Overview of the IA-64 Architecture

Karthik Swaminathan

Table Of Contents

<u>**Abstract & Introduction:**</u>

For the purpose of this assignment, I have chosen to document one of the most recent developments in the microprocessor industry. The architecture known as EPIC (Explicitly Parallel Instruction Computing) as applied particularly to IA-64, Intel's 64-bit microprocessor architecture, is going to be the object of my discussion. Throughout the course of this paper I shall be using the terms IA-64 and "Itanium" (the first processor in this series) synonymously.

Existing architectures are based on an out-of-order execution model, which require increasingly complex hardware mechanisms. Performance limiters such as branches and memory latency increasingly impede these processors.

Intel's IA-64 processor architecture is designed to overcome these limitations. In addition, the IA-64 architecture provides additional performance headroom and scalability needed for future computation-intensive applications. The IA-64 architecture features a revolutionary 64-bit instruction set architecture (ISA), which applies a new processor technology called EPIC. Jointly defined with Hewlett-Packard Company, EPIC embodies a set of advanced computer architecture techniques such as explicit parallelism, predication, and speculation. These techniques enable IA-64 processors to execute more instructions per clock cycle to deliver superior performance relative to today's out-of-order based RISC processors. The EPIC technique eliminates much of the dependency checking and grouping logic that consumes an increasingly large portion of advanced RISC and x86 processors. EPIC's flexible grouping mechanism solves VLIW based processors' two fatal flaws: excessive code expansion and lack of scalability.

This paper is organized into the following sections:

- Architecture Design – Key features of the IA-64 architecture and their explanation.
- Performance Comparison to RISC/CISC processors.
- Comparison to Sun Microsystems' SPARC series – This comparison is done at a high level and not at a feature level.
- Conclusion.
- Bibliography.
- List of Figures that highlight some of the architectural details.

<u>Architecture Design:</u>

I was able to locate some information about the IA-64 architecture and its programming model. In this section I shall try to condense, what in my opinion are, the most of the critical pieces of the IA-64 architecture. Please refer to **Figure 1**, in the List of Figures, for a diagram that depicts the high level "Itanium" architecture and exposes the synergy between software and hardware.

1. Support for two Operating System Environments:

IA-64 handles two operating system environments within a single architecture: The IA-32 environment that supports 32 bit operating systems and IA-64 (native) environment that supports 64 bit operating systems.

The IA-32 system environment, which includes full support for the IA-32 instruction set, can be used for IA-32 Protected Mode, Real Mode, and Virtual 8086 Mode applications and operating systems. This environment makes the IA-64 architecture backward compatible with the "Pentium" series of processors.

The IA-64 system environment can be used to run IA-32 Real Mode, Protected Mode, and Virtual Mode applications if supported by the 64-bit operating system. It naturally includes support for 64 bit applications. This way this is implemented is elaborated in the next.

2. Ability to handle IA-32 Instruction sets in the IA-64 operating environment:

The IA-64 operating environment allows the execution of full 32-bit binaries compiled on IA-32 systems provided the required platform and firmware support exists on the system.

This operating environment also gives the application developer the ability/power to **intermix older x86 instruction sets** with the native instruction sets. Moreover, "Itanium" can convert x86 instructions into internal native-mode instructions before executing them. Further, it can accept these native mode instructions directly from memory thereby eliminating the inefficiencies of hardware translation that asserts the fact that the **processor will be optimized for native-mode execution rather than x86 mode execution**.

From several technical publications, it can be gleaned that "Itanium" will allow x86 and IA-64 instructions to commingle at all levels of the memory hierarchy, including the on-chip cache, off-chip cache, and main memory. This method allows the chip to maintain a single system interface that knows how to fetch instructions into the CPU. A **"mode" bit** is present on the chip that will direct the instructions to either the x86 decoder or a native decoder. The presence of this bit avoids the need for encoding at the instruction level. Using this mechanism, the load can be placed as early as possible in the code, as long as the address can be computed. If the data is never checked, no exception will be triggered; if an exception occurs when the data is needed, the exception will be recognized in the load's original "home block". Speculative loads thus provide the compiler with maximum flexibility to hide cache latency.

Three special instructions and interruptions are defined to transition the processor between the IA-32 and IA-64 instruction sets. Also refer to **Figure 2**, in the List of Figures, for an illustration.

- jmpe (IA-32 instruction) – Jump to an IA-64 target instruction, and change the instruction set to IA-64.
- br.ia (IA-64 instruction) – IA-64 branch to an IA-32 target instruction, and change the instruction set to IA-32.
- rfi (IA-64 instruction) – Return from interruption. It has been defined to return either an IA-32 or IA-64 instruction when resuming from an interruption.
- Interruptions transition the processor to the IA-64 instruction set for all interruption conditions.

The jmpe and br.ia instructions provide a low overhead mechanism to transfer control between instruction sets. These primitives are typically incorporated into "thunks" or "stubs" that implement the required call linkage and calling conventions to call dynamically or statically linked libraries.

3. Unique Instruction Format aka "Bundling":

The IA-64 instructions use a unique format that allows the compiler to direct hardware execution without severely bloating the software. A single 128-bit aligned container called "bundle" contains three 41-bit IA-64 instructions along with the 5-bit "template" information about the "bundle".

The "template" field, in my opinion, seems to be the most interesting sequence of bits in the "bundle". This field stores, what are referred to as, architectural stops. These stops indicate that one or more instructions before the stop may have certain kind of dependencies with one or more instructions after the stop. Thus it indicates whether the instructions in the "bundle" can be executed in parallel or if one or more instructions must be executed serially, due to register dependencies. The template also indicates whether the bundle can be executed in parallel with the following "bundle". In addition to storing the location of stops the "template" field also specifies the mapping of instruction slots to execution unit types. **"Bundles" can be chained to form Instruction Groups** of any length. **Figure 3**, in the List of Figures, depicts an IA-64 "bundle".

"Bundles" are ordered from lowest to highest memory address. Instructions in "bundles" with lower memory addresses are considered to precede instructions in "bundles" with higher memory addresses. **The byte order of each "bundle" in memory is little-endian** (the template field is contained in byte 0 of a bundle). Within a "bundle" instructions are ordered from instruction slot 0 to instruction slot 2 as specified in **Figure 3**.

4. Instruction set features & Instruction Sequencing:

There are basically 6 instruction types – ALU (A), Integer (I), Memory (M), Floating-point (F), Branch (B), and Long/Extended (L). All instructions in the instruction set of the "Itanium" implementation are 41 bits in length. The leftmost 4 bits (40:37) of each instruction are the major opcode. **Opcode assignments** for each instruction type **are given in the "IA-64 application developers guide"** that has been published by Intel. A basic IA-64 instruction has the following syntax:

[qp] mnemonic [.comp] dest=srcs ,where,

"qp" specifies a qualifying predicate register. When the value of the register is true, the instruction executes, its results are committed, and any exceptions that occur are handled as usual. At a false value nothing is committed and no exceptions are handled.

"mnemonic" uniquely identifies an IA-64 instruction by name.

"comp" specifies one or more instruction completers, which indicate optional variations on a base instruction mnemonic.

"dest" represents the destination operand(s) and **"srcs"** represent source operands. Most IA-64 instructions have at least two input source operands.

Instruction execution consists of four phases: Reading the instruction from memory (**fetch**), Read architectural state, if necessary (**read**), Perform the specified operation (**execute**), and finally Update architectural state, if necessary (**update**). If the instructions in an "instruction group", defined in the previous section, meet all the resource dependency requirements then the behavior of a program will be as though each individual instruction is sequenced through these phases in the order listed afore.

The instruction sequencing rules, given below, prescribe the order of a phase of a given instruction relative to any phase of a previous instruction.

- Since there is no a priori relationship between the "fetch" of an instruction and the "read", "execute", and "update" of any previous instruction, the "sync.i" and "srlz.i" (synchronize and serialize) instructions can used to enforce a sequential relationship.

- Every instruction will behave as if its "read" occurred after the "update" of memory and ALAT state of all instructions from the previous instruction group and all instructions within the same instruction group. Moreover, within an instruction group, every instruction will behave as though its read of the register state occurred before the update of the register state by any instruction; prior or later. This eliminates WAR data dependencies.

- All instructions have unit latency and instructions on opposing sides of a "stop" are separated by at least one unit of latency.

Apparently, the "Itanium" hardware puts the instruction "bundle" through a **10-stage in-order hardware pipeline**. **I could not locate much information on the pipeline except in an old presentation by Mr. Harsh Sarangpani**, Principal Architect of the IA-64 micro-architecture. Intel has not yet released any technical documents on the working of the pipeline and hence I shall not be taking about it further.

Suffice it to note that the **front-end pre-fetch** and **fetch** stages of the pipeline will allow the pre-fetching of upto 8 bundles (storing them in a decoupled buffer upon fetch). This buffer allows the front-end to fetch even when the backend is stalled. The buffer also hides instruction cache misses and branch bubbles thereby augmenting performance.

The **issue logic**, a hardware implementation, incorporated into the pipeline issues the instruction to the CPU. This piece of hardware is designed in a manner that will promote binary compatibility across current and future IA-64 implementations. This issue logic makes an IA-64 processor more complex than a pure VLIW design, but the ability to have binary compatible processors is well worth the effort of incorporating this extra logic. The issue logic in the IA-64 is much less complicated than out-of-order superscalar processors, however

I could not find documentation on supported addressing modes. I assume that IA-64 will be supporting the same **addressing modes** as its IA-32 predecessors (extrapolated, however, for handling the Very Large Memory model). These addressing modes, as a refresher, are **absolute**, **register indirect**, **based, indexed**, **based indexed with displacement**, **based with scaled index**, and **based with scaled index and displacement**.

5. Registers:

The "Itanium" processor and in general the IA-64 family is going to be massively resourced. Registers are a critical component in a processor's performance. With this in mind the Intel architects have endowed "Itanium" with two distinct register groups viz. **Application State Registers** and **System State Registers**. These groups are further subdivided into several **register name spaces**. I have strived to describe them below; highlighting what I thought was interesting information.

Along with the Application State Registers is intertwined the functionality of the **Register Stack Engine**, and **Register Rotation**. I have devoted the next section to the Register Stack (and its usage and manipulation by the RSE) but will defer the definition and application of Register Rotation till I describe the close co-ordination between the compiler and the processor.

At this point I would like to bring to attention the fact that for all of the **application level resources** (eg. Application State Registers) **the processor guarantees serialization** in order to observe data and execution dependencies between subsequent instructions. This is done using "srlz.i" (serialize instruction), "rfi" (return from instruction), or "srlz.d" (serialize data). "srlz.i" and "rfi" serializes the application's processor resource that was modified by an instruction in the current instruction group and ensures that the changes are visible to the subsequent instruction group. "srlz.d" comes into play when data memory needs to be serialized as well. In the case of **system level resources** (eg. System State Registers), in order to eliminate hardware serialization overhead, **explicit instruction and data serialization is required of the executing software**.

5.1. Application State Registers and their usage:

- **General Registers** – 128 64-bit registers are made available as a central resource for all integer and integer multimedia computation. Numbered GR 0-GR 127, these are available at all privilege levels. Each of these has an extra 65th bit that acts to record the deferred speculative exceptions. This bit is called the **NaT (Not a Thing)** bit.

  The GR's are partitioned into two subsets: GR 0- GR 31 are called the **static register set** and GR 32 – GR 127 are termed as the **stacked register set**. GR 0 is always 0 (when sourced as an operand) and writing to it causes an Illegal Operation Fault. The **Register Stack Engine (RSE)** uses the stacked register set to accelerate process performance. This mechanism is called the **“Register Stack Mechanism”**. I shall explain this further in the **Register Stack & Renaming sub-section**.

- **Floating Point Registers** – 128 81-bit registers are provided to handle intensive FLOPS (floating point operations). As above FR 0- FR 31 are static and FR 32- FR 127 are rotating and are accessible at all privilege levels. An extra 82nd bit provides the **NaTVal (Not a Thing Value)**. FR 0 is always +0.0 and FR 1 is always +1.0 and they cannot be written to. FR 8 – FR31 will handle IA-32 FLOPS and multimedia instructions.

- **Predicate Registers –** 64 1-bit registers holds the result of IA-64 compare instructions and will be accessible at all privilege levels. As above, PR 0-15 are static and PR 16-63 are rotating.

- **Branch Registers –** 8 64-bit registers are employed to hold all the IA-64 branching information. Numbered BR 0-7, they are available at all privilege levels. **These are only used for specifying the target address for “indirect branches”**. “IP relative branches” and “Long branches” use a 21-bit and 60-bit displacement respectively to calculate the target address relative to the IP of the bundle containing the branch instruction.

- **Instruction Pointer –** The IP holds the address of the bundle that contains the current executing instruction. The IP can be read directly but cannot be written to directly. The 4 LSB of the IP are always 0 since IA-64 instruction bundles are 16 bytes long and are consequently 16 byte aligned. Also, IP will hold the 32-bit virtual linear address of the executing IA-32 instruction.

- **Current Frame Marker –** The CFM is a 38-bit register that holds the state of the current stack frame. The CFM cannot be directly read or written. I shall explain this further in the **sub-section on the Register Stack**.

- **Application Registers –** 128 64-bit application registers are available for application-visible processor functions for both the IA-32 and IA-64 instruction sets. These can be accessed by either an M (Memory) or an I (Integer) execution unit.

  Of these the most interesting ones, IMHO, are AR 0-7, BSP, BSPSTORE, PFS, RNAT, FPSR, LC, EC & ITC. **AR 0-7** are called **Kernel Registers** and they make kernel data structures available to application programs. I shall explain the **BSP** (Backing Store Pointer) and **BSPSTORE** (Backing Store Pointer for Memory Stores) and **RNAT** (RSE NaT collection Register), and **PFS** (Previous Frame Marker) further when describing the **Register Stack**.

  FPSR (Floating Point Status Register) controls traps, rounding mode, precision control, flags, and other control bits **for IA-64 FLOPS**. I have explained the use of **ITC** in the System State Registers. **LC** (Loop Counter) is used in counted IA-64 loops. **EC** (Epilog Counter) is a 6-bit register is used for counting in the final stages (Epilog Stages) of IA-64 modulo-scheduled-loops, which I have described in the **sub-section on Processor to Compiler communication**.

- **Performance Monitor Data Registers (PMD) –** A set of performance monitoring registers are used to contain sampled performance data from within an application. These can be configured for access at all privilege levels and can also be secured to restrict access. They are available for both IA-32 and IA-64 instructions. The number of these registers is variable (i.e. implementation specific).

- **User Mask –** This is a 5-bit subset of the PSR that is accessible to IA-64 applications programs. The bits are:
  Bit 0 – rv – reserved.
  Bit 1 – be – when 1 (0) enables "Big Endian" ("Little Endian") accesses
  Bit 2 – up – when 1 enables access to PMDs
  Bit 3 – ac – Enable/Disable Alignment Checks on IA-32/64 data memory references.
  Bit 4 – mfl – Becomes 1 when FR 2-31 are written to. This is a sticky bit.
  Bit 5 – mfh – Becomes 1 when FR 32-127 are written to. This is also a sticky bit.

- **Processor Identifiers –** CPUID 4 is the most interesting of this set and a value of 0 in this register the implementation of all the defined features.

5.2. System State Registers and their usage:

- **Processor Status Register (PSR)** – This is a 64-bit register that maintains control information for the currently running IA-64 or IA-32 process.

  The PSR is divided into four overlapping regions (User Mask Bits, System Mask Bits, the lower half, and the entire PSR). Of these, the User Mask Bits of the PSR (PSR(5:0)) are visible to the application programs. All other parts of the PSR require "privileged access".

  The processor implicitly serializes the **User Mask manipulation instructions (**sum, rum, & mov psr.um=gsr[32]**)**. All other PSR manipulation instructions require explicit serialization by the software.

- **Control Registers (CR) -** This register name space contains 128 64-bit registers that capture the state of the processor on an interruption, enable system-wide IA-64 or IA-32 features, and specify global processor parameters for interruptions and memory management. **Only "mov" to/from** operations can be performed to these registers at a privilege level 0 (the highest) and **reads** from any of these registers must be **data serialized with prior writes** to the same register.

- **Interrupt Registers -** Control Registers CR 64-CR 81 are used for interrupt handling. These include the LID (Local Identifier Register), TPR (Task Priority Register), IVR (External Interrupt Request Vector), IRR's (External Interrupt Request Registers), LRR's (Local Redirection Registers), and PMV (Performance Monitoring Vector). They are used to prioritize and deliver external interrupts, program external interrupt vectors for processor-internal interrupt sources such as interval timer, performance monitoring, corrected machine check, and also to assign vectors and redirect internal interrupts.

- **Interval Timer Facilities -** Two 64-bit registers (ITC and ITM) are provided for privileged and non-privileged use and as a time base for performance measurements.

  The Interval Time Counter (ITC) and Interval Timer Match (ITM) register support fine-grained time stamps and elapsed time notification.

The ITC (Application Register AR 44) is a free-running 64-bit counter that counts up at a fixed relationship to the processor clock. The ITC counting rate is not affected by power management mechanisms. **The ITC can be read at any privilege level iff PSR.si is zero; otherwise only at privilege level 0.**

The ITC can be read by the IA-32 rdtsc (read time stamp counter) instruction. When the value in the ITC is equal to the value in the ITM (Control Register CR 1) an Interval Timer Interrupt is raised. Once the interruption is taken by the processor and serviced by software, the ITC may not necessarily be equal to the ITM. **The ITM is accessible only at privilege level 0.**

- **Debug Breakpoint Registers (DBR/IBR) -** 64-bit Data and 64-bit Instruction Breakpoint Register pairs (DBR, IBR) can be programmed to fault on reference to a range of virtual and physical addresses generated by either IA-64 or IA-32 instructions. The minimum number of DBR register pairs and IBR register pairs is 4 in any implementation. On some implementations, a hardware debugger may use two or more of these register pairs for its own use.

- **Performance Monitor Configuration/Data Registers (PMC/PMD) -** Multiple performance monitors can be programmed to measure a wide range of user, operating system, or processor performance values. Performance monitors can be programmed to measure performance values from either the IA-32 or IA-64 instruction set. The minimum number of generic PMC/PMD register pairs in any implementation is 4.

- **Banked General Registers -** 16 of the 32 static Registers (GR 16- GR 31) are available as immediate temporary storage for register context when operating low-level interruption handlers (e.g. speculation and TLB miss handlers). Of these, GR 24 – GR 31 are volatile as their contents are preserved only when PSR.ic is 1. Upon interruption, the processor switches the 16 general-purpose registers (GR16 to GR31) to register bank 0 while preserving register bank 1's contents.

  Operating systems should ensure that IA-32 and IA-64 application code is executed within register bank 1. If IA-32 or IA-64 application code executes out of register bank 0, the IA-32 or IA-64 application register state will be lost on any interruption. During interruption processing the operating system uses register bank 0 as the initial working register context.

  When PSR.bn is 1, bank 1 for registers GR16 to GR31 is selected; when 0, bank 0 for registers GR16 to GR31 is selected. Banks are switched when an interruption selects bank 0, or "rfi" switches to the bank specified by PSR.bn, or "bsw" switches to the specified bank.

- **Region Registers (RR) -** Eight 64-bit region registers specify the identifiers and preferred page sizes for multiple virtual address spaces. Their usage will be described further in the **Memory Sub-Section** of this paper.

- **Protection Key Registers (PKR) -** At least sixteen 64-bit protection key registers contain protection keys and read, write, execute permissions for virtual memory protection domains. Their usage will also be described further in the **Memory Sub-Section** of this paper.

- **Translation Lookaside Buffer (TLB) -** Holds recently used virtual to physical address mappings. The TLB is divided into Instruction (ITLB), Data (DTLB), Translation Registers (TR) and Translation Cache (TC) sections. **TR**s are software-managed portions of the TLB and the processor directly manages the **TC**s section of the TLB. At the risk of sounding repetitive, the TLB usage will also be described further in the **Memory Sub-Section** of this paper.

- **Machine Specific Registers (MSR) -** This register file is privileged and all aspects of its content and operation is machine specific (i.e. it is used for test and diagnostics). This register file is accessed using the **indirect "mov" instruction**. A Privileged Operation fault is raised if the privilege level is not zero when this register is accessed.

6. Register Stack:

IA-64 avoids unnecessary spilling and filling of registers at procedure call and return interfaces through compiler-controlled renaming. The GRs (32-127), FRs (32-127), and the PRs (16-63) are the affected registers by this process. These are also known as the stacked subset of registers. The static subset of these register name spaces must be explicitly saved by software convention while the **RSE** takes care of saving and re-instating the stacked subset. The register stack operation is disabled during IA-32 execution mode.

I shall explain the mechanism with respect to the GRs. This principle just needs to be extended as is to the FRs and PRs. When a procedure is executing in the CPU, it is assigned a set of stacked GRs. This is known as the **stack frame**. The **CFM.sol** stores the **size of the local area** and the **CFM.sof** stores the **size of the entire frame**. The local area is the scratch pad for the current procedure. (**CFM.sof - CFM.sol**) gives us the **size of the output area** in the stack frame.

When the procedure is called, the RSE assigns a stack frame with only an output area. This area contains all the actual parameters passed to the procedure and also has space to hold the return parameter. Moreover this area overlaps with the output area of the callee. This overlap permits all parameter passing and inter-procedure communication to take place entirely through registers. The procedure can then allocate a local area using the **"alloc"** instruction that specifies immediates for the size of locals and the size of frame.

The caller's CFM is saved to the PFM (previous frame marker) field in the PFSR (Previous Function State Register) before the call is actually made. **The RSE ensures that when a stack frame is assigned to a callee the starting register (which is the first output register of the caller) is always (re)named GR32**.

When a return-type branch is executed, CFM is restored from PFM and the register renaming is restored to the caller's configuration. The PFM is procedure local state and must be saved and restored by non-leaf procedures. The CFM is not directly accessible in application programs and is updated only through the execution of calls, returns, "alloc", and "clrrrb".

When procedure calling is interleaved at more than 1 level and the register resources start to exhaust, then the RSE takes the onus of spilling the contents of some of the registers to the memory. IA-64 calls this memory store the "backup store". The "backup store" is handled by two ARs viz. **BSP (Backup Store Pointer)** and the **BSPSTORE (Backup Store Pointer for**

memory stores). The BSPSTORE contains the address in memory where the current frame must start spilling its contents. The BSP contains the address in memory where next spill of any of the previous frames' registers must occur. BSP can be read whereas BSPSTORE is inaccessible to application programs. The exhaustion of register resources can lead to stack overflow (when the callee has no registers available and the pipeline stalls) – when a spill is imminent – or stack underflow (when the caller's registers have to be restored) – fill is imminent. Both spill and fill occur at the best opportunity possible (independent of the calling and called procedures).

7. Memory and Addressing:

Objects in memory and I/O occupy a common 63-bit physical address space that is accessed using byte addresses. The accesses can be done either using direct physical addressing or through virtual addresses mapped to the physical address space. The current implementation, page table formats to be more specific, limit the mapping of virtual addresses to 50 bits of physical addresses.

Physical Addressing is enabled for instruction references when PSR.it is 0, for data references when PSR.dt is 0, and for stack references when PSR.rt is 0. Instruction references to unimplemented physical addresses result in an "Unimplemented Instruction Address Trap". Similarly, data references to unimplemented physical addresses result in an "Unimplemented Data Address fault" and memory references to an unpopulated address range results in an "Asynchronous Machine Check Abort".

Virtual Addressing model in IA-64 is fundamentally a flat 64-bit linear virtual address space. 64-bit GRs are used as pointers into the address space. **Figure 4** shows the mapping of a virtual address into a physical address. **Virtual Address** is composed of the **Virtual Region Number (VRN)**, **Virtual Page Number (VPN)**, and the **Page Offset (PO)**. The **VRN** is computed from the **upper 3 bits** of the 64-bit virtual address. This gives us a possible set of 8 regions, **each region containing 2^{61} bytes**. These **3-bits** are used to **select one of the 8 "Region Registers" (RRs)**, defined in the previous section. The least significant bits of the 64-bit virtual address forms the **PO** and the remaining bits form the **VPN**.

The RR contains a 24-bit value that chooses one of a possible 2^{24} virtual address spaces. Thus the IA-64 can consequently access 8 such address spaces. The 24-bit RR can also be considered as the upper 24 bits of an 85-bit global address space in a **"Single Address Model"**. This 24-bit value also serves as the unique address space number for the region (**region identifier aka RegionID**). Each Region is split into several pages (page level addressing); varying in size from 4K to 256M bytes. The RegionID is hashed with the VPN to give a key that is used to search the **TLB**, defined in the previous section, to obtain a **Physical Page Number (PPN)**.

The processor maintains **two** architectural **TLBs**, the instruction TLB (**ITLB**) and the data TLB (**DTLB**). As their name indicates each services translation requests for IA-64 and IA-32 instruction and data memory references. The **DTLB** also services translation requests from the RSE and the **VHPT (Virtual Hash Page Table)**.

On a memory reference, the VRN bits select a RegionID from one of the 8 RRs. The TLB is then searched for a translation entry with a matching VPN and RegionID value. If a matching translation entry is found, the entry's PPN is concatenated with the PO to form the physical address. Matching translations are qualified by page-granular **privilege level and access**

right checks and optional protection domain checks by verifying that the translation's key (a 24-bit value) is contained within a set of PKR's, defined in the previous section, and that read, write, and execute permissions are granted.

If the required translation is not resident in the TLB, then the processor may search the VHPT structure in memory. This is done based on the value contained in the **Page Table Address (PTA)** register. The value in this register defines both the base address of the VHPT (PTA.base) and the size of the VHPT ($2^{PTA.size}$). If the entry is found in the VHPT, the processor will install it in the TLB; a TLB miss fault will be raised otherwise to request the operating system to supply the translation. Once the operating system installs the translation in either the TLB or the VHPT, the faulting instruction will be restarted and execution resumed.

IA-64 supports 32-bit virtual addressing in three models: **zero-extension**, **sign-extension**, and **pointer swizzling**. In zero-extension the software ensures that the upper 32-bits of the virtual address are always 0's. This implies that only virtual region 0 is accessible (i.e RR0). Similarly in sign-extension, the upper 32-bits of the virtual address are made equal to bit 31. This implies that the 32-bit address space is split into two halves contained within 2^{31} bytes of virtual regions 0 and 7. Pointer swizzling is shown in **Figure 5**. This scheme divides the 32-bit virtual address space into 4 sections that are spread into 2^{30} bytes of virtual regions 0 to 3 within the 64-bit virtual address space. In pointer swizzling, mappings within each region do not necessarily start at offset 0 since the upper 2-bits of a 32-bit address serve as both the virtual region number and an offset within each region. Also, virtual address bits 62 & 61 do not participate in address addition and hence may lead to some regions being larger than 2^{30} bytes (due to the addition of a 32-bit offset and lack of a carry into bits 62 & 61).

8. Processor to Compiler interaction (& vice versa):

The IA-64 architecture provides mechanisms, such as instruction templates, branch hints, and cache hints to enable the compiler to communicate compile-time information to the processor. In addition, IA-64 allows compiled code to manage the processor hardware using run-time information. These communication mechanisms are vital in minimizing the performance penalties associated with branches and cache misses.

Every memory load and store in IA-64 has a 2-bit cache hint field in which the compiler encodes its prediction of the spatial and/or temporal locality of the memory area being accessed. An IA-64 processor can use this information to determine the placement of cache lines in the cache hierarchy. This leads to better utilization of the hierarchy since the relative cost of cache misses continues to grow.

The compiler also uses techniques such as speculation, predication, and software pipelining to improve IA-64's performance by exposing Instruction Level Parallelism (ILP).

Speculation is employed by the compiler to issue an operation speculatively iff the operation is statistically frequent enough that the probability that it will require recovery is small, and issuing the operation will expose further ILP enhancing optimization. Speculation is further divided into **Control** and **Data Speculation**. In **control speculation** the compiler executes an operation before a branch that guards it; exceptions, if they occur, are handled in the instruction's home block. For that to occur, the compiler leaves a check operation in the home block of the instruction that activates the recovery code upon exception. In **data speculation**, a memory load is performed before a memory store that may potentially alias with it. Data speculative loads are also referred to as **“Advanced Loads”** and the **ALAT (Advanced Load Address Table)** is used for this purpose. Just as in control speculation, the compiler puts a check, in the load’s home block, to handle address overlap exceptions and to start recovery.

Control Speculation will translate the code sequence:

```
if (a > b) load(ld_addr1, target1)
else load(ld_addr2, target2)
```

To

```
/* control speculation moves instruction before branch */
sload(ld_addr1, target1)
sload(ld_addr2, target2)

/* checks are inserted, in original instructions’ home blocks */
/*  to handle exceptions */
if (a > b) scheck(target1, recovery_addr1)
else scheck(target2, recovery_addr2)
```

Data Speculation will translate the code sequence:

```
store(st_addr, data)
load(ld_addr, target)
use target
```

To

```
/* data speculation moves load before store */
load(ld_addr, target)

/* checks are inserted, in original instructions’ home blocks */
/*  to handle exceptions */
store(st_addr, data)
acheck(target, recovery_addr)
use target
```

Predication is employed by the compiler to remove branches resulting in larger basic blocks and the elimination of associated mispredicts. This is even more highlighted given the fact that IA-64 has a deep 10-stage in-order pipeline. Predication also simplifies compiler optimizations by converting control dependence into data dependence. The PRs (Predicate Registers) are used for handling predicates.

Predication converts a code such as:

```
if (a > b) c = c+1
else d = d+1
```

To

```
/* result of comparison stored in PRs pT and pF */
pT, pF = compare (a,b)

/* a control dependence on (a > b) is reduced to a */
/* data dependence on compare (a,b) */
if (pT) c = c+1
if (pF) d = d+1
```

Software Pipelining is a technique that is employed by the compiler that seeks to overlap loop iterations in a manner that is analogous to hardware pipelining of a functional unit. Each iteration is partitioned into stages with 0 or more instructions in each stage. The number of cycles between the start of successive iterations is called the Iteration Interval (II). Each stage of a pipelined iteration is II cycles long. IA-64 uses **Modulo Scheduling**, a form of software pipelining to handle floating point/multimedia loops. In modulo scheduling II is a constant and every iteration of the loop has the same schedule. Software pipelined loops have **three phases**: **prolog**, **kernel**, and **epilog**.

During the **prolog phase**, a new loop iteration is started every II cycles to fill the pipeline. During the first cycle of the prolog, stage 1 of the first iteration executes. During the second cycle, stage 1 of the second iteration and stage 2 of the first iteration execute and so on. This happens till the pipeline gets full i.e. if the loop has 4-stages, then the pipeline has the fourth iteration in its first stage, third iteration in its second stage and so on. This takes the loop into its **kernel phase**. In this phase, a new loop iteration is started every II cycles and one is completed at the same time. Once there are no new iterations to be started, the pipeline starts to drain and the loop enters its **epilog phase**. The EC (Epilog Count Register) is used for the final stages of the Modulo Scheduled loops.

This concurrent multiple execution of iterations traditionally required loop unrolling and software register renaming. Since IA-64 hardware allows the renaming of registers to provide each iteration with its own set of registers **(a concept also termed as Register Rotation)**, Modulo Scheduled loops can be chosen over traditional loop unrolling methods, which cause code expansion and ask for the scheduling of multiple loop bodies.

9. Interruption Handling:

Interruptions are classified as IVA-based interruptions and PAL-based interruptions depending on how they are serviced. IVA-based interruptions are serviced by the operating system. These interruptions are vectored into the IVT (Interruption Vector Table) pointed to by the IVA control register (CR2). PAL-based interruptions are serviced by the PAL firmware, system firmware, and possibly the operating system. These interruptions are vectored through a set of hardware entry points directly into the PAL firmware.

Interruptions could be of 4 types: **Aborts, Interrupts, Faults**, and **Traps**. **Aborts** are PAL-based interruptions that could be synchronous or asynchronous with respect to the instruction stream. It may cause the processor to suspend the instruction stream at an unpredictable location with partially updated register and memory state. **Aborts** come in two flavors: Machine Check Aborts (MCA - due to the detection of a hardware failure) and Processor Reset (RESET – during power-on or reset request).

When an external or independent entity like an I/O device, a timer event, or another processor requires attention, **Interrupts** are triggered. They are asynchronous with respect to the instruction stream and come in 3 flavors viz. **Initialization Interrupts** (INIT – A PAL-based interrupt that signals the processor having received an initialization request), **Platform Management Interrupts** (PMI – A PAL-based interrupt that signals the processor having received a request to perform functions such as platform error handling, memory scrubbing, or power management), **External Interrupts** (INT – An IVA-based interrupt that signals the processor receiving requests from external devices). **External Interrupts** can be non-maskable (NMI) or external controller interrupts (ExtINT).

Faults are triggered when an IA-32 or IA-64 instruction requests an action, which cannot and should not be carried out. These are synchronous with respect to the instruction stream and are IVA-based. The processor completes state changes that have occurred in instructions prior to the faulting instruction and subsequent instructions have no effect on the machine state. **Traps** indicate that the instruction that just executed requires system intervention. These too are synchronous with respect to the instruction stream and are IVA-based.

In Comparison to RISC/CISC processors & areas of use:

Frequency (MHz) will no longer be the primary decision point in the choice of CPU architecture. Instead, instructions per clock cycle (IPC) will become a more important measure of system efficiency and power as calculated with the frequency.

With the IA-64, the users must focus on IPC and the clock frequency to arrive at a quick and dirty performance comparison number vis-à-vis the traditional RISC/CISC processor. The users must also take into account that the **IA-64 hardware is simpler because the compilers that target the IA-64 processors will be designed to simplify and optimize the instructions specifically for that target.**

EPIC enables "Itanium" to handle more IPC's than previous microprocessors, and to feed these instructions to multiple on-chip functional units for execution on every clock cycle. The architects of IA-64 have designed considerable performance headroom into the architecture allowing them the leeway of cramming in more IPC's in the future releases. The **traditional architectures are limited** on performance and scalability because the architecture feeds instructions and data to the CPU too slowly to fully stretch the ability to make data parallel. Hence, **in today's processors, the CPU has the additional task of parallelizing instructions on the fly** – a task that consumes valuable real estate and processing power on the CPU – **aka "Pure Hardware Pipelining".**

Today's processors attempt to guess which branch to execute after a decision point. To avoid predicting incorrectly the current RISC/CISC processors attempt to guess more intelligently ("branch prediction"), run both instruction streams in different pipelines ("speculative execution"), or execute one or both branches when the CPU is underutilized ("out-of-order execution"). Most of the time these techniques donot fully leverage CPU parallelism and can result in "mispredicts" (executing the wrong instruction stream) that cost up to 40% in CPU performance. **Predication** – IA-64's method, explained earlier in the paper – takes a new approach to the problem by looking ahead in the compiler before the CPU even receives the code. Predication can remove branches wherever possible and IA-64 can also tell the CPU more about each branch. **If predication fails the CPU can revert back to its ancestral techniques**: **owing to more real estate "Itanium" can do this even better than the current processors**. According to a study based on popular software benchmarks (ISCA '95, S. Malhlke et al.), **predication can, on average, reduce the number of branches by more than 50% and reduce mispredicts by as much as 40%**.

As a 64-bit architecture, "Itanium" will be especially appropriate for new and high-end applications. "Itanium" is expected to deliver performance and scalability beyond older architectures in certain key application areas, particularly the following:

- **Business Intelligence** – "Itanium" is expected to extend support for terabyte-and-beyond data warehousing and data mining that need VLM. Moreover, **Level 3 cache speeds CPU processing of large datasets by reducing time-to-load from memory for a larger amount of data versus disk I/O speeds**.

- **The Internet** – The Internet involves both multimedia and exceptionally rapid variation and increase in transactions (queries and OLTP). Thus processors need to scale rapidly for both decision support and e-commerce. "Itanium" also provides specific features for Internet applications viz. Security (encryption algorithm support) and streaming media (floating point performance improvements and support for applying a single instruction to multiple larger scale data streams).

- **New Line of Business Applications** – “Itanium’s” large register set allows for greater parallelism in small-size data thus increasing performance in many key LOB-application operations.

- **Technical/Scientific applications** – The “Itanium” processor is built to handle intensive computations; scientific or otherwise.

<u>Comparison to Sun Microsystems' SPARC:</u>

In this section I strive to compare the "Itanium" processor with Sun's SPARC albeit at a high level. Benchmark performance results between SPARC and "Itanium" are still not available (at least I was not able to get a hold of it). The comparison has been performed based on the following criteria.

- **Technology:** Sun's SPARC is a RISC processor while "Itanium" is an EPIC processor. This implies that all the comparison details mentioned in the prior section will hold.

- **Availability:** SPARC is available in the market today while "Itanium" has been delayed till the latter part of this year.

- **Market share:** Since SPARC has been in the 64-bit market for 3 years it has already established a niche for itself while Intel has a totally unproven ground to tread.

- **Compatibility on ISA:** Sun has retained its V9 ISA in its SPARC processor; maintaining binary compatibility at the same time. Intel is changing its ISA to achieve Instruction Level Parallelism (ILP) and the IA-32 bit compatibility.

- **OS:** Sun is retaining its robust SunOS operating systems' suite and application base. The "Itanium" on the other hand requires a whole new OS, new applications, and a new middleware. My understanding of Intel's strategy is that they are moving away from the traditional Intel-Microsoft pact. They want IA-64 to host a wide range of 64 bit OS's and hence they are wooing 64-bit compiler writers to target the IA-64 platform/architecture. This also explains their deal with Hewlett-Packard in coming up with the EPIC "formula".

- **Compilers:** While the UltraSPARC series uses existing and proven compilers, tweaked to optimize performance, "Itanium" requires completely new compilers. Moreover Sun is in the business of writing compilers while Intel is not in that ball game. "**Itanium**" **relies heavily on compiler to processor communication** (in the sense of software pipelining, branch predication, and speculation) and hence **shifts a lot of optimizing and efficiency burden to the compiler vendors while SPARC still relies a lot on hardware optimizations**. **This is going to be an empirical battle for cost to performance tradeoff**.

- **Scalability:** Sun makes changes at the micro-architecture level to enhance scalability to a level of upto 1000 processors in a single system. For Intel, and the IA-64, this is once again an untreaded ground.

- **Testing:** Sun is much ahead of Intel in this arena too. They entered the market when they tested their first 64-bit architecture in 1995. Intel is yet to conduct one; lagging behind due to complexity issues.

- **Expertise and Experience:** Clearly, Sun's technical expertise extends well beyond the realm of processors and into the realm of OS and system design. In Intel's case, however, there seems to be an unclear coordination between the OS manufacturer, system designer, and processor designer.

Conclusion:

All in all, this venture is going to be an uphill battle for Intel but with their fundamentals set right they should be able to get the “Itanium” implementation into the market later this year.

I have tried as best as possible, with the set of documentation available, to paint a picture of the IA-64 architecture. This paper certainly does not carry a comprehensive explanation of the IA-64 architecture but I have tried to cover as many critical pieces as possible in a term paper.

<u>Bibliography:</u>

[1] Intel® Itanium^{TM} Processor Micro-architecture Overview: by **Harsh Sharangpani**
- Obtained from Intel’s website. This overview was presented to the microprocessor forum in October of 1999.

[2] Inside Itanium^{TM} – A white paper by the Aberdeen Group, based in Boston
- Obtained from Intel’s website.

[3] Intel® IA-64 Architecture Software Developer’s Manual
- Obtained from Intel’s website.

[4] The IA-64 Instruction Set Architecture Guide
- Obtained from HP’s developer resource website.

[5] IA-64 Instruction set goes beyond traditional RISC, VLIW: by **Linley Gwennap**
- Obtained from MDROnline’s website.

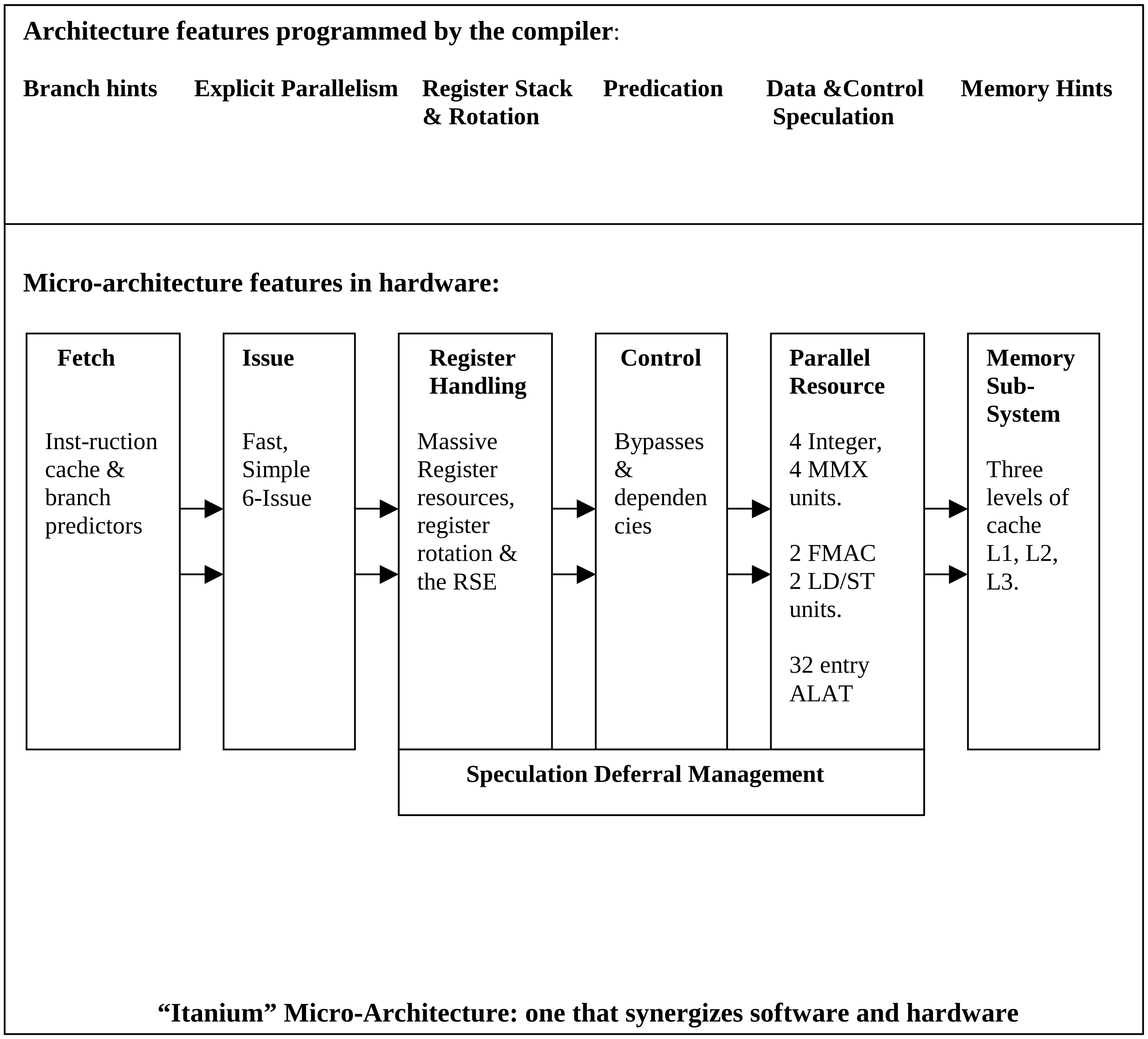
[6] A white paper on IA-64 and the competitive advantages for HP’s customers: by **David Scott**
- Obtained from HP’s website.

[7] Computer Architecture, A Quantitative Approach: by **Patterson & Hennesey**
- A very handy reference for the details of computer architecture.

[8] Other press releases and articles related to IA-64.
- Results of endless hours of web searches.

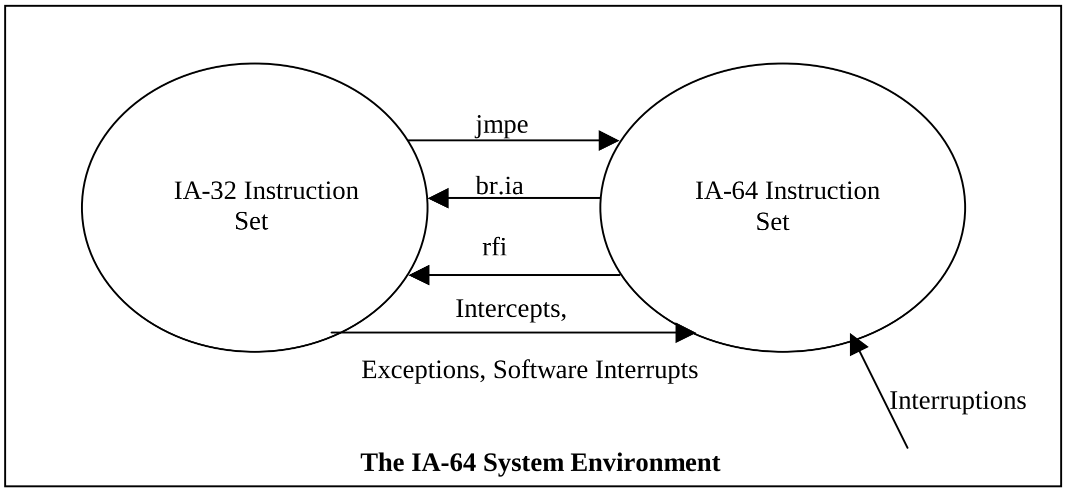
List of Figures:

Figure 1:


Architecture features programmed by the compiler:
Branch hints
Explicit Parallelism
Register Stack & Rotation
Predication
Data &Control Speculation
Memory Hints
Micro-architecture features in hardware:
Fetch
Inst-ruction cache & branch predictors
Issue
Fast, Simple 6-Issue
Register Handling
Massive Register resources, register rotation & the RSE
Control
Bypasses & dependen cies
Parallel Resource
4 Integer, 4 MMX units.
2 FMAC 2 LD/ST units.
32 entry ALAT
Memory Sub-System
Three levels of cache L1, L2, L3.
Speculation Deferral Management

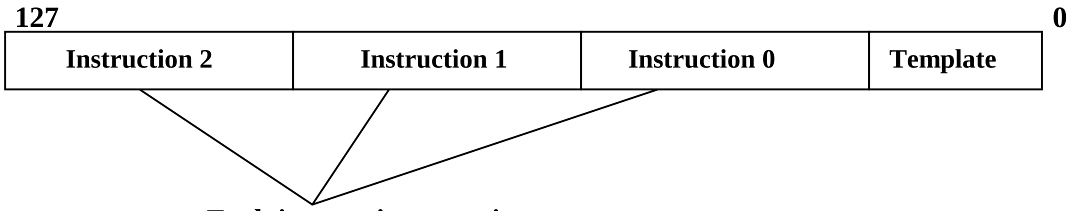
"Itanium" Micro-Architecture: one that synergizes software and hardware

Figure 2:



The IA-64 System Environment

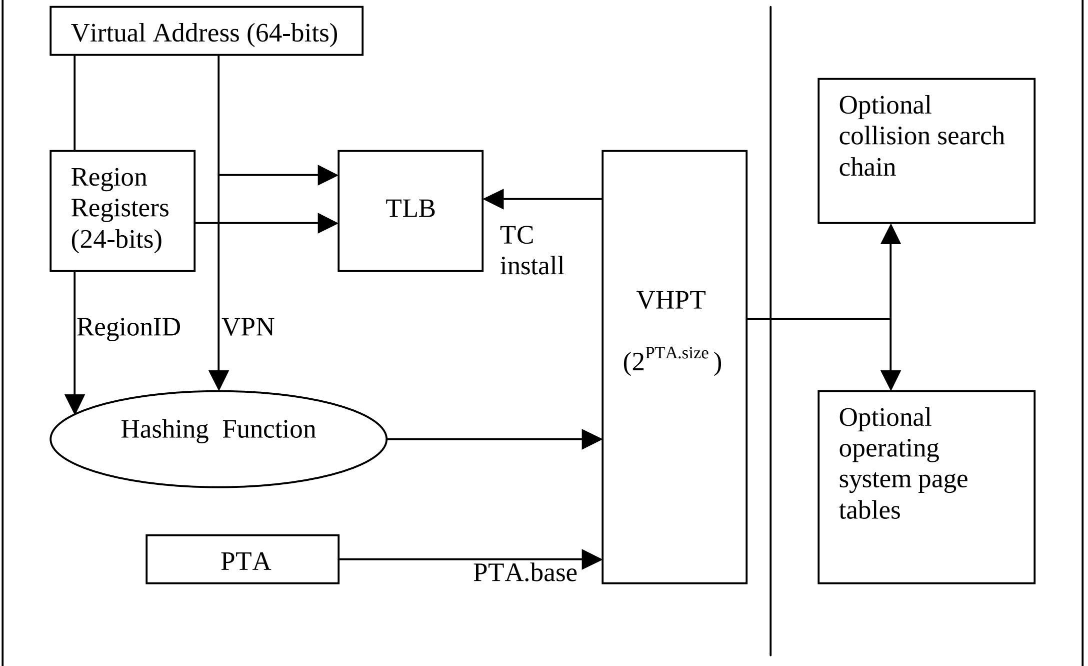
Figure 3:



Each instruction contains:

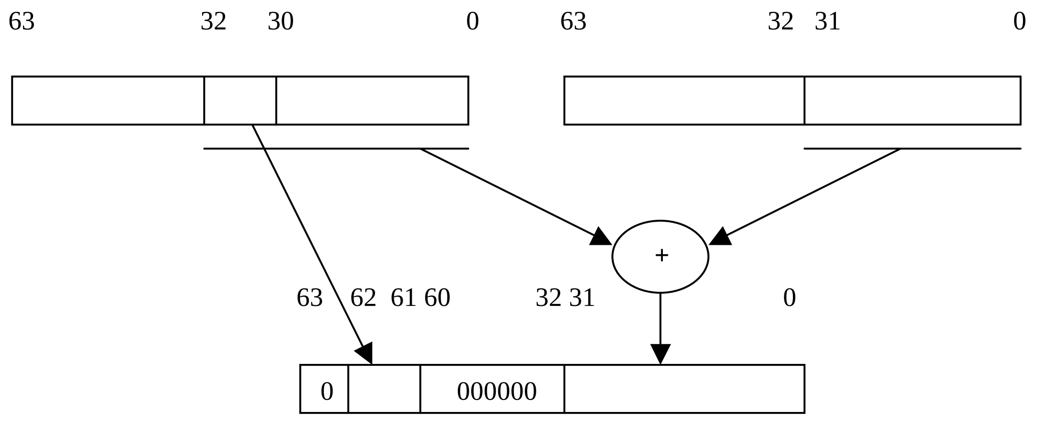
1. Opcode
2. Predicate Register (6 bits)
3. Source 1 (7 bits)
4. Source 2 (7 bits)
5. Destination Register (7 bits)
6. Opcode extension/ branch target/ misc.

Figure 4:


Virtual Address (64-bits)
Region Registers (24-bits)
TLB
TC install
VHPT
($2^{PTA.size}$)
Optional collision search chain
Optional operating system page tables
RegionID
VPN
Hashing Function
PTA
PTA.base

Virtual Memory Addressing Scheme

Figure 5:



Pointer Swizzling