# Tuning Instruction Customisation for Reconfigurable System-on-Chip

Chun Hok Ho[1], Wayne Luk[1], Jakub M. Szefer[2] and Ruby B. Lee[2]
[1]Department of Computing, Imperial College London, United Kingdom
[2]Department of Electrical Engineering, Princeton University, USA

**ABSTRACT**
This paper describes four techniques for tuning instruction customisation for reconfigurable system-on-chip (SoC) devices. These techniques involve tuning custom instruction granularity, tuning custom instruction hardware, tuning based on run-time information, and instrumentation for tuning analysis. The proposed approach has been used in deriving custom instructions for advanced bit manipulation applications for the MicroBlaze processor. We show that for a transfer coding application, custom instructions with an increase of 23% in area can improve performance by 13 times.

## I. Introduction

Advances in reconfigurable technology make it increasingly attractive for system-on-chip (SoC) applications. Such reconfigurable SoC devices, typically containing one or more instruction processors, enable post-fabrication customisation resulting in significant cost reduction, since fabrication costs grow exponentially. Moreover, many reconfigurable IP (Intellectual Property) blocks are becoming available, which reduces development cost by making complex designs reusable for different applications.

One drawback of reconfigurable SoC devices is performance. Vendor-supplied instruction processors for these devices, such as Nios [1] and MicroBlaze [9], have basic instruction sets, which result in small area but low performance. One way of improving performance is to find IP blocks for demanding computations for particular applications, which can then be included as custom instructions for these applications. Promising candidates for such instruction customisation can be obtained from experience in conventional microprocessors, such as the MMX instructions for media processing [3] and the parallel extract (pex) and parallel deposit (pdep) instructions for applications involving complex bit-level processing [5].

This paper presents techniques for tuning instruction customisation for reconfigurable SoC devices, so that the resulting design meets requirements in area, speed, flexibility and cost. In particular, our approach covers:
1) techniques for tuning custom instruction granularity, tuning custom instruction hardware, tuning based on run-time information, and instrumentation for tuning analysis (Section III);
2) application of these techniques for advanced bit manipulation instructions and multimedia extension instructions targeting the Xilinx MicroBlaze processor (Section IV);
3) evaluation of the proposed approach, showing that an increase of 23% in area can result in performance improvement of 13 times (Section V).

It is worth noting that, in contrast to other approaches which report results based on synthesis estimates [2] and simulation [5], our results are based on measurements of actual reconfigurable SoC implementations on the XUP development system [10].

## II. Related work

Many reconfigurable SoCs are based on advanced Field Programmable Gate Array (FPGA) devices. A convenient way of executing programs on these devices is to configure them to include one or more instruction processors. The instruction set of most of these instruction processors, such as MicroBlaze and Nios, is basic; moderately complex programs can take a long time to execute. While most microprocessors from Intel and AMD include instructions for multimedia or streaming applications, such instructions are typically absent from instruction processors for reconfigurable SoCs.

Fortunately, some of these reconfigurable SoCs provide an interface to IP cores that can be invoked by the use of custom instructions. Both MicroBlaze and Nios support custom instructions; other examples of FPGA-based processors supporting custom instructions have also been reported [4],[8]. Moreover, a coprocessor for MMX-style instructions has been developed [3].

There is much work on various aspects of customisable instruction processors, particularly for embedded systems [6]. For instance, techniques for automating the choice of custom instructions have been proposed [2],[7]. However, previous research does not elaborate, for instance, on how custom instruction granularity can be used to optimise performance.

This paper addresses the above issues. It also covers methods for instrumenting a reconfigurable SoC to evaluate the effectiveness of a particular customised architecture. Applications involving advanced bit manipulations [5] will be used to illustrate our approach.

## III. Tuning instruction customisation

This section proposes four effective techniques for tuning and analysing instruction customisation targeting reconfigurable SoC devices. While similar ideas, such as instruction granularity, have been reported [2], we believe this is the first time that these techniques and their trade-offs are systematically addressed in the context of reconfigurable SoC technology.

**1. Tuning custom instruction granularity.** Instead of having a separate custom instruction for each IP block, a custom instruction can be made more coarse-grained by combining multiple IP blocks; this makes the hardware more efficient in both speed and area, at the expense of generality – a more specific custom instruction is less widely-used than a more general one.

As a simple example, two possible options for a custom instruction for computing $(x + y)^n$ are: (a) to compute this expression directly, or (b) to compute $z^n$ after an addition instruction for calculating $z = (x + y)$. Option (a) is more efficient in time since it minimises instruction fetching and decoding, and also data transfer between the custom instruction datapath and the register file. For option (b), to compute $(x + y)^n$ would need two instructions, and it would take time to transfer the result of the *add* instruction back to the register file, before commencing the $z^n$ instruction which copies this result value to the hardware for computing $z^n$. Option (a) eliminates this overhead of data transfer for intermediate values.

Additionally, option (a) is likely to be more efficient in space than option (b), since option (a) often has more scope for optimising the control and the datapath. An example will be used to illustrate this point below.

**2. Tuning custom instruction hardware.** There are two common opportunities for tuning custom instruction hardware: core-driven and interface-driven.

First, core-driven optimisations usually involve coarse-grained custom instructions, each covering multiple IP blocks. The hardware for such custom instructions can often be further optimised. A simple example concerns custom instructions that convert data to a more efficient representation for carrying out certain calculations, and then convert the result back to the original data representation. A custom instruction combining several such calculations can avoid data conversion between successive calculations.

Second, interface-driven optimisations involve techniques for tuning the custom instruction hardware to match interface constraints, such as the access time for the register file of the base processor. This can be achieved, for instance, by varying the number of pipeline stages of the IP blocks in the custom instruction hardware, provided that such IP blocks can be optimised in this way.

**3. Tuning based on run-time information.** This optimisation is specific to reconfigurable SoC. Instead of using a coprocessor with custom instructions some of which may not be used, one only needs to include the hardware for custom instructions predicted to be present during execution by run-time information.

Indeed, if reconfiguration time permits, one could even arrange to download the hardware for the new custom instructions at run-time. The effectiveness of run-time reconfiguration depends on whether the additional efficiency that run-time reconfiguration brings would offset the reconfiguration overheads [8].

**4. Instrumentation for tuning analysis.** The efficiency of the above methods can be analysed by information obtained from instrumenting the customised processor. Such information can also be used to validate analytical performance models for the base processor as well as the customised datapath. It is often difficult to model, for instance, the communication overhead between the register file and the customised datapath, since bus arbitration can be unpredictable, and the overhead may vary due to different input/output characteristics.

A simple technique is to include a custom instruction for reading a clock cycle counter; this instruction can be used to obtain the number of cycles that one or more base instructions or custom instructions would take. The custom instructions for tuning analysis are only necessary during the optimisation process; the associated hardware is not necessary at run time, and can be optimised away.

The performance information can be used not only for evaluating current instruction customisations, but also for assessing the effect of technology evolution. For instance, one can study how the performance of specific instruction customisation varies with the clock speed of the base processor. An example will illustrate this opportunity in Section V.

## IV. Tuning bit manipulation for MicroBlaze

To illustrate the proposed techniques, we adopt the Xilinx MicroBlaze processor [9] targeting the Xilinx University Program (XUP) development system [10]. Custom instruction hardware can be implemented in a coprocessor which communicates with the MicroBlaze processor through FSL (Fast Simplex Link) connections. This architecture allows us to support new instructions seamlessly.

In order to obtain cycle accurate profiling results for tuning analysis, a dedicated counter is implemented in the MicroBlaze which can be read using a custom instruction. The counter simply records the number of clock cycles elapsed since the circuit is reset. Because FSL is employed to attach to the clock cycle counter, there is latency when accessing the counter. This latency can be obtained by accessing the clock cycle counter in consecutive instructions; the difference between the readings is the latency required to access the counter. Cycle count measurements can be calibrated to take into account the latency of the FSL.

Our benchmarks include applications involving transfer coding such as uudecode and uuencode, and integer compression based on variable byte encoding (compress). It has been shown that these benchmarks can be significantly accelerated for the Alpha processor by novel parallel extract (pex) and parallel deposit (pdep) instructions based on the butterfly and the inverse butterfly networks [5]. However, the MicroBlaze does not contain these instructions so they are promising candidates for custom instructions.

All the benchmarks are described in C and are compiled to MicroBlaze directly. They are then profiled and instrumented. The total number of clock cycle counts to complete the benchmarks is determined with appropriate calibration. The call count of each routine is also determined.

Hardware for six custom instructions has been developed, and each is embedded into the MicroBlaze using FSL. The custom instructions include byte-packed addition and subtraction (padd, psub), parallel extract (pex), parallel deposit (pdep) and count the number of leading zero (ctlz). There is an additional custom instruction paddpex, which combines the hardware for padd and pex, so that we can study the effect of custom instruction granularity.

For each custom instruction, a dedicated FSL bus is instantiated and attached to the MicroBlaze. The custom instruction blocks consist of four states, covering the cases when the block is: (a) idle, (b) reading data from the register file, (c) writing results to the register file, and (d) busy with the computation. The state machine is parameterised to allow different amounts of input, output and custom circuit latency; for instance, increasing the number of pipeline stages in the custom circuit would also increase the circuit latency.

Synplicity 9.0 is used for FPGA synthesis and Xilinx ISE9.2 is used for place and route. Also, the retiming facility in the synthesis tool is used extensively to control the number of pipeline stages of a custom circuit.

## V. Results

Table 1 reports the pure software implementation results. It shows each custom instruction. All the benchmarks invoke advanced bit manipulation instructions (pex/pdep). The padd and psub instruction is used by uuencode and uudecode respectively while compress requires the ctlz instruction. These results form the baseline for comparison with implementations involving custom instructions.

Table 1: Profiling: number of calls of custom instructions in applications.

| Instruction | uuencode | uudecode | compress |
|---|---|---|---|
| pex | 560 | 0 | 2500 |
| pdep | 0 | 68000 | 0 |
| padd | 560 | 0 | 0 |
| psub | 0 | 68000 | 0 |
| ctlz | 0 | 0 | 5000 |

Table 2: Hardware implementation results.

| Block | Slices | Latency | Clock (MHz) |
|---|---|---|---|
| pex / pdep | 539 | 3 | 181 |
| padd, psub | 96 | 1 | 380 |
| ctlz | 53 | 1 | 288 |
| paddpex | 612 | 4 | 181 |

Table 3: Clock cycle count for custom instructions.

| Instruction | Software | Core | Core + FSL | Speed up |
|---|---|---|---|---|
| pex | [628, 857] | 3 | 31 | [20.2, 27.6] |
| pdep | [624, 853] | 3 | 31 | [20.1, 27.5] |
| padd, psub | [79, 79] | 1 | 29 | [2.7, 2.7] |
| ctlz | [37, 129] | 1 | 24 | [1.5, 5.4] |
| paddpex | [707, 936] | 4 | 46 | [15.4, 20.3] |

Table 2 shows the reconfigurable hardware implementation results. For each custom instruction, we have a corresponding reconfigurable circuit. Since the MicroBlaze runs at 100MHz on the XUP board, all the custom instructions are designed to run at 100MHz or above. If a custom circuit does not achieve 100MHz, additional pipeline stages are introduced until it reaches 100MHz. In our design, the pex and pdep instructions can share one hardware block (pex/pdep) which requires three clock cycles to complete.

Table 3 shows the number of clock cycles for each instruction in software and in hardware; [$p$,$q$] indicates the minimum ($p$) and maximum ($q$) values. In addition to the latency of the custom circuit, there is communication overhead when moving data to and from the coprocessor using FSL. Instructions requiring more input or output data take more cycles. The communication overhead is significant, but good results are still obtained. Depending on the operation, the speed up ranges from 1.5 times to 27.6 times.

These results confirm the speed and area benefits of paddpex, the coarse-grained custom instruction. For speed, paddpex is about 30% faster than padd and pex

Table 4: Hardware implementation results.

| MicroBlaze | Slices | Clock (MHz) |
|---|---|---|
| Processor only (A) | 3997 | 100 |
| padd, psub, ctlz (C) | 4469 | 100 |
| padd, psub, ctlz, pex/pdep (D) | 4928 | 100 |
| padd, psub, ctlz, pex/pdep, paddpex (E) | 5412 | 100 |

Table 5: Cycle count and speed up of customisations.

| MicroBlaze design | uuencode | uudecode | compress | |
|---|---|---|---|---|
| no customisation (A) | 502035 | 60866891 | 3672613 | |
| pex/pdep (B) | 80495 | 9354891 | 1690413 | |
| padd, psub, ctlz (C) | 477885 | 56351891 | 3451713 | |
| padd, psub, ctlz, pex/pdep (D) | 62365 | 4700891 | 1436713 | Mean |
| speed up (A/B) | 5.97 | 6.51 | 2.17 | 4.88 |
| speed up (A/C) | 1.05 | 1.08 | 1.06 | 1.06 |
| speed up (A/D) | 8.05 | 12.95 | 2.56 | 7.85 |
| speed up (C/D) | 7.66 | 11.99 | 2.40 | 7.35 |

running in succession; for area, paddpex is about 4% smaller than the sum of the area of padd and pex.

We implement five different MicroBlaze designs. Design A is the original MicroBlaze without customisation. Design B covers bit manipulation instructions (pex, pdep). Design C supports padd, psub and ctlz instructions. Design D covers five custom instructions except paddpex, while Design E supports all six custom instructions including paddpex. Table 4 shows resource usage statistics; it indicates that D requires 23% more slices than A. Table 5 shows the clock cycle count of each benchmark. We compare each design with Design A. It shows that B can offer 4.88 times speed up (row A/B). C offers 1.06 times speed up (row A/C). These results show that the new bit manipulation instructions, pex and pdep [5], provide an additional 7.35 times speed up over multimedia instructions like padd, psub and ctlz which are already included in some processors. Note that our speed up results are better than those in [5]. For instance, D is 13 times faster than A for the uudecode benchmark, and is 23% larger than A.

The uuencode benchmark takes 53965 cycles on E, which is 9.3 times faster than A (502035 cycles), and 16% faster than D (62365 cycles). E has 5412 slices, which is 35% larger than A and 9.8% larger than D.

Our approach can be used not only for evaluating current instruction customisations, but also for assessing the effect of technology evolution. For instance, one can study how the performance of instruction customisation varies with the clock speed of the MircoBlaze processor. Although the MircoBlaze processor can only run at 100MHz on Virtex II devices, it can be clocked at 200MHz on more recent FPGAs such as Virtex 4 devices. It is useful to be able to estimate performance when custom instruction hardware is optimised so that its clock speed matches the MicroBlaze's.

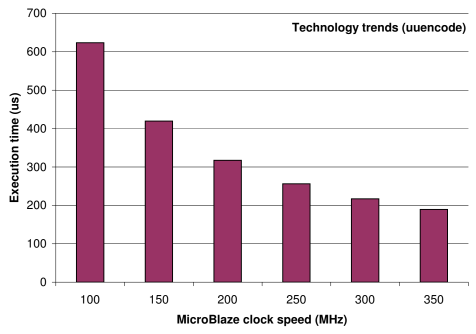

Figure 1: Variation of execution time of uuencode with clock speed of MicroBlaze.

Figure 1 covers this experiment, which assumes that the MicroBlaze can be scaled up to 350MHz. The number of pipeline stages of each custom instruction hardware has to be adjusted accordingly. For the uuencode benchmark, we find that the execution time is 623$\mu s$ for MicroBlaze running at 100MHz with all the custom instructions except paddpex. If the MicroBlaze can be clocked at 350MHz and the custom instruction hardware is further pipelined by adding 7 more stages to match the clock speed, the execution time will become 189$\mu s$, indicating that an increase in clock speed by 3.5 times would result in speed up of 3.3 times for the uuencode benchmark.

## VI. Summary

This paper covers four techniques for tuning instruction customisation: tuning custom instruction granularity, tuning custom instruction hardware, tuning based on run-time information, and instrumentation for tuning analysis. These techniques have been used in obtaining promising results for advanced bit manipulation applications. Current and future work includes automating the proposed techniques, and extending our approach to cover a variety of applications and SoC devices.

## References

[1] Altera, *Nios II Processor Reference Handbook*, 2005.
[2] K. Atasu et al, "CHIPS: Custom Hardware Instruction Processor Synthesis," *IEEE Trans. on CAD*, 27(3):528–541, 2008.
[3] M.H. Calvio et al, "Developing an MMX extension for the MicroBlaze soft processor," *Proc. ReConFig*, pp. 91–96, IEEE, 2008.
[4] R. Dimond, O. Mencer and W. Luk, "Combining instruction coding and scheduling to optimize energy in System-on-FPGA," *Proc. FCCM*, pp. 175–184, IEEE, 2006.
[5] Y. Hilewitz and R.B. Lee, "Fast bit gather, bit scatter and bit permutation instructions for commodity microprocessors," *J. Signal Process. Syst.*, 53(1–2):145–169, 2008.
[6] P. Ienne and R. Leupers (eds), *Customizable Embedded Processors: Design Technologies and Applications*, Morgan Kaufmann, 2007.
[7] S.K. Lam and T. Srikanthan, "Rapid design of area-efficient custom instructions for reconfigurable embedded processing," *Journal of Systems Architecture*, 55(1):1–14, 2009.
[8] S.P. Seng, W. Luk, and P.Y.K. Cheung, "Run-time adaptive flexible instruction processors," *Proc. FPL*, LNCS 2438, pp. 545–555, 2002.
[9] Xilinx, *MicroBlaze Processor Reference Guide*, 2007.
[10] Xilinx, *Xilinx University Program Virtex-II Pro Development System*, 2008.