# Appendix B

# The Pendulum instruction set architecture (PISA)

This appendix gives a detailed description of the assembly language instruction set for the Pendulum reversible microprocessor currently being fabricated by Carlin Vieri [178]. Some reversible instruction set issues encountered during the development of this instruction set (which I assisted Vieri with) were discussed in §9. The version of the instruction set described here was the target for the compiler described in §9.4.3 and appendix D.

Vieri's thesis describes these instructions at a more detailed level, that gives the precise instruction word layout for purposes of machine code assembly and instruction decoding in his real hardware implementation. For our purposes of testing compiler techniques, such details were unimportant. Thus, in this reference we only describe the instructions from an assembly language programmer's point of view.

## B.1 Overall organization

The PISA instruction set can be divided into three categories: reversible artihmetic/logical operations, ordinary branch instructions, and special instructions.

The set of arithmetic/logical operations is designed to be logically complete, yet purely reversible. To achieve this, the results of operations are generally XOR'ed into separate destination registers, an operation which can be inverted by simply repeating it. Certain "non-expanding" operations can be performed reversibly without a separate destination register.

As for the branch instructions, these are designed to be used in pairs, where each branch instruction points to a corresponding branch that points back to the original instruction, as per the discussion in §9.2.2. Given this, one way to implement branches reversibly is to have the branch instruction add its offset into a special

```
Key:
rsd,rt = 5-bit register identifier.
  No-Op if rsd is same reg as rt.
imm,amt = 16 bit signed immediate
  [imm] = imm sign-extended to 32 bits

"Non-expanding" arith./logical operations:

Mnem.   Args.      Forwards behavior
-----   ---------  ---------------------
NEG     rsd        rsd = -rsd
ADD     rsd,rt     rsd += rt (mod 2^32)
ADDI    rsd,imm    rsd += [imm] (mod 2^32)
SUB     rsd,rt     rsd -= rt (mod 2^32)
XOR     rsd,rt     rsd ^= rt
XORI    rsd,imm    rsd ^= [imm]
RL      rsd,amt    rsd = rsd rol amt
RLV     rsd,rt     rsd = rsd rol rt
RR      rsd,amt    rsd = rsd ror amt
RRV     rsd,rt     rsd = rsd ror rt
```

Figure B.1: “Non-expanding” arithmetic/logical operations in the 32-bit simulator/compiler version of the PISA instruction set.

```
Key:
rd,rs,rt = 5-bit register identifier.
  No-Op if rd is same reg as rs or rt.
imm,amt = 16 bit signed immediate
  [imm] = imm sign-extended to 32 bits

"Expanding" arith./logical operations:

Mnem.   Args.      Forwards behavior
-----   ---------  ---------------------
ANDX    rd,rs,rt   rd ^= rs&rt
ANDIX   rd,rs,imm  rd ^= rs&[imm]
NORX    rd,rs,rt   rd ^= ~(rs|rt)
ORX     rd,rs,rt   rd ^= rs|rt
ORIX    rd,rs,imm  rd ^= rs|[imm]
SLLX    rd,rs,amt  rd ^= rs<<amt
SLLVX   rd,rs,rt   rd ^= rs<<rt
SLTX    rd,rs,rt   rd ^= (rs<rt)?1:0
SLTIX   rd,rs,imm  rd ^= (rs<imm)?1:0
SRAX    rd,rs,amt  rd ^= rs>>amt
SRAVX   rd,rs,rt   rd ^= rs>>rt
SRLX    rd,rs,amt  rd ^= (unsigned)rs>>amt
SRLVX   rd,rs,rt   rd ^= (unsigned)rs>>rt
```

Figure B.2: "Expanding" arithmetic/logical operations in the PISA instruction set.

```
Key:
rd,ra,rb = 5-bit register identifier.
off = 16 bit signed offset
loff = 26 bit signed offset
dir = +1/-1 bit where +1=forward, -1=reverse
BR = internal "branch register"

Branch instructions:

BEQ     ra,rb,off  if ra=rb, BR+=off*dir
BGEZ    rb,off     if rb>=0, BR+=off*dir
BGTZ    rb,off     if rb>0, BR+=off*dir
BLEZ    rb,off     if rb<=0, BR+=off*dir
BLTZ    rb,off     if rb<0, BR+=off*dir
BNE     ra,rb,off  if ra!=rb, BR+=off*dir
BRA     loff       BR+=loff*dir
RBRA    loff       dir=-dir, BR+=loff*dir
SWAPBR  r          r<->BR

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

Memory & I/O instructions:

EXCH    rd,ra      rd <-> mem[ra]
READ    ra         ra ^= next word from input str.
SHOW    ra         Copies ra to output stream.
EMIT    ra         Emit ra to garbage stream.
```

Figure B.3: Branching, memory access, and input/output operations in the PISA instruction set.

"branch register" which is normally zero. Between instructions, if the branch register is non-zero, the program counter increments by the branch register value, instead of by the normal 1 instruction. Then the branch at the destination executes, canceling out the value stored in the branch register and resuming normal execution.

In this scheme, even if the programmer forgets to put in the branch at the destination, the resulting behavior will still be reversible. But it will not be useful behavior: the program counter will jump forward through the program in repeated leaps, of size equal to the original offset.

To implement subroutine calls, the branch destination can be the special SWAPBR instruction, which exchanges the branch register with an empty register. The body of the subroutine negates the register, so when the subroutine hits the next SWAPBR it branches back to the location it came from; the branch at that location cancels out the branch register and the processor continues sequentially. SWAPBR can also be used in a complementary way to perform switch statements.

All memory access happens through the EXCH instruction which exchanges a register with a variable memory location. There is an interesting case here, in which an EXCH instruction tries to exchange *itself* with a memory location. The machine can be designed to do nothing in such a case. Or, if the instruction fetch/unfetch mechanism works via an exchange, the register will actually be exchanged with the single constantly-moving value that sits in the instruction register between instructions, and in the current PC location in memory during instructions.

The processor direction can be reversed in software using the special RBRA (reversing branch) instruction, which toggles the processor direction bit while it is performing BRA functionality. This allows subroutines to be called either forwards or in reverse, thus reducing the need for repeated code.

Special instructions to perform reversible output are also available—the SHOW instruction which just exports a copy of a register, and the EMIT instruction which actually reversibly sends the information in the register out of the processor to whatever system it is embedded in. Input instructions could also be defined, and there could be two types: a SEE instruction which just XOR's an input word into a register, but does not consume it (the external system would be responsible for disposing of the original) and a TAKE instruction which would reversibly consume the outside information and bring it into a register, would have to be initially clear. Another option would just be a single IOEX instruction which simply exchanges a register with the value currently present in the external I/O system, which could then move the old value to its output, and move a new value into place from its input.

Finally, there are START/FINISH instructions for marking the start/endpoints of programs when running in the simulation environment. Presumably, a real processor would always be running its operating system, and would never need to halt.

Let us now give all the instructions, in a reference format.

## B.2 List of Instructions

Figures B.1, B.2, and B.3 list the name, arguments, and forwards behavior of all the instructions in the 32-bit version of the PISA instruction set that was used in the Pendulum simulator and the R language compiler.

## B.3 Arithmetic/logical ops

---

### `ADD`

Add one register into another.

**Usage:** `ADD` $reg_d$ $reg_s$

**Arguments:**

$reg_d$ — The destination register.

$reg_s$ — The source register.

**Description:**

Adds the value of register $reg_s$ into register $reg_d$, that is, modifies $reg_d$ be equal to the previous value of $(reg_d + reg_s) \bmod 2^{32}$. Note that this operation is inherently reversible, no matter the previous values of $reg_d$ and $reg_s$. It is inverted by SUB with the same arguments.

---

### `ADDI`

Add an immediate value into a register.

**Usage:** `ADDI` $reg_d$ $imm$

**Arguments:**

$reg_d$ — The destination register.

$imm$ — The immediate value to be added into $reg_d$.

**Description:**

Sign-extends the immediate 16-bit value $imm$ to 32 bits and adds it into $reg_d$. That is, $reg_d \leftarrow (reg_d + imm) \bmod 2^{32}$. Note that this operation is inherently reversible, no matter the previous value of $reg_d$. It is inverted by another ADDI with the immediate value negated, or by doing NEG of $reg_d$ followed by the identical ADDI, followed by another NEG of $reg_d$.

---

### `ANDX`

Exclusive-OR the result of an AND into a register.

**Usage:** `ANDX` $reg_d$ $reg_{s1}$ $reg_{s2}$

**Arguments:**

$reg_d$ — The destination register.

$reg_{s1}$ — The first source operand.

$reg_{s2}$ — The second source operand.

**Description:**

Computes the bitwise logical AND of the contents of $reg_{s1}$ and $reg_{s2}$, and exclusive-OR's the result into register $reg_d$. That is, $reg_d \leftarrow reg_d \oplus (reg_{s1} \wedge reg_{s2})$. Note that due to the XOR, this operation is inherently reversible, no matter the previous values of the registers. It is inverted by repeating the exact same instruction again. Note also that plain AND may be emulated by letting $reg_d$ be a register that was previously 0. ANDX corresponds to 32 Toffoli gates operating in parallel on the corresponding bits of the 3 operands.

---

## `ANDIX`

XOR an AND with an immediate value into a register.

**Usage:** `ANDIX` $reg_d$ $reg_s$ $imm$

**Arguments:**

$reg_d$ — The destination register.

$reg_s$ — The source register.

$imm$ — The immediate value to AND with.

**Description:**

Like ANDX, except the source register is AND'ed with the immediate value $imm$ instead of with a second source register.

---

## `NORX`

Exclusive-OR the result of a NOR into a register.

**Usage:** `NORX` $reg_d$ $reg_{s1}$ $reg_{s2}$

**Arguments:**

$reg_d$ — The destination register.

$reg_{s1}$ — The first source operand.

$reg_{s2}$ — The second source operand.

**Description:**

Computes the bitwise logical NOR of the contents of $reg_{s1}$ and $reg_{s2}$, and exclusive-OR's the result into register $reg_d$. That is, $reg_d \leftarrow reg_d \oplus \overline{(reg_{s1} \vee reg_{s2})}$. Note that due to the XOR, this operation is inherently reversible, no matter the previous values of the registers. It is inverted by repeating the exact same instruction again.

For no particular reason, there are no corresponding NORIX, NANDX, or NAND-IX instructions. I believe this was just to keep the instruction set smaller. But NORX itself is not strictly necessary either, since one can emulate it by using ORX and then XORI'ing $-1$ into the result.

---

## `NEG`

Two's-complement negate the given register.

**Usage:** `NEG` $reg_{sd}$

**Arguments:**

$reg_{sd}$ — The source/destination register.

**Description:**

Replace the contents of $reg_{sd}$ with its (two's complement) negative. That is, $reg_{sd} \leftarrow (2^{32} - reg_{sd}) \bmod 2^{32}$. Note that this operation is inherently reversible. It is its own inverse.

---

## `ORX`

Exclusive-OR the result of an OR into a register.

**Usage:** `ORX` $reg_d$ $reg_{s1}$ $reg_{s2}$

**Arguments:**

$reg_d$ — The destination register.

$reg_{s1}$ — The first source operand.

$reg_{s2}$ — The second source operand.

**Description:**

Computes the bitwise logical OR of the contents of $reg_{s1}$ and $reg_{s2}$, and exclusive-OR's the result into register $reg_d$. That is, $reg_d \leftarrow reg_d \oplus (reg_{s1} \vee reg_{s2})$. Note that due to the XOR, this operation is inherently reversible, no matter the previous values of the registers. It is inverted by repeating the exact same instruction again. Note also that plain AND may be emulated by letting $reg_d$ be a register that was previously 0.

---

## `ORIX`

XOR an OR with an immediate value into a register.

**Usage:** `ORIX` $reg_d$ $reg_s$ *imm*

**Arguments:**

$reg_d$ — The destination register.

$reg_s$ — The source register.

*imm* — The immediate value to AND with.

**Description:**

Like ORX, except the source register is AND'ed with the immediate value *imm* instead of with a second source register.

---

## RL

Rotate a register left by a fixed number of bits.

**Usage:** `RL` $reg_{sd}$ *amt*

**Arguments:**

$reg_{sd}$ — The source/destination register.

*amt* — The immediate number of bits to rotate by.

**Description:**

Rotate the bits in register $reg_{sd}$ left (that is, in the direction from least-significant positions to most-significant positions) by the given number of places (0-31). Bits rotated off the left end of the word rotate back onto the right end. RL is inherently reversible; it is inverted by RR *amt*, or by RL'ing by the amount $(32 - amt)$ mod 32.

---

## RLV

Rotate a register left by a variable number of bits.

**Usage:** `RLV` $reg_{sd}$ $reg_t$

**Arguments:**

$reg_{sd}$ — The source/destination register.

$reg_t$ — The register giving the number of bits to rotate by.

**Description:**

Rotate the bits in register $reg_{sd}$ left (that is, in the direction from least-significant positions to most-significant positions) by the number of places given by $reg_t$ mod 32. Bits rotated off the left end of the word rotate back onto the right end. RLV is inherently reversible; it is inverted by RRV.

---

## RR

Rotate a register right by a fixed number of bits.

**Usage:** `RR` $reg_{sd}$ $amt$

**Arguments:**

$reg_{sd}$ — The source/destination register.

$amt$ — The immediate number of bits to rotate by.

**Description:**

Rotate the bits in register $reg_{sd}$ right (that is, in the direction from most-significant positions to least-significant positions) by the given number of places (0-31). Bits rotated off the right end of the word rotate back onto the left end. RR is inherently reversible; it is inverted by RL $amt$, or by RR'ing by the amount $(32 - amt)$ mod 32.

---

## RRV

Rotate a register right by a variable number of bits.

**Usage:** `RRV` $reg_{sd}$ $reg_t$

**Arguments:**

$reg_{sd}$ — The source/destination register.

$reg_t$ — The register giving the number of bits to rotate by.

**Description:**

Rotate the bits in register $reg_{sd}$ right (that is, in the direction from most-significant positions to least-significant positions) by the number of places given by $reg_t$ mod 32. Bits rotated off the right end of the word rotate back onto the left end. RRV is inherently reversible; it is inverted by RLV.

---

## SLLX

XOR with result of shifting a register left logically by a fixed number of bits.

**Usage:** `SLLX` $reg_d$ $reg_s$ $amt$

**Arguments:**

$reg_d$ — The destination register.

$reg_s$ — The source register.

$amt$ — The immediate number of bits to shift by.

**Description:**

Logically shifts the given register left by $amt$, filling in 0's on the right, and XOR's the result into the destination register $reg_d$. Note that since the shift operation is information-losing, we cannot put the result back into the same register. Instead, we XOR the result into a register as with NANDX and other operations.

---

## SLLVX

XOR with result of shifting a register left logically by a variable number of bits.

**Usage:** `SLLVX` $reg_d$ $reg_s$ $reg_t$

**Arguments:**

$reg_d$ — The destination register.

$reg_s$ — The source register.

$reg_t$ — Register specifying amount to shift by.

**Description:**

Like SLLX but with a variable number of bits. See RLV.

---

## SRAX

XOR with result of shifting a register right arithmetically by a fixed number of bits.

**Usage:** `SRAX` $reg_d$ $reg_s$ $amt$

**Arguments:**

$reg_d$ — The destination register.

$reg_s$ — The source register.

$amt$ — The immediate number of bits to shift by.

**Description:**

Arithmetically shifts the given register right by *amt*, filling in with copies of the leftmost bit, and XOR's the result into the destination register $reg_d$. Note that since the shift operation is information-losing, we cannot put the result back into the same register. Instead, we XOR the result into a register as with NANDX and other operations.

---

## SRAVX

XOR with result of shifting a register right arithmetically by a variable number of bits.

**Usage:** `SLLVX` $reg_d$ $reg_s$ $reg_t$

**Arguments:**

$reg_d$ — The destination register.

$reg_s$ — The source register.

$reg_t$ — Register specifying amount to shift by.

**Description:**

Like SRAX but with a variable number of bits. See RLV.

---

## SRLX

XOR with result of shifting a register right logically by a fixed number of bits.

**Usage:** `SRLX` $reg_d$ $reg_s$ *amt*

**Arguments:**

$reg_d$ — The destination register.

$reg_s$ — The source register.

*amt* — The immediate number of bits to shift by.

**Description:**

Logically shifts the given register right by *amt*, filling in 0's on the left, and XOR's the result into the destination register $reg_d$. Note that since the shift operation is information-losing, we cannot put the result back into the same register. Instead, we XOR the result into a register as with NANDX and other operations.

---

## SRLVX

XOR with result of shifting a register right logically by a variable number of bits.

**Usage:** `SRLVX` $reg_d$ $reg_s$ $reg_t$

**Arguments:**

$reg_d$ — The destination register.

$reg_s$ — The source register.

$reg_t$ — Register specifying amount to shift by.

**Description:**

Like SRLX but with a variable number of bits. See RLV.

---

## SUB

Subtract one register from another.

**Usage:** `SUB` $reg_d$ $reg_s$

**Arguments:**

$reg_d$ — The destination register.

$reg_s$ — The source register.

**Description:**

Subtracts the value of register $reg_s$ from register $reg_d$, that is, modifies $reg_d$ be equal to the previous value of $(reg_d - reg_s + 2^{32}) \bmod 2^{32}$. Note that this operation is inherently reversible, no matter the previous values of $reg_d$ and $reg_s$. It is inverted by ADD with the same arguments.

---

## XOR

Exclusive-OR one register into another.

**Usage:** `XOR` $reg_d$ $reg_s$

**Arguments:**

$reg_d$ — The destination register.

$reg_s$ — The source register.

**Description:**

Exclusive-OR $reg_s$ into $reg_d$, that is, sets $reg_d$ equal to $reg_d \oplus reg_s$. This is a self-reversible operation.

---

## XORI

Exclusive-OR an immediate value into a register.

**Usage:** `XORI` $reg_d$ $imm$

**Arguments:**

$reg_d$ — The destination register.

$imm$ — The immediate value to XOR with.

**Description:**

Exclusive-OR the 16-bit immediate value $imm$ into $reg_d$, that is, sets $reg_d$ equal to $reg_d \oplus imm$. Self-reversible.

## B.4 Ordinary branches

---

### BEQ

Branch if equal.

**Usage:** `BEQ` $reg_a$ $reg_b$ $off$

**Arguments:**

$reg_a$, $reg_b$ — Registers to compare.

$off$ — Immediate offset.

**Description:**

If the contents of registers $reg_a$ and $reg_b$ are equal, arrange to branch to an instruction $off$ steps ahead of the current instruction. That is, add the signed 16-bit value $off$ into the branch register. The destination must be a branch pointing back to the current location, or a SWAPBR.

---

### BGEZ

Branch if greater than or equal to zero.

**Usage:** `BGEZ` $reg_a$ $off$

**Arguments:**

$reg_a$ — Register to compare.

$off$ — Immediate offset.

**Description:**

If the value in register $reg_a$, interpreted in two's complement form, is greater than or equal to zero (that is, if its high bit is 0), then branch to an instruction $off$ steps ahead of the current instruction. That is, add the signed 16-bit value $off$ into the branch register. The destination must be a branch pointing back to the current location, or a SWAPBR.

---

## BGTZ

Branch if greater than zero.

**Usage:** `BGTZ` $reg_a$ $off$

**Arguments:**

$reg_a$ — Register to compare.

$off$ — Immediate offset.

**Description:**

If the value in register $reg_a$, interpreted in two's complement form, is greater than zero (that is, if it is not zero but its high bit is 0), then branch to an instruction $off$ steps ahead of the current instruction. That is, add the signed 16-bit value $off$ into the branch register. The destination must be a branch pointing back to the current location, or a SWAPBR.

---

## BLEZ

Branch if less than or equal to zero.

**Usage:** `BLEZ` $reg_a$ $off$

**Arguments:**

$reg_a$ — Register to compare.

$off$ — Immediate offset.

**Description:**

If the value in register $reg_a$, interpreted in two's complement form, is less than or equal to zero (that is, if it is zero or its high bit is 1), then branch to an instruction $off$ steps ahead of the current instruction. That is, add the signed 16-bit value $off$ into the branch register. The destination must be a branch pointing back to the current location, or a SWAPBR.

---

## BLTZ

Branch if less than zero.

**Usage:** `BLTZ` $reg_a$ $off$

**Arguments:**

$reg_a$ — Register to compare.

$off$ — Immediate offset.

**Description:**

If the value in register $reg_a$, interpreted in two's complement form, is less than zero (that is, if its high bit is 1), then branch to an instruction $off$ steps ahead of the current instruction. That is, add the signed 16-bit value $off$ into the branch register. The destination must be a branch pointing back to the current location, or a SWAPBR.

---

## BNE

Branch if not equal to zero.

**Usage:** `BNE` $reg_a$ $reg_b$ $off$

**Arguments:**

$reg_a$, $reg_b$ — Registers to compare.

$off$ — Immediate offset.

**Description:**

If the contents of registers $reg_a$ and $reg_b$ are equal, then branch to an instruction $off$ steps ahead of the current instruction. That is, add the signed 16-bit value $off$ into the branch register. The destination must be a branch pointing back to the current location, or a SWAPBR.

---

## BRA

Unconditional branch.

**Usage:** `BRA` $loff$

**Arguments:**

$loff$ — Long immediate offset.

**Description:**

Unconditionally branch to a location $loff$ steps ahead of the current instruction. That is, add the signed value $loff$ into the branch register. The value $loff$ is a signed long immediate value, with more bits than the short offsets in the various conditional branches above. The exact number of bits depends on the instruction encoding and the number of bits reserved for opcodes. The destination must be a branch pointing back to the current location, or a SWAPBR.

## B.5 Special instructions

---

### `EXCH`

Exchange a register with a memory cell.

**Usage:** `EXCH` $reg_d$ $reg_a$

**Arguments:**

$reg_d$ — The data register.

$reg_a$ — The address register.

**Description:**

Exchanges the contents of the given data register $reg_d$ with the contents of the RAM memory cell at the 32-bit address given in register $reg_a$. If the address given happens to be the address of the EXCH instruction being executed, the hardware may treat this as a special case, and for example, ignore the instruction. Whatever it does, it must be reversible, however.

---

### `SWAPBR`

Exchange register with branch register.

**Usage:** `SWAPBR` $reg$

**Arguments:**

$reg$ — Register to swap with the branch register.

**Description:**

Swap the contents of register $reg$ with the contents of the branch register. This instruction is useful at the entry/exit points of subroutines and switch statements.

---

### `RBRA`

Direction-reversing unconditional branch.

**Usage:** `RBRA` $loff$

**Arguments:**

$loff$ — Long immediate offset.

**Description:**

Like BRA, but also toggles the processor direction bit. After the branch is taken, the processor will proceed in the opposite direction from the one it was traversing originally. Useful for making reverse subroutine calls.

---

## READ

Copy information from input device.

**Usage:** `READ` $reg$

**Arguments:**

$reg$ — Register to read data into.

**Description:**

This instruction XORs the next word of data from the processor's canonical input device into register $reg$. In a multiprocessing architecture, this might be a means to receive information from the interprocessor communication network.

---

## SHOW

Copy information to output device.

**Usage:** `SHOW` $reg$

**Arguments:**

$reg$ — Register whose contents to show.

**Description:**

This instruction copies the information in register $reg$ and sends the copy to the processor's canonical output device. In the Pendulum simulator, this is used to echo data to the standard output stream, for viewing program output. In a multiprocessing architecture, this might be a means to send information into the interprocessor communication network.

---

## EMIT

Emit information from the processor.

**Usage:** `EMIT` *reg*

**Arguments:**

*reg* — Register whose contents to emit.

**Description:**

Like SHOW, but moves the data out of *reg*, instead of making a copy. This is presumed to represent the explicit, reversible removal of unwanted data from the processor to an entropy-removal mechanism. This mechanism might be a sub-processor that first compresses the garbage stream, then erases it using as little energy as possible. Or, it might be a mechanism for reversible, digitally transmitting the information to the edge of a multiprocessor mesh, and then dissipating it there. Or, it might be another mechanism for moving information out to an interprocessor communication network.

---

## START/FINISH

End-points for computation.

**Usage:** `START`
`FINISH`

**Arguments:**

**Description:**

These instructions merely mark the start and end of a program, for ease of simulation. They could be used in a real processor: FINISH could halt the processor, and START could halt if running in reverse. Note, however, that actually halting the processor is an irreversible event, since one has lost the information about how long ago the processor halted. One should be careful not to build a very large, dense reversible mesh processor that will explode when all the processors simultaneously reach the FINISH instruction, as a result of each processing node dissipating the $k_{\mathrm{B}}T \ln 2$ energy to clear the bit of information that tells it that it is still running.

One could fix this problem by having FINISH merely switch to a mode where the processor starts counting the number of time-steps since it halted. When the counter runs out of space, however, the processor must either erase information, or start running again.