IA-64 Application Developer's Architecture Guide

The IA-64 architecture was designed to overcome the performance limitations of today's architectures and provide maximum headroom for the future. To achieve this, IA-64 includes an array of innovative features to extract greater instruction level parallelism including: speculation, predication, large register files, a register stack, advanced branch architecture, and many others. 64-bit memory addressability was added to met the increasing large memory footprint requirements of data warehouse, E-business, and other high performance server and workstation applications. IA-64 also has an enhanced system architecture with features that have been added for fast interrupt response and a flexible large virtual address mode.

To maintain the Intel promise of backward compatibility, the IA-64 architecture provides binary compatibility with the IA-32 instruction set. IA-64 processors support running unmodified IA-32 application binaries under an IA-64 operating system, as well as supporting IA-32 legacy operating systems and applications.

As IA-64 development systems will be available in the next few months, *the Intel IA-64 Application Developer's Architecture Guide* (ADAG) has been made publicly available to make it easier for all server and workstation software developers to become educated about the IA-64 application environment. It also opens up opportunities for any developer or interested party to begin the process towards obtaining a more in-depth understanding of IA-64. Such interested parties should look forward to more information on the IA-64 architecture and the system programming environment in the future. This document is intended to give an overview of the IA-64 application architecture described in the ADAG.

The publication of the *IA-64 Application Developer's Architecture Guide* is a key piece in a large program to enable IA-64 software for the first Intel IA-64 processor based systems deploying in 2H'00. Other key pieces in this program include the IA-64 fund, the application solution centers, and open source enabling. For more details on these programs and to get more information on IA-64 and what it means to you please consult the IA-64 website at http://developer.intel.com/info/IA-64.

Introduction

Computer architectures have continually evolved over the last few decades enabling the rapid advances in semiconductor process technology to meet the growing demands of increasingly more sophisticated operating systems and applications. As the semiconductor industry has consistently delivered on Moore's law of doubling the number of active devices on a given area of silicon every 18 months, computer architects have been challenged to design systems and software to deliver optimum performance on the ever growing resources at their disposal. In general, the answer has been an increasingly complex machine capable of executing multiple instructions in parallel, and O/S's capable of supporting multiple parallel processors. While performance has increased, the effective utilization of the available processor resources has been somewhat disappointing due to the inability of the hardware to efficiently parallelize instructions. Efforts to allow the compiler to parallelize instructions based on the capability to view thousands of lines of code have also proven difficult on traditional architectures. Additional limitations of today's architectures have been the inability to remove branch and procedure call overhead, and to effectively handle the increasing disparity between memory and processor speeds.

The goals for the initial architects of IA-64 were straightforward, although challenging. Design an architecture which can deliver industry leading performance, have the headroom to expand over the next few decades, while at the same time maintain full hardware compatibility with IA-32, thus giving IA-64 systems access to the world's largest base of software. To meet the demands of the high performance systems, IA-64 was also designed to support a large physical address space, an enhanced reliability and availability feature set, and an architecture which scaled well to 8, 16, 32 and greater way machines. The broad commitment from server and workstation vendors (both traditional RISC and Intel Architecture OEMs) and operating system vendors who have been working with Intel on IA-64 systems and software over the last couple of years is an endorsement of the fact that Intel has met its goals with this architecture.

IA-64 provides the architectural foundation to deliver the following goals.

Increase instruction-level parallelism – Today's ability to put millions of transistors on a single CPU die has enabled the development of superscalar architectures which are theoretically capable of executing many instructions in parallel. Today's processor architectures rely on hardware to try to identify opportunities for parallel execution - the key to higher performance. This process is inefficient not only because the hardware is often unable to identify these opportunities, but because it also requires valuable die area for instruction reordering that could be better used to do real work—like executing instructions. In the new IA-64 architecture model, the compiler analyzes and explicitly identifies parallelism in the software at compile time. This allows the most optimal structuring of the machine code to deliver maximum Instruction-Level Parallelism before the processor executes it rather than potentially wasting valuable processor cycles at run time.

Better manage memory latencies - Overcoming memory latency is another major performance challenge for today's processor architectures. Because memory speed is significantly slower than processor speed, the processor must attempt to load data from memory as early as possible to insure the data is available when needed. IA-64 supports a technique known as *speculation* which allows for aggressive preloading of data, even ahead of branches or possibly conflicting stores. Overcoming memory latency is important for almost all applications, but particularly for large server database programs which typically generate many cache accesses. Speculation therefore allows IA-64 to improve the performance of large server database applications over what is possible with traditional architectures.

Improve branch handling and management of branch resources - Simple decision structures or *code branches* are a severe performance challenge to today's heavily pipelined processors. In order to avoid stalling, traditional processors try to predict the result of a branch in advance. When predicted incorrectly, the branch will require restarting the instruction pipeline at a new instruction location. The overhead to purge the pipeline and restart execution at a new instruction location can quickly slow down the effective processing rate of even the fastest microprocessors. To address this problem and improve performance, the IA-64 architecture uses a technique known as predication to remove many branches and therefore remove the associated performance penalty. Predication is particularly important for large server applications and data mining software where branches have proven difficult to predict. Studies have shown that predication can increase processor performance in these areas by up to 40% over a similarly resourced processor without support for predication. IA-64 also defines an advanced branch prediction mechanism for branches which can not be removed.

Deliver exceptional floating point performance – IA-64 supports an enhanced floating point unit with a large floating point register file, and multiple FMAC (floating point multiply - accumulate) units. The first IA-64 processor is designed to deliver approximately six gigaflops/sec single precision through use of its four onboard FMAC units. Coupled with its extensive support for multimedia instructions and SIMD support, the IA-64 floating point architecture is designed to meet the most demanding workstation application requirements.

Reduce procedure call overhead – Typical programs today are very modular to reduce development timelines, take advantage of standard routines, and minimize validation efforts. However, procedure calls involve a significant performance penalty as the context of the machine has to be saved and restored. IA-64 implements a register stack engine which presents a logically infinite set of registers to the programmer, and allows for the processor to save and restore physical registers as required as a background task.

Provide massive resources - In order to fully support the ability to execute numerous instructions in parallel, the processor must provide massive hardware resources. IA-64 processors include 128 general-purpose integer registers, 128 floating-point registers, 64 predicate registers, and many execution units to ensure enough hardware resources for today's demanding software.

This document will cover how the IA-64 instruction set delivers the features to address the above goals, and their impact to applications and application programmers. It is intended as a high level overview, and the reader should consult the ADAG for additional technical information if interested. As this document is targeted for application developers, it will not cover IA-64 architectural system features such as protection, virtual memory, interrupt support, etc. or platform features such as reliability, availability, and scalability.

IA-64 Application Instruction Set Architecture Overview

Register Sets

The IA-64 architecture is *massively resourced* including a large number of general and special purpose registers which enable multiple computations to be performed without having to frequently save and restore intermediate data to and from memory*. Support for a large register set is key to IA-64 performance.* Below is a description of the primary register stacks visible to the application programmer.

General purpose registers (GR0-GR127): There are 128, 64-bit general-purpose registers that are used to hold values for integer and multimedia computations. Each of the 128 registers also has one additional NaT (Not a Thing) bit which is used to indicate whether the value stored in the register is valid. Execution of IA-64 speculative instructions can result in a register's NaT bit being set. R0 is read-only and always contains zero.

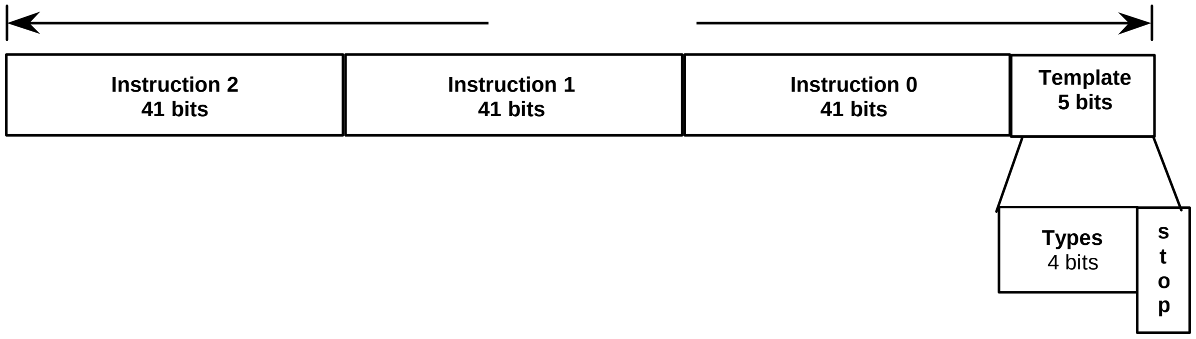
Floating point registers (FR0-FR127): There are 128, 82-bit floating-point registers that are used for floating-point computations. The first two registers, f0 and f1, are read-only and read as +0.0 and +1.0, respectively.

Predicate registers (PR0-PR63): There are 64, one-bit predicate registers that control conditional execution of instructions and conditional branches. The first register, p0, is read-only and always reads true (1).

Branch registers (BR0-BR7): There are 8, 64-bit branch registers (b0-b7) that are used to specify the target addresses of indirect branches.

IA-64 Instructions

IA-64 instructions are grouped into 128-bit *bundles*. A bundle consists of three instructions of 41 bits each, a *template* field of 5 bits. Each instruction occupies the first, second, or third syllable of a bundle.



IA-64 defines 6 types of instructions, and four types of execution units (instruction, memory, floating point, and branch). Below table defines the instruction types and the execution units on which they may be executed. More detail on the description can be found in the AIG.

Instruction Type	Description	Execution Unit Type
A	Integer ALU	I-unit or M-unit
I	Non-ALU integer	I –unit
M	Memory	M-unit
F	Floating Point	F-unit
B	Branch	B-unit
L	Long Immediate	I-unit

The first four bits of the template field specify the types of instructions in the bundle. As an example an MMI encoding would indicate that the three instructions are memory, memory, and integer. The template field is used by the processor to quickly decode the bundle, and issue the instructions to the appropriate execution units.

The last bit in the template field, the stop bit, indicates whether or not the next instruction bundle can be executed in parallel with the current bundle.

Expressing Parallelism

IA-64 requires an assembly program (typically generated by a compiler) to explicitly indicate groups of instructions, called instruction groups, that have no register read after write (RAW) or write after write (WAW) register dependencies. *The ability of the compiler to explicitly parallelize instructions is key to IA-64 performance.* Instruction groups are delimited by stops in the assembly source code. Since instruction groups have no RAW or WAW register dependencies, they can be issued without hardware checks for register dependencies between instructions. Both of the following examples show two instruction groups separated by stops (indicated by double semicolons):

```
ld      r1=[r5] ;;              // First group
add     r3=r1,r4                // Second group
```

A more complex example with multiple register flow dependencies is shown below:

```
ld      r1=[r5]                 // First group
sub     r6=r8,r9 ;;             // First group
add     r3=r1,r4                // Second group
st      [r6]=r12                // Second group
```

All instructions in a single instruction group may issue in parallel dependent on the resources available.

Note that bundle boundaries have no direct correlation with instruction group boundaries as instruction groups can extend over an arbitrary number of bundles. Instruction groups begin and end where s-bit are set in assembly code, and dynamically whenever a branch is taken or an s-bit is encountered.

Memory Access and Speculation

IA-64 provides memory access only through register load and store instructions and special semaphore instructions. IA-64 also provides extensive support for hiding memory latency via programmer-controlled *speculation*. Speculation allows a program to move instructions including loads past data or control dependencies that would normally limit code motion. *The ability of the compiler to speculate loads and remove memory latencies is key to IA-64 performance*. The two kinds of speculation are called control speculation and data speculation. This section summarizes IA-64 speculation.

Control Speculation

Control speculation allows loads and their dependent uses to be safely moved above branches. Support for this is enabled by special NaT bits that are part of to integer registers and by special NatVal values for floating-point registers. When a speculative load causes an exception, it is not immediately raised. Instead, the NaT bit is set on the destination register (or NatVal is written into the floating-point register). Subsequent speculative instructions that use a register with a set NaT bit propagate the setting until a non-speculative instruction checks for or raises the deferred exception.

For example, in the absence of other information, the compiler for a traditional architecture cannot safely move the load above the branch in the sequence below:

```
(p1) br.cond.dptk L1                    // Cycle 0
    ld    r3=[r5] ;;                    // Cycle 1
    shr   r7=r3,r87                     // Cycle 3
```

Supposing that the latency of a load is 2 cycles, the shift right (shr) instruction will stall for 1. However, by using the speculative loads and checks provided in IA-64, two cycles can be saved by rewriting the above code as shown below:

```
        ld.s    r3=[r5]                 // Earlier cycle
                                        // Other instructions
(p1)    br.cond.dptk L1 ;;              // Cycle 0
        chk.s   r3,                     // Cycle 1
        shr     r7=r3,r87               // Cycle 1
```

Data Speculation

Data speculation allows loads to be moved above possibly conflicting memory references. Advanced loads exclusively refer to data speculative loads. Review the order of loads and stores in this IA-64 assembly sequence:

```
st      [r55]=r45             // Cycle 0
ld      r3=[r5] ;;            // Cycle 0
shr     r7=r3,r87             // Cycle 2
```

IA-64 allows the programmer/compiler to move the load above the store even if it is not known whether the load and the store reference overlapping memory locations. This is accomplished using special advanced load and check instructions:

```
ld.a    r3=[r5]               // Advanced load
                              // Other instructions (possibly hundreds)

st      [r55]=r45             // Cycle 0
ld.c    r3=[r5]               // Cycle 0 - check
shr     r7=r3,r87             // Cycle 0
```

Note that shr instruction in this schedule could issue in cycle 0 if there were no conflicts between the advanced load and intervening stores. If there were a conflict, the check load instruction (ld.c) would detect the conflict and reissue the load.

Predication

Branches can be a primary performance limiter, and IA-64 architecture supports a concept known as predication to remove many branches and their associated performance penalty.

Predication is the conditional execution of an instruction based on a qualifying predicate. A qualifying predicate is a predicate register whose value determines whether the processor commits the results computed by an instruction. The values of predicate registers are set by the results of instructions such as compare (cmp) and test bit (tbit). When the value of a qualifying predicate associated with an instruction is true (1), the processor executes the instruction, and instruction results are committed. When the value is false (0), the processor discards any results and raises no exceptions. Consider the following C code:

```
if (a) {
        b = c + d;
}
if (e) {
        h = i + j;
}
```

This code can be implemented in IA-64 using qualifying predicates so that branches are removed. The IA-64 example shown below implements the C expressions without branches:

```
        cmp.ne  p1,p2=a,r0            // p1 <- a != 0
        cmp.ne  p3,p4=e,r0 ;;         // p3 <- e != 0
(p1)    add     b=c,d                 // If a != 0 then add
(p3)    sub     h=i,j                 // If e != 0 then sub
```

The ability of the compiler to remove branches through predication is key to IA-64 performance.

IA-64 Support for Procedure Calls

Calling conventions normally require callee and caller saved registers which can incur significant overhead during procedure calls and returns. To address this problem, a subset of the IA-64 general registers are organized as a logically infinite set of stack frames that are allocated from a finite pool of physical registers.

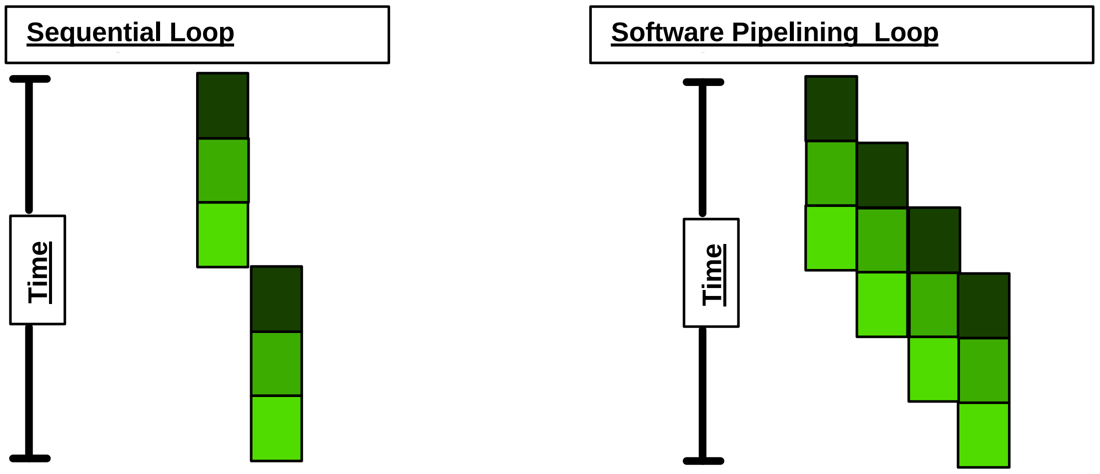
Stacked Registers

Registers GR0 through GR31 are called global or static registers and are not part of the stacked registers. The stacked registers are numbered GR32 up to a user-configurable maximum of GR127:

A called procedure specifies the size of its new stack frame using the alloc instruction. The called procedure can use this instruction to allocate up to 96 registers per frame shared amongst input, output, and local values. When a call is made, the output registers of the calling procedure are overlapped with the input registers of the called procedure, thus allowing parameters to be passed with no register copying or spilling. The global registers (GR0-GR31) are also visible in both the called and calling procedure.

Support for Software Pipelining

Compilers sometimes try to improve the performance of loops by using a technique known as *software pipelining*, in which the loop is unrolled and can be pipelined into the processor execution unit.



However, unrolling is not always effective on traditional architectures due to the lack of intrinsic support for this technique which can result in code expansion and increased cache misses.

To maintain the advantages of pipelining while overcoming these limitations, IA-64 provides architectural support for software pipelining *without* unrolling. In addition to dedicated instructions which support software pipelining, IA-64 also utilizes the concept of a rotating register stack. Rotating registers enable succinct implementation of software pipelining with predication, and support software pipelining of loops. Rotating registers are rotated by one register position each time one of the special loop branches is executed. Thus, after one rotation, the content of register N will be found in register N+1. The general purpose registers, the floating point registers, and the predicate registers all support register rotation and software pipelining. These registers, along with the special supported software pipelined loop branches, allow the compiler to generate very compact code for software loops, and thus greatly increase performance of software loops.

Register Stack Engine

Management of the register stack is handled by a hardware mechanism called the Register Stack Engine (RSE). The RSE moves the contents of physical registers between the general register file and memory without explicit program intervention. This provides a programming model that looks like an unlimited physical register stack to the application. The RSE saves and restores registers in a background mode, decreasing the overhead of procedure calls and returns seen on traditional architectures.

Summary

The IA-64 instruction set architecture and resources allow for the efficient instruction level parallelism, the effective memory latency management techniques, and the branch and procedure handling which allow IA-64 based systems to reach their performance goals The *IA-64 Application Developer's Architecture Guide* is available today with a detailed explanation of the application architecture for those who require it. This guide allows for a detailed understanding of the architecture and how it enables higher performance.

For more information on IA-64 or to download the IA-64 Application Developer's Architecture Guide, please visit the Intel IA-64 website at http://developer.intel.com/info/ia-64.