(s2_slaveValid,
s1s3_slaveAddr, io.s2_vaddr)(untagBits-1, log2Ceil(wordBits/8)), UInt(0,
log2Ceil(wordBits/8)))
val s2_dout = RegEnable(s1_dout, s1_clk_en)
val s2_way_mux = Mux1H(s2_tag_hit, s2_dout)

val s2_tag_disparity = RegEnable(s1_tag_disparity, s1_clk_en).asUInt.orR
val s2_tl_error = RegEnable(s1_tl_error.asUInt.orR, s1_clk_en)

```

 explanation

In dout variable, data_array reads mem_idx and !wen && s0_ren.

In when condition, wordMatch function is call in which Mux with s1_slaveValid and s1s3_slaveAddr, as a input and io.s1_paddr as a select pin are pass.

If wordMatch function return True then s1_dout is wired with dout.

In s1_clk_en, OR operation is perform between s1_valid s1_slaveValid.

In s2_tag_hit, RegEnable build-in function of chisel is call, s1_tag_hit and s1_clk_en are pass as parameters.

In s2_hit_way, OHToUInt build-in function of chisel is call, s2_tag_hit is pass as parameters.

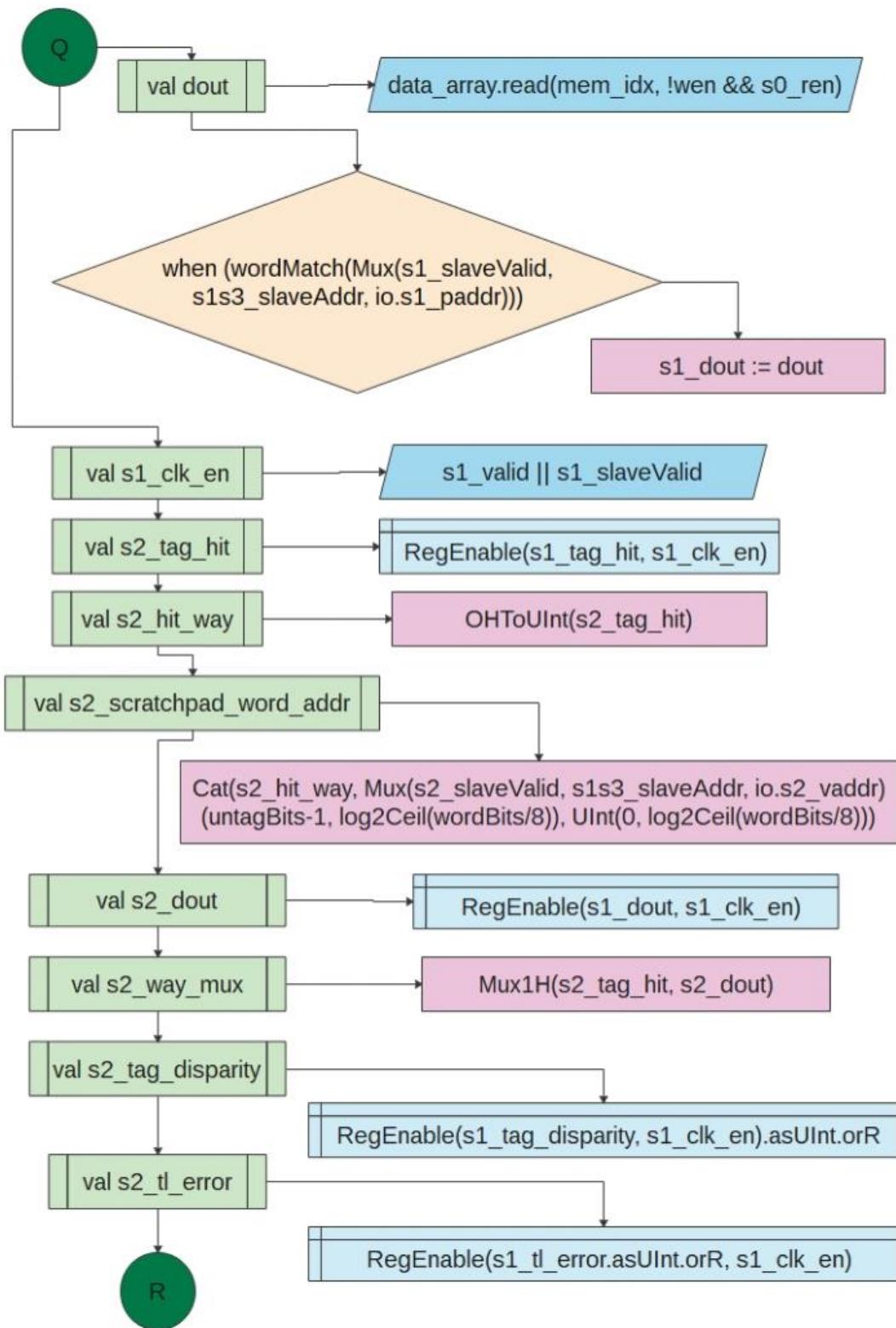
In s2_scratchpad_word_addr, Concatenate s2_hit_way, Mux(s2_slaveValid, s1s3_slaveAddr, io.s2_vaddr)(untagBits-1, log2Ceil(wordBits/8)) and UInt(0, log2Ceil(wordBits/8))).

In s2_dout, RegEnable build-in function of chisel is call, s1_dout and s1_clk_en are pass as parameters.

In s2_way_mux, create Mux1H and pass s2_tag_hit and s2_dout as inputs.

In s2_tag_disparity, RegEnable build-in function of chisel is call, s1_tag_disparity and s1_clk_en as unsigned integers are pass as parameters.

In s2_tl_error, RegEnable build-in function of chisel is call, s1_tl_error as unsigned integer and s1_clk_en are pass as parameters.



```

val s2_data_decoded = dECC.decode(s2_way_mux)
val s2_disparity = s2_tag_disparity || s2_data_decoded.error
val s2_full_word_write = Wire(init = false.B)
val s1_scratchpad_hit = Mux(s1_slaveValid,
lineInScratchpad(scratchpadLine(s1s3_slaveAddr)),
addrInScratchpad(io.s1_paddr))
val s2_scratchpad_hit = RegEnable(s1_scratchpad_hit, s1_clk_en)
val s2_report_uncorrectable_error = s2_scratchpad_hit &&
s2_data_decoded.uncorrectable && (s2_valid || (s2_slaveValid &&
!s2_full_word_write))
val s2_error_addr = scratchpadBase.map(base => Mux(s2_scratchpad_hit,
base + s2_scratchpad_word_addr, 0.U)).getOrElse(0.U)
// output signals
outer.icacheParams.latency match {
  case 1 =>

```

 explanation

s2_data_decoded is declared as dECC.decode (s2_way_mux).

In s2_disparity, OR operation is perform between s2_tag_disparity and s2_data_decoded.error.

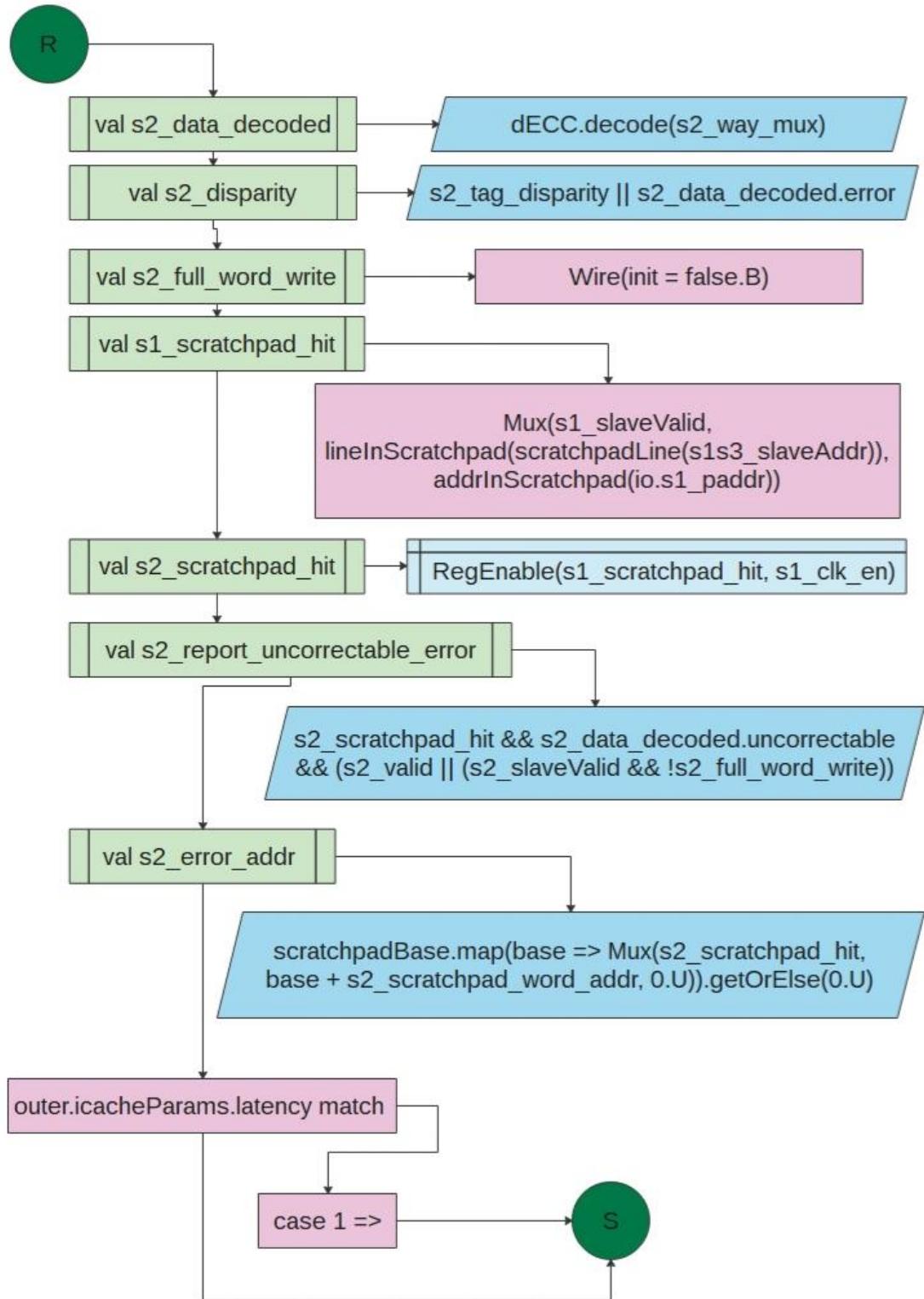
In s2_full_word_write, Wire build-in function of chisel is call, with value false.B

In s1_scratchpad_hit, create Mux and pass s1_slaveValid, lineInScratchpad(scratchpadLine function with parameter s1s3_slaveAddr and addrInScratchpad function with parameter io.s1_paddr) as inputs.

In s2_scratchpad_hit, RegEnable build-in function of chisel is call, s1_scratchpad_hit and s1_clk_en are pass as parameters.

In s2_report_uncorrectable_error , AND operation is perform between s2_scratchpad_hit, s2_data_decoded.uncorrectable and result of OR operation between s2_valid and (s2_slaveValid && !s2_full_word_write)

In s2_error_addr, scratchpadBas is map with (base => Mux(s2_scratchpad_hit, base + s2_scratchpad_word_addr, 0.U)).getOrElse scala function with parameter unsigned zero.



```

require(tECC.isInstanceOf[IdentityCode])
require(dECC.isInstanceOf[IdentityCode])
require(outer.icacheParams.itimAddr.isEmpty)
io.resp.bits.data := Mux1H(s1_tag_hit, s1_dout)
io.resp.bits.ae := s1_tl_error.asUInt.orR
io.resp.valid := s1_valid && s1_hit

case 2 =>
    // when some sort of memory bit error have occurred
    when (s2_valid && s2_disparity) { invalidate := true }
    io.resp.bits.data := s2_data_decoded.uncorrected
    io.resp.bits.ae := s2_tl_error
    io.resp.bits.replay := s2_disparity
    io.resp.valid := s2_valid && s2_hit
    io.errors.correctable.foreach { c =>
        c.valid := (s2_valid || s2_slaveValid) && s2_disparity &&
    !s2_report_uncorrectable_error
        c.bits := s2_error_addr
    }
}

```

explanation

If outer.icacheParams.latency where latency is a parameter of ICacheParams class match then there are 2 cases.

Case 1: (IN ABOVE CODE)

Require is a build-in function of scala.

Require function is call with parameter tECC.isInstanceOf[IdentityCode]

Require function is call with parameter dECC.isInstanceOf[IdentityCode]

Require function is call with parameter outer.icacheParams.itimAddr.isEmpty

io.resp.bits.data is wired with Mux1H(s1_tag_hit, s1_dout).

io.resp.bits.ae is wired with s1_tl_error as unsigned integer.

io.resp.valid is wired with result of AND operation between s1_valid and s1_hit.

Case 2:

when result of AND operation between s2_valid and s2_disparity is True. Then invalidate variable is wired with true.

io.resp.bits.data is wired with s2_data_decoded.uncorrected

io.resp.bits.ae is wired with s2_tl_error

io.resp.bits.replay is wired with s2_disparity

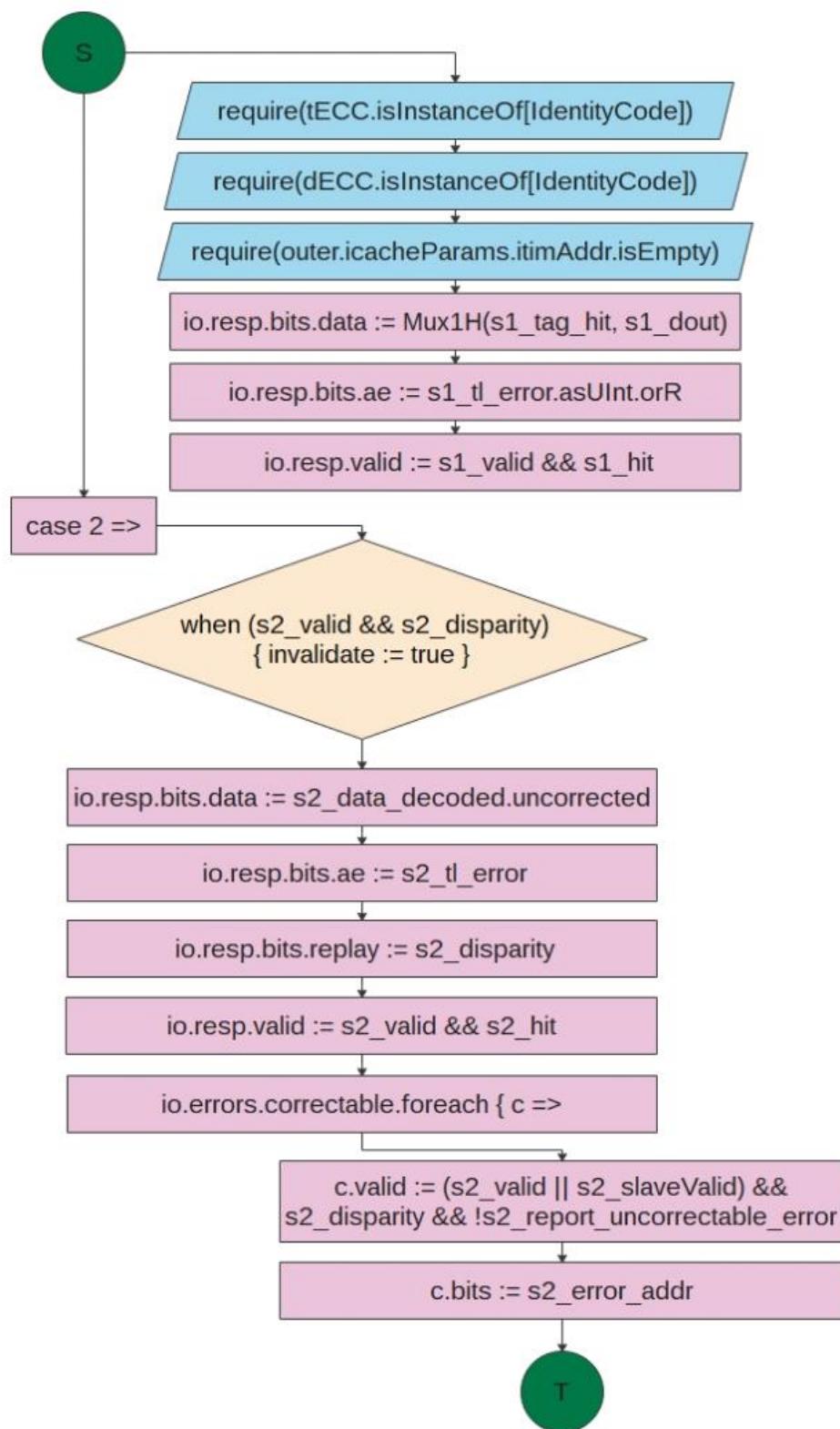
io.resp.valid is wired with result of AND operation between s2_valid and s2_hit

To iterate over a Scala [List](#), foreach method is used.

io.errors.correctable is list in which foreach method is used for iteration over list. c is the instance through which elements of list can access.

During iteration on list, c.valid is wired with result of AND operation between (s2_valid OR s2_slaveValid) and s2_disparity and not s2_report_uncorrectable_error.

During iteration on list, c.bits is wired with s2_error_addr



```

io.errors.uncorrectable.foreach { u =>
    u.valid := s2_report_uncorrectable_error
    u.bits := s2_error_addr
}
tl_in.map { tl =>
    val respValid = RegInit(false.B)
    tl.a.ready := !(tl_out.d.valid || s1_slaveValid || s2_slaveValid || s3_slaveValid || respValid || !io.clock_enabled)
    val s1_a = RegEnable(tl.a.bits, s0_slaveValid)
    s2_full_word_write := edge_in.get.hasData(s1_a) && s1_a.mask.andR
    when (s0_slaveValid) {
        val a = tl.a.bits
        s1s3_slaveAddr := tl.a.bits.address
        s1s3_slaveData := tl.a.bits.data
    }
}

```

 explanation

To iterate over a Scala [List](#), foreach method is used.

io.errors.correctable is list in which foreach method is used for iteration over list. U is the instance through which elements of list can access.

During iteration on list, u.valid is wired with s2_report_uncorrectable_error and u.bits is wired with s2_error_addr.

During tl_in mapping.

In respValid, register is initialized with value false.B.

tl.a.ready is wired with invert result of OR operation which is perform between tl_out.d.valid, s1_slaveValid, s2_slaveValid , s3_slaveValid, respValid, not io.clock_enabled)

In s1_a, RegEnable build-in function of chisel is call, tl.a.bits and s0_slaveValid are pass as parameters.

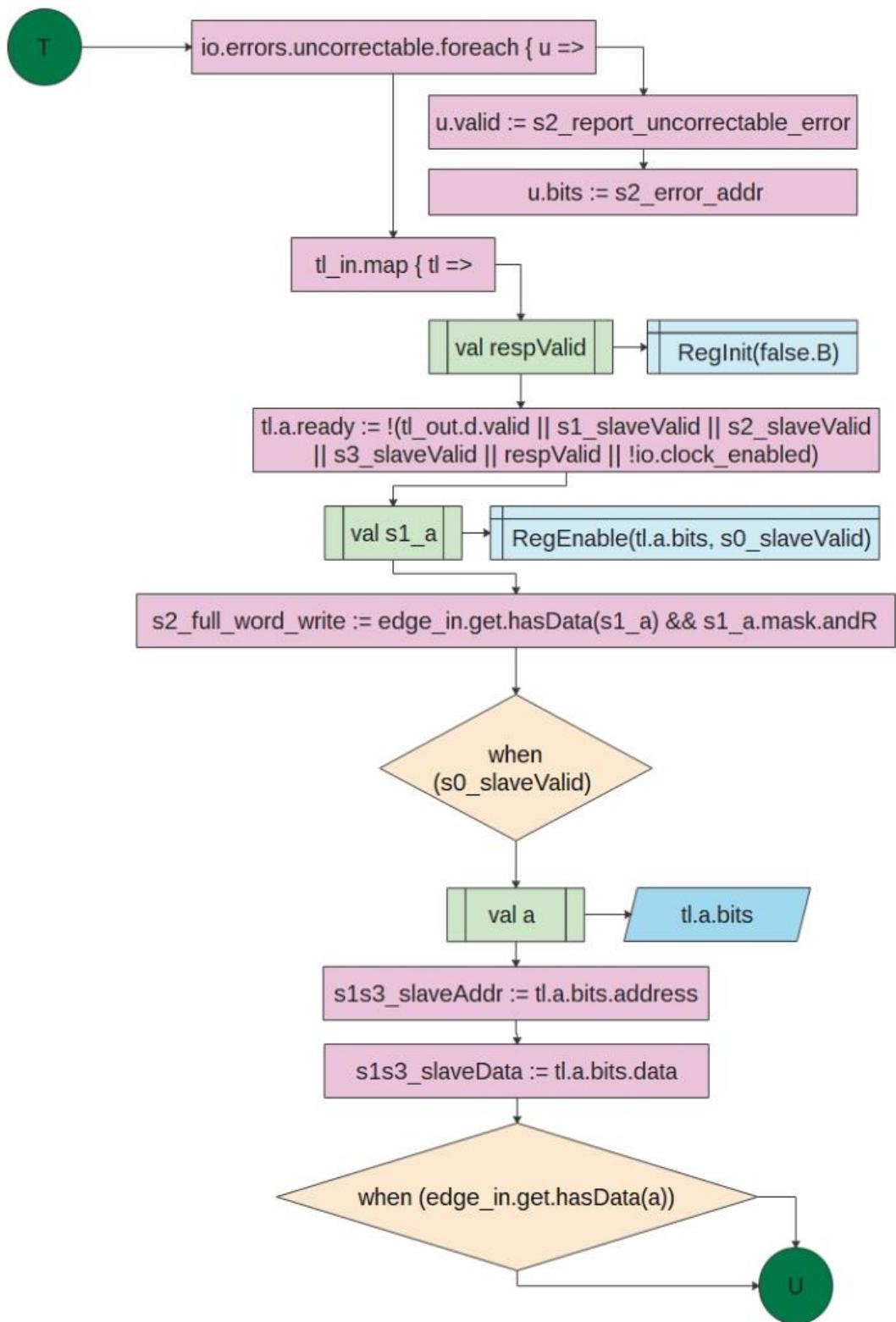
s2_full_word_write is wired with the result of AND operation between edge_in.get.hasData(s1_a) and s1_a.mask.andR.

when s0_slaveValid is True then:

variable a is assigned as tl.a.bits.

s1s3_slaveAddr is wired with tl.a.bits.address.

s1s3_slaveData is wired with tl.a.bits.data



```

when (edge_in.get.hasData(a)) {
    val enable = scratchpadWayValid(scratchpadWay(a.address))
    when (!lineInScratchpad(scratchpadLine(a.address))) {
        scratchpadMax.get := scratchpadLine(a.address)
        invalidate := true
    }
    scratchpadOn := enable

    val itim_allocated = !scratchpadOn && enable
    val itim_deallocated = scratchpadOn && !enable
    val itim_increase = scratchpadOn && enable &&
scratchpadLine(a.address) > scratchpadMax.get
    val refilling = refill_valid && refill_cnt > 0
    ccover(itim_allocated, "ITIM_ALLOCATE", "ITIM allocated")
    ccover(itim_allocated && refilling,
"ITIM_ALLOCATE_WHILE_REFILL", "ITIM allocated while I$ refill")
    ccover(itim_deallocated, "ITIM_DEALLOCATE", "ITIM deallocated")
    ccover(itim_deallocated && refilling,
"ITIM_DEALLOCATE_WHILE_REFILL", "ITIM deallocated while I$ refill")
    ccover(itim_increase, "ITIM_SIZE_INCREASE", "ITIM size
increased")
    ccover(itim_increase && refilling,
"ITIM_SIZE_INCREASE_WHILE_REFILL", "ITIM size increased while I$ refill")
}
}

```

- explanation -

when edge_in.get.hasData(a) is True.Then:

In enable variable, scratchpadWayValid function is call with parameter scratchpadWay(a.address)

When `lineInScratchpad(scratchpadLine` function with parameter `(a.address)` returns `False` then:

`scratchpadMax.get` is wired with `scratchpadLine(a.address)` and `invalidate` is wired with `true`.

scratchpadOn is wired with enable

In `itm_allocated`, result of AND operation between `not scratchpadOn` and `enable` is assigned.

In `itim_deallocated`, result of AND operation between `scratchpadOn` and `not enable` is assigned.

In `itlm_increas`, result of AND operation between `scratchpadOn` and `enable` and `scratchpadLine(a.address) > scratchpadMax.get` is assigned.

In refilling , result of AND operation between refill_valid and refill_cnt > 0 is assigned.

Ccover function is call and itim_allocated, "ITIM_ALLOCATE" and "ITIM allocated" are pass as parameters of function.

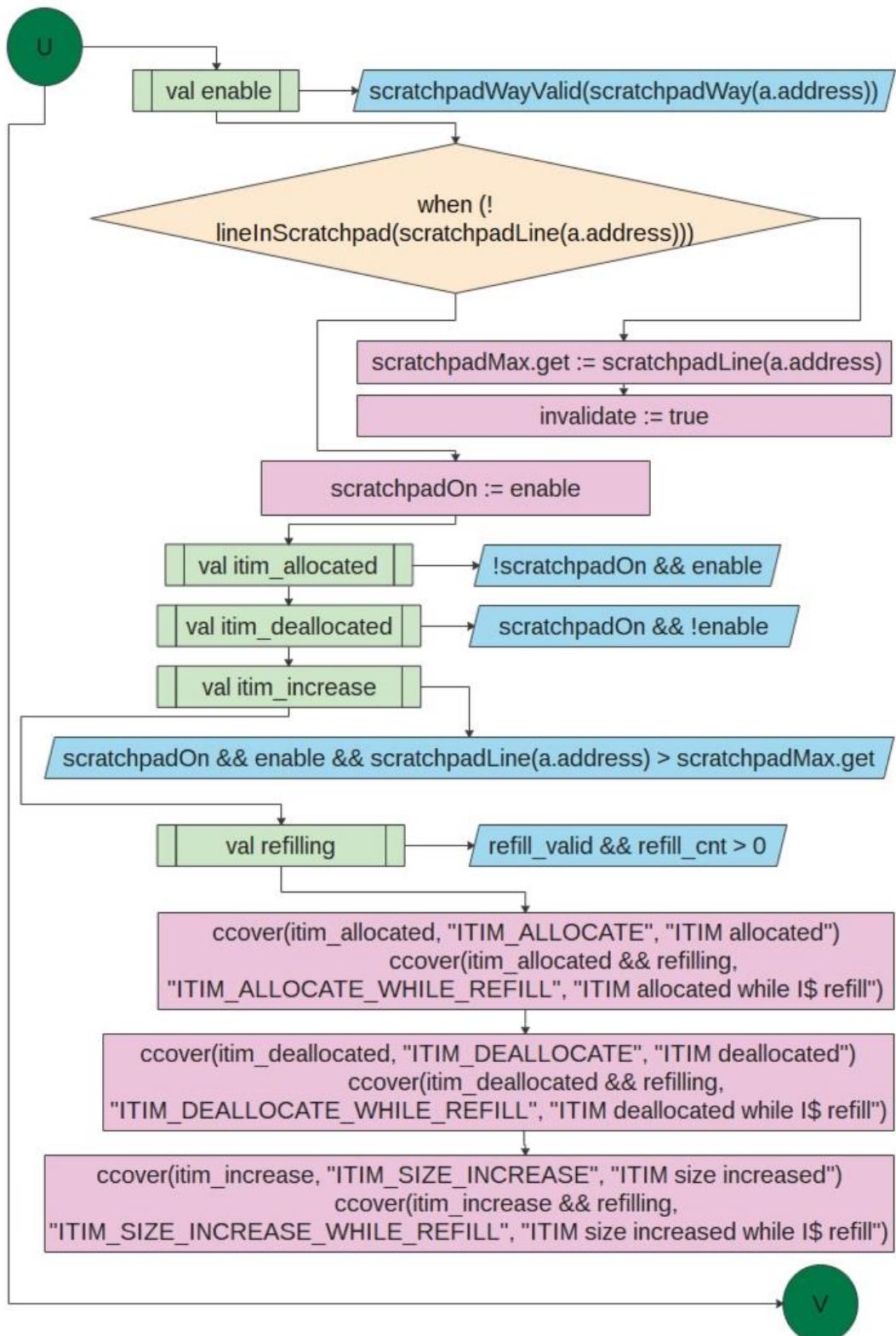
Ccover function is call and itim_allocated && refilling, "ITIM_ALLOCATE_WHILE_REFILL" and "ITIM allocated while I\$ refill" are pass as parameters of function.

Ccover function is call and itim_deallocated, "ITIM_DEALLOCATE" and "ITIM deallocated" are pass as parameters of function.

Ccover function is call and itim_deallocated && refilling, "ITIM_DEALLOCATE_WHILE_REFILL" and "ITIM deallocated while I\$ refill" are pass as parameters of function.

Ccover function is call and itim_increase, "ITIM_SIZE_INCREASE" and "ITIM size increased" are pass as parameters of function.

Ccover function is call and itim_increase && refilling, "ITIM_SIZE_INCREASE_WHILE_REFILL" and "ITIM size increased while I\$ refill" are pass as parameters of function.



```

assert(!s2_valid || RegNext(RegNext(s0_vaddr)) === io.s2_vaddr)
  when (!(tl.a.valid || s1_slaveValid || s2_slaveValid || respValid)
        && s2_valid && s2_data_decoded.error && !s2_tag_disparity) {
    // handle correctable errors on CPU accesses to the scratchpad.
    // if there is an in-flight slave-port access to the scratchpad,
    // report the a miss but don't correct the error (as there is
    // a structural hazard on s1s3_slaveData/s1s3_slaveAddress).
    s3_slaveValid := true
    s1s3_slaveData := s2_data_decoded.corrected
    s1s3_slaveAddr := s2_scratchpad_word_addr |
      s1s3_slaveAddr(log2Ceil(wordBits/8)-1, 0)
  }
  respValid := s2_slaveValid || (respValid && !tl.d.ready)
  val respError = RegEnable(s2_scratchpad_hit &&
    s2_data_decoded.uncorrectable && !s2_full_word_write, s2_slaveValid)
  when (s2_slaveValid) {

```

explanation

Then assert function is call which is scala build-in function. In assert function, !s2_valid is getting logically ORed (If the two operands are non zero then condition becomes true) with RegNext(RegNext(s0_vaddr)) and is equals to io.s2_vaddr.

If the two operands are non zero then condition becomes true.

The triple equals operator === is normally the Scala type-safe equals operator, overrides with a method in Column to create a new Column object that compares the Column to the left with the object on the right, returning a boolean.

When tl.a.valid , s1_slaveValid , s2_slaveValid and respValid get logically ORed (using OR operator i.e ||) along with s2_valid, s2_data_decoded.error, !s2_tag_disparity get logically AND (If both the operands are non-zero then condition becomes true) :

s3_slaveValid wired to true

s1s3_slaveData wired to s2_data_decoded.corrected

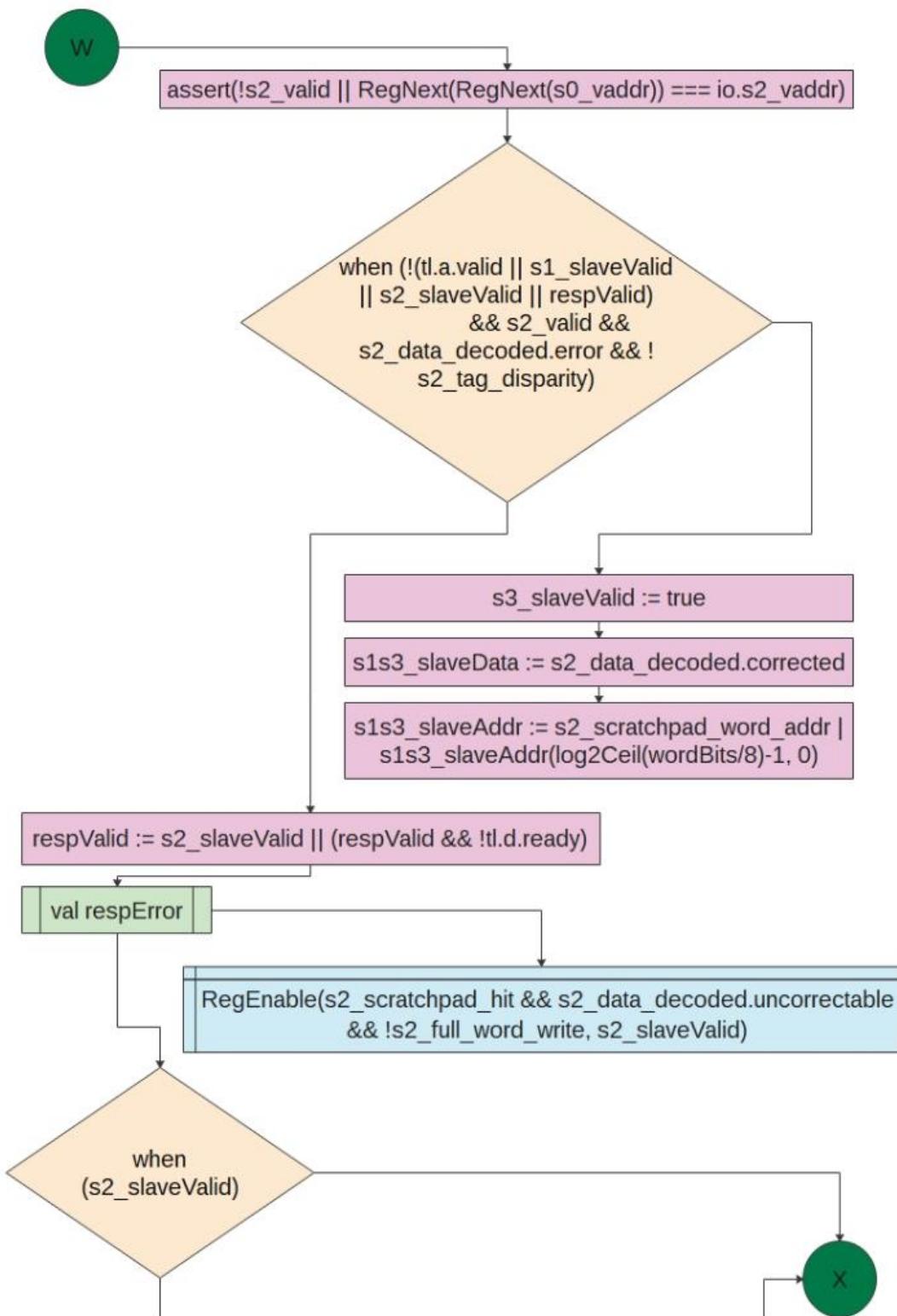
s1s3_slaveAddr wired to s2_scratchpad_word_addr get OR (BINARY) with s1s3_slaveAddr, integer with width of log2Ceil word bits.

log2Ceil is useful for getting the number of bits needed to represent some number of states of desired condition.

respValid wired to s2_slaveValid and there is logical OR operator between (respValid logically AND with !tl.d.ready).

immutable variable named respError initialized. Regenable function is called which chisel build-in function to enable register and performing AND operation among four parameters s2_scratchpad_hit , s2data_decoded.uncorrectable , !s2_full_word_write, s2_slaveValid

when condition occurs with s2_slaveValid



```

when (edge_in.get.hasData(s1_a) || s2_data_decoded.error) { s3_slaveValid
:= true }
    def byteEn(i: Int) = !(edge_in.get.hasData(s1_a) && s1_a.mask(i))
    s1s3_slaveData := (0 until wordBits/8).map(i => Mux(byteEn(i),
s2_data_decoded.corrected, s1s3_slaveData)(8*(i+1)-1, 8*i)).asUInt
}
    tl.d.valid := respValid
    tl.d.bits := Mux(edge_in.get.hasData(s1_a),
edge_in.get.AccessAck(s1_a),
edge_in.get.AccessAck(s1_a, UInt(0), denied = Bool(false),
corrupt = respError))
    tl.d.bits.data := s1s3_slaveData

    // Tie off unused channels
    tl.b.valid := false
    tl.c.ready := true
    tl.e.ready := true

    ccover(s0_valid && s1_slaveValid, "CONCURRENT_ITIM_ACCESS_1", "ITIM
accessed, then I$ accessed next cycle")
    ccover(s0_valid && s2_slaveValid, "CONCURRENT_ITIM_ACCESS_2", "ITIM
accessed, then I$ accessed two cycles later")
    ccover(tl.d.valid && !tl.d.ready, "ITIM_D_STALL", "ITIM response
blocked by D-channel")
    ccover(tl_out.d.valid && !tl_out.d.ready, "ITIM_BLOCK_D", "D-
channel blocked by ITIM access")
}
}

```

 explanation

When condition is occurred (edge_in.get.hasData(s1_a) and the operation logical OR is performed with s2_data_decoded.error) .

s3_slaveValid wired to true.

Function is initialized named byteEn allotted to !(edge_in.get.hasData(s1_a)) that is getting logically AND with s1_a.mask(i))

s1s3_slaveData wired to (0 until wordBits/8) map with Mux(byteEn(i), s2_data_decoded.corrected, s1s3_slaveData)(8*(i+1)-1, 8*i)) as unsigned integer.

tl.d.valid wired to respValid

tl.d.bits wired to Mux(edge_in.get.hasData(s1_a), edge_in.get.AccessAck(s1_a), edge_in.get.AccessAck(s1_a, UInt(0), denied = Bool(false), corrupt = respError)) to switch one of several input lines through to a single common output line by the application of a control signal.

tl.d.bits.data wired to s1s3_slaveData

Tie off unused channels

tl.b.valid wired to false

tl.c.ready wired to true

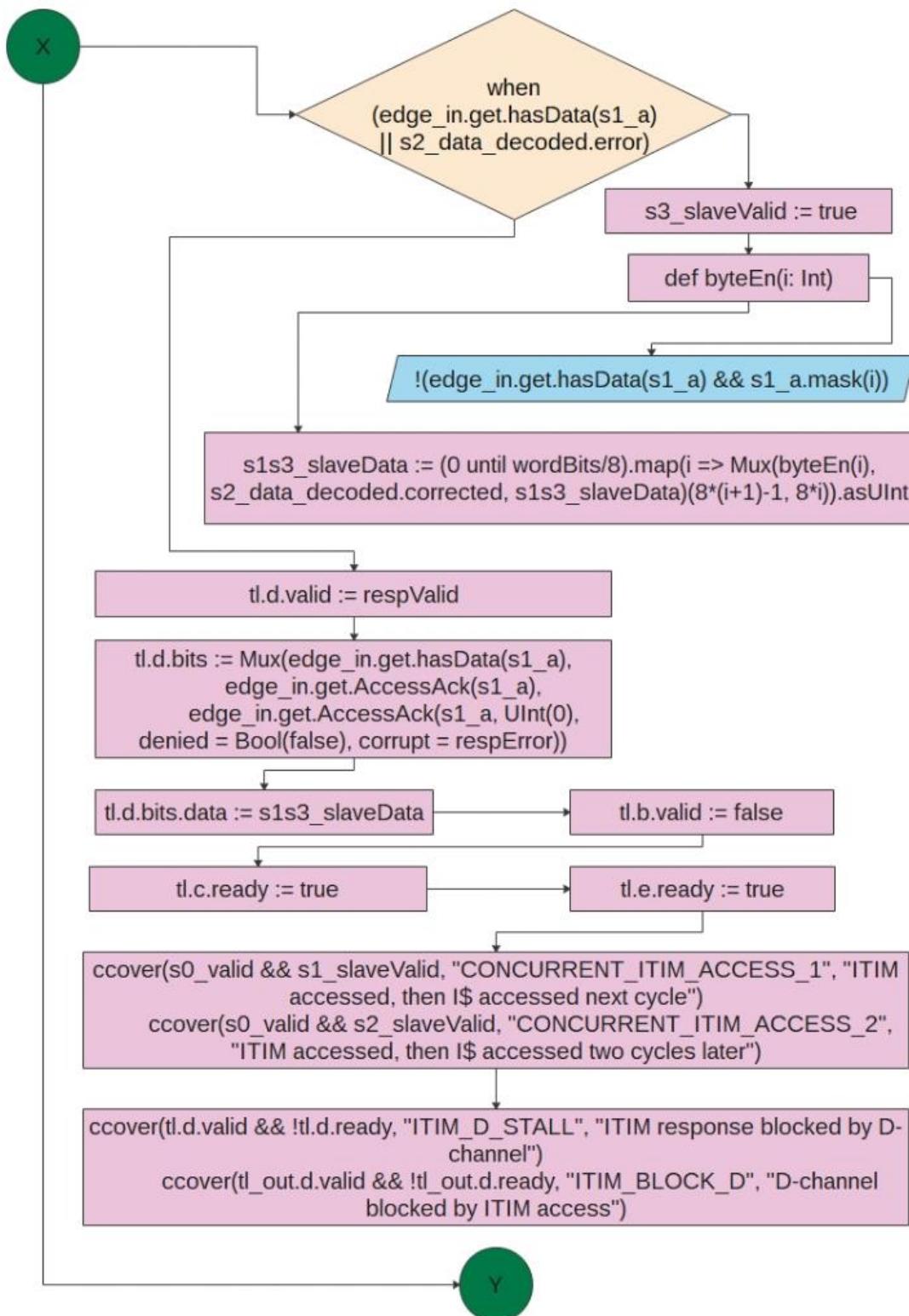
tl.e.ready wired to true

The ccover function is call, in which s0_valid and s1_slaveValid, "CONCURRENT_ITIM_ACCESS_1", "ITIM accessed are pass as a parameter then I\$ accessed next cycle. Here logical AND operation performed between s0_valid and s1_slaveValid

Then ccover function is call, in which s0_valid and s2_slaveValid, "CONCURRENT_ITIM_ACCESS_2", "ITIM accessed are pass as a parameter then I\$ accessed two cycles later. Here logical AND operation performed between s0_valid and s2_slaveValid

Then ccover function is call, in which tl.d.valid and !tl.d.ready, "ITIM_D_STALL" are pass as a parameter then ITIM response blocked by D-channel. Here logical AND operation performed between tl.d.valid and !tl.d.ready

Then ccover function is call, in which tl_out.d.valid and !tl_out.d.ready, "ITIM_BLOCK_D" are pass as a parameter then D-channel blocked by ITIM access. Here logical AND operation performed between tl_out.d.valid and !tl_out.d.ready



```

tl_out.a.valid := s2_request_refill
tl_out.a.bits := edge_out.Get(
    fromSource = UInt(0),
    toAddress = (refill_paddr >> blockOffBits) <<
blockOffBits,
    lgSize = lgCacheBlockBytes)._2

if (cacheParams.prefetch) {
    val (crosses_page, next_block) = Split(refill_paddr(pgIdxBits-1,
blockOffBits) +& 1, pgIdxBits-blockOffBits)
    when (tl_out.a.fire()) {
        send_hint := !hint_outstanding && io.s2_prefetch && !crosses_page
        when (send_hint) {
            send_hint := false
            hint_outstanding := true
        }
    }
    when (refill_done) {
        send_hint := false
    }
    when (tl_out.d.fire() && !refill_one_beat) {
        hint_outstanding := false
    }
}

```

 explanation

tl_out.a.valid wired to s2_request_refill

tl_out.a.bits wired to edge_out.Get

new variable fromSource assigned to UInt(0)

new variable toAddress assigned to:

The Bit positions of the left operand value is moved right by the number of bits specified by the right operand among refill_paddr and blockOffBits. Then bit positions of the left operands value is moved left by the number of bits specified by the right operand.with blockOffBits,

New variable named lgSize assigned to lgCacheBlockBytes)._2

Here is the condition if : cacheParams.prefetch then variable (crosses_page, next_block) get Split(refill_paddr(pgIdxBits-1, blockOffBits) +& 1, pgIdxBits-blockOffBits).), split function scala is call which is used for splitting.

The next condition is when tl_out.a.fire: send_hint wired to !hint_outstanding get logically AND with io.s2_prefetch get logically AND with !crosses_page

Within this condition , another condition occur when (send_hint) :

send_hint wired to false

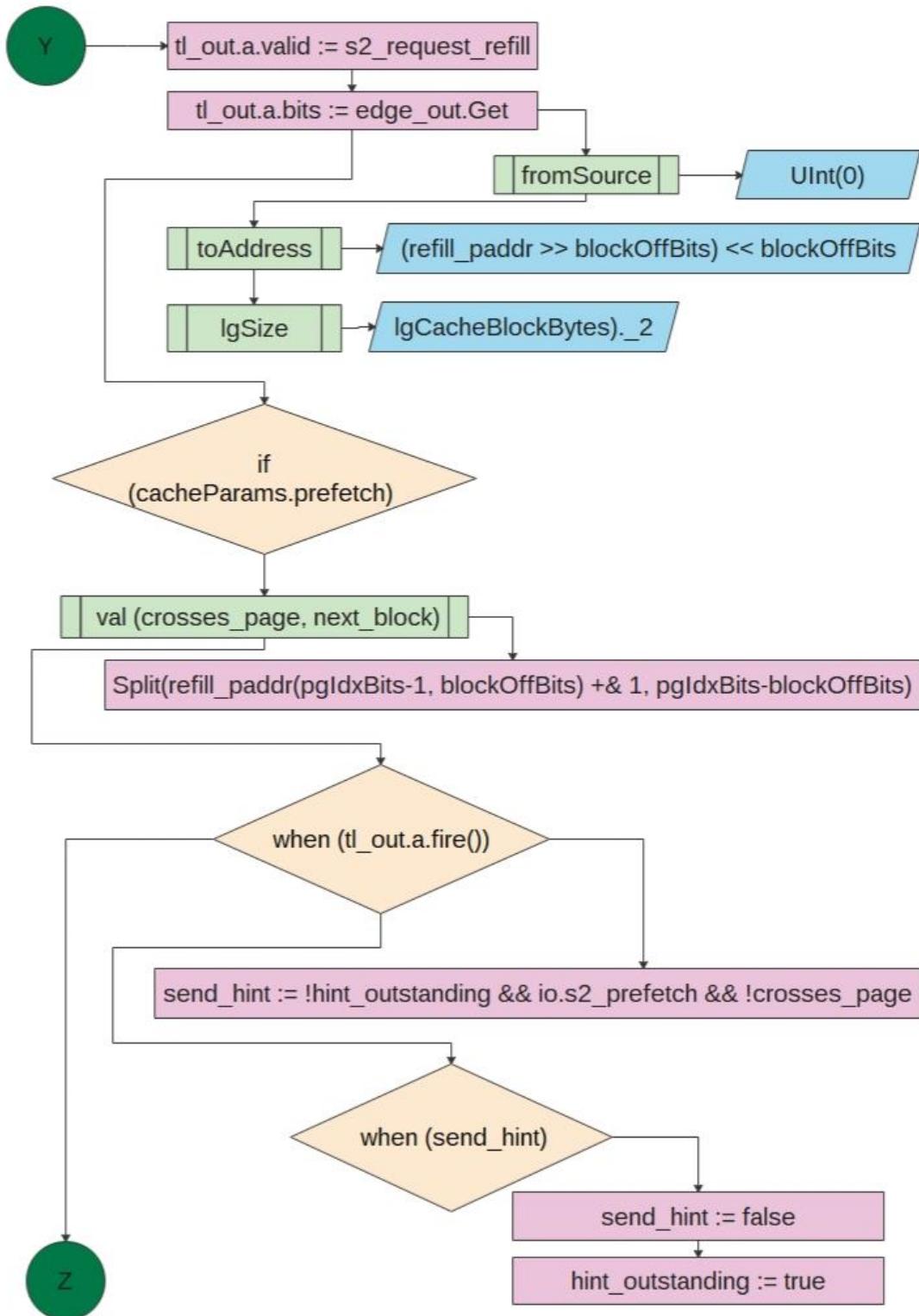
hint_outstanding := true

After that when (refill_done):

send_hint wired to false

and when (tl_out.d.fire() get logically AND with !refill_one_beat) :

hint_outstanding wired to false



```

when (send_hint) {
    tl_out.a.valid := true
    tl_out.a.bits := edge_out.Hint(
        fromSource = UInt(1),
        toAddress = Cat(refill_paddr >> pgIdxBits,
next_block) << blockOffBits,
        lgSize = lgCacheBlockBytes,
        param = TLHints.PREFETCH_READ)._2
}

ccover(send_hint && !tl_out.a.ready, "PREFETCH_A_STALL", "I$ prefetch
blocked by A-channel")
ccover(refill_valid && (tl_out.d.fire() && !refill_one_beat),
"PREFETCH_D_BEFORE_MISS_D", "I$ prefetch resolves before miss")
ccover(!refill_valid && (tl_out.d.fire() && !refill_one_beat),
"PREFETCH_D_AFTER_MISS_D", "I$ prefetch resolves after miss")
ccover(tl_out.a.fire() && hint_outstanding, "PREFETCH_D_AFTER_MISS_A",
"I$ prefetch resolves after second miss")
}

```

 explanation

Here is condition when (send_hint):

tl_out.a.valid wired to true

tl_out.a.bits wired to edge_out.Hint

the output from is Source assigned to UInt(1)

when it comes to Address it concatenate. Binary Right Shift Operator. The Bit positions of the left operand value is moved right by the number of bits specified by the right operand. (refill_paddr >> pgIdxBits, next_block). Binary Left Shift Operator. The bit positions of the left operands value is moved left by the number of bits specified by the right operand. << blockOffBits

Variable lgSize assigned to lgCacheBlockBytes

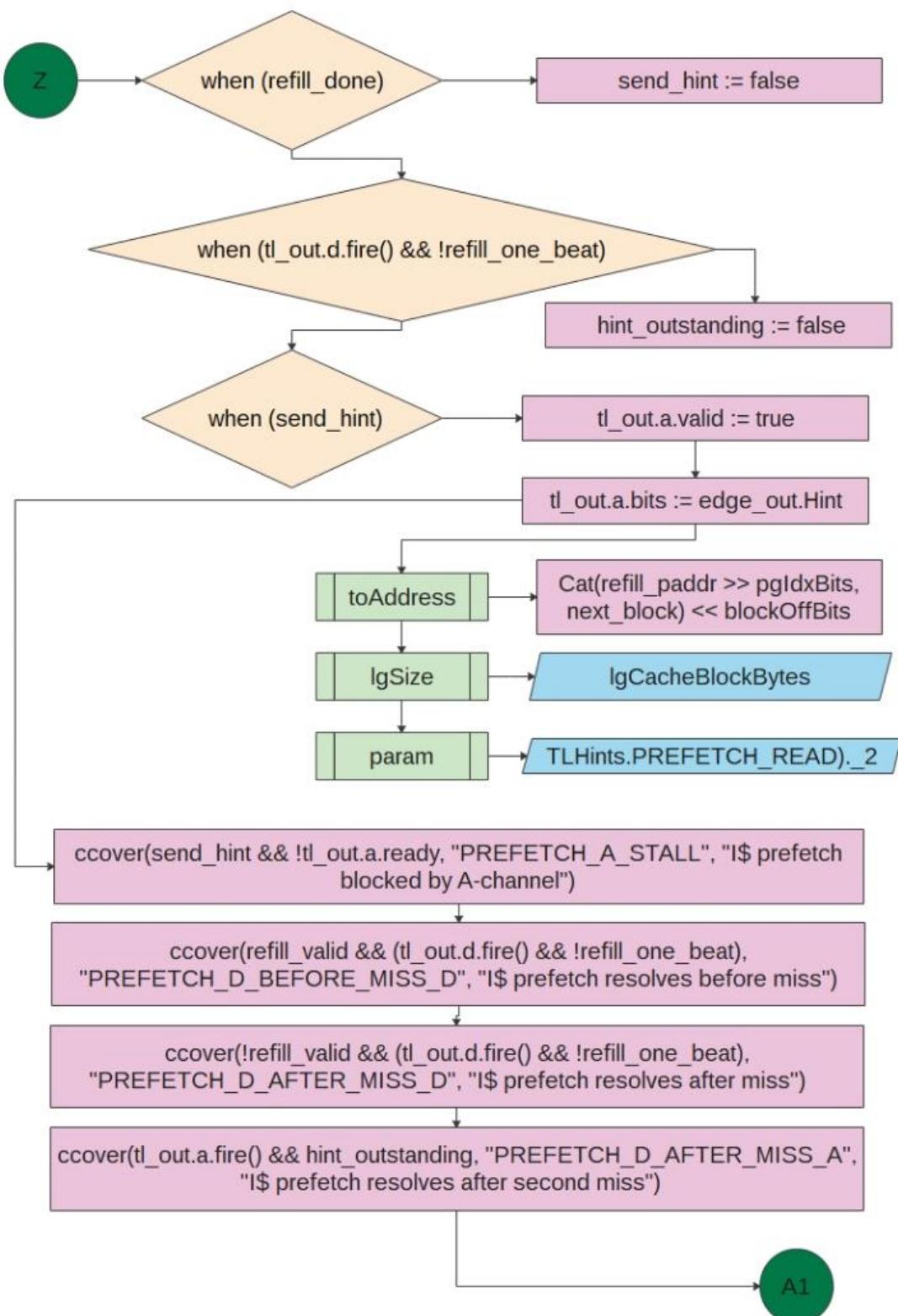
Variable param assigned to TLHints.PREFETCH_READ)._2

Then ccover function is call, in send_hint and !tl_out.a.ready, "PREFETCH_A_STALL" are pass as a parameter I\$ prefetch blocked by A-channel .Here logical AND operation performed between send_hint and !tl_out.a.ready.

ccover function is call refill_valid, (tl_out.d.fire() , !refill_one_beat), "PREFETCH_D_BEFORE_MISS_D are pass as a parameter I\$ prefetch resolves before miss. Here logical AND operation performed between refill_valid, (tl_out.d.fire() and !refill_one_beat)

ccover function is call (!refill_valid , (tl_out.d.fire() , !refill_one_beat), "PREFETCH_D_AFTER_MISS_D are pass as a parameter I\$ prefetch resolves after miss. Here logical AND operation performed between !refill_valid , (tl_out.d.fire() and !refill_one_beat)

ccover function is call (tl_out.a.fire() , hint_outstanding, "PREFETCH_D_AFTER_MISS_A are pass as a parameter I\$ prefetch resolves after second miss. Here logical AND operation performed between tl_out.a.fire() and hint_outstanding



```

tl_out.a.bits.user.lift(AMBAProt).foreach { x =>
    val user_bit_cacheable = true.B
    x.privileged := user_bit_cacheable
    x.bufferable := user_bit_cacheable
    x.modifiable := user_bit_cacheable
    x.readalloc := user_bit_cacheable
    x.writealloc := user_bit_cacheable
    x.fetch := true.B
    x.secure := true.B
}
tl_out.b.ready := Bool(true)
tl_out.c.valid := Bool(false)
tl_out.e.valid := Bool(false)
assert(!(tl_out.a.valid && addrMaybeInScratchpad(tl_out.a.bits.address)))

when (!refill_valid) { invalidated := false.B }
when (refill_fire) { refill_valid := true.B }
when (refill_done) { refill_valid := false.B}

```

— explanation —

Rocket caches all fetch requests, and it's difficult to differentiate privileged/unprivileged on cached data, so mark as privileged:

variable user_bit_cacheable assigned to true.B

Now, enable outer caches for all fetches

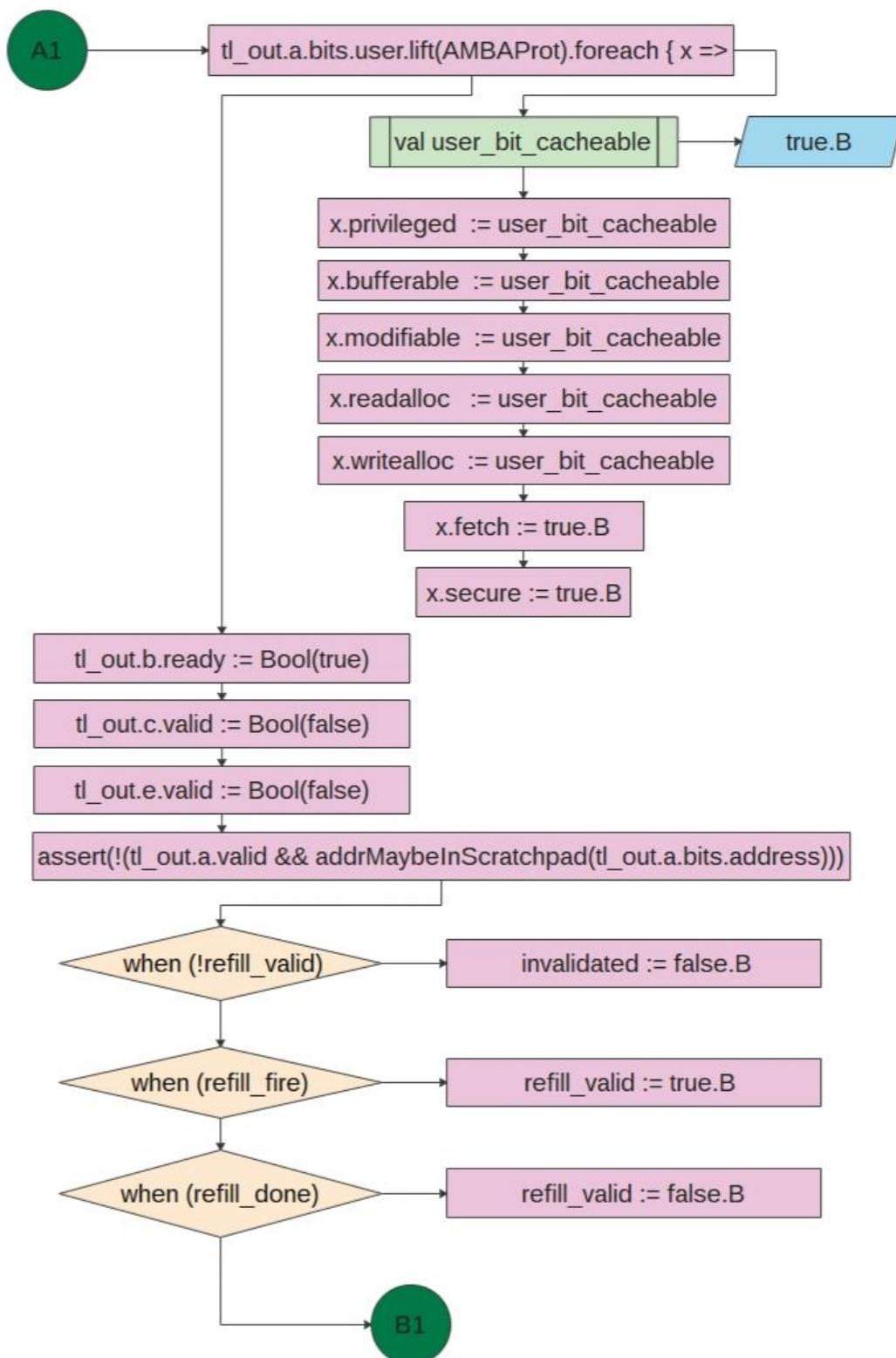
- x.privileged wired to user_bit_cacheable
- x.bufferable wired to user_bit_cacheable
- x.modifiable wired to user_bit_cacheable
- x.readalloc wired to user_bit_cacheable
- x.writealloc wired to user_bit_cacheable

Now, Following are always tied off

- x.fetch wired to true.B
- x.secure wired to true.B
- tl_out.b.ready wired to Bool(true)
- tl_out.c.valid wired to Bool(false)
- tl_out.e.valid wired to Bool(false)

Then assert function is call which is scala build-in function. assert function is call with parameters !(tl_out.a.valid getting logically AND with addrMaybeInScratchpad(tl_out.a.bits.address))

- when** (!refill_valid) : invalidated wired to false.B
- when** (refill_fire) : refill_valid wired to true.B
- when** (refill_done) : refill_valid wired to false.B



```

io.perf.acquire := refill_fire
io.keep_clock_enabled :=
  tl_in.map(tl => tl.a.valid || tl.d.valid || s1_slaveValid ||
s2_slaveValid || s3_slaveValid).getOrElse(false.B) || // ITIM
  s1_valid || s2_valid || refill_valid || send_hint || hint_outstanding
// I$
def index(vaddr: UInt, paddr: UInt) = {
  val lsbs = paddr(pgUntagBits-1, blockOffBits)
  val msbs = (idxBits+blockOffBits >
pgUntagBits).option(vaddr(idxBits+blockOffBits-1, pgUntagBits))
  msbs ## lsbs
}
ccover(!send_hint && (tl_out.a.valid && !tl_out.a.ready), "MISS_A_STALL",
"I$ miss blocked by A-channel")
ccover(invalidate && refill_valid, "FLUSH_DURING_MISS", "I$ flushed
during miss")

def ccover(cond: Bool, label: String, desc: String)(implicit sourceInfo:
SourceInfo) =
  cover(cond, s"ICACHE_$label", "MemorySystem;;" + desc)

val mem_active_valid = Seq(CoverBoolean(s2_valid, Seq("mem_active")))

```

— explanation —

io.perf.acquire wired to refill_fire

io.keep_clock_enabled wired to tl_in.map(tl Relational tl.a.valid ORed with tl.d.valid
ORed with s1_slaveValid ORed with s2_slaveValid ORed with
s3_slaveValid).getOrElse(false.B) ORed with // ITIM

s1_valid ORed with s2_valid ORed with refill_valid ORed with send_hint ORed with
hint_outstanding // I\$

Function is created named :

Variable lsbs assigned to paddr(pgUntagBits-1, blockOffBits)

Variable msbs assigned to (idxBits+blockOffBits >
pgUntagBits).option(vaddr(idxBits+blockOffBits-1, pgUntagBits))

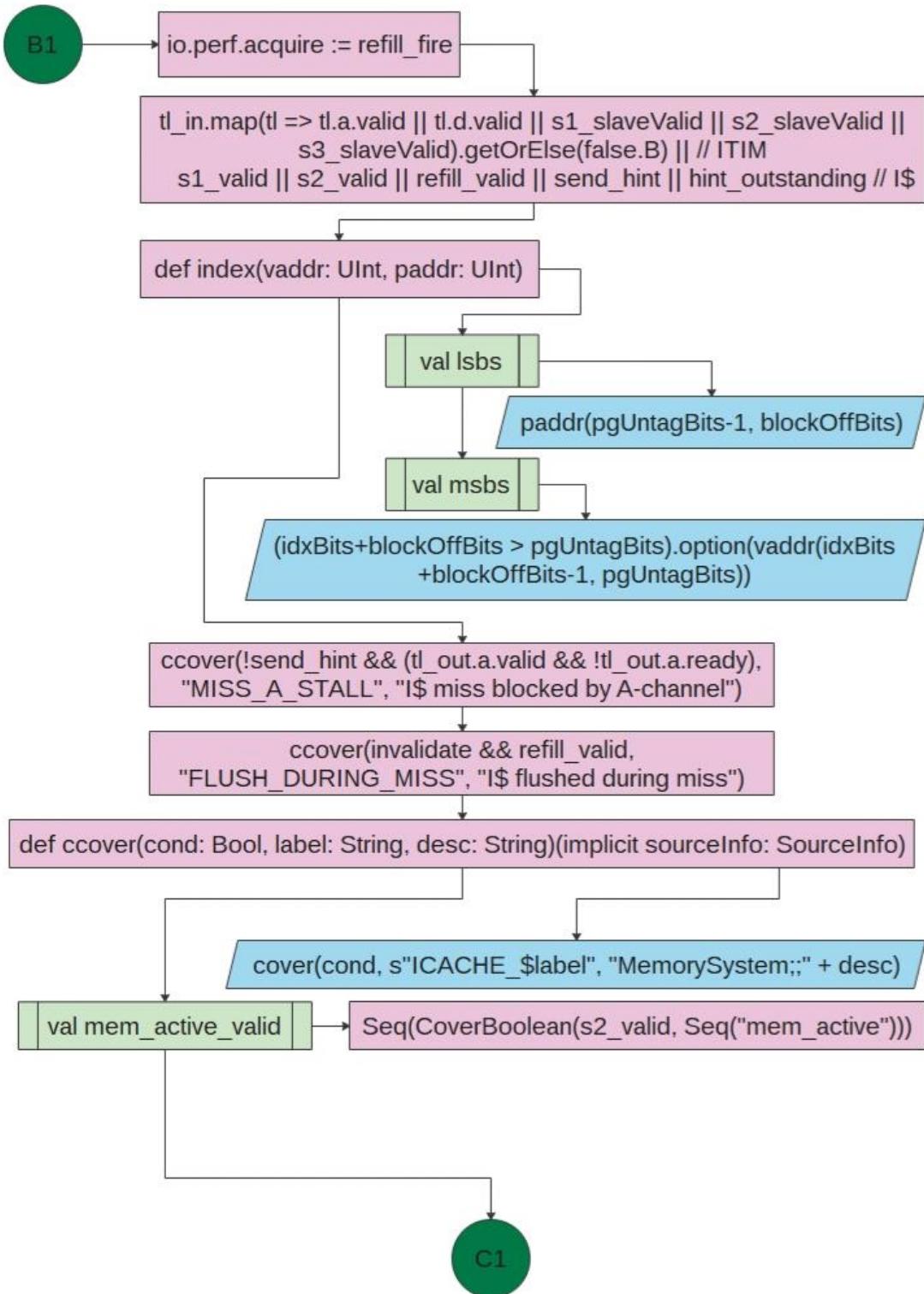
THE REASON BEHIND PERFORMING ABOVE OPERATION IS TO CONFIGURE MOST
SIGNIFICANT BITS AND LEAST SIGNIFICANT BITS.

Then ccover function is call, (!send_hint , (tl_out.a.valid , !tl_out.a.ready),
"MISS_A_STALL", are pass as a parameter I\$ miss blocked by A-channel . Here logical
AND operation performed between , (!send_hint , (tl_out.a.valid and !tl_out.a.ready)

Then ccover function is call, invalidate , refill_valid, "FLUSH_DURING_MISS", are pass as
a parameter I\$ flushed during miss . Here logical AND operation performed between
invalidate and refill_valid

Function ccover is initialized along with data type: Bool, label in the form of: String

Variable mem_active_valid is initialized and assigned to
Seq(CoverBoolean(s2_valid, Seq("mem_active")))



```

val data_error = Seq(
    CoverBoolean(!s2_data_decoded.correctable &&
!s2_data_decoded.uncorrectable, Seq("no_data_error")),
    CoverBoolean(s2_data_decoded.correctable,
Seq("data_correctable_error")),
    CoverBoolean(s2_data_decoded.uncorrectable,
Seq("data_uncorrectable_error")))
val request_source = Seq(
    CoverBoolean(!s2_slaveValid, Seq("from_CPU")),
    CoverBoolean(s2_slaveValid, Seq("from_TL"))
)
val tag_error = Seq(
    CoverBoolean(!s2_tag_disparity, Seq("no_tag_error")),
    CoverBoolean(s2_tag_disparity, Seq("tag_error"))
)
val mem_mode = Seq(
    CoverBoolean(s2_scratchpad_hit, Seq("ITIM_mode")),
    CoverBoolean(!s2_scratchpad_hit, Seq("cache_mode"))
)

val error_cross_covers = new CrossProperty(
    Seq(mem_active_valid, data_error, tag_error, request_source, mem_mode),
    Seq(
        // tag error cannot occur in ITIM mode
        Seq("tag_error", "ITIM_mode"),
        // Can only respond to TL in ITIM mode
        Seq("from_TL", "cache_mode")
    ),
    "MemorySystem;;Memory Bit Flip Cross Covers")

cover(error_cross_covers)

```

 explanation

Variable `data_error` is initialized and assigned to `Seq(`

`CoverBoolean(!s2_data_decoded.correctable logically AND with
!s2_data_decoded.uncorrectable, Seq("no_data_error"))`

`CoverBoolean(s2_data_decoded.correctable, Seq("data_correctable_error")),`

`CoverBoolean(s2_data_decoded.uncorrectable,
Seq("data_uncorrectable_error")))`

Variable `request_source` is initialized and assigned to `Seq(`

`CoverBoolean(!s2_slaveValid, Seq("from_CPU")),`

`CoverBoolean(s2_slaveValid, Seq("from_TL"))`

Variable `tag_error` is initialized and assigned to `Seq(`

`CoverBoolean(!s2_tag_disparity, Seq("no_tag_error")),`

CoverBoolean(s2_tag_disparity, Seq("tag_error"))

Variable mem_mode is initialized and assigned to Seq(

CoverBoolean(s2_scratchpad_hit, Seq("ITIM_mode")),

CoverBoolean(!s2_scratchpad_hit, Seq("cache_mode"))

)

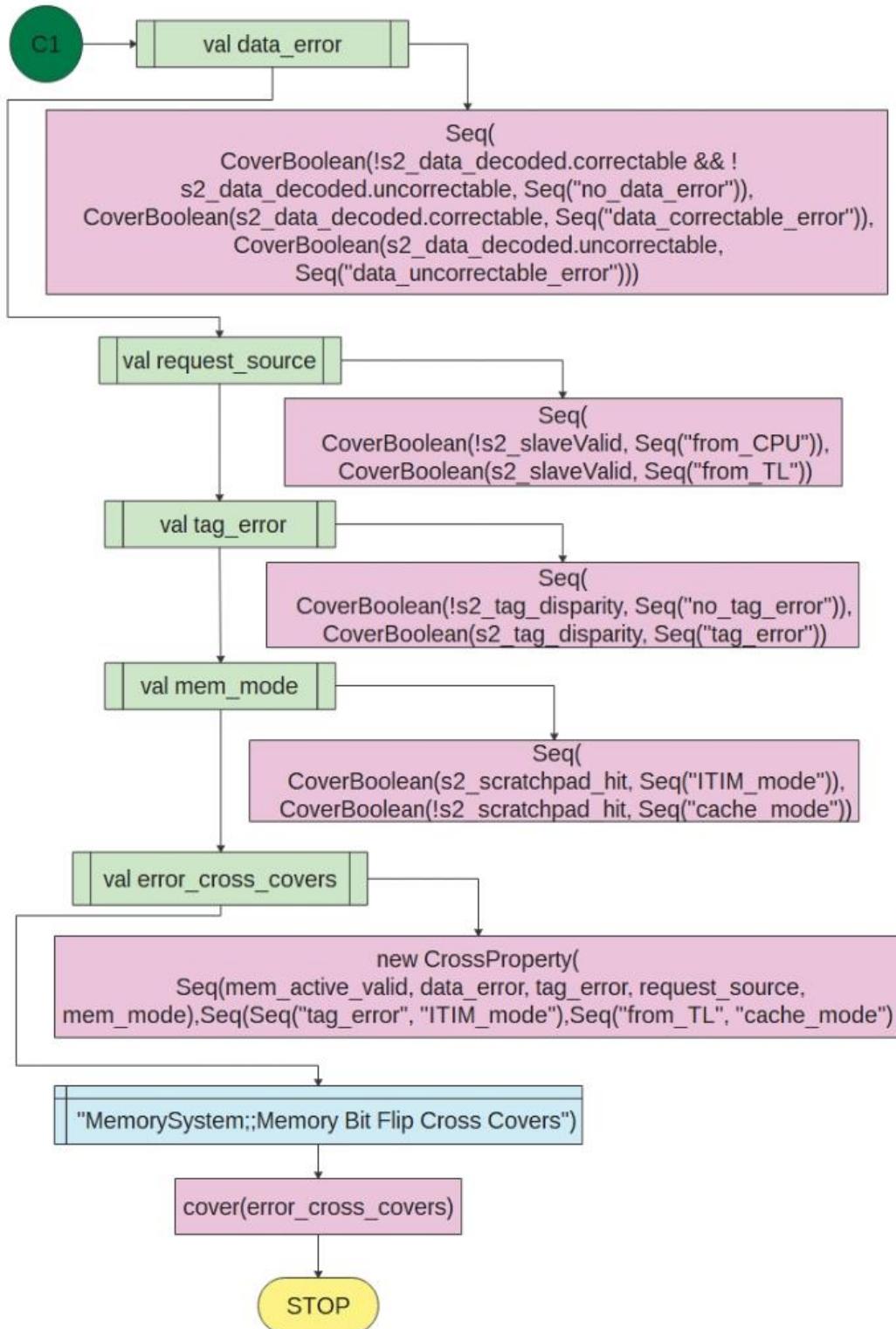
Variable error_cross_covers is initialized and assigned to new CrossProperty(

Seq(mem_active_valid, data_error, tag_error, request_source, mem_mode),

Seq(

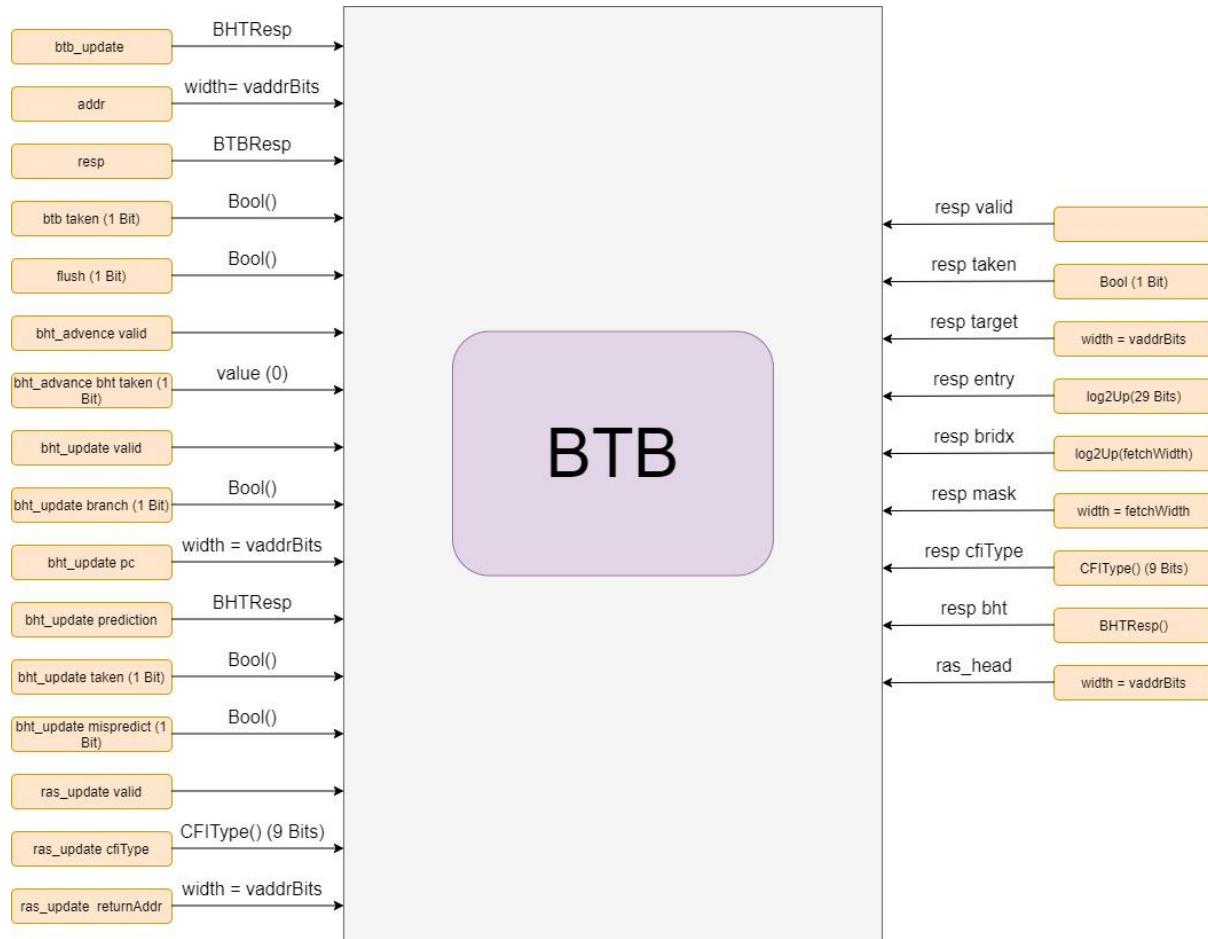
tag_error cannot occur in ITIM mode

Seq("tag_error", "ITIM_mode"),



Branch Target Buffer (BTB)

Block Diagram:



DEEP DIVE INTO CODE

Explanation (By Means of Flowcharts) :

```
case class BHTParams(
    nEntries: Int = 512,
    counterLength: Int = 1,
    historyLength: Int = 8,
    historyBits: Int = 3)

case class BTBParams(
    nEntries: Int = 28,
    nMatchBits: Int = 14,
    nPages: Int = 6,
    nRAS: Int = 6,
    bhtParams: Option[BHTParams] = Some(BHTParams()),
    updatesOutOfOrder: Boolean = false)
```

— explanation —

case class of BHTParams is initialized having,

nEntries of value 512

counterLength of value 1

historyLength of value 8

historyBits of value 3

another case class of BTBParams is defined having

nEntries of value 28

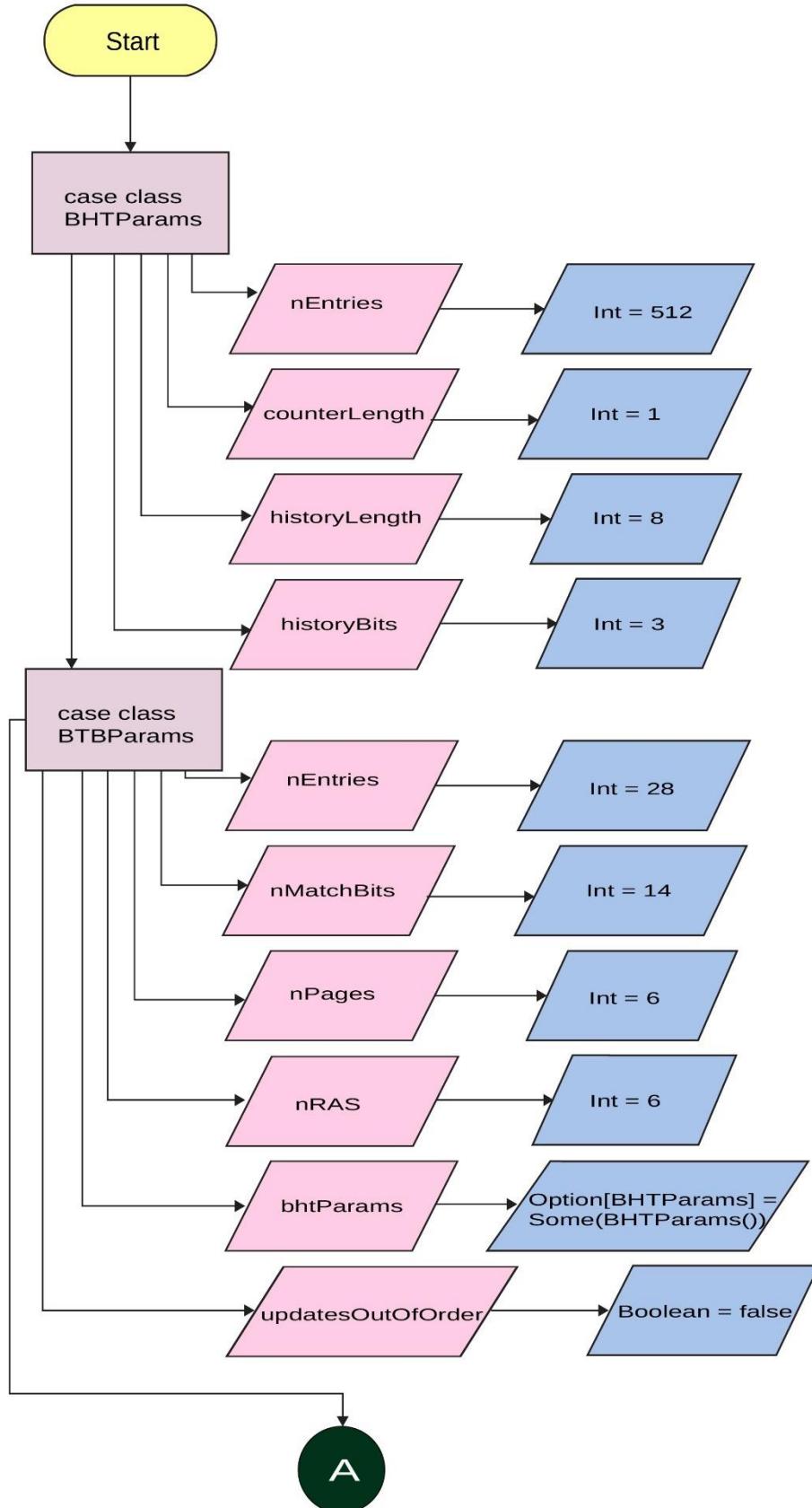
nMatchBits of value 14

nPages of value 6

nRAS of value 6

bhtParams of value BHTParams objecy called inside Some.

updatesOutOfOrder having false.



```

trait HasBtbParameters extends HasCoreParameters { this: InstanceId =>
  val btbParams = tileParams.btb.getOrElse(BTBParams(nEntries = 0))
  val matchBits = btbParams.nMatchBits max log2Ceil(p(CacheBlockBytes) * tileParams.icache.get.nSets)
  val entries = btbParams.nEntries
  val updatesOutOfOrder = btbParams.updatesOutOfOrder
  val nPages = (btbParams.nPages + 1) / 2 * 2 // control logic assumes 2 divides pages
}

abstract class BtbModule(implicit val p: Parameters) extends Module with HasBtbParameters {
  Annotated.params(this, btbParams)
}

abstract class BtbBundle(implicit val p: Parameters) extends Bundle with HasBtbParameters

```

→ explanation

`btbParams` initialized as an instance of `BTBParams` class either from `tileParams.btb` in file `BaseTile.scala` or from this file's case class refactoring the value of `nEntries` to 0

matchBits has btbParams.nMatchBit max with log2Ceil of p(CacheBlockBytes) multiplied with, tileParams.icache.get.nSets

CacheBlockBytes are imported from rocketchip.subsystem

trait tileParams is in BaseTile.scala defines icache se an instance of case class ICacheparams located in ICache.scala containing the value of nSets as Int 64,

```
case class ICacheParams(
```

nSet: Int = 64,

entries has btbParams.nEntries

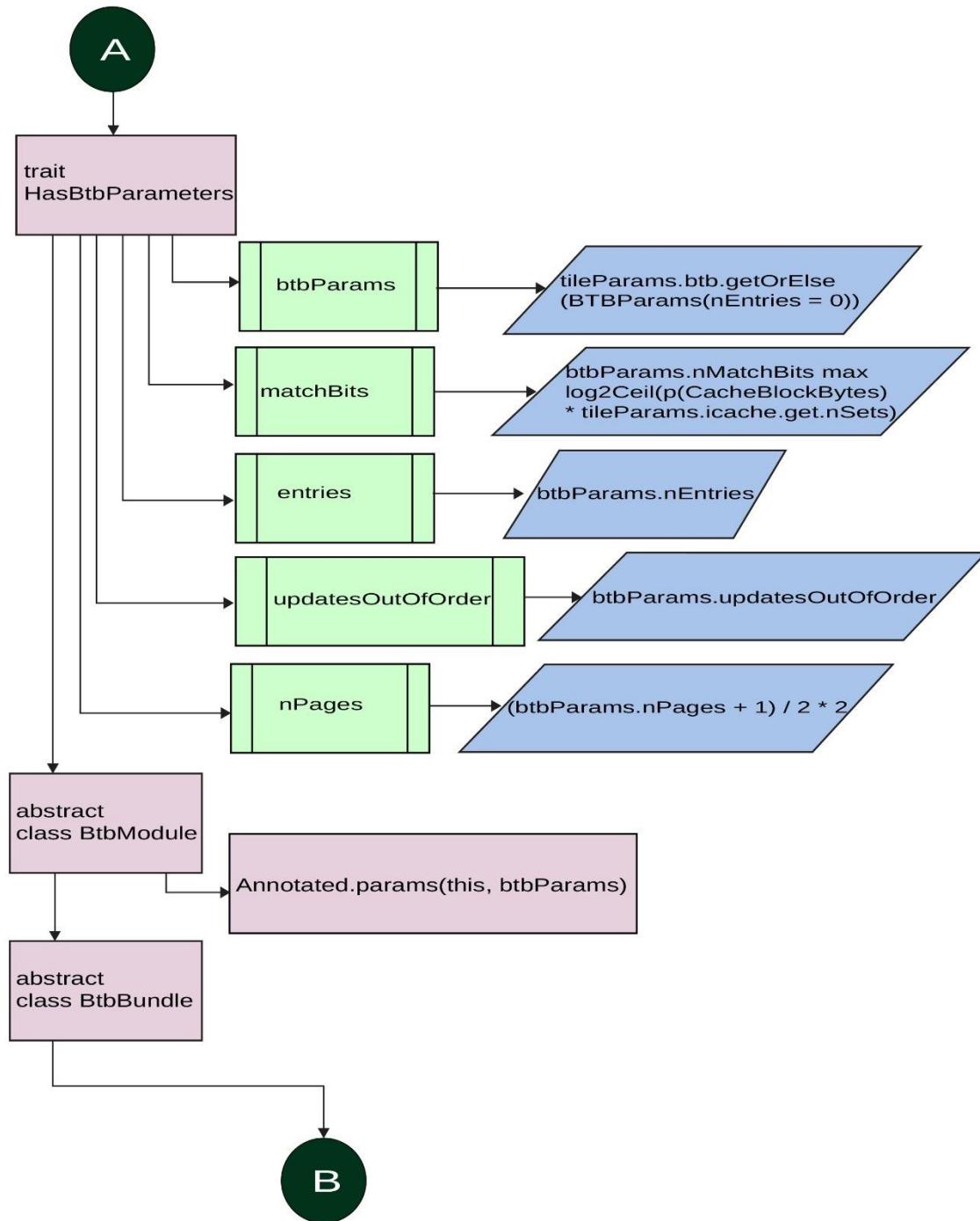
`updatesOutOfOrder` has `btbParams.updatesOutOfOrder`.

nPages has btbParams.nPages + 1, divided bt , then multiplied by 2

abstract class of BtbModule is initialized extended by Module, and HasBtbParameters trait, having

Annotated.params with this, and btbParams

Abstract class of BtbBundle is initialized extended by Bundle and HasBtbParameters



```

class RAS(nras: Int) {
    def push(addr: UInt): Unit = {
        when (count < nras) { count := count + 1 }
        val nextPos = Mux(Bool(isPow2(nras)) || pos < nras-1, pos+1, UInt(0))
        stack(nextPos) := addr
        pos := nextPos
    }
    def peek: UInt = stack(pos)
    def pop(): Unit = when (!isEmpty) {
        count := count - 1
        pos := Mux(Bool(isPow2(nras)) || pos > 0, pos-1, UInt(nras-1))
    }
}

```

 explanation

Class RAS is defined with parameter nras as Int,

Push method is defined with addr as UInt, returning a UInt,

When count is less than nras, count is wired to count + 1,

nextPos variable is defined with Mux having select as, Bool of isPow2 of nras, OR'ed with pos less than nras-1, with pos+1 and UInt of 0

stack method is called with, nestPos , is wired with addr

pos is wired with nestPos

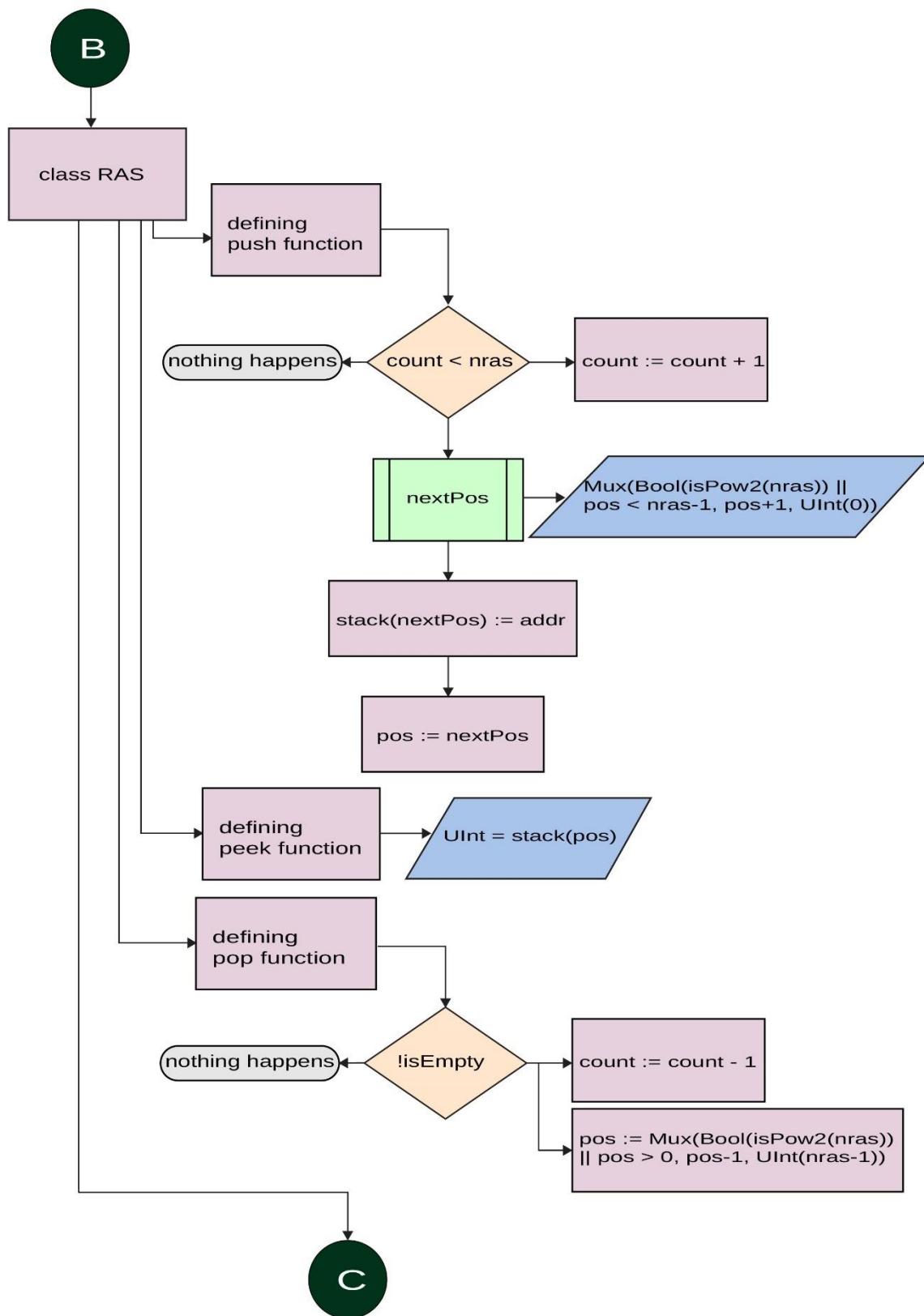
peek method is created returning UInt, having pos called within stack.

Pop method is created returning UInt,

When isEmpty is false, then

Count is wired to count -1

Pos is wired to a Mux, having sel as Bool of isPow2 of nras OR'ed with pos greater than 0, with pos-1, and UInt of nras-1



```

def clear(): Unit = count := UInt(0)
def isEmpty: Bool = count === UInt(0)
private val count = Reg(UInt(width = log2Up(nrás+1)))
private val pos = Reg(UInt(width = log2Up(nrás)))
private val stack = Reg(Vec(nrás, UInt()))
}

```

explanation

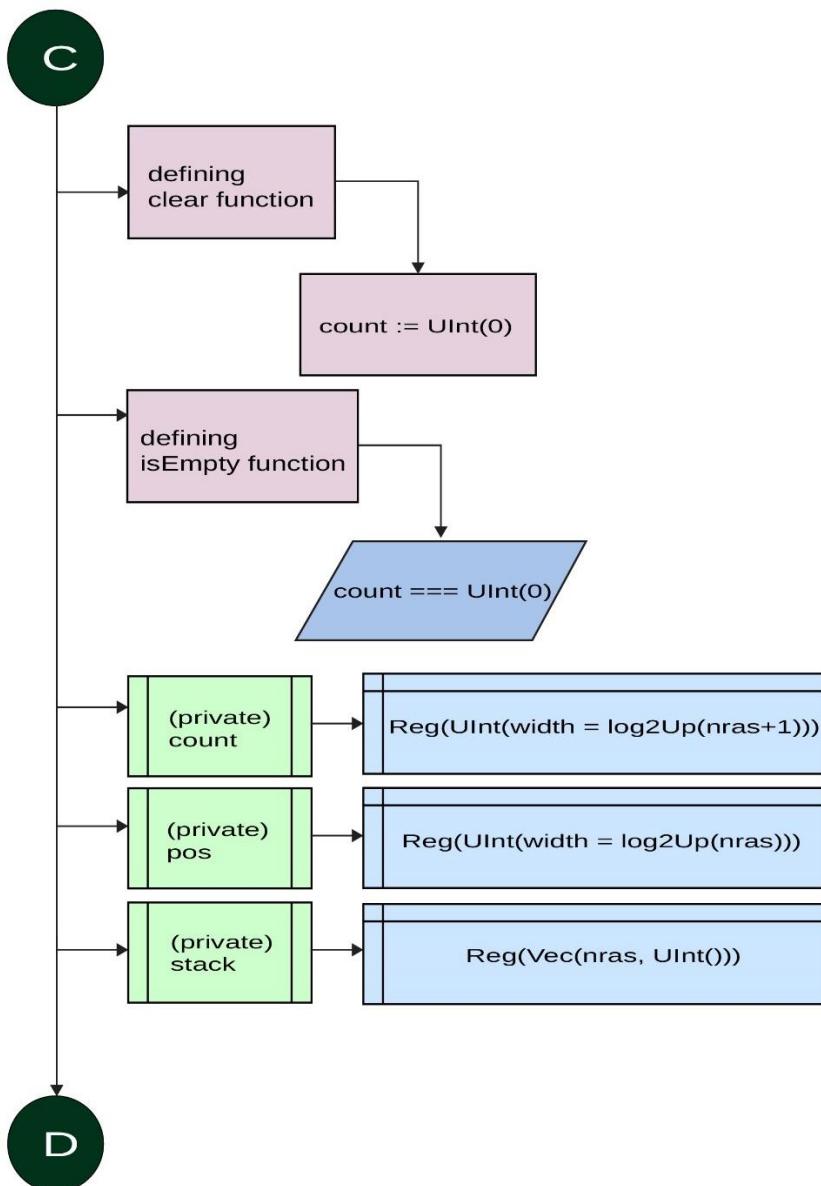
Clear method is created returning UInt, having count wired to UInt of 0

isEmpty method is created returning Bool, having count === UInt of 0

private variable count is defined having, Reg with UInt of width, log2Up of nrás+1

private variable pos is defined having, Reg with UInt of width, log2Up of nrás

private variable stack is defined having, Reg with Vec of length nrás, having UInt() on every index.



```
class BHTResp(implicit p: Parameters) extends BtbBundle()(p) {
    val history = UInt(width =
btbParams.bhtParams.map(_.historyLength).getOrElse(1))
    val value = UInt(width =
btbParams.bhtParams.map(_.counterLength).getOrElse(1))
    def taken = value(0)
    def strongly_taken = value === 1
}
```

— explanation —

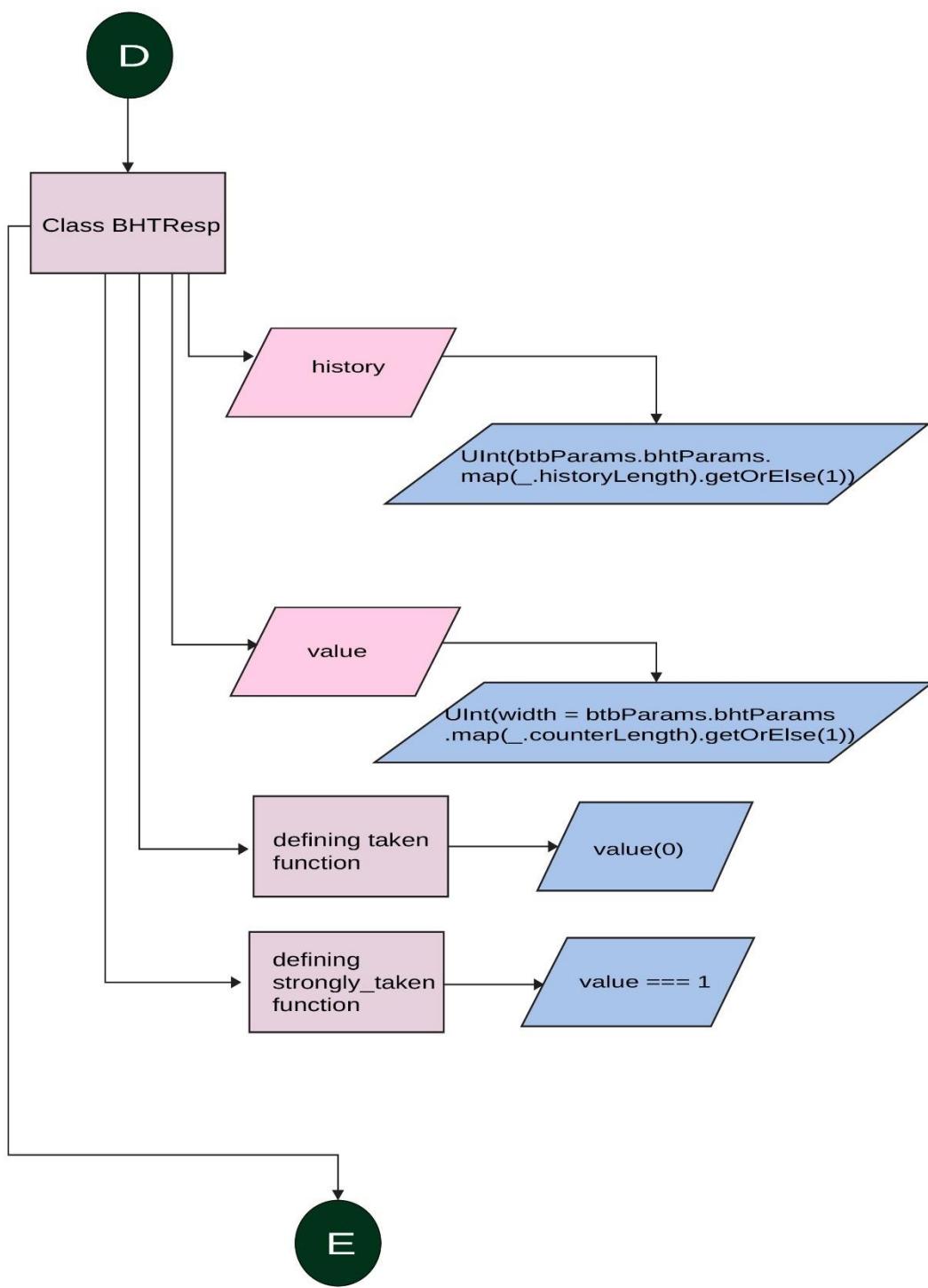
class BHTResp is initialized with parameters p, extended by BtbBundle, having,

history variable with UInt of width, btbParams.bhtParams mapped with
.historyLength with getOrElse of 1

value variable with UInt of width, btbParams.bhtParams mapped with
.counterLength with getOrElse of 1

taken method has value(1)

strongly_taken method has value === 1



```

class BHT(params: BHTParams) (implicit val p: Parameters) extends
HasCoreParameters {
  def index(addr: UInt, history: UInt) = {
    def hashHistory(hist: UInt) = if (params.historyLength == params.historyBits) hist else {
      val k = math.sqrt(3)/2
      val i = BigDecimal(k * math.pow(2, params.historyLength)).toBigInt
      (i.U * hist)(params.historyLength-1, params.historyLength-
params.historyBits)
    }
    def hashAddr(addr: UInt) = {
      val hi = addr >> log2Ceil(fetchBytes)
      hi(log2Ceil(params.nEntries)-1, 0) ^ (hi >>
log2Ceil(params.nEntries))(1, 0)
    }
    hashAddr(addr) ^ (hashHistory(history) << (log2Up(params.nEntries) -
params.historyBits))
  }
}

```

— explanation —

BHT class is defined with params as Parameters, extended by HasCoreParameters, having

Index method, with parameters, addr as UInt, and history as UInt, having

hashHistory method, with parameter, hist as UInt, in which

if params.historyLength === params.historyBits, then hist will be returned.

Else, k variable has square root of 3, divided by 2

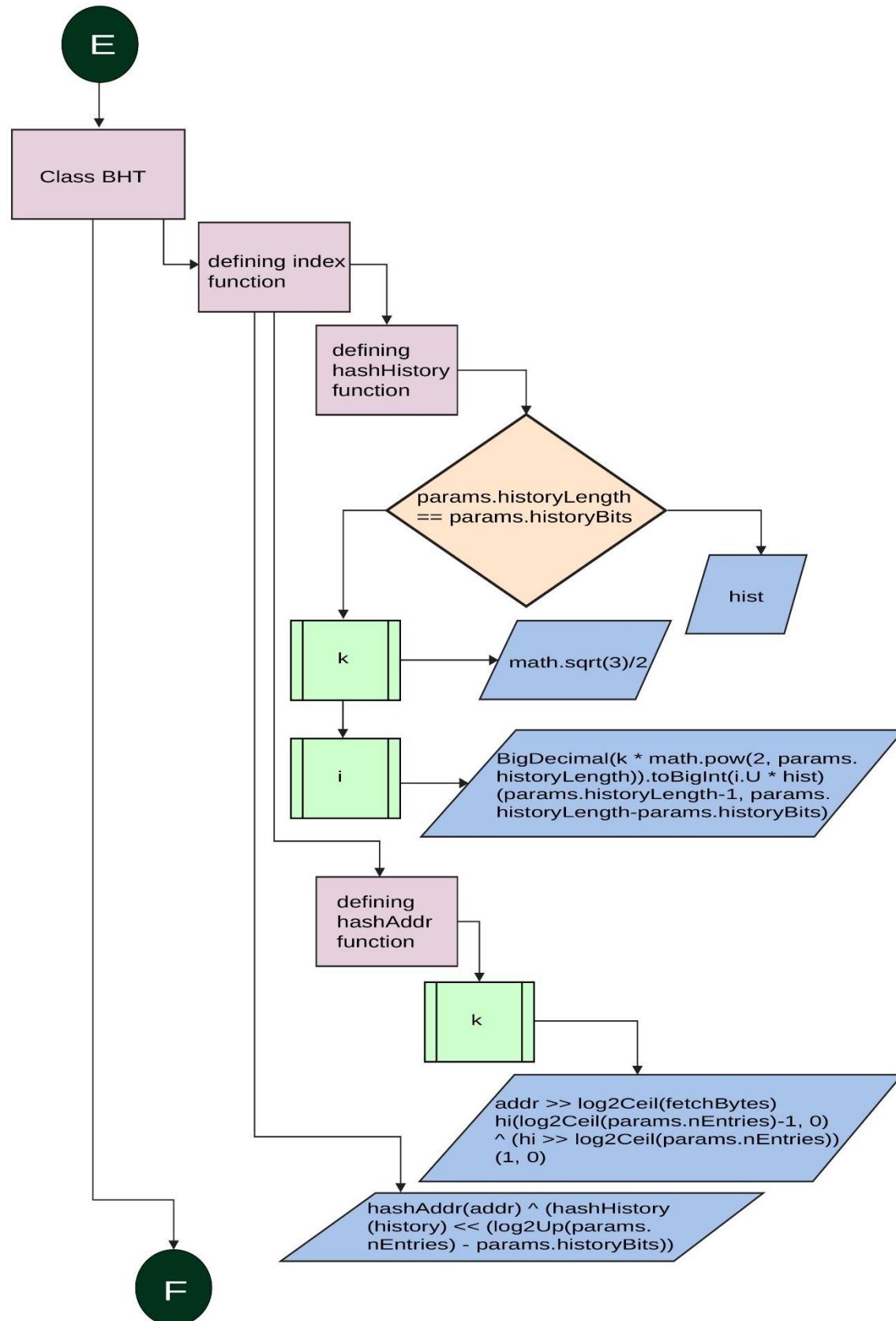
i variable has BigDecimal of k, multiplied by 2 of power params.historyLength, turned to BigInt of i.U multiplied by hist, and params.historyLength-1, and params.historyLength-params.historyBits.

hashAddr method has parameters, addr as UInt, having

hi variable, with addr right shifted by log2Ceil of fetchBytes.

hi with log2Ceil of params.nEntries -1, and 0, with power, hi right shifted by log2Ceil of params.nEntries, having 1, 0

hashAddr with addr, with power, hashHistory of history, left shifted by log2Up of params.nEntries – params.historyBits is returned



```

def get(addr: UInt): BHTResp = {
    val res = Wire(new BHTResp)
    res.value := table(index(addr, history))
    res.history := history
    res
}
def updateTable(addr: UInt, d: BHTResp, taken: Bool): Unit = {
    table(index(addr, d.history)) := (params.counterLength match {
        case 1 => taken
        case 2 => Cat(taken ^ d.value(0), d.value === 1 || d.value(1) && taken)
    })
}
def resetHistory(d: BHTResp): Unit = {
    history := d.history
}

```

 explanation

get method is defined with addr as UInt, returning a BHTResp object, having res variable has BHTResp object hardware wired.

res.value is wired to table of index called with addr, and history.

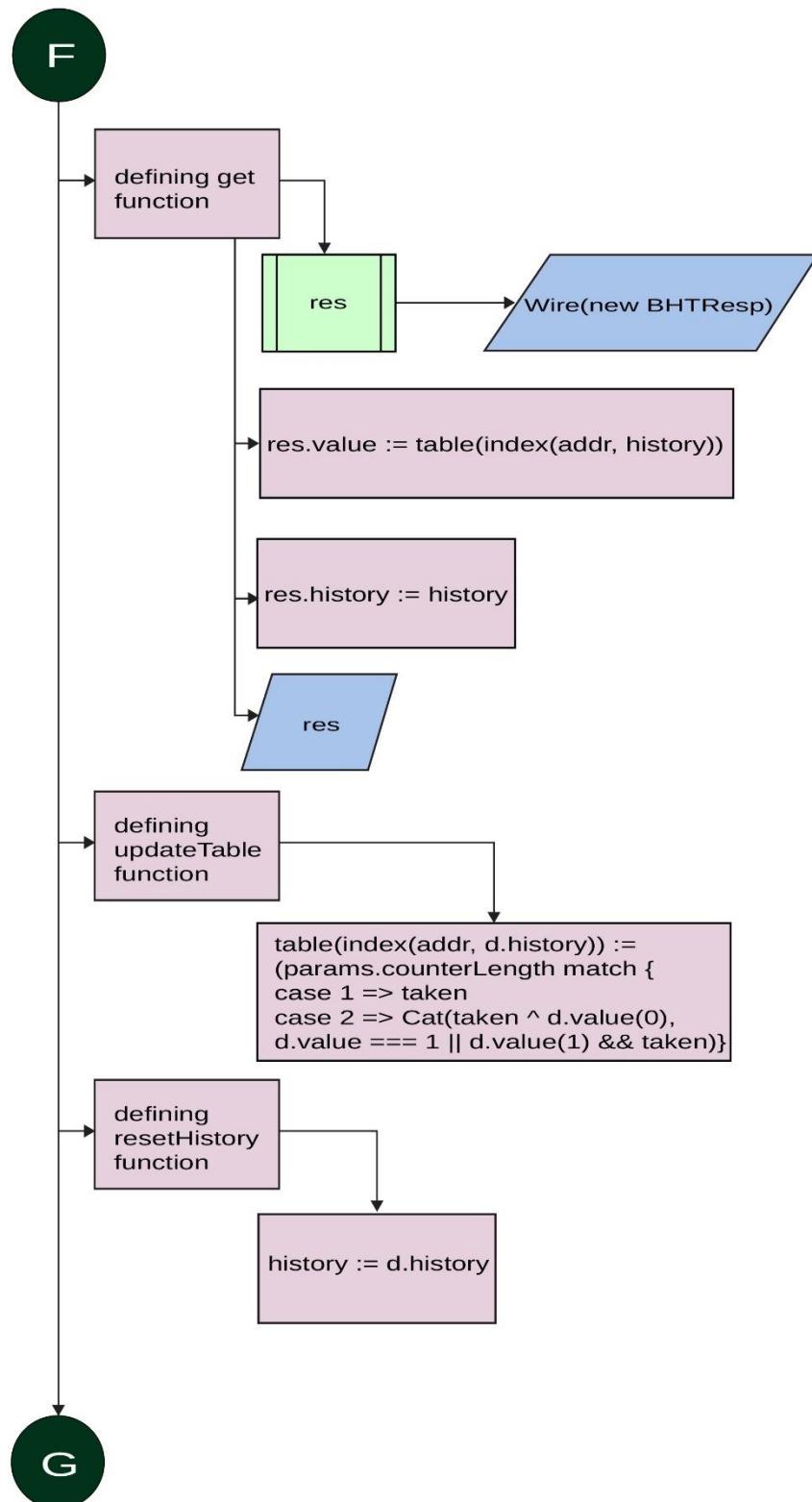
res.history si wired to history.

res is returned finally,

updateTable method is defined with addr as UInt, d as BHTResp, taken as Bool, returning UInt,

table of index addr and d.history is wired to params.counterLength with match method, os case 1 as taken, and case 2 as Concatenation of taken of power d.value(0), with d.value === 1, OR'ed with d.value(1) AND'ed with taken.

resetHistory method is defined with d as BHTResp, returning UInt, having history wired to d.history .



```

def updateHistory(addr: UInt, d: BHTResp, taken: Bool): Unit = {
    history := Cat(taken, d.history >> 1)
}
def advanceHistory(taken: Bool): Unit = {
    history := Cat(taken, history >> 1)
}

private val table = Mem(params.nEntries, UInt(width =
params.counterLength))
val history = Reg(UInt(width = params.historyLength))
}

object CFIType {
    def SZ = 2
    def apply() = UInt(width = SZ)
    def branch = 0.U
    def jump = 1.U
    def call = 2.U
    def ret = 3.U
}

```

— explanation —

updateHistory method has addr as UInt, d as BHTResp object, and taken as Boolean, returning UInt, having

history wired to Concatenation of taken, and d.history right shifted by 1

advanceHistory method has taken as Boolean, returning UInt, having

history wired as concatenation of taken, and history right shifted by 1

private variable table is defined with Mem.

History variable has Reg with UInt of width params.historyLength.

Object of CFIType is defined, having

SZ method as 2,

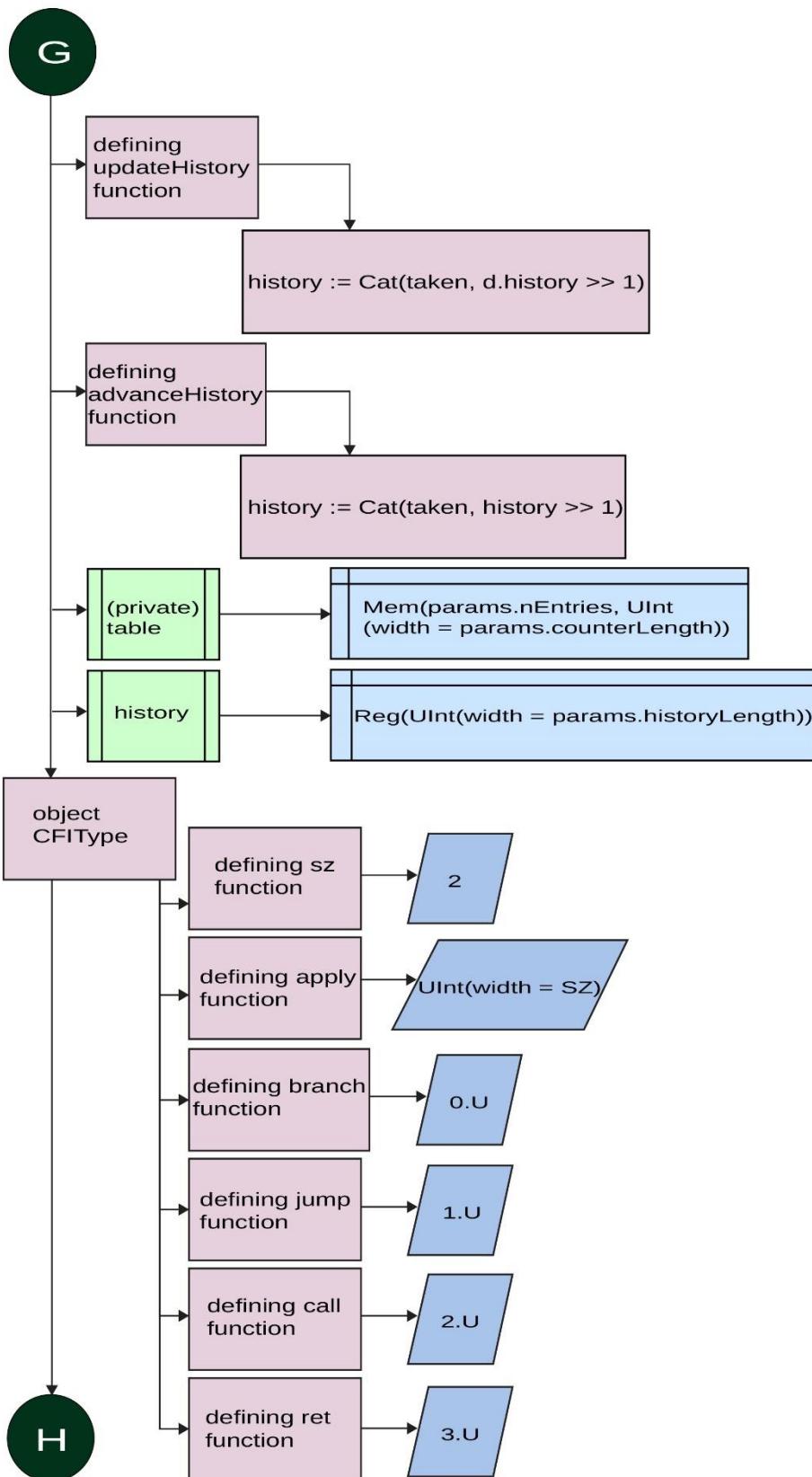
apply method as UInt of width SZ,

branch method as 0.U

jump method as 1.U

call method as 2.U

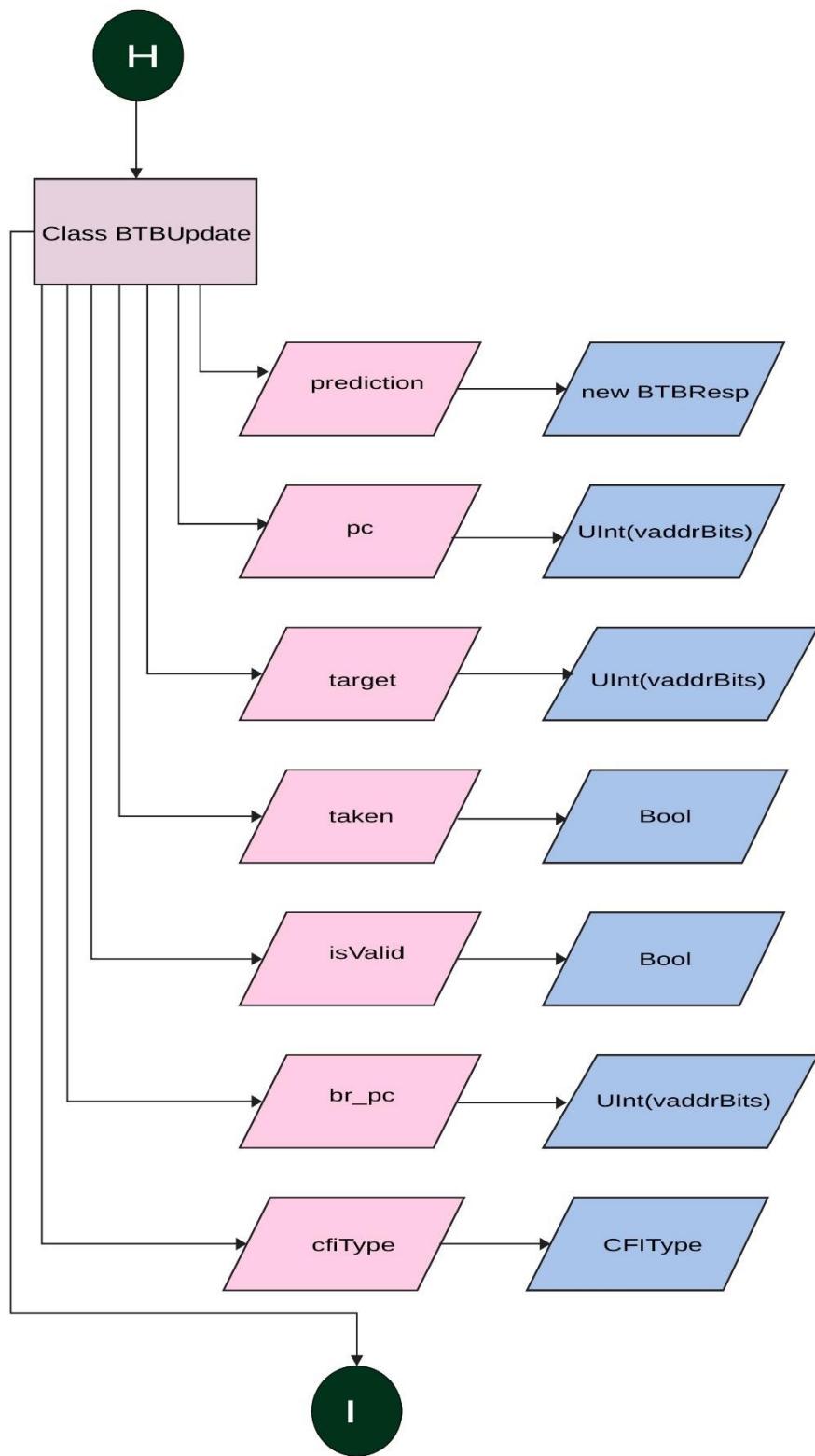
ret method as 3.U



```
class BTBUpdate(implicit p: Parameters) extends BtbBundle()(p) {
    val prediction = new BTBResp
    val pc = UInt(width = vaddrBits)
    val target = UInt(width = vaddrBits)
    val taken = Bool()
    val isValid = Bool()
    val br_pc = UInt(width = vaddrBits)
    val cfiType = CFIType()
}
```

explanation

BTBUpdate class is defined with parameter, p, extended by BtbBundle, having prediction has BTBResp object,
pc has UInt of width vaddrBits
target has UInt of width vaddrBits
taken has Boolean
isValid has Boolean
br_pc has UInt of width vaddrBits
cfiType has CFIType object.

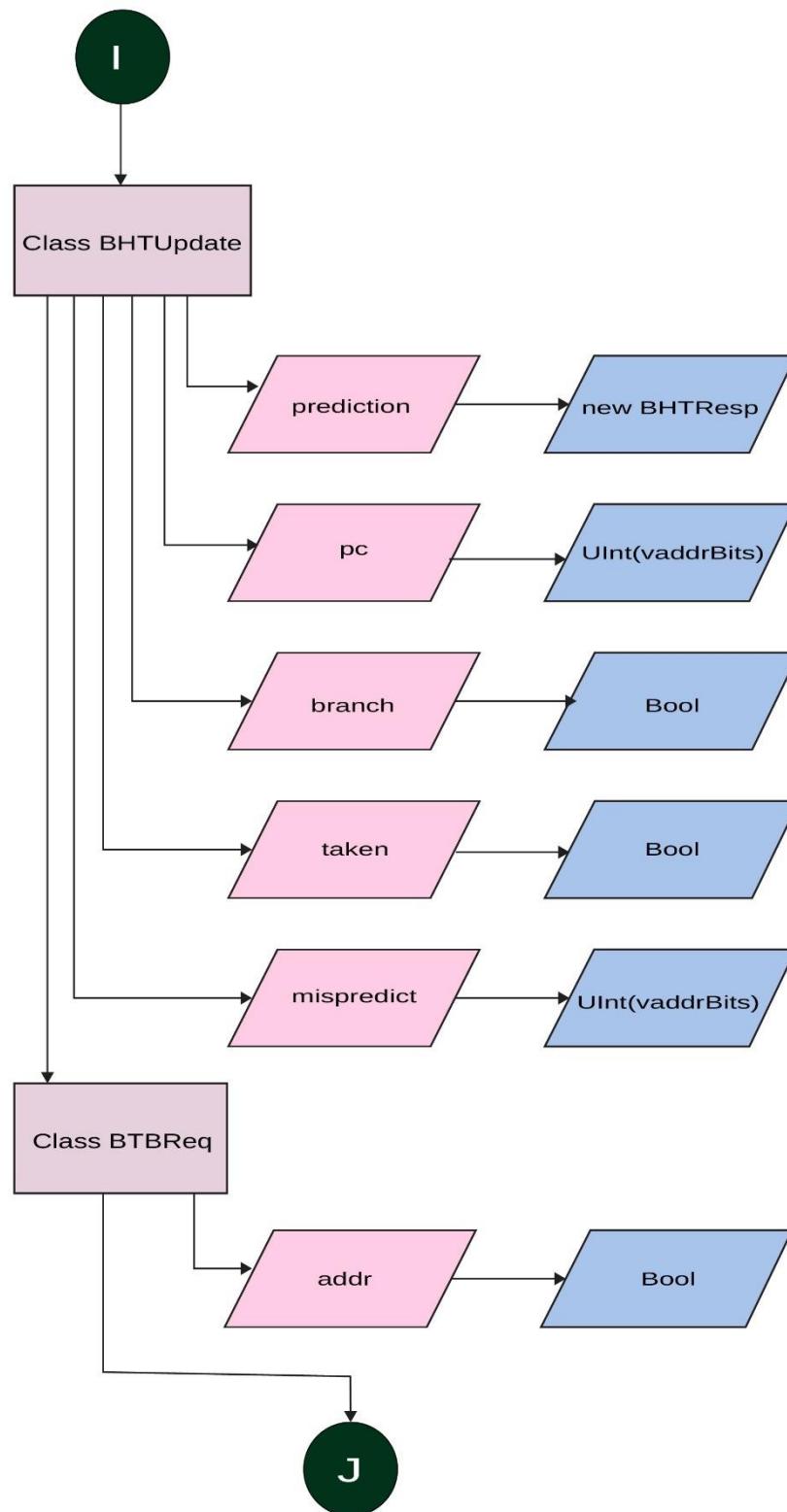


```
class BHTUpdate(implicit p: Parameters) extends BtbBundle()(p) {
    val prediction = new BHTResp
    val pc = UInt(width = vaddrBits)
    val branch = Bool()
    val taken = Bool()
    val mispredict = Bool()
}
class BTBReq(implicit p: Parameters) extends BtbBundle()(p) {
    val addr = UInt(width = vaddrBits)
}
```

explanation

BHTUpdate class is created with parameter, p, extended by BtbBundle, having prediction as BHTResp object,
pc as UInt of width vaddrBits,
branch as Boolean,
taken as Boolean,
mispredict as Boolean,

BTBReq class is created with parameter p, extended by BtbBundle, having
addr as UInt of width vaddrBits.

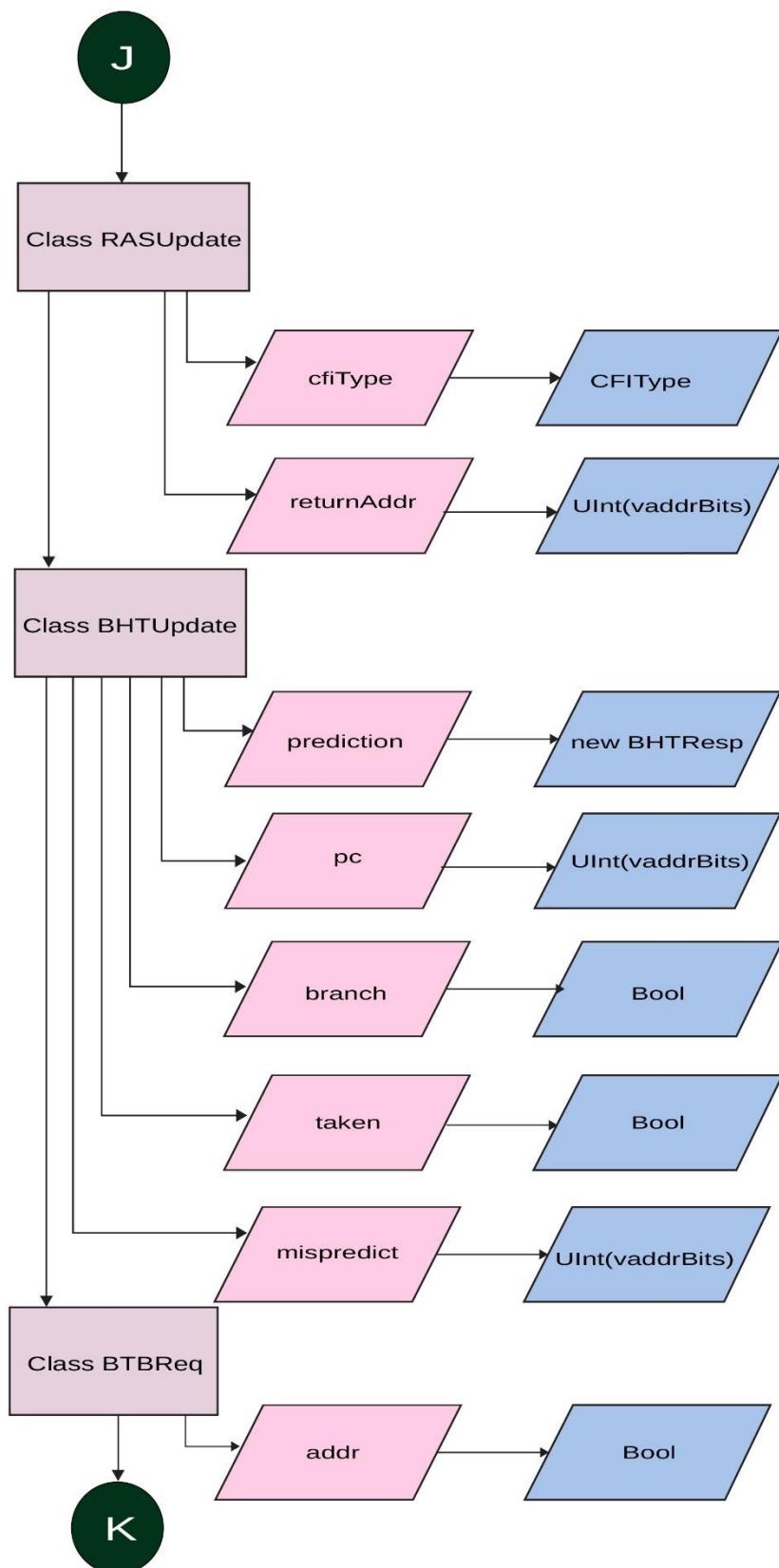


```
class RASUpdate(implicit p: Parameters) extends BtbBundle()(p) {  
    val cfiType = CFIType()  
    val returnAddr = UInt(width = vaddrBits)  
}
```

--repeated cause--

—— explanation ——

class RASUpdate is created with parameters, p, extended by BtbBundle, having, cfiType, as CFIType object,
returnAddr has UInt of width vaddrBits.



```

class BTB(implicit p: Parameters) extends BtbModule {
  val io = new Bundle {
    val req = Valid(new BTBReq).flip
    val resp = Valid(new BTBResp)
    val btb_update = Valid(new BTBUpdate).flip
    val bht_update = Valid(new BHTUpdate).flip
    val bht_advance = Valid(new BTBResp).flip
    val ras_update = Valid(new RASUpdate).flip
    val ras_head = Valid(UInt(width = vaddrBits))
    val flush = Bool().asInput
  }
}

```

 explanation

Class BTB extends BtbModule,

io is initialized as a new bundle

req is initialized as a valid flipped instance of class BTBReq

resp is initialized as a valid instance of class BTBResp

btb_update is initialized as a valid flipped instance of class BTBUpdate

bht_update is initialized as a valid flipped instance of BHTUpdate

bht_advance is initialized as a valid flipped instance of BTBResp

ras_update is initialized as a valid flipped instance of class RASUpdate

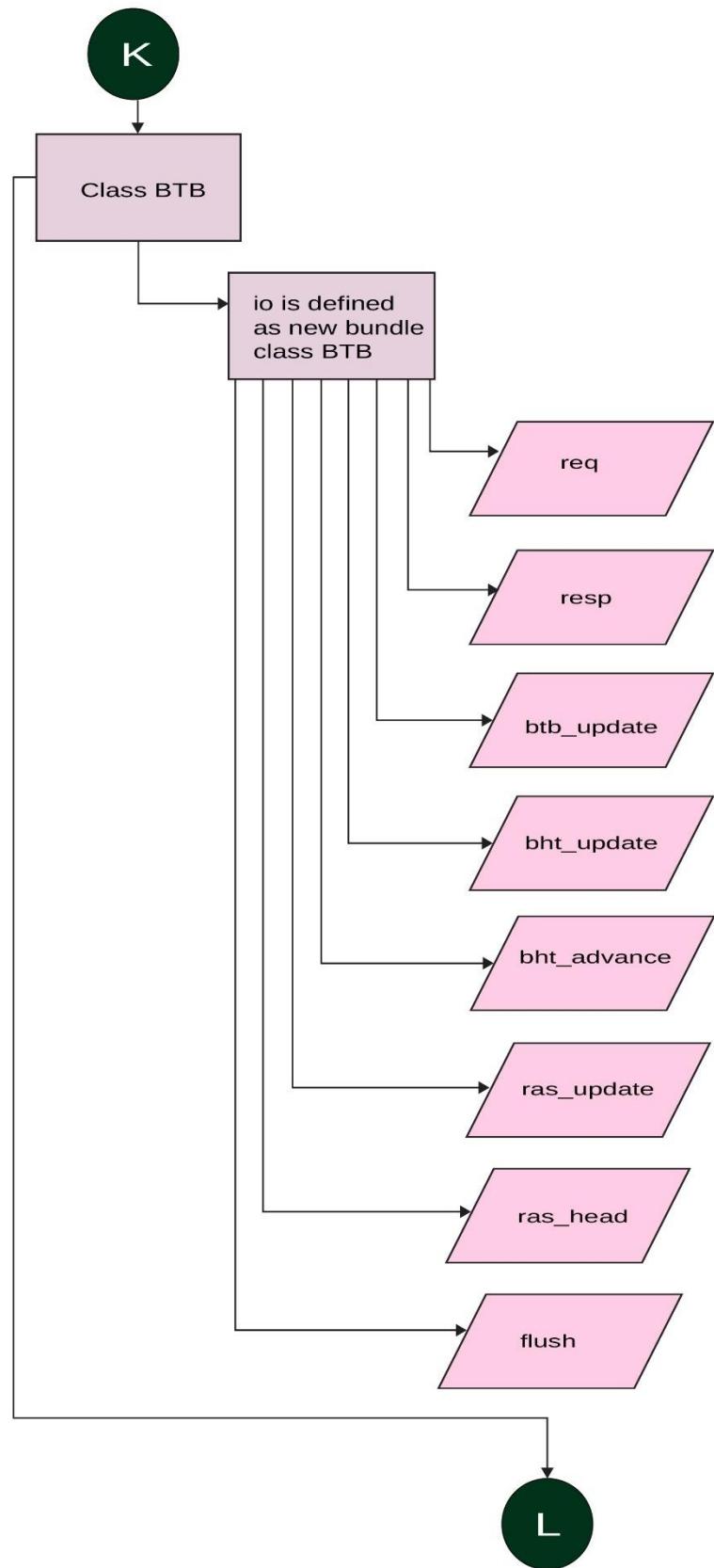
ras_head is initialized as valid UInt width that is equivalent to vaddrBits, vaddrBits is a function defined in BaseTile.scala whose return type is Int under the trait HasTileParameters which is extended by the trait HasCoreParamaters in Core.scala and HasCoreParameters is extended by HasBtbParameters extended by BtbModule extended by class BTB

```

trait HasTileParameters extends HasNonDiplomaticTileParameters {
  protected def tlBundleParams =
    p(TileVisibilityNodeKey).edges.out.head.bundle
  lazy val paddrBits: Int = {
    val bits = tlBundleParams.addressBits
    require(bits <= maxPAddrBits, s"Requested $bits paddr bits, but since
    xLen is $xLen only $maxPAddrBits will fit")
    bits
  }
  def vaddrBits: Int =
    if (usingVM) {
      val v = maxSVAaddrBits
      require(v == xLen || xLen > v && v > paddrBits)
      v
    } else {
      (paddrBits + 1) min xLen
    }
}

```

flush is initialized as Boolean input



```

val idxs = Reg(Vec(entries, UInt(width=matchBits -
log2Up(coreInstBytes))))
val idxPages = Reg(Vec(entries, UInt(width=log2Up(nPages) )))
val tgts = Reg(Vec(entries, UInt(width=matchBits -
log2Up(coreInstBytes))))
val tgtPages = Reg(Vec(entries, UInt(width=log2Up(nPages) )))
val pages = Reg(Vec(nPages, UInt(width=vaddrBits - matchBits)))
val pageValid = Reg(init = UInt(0, nPages))

val isValid = Reg(init = UInt(0, entries))
val cfiType = Reg(Vec(entries, CFIType()))
val brIdx = Reg(Vec(entries, UInt(width=log2Up(fetchWidth))))

private def page(addr: UInt) = addr >> matchBits

```

— explanation —

these values are initialized as registers

nPages is initialized as an integer of value 6 under BTBParams, used as Annotated.params under the abstract class BtbModule extended by the main class BTB.

coreInstBytes is initialized under the trait HasCoreParameters in the file Core.scala,

val coreInstBytes = coreInstBits/8

val coreInstBits = coreParams.instBits

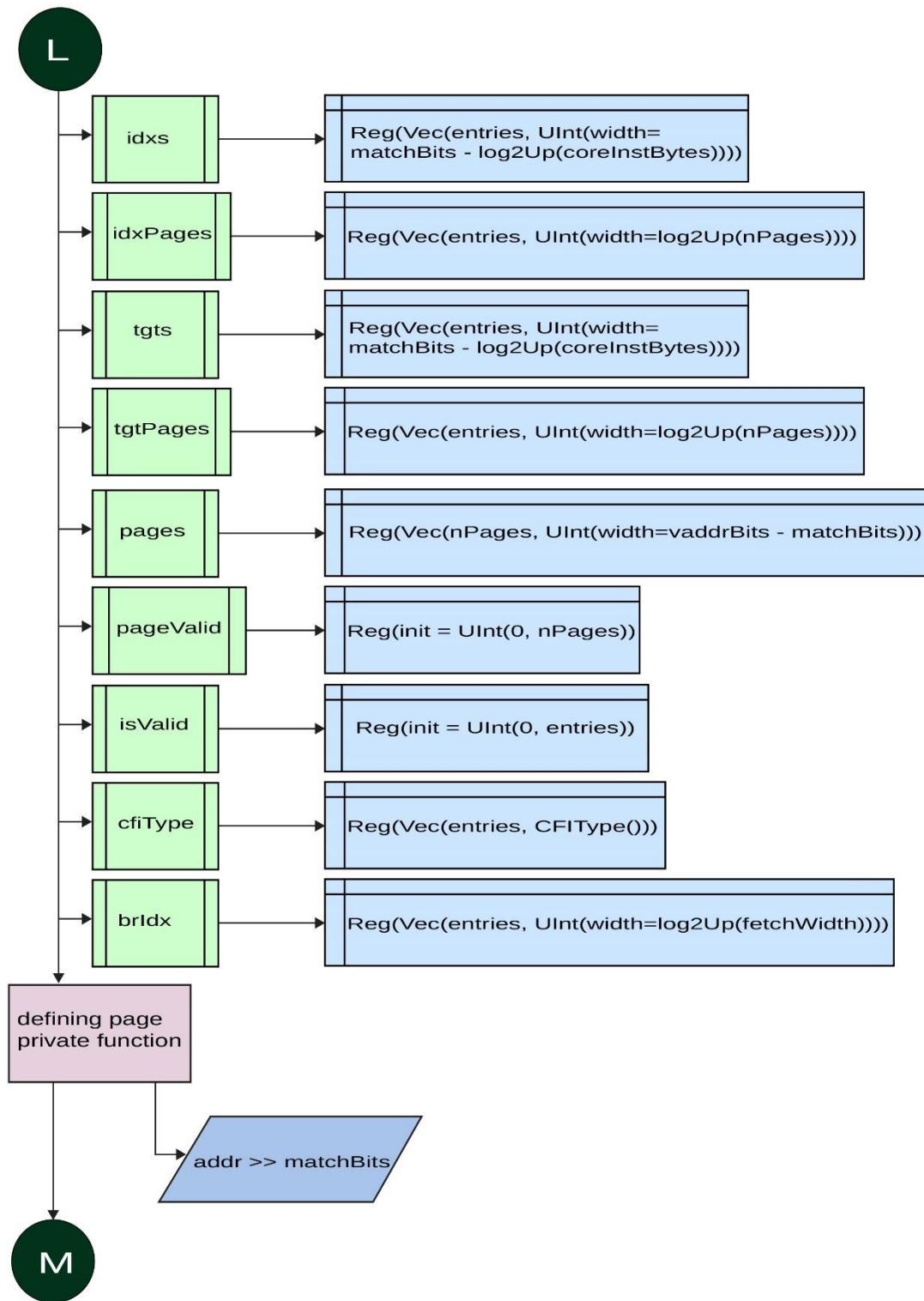
instBits is initialized as Int under the trait coreParams in Core.scala.

matchBits is initialized under HasBtbParameters which extends HasCoreParameters

entries is also initialized under the trait HasBtbParameters

fetchWidth is initialized as type Int in Core.scala Under the trait CoreParams also initialized in the trait HasCoreParameters extended by HasBtbParameters in BTB.scala.

private function page is defined, takes addr as UInt argument and right shift addr to the length of matchBits



```

private def pageMatch(addr: UInt) = {
  val p = page(addr)
  pageValid & pages.map(_ === p).asUInt
}
private def idxMatch(addr: UInt) = {
  val idx = addr(matchBits-1, log2Up(coreInstBytes))
  idxs.map(_ === idx).asUInt & isValid
}

val r_btb_update = Pipe(io.btb_update)
val update_target = io.req.bits.addr

val pageHit = pageMatch(io.req.bits.addr)
val idxHit = idxMatch(io.req.bits.addr)

val updatePageHit = pageMatch(r_btb_update.bits.pc)

```

 explanation

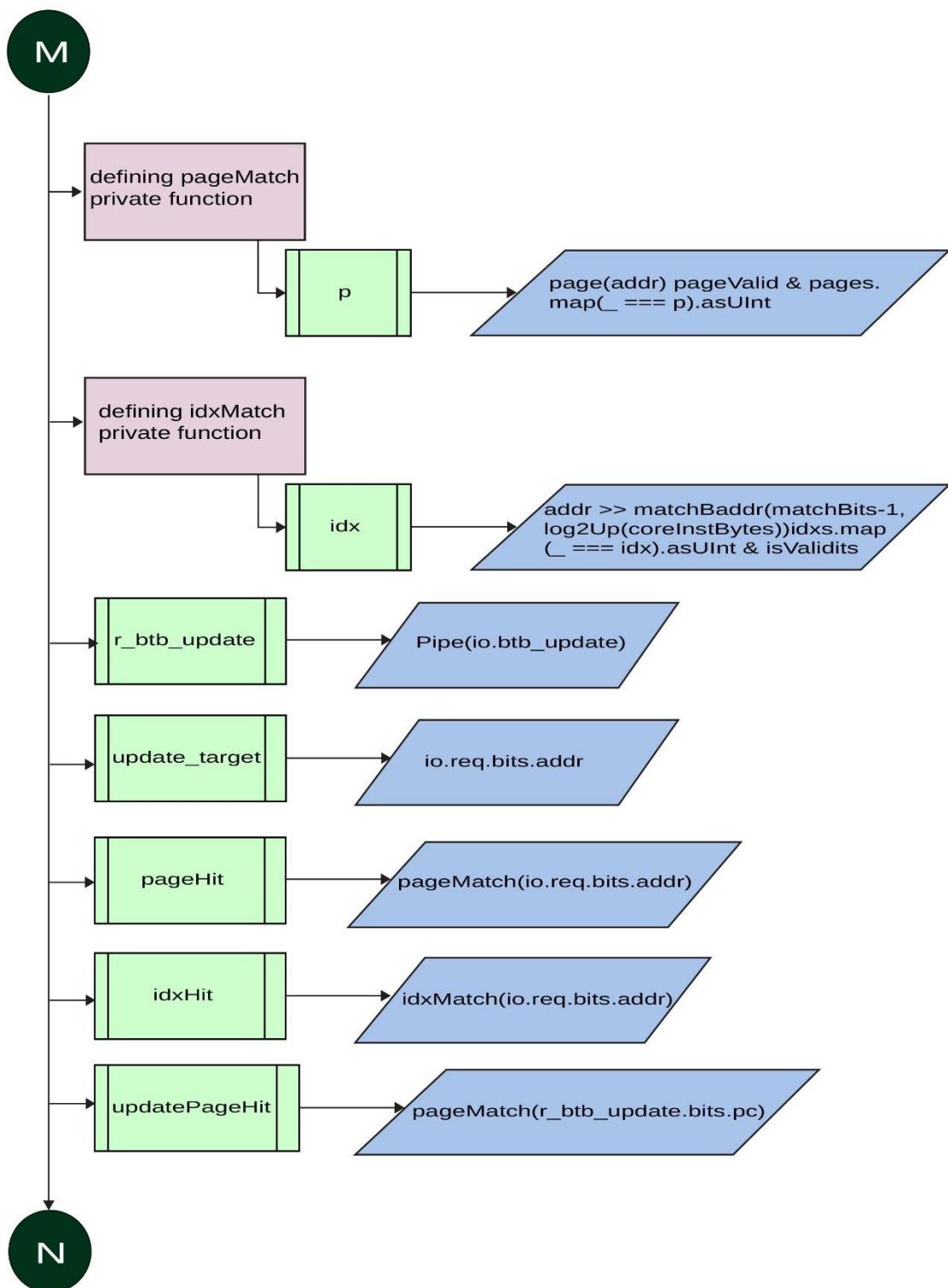
private function pageMatch is defined, takes addr as UInt argument, initializes p as the result of private function page passing addr as its parameter, returns a Boolean value for the AND product of pageValid and pages.map(_ === p).asUInt

private function idxMatch is defined taking addr as UInt argument, initializes idx

returns the boolean value for the AND product of register idxs maps (_ === idx) as unsigned integer and isValid

--- missing part ---

updatePageHit take the return value from private function pageMatch when passed in r_btb_update.bits.pc as parameter



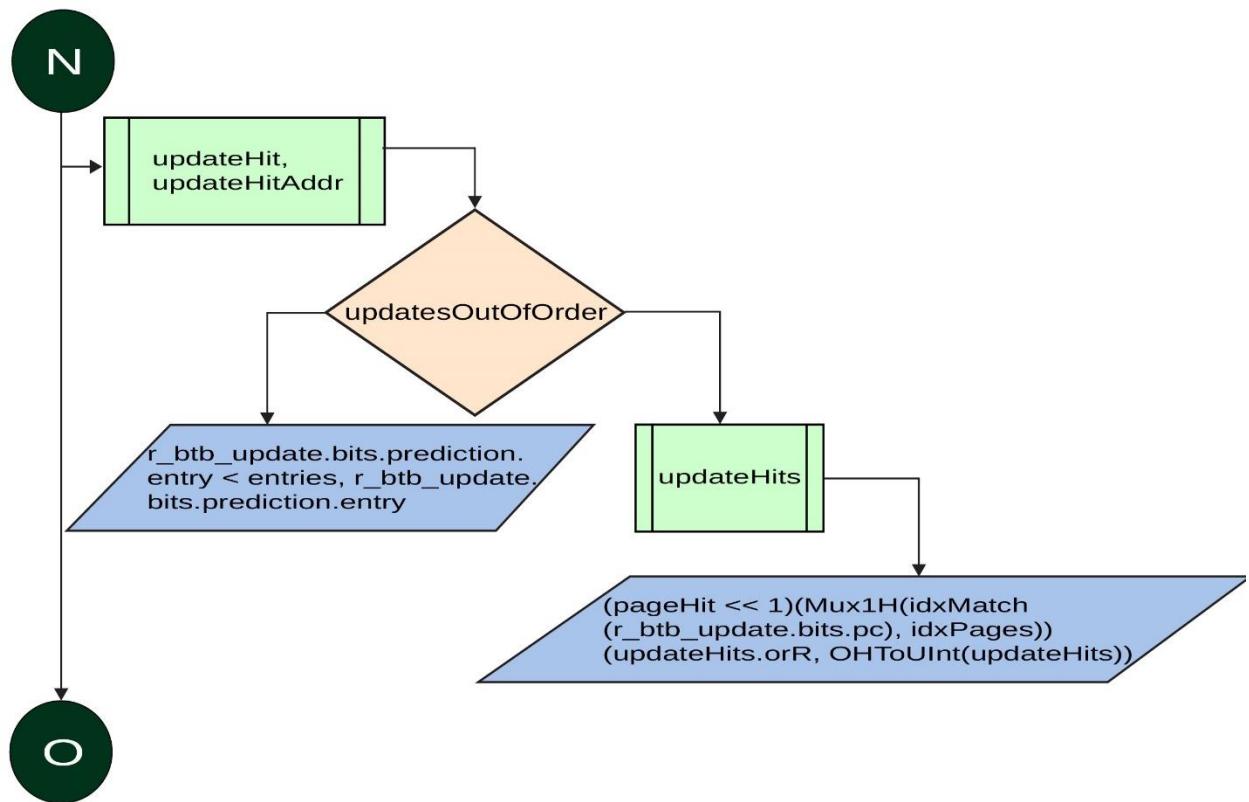
```

val (updateHit, updateHitAddr) =
  if (updatesOutOfOrder) {
    val updateHits = (pageHit << 1) (Mux1H(idxMatch(r_btb_update.bits.pc),
idxPages))
    (updateHits.orR, OHToUInt(updateHits))
  } else (r_btb_update.bits.prediction.entry < entries,
r_btb_update.bits.prediction.entry)

```

 explanation

updateHit and UpdateHitAddr are initialized to the resultant of a conditional statement in which, updateHits is initialized as pageHit leftshifted 1 bit and passed in some parameters to a mux tree if UpdatesOutOfOrder holds a value, else the Boolean value of (r_btb_update.bits.prediction.entry < entries, r_btb_update.bits.prediction.entry)



```

val useUpdatePageHit = updatePageHit.orR
val usePageHit = pageHit.orR
val doIdxPageRepl = !useUpdatePageHit
val nextPageRepl = Reg(UInt(width = log2Ceil(nPages)))
val idxPageRepl = Cat(pageHit(nPages-2,0), pageHit(nPages-1)) |
Mux(usePageHit, UInt(0), UIntToOH(nextPageRepl))
val idxPageUpdateOH = Mux(useUpdatePageHit, updatePageHit, idxPageRepl)
val idxPageUpdate = OHToUInt(idxPageUpdateOH)
val idxPageReplEn = Mux(doIdxPageRepl, idxPageRepl, UInt(0))

val samePage = page(r_btb_update.bits.pc) === page(update_target)
val doTgtPageRepl = !samePage && !usePageHit
val tgtPageRepl = Mux(samePage, idxPageUpdateOH,
Cat(idxPageUpdateOH(nPages-2,0), idxPageUpdateOH(nPages-1)))
val tgtPageUpdate = OHToUInt(pageHit | Mux(usePageHit, UInt(0),
tgtPageRepl))
val tgtPageReplEn = Mux(doTgtPageRepl, tgtPageRepl, UInt(0))

```

— explanation —

useUpdatePageHit is initialized as updatePageHit orR value

usePageHit is initialized as pageHit orR value

doIdxPageRepl initialized as NOT of useUpdatePageHit

nextPageRepl is initialized as a UInt register, given width defined as log2Ceil of nPages value in BTBParams

idxPageRepl initialized as a concatenation where pageHit is initialized above as pageMatch value and UIntToOH encodes results in "b0100".U.

idxPageUpdateOH is initialized as a Mux

idxPageUPdate is the OHToUInt(inverse of UIntToOH) value of idxPageUpdateOH

idxPageReplEn is initialized as a Mux

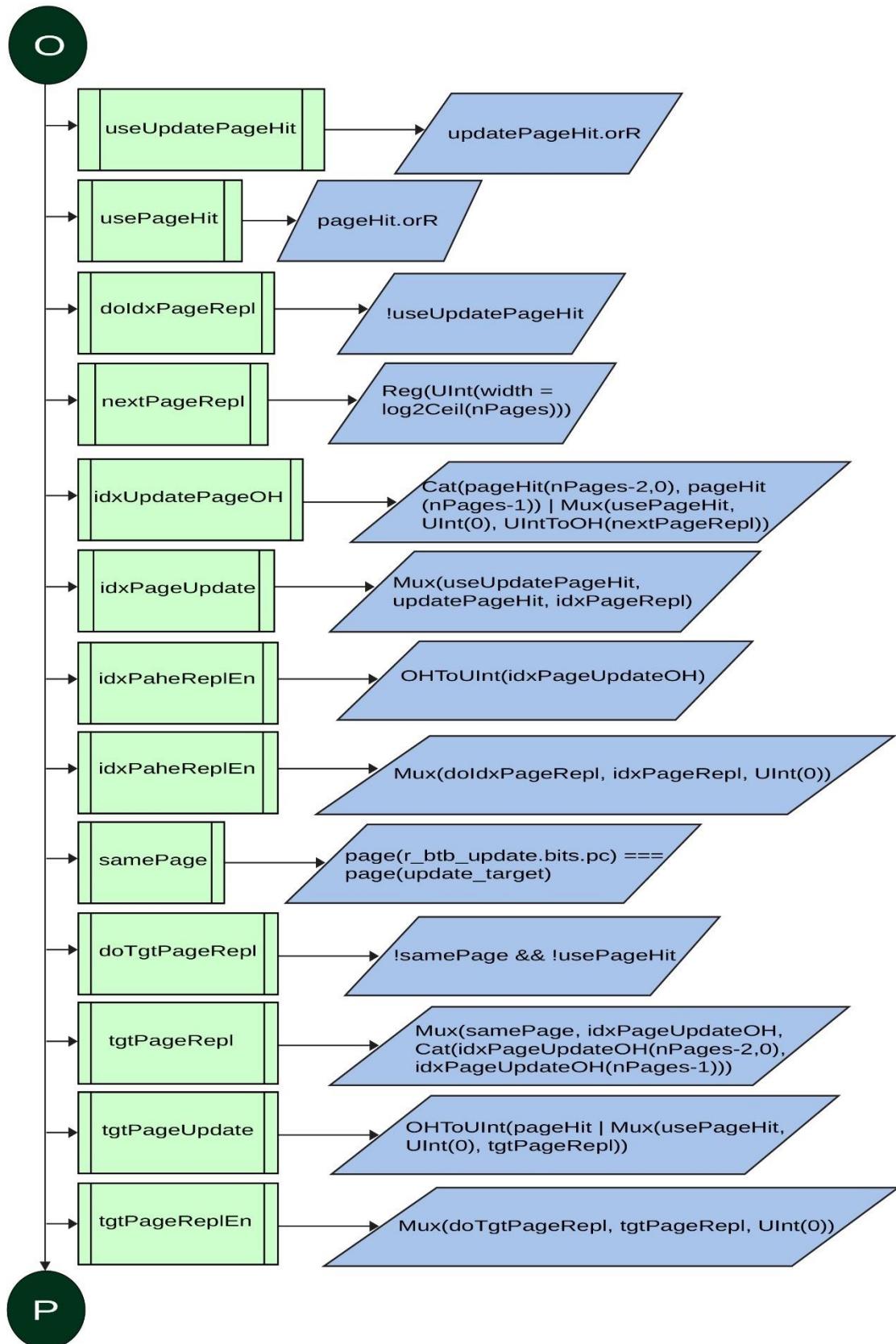
samePage hold a Boolean value for the return value of r_btb_update.bits.pc and update_target when passed to the private function page

doTgtPageRepl is initialized as AND product of NOT(samePage) and NOT(usePageHit)

tgtPageRepl is initialized as a Mux

tgtPageUpdate hold OHToUInt value of OR product of pageHit and the Mux which takes usePageHit, UInt(0) and tgtPageRepl as input parameters

tgtPageReplEn is initialized as a Mux



```

when (r_btb_update.valid && (doIdxPageRepl || doTgtPageRepl)) {
    val both = doIdxPageRepl && doTgtPageRepl
    val next = nextPageRepl + Mux[UInt](both, 2, 1)
    nextPageRepl := Mux(next >= nPages, next(0), next)
}

val repl = new PseudoLRU(entries)
val waddr = Mux(updateHit, updateHitAddr, repl.way)
val r_resp = Pipe(io.resp)

```

— explanation —

when the AND product of OR product of doIdxPageRepl and doTgtPageRepl, and r_btb_update.valid is True,

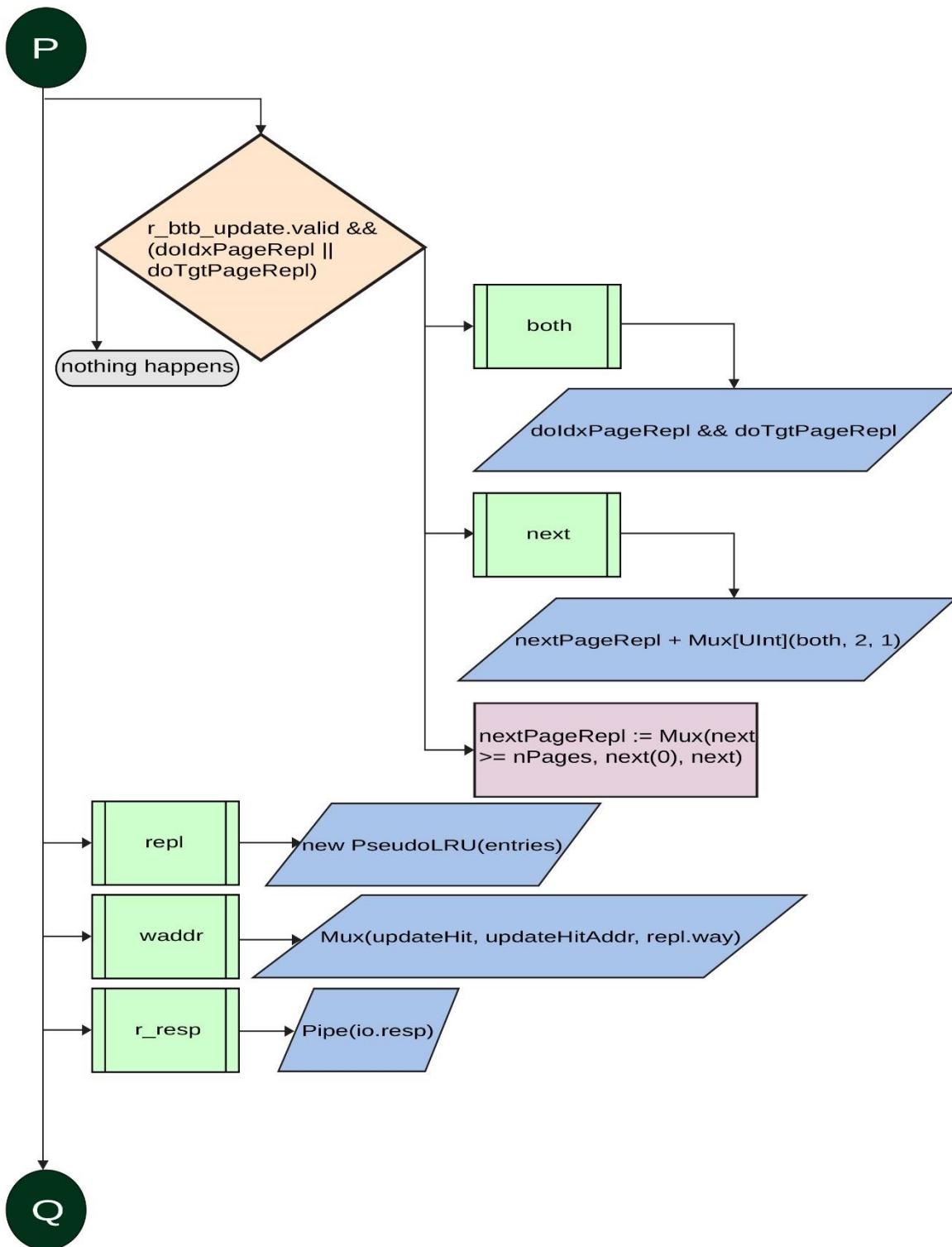
value both is initialized as AND product of doIdxPageRepl and doTgtPageRepl

value next is initialized as addition of nextPageRepl and an Unsigned Mux with parameters (both, 2, 1)

repl initialized as an instance of class PseudoLRU defined in Replacement.scala

waddr holds the Mux resultant of parameters (updateHit, UpdateHitAddr, repl.way)

r_resp is initialized as a pipe of io.resp



```

when (r_resp.valid && r_resp.bits.taken || r_btb_update.valid) {
    repl.access(Mux(r_btb_update.valid, waddr, r_resp.bits.entry))
}

when (r_btb_update.valid) {
    val mask = UIntToOH(waddr)
    idxs(waddr) := r_btb_update.bits.pc(matchBits-1, log2Up(coreInstBytes))
    tgts(waddr) := update_target(matchBits-1, log2Up(coreInstBytes))
    idxPages(waddr) := idxPageUpdate +& 1 // the +1 corresponds to the <<1
on io.resp.valid
    tgtPages(waddr) := tgtPageUpdate
    cfiType(waddr) := r_btb_update.bits.cfiType
    isValid := Mux(r_btb_update.bits.isValid, isValid | mask, isValid &
~mask)
}

```

— explanation —

when the combination is true, repl.access is given a Mux of parameters(r_btb_update.valid, waddr, r_resp.bits.entry)

When r_btb_update contains a valid value,

value mask is initialized as one hot encoding of waddr (i.e. UIntToOH(2.U) // results in "b0100".U)

the register idxs is given waddr as parameter and is wired to the value r_btb_update.bits.pc taking two arguments, the first one as matchBits (initialized in the class BTBParams) -1 and the second one as log2Up of coreInstBytes which is initialized in Core.scala under the trait HasCoreParameters,

val coreInstBytes = coreInstBits/8

val coreInstBits = coreParams.instBits

instBits is initialized as Int under the trait coreParams in Core.scala.

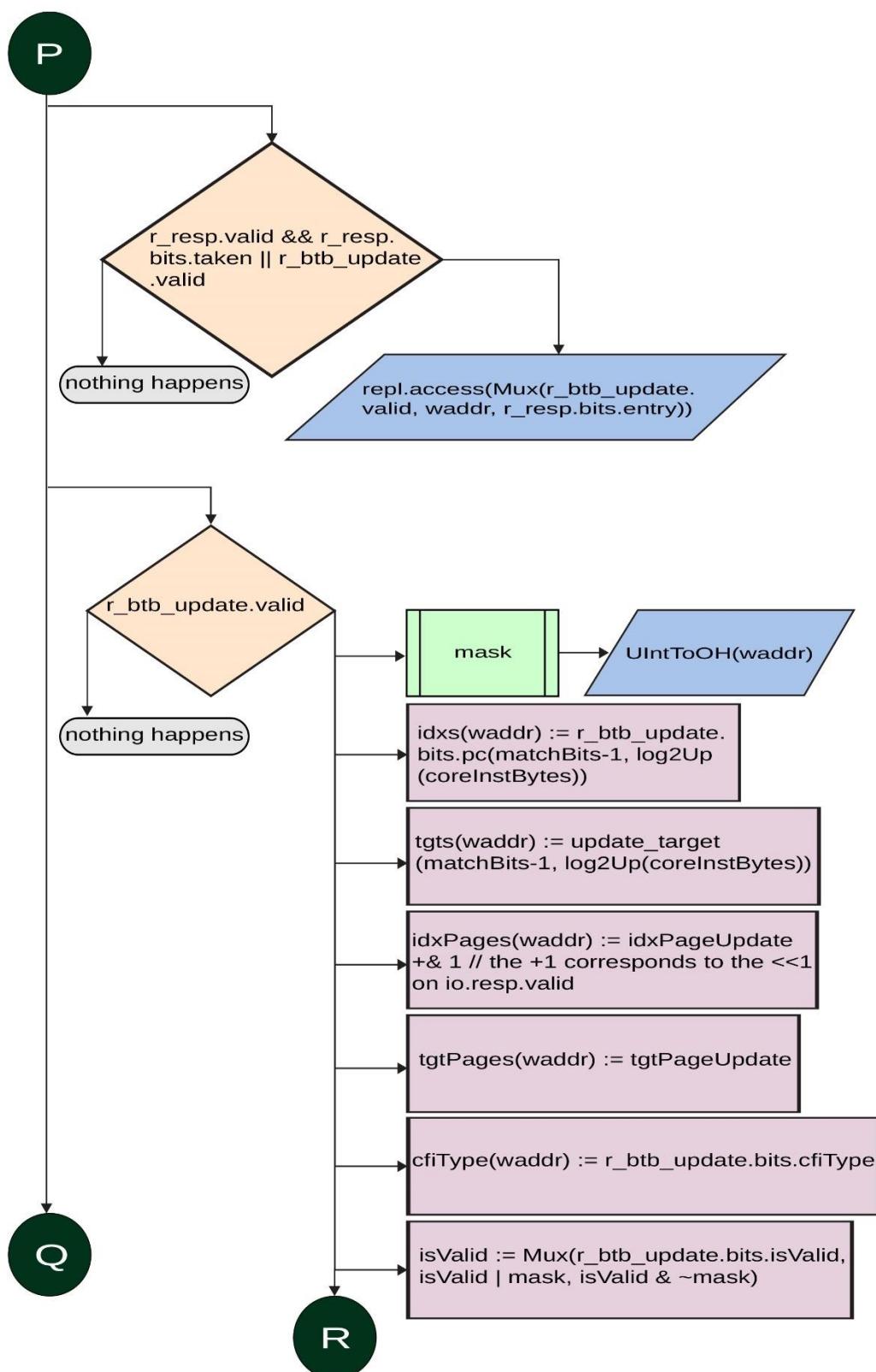
the register tgts is given the result of waddr as input parameter and is wired to the value update_target with matchBits-1 and log2UP(coreInstBytes) passed in as arguments.

the register idxPages takes waddr result as a parameter and is wired to the resultant of idxPageUpdate +& 1

the register tgtPages also takes waddr value as a parameter and is wired to tgtPageUpdate

cfiType takes waddr as an argument and is wired to the value of r_btb_update.bits.cfiType

isValid is wired to a Mux that takes r_btb_update.bits.isValid as the first argument, the OR product of isValid and mask as the second argument and the AND product of isValid and NOT(mask) as the third argument.



```

if (fetchWidth > 1)
    brIdx(waddr) := r_btb_update.bits.br_pc >> log2Up(coreInstBytes)

require(nPages % 2 == 0)
val idxWritesEven = !idxPageUpdate(0)

def writeBank(i: Int, mod: Int, en: UInt, data: UInt) =
    for (i <- i until nPages by mod)
        when (en(i)) { pages(i) := data }

writeBank(0, 2, Mux(idxWritesEven, idxPageReplEn, tgtPageReplEn),
         Mux(idxWritesEven, page(r_btb_update.bits.pc), page(update_target)))
writeBank(1, 2, Mux(idxWritesEven, tgtPageReplEn, idxPageReplEn),
         Mux(idxWritesEven, page(update_target), page(r_btb_update.bits.pc)))
pageValid := pageValid | tgtPageReplEn | idxPageReplEn
}

```

 explanation

if fetchWidth (initialized as Int in Core.scala under the trait CoreParams) is greater than 1, then r_btb_update.bits.br_pc is right shifted to number of bits decided by log2Up(coreInstBytes)

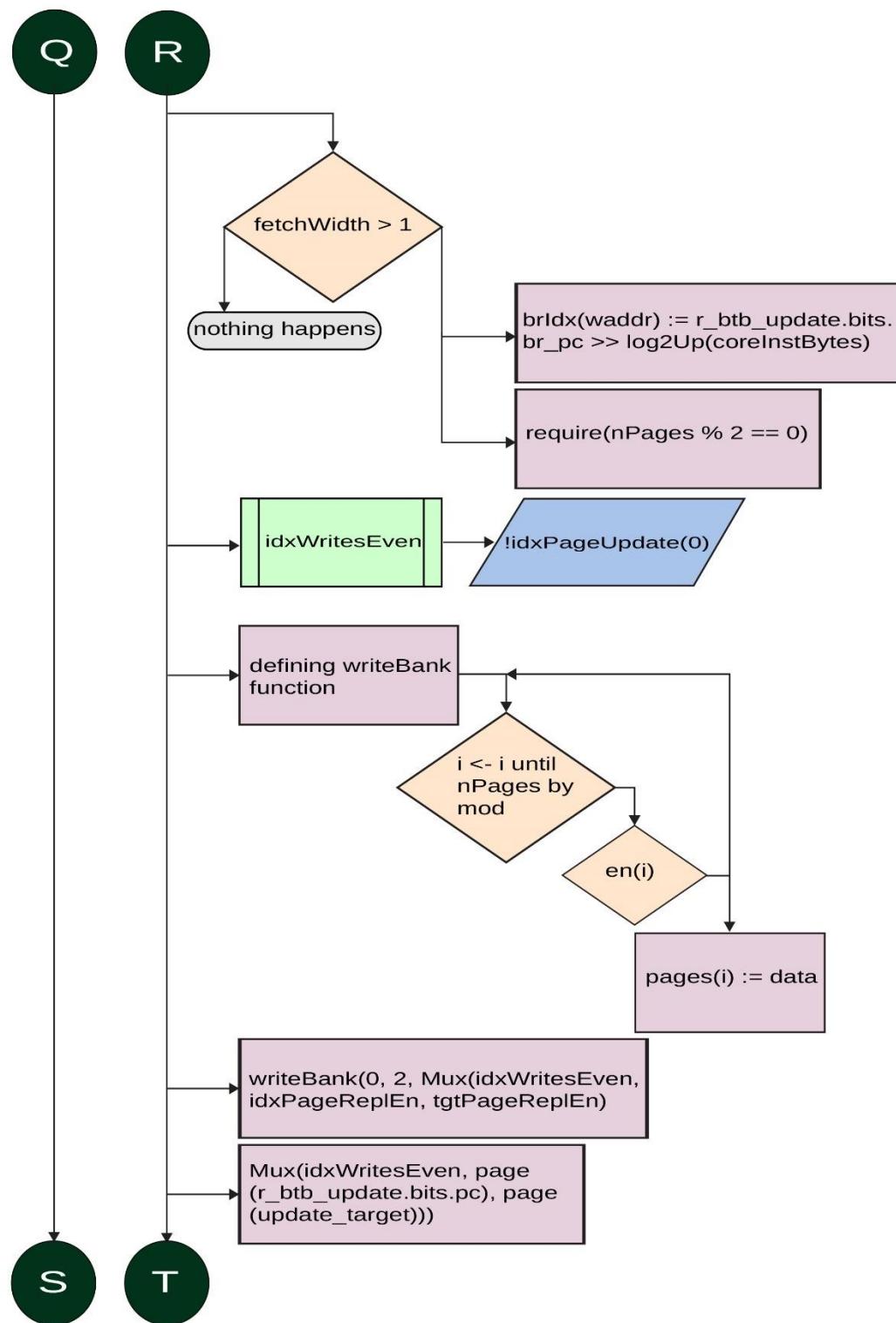
required to continue that nPages (initialized in BTBParams as Int = 6) when taken remainder with division by 2 is equal to 0

then idxWritesEven is initialized as the NOT of 0th register value of idxPageUpdate

A function writeBank is defined that takes 'i' and 'mod' as Int arguments and 'en' and 'data' as UInt arguments, this function runs a for loop that run for i in (i until nPages by mod), and when en(i) holds a valid value the pages(i) is wired to data

then the function is called passing some variables defined in the class

then pageValid is wired to the OR product of pageValid, tgtPageReplEn and idxPageReplEn, initialized in the same class



```

writeBank(1, 2, Mux(idxWritesEven, tgtPageReplEn, idxPageReplEn),
          Mux(idxWritesEven, page(update_target), page(r_btb_update.bits.pc)))
pageValid := pageValid | tgtPageReplEn | idxPageReplEn
}
io.resp.valid := (pageHit << 1) (Mux1H(idxHit, idxPages))
io.resp.bits.taken := true
io.resp.bits.target := Cat(pages(Mux1H(idxHit, tgtPages)), Mux1H(idxHit,
tgts) << log2Up(coreInstBytes))
io.resp.bits.entry := OHToUInt(idxHit)
io.resp.bits.bridx := (if (fetchWidth > 1) Mux1H(idxHit, brIdx) else
UInt(0))
io.resp.bits.mask := Cat((UInt(1) << ~Mux(io.resp.bits.taken,
~io.resp.bits.bridx, UInt(0)))-1, UInt(1))
io.resp.bits.cfiType := Mux1H(idxHit, cfiType)

```

explanation

A function writeBank is defined that takes 'i' and 'mod' as Int arguments and 'en' and 'data' as UInt arguments, this function runs a for loop that run for i in (i until nPages by mod), and when en(i) holds a valid value the pages(i) is wired to data then the function is called passing some variables defined in the class

then pageValid is wired to the OR product of pageValid, tgtPageReplEn and idxPageReplEn, initialized in the same class.

io.resp.valid is wired to pageHit left shift 1 bit and a Mux tree taking input parameters idxHit and idxPages

io.resp.bits.taken is hard wired to True

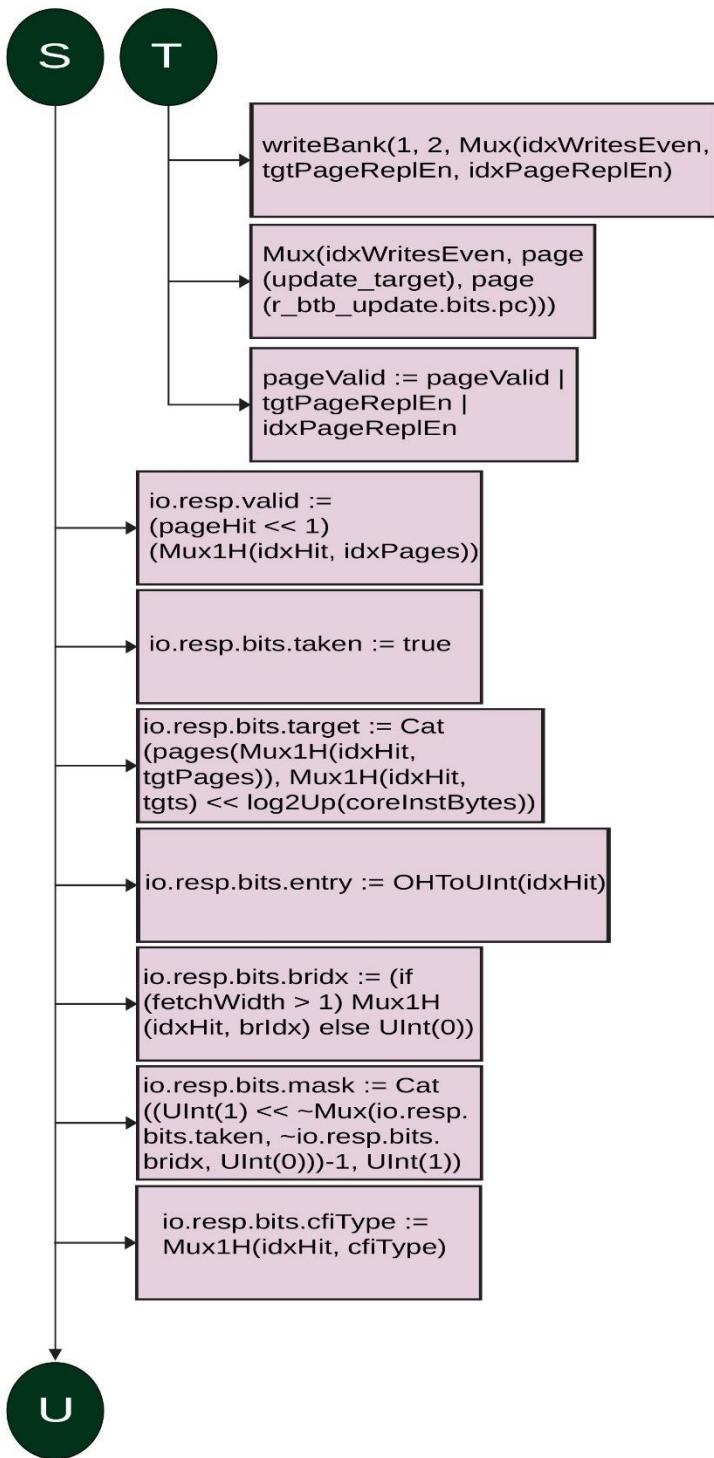
io.resp.bits.target is wired to concatenation of pages in Mux tree (idxHit, tgtPages) and the Mux tree (idxHit, tgts) leftshifted be the value log2Up(coreInstBytes)

io.resp.bits.entry is wired to OHToUInt (i.e. OHToUInt("b0100".U) // results in 2.U) bit conversion of idxHit

io.resp.bits.bridx will be wired to the Mux tree result of (idxHit, brIdx) if fetchWidth is greater than 1 else it will be wired to UInt(0)

io.resp.bits.mask is wired to the concatenation of UInt(1) left shifted to the value given by Mux(io.resp.bits.taken, Not(io.resp.bits.bridx), UInt(0)) -1 and UInt(1)

io.resp.bits.cfiType is wired to the Mux Tree (idxHit, cfiType)



```

when (PopCountAtLeast(idxHit, 2)) {
    isValid := isValid & ~idxHit
}
when (io.flush) {
    isValid := 0
}

if (btbParams.bhtParams.nonEmpty) {
    val bht = new BHT(Annotated.params(this, btbParams.bhtParams.get))
    val isBranch = (idxHit & cfitype.map(_ === CFIType.branch).asUInt).orR
    val res = bht.get(io.req.bits.addr)
    when (io.bht_advance.valid) {
        bht.advanceHistory(io.bht_advance.bits.bht.taken)
    }
}

```

explanation

When PopCountAtLeast, passed idxHit and 2 as parameters returns true then isValid is wired to the AND product of isValid and NOT(idxHit)

PopCountAtLeast is defined as Object in Misc.scala placed under util imported in BTB.scala

```

object PopCountAtLeast {
    private def two(x: UInt): (Bool, Bool) = x.getWidth match {
        case 1 => (x.asBool, Bool(false))
        case n =>
            val half = x.getWidth / 2
            val (leftOne, leftTwo) = two(x(half - 1, 0))
            val (rightOne, rightTwo) = two(x.getWidth - 1, half))
            (leftOne || rightOne, leftTwo || rightTwo || (leftOne && rightOne))
    }
    def apply(x: UInt, n: Int): Bool = n match {
        case 0 => Bool(true)
        case 1 => x.orR
        case 2 => two(x)._2
        case 3 => PopCount(x) >= UInt(n)
    }
}

```

when io.flush is True then isValid is wired to 0

If bhtParams initialized in btbParams is not Empty,

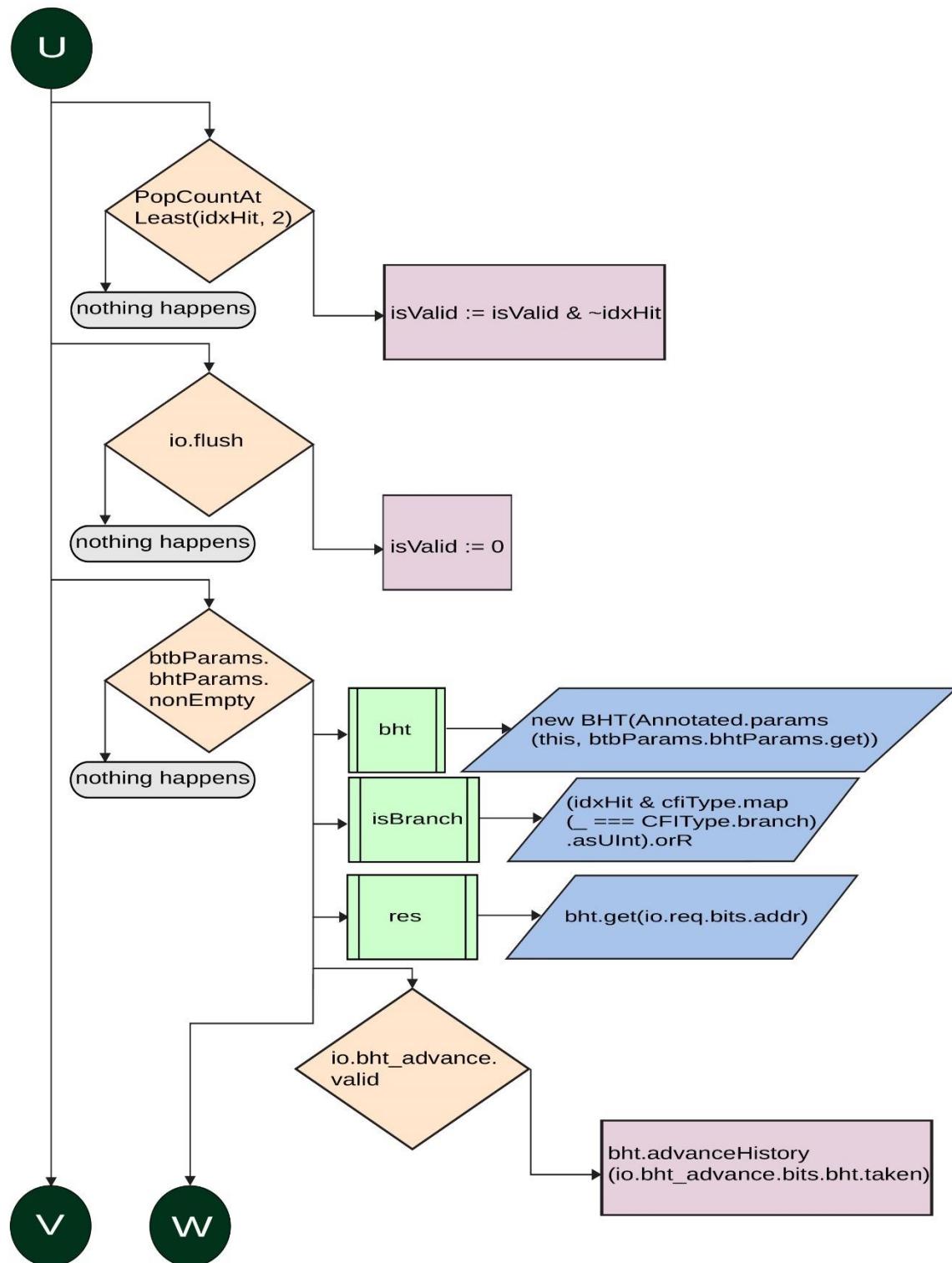
bht is initialized as a new instance of class BHT given annotated params from the variables in case class bhtParams

isBranch is initialized as the AND product of idxHit and cfitype.map(_ === CFIType.branch).asUInt

res is initialized as the return value of get function of the bht instance passing in io.req.bits.addr as parameter

when io.bht_advance is valid and when io.bht_update.bits.branch is True,

the updateTable function of bht is called given the parameters
(io.bht_update.bits.pc, io.bht_update.bits.prediction, io.bht_update.bits.taken)



```

    when (io.bht_update.bits.mispredict) {
        bht.updateHistory(io.bht_update.bits.pc,
io.bht_update.bits.prediction, io.bht_update.bits.taken)
    }
    .elsewhen (io.bht_update.bits.mispredict) {
        bht.resetHistory(io.bht_update.bits.prediction)
    }
}
when (!res.taken && isBranch) { io.resp.bits.taken := false }
io.resp.bits.bht := res
}

```

 explanation

when `io.bht_update.bits.mispredict` is True,

`updateTable` function of `bht` is called passing in the parameters
`(io.bht_update.bits.pc, io.bht_update.bits.prediction, io.bht_update.bits.taken)`

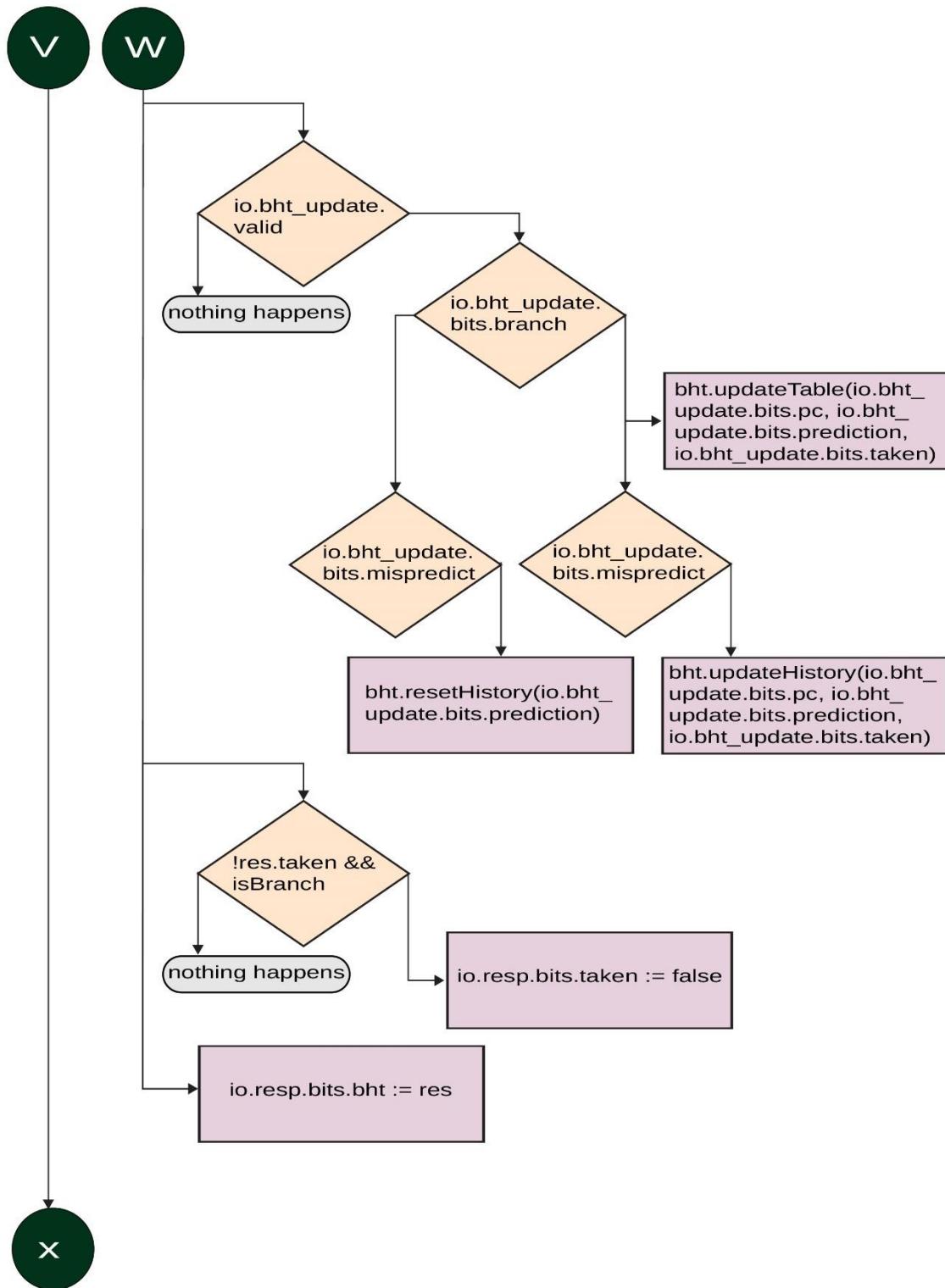
else when `io.bht_update` is valid, `io.bht_update.bits.branch` is False and
`io.bht_update.bits.mispredict` is True,

`resetHistory` function of `bht` is called passing the parameter
`(io.bht_update.bits.prediction)`

when the AND of NOT(`res.taken`) and `isBranch`, is True,

`io.resp.bits.taken` is wired to False

`io.resp.bits.bht` is wired to `res`



```

io.ras_head.valid := !ras.isEmpty
io.ras_head.bits := ras.peek
when (!ras.isEmpty && doPeek) {
    io.resp.bits.target := ras.peek
}
when (io.ras_update.valid) {
    when (io.ras_update.bits.cfiType === CFIType.call) {
        ras.push(io.ras_update.bits.returnAddr)
    } .elsewhen (io.ras_update.bits.cfiType === CFIType.ret) {
        ras.pop()
    }
}
}

```

 explanation

if btbParams.nRAS is greater than 0,

ras is initialized as an instance of RAS class, parameter passed -> btbParams.nRAS

dePeek is initialized as the AND product of idxHit and cfiType.map(_ === CFIType.ret).asUInt

io.ras_head.valid is wired to NOT of ras.isEmpty

io.ras_head.bits is wired to ras.peek

when AND product of NOT(ras.isEmpty) and doPeek is True,

io.resp.bits.target will be wired to ras.peek

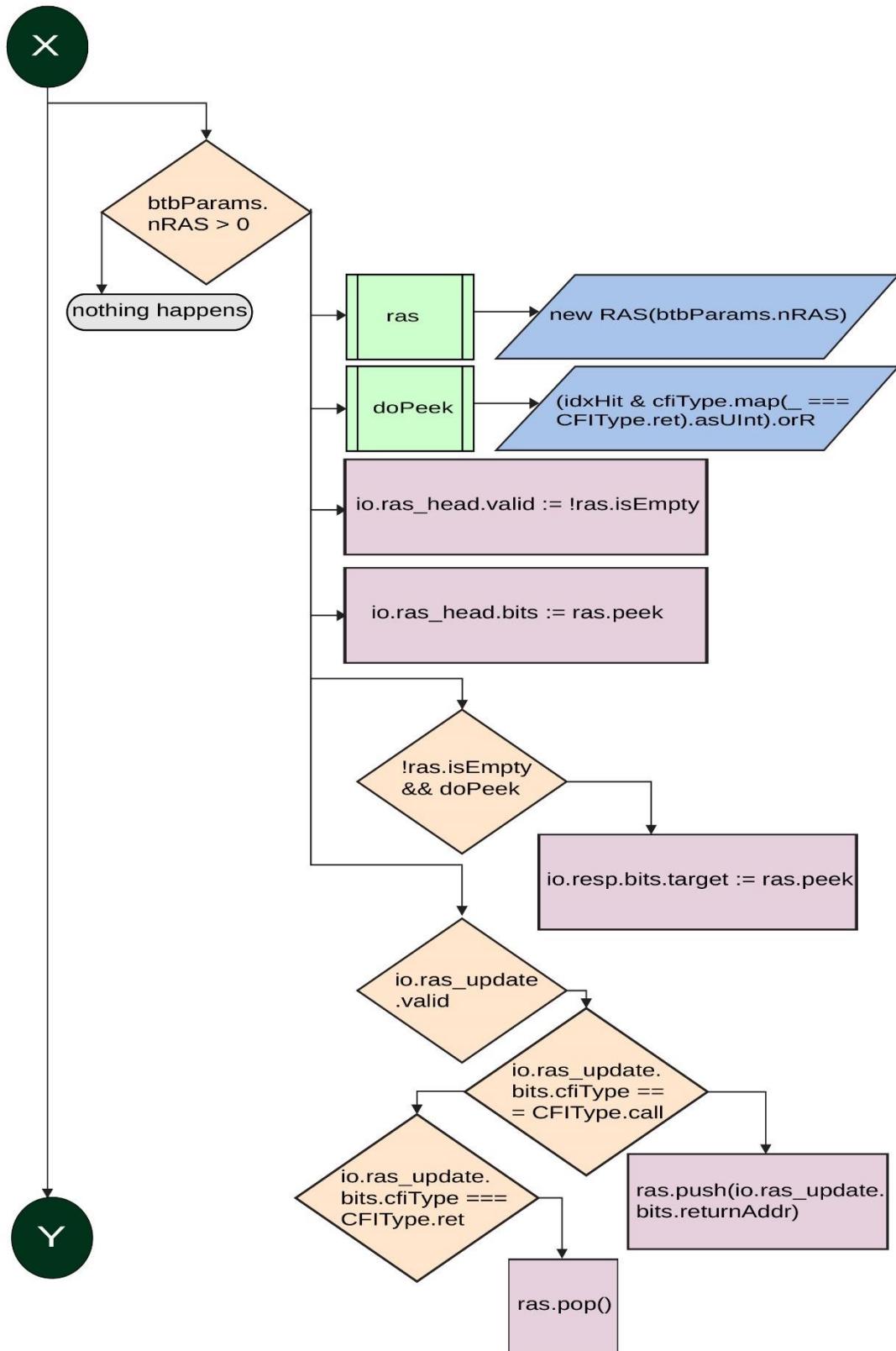
when io.ras_update.valid is True,

when io.ras_update.bits.cfiType is equivalent to CFIType.call then,

ras.push function is called with parameter io.ras_update.bits.returnAddr

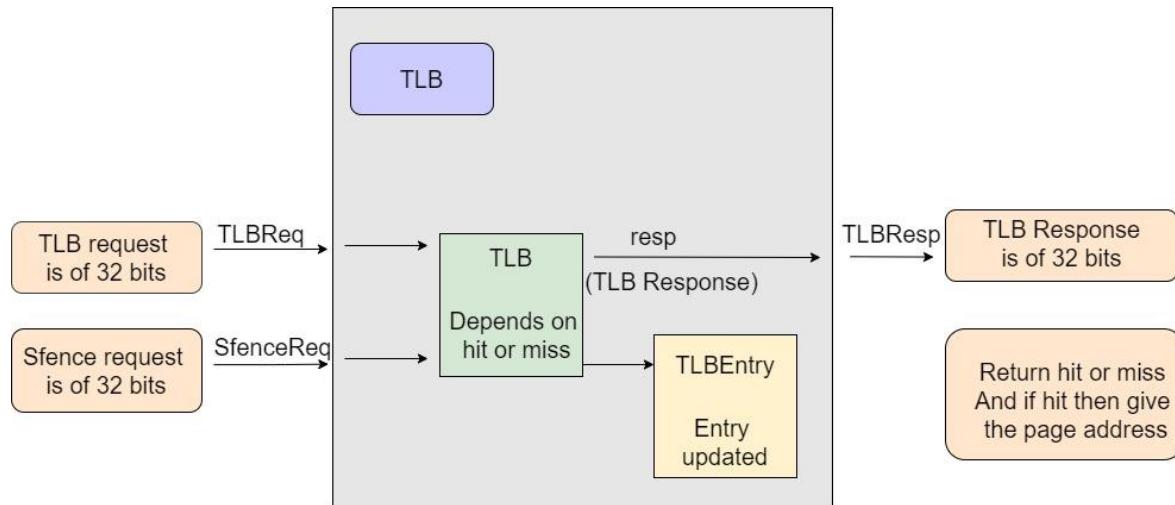
else when io.ras_update.bits.cfiType is equivalent to CFIType.ret then,

call ras.pop



TLB

Block Diagram:



DEEP DIVE INTO CODE

Explanation (By Means of Flowcharts) :

```
class SFenceReq(implicit p: Parameters) extends CoreBundle()(p) {  
    val rs1 = Bool()  
    val rs2 = Bool()  
    val addr = UInt(width = vaddrBits)  
    val asid = UInt(width = asIdBits max 1) // TODO zero-width  
}
```

— explanation —

class SFenceReq is created, with parameter, p, extended by CoreBundle,
rs1 variable has Bool,
rs2 variable has Bool,
addr has UInt of width vaddrBits,
asid has UInt of width, asldBits max with 1

```

class TLBReq(lgMaxSize: Int) (implicit p: Parameters) extends
CoreBundle() (p) {
    val vaddr = UInt(width = vaddrBitsExtended)
    val passthrough = Bool()
    val size = UInt(width = log2Ceil(lgMaxSize + 1))
    val cmd = Bits(width = M_SZ)

    override def cloneType = new TLBReq(lgMaxSize).asInstanceOf[this.type]
}

```

 explanation

class TLBReq is created with parameters, lgMaxSize as Int, extended by CoreBundle, having,

vaddr has UInt with width vaddrBitsExtended,

passthrough has Bool,

size has UInt of width log2Ceil of lgMaxSize +1

cmd has Bits of width M_SZ

CoreBundle's method cloneType is overrided in this class, having

TLBReq object with lgMaxSize as instance of this.type.

```
class TLBExceptions extends Bundle {
    val ld = Bool()
    val st = Bool()
    val inst = Bool()
}
```

— explanation —

class TLBExceptions is extended by Bundle, having

ld as Bool,

st as Bool,

inst as Bool,

```
class TLBResp(implicit p: Parameters) extends CoreBundle()(p) {
    // lookup responses
    val miss = Bool()
    val paddr = UInt(width = paddrBits)
    val pf = new TLBExceptions
    val ae = new TLBExceptions
    val ma = new TLBExceptions
    val cacheable = Bool()
    val must_alloc = Bool()
    val prefetchable = Bool()
}
```

— explanation —

Class TLBResp is created with parameters, p, extended by CoreBundle, having miss as Bool,
paddr as UInt of width paddrBits,
pf as TLBExceptions object
se as TLBException objrct,
ms as TLBException object,
cacheable as Bool,
must_alloc as Bool,
prefetchable as Bool

```
class TLBEntryData(implicit p: Parameters) extends CoreBundle()(p) {
    val ppn = UInt(width = ppnBits)
    val u = Bool()
    val g = Bool()
    val ae = Bool()
    val sw = Bool()
    val sx = Bool()
    val sr = Bool()
    val pw = Bool()
    val px = Bool()
    val pr = Bool()
    val ppp = Bool() // PutPartial
    val pal = Bool() // AMO logical
    val paa = Bool() // AMO arithmetic
    val eff = Bool() // get/put effects
    val c = Bool()
    val fragmented_superpage = Bool()
}
```

— explanation —

Class TLBEntryData is created with parameters, p, extended by CoreBundle, having Ppn as UInt of width ppnBits,
All other variables as Bool.

```

class TLBEntry(val nSectors: Int, val superpage: Boolean, val
superpageOnly: Boolean) (implicit p: Parameters) extends CoreBundle() (p) {
    require(nSectors == 1 || !superpage)
    require(!superpageOnly || superpage)

    val level = UInt(width = log2Ceil(pgLevels))
    val tag = UInt(width = vpnBits)
    val data = Vec(nSectors, UInt(width = new TLBEntryData().getWidth))
    val valid = Vec(nSectors, Bool())
    def entry_data = data.map(_.asTypeOf(new TLBEntryData()))
}

```

— explanation —

TLBEntry class is created with parameters, nSectors as Int, superpage as Boolean, superpageOnly as Boolean,

Require method is called with condition, nSectors == 1 OR'ed with NOT of superpage,

Require method is called with condition. NOT of superpageOnly OR'ed with superpage,

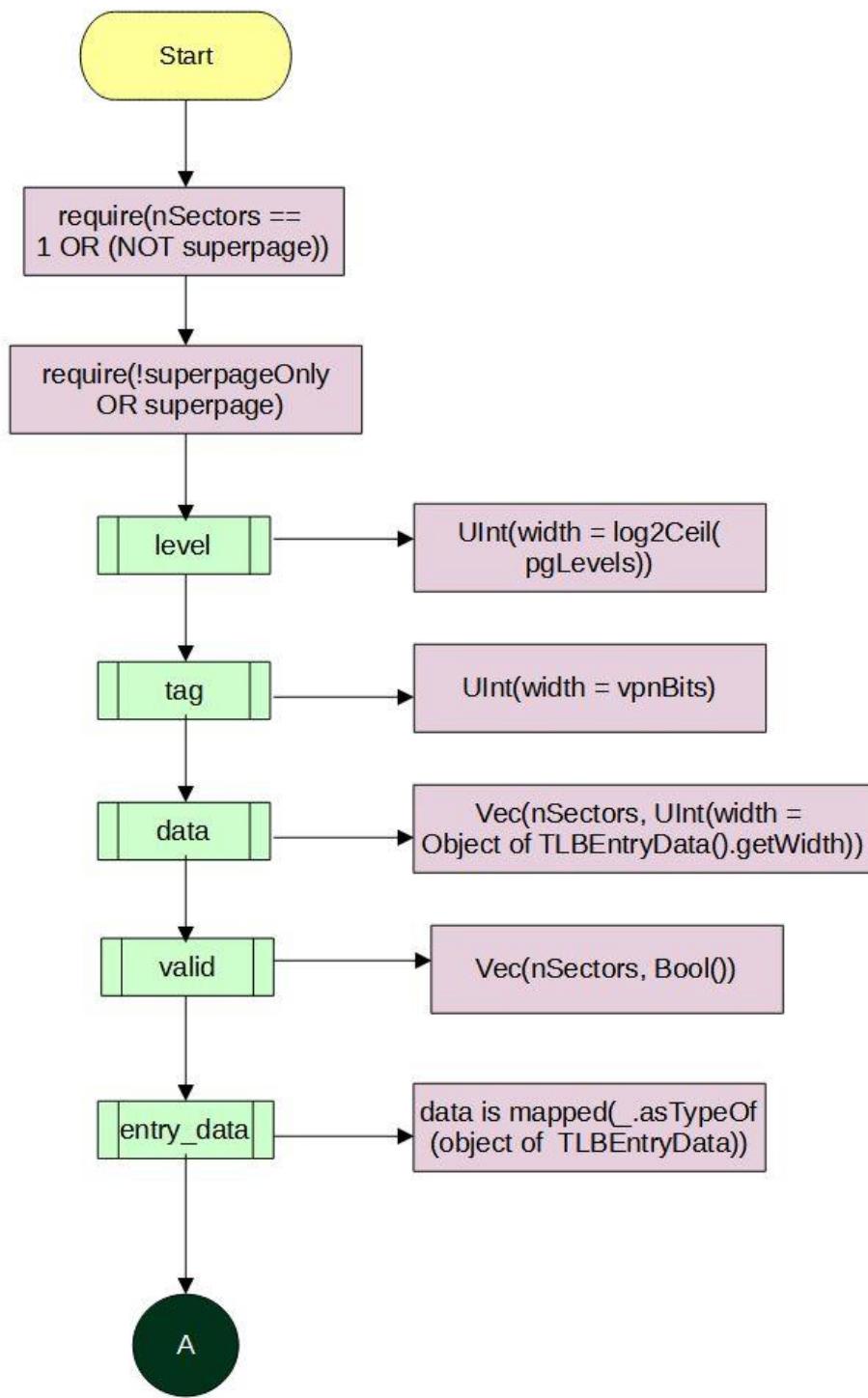
Level has UInt of width log2Ceil of pgLevels

Tag has UInt of width vpnBits,

Data has Vec of length nSectors and UInt of width, width of TLBEntryData class, on every index of vector,

Valid has Vec of length nSectors with Bool on every index,

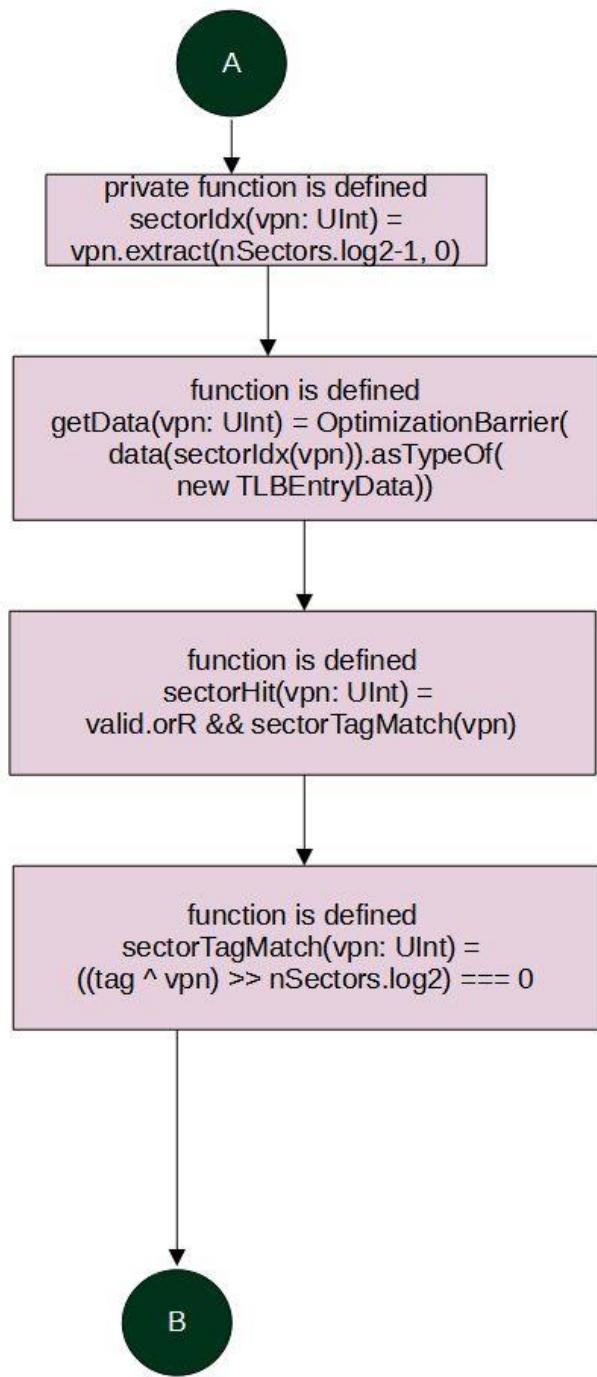
Entry_data method is initiated which has data mapped with _ as type of TLBEntryData object,



```
private def sectorIdx(vpn: UInt) = vpn.extract(nSectors.log2-1, 0)
def getData(vpn: UInt) =
OptimizationBarrier(data(sectorIdx(vpn)).asTypeOf(new TLBEntryData))
def sectorHit(vpn: UInt) = valid.orR && sectorTagMatch(vpn)
def sectorTagMatch(vpn: UInt) = ((tag ^ vpn) >> nSectors.log2) === 0
```

— explanation —

private method of sectorIdx is defined with parameter, vpn as UInt, having,
vpn extracted, by nSectors.log2 - 1, and 0,
getData method is defined with parameter, vpn as UInt, having,
OptimizatinBarrier of data at index sectorIdx of vpn, as type of TLBEntryData object,
sectorHit method is defined with parameter, vpn as UInt, having,
valid bits OR'ed with each other, AND'ed with sectorTagMatch of vpn,
sectorTagMatch method is defined with parameters, vpn as UInt, having,
tag of power vpn, right shifted by nSectors.log2, === 0.



```

def hit(vpn: UInt) = {
    if (superpage && usingVM) {
        var tagMatch = valid.head
        for (j <- 0 until pgLevels) {
            val base = vpnBits - (j + 1) * pgLevelBits
            val ignore = level < j || superpageOnly && j == pgLevels - 1
            tagMatch = tagMatch && (ignore || tag(base + pgLevelBits - 1, base)
==== vpn(base + pgLevelBits - 1, base))
        }
        tagMatch
    } else {
        val idx = sectorIdx(vpn)
        valid(idx) && sectorTagMatch(vpn)
    }
}

```

— explanation —

hit method is defined with vpn as UInt, having,

if superpage is true AND usingVM is also True, then

tagMatch variable is defined with valid.head,

for loop is iterated as j from 0 to pgLevels -1, in which,

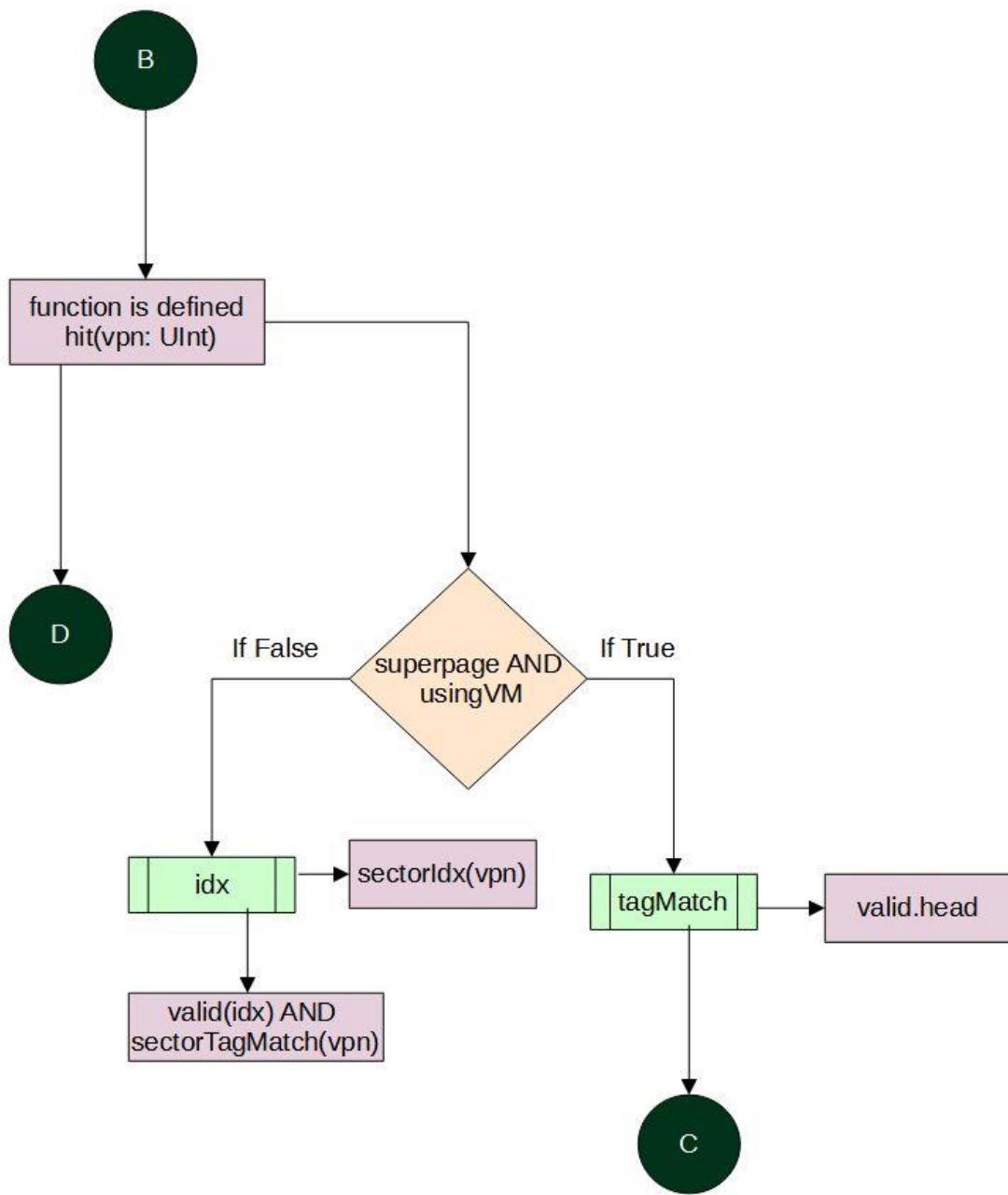
base variable is defined with, vpnBits – (j+1) multiplied by pgLevelBits,

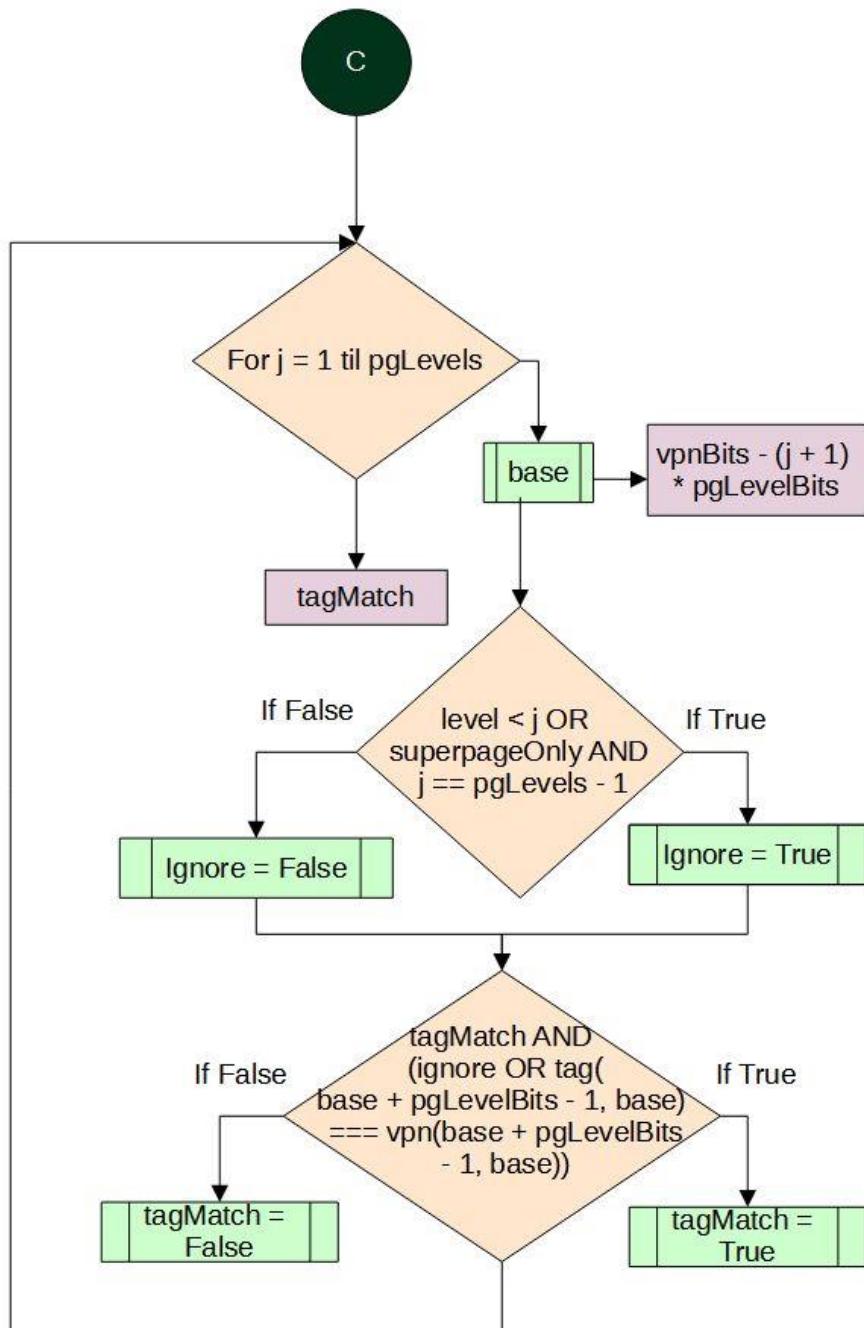
ignore variable has level less than j OR'ed with superpageOnly AND'ed with j === pgLevels -1

then, after the loop. tagMatch is returned,

else, if the if clause is false, then, idx variable is defined, with sectorIdx of vpn,

valid of idx is AND'ed with sectorTagMatch of vpn, is returned.





```

def ppn(vpn: UInt) = {
    val data = getData(vpn)
    if (superpage && usingVM) {
        var res = data.ppn >> pgLevelBits*(pgLevels - 1)
        for (j <- 1 until pgLevels) {
            val ignore = level < j || superpageOnly && j == pgLevels - 1
            res = Cat(res, (Mux(ignore, vpn, 0.U) | data.ppn)(vpnBits - j*pgLevelBits - 1, vpnBits - (j + 1)*pgLevelBits))
        }
        res
    } else {
        data.ppn
    }
}

```

 explanation

ppn method is defined with vpn as UInt, having,

data has getData of vpn,

if superpage is true, AND usingVM is true, then

res has data.ppn right shifted by, pgLevelBits multiplied by pgLevels – 1

for loop is iterated as j from 1 to pgLevels – 1, in which,

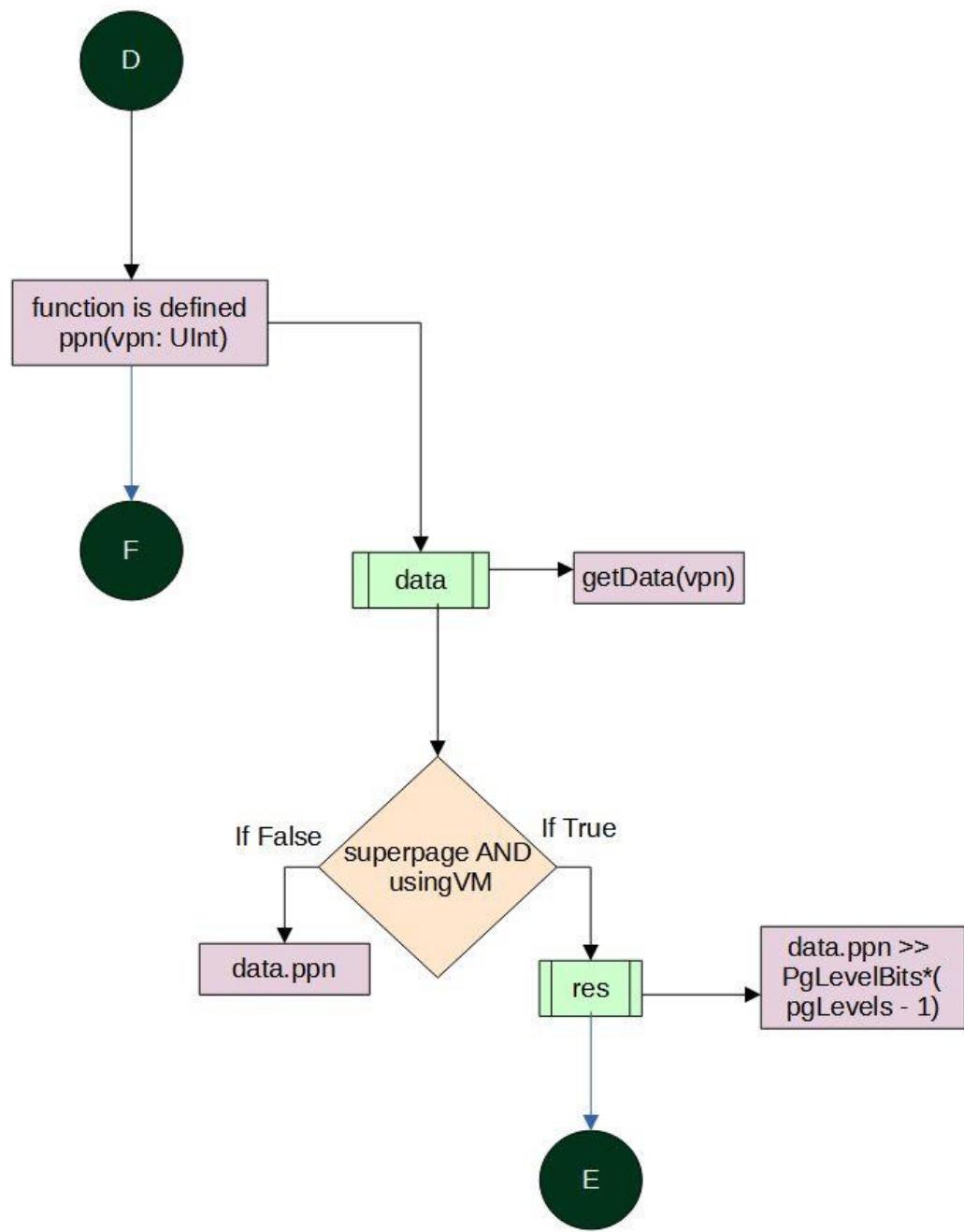
ignore has level less than j OR'ed with siperpageOnly AND'ed with j == pgLevels – 1,

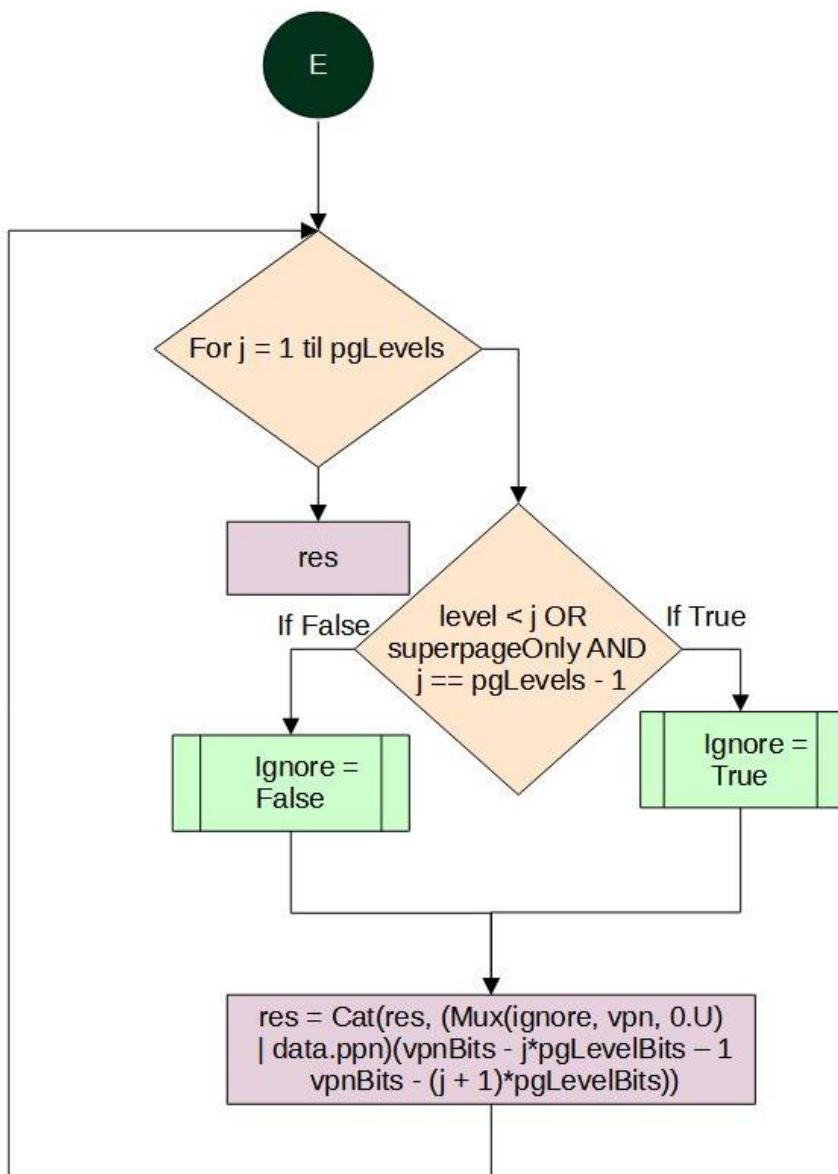
res has Concatenation of res and Mux of sel as ignore , with, vpn and 0.u, OR'ed with data.ppn, and vpnBits = j multiplied by pgLevelBits – 1, vpnBits – j + 1 multiplied with pgLevelBits.

res is returned, after loop

else ,

data.ppn is returned.





```
def insert(tag: UInt, level: UInt, entry: TLBEntryData) {
    this.tag := tag
    this.level := level.extract(log2Ceil(pgLevels - superpageOnly.toInt)-1,
0)

    val idx = sectorIdx(tag)
    valid(idx) := true
    data(idx) := entry.asUInt
}
```

— explanation —

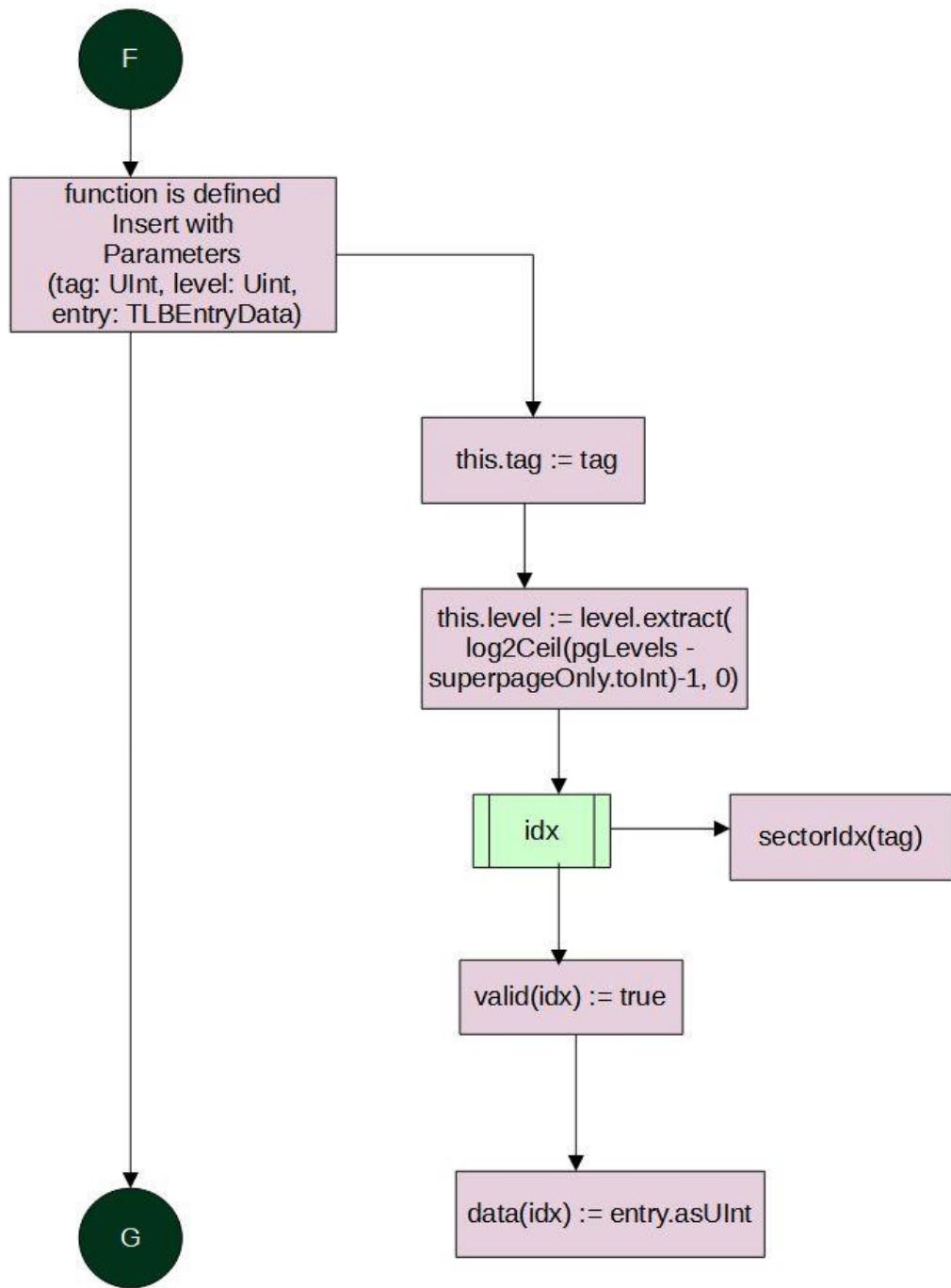
insert method is created, with tag as UInt, level as UInt, entry as TLBEntryData object,
this.tag is wired to tag,

this.level is wired to level extracted with log2Ceil of pgLevels – superpageOnly.toInt - 1, and 0

idx variable is defined with sectorIdx of tag,

valid of idx is wired true,

data of idx is wired to entry asUInt



```
def invalidate() { valid.foreach(_ := false) }
def invalidateVPN(vpn: UInt) {
    if (superpage) {
        when (hit(vpn)) { invalidate() }
    } else {
        when (sectorTagMatch(vpn)) { valid(sectorIdx(vpn)) := false }
    }
}
```

— explanation —

invalidate method is created with valid with every index wired false,

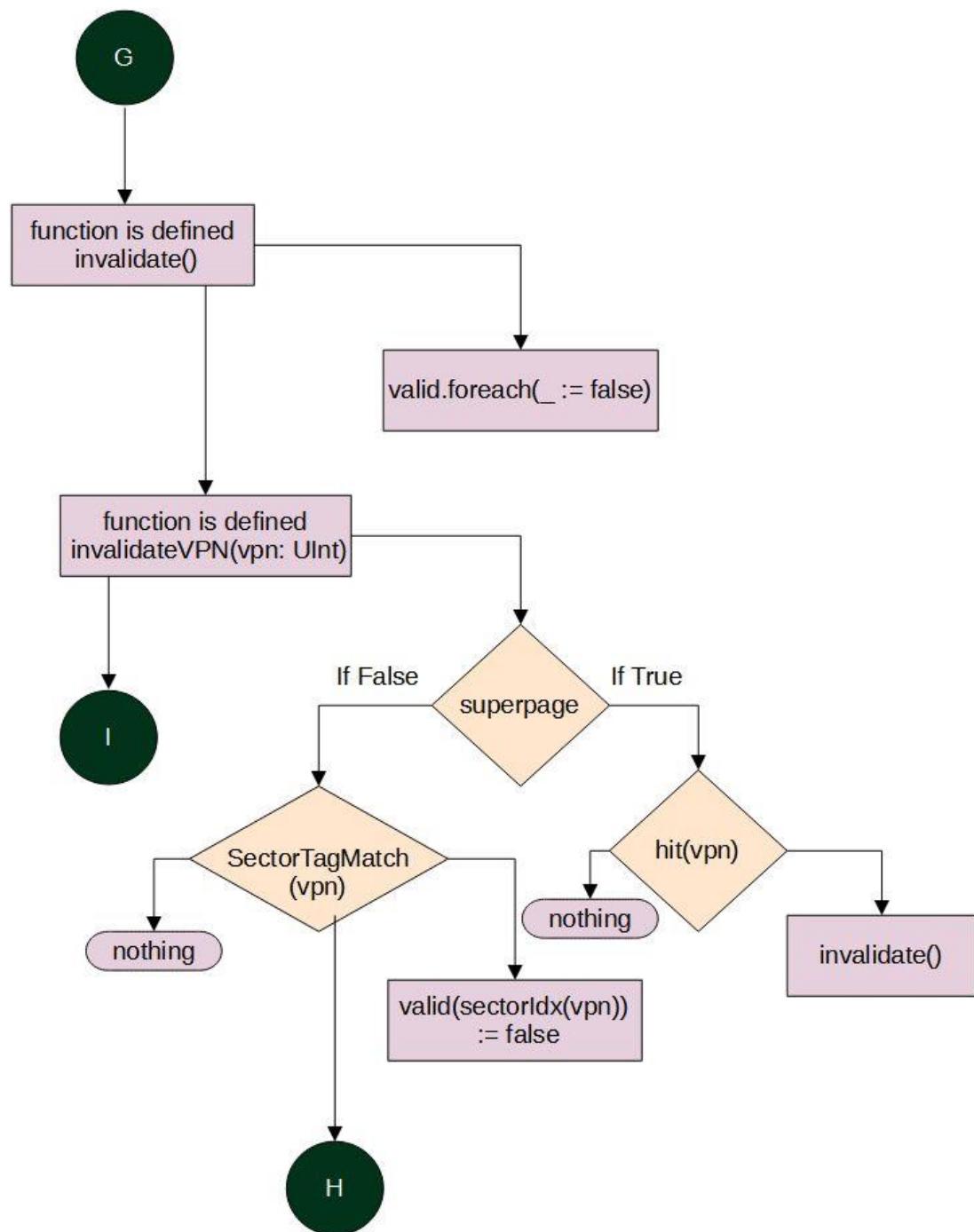
invalidateVPN method is created with vpn as UInt, having,

if superpage is true, then

when hit on index vpn will be true, then invalidate method will be called,

else, when sectorTagMatch of index vpn will be true , then,

sectorIdx of index vpn, is wired to false.



```

// For fragmented superpage mappings, we assume the worst (largest)
// case, and zap entries whose most-significant VPNs match
when (((tag ^ vpn) >> (pgLevelBits * (pgLevels - 1))) === 0) {
    for ((v, e) <- valid zip entry_data)
        when (e.fragmented_superpage) { v := false }
}
}
}

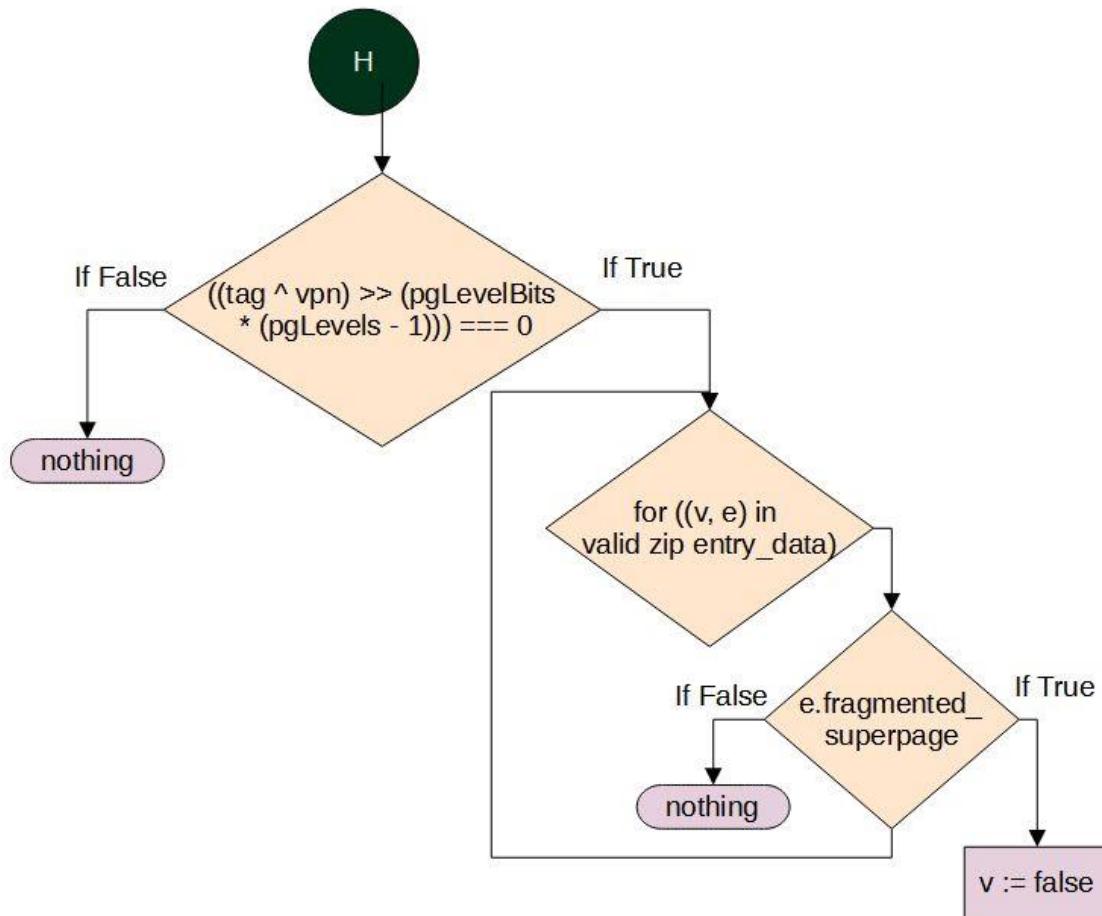
```

explanation

when tag of power vpn, is right shifted by pgLevelBits * pgLevels -1, === 0

for loop is iterated on entry, splitting thr data into v and e, in which

when e.fragmented_superpage is true, then v is wired false.



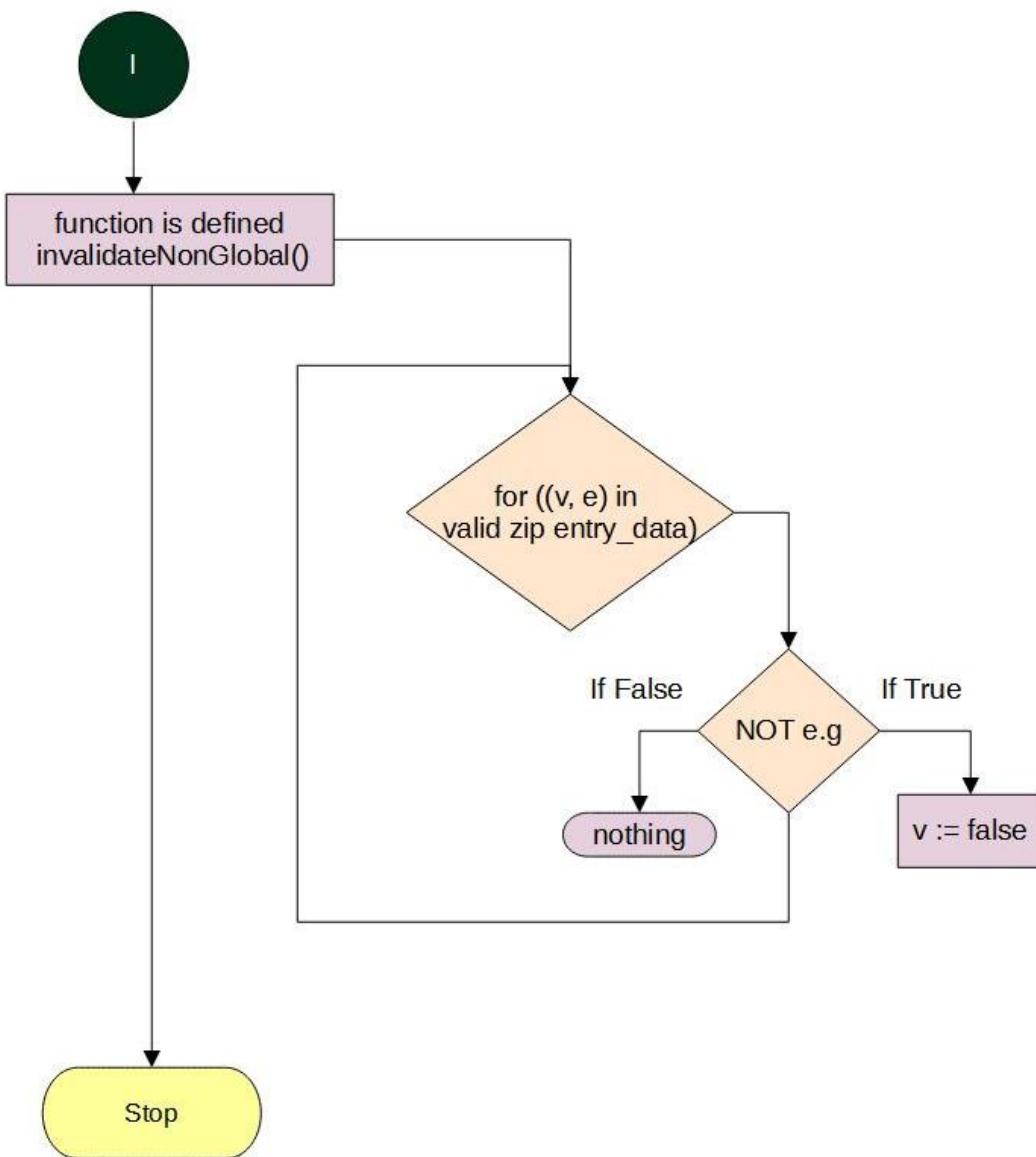
```

def invalidateNonGlobal() {
    for ((v, e) <- valid zip entry_data)
        when (!e.g) { v := false }
    }
}

```

explanation

invalidateNonGlobal method is created, in which,
 for loop is iterated over_entry with data split into v and e, in which,
 when e.g is false, then,
 v is wired false.



```
case class TLBConfig(  
    nSets: Int,  
    nWays: Int,  
    nSectors: Int = 4,  
    nSuperpageEntries: Int = 4)
```

explanation

case class of TLBConfig is created, having,
nSets as Int,
nWays as Int,
nSectors as Int of 4,
nSuperPageEntries as Int of 4.

```
class TLB(instruction: Boolean, lgMaxSize: Int, cfg: TLBConfig) (implicit
edge: TLEdgeOut, p: Parameters) extends CoreModule()(p) {
    val io = new Bundle {
        val req = Decoupled(new TLBReq(lgMaxSize)).flip
        val resp = new TLBResp().asOutput
        val sfence = Valid(new SFenceReq).asInput
        val ptw = new TLBPTWIO
        val kill = Bool(INPUT) // suppress a TLB refill, one cycle after a miss
    }
}
```

— explanation —

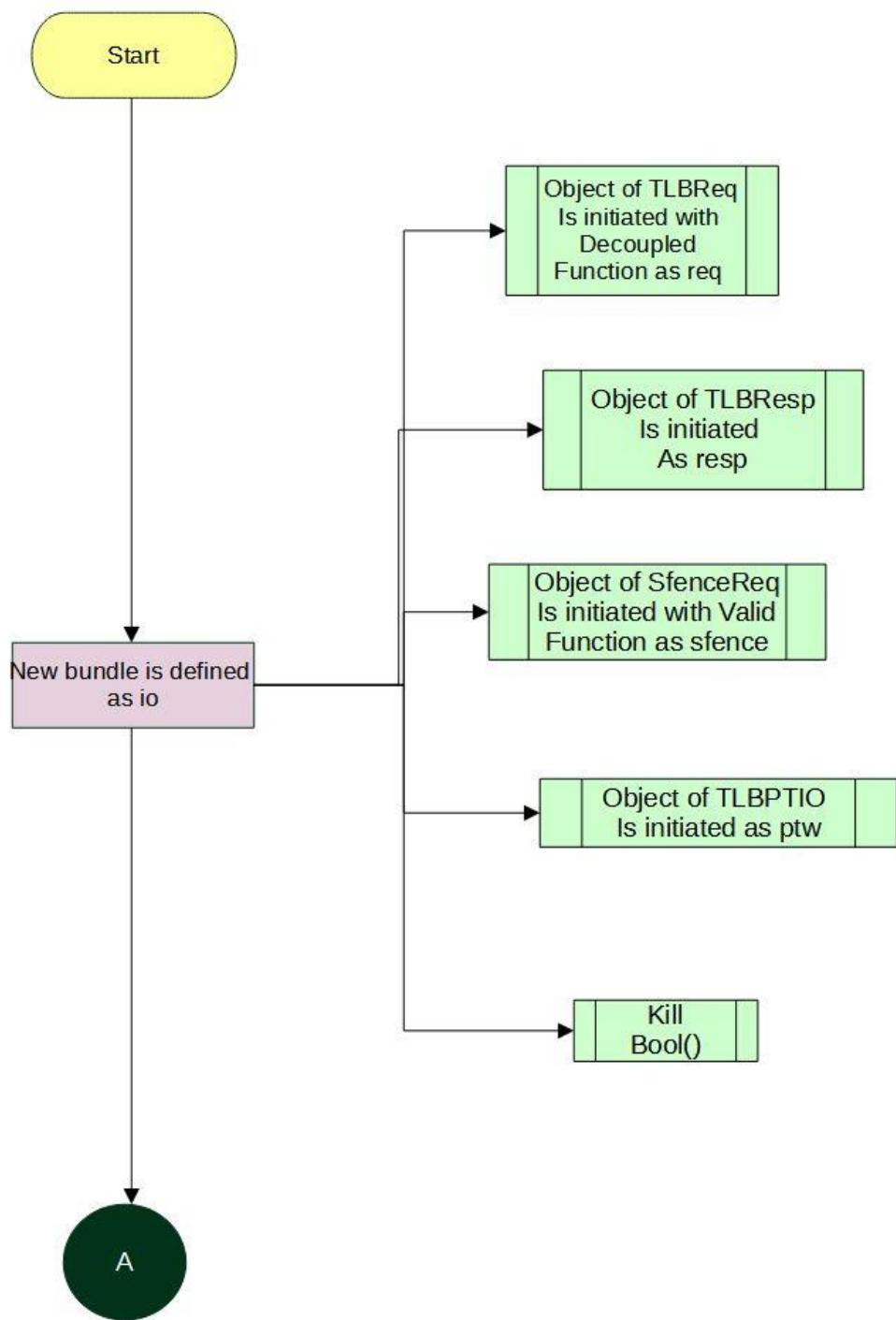
An object has been made of class TLB request with parameter lgMaxSize where Decoupled() adds a decoupled handshaking protocol to a data bundle. And the output of it is being flipped

An object has been made of class TLB response .

An object has been made of class SFenceReq, where Valid() checks whether the selected form is valid or whether all selected elements are valid (reference: <https://jqueryvalidation.org/valid/>)

An object for page table walker is being initialized which back traces to the class

--TLBPTWIO--



```
val pageGranularityPMPs = pmpGranularity >= (1 << pgIdxBits)
val vpn = io.req.bits.vaddr(vaddrBits-1, pgIdxBits)
val memIdx = vpn.extract(cfg.nSectors.log2 + cfg.nSets.log2 - 1,
cfg.nSectors.log2)
val sectored_entries = Reg(Vec(cfg.nSets, Vec(cfg.nWays / cfg.nSectors,
new TLBEntry(cfg.nSectors, false, false))))
```

— explanation —

A value has been assigned which is being compared to pmpGranularity which back traces to:

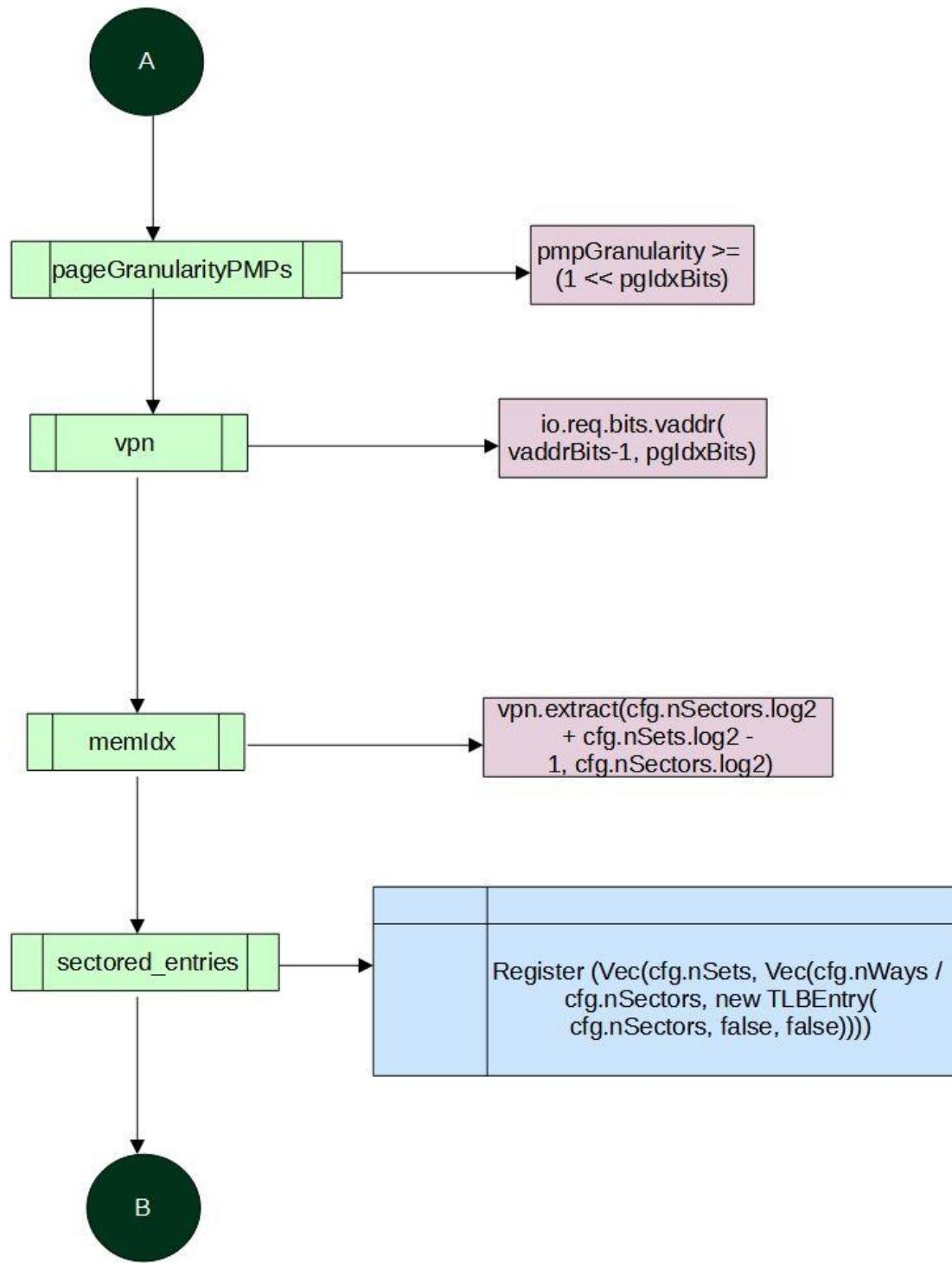
```
val pmpGranularity: Int = 4
```

And pgIdxBits which equals to:

```
def pgIdxBits: Int = 12
```

memIdx has been assigned a value referenced via vpn, Where the extract() function imports variables into the local symbol table from an array.

Value has been assigned to sectored_entries where reg() is a utility for constructing hardware registers in general Vec only needs to be used when there is a need to express the hardware collection in a Reg or IO Bundle or when access to elements of the array is indexed via a hardware signal



```
def all_entries = ordinary_entries ++ special_entry
def all_real_entries = sectored_entries.flatten ++ superpage_entries ++
special_entry

val s_ready :: s_request :: s_wait :: s_wait_invalidate :: Nil =
Enum(UInt(), 4)
val state = Reg(init=s_ready)
```

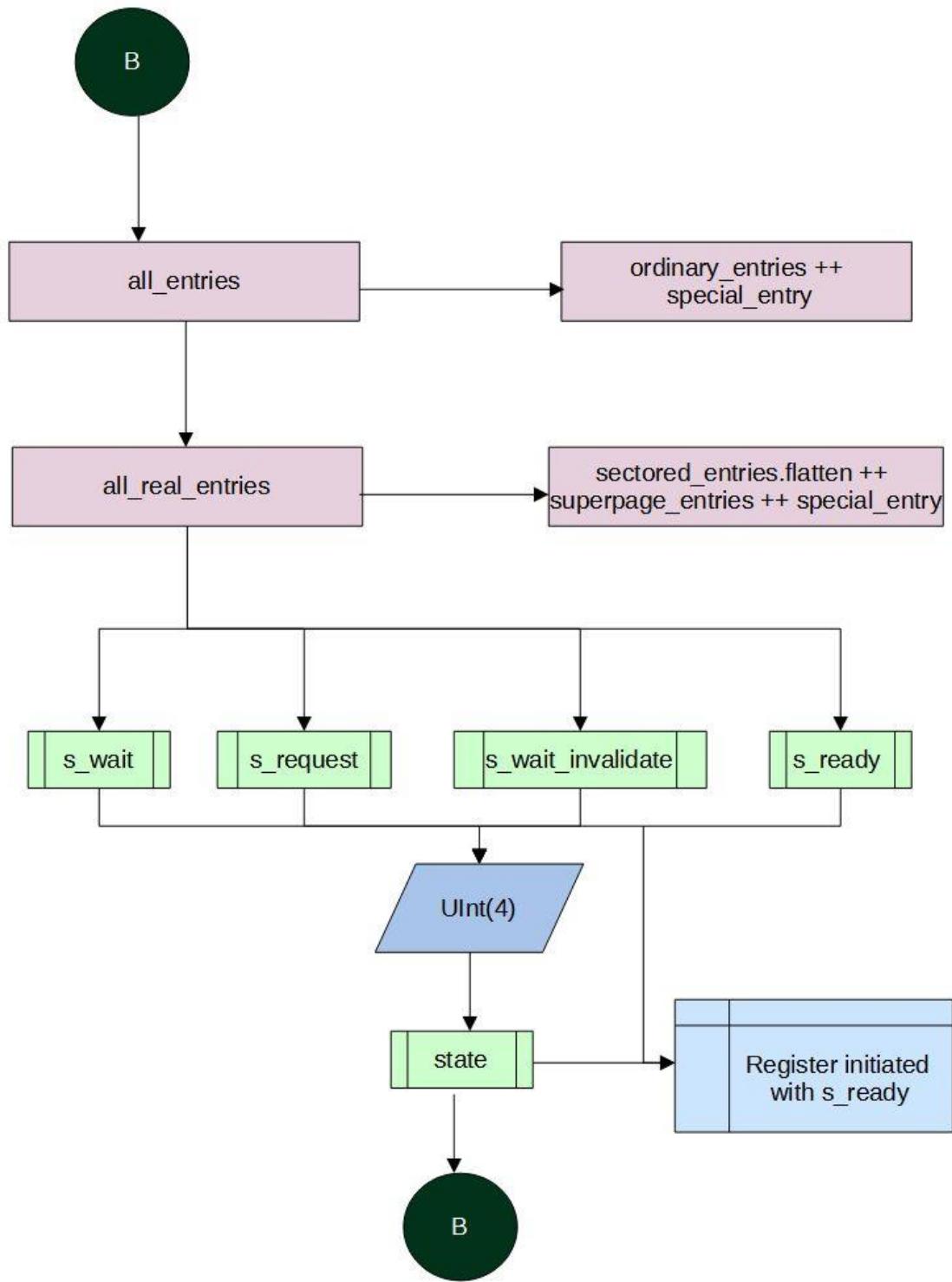
explanation

a function has been defined to concatenate ordinary_entries and special_entry

a function has been defined to concatenate sectored_entries, superpage_entries and special_entry

values of s_ready, s_wait, s_wait_invalidate being unpacked with a list using enum.scala

a value being assigned to variable state



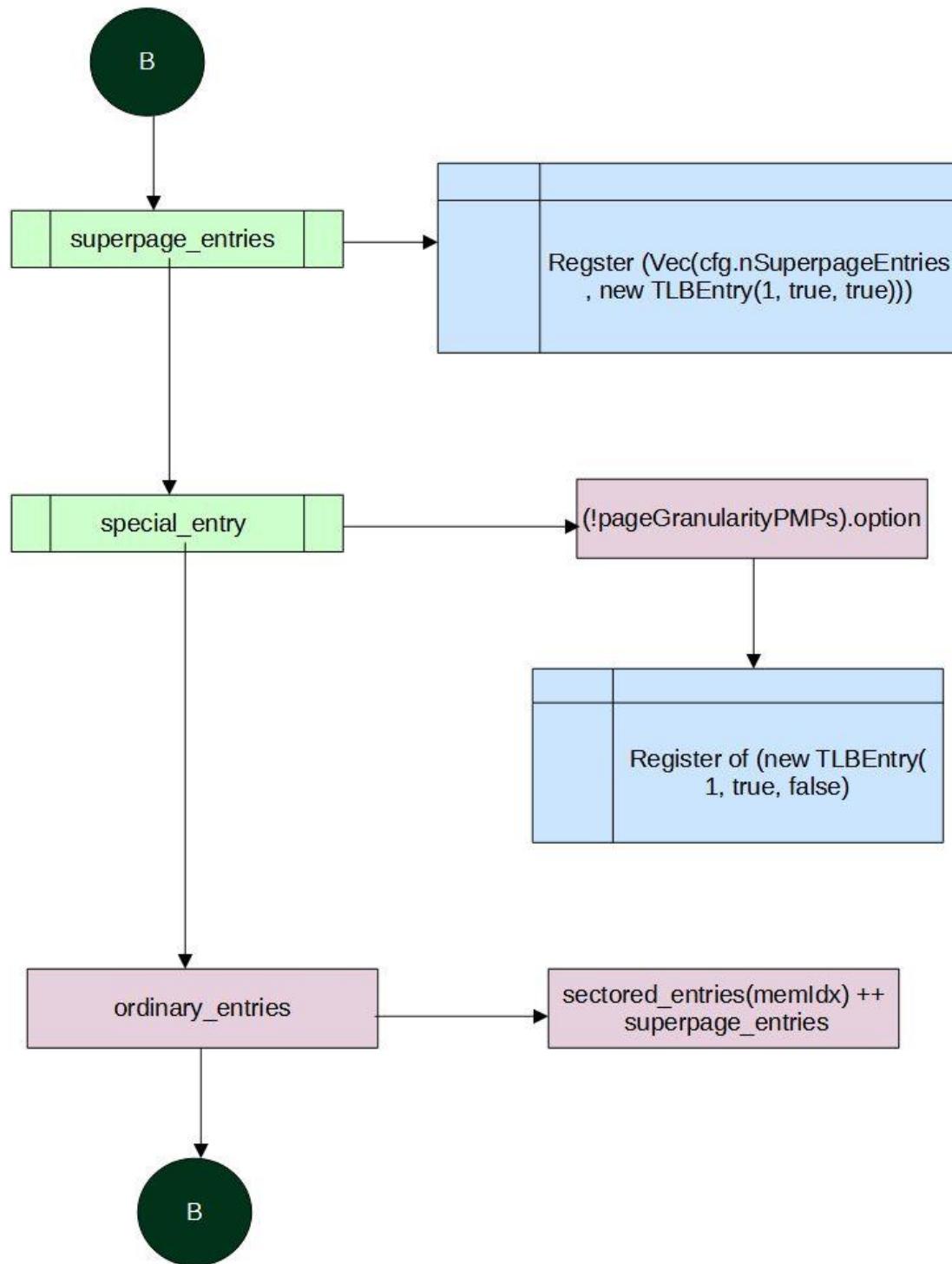
```
val superpage_entries = Reg(Vec(cfg.nSuperpageEntries, new TLBEntry(1,
true, true)))
val special_entry = (!pageGranularityPMPs).option(Reg(new TLBEntry(1,
true, false)))
def ordinary_entries = sectored_entries(memIdx) ++ superpage_entries
```

explanation

values declared for superpage_entries

values declared for special_entry

A function defined as ordinary_entries concatenating the values of sectored_entries which takes memidx as parameter and superpage_entries



```
val r_refill_tag = Reg(UInt(width = vpnBits))
val r_superpage_repl_addr = Reg(UInt(log2Ceil(superpage_entries.size).W))
val r_sectored_repl_addr =
Reg(UInt(log2Ceil(sectored_entries(0).size).W))
val r_sectored_hit_addr = Reg(UInt(log2Ceil(sectored_entries(0).size).W))
val r_sectored_hit = Reg(Bool())
```

explanation

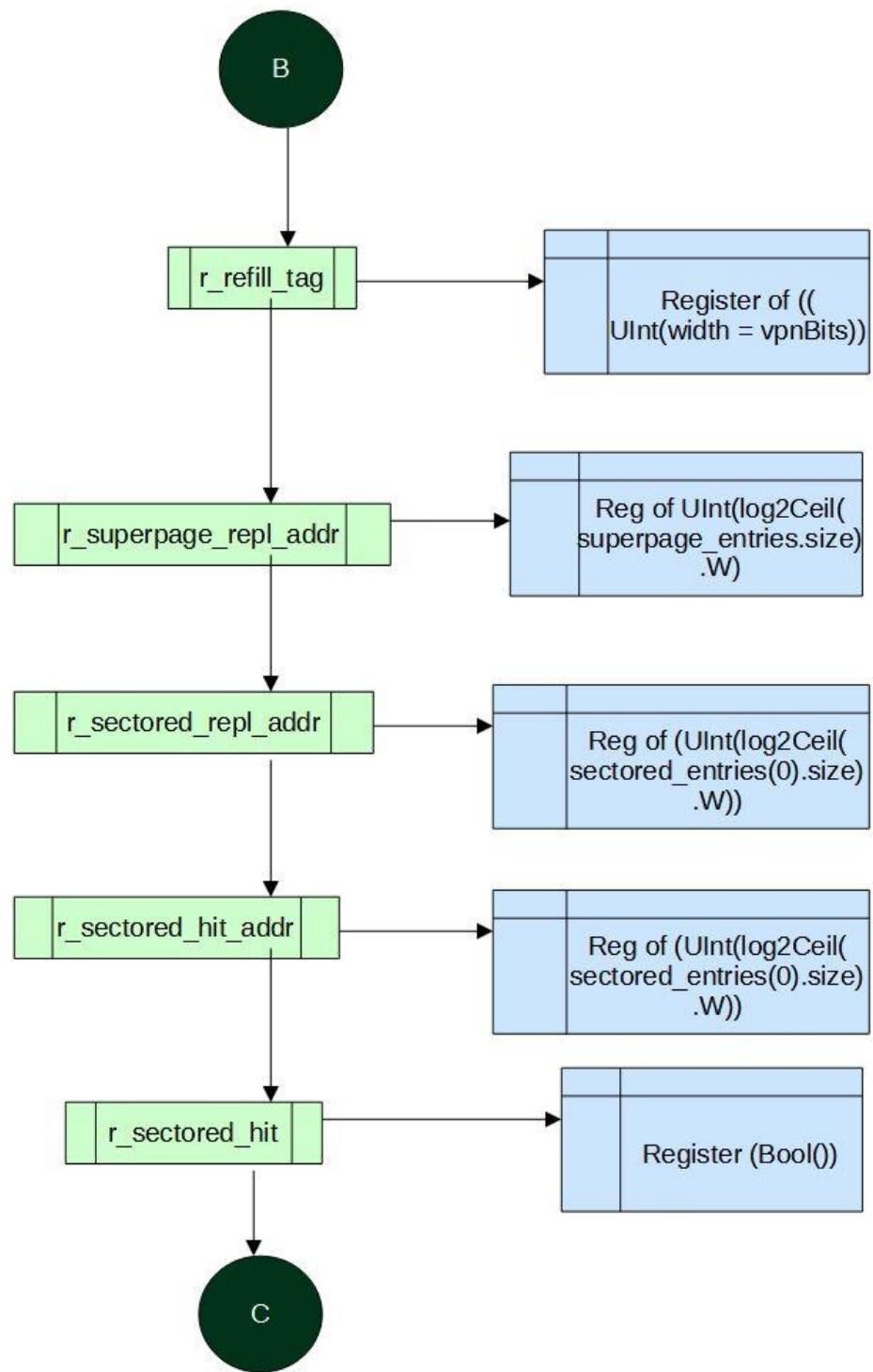
A hardware register is constructed for value of r_refil_tag

A hardware register is constructed for value of r_superpage_repl_addr

A hardware register is constructed for value of r_sectored_repl_addr

A hardware register is constructed for value of r_sectored_hit_addr

A hardware register is constructed for boolean value of r_sectored_hit



```

val priv = if (instruction) io.ptw.status.prv else io.ptw.status.dprv
val priv_s = priv(0)
val priv_uses_vm = priv <= PRV.S
val vm_enabled = Bool(usingVM) &&
io.ptw.ptbr.mode(io.ptw.ptbr.mode.getWidth-1) && priv_uses_vm &&
!io.req.bits.passthrough

```

 explanation

A condition has been implied, if the value of instruction is true, then the value of io.ptw.status.prv is assigned to the variable priv. Else the value of io.ptw.status.dprv is assigned

The value of priv is assigned to the variable priv_s referenced via a parameter value i.e 'o'

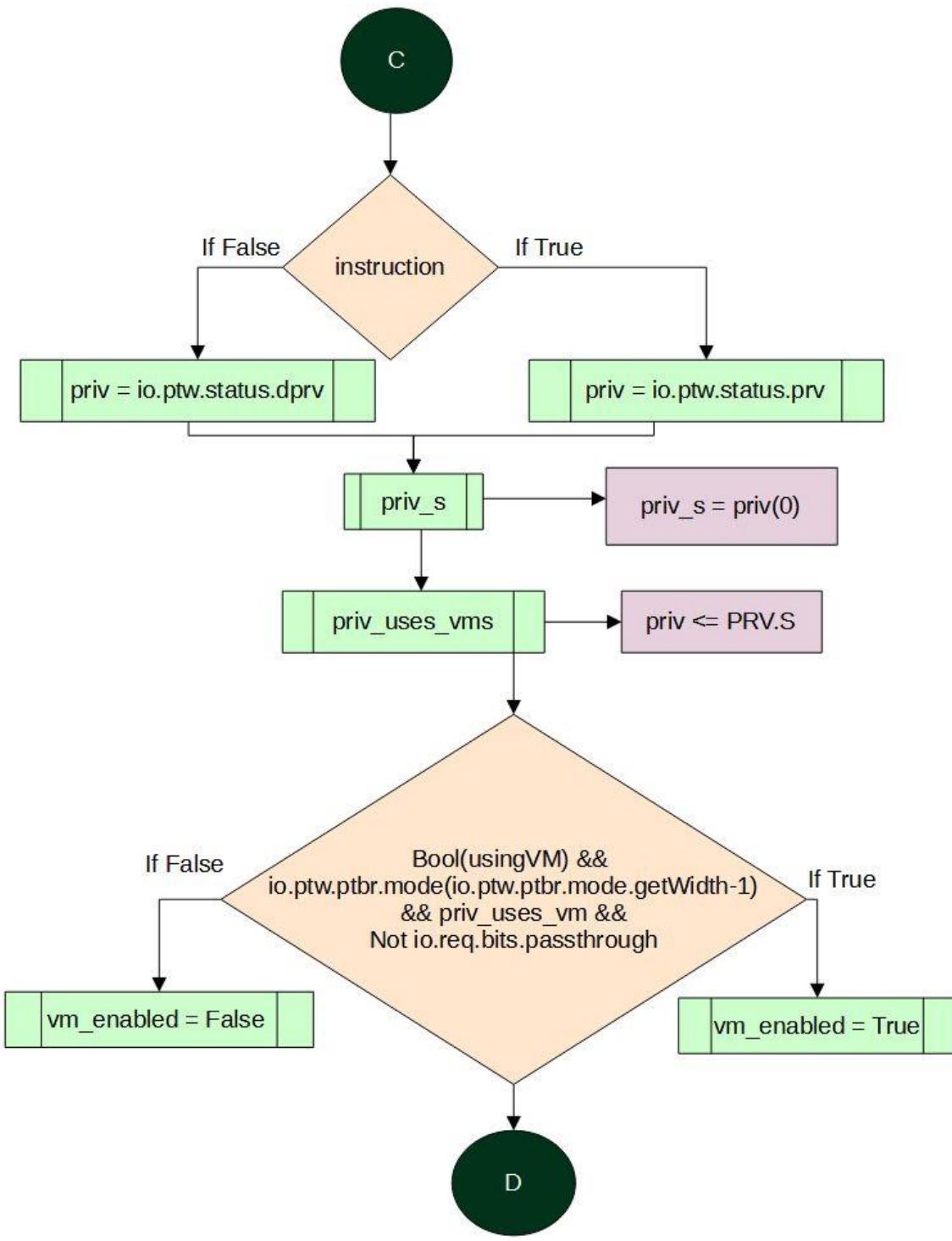
Value of 'priv' is compared to the object of PRV referenced with 'S' which has the value as follows i.e 1

```

object PRV
{
  val SZ = 2
  val U = 0
  val S = 1
  val H = 2
  val M = 3
}

```

The value for 'vm_enabled' is chosen by logical AND of three pre-defined values.



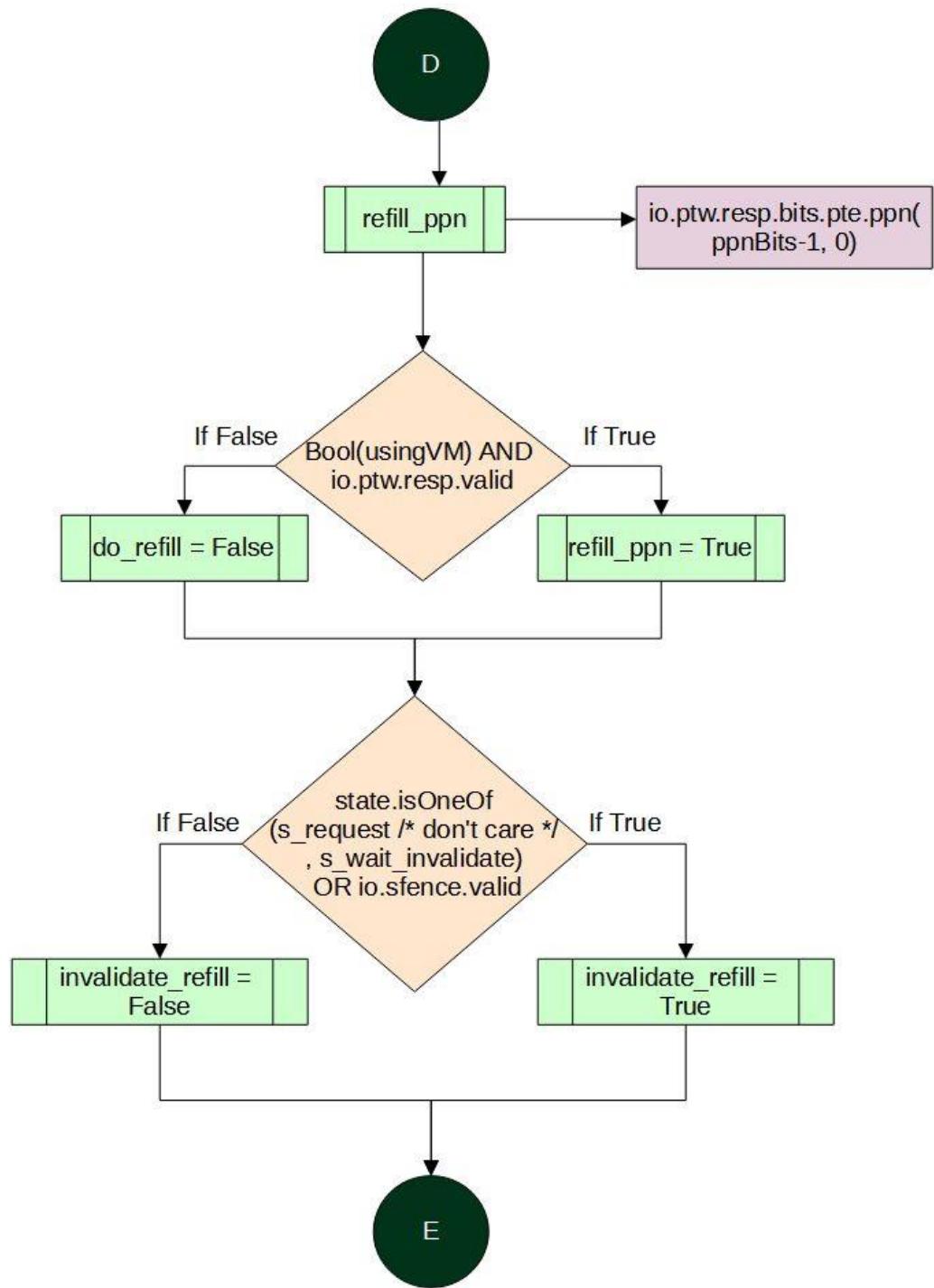
```
// share a single physical memory attribute checker (unshare if critical
path)
val refill_ppn = io.ptw.resp.bits.pte.ppn(ppnBits-1, 0)
val do_refill = Bool(usingVM) && io.ptw.resp.valid
val invalidate_refill = state.isOneOf(s_request /* don't care */,
s_wait_invalidate) || io.sfence.valid
```

— explanation —

Value is assigned to the variable 'refill_ppn' with parameter reference

Value is assigned by logical AND of two pre-defined values

A value is chosen to be assigned by logical OR operation between values



```

val mpu_ppn = Mux(do_refill, refill_ppn,
                    Mux(vm_enabled && special_entry.nonEmpty,
                     special_entry.map(_.ppn(vpn)).getOrElse(0.U), io.req.bits.vaddr >>
                     pgIdxBits))
val mpu_physaddr = Cat(mpu_ppn, io.req.bits.vaddr(pgIdxBits-1, 0))
val mpu_priv = Mux[UInt](Bool(usingVM) && (do_refill ||
io.req.bits.passthrough /* PTW */), PRV.S, Cat(io.ptw.status.debug, priv))

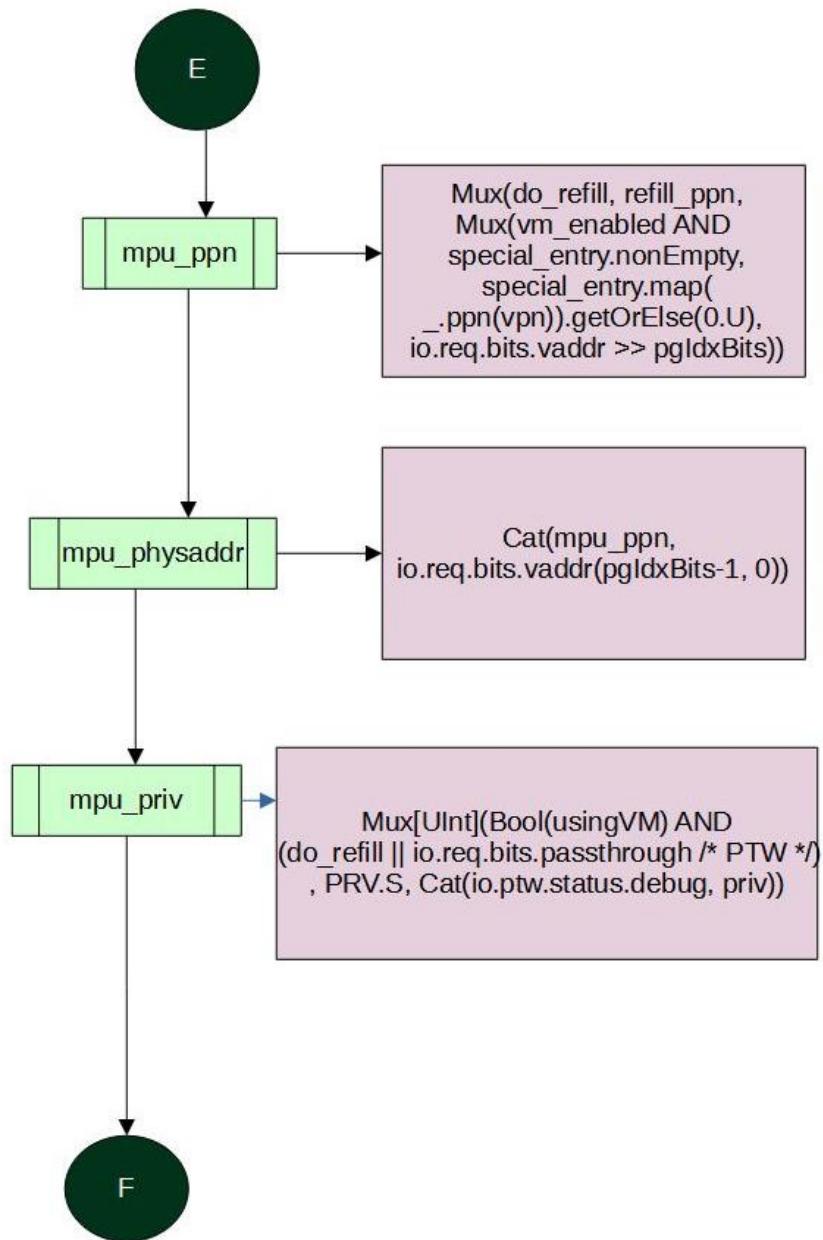
```

— explanation —————

A nested mux has been implemented

The value has been assigned to 'mpu_physaddr' by a concatenation of other values

The value has been assigned to variable via a mux which further has comparisons



```
val pmp = Module(new PMPChecker(lgMaxSize))
pmp.io.addr := mpu_physaddr
pmp.io.size := io.req.bits.size
pmp.io.pmp := (io.ptw.pmp: Seq[PMP])
pmp.io.prv := mpu_priv
```

— explanation —

An object has been created of class and parameter has been passed

'pmp.io.addr has been wired with the value of mpu_physaddr

pmp.io.size has been wired with the value of io.req.bits.size

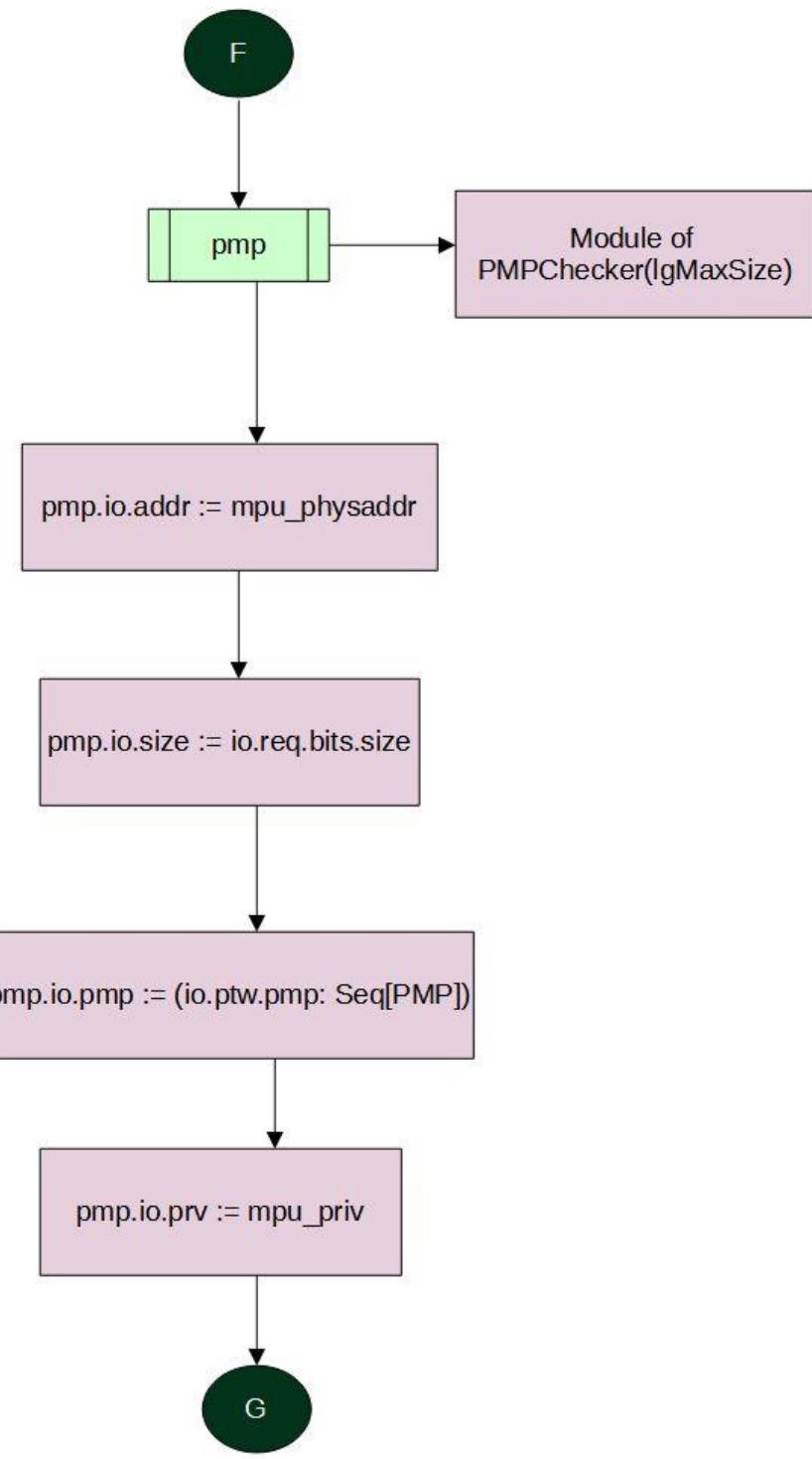
pmp.io.pmp has been wired with the value.

```
val pmp = Vec(nPMPs, new PMP).asOutput
```

:Seq[] = is a strong bulk connect, assigning elements in this Vec from elements in a Seq

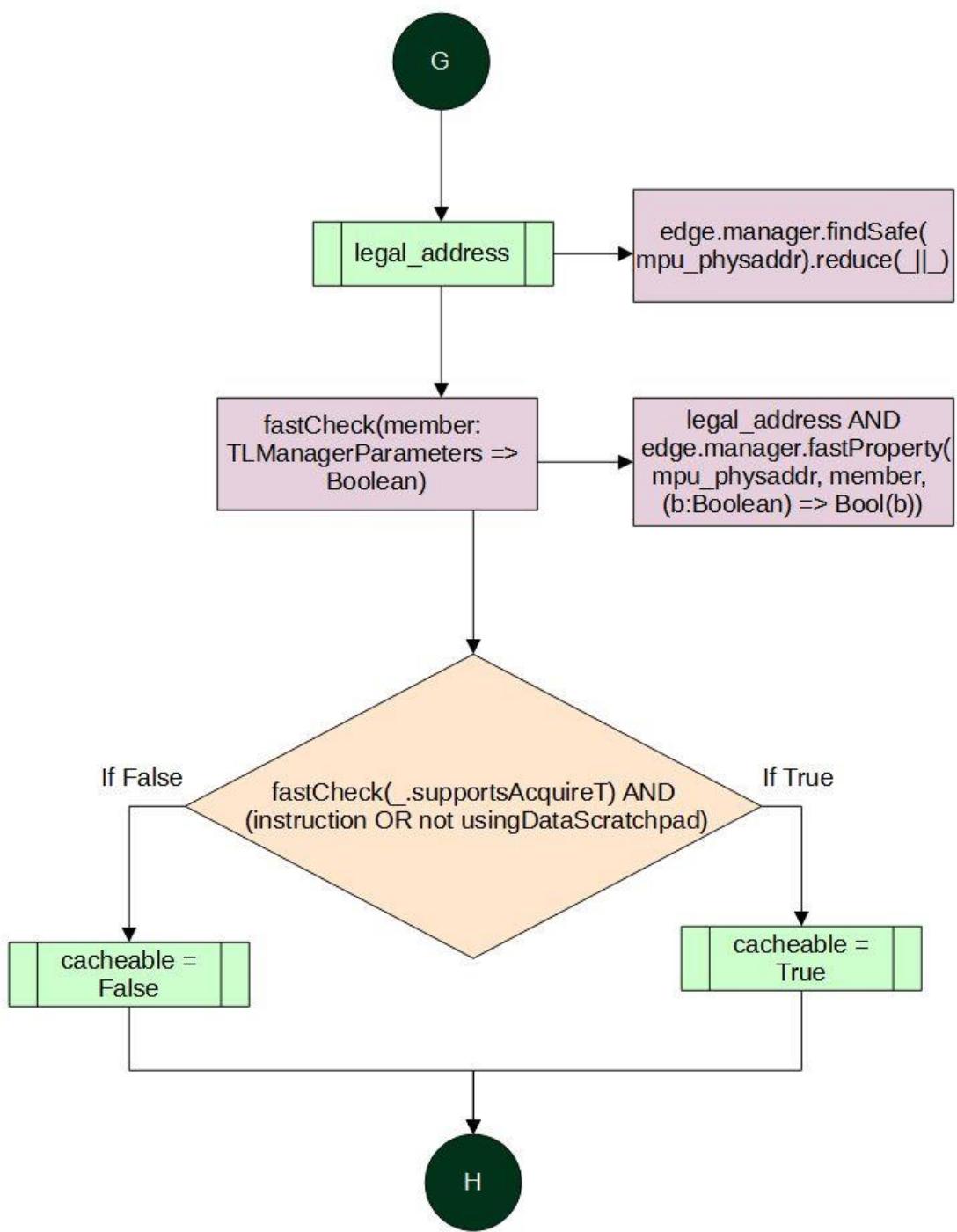
Note: length of this Vec must match the length of the input Seq

pmp.io.prv has been wired with the value mpu_priv



```
val legal_address = edge.manager.findSafe(mpu_physaddr).reduce(_ || _)
def fastCheck(member: TLManagerParameters => Boolean) =
  legal_address && edge.manager.fastProperty(mpu_physaddr, member,
(b:Boolean) => Bool(b))
  val cacheable = fastCheck(_.supportsAcquireT) && (instruction || !usingDataScratchpad)
```

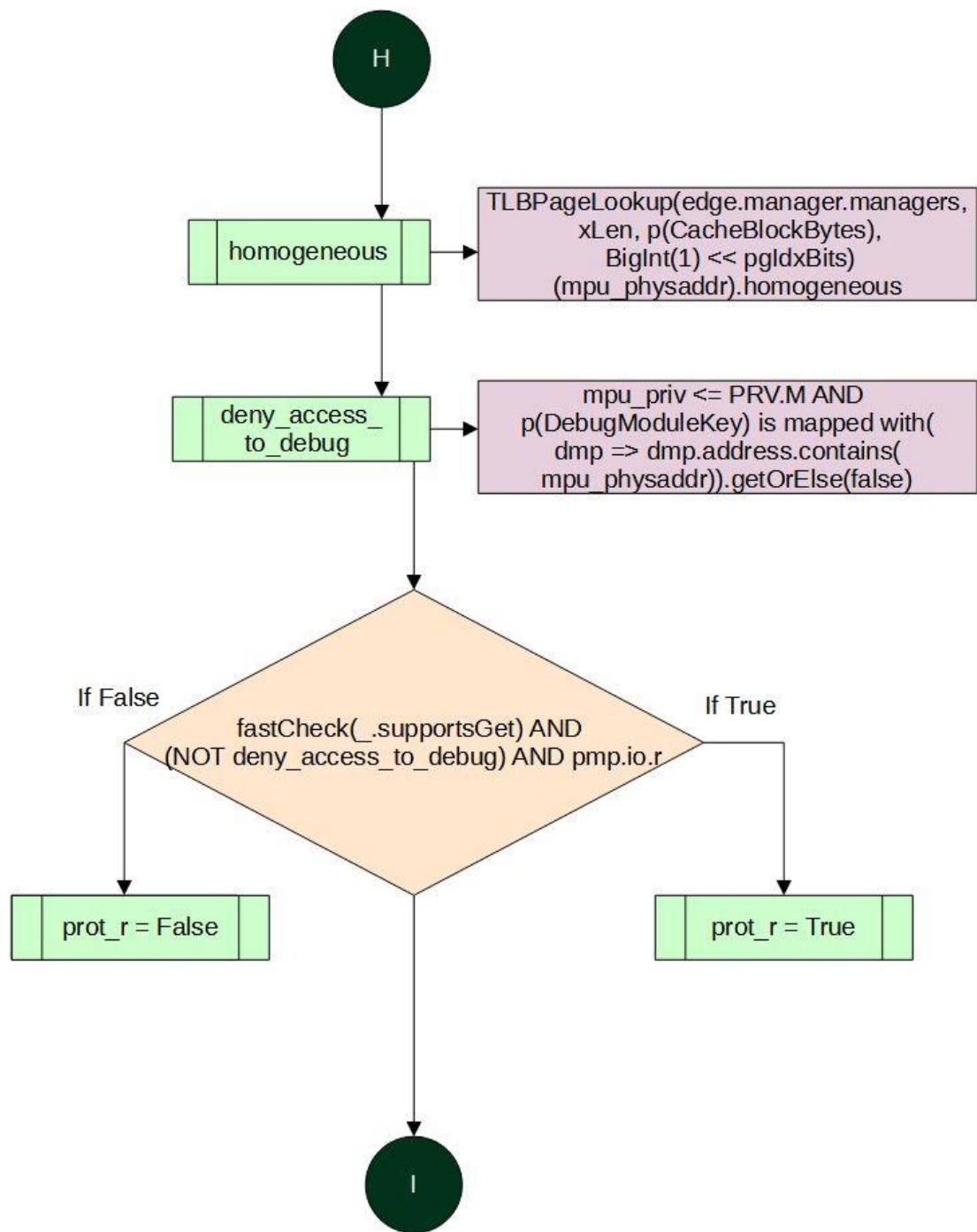
— explanation —



ROCKET-CHIP Micro Architecture Specification Document

```
val homogeneous = TLBPageLookup(edge.manager.managers, xLen,
p(CacheBlockBytes), BigInt(1) << pgIdxBits)(mpu_physaddr).homogeneous
val deny_access_to_debug = mpu_priv <= PRV.M && p(DebugModuleKey).map(dmp
=> dmp.address.contains(mpu_physaddr)).getOrElse(false)
val prot_r = fastCheck(_.supportsGet) && !deny_access_to_debug &&
pmp.io.r
```

— explanation —————



```
val prot_w = fastCheck(_.supportsPutFull) && !deny_access_to_debug &&
pmp.io.w
val prot_pp = fastCheck(_.supportsPutPartial)
val prot_al = fastCheck(_.supportsLogical)
val prot_aa = fastCheck(_.supportsArithmetic)
```

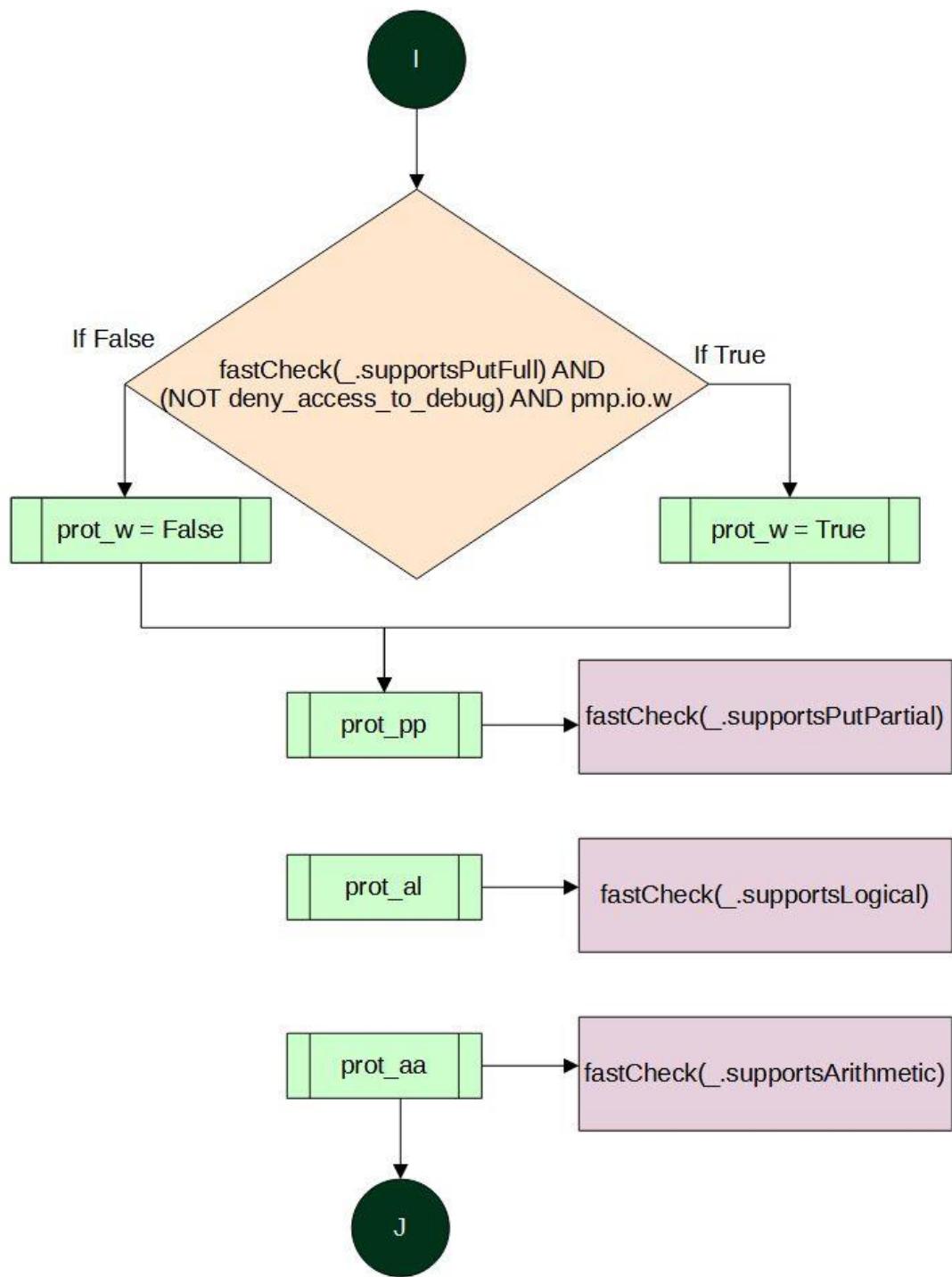
— explanation —

A function has been called with parameters being passed. The output is ANDed with two more values

A function has been called with parameters being passed. Value of it is assigned to val prot_pp

A function has been called with parameters being passed. Value of it is assigned to val prot_al

A function has been called with parameters being passed. Value of it is assigned to val prot_aa



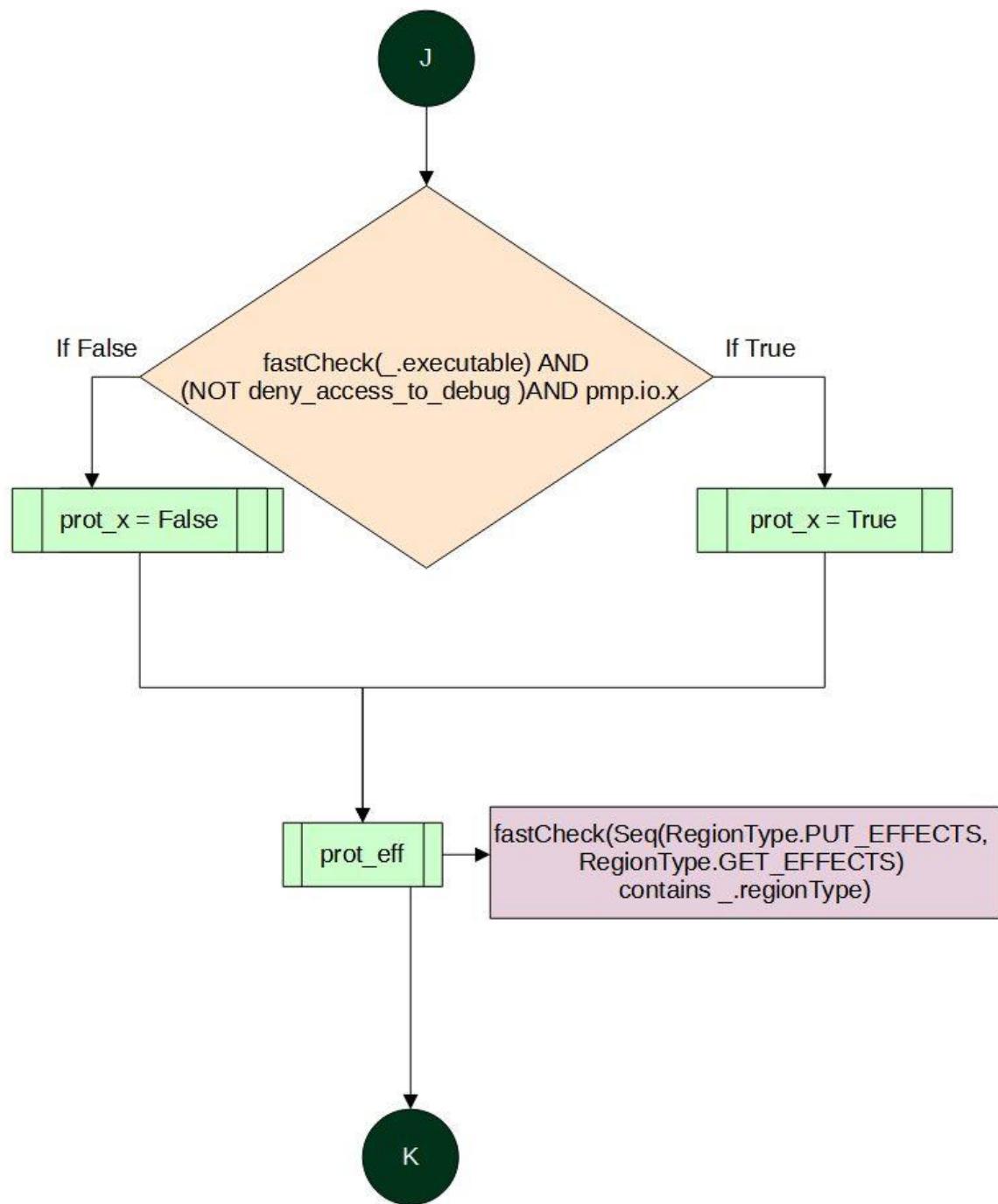
```
val prot_x = fastCheck(_.executable) && !deny_access_to_debug && pmp.io.x
val prot_eff = fastCheck(Seq(RegionType.PUT_EFFECTS,
RegionType.GET_EFFECTS) contains _.regionType)
```

— explanation —

A function has been called and parameters has been passed, the output is ANDed with two other values

```
def fastCheck(member: TLManagerParameters => Boolean) =
  legal_address && edge.manager.fastProperty(mu_physaddr, member,
(b:Boolean) => Bool(b))
  val cacheable = fastCheck(_.supportsAcquireT) && (instruction ||
!usingDataScratchpad)
  val homogeneous = TLBPageLookup(edge.manager.managers, xLen,
p(CacheBlockBytes), BigInt(1) << pgIdxBits)(mu_physaddr).homogeneous
  val deny_access_to_debug = mu_priv <= PRV.M && p(DebugModuleKey).map(dmp
=> dmp.address.contains(mu_physaddr)).getOrElse(false)
  val prot_r = fastCheck(_.supportsGet) && !deny_access_to_debug &&
pmp.io.r
  val prot_w = fastCheck(_.supportsPutFull) && !deny_access_to_debug &&
pmp.io.w
  val prot_pp = fastCheck(_.supportsPutPartial)
```

A function has been called and parameters has been passed, the output is ANDed with two other values. An addition eq() function has also been used.



```
val sector_hits = sectored_entries(memIdx).map(_.sectorHit(vpn))
val superpage_hits = superpage_entries.map(_.hit(vpn))
val hitsVec = all_entries.map(vm_enabled && _.hit(vpn))
val real_hits = hitsVec.asUInt
val hits = Cat(!vm_enabled, real_hits)
```

— explanation —

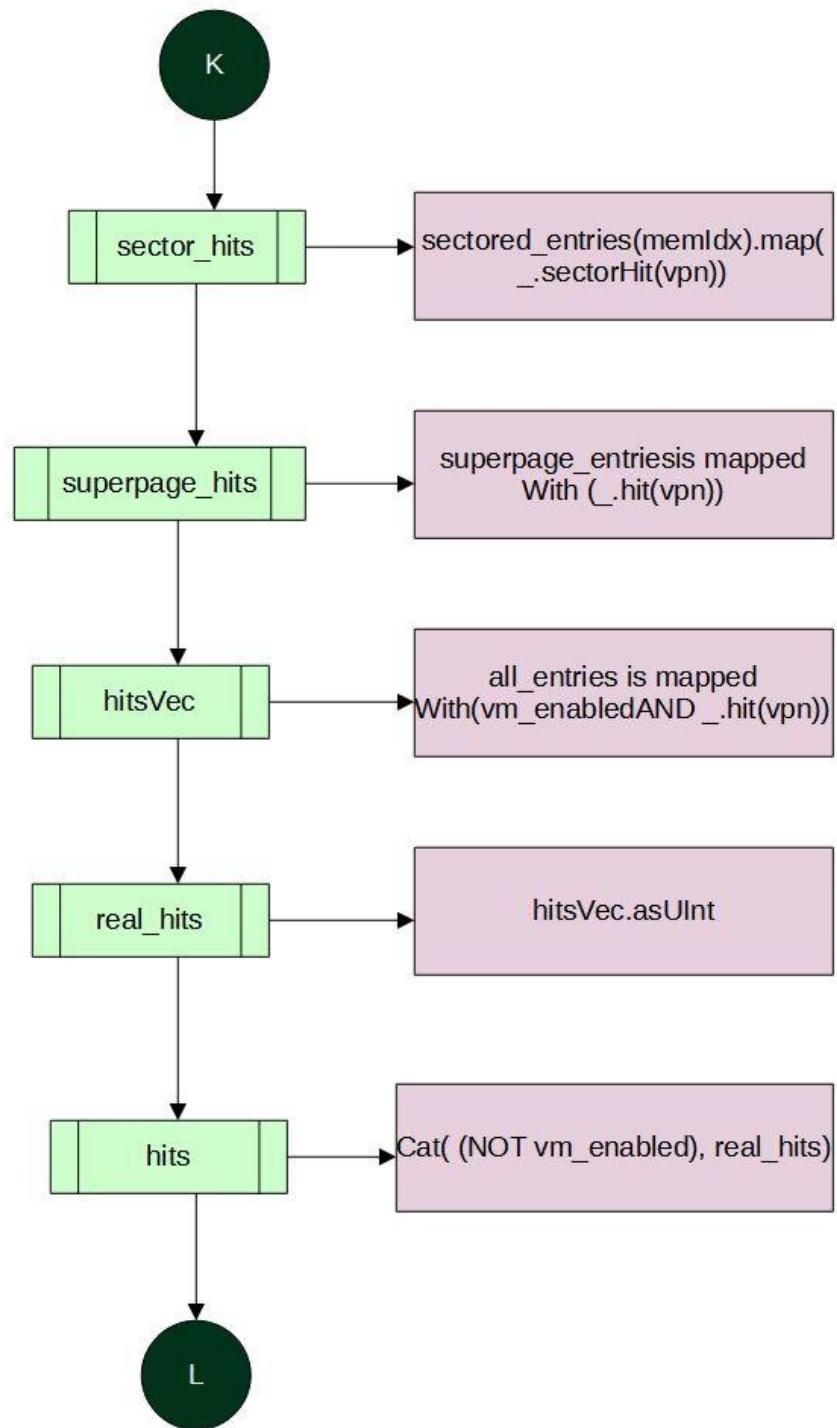
two methods have been called with parameters referenced where a method maps a value to the other

superpage_entries map its value to a function `def hit = {}` with parameter 'vpn'

all_entries map its value to a function `def hit = {}` with parameter 'vpn' AND ed with vm_enabled

real_hits has been assigned a value

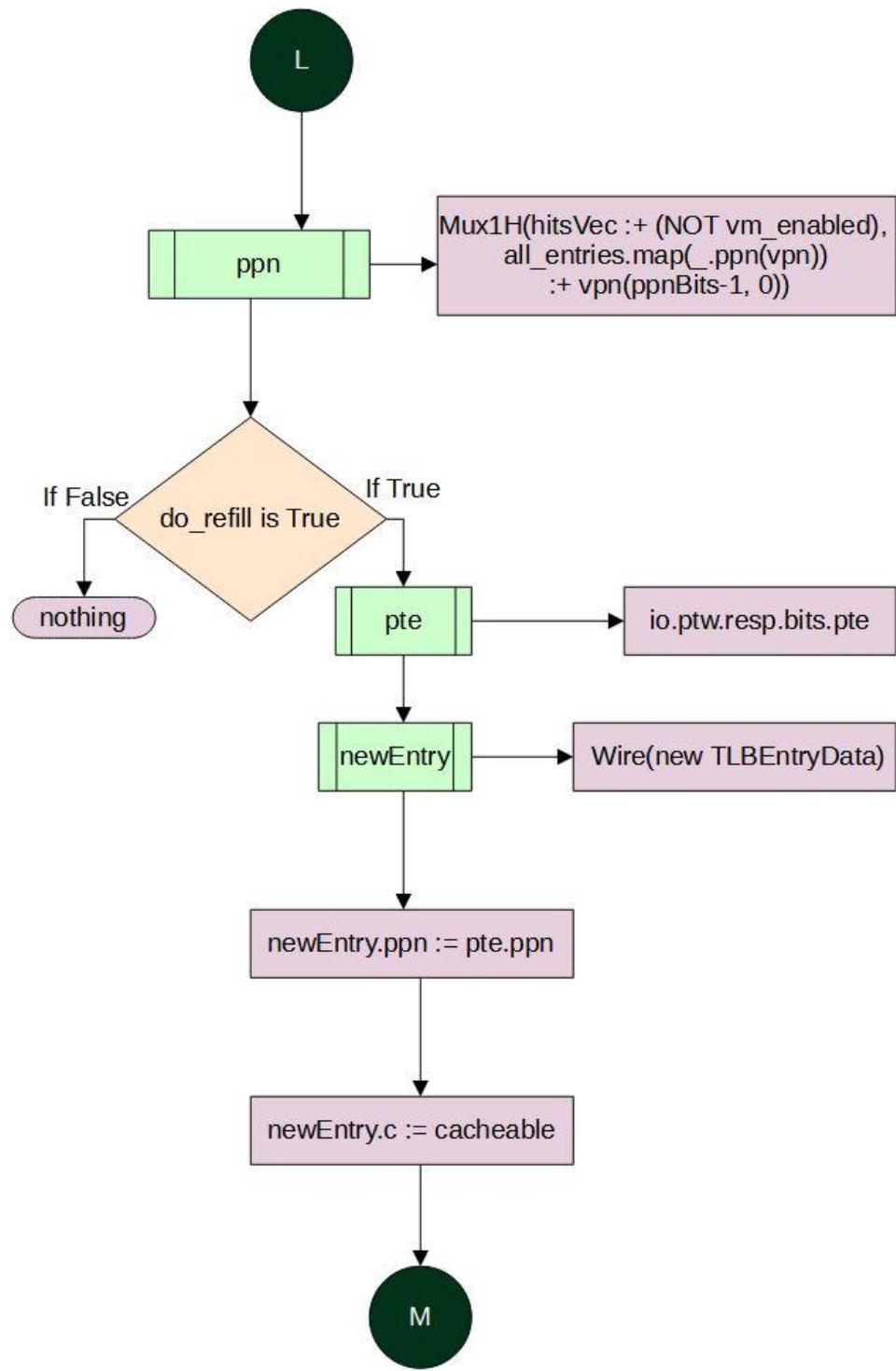
value has been assigned to 'hits' which is a concatenation of two values with the negation of the first



```
val ppn = Mux1H(hitsVec :+ !vm_enabled, all_entries.map(_.ppn(vpn)) :+  
vpn(ppnBits-1, 0))  
  
// permission bit arrays  
when (do_refill) {  
    val pte = io.ptw.resp.bits.pte  
    val newEntry = Wire(new TLBEntryData)  
    newEntry.ppn := pte.ppn  
    newEntry.c := cacheable
```

— explanation —

values has been assigned to pte and newEntry where newEntry.ppn and newEntry.c are wired to new values only if 'do_refill' is true



```
newEntry.u := pte.u
newEntry.g := pte.g && pte.v
newEntry.ae := io.ptw.resp.bits.ae
newEntry.sr := pte.sr()
newEntry.sw := pte.sw()
newEntry.sx := pte.sx()
newEntry.pr := prot_r
```

— explanation —

newEntry.u has been wired to the value of pte.u

newEntry.g has been wired to the value of AND of pte.g and pte.v

newEntry.ae has been wired to the value of io.ptw.resp.bits.ae

newEntry.sr has been wired to the function of pte.sr()

val sr = Bool()

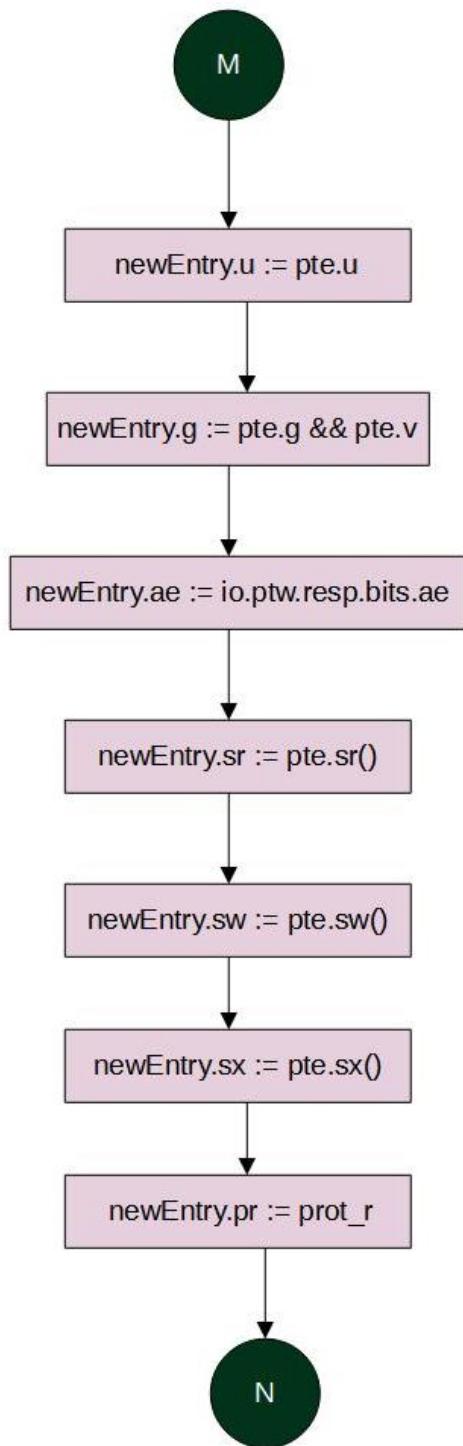
newEntry.sw has been wired to the function of pte.sw()

val sw = Bool()

newEntry.sx has been wired to the funstion of pte.sx()

val sx = Bool()

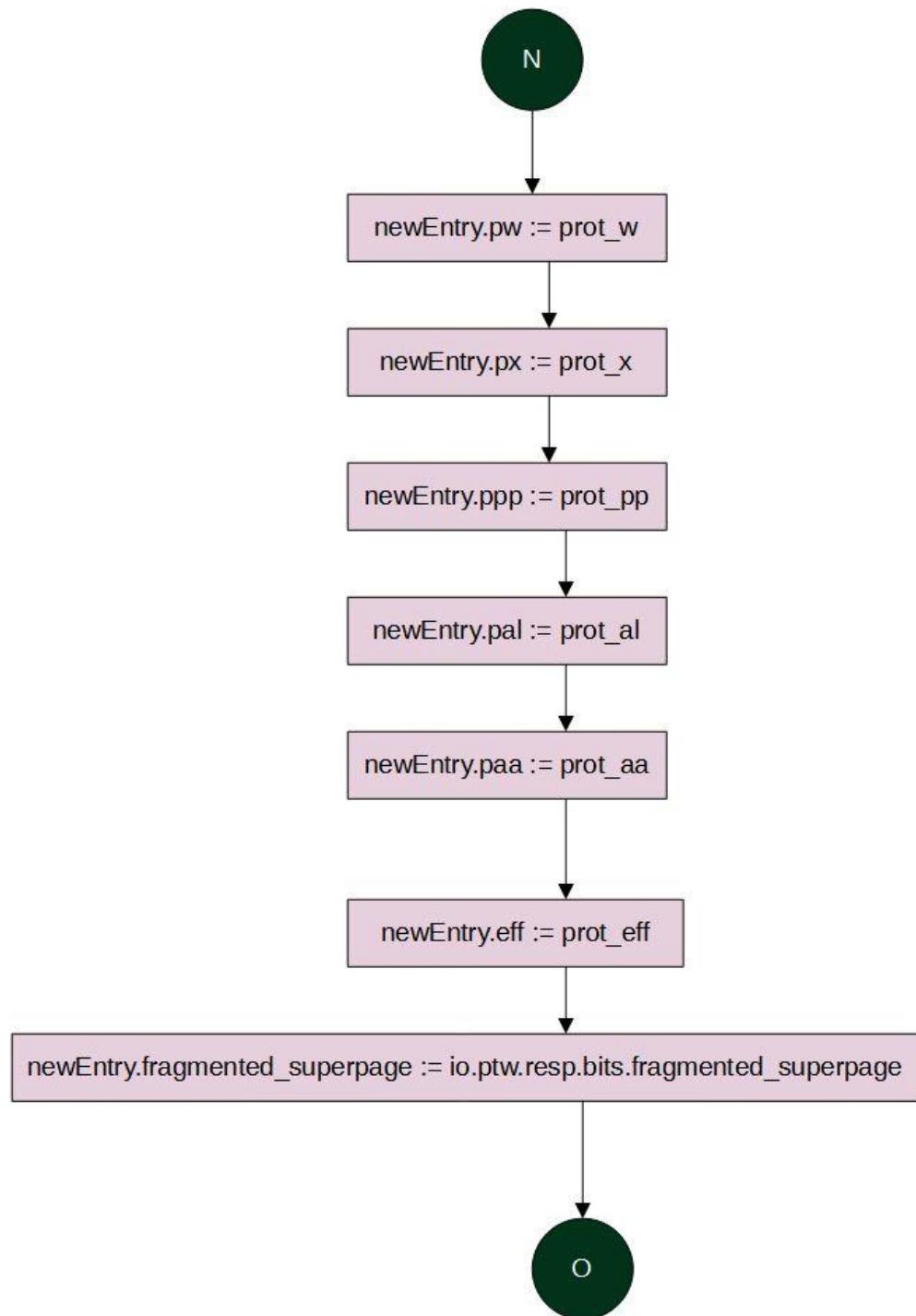
newEntry.pr has been wired to the value of prot_r



```
newEntry.pw := prot_w  
newEntry.px := prot_x  
newEntry.ppp := prot_pp  
newEntry.pal := prot_al  
newEntry.paa := prot_aa  
newEntry.eff := prot_eff  
newEntry.fragmented_superpage := io.ptw.resp.bits.fragmented_superpage
```

— explanation —

newEntry.pw is wired to the value of prot_w
newEntry.px is wired to the value of prot_x
newEntry.ppp is wired to the value of prot_pp
newEntry.pal is wired to the value of prot_al
newEntry.paa is wired to the value of prot_aa
newEntry.eff is wired to the value of prot_eff
newEntry.fragmented_superpage is wired to the value of
io.ptw.resp.bits.fragmented_superpage



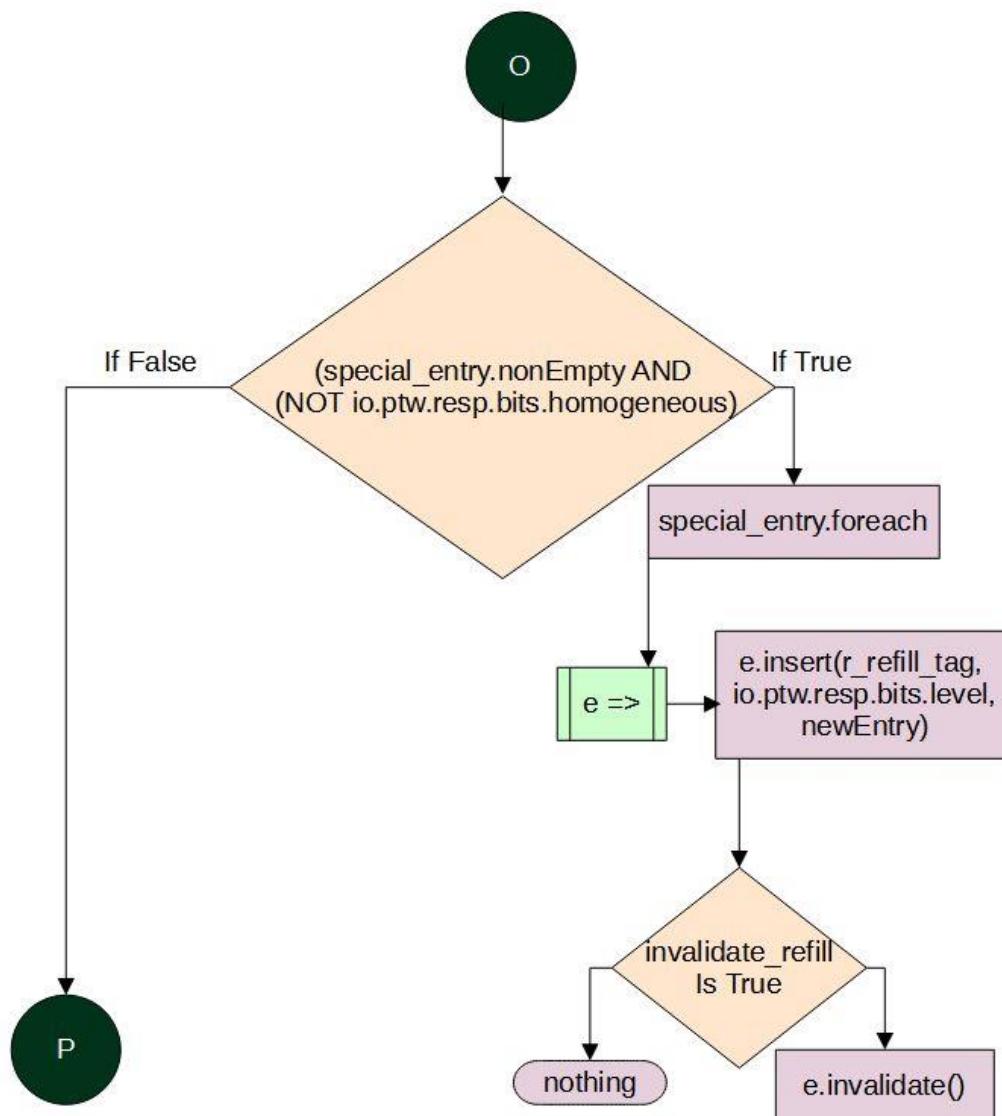
```

when (special_entry.nonEmpty && !io.ptw.resp.bits.homogeneous) {
    special_entry.foreach { e =>
        e.insert(r_refill_tag, io.ptw.resp.bits.level, newEntry)
        when (invalidate_refill) { e.invalidate() }
    }
}

```

explanation

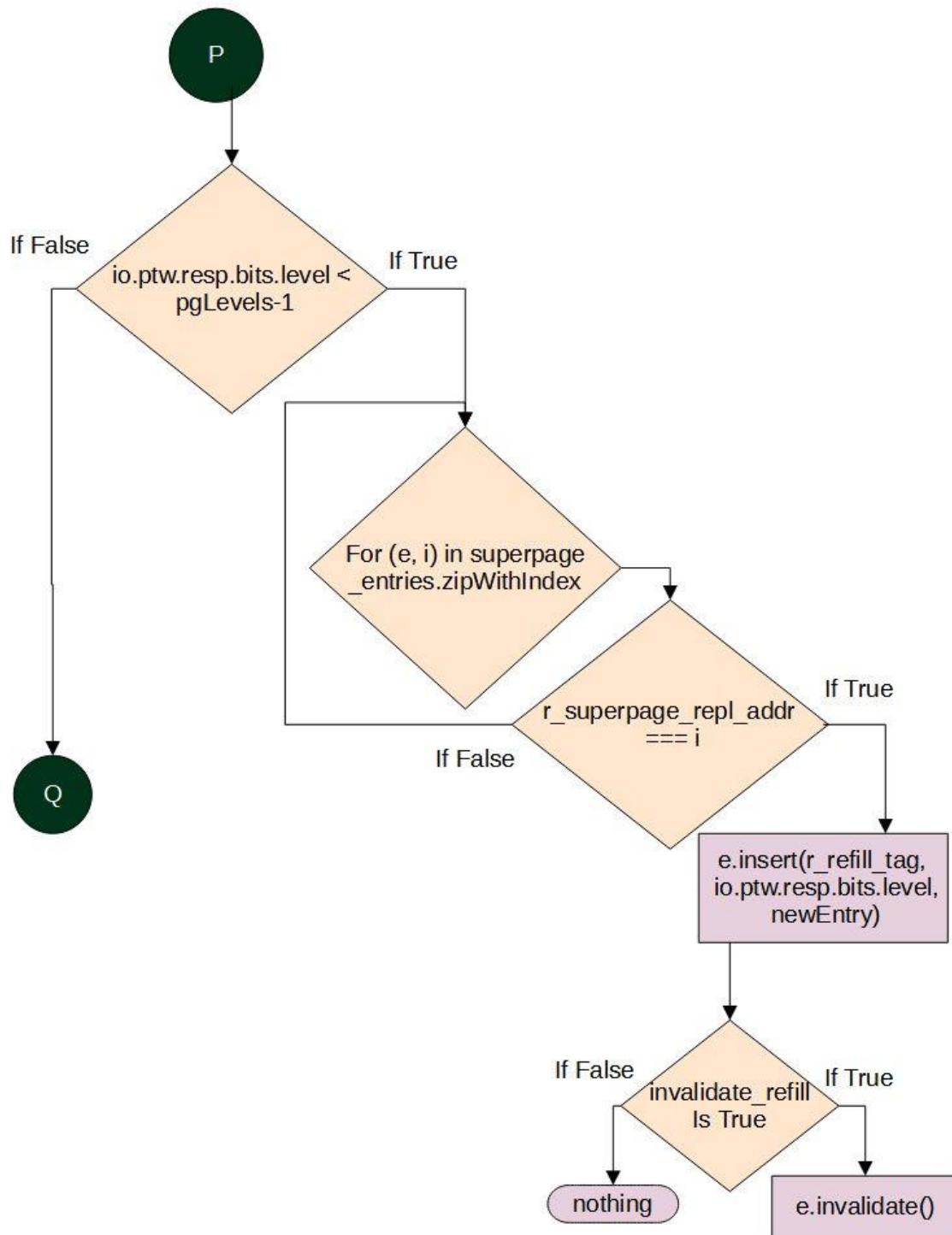
Nested when conditions are implied which modifies the value of 'e'



```
.elsewhen (io.ptw.resp.bits.level < pgLevels-1) {
    for ((e, i) <- superpage_entries.zipWithIndex) when
(r_superpage_repl_addr === i) {
    e.insert(r_refill_tag, io.ptw.resp.bits.level, newEntry)
    when (invalidate_refill) { e.invalidate() }
}
```

— explanation —

An elsewhen condition has been initialized that if it satisfies, a loop starts which further has nested conditions all of which assign value to 'e'



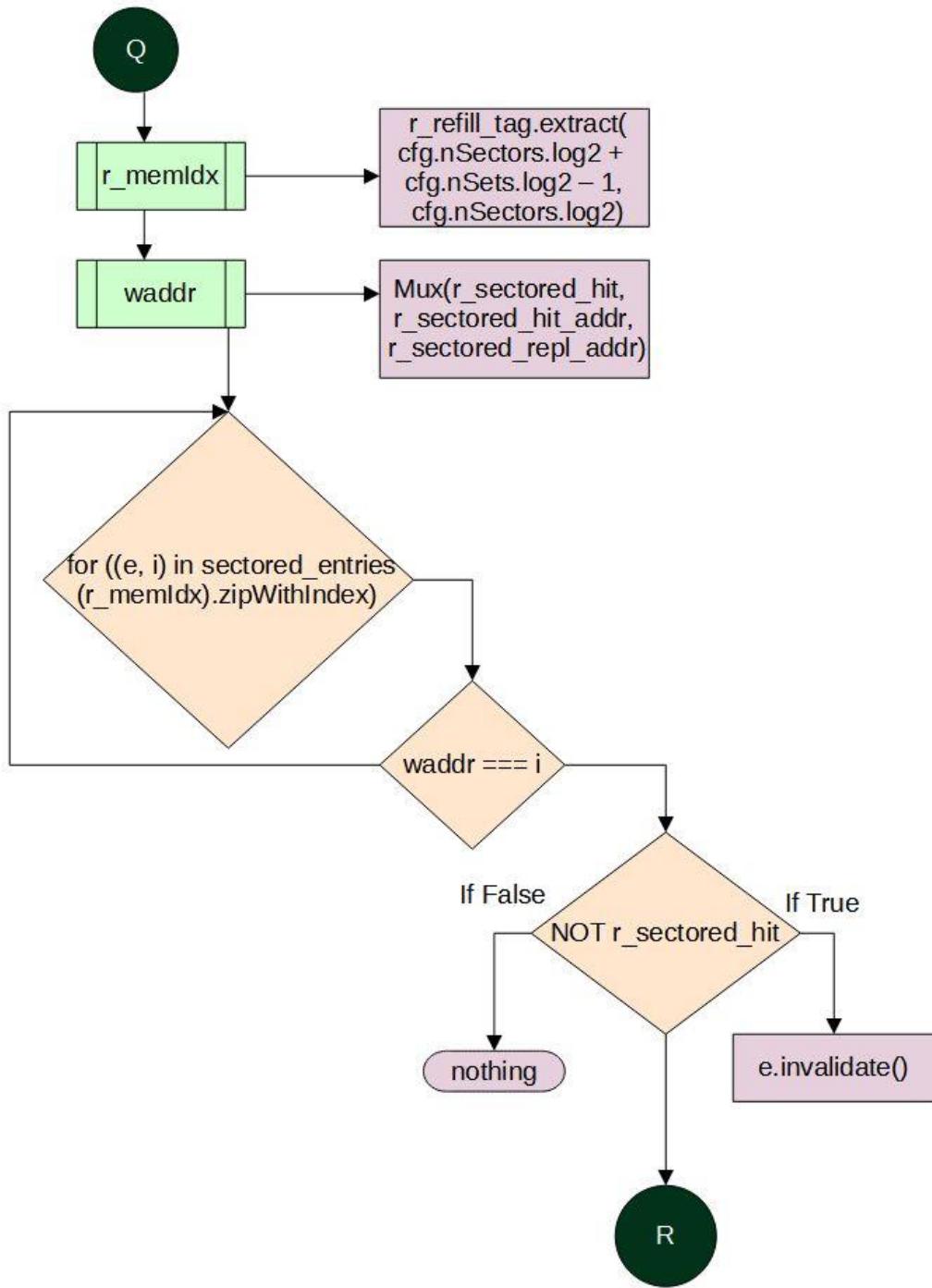
```
}.otherwise {
    val r_memIdx = r_refill_tag.extract(cfg.nSectors.log2 +
cfg.nSets.log2 - 1, cfg.nSectors.log2)
    val waddr = Mux(r_sectored_hit, r_sectored_hit_addr,
r_sectored_repl_addr)
    for ((e, i) <- sectored_entries(r_memIdx).zipWithIndex) when (waddr ==
== i) {
        when (!r_sectored_hit) { e.invalidate() }
```

explanation

a value have been assigned where the extract function is used to extract data and provide it to the inner route

value for 'waddr' is selected via a mux

A loop has been started under which a nested condition has been initialized which when satisfies calls a function invalidate() referenced with 'e'

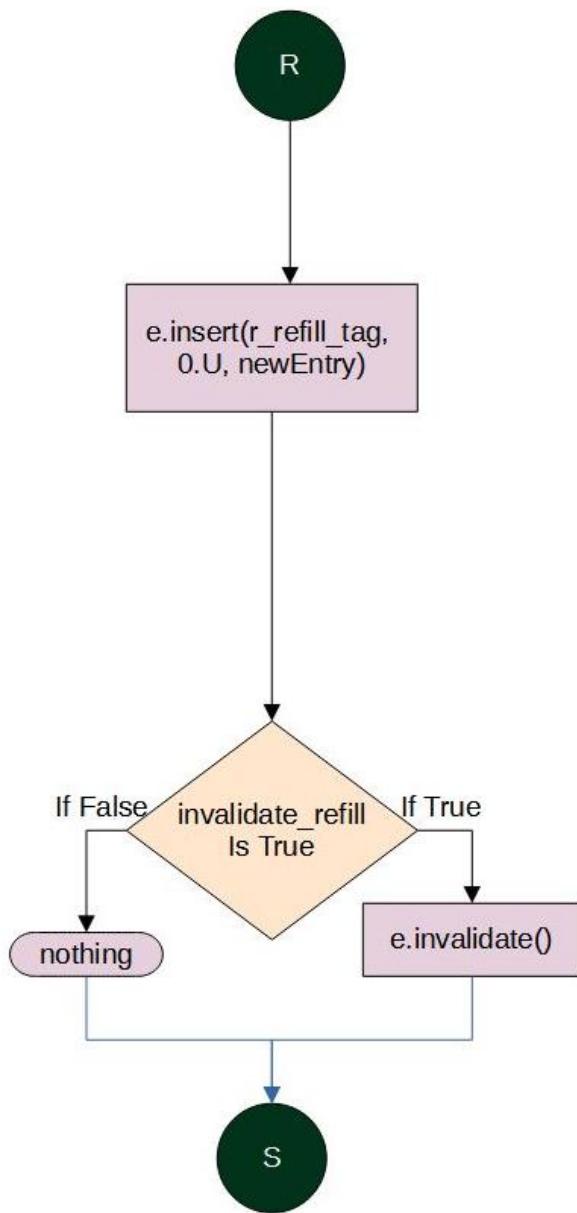


```
    e.insert(r_refill_tag, 0.U, newEntry)
    when (invalidate_refill) { e.invalidate() }
}
}
```

explanation

a value has been inserted in 'e'

a condition has been implied that if 'invalidate_refill' is true, a function named as invalidate() has been called with reference to 'e'



```

val entries = all_entries.map(_.getData(vpn))
val normal_entries = ordinary_entries.map(_.getData(vpn))
val nPhysicalEntries = 1 + special_entry.size
val ptw_ae_array = Cat(false.B, entries.map(_.ae).asUInt)
val priv_rw_ok = Mux(!priv_s || io.ptw.status.sum,
entries.map(_.u).asUInt, 0.U) | Mux(priv_s, ~entries.map(_.u).asUInt, 0.U)
val priv_x_ok = Mux(priv_s, ~entries.map(_.u).asUInt,
entries.map(_.u).asUInt)

```

explanation

A method 'getData' is being called and a variable 'all_entries' is being mapped to it.

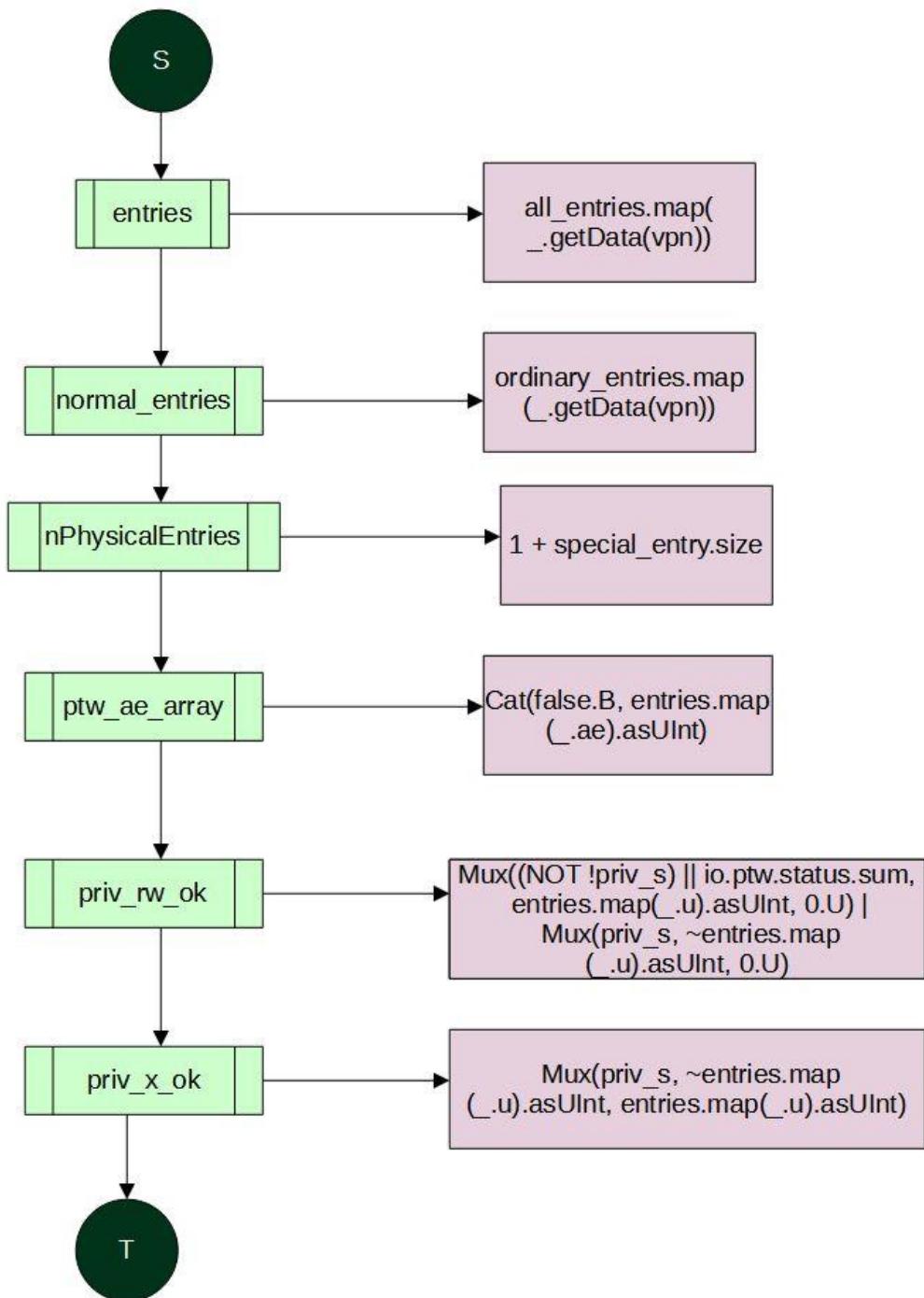
A method 'getData' is being called and a variable 'all_entries' is being mapped to it.

A mathematical operation is being performed and 1 is added to assign a value

A concatenated value is being allotted where methods are being called under the concatenation operation

An ORed output of two mux has been assigned as a value

The value is being assigned to the variable via a mux



```
val r_array = Cat(true.B, priv_rw_ok & (entries.map(_.sr).asUInt |  
Mux(io.ptw.status.mxr, entries.map(_.sx).asUInt, UInt(0))))  
val w_array = Cat(true.B, priv_rw_ok & entries.map(_.sw).asUInt)  
val x_array = Cat(true.B, priv_x_ok & entries.map(_.sx).asUInt)  
val pr_array = Cat(Fill(nPhysicalEntries, prot_r),  
normal_entries.map(_.pr).asUInt) & ~ptw_ae_array  
val pw_array = Cat(Fill(nPhysicalEntries, prot_w),  
normal_entries.map(_.pw).asUInt) & ~ptw_ae_array
```

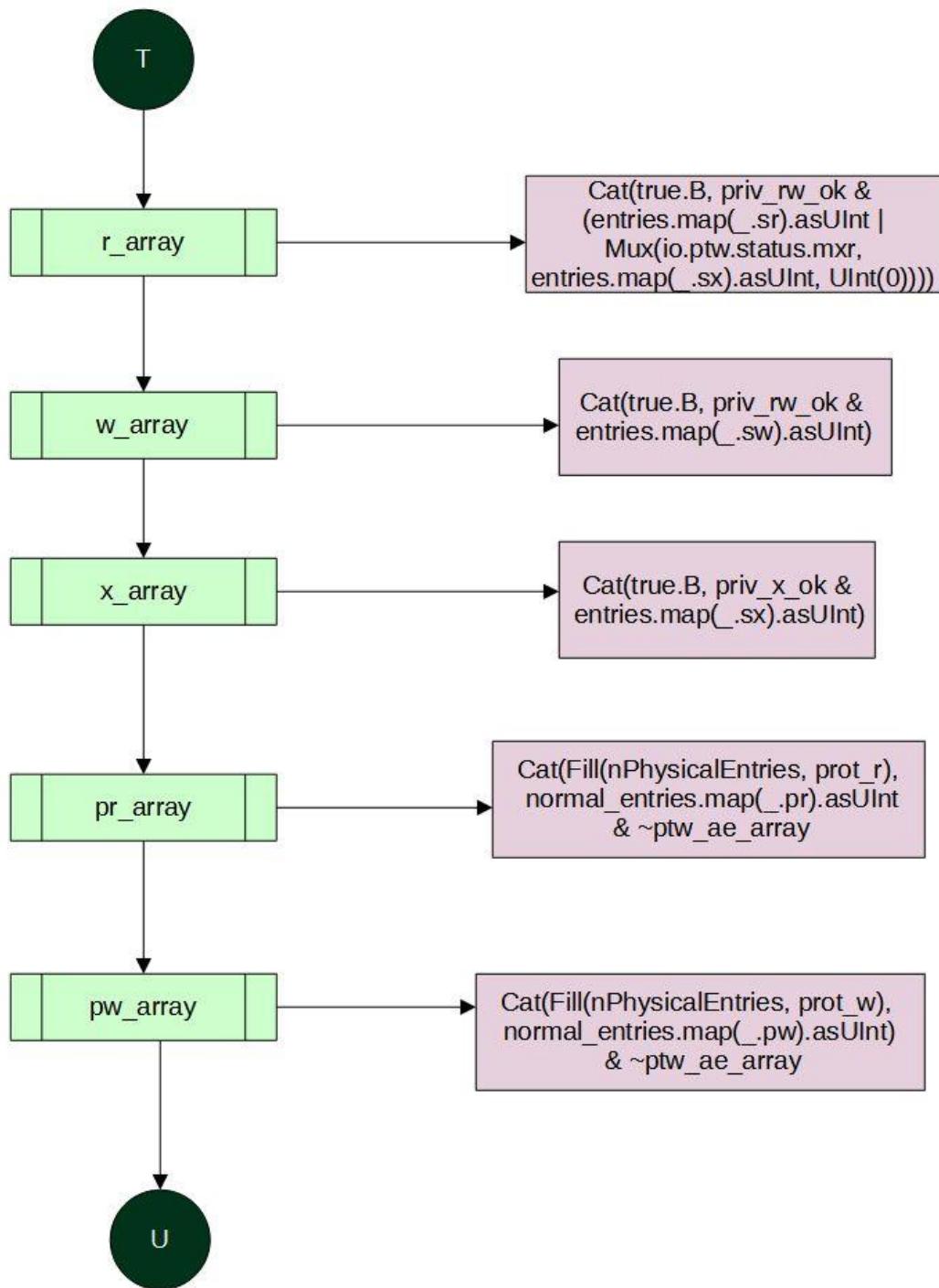
— explanation —

A concatenated output that is being ORed with a MUX is being allotted as value

A concatenated output of values ANDed earlier is being allotted

A concatenated output of values is being allotted which is ANDed later

A concatenated output of values is being allotted which is ANDed later

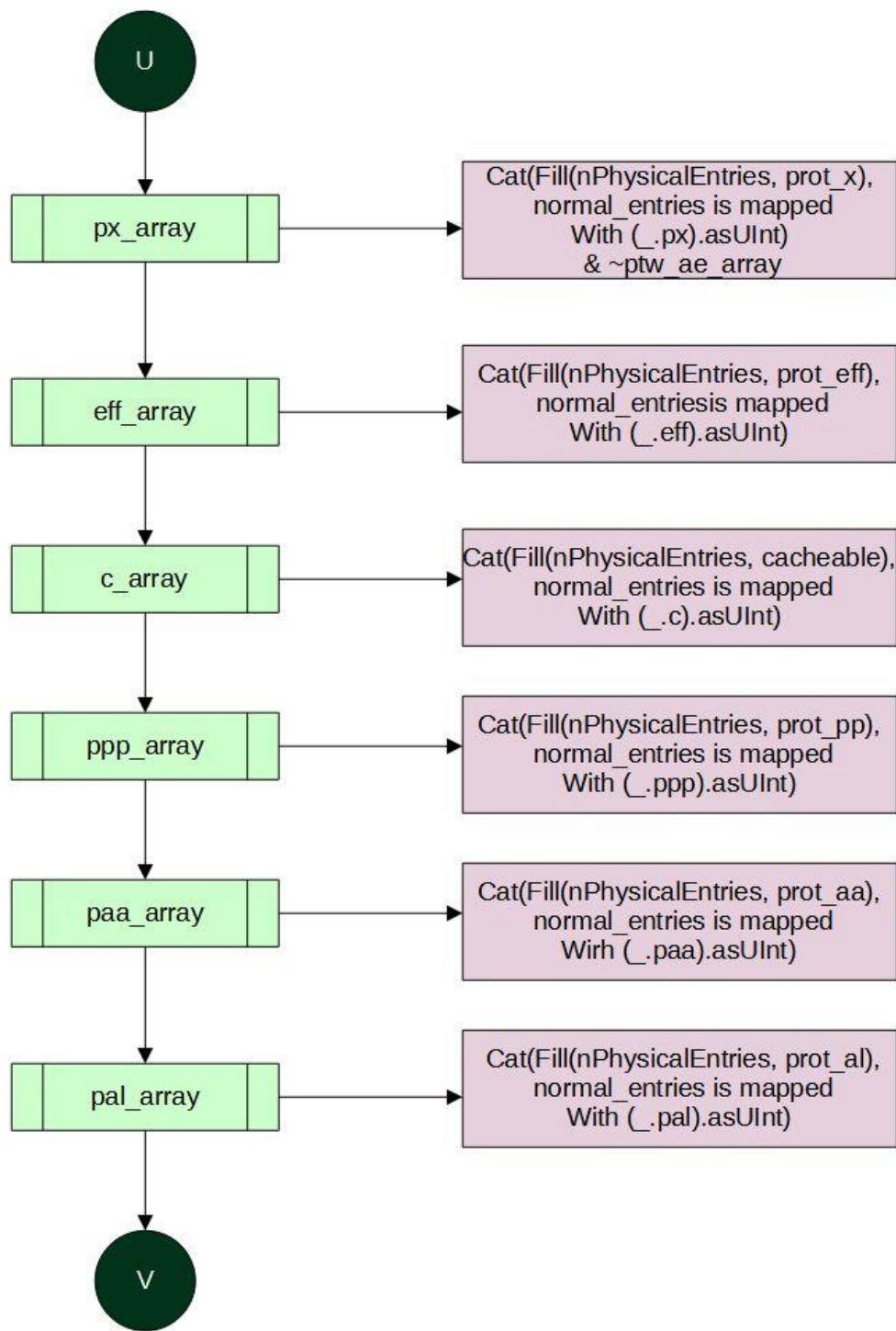


```
val px_array = Cat(Fill(nPhysicalEntries, prot_x),  
normal_entries.map(_.px).asUInt) & ~ptw_ae_array  
val eff_array = Cat(Fill(nPhysicalEntries, prot_eff),  
normal_entries.map(_.eff).asUInt)  
val c_array = Cat(Fill(nPhysicalEntries, cacheable),  
normal_entries.map(_.c).asUInt)  
val ppp_array = Cat(Fill(nPhysicalEntries, prot_pp),  
normal_entries.map(_.ppp).asUInt)  
val paa_array = Cat(Fill(nPhysicalEntries, prot_aa),  
normal_entries.map(_.paa).asUInt)  
val pal_array = Cat(Fill(nPhysicalEntries, prot_al),  
normal_entries.map(_.pal).asUInt)
```

— explanation —

The concatenated output ANDed with a negated value is being allotted

A concatenated output of values is being allotted



```
val ppp_array_if_cached = ppp_array | c_array
val paa_array_if_cached = paa_array | Mux(usingAtomicsInCache, c_array,
0.U)
val pal_array_if_cached = pal_array | Mux(usingAtomicsInCache, c_array,
0.U)
val prefetchable_array = Cat((cacheable && homogeneous) <<
(nPhysicalEntries-1), normal_entries.map(_.c).asUInt)
```

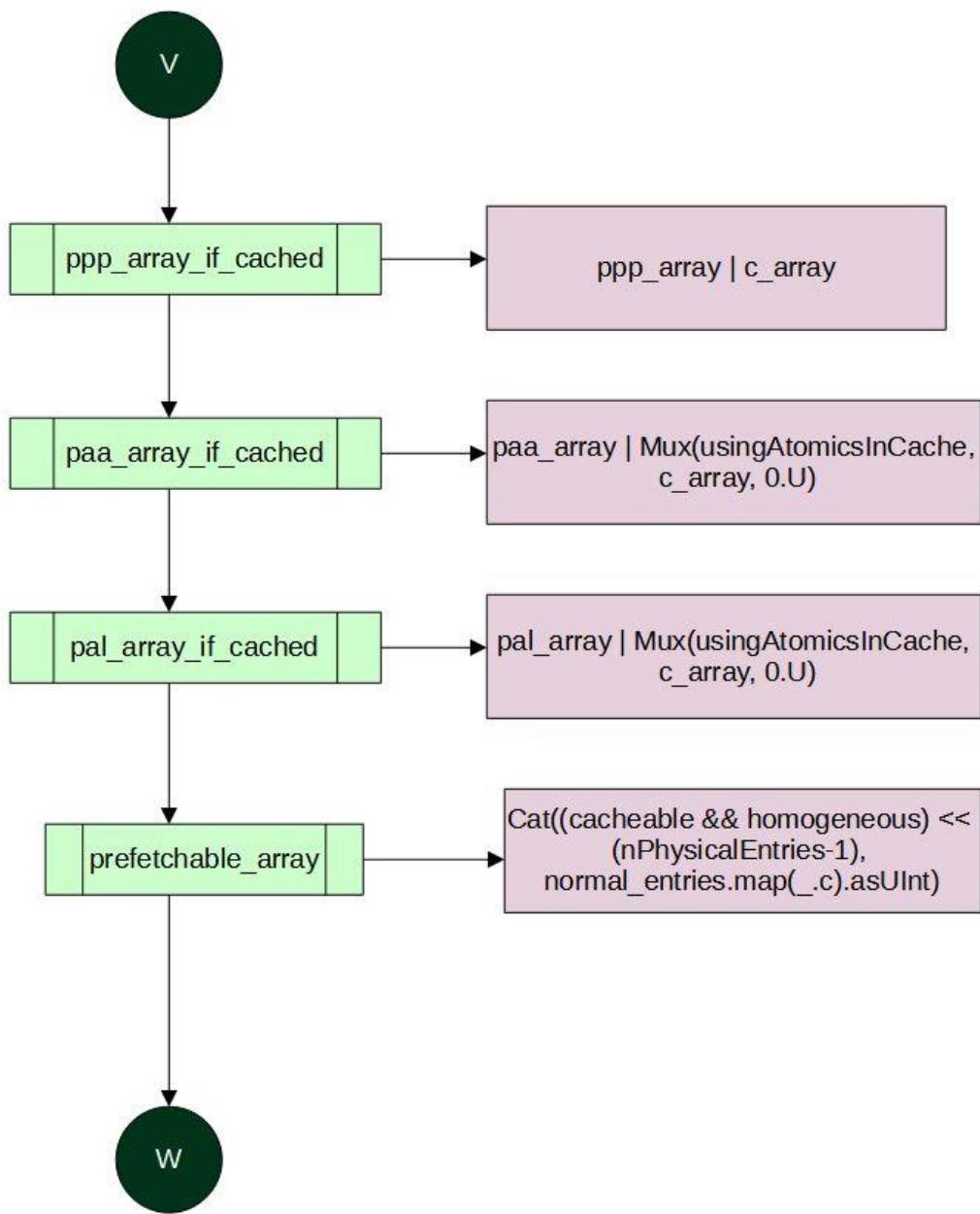
— explanation —

an ORed output of two values is being assigned

the ORed output of a mux and a value is being assigned

the ORed output of a mux and a value is being assigned

The concatenated output where comparison is being performed is allotted as a value

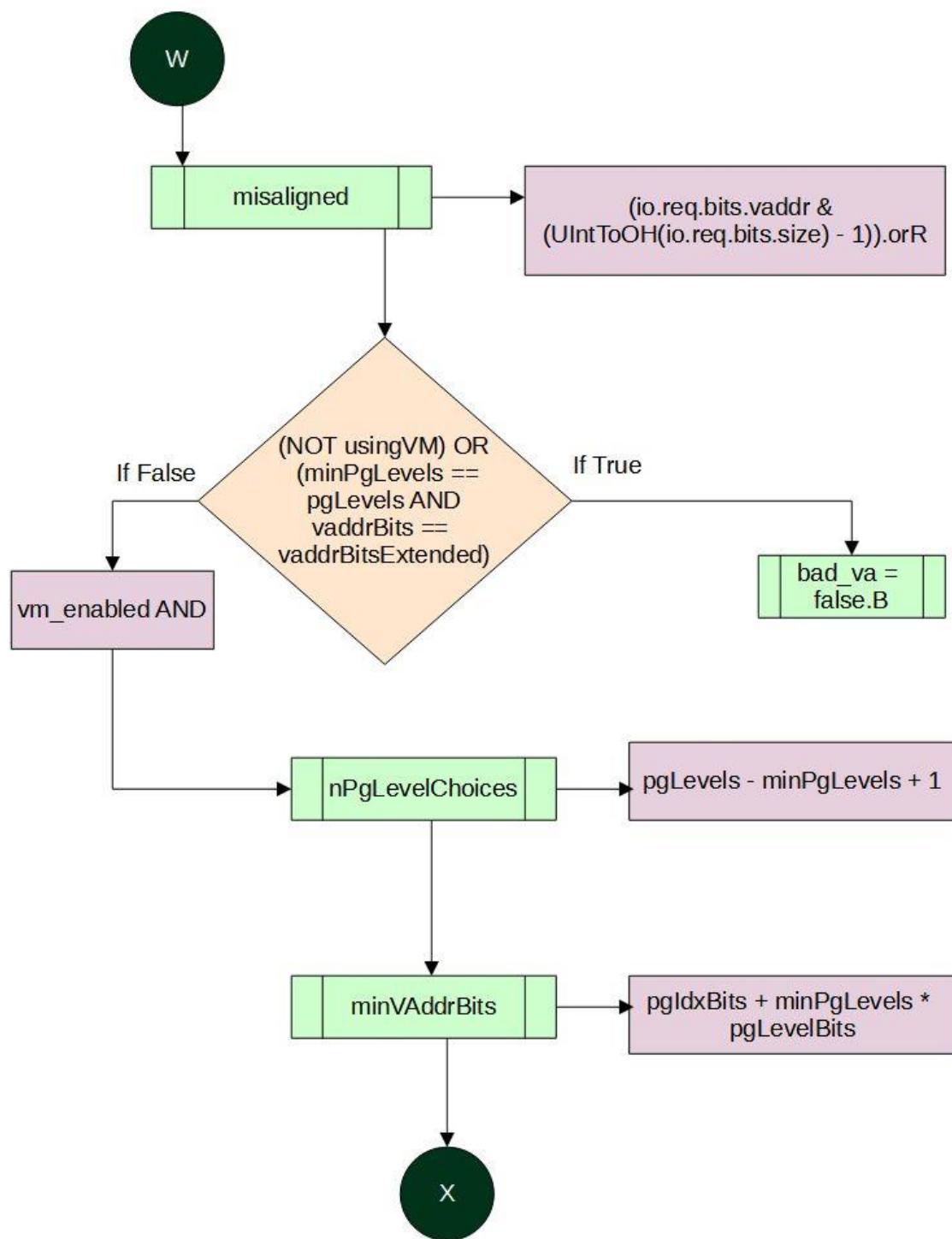


```
val misaligned = (io.req.bits.vaddr & (UIntToOH(io.req.bits.size) - 1)).orR
  val bad_va = if (!usingVM || (minPgLevels == pgLevels && vaddrBits ==
vaddrBitsExtended)) false.B else vm_enabled && {
    val nPgLevelChoices = pgLevels - minPgLevels + 1
    val minVAddrBits = pgIdxBits + minPgLevels * pgLevelBits
```

— explanation —

Value has been assigned by an AND operation between a value and a method passed with parameter. The entire output is being ORed

A condition has been initialized that the value of bad_va would be true if the logical operation is satisfied and in that case values has been allotted to variable 'nPgLevelChoice' and 'minVAddrBits'



```

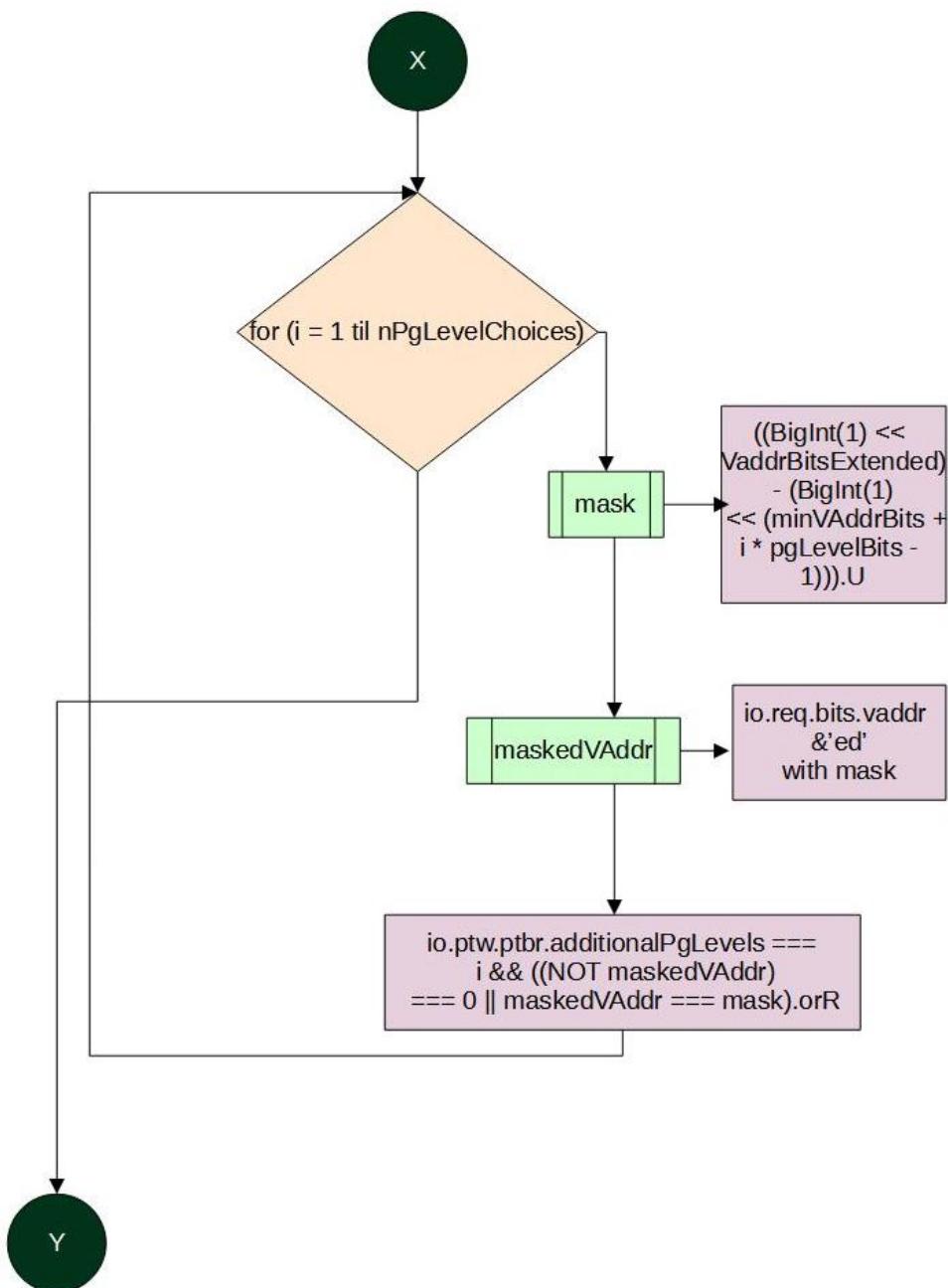
(for (i <- 0 until nPgLevelChoices) yield {
    val mask = ((BigInt(1) << vaddrBitsExtended) - (BigInt(1) <<
(minVAddrBits + i * pgLevelBits - 1))).U
    val maskedVAddr = io.req.bits.vaddr & mask
    io.ptw.ptbr.additionalPgLevels === i && !(maskedVAddr === 0 || 
maskedVAddr === mask)
}).orR
}

```

 explanation

A loop has been initialized comparing the value of i

Value has been assigned by logical and mathematical operations being performed and methods have called with parameters passed



```
val cmd_lrsc = Bool(usingAtomics) && io.req.bits.cmd.isOneOf(M_XLR,  
M_XSC)  
val cmd_amo_logical = Bool(usingAtomics) && isAMOLogical(io.req.bits.cmd)  
val cmd_amo_arithmetic = Bool(usingAtomics) &&  
isAMOArithmetic(io.req.bits.cmd)  
val cmd_put_partial = io.req.bits.cmd === M_PWR  
val cmd_read = isRead(io.req.bits.cmd)  
val cmd_write = isWrite(io.req.bits.cmd)
```

— explanation —

value has been assigned by an AND operation

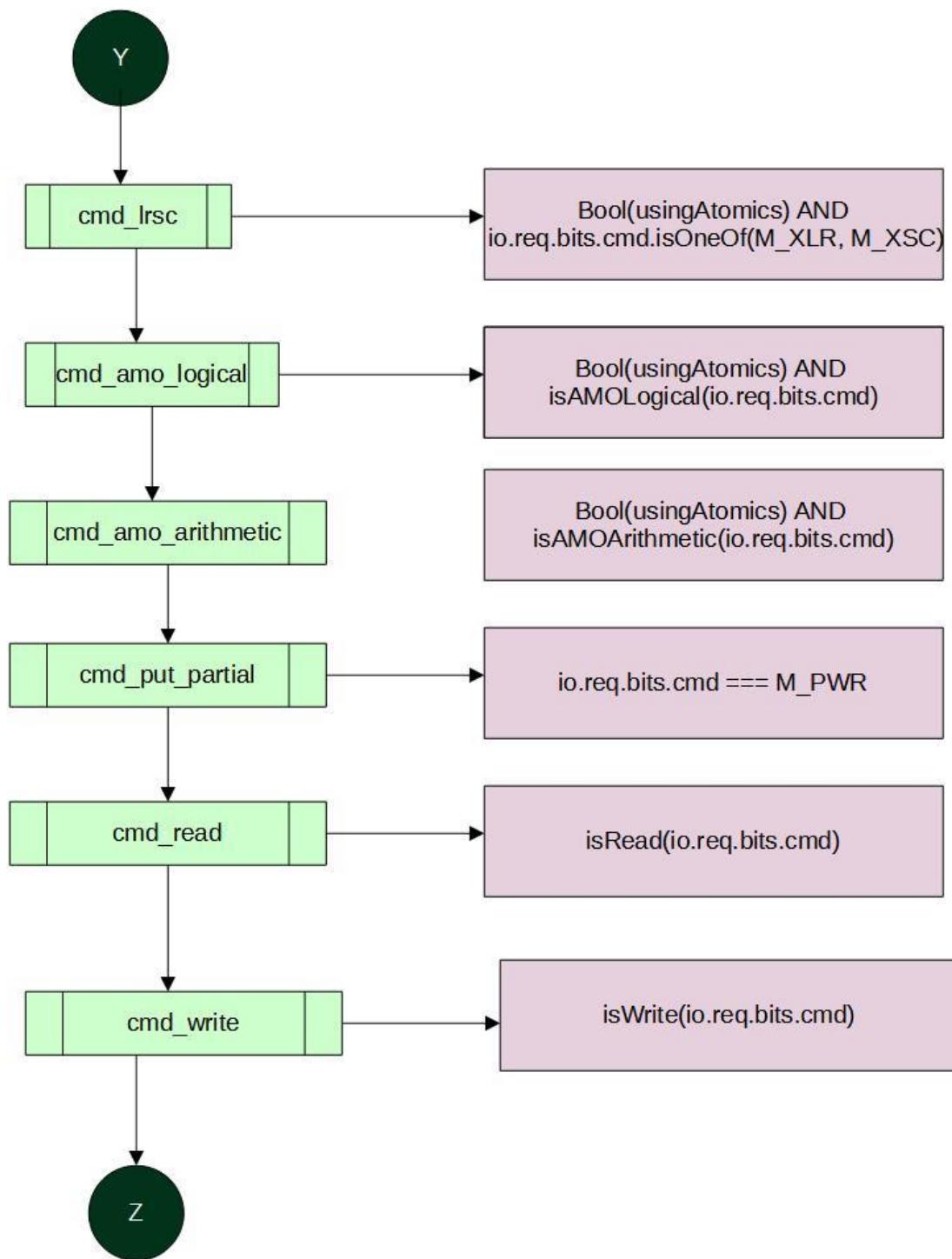
value has been assigned by an AND operation

value has been assigned by an AND operation

value has been assigned by an equalization comparison

value has been assigned by calling a method and parameter has been passed

value has been assigned by calling a method and parameter has been passed



```

val cmd_write_perms = cmd_write ||
    io.req.bits.cmd.isOneOf(M_FLUSH_ALL, M_WOK) // not a write, but needs
write permissions

val lrscAllowed = Mux(Bool(usingDataScratchpad || usingAtomicsOnlyForIO),
0.U, c_array)
val ae_array =
    Mux(misaligned, eff_array, 0.U) |
    Mux(cmd_lrsc, ~lrscAllowed, 0.U)
val ae_id_array = Mux(cmd_read, ae_array | ~pr_array, 0.U)

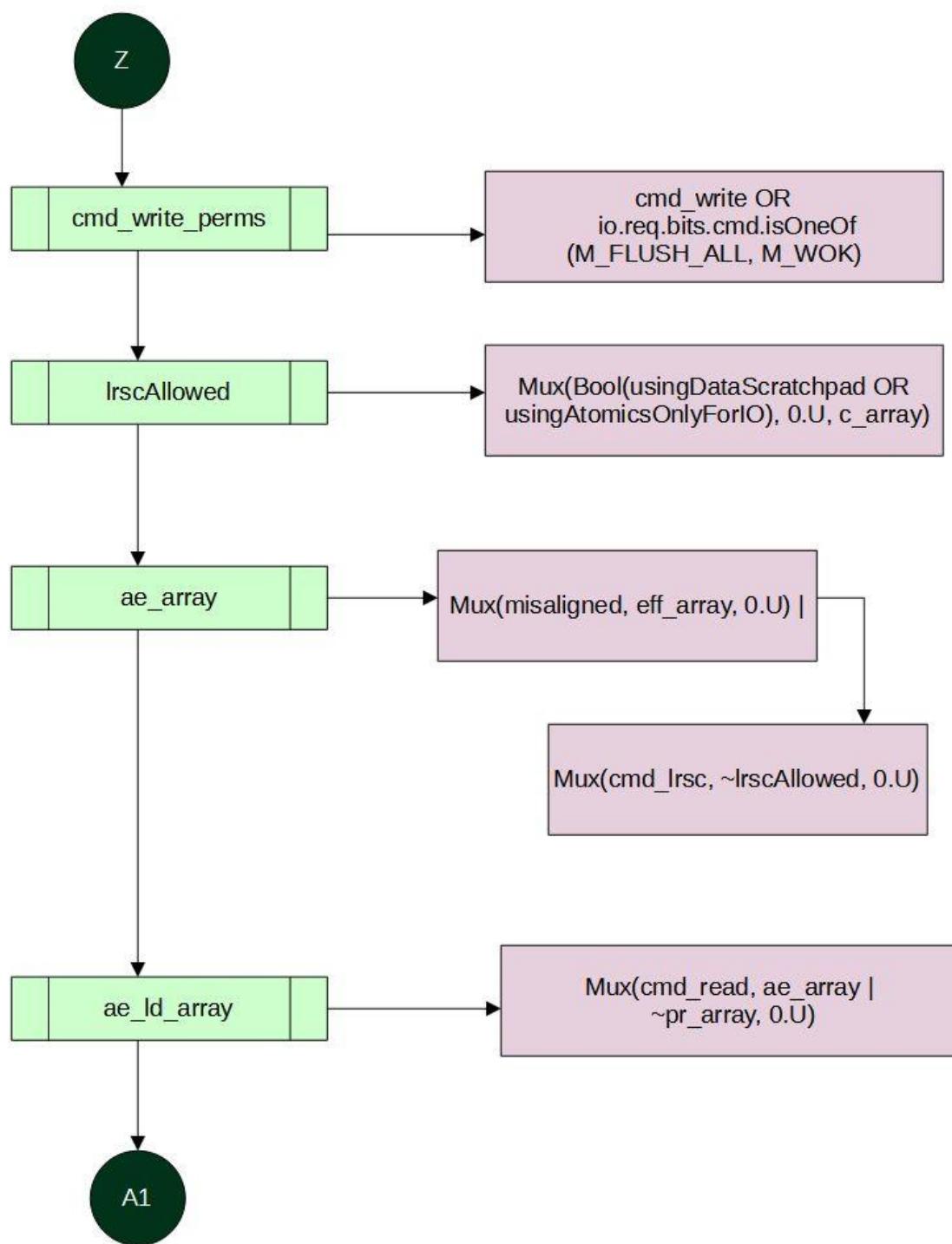
```

— explanation —

value has been assigned to variable lrscAllowed via a mux

value has been assigned to ae_array though an OR operation between two MUX

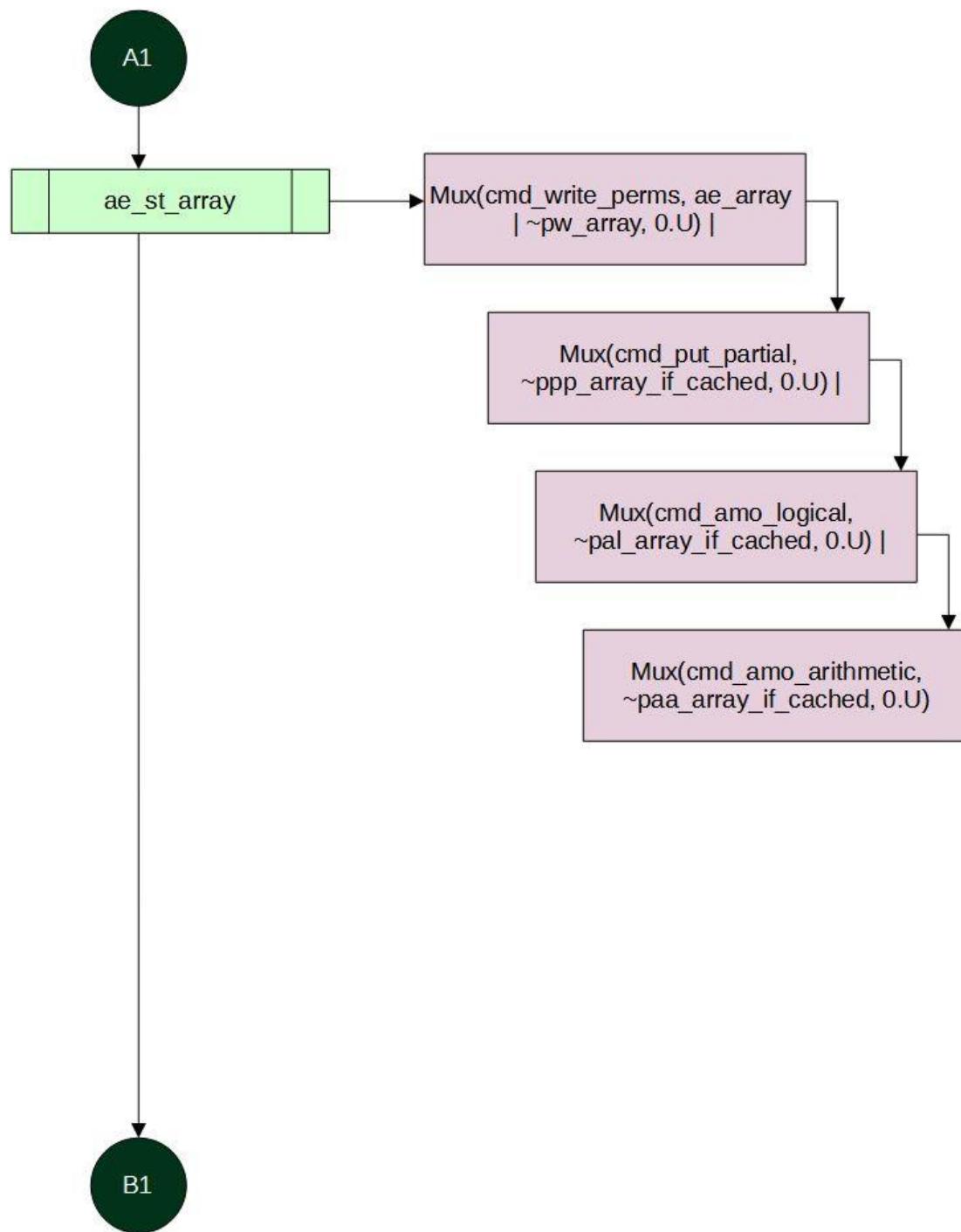
value has been assigned to ae_id_array though an OR and NEGATION operation in
a MUX



```
val ae_st_array =
  Mux(cmd_write_perms, ae_array | ~pw_array, 0.U) |
  Mux(cmd_put_partial, ~ppp_array_if_cached, 0.U) |
  Mux(cmd_amo_logical, ~pal_array_if_cached, 0.U) |
  Mux(cmd_amo_arithmetic, ~paa_array_if_cached, 0.U)
```

— explanation —

An OR and NEGATION operation has been performed on MUX to assign the value to variable

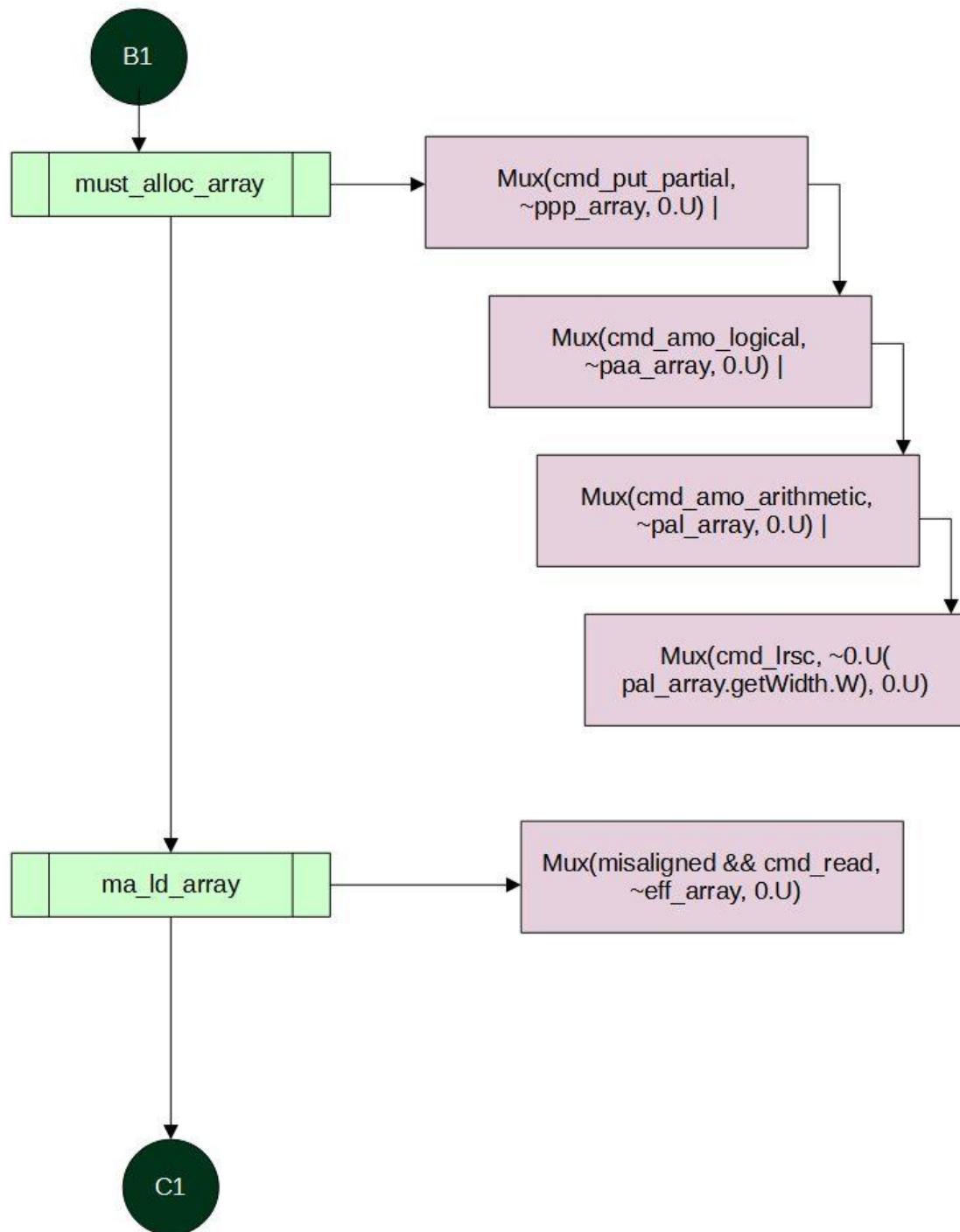


```
val must_alloc_array =
  Mux(cmd_put_partial, ~ppp_array, 0.U) |
  Mux(cmd_amo_logical, ~paa_array, 0.U) |
  Mux(cmd_amo_arithmetic, ~pal_array, 0.U) |
  Mux(cmd_lrsc, ~0.U(pal_array.getWidth.W), 0.U)
val ma_ld_array = Mux(misaligned && cmd_read, ~eff_array, 0.U)
```

— explanation —

An OR operation has been performed on MUX to assign the value

The value has been assigned via a MUX. Negation and AND operations are performed on the input pins



```
val ma_st_array = Mux(misaligned && cmd_write, ~eff_array, 0.U)
val pf_ld_array = Mux(cmd_read, ~(r_array | ptw_ae_array), 0.U)
val pf_st_array = Mux(cmd_write_perms, ~(w_array | ptw_ae_array), 0.U)
val pf_inst_array = ~(x_array | ptw_ae_array)
```

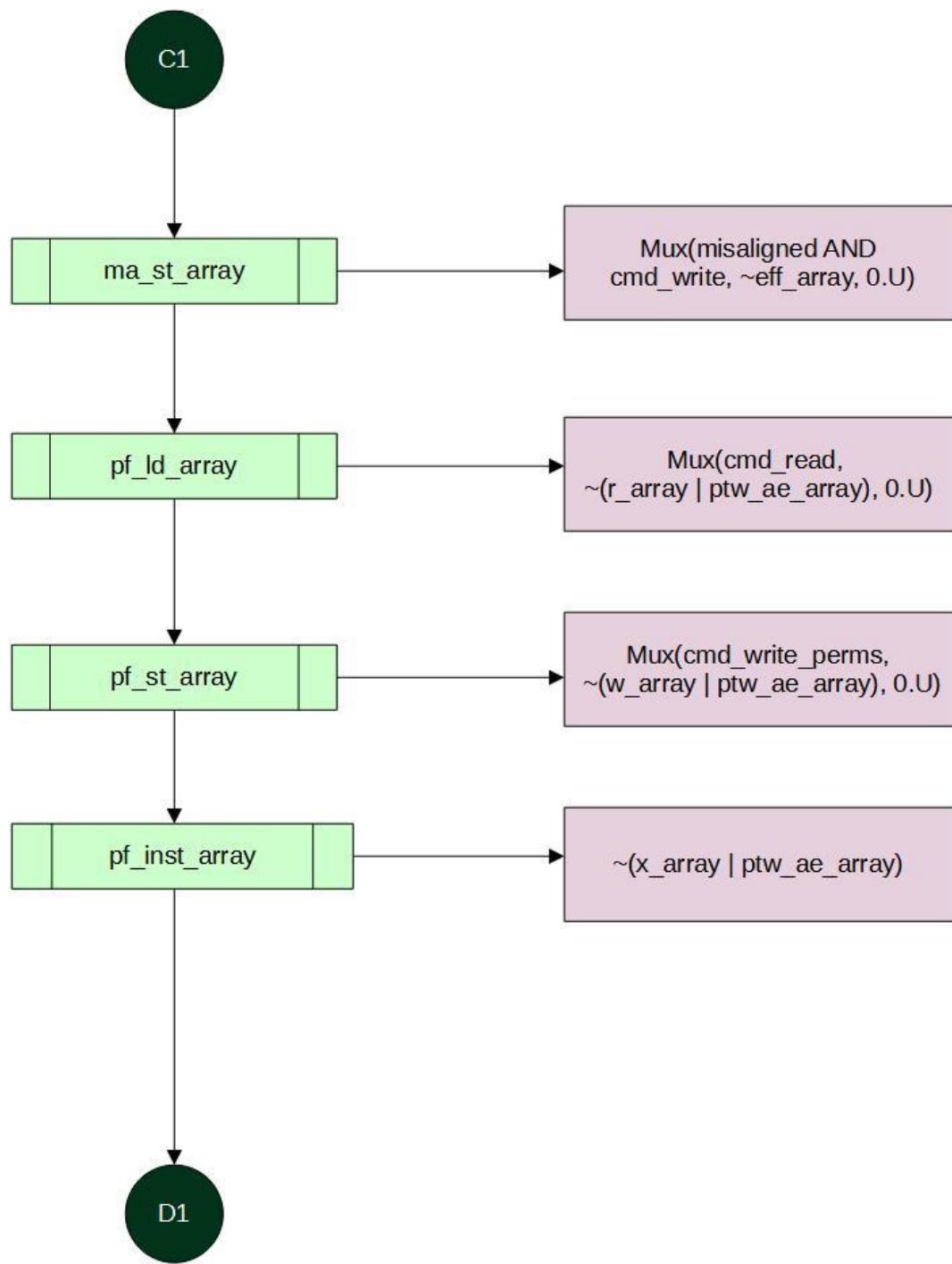
— explanation —

A mux has been initialized by AND and negation operation on input pins

A mux has been initialized by OR and negation operation on input pins

A mux has been initialized by OR and negation operation on input pins

Value has been assigned by OR and NEGATION operation.



```
val tlb_hit = real_hits.orR
val tlb_miss = vm_enabled && !bad_va && !tlb_hit

val sectored_plru = new SetAssocLRU(cfg.nSets, sectored_entries(0).size,
"plru")
val superpage_plru = new PseudoLRU(superpage_entries.size)
```

— explanation —

value has been assigned to tlb_hit and a reduction operator has been performed

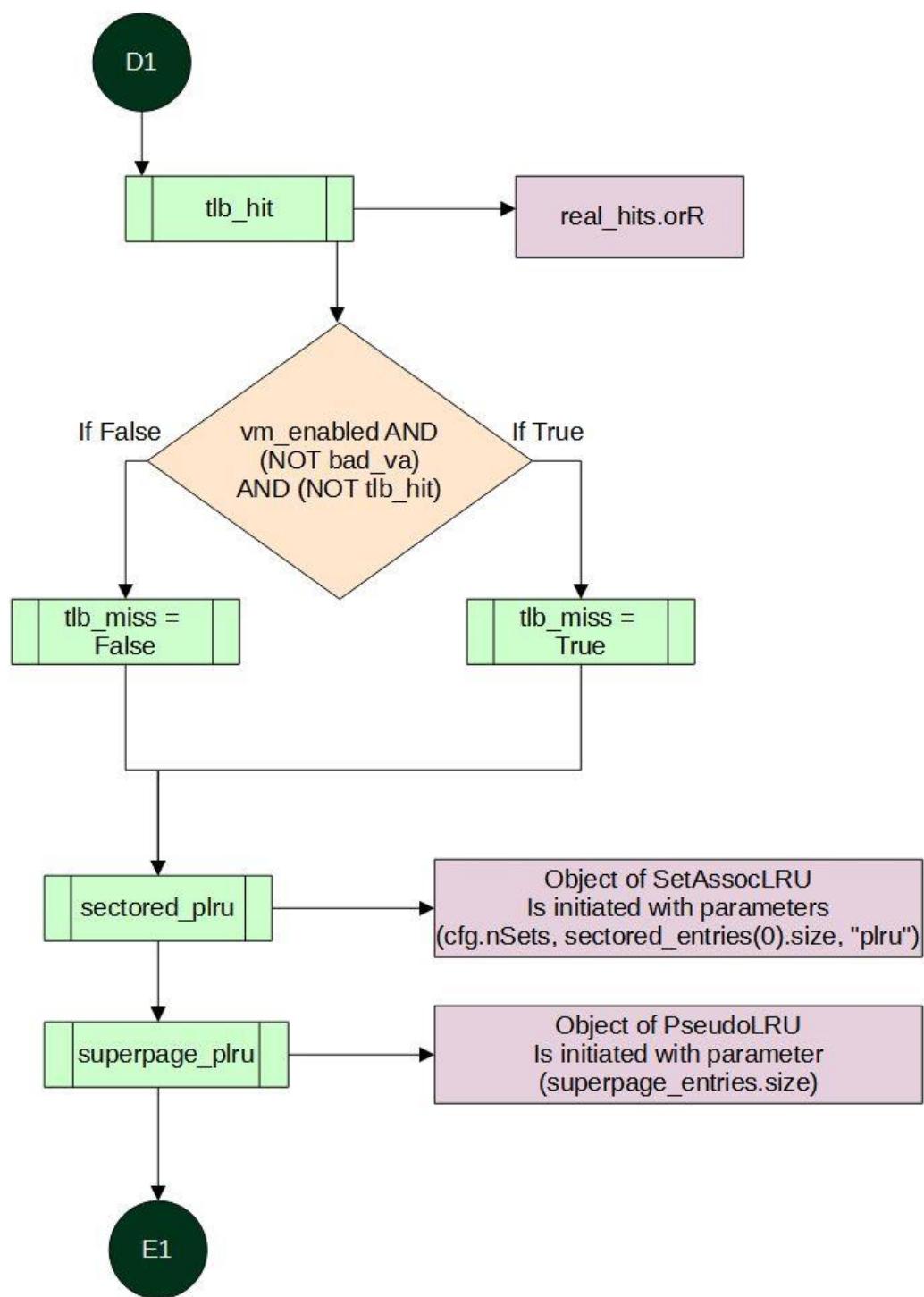
value has been assigned to tlb_miss and a logical AND has been performed with negation

An object has been created of the following class and parameters has been passed

```
class SetAssocLRU(n_sets: Int, n_ways: Int, policy: String) extends
SetAssocReplacementPolicy {
```

An object has been created of the following class and parameters has been passed

```
class PseudoLRU(n_ways: Int) extends ReplacementPolicy {
```



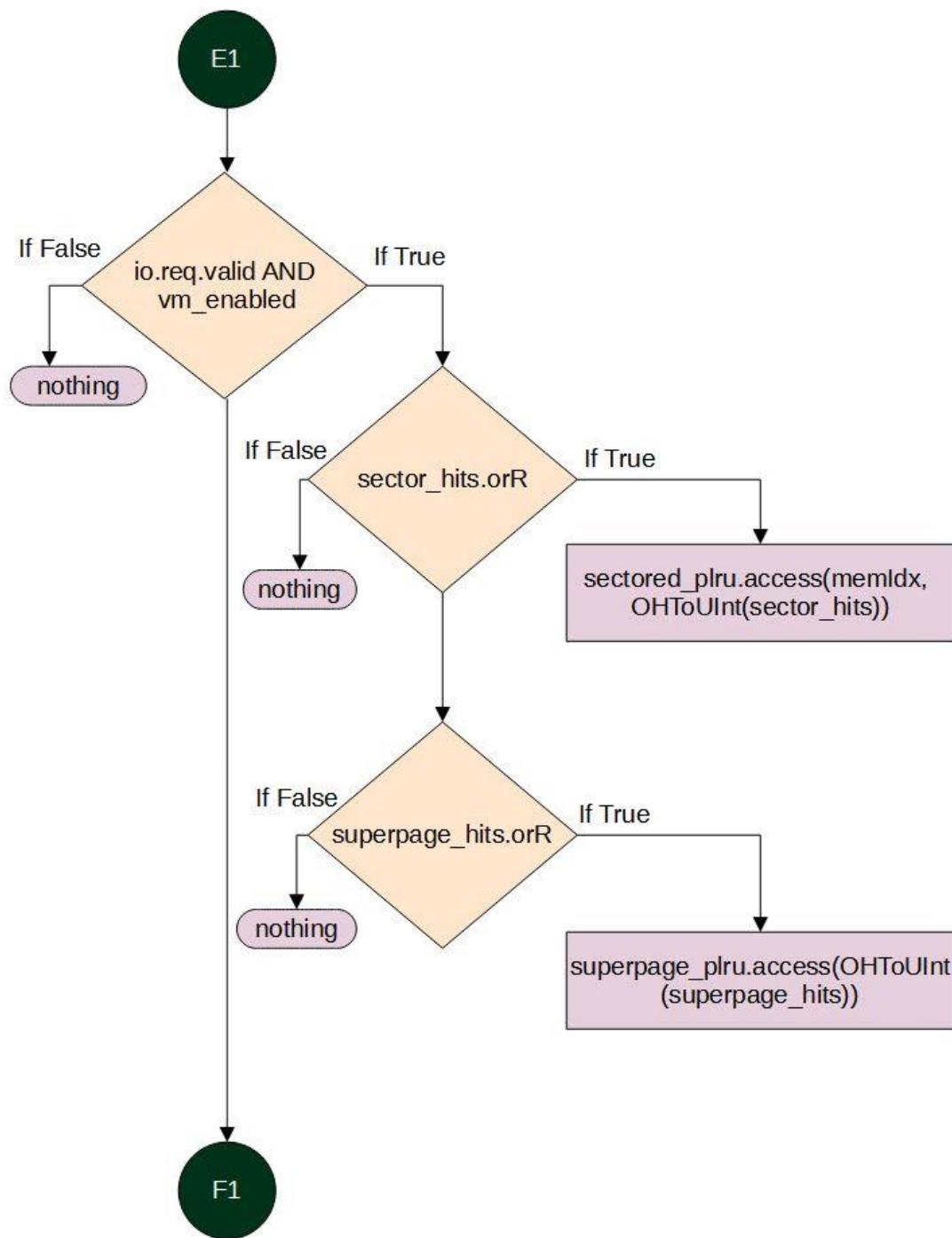
```
when (io.req.valid && vm_enabled) {
    when (sector_hits.orR) { sectored_plru.access(memIdx,
OHToUInt(sector_hits)) }
    when (superpage_hits.orR) {
superpage_plru.access(OHToUInt(superpage_hits)) }
}
```

— explanation —

If condition fulfills, code proceeds otherwise nothing

If condition fulfills, code proceeds and functions are called otherwise nothing

If condition fulfills, code proceeds and functions are called otherwise nothing



ROCKET-CHIP Micro Architecture Specification Document

```
// Superpages create the possibility that two entries in the TLB may match.  
// This corresponds to a software bug, but we can't return complete  
garbage;  
// we must return either the old translation or the new translation.  
This  
// isn't compatible with the Mux1H approach. So, flush the TLB and  
report  
// a miss on duplicate entries.  
val multipleHits = PopCountAtLeast(real_hits, 2)  
  
io.req.ready := state === s_ready  
io.resp.pf.ld := (bad_va && cmd_read) || (pf_ld_array & hits).orR  
io.resp.pf.st := (bad_va && cmd_write_perms) || (pf_st_array & hits).orR
```

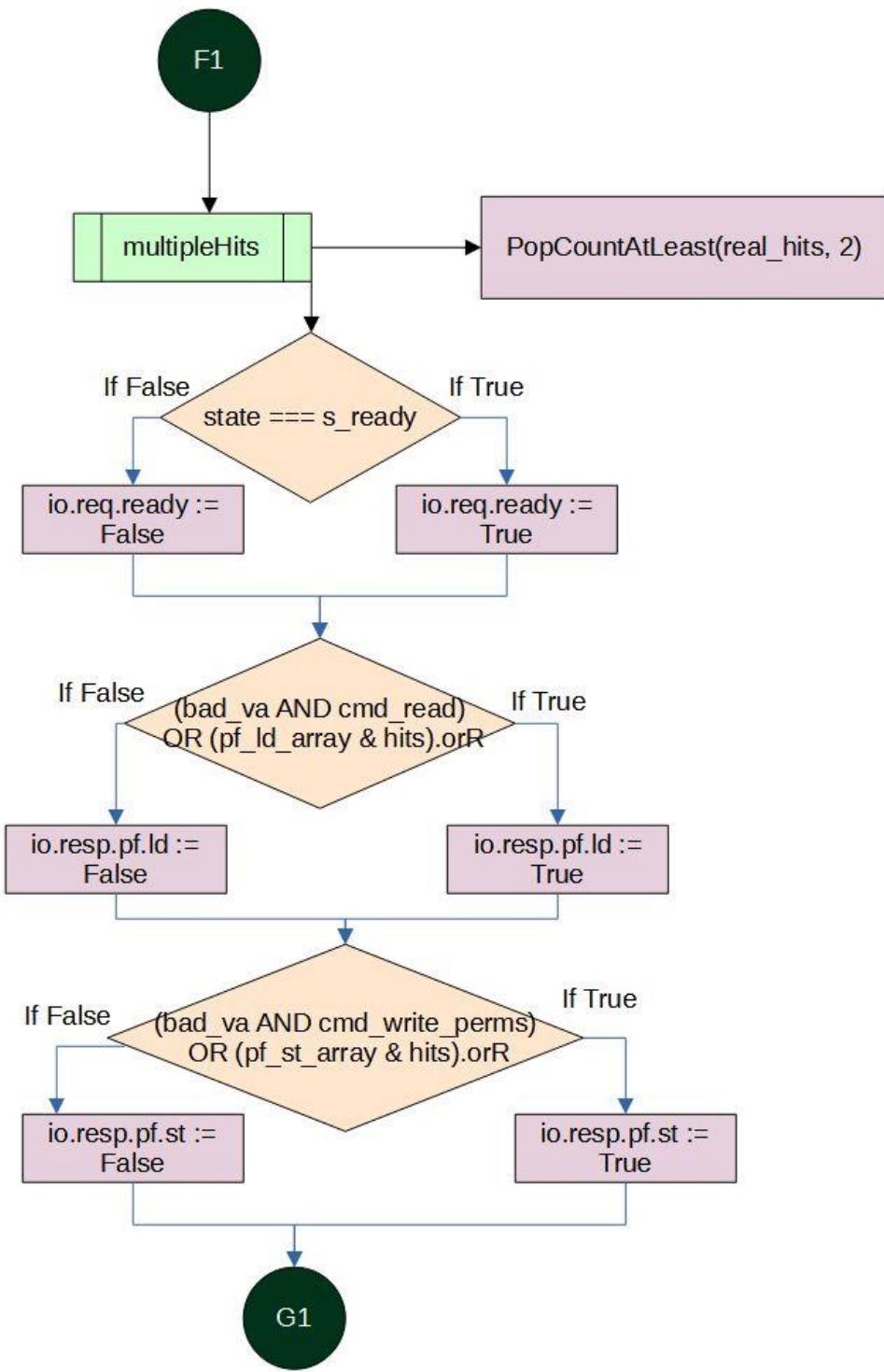
explanation

Method is being called with parameters passed

io.req.ready is true if comparison condition fulfills otherwise false

io.resp.pf.ld is true if comparison condition fulfills otherwise false

io.resp.pf.st is true if comparison condition fulfills otherwise false



```
io.resp(pf.inst := bad_va || (pf_inst_array & hits).orR  
io.resp(ae.ld := (ae_ld_array & hits).orR  
io.resp(ae.st := (ae_st_array & hits).orR  
io.resp(ae.inst := (~px_array & hits).orR  
io.resp(ma.ld := (ma_ld_array & hits).orR
```

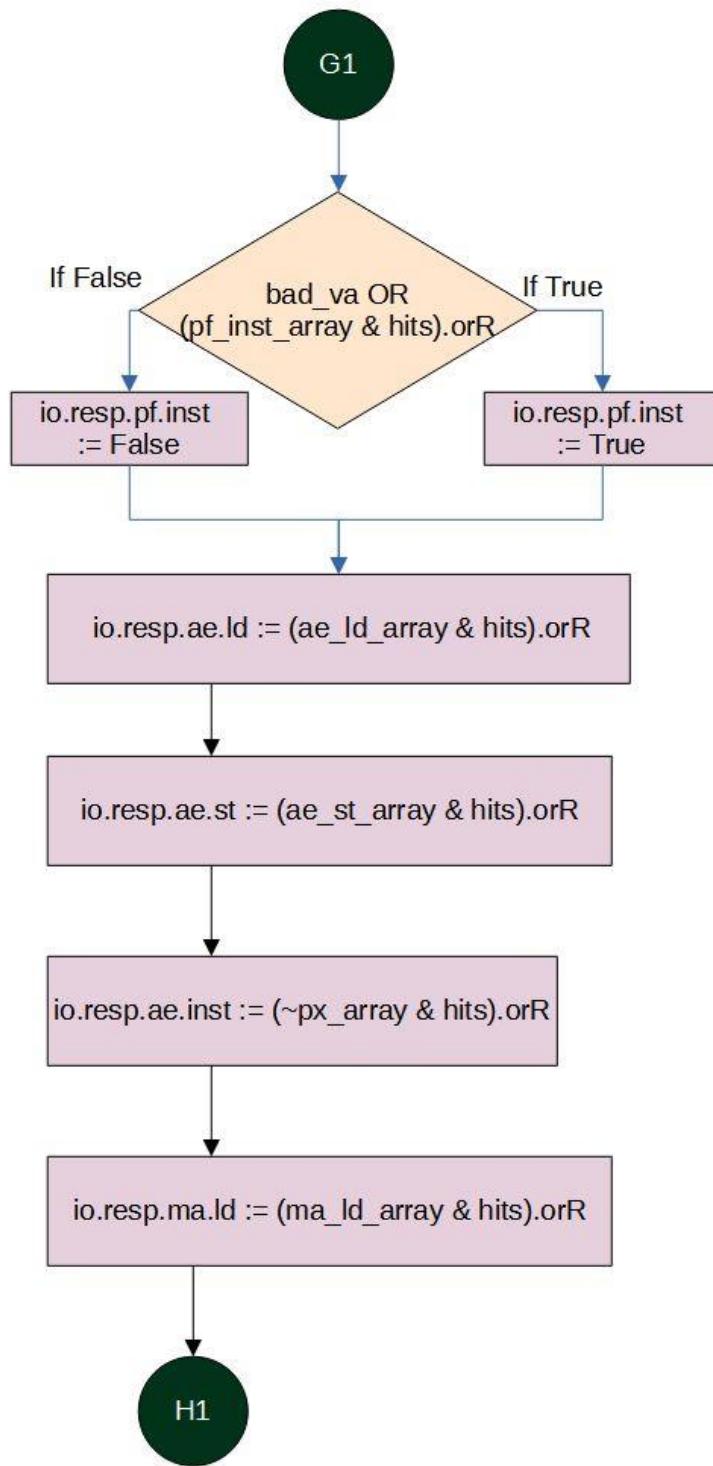
— explanation —

Variable has been wired via reduction operator and comparison operation

Variable has been wired via reduction operator and comparison operation

Variable has been wired via reduction operator and comparison operation with negation

Variable has been wired via reduction operator and comparison operation



```
io.resp.ma.st := (ma_st_array & hits).orR
io.resp.ma.inst := false // this is up to the pipeline to figure out
io.resp.cacheable := (c_array & hits).orR
io.resp.must_alloc := (must_alloc_array & hits).orR
io.resp.prefetchable := (prefetchable_array & hits).orR &&
edge.manager.managers.forall(m => !m.supportsAcquireB || m.supportsHint)
```

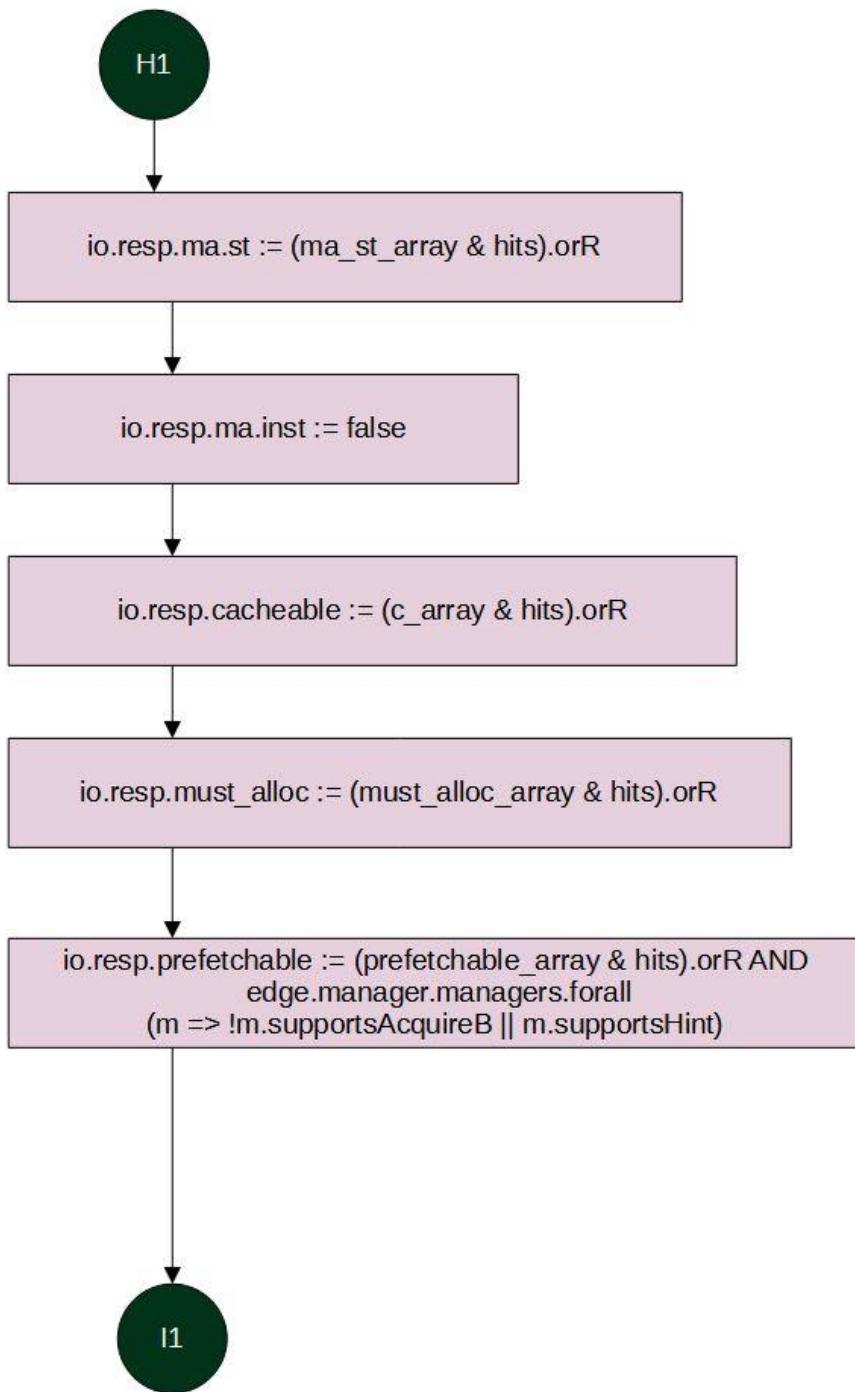
— explanation —

Variable has been wired via reduction operator

Variable has been wired to false

Variable has been wired via reduction operator and comparison operation

Variable has been wired via reduction operator and comparison operation



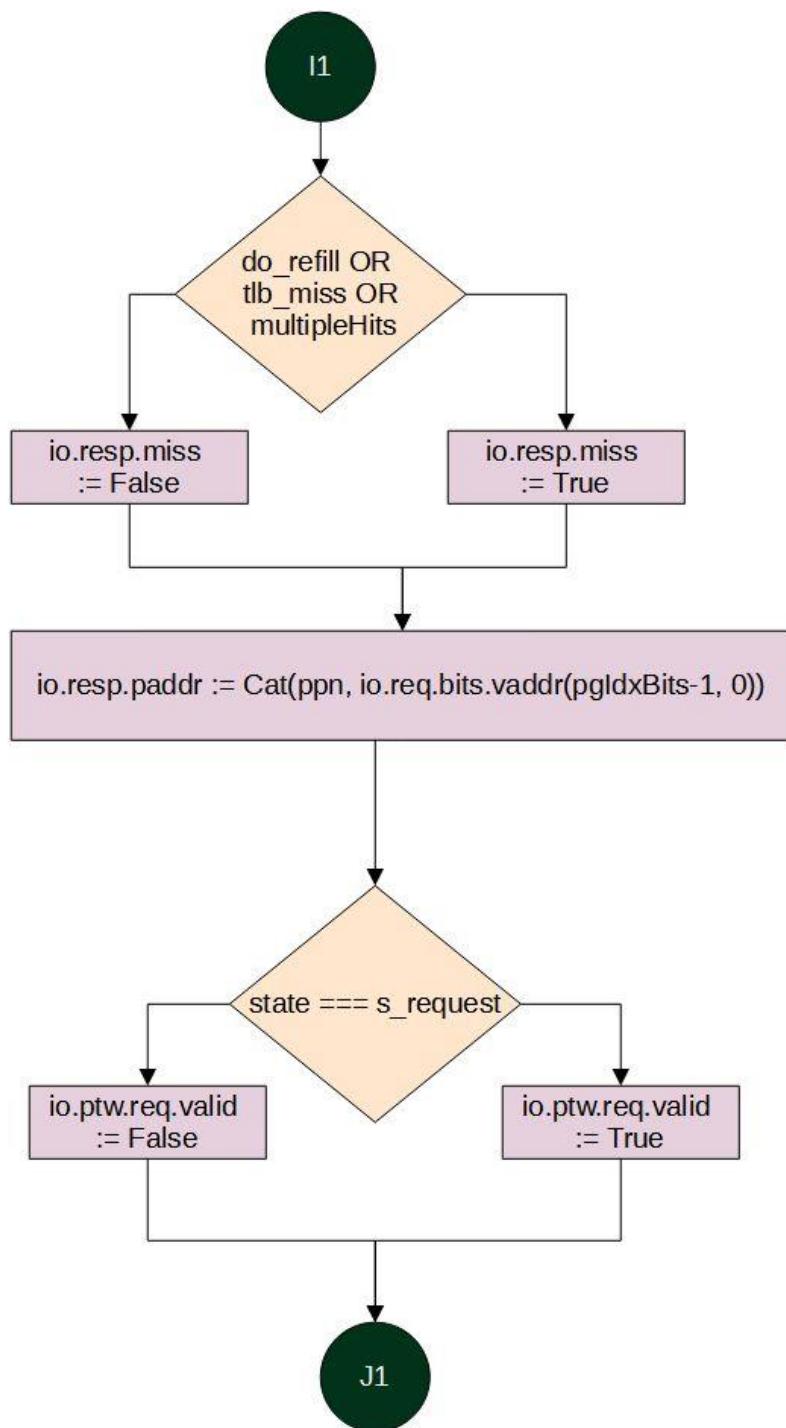
```
io.resp.miss := do_refill || tlb_miss || multipleHits  
io.resp.paddr := Cat(ppn, io.req.bits.vaddr(pgIdxBits-1, 0))  
  
io.ptw.req.valid := state === s_request
```

— explanation —

Value has been assigned by ORed output of three values

Value has been assigned by concatenation of values

Value has been assigned by equalization comparison



```
io.ptw.req.bits.valid := !io.kill
io.ptw.req.bits.bits.addr := r_refill_tag

if (usingVM) {
    val sfence = io.sfence.valid
    when (io.req.fire() && tlb_miss) {
        state := s_request
```

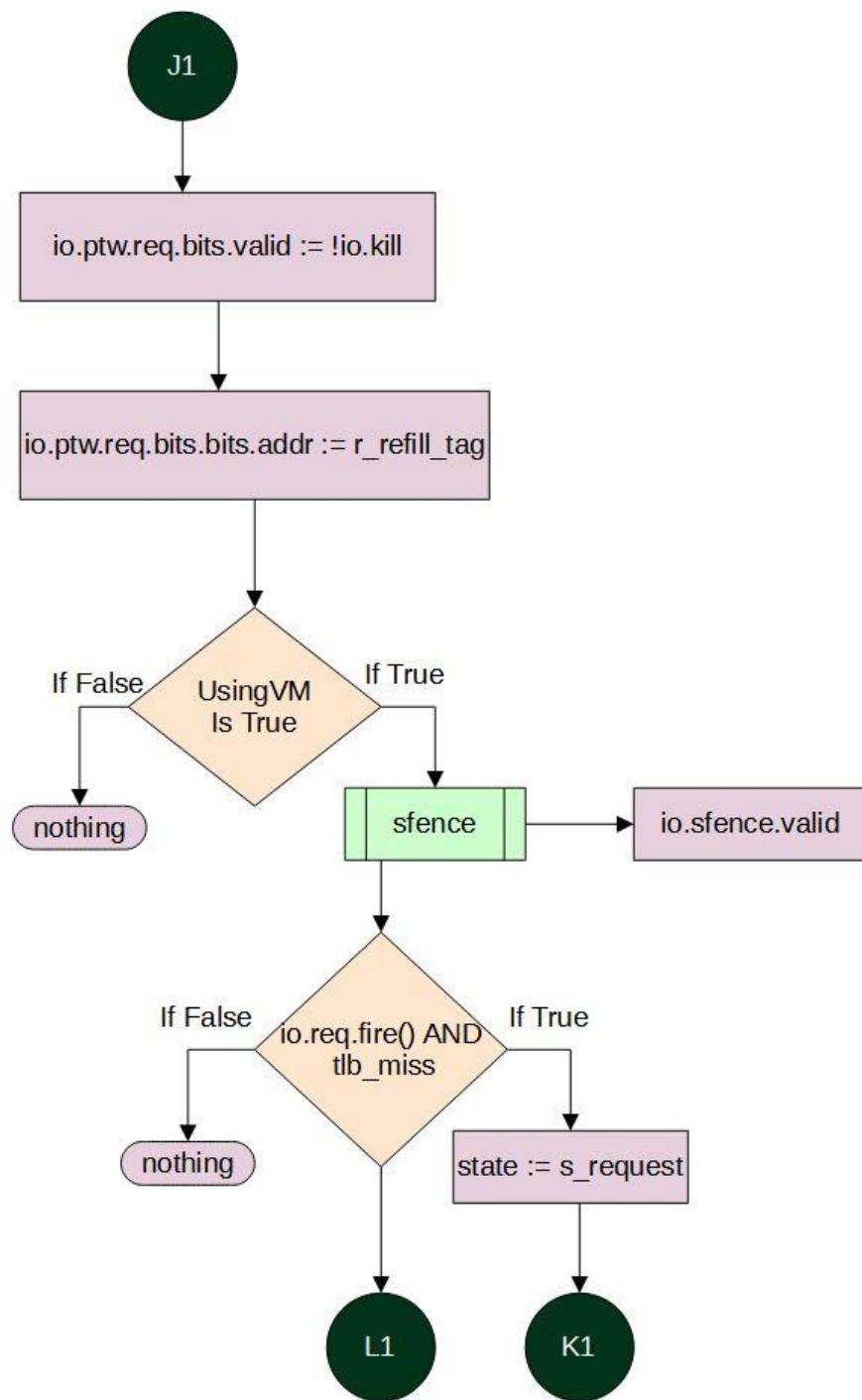
— explanation —

The value of negation of 'io.kill' is being wired

The value of 'r_refill_tag' is being wired

A condition has been placed that if 'using VM' is true then value has been assigned to 'sfence'

A condition has been applied where two values have been AND logically and a value to 'state' is assigned if the condition fulfills



```

r_refill_tag := vpn

r_superpage_repl_addr := replacementEntry(superpage_entries,
superpage_plru.way)
r_sectored_repl_addr := replacementEntry(sectored_entries(memIdx),
sectored_plru.way(memIdx))
r_sectored_hit_addr := OHToUInt(sector_hits)
r_sectored_hit := sector_hits.orR
}

```

— explanation —

Variable 'r_refill_tag' is being wired to the value of 'vpn'

A function is being initialized and parameters has been passed which works as follows:

```

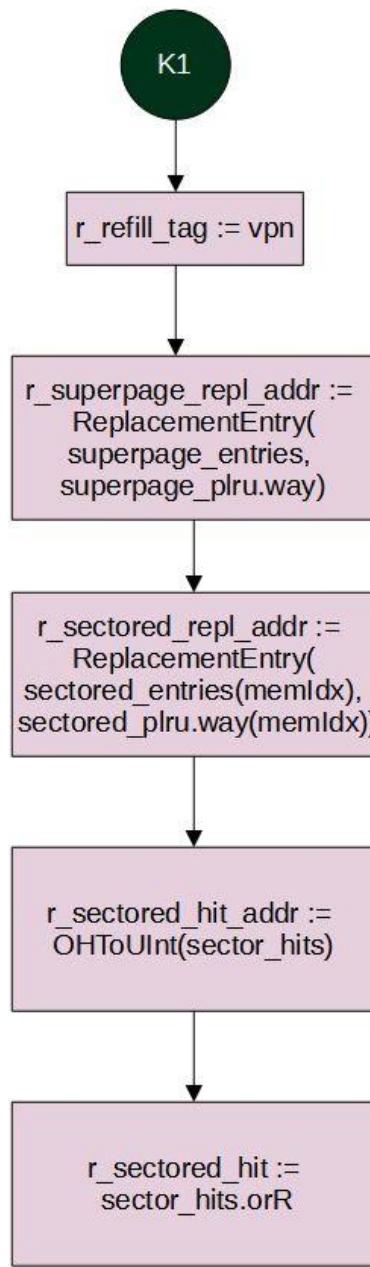
def replacementEntry(set: Seq[TLBEntry], alt: UInt) = {
    val valids = set.map(_.valid.orR).asUInt
    Mux(valids.andR, alt, PriorityEncoder(~valids))
}

```

Value is being assigned to the variable 'r_sectored_hit_addr'

OHToUInt = returns the bit position of the sole high bit of the inputbitvector.

Value is being assigned to variable 'r_sectored_hit'.



```
when (state === s_request) {
    when (sfence) { state := s_ready }
    when (io.ptw.req.ready) { state := Mux(sfence, s_wait_invalidate,
s_wait) }
    when (io.kill) { state := s_ready }
}
```

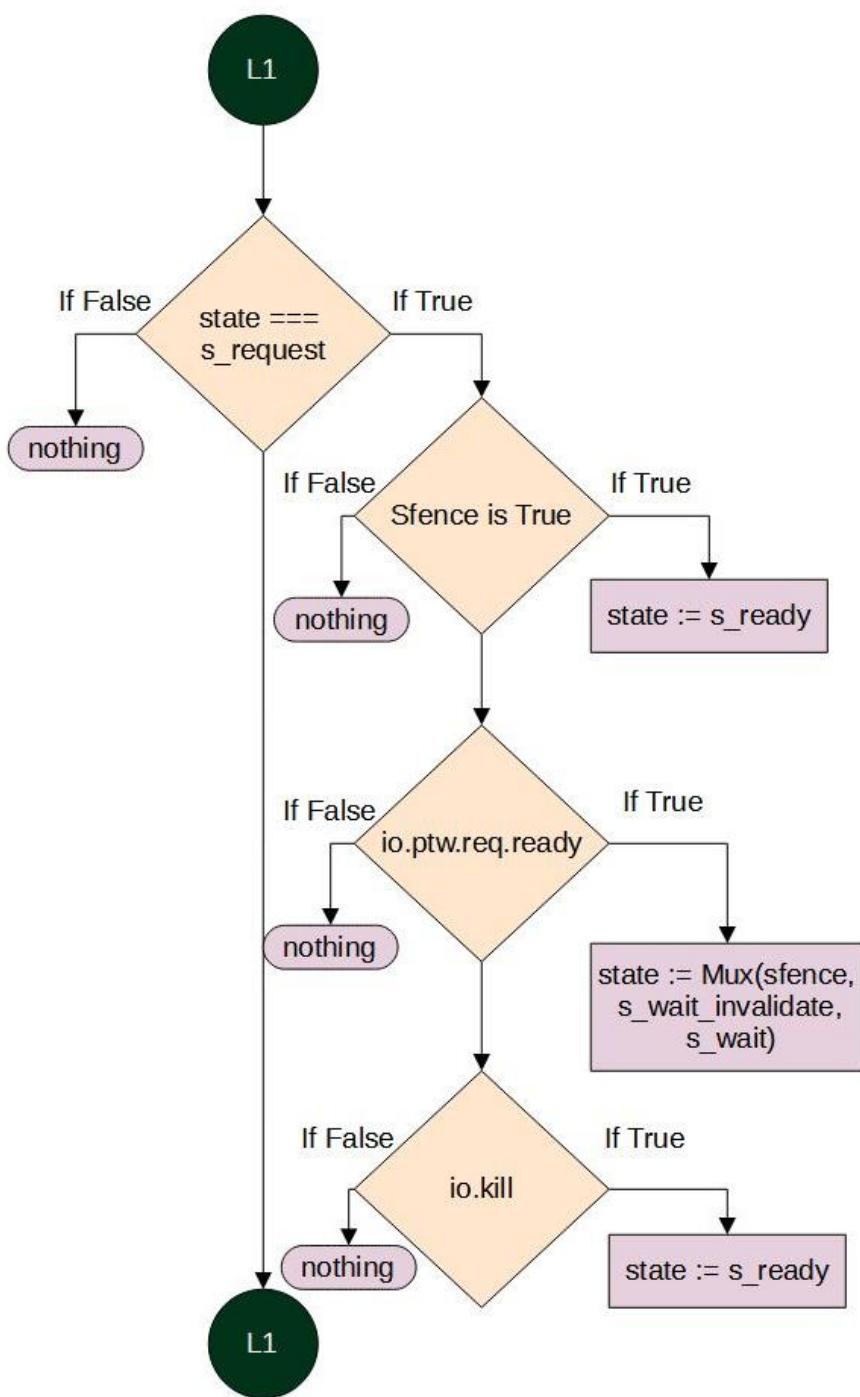
— explanation —

A condition has been inserted to check if value of 'state' is equal to the other value.

A value is being assigned to state if 'sfence' is true

A value is chosen for 'state' via a mux on if 'io.ptw.req.ready' is true

A value is being assigned to 'state' if 'io.lkill' is true

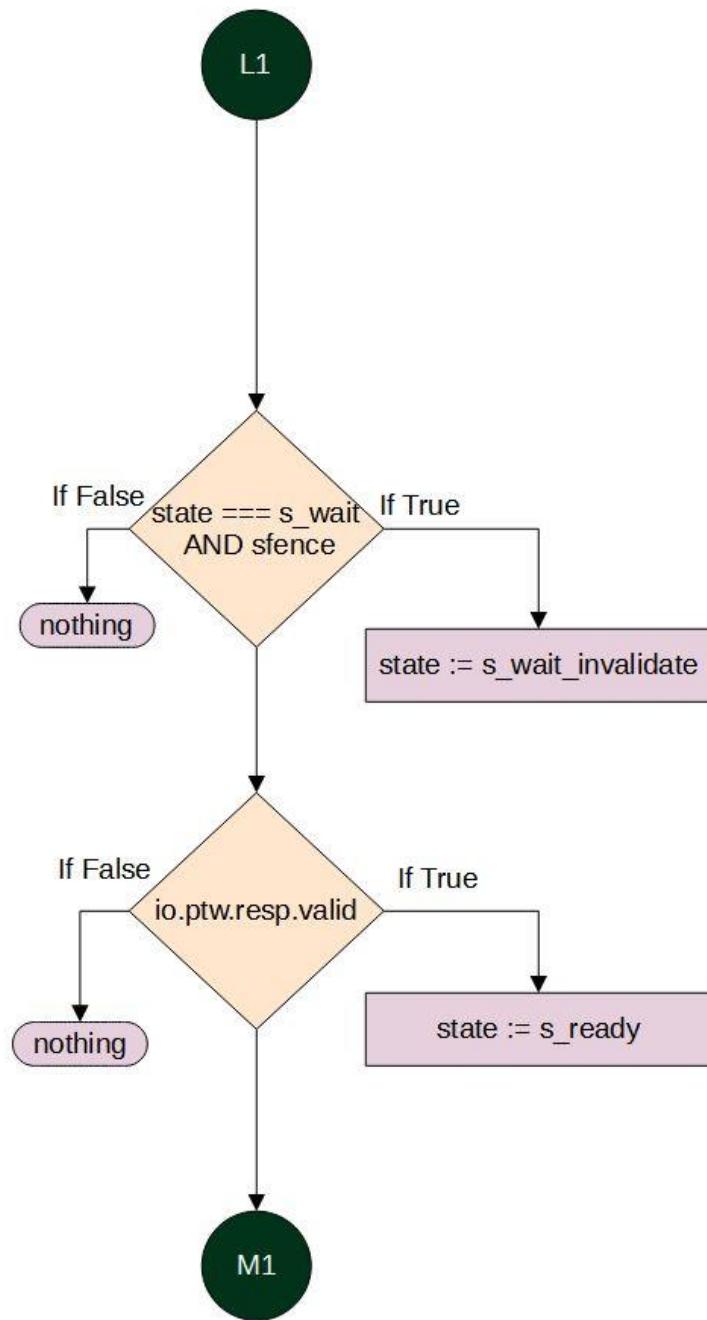


```
when (state === s_wait && sfence) {
    state := s_wait_invalidate
}
when (io.ptw.resp.valid) {
    state := s_ready
}
```

explanation

A comparison is being performed twice of equalization and logical AND in a condition. The condition further assigns a value to variable 'state'

A condition has been applied which checks if the value is true then assignment is made to variable 'state'



```

when (sfence) {
    assert(!io.sfence.bits.rs1 || (io.sfence.bits.addr >> pgIdxBits) ===
vpn)
    for (e <- all_real_entries) {
        when (io.sfence.bits.rs1) { e.invalidateVPN(vpn) }
        .elsewhen (io.sfence.bits.rs2) { e.invalidateNonGlobal() }
        .otherwise { e.invalidate() }
    }
}

```

 explanation

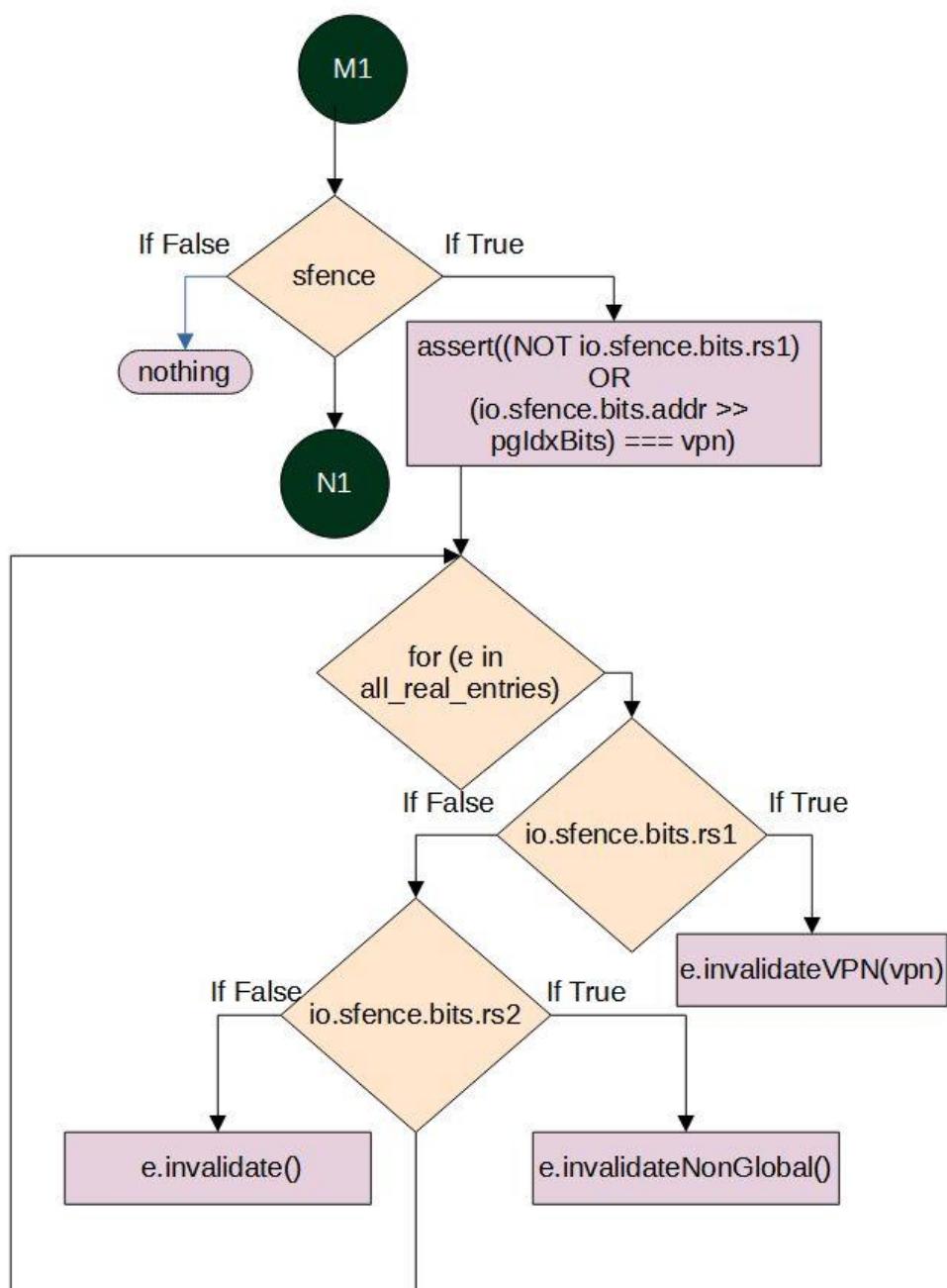
A condition has been applied where if sfence is true, then a comparison is performed if vpn is equal to negation of value 1 or great value from the second.

Assert() = is a workaround for default-value overloading problems in scala as
'assert(cond,"")'

==== = a hardware Bool asserted if this UInt is equal to that

A loop has been initiated which compares the value of e

A condition has been applied which has three possibilities, arrange as when, elsewhen and otherwise all of which modifies the value of e



```
when (multipleHits || reset) {
    all_real_entries.foreach(_.invalidate())
}

ccover(io.ptw.req.fire(), "MISS", "TLB miss")
ccover(io.ptw.req.valid && !io.ptw.req.ready, "PTW_STALL", "TLB miss,
but PTW busy")
ccover(state === s_wait_invalidate, "SFENCE_DURING_REFILL", "flush TLB
during TLB refill")
ccover(sfence && !io.sfence.bits.rs1 && !io.sfence.bits.rs2,
"SFENCE_ALL", "flush TLB")
```

— explanation —

A condition has been applied where if multipleHits is true or either reset, the value of all_real_entries is invalidated by a function being called.

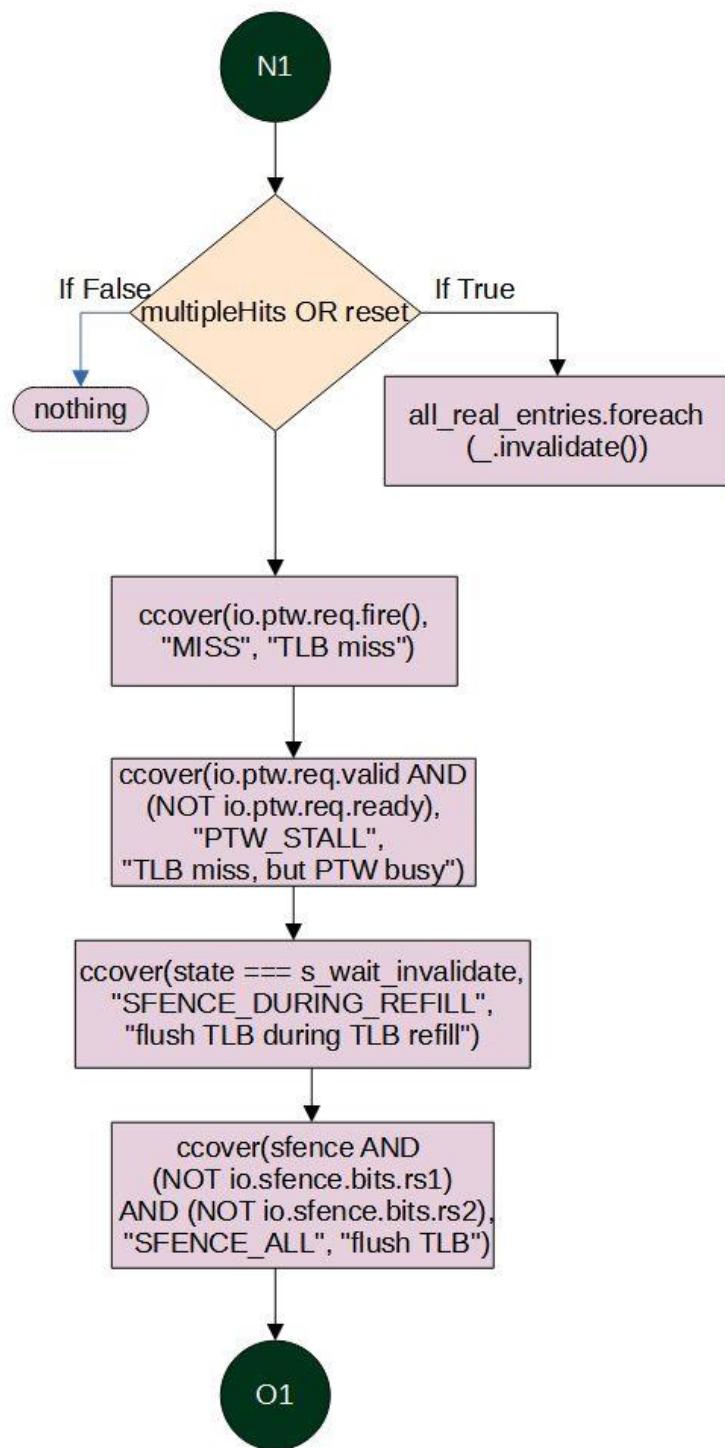
|| = compares two Boolean expressions and returns true if one or both of them evaluate to true

The function ccover has been called with first parameter passed.

The function ccover has been called with first parameter passed is a comparison of two values.

The function ccover has been called with parameters passed.

The function ccover has been called with first parameter passed is a comparison of three values



```
ccover(sfence && !io.sfence.bits.rs1 && io.sfence.bits.rs2,  
"SFENCE_ASID", "flush TLB ASID")  
    ccover(sfence && io.sfence.bits.rs1 && !io.sfence.bits.rs2,  
"SFENCE_LINE", "flush TLB line")  
        ccover(sfence && io.sfence.bits.rs1 && io.sfence.bits.rs2,  
"SFENCE_LINE_ASID", "flush TLB line/ASID")  
            ccover(multipleHits, "MULTIPLE_HITS", "Two matching translations in  
TLB")  
        }
```

explanation

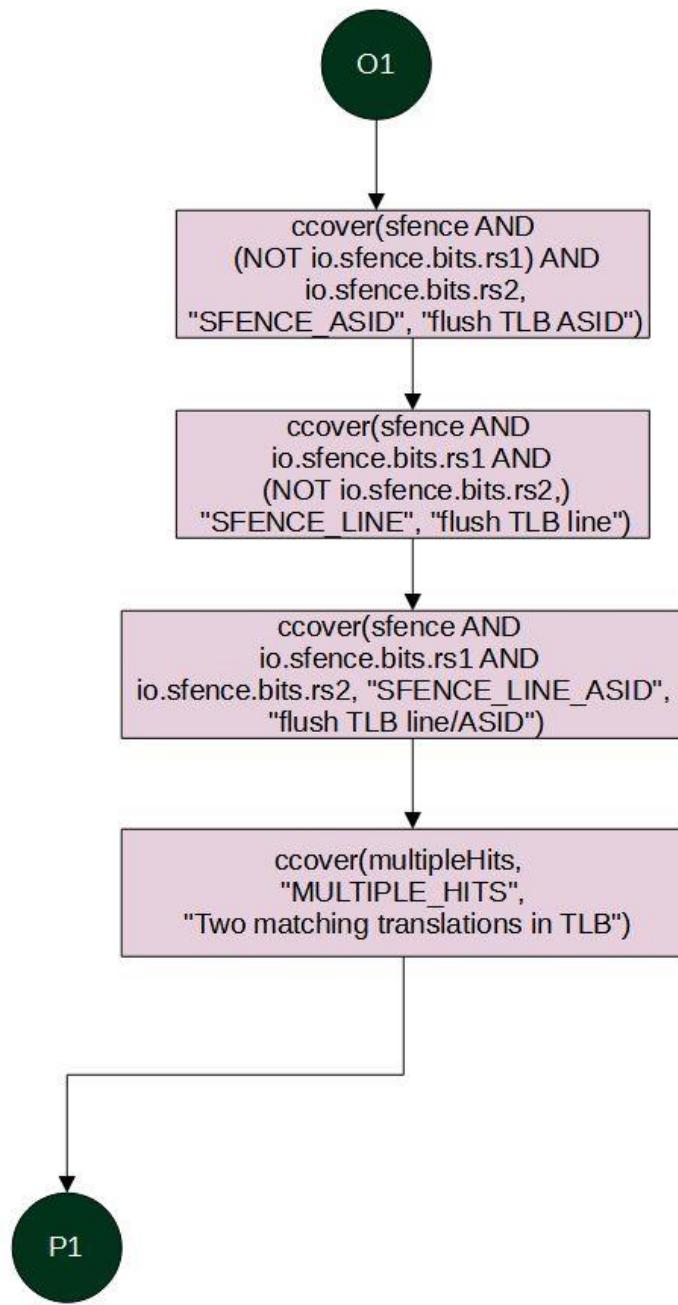
The function ccover has been called with first parameter passed is a comparison of three values

&& = compares two Boolean expressions and returns true if both of them evaluate to true

The function ccover has been called with first parameter passed is a comparison of three values

The function ccover has been called with first parameter passed is a comparison of three values

Function ccover has been called with parameters passed



```

def ccover(cond: Bool, label: String, desc: String)(implicit sourceInfo: SourceInfo) =
    cover(cond, s"${if (instruction) "I" else "D"}TLB_${label}",
"MemorySystem; ;" + desc)

def replacementEntry(set: Seq[TLBEntry], alt: UInt) = {
    val valids = set.map(_.valid.orR).asUInt
    Mux(valids.andR, alt, PriorityEncoder(~valids))
}

```

• explanation •

A function has been defined with parameters and their data types

A function named `cove()` has been called and arguments have been passed. The function `cover()` goes as

```
def cover() = eventSets.foreach { _ withCovers }
```

Also, another function i.e withCovers is explicitly called here which goes as follows:

```
def withCovers {
    events.zipWithIndex.foreach {
        case ((name, func), i) => cover(gate((1.U << i), (func() << i)),
name)
    }
}
```

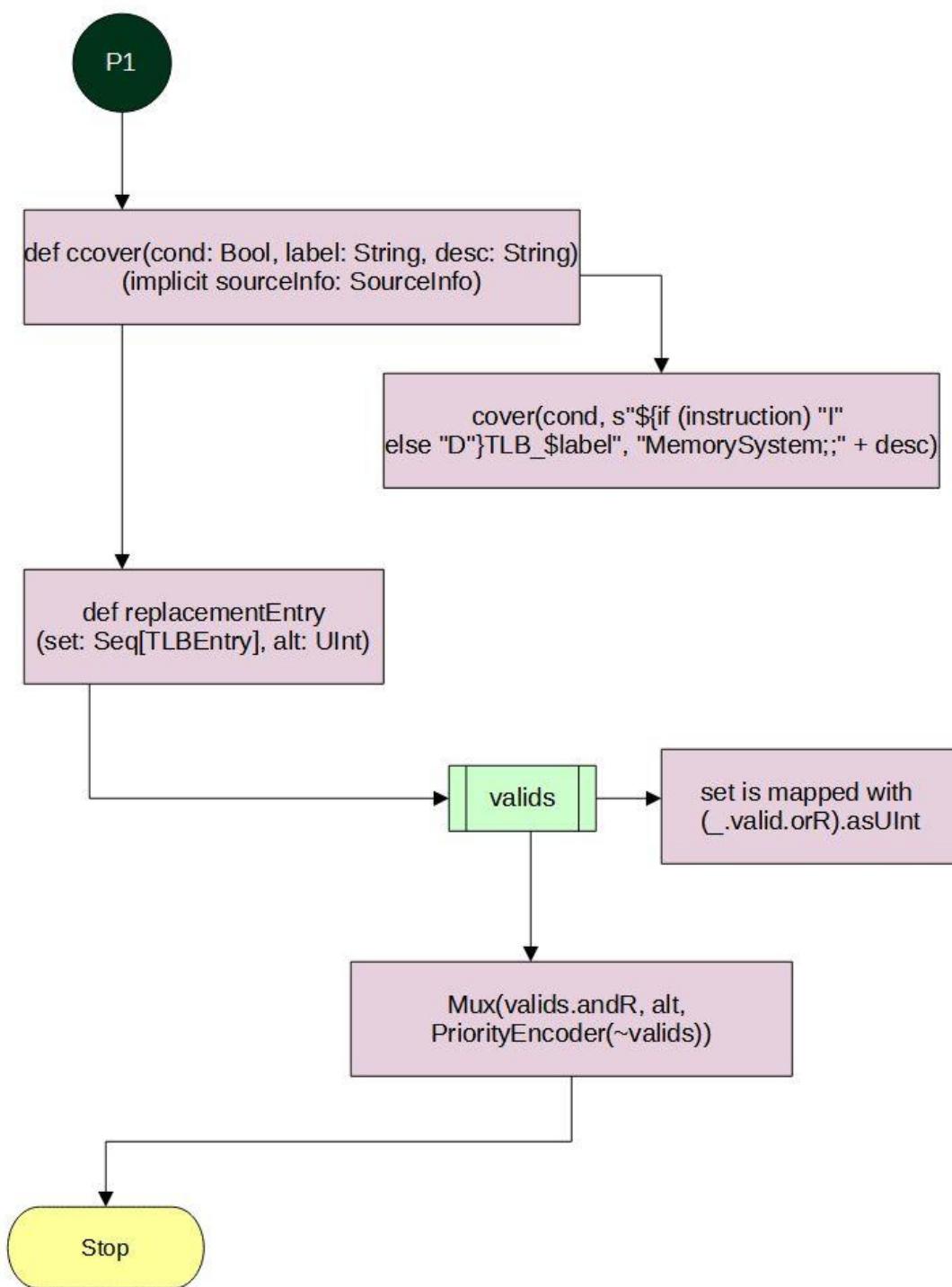
Where .foreach is a common way to iterate over a scala list.

`ForEach` = takes a procedure – a function with a result type `Unit` – as the right operand. It simply applies the procedure to each List element. The result of the operation is again `Unit`; no list of results is assembled.

A function has been defined where a value named as valid is initialized.

`orR` = is a reduction operator i.e. A hardware Bool resulting from every bit of this UInt or'd together

`andR` = is a reduction operator i.e. a hardware Bool resulting from every bit of this UInt and'd together



TLB Permissions

DEEP DIVE INTO CODE

Explanation (By Means of Flowcharts) :

```
case class TLBPermissions(
    homogeneous: Bool,
    r: Bool,
    w: Bool,
    x: Bool,
    c: Bool,
    a: Bool,
    l: Bool)
```

— explanation —————

case class of TLBPermissions is initiated having,

homogenous as Boolean,

r (readable permission) as Boolean,

w (writeable permission) as Boolean

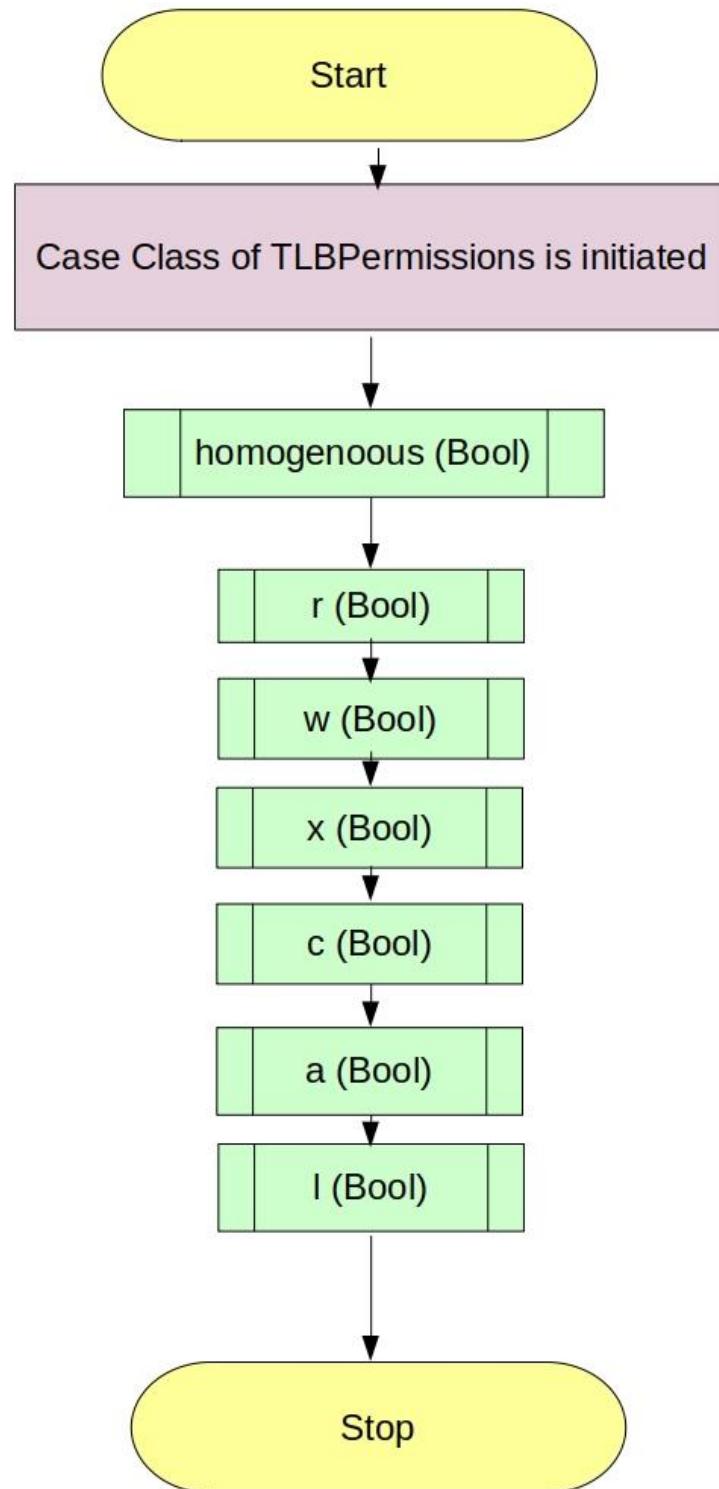
x (executable permission) as Boolean

c (cacheable permission) as Boolean

a (arithmetic ops permission) as Boolean

l (logical ops permission) as Boolean

This case class is defining the basic permissions TLB needs



```
object TLBPageLookup
{
    private case class TLBFixedPermissions(
        e: Boolean,
        r: Boolean,
        w: Boolean,
        x: Boolean,
        c: Boolean,
        a: Boolean,
        l: Boolean)
```

— explanation —

Object TLBPageLookup is created, which has

Private case class TLBFixedPermissions having,

e (get-/put-effects permission) as Boolean,

r (readable permission) as Boolean,

w (writeable permission) as Boolean,

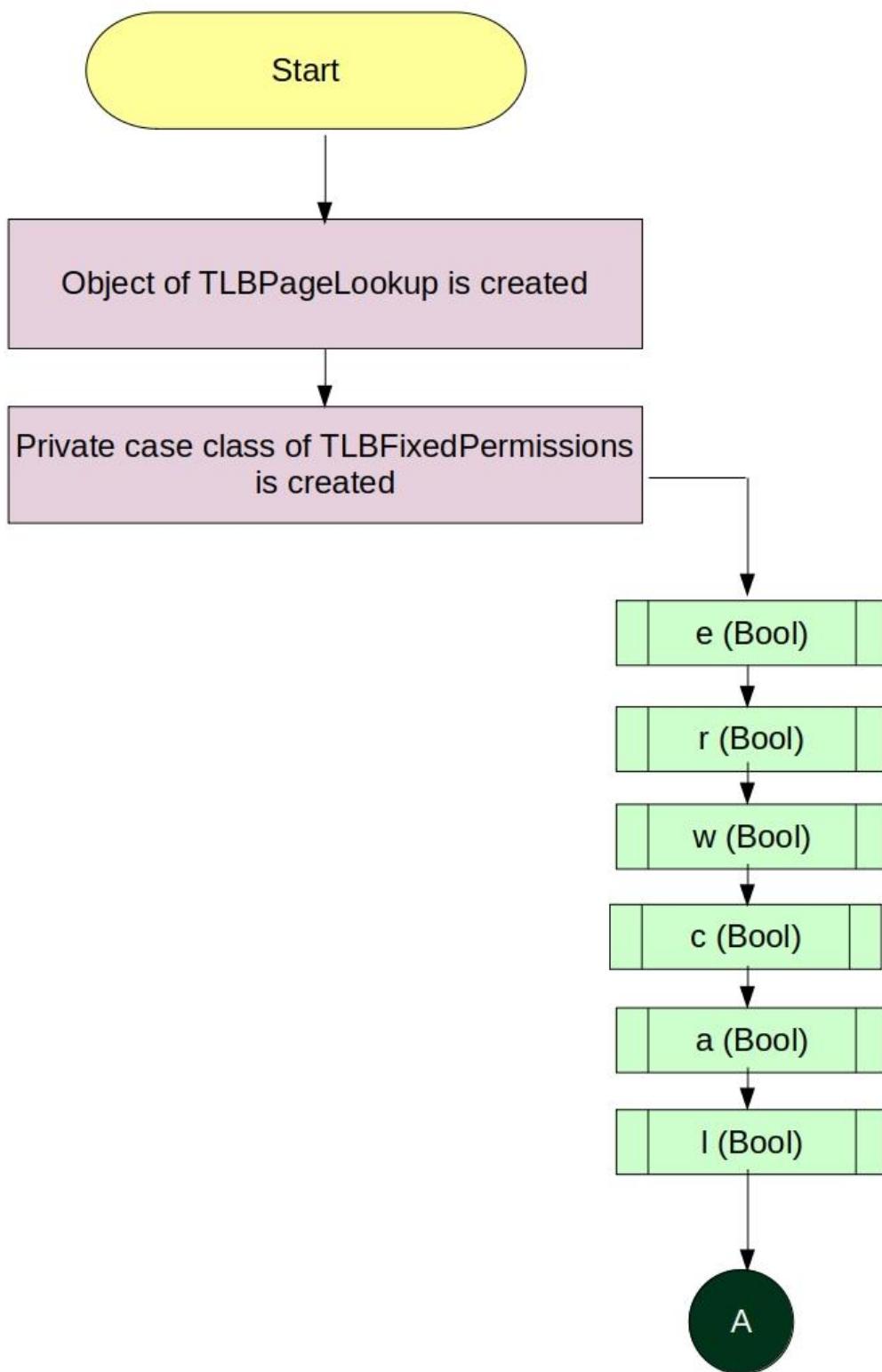
x (executable permission) as Boolean,

c (cacheable permission) as Boolean,

a (arithmetic ops permission) as Boolean,

l (logical ops permission) as Boolean

These permissions are fixed throughout the TLC Pages Lookup.



```

val useful = r || w || x || c || a || l
}

private def groupRegions(managers: Seq[TLManagerParameters]): Map[TLBFixedPermissions, Seq[AddressSet]] = {
  val permissions = managers.map { m =>
    (m.address, TLBFixedPermissions(
      e = Seq(RegionType.PUT_EFFECTS, RegionType.GET_EFFECTS) contains
      m.regionType,
      r = m.supportsGet || m.supportsAcquireB,
      w = m.supportsPutFull || m.supportsAcquireT,
      x = m.executable,
      c = m.supportsAcquireB,
      a = m.supportsArithmetic,
      l = m.supportsLogical)))
  }
}

```

 explanation

variable useful has OR'ed result of all permissions (i.e r,w,x,c,a,l)

private method groupRegions is defined, with parameters;

managers as Seq of TLManagerParameters, which is defined in Parameters.scala,

```

object TLManagerParameters{
  @deprecated("Use TLSlaveParameters.v1 instead of TLManagerParameters", "")
  def apply(
    address:           Seq[AddressSet],
    regionType:        RegionType.T = RegionType.GET_EFFECTS,
    supportsGet:       TransferSizes = TransferSizes.none,
    supportsAcquireB: TransferSizes = TransferSizes.none,
    supportsPutFull:  TransferSizes = TransferSizes.none,
    supportsAcquireT: TransferSizes = TransferSizes.none,
    executable:        Boolean     = false,
    supportsArithmetic: TransferSizes = TransferSizes.none,
    supportsLogical:   TransferSizes = TransferSizes.none,
  )
  object TransferSizes {
    def apply(x: Int) = new TransferSizes(x)
    val none = new TransferSizes(0)
  }
}

```

variable permissions has managers mapped in such a way that,

m.address is the key.

TLBFixedPermissions object is created as value, with,

e (get-/put-effects permission) as Seq of RegionType.PUT_EFFECTS and RegionType.GET_EFFECTS, called with contains method with parameter m.regionType

```

object RegionType {
  val cases = Seq(CACHED, TRACKED, UNCACHED, IDEMPOTENT, VOLATILE,
  PUT_EFFECTS, GET_EFFECTS)
  sealed trait T extends Ordered[T] {

```

```

def compare(that: T): Int = cases.indexOf(that) compare
cases.indexOf(this)
}
case object CACHED      extends T
case object TRACKED    extends T
case object UNCACHED   extends T
case object IDEMPOTENT extends T
case object VOLATILE   extends T
case object PUT_EFFECTS extends T
case object GET_EFFECTS extends T
}

```

r (readable permission) as OR'ed product of m.supportsGet and m.supportsAcquireB

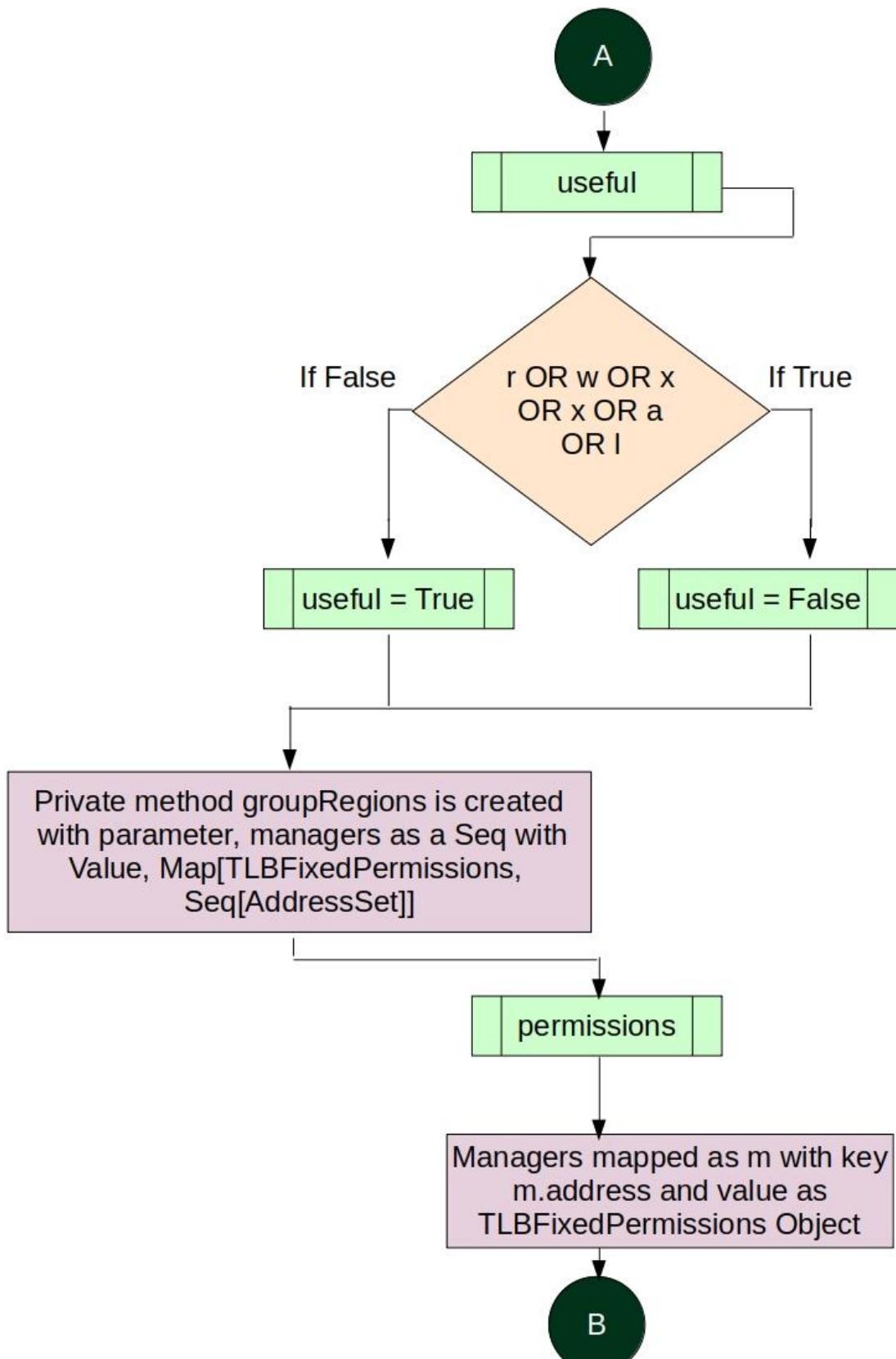
w (writeable permission) as OR'ed product of m.supportsPutFull and
m.supportsAcquireT

x (executable permission) as m.executable

c (cacheable permission) as m.supportAcquireB

a (arithmetic ops permission) as m.supportsArithmetic

l (logical ops permission) as m.supportsLogical



```

permissions
  .filter(_.2.useful)
  .groupBy(_.2)
  .mapValues(seq =>
    AddressSet.unify(seq.flatMap(_.1)))
}

def apply(managers: Seq[TLManagerParameters], xLen: Int, cacheBlockBytes: Int, pageSize: BigInt): UInt = {
  require (isPow2(xLen) && xLen >= 8)
  require (isPow2(cacheBlockBytes) && cacheBlockBytes >= xLen/8)
  require (isPow2(pageSize) && pageSize >= cacheBlockBytes)
}

```

 explanation

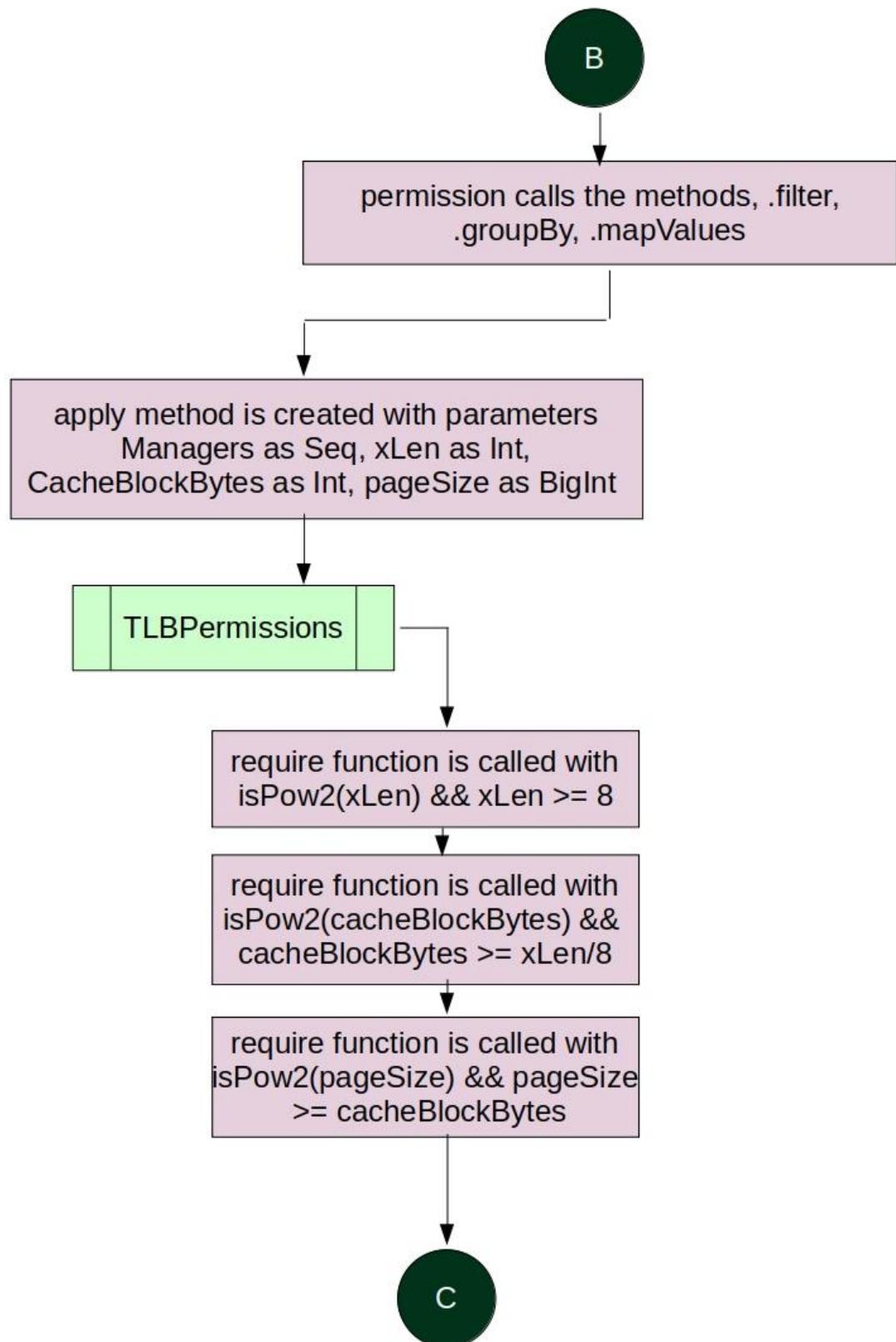
permissions is called with .filter method and .groupBy method, and .mapValues with seq as AddressSet unified with seq.flatMap

Method apply is created, with parameters, managers as Seq of TLManagerParameters and xLen as Int and cacheBlockBytes as Int and pageSize as BigInt.

Require function is called with isPow2 of xLen AND'ed with xLen >= 8

Require function is called with isPow2 of cacheBlockBytes AND'ed with cacheBlockBytes >= xLen/8

Require function is called with isPow2 of pageSize AND'ed with pageSize >= cacheBlockBytes.



```
val xferSizes = TransferSizes(cacheBlockBytes, cacheBlockBytes)
val allSizes = TransferSizes(1, cacheBlockBytes)
val amoSizes = TransferSizes(4, xLen/8)

val permissions = managers.foreach { m =>
```

— explanation —

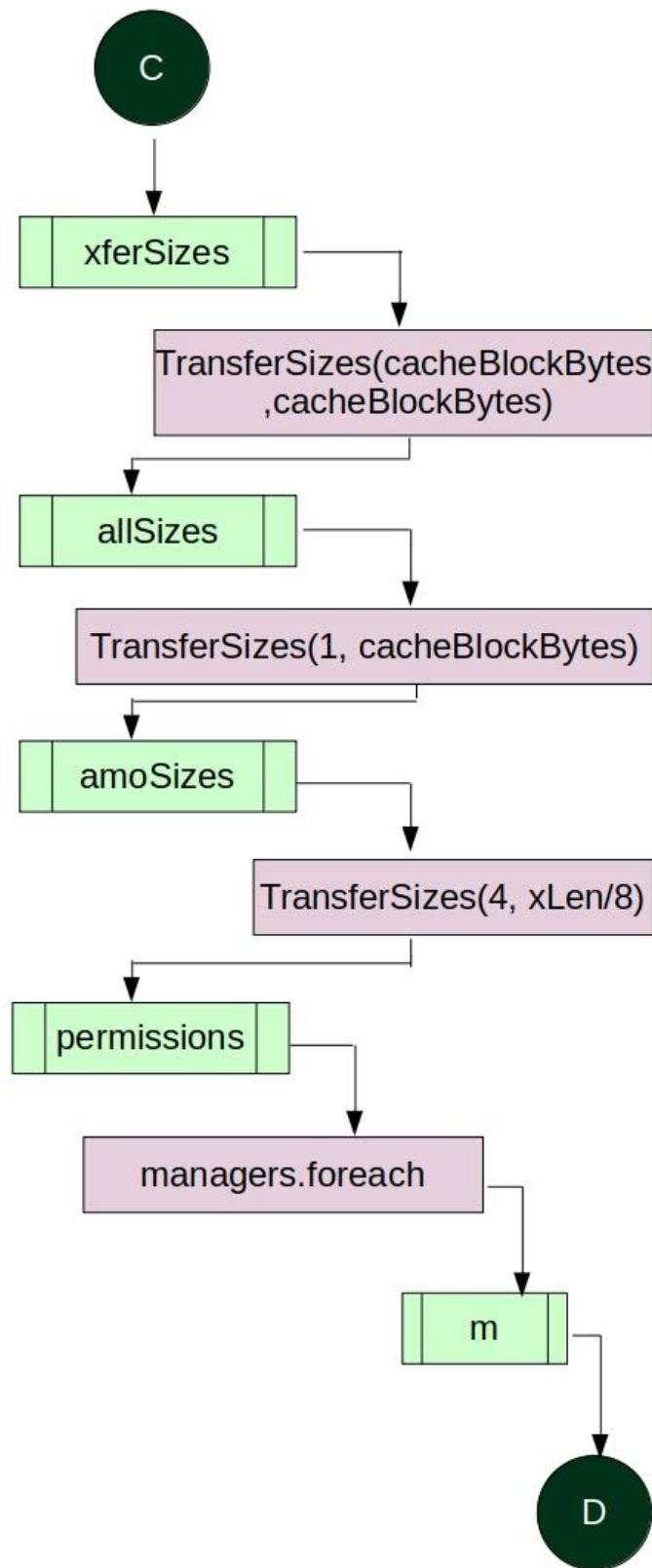
xferSizes variable has TransferSizes of cacheBlockBytes and cacheBlockBytes.

allSizes variable has TransferSizes of 1 and cacheBlockBytes

amoSizes variable has TransferSizes of 4 and xLen/8

```
object TransferSizes {
  def apply(x: Int) = new TransferSizes(x)
  val none = new TransferSizes(0)
}
```

permissions variable has managers .foreach with m



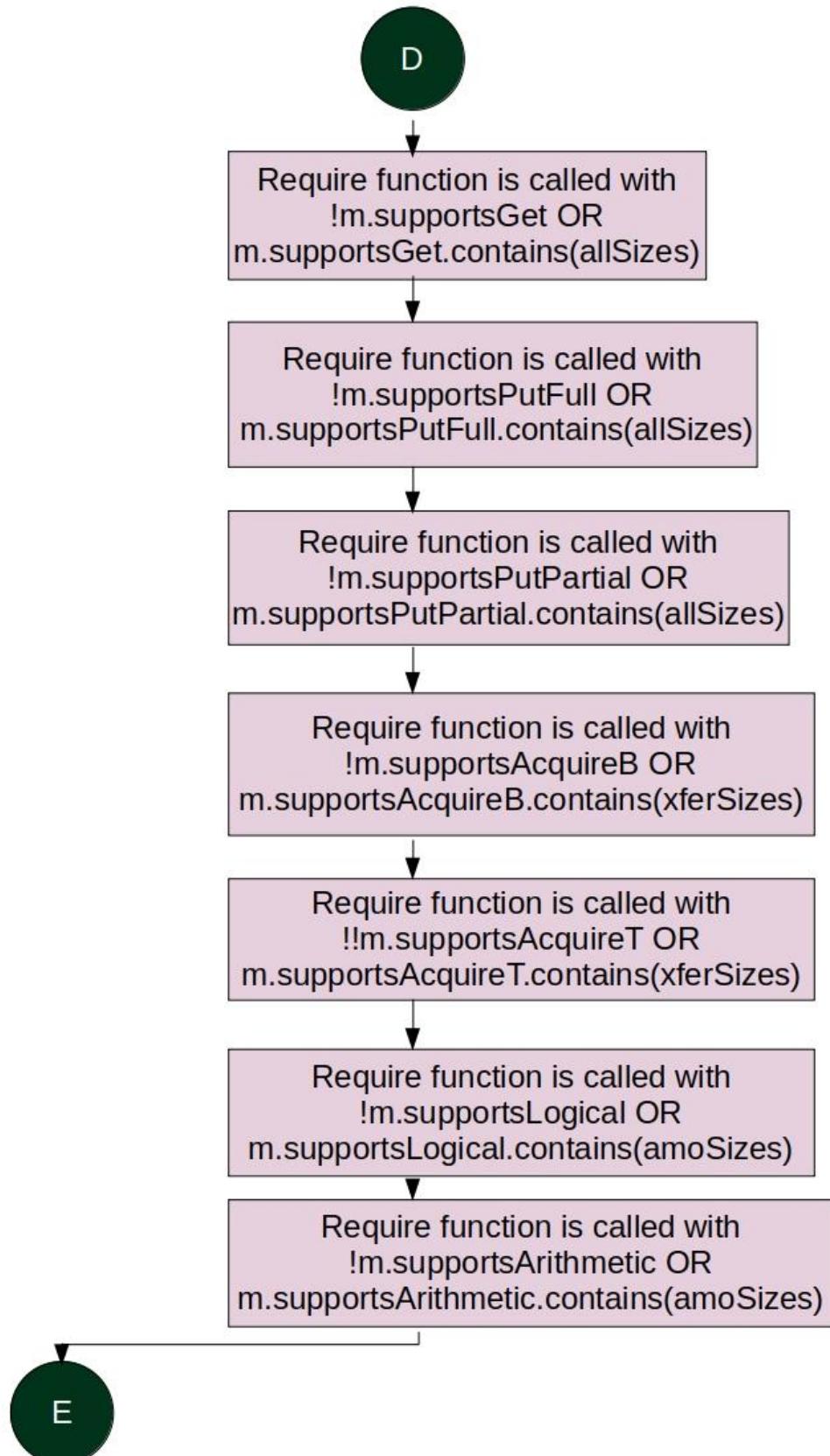
```

    require (!m.supportsGet      || m.supportsGet
.contains(allSizes), s"Memory region '${m.name}' at ${m.address} only
supports ${m.supportsGet} Get, but must support ${allSizes}")
    require (!m.supportsPutFull  || m.supportsPutFull
.contains(allSizes), s"Memory region '${m.name}' at ${m.address} only
supports ${m.supportsPutFull} PutFull, but must support ${allSizes}")
    require (!m.supportsPutPartial ||
m.supportsPutPartial.contains(allSizes), s"Memory region '${m.name}' at
${m.address} only supports ${m.supportsPutPartial} PutPartial, but must
support ${allSizes}")
    require (!m.supportsAcquireB || m.supportsAcquireB
.contains(xferSizes), s"Memory region '${m.name}' at ${m.address} only
supports ${m.supportsAcquireB} AcquireB, but must support ${xferSizes}")
    require (!m.supportsAcquireT || m.supportsAcquireT
.contains(xferSizes), s"Memory region '${m.name}' at ${m.address} only
supports ${m.supportsAcquireT} AcquireT, but must support ${xferSizes}")
    require (!m.supportsLogical  || m.supportsLogical
.contains(amoSizes), s"Memory region '${m.name}' at ${m.address} only
supports ${m.supportsLogical} Logical, but must support ${amoSizes}")
    require (!m.supportsArithmetic ||
m.supportsArithmetic.contains(amoSizes), s"Memory region '${m.name}' at
${m.address} only supports ${m.supportsArithmetic} Arithmetic, but must
support ${amoSizes}")
}

```

— explanation —

require functions are called with the following parameters for catching exceptions.



```

val grouped = groupRegions(managers).mapValues(_.filter(_.alignment >=
pageSize))

def lowCostProperty(prop: TLBFixedPermissions => Boolean): UInt => Bool
= {
    val (yesm, nom) = grouped.partition { case (k, eq) => prop(k) }
    val (yes, no) = (yesm.values.flatten.toList,
nom.values.flatten.toList)

    val decisionMask = AddressDecoder(Seq(yes, no))
}

```

 explanation

grouped variable has groupRegions method with parameter managers with .mapValues

lowCostProperty method is defined with parameters prop as TLBFixedPermissions object in Boolean, which returns a UInt.

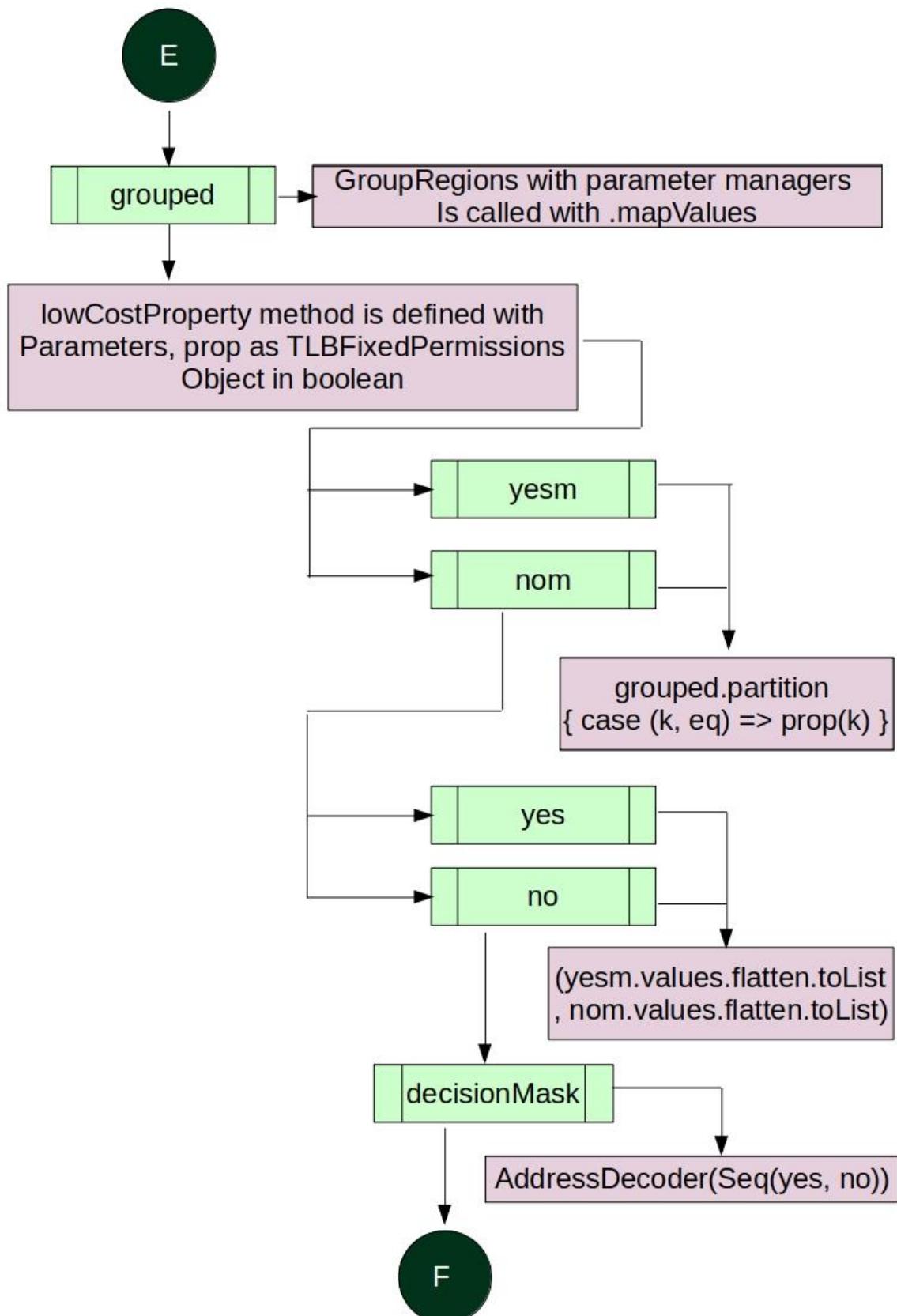
Yesm and nom variables are defined with grouped partitioned with case of k and eq with prop k.

Yes and no variables are defined with yesm values flattened into a list.

Decision mask variable is defined with AddressDecoder with Seq of yes and no

AddressDecoder is an object, defined in the AddressDecoder.scala

This chunk of code, maps and filters the values coming from managers, and then inside the method, those mapped values are used in such a way that AddressDecoder is sent the indices through which the Address should be decoded .



```

def simplify(x: Seq[AddressSet]) =
  AddressSet.unify(x.map(_.widen(~decisionMask)).distinct)
  val (yesf, nof) = (simplify(yes), simplify(no))
  if (yesf.size < no.size) {
    (x: UInt) => yesf.map(_.contains(x)).foldLeft(false.B) (_ || _)
  } else {
    (x: UInt) => !nof.map(_.contains(x)).foldLeft(false.B) (_ || _)
  }
}

```

 explanation

simplify method is defined in the lowCostProperty method, with parameters, x as Seq of AddressSet.

AddressSet is unified with x.map.

```

object AddressSet
{
  def unify(seq: Seq[AddressSet]): Seq[AddressSet] = {
    val bits = seq.map(_.base).foldLeft(BigInt(0))(_ | _)
    AddressSet.enumerateBits(bits).foldLeft(seq) { case (acc, bit) =>
      unify(acc, bit) }.sorted
  }

  def unify(seq: Seq[AddressSet], bit: BigInt): Seq[AddressSet] = {

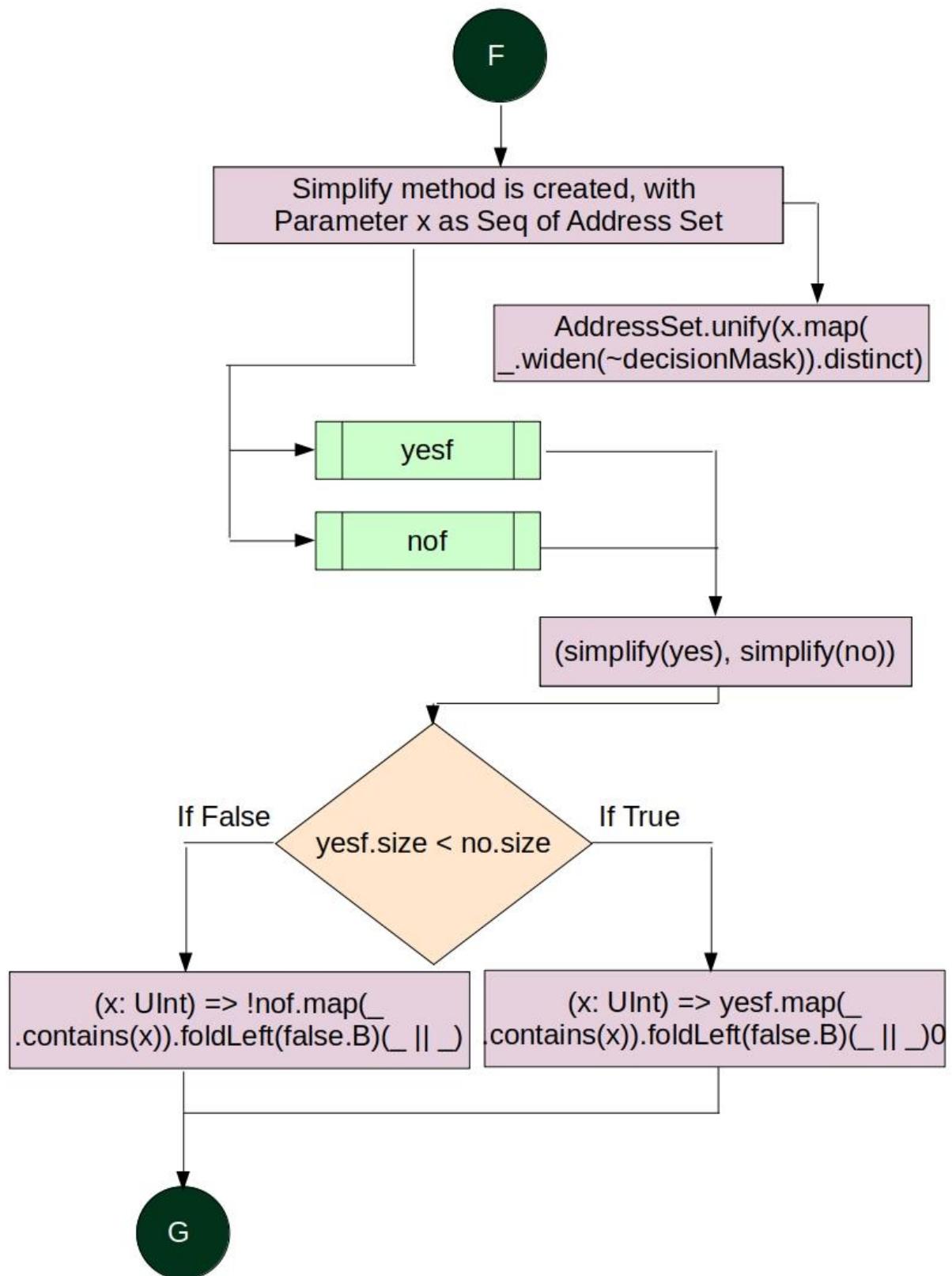
    seq.distinct.groupBy(x => x.copy(base = x.base & ~bit)).map { case
      (key, seq) =>
      if (seq.size == 1) {
        seq.head // singleton -> unaffected
      } else {
        key.copy(mask = key.mask | bit) // pair - widen mask by bit
      }
    }.toList
  }
}

```

Yesf and nof variables are defined with simplified method with yes and no as a parameter, respectively.

If size of yesf is less than size of no, then

x as UInt is defined with yesf mapped and foldLeft with false and _ OR'ed with _
else. x as UInt is defined with !nof mapped and foldLeft with false and _ OR'ed with _



```
val rfn = lowCostProperty(_.r)
val wfn = lowCostProperty(_.w)
val xfn = lowCostProperty(_.x)
val cfn = lowCostProperty(_.c)
val afn = lowCostProperty(_.a)
val lfn = lowCostProperty(_.l)

val homo = AddressSet.unify(grouped.values.flatten.toList)
```

Explanation

rfn variable is defined with lowCostProperty method with _.r as parameter.

wfn variable is defined with lowCostProperty method with _.w as parameter.

xfn variable is defined with lowCostProperty method with _.x as parameter.

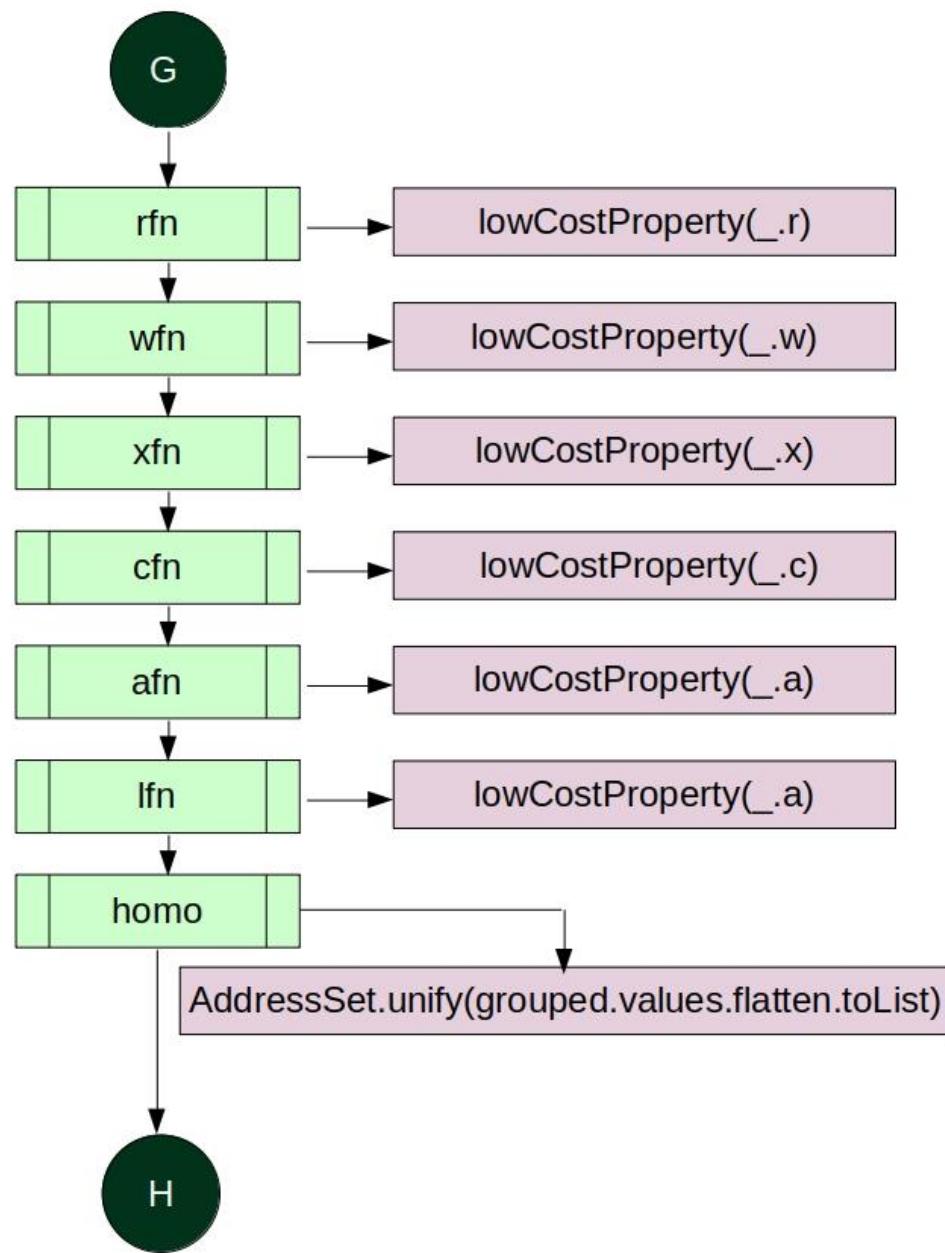
cfn variable is defined with lowCostProperty method with _.c as parameter.

afn variable is defined with lowCostProperty method with _.a as parameter.

lfn variable is defined with lowCostProperty method with _.l as parameter.

Homo variable is defined as AddressSet with unify method with parameter, grouped values flatten to list.

In this chunk of code, all the permissions are defined with lowCostProperty Method, and AddressSet si unifies with the grouped values.



```
(x: UInt) => TLBPermissions(
    homogeneous = homo.map(_.contains(x)).foldLeft(false.B) (_ || _),
    r = rfn(x),
    w = wfn(x),
    x = xfn(x),
    c = cfn(x),
    a = afn(x),
    l = lfn(x))
}

def homogeneous(managers: Seq[TLManagerParameters], pageSize: BigInt): Boolean = {
    groupRegions(managers).values.forall(_.forall(_.alignment >= pageSize))
}
}
```

 explanation

x as UInt has TLBPermissions object with;

homogenous as homo mapped and foldLeft with false and _ OR'ed with _

r as rfn of index x

w as wfn of index x

x as xfn of index x

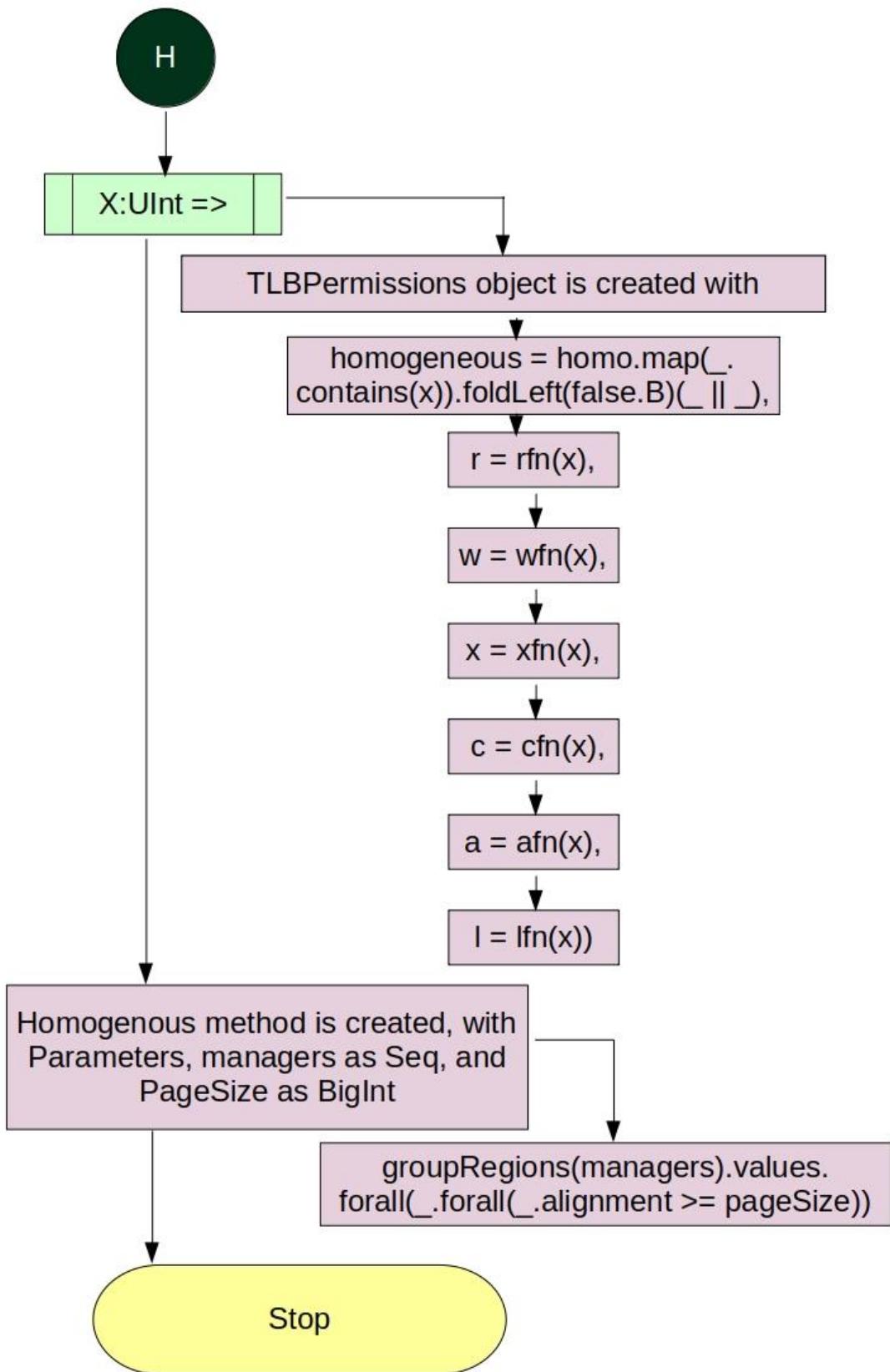
c as cfn of index x

a as afn of index x

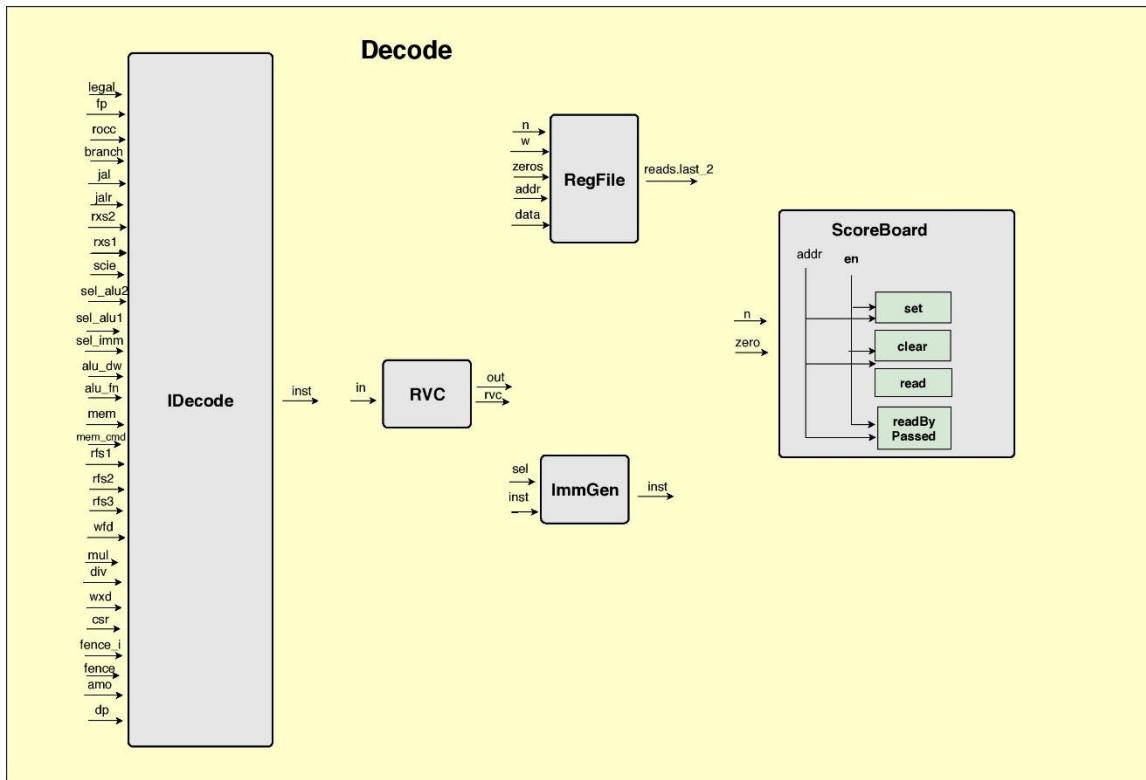
l as lfn of index x

homogenous variable is defined with parameters, managers as Seq of TLManagerParameters, and pageSize as BigInt, in which

groupRegions with parameter managers with values and forall methods.



DECODE STAGE



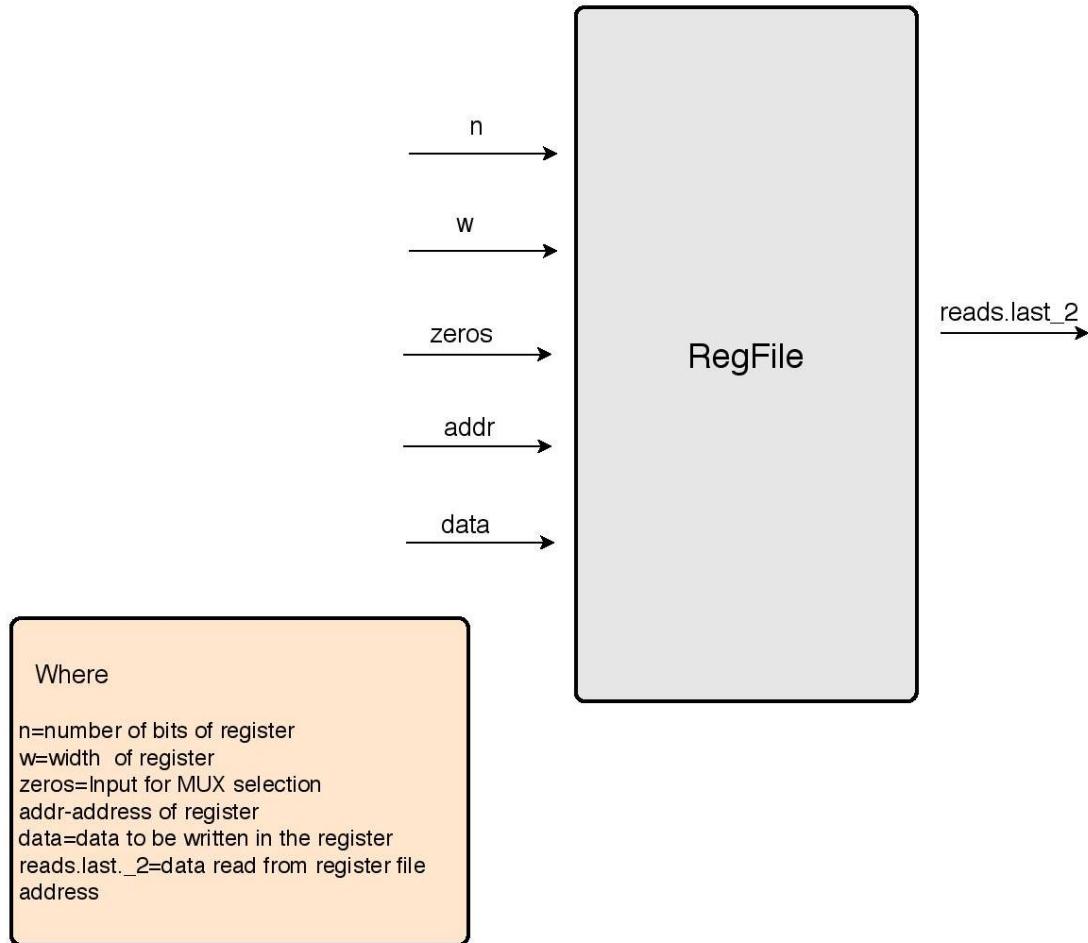
ImmGen sel=selections bits for selection different types of instruction inst=instruction coming from instruction memory	RegFile n=number of bits of register w=width of register zeros=Input for MUX selection addr=address of register data=data to be written in the register reads.last_2=data read from register file address	ScoreBoard n=number of bits of scoreboard zeros=Input for MUX selection addr=address from which read and set scoreboard en=enable pin of scoreboard which is used for setting and clearing scoreboard	RVC in=input of instruction rvc=Output of rvc instruction [Bool] out=Output of compressed instruction	IDDecode Legal=Has type Bool, it tells whether instruction is legal or not fp=Has type Bool, it tells whether instruction is of floating-point or not rocc=Has type Bool, has type Bool branch=Has type Bool, it tells whether instruction is of branch or not jal=Has type Bool, it tells whether instruction is of jump and link (jal) or not jalr=Has type Bool, it tells whether instruction is of jump and link return (jalr) or not rfs1=Has type Bool, it tells whether instruction has source register 1 or not rfs2=Has type Bool, it tells whether instruction has source register 2 or not rfs3=Has type Bool, it tells whether instruction has source register 3 or not wfd=Has type Bool, it tells whether instruction has write back flag alu_dw=Has type Bool, this variable tells Alu data width alu_fn=Has type Bool, this variable returns AluOp bits mem=Has type Bool, it tells whether instruction is of memory type mem_cmd=Has type Bool, this variable returns bits which can adjust memory size mul=Has type Bool, used later in List of BitPat for instruction comparing div=Has type Bool, used later in List of BitPat for instruction comparing wxd=Has type Bool, used later in List of BitPat for instruction comparing csr=Has type Bool, used later in List of BitPat for instruction comparing amo=Has type Bool, this will tell if instruction is of multiplication dp=Has type Bool, this will tell if instruction is of division xwd=Has type Bool, it tells whether instruction has write back flag dp=Has type Bool, used later in List of BitPat for instruction comparing csr=Has type Bool, used later in List of BitPat for instruction comparing fence_i=Has type input as Bool fence_o=Has type input as Bool amo=Has type input as Bool div=Has type input as Bool dp=Has type input as Bool inst=Instruction will be output
--	--	--	---	--

The Decode Stage contains the following Modules.

- Reg File
- Immediate Generation
- IDDecode
- RVC
- ScoreBoard

REG FILE

Block Diagram:



DEEP DIVE INTO CODE

Explanation (By Means of Flowcharts) :

```
class RegFile(n: Int, w: Int, zero: Boolean = false) {
    val rf = Mem(n, UInt(width = w))
    private def access(addr: UInt) = rf(~addr(log2Up(n)-1, 0))
    private val reads = ArrayBuffer[(UInt, UInt)]()
    private var canRead = true
    def read(addr: UInt) = {
        require(canRead)
```

explanation

Class RegFile is initiated having three parameters n and w having type Int and zero having type Boolean (false by default).

n in this case number of bits of register file

w is the width of register file

Mem function uses to create memory where n= number of bits, w= width of memory. First, memory rf is created. Size of that memory let's suppose is 31 data of type UInt width of 10, starting from 0 up to 30.

When we compute log2Up (n)-1 we get 4, and we have something like this ~ addr (4, 0). This gives us last five bits of addr.

Register file uses a little trick where it reverses the order of the registers physically compared to the RISC-V", that's why we use ~addr. And at least rf(~addr) gives us back what is in that memory location.

This is implemented in this way to provide adequate memory access. Take a look what would be if we try to get data from memory location that we don't have in our memory. If method access was called in this way access(42)

Memory location on 41th place is accessed, but there's only 31 memory location (30 is top). 42 binary is 101010.

This would return us 10101 or 21 in decimal. Because order of registers is reversed this is 10th memory location (we try to access 41th but only have 31, 41 - 31 is 10).

In order to access log2up we have to import this function

```
Import chisel3.util{Cat, log2Up, log2Ceil, log2Floor, Log2, Decoupled,
Enum, Valid, Pipe}
```

An ArrayBuffer is like an Array, except that we can additionally add and remove elements from the beginning and end of the sequence.

To use an ArrayBuffer, we must first import it from the mutable collections package:

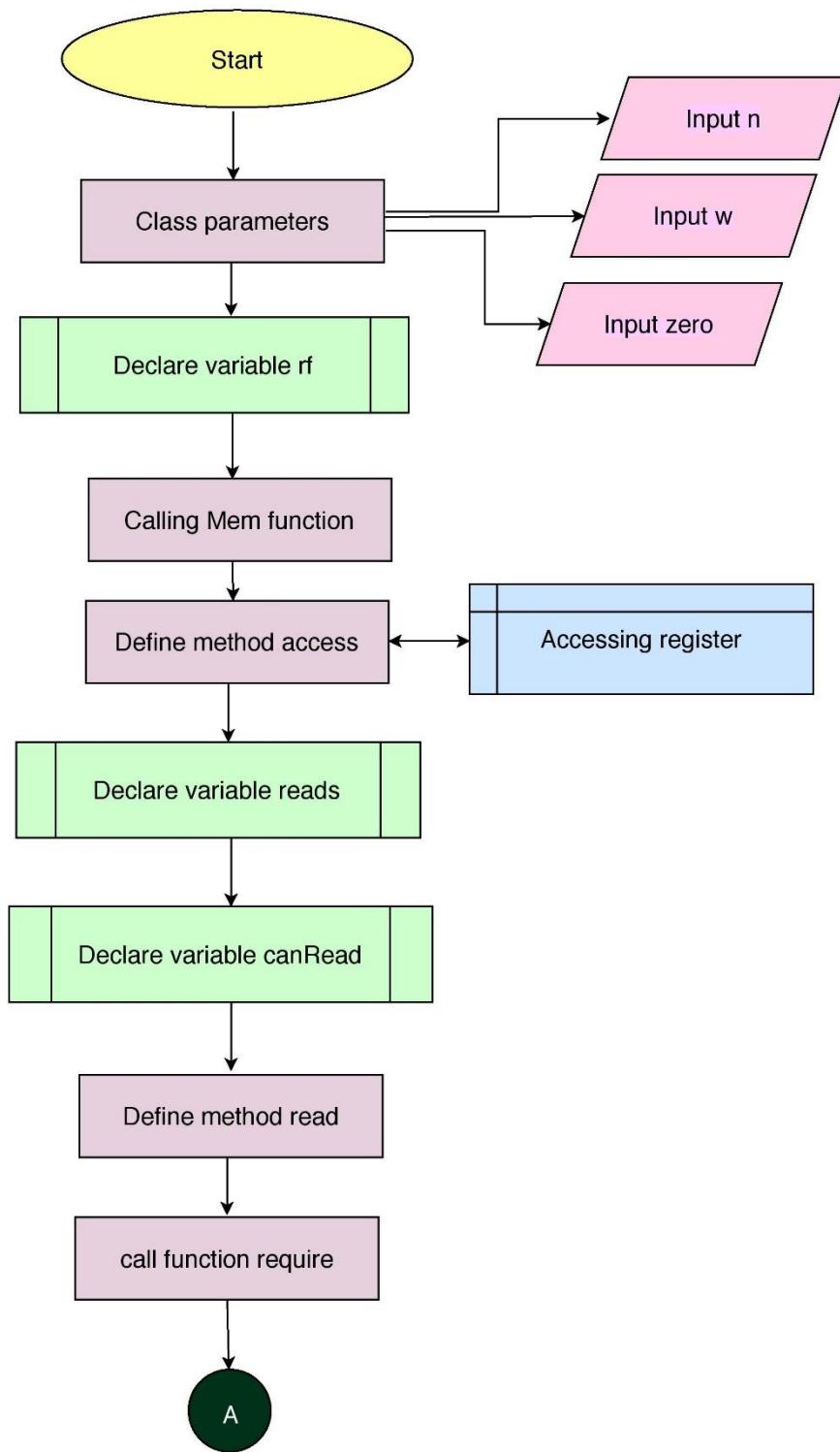
```
import scala.collection.mutable.ArrayBuffer .
```

In this way we instantiate buffer which contains tuples (pairs) that holding two elements of chisel type UInt (unsigned integer). This Line is filling up buffer with tuples created by using -> operator (key -> value, creates a tuple (key, value)).

This variable is mutable when we try to read from register file canRead will be true and for write state canRead will be false

The function read is defined and takes addr as input

Require: This method has an innovative mode of dealing with the problem. It usually blames the method caller if a certain condition is not satisfied. If the condition is satisfied, the program goes on executing, if it does not, it throws an exception.



```

require(canRead)
  reads += addr -> Wire(UInt())
  reads.last._2 := Mux(Bool(zero) && addr === UInt(0), UInt(0),
access(addr))
  reads.last._2
}
def write(addr: UInt, data: UInt) = {
  canRead = false
  when (addr /= UInt(0)) {
    access(addr) := data
  }
}

```

 explanation

Wire(UInt()) // width is inferred. The wire width inferred as addr value is not constant if it gets larger than wire width that problem arises
reads=reads+addr->Wire(UInt())

Mux(sel, in True, in False)

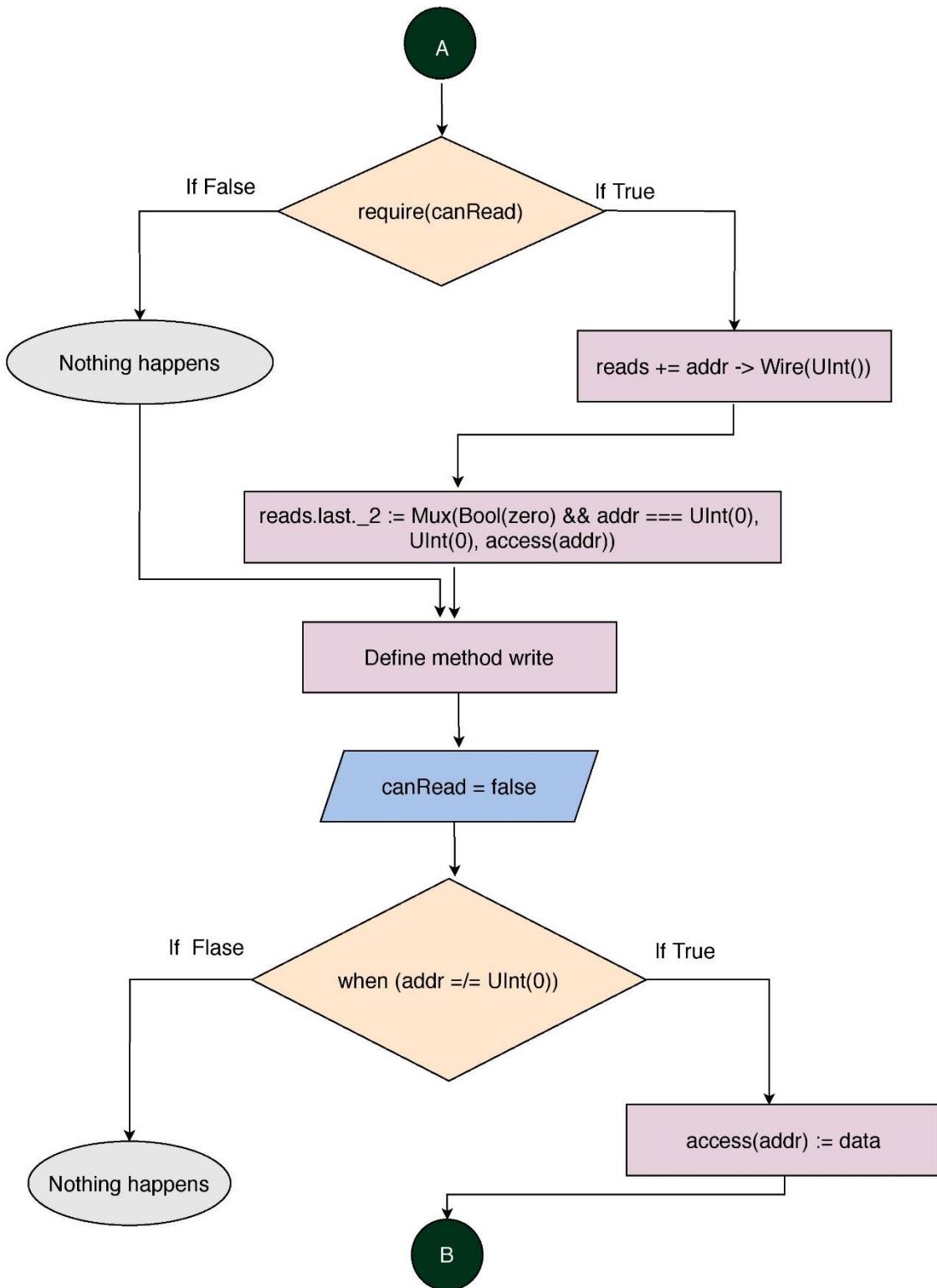
Two-input mux where sel is a Bool

By default we set zero as false. We have condition of And if zero and addr is equals to 0 first condition will be selected otherwise second condition access(addr) will be chosen)

A method write is defined which takes two input 1st one is addr and second is data

For write method canRead will be false

We have our condition if addr is not equals to UInt(0) we wired our input data to access (addr) method.



```

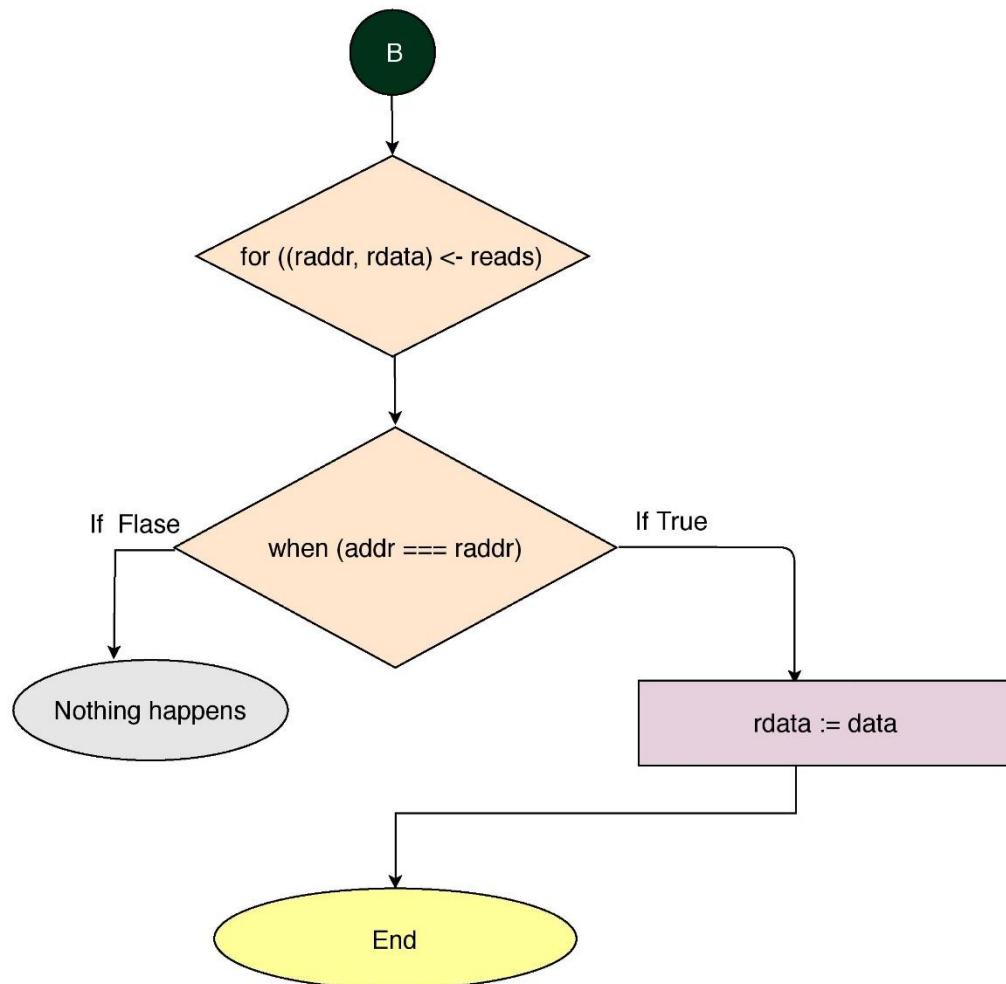
for ((raddr, rdata) <- reads)
    when (addr === raddr) { rdata := data }
}
}
}

```

explanation

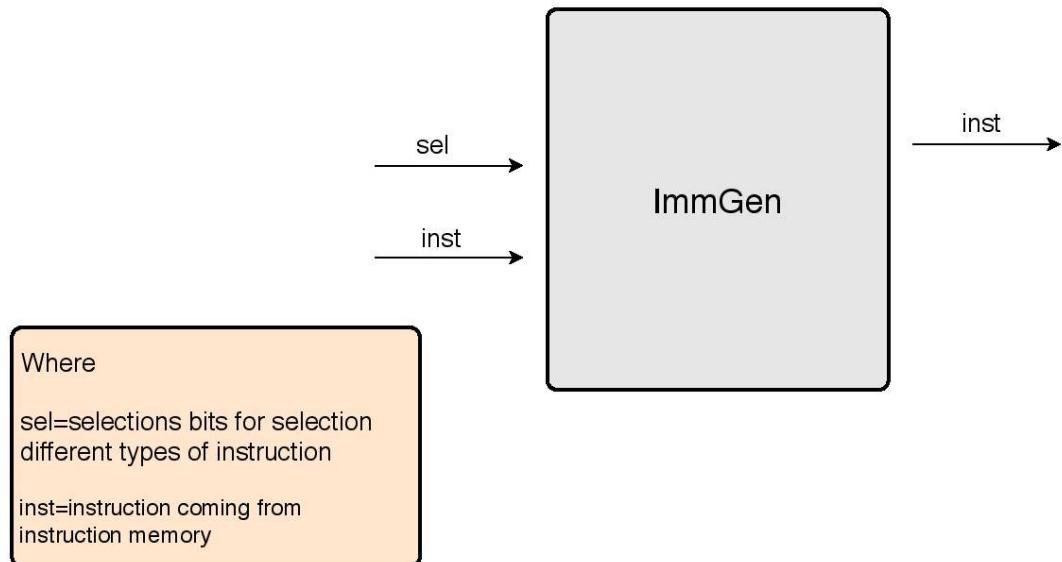
In for loop raddr and rdata are variables of for loop which move buffer values in these variables through reads.

When addr is equals to raddr, data is wired to rdata.



Immediate Generation

Block Diagram:



DEEP DIVE INTO CODE

Explanation (By Means of Flowcharts) :

```
object ImmGen {
def apply(sel: UInt, inst: UInt) = {
  val sign = Mux(sel === IMM_Z, SInt(0), inst(31).asSInt)
  val b30_20 = Mux(sel === IMM_U, inst(30,20).asSInt, sign)
  val b19_12 = Mux(sel /= IMM_U && sel /= IMM_UJ, sign, inst(19,12).asSInt)
```

explanation

The object ImmGen is created and a function is defined named apply which takes two parameters

sel=select lines for MUX

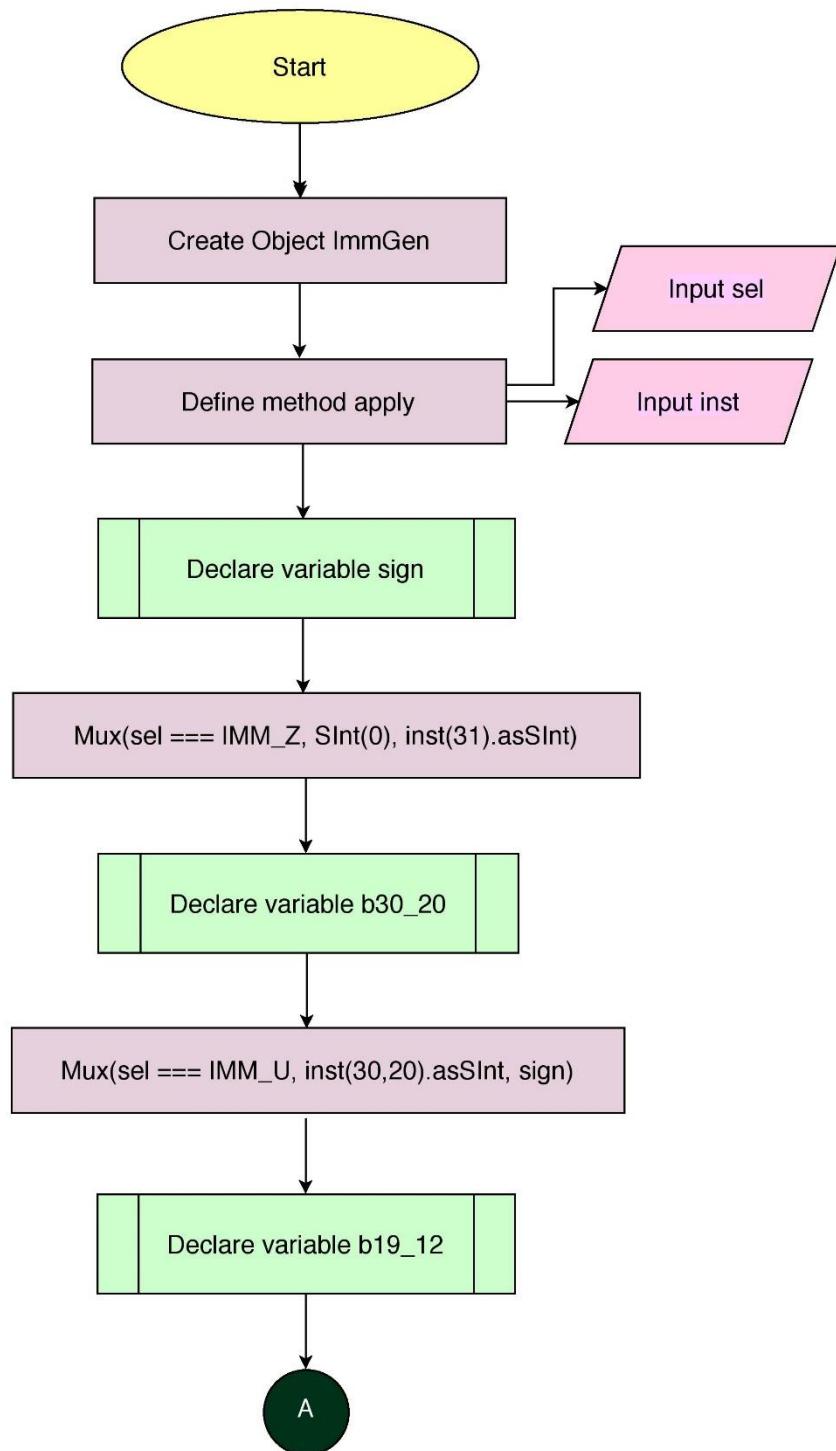
inst=instruction coming from instruction Memory

Then immutable variable sign is created and Mux is used in it .If our sel is equal to IMM_Z (IMM_Z in this condition is pointing to 0 offset, it will select SINT(0) otherwise it will select 31th bit from the instruction)

An immutable variable b30_20 is created in which immediate bits for Utype instruction is selected. If sel is equals to IMM_U(IMM_U in our case is pointing as selection bits for MUX)

If the instruction is of U type Mux will extract bits from 20 to 30 from the instruction otherwise Mux will select sign variable which we declared above

An immutable variable b19_12 is created in which we uses Mux and compared two immediate of U and UJ type instruction if our sel is not pointing towards IMM_U and IMM_UJ sign variable will be selected which we declared above, otherwise if Mux condition false we will extract bits from 12 to 19 from our instructions and converts it in Signed integer.



```

val b19_12 = Mux(sel == IMM_U && sel == IMM_UJ, sign, inst(19,12).asSInt)

val b11 = Mux(sel === IMM_U || sel === IMM_Z, SInt(0),
Mux(sel === IMM_UJ, inst(20).asSInt,
Mux(sel === IMM_SB, inst(7).asSInt, sign)))

val b10_5 = Mux(sel === IMM_U || sel === IMM_Z, Bits(0), inst(30,25))

```

 explanation

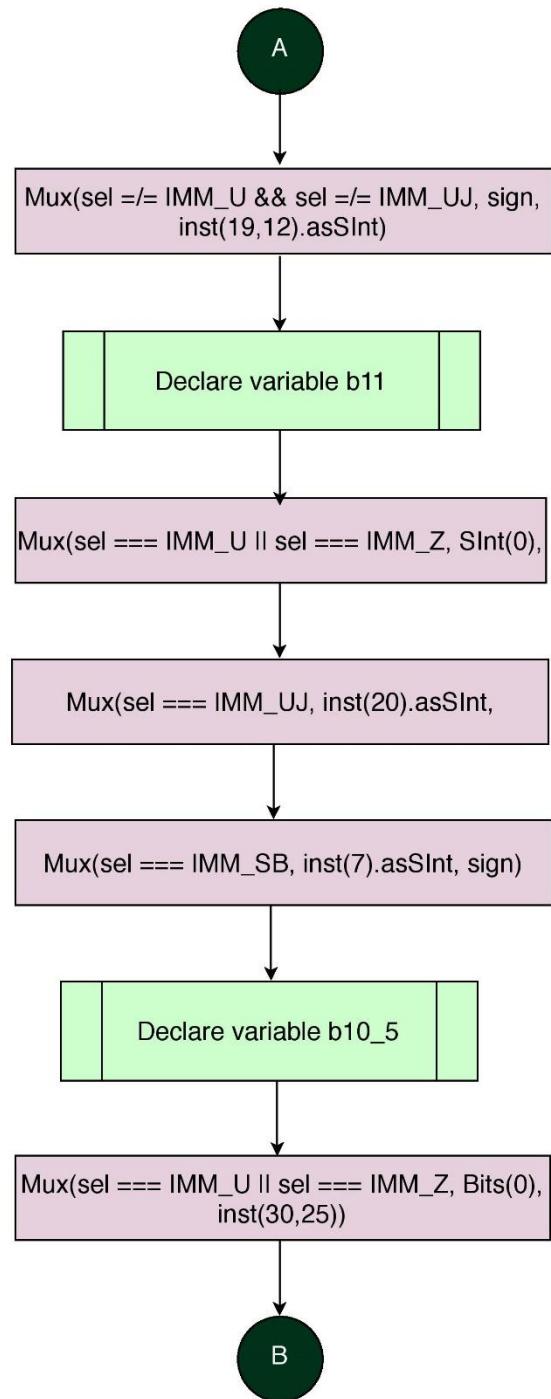
An immutable variable b11 is declared which select one bit from different Mux conditions on the basis of sel bits

The 1st Mux has OR condition between sel bits weather it points to IMM_U OR it points towards IMM_Z sel. If any of the above condition satisfies we will select Signed Integer(0)

If 1st Mux condition lies in false There's a 2nd Mux which will compare sel bits with IMM_UJ (immediate of Unconditional jump) if sel is equal to IMM_UJ we will select 20th bit from the instruction and convert it SignedInteger

If sel is not equals to IMM_UJ which means selection bits is not the same as IMM_UJ sel bits then there's a 3rd Mux comes into play in which we compare sel with IMM_SB(immediate of Store Branch) if 3rd Mux Condition satisfies we extract 7th bit from the instruction and converts it in SignedInteger otherwise we will select sign variable which we declared above

An immutable variable b10_5 is created for selection of bits from 25 to 30 .The Mux condition which compare sel with IMM_U OR sel with IMM_Z if any of the condition true we select Bits(0) otherwise we extract instruction bits from 25 to 30.



```

val b4_1 = Mux(sel === IMM_U, Bits(0),
Mux(sel === IMM_S || sel === IMM_SB, inst(11,8),
Mux(sel === IMM_Z, inst(19,16), inst(24,21)))) 

val b0 = Mux(sel === IMM_S, inst(7),
Mux(sel === IMM_I, inst(20),
Mux(sel === IMM_Z, inst(15), Bits(0))))

```

 explanation

An immutable variable b4_1 is declared for selection of bits from 8 to 11 or 21 to 24. The Mux condition compares sel with IMM_U if this condition satisfies we select Bits(0) otherwise

There's a 2nd Mux condition which compares sel with IMM_S and IMM_SB if any of these condition true, then bits 8 to 11 are extracted from our instruction

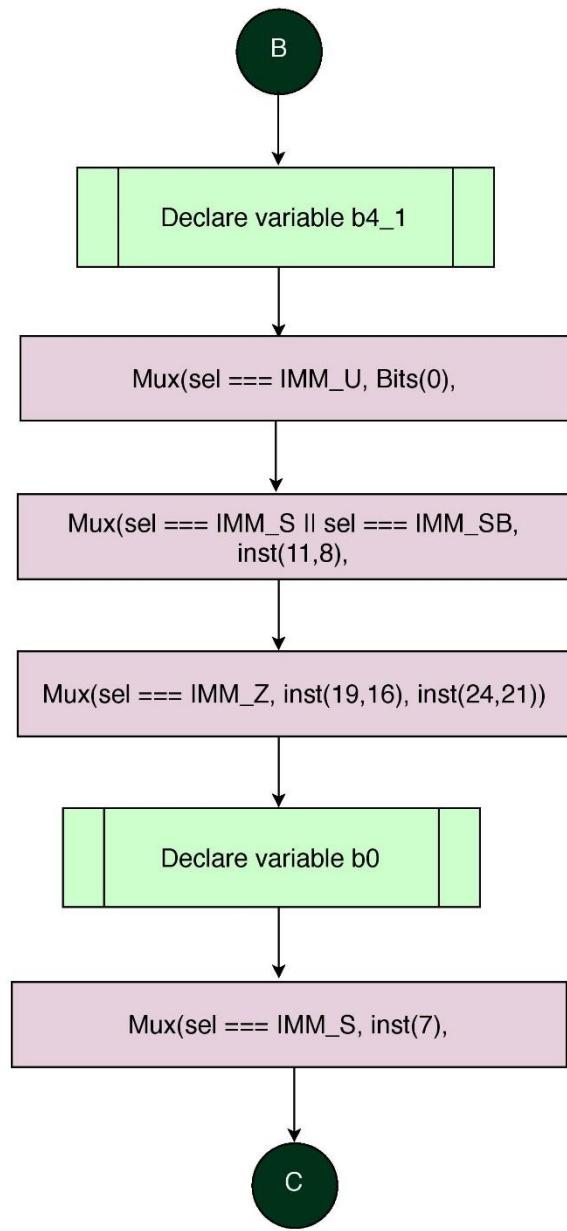
If 2nd Mux condition doesn't satisfies there's a 3rd Mux which will compare sel with IMM_Z, if this condition satisfies we extract bits from 16 to 19 from our instruction otherwise we extract bits from 21 to 24 from our instruction

We create a variable b0 which is selecting one bits from different Muxes condition

1st Mux will compare sel with IMM_S(immediate of store type instruction) if this condition satisfies we select 7th bit from the instruction

If 1st Mux condition fails we uses 2nd Mux condition which compare sel with IMM_I(immediate of I type instruction) if it satisfies we extract 20th bit from the instruction

If 2nd Mux doesn't satisfies there's a 3rd Mux which compares sel with IMM_Z if this condition satisfies we extract bit 15 from our instruction otherwise bits (0) will be selected.



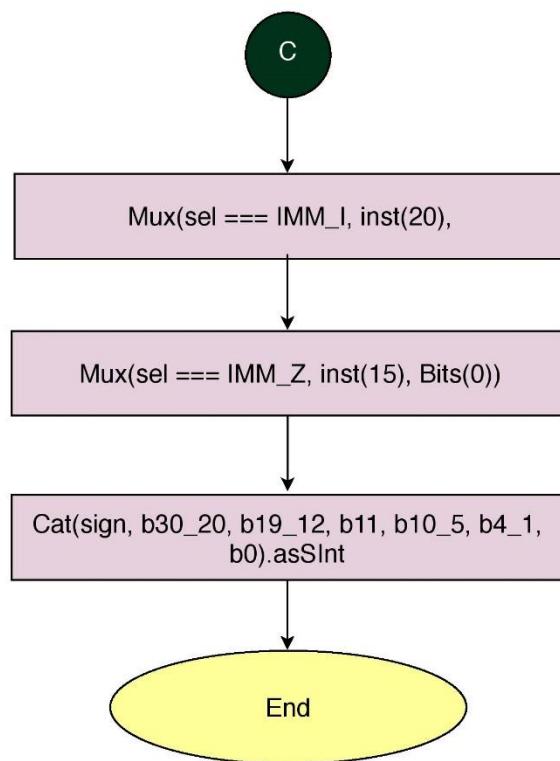
```

Cat(sign, b30_20, b19_12, b11, b10_5, b4_1, b0).asSInt
}
}

```

explanation

The Cat function is used to concatenate all the bits which we extracted from different immutable variables and then convert them in Signed-Integer.



Instruction Decode (IDecode)

DEEP DIVE INTO CODE

Explanation (By Means of Flowcharts) :

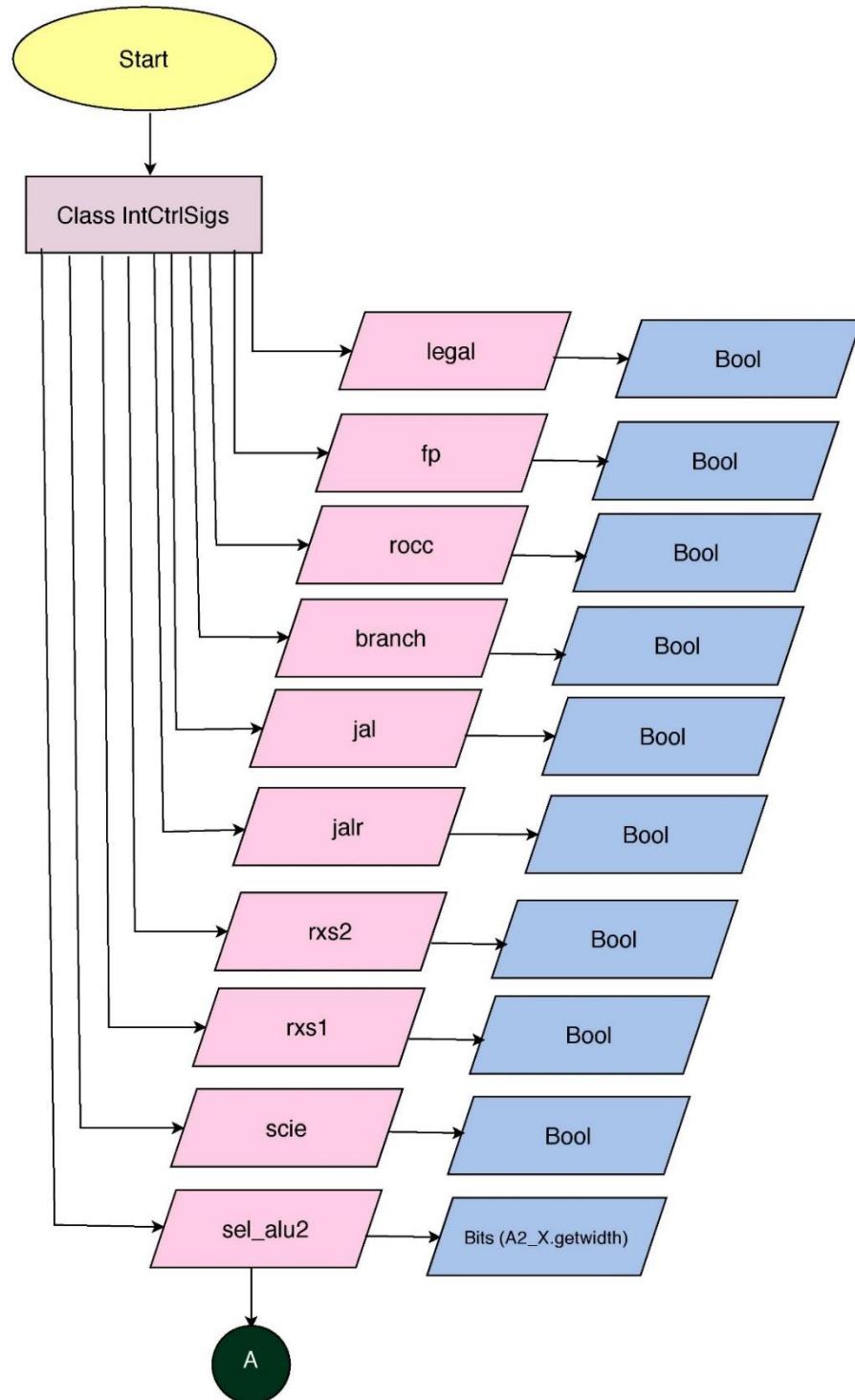
```
abstract trait DecodeConstants extends HasCoreParameters {
    val table: Array[(BitPat, List[BitPat])]
}

class IntCtrlSigs extends Bundle {
    val legal = Bool()
    val fp = Bool()
    val rocc = Bool()
    val branch = Bool()
    val jal = Bool()
    val jalr = Bool()
    val rxs2 = Bool()
    val rxs1 = Bool()
    val scie = Bool()
    val sel_alu2 = Bits(width = A2_X.getWidth)
}
```

explanation

Control Signals	Description
legal	Has type Bool, it tells whether instruction is legal or not
fp	Has type Bool, it tells whether instruction is of floating-point or not
rocc	rocc variable has type Bool
branch	Has type Bool, it tells whether instruction is of branch or not
jal	Has type Bool, it tells whether instruction is of jump and link (jal) or not
jalr	Has type Bool, it tells whether instruction is of jump and link return (jalr) or not

rxs2	Has type Bool, it tells whether instruction has source register 2 or not
rxs1	Has type Bool, it tells whether instruction has source register 1 or not
scie	scie variable has type Bool



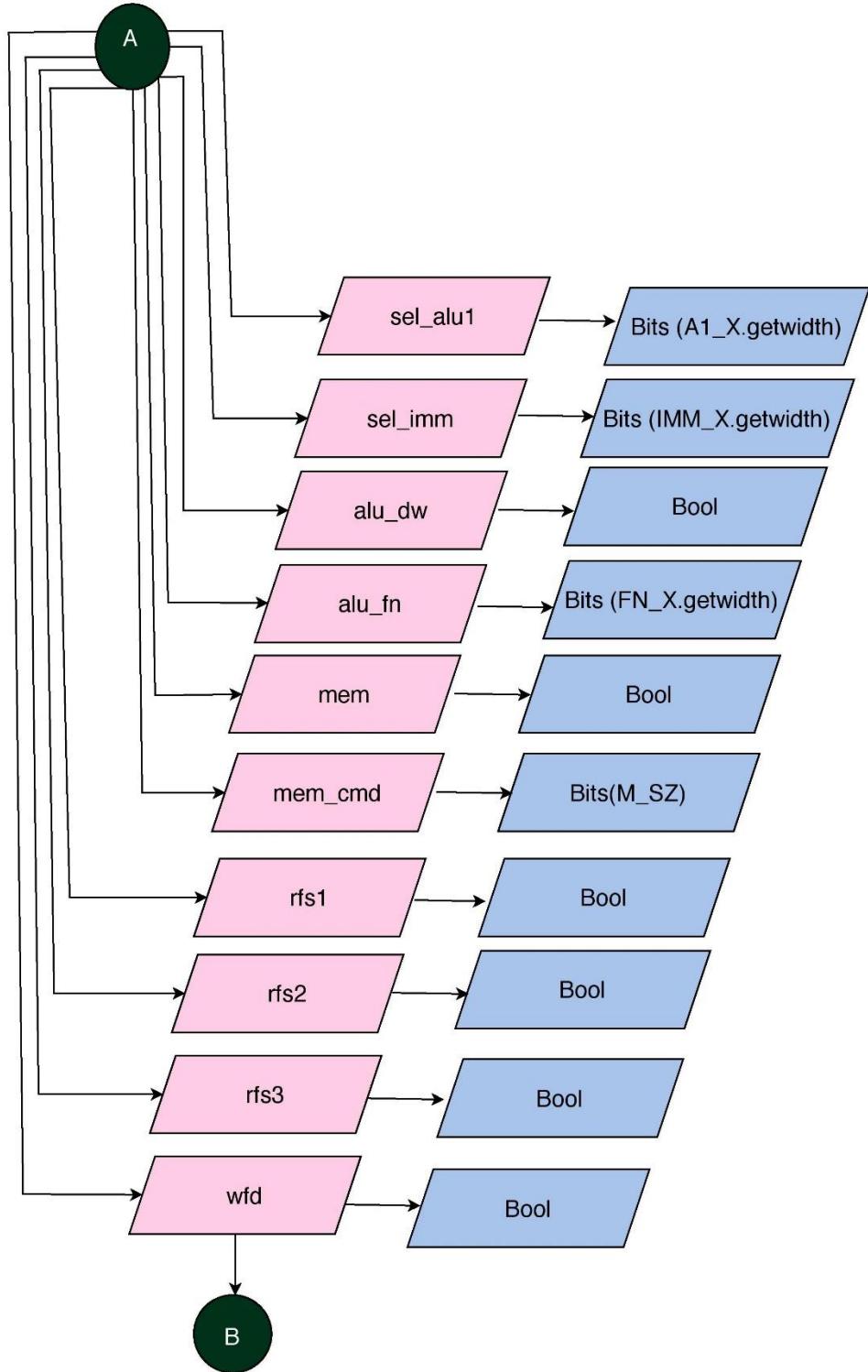
```

val sel_alu1 = Bits(width = A1_X.getWidth)
val sel_imm = Bits(width = IMM_X.getWidth)
val alu_dw = Bool()
val alu_fn = Bits(width = FN_X.getWidth)
val mem = Bool()
val mem_cmd = Bits(width = M_SZ)
val rfs1 = Bool()
val rfs2 = Bool()
val rfs3 = Bool()
val wfd = Bool()

```

explanation

Control Signals	Description
sel_alu1	This variable is used to select operand A of ALU by using bits
sel_imm	This variable is used to select immediate bits
alu_dw	Has type Bool, this variable tells Alu data width
alu_fn	This variable returns Aluop bits
mem	Memory variable having type as Bool
mem_cmd	This variable returns bits which can adjust memory size
rfs1	Has type Bool, used later in List of BitPat for instruction comparing
rfs2	Has type Bool, used later in List of BitPat for instruction comparing
rfs3	Has type Bool, used later in List of BitPat for instruction comparing
wfd	Has type Bool, used later in List of BitPat for instruction comparing



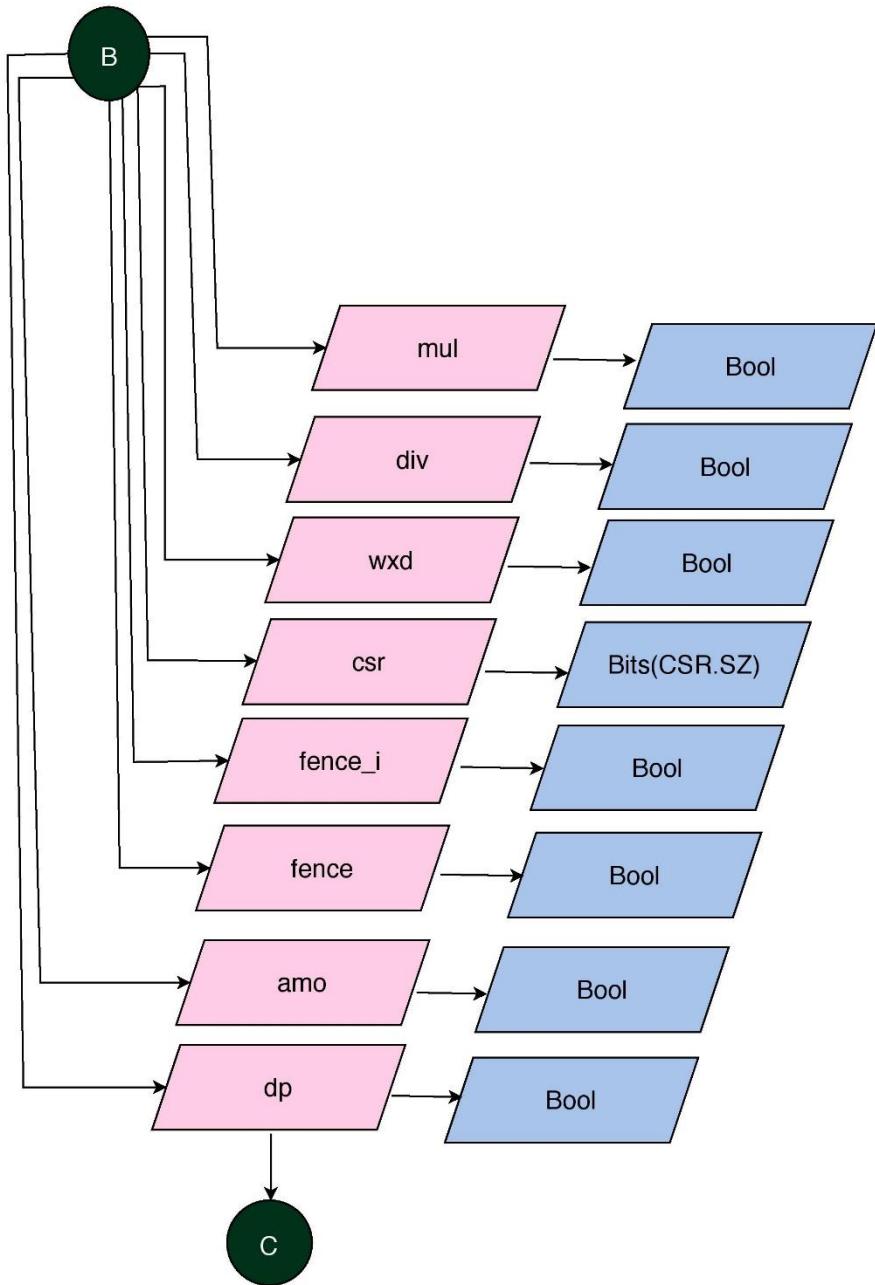
```

val mul = Bool()
val div = Bool()
val wxd = Bool()
val csr = Bits(width = CSR.SZ)
val fence_i = Bool()
val fence = Bool()
val amo = Bool()
val dp = Bool()

```

— explanation —

Control Signals	Description
mul	Has type Bool, this will tell if instruction is of multiplication
div	Has type Bool, this will tell if instruction is of division
wxd	Has type Bool, it tells whether instruction has write destination register or not
csr	Has type Bits as input
fence_i	Has type input as Bool
fence	Has type input as Bool
amo	Has type input as Bool
dp	Has type input as Bool



```

def default: List[BitPat] =
  List(N,X,X,X,X,X,X,X,X,A2_X,      A1_X,      IMM_X, DW_X, FN_X,      N,M_X,
X,X,X,X,X,X,CSR.X,X,X,X,X)

def decode(inst: UInt, table: Iterable[(BitPat, List[BitPat])]) = {
  val decoder = DecodeLogic(inst, default, table)
  val sigs = Seq(legal, fp, rocc, branch, jal, jalr, rxs2, rxs1, scie,
sel_alu2, sel_alu1, sel_imm, alu_dw, alu_fn, mem, mem_cmd, rfs1, rfs2,
rfs3, wfd, mul, div, wxd, csr, fence_i, fence, amo, dp)
  sigs zip decoder map {case(s,d) => s := d}
  this
}

}

class IDecode(implicit val p: Parameters) extends DecodeConstants
{
  val table: Array[(BitPat, List[BitPat])] = Array(
    BNE-> List(Y,N,N,Y,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_SB,DW_X,
FN_SNE,      N,M_X,      N,N,N,N,N,N,CSR.N,N,N,N,N),
    BEQ-> List(Y,N,N,Y,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_SB,DW_X,
FN_SEQ,      N,M_X,      N,N,N,N,N,N,CSR.N,N,N,N,N),
    BLT-> List(Y,N,N,Y,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_SB,DW_X,
FN_SLT,      N,M_X,      N,N,N,N,N,N,CSR.N,N,N,N,N),
  )
}

```

 explanation

A method default is defined in which, a List is created by using different variables, defined above in the trait. Later this List is used in Bit Pattern for the instruction matching

A decode method is defined which takes instruction and table as its input parameter. The table is taken as Iterable trait which means (This is a base trait for all Scala collections that define an iterator method to step through one-by-one the collection's elements)

A variable named decoder is created and takes DecodeLogic object as input which consist of instruct, default and table

sigs variable is created in which we arranged all the variables we define above in the class IntCtrlSigs by using Seq function in Scala.

zip function is used to merge sigs variable collection and decoder variable collection together.

map method is used, which takes a predicate function and applies it to every element in the collection. It creates a new collection with the result of the predicate function applied to each and every element of the collection

A case keyword is used for matching of sigs(s) and decoder(d) variable and if that match we wired decoder output with sigs variable.

A class IDecode (instruction Decode) is created, which is extended by the trait DecodeConstants.

table variable is created which returns array.

Instructions

BNE (Branch not equal)

BEQ (Branch equal)

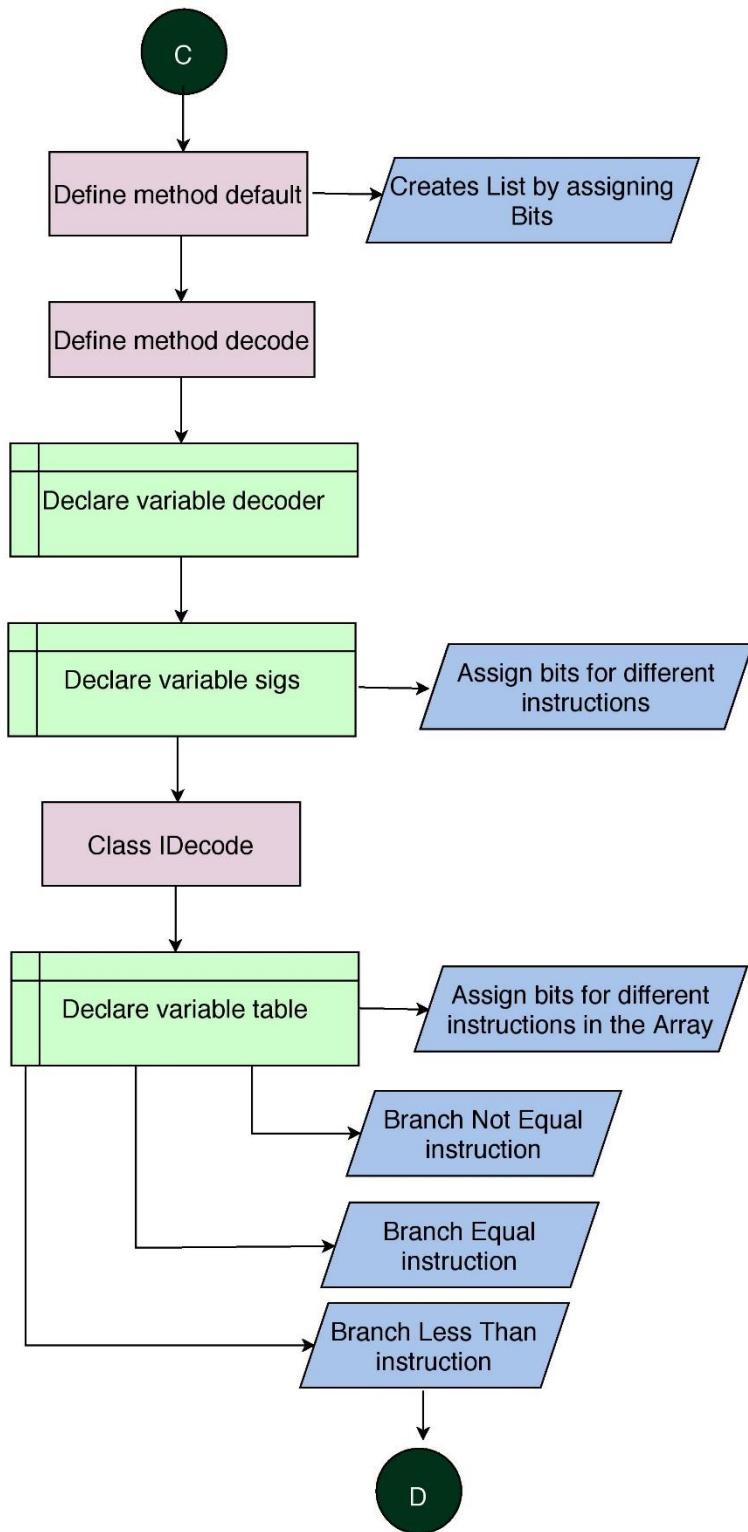
BLT (Branch less than)

There are Y and N which are used if there is true, then, there will be Y otherwise N. sigs variable is used for sequence in this List.

Example

In these branch instruction we have 1st, 4th, 7th and 8th element position as Y which means we have legal, branch, register file source 1 and register file source 2 as true. We have 2nd and 3rd position as N which means we don't have fp(floating point) instruction and rocc instruction.

We have IMM_SB (immediate store branch), DW_X(data width of Alu) and Aluop bits as input of List.



ROCKET-CHIP Micro Architecture Specification Document

BLTU-> List(Y,N,N,Y,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_SB,DW_X,
FN_SLTU, N,M_X, N,N,N,N,N,N,CSR.N,N,N,N,N),

BGE-> List(Y,N,N,Y,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_SB,DW_X,
FN_SGE, N,M_X, N,N,N,N,N,N,CSR.N,N,N,N,N),

BGEU-> List(Y,N,N,Y,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_SB,DW_X,
FN_SGEU, N,M_X, N,N,N,N,N,N,CSR.N,N,N,N,N),

JAL-> List(Y,N,N,N,Y,N,N,N,A2_SIZE,A1_PC,
IMM_UJ,DW_XPR,FN_ADD, N,M_X, N,N,N,N,N,Y,CSR.N,N,N,N,N),

JALR-> List(Y,N,N,N,N,Y,N,Y,N,A2_IMM, A1_RS1, IMM_I,
DW_XPR,FN_ADD, N,M_X, N,N,N,N,N,Y,CSR.N,N,N,N,N),

AUIPC-> List(Y,N,N,N,N,N,N,N,A2_IMM, A1_PC, IMM_U,
DW_XPR,FN_ADD, N,M_X, N,N,N,N,N,Y,CSR.N,N,N,N,N),

LB-> List(Y,N,N,N,N,N,N,Y,N,A2_IMM, A1_RS1, IMM_I,
DW_XPR,FN_ADD, Y,M_XRD, N,N,N,N,N,Y,CSR.N,N,N,N,N),

LH-> List(Y,N,N,N,N,N,N,Y,N,A2_IMM, A1_RS1, IMM_I,
DW_XPR,FN_ADD, Y,M_XRD, N,N,N,N,N,Y,CSR.N,N,N,N,N),

LW-> List(Y,N,N,N,N,N,N,Y,N,A2_IMM, A1_RS1, IMM_I,
DW_XPR,FN_ADD, Y,M_XRD, N,N,N,N,N,Y,CSR.N,N,N,N,N),

LBU-> List(Y,N,N,N,N,N,Y,N,A2_IMM, A1_RS1, IMM_I,
DW_XPR,FN_ADD, Y,M_XRD, N,N,N,N,N,Y,CSR.N,N,N,N,N),

— explanation —

Instructions

BLTU(Branch less than unsigned)

BGE(Branch greater than equal)

BGEU(Branch greater than equal unsigned)

JAL (Jump and Link)

JALR(Jump and Link return)

AUIPC(Add Upper Immediate to Program Counter)

LB (Load Byte)

LH (Load Half)

LW (Load Word)

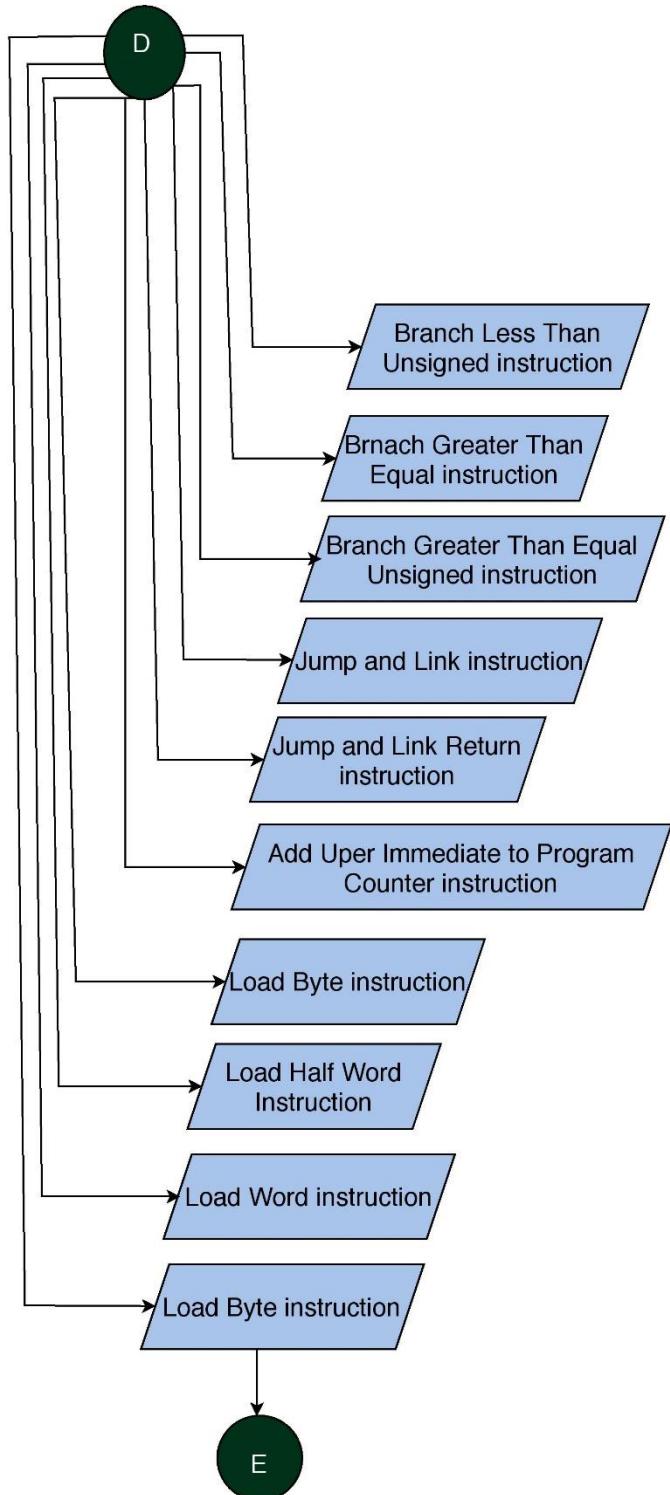
LBU (Load Byte Unsigned)

There are Y and N which are used if there is true then, there will be Y otherwise N. sigs variable is used for sequence in this List.

Example:

In these instructions we have 1st element position as Y which means we have legal instruction. We have 2nd, 3rd and 4th position as N which means we don't have fp(floating point) instruction, rocc instruction and branch instruction.

We have IMM_UJ(immediate of unconditional jump),IMM_I(immediate of I format instruction),IMM_U (immediate of unconditional),DW_X(data width of Alu) and Aluop bits as input of List.



ROCKET-CHIP Micro Architecture Specification Document

LHU-> List(Y,N,N,N,N,N,N,Y,N,A2_IMM, A1_RS1, IMM_I, DW_XPR, FN_ADD,
Y,M_XRD, N,N,N,N,N,N,Y,CSR.N,N,N,N,N),

SB-> List(Y,N,N,N,N,N,Y,Y,N,A2_IMM, A1_RS1, IMM_S,
DW_XPR, FN_ADD, Y,M_XWR, N,N,N,N,N,N,N,CSR.N,N,N,N,N),

SH-> List(Y,N,N,N,N,N,Y,Y,N,A2_IMM, A1_RS1, IMM_S,
DW_XPR, FN_ADD, Y,M_XWR, N,N,N,N,N,N,N,CSR.N,N,N,N,N),

SW-> List(Y,N,N,N,N,N,Y,Y,N,A2_IMM, A1_RS1, IMM_S,
DW_XPR, FN_ADD, Y,M_XWR, N,N,N,N,N,N,N,CSR.N,N,N,N,N),

LUI-> List(Y,N,N,N,N,N,N,N,A2_IMM, A1_ZERO, IMM_U,
DW_XPR, FN_ADD, N,M_X, N,N,N,N,N,Y,CSR.N,N,N,N,N),

ADDI-> List(Y,N,N,N,N,N,Y,N,A2_IMM, A1_RS1, IMM_I,
DW_XPR, FN_ADD, N,M_X, N,N,N,N,N,Y,CSR.N,N,N,N,N),

SLTI -> List(Y,N,N,N,N,N,Y,N,A2_IMM, A1_RS1, IMM_I,
DW_XPR, FN_SLT, N,M_X, N,N,N,N,N,N,Y,CSR.N,N,N,N,N),

SLTIU-> List(Y,N,N,N,N,N,Y,N,A2_IMM, A1_RS1, IMM_I,
DW_XPR, FN_SLTU, N,M_X, N,N,N,N,N,N,Y,CSR.N,N,N,N,N),

ANDI-> List(Y,N,N,N,N,N,Y,N,A2_IMM, A1_RS1, IMM_I,
DW_XPR, FN_AND, N,M_X, N,N,N,N,N,N,Y,CSR.N,N,N,N,N),

ORI-> List(Y,N,N,N,N,N,Y,N,A2_IMM, A1_RS1, IMM_I, DW_XPR, FN_OR,
N,M_X, N,N,N,N,N,Y,CSR.N,N,N,N,N),

— explanation —

Instructions

LHU (Load Half Unsigned)

SB (Store Byte)

SH (Store Half)

SW (Store Word)

LUI (Load Upper Immediate)

ADDI (Addition Immediate)

SLTI (Set Less Than Immediate)

SLTIU (Set Less Than Immediate Unsigned)

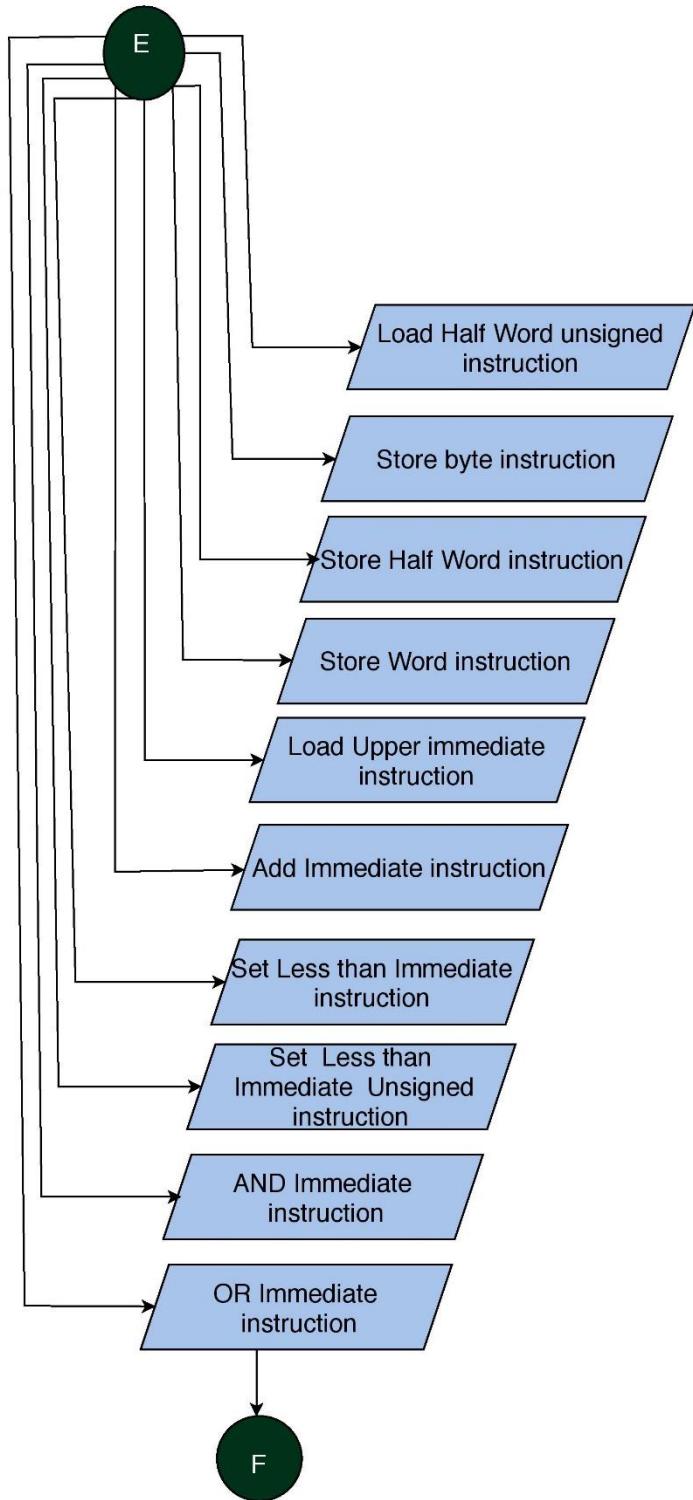
ANDI (And Immediate)

ORI (OR Immediate)

There are Y and N which are used if there is true then, there will be Y otherwise N. sigs variable is used for sequence in this List.

Example

In these load and store instruction we have 1st element position as Y which means we have legal instruction. We have 2nd, 3rd and 4th position as N which means we don't have fp(floating point) instruction, rocc instruction and branch instruction. We have IMM_S(immediate of store),IMM_I(immediate of I format instruction), DW_X(data width of Alu) and Aluop bits as input of List. We have single source register in load instruction and have two source register in store instruction.



ROCKET-CHIP Micro Architecture Specification Document

XORI-> List(Y,N,N,N,N,N,Y,N,A2_IMM, A1_RS1, IMM_I,
DW_XPR, FN_XOR, N,M_X, N,N,N,N,N,N,Y,CSR.N,N,N,N,N),

ADD-> List(Y,N,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X,
DW_XPR, FN_ADD, N,M_X, N,N,N,N,N,N,Y,CSR.N,N,N,N,N),

SUB-> List(Y,N,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X,
DW_XPR, FN_SUB, N,M_X, N,N,N,N,N,N,Y,CSR.N,N,N,N,N),

SLT-> List(Y,N,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X,
DW_XPR, FN_SLT, N,M_X, N,N,N,N,N,N,Y,CSR.N,N,N,N,N),

SLTU-> List(Y,N,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X,
DW_XPR, FN_SLTU, N,M_X, N,N,N,N,N,N,Y,CSR.N,N,N,N,N),

AND-> List(Y,N,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X,
DW_XPR, FN_AND, N,M_X, N,N,N,N,N,N,Y,CSR.N,N,N,N,N),

OR-> List(Y,N,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X, DW_XPR, FN_OR,
N,M_X, N,N,N,N,N,Y,CSR.N,N,N,N,N),

XOR-> List(Y,N,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X,
DW_XPR, FN_XOR, N,M_X, N,N,N,N,N,N,Y,CSR.N,N,N,N,N),

SLL-> List(Y,N,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X, DW_XPR, FN_SL,
N,M_X, N,N,N,N,N,N,Y,CSR.N,N,N,N,N),

SRL-> List(Y,N,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X, DW_XPR, FN_SR,
N,M_X, N,N,N,N,N,N,Y,CSR.N,N,N,N,N),

— explanation —

Instructions

XORI (XOR Immediate)

ADD (Addition)

SUB (Subtraction)

SLT (Set Less Than)

SLTU (Set Less Than Unsigned)

AND (AND Logical Operation)

OR (OR Logical Operation)

XOR (XOR logical Operation)

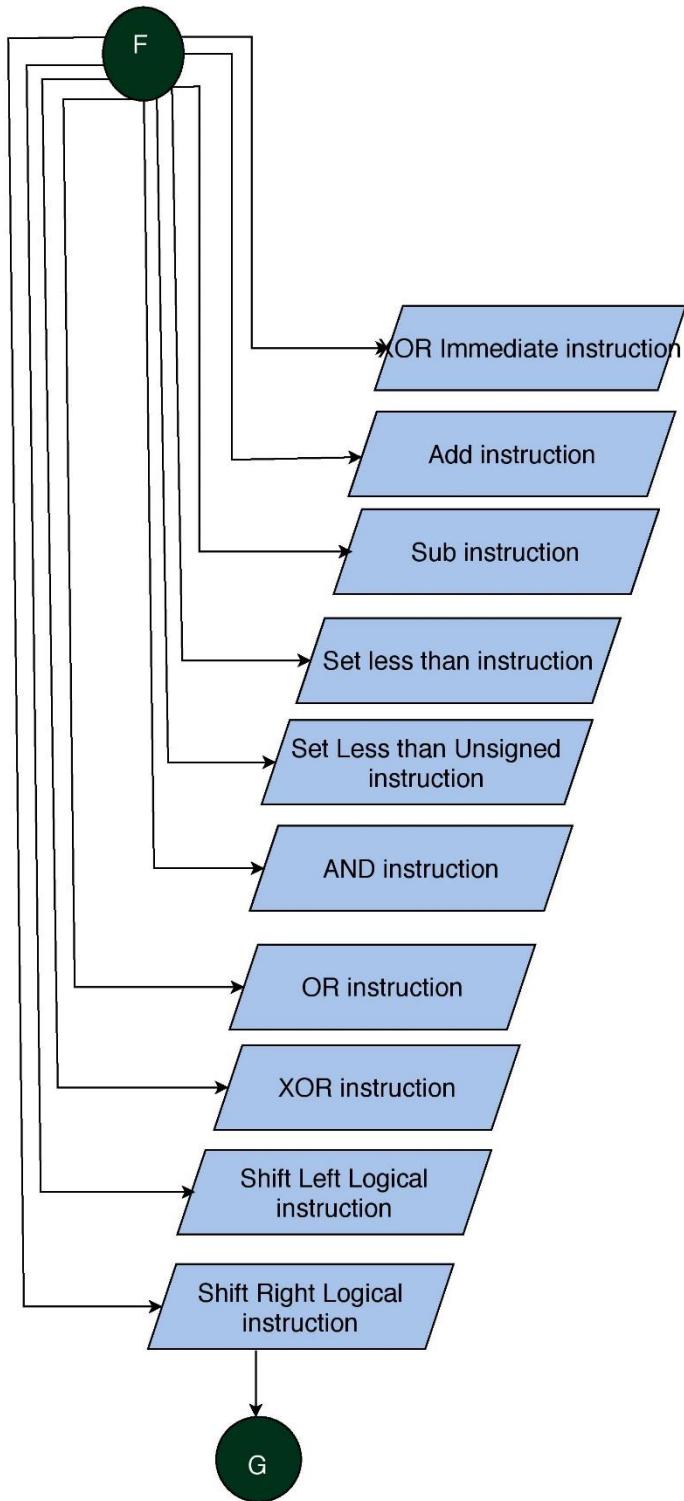
SLL (Shift Left Logical)

SRL (Shift Right Logical)

There are Y and N which are used if there is true then, there will be Y otherwise N. sigs variable is used for sequence in this List.

Example

In these R format instructions we have 1st element position as Y which means we have legal instruction. We have 2nd, 3rd and 4th position as N which means we don't have fp(floating point) instruction, rocc instruction and branch instruction. We have IMM_U(Upper Immediate), IMM_I(immediate of I format instruction), DW_X(data width of Alu) and Aluop bits as input of List.



ROCKET-CHIP Micro Architecture Specification Document

SRA-> List(Y,N,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X,
DW_XPR, FN_SRA, N,M_X, N,N,N,N,N,N,Y,CSR.N,N,N,N,N),

FENCE-> List(Y,N,N,N,N,N,N,N,A2_X, A1_X, IMM_X, DW_X, FN_X,
N,M_X, N,N,N,N,N,N,CSR.N,N,Y,N,N),

SCALL-> List(Y,N,N,N,N,N,N,X,N,A2_X, A1_X, IMM_X, DW_X, FN_X,
N,M_X, N,N,N,N,N,N,CSR.I,N,N,N,N),

SBREAK-> List(Y,N,N,N,N,N,N,X,N,A2_X, A1_X, IMM_X, DW_X, FN_X,
N,M_X, N,N,N,N,N,N,CSR.I,N,N,N,N),

MRET-> List(Y,N,N,N,N,N,N,X,N,A2_X, A1_X, IMM_X, DW_X, FN_X,
N,M_X, N,N,N,N,N,N,CSR.I,N,N,N,N),

WFI-> List(Y,N,N,N,N,N,N,X,N,A2_X, A1_X, IMM_X, DW_X, FN_X,
N,M_X, N,N,N,N,N,N,CSR.I,N,N,N,N),

CEASE-> List(Y,N,N,N,N,N,N,X,N,A2_X, A1_X, IMM_X, DW_X, FN_X,
N,M_X, N,N,N,N,N,N,CSR.I,N,N,N,N),

CSRRW-> List(Y,N,N,N,N,N,N,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR, FN_ADD, N,M_X, N,N,N,N,N,N,Y,CSR.W,N,N,N,N),

CSRRS-> List(Y,N,N,N,N,N,N,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR, FN_ADD, N,M_X, N,N,N,N,N,N,Y,CSR.S,N,N,N,N),

CSRRC-> List(Y,N,N,N,N,N,N,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR, FN_ADD, N,M_X, N,N,N,N,N,N,Y,CSR.C,N,N,N,N),

explanation

Instructions

SRA (Shift Right Arithmetic)

RV32I System Instructions

FENCE (Instruction memory ordering fence (I Format))

SCALL (System call (I Format))

SBREAK(Breakpoint (I Format))

MRET

WFI

CEASE

CSRRW (Read and write CSR (I Format))

CSRRS(Read and set bits in CSR (I Format))

CSRRC (Read and clear bits in CSR (I Format))

CSRRWI (Read and write CSR with immediate (I Format))

CSRRSI (Read and set bits in CSR with immediate (I Format))

CSRRCI (Read and clear bits in CSR with immediate (I Format))

System Instructions

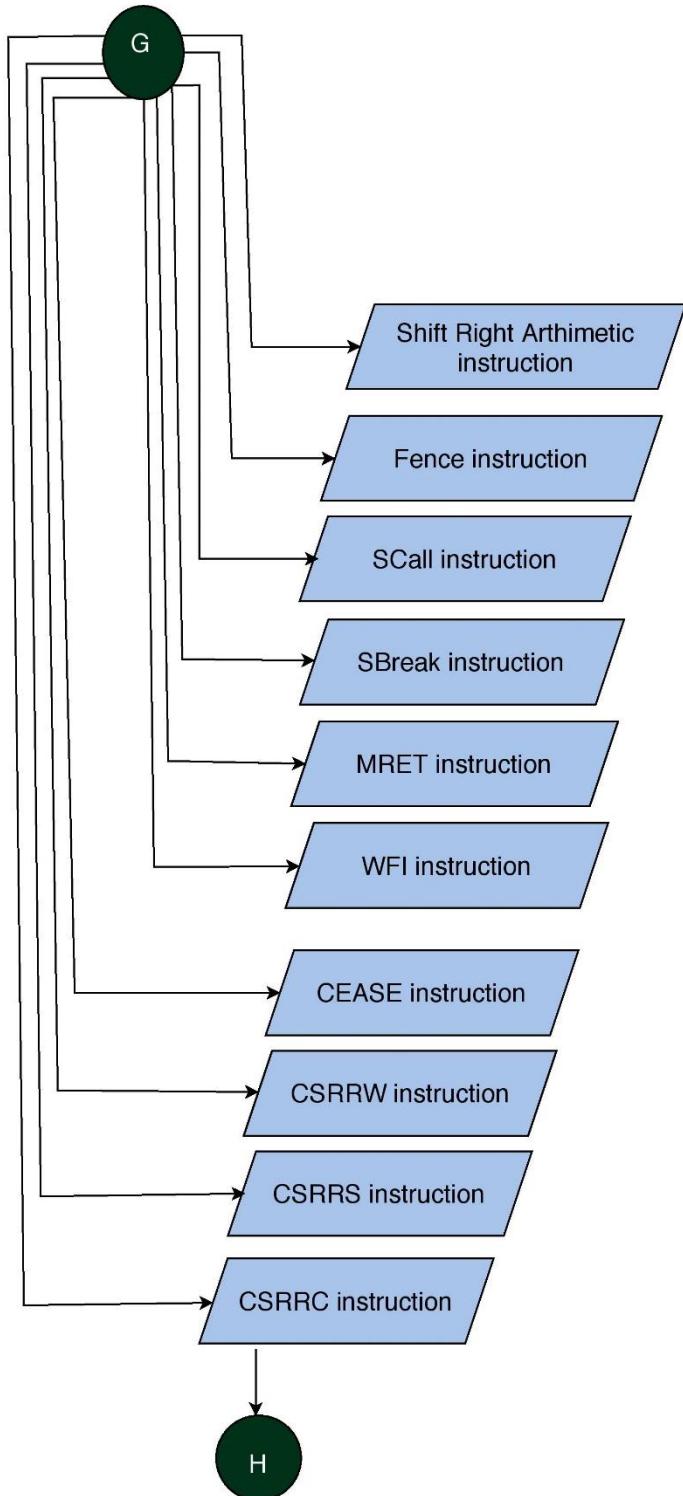
Rounding out RV32I are the eight system instructions. Simple implementations may choose to trap these instructions and emulate their functionality in system software, but higher performance implementations may implement more of their functionality in hardware

We have Y and N which we uses if we have true we have Y otherwise N. We uses sigs variable sequence in this List.

Example

In these RV32I system instruction format instruction we have 1st element position as Y which means we have legal instruction. We have 2nd, 3rd and 4th position as N which means we don't have fp(floating point) instruction, rocc instruction and branch instruction.

We have DW_X(data width of Alu) / DW_XPR(data width of Alu) and Aluop bits as input of List(FN_X / FN_ADD)



```

    CSRRWI->  List(Y,N,N,N,N,N,N,N,A2_IMM, A1_ZERO, IMM_Z,
DW_XPR,FN_ADD,   N,M_X,           N,N,N,N,N,N,Y,CSR.W,N,N,N,N),
    CSRRSI->  List(Y,N,N,N,N,N,N,N,A2_IMM, A1_ZERO, IMM_Z,
DW_XPR,FN_ADD,   N,M_X,           N,N,N,N,N,N,Y,CSR.S,N,N,N,N),
    CSRRCI->  List(Y,N,N,N,N,N,N,N,A2_IMM, A1_ZERO, IMM_Z,
DW_XPR,FN_ADD,   N,M_X,           N,N,N,N,N,N,Y,CSR.C,N,N,N,N)
}

}

```

```

class FenceIDelete(flushDCache: Boolean) (implicit val p: Parameters)
extends DecodeConstants

{
  private val (v, cmd) = if (flushDCache) (Y, BitPat(M_FLUSH_ALL)) else (N,
M_X)

  val table: Array[(BitPat, List[BitPat])] = Array(
    FENCE_I-> List(Y,N,N,N,N,N,N,N,A2_X, A1_X, IMM_X, DW_X, FN_X,
v,cmd,   N,N,N,N,N,N,CSR.N,Y,Y,N,N))
}

```

explanation

RV32I System Instructions

CSRRWI (Read and write CSR with immediate (I Format))

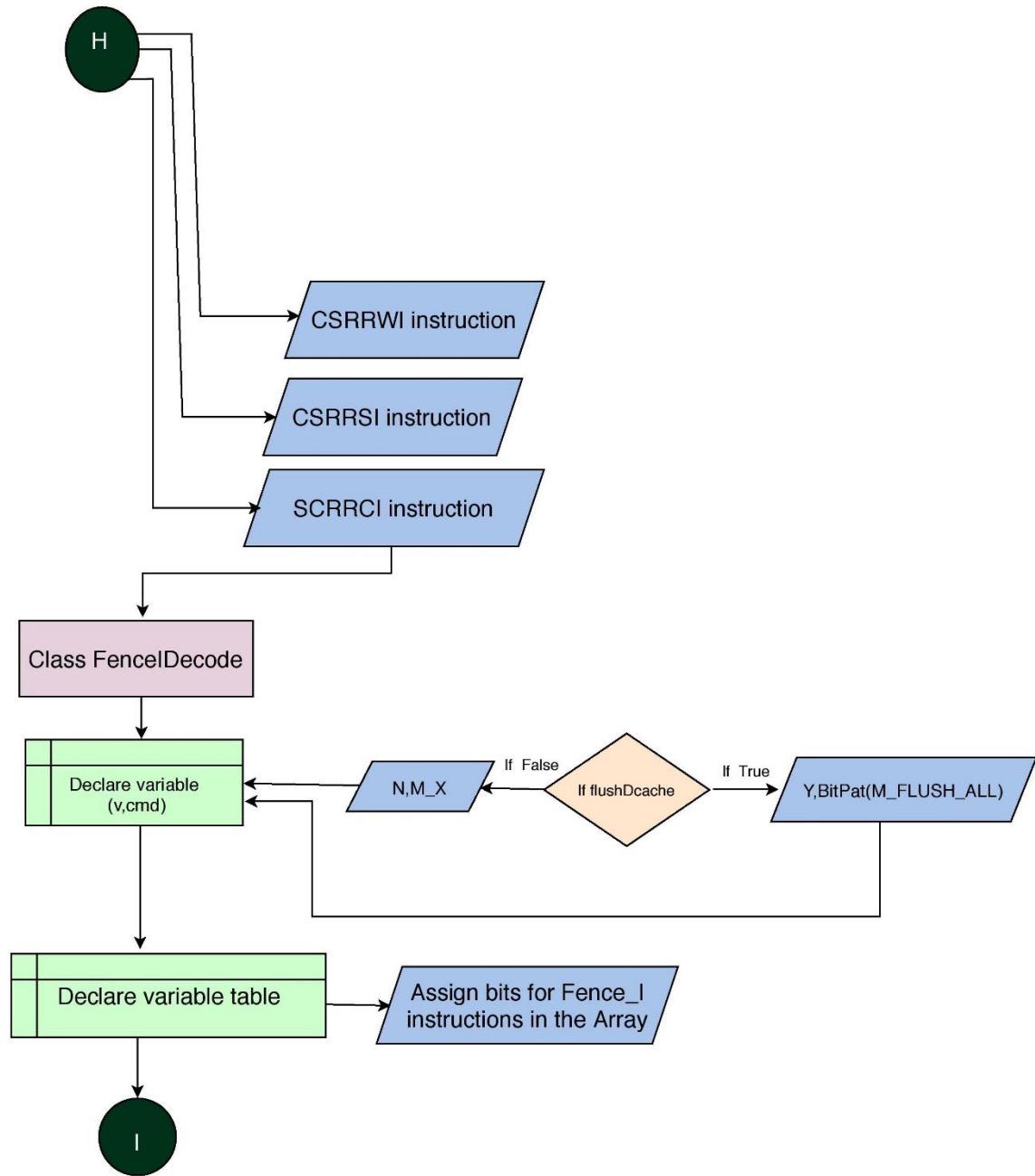
CSRRSI (Read and set bits in CSR with immediate (I Format))

CSRRCI (Read and clear bits in CSR with immediate (I Format))

Class FenceDecode is created, which takes parameter flushDCache as Boolean type which is extended by trait DecodeConstants. This class is used for pattern matching of RV32I memory access instruction. There are Y and N which are used if there is true then, there will be Y otherwise N. sigs variable is used for sequence in this List. There is private variable in which if condition is used. This variable is used to result later in our List.

Example

In these RV32I memory access instruction we have 1st element position as Y which means we have legal instruction. We have 2nd, position as N which means we have no floating point(fp) instruction. 3rd and 4th position as N which means we don't have rocc instruction and branch instruction. We have N on 7th and 8th position which means we have no source registers for other extensions. We have N at 16th signal indicating that we have no memcmd for Load instruction. We have Low (N) signal for source registers rfs1, rfs2, rfs3 and wfd (destination floating register) for RVF instruction which is at 17th ,18th, 19th and 20th position respectively. We have DW_X(data width of Alu) and Aluop bits as input of List(FN_X).We have private variable result v,cmd in our List



```

class CFlushDecode(supportsFlushLine: Boolean) (implicit val p: Parameters)
extends DecodeConstants

{

    private def zapRs1(x: BitPat) = if (supportsFlushLine) x else
    BitPat(x.value.U)

    val table: Array[(BitPat, List[BitPat])] = Array(

        zapRs1(CFLUSH_D_L1) ->
            List(Y, N, N, N, N, N, N, Y, N, A2_ZERO, A1_RS1, IMM_X,
        DW_XPR, FN_ADD, Y, M_FLUSH_ALL, N, N, N, N, N, N, CSR.I, N, N, N, N),

        zapRs1(CDISCARD_D_L1) ->
            List(Y, N, N, N, N, N, N, Y, N, A2_ZERO, A1_RS1, IMM_X,
        DW_XPR, FN_ADD, Y, M_FLUSH_ALL, N, N, N, N, N, N, CSR.I, N, N, N, N)

    )
}

class SVMDecode(implicit val p: Parameters) extends DecodeConstants

{
    val table: Array[(BitPat, List[BitPat])] = Array(

        SFENCE_VMA->List(Y, N, N, N, N, N, Y, Y, N, A2_ZERO, A1_RS1, IMM_X,
        DW_XPR, FN_ADD, Y, M_SFENCE, N, N, N, N, N, N, CSR.N, N, N, N, N)

    )
}

```

 explanation

Class CFlushDecode is created which takes parameter supportsFlushLine as Boolean type which is extended by trait DecodeConstants. There are Y and N which are used if there is true then, there will be Y otherwise N. sigs variable is used for sequence in this List. There is private variable in which if condition is used. Private method zapRs1 is used which takes BitPat as input parameter. There is if condition in zapRs1 method.

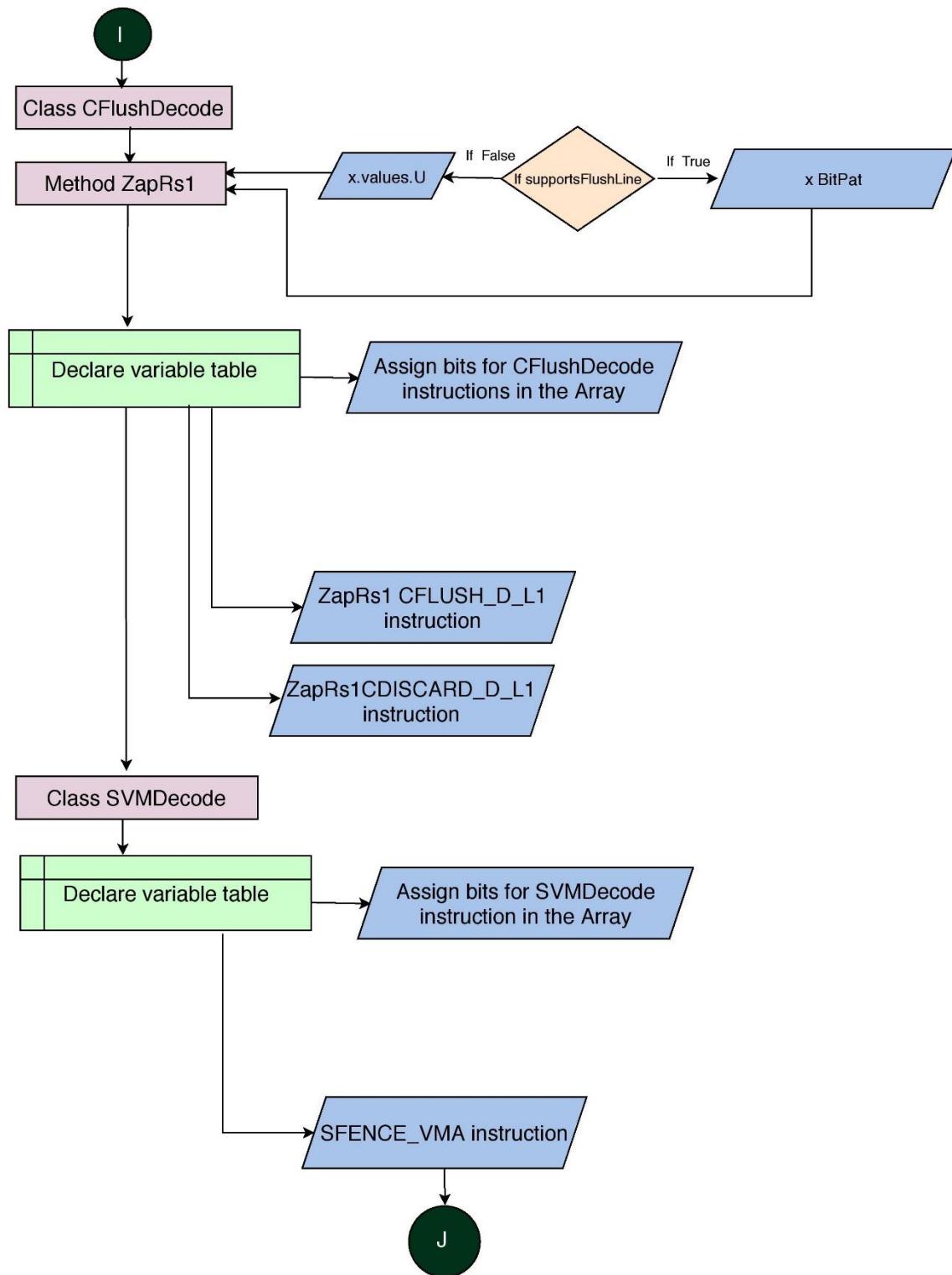
Example

We have 1st element position as Y which means we have legal instruction. We have 2nd, position as N which means we have no floating point(fp) instruction. 3rd and 4th position as N which means we don't have rocc instruction and branch instruction. We have N on 7th and at 8th position we have Y which means we have single source register.. We have Y at 16th signal indicating that we have memcmd. We have Low (N) signal for source registers rfs1, rfs2, rfs3 and wfd (destination floating register) for RVF instruction which is at 17th ,18th, 19th and 20th position respectively. We have DW_XPR(data width of Alu) and Aluop bits as input of List(FN_ADD).

Class SVMDecode, is created, which is extended by trait DecodeConstants. There are Y and N which are used if there is true then, there will be Y otherwise N. sigs variable is used for sequence in this List.

Example

We have SPENCE_VMA,SRET and DRET instructions in which we have 1st element position as Y which means we have legal instruction. We have 2nd, position as N which means we have no floating point(fp) instruction. 3rd and 4th position as N which means we don't have rocc instruction and branch instruction. We have Y on 7th and at 8th position of SPENCE_VMA and N. We have Y at 15th position which means we have single memval high for SPENCE_VMA and N for SRET and DRET. We have Low (N) signal for source registers rfs1, rfs2, rfs3 and wfd (destination floating register) for RVF instruction which is at 17th ,18th, 19th and 20th position respectively. We have DW_XPR / DW_X(data width of Alu) and Aluop bits as input of List(FN_X).



```

class SDecode(implicit val p: Parameters) extends DecodeConstants
{
  val table: Array[(BitPat, List[BitPat])] = Array(
    SRET-> List(Y,N,N,N,N,N,N,X,N,A2_X, A1_X, IMM_X, DW_X, FN_X,
N,M_X, N,N,N,N,N,N,CSR.I,N,N,N,N))
}

class DebugDecode(implicit val p: Parameters) extends DecodeConstants
{
  val table: Array[(BitPat, List[BitPat])] = Array(
    DRET-> List(Y,N,N,N,N,N,N,X,N,A2_X, A1_X, IMM_X, DW_X, FN_X,
N,M_X, N,N,N,N,N,N,CSR.I,N,N,N,N))
}

class I32Decode(implicit val p: Parameters) extends DecodeConstants
{
  val table: Array[(BitPat, List[BitPat])] = Array(
    SLLI_RV32-> List(Y,N,N,N,N,N,Y,N,A2_IMM, A1_RS1, IMM_I, DW_XPR,FN_SL,
N,M_X, N,N,N,N,N,Y,CSR.N,N,N,N,N),
    SRLI_RV32-> List(Y,N,N,N,N,N,Y,N,A2_IMM, A1_RS1, IMM_I, DW_XPR,FN_SR,
N,M_X, N,N,N,N,N,Y,CSR.N,N,N,N,N),
    SRAI_RV32-> List(Y,N,N,N,N,N,Y,N,A2_IMM, A1_RS1, IMM_I,
DW_XPR,FN_SRA, N,M_X, N,N,N,N,Y,CSR.N,N,N,N,N))
}

```

 explanation

Class SVMDecode,SDecode and DebugDecode are create, which are extended by trait DecodeConstants. There are Y and N which are used if there is true then, there will be Y otherwise N. sigs variable is used for sequence in this List.

Example

We have SPENCE_VMA,SRET and DRET instructions in which we have 1st element position as Y which means we have legal instruction. We have 2nd, position as N which means we have no floating point(fp) instruction. 3rd and 4th position as N which means we don't have rocc instruction and branch instruction. We have Y on 7th and at 8th position of SPENCE_VMA and N. We have Y at 15th position which means we have single memval high for SPENCE_VMA and N for SRET and DRET. We have Low (N) signal for source registers rfs1, rfs2, rfs3 and wfd (destination floating

register) for RVF instruction which is at 17th ,18th, 19th and 20th position respectively. We have DW_XPR / DW_X(data width of Alu) and Aluop bits as input of List(FN_X).

RV64I Instructions

SLLI_RV32 (Shift Left Logical Immediate (RV32))

SRLI_RV32 (Shift Right Logical Immediate (RV32))

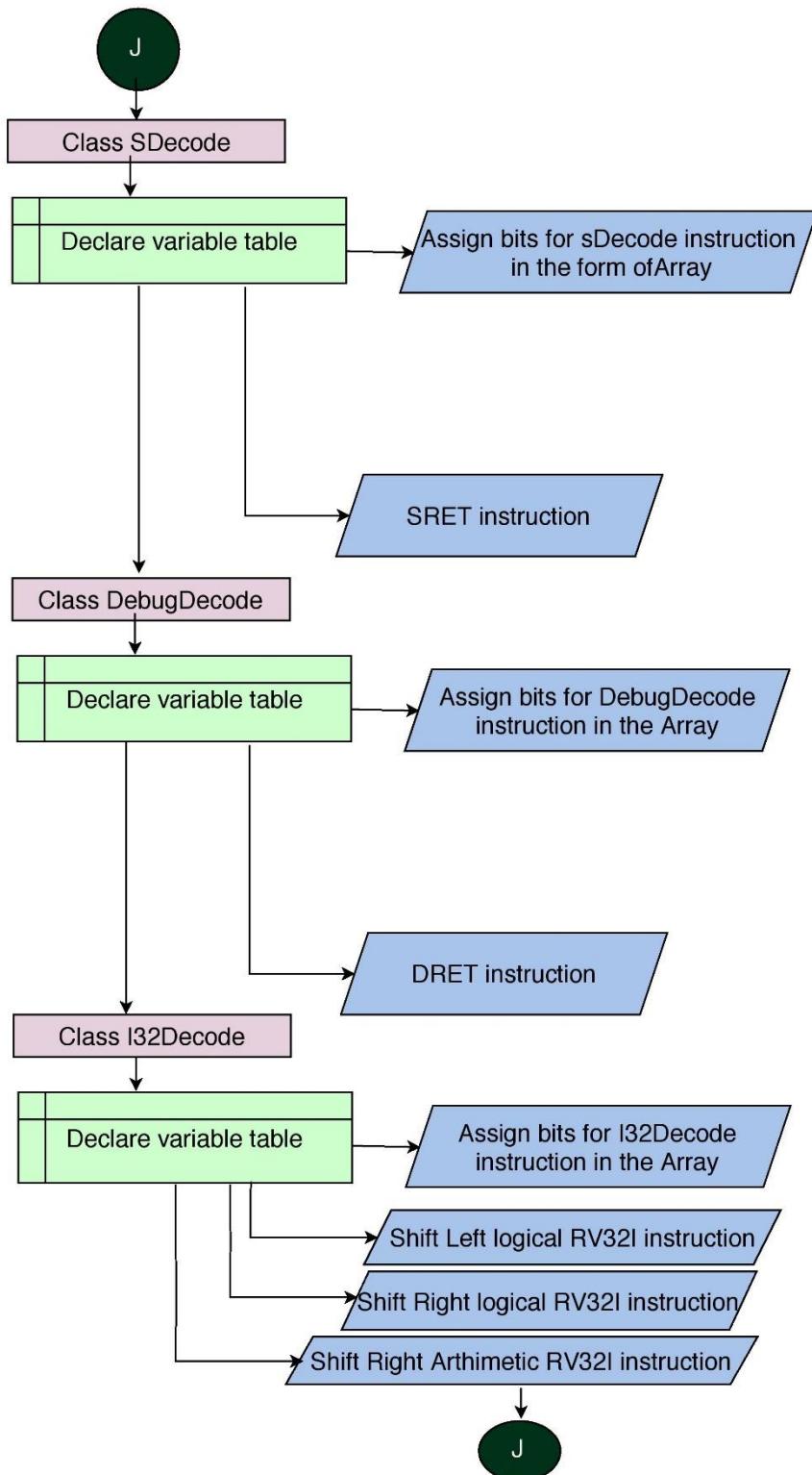
SRAI_RV32I (Shift Right Arithmetic Immediate (RV32))

Class I32Decode is created, which is extended by trait DecodeConstants. This class is used for pattern matching of RV32I instructions

There are Y and N which are used if there is true then, there will be Y otherwise N. sigs variable is used for sequence in this List.

Example

In these RV32I format instruction we have 1st element position as Y which means we have legal instruction. We have 2nd, 3rd and 4th position as N which means we don't have fp(floating point) instruction, rocc instruction and branch instruction. We have IMM_S(Immediate of Store),IMM_I(immediate of I format instruction), DW_XPR(data width of Alu) and Aluop bits as input of List(for example FN_SL,FN_SR etc).



RV64I Instructions

LD (Load Data)

LWU(Load Word Unsigned)

SD (Store Data)

SLLI(Shift Left Logical Immediate)

SRLI(Shift Right Logical Immediate)

SRAI(Shift Right Arithmetic Immediate)

ADDIW (Addition Immediate)

SLLIW (Shift Left Logical Immediate (RV64))

SRLIW (Shift Right Logical Immediate (RV64))

SRAIW (Shift Right Arithmetic Immediate (RV64))

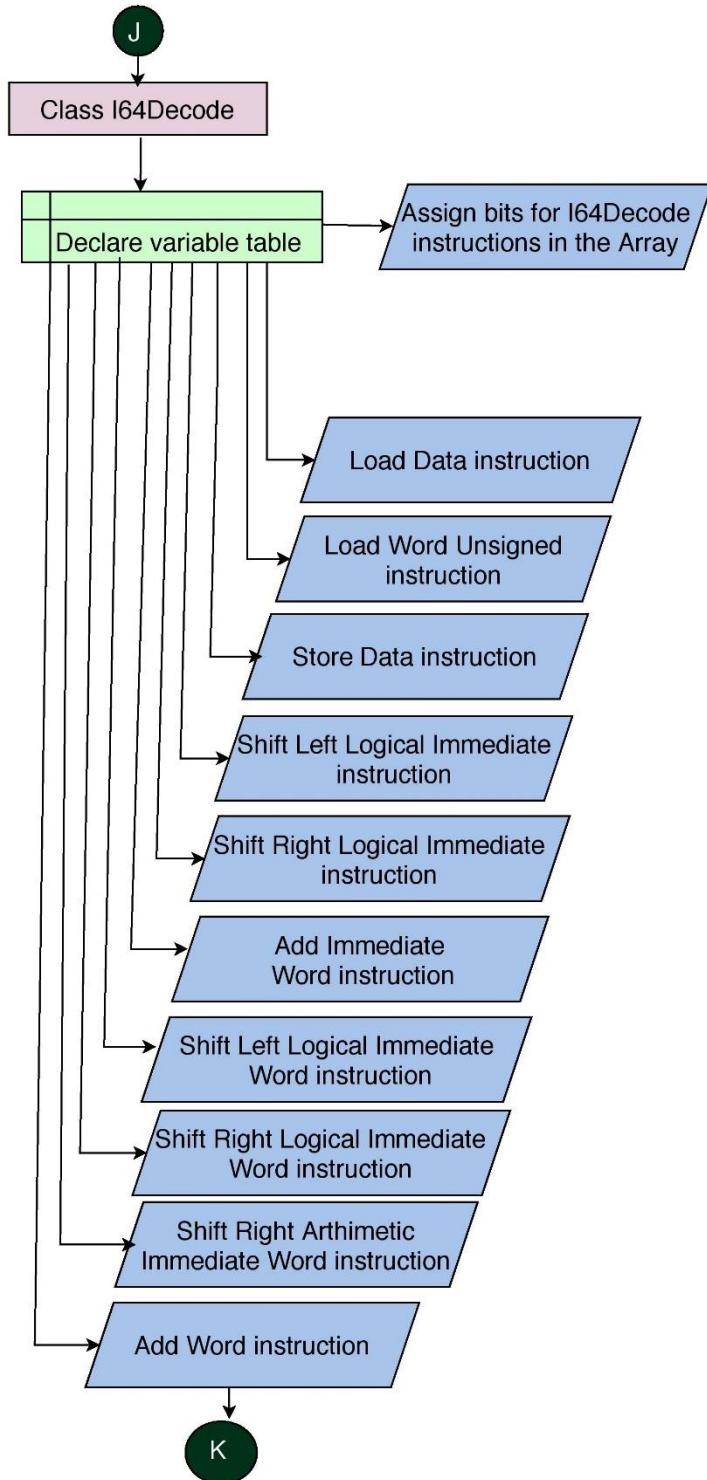
ADDW (Addition (RV64))

Class I64Decode is created, which is extended by trait DecodeConstants. This class is used for pattern matching of RV64I instructions

There are Y and N which are used if there is true then, there will be Y otherwise N. sigs variable is used for sequence in this List.

Example

In these RV64I format instruction we have 1st element position as Y which means we have legal instruction. We have 2nd, 3rd and 4th position as N which means we don't have fp(floating point) instruction, rocc instruction and branch instruction. We have IMM_S(immediate of Store),IMM_I(immediate of I format instruction), DW_32(data width of Alu) and Aluop bits as input of List(for example FN_ADD,FN_SL etc).



```

SUBW->      List(Y,N,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X, DW_32,FN_SUB,
N,M_X,          N,N,N,N,N,N,Y,CSR.N,N,N,N,N),
SLLW->      List(Y,N,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X, DW_32,FN_SL,
N,M_X,          N,N,N,N,N,N,Y,CSR.N,N,N,N,N),
SRLW->      List(Y,N,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X, DW_32,FN_SR,
N,M_X,          N,N,N,N,N,N,Y,CSR.N,N,N,N,N),
SRAW->      List(Y,N,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X, DW_32,FN_SRA,
N,M_X,          N,N,N,N,N,N,Y,CSR.N,N,N,N,N))
}


```

```

class MDecode(pipelinedMul: Boolean) (implicit val p: Parameters) extends
DecodeConstants

{
  val M = if (pipelinedMul) Y else N
  val D = if (pipelinedMul) N else Y

```

explanation

RV64I Instructions

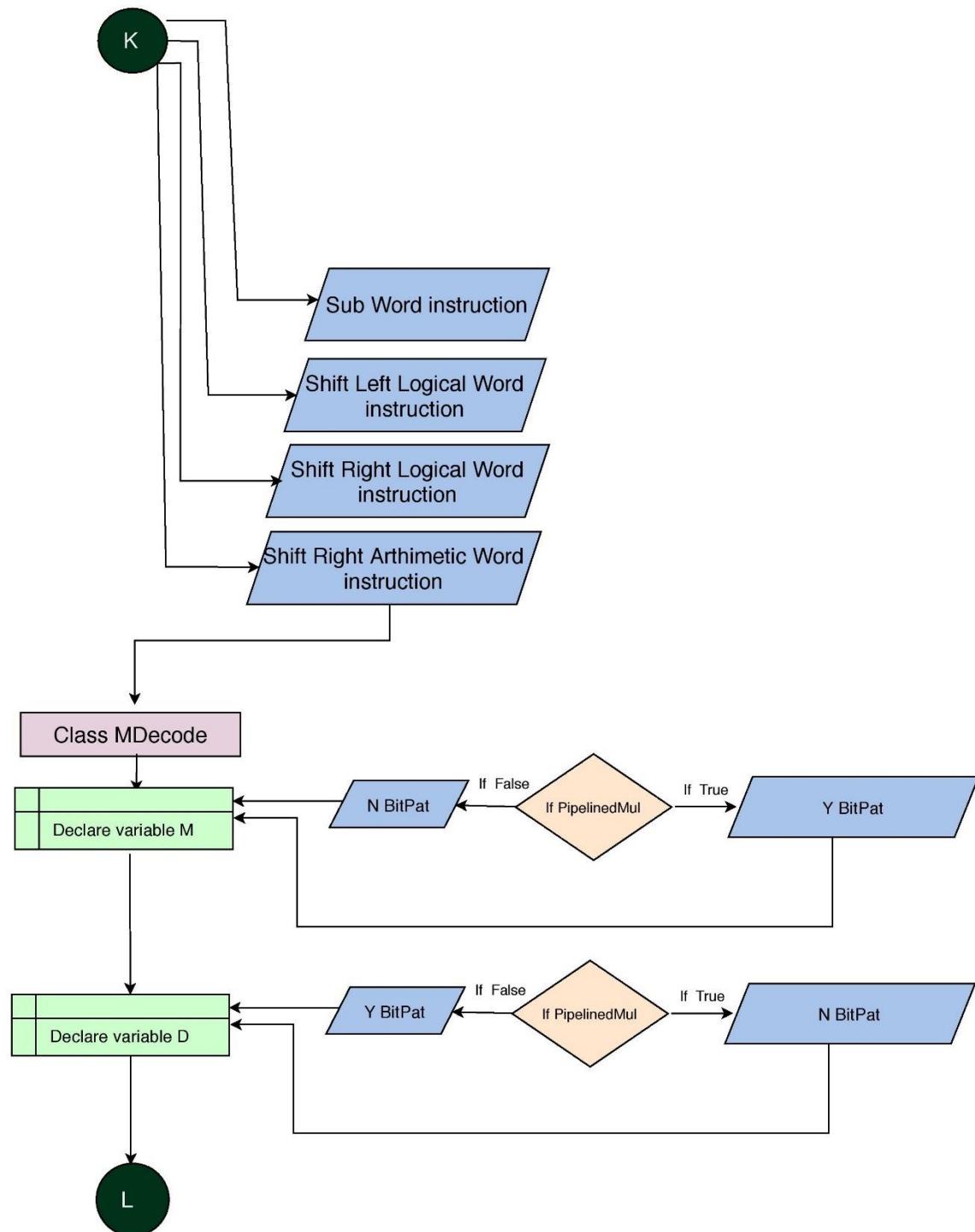
SUBW (Subtraction (RV64))

SLLW (Shift Left Logical (RV64))

SRLW (Shift Right Logical (RV64))

SRAW(Shift Right Arithmetic (RV64))

Class MDecode is created (takes pipelineMul as Bool type input) which is extended by trait DecodeConstants. This class is used for pattern matching of M extension instructions. We have two variables M and D in which we have if condition which decides whether we have Y and N



```

val table: Array[(BitPat, List[BitPat])] = Array(

  MUL->           List(Y,N,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X,
DW_XPR, FN_MUL,   N,M_X,          N,N,N,N,M,D,Y,CSR.N,N,N,N,N),
  MULH->          List(Y,N,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X,
DW_XPR, FN_MULH,  N,M_X,          N,N,N,N,M,D,Y,CSR.N,N,N,N,N),
  MULHU->         List(Y,N,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X,
DW_XPR, FN_MULHU, N,M_X,          N,N,N,N,M,D,Y,CSR.N,N,N,N,N),
  MULHSU->        List(Y,N,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X,
DW_XPR, FN_MULHSU, N,M_X,          N,N,N,N,M,D,Y,CSR.N,N,N,N,N),
  DIV->           List(Y,N,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X,
DW_XPR, FN_DIV,   N,M_X,          N,N,N,N,N,Y,Y,CSR.N,N,N,N,N),
  DIVU->          List(Y,N,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X,
DW_XPR, FN_DIVU,  N,M_X,          N,N,N,N,N,Y,Y,CSR.N,N,N,N,N),
  REM->           List(Y,N,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X,
DW_XPR, FN_Rem,   N,M_X,          N,N,N,N,N,Y,Y,CSR.N,N,N,N,N),
  REMU->          List(Y,N,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X,
DW_XPR, FN_Remu,  N,M_X,          N,N,N,N,N,Y,Y,CSR.N,N,N,N,N))
)

```

M extension Instructions

MUL (Multiplication)

MULH (Multiplication half word)

MULHU (Multiplication Half Word Unsigned)

MULHSU (Multiplication Half Word Signed Unsigned)

DIV (Division)

DIVU (Division Unsigned)

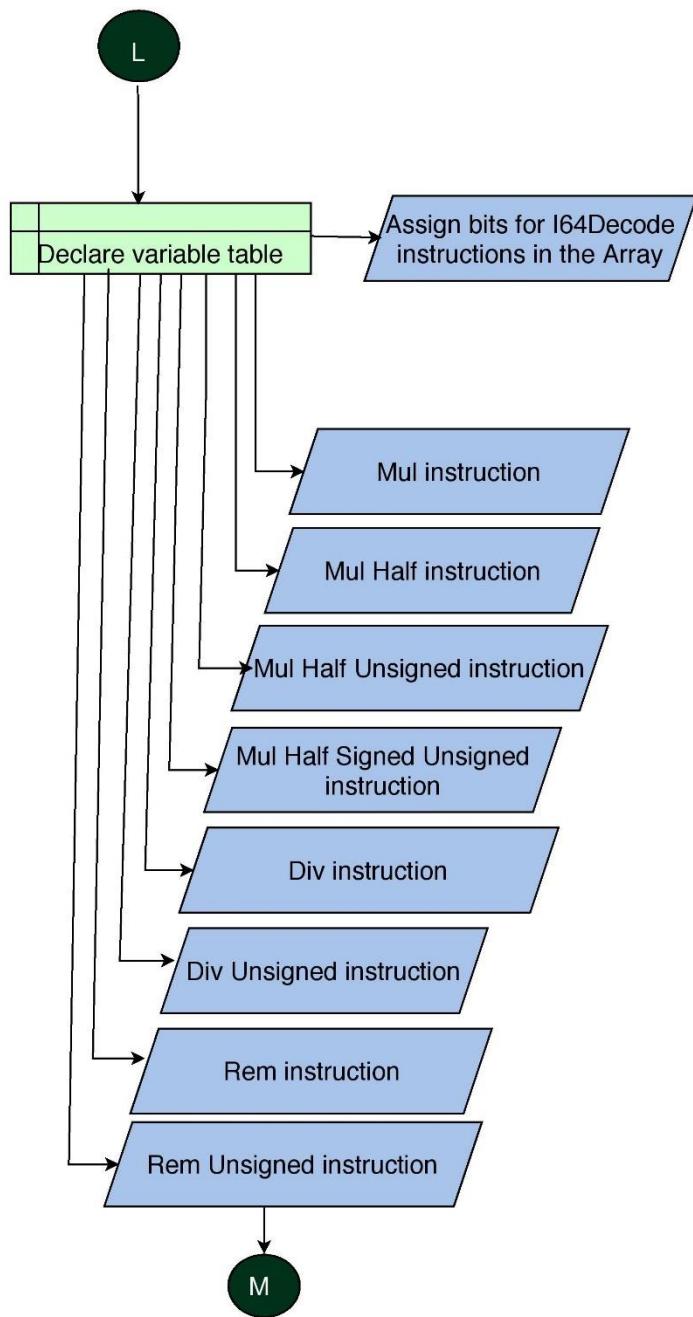
REM (Remainder)

REMU (Remainder Unsigned)

There are Y and N which are used if there is true then, there will be Y otherwise N. sigs variable is used for sequence in this List.

Example

In these M extension format instruction we have 1st element position as Y which means we have legal instruction. We have 2nd, 3rd and 4th position as N which means we don't have fp(floating point) instruction, rocc instruction and branch instruction. We have Y on 7th and 8th position which means we have two source registers. We have Y at 22nd position for division and remainder instructions and we have Y at 23rd signal indicating that we have write register. We have DW_XPR(data width of Alu) and Aluop bits as input of List(for example FN_DIV,FN_Rem etc).



```

class M64Decode(pipelinedMul: Boolean) (implicit val p: Parameters) extends
DecodeConstants

{
    val M = if (pipelinedMul) Y else N
    val D = if (pipelinedMul) N else Y
    val table: Array[(BitPat, List[BitPat])] = Array(

        MULW-> List(Y,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X, DW_32,
FN_MUL, N,M_X, N,N,N,N,M,D,Y,CSR.N,N,N,N,N),

        DIVW-> List(Y,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X, DW_32,
FN_DIV, N,M_X, N,N,N,N,Y,Y,CSR.N,N,N,N,N),

        DIVUW-> List(Y,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X, DW_32,
FN_DIVU, N,M_X, N,N,N,N,Y,Y,CSR.N,N,N,N,N),

        REMW-> List(Y,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X, DW_32,
FN_Rem, N,M_X, N,N,N,N,Y,Y,CSR.N,N,N,N,N),

        REMUW-> List(Y,N,N,N,N,Y,Y,N,A2_RS2, A1_RS1, IMM_X, DW_32,
FN_Remu, N,M_X, N,N,N,N,Y,Y,CSR.N,N,N,N,N))

}

```

— explanation —

M extension (RV64) Instructions

MULW (Multiplication (RV64))

MULH (Multiplication half word)

DIVW (Division (RV64))

DIVUW (Division Unsigned (RV64))

REMW (Remainder (RV64))

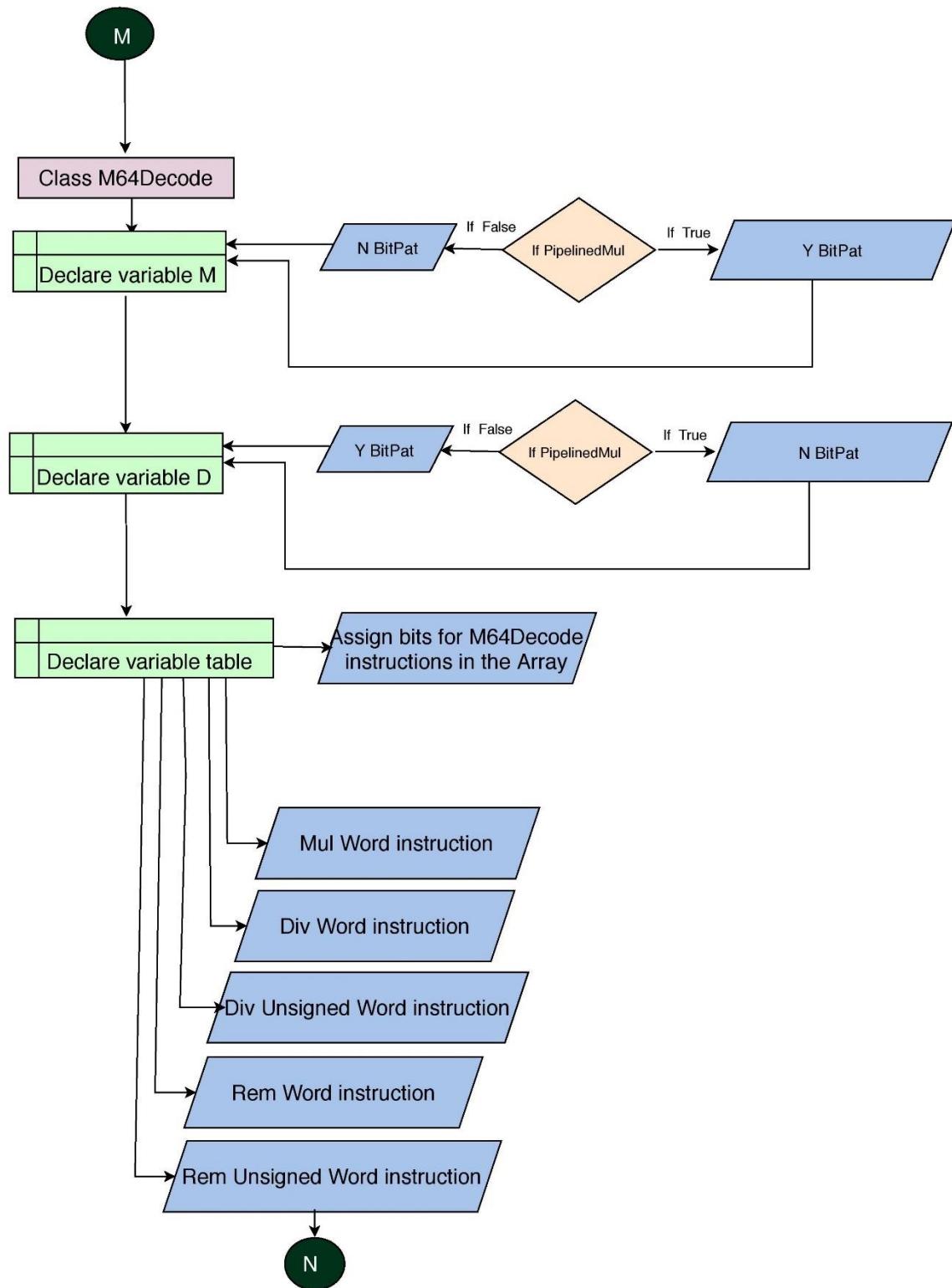
REMUW (Remainder Unsigned (RV64))

Class M64Decode is created, (takes pipelineMul as Bool type input) which is extended by trait DecodeConstants. This class is used for pattern matching of M extension instructions. We have two variables M and D in which we have if condition which decides whether we have Y and N

There are Y and N which are used if there is true then, there will be Y otherwise N. sigs variable is used for sequence in this List.

Example

In these M extension format instruction(RV64) we have 1st element position as Y which means we have legal instruction. We have 2nd, 3rd and 4th position as N which means we don't have fp(floating point) instruction, rocc instruction and branch instruction. We have Y on 7th and 8th position which means we have two source registers. We have Y at 22nd position for division and remainder instructions and we have Y at 23rd signal indicating that we have write register. We have DW_32(data width of Alu) and Aluop bits as input of List(for example FN_DIV,FN_REM etc).



```

class ADecode(implicit val p: Parameters) extends DecodeConstants
{
  val table: Array[(BitPat, List[BitPat])] = Array(

    AMOADD_W-> List(Y,N,N,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD, Y,M_XA_ADD, N,N,N,N,N,Y,CSR.N,N,N,Y,N),
    AMOXOR_W-> List(Y,N,N,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD, Y,M_XA_XOR, N,N,N,N,N,Y,CSR.N,N,N,Y,N),
    AMOSWAP_W-> List(Y,N,N,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD, Y,M_XA_SWAP, N,N,N,N,N,Y,CSR.N,N,N,Y,N),
    AMOAND_W-> List(Y,N,N,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD, Y,M_XA_AND, N,N,N,N,N,Y,CSR.N,N,N,Y,N),
    AMOOR_W-> List(Y,N,N,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD, Y,M_XA_OR, N,N,N,N,N,Y,CSR.N,N,N,Y,N),
    AMOMIN_W-> List(Y,N,N,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD, Y,M_XA_MIN, N,N,N,N,N,Y,CSR.N,N,N,Y,N),
    AMOMINU_W-> List(Y,N,N,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD, Y,M_XA_MINU, N,N,N,N,N,Y,CSR.N,N,N,Y,N),
    AMOMAX_W-> List(Y,N,N,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD, Y,M_XA_MAX, N,N,N,N,N,Y,CSR.N,N,N,Y,N),
    AMOMAXU_W-> List(Y,N,N,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD, Y,M_XA_MAXU, N,N,N,N,N,Y,CSR.N,N,N,Y,N),
}

```

explanation

RVA Instructions

AMOADD_W (Atomic addition)

AMOXOR_W(Atomic bitwise XOR)

AMOSWAP_W (Atomic swap)

AMOAND_W (Atomic bitwise AND)

AMOOR_W (Atomic bitwise OR)

AMOMIN_W (Atomic two's complement minimum)

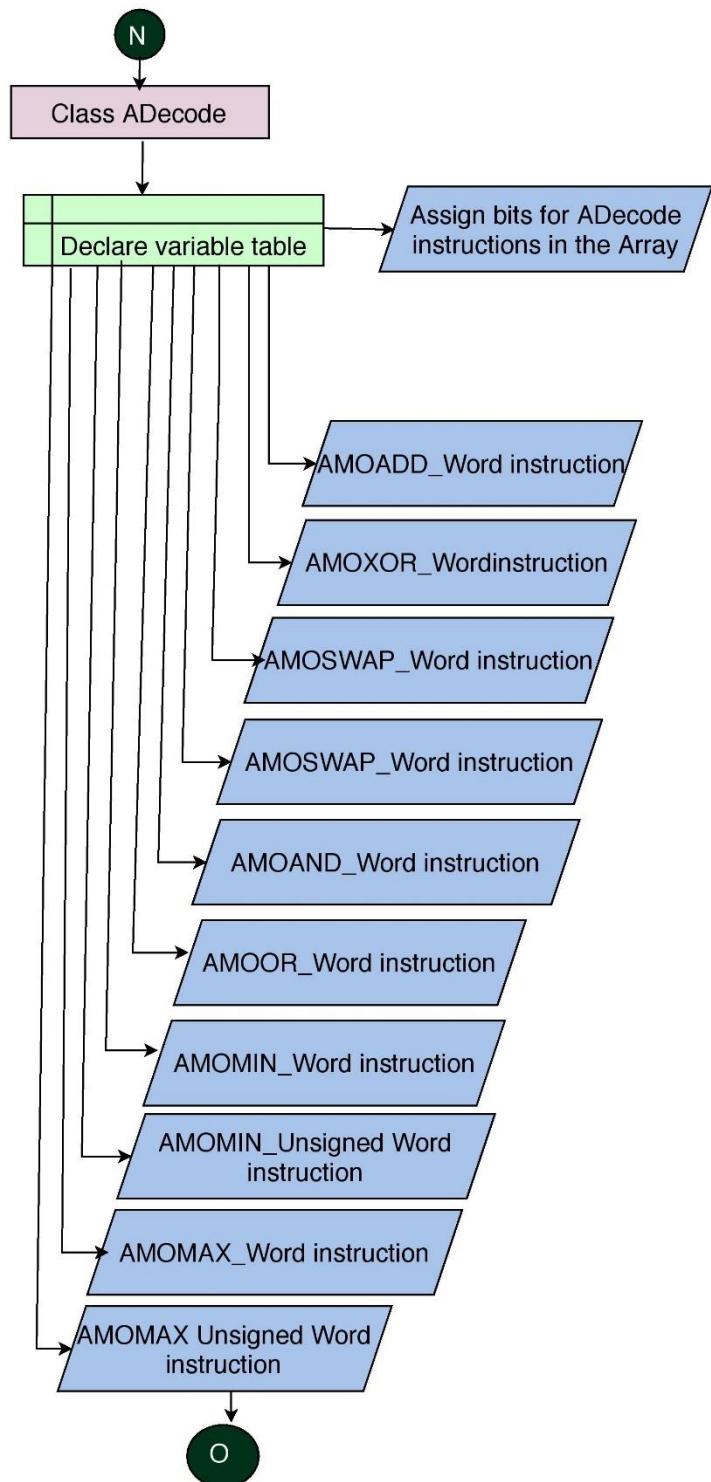
AMOMAX_W (Atomic two's complement maximum)

AMOMAXU_W (Atomic unsigned maximum)

The instructions with the w suffix operate on 32-bit words. Class ADecode is created, which is extended by trait DecodeConstants. This class is used for pattern matching of RVA Instructions. There are Y and N which are used if there is true then, there will be Y otherwise N. sigs variable is used for sequence in this List.

Example

In these RVA Instructions we have 1st element position as Y which means we have legal instruction. We have 2nd, 3rd and 4th position as N which means we don't have fp(floating point) instruction, rocc instruction and branch instruction. We have Y on 7th and 8th position which means we have two source registers. We have Y at 23rd signal indicating that we have write register. We have high signal amo (Y) for all RVA instruction which is at 27th position. We have DW_XPR(data width of Alu) and Aluop bits as input of List(for example FN_DIV,FN_REM etc).



```

    LR_W->      List(Y,N,N,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD,   Y,M_XLR,           N,N,N,N,N,N,Y,CSR.N,N,N,Y,N),
    SC_W->      List(Y,N,N,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD,   Y,M_XSC,           N,N,N,N,N,N,Y,CSR.N,N,N,Y,N))
}

class A64Decode(implicit val p: Parameters) extends DecodeConstants
{

  val table: Array[(BitPat, List[BitPat])] = Array(
    AMOADD_D-> List(Y,N,N,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD,   Y,M_XA_ADD,          N,N,N,N,N,N,Y,CSR.N,N,N,Y,N),
    AMOSWAP_D-> List(Y,N,N,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD,   Y,M_XA_SWAP,         N,N,N,N,N,N,Y,CSR.N,N,N,Y,N),
    AMOXOR_D-> List(Y,N,N,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD,   Y,M_XA_XOR,          N,N,N,N,N,N,Y,CSR.N,N,N,Y,N),
    AMOAND_D-> List(Y,N,N,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD,   Y,M_XA_AND,          N,N,N,N,N,N,Y,CSR.N,N,N,Y,N),
    AMOOR_D-> List(Y,N,N,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD,   Y,M_XA_OR,           N,N,N,N,N,N,Y,CSR.N,N,N,Y,N),
  )
}

```

explanation

RVA Instructions

LR_D (Load reserved)

SC_D (Store conditional)

RVA64 Instructions

AMOADD_D (Atomic addition)

AMOXOR_D(Atomic bitwise XOR)

AMOSWAP_D (Atomic swap)

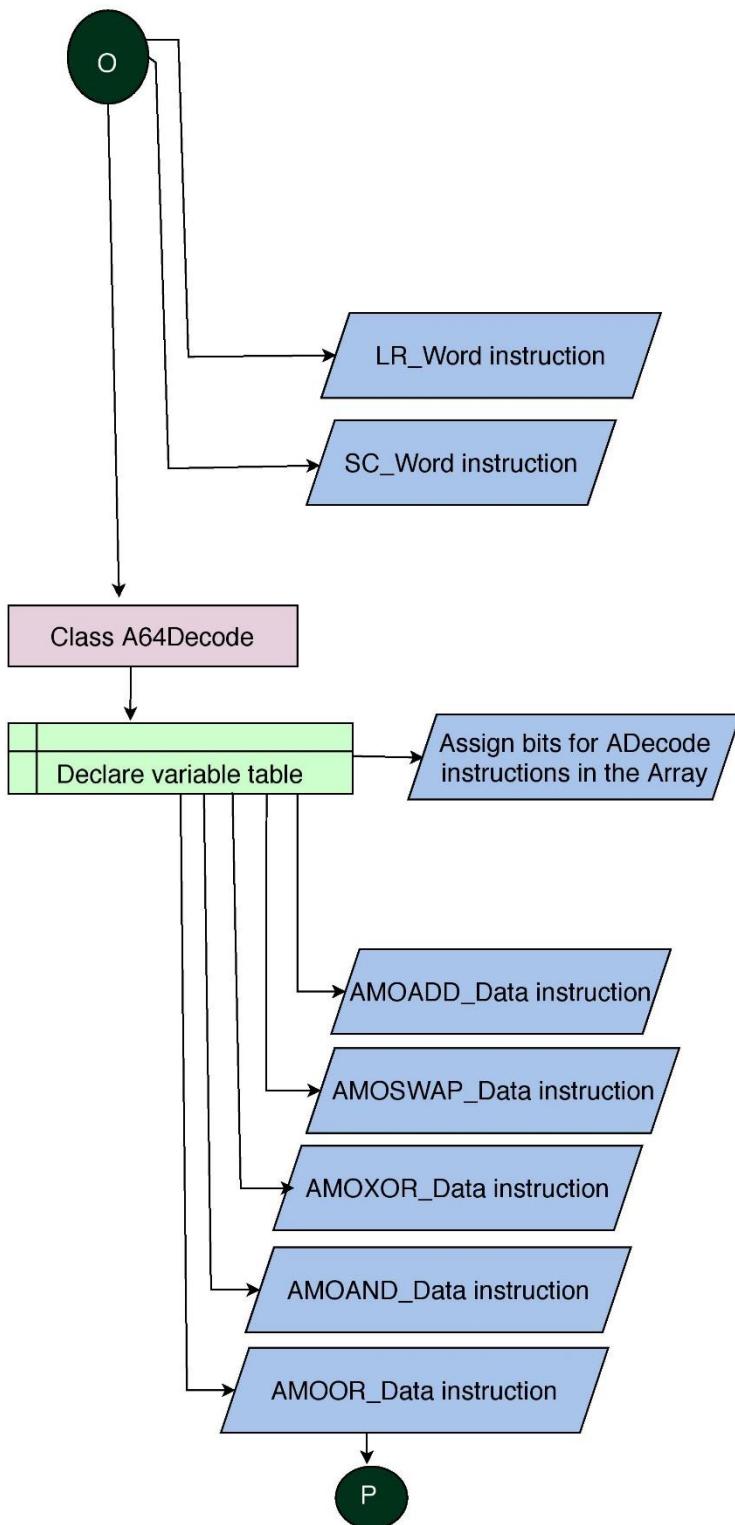
AMOAND_D (Atomic bitwise AND)

AMOOR_D (Atomic bitwise OR)

The instructions with the D suffix operate on 64-bit words. Class A64Decode is created, which is extended by trait DecodeConstants. This class is used for pattern matching of RVA Instructions. There are Y and N which are used if there is true then, there will be Y otherwise N. sigs variable is used for sequence in this List.

Example

In these RVA Instructions we have 1st element position as Y which means we have legal instruction. We have 2nd, 3rd and 4th position as N which means we don't have fp(floating point) instruction, rocc instruction and branch instruction. We have Y on 7th and 8th position which means we have two source registers. We have Y at 23rd signal indicating that we have write register. We have high signal amo (Y) for all RVA64 instruction which is at 27th position. We have DW_XPR(data width of Alu) and Aluop bits as input of List(for example FN_ADD).



```

AMOMIN_D-> List(Y,N,N,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD, Y,M_XA_MIN, N,N,N,N,N,N,Y,CSR.N,N,N,Y,N),
AMOMINU_D-> List(Y,N,N,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD, Y,M_XA_MINU, N,N,N,N,N,N,Y,CSR.N,N,N,Y,N),
AMOMAX_D-> List(Y,N,N,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD, Y,M_XA_MAX, N,N,N,N,N,N,Y,CSR.N,N,N,Y,N),
AMOMAXU_D-> List(Y,N,N,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD, Y,M_XA_MAXU, N,N,N,N,N,N,Y,CSR.N,N,N,Y,N),

LR_D-> List(Y,N,N,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD, Y,M_XLR, N,N,N,N,N,N,Y,CSR.N,N,N,Y,N),
SC_D-> List(Y,N,N,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD, Y,M_XSC, N,N,N,N,N,N,Y,CSR.N,N,N,Y,N))
}

```

```

class FDecode(implicit val p: Parameters) extends DecodeConstants
{
  val table: Array[(BitPat, List[BitPat])] = Array(
    FSGNJ_S-> List(Y,Y,N,N,N,N,N,N,N,A2_X, A1_X, IMM_X, DW_X, FN_X,
N,M_X, Y,Y,N,Y,N,N,N,CSR.N,N,N,N,N),
    FSGNJX_S-> List(Y,Y,N,N,N,N,N,N,N,A2_X, A1_X, IMM_X, DW_X, FN_X,
N,M_X, Y,Y,N,Y,N,N,N,CSR.N,N,N,N,N),
    FSGNJS_S-> List(Y,Y,N,N,N,N,N,N,N,A2_X, A1_X, IMM_X, DW_X, FN_X,
N,M_X, Y,Y,N,Y,N,N,N,CSR.N,N,N,N,N),

```

explanation

RVA64 Instructions

AMOMIN_D (Atomic two's complement minimum)

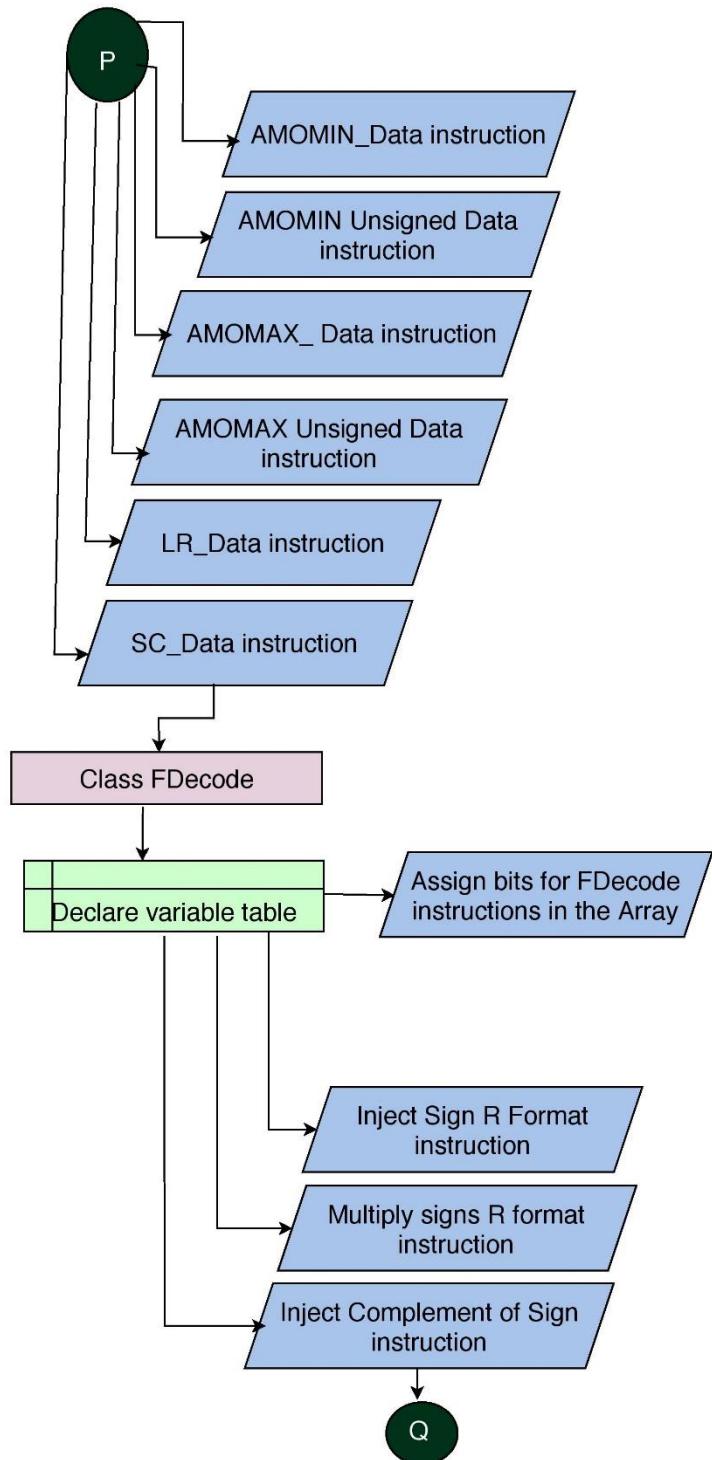
AMOMAX_D (Atomic two's complement maximum)

AMOMAXU_D (Atomic unsigned maximum)

LR_D (Load reserved)

SC_D (Store conditional)

Explanation of FDecode is written afterwards.



FMIN_S->	List(Y,Y,N,N,N,N,N,N,A2_X, N,M_X, Y,Y,N,Y,N,N,N,CSR.N,N,N,N,N),	A1_X,	IMM_X, DW_X, FN_X,
FMAX_S->	List(Y,Y,N,N,N,N,N,N,A2_X, N,M_X, Y,Y,N,Y,N,N,N,CSR.N,N,N,N,N),	A1_X,	IMM_X, DW_X, FN_X,
FADD_S->	List(Y,Y,N,N,N,N,N,N,A2_X, N,M_X, Y,Y,N,Y,N,N,N,CSR.N,N,N,N,N),	A1_X,	IMM_X, DW_X, FN_X,
FSUB_S->	List(Y,Y,N,N,N,N,N,N,A2_X, N,M_X, Y,Y,N,Y,N,N,N,CSR.N,N,N,N,N),	A1_X,	IMM_X, DW_X, FN_X,
FMUL_S->	List(Y,Y,N,N,N,N,N,N,A2_X, N,M_X, Y,Y,N,Y,N,N,N,CSR.N,N,N,N,N),	A1_X,	IMM_X, DW_X, FN_X,
FMADD_S->	List(Y,Y,N,N,N,N,N,N,A2_X, N,M_X, Y,Y,Y,N,N,N,CSR.N,N,N,N,N),	A1_X,	IMM_X, DW_X, FN_X,
FMSUB_S->	List(Y,Y,N,N,N,N,N,N,A2_X, N,M_X, Y,Y,Y,N,N,N,CSR.N,N,N,N,N),	A1_X,	IMM_X, DW_X, FN_X,
FNMADD_S->	List(Y,Y,N,N,N,N,N,N,A2_X, N,M_X, Y,Y,Y,N,N,N,CSR.N,N,N,N,N),	A1_X,	IMM_X, DW_X, FN_X,
FNMSUB_S->	List(Y,Y,N,N,N,N,N,N,A2_X, N,M_X, Y,Y,Y,N,N,N,CSR.N,N,N,N,N),	A1_X,	IMM_X, DW_X, FN_X,
FCLASS_S->	List(Y,Y,N,N,N,N,N,N,A2_X, N,M_X, Y,N,N,N,N,Y,CSR.N,N,N,N,N),	A1_X,	IMM_X, DW_X, FN_X,

explanation

RVF Instructions

FSGNJ_S (Inject sign (R Format))

FSGNJP_S(Multiply signs(R Format))

FSGNIN_S (Inject complement of sign (R Format))

FMIN_S (Compute minimum of two values (R Format))

FMAX_S(Compute maximum of two values (R Format))

FADD_S (Add two registers (R Format))

FSUB_S (Subtract two registers (R Format))

FMUL_S (Multiply two registers (R Format))

FMADD_S (**(fs1×fs2+fs3)** where s1,s2,s3 are source register (R Format))

FMSUB_S ((fs1×fs2-fs3) where s1,s2,s3 are source register (R Format))

FNMADD_S (- (fs1×fs2+fs3) where s1,s2,s3 are source register (R Format))

FNMSUB_S (- (fs1×fs2-fs3) where s1,s2,s3 are source register (R Format))

FCLASS_S (Classify floating-point value (R Format))

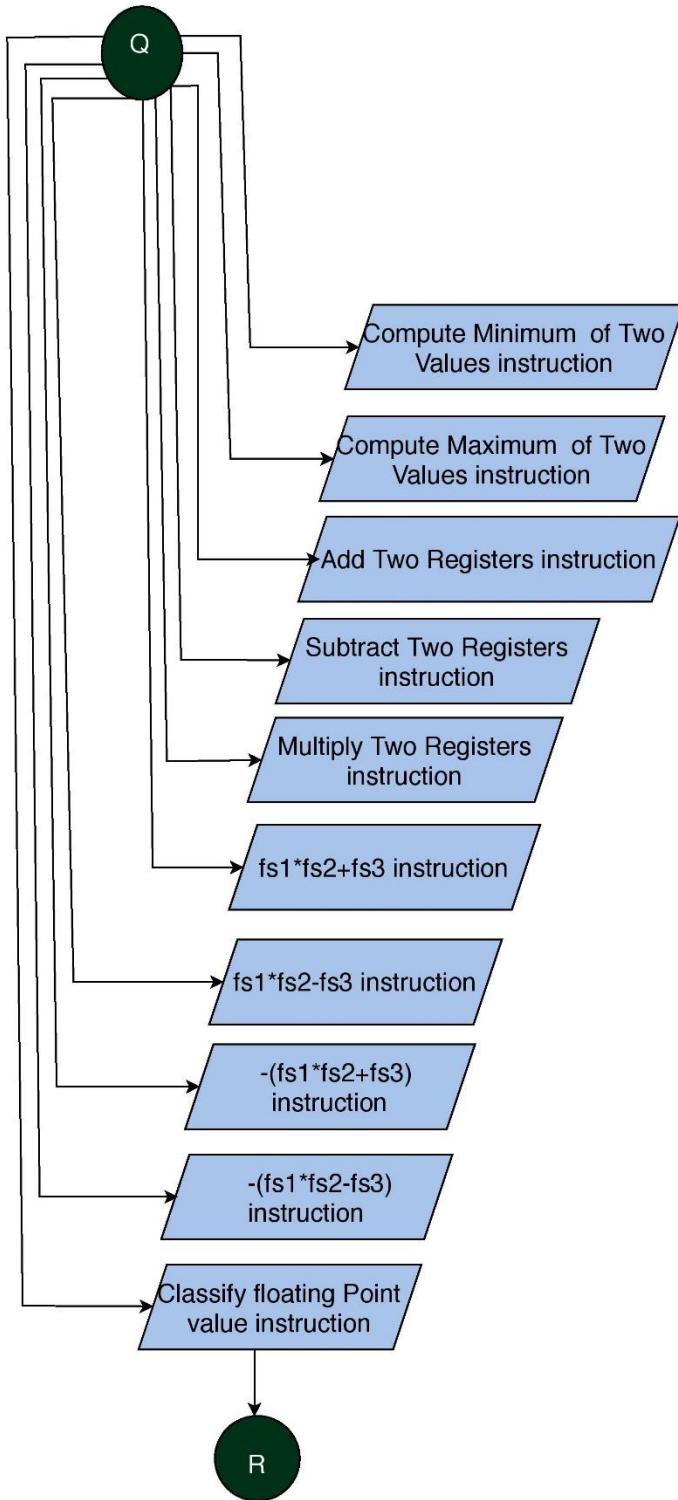
Class FDecode is created, which is extended by trait DecodeConstants. This class is used for pattern matching of RVF Instructions. There are Y and N which are used if there is true then, there will be Y otherwise N. sigs variable is used for sequence in this List.

Example

In these RVF Instructions we have 1st element position as Y which means we have legal instruction. We have 2nd, position as Y which means we have floating point(fp) instruction. 3rd and 4th position as N which means we don't have rocc instruction and branch instruction. We have N on 7th and 8th position which means we have no source registers for other extensions. We have Y at 23rd signal indicating that we have write register. We have high signal (Y) for source registers rfs1, rfs2, rfs3 and wfd (destination floating register) for RVF instruction which is at 17th, 18th, 19th and 20th position respectively. We have DW_X(data width of Alu) and Aluop bits as input of List(FN_X).

Instruction Example

FSGNJ copies the sign from a second number; FSGNIN copies the negated sign from a second number; and FSGNINX takes the new sign from the product of the two input sign



FMV_X_S-> $\text{List}(\text{Y}, \text{Y}, \text{N}, \text{N}, \text{N}, \text{N}, \text{N}, \text{N}, \text{A2_X},$ $\text{N}, \text{M_X},$ $\text{Y}, \text{N}, \text{N}, \text{N}, \text{N}, \text{N}, \text{Y}, \text{CSR.N}, \text{N}, \text{N}, \text{N}, \text{N}),$	A1_X, IMM_X, DW_X, FN_X,
FCVT_W_S-> $\text{List}(\text{Y}, \text{Y}, \text{N}, \text{N}, \text{N}, \text{N}, \text{N}, \text{N}, \text{A2_X},$ $\text{N}, \text{M_X},$ $\text{Y}, \text{N}, \text{N}, \text{N}, \text{N}, \text{N}, \text{Y}, \text{CSR.N}, \text{N}, \text{N}, \text{N}, \text{N}),$	A1_X, IMM_X, DW_X, FN_X,
FCVT_WU_S-> $\text{List}(\text{Y}, \text{Y}, \text{N}, \text{N}, \text{N}, \text{N}, \text{N}, \text{N}, \text{A2_X},$ $\text{N}, \text{M_X},$ $\text{Y}, \text{N}, \text{N}, \text{N}, \text{N}, \text{Y}, \text{CSR.N}, \text{N}, \text{N}, \text{N}, \text{N}),$	A1_X, IMM_X, DW_X, FN_X,
FEQ_S-> $\text{List}(\text{Y}, \text{Y}, \text{N}, \text{N}, \text{N}, \text{N}, \text{N}, \text{N}, \text{A2_X},$ $\text{N}, \text{M_X},$ $\text{Y}, \text{Y}, \text{N}, \text{N}, \text{N}, \text{Y}, \text{CSR.N}, \text{N}, \text{N}, \text{N}, \text{N}),$	A1_X, IMM_X, DW_X, FN_X,
FLT_S-> $\text{List}(\text{Y}, \text{Y}, \text{N}, \text{N}, \text{N}, \text{N}, \text{N}, \text{N}, \text{A2_X},$ $\text{N}, \text{M_X},$ $\text{Y}, \text{Y}, \text{N}, \text{N}, \text{N}, \text{Y}, \text{CSR.N}, \text{N}, \text{N}, \text{N}, \text{N}),$	A1_X, IMM_X, DW_X, FN_X,
FLE_S-> $\text{List}(\text{Y}, \text{Y}, \text{N}, \text{N}, \text{N}, \text{N}, \text{N}, \text{N}, \text{A2_X},$ $\text{N}, \text{M_X},$ $\text{Y}, \text{Y}, \text{N}, \text{N}, \text{N}, \text{Y}, \text{CSR.N}, \text{N}, \text{N}, \text{N}, \text{N}),$	A1_X, IMM_X, DW_X, FN_X,
FMV_S_X-> $\text{List}(\text{Y}, \text{Y}, \text{N}, \text{N}, \text{N}, \text{N}, \text{N}, \text{Y}, \text{N}, \text{A2_X},$ $\text{N}, \text{M_X},$ $\text{N}, \text{N}, \text{N}, \text{Y}, \text{N}, \text{N}, \text{N}, \text{CSR.N}, \text{N}, \text{N}, \text{N}, \text{N}),$	A1_RS1, IMM_X, DW_X, FN_X,
FCVT_S_W-> $\text{List}(\text{Y}, \text{Y}, \text{N}, \text{N}, \text{N}, \text{N}, \text{Y}, \text{N}, \text{A2_X},$ $\text{N}, \text{M_X},$ $\text{N}, \text{N}, \text{N}, \text{Y}, \text{N}, \text{N}, \text{N}, \text{CSR.N}, \text{N}, \text{N}, \text{N}, \text{N}),$	A1_RS1, IMM_X, DW_X, FN_X,
FCVT_S_WU-> $\text{List}(\text{Y}, \text{Y}, \text{N}, \text{N}, \text{N}, \text{N}, \text{Y}, \text{N}, \text{A2_X},$ $\text{N}, \text{M_X},$ $\text{N}, \text{N}, \text{N}, \text{Y}, \text{N}, \text{N}, \text{N}, \text{CSR.N}, \text{N}, \text{N}, \text{N}, \text{N}),$	A1_RS1, IMM_X, DW_X, FN_X,

explanation

RVF Instructions

FMV_X_S (Move from floating-point to integer register(R Format))

FCVT_W_S (Convert to signed 32-bit integer (R Format))

FCVT_WU_S (Convert to [un]signed 32-bit integer (R Format))

FEQ_S (Set if equal (R Format))

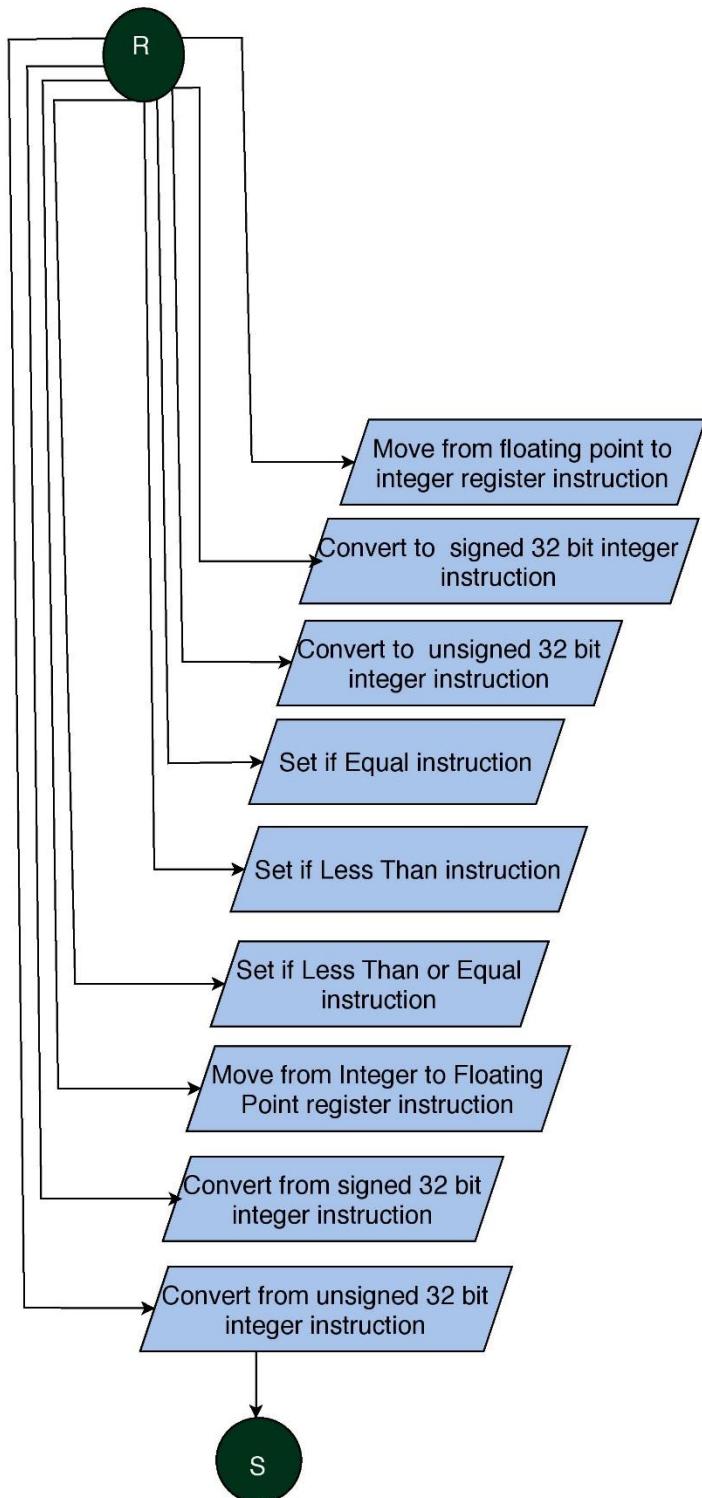
FLT_S (Set if less than (R Format))

FLE_S (Set if less than or equal (R Format))

FMV_S_X (Move from integer to floating-point register (R Format))

FCVT_S_W (Convert from signed 32-bit integer (R Format))

FCVT_S_WU (Convert from unsigned 32-bit integer (R Format))



```

FLW->      List(Y,Y,N,N,N,N,N,Y,N,A2_IMM, A1_RS1, IMM_I,
DW_XPR,FN_ADD,   Y,M_XRD,           N,N,N,Y,N,N,N,CSR.N,N,N,N,N),
FSW->      List(Y,Y,N,N,N,N,N,Y,N,A2_IMM, A1_RS1, IMM_S,
DW_XPR,FN_ADD,   Y,M_XWR,           N,Y,N,N,N,N,N,CSR.N,N,N,N,N),
FDIV_S->   List(Y,Y,N,N,N,N,N,N,A2_X,   A1_X,   IMM_X, DW_X, FN_X,
N,M_X,       Y,Y,N,Y,N,N,N,CSR.N,N,N,N,N),
FSQRT_S->  List(Y,Y,N,N,N,N,N,N,A2_X,   A1_X,   IMM_X, DW_X, FN_X,
N,M_X,       Y,Y,N,Y,N,N,N,CSR.N,N,N,N,N)
}


```

```

class DDecode(implicit val p: Parameters) extends DecodeConstants
{
  val table: Array[(BitPat, List[BitPat])] = Array(
    FCVT_S_D-> List(Y,Y,N,N,N,N,N,N,N,A2_X,   A1_X,   IMM_X, DW_X, FN_X,
N,M_X,       Y,N,N,Y,N,N,N,CSR.N,N,N,N,Y),
    FCVT_D_S-> List(Y,Y,N,N,N,N,N,N,A2_X,   A1_X,   IMM_X, DW_X, FN_X,
N,M_X,       Y,N,N,Y,N,N,N,CSR.N,N,N,N,Y),
    FSGNJ_D-> List(Y,Y,N,N,N,N,N,N,A2_X,   A1_X,   IMM_X, DW_X, FN_X,
N,M_X,       Y,Y,N,Y,N,N,N,CSR.N,N,N,N,Y),
  )
}


```

explanation

RVF Instructions

FSW (Store 32-bit floating-point word (I Format))

FDIV_S (Divide two registers (R Format))

FSQRT_S (Compute square root (R Format))

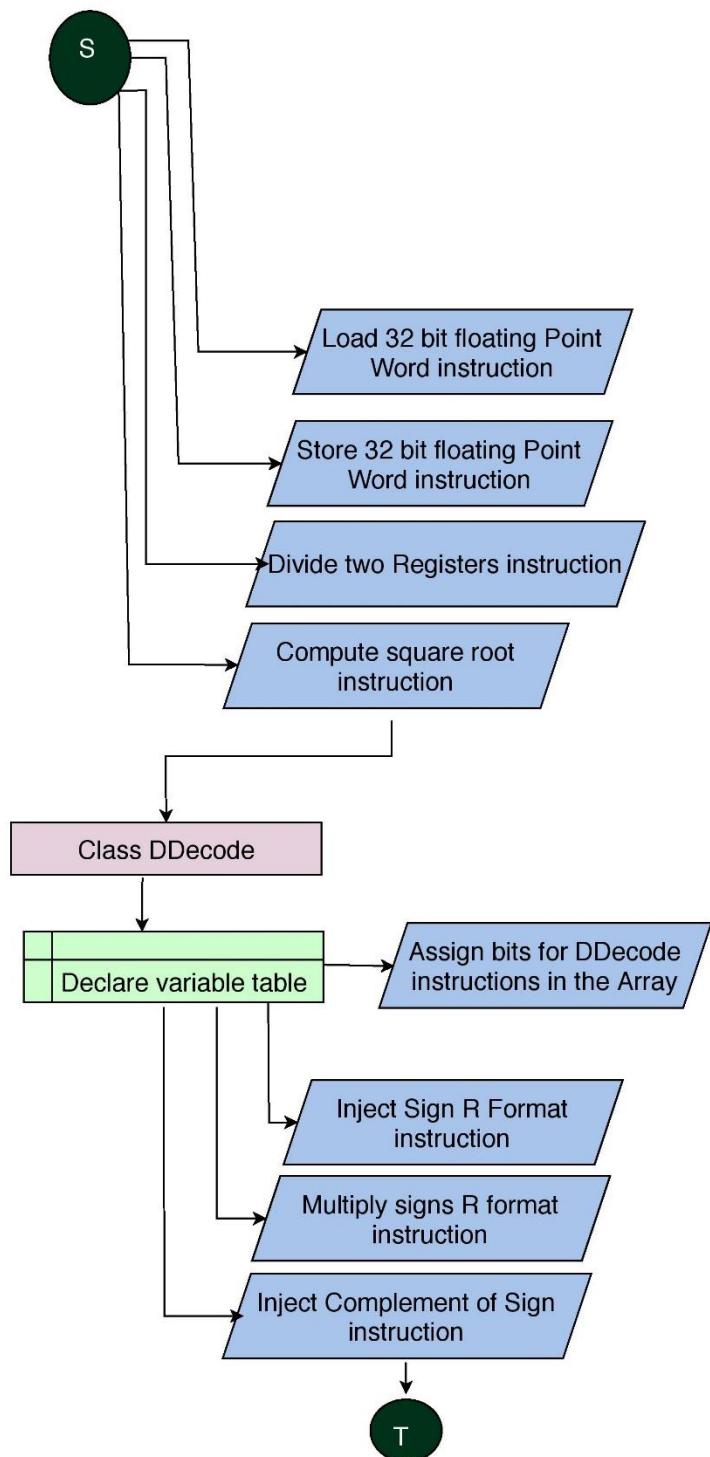
RVD Instructions

FCVT_S_D (Convert from double to single)

FCVT_D_S (Convert from single to double)

FSGNJ_D (Inject sign (R Format))

UDecode explanation is written afterwards.



FSGNJP_D->	List(Y, Y, N, N, N, N, N, N, A2_X, N, M_X, Y, Y, N, Y, N, N, N, CSR.N, N, N, N, Y),	A1_X,	IMM_X, DW_X, FN_X,
FSGNJP_D->	List(Y, Y, N, N, N, N, N, N, A2_X, N, M_X, Y, Y, N, Y, N, N, N, CSR.N, N, N, N, Y),	A1_X,	IMM_X, DW_X, FN_X,
FMIN_D->	List(Y, Y, N, N, N, N, N, N, A2_X, N, M_X, Y, Y, N, Y, N, N, N, CSR.N, N, N, N, Y),	A1_X,	IMM_X, DW_X, FN_X,
FMAX_D->	List(Y, Y, N, N, N, N, N, N, A2_X, N, M_X, Y, Y, N, Y, N, N, N, CSR.N, N, N, N, Y),	A1_X,	IMM_X, DW_X, FN_X,
FADD_D->	List(Y, Y, N, N, N, N, N, N, A2_X, N, M_X, Y, Y, N, Y, N, N, N, CSR.N, N, N, N, Y),	A1_X,	IMM_X, DW_X, FN_X,
FSUB_D->	List(Y, Y, N, N, N, N, N, N, A2_X, N, M_X, Y, Y, N, Y, N, N, N, CSR.N, N, N, N, Y),	A1_X,	IMM_X, DW_X, FN_X,
FMUL_D->	List(Y, Y, N, N, N, N, N, N, N, A2_X, N, M_X, Y, Y, N, Y, N, N, N, CSR.N, N, N, N, Y),	A1_X,	IMM_X, DW_X, FN_X,
FMADD_D->	List(Y, Y, N, N, N, N, N, N, N, A2_X, N, M_X, Y, Y, Y, N, N, N, CSR.N, N, N, N, N, Y),	A1_X,	IMM_X, DW_X, FN_X,
FMSUB_D->	List(Y, Y, N, N, N, N, N, N, A2_X, N, M_X, Y, Y, Y, N, N, N, CSR.N, N, N, N, N, Y),	A1_X,	IMM_X, DW_X, FN_X,
FNMADD_D->	List(Y, Y, N, N, N, N, N, N, A2_X, N, M_X, Y, Y, Y, N, N, N, CSR.N, N, N, N, N, Y),	A1_X,	IMM_X, DW_X, FN_X,

explanation

RVD Instructions

FSGNJP_D (Multiply signs (R Format))

FSGNJP_D (Inject complement of sign (R Format))

FMIN_D (Compute minimum of two values (R Format))

FMAX_D (Compute maximum of two values (R Format))

FADD_D (Add two registers (R Format))

FSUB_D (Subtract two registers (R Format))

FMUL_D (Multiply two registers (R Format))

FMADD_D ((fs1×fs2+fs3) where s1,s2,s3 are source register (R Format))

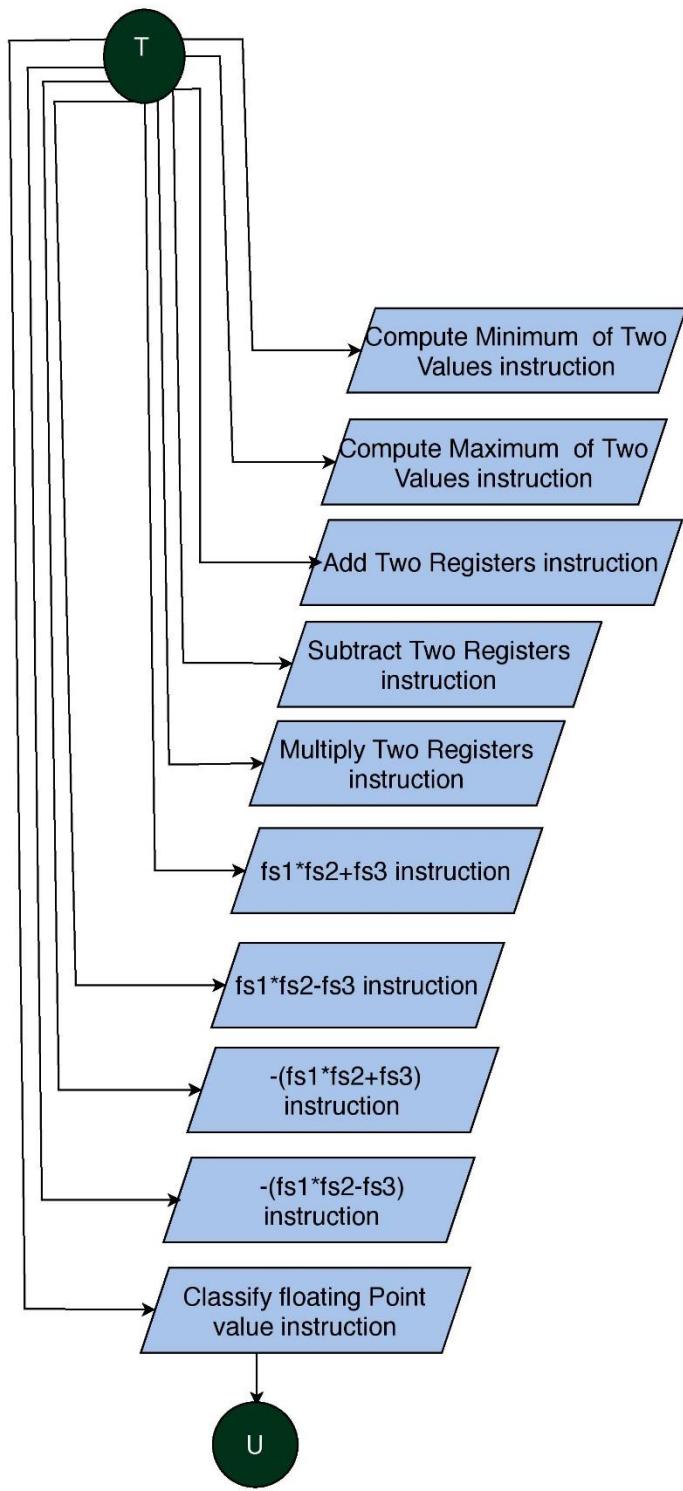
FMSUB_D ((fs1×fs2+fs3) where s1,s2,s3 are source register (R Format))

FNMADD_D (- (fs1×fs2+fs3) where s1,s2,s3 are source register (R Format))

The D extension is structured very similarly to the F extension, and indeed requires the presence of the F extension. The 32 floating-point registers are doubled in width to 64 bits, and new instructions are added that operate on double-precision values. 32-bit and 64-bit floats cannot be freely mixed in computational instructions, but instructions to convert between the formats (FCVT.D.S and FCVT.S.D) are provided to support mixed-format code. We have created class FDecode which is extended by trait DecodeConstants. This class is used for pattern matching of RVF Instructions There are Y and N which are used if there is true then, there will be Y otherwise N. sigs variable is used for sequence in this List.

Example

In these RVD Instructions we have 1st element position as Y which means we have legal instruction. We have 2nd, position as Y which means we have floating point(fp) instruction. 3rd and 4th position as N which means we don't have rocc instruction and branch instruction. We have N on 7th and 8th position which means we have no source registers for other extensions. We have Y at 27th signal indicating that we have dp (double floating point). We have high signal (Y) for source registers rfs1, rfs2, rfs3 and wfd (destination floating register) for RVF instruction which is at 17th, 18th, 19th and 20th position respectively. We have DW_X(data width of Alu) and Aluop bits as input of List(FN_X).



FNMSUB_D->	List(Y, Y, N, N, N, N, N, N, A2_X, N, M_X, Y, Y, Y, N, N, N, CSR.N, N, N, N, Y),	A1_X,	IMM_X, DW_X, FN_X,
FCLASS_D->	List(Y, Y, N, N, N, N, N, N, A2_X, N, M_X, Y, N, N, N, N, Y, CSR.N, N, N, N, Y),	A1_X,	IMM_X, DW_X, FN_X,
FCVT_W_D->	List(Y, Y, N, N, N, N, N, N, A2_X, N, M_X, Y, N, N, N, N, Y, CSR.N, N, N, N, Y),	A1_X,	IMM_X, DW_X, FN_X,
FCVT_WU_D->	List(Y, Y, N, N, N, N, N, N, A2_X, N, M_X, Y, N, N, N, N, Y, CSR.N, N, N, N, Y),	A1_X,	IMM_X, DW_X, FN_X,
FEQ_D->	List(Y, Y, N, N, N, N, N, N, A2_X, N, M_X, Y, Y, N, N, N, Y, CSR.N, N, N, N, Y),	A1_X,	IMM_X, DW_X, FN_X,
FLT_D->	List(Y, Y, N, N, N, N, N, N, A2_X, N, M_X, Y, Y, N, N, N, Y, CSR.N, N, N, N, Y),	A1_X,	IMM_X, DW_X, FN_X,
FLE_D->	List(Y, Y, N, N, N, N, N, N, A2_X, N, M_X, Y, Y, N, N, N, Y, CSR.N, N, N, N, Y),	A1_X,	IMM_X, DW_X, FN_X,
FCVT_D_W->	List(Y, Y, N, N, N, N, N, Y, N, A2_X, N, M_X, N, N, N, Y, N, N, N, CSR.N, N, N, N, Y),	A1_RS1,	IMM_X, DW_X, FN_X,
FCVT_D_WU->	List(Y, Y, N, N, N, N, N, Y, N, A2_X, N, M_X, N, N, N, Y, N, N, N, CSR.N, N, N, N, Y),	A1_RS1,	IMM_X, DW_X, FN_X,

explanation

RVD Instructions

FNMSUB_D (- (fs1×fs2-fs3) where s1,s2,s3 are source register (R Format))

FCLASS_D (Classify floating-point value (R Format))

FMV_X_D (Move from F.P. to integer register (RV64)(R Format))

FCVT_W_D (Convert to signed 32-bit integer (R Format))

FCVT_WU_D (Convert to [un]signed 32-bit integer (R Format))

FEQ_D (Set if equal (R Format))

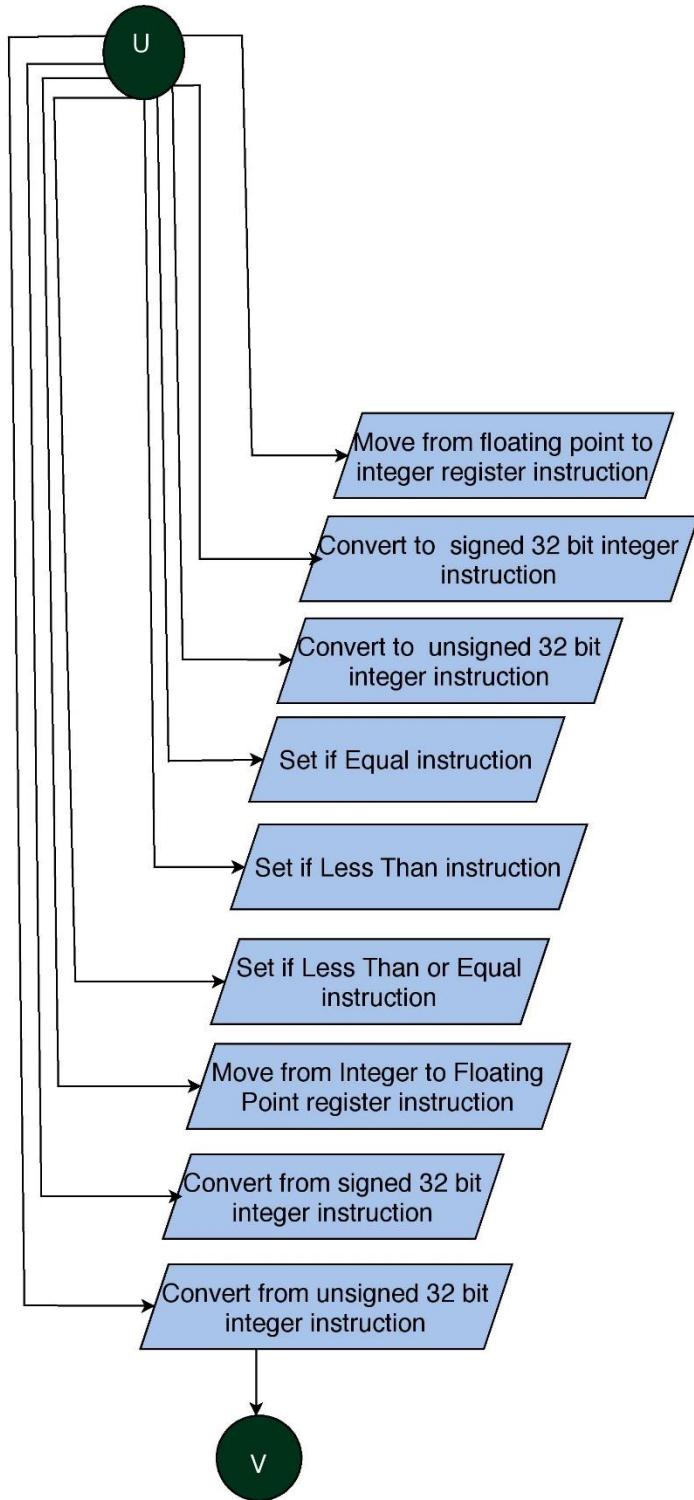
FLT_D (Set if less than (R Format))

FLE_D (Set if less than or equal (R Format))

FMV_D_X (Move from integer to floating-point register (R Format))

FCVT_D_W (Convert from signed 32-bit integer (R Format))

FCVT_D_WU (Convert from unsigned 32-bit integer (R Format))



```

    FLD->      List(Y,Y,N,N,N,N,N,Y,N,A2_IMM, A1_RS1, IMM_I,
DW_XPR,FN_ADD,   Y,M_XRD,           N,N,N,Y,N,N,N,CSR.N,N,N,N,Y),
    FSD->      List(Y,Y,N,N,N,N,N,Y,N,A2_IMM, A1_RS1, IMM_S,
DW_XPR,FN_ADD,   Y,M_XWR,           N,Y,N,N,N,N,N,CSR.N,N,N,N,Y),
    FDIV_D->   List(Y,Y,N,N,N,N,N,A2_X,   A1_X,   IMM_X, DW_X, FN_X,
N,M_X,           Y,Y,N,Y,N,N,N,CSR.N,N,N,N,Y),
    FSQRT_D->  List(Y,Y,N,N,N,N,N,A2_X,   A1_X,   IMM_X, DW_X, FN_X,
N,M_X,           Y,Y,N,Y,N,N,N,CSR.N,N,N,N,Y))
}


```

```

class F64Decode(implicit val p: Parameters) extends DecodeConstants
{
  val table: Array[(BitPat, List[BitPat])] = Array(
    FCVT_L_S-> List(Y,Y,N,N,N,N,N,N,A2_X,   A1_X,   IMM_X, DW_X, FN_X,
N,M_X,           Y,N,N,N,N,Y,CSR.N,N,N,N,N),
    FCVT LU_S-> List(Y,Y,N,N,N,N,N,N,A2_X,   A1_X,   IMM_X, DW_X, FN_X,
N,M_X,           Y,N,N,N,N,Y,CSR.N,N,N,N,N),
    FCVT S_L-> List(Y,Y,N,N,N,N,Y,N,A2_X,   A1_RS1, IMM_X, DW_X, FN_X,
N,M_X,           N,N,Y,N,N,N,CSR.N,N,N,N,N),
    FCVT S LU-> List(Y,Y,N,N,N,N,Y,N,A2_X,   A1_RS1, IMM_X, DW_X, FN_X,
N,M_X,           N,N,Y,N,N,N,CSR.N,N,N,N,N))
}

```

explanation

RVD Instructions

FLW (Load 32-bit floating-point word (I Format))

FDW (Store 32-bit floating-point word (I Format))

FDIV_D (Divide two registers (R Format))

FSQRT_D (Compute square root (R Format))

RVF64 Instructions

FCV_L_S (Convert to signed 64-bit integer (RV64) (R Format))

FCVT_LU_S(Convert to [un]signed 64-bit integer (RV64) (R Format))

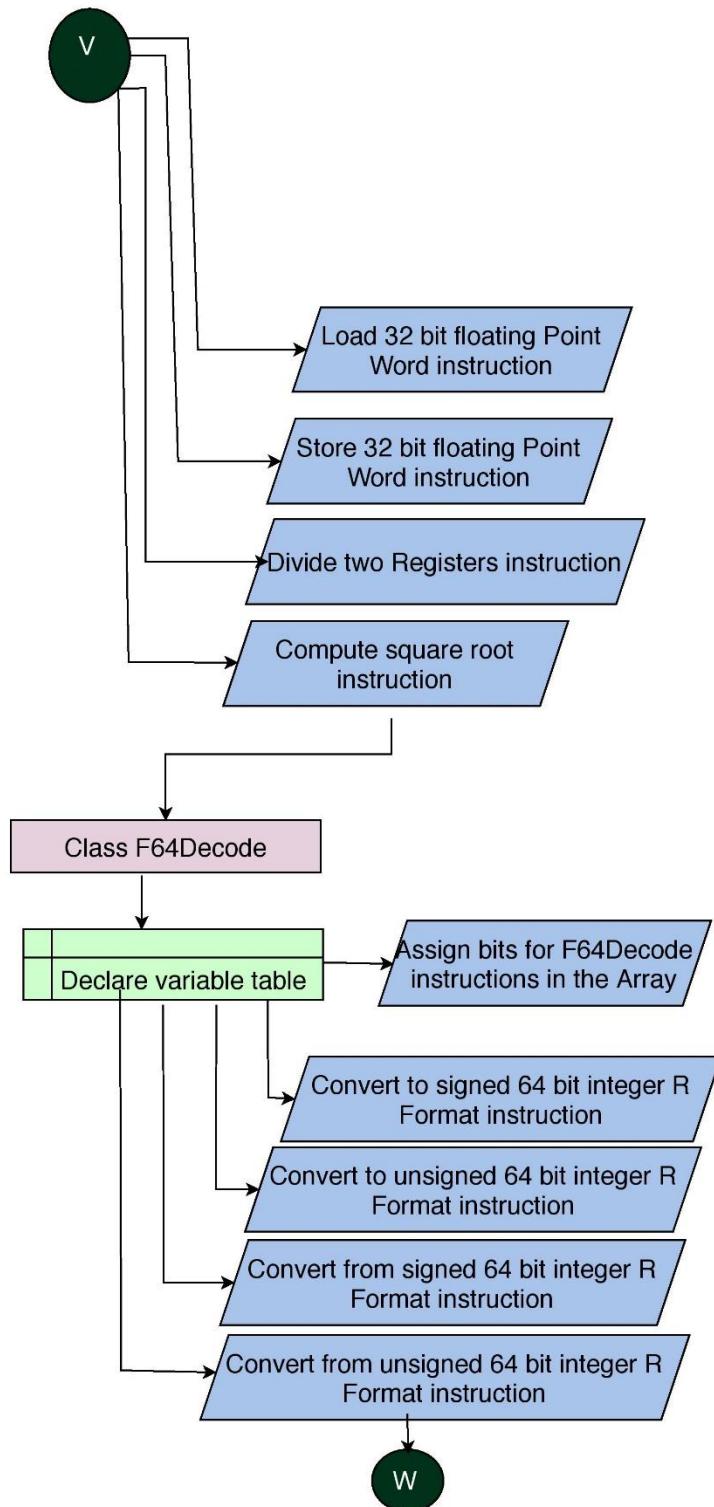
FCVT_S_L (Convert from signed 64-bit integer (RV64) (R Format))

FCVT_S_LU (Convert from [un]signed 64-bit integer (RV64) (R Format))

Class F64Decode is created, which is extended by trait DecodeConstants. This class is used for pattern matching of RVF64 Instructions. There are Y and N which are used if there is true then, there will be Y otherwise N. sigs variable is used for sequence in this List.

Example

In these RVF64 Instructions we have 1st element position as Y which means we have legal instruction. We have 2nd, position as Y which means we have floating point(fp) instruction. 3rd and 4th position as N which means we don't have rocc instruction and branch instruction. We have N on 7th and 8th position which means we have no source registers for other extensions. We have Y at 16th signal indicating that we have memcmd for Load instruction. We have high signal (Y) or Low (N) signal for source registers rfs1, rfs2, rfs3 and wfd (destination floating register) for RVF instruction which is at 17th ,18th, 19th and 20th position respectively. We have DW_X(data width of Alu) and Aluop bits as input of List(FN_X).



```

class D64Decode(implicit val p: Parameters) extends DecodeConstants
{
  val table: Array[(BitPat, List[BitPat])] = Array(
    FMV_X_D-> List(Y,Y,N,N,N,N,N,N,A2_X, A1_X, IMM_X, DW_X, FN_X,
N,M_X, Y,N,N,N,N,Y,CSR.N,N,N,N,Y),
    FCVT_L_D-> List(Y,Y,N,N,N,N,N,N,A2_X, A1_X, IMM_X, DW_X, FN_X,
N,M_X, Y,N,N,N,N,Y,CSR.N,N,N,N,Y),
    FCVT LU_D-> List(Y,Y,N,N,N,N,N,N,A2_X, A1_X, IMM_X, DW_X, FN_X,
N,M_X, Y,N,N,N,N,Y,CSR.N,N,N,N,Y),
    FMV_D_X-> List(Y,Y,N,N,N,N,N,Y,N,A2_X, A1_RS1, IMM_X, DW_X, FN_X,
N,M_X, N,N,N,Y,N,N,CSR.N,N,N,N,Y),
    FCVT_D_L-> List(Y,Y,N,N,N,N,N,Y,N,A2_X, A1_RS1, IMM_X, DW_X, FN_X,
N,M_X, N,N,N,Y,N,N,CSR.N,N,N,N,Y),
    FCVT_D_LU-> List(Y,Y,N,N,N,N,N,Y,N,A2_X, A1_RS1, IMM_X, DW_X, FN_X,
N,M_X, N,N,N,Y,N,N,CSR.N,N,N,N,Y))
}

class SCIEDecode(implicit val p: Parameters) extends DecodeConstants
{
  val table: Array[(BitPat, List[BitPat])] = Array(
    SCIE.opcode-> List(Y,N,N,N,N,N,Y,Y,Y,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_X, N,M_X, N,N,N,N,N,Y,CSR.N,N,N,N,N))
}

```

RVD64 Instructions

FMV_X_D (Move from F.P. to integer register (RV64) (R Format))

FCVT_L_D(Convert to signed 64-bit integer (RV64) (R Format))

FCVT_LU_D (Convert to [un]signed 64-bit integer (RV64) (R Format))

FMV_D_X (Move from integer to F.P. register (RV64) (R Format))

FCVT_D_L(Convert from signed 64-bit integer (RV64) (R Format))

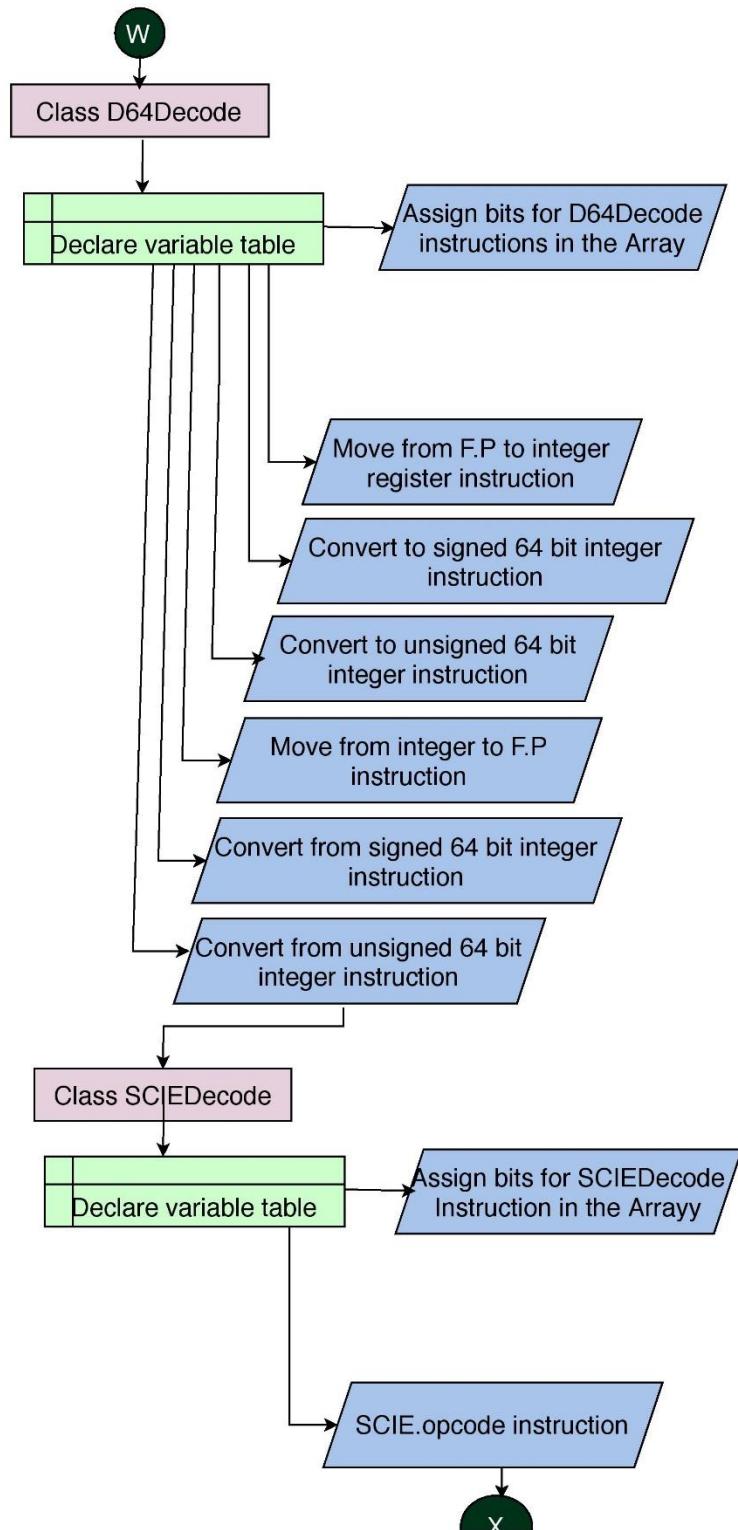
FCVT_D_LU(Convert from [un]signed 64-bit integer (RV64) (R Format))

Class D64Decode is created, which is extended by trait DecodeConstants. This class is used for pattern matching of RVD64 Instructions There are Y and N which are used if there is true then, there will be Y otherwise N. sigs variable is used for sequence in this List.

Example

In these RVD64 Instructions we have 1st element position as Y which means we have legal instruction. We have 2nd, position as Y which means we have floating point(fp) instruction. 3rd and 4th position as N which means we don't have rocc instruction and branch instruction. We have N on 7th and 8th position which means we have no source registers for other extensions. We have Y at 16th signal indicating that we have memcmd for Load instruction. We have high signal (Y) or Low (N) signal for source registers rfs1, rfs2, rfs3 and wfd (destination floating register) for RVF instruction which is at 17th, 18th, 19th and 20th position respectively. We have Y at 27th signal indicating that we have dp (double floating point). We have DW_X(data width of Alu) and Aluop bits as input of List(FN_X).

--scieDecode--



```

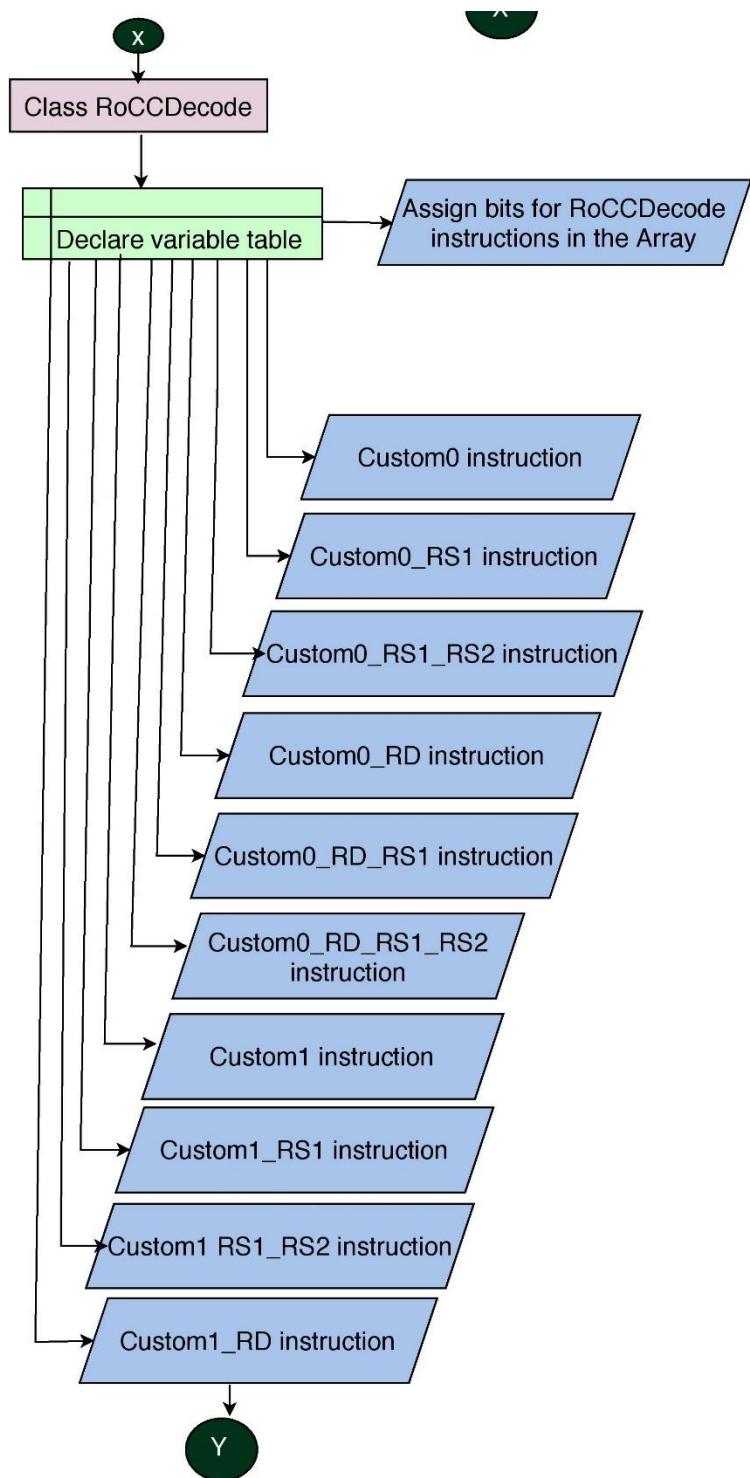
class RoCCDecode(implicit val p: Parameters) extends DecodeConstants
{
    val table: Array[(BitPat, List[BitPat])] = Array(

        CUSTOM0->          List(Y,N,Y,N,N,N,N,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD,      N,M_X,           N,N,N,N,N,N,CSR.N,N,N,N,N),
        CUSTOM0_RS1->       List(Y,N,Y,N,N,N,N,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD,      N,M_X,           N,N,N,N,N,N,CSR.N,N,N,N,N),
        CUSTOM0_RS1_RS2->   List(Y,N,Y,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD,      N,M_X,           N,N,N,N,N,N,CSR.N,N,N,N,N),
        CUSTOM0_RD->        List(Y,N,Y,N,N,N,N,N,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD,      N,M_X,           N,N,N,N,N,Y,CSR.N,N,N,N,N),
        CUSTOM0_RD_RS1->    List(Y,N,Y,N,N,N,N,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD,      N,M_X,           N,N,N,N,N,N,CSR.N,N,N,N,N),
        CUSTOM0_RD_RS1_RS2-> List(Y,N,Y,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD,      N,M_X,           N,N,N,N,N,N,CSR.N,N,N,N,N),
        CUSTOM1->          List(Y,N,Y,N,N,N,N,N,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD,      N,M_X,           N,N,N,N,N,N,CSR.N,N,N,N,N),
        CUSTOM1_RS1->       List(Y,N,Y,N,N,N,N,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD,      N,M_X,           N,N,N,N,N,N,CSR.N,N,N,N,N),
        CUSTOM1_RS1_RS2->   List(Y,N,Y,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD,      N,M_X,           N,N,N,N,N,N,CSR.N,N,N,N,N),
        CUSTOM1_RD->        List(Y,N,Y,N,N,N,N,N,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD,      N,M_X,           N,N,N,N,N,N,CSR.N,N,N,N,N),
    )
}

```

— explanation —

RoCCDecode's explanation is written afterwards

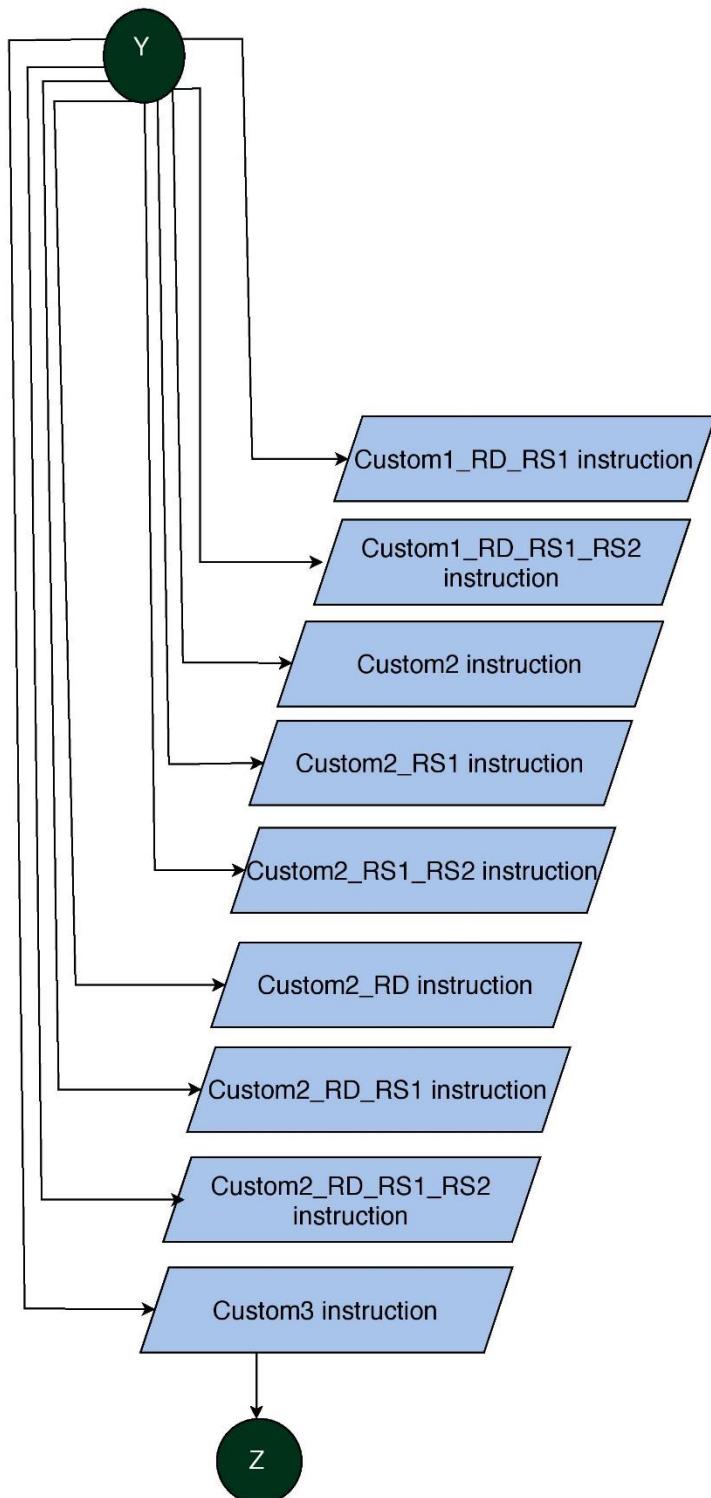


ROCKET-CHIP Micro Architecture Specification Document

```
CUSTOM1_RD_RS1->      List(Y,N,Y,N,N,N,N,Y,N,A2_ZERO,A1_RS1, IMM_X,  
DW_XPR,FN_ADD,      N,M_X,           N,N,N,N,N,N,Y,CSR.N,N,N,N,N),  
  
CUSTOM1_RD_RS1_RS2->List(Y,N,Y,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,  
DW_XPR,FN_ADD,      N,M_X,           N,N,N,N,N,N,Y,CSR.N,N,N,N,N),  
  
CUSTOM2->            List(Y,N,Y,N,N,N,N,N,A2_ZERO,A1_RS1, IMM_X,  
DW_XPR,FN_ADD,      N,M_X,           N,N,N,N,N,N,CSR.N,N,N,N,N),  
  
CUSTOM2_RS1->        List(Y,N,Y,N,N,N,N,Y,N,A2_ZERO,A1_RS1, IMM_X,  
DW_XPR,FN_ADD,      N,M_X,           N,N,N,N,N,N,CSR.N,N,N,N,N),  
  
CUSTOM2_RS1_RS2->List(Y,N,Y,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,  
DW_XPR,FN_ADD,      N,M_X,           N,N,N,N,N,N,CSR.N,N,N,N,N),  
  
CUSTOM2_RD->         List(Y,N,Y,N,N,N,N,N,A2_ZERO,A1_RS1, IMM_X,  
DW_XPR,FN_ADD,      N,M_X,           N,N,N,N,N,Y,CSR.N,N,N,N,N),  
  
CUSTOM2_RD_RS1->List(Y,N,Y,N,N,N,N,Y,N,A2_ZERO,A1_RS1, IMM_X,  
DW_XPR,FN_ADD,      N,M_X,           N,N,N,N,N,N,Y,CSR.N,N,N,N,N),  
  
CUSTOM2_RD_RS1_RS2->List(Y,N,Y,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,  
DW_XPR,FN_ADD,      N,M_X,           N,N,N,N,N,N,Y,CSR.N,N,N,N,N),  
  
CUSTOM3->            List(Y,N,Y,N,N,N,N,N,A2_ZERO,A1_RS1, IMM_X,  
DW_XPR,FN_ADD,      N,M_X,           N,N,N,N,N,N,CSR.N,N,N,N,N),
```

— explanation —

explanation is written afterwards.



```

    CUSTOM3_RS1->      List(Y,N,Y,N,N,N,N,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD,   N,M_X,           N,N,N,N,N,N,N,CSR.N,N,N,N,N),
    CUSTOM3_RS1_RS2->  List(Y,N,Y,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD,   N,M_X,           N,N,N,N,N,N,N,CSR.N,N,N,N,N),
    CUSTOM3_RD->      List(Y,N,Y,N,N,N,N,N,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD,   N,M_X,           N,N,N,N,N,Y,CSR.N,N,N,N,N),
    CUSTOM3_RD_RS1->  List(Y,N,Y,N,N,N,N,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD,   N,M_X,           N,N,N,N,N,Y,CSR.N,N,N,N,N),
    CUSTOM3_RD_RS1_RS2->List(Y,N,Y,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X,
DW_XPR,FN_ADD,   N,M_X,           N,N,N,N,N,Y,CSR.N,N,N,N,N))
}

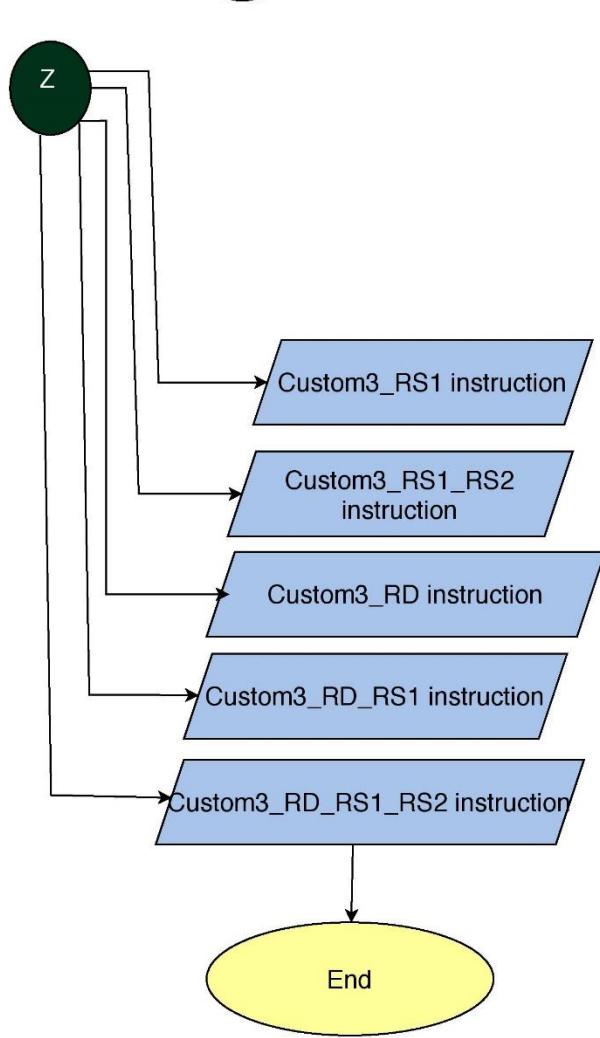
```

— explanation —

Class RoCCDecode is created, which is extended by trait DecodeConstants. This class is used for pattern matching of RoCC Instructions There are Y and N which are used if there is true then, there will be Y otherwise N. sigs variable is used for sequence in this List.

Example

In these RoCC Instructions we have 1st element position as Y which means we have legal instruction. We have 2nd, position as N which means we have floating point(fp) instruction. 3rd position as Y which means we have Rocc instructions. At 4th position we have N which means we don't have rocc instruction and branch instruction. We have N/Y on 7th and 8th position which means whether we have source registers or not . We have N at 16th signal indicating that we do not have memcmd for Load instruction. We have high signal Low (N) signal for source registers rfs1, rfs2, rfs3 and wfd (destination floating register) for RVF instruction which is at 17th ,18th, 19th and 20th position respectively. We have N at 27th signal indicating that we do not have dp (double floating point). We have DW_XPR(data width of Alu) and Aluop bits as input of List(FN_ADD).



RVC

DEEP DIVE INTO CODE

```

class ExpandedInstruction extends Bundle {
    val bits = UInt(32.W)
    val rd = UInt(5.W)
    val rs1 = UInt(5.W)
    val rs2 = UInt(5.W)
    val rs3 = UInt(5.W)
}

class RVCDecoder(x: UInt, xLen: Int) {
    def inst(bits: UInt, rd: UInt = x(11,7), rs1: UInt = x(19,15), rs2: UInt = x(24,20), rs3: UInt = x(31,27)) = {
        val res = Wire(new ExpandedInstruction)
        res.bits := bits
        res.rd := rd
        res.rs1 := rs1
        res.rs2 := rs2
        res.rs3 := rs3
        res
    }
}

```

Explanation

Class ExpandedInstruction

This class has Immutable Variable (bits) having bits width (width=32)

Immutable Variable includes
rd (destination register)
rs1 (source register 1)
rs2,(source register 2)
rs3(source register 3) (width=5)

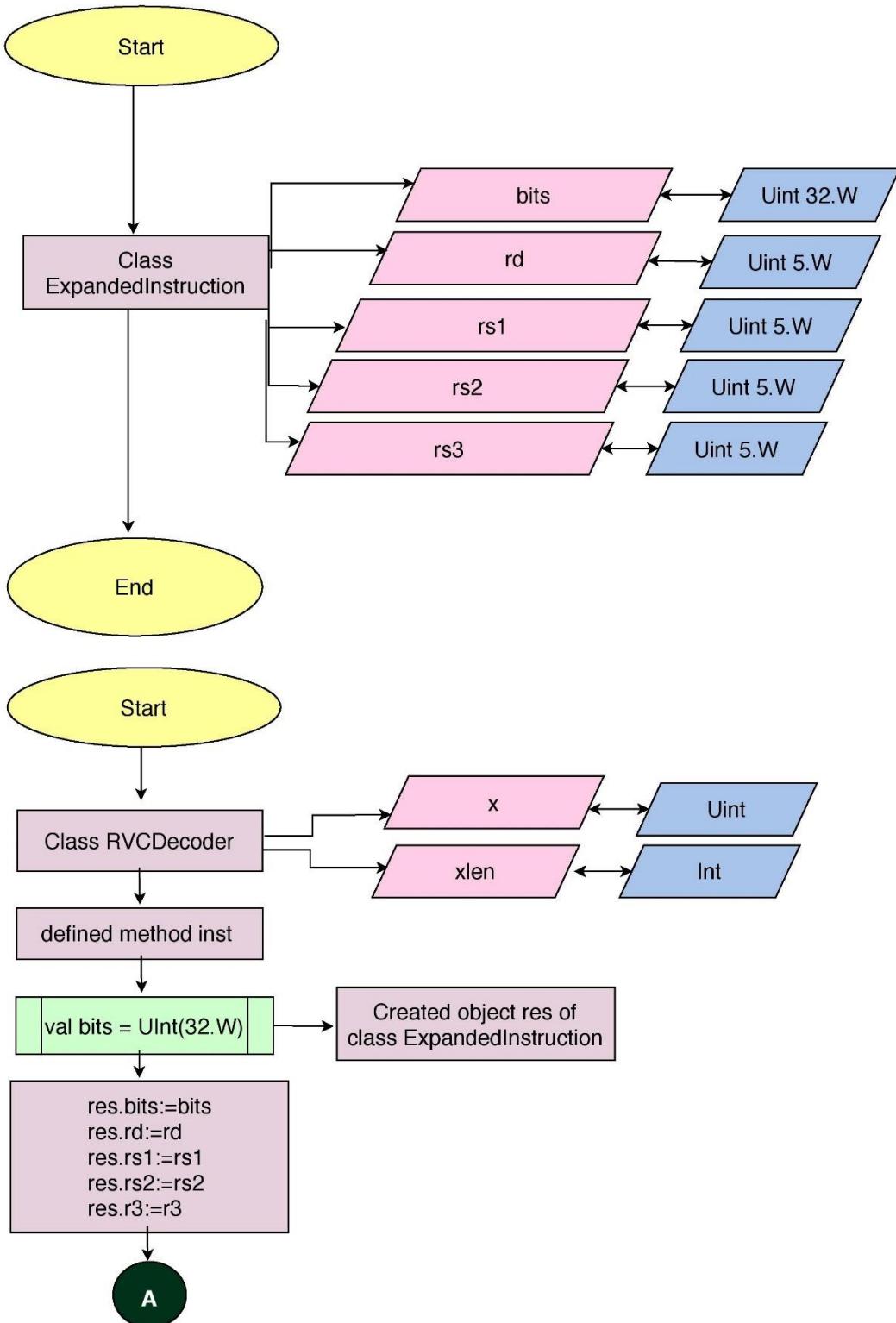
Class RVCDecoder

This code comprises of class RVC Decoder which has
Inputs parameters: x,len
where

x,len are input parameter of class RVCDecoder

This class has a reference variable res from the class of ExpandedInstruction

All variables of ExpandedInstruction to the parameters of method instWe wired all variables of ExpandedInstruction to the parameters of method inst



```

def rs1p = Cat(1.U(2.W), x(9,7))

def rs2p = Cat(1.U(2.W), x(4,2))

def rs2 = x(6,2)

def rd = x(11,7)

def addi4spnImm = Cat(x(10,7), x(12,11), x(5), x(6), 0.U(2.W))

def lwImm = Cat(x(5), x(12,10), x(6), 0.U(2.W))

def ldImm = Cat(x(6,5), x(12,10), 0.U(3.W))

def lwsplImm = Cat(x(3,2), x(12), x(6,4), 0.U(2.W))

def ldspImm = Cat(x(4,2), x(12), x(6,5), 0.U(3.W))

def swsplImm = Cat(x(8,7), x(12,9), 0.U(2.W))

def sdspImm = Cat(x(9,7), x(12,10), 0.U(3.W))

def luiImm = Cat(Fill(15, x(12)), x(6,2), 0.U(12.W))

```

 explanation

Class has method inst has been defined in which extraction of bits for rd,rs1,rs2 and rs3.where rd is destination register and rs1,rs2 and rs3 are source register

Object of class ExpandedInstruction in variable res has been created

All bits of rd,rs1,rs2 and rs3 of method inst wired to object variable res

Method of rs1p of width 2 has been defined in which concatenation bits of x (7 to 9)

Method of rs2p of width 2 has been defined in which concatenation bits of x (2 to 4)

Method rs2 of width 5 has been defined in which extraction bits of x (2 to 6)

Method rd of width 5 has been defined in which extraction bits of x (7 to 11)

Method addi4spnImm has been defined in which extraction bits of x (5 to 10)

Method lwImm (load word immediate) has been defined in which concatenation bits of x

Method ldImm (load data immediate) has been defined in which concatenation bits of x

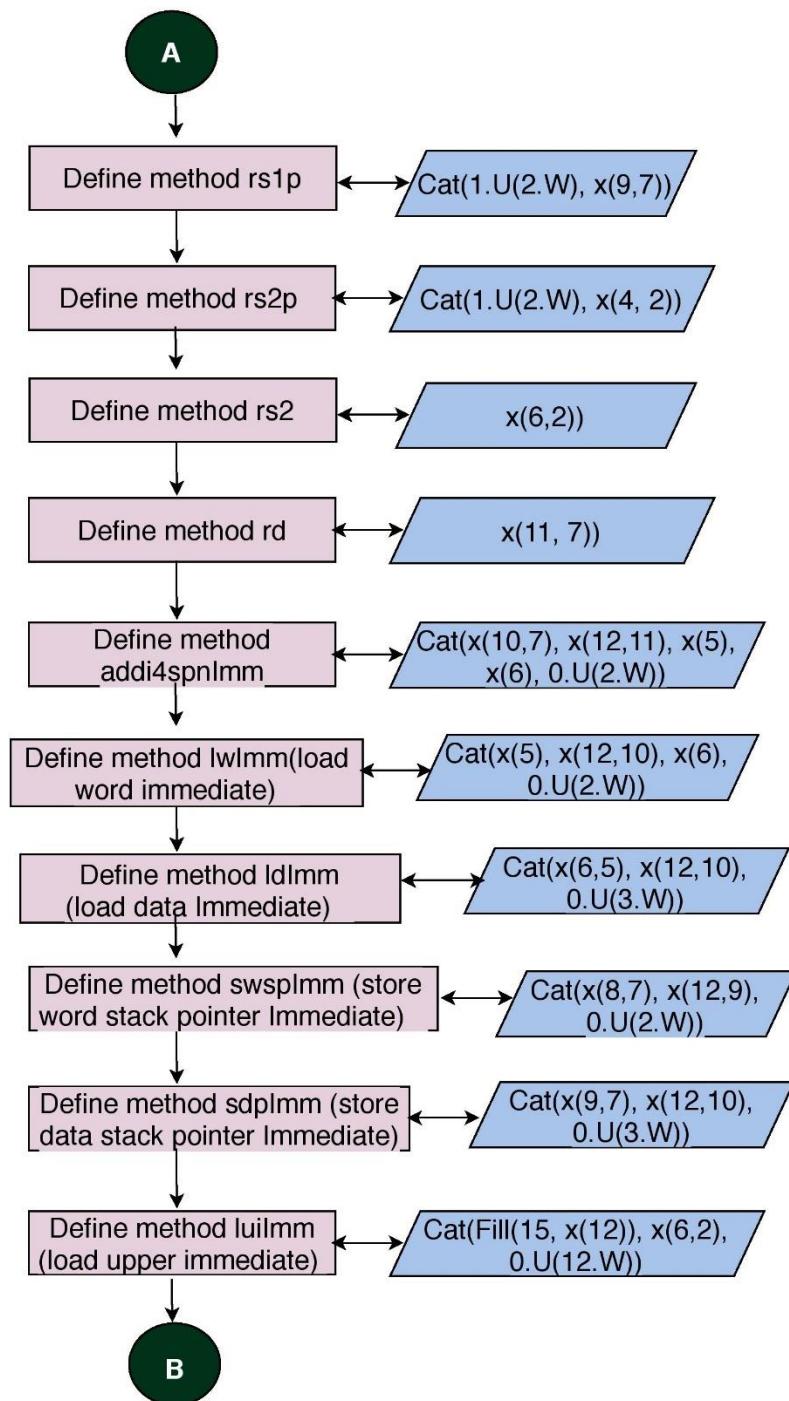
Method lwsplImm (load word stack pointer immediate) has been defined in which concatenation bits of x

Method ldspImm (load data stack pointer immediate) has been defined in which concatenation bits of x

Method swsplImm (store word stack pointer immediate) has been defined in which concatenation bits of x

Method `sdsplimm` (store data stack pointer immediate) has been defined in which concatenation bits of x

Method `luilimm` (load upper immediate) has been defined in which concatenation bits of x



```

def addi16spImm = Cat(Fill(3, x(12)), x(4,3), x(5), x(2), x(6), 0.U(4.W))

def addiImm = Cat(Fill(7, x(12)), x(6,2))

def jImm = Cat(Fill(10, x(12)), x(8), x(10,9), x(6), x(7), x(2), x(11),
x(5,3), 0.U(1.W))

def bImm = Cat(Fill(5, x(12)), x(6,5), x(2), x(11,10), x(4,3), 0.U(1.W))

def shamt = Cat(x(12), x(6,2))

def x0 = 0.U(5.W)

def ra = 1.U(5.W)

def sp = 2.U(5.W)

```

— explanation —

Method addi16splmm (add immediate of 16 bit stack pointer) has been defined in which extraction bits of x and concatenation them.Uses Fill command

Define method addilimm (add immediate) has been defined in which extraction bits of x and concatenation them.Uses Fill command

Method jlmm (jump immediate) has been defined in which extraction bits of x and concatenation them.Uses Fill command

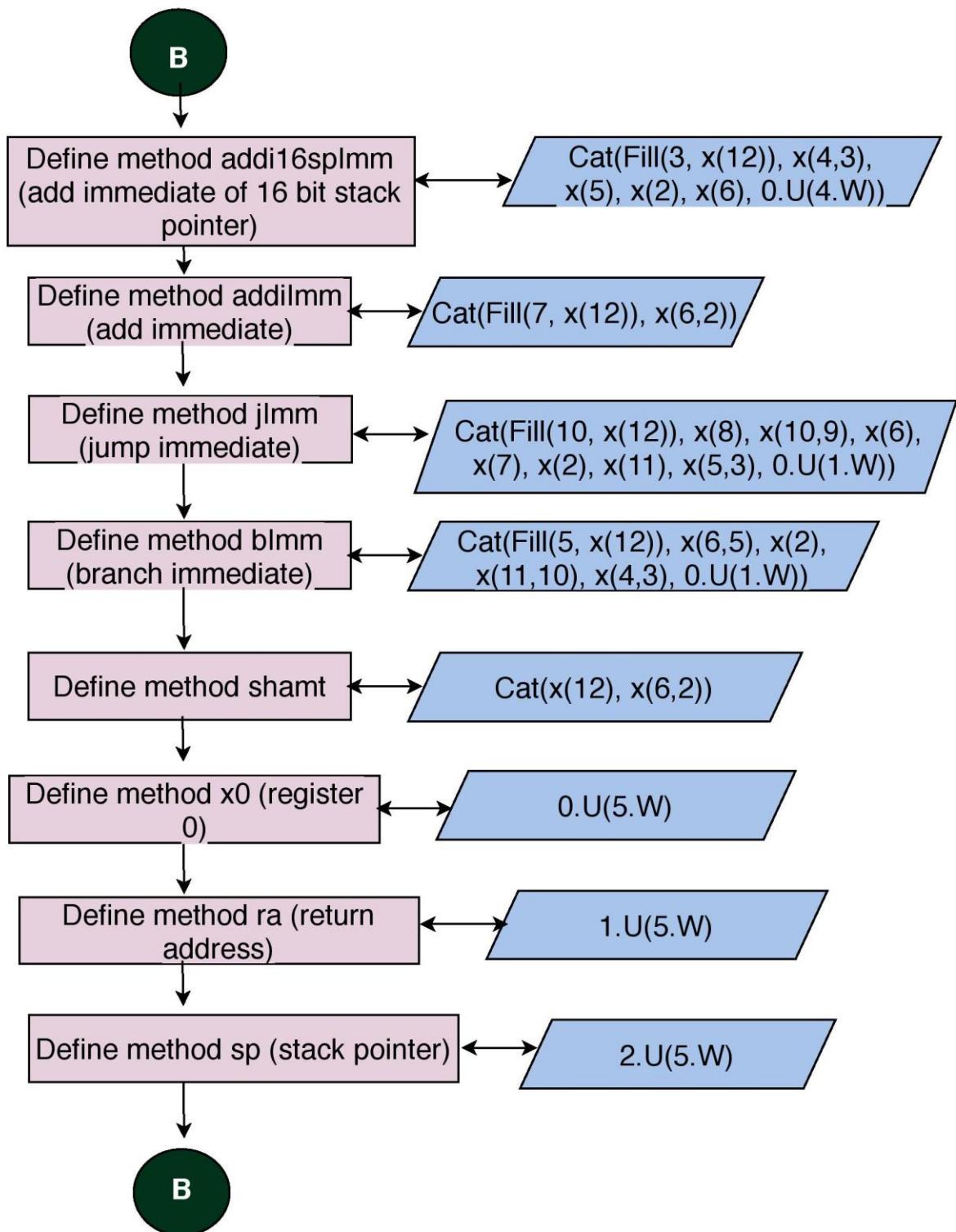
Method blmm (branch immediate) has been defined in which extraction bits of x and concatenation them.Uses Fill command

Method shamt has been defined in which extraction bits of x and concatenation them.

Method x0 (register 0) having width of 5

Method ra (return address) having width of 5

Method sp (stack pointer) having width of 5



```

def q0 = {
    def addi4spn = {
        val opc = Mux(x(12,5).orR, 0x13.U(7.W), 0x1F.U(7.W))
        inst(Cat(addi4spnImm, sp, 0.U(3.W), rs2p, opc), rs2p, sp, rs2p)
    }
    def ld = inst(Cat(ldImm, rs1p, 3.U(3.W), rs2p, 0x03.U(7.W)), rs2p,
    rs1p, rs2p)
    def lw = inst(Cat(lwImm, rs1p, 2.U(3.W), rs2p, 0x03.U(7.W)), rs2p,
    rs1p, rs2p)
    def fld = inst(Cat(ldImm, rs1p, 3.U(3.W), rs2p, 0x07.U(7.W)), rs2p,
    rs1p, rs2p)
    def flw = {
        if (xLen == 32) inst(Cat(lwImm, rs1p, 2.U(3.W), rs2p, 0x07.U(7.W)),
        rs2p, rs1p, rs2p)
        else ld
    }
}

```

 explanation

Method q0

Method addi4spn(add immediate stack pointer) has been defined in main method of q0

Declared variable opc

Mux selection has been defined in which select two conditions
 0x13.U will be output of Mux having width 7
 0x1F.U will be output of Mux having width 7

return instruction to method addi4sp (add immediate for stack pointer)

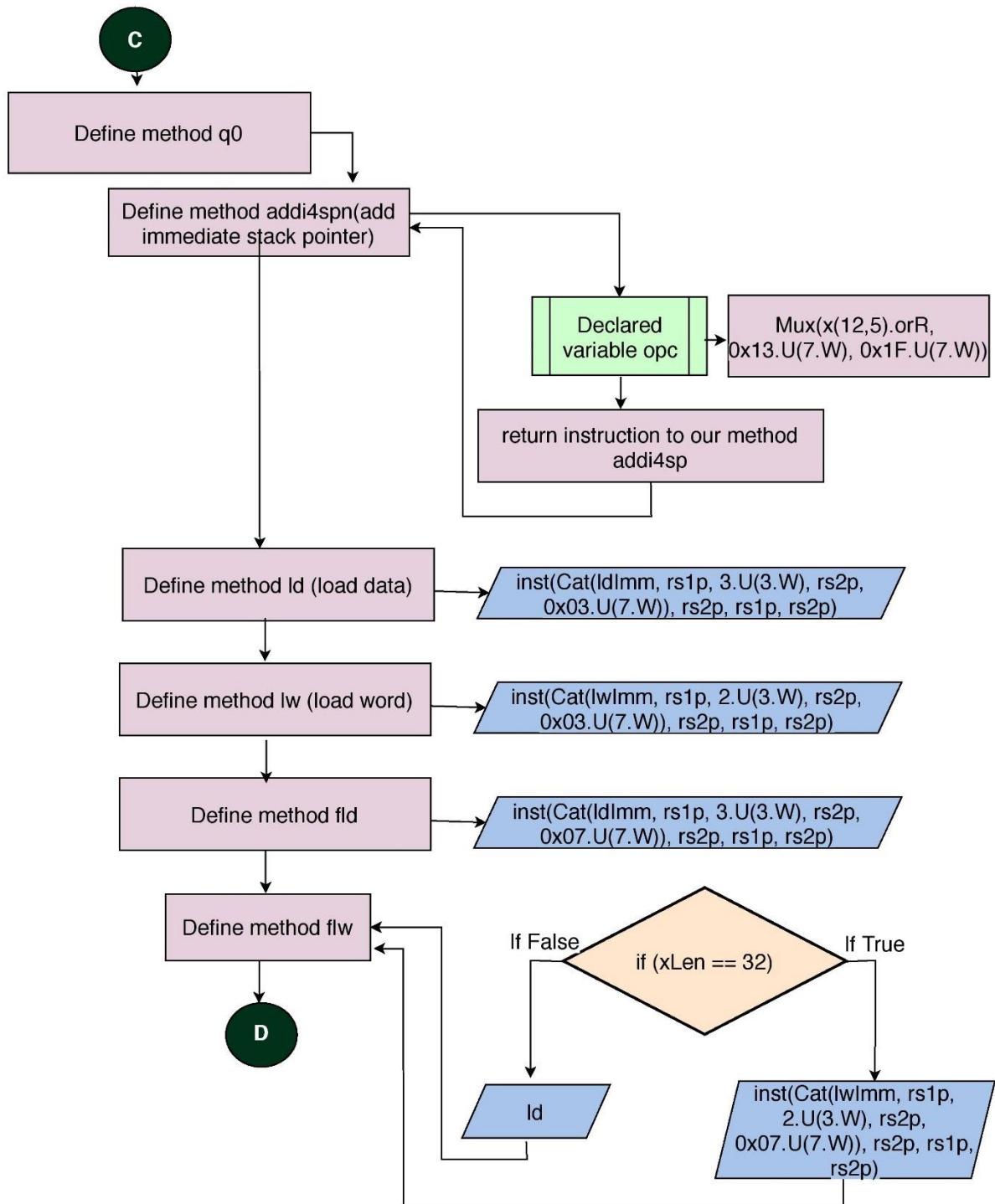
Method ld (load data) which will return inst(instruction)

Method lw (load word) which will return inst(instruction)

Method fld (floating load data) which will return inst(instruction)

Method flw (floating load word) which will return inst(instruction)

IF condition (xLen==32) has been defined in which instruction and method ld result as output



```

def unimp = inst(Cat(lwImm >> 5, rs2p, rs1p, 2.U(3.W), lwImm(4,0),
0x3F.U(7.W)), rs2p, rs1p, rs2p)

def sd = inst(Cat(ldImm >> 5, rs2p, rs1p, 3.U(3.W), ldImm(4,0),
0x23.U(7.W)), rs2p, rs1p, rs2p)

def sw = inst(Cat(lwImm >> 5, rs2p, rs1p, 2.U(3.W), lwImm(4,0),
0x23.U(7.W)), rs2p, rs1p, rs2p)

def fsd = inst(Cat(ldImm >> 5, rs2p, rs1p, 3.U(3.W), ldImm(4,0),
0x27.U(7.W)), rs2p, rs1p, rs2p)

def fsw = {

    if (xLen == 32) inst(Cat(lwImm >> 5, rs2p, rs1p, 2.U(3.W),
lwImm(4,0), 0x27.U(7.W)), rs2p, rs1p, rs2p)

    else sd

}

Seq(addi4spn, fld, lw, flw, unimp, fsd, sw, fsw)
}

```

— explanation —

Method unimp which will return inst (instruction)

Method sd (store data) which will return inst (instruction)

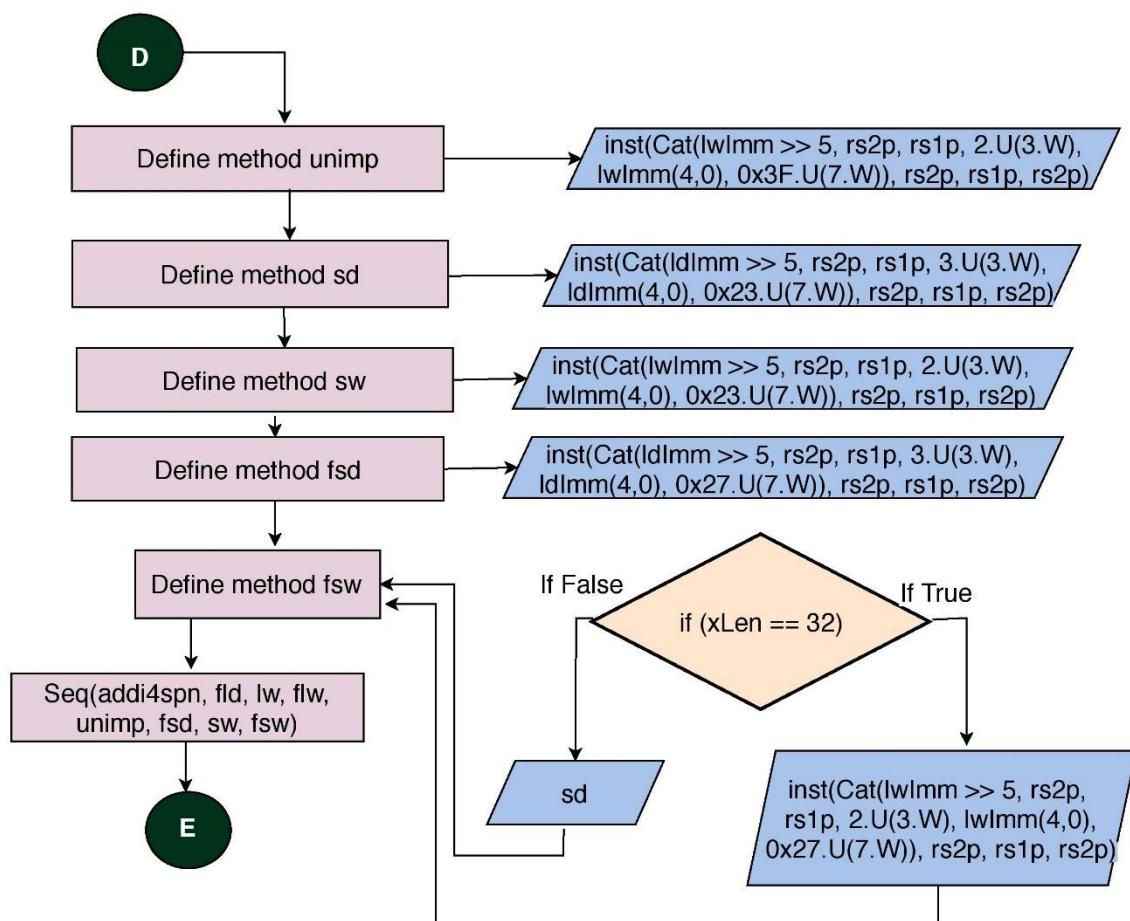
Method sw (store word) which will return inst (instruction)

Method fsd(floating store data) which will return inst(instruction)

Method fsw(floating store word)

IF condition (xLen==32) has been defined in which instruction and method sd result as output

return Seq(Data Structure) of all methods to main method q0



```

def q1 = {

    def addi = inst(Cat(addiImm, rd, 0.U(3.W), rd, 0x13.U(7.W)), rd, rd,
rs2p)

    def addiw = {

        val opc = Mux(rd.orR, 0x1B.U(7.W), 0x1F.U(7.W))

        inst(Cat(addiImm, rd, 0.U(3.W), rd, opc), rd, rd, rs2p)

    }

    def jal = {

        if (xLen == 32) inst(Cat(jImm(20), jImm(10,1), jImm(11), jImm(19,12),
ra, 0x6F.U(7.W)), ra, rd, rs2p)

        else addiw

    }

    def li = inst(Cat(addiImm, x0, 0.U(3.W), rd, 0x13.U(7.W)), rd, x0,
rs2p)

    def addi16sp = {

        val opc = Mux(addiImm.orR, 0x13.U(7.W), 0x1F.U(7.W))

        inst(Cat(addi16spImm, rd, 0.U(3.W), rd, opc), rd, rd, rs2p)

    }
}

```

 explanation

Method q1

Method addi(add immediate) has been defined in main method of q1

Method addiw(add immediate word) has been defined in main method of q1

Declared variable opc

Mux selection has been defined in which select two conditions

0x1B.U will be output of Mux having width 7

0x1F.U will be output of Mux having width 7

return instruction to method addiw (add immediate word)

Method jal(jump and link) has been defined in main method of q1

IF condition (xLen==32) has been defined in which instruction and method addiw result as output

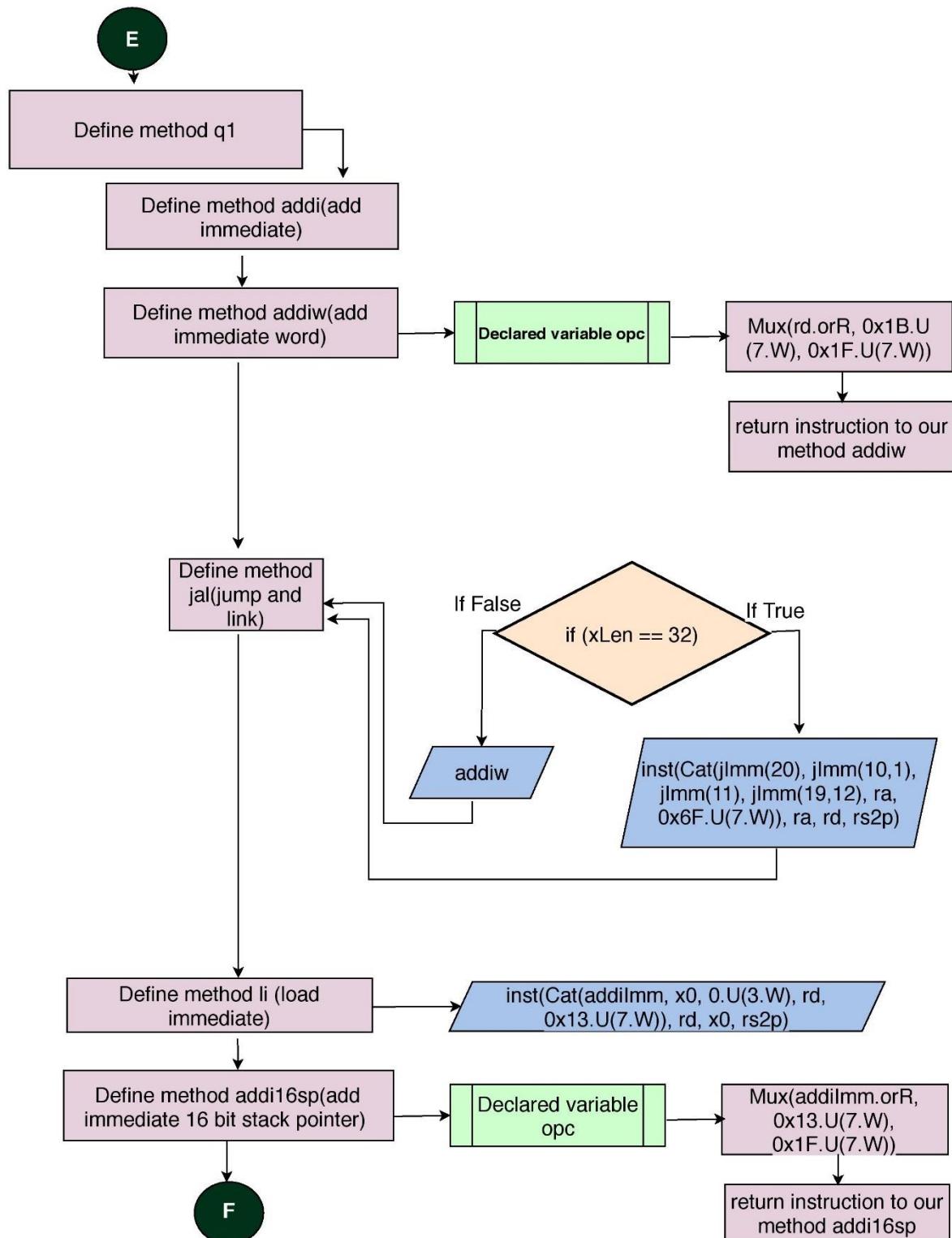
Method li(load immediate) has been defined in main method of q1

Method addi16sp(add immediate 16 bit stack pointer) has been defined in main method of q1

Declared variable opc in method addi16sp

Mux selection has been defined in which select two conditions
 0x13.U will be output of Mux having width 7
 0x1F.U will be output of Mux having width 7

return instruction to method addi16sp (add immediate word)



```

def lui = {
    val opc = Mux(addiImm.orR, 0x37.U(7.W), 0x3F.U(7.W))
    val me = inst(Cat(luiImm(31,12), rd, opc), rd, rd, rs2p)
    Mux(rd === x0 || rd === sp, addi16sp, me)
}

def j = inst(Cat(jImm(20), jImm(10,1), jImm(11), jImm(19,12), x0,
0x6F.U(7.W)), x0, rs1p, rs2p)

def beqz = inst(Cat(bImm(12), bImm(10,5), x0, rs1p, 0.U(3.W),
bImm(4,1), bImm(11), 0x63.U(7.W)), rs1p, rs1p, x0)

def bnez = inst(Cat(bImm(12), bImm(10,5), x0, rs1p, 1.U(3.W),
bImm(4,1), bImm(11), 0x63.U(7.W)), x0, rs1p, x0)

def arith = {
    def srl = Cat(shamt, rs1p, 5.U(3.W), rs1p, 0x13.U(7.W))
    def srai = srl | (1 << 30).U
    def andi = Cat(addiImm, rs1p, 7.U(3.W), rs1p, 0x13.U(7.W))
}

```

 explanation

Method lui(load upper immediate) has been defined in main method of q1

Declared variable opc

Mux selection has been defined in which select two conditions
 0x13.U will be output of Mux having width 7
 0x1F.U will be output of Mux having width 7

Declared variable me

Mux selection has been defined in which select two conditions
 method result of addi16sp will be output of Mux
 variable me result will be output of Mux

Method j(jump) has been defined in main method of q1

Method beqz(branch equals to zero) has been defined in main method of q1

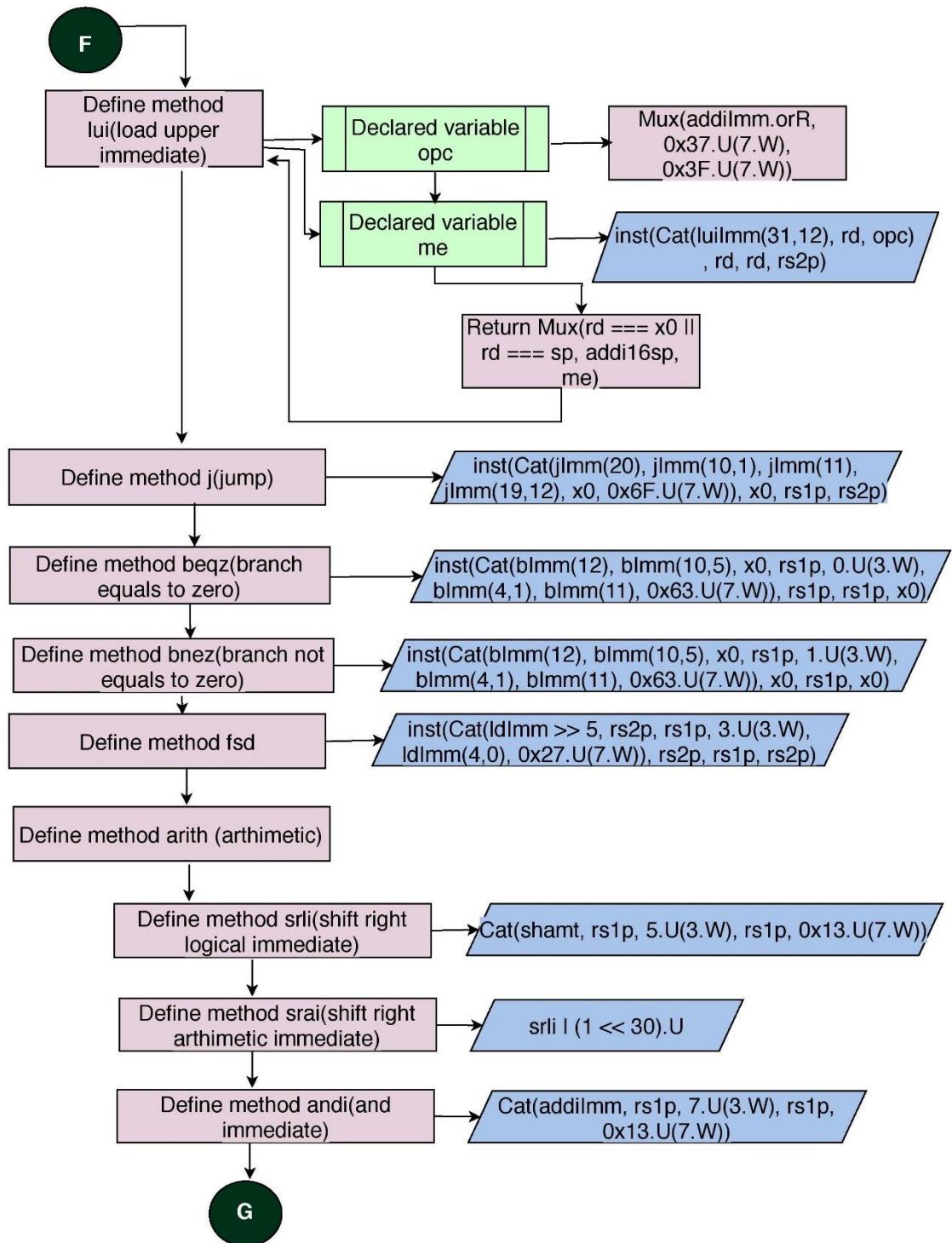
Method bnez(branch not equals to zero) has been defined in main method of q1

Method arith (arithmetic)

Method srl(shift right logical immediate) has been defined in sub method arith

Method srai(shift right arithmetic immediate) has been defined in sub method arith

Method andi(and immediate) has been defined in sub method arith



```

def rtype = {

    val funct = Seq(0.U, 4.U, 6.U, 7.U, 0.U, 0.U, 2.U, 3.U) (Cat(x(12),
x(6,5)))

    val sub = Mux(x(6,5) === 0.U, (1 << 30).U, 0.U)

    val opc = Mux(x(12), 0x3B.U(7.W), 0x33.U(7.W))

    Cat(rs2p, rs1p, funct, rs1p, opc) | sub

}

inst(Seq(srli, srai, andi, rtype)(x(11,10)), rs1p, rs1p, rs2p)

}

Seq(addi, jal, li, lui, arith, j, beqz, bnez)

}

```

 explanation

Method rtype has been defined

Declare variable funct (function) in method r type

Declare variable sub in method r type

Mux selection has been defined in which select two conditions
logical shift left performs
0.U result will be output of Mux

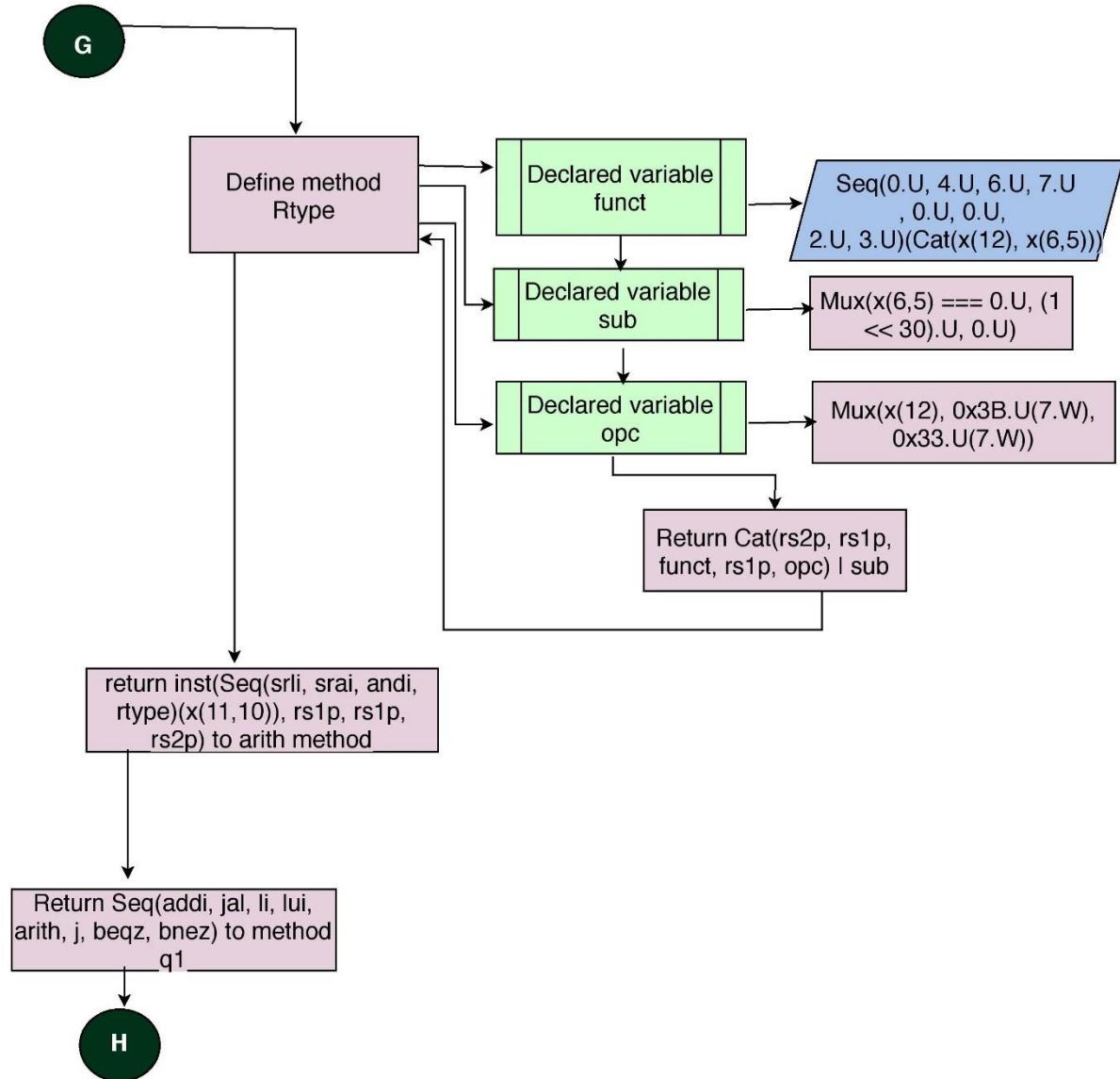
Declare variable opc

Mux selection has been defined in which select two conditions
0x3B.U will be output of Mux having width 7
0x33.U will be output of Mux having width 7

return all variable result

return instruction to method arith (add immediate word)

return sequence of method to main method q1



```

def q2 = {
    val load_opc = Mux(rd.orR, 0x03.U(7.W), 0x1F.U(7.W))
    def slli = inst(Cat(shamt, rd, 1.U(3.W), rd, 0x13.U(7.W)), rd, rd, rs2)
    def ldsp = inst(Cat(ldspImm, sp, 3.U(3.W), rd, load_opc), rd, sp, rs2)
    def lwsp = inst(Cat(lwspImm, sp, 2.U(3.W), rd, load_opc), rd, sp, rs2)
    def fldsp = inst(Cat(ldspImm, sp, 3.U(3.W), rd, 0x07.U(7.W)), rd, sp,
rs2)
    def flwsp = {
        if (xLen == 32) inst(Cat(lwspImm, sp, 2.U(3.W), rd, 0x07.U(7.W)), rd,
sp, rs2)
        else ldsp
    }
}

```

— explanation —

Declare method q2

Declare variable load_opc in method q2

Mux selection has been defined in which select two conditions

0x03.U will be output of Mux having width 7

0x1F.U will be output of Mux having width 7

Method slli(shift left logical immediate) has been defined in main method q2

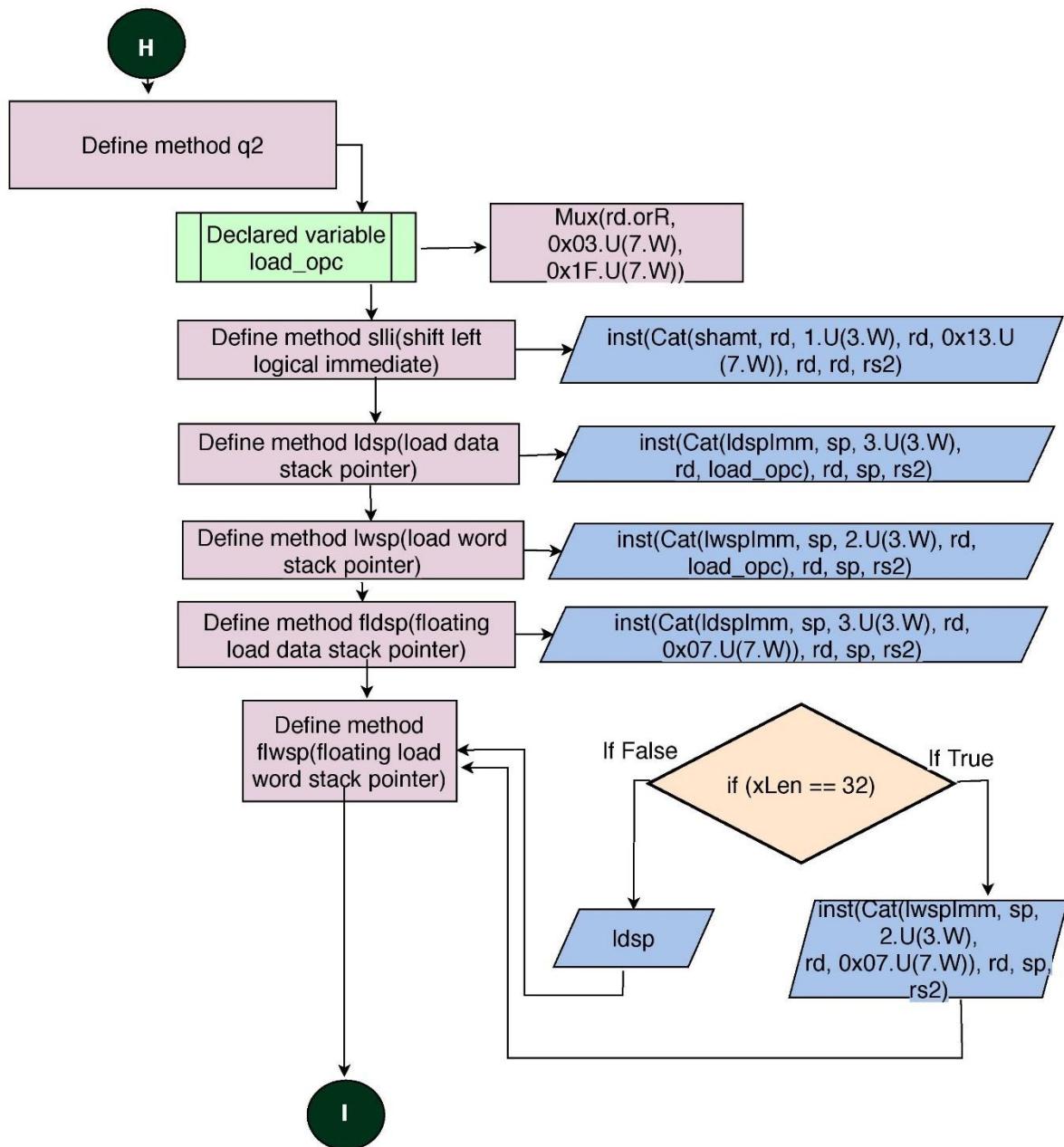
Method ldsp(load data stack pointer) has been defined in main method q2

Method lwsp(load word stack pointer) has been defined in main method q2

Method fldsp(floating load data stack pointer) has been defined in main method q2

Method flwsp(floating load word stack pointer) has been defined in main method q2

IF condition (xLen==32) has been defined in which instruction and method ldsp result as output



```

def sdsp = inst(Cat(sdspImm >> 5, rs2, sp, 3.U(3.W), sdspImm(4,0),
0x23.U(7.W)), rd, sp, rs2)

def swsp = inst(Cat(swspImm >> 5, rs2, sp, 2.U(3.W), swspImm(4,0),
0x23.U(7.W)), rd, sp, rs2)

def fsdsp = inst(Cat(sdspImm >> 5, rs2, sp, 3.U(3.W), sdspImm(4,0),
0x27.U(7.W)), rd, sp, rs2)

def fswsp = {

    if (xLen == 32) inst(Cat(swspImm >> 5, rs2, sp, 2.U(3.W),
swspImm(4,0), 0x27.U(7.W)), rd, sp, rs2)

    else sdsp

}

def jalr = {

    val mv = inst(Cat(rs2, x0, 0.U(3.W), rd, 0x33.U(7.W)), rd, x0, rs2)

    val add = inst(Cat(rs2, rd, 0.U(3.W), rd, 0x33.U(7.W)), rd, rd, rs2)

    val jr = Cat(rs2, rd, 0.U(3.W), x0, 0x67.U(7.W))
}

```

— explanation —

Method sdsp(store data stack pointer) has been defined in main method q2

Method swsp(store word stack pointer) has been defined in main method q2

Method fsdsp(floating store data stack pointer) has been defined in main method q2

Method fswsp(floating store word stack pointer) has been defined in main method q2

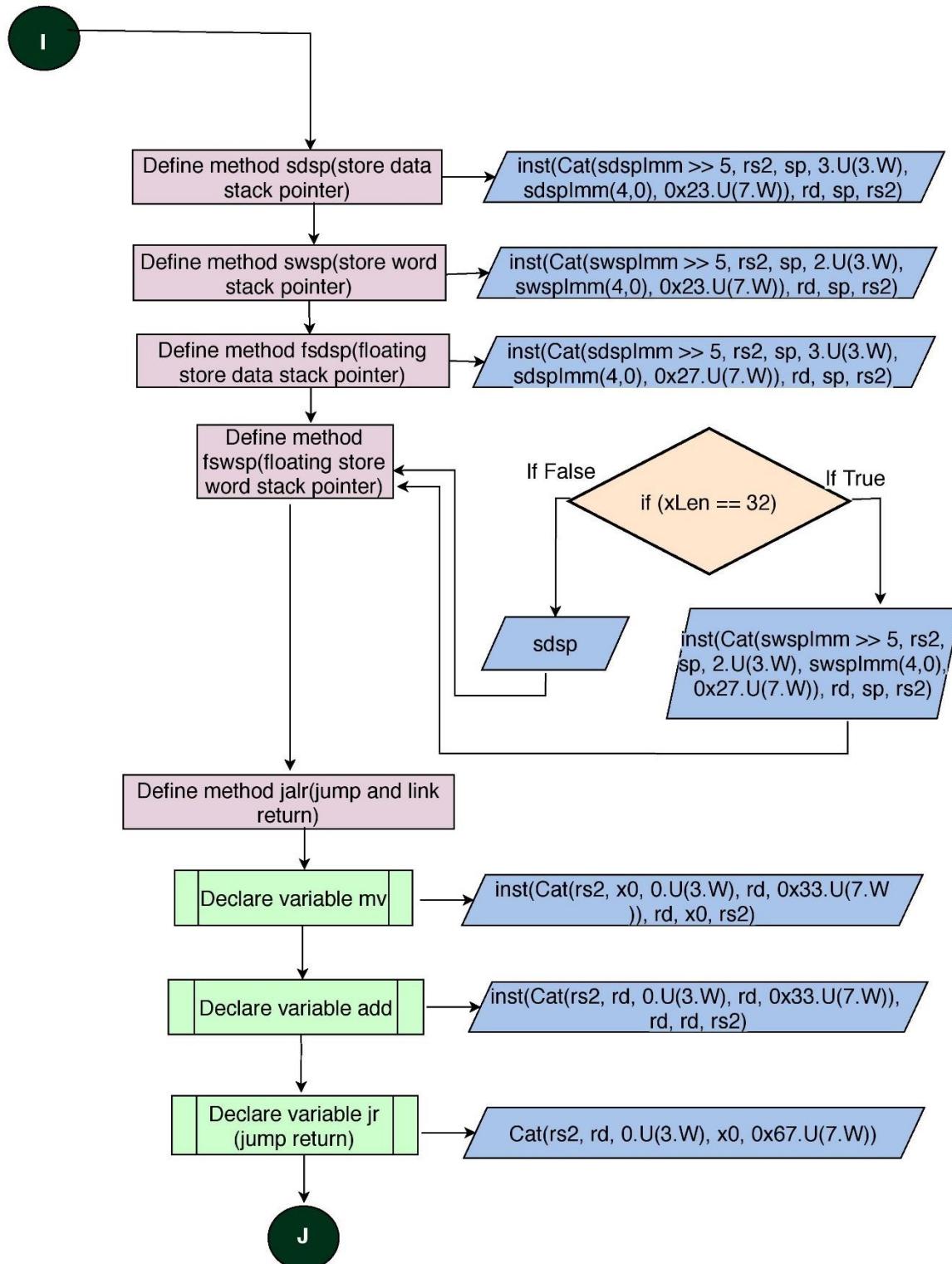
If condition (xLen==32) has been defined in which instruction and method sdsp result as output

Method jalr(jump and link return) has been defined in main method q2

Declare variable mv which returns instruction

Declare variable add which will return instruction

Declare variable jr (jump return)



```

val reserved = Cat(jr >> 7, 0x1F.U(7.W))

val jr_reserved = inst(Mux(rd.orR, jr, reserved), x0, rd, rs2)

val jr_mv = Mux(rs2.orR, mv, jr_reserved)

val jalr = Cat(rs2, rd, 0.U(3.W), ra, 0x67.U(7.W))

val ebreak = Cat(jr >> 7, 0x73.U(7.W)) | (1 << 20).U

val jalr_ebreak = inst(Mux(rd.orR, jalr, ebreak), ra, rd, rs2)

val jalr_add = Mux(rs2.orR, add, jalr_ebreak)

Mux(x(12), jalr_add, jr_mv)

}

Seq(slli, fldsp, lwsp, flwsp, jalr, fsdsp, swsp, fswsp)

}

```

— explanation —

Declare variable reserved

Declare variable jr_reserved which will return instruction

Declare variable jr_mv

Mux selection has been defined in which select two conditions
 mv variable result will be output of Mux
 jr_reserved_variable will be output of Mux

Declare variable jalr (jump and link return)

Declare variable ebreak

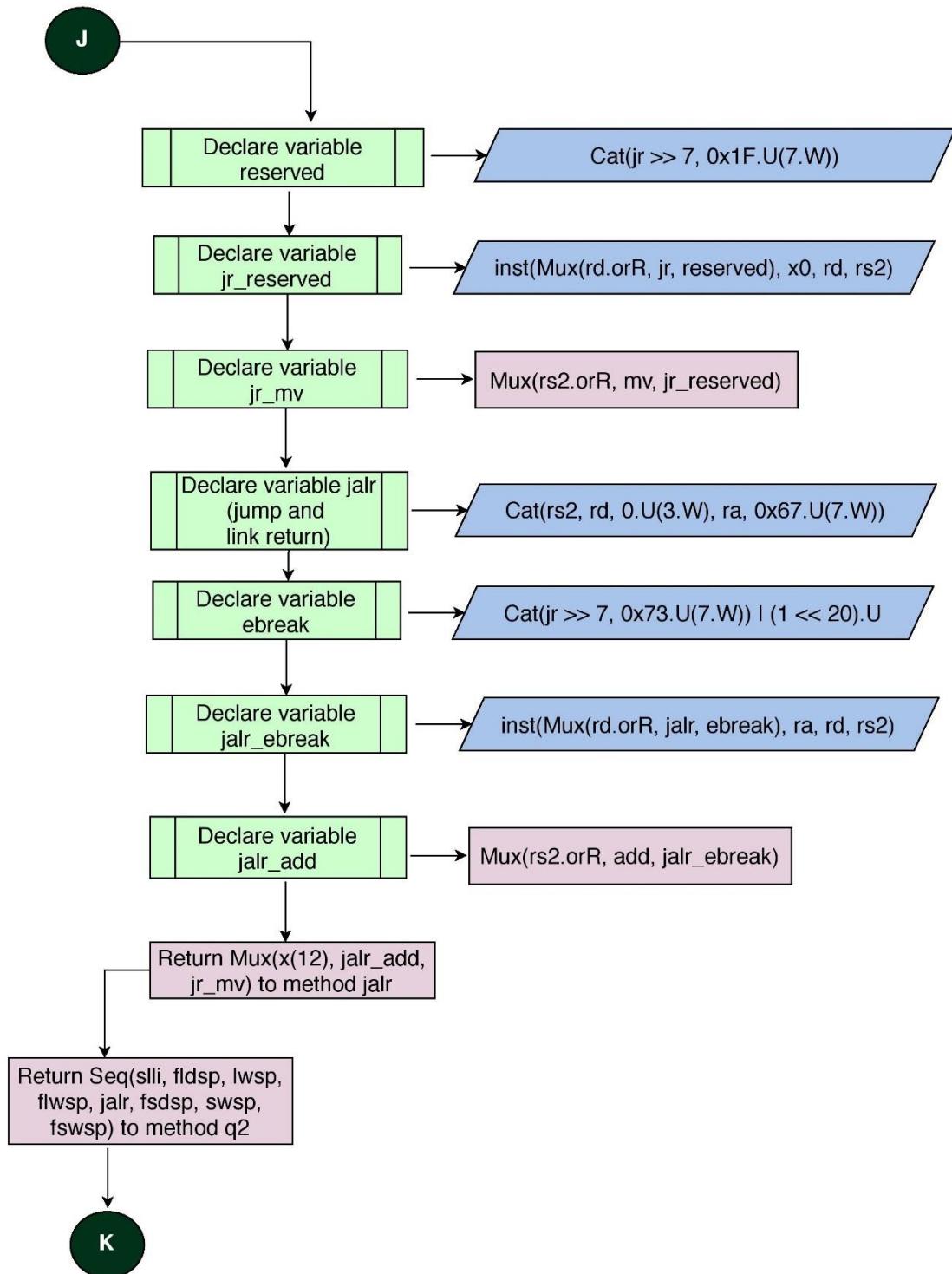
Declare variable jalr_ebreak

Declare variable jalr_add

Mux selection has been defined in which select two conditions
 add variable result will be output of Mux
 jalr_ebreak variable will be output of Mux

Mux selection has been defined in which select two conditions
 jalr variable result will be output of Mux
 jr_mv variable will be output of Mux

return sequence of methods to main method q2



```

def q3 = Seq.fill(8)(passthrough)

def passthrough = inst(x)

def decode = {

  val s = q0 ++ q1 ++ q2 ++ q3
  s(Cat(x(1,0), x(15,13)))
}

}

```

— explanation —

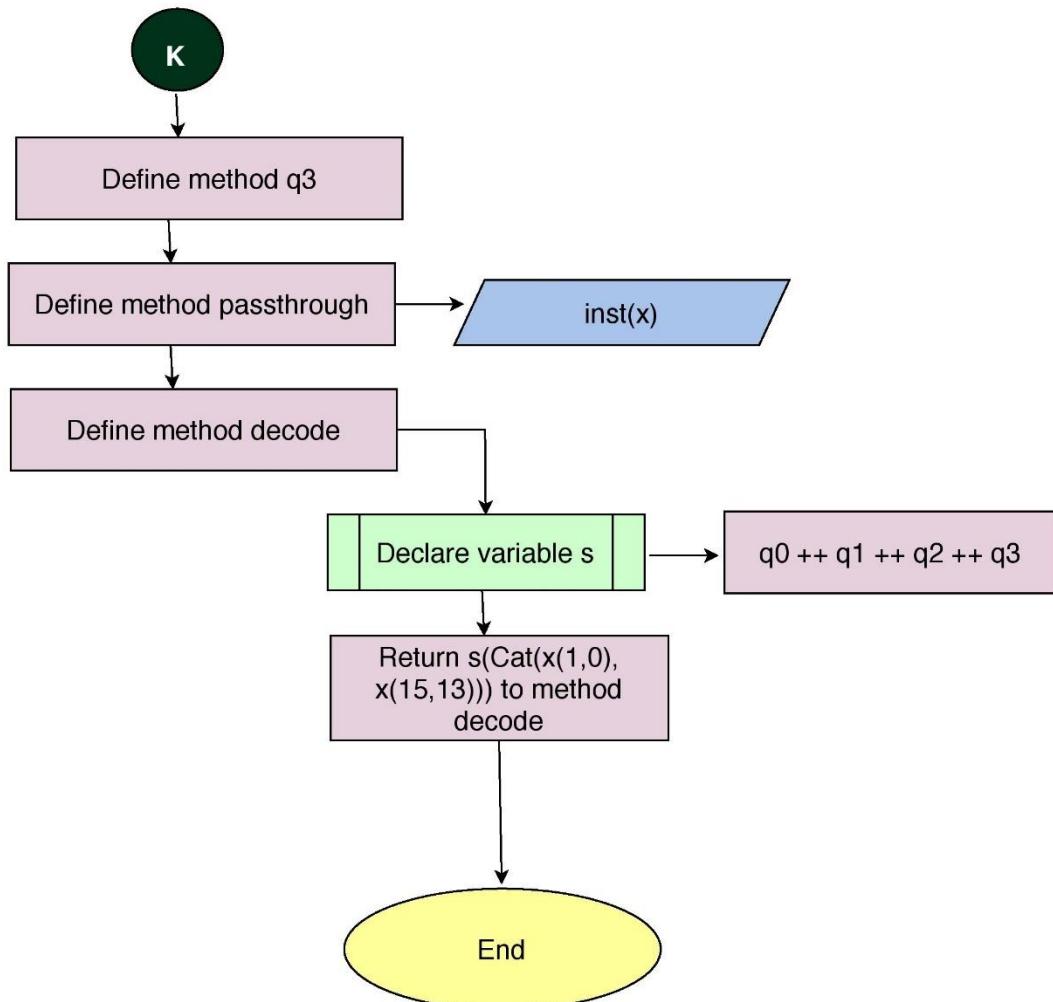
Method q3 is defined

Method passthrough is defined

Method decode is defined

Declare variable s in which concatenation mains methods q1,q2 and q3

return result of variable s and concatenation with x



```

class RVCExpander(implicit val p: Parameters) extends Module with
HasCoreParameters {

    val io = IO(new Bundle {
        val in = Input(UInt(32.W))
        val out = Output(new ExpandedInstruction)
        val rvc = Output(Bool())
    })

    if (usingCompressed) {
        io.rvc := io.in(1,0) /= 3.U
        io.out := new RVCDecoder(io.in, p(XLen)).decode
    } else {
        io.rvc := false.B
        io.out := new RVCDecoder(io.in, p(XLen)).passthrough
    }
}

```

 explanation

Create new object IO

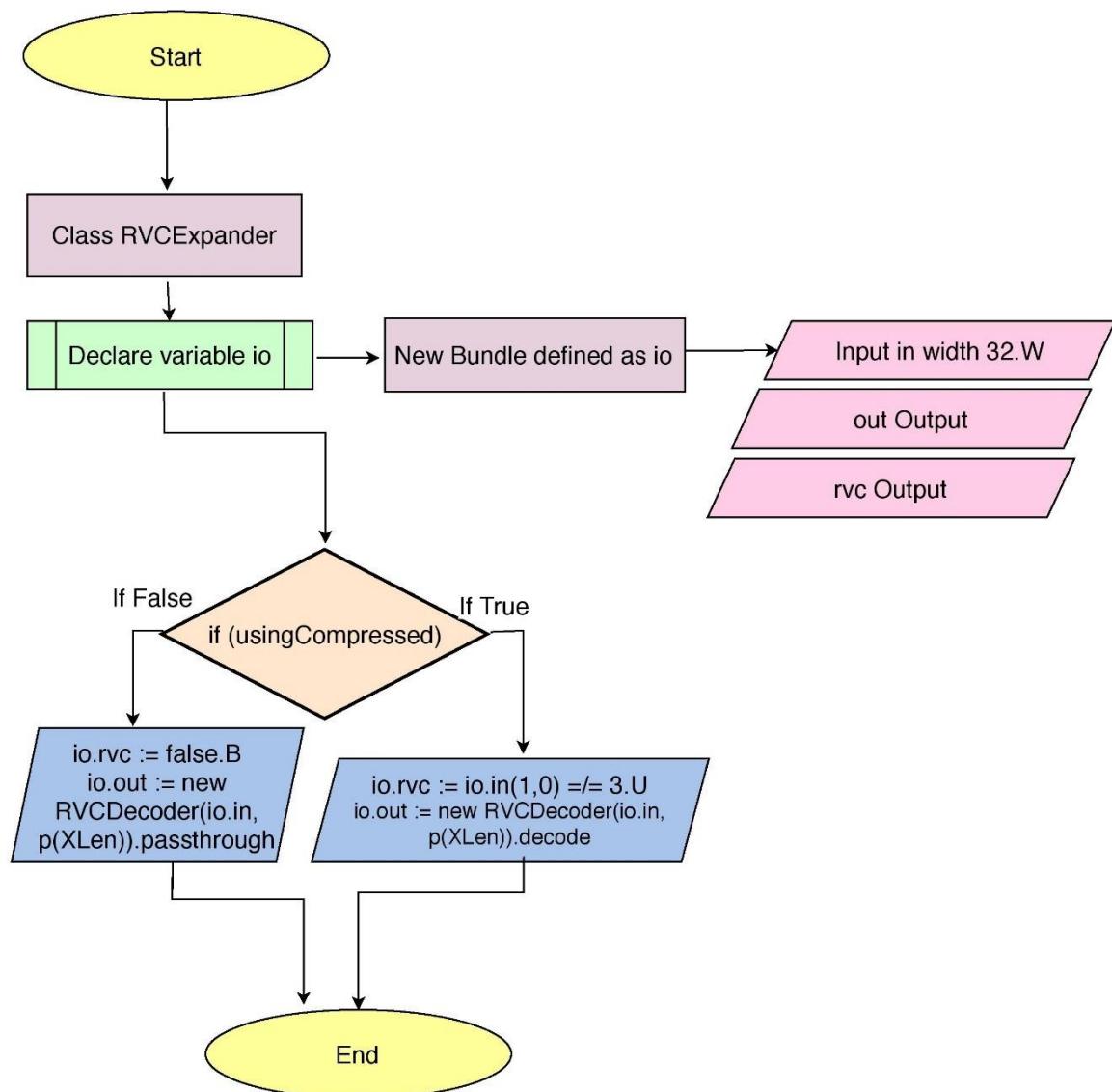
Input variable in of width 32

Output variable out has been defined in which new object created from class ExpandedInstruction

Output variable out

IF condition (xLen==32) has been defined in which rvc result as output

result of output by creating new object of class RVCDecoder



Scoreboard

DEEP DIVE INTO CODE

```

class Scoreboard(n: Int, zero: Boolean = false)
{

def set(en: Bool, addr: UInt): Unit = update(en, _next | mask(en, addr))
def clear(en: Bool, addr: UInt): Unit = update(en, _next & ~mask(en, addr))
def read(addr: UInt): Bool = r(addr)
def readBypassed(addr: UInt): Bool = _next(addr)
private val _r = Reg(init=Bits(0, n))
private val r = if (zero) (_r >> 1 << 1) else _r
private var _next = r
private var ens = Bool(false)
private def mask(en: Bool, addr: UInt) = Mux(en, UInt(1) << addr, UInt(0))
private def update(en: Bool, update: UInt) = {
    _next = update
    ens = ens || en
    when (ens) { _r := _next }
}
}
}

```

— explanation —

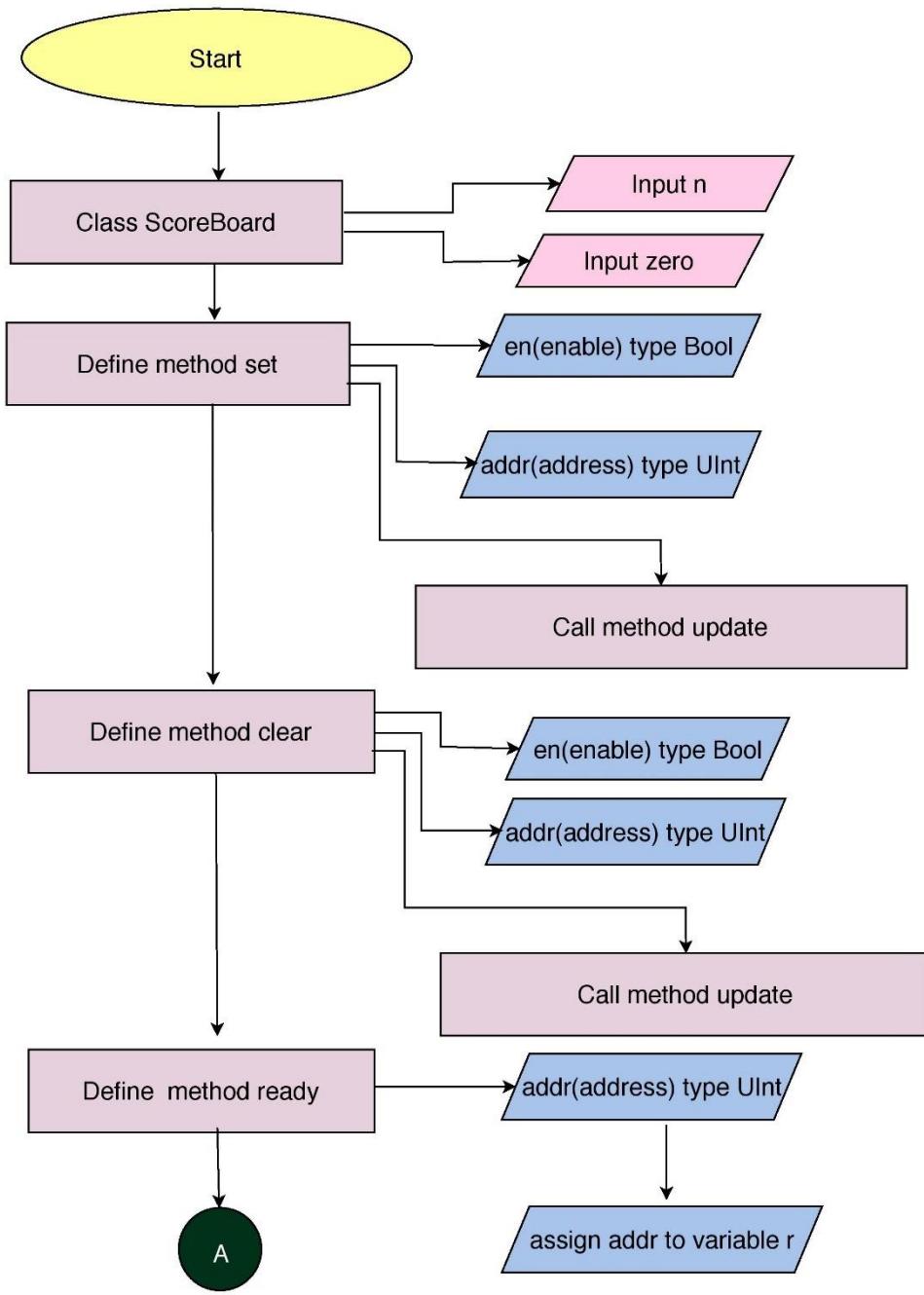
Define class of Scoreboard and pass two parameters n and zero as input. n will decide scoreboard bit and zero will be false as default

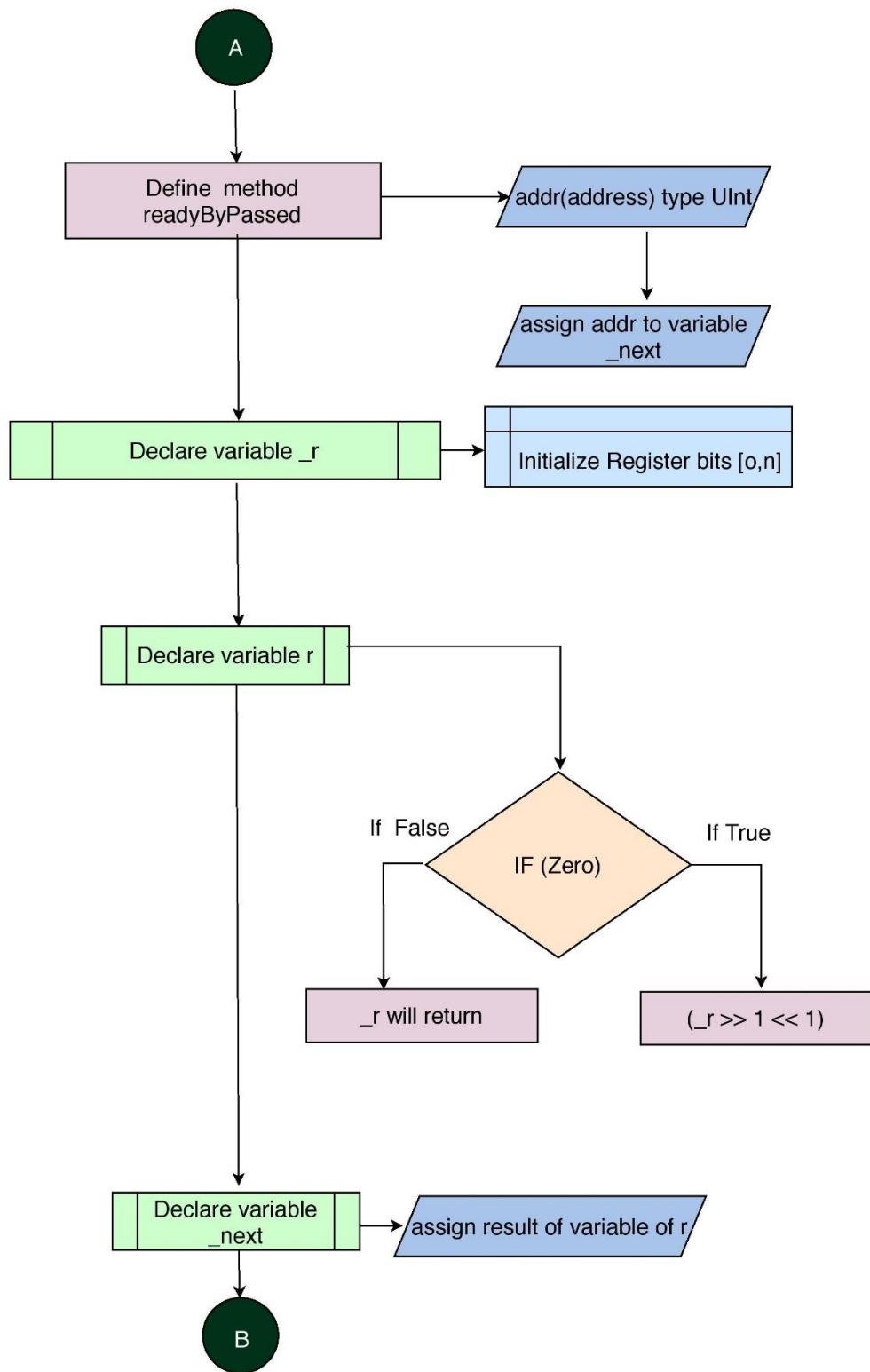
Define method set, clear, read and readBypassed for setting scoreboard, reading data, and clearing scoreboard once it is used. en and addr are the enable and addresses used in the method of scoreboard

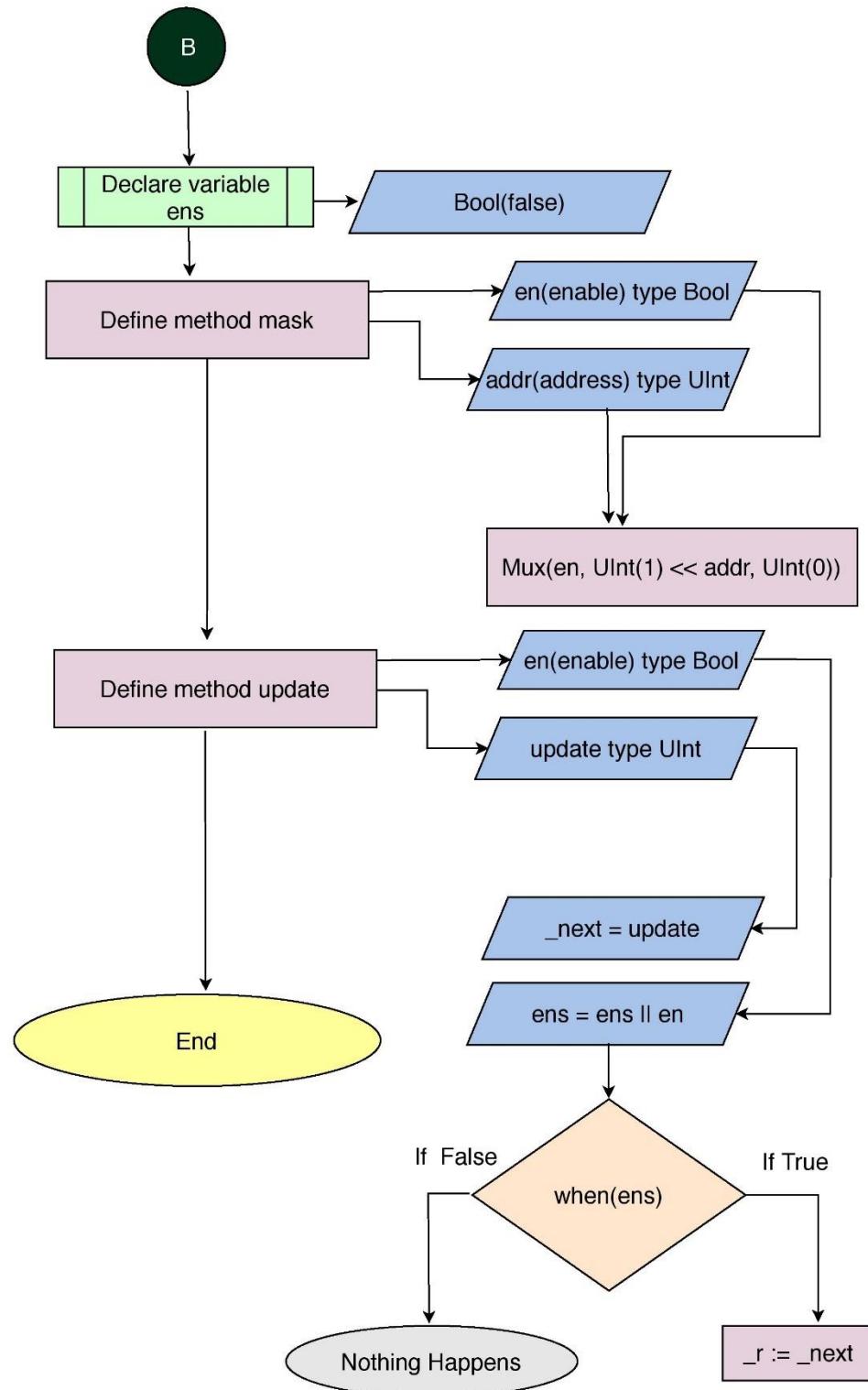
Used en and addr parameters in these methods ,en parameter will enable these methods and addr will indicated address

Define method mask and update which uses in scoreboard method

Update method will update the status of scoreboard.







```

val sboard = new Scoreboard(32, true)
private val _r = Reg(init=Bits(0, n))
private val r = if (zero) (_r >> 1 << 1) else _r
sboard.clear(ll_wen, ll_waddr)
def id_sboard_clear_bypass(r: UInt) = {
if (!tileParams.dcache.get.dataECC.isDefined) ll_wen && ll_waddr === r
else div.io.resp.fire() && div.io.resp.bits.tag === r || dmem_resp_replay
&& dmem_resp_xpu && dmem_resp_waddr === r
val id_sboard_hazard = checkHazards(hazard_targets, rd => sboard.read(rd)
&& !id_sboard_clear_bypass(rd))
sboard.set(wb_set_sboard && wb_wen, wb_waddr)

```

explanation

We create scoreboard object sboard by giving class parameters as n=32 and zero = true

_r private val has register of 32 bit.

val r has IF condition if it gets true ($_r >> 1 << 1$) condition execute else result of variable _r will be return to val r

Call scoreboard clear method

Define method id_sboard_clear_bypass

Code comprises of if condition in which it uses tile Parameter condition for checking hazard, execution of instruction is out of order by reshuffling instructions. It uses ll_wen (write enable and set them true Bool) and ll_waddr (write address) which are defined above in the code

```

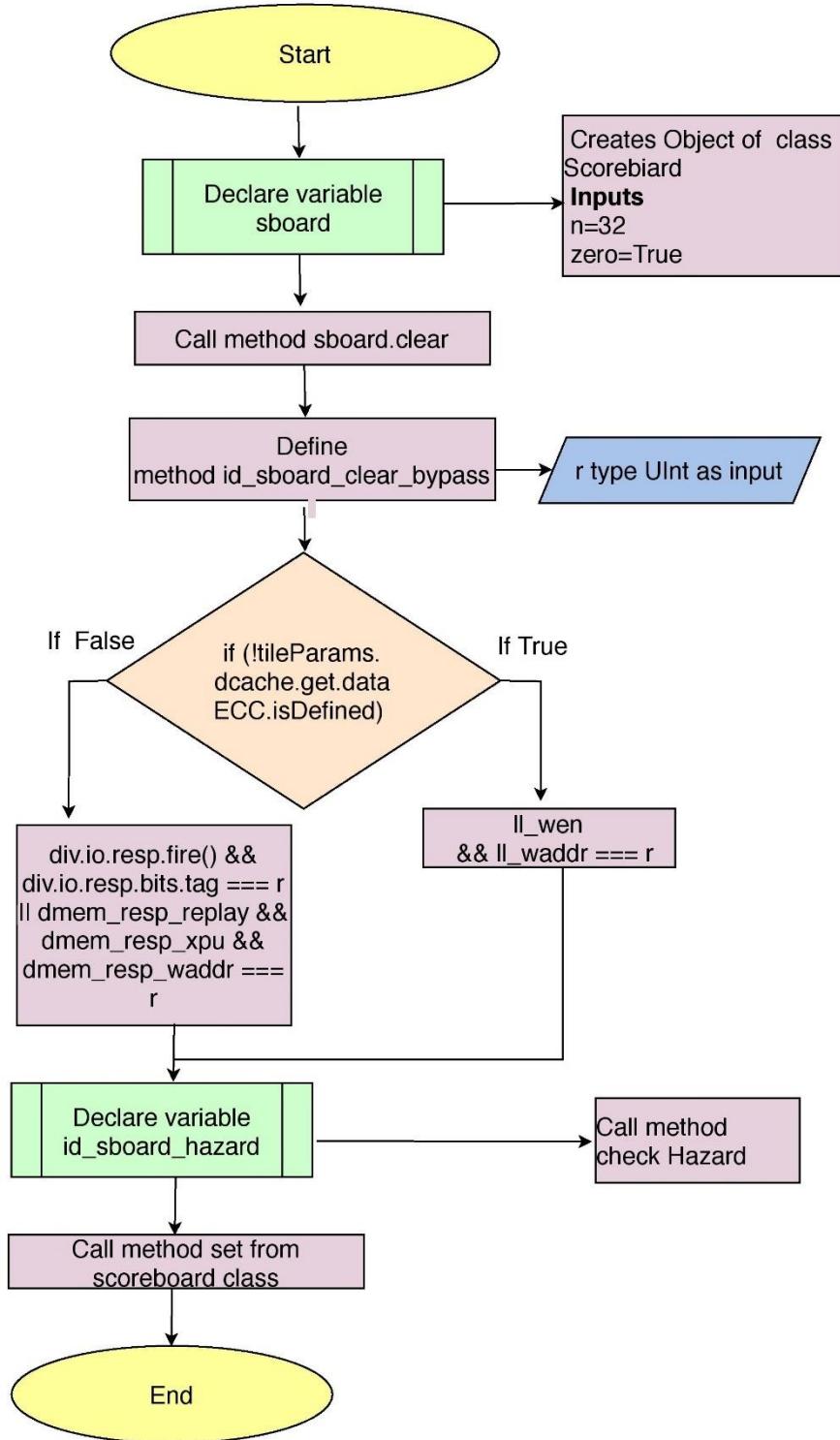
val ll_waddr = Wire(init = div.io.resp.bits.tag)
val ll_wen = Wire(init = div.io.resp.fire())

```

In else condition different variables from data memory and multiplier module uses in it which compares the result with r which is the input parameter of method id_sboard_clear_bypass

Define variable id_sboard_hazard which calls checkHazards method for checking any sort of hazard

Call scoreboard set method for setting scoreboard by taking wb_set_sboard && wb_wen, wb_waddr as input for enabling for enabling scoreboard



```

val id_stall_fpu = if (usingFPU) {
val fp_sboard = new Scoreboard(32)
fp_sboard.set((wb_dcache_miss && wb_ctrl.wfd || io.fpu.sboard_set) &&
wb_valid, wb_waddr)
fp_sboard.clear(dmem_resp_replay && dmem_resp_fpu, dmem_resp_waddr)
fp_sboard.clear(io.fpu.sboard_clr, io.fpu.sboard_clra)
checkHazards(fp_hazard_targets, fp_sboard.read _)
} else Bool(false)

```

explanation

Define variable id_stall_fpu in which it uses If condition which will execute if scoreboard using floating point unit (FPU)

Creating object from class Scoreboard of 32 bit in variable fp_sboard

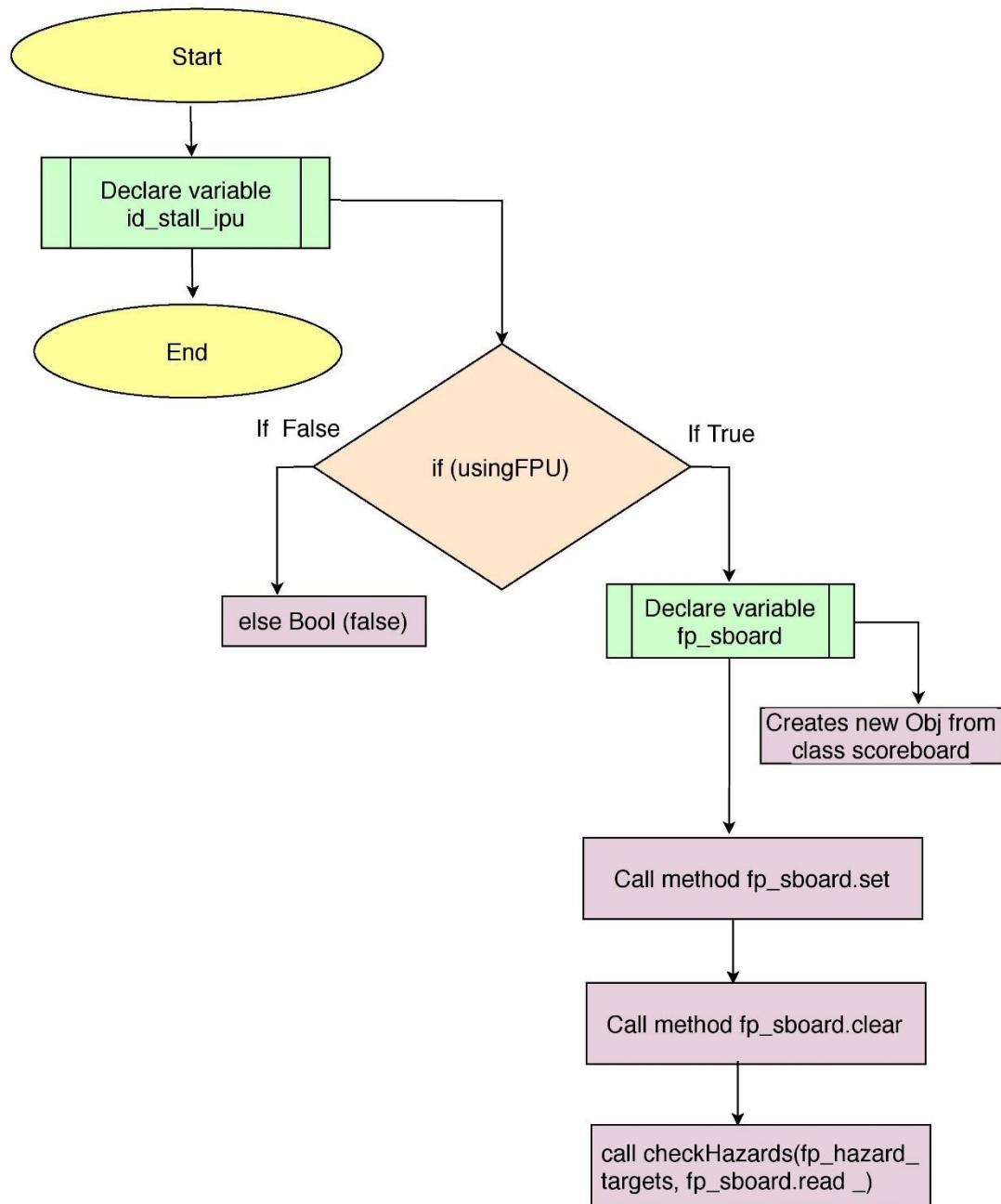
Three methods of scoreboard has been called in which it is setting scoreboard and clearing it after use it

In set method of scoreboard two parameters 1st is for enabling the set method and 2nd is for address which is define in main class of scoreboard

io.fpu.sboard_set and io.fpu.sboard_clr will decides set and clear of scoreboarding

CheckHazards method calls in which it uses fp_hazar_targets and fp_sboard.read_.

If condition gets fails Bool (false) will return to id_stall_fpu



APPENDIX

Scala & CHISEL keywords:

- `Wire()`: Constructs a Wire from type template.
- `Arbiter()`: Hardware module that is used to sequence n producers into 1 consumer. Priority is given to lower producers.
 - `def :=(that: Data)`: Connect this data to that data mono-directionally and element-wise.
 - `def <>(that: Data)`: Connect this data to that data bi-directionally and element-wise.
- `Reg(next: Any)`: Creates a register with optional next value.
- `RegEnable(next: Any, init: Any, enable:Bool)`: Returns a register with the specified next, update enable gate, and reset initialization.
- `Vec.fill(n:int)(gen:=> func)`: Creates a new Vec of length n composed of the result of the given function repeatedly applied.
- `assert(cond:Bool)`: A workaround for default-value overloading problems in Scala, just '`assert(cond, "")`'
- `require(requirement:Bool)`: Tests an expression, throwing an `IllegalArgumentException` if false. This method is similar to `assert`, but blames the caller of the method for violating the condition.
- `Some(value:A)`: represents existing values of type A.
- `log2Up(in:Int)`: Compute the log2 rounded up with min value of 1
- `UIInt(width: Width)`: Create a UIInt port with specified width.
- `orR`: a hardware Bool resulting from every bit of this UIInt or'd together
- `andR`: And reduction operator
- `Mux(cond:Bool, con: Any, alt: Any)`: Creates a mux, whose output is one of the inputs depending on the value of the condition.
- `Mux1H`: Builds a Mux tree out of the input signal vector using a one hot encoded select signal. Returns the output of the Mux tree.
- `UIIntToOH`: Returns the one hot encoding of the input UIInt.
- `OHToUIInt`: Returns the bit position of the sole high bit of the input bitvector.
- `Split(Arr, regex)`: splits the array by means of regex given
- `flip()`: Flips the data
- `Valid()`: Adds a Valid bit in the data
- `Bits(width= Any)`: Create a UIInt with a specified width
- `getOrElse()`: Returns the option's value if the option is nonempty, otherwise return the result of evaluating default.
- `.asInput`: makes the variable a input of the class
- `dontCare(width=Int)`: Creates a BitPat of all don't cares of the specified bitwidth.
- `Implicit`: A type for which there is always an implicit value.
- `withClock()`: Creates a new Clock scope
- `~`: Bitwise inversion operator

References

Scala: <https://www.scala-lang.org/api/current/index.html>

CHISEL: <https://www.chisel-lang.org/api/latest/index.html>

Rocket-Chip: <https://github.com/chipsalliance/rocket-chip>

MERL-UIT: <https://www.merledupk.org>