ITANIUM INSTRUCTION-SET SIMULATOR

BY

PRIYANKA SHAH

A Thesis Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirement for the Degree of
Master of Science

Southern Connecticut State University
New Haven, Connecticut
August 2006
INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.
ITANIUM INSTRUCTION-SET SIMULATOR

BY

PRIYANKA SHAH

The thesis was prepared under the direction of the candidate’s thesis advisor, Dr. Ata Elahi, Department of Computer Science, and it has been approved by the members of the candidate’s thesis committee. It was submitted to the school of Graduate Studies and was accepted in partial fulfillment of the requirements for the degree of Master of Science.

Dr. Ata Elahi, Ph.D.
Thesis Advisor

Hrvoje Podnar, Ph.D.
Second Reader

Sandra C. Holley, Ph.D.
Dean of Graduate Studies

August 25, 2006
Date
Today most large organizations need high performance powerful machines for enterprise computing. Although majority of the servers still processors data and instructions in 32-bit chunks, there is a steady increase in the demand for 64 bit processors. Many organizations recognize that machines which process data in 64-bits provide a sound infrastructure for supercomputing and provide for significant benefits over 32 bit processor machines.

One of the most popular 64-bit processors is Intel’s Itanium 64-bit processor. It is specially designed for high-performance scientific and enterprise computing. Since
Itanium machines today, are essentially used as servers and the costs of owning them is very high, it is very difficult for an educator to learn more about its workings. If one wishes to learn more about the Itanium machine and the Itanium instruction set there is very limited number software that can help them with it.

The objective of this thesis is to develop a web-based 64 bit Itanium Instruction Set Simulator (IISS) application that will understand, interpret and run Itanium instructions. Since this application is web based, users can run it on practically any non Itanium platform including Windows and Linux.

The software will serve as a useful tool for users who want to test or develop Itanium programs without having to own an Itanium machine. It allows users to enter Itanium-instructions, and see the results. It will also serve as a tutorial for students learning the Itanium architecture and its instruction set.
1. INTRODUCTION .............................................................................................................1

1.1 64-bit Computing.........................................................................................................1

1.2 The 64 bit Power PC G5 Architecture ......................................................................3

1.2.1 Introduction.............................................................................................................3

1.2.2 Power PC Registers...............................................................................................3

1.2.3 Power PC Instruction Format ...............................................................................6

1.2.4 Power PC Architecture.........................................................................................14

2. THE ITANIUM ARCHITECTURE ..............................................................................17

2.1 Supporting Environments .........................................................................................18

2.2 Speculation ................................................................................................................19

2.2.1 Control Speculation..............................................................................................20

2.2.2 Data Speculation..................................................................................................20

2.3 Predication ..................................................................................................................22

2.4 Instruction Level Parallelism ....................................................................................23

3. ITANIUM REGISTERS .................................................................................................24

3.1 Integer Registers ........................................................................................................25

3.2 Floating Point Registers ........................................................................................25

3.3 Qualify Predicate Registers ....................................................................................25

v

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
3.4 Branch Register .................................................................26
3.5 Loop Count (LC) Register .....................................................26
3.6 Current Frame Maker (CFM) Register ....................................26
3.7 Instruction Pointer (IP) ..........................................................26
3.8 Control and status Registers ................................................27
    3.8.1 Application Registers ................................................27
    3.8.2 User Mask (UM) Registers ..........................................27
    3.8.3 CPU Identification Registers ......................................27
3.9 Register Stack .................................................................27
3.10 Branching ........................................................................28
3.11 Register Rotation .............................................................28

4. ITANIUM INSTRUCTION FORMAT ........................................29
   4.1 General Instruction Format ..............................................29
   4.2 Itanium's Execution Units ..............................................30
   4.3 Instruction Formats .......................................................31
   4.4 Instruction Bundles .......................................................32
   4.5 A type instruction format ..............................................33
   4.6 A2 type instruction format ............................................36

5. ITANIUM INSTRUCTIONS ....................................................40
   5.1 Add Instruction ............................................................40
5.2 Subtraction Instruction ....................................................40
5.3 Logical AND Instruction ..................................................41
5.4 AND Complement Instruction ..........................................42
5.5 Exclusive OR Instruction ..................................................42
5.6 Move Instruction ..........................................................43
5.7 Shift Left Instruction ......................................................43
5.8 Shift Right Instruction ....................................................44
5.9 Load Instruction ..........................................................45
5.10 Store Instruction ..........................................................45
5.11 Compare Instruction ......................................................46
  5.11.1 Compare equals ......................................................46
  5.11.2 Compare less than ...................................................46
  5.11.3 Compare greater than ..............................................47
  5.11.4 Unconditional Compare equals ..................................47
  5.11.5 Unconditional Compare less than ................................47
  5.11.6 Unconditional Compare greater than ..........................48
5.12 Parallel Compare ..........................................................49
  Compare equal or ..........................................................49
  Compare equal and ..........................................................49
  Compare equal and or .....................................................49
5.13 Comparison of Itanium and Pentium Instructions ..................50
10. CONCLUSION AND FUTURE WORK .................................................. 85

APPENDICES

APPENDIX A RUNNING THE ITANIUM INSTRUCTION-SET SIMULATOR
SOFTWARE ......................................................................................... 87

APPENDIX B CODE USED IN THE PROGRAM ...................................... 89

REFERENCES .................................................................................... 134
LIST OF TABLES

Table 1. Comparison between 32 bit and 64 bit machines ...........................................2
Table 2. Itanium architecture supporting environments .............................................18
Table 3. Itanium Instruction formats ........................................................................31
Table 4. IA64 Bundle Formats ..................................................................................33
Table 5. A type instruction opcodes ..........................................................................36
Table 6. A2 type instruction opcodes ..........................................................................38
Table 7. Integer Compare Instruction format ..............................................................39
Table 8. Pentium Program and Description .................................................................51
Table 9. Itanium Program and Description ................................................................51
LIST OF FIGURES

Figure 1. PowerPC Registers ................................................................. 6

Figure 2. PowerPC Register Instruction Format ...................................... 7

Figure 3. PowerPC Immediate Instruction Format .................................... 8

Figure 4. PowerPC Unconditional Branch Format .................................. 9

Figure 5. Register Indirect Mode Format .............................................. 11

Figure 6. Register Indirect with Indexing Mode Format ......................... 12

Figure 7. Power PC Architecture ......................................................... 14

Figure 8. Itanium Registers .................................................................... 24

Figure 9. General register with an extra bit called Not a Thing (NaT) ....... 25

Figure 10. 41 bit Instruction Format ..................................................... 29

Figure 11. Block diagram of Itanium ................................................... 30

Figure 12. Instruction Bundle format ................................................... 32

Figure 13. A Type Instruction format .................................................... 33

Figure 14. Opcode for Add Instruction .................................................. 34

Figure 15. A Type Add Instruction with X2b field ................................ 34

Figure 16. A Type Subtract Instruction .................................................. 35

Figure 17. A Type Subtract Instruction with X2b field ......................... 35

xi
Figure 18. A2 Type Instruction format.................................................................36
Figure 19. A2 Type Subtract Instruction format.........................................................37
Figure 20. A2 Type Integer Compare Instruction format..............................................38
Figure 21. Unconditional Compare along with Itanium instructions.................................48
Figure 22. Pentium program's flow chart.................................................................50
Figure 23. The Program Controller..........................................................................57
Figure 24. The Program Window...............................................................................61
Figure 25. The Data Window.....................................................................................62
Figure 26. The Results Window..................................................................................63
Figure 27. Program 1 Instructions..............................................................................67
Figure 28. Program 1 Memory....................................................................................68
Figure 29. Program 1 Results (Registers and Memory).....................................................68
Figure 30. Program 1 Results (Output).....................................................................69
Figure 31. Program 2 Instructions..............................................................................71
Figure 32. Program 2 Memory....................................................................................72
Figure 33. Program 2 Results (Registers and Memory)....................................................72
Figure 34. Program 2 Results (Output).....................................................................74
Figure 35. Program 3 Instructions..............................................................................77
Figure 36. Program 3 Memory....................................................................................78
Figure 37. Program 3 Results (Registers and Memory)....................................................78
Figure 38. Program 3 Results (Output).....................................................................80
1. INTRODUCTION

There is an ever increasing demand for computers that can perform complex computations as well as process more data faster. More and more companies realize the limitations of their computers with 32 bit processors that begin to display performance limitations as the complexity of their tasks increase.

Using 64 bit processors provides a huge edge for companies that need to deliver peak performance. The processors provide increased scalability, increased performance, reduced query time, more effective use of memory and better computational processing. Given their processing power, the processors are ideally fitted for supercomputing environments.

1.1 64-bit Computing

Very often processors are described as 32 bits or 64 bits and one often wonders what this actually means. A bit (Jupitermedia Corporation, 2005), or a binary digit, is the smallest unit of information used by a computer. It can hold only one of two values: 0 or 1. When consecutive bits are combined together into larger units they become more meaningful.
Computers are classified by the number of bits they are capable of processing at any given time and also by the number of bits they use to identify each address in memory. Describing a computer as a 64 bit machine means that it has processor registers with the size of 64 bits, and it is able to process 64 bits of data at any given time. This allows a computer to run faster and support longer programs. The basic properties related to the numbering systems used in 32 and 64 bit computers are presented in Table 1.

Table 1.

*Comparison between 32 bit and 64 bit machines*

<table>
<thead>
<tr>
<th>32 Bit Machine</th>
<th>64 Bit Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>➢ A 32 Bit Machine represents numbers from 0 to $2^{32} - 1$ which is 4,294,967,295 in decimal</td>
<td>➢ A 64 Bit Machine represents numbers from 0 to $2^{64} - 1$ which is 18,446,744,073,709,551,615 in decimal</td>
</tr>
<tr>
<td>➢ Up to $1111111111111111111111111$ numbers in binary</td>
<td>➢ Up to $1111111111111111111111111$ numbers in binary</td>
</tr>
<tr>
<td>➢ Can process 32 bit binary instructions per clock cycle</td>
<td>➢ Can process 64 bit binary instructions per clock cycle</td>
</tr>
<tr>
<td>➢ Arithmetical and logic operations on 32 bits of data</td>
<td>➢ Arithmetical and logic operations on 64 bits of data</td>
</tr>
</tbody>
</table>
The 64 bit systems are generally used by companies and organizations that run demanding applications like video encoding, searching massive databases, loading massive data in memory, conduct scientific research or support large number of users. A majority of home computer users do not require large amounts of memory and computational power, thus 32 bit machine suffices their needs. It is expected that overtime the 64 bit personal computers will be used more frequently.

1.2 64 bit Power PC Architecture

1.2.1 Introduction. IBM, Motorola and Apple have worked together to develop the PowerPC Architecture which can be used in personal computers, workstations and servers. The PowerPC architecture uses 64 bits and also has native support for 32 bit applications. The PowerPC G5 design is based on the PowerPC instruction set, as well as the IBM Power Architecture.

1.2.2 Power PC Registers. The working operations of various registers that are used by PowerPC are as follows (Silha, May, & Frey, 2003a, 2003b, 2003c):

1. General-Purpose Registers (GRs): There are thirty two 64 bit General Registers, (GR0 - GR31), as shown in Figure 1. They are the source and destination operands for all integer operations as well as the sources for address operands for all load/store operations. The General Purpose Registers provide access to the Special-Purpose Registers mentioned later in
this section. All of the General Purpose Registers can be used by the processor except for the
GR0, which has the constant value of 0 and it cannot be changed.

2. Floating-point registers (FRs): There are 32 Floating Point Registers (FR0 - FR31) which
are the source and destination operands of all floating-point operations. They can contain 32-
bit and 64-bit signed and unsigned integer values as well as single-precision and double-
precision floating-point values. Floating-point registers can be combined to form 64 and 128
bit registers.

3. Floating Point Status and Control Register (FPSCR) is a 32 bit register which handles
floating point exceptions and floating point operation record status.

4. Special Purpose Registers (SPR) can be accessed by external applications without system
service support. These include the following:
   a. The Instruction Address Register (IAR) is a 64 bit register used to store the address of the
      current instruction which is why it is also known as the program counter or instruction
      pointer.
   b. The Link Register (LR) is a 64 bit register used to store the address to return to at the end
      of a function call.
   c. The Fixed-Point Exception Register (XER) is 64 bit register which has the carry and
      overflow information from integer arithmetic operations.
d. The Count Register (CTR) is a 64 bit register and acts as a loop counter decremented on certain branch operations.

e. The Condition Register (CR) register is a 32 bit register which is divided into eight CR fields (CR0 – CR7) and each field is 4 bits. CR0 is used to capture integer instruction results. The 4 bits of all CR fields correspond to less than (LT), greater than (GT), equal (EQ) to and overflow (SO) conditions. Floating point exceptions are captured using CR1. Rest of the CR fields are used for capturing the LT, GT, EQ, SO conditions for integer and floating point instructions.

f. The Processor Version Register (PVR) is a 32-bit read-only register that identifies the version of the processor. The processor versions are assigned by the PowerPC architecture process.
<table>
<thead>
<tr>
<th>General Registers</th>
<th>Floating Registers</th>
<th>Conditional Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>63 0</td>
<td>63 0</td>
<td>0</td>
</tr>
<tr>
<td>GR0 = 0</td>
<td>FR0</td>
<td>64 4 4 4 4 4 4 4</td>
</tr>
<tr>
<td>GR1</td>
<td>FR1</td>
<td></td>
</tr>
<tr>
<td>GR31</td>
<td>FR31</td>
<td>63 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IAR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Count Register</th>
<th>FPSCR</th>
<th>Link Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>63 0</td>
<td>31 0</td>
<td>63 0</td>
</tr>
<tr>
<td>XER</td>
<td>PVR</td>
<td>SPR</td>
</tr>
</tbody>
</table>

*Figure 1. PowerPC Registers*

More detailed descriptions can be found in Silha et al. (2003a, 2003b, 2003c).

1.2.3 *PowerPC Instruction Format.* Each instruction is 32 bit long and there are several instruction forms. The 0 bit is the most significant bit of all the bits and the 31st bit is the least significant bit. Most instructions have the 0-5 bits specifying the operation code, also known as opcode. The latter 26 bits contain operand specifiers, immediate operands and extended opcode. Some basic instruction formats are described below (IBM Microelectronics, Motorola, 1993).
The Register Instruction Format

This instruction format as shown in Figure 2 is used by most of the arithmetic and the logical instruction types.

<table>
<thead>
<tr>
<th>opcode</th>
<th>rD</th>
<th>rA</th>
<th>rB</th>
<th>OE</th>
<th>Sub operation specification</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
<td>10</td>
<td>11</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

*Figure 2. PowerPC Register Instruction Format*

- Source operands: rA and rB
- Destination operand: rD
- The OE bit indicated weather the bits of the XER register is altered. If the XER bit are altered then OE = 1 and if they are not OE = 0.
- The rC bit specifies if the less than (LT), greater than (GR), equals to (EQ), and overflow (SO) of the Conditional Register CR0 should be altered. This bit is set for instructions altering these bits.

Examples for Register Instructions:

Here the contents of the rA and rB are added and then stored in rD. The variations are shown below

- add rD, rA, rB

  The CR and the XER bits are not altered. rC = 0 and OE = 0 (not altered).
• add. rD, rA, rB
  
rC = 1 for the dot versions of the add instruction. OE is unaltered and OE=0.
• addo rD, rA, rB
  
rC = 0 for the non-dot versions of the add instruction. OE is altered and OE=1.
• addo. rD, rA, rB
  
rC = 1 for the dot versions of the add instruction. OE is altered and OE=1.

➢ Immediate Instruction Format

This instruction format as shown in Figure 3 is used by instructions that specify the immediate operand.

<table>
<thead>
<tr>
<th>opcode</th>
<th>rD</th>
<th>rA</th>
<th>16 bit signed/unsigned immediate value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5 6</td>
<td>10 11</td>
<td>15 16 31</td>
</tr>
</tbody>
</table>

Figure 3. PowerPC Immediate Instruction Format

• Source operands: rA and the 16 bit immediate value
• Destination operand: rD
Example of Immediate Format:

- `addi rD, rA, Simm16`

  The instruction works like the add instruction except for a few noted differences. It does not affect other condition and status bits and it also uses a value of 0 if the rA is specified as 0 instead of using GR0. This also allows addi instruction to implement load immediate (li), load address(la) and subtract immediate (subi) instructions.

- `li rD, value`

  This is the same as addi rD,0, value.

> Unconditional Branch Format

This instruction format as shown in Figure 4 is used by branch instructions.

<table>
<thead>
<tr>
<th>opcode</th>
<th>24 bit immediate displacement</th>
<th>AA</th>
<th>LK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>56</td>
<td>29</td>
<td>30</td>
</tr>
</tbody>
</table>

*Figure 4. PowerPC Unconditional Branch Format*

- This format allows 24 bit target address specification.
- The AA bit indicates if the address is an absolute address (AA = 1) or a PC relative address (AA = 0).
• The LK bit to link the return address. It is used to convert the branch instruction into a procedure call instruction. When the LK bit is set, the returning address of the instruction following the branch is placed in the Link Register (LR)

Examples of Unconditional Branch Format (There are four possibilities for values in the AA and LK registers):

• Branch
  
  b target (AA = 0 and LK = 0)

• Branch Absolute
  
  ba target (AA = 1 and LK = 0)

• Branch then Link
  
  bl target (AA = 0 and LK = 1)

• Branch Absolute then Link
  
  bla target (AA = 1 and LK = 1)

The control is passed to the target address in case of all instructions. The bl and bla instructions load the Link register (LR) with the address of the instruction following branch instruction. There are also other branch instructions which are outside the scope of this paper.
Load Store Instruction Format

This instruction format as shown in Figure 5 is used by load store instructions.

<table>
<thead>
<tr>
<th>opcode</th>
<th>rD</th>
<th>rA</th>
<th>16 bit signed/unsigned immediate value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
<td>10 11 15 16 31</td>
</tr>
</tbody>
</table>

*Figure 5. Register Indirect Mode Format*

- Source operands: rA and the 16 bit immediate value
- Destination operand: rD

This format is used by load store instructions that specify the following two addressing modes.

- Register Indirect Addressing: The general purpose register rA's contents are used as the effective address in this mode.
- Register Indirect with Immediate Index Addressing: The instructions contain a 16 bit immediate index value imm16. The effective address is computed based on the addition of imm16 and the contents of the general purpose register rA.
Figure 6. Register Indirect with Indexing Mode Format

- Source operands: rA and rB
- Destination operand: rD

The instruction format as shown in Figure 6 is also used by load/store instructions that specify the following addressing mode.

- Register Indirect with Index Addressing: Two General Purpose Registers are specified and the effective address is computed by adding the contents of the two registers.

Examples of Load Store Format:

Based on the three addressing modes above, the can specify rA, rB as the General Purpose Registers and the third register can be rD for load operations and rS for store operations.

- Load Byte and Zero
  
  **lbz** rD, disp (rA)
  
  The address is computed as the sum of the contents of rA and the displacement disp.

- Load Byte and Zero Indexed
  
  **lbzx** rD, rA, rB
  
  The address is computed as the sum of the contents of register rA and rB
This thesis presentation proceeds with the list of selected arithmetic and logical instructions and their detailed descriptions.

- **Add Instruction:** $add\ r6, r4, r3$
- This instruction adds the contents of General Register 4 (GR4) and GR5 and writes the result to GR6. The registers GR4 and GR5 are not affected.
- **Subtract from Instruction:** $subf\ r6, r4, r3$
- This instruction subtracts GR3 – GR4 and writes the result to GR6.
- **Negate Instruction:** $neg\ r6, r4$
- This instruction will take the value from GR4 negate it and assign it to GR6
- **Divide Word Instruction:** $divw\ r6, r4, r3$
- This instruction divides GR4 by GR3 and assigns the result to GR6
- **Multiply low word instruction:** $mullw\ r6, r4, r3$
- This multiples GR4 and GR3 and assigns it to GR6 [low 32 bits]
- **Multiply high word instruction:** $MULHW\ r6, r4, r3$
- This multiples GR4 and GR3 and assigns it to GR6 [high 32 bits]
- **Logical AND:** $and\ r6, r4, r3$
- This instruction performs a logical AND on GR4 and GR3 and assigns result to GR6
- **Logical OR:** $or\ r6, r4, r3$
- This instruction performs a logical OR operation on GR4 and GR3 and assigns the result to GR6

More detailed descriptions can be found in IBM Microelectronics, Motorola (1993).
1.2.4 *Power PC Architecture.* An overview of the Power PC Architecture can be inspected in Figure 7.

*Figure 7. Power PC Architecture*
From Figure 7, it can be seen that the main components of the Power PC Architecture are (Apple Corporation, 2005):

- **L2 Cache**: A large L2 cache (512K) allows for super fast 64GB per second access to data and instructions.

- **L1 Instruction Cache**: The L1 cache (64K) stores the instructions that are pre-fetched from the L2 cache at 64GB per second.

- **L1 Data Cache**: The L1 Data cache (32K) can pre-fetch up to eight active data streams at the same time.

- **Fetch and Decode**: Eight instructions are fetched and decoded per clock cycle from the L1 cache and divided into smaller operations. This helps to increase the processing speed as instructions are dispatched to the execution core and data is loaded in the registers.

- **Instr. 1 – 5**: Instructions are arranged in groups of five before they are dispatched into functional units. The instruction group is broken into individual instructions when dispatched to the execution core. The individual instructions then proceed to their respective function units.

- **Instruction Queues**: All units have their dedicated queues and the instructions are processed in the order needed.

- **Velocity Engine**: Data manipulation takes place at high speed because a single instruction is applied to multiple data units by the Velocity Engine. It has two queues independent of each other and 128 bit registers and data paths.

- **Floating Point units**: Complex scientific calculations and computations can be achieved with the Floating Point units. These units also help in providing higher levels of precision.
- Integer units: The Integer units are used to perform common mathematics like addition and subtraction.

- Load Store: When the queued instructions are processed and the load /store units load the data from the L1 cache and put them in the registers to further process the data. Once the data is processed it is stored in to the L1 and L2 caches. Since there are two load /store units the processing capacity increases dramatically.

- Condition Register: The results of the Condition Register are results of the comparison operators like the LT, GT, EQ and SO conditions. The results are helpful while testing them as branch conditions. This helps improve the data flow in the execution core.

- Branch: Branch predication and speculation is used increase efficiency. This simply means that the next instruction is anticipated and speculation causes that instruction to be executed before it is required.

- Complete Instr. 1 - 5: When processing is completed the instructions are grouped back to five and stored in the cache by the store units.

More detailed descriptions can be found in Apple Corporation (2005).
2. THE ITANIUM ARCHITECTURE

The Itanium architecture is a new parallel architecture jointly developed by HP and Intel under the EPIC (Explicit Parallel Instruction Computing) standard. The design process was driven by the need for high-performance scientific and enterprise applications. Intel's Itanium processor is the first EPIC silicon implementation.

The Itanium uses 64-bit registers and performs 64-bit arithmetic and logic operations; hence it is also known IA-64 (Intel Architecture 64 bit). The Itanium architecture also provides full compatibility with Intel's 32-bit architecture also known as IA-32.

EPIC (Triebel, 2000) has many new features to offer, including such as explicit instruction level parallelism, predication, speculation, advanced branch instructions, a large register file, extensive arithmetic support, rotating registers, large and fast caches, high clock speed, scalability, fast bus architecture, a number of execution units and explicit parallelism which can be used to provide a very sound infrastructure for supercomputing. These features improve its performance massively over the older processor architectures.
Itanium’s ability to address 64 bit memory addressability allows it to be an ideal environment for data warehousing and e-business. The floating point architecture allows it to support high performance digital applications like digital content creation and design engineering.

2.1 Supporting environments

The Itanium architecture supports IA-32 based applications as well as Itanium based applications. Table 2 defines the major systems environments supported by Itanium architectures (Intel Corporation, 2002).

Table 2.

**Itanium architecture supporting environments**

<table>
<thead>
<tr>
<th>System Environment</th>
<th>Application Environment</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IA-64 Instruction Set</td>
<td>Itanium based applications cannot execute in IA-32 system environment</td>
</tr>
<tr>
<td>IA-64 (Itanium)</td>
<td>IA-32 Protected Mode</td>
<td>IA-32 Protected Mode applications in the Intel Itanium system environment.</td>
</tr>
<tr>
<td></td>
<td>IA-32 Real Mode</td>
<td>IA-32 Real Mode applications in the Intel Itanium system environment.</td>
</tr>
<tr>
<td></td>
<td>IA-32 Virtual Mode</td>
<td>IA-32 Virtual 86 Mode applications in the Intel Itanium system environment.</td>
</tr>
<tr>
<td></td>
<td>IA-64 Instruction Set</td>
<td>IA-64 or Itanium based applications in the Intel Itanium based operating systems.</td>
</tr>
</tbody>
</table>
Let us address what makes computers more powerful. More power can be generated by one of the two things; by performing multiple tasks at the same time in parallel, or by processing more tasks in the same amount of time. In older microprocessors, performing multiple tasks required significant amount of synchronization of time and resources. The tasks that take longer to execute, would need to be loaded multiple times, and their state to be reconstructed. Tasks might be required to wait for each other, in order to proceed. 64-bit Itanium processors provide additional functionality in order to eliminate this waiting time. One way to achieve the speed-up can be done by speculation.

2.2 Speculation

The Itanium architecture uses additional special features for speculation that can help detect these exceptions seen above. One of the rules imposed by EPIC is that programs must be written according to sequential semantics. Given a sequential program, speculation allows computation of instructions before they are needed. Instruction Level Parallelism (ILP) is used in order to perform the operations earlier. This in turn will remove the latency associated with operation synchronization. Speculation allows a compiler to exploit ILP and minimize latencies. There are two types of Speculation: Control Speculation and Data Speculation (Intel Corporation, 2002).
2.2.1 Control Speculation. Control Speculation involves executing the operation before the branch that guards it. Control Speculation includes speculative memory loads before a branch that it may or may not be directed to. Based on the outcome of the branch the program will execute only the valid branch.

Example:

\[
\begin{align*}
\text{if (} x > y \text{) load (ld_addr1, target1)} \\
\text{else load (ld_addr2, target2)}
\end{align*}
\]

In the example above, the operations ‘load (ld_addr1, target1)’ and ‘load (ld_addr2, target2)’ are executed before the condition if(x>y) is executed. Since the load is executed before the controlling condition, it is called Control Speculation. If the condition happens to be true and the load operation ‘load (ld_addr1, target1)’ is executed and the remaining load operation ‘load (ld_addr2, target2)’ is discarded. In the case of loading error, the control is passed on to previously defined recovery code.

2.2.2 Data Speculation. Data Speculation, also known as ‘Advanced Loads’ includes loading data from memory before a store instruction is executed. This process could possibly modify the memory which was used within the load if so requested. Data Speculation is capable of recognizing that the data loaded in memory was overwritten or changed and therefore needs to be reloaded. Let us proceed with an example of Data Speculation.
Example:

store (store_address, data)
load (load_address, target)
use (target)

In the example, the store operation stores the data from a register to memory. The load instruction then loads the data from memory possibly to another register. The memory load is executed before the store that precedes it. As an advanced load, its future data values are speculative in nature. If the memory address overlaps at execution, the store might return a different value, therefore the compiler leaves a check instruction at the location of the load. If there was an overlap the check instruction branches to recovery code. The program will execute the instructions in the following sequence:

load (load_address,target)
store(store_address, data)
check(target_, recovery_address)
use(target)

Speculations are supported by the large number of registers. There are 128 Integer registers with length of 64 bits. There are also 128 Floating point registers, as well as 64 predicate registers. The small register sizes in non-64 bit processors limit performance; the processor has to shuffle data in and out of registers more often. This is not the case with the Itanium.
architecture because it provides rotating registers, which reduces the need for register shuffling. More information about rotating registers can be found in section 3.11.

2.3 Predication

Predication is another useful concept for achieving parallelism (Intel Corporation, 2002). All instructions in the registers are conditional and will only execute based on the values in the predicate register. If the predicate register holds a true value, the instruction will be executed, otherwise the instruction will be skipped.

Example:

if (p3 is true) rD = rA + rB

In this example, the execution of the instruction depends on the predicate p3. The predicate register can only hold either zero(false) or one (true). The predication of instructions allows for very simple branch elimination. This helps the compiler to focus on parallelism; instead of branching over instructions that do not need to be executed. All instructions can be now, simply predicated; so that the instructions which we do not want to execute are prefixed by a false predicate.
2.4 Instruction Level Parallelism

Instruction Level Parallelism, also known as ILP (Intel Corporation, 2002) allows multiple instructions can be executed at the same time. Instructions are issued in bundles, each comprising of three instructions. It is possible to execute multiple bundles per clock cycle. The large number parallel resources, including the registers and the execution units allow the compiler to schedule threads simultaneously and manage multiple tasks.

With EPIC, the compiler turns sequential code into parallelized 128-bit bundles. The bundles are directly processed by the processor without need for further interpretation or optimization. This explicit parallelism allows the processor to execute the code faster.

In a regular non-Itanium’s processor family compiler, the sequential code needs to be examined and optimized for parallelism. The processor has to regenerate sequential code in a way that allows it to re-extract the parallelism from it. It then has to read this implied parallelism from the machine code, rebuild and run it. The parallelism is not obvious to the processor and more work is needed by the hardware before parallelism is utilized.

More detailed descriptions can be found in Intel Corporation (October 2002).
3. ITANIUM REGISTERS

An overview of the Itanium Registers can be inspected in Figure 8.

![Diagram of Itanium Registers]

*Figure 8. Itanium Registers*
3.1 Integer Register

![General register with an extra bit called Not a Thing (NaT)](image)

There are 128 (64-bit + 1 bit NaT) general registers represented from GR_0 through GR_127. GR_0 is hardwired to zero, thus the content of it is set to zero. Figure 9 shows general register with an extra bit called Not a Thing (NaT).

3.2 Floating Point Register

There are 128 (82-bit) floating point registers represented from FR_0 through FR_127, where FR_0 and FR_1 are set to zero and 1 respectively.

3.3 Qualify Predicate Register

There are 64 (1-bit) predicate registers represented from P_0 through P_63. P_0 is hardwired to one or true. When the value of P_i is true, the instruction using P_i is executed. When value of P_i for an instruction is false (0), the instruction is not executed.
3.4 Branch Register

There are 8 (64-bit) Branch Registers which are represented by Br₀ through Br₇.

3.5 Loop Count (LC) Register

The Loop Count (LC) register is a 64-bit counter that is used for counting loops.

3.6 Current Frame Maker (CFM) Register

CMF register is a 38-bit register and is used to represent register stack.

3.7 Instruction Pointer (IP)

The 64-bit instruction pointer is used to hold the address of the bundle of the currently executing instruction. Each bundle consists of three instructions and a 5-bit template. The instruction pointer cannot be directly read or written to, thus it is incremented as instructions are executed. Each time IP is incremented by 16 since each instruction bundle requires 16 bytes.
3.8 Control and Status Registers

3.8.1 Application Registers. There are 128 (64-bit) application registers from $A_{10}$ through $A_{127}$. Some of the application registers are used for compatibility with IA-32 architecture.

3.8.2 User Mask (UM) Register. The User Mask register is 6-bit register. It is used for defining data operation types such as big endian, little endian, and enabling or disabling performance monitor.

3.8.3 CPU Identification Registers. There are 5 (64-bit) registers called processor identification register or CPUID. These registers hold information about CPU such as processor ID, processor revision numbers, processor family number, and processor architecture revision number.

3.9 Register Stack

When a procedure is called, the Itanium processor makes a new set of registers available at the call site. The new sets of registers are the Register Stack. This helps to reduce the number of times the registers would have otherwise have had to be filled and emptied.
3.10 Branching and Branch Predication

In programming, conditional statements direct a program to some other part of the program. This capability to jump to another part of a program is branching.

The Itanium Architecture uses Branch Predication which involves predicting the outcome of a branch so that those instructions can be executed ahead of time. This helps to increase the overall performance and processing speed.

3.11 Register Rotation

Traditional architectures have achieved executing loop iterations by unrolling the loop and allowing the software to rename the registers.

The Itanium architecture allows the compiler to execute loop iterations in parallel rather than sequentially. It provides all iterations with their own register by renaming the registers internally. This avoids the need to unroll or rename registers programmatically. Renaming registers internally is called register rotation and it significantly reduces overhead for both large and small loops.

More detailed descriptions can be found in Intel Corporation (2002).
4. ITANIUM INSTRUCTION FORMAT

4.1 General Instruction Format

The general Itanium instruction format, which most instructions follow, is:

\[ [qp] \text{mnemonic[.comp]} \text{ dest} = \text{srcs} \]

where

- qp is the qualifying predicate register
- mnemonic is a name that uniquely identifies an IA-64 instruction
- comp is one of the instruction completers
- dest is a destination operand
- srcs is a source operand

Each Itanium instruction is 41 bits. Figure 10 displays the 41 bit format.

<table>
<thead>
<tr>
<th></th>
<th>Opcode</th>
<th>Mnemonic</th>
<th>Src2</th>
<th>Src1</th>
<th>Dest</th>
<th>qp</th>
</tr>
</thead>
</table>

*Figure 10. 41-bit Instruction Format*
4.2 Itanium's Execution Units

The following execution units can be seen in Figure 11 (Elahi, 2005).

- **Integer Unit (I-unit):** Used for execution of Integer Arithmetic, Shift, Logical Operation, and Compare and Multimedia instruction
- **Memory Unit (M-unit):** Used for load and store operations between registers and memory
- **Branch Unit (B-unit):** Used for execution of Branch Instruction
- **Loading Unit (F-unit):** Used for floating point instructions

*Figure 11. Block diagram of Itanium*
In this block, there are three branch units that are used for execution of branch instructions, four integer and memory management units. The integer unit is used for integer operation such addition, subtraction and logical operations. MMU unit is used for calculation of memory address to store and load instructions. There are two floating-point units for execution of floating point instructions. There are nine units and they can execute instructions in parallel. In order the Itanium execute instructions in parallel, the instructions should not dependent on each other and this brings us to define different types of instructions formats.

4.3 Instruction Formats

The Itanium instruction formats are divided into 6 different types as shown below in Table 3 (Elahi, 2005).

Table 3.

Itanium Instruction formats

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Integer ALU</td>
</tr>
<tr>
<td>I</td>
<td>Non ALU integer</td>
</tr>
<tr>
<td>M</td>
<td>Memory</td>
</tr>
<tr>
<td>F</td>
<td>Floating Point</td>
</tr>
<tr>
<td>B</td>
<td>Branch</td>
</tr>
<tr>
<td>L+X</td>
<td>Extended</td>
</tr>
</tbody>
</table>

Users can learn instructions of the Integer ALU, Non ALU Integer and Memory Type.
4.4 Instruction Bundles

The instructions are logically structured under their types and physically organized in bundles. Itanium groups three instructions into a bundle. Each instruction is 41 bits long and $41 \times 3 = 123$. Each bundle is however 128 bits since the template field is 5 bits. Figure 12 shows an instruction bundle.

<table>
<thead>
<tr>
<th>127</th>
<th>87</th>
<th>86</th>
<th>46</th>
<th>45</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction slot 2</td>
<td>Instruction slot 1</td>
<td>Instruction slot 0</td>
<td>Template</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 12. Instruction Bundle format

A bundle (Demone, 2001) can include instructions of two different types. The instruction fields in a bundle are called slots and are named Slot0, Slot1 and Slot2. The template field helps to encode the instruction types in the bundle. Since there are 6 different types of instructions and 3 slot positions, there are several combinations which can be encoded in the template slot. The template field is also used to determine the location of the stop, which is like flags that indicate where instructions begin or end. There are 32 combinations of values which can be stored in the template of which 24 values are defined by the IA64 and eight are intended for future use. Some IA64 bundle formats are shown in Table 4.

32
Table 4.

IA64 Bundle Formats

<table>
<thead>
<tr>
<th>Template</th>
<th>Slot0</th>
<th>Slot1</th>
<th>Slot2</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>M</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>01</td>
<td>M</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>02</td>
<td>M</td>
<td>I</td>
<td>I</td>
</tr>
</tbody>
</table>

The above bundle has two separate template codes based on the stops (represented by squares). For example: Template 01 has a stop towards the end at slot2, however template 02 has a stop before slot2. This means that incase of template 02, the CPU will execute instructions slot0 and slot1 and then execute slot2.

4.5 A Type Instruction Format

A Type instructions (Elahi, 2005) use Arithmetic Logical Unit (ALU) for their operations. Figure 13 displays the A Type instruction format

```
40-37  36  35-34  33  32-29  28-27  26-20  19-13  12-6  5-0
Op code   S  X2a  V  X4  X2b  Src2  Src1  Des  qp
```

Figure13. A Type Instruction format
In the S field, a value of 1 indicates the immediate operand. The X2a, V, X4 and X2b are Opcode extensions. The Qualifying Predicate (qp) represents the predicate bit. The X2b field changed based on the instruction. If an add instruction is used and two sources are added the X2b remains 0 but if there is 1 accompanied by the two sources, the X2b is 1.

The Src2 and Src1 are the sources used and the Des represents the destination register.

Some common A type instructions include the following:

- **Add Instruction**

  - add rdest = rsrcl, rsrc2
    
    Example: Figure 14 shows the opcode for add r1 = r2, r3

    ![Figure 14](image)

    | Op code | S | X2a | V | X4 | X2b | Src2 | Src1 | Des | qp |
    |---------|---|-----|---|----|-----|------|------|-----|----|
    | 8       | 0 | 0   | 0 | 0  | 3   | 2    | 1    | 0   | 0  |

    *Figure 14. Opcode for Add Instruction*

- add rdest = rsrcl, rsrc2, 1

  Example: Figure 15 shows the opcode for add r1 = r2, r3, 1. The X2b field changes to 1)

    ![Figure 15](image)

    | Op code | S | X2a | V | X4 | X2b | Src2 | Src1 | Des | qp |
    |---------|---|-----|---|----|-----|------|------|-----|----|
    | 8       | 0 | 0   | 0 | 0  | 1   | 3    | 2    | 1   | 0  |

    *Figure 15. A Type Add Instruction with X2b field*
Subtract Instruction

- sub rdest = rsrcl, rsrcl2

Example: Figure 16 shows the opcode for sub r1 = r2, r3

```
40-37  36 35-34  33  32-29  28-27  26-20  19-13  12-6  5-0

<table>
<thead>
<tr>
<th>Op code</th>
<th>S</th>
<th>X2a</th>
<th>V</th>
<th>X4</th>
<th>X2b</th>
<th>Src2</th>
<th>Src1</th>
<th>Des</th>
<th>qp</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
```

*Figure 16. A Type Subtract Instruction*

- sub rdest = rsrcl, rsrcl2, I

Example: Figure 17 shows the opcode for sub r1 = r2, r3, I

```
<table>
<thead>
<tr>
<th>Op code</th>
<th>S</th>
<th>X2a</th>
<th>V</th>
<th>X4</th>
<th>X2b</th>
<th>Src2</th>
<th>Src1</th>
<th>Des</th>
<th>qp</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

*Figure 17. A Type Subtract Instruction with X2b field*
The X4 and X2b opcode extensions change based on the instruction used. This can be seen below in Table 5.

**Table 5.**

*Type instruction opcodes*

<table>
<thead>
<tr>
<th>Opcode</th>
<th>S</th>
<th>X2a</th>
<th>V</th>
<th>X4</th>
<th>X2b</th>
<th>Src2</th>
<th>Src1</th>
<th>Des</th>
<th>qp</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Andcmp</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Or</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Values here depend on instruction source, destination and predicate value.

### 4.6 A2 Type Instruction Format (Elahi, 2005)

**Figure 18.** A2 Type Instruction format

In the S field of Figure 18, a value of 1 indicates the immediate operand. The X2a, V, X4 and X2b are Opcode extensions. The Qualifying Predicate (qp) represents the predicate bit. The ct2d field is the shift count. The Src2 and Src1 are the sources used and the Des represents the destination register.
Some common A type instructions include the following:

- Shift left and add Instruction

  - shladd  r1, r2, count, r3  (count is a constant value between 1 and 4)

  Shift r2 count times to the left and then add the result with r1 and place the result into r3

- Subtract immediate 8 bits

  - sub  r1 = immediate8- r3  (r1= imm8- r3)

Example: Figure 19 shows the opcode for sub r1 = immediate8- r3  (r1= imm8- r3)

```
40-37  36  35-34  33  32-29  28-27  20-20  19-13  12-6  5-0
Opcode   s X2a Ve X4 X2b src2 imm7 Des qpp
8  1  0  0  9  1 r3 imm r1
```

*Figure 19. A2 Type Subtract Instruction format*

In the S field, a value of 1 indicates the immediate operand as seen in Table 6.
Table 6.

*A2 type instruction opcodes*

<table>
<thead>
<tr>
<th>Opcode</th>
<th>S</th>
<th>X2a</th>
<th>V</th>
<th>X4</th>
<th>X2b</th>
<th>Src2</th>
<th>imm</th>
<th>Des</th>
<th>qp</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>B</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Andcmp</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>B</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Or</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>B</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>B</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Values here depend on instruction source, immediate, destination and predicate value.

➢ Integer Compare Instructions Format

- cmp.lt p1,p2= r2,r3

Example: Figure 20 shows the opcode for sub r1 = immediate8- r3 (r1= imm8- r3)

<table>
<thead>
<tr>
<th>40-37</th>
<th>36</th>
<th>35-34</th>
<th>33</th>
<th>32-27</th>
<th>26-20</th>
<th>19-13</th>
<th>12</th>
<th>11-6</th>
<th>5-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>tb</td>
<td>x2</td>
<td>ta</td>
<td>p2</td>
<td>src2</td>
<td>scr1</td>
<td>c</td>
<td>p1</td>
<td>qp</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>p2</td>
<td>r3</td>
<td>r2</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 20. A2 Type Integer Compare Instruction format*
Other types of Integer Compare instruction formats are described in Table 7.

Table 7.

*Integer Compare Instruction format*

<table>
<thead>
<tr>
<th>Cmp.ltu</th>
<th>D</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>P2</th>
<th>R3</th>
<th>R2</th>
<th>0</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cmp.eq</td>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>P2</td>
<td>R3</td>
<td>R2</td>
<td>0</td>
<td>P1</td>
</tr>
<tr>
<td>Cmp.lt.unc</td>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>P2</td>
<td>R3</td>
<td>R2</td>
<td>1</td>
<td>P1</td>
</tr>
<tr>
<td>Cmp.ltu.unc</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>P2</td>
<td>R3</td>
<td>R2</td>
<td>1</td>
<td>P1</td>
</tr>
<tr>
<td>Cmp.eq.unc</td>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>P2</td>
<td>R3</td>
<td>R2</td>
<td>0</td>
<td>P1</td>
</tr>
<tr>
<td>Cmp.eq.and</td>
<td>C</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>P2</td>
<td>R3</td>
<td>R2</td>
<td>0</td>
<td>P1</td>
</tr>
<tr>
<td>Cmp.eq.or</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>P2</td>
<td>R3</td>
<td>R2</td>
<td>0</td>
<td>P1</td>
</tr>
<tr>
<td>Cmp.eq.or.andcm</td>
<td>E</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>P2</td>
<td>R3</td>
<td>R2</td>
<td>0</td>
<td>P1</td>
</tr>
</tbody>
</table>

More detailed descriptions can be found in Elahi (2005).
5. ITANIUM INSTRUCTIONS

5.1 Add Instruction (Add)

'Add' is an Integer type of instruction which adds the contents of registers from the sources and puts them in the destination register.

For example:

\[(qp)\ add\ r1 = r2, r3\]

Here the source registers r2 and r3's contents are added and the results are put in r1, the destination register. If the predicate register's contents or qp is equal to one, the instruction will execute and if the predicate register’s contents are equal to zero, the instruction will not execute. The qp can be any bit in the predicate register.

5.2 Subtraction Instruction (Sub)

'Sub' is an Integer type of instruction which subtracts the contents of registers from the sources and puts them in the destination register.
For example:

\[(qp) \text{ sub} \ r1 = r2, r3\]

Here the source registers r2 and r3’s contents are subtracted and the results are put in r1, the destination register. If the predicate register’s contents are equal to one, the instruction will execute and if the predicate register’s contents are equal to zero, the instruction will not execute. The qp can be any bit in the predicate register.

5.3 Logical AND Instruction (And)

‘And’ is an Integer type of instruction which performs a logical and operation on the contents of registers from the sources and puts them in the destination register.

For example:

\[(qp) \text{ and} \ r1 = r2, r3\]

Here the source registers r2 and r3’s contents are logically added (bitwise operations) and the results are put in r1, the destination register. If the predicate register’s contents are equal to one, the instruction will execute and if the predicate register’s contents are equal to zero, the instruction will not execute. The qp can be any bit in the predicate register.
5.4 AND Complement Instruction (Andcm)

‘Andcm’ is an Integer type of instruction which performs an and complement operation on the contents of registers from the sources and puts them in the destination register.

For example:

\[(qp) \text{ andcm } r1 = r2, r3\]

Here the source registers r2 and r3’s contents are logically added (bitwise operations) and a complement of r3 is taken. The results are put in r1, the destination register. Only if the predicate register’s contents are equal to one, the instruction will execute. The qp can be any bit in the predicate register.

5.5 Exclusive OR Instruction (Or)

‘Or’ is an Integer type of instruction which performs a logical or bitwise OR operation on the contents of registers from the sources and puts them in the destination register.

For example:

\[(qp) \text{ xor } r1 = r2, r3\]

Here a bitwise OR is performed on the source registers r2 and r3’s and the results are put in r1, the destination register. If the predicate register’s contents are equal to one, the instruction
will execute and if the predicate register’s contents are equal to zero, the instruction will not execute. The \( qp \) can be any bit in the predicate register.

5.6 Move Instruction (Mov)

'Mov' is an Non Integer type of instruction which moves the contents of sources register and puts them in the destination registers

For example:

\[(qp) \text{mov } r1 = r2\]

Here the contents of source registers \( r2 \) are taken and moved to the destination register \( r1 \). If the predicate register’s contents are equal to one, the instruction will execute and if the predicate register’s contents are equal to zero, the instruction will not execute. The \( qp \) can be any bit in the predicate register.

5.7 Shift Left Instruction (Shl)

'Shl' is an Integer type of instruction which performs a logical shift left operation.

For example:

\[(qp) \text{Shl } r1 = r2, r\]
Here the source registers r3 defines, the number of times that r2 must be shifted to the left. The max value of r3 can be 63. If the predicate register’s contents are equal to one, the instruction will execute and if the predicate register’s contents are equal to zero, the instruction will not execute. The qp can be any bit in the predicate register.

5.8 Shift Right Instruction (Shr)

‘Shr’ is an Integer type of instruction which performs a logical shift left operation.

For example:

\[(qp) Shr r1 = r2, r3\]

Here the source registers r3 defines, the number of times that r2 must be shifted to the right. The max value of r3 can be 63. If the predicate register’s contents are equal to one, the instruction will execute and if the predicate register’s contents are equal to zero, the instruction will not execute. The qp can be any bit in the predicate register.
5.9 Load Instruction \((Ld)\)

‘Ld’ is a Memory type of instruction which performs loads the register with data from the memory.

For example:

\[(qp) \; Ld \; r1 = \{r1\}\]

Here \(r1\), the destination register loads itself with the value that is got from the memory location \(r1\). If the predicate register’s contents are equal to one, the instruction will execute and if the predicate register’s contents are equal to zero, the instruction will not execute. The \(qp\) can be any bit in the predicate register.

5.10 Store Instruction \((St)\)

‘St’ is a Memory type of instruction which stores the data from the register to the memory.

For example:

\[(qp) \; St \; \{r1\} = r1\]

Here \(r1\), the destination is the memory location \(r1\). The value that is got from the general register 1 is stored in the memory. If the predicate register’s contents are equal to one, the
instruction will execute and if the predicate register’s contents are equal to zero, the
instruction will not execute. The qp can be any bit in the predicate register.

5.11 Compare Instruction (Cmp)

There are three types of compare instructions the normal compare, the unconditional compare,
and the parallel compare. They are described in detail below.

• The Normal Compare Instruction

5.11.1 Compare equals (Cmp.eq). For example: (qp) cmp.eq p1,p2=r3,r4. If the condition
true i.e. if the value on register r3 is equal to the value in r4, then p1 is set to one and p2 is set
to zero. If however, the condition false and r3 is not equal to r4, p1 is set to zero and p2 is set
to one. If the predicate register’s contents are equal to one, the instruction will execute and if
the predicate register’s contents are equal to zero, the instruction will not execute. The qp can
be any bit in the predicate register.

5.11.2 Compare less than (Cmp.lt). For example: (qp) cmp.lt p1,p2=r3,r4. If the condition
true i.e. if the value on register r3 is less than the value in r4, then p1 is set to one and p2 is set
to zero. If however, the condition false and r3 is greater than r4, p1 is set to zero and p2 is set
to one. If the predicate register’s contents are equal to one, the instruction will execute and if
the predicate register’s contents are equal to zero, the instruction will not execute. The qp can
be any bit in the predicate register.
5.11.3 Compare greater than (Cmp.gt). For example: (qp) cmp.gt p1,p2=r3,r4. If the condition true i.e. if the value on register r3 is greater than the value in r4, then p1 is set to one and p2 is set to zero. If however, the condition false and r3 is lesser than r4, p1 is set to zero and p2 is set to one. If the predicate register’s contents are equal to one, the instruction will execute and if the predicate register’s contents are equal to zero, the instruction will not execute. The qp can be any bit in the predicate register.

- The Unconditional Compare Instruction

5.11.4 Unconditional Compare equals (Cmp.eq.unc). For example: (qp) cmp.eq.unc p1,p2=r3,r4. If the qp bit is zero, then p1 and p2 are set to zero. If qp bit is 1, then the cmp instruction is executed like a normal instruction. So if the qp bit is one and the condition is true i.e. if the value on register r3 is equal to the value in r4, then p1 is set to one and p2 is set to zero. If however, the condition false and r3 is not equal to r4, p1 is set to zero and p2 is set to one. The qp can be any bit in the predicate register.

5.11.5 Unconditional Compare less than (Cmp.lt.unc). For example: (qp) cmp.eq.lt p1,p2=r3,r4. If the qp bit is zero, then p1 and p2 are set to zero. If qp bit is 1, then the cmp instruction is executed like a normal instruction. So if the qp bit is one and the condition is true i.e. if the value on register r3 is less than the value in r4, then p1 is set to one and p2 is set to zero. If however, the condition false and r3 greater than r4, p1 is set to zero and p2 is set to one. The qp can be any bit in the predicate register.

47
5.11.6 Unconditional Compare greater than (Cmp.gt.unc). For example: (qp) cmp.eq.gt p1, p2=r3, r4. If the qp bit is zero, then p1 and p2 are set to zero. If qp bit is 1, then the cmp instruction is executed like a normal instruction. So if the qp bit is one and the condition is true i.e. if the value on register r3 is greater than the value in r4, then p1 is set to one and p2 is set to zero. If however, the condition false and r3 lesser than r4, p1 is set to zero and p2 is set to 1. The qp can be any bit in the predicate register.

Figure 21. Unconditional Compare along with Itanium instructions
The program below is described in the Figure 21

(p0) cmp.gt p1,p2 = r1,r2

(p1) add r3 = r3,2

(p2) add r4 = r4+5

(p2) cmp.eq.un p3,p4 = r3,r7

(p3) sub r6 = r6-r7

(p4) add r6 = r6+r7

5.12 Parallel Compare

Parallel compare uses OR, AND, ANDOR and complement operations.

5.12.1 Compare equal or. For example: cmp.eq.or p1, p2 = r32,r34. If the result of comparison is false, it will set the value of P1 and P2 to 1. If result of comparison is true, then values of P1 and P2 remain unchanged.

5.12.2 Compare equal and. For example: cmp.eq.and p1,p2 = r3,r5. P1 and P2 will set to zero if result of comparison true, if result of comparison be false the P1 and P2 do not change.

5.12.3 Compare equal or. For example: cmp.eq andor p1,p2 = r2,r3. If the condition is true the value of P1 and P2 do not change, if condition false the value of P1=1 and P2=0.
5.13 Comparison of Itanium and Pentium Instructions

*Figure 22. Pentium program’s flow chart*

Figure 22 (Elahi, 2005) describes the Pentium program’s flow chart and Table 8 describes the Pentium Instructions. Assume that AX = A, BX = B and CX = C in Table 8.
Table 8.

**Pentium Program and Description**

<table>
<thead>
<tr>
<th>Pentium Program</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP AX, 0</td>
<td>Compare AX with 0</td>
</tr>
<tr>
<td>JE L1</td>
<td>If equal Jump to L1</td>
</tr>
<tr>
<td>CMP BX,0</td>
<td>Compare BX with 0</td>
</tr>
<tr>
<td>JE L1</td>
<td>If equal Jump to L1</td>
</tr>
<tr>
<td>J=J+1</td>
<td>Add 1 to J</td>
</tr>
<tr>
<td>JMP L3</td>
<td>Jump to L3</td>
</tr>
<tr>
<td>L1: CMP CX,0</td>
<td>L1: Compare CX with 0</td>
</tr>
<tr>
<td>JE L2</td>
<td>If equal Jump to L2</td>
</tr>
<tr>
<td>K=K+1</td>
<td>Add 1 to K</td>
</tr>
<tr>
<td>JMP L3</td>
<td>Jump to L3</td>
</tr>
<tr>
<td>L2: K=K-1</td>
<td>L2: Subtract 1 from K</td>
</tr>
<tr>
<td>L3: I=I+1</td>
<td>L3: Add 1 to I</td>
</tr>
</tbody>
</table>

Table 9 describes the Itanium instructions for the program described in Figure 22.

Table 9.

**Itanium Program and Description**

<table>
<thead>
<tr>
<th>Itanium Program</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. (P0) CMP.EQ P1, P2=0, R1</td>
<td>1. Compare R1 with 0 if equal then set P1 to true and P2 to false otherwise set P1 to false and P2 to true.</td>
</tr>
<tr>
<td>2. (P2) CMP.EQ P1,P3 =0,R2</td>
<td>2. Compare R2 with 0 if equal then set P1 to true and P3 to false otherwise set P1 to false and P3 to true.</td>
</tr>
<tr>
<td>3. (P3) ADD J=J+1</td>
<td>3. If P3 is true, Add J = J +1</td>
</tr>
<tr>
<td>4. (P1) CMP.NE P4,P5 =0,R3</td>
<td>4. Compare R3 with 0 if not equal then set P4 to true and P5 to false otherwise set P4 to false and P5 to true.</td>
</tr>
<tr>
<td>5. (P4) K=K+1</td>
<td>5. If P4 is true, Add K = K +1</td>
</tr>
<tr>
<td>6. (P5) K=K-1</td>
<td>6. If P5 is true, Add K = K -1</td>
</tr>
<tr>
<td>7. (P0) I=I+1</td>
<td>7. If P0 is always true, Add I = I +1</td>
</tr>
</tbody>
</table>

51
6. ITANIUM SIMULATORS

The IA-64 architecture is gaining in its popularity and it is being used in the capacity of a server by many companies. Limited number of resources and applications are available which simulate the IA-64 (Itanium Processor Family, IPF) instruction set.

6.1 HP SKI Simulator

HP, who is developed the Itanium 64 bit processor, also has a proprietary instruction-set simulator for the IA-64 architecture for application- and kernel-level software development. This simulator is known as SKI. The simulator is designed to work like an application on the Linux operating system.

The software works in two ways:

- One way to run the software is to have a compiled IA-64 program (a program compiled with the Itanium machine). This program is then transferred to a 32 bit machine running the Linux Operating System. SKI understands this Itanium compiled program on Linux, interprets it and displays the contents of registers and memory. It helps uses to see the
internal workings of the Itanium architecture. Currently most users use Windows and Itanium machines are not yet easily accessible to users.

- The other option to use SKI without access to an Itanium machine is to download a Linux kernel that can be booted with SKI. This process requires advanced knowledge of the workings of the Linux Operating System.

The learning curve for understanding the SKI software and running it is very high and not easy for beginner students or users.

6.2 Itanium Instruction Set Simulator

The Itanium Instruction Set Simulator is a customized unique solution which has been designed especially to work on Windows IA-32 users. It can however also be accessed on any operating system, since can be run using a Web Browser.

It is an extremely user-friendly and intuitive GUI environment. The simulator would really help students and users who want to learn IA-64 assembly language code and run it. The Itanium Instruction Set-Simulator displays results of the program and the contents of the Registers and Memory. There is a complete log of information which displays step by step, mentioning which instruction lead to what result.
The simulator has been created in the Java programming language. The Java 2 Enterprise Edition (J2EE) is also used to create the web interface. Extensible markup language (XML) has been used to manage information and content. Various websites and books are available that would help us in understanding the Java programming language. Sun Microsystems provides us with in depth technical details of the Java programming language as well as J2EE. Detailed information about XML is also provided by the World Wide Web Consortium (W3C).
7. ITANIUM INSTRUCTION-SET SIMULATOR (IISS) SOFTWARE

The software is a unique and one of a kind solution which can be used by users to learn the Itanium instructions and write assembly language programs.

7.1 The Main Components of the software

- The Data Window which allows users to enter the data (numbers) that will be loaded in to the memory when the program runs

- The Program Window which is used to enter the Itanium instructions which the user wants to simulate

- The Results Window which informs the user about the values in the General Registers, the data which was loaded in memory, details on the instruction executed. It also informs users about the results of those instructions as well as splits the instruction in to Qualifying Predicate, mnemonic, completer1, completer2, destination and sources

55
7.2 Registers and Memory

The 128 General Registers, the 64 Predicate Registers and the 1000 Memory locations are simulated using Java Hash Maps or data structures.

The Registers and Memory implement the singleton pattern which means that they are serve as a single global repository for the entire application, until a new set of instructions is executed.

7.3 The Program Controller

The Controller is the heart of the application. It directs the program to the appropriate instruction parsers and instruction executers after recognizing what kind of instruction the user has entered.
Figure 23. The Program Controller

Figure 23 demonstrates the functionality of the program controller. For instance, if the user enters an instruction, the controller manages the following:
• The controller first sends the instruction line to an instruction reader which tells it what kind of instruction the user has entered, based on information in an xml file. The xml file has information on the name of the instruction format (like Ld, Add, etc.), its type (M for Memory, I for Integer ALU, etc.), its description, its opcode and also the class name of the parser.

• Once the controller gets the instruction type, it calls the appropriate parser to separate the instruction which returns a Hash Map data structure back to the controller containing the separated values of the qualifying predicate, the mnemonic, the completers, the sources and the destination. Note that there are 3 different parsers (Integer ALU, Non Integer ALU and Memory) which specifically handle their own instruction type.

• The controller then based on the Qualifying Predicate decides if that instruction should be sent to the relevant executer program. If the value in the Qualifying Predicate points to a specific Predicate Register. The controller looks up the predicate register and if the predicate register is 1 or true, the instruction is sent for execution. If the value of the predicate register is 0 or false, a message is sent back to the Output Window specifying that the instruction could not be executed because the value in the predicate register was false.

• If the instruction is meant to be executed, the appropriate executer is called. Note that there are 3 different executers (Integer ALU, Non Integer ALU and Memory) for the 3
types of instructions the Itanium Instruction Set Simulator (IISS) software is capable of handling.

7.4 *The Instruction Parsers*

There are 3 different Instruction Parsers and they perform the task of separating the instruction from a single line to a data structure which contains the Qualifying Predicate, the mnemonic, the completers (completer 1 and completer 2), the sources, the mnemonic and the destinations.

If the user enters a memory instruction type, the memory parser takes up the responsibility to separate the instruction. If an integer instruction type is entered, the integer parser takes over the separation task and finally if a non integer type is entered, the non integer parser handles the separation task.

More parsers can be easily added, based on the user’s needs and preferences. The addition of parser requires the user to code an additional Java parser class. This class can be then easily added and used within the project.
7.5 The Instruction Executors

There are 3 different Instruction Executers which carry out the execution. Once the controller delegates the task to an executer, it takes a Hash Map data structure containing the Qualifying Predicate, the mnemonic, the completers (completer 1 and completer 2), the sources and the destinations and executes the instruction.

Since there are several instructions which can be executed by a single executer, the executer decides to execute the instruction, based on the mnemonic provided by the controller.

If the instruction is of type Add, the controller would have given it to the Integer executer. The Integer executer can handle several instructions such as the Add, Subtract, And, OR, XOR, AND complement, Shift left, Shift right, Compare (equals, less than, greater than). The executer looks up the mnemonic the Hash Map provided to it and then recognizes that the user wishes to execute an Add instruction. It then gets the general register information from the sources, looks up the general registers, performs the Add operation. It also gets the destination general register information and sets the added value to it.

More executers can be easily added, based on the user’s needs and preferences. The addition of executer requires the user to code an additional Java executer class. This class can be then easily added and used within the project. Additional instructions can be added
for the executers by simply adding the instruction information in the instruction formats xml file and adding the instruction's executing code in the executor.

Screen shots of the Itanium Instruction Set Simulator can be seen in Figures 24, 25 and 26.

Figure 24. The Program Window
Figure 25. The Data Window
<table>
<thead>
<tr>
<th>GENERAL REGISTERS</th>
<th>MEMORY</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0 → 0000000 (0)</td>
<td>M0 → 1010 (10)</td>
<td>Initializing Registers</td>
</tr>
<tr>
<td>R1 → 10100 (20)</td>
<td>M1 → 10100 (20)</td>
<td>Initializing Instruction Executers</td>
</tr>
<tr>
<td>R2 → 11110 (30)</td>
<td>M2 → 11110 (30)</td>
<td>Reading data from file</td>
</tr>
<tr>
<td>R3 → 110010 (50)</td>
<td>M3 → 101000 (40)</td>
<td>Adding 1010(10) to memory location 0</td>
</tr>
<tr>
<td>R4 → 10100 (20)</td>
<td>M4 → 110010 (50)</td>
<td>Adding 10100(20) to memory location 1</td>
</tr>
<tr>
<td>R5 → 0000000 (0)</td>
<td>M5 → 111100 (50)</td>
<td>Adding 11110(30) to memory location 2</td>
</tr>
<tr>
<td>R6 → 0000000 (0)</td>
<td>M6 → 1000110 (70)</td>
<td>Adding 10001(40) to memory location 3</td>
</tr>
<tr>
<td>R7 → 0000000 (0)</td>
<td>M7 → 11000100000 (800)</td>
<td>Adding 11001(30) to memory location 4</td>
</tr>
<tr>
<td>R8 → 0000000 (0)</td>
<td>M8 → 1110000100 (900)</td>
<td>Adding 11110(60) to memory location 5</td>
</tr>
<tr>
<td>R9 → 0000000 (0)</td>
<td>M9 → 1100100 (100)</td>
<td>Adding 10001(70) to memory location 6</td>
</tr>
<tr>
<td>R10 → 0000000 (0)</td>
<td>M10 → 1100100 (100)</td>
<td>Adding 1100100000(800) to memory location 7</td>
</tr>
<tr>
<td>R11 → 0000000 (0)</td>
<td>M11 → 1100100 (100)</td>
<td>Adding 1100100000(800) to memory location 8</td>
</tr>
<tr>
<td>R12 → 0000000 (0)</td>
<td>M12 → 1100100 (100)</td>
<td>Adding 1100100000(800) to memory location 9</td>
</tr>
<tr>
<td>R13 → 0000000 (0)</td>
<td>M13 → 1100100 (100)</td>
<td>Adding 1100100000(800) to memory location 10</td>
</tr>
<tr>
<td>R14 → 0000000 (0)</td>
<td>M14 → 1100100 (100)</td>
<td>Adding 1100100000(800) to memory location 11</td>
</tr>
<tr>
<td>R15 → 0000000 (0)</td>
<td>M15 → 1100100 (100)</td>
<td>Adding 1100100000(800) to memory location 12</td>
</tr>
<tr>
<td>R16 → 0000000 (0)</td>
<td>M16 → 1100100 (100)</td>
<td>Adding 1100100000(800) to memory location 13</td>
</tr>
<tr>
<td>R17 → 0000000 (0)</td>
<td>M17 → 1100100 (100)</td>
<td>Adding 1100100000(800) to memory location 14</td>
</tr>
<tr>
<td>R18 → 0000000 (0)</td>
<td>M18 → 1100100 (100)</td>
<td>Adding 1100100000(800) to memory location 15</td>
</tr>
<tr>
<td>R19 → 0000000 (0)</td>
<td>M19 → 1100100 (100)</td>
<td>Adding 1100100000(800) to memory location 16</td>
</tr>
<tr>
<td>R20 → 0000000 (0)</td>
<td>M20 → 1100100 (100)</td>
<td>Adding 1100100000(800) to memory location 17</td>
</tr>
<tr>
<td>R21 → 0000000 (0)</td>
<td>M21 → 0000000 (0)</td>
<td>Adding 1100100000(800) to memory location 18</td>
</tr>
<tr>
<td>R22 → 0000000 (0)</td>
<td>M22 → 0000000 (0)</td>
<td>Adding 1100100000(800) to memory location 19</td>
</tr>
<tr>
<td>R23 → 0000000 (0)</td>
<td>M23 → 0000000 (0)</td>
<td>Adding 1100100000(800) to memory location 20</td>
</tr>
<tr>
<td>R24 → 0000000 (0)</td>
<td>M24 → 0000000 (0)</td>
<td>Finished reading data from file</td>
</tr>
<tr>
<td>R25 → 0000000 (0)</td>
<td>M25 → 0000000 (0)</td>
<td>Executing 1 line from the file:</td>
</tr>
<tr>
<td>R26 → 0000000 (0)</td>
<td>M26 → 0000000 (0)</td>
<td>(PO) Ld r1 = r1</td>
</tr>
<tr>
<td>R27 → 0000000 (0)</td>
<td>M27 → 0000000 (0)</td>
<td>Instruction type is Memory Instruction Type</td>
</tr>
<tr>
<td>R28 → 0000000 (0)</td>
<td>M28 → 0000000 (0)</td>
<td>op = P0</td>
</tr>
<tr>
<td>R29 → 0000000 (0)</td>
<td>M29 → 0000000 (0)</td>
<td>mnemonics = (integer ALU) -&gt; Ld</td>
</tr>
<tr>
<td>R30 → 0000000 (0)</td>
<td>M30 → 0000000 (0)</td>
<td>computer1 =</td>
</tr>
<tr>
<td>R31 → 0000000 (0)</td>
<td>M31 → 0000000 (0)</td>
<td>computer2 =</td>
</tr>
<tr>
<td>R32 → 0000000 (0)</td>
<td>M32 → 0000000 (0)</td>
<td>dest = r1</td>
</tr>
<tr>
<td>R33 → 0000000 (0)</td>
<td>M33 → 0000000 (0)</td>
<td>sources = [r1]</td>
</tr>
<tr>
<td>R34 → 0000000 (0)</td>
<td>M34 → 0000000 (0)</td>
<td>Loading 20 which was in memory location r1 to general register r1</td>
</tr>
<tr>
<td>R35 → 0000000 (0)</td>
<td>M35 → 0000000 (0)</td>
<td>Executing 2 line from the file:</td>
</tr>
<tr>
<td>R36 → 0000000 (0)</td>
<td>M36 → 0000000 (0)</td>
<td>(PO) Ld r2 = r2</td>
</tr>
<tr>
<td>R37 → 0000000 (0)</td>
<td>M37 → 0000000 (0)</td>
<td>Instruction type is Memory Instruction Type</td>
</tr>
<tr>
<td>R38 → 0000000 (0)</td>
<td>M38 → 0000000 (0)</td>
<td>op = P0</td>
</tr>
<tr>
<td>R39 → 0000000 (0)</td>
<td>M39 → 0000000 (0)</td>
<td>mnemonics = (integer ALU) -&gt; Ld</td>
</tr>
<tr>
<td>R40 → 0000000 (0)</td>
<td>M40 → 0000000 (0)</td>
<td>computer1 =</td>
</tr>
<tr>
<td>R41 → 0000000 (0)</td>
<td>M41 → 0000000 (0)</td>
<td>computer2 =</td>
</tr>
<tr>
<td>R42 → 0000000 (0)</td>
<td>M42 → 0000000 (0)</td>
<td>dest = r2</td>
</tr>
<tr>
<td>R43 → 0000000 (0)</td>
<td>M43 → 0000000 (0)</td>
<td>sources = [r2]</td>
</tr>
<tr>
<td>R44 → 0000000 (0)</td>
<td>M44 → 0000000 (0)</td>
<td>Loading 30 which was in memory location r2 to general register r2</td>
</tr>
</tbody>
</table>

**Figure 26. The Result Window**

63
8. ITANIUM INSTRUCTION SET SIMULATOR (IISS) USER GUIDE

8.1 Requirements

To run the Itanium Instruction Set Simulator you will need the following:

- Windows based operating system such as Windows NT, ME, XP, 2000, 2003 etc.
- Java 2 Platform, Standard Edition (J2SE 1.4) which can be downloaded from  
  http://java.sun.com/j2se/index.jsp
- Java 2 Platform, Enterprise Edition (J2EE 1.3) which can be downloaded from  
  http://java.sun.com/j2ee/index.jsp
- Eclipse Toolset which can be downloaded from  
- JDOM jar files which are used to read XML files can be downloaded from  
  http://jdom.org/downloads/index.html
- Jakarta Commons which is used by the Itanium Instruction Set Simulator (IISS) parser  
  can be downloaded from http://jakarta.apache.org/commons/lang/
- The Itanium Instruction Set Simulator jar file
8.2 Setting up your Environment

The following steps must be taken to setup your Windows environment to run the software.

- Put all the JDOM jar files and Jakarta jar files in a folder called c:\jarsNeeded.

- Setting up environment variables
  - Set Path to C:\java\j2sdk1.4.2\bin; C:\jarsNeeded;
  - Set Classpath to C:\java\j2sdk1.4.2\lib;C:\java\j2sdkkee1.3.1\lib; (Please note that if your JDK is placed in a different directory, you must point to that)
  - Create a System Variable named J2EE_HOME and set it to c:\java\j2sdkkee1.3.1
  - Create a System Variable named JAVA_HOME and set it to C:\java\j2sdk1.4.2

65
8.3 Running the Simulator

- Follow the steps below to run the Itanium Instruction Set-Simulator
- Start the Eclipse Toolkit and create a workbench for example Testing Itanium Simulator
- Create a java project within Eclipse and add the libraries – Add jars from C:\jarsNeeded
- Import the ItaniumSimulator.jar file into the project (Note sometimes after adding the jar files, you might see some warnings. You can ignore them or accept suggestions made.). The ItaniumSimulator.jar is made available by the author.
- You can then select the Start.java file and run the program
- You need to place the data.txt file in c:\. The data is the data you want in memory
- You also need to place the instruction file in c:\Itanium-instructions.txt
- You can test the program by changing the data.txt file and Itanium-instructions.txt file.
9. SAMPLE PROGRAMS

9.1 Program 1

```
(P0) Ld r1 = [r10]
(P0) Ld r2 = [r20]
(P0) Add r3=r2,r1
(P0) Cmp.eq p1,p2 = r1,r2
(P1) Sub r4 = r2,r1
(P2) And r4 = r2,r1
```

For your reference this code is kept in your C:\itanium-instructions.txt

Figure 27. Program 1 Instructions
Figure 28. Program 1 Memory

<table>
<thead>
<tr>
<th>GENERAL REGISTERS</th>
<th>MEMORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0 --&gt; 0000000 (0)</td>
<td>M0 --&gt; 1 (1)</td>
</tr>
<tr>
<td>R1 --&gt; 1100100 (100)</td>
<td>M1 --&gt; 1010 (10)</td>
</tr>
<tr>
<td>R2 --&gt; 1100100 (100)</td>
<td>M2 --&gt; 10100 (20)</td>
</tr>
<tr>
<td>R3 --&gt; 11001000 (200)</td>
<td>M3 --&gt; 11110 (30)</td>
</tr>
<tr>
<td>R4 --&gt; 0000000 (0)</td>
<td>M4 --&gt; 101000 (40)</td>
</tr>
<tr>
<td>R5 --&gt; 0000000 (0)</td>
<td>M5 --&gt; 110010 (50)</td>
</tr>
<tr>
<td>R6 --&gt; 0000000 (0)</td>
<td>M6 --&gt; 111100 (60)</td>
</tr>
<tr>
<td>R7 --&gt; 0000000 (0)</td>
<td>M7 --&gt; 1000110 (70)</td>
</tr>
<tr>
<td>R8 --&gt; 0000000 (0)</td>
<td>M8 --&gt; 11001000 (800)</td>
</tr>
<tr>
<td>R9 --&gt; 0000000 (0)</td>
<td>M9 --&gt; 111000100 (900)</td>
</tr>
<tr>
<td>R10 --&gt; 0000000 (0)</td>
<td>M10 --&gt; 1100100 (100)</td>
</tr>
<tr>
<td>R11 --&gt; 0000000 (0)</td>
<td>M11 --&gt; 1100100 (100)</td>
</tr>
<tr>
<td>R12 --&gt; 0000000 (0)</td>
<td>M12 --&gt; 1100100 (100)</td>
</tr>
<tr>
<td>R13 --&gt; 0000000 (0)</td>
<td>M13 --&gt; 1100100 (100)</td>
</tr>
<tr>
<td>R14 --&gt; 0000000 (0)</td>
<td>M14 --&gt; 1100100 (100)</td>
</tr>
<tr>
<td>R15 --&gt; 0000000 (0)</td>
<td>M15 --&gt; 1100100 (100)</td>
</tr>
<tr>
<td>R16 --&gt; 0000000 (0)</td>
<td>M16 --&gt; 1100100 (100)</td>
</tr>
<tr>
<td>R17 --&gt; 0000000 (0)</td>
<td>M17 --&gt; 1100100 (100)</td>
</tr>
<tr>
<td>R18 --&gt; 0000000 (0)</td>
<td>M18 --&gt; 1100100 (100)</td>
</tr>
<tr>
<td>R19 --&gt; 0000000 (0)</td>
<td>M19 --&gt; 1100100 (100)</td>
</tr>
<tr>
<td>R20 --&gt; 0000000 (0)</td>
<td>M20 --&gt; 1100100 (100)</td>
</tr>
</tbody>
</table>

Figure 29. Program 1 Results (Registers and Memory)
<table>
<thead>
<tr>
<th>OUTPUT</th>
<th>Executing 3 line from the file:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initializing Registers</td>
<td>(P0) Add r3=r2,r1</td>
</tr>
<tr>
<td>Initializing Instruction Executors</td>
<td>Instruction Type was Integer ALU</td>
</tr>
<tr>
<td>Reading memory data from file ...</td>
<td>ap = P0</td>
</tr>
<tr>
<td>Adding 1(1) to memory location 0</td>
<td>mnemonic = (IntegerALU)--- &gt;Add</td>
</tr>
<tr>
<td>Adding 101(10) to memory location 1</td>
<td>completer1 =</td>
</tr>
<tr>
<td>Adding 10100(20) to memory location 2</td>
<td>completer2 =</td>
</tr>
<tr>
<td>Adding 11110(30) to memory location 3</td>
<td>dest = r3</td>
</tr>
<tr>
<td>Adding 101000(40) to memory location 4</td>
<td>sources = r2</td>
</tr>
<tr>
<td>Adding 110010(50) to memory location 5</td>
<td>sources = r1</td>
</tr>
<tr>
<td>Adding 111100(60) to memory location 6</td>
<td>41 Bit INSTRUCTION / OPCODE IS 8000002130</td>
</tr>
<tr>
<td>Adding 1000110(70) to memory location 7</td>
<td>Destination r3 = 11001000(200)</td>
</tr>
<tr>
<td>Adding 1100110000000(800) to memory location 8</td>
<td></td>
</tr>
<tr>
<td>Adding 11100001000(900) to memory location 9</td>
<td>Executing 4 line from the file:</td>
</tr>
<tr>
<td>Adding 11001000(100) to memory location 10</td>
<td>(P0) Cmp.eq p1,p2 = r1,r2</td>
</tr>
<tr>
<td>Adding 1100100(100) to memory location 11</td>
<td>Instruction Type was Integer ALU</td>
</tr>
<tr>
<td>Adding 1100100(100) to memory location 12</td>
<td>ap = P0</td>
</tr>
<tr>
<td>Adding 1100100(100) to memory location 13</td>
<td>mnemonic = (IntegerALU)--- &gt;Cmp</td>
</tr>
<tr>
<td>Adding 1100100(100) to memory location 14</td>
<td>completer1 = eq</td>
</tr>
<tr>
<td>Adding 1100100(100) to memory location 15</td>
<td>completer2 =</td>
</tr>
<tr>
<td>Adding 1100100(100) to memory location 16</td>
<td>dest = p1,p2</td>
</tr>
<tr>
<td>Adding 1100100(100) to memory location 17</td>
<td>sources = r1</td>
</tr>
<tr>
<td>Adding 1100100(100) to memory location 18</td>
<td>sources = r2</td>
</tr>
<tr>
<td>Adding 1100100(100) to memory location 19</td>
<td>Predicate register p1 is set to true</td>
</tr>
<tr>
<td>Adding 1100100(100) to memory location 20</td>
<td>Predicate register p2 is set to false</td>
</tr>
<tr>
<td>Executing 5 line from the file:</td>
<td></td>
</tr>
<tr>
<td>Executing 6 line from the file:</td>
<td></td>
</tr>
<tr>
<td>Executing 1 line from the file:</td>
<td></td>
</tr>
<tr>
<td>(P0) Ld r1 = [r10]</td>
<td></td>
</tr>
<tr>
<td>Instruction type is Memory Instruction Type</td>
<td></td>
</tr>
<tr>
<td>ap = P0</td>
<td></td>
</tr>
<tr>
<td>mnemonic = (Memory type Instruction)--- &gt;Ld</td>
<td></td>
</tr>
<tr>
<td>completer1 =</td>
<td></td>
</tr>
<tr>
<td>completer2 =</td>
<td></td>
</tr>
<tr>
<td>dest = r1</td>
<td></td>
</tr>
<tr>
<td>sources = [r10]</td>
<td></td>
</tr>
<tr>
<td>Loading 1100100 (100) which was in memory location r10 to general</td>
<td></td>
</tr>
<tr>
<td>register r1</td>
<td></td>
</tr>
<tr>
<td>Executing 2 line from the file:</td>
<td></td>
</tr>
<tr>
<td>(P0) Ld r2 = [r20]</td>
<td></td>
</tr>
<tr>
<td>Instruction type is Memory Instruction Type</td>
<td></td>
</tr>
<tr>
<td>ap = P0</td>
<td></td>
</tr>
<tr>
<td>mnemonic = (Memory type Instruction)--- &gt;Ld</td>
<td></td>
</tr>
<tr>
<td>completer1 =</td>
<td></td>
</tr>
<tr>
<td>completer2 =</td>
<td></td>
</tr>
<tr>
<td>dest = r2</td>
<td></td>
</tr>
<tr>
<td>sources = [r20]</td>
<td></td>
</tr>
<tr>
<td>Loading 1100100 (100) which was in memory location r20 to general</td>
<td></td>
</tr>
<tr>
<td>register r2</td>
<td></td>
</tr>
</tbody>
</table>

Figure 30. Program 1 Results (Output)
Explanation:

- (P0) Ld r1 = [r10]
  - P(0) is always 1 (true) and so this instruction executes. It loads 100 which was in memory location r10 to general register r1

- (P0) Ld r2 = [r20]
  - P(0) is 1 (true) again and so this instruction executes. It loads 100 which was in memory location r20 to general register r2

- (P0) Add r3=r2, r1
  - P(0) is 1 (true) again and so this instruction executes. It adds the contents of general registers r2 and r1 and puts it in general register r3.
  - r3 is 300 now

- (P0) Cmp.eq p1,p2 = r1, r2
  - P(0) is 1 (true) again and so this instruction executes. It compares the contents of general registers r2 and r1 to check if they are equal.
  - Since r1 and r2 are equal it assigns P(1) the value of 1 (true) and P(2) the value of 0 (false)

- (P1) Sub r4 = r2, r1
  - P(1) is 1 (true) again and so this instruction executes. It subtracts the contents of general registers r2 and r1 and assigns it to general register r4
  - r4 is now 0
• (P2) And r4 = r2, r1
  - P(2) is 0 (false) and so this instruction does not execute.
  - r4 is still equal to 0

9.2 Program 2

```
(P1) Ld r100 = [r10]
(P0) Ld r1 = [r1]
(P0) Ld r2 = [r2]
(P0) Cmp.lt p1, p2 = r1, r2
(P1) St [r0] = r2
(P2) St [r1] = r2
(P1) Cmp.gt p3, p4 = r1, r2
(P3) Add r3 = r1, r2
(P4) Ld r3 = [r10]
(P4) Cmp.gt p3, p4 = r3, r2
(P3) Mov r4 = r3
(P4) Add r3 = r1, r2
```

For your reference this code is kept in your C: Itanium-instructions.txt

Figure 31. Program 2 Instructions
Figure 32. Program 2 Memory

<table>
<thead>
<tr>
<th>GENERAL REGISTERS</th>
<th>MEMORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0 --&gt; 0000000 (0)</td>
<td>M0 --&gt; 10100 (20)</td>
</tr>
<tr>
<td>R1 --&gt; 1010 (10)</td>
<td>M1 --&gt; 1010 (10)</td>
</tr>
<tr>
<td>R2 --&gt; 10100 (20)</td>
<td>M2 --&gt; 10100 (20)</td>
</tr>
<tr>
<td>R3 --&gt; 1100100 (100)</td>
<td>M3 --&gt; 11110 (30)</td>
</tr>
<tr>
<td>R4 --&gt; 1100100 (100)</td>
<td>M4 --&gt; 101000 (40)</td>
</tr>
<tr>
<td>R5 --&gt; 0000000 (0)</td>
<td>M5 --&gt; 110010 (50)</td>
</tr>
<tr>
<td>R6 --&gt; 0000000 (0)</td>
<td>M6 --&gt; 111100 (60)</td>
</tr>
<tr>
<td>R7 --&gt; 0000000 (0)</td>
<td>M7 --&gt; 1100110 (70)</td>
</tr>
<tr>
<td>R8 --&gt; 0000000 (0)</td>
<td>M8 --&gt; 1100100000 (800)</td>
</tr>
<tr>
<td>R9 --&gt; 0000000 (0)</td>
<td>M9 --&gt; 1111000100 (900)</td>
</tr>
<tr>
<td>R10 --&gt; 0000000 (0)</td>
<td>M10 --&gt; 1100100 (100)</td>
</tr>
<tr>
<td>R11 --&gt; 0000000 (0)</td>
<td>M11 --&gt; 1100100 (100)</td>
</tr>
<tr>
<td>R12 --&gt; 0000000 (0)</td>
<td>M12 --&gt; 1100100 (100)</td>
</tr>
<tr>
<td>R13 --&gt; 0000000 (0)</td>
<td>M13 --&gt; 1100100 (100)</td>
</tr>
<tr>
<td>R14 --&gt; 0000000 (0)</td>
<td>M14 --&gt; 1100100 (100)</td>
</tr>
<tr>
<td>R15 --&gt; 0000000 (0)</td>
<td>M15 --&gt; 1100100 (100)</td>
</tr>
<tr>
<td>R16 --&gt; 0000000 (0)</td>
<td>M16 --&gt; 1100100 (100)</td>
</tr>
<tr>
<td>R17 --&gt; 0000000 (0)</td>
<td>M17 --&gt; 1100100 (100)</td>
</tr>
<tr>
<td>R18 --&gt; 0000000 (0)</td>
<td>M18 --&gt; 1100100 (100)</td>
</tr>
<tr>
<td>R19 --&gt; 0000000 (0)</td>
<td>M19 --&gt; 1100100 (100)</td>
</tr>
<tr>
<td>R20 --&gt; 0000000 (0)</td>
<td>M20 --&gt; 1100100 (100)</td>
</tr>
</tbody>
</table>

Figure 33. Program 2 Results (Registers and Memory)
OUTPUT

Initializing Registers
Initializing Instruction Executors
Reading memory data from file ...
Adding 1(1) to memory location 0
Adding 1010(10) to memory location 1
Adding 10100(20) to memory location 2
Adding 11110(30) to memory location 3
Adding 10100(40) to memory location 4
Adding 11010(50) to memory location 5
Adding 11110(60) to memory location 6
Adding 1000110(70) to memory location 7
Adding 110010000(800) to memory location 8
Adding 111000000(900) to memory location 9
Adding 1101000(100) to memory location 10
Adding 1100100(100) to memory location 11
Adding 1100100(100) to memory location 12
Adding 1100100(100) to memory location 13
Adding 1100100(100) to memory location 14
Adding 1100100(100) to memory location 15
Adding 1100010(100) to memory location 16
Adding 1100100(100) to memory location 17
Adding 1100010(100) to memory location 18
Adding 1100010(100) to memory location 19
Adding 1100010(100) to memory location 20
Adding 1100010(100) to memory location 21
Finished memory reading data from file

Executing 1 line from the file:
(P1) Ld r100 = [r10]
Instruction type is Memory Instruction Type
ap = P1
mnemonic = (Memory type Instruction)--- »Ld
completer1 =
completer2 =
dest = r100
sources = [r10]
NOTE -> (P1) Ld r100 = [r10] was not executed because predicate register was set to false

Executing 2 line from the file:
(P0) Ld r1 = [r1]
Instruction type is Memory Instruction Type
ap = P0
mnemonic = (Memory type Instruction)--- »Ld
completer1 =
completer2 =
dest = r1
sources = [r1]
Loading 1010 (10) which was in memory location r1 to general register r1

Executing 3 line from the file:
(P0) Ld r2 = [r2]
Instruction type is Memory Instruction Type
ap = P0
mnemonic = (Memory type Instruction)--- »Ld
completer1 =
completer2 =
dest = r2
sources = [r2]
Loading 10100 (20) which was in memory location r2 to general register r2

Executing 4 line from the file:
(P0) Cmp Il p1, p2 = r1, r2
Instruction Type was Integer ALU
ap = P0
mnemonic = (Integer ALU)--- »Cmp
completer1 =
completer2 =
dest = p1, p2
sources = r1
sources = r2
Predicate register p1 is set to true
Predicate register p2 is set to false

Executing 5 line from the file:
(P1) St [r0] = r2
Instruction type is Memory Instruction Type
ap = P1
mnemonic = (Memory type Instruction)--- »St
completer1 =
completer2 =
dest = [r0]
sources = r2
Storing r2 to general memory in location r0

Executing 6 line from the file:
(P2) St [r1] = r2
Instruction type is Memory Instruction Type
ap = P2
mnemonic = (Memory type Instruction)--- »St
completer1 =
completer2 =
dest = [r1]
sources = r2
NOTE -> (P2) St [r1] = r2 was not executed because predicate register was set to false

Executing 7 line from the file:
(P1) Cmp.gt p3, p4 = r1, r2
Instruction Type was Integer ALU
ap = P1
mnemonic = (Integer ALU)--- »Cmp
completer1 = gt
completer2 =
dest = p3, p4
sources = r1
sources = r2
Predicate register p3 is set to false
Predicate register p4 is set to true
Figure 34. Program 2 Results (Output)
Explanation:

- (P1) Ld r100 = [r10]
  - P1 is 0 (false) by default so this instruction does not execute.

- (P0) Ld r1 = [r1]
  - P0 is 1 (true) by default and so this instruction executes. It loads 10 which was in memory location r1 to general register r1

- (P0) Ld r2 = [r2]
  - P0 is 1 (true) by default and so this instruction executes. It loads 20 which was in memory location r2 to general register r2

- (P0) Cmp.lt p1,p2 = r1,r2
  - P0 is 1 (true) by default and so this instruction executes. It compares to check if general register r1 is less than general register r2. Predicate register P1 is set to true and predicate register P2 is set to false since r1 is less than r2

- (P1) St [r0] = r2
  - P1 which was by default false is now true so the instruction executes. It stores contents of general register r2 in to memory at location r0. Memory location r0 now contains 20

- (P2) St [r1] = r2
  - The instruction was not executed because predicate register P2 was set to false

- (P1) Cmp.gt p3,p4 = r1,r2
  - P1 is set to 1 (true) from the previous instruction and so this instruction executes. It compares general register r1 and checks if it is greater than general register r2.

75
Since r1 is lesser than r2, predicate register P3 is set to false and predicate register P4 is set to true.

- (P3) Add r3 = r1, r2
  - The instruction is not executed since P3 is set to false by the previous instruction

- (P4) Ld r3 = [r10]
  - The instruction is executed since P4 was set to true. It loads the contents of memory location r0 into general register r3.
  - r3 now contains 100

- (P4) Cmp.gt p3, p4 = r3, r2
  - The instruction is executed since P4 was set to true. General registers r3 and r2 are compared.
  - Predicate register P3 is set to true and predicate register P4 is set to false since r3 is greater than r2.

- (P3) Mov r4 = r3
  - The instruction is executed since predicate register P3 is true. The contents of general register r3 is moved to general register r4
  - r4 is now 100

- (P4) Add r3 = r1, r2
  - The instruction is not executed since predicate register P4 is false.
9.3 Program 3

Enter Your Itanium Program

(P0) Ld r1 = [r1]
(P0) Ld r2 = [r2]
(P0) Add r3=r2,r1
(P0) St [r30] = r3
(P0) Ld r30 = [r30]
(P0) Sub r20=r30,r1
(P0) Mov r4 = r20
(P0) And r5=r4,r3
(P0) Or r6=r4,r3
(P0) Xor r7=r4,r3
(P0) Andcm r8=r4,r3
(P0) Ld r50 = [r8]
(P0) Shl r9=r50,r1
(P0) Ld r51 = [r0]
(P0) Shr r10=r50,r51

For your reference this code is kept in your C:\itanium-instructions.txt

Submit

Figure 35. Program 3 Instructions
Figure 36. Program 3 Memory

```
Figure 37. Program 3 Results (Registers and Memory)
```
Initializing Registers
Initializing Instruction Executors
Reading memory data from file ...
Adding 1(1) to memory location 0
Adding 1010(10) to memory location 1
Adding 10100(20) to memory location 2
Adding 1110(30) to memory location 3
Adding 101000(40) to memory location 4
Adding 110010(50) to memory location 5
Adding 111000(60) to memory location 6
Adding 100010(70) to memory location 7
Adding 1100000000(800) to memory location 8
Adding 1110000000(900) to memory location 9
Adding 1100100(100) to memory location 10
Adding 1100100(100) to memory location 11
Adding 1100100(100) to memory location 12
Adding 11000100 (100) to memory location 13
Adding 11001000(100) to memory location 14
Adding 11001000(100) to memory location 15
Adding 11001000(100) to memory location 16
Adding 11001000(100) to memory location 17
Adding 11001000(100) to memory location 18
Adding 11001000 (100) to memory location 19
Adding 11001000(100) to memory location 20
Adding 11001000(100) to memory location 21
Finished memory reading data from file

Executing 1 line from the file:
(P0) Ld r1 = [r1]
Instruction type is Memory Instruction Type
op = P0
mnemonic = (Memory type Instruction) --- Ld
completer1 =
completer2 =
dest = r1
sources = [r1]
Loading 1010 (10) which was in memory location r1 to
general register r1

Executing 2 line from the file:
(P0) Ld r2 = [r2]
Instruction type is Memory Instruction Type
op = P0
mnemonic = (Memory type Instruction) --- Ld
completer1 =
completer2 =
dest = r2
sources = [r2]
Loading 101000 (20) which was in memory location r2 to
general register r2

Executing 3 line from the file:
(P0) Add r3=r2,r1
Instruction Type was Integer ALU
op = P0
mnemonic = (IntegerALU) --- Add
completer1 =
completer2 =
dest = r3
sources = r2
sources = r1
41 BIT INSTRUCTION / OPCODE IS 8000002130
Destination r3 = 11110(30)

Executing 4 line from the file:
(P0) St [r30] = r3
Instruction type is Memory Instruction Type
op = P0
mnemonic = (Memory type Instruction) --- St
completer1 =
completer2 =
dest = [r30]
sources = r3
Storing r3 to general memory in location r30

Executing 5 line from the file:
(P0) Ld r30 = [r30]
Instruction type is Memory Instruction Type
op = P0
mnemonic = (Memory type Instruction) --- Ld
completer1 =
completer2 =
dest = r30
sources = [r30]
Loading 11110 (30) which was in memory location r30 to
general register r30

Executing 6 line from the file:
(P0) Sub r20=r30,r1
Instruction Type was integer ALU
op = P0
mnemonic = (IntegerALU) --- Sub
completer1 =
completer2 =
dest = r20
sources = r30
sources = r1
41 BIT INSTRUCTION / OPCODE IS 800010301200
Destination r20 = 20

Executing 7 line from the file:
(P0) Mov r4 = r20
Instruction type is Non Integer Instruction Type
op = P0
mnemonic = (NonIntegerALU) --- Mov
completer1 =
completer2 =
dest = r4
sources = r20
Moving 20 from register r20 to general register r4

Executing 8 line from the file:
(P0) And r5=r4,r3
Instruction Type was Integer ALU
op = P0
mnemonic = (IntegerALU) --- And
completer1 =
completer2 =
dest = r5
sources = r4
sources = r3
41 BIT INSTRUCTION / OPCODE IS 8000304350
Destination r5 = 20
Figure 38. Program 3 Results (Output)

80

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Explanation:

In all the instructions below P0 is true and so all the instructions will execute. The primary goal of this program is to demonstrate the different instructions that can be run on the Itanium Instruction Set-Simulator

- (P0) Ld r1 = [r1]
  - It loads 10 which was in memory location r1 to general register r1
- (P0) Ld r2 = [r2]
  - It loads 20 which was in memory location r2 to general register r2
- (P0) Add r3=r2,r1
  - It adds the contents of general register r2 and r1 and puts the result in the destination register r3.
  - r3 now contains 30
- (P0) St [r30] = r3
  - It stores the contents of general register r3 to general memory in location r30
- (P0) Ld r30 = [r30]
  - Loading 30 which was in memory location r30 to general register r30
- (P0) Sub r20=r30,r1
  - Subtracts the contents of general register r1 from r30 and puts it in r20
  - r20 is now 20
• (P0) Mov r4 = r20
  o Moves 20 from register r20 to general register r4
• (P0) And r5=r4,r3
  o This is the And instruction which performs a bitwise ‘AND’ on the contents of
general register r4 and r3. In a bitwise ‘And’ only 1 and 1 = 1.
  o The result of a bitwise And operation on r4 which contains 20 and r3 which contains
30 is 20 which is put in r5

| R4 = 20 (0010100) |
| R3 = 30 (0011110) |
| R5 = 20 (0010100) |

• (P0) Or r6=r4,r3
  o This is the Or instruction which performs a bitwise ‘OR’ on the contents of general
register r4 and r3. In a bitwise ‘OR’ 0 and 1 or 1 and 0 or 1 and 1 is 1.
  o The result of a bitwise OR operation on r4 which contains 20 and r3 which
contains 30 is 30 which is put in r6

| R4 = 20 (0010100) |
| R3 = 30 (0011110) |
| R6 = 30 (0011110) |

82
• (P0) Xor r7=r4,r3
  o This is the exclusive or (XOR) instruction which performs a bitwise ‘XOR’ on the
contents of general register r4 and r3. Here 0 and 1 is 1 but 1 and 1 is 0.
  o The result of a bitwise XOR operation on r4 which contains 20 and r3 which
contains 30 is 10 which is put in r7

<table>
<thead>
<tr>
<th>R4 = 20 (0010100)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3 = 30 (0011110)</td>
</tr>
<tr>
<td>R7 = 10 (0001010)</td>
</tr>
</tbody>
</table>

• (P0) Andcm r8=r4,r3
  o This is the ‘And compliment’ instruction which compliments the second source
and then performs a bitwise ‘And’ on it.
  o The value of r3 which is the second source is complimented and then a bitwise add
is performed with r4
  o The result of the r3 compliment is 97 and when the bitwise and is performed, the
result is 0. General register r8 is now 0

<table>
<thead>
<tr>
<th>R3 = 30 (0011110)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3 = 97 (1100001) Compliment</td>
</tr>
<tr>
<td>R4 = 20 (0010100)</td>
</tr>
<tr>
<td>R8 = 0 (0000000)</td>
</tr>
</tbody>
</table>
• (P0) Ld r50 = [r8]
  o It loads 800 which was in memory location r8 to general register r50

• (P0) Shl r9=r50,r1
  o It performs a ‘Shift Left’ operation. The second source r1 determines the number
    of times r50 must be shifted to the left
  o The max shift value is 63. Values larger than 63 cannot be shifted left
  o The value of r9 is now 819200

<table>
<thead>
<tr>
<th>R50 = 800 (1100100000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1 = 10 so move r50 10 spaces to the left</td>
</tr>
<tr>
<td>R9 = 819200 (110010000000000000000000000)</td>
</tr>
</tbody>
</table>

• (P0) Ld r51 = [r0]
  o It loads 1 which was in memory location r0 to general register r51

• (P0) Shr r10=r50,r51
  o It performs a ‘Shift Right’ operation. The second source r51 determines the
    number of times r50 must be shifted to the right
  o The value of r10 is now 400

<table>
<thead>
<tr>
<th>R50 = 800 (1100100000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R51 = 1 so move r50 1 spaces to the right</td>
</tr>
<tr>
<td>R9 = 400 (01100100000)</td>
</tr>
</tbody>
</table>
10. CONCLUSION AND FUTURE WORK

The Itanium Instruction Set Simulator is ideal software for new students and enthusiastic users who want to study the Itanium Architecture and run programs. The software design is flexible, cohesive and new features can be easily added to it without changing what has already been developed.

One of the areas that can be developed further is the instruction type support. The Itanium Instruction Set Simulator (IISS) can run many instructions types like the Integer, Memory and Non Integer instructions. It can further be developed to run Floating-point type instructions, Branch instructions and Extended instruction types.

Additional instructions can be added to the IISS for its currently existing instruction types. For instance, support for Multimedia Shift Left can be added. Such additional instructions will help enhance the software further.
APPENDICES
APPENDIX A

RUNNING THE ITANIUM INSTRUCTION-SET SIMULATOR SOFTWARE
Chapter 8 provides detailed instructions for setting up the Itanium Instruction Set Simulator (IISS) software as well as setup information and how to run the software. Furthermore a video of how to setup the software is also made available by the author.

The program can be run in 2 modes

1. The Application mode which displays its results on the console. The users must setup the environment as indicated in Chapter 8 and then:
   - enter the memory data in c:\data.txt
   - the instructions which are to be run in c:\Itanium-instructions.txt

2. A Web mode which displays a data window on the browser to enter data and an instruction window to enter instructions. It also displays results on the browser which include the contents of the registers, the memory and the results.
APPENDIX B

CODE USED IN THE PROGRAM
Start.java

/**
 * Written by Priyanka Shah at the Southern Connecticut State University, New Haven 2005.
 */
package edu.scsu.cs.Itanium.main;

import edu.scsu.cs.Itanium.executers.ExecuterFactory;
import edu.scsu.cs.Itanium.memory.MemoryManager;
import edu.scsu.cs.Itanium.registers.ManageRegisters;

//Main class for the Application
public class Start {

    public static void main(String[] args) {

        // Create Registers
        ManageRegisters.createRegisters();

        // Create Executers
        ExecuterFactory.getInstance().addExecuters();

        // Load Data into Memory from Data File
        MemoryManager.getInstance().addToMemory();

        //Launch program
        RunProgram.launch();

    }
}

90
Controller.java

/*
* Written by Priyanka Shah at the Southern Connecticut State University, New Haven 2005.
*/
package edu.scsu.cs.Itanium.main;

import java.util.HashMap;
import org.apache.commons.lang.StringUtil;
import logs.Logs;
import edu.scsu.cs.Itanium.executers.*;
import edu.scsu.cs.Itanium.instructions.InstrIntegerALU;
import edu.scsu.cs.Itanium.instructions.InstrMemory;
import edu.scsu.cs.Itanium.instructions.InstrNonIntegerALU;
import edu.scsu.cs.Itanium.instructions.exceptions.*;
import edu.scsu.cs.Itanium.instructions.exceptions.InstrIntegerALUException;
import edu.scsu.cs.Itanium.instructions.exceptions.InstrMemoryException;
import edu.scsu.cs.Itanium.registers.PredicateRegisters;
import edu.scsu.cs.Itanium.tools.InstructionsTypeXMLFileReader;

/**
 * This class is the program controller.
 * All Requests flow through this file
 */
public class Controller {

    private static HashMap hm;
    private static String opcode;
    private static String instructionType;

    public static void send(String line) {
        hm = InstructionsTypeXMLFileReader.getInstance().getInstructionType(line);
        instructionType = (String) hm.get("instructionType");
        System.out.println(instructionType);
        opcode = (String) hm.get("opcode");

        // Based on instruction call the respective Instruction parser

        // ALU parser
        HashMap hmIntALU = null;
        Boolean executeALUFlag = new Boolean(false);
    }
if (instructionType.equals("A")){
  System.out.println("Instruction Type was ALU");
  Logs.getInstance().setLog(line);
  Logs.getInstance().setLog("Instruction Type was Integer ALU");
  System.out.println("Sending Instruction to InstrIntegerALU to separate instruction ...");
}

try{
  hmIntALU = InstrIntegerALU.execute(line,opcode);
}catch(InstrIntegerALUException e){
  e.printStackTrace();
}

//Check Predicate register to decide if the instruction should be executed
String predicateRegisterNumberKey = (String)hmIntALU.get("qp");
executeALUFlag = (Boolean)
PredicateRegisters.getInstance().getRegisterValue(predicateRegisterNumberKey);

//Once HashMap is got, execute the function based on predicate register
if(executeALUFlag.booleanValue()==true){
  Executer ex =
  ExecuterFactory.getInstance().getExecuter("IntegerALU");
  System.out.println("Sending Instruction to IntegerALUExecuter");
  ex.execute(hmIntALU);
} else{

//Incase of unconditional completers which are completer2 follow the below.

if(!((String)hmIntALU.get("completer2").equals("")) {  
  //Since qp is zero the predicates are made zero
  String dest = (String)hm.get("dest");
  //Seperate predicates and ensure compatability
  String dest1 = StringUtils.substringBefore(dest,",");
  String dest2 = StringUtils.substringAfter(dest,",");
  dest1 = dest1.toLowerCase();
  dest2 = dest2.toLowerCase();

  PredicateRegisters.getInstance().setRegisterValue(dest1,new Boolean(false));
  PredicateRegisters.getInstance().setRegisterValue(dest2,new Boolean(false));
}
PredicateRegisters.getInstance().setRegisterValue(dest2, new Boolean(false));

System.out.println("Predicate register "+dest1+" is set to false");
System.out.println("Predicate register "+dest2+" is set to false");
Logs.getInstance().setLog("Predicate register "+dest1+" is set to false");
Logs.getInstance().setLog("Predicate register "+dest2+" is set to false");

//Predicate Register is false
System.out.println("Note -> "+line+" was not executed because predicate register was set to false");
Logs.getInstance().setLog("Note -> "+line+" was not executed because predicate register was set to false");

//Memory Instruction parser
HashMap hmMem = null;
Boolean executeMemFlag = new Boolean(false);
if (instructionType.equals("M")){
    System.out.println("Memory Instruction Type");
    System.out.println("Sending Instruction to InstrMemory to seperate instruction ...");
    Logs.getInstance().setLog(line);
    Logs.getInstance().setLog("Instruction type is Memory Instruction Type");
    try{
        hmMem = InstrMemory.execute(line,opcode);
        //stores seperated values (qp,mnemonic,etc) from line
    }catch(InstrMemoryException e){
        e.printStackTrace();
    }
}

//Check predicate register value to decide if the instruction should be executed
String predicateRegisterNumberKey = (String)hmMem.get("qp");
executeMemFlag = (Boolean)
PredicateRegisters.getInstance().getRegisterValue(predicateRegisterNumberKey);

//Get HashMap and execute the function based on predicate register
if(executeMemFlag.booleanValue()==true){
Executor ex = ExecuterFactory.getInstance().getExecutor("Memory");
System.out.println("Sending Instruction to MemoryExecutor");
ex.execute(hmMem);
}
else{
System.out.println("Note -> "+line+" was not executed because predicate register was set to false");
Logs.getInstance().setLog("Note -> "+line+" was not executed because predicate register was set to false");
}

//Non Integer Instruction parser
HashMap hmNonInt = null;
Boolean executeNonIntFlag = new Boolean(false);

if (instructionType.equals("I")) {

Logs.getInstance().setLog(line);
Logs.getInstance().setLog("Instruction type is Non Integer Instruction Type");
System.out.println("Non Integer Instruction Type");
System.out.println("Sending Instruction to Instr-NonInteger to separate instruction ...");
try{

hmNonInt = InstrNonIntegerALU.execute(line,opcode);
//stores seperated values (qp,mnemonic,etc) from line
}

} catch(InstrNonIntegerALUException e) {

e.printStackTrace();
}

//Check predicate register value to decide if the instruction should be executed
String predicateRegisterNumberKey = (String)hmNonInt.get("qp");
executeNonIntFlag = (Boolean)
PredicateRegisters.getInstance().getRegisterValue(predicateRegisterNumberKey);
// Once HashMap is got execute the function based on predicate register

if (executeNonIntFlag.booleanValue() == true) {
    Executer ex =
    ExecuterFactory.getInstance().getExecuter("NonIntegerALU");
    System.out.println("Sending Instruction to Non Integer Executer");
    ex.execute(hmNonInt);
} else {
    System.out.println("Note -> +line+" was not executed because
    predicate register was set to false");
    Logs.getInstance().setLog("Note -> +line+" was not executed because
    predicate register was set to false");
} // Add more Instruction types here later to develop program further.

RunProgram.java

/**
 * Written by Priyanka Shah at the Southern Connecticut State University, New Haven 2005.
 */

package edu.scsu.cs.Itanium.main;
import java.io.BufferedReader;
import java.io.IOException;
import logs.Logs;
import edu.scsu.cs.Itanium.tools.InstructionFileReader;

public class RunProgram {

    /**
     * This class launches the program and acts like a client.
     * It sends the instructions one line at a time from the file to the main controller of
     * the program.
     */
    public static void launch() {

        BufferedReader br =
        InstructionFileReader.read("C:\Itanium-instructions.txt");
        // Read instructions from file & Pass the instruction to the Program Controller
        try {

        95

    Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
String line = br.readLine();
if (line == null) {
    Logs.getInstance().setLog("Please enter instructions in the Itanium-instructions.txt file");
} 
int countLine = 0;
while (line != null) {
    countLine++;
    System.out.println("Executing " + countLine + " line from the file:");
    Logs.getInstance().setLog("Executing " + countLine + " line from the file:");
    System.out.println(line);
    Controller.send(line);
    line = br.readLine();
}
catch (IOException e) {
    e.printStackTrace();
}

ManageRegisters.java

/*
 * Written by Priyanka Shah at the Southern Connecticut State University, New Haven 2005.
 */
package edu.scsu.cs.Itanium.registers;

import logs.Logs;

public class ManageRegisters {

    private ManageRegisters() {
    }

    public static void createRegisters() {
        Logs.getInstance().setLog("Initializing Registers");
        GeneralRegisters.getInstance().createResetRegister();
        PredicateRegisters.getInstance().createResetRegister();
    }

    96

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
GeneralRegisters.java

/*
 * Written by Priyanka Shah at the Southern Connecticut State University, New Haven 2005.
 */
package edu.scsu.cs.Itanium.registers;
import java.util.HashMap;

/**
 * @author Priyanka Shah
 * * This class is a singleton so only one instance is available for the entire application
 */
public class GeneralRegisters {

    private static GeneralRegisters instance = new GeneralRegisters();
    private HashMap generalRegister;

    private GeneralRegisters() {
    }

    public static GeneralRegisters getInstance()
    { return instance; }

    public void createResetRegister()
    { generalRegister = new HashMap(128);
      String key = null;
      for (int i = 0; i < 128; i++)
      {
        if(i == 0)
        {
          // Set the first register gr0 to 0.
          // Cannot change gr0
          String r0 = "0";
          generalRegister.put("r0", r0);
        }
        else
        {
          key = "r" + String.valueOf(i);
          generalRegister.put(key, new String("0"));
        }
      }
    }
}
public void setRegister(String key, String value){
    generalRegister.put(key, value);
}

public String getRegister(String key){
    return (String)generalRegister.get(key);
}

PredicateRegisters.java

/* Written by Priyanka Shah at the Southern Connecticut State University, New Haven 2005.
 * createResetRegister(): Fills predicate register with values
 * There are 64 predicate registers
 */
package edu.scsu.cs.Lтанium.registers;
import java.util.HashMap;
import logs.Logs;
/**
 * @author Priyanka Shah
 * This class is a singleton so only one instance is available for the entire application
 */
public class PredicateRegisters{

    private static PredicateRegisters instance = new PredicateRegisters();
    private static HashMap predicateRegister;

    private PredicateRegisters() {
    }

    public static PredicateRegisters getInstance(){
        return instance;
    }
}
public void createResetRegister()
{
    predicateRegister = new HashMap(64);
    String key = "p0";

    // Fill predicate register with values
    for (int i = 0; i < 64; i++){
        if(i == 0){
            // Set pr0 to 1 or true. Cannot change it.
            final Boolean b = new Boolean(true);
            predicateRegister.put(key,b);
        }
        else{
            key = "p" + String.valueOf(i);
            predicateRegister.put(key,new Boolean(false));
        }
    }
}

public void setRegisterValue(String key, Object value){
    key = key.toLowerCase();

    if(key.equals("p0") || key.equals("P0")){
        Logs.getInstance().setLog("Error: Values cannot be set for p0 register");
        Logs.getInstance().setLog("Please change the program and try again.");
    }
    predicateRegister.put(key,value);
}

public Object getRegisterValue(String key){
    key = key.toLowerCase();
    return predicateRegister.get(key);
}

/***** Get Registers value for key ******/
`ExecutorFactory.java`
/*
 */
package edu.scsu.cs.Itanium.executers;
import java.util.HashMap;
import Logs.Logs;

public class ExecutorFactory {

    /**
     * This is an executor factory which returns the correct executor to the program
     */

    private static ExecutorFactory instance = new ExecutorFactory();
    private static HashMap executorMap = new HashMap();

    private ExecutorFactory() {
    }

    public static ExecutorFactory getInstance() {
        return instance;
    }

    public void addExecuters()
        Logs.getInstance().setLog("Initialzing Instruction Executers");
        executorMap.put("IntegerALU", new IntegerALUExecutor());
        executorMap.put("Memory", new MemoryExecutor());
        executorMap.put("NonIntegerALU", new NonIntegerALUExecutor());
    }

    public Executor getExecuter(String type)
        return (Executor)executorMap.get(type);
    }
}
**Executer.java**

```java
/*
* Written by Priyanka Shah at the Southern Connecticut State University, New Haven 2005.
*/
package edu.scsu.cs.Itanium.executers;
import java.util.HashMap;

public interface Executer {
    void execute(HashMap hmIntALU);
}
```

**IntegerALUExecuter.java**

```java
/*
* Written by Priyanka Shah at the Southern Connecticut State University, New Haven 2005.
*/
package edu.scsu.cs.Itanium.executers;
import java.util.HashMap;
import logs.Logs;
import org.apache.commons.lang.StringUtils;
import edu.scsu.cs.Itanium.registers.PredicateRegisters;

public class IntegerALUExecuter implements Executer {
    public IntegerALUExecuter() {
        super();
    }

    public void execute(HashMap hm) {
        //Add
        if (hm.get("mnemonic").equals("Add")) {
            //Get the values of the sources from the registers and add them & put in dest
            //1. Get values
            String[] sources = (String[])(hm.get("sources"));

            // This has sources r2 and r3 from program
```
sources[0] = StringUtils.replace(sources[0], " ", ",");
sources[1] = StringUtils.replace(sources[1], " ", ",");

Integer val1 = new Integer(Integer.parseInt(GeneralRegisters.getInstance().getRegister(sources[0])));

Integer val2 = new Integer(Integer.parseInt(GeneralRegisters.getInstance().getRegister(sources[1])));

//2. Add them
int destVal = val1.intValue() + val2.intValue();
String destV = "" + destVal;
//3. Put in dest
String destKey = (String)hm.get("dest");
GeneralRegisters.getInstance().setRegister(destKey, destV);
System.out.println("Destination "+destKey+= "" +destVal);
Logs.getInstance().setLog("Destination +destKey+= "" +destVal);
}

// Subtract
if (hm.get("mnemonic").equals("Sub")) {

// Get the values of the sources from the registers and subtract 1st source - second
//1. Get values
String[] sources = (String[])hm.get("sources"); // This has sources r2 and r3 from program
sources[0] = StringUtils.replace(sources[0], " ", ",");
sources[1] = StringUtils.replace(sources[1], " ", ",");

Integer val1 = new Integer(Integer.parseInt
(GeneralRegisters.getInstance().getRegister(sources[0])));

Integer val2 = new Integer(Integer.parseInt
(GeneralRegisters.getInstance().getRegister(sources[1])));

//2. Subtract them
int destVal = val1.intValue() - val2.intValue();
String destV = "" + destVal;

102
//3. Put in dest
String destKey = (String)hm.get("dest");

GeneralRegisters.getInstance().setRegister(destKey,destV);
System.out.println("Destination "+destKey+" = "+destVal);
Logs.getInstance().setLog("Destination "+destKey+" = "+destVal);
}

//And

if (hm.get("mnemonic").equals("And")) {

  String[] sources = (String[])hm.get("sources");
sources[0] = StringUtils.replace(sources[0],"","");
sources[1] = StringUtils.replace(sources[1],"","");

  int val1 =
    Integer.parseInt(GeneralRegisters.getInstance().getRegister(sources[0]));
  int val2 =
    Integer.parseInt(GeneralRegisters.getInstance().getRegister(sources[1]));

  //2. Perform bitwise & on them
  int destVal = val1 & val2;
  String destV = ""+destVal;

  //3. Put in dest
  String destKey = (String)hm.get("dest");

  GeneralRegisters.getInstance().setRegister(destKey,destV);
  System.out.println("Destination "+destKey+" = "+destVal);
  Logs.getInstance().setLog("Destination "+destKey+" = "+destVal);
}

// Or

if (hm.get("mnemonic").equals("Or")) {

  String[] sources = (String[])hm.get("sources");
sources[0] = StringUtils.replace(sources[0],"","");

  // Further code

  String destKey = (String)hm.get("dest");

  GeneralRegisters.getInstance().setRegister(destKey,destV);
  System.out.println("Destination "+destKey+" = "+destVal);
  Logs.getInstance().setLog("Destination "+destKey+" = "+destVal);
}
sources[1] = String.valueOf(sources[1].replace("",""));

int val1 = Integer.parseInt((GeneralRegisters.getInstance().getRegister(sources[0])));
int val2 = Integer.parseInt((GeneralRegisters.getInstance().getRegister(sources[1])));

// 2. Perform bitwise | on them
int destVal = val1 | val2;
String destV = ""+destVal;

// 3. Put in dest
String destKey = (String)hm.get("dest");
GeneralRegisters.getInstance().setRegister(destKey,destV);
System.out.println("Destination "+destKey+" "+destVal);
Logs.getInstance().setLog("Destination "+destKey+" "+destVal);

// Xor

if (hm.get("mnemonic").equals("Xor")) {

// 1. Get values
String[] sources = (String[])hm.get("sources");
sources[0] = String.valueOf(sources[0].replace("",""));
sources[1] = String.valueOf(sources[1].replace("",""));

int val1 = Integer.parseInt((GeneralRegisters.getInstance().getRegister(sources[0])));
int val2 = Integer.parseInt((GeneralRegisters.getInstance().getRegister(sources[1])));

// 2. Perform bitwise ^ on them
int destVal = val1 ^ val2;
String destV = ""+destVal;

// 3. Put in dest
String destKey = (String)hm.get("dest");
GeneralRegisters.getInstance().setRegister(destKey,destV);
System.out.println("Destination "+destKey+" "+destVal);
Logs.getInstance().setLog("Destination "+destKey+" "+destVal);
}
// And compliment

if (hm.get("mnemonic").equals("Andcm")) {

    // 1. Get values
    String[] sources = (String[])hm.get("sources");
    sources[0] = StringTools.replace(sources[0], " ", " ");
    sources[1] = StringTools.replace(sources[1], " ", " ");
    int val1 = Integer.parseInt
        (GeneralRegisters.getInstance().getRegister(sources[0]));
    int val2 = Integer.parseInt
        (GeneralRegisters.getInstance().getRegister(sources[1]));

    // 2. Perform bitwise and and then ~ on the second source
    val2 = ~ val2;
    int destVal = val1 & val2;
    String destV = "\"\"+destVal;

    // 3. Put in dest
    String destKey = (String)hm.get("dest");
    GeneralRegisters.getInstance().setRegister(destKey, destV);
    System.out.println("Destination "+destKey+" = "+destVal);
    Logs.getInstance().setLog("Destination "+destKey+" = "+destVal);
}

// Shift Left

if (hm.get("mnemonic").equals("Shl")) {
    System.out.println("Executing Shift left");

    // 1. Get values
    String[] sources = (String[])hm.get("sources");
    sources[0] = StringTools.replace(sources[0], " ", " ");
    sources[1] = StringTools.replace(sources[1], " ", " ");
    int val1 = Integer.parseInt
        (GeneralRegisters.getInstance().getRegister(sources[0]));
    int val2 = Integer.parseInt
        (GeneralRegisters.getInstance().getRegister(sources[1]));

    // 2. Perform shift left
    // val2 defines the number of times that val1 must be shifted to left
    // The max value in val2 can be 63
    if(val2 > 63){
        System.out.println("Value larger than 63 so cannot shift left");
    }
System.exit(1);
}

int destVal = val1 << val2;
String destV = ""+destVal;

// 3. Put in dest
String destKey = (String)hm.get("dest");
GeneralRegisters.getInstance().setRegister(destKey,destV);
System.out.println("Destination "+destKey+" "+destVal);
Logs.getInstance().setLog("Destination "+destKey+" "+destVal);
}

// Shift Right

if (hm.get("mnemonic").equals("Shr")) {
    System.out.println("Executing Shift right");

    // 1. Get values
    String[] sources = (String[])hm.get("sources");
    sources[0] = String.valueOf(sources[0], " ", "");
    sources[1] = String.valueOf(sources[1], " ", "");

    int val1 =
    Integer.parseInt(GeneralRegisters.getInstance().getRegister(sources[0])
    );
    int val2 =
    Integer.parseInt(GeneralRegisters.getInstance().getRegister(sources[1])
    );

    // 2. Perform shift right
    // val2 defines the number of times that val1 must be shifted to right.
    The max value in val2 can be 63
    if(val2 > 63){
        System.out.println("Value larger than 63 so cannot shift left");
        System.exit(1);
    }
}

106
int destVal = val1 >> val2;
String destV = ""+destVal;

//3. Put in dest
String destKey = (String)hm.get("dest");
GeneralRegisters.getInstance().setRegister(destKey,destV);
System.out.println("Destination "+destKey+" = "+destVal);
Logs.getInstance().setLog("Destination "+destKey+" = "+destVal);
}

// Compare
if (hm.get("mnemonic").equals("Cmp")) {

    System.out.println("In Cmp");
    String[] sources = (String[])hm.get("sources");// This has sources r2 and r3 from program
    String completer1 = (String)hm.get("completer1");
    String completer2 = (String)hm.get("completer2");
    String dest = (String)hm.get("dest");
    //Remove spaces
    dest = StringUtils.replace(dest," ","");

    sources[0] = StringUtils.replace(sources[0]," ","");
    sources[1] = StringUtils.replace(sources[1]," ","");
    //Separate predicates and ensure comatibility
    String dest1 = StringUtils.substringBefore(dest,".");
    String dest2 = StringUtils.substringAfter(dest,".");
    dest1 = dest1.toLowerCase();
    dest2 = dest2.toLowerCase();

    System.out.println("Source 0 = "+sources[0]);
    System.out.println("Source 1 = "+sources[1]);
    System.out.println("Completer 1 = "+completer1);
    System.out.println("Completer 2 = "+completer2);
    System.out.println("Dest = "+dest);
    System.out.println("Dest 1 = "+dest1);
    System.out.println("Dest 2 = "+dest2);

    //Execute if eq
    if(completer1.equals("eq")) {

107

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Integer val1 = new
Integer(Integer.parseInt(GeneralRegisters.getInstance().getRegister(sources[0])));
Integer val2 = new
Integer(Integer.parseInt(GeneralRegisters.getInstance().getRegister(sources[1])));

if(val1.intValue() == val2.intValue()) {
    PredicateRegisters.getInstance().setRegisterValue(dest1,new
    Boolean(true));
    PredicateRegisters.getInstance().setRegisterValue(dest2,new Boolean(false));
    System.out.println("Predicate register "+dest1+" is set to true");
    System.out.println("Predicate register "+dest2+" is set to false");
    Logs.getInstance().setLog("Predicate register "+dest1+" is set to true");
    Logs.getInstance().setLog("Predicate register "+dest2+" is set to false");
} else {
    PredicateRegisters.getInstance().setRegisterValue(dest1,new Boolean(true));
    PredicateRegisters.getInstance().setRegisterValue(dest2,new Boolean(true));
    System.out.println("Predicate register "+dest1+" is set to true");
    System.out.println("Predicate register "+dest2+" is set to true");
    Logs.getInstance().setLog("Predicate register "+dest1+" is set to true");
    Logs.getInstance().setLog("Predicate register "+dest2+" is set to true");
}

// Less Than

if(completer1.equals("lt") || completer1.equals("ltu")) {
    Integer val1 = new
    Integer(Integer.parseInt(GeneralRegisters.getInstance().getRegister(sources[0])));

    Integer val2 = new
    Integer(Integer.parseInt(General Registers.getInstance().getRegister(sources[1])));

    if(val1.intValue() < val2.intValue()){
        PredicateRegisters.getInstance().setRegisterValue(dest1,new Boolean(true));
        PredicateRegisters.getInstance().setRegisterValue(dest2,new Boolean(false));
        System.out.println("Predicate register "+dest1+" is set to true");
        System.out.println("Predicate register "+dest2+" is set to false");
    }
Logs.getInstance().setLog("Predicate register "+dest1++ is set to true");
Logs.getInstance().setLog("Predicate register "+dest2++ is set to false");
} else {
    PredicateRegisters.getInstance().setRegisterValue(dest1,new Boolean(false));
    PredicateRegisters.getInstance().setRegisterValue(dest2,new Boolean(true));
    System.out.println("Predicate register "+dest1++ is set to false");
    System.out.println("Predicate register "+dest2++ is set to true");
}

// Greater Than

if(completer1.equals("gt") || completer1.equals("gtu")){
    Integer val1 = new Integer(Integer.parseInt(GeneralRegisters.getInstance().getRegister(sources[0])));
    Integer val2 = new Integer(Integer.parseInt(GeneralRegisters.getInstance().getRegister(sources[1])));
    if(val1.intValue() > val2.intValue()){
        PredicateRegisters.getInstance().setRegisterValue(dest1,new Boolean(true));
        PredicateRegisters.getInstance().setRegisterValue(dest2,new Boolean(false));
        System.out.println("Predicate register "+dest1++ is set to true");
        System.out.println("Predicate register "+dest2++ is set to false");
        Logs.getInstance().setLog("Predicate register "+dest1++ is set to true");
        Logs.getInstance().setLog("Predicate register "+dest2++ is set to false");
    } else {
        PredicateRegisters.getInstance().setRegisterValue(dest1,new Boolean(false));
        PredicateRegisters.getInstance().setRegisterValue(dest2,new Boolean(true));
        System.out.println("Predicate register "+dest1++ is set to false");
        System.out.println("Predicate register "+dest2++ is set to true");
        Logs.getInstance().setLog("Predicate register "+dest1++ is set to false");
        Logs.getInstance().setLog("Predicate register "+dest2++ is set to true");
    }
}
}}
MemoryExecuter.java

/*
 */

package edu.scsu.cs.Itanium.executers;
import java.util.HashMap;
import logs.Logs;
import org.apache.commons.lang.StringUtils;
import edu.scsu.cs.Itanium.memory.Memory;

public class MemoryExecuter implements Executer{

    public void execute(HashMap hm){

        //Load
        if (hm.get("mnemonic").equals("Ld")) {
            String[] sources = (String[])hm.get("sources");
            sources[0] = StringUtils.replace(sources[0],"[","");
            sources[0] = StringUtils.replace(sources[0],"]","");
            sources[0] = StringUtils.replace(sources[0]," "," ");

            String dest = (String) hm.get("dest");
            String value =
            Memory.getInstance().getValueFromMemory(sources[0]);
            GeneralRegisters.getInstance().setRegister(dest,value);
            System.out.println("Loading "+value+" which was in memory location "+sources[0]+" to general register "+dest);
            Logs.getInstance().setLog("Loading "+value+" which was in memory location "+sources[0]+" to general register "+dest);
        }

        //Store
        if (hm.get("mnemonic").equals("St")) {

            String[] sources = (String[])hm.get("sources"); //r3
            sources[0] = StringUtils.replace(sources[0]," "," ");

            String dest = (String) hm.get("dest");
            dest = StringUtils.replace(dest,"[","");
            dest = StringUtils.replace(dest,"]","");
            dest = StringUtils.replace(dest," "," ");

        }

    }

}
System.out.println(dest);

String regVal = (String) GeneralRegisters.getInstance().getRegister(sources[0]);

Memory.getInstance().addToMemory(dest, regVal);
System.out.println("Storing "+sources[0]+" to general memory in location "+dest);
Logs.getInstance().setLog("Storing "+sources[0]+" to general memory in location "+dest);

Memory.java

/**
 * Written by Priyanka Shah at the Southern Connecticut State University, New Haven 2005.
 */
package edu.scsu.cs.Itanium.memory;
import java.util.HashMap;

/**
 * This class is a singleton since we want a single repository or just one instance of memory
 * for the entire application
 */
public class Memory {
    private static Memory instance = new Memory();
    private static HashMap ht=null;

    private Memory(){
    }

    public static Memory getInstance(){
        return instance;
    }

    //Key is like memory location and value is memory value in location
    public void addToMemory(String key,String value){
        ht.put(key,value);
        System.out.println("Added value of "+value+" to memory location "+key);
    }
public String getValueFromMemory(String key) {
    return (String)ht.get(key);
}

public void resetMemory() {
    ht = new HashMap(100);
    String key = null;
    for (int i = 0; i < 100; i++) {
        key = "r" + String.valueOf(i);
        ht.put(key, new String("0"));
    }
}

NonIntegerALUExecuter.java

/*
*/
package edu.scsu.cs.Itanium.executers;
import java.util.HashMap;
import logs.Logs;
import org.apache.commons.lang.StringUtils;
import edu.scsu.cs.Itanium.memory.Memory;

public class NonIntegerALUExecuter implements Executer {

    public void execute(HashMap hm) {

        //Move
        if (hm.get("mnemonic").equals("Mov")) {
            String[] sources = (String[])hm.get("sources");
            sources[0] = StringUtils.replace(sources[0], ",", ",");
            String dest = (String) hm.get("dest");
            dest = StringUtils.replace(dest, ",", ",");
        }
    }
}
String source = GeneralRegisters.getInstance().getRegister(sources[0]);

GeneralRegisters.getInstance().setRegister(dest, source);
System.out.println("Moving "+source+" from register "+sources[0]+" to general register "+dest);
Logs.getInstance().setLog("Moving "+source+" from register "+sources[0]+" to general register "+dest);
}

}

InstrIntegerALU.java

package edu.scsu.cs.Itanium.instructions;
import java.util.HashMap;
import logs.Logs;
import org.apache.commons.lang.StringUtils;
import edu.scsu.cs.Itanium.instructions.exceptions.*;

/**
 * This class knows how to interpret instruction types of Integer ALU
 */

public class InstrIntegerALU {

    //Create HashMap to store seperatedValues
    private static HashMap hm = new HashMap();

    public static HashMap execute(String instruction, String opcode)
        throws InstrIntegerALUException {

            if (opcode.length() == 0) {
                throw new InstrIntegerALUException("No opcode has been specified.
Correct the instruction-formats.xml");
            }

            if (instruction.length() == 0) {
                throw new InstrIntegerALUException("No instruction has been
specified. Correct the Itanium-instructions.txt");
            }


113
```java
/****** QP ******/
int indexCloseBrack = 0;
String qp = "p0";
if (StringUtils.contains(instruction, ")") {
    qp = StringUtils.substringBetween(instruction, ", ");
    indexCloseBrack = StringUtils.indexOf(instruction, ")");
}

hm.put("qp", qp);
System.out.println("qp = " + qp);
Logs.getInstance().setLog("qp = " + qp);

/****** Mnemonic ******/

boolean completerExists = false;
String mnemonic = null;

// Does a completer exist?
completerExists = StringUtils.contains(instruction, ".");
// Is there a qp specified with completer?
if (indexCloseBrack == 0 && completerExists) {
    mnemonic = StringUtils.substringBefore(instruction, ".");
    completerExists = true;
}
// Is there a qp specified and no completer?
if (indexCloseBrack == 0 && (!(completerExists)) {
    mnemonic = StringUtils.substringBefore(instruction, " ");
}
// There is a qp & is there is a completer?
if (indexCloseBrack != 0 && completerExists) {
    mnemonic = StringUtils.substringBetween(instruction, ", " , ".");
    completerExists = true;
}
// Is there is no qp and no completer?
if (indexCloseBrack != 0 && !(completerExists)) {
    mnemonic =StringUtils.substringBetween(instruction, " ", " ");
}

mnemonic = StringUtils.deleteWhitespace(mnemonic);
System.out.println(" mnemonic = (IntegerALU)--- >" + mnemonic);
Logs.getInstance().setLog("mnemonic = (IntegerALU)--- >" + mnemonic);
```

114
/** Complete lres ******************/

boolean secondCompleterExists = false;
String completer1 = null;

// If completer exists
if (completerExists) {
  // How many completers exist?
  int count = StringUtils.countMatches(instruction, ".");
  // If more than 1 then it means second completer also exists
  if (count == 2) {
    completer1 =
      StringUtils.substringBetween(instruction, ".", ".");
    secondCompleterExists = true;
  } else {
    completer1 =
      StringUtils.substringBetween(instruction, ".", ".");
  }
}
if(completer1==null){
  completer1="";
}
System.out.println(" completer1 = " + completer1);
Logs.getInstance().setLog("completer1 = " + completer1);
hm.put("completer1", completer1);

/******** Complete r2 *************/

String completer2 = null;
// Does second completer exist?
if (secondCompleterExists) {
  String instructionAfterCompleter1 =
    StringUtils.substringAfter(instruction, completer1);
  completer2 =
    StringUtils.substringBetween(
      instructionAfterCompleter1,
      ".",
      ".");
}
if(completer2==null){
  completer2="";
}
System.out.println("completer2 = " + completer2);

115
Logs.getInstance().setLog("completer2 = " + completer2);
hm.put("completer2", completer2);

/****** Dest ******/

String dest = null;
if (completerExists) {
    dest = StringUtils.substringBetween(instruction, completer1, ",");
}
if (secondCompleterExists) {
    dest = StringUtils.substringBetween(instruction, completer2, ",");
}
if (!completerExists && !secondCompleterExists) {
    dest = StringUtils.substringBetween(instruction, mnemonic, ",");
}
dest = StringUtils.strip(dest);

System.out.println(" dest = " + dest);
Logs.getInstance().setLog("dest = " + dest);

/****** Sources ******/

String[] sources =
    StringUtils.split(
        StringUtils.substringAfter(instruction, ","),
    ",");
for (int i = 0; i < sources.length; i++) {
    System.out.println(" sources = " + sources[i]);
    Logs.getInstance().setLog("sources = " + sources[i]);
}

/****** opcode ******/

if (sources.length == 3) {
    System.out.println(" Number of Sources = 3 so change opcode");
    //If opcode is of normal length
    if (opcode.length() == 6) {
        //Change last digit for Add to 1
        if (mnemonic.equals("add")) {
            opcode = "800011";
        }
        if (mnemonic.equalsIgnoreCase("sub")) {
            opcode = "800011";
        }
    }
}

116
System.out.println(" opcode = "+opcode);

/**** Add to hashmap ****/

if (mnemonic == null
    || dest == null
    || sources == null
    || opcode == null) {
    throw new InstrIntegerALUException("Please enter instructions in the correct format");
}

hm.put("mnemonic", mnemonic);
hm.put("dest", dest);
hm.put("sources", sources);
hm.put("opcode", opcode);

// Print the 41 bits
String source1 = String.valueOf(sources[0], "r", ",");
String source2 = String.valueOf(sources[1], "r", ",");
String destination = String.valueOf(dest, "r", ",");

String qpVal = qp.toLowerCase();
qpVal = String.valueOf(qp, "p", ",");
qpVal = String.valueOf(qp, "P", ",");
System.out.println("\n");

return hm;
}

}
/**
 * Written by Priyanka Shah at the Southern Connecticut State University, New Haven 2005.
 */
package edu.scsu.cs.Itanium.instructions;
import java.util.HashMap;
import logs.Logs;
import org.apache.commons.lang.StringUtils;
import edu.scsu.cs.Itanium.instructions.exceptions.*;

/**
 * This class knows how to interpret instruction types of Memory type
 */

public class InstrMemory {

    // Create HashMap to store seperatedValues
    private static HashMap hm = new HashMap();

    public static HashMap execute(String instruction, String opcode) throws InstrMemoryException {

        if (opcode.length() == 0) {
            throw new InstrMemoryException("No opcode has been specified. Correct the instruction-formats.xml");
        }
        if (instruction.length() == 0) {
            throw new InstrMemoryException("No instruction has been specified. Correct the Itanium-instructions.txt");
        }

        int indexCloseBrack = 0;
        String qp = "p0";
        if (StringUtils.contains(instruction, "(")) {
            //qp is string between ")" and ")"
            qp = StringUtils.substringBetween(instruction, ")", ")");
            indexCloseBrack = StringUtils.indexOf(instruction, ")");
        }
        //Qp seems okay so add to HashMap
        hm.put("qp", qp);
        System.out.println(" qp = " + qp);
        Logs.getInstance().setLog("qp = " + qp);

    }
}
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
completer1 = StringUtils.substringBetween(instruction, ".", " ");
}
if(completer1==null){
    completer1="";
}
System.out.println(" completer1 = " + completer1);
Logs.getInstance().setLog("completer1 = " + completer1);
hm.put("completer1", completer1);

/**** Completer2 ***********/

String completer2 = null;
// Does second completer exist?
if (secondCompleterExists) {
    String instructionAfterCompleter1 =
        StringUtils.substringAfter(instruction, completer1);
    completer2 =
        StringUtils.substringBetween(instructionAfterCompleter1, ".", " ");
}
if(completer2==null){
    completer2="";
}
System.out.println(" completer2 = " + completer2);
Logs.getInstance().setLog("completer2 = " + completer2);
hm.put("completer2", completer2);

/**** Dest ***********/

String dest = null;
if (completerExists) {
    dest = StringUtils.substringBetween(instruction, completer1, "=");
}
if (secondCompleterExists) {
    dest = StringUtils.substringBetween(instruction, completer2, "=");
}
if(!(completerExists) && !(secondCompleterExists)) {
    dest = StringUtils.substringBetween(instruction, mnemonic, "=");
}
dest = StringUtils.strip(dest);
Logs.getInstance().setLog("dest = " + dest);
System.out.println(" dest = " + dest);

/*** Sources ******/

String[] sources =
StringUtils.split(StringUtils.substringAfter(instruction, "="), ', ');
for (int i = 0; i < sources.length; i++) {
    System.out.println(" sources = " + sources[i]);
    Logs.getInstance().setLog("sources = " + sources[i]);
}

/*** Add to hashmap ******/

if (mnemonic == null || dest == null || sources == null
|| opcode == null) {
    throw new InstrMemoryException("Please enter instructions in the
correct format");
}

hm.put("mnemonic", mnemonic);
hm.put("dest", dest);
hm.put("sources", sources);
hm.put("opcode", opcode);

String [] printSources = new String[sources.length];
for(int i=0;i<sources.length;i++){
    printSources[0] = sources[i];
}

for(int i=0; i < printSources.length; i++){
    printSources[i] = StringUtils.replace(printSources[i], ",", ", ");
    printSources[i] = StringUtils.replace(printSources[i], ",", ", ");
    printSources[i] = StringUtils.replace(printSources[i], ",", ", ");
}

String destination = StringUtils.replace(dest, ",", "");
destination = StringUtils.replace(destination, ",", ", ");
destination = StringUtils.replace(destination, "",");

String qpVal = qp.toLowerCase();
qpVal = StringUtils.replace(qp, ",", ", ");

121
qpVal = StringUtils.replace(qp, "P", "");
System.out.println("");
System.out.println("41 Bit INSTRUCTION IS " + opcode + printSources[0]
+ destination + qpVal);
System.out.println("");
Logs.getInstance().setLog("41 Bit INSTRUCTION IS "
+ opcode +printSources[0]+ destination+ qpVal);
return hm;
}


InstrNonIntegerALU.java

/*
 * Written by Priyanka Shah at the Southern Connecticut State University, New Haven 2005.
 */
package edu.scsu.cs.Itanium.instructions;
import java.util.HashMap;
import logs.Logs;
import org.apache.commons.lang.StringUtils;
import edu.scsu.cs.Itanium.instructions.exceptions.);

/**
 * This class knows how to interpret instruction types of Non Integer type
 */
public class InstrNonIntegerALU {

    // Create HashMap to store seperatedValues
    private static HashMap hm = new HashMap();

    public static HashMap execute(String instruction, String opcode)
    throws InstrNonIntALUException {

        if (opcode.length() == 0) {
            throw new InstrNonIntALUException("No opcode has been
specified. Correct the instruction-formats.xml");
        }

        if (instruction.length() == 0) {


122
throw new InstrNonIntaluException("No instruction has been specified. Correct the Itanium-instructions.txt");
}

/*** QP ************
int indexCloseBrack = 0;
String qp = "p0";
if (StringUtils.contains(instruction, "(")) {

    qp =
    StringUtils.substringBetween(instruction, ", ( ");
    indexCloseBrack = StringUtils.indexOf(instruction, ")");
}

hm.put("qp", qp);
System.out.println(" qp = "+ qp);
Logs.getInstance().setLog("qp = "+ qp);

/*** Mnemonic **********/

boolean completerExists = false;
String mnemonic = null;

// Does a completer exist?
completerExists = StringUtils.contains(instruction, ",.");
// Is there a qp specified with completer?
if (indexCloseBrack == 0 && completerExists) {
    mnemonic = StringUtils.substringBefore(instruction, ",.");
    completerExists = true;
}
// Is there a qp specified and no completer?
if (indexCloseBrack == 0 && (!completerExists)) {
    mnemonic = StringUtils.substringBefore(instruction, " ");
}
// There is a qp & is there is a completer?
if (indexCloseBrack != 0 && completerExists) {
    mnemonic = StringUtils.substringBetween(instruction, ", ", ",.");
    completerExists = true;
}
// Is there is no qp and no completer?
if (indexCloseBrack != 0 && (!completerExists)) {
    mnemonic = StringUtils.substringBetween(instruction, ", ", "");
}
mnemonic = StringUtils.deleteWhitespace(mnemonic);
System.out.println("mnemonic = (NonIntegeralu)---" +
mnemonic);
Logs.getInstance().setLog("mnemonic = (NonIntegeralu)---" +
mnemonic);

/********** Completer1 **************/
boolean secondCompleterExists = false;
String completer1 = null;

// If completer exists
if (completerExists) {
    //How many completers exist?
    int count = StringUtils.countMatches(instruction, ".");
    //If more than 1 then it means second completer also exists
    if (count == 2) {
        completer1 =
        StringUtils.substringBetween(instruction, ".", ".");
        secondCompleterExists = true;
    } else {
        completer1 =
        StringUtils.substringBetween(instruction, ".", ".");
    }
}
if (completer1 == null) {
    completer1 = "";
}
System.out.println("completer1 = " + completer1);
Logs.getInstance().setLog("completer1 = " + completer1);
hm.put("completer1", completer1);

/********** Completer2 **************/

String completer2 = null;
// Does second completer exist?
if (secondCompleterExists) {
    String instructionAfterCompleter1 =
    StringUtils.substringAfter(instruction, completer1);
    completer2 =
    StringUtils.substringBetween(instructionAfterCompleter1, ".", ".");
}
if (completer2 == null) {
    completer2 = "";
}
Logs.getInstance().setLog("completer2 = " + completer2);
hm.put("completer2", completer2);

/****** Dest ******/

String dest = null;
if (completerExists) {
dest = StringUtils.substringBetween(instruction, completer1, ",\"\"); 
}
if (secondCompleterExists) {
dest = StringUtils.substringBetween(instruction, completer2, "\"\"); 
}
if (!completerExists && !secondCompleterExists) {
dest = StringUtils.substringBetween(instruction, mnemonic, "\"\"); 
}
dest = StringUtils.strip(dest);
Logs.getInstance().setLog("dest = " + dest);

/****** Sources ******/

String[] sources =
    StringUtils.split(StringUtils.substringAfter(instruction, ",\"\"), ",\"\");
    for (int i = 0; i < sources.length; i++) {
        System.out.println(" sources = " + sources[i]);
        Logs.getInstance().setLog("sources = " + sources[i]);
    }

/****** Add to hashmap ******/

if (mnemonic == null || dest == null || sources == null || opcode == null) {
    throw new InstrNonIntALUException("Please enter instructions in the correct format");
}

hm.put("mnemonic", mnemonic);
hm.put("dest", dest);
hm.put("sources", sources);
hm.put("opcode", opcode);

// Print the 41 bits
String[] printSources = new String[source.length];
    for (int i = 0; i < sources.length; i++) {
        printSources[0] = sources[i];
    }
for(int i=0; i < printSources.length; i++){
    printSources[i] = StringUtilities.replace(printSources[i], "r", "r");
    printSources[i] = StringUtilities.replace(printSources[i], ",", ",");
    printSources[i] = StringUtilities.replace(printSources[i], "]", "]");
}

String destination = StringUtilities.replace(dest, "r", "r");
destination = StringUtilities.replace(destination, ",", ",");
destination = StringUtilities.replace(destination, "]", "]");

String qpVal = qp.toLowerCase();
qpVal = StringUtilities.replace(qp, "p", "p");
qpVal = StringUtilities.replace(qp, "P", "P");
System.out.println(""");
System.out.println("41 Bit INSTRUCTION IS " + opcode + printSources[0] + destination + qpVal);

System.out.println(""");
Logs.getInstance().setLog("41 Bit INSTRUCTION IS " + opcode + printSources[0] + destination + qpVal);

    return hm;
}

InstrIntegerALUException.java

/**
 * Written by Priyanka Shah at the Southern Connecticut State University, New Haven 2005.
 */
package edu.scsu.cs.Itanium.instructions.exceptions;

/**
 * Exceptions for IntegerALU Instructions
 */
public class InstrIntegerALUException extends Exception {

    public InstrIntegerALUException() {
        super();
    }
}
public InstrIntegerALUException(String msg) {
    super(msg);
}

InstrMemoryException.java

/*
 * Written by Priyanka Shah at the Southern Connecticut State University, New Haven 2005.
 */

package edu.scsu.cs.Itanium.instructions.exceptions;

/**
 * Exceptions for Memory Instructions
 */

public class InstrMemoryException extends Exception {

    public InstrMemoryException() {
        super();
    }

    public InstrMemoryException(String arg0) {
        super(arg0);
    }
}

InstrNonIntALUException.java

/*
 * Written by Priyanka Shah at the Southern Connecticut State University, New Haven 2005.
 */

package edu.scsu.cs.Itanium.instructions.exceptions;

public class InstrNonIntALUException extends Exception {

    public InstrNonIntALUException() {
        super();
    }
}

127
public InstrNonIntALUException(String arg0) {
    super(arg0);
}

/**
 * Written by Priyanka Shah at the Southern Connecticut State University, New Haven 2005.
 */
package edu.scsu.cs.Itanium.tools;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class ConsoleReader {

    private ConsoleReader() {
    }

    public static String getInput(String prompt) {
        String str = "";
        try {
            BufferedReader in =
                    new BufferedReader(new InputStreamReader(System.in));
            System.out.println(prompt);
            str = in.readLine();
        }
        catch (IOException e) {
            System.out.println("IOException "+ e);
            e.printStackTrace();
        }
        return str;
    }

    public static void setOutput(String prompt) {
        System.out.println(prompt);
    }

}
InstructionFileReader.java

/*
 * Written by Priyanka Shah at the Southern Connecticut State University, New Haven 2005.
 */

package edu.scsu.cs.Itanium.tools;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class InstructionFileReader {
    /**
     * Reads one line at a time from file and returns it to the caller
     */

    public static BufferedReader read(String path) {
        BufferedReader br = null;
        try {
            File f = new File(path);
            br = new BufferedReader(new FileReader(f));
            return br;
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        return br;
    }
}
InstructionsTypeXMLFileReader.java

/**
 * Written by Priyanka Shah at the Southern Connecticut State University, New Haven 2005.
 */
package edu.scsu.cs.Itanium.tools;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.HashMap;
import java.util.List;
import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

/**
 * @author Priyanka Shah
 * This is also a singleton class
 */
public class InstructionsTypeXMLFileReader {

    private static InstructionsTypeXMLFileReader instance =
        new InstructionsTypeXMLFileReader();

    private InstructionsTypeXMLFileReader() {
    }

    public static InstructionsTypeXMLFileReader getInstance() {
        return instance;
    }

    /**
     * @param instruction
     * - Takes entire line of the instruction
     * - Separates the mnemonic
     * - Finds out which instruction type is suited for the mnemonic
     * @return - returns the instruction type
     */
    public HashMap getInstructionType(String instruction) {
        HashMap hm = null;

    }
String instructionType = null;
String opcode = null;
String mnemonic = MnemonicSeperator.separate(instruction);

SAXBuilder builder = new SAXBuilder();
try {
    Document doc =
        builder.build(this.getClass().getClassLoader().getResourceAsStream("resources/instruction-formats.xml");
    Element root = doc.getRootElement();
    List instructionFormatList = root.getChildren();

    for (Iterator iter = instructionFormatList.iterator(); iter.hasNext();)
    {
        Element element = (Element) iter.next();
        //Add
        if (((element.getAttributeValue("name").equals(mnemonic))){
            hm = new HashMap();
            instructionType = element.getAttributeValue("type");
            opcode = element.getAttributeValue("opcode");

            hm.put("instructionType",instructionType);
            hm.put("opcode",opcode);
            return hm;
        }
    }
}

} catch (JDOMException e) {
    System.out.println("Jdom exception");
    e.printStackTrace();
} catch (IOException e) {
    System.out.println("Io exception");
    e.printStackTrace();
} catch (Exception e) {
    System.out.println("Exception");
    e.printStackTrace();
}

return hm;
}
MnemonicSeparator.java

/*
 * Written by Priyanka Shah at the Southern Connecticut State University, New Haven 2005.
 */
package edu.scsu.cs.Itanium.tools;
import org.apache.commons.lang.StringUtils;

/**
 * returns mnemonic
 */
public class MnemonicSeparator {

    public static String seperate(String instruction) {

        //No instruction entered
        if (instruction == null || instruction.equals("")) {
            System.out.println("No expression found in file. Try again");
            System.exit(1);
        }

        //******** OP ************
        int indexCloseBrack = 0;
        String qp = "p0";
        if (StringUtils.contains(instruction, ")") {
            qp = StringUtils.substringBetween(instruction, ", ", "");
            System.out.println("qp = " + qp);
            indexCloseBrack = StringUtils.indexOf(instruction, ")");
        }

        //******** Mnemonic ************

        boolean completerExists = false;
        String mnemonic = null;
        // Does a completer exist ?
        completerExists = StringUtils.contains(instruction, ".");
        // Is there a qp specified with completer?
        if (indexCloseBrack == 0 && completerExists) {
            mnemonic = StringUtils.substringBefore(instruction, ".");
            completerExists = true;
        }
        // Is there a qp specified and no completer?
        if (indexCloseBrack == 0 && !(completerExists)) {
            mnemonic = StringUtils.substringBefore(instruction, ");
        }
    }

}
}  //There is a qp & is there is a completer?
if (indexCloseBrack != 0 && completerExists) {
    mnemonic = StringUtils.substringBetween(instruction, "", ":");
    completerExists = true;
}

// Is there is no qp and no completer?
if (indexCloseBrack != 0 && (!completerExists)) {
    mnemonic = StringUtils.substringBetween(instruction, "","" );
}

mnemonic = StringUtils.deleteWhitespace(mnemonic);
System.out.println(" (Mnemonic seperator)-- " + mnemonic);

return mnemonic;
}
REFERENCES


