

Program Analysis and Transformation

Final Assignment

Benjamin Brandt Ohrt, zpn492

June 2, 2019

Abstract

The Pendulum Instruction Set Architecture (PISA) was first introduced in 1995 by Carlin James Vieri and is a reversible assembly language. The assembly language has later been improved and several high-level languages has been build upon it. One of those languages is the extension to the reversible object-oriented programming language (ROOPL) ROOPL++ presented in 2018. One of the issues with ROOPL++ was the amount of produced target code. In this paper we compile source code for the data structure of a binary tree, analyse the possibility of redundant target code and points out where an optimization could be done.

Contents

1	Introduction	3
1.1	Assembly language	3
1.1.1	Example program	3
1.2	Reversible computing	4
1.3	Pendulum microprocessor	5
1.4	Motivation	6
1.5	Outline	6
2	Programming languages	7
2.1	Pendulum Assembly Language	7
2.1.1	Example program	7
2.2	Reversible Object-Oriented Programming	8
3	Analysis of redundancy	9
3.1	First part - patterns	10
3.1.1	Getting data from memory	10
3.1.2	Special if-statements	11
3.2	Second part - methods for reduced redundancy	12
3.2.1	PendVM - OUT	12
3.2.2	PendVM - EXCHI	13
3.2.3	Algorithm - Memory exchange	14
3.2.4	Algorithm - Special if-statements	14
3.2.5	Algorithm - Remove redundancy	14
4	Results	16
5	Data structures	17
5.1	Binary tree	17
6	Conclusion	17
7	Further work	17
7.0.1	Instruction operands - [OVERVEJ]	17
7.0.2	Pendulum operation	18
7.1	Register values	19
7.2	Pendulum	20
8	Appendix A	21
9	Appendix B	23
9.1	Full program	23
9.1.1	Malloc	31

1 Introduction

1.1 Assembly language

A CPU understands its own machine language. Instructions in machine language are numbers stored as bytes in memory. Each instruction has its own unique numeric code called its operation code or opcode for short. An example is the instruction that says add EAX and EBX registers together and store the result back into EAX is encoded by the following hex code:

03 C3

This not readable for the blunt eye. Which is why we have a program called an assembler. An assembler is a program that reads the text of a assembly language program and converts the assembly into machine code. [6, p.11].

The assembly language origins back to Birkbeck College 1946-1962, where the creation was credited to Kathleen Booth.[1] Kathleen Booth was a PhD whom both visited with John Von Neumann with the famous The Von Neumann Architecture[3], helped with the design of multiple machines and was one the founders of the Birkbeck department of computer science.

The assembly language is a symbolic programming language, closest to machine code.[2] An assembly language program is stored as plain text. The text consists of a set of instructions which is processed chronological. Each assembly instruction is equal to one machine instruction for example could above addition be shown as:

add EAX EBX

With the assembly language words as add can be used as mnemonic for the instruction add and so fourth. [6, p.11].

1.1.1 Example program

ForITo100.asm

1	J main	<i>; for int i = 0; i < 100; i++</i>
2		<i>; count += i * 2</i>
3	loop_i:	
4	MUL \$t0, \$a1, 2	
5	ADD \$a0, \$a0, \$t0	<i>; count += i * 2</i>
6	ADDI \$a1, \$a1, 1	<i>; i++</i>
7	SLT \$t0, \$a1, \$a2	
8	BEQ \$t0, 1, loop_i	<i>; i < 100</i>
9	JR \$ra	<i>; return</i>
10		
11	main:	
12	ADDI \$a0, \$zero, 0	<i>; count = 0</i>
13	ADDI \$a1, \$zero, 0	<i>; i = 0</i>
14	ADDI \$a2, \$zero, 100	<i>; max</i>
15	JAL loop_i	

1.2 Reversible computing

A reversible computing system has, at any time, at most a single previous computation state as well as a single next computation state, and thus a reversible computing system.[7] It comes with the promise of reduced energy dissipation, when erasure of information is left out of the program.

The inspiration of reversible computing dates back to 1867 and the study of thermodynamics where James Clark Maxwell made a thought experiment also known as Maxwell's Demon. In the experiment he questions that of the second law of thermodynamics which says:

[..] It is impossible in a system enclosed in an envelope which permits neither change of volume nor passage of heat, and in which both the temperature and the pressure are everywhere the same, to produce any inequality of temperature or pressure without expenditure of work.[8, p.16]

The experiment was later published in 1872 in a book by Maxwell: Theory of Heat[11], but the connection between thermodynamics and computations was first seen in 1949 by John von Neumann, where he talks about a computation must have a minimum thermodynamic energy dissipation which he determines to[5, p.20][8, p.18]:

$k_B T \ln N$, k_B as the Boltzmann's constant, T as the temperature and $N = 2$.

Rolf Landauer realize that modern computers with irreversible processes must dissipate that minimum energy as von Neumann described if the erasure of a bit occurs thus resulting in atleast the increase of entropy by $k_B \ln 2$. [5, p.20][8, p. 18] He also states that reversible operations does not produce the dissipation and irreversible operations can avoid the dissipation by storing information of the computational history. At last Landauer states that the stored history must be erased irreversibly which would just postpone the dissipation of entropy, That last statement is later proven wrong by Charles Bennett.[8, p.19]

In 1970 Charles Bennett looks at Landauers last statement and decides to make an experiment. He creates a reversible program consisting of two halves. The first which provided the calculations intended and the second which undid the calculations from the first halves thus ending the program in its starting point. In the experiment he uses a Turing Machine[10] and shows that the information produces by the intended calculation could be used to undo the calculation instead of just be thrown away. With that experiment Bennett had shown that a reversible process would not just postpone the dissipation of energy it would be able to remove it from the equation.[8, p.18]

From Bennetts experiment he came up with a description for an "enzymatic Turing machine" in which reversible logically operations could be executed[8, p.19]. To fulfill the promise of reduced energy dissipation their was final need of physically logic devices to perform the reversible operations on. Fredkin, Toffoli and Feynmann all aided this need by introducing reversibly logic-gates.[8, p. 19, 22]

1.3 Pendulum microprocessor

Pendulum is a reversible microprocessor, invented by Carlin James Vieri at MIT in 1995. The invention took offset in the existing MIPS R2000 architecture.[REF til ARK][4, p. 29] Vieri's motivation was to create a reversibly processor which would avoid destruction of information and reduce the energy dissipation shown in thermodynamics[1.2] and discussed by Charles Bennet.[8] For this he would focus on memory access, datapath operations on stored values and control flow operations.

The processor has three registers for used in the control flow:

- 1 The program counter (PC) for storing adress of the current instruction
- 2 The branch register (BR) for storing jump offsets
- 3 The direction bit (DIR) for keeping track of the execution direction.

PC is incremented/decremented by the value stored in DIR, thus changing DIR between 1 & -1, will decide in which direction the instructions will be executed. When the DIR is -1, all instructions is inverted.[4, Vieri - Chapter 3, p. 21]

Control flow - Branch instructions like branch or jump is in normal architectures not reversible, that is because the come-from instruction is not stored anywhere, thus the goto instruction not knowing who called it. This could be handled by storing the PC just before the branch/jump in a special register. In later versions of the Pendulum ISA architecture paired branches is introduced, such that each goto instruction should have branch instruction to the come-from.

Control flow - If/then statements is in need of an exit condition known as assert.

if e1 **then** s1 **else** s2 **fi** e2

Control flow - for-loop is in need of an entry, loop and exit condition. Where the e1 should only be true upon entry and e2 only on exit.

from e1 **do** s1 **loop** s2 **until** e2

Control flow - subrutines in encased between a top and a bottom, which both contains a branch to eachother. It follows that the subroutine is skipped when executing instructions sequentially.

top: BRA bot

<..rutine definition goes here..>

bot: BRA top

Memory access is always an exchange. The exchange instruction swaps register value with the value in memory at an address specified by another register.[4, p. 32]

1.4 Motivation

1.5 Outline

2 Programming languages

2.1 Pendulum Assembly Language

Pendulum Instruction Set Architecture(PISA) Assembly Language or a shorthand PAL, is a reversible language. It assembles by the Pendulum microprocessor.[4, p. 48] In this paper we use the Pendulum virtual machine PendVM[9], which executes programs written in PAL. Pendulum instructions is almost identically to the conventional RISC(Reduced Instruction Set Computing) processor. Thus reversible it introduces some new features.

Branch register is normally zero, if the branch register is not zero, the PC increments by the value in the register.[REF - PENDULUM 248] When it happens it the instruction at the destination executes and the branch register is cleared. When implementing subroutines one must use SWAPBR which allows direct access to the branch register, and therefor an exchange the value. It must follow that the subroutine negates the value in the branch register, such that the next SWAPBR will branch back to the location it came from. The branch at the location cancels out the branch register and the PC continues sequentially. SWAPBR has also the capability of performing switch statements. [REF - PENDULUM p. 281] START and FINISH marks the beginning and end of a program. BRA is an unconditional jump.

2.1.1 Example program

ForITo100.pal

```
1  ;; main
2  ;;; increment a from [1..n]
3  subtop:  BRA subbot
4  main:    SWAPBR $2          ; entry/exit point
5           NEG $2            ; negate offset to return caller
6           EXCH $2 $1        ; push return offset to stack
7           ADDI $30 100      ; a += n
8           BRA swap          ; set limit ($28) += a, set a = 0
9  looptop: BNE $30 $0 loopbot ; from a = 0 do
10          ADDI $30 1         ; a += 1
11  loopbot: BNE $30 $28 looptop; until a = limit loop body
12          SUB $28 $30        ; set limit = 0
13          ADDI $30 -100      ; a -= n
14          EXCH $2 $1        ; pop return address
15  subbot:  BRA subtop
```

The entire program is shown in the Appendix 8. We have in section 1.1.1 seen how a similar for-loop is written in the irreversible assembly language MIPS. Now lets take a look on the for-loop in the reversible assembly language PAL.

First off the main method is wrapped within a top and a bot. Next we store the offset, so on the end of the method we are able to return to the caller. Now the looptop is checked once like from a = 0 then proceed to the body. From here on the body is executed once and once again until the criteria in loopbot is true.

2.2 Reversible Object-Oriented Programming

Reversible Object-Oriented Programming (ROOPL) and its predecessor ROOPL++ which expanded the language with dynamic memory was created by Tue Haulund and Martin Cservenka. It is build with inspiration from the reversible programming language Janus and compiles to the Pendulum Assembly language.

simplePrg.rplpp

```
1 class Program
2     int nodeCount
3     int limit
4     method main()
5         limit += 100
6         from nodeCount = 0 do
7             skip
8         loop
9             nodeCount += 1
10            until nodeCount = limit
11            nodeCount -= limit
12            limit -= 100
```

This simple program is similar to the two earlier shown assembly programs. Though when compiled the number of instructions explodes. In Appendix 9 the full list of instructions is shown, it sums up to near 400 lines of program.

Lets divide the program into bites:

Part	Lines	Description
Static	4	DATA instruction
Malloc	205	Method for memory allocation
Main	134	Main method equal two the other assembly programs
Program structure	54	START .. FINISH

With the above division, its clear the that even without the malloc method, the compiled program is still way bigger than the simple self-written PAL program.

Now lets take a closer look on the main method. We see that 26 lines holds a bunch of if-statements with nameded macros as **cmp** og **f**. The entire entry macro counts 57 lines including the special if-statements. The initialization and clean-up is 20 and 35 lines, where the loop-body is 14 lines.

Part	Lines
1_ main_ 0 to entry	20
entry to test	57
test	14
after exit	35

In the next chapter the focus is on the main method and the program structure. We look into the possibility of redundancy and search for patterns which is unnecessary for the program.

3 Analysis of redundancy

In this section we analyse the possibility of redundant target code after compilation of the ROOPL++ program introduced in section 2.2. The analysis is split into two parts.

The first part will be a stepwise approach going through the following states:

Discover patterns which is repeated multiple times through out the target code.

Isolate the patterns into subroutines to make the target code more readable.

Group subroutines by purpose

Try to reduce the groups, for example by introducing new instructions.

The second part It also describes methods for reducing redundancy. Within this we define a smaller area of redundancy:

Redundancy is the same instruction occuring multiple times in a row in which all affect the same register. For example is the two instructions below seen as redundant:

```
ADDI $3 10
```

```
ADDI $3 8
```

Instead the target code could have used a single instruction:

```
ADDI $3 18
```

3.1 First part - patterns

This is the first part of the analysis, where we try to identify patterns which is occurring multiple times throughout the target code.

3.1.1 Getting data from memory

Multiple times we'll see that, the pattern for exchange between a register value and the place in memory. We look into an example starting between entry and the first **cmp**.

The pattern goes like this: Put the address of nodeCount into register 7. Use the address in register 7 to exchange the nodeCount in memory with the zero value in register 8. Register 8 now holds the value of nodeCount. This part is reached several times, once for each time nodeCount is not 100. This exchange is done 6 times.

We isolate the pattern by introducing a subroutine count. Count can replace the 6 redundant patterns:

```
subtop: BRA subbot
count:  SWAPR    $2
        NEG      $2
        EXCH     $2 $1
        ADD      $7 $3
        ADDI     $7 2
        EXCH     $8 $7
        ADDI     $7 -2
        SUB      $7 $3
        EXCH     $2 $1
subbot: BRA subtop
```

We see that the exchange part takes up 5 lines, where the subroutine takes 10 +1 for the call. Which means all exchanges which are done more than 2 times could line wise be subroutines. Another benefit by isolating exchanges is the readability in the actual main method.

This pattern is not only seen when retrieving the nodeCount, it is seen every time memory and a register has to communicate. But this is the same as change the register value r1 with the value in memory of the stack pointer + some offset. One could formalize a method m with three arguments: m(r1, sp, offset). Why this could be a place to introduce a new instruction.

3.1.2 Special if-statements

Through out the target code, we seen multiple if-statements, properly auto-generated, with labels like **cmp** and **f**. Lets start by isolating the areas with special if-statements.

```
subtop_if_1:      BRA subbot_if_1
if_1:             SWAPR $2
                  NEG $2
                  EXCH $2 $1
cmp_top_15:       BNE $8 $0 cmp_bot_16
                  XORI $9 1
cmp_bot_16:       BNE $8 $0 cmp_top_15
f_top_17:         BEQ $9 $0 f_bot_18
                  XORI $10 1
f_bot_18:         BEQ $9 $0 f_top_17
                  XOR $6 $10
f_bot_18_i:       BEQ $9 $0 f_top_17_i
                  XORI $10 1
f_top_17_i:       BEQ $9 $0 f_bot_18_i
cmp_bot_16_i:     BNE $8 $0 cmp_top_15_i
                  XORI $9 1
cmp_top_15_i:     BNE $8 $0 cmp_bot_16_i
                  EXCH $2 $1
subbot_if_1:      BRA subtop_if_1
```

At first glance we see that the purpose is to set \$6 at one or zero. Which means that a simple if-statement could possibly replace the above both complicated and hard-to-read set of instructions. Lets try to figurer out the parts. Lets say if $8 == 0$, then XOR \$6 by \$10, where \$10 in is the same as 1. This could be written as below.

```
iftop_3:          BNE $8 $0 ifbot_3
                  XORI $6 1
ifbot_3:          BNE $8 $0 iftop_3
```

If this holds, we could when approaching the part where the special if-statement occures take the first line and let it be the condition, find the XOR which is not encapsulated within a label, exchange the XOR with XORI and exchange the second register value with a 1.

3.2 Second part - methods for reduced redundancy

In this section we upgrade PendVM^[9] the virtual machine on which we can run PAL programs. We further introduce some new algorithm which can apply on the target code of a compiled ROOPL++ program.

Main algorithm of the section is remove_redundancy which we'll later apply to sample programs for testing.

3.2.1 PendVM - OUT

PendVM does not support methods to save the results of an executed program, which is why i have expanded one of the instructions: OUT.

OUT is now able to write or append a register value to the text-file static_storage.text. For now ROOPL has not been expanded which means, that using the OUT functionality requires three steps:

The immediate value in ADDI just after START has to be incremented by the value of need OUT instructions.

The immediate value in ADDI just before FINISH has to be incremented by the value of need OUT instructions.

OUT instruction has to be manually inserted into the resulting .pal file of a compiled ROOPL++ program.

The expansion of PendVM exists of the following files:

```
int i_out(WORD WORD WORD);
```

pendvm.h

```
("OUT", {REG, REG, NIL}, i_out)
```

pal_parse.c

```
int
i_out(WORD r, WORD u1, WORD u2)
{
    char buf[.];
    sprintf(buf, '%d', m->reg[r1]);
    const char *p = buf;

    FILE *file;
    file = fopen('static_storage.txt', m->reg[r2]);
    fputs(p, file);
}
```

```

fclose(file);
return 0;
}

```

machine.c

3.2.2 PendVM - EXCHI

Now what we have seen is that the exchange of memory and register values is a 5 line procedure, which is occurring multiple times within the target code of a ROOPL++ program. Which is why i want to introduce a new instruction called EXCHI, it takes the 3 arguments, two registers and an immediate. First register will get its value swapped with the value in the memory of position of the second register + immediate. The expansion of PendVM exists of the following files:

```
int i_exchpi(WORD WORD WORD);
```

pendvm.h

```
("EXCHI", {REG, REG, IMM}, i_exchi)
```

pal_parse.c

```

int
i_exchi(WORD rd, WORD ra, WORD ul)
{
    WORD tmp;
    /* ra is the stack pointer
     * ul is the offset */
    MEMORY *loc = mem_get(m->reg[ra]+ul);

    if (loc->type == MEM_INST) {
        pendvm_error("..");
        return -4; /* "exchange with instruction" error */
    }

    tmp=m->reg[rd];
    m->reg[rd]=loc->value;
    loc->value=tmp;
    return 0;
}

```

machine.c

3.2.3 Algorithm - Memory exchange

Find the pattern:

ADD	\$r1 \$r2
ADDI	\$r1 imm
EXCH	\$r3 \$r1
ADDI	\$r1 -imm
SUB	\$r1 \$r2

Replace it with:

EXCHI	\$r3 \$r2 imm
--------------	---------------

3.2.4 Algorithm - Special if-statements

Find the pattern:

cmp_top_x	inst_x
cmp_bot_x	inst_x
<... >	
XOR	\$r1 \$r2
<... >	
cmp_bot_x_i	inst_x_i
cmp_top_x_i	inst_x_i

Replace it with:

if_top_x:	inst_x
XORI	\$r1 1
if_bot_x:	inst x

3.2.5 Algorithm - Remove redundancy

For the task of a successfull analysis we need an algorithm in which can find *redundancy*, merge and remove instruction and atleast update the stack point with the new immediate value. This algorithm can be applied to an existing PAL program, which means, that is first useful, when a ROOPL++ program has been compiled to target code. Thus the algorithm must take a .PAL file and produce a .PAL file.


```

remove_redundancy(filepath fp):
    /* read is a function which takes a filepath
     * and returns a array of elements, where each
     * element is a line the the file. */

    file = read(fp)

    /* Replace patterns with simplifications */
    call memory_exchange(file)
    call special_if_statements(file)

    /* map(func1, l[]) applies func1
     * to every element of l */
    lines = map(parse_inst, file)

    lastRegister = ''
    lastInstruction = 0
    removeableLines []

    for i = 0 to length(lines)
        if lines[i][0] in {'ADDI', 'EXCHI'} then
            if lastRegister == lines[i][1] then
                lines[lastInstruction][2] += lines[i][2]
                insert(removeableLines, i)
            else
                lastRegister = lines[i][1]
                lastInstruction = i
        else
            lastRegister = ''
            lastInstruction = 0

    /* Remove all redundant instructions, update stack pointer */
    lines = removeLinesAdjustSP(lines, removeableLines)

    /* write is a function which takes an array
     * and writes a line for each element to a named file */

    write(lines, 'reduced.pal')

/* A method to divide an instruction
 * into opcode, arg0, arg1, arg2 */
parse_inst(instruction):
    return split(instruction, '_')

```

4 Results

Now we apply all our new tools to the target code, after we'll see the difference instructions and for test purpose, we'll write the values of nodeCount from 1..100 to a .txt file both for the program p which was created by the ROOPL++ compiler and the program p' which is the reduced program of p.

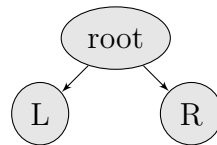
Program	PendVM step	main lines	start lines	Description
p	6572	134	50	Original program produced by the ROOPL++ compiler
p'	2102	50	42	A reduced version of p, where the algorithm remove redundancy has been applied
p''	1328	41	5	The example PAL program from the section of Pendulum Assembly Language: ForITo100.pal

On this simple program counting from 1..100, we have reduced the needed steps in PendVM to one third of the original need steps. Furthermore the number of lines have been reduced which should be reflected on the readability.

5 Data structures

5.1 Binary tree

A binary tree in computer science is a data structure, which is either empty or consist of a root node, where each node has atmost two children. You could see the binary tree as a tuple of the single node root and the two binary trees L & R.



Simple binary tree, with the a root node and two binary trees L & R as children to the root.

There are different types of binary tree all with their own properties. For example, A full binary tree is one where every node either have zero or two child-nodes. A perfect binary tree is one, where all interior nodes have two children and all leaves have none. A balanced binary tree is one where L & R of every node only differ in height with no more than one. [REF-Introduction to algorithms]

Properties of binary tree is for example maximum height, number of nodes, number of leafs and so on. Operations of different sort can be used of a binary tree some of them are:

Traversal: Search for a node

Insertion: Insert a new node

Deletion: Remove an existing node

6 Conclusion

7 Further work

7.0.1 Instruction operands - [OVERVEJ]

Machine code instructions vary, but in general each instruction itself will have fixed number of operands (0 to 3). Operands can have the following types:[6, p. 12]

register: These operands refer directly to the content of the CPU-register.

memory: These refer to data in memory. The address of the data may be a constant hardcoded into the instruction or may be computed using values of registers. Address are always offsets from the beginning of a segment.

immediate: These fixed values that are listed in the instruction itself. They are stored in the instruction itself (in code segment), not in the data segment.

implied: These operands are not explicitly shown. For example the increment instruction adds one to the register or memory. The one is implied.

7.0.2 Pendulum operation

Operation type					
R bit	op 6	rsd 5	rs 5	sh/rot 5	func 11
J bit	j/cf 6	Target 26			
B bit	j/b op 6	ra 5	rb 5	offset 16	
I bit	op 6	rsd 5	immediate 21		

Jump instructions other than j and all conditional branches uses B-type instruction encoding and specify two registers and an offset.

Immediate instructions, I-type, specify one register and a 21 bit signed or unsigned, depending on the opcode, immediate value.

Instruction

ADD

SUB

ADDI

AND

ANDI

BEQ

BGEZ

BGEZAL

BGTZ

BLEZ Branch on less than or equal to zero

BLTZ Branch on less than zero

BLTZAL Branch on less than than zero and link

BNE Branch on not equal

CF Come-from

7.1 Register values

Before entry

Register	Value
1	2462
2	0
3	2464
6	1
7	0
8	0
9	0
10	0
11	0
12	0

subtop_if_2:	BRA subbot_if_2
if_2:	SWAP \$2
	NEG \$2
	EXCH \$2 \$1
cmp_top_19:	BNE \$8 \$10 cmp_bot_20
	XORI \$11 1
cmp_bot_20:	BNE \$8 \$10 cmp_top_19
f_top_21:	BEQ \$11 \$0 f_bot_22
	XORI \$12 1
f_bot_22:	BEQ \$11 \$0 f_top_21
	XOR \$6 \$12
f_bot_22_i:	BEQ \$11 \$0 f_top_21_i
	XORI \$12 1
f_top_21_i:	BEQ \$11 \$0 f_bot_22_i
cmp_bot_20_i:	BNE \$8 \$10 cmp_top_19_i
	XORI \$11 1
cmp_top_19_i:	BNE \$8 \$10 cmp_bot_20_i
	EXCH \$2 \$1
subbot_if_2:	BRA subtop_if_2

And see what entry has become, simple 9 lines.

entry_11:	BEQ \$6 \$0 assert_13
	BRA get
	BRA if_1
	BRA get
	BRA get
	BRA get_lim
	BRA if_2
	BRA get_lim

7.2 Pendulum

, with exception of two constraints:

Memory access is always an exchange. The exchange instruction swaps register value with the value in memory at an address specified by another register.[4, p. 32]

Non-sequential instructions like jumps to another macro must have a return address. This is needed to keep the reversibility. [4, p. 36] Say an instruction is reached by one jump, then that instruction only needs to keep track a single destination. In case an instruction is reached from multiple other instructions, it needs to keep track of which instruction that made the jump.[4, p. 25]

8 Appendix A

ForITo100.pal

```

1  ;;; increment a from [1..n]
2  subtop:   BRA subbot
3  main:     SWAPBR $2           ; entry/exit point
4           NEG $2              ; negate offset to return caller
5           EXCH $2 $1          ; push return offset to stack
6           ADDI $30 100        ; a += n
7           BRA swap            ; set limit ($28) += a, set a = 0
8  looptop:  BNE $30 $0 loopbot ; from a = 0 do
9           ADDI $30 1          ; a += 1
10 loopbot:  BNE $30 $28 looptop; until a = limit loop body
11           SUB $28 $30        ; set limit = 0
12           ADDI $30 -100      ; a -= n
13           EXCH $2 $1        ; pop return address
14 subbot:   BRA subtop
15 ;; swap(int a, int b)
16 subtop_3: BRA subbot_3
17 swap:     SWAPBR $2
18           NEG $2
19           EXCH $2 $1
20           ADD $28 $30
21           SUB $30 $28
22           EXCH $2 $1
23 subbot_3:  BRA subtop_3
24 ;; increment (int a)
25 subtop_2:  BRA subbot_2
26 incr:     SWAPBR $2
27           NEG $2
28           EXCH $2 $1
29           ADDI $30 1
30           EXCH $2 $1
31 subbot_2:  BRA subtop_2
32 ;; write (int a, int mode)
33 subtop_1:  BRA subbot_1
34 write:     SWAPBR $2
35           NEG $2
36           EXCH $2 $1
37           OUT $30 $31
38           EXCH $2 $1
39 subbot_1:  BRA subtop_1

```

9 Appendix B

9.1 Full program

```
                                prg.pal
1  ;; pendulum pal file
2  top:                        BRA    start
3  l_r_nodeCount:             DATA  0
4  l_r_limit:                 DATA  0
5  l_Program_vt:              DATA  214
6  l_malloc_top:              BRA     l_malloc_bot
7  l_malloc:                  SWAPBR $2
8                              NEG     $2
9                              ADDI    $9 2
10                             XOR     $8 $0
11                             ADDI    $1 1
12                             EXCH    $6 $1
13                             ADDI    $1 1
14                             EXCH    $7 $1
15                             EXCH    $2 $1
16                             ADDI    $1 -1
17                             BRA     l_malloc1
18                             ADDI    $1 1
19                             EXCH    $2 $1
20                             EXCH    $7 $1
21                             ADDI    $1 -1
22                             EXCH    $6 $1
23                             ADDI    $1 -1
24                             XOR     $8 $0
25                             ADDI    $9 -2
26  l_malloc_bot:             BRA     l_malloc_top
27  l_malloc1_top:            BRA     l_malloc1_bot
28                             ADDI    $1 1
29                             EXCH    $2 $1
30                             SUB     $17 $8
31                             XOR     $17 $4
32  l_malloc1:                SWAPBR $2
33                             NEG     $2
34                             EXCH    $2 $1
35                             ADDI    $1 -1
36                             XOR     $17 $4
37                             ADD     $17 $8
38                             EXCH    $19 $17
39                             XOR     $18 $19
```

40		EXCH	\$19 \$17
41		XOR	\$13 \$9
42		SUB	\$13 \$7
43	cmp_top_1:	BGEZ	\$13 cmp_bot_2
44		XORI	\$14 1
45	cmp_bot_2:	BGEZ	\$13 cmp_top_1
46		XOR	\$10 \$14
47	cmp_bot_2_i:	BGEZ	\$13 cmp_top_1_i
48		XORI	\$14 1
49	cmp_top_1_i:	BGEZ	\$13 cmp_bot_2_i
50		ADD	\$13 \$7
51		XOR	\$13 \$9
52	l_o_test:	BEQ	\$10 \$0 l_o_test_false
53		XORI	\$10 1
54		ADDI	\$8 1
55		EXCH	\$19 \$17
56		XOR	\$18 \$19
57		EXCH	\$19 \$17
58		RL	\$9 1
59		EXCH	\$10 \$1
60		ADDI	\$1 -1
61		EXCH	\$11 \$1
62		ADDI	\$1 -1
63		EXCH	\$12 \$1
64		ADDI	\$1 -1
65		EXCH	\$14 \$1
66		ADDI	\$1 -1
67		EXCH	\$16 \$1
68		ADDI	\$1 -1
69		EXCH	\$17 \$1
70		ADDI	\$1 -1
71		EXCH	\$18 \$1
72		ADDI	\$1 -1
73		EXCH	\$20 \$1
74		ADDI	\$1 -1
75		EXCH	\$21 \$1
76		ADDI	\$1 -1
77		EXCH	\$22 \$1
78		ADDI	\$1 -1
79		EXCH	\$23 \$1
80		ADDI	\$1 -1
81		BRA	l_malloc1
82		ADDI	\$1 1
83		EXCH	\$23 \$1
84		ADDI	\$1 1

85		EXCH	\$22 \$1
86		ADDI	\$1 1
87		EXCH	\$21 \$1
88		ADDI	\$1 1
89		EXCH	\$20 \$1
90		ADDI	\$1 1
91		EXCH	\$18 \$1
92		ADDI	\$1 1
93		EXCH	\$17 \$1
94		ADDI	\$1 1
95		EXCH	\$16 \$1
96		ADDI	\$1 1
97		EXCH	\$14 \$1
98		ADDI	\$1 1
99		EXCH	\$12 \$1
100		ADDI	\$1 1
101		EXCH	\$11 \$1
102		ADDI	\$1 1
103		EXCH	\$10 \$1
104		RR	\$9 1
105		ADDI	\$8 -1
106		XORI	\$10 1
107	l_o_assert_true:	BRA	l_o_assert
108	l_o_test_false:	BRA	l_o_test
109	cmp_top_5:	BEQ	\$18 \$0 cmp_bot_6
110		XORI	\$20 1
111	cmp_bot_6:	BEQ	\$18 \$0 cmp_top_5
112		XOR	\$11 \$20
113	cmp_bot_6_i:	BEQ	\$18 \$0 cmp_top_5_i
114		XORI	\$20 1
115	cmp_top_5_i:	BEQ	\$18 \$0 cmp_bot_6_i
116	l_i_test:	BEQ	\$11 \$0 l_i_test_false
117		XORI	\$11 1
118		ADD	\$6 \$18
119		SUB	\$18 \$6
120		EXCH	\$12 \$6
121		EXCH	\$12 \$17
122		XOR	\$12 \$6
123		XORI	\$11 1
124	l_i_assert_true:	BRA	l_i_assert
125	l_i_test_false:	BRA	l_i_test
126		ADDI	\$8 1
127		RL	\$9 1
128		EXCH	\$10 \$1
129		ADDI	\$1 -1

130	EXCH	\$11 \$1
131	ADDI	\$1 -1
132	EXCH	\$12 \$1
133	ADDI	\$1 -1
134	EXCH	\$14 \$1
135	ADDI	\$1 -1
136	EXCH	\$16 \$1
137	ADDI	\$1 -1
138	EXCH	\$17 \$1
139	ADDI	\$1 -1
140	EXCH	\$18 \$1
141	ADDI	\$1 -1
142	EXCH	\$20 \$1
143	ADDI	\$1 -1
144	EXCH	\$21 \$1
145	ADDI	\$1 -1
146	EXCH	\$22 \$1
147	ADDI	\$1 -1
148	EXCH	\$23 \$1
149	ADDI	\$1 -1
150	BRA	l_malloc1
151	ADDI	\$1 1
152	EXCH	\$23 \$1
153	ADDI	\$1 1
154	EXCH	\$22 \$1
155	ADDI	\$1 1
156	EXCH	\$21 \$1
157	ADDI	\$1 1
158	EXCH	\$20 \$1
159	ADDI	\$1 1
160	EXCH	\$18 \$1
161	ADDI	\$1 1
162	EXCH	\$17 \$1
163	ADDI	\$1 1
164	EXCH	\$16 \$1
165	ADDI	\$1 1
166	EXCH	\$14 \$1
167	ADDI	\$1 1
168	EXCH	\$12 \$1
169	ADDI	\$1 1
170	EXCH	\$11 \$1
171	ADDI	\$1 1
172	EXCH	\$10 \$1
173	RR	\$9 1
174	ADDI	\$8 -1

175		XOR	\$12 \$6
176		EXCH	\$12 \$17
177		ADD	\$6 \$9
178	l_i_assert:	BNE	\$11 \$0 l_i_assert_true
179		EXCH	\$12 \$17
180		SUB	\$6 \$9
181	cmp_top_7:	BEQ	\$6 \$12 cmp_bot_8
182		XORI	\$21 1
183	cmp_bot_8:	BEQ	\$6 \$12 cmp_top_7
184	cmp_top_9:	BNE	\$12 \$0 cmp_bot_10
185		XORI	\$22 1
186	cmp_bot_10:	BNE	\$12 \$0 cmp_top_9
187		ORX	\$23 \$21 \$22
188		XOR	\$11 \$23
189		ORX	\$23 \$21 \$22
190	cmp_bot_10_i:	BNE	\$12 \$0 cmp_top_9_i
191		XORI	\$22 1
192	cmp_top_9_i:	BNE	\$12 \$0 cmp_bot_10_i
193	cmp_bot_8_i:	BEQ	\$6 \$12 cmp_top_7_i
194		XORI	\$21 1
195	cmp_top_7_i:	BEQ	\$6 \$12 cmp_bot_8_i
196		ADD	\$6 \$9
197		EXCH	\$12 \$17
198	l_o_assert:	BNE	\$10 \$0 l_o_assert_true
199		XOR	\$15 \$9
200		SUB	\$15 \$7
201	cmp_top_3:	BGEZ	\$15 cmp_bot_4
202		XORI	\$16 1
203	cmp_bot_4:	BGEZ	\$15 cmp_top_3
204		XOR	\$10 \$16
205	cmp_bot_4_i:	BGEZ	\$15 cmp_top_3_i
206		XORI	\$16 1
207	cmp_top_3_i:	BGEZ	\$15 cmp_bot_4_i
208		ADD	\$15 \$7
209		XOR	\$15 \$9
210	l_malloc1_bot:	BRA	l_malloc1_top
211	l_main_0_top:	BRA	l_main_0_bot
212		ADDI	\$1 1
213		EXCH	\$2 \$1
214		EXCH	\$3 \$1
215		ADDI	\$1 -1
216	l_main_0:	SWAPBR	\$2
217		NEG	\$2
218		ADDI	\$1 1
219		EXCH	\$3 \$1

220		EXCH	\$2	\$1	
221		ADDI	\$1	-1	
222		ADD	\$6	\$3	
223		ADDI	\$6	3	
224		EXCH	\$7	\$6	
225		ADDI	\$6	-3	
226		SUB	\$6	\$3	
227		XORI	\$8	100	
228		ADD	\$7	\$8	
229		XORI	\$8	100	
230		ADD	\$6	\$3	
231		ADDI	\$6	3	
232		EXCH	\$7	\$6	
233		ADDI	\$6	-3	
234		SUB	\$6	\$3	
235		XORI	\$6	1	
236	entry_11:	BEQ	\$6	\$0	assert_13
237		ADD	\$7	\$3	
238		ADDI	\$7	2	
239		EXCH	\$8	\$7	
240		ADDI	\$7	-2	
241		SUB	\$7	\$3	
242	cmp_top_15:	BNE	\$8	\$0	cmp_bot_16
243		XORI	\$9	1	
244	cmp_bot_16:	BNE	\$8	\$0	cmp_top_15
245	f_top_17:	BEQ	\$9	\$0	f_bot_18
246		XORI	\$10	1	
247	f_bot_18:	BEQ	\$9	\$0	f_top_17
248		XOR	\$6	\$10	
249	f_bot_18_i:	BEQ	\$9	\$0	f_top_17_i
250		XORI	\$10	1	
251	f_top_17_i:	BEQ	\$9	\$0	f_bot_18_i
252	cmp_bot_16_i:	BNE	\$8	\$0	cmp_top_15_i
253		XORI	\$9	1	
254	cmp_top_15_i:	BNE	\$8	\$0	cmp_bot_16_i
255		ADD	\$7	\$3	
256		ADDI	\$7	2	
257		EXCH	\$8	\$7	
258		ADDI	\$7	-2	
259		SUB	\$7	\$3	
260		ADD	\$7	\$3	
261		ADDI	\$7	2	
262		EXCH	\$8	\$7	
263		ADDI	\$7	-2	
264		SUB	\$7	\$3	

265		ADD	\$9 \$3
266		ADDI	\$9 3
267		EXCH	\$10 \$9
268		ADDI	\$9 -3
269		SUB	\$9 \$3
270	cmp_top_19:	BNE	\$8 \$10 cmp_bot_20
271		XORI	\$11 1
272	cmp_bot_20:	BNE	\$8 \$10 cmp_top_19
273	f_top_21:	BEQ	\$11 \$0 f_bot_22
274		XORI	\$12 1
275	f_bot_22:	BEQ	\$11 \$0 f_top_21
276		XOR	\$6 \$12
277	f_bot_22_i:	BEQ	\$11 \$0 f_top_21_i
278		XORI	\$12 1
279	f_top_21_i:	BEQ	\$11 \$0 f_bot_22_i
280	cmp_bot_20_i:	BNE	\$8 \$10 cmp_top_19_i
281		XORI	\$11 1
282	cmp_top_19_i:	BNE	\$8 \$10 cmp_bot_20_i
283		ADD	\$9 \$3
284		ADDI	\$9 3
285		EXCH	\$10 \$9
286		ADDI	\$9 -3
287		SUB	\$9 \$3
288		ADD	\$7 \$3
289		ADDI	\$7 2
290		EXCH	\$8 \$7
291		ADDI	\$7 -2
292		SUB	\$7 \$3
293	test_12:	BNE	\$6 \$0 exit_14
294		ADD	\$7 \$3
295		ADDI	\$7 2
296		EXCH	\$8 \$7
297		ADDI	\$7 -2
298		SUB	\$7 \$3
299		XORI	\$9 1
300		ADD	\$8 \$9
301		XORI	\$9 1
302		ADD	\$7 \$3
303		ADDI	\$7 2
304		EXCH	\$8 \$7
305		ADDI	\$7 -2
306		SUB	\$7 \$3
307	assert_13:	BRA	entry_11
308	exit_14:	BRA	test_12
309		XORI	\$6 1

310		ADD	\$6	\$3
311		ADDI	\$6	2
312		EXCH	\$7	\$6
313		ADDI	\$6	-2
314		SUB	\$6	\$3
315		ADD	\$8	\$3
316		ADDI	\$8	3
317		EXCH	\$9	\$8
318		ADDI	\$8	-3
319		SUB	\$8	\$3
320		SUB	\$7	\$9
321		ADD	\$8	\$3
322		ADDI	\$8	3
323		EXCH	\$9	\$8
324		ADDI	\$8	-3
325		SUB	\$8	\$3
326		ADD	\$6	\$3
327		ADDI	\$6	2
328		EXCH	\$7	\$6
329		ADDI	\$6	-2
330		SUB	\$6	\$3
331		ADD	\$6	\$3
332		ADDI	\$6	3
333		EXCH	\$7	\$6
334		ADDI	\$6	-3
335		SUB	\$6	\$3
336		XORI	\$8	100
337		SUB	\$7	\$8
338		XORI	\$8	100
339		ADD	\$6	\$3
340		ADDI	\$6	3
341		EXCH	\$7	\$6
342		ADDI	\$6	-3
343		SUB	\$6	\$3
344	l_main_0_bot:	BRA	l_main_0_top	
345	start:	BRA	top	
346		START		
347		ADDI	\$4	393
348		XOR	\$5	\$4
349		ADDI	\$5	10
350		XOR	\$7	\$5
351		ADDI	\$4	10
352		ADDI	\$4	-1
353		EXCH	\$7	\$4
354		ADDI	\$4	1

355	ADDI	\$4	-10
356	XOR	\$1	\$5
357	ADDI	\$1	2048
358	ADDI	\$1	-4
359	XOR	\$3	\$1
360	XORI	\$6	3
361	EXCH	\$6	\$3
362	ADDI	\$1	-1
363	EXCH	\$3	\$1
364	ADDI	\$1	-1
365	BRA	l_main_0	
366	ADDI	\$1	1
367	EXCH	\$3	\$1
368	ADDI	\$3	1
369	ADDI	\$3	1
370	EXCH	\$6	\$3
371	XORI	\$7	1
372	EXCH	\$6	\$7
373	XORI	\$7	1
374	ADDI	\$3	-1
375	ADDI	\$3	-1
376	ADDI	\$3	1
377	ADDI	\$3	2
378	EXCH	\$6	\$3
379	XORI	\$7	2
380	EXCH	\$6	\$7
381	XORI	\$7	2
382	ADDI	\$3	-2
383	ADDI	\$3	-1
384	ADDI	\$1	1
385	EXCH	\$6	\$3
386	XORI	\$6	3
387	XOR	\$3	\$1
388	ADDI	\$1	4
389	ADDI	\$1	-2048
390	XOR	\$1	\$5
391	ADDI	\$5	-10
392	XOR	\$5	\$4
393	ADDI	\$4	-393
394	finish :	FINISH	

9.1.1 Malloc

malloc

```

1  l_malloc_top:      BRA      l_malloc_bot
2  l_malloc:          SWAPBR $2
3                      NEG      $2
4                      ADDI     $9 2
5                      XOR      $8 $0
6                      ADDI     $1 1
7                      EXCH     $6 $1
8                      ADDI     $1 1
9                      EXCH     $7 $1
10                     EXCH     $2 $1
11                     ADDI     $1 -1
12                     BRA      l_malloc1
13                     ADDI     $1 1
14                     EXCH     $2 $1
15                     EXCH     $7 $1
16                     ADDI     $1 -1
17                     EXCH     $6 $1
18                     ADDI     $1 -1
19                     XOR      $8 $0
20                     ADDI     $9 -2
21  l_malloc_bot:     BRA      l_malloc_top

```

malloc1

```

1  l_malloc1_top:     BRA      l_malloc1_bot
2                      ADDI     $1 1
3                      EXCH     $2 $1
4                      SUB      $17 $8
5                      XOR      $17 $4
6  l_malloc1:         SWAPBR $2
7                      NEG      $2
8                      ADD      $15 $7
9                      XOR      $15 $9
10 l_malloc1_bot:     BRA      l_malloc1_top

```

References

- [1] <http://www.computinghistory.org.uk/det/32489/Kathleen-Booth/>
- [2] https://www.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.asma400/asmr102112.htm
- [3] https://en.wikipedia.org/wiki/Von_Neumann_architecture
- [4] Vieri, C. J. et al. *Pendulum: A Reversible Computer Architecture*. Master's Thesis. University of California at Berkeley 1993.
- [5] Michael P. Frank. *Reversibility for Efficient Computing*. Ph.D Thesis. University of Florida, 1999.
- [6] Carter, P. A. *PC Assembly Language*. 2006
- [7] Yokoyama, T. and Glück, R. *A reversible programming language and its invertible self-interpreter*. ACM, 2007.
- [8] C. H. Bennet. *Notes on the history of reversible computation*. IBM J. Res. Dev., 32(1), 1988
- [9] <https://github.com/TueHaulund/PendVM>
- [10] <https://plato.stanford.edu/entries/turing-machine/>
- [11] J. C. Maxwell. *Theory of Heat*, 4th Ed., Longmans, Green & Co., London, 1875 (1st Ed. 1871)