

# PTLsim User's Guide and Reference

*The Anatomy of an x86-64 Out of Order Microprocessor*

Matt T. Yourst  
<yourst@yourst.com>

Revision 20051010

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

**[www.ptlsim.org](http://www.ptlsim.org)**





<b>9</b>	<b>Scheduling, Dispatch and Issue</b>	<b>38</b>
9.1	Clustering and Issue Queue Configuration . . . . .	38
9.2	Cluster Selection . . . . .	38
9.3	Issue Queue Structure and Operation . . . . .	39
9.3.1	Implementation . . . . .	40
9.3.2	Other Designs . . . . .	40
9.4	Issue . . . . .	40
<b>10</b>	<b>Speculation and Recovery</b>	<b>42</b>
10.1	Misspeculation Cases . . . . .	

## 15 Cache Hierarchy

## **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



## 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/.ptl sim/path/to/program/executablename.conf`. To set options for each program, you'll need to create a directory of the form `/home/username/.ptl sim` 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/.ptl sim/bin"` and then edit `"/home/username/.ptl sim/bin/ls.conf"` with the appropriate options. For example, try putting the following in `ls.conf` as described:

```
-logfile ls.ptl sim -loglevel 9 -stats ls.stats -stopinsns 10000
```

Then run:

```
ptl sim /bin/ls -la
```

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

**-trigger**            Trigger mode: wait for user process to use special function `ptl call _switch_to_sim()` before entering simulation mode [disabled]. This is described in [Section 4.3](#).

***Simulation Stop Point:***

**-stop *N***           Stop after *N*





- Template based metaprogramming functions including `lengthof` (finds the `length` of 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% ] wai t-storedata-sfraddr = 9755097; [33% ] wai t-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



## 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











## **Part III**

# **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()

## Chapter 8

# Frontend and Key Structures

### 8.1 Resource Allocation



## 8.4 Load Store Queue Entries

Load Store Queue (LSQ) Entries (the `LoadStoreQueueEntry` structure in PTLsim) are used to track additional



# 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

a functional unit for the uop in that slot and executes it via 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

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

## 11.1 Issuing Loads

### Load Issue

and the memory range needed by the load overlaps the memory range touched by the store, the load effectively has a dependency on the earlier store that must be resolved before  $\text{ear}_{39}(\text{load})-3$  issue. The meaning of “overlapping memory range” is defined more specifically in Section [12.1](#).

## 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 point, the processor writes the address into the `LoadStoreQueueEntry` structure before setting its the `addrvalid` bit as described in Section 8.4. If an exception is detected at this point, the processor writes the address into the `LoadStoreQueueEntry` structure before setting its the `addrvalid` bit as described in Section 8.4.





the `wri teback()` function; its sole purpose is to place the uop's physical register into the written state (via the `Physi cal Regi ster::wri teback()` method) and to move the ROB into its terminal state, *ready-to-commit*.

## Chapter 14

# Commitment

### 14.1 Introduction

The commit stage examines uops from the head of the ROB, blocks until all uops comprising a given x86 instruction

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 other uops simply increment



## 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 IV

## 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 + rb
sub	rd = ra, rb	rd = ra - rb
adda	rd = ra, rb, rc*S	rd = ra + rb + (rc << S)
adds	rd = ra, rb, rc*S	rd = ra + rb + (rc << S)
suba	rd = ra, rb, rc*S	rd = ra + rb + (rc << S)
subs	rd = ra, rb, rc*S	rd = ra + rb + (rc << S)
addm	rd = ra, rb, rc	rd = ra + rb & ((1 << rc) - 1)
subm	rd = ra, rb, rc	rd = ra - rb & ((1 << rc) - 1)
addc	rd = ra, rb, rc	rd = ra + rb + rc.cf
subc	rd = ra, rb, rc	rd = ra - rb - rc.cf

Notes:



sel

## Conditional Select

Mnemonic	Syntax	Operation
sel . cc	rd = (ra), rb, rc	rd = (EvalFlags(ra)) ? rc : rb

**Notes:**

- **cc** is any 09.8729id condition code flag e09.8729uation
- The sel sub(is) 27 gets the isel bited 8 (e) Bappt(e)-27 that the

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. <i>cc</i>	ri p = (ra), ri ptaken, ri pseq	rip = EvalFlags(ra) ? riptaken : ripseq

### Notes:

- *cc* is any valid condition code defining

br. sub br. and

## Compare and Conditional Branch

---

Mnemonic	Syntax	Operation
br. cc	ri p = ra, rb, ri ptaken, ri pseq	rip = EvalFlags(ra - rb) ? riptaken : ripseq
br. cc	ri p = ra, rb, ri ptaken, ri pseq	rip = EvalFlags(ra & rb) ? riptaken : ripseq

### Notes:

- The br. sub and br. and uops take the place of a

## j mp Indirect Jump

---

Mnemonic	Syntax	Operation
j mp	ri p = ra, ri ptaken	rip = ra

### Notes:

- The ri p (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: 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 within an 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 load semantics.

- 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

- `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



i nshb exthb movhb

## Byte Operations

---



## shl shr sar rotl rotr rotcl rotrc

### 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 <i>rotateleft</i> rb)
rotr	rd = ra, rb, rc	rd = ra (ra <i>rotateright</i> rb)
rotcl	rd = ra, rb, rc	rd = ra ({rc.cf, ra} <i>rotateleft</i> rb)
rotrc	rd = ra, rb, rc	rd = ra ({rc.cf, ra} <i>rotateright</i> 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  $rb$ )

## dupbi t

### Duplicate Bit

---

Mnemonic	Syntax	Operation
dupbi t	rd = ra, rb, rc	rd = ra    (rb[rc], rb[rc], rb[rc], ...)

Notes:

-

extr extrx  
Extract Bit Field

i nsr

Inser Bit Field/F72ET0150407.002.0892BT/q[]0d0J0.3985w00.1992Bmm0g.1992BISQ1-



## bswap Byte Swap

bswap      rd = ra    rd = ra    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.



col l cc

## Collect Condition Codes

---

Mnemonic	Syntax	Operation
col l cc	rd = ra, rb, rc	rd.zaps = ra.zaps rd.cf = rb.cf rd.of = rc.of rd = rd.flags

### Notes:

- The col l cc

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 attached 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 attached

andcc orcc ornotcc xorcc

## Logical Operations on Condition Codes

---

mul l   mul h

# Integer Multiplication

---

Mnemonic	Syntax	Operation
mul l	rd = ra, rb	rd = ra   lowbits(ra × rb)
mul h	rd = ra, rb	rd = ra   highbits(ra × rb)

Notes:

and , the bits starting from the low (where N is  
s then merged into

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

operand may be an immediate



ctz clz

Count Trailing or Leading Zeros

---

Mnemonic	Syntax	Operation
----------	--------	-----------

---

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 mul f di vf mi nf maxf

## Floating Point Arithmetic

---

Mnemonic	Syntax	Operation
addf	rd = ra, rb	rd = ra + rb
subf	rd = ra, rb	rd = ra - rb
mul f	rd = ra, rb	rd = ra × rb
di vf	rd = ra, rb	rd = ra / rb
mi nf	rd = ra, rb	rd = ra (ra < rb) ? ra : rb
maxf	rd = ra, rb	rd = ra (ra >= rb) ? ra : rb

Notes:

maddf msubf

## Fused Multiply Add and Subtract

---

Mnemonic	Syntax	Operation
maddf	rd = ra, rb, rc	rd = ra (ra × rb) + rc
msubf	rd = ra, rb, rc	rd = ra (ra × rb) - rc

### Notes:

- The `maddf` and `msubf`

`sqrtf rcpf rsqrtf`

## Square Root, Reciprocal and Reciprocal Square Root

---

Mnemonic	Syntax	Operation
<code>sqrtf</code>	<code>rd = ra, rb</code>	<code>rd = ra    sqrt(rb)</code>
<code>rcpf</code>	<code>rd = ra, rb</code>	<code>rd = ra    1 / rb</code>
<code>rsqrtf</code>	<code>rd = ra, rb</code>	<code>rd = ra    1 / sqrt(rb)</code>

### 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. i ns    cvtf. q2d

## Convert 64-bit Integer to Floating Point

---

Mnemonic	Syntax	Operation	Used By
cvtf. q2s. i ns			

cvtf.s2i cvt.s2q cvtf.s2i .p

Convert Single Precision Floating Point to Integer

---



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(rbDoubleToInt32(DoubleToFP32(a)))	TJ/F469.9626Tf263.9

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

## Converto 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 `ptl hwdef. h` and assigned to various opcodes in `ptl hwdef. cpp`.

**branchpred:**



sfr-addr-not-ready:





