/*

- Copyright (c) 2011,2013,2016 ARM Limited
- Copyright (c) 2013 Advanced Micro Devices, Inc.
- All rights reserved.
  *
- The license below extends only to copyright in the software and shall
- not be construed as granting a license to any other intellectual
- property including but not limited to intellectual property relating
- to a hardware implementation of the functionality of the software
- licensed hereunder. You may use the software subject to the license
- terms below provided that you ensure that this notice is replicated
- unmodified and in its entirety in all distributions of the software,
- modified or unmodified, in source code or in binary form.
  *
- Copyright (c) 2004-2006 The Regents of The University of Michigan
- Copyright (c) 2009 The University of Edinburgh
- All rights reserved.
  *
- Redistribution and use in source and binary forms, with or without
- modification, are permitted provided that the following conditions are
- met: redistributions of source code must retain the above copyright
- notice, this list of conditions and the following disclaimer;
- redistributions in binary form must reproduce the above copyright
- notice, this list of conditions and the following disclaimer in the
- documentation and/or other materials provided with the distribution;
- neither the name of the copyright holders nor the names of its
- contributors may be used to endorse or promote products derived from
- this software without specific prior written permission.
  *
- THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
- "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
- LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
- A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
- OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
- SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
- LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
- DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
- THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
- (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE

- OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
  *
- Authors: Kevin Lim
- Timothy M. Jones
  */

#ifndef **CPU_BASE_DYN_INST_HH__ #define __CPU_BASE_DYN_INST_HH**

#include
#include
#include
#include
#include

#include "arch/generic/tlb.hh"
#include "arch/utility.hh"
#include "base/trace.hh"
#include "config/the_isa.hh"
#include "cpu/checker/cpu.hh"
#include "cpu/exec_context.hh"
#include "cpu/exetrace.hh"
#include "cpu/inst_res.hh"
#include "cpu/inst_seq.hh"
#include "cpu/o3/comm.hh"
#include "cpu/op_class.hh"
#include "cpu/static_inst.hh"
#include "cpu/translation.hh"
#include "debug/Reexecute.hh"
#include "mem/packet.hh"
#include "mem/request.hh"
#include "sim/byteswap.hh"
#include "sim/system.hh"

/**

- @file
- Defines a dynamic instruction context.
  */

template
class BaseDynInst : public ExecContext, public RefCounted
{
public:

// Typedef for the CPU.
typedef typename Impl::CPUType ImplCPU;
typedef typename ImplCPU::ImplState ImplState;
using VecRegContainer = TheISA::VecRegContainer;

```
// The DynInstPtr type.
typedef typename Impl::DynInstPtr DynInstPtr;
typedef RefCountingPtr<BaseDynInst<Impl> > BaseDynInstPtr;

// The list of instructions iterator type.
typedef typename std::list<DynInstPtr>::iterator ListIt;

enum {
    MaxInstSrcRegs = TheISA::MaxInstSrcRegs,        /// Max source regs
    MaxInstDestRegs = TheISA::MaxInstDestRegs       /// Max dest regs
};
```

protected:
enum Status {
IqEntry, /// Instruction is in the IQ
RobEntry, /// Instruction is in the ROB
LsqEntry, /// Instruction is in the LSQ
Completed, /// Instruction has completed
ResultReady, /// Instruction has its result
CanIssue, /// Instruction can issue and execute
Issued, /// Instruction has issued
Executed, /// Instruction has executed
CanCommit, /// Instruction can commit
AtCommit, /// Instruction has reached commit
Committed, /// Instruction has committed
Squashed, /// Instruction is squashed
SquashedInIQ, /// Instruction is squashed in the IQ
SquashedInLSQ, /// Instruction is squashed in the LSQ
SquashedInROB, /// Instruction is squashed in the ROB
RecoverInst, /// Is a recover instruction
BlockingInst, /// Is a blocking instruction
ThreadsyncWait, /// Is a thread synchronization instruction
SerializeBefore, /// Needs to serialize on
/// instructions ahead of it
SerializeAfter, /// Needs to serialize instructions behind it
SerializeHandled, /// Serialization has been handled
NeedBypass,
Reexecuted,

Reexecuting,
SquashDueToReexecute,
NumStatus

```
};

enum Flags {
    NotAnInst,
    TranslationStarted,
    TranslationCompleted,
    PossibleLoadViolation,
    HitExternalSnoop,
    EffAddrValid,
    RecordResult,
    Predicate,
    PredTaken,
    IsStrictlyOrdered,
    ReqMade,
    MemOpDone,
    NeedReexecute,
    MaxFlags
};
```

public:
uint64_t needpdt = 0;
/** The sequence number of the instruction. */
InstSeqNum seqNum = 0;

```
StoreSeqNum maybeBypassSSN = 0;
/** The store sequence number of the instruction. */
StoreSeqNum SSN = 0;

StoreSeqNum gSSN = 0;

StoreSeqNum bypassSSN = 0;

StoreSeqNum diffSSN = 0;   // gloabSSN - SSNbypassSSN;
/** The StaticInst used by this BaseDynInst. */
const StaticInstPtr staticInst;

DynInstPtr BypassInst = nullptr;

/** Pointer to the Impl's CPU object. */
ImplCPU *cpu;

BaseCPU *getCpuPtr() { return cpu; }

/** Pointer to the thread state. */
ImplState *thread;
```

```
/** The kind of fault this instruction has generated. */
Fault fault;

/** InstRecord that tracks this instructions. */
Trace::InstRecord *traceData;

std::vector<RegId> addSrcReg;
```

protected:
/** The result of the instruction; assumes an instruction can have many
* destination registers.
*/
std::queue instResult;

```
/** PC state for this instruction. */
TheISA::PCState pc;

/* An amalgamation of a lot of boolean values into one */
std::bitset<MaxFlags> instFlags;

/** The status of this BaseDynInst.  Several bits can be set. */
std::bitset<NumStatus> status;

 /** Whether or not the source register is ready.
  *  @todo: Not sure this should be here vs the derived class.
  */
std::bitset<MaxInstSrcRegs> _readySrcRegIdx;
```

public:
/** The thread this instruction is from. */
ThreadID threadNumber;

```
/** Iterator pointing to this BaseDynInst in the list of all insts. */
ListIt instListIt;

////////////////////// Branch Data ///////////////
/** Predicted PC state after this instruction. */
TheISA::PCState predPC;

/** The Macroop if one exists */
const StaticInstPtr macroop;

/** How many source registers are ready. */
uint8_t readyRegs;
```

public:
////////////////////// Load Store Data //////////////////////
/** The effective virtual address (lds & stores only). */
Addr effAddr;

```
/** bypass ffective virtual address **/
Addr bpeffAddr;

/** The effective physical address. */
Addr physEffAddrLow;

/** The effective physical address
 *  of the second request for a split request
 */

Addr physEffAddrHigh;

/** The memory request flags (from translation). */
unsigned memReqFlags;

/** data address space ID, for loads & stores. */
short asid;

/** The size of the request */
uint8_t effSize;

/** Pointer to the data for the memory access. */
uint8_t *memData;

/** Pointer to the data for the Reexecute memory access.**/

uint8_t *reexecute_memData;

/** Load queue index. */
int16_t lqIdx;

/** Store queue index. */
int16_t sqIdx;


////////////////////// TLB Miss //////////////////////
/**
 * Saved memory requests (needed when the DTB address translation is
 * delayed due to a hw page table walk).
 */
RequestPtr savedReq;
RequestPtr savedSreqLow;
RequestPtr savedSreqHigh;

////////////////////// Checker //////////////////////
// Need a copy of main request pointer to verify on writes.
RequestPtr reqToVerify;
```

protected:
/** Flattened register index of the destination registers of this
* instruction.
*/
std::array _flatDestRegIdx;

```
/** Physical register index of the destination registers of this
 *  instruction.
 */
std::array<PhysRegIdPtr, TheISA::MaxInstDestRegs> _destRegIdx;

/** Physical register index of the source registers of this
 *  instruction.
 */
std::array<PhysRegIdPtr, TheISA::MaxInstSrcRegs> _srcRegIdx;

/** Physical register index of the previous producers of the
 *  architected destinations.
 */
std::array<PhysRegIdPtr, TheISA::MaxInstDestRegs> _prevDestRegIdx;
```

public:
/** Records changes to result? */
void recordResult(bool f) { instFlags[RecordResult] = f; }

```
/** Is the effective virtual address valid. */
bool effAddrValid() const { return instFlags[EffAddrValid]; }

/** Whether or not the memory operation is done. */
bool memOpDone() const { return instFlags[MemOpDone]; }
void memOpDone(bool f) { instFlags[MemOpDone] = f; }

bool notAnInst() const { return instFlags[NotAnInst]; }
void setNotAnInst() { instFlags[NotAnInst] = true; }


////////////////////////////////////////
//
// INSTRUCTION EXECUTION
//
////////////////////////////////////////

void demapPage(Addr vaddr, uint64_t asn)
{
    cpu->demapPage(vaddr, asn);
}
void demapInstPage(Addr vaddr, uint64_t asn)
{
    cpu->demapPage(vaddr, asn);
```

```
}
void demapDataPage(Addr vaddr, uint64_t asn)
{
    cpu->demapPage(vaddr, asn);
}

Fault initiateMemRead(Addr addr, unsigned size, Request::Flags flags);

Fault writeMem(uint8_t *data, unsigned size, Addr addr,
               Request::Flags flags, uint64_t *res);

/** Splits a request in two if it crosses a dcache block. */
void splitRequest(const RequestPtr &req, RequestPtr &sreqLow,
                  RequestPtr &sreqHigh);

/** Initiate a DTB address translation. */
void initiateTranslation(const RequestPtr &req, const RequestPtr &sreqLow,
                         const RequestPtr &sreqHigh, uint64_t *res,
                         BaseTLB::Mode mode);

/** Finish a DTB address translation. */
void finishTranslation(WholeTranslationState *state);

/** True if the DTB address translation has started. */
bool translationStarted() const { return instFlags[TranslationStarted]; }
void translationStarted(bool f) { instFlags[TranslationStarted] = f; }

/** True if the DTB address translation has completed. */
bool translationCompleted() const { return instFlags[TranslationCompleted]; }
void translationCompleted(bool f) { instFlags[TranslationCompleted] = f; }

/** True if this address was found to match a previous load and they issued
 * out of order. If that happend, then it's only a problem if an incoming
 * snoop invalidate modifies the line, in which case we need to squash.
 * If nothing modified the line the order doesn't matter.
 */
bool possibleLoadViolation() const { return instFlags[PossibleLoadViolation]; }
void possibleLoadViolation(bool f) { instFlags[PossibleLoadViolation] = f; }

/** True if the address hit a external snoop while sitting in the LSQ.
 * If this is true and a older instruction sees it, this instruction must
 * reexecute
 */
bool hitExternalSnoop() const { return instFlags[HitExternalSnoop]; }
void hitExternalSnoop(bool f) { instFlags[HitExternalSnoop] = f; }

/**
 * Returns true if the DTB address translation is being delayed due to a hw
 * page table walk.
 */
bool isTranslationDelayed() const
{
    return (translationStarted() && !translationCompleted());
}
```

public:
#ifdef DEBUG
void dumpSNList();
#endif

```
/** Returns the physical register index of the i'th destination
 *  register.
 */
PhysRegIdPtr renamedDestRegIdx(int idx) const
{
    return _destRegIdx[idx];
}

/** Returns the physical register index of the i'th source register. */
PhysRegIdPtr renamedSrcRegIdx(int idx) const
{
    assert(TheISA::MaxInstSrcRegs > idx);
    return _srcRegIdx[idx];
}

/** Returns the flattened register index of the i'th destination
 *  register.
 */
const RegId& flattenedDestRegIdx(int idx) const
{
    return _flatDestRegIdx[idx];
}

/** Returns the physical register index of the previous physical register
 *  that remapped to the same logical register index.
 */
PhysRegIdPtr prevDestRegIdx(int idx) const
{
    return _prevDestRegIdx[idx];
}

/** Renames a destination register to a physical register.  Also records
 *  the previous physical register that the logical register mapped to.
 */
void renameDestReg(int idx,
                   PhysRegIdPtr renamed_dest,
                   PhysRegIdPtr previous_rename)
{
    _destRegIdx[idx] = renamed_dest;
    _prevDestRegIdx[idx] = previous_rename;
}

/** Renames a source logical register to the physical register which
 *  has/will produce that logical register's result.
 *  @todo: add in whether or not the source register is ready.
 */
void renameSrcReg(int idx, PhysRegIdPtr renamed_src)
{
```

```
    _srcRegIdx[idx] = renamed_src;
}

/** Flattens a destination architectural register index into a logical
 * index.
 */
void flattenDestReg(int idx, const RegId& flattened_dest)
{
    _flatDestRegIdx[idx] = flattened_dest;
}
/** BaseDynInst constructor given a binary instruction.
 *  @param staticInst A StaticInstPtr to the underlying instruction.
 *  @param pc The PC state for the instruction.
 *  @param predPC The predicted next PC state for the instruction.
 *  @param seq_num The sequence number of the instruction.
 *  @param cpu Pointer to the instruction's CPU.
 */
BaseDynInst(const StaticInstPtr &staticInst, const StaticInstPtr &macroop,
            TheISA::PCState pc, TheISA::PCState predPC,
            InstSeqNum seq_num, StoreSeqNum ssn, ImplCPU *cpu);

/** BaseDynInst constructor given a StaticInst pointer.
 *  @param _staticInst The StaticInst for this BaseDynInst.
 */
BaseDynInst(const StaticInstPtr &staticInst, const StaticInstPtr &macroop);

/** BaseDynInst destructor. */
~BaseDynInst();
```

private:
/** Function to initialize variables in the constructors. */
void initVars();

public:
/** Dumps out contents of this BaseDynInst. */
void dump();

```
/** Dumps out contents of this BaseDynInst into given string. */
void dump(std::string &outstring);

/** Read this CPU's ID. */
int cpuId() const { return cpu->cpuId(); }

/** Read this CPU's Socket ID. */
uint32_t socketId() const { return cpu->socketId(); }

/** Read this CPU's data requestor ID */
MasterID masterId() const { return cpu->dataMasterId(); }

/** Read this context's system-wide ID **/
ContextID contextId() const { return thread->contextId(); }
```

```
/** Returns the fault type. */
Fault getFault() const { return fault; }

/** Checks whether or not this instruction has had its branch target
 *  calculated yet.  For now it is not utilized and is hacked to be
 *  always false.
 *  @todo: Actually use this instruction.
 */
bool doneTargCalc() { return false; }

/** Set the predicted target of this current instruction. */
void setPredTarg(const TheISA::PCState &_predPC)
{
    predPC = _predPC;
}

const TheISA::PCState &readPredTarg() { return predPC; }

/** Returns the predicted PC immediately after the branch. */
Addr predInstAddr() { return predPC.instAddr(); }

/** Returns the predicted PC two instructions after the branch */
Addr predNextInstAddr() { return predPC.nextInstAddr(); }

/** Returns the predicted micro PC after the branch */
Addr predMicroPC() { return predPC.microPC(); }

/** Returns whether the instruction was predicted taken or not. */
bool readPredTaken()
{
    return instFlags[PredTaken];
}

void setPredTaken(bool predicted_taken)
{
    instFlags[PredTaken] = predicted_taken;
}

/** Returns whether the instruction mispredicted. */
bool mispredicted()
{
    TheISA::PCState tempPC = pc;
    TheISA::advancePC(tempPC, staticInst);
    return !(tempPC == predPC);
}

//
//  Instruction types.  Forward checks to StaticInst object.
//
bool isNop()          const { return staticInst->isNop(); }
bool isMemRef()       const { return staticInst->isMemRef(); }
bool isLoad()         const { return staticInst->isLoad(); }
bool isStore()        const { return staticInst->isStore(); }
bool isAtomic()       const { return staticInst->isAtomic(); }
bool isStoreConditional() const
```

```
{ return staticInst->isStoreConditional(); }
bool isInstPrefetch() const { return staticInst->isInstPrefetch(); }
bool isDataPrefetch() const { return staticInst->isDataPrefetch(); }
bool isInteger()      const { return staticInst->isInteger(); }
bool isFloating()     const { return staticInst->isFloating(); }
bool isVector()       const { return staticInst->isVector(); }
bool isControl()      const { return staticInst->isControl(); }
bool isCall()         const { return staticInst->isCall(); }
bool isReturn()       const { return staticInst->isReturn(); }
bool isDirectCtrl()   const { return staticInst->isDirectCtrl(); }
bool isIndirectCtrl() const { return staticInst->isIndirectCtrl(); }
bool isCondCtrl()     const { return staticInst->isCondCtrl(); }
bool isUncondCtrl()   const { return staticInst->isUncondCtrl(); }
bool isCondDelaySlot() const { return staticInst->isCondDelaySlot(); }
bool isThreadSync()   const { return staticInst->isThreadSync(); }
bool isSerializing()  const { return staticInst->isSerializing(); }
bool isSerializeBefore() const
{ return staticInst->isSerializeBefore() || status[SerializeBefore]; }
bool isSerializeAfter() const
{ return staticInst->isSerializeAfter() || status[SerializeAfter]; }
bool isSquashAfter() const { return staticInst->isSquashAfter(); }
bool isMemBarrier()   const { return staticInst->isMemBarrier(); }
bool isWriteBarrier() const { return staticInst->isWriteBarrier(); }
bool isNonSpeculative() const { return staticInst->isNonSpeculative(); }
bool isQuiesce() const { return staticInst->isQuiesce(); }
bool isIprAccess() const { return staticInst->isIprAccess(); }
bool isUnverifiable() const { return staticInst->isUnverifiable(); }
bool isSyscall() const { return staticInst->isSyscall(); }
bool isMacroop() const { return staticInst->isMacroop(); }
bool isMicroop() const { return staticInst->isMicroop(); }
bool isDelayedCommit() const { return staticInst->isDelayedCommit(); }
bool isLastMicroop() const { return staticInst->isLastMicroop(); }
bool isFirstMicroop() const { return staticInst->isFirstMicroop(); }
bool isMicroBranch() const { return staticInst->isMicroBranch(); }

/** Temporarily sets this instruction as a serialize before instruction. */
void setSerializeBefore() { status.set(SerializeBefore); }

/** Clears the serializeBefore part of this instruction. */
void clearSerializeBefore() { status.reset(SerializeBefore); }

/** Checks if this serializeBefore is only temporarily set. */
bool isTempSerializeBefore() { return status[SerializeBefore]; }

/** Temporarily sets this instruction as a serialize after instruction. */
void setSerializeAfter() { status.set(SerializeAfter); }

/** Clears the serializeAfter part of this instruction.*/
void clearSerializeAfter() { status.reset(SerializeAfter); }

/** Checks if this serializeAfter is only temporarily set. */
bool isTempSerializeAfter() { return status[SerializeAfter]; }

/** Sets the serialization part of this instruction as handled. */
void setSerializeHandled() { status.set(SerializeHandled); }
```

```
/** Checks if the serialization part of this instruction has been
 *  handled.  This does not apply to the temporary serializing
 *  state; it only applies to this instruction's own permanent
 *  serializing state.
 */
bool isSerializeHandled() { return status[SerializeHandled]; }

/** Returns the opclass of this instruction. */
OpClass opClass() const { return staticInst->opClass(); }

/** Returns the branch target address. */
TheISA::PCState branchTarget() const
{ return staticInst->branchTarget(pc); }

void addRegToSrc(RegId x){
  addSrcReg.push_back(x);
}
/** Returns the number of source registers. */
int8_t numSrcRegs() const {
  return staticInst->numSrcRegs() + int8_t(addSrcReg.size());
}

/** Returns the number of destination registers. */
int8_t numDestRegs() const { return staticInst->numDestRegs(); }

// the following are used to track physical register usage
// for machines with separate int & FP reg files
int8_t numFPDestRegs()  const { return staticInst->numFPDestRegs(); }
int8_t numIntDestRegs() const { return staticInst->numIntDestRegs(); }
int8_t numCCDestRegs() const { return staticInst->numCCDestRegs(); }
int8_t numVecDestRegs() const { return staticInst->numVecDestRegs(); }
int8_t numVecElemDestRegs() const {
    return staticInst->numVecElemDestRegs();
}

/** Returns the logical register index of the i'th destination register. */
const RegId& destRegIdx(int i) const { return staticInst->destRegIdx(i); }

/** Returns the logical register index of the i'th source register. */
const RegId& srcRegIdx(int i) const {
  if (i < staticInst->numSrcRegs())
    return staticInst->srcRegIdx(i);
  else{
    return addSrcReg[i - staticInst->numSrcRegs()];
  }
}

/** Return the size of the instResult queue. */
uint8_t resultSize() { return instResult.size(); }

/** Pops a result off the instResult queue.
 * If the result stack is empty, return the default value.
 * */
InstResult popResult(InstResult dflt = InstResult())
{
    if (!instResult.empty()) {
```

```
            InstResult t = instResult.front();
            instResult.pop();
            return t;
        }
        return dflt;
    }

    /** Pushes a result onto the instResult queue. */
    /** @{ */
    /** Scalar result. */
    template<typename T>
    void setScalarResult(T&& t)
    {
        if (instFlags[RecordResult]) {
            instResult.push(InstResult(std::forward<T>(t),
                        InstResult::ResultType::Scalar));
        }
    }

    /** Full vector result. */
    template<typename T>
    void setVecResult(T&& t)
    {
        if (instFlags[RecordResult]) {
            instResult.push(InstResult(std::forward<T>(t),
                        InstResult::ResultType::VecReg));
        }
    }

    /** Vector element result. */
    template<typename T>
    void setVecElemResult(T&& t)
    {
        if (instFlags[RecordResult]) {
            instResult.push(InstResult(std::forward<T>(t),
                        InstResult::ResultType::VecElem));
        }
    }
    /** @} */

    /** Records an integer register being set to a value. */
    void setIntRegOperand(const StaticInst *si, int idx, IntReg val)
    {
        setScalarResult(val);
    }

    /** Records a CC register being set to a value. */
    void setCCRegOperand(const StaticInst *si, int idx, CCReg val)
    {
        setScalarResult(val);
    }

    /** Records an fp register being set to a value. */
    void setFloatRegOperand(const StaticInst *si, int idx, FloatReg val)
    {
        setScalarResult(val);
```

```
}

/** Record a vector register being set to a value */
void setVecRegOperand(const StaticInst *si, int idx,
        const VecRegContainer& val)
{
    setVecResult(val);
}

/** Records an fp register being set to an integer value. */
void
setFloatRegOperandBits(const StaticInst *si, int idx, FloatRegBits val)
{
    setScalarResult(val);
}

/** Record a vector register being set to a value */
void setVecElemOperand(const StaticInst *si, int idx, const VecElem val)
{
    setVecElemResult(val);
}

/** Records that one of the source registers is ready. */
void markSrcRegReady();

/** Marks a specific register as ready. */
void markSrcRegReady(RegIndex src_idx);

/** Returns if a source register is ready. */
bool isReadySrcRegIdx(int idx) const
{
    return this->_readySrcRegIdx[idx];
}

/** Sets this instruction as completed. */
void setCompleted() { status.set(Completed); }

/** Returns whether or not this instruction is completed. */
bool isCompleted() const { return status[Completed]; }

/** Marks the result as ready. */
void setResultReady() { status.set(ResultReady); }

/** Returns whether or not the result is ready. */
bool isResultReady() const { return status[ResultReady]; }

/** Sets this instruction as ready to issue. */
void setCanIssue() { status.set(CanIssue); }

/** Returns whether or not this instruction is ready to issue. */
bool readyToIssue() const { return status[CanIssue]; }

/** Clears this instruction being able to issue. */
void clearCanIssue() { status.reset(CanIssue); }

/** Sets this instruction as issued from the IQ. */
```

```
    void setIssued() { status.set(Issued); }

    /** Returns whether or not this instruction has issued. */
    bool isIssued() const { return status[Issued]; }

    /** Clears this instruction as being issued. */
    void clearIssued() { status.reset(Issued); }

    /** Sets this instruction as executed. */
    void setExecuted() { status.set(Executed); }

    /** Returns whether or not this instruction has executed. */
    bool isExecuted() const { return status[Executed]; }

    /** Sets this instruction as executed. */
    void setReexecuted() { status.set(Reexecuted);}

    /** Clears this instruction as executed. */
    void clearReexecuted() { status.reset(Reexecuted);}

    /** Returns whether or not this instruction has Reexecuted. */
    bool isReexecuted() const { return status[Reexecuted]; }

    /** Sets this instruction as Reexecuting. */
    void setReexecuting() { status.set(Reexecuting);}

    /** Returns whether or not this instruction is Reexecuting. */
    bool isReexecuting() const { return status[Reexecuting]; }

    /** Sets this instruction as NeedBypass. */
    void setNeedBypass() { status.set(NeedBypass);}

    void clearNeedBypass() { status.reset(NeedBypass);}

    /** Returns whether or not this instruction is NeedBypass. */
    bool isNeedBypass() const { return status[NeedBypass]; }

    /** Sets this instruction as NEED SQUASH. */
    void setSquashDueToReexecute() { status.set(SquashDueToReexecute);}

    /** Returns whether or not this instruction NEED SQUASH. */
    bool isSquashDueToReexecute() const { return status[SquashDueToReexecute];}

    /** Clears this instruction as SquashDueToReexecute. */
    void clearSquashDueToReexecute() { status.reset(SquashDueToReexecute);}

    /** Sets this instruction as ready to commit. */
    void setCanCommit() { status.set(CanCommit); }

    /** Clears this instruction as being ready to commit. */
    void clearCanCommit() { status.reset(CanCommit); }

    /** Returns whether or not this instruction is ready to commit. */
    bool readyToCommit() const { return status[CanCommit]; }

    //bool readyToFinish() const
```

```
// { return status[CanCommit]&status[Reexecuted]; }
bool readyToFinish() const { return status[Reexecuted]; }

void setAtCommit() { status.set(AtCommit); }

bool isAtCommit() { return status[AtCommit]; }

/** Sets this instruction as committed. */
void setCommitted() { status.set(Committed); }

/** Returns whether or not this instruction is committed. */
bool isCommitted() const { return status[Committed]; }

/** Sets this instruction as squashed. */
void setSquashed() { status.set(Squashed); }

/** Returns whether or not this instruction is squashed. */
bool isSquashed() const { return status[Squashed]; }

//Instruction Queue Entry
//-----------------------
/** Sets this instruction as a entry the IQ. */
void setInIQ() { status.set(IqEntry); }

/** Sets this instruction as a entry the IQ. */
void clearInIQ() { status.reset(IqEntry); }

/** Returns whether or not this instruction has issued. */
bool isInIQ() const { return status[IqEntry]; }

/** Sets this instruction as squashed in the IQ. */
void setSquashedInIQ() { status.set(SquashedInIQ); status.set(Squashed);}

/** Returns whether or not this instruction is squashed in the IQ. */
bool isSquashedInIQ() const { return status[SquashedInIQ]; }


//Load / Store Queue Functions
//-----------------------
/** Sets this instruction as a entry the LSQ. */
void setInLSQ() { status.set(LsqEntry); }

/** Sets this instruction as a entry the LSQ. */
void removeInLSQ() { status.reset(LsqEntry); }

/** Returns whether or not this instruction is in the LSQ. */
bool isInLSQ() const { return status[LsqEntry]; }

/** Sets this instruction as squashed in the LSQ. */
void setSquashedInLSQ() { status.set(SquashedInLSQ);}

/** Returns whether or not this instruction is squashed in the LSQ. */
bool isSquashedInLSQ() const { return status[SquashedInLSQ]; }


//Reorder Buffer Functions
```

```
//-----------------------
/** Sets this instruction as a entry the ROB. */
void setInROB() { status.set(RobEntry); }

/** Sets this instruction as a entry the ROB. */
void clearInROB() { status.reset(RobEntry); }

/** Returns whether or not this instruction is in the ROB. */
bool isInROB() const { return status[RobEntry]; }

/** Sets this instruction as squashed in the ROB. */
void setSquashedInROB() { status.set(SquashedInROB); }

/** Returns whether or not this instruction is squashed in the ROB. */
bool isSquashedInROB() const { return status[SquashedInROB]; }

/** Read the PC state of this instruction. */
TheISA::PCState pcState() const { return pc; }

/** Set the PC state of this instruction. */
void pcState(const TheISA::PCState &val) { pc = val; }

/** Read the PC of this instruction. */
Addr instAddr() const { return pc.instAddr(); }

/** Read the PC of the next instruction. */
Addr nextInstAddr() const { return pc.nextInstAddr(); }

/**Read the micro PC of this instruction. */
Addr microPC() const { return pc.microPC(); }

bool readNeedReexecute(){
  return instFlags[NeedReexecute];
}

void setNeedReexecute(bool val)
{
  instFlags[NeedReexecute] = val;
}
bool readPredicate()
{
    return instFlags[Predicate];
}

void setPredicate(bool val)
{
    instFlags[Predicate] = val;

    if (traceData) {
        traceData->setPredicate(val);
    }
}

/** Sets the ASID. */
void setASID(short addr_space_id) { asid = addr_space_id; }
```

```
    /** Sets the thread id. */
    void setTid(ThreadID tid) { threadNumber = tid; }

    /** Sets the pointer to the thread state. */
    void setThreadState(ImplState *state) { thread = state; }

    /** Returns the thread context. */
    ThreadContext *tcBase() { return thread->getTC(); }
```

public:
/** Returns whether or not the eff. addr. source registers are ready. */
bool eaSrcsReady();

```
    /** Is this instruction's memory access strictly ordered? */
    bool strictlyOrdered() const { return instFlags[IsStrictlyOrdered]; }

    /** Has this instruction generated a memory request. */
    bool hasRequest() { return instFlags[ReqMade]; }

    /** Returns iterator to this instruction in the list of all insts. */
    ListIt &getInstListIt() { return instListIt; }

    /** Sets iterator for this instruction in the list of all insts. */
    void setInstListIt(ListIt _instListIt) { instListIt = _instListIt; }
```

public:
/** Returns the number of consecutive store conditional failures. */
unsigned int readStCondFailures() const
{ return thread->storeCondFailures; }

```
    /** Sets the number of consecutive store conditional failures. */
    void setStCondFailures(unsigned int sc_failures)
    { thread->storeCondFailures = sc_failures; }
```

public:
// monitor/mwait funtions
void armMonitor(Addr address) { cpu->armMonitor(threadNumber, address); }
bool mwait(PacketPtr pkt) { return cpu->mwait(threadNumber, pkt); }
void mwaitAtomic(ThreadContext *tc)
{ return cpu->mwaitAtomic(threadNumber, tc, cpu->dtb); }
AddressMonitor *getAddrMonitor()
{ return cpu->getCpuAddrMonitor(threadNumber); }
};

template
Fault
BaseDynInst::initiateMemRead(Addr addr, unsigned size,
Request::Flags flags)
{
DPRINTF(Reexecute,"Reexecute-initiateMemRead-1\n");
instFlags[ReqMade] = true;
RequestPtr req = NULL;
RequestPtr sreqLow = NULL;
RequestPtr sreqHigh = NULL;

```
if (instFlags[ReqMade] && translationStarted()) {
    DPRINTF(Reexecute,"Reexecute-initiateMemRead-2\n");
    req = savedReq;
    sreqLow = savedSreqLow;
    sreqHigh = savedSreqHigh;
    DPRINTF(Reexecute,"Reexecute-initiateMemRead-2-2\n");
} else {
  DPRINTF(Reexecute,"Reexecute-initiateMemRead-3\n");
    req = std::make_shared<Request>(
        asid, addr, size, flags, masterId(),
        this->pc.instAddr(), thread->contextId());

    req->taskId(cpu->taskId());

    // Only split the request if the ISA supports unaligned accesses.
    if (TheISA::HasUnalignedMemAcc) {
        splitRequest(req, sreqLow, sreqHigh);
    }
    DPRINTF(Reexecute,"Reexecute-initiateMemRead-3-2\n");
    initiateTranslation(req, sreqLow, sreqHigh, NULL, BaseTLB::Read);
    DPRINTF(Reexecute,"Reexecute-initiateMemRead-3-3\n");
}

if (translationCompleted()) {
  DPRINTF(Reexecute,"Reexecute-initiateMemRead-4\n");
    if (fault == NoFault) {
      DPRINTF(Reexecute,"Reexecute-initiateMemRead-4-1\n");
        effAddr = req->getVaddr();
        DPRINTF(Reexecute,"Reexecute-initiateMemRead-4-2\n");
        effSize = size;
        instFlags[EffAddrValid] = true;
        DPRINTF(Reexecute,"Reexecute-initiateMemRead-4-3\n");
        if (cpu->checker) {
            reqToVerify = std::make_shared<Request>(*req);
        }
        fault = cpu->read(req, sreqLow, sreqHigh, lqIdx);
    } else {
        // Commit will have to clean up whatever happened.  Set this
        // instruction as executed.
        this->setExecuted();
```

```
    }
}

if (traceData)
    traceData->setMem(addr, size, flags);
```

DPRINTF(Reexecute,"Reexecute-initiateMemRead-5\n");
return fault;
}

template
Fault
BaseDynInst::writeMem(uint8_t *data, unsigned size, Addr addr,
Request::Flags flags, uint64_t *res)
{
if (traceData)
traceData->setMem(addr, size, flags);

```
instFlags[ReqMade] = true;
RequestPtr req = NULL;
RequestPtr sreqLow = NULL;
RequestPtr sreqHigh = NULL;

if (instFlags[ReqMade] && translationStarted()) {
    req = savedReq;
    sreqLow = savedSreqLow;
    sreqHigh = savedSreqHigh;
} else {
    req = std::make_shared<Request>(
        asid, addr, size, flags, masterId(),
        this->pc.instAddr(), thread->contextId());

    req->taskId(cpu->taskId());

    // Only split the request if the ISA supports unaligned accesses.
    if (TheISA::HasUnalignedMemAcc) {
        splitRequest(req, sreqLow, sreqHigh);
    }
    initiateTranslation(req, sreqLow, sreqHigh, res, BaseTLB::Write);
}

if (fault == NoFault && translationCompleted()) {
    effAddr = req->getVaddr();
    effSize = size;
    instFlags[EffAddrValid] = true;

    if (cpu->checker) {
        reqToVerify = std::make_shared<Request>(*req);
    }
```

```
        fault = cpu->write(req, sreqLow, sreqHigh, data, sqIdx);
    }

    return fault;
```

}

template
inline void
BaseDynInst::splitRequest(const RequestPtr &req, RequestPtr &sreqLow,
RequestPtr &sreqHigh)
{
// Check to see if the request crosses the next level block boundary.
unsigned block_size = cpu->cacheLineSize();
Addr addr = req->getVaddr();
Addr split_addr = roundDown(addr + req->getSize() - 1, block_size);
assert(split_addr <= addr || split_addr - addr < block_size);

```
    // Spans two blocks.
    if (split_addr > addr) {
        req->splitOnVaddr(split_addr, sreqLow, sreqHigh);
    }
```

}

template
inline void
BaseDynInst::initiateTranslation(const RequestPtr &req,
const RequestPtr &sreqLow,
const RequestPtr &sreqHigh,
uint64_t *res,
BaseTLB::Mode mode)
{
translationStarted(true);

```
    if (!TheISA::HasUnalignedMemAcc || sreqLow == NULL) {
        WholeTranslationState *state =
            new WholeTranslationState(req, NULL, res, mode);

        // One translation if the request isn't split.
        DataTranslation<BaseDynInstPtr> *trans =
            new DataTranslation<BaseDynInstPtr>(this, state);
```

```
        cpu->dtb->translateTiming(req, thread->getTC(), trans, mode);

        if (!translationCompleted()) {
            // The translation isn't yet complete, so we can't possibly have a
            // fault. Overwrite any existing fault we might have from a previous
            // execution of this instruction (e.g. an uncachable load that
            // couldn't execute because it wasn't at the head of the ROB).
            fault = NoFault;

            // Save memory requests.
            savedReq = state->mainReq;
            savedSreqLow = state->sreqLow;
            savedSreqHigh = state->sreqHigh;
        }
    } else {
        WholeTranslationState *state =
            new WholeTranslationState(req, sreqLow, sreqHigh, NULL, res, mode);

        // Two translations when the request is split.
        DataTranslation<BaseDynInstPtr> *transLow =
            new DataTranslation<BaseDynInstPtr>(this, state, 0);
        DataTranslation<BaseDynInstPtr> *transHigh =
            new DataTranslation<BaseDynInstPtr>(this, state, 1);

        cpu->dtb->translateTiming(sreqLow, thread->getTC(), transLow, mode);
        cpu->dtb->translateTiming(sreqHigh, thread->getTC(), transHigh, mode);

        if (!translationCompleted()) {
            // The translation isn't yet complete, so we can't possibly have a
            // fault. Overwrite any existing fault we might have from a previous
            // execution of this instruction (e.g. an uncachable load that
            // couldn't execute because it wasn't at the head of the ROB).
            fault = NoFault;

            // Save memory requests.
            savedReq = state->mainReq;
            savedSreqLow = state->sreqLow;
            savedSreqHigh = state->sreqHigh;
        }
    }
```

}

template
inline void
BaseDynInst::finishTranslation(WholeTranslationState *state)
{
fault = state->getFault();

```
    instFlags[IsStrictlyOrdered] = state->isStrictlyOrdered();
```

```
  if (fault == NoFault) {
      // save Paddr for a single req
      physEffAddrLow = state->getPaddr();

      // case for the request that has been split
      if (state->isSplit) {
        physEffAddrLow = state->sreqLow->getPaddr();
        physEffAddrHigh = state->sreqHigh->getPaddr();
      }

      memReqFlags = state->getFlags();

      if (state->mainReq->isCondSwap()) {
          assert(state->res);
          state->mainReq->setExtraData(*state->res);
      }

  } else {
      state->deleteReqs();
  }
  delete state;

  translationCompleted(true);
```

}

#endif // **CPU_BASE_DYN_INST_HH**