PULP Simulator and SDK

Nazareno Bruschi <nazareno.bruschi@unibo.it>
Germain Haugou <germain.haugou@iis.ee.ethz.ch>
Pulp Architecture

PULPissimo

Ext. Mem

Mem Cont

L2 Mem

RISC-V core

I/O

interconnect

DMA

Tightly Coupled Data Memory

Mem

Mem

Mem

Mem

Mem

Logarithmic Interconnect

HW ACCELERATOR

RISC-V core

RISC-V core

RISC-V core

RISC-V core

I$

I$

I$

I$
RISCY and PULP Toolchain
### GVSoc – PULP Simulator

#### Platforms

<table>
<thead>
<tr>
<th>RISC-V Cores</th>
<th>Peripherals</th>
<th>Interconnect</th>
</tr>
</thead>
<tbody>
<tr>
<td>RISC-V Cores</td>
<td>Peripherals</td>
<td>Interconnect</td>
</tr>
<tr>
<td>RI5CY 32b</td>
<td>JTAG, SPI, UART, I2S, DMA, GPIO</td>
<td>Logarithmic interconnect</td>
</tr>
<tr>
<td>Ibex 32b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Snitch 32b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ariane + Ara 64b</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Interconnect

<table>
<thead>
<tr>
<th>RISC-V Cores</th>
<th>Peripherals</th>
<th>Interconnect</th>
</tr>
</thead>
<tbody>
<tr>
<td>RI5CY 32b</td>
<td>JTAG, SPI, UART, I2S, DMA, GPIO</td>
<td>Logarithmic interconnect</td>
</tr>
<tr>
<td>Ibex 32b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Snitch 32b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ariane + Ara 64b</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Potentially could simulate every PULP Platforms, building blocks thanks to its Instruction Set Simulator based on RISC-V and PULP Extensions
GVSoC – Features

- **Virtual platform features:**
  - C++ for fast native simulation
  - *Python* for instantiation + JSON for configuration
  - Complete set of traces to see what happen

- **Timing model:**
  - Fully-event based, instances can generate events at specific time
  - Includes timing models for interconnects, DMACs, memories…
  - Performance counters for information from the execution

- **Simulation performance:**
  - Around 1MIPS simulation speed
  - Functionally aligned and calibrated with HW
  - Timing accuracy is within 10-20% of target HW

---

Many more details in gvsoc documentation
https://gvsoc.readthedocs.io/en/latest/ about the concepts (not updated for all the commands)
PULP Software Environment

Building/Compiling SDK
- JSON Files
- Python Generators
- GVSOC
- C++ Models
- C++ Engines

Compiling App
- C App Code
- PMSIS

Executing App
- GVSOC
- JSON Files
- C++ Models
- RTL Platform
- Board/FPGA
PMSIS – PULP Runtime

Native RTOS features (threading, native SPI ...)

RTOS (freertos, pulpos, zephyr)

Chip features (SPI, DMA, PMU ...)

Device features (camera ...)

PMSIS BSP
hyperram, hyperflash

PMSIS Drivers
Hyperbus, Uart, DMA, cluster ...
PULP-SDK – Directory organization

- rtos/
  - pmsis
  - pulpos

- tools/
  - gap-configs
  - gapy
  - gvsoc

- tests/

- applications/

- Contains runtime code and every functions of the software stack
- Contains configuration files, python generators, pulp runner and every gvsoc model and components
- Contains small examples to test few basic pulp features on GVSoC
- Contains relevant example applications such as MobileNetV1
PMSIS APIs – Documentation


- APIs description and functionalities are also briefly explained in header files, located in the rtos/pmsis/pmsis_api/include/pmsis
Requirements – PULP Toolchain

- PULP toolchain is available at [https://github.com/pulp-platform/pulp-riscv-gnu-toolchain](https://github.com/pulp-platform/pulp-riscv-gnu-toolchain)

- On Ubuntu 18.04 this packages should be required and you can install them with:
  
  - `$ sudo apt-get install autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev`
Installation – PULP Toolchain

To install the PULP toolchain follow these steps:

- $ git clone https://github.com/pulp-platform/pulp-riscv-gnu-toolchain
- $ cd pulp-riscv-gnu-toolchain
- $ git submodule update --init --recursive
- $ export PATH=<INSTALL_DIR>/bin:$PATH
- $ ./configure --prefix=<INSTALL_DIR> --with-arch=rv32imc --with-cmodel=medlow --enable-multilib
- $ make

Requirements – PULP-SDK

- PULP-SDK is available at https://github.com/pulp-platform/pulp-sdk

- On Ubuntu 18.04 this packages should be required and you can install them with:
  - $ sudo apt-get install -y build-essential git libftdi-dev libftdi1 doxygen python3-pip libSDL2-dev-dev curl cmake libusb-1.0-0-dev scons gtkwave libsndfile1-dev rsync autoconf automake texinfo libtool pkg-config libSDL2-ttf-dev

- You may have needed of other python3 packages and you can install them with:
  - $ pip install argcomplete pyelftools six
Installation – PULP-SDK

To install the PULP-SDK follow these steps:

- $ git clone https://github.com/pulp-platform/pulp-sdk
- $ export PULP_RISCV_GCC_TOOLCHAIN=<INSTALL_DIR>
- $ cd pulp-sdk
- $ source configs/pulp-open.sh
- $ make build

More details at https://github.com/pulp-platform/pulp-sdk
Setup the environment for GVSoC execution

- Assuming that PULP Toolchain and PULP-SDK are correctly built and installed
  - $ cd pulp-sdk/

- Setting up PULP Toolchain path
  - $ export PULP_RISCV_GCC_TOOLCHAIN=<INSTALL_DIR>

- Setting up the environment sourcing a configuration file
  - $ source configs/pulp-open.sh

- Build and compile GVSoC (if target or GVSoC is changed)
  - $ make build
Run first simple test: An Hello from PULP!

TIY:
> cd tests/hello
> make clean all run VERBOSE=1

Have a look at the Makefile

Make options

Compiler flags

Rules to make target
Run first simple test: An Hello from PULP!

```c
#include "pmsis.h"

#define CLUSTER

int test_entry()
{
    struct pi_device cluster_dev;
    struct pi_cluster_conf cl_conf;
    struct pi_cluster_task cl_task;
    int ret = test_entry();
    printf("Hello from PULP!");
    return ret;
}

int main()
{
    return pmsis_kickoff(void *test_kickoff);
}
```

Include all runtime basic functions and libraries (rtos/pulpos/common/)

An Hello from fabric controller
Run first simple test: An Hello from PULP!

```
#include "pmisi.h"

#if defined(CLUSTER)
void pe_entry(void *arg)
{
    printf("Hello from cluster_id: %d, core_id: %d\n", pi_cluster_id(), pi_core_id());
}
#endif

void cluster_entry(void *arg)
{
    pi_cl_team_fork(NUM_CORES), pe_entry, 0);
}

static int test_entry()
{
    if defined(CLUSTER)
        struct pi_device cluster_dev;
        struct pi_cluster_conf cl_conf;
        struct pi_cluster_task cl_task;
        pi_cluster_conf_init(&cl_conf);
        if (pi_cluster_open(&cl_dev, &cl_conf))
            return -1;
        pi_cluster_send_task_to_cl(&cl_dev, pi_cluster_task(&cl_task, cluster_entry, NULL));

    pi_cluster_close(&cl_dev);
} #if defined(CLUSTER)

TIY: > cd tests/hello
       > make clean all run USE_CLUSTER=1 NUM_CORES=8
```

An Hello from every core

Task fork on settable number of cores

Cluster call, offload and close
Disassembled – Show the real code

- $ cd tests/hello
- $ make clean all
- $ make dis > test.s

TIY: > cd tests/hello
     > make clean all USE_CLUSTER=1 CORES=8
     > make dis > test.s
System Traces – What is it doing?

- $ cd tests/hello
- $ make clean all run runner_args="--trace=<PATH>:log.txt"
- If <PATH>=.*, every trace will be dumped in BUILD/PULP/GCC_RISCV/log.txt file. HUGE!

<table>
<thead>
<tr>
<th>Path</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/sys/board/chip/&lt;PATH&gt;</td>
<td>Processing element, useful to see the IOs made by the core, and the instruction it executes. You can add /iss to just get instruction events</td>
</tr>
<tr>
<td>cluster/pe0</td>
<td></td>
</tr>
<tr>
<td>cluster/event_unit</td>
<td>Hardware synchronizer events, useful for debugging inter-core synchronization mechanisms</td>
</tr>
<tr>
<td>cluster/pcache</td>
<td>Shared program cache accesses</td>
</tr>
<tr>
<td>cluster/l1_ico</td>
<td>Shared L1 interconnect</td>
</tr>
<tr>
<td>cluster/l1/bankX</td>
<td>L1 memory banks (the X should be replaced by the bank number)</td>
</tr>
<tr>
<td>soc/l2</td>
<td>L2 memory accesses</td>
</tr>
<tr>
<td>cluster/dma</td>
<td>DMA events</td>
</tr>
</tbody>
</table>
Understanding GVSoC System Traces

- 4890000: 489: [/sys/board/chip/soc/cluster/pe0/insn] M 1c001252 p.sw 0, 4(a5!) a5=10000010 a5:1000000c PA:1000000c
- <timestamp> <cycles> <path> <address> <instruction> <operands> <operands info>

Where:
- <timestamp> is the timestamp of the event in picoseconds
- <cycles> is the number of cycles
- <path> is the path in the architecture where the event occurred
- <address> is the address of the instruction
- <instruction> is the instruction label
- <operands> is the part of the decoded operands
- <operands info> is giving details about the operands values and how they are used
- The timestamp is absolute. The cycle count is local to the frequency domain
Understanding GVSoC System Traces

- 4890000: 489: [/sys/board/chip/soc/cluster/pe0/insn] M 1c001252 p.sw 0, 4(a5!) a5=10000010 a5:1000000c PA:1000000c
- <timestamp> <cycles> <path> <address> <instruction> <operands> <operands info>

TIY: > cd tests/hello
    > make clean all run runner_args="--trace=insn"
A little bit complex: Vector x Vector

- Compute the partial output results in the core, taking the input data from L2 memory and then store them back in L2.
Performance counters

```c
int main()
{
    // Initialize matrix operands
    task_initMat();
    #ifdef PRINT_MATRIX
    printf("This is the Matrix A\n");
    print_matrix(A, N);
    printf("This is the Matrix B\n");
    print_matrix(B, N);
    #endif
    #ifdef STATS
    // Initialize performance counters
    PI_PERF Conf =
        1 << PI_PERF_CYCLES |
        1 << PI_PERF_INSTR;
    // Measure statistics on matrix operations
    PI_perf_reset();
    PI_perf_start();
    #else
    INIT_STATS();
    PRE_START_STATS();
    START_STATS();
    #endif
    for(int i=0; i<N; i++){
        task_A[i][i] = A[i][i];
    }
    #ifdef STATS
    PI_perf_stop();
    uint32_t instr_cnt = PI_perf_read(PI_PERF_INSTR);
    uint32_t cycles_cnt = PI_perf_read(PI_PERF_CYCLES);
    printf("Number of Instructions: \nNumber of Clocks: \nNumber of CPI: \n\n");
    instr_cnt, cycles_cnt, (float) cycles_cnt/instr_cnt;
    #else
    STOP_STATS();
    #endif
}
```

Select type of performance to measure
Start the counting
Stop and read
Performance counters

```c
int main()
{
    // initialize matrix operands
    task_initMat();

    #ifdef PRINT_MATRIX
    printf("\n\nThis is the Matrix A\n");
    printf(matrix(A, N));
    printf("\n\nThis is the Matrix B\n");
    printf(matrix(B, N));
    #endif

    #ifdef STATS
    // Initialize performance counters
    pi_perf_conf(1 << PI_PERF_CYCLES | 1 << PI_PERF_INSTR);
    // measure statistics on matrix operations
    pi_perf_reset();
    pi_perf_start();
    #else
    INIT_STATS();
    PRE_START_STATS();
    START_STATS();
    #endif

    for(int i=0; i<N;i++)
    {
        task_VectProdScalar(A[i], B, tempC, N);
    }

    #ifdef STATS
    pi_perf_stop();
    uint32_t instr_cnt = pi_perf_read(PI_PERF_INSTR);
    uint32_t cycles_cnt = pi_perf_read(PI_PERF_CYCLES);
    printf("Number of Instructions: %d\n\nNumber of Cycles: %.2f\n\nCPI: %.2f\n\n", instr_cnt, cycles_cnt, (float) cycles_cnt/instr_cnt);
    #else
    STOP_STATS();
    #endif
}
```

TIY:

- cd tests/perf/matmult
- make clean all run

Application to evaluate
Performance counters

Real chips have only 1 counter to be activated at the same time while other platforms could have one per event

```c
typedef enum {
    PI PERF_CYCLES = 17, /**< Total number of cycles (also includes the cycles where the core is sleeping). Be careful that this event is using a timer shared within the cluster, so resetting, starting or stopping it on one core will impact other cores of the same cluster. */
    PI PERF_ACTIVE_CYCLES = 0, /**< Counts the number of cycles the core was active (not sleeping). */
    PI PERF_INSTM = 1, /**< Counts the number of instructions executed. */
    PI PERF_LO_STALL = 2, /**< Number of load data hazards. */
    PI PERF_JR_STALL = 3, /**< Number of jump register data hazards. */
    PI PERF_DMISS = 4, /**< Cycles waiting for instruction fetches, i.e. number of instructions wasted due to non-ideal caching. */
    PI PERF_LD = 5, /**< Number of data memory loads executed. Misaligned accesses are counted twice. */
    PI PERF_ST = 6, /**< Number of data memory stores executed. Misaligned accesses are counted twice. */
    PI PERF_JUMP = 7, /**< Number of unconditional jumps (j, jal, jr, jalr). */
    PI PERF_BRANCH = 8, /**< Number of branches. Counts both taken and not taken branches. */
    PI PERF_BTAKEN = 9, /**< Number of taken branches. */
    PI PERF_RVC = 10, /**< Number of compressed instructions executed. */
    PI PERF_LD_EXT = 12, /**< Number of memory loads to EXT executed. Misaligned accesses are counted twice. Every non-TCOM access is considered external (cluster only). */
    PI PERF_ST_EXT = 13, /**< Number of memory stores to EXT executed. Misaligned accesses are counted twice. Every non-TCOM access is considered external (cluster only). */
    PI PERF_LD_EXT_CYC = 14, /**< Cycles used for memory loads to EXT. Every non-TCOM access is considered external (cluster only). */
    PI PERF_ST_EXT_CYC = 15, /**< Cycles used for memory stores to EXT. Every non-TCOM access is considered external (cluster only). */
    PI PERF_TCOM_CONT = 16, /**< Cycles wasted due to TCOM/log-interconnect contention (cluster only). */
} pi perf_event_e;
```
Performance counters

TIY:  > cd tests/perf/matmult
      > make clean all run VERBOSE_PERF=1

SPOILER: Load stalls are the problem of the differences between instructions and cycles!
A little bit complex: Ram Transfers

- First compute the partial output results in the core, taking the input data from L2 memory and then store them back in L2.

- Copy partial output results in HYPERRAM, going through the LIC, µDMA and then HyperBus.
Ram Transfers – Sync or Async

How can we choose between them?
- Let’s see again the performance

TIY: > cd tests/perf/double_buffering
> make clean all run DB=1
"µDMA Messages"

From the activation of the hyperbus module to the enqueuing of the first request are passed 382 µDMA cycles and 7,441 µs
TIY: > cd tests/perf/double_buffering
    > make clean all run DB=1 runner_args="--trace=soc/udma:udma.txt"
VCD Traces – More human readable

- Many more details in the gvsoc documentation
  about the concepts (not updated for all the commands)
  - `$ cd tests/perf/double_buffering`
  - `$ make clean all run runner_args="--vcd"`

- This command will create a VCD file at
  BUILD/PULP/RISCV_GCC/all.vcd and a file at
  BUILD/PULP/RISCV_GCC/view.gtkw

- Terminal will print out the command to open the latter with
  Gt KWaves
VCD Traces – More human readable

- Many more details in the gvsoc documentation https://gvsoc.readthedocs.io/en/latest/vcd_traces.html# about the concepts (not updated for all the commands)
  - $ cd tests/perf/double_buffering
  - $ make clean all run runner_args="--vcd"

TIY: > cd tests/perf/double_buffering
    > make clean all run DB=1 runner_args="--vcd"
    > gtkwave <INSTALLATION_PATH>/pulp-sdk/tests/hello/BUILD/PULP/GCC_RISCV/view.gtkw
VCD Traces - μDMA vs. CORE

That duration corresponds (at 50MHz and for 500 transitions) to about 995k cycles.

While the difference (in cycles) between both implementations is 990k (from performance counters).