Extending the RISC-V ISA for exploring advanced reconfigurable SIMD instructions

Fifth Workshop on Computer Architecture Research with RISC-V (CARRV 2021)

Philippos Papaphilippou, Paul H. J. Kelly, Wayne Luk

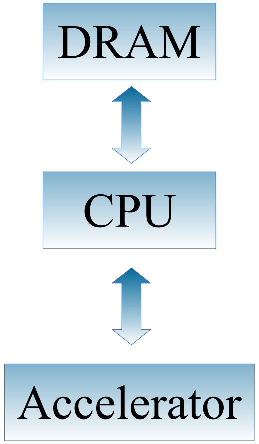
Department of Computing, Imperial College London, UK

Questions? pp616@ic.ac.uk

Source available: philippos.info/simdsoftcore

Introduction - Motivation

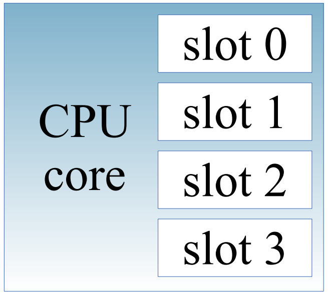
- Tradeoff between using CPUs with SIMD vs FPGAs
  - ISAs provide a fixed set of vector extensions in CPUs
    - Overspecialised instructions, bloated extensions
    - Still inefficient in certain applications
    - Increased hardware complexity
    - Relying on sophisticated high-end micro-architecture
  - FPGA are flexible, but are found in highly-heterogeneous systems
    - Main memory bottleneck for HPC: high latency, low bandwidth
    - Complicating the programming models
    - Communication logic impacts operating frequency
- A possible future path in computer architecture
  - Small FPGAs on modern CPUs working as instructions
- No easy way to explore custom SIMD instructions currently



DRAM

CPU

Accelerator



CPU core

slot 0

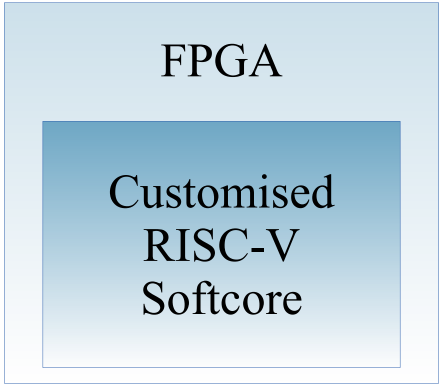
slot 1

slot 2

slot 3

Contributions

- Experimental non-standard instruction types
  - Enabling optional access to a high number of registers
  - Expressing complex SIMD tasks with fewer instructions
- An open-source softcore
  - Complete framework to evaluate novel custom SIMD instructions
  - Micro-architectural design choices to maximise streaming performance
  - Source available: `philippos.info/simdsoftcore`
- Verilog (HDL) templates for custom SIMD instructions
  - Allowing implementations of arbitrary pipeline lengths
- Evaluation of novel instructions uses cases on an FPGA for
  - Sorting
  - Prefix sum

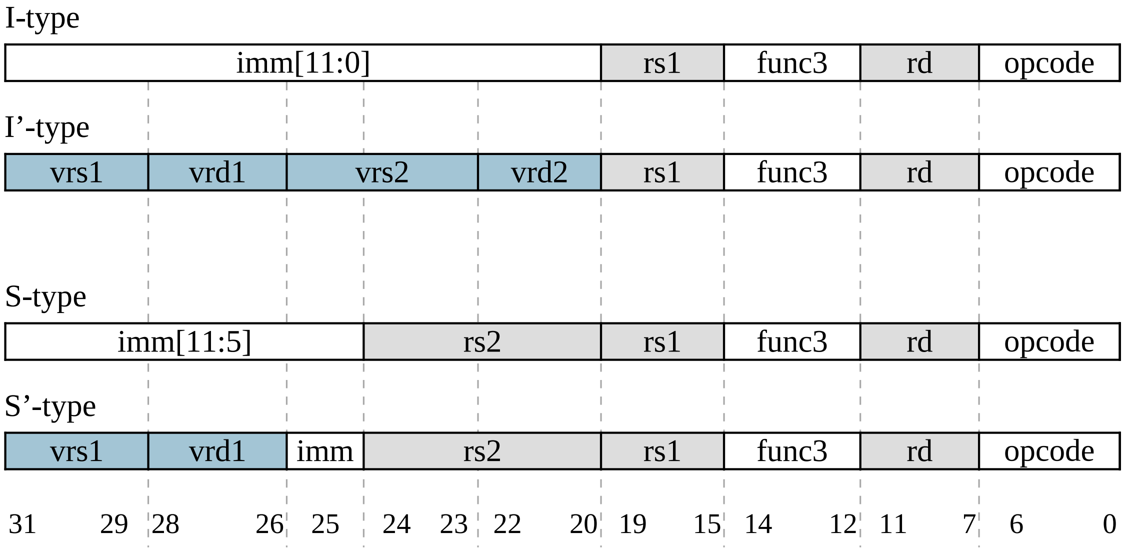


FPGA

Customised RISC-V Softcore

Custom vector instruction types

- Based on RV32I base instruction types
  - R, I, S/B, U/J
- Two new custom instruction types
  - I'-type
    - 2 vector registers for input and 2 for output
    - 1 32-bit register for input and 1 for output
  - S'-type
    - 1 vector register for input and 1 for output
    - 2 32-bit registers for inputs and 1 for output
- "V" extension not followed
  - Targeting high-end processors
    - (e.g. 32 vectors, 100s of instructions)
  - Draft state
    - Toolchain support



I-type

imm[11:0] | rs1 | func3 | rd | opcode

I'-type

vrs1 | vrd1 | vrs2 | vrd2 | rs1 | func3 | rd | opcode

S-type

imm[11:5] | rs2 | rs1 | func3 | rd | opcode

S'-type

vrs1 | vrd1 | imm | rs2 | rs1 | func3 | rd | opcode

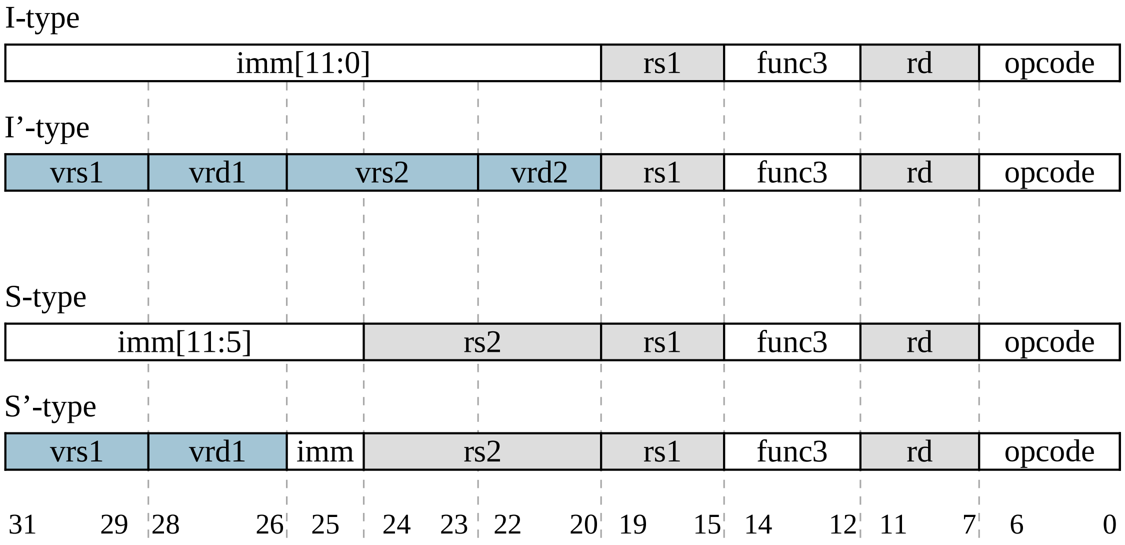
31 29 28 26 25 24 23 22 20 19 15 14 12 11 7 6 0

Naming conventions:
(v)rs: source (vector) register
(v)rd: destination (vector) register

Custom vector instruction types

- Main ideas
  - Accessing multiple register operands
    - Advanced SIMD instructions
  - Only 8 vector registers
    - Fit many operands in (32-bit) instructions
    - Reduced need for chaining
  - Vector 0 representing 0
    - Easily alias unused operands with 0
- Minimal toolchain modification
  - Using the custom instruction opcodes
    - 0x2, 0xa, 0x16, 0x1e
  - Using the immediate fields for Vector registers
    - Using pre-existing types I and S
  - Call using inline assembly
    - Example: load data of address x+y to vector register 1 in C/C++:

```
asm volatile ("c0_lv x0, %0, %1, %2":: "r"(x), "r"(y), "I"(1<<(6)) );
```



	31 29	28 26	25	24 23	22 20	19 15	14 12	11 7	6 0
I-type	imm[11:0]					rs1	func3	rd	opcode
I’-type	vrs1	vrd1	vrs2		vrd2	rs1	func3	rd	opcode
S-type	imm[11:5]			rs2		rs1	func3	rd	opcode
S’-type	vrs1	vrd1	imm	rs2		rs1	func3	rd	opcode

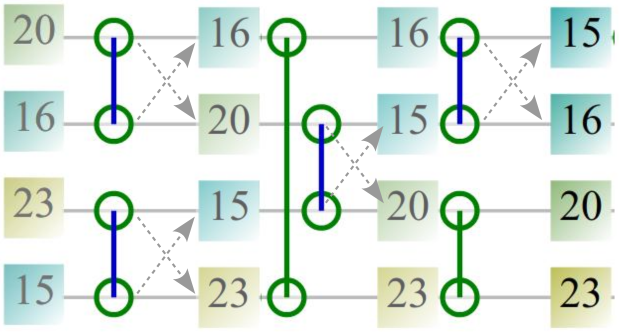
Verilog instruction templates

- Supporting arbitrary pipelines
  - Custom code after “`/// User code ///`”
- Parameters
  - Vector register width (`VLEN`)
  - Pipeline latency (`c1_cycles`)
- Template for instruction type I’
- Inputs register values:
  - `in_data`, `in_vdata1`, `in_vdata2`
- Destination register names:
  - `rd`, `vrd1`, `vrd2`
  - (will eventually reappear as output)
- Equivalents for output
  - `out_` prefix

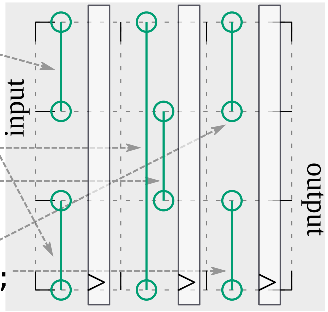
```verilog
`define XLEN 32  // 32-bit base registers
`define VLEN 128 // 128-bit vector registers
`define c1_cycles 3 // pipeline length of c1 custom instruction

module c1 [...]
    input clk, reset,  in_valid; // valid bit for input

    // Destination register names (1 base and 2 vector)
    input [4:0] rd;  input [2:0] vrd1, vrd2;

    input [`XLEN-1:0] in_data; // 32-bit input
    input [`VLEN-1:0] in_vdata1, in_vdata2; // 128-bit input

    // (Delayed) output valid bit and output register names
    output out_v;
    output [4:0] out_rd; output [2:0] out_vrd1, out_vrd2;

    output  [`XLEN-1:0] out_data; // 32-bit output
    output [`VLEN-1:0] out_vdata1, out_vdata2; // 128-bit output

    // Shift register logic to delay rd, vrd1, vrd2 and in_valid
    // by c1_cycles, to out_rd, out_vrd1, out_vrd2 and out_v resp.
    [...]
    //// User code ////
    [...]
endmodule
```

Verilog instruction templates

- Simple example: sort 4 values inside a vector register
  - 4 × 32-bit integers = 128-bit vector (for VLEN=128)
- Implementing a small bitonic sorter
  - Consisting of compare-and-swap (CAS) units: sorters of 2 elements
  - 3 pipeline stages for 3 layers, required to sort 4 elements

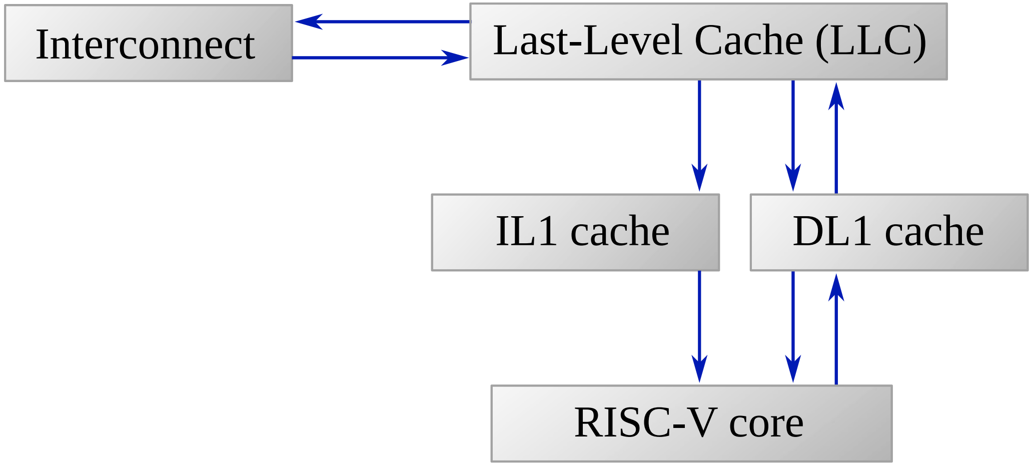


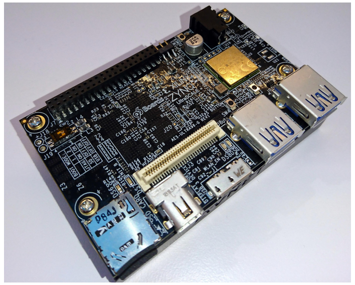
```
wire [31:0] net [15:0]; // 32-bit wires inside the sorting network
// On pos. edge: a, b -> min(a,b), max(a,b)
CAS cas0(clk, net[0], net[1], net[4], net[5]);
CAS cas1(clk, net[2], net[3], net[6], net[7]);

CAS cas2(clk, net[4], net[7], net[8], net[11]);
CAS cas3(clk, net[5], net[6], net[9], net[10]);

CAS cas4(clk, net[8], net[9], net[12], net[13]);
CAS cas5(clk, net[10], net[11], net[14], net[15]);

// Assigning input and output to wires in the sorting network
// (only using 1 input and 1 output reg. for this instruction)
for (i=0; i<4; i=i+i) assign net[i]=in_vdata1[32*(i+1)-1-:32];
assign out_vdata1={net[12], net[13], net[14], net[15]};
```



Softcore Micro-architecture

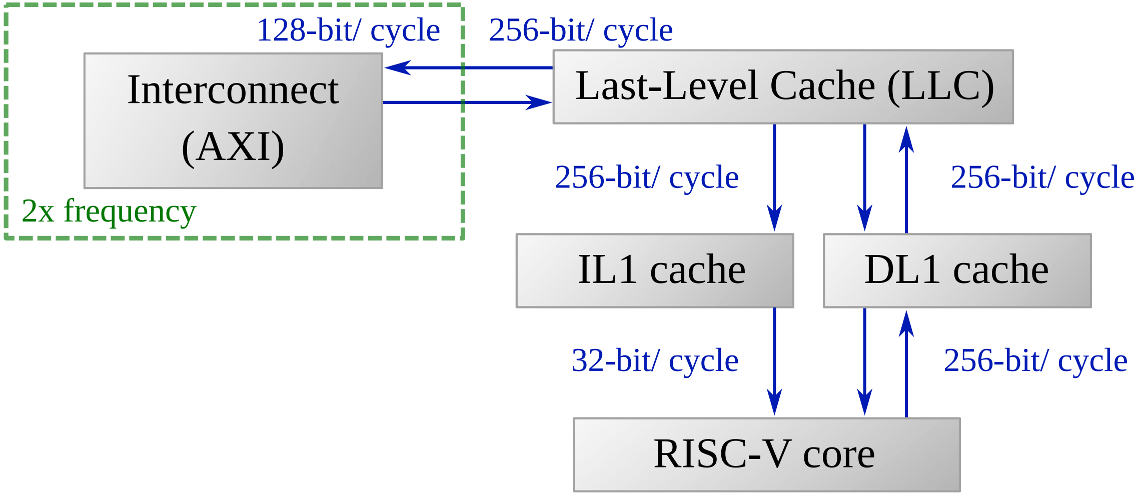
- RISC-V Core
  - Specification: RV32 I, M
  - Single pipeline stage for basic instructions
    - But separate pipelines for caches, “M”, SIMD
- Cache hierarchy
  - DL1, IL1, LLC caches
  - Traditional (modified Harvard) model
    - All caches accessing the same address space
    - Not a given in softcores
  - NRU cache replacement policy
  - Writeback policy
- Tested on Ultra96 FPGA, running at a frequency of 150 MHz
  - Sharing same DRAM as ARM A53 cores





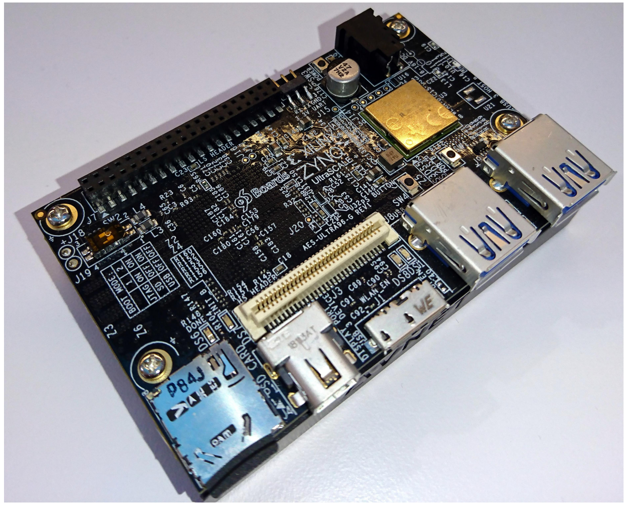
Optimisations for SIMD and streaming

- DL1 block size equal to VLEN
  - Supporting throughput for SIMD
  - No need to fetch block on SIMD writes
- Very wide LLC blocks
  - Efficient bursts to main memory
  - Localities favouring streaming pattern
- LLC strobe functionality
  - Support storing wide blocks efficiently in FPGA's block RAM
- Doubling the frequency of the interconnect (platform-specific)
  - Saturate the port bandwidth more easily with 300 MHz at 128-bit/ cycle
  - Emulating a port of double the width (VLEN)



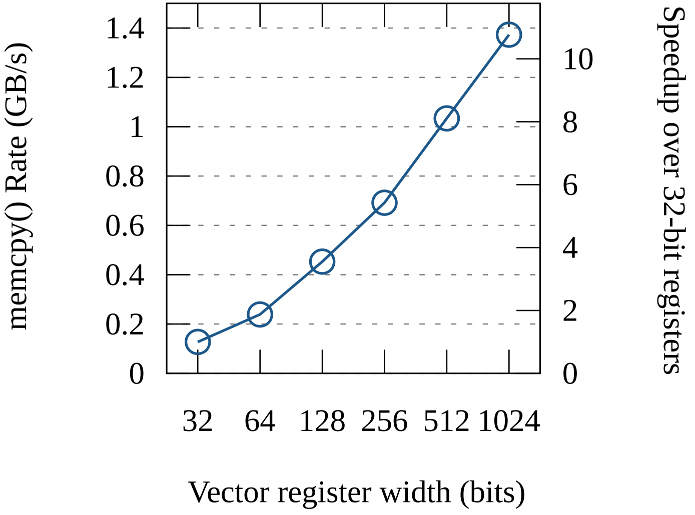
Evaluation

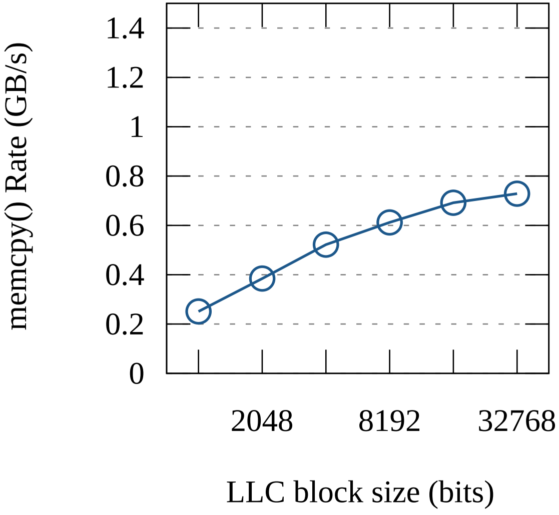
- Three goals
  - Evaluate the efficiency of the softcore implementation
  - Justify the design choices related to streaming performance
  - Explore the behaviour and efficiency of example novel custom SIMD instructions
- Evaluation platform (Ultra96 board)
  - Ultra96 board
    - Xilinx Zynq UltraScale+ device: 4 ARM cores & FPGA
    - 2 GB RAM: 1GB for ARM, 1GB for RISC-V softcore
  - Controlling the RISC-V softcore through the ARM cores running Linux



Evaluation: Design space exploration

- Parameters
  - Vector register size (VLEN) → impacting SIMD parallelism and localities
  - LLC block size → impacting burst size through the AXI bus





- Selected configuration:
  - 256-bit vector registers, 16384-bit LLC blocks

Evaluation: softcore implementation details

- Final configuration:
  - For Ultra96
  - Through parameters

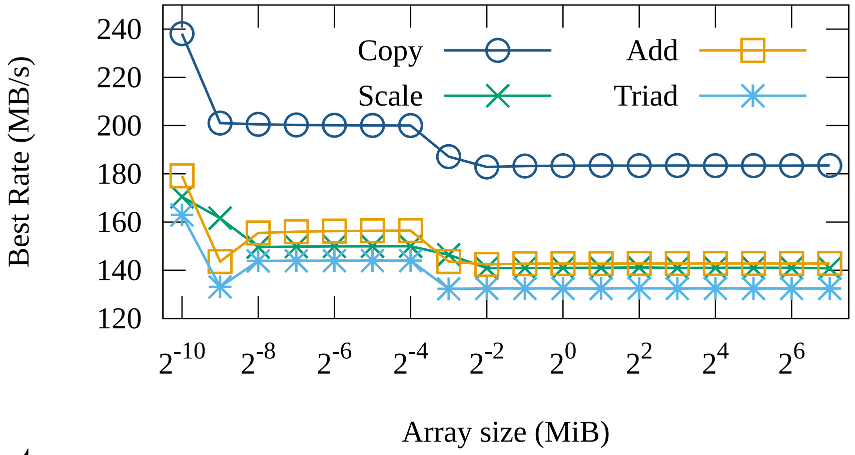
IL1		DL1			LLC				VLEN	f_{max}
sets	block (bits)	sets	ways	block (bits)	sets	ways	block (bits)	sub-blocks	(bits)	(MHz)
64	256	32	4	256	32	4	16384	32	256	150
(=2KiB)		(=4KiB)			(=256KiB)					

- Indicative performance comparison with other (non-SIMD) softcores in the literature:

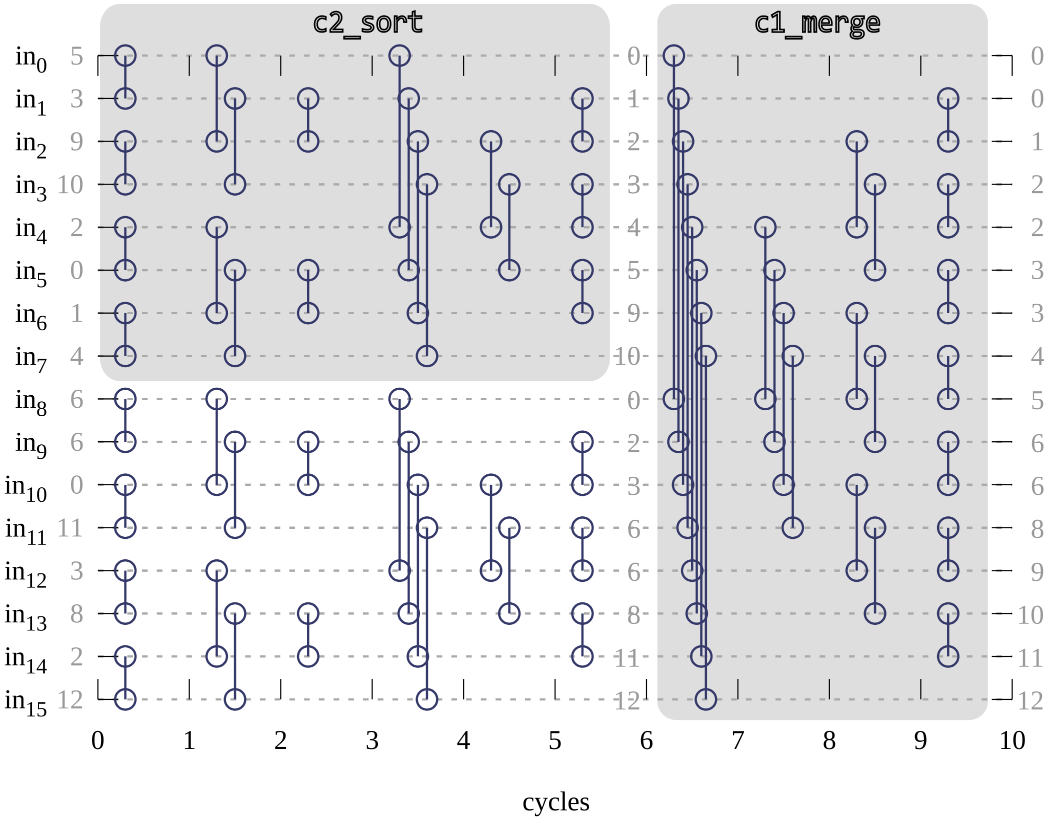
	DMIPS/MHz	Coremark/MHz	f_{max}	FPGA architecture
RVCoreP/radix-4	1.25	1.69	169	Xilinx Artix-7
RVCoreP/DSP	1.4	2.33	169	Xilinx Artix-7
PicoRV32	0.52	*N/A*	*N/A*	(simulation)
RSD/hdiv	2.04	*N/A*	95	Zynq
BOOM/hdiv	1.06	*N/A*	76	Zynq
Taiga	>1	2.53	∼200	Xilinx Virtex-7
This work	*1.47*	*2.26*	*150*	*Zynq UltraScale+*

Evaluation: streaming performance

- Evaluating communication efficiency
- STREAM benchmark suite
  - Adopted for RV32I, non-SIMD
  - 4 kernels: Copy, Add, Scale, Triad
    - Copy: 0.18 GB/s
    - SIMD Copy: 1.37 GB/S (through memcpy())
- Indicative comparison with a drop-in replacement
  - PicoRV32 working as an AXI peripheral at 300 MHz
  - STREAM best rates
    - 4.8, 3.6, 4.4, 4 MB/s for Copy, Scale, Add, Triad
  - The proposed softcore is 38x faster for copy, and 144x faster with SIMD



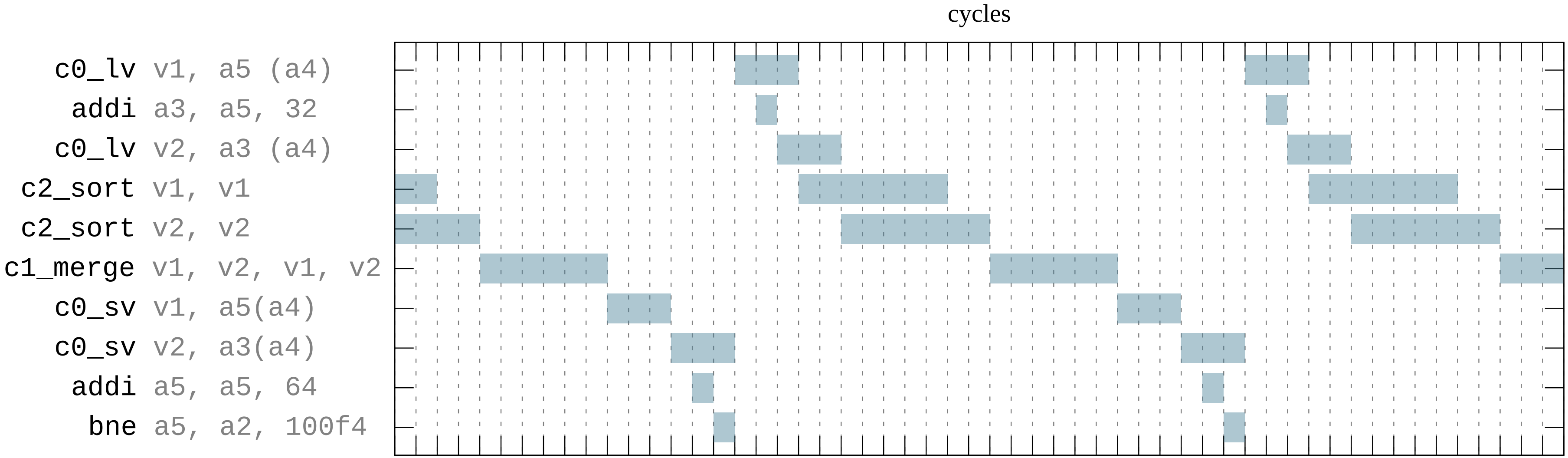
Use case 1: sorting



- Complete 32-bit sorting
  - Based on Batcher's odd-even mergesort
  - Pipelined implementations
- Two new instructions
  - `c2_sort`
    - Sort 8 × 32-bit integers (1 vector)
  - `c1_merge`
    - Merge two sorted lists of 8 × 32-bit integers (2 vectors)
    - Modification to support arbitrarily long inputs
- Algorithm phases
  - Sort all input in chunks of 16 elements
    - (2 calls of `c2_sort`, 1 call of `c1_merge`)
  - Merge hierarchically using c1_merge

Use case 1: sorting

- Sorting-in-chunks loop: execution timeline visualisation



```
  c0_lv v1, a5 (a4)
   addi a3, a5, 32
  c0_lv v2, a3 (a4)
c2_sort v1, v1
c2_sort v2, v2
c1_merge v1, v2, v1, v2
  c0_sv v1, a5(a4)
  c0_sv v2, a3(a4)
   addi a5, a5, 64
    bne a5, a2, 100f4
```

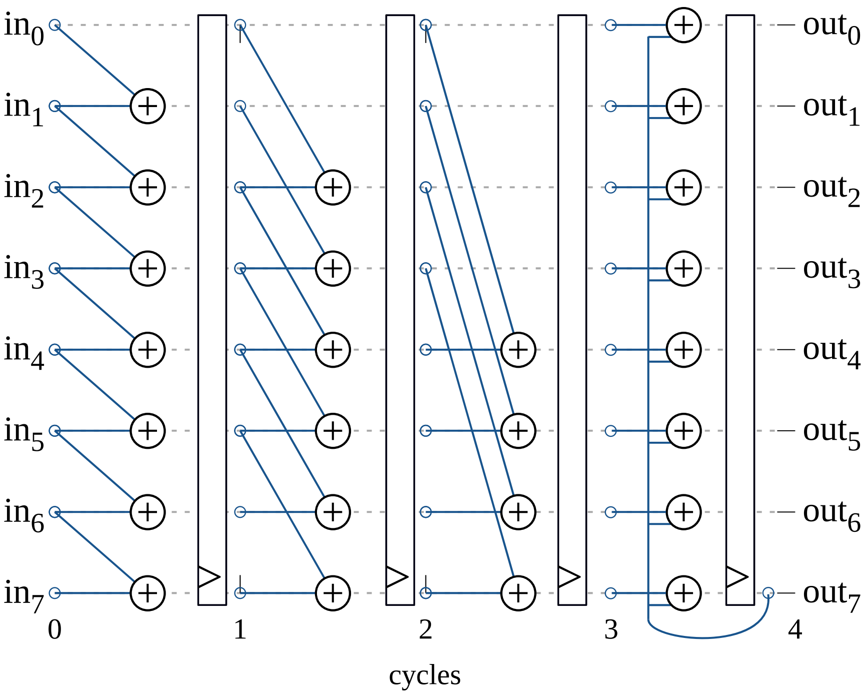
- Performance results for complete 32-bit sorting of 64 MiB of random input
  - 12.1 times speedup over qsort()
  - 1.8 times speedup over qsort() on the hardened ARM A53 core

Use case 2: prefix sum

- Prefix sum
  - Useful in many database operations
  - Serial version pseudocode:

```
sum = 0
for num in L[0, ..., N-1]:
    sum += num
    print sum
```

- Pipelined implementation
  - Based on the Hillis and Steele algorithm
  - Modified to support arbitrarily long input
- Results
  - 4.1 times speedup over the serial version



Discussion

- Comparing use case 1 - sorting to an older Intel approach [Chhugani et al. 2008]
  - Sorting 16 integers (for phase 1) over 4 integers using Intel SIMD intrinsics
    - 13x lower instruction count
    - 4.3x fewer cycles
- Ideally, the reconfigurability is left for the instructions
  - Simple instruction implementations, benefiting from higher-end CPU features
  - Potentially operating at much higher frequencies
- Challenges for reconfigurable instructions in more advanced CPUs
  - Holding states inside the instruction (and not through operands)
  - Wrong execution path, context switching support
- Holding states inside the instruction
  - Challenging, but it would be necessary for approaching FPGA-like performance
  - FPGAs benefit from on-chip memories
  - Can complicate programming models, verification etc.

Conclusions

- Contributions
  - Experimental non-standard instruction types
  - An open-source softcore for exploring custom SIMD instructions
  - Templates for implementing advanced reconfigurable instructions
  - Micro-architectural design choices to maximise streaming performance
- Small FPGAs working as custom instructions on modern CPUs can
  - Reduce instructions and cycles
  - Remove the bandwidth bottleneck in FPGAs
  - Simplify the ISA vector extensions
  - Lower the hardware complexity
- Future work
  - Further improvements and pipelining of the framework
  - Experiment with partial reconfiguration for a more modular approach
  - Elaborate on possible real-world workarounds for the internal state challenge

Thank you for your attention!

Questions?

Source available:
`philippos.info/simdsoftcore`