IATO, IA64 Toolkit

User Guide

Revision 1.6 - 2004

Amaury Darsch - Pierre Villalon - Andre Seznec

Contents

Chapter 1

Getting started

This chapter is an introduction to **IATO**, the **IAOO toolkit**; a flexible environment that permits to analyze, emulate or simulate the IA64 Instruction Set Architecture (ISA) binary executables. **IATO** is a flexible and portable toolkit that is built around a set of C++ libraries and client programs. This chapter presents the general organization as well as how to get started.

1.1 Distribution

IATO is distributed ;as a set of libraries and application clients. The **IATO** clients are applications programs based on the **IATO** libraries. The fundamental clients are the IA64 emulator and simulator. Other clients provides supports for program analysis and statistical computation. The **IAOO toolkit** is distributed four client programs, as described below:

- **IAOS client program**
  A client program that analyzes IA64 binary executables.

- **IAKA client program**
  A client program that emulates and traces IA64 binary executables.

- **IAIO client program**
  A client program that simulates and traces IA64 binary executables in an in-order mode.

- **IAOO client program**
  A client program that simulates and traces IA64 binary executables in an out-of-order mode.

- **IATA client program**
  A client program that computes statistics from trace produced by the **IAKA** and **IAOO** clients.

1.1.1 Supported platforms

The **IATO** environment is supported on various platforms. The primary platform is the Linux based IA64 machine. Linux IA32 platforms are also supported as well as SUN SPARC Solaris machines.

1.1.2 IA64 binary executables

The **IATO** environment operates directly with ELF binary executables. As of release 1.0, fully static binary executables are only supported. In the presence of dynamically linked executables, the **IATO** clients reports an error and terminates. The best method to check for a file type is to use the `file` command.

Example 1 The UNIX file command with program p0001.

```
zsh> file p0001
p0001: ELF 64-bit LSB executable, IA-64 version 1 (SYSV),
       statically linked, not stripped
```

1.2 Installing IATO

IATO is distributed as a compressed tar file. **IATO** has been successfully tested with the GCC 3 compiler. **IATO** is completely written in C++ and uses the STL.

1.2.1 Instruction

The **IATO**distribution is available as a compressed tar file. There is no need for special privileges to install the distribution. The following command unpacks the distribution.

```
zsh> gzip -d iato-src-[version].tar.gz
zsh> tar  xf iato-src-[version].tar
```

The distribution is placed into a directory call `iato`. The `make` command is used to compile the distribution. By default, the distribution is compiled in debug mode. In order to compile it in optimized mode, the `make world` command is appropriate.

```
zsh> cd iato
zsh> make world
```

Once compiled, the distribution can be installed with the help of the `make install` command. The default location for this installation is the `/usr/local` directory. This can be overwritten by using the `DIRINSTALL` variable.

```
zsh> make install DIRINSTALL=/usr
```

The previous command will install the distribution under the `/usr` directory. The executable will be placed in the `/usr/bin` directory while the libraries will be placed into the `/usr/lib` directory. The root directory in built from the `DIRINSTALL` variable. Note that the `LIBINSTALL` and `BININSTALL` variables control the final location of the libraries and binary programs.

1.3 Testing the distribution

IATO is distributed with a set of test programs that are located in the `tst` directory. There are two classes of test programs. The first class is located under the `tst/lib` directory and is cross-platform independent. This simple test program permits to simulate the libraries and are not exhaustive. ON the other end, the `tst/clt` contains a complete test suite that can be used to stimulate the framework. In order to use this distribution, an IA64 platform is required for to perform the compilation. If this is not possible, the binary version is available for your convenience.

Chapter 2

Client Overview

The **IATO** clients are specific applications based on the **IATO** libraries. The fundamental clients are the IA64 emulator and simulator. Other clients provides supports for program analysis and statistical computation. The **IATO** programming environment is divided into four client programs, as described below:

- **IAOS client program**
  A client program that analyzes IA64 binary executables.
- **IAKA client program**
  A client program that emulates and traces IA64 binary executables.
- **IAOO client program**
  A client program that simulates and traces IA64 binary executables.
- **IATA client program**
  A client program that computes statistics from trace produced by the **IAKA** and **IATO** clients.

2.1 IAOS client program

IAOS is a simple client program that produces static information from a binary executable. The resulting information comprises, among other things, disassembled object code and various statistics. The primary reason to use the **IAOS** client is to collect information about a binary executable.

2.1.1 IAOS options

The `-h` option can be used to report the program usage as indicated below.

```
usage: iaos [option] name
  -h        help message

  options related to elf image
  -e        report elf information
  -e:h      report elf header
```

```
  -e:e      report elf executable
  -e:t      report elf text section

  options related to verification
  -v        verify instructions
  -v:b      verify bundles only

  options related to disassembly
  -d        disassemble
  -d:a      disassemble address
  -d:b      disassemble bundles
  -d:i      disassemble instructions
  -d:f      ignore disassembly error

  options related to statistic operation
  -s        compute statisitics
  -s:b      compute bundle distibution
  -s:i      compute instruction distribution
  -s:n      compute percentage of nops
```

2.1.2 ELF information

The major '-e' option permits to collect some information about the binary executable. UNIX platform uses the ELF (Executable Library Format) as a standard to encode binary executable. With **IAOS** it is possible to dump the file header as well as other information.

Reporting the ELF header

The '-e:h' option permits to report some ELF header information. This option is mostly useful for checking a program type.

Example 2 Reporting the ELF header with iaos

```
zsh> ./iaos -e:h p_0001
file name       : p_0001
file type       : static executable file, 64 bits class
interpreter     : not specified
entry point     : 0x4000000000000220
program headers : 4
section headers : 32
```

This example reports the ELF header for the program `p_0001` which is part of the distribution. The interesting information is that the program is statically linked (as reported by the `file` command) and the entry point (i.e the program start address) is available.

2.1.3 Binary executable report

The '-s' and other related options permits to gather some program statistics. In the simplest form, the number of instructions can be computed as illustrated below.

Example 3 Getting program statistics with iaos

```
zsh> ./iaos -s p_0001
number of instructions         : 122607
nop instructions               : 36696  (29.9%)
predicated instructions        : 16573  (13.5%)
predicated instructions non br : 8647   (7.05%)
```

Instruction nop distribution

The frequency of "nop" instructions can be computed with the '-s:n' option. In such case, the global statistic is also computed.

Example 4 Getting nop statistics with iaos

```
zsh> ./iaos -s:n p_0001
number of nop instructions [M] : 9442
number of nop instructions [I] : 8730
number of nop instructions [F] : 15005
number of nop instructions [B] : 3519
number of nop instructions [X] : 0
number of nop instructions     : 36696

number of instructions         : 122607
nop instructions               : 36696  (29.9%)
predicated instructions        : 16573  (13.5%)
predicated instructions non br : 8647   (7.05%)
```

Bundle and instruction distribution

A detailed bundle distribution can be computed with the '-s:b' option. With the '-s:i' option, a detailed instruction frequency is reported. In each case, the global statistic is also computed.

2.2 IAKA client program

IAKA is a client program that emulates and traces IA64 binary executables. The emulator executes the common loop: fetch, decode and execute. The emulator operates on statically linked IA64 compliant binary program. The program must be in the ELF

format (as found on UNIX compliant machine). During the execution, the emulator computes the true results for each instruction. System calls are emulated as well. The standard input, output and error stream are shared between the emulator and the emulated program, although, by default, the emulator operates silently.

2.2.1 IAKA options

The **IAKA** client as numerous options. The `-h` option can be used to report the program usage as indicated below.

```
usage: iaka [option] name [args]
  -h                    help message

  options related to the tracer
  -t:v                  verbose tracer
  -t:r rcd1,rcd2        record type to trace
  -t:s src1,src2        record source to trace
  -t:f name             set tracer file name
  -t:t threshold        set tracer threshold
  -t:i                  set branch target tracing
  -t:o                  set branch tracing only
  -t:p                  set predicate tracing only
  -t:b #                begin trace cycle number
  -t:e #                end trace cycle number

  options related to the emulator
  -c                    enable checker mode
  -s                    enable stat report
  -m #                  maximum bundle count
  -p:s name:type=value  set a context parameter
  -p:d                  dump the context to the output

  options related to the architecture
  -W #                  issue width in bundles
  -G #                  number of general registers
  -a:d                  enable disperse mode
  -a:p                  branch pipeline refill
  -a:b bprd             branch predictor type
```

2.2.2 Trace generation

The '-t' based option control the tracer behavior. The '-t:f' is used to specify the trace file name. The traces are stored in a binary form that can be processed later by the **IATA** client. When operating with the verbose mode (option '-t:v'), the trace is reported on the standard output in a readable form.

The example above show a trace record produced in verbose mode. The trace identification is built with two names. The first one is the record source name (EMU,

Example 5 Trace record produced in verbose mode.

```
zsh> ./iaka -t:v p_0001
...
trace 408 {
 EMU:BUNDLE @0x4000000000003c60 19 70 01 4a 00 21 e0 00 3c
                                30 20 00 00 00 00 20
 RBK:RGREAD gr[037][107]        0x0000000000000040[F]
 EMU:RINSTR @0x4000000000003c60 [A04] [M] [T] mov r46=r37
 RBK:REGUPD gr[046][116]        0x0000000000000040[F]
 RBK:RGREAD gr[015][015]        0x6000000000010310[F]
 EMU:RINSTR @0x4000000000003c60 [M01] [M] [T] ld8 r14=[r15]
 MLG:SMEMRD @0x6000000000010310 0x40000000000b12a0 [8]
 RBK:REGUPD gr[014][014]        0x40000000000b12a0[F]
 EMU:RINSTR @0x4000000000003c60 [B09] [B] [T] nop.b 0x0;;
}
...
```

RBK, etc...), and the second one is the record type (BUNDLE, RINSTR, etc...). Two options are provided as a way to filter the record generation. The '-t:s' permits to select the record source while the '-t:r' options permits to select the record type. Each name or type must be comma separated.

Example 6 Trace record produced in verbose mode with filtering.

```
zsh> ./iaka -t:v -t:r RINSTR p_0001
...
trace 408 {
 EMU:RINSTR @0x4000000000003c60 [A04] [M] [T] mov r46=r37
 EMU:RINSTR @0x4000000000003c60 [M01] [M] [T] ld8 r14=[r15]
 EMU:RINSTR @0x4000000000003c60 [B09] [B] [T] nop.b 0x0;;
}
...
```

The '-t:b' and '-t:e' options permits to select the begin and end cycle cycle for the generated traces. If the traces are saved in a trace file, the trace file header will save this options in its header.

2.2.3 Context setting

All client parameters are stored in an application context. When the emulator is started, the context is initialized with some default values that have been found to be appropriate. The '-p:d' option can be used to dump the context parameters. Each parameter comes in the form of a name, a type and a value. The format 'name:type=value' is the one used during the dumping operation and can be used to change a particular value with the '-p:s' option.

Example 7 Changing a client parameter.

```
zsh> ./iaka -p:s "STB-SIZE:long=32" p_0001
```

2.2.4 Emulator statistics

On top of its emulation function, the **IAKA** client can perform additional operation such like data checking or statistics computation. Data checking is primarily used for testing. Statistics, on the other end, is useful for almost any emulated program. The '-s' option enables statistics generation, which are produced at the end of the program emulation.

Example 8 Statistics generation with iaka.

```
zsh> ./iaka -s p_0001

program name                   : p_0001
number of bundles              : 1205
number of instructions         : 3547
nop instructions               : 1058   (29.8%)
predicated instructions        : 451    (12.7%)
predicated instructions non br : 175    (4.93%)
```

Note that such statistics can also be computed from the trace file, by using the **IATA** client.

2.3 IAIO client program

IAIO is a client program that simulates and traces binary IA64 executables in an in-order mode. The **IAIO** simulator implements an 8 stages in-order core engine, with true predicated instruction execution based on the Itanium 2 microarchitecture. The simulator is cycle accurate and provides numerous architectural options that permits to perform a resource fine tuning. During the course of the simulation, a stage by stage trace can be generated as well as a summarized statistic. The simulator is primarily used to computed the program IPC.

2.3.1 IAIO options

The **IAIO** client program has numerous options. The `-h` option can be used to report the program usage as indicated below.

```
usage: iaio [option] name [args]
  -h                    help message
  -r                    report all resources

  options related to the tracer
```

```
  -t:v                    enable tracer verbose
  -t:r rcd1,rcd2          record type to trace
  -t:s src1,src2          record source to trace
  -t:f name               set tracer file name
  -t:t threshold          set tracer threshold
  -t:h                    set tracer reschedule
  -t:i                    set branch target tracing
  -t:b #                  begin trace cycle number
  -t:e #                  end trace cycle number

  options related to the simulator
  -c                      enable checker mode
  -s                      enable stat report
  -s:c #                  enable cycle stat report
  -m   #                  maximum cycle count
  -p:s name:type=value    set a context parameter
  -p:d                    dump the context to the output

  options related to the architecture
  -W   #                  issue width in bundles
  -M   #                  number of M units
  -I   #                  number of I units
  -F   #                  number of F units
  -B   #                  number of B units
  -G   #                  number of gr (registers)
  -a:b bprd               branch predictor type
```

2.3.2 Trace generation

Trace generation within the simulator is similar with the emulator. Actually, the options are the same. The only difference is with the number of resources available, since the simulator consider each pipeline stage as a resource, the full trace include all the information produced by all stages. Generally, only the trace produced by the *commit* stage (CMT) is needed.

Example 9 Write-backed instructions with the iaio client.

```
zsh> ./iaio -t:v -t:s WRB p_0001
...
trace 222 {
 WRB:RINSTR @0x4000000000000530 [A08] [M] [T] cmp.eq p7,p6=0,r14
 WRB:RINSTR @0x4000000000000530 [I19] [I] [T] nop.i 0x0
 WRB:RINSTR @0x4000000000000540 [M37] [M] [T] nop.m 0x0
 WRB:RINSTR @0x4000000000000540 [F15] [F] [T] nop.f 0x0
}
...
```

2.4 IAOO client program

IAOO is a client program that simulates and traces binary IA64 executables. The **IAOO** simulator implements a 12 stages out-of-order core engine, with true predicated instruction execution based on translation register buffer (TRB). The simulator is cycle accurate and provides numerous architectural options that permits to perform a resource fine tuning. During the course of the simulation, a stage by stage trace can be generated as well as a summarized statistic. The simulator is primarily used to computed the program IPC.

2.4.1 IAOO options

Similar to the **IAIO** client program, the **IAOO** client program has numerous options. The `-h` option can be used to report the program usage as indicated below.

```
usage: iaoo [option] name [args]
  -h                    help message
  -r                    report all resources

  options related to the tracer
  -t:v                  enable tracer verbose
  -t:r rcd1,rcd2        record type to trace
  -t:s src1,src2        record source to trace
  -t:f name             set tracer file name
  -t:t threshold        set tracer threshold
  -t:h                  set tracer reschedule
  -t:i                  set branch target tracing
  -t:b #                begin trace cycle number
  -t:e #                end trace cycle number

  options related to the simulator
  -c                    enable checker mode
  -s                    enable stat report
  -s:c #                enable cycle stat report
  -m #                  maximum cycle count
  -p:s name:type=value  set a context parameter
  -p:d                  dump the context to the output

  options related to the architecture
  -W #                  issue width in bundles
  -M #                  number of M units
  -I #                  number of I units
  -F #                  number of F units
  -B #                  number of B units
  -G #                  number of general registers
  -a:n                  ignore nop for execution
  -a:b bprd             branch predictor type
```

```
  -a:p pprd              predicate predictor type
```

2.4.2 Trace generation

Trace generation within the simulator is similar with the emulator. Actually, the options are the same. The only difference is with the number of resources available, since the simulator consider each pipeline stage as a resource, the full trace include all the information produced by all stages. Generally, only the trace produced by the *commit* stage (CMT) is needed.

Example 10 Committed instructions with the iaoo client.

```
zsh> ./iaoo -t:v -t:s CMT p_0001
...
trace 222 {
 CMT:RINSTR @0x4000000000000530 [A08] [M] [T] cmp.eq p7,p6=0,r14
 CMT:RINSTR @0x4000000000000530 [I19] [I] [T] nop.i 0x0
 CMT:RINSTR @0x4000000000000540 [M37] [M] [T] nop.m 0x0
 CMT:RINSTR @0x4000000000000540 [F15] [F] [T] nop.f 0x0
}
...
```

2.4.3 Resource report

All processor resources can be reported with the '-r' option. The resource name and associated parameters are shown at the end of the simulation. Generally, the resource can be altered by changing their respective parameters as described in the **iaka** section.

Example 11 Resource report by the iaoo client.

```
zsh> ./iaoo -r p_0001
...
resource : IIB
        resource type      : instruction interrupt buffer
        iib size           : 128
...

resource : IPG
        resource type      : instruction pointer generation
        bundle window size : 2
        bytes  window size : 32
        delayable latency  : 1
...
```

When the resource is a pipeline stage, the name can be used as source filter for the tracer. For information, the following stages are currently implemented in the

simulator.

Table 2.1 Pipeline stages summary.

Stage	Description
IPG	Instruction fetch
DEC	Decode
LRN	Logical renaming
PRN	translation renaming
EXP	Instruction expansion
SLC	Instruction selection
REG	Register read
EXE	Instruction execution
WBK	Register write-back
CMT	Instruction commit

Multiple stages can be combined in order to produce a multi-stage trace report. For example, the LRN and CMT stages are combined in the same trace file with the '-t:s' option.

Example 12 Combined stage for simulation report.

```
zsh> ./iaoo -t:f p_0001.trc -t:s "LRN,CMT" p_0001
...
trace 564 {
 CMT:RINSTR @0x40000000000190a0 [I19] [I] [T] nop.i 0x0
 CMT:RINSTR @0x40000000000190b0 [M37] [M] [T] nop.m 0x0
 LRN:RINSTR @0x40000000000191b0 [A04] [M] [T] adds r15=16,r12;;
 LRN:RINSTR @0x40000000000191b0 [M01] [M] [T] ld1 r14=[r15]
 LRN:RINSTR @0x40000000000191b0 [I19] [I] [T] nop.i 0x0;;
 LRN:RINSTR @0x40000000000191c0 [M37] [M] [T] nop.m 0x0
 LRN:RINSTR @0x40000000000191c0 [I29] [I] [T] sxt1 r14=r14;;
 LRN:RINSTR @0x40000000000191c0 [A08] [I] [T] cmp4.eq p6,p7=91,r14
}
...
```

2.5 IATA client program

The **IATA** client program analyzes and prints binary traces produced by the other clients. It can also generate one trace from another. Trace generation can be seen as a post-processing operation. Another useful feature is the statistic computation. Given a trace file, the **IATA** client can extract for example, the instruction distribution or other runtime information.

2.5.1 IATA options

The `-h` option can be used to report the program usage as indicated below.

```
usage: iata [option] name
  -h               help message

  options related to filter
  -f:t rcd1,rcd2         record type selection filter
  -f:n src1,src2         record source selection filter
  -f:p src1:rcd1         pair selection filter
  -f:g igr1:igr2         instruction group selection

  options related to trace file
  -t:h                   write trace header
  -t:b #                 starting trace number
  -t:e #                 ending trace number

  options related to statistics
  -s                     compute all statisitic
  -s:b                   add bundle distibution
  -s:i                   add instruction distribution
  -s:n                   add nop distribution
  -s:e #                 ignore # empty traces

  option related to trace generation
  -g                     generate default trace
  -g:o                   trace output
  -g:w #                 the instruction window size
  -g:p #                 the branch penalty
  -g:f #                 the f-unit latencie

  option related to the analyzer
  -a:p name              analyze a predicate predictor

  options related to the analyzer
  -d                     dump all traces
  -p:s name:type=value   set a context parameter
```

2.5.2 Statistical results

The results produced by the statistical analyzer are dynamic results, as opposed to the results produced by the **IAOS** client. The capability to manipulate traces, permits also to produce statistical results that includes specific target machine parameters (like the F unit latency or the branch penalty).

Chapter 3

Library Overview

This chapter describes the **IAOO Toolkit**; a flexible environment that permits to analyze, emulate or simulate the IA64 Instruction Set Architecture (ISA) binary executables. **IATO** is a flexible and portable toolkit that is built around a set of C++ libraries and client programs. This chapter describes the library functionalities.

3.1 Organization

The **IATO** programming environment is divided into four libraries, as described below:

- **ISA library**
  A library that implements the IA64 instruction set.
- **ELF library**
  A library that implements the support for IA64 binary executables.
- **KRN library**
  A library that implements the support for Linux compatible IA64 system calls.
- **MAC library**
  A library that implements the support for detailed architectural simulation.
- **ECU library**
  A library that implements the support for special architectures.

3.2 ISA library

The ISA library is a set of classes that provides supports for the IA64 instruction set. Among other things, the library provides an instruction decoder as well as support for instruction execution. Additionally, the library implements the resources defined by the ISA, like the register bank. The table below illustrates the base classes available in the library.

The library has been designed to be portable across various platforms. Although, the IA64 is a 64 bits architecture, the library has been successfully tested within a 32 bits environment. Such implementation is possible with a careful design of base

Table 3.1 ISA library main classes.

Class	Description
Register	IA64 register bank
Bundle	IA64 bundle decoder
Instr	IA64 instruction
Operand	IA64 instruction source register
Result	IA64 instruction destination register
Mdecode	IA64 M instruction type decoder
Adecode	IA64 A instruction type decoder
Idecode	IA64 I instruction type decoder
Fdecode	IA64 F instruction type decoder
Bdecode	IA64 B instruction type decoder
Xdecode	IA64 X instruction type decoder
Mexecute	IA64 M execution unit type
Iexecute	IA64 I execution unit type
Fexecute	IA64 F execution unit type
Bexecute	IA64 B execution unit type
Memory	IA64 memory interface
Alat	IA64 Advanced Load Address Table
Rse	IA64 Register Stack Engine
Record	Trace record information
Tracer	Trace record handler
Interrupt	IA64 interruption

types. Note also that the floating-point computation has been re-implemented within this library.

3.2.1 Theory of operations

The best way to visualize the ISA library operations is to describe a complete flow that starts at the instruction fetch and finishes at the register update. Starting with a memory address, a bundle object is constructed by pushing the object bytes. The class decodes the bundle and produces up to 3 instructions. The instruction is decoded during the object construction. Note that the instruction class also provides mechanism to reconstruct the instruction (i.e recode). When the instruction is successfully decoded, the associated operands are constructed and evaluated through the register bank. The combination of the instruction and the evaluated operands is used to execute the instruction. Since the library provides one execution engine per unit, a dispatch operation is necessary, in order to route the instruction to the appropriate execution object. The instruction execution produces an result object, that is later used to update the memory and the register bank.

3.3 ELF library

The Executable and Linking Format (ELF) library is a set of classes that provides support for ELF standard. It also provides support for processor-specific Application Binary Interface (ABI) [5]. The library is responsible to load the program and initialize the memory as well as to manage memory accesses during IA64's code emulation or simulation. The table below illustrates the base classes available in the library.

Table 3.2 ELF library main classes.

Class	Description
ElfImage	Complete ELF object representation
ElfInterp	ELF interpreter
ElfKernel	Process kernel parameters
ElfArgs	Program arguments manager
ElfExec	All process memory segments
ElfBsa	Process backing store area
ElfStack	Process memory stack
ElfMap	Memory mappable segment
ElfBrk	Memory heap segment

3.3.1 Theory of operations

As explained before the ELF library is responsible for loading and initializing the memory image. During this process, the binary image is extracted by the **ElfImage**. The **ElfImage** is also responsible to determine the organization of the image (statically or dynamically linked executable). Once initialized, a memory heap is created as well as the memory program stack, the memory backing store area and the shared memory segment. The final operation initializes the program arguments in the program stack.

3.4 Kernel emulation library

The kernel (KRN) library is a set of classes that handles Linux system calls. Systems calls are vectored traps sent by the program. They are caught by the emulator or the simulator and routed to the system call handler. The **Syscall** class encapsulates all Linux system calls. Note that a system call argument mapping procedure is also included into this library.

3.4.1 Theory of operations

The **Syscall** object receives system calls that are encapsulated into an **Interrupt** objects. When routed, the system call can be either mapped to a host system call or processed by a specific library handler. Mapped system calls are those using the host resources; like `open`, `read` or `write`. Specific system calls are those using process dependent data like `mmap` or `sbrk`. Upon completion, the system call return value is available in a specific register (generally r8 or r10).

3.4.2 Portability

The kernel library is extremely sensitive to the supported operating system. As of this writing, the library support the Linux kernel 2.4 series. There is some limitations that are related to system calls that manipulates complex kernel data structures. For instance, the `sigaction` system call requires to manage the internal process data structure; a task that goes beyond the scope of this preliminary library version.

3.5 MAC library

The MAC library is a set of classes that permit to build a micro-architectural simulator. Most of design is centered around an out of order architecture. There is numerous objects that implements small functionalities available in a superscalar processor. Among others, the basic pipeline operation with instruction decode and renaming is supported. Standard instruction scheduling with reservation station is also available in this library. The table below illustrates the base classes available in the library.

Table 3.3 Mac library main classes.

Class	Description
Stage	Base pipeline stage
Pipelane	micro-pipeline object
Pipeline	Pipeline object

3.5.1 Theory of operations

The library provides the base implementation for a superscalar IA64 processor. Each simulator implements its stage description locally with the help of the library functionality. The stage sequencing is done by the library. Numerous resources are available as well and can be used to design an innovative microarchitecture.

Bibliography

[1] *Intel Itanium Architecture. Software Developer's Manual. Application Architecture.*, 2000.

[2] *Intel Itanium Architecture. Software Developer's Manual. Instruction Set Reference*, 2000.

[3] *Intel Itanium Architecture. Software Developer's Manual. System Architecture.*, 2000.

[4] *Intel Itanium Processor. Reference Manual for Software Development.*, 2000.

[5] *Intel Itanium Processor-specific Application Binary Interface (ABI).*, 2001.

[6] *Itanium Software Conventions and Runtime Architecture Guide.*, 2001.

[7] Harsh Sharangpani, Ken Arora. Itanium Processor Microarchitecture. *IEEE Micro*, pages 24–43, September 2000.