MIT Reversible Computing Project

Memo #M7

Modifications to PISA architecture to support guaranteed reversibility and other features

WORKING DRAFT MEMO

Revision: 1.3

Michael P. Frank
http://www.ai.mit.edu/~mpf

MIT AI Lab
545 Technology Sq.
Cambridge, MA 02139

Started February 28, 1997.
Revision Date: 1997/06/30 20:52:38 GMT
Typeset July 23, 1997.

A current version is available online at
http://www.ai.mit.edu/~mpf/rc/memos/M07/M07_revarch.html

Abstract

This document specifies a number of proposed modifications to the PISA (Pendulum Instruction Set Architecture) assembly-level instruction set and instruction semantics, the original version of which was specified in `http://www.ai.mit.edu/~cvieri/pisa.ps`, version of May 5. The advantages of these new modifications are:

- Machine operation may be guaranteed reversible independently of the correctness of user source programs, while still allowing programs to compile to efficient code.
- The number of different instruction opcodes is reduced.
- Subroutine calls require fewer instructions in the caller.
- The direction of program execution can be controlled in software, allowing a single copy of a subroutine to be called either forwards or in reverse.
- Instruction and data memories are merged, making program loaders and other OS features newly possible.
- I/O instructions are provided.
- A special instruction for hardware garbage disposal is provided.

We do not modify the overall instruction set encoding in terms of the number, size, or meaning of the various bit-fields. However, some opcodes will need to be reassigned since some instructions have been added, and others removed. This document does not specify the new opcodes.

1 Introduction

See the abstract for a summary. This memo is organized into sections describing the different suggested changes.

2 Illegal Instructions = NO-OP

The current PISA spec implies that some instructions are illegal. For example, if the destination register and first source register of an ADD are not the same, or if the destination and first source of an AND are the same, then the instruction is illegal. In the case of the AND, executing the instruction would not be reversible, and in the case of the ADD, although adding both of the two source registers into the distinct destination register is logically possible to do reversibly, it might require more complexity in the datapath than is supported.

Also, if some opcodes are unused, they would be considered illegal instructions as well.

However, rather than declaring all these instructions illegal, one could specify that they are perfectly legal but behave as NO-OPs. Such a provision, if followed consistently for all cases where the result of executing the instruction would otherwise be undefined or irreversible, would permit a guarantee of reversibility even if the machine attempts to execute arbitrary words of data. This seems a good safety feature, especially in light of the merged instruction/data memory change discussed elsewhere, which permits programs running on the Pendulum to actually generate new machine code to execute (e.g., a compiler).

If the instruction and data memories were kept separate, then the issue would be somewhat moot because the guarantee of reversibility could be provided at the compiler level rather than at the hardware level, by designing the compiler to never emit an illegal instruction. But since this separation precludes having a real operating system, we suggest the NO-OP behavior instead.

Aside from this issue of how to handle mismatched source/destination registers, all of the non-expanding arithmetic/logical instructions ADD, ADDI, NEG, RL, RLV, RR, RRV, XOR, and XORI are left untouched from the original PISA spec in this proposal.

3 Expanding instructions do XOR

For all the expanding arithmetic/logical operations AND, ANDI, NOR, OR, ORI, SLL, SLLV, SLT, SLTI, SRA, SRAV, SRL, SRLV, the current spec specifies that the destination register must be zero before the operation.

Unfortunately, it is impossible (undecidable) to determine at compile-time whether a given register is actually clear. The user may have arranged through a complicated series of manipulations for the value in a variable to always end up zero, yet there is know way for the compiler to know this.

The compiler *could* have the policy of always assuming the register is *not* clear, and pushing its old value onto a garbage stack, or alternatively, the compiler could insert a run-time check of whether the value is clear, and halt the machine if not. But either of these approaches incurs unnecessary inefficiencies in both amount of code used and run time. The garbage stack approach will, even worse, in general asymptotically increase the space requirements of a program beyond what could be achieved otherwise.

The action of halting the machine if the register is not clear could be handled at the hardware level; this could be a reasonable alternative to performing the AND dissipatively, depending on the context. (For debugging, the halt would be better because it would tell you that a precondition had been violated.)

However, we propose a different solution: have all the expanding instructions XOR their result into the destination register, regardless of the destination register's previous value. This is 100% reversible regardless of the data present in any registers.

Moreover, this provision entirely eliminates the need for all the "Undo" instructions UAND, UANDI, UNOR, UOR, UORI, USLL, USLLV, USLT, USLTI, USRA, USRAV, USRL, USRLV, since the XOR version of the instructions will undo themselves if repeated.

We propose that the expanding instruction names be changed by appending "X" to make their new XOR semantics explicit: ANDX, ANDIX, NORX, ORX, ORIX, SLLX, SLLVX, SLTX, SLTIX, SRAX, SRAVX, SRLX, SRLVX.

4 Eliminate BAL and BAL_r.

The subroutine-calling mechanism in the old PISA was ugly: it required both BAL and BAL_r instructions in the caller, with a no-op between at the address to be returned to. Actually, even this didn't really work because the sequence of instructions executed when executing a call in the forwards direction wasn't exactly the reverse of the sequence executed in the reverse direction: going forwards, you execute BAL, then JR_r; going backwards, you do JR_r, then the NO-OP, then BAL. There would likely have been some irreversible operations happening in here, or a lot of hardware complexity to avoid it.

In the new scheme, subroutine calling is performed via a *single* branch instruction in the caller, which is exactly the same instruction as that used for ordinary (non-subroutine-call) branches. All the intelligence in in the callee (which makes sense since a subroutine is in general called from many places, so more instructions are saved by having as many instructions as possible be in the caller).

We will describe the new subroutine calling mechanism in more detail later. For now, suffice it to say that it allows us to eliminate both BAL and BAL_r from the language.

5 New mechanism for all branches.

The old branch instructions were unidirectional: they were no-ops when executed in one direction, and only provided the possibility of branching when executed in the other, appropriate direction. Branches always came in pairs, with a forward branch pointing to a matching reverse branch, and vice-versa. The motivation here is that we do not want a pair of branches to branch back and forth to each other forever, so when we hit a branch instruction we need to have some way of knowing whether it is a branch we are supposed to actually take given our execution direction. This was the reason for the two opposite types of branches.

Moreover, the behavior of a branch is undefined if the instruction being branched to is not a branch back to the first instruction, or if it is a conditional branch that doesn't actually succeed.

There are two problems with this scheme. First, there is the obvious one that arbitrary code is not guaranteed to have paired branches and thus not guaranteed reversible. If the only entity that can submit new code into the processor for execution is a guaranteed-correct compiler that always provides matching pairs of branches, this is not a problem; but if we want to allow possibly-imperfect native user programs to generate code as well, it is a problem.

However, even aside from the issue of providing a branch at the destination, there is the deeper problem: What if the code in between a pair of branches changes the success/failure of the branch condition? There is no way to tell at compile time whether the user body in the body if an IF statement, for example, will leave the condition's truth value alone, unless side effects in the body code are forbidden, which reduces us to pure functional programming. This implies a sacrifice in asymptotic speed which I would rather not be forced to make.

Again, a run-time check either in software or hardware could allow us to halt instead of dissipating in cases where the branch condition is changed in between branches, but this seems non-ideal.

Our proposal is a new method for doing branches in which which each branch instruction is always atomically reversible in and of itself, regardless of the presence or absence or success or failure of a corresponding branch at the destination. In cases where the branches do not match properly, the behavior will be bizarre, but reversible and well-defined.

The mechanism involves a new entity called the *branch register*, or BR. It controls how the PC is updated between instructions. If it is 0, the PC is incremented by +1 or -1 depending on the current processor direction. If it is nonzero, the PC is incremented by BR. When the processor direction is reversed between instructions by an external signal, BR is two's-complement negated at the same time that the direction bit is flipped. We will arrange things so that the sign of BR is invisible to the programmer; this means that a program has no way of following a different path when running in reverse than the path that is the reverse of the path that it follows when running forwards.

A branch instruction (executed in either direction) works by adding the offset (if the test succeeds) to the destination into the branch register. The matching branch at the destination is just the same, but with a negative offset. Thus when the second branch is executed, the offset in the branch register is canceled out to zero and normal sequential execution resumes. When running in reverse, the lexically-later branch runs first, setting BR to the negated offset to the lexically-earlier branch, and then that earlier branch cancels this out with its offset.

Note that if the first of a pair of branches succeeds but there is no second branch at the destination or it fails, the effect will be that the PC will keep hopping through the code in increments of the offset. This does not seem like useful behavior, but it is well-defined and reversible.

Similarly, if the body of an IF changes the IF-condition to be false, then the branch at the end of the IF will succeed instead of failing, and control will loop back to the start of the body, and looping will continue until the condition becomes false. So a single branch pair really serves as both a loop and a conditional, the exact behavior determined by when and whether the tests succeed or fail.

This scheme has the advantage of being simple to implement, and reversible all the time. Moreover, since the same instructions are used for branching when running in both forwards and reverse directions, we eliminate all the reverse branch instructions BEQ_r, BGEZ_r, BGTZ_r, BLEZ_r, BLTZ_r, and BNE_r.

6 Merge Instruction and Data Memories

Currently the EXCH instruction is defined to access a special data memory that is separate from the instruction memory where programs reside. This makes it more or less impossible to write a full operating system including programs that compile, load, and execute other programs. It also makes it difficult to write programs that include data inline; there is no way for a program to even read instruction memory.

The solution is to merge the two memories. This does nothing strange unless an exchange instruction attempts to perform an exchange with its own address. What should happen? If the instruction were to actually change while being executed this would in general be irreversible.

A natural alternative behavior occurs if there is a special instruction register and the execution cycle works as follows:

```
1. Swap MEM[PC] <-> IR.
2. Execute instruction in IR. (May be an EXCH.)
3. Swap MEM[PC] <-> IR.
4. Update PC.
```

What will happen if step 2 does an EXCH with a memory location that happens to be MEM[PC]? The value it will read is the value that was in IR before step 1. The value it writes will end up in IR after step 3. The contents of MEM[PC] will be the same (namely the EXCH instruction) after step 3 as they were before step 1.

In other words, any instruction that tries to read/write itself instead effectively reads/writes a special variable (which we might call "Itself") which resides in IR between instructions, and in MEM[PC] during an instruction. Initially this variable will contain 0, and it will only change if some instruction actually tries to overwrite itself.

A simple loop of code that tries to make a copy of itself elsewhere in memory will instead produce a copy with a hole, but an extra EXCH instruction after the loop can fill in the hole.

This seems like behavior we can cope with, and it should be easy to implement, and it eliminates the separate memories, making true OS facilities possible.

7 Eliminate START and FINISH

START and FINISH are inherently irreversible. After executing FINISH there is no way to know whether the previously-executed instruction was same FINISH instruction, or the previous instruction. Thus there is an ambiguity as to the predecessor state, and charge recovery cannot be performed perfectly. Ditto for START when running in reverse.

The instructions could be left in with a warning that they cause dissipation when they are first encountered, or they could be declared to be NO-OPs on the real processor, and used only for debugging purposes in the simulator. A real machine will never use them anyway, because operating systems run forever.

8 Replace J with BRA

J (jump) could actually be left in, but to work with the branch register mechanism it should be defined to work as follows:

```
1. Compute the destination address DEST as in PISA spec.
         DEST = PC_{31..26} || target
2. offset = DEST-PC
2. BR += offset
```

This has the same effect as the original J of going to address DEST, but it uses the mechanism of an unconditional branch, one in which the offset is specified in kind of a weird way by a combination of the 26-bit target and the instruction's address.

Aesthetically I would prefer to just use the 26-bit field directly as an offset, and call the instruction BRA; it would work as follows.

```
1. BR += (target_25)^6 || target
```

It's simpler to implement.

Whichever one we keep, J or BRA (or both), the instruction can also be used to make subroutine calls as we will describe. In either case, J_r can be eliminated; the same instruction can be used to either unconditionally begin or unconditionally terminate a branch. If not paired properly it will have the same weird behavior as other branches (hopping off into space, etc.).

9 Replace JR and JR_r by SWAPBR

In the old subroutine calling convention, a BAL in the caller pointed to the start of the subroutine, and a BAL_r to the end. We have already pointed out that this was ugly.

In the new call mechanism, a single BRA (or J if it is kept) points to a single point in the subroutine called the *entry/exit point* or *portal*. At this point is an instruction called SWAPBR which exchanges the contents of the branch register times the direction sign with the contents of another register. That is,

$$\mathtt{BR}_{\text{new}} = dir \cdot reg_{\text{old}} \tag{1}$$

$$reg_{\text{new}} = dir \cdot \mathtt{BR}_{\text{old}} \tag{2}$$

If you branch to a SWAPBR while going forwards, you will end up going forwards from the SWAPBR with the original offset stored in the given register. If you now negate that register (using NEG), branch to just above the SWAPBR, and execute it again (coming into it from above), now the value in the branch register will be the exact right offset to return you to the caller. The BRA (or whatever) in the caller will cancel this offset and normal execution will continue from the caller.

When branching to a SWAPBR while in reverse, note that we will come out above the SWAPBR with the negative of the offset in the given register. Note this exactly mirrors the state of that register just before returning when running forwards. Thus, the behavior of SWAPBR makes the direction of execution invisible to the user, and the subroutine cannot do anything when run in reverse except exactly undo what it did when running forwards.

Since all subroutine entry points will want to negate the given register after doing a SWAPBR on it, we could merge the negation functionality into the SWAPBR instruction and call it SWAPBRN. But this would require that we take care in implementation to do the two steps in the opposite order — first negate, then swap — when SWAPBRN is executed in reverse.

The BRA/SWAPBR combination can easily be used in sort of an inverted way to implement switch statements as well as subroutines, similar to how I described doing this in my earlier memo using BAL/BAL_r/JR/JR_r.

10 New RJ or RBRA instruction.

This would be just like J or BRA (depending on which one is kept in the instruction set) except that it would also flip/negate the CPU direction bit.

The direction change should take effect directly after the instruction is executed, immediately affecting how the PC is to be updated.

This instruction would allow us to branch to a fragment of code to be run in reverse, or to call a subroutine in reverse. This will cut in half the code size of many programs that clean up their own garbage, since the same subroutine that created the garbage can be used in reverse to clean it up. Otherwise, two copies of the subroutine with opposite behavior would generally be needed.

11 I/O instructions

SEE would XOR the next word of data from an input stream into the given register. Running SEE in reverse does the same thing; the direction of flow of the input stream is not necessarily dependent to the direction of PC incrementing. Thus a subroutine that does SEE cannot generally be expected to clean up after itself properly by running it backwards.

SHOW would echo a copy of the given register to an output stream. (Ditto in reverse mode.) We don't want to erase the register, because the point of SHOW is to provide a way to view useful pieces of the state of the machine, not to provide a way to get rid of garbage data and its excess entropy. (Wait till the next section.)

Whether SEE and SHOW are reversible with respect to their effect on the outside world depends on the implementation. Neither need cause internal dissipation in the CPU.

SEE and SHOW will be useful for debugging Pendulum programs under a simulator.

Although the direction of the streams is not necessarily tied to the CPU direction, we may want to be able to reverse both at once in the overall debugging system.

12 Garbage Dispsal instruction

EMIT would empty a register of its contents (leaving it zero). The hardware may do something reversible with the contents (like send them to some large storage it has available), or do something irreversible. The user should be aware that if he does lots of EMIT operations in a sufficiently large 3-D array of Pendlum processors running at constant speed, physics (heat) will inevitably force the EMIT operations to eventually not work any more, and the register will not always be zero after the EMIT.

EMIT removes the data whether run in "forwards" or in "reverse" PC updating mode—the data cannot be reobtained by executing the REMOVE instruction backwards. The user should be aware of this as well.

The point of EMIT is to provide a way to compare partial reversibility with full reversibility. A program running outside the simulator can easily measure the total amount of data emitted, and we can collect statistics on how low the rate of emission can be reduced without impacting speed or space requirements too much.

Of course, programs that use EMIT cannot necessarily reverse themselves, although depending on whether the implementation really keeps the data around or not, we may be able to still reverse the system as a whole.

Note that we don't call the instruction ERASE, because we want to emphasize that physics forbids information erasure; the closest thing we can do with information is to emit it out into some system we don't control and can't reverse (thermal motions).

Note that with the entire architecture guaranteed reversible except for EMIT, we have left open all possibilities yet wrapped up all the irreversibility into a neat package. A pendulum program is reversible iff it never uses EMIT. A pure-reversible version of Pendulum is just a Pendulum in which EMIT isn't provided.

A good implementation of EMIT will attempt to tie up as little energy as possible in the removed data, though of course this can never be less than $k \ln 2$ times the temperature of the system where the data finally ends up.

13 Summary of new ISA

```
Key:
----
r,rsd,rd,rs,ra,rb = 5-bit register identifier.
   No-Op if rd is the same reg as rs or rt.
imm = 16 bit signed immediate
  [imm] imm sign extended to 32 bits
off = 16 bit signed offset
loff = 26 bit signed offset
dir = +1/-1 bit where +1=forward or -1=reverse

Mnem.   Args.           Forwards behavior
------  ---------       ------------------
ADD     rsd,rt          rsd += rt
ADDI    rsd,imm         rsd += [imm]
ANDX    rd,rs,rt        rd ^= rs&rt
ANDIX   rd,rs,imm       rd ^= rs&[imm]
BEQ     ra,rb,off       if ra=rb, BR+=off
BGEZ    rb,off          if rb>=0, BR+=off
BGTZ    rb,off          if rb>0, BR+=off
BLEZ    rb,off          if rb<=0, BR+=off
BLTZ    rb,off          if rb<0, BR+=off
BNE     ra,rb,off       if ra!=rb, BR+=off
BRA     loff            BR+=loff
EMIT    r               0 -> r -> bitsink (irrev.)
EXCH    rd,ra           rd <-> mem[ra]
NORX    rd,rs,rt        rd ^= ~(rs|rt)
NEG     rsd             rsd = -rsd
ORX     rd,rs,rt        rd ^= rs|rt
ORIX    rd,rs,imm       rd ^= rs|[imm]
RBRA    loff            BR-=loff, dir=reverse
```

```
RL      rsd,imm        rsd = rsd rol imm
RLV     rsd,rt         rsd = rsd rol rt
RR      rsd,imm        rsd = rsd ror imm
RRV     rsd,rt         rsd = rsd ror rt
SEE     r              r ^= next word fr. input stream
SHOW    r              echo r to output stream
SLLX    rd,rs,imm      rd ^= rs<<imm
SLLVX   rd,rs,rt       rd ^= rs<<rt
SLTX    rd,rs,rt       rd ^= (rs<rt)?1:0
SLTIX   rd,rs,imm      rd ^= (rs<imm)?1:0
SRAX    rd,rs,imm      rd ^= rs>>imm
SRAVX   rd,rs,rt       rd ^= rs>>rt
SRLX    rd,rs,imm      rd ^= (unsigned)rs>>imm
SRLVX   rd,rs,rt       rd ^= (unsigned)rs>>rt
SUB     rsd,rt         rsd -= rt
SWAPBR  r              r<->BR
XOR     rsd,rt         rsd ^= rt
XORI    rsd,imm        rsd ^= [imm]

PC update between instructions:
  if (BR=0) pc+=dir else pc+=BR*dir
```