LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

A TRS Model for a Modern Microprocessor

Computation Structures Group Memo 408
June 25, 1998

Lisa A. Poyneer, James C. Hoe and Arvind

This paper describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this work is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Ft. Huachuca contract DABT63-95-C-0150.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

A TRS Model for a Modern Microprocessor

Lisa A. Poyneer, James C. Hoe and Arvind

June 25, 1998

1 Background

The AX instruction set, a minimalist RISC ISA, was described by Xiaowei Shen and Arvind in [1] and [2]. Term rewriting system models of AX were created, described and proved equivalent. The first model was a single-cycle non-pipelined model called P_B. The second model, P_S was a speculative processor that used register renaming. Please see [2] for complete descriptions of term rewriting systems, the AX ISA and these two models. A more fully-featured model, P_P, is presented here. It is a pipelined, speculative processor that uses register renaming and multiple functional units for execution. This memo describes and explains P_P. In addition, an implementation of P_P in Java is described in the Appendix.

1.1 Definition

P_P contains five main functional units. Communication between these units is done with queues. The flow of information among these units is illustrated in Figure 1. The Funit contains the program counter, instruction memory and Branch Translation Buffer (BTB.) Instructions are fetched from the memory and sent to the decode unit. The BTB provides predictions of the next pc. The Decode contains the register file, Re-Order Buffer (ROB) and the reset counter. The ROB does most of the work in the decode unit. It uses register renaming, assigning tags to register and keeping track of updating tags and registers with values. From the Decode, instructions are dispatched for execution to the three smaller units: Exec for ALU operations, Mem for Memory accesses and BP for branch resolution. The results are returned to Decode, where instructions are either committed if correctly speculated, or removed if incorrectly speculated, with appropriate reseting of the instruction fetch. The grammar for P_P is defined in Figure 2.

2 Rules

The rules below are formatted to simplify comprehension. In each old TRS expression the key triggering terms are in bold face. In each new expression the changed terms are in bold face. Operations described in the *where* clauses are abstractions for concepts discussed in the text. The many operations on ROB are semantically defined in Section 2.10.

2.1 Fetch

The Funit optimistically fetches rules, speculating with the predictions of the BTB. When a prediction has been determined to be incorrect, a message arrives from the Decode unit via the reset

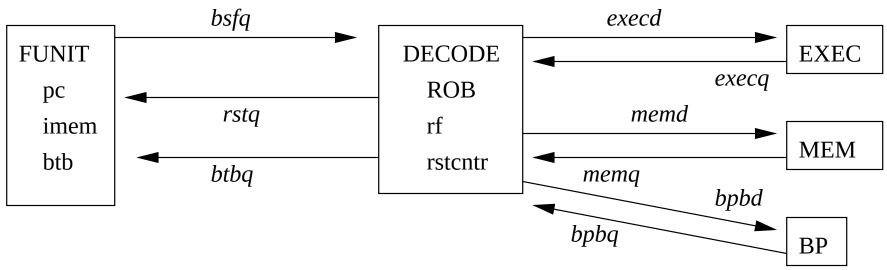

Figure 1: **Schematic of the units of P_P.** Information flows through the processor through the queues connecting the units. Speculative instruction fetch and branch prediction occur in Funit. Instruction decode and dispatch, as well as committal and completion occur within the Decode Unit. The Re-Order Buffer (ROB) takes care of register renaming. The three smaller functional units take care of actual execution of the instructions.

queue (rstq), causing instruction fetch to continue at the new, correct pc provided in the message. Update messages for the BTB are received from the branch translation buffer queue (btbq.) Though we do not discuss them here, there are many different branch prediction schemes possible.

Instruction Fetch Rule

Funit(pc, btb, prog): btbq, ϵ : bsfq

$\Longrightarrow$ Funit(pc', btb, prog): btbq, ϵ : **bsfq;Ib(pc, pc', inst)**

where pc' = btb[pc] *and* inst = prog[pc]

Restart at new PC Rule

Funit(pc, btb, prog): btbq, **(newpc);rstq** : bsfq

$\Longrightarrow$ Funit(newpc, btb, prog): btbq, rstq : **Restart;bsfq**

Update branch prediction Rule

Funit(pc, btb, prog): **(pc', npc, res);btbq** , rstq : bsfq

$\Longrightarrow$ Funit(pc, **btb'**, prog): **btbq**, rstq : bsfq

where btb' = updateBTB(btb, pc', npc, res)

2.2 Decode

The Decode unit receives instructions from the Funit through the instruction fetch queue (bsfq.) Instructions are speculatively decoded and enqueued in the Re-Order Buffer (ROB) unless a non-zero reset counter (rstcntr) indicates that the incoming instructions are known to be invalid. Invalid instructions are discarded until a reset acknowledgment is received.

Discard mispredicted fetches Rule

Decode (rf, rob, **cntr**) : **Ib(-,-,-);bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd

if **cntr** $\neq$ **0**

$\Longrightarrow$ Decode (rf, rob, cntr) : **bsfq**, execq, memq, bpq : btbq, rstq , execd, memd, bpd

Restart on cue Rule

Decode (rf, rob, **cntr**) : **Restart;bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd

SYS	=	Sys (<FUNIT: BTBQ, RSTQ: BSFQ>, <DECODE: BSFQ, EXECQ, MEMQ, BPQ: BTBQ, RSTQ, EXECD, MEMD, BPD>, <EXEC: EXECD: EXECQ>, <MEM: MEMD: MEMQ>, <BP: BPD: BPQ>)	*System*
FUNIT	=	Funit(PC, BTB, IMEM)	*Fetch Unit*
DECODE	=	Decode(RF, ROB, RSTCNTR)	*Decode Unit*
EXEC	=	Exec	*Execution Unit*
MEM	=	Mem(v)	*Data Memory Unit*
BP	=	Bp	*Branch Resolution Unit*
ROB	=	List IRB	*Re-Order Buffer*
IMEM	=	Mem(INST)	*Instruction Memory*
RF	=	ϵ $\|$ Reg(r, v);RF	*Register File*
Mem(x)	=	ϵ $\|$ Cell(a, x);Mem(x)	*Definition of a Memory*
BTBQ	=	Queue (IA, IA, RES)	*Decode to Funit, for btb updates*
RSTQ	=	Queue (IA)	*Decode to Funit, for reset of pc*
BSF	=	Queue IB	*Funit to Decode, for instructions*
EXECQ	=	Queue IDB	*Exec to Decode*
MEMQ	=	Queue IDB	*Mem to Decode*
BPQ	=	Queue IDB	*BP to Decode*
EXECD	=	Queue IDB	*Decode to Exec*
MEMD	=	Queue IDB	*Decode to Mem*
BPD	=	Queue IDB	*Decode to Bp*
IB	=	Ib (PC, PC, INST) $\|$ Restart	*Instruction Buffer*
IDB	=	Itb (TAG, INSTTEMP)	*Instruction Template Buffer*
IBPB	=	Itb (TAG, PC, INSTTEMP)	*Instruction Template Buffer*
IRB	=	Itb (TAG, PC, INSTTEMP, STATE)	*Element of ROB*
STATE	=	Wait $\|$ Exec $\|$ Done $\|$ Miss $\|$ Kill	*States used in IRB*
RES	=	Correct $\|$ Miss	*Result of prediction, given to BTB*
INST	=	r:=loadc(tv) $\|$ r:=loadpc $\|$ r:=Op(r1, r2) $\|$ Jz(rc, ra) $\|$ r:=load(ra) $\|$ Store(ra, rv)	*Instructions*
INSTTEMP	=	r:=loadc(tv) $\|$ r:=loadpc $\|$ r:=Op(tv1,tv2) $\|$ Jz(tvc,tva,ppc) $\|$ r:=load(a) $\|$ Store(a,tv) $\|$ r:=tv $\|$ JzIncorrect(pc) $\|$ JzCorrect() $\|$ StoreDone $\|$ v	*Instructions used during execution*
Queue xs	=	ϵ $\|$ x;queue xs	*Definition of a Queue*

Figure 2: P_P **grammar**

$\Longrightarrow$ Decode (rf, rob, **cntr - 1**) : **bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd

If the rstcntr is zero, instructions are decoded. There are two cases for enqueueing instructions in the ROB. If the value of an assignment can be determined immediately (e.g. r := Loadc(v) or r := LoadPC()) there is no need for the instruction to be dispatched to an execution unit. The ROB assigns the target register a tag and enqueues that instruction with state Done.

Decode LoadC instructions Rule

Decode (rf, rob, **0**) : **Ib(pc, -, r: = Loadc(v));bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd

$\Longrightarrow$ Decode (rf, **rob'**, 0) : **bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd

where rob' = enqueue_done(pc, insttemp) *and* insttemp = v

Decode LoadPC instructions Rule

Decode (rf, rob, **0**) : **Ib(pc, -, r: = Loadpc());bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd

$\Longrightarrow$ Decode (rf, **rob'**, 0) : **bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd

where rob' = enqueue_done(pc, insttemp) *and* insttemp = pc

The second case is when the instruction requires an execution unit for the result to be determined. In this case, tags are assigned to the target register if necessary, and the instruction is enqueued with state Wait. The register operands of the instruction are looked up in the ROB and the correct tag or value is returned. The enqueued instruction will then wait to be dispatched to the appropriate functional unit.

Decode Op instructions Rule

Decode (rf, rob, **0**) : **Ib(pc, -, r: = Op(r1,r2));bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd

$\Longrightarrow$ Decode (rf, **rob'**, 0) : **bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd

where rob' = enqueue(pc, insttemp) *and* insttemp = r: = Op(vt1,vt2) *and* vt1 = lookup(r1,rob,rf) *and* vt2 = lookup(r2,rob,rf)

Decode Jz instructions Rule

Decode (rf, rob, **0**) : **Ib(pc, ppc, Jz(rc,ra));bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd

$\Longrightarrow$ Decode (rf, **rob'**, 0) : **bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd

where rob' = enqueue(pc, insttemp) *and* insttemp = Jz(vtc,vta,ppc) *and* vtc = lookup(rc,rob,rf) *and* vta = lookup(ra,rob,rf)

Decode Load instructions Rule

Decode (rf, rob, **0**) : **Ib(pc, -, r: = Load(ra));bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd

$\Longrightarrow$ Decode (rf, **rob'**, 0) : **bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd

where rob' = enqueue(pc, insttemp) *and* insttemp = r: = load(vta) *and* vta = lookup(ra,rob,rf)

Decode Store instructions Rule

Decode (rf, rob, **0**) : **Ib(pc, -, Store(ra,rv));bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd

$\Longrightarrow$ Decode (rf, **rob'**, 0) : **bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd

where rob' = enqueue(pc, insttemp) *and* insttemp = store(vta,vtv) *and* vta = lookup(ra,rob,rf) *and* vtv = lookup(rv,rob,rf)

2.3 Dispatch

Instructions with state Wait are dispatched to the appropriate functional units and changed to state Exec only when certain conditions are met. First, all the operands must be values, not tags. Second, if any instructions ahead in the ROB are in state Miss (incorrectly predicted jump) then the instructions are not dispatched because they are sure to be discarded. Finally, memory operations cannot be dispatched until they are at the head of the ROB. This strict memory model insures that memory reads and writes will only occur when the instruction is known to be correctly speculated, and in proper order with all other reads and writes. Instructions to the Execute Unit are dispatched through the execute dispatch queue (execd.) Instructions to the Memory Unit and sent through the memory dispatch queue (memd). Likewise, instructions to the BP are sent through the bp dispatch queue (bpd.)

Dispatch Op instructions Rule

Decode (rf, rob1;**Itb(tag, pc, insttemp, Wait)**;rob2, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

if insttemp = r := Op(v1,v2) *and* no_miss(rob1)

$\Longrightarrow$ Decode (rf, rob1;**Itb(tag, pc, insttemp, Exec)**;rob2, cntr) : bsfq , execq, memq, bpq : btbq, rstq, **execd;Idb(tag, insttemp)** , memd, bpd

Dispatch Jz instructions Rule

Decode (rf, rob1;**Itb(tag, pc, insttemp, Wait)**;rob2, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

if insttemp = Jz(vc,va,ppc) *and* no_miss(rob1)

$\Longrightarrow$ Decode (rf, rob1;**Itb(tag, pc, insttemp, Exec)**;rob2, cntr) : bsfq , execq, memq, bpq : btbq, rstq, execd, memd, **bpd;Idb(tag, pc, insttemp)**

Dispatch Load instructions Rule

Decode (rf, **Itb(tag, pc, insttemp, Wait)**;rob, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

if insttemp = r: = Load(a)

$\Longrightarrow$ Decode (rf, **Itb(tag, pc, insttemp, Exec)**;rob, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, **memd;Idb(tag, insttemp)**, bpd

Dispatch Store instructions Rule

Decode (rf, **Itb(tag, pc, insttemp, Wait)**;rob, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

if insttemp = Store(a,v)

$\Longrightarrow$ Decode (rf, **Itb(tag, pc, insttemp, Exec)**;rob, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, **memd;Idb(tag, insttemp)**, bpd

2.4 Complete

Instructions are received from the functional units in three queues. These queues are the execute queue (execq), memory unit queue (memq) and bp queue (bpq.) The results returned in these queues are used to update the ROB. If the instruction returned a value (i.e. was Load or Op) that value is given to the ROB, which updates all occurrences of the destination tag following it in the ROB. These instructions and StoreDone acknowledgments are marked as Done in the ROB.

Complete an Op Instruction Rule

Decode (rf, rob, cntr) : bsfq, **Idb(tag, insttemp);execq**, memq, bpq : btbq, rstq, execd, memd, bpd

$\Longrightarrow$ Decode (rf, **rob'**, cntr) : bsfq, **execq**, memq, bpq : btbq, rstq, execd, memd, bpd *where* rob' = updaterob(rob, tag, insttemp)

Complete an Load/Store Instruction Rule

Decode (rf, rob, cntr) : bsfq, execq, **Idb(tag, insttemp);memq**, bpq : btbq, rstq, execd, memd, bpd

$\Longrightarrow$ Decode (rf, **rob'**, cntr) : bsfq, execq, **memq**, bpq : btbq, rstq, execd, memd, bpd

where rob' = updaterob(rob, tag, insttemp)

Results returning from the BP fall into one of three cases. If the jump was correctly predicted, the instruction is marked Done. If the result was incorrectly predicted, the ROB searches for a Missed jump ahead. If there is another misprediction ahead in the ROB, the current misprediction is ignored and marked as Done. If there is not a misprediction, the flow of control must be changed, so a Reset message is sent to the Funit via the rstq, the rstcntr is incremented and the state of the instruction is marked Miss.

Complete a JzCorrect Instruction Rule

Decode (rf, rob, cntr) : bsfq, execq, memq, **Idb(tag, JzCorrect());bpq** : btbq, rstq, execd, memd, bpd

$\Longrightarrow$ Decode (rf, **rob'**, cntr) : bsfq, execq, memq, **bpq** : btbq, rst q, execd, memd, bpd

where rob' = updaterob(rob, tag, insttemp)

Complete a JzInCorrect() Instruction Rule

Decode (rf, rob1;**Itb(tag, pc, -, Exec)**;rob2, cntr) : bsfq, execq, memq, **Idb(tag, JzInCorrect(newPC));bpq** : btbq, rstq, execd, memd, bpd

if not no_miss(rob1)

$\Longrightarrow$ Decode (rf, rob1;**Itb(tag, pc, insttemp, Done)**;rob2, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

Complete a JzInCorrect() Instruction Rule

Decode (rf, rob1;**Itb(tag, pc, -, Exec)**;rob2, cntr) : bsfq, execq, memq, **Idb(tag, JzInCorrect(newPC));bpq** : btbq, rstq, execd, memd, bpd

if no_miss(rob1)

$\Longrightarrow$ Decode (rf, rob1;**Itb(tag, pc, insttemp, Miss)**;rob2, **cntr + 1**) : bsfq, execq, memq, bpq : btbq, **rstq;(newPC)**, execd, memd, bpd

2.5 Rewind

When an instruction with state Miss is in the ROB, the instructions following it must be removed. This can only occur if no instructions following it are still being executed. If Exec state instructions were removed, the functional units could return with values and attempt to update the ROB with non-existent tags and values. Despite having to wait for all the rules currently being executed to return, this rule can be assured of firing. This is because no new instructions will be decoded (since the reset counter is non-zero) and no Waiting instructions will be dispatched (because there is an instruction in state Wait ahead.) Therefore, the safe_to_kill operation on the ROB used below will always eventually be true.

Rewind Rule

Decode (rf, rob1;**Itb(tag, pc, insttemp, Miss);rob2**, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

if no_miss(rob1) *and* safe_to_kill(rob2)

$\Longrightarrow$ Decode (rf, rob1;**Itb(tag, pc, insttemp, Done)**, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

2.6 Commit

Commitment of an instruction only occurs when it reaches the head of the ROB. This is to ensure that an instruction is never committed when it should not be. If an interrupt or misprediction occurs in front of a Done instruction, it should not be committed. All committed instructions are removed from the ROB and the tag is freed. Committed jumps send update information back through the btbq. Committed assignment to a register produces actual writing to the RF.

Commit to register and remove Rule

Decode (rf, **Itb(tag, pc, v, Done)**;rob, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

$\Longrightarrow$ Decode (**rf'**, **rob'**, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

where rf' = update(rf,r,v) *and* rob = dequeue(rob)

Commit a store completion Rule

Decode (rf, **Itb(tag, pc, Store(-,-), Done)**;rob, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

$\Longrightarrow$ Decode (rf, **rob'**, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

where rob = dequeue(rob)

Commit a Jz complete, and update btb Rule

Decode (rf, **Itb(tag, pc, JzCorrect(), Done)**;rob, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

$\Longrightarrow$ Decode (rf, **rob'**, cntr) : bsfq, execq, memq, bpq : **btbq;(pc ,newpc, Correct)**, rstq, execd, memd, bpd

where rob = dequeue(rob)

Commit a Jz complete, and update btb Rule

Decode (rf, **Itb(tag, pc, JzInCorrect(newpc), Done)**;rob, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

$\Longrightarrow$ Decode (rf, **rob'**, cntr) : bsfq, execq, memq, bpq : **btbq;(pc ,newpc, InCorrect)**, rstq, execd, memd, bpd

where rob = dequeue(rob)

Throughout these previous sections, the state of an ROB entry changes many times. Figure 3 shows these transitions.

2.7 Interrupt

P_P must handle precise interrupts correctly. Precise interrupts are defined as happening at a specific point - all the instructions before the one that generated the interrupt or exception have fully completed execution, and none of the ones after it have begun. To ensure precise interrupts, all the instructions ahead of the target instruction in the ROB must complete and all those after it must not complete. For synchronous interrupts, control flow is changed to the interrupt handler address. The Rewind rule then cleans up the invalid instructions after the target.

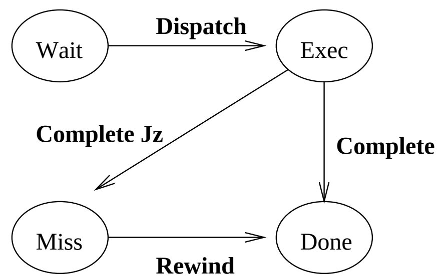

Figure 3: **State transitions for IRB entries in ROB** Decode of an instruction with a value as an argument creates an IRB with state Done. Decode of other instructions creates the IRB with the state Wait. Dispatch rules transition to the Exec state. If the IRB is an incorrect jump, the Completion rules transition the state to Miss. Other IRBs move to Done with Completion rules. The Rewind rule moves from Miss to Done. IRB's with the state Done are retired by the Commit rules.

Interrupt Invalid Rule

Decode (rf, rob1;**Itb(tag, pc, insttemp, state)**;rob2, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

if synchronous_interrupt(insttemp) *and* not no_miss(rob1)

$\Longrightarrow$ Decode (rf, rob1;**Itb(tag, pc, insttemp, Done)**;rob2, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

Interrupt Valid Rule

Decode (rf, rob1;**Itb(tag, pc, insttemp, state)**;rob2, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

if synchronous_interrupt(insttemp) *and* no_miss(rob1)

$\Longrightarrow$ Decode (**rf'**, rob1;**Itb(tag, pc, insttemp, Miss)**;rob2, **cntr + 1**) : bsfq, execq, memq, bpq : btbq, **rstq;(interruptHandlerPC)**, execd, memd, bpd

where rf' = update(rf, excep_pointer, pc)

2.8 Execute

The three execution units execute specific types of instructions and return results. In an ISA with more types of instructions, this model could be simply expanded to include floating point or other specialized units. In P_P, the execute unit applies the operation to the arguments. The Memory unit reads or writes to memory. The BP resolves branches and determines whether or not the speculation made in the Funit was correct.

Exec Rule

Exec : **Itb(tag, r: = Op(v1,v2));execd** : execq

$\Longrightarrow$ Exec : execd : **execq;Itb(tag, v)**

where v = Op(v1,v2)

Mem Load Rule

Mem mem : **Itb(tag, r: = Load(a));memd** : memq

$\Longrightarrow$ Mem mem : memd : **memq;Itb(tag, v)** *where* v = mem[a]

Mem Store Rule

Mem mem : **Itb(tag, Store(a,v));memd** : memq

$\Longrightarrow$ Mem mem : memd : **memq;Itb(tag, StoreDone)**

where mem[a = v]

Bp Taken Correct Rule

Bp : **Itb(tag, -, Jz(0,va,ppc));bpd** : bpq

if va = ppc

$\Longrightarrow$ Bp : bpd : **bpq;Itb(tag, JzCorrect())**

Bp Taken Incorrect Rule

Bp : I**tb(tag, -, Jz(0,va,ppc));bpd** : bpq

if va $\neq$ ppc

$\Longrightarrow$ Bp : bpd : b**pq;Itb(tag, JzInCorrect(va))**

Bp Not Taken Correct Rule

Bp : **Itb(tag, pc, Jz(1,va,ppc));bpd** : bqp

if pc+1 = ppc

$\Longrightarrow$ Bp : bpd : **bpq;Itb(tag, JzCorrect()**)

Bp Not Taken Incorrect Rule

Bp : **Itb(tag, Jz(1,va,ppc));bpd** : bpq

if pc+1 $\neq$ ppc

$\Longrightarrow$ Bp : bpd : **bpq;Itb(tag, JzInCorrect(pc+1))**

2.9 Kill procedure

The jump incorrect completion rules and interrupt rule can be viewed as special cases of the kill function. This function resets the pc to that of a given instruction, and trashes all instructions after it, without changing the behavior of the processor. For the P_S model, the rule would be as follows:

Kill for P_S Rule

(ia, rf, ROB(t, ia', it);rob2, btb, im)

$\Longrightarrow$ (ia', rf, ϵ, btb, im)

This function can be extended to P_P. Provided there are no misses in front of the target instruction in the ROB, the pc can be reset. The instructions following this kill are cleaned up by the rewind rule.

Kill for P_P Rule

Decode (rf, rob1;**Itb(tag, pc, insttemp, state)**;rob2, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

if kill(insttemp) *and* no_miss(rob1)

$\Longrightarrow$ Decode (rf, rob1;**Itb(tag, pc, insttemp, Miss)**;rob2, **cntr + 1**) : bsfq, execq, memq, bpq : btbq, **rstq;(pc)**, execd, memd, bpd

The Jump Incorrect Completion rule uses the kill rule triggered by a JumpIncorrect() received in the bpq and resets the pc to the correctly branch target. The Interrupt rule uses the kill rule triggered by an interrupt and resets the pc to the interrupt handler address.

2.10 ROB operations

Throughout the above rules, many operations on ROB were described or used within the rules. The exact semantics of these operations are provided below.

enqueue(rob, pc, insttemp) adds the IRB(tag, pc, insttemp, Wait) to the end of the ROB. The tag is a fresh tag. This is used by the decode rules for instructions with registers as arguments.

enqueue_done(rob, pc, insttemp) adds the IRB(tag, pc, insttemp, Done) to the end of the ROB. The tag is a fresh tag. This is used by the Loadpc and Loadc instruction decode rules.

lookup(r, rob, rf) Looks up the value of the register. If that register does not have a tag in the ROB, the value from the register file is returned. If a tag exists, but is mapped to a value not yet committed, that value is returned. Otherwise the tag is returned.

no_miss(rob) determines if an irb in rob has the state Miss. If it does, it returns true, else it returns false. This is used by dispatch and complete rules.

updaterob(rob, tag, insttemp) takes the insttemp and tag and updates all occurrences of that tag appropriately. If the insttemp is different from the one already in the ROB entry, the new one replaces the older. (E.g. insttemp is r1 := 5. If the tag is t2, all occurrences of t2 are replaced by 5 and the IRB for that tag and insttemp has its state set to Done.) This is used by the complete rules.

dequeue(rob) removes the element at the head of the rob and deallocates its tag. The element had state Done.

safe_to_kill(rob) searches the ROB for instructions with state Exec. If any exist, it returns false, else it returns true. This is used by the rewind rule.

3 Conclusions and Future Work

The P_P model provides a compact and useful representation of a modern microprocessor. We are currently exploring ways to build on P_P. This work includes extending the model to a more fully-featured instruction set such as DLX. The conversion of a TRS model to a Java program is being explored, either through a compiler or transforms of existing code. This Java implementation could also be a starting point for a simulator for TRS models. The possibility of directly compiling Java (and this program) into hardware is also being researched by Martin Rinard.

4 Credits

The Java program described below was designed and implemented by Lisa A. Poyneer.

A The Java Program

A.1 Discussion

This Java implementation of P_P chose to group rules together into several different threads. Analysis of the TRS revealed a simple grouping of the rules by the 'signature' they took as the initial state. Since multiple rules with the same signature (e.g. decode rules) could not fire at once, one thread can choose between them with a case statement. (Note that the Interrupt and Kill rules were not implemented.) The grouping implemented corresponds to creating an interference graph of the rules and picking out connected components for each thread. There were other options for simulating P_P. One possibility was to create a thread for every rule. While this would simplify translation of rules to code, the extra scheduling overhead and competition for queue elements seemed prohibitive. Another option was to have a single thread which checked for firing conditions. This had the disadvantage of starvation of the rules checked last and did not exploit the independence that many rules had.

The Java design is made of five main modules, nine instances of the Queue class and thirteen separate threads of execution. The five main modules correspond to the five state groups in the TRS. For reference, the main features of each module are described below. The Java implementation of P_P was in many places a nearly straight translation of rules into code. The program prints out rules as they are fired, allowing for observation of the behavior of the TRS. It can optionally print out more information, including but not limited to traces of lookups in the BTB and updating of values.

In practice, this program has shown some limitations. As with any multi-threaded program, the actual execution varies from instance to instance. Testing has shown that the key problem in the implementation is that sometimes the update rule to the BTB are not triggered (since the thread is not executing) and the processor will incorrectly speculate past a large number of branches. The currently unbounded ROB seems to sometimes favor the decode thread, with the result being that other threads can not catch up and empty out the instruction. Implementation of a bounded ROB would be an improvement. This, however, brings up the issue of simulating TRS models that use infinite queue or lists (as P_P does) with finite data structures, which will not be discussed here.

A.1.1 Fetch Unit

The FetchUnit (`FetchUnit`) implements the 3 fetch rules in Section 2.1. It contains the BTB and the instruction memory. It communicates via the bsfq, btbq and rst queues. The FetchUnit has two threads - a main one in the unit and an instance of `ThreadF2` which updates the BTB.

BTB (`BTB`) The BTB stores branch predictions and can be updated. This particular implementation does a simple 1 bit prediction - the predicted pc is whatever the last branch target actually was.

Instruction Memory (`InstMem`) The InstMem is an array of `Instruction`, which can be initialized with a given program.

A.1.2 Decode Unit

The DecodeUnit implements most of the rules in the TRS. It contains the Re-Order Buffer (ROB), the register file and a counter for restarts. Seven threads are used to the five major groups of rules (see Sections 2.2 to 2.6.) The `ThreadDec` implements the decode rules; `ThreadDisp` implements the dispatch rules; `ThreadComp1, 2,` and `3` implement the completion rules; `ThreadComm` implements the committal rules and `ThreadRew` implements the rewind rule.

ROB (`ROB`) The ROB is the most complex class in this program. It allocates tags to registers, keeps track of looking up values, can fetch instructions from the middle or head of itself, can check for the presence of certain entries, and deallocates and removes entries at the head. Currently ROB is implemented as a `Vector`

Register File (`RegFile`) The RegFile is a small array of `Value`.

A.1.3 Other classes

The three other units are for the actual execution of the instructions.

Execute Unit (`ExecUnit`) The Execute Unit does integer operations (currently only addition) on values.

Data Memory (`DataMem`) The Data Memory does loads and store. It is an array of `Value`.

Branch Unit (`BPUnit`) The Branch Unit determine the actual target of the jump and whether or not the prediction in the fetch stage was correct.

A.1.4 Communication

For communication between these five units, queues and container classes of elements were created.

Queue (`Queue`) This queue is an atomic put and get FIFO data structure of limited size. It is implemented as a circular buffer.

Queue Elements There are many classes that are used to hold information and are passed on the queues. These containers include `InstB`, `InstBPB`, `InstBTB`, `InstDB` and `InstTB`. The items these containers hold include several subclasses of `Instruction` and `InstTemp`

References

[1] Xiaowei Shen and Arvind, **Modeling and Verification of ISA Implementations**, *Proceedings of the Australasian Computer Architecture Conference* Perth, Australia, February 1998.

[2] Xiaowei Shen and Arvind, **Design and Verification of Speculative Processors**, *Proceedings of the Workshop on Formal Techniques for Hardware and Hardware-like Systems*, Marstrand, Sweden, June 1998.