PTLsim User's Guide and Reference

The Anatomy of an x86-64 Out of Order Microprocessor

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 operationž

Chapter 2

Getting Started with PTLsim

2.1 Building PTLsim

PTLsim reads configuration options for running various user programs by looking for a configuration file named `/home/`

-trigger

Chapter 3

PTLsim Internals

3.1 Overview

The following is an overview of the source files for PTLsim:

- `ooocore.cpp` is the out of order simulator itself. The microarchitectural model implemented by this simula-

- 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`). Note that the `bitvec` template (see below) should be used in place of these macros wherever it is more convenient.
- Comparison functions (`aligned`,`strequal`,`inrange`,`clipto`)
- Modulo arithmetic (`add_index_modulo`,`modulo_span`, et al)
- Definitions of basic x86 SSE vector functions (e.g. `x86_cpu_pcmpeqb` et al)
- Definitions of basic x86 assembly language functions (e.g. `x86_bsf64` et al)
- A full suite of bit scanning functions (`lsbindex`,`msbindex`,`popcount` et al)
- Miscellaneous functions (`arraycopy`,`setzero`, etc)

- Index reference (`indexref`) is a smart pointer which compresses a full pointer into an index into a specific structure (made unique by the template parameters). This class behaves exactly like a pointer when referenced,

At this point, the PTLsim image injected into the user process exists in a bizarre environment: if the user program is 32 bit, the boot code will need to switch to 64-bit mode before calling the 64-bit PTLsim entrypoint. Fortunately

Chapter

Statisticsol

Chapter 5

x86 Instructions and Micro-Ops (uops)

5.1 Micro-Ops (uops) and TransOps

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 use a trace cache to store pre-decoded uops. PTLsim takes a middle ground to allow maximum simulation flexibility.

`branchpred.predict()` function is used to redirect fetching. If the branch is predicted not taken, the sense of the branch's condition code is inverted and the transop's `riptaken` and `ripseq` fields are swapped; this ensures all branches are considered correct only if taken. Indirect branches (jumps) have their(80(,6.74769626Tf246.33430Td[(riptaken)]

Chapter 8

Frontend and Key Structures

8.1 Resource Allocation

During the Allocate stage, PTLsim dequeues uops from the fetch queue, ensures all resources needed by those uops

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)

Table 9.1: Issue Queue State Machine

Valid	Issued

a functional unit for the uop in that slot and executes it via the `ReorderBufferEntry::issue()` method. After the uop has completed execution (i.e. it cannot possibly be replayed), the `release()` method is called to remove the slot from the issue queue, freeing it up for incoming uops in the dispatch stage. The collapsing design of the issue queue means that the slot is not simply marked as invalid - all slots after it are physically shifted left by one, leaving a free slot at the end of the array. This design is relatively simple to implement in hardware and makes determining the oldest ready to issue uop very trivial.

Because of the collapsing mechanism, it is critical to note that the slot index returned by `issue()` will become invalid after the next call to the `remove()` method; hence, it should never be stored anywhere if a slot could be removed from the issue queue in the meantime.

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

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 ear39(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 `invalid` bit in the store queue entry is set and the destination physical register's `FLAG_inv` flag is set so any attempt to commit the store will fail.

12.2.1 Load Queue Search (Alias Check)

The load queue is then searched to find any loads after the current store in program order which have already issued but have done so without forwarding data from the current store. These loads erroneously issued before the current store (now known to overlap the load's address) was able to forward the correct data to the offending load(s). This situation is known as *aliasing*

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

Chapter 14

Commitment

14.1 Introduction

Some uops may also commit to a subset of the x86 flags, as specified in the uop encoding. For these uops, in

Chapter 15

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 the way to main memory results in `STATE_DELIVER_TO_L3`.

In the very unlikely event that either the LFRQ slot or miss buffer are ful6, an exception is returned to out of order core, which typically replays the affected load until space in these structures becomes available. For prefetch requests, only a miss buffer is allocated; no LFRQ slot is needed.

15.3 Filling a Cache Miss

The `MissBuffer::clock()`

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

all the information needed to eventually update the branch predictor at the end of the pipeline. The contents will

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()`

Part IV

Appendices

Chapter 17

PTLsim uop Reference

The following sections document the semantics and encoding of each micro-operation (uop) supported by the PTLsim processor core. The `opinfo[]` table in `ptlhwdef.cpp` and constants in `ptlhwdef.h` give actual numerical values for the opcodes and other fields described below.

`mov and or xor andnot ornot nand nor eqv`

Logical Operations

Mnemonic	Syntax	Operation
`mov`	`rd = ra,rb`	rd = ra ← rb
`and`	`rd = ra,rb`	rd = ra ← ra & rb
`or`	`rd = ra,rb`	rd = ra ← ra | rb
`xor`	`rd = ra,rb`	rd = ra ← ra ^ rb
`andnot`	`rd = ra,rb`	rd = ra ← (~ra) & rb
`ornot`	`rd = ra,rb`	rd = ra ← (~ra) | rb
`nand`	`rd = ra,rb`	rd = ra ← ~(ra & rb)
`nor`	`rd = ra,rb`	rd = ra ← ~(ra | rb)
`eqv`	`rd = ra,rb`	rd = ra ← ~(ra ^ rb)

Notes:

•

`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*`	

`sel`

Conditional Select

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

`set`

Conditional Set

[illegible]

`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:

-

br

`jmp`

Indirect Jump

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

rip **Notes:**

ripa rip [illegible] *riptaken* [illegible] register) is reset to the target address specified by *ripa*

`brp`

Unconditional Branch Within Microcode

Mnemonic	Syntax	Operation
`bru`	`null = riptaken`	

`chk`
Check Speculation

Mnemonic	Syntax	Operation
`chk.`*cc*	`rd = ra,recrip,extype`	rd = EvalCheck(ra) ? 0 : recrip

Notes:

- The `chk` uop verifies *certain* properties about ra. If this verification check passes, no action is taken. If the check fails, `chk` signals an exception of the user specified type in the *rc* immediate. The result of the `chk` uop in this case is the user specified RIP to recover at after the check failure is handled in microcode. This recovery RIP is saved in the `recoveryrip` internal register.
- This mechanism is intended to allow simple inlined uop sequences to branch into microcode if certain conditions fail, since normally inlined uop sequences cannot contain embedded branches. One example use is in the `REP` series of instructions to ensure that the count is not zero on entry (a special corner case).
- Unlike most conditional uops, the `chk`

- `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.
- Various other exceptions and replay conditions may exist depending on the specific processor core model.

`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:

- *The PTLsim store unit model is described in substantial detail in Section 12.1; this section only gives an overview of the store uop semantics.*
- The `st` family of uops prepares values to be stored to the virtual address specified by the sum *ra* + *rb*.
- The *sfra* operand specifies the store forwarding register (a.k.a. store buffer) to merge the data to be stored (the

- `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 541068ihTd[leects91(thts)-291(e)30(xcep068ias068idescr)-ems

`ldp ldxp`
Load from Internal Space

Mnemonic	Syntax	Operation
`ldp`	`rd = [ra,rb]`	rd = MSR[ra+rb]
`ldxp`	`rd = [ra+rb]`	rd = SignExt(MSR[ra+rb])

Notes:

The `ldp` and `ldxp` uops load values from the internal PTLsim address space not accessible to x86 code. Typically this address space is mapped to internal machine state registers (MSRs) and microcode scratch space. The internal address to access is specified by the sum

`stp`
Store to Internal Space

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

Notes:

-

`inshb exthb movhb`

Byte Operations

Mnemonic	Syntax	Operation
`inshb`	`rd = ra,rb`	rd = ra[15:8] ← rb[7:0]
`exthb`	`rd = ra,rb`	rd = ra[7:0]

`movhl movl`

Merge 32-bit Words

`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 condition code flag bits attached to the result.
- The bits moved consist of the ZF, PF, SF, CF, OF flags
- The WAIT and INV flags of the result are always cleared since the uop would not even issue if these were set in *ra*.

`andcc orcc ornotcc xorcc`

Logical Operations on Condition Codes

`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:

-

`bt bts btr btc`

Bit Testing and Manipulation

Mnemonic	Syntax	Operation	
`bt`	`rd = ra,rb`	rd.cf = ra[rb]	
		rd = ra ← (rd.cf) ? -1 : +1	rd.cf = ra[rb]
			rd = ra ← (rd.cf)

`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.
- The ZF flag of the result is 1 if *ra* was zero,r7cp7N89704on151.74970T. O0.9a

Floating Point Format and Merging

All floating point uops use the same encoding to specify the precision and vector format of the operands. The uop's *size* field is encoded as follows:

- `00`:

Mnemonic **Syntax** **Operation** Floating Point Arithmetic

ra,rbrd = ra ← ra divfrd = ra,rbrd = ra ← ra / r

maddf msubf

madTd593.5979Tf5F459-6Tf6m54-6T7-5.4rB593.5979T/+61ž5040.1992l2775.80a.1992n.madTd593.The.9626

Fused Multiply Add and Subtract

Mnemonic	Syntax	Operation
maddf		

`cmpf`

Compare Floating Point

Mnemonic	Syntax	Operation
`cmpf.`*`type`*	`rd = ra,rb`	rd = ra $\leftarrow$ CompareFP(ra, rb, type) ? -1 : 0

Notes:

- This uop performs the specified comparison of *ra* and *rb*. If the comparison is true, the result is set to all ’1’ bits; otherwise it is zero. The result is then merged into ra.
- The *cond* field in the uop encoding holds the comparison type. The set of compare types matches the x86 SSE/SSE2 CMPxx instructions.

`cvtf.i2s.ins cvtf.i2s.p cvtf.i2d.lo cvtf.i2d.hi`
Convert 32-bit Integer to Floating Point

Mnemonic	Syntax	Operation	Used By
`cvtf.i2s.ins`	`rd = ra,rb`	rd = ra ← Int32ToFloat(rb)	`CVTSI2SS`

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

Mnemonic	Syntax	Operation	Used By
`cvtf.q2s.ins`	`rd = ra,rb`	rd = ra ← Int64ToFloatl0irbl01	`CVTSI2SS` (x86-64l01
`cvtf.q2d`	`rd = ra`	rd = Int64ToDoublel0iral01	`CVTPI2PS` (x86-64l01

Notes:

- These uops convert 64-bit integers to single or double precision floating point
- The semantics of these instructions are identical to the semantics of the x86 SSE/SSE2 instructions shown in the table
- The uop *size* field is not used by these uops

`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)	`CVTPD2PI`
		rd[31:0] = DoubleToInt32(rb)	`CVTPD2DQ`
`cvtf.e2q`	`rd = ra`	rd = DoubleToInt64(ra)	`CVTSD2SI` (x86-64)

Notes:

- These uops convert drd = precision floating point values to 32-bit or 64-bit integers
- The semantics of these instructions are identical to the semantics of the x86 SSEx-M3a6tocon -278(or)-278(64

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

Convert Between Double Precision and Single Precision Floating Point

Chapter 18

Performance Counters

PTLsim maintains hundreds of performance and statistical counters and data points as it simulates user code. In Section 4, the basic mechanisms and data structures through which PTLsim collects these data were disclosed, and a guide to extending the existing set of collection points was presented.

This section is a reference listing of all the current performance counters present in PTLsim by default. The sections below are arranged in a hierarchical tree format, just as the data are represented in PTLsim's data store.

18.1 General

As described in Section 4, PTLsim maintains a hierarchical tree of statistical data. At the root of the tree are a potentially large number of snapshots, numbered starting at 0. The final snapshot, taken just before simulation completes, is labeled as "final". Each snapshot branch contains all of the data structures described in the next few sections. Snapshots -snapshot configuration (Section 2.3); ifnywithn the "0" and "final" snapshots are provided.

In addition to the snapshots [illegible] tree witn thecbelodellaneouionsP andjust

18.2 Out of Order Core

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

- **cycles:** total number of processor cycles simulated
- **commits:** total number of committed uops
- **usercommits:** total number of committed x86 instructions
- **issues:** total number of uops issued. This includes uops issued more than once by through replay (Section 9.3).
- **ipc:** Instructions Per Cycle (IPC) statistics
  - **commit-in-uops:** average number of uops committed per cycle
  - **issue-in-uops:** average number of uops issued per cycle
  - **commit-in-user-insns:** average number of x86 instructions committed per cycle

    NOTE: Because one x86 instruction may be broken up into numerous uops, it is ***never***

- **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.
-

-