Understanding and working with PULP

Pasquale Davide Schiavone
and the PULP team

1Integrated System laboratory, ETH, Zurich, Switzerland
2Energy Efficient Embedded Systems Laboratory, University Of Bologna, Bologna, Italy

13.06.2019
Near Sensor (aka Edge) Processing

Sense

- MEMS IMU
- MEMS Microphone
- ULP Imager
- EMG/ECG/EIT

100 µW ÷ ~10 mW

Onchip Processing

1 ÷ 3 GOPS
1 ÷ 30 mW

Smart Architecture
Parallel Processing
Power-saving Design
Near-Threshold
Low Power Technology

Transmit

Idle: ~1 µW
Active: ~ 50 mW

Battery + Harvesting powered → a few mW power envelope
PULPissimo Architecture

SoC

Logarithmic Interconnect

Pad Control

APB

L2 Bank

L2 Bank

L2 Bank

ROM

DEBUG

GPIO

I2S

SPI M

CAMIF

I2C

UART

SDIO

μDMA

CLK

TIMER

EVENT

SoC CTRL

interleaved banks

private banks

pad control

L2 Bank

Interleaved banks

Private banks
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
• 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 management
• DMA for efficient L2 $\leftrightarrow$ 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

- Micro-riscy
- Zero-riscy
- RI5CY
- RI5CY+FPU
- Ariane
Different Workload? Different core

Now part of lowRISC under the «IBEX» name
Different Workload? Different core

Going to be part of 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

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

https://github.com/pulp-platform/riscv
RI5CY simplified pipeline

Instr Memory

PC gen

Instr Address

Instr Data

Align and Decompress

IF/ID

Jumps

ID/E

Branches

Decode operands fwd

EX

Read RF

Data Address

AGU

Write RF

Data Memory

Data

E WB

Instr Memory
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 Extensions: 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

<table>
<thead>
<tr>
<th>Original RISC-V</th>
<th>Misaligned Ext</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw x2, 0(x10)</td>
<td>lw x2, 2(x10)</td>
</tr>
<tr>
<td>lw x3, 4(x10)</td>
<td></td>
</tr>
<tr>
<td>slri x2, x2, 16</td>
<td></td>
</tr>
<tr>
<td>slli x3, x3, 16</td>
<td></td>
</tr>
<tr>
<td>or x3, x3, x2</td>
<td></td>
</tr>
</tbody>
</table>
• **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

\n- \texttt{lw x2, 4(x10)}
- \texttt{lw x3, 4(x12)}
- \texttt{addi x10, x10, 4}
- \texttt{addi x12, x12, 4}
- \texttt{...}
- \texttt{LOOP}

### AutoIncrement Load/Store Ext

\n- \texttt{lw x2, 4(x10!)}
- \texttt{lw x3, 4(x12!)}
- \texttt{...}
- \texttt{LOOP}
Xpulp Extensions: 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

<table>
<thead>
<tr>
<th>Original RISC-V</th>
<th>BitMan Ext</th>
</tr>
</thead>
<tbody>
<tr>
<td>mv x5, 0</td>
<td>mv x5, 0</td>
</tr>
<tr>
<td>mv x7, 0</td>
<td>mv x7, 0</td>
</tr>
<tr>
<td>mv x4, 32</td>
<td>mv x4, 32</td>
</tr>
<tr>
<td>Lstart:</td>
<td>Lstart:</td>
</tr>
<tr>
<td>andi x6, x8, 1</td>
<td>andi x6, x8, 1</td>
</tr>
<tr>
<td>add x7, x7, x6</td>
<td>add x7, x7, x6</td>
</tr>
<tr>
<td>addi x4, x4, 1</td>
<td>addi x4, x4, 1</td>
</tr>
<tr>
<td>slri x8, x8, 1</td>
<td>slri x8, x8, 1</td>
</tr>
<tr>
<td>bne x4, x5, Lstart</td>
<td>bne x4, x5, Lstart</td>
</tr>
</tbody>
</table>
Xpulp Extensions: 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

<table>
<thead>
<tr>
<th>Computation</th>
<th>add, sub, shift, avg, abs, dot product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare</td>
<td>min, max, compare</td>
</tr>
<tr>
<td>Manipulate</td>
<td>extract, pack, shuffle</td>
</tr>
</tbody>
</table>
Xpulp Extensions: packed-SIMD 2/4

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

<table>
<thead>
<tr>
<th>rs1</th>
<th>0x03</th>
<th>0x02</th>
<th>0x01</th>
<th>0x00</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs2</td>
<td>0x0D</td>
<td>0x0C</td>
<td>0x0B</td>
<td>0x0A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add rD, rs1, rs2</code></td>
<td>rD = 0x03020100 + 0x0D0C0B0A</td>
</tr>
<tr>
<td><code>add.h rD, rs1, rs2</code></td>
<td>rD[0] = 0x0100 + 0x0B0A</td>
</tr>
<tr>
<td></td>
<td>rD[1] = 0x0302 + 0x0D0C</td>
</tr>
<tr>
<td><code>add.b rD, rs1, rs2</code></td>
<td>rD[0] = 0x00 + 0x0A</td>
</tr>
<tr>
<td></td>
<td>rD[1] = 0x01 + 0x0B</td>
</tr>
<tr>
<td></td>
<td>rD[2] = 0x02 + 0x0C</td>
</tr>
<tr>
<td></td>
<td>rD[3] = 0x03 + 0x0D</td>
</tr>
</tbody>
</table>
Xpulp Extensions: 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
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

- \( \text{pv.shuffle2.b } rD, rA, rB \)
  - \( rD[0] = (rB[2]==0) \) ? \( rA:rD \) {\( rB[1:0] \)}

- With \( rX[i] = rX[(i+1)*8-1:i*8] \)
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
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
Dot Product Multiplier

- Dot Product: (half word example)

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

32 bit

32 bit

32 bit

Partial Product Generator

35:2 compressor
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 $\rightarrow$ Scalar version $>$100 instr. / output pixel
PULP core examples – RV32IMC vs RV32IMCxPulp

General code

**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:**
```
loop0:
```
```
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
```

• 2 bytes saved (X instructions not compressed)
• Number of instructions reduced (21 vs 18)
• Removed branch penalties
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  lp.setup x1,a6,stop1
p.lbu  a0,1(a3!)        p.lw   a1,4(t1!) //load 4-bytes with post inc
p.lbu  a1,1(a2!)        p.lw   a5,4(t3!)
stop1: p.mac  a5,a0,a1  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
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

<table>
<thead>
<tr>
<th>PC</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>A+4 to Imem</td>
<td>Y from Imem[A]</td>
<td>add needs value from lw → STALL</td>
<td>addr to Dmem</td>
<td>-</td>
</tr>
<tr>
<td>A+8 to Imem</td>
<td>Y from Imem[A]</td>
<td>add needs value from lw → FWD</td>
<td>Bubble</td>
<td>D from Dmem</td>
</tr>
<tr>
<td>A+12 to Imem</td>
<td>Z from Imem[A+4]</td>
<td>decode Y</td>
<td>add</td>
<td></td>
</tr>
</tbody>
</table>

//LOAD STALL
lw x10, 0x0(x2)
add x10, x10, 0x4
Jump stalls (jalr)

- Stall to break path from EX stage to Imem

<table>
<thead>
<tr>
<th>PC</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>A+4 to Imem</td>
<td>Y from Imem[A]</td>
<td>jalr needs x10 Ϩ STALL</td>
<td>mul</td>
<td>-</td>
</tr>
<tr>
<td>x10+0x4 to Imem</td>
<td>Bubble</td>
<td>Jump to x10+0x4</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

//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

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

<table>
<thead>
<tr>
<th>PC</th>
<th>Instr encoding</th>
<th>Relative jumps/branch target</th>
<th>Disassembled instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000 &lt;main&gt;:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400:  00100537</td>
<td>lui a0,0x100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>404:  ff010113</td>
<td>addi sp,sp,-16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>408:  00050513</td>
<td>mv a0,a0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40c:  00112623</td>
<td>sw ra,12(sp)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>410:  03010ef</td>
<td>jal 1440 &lt;puts&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>414:  00c10283</td>
<td>lw ra,12(sp)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>418:  00000513</td>
<td>li a0,0 # 0 &lt;_DYNAMIC&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41c:  01010113</td>
<td>addi sp,sp,16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>420:  000008067</td>
<td>ret</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Time</th>
<th>Cycles</th>
<th>PC</th>
<th>Instr.</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>18880000</td>
<td>455</td>
<td>0000008000 00c0006f</td>
<td>jal</td>
<td>x0, 12</td>
</tr>
<tr>
<td>18960000</td>
<td>457</td>
<td>0000008c 30501073</td>
<td>csrrw</td>
<td>x0, x0, 0x305</td>
</tr>
<tr>
<td>19080000</td>
<td>458</td>
<td>00000090 0000093</td>
<td>addi</td>
<td>x1, x0, 0</td>
</tr>
<tr>
<td>19090000</td>
<td>459</td>
<td>00000094 0008113</td>
<td>addi</td>
<td>x2, x1, 0</td>
</tr>
<tr>
<td>19100000</td>
<td>460</td>
<td>00000098 0000193</td>
<td>addi</td>
<td>x3, x1, 0</td>
</tr>
<tr>
<td>19110000</td>
<td>461</td>
<td>0000009c 0008213</td>
<td>addi</td>
<td>x4, x1, 0</td>
</tr>
<tr>
<td>19120000</td>
<td>462</td>
<td>000000a0 0008293</td>
<td>addi</td>
<td>x5, x1, 0</td>
</tr>
<tr>
<td>19130000</td>
<td>463</td>
<td>000000a4 0008313</td>
<td>addi</td>
<td>x6, x1, 0</td>
</tr>
<tr>
<td>19140000</td>
<td>464</td>
<td>000000a8 0008393</td>
<td>addi</td>
<td>x7, x1, 0</td>
</tr>
<tr>
<td>19150000</td>
<td>465</td>
<td>000000ac 0008413</td>
<td>addi</td>
<td>x8, x1, 0</td>
</tr>
<tr>
<td>19160000</td>
<td>466</td>
<td>000000b0 0008493</td>
<td>addi</td>
<td>x9, x1, 0</td>
</tr>
<tr>
<td>19170000</td>
<td>467</td>
<td>000000b4 0008513</td>
<td>addi</td>
<td>x10, x1, 0</td>
</tr>
</tbody>
</table>
Hybrid Logarithmic 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)
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)
Peripheral Bus

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

APB (COMING FROM HYBRID LOG INT)

\[0x1A1_00000\]
μDMA: An Autonomous I/O Subsystem
I/O requirements

46Mbit/s
320x240@25fps

3Mbit/s per channel

RF Transceivers
> 100Mbit/s

Up to 2.4GBit/s

New SD standard up to 800Mbit/s

QuadSPI up to 400Mbit/s

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)
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

1. Motivation
2. BNNs
3. Architecture
4. Results

- INPUT BUFFER
  - TP-bit

- XNOR & POPCOUNT
  - TP xnor + reduction tree to 16-bit

- POPCOUNT ACCUMULATORS
  - TP x 16-bit

- THRESHOLD
  - TP-bit

- CTRL FSM
- UCODE PROC
- REG FILE SLAVE

TP/32 x 32-bit memory master

TP-bit stream

INPUT SOURCE
WEIGHT SOURCE
OUTPUT SINK

STATIC MUXING

32-bit periph target
XNOR Neural Engine in PULPissimo

1. Motivation
2. BNNs
3. Architecture
4. Results
Operation in time

STREAMER

FEAT REGISTER

XNOR POPCOUNT

UCODE PROCESSOR

CONTROLLER REG FILE

GET FEAT GET WEIGHT GET FEAT GET WEIGHT GET WEIGHT

REG FEAT

XNOR ACCUM XNOR ACCUM XNOR ACCUM XNOR ACCUM XNOR CLEAR

UPDATE IDX UPDATE IDX UPDATE IDX

PROG
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 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
- Set, Clear, Read and Write 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)

- EVENTS FROM SYSTEM
  - GPIO
  - TIMERS
  - HWCE
  - uDMA

- EVENT DOUBLE BUFFERING

- HIGH PRI. SEL.

- MASKING UNIT
  - SOC
  - PER.
  - ACC.

- SERIALIZER

- FROM APB

- uDMA is waiting for some event – e.g. SPI starts when GPIO rises
Interrupts Source

EVENTS FROM SYSTEM
- GPIO
- TIMERS
- HWCE

EVENT FROM EVENT GENERATOR
- READY
- EVT ID
- VALID

SW EVENTS (INTC APB Registers)
- 8

DIFFERENT FROM BEFORE

MASKING UNIT and PRIORITY DECODER

PULP Core

req
id
ack
id
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
PULPissimo on GitHub

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

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

• 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
• 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
.....
```

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

I am a cool GUY
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
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