An Introduction to Reversible Computing Using phPISA v1.0

Andrew Dickinson
December 2001

http://www.cise.ufl.edu/research/revcomp/phpisa/

Table of Contents

1

Introduction

1.1 What is phPISA?

phPISA is a Pendulum simulator designed to run on the World Wide Web. The name *phPISA* comes from the fact that the simulator itself is written in PHP (http://www.php.net), a loosely−typed scripting language. The source code for phPISA can be found in Appendix B. phPISA implements all of the operations described in Frank's 1999 Thesis (Appendix B). Additionally, there are a few extra "virtual instructions" that have been included for convenience and will be described later.

phPISA is designed to be easy to use and to support a wide range of clients. It is intended as a tool for designing, testing, and understanding the concepts of reversible software and algorithms. Also, because it is available online, ideas can be tested at any time and by any user with a web browser and internet connection. This allows curious passers−by to play with the concepts of reversible computer to gain a better understanding; the goal of this is to spread curiosity, understanding, and respect for the concepts of reversible computing.

1.2 What is Pendulum?

Pendulum is a fully adiabatic (reversible) general purpose processor. In the most basic of terms, it is a computer processor that not can not only processes data (like any CPU should!), but can also "unprocess" the data and return the energy used in the computation to its provider. While the pendulum chip is a general purpose CPU, it is a "proof−of−concept" and provides a very minimal set of instructions. The chip was originally designed at MIT for the study of reversible computing (Vieri).

It should be noted that Pendulum is more than just a simple instruction set architecture (ISA) for reversible computing. It is, on a physical level, fully reversible. While it is possible to implement a reversible architecture using standard "irreversible" logic, this design would not have all the benefits that would be seen by a truly reversible processor, such as Pendulum.

1.3 Why reversibility?

As mentioned above, the basic concept behind a reversible processor is the ability of a reversible processor to "uncompute" data and obtain its original input values (and thus return energy to its source). There are a few applications today where this could be useful; low−power devices such as cellular telephones and PDAs are requiring ever−powerful processors, while consumer demand for longer battery life is also increasing. Reversible computing might be one solution to this problem because of the significantly lower power usage of a reversible processor.

While there are currently only a few uses for a reversible processor, there is a possible long−term demand for this technology. As processors are becoming increasing

smaller and faster, the amount of energy being dissipate is also increasing. At some point in time (about 30 years from now) the amount of energy being dissipated will exceed the energy holding the processor together (i.e. it will severely overheat). This trend is unsurprising considering the pains taken by current manufacturers to keep their CPUs cool.

2

phPISA User's Guide

2.1 Getting Started

To begin with, load the following URL into your web browser: http://www.cise.ufl.edu/research/revcomp/phpisa/phpisa.html. Once the page is loaded, you will see two frames. The frame on the left is the editor frame. The frame on the right is the output frame.

2.2 Entering Program Code

To enter code for a PISA assembly program, simply type the code into the text area in the editor frame. The text area is identified by the title "Program Here:". Additionally, you can use your favorite text editor to write PISA assembly and paste the code into the text area on the phPISA webpage.

2.3 Executing a Program

Once you have entered your program, click the "Run" button at the bottom of the editor frame. Your program will be compiled and any errors in your code will be displayed in the frame on the right. If compilation is successful, phPISA will begin executing your code. Please be patient; it may take a while for your code to finish executing, especially if your program will be executing a lot of instruction calls.

2.4 Simulation Options

When you execute code, there are a few options to help you understand what phPISA is doing. The first option is "Debug", you can enable it by enabling the checkbox labeled "Debug." While running in Debug mode, you will see the instructions as they execute. This is useful to confirm that the code that you wrote is in fact executing as you intended on the simulator.

The other option that is available is "Super Debug". Super Debug will display the contents of the values of the registers of the CPU as well as internal state data for the processor (such as the program counter, the branch register, and the direction of execution). As a warning, enabling Super Debug will seriously impact the speed at which phPISA executes.

2.5 Understanding the Output

There are four categories into which output will fall. This section will help you understand what the output (in the output frame) means. It is important to note that phPISA follows the "no news is good news" philosophy. If you don’t see output for a

while, it is probably busy processing (or it's stuck in an infinite loop).

2.5.1 Error Messages

If there is a problem compiling your code, you will be presented with a message indicating the problem. Typical causes for error messages include: attempting to use instructions not in the ISA (see appendix A), attempting to branch to an undefined label, or attempting to get the address of a label that is undefined.

2.5.2 Program Output

If your program has an `OUTPUT` or `SHOW` statement, this will be displayed (in blue) in the form "REG = VALUE" where REG is the name of the register and VALUE is the value stored in that register (in decimal).

2.5.3 Debug Output

If you have enabled Debug (see above), you will see a message for every instruction that is executed. There are a few things to note about reading the debug output. When you "Run" your code, phPISA compiles it into an internal binary format. Enabling Debug causes this code to be reverse−engineered. So labels and offsets will be converted to their numeric values, all registers will be denoted by R## where ## is the number of the register and some instructions (namely `OUTPUT`) may be converted (for example to `SHOW`). The number in front of the operation is the address of memory at which that instruction is stored (in other words, the value of the program counter when that instruction was executed).

2.5.4 Super−Debug Output

Super Debug will display a table showing the name of the register and its value (in hexadecimal notation). Additionally, it shows the current value of the program counter, the value of the branch register, and the direction that the processor is executing (either forward or reverse). When used in conjunction with the Debug option, you can see the result of an instruction immediately after that instruction executes.

3

Reversible Programming Tutorial

Before we delve into the details of reversible programming, we need to cover the basics of PISA. We'll start off with the basic syntax, a few basic operands. Then we will go into control statements for conditionals and subroutines. We will finish up with looping structures and data values.

3.1 The Basics

3.1.1 Syntax

The basic syntax for an operand is: *`[label:] OP arg0 [arg1 [arg2]]`* where *`OP`* is a PISA operation (see Appendix A) and *`arg0`*, *`arg1`*, and *`arg2`* are either register values, immediate values or labels, depending on the operation. Register values can be denoted by either #, `$#` or `R#` format (where # is the register number). Labels *are not* case sensitive '`foo`' is the *same* as '`FOO`'. Immediate values are decimal values; negative numbers and zero are also supported.

3.1.2. Comments

PISA comments begin with a semicolon, `';'`, and continue to the end of the line. They can be placed at the beginning of the line or anywhere after.

3.2 Introduction to Reversible Instructions

A reversible instruction is one that, given the result and the operation can be "undone". This seems vague, so let's look at some examples. We'll begin with a 3rd grade example.

Example 3.1: 3rd Grade Example

If Jane gives Sally 3 apples, Sally now has 3 more apples than she did before. To reverse the "operation", Jane can take her 3 apples back. While this is an extremely simple (and lame) example, it demonstrates the basic concepts of reversibility.

Now, let's look at some more relaxant examples. We will start off by showing how we can manually reverse data. Later we'll look at how the processor can be reversed to take care of reversing operations for us. For now we are going to stick with simple examples and work our way into the exciting "stuff" shortly. Example 3.2 is basically the Sally and Jane example, from above, expressed in PISA assembly. Notice how PISA assembly is different from most other register−memory architectures in that simple operations like ADD use only two registers (a destination and a source).

Example 3.2: Manual Reversibility

```
ADDI R1, 3          ; R1 <- R1 + 3   we have now added 3 to
                    ; .. the value of R1
ADDI R1, -3         ; R1 -> R1 + 3   basically we have just
                    ; .. undone our calculation.
```

In Example 3.2, we added 3 to the value of R1, and then manually reversed it. The first ADDI instruction adds 3 to register R1 and the second ADDI adds −3 to register R1. Notice the "arrow−notation" in the comments. While it is easy to think about simple adding and subtracting 3 from R1, we like to think of adding and "unadding" to R1.

The last two examples were very simple. The basic idea was that you need to be able to undo the operations that you perform. This raises questions−−how do you handle a bitwise shift? If you shift 1011_2 to the right, the rightmost bit "falls off the end". How do you reverse something like that?

The solution is a clever usage of a logical XOR. Let's continue working with the binary 1011_2; if you logically shift it to the right one bit, you get 0101. It would be nice if we could simply shift to the left and get our number back, but we cannot. The solution is to store the shifted result into a different register; and when storing it, XOR it with the value that is already stored in that location. To "undo" this operation, perform it again, the XOR will clear it. Let's look at an example, assume all the registers are 0:

Example 3.3: A look at an XOR−style operations

```
ADDI R1, 11         ; R1 <- R1 + 1011₂
SRLX R2, R1, 1      ; R2 <- R1 >> 1 (this should be 0101₂)
SRLX R2, R1, 1      ; R2 -> R1 >> 1 (R2 is now 0000)
ADDI R1, -11        ; R1 -> R1 + 1011₂
```

Note that this still works even if R2 already has a value. The "double XOR" undoes itself. Many operations use this tactic, NANDX, ORX, etc. (See Appendix A).

3.3 Control Structures

3.3.1 Labels

Now that we have seen a few simple examples, let's take a look at control structures (conditionals and looping structures). We'll start by looking at labels. A label is just a means of identifying a location in a program; it is used to identify the target of a branch and to address data (or subroutines, but that's another story). A label is a string of one or more alphanumeric characters followed by a colon (':'). phPISA is not case−sensitive with respect to labels.

Example 3.4: A label

```
Foo: ADDI R1 3      ; mark this as label 'Foo'
```

3.3.2 Conditionals and Subroutines

Branching poses interesting problems with reversibility. When executing forward, if the simulator executes a branch statement, it either takes the branch or it continues executing. When executing in reverse (i.e. approaching a if−statement "from the bottom", we need to know whether we entered the body of the statement when we went through it last time (forwards). For example, in this C snippet:

```
if( a == b) {
     c + = 1;
}
```

When executing forwards, you check if ’a’ is equal to ’b’. If so, you add one to ’c’. When executing in reverse you need to "know" that ’a’ is still equal to ’b’, and if so, "unadd" one to ’c’. In this example, this is easy, the value of ’a’ and ’b’ didn’t change inside the loop, so they can be tested again. Let’s take a look at how to do this in PISA assembly.

Example 3.5: An if statement

```
            ADDI R1 3               ; a <- 3
            ADDI R2 5               ; b <- 5
iftop:      BNE R1 R2 ifbot         ; if(a==b) {
            ADDI R3 1               ;    c += 1;
ifbot:      BNE R1 R2 iftop         ; } fi(a==b)
            ADDI R2 -5              ; b -> 5
            ADDI R3 -3              ; a -> 3
```

Now, if you are used to programming in assembly on other architectures, this probably looks quite strange. It seems like the two branch statements would get caught in a loop. The reason for this difference is how Pendulum handles branch statements. When it is executing "normal" non−branch statements, it simply adds to the Program Counter (PC). When it encounters a branch statement, if it decides to take the branch, it *adds* the *offset* to a special "Branch Register". The value of the Branch Register is added to the PC to determine the next memory address to execute.

If you look at example 3.5, you can see that the branch on the third line *would* be taken (BNE is "Branch if Not Equal", see Appendix A). The target of the branch is +2 from the current location. So we add 2 to the Branch Register and the PC is set to this new address (2 lines from the branch we just encountered), which happens to be another branch that is, again, taken. So we add the new offset (−2) to the Branch Register and it goes back to zero. This may be confusing, but it allows us to execute conditionals forwards and reversibly.

The Branch Register is accessible to the programmer, this will allow us to write subroutines. By saving the value of the Branch Register, we can write a subroutine that

will return to the caller. PISA has an instruction, SWAPBR, that exchanges the value of a register with the value in the Branch Register. Let's examine how this can be used.

Example 3.6: The Branch Register

```
100:              BRA foo           ; go to foo
                  ; ...
200:  foo_top:    BRA foo_bot
201:  foo:        SWAPBR R1         ; <-- this is foo
202:              NEG R1
                  ; ...
250:  foo_bot:    BRA foo_top
```

Let's step through and see what happens, for the purpose of this example assume the numbers in the left column are the memory addresses for the instructions to their right, respectively. The BRA at address 100 is jumping to 201 (the offset 101 is added to the Branch Register). At that memory location, the SWAPBR instruction, exchanges the value in the Branch Register (101) with the value in Register 1 (let's assume it's zero). Next we negate the value of Register 1 (you'll see why shortly). With the Branch Register back at zero, the PC is incremented "normally" and the program continues. At line 250, it jumps unconditionally to line 200 and sets the Branch Register to −50. When execution continues at line 200, another unconditional branch adds +50 to the Branch Register (which is now zero, again), and execution continues downward. The next instruction we encounter is the SWAPBR (again). This time we exchange the value in Register 1 (which should be holding −101) with the Branch Register. This causes the PC to be set to 100. At line 100, the unconditional branch adds 101 to the Branch Register, and we continue down code!

It works just as well in reverse; this time we'll go a bit quicker. When we encounter the `BRA foo` in reverse, we jump to `foo` at line 201. We swap the Branch Register (BR) with Register 1, and continue *up* the code. We hit the branch at line 200 which jumps us to the branch at 250 (and thus returning the BR to zero, and continuing up the code). At line 202, we negate the value stored in Register 1 and continue to line 201, where we swap the BR. We're now taken back to the `BRA foo`, which resets the BR to zero. Follow that?? It's really not that complicated, just different.

One quick note before we get into Looping. To have phPISA begin execution at a location other than the top of your code, use the `.start <label>` identifier to tell it which subroutine to start executing first. It defaults to the top of the code otherwise.

3.3.3 Looping

Looping structures are very similar to conditionals. The difference is that when we get to the bottom of the loop, we want to continue back at the top (rather than continuing normal code execution. A reversible for−loop can be created by having a upper and lower limit; the "i" will walk from the lower limit to the upper limit going forwards, and from the upper to the lower limit in reverse. Let's look at an example:

Example 3.7: A for loop

```
            ADDI R1 0           ; LL <- 0
            ADDI R30 10         ; UL <- 10
loop_top:   BNE R1 R2 loop_bot  ; unless (LL != i) do
            OUTPUT R2           ;    ... =)
            ADDI R2 1           ;    i++;
loop_bot:   BNE R30 R2 loop_top ; repeat while( i != UL)
            ADDI R2 -10         ; i -> 10
            ADDI R30 -10        ; UL -> 10
```

If you type this code (or copy and paste it) into phPISA and run it, you will see that it does, in fact, loop from 0 to 9. Observe in the comments that the `LL` (lower limit) is 0 and the `UL` (upper limit) is 10. This code could be expressed in this C code snippet:

```
for(i = 0; i <10; i++) {
     printf("%d", i);
}
```

3.4 Data Values and Memory Addresses

phPISA has support for data values being loaded into memory on program load. This saves time and program space. Here's an example of how to use phPISA's `DATA` tag:

Example 3.8: The DATA tag

```
            ADDI R1 foo       ; load the data address
            EXCH R2 R1        ; grab the data there
            OUTPUT R2         ; show it
            EXCH R2 R1        ; put it back
            ADDI R1 -foo      ; unload the address
            FINISH            ; done
foo:        DATA   -99        ; here's some data
            DATA 172          ; here's more data
```

There are a few things going on here, let's examine them piece−by−piece. The first instruction loads the address of label `foo` into register 1. The second instruction exchanges the value of Register 2 with the memory address stored in register 1. Note that this is an *exchange*, and *not* a load or copy. This is important for reversibility. The next important item to observe is the `ADDI R1 -foo`. This "unadds" the address at label `foo`. Finally, the DATA tag (last two lines) just indicates what data is stored at that label. If there is no label (as in the last line), the data is stored in the location immediately following the previous location that was filled. With this, you can easily create large arrays or other data structures that get pre−loaded with the program.

3.5 Reversing Processor Direction

You've probably been wondering, "how does the processor change direction?" And we have successfully managed to avoid the issue, until now. The PISA instruction RBRA (Reverse and Branch) will branch to a given label and switch the direction of processing. We'll look at a few examples before we put it all together wrap things up.

Example 3.9: RBRA to reverse!

```
            BRA allocate      ; allocate some space
            ; ... do stuff
            RBRA allocate     ; unallocate the space!!!
```

This may seem like a rather terse example for a demonstration of the "big" instruction in this ISA, but it really exemplifies the simplicity of the instruction. In the first line, we jump to some subroutine that allocates a few registers for us (by loading their values onto the stack). Then we can use the registers we need and when we're done, we call the allocate subroutine in reverse, and it will reverse the allocation and return the values to their original states (thus de−allocating them).

There is really nothing "scary" about reversing the direction of the processor. It is still performing operations (like normal) and moving from instruction to the next (like normal, just backwards), and still branching, etc. It allows for easy cleanup of subroutine calls (as seen above) or you could reverse and entire program, etc.

4

Sample PISA Code

4.1 QuickSort()

4.1 QuickSort()

The quicksort algorithm for sorting an array of data, can be implemented on the Pendulum architecture. To begin with, here is a quick refresher on the quicksort algorithm in C:

```
void qsort(int *array, int left, int right) {
     int i, last;
     if(left >= right)
          return;
     swap(array, left, (int)((left+right)/2));
     last = left;
     for(i = left + 1; i <= right; i++)
          if(array[i] < array[left])
               swap(array, ++last, i);
     swap(array, left, last);
     qsort(array, left, last - 1);
     qsort(array,last + 1, right);
}

void swap(int * array, int left, int right) {
     int temp;
     temp = array[left];
     array[left] = array[right];
     array[right] = array[temp];
}
```

Obviously, the code above is not a complete C program, but it shows how the algorithm works. Let's start off by showing how to implement a main method in PISA. We start of by telling the assembler what subroutine to execute first, in this case '`main`'

```
;; pendulum pal file
      .start main
main:
```

Next, initialize a few registers that will be used through the course of the program

```
ADDI $1 1400            ; address of the data
ADDI $2 0               ; first argument (left)
ADDI $3 5               ; second argument (right)
ADDI $4 1000            ; stack for subroutine args
ADDI $19 1200           ; stack for return addresses
ADDI $16 1600           ; stack for temp data
```

This program passes its arguments via a stack, so we need to push the arguments onto the stack at the appropriate location and adjust the stack pointer accordingly.

```
EXCH $2 $4              ; put the arguments...
ADDI $4 1               ; ... on the stack
EXCH $3 $4
ADDI $4 1
```

We are going to call a few subroutines here (which will be described shortly) to help

setup and display our data.

```
            BRA  load               ; load the values
            BRA  disp               ; display them (first 6)
            BRA  qsort              ; sort them
            BRA  disp               ; display them (second 6)
```

At this point, we have loaded the values, displayed the unsorted array, sorted the data using the qsort subroutine and displayed the sorted values. Now we reverse the operations, to demonstrate that these routines can, in fact, be performed in reverse. The result of this will be that the data will be 'unsorted' into its initial state. Also, the values can be unloaded from memory by calling the '`load`' subroutine in reverse.

```
            RBRA qsort              ; un-sort them
            BRA disp                ; display them (third 6)
            RBRA load               ; unload the values
            finish                  ; done
```

Next, we will examine the details of the swap operation. First, examine the way that a subroutine is defined. Because of the need to be able to reverse the operations, subroutines are wrapped in branch statements. When a branch statement is executed, the Pendulum *adds* the offset value of the destination minus the source in a special branch register. You'll see how this is useful in a bit.

Observe the SWAPBR instruction. It swaps the value in the branch register with a value that is stored in a register. Usually the value in the register should be 0 (before the swap). Notice how the branch register is swapped into register 21 (which is initially 0) and then negated; you'll see why this is useful later.

```
swaptop:    BRA swapbot         ; swap ($30, $31) {
swap:       SWAPBR $21
            NEG $21
```

Now the swap instruction begins its real work. First, to save time and to avoid a hazard with the way this swap works, we're going to skip over the body of the code if we are swapping the same memory location. Notice that the first `BEQ` statement has a matching `BEQ` at the target of the branch `ifbot_2.`

Inside the if statement, we simply calculate the memory addresses, obtain the values from memory, perform the swap, and return them to memory. Notice how the swap executes. That simple patter of XOR operations is all that is necessary to swap values; no temporary variables are needed. After returning the values to memory, the memory addresses are uncomputed.

```
iftop_2:    BEQ $30 $31 ifbot_2
            ADD $30 $1          ; grab them from memory
            ADD $31 $1
            EXCH $28 $30
            EXCH $29 $31
                                ;swap the values
            XOR $28 $29
            XOR $29 $28
```

```
            XOR $28 $29

            EXCH $28 $30      ;put them back into memory
            EXCH $29 $31
            SUB $30 $1
            SUB $31 $1
ifbot_2:    BEQ $30 $31 iftop_2
swapbot:    BRA swaptop       ; }
```

In the above code, when execution reaches the final BRA statement, it branches it's matching branch statement, near the top. Remember that the branch resister adds the offset from the source to the destination to the branch register. Because the offsets are the opposite of each other, the result is that after the second branch statement, the branch register holds zero (which causes the Pendulum to continue executing in normal fashion).

After execution branches to the top, the SWAPBR instruction is encountered again, and the value that was previously stored is loaded into the branch register and the Pendulum returns to the location that it was called from. Again, the branch offset is added and execution continues from that location.

The load and disp subroutines are both very simple. They each perform a similar tasks; stepping through the array performing an operation. The code for each follows:

```
Loadtop:    BRA loadbot             ; load () {
load:       SWAPBR $21
            NEG $21

            ADDI $5 56
            EXCH $5 $1
            ADDI $1 1
            ADDI $5 23
            EXCH $5 $1
            ADDI $1 1
            ADDI $5 45
            EXCH $5 $1
            ADDI $1 1
            ADDI $5 67
            EXCH $5 $1
            ADDI $1 1
            ADDI $5 12
            EXCH $5 $1
            ADDI $1 1
            ADDI $5 35
            EXCH $5 $1
            ADDI $1 -5
loadbot:    BRA loadtop             ; }

disptop:    BRA dispbot             ; disp () {
disp:       SWAPBR $21
            NEG $21

            EXCH $17 $1
            OUTPUT $17
            EXCH $17 $1
            ADDI $1 1
            EXCH $17 $1
            OUTPUT $17
            EXCH $17 $1
```

```
            ADDI $1 1
            EXCH $17 $1
            OUTPUT $17
            EXCH $17 $1
            ADDI $1 1
            EXCH $17 $1
            OUTPUT $17
            EXCH $17 $1
            ADDI $1 1
            EXCH $17 $1
            OUTPUT $17
            EXCH $17 $1
            ADDI $1 1
            EXCH $17 $1
            OUTPUT $17
            EXCH $17 $1
            ADDI $1 -5

dispbot:    BRA disptop             ; }
```

Here we will work through the reversible, recursive code for the QuickSort algorithm. The first thing that we do is push the value from the branch register onto a stack. We need to do this because this method is recursive and needs to be able to keep track of the the location to which we are returning.

```
subtop:     BRA subbot
qsort:      SWAPBR $20              ; swap the BR into $20
            NEG $20                 ; make it negative
            EXCH $20 $19            ; return address -> stack
            ADDI $19 1
```

Next, we pop the arguments of their stack.

```
            ADDI $4 -1
            EXCH $3 $4              ; fetch it from the stack
            ADDI $4 -1
            EXCH $2 $4
```

Inside the qsort routine, we first handle the base−case for the recursion. The base case in C (from above) is:

```
      if(left >= right)
            return;
```

Here is one way to convert this to PISA. Notice that the values that that are computed into register 5 are uncomputed after we are done using them. This ensures that we are not accidentally making invalid assumptions about the values of the registers later.

```
            ADD $5 $2
            SUB $5 $3               ; temp1 <- (left-right)
iftop_0:    BLTZ $5 ifbot_0         ; if(left >= right) {
            ADD $5 $3               ;    temp1 -> (left-right)
            SUB $5 $2
            XORI $18 1
returntop:  BNE $18 $0 returnbot    ;    and return!
ifbot_0:    BLTZ $5 iftop_0         ; }
            ADD $5 $3               ; temp1 -> (left-right)
            SUB $5 $2
```

The next line of C code is:

```
     swap(array, left, (int)((left+right)/2));
```
To convert this to assembly, we first need to calculate the values for the arguments. The first argument is easy, it is just the value of left from before. The second argument can be computed by adding left and right into a register and then shifting them right by one into another register. Again, the argument values (in registers 30 and 31) are uncomputed. Notice that the Shift Right Logical (SRLX) is XORed with the value that is in the destination register. The neat thing about this is that you can uncompute the result of a SRLX operation by performing it again with the same arguments (and values).

```
            ; do the swap
            ADD $30 $2              ; argL <- left;
            ADD $27 $2
            ADD $27 $3              ; temp <- left+right;
            SRLX $31 $27 1          ; argR <- (left+right)/2;
            BRA swap                ; swap(argL, argR);
            SRLX $31 $27 1          ; argR -> (left+right)/2;
            SUB $27 $3
            SUB $27 $2              ; temp -> left+right;
            SUB $30 $2              ; argL -> left
```

Now we setup for the C loop:

```
     for(i = left + 1; i <= right; i++)
          if(array[i] < array[left])
               swap(array, ++last, i);
```

First, we need to initialize our values for i (our counter), and our limit. Additionally we need a "reverse limit", which can basically be thought of as register that holds the same value as the initial value of i.

```
            ADD $6 $2               ; last <- left;
            ADD $7 $2               ;
            ADDI $7 1               ; reverse_limit <- left + 1
            ADD $5 $7               ; i <- left + 1;
            ADD $8 $3               ;
            ADDI $8 1               ; limit <- right +1;
```

Inside the loop, we calculate the memory addresses of the array that we need and load those values from memory.

```
looptop:    BNE $5 $7 loopbot       ; unless (i != left+1) do
            ADD $9 $1
            ADD $9 $5               ; $9 <- &array[i];
            ADD $10 $1
            ADD $10 $2              ; $10 <- &array[left];
            ; get the values
            EXCH $11 $9             ; $11 <-> array[i]
            EXCH $12 $10            ; $12 <-> array[left]
```

Next, we perform our conditional statement

```
            ADD $13 $12
            SUB $13 $11             ; $13 <- ($12 - $11)
iftop_1:    BLEZ $13 ifbot_1        ; if(array[i] < array[left]) {
```

Before we can call the swap operation, we need to exchange the values back into memory and then setup the arguments to the swap operation. After calling the swap operation, we uncompute the arguments and exchange the values back into the registers.

```
            EXCH $11 $9             ; $11 <-> *($9)
            EXCH $12 $10            ; $12 <-> *($10)
            ADDI $6 1               ; last <- (last++)
            ADD $30 $6              ; argL <- last;
            ADD $31 $5              ; argR <- i;
            BRA swap                ; swap(argL, argR);
            SUB $30 $6              ; argL -> last
            SUB $31 $5              ; argR -> i
            EXCH $11 $9             ; $11 <-> *($9)
            EXCH $12 $10            ; $12 <-> *($10)
ifbot_1:    BLEZ $13 iftop_1        ; }
```

We have to save the value of register 13 (which holds the value of array[left] − array[i]). It is necessary when we execute in reverse to determine if we should enter the if statement or not. Unfortunately, because 11 and 12 have changed, it is difficult to uncompute, but it is easy to store!

```
            EXCH $13 $16            ; save it for later
            ADDI $16 1
```

Now we return the values to memory and uncompute their addresses. Also, since this is the end of the loop, increment i. After the loop, uncompute the values that we setup before the loop.

```
            EXCH $11 $9             ; $11 <-> array[i]
            EXCH $12 $10            ; $12 <-> array[left]
            SUB $10 $2              ; $10 -> &array[left]
            SUB $10 $1
            SUB $9 $5               ; $9 -> &array[i]
            SUB $9 $1
            ADDI $5 1               ; i <- ($i++);
loopbot:    BNE $5 $8 looptop       ; repeat while(i != right+1)
            ADDI $8 -1              ;
            SUB $8 $3               ; limit -> (right + 1)
            SUB $5 $7               ; i -> left + 1
            SUB $5 $3               ; i -> right
            ADDI $7 -1
            SUB $7 $2               ; temp1 -> left + 1
```

Finally, we perform the following operations in C:

```
swap(array, left, last);
qsort(array, left, last - 1);
qsort(array,last + 1, right);
```

The swap operation is simple, compute the arguments, call the subroutine, and uncompute the arguments.

```
            ADD $30 $2              ; argL <- left
            ADD $31 $6              ; argR <- last
            BRA swap                ; swap(argL, argR);
            SUB $31 $6              ; argR -> last
            SUB $30 $2              ; argL -> left
```

We need to put the arguments to the two qsort operations on the stack. We need to load the values for the second qsort first (LIFO).

```
            ADD $14 $6              ; left = last + 1;
            ADDI $14, 1
            EXCH $14 $4
            ADDI $4 1
```

```
            EXCH $3 $4              ; right = right;
            ADDI $4 1

            EXCH $2 $4              ; left = left;
            ADDI $4 1

            ADDI $6 -1              ; right = last - 1;
            EXCH $6 $4
            ADDI $4 1
```

Finally, we call the qsort calls recursively and do some final cleanup.

```
            BRA qsort               ; qsort(left, last-1);
            BRA qsort               ; qsort(last + 1, right);

returnbot:  BNE $18 $0 returntop
            EXCH $18 $16
            ADDI $16 1
            ADDI $19 -1
            EXCH $20 $19
subbot:     BRA subtop
```

5

Future Work

5.1 phPISA Modifications
5.2 Tutorial
5.3 Code Samples

5.1 phPISA Modifications

There are a number of changes that could be made to phPISA. Firstly. It would be nice if the interface was more "IDE−like". If it had the ability to add break points, step a certain number of instructions, change memory and register values, etc. The main remans for not doing that at this stage is cross−browser compatibility. phPISA is a learning tool, and as such should be readily available to anybody with a knowledge−thirsty mind.

Currently, phPISA compiles to an internally used binary format. A nice feature would be the ability to save that binary for later use/analysis and the ability to link binaries together to create libraries, etc. Also, it would be nice if binary compatibility existed between this project and some of the other PISA projects.

5.2 Tutorial

The tutorial could be longer and have more examples. It could go into greater depth on a number of the topics. It could easily be expanded into a small book, or a chapter or two in a larger manuscript. The subject of reversible looping structures and methods/uses for garbage handling (which was not even discussed in this paper) could be covered significantly better, but would outside the scope of this simple tutorial.

5.3 Code Samples

A larger library of code samples would be great for people to study and learn from. Varying degrees of difficulty could help people interested in reversible computing understand the basic concepts without being scared away from the area. A library of common algorithms (sorting, graphs, etc.) would be a great reference for programmers that are already familiar with these concepts. It would also be a great way to show what does and what does not work well reversibly, as some algorithms can be made reversible more easily than others.

If Pendulum supported better input/output, a simple reversible game would be a really great way to get people excited about the concepts.

A

List of PISA Instructions

A.1 Table of PISA operations

Appendix A: List of PISA Instructions

Note: standard C notation is used to define what the operation does

Instruction	***Operation performed***
ADD regd regs	regd += regs
ADDI regd imm	regd += imm
ANDX regd reg1 reg2	regd = regd ^ (reg1 & reg2)
ANDIX regd regs imm	regd = regd ^ (regs & imm)
NORX regd reg1 reg2	regd = reg ^ ~(reg1 | reg2)
NEG regsd	regsd = ~regsd
ORX regd regs reg2	Regd = regd ^ (regs | reg2)
ORIX regd regs imm	regd = regd ^ (regs | imm)
RL regsd imm	regsd = regsd rotated left by imm bits
RLV regsd regt	regsd = regsd rotated left by regt bits
RR regsd imm	regsd = regsd rotated right by regt bits
RRV regsd regt	regsd = regsd rotated ritght by regt bits
SLLX regd reg1 imm	regd = regd ^ (reg1 << imm)
SLLVX regd reg1 reg2	regd = regd ^ (reg1 << reg2)
SRAX regd reg1 imm	regd = regd ^ (arithmetic shift of reg1 by imm)
SRAVX regd reg1 reg2	regd = regd ^ (arithmetic shift of regs by regt amount)
SRLX regd reg1 imm	regd = regd ^ (reg1 >> imm)
SRLVX regd reg1 reg2	regd = regd ^ (rreg1 >> reg2)
SUB regd regs	regd −= regs
XOR regd regs	regd = regd ^ regs
XORI regd imm	regd = regd ^ imm
BEQ reg1 reg2 label	Branch to label if reg1 == reg2
BGEZ reg	Branch to label if reg >= 0
BGTZ reg	Branch to label if reg > 0
BLEZ reg	Branch to label if reg <= 0
BLTZ reg	Branch to label if reg < 0
BNE reg1 reg2 label	Branch to label if reg1 != reg2
BRA label	Branch to label

Instruction	***Operation performed***
EXCH reg1 reg2	Swap the value in reg2 with the data in address reg1
SWAPBR reg	Swap the BR with reg
RBRA label	Reverse and Branch to Label
SHOW reg	Displays the value of reg
OUTPUT reg	Displays the value of reg
FINISH	Halts the simulator
DUMP	Virtual routine to display the register contents

B

phPISA Source Code

B.1 asm_arith_log.php

```
<?
/* asm_arith_log.php */
function r_ADD( $regd, $regs) {
      debug_msg("ADD($regd, $regs)");

      $src = get_reg($regs);
      $dest = get_reg($regd);

      $dest = $src + $dest;

      set_reg($regd, $dest);
      return true;
}
$reverse_op["r_ADD"] = "SUB arg0 arg1";

function r_ADDI($regd, $imm) {
      debug_msg("ADDI($regd, $imm)");

      $dest = get_reg($regd);
      $dest += $imm;
      set_reg($regd, $dest);
      return true;
}
$reverse_op["r_ADDI"] = "NEG arg0;ADDI arg0 arg1;NEG
arg0";

function r_ANDX($regd, $reg1, $reg2) {
      debug_msg("ANDX($regd, $reg1, $reg2)");

      $dest = get_reg($regd);
      $src1 = get_reg($reg1);
      $src2 = get_reg($reg2);

      $temp = $src1 & $src2;
      $dest = $dest ^ $temp;

      set_reg($regd, $dest);
      return true;
}
$reverse_op["r_ANDX"] = "ANDX arg0 arg1 arg2";

function r_ANDIX($regd, $reg1, $imm) {
      debug_msg("ANDIX($regd, $regs, $imm)");

      $dest = get_reg($regd);
      $src1 = get_reg($reg1);

      $temp = $src1 & $imm;
      $dest = $dest ^ $temp;

      set_reg($regd, $dest);
      return true;
}
$reverse_op["r_ANDIX"] = "ANDIX arg0 arg1 arg2";

function r_NORX($regd, $reg1, $reg2) {
      debug_msg("NORX($regd, $reg1, $reg2)");
```

```
	$dest = get_reg($regd);
	$src1 = get_reg($reg1);
	$src2 = get_reg($reg2);

	$temp = ~($src1 | $imm);
	$dest = $dest ^ $temp;

	set_reg($regd, $dest);
	return true;
}
$reverse_op["r_NORX"] = "NORX arg0 arg1 arg2";

function r_NEG($reg) {
	debug_msg("NEG($reg)");

	$foo = get_reg($reg);
	set_reg($reg, -1 * $foo);
	return true;
}
$reverse_op["r_NEG"] = "NEG arg0";

function r_ORX($regd, $reg1, $reg2) {
	debug_msg("ORX($regd, $reg1, $reg2)");

	$dest = get_reg($regd);
	$src1 = get_reg($reg1);
	$src2 = get_reg($reg2);

	$temp = $src1 | $src2;
	$dest = $dest ^ $temp;

	set_reg($regd, $dest);
	return true;
}
$reverse_op["r_ORX"] = "ORX arg0 arg1 arg2";

function r_ORIX($regd, $reg1, $imm) {
	debug_msg("ORIX($regd, $reg1, $imm)");

	$dest = get_reg($regd);
	$src1 = get_reg($reg1);

	$temp = $src1 | $imm;
	$dest = $dest ^ $temp;

	set_reg($regd, $dest);
	return true;
}
$reverse_op["r_ORIX"] = "ORIX arg0 arg1 arg2";

function r_RL($regsd, $amt) {
	debug_msg("RL($regsd, $amt)");

	$temp = get_reg($regsd);
	$MSB = "0";

	for($i = 0; $i < $amt; $i++) {
		$MSB = 0;
		if($temp < 0) {
```

```
                $MSB = 1;
            }
      $temp <<= 1;
            $temp |= $MSB;
    }

      set_reg($regsd, $temp);
      return true;
}
$reverse_op["r_RL"] = "RR arg0 arg1";

function r_RLV($regsd, $regt) {
      debug_msg("XOR($regsd, $regt)");

      $temp = get_reg($regsd);
      $amt = get_reg($regt);

      for($i = 0; $i < $amt; $i++) {
          $MSB = 0;
            if($temp < 0) {
                $MSB = 1;
            }
      $temp <<= 1;
            $temp |= $MSB;
    }

      set_reg($regsd, $temp);
      return true;
}
$reverse_op["r_RLV"] = "RRV arg0 arg1";

function r_RR($regsd, $amt) {
  //RR is the same as (32 - amt) RL
      debug_msg("RR($regsd, $amt)");

      $temp = get_reg($regsd);
      for($i = 0; $i < (32 - $amt); $i++) {
          $MSB = 0;
            if($temp < 0) {
                $MSB = 1;
            }
      $temp <<= 1;
            $temp |= $MSB;
  }

      set_reg($regsd, $temp);
      return true;
}
$reverse_op["r_RR"] = "RL arg0 arg1";

function r_RRV($regsd, $regt) {
      debug_msg("RRV($regsd, $regt)");

      $temp = get_reg($regsd);
      $amt = get_reg($regt);
      for($i = 0; $i < (32 - $amt); $i++) {
          $MSB = 0;
            if($temp < 0) {
                $MSB = 1;
            }
```

```
      $temp <<= 1;
            $temp |= $MSB;
     }
      set_reg($regsd, $temp);
      return true;
}
$reverse_op["r_RRV"] = "RLV arg0 arg1";

function r_SLLX($regd, $regs, $amt) {
      debug_msg("SLLX($regg, $regs, $amt)");

      $dest = get_reg($regd);
      $src = get_reg($regs);

      $temp = $src * pow(2, $amt); // to do a shift,
instead of a rotate.
      $dest = $dest ^ $temp;

      set_reg($regd, $dest);
      return true;
}
$reverse_op["r_SLLX"] = "SLLX arg0 arg1 arg2";

function r_SLLVX($regd, $regs, $regt) {
      debug_msg("SLLVX($regd, $regs, $regt)");

      $dest = get_reg($regd);
      $src = get_reg($regs);
      $amt = get_reg($regt);

      $temp = $src * pow(2, $amt); // to do a shift,
instead of a rotate.
      $dest = $dest ^ $temp;

      set_reg($regd, $dest);
      return true;
}
$reverse_op["r_SLLVX"] = "SLLVX arg0 arg1 arg2";

function r_SRAX($regd, $regs, $amt) {
      debug_msg("SRAX($regd, $regs, $amt)");

      $dest = get_reg($regd);
      $src = get_reg($regs);

      $temp = $src / pow(2, $amt); // to do a shift,
instead of a rotate.
      $dest = $dest ^ $temp;

      set_reg($regd, $dest);
      return true;
}
$reverse_op["r_SRAX"] = "SRAX arg0 arg1 arg2";

function r_SRAVX($regd, $regs, $regt) {
      debug_msg("SRAVX($regd, $regs, $regt)");

      $dest = get_reg($regd);
      $src = get_reg($regs);
      $amt = get_reg($regt);
```

```
      $temp = $src / pow(2, $amt); // to do a shift,
instead of a rotate.
      $dest = $dest ^ $temp;

      set_reg($regd, $dest);
      return true;
}
$reverse_op["r_SRAVX"] = "SRAVX arg0 arg1 arg2";


function r_SRLX($regd, $regs, $amt) {
      debug_msg("SRLX($regd, $regs, $amt)");

      $dest = get_reg($regd);
      $src = get_reg($regs);

  $temp = $src;
      for($i = 0; $i < $amt; $i++) {
            $temp = $src >>  1;
            $temp &= 0x7FFFFFFF;
      }

      $dest = $dest ^ $temp;
      set_reg($regd, $dest);
      return true;
}
$reverse_op["r_SRLX"] = "SRLX arg0 arg1 arg2";

function r_SRLVX($regd, $regs, $regt) {
      debug_msg("SRLVX($regd, $regs, $regt)");

      $dest = get_reg($regd);
      $src = get_reg($regs);
      $amt = get_reg($regt);

  $temp = $src;
      for($i = 0; $i < $amt; $i++) {
            $temp = $src >>  1;
            $temp &= 0x7FFFFFFF;
      }
      $dest = $dest ^ $temp;

      set_reg($regd, $dest);
      return true;
}
$reverse_op["r_SRLVX"] = "SRLVX arg0 arg1 arg2";

function r_SUB( $regd, $regs) {
      debug_msg("SUB($regd, $regs)");
      $dest = get_reg($regd);
      $src = get_reg($regs);

      $dest = $dest − $src;

      set_reg($regd, $dest);
      return true;
}
$reverse_op["r_SUB"] = "ADD arg0 arg1";
```

```
function r_XOR($regd, $regs) {
      debug_msg("XOR($regd, $regs)");

      $dest = get_reg($regd);
      $src = get_reg($regs);

      $dest = $dest ^ $src;

      set_reg($regd, $dest);
      return true;
}
$reverse_op["r_XOR"] = "XOR arg0 arg1";

function r_XORI($regd, $imm) {
      debug_msg("XORI($regd, $imm)");

      $dest = get_reg($regd);

      $dest = $dest ^ $imm;

      set_reg($regd, $dest);
      return true;
}
$reverse_op["r_XORI"] = "XORI arg0 arg1";
?>
```

B.2 asm_branch.php

```
<?

function r_BEQ( $rega, $regb, $of) {
      global $BRANCH_REG;

      debug_msg("BEQ($rega, $regb, $of)");

      $val1 = get_reg($rega);
      $val2 = get_reg($regb);
      if($val1 == $val2) {
            $BRANCH_REG += $of; //fset;
      }
      return true;
}
$reverse_op["r_BEQ"] = "BEQ arg0 arg1 arg2";

function r_BGEZ( $rega,  $of) {
      global $BRANCH_REG;

      debug_msg("BGEZ($rega, $of)");

      $val1 = get_reg($rega);
      if($val1 >= 0) {
            $BRANCH_REG += $of; //fset;
      }
      return true;
}
$reverse_op["r_BGEZ"] = "BGEZ arg0 arg1 ";

function r_BGTZ( $rega,  $of) {
      global $BRANCH_REG;

      debug_msg("BGTZ($rega, $of)");

      $val1 = get_reg($rega);
      if($val1 > 0) {
            $BRANCH_REG += $of;//fset;
      }
      return true;
}
$reverse_op["r_BGTZ"] = "BGTZ arg0 arg1 ";

function r_BLEZ( $rega,  $of) {
      global $BRANCH_REG;

      debug_msg("BLEZ($rega, $of)");

      $val1 = get_reg($rega);
      if($val1 <= 0) {
            $BRANCH_REG += $of;//fset;
      }
      return true;
}
$reverse_op["r_BLEZ"] = "BLEZ arg0 arg1 ";

function r_BLTZ( $rega,  $of) {
      global $BRANCH_REG;
```

```
	debug_msg("BLTZ($rega, $of)");

	$val1 = get_reg($rega);
	if($val1 < 0) {
		$BRANCH_REG += $of;//fset;
	}
	return true;
}
$reverse_op["r_BLTZ"] = "BLTZ arg0 arg1 ";


function r_BNE( $rega, $regb, $of) {
	global $BRANCH_REG;

	debug_msg("BNE($rega, $regb, $of)");

	$val1 = get_reg($rega);
	$val2 = get_reg($regb);
	if($val1 != $val2) {
		$BRANCH_REG += $of;//fset;
	}
	return true;
}
$reverse_op["r_BNE"] = "BNE arg0 arg1 arg2";

function r_BRA($of) {
	global $BRANCH_REG, $REVERSE;

	debug_msg("BRA($of)");

	//$offset = find_offset_line($of);
	//if($offset == 0) die("FATAL ERROR: label '$of'
not found");
	$BRANCH_REG += $of;//fset;
	return true;
}
$reverse_op["r_BRA"] = "BRA arg0";

function sign_extend11($of) {
	if($of & 0x400) {
		$bits = sprintf("%11b", $of);
		for($i = 10; $i >=0; $i--) {
			if($bits[$i] == 0) {
				$result += pow(2, 10 - $i);
			}
		}
		$result += 1;
		$of = $result * -1;
	}

	return $of;
}
?>
```

B.3 asm_special.php

```
<?

function r_EXCH($regd, $rega) {
      debug_msg("EXCH($regd, $rega)");

      $addr = get_reg($rega);
      $val = get_reg($regd);

      $temp = get_mem($addr);

      set_mem($addr, $val);
      set_reg($regd, $temp);
      return true;
}
$reverse_op["r_EXCH"] = "EXCH arg0 arg1";

function r_SWAPBR($reg) {
      global $BRANCH_REG;
      debug_msg("SWAPBR($reg)");

      $temp = $BRANCH_REG;
      $BRANCH_REG = get_reg($reg);
      set_reg($reg, $temp);
      return true;
}
$reverse_op["r_SWAPBR"] = "SWAPBR arg0";

function r_RBRA($of) {
      global $REVERSE, $BRANCH_REG;

      debug_msg("RBRA($of)");

      $REVERSE *= -1;
      //$BRANCH_REG *= -1;

      $BRANCH_REG += $of;
      return true;
}
$reverse_op["r_RBRA"] = "RBRA arg0";

function r_SHOW($reg) {
      debug_msg( "SHOW($reg)");
      output("$reg = " . get_reg($reg)); flush();
      return true;
}
$reverse_op["r_SHOW"] = "SHOW arg0";

function r_OUTPUT($reg) {
      debug_msg( "OUTPUT($reg)");
      output("$reg = " . get_reg($reg));
      return true;
}
$reverse_op["r_OUTPUT"] = "OUTPUT arg0";

function r_FINISH() {
     die("Pendulum Terminated by FINISH op"); flush();
      return true;
```

```
}

function r_START() {
     return true;
}
$reverse_op["r_START"] = "START";
?>
```

B.4 cpu.php

```
#!/usr/local/bin/php

<html>
<style type="text/css">
      BODY { font-family: monotype; }
      TD { font-size: x-small; }
      .output{font-size: medium; font-weight: bold;
color: blue}
      .debug{ font-size: x-small; color: red}
</style>
<font style="font-family: monotype">
<?
set_time_limit(30);


// LOAD FUNCTIONS
include("asm_arith_log.php");         //the alu-type ops
include("asm_branch.php");       //the branch-type ops
include("asm_special.php");           //others
include("enc_arith_log.php");         //the alu-type ops
include("enc_branch.php");       //the branch-type ops
include("enc_special.php");           //others
include("php_functions.php");         //helper functions
include("virt_asm_functions.php");          //some
virtual functions (like DUMP)

// INITIALIZE REGISTERS
$registers = Array();
for($i = 0; $i < 32; $i++) {
      $registers["R$i"] = 0;
}

/* INITIALIZE SYSTEM SETTINGS AND GLOBAL VARS */
$PC = 0;         // current program counter
$REVERSE = 1;          // 1 for forward, -1 for reverse
$BRANCH_REG = 0;       // the branch register
$RAM = Array();        // allocate some memory
$INSTR = "";           // the current instruction
$DECODE_IDX = 0;       // this is used for pseudo
instructions
$SYMBOLS = Array();    // symbol table

/* HANDLE USER SUPPLIED SETTINGS */
$debug = ($debug == "on") ? 1 : 0;
$super_debug = ($super_debug == "on") ? 1 : 0;

$program = compile($code);
/*while(list($symbol, $line) = each($SYMBOLS)) {
      echo "<br>$symbol: $line";
}*/

/* Start!!
 * Here's the drill, this is pretty simple..
 * we keep looping as long as our ProgramCounter is
within range
 * inside the loop we:
 *    -  fetch the isntruction
```

```
 *    -  decode the isntruction
 *    -  execute the instruction
 *    -  dump the registers if we're in super debug
 */
flush();
for($PC = $START_PC; $PC < count($program) && $PC >= 0;
) {
     fetch($PC);
     while(decode_more()) {
          list($op, $arg0, $arg1, $arg2) = decode();
          if(!$op($arg0, $arg1, $arg2)) echo "ERROR:
'$op' is not a valid PISA operation<br>";
          if($super_debug) r_DUMP();
     }
     $PC += ($BRANCH_REG == 0) ? $REVERSE : $BRANCH_REG;
}
?>
</font>
```

B.5 editor.html

```
<html>
<head><title>Pedumum ISA Simulator</title>
<body>
<h3>Pendulum ISA Simulator</h3>
<form action="cpu.php" method="post"
enctype="multipart/form-data" target="cpu">
Options:<br>
<input type="checkbox" name="debug"> Debug (displays the
operations as they're executed)<br>
<input type="checkbox" name="super_debug"> Super Debug
(displays register and memory dump after each
operation).<br>
<br>
Program Here:<br>
<textarea cols="80" rows="20"
name="code"></textarea><br>
<input type=submit value="Run">
</form>



<p>
    <a
href="http://validator.w3.org/check?uri=http%3A%2F%2Fwww
.cise.ufl.edu%2Fresearch%2Frevcomp%2Fphpisa&doctype=Inli
ne" target="_top"><img border="0"
        src="http://www.w3.org/Icons/valid-html401"
        alt="Valid HTML 4.01!" height="31"
width="88"></a>
  </p>
</html>
```

B.6 enc_arith_log.php

```
<?
function e_ADD( $regd, $regs) {
      reg($regd); reg($regs);
      return encode(0, $regd, $regd, $regs, 0);
}
$decode_op[0] = "ADD reg1 reg3";

function e_ADDI($regd, $imm) {
      reg($regd);

         if(ereg("[[:alpha:]]+", $imm)) {
                 $imm = find_address($imm);
         }

      return encode(1, $regd, $regd, ($imm &
0x7C0)/pow(2,6), $imm & 0x3F);
}
$decode_op[1] = "ADDI reg1 imm";

function e_ANDX($regd, $reg1, $reg2) {
      reg($regd); reg($reg1); reg($reg2);
      return encode(2, $regd, $reg1, $reg2, 0);
}
$decode_op[2] = "ANDX reg1 reg2 reg3";

function e_ANDIX($regd, $reg1, $imm) {
      reg($regd); reg($reg1);
      return encode(3, $regd, $reg1,  ($imm &
0x7C0)/pow(2,6), $imm & 0x3F);
}
$decode_op[3] = "ANDIX reg1 reg2 imm";

function e_NORX($regd, $reg1, $reg2) {
      reg($regd); reg($reg1); reg($reg2);
      return encode(4, $regd, $reg1, $reg2, 0);
}
$decode_op[4] = "NORX reg1 reg2 reg3";

function e_NEG($reg) {
      reg($reg);
      return encode(5, $reg, $reg, $reg, 0);
}
$decode_op[5] = "NEG reg1";

function e_ORX($regd, $reg1, $reg2) {
      reg($regd); reg($reg1); reg($reg2);
      return encode(6, $regd, $reg1, $reg2);
}
$decode_op[6] = "ORX reg1 reg2 reg3";

function e_ORIX($regd, $reg1, $imm) {
      reg($regd); reg($reg1);
      return encode(7, $regd, $regd,  ($imm &
0x7C0)/pow(2,6), $imm & 0x3F);
}
$decode_op[7] = "ORIX reg1 reg2 imm";
```

```
function e_RL($regsd, $amt) {
      reg($regsd);
      return encode(8, $regsd, $regsd, ($amt &
0x7C0)/pow(2,6), $amt & 0x3F);
}
$decode_op[8] = "RL reg1 imm";

function e_RLV($regsd, $regt) {
      reg($regsd);reg($regt);
      return encode(9, $regsd, $regsd, $regt, 0);
}
$decode_op[9] = "RLV reg1 reg3";

function e_RR($regsd, $amt) {
      reg($regsd);
      return encode(10, $regsd, $regsd, ($amt &
0x7C0)/pow(2,6), $amt & 0x3F);
}
$decode_op[10] = "RR reg1 imm";

function e_RRV($regsd, $regt) {
      reg($regsd);reg($regt);
      return encode(11, $regsd, $regsd, $regt, 0);
}
$decode_op[11] = "RRV reg1 reg3";

function e_SLLX($regd, $regs, $amt) {
      reg($regd);reg($regs);
      return encode(12, $regd, $regs, ($amt &
0x7C0)/pow(2,6), $amt & 0x3F);
}
$decode_op[12] = "SLLX reg1 reg2 imm";

function e_SLLVX($regd, $regs, $regt) {
      reg($regd);reg($regs);reg($regt);
      return encode(13, $regd, $regs, $regt, 0);
}
$decode_op[13] = "SLLVX reg1 reg2 reg3";

function e_SRAX($regd, $regs, $amt) {
      reg($regd);reg($regs);
      return encode(14, $regd, $regs, ($amt &
0x7C0)/pow(2,6), $amt & 0x3F);
}
$decode_op[14] = "SRAX reg1 reg2 imm";

function e_SRAVX($regd, $regs, $regt) {
      reg($regd);reg($regs);reg($regt);
      return encode(15, $regd, $regs, $regt, 0);
}
$decode_op[15] = "SRAVX reg1 reg2 reg3";

function e_SRLX($regd, $regs, $amt) {
      reg($regd);reg($regs);$amt+=0;
      return encode(16, $regd, $regs, ($amt &
0x7C0)/pow(2,6), $amt & 0x3F);
}
$decode_op[16] = "SRLX reg1 reg2 imm";
```

```
function e_SRLVX($regd, $regs, $regt) {
      reg($regd);reg($regs);reg($regt);
      return encode(17, $regd, $regs, $regt, 0);
}
$decode_op[17] = "RRV reg1 reg2 reg3";

function e_SUB( $regd, $regs) {
      reg($regd);reg($regs);
      return encode(18, $regd, $regd, $regs, 0);
}
$decode_op[18] = "SUB reg1 reg3";

function e_XOR($regd, $regs) {
      reg($regd);reg($regs);
      return encode(19, $regd, $regd, $regs, 0);
}
$decode_op[19] = "XOR reg1 reg3";

function e_XORI($regd, $imm) {
      reg($regd);
      return encode(20, $regd, $regd,  ($imm &
0x7C0)/pow(2,6), $imm & 0x3F);
}
$decode_op[20] = "XORI reg1 immm";

function e_DATA($imm) {
      return $imm;
}
?>
```

B.7 enc_branh.php

```
<?
function e_BEQ( $rega, $regb, $of) {
      $of = find_offset_line($of);
      reg($rega);reg($regb);
      return encode(21, $rega, $regb, (0x7C0 &
$of)/pow(2,6), 0x3F & $of);
}
$decode_op[21] = "BEQ reg1 reg2 imm";

function e_BGEZ( $rega,  $of) {
      $of = find_offset_line($of);
      reg($rega);
      return encode(22, $rega, $rega,  (0x7C0 &
$of)/pow(2,6), 0x3F & $of);
}
$decode_op[22] = "BGEZ reg1  imm";

function e_BGTZ( $rega,  $of) {
      $of = find_offset_line($of);
      reg($rega);
      return encode(23, $rega, $rega,  (0x7C0 &
$of)/pow(2,6), 0x3F & $of);
}
$decode_op[23] = "BGTZ reg1 imm";

function e_BLEZ( $rega,  $of) {
      $of = find_offset_line($of);
      reg($rega);
      return encode(24, $rega, $rega,  (0x7C0 &
$of)/pow(2,6), 0x3F & $of);
}
$decode_op[24] = "BLEZ reg1 imm";

function e_BLTZ( $rega,  $of) {
      $of = find_offset_line($of);
      reg($rega);
      return encode(25, $rega, $rega,  (0x7C0 &
$of)/pow(2,6), 0x3F & $of);
}
$decode_op[25] = "BLTZ reg1 imm";

function e_BNE( $rega, $regb, $of) {
      $of = find_offset_line($of);
      reg($rega); reg($regb);
      return encode(26, $rega, $regb, (0x7C0 &
$of)/pow(2,6), 0x3F & $of);
}
$decode_op[26] = "BNE reg1 reg2 imm";

function e_BRA($of) {
      $of = find_offset_line($of);
      return encode(27, 0, 0, (0x7C0 & $of)/pow(2,6),
0x3F & $of);
}
$decode_op[27] = "BRA imm";

?>
```

B.8 enc_special.php

```
<?

function e_EXCH($regd, $rega) {
      reg($regd);reg($rega);
      return encode(28, $regd, $regd, $rega, 0);
}
$decode_op[28] = "EXCH reg1 reg3";

function e_SWAPBR($reg) {
      reg($reg);
      return encode(29, $reg, $reg, $reg, 0);
}
$decode_op[29] = "SWAPBR reg1";

function e_RBRA($of) {
      $of = find_offset_line($of);
      return encode(30, 0, 0, (0x7C0 & $of)/pow(2,6),
0x3F & $of);
}
$decode_op[30] = "RBRA imm";

function e_SHOW($reg) {
      reg($reg);
      return encode(31, $reg, $reg, $reg, 0);
}
$decode_op[31] = "SHOW reg1";

function e_OUTPUT($reg) {
      reg($reg);
      return e_SHOW($reg);
}

function e_FINISH() {
      return encode(32, 0, 0, 0, 0);
}
$decode_op[32] = "FINISH";

function e_START() {
      return encode(33, 0, 0, 0, 0);
}
$decode_op[33] = "START";

?>
```

B.9 php_functions.php

```
<?

function get_reg($reg) {
      global $registers;
      return $registers[$reg];
}

function set_reg($reg, $val) {
      global $registers;
      $registers[$reg] = $val;
}

function get_mem($addr) {
      global $program;
      if(isset($program[$addr])) {
            $result =
ereg_replace("(.*:[[:space:]]+)?DATA([[:space:]]+)", "",
$program[$addr]);
            return ($result + 0);
      } else {
            return 0;
      }
}

function set_mem($addr, $val) {
      global $program;
            $program[$addr] = $val;
}

function debug_msg($msg) {
      global $debug, $PC;
      if($debug) {
            echo "<div class=debug>$PC: $msg</div>";
            flush();
      }
}

function output($msg) {
      echo "<div class=output>$msg</div>";
      flush();
      return true;
}

function find_offset_line($label) {
      global $SYMBOLS, $PC;

      if(isset($SYMBOLS[$label])) {
            return ($SYMBOLS[$label] - $PC) + 1;
      }
      die("<br>FATAL ERROR: Label $label not found");

}

function find_address($label) {
      global $SYMBOLS;

      $x = 1;
```

```
	if(substr($label, 0, 1) == "-") {
		$label = substr($label,1,strlen($label));
		$x = -1;
	}

	if(isset($SYMBOLS[$label])) {
		return($x * $SYMBOLS[$label]);
	}
	die("<br>FATAL ERROR: Address '$label' not found");
}

function fetch($inst) {
	global $program, $INSTR, $DECODE_IDX, $decode_op;
	$instruction = $program[$inst];

	$operation = $decode_op[get_part($instruction,
"opcode") + 0];
	$parts = split(" ", $operation);
	$INSTR = $parts[0];
	next($parts);
	while(list(,$part) = each($parts)) {
		$INSTR .= " " . get_part($instruction, $part);
	}

	$DECODE_IDX = 0;
}

function decode() {
	global $INSTR, $REVERSE, $DECODE_IDX, $reverse_op;


	@list($op, $arg0, $arg1, $arg2, $arg3) = split(" ",
$INSTR);
	$INSTR = str_replace("$", "R", $INSTR); //allow $
or R registers
	$INSTR = ereg_replace("([[:space:]]+)", " ",
$INSTR); //replace space runs with single spaces
	$INSTR = ereg_replace("(.*:[[:space:]]+)+", "",
$INSTR); //strip off the label
	$INSTR = ereg_replace("^([[:space:]])+", "",
$INSTR); //remove any spaces at the beginning
	$INSTR = ereg_replace(";.*", "", $INSTR);
	@list($op, $arg0, $arg1, $arg2, $arg3) = split(" ",
$INSTR);

	$op = "r_" . $op;

	if($REVERSE == 1) {
		$DECODE_IDX = -1;
		return split(" ", $op . " " . $arg0 . " " .
$arg1 . " " . $arg2);
	} else {
		$operations = split(";", $reverse_op[$op]);
		if($DECODE_IDX < count($operations)) {
			$line = $operations[$DECODE_IDX];
			$line = str_replace("$", "R", $line);
//allow $ or R registers
			$line = ereg_replace("([[:space:]]+)", "
", $line); //replace space runs with single spaces
			$line = ereg_replace("^([[:space:]])+",
```

```
"", $line); //remove any spaces at the beginning
                @list($Rop, $Rarg0, $Rarg1, $Rarg2) =
split(" ", $line);
                $Rop = "r_" . $Rop;

                $DECODE_IDX++;
           }

           if($DECODE_IDX == count($operations)) {
                $DECODE_IDX = -1;
           }
           //output("$Rop $Rarg0 $Rarg1 $Rarg2");
           return @split(" ", $Rop . " " . $$Rarg0 . " "
. $$Rarg1 . " " . $$Rarg2);
      }

}

function decode_more() {
      global $REVERSE, $DECODE_IDX;
      if($REVERSE == 1) {
           if($DECODE_IDX == 0)
                return true;
           else
                return false;
      } else {
           if($DECODE_IDX < 0)
                return false;
           else
                return true;
      }
}


function encode($op, $r1, $r2, $r3, $imm6) {
      $result = ($op & 0x3F); // 6 bit op
      $result *= pow(2,5);
      $result += ($r1 & 0x1F); // 5 bit reg1
      $result *= pow(2,5);
      $result += ($r2 & 0x1F); // 5 bit reg2
      $result *= pow(2,5);
      $result += ($r3 & 0x1F); // 5 bit reg3 or upper 5
bits of imm
      $result *= pow(2,6);
      $result += ($imm6 & 0x3F); // lower 6 bits of imm
      return $result;

}

function get_part($data, $part) {
      switch ($part) {
           case "opcode":
                return (($data / pow(2, 21)) & 0x3F);
                break;
           case "reg1":
                return "R" . (($data / pow(2, 16)) &
0x1F);
                break;
           case "reg2":
                return "R" . (($data / pow(2,11)) &
```

```
0x1F);
                break;
        case "reg3":
                return "R" . (($data / pow(2,6)) &
0x1F);
                break;
        case "imm":
                return sign_extend11($data & 0x7FF);
                break;
    }
}

function compile($code) {
    global $SYMBOLS, $START_PC, $PC; // the pc is
needed to find the offsets... we'll put it back when
we're done

    $code = split("\n", $code);
    $clean = Array();
    $program = Array();
    //first, clean it up... ditch the blank lines.. and
comments
    foreach($code as $line) {
        $line = strtoupper($line);
        $line = str_replace(",", "", $line); //ditch
the commas
        $line = str_replace("$", "R", $line); //allow
$ or R registers
        $line = ereg_replace("([[:space:]]+)", " ",
$line); //replace space runs with single spaces
        $line = trim($line);

        if(strlen(stristr($line,".start")) > 0) {
            $line = trim($line);
            list(,$start) = split(" ", $line);
            $START_PC = $start;
            continue;
        }

        if(strlen($line) == 0) {
            continue;
        }

        $pos = strpos($line, ";");
        if($pos >0 || substr($line, 0, 1) == ";" ) {
            $line = substr($line, 0, $pos);
        }
        $line = trim($line);
        if(strlen($line) == 0)
            continue;
        $clean[] = $line;
    }

    for($i = 0; $i < count($clean); $i++) {
        $line = $clean[$i];
        $pos = strpos($line, ":");
        if($pos > 0) {
            $label = substr($line, 0, $pos);
            $SYMBOLS[$label] = $i;
            $clean[$i] = trim(substr($line, $pos +
```

```
1, strlen($line)));
            }
      }

      reset($clean);
      foreach($clean as $line) {
            list($op, $arg1, $arg2, $arg3) = split(" ",
$line);

            $PC++;
            if(!function_exists("e_$op")) {
                  echo "<br>ERROR: $op is not a PISA
instruction";
                  exit();
            }
            $op = "e_$op";
            $program[] = $op($arg1, $arg2, $arg3);
      }

      if(isset($START_PC) && strlen($START_PC) > 0) {
            $PC = 1;
            $START_PC = find_offset_line($START_PC);
      } else {
            $START_PC = 0;
      }

      $PC = 0; //just like i promised!
      reset($program);
      return $program;

}


function reg(&$reg) {
      $reg = str_replace("R", "", $reg);
}
?>
```

B.10 phpisa.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Frameset//EN">

<html>
<head>
     <title>PISA Emulator</title>
</head>

<!-- frames -->
<frameset  cols="*,320">
    <frame name="editor" src="editor.html"
scrolling="auto" frameborder="0">
    <frame name="cpu" src="cpu.php" scrolling="auto"
frameborder="0">
</frameset>

</html>
```

B.11 virt_asm_functions.php

```
<?
function r_NOOP() {
//do nothing
}
$reverse_op["r_NOOP"] = "NOOP";

function r_DUMP() {
     global $registers, $PC, $BRANCH_REG, $REVERSE,
$program;
     //debug_msg("DUMP()");
     reset($registers);
     echo "<table>";
     for($i = 0; list($name, $value) = each
($registers); $i++) {
          echo ($i % 4 == 0) ? "<tr>" : "";
          printf("<td>%3s: %08x</td>", $name, $value);
          echo ($i % 4 == 3) ? "</tr>" : "";
     }
     echo "<tr>";
     printf("<td>PC: %08x</td>", $PC);
     printf("<td>BR: %08x</td>", $BR);
     printf("<td>DIR: %s</td>", ($REVERSE == 1) ?
"FORWARD" : "REVERSE");
     printf("<td> </td>");
     echo "</tr></table>\n";
     /*
     //now let's dump the memory..
     echo "RAM: <table><tr><td>";
     reset($program);
     ksort($program);
     while(list($addr, $value) = each($program)) {
          echo "$addr = $value<br>";
     }
     echo "</td></tr></table>";
     */
     return true;
}
$reverse_op["r_DUMP"] = "DUMP";


function r_WARN($op) {
     debug_msg("WARNING: No reverse operation for $op is
defined");
}
$reverse_op["r_WARN"] = "NOOP";



?>
```

C

Works Cited / Website Links

C.1 Works Cited

Frank, Michael. Reversibility for Efficient Computing
May, 1999 <http://www.cise.ufl.edu/~mpf/rc/thesis/phdthesis.html>

Vieri, Carlin. Reversible Computing for Energy Efficient and Trustable Computation
April, 1998 <http://www.ai.mit.edu/~cvieri/reversible.html>

C.2 Website Links

University of Florida Reversible Computing Homepage
<http://www.cise.ufl.edu/research/revcomp/>

phPISA Homepage
<http://www.cise.ufl.edu/research/revcomp/>

PHP Homepage
<http://www.php.net/>