Application Specific Forwarding Network and Instruction Encoding for Multi-pipe ASIPs

Swarnalatha Radhakrishnan, Hui Guo, Sri Parameswaran, Aleksandar Ignjatovic
School of Computer Science & Engineering
University of New South Wales
Sydney, Australia
{swarnar, huig, sridevan, ignjat}@cse.unsw.edu.au

ABSTRACT

Small area and code size are two critical design issues in most of embedded system designs. In this paper, we tackle these issues by customizing forwarding networks and instruction encoding schemes for multi-pipe Application Specific Instruction-Set Processors (ASIPs). Forwarding is a popular technique to reduce data hazards in the pipeline to improve performance and is applied in almost all modern processor designs; but it is very area expensive. Instruction encoding schemes have a direct impact on code size; an efficient encoding method can lead to a small instruction width, and hence reducing the code size. We propose application specific techniques to reduce forwarding networks and instruction widths for ASIPs with multiple pipelines. By these design techniques, it is possible to reduce area, code size, and even power consumption (due to reduced area), without costing any performance. Our experiments, on a set of benchmarks using the proposed customization approaches show that, on average, there are 27% savings on area, 30% on leakage power, 16.7% on code size, and at the same time, performance even improves by 4% because of the reduced clock period.

Categories and Subject Descriptors: C.1.4 [Processor Architecture]: Parallel Architectures
General Terms: Design
Keywords: Forwarding, Instruction Encoding, Multi-pipe ASIP, VLIW

1. INTRODUCTION

Embedded systems are becoming ubiquitous, cheaper, more 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, area, 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 have been developed for rapid ASIP generation [1][2]. Usually ASIPs contain a single execution pipeline. Recently however, there has been trend towards having multiple pipelines [2][16]. In [16], a design system 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 pipelines specifically suitable to the application. Each pipeline is customized, with a differing instruction set. The instructions execute in parallel in all pipelines. In the method described in [16], the instruction width for each pipeline is identical; and to maximally reduce data hazards, there is a full forwarding network which spans the entire processor, both within each pipeline and across all pipelines.

CODES+ISSS'06, October 22–25, 2006,Seoul,Korea.

This paper aims at reducing processor size and code size for such a design by systematically reducing the forwarding network, and customizing the instruction encoding with differing instruction width for each pipe, without affecting the performance of the processor.

Motivation

Data hazards cause pipeline stall, degrading performance. To reduce hazards, forwarding is used. Figure 1(a) shows a forwarding network in a processor containing two pipelines. Each pipeline has four stages: IF (instruction fetch), RR (register read), EX(execution), and RW (register write). The eight forwarding paths (shown in dashed lines) make the resultant data available for subsequent instructions, thus eliminating data hazards and improving performance. As can been seen, this performance improvement is at the cost of area. The more forwarding paths, the larger the processor. If the processor only executes a program for a given application, the instruction sequence for each pipeline is fixed, and some forwarding paths may never be used by the program. They are redundant and can therefore be deleted without affecting performance, as illustrated by Figure 1(b), where the forwarding paths are reduced to five from eight.

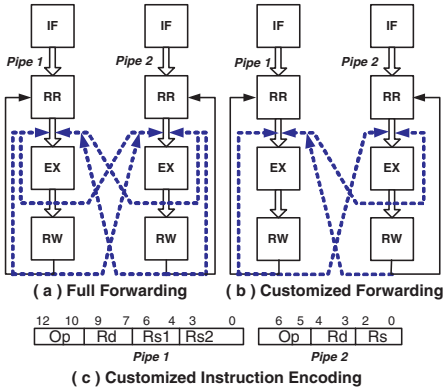

Figure 1: Forwarding and Instruction Encoding Customization

Similarly, customization can be applied to instruction encoding. The customization is based on the fact that the instruction set for each pipe is different. Instead of using an identical instruction width for instruction encoding, each pipeline can have its own encoding scheme with differing instruction width (as illustrated in Figure 1(c)), thus, high density code can be achieved, saving instruction memory.

In this paper, we attempt to maximally reduce the forwarding network and instruction code size at no performance cost.

1.1 Related Work

Research and development in the area of ASIPs has been flourishing for a couple of decades. Large amount of work has been devoted to special instruction generation to improve performance while reducing cost [4] [8] [12] [14][17] [20].

Recently, study on parallel architectures for ASIP design has begun to appear in the research literature. In [10], the authors presented a Very Large Instruction Word (VLIW) ASIP with distributed

register structure. Jacome et. al in [11] proposed a design space exploration method for VLIW ASIP datapaths. In [13], Kathail et al. proposed a design flow for a VLIW processor which allowed for Non-Programmable hardware. Sun et al. in [19] presented a design for customized multi-processors. In [16], authors proposed an ASIP design with varying number of pipelines.

Issues involved in the forwarding design for a VLIW processor is discussed in [3]. In [7], authors proposed a systematic forwarding customization approach for design exploration. Instruction scheduling for processors with a given incomplete forwarding network is presented in [18]. All of the above approaches are based on a fixed processor architecture, where the main focus is the scheduling of instructions. In contrast, our approach simultaneously takes both instruction scheduling and pipeline customization into account; hence, a better customization could be achieved yet with a simple design method.

For instruction encoding, many approaches have been proposed. In [6], 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 applications is given in [5]. The encoding uses a sophisticated approach to achieve good encoding for a given processor architecture. In [15], authors presented an instruction encoding generation technique for fast and automatic architectural space exploration. The approach focuses on encoding instructions for opcode field, assuming operand field length for each instruction is given. These approaches are not tailored to our target processors proposed in [16]. The techniques presented in this paper exploits the unique architectural features of our target ASIP, where each pipeline has different control unit for different set of instructions. 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.

1.2 Contributions

The work presented in this paper, to our best knowledge, is the first study on customizing forwarding and instruction encoding for ASIPs with multiple number of pipelines. The customization takes into account the unique features of such a processor and produces a highly efficient design. In particular, we introduce

- A new forwarding customization approach to reduce area overhead and leakage power;
- An efficient encoding technique to reduce code size, and additionally, area;

Our extensive study has shown that the proposed customizations increase the design area and power efficiency significantly without degrading performance.

1.3 Paper Organization

The rest of the paper is organized as follows: section 2 presents an overview of our target multiple pipeline ASIP design; section 3 describes the method taken to customize forwarding for such a processor; in section 4, we explain how instruction encoding for each pipeline is carried out; experimental results are given in section 5; and the paper is concluded in section 6.

2. MULTI-PIPE ASIP DESIGN OVERVIEW

The design flow for our target multi-pipe ASIPs is given in Figure 2. The design flow starts with an application written in C, which is compiled into a single-pipeline assembly code, as illustrated in Figure 3(a). The single pipeline code is then scheduled into a number of parallel pipelines, as shown in Figure 3(b), which gives the instruction set for each individual pipeline (Figure 3(c)). Next, ASIP-Meister [1], a single-pipe ASIP design software tool, is used to create a design for each pipeline. All pipelines are then integrated into a multi-pipeline processor containing a parallel structure as shown in Figure 3(d), where the register file is shared by all pipelines. Each

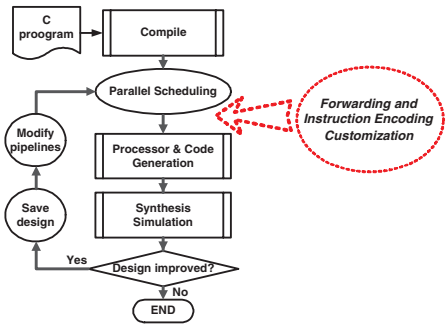

Figure 2: Multi-pipe ASIP Design Flow

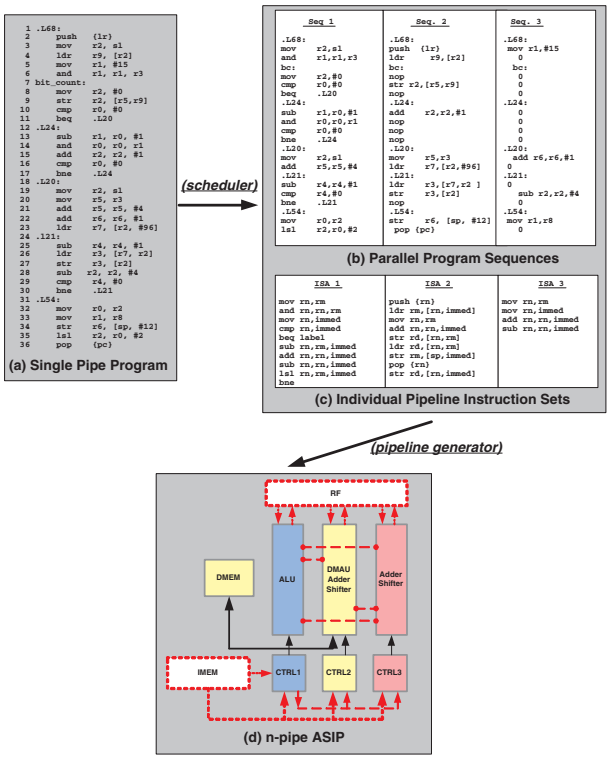

Figure 3: Design Overview

pipeline performs its own program sequence and has a separate control unit that controls the operation of the related functional units on that pipe. This design process is repeated for different number of pipelines (as shown in the design flow (Figure 2) until a design that meets a given design criteria is created.

The forwarding and instruction encoding methods introduced in this paper can be incorporated into the design systems, as indicated by the dashed block in Figure 2.

3. FORWARDING DESIGN

Forwarding improves performance at the cost of area. The cost grows as the complexity of the network becomes large. The complexity of the forwarding network is determined by the processor structure. The number of forwarding paths for a full forwarding network is

$$N_f = \sum_{i=1}^{n} (s_i \sum_{j=1}^{n} d_j), \quad (1)$$

where n is the number of pipes in the processor, d_j the number of stages from execution stage to the register write back stage in *pipe j*, and s_i the number of source operands taken by the execution stage in *pipe i*. The formula indicates that the deeper the pipeline and greater

the number of execution operands, the higher the complexity of the forwarding network. For a particular application (as in an embedded system), not all forwarding paths may be used. Some paths are redundant, and can be deleted without any effect on performance.

We determine the redundant forwarding paths based on the instruction data dependency. Assume a processor has n pipelines with all of them having d stages from EX (execution) to RW (result write back to register file). The stages are numbered, with EX as the first and RW as the last. We check the data dependency of an instruction in the parallel program sequence within a window of d parallel instructions. For an instruction i, if it has data on pipe a that depends on instruction i-k on pipe b, then there needs to be a forwarding path from the k^{th} stage in pipe b to pipe a. We use an array, denoted by $\mathbb{P}$ for such information. The element $\mathbb{P}(i,j,k,l)$ represents the number of instructions that use the forwarding path from pipe i, stage j to pipe k source l. For any forwarding path with corresponding 0 value in $\mathbb{P}$, this forwarding path will be deleted.

However, some forwarding paths may be under-utilized with low values in $\mathbb{P}$. We try to avoid usage of such under-utilized forwarding paths, using *local rescheduling* of instructions. Once no instruction uses such paths, they can be removed.

Local rescheduling of instructions consists of either shifting their position within the same pipeline, or placing them on other available pipelines. For each new position the utilization of forwarding paths is recalculated, and position that reduces the utilization of the infrequently used forwarding paths is accepted. This procedure is continued until as many zeros in $\mathbb{P}$ are generated as possible; finally, all forwarding paths that are not used at all (i.e. with a zero entry in $\mathbb{P}$) are eliminated. At each step, we examine all possible placements within a window of parallel instructions, exhaustively searching for the best placement of that instruction. We start with a window consisting of a single parallel instruction containing the instruction being rescheduled. After completing the optimization, the size of the window is increased, and the optimization is rerun. If the number of forwarding paths has dropped, the window is increased again, and the optimization is repeated until no further reduction is achieved.

Algorithm 1 Forwarding Network Reduction:

```
reductionDone = FALSE;
while reductionDone = FALSE do
   ℙ = generateForwardingPathArray();
   findInstructionCausingLowPathUtilization(ℙ);
   AttemptLocalScheduling();
   if all such instructions have been attempted then
      reductionDone = TRUE;
   end if
end while
```

The forwarding customization approach is given in Algorithm 1. An example is given in Figure 4. In this example, there are three pipelines in the processor. All three pipelies have instructions. Figure 4(a) gives stage structure for a single pipeline, where *IF* is for instruction fetch, *RR* for register read, *EX* for execution, *MA* for memory access, and *RW* for register write. Data can be forwarded from three stages: EX, MA and RW to any source inputs of EX stages. Instruction sequences for each pipeline are shown in Figure 4(b), where instructions are denoted in a general format with their operands listed in the brackets, where symbols *s1, s2, s3* represent the first, second and third operands, respectively. The arrows in Figure 4(b) indicate the data dependency; for clarity, the related dependent operands are specifically identified with the same name. For example, instruction *instr8* in pipe 2 has three source operands, with *s1* (identified as *b*) dependent on the result of *instr4* on pipe 1; *s2* (as *c*) dependent on *instr2* on pipe 2 and *s3* (as *d*) dependent on *instr6* on pipe 3. Note that some operands can be both sources and destinations, such as *b* in instruction *instr8*; they can be dependent on other operands or other operands can depend on them. Based on the dependency, a forwarding path array can be generated as shown in Figure 4(c), where a non-zero value indicates a required path. Only eight forwarding paths are required for this (exemplified) program, as compared to 72 forwarding paths in the full forwarding network. Also, from the array in Figure 4(c), we can see that instruction *instr18* is the one that incurs three extra forwarding paths. We therefore first try to re-schedule it locally (the local area is shown in the shaded region). It is found that switching *instr18* with the *nop* instruction on pipe 2 (both are underlined) reduces the forwarding paths from 8 to 7, with the related forwarding path array shown in Figure 4(d) (the changed elements are underlined). This process to identify an instruction that could reduce forwarding paths and then attempt local scheduling, is iteratively applied until no further improvement can be achieved. Details are omitted due to limited space.

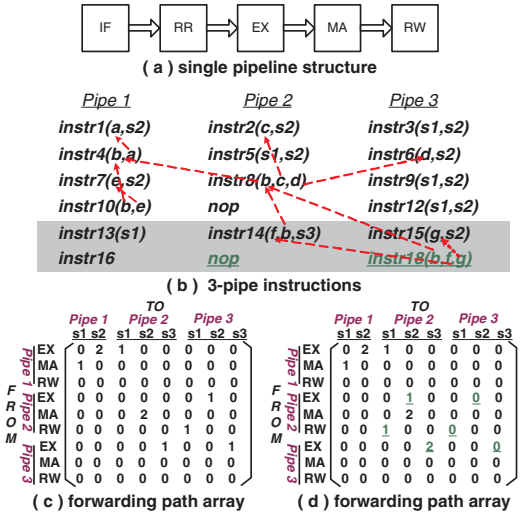

Figure 4: Forwarding Design Example

4. INSTRUCTION ENCODING

Instruction encoding uses binary bits to represent instructions. The format can be generally divided into two parts: one, operation field, for encoding operation; and another, 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 sequences produced by the scheduling algorithm. Encoding for each pipe is carried out separately. The instructions in a sequence are grouped based on the instruction type. An **instruction type** represents all instructions that have the same operation with a fixed operand structure (i.e. same number of operands and same addressing mode). Each operand can have different values. The pattern of these values for each operand is analyzed and operands are encoded with minimal 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 our ISA, operands can be generally classified into two types: registers and immediate values. We use same treatment for both types of operands. 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 since any general purpose register in the register file can be used in the instructions. However, with a given program sequence, a certain type instructions may not use all of those registers. Therefore, the field size can be reduced.

For example, assume in a program, among 16 general purpose registers, only four registers, r2, r4, r11 and r13, are used in an operand field. Instead of using four bits to encode the four registers (henceforth called *full bit binary encoding, or FBE*), we can use two bit binary code. We refer this encoding as *reduced-bit binary encoding, or RBE*. The four registers are encoded as shown in column 3 in Table 1, with the full bit encoding in column 2.

Rather than arbitrarily assigning code-words to each of the registers, we introduce an encoding approach such that the decoding will be simpler, thus reducing hardware complexity, and hence improv-

Table 1: Three Encoding Approaches

registers	FBE ($b_3b_2b_1b_0$)	RBE (c_1c_0)	REE (c_1c_0)
r2	0010	00	10
r4	0100	01	00
r13	1101	10	01
r11	1011	11	11

Table 2: Decoding Logic (a)RBE (b)REE

$$\begin{array}{lcl} b_3 = c_1 \\ b_2 = c_1 \oplus c_0 \\ b_1 = c_1 \bar{\oplus} c_0 \\ b_0 = c_1 \end{array} \quad \text{(a)} \qquad , \qquad \begin{array}{lcl} b_3 = c_0 \\ b_2 = \bar{c_1} \\ b_1 = c_1 \\ b_0 = c_0 \end{array} \quad \text{(b)}$$

ing performance, area and power. We call this technique *reduced-bit efficient encoding or REE*. Column 4 in Table 1 gives an example of such an encoding. The related decoding logic functions for RBE and REE encodings are given in Figures 2(a) and 2(b), respectively. As can be seen from the two sets of logic functions, the logic from REE is much simpler than that from RBE.

The code assignment for REE is elaborated as follows.

We list the registers to be encoded in an array with full bit binary values. For example, the array in Figure 5(a) represents four registers: r10, r12, r31 and r33 in a 64-register file. There are six bit-columns.

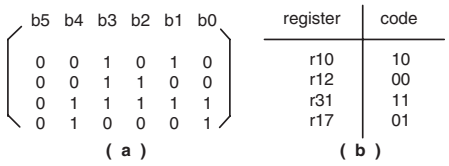

Figure 5: Register Encoding Example

Given a register array for a register operand field, we determine the encoding values by Algorithm2. Some terms used in the algorithm are given below.

A **redundant bit column** in the register array satisfies one of the following conditions: a) the column has a fixed constant value; b) the column is a repetition of another column with exactly the same values; and c) the column values are complements of the corresponding values of another column.

A one bit column is said to be **highly redundant** if there are many columns with exactly the same (or complemented) values. Columns with high redundancy allow simpler decoders.

A set of columns are **fully representative** if they form all possible code values for a given number of bits.

Take Figure 5(a) as an example. Column b5 is a constant column with fixed value 0; column b4 repeats column b0; thus, both columns are redundant. Algorithm 2 first deletes the two redundant columns and the register array is left with four columns, which is greater than two (the number of bits required for encoding). Therefore, the algorithm is then looking for two columns that are fully representative and preferably of high redundancy. Three column pairs (b_2, b_1), (b_2, b_0) and (b_1, b_0), are fully representative, because all of them have four 2-bit distinct row values (00, 01, 10, 11). Since column b_0 is repeated by column b4, it is of high redundancy as compared to other columns, we therefore choose one of the column pairs that include b_0 as the encoding code values, as shown in Figure 5(b).

Applying the field encoding 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, as specified in Algorithm 2.

Algorithm 2 Operand Field Encoding:

```
//determine the number of bits required for the field
field_size = log2(no_of_registers);
delete_redundant_columns();
if number_of_remaining_columns = field_size then
   use the columns' row values as the encoding code values;
else
   //generate code values based on the remaining columns
   find field_size full representative and of high redundancy columns;
   if found then
      use the columns' row values as code values;
   else
      find field_size high redundant columns;
      if found then
         use the columns' distinct row values as code values;
      end if
      if encoding is not complete then
         encode the rest of registers with the remaining binary code values;
      end if
   end if
end if
return field_size;
```

Instruction Encoding

The operation encoding is constructed progressively into multiple levels. Instructions with largest operand fields have minimal levels; while those with smallest operand fields have maximal levels.

The encoding starts from instructions with the smallest total number of bits for the operand fields. To explain, we refer to the example in Figure 6. As presented in the (a) part of this figure, there are ten different types of instructions. Each type of instruction may have varied operand values that are enumerated in braces.

We group these instructions according to the number of bits needed for the operand fields, as follows. Instructions 1-3 are in the first group, and they have fixed operand values (e.g., *asr1* is always applied to *r2,r5,#3*). Thus, no bits are required for the operand fields of the first group. Instructions 4-6 are in the second group, and require one bit for each operand field; thus require three bits in total. Instructions 7 and 8 in the third group and require for operand fields five bits each (note: in different format), and finally the last two instructions are in the fourth group and require six bits for operand fields. The number of bits for operand fields are indicated by the number of "*" in the related instruction formats shown on the right in Figure 6(b). All instructions are sorted based on operand field size (as already done in Figure 6(a)).

The first three instructions with zero operand size, grouped into the first group, are encoded with two bits. These two bits can denote four code values in total, and among them three code values 11, 01, 00 (arbitrarily) chosen to represent the three instructions of the group. This is the lowest level encoding and is denoted by *OP1* in Figure 6(b). We refer this encoded bit size as the **partial encoding size** (that will be used in Algorithm 3).

We now proceed to the second group of instructions consisting of instructions 4-6. Since these instructions require three bits for operand fields, we first pad the first two bits of the partial encoded size of the first group *PC1* with an extra bit of zero value (*PC2*). We need two bits to enumerate four cases: three (10,01,00) instructions of group two, plus one (11) for all three instructions of group one. This completes partial encoding *PC3*. In the same manner, we see that we need two more bits to enumerate instructions 7 and 8 of the third group (using 01 and 00), and also all instructions of all previously encoded groups (using 11). This constitutes partial encoding *PC4*. Since the number of bits needed for the operands and instructions of group two matches exactly the number of bits needed for the operands of the third group, no padding of any kind is needed in this case. Next, we need six bits for operands of the fourth group, and one bit for the two instructions of group four. This instruction bit forms partial encoding *PC5*. Finally, we need one extra bit to distinguish between instructions of the first three groups from the instructions of the fourth group. This completes the encoding of the entire instruction set. This method is formulated as Algorithm 3.

The operation field encoding is summarized in Algorithm 3. For each instruction type, the instruction operation encoding progresses from inner most level to the outmost level; and the code size grows accordingly.

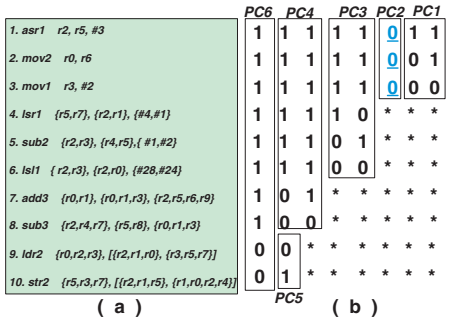

Figure 6: Instruction Encoding Example

Algorithm 3 Instruction Encoding

```
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
```

5. SIMULATIONS AND RESULTS

By applying forwarding network reduction and instruction encoding to the multi-pipe processor design system in [16], we designed ASIPs for a set of applications from Mibench [9]. All results are associated with a 3-pipe architecture.

As described in section2, we utilized ASIPmeister [1], an ASIP design tool for single pipeline processor, in generating multi-pipeline processor VHDL models for each of the applications. The designs were synthesized using Synopsys Design Compiler based on the TSMC 90nm core library, and simulated with the Modelsim simulator. The experimental setup is shown in Figure 7.

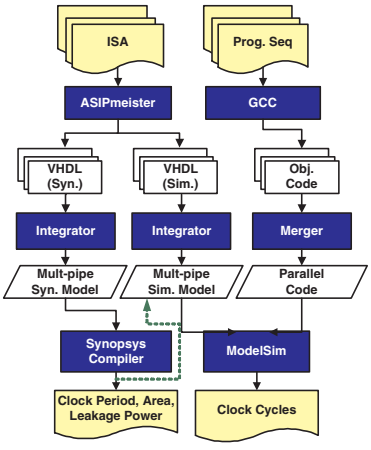

Figure 7: Experimental Setup

Performance is evaluated in the processor clock speed, which is given by Design Compiler, and the clock cycles given by Modelsim. The leakage power is estimated by the Synopsys Design Compiler.

Forwarding Network Customization

The forwarding network reduction was tested and its effect on performance, area and leakage power of entire processors is shown in Figures 8, 9 and 10, respectively. Designs with full forwarding were also conducted for comparison. As can be seen from these three figures, on average, savings of 25% on area and 27% on leakage power consumption can be obtained with 3.9% improvement on performance, as compared to the designs with the full forwarding network. This performance improvement is due to the reduced complexity of the forwarding network, which reduces the size of the processor, impacting upon the critical path, which in turn reduces the clock width.

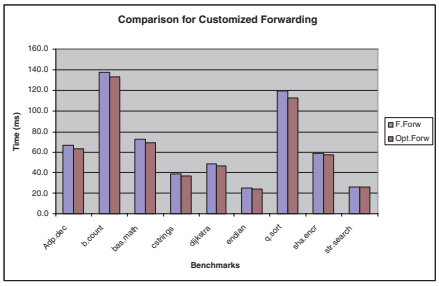

Figure 8: Full Forward. vs. Custom Forward. (Exec. Time)

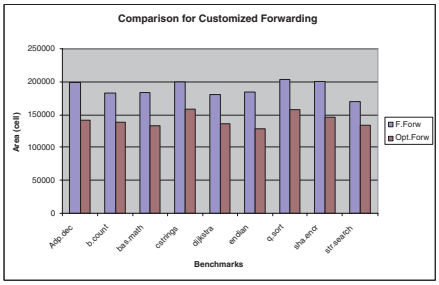

Figure 9: Full Forward. vs. Custom Forward. (Area)

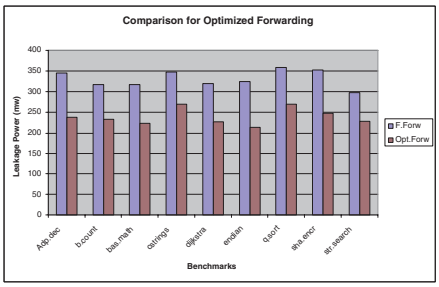

Figure 10: Full Forward. vs. Custom Forward. (Leakage Pow.)

Instruction Encoding

Figure 11 shows the instruction sizes for each of the benchmarks, produced by the proposed encoding technique. For each benchmark, the instruction widths for three individual pipes and the total parallel instruction width for the whole processor that contains the three pipes are displayed. Without code reduction, the instruction size for all three pipelines are initially 16 bits (thus making it 48 bits in total). As can be seen, by applying our encoding approach, the instruction width for each pipeline becomes smaller, which is particularly evident for pipe 3, which has instruction width below 8 bits. On average, there is about 69% saving on instruction memory size.

To show the improvement of the REE approach compared to the RBE approach, we have encoded pipe 3 for the benchmarks using RBE. The area cost for both is shown in Figure 12. As can be seen REE is better than RBE, with less area cost, as expected.

Table 3: 3-pipe Designs with forwarding and instruction encoding customization

	Clk Period (ns)		Exec.Time (ms)		Area (UnitCell)		Leak.Pow (mw)		C.Size (byte)		Perf.	Area	Power	C.Size
Benchmark	no-cus.	cus.	no-cus.	cus.	no-cus.	cus.	no-cus.	cus.	no-cus.	cus.	%	%	%	%
Adp.dec	3.23	3.07	66.1	62.8	198490	135751	345	228	1626	1355	5.0	31.6	33.9	16.7
b.count	3.18	3.07	137.5	132.7	183154	133129	316	224	1350	1125	3.5	27.3	29.0	16.7
bas.math	3.21	3.06	72.8	69.4	183787	128335	316	216	564	470	4.7	30.2	31.7	16.7
cstrings	3.63	3.42	38.9	36.6	199537	150923	347	259	690	575	5.8	24.4	25.3	16.7
dijkstra	3.17	3.03	48.4	46.3	180817	133654	320	220	1554	1295	4.4	26.1	31.3	16.7
endian	3.19	3.05	24.9	23.8	184566	125936	324	210	1002	835	4.4	31.8	35.0	16.7
q.sort	3.62	3.41	119.6	112.6	203324	154692	359	266	4452	3710	5.8	23.9	26.0	16.7
sha.encr	3.20	3.11	58.6	56.9	200599	138789	351	236	2622	2185	2.8	30.8	32.8	16.7
str.search	3.10	3.08	25.7	25.5	170020	136911	297	222	1374	1145	0.6	19.5	25.1	16.7

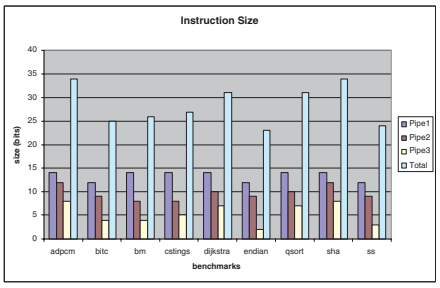

Figure 11: Code Widths with REE approach

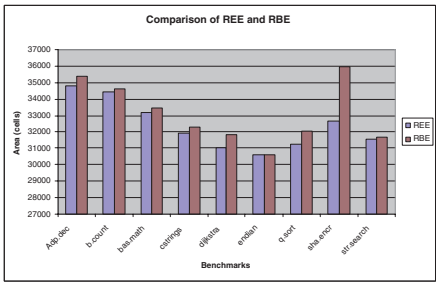

Figure 12: Comparison of REE and RBE

Forwarding Network and Instruction Encoding

Table 3 gives the simulation results for designs with both forwarding and instruction encoding in place, where the first column lists the name of benchmarks, the ten columns provides the related clock period (columns 2&3), execution time (columns 4&5), area (columns 6&7), leakage power (columns 8&9) and code size (columns 10&11) for each of the designs under two different design conditions of forwarding design and instruction encoding: without customization; and with customization. The last four columns give, in percentage, the performance improvement, area savings, power savings and code size savings of the custom design compared to non-custom designs. Note that due to encoding implementation limitation imposed by ASIPmeister, instruction code sizes are limited to 16-bit widths or 8-bit widths. As can be seen from Table 3, the average savings are 27% on area, 30% on leakage power, 16.7% on code size, and at the same time performance is improved by 4.1% on average due to the reduced clock period, which is in turn resulted from the reduction of forwarding network.

6. CONCLUSIONS

We presented techniques to customize forwarding and instruction encoding for a multiple pipe processor. The forwarding network in an ASIP for a given application is customized such that the redundant forwarding paths are deleted without affecting the performance. We proposed an approach to remove under-utilized forwarding paths so that a highly efficient application specific forwarding network can be achieved. In order to reduce code size of the parallel program sequence, we introduced an encoding technique that best trades off the simplicity of fixed size encoding approaches and high density of varied size encoding techniques. The encoding result produced by our proposed techniques has fixed instruction size for each individual pipeline, but varied from pipeline to pipeline.

Our experiments, on a set of benchmarks using the proposed customization approaches show that, on average, there are 27% savings on area, 30% on leakage power, 16.7% on code size, and at the same time, performance even improves by 4% due to the reduced clock period. It is worth noting that unlike Application Specific Integrated Circuits (ASICs), where one design is just for one application, the customized multi-pipe ASIP can also be programmed to run other applications (but may not perform optimally), which is just the very feature of ASIPs.

7. REFERENCES

[1] Asip-meister. (http://www.eda-meister.org/asip-meister/).

[2] Xtensa processor. Tensilica Inc. (http://www.tensilica.com).

[3] A. Abnous and N. Bagherzadeh. Pipelining and bypassing in a vliw processor. *IEEE Trans. on Parallel and Distributed Systems*, 5:658 – 664, 1994.

[4] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. In *CASES*, 2002.

[5] C.-H. L. et al. Hierarchical instruction encoding for vliw digital signal processors. In *Proceedings. ISCAS*, pages 3503 – 3506, 2005.

[6] J. eun Lee, K. Choi, and N. Dutt. 7. efficient instruction encoding for automatic instruction set design of configurable asips. In *Proceedings. ICCAD*, pages 649 – 654, 2002.

[7] K. Fan, N. Clark, M. Chu, K. Manjunath, R. Ravindran, M. Smelyanskiy, and S. Mahlke. 2. systematic register bypass customization for application-specific processors. In *Proceedings of IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pages 64–74. IEEE Computer Society, 2003.

[8] D. Goodwin and D. Petkov. International conference on compilers, architecture and synthesis for embedded systems. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 137 – 147. ACM Press New York, NY, USA, 2003ISBN:1-58113-676-5.

[9] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. *In IEEE 4th Annual Workshop on Workload Characterization, Austin, TX*, pages 83–94, December 2001.

[10] M. Jacome, G. de Veciana, and C. Akturan. Resource constrained dataflow retiming heuristics for vliw asips. In *Proceedings of the seventh CODES*, pages 12–16. ACM Press, 1999.

[11] M. F. Jacome, G. de Veciana, and V. Lapinskii. Exploring performance tradeoffs for clustered vliw asips. In *Proceedings of the 2000 ICCAD*, pages 504–510. IEEE Press, 2000.

[12] R. Kastner, S. Ogrenci-Memik, E. Bozorgzadeh, and M. Sar-rafzadeh. Instruction generation for hybrid reconfigurable systems. In *ICCAD*, 2001.

[13] V. Kathail, shail Aditya, R. Schreiber, B. R. Rau, D. C. Cron-quist, and M. Sivaraman. Pico: Automatically designing custom computers. In *Computer*, 2002.

[14] S. Kobayashi, K. Mita, Y. Kakeuchi, and M. Imai. Rapid prototyping of jpeg encoder using the asip development system: Peas-iii. In *Proc. of 2003 IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 485–488, 2003.

[15] A. Nohl, V. Greive, G. Braun, A. Hoffman, R. Leupers, O. Schliebusch, and H. Meyr. Instruction encoding synthesis for architecture exploration using hierarchical processor models. In *Proceedings. DAC*, pages 262 – 267, 2003.

[16] S. Radhakrishnan, H. Guo, and S. Parameswara. Customization of application specific heterogeneous multipipeline processors. In *Proceedings of DATE*. IEEE Computer Society, 2006.

[17] O. Schliebusch, A. Hoffmann, A. Nohl, G. Braun, and H. Meyr. Architecture implementation using the machine description language lisa. In *Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design*, page 239. IEEE Computer Society, 2002.

[18] A. Shrivastava, N. Dutt, A. Nicolau, and E. Earlie. Pbexplore: a framework for compiler-in-the-loop exploration of partial bypassing in embedded processors. In *Proceedings of DATE*. IEEE Computer Society, 2005.

[19] F. Sun, N. Jha, S. Ravi, and A. Raghunathan. Synthesis of application-specific heterogeneous multiprocessor architectures using extensible processors. In *Proceedings of Real Time and Embedded Technology and Applications Symposium*, pages 551 – 556. IEEE Computer Society, 2005.

[20] F. Sun, S. Ravi, A. Raghunathan, and N. Jha. Synthesis of custom processors based on extensible platforms. In *ICCAD*, 2002.