Heterogeneous Multi-pipeline Application
Specific Instruction-set Processor
Design and Implementation

Swarnalatha Radhakrishnan
B.Tech. (IT-BHU), India.

This Dissertation is submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at the School of Computer Science and Engineering.

August 2006
I hereby declare that this submission is my own work and to the best of
my knowledge it contains no materials previously published or written by
another person, or substantial proportions of material which have been
accepted for the award of any other degree or diploma at UNSW or any
other educational institution, except where due acknowledgment is made
in the thesis. Any contribution made to the research by others, with
whom I have worked at UNSW or elsewhere, is explicitly acknowledged
in the thesis. I also declare that the intellectual content of this thesis
is the product of my own work, except to the extent that assistance from
others in the project’s design and conception or in style, presentation and
linguistic expression is acknowledged.

Signed ...................................... (Swaralatha Radhakrishnan)

Date .................................................... (August 2006)
COPYRIGHT STATEMENT

‘I hereby grant the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all proprietary rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstract International (this is applicable to doctoral theses only).

I have either used no substantial portions of copyright material in my thesis or I have obtained permission to use copyright material; where permission has not been granted I have applied/will apply for a partial restriction of the digital copy of my thesis or dissertation.’

Signed ……………………………………………...........................

Date 14/08/2006

AUTHENTICITY STATEMENT

‘I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis. No emendation of content has occurred and if there are any minor variations in formatting, they are the result of the conversion to digital format.’

Signed ……………………………………………...........................

Date 14/08/2006
Abstract

Embedded systems are becoming ubiquitous, primarily due to the fast evolution of digital electronic devices. The design of modern embedded systems, requires systems to exhibit, high performance and reliability, yet have short design time and low cost. Application Specific Instruction set processors (ASIPs) are widely used in embedded system since they are economical to use, flexible, and reusable (thus saves design time). During the last decade research work on ASIPs have been carried out in mainly for single pipelined processors. Improving performance in processors is possible by exploring the available parallelism in the program. Designing of multiple parallel execution paths for parallel execution of the processor naturally incurs additional cost.

The methodology presented in this dissertation has addressed the problem of improving performance in ASIPs, at minimal additional cost. The devised methodology explores the available parallelism of an application to generate a multi-pipeline heterogeneous ASIP. The processor design is application specific. no pre-defined IPs are used in the design. The generated processor contains multiple standalone pipelined data paths, which are not necessarily identical, and are connected by the necessary bypass paths and control signals. Control unit are separate for each pipeline (though with the same clock) resulting in a simple and cost effective design. By using separate instruction and data memories (Harvard architecture) and by allowing memory access by two separate pipes, the com-
plexity of the controller and buses are reduced. The impact of higher memory latencies is nullified by utilizing parallel pipes during memory access. Efficient bypass network selection and encoding techniques provide a better implementation.

The initial design approach with only two pipelines without bypass paths show speed improvements of up to 36% and switching activity reductions of up to 11%. The additional area costs around 16%. An improved design with different number of pipelines (more than two) based on applications show on average of 77% performance improvement with overheads of: 49% on area; 51% on leakage power; 17% on switching activity; and 69% on code size. The design was further trimmed, with bypass path selection and encoding techniques, which show a saving of up to 32% of area and 34% of leakage power with 6% performance improvement and 69% of code size reduction compared to the design approach without these techniques in the multi pipeline design.
Parts of the research work presented in this dissertation have been previously published in the following papers.


# Table of Contents

1 Introduction ........................................... 1

2 Literature Survey ....................................... 11
   2.1 Methodologies and Design Tools ....................... 12
      2.1.1 Application Analysis .............................. 13
      2.1.2 Architectural Exploration ......................... 14
      2.1.3 Re-targetable Compilers ............................ 19
      2.1.4 Design Environments and Methodologies .............. 23
   2.2 Techniques Used in ASIP Design ....................... 45
      2.2.1 Scheduling for Design Space Exploration .............. 45
      2.2.2 Methods for Instruction Set Generation and Selection ..... 50
      2.2.3 Bypassing Network for Processor-pipeline Performance Improvement ........................................... 67
   2.3 Testing and Evaluation Methods ....................... 71
      2.3.1 Summary ........................................ 76

3 Overview of the work .................................... 77

4 Dual-Pipeline ASIP ....................................... 83
   4.1 Dual Pipeline Architecture ............................. 84
   4.2 Design Methodology of Dual Pipeline Processor .......... 86
      4.2.1 Target Instruction Set Generation ................... 86
      4.2.2 Dual Pipeline Instruction Set Generation ............. 91
      4.2.3 Dual Pipeline Processor and Code Generation ........ 101
   4.3 Simulations and Results .................................. 103
   4.4 Conclusions .......................................... 105

5 Multi-Pipeline ASIPs ..................................... 107
   5.1 Multi-Pipeline Architecture ............................ 108
   5.2 Design Methodology of Multi-Pipeline Processor ........ 110
      5.2.1 Design Approach .................................. 110
      5.2.2 Exploitation of Instruction Parallelism .............. 115
6 Specific Customization Techniques
6.1 Forwarding Design ...................................................... 148
6.2 Instruction Encoding .................................................... 151
6.3 Simulations and Results .................................................. 157
6.4 Conclusions ............................................................. 167

7 Conclusions ............................................................... 169

A Glossary ................................................................. 171
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Technology Growth Vs Productivity</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Embedded Processors Sale. Taken from [1]</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Embedded Systems Lifecycle. Taken from [161]</td>
<td>3</td>
</tr>
<tr>
<td>1.4</td>
<td>Embedded Processors Market Share. Taken from <a href="http://www.extremetech.com">www.extremetech.com</a></td>
<td>4</td>
</tr>
<tr>
<td>1.5</td>
<td>ASIP comparison</td>
<td>5</td>
</tr>
<tr>
<td>1.6</td>
<td>Designing Sections</td>
<td>8</td>
</tr>
<tr>
<td>2.1</td>
<td>ASIP Design Flow</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>Optimization Flow of Optimization phase. Taken from [122]</td>
<td>17</td>
</tr>
<tr>
<td>2.3</td>
<td>Compiler Flow. Taken from [15]</td>
<td>18</td>
</tr>
<tr>
<td>2.4</td>
<td>Re-targetable compiler</td>
<td>20</td>
</tr>
<tr>
<td>2.5</td>
<td>Re-targetable code generation in Record</td>
<td>20</td>
</tr>
<tr>
<td>2.6</td>
<td>CHESS/CHECKERS Re-targetable Tool Suites</td>
<td>21</td>
</tr>
<tr>
<td>2.7</td>
<td>Development of code</td>
<td>22</td>
</tr>
<tr>
<td>2.8</td>
<td>Xtensa Design Environment based on <a href="http://www.tensilica.com">www.tensilica.com</a></td>
<td>24</td>
</tr>
<tr>
<td>2.9</td>
<td>PICO</td>
<td>25</td>
</tr>
<tr>
<td>2.10</td>
<td>Comparison between different Architecture Description Languages</td>
<td>28</td>
</tr>
<tr>
<td>2.11</td>
<td>Overview of PEAS-III system. Taken from [94]</td>
<td>29</td>
</tr>
<tr>
<td>2.12</td>
<td>Overview of Metacore System</td>
<td>31</td>
</tr>
<tr>
<td>2.13</td>
<td>Design by Iterative Improvement. Taken from [68]</td>
<td>33</td>
</tr>
<tr>
<td>2.14</td>
<td>Overview of EXPRESSION system. Taken from [69]</td>
<td>34</td>
</tr>
<tr>
<td>2.15</td>
<td>Overview of Flexware. Taken from [118]</td>
<td>35</td>
</tr>
<tr>
<td>2.16</td>
<td>LISA</td>
<td>37</td>
</tr>
<tr>
<td>2.17</td>
<td>SystemC Design Environment taken from [50]</td>
<td>38</td>
</tr>
<tr>
<td>2.18</td>
<td>LISA ASIP Design. Taken from [55]</td>
<td>41</td>
</tr>
<tr>
<td>2.19</td>
<td>Multiprocessor Design. Taken from [144]</td>
<td>43</td>
</tr>
<tr>
<td>2.20</td>
<td>Instruction Level Parallelism of frequently executed routines. Taken from</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>[47]</td>
<td></td>
</tr>
<tr>
<td>2.21</td>
<td>A RAW Microprocessor. Taken from [102]</td>
<td>47</td>
</tr>
<tr>
<td>2.22</td>
<td>Three approaches for cluster scheduling. Taken from [117]</td>
<td>48</td>
</tr>
<tr>
<td>2.23</td>
<td>Template Mapping. Taken from [37]</td>
<td>51</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>2.25</td>
<td>Instruction Set Derivation. Taken from [73]</td>
<td>54</td>
</tr>
<tr>
<td>2.26</td>
<td>A comparison of IMSP1 and IMSP2. Taken from [9]</td>
<td>56</td>
</tr>
<tr>
<td>2.27</td>
<td>Scheduling comparison. Taken from [165]</td>
<td>58</td>
</tr>
<tr>
<td>2.28</td>
<td>The Illustration of Template Insertion. Taken from [142]</td>
<td>59</td>
</tr>
<tr>
<td>2.29</td>
<td>Instruction Synthesis. Taken from [44]</td>
<td>61</td>
</tr>
<tr>
<td>2.30</td>
<td>TIE Development Cycle</td>
<td>63</td>
</tr>
<tr>
<td>2.31</td>
<td>MINCE in ASIP generation. Taken from [30]</td>
<td>65</td>
</tr>
<tr>
<td>2.32</td>
<td>Impact of missing forward paths. Taken from [8]</td>
<td>68</td>
</tr>
<tr>
<td>2.33</td>
<td>Steps within the Object oriented Re-targetable Architecture Simulator.</td>
<td>72</td>
</tr>
<tr>
<td>2.34</td>
<td>Two phase performance-analysis methodology. Taken from [96]</td>
<td>73</td>
</tr>
<tr>
<td>2.35</td>
<td>Processor evaluation methodology. Taken from [65]</td>
<td>74</td>
</tr>
<tr>
<td>3.1</td>
<td>General Design Flow of the methodology</td>
<td>79</td>
</tr>
<tr>
<td>4.1</td>
<td>Architecture Template</td>
<td>85</td>
</tr>
<tr>
<td>4.2</td>
<td>Design Flow</td>
<td>87</td>
</tr>
<tr>
<td>4.3</td>
<td>Specific Instruction Generation (a)Original Assembly Code (b)Re-ordered</td>
<td>88</td>
</tr>
<tr>
<td></td>
<td>Assembly Code (c)New Assembly Code</td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>Microcode Details of Specific Instruction Generation</td>
<td>89</td>
</tr>
<tr>
<td>4.5</td>
<td>Microcode Details of MOV</td>
<td>90</td>
</tr>
<tr>
<td>4.6</td>
<td>Microcode Details of SUB</td>
<td>91</td>
</tr>
<tr>
<td>4.7</td>
<td>Microcode Details of LDR</td>
<td>92</td>
</tr>
<tr>
<td>4.8</td>
<td>Target Instruction Set</td>
<td>93</td>
</tr>
<tr>
<td>4.9</td>
<td>Example of Instruction Graph</td>
<td>95</td>
</tr>
<tr>
<td>4.10</td>
<td>Two Instruction Sets and Parallel Sequences</td>
<td>98</td>
</tr>
<tr>
<td>4.11</td>
<td>Dual Instruction Set Generation Algorithm</td>
<td>99</td>
</tr>
<tr>
<td>4.12</td>
<td>2-Pipline Processor Generation</td>
<td>101</td>
</tr>
<tr>
<td>4.13</td>
<td>2-pipes .vs. 1-pipe</td>
<td>102</td>
</tr>
<tr>
<td>4.14</td>
<td>Synthesis/Simulation System</td>
<td>103</td>
</tr>
<tr>
<td>5.1</td>
<td>Architecture Template</td>
<td>109</td>
</tr>
<tr>
<td>5.2</td>
<td>Design Flow</td>
<td>111</td>
</tr>
<tr>
<td>5.3</td>
<td>Basic Block Graph</td>
<td>112</td>
</tr>
<tr>
<td>5.4</td>
<td>Parallel Program Sequences and Instruction Sets</td>
<td>113</td>
</tr>
<tr>
<td>5.5</td>
<td>Example Processor Architecture</td>
<td>116</td>
</tr>
<tr>
<td>5.6</td>
<td>Processor/Code Generation and Testing System</td>
<td>117</td>
</tr>
<tr>
<td>5.7</td>
<td>Basic Block Scheduling</td>
<td>122</td>
</tr>
<tr>
<td>5.8</td>
<td>Clock Period (ns)</td>
<td>129</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Area Efficiency .................................................. 97
4.2 Simulation Results .............................................. 100

5.1 Example of Basic Block Scheduling ............................ 123
5.2 Synthesis and Simulation Results for 1-3 Pipe Designs .......... 126
5.3 Performance Improvements and Overheads .................... 127
5.4 2-pipe Designs with Different Memory Access Time ........... 128

6.1 Three Encoding Approaches ................................. 153
6.2 Decoding Logic (a)RBE (b)REE ............................. 153
6.3 3-pipe Designs with forwarding and instruction encoding customization . 166
Embedded systems are an integral part of many modern devices we utilize daily, such as telephones, PDAs, cars, cameras etc. Typically the functionality of the device is embedded within the device in the form of a digital circuit. The demand for devices encompassing embedded systems is rapidly growing, and the capability and complexity of the devices are ever increasing. The digital design productivity of an engineer cannot keep up with the technological advances. This gap is seen in Figure 1.1, where the productivity of an engineer increases a lot slower than the advance in technology. With time to market pressures and increasing non recurring engineering costs, the use of smaller processes in embedded devices is increasingly difficult.

Programmable processors are a possible solution to overcome the productivity gap at a
reasonable cost. Programmable processors provide more flexibility and re-usability. This leads to having software implementations instead of just pure hardware solutions. Embedded software increases design productivity due to its short design time, and the ability to reuse legacy code. Since software has to meet real-time system requirements and has to be energy efficient (for mobile devices) specialized embedded processors (such as DSPs and microcontrollers) are becoming popular, as these are more cost-effective than general purpose processors. Since these processors target a broad range of applications, their energy-efficiency is inferior in comparison to a more application specific implementation.

Embedded processors are the most widely used processors (as opposed to general purpose processors). Embedded processors consist of 98% of the market for all microprocessors. Figure 1.2 shows the sales of differing processor types over several years. In the year 2000, a quarter of a billion, eight bit microprocessor chips were sold every month. Each middle-class household in developed countries have about 40-50 embedded processors [95]. Microwave oven, dish washer, washer, dryer, TV, and remote control etc, are all example of house hold devices with a microprocessor inside it. Modern cars can have hundreds of microprocessors, and are used in engine control, brake control, doors, windows etc. Such electronics allow very high level of sophisticated functionality. For
example, the processor controlling airbags can be made capable of communicating with cellphone and GPS to inform the location, in case of an accident. Thus embedded systems are becoming more ubiquitous, cheaper and increasingly pervasive and demand designs which are high in performance while low in cost.

![Embedded Systems Lifecycle](image)

Figure 1.3: Embedded Systems Lifecycle. Taken from [161]

There are many different processors available for use as embedded processors. Due to the diverse nature of the embedded systems an embedded processor does not possess a unique character. The CPUs in desktop computers are used primarily for computing, and are expected to have high computing efficiency. On the other hand embedded systems are used to serve a different purpose, and are expected to be efficient for different tasks. Unlike desktop computers (widely used indoor), embedded systems may need to be robust (scientific tools used in SITU-Small Integrated Transmitter Unit) and more reliable for time critical systems (such as flight systems). The design constraints of an embedded system will also vary depending upon requirements. Some of the well known constraints of embedded systems are real time operation response to external events, size, weight, silicon area, cooling, safety and power consumption. For example for a mobile phone or MP3 player the memory footprint is important, therefore the programmer needs to produce small code. For such a case the CISC processors with high code density are
better compared to a RISC processor. Different processors have distinguishing features and can be used as per their requirement.

The Lifecycle of an embedded system is shown in Figure 1.3. Upgrading may well be a part of the life cycle of an embedded system, in which case a complete interface, timing and functional compatibility should be ensured for any upgrade to be successful.

Most of today’s embedded processors were designed for desktop computers and later used as an embedded processor. The market share of some modern processors is shown in Figure 1.4.

Given below are some processors and their usage in devices: AMD AU1100 (MIPS), ARM7, Intel StrongARM, PowerPC, Intel XScale are some of the processors used in PDAs. Intel’s StrongARM, ARM9, TI OMAP1510 and OMAP850 are used in mobile phones. TI’s OMAP processors are designed for high-performance, power efficient, multimedia-enhanced applications such as industry-specific PDAs, Web pads, telematics, enhanced gaming, medical instrumentation, and point-of-sale terminals.

![Embedded RISC Lead Swings Constantly](image)

Figure 1.4: Embedded Processors Market Share. Taken from www.extremetech.com

An embedded system may or may not contain a microprocessor. In general, functionality within an embedded system is implemented using either general purpose processor(s),
Application Specific Integrated Circuit(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. Programability, availability of tools, and ability to rapidly deploy general purpose processors in embedded systems are all reasons for the common use of general purpose processors in embedded systems. ASICs (dedicated hardware) 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.

As shown in Figure 1.5 a compromise solution between these two extremes, are Application Specific Instruction Set Processors (ASIPs). ASIPs were able to fill the energy-flexibility gap between the general purpose processors and the ASIC. As a result of data-path optimization and special instructions ASIPs provide higher performance with lower energy and area. ASIPs with customized instructions are advantageous to a particular program or a class of programs.

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 customized without compromising functionality. ASIPs in particular are suited for utilization in embedded systems where customization allows increased performance, yet
CHAPTER 1. INTRODUCTION

reduces power consumption by not having unnecessary functional units.

Compared to ASICs, ASIPs are programmable. Programability allows easy, flexible and upgradable designs, and reduces software design time (as in a general purpose processor). Unlike general purpose processor, an ASIP can execute an application with superior performance, cost and power tradeoffs for the application to which it is targeted. Although, the ASIPs are targeted for a single or a group of applications, they are capable of executing any other program (though usually with greatly reduced efficiency).

Multiple Instructions are issued on the same clock cycle in many of the modern general purpose processors to improve performance. In contrast, current ASIPs provide only a single pipeline. The work described in this thesis explores the possibility of increasing the number of pipelines, and describes a methodology to enlarge the design space available to an ASIP architect.

Research on instruction parallelism exploration in ASIPs so far has mainly focused on simple single pipelines and Very Long Instruction Word (VLIW) ASIPs. Single pipeline implementations limit the exploration of instruction parallelism; while the VLIW processors with a given structure of parallel processing components would incur very large code size and high compiler complexity, which overshadows the performance gain offered by VLIW processors.

Tools such as ASIPMeister [119] and commercial XPRES from Tensilica [158], empower the rapid design space exploration and effective ASIP generation. ASIPMeister is capable of generating a single pipe ASIP. To exploit parallelism for high performance, multi pipeline ASIPs need to be considered. Recently Tensilica has added the FLIX instruction set to facilitate multi-pipeline execution with the limitation of 64 bit instruction width.

Tensilica’s pipelines are not very similar to the design proposed in this work. They use a base processor called Xtensa along with extended limited pipelining. To exploit parallelism for high performance, multi-pipeline ASIPs need to be considered without limitation. We believe that this is an important design area, as more and more systems are
expected to use ASIPs. It will only be a matter of time before performance issues push designers towards heterogeneous multi-pipeline application specific processors.

For a given application the application is well understood, so it is possible to design an ASIP with customized multiple pipelines. The number of pipelines and each of the individual pipes can be customized. In this thesis, we present a design approach for such a customization so that the resulting heterogeneous multi-pipeline processor can be maximally utilized to achieve high performance at low cost on area and code size. Our methodology constructs a processor given an application. We do not use any predesigned IPs in the system. PParametrized functional units are selected based on the operations present in the application. The pipes are heterogeneous as they do not contain identical set of functional units. Unlike VLIW processors, we use individual controllers for each pipe thus reducing the complexity of circuits and area. These pipelines are not connected to each other except for a few control signals. Each pipeline is like a cluster in a VLIW processor. But there are significant differences such as, the output data from each pipes are forwarded to other pipes in this design, whereas in cluster a copy operation is necessary since the registerfile is distributed in clusters. We use the same registerfile for all the pipes, thus no copy operation is required. The system does not incur data hazard penalties due to forwarding. The designing stages of the design methodology are given in Figure 1.6

It is essential to add forwarding paths to bypass the data along the pipeline stages and also across pipelines to make sure the availability of data as soon as it is computed. Forwarding paths could be from all outputs of the functional units to all inputs of the functional units or it could be selectively implemented.

Two of the issues with a multi-pipeline processor are the instruction width and the forward path redundancy. Instruction width of a processor is determined by the total number of operations performed and the length of the operands. Processors can be classified according to the forwarding path implementation and according to the instruction coding. In this thesis we have paid attention to reducing processor size and code size for such a design by systematically reducing the forwarding network, and customizing the instruction en-
coding with differing instruction width for each pipe, without affecting the performance of the processor.

This is the first known method which attempts to generate the multi-pipeline processors to increase performance and perform the reduction in the area of heterogeneous multi-pipeline system.

**Organization of Thesis**

The rest of the thesis is organized as follows: In chapter 2 a detailed literature survey regarding research on Application Specific Instruction set Processor generation and related topics is presented. Chapter 3 describes the work of this thesis.

Chapter 4 describes a two pipeline heterogeneous processor generation to demonstrate the feasibility of multiple pipeline ASIP processors. This work was performed to show the feasibility of multi-pipeline processor generation. 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 effectively, improving performance and reducing the area cost required.

In Chapter 5, we expand the work in Chapter 4 to a multiple pipeline structure, which is customizable to a given application. We enhance the pipeline structure by utilizing a forwarding scheme so that the data hazards in the pipelines can be reduced. Also, unlike in Chapter 4, where a single clock cycle penalty for memory accesses is used, we consider different wait cycles for memory access so that the effect of memory access penalty on performance can be observed. Moreover, instead of using small and non-standard benchmarks as in Chapter 4, we target the applications from Mibench benchmark suite (popular in embedded system design).

Chapter 6 presents the customization of Multi-pipeline Processor, described in the previous chapter. The customization has been done for forwarding network and instruction encoding. The customization leads to a more application based efficient design. Customization of forwarding network results in the reduction of area and power, and the customized encoding technique saves on code size and (possibly) area. These customization were done in no cost to performance.

The thesis is concluded in Chapter 7 with a summary.
Research and development on ASIP design have been intense in the last couple of decades. In this chapter, the overall design methodologies and tools are first reviewed followed by the elaboration of some specific design techniques in ASIP designs.

The major steps involved in an ASIP generation are Application analysis, architectural design space exploration, instruction set generation, code generation and hardware synthesis [82], and are discussed in detail in this chapter:

- **Application analysis:** The application along with the data set, and constraints are analyzed, to study the properties of the application. This information is used to guide the design of hardware synthesis to fulfill application requirements.

- **Architectural exploration:** A set of possible architectures are selected based on the analysis performed on the application and the given constraints. Performance of these architectures along with other constraints such as power and area are estimated to select the most suitable architecture.

- **Instruction set generation:** An efficient instruction set is generated. This is a part of the architectural exploration as well.
• Code generation: The code for the target architecture is generated with the help of the compiler which is re targetable.

• Hardware Synthesis: Finally the hardware synthesis is performed for the selected architecture with the designed ISA using a machine description in VHDL or similar language.

### 2.1 Methodologies and Design Tools

In this section, we discuss the typical approaches that have been proposed in each of the design steps shown in Figure 2.1.
2.1. METHODOLOGIES AND DESIGN TOOLS

2.1.1 Application Analysis

Analysis of an application is necessary, to study the behavior and decide the system architecture that would best suit an application. Analysis can be carried out based on different metrics such as performance, area and power. For each metric there will be several issues (such as clock period and clock cycle in case of performance) affecting them. Analysis of the impact of issues on the metric of interest is necessary to improve the design. Analysis could be either static or dynamic.

Static analysis is performed by using estimation methods based on scheduling and architecture attributes. Dynamic analysis is done by executing the application on the target architecture. Both static and dynamic analysis are performed and the information is utilized to generate an efficient system. Generally analysis is aimed at producing compact code with high performance, low power and minimum area.

The first step towards analyzing a high-level program is to use a compiler front end. Compiler development for embedded systems requires a front end software to perform analysis of the given application. Lexical, syntactical, and semantic analysis will provide information to design an efficient processor. Some researchers have developed front ends as a part of their work. SUIF compiler system [140] and Trimaran system [2] are used in parallel processor designs. LANCE [97] is another such system for compiler infrastructure used in industry and academia. LANCE consists of a front end based on ANSI C, a library of Intermediate Representation (IR) optimizations, and a backend interface capable of generating data flow trees. IR optimizations could be easily added and validated for transformation by using the executable IR in low-level C code. In addition, source-level optimization can also be performed.

Edison Design Group [62] has designed front ends for java, C++, and Fortran. These front ends are used in compilers and source analysis tools. The front end could be used with SUIF as well.

GNU C Compiler (GCC) [138] is used in analyzing applications even though it is difficult
to adapt to irregular processors such as the one investigated in this thesis.

The Application Program Analyzer (APA) used in [130] is capable of detecting data types, data access methods, and execution count of operations and functions. It is also used to find the frequency of instructions and groups of sequential instructions.

In [65] the SUIF IR is used to extract a number of parameters such as the average number of parallel instructions, average block size, number of Multiply-Accumulate operations, ratio of address computation instructions to data computation instructions, ratio of Input Output instructions to total instructions, average length of data flow graph, and the As Soon As Possible scheduling results. A similar approach using the SUIF IR is used to analyze an application in [51].

In [116] an optimizing compiler uses scheduling techniques to analyze programs. The feedback based on the analyzed programs are given to the design system to generate ASIP core along with a customized compiler.

A micro profiling strategy is presented in [85] to overcome the problem of low accuracy of source code level profiling and the low speed of machine code level profiling.

The authors in [111] examine the problems in static performance analysis of the program. Static time analyzes are techniques using the information collected at or before compile time. Different metrics important to the performance have been studied and many analysis techniques have been examined. The analyzes is classified into path analysis and system resources utilization analysis techniques. Program execution time might vary depending upon the program path or depend the resource usage.

2.1.2 Architectural Exploration

Architectural space exploration is performed using optimization of a chosen metric of interest to select a suitable architecture from a range of architectures. There are several methodologies used for architectural exploration. Methodologies proposed vary in
the range and nature of architectures used, and in the evaluation techniques employed. Generally application analyzes discussed in the previous section are used in the selection process. Profiling and evaluation for metrics are part of the architecture exploration.

Two typical architecture exploration methods for ASIP design are template-based and language-based. Configuration of Xtensa [155] and Jazz [83] are template-based. In such template-based approaches, compilers and simulators can be easily generated but the exploration is performed is limited, mainly to instruction format and operations. Thus it is difficult to modify the processor architecture itself, such as pipeline stages.

The language based architectural exploration methods for ASIPs using dedicated Architectural Description Languages (ADL) includes Expression [69], Mescal, ArchC, AVIV [70], CHESS [152], ASIPMeister [78], ISDL [67], and FlexWare [118]. In the language-based methods [42, 67, 68, 69, 72], the architecture could easily be changed by using architecture description languages, but consumes a large amount of time. Some methodologies are not capable of translating information described in design languages into Register Transfer Level (RTL) descriptions [72]. All languages are not capable of describing entire specifications of a processor design since they are limited by the available features [43].

There are some scheduling techniques used to explore the architecture and choose the more suitable and cost effective one.

A partitioning and scheduling technique with two interleaved phases is proposed in [103]. The initial binding is improved by the simulated annealing algorithm. For a given code, partitioning is done and a detailed schedule is generated based on the partitioned code and the corresponding latency is used as a cost function for optimization. This approach has been verified using a two cluster TIC6201 VLIW processor. Performance improvements between 7 to 26% is gained at the cost of compile time for the simulated annealing algorithm and this improvement might increase for systems with more clusters.

An approach to compute the lower bounds of execution delay in DFGs of VLIW ASIP
data path for a given binding is demonstrated in [80]. The binding for clustered VLIW ASIPs must consider the data transfer and serialization penalties for dependent operations. Both transferring data among register files and executing the dependent operations in the same cluster may cause delay. The scheduling problem is decomposed into sub problems and is represented in a window dependency graph model. This window model is used to measure the mentioned delays. This computation, which is based on a DFG bound to a clustered VLIW, results in a tight bound. This problem has been addressed with the help of a window dependency graph model to capture the delay caused by the serialized operations.

A technique to handle time-critical loops efficiently, during code generation is given in [79]. The process first does binding, then re-timing and finally scheduling. Two heuristics are presented in [79] to find the minimum latency, with resource and code width constraints and the minimum code width, with latency and resource constraints. Here, the code width denotes the number of clusters.

Another approach, which is defined by specified constraints such as code size to find the minimum latency for a given time critical segment of DFG over a number of clustered VLIWs is given in [81]. The given algorithm (1) finds the optimal data path, (2) binds and schedules the operations, and (3) calculates the minimum latency of the clustered VLIWs family. A simultaneous allocation and binding with constraints problem is solved in [81]. DFGs are decomposed into small DFGs and the optimal clustered structure and required components are found. The operations in DFGs are ranked according to the latency. The algorithm decides, the data path clusters, bindings to a suitable cluster, and partial scheduling for the sub-DFGs with minimum latency.

The algorithm which is presented in [81] has been used by the authors of [99] to identify the regions of interest in the design space. The methodology supports the early design space exploration of clustered VLIW datapaths for specific applications with time critical kernels. Since exploration of design space for large kernel will be time consuming, the fast design space algorithm proposed in [81] is used first to identify the range of param-
eters such as cluster capacity, number of clusters, and bus capacity. This is known as parametrization of the design space of clustered VLIW datapaths. Based on these, critical dimensions affecting performance/cost tradeoffs within the narrowed design space, a brute-force algorithm is used to search the best design. The algorithm used for binding in [99] is improved further [98] for binding/assigning operations in dataflow graph to the datapath clusters with minimum latency and data transfers. This binding algorithm consists of two phases: (1) initial binding, and (2) iterative binding for improvement. In contrast to [81], due to the efficiency of the binding algorithm the smaller clusters outperformed the larger clusters.

An algorithm for ILP extraction in clustered EPIC machines that integrates three different
approaches and a framework for scheduling and binding are presented in [122]. Predication, speculation and modulo scheduling are the three techniques used in the algorithm. This algorithm iterates through these techniques as shown in Figure 2.2. During the iteration process, this algorithm extracts additional ILP, while maintaining load balancing by using effective heuristics. This speculates loop body operations that might have minimized the initiation interval during modulo scheduling. The extracted loop operations are then assigned to the clusters and scheduled using modulo scheduling to generate the VLIW code. After the loop binding, an additional step takes care of data transfer across the clusters in order to reduce the impact on performance.

A binding technique for clustered VLIW processors is proposed in [121]. The authors have come up with an algorithm to extract all possible solutions from the conjoined OBDD (Ordered binary decision diagram). The combined two cost functions, explore trade-offs on the set of solutions for OBDD. Another set of heuristic algorithms search the OBDD for solutions that optimizes the cost functions.

![Figure 2.3: Compiler Flow. Taken from [15]](image)

In [15], a set of SUIF compiler passes are used to compile a given application into a
2.1. METHODOLOGIES AND DESIGN TOOLS

hardware. Based on the compiler analyzes a memory system and spatially distributed computing elements are constructed. A space time scheduler is used to assign instructions in such a way as to maximize the locality and minimize the physical communication distance. The interconnect operations are statically scheduled. In this approach a multi-process state machine for synthesizing is produced in Verilog as shown in the Figure 2.3. Here as the authors have mentioned we need to take care of the physical communication distance of the distributed computing elements, this problem is avoided in the proposed architecture by having the parallel processing elements within the same processor and by using the common register file.

2.1.3 Re-targetable Compilers

A re-targetable compiler is a compiler able to generate code for different configuration of processors, provided the architecture information is supplied to the compiler. Some of the re-targetable compilers along with the techniques used in them are discussed below.

Re-targetable compilers are the best way to generate the code for different target processors [105]. A machine model in description language is used for this purpose. The compiler can be used with different configurations of a parametrized ASIP in hardware/software codesigns. Re-targetable compiler technology such as MSSQ [112], CodeSyn, SPAM [11], CHESS [152], RECORD [106], FlexWare [109] and AVIV [70] are available for embedded processor designs. In embedded systems low power for mobile applications and re-targetability of architecture exploration are necessary. Therefore power optimization techniques and re-targetability have become some of the more important attributes in compiler design.

RECORD [106] is a re-targetable compiler capable of automatically deriving features of a processor necessary for code generation from a MIMOLA HDL model [131]. Thus, it automatically generate a code selector from the give processor description. The application program is written in a Data Flow Language (DFL). The DFL is translated into a
Control and Data Flow Graph (CDFG) containing atomic entities as expression trees. The code selector specific to the target processor, performs instruction selection and register allocation. All covered trees are scheduled and assigned to generate a compact code as shown in Figure 2.5.

MSSQ [112] uses RT-Level processor models in MIMOLA HDL (a structural VHDL). A rule based compiler [109] which is used to transform the source code into assembly and
2.1. METHODOLOGIES AND DESIGN TOOLS

Machine code is implemented in FlexWare system as FlexCC1, a compiler and assembler/linker. There is no IR representation and no central model maintenance. Rules are written by the user and a quick and dedicated compiler can be derived.

CHESS [152], a product of Target Compiler Technologies is a re-targetable code generator. It targets fixed-point application for specific and general purpose DSP processors. The target processor is described in nML language and is translated into an Instruction Set Graph (ISG). The ISG model captures the processor features such as connectivity, parallelism and structural hazards. This ISG model is used by the re-targetable and optimizing code generator CHESS.

Figure 2.6: CHESS/CHECKERS Re-targetable Tool Suites taken from www.retarget.com

The SPAM [11] compiler uses SUIF as its front end to translate C programs to IR. SPAM applies machine independent optimizations on the IR code, and the back-end, called TWIF, performs machine dependent optimizations.

Using the LISA ADL re-targetable compilation technique with instruction semantics is presented in [27]. This method is used to generate mapping rules of the code selector from the instruction semantics. Code selector is the core part of the back end, and is mostly automatically extracted from the ADL.

No re-targetable compilers are used in our design methodology, therefore the process
of designing is simple and any cross compiler could be used to get the assembly level program.

![Diagram of code development process](image)

**Figure 2.7: Development of code**

### Power optimization in compilers

The requirement to implement power optimization in compilers requires additional techniques. Many researchers have examined the impact of different compiler techniques on power. Instruction selection and scheduling have a great impact on the power consumed by the code. Apart from that, instruction and data encoding techniques and power efficient memories too can contribute to low power. The impact of instruction selection and scheduling on the power consumption of the code has been studied in [48, 101]. Power reduction for the proposed architecture is a vital area for future work.
2.1. METHODOLOGIES AND DESIGN TOOLS

2.1.4 Design Environments and Methodologies

There are many methodologies for ASIP design. In this section, we review approaches in two typical design paradigms: template based and language-description based. They are popular in ASIP design.

Template Based Design Approaches

[155, 83, 10, 87, 147] are some of the template based configurable and extensible processors.

Xtensa

Xtensa system-on-chip platform [155] from Tensilica enables rapid generation of processors by providing a base architecture and an automated method to extend the processor hardware and software to suit applications. The extensible instructions are implemented using TIE, an instruction set architecture description language. The TIE compiler compiles the TIE instructions and generates additional hardware logic and additions to existing software tools such as compiler, ISS, debugger, and libraries. TIE is simple but can describe a variety of instructions that can be translated into an efficient hardware implementation by the TIE compiler.

Apart from using TIE to manually customize instructions, designers can utilize Xtensa Processor Extension Synthesis (XPRES) Compiler for automatic design. Figure 2.8 [158] shows the design environment of XPRES compiler. It automatically generates RTL (TIE language) code for Tensilica’s Xtensa LX embedded processor from C language algorithms. Designers first compile ANSI C/C++ code for the base Xtensa LX processor unit. Based on the compiled program, XPRES generates a number of possible configurations, along with a graphical representation of speed and power tradeoffs versus gate count. The user can select a configuration, and hand-modify it using TIE language. Finally, the
RTL along with a software development tool chain including an instruction-set simulator, debugger, compiler and linker for the customized processor are generated by XPRES.

![Diagram](image)

**Figure 2.8: Xtensa Design Environment based on www.tensilica.com**

Xtensa uses a base processor while allowing extensible instructions, the ASIP design will contain the base processor as well. But in our methodology we do a completely application oriented designing, in which case we will have the entire processor containing only relevant instructions and Functional Units. Also, unlike the Xtensa’s parallel architecture which has a instruction width limitation, our architecture can has no limitation on the instruction width. We have designed the processor for fewer pipes therefore lower instruction length we assume we may have limitations of registerfile ports and memory ports when the number of pipes are increased but yet our parallelism can be extended with subject to area and power dissipation.

**Jazz**

The Jazz Processor from Improve [83] combines a VLIW processor and a Programmable System Architecture (PSA) with which the processor architecture can be tuned for a given application. The Jazz design system allows the modelling and simulating of a multiprocessor architecture. It can also automatically create tools for code generation and simulation.
PEAS-I

PEAS-I [10] is a hardware/software co-design environment which generates an ASIP and evaluates its performance, area, and power. The system generates the hardware core and software tools such as compiler, assembler, and simulator for the ASIP. The application is analyzed to generate ISA and processor architecture. In PEAS-I [130] system the reconfigurable architecture is configured for a given application by changing the architecture parameters. The special instructions are selected from the predefined instruction set.

PICO

![Diagram of PICO system]({{image_url}})

**Figure 2.9: PICO**

PICO is an automatic design system [87] for multiple processors on a chip. It is frame-
work based and uses hierarchical design methodology. The system design consists of one EPIC/VLIW processor, an optional NPA subsystem consisting of one or more NPAs, both connected to the two level cache system. Each NPA is customized to execute compute intensive loop nests.

The PICO system consists of a parametrized architectural template that defines the design space, a spacewalker for exploring the design space, a constructor to construct each design using components in the library, and an evaluator to evaluate the design. In a template the number of FUs and the way the FUs are connected are fixed. A template contains parameters representing the design space, and rules and constraints to apply on them.

The design flow of PICO is shown in Figure 2.9. The spacewalker searches the design space for a Pareto-optimal design. The constructor generates the machine based on the specifications selected by the spacewalker by assembling already optimized components. These components are automatically instantiated according to the parameters, which are thus easy to assemble. Evaluator is capable of estimating performance by combining static estimation and simulation, and area of the detailed design.

For large design spaces heuristics are used to examine the designs that are close to the Pareto-optimal points. Spacewalker will continue to explore the design space until the best design is found. The PICO system generates structural hardware descriptions in Verilog/VHDL for the hardware components, re-targets the compiler, assembler, and simulator to the custom VLIW processor, inserts software interfaces for the generated hardware in the application program.

The PICO system re-targets the Elcor compiler for VLIW architectures, to the target processor. The Trimaran Elcor compiler uses machine-description database (MDes) processor description language to re-target compiler, assembler and the cycle-level simulator. Elcor is restricted to re-target to HPL-PD architectures therefore, the PICO system is also bound to generate only HPL-PD architectures.
2.1. METHODOLOGIES AND DESIGN TOOLS

ARC

ARCs configurable CPU/DSP cores [146] enable SoC designers to build highly competitive products by easily customizing the processor to a target application. The cores can be optimized to achieve the good performance, small size, and low power consumption. All ARC processors utilize a 16/32 bit ISA that provides both RISC and full DSP capabilities in a unified architecture. Each may be used as-is, or because they are all configurable, quickly customized to meet a wide spectrum of requirements from an ultra small task-specific processor, to a programmable and robust application processor subsystem. And, all of ARC’s processors are synthesizable, can be easily implemented in standard foundry processes, and are supported by a wide range of development tools.

ARChitect is ARC’s revolutionary configuration tool that facilitates the licensees to unlock the power of ARC’s patented CPU/DSP processors and multimedia subsystem. Using ARChitect’s GUI based development environment and simple drag and drop menu, system-on-chip (SoC) designers select from more than 20000 pre-configured options and/or create custom instruction extensions that are optimized to the 16, 32 bit ARCompact ISA. ARChitect is capable of generating a highly differentiated, proprietary ARC-based processor or subsystem that consumes less power and area than can be created using a “fixed architecture”.

Designer has the flexibility to either add or remove a feature of the processor such as type and size of registers, address width, instruction set options, type and size of caches, interrupts, DSP subsystem, timers, and debug components. Performance and size tradeoffs are easily accomplished to get an optimized solution. Using ARChitect’s extension wizard, custom instructions, registers, and other logic could be added to reduce the number of cycles required to execute inner loops and other critical code segments. Thus, resulting in high performance and lower device frequency and power.

The ARChitect tools generates verified RTL and synthesis scripts that are compatible with industry-standard design flows. Also, it produces files and scripts which create a
set of development tools such as test bench, simulators, compiler/debugger, prototyping platform and documentation.

**Language Based Design Approaches**

<table>
<thead>
<tr>
<th>Attributes</th>
<th>nML</th>
<th>LISA</th>
<th>ISDL</th>
<th>MIMOLA</th>
<th>MDes</th>
<th>EXPRESSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler support</td>
<td>✓</td>
<td>---</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Cycle-acc sim</td>
<td>---</td>
<td>✓</td>
<td>?</td>
<td>?</td>
<td>---</td>
<td>✓</td>
</tr>
<tr>
<td>Pipeline support</td>
<td>?</td>
<td>✓</td>
<td>---</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Multi-cycle support</td>
<td>---</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Net-list info</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Instruction-set info</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>---</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Reserve tables</td>
<td>---</td>
<td>✓</td>
<td>---</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Memory Hierarchy</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>?</td>
<td>✓</td>
</tr>
</tbody>
</table>

Figure 2.10: Comparison between different Architecture Description Languages. Taken from [69]

As opposed to template based methodologies, language based methodologies are more flexible. Significant attributes of different languages are given in Figure 2.10. The languages describe either the instruction set or the architectural model. nML and Instruction Set Description Language are instruction set based languages. MIMOLA [107] is an architectural model based language and it uses hierarchical RTL netlist to describe the target machine. Many design environments with these languages have been developed. CHESS [152] is based on nML with extensions for describing architectural attributes such as pipelining. ISDL [67] environment uses ISDL to focus on orthogonal VLIW architectures.

**PEAS-III**

PEAS-III [78] is an architectural level design environment for pipelined processor generation. It is based on the micro-operation description of an instruction set. An overview of the system is given in Figure 2.11.
2.1. METHODOLOGIES AND DESIGN TOOLS

The design system needs the following inputs: Architecture parameters such as number of pipelines, delay for branch, details of pipeline stages, etc.; Functional resources to be added in the design; Instruction format details; Interrupt conditions and delays; and Micro-operations of the instructions and interrupts. Based on the specified information the tool generates a pipeline processor in two VHDL models: one for simulation and one for low level synthesis.

The flexible hardware models [114, 149] used in PEAS-III are parametrized with various characteristics, which facilitates fast resource exploration. Data paths of all instructions are constructed separately and are merged into one by glue circuits such as multiplexers and pipeline registers.

The effectiveness of PEAS-III design system has been evaluated in [91] using DLX, a RISC controller, PEAS-I core, MIPS R3000 compatible processor. The experiments have shown that PEAS-III can easily explore the design space and create a suitable design in

![Figure 2.11: Overview of PEAS-III system. Taken from [94]](image)
a short period. The work has been extended in [93] to generate re-targettable compiler from the same information used to generate synthesizable HDL files. In the process of compiler generation, the architecture attributes are analyzed by the compiler generator, to generate the mapping rule between instruction and medium level representation (IR), and scheduling information. For code generation a medium level representation of the source code is generated, optimized, mapped to the machine, and scheduled. The re-targetable compiler makes it possible to evaluate different processor structures and architectures easily and quickly so that the trade-off between performance and area can be explored and an optimal processor can be selected.

The generation of CISC ASIPs using PEAS-III is discussed in [92]. The instruction set of H8S/2600 CPU was used as the core processor to generate different ASIPs for Digital Signal Processors for DCT and FIR filter applications. Their work shows that the code size and execution cycles of CISC are better compared to RISC while the area and clock period are compatible with that of a RISC.

The effectiveness of the PEAS-III design system has been demonstrated in [94] where twelve different ASIP architectures for JPEG were generated and the evaluation results were obtained within a short time.

**Metacore**

Metacore system [42] is based on structural and behavior specification languages. The structural language is used to describe the data path structure of the target microarchitecture, while the behavioral language is used to specify the architectural parameters and the behavior of the instructions. The system is targeted for digital signal processor applications. Given an application program, the design space is explored by the recursive process shown in Figure 2.12. For each iteration, a design is evaluated for performance, area and power consumption. The output of the design system is an HDL synthesizable core and associated software tools.
MIMOLA

The MIMOLA system [167] is based on the MIMOLA machine description language, a structural low-level description language strongly oriented towards the architecture. Since the description is in low level it is suitable for synthesizable hardware models. The same machine description is used for the generation of machine code by the compiler. But the description of MIMOLA is long and complex and this affects the simulator speed. Code optimization is also limited since explicit behavioral information is not available.

Additionally, the MIMOLA design system consists of a re-targetable MSSQ compiler
which is presented in [107].

**ISDL**

Instruction Set Description Language (ISDL) presented in [67] is used to describe target architectures, especially VLIW, with its capacity to define valid operation groupings within an instruction. The language explicitly lists the ISA of the target architecture. The description consists of a global definition section, a storage section, an instruction section, and a constraint section. These sections provide the facility to specify attributes of assembly language and operations, to ensure the constraints of the architecture are met while generating the ISA. In [68] the ISDL is used to automatically generate a synthesizable Verilog processor and a cycle accurate bit-true instruction level simulator. The input of this system is the description of the target processor in ISDL machine description language. Instruction level simulation is used to compute the clock cycle, and the hardware model is used to compute the clock period, area, and power. The iterative design process is shown in Figure 2.13.

As shown in Figure 2.13, for an iterative improvement system, it is essential to have an automated simulator generation and a core generator. GENSIM, which is part of the system, generates the Instructor level simulator. This simulator consists of a state data structures, a disassembler, and the processing core routines which are specific to each architecture and are generated as C source code from the ISDL description. State data structure generation is done by sufficient memory allocation for storage elements. The assembly function is reversed to generate a disassembler. Processing core is a collection of routines. The RTL description in ISDL are translated into C language functions and compiled into a collection of routines. These routines are called, when the instruction is executed in ILS. In addition, with the simulator user interface, users can manually optimize code. The synthesis system produces ISDL description with some missing information such as timing, which is used to drive evaluation tools and an ISDL to hardware compiler. The compiled Verilog code is used to synthesize the core and the compiler is responsible to provide the
missing information in the HDL code.

**EXPRESSION**

EXPRESSION is a language for architectural design space exploration of embedded system-on-chip. It is used in the generation of re-targetable compiler and simulator. A language driven automatic design methodology based on EXPRESSION is presented in [69]. The system uses a mixed structural/behavioral representation for specifying system architectures including memory organizations.

The benefit of the EXPRESSION based methodology [69] is the ability to perform verification and consistency checking, to modify the architecture and memory organization and to generate tool kits from the specification.

Figure 2.14 shows the design methodology using EXPRESSION. The EXPRESSION description is used in two modes. In the exploration phase different base processors and
memory organizations are explored and evaluated. The toolkit generator is used to produce an exploration simulator and exploration compiler. A rapid design space exploration is done with a fast simulator and by using the compiler in estimation mode to evaluate architectures. In the refinement phase, a cycle accurate simulator and an optimizing ILP compiler are generated to fine tune the base processor and memory systems.

The toolkit generator take the EXPRESSION description to generate the information required to re-target the compiler and the simulator. The data-path netlist and the execution semantics of the architectural components are modified to re-target the structural simulator. The machine dependent parameters of instruction selection, resource allocation, ILP scheduling and memory optimization techniques are modified to re-target the ILP compiler. Re-targetting the compiler and simulator is not necessary in case of the proposed architecture. We can use any cross compiler of the target architecture and use simulators to simulate the HDL models.
2.1. METHODOLOGIES AND DESIGN TOOLS

FlexWare

FlexWare[118] is a design environment for embedded software development. It consists of four components: FlexCC for high performance C compilation, FlexSim for instruction set simulation, FlexGdb for source-level debugging and FlexPerf for performance analysis and profiling. The environment helps to explore the design space of a given set of targeting processors for an application.

FlexWare is an instruction set and architecture based embedded software development environment. The FlexWare Architecture intermediate representation (FLAIR) that supports FlexWare is an architecture database format. As shown in Figure 2.15 processor targeting information is required to drive the FlexWare tools.

![Diagram of FlexWare](image)

Figure 2.15: Overview of Flexware. Taken from [118]

LISA

The LISA language is created to overcome the gap between structural language such as VHDL and instruction set languages. The language is capable of describing the instruction set, the behavioral and timing model for different architectures such as SIMD, MIMD and VLIW. Additionally complex pipelines can be easily modeled. A re-targetable frame-
work based on machine descriptions in the LISA language is given in [72].

The LISA machine description provides information consisting of model components such as memory, resources, instruction set, behavioral, timing, and micro-architecture. Different abstract levels are covered by the LISA model and are used for the HDL code generation and software tools generation. Figure 2.16 shows model requirements of different development tools. For example the resource model is used by the HLL compiler for instruction scheduling, HDL generator for write conflict resolution, and debugger for profiling.

The implementation and the generation of synthesizable hardware description model from the LISA model is discussed in [134]. The model is spread over the different abstraction levels such as functional level model of the data path architecture to the RTL accurate models. RTL models contain good description of the structure and the information of micro-architecture model. RTL models are used in HDL generation and models from all abstraction levels are used to generate software tools. The LISA language compiler generates software tools and HDL descriptions from a single specification of the target architecture in the LISA language. The HDL description can be in VHDL, Verilog, or SystemC and the user can modify either LISA or HDL to make changes to meet data path requirements and processor constraints. The restriction of this method is the inefficient transformation of LISA behavior model in C/C++ languages into HDL languages, and therefore, the frames of functional units are generated and filled manually. If systemC is chosen as the HDL language the data path is also generated. The decoder is generated from instruction set model, timing model, and micro architecture model. Functional units in data path are generated from micro architecture model. If the HDL selected is VHDL or Verilog only the ports are derived from LISA descriptions. The behavior of the architecture, which is split into different LISA operations is combined in the generated functional units.
Figure 2.16: LISA
SystemC

SystemC design platform[50] combines the needs of system level design and hardware implementation. SystemC is an extension of C++ with classes to include modeling constructs to model concurrency, timing, and reactivity. It has features such as modules, processes, different types of signals, clocks and it enables multiple levels of abstractions, functional model creations and cycle based simulation. The design environment is given in figure 2.17. For hardware implementation models can be written either at functional level or RTL level. Software part of the system can be described in C or C++. Interface between software and hardware or between hardware blocks can be written in either cycle accurate level or at transaction accurate level models. The systemC library with simulation kernel is used for linking and the resulting output is used as a simulator for the system. Different parts of system modeled at different abstraction levels can be used for simulation.

There are some ASIP design methodologies which give priority for power optimization.
Golkler et al have developed such a methodology based on LISA tools. It is given below in detail.

An ICORE design methodology[58], is optimizes the processor core by extending the instruction set with the power and performance optimized instructions. The acquisition and tracking algorithm for a DVB-T receiver is used. A subset of a processor is used to implement the algorithm with the primitive instructions. The assembly program is simulated with the LISA toolset. Once the clock period of the architecture is verified, most frequently executed instructions and sequence of instructions are implemented in a more power efficient way. If simple, even the less-frequent tasks are implemented with hardware modules. The average power is computed by Synopsys DesignPower which is a part of Design Compiler and it is used to compute the power per task. The aim of the methodology is to minimize the average energy per task. In [57], authors have implemented optimizations for the most frequently executed saturation and Coordinate Rotation DIgital Computer (CORDIC) tasks in the ICORE system. It seems their techniques such as specialized instructions and highly optimized instructions have yielded good power saving.

In [59] a case study has been done on the ASIP design methodology. This methodology considers computational performance and area as well as power consumption simultaneously. Several ASIP power optimizations such as clock-gating, logic netlist restructuring, ISA optimization, instruction memory power reduction and use of dedicated coprocessor, have been applied and evaluated. The ICORE design has been mainly done manually, and the LISA tool set is used to generate the compiler tool set and fast simulator. The optimizations can be applied to the logic level, RTL, program, and system levels. The RTL level can be either the ISA definition, or the data path and control unit implementation. System level implementations could be programmable architectures or special hardware modules. At RTL level, optimization is done with techniques such as, ISA optimizations by selecting resources and special instructions, blocking gates to reduce toggle, encoding to reduce memory power, and using sleep mode during stand by. At software level, the
assembly code is hand optimized with different techniques. At architectural system level, partitioning into programmable devices, configurable HW blocks, and dedicated hardware is done with concern for power consumption. Although the co-processor reduces the power consumption, it is not recommended by the authors of [59] due to its reduced flexibility, and difficulty in maintainability and re-usability.

As a continuation of this work, the authors have developed a technique for automatic instruction encoding and instruction pre-decoder generation for low power ASIPs as given in [56]. Pre-decoder is the coder section connected to the memory. In order to save memory power, along with the reduction of word width and the number of words in memory, they have implemented techniques such as bit inversion to reduce the toggle activity within the memory.

The work is extended to a low power ASIP design space exploration with LISA language description in [55]. In this work the LISA methodology has been extended to power aware ASIP design. The LISA Processor Development Platform (LPDP) is used to explore the design space and generate the architecture and its software tools. Application profiling is done to identify the critical paths, the operations per time unit and also the profiled information is used to get an initial structure of an ASIP. Instruction set architecture exploration and hardware implementation is an iterative process. LPDP supports tool changes thus it is easy to verify the suitability of ISA to the application. The preference is given to performance but the HDL model is used to measure power for those designs which comply with performance requirements. Since the re-targetable high-level Language (HLL) compiler is not capable of supporting the coprocessor and complex functional units, optimization to these are done manually.

Based on the statistics, the designer models the data-path and generic instructions, code, and adapts the assembly language repeatedly until the best model is found. Extra instructions and functional units are added to meet the clock cycle count. Critical paths are modified. The semiautomatic encoding method in [56] is used and evaluated. Once the coding is done the software tools are generated along with the HDL descriptions contain-
2.1. METHODOLOGIES AND DESIGN TOOLS

Figure 2.18: LISA ASIP Design. Taken from [55]

ing pipeline registers, register files, internal interconnections, and the control logic of the architecture. But the data-path RTL description is written manually with hand optimized functional units. The debugger front-end and the interfaces for HW/SW co-simulation are generated from LISA description. Verification and system integration are done with the generated tools. Various parameters such as pipeline stages are used in design space exploration. Optimization is done at different levels. Optimizations are applied starting from logic level restructuring and ends with clock gating, and data-path optimizations.

Generally languages are confined to a control scheme but there are some exceptions. A framework with a language and methodology for constructing and simulating a wide range of programmable architectures is given in [156]. This framework is based on
the fact that designers are concerned with only the data-path exploitation and not the
instruction set and control. The processing elements (a statically scheduled horizontally-
microcoded machine) are described in a structural language with specifications of the data
path and constraints of usage. Primitive operations are extracted from the data-path de-
scription, and the ISA, to specify the operations and a controller to implement the ISA are
generated. Various architectures are realized by composing the processing elements. A
simple horizontally microcoded control scheme is also generated. A hardware description
of the architecture and bit-true cycle-accurate simulators are automatically generated.

There are also some studies for the design systems that allow many languages. Example
can be found in [133]. This system supports various hardware description languages. It
takes an architectural language such as LISA and convert it into an intermediate represent-
tation (IR), which is then optimized. The optimized IR can be implemented by different
hardware description languages to produce design results at the RTL level.

There are some ASIP design methodologies, which combine the template based and the
language based approaches. One such approach is presented in [153]. This approach
consists of the template based approach and free processor specifications. A scalable
ASIP architecture space exploration expedites the process and permits a highly optimizing
C compiler. If explorations prove the architecture range of the template based approach is
not suitable for the application, the designer can use LISA and CoSy models for further
separate optimizations. Using LISA a wide range of processors could be described, and
CoSy can target many architectural features with its extensible library of engines.

**Methodologies For Parallel Architectures**

The above methodologies mainly focused on simple processor architectures. The method-
ologies to be discussed in this section is more related to the systems with parallel archi-
tectures.
2.1. METHODOLOGIES AND DESIGN TOOLS

Multiprocessor Design

Heterogeneous multi-processor system-on-chip consists of many customized processors. The methodology given in [144] is to design such a multi-processor architecture. The architecture consists of multiple extensible processors. An iterative improvement algorithm is used to simultaneously assign, schedule tasks and select the associated complex instructions in an interleaved manner. Figure 2.19 shows the design flow of this methodology.

Initially, the processors are homogeneous and are customized through the design process. The tasks are assigned and scheduled for all processors, using the modified critical path algorithm [157]. Then, the custom instructions for the tasks are selected in order to reduce the overall execution time. The process is iterated until the execution time of the task graph cannot be improved further. In the proposed architecture the designing is done by studying the application thus reducing the iterations. Also, resulting in a less complex architecture.
Re-configure VLIW

A hardware/software co-design approach has been discussed in [47]. The methodology devised is for an application driven customization of a VLIW architecture. The VLIW example used is the VEX system. The VEX compiler is derived from Multiflow C [120] compiler. The compiler is modified to support a customized architecture. The application is translated into intermediate stage and simulated on various architectures to study the behavior. Using the VEX re-targetable compiler different architectures and micro-architectures are explored, and the designs are evaluated using VEX compiled simulator. A cost function is used to evaluate the architecture. In this design the clusters used are identical. Once the optimal architecture is chosen the source code is tuned for higher performance by unrolling loops for the highly executed routines. Additionally custom instructions are introduced for those routines. The average Instruction level parallelism of frequently executed routines is given in figure 2.20. In proposed architecture a simple method has been used to compile the application to the target architecture. Thus reducing the time to reconfigure the compiler.

Figure 2.20: Instruction Level Parallelism of frequently executed routines. Taken from [47]
2.2. TECHNIQUES USED IN ASIP DESIGN

Multi-issue processor

The feasibility of architectural level synthesis has been analyzed in [35]. A near optimal search algorithm is used to select and evaluate the performance and area of the cost effective multiple issue processor. A trace driven static model with branch prediction is used to estimate the performance. Branch prediction could be incorporated in the proposed architecture to improve the scheduling to a global scheduling.

2.2 Techniques Used in ASIP Design

As seen in the previous sections, there are many design issues in ASIP design. Three issues are very closely related to our design methodology. They are instruction scheduling, forward network and instruction encoding. In this section we review the approaches related to these specific design issues.

2.2.1 Scheduling for Design Space Exploration

Instruction scheduling is used to improve performance on machines with instruction pipelines. It explores instruction level parallelism to maximize the number of functional units operating in parallel and to minimize the time they spend waiting for each other, without changing the meaning of the code. To maximally exploit the instruction parallelism in a program, parallel structures such as multi-issue processors with parallel functional units can be used. As the functional units operate in parallel increase, the register file requires a large number of read/write ports. The increase in number of ports may increase the clock period [3] thus degrading performance.

The increase in number of ports in a register file could be avoided by partitioning the register file and making it distributive. Each small distributed register file serves a group of functional units and they form a cluster that can work in parallel with other clusters. Thus short clock periods can be achieved. However, the compilation process for the
clustered architecture requires the scheduling of data and operations within clusters and between clusters, which makes the scheduling complex as compared to traditional instruction scheduling [34]. As the authors have mentioned distributed registerfile will reduce the clock periods but scheduling is the problem. In order to solve this problem we have opted to use a global registerfile thus no scheduling between between pipes. Since the benchmarks used in our approach needed processors with only few pipelines the clock period was not affected.

Instruction scheduling basically can be divided into static and dynamic scheduling. Static scheduling is uses the compiler to schedule instructions into different computing components; while dynamic scheduling is carried out by hardware scheduling component during the program execution. Static scheduling requires more intelligent compiler and dynamic scheduling demands more sophisticated hardware. As a compromise between these two extremes one can choose static scheduling to assists the dynamic scheduling with reduced hardware cost to improve performance. Usually the hardware interlock is used to assure the correctness of the static scheduled code execution. An experimental comparison and discussion about static and dynamic code scheduling is presented in [28]. Our architecture is somewhat similar to VLIW therefore we opted to use the static. This helps to have a less complex circuit. We are not using an intelligent compiler for our target processor but we take advantage of the existing compiler for a single pipeline processor of the target ISA. Along with that our scheduler is used to schedule the code for the target architecture.

In [41], a compiler called bulldog was introduced. This is the first known work on scheduling for VLIW architecture with a partitioned register file. Bulldog uses several techniques such as trace scheduling to find more parallelism, memory-reference and memory-bank disambiguation to increase memory bandwidth, and new code-generation algorithms. Bulldog uses a sequential, two-phase process for scheduling. The two phases are: (1) assigning the operations in the trace and registers to clusters using Bottom up greedy (BUG) algorithm; and (2) scheduling the instructions using the list scheduler algorithm. In phase one, BUG traverses through the directed acyclic graph (DAG) recursively to make esti-
mation about functional unit and operand availability for each operation and maintain a record for that. When the operations are assigned to the clusters, the list scheduler, in phase two, will take care of the insertion of communication operations. BUG is an intuitive algorithm that has a complete resource model. Since it does not produce a final scheduled code, it cannot predict the actual resource usage during scheduling. Also, it does not keep a record of intercluster bus utilization. Therefore, the cluster assignments made by the BUG algorithm are based on estimated values and not optimized.

An algorithm similar to the one used in bulldog with multi-phases was used to partition code into clusters for a limited connectivity VLIW (LC-VLIW) architecture with homogeneous clustering [26, 4], where each cluster has its own register file and communication between clusters is through a number of buses that connect all the clusters. With such a connection model, when there is a need to obtain data from one cluster to another, a specific data move instruction is inserted. The code is then compacted locally to reduce the delay incurred from insertion of copy operations. A partitioning algorithm was proposed to minimize a given cost function. This technique was used in Percolation Scheduling compiler developed at UC-Irvine [125].

![Figure 2.21: A RAW Microprocessor. Taken from [102]](image)
Compilation on machines with distributed resources requires spatial as well as traditional temporal scheduling [102]. For instance, a greedy, critical path based technique called Dominant Sequent Clustering (DSC) [160] is used in [102] for RAWCC, which a compiler for the RAW \(^1\) architecture. Figure 2.21 depicts the RAW machine, it comprises a simple, replicated tiles and a programmable tightly integrated interconnect between tiles. The tile with five-stage pipeline is connected by a pipelined point-to-point network. The network in between the register file and the functional unit allows fast communication. The interface of the interconnect is exposed to the software.

The compiler for the RAW machine, RAWCC has multiple phases in cluster assignment. During the first phase called clustering, the instructions with no or negligible parallelism are grouped together by the DSC [160] algorithm. This algorithm in RAWCC is similar to the BUG algorithm used in bulldog. In the merging phase, the clusters are combined to ensure the resulting number of clusters does not exceed the number of the available execution units. Two heuristics in the merging are used to load balance and to minimize the communication cost. Here, the primary goal is to reduce the intercluster communication.

In [117], a unified-assign-and-schedule (UAS), approach is proposed to overcome the

\[\text{a) BUG b) Limited Connectivity VLIW c) UAS}\]

Figure 2.22: Three approaches for cluster scheduling. Taken from [117]

\(^1\)The RAW microprocessor has distributed all its resources over a pipelined two-dimensional interconnect, and exposing them fully to the compiler. A RAW machine is similar to a VLIW machine but has multiple flows of control to exploit parallelism. Further, it assumes a non-uniform memory access and multi-sequential instruction streams
constrained scheduling problem that existed in BUG and LC-VLIW. As illustrated in Figure 2.22, in BUG and LC-VLIW, scheduling and cluster assignments are done in different phases. Compared to BUG and LC-VLIW, UAS is a single pass approach. In UAS approach, a list scheduling algorithm is used and the cluster assignment is integrated into the list scheduler so that cluster assignment and instruction schedule can be performed simultaneously. Since finding optimal cluster assignment and scheduling is an NP complete problem, a heuristic algorithm for a best possible cluster assignment and minimal schedule length was used.

Another single pass scheduling algorithm for clustered VLIW machines is presented in [128]. This scheduling algorithm is especially efficient for designs, where the cross cluster communication is heavy, which forms the bottleneck to the performance. The approach schedules different loop iterations to different clusters and uses loop unrolling to reduce the loop dependency so that the communication between different clusters can be reduced. In [129], the authors have extended their work with a new architecture where each cluster not only has a dedicated register file but also has its own instruction cache and data cache so that communication locality can be exploited. Such benefit is fully realized depending on the ability of the compiler to generate a more effective code. Since the coherence among the distributed local caches and the main memory has to be maintained, the snoopy protocol for write invalidation [38] has been implemented. This protocol is completely transparent to the ISA and the bus arbitration and coherence are managed by hardware. Some of the problems faced in the above work such as inter cluster communication, locality exploitation, and main memory maintainance are not faced in the proposed architecture due to its simplicity.

In [53], further work on scheduling with distributed memory hierarchy was performed. The work was focused on word-interleaved cache clustered VLIW architectures. The scheduling algorithm has four steps: loop unrolling; memory instruction latency assignment; instruction ordering; and cluster assignment and instruction scheduling. Loop unrolling is yet to experimented in the proposed architecture, we expect a great performance
improvement with loop unrolling. Impact of the variation of the memory instruction latency has been already studied in our architecture.

In [54], the authors have extended the work presented in [53] to deal with the problem of memory coherence. Two local scheduling techniques were used: one for instruction scheduling within clusters and another for synchronization among memory dependent instructions.

Their work concluded the following findings: (1) Out of order execution can implement load bypassing (that is, executing loads before stores to reduce critical path); (2) For static scheduling loop unrolling and peeling are turned off for big or outer loops; (3) For out of order execution the two iterations can overlap; and (4) the static code uses a stall for a cache-miss whereas the out of order execution hides the latency of misses by allowing other operations to execute.

The study reveals that load bypassing and hidden cache miss latency improves the performance of restricted out of order processor. But with general code percolation support the in order processor out performs the restricted out of order processor.

### 2.2.2 Methods for Instruction Set Generation and Selection

For an application specific processor, efficient instruction set generation is essential to meet the performance and other design constraints.

**Instruction set mapping:**

In high-level synthesis, replacing a group of primitive instructions with a complex instruction, known as instruction mapping, is a powerful transformation to improve performance. Most of instruction set generation approaches are based on such a mapping technique, where the instruction set is generated from a base instruction set.

The main issue in re-targetable code generation is pattern matching and selection of in-
2.2. TECHNIQUES USED IN ASIP DESIGN

Instructions. In high-level synthesis replacing a group of primitive instructions with complex instructions is the powerful transformation known as instruction mapping to improve the performance. Matching is performed to find the appropriate complex operations in the instruction sets corresponding to the primitive operations in data flow graphs. Selection is to select small number of patterns to cover the graph to minimize the size of instruction set. This requires the replacement or addition of complex hardware units. The graph matching is used in [76, 19, 12, 13, 63, 86]. Again it is separated into pattern matching [76, 19, 12, 13] and instruction-set matching in [63, 86].

In [36], the authors proposed a template based approach. The approach consists of three tasks: template matching, instruction selection and clock selection. The template matching finds the control/data flow graphs of primitive instructions that match with templates of complex instructions. The complex instructions can be implemented with specialized hardware units for high performance. Then among all the found matches, the instruction selection chooses some matches such that by using the related special complex instructions the overall critical path (under a given clock period) can be reduced, hence improving performance. Since the clock period affects the formation of critical path, the instruction selection task is repeated with different clock period and the clock selection is performed with an efficient algorithm that tries as few clock periods as possible to quickly achieve best design result.

![Template Mapping](image)

**Figure 2.23: Template Mapping**. Taken from [37]

The authors extended their work in [37] to use partial pattern matching so that special
hardware units for complex instructions can be highly utilized. Figure 2.23 shows partial matching (d) as compared to complete matching (c).

A predefined library of patterns is used for matching and covering of recurring patterns in [108]. Source code is translated into intermediate code and to source control/data flow graph. Similarly the target micro operations are represented as patterns. These patterns are matched to the source code graphs. Dynamic programming is used in the instruction selection. Here resource classification is used to identify the architectural features. The behavioral representation can contain explicit information. This work is implemented in CodySyn system shown in Figure 2.24.

The above approaches do not deal with template generation. For efficient template generation, authors in [84] proposed a bottom-up tree pattern matching approach. With this approach, the intermediate representation of an instruction program is represented by a tree and an efficient bottom-up matching strategy is used to generate templates for automatic instruction set selection.

In [137] a tree structure created by a pattern queue and a flag table is used for instruction matching. Pattern selection is done using genetic algorithms to get optimal or near-optimal results. The model used in the work is a linear programming model with an integrated approach to solve several code generation problems. A backtracking genetic algorithm is used to achieve better covering of the data flow graph.

An automatic recurring pattern detection is used to generate custom instructions in [12]. These patterns extracted from a number of applications leads to re-use of hardware modules. The work consists of two phases: pattern library construction phase, and matching and covering phase. The algorithms used during these phases are given in detail in [13]. The matching algorithm is based on incremental matching. Using conventional methods requires a predefined pattern library but in this approach the extended matching algorithm automatically constructs libraries. The covering heuristic is similar to the tree covering dynamic programming approach.
Instruction set model is an essential part of the re-targetable code generation. An instruction model should reflect resource conflicts, instruction set conflicts, alternative code versions, side effects and residual control of the ASIP architecture. The authors in [104] have analyzed the demands of the instruction set model and presented a formal model to meet these demands. The model is used for the ASIPs with programmable micro coded controller architecture and is based on Register Transfers and No-Ops.

In [75, 74] the instruction set generation has been treated as a modified scheduling problem. The micro operations of the benchmarks are represented as data/control graphs and scheduled in time slots with different combinations. This is subject to several constraints such as resources, size of the instruction set and data/control dependencies. The problem formulation is based on the basic block. Another approach using scheduling to generate instruction set is presented in [73]. This process consists of the compilation of a benchmark, execution trace generation, formation of code segments, optimization of code segments, and instruction set formation. Reccompilation is done by symbolically executing the register transfers feasible with the data path. On completion of a successful recompi-
lation a pattern matching is made in the register transfer operations to fill the fields of the instructions. The overall instruction selection process is shown in Figure 2.25.

![Figure 2.25: Instruction Set Derivation. Taken from [73]](image)

A structural model is used to generate complex instructions from primitive instructions [32]. The path analysis and system utilization techniques for static performance evaluation in instruction selection are discussed in [111]. The authors of [32] use this static timing analysis technique based on abstract simulation to get the maximum execution time of the micro operation list. Complex instructions require space in micro-ROM therefore they are kept to a minimum. Complex instruction generation problem is formed as a subset-sum problem. The MOPs are grouped into a set, provided they can execute in single cycle and without dependency. The sets are represented in a conflict graph. The sets, which can become one complex instruction, are named c-isomorphic. Among these if one c-isomorphic is selected, all sets related to that c-isomorphic will be selected. A modified subset problem solver finds the optimal solution to generate single cycle complex instructions (SCC) and the multi cycle complex instructions (MCC). Here the MOPs
2.2. TECHNIQUES USED IN ASIP DESIGN

which can execute sequentially or in parallel are grouped as sets.

The instruction set selection in a re-targetable compiler has been handled by operation coupling [123]. A bundle is a micro operation sequences directly coupled. A combined data path and instruction set model along with the benchmark data/control graph is used. The heuristic algorithm incrementally couple the nodes until it reaches proper bundling. The instruction generation approach has been used in the CHESS design tool [152].

Several aspects of code generation in AVIV re-targetable compiler such as instruction selection, resource allocation and scheduling are discussed in [70]. A split-node direct acyclic graph model has been used to capture all the possible operation bindings with the necessary data transfers.

Implementing operations in ASIP is an important task. An operation can be implemented in hardware or software. The authors in [77] have presented an integer programming formalization to solve the instruction set implementation method selection problem-type1(IMSP1). This was done assuming that no more than a single operation would use a particular hardware-module. It is in contrast to the module-sharing concept. This IMSP as a NPhard problem might be time consuming, therefore a branch and bound method is used to reduce the time consumed by the IMSP problem.

A tight Lower bound heuristic function is used to prune the non-optimal solutions. This problem has been extended to a module sharing problem (IMSP2) in [9]. An additional search is conducted in the tree space to integrate the module sharing operations into a single hardware-module. The IMSP2 solver does select the optimal implementation of the functionality and if possible integrate them together. The IMSP solvers predict performance and area to facilitate instruction set selection.

A comparison of IMSP1 [13] and IMSP2 [9] is given in Figure 2.26.

The IMSP2 has been extended to IMSP2P [22] toward the pipelined architecture to find and resolve the pipeline hazards. This co-design system uses a pipeline scheduler to detect and solve the pipeline hazards. Branch and bound method and module sharing
with reordering are used in the problem solver. IMSP2P solver produces selection of optimum implementation for operations and scheduling of basic blocks for a pipelined ASIP for maximum performance under given constraints.

In [20] the authors have presented an algorithm for selecting an optimal pipelined architecture. An adaptive database approach enhances the optimality of the design through very accurate estimation of the performance of an ASIP.

In [21] the IMSP3P instruction selection problem is formalized as a combinatorial optimization problem to design a pipelined ASIP with least hardware cost, subject to execution cycle and power constraints.

The IMSP-2P problem is extended to Multiple Identical Functional Unit selection IMSP-2P_MIFU in [76].

The hardware/software partitioning problem IMOP-2P in [19] is the extension of the previous combinatorial problems discussed in [21, 76] toward CPU RAM ROM size optimization. The solution searched is to increase the performance subject to area of CPU
2.2. TECHNIQUES USED IN ASIP DESIGN

and memories. The branch and bound method with a lower bound is used to search
the search tree by pruning the non-optimum solutions to expedite the search. The lower
bound functions are generated based on performance and area constraints. The output of
the method are the optimum implementation of operations, scheduled basic blocks, CPU,
RAM, and ROM sizes. The heuristic reordering and module sharing is similar to the one
described in [9], and the adaptive database generator of [20] is also used to calculate the
execution cycle estimation of software implementation of operations.

The algorithm presented in [86] is for instruction generation of hybrid reconfigurable sys-
tems. This algorithm profiles the data-flow graph and iteratively contracts the edges to
design templates. This method finds the best templates irrespective of the area and size
but the feasibility to add these restrictions are said to be easy. Apart from making com-
putational units, the instruction generation can be used to identify the tightly sequenced
operations to create soft reconfigurable macros. The soft macro can be placed in the re-
configurable fabric and it could reduce the synthesizing time and cost for reconfigurable
architectures. This approach is to cluster the repeated successive patterns based on ap-
pearance frequency in the graph rather than the execution frequency of the patterns. A
continuation of this work to find the parallel templates are discussed in [23]. Parallel tem-
plates require regularities without data dependencies. For a given graph clustering is done
for most suitable sequential or parallel patterns.

The method to identify and construct a static resource model from a given instruction set
is discussed in [164]. The instruction set constraints of a highly encoded instruction set is
replaced with a static resource model (SRM). The architectural constraints are modeled as
an instruction set or by using functional units and its operations. The resource constraints
are static.

The method described in [113] is used to handle the strong phase coupling between the
scheduling and register binding. The inequalities of the instruction set are translated into
virtual resources. Each operation uses the virtual resources it is associated with. These
virtual resources help yield a better schedule. This SRM model is used for instruction
set design in [165, 166] along with the conventional high-level synthesis method. Performance is evaluated with a default ISA and processor, and if not satisfied, then bottlenecks are identified to modify the SRM with virtual resources. A lower bound on the initiation interval of a pipelined schedule is obtained to identify bottlenecks. The resulting instruction set is used in an efficient classic resource constrained compilation, and its performance is better than explicit instruction selection. With this static resource model, the scheduler has the opportunity to generate better schedules in terms of timing and register requirements.

![Figure 2.27: Scheduling comparison. Taken from [165]](image)

The use of extended instructions will reduce the code size, energy and improve performance. A more general algorithm to choose maximal speedup convex sub graphs of an application is presented in [14]. This method identifies the patterns with any number of outputs depending on the user specified limit. Any kind of disconnected graphs can be identified to execute in parallel. Identification and selection are coupled to handle at once. The data path of a basic block is represented by a DAC. All possible sub graphs are analyzed to find best sub graphs to convert into extended instruction. The chosen sub-graph has to satisfy constraints such as input and output nodes and should also be convex. The algorithm first finds the single sub-graph or cut in a basic block and also finds the optimal and near optimal non-overlapping cuts in many basic blocks. To make it an easy task, the algorithm does a topological ordering in the graph before searching it to create a search tree. The binary search tree nodes represent the possible cuts, and if some constraints are violated at an early tree node, the sub tree is pruned. The optimal selection algorithm is
used to get multiple cuts in a single graph just by increasing the number of cuts per block and by iterating the algorithm.

\[
\begin{align*}
& t = n >> 24; & & //1 \\
& r = t & & 0*ff; & & //2 \\
& a[5] = t; & & //3 \\
& m = b[0]; & & //4 \\
& y = r + m; & & //5 \\
& m = b[0]; & & //4 \\
& y = CustomerInst (n, m); & & //1,2,5 \\
& t = RUR (0); & & //1,2,5 \\
& a[5] = t; & & //3
\end{align*}
\]

![Illustration of template insertion: (a) original code fragment and its data dependence, and (b) modified program to enable insertion](image)

Figure 2.28: The Illustration of Template Insertion. Taken from [142]

The authors in [142] have devised a method to automatically select custom instructions from operation patterns for an extensible processor. Figure 2.28 depicts the template insertion for a given fragment of code. The cost functions and pruning techniques are developed to help custom instruction generation. Proper logic-synthesis and cycle-accurate instruction set simulation of the custom instructions are done to estimate the performance and hardware overhead. The effect of individual and group of custom instructions are evaluated to select the best combination to improve performance and meet the area constraint. The Tensilica Xtensa is used as the target platform. From the given program several program dependent graphs are generated such as control dependence graph with control block, data dependence graph, and control flow graph. Program is simulated and profiled at functional and line level. Performance, energy, energy-delay product are in-
included in the criteria to rank the control blocks.

The authors have selected custom instructions inside control blocks. The candidate is selected and implemented as TIE to be compiled to get RTL description and evaluated for the objective metric. If the TIE does not fulfill the requirement, this step will be repeated until a suitable instruction is generated. The authors have proposed another scalable methodology to generate extensible instructions in [143]. A soft instruction template concept which allows the addition or deletion operation as per the requirement, while considering the global design space is used. The algorithm performs hardware resource budgeting, local optimization, and global exploration. Area consumed by a single instruction can be reduced by sharing area by using same hardware module or same number of input/output. With this instruction generation approach, the input C program is transformed into an intermediate stage representation. The profiling statistics, dependence and call graphs are then generated. For each basic block based on the available information a largest template is generated. Pruning is left for the next phase. On completion of the largest template generation in all basic blocks the templates with identical properties are merged into together and they will have the attributes of all the merged instructions. The generated templates are included in the intermediate representation of the program and given to the next phase. In order to make the task simple the program is divided into small programs. The small search space solutions are then converged into the main program. The attributes of the templates of a function including area are computed dynamically and the instruction templates are scaled based on the target area. This is done to the functions in the same level and then repeated at the next level. Finally when all the functions are done a set of templates satisfying area constraints will be selected.

The author of [44] studied different encoding for a library of extended instructions to meet the instruction bit width constraint. Their approach is applied to pipelined RISC processor with multi-cycle instructions implemented in configurable architectures. The restriction of instruction size was overcome either by using the un-defined instructions of an ISA or by generating basic ISA from a portion of the instruction code space and in-
2.2. TECHNIQUES USED IN ASIP DESIGN

Including the complex instructions in the remaining limited space. The library of complex instructions is generated as shown in Figure 2.29 from the basic assembly of the application program, and information such as frequency of the instruction and the required bit length of operands etc. The target processor is represented by the basic instructions and structural information of each operation. Complex instructions are formed from assembly code with the constraints of the processor. A resource-constrained scheduling is used to find the saving in clock cycle. Cost is a function of bit size. The instruction set selection involves choosing the best-superset instruction, multiple cover instructions by repeatedly using a heuristic algorithm. The reduction in the opcodes and operand sizes leads to fast fetching and decoding thus improving performance. Complex instruction generation has been carried out in the proposed architecture. We used the un-defined instructions of an IAS or the un-utilized instruction to include the newly added complex (special) instructions.

Figure 2.29: Instruction Synthesis. Taken from [44]

Instructions are grouped depending on the total required operand length. The algorithm used to solve this problem in [115] packs the opcode of instructions with the same operand length within the operand length of the next set of instructions. Some bits are used to distinguish the instructions groups. This process leaves redundancy bits in the opcode field. The instruction set is partitioned into several groups and subgroups according to the different parameters or attributes and hierarchical processor models are generated using LISA processor modeling language. The authors use a hybrid of local and global encoding. Customization of Instruction encoding has been done in the proposed architecture, in which a method similar to the one in [115] has been used to group the instructions. We do the encoding only at local level i.e within each pipe.
The authors of [132] have discussed the Explicitly Parallel Instruction Computing style architecture to have more instruction level parallelism with less complex hardware. It is an evolution of VLIW but consists of many superscalar features. EPIC denotes a class of architectures, which can use its features. Some compression techniques to handle the problem of no-ops in non-empty instructions are discussed.

A methodology to automatically generate instruction formats for EPIC processor is presented in [6]. EPIC processors have a VLIW architecture that allows multiple operations grouped into one instruction and executed in parallel in the processor. A multi-op format known as canonical format with many slots, each corresponding to a functional unit is used. The code wastage arising from no-ops is solved by using multi no-op field to explicitly specify the number of no-ops [17]. But in case of non-empty instructions we need to have no-ops. Multitemplate is used to handle this matter in [124], which discusses a number of possible solutions. More detailed explanation of instruction set format generation is given in [7].

Some techniques have been used in the PICO system [5] to reduce instruction length during instruction generation and program assembly. Their encoding uses variable length op-code. Compression technique with Huffman encoding is used to investigate the possibility of minimizing the average instruction length. The authors use a canonical instruction format with each slot of the format corresponding to a functional unit depending on its position. Apart from canonical formats there are custom templates used. Custom templates are generated using the process described in [6].

The Tensilica’s Xtensa facilitates the addition of extensible instructions using a language called TIE (Tensilicas Instruction Extension) as mentioned in [60]. The application is compiled for the base processor and the performance is estimated. If that is not good enough designer can profile the application, find the bottlenecks and suitable extensible instructions, implement the instructions in TIE, and debug the instructions. Once the instructions are verified the application is recompiled and performance is estimated. A typical TIE development cycle is shown in Figure 2.30.
In [61], the TIE language is incorporated in AutoTIE system to automatically extend base processor with application specific register files, operations and instructions. This method examines several possible designs to choose the best to meet the cost and performance. A given program in C/C++ is profiled to identify the critical loops and the compiler analyzes the application to gather information of operations in loops. The analysis is used to generate extended instructions. The different extensions are generated and estimated. Finally the application is compiled with the chosen extensions to optimize the performance.

A compiler directed design methodology is given in [64]. Pre-fabricated components such as MAC, floating-point unit, multiport memory, and non-pipelined memory unit are used as parameters. The performance with these units and alternatives with conventional units
are compared. The profiler computes the execution frequency. The scheduler computes the execution time for the schedule based on the profiled information. The approach used by scheduler is the list scheduling.

The authors of [29] have proposed a design flow with a methodology for rapid selection of extensible instructions from a pre-designed instruction library. The library consists of optimally designed instructions. The Xtensa environment is used to generate extensible configurable processors. The authors have used a methodology to select the program sections to generate instruction extension and a heuristic algorithm to select the pre-fabricated co-processors/functional units. A greedy approach is used to select the pre-fabricated units and TIE instructions. The Xtensa core along with all existing co-processors is simulated and the effectiveness is computed using clock cycle, area, and clock period to find the best co-processor core combination for a given area constraint. After that TIE instructions are selected. Finally the execution time of the application in the configured processor with extension instruction is computed. The change in latency due to the TIE is taken into account during the estimation of execution time. This is done iteratively to select the best set of TIE.

As an extension of the above work, in [31] the authors present a partially automated system to generate and select the extensible instructions. The configurable processors are used to simulate and profile the application to get information. The best configurable processor is chosen by evaluating the effectiveness of performance/area ratio and area. The program is profiled and segmented, according to ranking based on a fitting function reflecting the speedup/area ratio. Provided no matching is found in the library, these sections are converted into instructions and characterized using the Tensilicas Xtensa tool and relevant tools. Instructions are selected based on the priority of speedup/area ratio of that instruction. The total area of the preconfigured processor with the selected instruction should be within the area constraint. This is repeated to add instructions until the area constraint is met. Finally performance/area of the selected processor and instructions combination is estimated.
In the work mentioned above matching the critical program sections to the pre-designed instructions is a critical task. This task is automated by MINCE tool in [30]. Figure 2.31 shows the ASIP design flow with MINCE. The MINCE tool consists a translator, a filtering algorithm and a combinational equivalence checking tool. The program sections are selected by the method described in [31] and MINCE tool does the translation of code segment into HDL code. First the code is converted into register transfer level operations. The conversion is for easy matching of code segments to the instructions in HDL. The instruction hardware library consists base hardware modules that are assembler instructions are implemented in HDL. The operations of the translated code segment are matched to these modules and brought together to make a combinational hardware module of an extensible instruction. The unmatchable code segments are filtered. By using standard methods both HDL files and hierarchy modules are flattened to a gate level description. MINCE verify the combinational equivalence of the HDL files to assure they
Another scalable custom instructions identification method for extensible processors is given in [162]. Heuristic techniques have been used to explore the design space to select the optimal set of custom instructions. For a given DFG of a code fragment the algorithm enumerates all possible candidate instructions.

Instead of designing a complete ASIP, a top down approach of re-using existing IPs is given in [40]. Here a subset of an existing instruction set is used to generate the ASIP. This could be done by either modifying the behavioral descriptions or by hand-optimizing the RTL descriptions. The former gives a better saving in energy mostly from reduced data-path and interconnections. The reduction of complex instructions leads to higher code size and more clock cycles, but energy saving per clock cycle is more due to simplified logic and interconnections. This method enables the usage of existing compilers, simulators for future applications. Our methodology too we use the existing ISA and the hardware library attached to the processor designing tool. We do modify the RTL descriptions. We need not have to reduce the complex instructions since our ISA was simple. We too had simple logic and interconnections due to simplicity of the ISA. Although we could use the existing simulators, due to the novelty of the architecture we were unable to use the existing compilers to the full extend.

In [56] a method to increase of code density with fixed length encoding is used to reduce power consumption of memory. Trace files generated by LISA simulator are used to perform optimizations based on the histogram of the operands. Another way of achieving this is to use look up tables for operands when all possible values of the operands are not used. But this look up table is subject to design constraints. The optimized encoding with inverse op-code has been implemented to save power.
2.2.3 Bypassing Network for Processor-pipeline Performance Improvement

Bypassing (also called forwarding and sometimes short circuiting) is a hardware technique that tries to reduce performance penalties due to the data hazards introduced by the micro processor pipeline [71]. Forwarding units send the results of previous instructions to the following instructions by connecting the inter-stage pipeline registers to the functional units. Bypassing is a simple technique but can be very costly in terms of area, especially in multi issue and deeply pipelined processors. The wide issue machines can have significant delay due to the bypass logic. The bypass circuits increase not only the area and cycle time, but also, the power consumption. The increase in parallelism and long pipelines in modern processors will tend to increase the problems.

The bypass interconnect affects performance in two ways. The forwarding mechanism in a VLIW processor can reduce the number of stalls and increase the performance. At the same time the complexity of the logic might increase the clock cycle width. In [3] different bypassing interconnections are used for increasing the level of forwarding among the functional units of a VIPER VLIW processor. The analysis includes both the frequency of stalls and cycle time penalties and shows that bypassing network that completely connects all of the functional units does not provide for enhanced performance when its cycle time penalty and the area are taken into consideration.

To reduce the bypassing network complexity, incomplete bypassing network structure was proposed in [8]. The researchers have used cycle level simulation model of an integer and floating-point pipeline processors to run SPEC92 benchmarks and found that half of the instructions used bypassing. Interlock stalls will be induced for missing bypasses. The effect of incomplete bypass network in many pipeline processors has degraded the performance as shown in Figure 2.32. To reduce interlock stalls, two code alteration approaches have been used. One is interchanging operands of commutative operations; another is rescheduling the basic block to avoid missing bypasses.
The register file access has to be done in a single cycle in a pipelined processor. But in many VLIW processors with 128 or even more number of registers it is hard to maintain the single cycle latency with a high rate of frequency. A method combining the register bypassing and caching is discussed in [163]. It is believed the conflict between high processor frequency and a register file with many ports and a large number of registers can be resolved by exploiting the frequent bypassing of register operands within processor pipelines. The bypassing is frequent enough to use a cache in place of the register file (called Register Scoreboard and Cache). There should be enough bypassing register values to replace the register file with only a small cache. In order to get the high performance gain the hit rate in the Register Scoreboard and Cache should be high and the access to the backing store has to be fast enough. A fully associative cache with a Least Recently Used replacement policy is used and the bypassing network is complete. It is shown that the method is effective for windowed-register architectures.
Most of the operands are used only within a few consecutive instructions. It is not worth storing them in the Register File. As mentioned in [110] in dynamically scheduled superscalar processors using mechanisms like the reorder buffer, the live ranges for these short-lived variables may occur entirely within the reorder buffer. Therefore there is no need to retire (commit) the values produced by these live ranges to the register file. Therefore a compiler analysis and a simple architecture extension to avoid the useless commits of the values generated for these short-lived variables is proposed. Moreover, an extension to the existing register allocation mechanism that does not assign these short-lived variables to locations in the register file is proposed. Their analysis has revealed that the compiler could capture useless commits around 90% when the reorder buffer size is 16.

With the assumption of a perfect cache, and a large load/store buffer the improvement in performance for a given register file size depends on the reorder buffer. As the reorder buffer size increases the performance increases. Here the perfect cache means that there is enough functional units to execute the operations that are not related to memory accesses and that the latency of these operations is one cycle.

All applications might not require a complete bypass network. In [45] a systematic design customization process along with a bypass-cognizant compiler scheduler is proposed. Compiler scheduling for sparse bypass processors is accomplished by prioritizing function unit choices for each operation prior to scheduling using global information. Bypass paths are exposed to the compiler and intelligent scheduling techniques are used to effectively use the partially bypassed machine. The applications from the same Bench Marks have different requirement of bypass paths. It has been shown that in a five issue VLIW processor on average a 70% reduction in non-zero utilized bypass can be achieved while maintaining 90% performance. Our experiments also proved what the authors have mentioned.

Pipelined functional units and multi-cycle register files may require multi-level bypass networks to assure the availability of the result in the cycle next to its generation. The discussion in [24] elaborates the effects of a reduction of one level of bypass in a multi-
level bypass network.

In multi issue processors like VLIW, it is essential to copy the operands from one cluster to the other if there are dependencies across clusters. In the work presented in [25] a few bypassing lines are used to forward operands stored in pipeline registers to other datapaths. Pipeline bypasses can be added between datapaths within the same cluster or between datapaths in distinct clusters. The goal of inserting a bypass interconnection between two datapaths inside the same cluster is to reduce the number of NOP operations required to solve the data-hazard between the dependent instructions in the datapaths. By assigning a bypass interconnection between two datapaths in distinct clusters, the number of copy operations required to use the copy-bus are reduced. If there is a dependency between the instructions in different clusters, the copy operation is issued by the compiler. The presented scheduling and portioning algorithm assigns higher communication data-paths to the same register file, while tailoring bypass interconnections to the application. Therefore, a potential performance improvement is achieved.

A power optimization in the design of VLIW processor architectures given in [127] avoids the writing/reading of the short-lived variables to/from the register file. The decision to write-back to RF is by the compiler’s static scheduling algorithm. To evaluate this approach the authors have set up an experimental methodology to evaluate the power savings on a set of multimedia applications executed by an industrial VLIW embedded processor, jointly developed by Hewlett-Packard and STMicroelectronics [46] for which an optimized compiler has been developed to exploit the architecture parallelism.

In [136], an approach that uses an Operation table was proposed to detect data hazard in the incomplete bypassing network. The operation table was used to model resources and register usage. This method detects resource and data hazards more accurately. The detection technique has shown 20% performance improvement over the best performing code generated by GCC on benchmarks from the MiBench applications for Intel XScale.

As we have seen so far varying partial bypassing in pipelined processors in embedded
system was an effective way of making performance, area and energy tradeoffs. It is important to have a partial bypass sensitive re-targetable compiler and a multi-dimensional exploration framework that estimates the power and area cost of the partial bypassing.

The authors have extended their work and produced the PBExplore [135], a framework for compiler in the loop exploration of partial bypassing in processors. It used a generic bypass-sensitive compilation technique which enables the multi-dimensional exploration of the partial bypass design space.

Our proposed architecture with complete forwarding path faced the problem of high area, cycle-time and power consumption and thus we opted to do the customization of forwarding paths. When customized there was a considerable improvement in the above mentioned metrics. While customizing the forwarding paths we did re-arrange the code so that there is no need for extra techniques to detect the data-hazard caused by the removal of the under utilized forwarding path. Forwarding between datapaths (Pipelines) is also available and no copy operation is required. The simplified architecture avoids the requirement of complex compiler to handle partial bypassing network.

2.3 Testing and Evaluation Methods

Verification and Evaluation are important tasks in design. ASIP evaluation techniques are mostly either scheduler based or simulator based. In scheduler based approaches, resource constraint scheduling and execution cycle estimation are performed. Profiler is used to obtain the frequency of operations. In simulator based approaches the application is simulated on the architecture model.

Testing using architecture model simulation [33, 16, 161] consume more time and not suitable for design space exploration system design.

The re-targetable simulator described in [88] is constructed for a stream oriented application specific dataflow architecture. For each instance of the architecture template a
specific simulator is derived in 3 steps as shown in Figure 2.33. An architecture instance is constructed, an execution model is added, and the executable architecture is instrumented to obtain performance figures. Architecture elements and execution mechanisms are coordinated using PAMILA[49] constructs. The proposed architecture do not need a re-targetable simulator it is simulated with the ModelSim HDL simulator therefore there is no need to generate a simulator for each design.

Figure 2.33: Steps within the Object oriented Re-targetable Architecture Simulator. Taken from [88]

The IMPACT re-targetable instruction level simulator Lsim is used to evaluate the power consumption of ASIPs in [90]. A search algorithm is used to choose the low power architecture within the area constraint.

The parametrized simulator in[39] uses pre-defined modules and the architecture description to estimate performance and resource utilization. The simulator uses an event-driven
algorithm and supports various memory hierarchy structures (such as D-cache, I-Cache, bank interleaving).

The method in [96] uses model analysis based on the data transfer to estimate performance. The data transfer is focused on bus transfer operations. The evaluation is performed for different bus architectures. The methodology is given in Figure 2.34.

![Figure 2.34: Two phase performance-analysis methodology. Taken from [96]](image)

Customizable IPs are difficult to be verified. In [139], a verification database with different IP-configurable constraints is used. The assembler code pre selected by the IP-configurable constraints, forms the verification database. A model of IP is used to derive expected responses suitable for any possible configuration of the final ASIP implementation. The cycle-based verification is performed by stimulating the RTL model with the assembled stimuli and by comparing the output against the expected responses.

An approach [65] with two stages (one is architecture independent and other is architecture dependent) is discussed. Both the Intermediate Representation (IR) of application
program and processors are used. First the parameters extracted from the application and the processor, which are uncorrelated, are eliminated. In the next stage remaining processors are evaluated for number of cycles, by an architecture constrained scheduler that schedules the application on a processor architecture. The processor architecture description file contains the operations that are executed by the different functional units along with their execution time, concurrency of execution and constraints on their concurrency. Figure 2.35 shows the processor evaluation methodology.

![Diagram](image)

Figure 2.35: Processor evaluation methodology. Taken from [65]

An evaluation scheme based on phase-accurate simulation with only delay model and trace, for pipeline architecture is proposed in [89]. This method is for the language-based hierarchical methodology. In such a methodology the ISA is developed before the pipeline architecture is developed. An abstraction level called token-level, which is not a cycle accurate model, since it does not provide cycle-true behavior (including the details of storage memory) is generated for fast evaluation. This model uses a trace generated by the fast instruction set simulator and is free of control dependency to evaluate the pipeline
2.3. TESTING AND EVALUATION METHODS

A system level profiling is proposed in [150] for architecture level performance estimation. This method automatically constructs an execution order graph and execution dependency graph. Graph analysis is used to speed up the performance evaluation. A system-level model is written in SystemC for profiling and the amount of data transferred, execution order of the data transfers and data processing are obtained.

In [52] a method for generation of test microcode based on analysis of the structural register transfer level implementation of Application specific Programmable Processors or Application Specific Instruction Processors is given. The test micro code dictates a new control/data flow in the circuits, and it helps to exploit the inherent programmability of these circuits during testing. The new control/data-path is used to justify pre-computed test sets from the system primary inputs to the module inputs and propagate error responses from the module output to the system primary outputs. When the method is unsuccessful in testing the data path completely, some test multiplexers are added to the data path. Thus, this method tests all modules in the data path completely.

An automatic tests derivation is discussed in [18] from the descriptions of the user-defined extensions. This method alleviates the problem in using predetermined tests for the automatically generated pipeline control logic in user-defined extensions. These tests are derived from the ISA description and the scheduling information of the new instruction set, not the functionality of the instruction set. The extension to the base processors could be described using an ISA description language such as Tensilica Instruction Extension language [159]. This algorithm inserts no-ops in between the instructions to avoid control or data hazards.

In [145] the functional verification method of the super-scalar multi-issue Alpha processor is discussed. This is a simulation based verification performed on the logic design. The functional verification uses implementation-directed, pseudo-random exercisers, supplemented with implementation-specific, hand-generated tests. The Alpha micro-
processor has been modeled in different abstraction levels such as performance model, RTL/behavioral model, a gate-level model, and a three-state simulation model.

2.3.1 Summary

A review of the recent research work in the ASIP design area is provided in this chapter. Application analyzes, architectural exploration, re-targetable compilation, ASIP design methodologies, techniques used in designing, and testing methods have all been discussed. Architectural exploration methods based on architectural description languages, templates, and a combination of these two have been discussed with examples. Tools and methods supporting the architecture exploration such as profilers, compiler front ends for analysis, re-targetable compilers, simulators and estimators are discussed.

Techniques supporting the refinement process of ASIP generation, such as instruction scheduling, instruction generation, and bypass network selection based on different approaches were given with their pros and cons. Instruction scheduling is vital to design architecture for high performance. From the works elaborated in the literature, it is evident that assigning and scheduling plays a critical role in processor design. By making the assign and schedule approach more sophisticated, it is possible to achieve better utilization of resources. The research on bypass network has shown the benefits of using a bypass network along with the benefits of selectively using the by pass network for different architectures.
CHAPTER 3

Overview of the work

Classification of architecture could be done based on different features such as: single cycle, multi cycle, or pipelined based on instruction execution; RISC, CISC, VLIW, etc with respect to instruction set; and single issue or multi issue depending on the number of instructions issued per clock cycle. On the other hand, a single architecture may contain a combination of these features, such as a pipelined RISC processor. These different features of architectures are the result of approaches aimed for improving performance while reducing area, power consumption and cost. Fabrication technologies have made it feasible to integrate many processors in a single chip.

Superscalar, multiple pipeline processors are common in most modern general purpose processors, usually dedicating a few issues for general processing and others for floating point processing. For a general purpose processor, high performance cannot be guaranteed for a specific application due to its (general purpose) nature. 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 additional area cost. However, researchers so far have mainly focused on single pipeline structures in ASIPs. 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. The work described in this thesis is in-
spired by the two methods usually taken for instruction level parallelism: one, dynamic
hardware-based approach (superscalar); and two, static software-based approach (VLIW).

Both approaches entail parallel hardware processing units, but they differ in the paral-
lelism exploitation scheme used. Superscalars use complicated hardware to dynamically
identify parallel processing opportunities. They are, therefore expensive and suitable
mainly for high end systems. VLIW processors, on the other hand, use compilers to
find unrelated instructions for parallel execution so that the hardware structure is sim-
plified and less expensive. With given parallel functional units, in order to schedule the
instructions to maximally utilize available processing resources, a sophisticated compiler
is required. In this method, static scheduling is used to to fix the parallelism, and the
compiler is expected to take care of hazards when the scheduling is performed. Thus the
hardware circuits will be less complicated.

Compared to normal VLIW processors, the design proposed in this thesis is strongly
application oriented. The number of pipelines is determined specifically for the application.
Further, the functional units on each pipeline is purely based on the application itself.
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 proces-
sor thus created will be a heterogeneous multi-pipeline processor. Heterogeneity of the
processor arises by the differing sets of instruction issued to the two pipes. Since the ap-
plication is well understood, it is possible to do so, improving performance and reducing
the area cost required.

The design flow of the architecture proposed in this thesis is shown in Figure 3.1. The
input to the design is a C program. With the GCC cross compiler of the target instruc-
tion set, the C program is compiled into an assembly program. The assembly program
is scheduled into multiple pipeline streams by an efficient scheduler that we have imple-
mented. This scheduler will carefully explore the parallelism within the assembly code to
yield a compact schedule. Heavy functional units will cost more area and power, there-

Figure 3.1: General Design Flow of the methodology
fore functional units such as an ALU are added to a particular pipe based on demand. During the scheduling process the cheapest functional units are given priority in selection. Scheduling is done on the basis of basic blocks. The resulting pipeline data paths are non-identical thus the processor is heterogeneous in nature.

The scheduled code is a multi stream code. Each instruction stream or data path has its own instruction set. Each data path is implemented as a pipelined data path and is called a pipe. The processor structure which consists of multiple pipes, each of which executes a different set of instructions, is called a Multi-pipeline heterogeneous processor. The generation of object code and processor core for the multi-pipeline processor are generated using tools. The assembly code corresponding to each pipe is assembled into object code by the assembler. These object codes for each pipe are concatenated to generate the final object code for the processor. Pipes corresponding to each instruction set is generated separately using a single pipeline processor generation tool, ASIPmeister. The generated pipes are fused together by modifying and adding, the HDL description of single pipe processors.

The generated object code of multi-pipeline processor is executed on the generated processor to verify the functionality and to evaluate for clock cycle, area and power.

In this design flow some additional techniques are used to reduce cost. The scheduled code is analyzed for data bypass requirements and the non-utilized and under utilized bypass paths are excluded in the design. A local scheduling is done to remove the under utilized bypass paths. Further, an encoding technique is used to reduce the instruction width. By reducing the instruction width, the code size is reduced.

In chapter 4 of this thesis, the author describes a two pipeline heterogeneous processor to demonstrate the feasibility of the multiple pipeline ASIP processor. The dual-pipe work is extended in Chapter 5 for multi-pipe heterogeneous ASIPs, where the number of pipelines are customized and full forwarding technique is used to reduce pipeline hazard. To further optimize multi-pipe ASIP design for high performance, low area and small
code size. Chapter 6 investigates customization techniques which focuses on forwarding network and instruction encoding.
CHAPTER 4

Dual-Pipeline Heterogeneous ASIP Design

Introduction

In this chapter 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) for the two instruction sets, utilizing the ASIP-Meister Tool Suite, and fusing the two VHDL descriptions to construct a dual pipeline processor.

The design of the Dual Pipe Processor can be seen as an initial step for the Multiple Heterogeneous ASIPs Generation. Multiple Instructions are issued on the same clock cycle in many modern processors to improve performance. In contrast, current ASIPs provide only a single pipeline. The work described in this chapter explores the possibility of increasing the number of pipelines to two, and describes a methodology to enlarge the design space available to an ASIP architect.
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.

Chapter Organization

The rest of the chapter is organized in the following way. Section 4.1 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 4.3. Finally, the chapter is concluded in section 4.4.

4.1 Dual Pipeline 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 4.1.

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 enough 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. Two separate buses are used for instruction fetching and data fetching. No bus arbitration is required.

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 4.1, 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 4.1). 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. It is preferred to execute the add in the right pipe instead of in the left pipe. This is due to the reason that the switching activity of an adder is less than that of the ALU which is more complex. A methodology for determining the functional units that have to be duplicated is given in section 4.2.

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.

### 4.2 Design Methodology of Dual Pipeline Processor

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 4.2.

#### 4.2.1 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 4.2.

A C program is compiled and assembly code is produced for a base single pipeline 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 [100] and [141].

Figure 4.3 gives an example of how an assembly code is transformed by replacing blocks by specialized instructions.

Figure 4.3(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 re-ordered (RAC - reordered assembly code), as shown in Figure 4.3(b). Figure 4.3(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 4.3(c) (NAC - new assembly code), where highlighted new instructions replace the corresponding blocks in Figure 4.3(b).

Thus final instruction set is given in Figure 4.8, as the target instruction set (TIS) of the processor for the example given in Figure 4.3.
4.2. DESIGN METHODOLOGY OF DUAL PIPELINE PROCESSOR

Generation of merged instructions

![Diagram of instruction generation]

```
MOV r3, r7
SUB r3, r3, #12
LDR r3, [r3]

SLDR r3, [r7, #12]
```

```
wire [31:0] source0;
wire [31:0] source1;
wire [31:0] addr;
wire [31:0] result;
wire [3:0] flag;
wire [2:0] rs0;
wire [31:0] current_pc;
wire [15:0] inst;

current_pc = PC.read();
inst = IMAU.read(current_pc);
null = IR.write(inst);
null = PC.inc();

rs0 = "111";
source0 = GPR.read0(rs0);
source1 = EXT8.zero(const);

<addr,flag> = ALU0.sub(source0,source1);

wire addr_err;
<result, addr_err> = DMAU.load(addr);
/* result is 32bit data. already extended*/

null = GPR.write0(rd, result);
```

Figure 4.4: Microcode Details of Specific Instruction Generation

Most of our special instructions were able to use the functional units required by the instruction in the blocks chosen to be merged. An example of this is given in 4.4. Here the (a) indicates a small block of instructions and (b) indicates the special instruction to replace the block in (a). For greater clarification the MicroOperations of the individual instructions in the chosen block is given in 4.5, 4.6, 4.7.

Instruction sequences for merging are chosen in such a way that when merged the work load of any pipeline stage is not increased. This considerably reduces the increase in clock period and area. Consider the mov, sub and ldr instructions, when they are merged each
of the pipeline stages will execute the standard operations. The Sldr’s decode stage will read the operands, execution stage will perform the subtraction, memory stage will load the data. An example of Sldr is shown in figure 4.4

Figure 4.5: Microcode Details of MOV
### 4.2.2 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 4.2.
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 4.3(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. This is due to the reason we can use only one $cmp$ in a given Clock Cycle, and also we need to have the $cmp$ in the same pipe as the branch instruction. This is since $cmp$ and branch has to be consecutive instructions with data dependency. It is efficient to use them in one pipe and use same controller to execute both instructions. 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 since they decide the next instruction to be fetched and need control on instruction fetch. The target instruction set uses condition evaluation and branching as two different instructions, we need to have both of them in same pipe to facilitate the controller designing to be compact. The condition evaluation has to be done in a clock cycle prior to the branch instruction. By assigning them to the same pipe we need not have to modify the VHDL to read the flag register of the other pipe to check for the evaluated
condition. Modifying VHDL code and creating extra wires for connection will increase logic thus area and also may increase the clock cycle.

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 Chapter.

**Definition 1:** *Dependency Graph, G.* The graphical representation 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 4.10(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, \( \Psi \).* The set of connected graphs that contain nodes of a given type. For an instruction, \( \text{Ins} \) in the instruction set, its Associated Graph Set is denoted by \( \Psi_i \).

**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 4.9 shows an example. The graph represents a trace of 10 instructions with the following instruction set: \( \text{Ins}1, \text{Ins}2, \text{Ins}3 \) and \( \text{Ins}4 \). Nodes that represent same type of instructions are shaded similarly for clarity. There are two connected sub-graphs: \( g1 \) and \( g2 \). For \( \text{Ins}1 \), its Associated Graph Set, \( \Psi_1 = \{g1,g2\} \). For \( \text{Ins}3 \), its Associated Graph
set, \( \Psi_3 = \{ g_1 \} \). For \( \text{Ins}_1 \) in sub-graph \( g_1 \), 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 \( g_2 \), in Figure 4.9, 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 \( g_2 \) can be paired with any instruction node in \( g_1 \). But this inter-graph matching may result in longer execution. For example, by grouping instruction nodes 5 and 6, \( g_1 \) and \( g_2 \) 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, \( \sigma \). The potential for an instruction to be executed in parallel with another instruction of the same type. For instruction \( \text{Ins}_i \), it is
defined as
\[ \sigma_i = \frac{1}{N}(LP_i + GP_i), \] (4.1)

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
\[ LP_i = \sum_{g \in \Psi_i} \sum_{j=1}^{d_g} \lfloor \frac{k_j}{2} \rfloor, \] (4.2)

where \( k_j \) is the number of nodes (representing \( Insi \)) of depth \( j \) in graph \( g \), \( |\Psi_i| \) is the number of elements in set \( \Psi_i \); and
\[ GP_i = \left( \frac{\sum_{g \in \Psi_i} \sum_{j=1}^{d_g} k_j \mod 2}{|\Psi_i|} \right) \times \left\lfloor \frac{|\Psi_i|}{2} \right\rfloor, \] (4.3)

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 \( \sigma \) 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, \( \eta \). Given an instruction, \( Insi \), from the target instruction set, assume the cost for the instruction is \( c_i \) and its parallelizability is, \( \sigma_i \), 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 \( \Theta \). An instruction is implemented in both pipes if \( \eta \geq \Theta \). The value \( \Theta \) could be derived in many different ways. We use an average-value based scheme by
using the average value of $\sigma$ 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 $m$,

\[
\bar{c} = \frac{1}{m} \sum_{i=1}^{m} c_i, \quad \bar{\sigma} = \frac{1}{m} \sum_{i=1}^{m} \sigma_i
\]  

(4.4)

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 4.3(c), and the target instruction set shown in Figure 4.8. 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).

<table>
<thead>
<tr>
<th>Ins.</th>
<th>$c$</th>
<th>$\sigma$</th>
<th>$\eta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>0.02</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cmp</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lsl</td>
<td>0.04</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mov</td>
<td>0.004</td>
<td>0.22</td>
<td>55</td>
</tr>
<tr>
<td>mul</td>
<td>0.12</td>
<td>0.11</td>
<td>0.92</td>
</tr>
</tbody>
</table>

Table 4.1: 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 4.10(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 4.10(b) (in bold).

<table>
<thead>
<tr>
<th>IS2</th>
<th>IS1</th>
</tr>
</thead>
<tbody>
<tr>
<td>lsl Rn, Rm, #N</td>
<td>Sldr Rn, [Rm, #N]</td>
</tr>
<tr>
<td>add Rn, Rm, #N</td>
<td>ldr Rn, [Rm]</td>
</tr>
<tr>
<td>cmp Rn, Rm</td>
<td>Sstr Rn, [Rm, #N]</td>
</tr>
<tr>
<td>ble address</td>
<td>mov Rn, Rm</td>
</tr>
<tr>
<td>mov Rn, Rm</td>
<td></td>
</tr>
</tbody>
</table>

(a)

(b)  

1  mov r7, r0
2  mov r6, r1
3  Sldr r3, [r7, #12] -- 1
4  lsl r2, r3, #2 -- 3
5  Sldr r3, [r7, #4] -- 1, 4
6  add r3, r2, r3 -- 4, 5
7  Sldr r2, [r7, #8] -- 1, 6
8  ldr r3, [r3] -- 6
9  cmp r2, r3 -- 7, 8
10 ble .L4 -- 9
11 mul r0, r5 --
12 mul r5, r6 --
13 Sldr r3, [r7, #12] -- 1, 9
14 lsl r2, r3, #2 -- 13
15 Sldr r3, [r7, #4] -- 1, 14
16 add r3, r2, r3 -- 14, 15
17 ldr r3, [r3] -- 16
18 Sstr r3, [r7, #8] -- 1, 17
19 mov r1, r3 -- 17
20 mov r0, r3 -- 17

(c)  

1  mov r7, r0  mov r6, r1
2  NOP  Sldr r3, [r7, #12]
3  lsl r2, r3, #2  Sldr r3, [r7, #4]
4  add r3, r2, r3  Sldr r2, [r7, #8]
5  NOP  ldr r3, [r3]
6  cmp r2, r3  NOP
7  ble .L4  NOP
8  mul r0, r5  Sldr r3, [r7, #12]
9  mul r5, r6  NOP
10 lsl r2, r3, #2  Sldr r3, [r7, #4]
11 add r3, r2, r3  NOP
12 ldr r3, [r3]  Sldr r3, [r7, #8]
13 NOP  Sstr r3, [r7, #8]
14 mov r1, r3  mov r0, r3

Figure 4.10: Two Instruction Sets and Parallel Sequences

The two set instruction generation approach is given as an algorithm and is depicted in Figure 4.11.
4.2. DESIGN METHODOLOGY OF DUAL PIPELINE PROCESSOR

/* 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 Insi ∈ TIS
    if ηi ≤ Θ
        /* One pipe, if instruction is an ALU instruction, it is already assigned to IS2, no further assignment is required. Otherwise, */
        if Insi is not an ALU instruction
            IS2 ≜ Insi;
        endif
    else
        /* Two pipe implementation. If it is an ALU instruction, further assign it to IS1; otherwise, assign it to both sets */
        if Insi is not an ALU instruction
            IS2 ≜ Insi;
        endif
        IS1 ≜ Insi
    endif
endfor
/* Based on the generated IS1 and IS2, create two parallel sequences */
parallel_seq(IS1, IS2)

Figure 4.11: Dual Instruction Set Generation Algorithm

The complexity of the algorithm given in Figure 4.11 is $O(n)$ where $n$ is the number of instructions in a trace of the application.
<table>
<thead>
<tr>
<th>App.</th>
<th>AC Single</th>
<th>AC Parallel</th>
<th>CC Single</th>
<th>CC Parallel</th>
<th>SA Single</th>
<th>SA Parallel</th>
<th>Clk. per Single (ns)</th>
<th>Clk. per Parallel (ns)</th>
<th>AC % Penal.</th>
<th>Perf % Impr.</th>
<th>SA % Red.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PNF</td>
<td>22159</td>
<td>26194</td>
<td>24272230</td>
<td>20547000</td>
<td>3029453359</td>
<td>2777023291</td>
<td>13.417</td>
<td>13.104</td>
<td>18.2</td>
<td>17.3</td>
<td>8.3</td>
</tr>
<tr>
<td>BS</td>
<td>22858</td>
<td>26569</td>
<td>11989268</td>
<td>8218985</td>
<td>1065312576</td>
<td>1014212135</td>
<td>13.888</td>
<td>13.415</td>
<td>16.2</td>
<td>33.8</td>
<td>4.8</td>
</tr>
<tr>
<td>GCD</td>
<td>22165</td>
<td>25779</td>
<td>8222364</td>
<td>7119392</td>
<td>1112552403</td>
<td>1060485171</td>
<td>13.703</td>
<td>12.974</td>
<td>16.3</td>
<td>18.0</td>
<td>4.7</td>
</tr>
<tr>
<td>MM</td>
<td>27711</td>
<td>32180</td>
<td>74272740</td>
<td>51202126</td>
<td>8715574361</td>
<td>7704880016</td>
<td>13.749</td>
<td>13.049</td>
<td>16.1</td>
<td>34.6</td>
<td>11.6</td>
</tr>
<tr>
<td>IS</td>
<td>22458</td>
<td>26269</td>
<td>19123756</td>
<td>14021924</td>
<td>1690419837</td>
<td>1652783482</td>
<td>13.631</td>
<td>11.876</td>
<td>17.0</td>
<td>36.1</td>
<td>2.3</td>
</tr>
<tr>
<td>SS</td>
<td>22858</td>
<td>26569</td>
<td>26152656</td>
<td>19876018</td>
<td>2198453345</td>
<td>2085369586</td>
<td>13.659</td>
<td>13.335</td>
<td>16.2</td>
<td>25.8</td>
<td>5.1</td>
</tr>
</tbody>
</table>

Table 4.2: Simulation Results
4.2.3 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 4.2.

Given the two instruction sets, we generate a two-pipe processor in two steps as illustrated in Figure 4.12.

![Figure 4.12: 2-Pipeline Processor Generation](image)

**Step 1:**

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.

**Step 2:**

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 4.1.

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.
Figure 4.13: 2-pipes vs. 1-pipe

Benchmark:

- PNF
- IS
- MM
- GCD
- BS

Area Cost

Perf. Improv.

Switch. Act.

Reduction
4.3 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 effectiveness 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 4.14. 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 4.14: Synthesis/Simulation System](image)

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 Model-
Sim 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 synthesized 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 4.2.2. 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 4.13.

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.
4.4 Conclusions

In this chapter 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. But these operations are executed in simple hardware modules such as adder instead of an ALU. Thus area is utilized efficiently without redundancy. We see speed improvements of up to 36% and switching activity reductions of up to 11%. These improvements come at the cost of increased area which for benchmarks considered is 16.7% on average.

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.
CHAPTER 5

Application Specific Heterogeneous Multi-Pipeline Processor Design

Introduction

For a given application, it is possible to design an ASIP with customized multiple pipelines. Since the application is well understood, the number of pipelines and each of the individual pipes can be customized.

In this chapter we propose Application Specific Instruction Set Processors with heterogeneous multiple pipelines to efficiently exploit the available parallelism at instruction level. We have developed a design system based on the Thumb processor architecture. Given an application specified in C language, the design system can generate a processor with a number of pipelines specifically suitable to the application and the parallel code associated with the processor. Each pipeline in such a processor is customized, and implements its own special instruction set so that the instructions can be executed in parallel with low hardware overhead.

The design approach for a customization is developed in such a way so that the resulting heterogeneous multi-pipeline processor can be maximally utilized to achieve high perfor-
mance at low cost on area and code size.

Contributions

For the first time we propose to design ASIPs with varying number of pipelines. With such a design strategy, maximal parallelism can be efficiently exploited. In particular:

- we show a novel architecture which tightly couples n-pipes via the register file;
- for the first time we present a method to customize the number of pipelines and instruction sets for each of the pipelines; and
- we developed an implementation system with such a design approach.

To show the efficiency of our approach we study performance, area, code size, and energy consumption to demonstrate the viability of such an approach.

Chapter Organization

The rest of the chapter is organized as follows: section 5.1 describes the architecture template of the multi-pipeline processor to be implemented; while section 5.2 describes the methodology taken to design such a processor. Experimental setup and results are given in section 5.3; and the chapter is concluded in section 5.4.

5.1 Multi-Pipeline Architecture

Our design approach is based upon the Thumb processor instruction set architecture (Thumb ISA [148]), which is simple and small. Figure 5.1 illustrates the general architecture of our ASIP design. It consists of at least two pipelines, which are necessary for primary functions of all applications. These two pipes are known as the primary pipes:
5.1. MULTI-PIPELINE ARCHITECTURE

Pipe 1 and Pipe 2. Each of the two pipes essentially performs different functions (though they can share some instructions). Pipe 1 is specifically designated for program flow control. This pipeline is primarily responsible for fetching instructions from the instruction memory and dispatching them to all other pipelines. When the program branches, Pipe 1 flushes all pipelines. Pipe 2 performs data memory access, transferring data between the register file and the data memory. Pipe 1 contains (at least) an ALU, while Pipe 2 contains (at least) a data memory access unit (DMAU). This structure can be augmented when the instruction sets for the pipelines are enlarged.

Extra pipelines are utilized based on the parallelism exhibited in the application. All pipelines share one register file which is multi-ported, so that all pipelines can access
the register file simultaneously. Note that we utilize multi-port register file structure to connect to multiple pipelines, thus increasing the design area size. For every additional pipe, we need two extra read ports and another write port. However, as will be seen in Section 5.3, the number of pipelines required is usually small. This is due to the limitation of the compiler in the parallelism extraction.

Each pipeline has a separate control unit that controls the operation of the related functional units on that pipe. But the limited controls such as flushing during branching, pc address supply for address computation are allowed to synchronize the architecture. This is due to the nature of static scheduling and in order execution. Forwarding is enabled in all pipelines so that the results from the execution unit can be forwarded within a pipeline and between pipelines.

Based on the architecture and a given application, we:

- determine a suitable number of pipelines and the functional units for each of the pipelines;
- schedule the program instructions into the determined number of pipelines; and
- try to limit the area overhead, total energy consumption and code size.

## 5.2 Design Methodology of Multi-Pipeline Processor

In this section, we first give an overview of our design methodology and then present the algorithms used in the design.

### 5.2.1 Design Approach

The design flow described in this chapter is illustrated in Figure 5.2. It takes as input an application written in C. The program is first compiled into single-pipeline assembly
code based on the Thumb ISA (step 1). Thumb is a high code density subset of ARM ISA. The assembly code can be represented by a basic block graph, an illustration of which is given in Figure 5.3(b). The assembly code represented in the basic block graph is given in Figure 5.3(a).
In the next step (step 2), an initial pipeline number is chosen as the starting searching point of the design space exploration. We start from the minimal 2-pipe structure and the number of pipelines is iteratively increased as the exploration continues.

We use the number of cycles required for a memory access as an input to the scheduling step. If more cycles are required to access memory, then greater number of instructions can be scheduled in parallel with the memory access instruction. Since our design is based on a static schedule in order to synchronize all pipes, during memory access by one, we add no-ops after the memory operation, and in parallel schedule standard operations in the other pipes. Since the hazard caused by load data dependency is taken care of by the scheduler we have not used the hardware stall. This produces a simple scheduler and
5.2. DESIGN METHODOLOGY OF MULTI PIPELINE PROCESSOR

(a) Parallel Program Sequences

(b) Individual Pipeline Instruction Sets

The output of the scheduling step is illustrated in Figure 5.4(a). (More details on the algorithm for step 3 are given in section 5.2.2.)

The original one-pipe program, shown in Figure 5.3(a), is now divided into several sequences (shown in columns in Figure 5.4(a)). Instructions that are scheduled in the same time slot (shown on the same row in the figure) are executed simultaneously on different pipelines. Each of the sequences forms an instruction set for the corresponding pipeline, as illustrated in Figure 5.4(b), where instruction set, ISA $i$ is obtained from program Se-

relatively simple processor design. In addition, this saves clock cycles which otherwise would have been wasted in all pipes waiting for the load.
quence \( i \) (\( \text{Seq. } i \) in the figure). Since we get a subset of the ISA of the target architecture in each pipe we will have pipes with smaller ISA set. Therefore, reduction of area will be achieved in each pipe compared to the datapath will the entire set of ISA.

In step 4 we use ASIPMeister, a single-pipe ASIP design software tool, to create a design for each pipeline. The tool takes as input the instruction set, functional unit specification, and instruction microcode, and produces a VHDL simulation model and a VHDL synthesis model. The functional units are parameterized, and therefore could be used for various architectures with different instruction and data sizes. The function of each instruction is specified in microcode, and therefore we can implement instructions with many different operations and combinations of operations. We can even modify the target basic ISA to improve performance, such as merging instructions. In our system we started with the Thumb ISA (Thumb has multi-cycle PUSH and POP instructions but due to the static parallel scheduling nature of our architecture design we implemented only single-cycle PUSH and POP, and translated these multi-cycle PUSH and POP to single-cycle ones).

We used instruction micro code to implement the instructions and modified assembly code.

A processor generated using ASIPmeister is shown in Figure 5.5(a), where three single-pipe processors have been created. Each processor only performs the instructions specified by its own instruction set. All pipelines are then integrated into a multi-pipeline processor with a parallel structure as shown in Figure 5.5(b), where the register files are consolidated into one shared component with 11 ports (Pipe 1 and Pipe 3 each have two read ports and one write port, while Pipe 2 has three read ports and two write ports due to address data operations); the data forward paths between pipes are established (as illustrated in the dashed connection lines in the figure); the three individual instruction memories are merged into one memory (each word is a concatenation of the three individual instructions issued in a particular clock cycle, and is three times longer in the given example). Instruction fetching is done by Pipe 1 therefore the PC in Pipe 1 is the active PC. Other pipes do not have a PC. The active PC is used by other Pipes to read
current address for any computation. Instruction address bus is connected from Pipe 1 to Instruction memory. And control signals are sent by Pipe 1 to Instruction memory. The instructions are sent in instruction data buses corresponding to each pipe. The control units are modified so that the instruction word is dispatched to all pipelines at once by Pipe 1. If the program branches, Pipe 1 sends a control signal to flush all three pipelines.

The parallel code is generated based on the object code of each of the program sequences, which is obtained from the assembly output of the GCC compiler.

In the last step, the multi-pipe processor model is simulated using ModelSim for functional verification, and is then synthesized with Synopsys Design Compiler. The simulation and synthesis provide the performance, area and power consumption of the design, which are used in the evaluation of the design. The iterative process, formed by steps 3 to 5, is repeated for designs with increased pipelines until no further improvement can be obtained.

The approach used for processor/code generation and evaluation is summarized in Figure 5.6. Note in order to obtain accurate switching activity, a second simulation with the gate-level VHDL model produced by Synopsys Compiler, is performed (as indicated by the dashed line in the figure). However, with the gate-level VHDL model the simulation time increases dramatically with the input data size. Since in our work we are only interested in the comparative overheads, we use a scaled-down input data to speed up the switching activity simulation process.

5.2.2 Exploitation of Instruction Parallelism

If an instruction is executed within a basic block, then all instructions in the block are executed. Therefore, parallelism within blocks is static and can be easily extracted at the initial stage of the design. Our parallel scheduling is performed within each basic block, and as each basic block is quite small, the complexity of scheduling is low. Note that although we allow loop unrolling in our method, due to the large code size it may
Figure 5.5: Example Processor Architecture
cause, this practice is limited only for loops with very high frequency and low parallelism. Another advantage of scheduling within basic blocks is the avoiding of the complicated speculation/prediction issue which would otherwise need to be addressed.
Scheduling Techniques

Given the number of pipelines, our scheduling determines the time slot and the executing pipeline for all instructions such that the resulting processor area and power cost is small while the execution time for the block of instructions is minimal. Our scheduling algorithms are based upon the following assumptions/considerations.

- Power consumption can be measured in terms of dynamic switching activity and leakage power. Leakage power is closely related to chip area, while dynamic power can also be loosely associated to the area, since larger the area, greater the number of logic gates. Without considering any special power control schemes, such as clock gating and power supply gating, we assume that reducing the area will also reduce power consumption. More elaborate power reduction schemes can be employed, but they are beyond the scope of this work. We try to minimize the power by efficiently assigning and utilizing functional units. For example if we use ALU to perform the addition or substraction the area and Leakage power will be high compare to using an adder or substractor. Complex functional units will toggle more switches to perform an operation compare to that with simple circuits. By efficiently selecting instructions and functional units to a particular pipe we can reduce the overall cost.

- The functionality of a pipeline is determined by the instruction set, which is formed during the instruction scheduling phase. Scheduling an instruction to a pipeline may incur area overhead. If the pipeline already has the functional components used by the instruction, then the overhead is nil; otherwise, the overhead is the area of the extra functional component required for the instruction. For example, given a pipeline with an ALU function, scheduling an add instruction to this pipeline does not incur any additional area. Therefore, the overhead of the instruction is 0. If, however, this instruction is scheduled into a pipeline which only consists of a shift operation, then scheduling the add instruction will require at least an adder, and the
area of the added functional unit is the incurred overhead. We call such an overhead as scheduling overhead.

- If an instruction can be executed by one of two pipelines, one that is complex and the other simple, executing the instruction in the complex pipe is assumed to be more costly as more logic gates are exercised, thus consuming more power. Since the complexity is closely related to the area, we use area cost to present the complexity of the pipeline. As such, one of our scheduling strategies is to schedule an instruction to a pipeline with a cheap area cost.

- Scheduling instructions to a pipeline of a lower workload is preferred, so that more instructions can be scheduled at the earliest time slot as possible. We define the potential workload of a pipeline as the percentage of instructions in the basic block which can be executed by the pipe. For example, in a pipe, Pipe i, assume we are able to execute any type of instruction other than memory and branch type instructions. If we have 10 instructions in the basic block, and three of them are memory and branch instructions, the potential workload of the pipe i would be 0.7.

- For a single pipeline, a memory load instruction will cause a pipeline stall. The pipeline is idle until the memory operation is complete. Instead of stalling all pipelines in the processor during a load operation in one pipeline, we allow instructions to be scheduled in other pipes such that the effect of the hazard is limited.

To aid the explanation of the these algorithms, some terms are given here.

- **instruction area cost,** $AC(i)$: the area of the basic functional unit for instruction $i$. For example, for a 16-bit add instruction, its area cost is the chip area required for a 16-bit adder.

- **instruction scheduling overhead,** $SO(i,j)$: the extra area incurred when instruction $i$ is scheduled in pipeline $j$. For example, given a pipeline with an ALU function,
scheduling an *add* instruction to this pipeline, will have an SO value of 0; if, however, this instruction is scheduled into a pipeline which currently contains only a shift operation, then \( SO(\text{add}, j) = AC(\text{add}) \).

- **pipeline load**, \( PL(j) \): the percentage of instructions in the basic block which can be executed by pipe \( j \). For example, in a pipe, Pipe \( i \), assume we are able to execute any type of instruction other than memory and branch type instructions. If we have 10 instructions in the basic block, and three of them are memory and branch instructions, \( PL(i) \) would be 0.7.

- **instruction scheduling time slot**, \( T(i) \): the execution time slot in which instruction \( i \) is scheduled.

- **instruction earliest time slot**, \( ET(i) \): the earliest time slot in which instruction \( i \) can be scheduled within the basic block. It is determined by the data dependency of the instructions and memory access time delay cycles. Assume instruction \( i \) is dependent on instruction \( j \) and \( j \) is scheduled to a time slot \( k \). Assume the the number of clock cycles to memory access is \( x \). If \( j \) is a loading instruction, then \( ET(i) = k + x \); otherwise, \( ET(i) = k + 1 \).

- **design area efficiency**, \( \eta \): the maximum possible execution frequency of an application per area unit. It is defined as

\[
\eta = 1/(T \times A),
\]

where \( T \) is the execution time of the application and \( A \) the area of the processor that executes the application. We use *eta* to evaluate the design improvement of a multi-pipe ASIP. Large \( \eta \) means high performance with small area cost.

We use **design area efficiency**, \( \eta \), to evaluate the design quality.
5.2. DESIGN METHODOLOGY OF MULTI-PIPELINE PROCESSOR

Basic Block Scheduling Algorithm

Our scheduling method is presented in a bottom-up manner, where the pipeline selection for an instruction (Algorithm 1) is presented first, followed by the basic block instruction scheduling procedure (Algorithm 2).

In Algorithm 1, we find a suitable pipeline for an instruction. Three parameters: scheduling overhead, pipeline area cost, and pipeline work load, are used here to guide the pipeline allocation for instructions. As can be seen from the algorithm, scheduling overhead takes the highest priority among the three parameters, with the workload coming the second and area cost the last. This is since adding a new functional unit will certainly increase the area therefore scheduling overhead is reduced to reduce overall area.

Algorithm 1 Instruction pipeline selection: $\text{PipeSelection}(i,P):$

- **step 1:** find the pipe where the scheduling overhead is minimal if the instruction is placed in that pipe;
- **step 2:** if more than one pipe is found in the previous step, find the pipe with minimal load (i.e., one with fewer instructions);
- **step 3:** If more than one pipe is found in the previous step, find the pipe with the smallest area;
- **step 4:** return the found pipe;

With the above pipeline selection algorithm, the basic block scheduling method is given in Algorithm 2. The algorithm takes a basic block and schedules its instructions to a set of pipes, $P$. We use an array, $\text{Sched}[\text{timeslot}][\text{pipeline}]$, to represent the scheduling result. The notation, $\text{Sched}[i][j] = k$, means instruction $k$ is scheduled to time slot $i$ on pipe $j$.

We demonstrate the scheduling algorithm with a basic block as shown in Figure 5.7(a). The block contains nine instructions. Assume the instructions are to be scheduled into 3 pipes: Pipe 1, Pipe 2 and Pipe 3. Pipe 1 contains an ALU and can perform all but memory access instructions. The memory access instructions are exclusively performed by Pipe 2. The functions in Pipe 3 are initially undefined but can be any functions (except memory and branch type), as requested by the scheduling algorithm.
Figure 5.7: Basic Block Scheduling
Algorithm 2 Basic Block Scheduling: \textit{blockScheduling}(B, P)

// Initialize array, \textit{Sched}, with Nop instructions
\textbf{Initialize}(\textit{Sched});

// schedule instructions in Block, B, to a set of pipes, P.
\textbf{for all} \(i \in B\) (in program sequence) \textbf{do}
  \textbf{scheduling\_done}(i) = FALSE;
  \textbf{while} \textbf{scheduling\_done}(i) is FALSE \textbf{do}
    find the earliest time slot, \(t\), for instruction \(i\);
    find all available pipes, \(P_{avail}\), at \(t\);
    find a suitable pipe, \(p\), for instruction \(i\) using Algo. 1;
    \textbf{if} \(p\) exists \textbf{then}
      \(\text{Sched}[t][p] = i;\)
      \textbf{scheduling\_done}(i) = TRUE;
    \textbf{else}
      \textit{try scheduling the instruction to the next time slot}
      \(t++;\)
    \textbf{end if}
  \textbf{end while}
\textbf{end for}

<table>
<thead>
<tr>
<th>Instr.</th>
<th>step 1 (t)</th>
<th>step 2 (P_{avail})</th>
<th>step 3 (p)</th>
<th>step 4 (\text{Sched}[t][p])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: push</td>
<td>1</td>
<td>1, 2, 3</td>
<td>2</td>
<td>\text{sched}[1][2]=1</td>
</tr>
<tr>
<td>2: mov</td>
<td>1</td>
<td>1, 3</td>
<td>1</td>
<td>\text{sched}[1][1]=2</td>
</tr>
<tr>
<td>3: lsl</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>\text{sched}[1][3]=3</td>
</tr>
<tr>
<td>4: and</td>
<td>2</td>
<td>1, 2, 3</td>
<td>1</td>
<td>\text{sched}[2][1]=4</td>
</tr>
<tr>
<td>5: add</td>
<td>2</td>
<td>2, 3</td>
<td>3</td>
<td>\text{sched}[2][3]=5</td>
</tr>
<tr>
<td>6: ldr</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>\text{sched}[2][2]=6</td>
</tr>
<tr>
<td>7: add</td>
<td>4</td>
<td>1, 2, 3</td>
<td>3</td>
<td>\text{sched}[4][3]=7</td>
</tr>
<tr>
<td>8: cmp</td>
<td>3</td>
<td>1, 2, 3</td>
<td>1</td>
<td>\text{sched}[3][1]=8</td>
</tr>
<tr>
<td>9: beq</td>
<td>4</td>
<td>1, 2</td>
<td>1</td>
<td>\text{sched}[4][1]=9</td>
</tr>
</tbody>
</table>

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
Instr. & step 1 \(t\) & step 2 \(P_{avail}\) & step 3 \(p\) & step 4 \(\text{Sched}[t][p]\) \\
\hline
1: push & 1 & 1, 2, 3 & 2 & \text{sched}[1][2]=1 \\
2: mov  & 1 & 1, 3 & 1 & \text{sched}[1][1]=2 \\
3: lsl  & 1 & 3 & 3 & \text{sched}[1][3]=3 \\
4: and  & 2 & 1, 2, 3 & 1 & \text{sched}[2][1]=4 \\
5: add  & 2 & 2, 3 & 3 & \text{sched}[2][3]=5 \\
6: ldr  & 2 & 2 & 2 & \text{sched}[2][2]=6 \\
7: add  & 4 & 1, 2, 3 & 3 & \text{sched}[4][3]=7 \\
8: cmp  & 3 & 1, 2, 3 & 1 & \text{sched}[3][1]=8 \\
9: beq  & 4 & 1, 2 & 1 & \text{sched}[4][1]=9 \\
\hline
\end{tabular}
\caption{Example of Basic Block Scheduling}
\end{table}

The scheduling starts with the first instruction, \textit{push}. since it is not dependent on any other instruction, its earliest scheduling time slot is 1 (i.e. \(t=1\)). At this moment, all three pipes are available. Amongst them, Pipe 2 is the most suitable pipe (the only pipe with 0 \textit{scheduling overhead}). Next instruction is \textit{mov}, and it can be scheduled in the first time slot. Among the two available pipes: Pipe 1 and Pipe 3, Pipe 1 is selected (since Pipe 1’s \textit{scheduling overhead} is 0). The following non-dependent instruction \textit{lsl} is then scheduled to the last pipe: Pipe 3, with the \textit{scheduling overhead} of a Shifter. The next instruction \textit{and} is dependent on the second instruction, and therefore its earliest scheduling time slot is 2.
Algorithm 3 Multi-pipe Processor Design:

```java
//best design and its related design efficiency are initialized.
best_design = NULL;
η_{best} = 0;
//the design iteration starts from 2 pipe structure.
N_p = 2;
design_done = FALSE;
while design_done is FALSE do
    for all \( B \in G \) do
        blockScheduling(\( B, N_p \));
    end for
    processor_generation();
    processor_simulation();
    η = calculate_design_efficiency();
    //if design is improved
    if η > η_{best} then
        best_design = current_design;
        η = η_{best};
        N_p++;
    else
        design_done = TRUE;
    end if
end while
output best_design;
```

With zero scheduling overhead to Pipe 1, it is assigned to the second time slot in Pipe 1. The remaining two pipes are available to the following add instruction. Both pipes do not have functional unit for add instruction. The related scheduling overheads are therefore same. Since the potential load (PL) for pipe 2 (7/9) is greater than pipe 3 (4/9), pipe 3 is selected for the add instruction, which leaves pipe 2 for the next instruction ldr to be scheduled in the time slot. This process is repeated for the rest of the instructions.

The intermediate results of the scheduling is given in Table 5.1. The final schedule is shown in Figure 5.7(b), where slots with 0’s represent Nop operations.

**Design Algorithm**

Based on the above basic block scheduling algorithm, an application program can be scheduled into multiple pipes. The overall design is summarized in Algorithm 3. Given
5.3 SIMULATIONS AND RESULTS

the basic block graph, $G$ for an application, the algorithm gives an efficient design with a suitable number of pipelines and special instruction sets for each of the pipelines. Each design iteration is evaluated with performance/area ratio, $\eta$. The design loops stops when new design cannot bring further improvement.

5.3 Simulations and Results

With the above methodology we designed multi-pipeline processors for a set of applications mainly from Mibench [66]. The applications we implemented are: Adpcm decode, Adpcm encode, Basic math, Bitcount, Crc32, quick sort, Sha, string search, endian, bin-search, cstrings, and dijkstra. These benchmarks represent a variety of application fields such as network, security, telecommunication and automotive, which are frequently encountered in embedded systems.

As described in Section 5.2, our base instruction set architecture was based on the Arm-Thumb processor. We generated VHDL models and the associated executable code for the multi-pipeline processor for each of the applications, with memory access latencies. The designs were then synthesized using Synopsys Design Compiler based on the TSMC 90nm core library, and simulated with the ModelSim simulator.

Performance is evaluated with the processor clock speed, which is given by Design Compiler, and the clock cycles given by ModelSim. Power consumption is divided into dynamic power and leakage power consumptions. The leakage power is estimated by the Synopsys Design Compiler, along with that the Design Compiler produces a gate-level model of the design. This gate-level model is simulated using ModelSim to calculate the switching activity of the nets, which is used by the Synopsys Design Compiler (Prime-Power) to estimate the dynamic power.

Table 5.2 records the performance, area, power and code size of designs for all applications (application name is listed in the first row of the table. To save the space, the
## Table 5.2: Synthesis and Simulation Results for 1-3 Pipe Designs

<table>
<thead>
<tr>
<th>BenchMarks</th>
<th>adp.dec</th>
<th>adp.enc</th>
<th>bas.math</th>
<th>b.count</th>
<th>crc32</th>
<th>q.sort</th>
<th>sha.encr.</th>
<th>str.search</th>
<th>endian</th>
<th>b.search</th>
<th>estr.</th>
<th>dijkstra</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Clock Period (ns)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Pipe</td>
<td>4.43</td>
<td>4.43</td>
<td>4.55</td>
<td>4.58</td>
<td>4.39</td>
<td>4.52</td>
<td>4.44</td>
<td>4.58</td>
<td>4.35</td>
<td>4.63</td>
<td>4.56</td>
<td>4.58</td>
</tr>
<tr>
<td>3 Pipe</td>
<td>3.55</td>
<td>3.55</td>
<td>3.61</td>
<td>3.18</td>
<td>N/A</td>
<td>3.62</td>
<td>3.13</td>
<td>2.97</td>
<td>3.19</td>
<td>N/A</td>
<td>3.63</td>
<td>3.17</td>
</tr>
<tr>
<td><strong>Exec. Time (ms)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Pipe</td>
<td>316</td>
<td>382</td>
<td>142</td>
<td>266</td>
<td>45</td>
<td>202</td>
<td>108</td>
<td>51</td>
<td>46</td>
<td>169</td>
<td>59</td>
<td>88</td>
</tr>
<tr>
<td>2 Pipe</td>
<td>238.0</td>
<td>74.6</td>
<td>135.0</td>
<td>25.2</td>
<td>123.0</td>
<td>58.8</td>
<td>26.0</td>
<td>25.8</td>
<td>99</td>
<td>39.4</td>
<td>48.0</td>
<td></td>
</tr>
<tr>
<td>3 Pipe</td>
<td>72.7</td>
<td>254.9</td>
<td>81.8</td>
<td>137.5</td>
<td>N/A</td>
<td>119.6</td>
<td>57.3</td>
<td>24.7</td>
<td>24.9</td>
<td>N/A</td>
<td>38.9</td>
<td>48.4</td>
</tr>
<tr>
<td><strong>Area (cells)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Pipe</td>
<td>89592</td>
<td>88592</td>
<td>82752</td>
<td>80740</td>
<td>85875</td>
<td>93542</td>
<td>88634</td>
<td>80746</td>
<td>86808</td>
<td>91193</td>
<td>102610</td>
<td>82955</td>
</tr>
<tr>
<td>2 Pipe</td>
<td>123431</td>
<td>119791</td>
<td>129552</td>
<td>137167</td>
<td>118521</td>
<td>151767</td>
<td>128727</td>
<td>126286</td>
<td>138197</td>
<td>138978</td>
<td>123087</td>
<td>131284</td>
</tr>
<tr>
<td>3 Pipe</td>
<td>215498</td>
<td>207225</td>
<td>192375</td>
<td>183154</td>
<td>N/A</td>
<td>203326</td>
<td>202045</td>
<td>163223</td>
<td>186590</td>
<td>N/A</td>
<td>206305</td>
<td>181025</td>
</tr>
<tr>
<td><strong>Leakage Power (µW)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Pipe</td>
<td>151</td>
<td>148</td>
<td>135</td>
<td>139</td>
<td>145</td>
<td>160</td>
<td>148</td>
<td>135</td>
<td>147</td>
<td>156</td>
<td>175</td>
<td>139</td>
</tr>
<tr>
<td>2 Pipe</td>
<td>210</td>
<td>206</td>
<td>218</td>
<td>230</td>
<td>204</td>
<td>258</td>
<td>217</td>
<td>213</td>
<td>237</td>
<td>219</td>
<td>225</td>
<td>228</td>
</tr>
<tr>
<td>3 Pipe</td>
<td>371</td>
<td>363</td>
<td>334</td>
<td>316</td>
<td>N/A</td>
<td>359</td>
<td>354</td>
<td>283</td>
<td>327</td>
<td>N/A</td>
<td>359</td>
<td>320</td>
</tr>
<tr>
<td><strong>Switch. Activ. (1000's)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Pipe</td>
<td>12515</td>
<td>32497</td>
<td>2702</td>
<td>1493</td>
<td>30265</td>
<td>10843</td>
<td>110828</td>
<td>142902</td>
<td>40799</td>
<td>515</td>
<td>46483</td>
<td>59454</td>
</tr>
<tr>
<td>2 Pipe</td>
<td>15464</td>
<td>34796</td>
<td>2937</td>
<td>1948</td>
<td>30809</td>
<td>14850</td>
<td>141540</td>
<td>167855</td>
<td>46189</td>
<td>5692</td>
<td>57786</td>
<td>59989</td>
</tr>
<tr>
<td>3 Pipe</td>
<td>17709</td>
<td>37529</td>
<td>3019</td>
<td>2085</td>
<td>N/A</td>
<td>1939</td>
<td>162274</td>
<td>175771</td>
<td>46444</td>
<td>N/A</td>
<td>69357</td>
<td>62651</td>
</tr>
<tr>
<td><strong>Code Size (bytes)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Pipe</td>
<td>726</td>
<td>406</td>
<td>254</td>
<td>520</td>
<td>132</td>
<td>1774</td>
<td>1250</td>
<td>502</td>
<td>410</td>
<td>258</td>
<td>286</td>
<td>608</td>
</tr>
<tr>
<td>2 Pipe</td>
<td>1208</td>
<td>652</td>
<td>404</td>
<td>900</td>
<td>228</td>
<td>3020</td>
<td>1816</td>
<td>928</td>
<td>696</td>
<td>480</td>
<td>484</td>
<td>1044</td>
</tr>
<tr>
<td>3 Pipe</td>
<td>1626</td>
<td>960</td>
<td>564</td>
<td>1350</td>
<td>N/A</td>
<td>4452</td>
<td>2622</td>
<td>1374</td>
<td>1002</td>
<td>N/A</td>
<td>690</td>
<td>1554</td>
</tr>
<tr>
<td>BenchMarks</td>
<td>adp.dec</td>
<td>adp.enc</td>
<td>bas.math</td>
<td>bcount</td>
<td>crc32</td>
<td>q.sort</td>
<td>sha.encr.</td>
<td>str.search</td>
<td>endian</td>
<td>b.search</td>
<td>cstr.</td>
<td>dijkstra</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>---------</td>
<td>----------</td>
<td>--------</td>
<td>-------</td>
<td>--------</td>
<td>----------</td>
<td>-----------</td>
<td>--------</td>
<td>---------</td>
<td>-------</td>
<td>----------</td>
</tr>
<tr>
<td>Performance %</td>
<td>2 Pipe</td>
<td>69.1</td>
<td>60.5</td>
<td>90.9</td>
<td>97.0</td>
<td>79.2</td>
<td>63.8</td>
<td>84.5</td>
<td>97.1</td>
<td>78.4</td>
<td>69.5</td>
<td>50.5</td>
</tr>
<tr>
<td></td>
<td>3 Pipe</td>
<td>60.0</td>
<td>49.9</td>
<td>74.1</td>
<td>93.6</td>
<td>0.0</td>
<td>69.0</td>
<td>89.2</td>
<td>107.9</td>
<td>85.2</td>
<td>N/A</td>
<td>52.6</td>
</tr>
<tr>
<td>Area %</td>
<td>2 Pipe</td>
<td>37.8</td>
<td>35.2</td>
<td>56.6</td>
<td>69.9</td>
<td>38.0</td>
<td>62.2</td>
<td>45.2</td>
<td>56.4</td>
<td>59.2</td>
<td>52.4</td>
<td>20.0</td>
</tr>
<tr>
<td></td>
<td>3 Pipe</td>
<td>140.5</td>
<td>134.0</td>
<td>132.5</td>
<td>126.8</td>
<td>N/A</td>
<td>117.4</td>
<td>128.0</td>
<td>102.1</td>
<td>114.9</td>
<td>N/A</td>
<td>101.1</td>
</tr>
<tr>
<td>Leak. Pow. %</td>
<td>2 Pipe</td>
<td>39.1</td>
<td>39.2</td>
<td>61.5</td>
<td>65.5</td>
<td>40.7</td>
<td>61.3</td>
<td>46.6</td>
<td>57.8</td>
<td>61.2</td>
<td>53.2</td>
<td>28.6</td>
</tr>
<tr>
<td></td>
<td>3 Pipe</td>
<td>145.7</td>
<td>145.3</td>
<td>147.4</td>
<td>127.3</td>
<td>N/A</td>
<td>124.4</td>
<td>139.2</td>
<td>109.6</td>
<td>114.9</td>
<td>N/A</td>
<td>105.1</td>
</tr>
<tr>
<td>Switch. Activ. %</td>
<td>2 Pipe</td>
<td>23.6</td>
<td>7.1</td>
<td>8.7</td>
<td>30.5</td>
<td>1.8</td>
<td>37.0</td>
<td>27.7</td>
<td>17.5</td>
<td>13.2</td>
<td>10.3</td>
<td>24.3</td>
</tr>
<tr>
<td></td>
<td>3 Pipe</td>
<td>41.5</td>
<td>15.5</td>
<td>11.7</td>
<td>39.7</td>
<td>N/A</td>
<td>47.0</td>
<td>46.4</td>
<td>23.0</td>
<td>13.8</td>
<td>N/A</td>
<td>49.2</td>
</tr>
<tr>
<td>Code Size %</td>
<td>2 Pipe</td>
<td>66.4</td>
<td>60.6</td>
<td>59.1</td>
<td>73.1</td>
<td>72.7</td>
<td>70.2</td>
<td>45.3</td>
<td>84.9</td>
<td>69.8</td>
<td>86.0</td>
<td>69.2</td>
</tr>
<tr>
<td></td>
<td>3 Pipe</td>
<td>124.0</td>
<td>136.5</td>
<td>122.0</td>
<td>159.6</td>
<td>N/A</td>
<td>151.0</td>
<td>109.8</td>
<td>173.7</td>
<td>144.4</td>
<td>N/A</td>
<td>141.3</td>
</tr>
<tr>
<td></td>
<td>mem latency</td>
<td>perf. metrics</td>
<td>adp dec</td>
<td>adp enc</td>
<td>bas.math</td>
<td>b.count</td>
<td>crs 32</td>
<td>q.sort</td>
<td>sha.encr</td>
<td>str.search</td>
<td>endian</td>
<td>b.search</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>---------------</td>
<td>---------</td>
<td>---------</td>
<td>-----------</td>
<td>---------</td>
<td>--------</td>
<td>--------</td>
<td>----------</td>
<td>------------</td>
<td>--------</td>
<td>---------</td>
</tr>
<tr>
<td>1-pipe</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1CC</td>
<td></td>
<td>CC ('000s)</td>
<td></td>
<td></td>
<td>2624</td>
<td>116</td>
<td>202</td>
<td>1029</td>
<td>1029</td>
<td>4470</td>
<td>1018</td>
<td>2441</td>
</tr>
<tr>
<td>Exec. Time (ms)</td>
<td>10000</td>
<td>6266</td>
<td>142</td>
<td>266</td>
<td>45</td>
<td>262</td>
<td>108</td>
<td>51</td>
<td>46</td>
<td>169</td>
<td>59</td>
<td>88</td>
</tr>
<tr>
<td>2CC</td>
<td></td>
<td>CC ('000s)</td>
<td></td>
<td></td>
<td>2856</td>
<td>127</td>
<td>240</td>
<td>1403</td>
<td>5299</td>
<td>3050</td>
<td>1214</td>
<td>1420</td>
</tr>
<tr>
<td>Exec. Time (ms)</td>
<td>10000</td>
<td>9401</td>
<td>148</td>
<td>308</td>
<td>62</td>
<td>1403</td>
<td>135</td>
<td>55</td>
<td>61</td>
<td>203</td>
<td>73</td>
<td>107</td>
</tr>
<tr>
<td>3CC</td>
<td></td>
<td>CC ('000s)</td>
<td></td>
<td></td>
<td>3057</td>
<td>135</td>
<td>6221</td>
<td>7628</td>
<td>1710</td>
<td>5299</td>
<td>1423</td>
<td>1780</td>
</tr>
<tr>
<td>Exec. Time (ms)</td>
<td>10000</td>
<td>10177</td>
<td>154</td>
<td>349</td>
<td>75</td>
<td>7628</td>
<td>163</td>
<td>65</td>
<td>77</td>
<td>237</td>
<td>87</td>
<td>126</td>
</tr>
</tbody>
</table>

|          |             |               |         |         |           |         |        |        |          |            |        |         |      |          |
| 2 Pipes  |             |               |         |         |           |         |        |        |          |            |        |         |      |          |
| 1CC      |             | CC ('000s)    |         |         | 2083      | 68      | 3399   | 810    | 3399     | 1830       | 832    | 810     | 3085 | 1100     |
| Exec. Time (ms) | 10000     | 7215     | 74      | 135     | 25        | 810    | 58     | 26     | 99       | 39        | 48    | 99      |      | 48       |
| 2CC      |             | CC ('000s)    |         |         | 2332      | 77      | 3826   | 1278   | 3826     | 2135       | 892    | 1120    | 3744 | 1275     |
| Exec. Time (ms) | 10000     | 7884     | 74      | 163     | 39        | 1278   | 68     | 27     | 121      | 45        | 48    | 121     |      | 48       |
| 3CC      |             | CC ('000s)    |         |         | 2519      | 83      | 4641   | 1590   | 4641     | 2441       | 952    | 1415    | 4401 | 1453     |
| Exec. Time (ms) | 10000     | 8607     | 78      | 194     | 49        | 1590   | 78     | 30     | 45       | 142       | 52    | 71      |      | 71       |
| improvement |             |               |         |         |           |         |        |        |          |            |        |         |      |          |
| 1CC      |             | Exec. Time    |         |         | 69        | 64      | 97     | 79     | 64       | 84        | 97     | 78      | 51   | 83       |
| (%)      |             |               |         |         | 60        | 60      | 96     | 55     | 72       | 97        | 99     | 73      | 68   | 60       |
| 2CC      |             | Exec. Time    |         |         | 64        | 64      | 91     | 87     | 67       | 108       | 119    | 72      | 67   | 67       |
| (%)      |             |               |         |         | 60        | 60      | 91     | 87     | 67       | 108       | 119    | 72      | 67   | 67       |
| 3CC      |             | Exec. Time    |         |         | 63        | 63      | 91     | 87     | 67       | 108       | 119    | 72      | 67   | 67       |
| (%)      |             |               |         |         | 60        | 60      | 91     | 87     | 67       | 108       | 119    | 72      | 67   | 67       |

Table 5.4: 2-pipe Designs with Different Memory Access Time
Figure 5.8: Clock Period (ns)
Figure 5.9: Execution Time (ms)
Figure 5.10: Area (cells)
Figure 5.11: Performance/Area (1/(second \times million cells))
Figure 5.12: Leakage Power ($\mu$W)
Figure 5.13: Switching Activity
5.3. SIMULATIONS AND RESULTS

Figure 5.14: Code Size (bytes)
names are abbreviated. For example adpcm stands for adpcm encoder) with 1 clock cycle memory access latency. In order to evaluate the performance improvement and related incidental overheads, we also implemented the Thumb ISA 1-pipe processor for each of the applications. Performance data is given under the categories of clock period in nanoseconds (rows 2-4) and execution time in milliseconds (rows 5-7). Area figures in cells are given in rows 8-10. Leakage power in microWatts and switching activity (scaled down) in counts are grouped in rows 11-13 and rows 14-16. The last group gives the code sizes (in bytes) for the instruction memory for each design. Note for application crc32 and binsearch (under the name b.search in the table), there is no 3-pipe design due to their low parallelism and intensive memory access nature.

For the convenience of observation, some data in the table is plotted for: clock period (Figure 5.8), area (Figure 5.10), leakage power (Figure 5.12) and design efficiency (Figure 5.11, switching activity (Figure 5.13), and code size (Figure 5.14) execution time (Figure 5.9),

The percentage improvement of performance and related overheads in area, power and code size are summarized in Table 5.3.

As can be seen from Figure 5.8, 1-pipe processors demonstrate higher clock period than multiple-pipe processors. It is because the 1-pipe processors have a large control unit that needs to control the execution of all instructions in the instruction set while in multiple-pipe processors, the control unit for each pipeline only implements a small subset of instructions, resulting in short critical paths.

When the design changes from 1-pipe to 2-pipe, significant performance improvements can be obtained. In contrast, there is little or no performance gain when going from 2-pipe designs to 3-pipe designs, but the design overheads become a heavy burden, as illustrated in Figure 5.11, where 2-pipe processors seem to give the best designs, with high execution capacity per million cells for all of the tested applications. This is because the average instruction parallelism for the basic blocks is below 3. Unrolling loops would
have improved parallelism, but unrolling of loops was not considered in the experiments.

It is worth noting that leakage power closely follows the area cost as can be seen from Figures 5.10 and 5.12, where both figures show a near identical trend. Even though this similarity is not obvious in switching activity, a trend can be seen which indicates a relationship with area and switching power.

It also can be seen from the table, the code size increases as the number of pipelines increases because it is unlikely that all pipelines can be fully utilized during application execution. Although the pipes are reducing the total Clockcycles thus execution time, the pipelines are not executing a valid instruction during each Clock Cycle. Therefore for a given Clock Cycle some pipes may be idle by executing a No-op instruction. As we are aware as each pipe is added instruction width is increased by 2 bytes. Unless the code length is reduced by scheduling instructions into each pipe and fully utilize them the total code size in terms of bytes will be increasing. We can say the increase in code size increment factor is inversely proportional to utilization of the pipes.

Since memory is typically slower than the processor, we examined the effect of slower memory on performance in terms of both clock cycles and execution time for the 2-pipe case, along with the 1-pipe designs for comparison. The clock cycles and execution time for each of the applications (columns 4-15) under different memory access latencies (ranging from 1 clock cycle to 3 clock cycles) for 1-pipe and 2-pipe processors are tabulated in Table 5.4, where the rows with the metrics labeled as CC(’000s) give the clock cycles (in thousand) taken by each of the application programs, while rows with the label Exec. Time (ms) provide the execution times. The performance improvement of the 2-pipe designs over 1-pipe designs with different memory access latencies are also given in the table and it is graphically represented in Figure 5.15. As can be seen from Figure 5.15, there is little difference in performance improvement between different memory access latency schemes. Longer memory latency rarely affects the performance improvement. This is due to the efficiency of our parallel scheduling algorithm. As we already mentioned the system is not stalled for memory access instead all pipes will execute in-
Figure 5.15: Performance of Varying Memory Access Latency
5.3. SIMULATIONS AND RESULTS

Instructions in parallel to memory access pipe. Irrespective of the number of Clock Cycles spent on memory access, scheduler will schedule valid instructions for the memory access period in all other pipes but memory access pipe will be assigned No-ops. Therefore, comparable performance can be achieved in our multi-pipe processor containing a slower memory (thus lower cost) as with the single-pipe processor containing a faster memory.

Implications of the Architecture

The number of ports in the register file has to be increased as the number of pipes increases. This increase will contribute to the increase of clock cycle due to the complex routing and higher MUX logic at the input. The area will also increase due to the drive capacity needed at the output. There will be more power dissipation due to the additional logic. If we get a memory intensive application we may need more than one pipeline for memory access. This will require memory with multi-ports and additional logic in the processor which will also contribute to longer clock cycle, higher area, and higher power consumption.

Limitations of the Compiler

If we are to evaluate the performance and other metrics of an embedded system we need to exploit the maximum parallelism of a given program. We need a proper compiler to achieve this. The compiler should be re-targetable or at least it should be able to produce code for a parallel architecture. But in our case we used the gcc cross compiler for arm ISA which generates code for a single pipeline processor. Therefore the code generated had the minimum parallelism. That limited parallelism was a bottle-neck of the work performed. We could not utilize the proposed architecture to its maximum possible extent to study the benefits and drawbacks of the architecture. From our research so far we have learnt that a versatile compiler will enhance the performance of the proposed architecture and also it will open up ventures for further research.
5.4 Conclusions

We presented an approach to customize a multiple pipe processor. The approach relates application instruction level parallelism with the multiple pipeline architecture. An effective parallel instruction scheduling algorithm is used to determine the number of pipelines and the instruction sets to be implemented by each of the pipelines such that the high performance improvement can be achieved with small area overhead. The performance improvement is achieved by specific instructions, improved pipeline (with forwarding circuit) structure, and parallel instructions executing on the multiple pipelines. The small area overhead is retained by utilizing a distributed controller, minimized instruction set overlap between pipelines, and appropriate number of pipelines.

Our designs for a given set of benchmarks show, that on average 77% performance improvement can be achieved with some overheads: 49% on area; 51% on power; 17% on switching activity; and 69% on code size. The parallel scheduling algorithm proposed in this chapter can also efficiently utilize the memory access latency so that the effect of slow memory on the overall execution performance is reduced, with average standard deviation below 6% in our simulation experiments.

Through our experiments, we also have following findings:

• Based on our design methodology, 2-pipe ASIPs are better than single pipeline ASIP in terms of the performance/area ratio.

• All applications we experimented in Mibench possess similar ILP, therefore 2-pipe ASIP is the optimal ASIP structure for all benchmarks examined.

• The negative effect of memory latency on performance can be reduced in multiple-pipeline processors.

The low ILP is due to the fact that we use basic block scheduling. As mentioned in [154] the ILP of the basic block is around 2. According to [126] the average ILP of instruction of an application is 2.11.
Lastly, our work in this chapter has identified a number of interesting topics for future work. Applications in a benchmark suite probably possess a certain common execution behavior, which could result in a general pipeline structure for all applications in the suite. Unrolling loops will increase the instruction level parallelism and enable our design approach to achieve even higher performance gains.

Our simulations and experiments with a group of benchmarks, largely from Mibench suite, show that on average, 77% performance improvement can be achieved compared to a single pipeline ASIP, with the overheads of 49% on area, 51% on power, 17% on switching activity, and 69% on code size.
CHAPTER 6

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

Introduction

Small area and code size are two critical design issues in most of embedded system designs. In this chapter, 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, without any cost of performance.

Our experiments, on a set of benchmarks with 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 improves by 4%.

For pipelined processors, data hazards will stall the pipeline execution and degrade the performance. To reduce hazards, forwarding is used. Figure 6.1(a) shows a forwarding network in a processor of 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 6.1(b), where the forwarding paths are reduced to five, from eight.

Furthermore, the instruction set for each pipe is different. Therefore, instead of using identical instruction widths for instruction encoding, each pipeline can have its own encoding scheme with differing instruction width (as illustrated in Figure 6.1(c)), thus, high density code can be achieved, saving instruction memory.

**Contributions**

The work presented in this chapter 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;
Figure 6.1: Forwarding and Instruction Encoding Customization
Our extensive study has shown the proposed customizations increase the design area and power efficiency significantly without degrading performance.

**Chapter Organization**

The rest of the chapter is organized as follows: section 6 presents an overview of our target multiple pipeline ASIP design; section 6.1 describes the method taken to customize forwarding for such a processor; in section 6.2, we explain how instruction encoding for each pipeline is carried out; experimental results are given in section 6.3; and the chapter is concluded in section 6.4.

**Multi-Pipe ASIP Design Overview**

The design flow for our target multi-pipe ASIPs is given in Figure 6.2. The design flow starts with an application written in C and is compiled into a single-pipeline assembly code using GCC cross compiler for arm, as illustrated in Figure 6.3(a). The single
pipeline code is then scheduled into a number of parallel pipelines, as shown in Figure 6.3(b), which gives the instruction set for each individual pipeline (Figure 6.3(c)). Next, ASIPMeister, 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 6.3(d), 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 pipe. This design process is repeated for different number of pipelines (as shown in the design flow (Figure 6.2) until a design that meets a given design criteria is created.

The forwarding and instruction encoding methods introduced in this chapter can be incorporated into the design systems, as indicated by the dashed block in Figure 6.2.

### 6.1 Forwarding Design

Forwarding improves performance at the cost of area. Our experimental study shows a full forwarding can incur over 30% area cost when compared to the design without forwarding. This value can grow if the complexity of the network becomes large. 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

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

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 $P$ for such information. The element $P(i, j, k, l)$ 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 $P$, this forwarding path will be deleted.

However, some forwarding paths may be under-utilized with low values in $P$. 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 $P$ are generated as possible; finally, all forwarding paths that are not used at all (i.e. with a zero entry in $P$) 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, as shown on the example in Figure 6.4. 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.

The forwarding customization approach is given in Algorithm 4. An example is given in Figure 6.4. In this example, there are three pipelines in the processor. All three pipelines have instructions. Figure 6.4(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
Algorithm 4 Forwarding Network Reduction:

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

source inputs of EX stages. Instruction sequences for each pipeline are shown in Figure 6.4(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 6.4(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 6.4(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 6.4(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 6.4(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.

In the instruction $instr18$ its operand $f$ dependents on $instr14$, operand $b$ depends on $instr8$
and the operand \( g \) depends on \( \text{instr15} \). If the \( \text{instr18} \) is in pipe 3, there need to be 3 forwarding paths as follows: the operand \( b \) is forwarded from the \( RW \) stage of the pipe 2 to the decoding stage of the pipe 3 as the operand \( s_1 \), the operand \( f \) is forwarded from the \( EX \) stage of the pipe 2 to the decoding stage of the pipe 3 as the operand \( s_2 \), and the operand \( g \) if forwarded from the \( EX \) stage of the pipe 3 to the decoding stage of the pipe 3 as the operand \( s_3 \). There exist 3 forwarding paths used by only these operands. The operands \( b, f \) and \( g \) of \( \text{instr18} \) depend on \( \text{instr8}, \text{instr14} \) and \( \text{instr15} \) respectively. Here if the instruction \( \text{instr18} \) is shifted to pipe 2, operand \( g \) can use the existing forwarding path from \( EX \) stage of pipe 2 to the operand \( s_3 \) of pipe 2, thus getting the operand from \( \text{instr15} \) to \( \text{instr18} \). Now the other 2 operands \( b, f \) of \( \text{instr18} \) need two new forwarding paths to be created since there is no path existing in pipe 2 for operands \( b, f \) from \( RW \) to \( s_1 \) and \( EX \) to \( s_2 \) respectively. These required paths have been created still the total number of forwarding paths were reduced to 7 from 8 as indicated in Figure 6.4(d) thus benefitting the design.

### 6.2 Instruction Encoding

Instruction encoding is using binary bits with certain format 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
6.2. INSTRUCTION ENCODING

Table 6.1: Three Encoding Approaches

<table>
<thead>
<tr>
<th>registers</th>
<th>RBE $(b_3b_2b_1b_0)$</th>
<th>RBE $(c_1c_0)$</th>
<th>REE $(c_1c_0)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>r2</td>
<td>0010</td>
<td>00</td>
<td>10</td>
</tr>
<tr>
<td>r4</td>
<td>0100</td>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>r13</td>
<td>1101</td>
<td>10</td>
<td>01</td>
</tr>
<tr>
<td>r11</td>
<td>1011</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 6.2: Decoding Logic (a)RBE (b)REE

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_3$</td>
<td>$c_1$</td>
<td>$b_3$</td>
<td>$c_0$</td>
</tr>
<tr>
<td>$b_2$</td>
<td>$c_1 \oplus c_0$</td>
<td>$b_2$</td>
<td>$\bar{c}_1$</td>
</tr>
<tr>
<td>$b_1$</td>
<td>$c_1 \oplus c_0$</td>
<td>$b_1$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>$b_0$</td>
<td>$c_1$</td>
<td>$b_0$</td>
<td>$c_0$</td>
</tr>
</tbody>
</table>

(a) (b)

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 6.1, 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 improving performance, area and power. We call this technique reduced-bit efficient encoding or REE. Column 4 in Table 6.1 gives an example of such an encoding. The related decoding logic functions for RBE and REE encoding are given in Figures 6.2(a) and 6.2(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 6.5(a) represents four registers: r10, r12, r31 and r17 in a 64-register
file. There are six bit-columns.

<table>
<thead>
<tr>
<th>b5</th>
<th>b4</th>
<th>b3</th>
<th>b2</th>
<th>b1</th>
<th>b0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 6.5: Register Encoding Example

Given a register array for a register operand field, we determine the encoding values by Algorithm 5. 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 6.5(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 5 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
6.2. INSTRUCTION ENCODING

Algorithm 5 Operand Field Encoding:

// determine the number of bits required for the field
field_size = \log_2(\text{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;

$b_0$ 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 6.5(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 5.

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 6.6. 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., \texttt{asrl} 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.

As represented on Figure 6.6(b), instructions 4-10 require encoding of the operand field, and their operand field bits are indicated by "*". Each instruction on the left figure (a) is associated with the instruction code on the right figure (b). All instructions are sorted based on operand field size (as already done in Figure 6.6).

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 \(PC1\) in Figure 6.6(b). We refer this encoded bit size as the \textbf{partial encoding size} (that will be used in Algorithm 6).

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 $PC_5$. 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 6.

The operation field encoding is summarized in Algorithm 6. For each instruction type, the instruction operation encoding progresses from inner most level to the outmost level; and the code size grows accordingly.

Algorithm 6 Instruction Encoding

1. group instructions according to their total size of operand fields;
2. encodingDone = FALSE;
3. while encodingDone = FALSE do
4. current_group = instructions with smallest partial encoding size;
5. encode the current group with smallest number of bits;
6. bit_difference = operand field size of next group - partial encoded size of current group
7. if bit_difference $\neq 0$ then
8. pad groups with $bit\_difference$ zeros()as necessary to match the partial code sizes;
9. encode current_group with minimal number of bits;
10. end if
11. if all instruction types have been fully encoded then
12. encodingDone = TRUE;
13. end if
14. end while

6.3 Simulations and Results

By applying forwarding network reduction and instruction encoding to the multi-pipe processor design system in Chapter 5, we designed ASIPs for a set of applications from Mibench [66]. All results are associated with a 3-pipe architecture.

As described in section 6, we utilized ASIPmeister, an ASIP design tool for single pipeline processor, in generating multi-pipeline processor VHDL models for each of the appli-
1. `asr1` `r2`, `r5`, `#3`
2. `mov2` `r0`, `r6`
3. `mov1` `r3`, `#2`
4. `lsr1` `{r5, r7}`, `{r2, r1}`, `{#4, #1}`
5. `sub2` `{r2, r3}`, `{r4, r5}`, `{#1, #2}`
6. `lsl1` `{r2, r3}`, `{r2, r0}`, `{#28, #24}`
7. `add3` `{r0, r1}`, `{r0, r1, r3}`, `{r2, r5, r6, r9}`
8. `sub3` `{r2, r4, r7}`, `{r5, r8}`, `{r0, r1, r3}`
9. `ldr2` `{r0, r2, r3}`, `[[r2, r1, r0], {r3, r5, r7}]`
10. `str2` `{r5, r3, r7}`, `[[r2, r1, r5], {r1, r0, r2, r4}]`

**Figure 6.6: Instruction Encoding Example**
6.3. SIMULATIONS AND RESULTS

cations. 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 6.7.

![Experimental Setup Diagram]

**Figure 6.7: Experimental Setup**

Performance is evaluated with the processor clock speed, which is given by Design Compiler, and the clock cycles given by Modelsim. The leakage power and area are estimated by the Synopsys Design Compiler.
Forwarding Network Customization

The forwarding network reduction was tested and its effect on performance, area and leakage power is shown in Figures 6.8, 6.9 and 6.10, 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.

Instruction Encoding

Figure 6.11 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 6.12. As can be seen REE is better than RBE, with less area cost, as expected.

Forwarding Network and Instruction Encoding

Table 6.3 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
Comparison for Customized Forwarding

Figure 6.8: Full Forward vs. Custom Forward (Exec. Time)
Figure 6.9: Full Forward. vs. Custom Forward. (Area)
Figure 6.11: Code Widths with REE approach
Comparison of REE and RBE

Figure 6.12: Comparison of REE and RBE
Table 6.3: 3-pipe Designs with forwarding and instruction encoding customization

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Clk Period (ns)</th>
<th>Exec. Time (ms)</th>
<th>Area (UnitCell)</th>
<th>Leak.Pow (mw)</th>
<th>C.Size (byte)</th>
<th>Perf.</th>
<th>Area</th>
<th>Power</th>
<th>C.Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>no-cus.</td>
<td>cus.</td>
<td>no-cus.</td>
<td>cus.</td>
<td>no-cus.</td>
<td>cus.</td>
<td>no-cus.</td>
<td>cus.</td>
<td>no-cus.</td>
</tr>
<tr>
<td>Adp.dec</td>
<td>3.23</td>
<td>3.07</td>
<td>66.1</td>
<td>62.8</td>
<td>198490</td>
<td>135751</td>
<td>345</td>
<td>228</td>
<td>1626</td>
</tr>
<tr>
<td>b.count</td>
<td>3.18</td>
<td>3.07</td>
<td>137.5</td>
<td>132.7</td>
<td>183154</td>
<td>133129</td>
<td>316</td>
<td>224</td>
<td>1350</td>
</tr>
<tr>
<td>bas.math</td>
<td>3.21</td>
<td>3.06</td>
<td>72.8</td>
<td>69.4</td>
<td>183787</td>
<td>128335</td>
<td>316</td>
<td>216</td>
<td>564</td>
</tr>
<tr>
<td>cstrings</td>
<td>3.63</td>
<td>3.42</td>
<td>38.9</td>
<td>36.6</td>
<td>199537</td>
<td>150923</td>
<td>347</td>
<td>259</td>
<td>690</td>
</tr>
<tr>
<td>dijkstra</td>
<td>3.17</td>
<td>3.03</td>
<td>48.4</td>
<td>46.3</td>
<td>180817</td>
<td>133654</td>
<td>320</td>
<td>220</td>
<td>1554</td>
</tr>
<tr>
<td>endian</td>
<td>3.19</td>
<td>3.05</td>
<td>24.9</td>
<td>23.8</td>
<td>184566</td>
<td>125936</td>
<td>324</td>
<td>210</td>
<td>1002</td>
</tr>
<tr>
<td>q.sort</td>
<td>3.62</td>
<td>3.41</td>
<td>119.6</td>
<td>112.6</td>
<td>203324</td>
<td>154692</td>
<td>359</td>
<td>266</td>
<td>4452</td>
</tr>
<tr>
<td>sha.encr</td>
<td>3.20</td>
<td>3.11</td>
<td>58.6</td>
<td>56.9</td>
<td>200599</td>
<td>138789</td>
<td>351</td>
<td>236</td>
<td>2622</td>
</tr>
<tr>
<td>str.search</td>
<td>3.10</td>
<td>3.08</td>
<td>25.7</td>
<td>25.5</td>
<td>170020</td>
<td>136911</td>
<td>297</td>
<td>222</td>
<td>1374</td>
</tr>
</tbody>
</table>
6.4. CONCLUSIONS

(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 6.3, 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.

6.4 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. In addition, with the proposed instruction encoding techniques, instruction memory size can be theoretically reduced by 69%.

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 pro-
grammed to run other applications (but may not perform optimally), which is just the very
feature of ASIPs.
In this dissertation, we have presented a design methodology to generate multi-pipeline heterogeneous application specific instruction set processor. Given an application, the methodology is capable of yielding an efficient design with multiple pipelines. The parallelism of the program is maximally explored to generate a processor with high performance and low power and area.

Researches on ASIPs so far have mainly focused on single pipeline processor, which can execute instructions in sequence only. One of the ways to increase the performance is to explore the instruction level parallelism of a given application. By doing so there will be increase in logic circuit for a parallel processor structure and hence increasing the area and power. Application specific processors are designed for a given application therefore it is much easier to tailor the processor for the given application and make it cost effective. A heterogeneous parallel design can reduce the replication of each data path and thereby maximally utilizing existing datapaths.

This thesis has aimed to design a processor structure to maximally exploit the parallelism of a given application with minimal area cost.

The proposed design methodology in the thesis uses a parallel processor structure with
heterogeneous multiple pipelines. In this methodology, an efficient parallel instruction scheduler is used to maximally explore the parallelism of an application with a suitable number of pipelines and carefully assign functional units to each pipeline. The experimental simulation results show the design methodology is feasible and can achieve high performance/area for a given application.

The proposed methodology and techniques have resulted in high performance with low area and power along with a code size reduction. This methodology can be used with simple and existing tools, it is not time consuming. The design flow can be followed by anyone as the methodology is not tool dependent. The scheduler is independent of the compiler and can be modified according to the target processor used.

The approach presented in this dissertation is application oriented, therefore more application oriented techniques can be applied to further reduce the cost. An example of such a techniques is reducing the bus communication cost by rearranging the memory contents. Also, multiple multi-pipeline processors can be generated to efficiently execute an application.

The proposed architecture is more suitable for more compute intensive but less memory oriented application with more parallelism.

**Future Work**

Future extensions to the proposed architectures in each chapter are discussed in the same chapters. Further to this, an important topic for the future work is applying the proposed architecture to larger applications. Such a work may reveal the hidden benefits of the proposed architecture for that application area.
This glossary defines some of the terminology and acronyms that are used throughout the dissertation.

**ADL**  
Architectural Description Language

**ALU**  
Arithmetic Logic Unit

**ANSI**  
American National Standards Institute

**ASIC**  
Application Specific Integrated Circuit

**ASIP**  
Application Specific Instruction-set Processor

**CC**  
Clock Cycle

**CDFG**  
Control and Data Flow Graph

**CISC**  
Complex Instruction Set Computer

**DAG**  
Directed Acyclic Graph

**DFG**  
Data Flow Graph

**DFL**  
Data Flow Language
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMAU</td>
<td>Data Memory Access Unit</td>
</tr>
<tr>
<td>DMEM</td>
<td>Data Memory</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>EPIC</td>
<td>Explicitly Parallel Instruction Computing</td>
</tr>
<tr>
<td>FU</td>
<td>Functional Unit</td>
</tr>
<tr>
<td>GCC</td>
<td>GNU C Compiler</td>
</tr>
<tr>
<td>GPP</td>
<td>General Purpose Processor</td>
</tr>
<tr>
<td>GPS</td>
<td>Global Positioning System</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>HW</td>
<td>Hardware</td>
</tr>
<tr>
<td>HW/SW</td>
<td>Hardware Software</td>
</tr>
<tr>
<td>ILP</td>
<td>Instruction Level Parallelism</td>
</tr>
<tr>
<td>IMEM</td>
<td>Instruction Memory</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>IR</td>
<td>Intermediate Representation</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>ISG</td>
<td>Instruction Set Graph</td>
</tr>
<tr>
<td>ISS</td>
<td>Instruction Set Simulator</td>
</tr>
<tr>
<td>JPEG</td>
<td>Joint Photographic Experts Group</td>
</tr>
<tr>
<td>MCC</td>
<td>Multi Cycle Complex instruction</td>
</tr>
<tr>
<td>MIMD</td>
<td>Multiple Instruction Multiple Data</td>
</tr>
<tr>
<td>Acronym</td>
<td>Full Form</td>
</tr>
<tr>
<td>---------</td>
<td>-----------</td>
</tr>
<tr>
<td>MOP</td>
<td>Micro OPeration</td>
</tr>
<tr>
<td>NOP</td>
<td>No Operation</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter</td>
</tr>
<tr>
<td>PDA</td>
<td>Personal Digital Assistant</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>ROM</td>
<td>Read Only Memory</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>SCC</td>
<td>Single Cycle Complex instruction</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>SITU</td>
<td>Small Integrated Transmitter Unit</td>
</tr>
<tr>
<td>VLIW</td>
<td>Very Long Instruction Word</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very-High-Speed Integrated Circuit</td>
</tr>
</tbody>
</table>


[97] LANCE-TEAM. Lance c compile system. (http://www.iss.rwth-aachen.de/).


[119] PEAS-TEAM. ASIP-meister.


[136] Aviral Shrivastava, Eugene Earlie, Nikil Dutt, and Alex Nicolau. Operation tables for scheduling in the presence of incomplete bypassing. In *CODES+ISSS*


[148] ARM TEAM. Thumb processor.


[158] Xtensa-TEAM. Xtensa processor.


