Understanding and working with PULP

Pasquale Davide Schiavone

and the PULP team

[1]Integrated System laboratory, ETH, Zurich, Switzerland

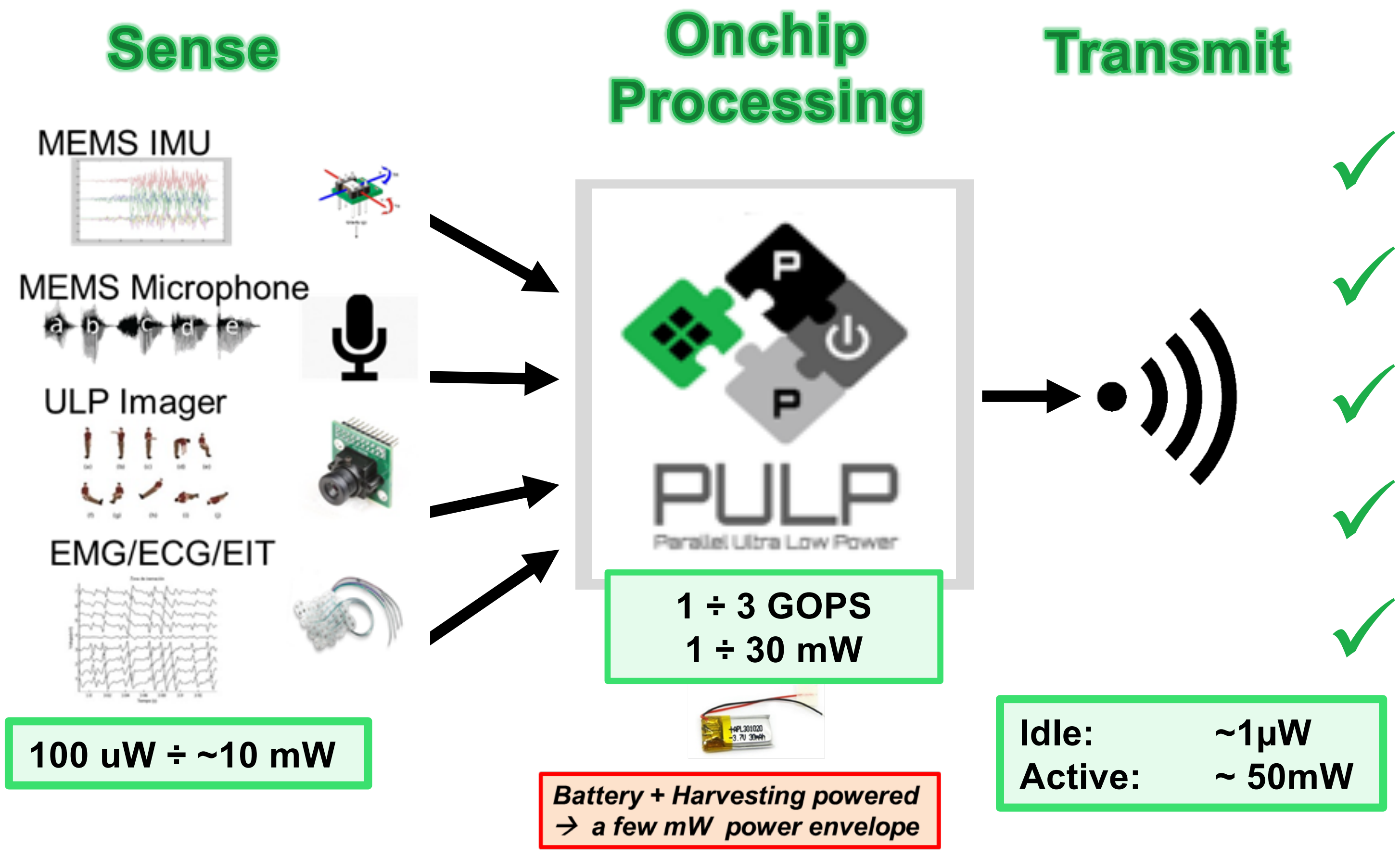
[2]Energy Efficient Embedded Systems Laboratory, University Of Bologna, Bologna, Italy

[1]Department of Electrical, Electronic and Information Engineering

[2]Integrated Systems Laboratory

13.06.2019

Near Sensor (aka Edge) Processing



100 uW ÷ ~10 mW

1 ÷ 3 GOPS
1 ÷ 30 mW

Battery + Harvesting powered
→ a few mW power envelope

Idle: ~1µW
Active: ~ 50mW

✓ Smart Architecture

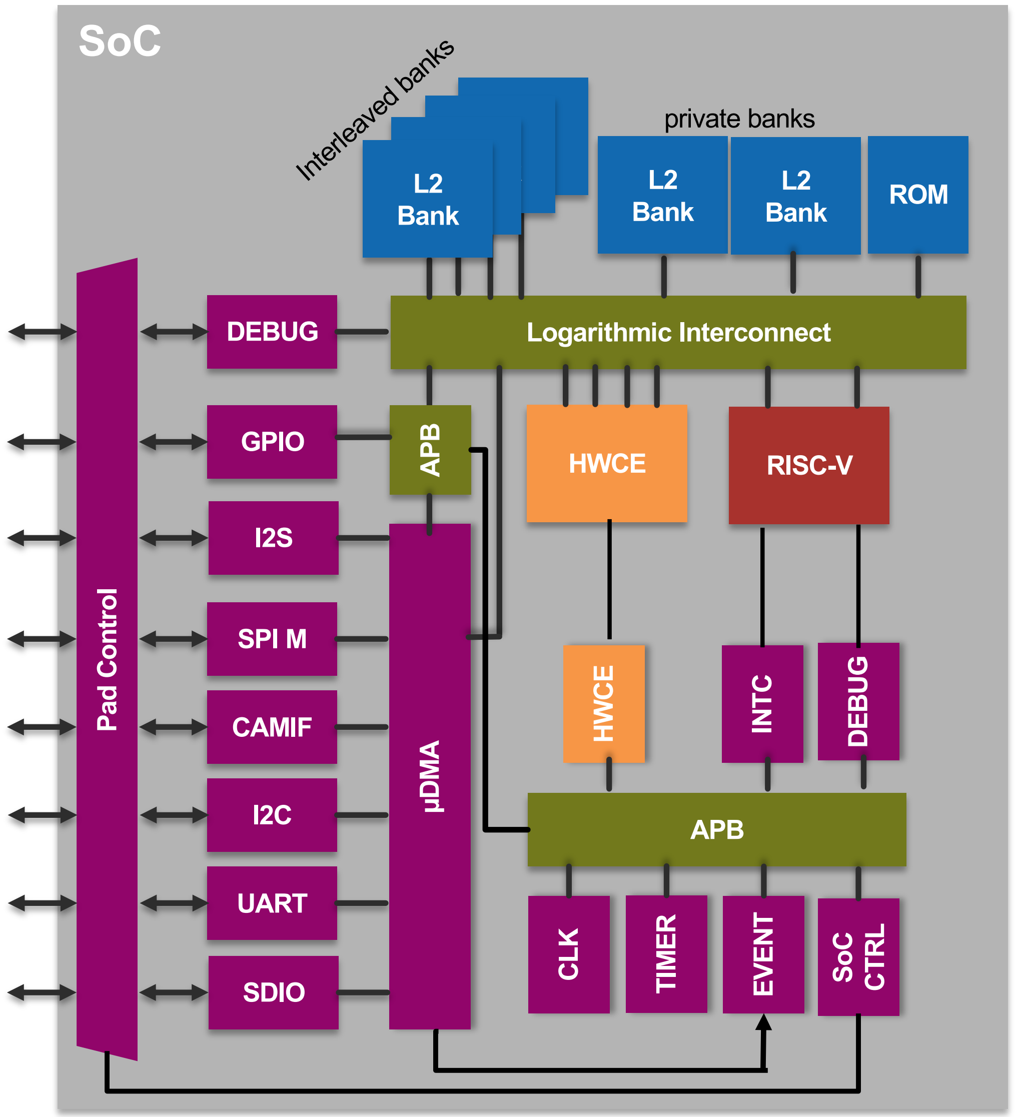
✓ Parallel Processing

✓ Power-saving Design

✓ Near-Threshold

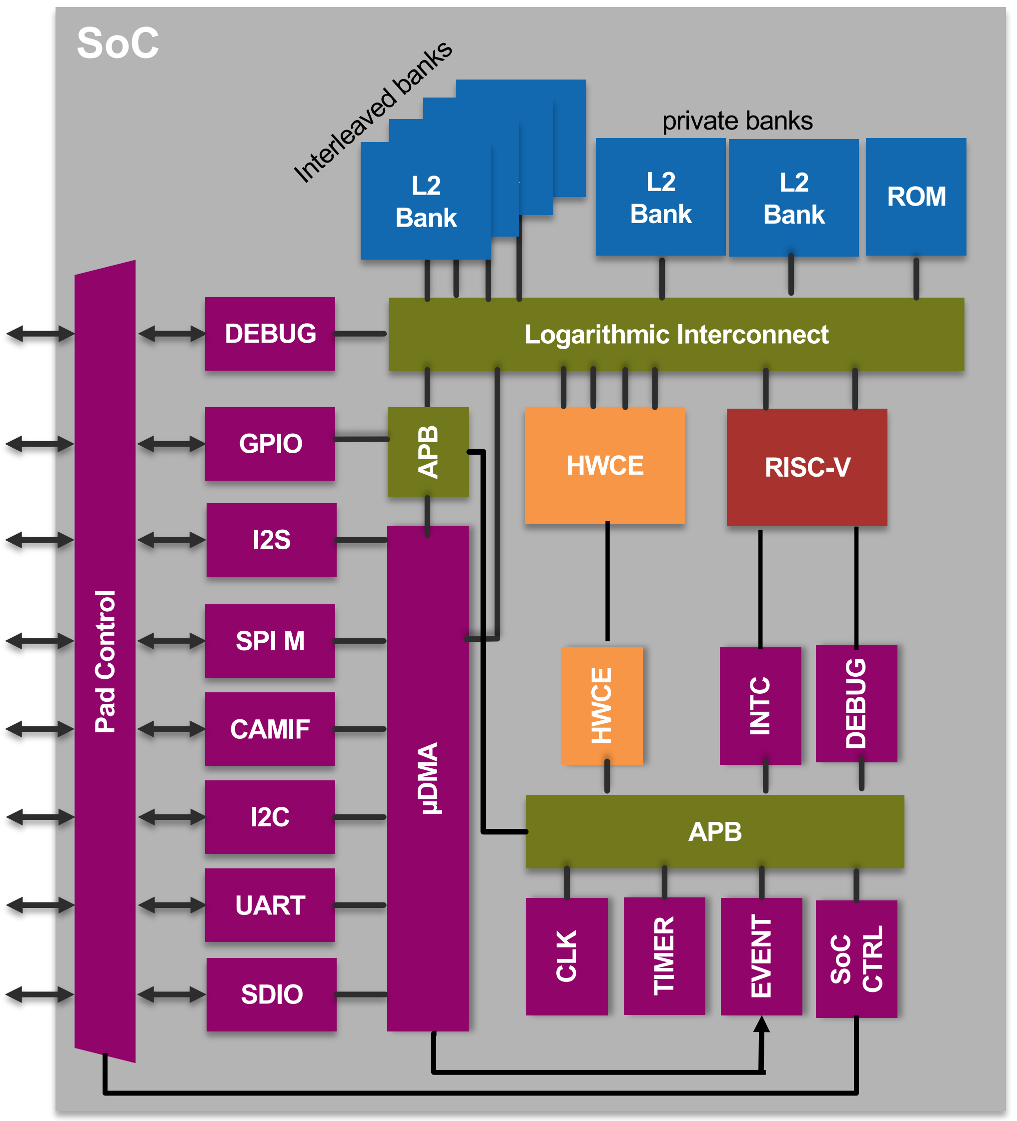
✓ Low Power Technology

PULPissimo Architecture

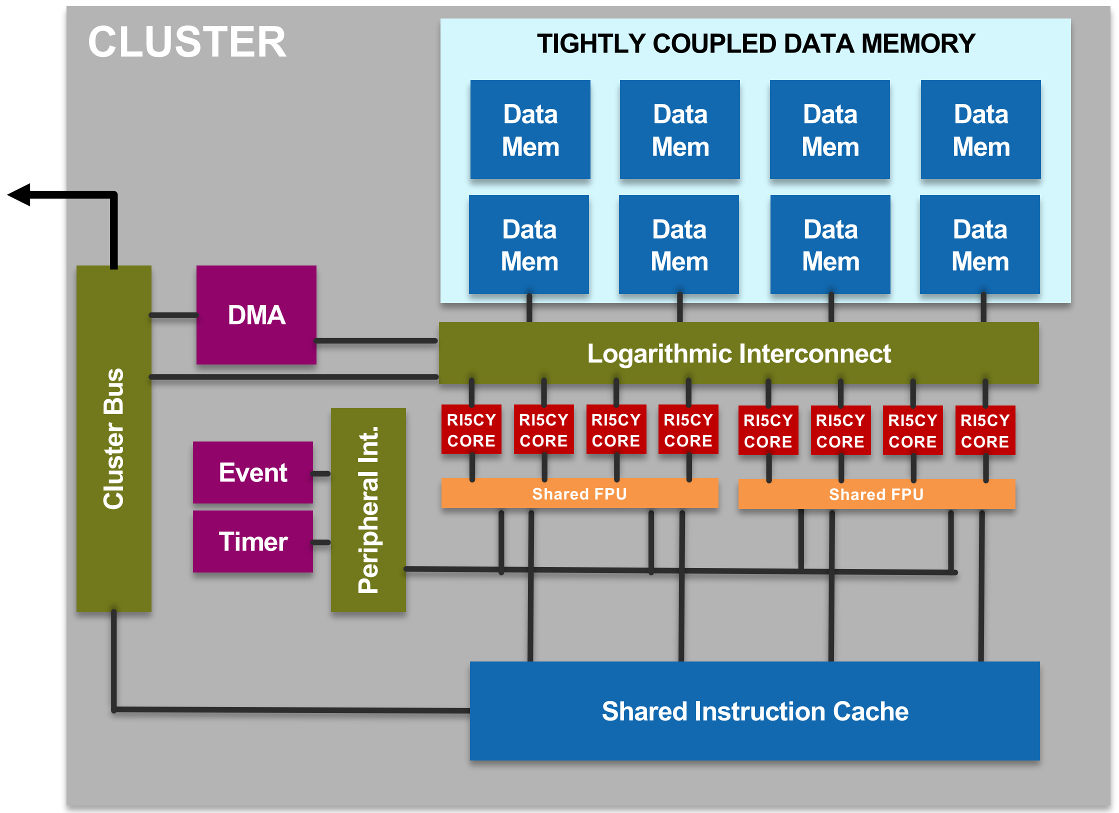


PULPissimo Architecture

- RISC-V based advanced microcontroller
  - 512kB of L2 Memory
  - 16kB of energy efficient latch-based memory (L2 SCM BANK)
- Rich set of peripherals:
  - QSPI (up to 280 Mbps)
  - Camera Interface (up to 320x240@60fps)
  - I2C, I2S (up to 4 digital microphones)
  - JTAG (Debug), GPIOs,
  - Interrupt controller, Bootup ROM
- Autonomous IO DMA Subsystem (µDMA)
- Power management
  - 2 low-power FLLs (IO, SoC)

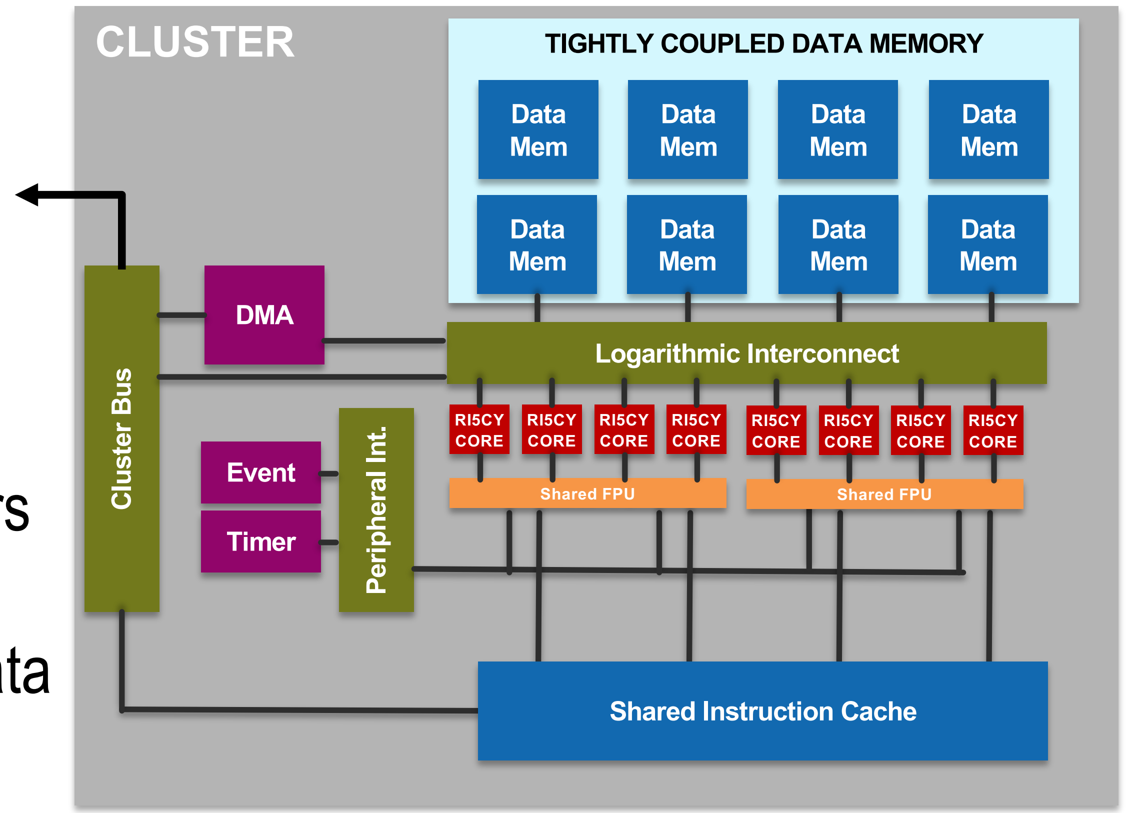


PULP Cluster Architecture



PULP Cluster Architecture

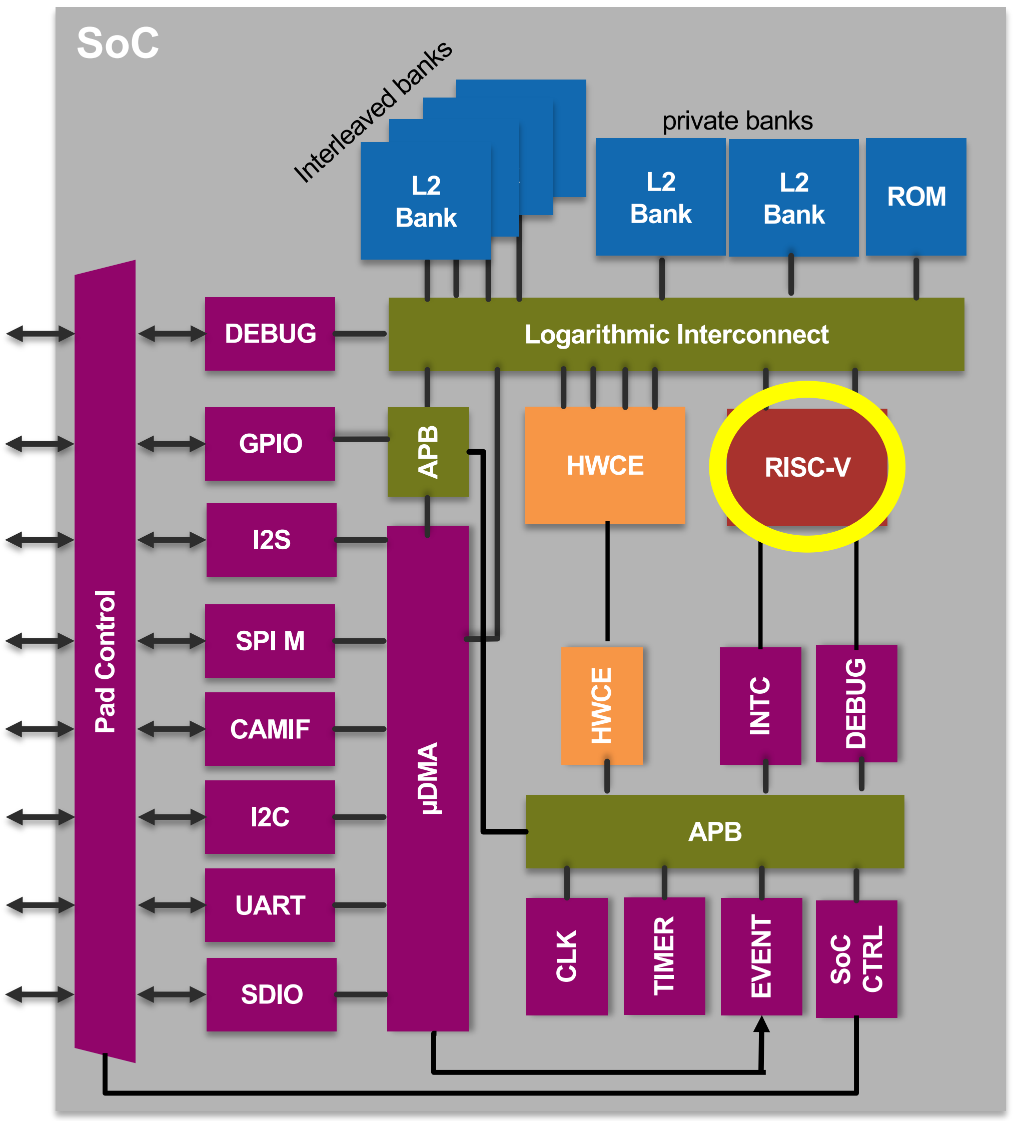
- 8 RISC-V multicore cluster
  - 64kB of L1 Memory
- Shared FPU for efficient resources minimization
  - 2 FPU, 1 every 4 cores
- Shared I$
  - Optimize cache usage
- Multi-Core event unit for barriers and clock-gate managment
- DMA for efficient L2←→ L1 data transfers



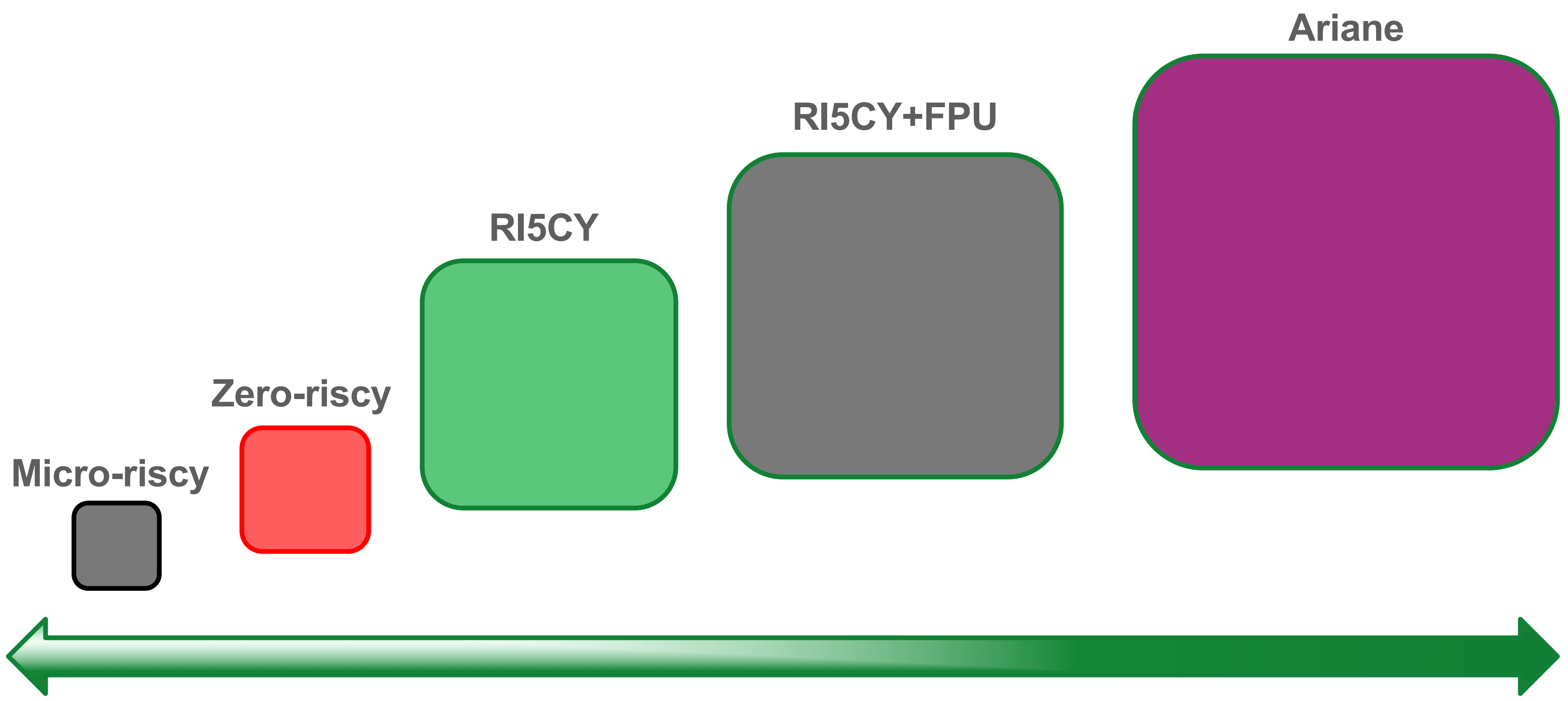
The RISC-V PULP cores

PULPissimo Architecture

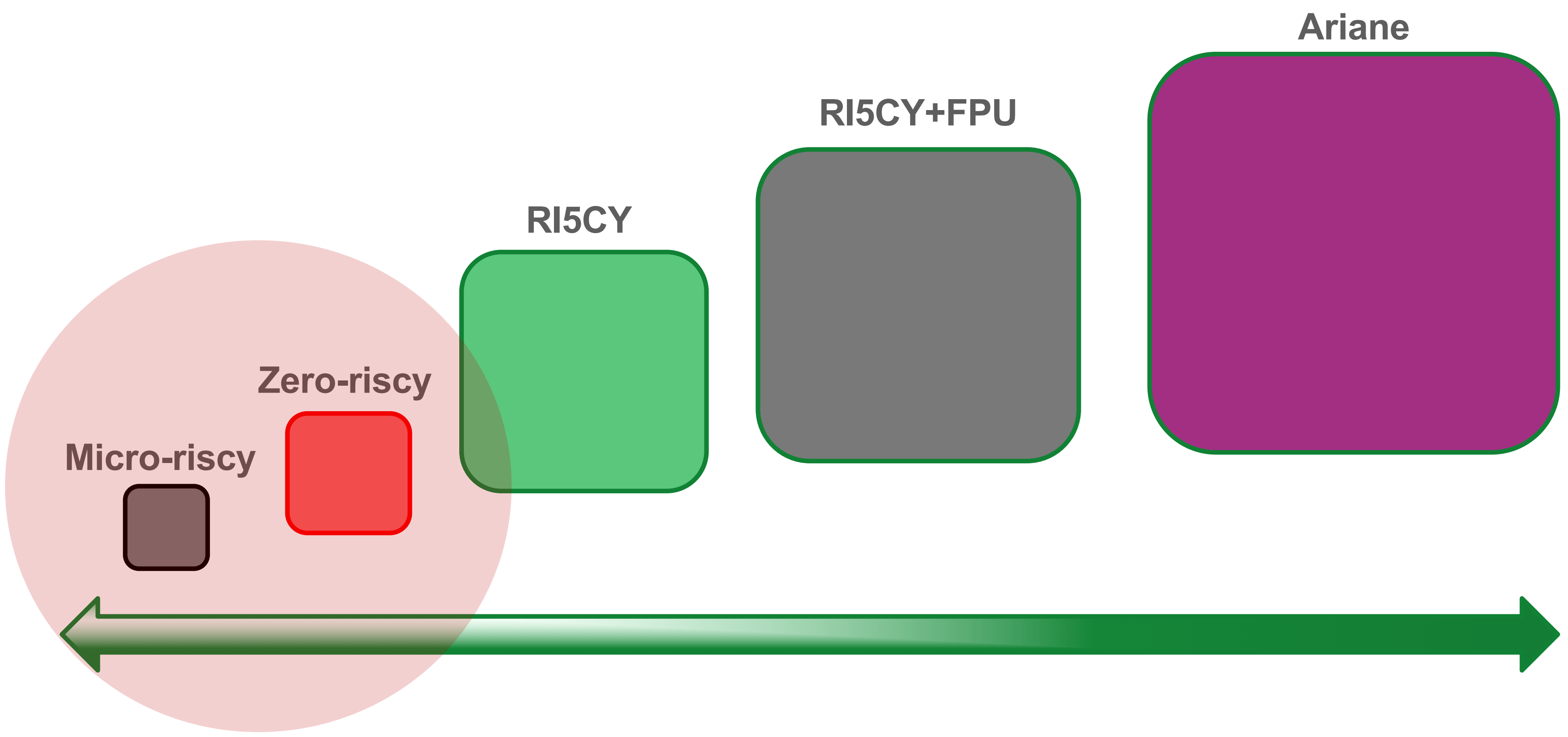
- RISC-V based advanced microcontroller
  - 512kB of L2 Memory
  - 16kB of energy efficient latch-based memory (L2 SCM BANK)
- Rich set of peripherals:
  - QSPI (up to 280 Mbps)
  - Camera Interface (up to 320x240@60fps)
  - I2C, I2S (up to 4 digital microphones)
  - JTAG (Debug), GPIOs,
  - Interrupt controller, Bootup ROM
- Autonomous IO DMA Subsystem (µDMA)
- Power management
  - 2 low-power FLLs (IO, SoC)



Different Workload? Different core

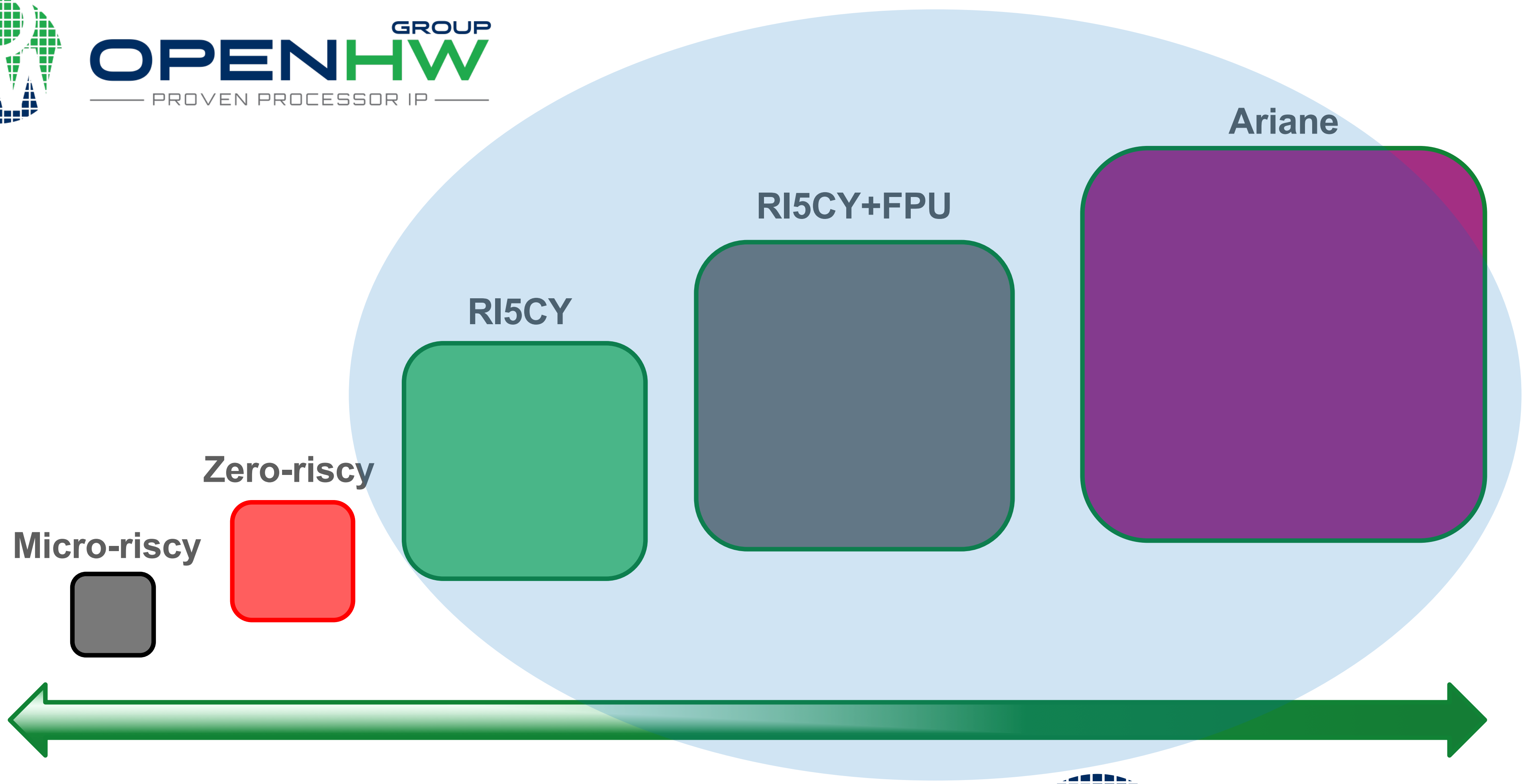


Different Workload? Different core



Now part of lowRISC *under the «IBEX» name*

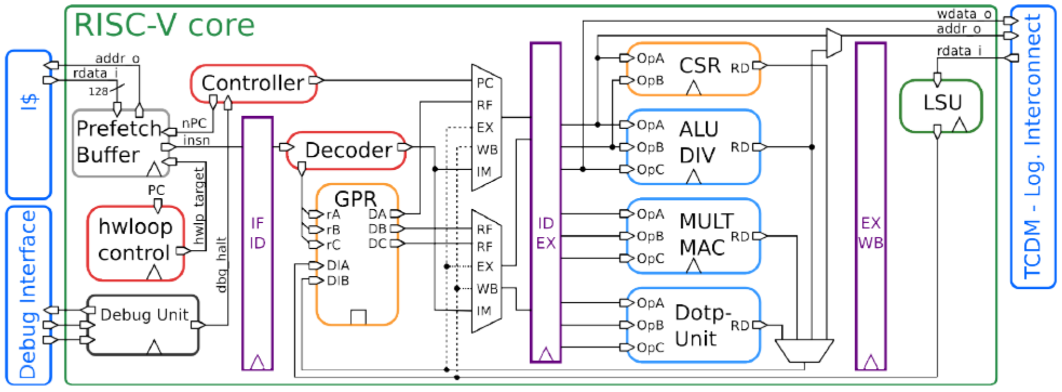
Different Workload? Different core



Going to be part of CORE-V™ *IP core family*

RI5CY Processor: our workhorse core

- 4-stage pipeline
  - RV32IMFCXpulp
  - 70K GF22 nand2 equivalent gate (GE) + 30KGE for FPU
  - Coremark/MHz 3.19
  - Includes various extensions
    - pSIMD
    - Fixed point
    - Bit manipulations
    - HW loops
- Silicon Proven
  - SMIC130, UMC65, TSMC55LP, TSMC40LP, GF22FDX

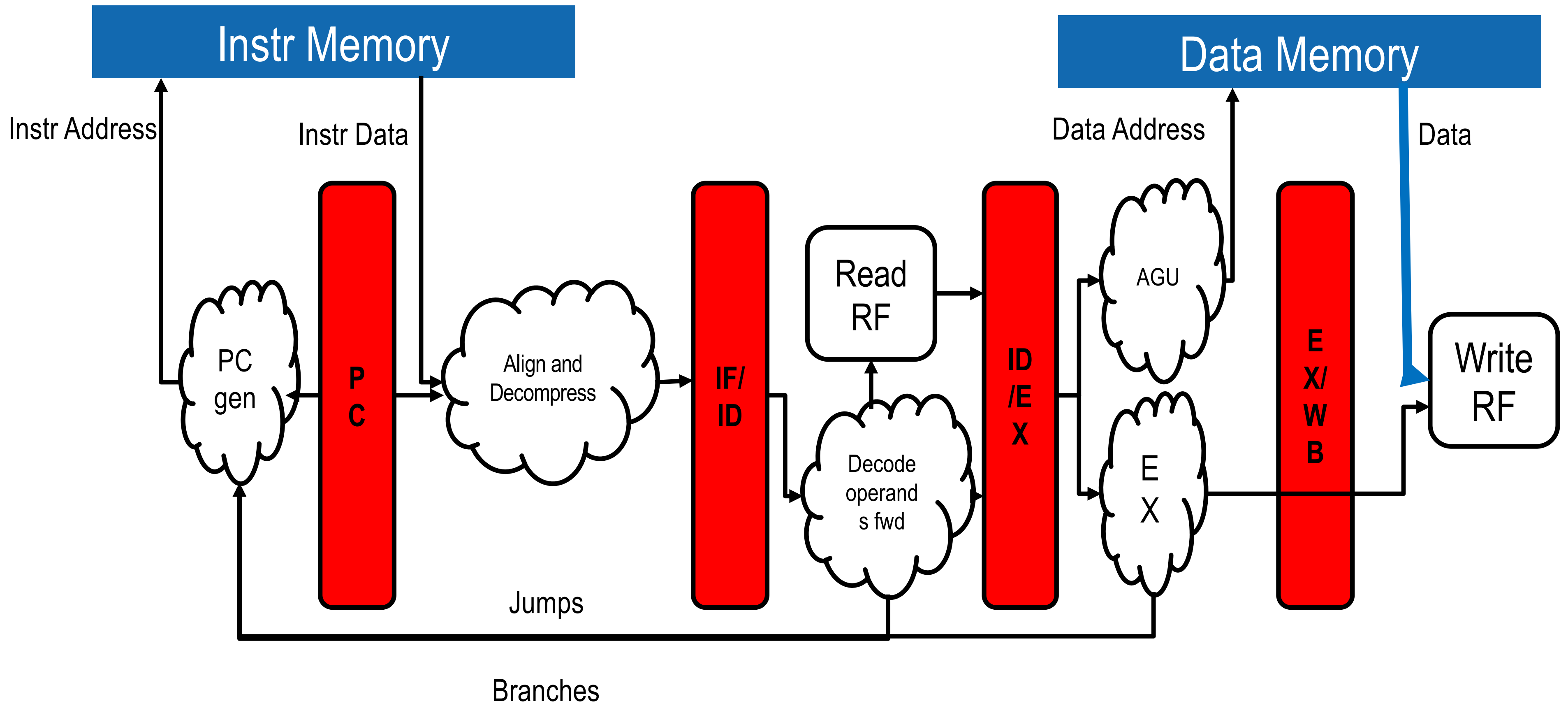


https://github.com/pulp-platform/riscv

- **NEW** Floating Point Unit:
  - Iterative DIV/SQRT (9 cycles)
  - Parametrizable latency for MUL, ADD, SUB, Cast
  - Single cycle load, store

PULP

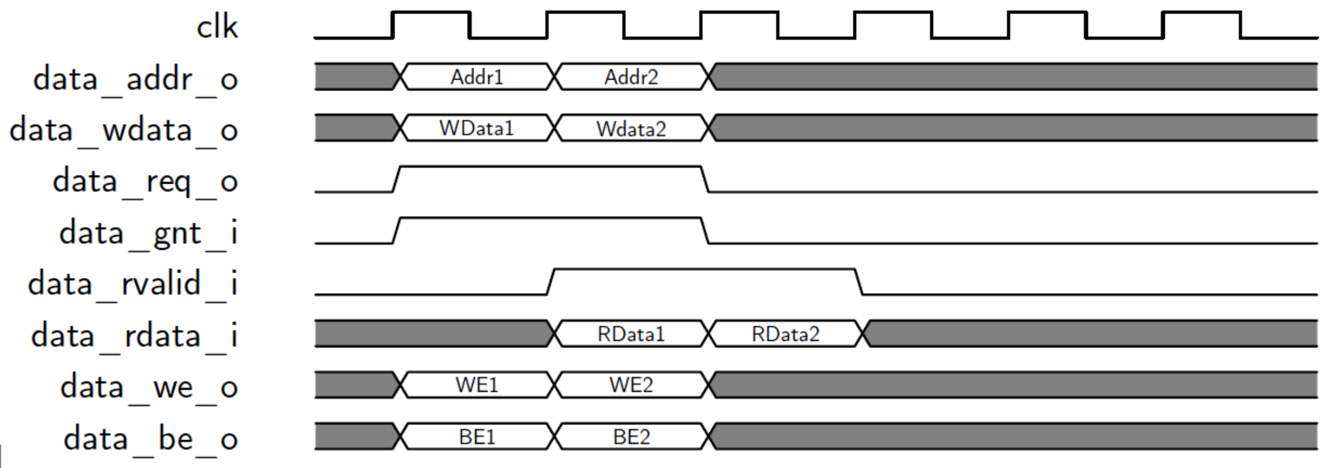
RI5CY simplified pipeline



PULP Cores Memory Interface (1/2)

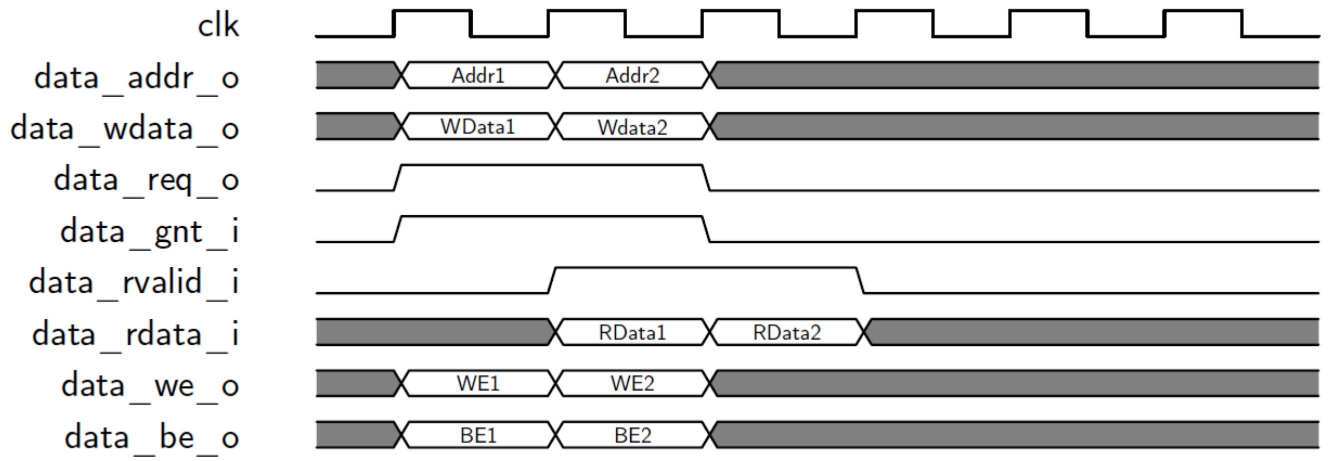
- Request with Address (32bits) and request (1bit) signal
  - Byte Enable (BE) (4bits): byte, short or word memory transaction) in case of Load/Store
  - Write Enable (WE) (1 bit)
  - wdata (32bits): data to write in case of store operations

- Response with Grant signal and Valid signal
  - The core can be interfaced with multicycle memory accesses
    - Grant comes from the arbiter
    - Valid from the memory subsystem
  - rdata (32bits): data to read. It has to be sampled when the valid signal is high

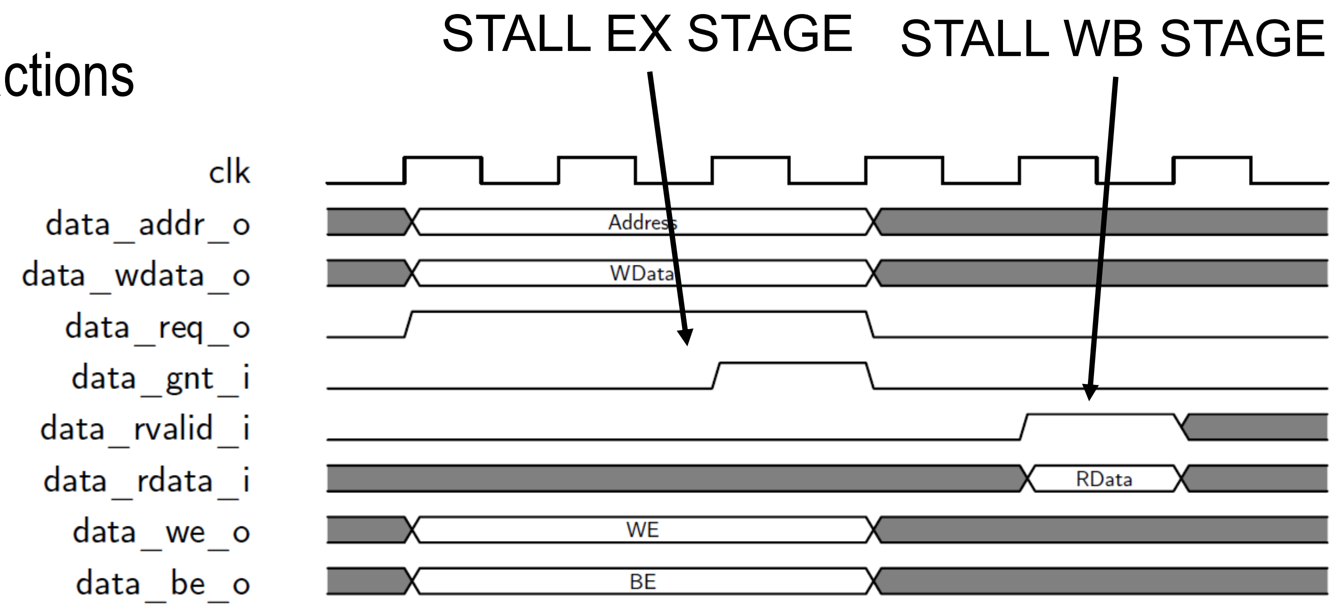


PULP Cores Memory Interface (2/2)

- Back2Back Memory Transactions



- Slow Memory Transactions



Xpulp Extentions: General Purposes Extensions 1

- Memory Access Extensions
  - Misaligned memory accesses (not ISA extension)
    - Load or Store 32/16bit values with non-multiple of 4/2 addresses
    - Useful when dealing with packet-data (32bits values holding 2/4 elements)
    - It requires 2 access to the memory, data manipulation done in the load-store-unit
      - e.g. LOAD 32bit at 0x0000_0002
        - Read from memory higher 16bits at 0x0000_0000
        - Read from memory lower 16bits at 0x0000_0004 and pack the data
    - Save instructions (code size) and speed up execution
      - Explicit load to 0x0000_0000 and 0x0000_0004, shift and or operations

Original RISC-V

```
lw x2, 0(x10)
lw x3, 4(x10)
slri x2, x2, 16
slli x3, x3, 16
or  x3, x3, x2
```

Misaligned Ext

```
lw x2, 2(x10)
```

Xpulp Extentions: General Purposes Extensions 2

- Memory Access Extensions
  - Post Increment Load/Store
    - Automatic register update with computed address
    - Useful in array iterations
    - Save instructions
    - It requires extra write register file port or slower execution
  - Register-Register Load/Store (and Post Increment)
    - Immediate is only 12bits
    - Use register-register address calculation for 32bits offset

Original RISC-V

```
lw x2, 4(x10)
lw x3, 4(x12)
addi x10, x10, 4
addi x12, x12, 4
....
LOOP
```

AutoIncrement Load/Store Ext

```
lw x2, 4(x10!)
lw x3, 4(x12!)
...
LOOP
```

Xpulp Extentions: General Purposes Extensions 3

- Hardware loops extensions
  - HWLs or Zero Overhead Loops to remove branch overheads in for loops.
    - Smaller loop benefit more!
  - Loop needs to be set up beforehand and is fully defined 3 SP regs by:
    - Start address → lp.starti L, Imm12 → START_REG[L] = PC + 2*Imm12
    - End address → lp.endi L, Imm12 → END_REG[L] = PC + 2*Imm12
    - Counter → lp.count{-,i}, L, {rs1,Imm12} → COUNT_REG[L] = rs1/Imm12
    - Short-cut → lp.setup{-,i}, L, {rs1,ImmC}, Imm12
      - START_REG[L] = PC + 4, END_REG[L] = PC + 2*Imm12, COUNT_REG[L] = {rs1,ImmC}
  - Two sets registers implemented to support nested loops (L=0 or 1)
  - Performance:
    - Speedup can be up to factor 2!

Original RISC-V

```
mv   x5, 0
mv   x4, 100
Lstart:
  addi  x4, x4, -1
  nop
  nop
bne  x4, x5, Lstart
```

HW Loop Ext

```
lp.setupi 100, Lend
nop
Lend: nop
```

Xpulp Extentions: Bit Manipulation

- Bit manipulation extensions
  - RISC-V reserved the “RVB” extensions but it is still an on-going topic
  - PULP developed its own bit-manipulation and possibly will align with RVB
    - Contribution to the official task in the RISC-V community
  - Bit Manipulation instructions list
    - Extract N bits starting from M from a word and extend (or not) with sign
    - Insert N bits starting from M in a word
    - Clear N bits starting from M in a word
    - Set N bits starting from M in a word
    - Find first bit set
    - Find last bit set
    - Count numbers of 1 (popcount)
    - Rotate

Original RISC-V

```
mv   x5, 0
mv   x7, 0
mv   x4, 32
Lstart:
  andi x6, x8, 1
  add x7, x7, x6
  addi  x4, x4, 1
  slri x8, x8, 1
bne  x4, x5, Lstart
```

BitMan Ext

```
p.cnt x8, x8
```

Xpulp Extentions: DSP

- DSP extensions
  - General purposes
    - ABS, CLIP/Saturation
    - MIN, MAX
    - MAC and MSU
  - Fixed Point Support
    - ADD and SUB with normalization and round
    - MUL and MAC with normalization and round
  - Possibility to share some resources
    - ABS reuses the adder and comparator in the ALU
    - Clip adds a comparator but reuses adder and previous comparator
    - Normalization done by connecting adder output to the shifter
    - Round done by exploiting multi-operand adders

Original RISC-V

```
add  x4, x4, x5
addi x4, x4, 1
slri  x4, x4, 1
```

DSP Ext

```
p.addRN x4, x5, x5, 1
```

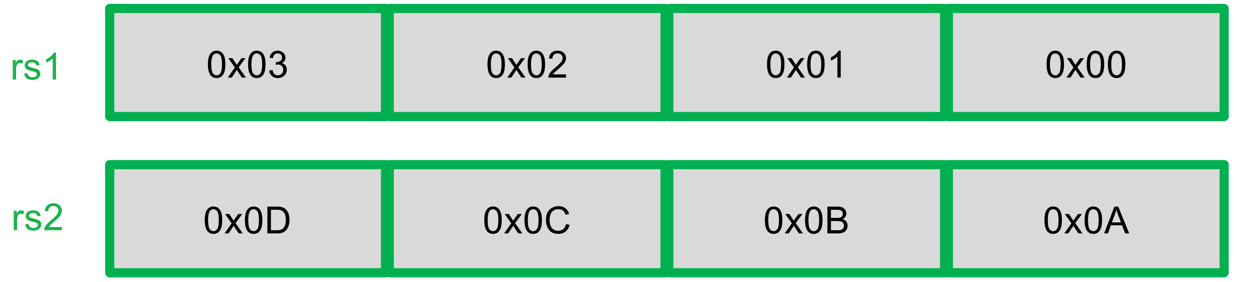
Xpulp Extentions: packed-SIMD 1/4

- packed-SIMD extensions
  - RISC-V reserved the “RVP” extensions but it is still an on-going topic
    - It also includes DSP extensions
  - Differently from “RVV” vectorial extensions, vectors are packet to the integer RF
    - Make usage of resources the best in performance with little overhead
    - Target for embedded systems, RVV is for high performance
  - pSIMD in 32bit machines
    - Vectors are either 4 8bits-elements or 2 16bits-elements
  - pSIMD instructions

Computation	add, sub, shift, avg, abs, dot product
Compare	min, max, compare
Manipulate	extract, pack, shuffle

Xpulp Extentions: packed-SIMD 2/4

- Same Register-file
  - The instruction encode how to interpret the content of the register



add rD, rs1, rs2	rD = 0x03020100 + 0x0D0C0B0A

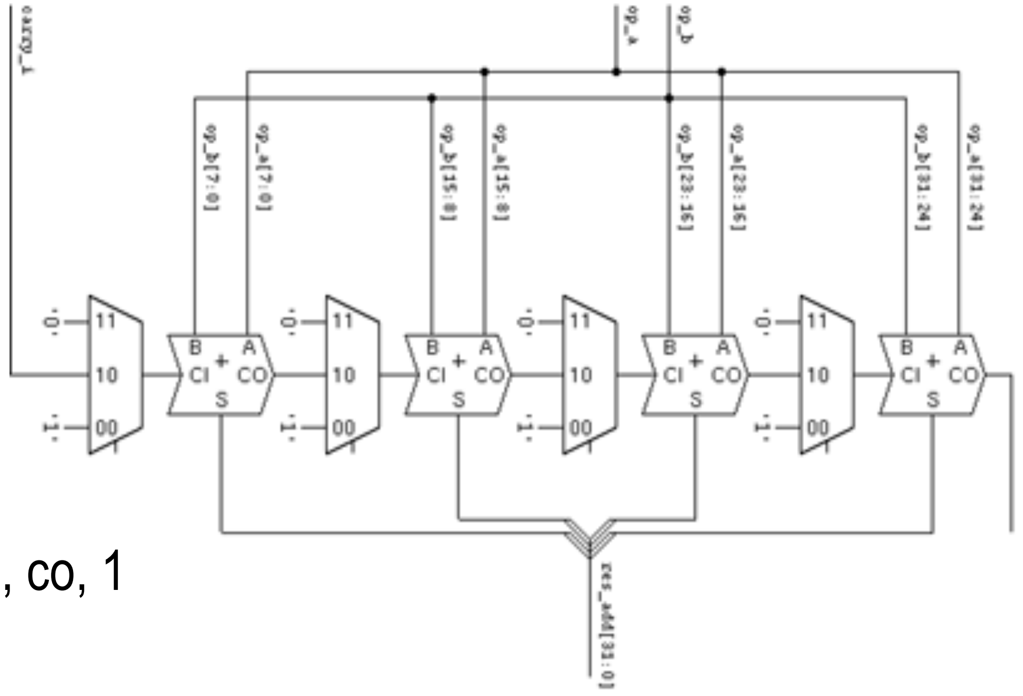
add.h rD, rs1, rs2	rD[0] = 0x0100 + 0x0B0A
	rD[1] = 0x0302 + 0x0D0C

add.b rD, rs1, rs2	rD[0] = 0x00 + 0x0A
	rD[1] = 0x01 + 0x0B
	rD[2] = 0x02 + 0x0C
	rD[3] = 0x03 + 0x0D

Xpulp Extentions: packed-SIMD 3/4

- HW reuse for small overhead
- Vector modes:
  - bytes, halfwords, word
  - 4 byte operations
    - With byte select
  - 2 halfword operations
    - With halfword select
  - 1 word operation
- Play with carry chain
  - 32bit adder → 35bit adder
  - Vector halfword sub → Carry = co, 1, co, 1
  - Vector byte sub → Carry = 1, 1, 1, 1
  - word sub → Carry = co, co, co, 1

Vectorial Adder



Xpulp Extentions: packed-SIMD 4/4

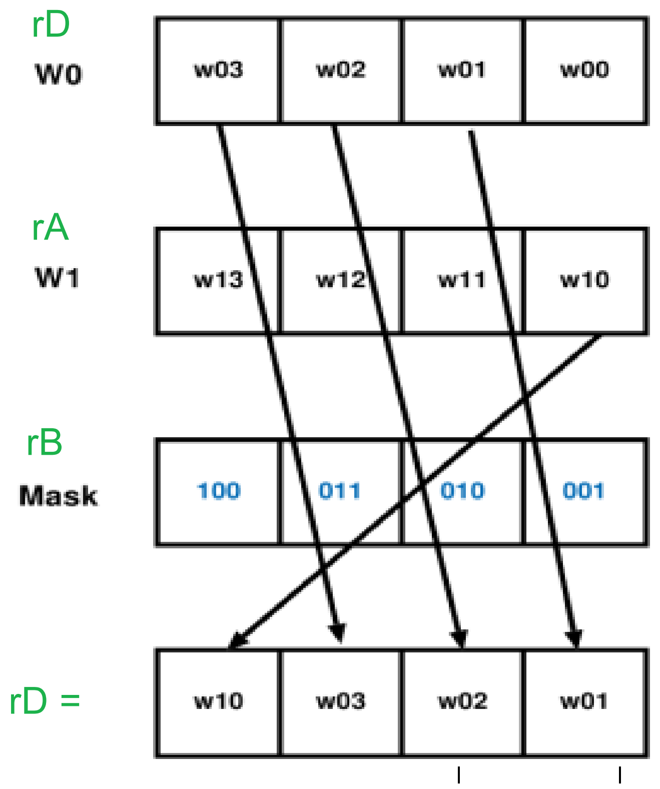
- Shuffle instructions
- In order to use the vector unit the elements have to be aligned in the register file
- Shuffle allows to recombine bytes into 1 register

- pv.shuffle2.b rD, rA, rB

Mask bits

```
rD{3} = (rB[26]==0) ? rA:rD  {rB[25:24]}
rD{2} = (rB[18]==0) ? rA:rD  {rB[17:16]}
rD{1} = (rB[10]==0) ? rA:rD  {rB[ 9: 8]}
rD{0} = (rB[ 2]==0) ? rA:rD  {rB[ 1: 0]}
```

- With rX{i} = rX[(i+1)*8-1:i*8]



ISA Extensions: Putting it All Together

```
for (i = 0; i < 100; i++)
   d[i] = a[i] + b[i];
```

Baseline

```
mv   x5, 0
mv   x4, 100
Lstart:
  lb     x2, 0(x10)
  lb     x3, 0(x11)
  addi   x10,x10, 1
  addi   x11,x11, 1
  add    x2, x3, x2
  sb     x2, 0(x12)
  addi   x4, x4, -1
  addi   x12,x12, 1
bne      x4, x5, Lstart
```

Auto-incr load/store

```
mv   x5, 0
mv   x4, 100
Lstart:
  lb    x2, 1(x10!)
  lb    x3, 1(x11!)
  addi x4, x4, -1
  add   x2, x3, x2
  sb    x2, 1(x12!)
bne     x4, x5, Lstart
```

HW Loop

```
lp.setupi 100, Lend
  lb    x2, 1(x10!)
  lb    x3, 1(x11!)
  add   x2, x3, x2
Lend:   sb x2, 1(x12!)
```

Packed-SIMD

```
lp.setupi 25, Lend
  lw   x2, 4(x10!)
  lw   x3, 4(x11!)
  pv.add.b x2, x3, x2
Lend: sw x2, 4(x12!)
```

11 cycles/output

8 cycles/output

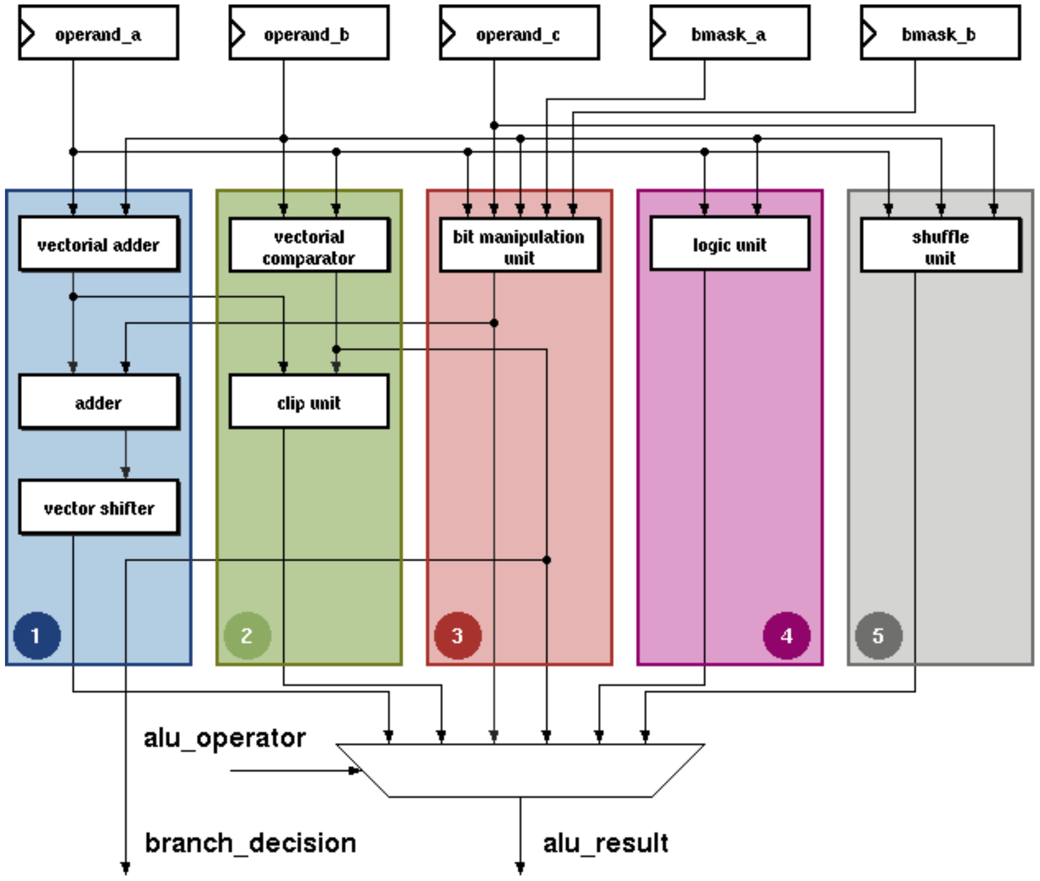
5 cycles/output

1,25 cycles/output

PULP

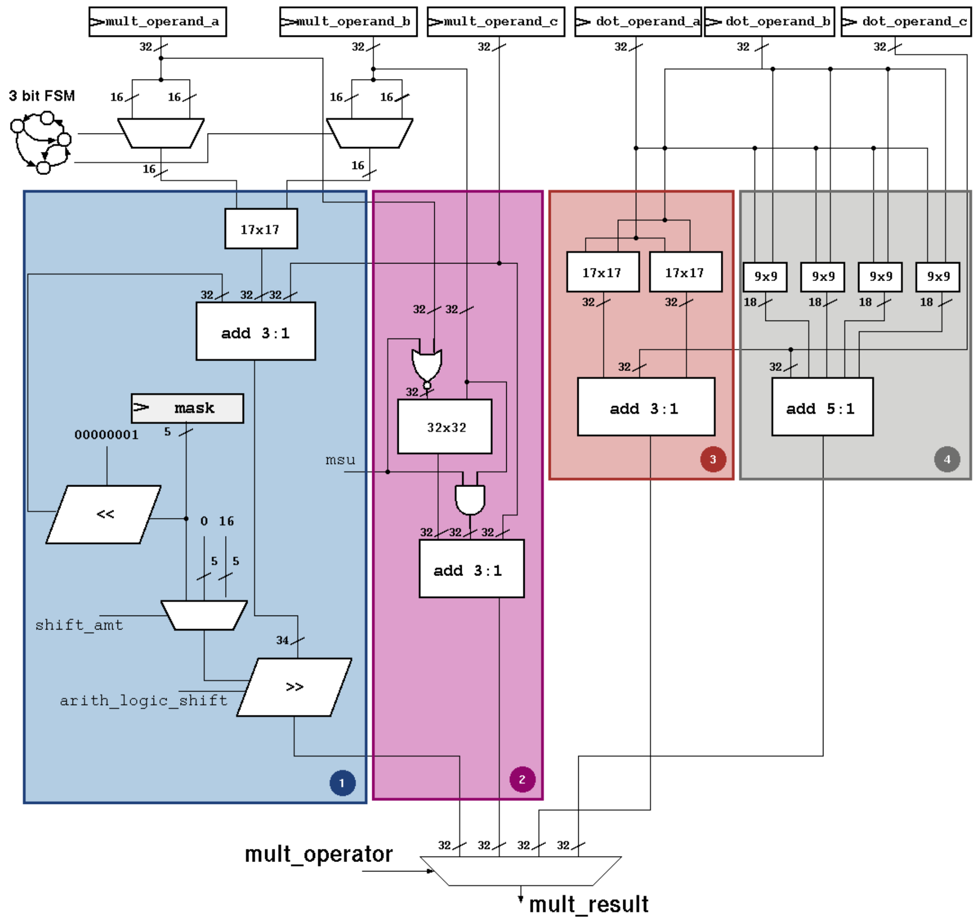
ALU architecture

- Advanced ALU for Xpulp extensions
- Optimized datapath to reduce resources
- Multiple-adders for round
- Adder followed by shifter for fixed point normalization
- Clip unit uses one adder as comparator and the main comparator



MUL architecture

- (blue) 16x16 with sign selection for short multiplications [with round and normalization]. 5 cycles FSM for higher 64-bits (mulh* instructions)
- (purple) One single cycle mac unit that performs MAC, MSU and MUL
- (red) short parallel dot product
- (grey) byte parallel dot product
- Clock gating to reduce switching activity between the integer and SIMD multiplier



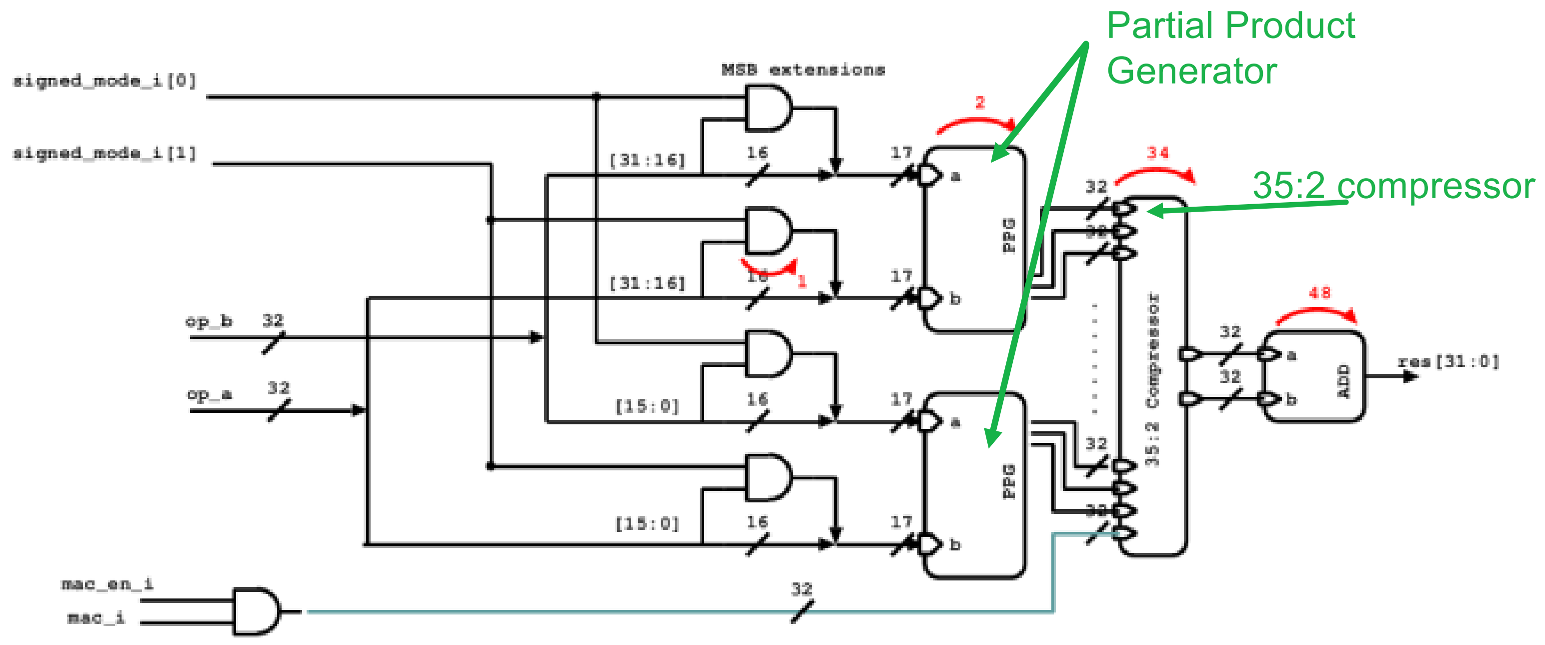
PULP

Dot Product Multiplier

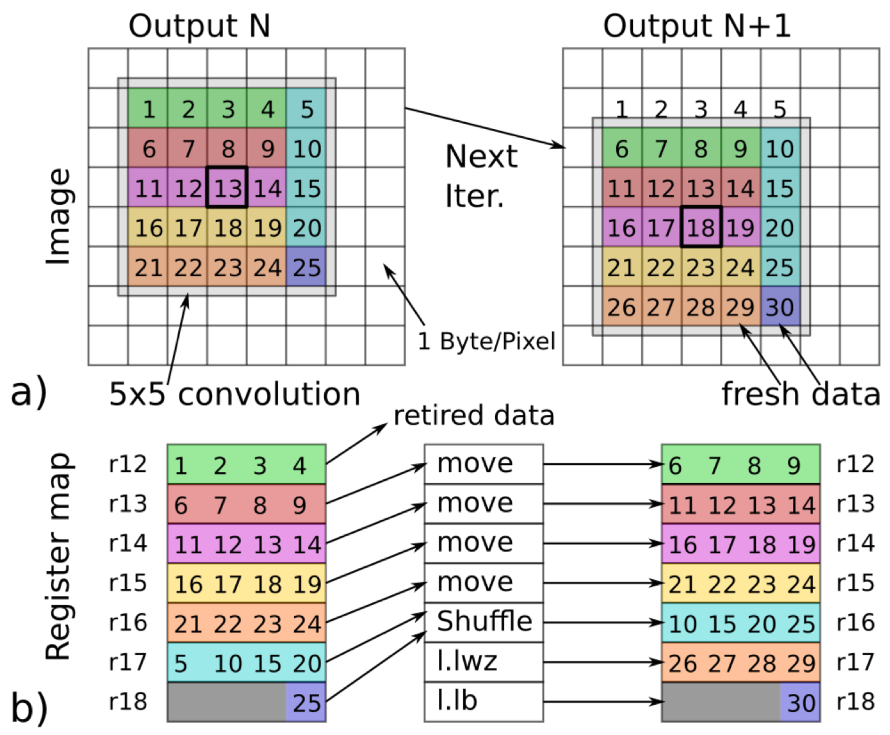
- Dot Product: (half word example)

$$C[31:0] = \underbrace{A[31:16]*B[31:16]}_{\text{32 bit}} + \underbrace{A[15:0]*B[15:0]}_{\text{32 bit}} + \underbrace{C[31:0]}_{\text{32 bit}}$$

=> 2 multiplications, 1 addition, 1 accumulation in 1 cycle!

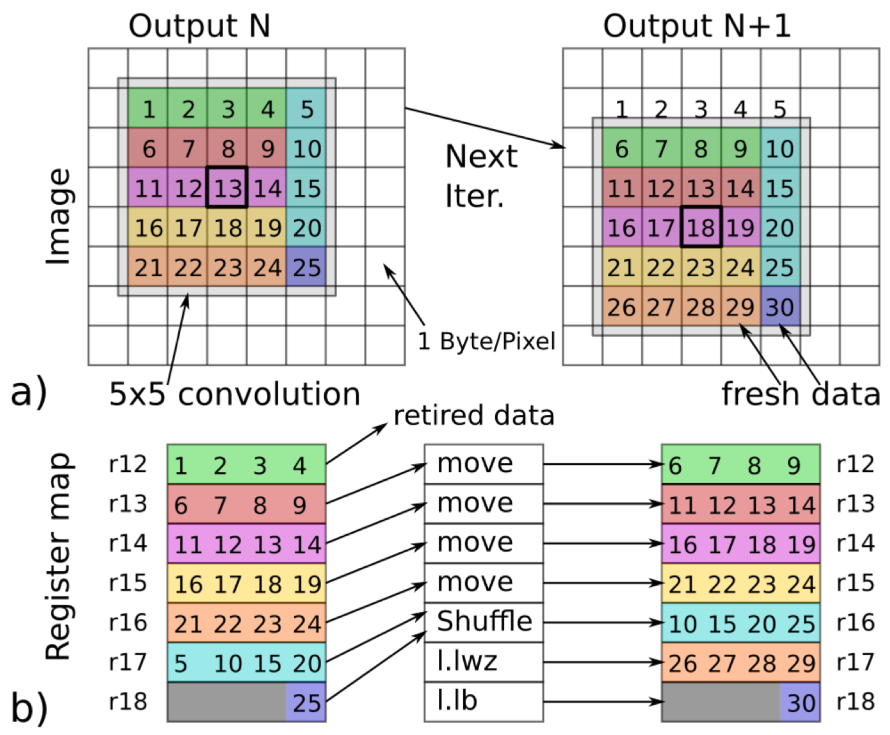


2D Convolution with Xpulp Extensions: performance + less memory pressure



- Convolution in registers
- 5x5 convolutional filter

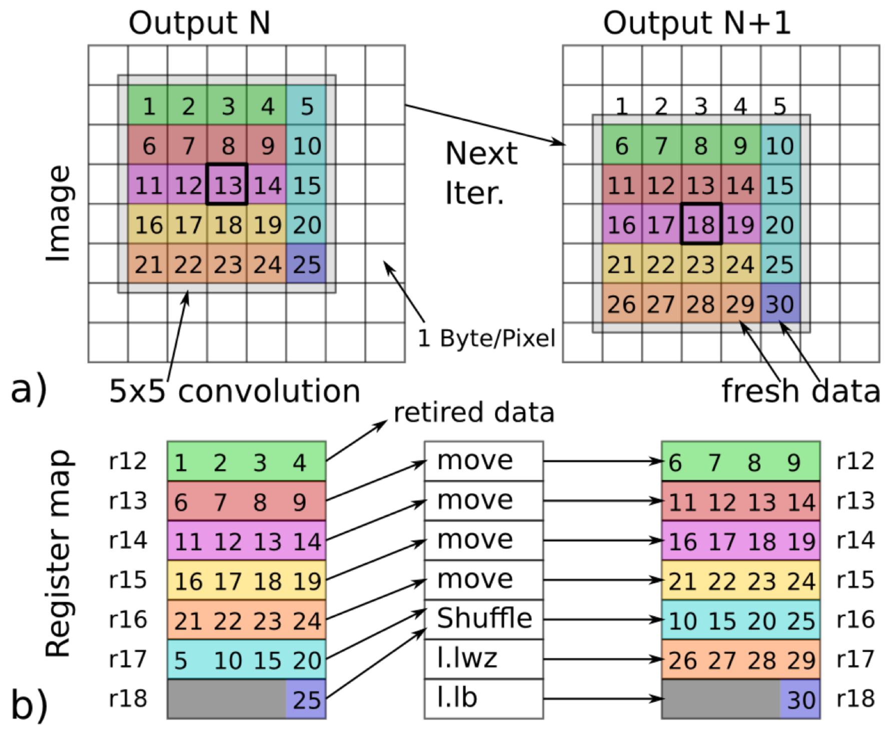
2D Convolution with Xpulp Extensions: performance + less memory pressure



- Convolution in registers
- 5x5 convolutional filter

- 7 Sum-of-dot-product
- 4 move
- 1 shuffle
- 3 lw/sw
- ~ 5 control instructions

2D Convolution with Xpulp Extensions: performance + less memory pressure



- Convolution in registers
- 5x5 convolutional filter

- 7 Sum-of-dot-product
- 4 move
- 1 shuffle
- 3 lw/sw
- ~ 5 control instructions

20 instr. / output pixel → Scalar version >100 instr. / output pixel

PULP core examples – RV32IMC vs RV32IMCXpulp General code

- 2 bytes saved (X instructions not compressed)
- Number of instructions reduced (21 vs 18)
- Removed branch penalties

```
start_loop:
addi    a6,t1,-32
c.mv  a7,t5 //address of matA
addi    t3,a0,-32
loop0:
c.mv  a4,t3 //address of matA
c.mv  a2,a7
c.li  a5,0
loop1:
lbu a3,0(a4) //load byte
lbu a1,0(a2)
c.addi    a4,a4,1 //post increment
mul a3,a3,a1 //mul
c.add a5,a5,a3 //acc after mul
andi    a5,a5,255
c.addi    a2,a2,1
bne a4,a0,loop1 // branch penalty
sb  a5,0(a6)
c.addi    a6,a6,1
addi    a7,a7,32
bne a6,t1,loop0 //branch penalty
addi    t1,a6,32
addi    a0,a4,32
bne t1,t4,start_loop
```

```
start_loop:
addi    t3,t5,-32
c.mv   a7,s2 //address of matB
addi    t1,t4,-32
lp.setupi   x0,32,stop0
c.mv  a3,t1 //address of matA
c.mv  a2,a7
c.li  a5,0
sub a4,t4,t1 //loop count1
lp.setup    x1,a4,stop1 //hw loop
p.lbu   a0,1(a3!) //load byte with post increment
p.lbu   a1,32(a2!)
p.mac   a5,a0,a1 //mac
stop1: andi    a5,a5,255
p.sb    a5,1(t3!) //store result with post increment
stop0: c.addi    a7,a7,1
addi    t5,t5,32
addi    t4,a3,32
bne t5,t6,start_loop
```

PULP core examples – RV32IMCXpulp General code vs Opt code

- The innermost loop has 4x less iterations
  - 4 bytes per matrix are loaded as a 32b word
  - Dot product with accumulation performs in 1 cycle 4 macs

```
...
lp.setup    x1,a4,stop1
p.lbu   a0,1(a3!)
p.lbu   a1,1(a2!)
stop1: p.mac   a5,a0,a1
....
```

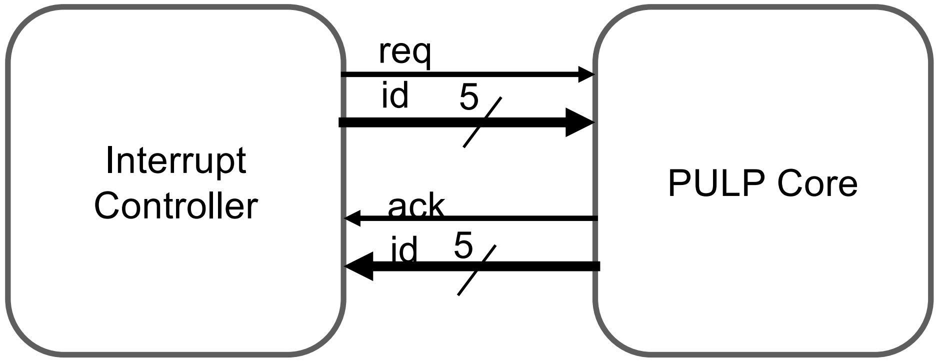
```
... //iterate #COL/4
lp.setup x1,a6,stop1
p.lw    a1,4(t1!) //load 4-bytes with post inc
p.lw    a5,4(t3!)
stop1: pv.sdotsp.b a7,a1,a5 //4 mac
....
```

PULP cores Interrupts

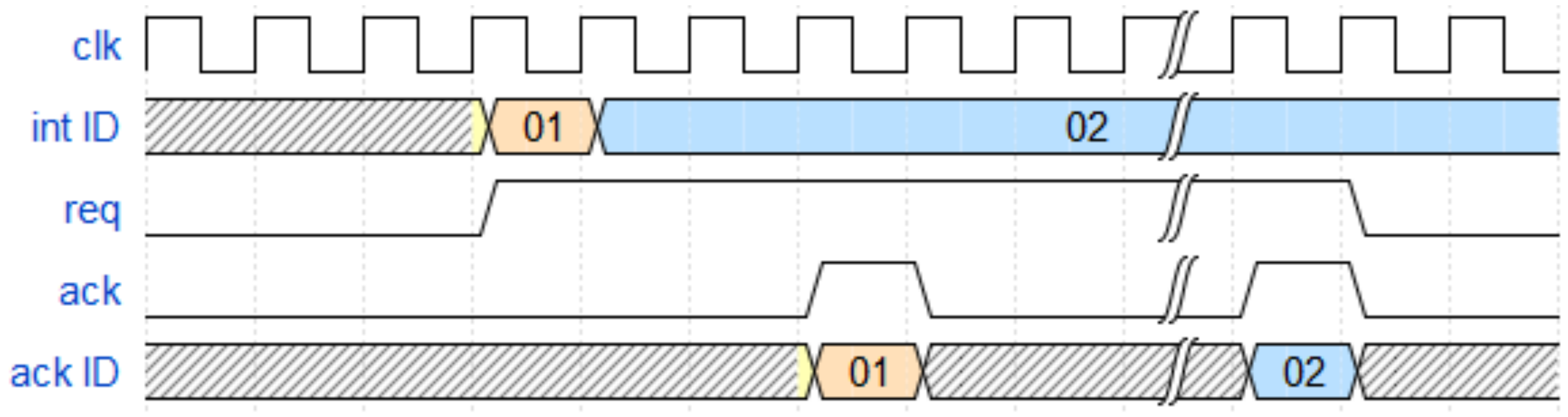
- *Asynchronous events*
  - If interrupt is taken, jump to xtvec
    - xtvec holds the base address to jump
      - + 4*interrupt ID for computing the actual address
  - No delegation supported
    - All interrupts are handled in machine mode
  - External interrupt controller interact with peripheral subsystem and SW events

PULP cores interrupts protocol

- Asynchronous protocol between CORE and INTController
  - The core takes few cycles before jumping
  - The external interrupt controller may change ID number
    - e.g. higher priority requests from peripherals
  - The core tells the interrupt controller which ID has been used to calculate the address of the interrupt vector table
  - The interrupt controller clears the taken ID

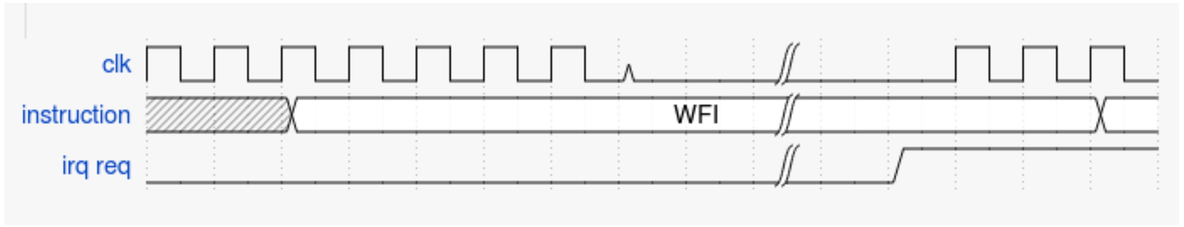


PULP cores interrupts protocol – timing diagram

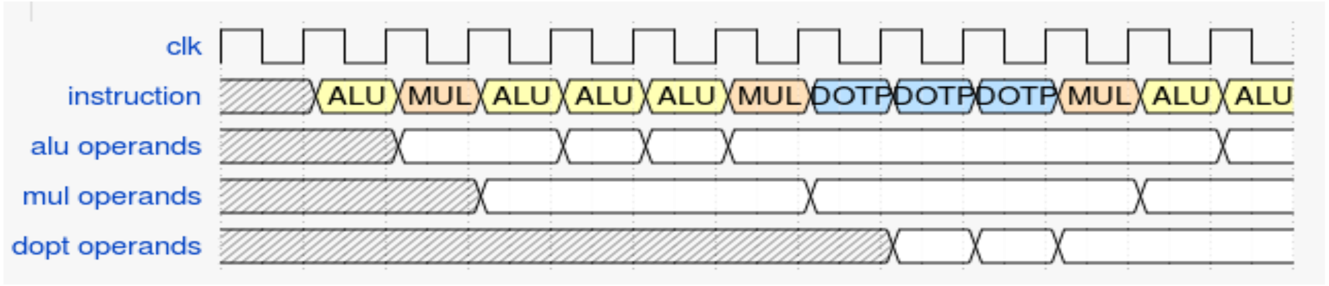


Wait For Interrupt & Power manager

- WFI instruction disables the clock
  - Dynamic power saved when core is in IDLE
  - Taken or Not interrupts wake up the core that starts from the instruction after WFI
  - The core waits for all the inflight instructions before switching off the clock
    - eg if a load is waiting for the valid signal, long divisions, floating point mac, debug



- The pipeline and state registers are clock gated when not used
  - The ALU, Integer Multiplier and Dot Product units have different operands registers
    - In the ID stage, the decoded instruction can be part of one of this 3 domains, the others 2 are clock gated



Performance Counter 1/3

- Registers in the CSR space that counts events
- Number of cycles and number of retired instructions used to calculate “IPC – Instructions per cycle”
- Performance counters used for counting the stalls
  - **Load stalls**
    - **Value not yet returned from memory**

```
//LOAD STALL
lw     x10, 0x0(x2)
add  x10, x10, 0x4
```

PC	IF	ID	EX	WB
A+4 to Imem	*Y from Imem[A]*	***add needs value from lw → STALL***	addr to Dmem	-
A+8 to Imem	*Y from Imem[A]*	***add needs value from lw → FWD***	Bubble	D from Dmem
A+12 to Imem	Z from Imem[A+4]	decode Y	add	

Performance Counter 2/3

- **Jump stalls (jalr)**
  - **Stall to break path from EX stage to Imem**

PC	IF	ID	EX	WB
A+4 to Imem	*Y from Imem[A]*	*jalr needs x10→ STALL*	mul	-
x10+0x4 to Imem	*Bubble*	*Jump to x10+0x4*	-	-

```
//JALR STALL
mul   x10, x10, x10
jalr  x11, x10, 0x4
```

Performance Counter 3/3

- Other performance counters used to monitor
  - Number of cycles lost for fetching (Instruction Cache for instance)
  - Number of load, stores, branches, taken branches, jumps and compressed instructions

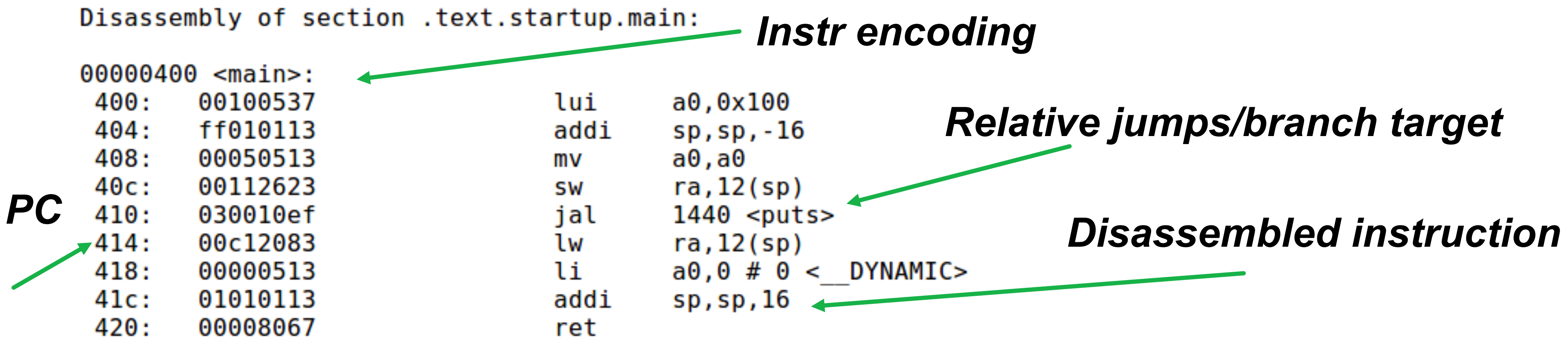
Address	Perf Counter	Description
0x782	LD_STALL	Number of load data hazards
0x783	JR_STALL	Number of jump register data hazards
0x784	IMISS	Cycles waiting for instruction fetches, i.e. number of instructions wasted due to non-ideal caching
0x785	LD	Number of data memory loads executed. Misaligned accesses are counted twice
0x786	ST	Number of data memory stores executed. Misaligned accesses are counted twice
0x787	JUMP	Number of unconditional jumps (j, jal, jr, jalr)
0x788	BRANCH	Number of branches. Counts taken and not taken branches
0x789	BTAKEN	Number of taken branches.
0x78A	RVC	Number of compressed instructions executed

Example Performance Counter

```
…enable perf counters…
csrw      0x782,x0  //reset perf counter LD_STALL
// loop 100 times over load stall
lp.setupi  x1,100,stop_loop
lw        x10,4(x14!)
stop_loop: add       x11,x11,x10 //stall due to load dependency
csrr      x15,0x782 // → x15 contains 100
```

Simulation Tracer

- For every instruction executed, the core prints on a file the
  - “TIME STAMP – PC – INSTRUCTION – OPERANDs and RESULTs”



```
Disassembly of section .text.startup.main:

00000400 <main>:
 400:   00100537                lui     a0,0x100
 404:   ff010113                addi    sp,sp,-16
 408:   00050513                mv      a0,a0
 40c:   00112623                sw      ra,12(sp)
 410:   030010ef                jal     1440 <puts>
 414:   00c12083                lw      ra,12(sp)
 418:   00000513                li      a0,0 # 0 <__DYNAMIC>
 41c:   01010113                addi    sp,sp,16
 420:   00008067                ret
```

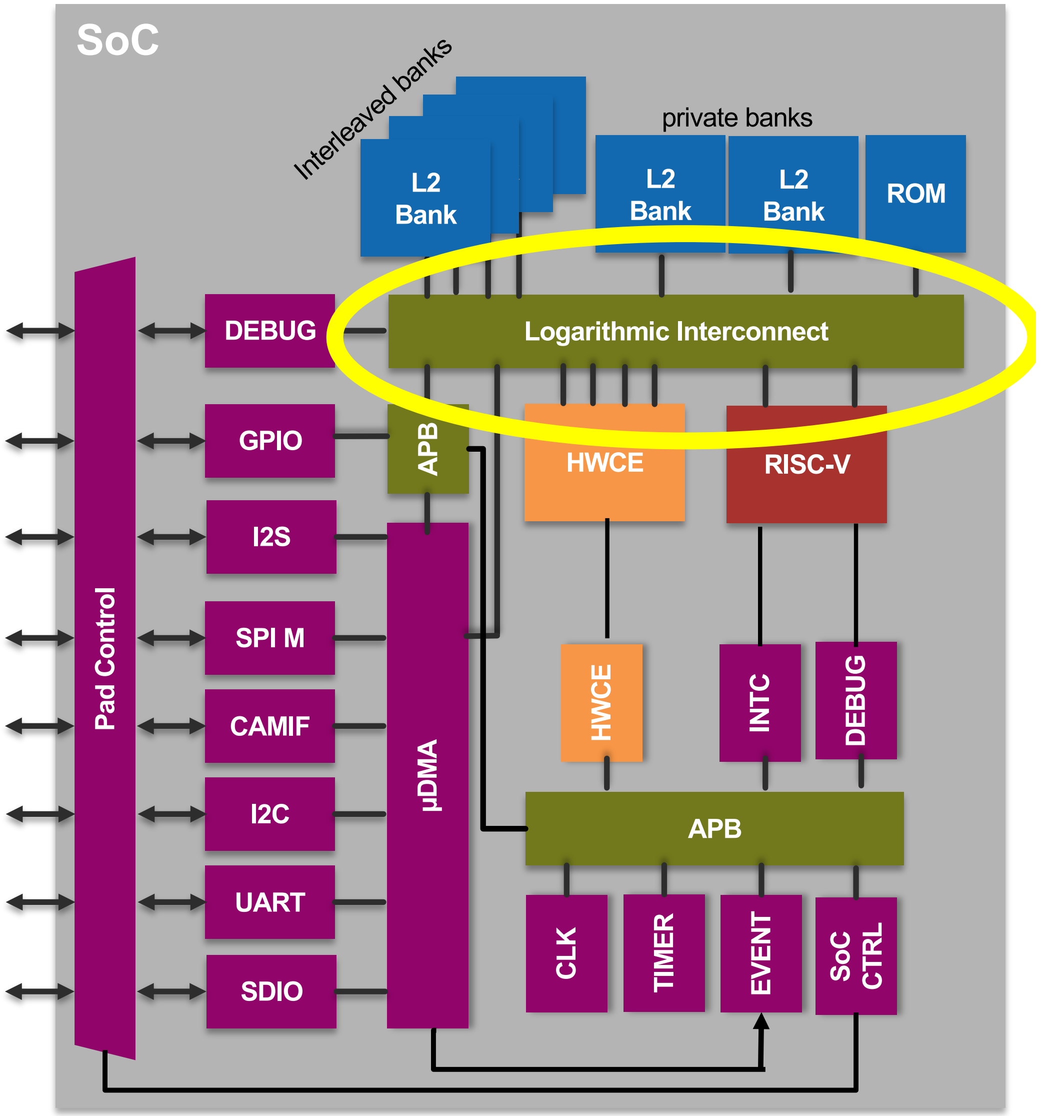
- **Trace file (build/pulpissimo/trace_core_1f_0.log):**

```
    Time          Cycles PC       Instr    Mnemonic
18880000             455 00000080 00c0006f jal              x0, 12
18960000             457 0000008c 30501073 csrrw            x0, x0, 0x305
19000000             458 00000090 00000093 addi             x1, x0, 0            x1=00000000
19040000             459 00000094 00008113 addi             x2, x1, 0            x2=00000000  x1:00000000
19080000             460 00000098 00008193 addi             x3, x1, 0            x3=00000000  x1:00000000
19120000             461 0000009c 00008213 addi             x4, x1, 0            x4=00000000  x1:00000000
19160000             462 000000a0 00008293 addi             x5, x1, 0            x5=00000000  x1:00000000
19200000             463 000000a4 00008313 addi             x6, x1, 0            x6=00000000  x1:00000000
19240000             464 000000a8 00008393 addi             x7, x1, 0            x7=00000000  x1:00000000
19280000             465 000000ac 00008413 addi             x8, x1, 0            x8=00000000  x1:00000000
19320000             466 000000b0 00008493 addi             x9, x1, 0            x9=00000000  x1:00000000
19360000             467 000000b4 00008513 addi             x10, x1, 0          x10=00000000  x1:00000000
```

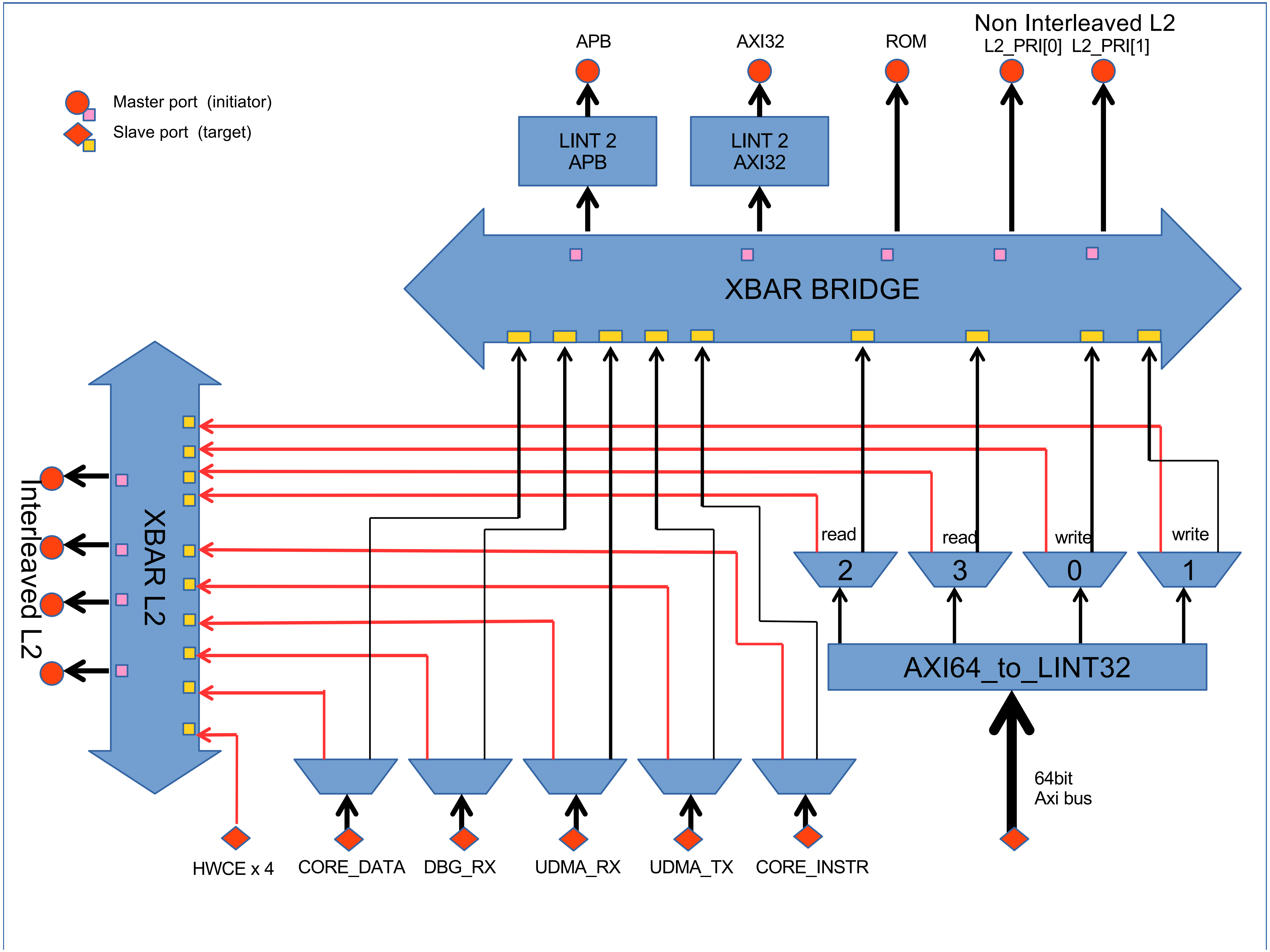
Hybrid Logaritmic Interconnect

PULPissimo Architecture

- RISC-V based advanced microcontroller
  - 512kB of L2 Memory
  - 16kB of energy efficient latch-based memory (L2 SCM BANK)
- Rich set of peripherals:
  - QSPI (up to 280 Mbps)
  - Camera Interface (up to 320x240@60fps)
  - I2C, I2S (up to 4 digital microphones)
  - JTAG (Debug), GPIOs,
  - Interrupt controller, Bootup ROM
- Autonomous IO DMA Subsystem (µDMA)
- Power management
  - 2 low-power FLLs (IO, SoC)

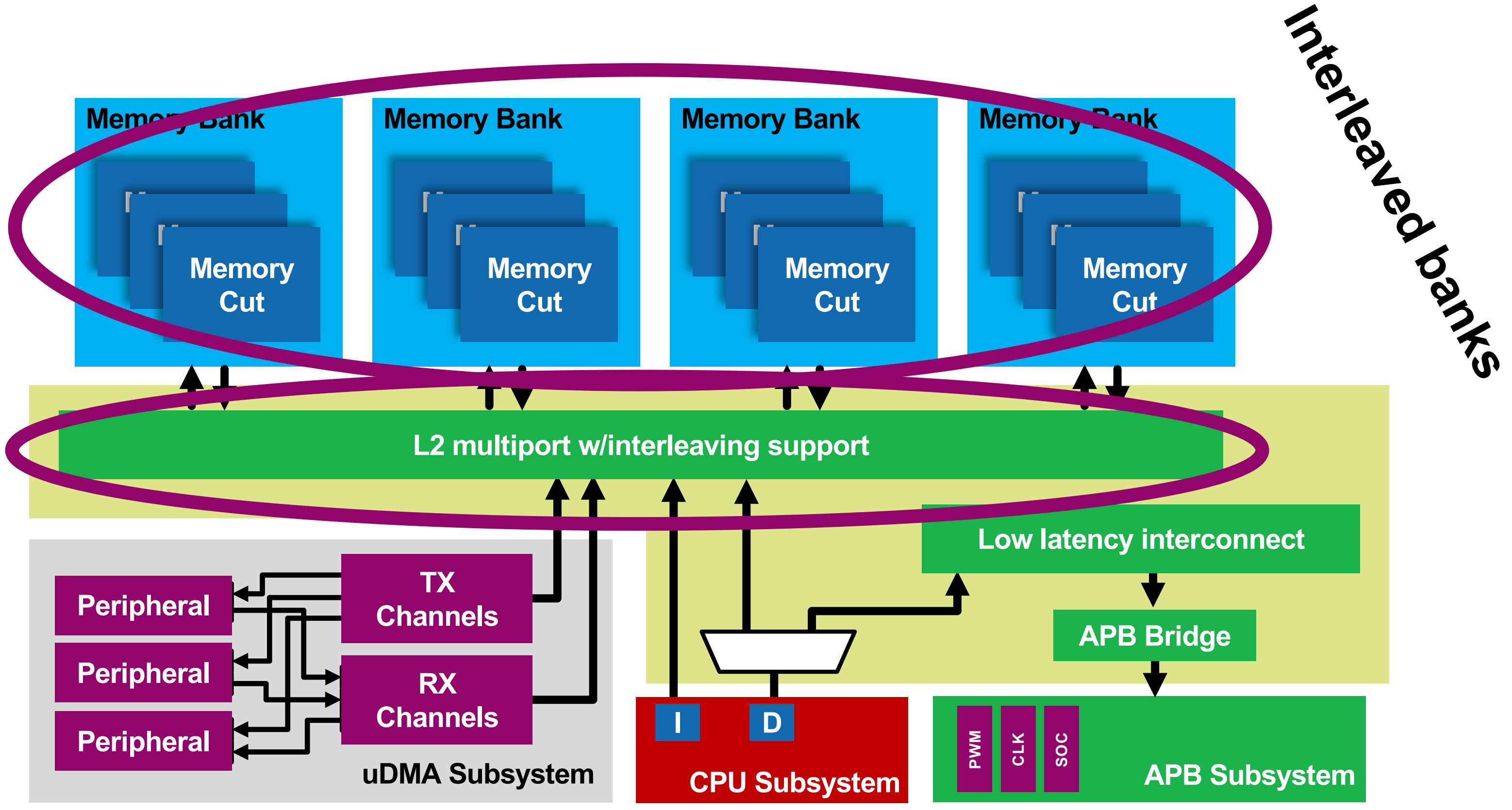


PULP

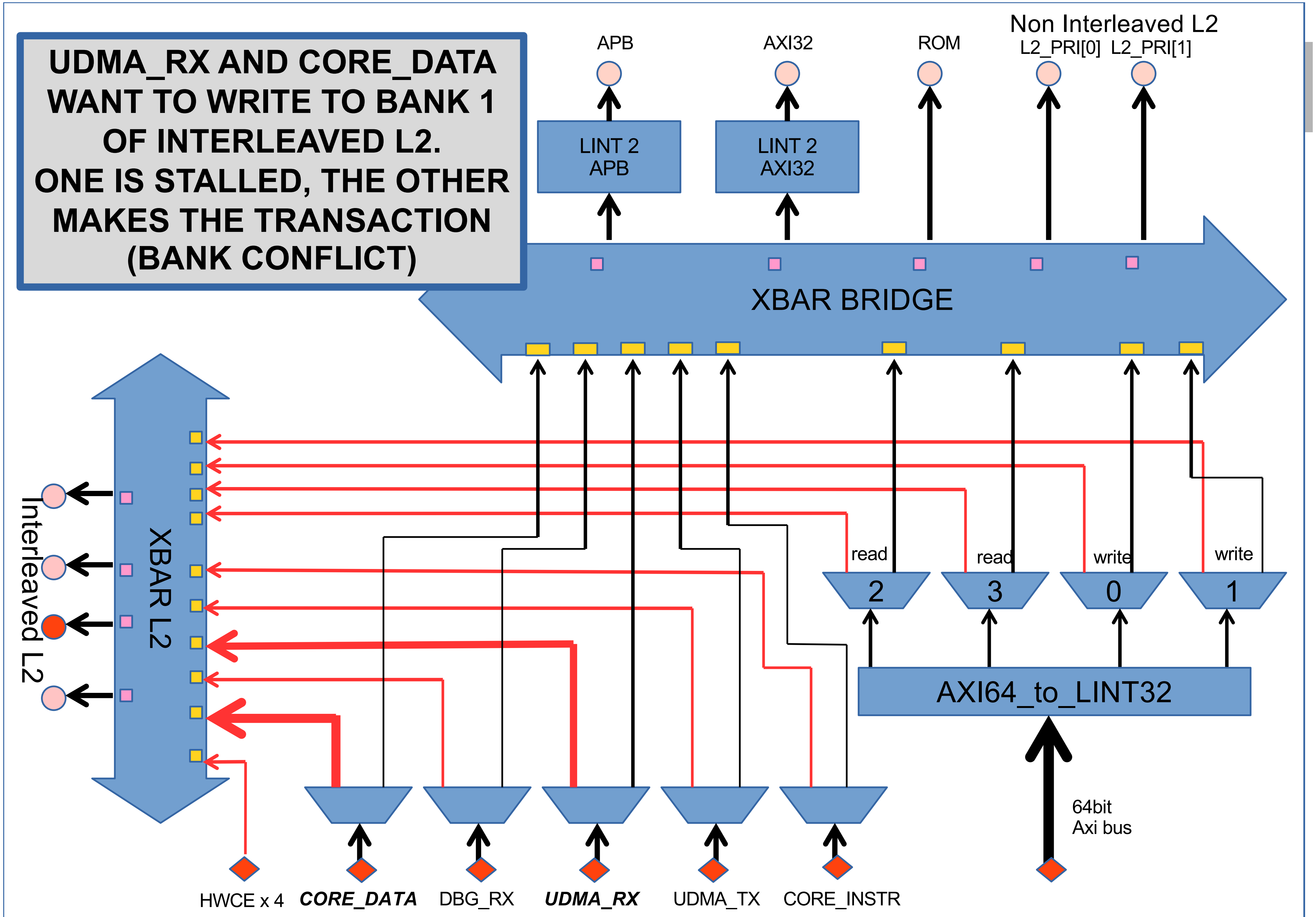

Master port (initiator)
Slave port (target)
APB
AXI32
ROM
Non Interleaved L2
L2_PRI[0]
L2_PRI[1]
LINT 2 APB
LINT 2 AXI32
XBAR BRIDGE
XBAR L2
Interleaved L2
read
read
write
write
2
3
0
1
AXI64_to_LINT32
64bit Axi bus
HWCE x 4
CORE_DATA
DBG_RX
UDMA_RX
UDMA_TX
CORE_INSTR

Interconnect performance

- Low latency interconnect with word level interleaving to reduce contention
- 4 PORT memory capable of handling maximum BW of 4*32*Freq
- High performance plugs to the processing subsystem



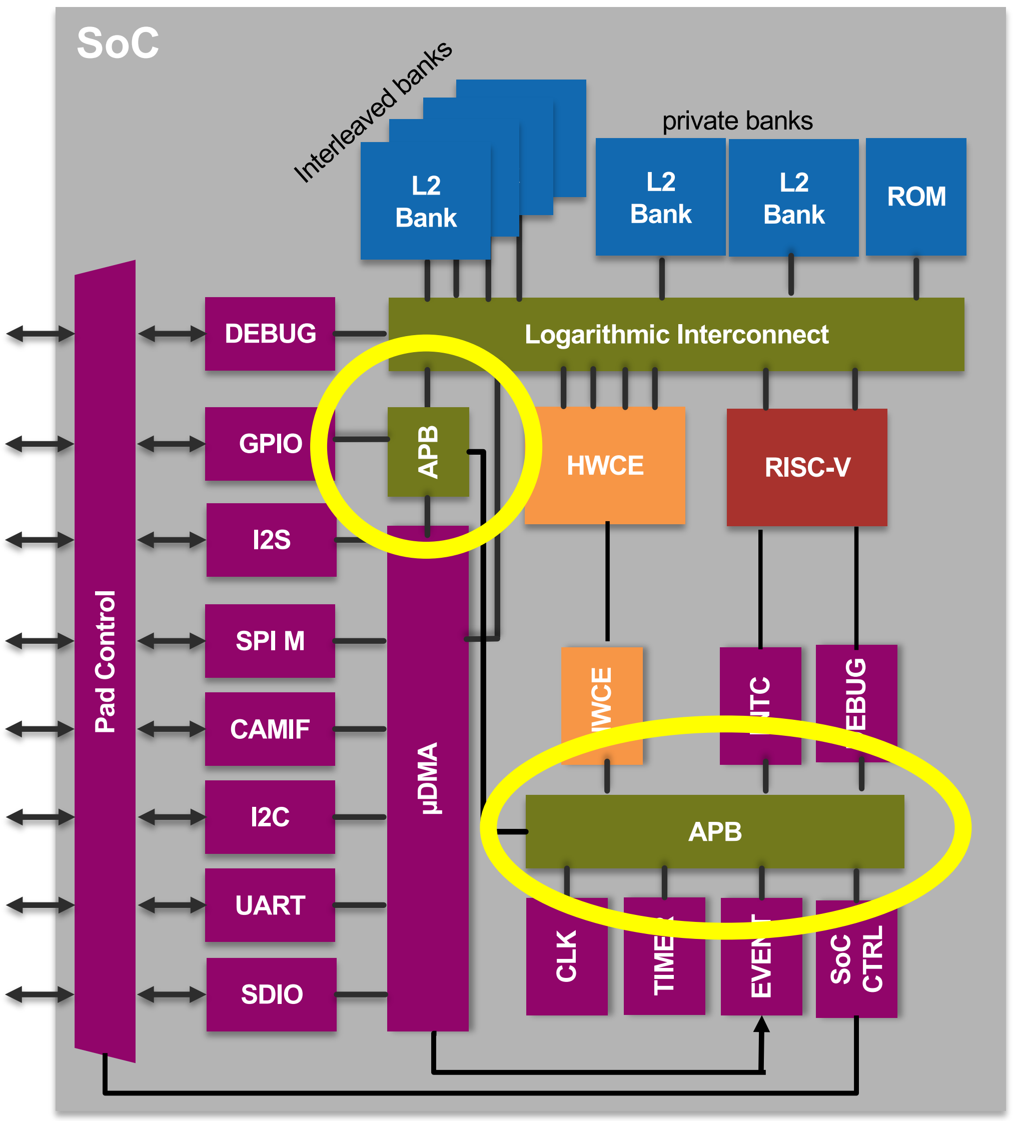
UDMA_RX AND CORE_DATA WANT TO WRITE TO BANK 1 OF INTERLEAVED L2. ONE IS STALLED, THE OTHER MAKES THE TRANSACTION (BANK CONFLICT)



Peripheral Interconnect

PULPissimo Architecture

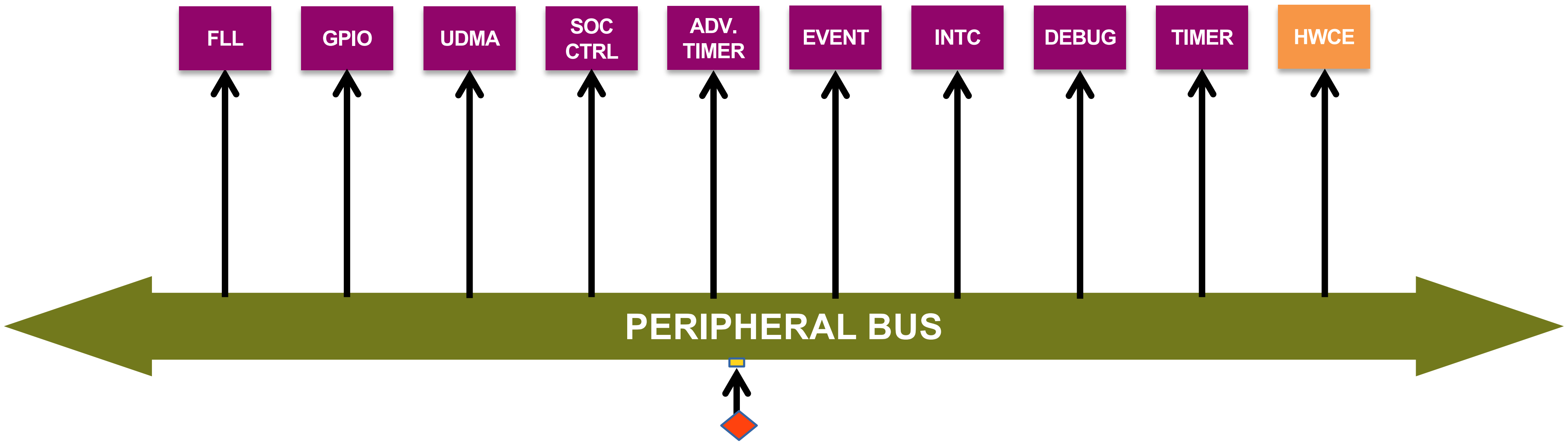
- RISC-V based advanced microcontroller
  - 512kB of L2 Memory
  - 16kB of energy efficient latch-based memory (L2 SCM BANK)
- Rich set of peripherals:
  - QSPI (up to 280 Mbps)
  - Camera Interface (up to 320x240@60fps)
  - I2C, I2S (up to 4 digital microphones)
  - JTAG (Debug), GPIOs,
  - Interrupt controller, Bootup ROM
- Autonomous IO DMA Subsystem (µDMA)
- Power management
  - 2 low-power FLLs (IO, SoC)



PULP

Peripheral Bus

Only one APB request! If more, stalled in the HYBRID LOGARITMIC INTERCONNECT (AS BANK CONFLICT)

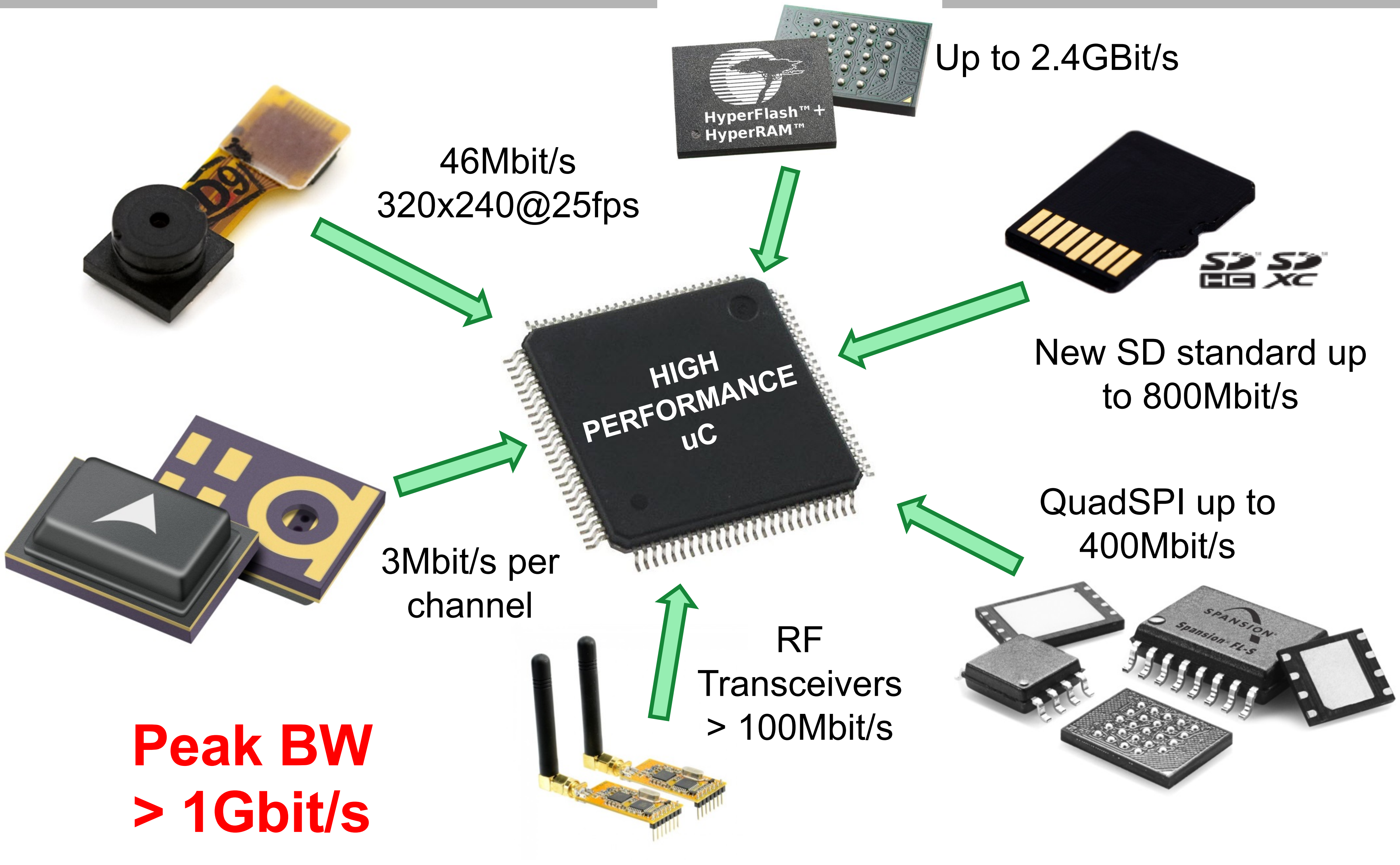


APB (COMING FROM HYBRID LOG INT)

*0x1A1*_**00000**

µDMA: An Autonomous I/O Subsystem

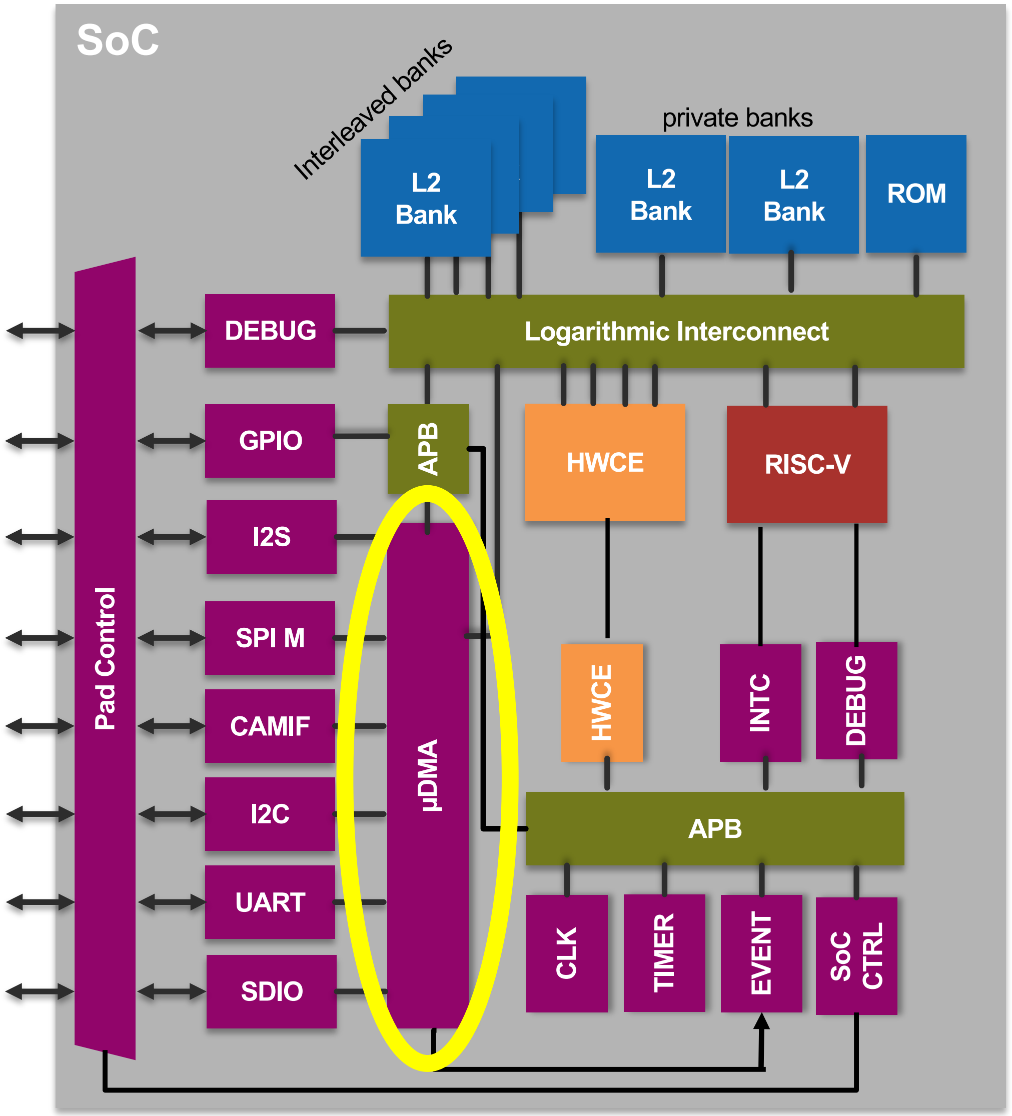
I/O requirements



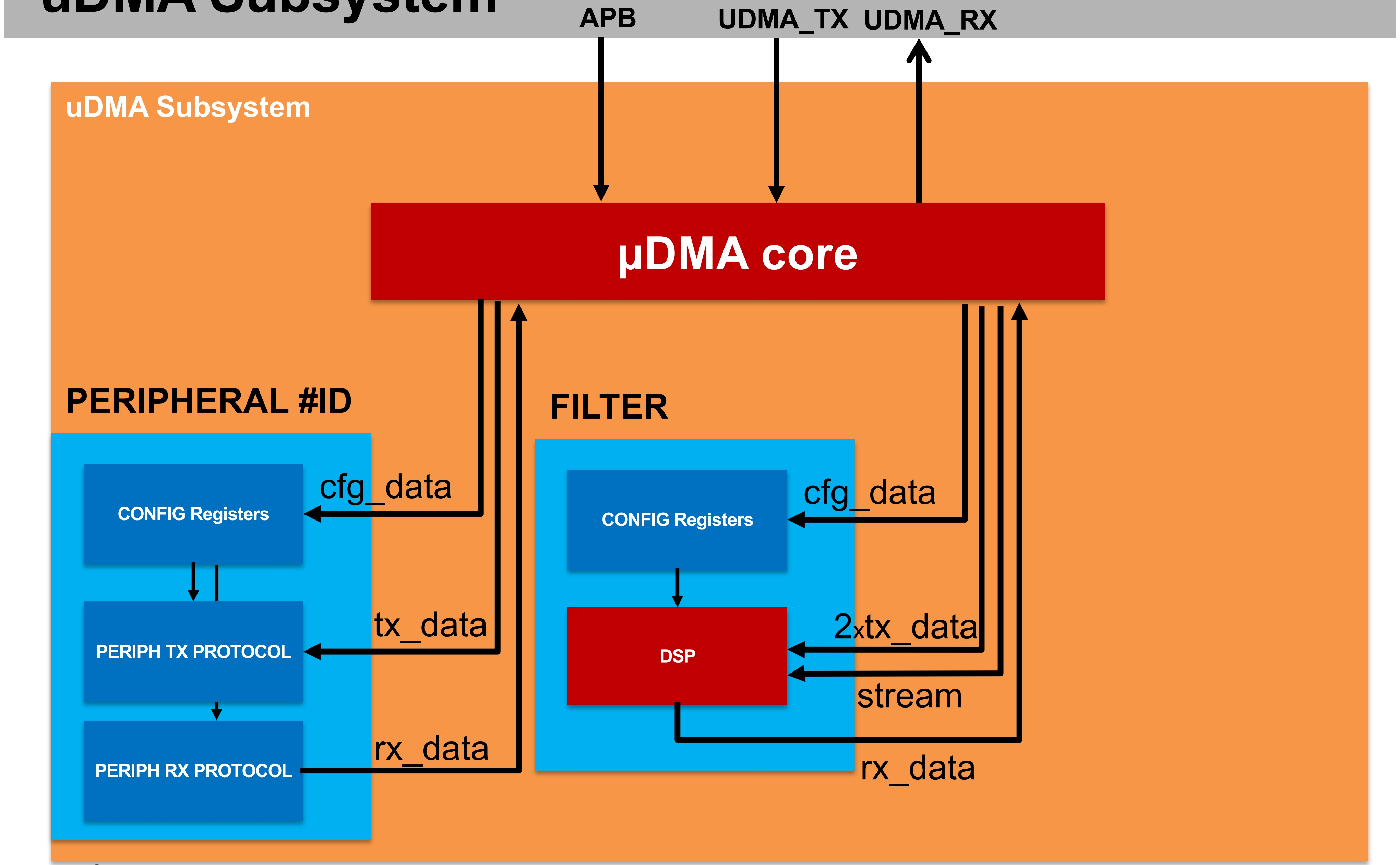
Peak BW > 1Gbit/s

PULPissimo Architecture

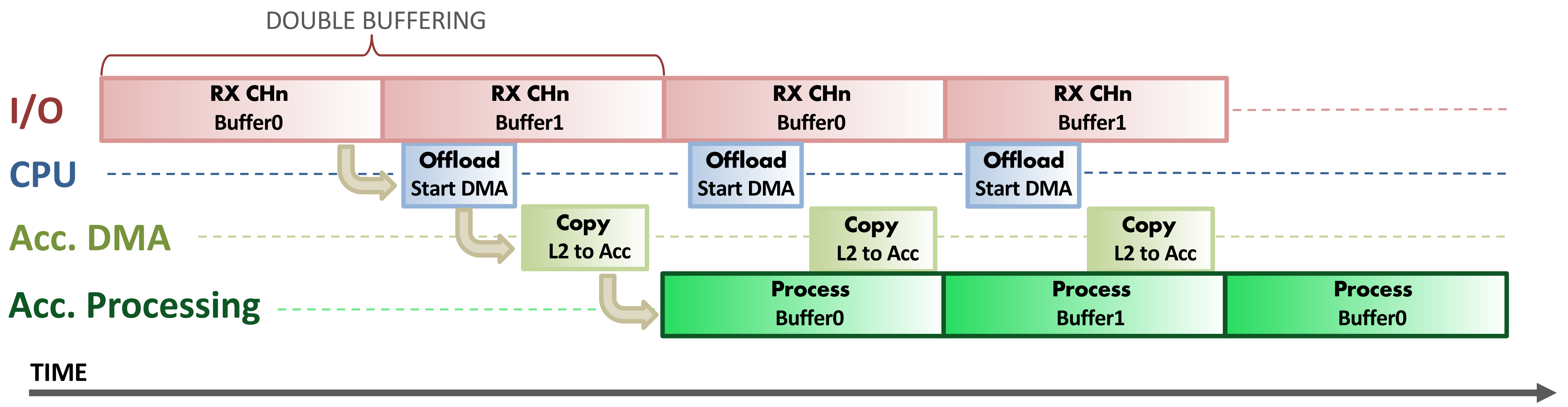
- RISC-V based advanced microcontroller
  - 512kB of L2 Memory
  - 16kB of energy efficient latch-based memory (L2 SCM BANK)
- Rich set of peripherals:
  - QSPI (up to 280 Mbps)
  - Camera Interface (up to 320x240@60fps)
  - I2C, I2S (up to 4 digital microphones)
  - JTAG (Debug), GPIOs,
  - Interrupt controller, Bootup ROM
- Autonomous IO DMA Subsystem (µDMA)
- Power management
  - 2 low-power FLLs (IO, SoC)



uDMA Subsystem



Offload pipeline

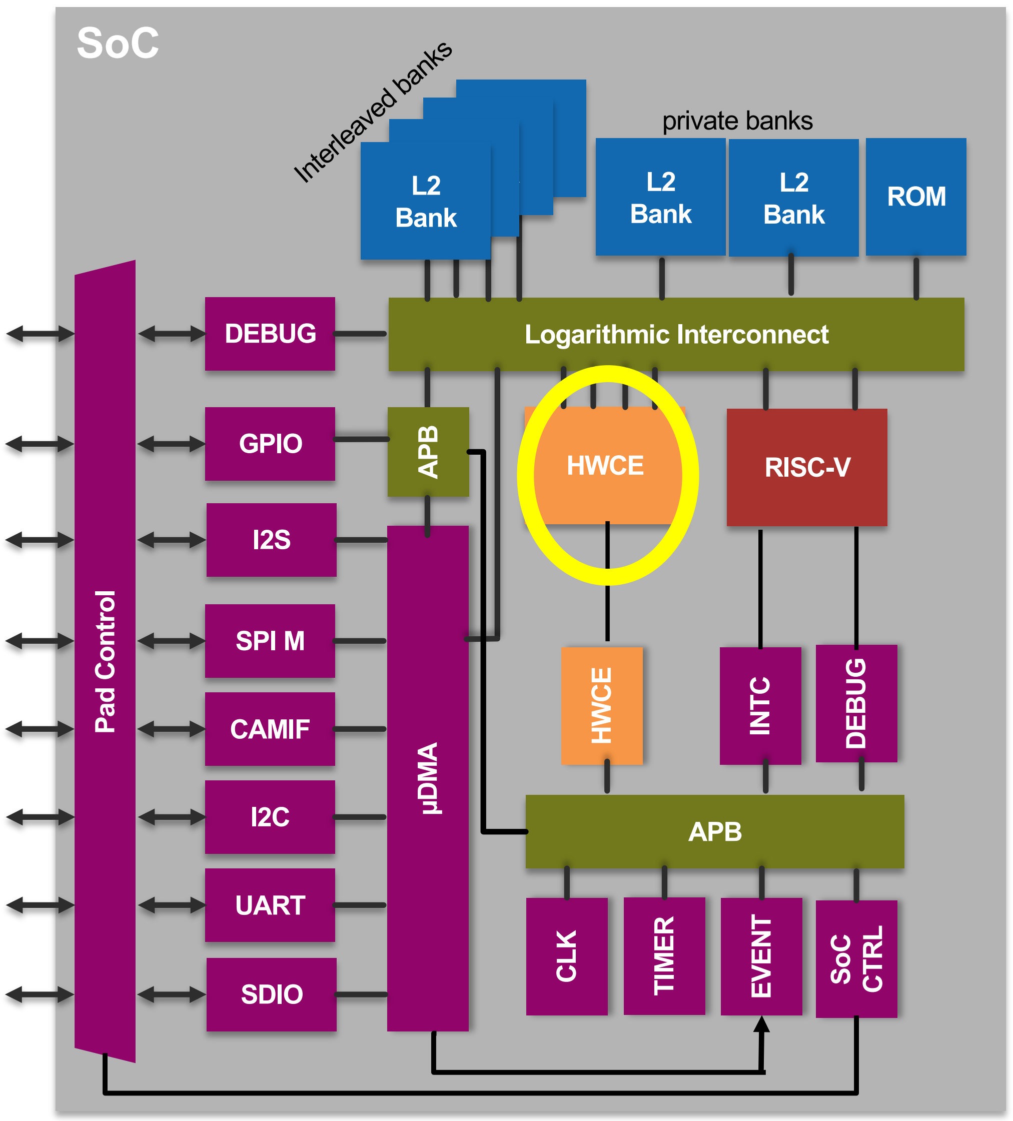


- Efficient use of system resources
- HW support for double buffering allows continuous data transfers
- Multiple data streams can be time multiplexed

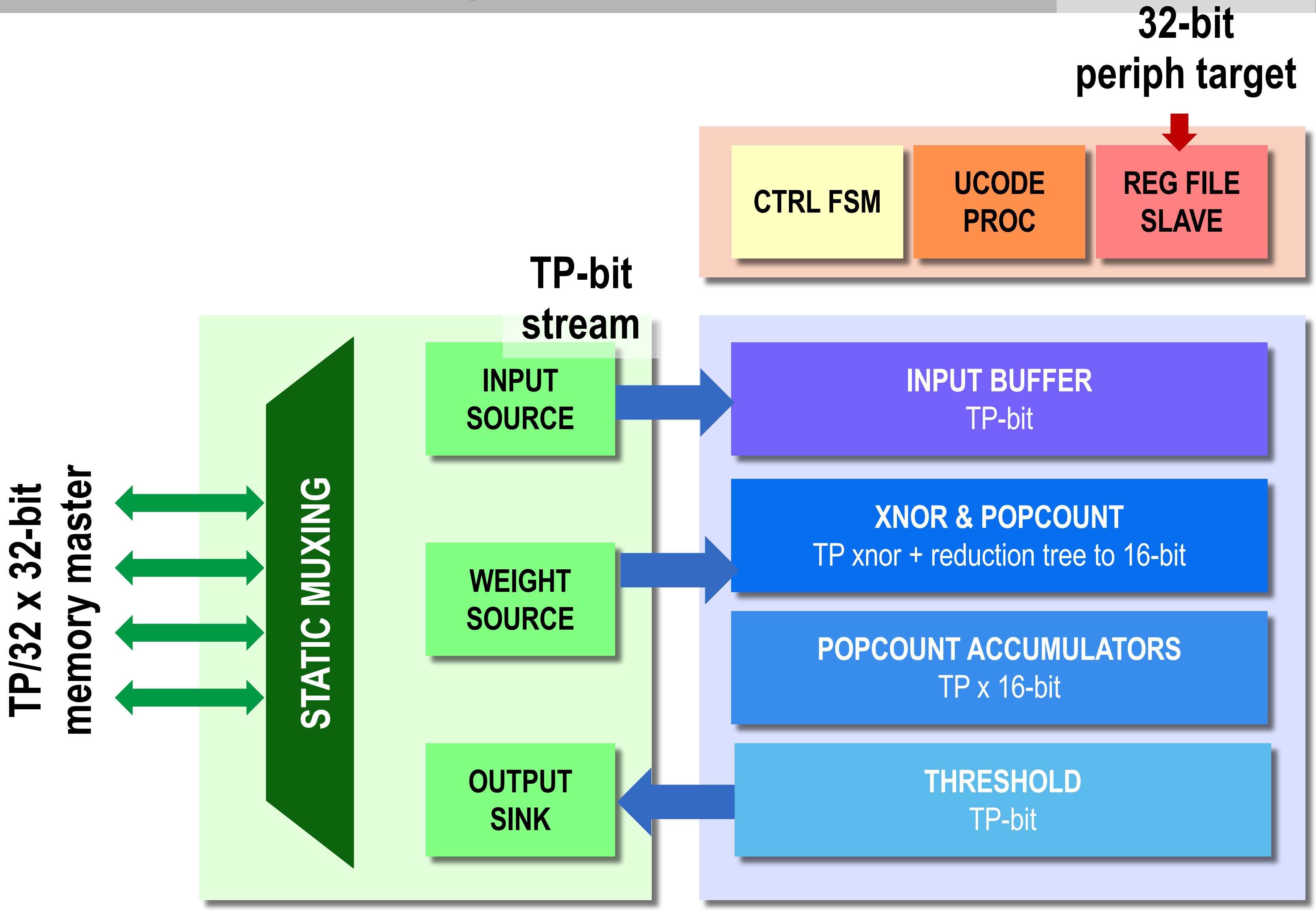
Hardware Accelerator for Neural Networks

PULPissimo Architecture

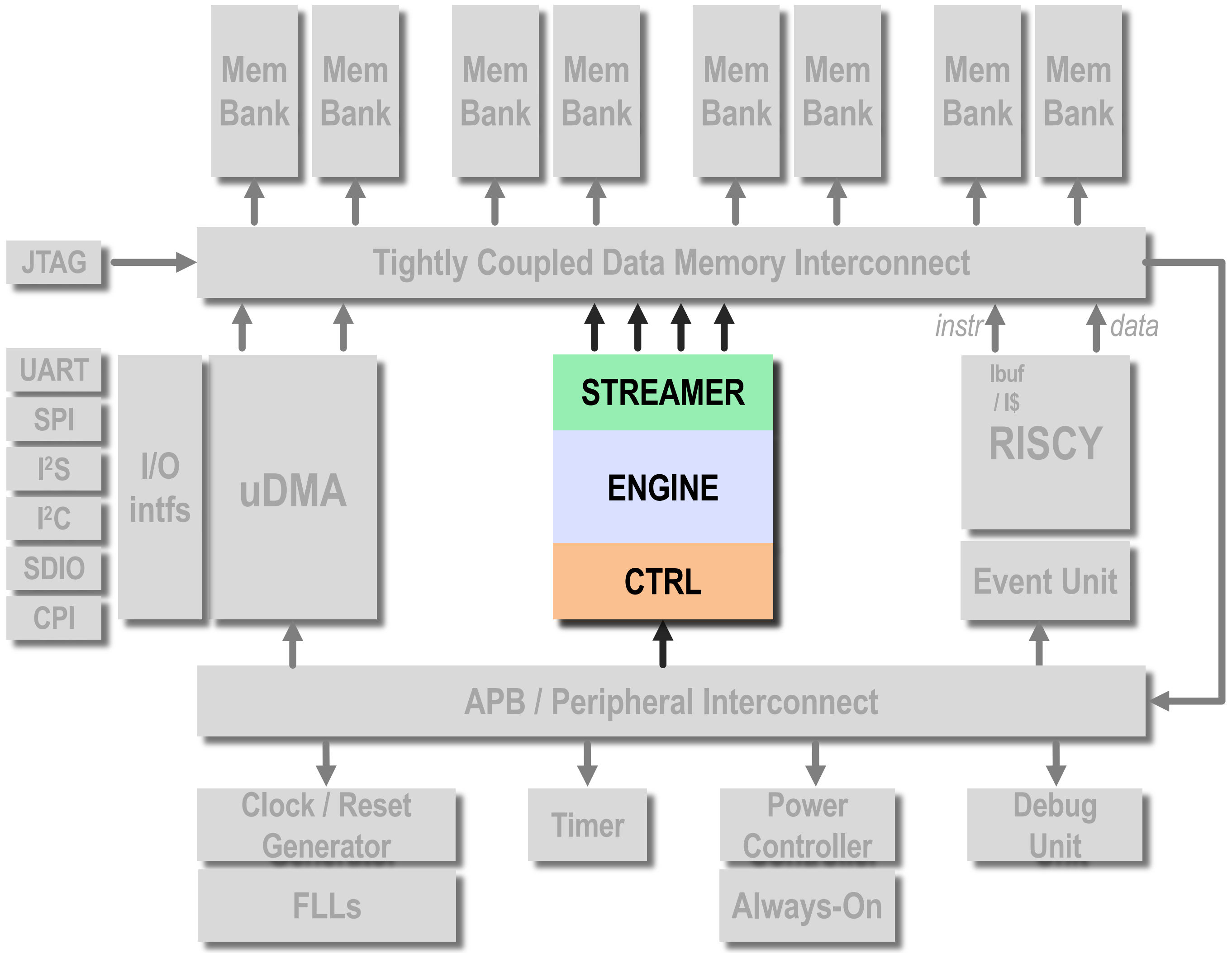
- RISC-V based advanced microcontroller
  - 512kB of L2 Memory
  - 16kB of energy efficient latch-based memory (L2 SCM BANK)
- Rich set of peripherals:
  - QSPI (up to 280 Mbps)
  - Camera Interface (up to 320x240@60fps)
  - I2C, I2S (up to 4 digital microphones)
  - JTAG (Debug), GPIOs,
  - Interrupt controller, Bootup ROM
- Autonomous IO DMA Subsystem (µDMA)
- Power management
  - 2 low-power FLLs (IO, SoC)



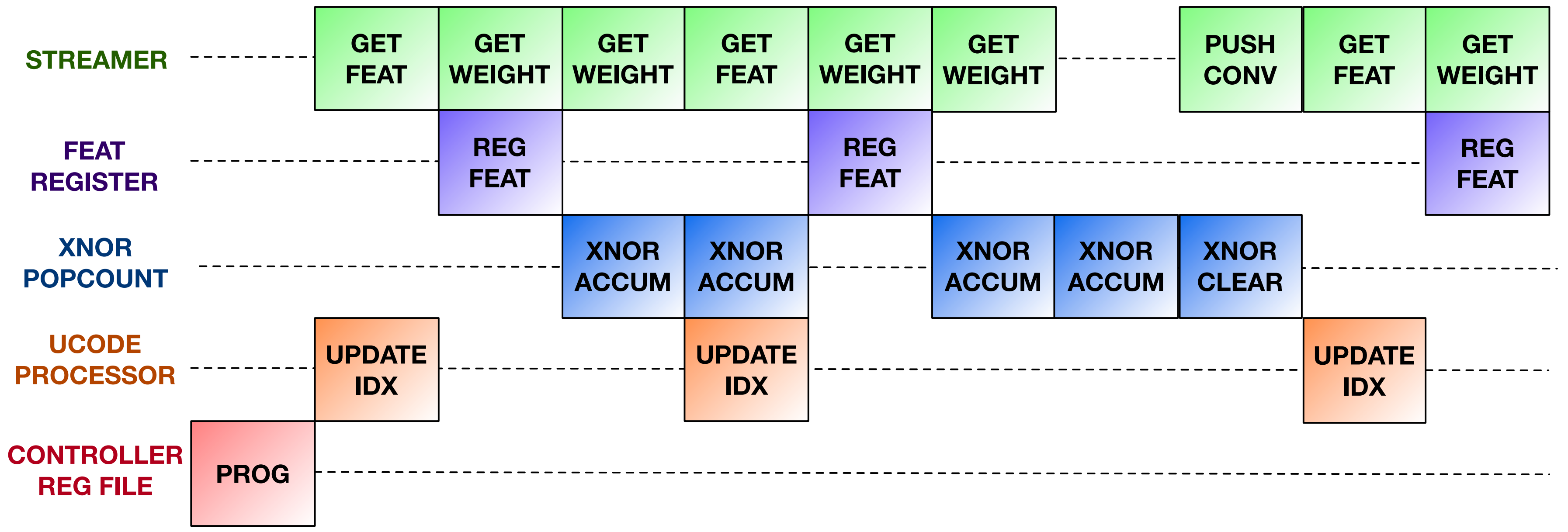
XNOR Neural Engine



XNOR Neural Engine in PULPissimo



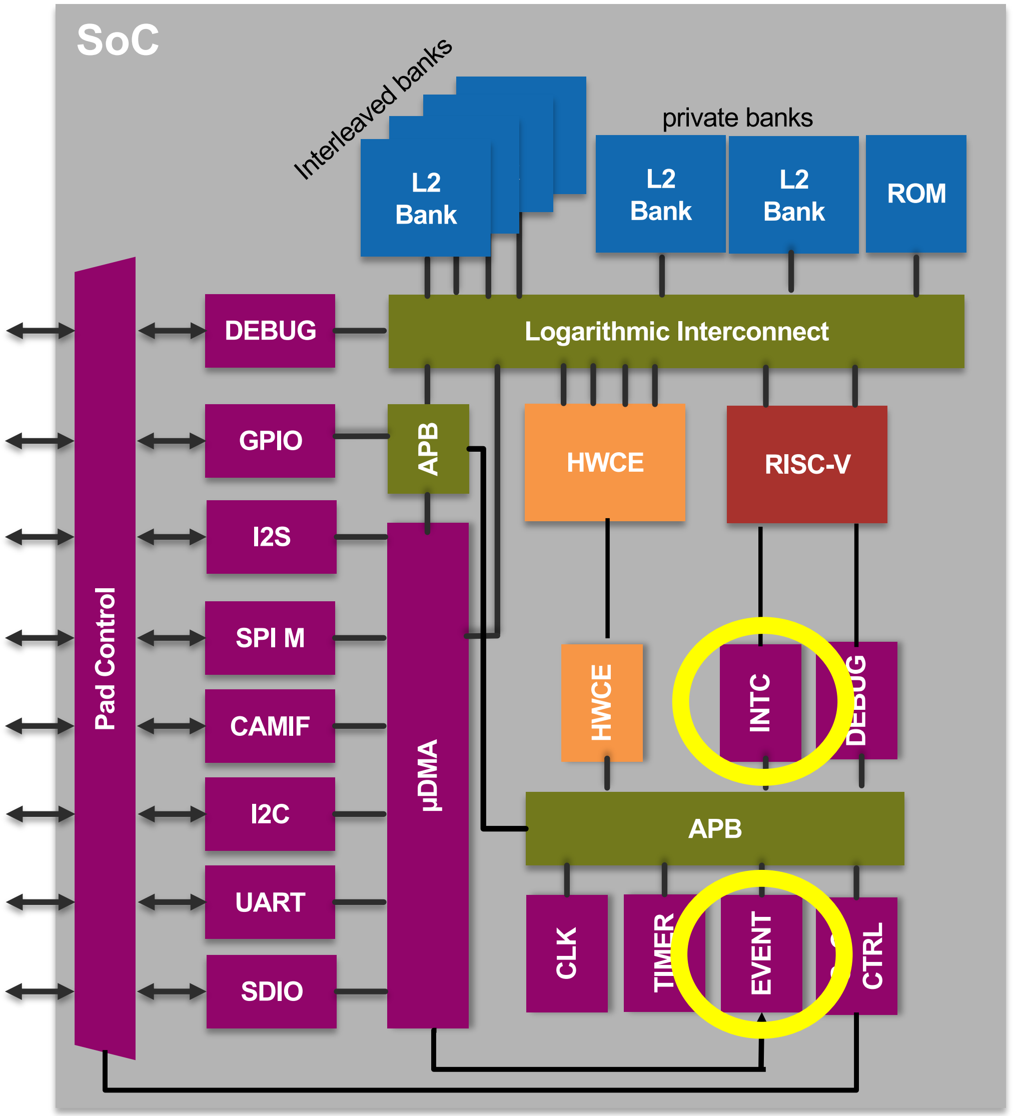
Operation in time



Interrupt Controller and Event Generator

PULPissimo Architecture

- RISC-V based advanced microcontroller
  - 512kB of L2 Memory
  - 16kB of energy efficient latch-based memory (L2 SCM BANK)
- Rich set of peripherals:
  - QSPI (up to 280 Mbps)
  - Camera Interface (up to 320x240@60fps)
  - I2C, I2S (up to 4 digital microphones)
  - JTAG (Debug), GPIOs,
  - Interrupt controller, Bootup ROM
- Autonomous IO DMA Subsystem (µDMA)
- Power management
  - 2 low-power FLLs (IO, SoC)

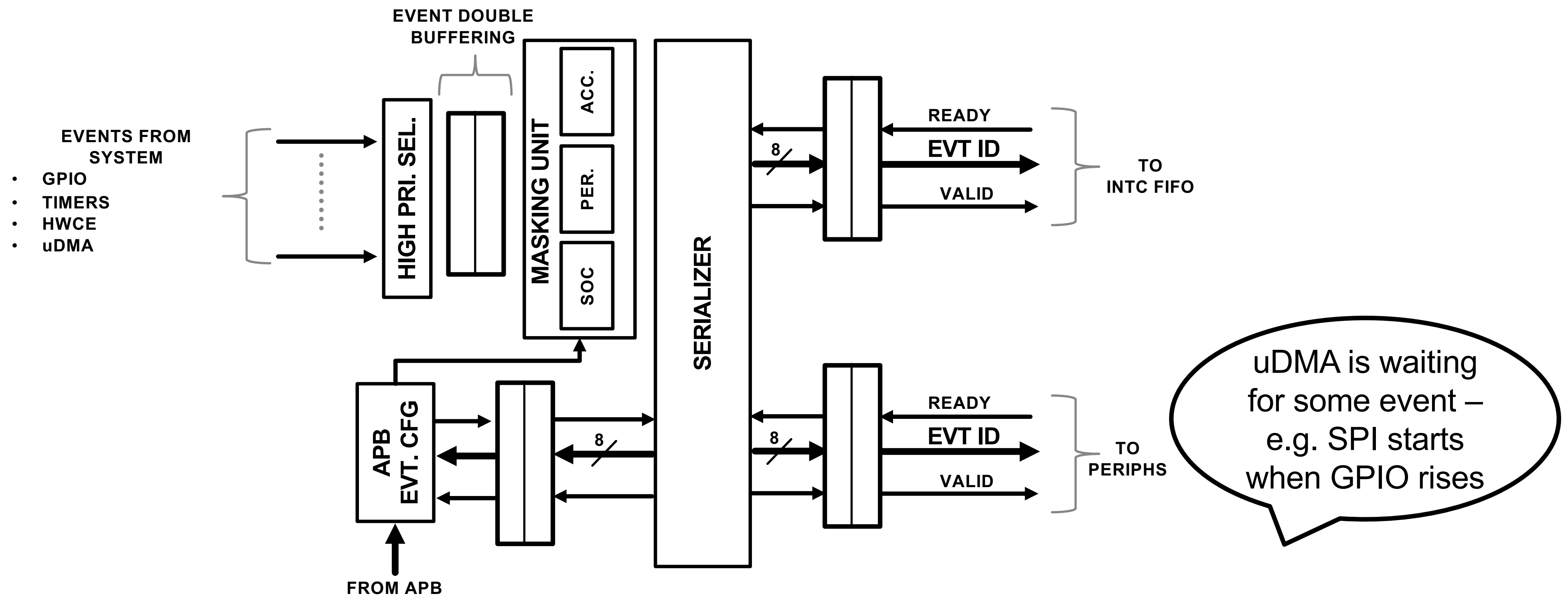


PULP

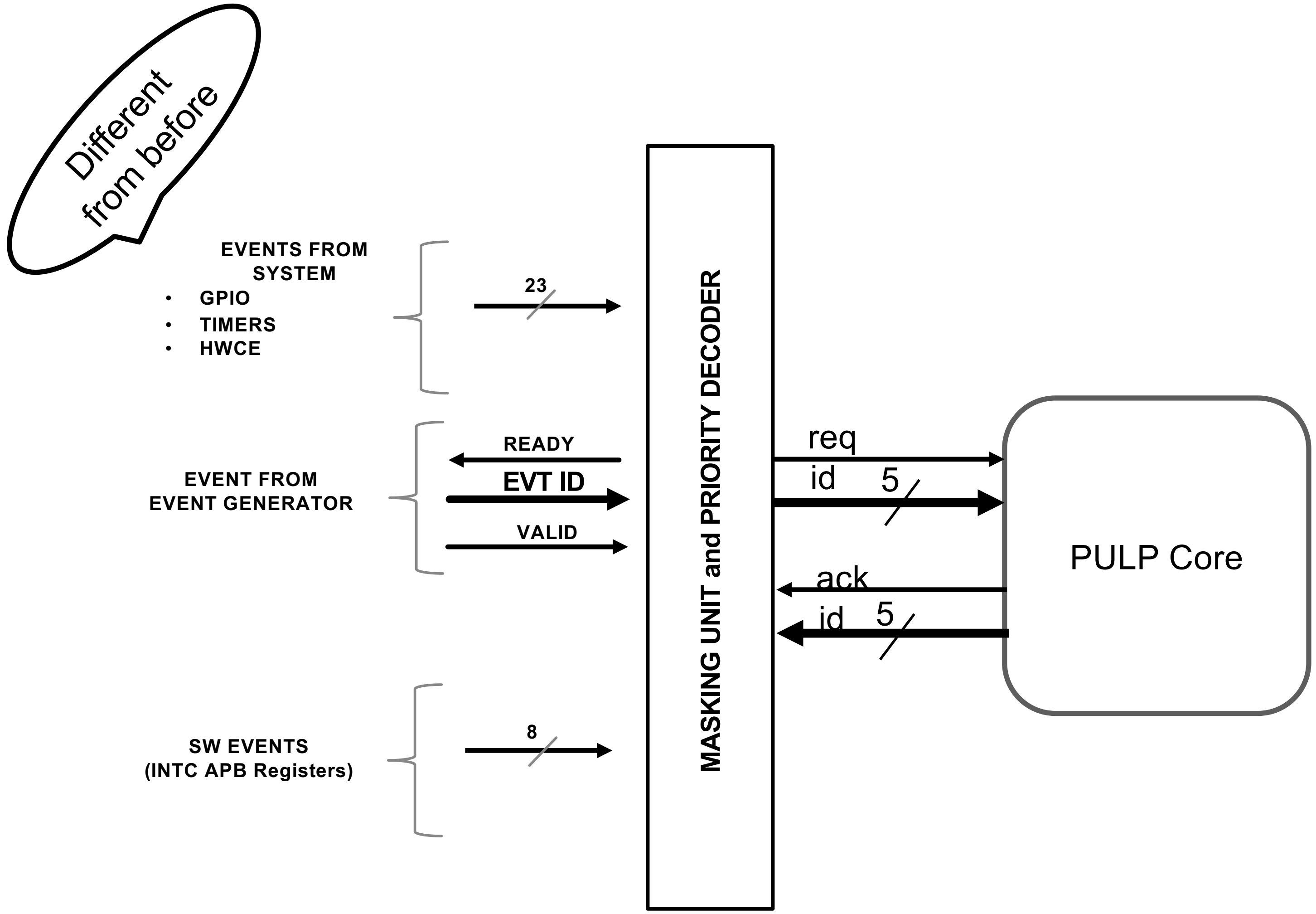
PULP interrupts controller (INTC)

- It generates interrupt requests from 0 to 31
- Mapped to the APB bus
- Receives events in a FIFO from the SoC Event Generator (i.e. from peripherals)
  - Unique interrupt ID (26) but different event ID
- Mask, pending interrupts, acknowledged interrupts, event id registers
- <u>*Set, Clear, Read and Write*</u> operations by means of load and store instructions (memory mapped operations)
- Interrupts come from:
  - Timers
  - GPIO (rise, fall events)
  - HWCE
  - **Events i.e. uDMA**

PULP Event Generator (EVENT)



Interrupts Source



TestBench

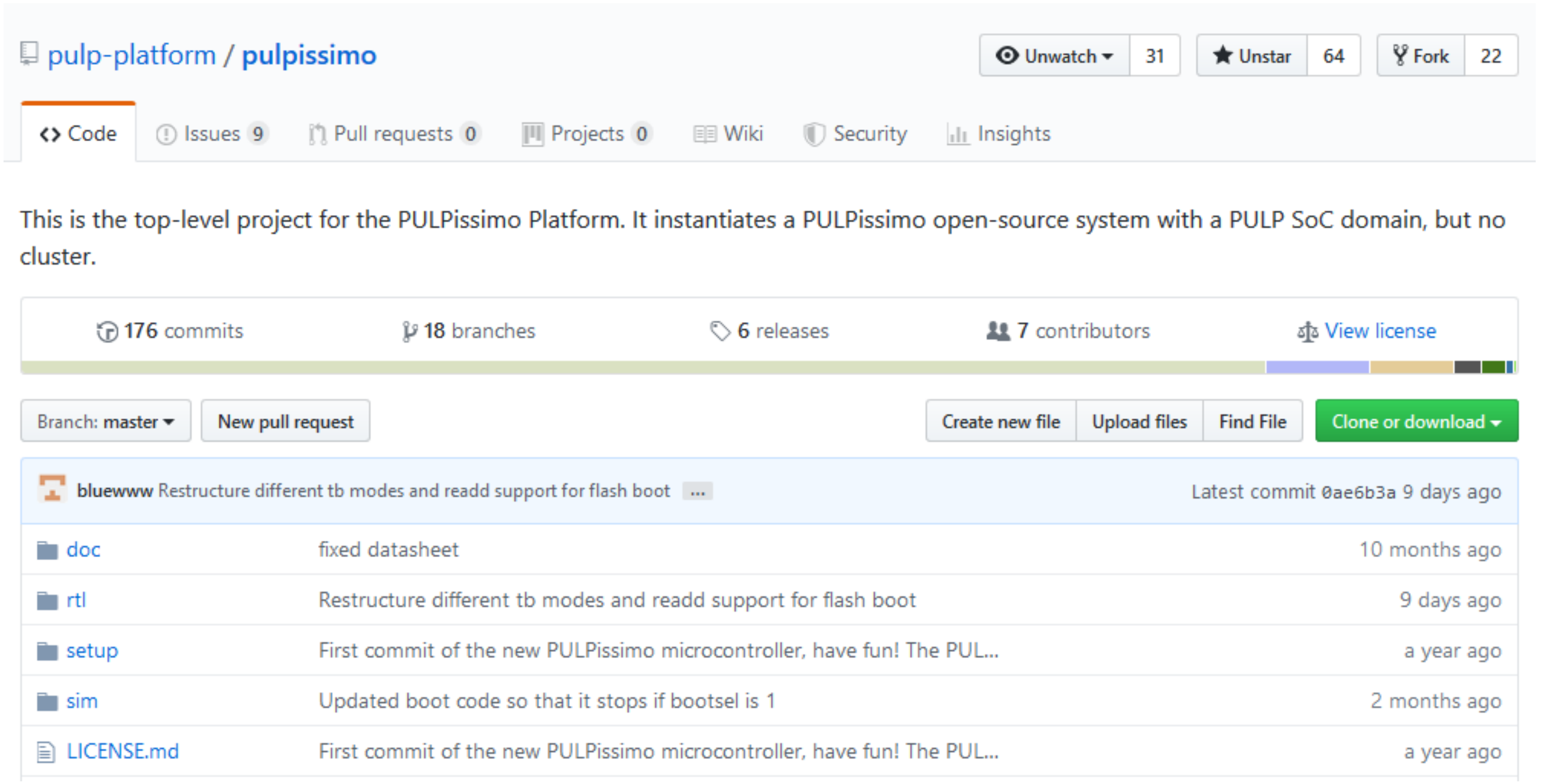
PULP TestBench

- It reads the compiled file (ADDRESS – INSTRUCTION)
- It sets with JTAG configuration registers
- It loads via JTAG the compiled file into the memory
- It writes to the FETCH_ENABLE register in the APB (Soc Control)
  - Now the core starts running the application
- It waits for the END-OF-COMPUTATION bit
  - When the core returns from the “main” function, it writes to a specific memory location in the APB (SoC Control) the word “1XXX_XXXX”, where 1 indicates the core finished its program and XXX_XXXX is the returned value (e.g. “**return 0**;”)
- It reports an error if XXX_XXXX is not 0

HANDS-ON

PULPissimo on GitHub

- PULPissimo is available @ https://github.com/pulp-platform/pulpissimo
  - git clone git@github.com:pulp-platform/pulpissimo.git



pulp-platform / pulpissimo

This is the top-level project for the PULPissimo Platform. It instantiates a PULPissimo open-source system with a PULP SoC domain, but no cluster.

176 commits | 18 branches | 6 releases | 7 contributors | View license

bluewww Restructure different tb modes and readd support for flash boot ... Latest commit 0ae6b3a 9 days ago

doc	fixed datasheet	10 months ago
rtl	Restructure different tb modes and readd support for flash boot	9 days ago
setup	First commit of the new PULPissimo microcontroller, have fun! The PUL...	a year ago
sim	Updated boot code so that it stops if bootsel is 1	2 months ago
LICENSE.md	First commit of the new PULPissimo microcontroller, have fun! The PUL...	a year ago

Dependecies

- "*ips_list.yml*" holds the needed sub-IPs.
- "*update-ips*" to download them
- *iptools* downloads the IPs recursively
- *iptools* generates compilation scripts and [synthesis] scripts
- "*ips*" folder contains downloaded IPs
- "rtl" folder contains PULPissimo RTL, testbench, etc

PULP IPs

- Every IP is a different GIT repository
  - Easier to maintain and creates little mess on many-people projects
  - Every IP has one or more maintainers of the PULP group

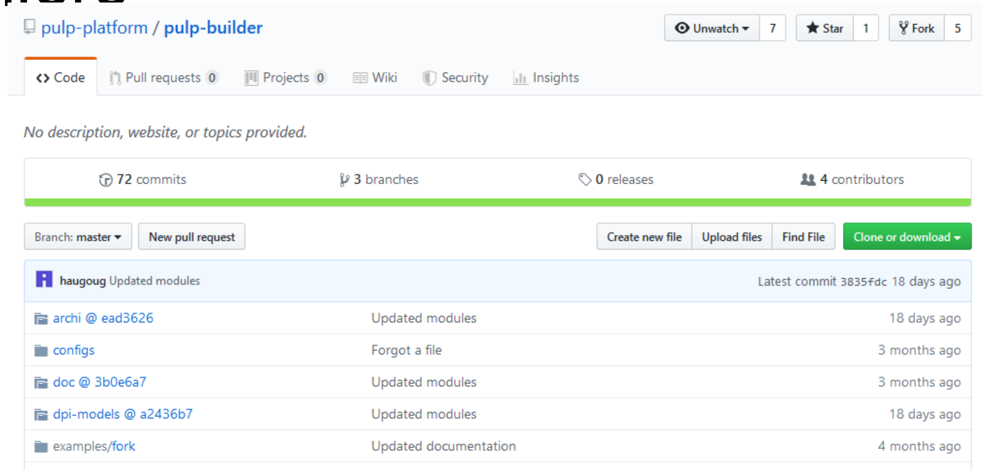
- "*src_files.yml*" for each IP to list the RTL files
  - used to generate scripts, modelsim library names, options, etc

- PULPissimo IPs are also available on GitHub

- *make sdk* to download and install the PULP SDK

PULP SDK

- The SDK contains all the tools and runtime support for PULP based microcontrollers



pulp-platform / pulp-builder

Unwatch 7 | Star 1 | Fork 5

Code | Pull requests 0 | Projects 0 | Wiki | Security | Insights

No description, website, or topics provided.

72 commits | 3 branches | 0 releases | 4 contributors

Branch: master | New pull request | Create new file | Upload files | Find File | Clone or download

haugoug Updated modules — Latest commit 3835fdc 18 days ago

archi @ ead3626	Updated modules	18 days ago
configs	Forgot a file	3 months ago
doc @ 3b0e6a7	Updated modules	3 months ago
dpi-models @ a2436b7	Updated modules	18 days ago
examples/fork	Updated documentation	4 months ago

- The SDK contains from low-level bare-metal procedure for setting the PULP cores and peripherals (e.g. crt0) all the way up to a set of higher level functions (API) to help applications developers to leverage all the supported features

Environment Variable

- *VSIM_PATH* points to your *pulpissimo/sim* folder
  - Execute make clean lib build opt

- *PULP_RISCV_GCC_TOOLCHAIN* to your bin folder of the PULP GCC

Compile & Simulate PULP

- PULP compilation and simulation scripts and flow are based on *modelsim*
  - **To compile**
    - *cd pulpissimo/sim*
    - *make clean lib build opt*
  - **To execute an application**
    - *cd yourapplicationfolder*
    - *make clean all (to compile it)*
      - *make dis > dis.s (to generate the object dump)*
      - *make run gui=1 (to run modelsim with GUI)*
    - *make run*
  - **Assembler, Simulation Trace and Performance counter to analyze performance**

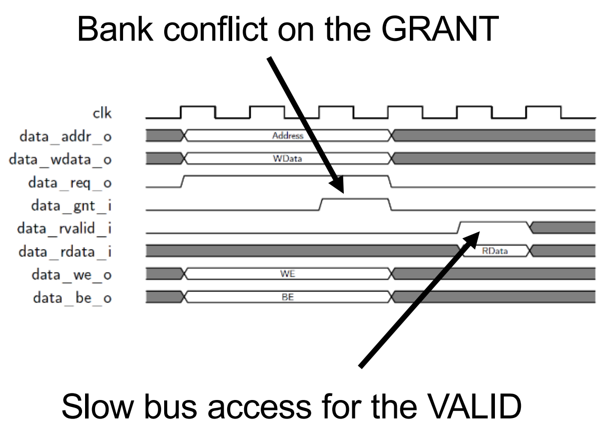
Programming PULP

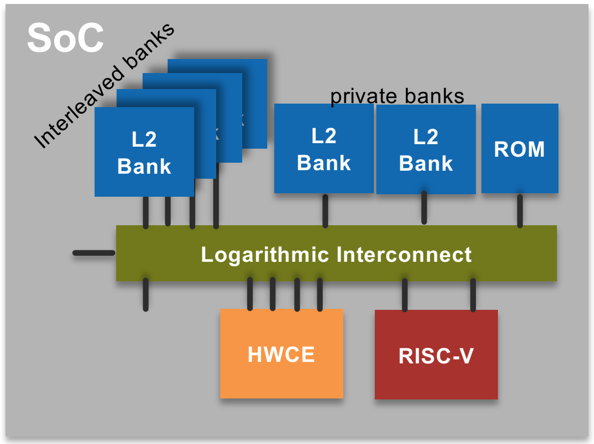
- When programming for embedded system, the very first thing that should come to your mind is
  - LIMITED RESOURCES
    - It is completely different to write application for your personal PC than a microcontroller
    - You MUST know the total memory available, the architecture, the instructions of the core etc
    - In the context of embedded programming, you have the possibility to finely optimize all the SW stack to leverage your HW at the best

- Some tips are coming ☺

The C->ASM->MONITOR Loop

- When you write your C program, you must have in mind many things:
  - Where are my DATA? In which BANK? Will I have BANK conflicts? With whom?
  - Where are my instructions? In which BANK? Will I have BANK conflicts? With whom?
    - → This tells you whether you will have stalls from outside due to the system rather than the program per se, yet it is very important to know





- Core data stack in L2 Private Bank0
- Instructions in L2 Private Bank1
- HWCE data in L2 interleaved

None of the master ports in the Log. Interconnect Will create bank conflicts ☺

The C->ASM->MONITOR Loop

- When you write your C program, you must have in mind many things:
  - What is the ISA of my core?
  - Is my kernel (e.g. MatMul) using all the instructions of my ISA in an optimized way?
    - → You check this by generating the assembler and double check the instructions. Try to reverse what the compiler did as see whether you can do better or not
      - If not, you can use builtins or asm volatile statements to force the use of some instructions! (Or rewrite properly the C code)

```
...
lp.setup    x1,a4,stop1
p.lbu   a0,1(a3!)
p.lbu   a1,1(a2!)
stop1: p.mac   a5,a0,a1
....
```

```
... //iterate #COL/4
lp.setup x1,a6,stop1
p.lw    a1,4(t1!)
p.lw    a5,4(t3!)
stop1: pv.sdotsp.b a7,a1,a5
....
```

The C->ASM->MONITOR Loop

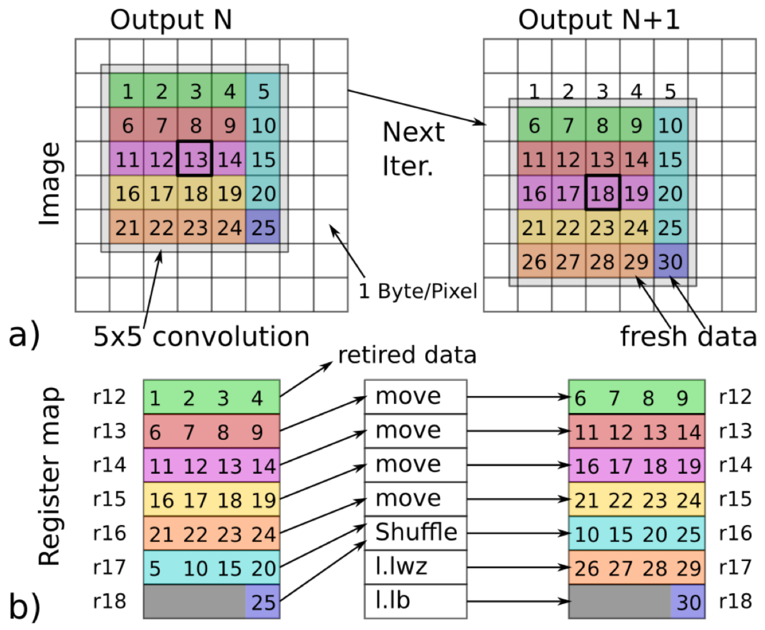
- When you write your C program, you must have in mind many things:
  - What are the performance I should expect?
  - Can I achieve that performance? Why?
    - If I don't have a clue, I should open the waveform and see where the stalls are coming from
  - How can I solve it? → Back to writing CODE (e.g. loop unrolling)

Hands-On → The Dot Product

- The dot product is an extremely common kernel in Artificial Intelligence operations
  - **RISCY extensions to achieve top performance**
  - **We are going to see**
    - *Optimized assembly code that uses*
      - *the MAC instruction*
      - *Zero-overhead HW-Loop*
      - *Automatic increment load/store*
    - *Loop unrolling to eliminate stalls*
    - *Optimized code that uses the SIMD extensions*

Hands-On → The 2D Convolution

- The 2D Convolution is the central kernel of Convolutional Neural Network
  - **RISCY extensions to achieve top performance**
  - **We are going to see**
    - *Optimized C code that uses*
      - ***gcc** vectors*
      - *the shuffle instruction*
      - *the dot product*
      - *normalization and clip*



Thanks a lot

- Thanks a lot for your attention
- I hope you enjoyed it ☺
- Get ready for the Hands On session