120-core microAptiv MIPS Overlay for the Terasic DE5-NET FPGA board

Chethan Kumar H B
chethank001@e.ntu.edu.sg
School of Computer Science and Engineering
Nanyang Technological University
Singapore

Prashant Ravi
prashant014@e.ntu.edu.sg
School of Computer Science and Engineering
Nanyang Technological University
Singapore

Gourav Modi
gourav001@e.ntu.edu.sg
School of Computer Science and Engineering
Nanyang Technological University
Singapore

Nachiket Kapre
nachiket@uwaterloo.ca
Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada.

ABSTRACT

We design a 120-core 94 MHz MIPS processor FPGA overlay interconnected with a lightweight message-passing fabric that fits on a Stratix V GX FPGA (5SGXEA7N2F45C2). We use silicon-tested RTL source code for the microAptiv MIPS processor made available under the Imagination Technologies Academic Program. We augment the processor with suitable custom instruction extensions for moving data between the cores via explicit message passing. We support these instructions with a communication scratchpad that is optimized for high throughput injection of network traffic. We also demonstrate an end-to-end proof-of-concept flow that compiles C code with suitable MIPS UDI-supported (user-defined instructions) message passing workloads and stress-test with synthetic workloads.

1. INTRODUCTION

Modern FPGAs now contain millions of LUTs, hundreds of DSP blocks and on-chip RAMs, supported by a rich, configurable, spatial interconnect fabric. However, making effective use of this raw capacity continues to be a challenge to RTL designers. Recent interest in processor-centric overlays such as the GRVI-Phalanx [1] show us how to tile hundreds of lightweight CPU cores into an FPGA with an emphasis of FPGA-centric logic mapping and optimization. Such processor arrays can be programmed in a higher-level programming environment such a C/C++ or OpenCL instead of low-level RTL. We can then focus optimization effort towards customizability and efficiency of the overlay processors. While RISC-V [2] is an interesting and useful open platform for collaborative design of processor hardware, the software ecosystem is still not mature. Existing frameworks, compilers, and development environments are substantially more mature for established ISAs such as MIPS, and ARM. For this work, we choose to use the more familiar MIPS ISA [3] with its broader adoption in the embedded markets and well-developed software ecosystem. The MIPS RTL core is made available by Imagination Technologies [4, 5] and is a microAptiv embedded implementation suitable for academic research. While other MIPS-based or MIPS-like FPGA soft processors have been demonstrated [6, 7], we use the newer MIPS3 ISA-capable release from Imagination. It is an alternative effort that complements the RISC-V projects, and we want to provide more options and variations in ISA styles for FPGA overlay research. Starting from an embedded MIPS RTL, and optimizing it for an FPGA implementation, we can make the core more attractive for FPGA overlay designers.

FPGA '17, February 22 - 24, 2017, Monterey, CA, USA

ISBN 978-1-4503-4354-1/17/02...$15.00

DOI: http://dx.doi.org/10.1145/3020078.3021751

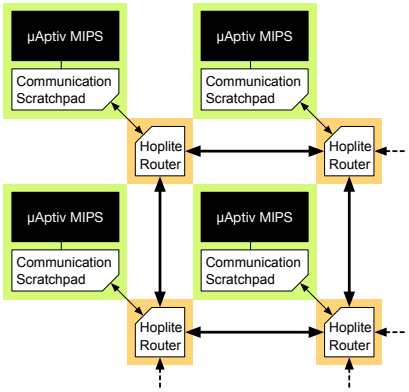


Figure 1: Multi-core μaptiv MIPS arrays interconnected with a Hoplite NoC. RTL modified to support message-passing instructions that do not use cache coherency. Each processor core has separate memory space (scratchpad) dedicated for inter-core communication. Physical implementation merges this with the Register File for efficient use of M20Ks.

In this paper, we make the following contributions to improve the MIPS design for FPGA compatibility:

- **FPGA-centric optimization**: While the original release from Imagination Technologies is silicon tested, it is meant for ASIC substrates without optimizations for FPGA features. We reduce FPGA logic utilization of the processor core to make it suitable for FPGA tiling as well as help improve clock frequency. We quantify the extent of these improvements by measuring resource utilization and frequency of the original MIPS RTL and our optimized solution.
- **Message-Passing support**: Modern FPGAs are wire-rich substrates and communication-centric design is important to expose FPGA capacity to developers. We augment the core with custom instructions and a communication scratchpad to support message-passing functionality. We interface the core with the lightweight Hoplite [8] NoC router. We demonstrate and programming flow to using these instructions in a high-level language such as C. We also simulate RTL for the complete system under various synthetically-generated instruction sequences to stress-test the communication mechanisms.

2. MESSAGE-PASSING MIPS DESIGN

2.1 FPGA-centric optimization

The original MIPS RTL released by Imagination Technologies is a silicon-tested design that has not been optimized for an FPGA implementation. Hence, we first understand the architecture details of the MIPS processor design and identify opportunities for a better matching to the FPGA fabric. The original μaptiv MIPSfpga [5] core occupies 8 K ALMs per core on a DE5-NET board along with 17 M20K RAM blocks while operating at 80 MHz clock frequency. This large footprint makes the core too large for tiling on top of an FPGA device. For instance, we can only fit ≈30 cores on the DE5-NET board.

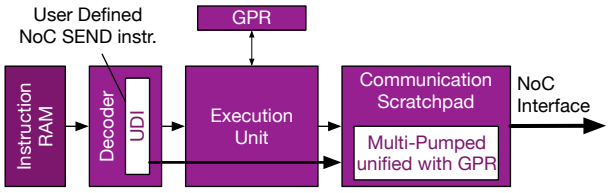


Figure 2: Modifications to MIPS processor pipeline. The instruction and data cache hierarchies have been removed and replaced with Instruction and Data scratchpads. Multiplication unit has been modified to use the Altera DSP block. NoC-specific modifications made to the decoder, along with the addition of a communication-scratchpad.

We make significant structural changes to the RTL to enable lighter weight implementation. The MIPS micro-architecture (with modifications) is shown in Figure 2. The key building blocks of the processor are the ALU (Execution Unit), the Instruction RAM and decoder, along with the general-purpose Register File (GPR). The baseline design uses a cache hierarchy which is not directly suitable for an FPGA overlay. The register file and arithmetic blocks are not optimized in any way to use the M20Ks or the hard DSP blocks available on the FPGA chip.

- First, we dismantle the complex memory hierarchy that connects the 256 KB RAM through the AHB (AMBA High-Performance Bus) interface into caches inside the core with a flat organization. We directly load the program code into an on-chip Instruction RAM that replaces the expensive cache controllers with single-cycle implementations to drive instructions directly into the decoder. We also replace the data cache with a scratchpad that provide direct physical address space for data storage.

- The instruction cache interface that is included in the MIPSfpga release does not support instruction prefetching which results in very poor IPC rates. In fact, we observed rates as low as 0.25 due to a single allowable fetch every four cycles. The use of single-cycle RAMs to store small MIPS program code inside the core, and modifications to the instruction issue logic eliminate this bottleneck as well.

- Next, we remove unnecessary and unused portions of the RTL including the JTAG interfaces, co-processor connectors, and other peripheral logic that is not directly relevant for correct operation.

- Furthermore, we provide Quartus compiler options to encourage use of DSP units for mapping ALU operations to further reduce area utilization per core.

Next, we show a visual breakdown of resource usage of the unoptimized MIPS processor in Figure 3. From this perspective, it is clear that the cache managers and TLB units occupy bulk of the resource of the core. A leaner core with a flat Instruction+Data RAM would greatly simplify design logic. We can use multi-ported M20K RAM units to implement the register file for tighter implementation. Additionally, we can remap the multiplication logic to use DSP units on the FPGA.

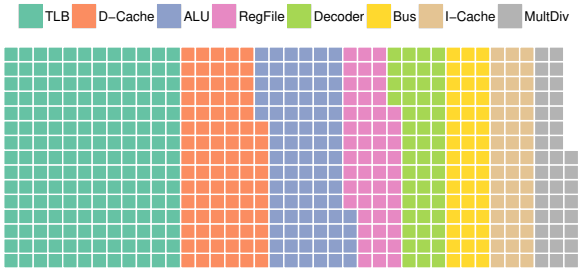


Figure 3: Breakdown of resource usage for the unoptimized MIPS microAptiv core.

We reduce resource usage in the different components of the MIPS organization while retaining functional correctness. We highlight the specific savings per block below:

1. **Execution Datapath (EDP)**: The EDP block consists of the Execution Unit, the Instruction Decoder and the System Co-Processor interface units. The Multiply-Divide Unit (MDU) is connected to the EDP for supporting larger and complicated multiplication and division operations. In adapting this block, we

first redesign the MDU unit, to map to hard DSP blocks on the FPGA to implement signed and unsigned multiplication and fused multiplication-addition operations. The original Verilog used Booth-encoded multiplication directly implemented using logic equations that map to FPGA LUTs. We also removed hardware support for division. A critical modification to the EDP is the adaptation of instruction address generation and interface to the instruction cache. We replace the cache with a simple RAM to hold program code and modify the instruction decode pathway to support this new RAM-based configuration.

2. **Memory Management Unit (MMU) and Translation Lookaside Buffer (TLB)**: The MMU block converts virtual address generated from load and store instructions into physical addresses that map to the global memory space of the system. There are Instruction and Data Caches connected to the MMU. In our simplified MIPS organization, we completely remove the MMU and rely solely on direct physical addressing. This simplification accounts for bulk of the resource utilization reductions as complicated TLB (translation lookaside buffer) and associated circuitry is eliminated. This does mean that we cannot run a full-blown OS or arbitrary code on this MIPS processor. Instead, we have to write C code carefully tailored to this system that is more amenable to coarse-grained parallelization. This simplification also removes the cache controllers, but to retain functional correctness in the instruction decoder, we have to keep certain signal generation logic intact.

3. **Bus Interface Unit (BIU)**: The external peripherals and memory components are connected to the MIPS core through the AHB bus interface. Again, while we can ideally remove this component when tiling the MIPS cores across the FPGA fabric, we had to retain some small portions of this component to ensure the design still operated correctly.

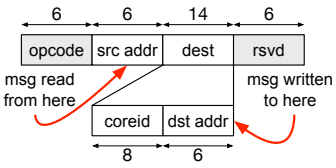
4. **Register File (RF)**: The original register file was implemented directly in FFs to support multi-ported operation. To reduce this cost, we replaced the design with a functionally identical and cycle-compatible multi-pumped [9] M20K-based block with two read ports and one write port. This also resized the RF to hold 512 32b registers.

2.2 Message-Passing Support

Once we have pruned the design, we now integrate communication oriented modifications into the design taking care not to inflate the design size too much. As shown in Figure 2, we modify the μaptiv MIPS RTL to support message-passing instructions. To do this, we have to modify the decoder to interpret the instructions correctly which is achieved through a minor modification of the decoder RTL. We also provide a separate communication-specific address space in the form of scratchpad RAMs. These allow a clean separation between the local address space of the running code, and the address space reserved for global visibility through inter-processor communication. This acts as an extension of the register file dedicated to message passing state and is physically mapped to the same M20K as the Register File. As communication operations do not interfere with register operations, the number of M20K ports is sufficient to accommodate communication operations along with register operations.

Similar to external memory operations, all interaction with a shared medium such as a NoC may take a variable number of cycles that change with the load on the medium. For instance, when the NoC is congested, a processor may be unable to inject a message into the NoC right away and has to stall. Our decoder RTL has been suitably modified to detect network readiness to determine whether the NoC instruction can execute right away or whether the processor needs to be stalled. A simple implementation is a blocking call that will stall the processor. We support a FIFO-based implementation that queues multiple NoC operations (up to a limit) before the processor must be stalled. At the receiving end of the message, the dedicated multi-pumped M20K port into the communication scratchpad ensure stall-free message receipt.

We use a simple push-based message-passing mechanism that allows communicating processors to perform **write** operations to remote scratchpads. The address space of the communication scratchpads is globally visible, and uniquely addressable from all MIPS cores. We do not yet support request-reply style traffic (*i.e.* remote **read** operations) but expect to do so in the near future through parallel physical request network. Without request-reply style traffic, our NoC does not allow application level deadlock. When we eventually support requests on a separate physical track, the absence of cycles on the same channel will continue to ensure deadlock-free operation. Despite this constraint, the protocol is sufficiently rich to allow us to express bulk synchronous and token dataflow workloads. We use the MIPS **UDI** (User-Defined Instruction) extension to configure the send operation. This allows us to fully customize the interpretation of the certain instruction bits to our application requirements. In our case, we customize the instruction to support NoC operation. The exact interpretation and encoding of 32 bit UDI instruction is shown here.



The instruction uses immediate data to encode the source and destination operands of the message to be transmitted over the NoC. Given the 20b field of user-customizable bits in the instruction encoding, we currently support 256 cores (8b address) and 64 scratchpad locations per core (2×6b address).

2.3 Software Support for Message-Passing

We use a ASM intrinsics for supplying message passing instructions directly from the user program in C. Unlike [1], we are capable of executing code produced by the `mips-gcc` toolchain[1]. We use the MIPS *User-Defined Instruction* (UDI) feature to build a simple wrapper for supplying a message-passing workload directly to the C compiler. In Listing 1, we show a sample series of UDI instructions in a C program directly supplying immediate data for the custom `SEND` instruction. The first argument specifies the choice of UDI command (in this case just `SEND`), and the second argument provides the 20b of message passing state including

[1]subject to instruction and code size limits

the different fields shown in the instruction breakdown earlier (source address, destination MIPS core index, destination address). Note that the UDI start/end instructions are a marker to configure the M20K ports for NoC access, and should be used once for a batch of NoC packets instead of each individual instruction. In addition, we support a global synchronization function that allows all processors to synchronize before safely accessing received data in their communication scratchpads. The reserved bits can be used to either support more than 256 processors, or encode additional priority information to help improve end-to-end packet latencies from important data.

We expect this software API to be useful for bulk synchronous applications that naturally support push-style message passing. Operations or traversals on irregular datastructures such as graphs would be much faster through direct point-to-point messages. It is also possible to build a streaming API using the non-blocking formulation. By adjusting the FIFO depths to application needs, high throughputs can be supported. Token dataflow problems naturally support a message-propagation down a dataflow graph. These are also naturally expressed as send operations.

```
#include <stdio.h>
#include "udi.h"

int main()  {
  mips_udi_i_nv(0x09,0x00000); // UDI Start
  mips_udi_i_nv(0x08,0xFFFFF); // SEND message
  mips_udi_i_nv(0x0A,0x00000); // UDI END
  return 0;
}
```

```
mips-mti-elf-gcc -march=m14kc -c mips.c -o mips.o
```

Listing 1: Code sketch showing example of inline UDI `SEND` instruction. Also showing a one-line `gcc` compile command to target the μaptiv MIPS processor.

We show a concrete application example of using our UDI SEND instruction in Listing 2. This is a code sketch for parallel sparse matrix-vector multiply computation operating on integer data. The same code template runs on all cores but on different portions of the matrix. In our implementation, we parallelize the computation by partitioning the matrix across the MIPS cores, assigning a subset of rows to each core. We store the sparse matrix A in a compressed sparse row (CSR) format. The structures *a_index* and *col_index* are used to locate non-zero entries in the matrix A with the actual values stored as an array *a_nz*. The algorithm is iterative, and requires calculation of $b = A \cdot x$ multiple times. The result vector b is processed to build the vector x for the next iteration. Within each core, we loop over the rows m, and process each non-zero entry in A to accumulate the row dot-product result in $b[m]$. Once we compute b, we distribute the results to other cores that will need this in the next iteration using the fanout addressing structure *s_index* and *invcol_index*. We use our UDI SEND primitive for quick direct message transfer for b vector values. On a conventional CPU architecture, this would require relying on cache coherency to enforce ordering. On GPUs, we would need to synchronize data through the off-chip DRAM instead. Unlike both these solution, our approach provides an explicit way to perform data transfer over on-chip NoCs.

```
#include <stdio.h>
#include "udi.h"

int main()  {
  // Define SpMV structures
  int *a_index, *col_index,
  int *s_index, *invcol_index;
  int *a_nz, *b, *x;
  int M, m, src, dest, pkt;
  // Loop over non-zeros, to compute Ax=b
  for(m=0; m<M; m++) {
    b[m]=0;
    int edges = a_index[m+1]-a_index[m];
    for(n=0; n<edges; n++) {
      b[m] += a_nz[a_index[m]+n]
        *x[col_index[a_index[m]+n]];
    }
  }
  // Send vector values to cores that need it
  mips_udi_i_nv(0x09,0x00000); // UDI Start
  for(int m=0; m<M; m++) {
    src = &b[m];
    int edges = s_index[m+1]-s_index[m];
    pkt = src<<14;
    for(int n=0; n<edges; n++) {
      // construct packet
      dest = &x[invcol_index[s_index[m]+n]];
      pkt |= dest<<5;
      // send packet
      mips_udi_i_nv(0x08,pkt);
    }
  }
  mips_udi_i_nv(0x0A,0x00000); // UDI End
}
```

Listing 2: Sparse Matrix-Vector Multiplication code sketch with UDI `SEND` instructions for b vector movement.

3. RESULTS

We map the MIPS processor overlay on the DE5-NET FPGA board. We are able to fit a 120-core implementation after applying optimizations described in Section 2.1. We floorplan the design as shown in Figure 4 to constrain the processors into small wide rectangular regions of the chip. We organize them into two columns to mimic the physical structure of the two DSP columns available on the chip. This columnar structure is highlighted in Figure 5. The DSP block alignment on this chip restricts us into having a X×2 design which can be modified for other chips as per their physical geometry. The careful layout allows the design to run close to the critical path the MIPS execution paths (94 MHz) rather than the inter-MIPS links (200 MHz+). A closer inspection of the timing report shows long combinational paths in the MIPS decoders. While this is relatively easy to pipeline, this has downstream impact on the timings of register accesses, and other dependent operations. We have chosen not to overhaul the delicate silicon-verified pipelines for this work, but seek to identify ways to simplify this critical path as part of future work. Our 120-core implementation is close to exhausting the ALMs on the chip.

In Table 1, we quantify the FPGA logic utilization breakdown of the various constituent blocks of the microAptiv core and show the effect of our optimizations aimed at reducing implementation costs and make it more amenable to lightweight FPGA implementation. A large portion of the savings come from eliminating the Caches (1.7 K ALMs) and

Table 1: FPGA Resource Utilization before and after optimization using Quartus 16.0 Prime Standard Edition on Stratix V FPGA (5SGXEA7N2F45C2) on the Terasic DE5-Net FPGA card.

Module	Before				After			
	ALMs	FFs	DSPs	M20Ks	ALMs	FFs	DSPs	M20Ks
MMU `mmu`	1771	2499	0	0	0	0	0	0
Execution Datapath `m14k_edp`	1211	347	0	0	932	345	0	0
Co-Processor `m14k_cpz`	963	1112	0	0	1406	781	0	0
Decoder `m14k_mpc`	602	464	0	0	462	325	0	0
Multiply-Divide Unit `m14k_mdunit`	310	214	0	0	101	71	4	0
I$ + Ctrl. `m14k_icc`	428	402	0	4	70	93	0	0
D$ + Ctrl. `m14k_dcc`	970	784	0	13	30	36	0	0
Reg. File `m14k_rf_reg`	602	1138	0	0	56	105	0	1
Bus I/F Unit `m14k_biu`	614	716	0	0	21	45	0	0
UDI `m14k_udi`	0	0	0	0	104	141	0	1
Total `m14k_top`	8144[1]	8430	0	17	2436	2093	2	3[2]

[1]extra ALMs due to other blocks not reported here such as `m14k_siu`, `m14k_ahb`. [2]extra M20K due to `m14k_ahb` block with Instruction RAM.

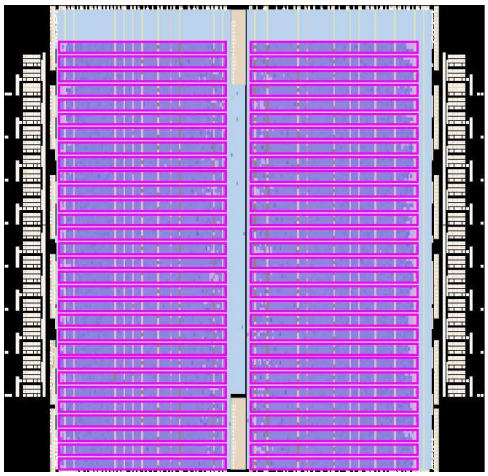


Figure 4: Logilock floorplanning constraints for 64-core design on DE5-NET FPGA board.

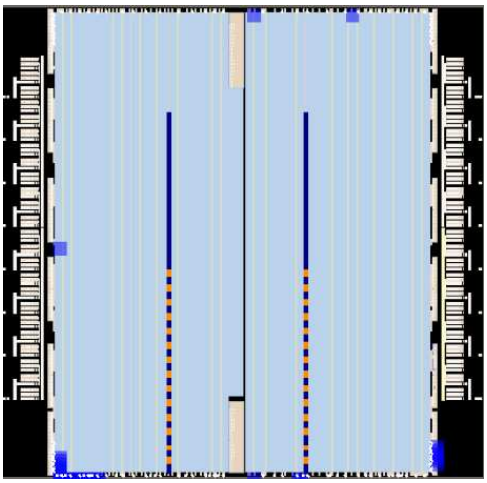


Figure 5: Logilock floorplanning constraints for 64-core design on DE5-NET FPGA board (DSP block columns highlighted in orange and blue).

Caches (1.3 K ALMs). We also efficiently pack the multi-ported Register File to multi-pumped M20K block thereby saving ≈500 ALMs. The newly added NoC interface and instructions consume 300–400 extra ALMs. Overall, we still save 3.3× the resource requirements of the original unoptimized MIPS design.

To verify correctness of the NoC instructions and connectivity, we perform Modelsim simulations of the 120-core MIPS overlay with assembly-level MIPS code generated for various traffic patterns. This also helps us contrast the out-of-the-box performance of the MIPS RTL implementations from Imagination Technologies against our FPGA-optimized configuration. In particular, our modifications to the memory hierarchy and instruction issue logic are crucial to deliver high NoC packet rates. As indicated earlier, the raw MIPS RTL only supports injection rates below 25% due to limits of the instruction issue logic. We modify this logic and streamline the design of the NoC instruction decoding to permit 100% injection rates if so desired (limited by NoC congestion). We simulated the MIPS processor under synthetically-generated traffic patterns and observed similar NoC bandwidth and latency trends as the original Hoplite [8] design.

4. RELATED WORK

In this section we review a prior work on soft processor overlays for communicating applications:

Heracles [10] is an integer-based MIPS-III soft processor array mapped to a Virtex-5 LX330T FPGA board. The multiple cores can communicate with a NoC but retains caches (unlike our design), resulting in an implementation that requires 5.5K LUTs, 2.6K FFs, 75 BlockRAMs and runs at 155 MHz. Furthermore, their virtual-channel based router alone takes 2K LUTs, 2.8K FFs, and runs at 71 MHz. They can fit a 4×4 system on the LX330T board, while our design can scale to 120 cores on the newer, larger DE5-NET board.

The recent GRVI-Phalanx [1] offers a different approach based on RISC-V instead of MIPS instruction sets. The design is extremely lightweight and uses a clustered approach for tight integration of multiple ALUs per Hoplite NoC router. This concentration of ALUs per NoC router is restrictive to applications that demand non-local, intense communication between the parallel elements. Our approach provides a NoC router per MIPS core targeting more irregular problems than the GRVI-Phalanx design. Regardless, the GRVI-Phalanx architecture is a great example of ex-

ploiting FPGA features to the fullest extent possible for accelerator integration.

The Adapteva Epiphany [11] ASIC is a 16-core floating-point RISC processor with local scratchpads (no caches) that is supported by NoCs for inter-processor and off-chip communication. The Epiphany processors uses a bespoke RISC instruction set that is incompatible with any standard (industrial or open-source). Our MIPS overlay is a configurable FPGA design that runs an industry-standard MIPS instructions set. We do not currently support floating-point but that can be added easily if required. Furthermore, we are able to use a single NoC for all communication needs by supporting a `SEND`-only message-passing paradigm. Despite the differences, the programming models of our overlay is very similar to the flat addressing mode of communicating memory elements on the Epiphany chip.

Our prior work [12, 13, 14] describes various approaches for developing overlay FPGA designs supported by NoCs for accelerating the SPICE circuit simulator. In these studies we develop customized VLIW, Dataflow, and Streaming overlays for various phases of SPICE but rely on fast integration with a communicating NoC in all cases. In this paper, we modify a general-purpose MIPS core instead of custom datapaths to support fast communication. This is a harder challenge as the instruction pipelines and memory interfaces are dictated by the MIPS compute organization and offers a constrained space of modification opportunities.

5. CONCLUSIONS AND FUTURE WORK

We implement a 94 MHz 120-core MIPS overlay to the large Stratix V GX FPGA device (5SGXEA7N2F45C2) supported by a lightweight message-passing NoC. We augment the MIPS processor with UDI-based modifications to the decoder and a separate communication scratchpad. This modified hardware can be programmed directly in C using a message-passing API layer with MIPS UDI instrinsics. We use the MIPSfpga RTL released under the Imagination Technologies Academic Program as a starting point for our work. One of our key contributions is the FPGA-specific optimization that significantly prunes the implementation cost by 3–4× by dismantling the complex cache-based memory hierarchy and using the M20K RAM blocks and DSPs wherever possible. We improve permissible injection rates from 25% to 100% through suitable adaptation of the instruction issue logic.

As part of future work we will continue to prune logic utilization of the microAptiv core by customizing a MCU (microcontroller) version of the design. This is less capable and lower performance core, but may provide a better MIPS/LUT efficiency that the GRVI-Phalanx [1] design is able to achieve. The Verilog RTL released by Imagination Technologies is not particularly easy to modify from due to complex ASIC-specific optimizations. Thus, we were sometimes forced to retain certain signals and small code blocks that were vital for correct operation but unnecessary from the system-level perspective. There are still sufficient opportunities for pruning resource costs with a deeper study of the micro-architecture. We aim to boost operating frequencies which are currently bottlenecked in the execution pipeline through appropriate retiming of the design while being careful to retain functional correctness. We also wish to further streamline the NoC interface through clustering and hierarchy to minimize resource overheads. We also want to support request-reply style NoC traffic to enable pull-style message passing in addition to current `SEND` instruction support. This is possible by instantiating a physically distinct request network that naturally helps avoid deadlock by isolating requests from replies on separate physical networks.

6. REFERENCES

[1] J. Gray, "GRVI phalanx: A massively parallel RISC-V FPGA accelerator accelerator," *CoRR*, vol. abs/1606.01037, 2016. [Online]. Available: http://arxiv.org/abs/1606.01037

[2] K. Asanović and D. A. Patterson, "Instruction Sets Should Be Free: The Case For RISC-V," 2014.

[3] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross, F. Baskett, and J. Gill, "MIPS: A Microprocessor Architecture," *SIGMICRO Newsl.*, vol. 13, no. 4, pp. 17–22, Oct. 1982. [Online]. Available: http://dl.acm.org/citation.cfm?id=1014194.800930

[4] S. L. Harris, R. Owen, E. Sedano, and D. C. Martinez, "Mipsfpga: Hands-on learning on a commercial soft-core," in *2016 11th European Workshop on Microelectronics Education*, May 2016, pp. 1–5.

[5] R. Owen, "Mipsfpga university program," https://community.imgtec.com/university/.

[6] H. Nakatsuka, Y. Tanaka, T. V. Chu, S. Takamaeda-Yamazaki, and K. Kise, "Ultrasmall: The smallest mips soft processor," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2014, pp. 1–4.

[7] S. Moore and G. Chadwick, "The TIGER-MIPS processor," 2011.

[8] N. Kapre and J. Gray, "Hoplite: Building austere overlay nocs for FPGAs," in *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, Sept 2015, pp. 1–8.

[9] C. E. LaForest and J. G. Steffan, "Efficient Multi-ported Memories for FPGAs," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '10. New York, NY, USA: ACM, 2010, pp. 41–50. [Online]. Available: http://doi.acm.org/10.1145/1723112.1723122

[10] M. A. Kinsy, M. Pellauer, and S. Devadas, "Heracles: Fully synthesizable parameterized mips-based multicore system," in *2011 21st International Conference on Field Programmable Logic and Applications*, Sept 2011, pp. 356–362.

[11] L. Gwennap, "Adapteva: More flops, less watts," *Microprocessor Report*, vol. 6, no. 13, pp. 11–02, 2011.

[12] N. Kapre and A. DeHon, "Accelerating SPICE Model-Evaluation using FPGAs," in *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, April 2009, pp. 37–44.

[13] ——, "Parallelizing sparse Matrix Solve for SPICE circuit simulation using FPGAs," in *2009 International Conference on Field-Programmable Technology*, Dec 2009, pp. 190–198.

[14] ——, "VLIW-SCORE: Beyond C for sequential control of SPICE FPGA acceleration," in *2011 International Conference on Field-Programmable Technology*, Dec 2011, pp. 1–9.