PBExplore: A Compiler-in-the-Loop Framework for Design Space Exploration of Partially Bypassed Processor Pipelines

User Manual

Version 1.0

01/15/2008

Sanghyun Park§, Aviral Shrivastavaψ,
Nikil Dutt†, Eugene Earlie‡,
Alex Nicolau† and Yunheung Paek§

shpark@compiler.snu.ac.kr §, Aviral.shrivastava@asu.edu ψ,
dutt@uci.edu †, eugene.earlie@intel.com ‡,
nicolau@uci.edu †, ypaek@ee.snu.ac.kr §

Compiler Microarchitecture Labs, Arizona State University ψ
ACES Laboratory, University of California Irvine†
SO&R Laboratory, Seoul National University§
Strategic CAD Labs, Intel Corp‡

1. Introduction

Bypasses or forwarding paths are simple yet powerful and widely used feature in modern processors to eliminate some data hazards [4]. With Bypasses, additional datapaths and control logic are added to the processor so that the result of an operation is available for subsequent dependent operations even before it is written in the register file. This benefit of bypassing comes with significant impact on the wiring area on the chip, possibly widening the pitch of the execution-unit datapaths. Paths including the bypasses often are timing critical and cause pressure on cycle time, especially the single cycle paths. The delay of bypass logic can be significant for wide issue machines. Due to extensive bypassing very wide multiplexers or buses with several drivers may be needed. Apart from the delay, bypass paths increase the power consumption of the processor. Thus bypasses have a significant impact, in terms of area, cycle time, and power consumption of the processor [2]. The situation is aggravated owing to the trend of long pipelines and high degrees of parallelism in modern processors.

Initial exploration experiments have shown that some bypasses may be rarely used in an application, and a few of them may not be used at all. Thus some bypasses may be removed with a very minimal impact on performance [3]. With incomplete bypassing becoming popular in modern processors, developing retargetable compilers for such processors is important not only to easily adapt to minor changes in the design, but also for rapid and automated design space exploration of processors with such features.

Traditionally retargetable compilers use constant operation latencies to avoid data hazards [5]. Operation latency is defined as the delay (in cycles) between the cycles when the operation is issued to the cycle when dependent operations can use its results. Complete bypassing enables the subsequent dependent operations to read the value of the result as soon as the result is generated, irrespective of when it is written back in the register file. Complete bypassing reduces the operation latency. In the absence of bypassing or in the presence of complete bypassing, the operation latency can be modeled by a single value, and can be used in a DFG to detect data hazards. In processors with incomplete bypassing however, the result of an operation is only sometimes bypassed and thus available for dependent operations intermittently. Due to incomplete bypassing, the operation latency cannot be accurately described by a single value.

To our knowledge there is no existing technique that models such multi-valued operation latency, and uses it to detect data hazards. However conservative scheduling can be performed by assuming the operation latency to be equal to the latency in absence of any bypassing. Although such scheduling produces legitimate schedules (even for statically scheduled processors), it does not allow the compiler to effectively exploit available bypasses. The other option is to perform optimistic scheduling by assuming the operation latency to be equal to the latency in presence of complete bypassing. It can be shown that any scheduling technique that uses constant operation latency will produce sub-optimal results, and that a better schedule may be generated using precise latency of operations. Thus there is a growing need for a schema to detect data hazards in a retargetable compiler framework in the presence of incomplete bypassing.

We solved this problem using Operation Tables[1] (OTs). An OT models the resources and registers used by each operation supported by the processor. OTs can then be used to detect both the resource

and data hazards more accurately in an integrated manner. This OT-Compiler is also using AutoOT technique, an algorithm to Automatically generate Operation Tables from a high-level processor description. For the detail of AutoOT technique, refer to [7].

This document will serve as a manual for users who want to use the OT-based compiler which is able to schedule the code precisely in the presence of incomplete bypassing.

1.1 Organization of User Manual

This user manual is organized as follows.

Section 2 explains how to set up the PBExplore on your system.

Section 3 describes the command line option of PBExplore and how to run it.

Finally, **Section 4** presents useful references for anyone interested in understanding the theory behind the PBExplore

Appendix A briefly describes different sections of EXPRESSION ADL language. The detailed description of the EXPRESSION language can be found in [6].

1.2 Recommended System Configuration

The OT-Compiler has been tested on the following system:

OS Name : Microsoft Windows XP Professional
Version : 5.1.2600 Build 2600
System Type : X86-based PC
Processor : x86 Family 15 Model 1 Stepping 2 GenuineIntel ~1 Ghz
Total Physical Memory : 512 MB
Total Virtual Memory : 1.72 GB
Page File Space : 1.22 GB

Development Platform : Microsoft Visual C++ .NET (Express Edition)

2. Installation

The download contains
There are two versions of OT-Compiler: Windows and Linux version. Windows version of OT-Compiler is implemented with **Microsoft Visual C++ .NET** on an **i686** machine running Microsoft Windows XP.

With the functional simulator, the correctness of the scheduled file can be verified, and with the cycle-accurate simulator, you can get statistics for the scheduled file.

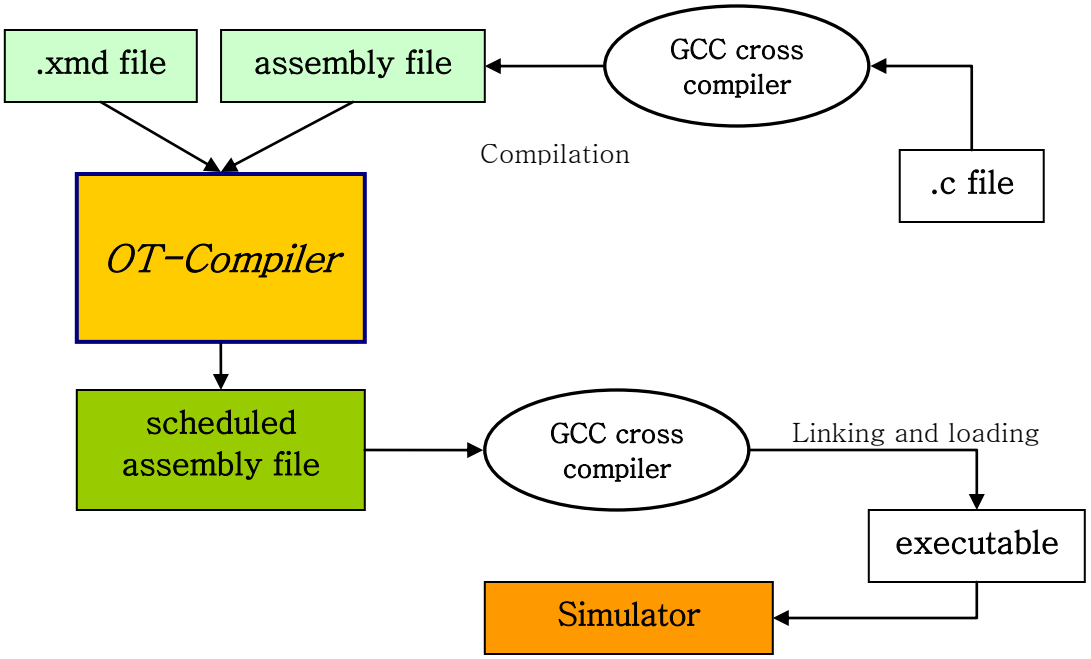


Figure 1. OT-Compiler flow

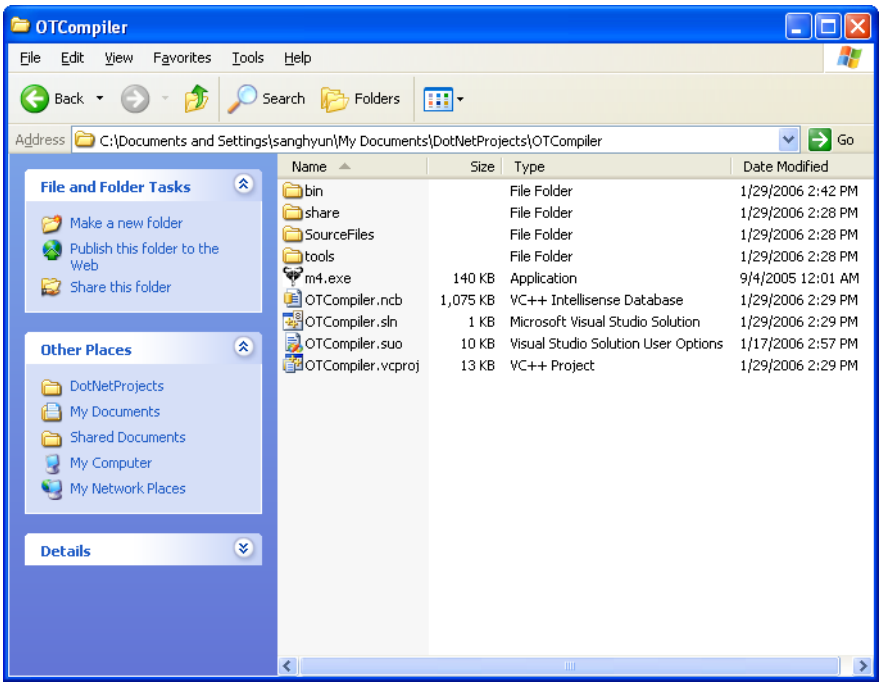
OT-Compiler takes two input files: EXPRESSION Architecture Description Language(ADL) file and input assembly file (see figure 1).

The input assembly file is compiled file using GCC which is ported on the target machine. And the .xmd is a file which describes the target architecture using EXPRESSION ADL.

In this section, the setup processes of the two OT-Compiler versions and its overall flow will be presented. Also, the simulation process will be described briefly.

2.1 OT-Compiler Setup on Windows

Unzipping OTCompiler.zip yields the following directories and files.



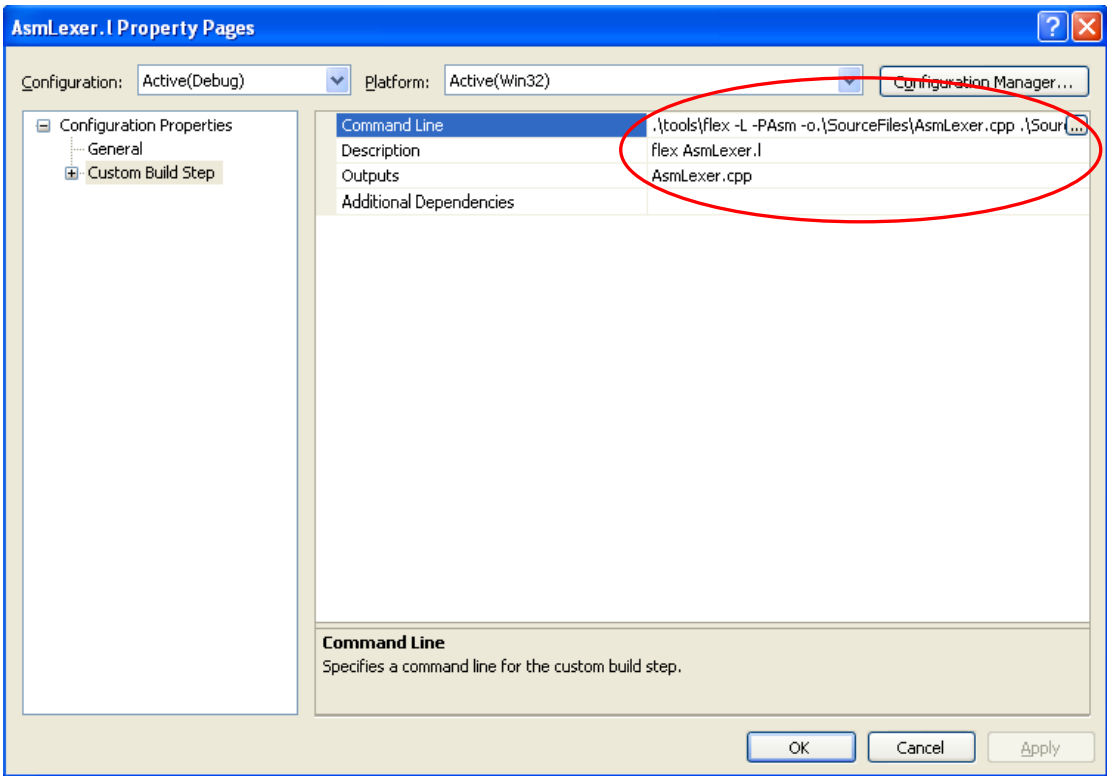
Directories

bin	: work directory that contains files to be compiled
share	: contains files needed for using flex/bison on Window
SourceFiles	: contains source files of OT-Compiler
tools	: contains files needed for using flex/bison on Window

Files

m4.exe
OTCompiler.sln
OTCompiler.suo
OTCompiler.vcproj

You can either create a new Visual C++ console project or use the existing solution file provided(if you create a new project, you should select "Not Using Precompiled Headers" on the OTCompiler Property Page→ C/C++ → Precompiled Headers). Once you create the project file and add all source files and header files to that, the next step is setting the properties for flex/bison files. There are 2 flex files (AsmLexer.l and MachineLexer.l) and 2 bison files(AsmParser.y and MachineParser.y). At the 'property' pages of each file, you can set the properties like below.



AsmLexer.l
- Command Line :
  .\tools\flex -L -PAsm -o.\SourceFiles\AsmLexer.cpp .\SourceFiles\AsmLexer.l
- Description (optional)
  flex AsmLexer.l
- Outputs
  AsmLexer.cpp

AsmParser.y
- Command Line
  .\tools\bison -l -p Asm -d .\SourceFiles\AsmParser.y
  move .\SourceFiles\AsmParser.tab.c .\SourceFiles\AsmParser.tab.cpp
- Description (optional)
  bison AsmParser.y
- Outputs
  AsmParser.tab.cpp AsmParser.tab.h

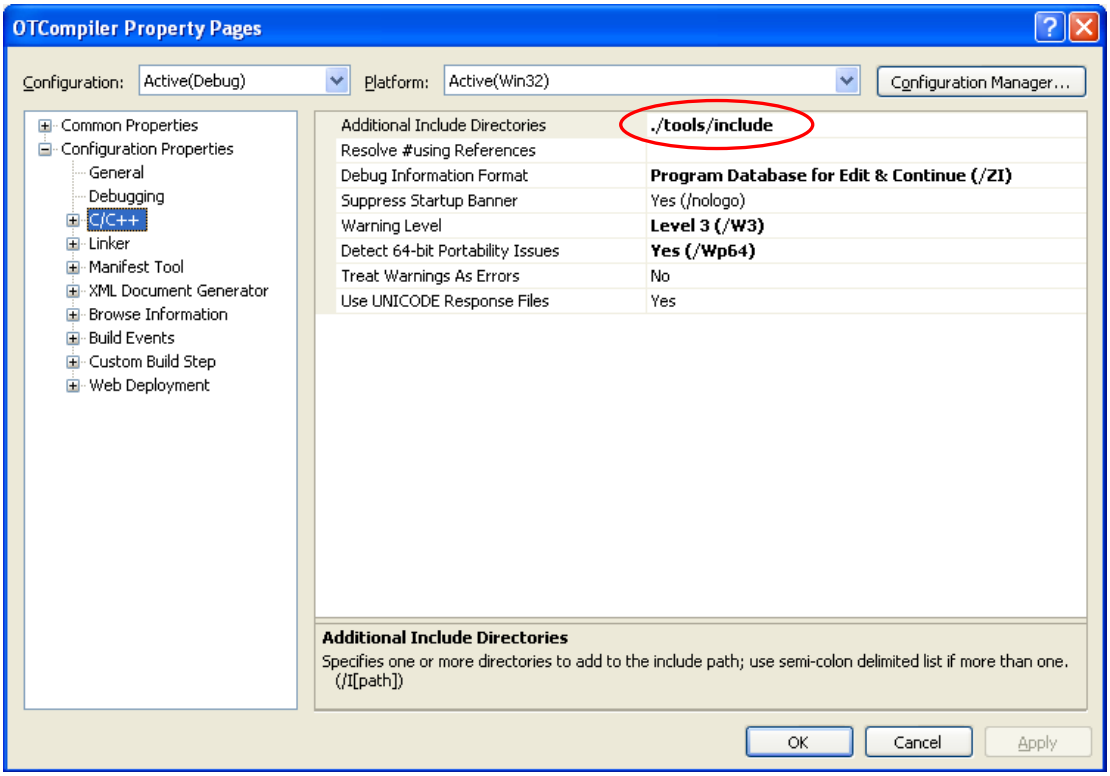
MachineLexer.l
- Command Line
  .\tools\flex -L -PMachine -o.\SourceFiles\MachineLexer.cpp .\SourceFiles\MachineLexer.l
- Description (optional)
  flex MachineLexer.l
- Outputs

MachineLexer.cpp

MachineParser.y

- Command Line
  .\tools\bison -l -p Machine -d .\SourceFiles\MachineParser.y
  move .\SourceFiles\MachineParser.tab.c .\SourceFiles\MachineParser.tab.cpp
- Description (optional)
  bison MachineParser.y
- Outputs
  MachineParser.tab.cpp MachineParser.tab.h

To compile flex and bison files on Visual Studio .NET, you should include additional directory like below.



Now that all setting is done, you are ready to build OT-Compiler and get the executable file. We are recommending Release mode rather than Debug mode if you intend to just run this compiler.

2.2 OT-Compiler Setup on Linux

Unzipping OTCompiler.tar.gz yields the following directories and files.

Directories
  bin
  src

XScale ADL file is in bin directory, and the executable file of OT-Compiler will be put in it. src directory contains OT-Compiler source files and a Makefile for them. Just building them using ‘make’ will generate the compiler in bin directory. No other setting is required.

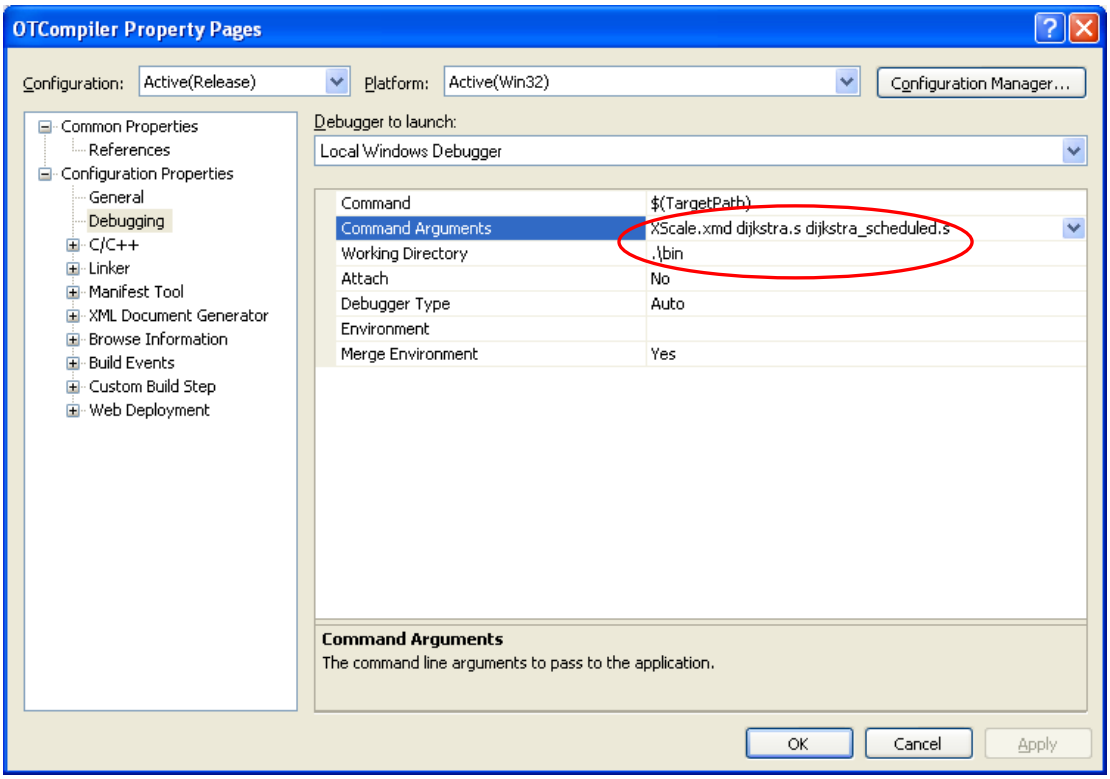
2.3 OT-Compiler Flow

To begin with, we take a Intel XScale based architecture. In the rest of this section, this architecture will be frequently referred to for the purpose of illustration. The EXPRESSION ADL description of XScale is available in <OTCompiler>\bin directory. We will present only the OT-Compiler flow for Window, because the flow on Linux is very similar to this.

Our compiler uses a GCC front end. First the applications are compiled using GCC cross-compiler, that generates code for the Intel XScale processor. The assembly code is generated using GCC, by enabling all performance optimizations (with -O3). Our compiler reads the assembly file generated by GCC, performs instruction reordering to reduce the execution cycle.

Thus the first step is making a GCC cross-compiler for Intel XScale to get input assembly file. The test file (dijkstra.s) that is generated by GCC cross-compiler for Intel XScale is also available in <OTCompiler>\bin directory.

If you are running OT-Compiler using Visual Studio .NET, you can set the command arguments as follows.



Or you can run OT-Compiler with command window. Then, you can change directory to <OTCompiler>\bin, and type this :

..\Release\OTCompiler.exe XScale.xmd dijkstra.s dijkstra_schedule.s

The new assembly (dijkstra_schedule.s) is linked using the GCC linker, and executable is generated. The executable is simulated on our XScale cycle-accurate simulator (or functional simulator) to find out the performance of the application

2.4 Simulation Flow

The correctness of the scheduled file can be verified with the functional simulator. And with the cycle-accurate simulator, you can get statistics for the scheduled file. The simulators take disassembly file generated by GCC cross-compiler as an input. The test disassembly file for dijkstra is in the bin directory.

For the explanation, we assume that you located the unzipped directory to <OTCompiler>. To use XScale functional simulator, move to <OTCompiler>\XScaleFunctionalSimulator\
and type this :

XScaleFunctionalSimulator.exe ..\bin\dijkstra.dis ..\bin\dijkstra_input.dat

dijkstra_input.dat is the input file of the dijkstra program. If you want to simulate another benchmark program, you should give the parameters for the program to the simulator as inputs like above.

Using the cycle-accurate simulator for XScale is exactly same with the functional simulator.

3. Command Line Option

OT-Compiler takes 3 parameters : (1) EXPRESSION ADL file, (2) input assembly file name and (3) output assembly file name. The EXPRESSION file(.xmd) is a file that describes target architecture, and the input assembly file is a compiled file using GCC which is ported to the target machine. The third parameter is just a name for output file.

Ex > OTCompiler.exe XScale.xmd dijkstra.s dijkstra_scheduled.s

4. References

[1]A. Shrivastava, E. Earlie, N. Dutt, and A. Nicolau. Operation tables for scheduling in the presence of incomplete bypassing. In *Proc. of CODES+ISSS 2004*, 2004.

[2] P. Ahuja, D. W. Clark, and A. Rogers. The performance impact of incomplete bypassing in processor pipelines. In *Proc. of Symposium on Microarchitecture MICRO-28*, 1995

[3] K. Fan, N. Clark, M. Chu, K. V. Manjunath R. Ravindran, M. Smelyanskiy, and S. Mahlke. Systematic register bypass customization for application-specific processors. In *Proc. of IEEE Intl. Conf. on ASSAP*, 2003.

[4] P. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. 1990.

[5] S. Muchnick. *Advanced Compiler Design and Implementation*. 1998.

[6] P. Grun, A. Halambi, A. Khare, V. Ganesh, N. Dutt and A. Nicolau. EXPRESSION: An ADL for System Level Design Exploration. *ICS Technical Report # 98-29, University of California, Irvine, September 1999.*

[7] S. Park, A. Shrivastava, N. Dutt, E. Earlie, A. Nicolau and Y. Paek. Automatic Generation of Operation Tables for Fast Exploration of Bypasses in Embedded Processors. In *proc. of DATE, 2006.*

Appendix A : EXPRESSION ADL

EXPRESSION employs a simple LISP-like syntax to ease specification and enhance readability. An EXPRESSION description is composed of two main sections: Behavior (or IS) and Structure. The Behavior section is further sub-divided into Operations, Instruction and Operation Mappings sections. The Structure section is sub-divided into Components, Pipeline/Data Transfer Paths and Memory Subsystem sections. The brief description of the EXPRESSION ADL components will be presented below.

There are 6 components in EXPRESSION file :

1. Register Files
2. Register File Connections
3. Operation Groups
4. Operation Format
5. Units
6. Latches

Designer can define the register file in Register Files section. In this section, the number of register in the register file is specified and the read/write ports are described.

Register File Connections section contains the data paths of the architecture. Since all memory component and pipeline unit has the ports, designer can describe the connection as a pair of ports.

In Operation Groups, designer can merge the operation with similar characteristics (load_op, RF_op, .etc) into one abstract group. In the XScale.xmd, the operations are grouped according to the pipeline stage. For example, X1_ops is the group of operations that should be passed through X1 pipeline stage.

Operation format defines the instruction format of the operand. The operand which has multiple formats can be easily defined in this section.

The pipeline stages(units) are defined in Units section. It contains the operation groups that should be passed through, ports on it, and latches. Designer can also describe the micro operation that is broken in that unit.

Finally, latches between the pipeline units are defined in Latches section.