Full Instruction Encoding for Heterogeneous Multi Pipeline Application Specific Instruction-Set Processors

S. Radhakrishnan* and R.G. Ragel

Department of Computer Engineering, Faculty of Engineering, University of Peradeniya

Introduction

Embedded systems are becoming ubiquitous, cheaper, powerful, and increasingly ever present in people's life. Since embedded systems usually execute a single application or a class of applications, customization can be applied to optimize for performance, cost, power etc. One popular design platform for embedded systems is the Application Specific Instruction-set Processor (ASIP), which allows such customizability without overly hindering design flexibility. Numerous tools and design systems such as *ASIP-meister* (PEAS Team, 2002) and *Xtensa* have been developed for rapid ASIP generation.

Usually ASIPs contain a single execution pipeline. Recently however, there has been trend towards having multiple pipelines. For the first time, in (Radhakrishnan *et al*., 2005), a design framework was proposed for ASIPs with varying number of pipelines. Given an application specified in C, the design system generates a processor with a number of heterogeneous pipelines specifically suitable to that application. Each pipeline is customized, with a differing instruction set and the instructions are executed in parallel in all pipelines.

In the framework described in (Radhakrishnan *et al*., 2005), the instruction widths for each pipeline are identical. Later, this work was extended with the aim of reducing processor and code size for such a design by systematically customizing the instruction encoding with differing instruction width for each pipeline, without affecting the performance of the processor. The instruction set for each pipeline is different. Therefore, instead of using identical instruction widths for instruction encoding, each pipeline can have its own encoding scheme with differing instruction width; thus, high density code can be achieved, saving instruction memory.

As an initial attempt to the extension, in (Radhakrishnan *et al*., 2006) the authors have implemented the customization of instruction encoding and forwarding path for a single pipeline (which was the third pipeline in a three pipeline design) and have analyzed the benefits of such an approach. The decision to customize the encoding of only the third pipeline was made based on: (1) the observation that it had a smaller subset of instructions compared to the first and the second pipelines and this subset does not require the same instruction width as the first two pipelines do; (2) a limitation on the width of the processors (instruction width could only be a multiplication of eight bits) which could be developed in the tool (*ASIP-meister*) that was used to generate each pipeline.

In this paper the authors have extended the customization of encoding for all three pipelines. This is achieved by customizing *ASIP-meister* to generate processors with any instruction width. The authors were able to prove that the customization of all pipelines even with a considerably larger subset of the instructions could yield a saving in overheads. Even though the saving of the number of bits in each pipeline may not be very significant in some of the cases, there is a reduction in area, leakage power and clock cycle of every design considered. This is the first known attempt to perform this reduction in the area of heterogeneous multi-pipeline system through customizing the encoding of all the pipelines.

Design overview

The design flow of our target multi-pipeline ASIP framework is as follows: it starts with an application written in C, which is compiled into a single-pipeline assembly code. The single pipeline code is then scheduled into a number of parallel pipelines, from which the instruction set for each individual pipeline is deduced. Next, *ASIP-meister*, a single-pipeline ASIP design tool, is used to design each single pipeline. All these pipelines are then integrated into a multi-pipeline processor containing a multi-issue structure where the register file is shared by all pipelines. Each pipeline performs its own program sequence and has a separate control unit that controls the operation of the related functional units on that pipeline. This design process is repeated with different number of pipelines until a design that meets the given design criteria is obtained.

Instruction encoding

Instruction encoding is the act of using binary bits with certain format to represent instructions. The format of the instructions can be generally divided into two parts: one, operation field or the operation code (*op-code*), for encoding operation; and the other, multiple operand fields for encoding operands. Our encoding approach uses a fixed instruction width for each individual pipeline, with the width varying from one pipe to another. The encoding starts with the existing instruction sequence produced by the scheduling algorithm. Encoding for each pipe is carried out separately. The instructions in a sequence are grouped based on the type of the instructions. The pattern of operand values is analyzed and operands are encoded with minimal possible number of bits. Based on the operand encoding, the operations of the whole instruction set used by the application program is then encoded. Details for both encoding tasks are given below.

Operand encoding

In the instruction set architecture (ISA) used in our design, the operands can generally be classified into two types: registers and immediate values. The encoding strategy is demonstrated in the following with register operands (or registers for simplicity). Conventionally, the size of a register field (or operand field), is determined by the size of the register file. However, with a given program sequence, a certain type of instructions may not use all of those registers. Therefore, the field size can be reduced. Rather than randomly assigning code-words to each of the registers, we used an encoding approach such that the decoding will be simpler, thus reducing hardware complexity, and hence improving performance, area and power. We call this technique reduced-bit efficient encoding or REE. Table 1 compares a typical full bit encoding (FBE) against REE for a given set of register usage.

Table 1. FBE Vs. REE

Registers	R2	R4	R13	R11
FBE ($b_3b_2b_1b_0$)	0010	0100	1101	1011
REE (c_1c_0)	10	00	01	11

Row 3 in Table 1 gives an example of such an encoding. Given a register array for a register operand field, we determine the encoding values by using REE algorithm. The full REE algorithm is omitted from this presentation for brevity. Applying this algorithm to each of the operand fields, we can obtain encoding for all operand fields in an instruction type with a minimal number of bits.

Operation encoding

We group the instructions according to the number of bits needed for the operand fields. When the instruction type is unique no bits are used for operand fields. The operation field encoding is summarized in Algorithm 1. For each instruction type, the instruction operation encoding progresses from inner most level to the outmost level; and the code size grows accordingly.

```
group instructions according to their total size of operand fields;
encodingDone = FALSE;
while encodingDone = FALSE do
   current_group = instructions with smallest partial encoding size;
   encode the current group with smallest number of bits;
   bit_difference = operand field size of next group - partial encoded
   size of current group
   if bit_difference ≠ 0 then
      pad_groups_with_bit_difference_zeros()as necessary to match
      the partial code sizes;
      encode current_group with minimal number of bits;
   end if
   if all instruction types have been fully encoded then
      encodingDone = TRUE;
   end if
end while
```

Algorithm 1. Operation Encoding

Results and discussion

We designed ASIPs for two applications from *Mibench* embedded systems benchmark suite. The designs were later synthesized using *Synopsys Design Compiler* based on the Taiwan Semiconductor Manufacturing Company's (TSMC) 90nm core library, and simulated with the *Modelsim* simulator. Figure 1 depicts the area, clock cycle and leakage power comparisons of the non-customized (typically encoded) and customized (fully encoded) designs. The average savings are 26.9% on area, 27.8% on leakage power and at the same time there is a reduction in clock cycle by 1.4%. In addition, the average instruction memory size saving is about 69%.

For instruction encoding, many approaches have been proposed. In (Lee *et al.*, 2002), authors presented a technique that encodes all instructions, required by an ASIP, with a given instruction size. A hierarchical instruction encoding for VLIW-based architecture application is presented in (Liu, 2005). These approaches are not tailored to our target processors proposed in (Radhakrishnan *et al.*, 2005). The techniques presented in this paper exploit the unique architectural feature of our target heterogeneous multi-pipeline ASIP.

Conclusions

We presented techniques to fully customize instruction encoding for a multiple pipe processor. This approach best trades off the simplicity of fixed size encoding approaches and high density of varied size encoding techniques. The encoding is customized, with each pipeline having its own fixed instruction width and the instruction width varying from one pipe to another, and hence achieving a better trade-off between the design simplicity of the fixed-width encoding and code reduction efficiency of the varied-width encoding. Due to the resource and time limitation only few applications were implemented for the current work. Many versatile applications will be implemented in the future work for customization of processor.

References

Lee, J.S., Choi, K. and Dutt, N. (2002) Efficient instruction encoding for automatic instruction-set design of configurable ASIPs, *ICCAD*, 649-654.

Liu, C.H. (2005) Hierarchical instruction encoding for VLIW digital signal processors, *ISCAS*, 3053-3056.

PEAS Team. (2002) *ASIP Meister Toolset.* Retrieved 4 2007, from ASIP Meister: http://vlsilab.ics.es.osaka-u.ac.jp/dac2003/.

Radhakrishnan, S., Hui, G. and Parameswaran, S. (2005) n-pipe: Application specific heterogeneous multi-pipeline processor design, *Workshop for Application Specific Processors.*

Radhakrishnan, S., Hui, G., Parameswaran, S. and Aleksandar, I. (2006) Application specific forwarding network and instruction encoding for multipipe ASIPs, *CODES_ISSS: 06, Proceedings of the fourth International Conference on CODES.*

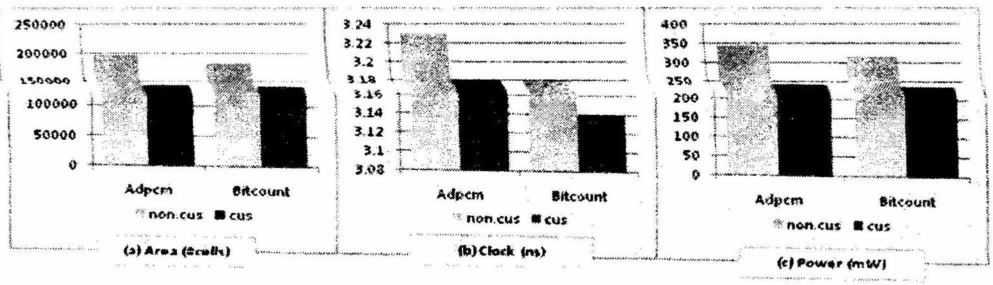

Figure 1. Savings due to full customization of encoding