BOOM

An open-source out-of-order processor

Christopher Celio, Jerry Zhao, Abraham Gonzalez,
Ben Korpan, Krste Asanovic, David Patterson

Chisel Community Conference 2018

https://github.com/riscv-boom

Esperanto TECHNOLOGIES

RISC-V

ADEPT

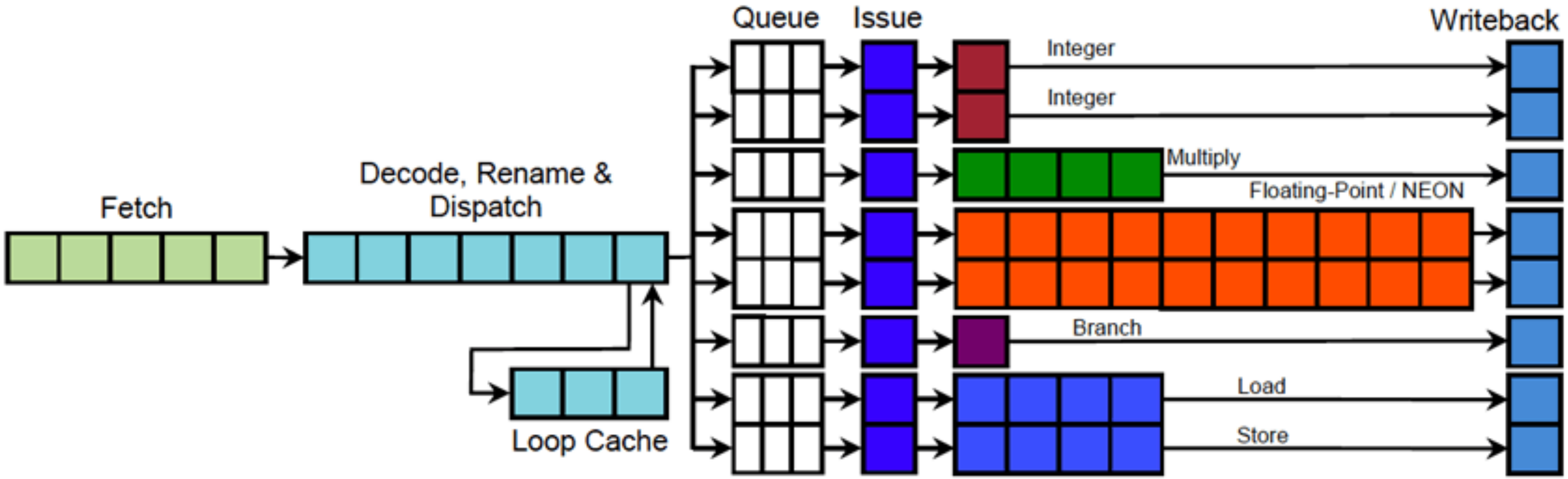
ASPIRE UC Berkeley

Berkeley Architecture Research

Outline

- Motivation
- Microarchitectural Overview
- Current state of the project
- How to get started
- How to contribute
- Things to work on
- Discussion

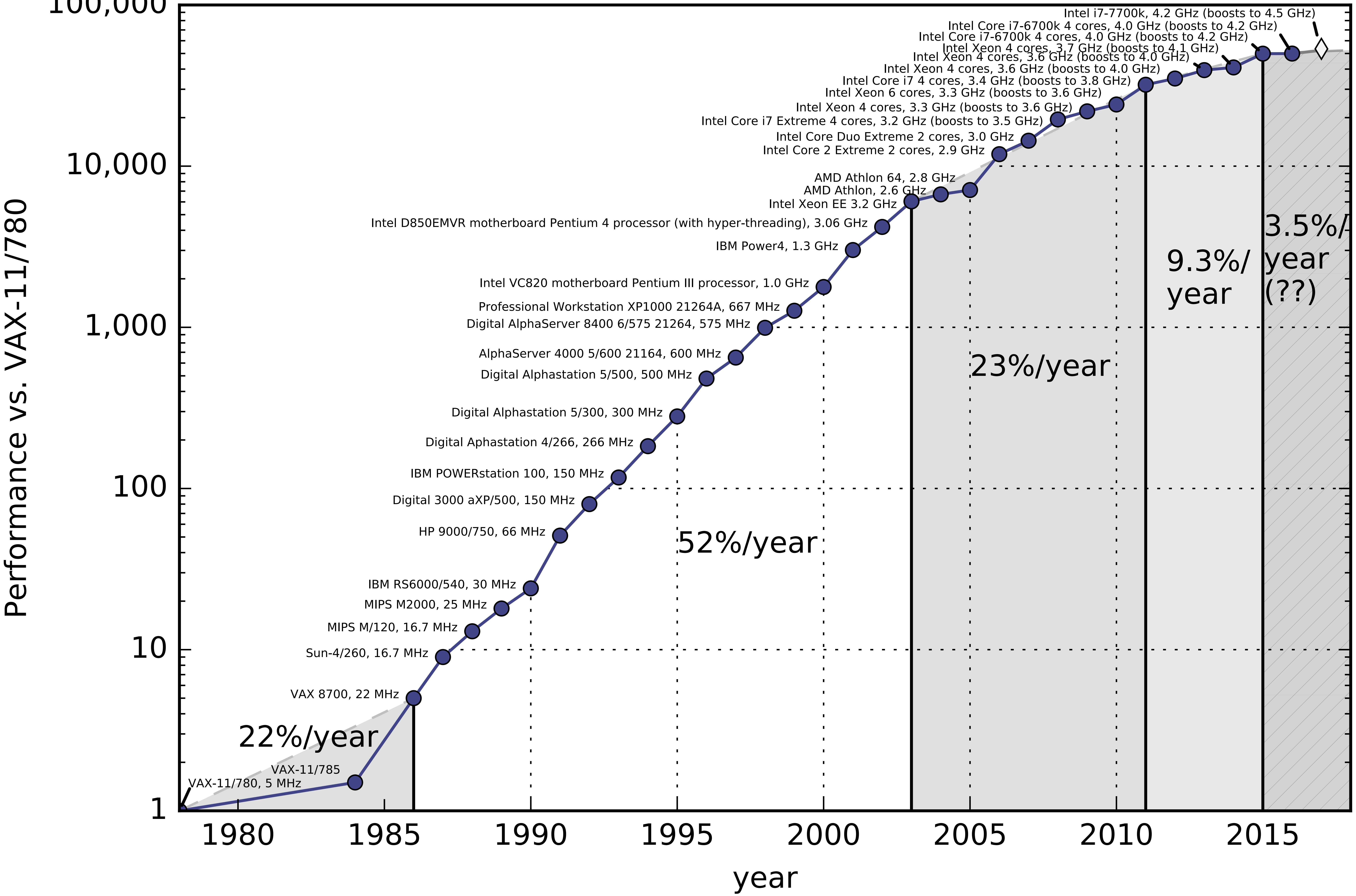
What does a modern processor look like?



ARM Cortex A-15

- pipelining
  - overlap execution of multiple instructions
- superscalar
  - multiple instructions / cycle / stage
- out-of-order
  - execute instructions in dependence order

40 years of Processor Performance



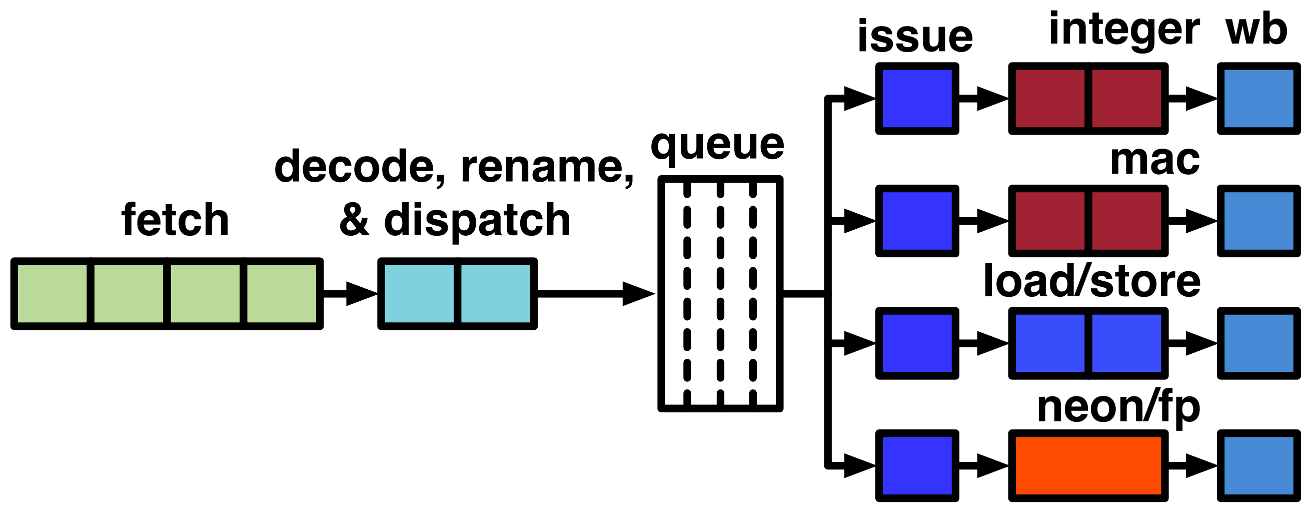
Data (mostly) from H&P CAAQA 6th Ed

Academic OoO Research

- Lack of effort in academia to build, evaluate OoO designs
  - NCSU's Fabscalar/AnyCore
  - MIT's riscy-OOO (bluespec)
- most research uses software simulators
  - SimpleScalar (5000 cites), GEM5 (1000 cites), SESC (250 cites)
  - generally don’t support full systems
  - cannot produce area, power numbers
  - hard to trust, verify results
    - McPAT is calibrated against 90nm Niagara, 65nm Niagara 2, 65nm Xeon, and 180nm Alpha 21364
  - very slow (~100 KHz)
    - ~10 hours of sim is 1 second of target

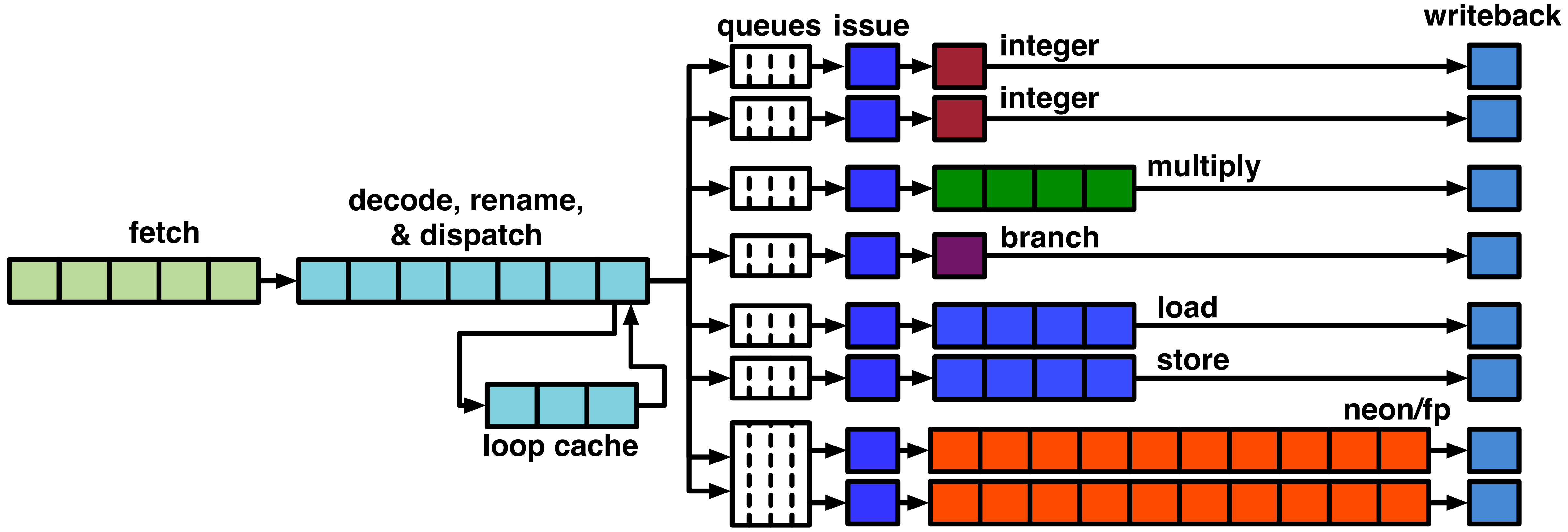
ARM's Cortex-A9, A15, A73

2007



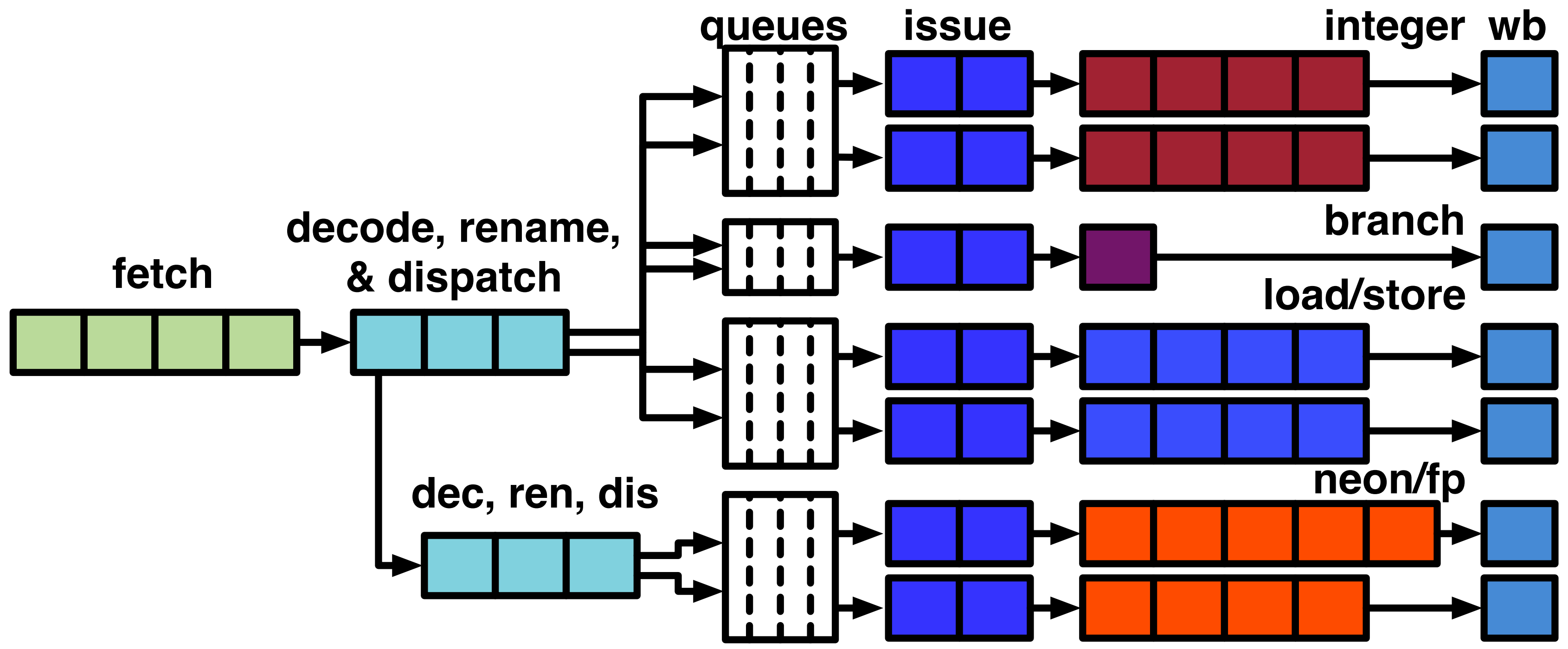
Cortex-A9

2012



Cortex-A15

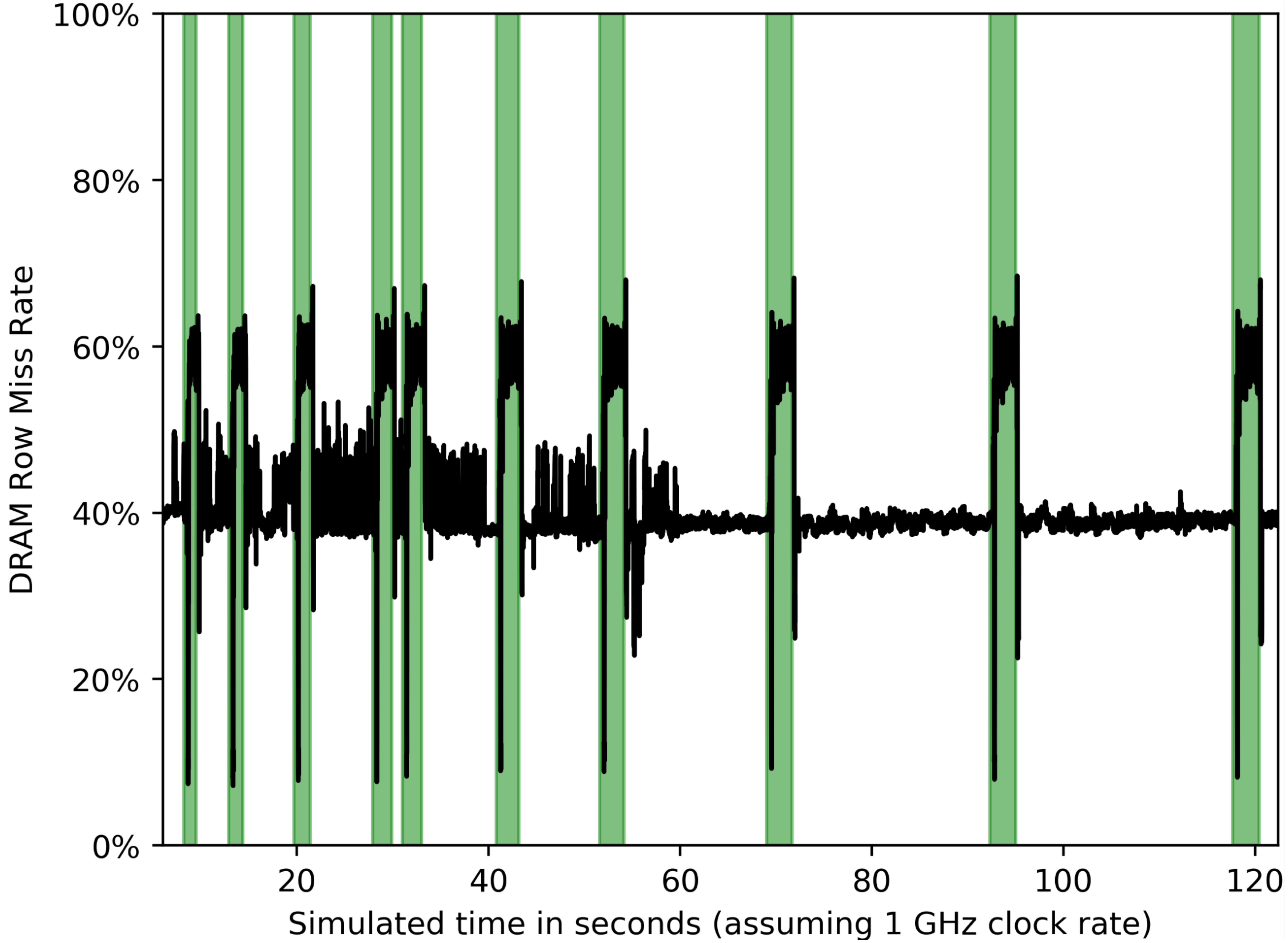
2016



Cortex-A73

- RTL of processor system
- FPGA-based simulators (50-100 MHz)
- **trillions** of instructions simulated (full workloads)
- power models built from actual activity
- can generate floor-plans, area, timing reports

2 minutes of Java Workload on Linux on RTL

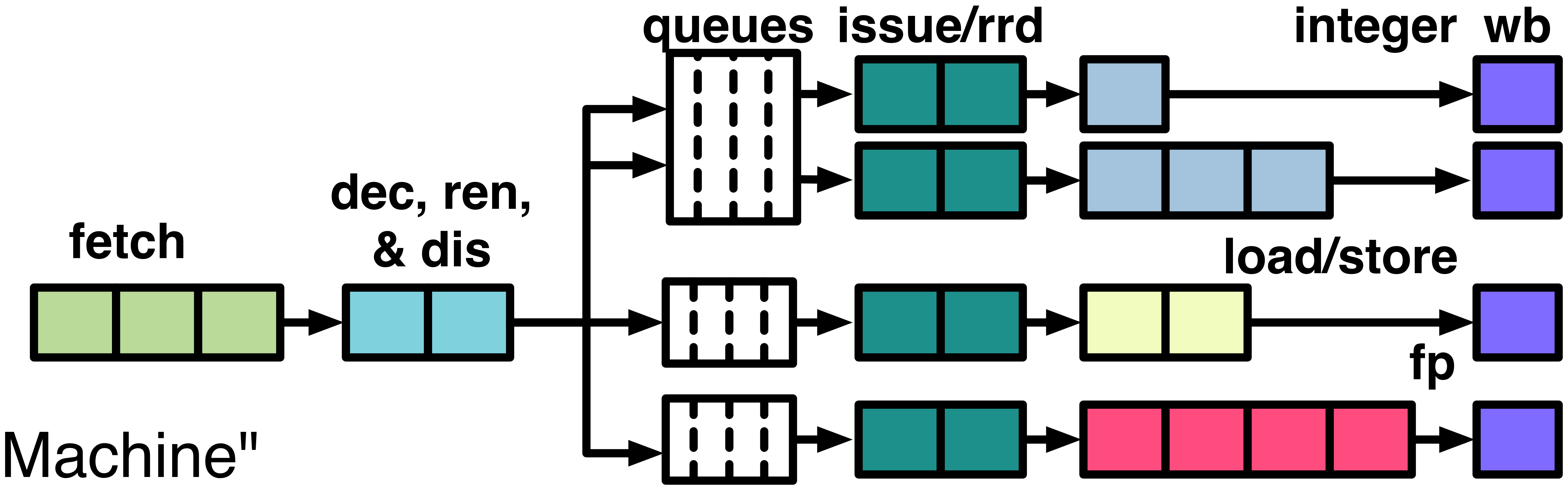


data collected by
Martin Maas, et. al.

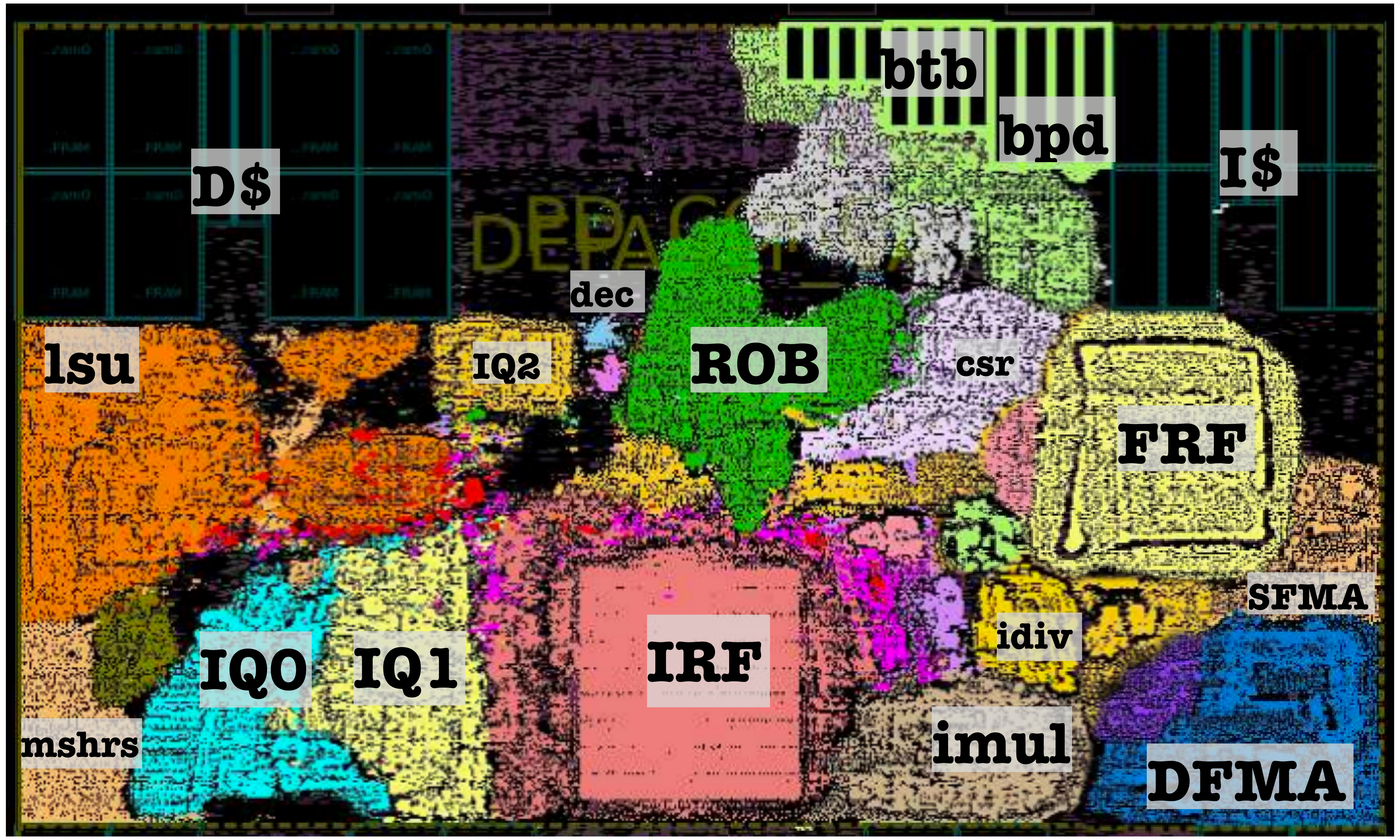
What is BOOM useful for?

- As a complex IP for methodology studies
  - How do you model RTL performance?
  - How do you measure RTL power?
  - How can we help build an agile verification story?
  - How do we do open-source cad flows?
- Software Studies
  - hardware/software co-design
  - high visibility of very long-running applications
- Off-the-shelf out-of-order core
  - need a core for your research tape-out to talk to your IP?
  - drive memory-mapped accelerators
- Realistic Microarchitecture Studies of contained blocks
  - branch prediction, issue queue design, etc.
- Security Research

What is BOOM?

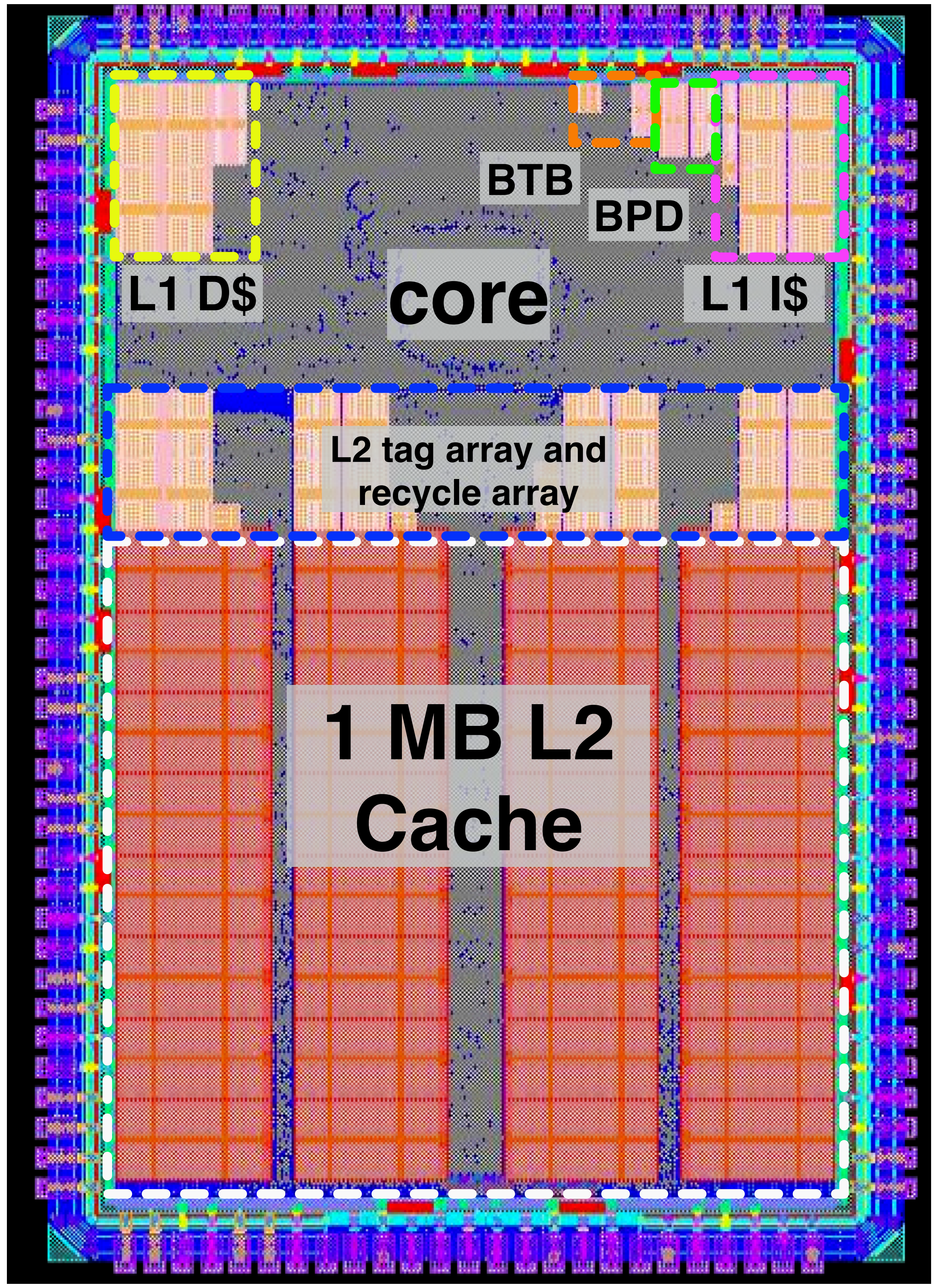


- "Berkeley Out-of-Order Machine"
- superscalar
- out-of-order
- implements **RV64G**, boots Linux
- It is synthesizable
- it is open-source
- written in **Chisel** (16k loc)
- It is parameterizable generator
- built on top of Rocket-chip SoC Ecosystem

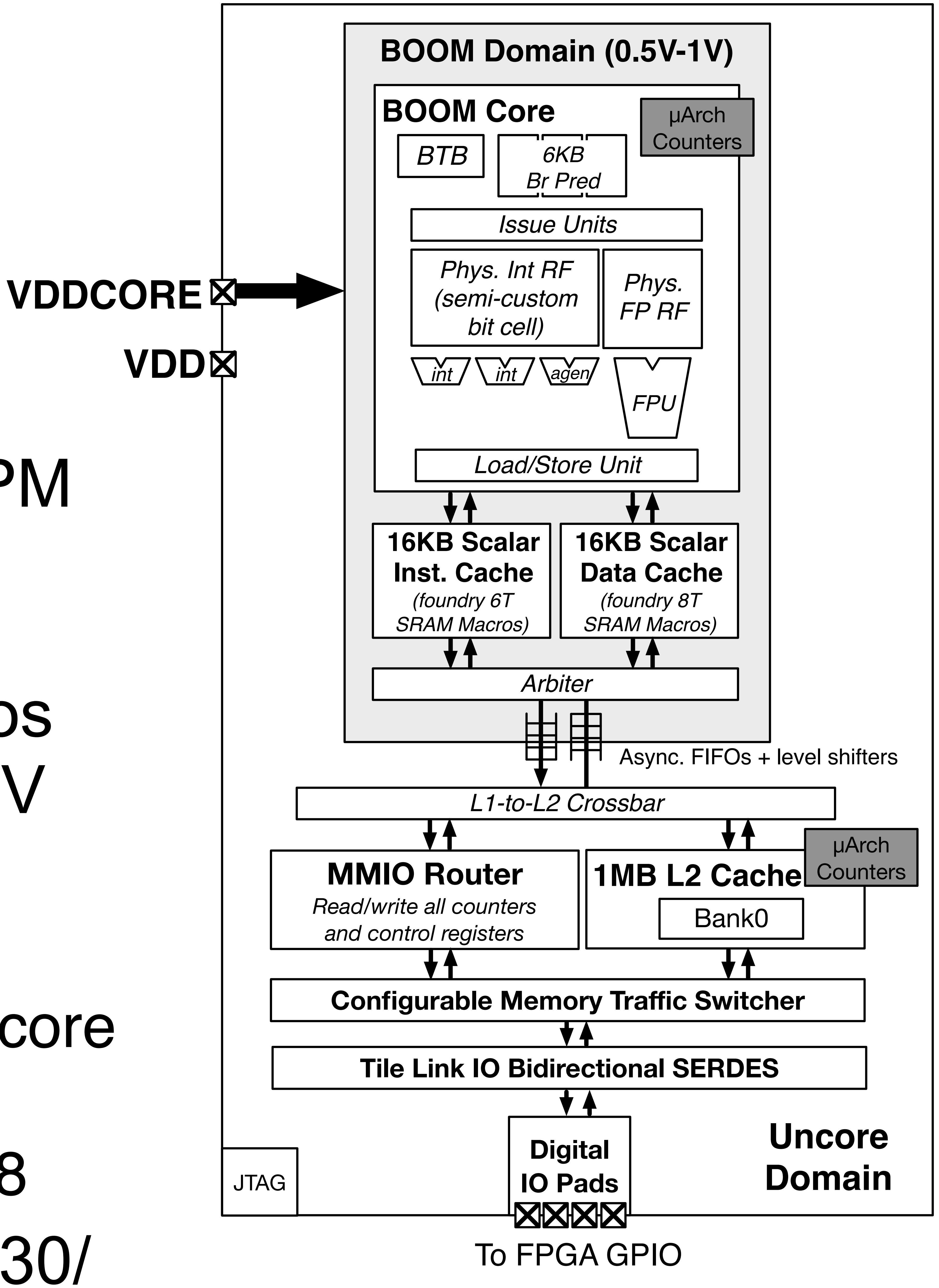


http://ucb-bar.github.io/riscv-boom

BROOM Chip (Taped out Aug 2017)



TSMC 28 nm HPM
6 mm²
417k std cells
73 SRAM macros
1.0 GHz @ 0.9 V

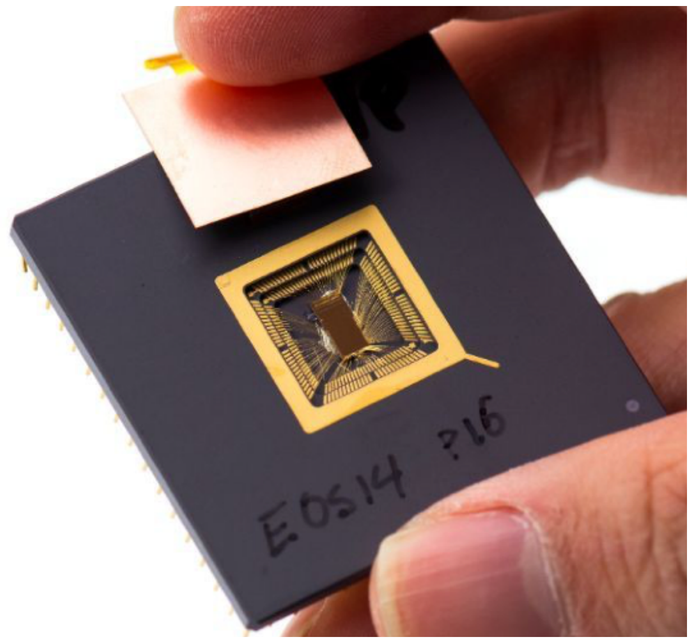

BOOM Domain (0.5V-1V)
BOOM Core
µArch Counters
BTB
6KB Br Pred
Issue Units
Phys. Int RF (semi-custom bit cell)
Phys. FP RF
int
int
agen
FPU
Load/Store Unit
VDDCORE
VDD
16KB Scalar Inst. Cache (foundry 6T SRAM Macros)
16KB Scalar Data Cache (foundry 8T SRAM Macros)
Arbiter
Async. FIFOs + level shifters
L1-to-L2 Crossbar
MMIO Router
Read/write all counters and control registers
1MB L2 Cache
µArch Counters
Bank0
Configurable Memory Traffic Switcher
Tile Link IO Bidirectional SERDES
JTAG
Digital IO Pads
Uncore Domain
To FPGA GPIO

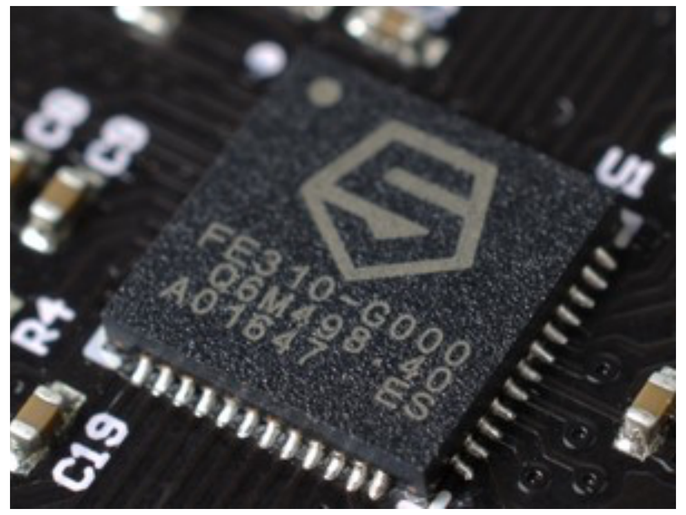
- Open-source superscalar out-of-order RISC-V core
- Resilient cache for low-voltage operation
- See BROOM work presented in Hot Chips 2018

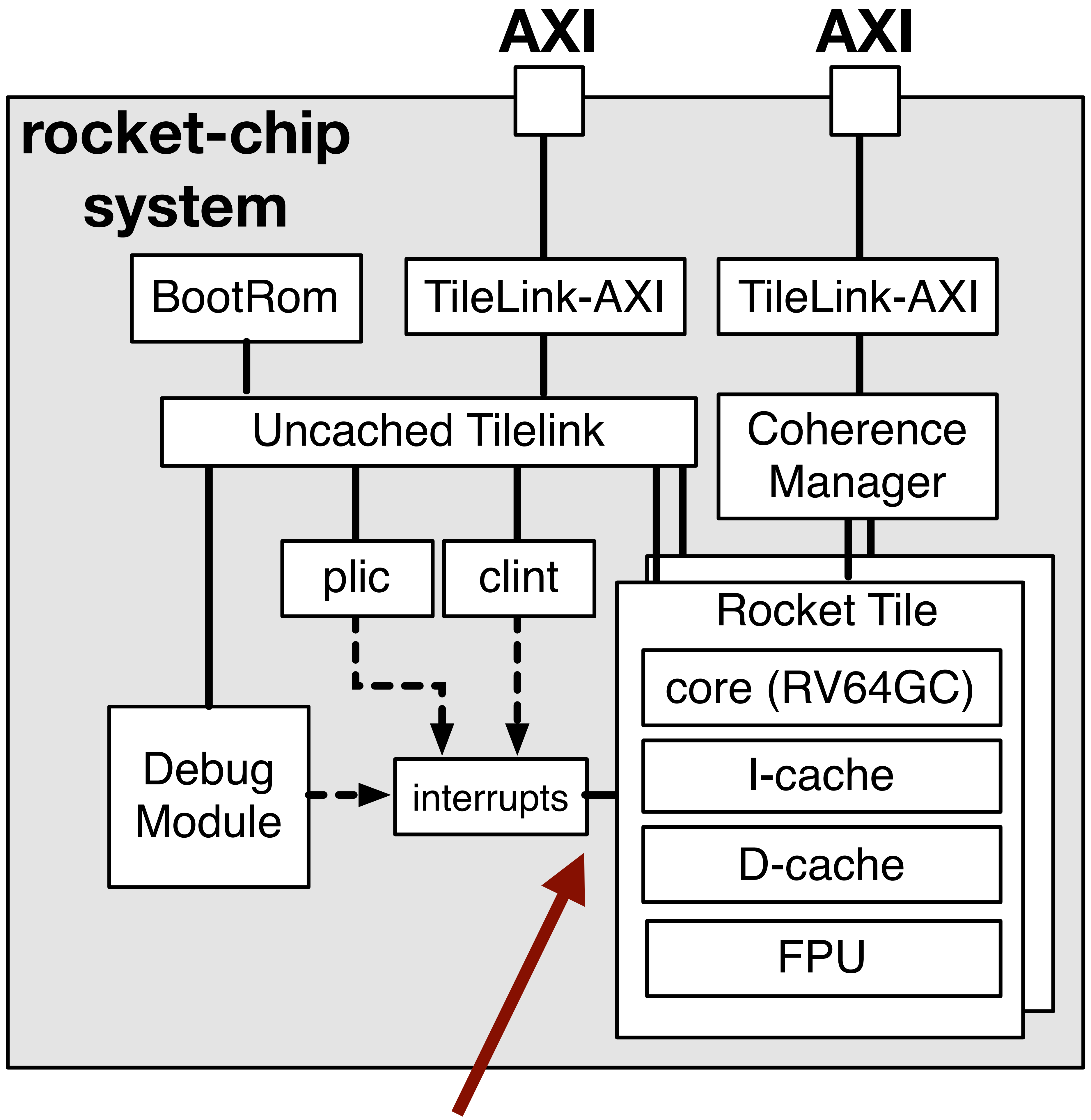
https://www.hotchips.org/archives/2010s/hc30/

Leveraging Open-source RTL

- The Rocket-chip SoC Generator
- Started in 2011
- Taped out ~~10~~ (~~13~~?) *17* times by Berkeley + many others
- 6,257 commits
- 72 contributors
- Commercial quality
- Replace standard in-order core with BOOM
- Leverage Rocket-chip as a library of processor components



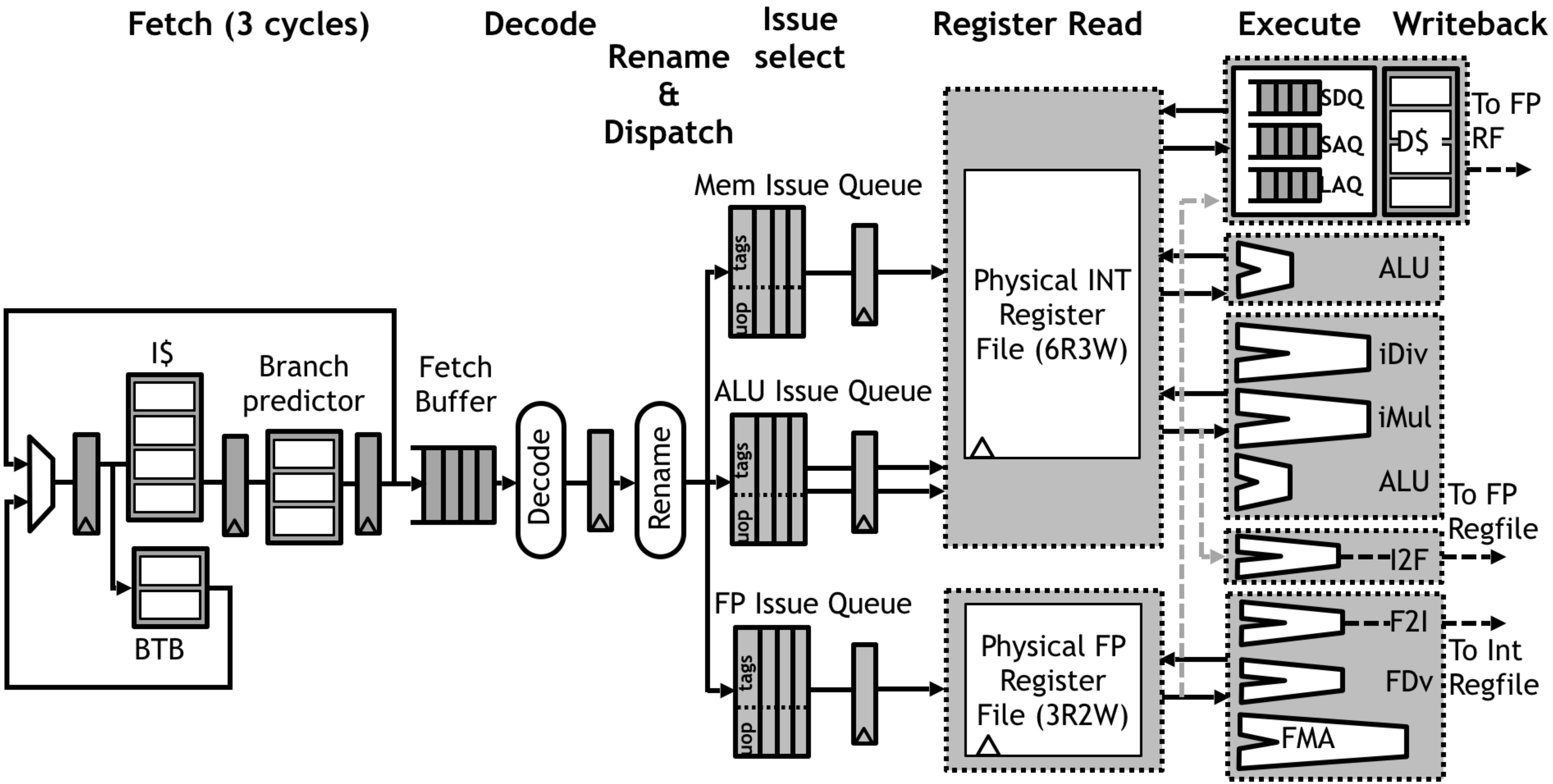




BOOM goes here

https://github.com/freechipsproject/rocket-chip

The BOOMv2 Core



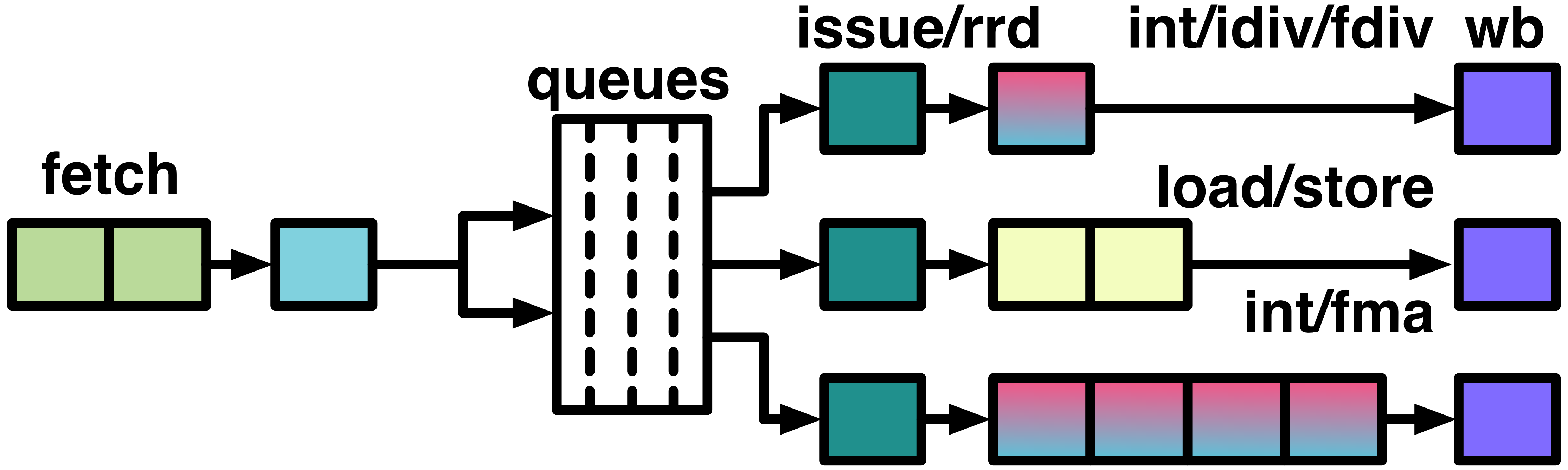
Core Comparison

Processor	SiFive U54 Rocket (RV64GC)	Berkeley BOOMv2 (RV64G)	OpenSPARC T2	ARM Cortex-A9	Intel Xeon Ivy
Language	Chisel	Chisel	Verilog	-	SystemVerilog
Core LoC	8,000	16,000	290,000	-	-
SoC LoC	34,000	50,000	1,300,000	-	-
Foundry	TSMC	TSMC	TI	TSMC	Intel
Technology	28 nm (HPC)	28 nm (HPM)	65 nm	40 nm (G)	22 nm
Core+L1 Area	0.54 mm^2	0.52 mm^2 16kB/16kB	~12 mm^2	~2.5 mm^2	~12 mm^2 core+L1+L2
Coremark/MHz	2.75	3.77	1.64*	3.71	5.60
Frequency	1.5 GHz	1.0 GHz	1.17 GHz	1.4 GHz	3.3 GHz

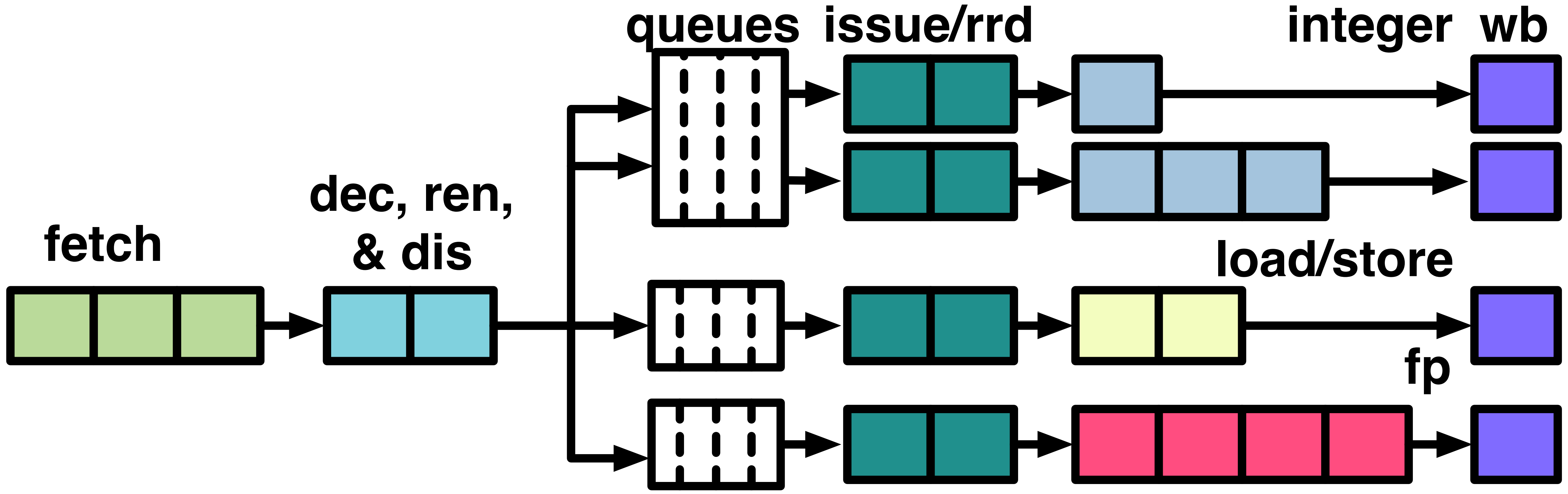
*From eembc.org. 32 threads/8 cores achieve 13 Cm/MHz.

The Evolution of BOOM

	BOOMv1	BOOMv2
BTB entries	40 (fully-assoc)	64 x 4 (set-assoc)
Fetch Width	2 insts	2 insts
Issue Width	3 micro-ops	4 micro-ops
Issue Entries	20	16/20/10
Regfile	7r3w (unified)	6r3w (inst), 3r2w (fp)
Exe Units	iALU+iMul+FMA iALU+fDiv Load/Store	iALU+iMul+iDiv iALU FMA+fDiv Load/Store



BOOM v1 (April 2017)

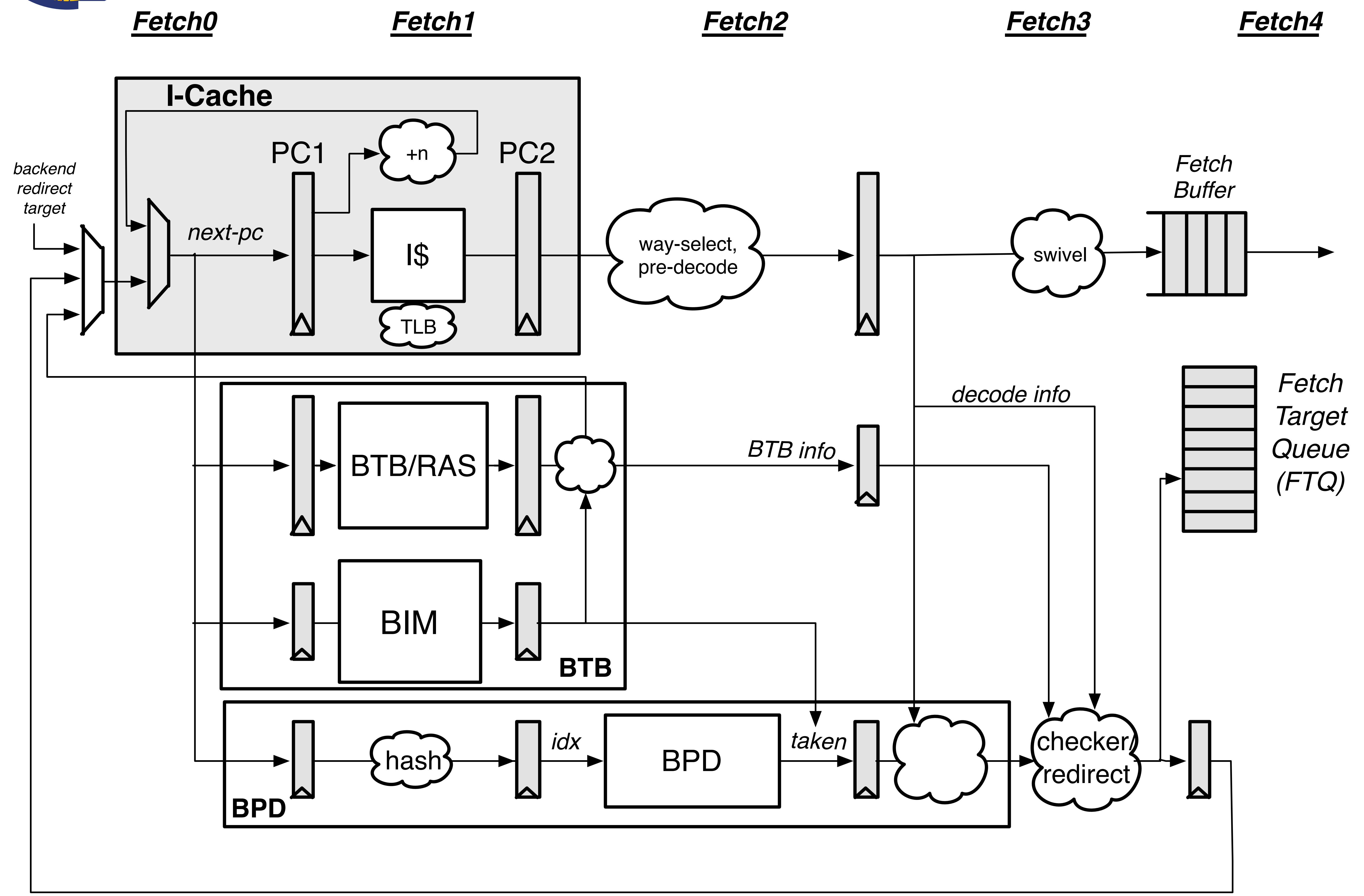


BOOM v2 (Aug 2017)

BOOMv3 (tentative)

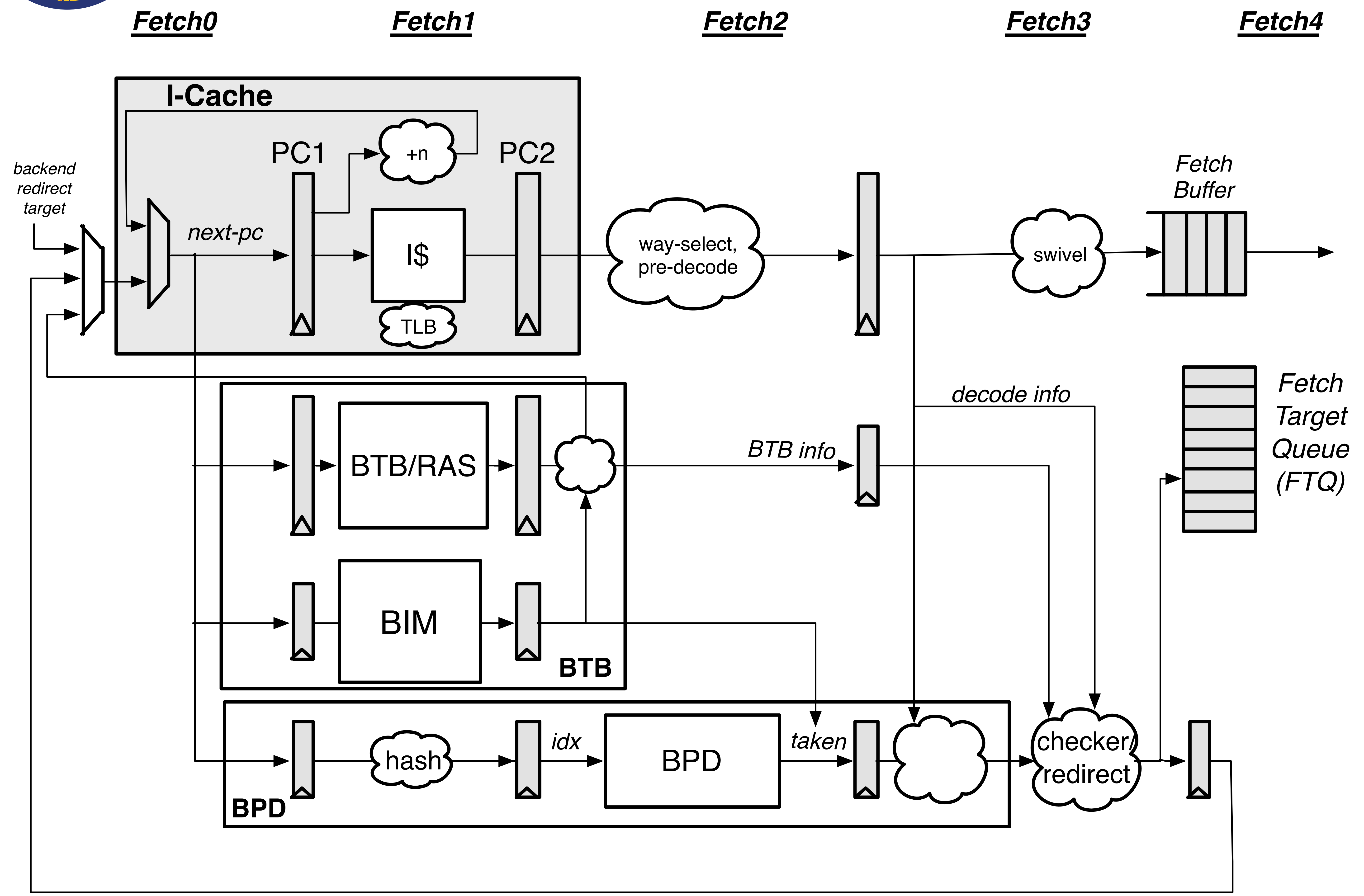
- Goal
  - Use lessons learned from BOOMv2 tape-out to improve core.
  - Update to the latest RISC-V standards.
- Privileged Spec v1.11, User 2.3
  - Done.
- RISC-V Compressed support
  - TODO.
- RISC-V WMO memory consistency model
  - Must order loads to the same address.
  - TODO.
- updated front-end and branch prediction
  - Performance debugging needed.
- 4-cycle load-use
  - speculates load-hit to save two cycles.
  - Done.

Current frontend



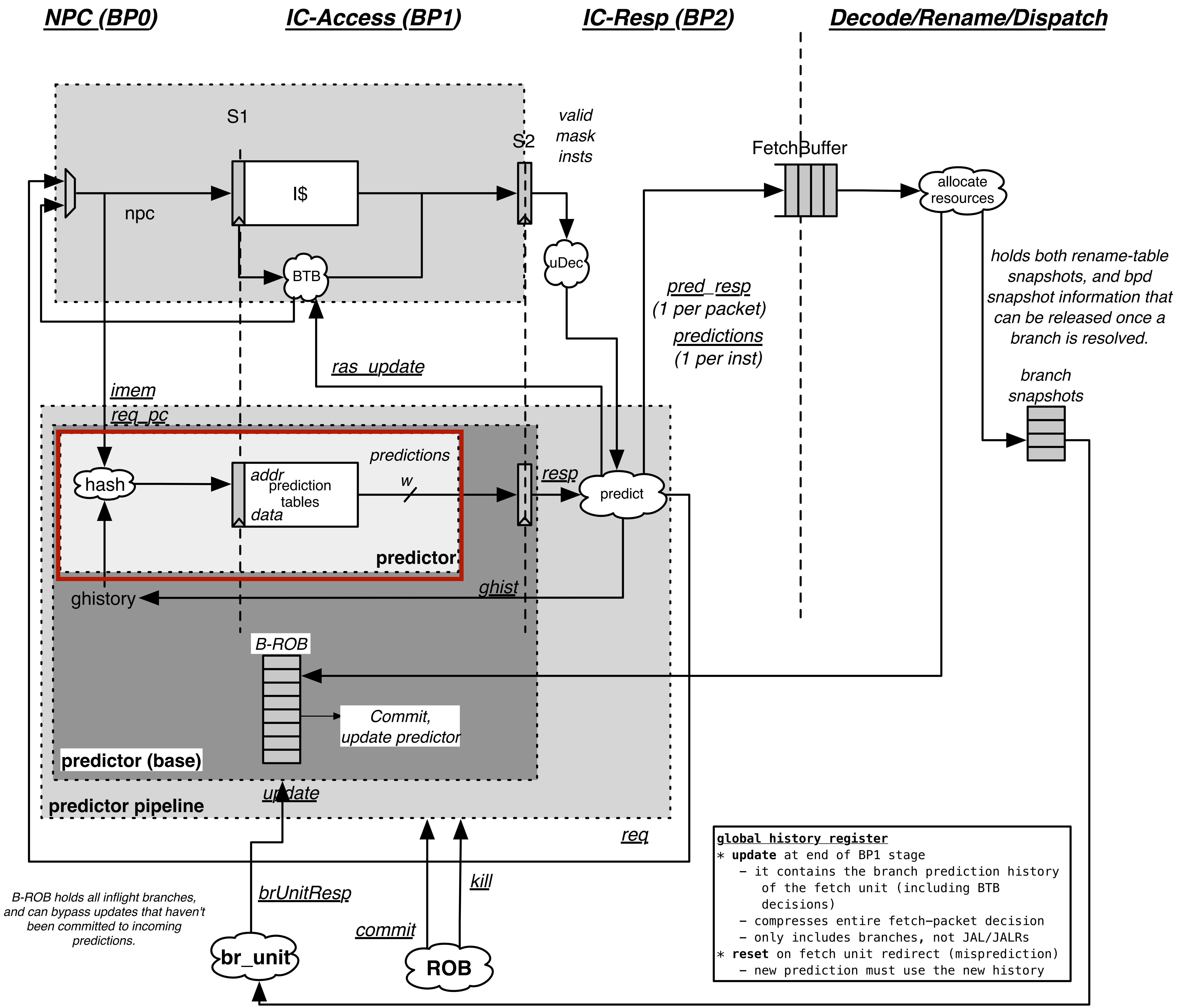
- BTB (branch target buffer)
  - predicts without seeing instructions
  - set-associative, partially tagged
  - checker to verify integrity
- BIM (bimodal)
  - a table of two-bit counters
  - used by BTB and optionally BPD to decide direction
- RAS (return address stack)
  - predicts returns
  - driven by BTB to make decisions

Current frontend



- BPD (conditional predictor)
  - provide your own (e.g., gshare or TAGE)
  - decides taken/not-taken based on instruction bits
  - uses path history
- Fetch Target Queue
  - stores fetch PC, branch prediction information
  - one entry == one fetch bundle

Abstract Branch Predictors

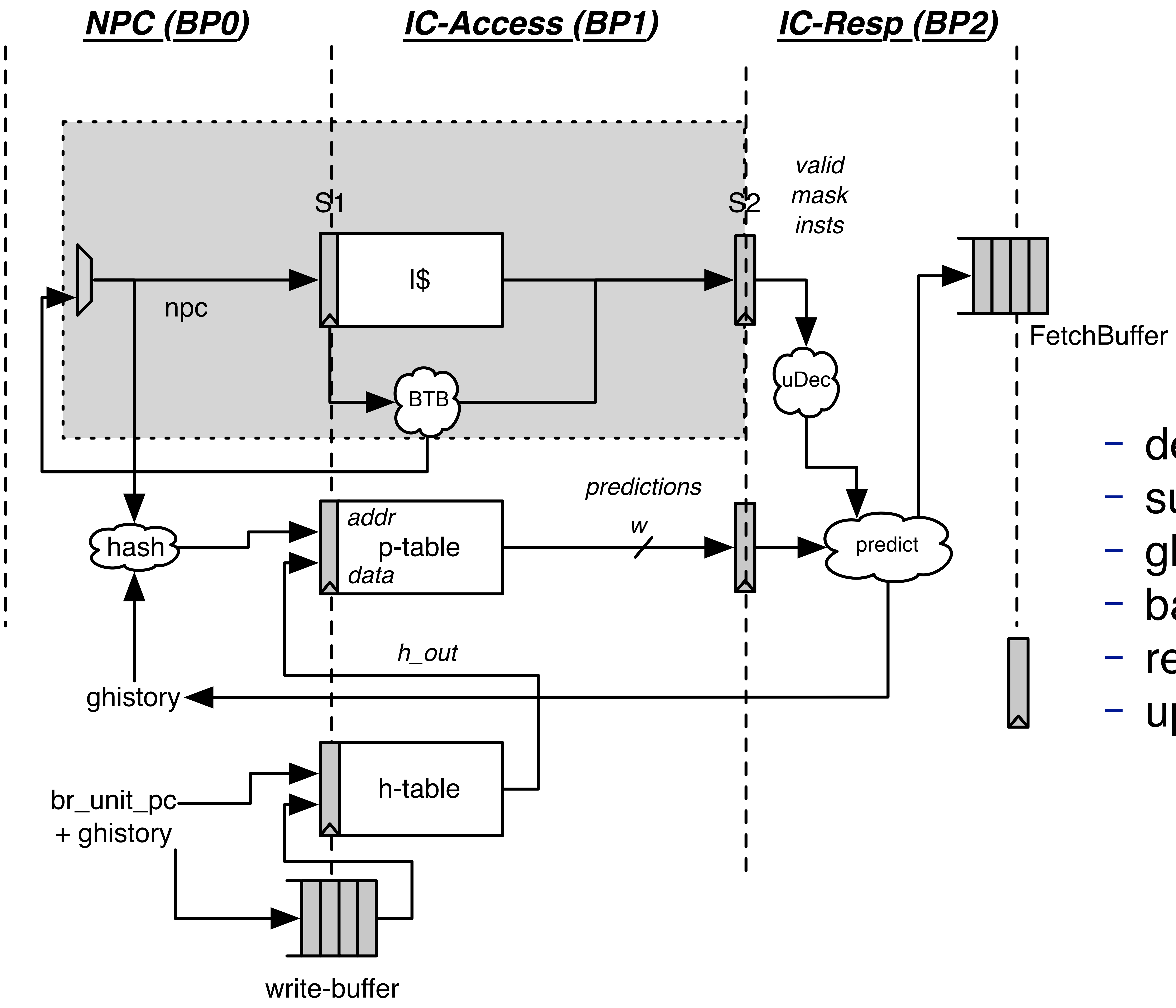


holds both rename-table snapshots, and bpd snapshot information that can be released once a branch is resolved.

B-ROB holds all inflight branches, and can bypass updates that haven't been committed to incoming predictions.

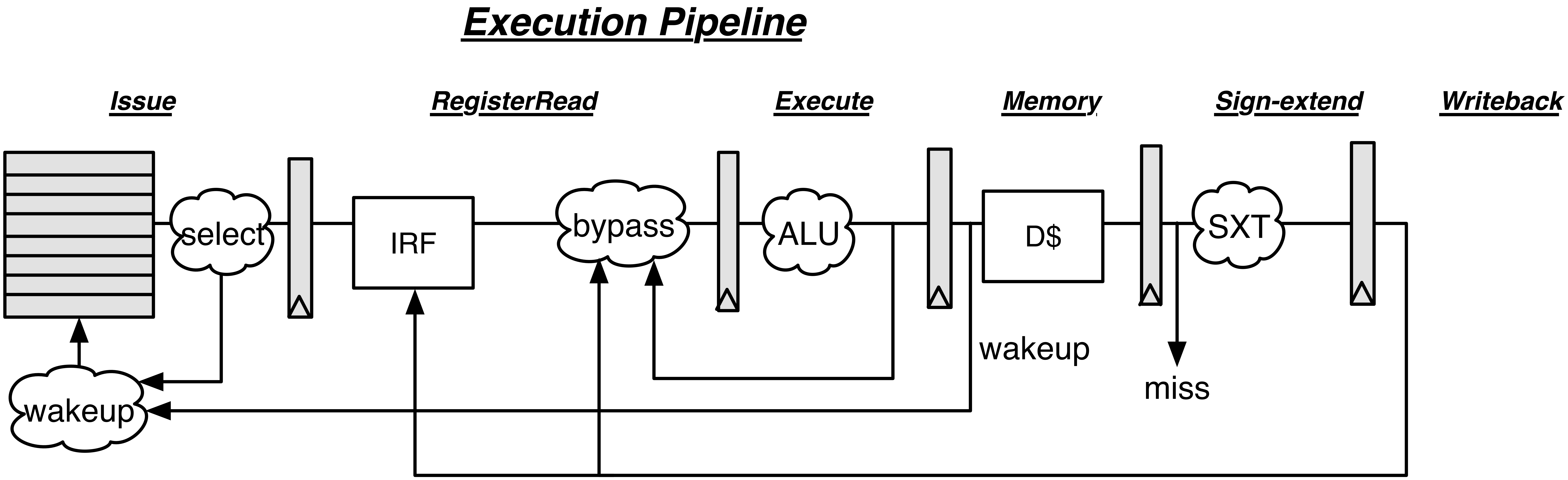
```
global history register
* update at end of BP1 stage
   - it contains the branch prediction history
     of the fetch unit (including BTB
     decisions)
   - compresses entire fetch-packet decision
   - only includes branches, not JAL/JALRs
* reset on fetch unit redirect (misprediction)
   - new prediction must use the new history
```

GShare in single-ported SRAM



- delayed ghistory update
- super-scalar predictions
- ghistory is fetch packet granularity
- banked p-table
- reset ghistory on misspeculations
- update during commit

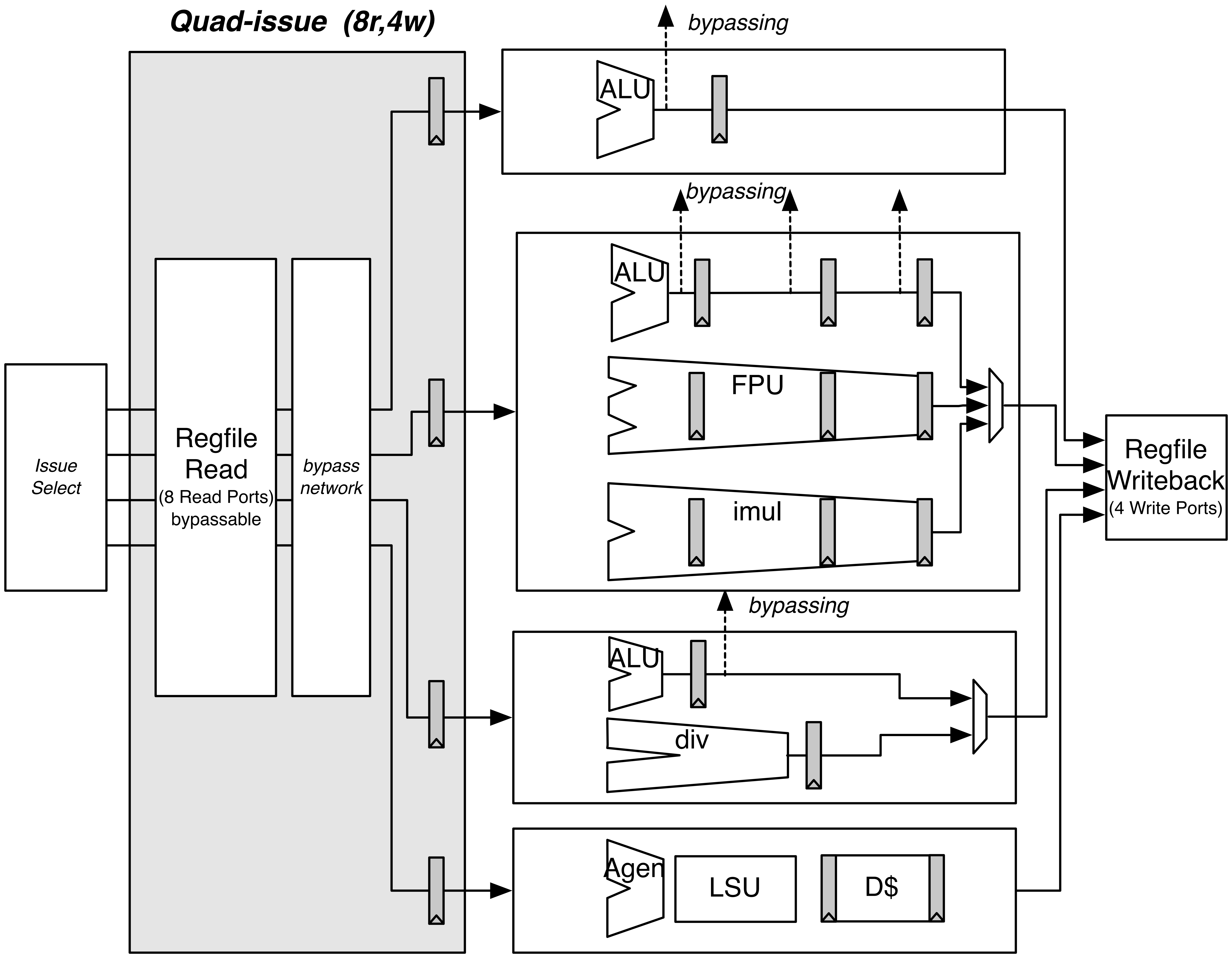
4-cycle load-use



- Speculatively wakeup uops that depend on loads
- Kill them on the next cycle if miss occurs

Execution Pipeline Generator

- FPU needs 3 sources
- Only support one Mem unit (one load/store)
- ALU is padded out to max latency
- div unit is unpipelined, can apply back-pressure

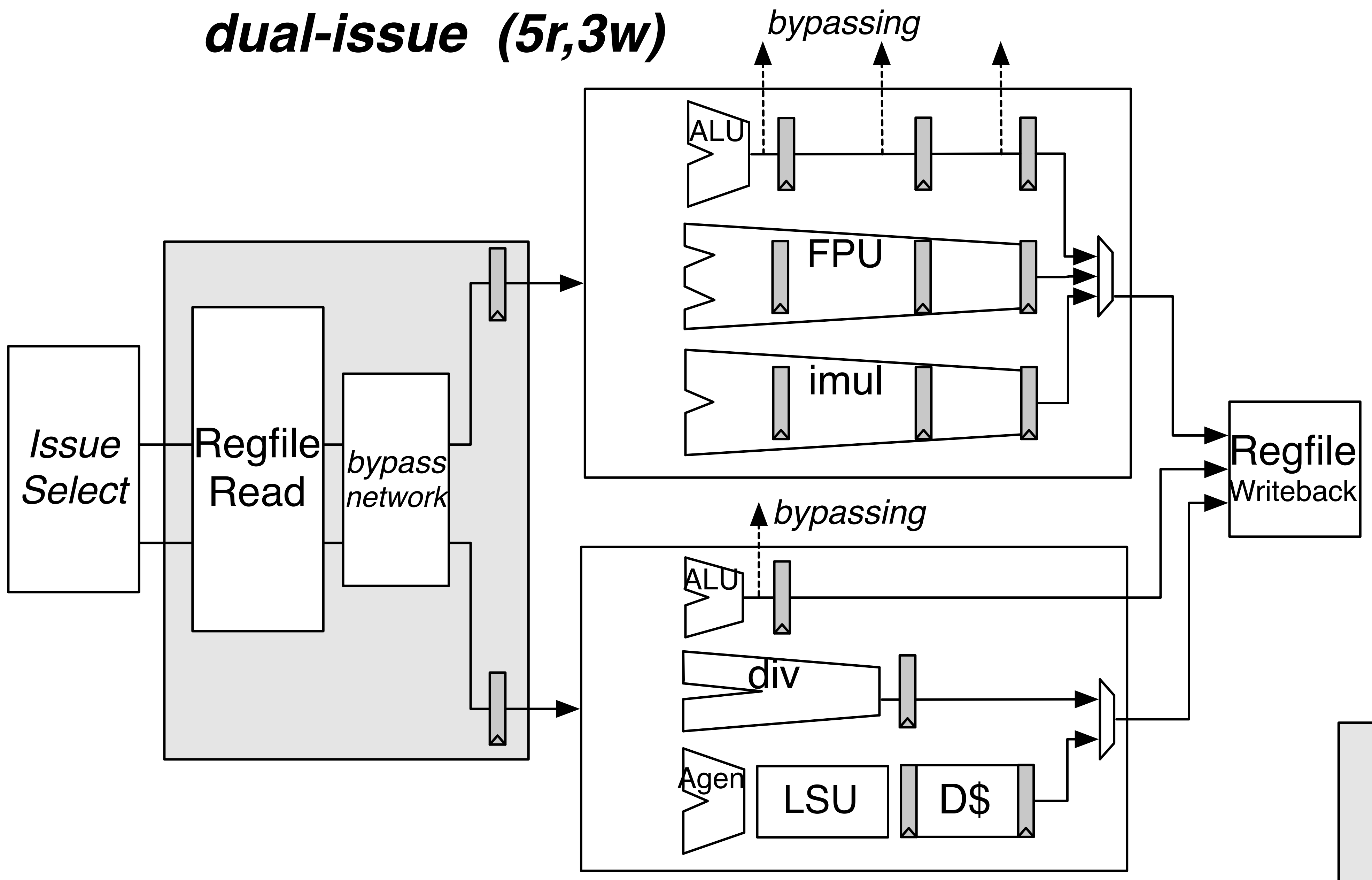


Branches

- MIPS R10K style
- Every branch:
  - is given a tag
  - takes a snapshot of the rename map tables
  - is given an empty "allocation list"
- Following instructions:
  - all uops have a branch-mask
  - if bit is set, they depend on that unresolved branch
- When branch is resolved:
  - the branch tag is broadcast across the machine
  - everyone clears their bit
  - allows new allocation
- When branch is mispredicted:
  - all uops with matching branch tag are killed immediately
  - rename tables are set to the snapshot
  - allocated physical registers are added back to free list

Parameterized Superscalar

dual-issue (5r,3w)

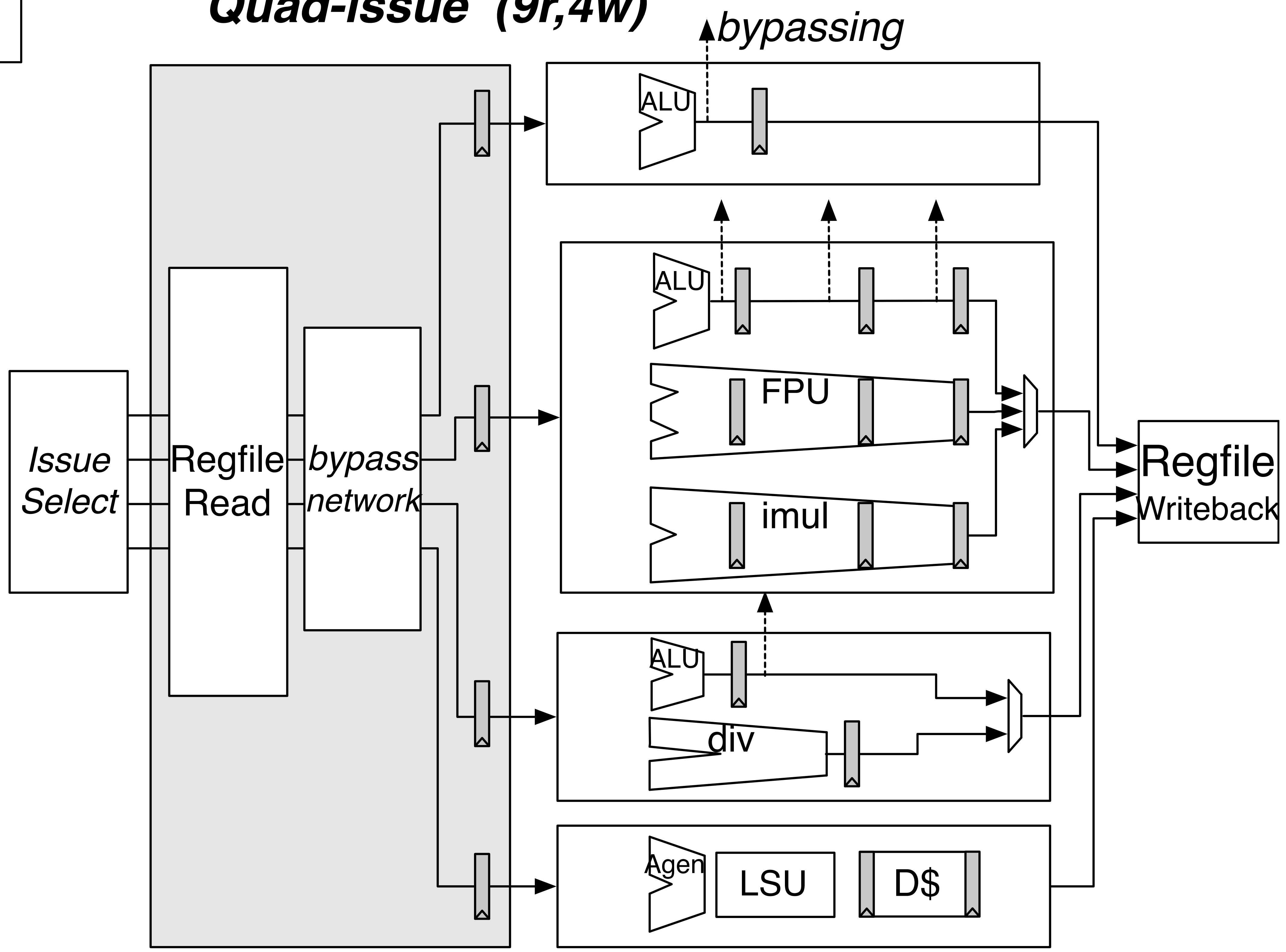


```
val exe_units = ArrayBuffer[ExecutionUnit]()
exe_units += Module(new ALUExeUnit(is_branch_unit    = true
                                  , has_fpu         = true
                                  , has_mul         = true
                                  ))
exe_units += Module(new ALUMemExeUnit(fp_mem_support = true
                                  , has_div         = true
                                  ))
```

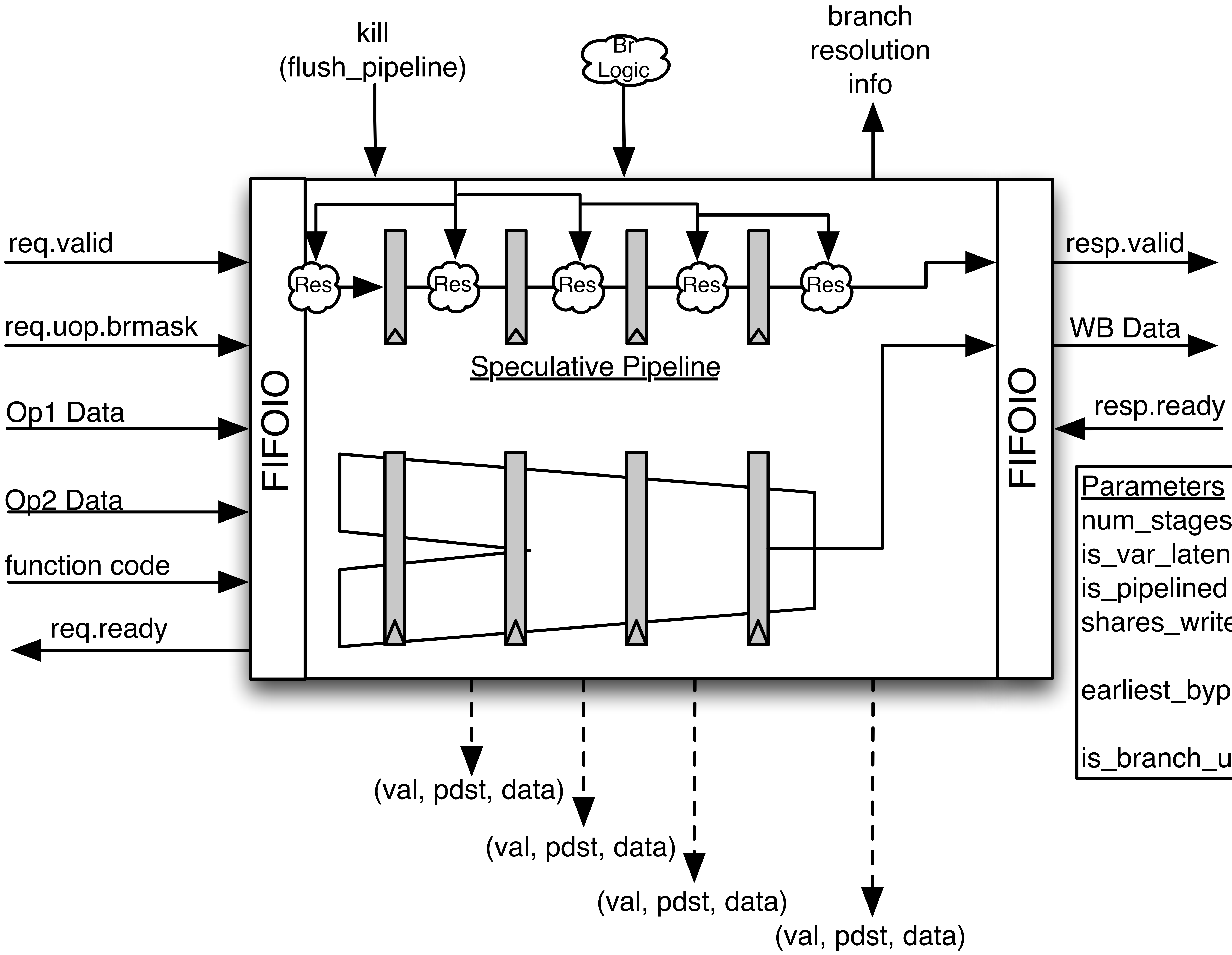
OR

```
exe_units += Module(new ALUExeUnit(is_branch_unit = true))
exe_units += Module(new ALUExeUnit(has_fpu = true
                                  , has_mul = true
                                  ))
exe_units += Module(new ALUExeUnit(has_div = true))
exe_units += Module(new MemExeUnit())
```

Quad-issue (9r,4w)



A Functional Unit



Parameters
num_stages = 4
is_var_latency = false
is_pipelined = true
shares_writeport= false

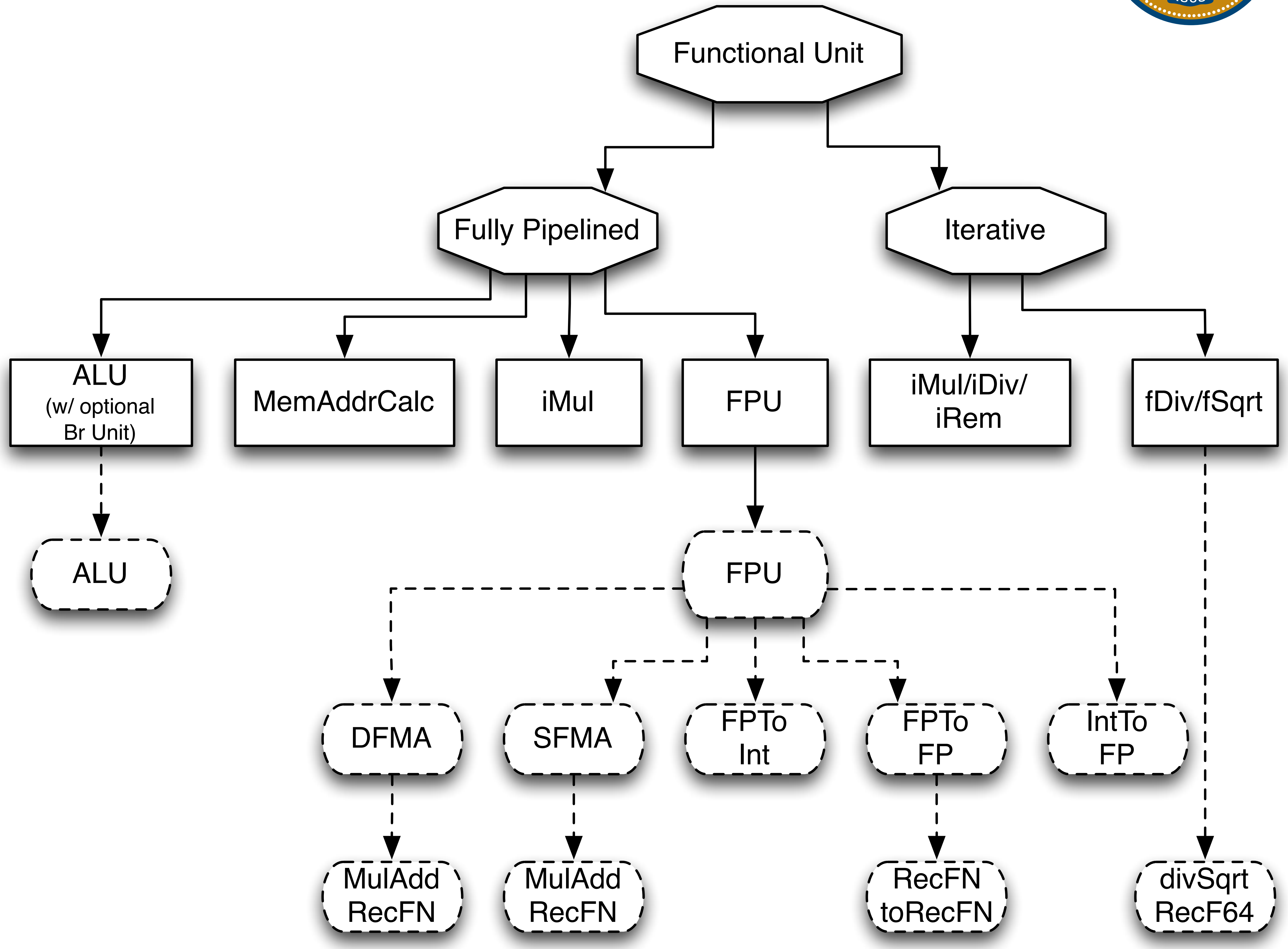
earliest_bypass_stage=4

is_branch_unit = true

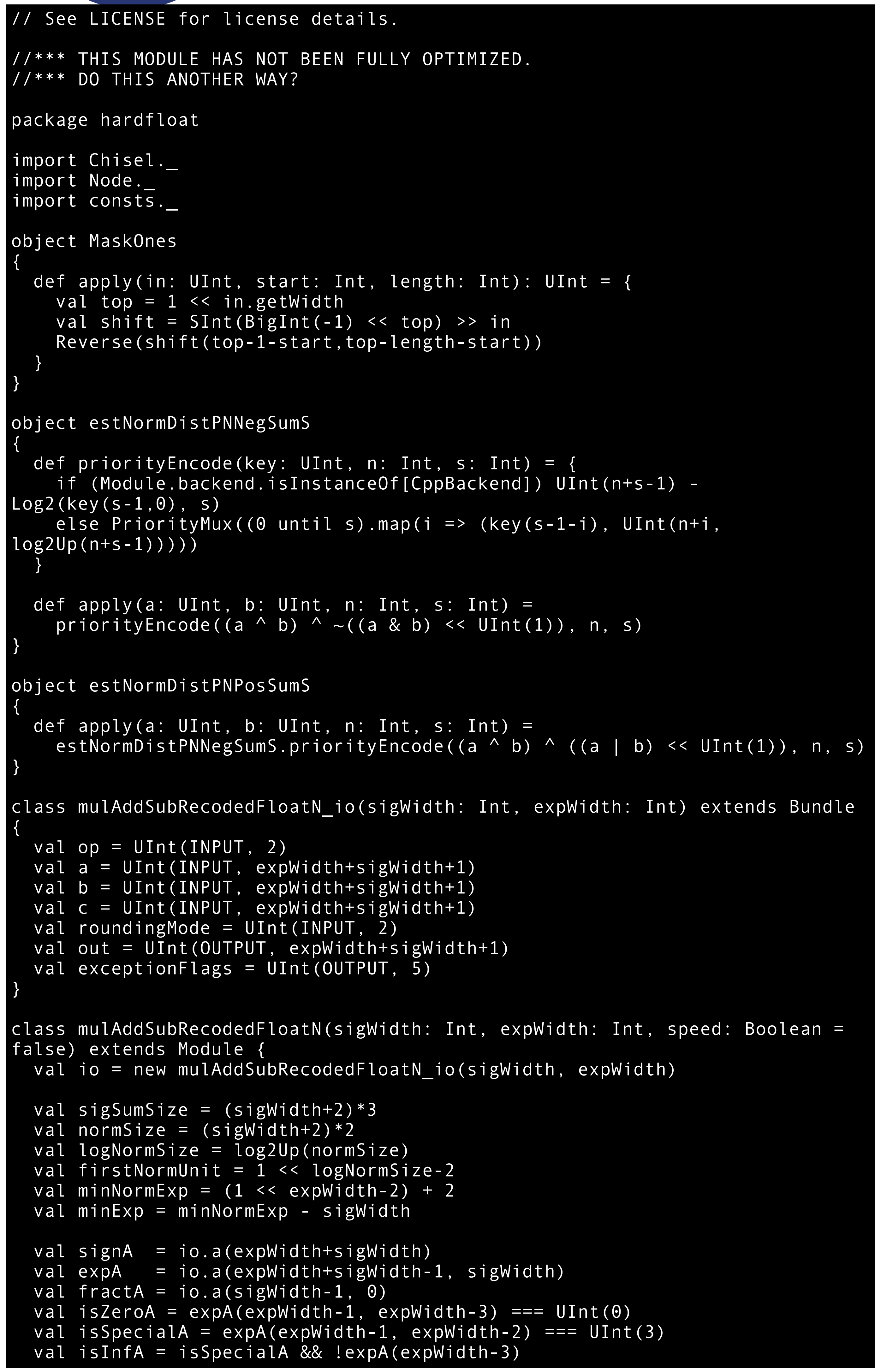
BYPASSES

A Functional Unit Hierarchy

- **Abstract FunctionalUnit**
  - describes common IO
- **Pipelined/Iterative**
  - handles storing uop metadata, branch resolution, branch kills
- **Concrete Subclasses**
  - instantiates the actual expert-written FU
  - no modifications required to get FU working with speculative OoO
  - allows easy “stealing” of external code



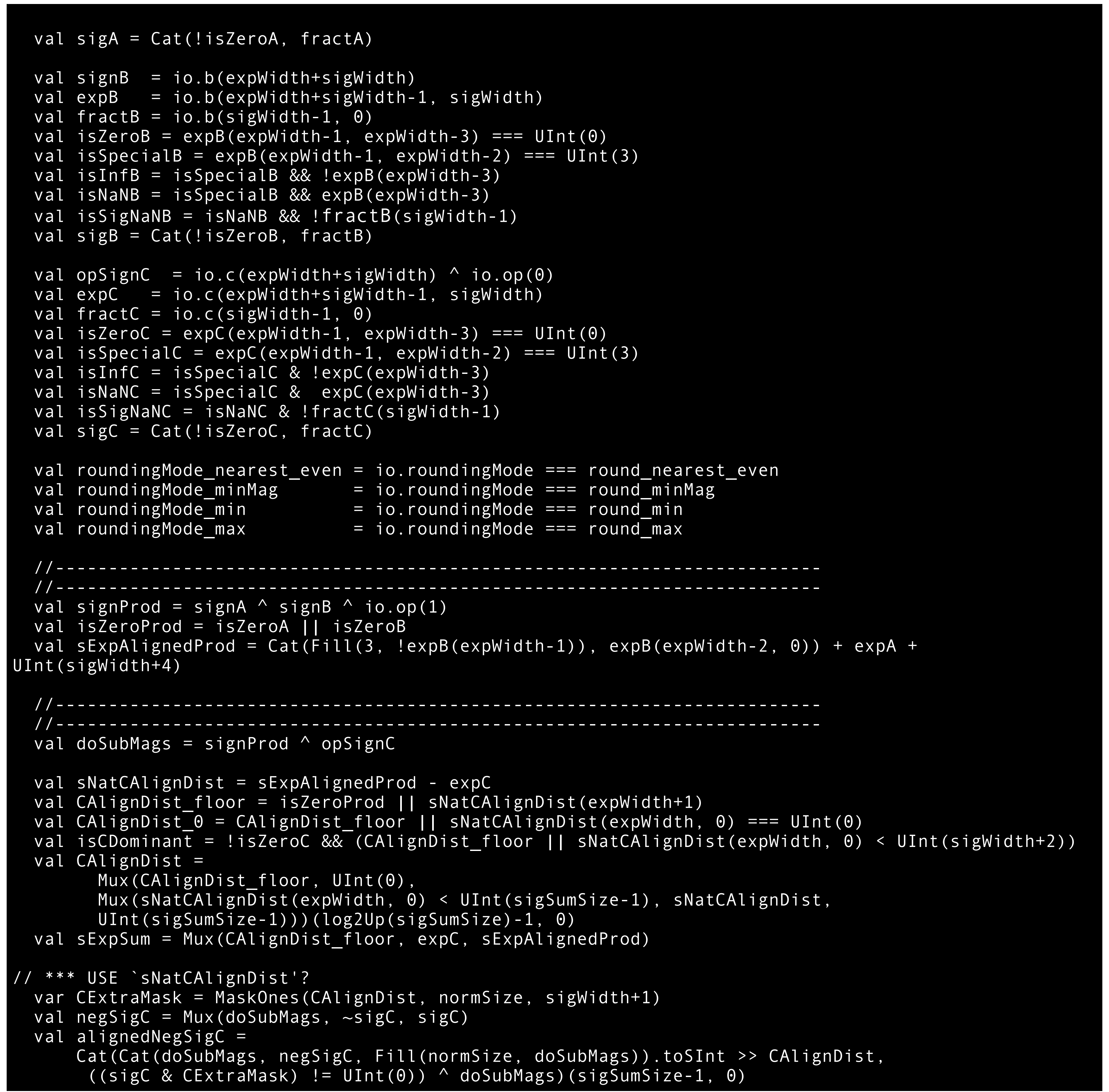
hardfloat: mulAddSubRecodedFloatN (Expert-written)



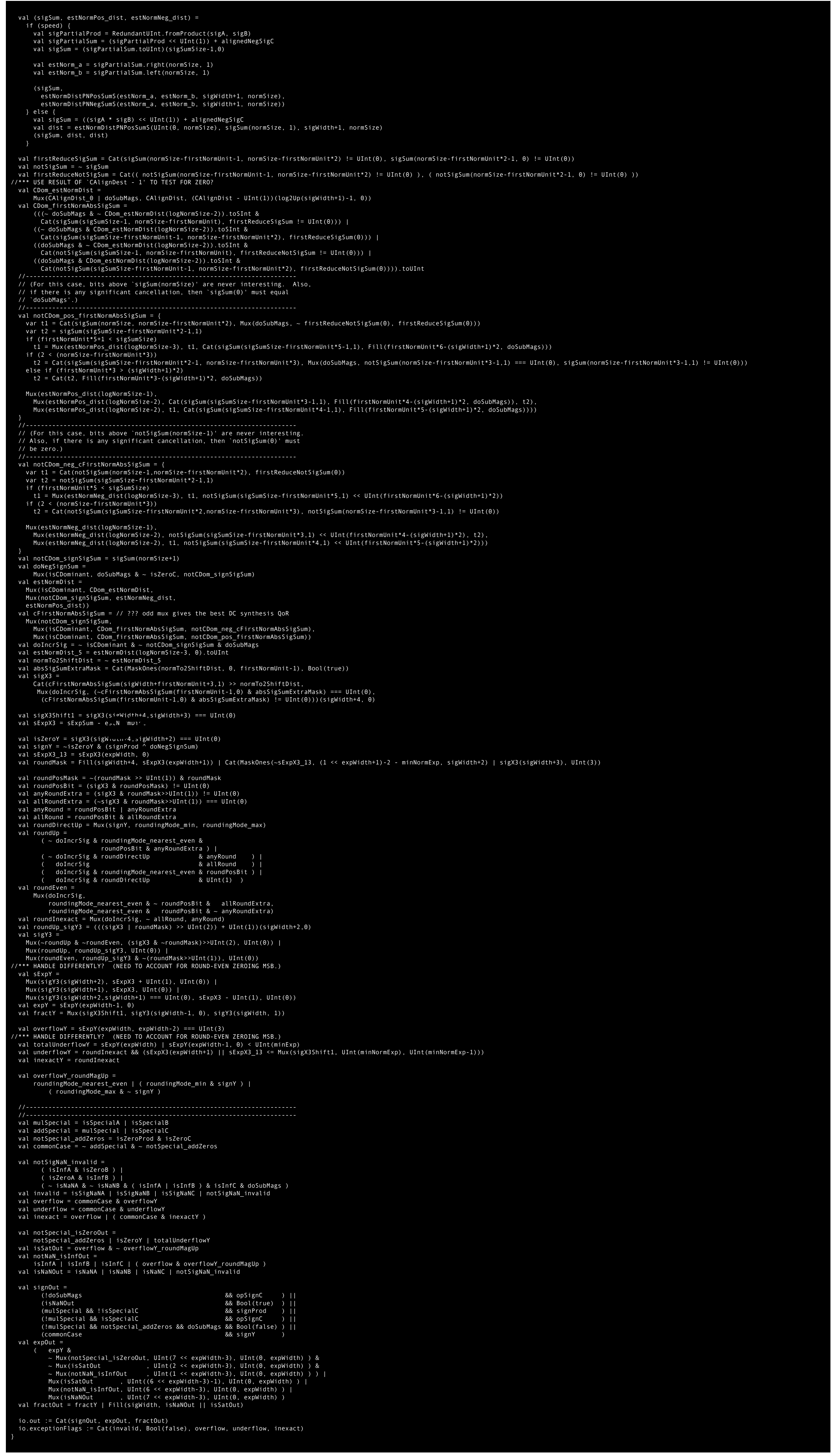
```
// See LICENSE for license details.

//*** THIS MODULE HAS NOT BEEN FULLY OPTIMIZED.
//*** DO THIS ANOTHER WAY?

package hardfloat

import Chisel._
import Node._
import consts._

object MaskOnes
{
  def apply(in: UInt, start: Int, length: Int): UInt = {
    val top = 1 << in.getWidth
    val shift = SInt(BigInt(-1) << top) >> in
    Reverse(shift(top-1-start,top-length-start))
  }
}

object estNormDistPNNegSumS
{
  def priorityEncode(key: UInt, n: Int, s: Int) = {
    if (Module.backend.isInstanceOf[CppBackend]) UInt(n+s-1) -
Log2(key(s-1,0), s)
    else PriorityMux((0 until s).map(i => (key(s-1-i), UInt(n+i,
log2Up(n+s-1)))))
  }

  def apply(a: UInt, b: UInt, n: Int, s: Int) =
    priorityEncode((a ^ b) ^ ~((a & b) << UInt(1)), n, s)
}

object estNormDistPNPosSumS
{
  def apply(a: UInt, b: UInt, n: Int, s: Int) =
    estNormDistPNNegSumS.priorityEncode((a ^ b) ^ ((a | b) << UInt(1)), n, s)
}

class mulAddSubRecodedFloatN_io(sigWidth: Int, expWidth: Int) extends Bundle
{
  val op = UInt(INPUT, 2)
  val a = UInt(INPUT, expWidth+sigWidth+1)
  val b = UInt(INPUT, expWidth+sigWidth+1)
  val c = UInt(INPUT, expWidth+sigWidth+1)
  val roundingMode = UInt(INPUT, 2)
  val out = UInt(OUTPUT, expWidth+sigWidth+1)
  val exceptionFlags = UInt(OUTPUT, 5)
}

class mulAddSubRecodedFloatN(sigWidth: Int, expWidth: Int, speed: Boolean =
false) extends Module {
  val io = new mulAddSubRecodedFloatN_io(sigWidth, expWidth)

  val sigSumSize = (sigWidth+2)*3
  val normSize = (sigWidth+2)*2
  val logNormSize = log2Up(normSize)
  val firstNormUnit = 1 << logNormSize-2
  val minNormExp = (1 << expWidth-2) + 2
  val minExp = minNormExp - sigWidth

  val signA  = io.a(expWidth+sigWidth)
  val expA   = io.a(expWidth+sigWidth-1, sigWidth)
  val fractA = io.a(sigWidth-1, 0)
  val isZeroA = expA(expWidth-1, expWidth-3) === UInt(0)
  val isSpecialA = expA(expWidth-1, expWidth-2) === UInt(3)
  val isInfA = isSpecialA && !expA(expWidth-3)
```

OMG

(300+ in this file 2061 lines more)



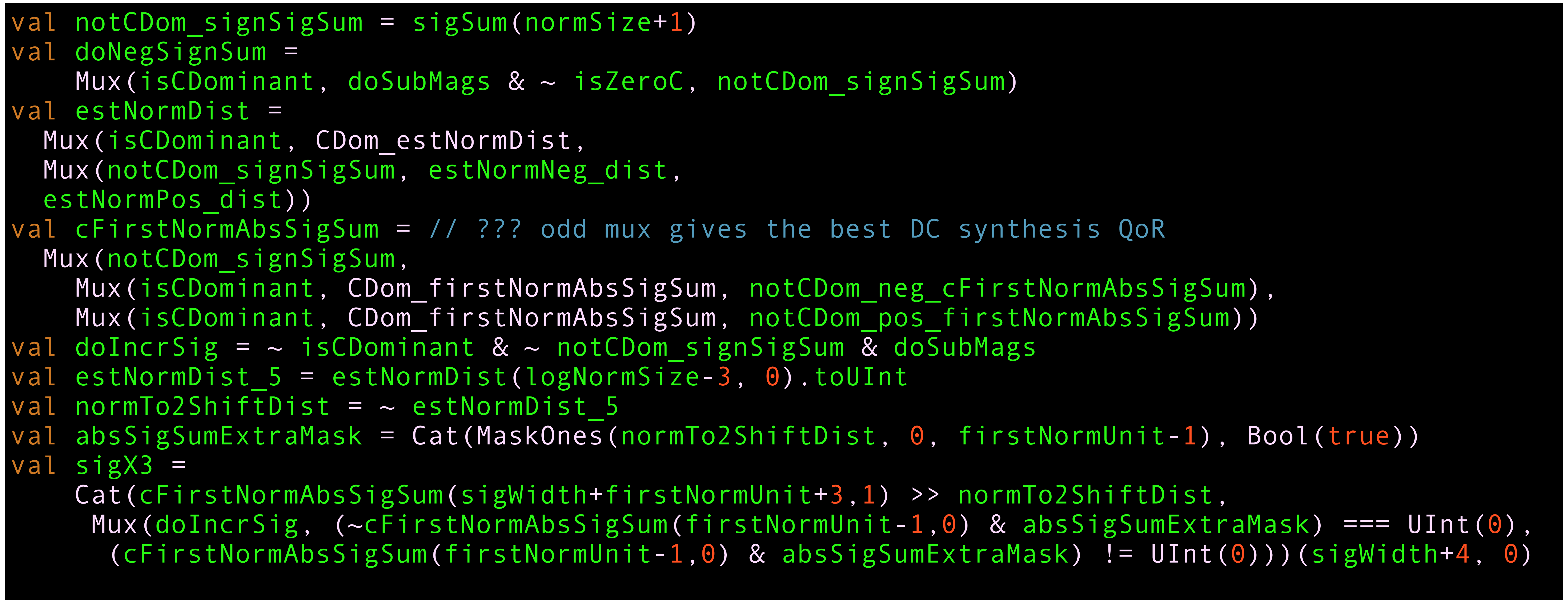
```
  val sigA = Cat(!isZeroA, fractA)

  val signB  = io.b(expWidth+sigWidth)
  val expB   = io.b(expWidth+sigWidth-1, sigWidth)
  val fractB = io.b(sigWidth-1, 0)
  val isZeroB = expB(expWidth-1, expWidth-3) === UInt(0)
  val isSpecialB = expB(expWidth-1, expWidth-2) === UInt(3)
  val isInfB = isSpecialB && !expB(expWidth-3)
  val isNaNB = isSpecialB && expB(expWidth-3)
  val isSigNaNB = isNaNB && !fractB(sigWidth-1)
  val sigB = Cat(!isZeroB, fractB)

  val opSignC  = io.c(expWidth+sigWidth) ^ io.op(0)
  val expC     = io.c(expWidth+sigWidth-1, sigWidth)
  val fractC   = io.c(sigWidth-1, 0)
  val isZeroC = expC(expWidth-1, expWidth-3) === UInt(0)
  val isSpecialC = expC(expWidth-1, expWidth-2) === UInt(3)
  val isInfC = isSpecialC & !expC(expWidth-3)
  val isNaNC = isSpecialC &  expC(expWidth-3)
  val isSigNaNC = isNaNC & !fractC(sigWidth-1)
  val sigC = Cat(!isZeroC, fractC)

  val roundingMode_nearest_even = io.roundingMode === round_nearest_even
  val roundingMode_minMag       = io.roundingMode === round_minMag
  val roundingMode_min          = io.roundingMode === round_min
  val roundingMode_max          = io.roundingMode === round_max

  //------------------------------------------------------------------------
  //------------------------------------------------------------------------
  val signProd = signA ^ signB ^ io.op(1)
  val isZeroProd = isZeroA || isZeroB
  val sExpAlignedProd = Cat(Fill(3, !expB(expWidth-1)), expB(expWidth-2, 0)) + expA +
UInt(sigWidth+4)

  //------------------------------------------------------------------------
  //------------------------------------------------------------------------
  val doSubMags = signProd ^ opSignC

  val sNatCAlignDist = sExpAlignedProd - expC
  val CAlignDist_floor = isZeroProd || sNatCAlignDist(expWidth+1)
  val CAlignDist_0 = CAlignDist_floor || sNatCAlignDist(expWidth, 0) === UInt(0)
  val isCDominant = !isZeroC && (CAlignDist_floor || sNatCAlignDist(expWidth, 0) < UInt(sigWidth+2))
  val CAlignDist =
        Mux(CAlignDist_floor, UInt(0),
        Mux(sNatCAlignDist(expWidth, 0) < UInt(sigSumSize-1), sNatCAlignDist,
        UInt(sigSumSize-1)))(log2Up(sigSumSize)-1, 0)
  val sExpSum = Mux(CAlignDist_floor, expC, sExpAlignedProd)

// *** USE `sNatCAlignDist'?
  var CExtraMask = MaskOnes(CAlignDist, normSize, sigWidth+1)
  val negSigC = Mux(doSubMags, ~sigC, sigC)
  val alignedNegSigC =
      Cat(Cat(doSubMags, negSigC, Fill(normSize, doSubMags)).toSInt >> CAlignDist,
        ((sigC & CExtraMask) != UInt(0)) ^ doSubMags)(sigSumSize-1, 0)
```



```
  val (sigSum, estNormPos_dist, estNormNeg_dist) =
    if (speed) {
      val sigPartialProd = RedundantUInt.fromProduct(sigA, sigB)
      val sigPartialSum = (sigPartialProd << UInt(1)) + alignedNegSigC
      val sigSum = (sigPartialSum.toUInt)(sigSumSize-1,0)

      val estNorm_a = sigPartialSum.right(normSize, 1)
      val estNorm_b = sigPartialSum.left(normSize, 1)

      (sigSum,
        estNormDistPNPosSumS(estNorm_a, estNorm_b, sigWidth+1, normSize),
        estNormDistPNNegSumS(estNorm_a, estNorm_b, sigWidth+1, normSize))
    } else {
      val sigSum = ((sigA * sigB) << UInt(1)) + alignedNegSigC
      val dist = estNormDistPNPosSumS(UInt(0, normSize), sigSum(normSize, 1), sigWidth+1, normSize)
      (sigSum, dist, dist)
    }

  val firstReduceSigSum = Cat(sigSum(normSize-firstNormUnit-1, normSize-firstNormUnit*2) != UInt(0), sigSum(normSize-firstNormUnit*2-1, 0) != UInt(0))
  val notSigSum = ~ sigSum
  val firstReduceNotSigSum = Cat(( notSigSum(normSize-firstNormUnit-1, normSize-firstNormUnit*2) != UInt(0) ), ( notSigSum(normSize-firstNormUnit*2-1, 0) != UInt(0) ))
//*** USE RESULT OF `CAlignDest - 1' TO TEST FOR ZERO?
  val CDom_estNormDist =
      Mux(CAlignDist_0 | doSubMags, CAlignDist, (CAlignDist - UInt(1))(log2Up(sigWidth+1)-1, 0))
  val CDom_firstNormAbsSigSum =
      (((~ doSubMags & ~ CDom_estNormDist(logNormSize-2)).toSInt &
        Cat(sigSum(sigSumSize-1, normSize-firstNormUnit), firstReduceSigSum != UInt(0))) |
      ((~ doSubMags & CDom_estNormDist(logNormSize-2)).toSInt &
        Cat(sigSum(sigSumSize-firstNormUnit-1, normSize-firstNormUnit*2), firstReduceSigSum(0))) |
      ((doSubMags & ~ CDom_estNormDist(logNormSize-2)).toSInt &
        Cat(notSigSum(sigSumSize-1, normSize-firstNormUnit), firstReduceNotSigSum != UInt(0))) |
      ((doSubMags & CDom_estNormDist(logNormSize-2)).toSInt &
        Cat(notSigSum(sigSumSize-firstNormUnit-1, normSize-firstNormUnit*2), firstReduceNotSigSum(0)))).toUInt
  //------------------------------------------------------------------------
  // (For this case, bits above `sigSum(normSize)' are never interesting.  Also,
  // if there is any significant cancellation, then `sigSum(0)' must equal
  // `doSubMags'.)
  //------------------------------------------------------------------------
  val notCDom_pos_firstNormAbsSigSum = {
    var t1 = Cat(sigSum(normSize, normSize-firstNormUnit*2), Mux(doSubMags, ~ firstReduceNotSigSum(0), firstReduceSigSum(0)))
    var t2 = sigSum(sigSumSize-firstNormUnit*2-1,1)
    if (firstNormUnit*5+1 < sigSumSize)
      t1 = Mux(estNormPos_dist(logNormSize-3), t1, Cat(sigSum(sigSumSize-firstNormUnit*5-1,1), Fill(firstNormUnit*6-(sigWidth+1)*2, doSubMags)))
    if (2 < (normSize-firstNormUnit*3))
      t2 = Cat(sigSum(sigSumSize-firstNormUnit*2-1, normSize-firstNormUnit*3), Mux(doSubMags, notSigSum(normSize-firstNormUnit*3-1,1) === UInt(0), sigSum(normSize-firstNormUnit*3-1,1) != UInt(0)))
    else if (firstNormUnit*3 > (sigWidth+1)*2)
      t2 = Cat(t2, Fill(firstNormUnit*3-(sigWidth+1)*2, doSubMags))

    Mux(estNormPos_dist(logNormSize-1),
      Mux(estNormPos_dist(logNormSize-2), Cat(sigSum(sigSumSize-firstNormUnit*3-1,1), Fill(firstNormUnit*4-(sigWidth+1)*2, doSubMags)), t2),
      Mux(estNormPos_dist(logNormSize-2), t1, Cat(sigSum(sigSumSize-firstNormUnit*4-1,1), Fill(firstNormUnit*5-(sigWidth+1)*2, doSubMags))))
  }
  //------------------------------------------------------------------------
  // (For this case, bits above `notSigSum(normSize-1)' are never interesting.
  // Also, if there is any significant cancellation, then `notSigSum(0)' must
  // be zero.)
  //------------------------------------------------------------------------
  val notCDom_neg_cFirstNormAbsSigSum = {
    var t1 = Cat(notSigSum(normSize-1,normSize-firstNormUnit*2), firstReduceNotSigSum(0))
    var t2 = notSigSum(sigSumSize-firstNormUnit*2-1,1)
    if (firstNormUnit*5 < sigSumSize)
      t1 = Mux(estNormNeg_dist(logNormSize-3), t1, notSigSum(sigSumSize-firstNormUnit*5,1) << UInt(firstNormUnit*6-(sigWidth+1)*2))
    if (2 < (normSize-firstNormUnit*3))
      t2 = Cat(notSigSum(sigSumSize-firstNormUnit*2,normSize-firstNormUnit*3), notSigSum(normSize-firstNormUnit*3-1,1) != UInt(0))

    Mux(estNormNeg_dist(logNormSize-1),
      Mux(estNormNeg_dist(logNormSize-2), notSigSum(sigSumSize-firstNormUnit*3,1) << UInt(firstNormUnit*4-(sigWidth+1)*2), t2),
      Mux(estNormNeg_dist(logNormSize-2), t1, notSigSum(sigSumSize-firstNormUnit*4,1) << UInt(firstNormUnit*5-(sigWidth+1)*2)))
  }
  val notCDom_signSigSum = sigSum(normSize+1)
  val doNegSignSum =
      Mux(isCDominant, doSubMags & ~ isZeroC, notCDom_signSigSum)
  val estNormDist =
    Mux(isCDominant, CDom_estNormDist,
    Mux(notCDom_signSigSum, estNormNeg_dist,
    estNormPos_dist))
  val cFirstNormAbsSigSum = // ??? odd mux gives the best DC synthesis QoR
    Mux(notCDom_signSigSum,
      Mux(isCDominant, CDom_firstNormAbsSigSum, notCDom_neg_cFirstNormAbsSigSum),
      Mux(isCDominant, CDom_firstNormAbsSigSum, notCDom_pos_firstNormAbsSigSum))
  val doIncrSig = ~ isCDominant & ~ notCDom_signSigSum & doSubMags
  val estNormDist_5 = estNormDist(logNormSize-3, 0).toUInt
  val normTo2ShiftDist = ~ estNormDist_5
  val absSigSumExtraMask = Cat(MaskOnes(normTo2ShiftDist, 0, firstNormUnit-1), Bool(true))
  val sigX3 =
      Cat(cFirstNormAbsSigSum(sigWidth+firstNormUnit+3,1) >> normTo2ShiftDist,
       Mux(doIncrSig, (~cFirstNormAbsSigSum(firstNormUnit-1,0) & absSigSumExtraMask) === UInt(0),
        (cFirstNormAbsSigSum(firstNormUnit-1,0) & absSigSumExtraMask) != UInt(0)))(sigWidth+4, 0)

  val sigX3Shift1 = sigX3(sigWidth+4,sigWidth+3) === UInt(0)
  val sExpX3 = sExpSum - estNormDist

  val isZeroY = sigX3(sigWidth+4,sigWidth+2) === UInt(0)
  val signY = ~isZeroY & (signProd ^ doNegSignSum)
  val sExpX3_13 = sExpX3(expWidth, 0)
  val roundMask = Fill(sigWidth+4, sExpX3(expWidth+1)) | Cat(MaskOnes(~sExpX3_13, (1 << expWidth+1)-2 - minNormExp, sigWidth+2) | sigX3(sigWidth+3), UInt(3))

  val roundPosMask = ~(roundMask >> UInt(1)) & roundMask
  val roundPosBit = (sigX3 & roundPosMask) != UInt(0)
  val anyRoundExtra = (sigX3 & roundMask>>UInt(1)) != UInt(0)
  val allRoundExtra = (~sigX3 & roundMask>>UInt(1)) === UInt(0)
  val anyRound = roundPosBit | anyRoundExtra
  val allRound = roundPosBit & allRoundExtra
  val roundDirectUp = Mux(signY, roundingMode_min, roundingMode_max)
  val roundUp =
        ( ~ doIncrSig & roundingMode_nearest_even &
                        roundPosBit & anyRoundExtra ) |
        ( ~ doIncrSig & roundDirectUp             & anyRound    ) |
        (   doIncrSig                             & allRound    ) |
        (   doIncrSig & roundingMode_nearest_even & roundPosBit ) |
        (   doIncrSig & roundDirectUp             & UInt(1)  )
  val roundEven =
      Mux(doIncrSig,
          roundingMode_nearest_even & ~ roundPosBit &   allRoundExtra,
          roundingMode_nearest_even &   roundPosBit & ~ anyRoundExtra)
  val roundInexact = Mux(doIncrSig, ~ allRound, anyRound)
  val roundUp_sigY3 = (((sigX3 | roundMask) >> UInt(2)) + UInt(1))(sigWidth+2,0)
  val sigY3 =
    Mux(~roundUp & ~roundEven, (sigX3 & ~roundMask)>>UInt(2), UInt(0)) |
    Mux(roundUp, roundUp_sigY3, UInt(0)) |
    Mux(roundEven, roundUp_sigY3 & ~(roundMask>>UInt(1)), UInt(0))
//*** HANDLE DIFFERENTLY?  (NEED TO ACCOUNT FOR ROUND-EVEN ZEROING MSB.)
  val sExpY =
    Mux(sigY3(sigWidth+2), sExpX3 + UInt(1), UInt(0)) |
    Mux(sigY3(sigWidth+1), sExpX3, UInt(0)) |
    Mux(sigY3(sigWidth+2,sigWidth+1) === UInt(0), sExpX3 - UInt(1), UInt(0))
  val expY = sExpY(expWidth-1, 0)
  val fractY = Mux(sigX3Shift1, sigY3(sigWidth-1, 0), sigY3(sigWidth, 1))

  val overflowY = sExpY(expWidth, expWidth-2) === UInt(3)
//*** HANDLE DIFFERENTLY?  (NEED TO ACCOUNT FOR ROUND-EVEN ZEROING MSB.)
  val totalUnderflowY = sExpY(expWidth) | sExpY(expWidth-1, 0) < UInt(minExp)
  val underflowY = roundInexact && (sExpX3(expWidth+1) || sExpX3_13 <= Mux(sigX3Shift1, UInt(minNormExp), UInt(minNormExp-1)))
  val inexactY = roundInexact

  val overflowY_roundMagUp =
      roundingMode_nearest_even | ( roundingMode_min & signY ) |
          ( roundingMode_max & ~ signY )

  //------------------------------------------------------------------------
  //------------------------------------------------------------------------
  val mulSpecial = isSpecialA | isSpecialB
  val addSpecial = mulSpecial | isSpecialC
  val notSpecial_addZeros = isZeroProd & isZeroC
  val commonCase = ~ addSpecial & ~ notSpecial_addZeros

  val notSigNaN_invalid =
        ( isInfA & isZeroB ) |
        ( isZeroA & isInfB ) |
        ( ~ isNaNA & ~ isNaNB & ( isInfA | isInfB ) & isInfC & doSubMags )
  val invalid = isSigNaNA | isSigNaNB | isSigNaNC | notSigNaN_invalid
  val overflow = commonCase & overflowY
  val underflow = commonCase & underflowY
  val inexact = overflow | ( commonCase & inexactY )

  val notSpecial_isZeroOut =
      notSpecial_addZeros | isZeroY | totalUnderflowY
  val isSatOut = overflow & ~ overflowY_roundMagUp
  val notNaN_isInfOut =
      isInfA | isInfB | isInfC | ( overflow & overflowY_roundMagUp )
  val isNaNOut = isNaNA | isNaNB | isNaNC | notSigNaN_invalid

  val signOut =
        (!doSubMags                                      && opSignC     ) ||
        (isNaNOut                                        && Bool(true)  ) ||
        (mulSpecial && !isSpecialC                       && signProd    ) ||
        (!mulSpecial && isSpecialC                       && opSignC     ) ||
        (!mulSpecial && notSpecial_addZeros && doSubMags && Bool(false) ) ||
        (commonCase                                      && signY       )
  val expOut =
      (   expY &
          ~ Mux(notSpecial_isZeroOut, UInt(7 << expWidth-3), UInt(0, expWidth) ) &
          ~ Mux(isSatOut            , UInt(2 << expWidth-3), UInt(0, expWidth) ) &
          ~ Mux(notNaN_isInfOut     , UInt(1 << expWidth-3), UInt(0, expWidth) ) ) |
          Mux(isSatOut       , UInt((6 << expWidth-3)-1), UInt(0, expWidth) ) |
          Mux(notNaN_isInfOut, UInt(6 << expWidth-3), UInt(0, expWidth) ) |
          Mux(isNaNOut       , UInt(7 << expWidth-3), UInt(0, expWidth) )
  val fractOut = fractY | Fill(sigWidth, isNaNOut || isSatOut)

  io.out := Cat(signOut, expOut, fractOut)
  io.exceptionFlags := Cat(invalid, Bool(false), overflow, underflow, inexact)
}
```

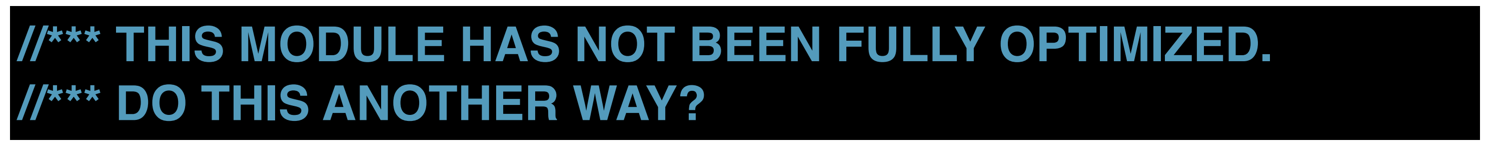
hardfloat: mulAddSubRecodedFloatN (Expert-written)

a little snippet of lines (189-207):



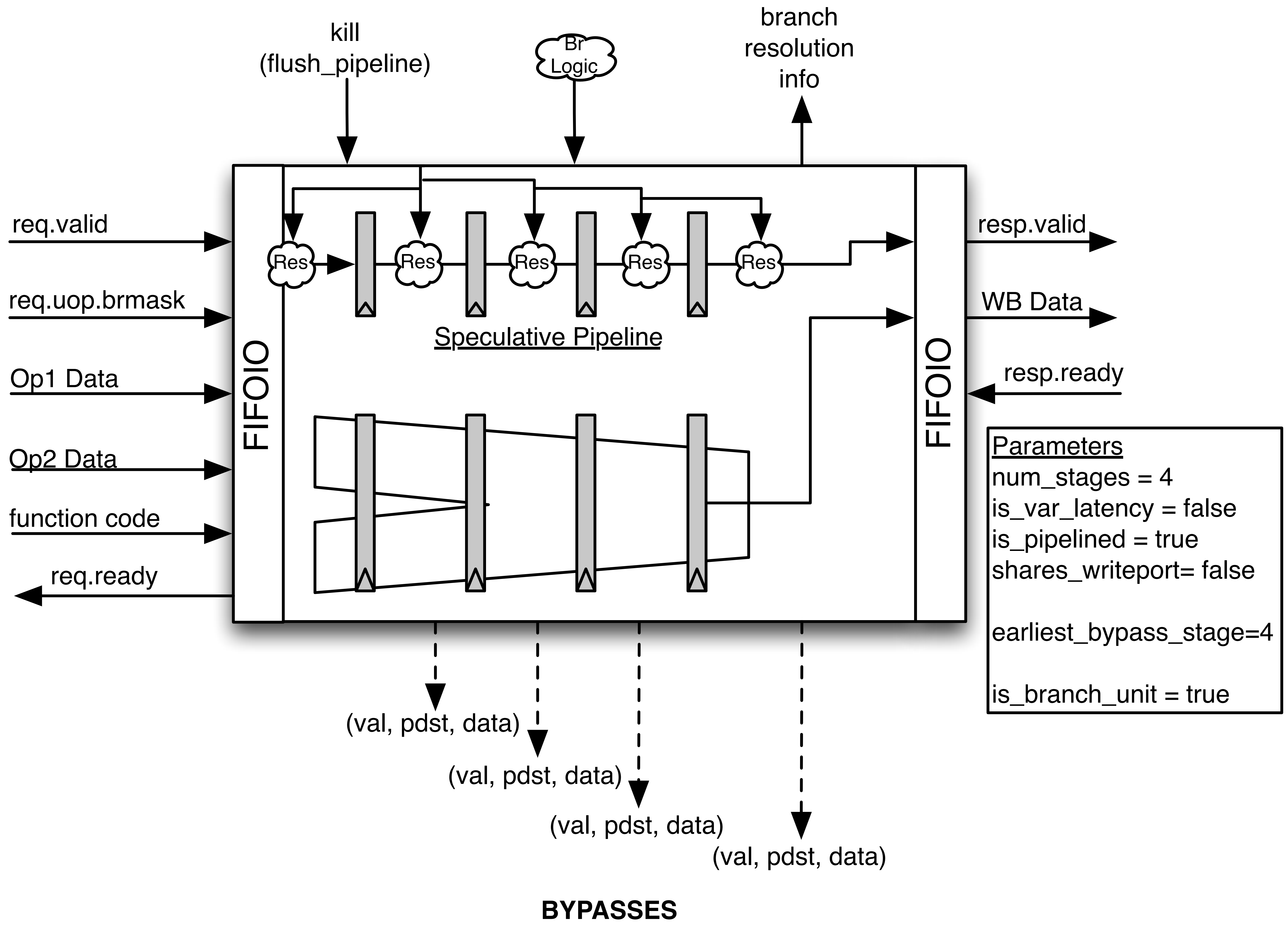
```
val notCDom_signSigSum = sigSum(normSize+1)
val doNegSignSum =
    Mux(isCDominant, doSubMags & ~ isZeroC, notCDom_signSigSum)
val estNormDist =
  Mux(isCDominant, CDom_estNormDist,
  Mux(notCDom_signSigSum, estNormNeg_dist,
  estNormPos_dist))
val cFirstNormAbsSigSum = // ??? odd mux gives the best DC synthesis QoR
  Mux(notCDom_signSigSum,
    Mux(isCDominant, CDom_firstNormAbsSigSum, notCDom_neg_cFirstNormAbsSigSum),
    Mux(isCDominant, CDom_firstNormAbsSigSum, notCDom_pos_firstNormAbsSigSum))
val doIncrSig = ~ isCDominant & ~ notCDom_signSigSum & doSubMags
val estNormDist_5 = estNormDist(logNormSize-3, 0).toUInt
val normTo2ShiftDist = ~ estNormDist_5
val absSigSumExtraMask = Cat(MaskOnes(normTo2ShiftDist, 0, firstNormUnit-1), Bool(true))
val sigX3 =
    Cat(cFirstNormAbsSigSum(sigWidth+firstNormUnit+3,1) >> normTo2ShiftDist,
     Mux(doIncrSig, (~cFirstNormAbsSigSum(firstNormUnit-1,0) & absSigSumExtraMask) === UInt(0),
      (cFirstNormAbsSigSum(firstNormUnit-1,0) & absSigSumExtraMask) != UInt(0)))(sigWidth+4, 0)
```

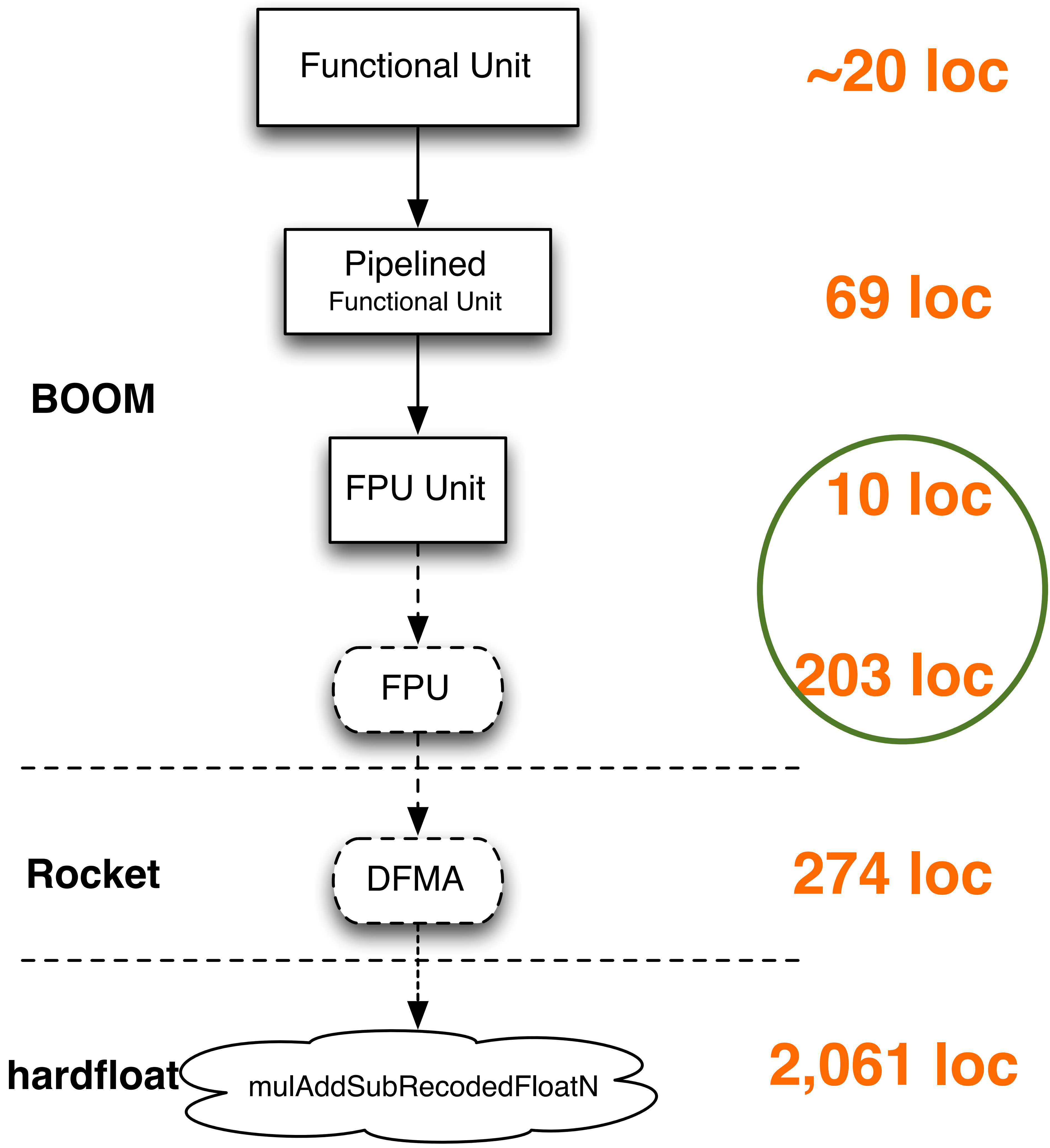
and at the top...



```
//*** THIS MODULE HAS NOT BEEN FULLY OPTIMIZED.
//*** DO THIS ANOTHER WAY?
```

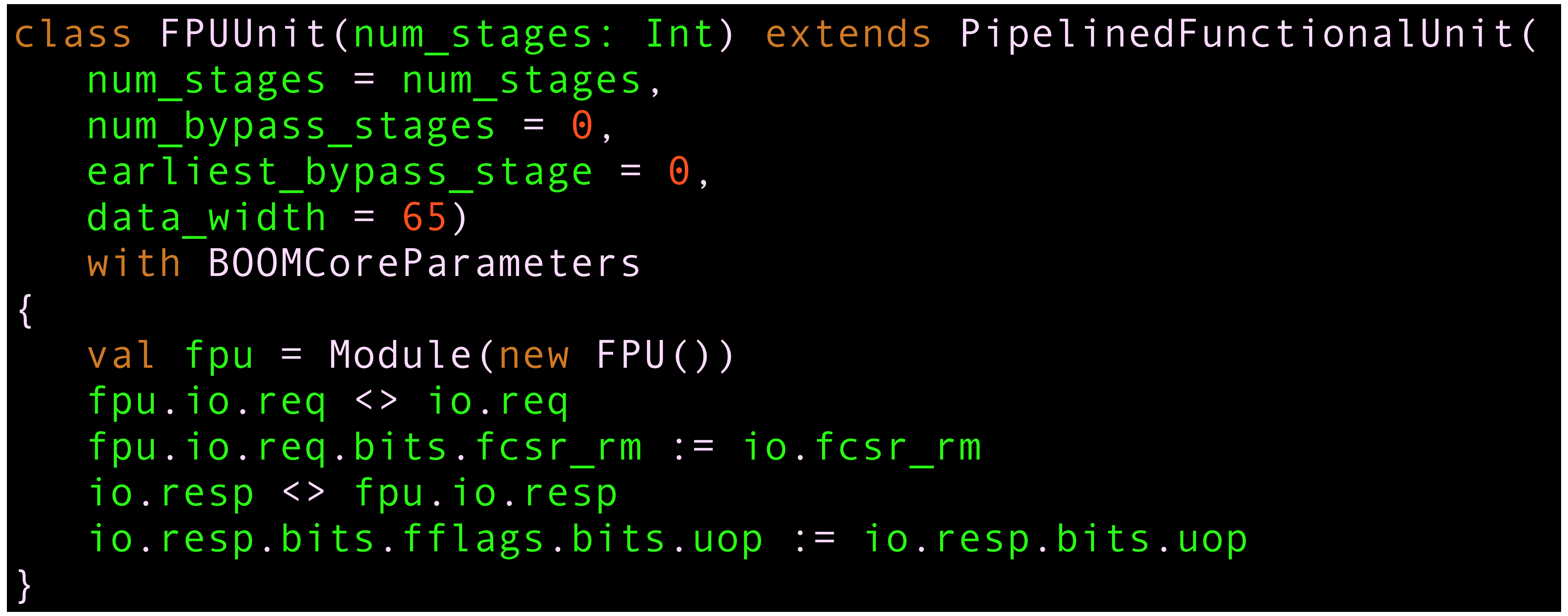
Adding Floating-point



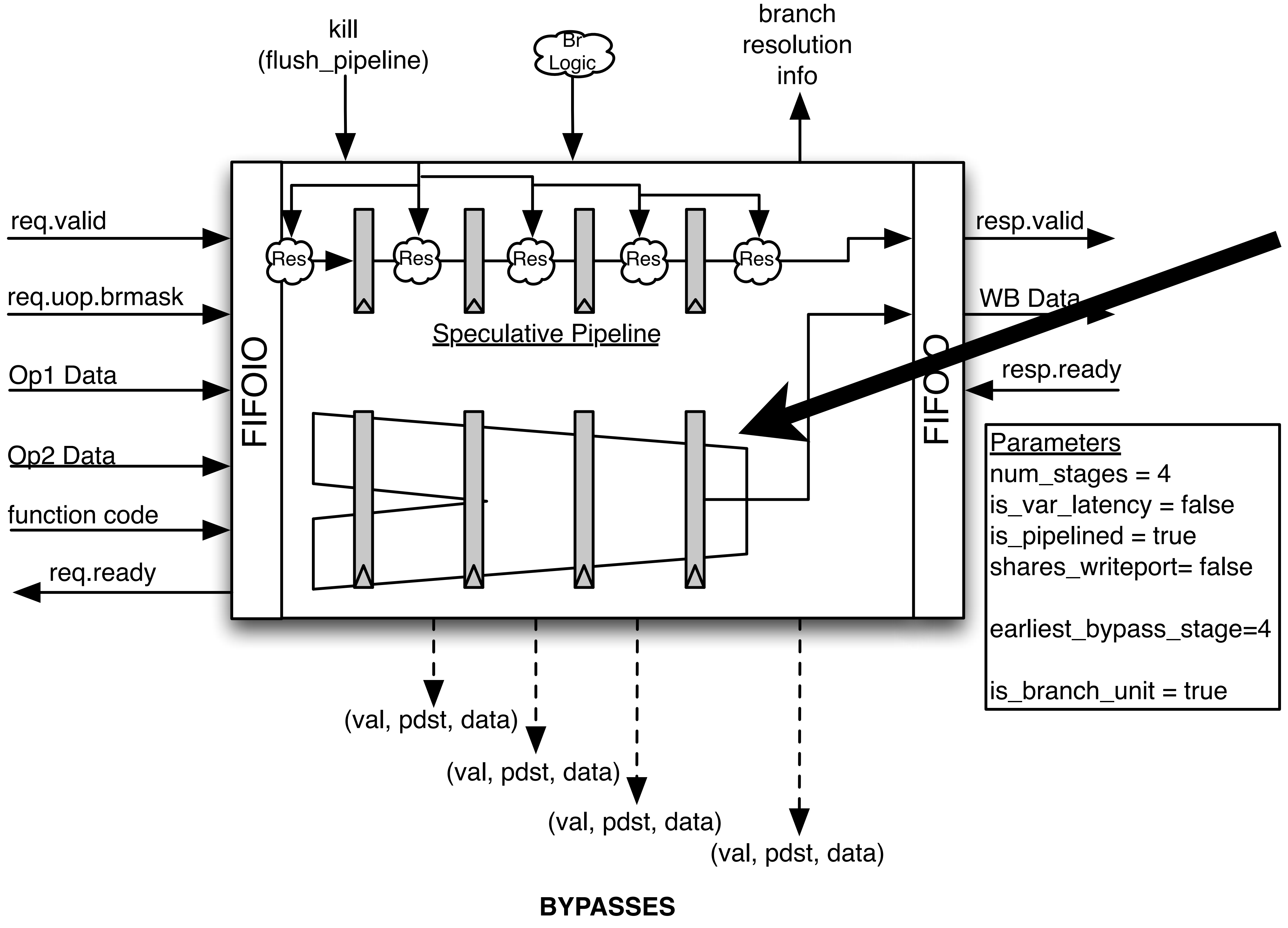


- 12 days to add SP, DP floating point
- 1092 lines of code added

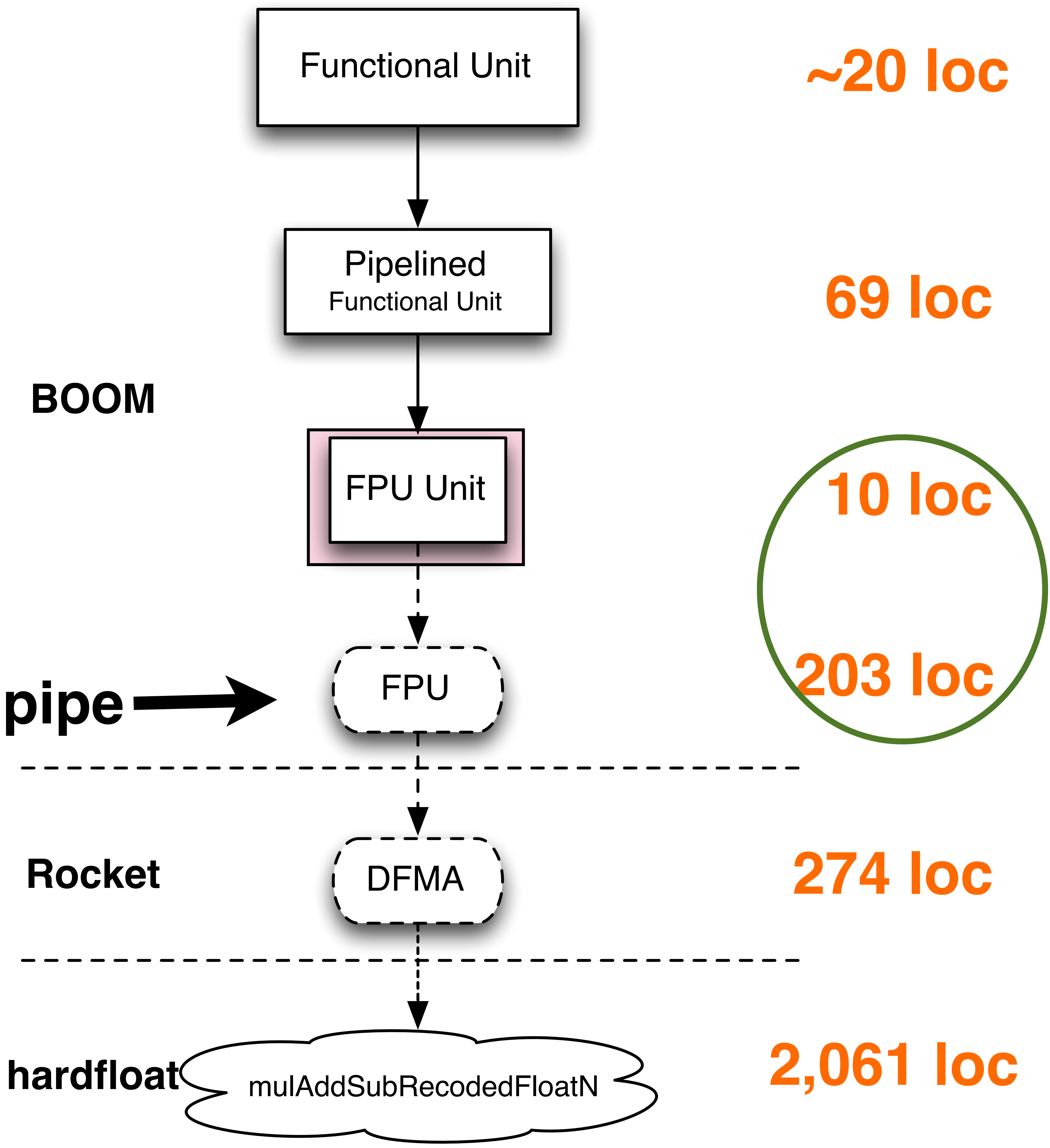
Adding Floating-point



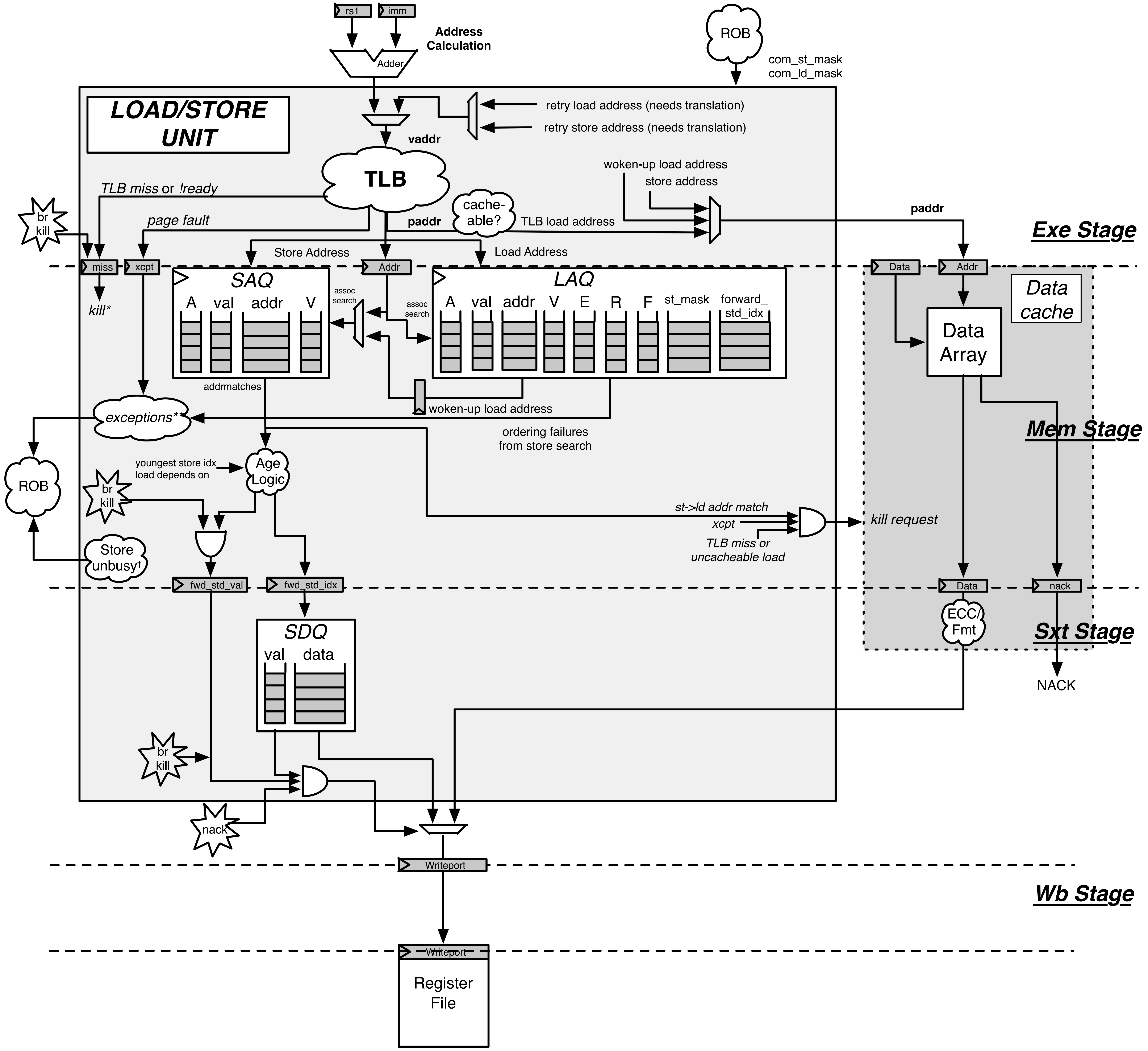
```
class FPUUnit(num_stages: Int) extends PipelinedFunctionalUnit(
   num_stages = num_stages,
   num_bypass_stages = 0,
   earliest_bypass_stage = 0,
   data_width = 65)
   with BOOMCoreParameters
{
   val fpu = Module(new FPU())
   fpu.io.req <> io.req
   fpu.io.req.bits.fcsr_rm := io.fcsr_rm
   io.resp <> fpu.io.resp
   io.resp.bits.fflags.bits.uop := io.resp.bits.uop
}
```



dumb, compute pipe

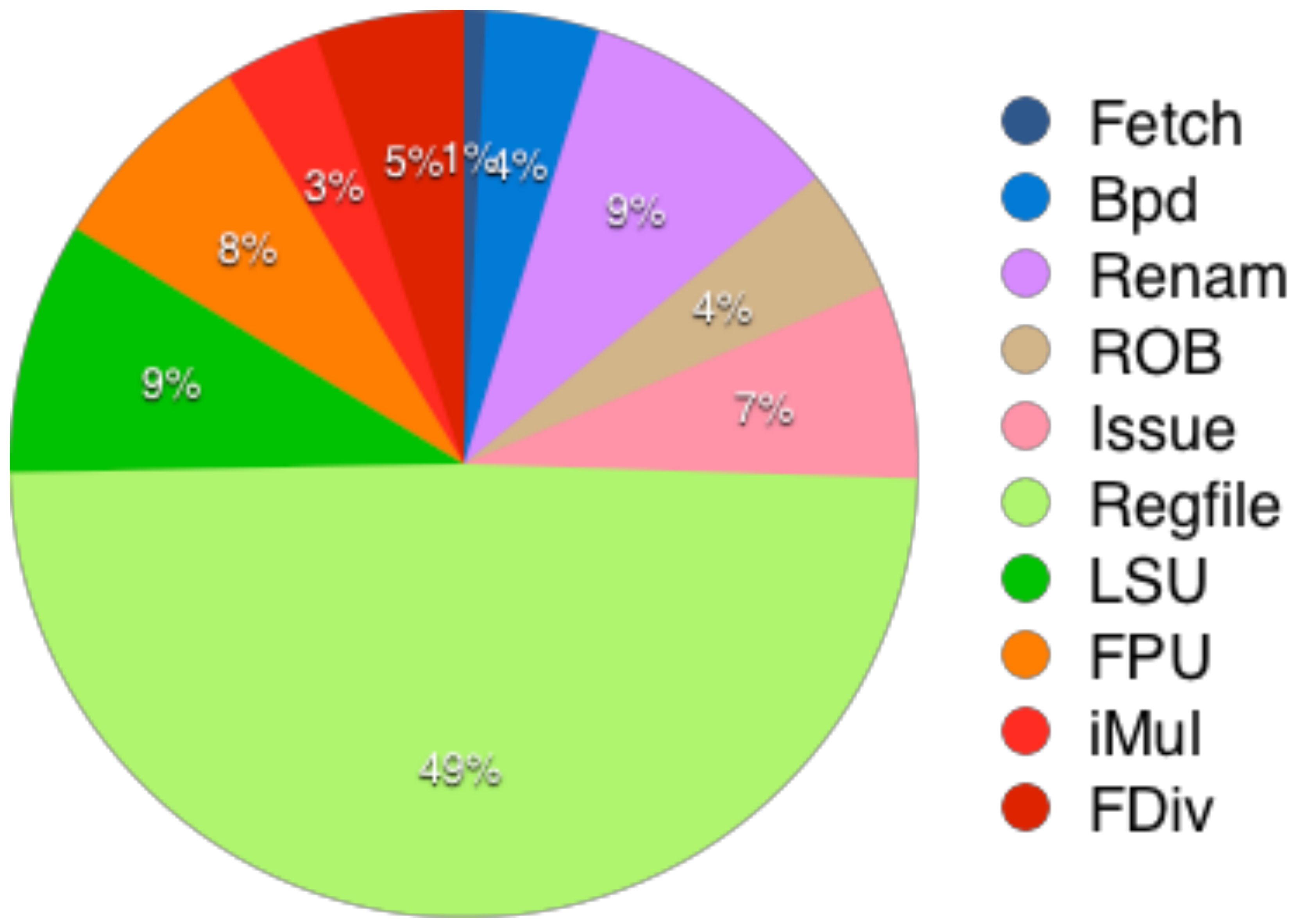


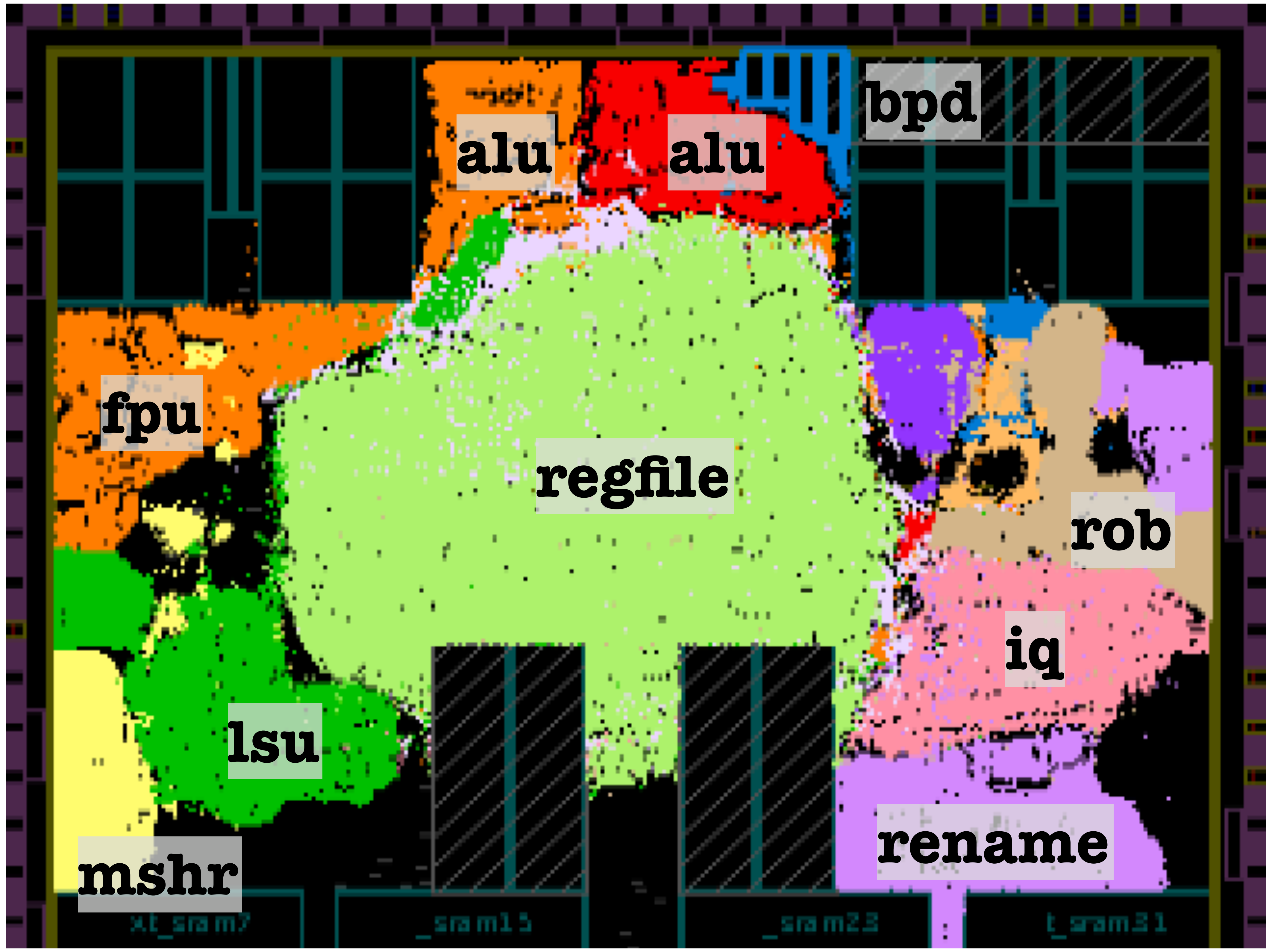
Load/Store Unit



Building a Register File (the first P&R)

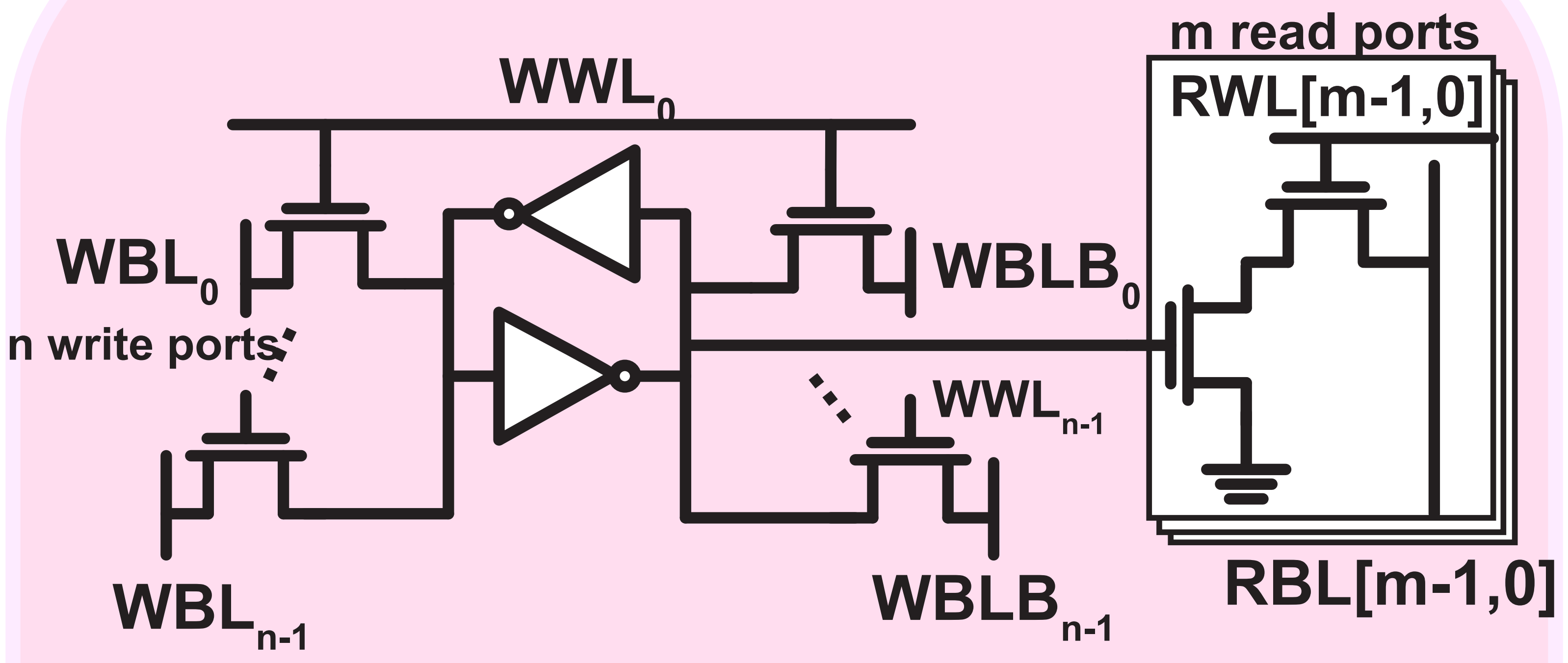
- BOOMv1 -- 7r3w with 110 registers (INT/FP)
- Initial Regfile design was infeasible for layout
- critical paths in issue-select and register read
- Not DRC/LVS clean





Multi-port Register File for Design Exploration

Transistor-level



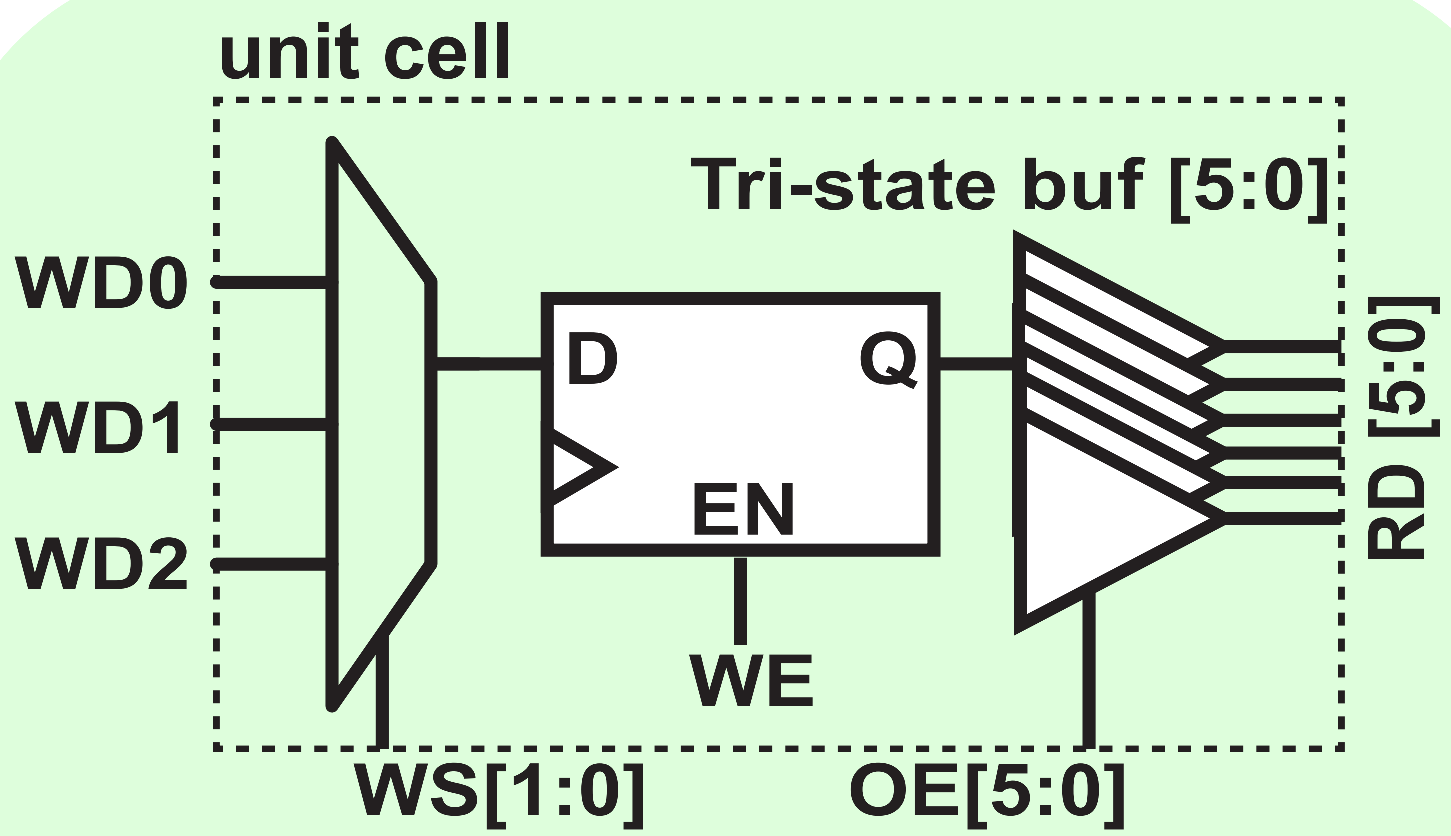
Advantage

- Compact area
- Higher performance

Challenge

- Long design cycle
- Difficult for architecture design exploration

Gate-level



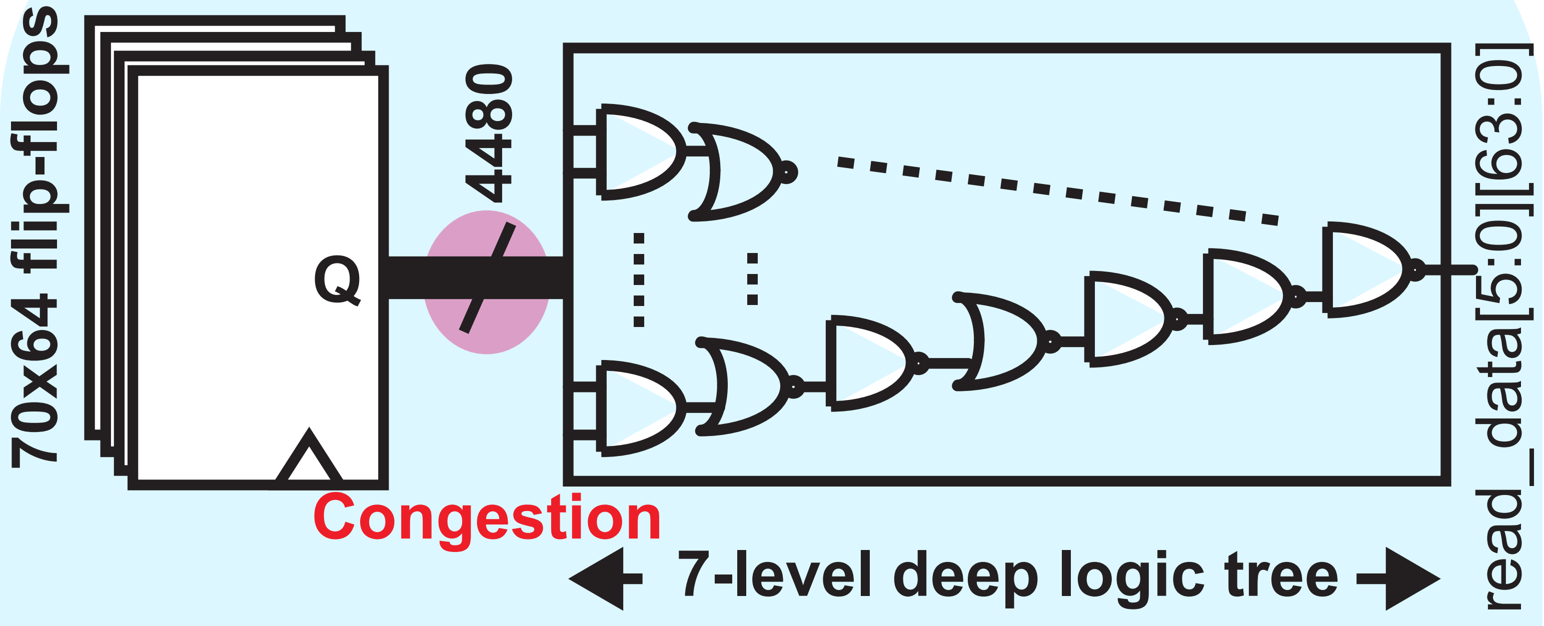
Advantage

- Rapid design exploration
- Shared read wires solve routing congestion

Challenge

- Guided place-and-route for area/performance optimization

RTL



Advantage

- Low design effort
- Rapid design exploration

Challenge

- Large area
- Bad performance
- Routing congestion

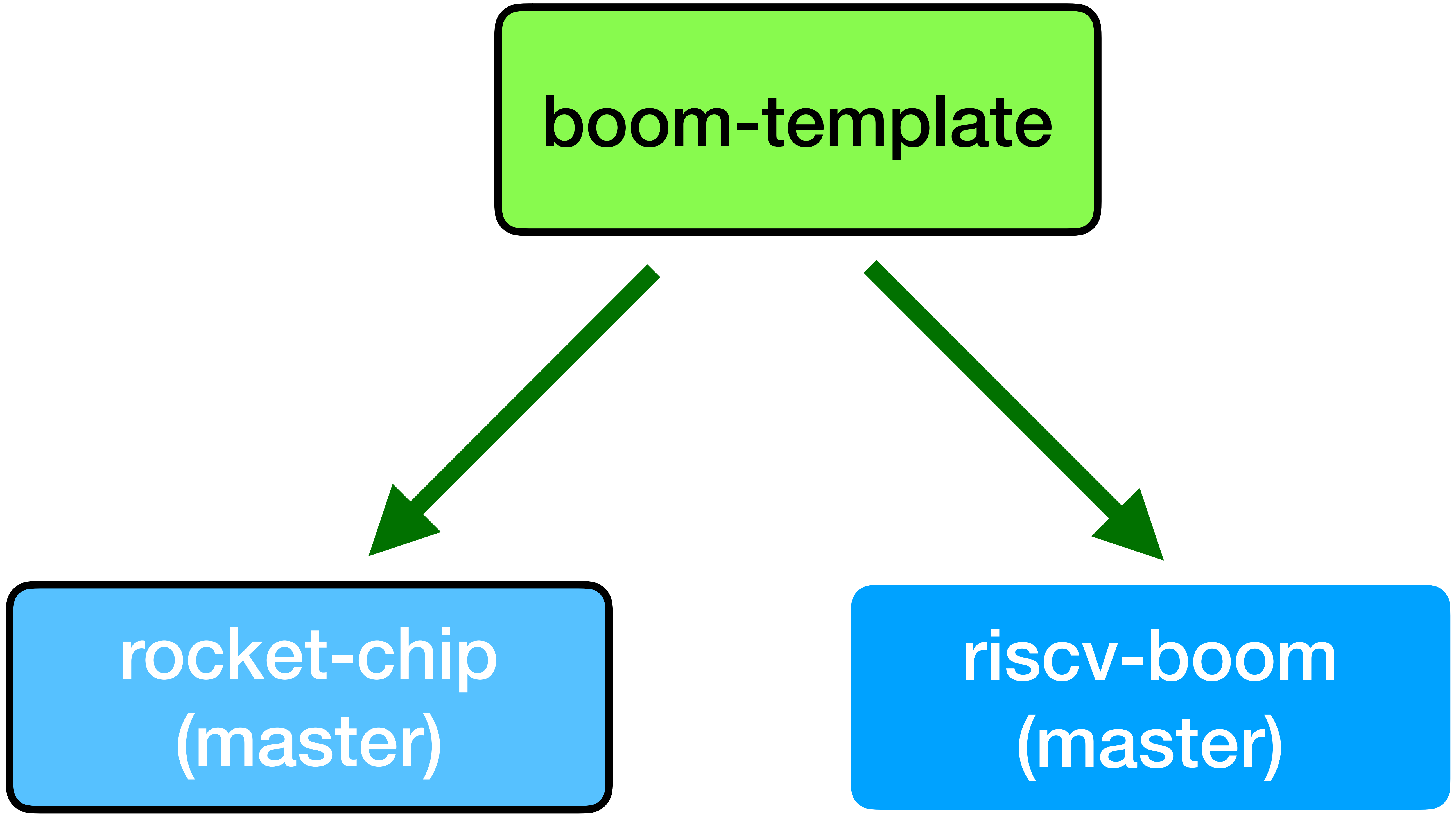
Verification

- Directed tests and a randomized torture generator (riscv-torture).
- Verilator/VCS/FPGA simulation at RTL.
- VCS for post-gl/par simulation.
- Speculative OOO pipelines are difficult to get good coverage on.
  - Need tests that build up a lot of speculative state.
  - Need tests that cover OS- and platform-level use-cases.
- Assertions are king.
- Currently moving towards using **co-simulation** against an ISA simulator (using CSRFile's trace port).

Using the Code

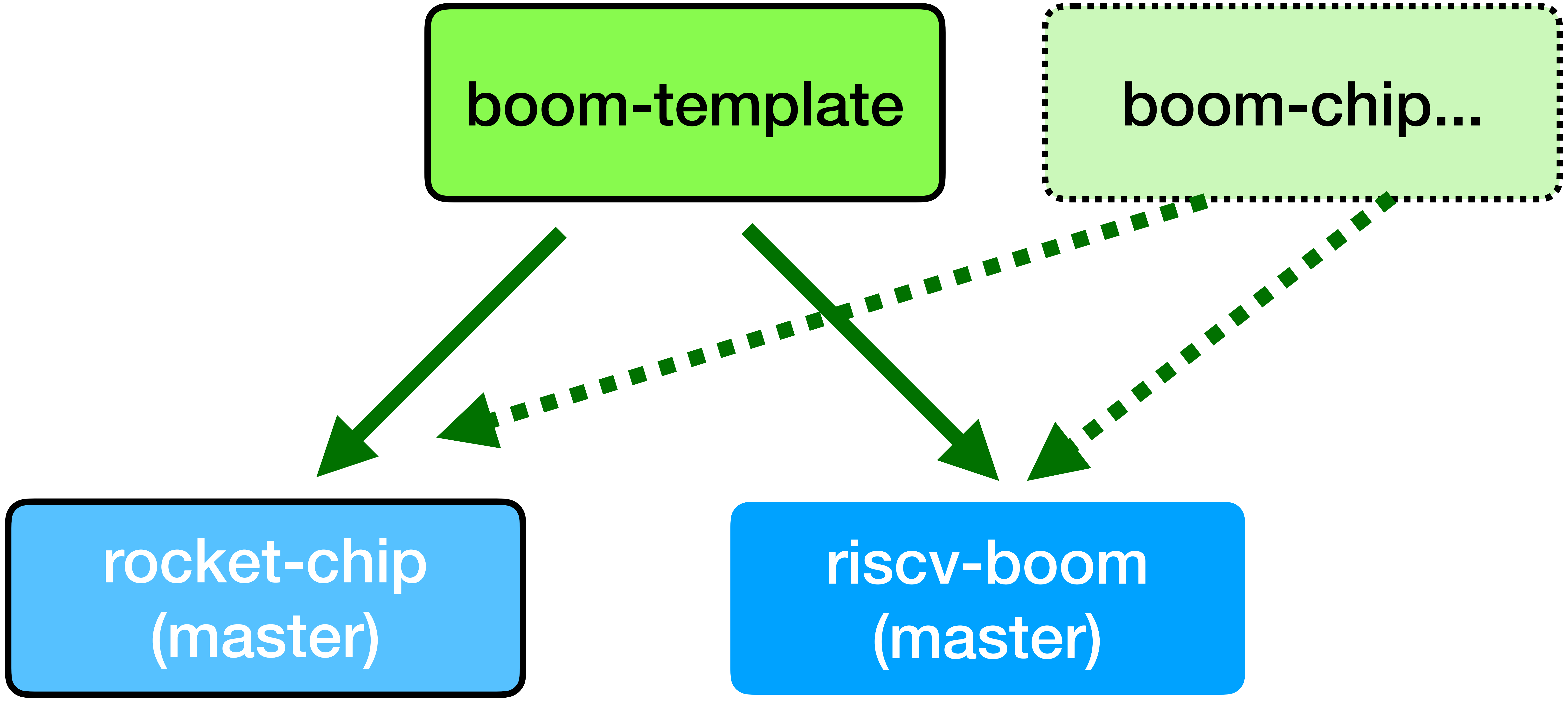
- https://github.com/riscv-boom/riscv-boom
- IntelliJ is awesome
  - https://github.com/riscv-boom/boom-template#vimbash-isnt-a-development-environment-how-do-i-setup-an-intellij-ide

Code Repo Organization



- rocket-chip, riscv-boom are git submodules of boom-template
- boom-template is just a template for gluing an SoC together
- source code lies in riscv-boom
- rocket-chip is a library, provides the uncore

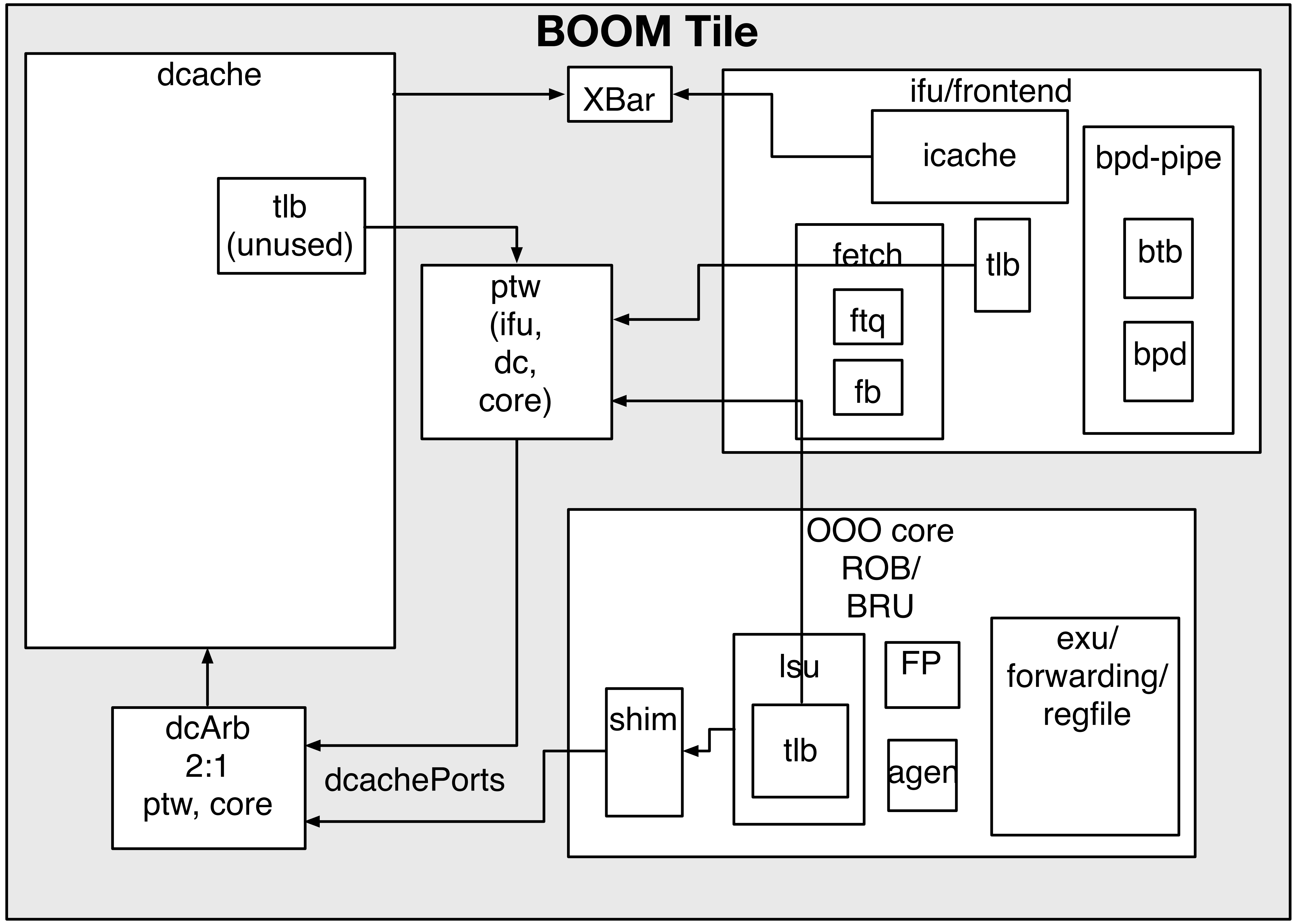
Code Repo Organization



- boom-template is just a template for gluing an SoC together
- fork template for tape-outs

Tile Hierarchy

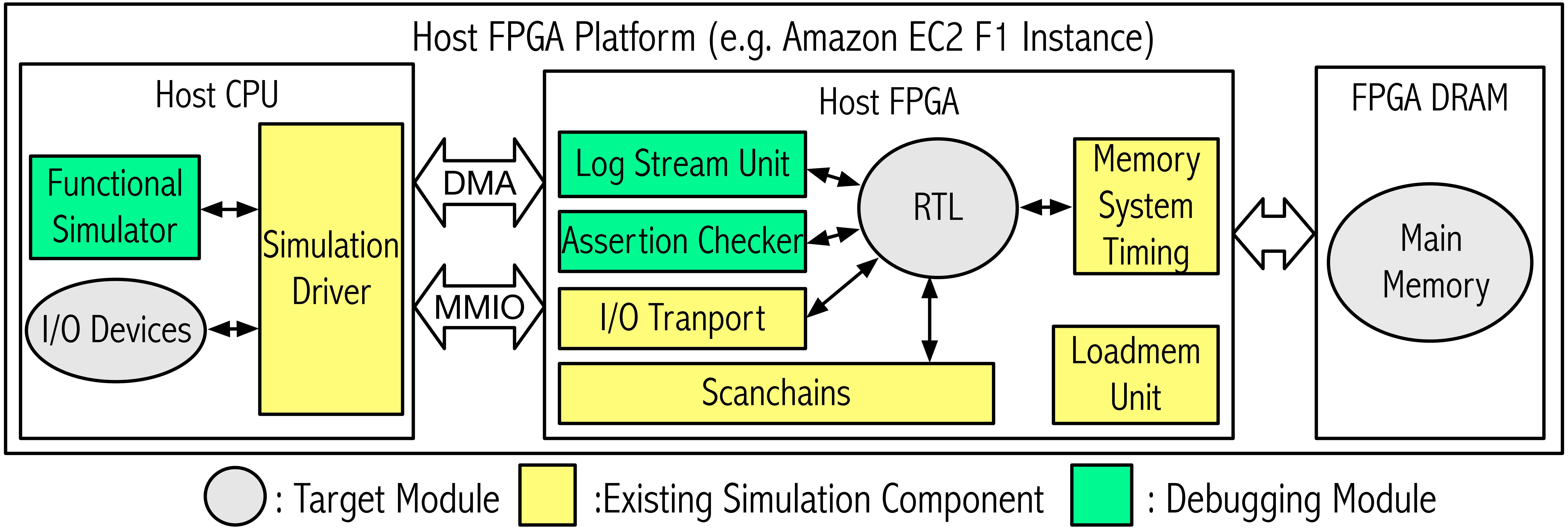
- From rocket-chip:
  - dcache, icache, tlb, ptw, XBar, dcArb
  - caches have been hard forked for BOOM-specific changes (e.g., Spectre-related)
- Historical artifact:
  - OOO core was "BOOM", IFU and DC came from rocket-chip



Questions before the transition?

Extra Slides

DESSERT: Debugging RTL Effectively with State Snapshotting for Error Replays across Trillions of Cycles



- Co-simulate, find bugs, and get waveforms from Cloud FPGA-based simulation!
- Donggyu Kim, et. al. CARRV 2018
- https://carrv.github.io/2018/papers/CARRV_2018_paper_10.pdf

Incorrect Jump Target

- 401.bzip2 (assertion error at 500 billion cycles)
  - JAL jumps to wrong target.
  - Due to improper signed arithmetic.
  - 2-3 year old bug.
  - 3 hours of FPGA time.
  - Would require 39 years of Verilator simulation to find.
  - DESSERT found this via a synthesized assertion.

Incorrect Writeback from FPtoInt

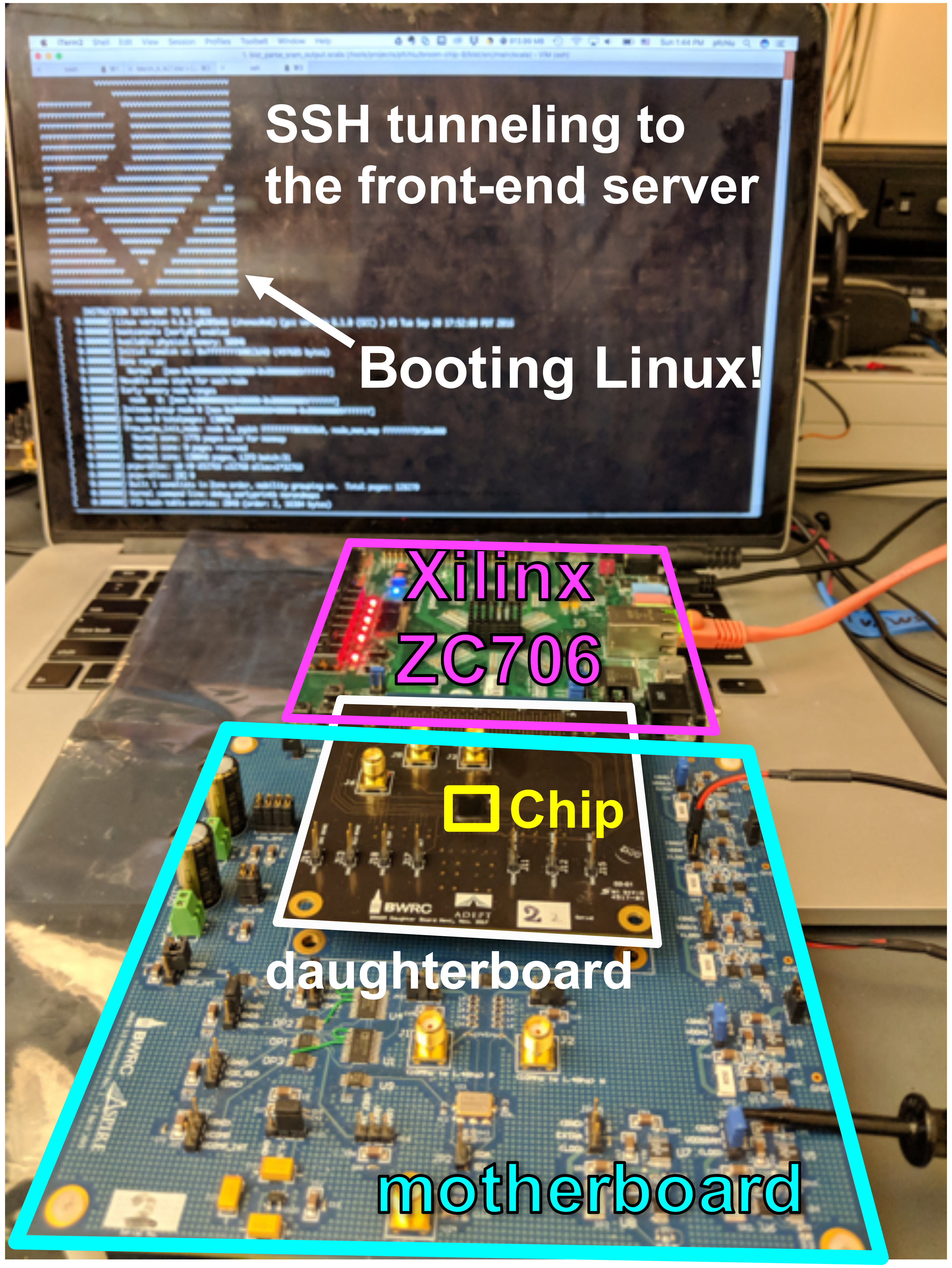
(sometimes)

- 445.gobmk (assertion error at 14.9 billion cycles)
  - misspeculated FPtoInt writes back to invalid ROB entry (after being killed)
  - introduced when splitting regfile into separate integer and fp regfiles
- Problem
  - FPtoInt moves share write port with loads
  - FPtoInt gets buffered in a queue; then gets killed
  - queue then later writes back the value anyways when next FPtoInt instruction comes in
- Cause
  - copy/paste error from a non-speculative flow-through queue
- Found
  - fpga simulation (as opposed to 431 days of verilator)

Load-reserve Didn't Page Fault

- LR is a member of the Atomics Extension
- Thought it was an AMO -- so ignored load page fault signal
- Returned garbage from memory
- 4 year old bug
- But... all LRs are followed by a Store-conditional
- The SC
  - takes the page fault
  - fails to get reservation on the first try
  - forces a retry of the LR
  - The LR succeeds and gets the correct data!

Experimental Setup

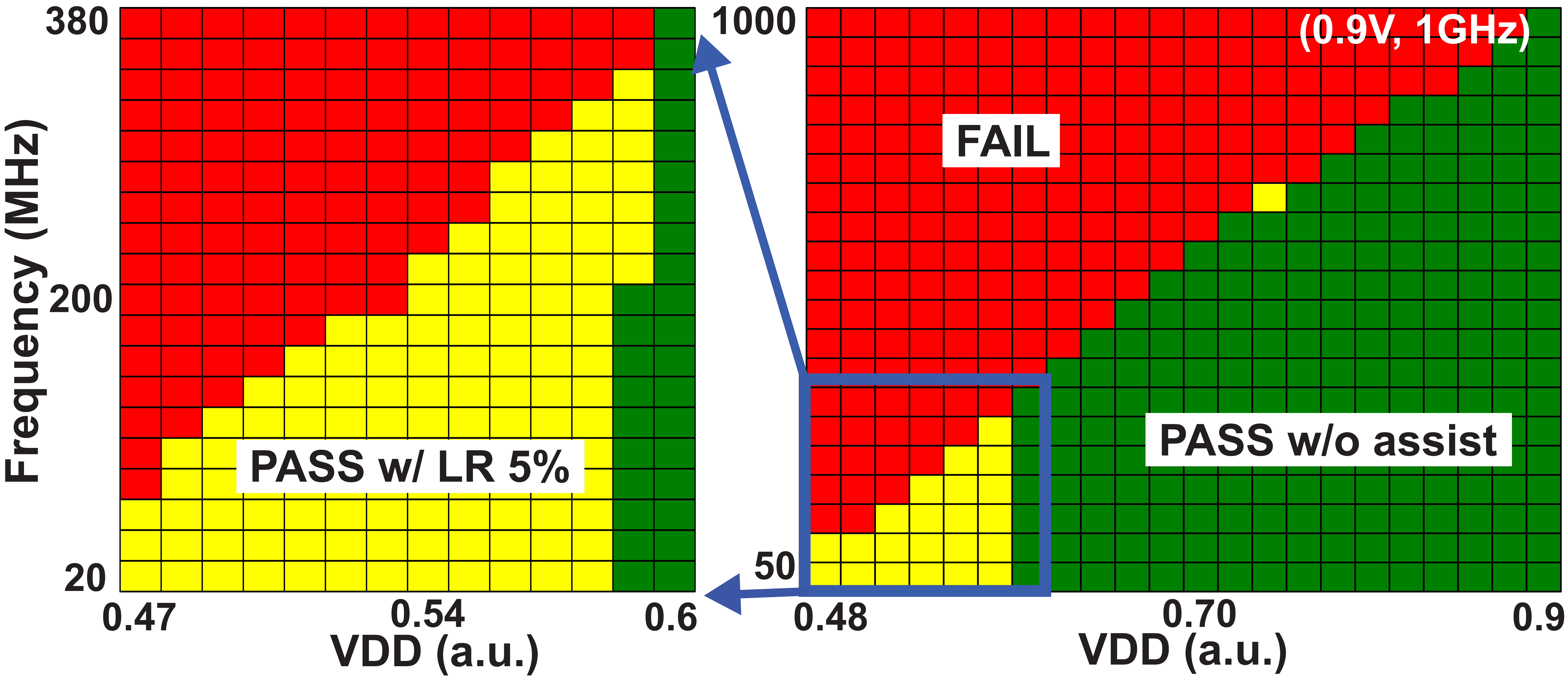


- Chip-on-board (COB) package
- Voltage and clock generation on the motherboard
- Cortex A9 on ZC706 works as the front-end server
- Boot Linux

Performance	
Clock frequency	1GHz @0.9V
	320MHz @0.6V
Coremark/MHz	3.77
Instruction Per Cycle	1.11 (@Coremark)

Operating voltage and frequency

Benchmark: vvadd



- With LR and 5% loss of L2 cache capacity, Vmin is reduced to 0.47V@70MHz
- 2.3% increase in L2 misses, but only 0.2% degradation in IPC

Meltdown

- Nope, I'm fine actually.
- A TLB permission escalation error
  - allows user to speculatively execute on supervisor data
  - speculative execution leaks information
- In BOOM, TLB permission check fails immediately and squashes load-data bypass
  - no speculative user-level execution using privileged data

Spectre

- Uh oh
- In BOOM, BTB is shared across threads.
  - allows attacker to force a victim to execute malicious gadget
  - need to flush BTB on context switch, etc.
- What if attacker thread is the victim thread?
  - Uh oh
- What if the attacker invokes a syscall with untrusted input which leaks speculative information?
  - Uh oh
- Great opportunity for security research!
- See my thoughts on this here:
  - https://content.riscv.org/wp-content/uploads/2018/05/13.00-13.15-Celio-Barcelona-Workshop-8-Talk.pdf