Dual-Pipeline Heterogeneous ASIP Design

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

ABSTRACT

In this paper we demonstrate the feasibility of a dual pipeline Application Specific Instruction Set Processor. We take a C program and create a target instruction set by compiling to a basic instruction set, from which some instructions are merged, while others discarded. Based on the target instruction set, parallelism of the application program is analyzed and two unique instruction sets are generated for a heterogeneous dual-pipeline processor. The dual pipe processor is created by making two unique ASIPs (VHDL descriptions) utilizing the ASIP-Meister Tool Suite, and fusing the two VHDL descriptions to construct a dual pipeline processor. Our results show that in comparison to the single pipeline Application Specific Instruction Set Processor, the performance improves by 27.6% and switching activity reduces by 6.1% for a number of benchmarks. These improvements come at the cost of increased area which for benchmarks considered is 16.7% on average.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture styles—*heterogeneous systems, pipeline processors*

General Terms

Design

Keywords

Instruction Set Generation, Dual-pipeline, ASIP, Superscalar

1. INTRODUCTION

Embedded systems are becoming more ubiquitous, cheaper and increasingly pervasive. They are in application specific equipment such as telephones, PDAs, cars, cameras etc.. Functionality within an embedded system is usually implemented using either general purpose processor(s), ASIC(s) or a combination of both. General Purpose Processors (GPP) are programmable, but consume more power than any alternate method due to execution units which are not efficiently utilized in the application. Programmability, availability of tools, and ability to rapidly deploy GPPs in embedded systems are all reasons for the common use of GPPs in embedded systems. ASICs on the other hand, are low power devices, having a small foot print, but are not upgradable and are complex to design, resulting in drawn out time to market.

A compromise solution between these two extremes, are Application Specific Instruction Set Processors (ASIPs). These are processors, with customised instructions advantageous to the particular program or class of programs. An ASIP will execute an application with great efficiency for which it was designed, though they are capable of executing any other program (usually with greatly reduced efficiency). ASIPs are programmable, quick to design and consume less power than GPPs (but more than ASICs). Programmability allows the ability to upgrade, and reduces software design time. Tools such as ASIP-meister [2], Tensilica [3], ARC [1], enable rapid creation of ASIPs.

Embedded systems differ from general purpose computing machinery since a single application or a class of applications are repeatedly executed. Thus, processing units can be customised without compromising functionality. ASIPs in particular are suited for utilisation in embedded systems where customisation allows increased performance, yet reduces power consumption by not having unnecessary functional units.

1.1 Motivation for this work

Superscalar, multiple pipeline processors are common in most modern GPPs, usually dedicating a few issues for general processing and others for floating point processing. Due to the (general) nature of GPPs, it is impossible to tailor pipelines to be more precise.

Since the application to be executed is well understood in the case of an ASIP, it is possible to accommodate customized versions of the instruction pipelines to improve performance at minimal area cost. However, research so far has only focussed on single pipeline structures. This limits instruction parallelism. ASIPs with multiple pipelines enable the execution of multiple instructions simultaneously. A processor so designed, allows a greater design space to be explored by the designer, and allows code generated for a single pipeline processor to be utilized without major modification. This method of parallelizing execution is somewhat similar to the VLIW approach, though in the case of VLIW, the compiler must necessarily be more complex.

This paper describes a two pipeline heterogeneous processor to demonstrate the feasibility of multiple pipeline ASIP processors. Each of the pipelines can be created with a subset of the total set of instructions. They can share some components such as the register file, parts of the controller etc. A processor thus created will be a heterogeneous multi-pipeline (superscalar) processor. Heterogeneity of the processor arises by the differing sets of instruction issued to the two pipes. Since the application is well understood, it is possible to do so, improving performance and reducing the area cost required.

1.2 Related Work

A number of researchers from around the globe have been working to both systematise and automate the process of ASIP design.

The overall design flow for ASIPs involves a combina-

CODES+ISSS'04, September 8–10, 2004, Stockholm, Sweden.

tion of instruction generation and design space exploration tools. In [4, 17, 22] tool suites take a specification (written in an architectural description language) and generate retargetable compilers, instruction set simulators (ISS) of the target architecture, and synthesisable HDL models of the target processor. The generated tools allow valid assembly code generation and performance estimation for each specified architecture. In [21] the design flow consists of generating instructions automatically, inserting instructions, and performing a heuristic design space exploration. Automatic instruction generation locates the regular templates derived from program dependence graphs, and implements the most suitable ones as extensible instructions, enhancing performance of the application program. Their design flow takes an application, which is profiled, and from the profile a program dependence graph is created. Blocks within the graph are ranked, and the highest ranking blocks are implemented as instructions.

In [6], the authors searched for regularity in sequential, parallel and combined sequential/parallel basic instructions in a dataflow graph. New sets of instructions were generated by combining basic instructions. Kastner et al. in [14] searched for an optimal cover of a set of regular instructions, and then constructed an optimal set of sequential instructions. Zhao et al. in [23] used static resource models to explore possible new instructions that can be added to the data path to enhance performance.

A system called PEAS-III for the creation of pipelined ASIPs is described in [16]. A parallel and scalable ASIP architecture suitable for reactive systems is described in [19]. A novel approach to select the Intellectual Properties (IP) and interfaces for an ASIP core to accelerate the application is proposed in [7]. A Hardware/Software partitioning algorithm for automatic synthesis of a pipelined ASIP with multiple identical functional units with area constraints is introduced in [5]. The code generation for time critical loops for Very Large Instruction Word (VLIW) ASIPs with heterogenous distributed register structure is addressed in [11]. In [12] a methodology for early space exploration of VLIW ASIPs with a clustered datapath is proposed. A methodology to customize the existing processor instruction set and architecture is presented in [9]. In[8] using power estimation techniques from high level synthesis, a low power ASIP is synthesized from a customized ASIC. Case study of power reduction is given in [10]. Evaluation of the effect of register file size in ASIP performance and power is done in [13].

In [15], Kathail et al. proposed a design flow for a VLIW processor consisting of a selection of Non-Programmable hardware Accelerators (NPAs), design space exploration of implementing different combinations of NPAs, and evaluation of the designs. An NPA is a co-processor for functions expressed as compute-intensive nested loops in C.

In [20] the author discusses a decoupled Access/Execute architecture, with two computation units containing own instruction streams. The processor decouples data accesses and execution (for example ALU instructions). One of them does all the memory operations, while both can perform non-memory access operations. The architecture issues two instructions per clock cycle. The two instruction streams communicate via architectural queues.

Our architecture is somewhat similar to the one proposed in [20]. We customise the processor for a particular application, while the processor in [20] is generic. They use separate register files for each pipe, with a common copy area. We use a global register file. We also avoid an instruction queue, preferring a compile time schedule of instructions and avoid data hazards by inserting NOPs appropriately.

1.3 Contributions

For the first time we create a dual pipeline ASIP and demonstrate that it is feasible to utilize such a processor to extend the design space of an application. In particular:

- a simple heterogeneous architecture has been proposed with data access instructions allocated to one pipe, instruction fetch and branch instructions allocated to the other pipe, and all other instructions implemented on one or both pipes (note we have an ALU on one pipe, and if necessary some of the operations within the ALU can be duplicated on the other);
- and, a methodology containing an algorithm with polynomial complexity has been proposed to determine which instructions should be in both pipes.

1.4 Paper Organization

The rest of the paper is organized in the following way. Section 2 describes the architecture template of the dual pipeline processor to be implemented. The following section describes the methodology taken to design dual pipeline processor. Simulations and results are given in section 4. Finally, the paper is concluded in section 5.

2. ARCHITECTURE

Our goal is to design an application-specific dual pipeline processor to improve performance, reduce energy consumption with minimal area penalty.

The architecture template adopted is shown in Figure 1.

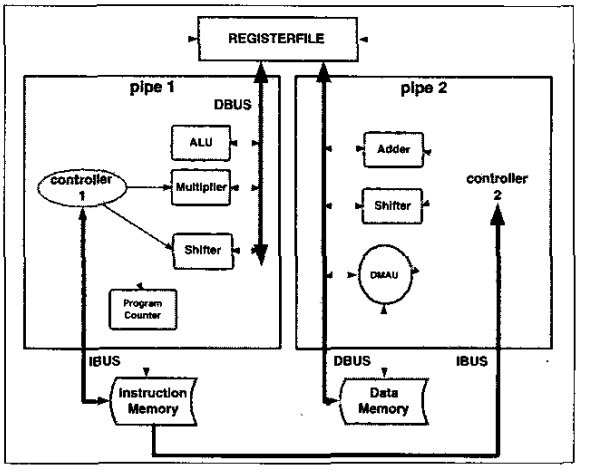

Figure 1: Architecture Template

The two-pipeline processor has two functional data paths. Both paths share the same register file, data memory and instruction memory. The register file has two-ports to enable data traffic on both pipeline paths at the same time.

The control unit on one of the pipelines, controls the instruction fetch from the instruction memory and dispatches instructions to both paths. The other path controls data memory access to transfer data between register file and data memory. Each path has a separate control unit that controls the operation of the related functional units on that path. We separate the instruction fetch and data fetch into two separate pipes to reduce the controller complexity. Some instructions will appear on both paths. The functional units in each path are determined by the instruction set designed for that path.

In the example template shown in Figure 1, some of the instructions allocated to the left pipe are ALU, multiply, and shift instructions. The add and shift instructions can also be executed on the right pipe (as shown in Figure 1). Thus it is possible to execute two add instructions simultaneously, one by the ALU on the left and one on the adder on the right. A methodology for determining the functional units that have to be duplicated is given in section 3.

Based on the architecture, we attempt:

- to efficiently exploit parallelism by dual pipelines;

- to minimize additional area cost;
- and, to minimize the total energy consumption.

3. METHODOLOGY

Our design approach for a given application consists of three tasks: target instruction set generation (Phase I); dual pipeline instruction set creation (Phase II); and, dual pipeline ASIP construction and code generation (Phase III). The design flow is shown in Figure 2.

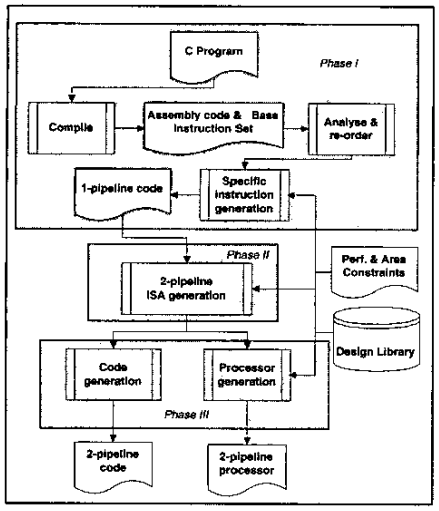

Figure 2: Design Flow

3.1 Target Instruction Set Generation

The first step of the methodology is the identification of the target instruction set. This step is marked Phase I in Figure 2.

A C program is compiled and assembly code is produced for a base RISC machine. The instruction set is first reduced by eliminating all un-used instructions.

The generated assembly code contains a number of mergeable instruction blocks. The instructions within a block are highly data dependent, which hinders parallel processing. Where possible, a specialized instruction is created for each of these blocks, and replaces them. However, specialized instructions may require more functional units in the processor, resulting in extra chip area. Therefore we create instructions only for those blocks with high execution frequency, such as blocks within loops. Methods for creation of specialized instructions are given in [18] and [21].

Figure 3 gives an example of how an assembly code is transformed by replacing blocks by specialized instructions.

Figure 3(a), shows an assembly code (AC) of a loop body produced by a compiler (the code was slightly modified to enhance explanation of methodology). Without loss of functional correctness, some instructions (in bold font) are reordered (RAC - reordered assembly code), as shown in Figure 3(b). Figure 3(b) contains six basic blocks (in rectangles). These blocks can be divided into two groups: *G1* and *G2*. Blocks in *G1 – load data to a register from memory using an indirect addressing mode with an offset. G2 stores data to memory from a register, the memory is also addressed indirectly with an offset.* For these blocks, two new instructions are generated: *Sldr* and *Sstr*. By merging instructions in those blocks, we obtain the new code as shown in Figure 3(c) (NAC - new assembly code), where highlighted new instructions replace the corresponding blocks in Figure 3(b).

Thus final instruction set is given in Figure 4, as the target instruction set (TIS) of the processor for the example given in Figure 3.

```
mov r7, r0
mov r6, r1
mov r1, r7
sub r1, r1,#8
mov r3, r7
sub r3, r3, #12
ldr r3, [r3]
lsl r2, r3, #2
sub r3, r7, #4
ldr r3, [r3]
add r3, r2, r3
ldr r2, [r1]
ldr r3, [r3]
cmp r2, r3
ble .L4
mul r0, r5
mul r5, r6
mov r1, r7
sub r1, r1, #8
mov r3, r7
sub r3, r3, #12
ldr r3, [r3]
lsl r2, r3, #2
sub r3, r7, #4
ldr r3, [r3]
add r3, r2, r3
ldr r3, [r3]
str r3, [r1]
mov r1, r3
mov r0, r3
```

(a)

```
    mov r7, r0
    mov r6, r1
G1  mov r3, r7
    sub r3, r3, #12
    ldr r3, [r3]
    lsl r2, r3, #2
G1* sub r3, r7, #4
    ldr r3, [r3]
    add r3, r2, r3
G1  mov r1, r7
    sub r1, r1, #8
    ldr r2, [r1]
    ldr r3, [r3]
    cmp r2, r3
    ble .L4
    mul r0, r5
    mul r5, r6
G1  mov r3, r7
    sub r3, r3, #12
    ldr r3, [r3]
    lsl r2, r3, #2
G1* sub r3, r7, #4
    ldr r3, [r3]
    add r3, r2, r3
    ldr r3, [r3]
G2  mov r1, r7
    sub r1, r1, #8
    str r3, [r1]
    mov r1, r3
    mov r0, r3
```

(b)

```
mov r7, r0
mov r6, r1
Sldr r3, [r7, #12]
lsl r2, r3, #2
Sldr r3, [r7, #4]
add r3, r2, r3
Sldr r2, [r7, #8]
ldr r3, [r3]
cmp r2, r3
ble .L4
mul r0, r5
mul r5, r6
Sldr r3, [r7, #12]
lsl r2, r3, #2
Sldr r3, [r7, #4]
add r3, r2, r3
ldr r3, [r3]
Sstr r3, [r7, #8]
mov r1, r3
mov r0, r3
```

(c)

Figure 3: Specific Instruction Generation (a)Original Assembly Code (b)Re-ordered Assembly Code (c)New Assembly Code

```
Sldr  Rn, [Rm, #N]
Sstr  Rn, [Rm, #N]
lsl   Rn, Rm, #N
add   Rn, Rm, Rn
ldr   Rn, [Rm]
cmp   Rn, Rm
ble   address
mov   Rn, Rm
mul   Rn, Rm
```

Figure 4: Target Instruction Set

3.2 Dual Pipeline Instruction Set Generation

Dual pipeline instruction set generation is where the target instruction set is divided into two sets (some instructions can be in both sets). This is shown as Phase II in Figure 2.

Given the *target instruction set, TIS*, we proceed to create a heterogeneous dual pipeline ASIP processor. Both pipes can have differing instructions, allowing the processor to be small, yet have fast processing speed. Take the code in Figure 3(c) for example. Instruction *cmp* can be implemented in just a single pipeline. If *cmp* is implemented in both pipes, the extra resource in one path will not be used. Thus we implement *cmp* in just one path.

A primitive processor provides functionality for memory accesses, ALU calculations and control. The memory access instructions can be paired together with ALU instructions, and can be scheduled to execute simultaneously, such that a memory instruction fetches data from memory for later use by an ALU instruction, while the ALU instruction produces results for storage (later) by another memory access instruction. Therefore, we separate the two pipe instruction sets, IS_1 and IS_2 by allocating memory access instructions to one pipeline and basic ALU instructions to other. Rest of the instructions (ALU, and specially created non-memory access instructions) are then spread over the two pipes, with some overlap of instructions in both pipes. Branches are only implemented on the instruction fetch pipe.

The `implementation efficiency` of an instruction in both pipes is proportional to the number of times that instruction can be executed in parallel, and is inversely proportional to the additional area cost of the instruction. The more frequently an instruction is executed in parallel with another instruction of the same type, the greater the `implementation efficiency`.

We define the following terms, to explain the rest of the paper.

Definition 1: *Dependency Graph, G.* The graphical rep-

resentation depicting the dependency of instructions in an instruction trace, where nodes represent instructions, and directed edges represent dependency. A node has a type that corresponds to the type of instruction it represents. An instruction is dependent on another if the first instruction can only be executed after the second is completed. Some dependent instructions can be executed simultaneously (for example see lines 4 and 5 in Figure 6(b), which can be executed together due to the pipeline execution).

Definition 2: *Connected Graph, g.* The subgraph of G, where all nodes in g are connected by directed edges.

Definition 3: *Associated Graph Set, Ψ.* The set of connected graphs that contain nodes of a given type. For an instruction, $Ins1$ in the instruction set, its *Associated Graph Set* is denoted by Ψ_i.

Definition 4: *Dependency Depth.* The depth of a node in a connected graph, g, from the starting nodes. A *starting node* is not dependent on any other nodes and has a depth of 1. The total depth of graph g is denoted by d_g.

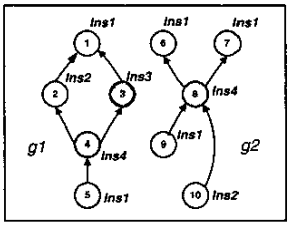

Figure 5: Example of Instruction Graph

Figure 5 shows an example. The graph represents a trace of 10 instructions with the following instruction set: $Ins1$, $Ins2$, $Ins3$ and $Ins4$. Nodes that represent same type of instructions are shaded similarly for clarity. There are two connected sub-graphs: *g1* and *g2*. For $Ins1$, its *Associated Graph Set*, $\Psi_1 = \{g1, g2\}$. For $Ins3$, its *Associated Graph set*, $\Psi_3 = \{g1\}$. For $Ins1$ in sub-graph *g1*, the depth of node 1 is 1 and the depth of node 5 is 4.

It is possible for any instructions with the same dependency depth to be grouped in pairs for parallel execution. Take the instruction nodes 6 and 7 of graph g2, in Figure 5, as an example. Both have the same depth, therefore they can be grouped into a parallel execution pair. For instructions in different connected graphs, because there is no dependency, they can always form the parallel execution pairs. For example, instruction node 6 in g2 can be paired with any instruction node in g1. But this *inter-graph matching* may result in longer execution. For example, by grouping instruction nodes 5 and 6, g1 and g2 are put in sequence. The overall execution time will be at least 8 instruction cycles. As such, it is advisable to match parallel instructions locally within connected graphs. Only left-over instructions are considered for inter-graph matching. The following definition formalizes the parallelizability.

Definition 5: *Instruction parallelizability, σ.* The potential for an instruction to be executed in parallel with another instruction of the same type. For instruction Ins_i, it is defined as

$$\sigma_i = \frac{1}{N}(LP_i + GP_i), \tag{1}$$

where N is the total number of instructions in the target instruction set; and LP_i denotes the intra-graph parallelism of *Instruction* i, and is defined as

$$LP_i = \sum_{g \in \Psi_i} \sum_{j=1}^{d_g} \lfloor \frac{k_j}{2} \rfloor, \tag{2}$$

where k_j is the number of nodes (representing Ins_i) of depth j in graph g, $|\Psi_i|$ is the number of elements in set Ψ_i; and

$$GP_i = (\frac{\sum_{g \in \Psi_i} \sum_{j=1}^{d_g} k_j \mod 2}{|\Psi_i|}) \times \lfloor \frac{|\Psi_i|}{2} \rfloor, \tag{3}$$

where the first part of the product stands for average left-over instructions per connected graph. Therefore, GP_i gives an approximate value of possible instruction matching pairs across the connected graphs.

We use the σ value to estimate how often two of the same instruction type can be scheduled simultaneously.

Definition 6: *instruction cost, c.* The area overhead, due to augmenting the basic processor by implementing the instruction.

Based on the above definitions, we define the implementation efficiency for an instruction as follows.

Definition 7: *Implementation efficiency, η.* Given an instruction, Ins_i, from the target instruction set, assume the cost for the instruction is c_i and its parallelizability is, σ_i, the implementation efficiency of implementing the instruction in both pipes is $\eta_i = \sigma_i / c_i$

In order to determine whether to implement an instruction in two pipes, we set a criteria value, denoted by Θ. An instruction is implemented in both pipes if $\eta \geq \Theta$. The value Θ could be derived in many different ways. We use an average-value based scheme by using the average value of σ and c over all instructions, which can be implemented on both pipeline paths. Assume the number of instructions (implementable in both paths) in the instruction set is m,

$$\bar{c} = \frac{1}{m}\sum_{i=1}^{m} c_i, \quad \bar{\sigma} = \frac{1}{m}\sum_{i=1}^{m} \sigma_i \tag{4}$$

Taking these two average values, we have $\Theta = \bar{\sigma}/\bar{c}$

Any instruction with $\eta \leq \Theta$ is deemed not efficient and will only be implemented in one of the pipes.

To illustrate the methodology, we continue the example in Figure 3(c), and the target instruction set shown in Figure 4. The *LOAD/STORE* instructions are in one set, and basic ALU and branch instructions are in another set. We allocate instructions *ldr*, *Sstr* and *Sldr*, to set IS_1, and ALU instructions, *add* and *lsl* and branch instruction *ble* to set IS_2. All other instructions,*mov*, *add*, *cmp*, *lsl* and *mul*, are checked for implementation efficiency. Assume the costs of each instruction are given. The implementation efficiency values for all non-memory access instructions are calculated and listed in Table 1. Note that some specialized instructions too can be implemented in both pipes (though that is not the case in this example).

Ins.	c	σ	η
add	0.02	0	0
cmp	0.01	0	0
lsl	0.04	0	0
mov	0.004	0.22	55
mul	0.12	0.11	0.92

Table 1: Area Efficiency

From the table, $m = 5$, $\bar{\sigma} = 0.066$ and $\bar{c} = 0.98$. Hence, $\Theta = 0.067$. Therefore, *mov* is the only instruction which is implementable in both pipes. Thus *mov* is implemented on both pipes, as shown in Figure 6(a). Based on the instruction set allocation to the pipes, and dependency (as shown in the (b), where the left column labels the instruction, and the right column gives the parent instruction(s)), two parallel code sequences are hand generated, as shown in figure(c) (we aim to automate this step at a later stage). Note, *NOP* instructions indicate cases where there are no parallel execution pairs. *NOP* instruction is implemented here by using a *mov* instruction – moving data between the same register, thus saving area. Note that naming dependency is also considered as illustrated in Figure 6(b) (in bold).

The two set instruction generation approach is given as an algorithm and is depicted in Figure 7.

The complexity of the algorithm given in Figure 7 is $O(n)$ where n is the number of instructions in a trace of the application.

App.	AC Single	AC Parallel	CC Single	CC Parallel	SA Single	SA Parallel	Clk. per Single (ns)	Clk.per Parallel (ns)	AC % Penal.	Perf % Impr.	SA % Red.
PNF	22159	26194	24272230	20547000	3029453359	2777023291	13.417	13.104	18.2	17.3	8.3
BS	22858	26569	11989268	8218985	1065312576	1014212135	13.888	13.415	16.2	33.8	4.8
GCD	22165	25779	8222364	7119392	1112552403	1060485171	13.703	12.974	16.3	18.0	4.7
MM	27711	32180	74272740	51202126	8715574361	7704880016	13.749	13.049	16.1	34.6	11.6
IS	22458	26269	19123756	14021924	1690419837	1652783482	13.631	11.876	17.0	36.1	2.3
SS	22858	26569	26152656	19876018	2198453345	2085369586	13.659	13.335	16.2	25.8	5.1

Table 2: Simulation Results

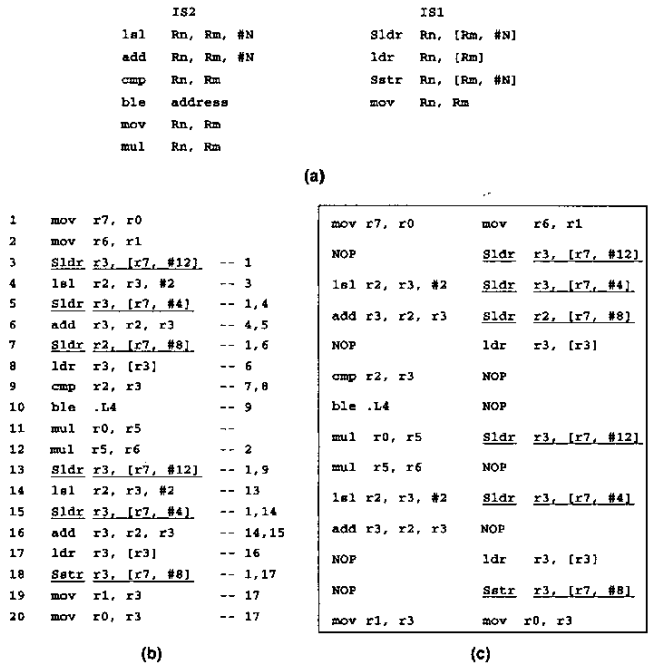

Figure 6: Two Instruction Sets and Parallel Sequences

```
/* Algorithm: Given the assembly program (NAC) and
Target Instruction Set (TIS), find two instruction sets,
IS1 and IS2 for two pipes*/
/* Initialize the two instruction sets with Load/Store
instructions and ALU/CTRL instructions in TIS*/
IS1 = LoadStore(TIS);
IS2 = ALU(TIS) + CTRL(TIS);
/* Get all non-memory instructions in TIS
and check their area efficiency */
TIS = TIS - IS1⋃CTRL(TIS);
/* Calculate η for each instruction and Theta
Calc(η, Θ);
/* Determine whether to implement instruction in one or two
pipes*/
for all Ins_i ∈ TIS
    if η_i ≤ Θ
        /* One pipe, if instruction is
         an ALU instruction, it is already assigned to
         IS2, no further assignment is required.
         Otherwise, */
        if Ins_i is not an ALU instruction
            IS2 ⇐ Ins_i;
        endif
    else
        /* Two pipe impelemtation. If it is an ALU
         instruction, further assign it to IS1;
         otherwise, assign it to both sets */
        if Ins_i is not an ALU instruction
            IS2 ⇐ Ins_i;
        endif
        IS1 ⇐ Ins_i
    endif
endfor
/* Based on the generated IS1 and IS2, create two
parallel sequences */
parallel_seq(IS1,IS2)
```

Figure 7: Dual Instruction Set Generation Algorithm

3.3 Dual Pipeline Processor and Code Generation

Finally we construct the dual pipeline processor and generate machine code for the application. This is shown as Phase III in Figure 2.

Given the two instruction sets, we generate a two-pipe processor in two steps as illustrated in Figure 8.

Step1:

Create VHDL descriptions for each instruction set by using ASIPMeister, a single-pipe ASIP design software tool. The tool takes as input the instruction set, constraints, and microcode for each instruction, and produces a synthesizable VHDL description for the processor. We create two separate processors, one for each instruction set.

Step2:

Construct the VHDL description for the dual pipe processor by modifying and integrating the two single-pipe VHDL descriptions as per designated architecture shown in Figure 1.

The instruction code for the dual pipe processor is generated by merging the two parallel assembly sequences (created during the previous stage - two instruction set generation). The merge is performed in two steps. First, each of the two parallel sequences is assembled by the GCC and object code obtained. Next, the two sets of binary code for each of the parallel sequences are merged such that a parallel instruction pair from both sequences forms a single instruction line. Each line in the code contains two instructions, one for each pipeline. Thus the resulting code forms code for the two pipe processor.

This phase is presently hand generated, though automation is possible.

4. SIMULATIONS AND RESULTS

With the proposed methodology, we designed six dual pipe processors for the following programs: Greater Common Divisor (GCD), Prime Number Finder (PNF), Matrix Multiplication (MM), Bubble Sort(BS), InSort(IS) and ShellSort(SS).

The base instruction set chosen was similar to the THUMB (ARM) instruction set. In order to verify the effective-

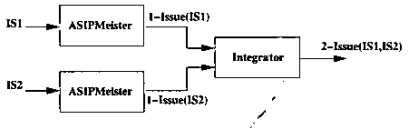

Figure 8: 2-Pipline Processor Generation

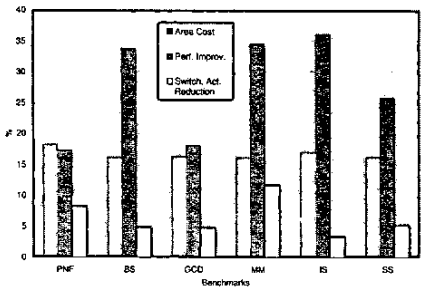

Figure 9: 2-pipes .vs. 1-pipe

ness of the created two-pipe processors, and our design approach, we also implemented those functions with single-pipe ASIPs produced by ASIPMeister. The verification system for single-pipe and two-pipe processors are shown in Figure 10. Note that the simulated results are for the instruction set after Phase I. All applications used the same sample data set for both single and parallel implementations.

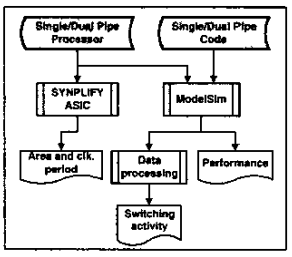

Figure 10: Synthesis/Simulation System

Given an application program, we generate a specific processor and related executable code. The execution of the program on the designed processor is simulated using ModelSim simulator, which measures the time taken to complete the program in terms of clock cycles (CC). We also obtain switching activity by modifying the output files of Modelsim. The ASIC implementation of the processor is simulated by Synplify ASIC 3.0.1, which provides the area cost in number of cells. The system was synthesized to a cell library of 0.35 microns from AMI.

The simulation results are shown in Table 2. The first column gives the name of the application, the second and third the area cost of single and parallel implementations respectively, the fourth and the fifth columns give the number of clock cycles for execution of the application, and the sixth and the seventh gives the switching activity when the application was executed. Eight and ninth columns give the clock period for the processors designed (results obtained from Synplify ASIC). Tenth, eleventh and the twelfth columns give the percentage increase in area, percentage speed improvements, and percentage switching activity reductions. These three columns (10,11 and 12) are plotted in Figure 9.

As can be seen from the table, there is up to 36% improvement in speed. The speed improvement is at the cost of some extra chip area. The two-instruction sets in each design were created with very little overlap, i.e., few instructions were in both pipes. The clock period reductions came from the decreased controller complexity. The extra area cost mainly comes from the second controller, additional functional blocks and data buses for parallel processing, which is common to all dual-pipe designs. Therefore, the extra area cost is almost same for all the design, which is around 16%.

Switching activity reductions are obtained by reduced switching in program counter and simplified functional circuitry. For example, two parallel *add* instructions can have less number of switches than the two instructions executed in a single pipe, since addition is implemented with an adder instead of an ALU in second pipe.

5. CONCLUSIONS

In this paper we have described a system which expands the design space of ASIPs, by increasing the number of pipelines. We allocate load/store operations to one of the pipes, and in general ALU operation to the other pipe. If there are additional ALU operations which can be parallelized, then they are spread over the next pipe. We see speed improvements of up to 36% and switching activity reductions of up to 11%. The additional area costs approximately 16%.

6. REFERENCES

[1] Arctangent processor. ARC International. (http://www.arc.com).

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

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

[4] N. Binh, M. Imai, and Y. Takeuchi. A performance maximization algorithm to design asips under the constraint of chip area including ram and rom size. In *ASP-DAC*, 1998.

[5] Nguyen Ngoc Binh, Masaharu Imai, Akichika Shiomi, and Nobuyuki Hikichi. A hardware/software partitioning algorithm for designing pipelined asips with least gate counts. In *Proc. of the 33rd DAC*, pages 527–532. ACM Press, 1996.

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

[7] Hoon Choi, Ju Hwan Yi, Jong-Yeol Lee, In-Cheol Park, and Chong-Min Kyung. Exploiting intellectual properties in asip designs for embedded dsp software. In *Proceedings of the 36th DAC*, pages 939–944. ACM Press, 1999.

[8] J.G. Cousin, M. Denoual, D. Saille, and O. Sentieys. Fast asip synthesis and power estimation for dsp application. *IEEE Sips 2000 : Design and Implementation*, October 2000.

[9] Kayhan K et.al. An asip design methodology for embedded systems. In *Proceedings of the seventh international workshop on Hardware/software codesign*, pages 17–21. ACM Press, 1999.

[10] Tilman Glokler and Heinrich Meyr. Power reduction for asips: A case study.

[11] 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.

[12] Margarida F. Jacome, Gustavo de Veciana, and Viktor Lapinskii. Exploring performance tradeoffs for clustered vliw asips. In *Proceedings of the 2000 ICCAD*, pages 504–510. IEEE Press, 2000.

[13] M. K. Jain, L. Wehmeyer, S. Steinke, P. Marwedel, and M. Bal-akrishnan. Evaluating register file size in asip design. In *CODES*, 2001.

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

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

[16] Akira Kitajima, Makiko Itoh, Jun Sato, Akichika Shiomi, Yoshinori Takeuchi, and Masaharu Imai. Effectiveness of the asip design system peas-iii in design of pipelined processors. In *Proc. of the 2001 ASP-DAC*, pages 649–654. ACM Press, 2001.

[17] S. Kobayashi, H. Mita, Y. Takeuchi, and M. Imai. Design space exploration for dsp applications using the asip development system peas-iii. In *ASSP*, 2002.

[18] J. Lee, K. Choi, and N. Dutt. Efficient instruction encoding for automatic instruction set desifn of configurable asips. In *ICCAD*, 2002.

[19] A. Pyttel, A. Sedlmeier, and C. Veith. Pscp: a scalable parallel asip architecture for reactive systems. In *Proceedings of DATE*, pages 370–376. IEEE Computer Society, 1998.

[20] James E. Smith. Decoupled access/execute computer architectures. *ACM Trans. Comput. Syst.*, 2(4):289–308, 1984.

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

[22] J.-H. Yang, B.-W. Kim, et al. Metacore: an application specific dsp development system. In *DAC*, 1998.

[23] Q. Zhao, B. Mesman, and T. Basten. Practical instruction set design and compiler retargetability using static resource models. In *DATE*, 2002.