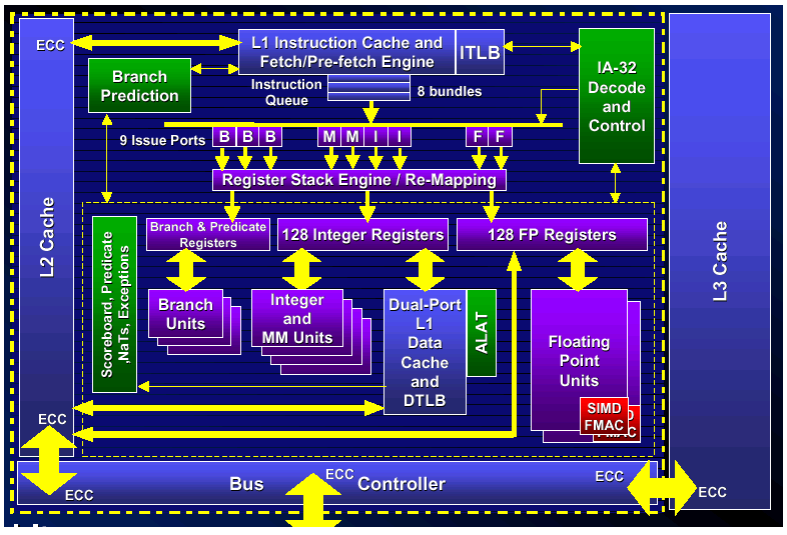
Cache, Virtual Memory Architectures
Modern Processors

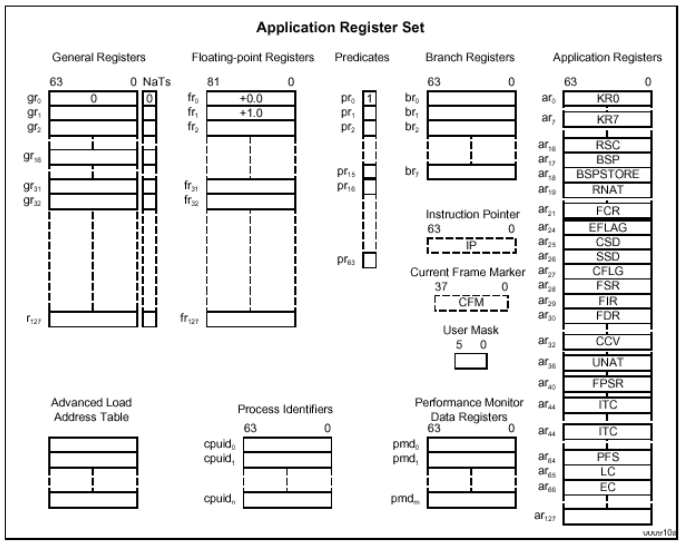
- Will look at Cache, Virtual memory architecture for the IA-64
- IA-64 ISA is successor to the Merced (Pentium 4), which was the successor to the Pentium 3/2/1.
  - 64-bit architecture, all registers 64-bits wide
  - 128 General Registers, 128 Floating Point Registers
  - G0-G31 are global registers, G32-G127 are part of the "Register Stack" where a dynamic number of them can be allocated as part of procedure call/return and be visible to only that procedure (similar to Sparc register windows).
  - Superscalar, maximum issue of 6 instructions per clock
  - Supports both speculative branching and speculative loading
- *Itanium* is the first implementation of the IA-64 ISA.

Itanium Block Diagram



Register Set (Integer RF has 8 read ports, 6 Write ports)

Figure 3-1. Application Register Model



Itanium Caches

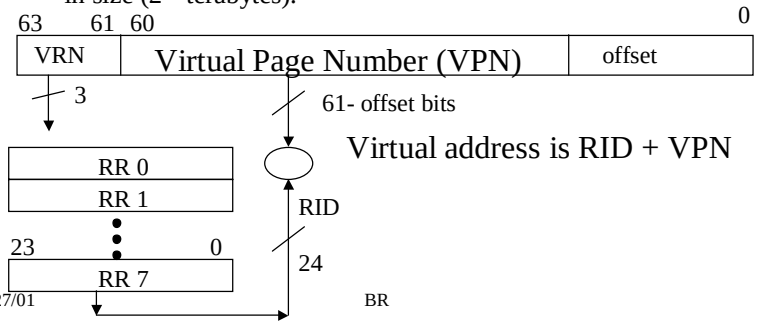
- L1 Data Cache (128 bits in one read)
  - 16Kbytes, 4-way set associative, write through, no write allocate with 32-byte lines. "No Write Allocate" means that on a write miss, the missed block is NOT loaded into the cache.
  - Can sustain 2 loads, 2 stores, or 1 load and 1 store per clock (dual ported cache!!! Needed for superscalar operation!).
  - Integer mem op hits in L1 have a 2 cycle latency to most operations. Floating point mem ops bypass this cache.
- L1 Instruction Cache
  - 16 Kbytes, 4-way set associative with 32-byte lines
  - 1 Read = 128 bits = 3 instruction 'bundle' + 2 parity bits (1 inst = 41 bits)
- L2 Unified Cache
  - 96Kbyte, 6 way set associative, write back, write allocate (on write miss, load missed block into cache), 64 byte line.
  - Two ports, can sustain two memory ops/clock or one line fill.
  - Integer mem op hits have 6 cycle latency, Floating point mem op hits have 9 cycle latency.

Itanium Caches (continued)

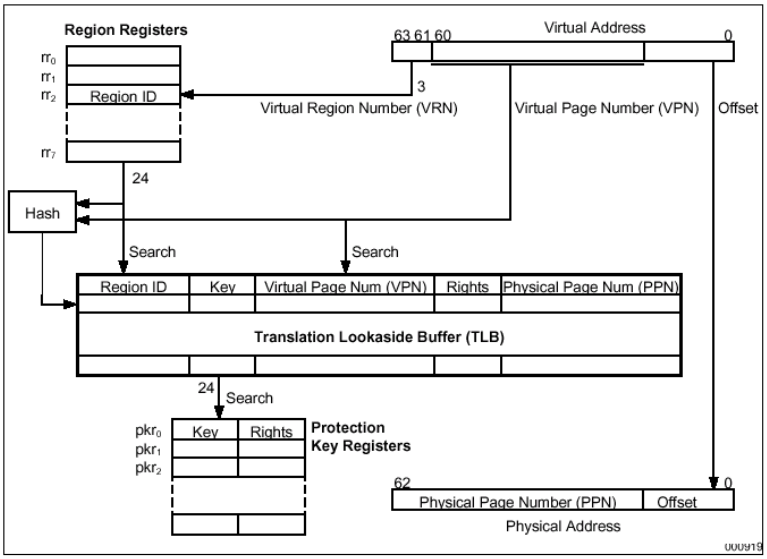
- L3 Unified Cache (off chip, on package)
  - Size, organization varies with package. First versions have 4 MB.
  - Integer mem op hits have 21 cycle latency, Floating point 24 cycles.
- Latency is high even for hits!!!!
  - Architecture has an Advanced Load Address Table (ALAT) that supports speculative loads. Helps compiler schedule loads to work around latencies.
  - Also has explicit instructions for use by the compiler to move blocks between cache hierarchies.
  - Will discuss both of these later
- L1 cache has parity bit, L2/L3 caches have 1 bit error correction, 2 bit error detection.
- All caches are PHYSICAL caches, virtual to physical translation takes place before cache access.

IA-64 Virtual Memory

- Address are 64 bits
  - Upper 3 bits (Virtual Region Number) use to select 1 of 8 Region Registers (RR) that contains a Region ID (RID).
  - The contents of Region Registers managed by OS. Typically, one region register is constant and is assigned to the OS, while the other 7 are owned by the current task.
  - A Region Register is 24 bits.
  - RID gives 2^{24} address spaces (16 Million), each space 2^{61} bytes in size (2^{21} terabytes).



Virtual to Physical Translation



More on Regions

- One region usually always reserved for OS
- Other 7 regions for a task, usually split into:
  - text (code)
  - stack
  - heap
  - static data
  - ????
- One task will basically see a flat 64-bit address space.

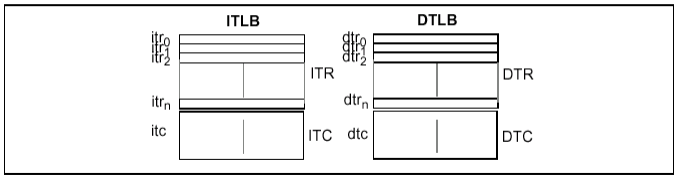
Address Translation

- TLBs for both Data and Instruction
- TLB tag contains both RID and VPN and is matched against incoming RID and VPN
  - The TLB can contain entries from different Regions (different RIDs).
  - If the OS changes the contents of a RID register, then all TLB entries that correspond to that RR must be flushed.
- Itanium (1st implementation of IA-64) has L1 DTLB (data TLB) and L2 DTLB
  - L1 DTLB has 32 entries, fully associative
  - L2 DTLB has 96 entries, fully associative (L2 DLTB needed because floating point mem accesses bypass L1 cache).
- ITLB (instruction TLB) has 64 entries, fully associative
- Page size configurable as 4K, 8K, 16K, 256K, 1M, 4M, 16M, 64M, and 256M.

Page Pinning

- Some pages need to be 'pinned', i.e, never swapped out to disk
  - interrupt vector pages, root page tables
- Each TLB contains 8 Translation Registers (TRs) that are managed by software independent of the normal cache entries (TCs)
  - The page referenced by a TR is essentially pinned in memory – it will only be invalidated if the software managing the TR purges the TR contents

Figure 4-3. TLB Organization



Virtual Hash Page Table (VHPT)

- The hardware page table entry lookup mechanism on the IA-64 is called the Virtual Hash Page Table (VHPT)
  - The page table is mapped into virtual memory because the size of the page table would be too large to keep in physical memory
  - Use of the VHPT search mechanism on a TLB miss is optional, the OS can disable the VHPT and do the search (page table walk) entirely in software
- The VHPT search mechanism can be configured as
  - a linear page table (one PTE per VPN)
  - as a hashed page table (multiple VPNs mapped to same PTE, collisions resolved in software via collision chain)

Itanium Virtual Memory Implementation

- Implements 54 bit Virtual address (3 bit VRN, 51 VPN)
- Implements 44 bit physical address
- Assume a page size of 16 Kbytes (2^{14}). How many Virtual Pages?
  - Ignore VRN, specifies Region ID. For linear page table searching by VHPT walker, RID is ignored (linear table per region)
  - $2^{51} / 2^{14} = 2^{37}$ PTEs. For linear table format, each PTE is 8 bytes, so page table size = 2^{40}, or 1 Terabyte!!??!!
- Obviously, can't reserve 1 Terabyte for the page table and there is no need to statically allocate the page table.
- The OS can build the page table **dynamically.**

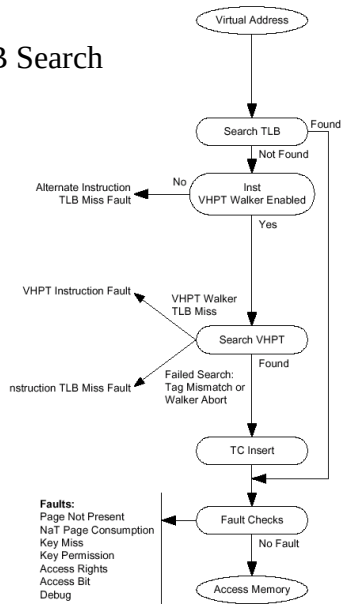
Building a Page Table dynamically

- The entire page table is kept in Virtual memory
- When task is initialized, storage requirements for code, static data, stack are known.
  - Pages for PTEs for this storage is allocated by OS in Virtual Memory.
  - Program requests for dynamic data storage goes through OS (malloc, sbrk, etc.), so page table information is built for these pages as they are allocated.
- Making PTE's stored in Virtual Memory means that pages of PTEs can be swapped just like data/code pages.

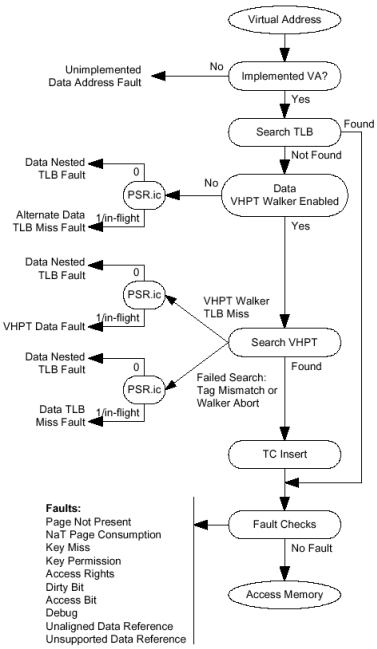
Invalid Address Protection

- What happens if a program generates an erroneous address for which there is no PTE?
  - Call this address BAD_VMADDR
- VHPT calculates the address of the PTE for BAD_VMADDR based upon either the linear mapping or hash function
  - Call this address PTEBAD_VMADDR
- Since PTEBAD_VMADDR is virtual, goes through the TLB
  - generates a MISS in the TLB
  - When a TLB MISS for a PTE is generated by the VHPT, this MISS is handed to the OS, which then detects the invalid virtual address.

TLB Search



3/ **Instruction TLB VHPT Search**



BR **Data TLB VHPT Search**

Some Fault Descriptions

- VHPT Instruction/Data Fault.
  - Raised when there is an additional TLB miss when the VHPT walker attempts to access the VHPT. Typically used to construct leaf table mappings for linear page table configurations.
- Instruction/Data TLB Miss. Raised when
  - cannot locate the required VHPT entry
  - VHPT walker is not implemented
  - reference is to a non-supported VHPT preferred page size
  - ill-formed PTE (reserved fields used, etc.)
- Miss handlers are essentially software walkers
- The VHPT walker can be disabled. Is only a performance enhancing mechanism for locating PTEs after a TLB miss.

A Numerical Example

Assume a 64K byte page size (2^{16}) with a VMA size of 48 bits without the region bits.

What is the address of the PTE for the address given that each PTE is 8 bytes using a linear mapping?

Let VMA = 0x 07FF1A002000

Offset = 0x2000 (lower 16 bits).

VPN = 0x07FFF1A00

VMA of PTE = VPN * 8 = 0x0000 07FFF1A00 * 2^3 =
= 0x0000FFF8D000

PTE attributes defines Cache Policy For Page

Table 4-10. Virtual Addressing Memory Attribute Encodings

Attribute	Mnemonic	ma	Cacheability	Write Policy	Speculation	Coherent[a] with respect to
Write Back	WB	000	cacheable	write back	non-sequential & speculative	WB, WBL
Write Coalescing	WC	110	uncacheable	coalescing		not MP coherent[b]
Uncacheable	UC	100		non-coalescing	sequential & non-speculative	UC, UCE
Uncacheable Exported	UCE	101				
Reserved[c]		001				
Reserved		010 011				
NaTPage	NaTPage	111	cacheable	n/a	speculative	n/a

Basically can mark as cacheable or uncacheable.

'Coherence' refers to multi-processor cache coherence.

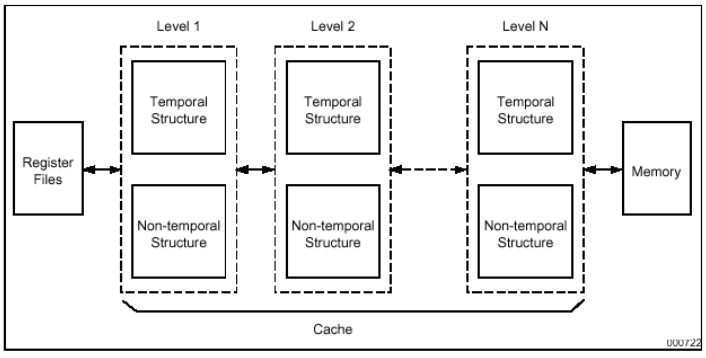
Coalescing means delaying writes so that they can be written as a block of writes to consecutive locations for higher performance. This means that the writes can be performed out of order.

Deferred Speculative Exceptions

- The IA-64 supports speculative memory load operations
  - Because of latencies, loads can be moved by compiler many instructions ahead of where data is used
  - If moved before a STORE, the load become data speculative because the store may alter the memory value
  - If moved before a BRANCH, the load becomes control speculative because the branch may not be executed
- What happens if an exception (like a page fault) is raised for control speculative loads?
  - Typically deferred – wait until we know if it is needed
  - A bit called the NaT (Not a Thing) bit is used with each General Register to track if this is register is part of a deferred speculative exception
- The NaTPage policy causes all control speculative loads to this page to be deferred (i.e aborted)
  - Will discuss speculative loading in more detail later.

Cache Support for Streaming Data Operations

Figure 4-4. Memory Hierarchy



Special cache instructions allow support for two views of data:

a. Data with both temporal and spatial locality

b. Data with spatial locality only (streaming data)

Both types of data can exist in cache at the same time.

Locality Hints, Implicit Prefetching

Table 4-18. Locality Hints Specified by Each Instruction Class

Mnemonic	Locality Hint	Instruction Type		
		Load	Store	Ifetch, Ifetch.fault
none	Temporal, level 1	x	x	x
nt1	Non-temporal, level 1	x		x
nt2	Non-temporal, level 2			x
nta	Non-temporal, all levels	x	x	x

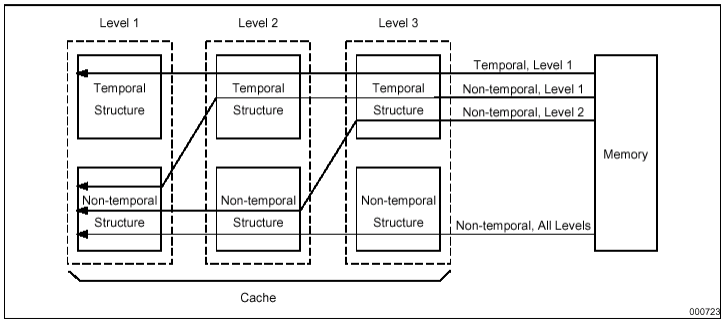
The above hints are for data references. Instructions are assumed Temporal, Level 1.

Future implementations of the IA-64 may have separate caches for streaming and non-streaming data. Data is allocated between the caches according to the hints (see next page).

The IA-64 implements implicit prefetching for loads/stores that use a post-increment addressing mode (ld r3, [r4++]). The cache block of the post-incremented address is automatically loaded into cache.

Locality Hints affect Data distribution between temporal/non-temporal caches

Figure 4-5. Allocation Paths Supported in the Memory Hierarchy

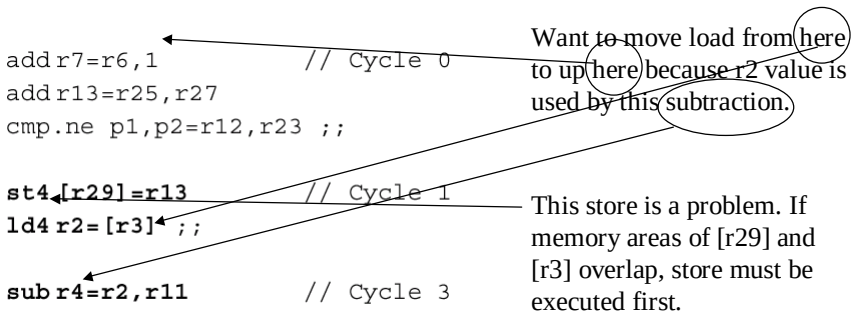


Locality hints allow compiler to designate which addresses contain streaming data. Streaming caches obviously will have a different block replacement strategy than temporal cache.

The Itanium does NOT implement a separate cache for streaming data, so locality hints are ignored.

Speculative Loads

- Data Speculative Loads – load is moved ahead of an *ambiguous Store*
  - An ambiguous Store is one in which compiler does not know the store address:



Allowing Data Speculative Loading

- An advanced load instruction (*ld.a*) is used for data speculative loading
  - The register number used in the load and load address is stored in an Advanced Load Address Table (ALAT).
- Before the result of the load can be used by a non-speculative instruction, the check instruction (chk.a) must be used to see if load value is valid
  - Check must use same register number, simply checks to see if the ALAT entry is still present. If NOT PRESENT, then jump to compiler-generated recovery code that re-executes the load and dependent instructions
- An ALAT table entry is removed if
  - there is a STORE that overlaps an ALAT entry
  - An ALAT entry has been replaced by another entry due to the same register or cache replacement choice
  - The OS or other hardware conditions invalidate the entry

Itanium ALAT

- Is a 32 entry, two-way set associative cache
- The "tag" is the register number
- The "data" is the address and the size of the load (byte, 2-byte, 4-byte, 8-byte).
- A miss causes a 10-cycle pipeline flush in addition to the recovery code
  - Recovery code on Itanium is an OS trap, requires several status register reads
  - Overhead is estimated at 50+ cycles plus recovery code execution
  - Miss penalty is very steep.

Predicated Execution (Guarded Execution)

- Branches are obviously difficult to deal with in superscalar architectures
- Would like to reduce number of branches. One way is via instructions that support *predicated execution* (guarded execution)

```
(p1)  add  r1=r2,r3
```

Predicate register . Each predicate register is 1-bit

This add is executed if P1 = '1', else it is a NOP.

Converting an If statement to Predicated Execution

```
if (r4) {                          cmp.ne p1,p0=r4,0
    add  r1=r2,r3        ⇒     (p1)add   r1=r2,r3
    ld8  r6=[r5]               (p1)ld8   r6=[r5]
}
```

The cmp.ne instruction sets the predicate registers p1, p0 based on the comparison of r4 to '0'.

P1 register will have '1' if comparison is true, P0 will be complement of P1.

Note that the compiler is free to place additional independent instructions between the 'cmp.ne' and subsequent 'add', 'ld8' instructions.

Pros/Cons of Predicated Execution

- Cons
  - Even if predicate is false, instructions are executed as NOPs
  - For If-else code, both the TRUE and FALSE instructions are executed

```
if (condition) {
  /* true code */
} else {
  /* false code */
}
```

- Pros
  - For short sequences of predicated instructions, can be faster than using branches since the instructions are in the pipeline anyway and it is more efficient to go ahead and execute them as NOPs since a branch mis-prediction can be expensive
- It is the compiler's decision whether to use branches or predicated instructions

Branch Prediction

- IA-64 has many ways to reduce branch mis-prediction
- Branch prediction hints can be given by compiler
  - within the branch itself
  - via a separate instruction (branch hint instruction)
- For branches, hints added by compiler are:
  - Prediction method: Static Not-Taken, Static Taken, Dynamic-Not-Taken (use dynamic predictor, 1st guess is not-taken), Dynamic-Taken
  - Sequential Prefetch hint of FEW or MANY. If FEW, then only prefetch a few instruction cache lines. If MANY, then prefetch more.
  - Predictor de-allocation hint of NONE or CLEAR. If NONE, then remember branch history and branch prediction of this branch. If CLEAR, then free all branch prediction resources for this branch after execution (guess that you will not execute this branch again).

Branch Prediction Instructions

- Explicit instruction used to provide early information about future branches
  - location of branch
  - target address of branch
  - branch importance (a hardware hint as to how much predication hardware should be used on branch). Can speed up tight loops.
  - Branch hints. Can hint that branch is static taken, should use dynamic hardware, or is a special branch type like a counting loop branch or exit branch.

Special Branches

- In addition to the normal conditional branch instructions, there are special branch instructions that are useful for common structures and which can eliminate misprediction
- *Counted loop* Branch (cloop) – uses a special register for counting executing of a loop. No mispredictions for this branch.
- While.Top, While.Exit loop branches for while loops with tests at top of loop and end of loop
  - Does not affect branch prediction
  - Works with hardware support for software pipelining of loops – may discuss this later in the course.