PTLsim User's Guide and Reference

The Anatomy of an x86-64 Out of Order Microprocessor

Matt T. Yourst
`<yourst@yourst.com>`

Revision 20051202

The latest version of PTLsim and this document are always available at:

www.ptlsim.org

Contents

Part I

PTLsim User's Guide

Chapter 1

Introducing PTLsim

1.1 Introducing PTLsim

PTLsim is a state of the art cycle accurate microprocessor simulator and virtual machine for the x86 and x86-64

1.3 Documentation Roadmap

This manual has been divided into several parts:

- Part I introduces PTLsim and describes its structure and operation
- Part 5 describes the PTLsim internal micro-operation instruction set and its relation to x86 and x86-64
- Part II details the design and implementation of the PTLsim out of order core model
- Part III

Chapter 2

Getting Started with PTLsim

2.1 Building PTLsim

PTLsim is written in C++ with extensive use of x86 and x86-64 inline assembly code for performance and virtualization purposes. In its present release, it is designed for use on an x86-64 host system running Linux 2.6.

Notes:

- **PTLsim is currently intended for x86-64 machines only.** Do not attempt to build it on a normal 32-bit x86 machine - it will not work. However, we will be modifying PTLsim in the near d.3wre to run on regular 32-bit x86 systems (albeit with lower performance and the lack of x86-64 support).
- PTLsim is very sensitive to the **Linux kernel** version it is running on. We have tested this version of PTLsim

PTLsim reads configuration options for running various user programs by looking for a configuration file named `/home/`*username*`/.ptlsim/`*path/to/program/executablename*`.conf`. To set options for each program, you'll need to create a directory of the form `/home/`*username*`/.ptlsim` and make sub-directories under it corresponding to the full path to the program. For example, to configure `/bin/ls` you'll need to run "`mkdir /home/`*username*`/.ptlsim/bin`" and then edit "`/home/`*username*`/.ptlsim/bin/ls.conf`" with the appropriate options. For example, try putting the following in `ls.conf` as described:

```
    -logfile ls.ptlsim -loglevel 9 -stats ls.stats -stopinsns 10000
```

Then run:

```
    ptlsim /bin/ls -la
```

PTLsim should display its system information ban8er, then the output of simulating the directory listing. With the

- Template based metaprogramming functions including `lengthof` (finds the lengthincludof any static array) and `log2` (takes the base-2 log of any constant at compile time)
- Floor, ceiling and masking functions for integers and powers of two (`floor`, `trunc`, `ceil`, `mask`, `floorptr`, `ceilptr`, `maskptr`, `signext`, etc)
- Bit manipulation macros (`bit`, `bitmask`, `bits`, `lowbits`, `setbit`, `clearbit`, `assignbit`

- Index reference (`indexref`) is a smart pointer which compresses a full pointer into an index into a specific

- `FullyAssociativeTags8bit` and `FullyAssociativeTags16bit` work just like `FullyAssociativeTags`, except that these classes are dramatically faster when using small 8-bit and 16-bit tags. This is possible through the clever use of x86 SSE vector instructions to associatively match and process 16 8-bit tags or 8 16-bit tags every cycle. In addition, these classes support features like removing an entry from the mid-

At this point, the PTLsim image injected into the user process exists in a bizarre environment: if the user program

Chapter 4

Statistics Collection and Control

4.1 Using PTLstats to Analyze Statistics

PTLsim maintains a huge number of statistical counters and data points during the simulation process, and can optionally save this data to a statistics data store by using the "`-stats` *filename*" configuration option introduced in Section 2.3. The data store is a binary file format (defined in

```
[33% ] wait-storedata-sfraddr = 9755097;[33% ] wait-storedata-sfraddr-sfrdata = 9
```

values to a text file). It is further suggested that only raw values be saved, rather than doing computations in the

4.5 Simulation Warmup Periods

In some simulators, it is possible to quickly skip through a specific number of instructions before starting to gather

Chapter 5

x86 Instructions and Micro-Ops (uops)

5.1 Micro-Ops (uops) and TransOps

PTLsim presents to user code a full implementation of the x86 and x86-64 instruction set (both 32-bit and 64-bit

5.7 Unaligned Loads and Stores

Compared to RISC architectures, the x86 architecture is infamous for its relatively widespread use of unaligned memory operations; any implementation must efficiently handle this scenario. Fortunately, analysis shows that unaligned accesses are rarely in the performance intensive parts of a modern program, so we can aggressively eliminate them on contact through rescheduling. PTLsim does this by initially causing all unaligned loads and stores to raise an `UnalignedAccess`

5.11 x87 Floating Point

The legacy x87 floating point architecture is the bane of all x86 processor vendors' existence, largely because its stack based nature makes out of order processing so difficult. While there are certainly ways of translating stack based instruction sets into flat addressing for scheduling purposes, we do not do this. Fortunately, following the Pentium III and AMD Athlon's introduction, x87 is rapidly headed for planned obsolescence; most major applications released within the last three years now use SSE instructions for their floating point needs either exclusively or in all performance critical parts. To this end, even Intel has relegated x86 support on the Pentium 4 to a separate low

Part II

Out of Order Processor Model

Chapter 6

Chapter 7

Fetch Stage

7.1 Instruction Fetching and the Basic Block Cache

As described in Section 5.1, x86 instructions are decoded into transops prior to actual execution by the out of order core. Some processors do this translation as x86 instructions are fetched from an L1 instruction cache, while others

```
branchpred.predict()
```

Table 8.1: Architectural registers and pseudo-registers used for renaming.

Chapter 9

Scheduling, Dispatch and Issue

9.1 Clustering and Issue Queue Configuration

The PTLsim out of order model can simulate an arbitrarily complex set of functional units grouped into *clusters*. Clusters are specified by the `Cluster` class and are defined by the `clusters[]` array in `ooohwdef.h`. Each

`fu_mask` field)]-234(and)-234(the)

B

Table 9.1: Issue Queue State Machine

Valid	Issued	Meaning
0	0	Unused slot
0	1	(invalid)
1	0	Dispatched but waiting for operands
1	1	

The `issue()` method simply finds the index of the first set bit in the `allready` bitmap (this is the slot of the oldest ready uop in program order), marks the corresponding slot as issued, and returns the slot. The processor then selects a functional unit for the uop in that slot and executes it via the `ReorderBufferEntry::issue()` method. After `iqslot, issue())` macro (Section 9.1) is used to invoke the

The first uop to annul is determined in the `annul()` method by scanning backwards in time from the excepting uop until a uop with its SOM (start of macro-op) bit is set, as described in Section 5.1. This SOM uop represents the

Chapter 11

Load Issue

11.1 Issuing Loads

The `ReorderBufferEntry::issueload()` function is responsible for issuing all load uops. The `issueload()`

Chapter 12

Stores

12.1 Store to Store Forwarding and Merging

writes the address into the corresponding `LoadStoreQueueEntry` structure before setting its the `addrvalid` bit
as described in Section 8.4. If an exception is detected at this pod9091TfS(an)4]TJ/F469.9626Tf134.5098.77Td[(bit)]inbit

Some uops may also commit to a subset of the x86 flags, as specified in the uop encoding. For these uops, in theory no rename tables need updating, since the flags can be directly masked into the REG_flags architectural pseudo-register. Should the pipeline be flushed, the rename table entries for the ZAPS, CF, OF flag sets will all be reset to point to the REG_flags pseudo-register anyway. However, for the speculation recovery scheme described in Section 10.2, the REG_zf, REG_cf, and REG_of commit RRT entries are updated as well to match the updates done to the speculative RRT.

Branches and jumps update the REG_rip pseudo architectural register, while all 7ther uops simply increment

One solution (the one used by PTLsim) is to give each physical register a reference counter. Physical registers can be referenced from three structures: as operands to ROBs, from the speculative RRT, and from the committed RRT. As each uop operand is renamed, the counter for the corresponding physical register is incremented by calling the `PhysicalRegister::addref()` method. As each uop commits, the counter for each of its operands is decremented via the `PhysicalRegister::unref()` method. Similarly, `unref()` and `addref()` are used whenever an entry in the speculative RRT or commit RRT is updated. During mis-speculation recovery (see Section 10.2), `unref()` is also used to unlock the operands of uops slated for annulment. Fi7ally, `unref()` and `addref()` are a fixed physical register for each of the 64 architectural registers in `ctx.commitarf[]`, setting the speculative and committed rename tables to their proper cold start values, and resetting all reference counts on physical registers as appropriate. If the processor is configured with multiple physical register files (Section 8.3), the initial physical register for each architectural register is allocated in the first physical register file only (this is configurable by modifying

Chapter 15

Cache Hierarchy

The PTLsim cache hierarchy model is highly flexible and can be used to model a wide variety of contemporary cache

In dcacheint.h, the two base classes `CacheLine` and `CacheLineWithValidMask` are interchangeable, depending on the model being used. The `CacheLine` class is a standard cache line with no actual data (since the bytes in each the L1 cache. Similarly, an L2 miss but L3 hit results in the `STATE_DELIVER_TO_L2` state, and a miss all simply checks one of the simulator's Shadow Page Access Tables (SPATs) as described in Section 3.5. For DTLB accesses, the `dtlbmap` SPAT is used, while ITLB accesses use the `itlbmap` SPAT. If a bit in the appropriate SPAT

To solve this problem, the RAS is only updated in the allocate stage immediately after fetch. In the out of order core's `rename()` function, the `BranchPredictorInterface::updateras()` method is called to either push

Part III

Appendices

Chapter 17

PTLsim uop Reference

`add sub addadd addsub subadd subsub addm subm addc subc`

Add and Subtract

Mnemonic	Syntax	Operation
`add`	`rd = ra,rb`	rd = ra ← ra + rb
`sub`	`rd = ra,rb`	rd = ra ← ra - rb
`adda`	`rd = ra,rb,rc*S`	rd = ra ← ra + rb + (rc < < S)
`adds`	`rd = ra,rb,rc*S`	rd = ra ← ra + rb + (rc < < S)
`suba`	`rd = ra,rb,rc*S`	rd = ra ← ra + rb + (rc < < S)
`subs`	`rd = ra,rb,rc*S`	rd = ra ← ra + rb + (rc < < S)
`addm`	`rd = ra,rb,rc`	rd = ra ← (ra + rb) & ((1 < < rc) - 1)
`subm`	`rd = ra,rb,rc`	rd = ra ← (ra - rb) & ((1 < < rc) - 1)
`addc`	`rd = ra,rb,rc`	rd = ra ← (ra + rb) + rc.cf
`subc`	`rd = ra,rb,rc`	rd = ra ← (ra - rb) - rc.cf

Notes:

`sel`

Conditional Select

`set`

Conditional Set

`set.sub set.and`

Conditional Compare and Set

Mnemonic	Syntax	Operation
`set.sub.`*`cc`*	`rd = ra,rb,rc`	rd = rc ← EvalFlags(ra - rb) ? 1 : 0
`set.and.`*`cc`*	`rd = ra,rb,rc`	rd = rc ← EvalFlags(ra & rb) ? 1 : 0

Notes:

- The `set.sub` and `set.and` uops take the place of a `sub` or `and` uop immediately consumed by a `set` uop; this is intended to shorten the c(0)1659a `br`

Conditional Branch

Mnemonic	Syntax	Operation
`br.`*`cc`*	`rip = (ra),riptaken,ripseq`	rip = EvalFlags(ra) ? riptaken : ripseq

Notes:

- ***cc*** is any valid condition code flag

```
br.sub br.and
```

`jmp`

Indirect Jump

Mnemonic	Syntax	Operation
`jmp`	`rip = ra,riptaken`	rip = ra

Notes:

- The `rip` (user-visible instruction pointer register) is reset to the target address specified by *ra*
- If the *ra* operand does not match the *riptaken*

`bru`

Unconditional Branch

Mnemonic	**Syntax**	**Operation**
`bru`	`rip = riptaken`	**Notes** rip = riptaken

- The `rip` (user-visible instruction pointer register) is reset to the specified immediate. The processor may redirect fetching from the new RIP
- No exceptions are possible with unconditional branches
- If the target RIP falls withi8 a8 unmapped page, not present page or a marked as no-execute (NX), the `PageFaultOnExec` exception is taken.
- No flags are generated by this uop

chk
Check Speculation

Mnemonic	Syntax	Operation
chk		

`ld ld.lo ld.hi ldx ldx.lo ldx.hi`

Load

Mnemonic	**Syntax**	**Operation**
`ld`	`rd = [ra,rb],sfra`	rd = MergeWithSFR(mem[ra + rb], sfr8(+)-)
`ld.lo`	`rd = [ra+rb],sfra`	rd = MergeWithSFR(mem[floor(ra + rb), 8], sfr8(+)-)
`ld.hi`	`rd = [ra+rb],rc,sfra`	rd = MergeAlign(MergeWithSFR(mem[(floor(ra + rb), 8) + 8], sfr8(+)-), rc)

Notes:

- *The PTLsim load unit model is described in substantial detail in Section 11.1; this section only gives an overview of the l [illegible]*

- `PageFaultOnRead` if the virtual address (*ra* + *rb*) falls on a page not accessible to the caller in the current operating mode, or a page marked as not present.
-

`st`

Store

Mnemonic	Syntax	Operation
`st`	`sfrd = [ra,rb],rc,sfra`	sfrd = MergeWithSFR((ra + rb), sfra, rc)
`st.lo`	`sfrd = [ra+rb],rc,sfra`	sfrd = MergeWithSFR(floor(ra + rb, 8), sfra, rc)
`st.hi`	`sfrd = [ra+rb],rc,sfra`	sfrd = MergeWithSFR(floor(ra + rb, 8) + 8, sfra, rc)

Notes:

- `UnalignedAccess` if the address (*ra* + *rb*) is not aligned to an integral multiple of the size in bytes of the store. Unaligned stores (`st.lo` and `st.hi`) do not generate this exception. Since x86 automatically corrects alignment problems, microcode must handle this exception as described in Section 5.7.
- `PageFaultOnWrite` if the virtual address (*ra* + *rb*

`stp`

Store to Internal Space

Mnemonic	**Syntax**	**Operation**
`stp`	`null = [ra,rb],rc`	MSR[ra+rb] = rc

Notes:

- The `stp`
  Typically this address space is mapped to internal machine state registers (MSRs) and microcode

`shl shr sar rotl rotr rotcl rotcr`

Shifts and Rotates

Mnemonic	Syntax	Operation
`shl`	`rd = ra,rb,rc`	rd = ra ← (ra << rb)
`shr`	`rd = ra,rb,rc`	rd = ra ← (ra >> rb)
`sar`	`rd = ra,rb,rc`	rd = ra ← SignExt(ra >> rb)
`rotl`	`rd = ra,rb,rc`	rd = ra ← (ra *rotateleft* rb)
`rotr`	`rd = ra,rb,rc`	rd = ra ← (ra *rotateright* rb)
`rotcl`	`rd = ra,rb,rc`	rd = ra ← ({rc.cf, ra} *rotateleft* rb)
`rotcr`	`rd = ra,rb,rc`	rd = ra ← ({rc.cf, ra} *rotateright* rb)

Notes:

- The shift and rotate instructions have some of the most bizarre semantics in the entire x86 instruction set: they may or may not modify flags depending on the rotation count operand, which we may not even know until the instruction issues. This is introduced in Section 5.9.
- The specific rules are as follows:
  - If the count `rb = 0` is zero, no flags are modified
  - If the count `rb = 1`, both OF and CF are modified, but ZAPS is preserved
  - If the count `rb > 1`, only the CF is modified. (Technically the value in OF is undefined, but on K8 and P4, it retains the old value, so we try to be compatible).
  - Shifts also alter the ZAPS flags while rotates do not.
- For constant counts (immediate *rb8.0cmyk.4dv[)25values)27the)28semantics)25are)25easy)26to)25deter)25mine)28in)28adv)25ance*

`bswap`
Byte Swap

`bswap`	`rd = ra`	rd = ra $\leftarrow$ ByteSwap(ra)

- The `bswap` uop reverses the endianness of the *ra* operand. The uop's effective result size determines the range of bytes which are reversed.
- This uop's semantics are identical to the x86 `bswap` instruction.
- This uop does not generate any condition code flags.

`collcc`

Collect Condition Codes

Mnemonic	Syntax	Operation

`collcc`	`rd = ra,rb,rc`	rd.zaps = ra.zaps
		rd.cf = rb.cf
		rd.of = rc.of
		rd = rd.flags

Notes:

- The `collcc`

`movccr movrcc`

Move Condition Code Flags Between Register Value and Flag Parts

Mnemonic	Syntax	Operation

`movccr`	`rd = ra`	rd = ra.flags
		rd.flags = 0

`movrcc`	`rd = ra`	rd.flags = ra
		rd = ra

Notes:

- The `movccr` uop takes the condition code flag bits attache to *ra* and copies them into the 64-bit register part of the result.
- The `movrcc` uop takes the low bits of the *ra* operand and moves those bits into the conditionaflag bits attache

`andcc orcc ornotcc xorcc`
Logical Operations on Condition Codes

Mnemonic	Syntax	Operation
`andcc`	`rd = ra,rb`	rd.flags = ra.flags & rb.flags
`orcc`	`rd = ra,rb`	rd.flags = ra.flags | rb.flags
`ornotcc`	`rd = ra,rb`	rd.flags = ra.flags | (~rb.flags)
`xorcc`	`rd = ra,rb`	rd.flags = ra.flags ^ rb.flags

Notes:

- These uops are used to perform logical operations on the condition code flags attached to *ra* and *rb*.
- If the *rb* operand is an immediate, the immediate data is used instead of the flags normally attached to a register operand.
- The 64-bit value of the output is always set to zere.

`mull mulh`

Integer Multiplication

Mnemonic	**Syntax**	**Operation**
`mull`	`rd = ra,rb`	rd = ra ← lowbits(ra × rb)
`mulh`	`rd = ra,rb`	rd = ra ← highbits(ra × rb)

Notes:

and , the bits are high only the low result (where N is
s then merged into

- imul); the flagn are calculated relative to the effective result size.

- operand may be an immediate

`ctz clz`
Count Trailing or Leading Zeros

`ctpop`

Count Population of ’1’ Bits

Mnemonic	Syntax	Operation

`ctpop`	`rd = ra`	rd.zf = (ra == 0)
		rd = PopulationCount(ra)

Notes:

- The `ctpop` uop counts the number of ’1’ bits in the *ra* operand.
-

Floating Point Format and Merging

```
addf subf mulf divf minf maxf
```

[illegible]4699.06218cmq[]0dm0J0.3985wm00.1992m504.1

`sqrtf rcpf rsqrtf`

Square Root, Reciprocal and Reciprocal Square Root

Mnemonic	Syntax	Operation
`sqrtf`	`rd = ra,rb`	rd = ra ← sqrt(rb)
`rcpf`	`rd = ra,rb`	rd = ra ← 1 / rb
`rsqrtf`	`rd = ra,rb`	rd = ra ← 1 / sqrt(rb)

Notes:

- These uops perform the specified unary operation on rb and merge the result into ra (for a single precision scalar mode only)
- The `rcpf` and `rsqrtf` uops are approximates - they do not provide the full precision results. These approximations are in accordance with the standard x86 SSE/SSE2 semantics.

`cmpccf`

Compare Floating Point and Generate Condition Codes

`cvtf.i2s.ins cvtf.i2s.p cvtf.i2d.lo cvtf.i2d.hi`

Convert 32-bit Integer to Floating Point

`cvtf.q2s.ins cvtf.q2d`
Convert 64-bit Integer to Floating Point

Mnemonic	Syntax	Operation	Used By
`cvtf.q2s.ins`			

`cvtf.d2i cvtf.d2q cvtf.d2i.p`

Convert Double Precision Floating Point to Integer

Mnemonic	Syntax	Operation	Used By
`cvtf.d2i`	`rd = ra`	rd = DoubleToInt32(ra)	`CVTSD2SI`
`cvtf.d2i.p`	`rd = ra,rb`	rd[63:32] = DoubleToInt32(ra)	
		rd[31:0] = DoubleToInt32(rb[illegible]	

`cvtf.d2s.ins cvtf.d2s.p cvtf.s2d.lo cvtf.s2d.hi`

Convero Between Double Precision and Single Precision Floating Point

Mnemonic	Syntax	Operation	Used By
`cvtf.d2s.ins`	`rd = ra,rb`	rd	

Chapter 18

Performance Counters

PTLsim maintains hundreds of performance and statistical counters and data points as it simulates user code. In Section 4

18.2 Out of Order Core

summary: summarizes the performance of user code running on the simulator

- **cycles:**

– **br:**

- **width:** histogram of the issue width actually used on each cycle in each cluster. This object is further broken down by cluster, since various clusters have different issue width and policies.
- **opclass:** histogram of how many uops of various operation classes were issued. The operation classes are defined in `ptlhwdef.h` and assigned to various opcodes in `ptlhwdef.cpp`.

branchpred:

sfr-addr-not-ready:

-