Chapter 9

Design and programming of reversible processors

In the previous two chapters we reviewed a variety of computing technologies, ranging from the standard VLSI technology of today to visionary nanotechnologies that we cannot yet manufacture, that are all fundamentally capable of operating reversibly in a time-proportionate way that facilitates the central scaling advantages that we discussed in chapter 6. As technology improves, the potential advantage from reversibility becomes increasingly great.

However, independently of the precise hardware, we can ask: How should a reversible machine be programmed, so as to realize these potential benefits? In this chapter, we begin to answer this question, by describing our experience illustrating that it is actually quite straightforward to design reversible microprocessor instruction sets and reversible programming languages that allow the inherent asymptotic cost-efficiency of the underlying logic hardware to be preserved at higher levels. We describe the assembly language, high-level language, and compiler for the reversible processor designed in our group. We also give examples of reversible programs and algorithms, including a constant-space, linear-time simulation of a reversible physical system.

The overall message of this chapter is that once some basic concepts of reversible computing are understood, programming reversible hardware need not be significantly more difficult than programming normal machines. In this chapter we primarily focus on serial programming, but we close with some thoughts on the need for fundamentally new kinds of abstraction and programming techniques for expressing parallel "physical" algorithms.

9.1 Context of this work

First we briefly review the historical context of the present reversible systems design effort.

9.1.1 Previous reversible architectures

The first reversible computer architectures that we know of were designed by Barton (1978, [11]) and Ressler (1981, [140]) as thesis projects at MIT. These designs were based on the conservative (reversible and 1-bit-conserving) logic model developed by Fredkin and Toffoli (*cf.* [74]). The "conservative" aspect of the model actually seems rather irrelevant to efficiency issues, since both conservative and nonconservative logic systems can simulate each other with only small constant-factor overheads.

Several years later (1994), Hall [81] described a reversible instruction set architecture based on his "retractile cascade" reversible circuit style and the PDP-10 instruction set.

9.1.2 Pendulum architecture

In 1995, Vieri [179] developed the first version of the Pendulum architecture. Pendulum was unique in that it was the first reversible computing architecture designed for implementation in a real reversible silicon technology (SCRL, see §7.5). For conceptual simplicity and ease of implementation, it was a RISC (Reduced Instruction Set Computing) style architecture, in contrast to the CISC basis of Hall's architecture.

The original version of the Pendulum instruction set archietcture (PISA) had a few drawbacks. There was a lack of software control over garbage data, which meant that the architecture was not capable of realizing the full potential asymptotic cost-efficiency afforded by SCRL. (For example, one could not efficiently run Bennett's 1989 algorithm [19] or reversible physical simulations on it.) Also, the instruction set did not guarantee full reversibility independently of program correctness, which precluded some of the possible alternative applications for reversibility, such as bi-directional debugging (see ch. 10).

In our work we therefore studied several improved variations of PISA (*cf.* [64, 67]), which were used in our prototype reversible processor [65] and in the compiler design effort. Yet another improved and simplified version of PISA is being implemented now by Vieri for his Ph.D. dissertation research [178]. Since reversible architectures still only exist for purposes of isolated academic research, there has not yet been much need to standardize on a particular version of the instruction set. This would be easy enough to do at a later time, if and when more widespread interest develops.

Appendix B lists the version of PISA that we used for developing our reversible high-level language and compiler, which we will discuss later, in §9.4.

9.2 Reversible instruction set architectures

We now delve into some of the important issues in the design of reversible instruction set architectures in a bit more detail.

9.2.1 Asymptotic efficiency

One important desideratum for a reversible instruction set architecture is that it should be possible to write programs for it that perform tasks with the same *asymptotic* efficiency that could be achieved by a custom reversible circuit for that task. That is, the architecture should not hide the underlying efficiency of the circuit model. A sufficient condition for this is if a program for the processor can efficiently simulate a model of the hardware.

Of course, a single serial von Neumann style processor cannot be expected to be asymptotically as efficient as an arbitrarily-large parallel circuit. So in order to judge such an processor fairly, we imagine that it just represents a single node in an arbitrarily-large mesh of such processors, and ask whether the resulting mesh can simulate arbitrary circuits efficiently.

One consequence of this criterion is that the architecture should permit running arbitrarily-long reversible physical simulations with only *constant* space usage. This immediately rules out instruction set architectures that provide reversibility by constantly pushing garbage data onto an ever-growing stack. In such architectures, space usage increases asymptotically with time, and so cannot be bounded by a constant in an arbitrarily long simulation.

For instance, the first version of the Pendulum architecture [179] had this problem, since all branches and many data operations pushed information onto a "garbage stack" which could not be uncomputed except by reversing the entire processor. So one could not write a constant-space loop, for example.

Later versions of Pendulum avoided this problem by allowing garbage data to be uncomputed in software, and by using paired branches to avoid the generation of garbage during control flow operations. This type of branching is an important concept in reversible instruction sets and deserves further discussion.

9.2.2 Use of paired branches

The control flow instructions such as branches and jumps in normal architecures are generally not reversible, because after branching to a location, there is in general no way of telling which of many possible locations one might have branched from.

Branch stacks. One way to make branches reversible would be to treat every branch like a subroutine call, in that the address branched from (the old value of the program counter, or PC) becomes pushed onto a special stack, along with the value of a branch counter that has been keeping time since the previous branch. In reverse, when the branch counter reaches zero, one pops the previous PC and branch counter values from the stack, which undoes the branch.

Vieri's original Pendulum architecture took this approach, but avoided the need for the branch counter by placing special "come-from" instructions at branch destinations. These would trigger the popping of the old PC value when encountered when running in reverse.

Unfortunately, though it is simple, the branch stack approach is not asymptotically efficient because of the potentially large size of the stack of branch information that must be maintained. A loop that executes N times would require $\mathbf{\Theta}(N)$ space with this approach, even if it only explicitly manipulates a constant number $\mathbf{\Theta}(1)$ of variables.

A better approach. To solve this problem, and avoid generating extra garbage data on every branch instruction, one can take the approach of using *paired branches*. That is, the destination of each branch instruction should be another branch instruction, which refers back to the original instruction and takes care of absorbing the old PC value, or performing the backwards branch when running in reverse. The resulting control flow constructs are completely time-symmetric.

Control flow structures. Any of the usual high-level patterns of structured control flow can be implemented using paired branches. Figure 9.1 schematically illustrates some examples. Detailed examples of some of these can be seen in the example program in §9.3, p. 232.

If/then statement. A reversible implementation of a plain conditional statement (if/then) requires two branch instructions, one at the beginning and one at the end of the body of the conditional code. Before the first branch, the condition being tested is computed, and the desination address is loaded. Then if the condition fails, the branch instruction changes the machine state so that before the next instruction, the PC will be updated to point to the branch at the end of the IF body. For example, the PC could be swapped with a register holding the destination address. Then when the second branch executes (testing the same condition), it toggles the mode back to normal sequential operation, and the code after the second branch uncomputes the condition and unloads the address of the first branch.

There are a variety of straightforward mechanisms which will work for low-level implementation of paired branch functionality like this. We describe one in appendix B.

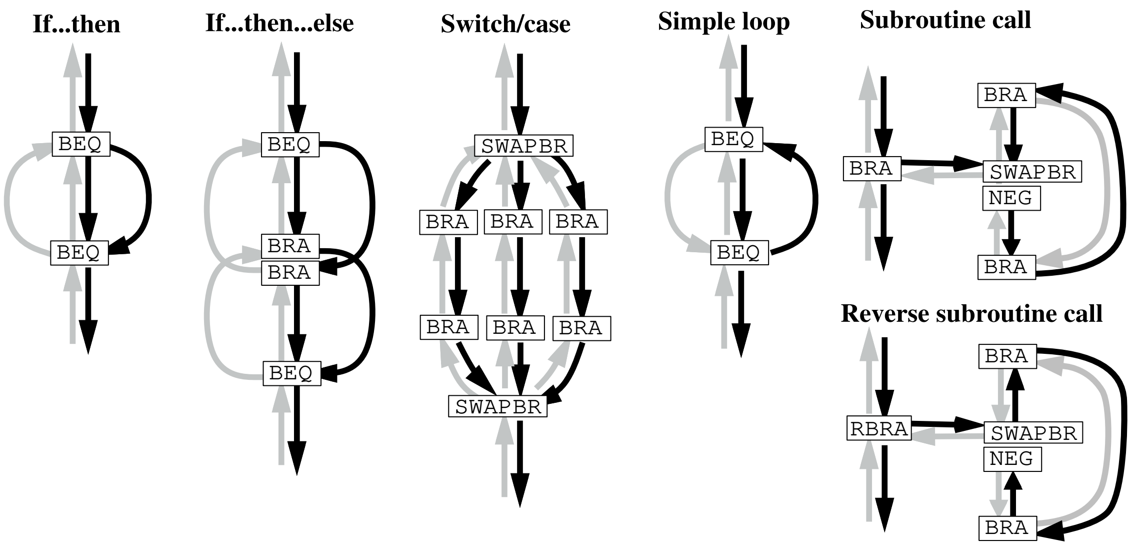

Figure 9.1: Abstract schematics of some reversible control-flow structures, using the PISA instruction set of Appendix B. Vertical arrows represent sequential flow of control through the program, other arrows represent non-local flow of control. Gray arrows represent the flow of control when the code is run in reverse.

If/then/else. Like if/then, but each of the two conditional branches is paired with an unconditional branch which serves to separate the two alternative paths through the construct. See the second diagram in figure 9.1.

Switch/case statement. The destination address is computed based on data, and we branch to it using a branch instruction that gets its destination from a register. The entry point is a literal branch that refer to the dispatching instruction to receive control from it. At the end of the case body we branch to a literal point in the outer context at which the address of the end of the case body is uncomputed based on the same data that was used to compute the start address.

Simple loop. A simple loop (such as a FOR loop) has the same structure as an IF except that the branches at the start and end are activated at different times. The first branch tests a loop-entry condition, and the second a loop-exit condition—these may or may not be identical. When we first hit the initial branch, the loop-entry condition succeeds and allows us into the body of the loop (the branch is not activated). When we hit the trailing branch, it should activate if we are to perform another loop iteration. We go back up to the first branch, which should now also be activated, to receive us. The process repeats until the final branch condition fails (loop exit condition succeeds) and we fall out of the loop.

A WHILE or UNTIL loop can also be implemented this way, so long as there is some piece of information maintained within the loop that is sufficient to tell when the loop was first entered; this information controls entry to the loop. For example, it suffices to keep a count of the number of times through the loop; the loop is only entered (initial branch deactivated) when this is zero. One must remember though that in that case, the count remains around after the loop is completed.

More complex loops with alternative entry/exit points can be constructed, but they require matching code on the outside of the loop that knows how to dispatch or absorb control to/from the proper points inside the loop.

Subroutine call. There are several ways to implement this; one of the simplest is the structure shown in the figure. The entry/exit points of the subroutine are at the *same* address; the body of the subroutine can loop around so as to exit from the same point where it was entered. The call instruction refers to the entry/exit point. We branch to it, for example by loading up a special branch register with the offset. The body of the subroutine saves away the branch register, using a free register and/or a stack. The offset is negated, and then used again for the branch at the subroutine's return. We branch back to the call instruction, which re-absorbs the offset. This is the approach that was used in most of the recent variants of the Pendulum design.

An alternative mechanism is to have separate entry and exit points at the beginning and end of the subroutine. The code before the call loads up the address of the start of the subroutine into a register. We branch to it (swapping PC and register, for example). The body of the subroutine saves away the address of the branch instruction that we came from. This same address is then used for the branch at the subroutine's return. We branch back to the entry point, which activates and receives the address of the subroutine's end point back into a register. Then the code after the call uncomputes the address of the end-point.

One very nice feature to have in the subroutine call instruction is a way to reverse the processor direction when entering and leaving the subroutine. This way the same subroutine can be used to either compute or uncompute some result, depending on which direction it is called in. This may reduce program size by up to a factor of two, compared to the alternative approach of maintaining two separate versions (one forward and one backward) of every subroutine whose results may need uncomputing.

Summary of paired branches: All the standard structured control-flow constructs are straightforward to implement using paired branches. This approach helps to minimize the generation of garbage data as code is executed.

9.2.3 Reversible logic/arithmetic operations

Many standard machine data instructions are already reversible. Fixed-length integer addition and subtraction, logical and two's complement negation, bit rotation, and exclusive-OR'ing one register into another are all examples. However, other operations, such as ANDing one register into another, normally lose information. There are several means for allowing such operations to be performed reversibly. One is to save on an internal machine stack the word that would otherwise be demolished. But this would not give the programmer the opportunity to uncompute that word later when it is no longer needed. A better approach is to require that operations like AND should write their result into a third register, distinct from the other two inputs, which is either required to be previously clear, or else the result is XOR'ed into it, or added into it, for example. Then a matching instruction should be provided that can uncompute the result. (In the case of XOR'ing the result into the destination register, the uncomputing instruction is just the same instruction over again.)

9.2.4 Data transfer operations

Similarly, operations that move values around, such as between registers and other registers or memory, must either (1) require that the destination be initially clear, (2) XOR/ADD the source into the destination, or (3) swap the source and destination. A swap can be implemented with 3 XORs, or 2 XORs if one location is initially clear.

9.2.5 Hardware-guaranteed reversibility

An important issue to decide about a reversible architecture is: Does it guarantee full reversibility of operation at the hardware level, independently of program correctness? If the hardware does not guarantee reversibility, then there is the risk that an incorrect program could inadvertently cause dissipation, and burn up the hardware.

One could imagine instead guaranteeing reversibility at the software level, in a compiler, but there are some problems with that approach. First, if the compiler allows asymptotically efficient reversible programs, then it can only guarantee reversibility by compiling programs to a set of guaranteed-reversible pseudocode primitives, which then might as well have been implemented directly by the hardware.

To understand why it is hard to introduce a guarantee of reversibility above the instruction set level, consider for example an instruction set whose expanding logic instructions (*e.g.*, NAND) do not guarantee reversibility unless the destination register is initially empty. So, in order to guarantee reversibility, the compiler has to guarantee that the program never accidentally does a NAND into a destination register that contains a value that is possibly non-zero.

Now, suppose one were to write a section of code that starts with some initially empty memory, and then uses that memory as scratch storage for performing some complex computation. One may know how to prove mathematically that after the code segment is finished, the memory it worked with will all have been restored to zero.

For example, suppose I first move to the scratch storage two primes a, b ($a < b$) that I wish to multiply. I generate their product ab and put it elsewhere, then I run a factoring algorithm to compute a, b given ab, and thereby subtract or XOR out a, b from the scratch storage, leaving me with just ab. Since I have proven that my factoring algorithm is correct, I know that the locations that held a, b are now empty, and I can go on to use them for some other computation. The emptiness of the location is an invariant that I can prove is maintained by my code section.

On the other hand, unless the compiler is required to be able to find proofs of such invariants, or the user is required to supply them, the compiler cannot assume that after my user-defined manipulation is completed, the locations are really empty. Therefore, the compiler will have to consider those locations to contain indestructible garbage, and it will have to allocate new memory for use in future operations. If the program involves a long sequence of manipulations like this, it will not be as space-efficient, asymptotically, as it could have been if the compiler was not responsible for guaranteeing reversibility.

9.3 Simple example PISA program: Multipication algorithm

As a simple example of reversible programming techniques, figure 9.2 shows a simple hand-coded multiplication subroutine for one 32-bit version of PISA. See appendix B for detailed specifications of individual instructions. The registers used in the routine are documented in table 9.1. Let us go through this routine line-by-line, to explain its operation. It is based on the simple grade-school multiplication algorithm, in which we just march through the digits of one multiplicand, multiplying them individually by the other multiplicand, and adding up the partial products, shifted appropriately, to form the complete product.

Line 1: `subtop:  BRA subbot` This is the first of a pair of labeled unconditional branches, pointing to each other, that frame the entire subroutine. These permit the subroutine to exit from the same point as where it is entered. They also have the side effect that if the flow of control encounters the subroutine sequentially, it will just skip over it.

Line 2: `mult:  SWAPBR R2` This is a conventional subroutine entry/exit point. On entry, the branch register is saved away into register R2, which is reserved for this purpose. On exit, R2 is swapped back into the branch register, causing control to be transfered back to outside the subroutine.

```
;; Label     Instr  Args            ; Pseudocode description
;; --------  ------ ----            ; ----------------------
   subtop:   BRA    subbot          ; MULT top.
   mult:     SWAPBR R2              ; Subroutine entry/exit point.
             NEG    R2              ; Negate offset to return to caller.
             EXCH   R2 R1           ; Push return offset to stack.
             BRA    alloc4          ; Allocate 4 empty registers (R28-R31).
             ADDI   R31 32          ; limit <- 32
             ADDI   R2 1            ; mask <- 1
   looptop:  BNE    R30 R0 loopbot  ; unless (position != 0) do
             ANDX   R28 R3 R2       ;     bit <- m1&mask
   iftop:    BEQ    R28 R0 ifbot    ;     if (bit != 0) then
             SLLVX  R29 R4 R30      ;        shifted <- m2<<position
             ADD    R5  R29         ;        product += shifted
             SLLVX  R29 R4 R30      ;        shifted -> m2<<position
   ifbot:    BEQ    R28 R0 iftop    ;     end if
             ANDX   R28 R3 R2       ;     bit -> m1&mask
             RL     R2 1            ;     mask <=< 1 (rotate left by 1)
             ADDI   R30 1           ;     position++
   loopbot:  BNE    R30 R31 looptop ; and repeat while (position != limit).
             SUB    R30 R31         ; position -> limit
             ADDI   R2 -1           ; mask -> 1
             ADDI   R31 -32         ; limit -> 32
             RBRA   alloc4          ; Deallocate 4 registers (R28-R31).
             EXCH   R2 R1           ; Pop return address.
   subbot:   BRA    subtop          ; MULT bottom.

   alloctop: BRA    allocbot
   alloc4:   SWAPBR R2          ; This sub-subroutine frees
             NEG    R2          ; 4 registers for use in the
             ADDI   R1  1       ; MULT subroutine.  It leaves
             EXCH   R31 R1      ; the stack pointer pushed
             ADDI   R1  1       ; above, but we don't mind.
             EXCH   R30 R1
             ADDI   R1  1
             EXCH   R29 R1
             ADDI   R1  1
             EXCH   R28 R1
   allocbot: BRA    alloctop
```

Figure 9.2: Hand-coded reversible assembly-language multiplication routine. The registers used are documented in table 9.1.

Register	Variable name	Purpose
R0	ZERO	Constant zero.
R1	SP	Stack pointer.
R2	SRO	Subroutine return offset.
”	mask	Bit in some position 0–31.
R3	m1	Arg 1: First multiplicand.
R4	m2	Arg 2: Second multiplicand.
R5	product	Arg 3: Product accumulator.
R28	bit	A single bit of m1, in place.
R29	shifted	bit, shifted to proper position.
R30	position	Index of current bit position.
R31	limit	Bit position limit (32).

Table 9.1: Registers used in the MULT routine shown in figure 9.2.

Line 3: `NEG R2` This negates the subroutine return offset so that when we exit the subroutine we will take exactly the opposite offset of the one that got us into the subroutine, so that we will return to exactly the point where we were called from.

Line 4: `EXCH R2 R1` `R1` is by convention the stack pointer. This instruction pushes `R2` onto the (presumed empty) top-of-stack location, so that `R2` will be available for use in calling further subroutines. (And also to be a temporary variable.)

Line 5: `BRA alloc4` This is a call to the subroutine `alloc4` (see lines 25–36) which simply pushes the upper four registers onto the stack, so we may safely use them for holding temporary values. (This is a “callee saves” register saving convention.) Routines such as `alloc4` may be shared by many subroutines.

Line 6–7: Here we just initialize a couple of registers. `limit` is just a constant 32 for use in the loop termination condition. `mask` is a bit, initially at position 0.

Line 8: `fortop:  BNE R30 R0 forbot` This is the loop entry condition. The loop is entered if the `position` variable is zero, which initially it is. Otherwise the loop would be skipped over.

Line 9: `ANDX R28 R3 R2` Simply extracts the desired bit from the first multiplicand.

Line 10: `BEQ R28 R0 ifbot` Skips the IF body if the extracted bit was zero.

Line 11: `SLLVX R29 R4 R30` Shifts the second multiplicand by an amount corresponding to which bit of the first multiplicand we are currently multiplying by.

Line 12: `ADD R5 R29` Add the appropriately-shifted second multiplicand into the accumulating product.

Line 13: Undo line 11 to clear register `R29`.

Line 14: Absorbs the transfer of control if the IF body was skipped.

Line 15: Undo line 9 to clear register `R28`.

Line 16: `RL R2 1` Shift the bit-mask left to the next position.

Line 17: `ADDI R30 1` Increment the position index.

Line 18: `loopbot:  BNE R30 R31 looptop` Loop exit condition. If we're not yet at the position limit, then branch back to the loop top.

Line 19: `SUB R30 R31` `position` is now equal to `limit`, so subtract `limit` out of it to restore it to zero.

Lines 20–21: Uncompute the constants that we set up in lines 6–7. `mask` is 1 because it has rotated from position 0 by 32 positions, back to position 0.

Line 22: `RBRA alloc4` Reverse-call the register-allocation subroutine, to restore the caller's registers back off the stack.

Line 23: `EXCH R2 R1` Pop return address back off the stack.

Line 24: `subbot:  BRA subtop` Subroutine bottom: branch back up to `subtop`, to get back to the entry/exit point.

Line 25–36: Register allocation subroutine. Alternates between incrementing the stack pointer, and exchanging a register we want to use with the current stack location. Effectively, pushes those registers onto the stack. They can be restored from the stack later by calling the subroutine in reverse.

9.3.1 Discussion

The routine illustrates several of the general reversible programming techniques we discussed in section 9.2. In particular, note the following points:

- Subroutine calls are implemented using save/restore of the branch-register offset, and a single subroutine entry/exit point.
- Any registers we use to hold temporary values are always restored to zero when we are finished with them.
- The subroutine works by accumulating the desired result in one of its arguments. This product can later be uncomputed by simply calling the subroutine again in reverse (using RBRA).
- Similarly, the auxilliary routine `alloc4` is called in reverse at the end of the MULT routine, in order to undo its earlier effects.
- An IF functionality is implemented via a matching pair of branches.
- A looping functionality is implemented using a pair of branches that determine the entry and exit conditions for the loop.
- Note that although the routine uses order n repetitions of the inner loop (where $n = 32$ is the word length), it only uses a constant amount of temporary storage, just as an irreversible version would. This is a good example of an algorithm that requires asymptotically no more space or time to do reversibly than irreversibly.

This concludes our discussion of reversible instruction sets. Many variations on the above theme are of course possible, but the above discussion should address many of the common underlying issues. More details would of course be necessary to support features such as floating-point arithmetic, arithmetic overflows, and asynchronous interrupts. But we believe that most of these features will be similarly straightforward to implement reversibly.

9.4 Reversible programming languages

9.4.1 General issues

If we are seriously considering the implications of building a reversible computer, then naturally we will want to investigate the possibility of programming that computer in a high-level language, rather than directly in machine code.

One approach to high-level programmability would allow programs to be written in a standard, irreversible programming language (for example, C) and then provide an interpreter or translator that allows them to be run on a reversible architecture. This would be straightforward, given the known general algorithms for simulation of irreversible machines on reversible ones (see §3.3).

However, this approach incurs a cost in terms of asymptotic inefficiency. As we saw in chapter 3, general-purpose reversible simulations are expected to require increased asymptotic time or space. However, for a particular problem, there may be an alternative reversible algorithm that is either just as asymptotically efficient as the original irreversible algorithm, or is only insignificantly less efficient. But an asymptotically good reversible algorithm for a problem cannot in general be expected to be a straightforward translation of the best irreversible algorithm. It may require a completely different structure. (For an example, see §9.5.5.) The programmer's ability to write asymptotically well-performing programs for the machine will in general be crippled if its underlying reversibility is hidden from him/her. Since all the known general-purpose simulation techniques incur at least polynomial asymptotic overheads, writing efficient reversible programs requires exposing a universal set of reversible constructs that incur *no* hidden asymptotic overheads, so that the programmer can explicitly manipulate information in a way that constitutes an asymptotically good reversible algorithm for the problem at hand, an algorithm that no automatic "reversibilizing" system could be expected to have discovered.

This leads to an approach wherein the input is allowed to be in a general irreversible language, but if a given program only uses a certain reversible subset of that language's constructs, then that program will be compiled in such a way that it incurs no asymptotic overheads, compared to what the programmer could have written if he were hand-coding in assembly language.

For example, if one is coding in C, but if assignment statements are eschewed in favor of reversible mutation statements such as `+=`, and if all local variables are asserted (see the Unix `assert(3)` manual page) to be restored to zero before function return, and if other `assert()`s are used to inform the compiler of loop entry conditions, and if one avoids frequent use of dynamic memory allocation (because garbage collection is irreversible; see [8, 7]), and overall if one's programs are written essentially in a style that looks basically like assembly language augmented with named variables, nested expressions, and structured control-flow, then it should be possible to compile the resulting programs to efficient reversible machine code without a need for an ever-growing garbage stack, or other asymptotically inefficient run-time support mechanisms.

Coding up a complete PISA-targeted compiler system for even a fairly simple conventional programming language was deemed to be too time-consuming a goal to fit within the scope of the present research project, especially given that there is no commercial interest yet in reversible machines, and also that no lessons of even much academic interest would be expected to be learned from such an exercise.

However, it was deemed feasible and useful to write a simple compiler for an extremely simple toy programming language similar to a reversible subset of C. In reference to the 1-letter programming language naming convention, we called our

```
(defsub mult (m1 m2 prod)
   ;; Use grade-school algorithm:
   (for pos = 0 to 31                ; For each of the 32 bit-positions,
      (if (m1 & (1 << pos)) then     ;   if that bit of m1 is 1, then
         (prod += (m2 << pos)))))    ;     add m2, shifted over to that
                                     ;       position, into prod.
```

Figure 9.3: A simple, efficient multiplication routine in the R language. This is essentially the same algorithm as that used in the hand-coded assembly-language routine in figure 9.2 (p. 233). Note that the high-level code is much more concise. The compiler (see §9.4.3) converts this routine into a sequence of 66 assembly code instructions: not quite as concise as our 36-instruction hand-coded routine, but reasonable.

---

language "R." [1]

9.4.2 "R," a reversible language

The essence of R is a very simple procedural C-like language based on machine-supported fixed-precision integer arithmetic, with nested expressions, arrays, efficient control flow statements, and with a Lisp-like (parenthesis-based) syntax for ease of parsing. The user-level constructs in the current version of R are documented in appendix C. To quickly illustrate what R code looks like, figure 9.3 shows a simple multiplication subroutine.

R is not actually the first reversible high-level language. Recently, we learned that around 1982, Chris Lutz and Howard Derby created a reversible programming language called "Janus" for a class at CalTech. (Our source is a letter [114] from Lutz to Rolf Landauer, describing the language.) It turns out that Janus's feature set is very similar to R's, which is interesting given that the two languages were developed entirely independently. However, Janus ran only under SIMULA on a DECSYSTEM-20, and it may no longer exist anywhere in usable form.

Also, Henry Baker described a reversible Lisp-like language called "Ψ-lisp" in a 1992 paper [8]. Ψ-lisp was based on so-called "linear" functional languages, *cf.* [7], which have the additional restriction that references must be conserved; there is always exactly one pointer to any given storage cell. Ψ-lisp is theoretically interesting, but it is not clear to us whether it constitutes an asymptotically efficient programming language. Also, the reasons for and implications of linearity seem to us to be mostly

[1] After naming our R language, we learned there is a statistics package called "R": see `http://www.ci.tuwien.ac.at/R/contents.html`. So if our R ever hits the big-time, we may want to rename it "RL", or something, to avoid confusion.

orthogonal to the reasons for and implications of reversibility.

9.4.3 The R compiler

In addition to specifying the R language, I also wrote a simple compiler for translating R programs into assembly code executable on a certain version of the Pendulum architecture. The main points of this exercise were (1) to demonstrate that the R language, as envisioned, is easy to implement, and (2) to provide a convenient way to create substantially-sized test programs for the Pendulum architecture.

Since both the Pendulum architecture and the R language design are in flux, the compiler was written so as to make modifications very easy. The compiler is written in Common Lisp, and works through a process similar to macro-expansion, where high-level language constructs are broken down into sequences of lower-level constructs, and the process finally bottoms out when the lowest-level constructs are translated directly into assembly language instructions. This design makes it very easy to add new high-level constructs, or change the compiler to support a different low-level architecture.

The R compiler did indeed turn out to be completely straightforward to implement (no surprises), and it was used to successfully compile a number of test programs, which were run under a simulator for the Pendulum architecture. It would be easy to extend the language and write more programs, if that were a priority.

The R compiler source code, internal constucts, and a brief usage summary are given in appendix D. The source files can also be downloaded from `http://www.ai.-mit.edu/~mpf/rc/memos/M08/*.lisp`.

9.5 Reversible algorithms

In this section we summarize what we have learned about efficient reversible serial algorithms for a variety of problems in computer science and physics. These include sorting, arithmetic, matrix operations, graph problems, and physical simulations. Due mainly to a lack of time, we have not written down sample code for any of these except our physical simulation (appendix E). An important area for future work is to specify a broad range of reversible algorithms in complete detail, with sample code. However in most cases the details are fairly obvious and straightforward.

We focus on serial algorithms in this section primarily for simplicity. To support the long-term applications of reversibility in massively parallel computing, the development of good reversible parallel algorithms (for a mesh architecture) for a variety of problems would also be desirable.

9.5.1 Sorting

Sorting a list in place is an inherently non-reversible operation, because information is lost: namely, the original order of the elements. So in general, a reversible sort must produce some extra garbage data. For arbitrary lists, the minimal worst-case garbage is $\mathbf{\Theta}(\log n!)$ since that is the number of bits required to specify the original permutation of the elements.

Simple insertion sort performs $\mathbf{\Theta}(n^2)$ comparisons in the worst case. Hall [81] observed however that only $\mathcal{O}(n \log n)$ garbage bits need be generated to run this algorithm reversibly: n integers each $\mathcal{O}(\log n)$ bits long telling how far each element was moved down the list before being inserted in the proper place. If these numbers are stored as variable-length bit-strings instead of fixed-length words, the garbage space usage becomes $\mathbf{\Theta}(\log n!)$, exactly the minimum.

Similarly, we realized that any of the standard efficient $\mathbf{\Theta}(n \log n)$-time comparison-based sorting algorithms, such as quicksort, are easy to turn into good reversible algorithms, by simply saving away bits giving the result of each comparison, telling whether two elements were exchanged or not on any given step of the algorithm.

Even radix sort, which takes $\mathbf{\Theta}(n)$ time for n $\mathbf{\Theta}(1)$-size elements, can be easily turned into an efficient reversible sort which takes $\mathbf{\Theta}(n)$ time and produces $\mathbf{\Theta}(n)$ garbage.

9.5.2 Arithmetic

Addition or subtraction of one n-bit number into another can be performed reversibly in $\mathbf{\Theta}(n)$ time with no garbage data.

The simple $\mathbf{\Theta}(n^2)$-time grade-school algorithm for multiplication (*e.g.*, see fig. 9.3) can also be performed reversibly with no loss in asymptotic efficiency, and no garbage other than the operands (if they are no longer needed). If the multiplicands are specified to be nonzero, then one of them can be uncomputed based on the product and the other multiplicand, thus reducing the garbage further. (After computing the product, the multiplicand can be uncomputed using $\mathbf{\Theta}(n^2)$-time division.)

If the multiplicands are prime and sorted, then in principle *both* of them could be uncomputed from the product reversibly, and the operation would create *no* garbage (since the number of bits in the product will be roughly the sum of the number of bits in the two multiplicands). However, since there is no known efficient classical algorithm for factoring, doing this is in general very slow.

9.5.3 Matrices

Similarly, the simple $\mathbf{\Theta}(n^3)$ algorithm for multiplying $n \times n$ matrices can also be efficiently reversible. If the left multiplicand is nonsingular, then the right multiplicand can be uncomputed given the product, reversibly and efficiently ($\mathbf{\Theta}(n^3)$).

9.5.4 Searches

A systematic depth-first or breadth-first search of a tree can be carried out reversibly; the nodes are visited in a particular sequence, and the prior node in the sequence can be determined from the current node. So, for example, the naive SAT algorithm, of generating and testing all possible assignments to the boolean variables, incurs no additional asymptotic overheads when performed reversibly.

9.5.5 Graph problems

For the all-pairs shortest-path problem, one of the best algorithms is the Floyd-Warshall algorithm, which takes time $\mathbf{\Theta}(n^3)$, space $\mathbf{\Theta}(n^2)$. This is an example of an algorithm which appears to require asymptotically either more time or space when performed reversibly. The reason is that the Floyd-Warshall algorithm performs $\mathbf{\Theta}(n^3)$ irreversible updates of array elements in working storage, in the worst case. Performing all those updates reversibly would thus require $\mathbf{\Theta}(n^3)$ temporary storage, significantly worse than the original irreversible algorithm.

However, if one instead uses an alternative algorithm, of repeatedly "squaring" a connectivity matrix between graph edges, then all-pairs shortest path can be performed reversibly in time $\mathbf{\Theta}(n^3 \log n)$ and space $\mathbf{\Theta}(n^2 \log n)$—only slightly worse than the irreversible Floyd-Warshall algorithm on both time and space. Thanks are due to F. Thomson Leighton and his collaborators for pointing out the high reversible efficiency of this alternative approach, in personal discussions.

This example of the all-pairs shortest path problem illustrates how the best reversible algorithm for a problem might not necessarily correspond to a simple modification of the best irreversible algorithm for that problem, which is why hiding the fact of a machine's underlying reversibility from the programmer (or algorithm designer) is not a good idea.

9.5.6 Physical simulations

Direct, dynamic spatial simulation of reversible physical systems is often straightforward to perform reversibly with no asymptotic overheads compared to an irreversible version of the simulation. This is true at least when the system model being simulated is reversible. Since physics really is reversible at a low level, such models are often appropriate, and can exhibit much more stable and realistic behavior than competing irreversible dynamic models [75].

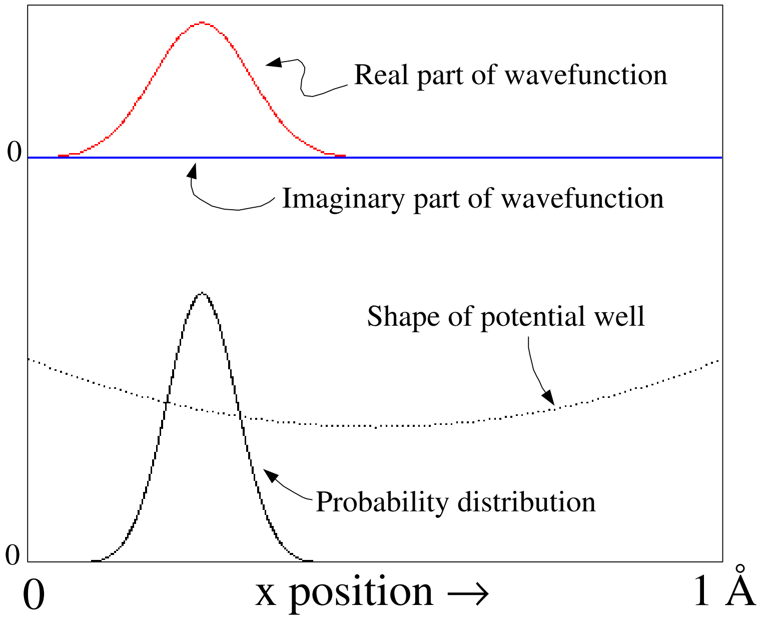

Figure 9.4: Example of an initial state of the Schrödinger wavefunction simulation. A wave for an initially stationary electron is placed up along the side of a 1 Å wide parabolic potential well. Its initial potential is approximately 4000 eV, relative to the bottom of the well. (A very strong potential given the small size of this space!)

As an example, we wrote a reversible simulation of the evolution of a simple quantum wavefunction according to Schrödinger's wave equation (see figures 9.4 and 9.5 for example output). The original irreversible version of the algorithm behaved well for a few hundred steps, but was eventually swamped by ever-growing artifacts of the simulation technique. We then reimplemented our update rule so that it would perform a perfectly reversible transformation of the wavefunction state on each step. The artifacts disappeared; the system was never observed to blow up again even after runs lasting millions of steps.

Moreover, the new version of the algorithm could be implemented under our reversible programming language in such a way that *no* garbage data would be accumulated; the algorithm required only *constant* space on our reversible architecture, independently of the number of simulation steps that were performed.

The mathematical derivation and code for the algorithm (in C, R and PISA versions) are given in appendix E.

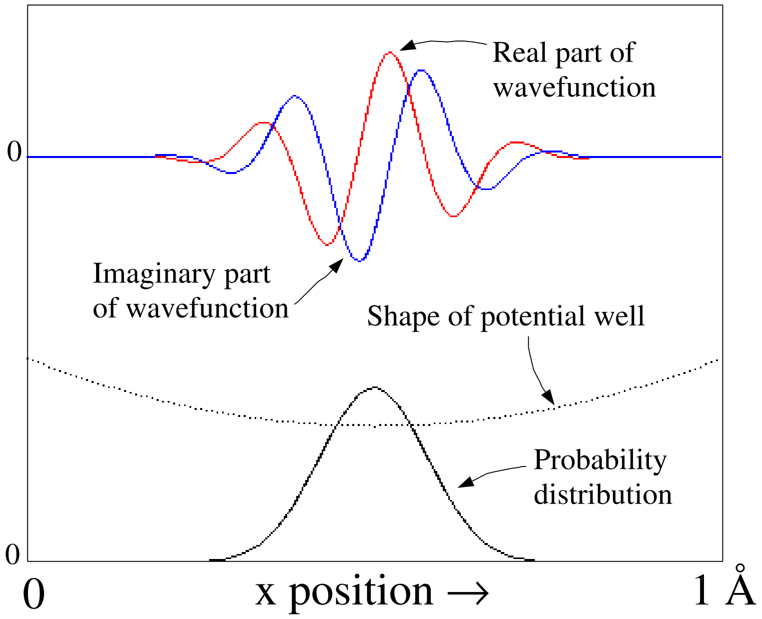

Figure 9.5: The state of the simulated wavefunction of fig. 9.4 after ~ 1000 simulation steps, representing about $\frac{1}{2}$ atto-second (5×10^{-19} sec) of physical time. The electron wave packet has fallen to the bottom of the potential well, and is now moving to the right at about $\frac{1}{10}$ of the speed of light.

9.6 Operating system issues

So far I have discussed reversible computer programming from the point of view of running a single reversible program at a time. However, if we want to allow for the possibility that eventually reversibility will, for one reason or another, be in widespread use for general-purpose computing, then we should address the issue of how users might be enabled to run multiple programs concurrently on a single reversible computer, since this sort of multitasking has proven to be extremely useful in current computer systems.

Running multiple programs concurrently is traditionally part of the job of an operating system. In this thesis, we have not attempted to study how the wide range of services provided by modern operating systems might be implemented on a reversible computer, since we believe that most of these services would straightforward to implement reversibly (given our general purpose reversible-programming framework) and thus would not be very interesting.

However, we now briefly examine how a multitasking facility might be implemented on a reversible computer, since we know of some interesting problems associated with that goal.

If the multiple reversible programs are not interacting at all with each other, then it appears straightforward to run them concurrently on a reversible processor. There simply needs to be a mechanism for CPU control to be periodically transfered from within the individual programs out to an external scheduler, which decides which other program to run next, and makes the appropriate changes to machine state to transfer control to within the body of that other program, in order to continue from the point where that program left off.

One way to achieve this "escaping to the scheduler" would be to simply have the compiler periodically insert in program code instructions that transfer control to the scheduler. Timed hardware interrupts are also quite possible and straightforward.

In fact, even if the computer's instruction set includes software control over the direction of execution, and even if the running of individual programs involves those programs switching the CPU direction, and running pieces of their code backwards and forwards, it is still straightforward to integrate this activity with process switching, by having the scheduler keep track of which way the CPU was going when a process was last being run, as part of the saved state of that process.

However, a problem arises if we wish to allow multiple concurrently executing reversible programs to communicate with each other. Namely, what happens when a program reverses over an I/O instruction that communicates with another reversible process?

For example, suppose process A is running forwards and outputs a piece of data X through a stream-like facility, and later the datum X is received by another process B, which is also running forwards. Then process A reverses CPU direction, and attempts to undo the operating system call that sent X. How does the OS handle this situation?

Similarly, looking at the situation from B's point of view, suppose B reverses and undoes the call that received X, and then proceeds forwards again. Will the next piece of data received be X again, or will it be the next datum produced by A after it produced X? (We note that this question is related to the question of whether a theoretical reversible finite automaton (RFA) should be defined as having a one-way read-only input, as Pin defined it (see §3.3.2.2, p. 60), or as having a two-way input with operations to both read and unread data.)

There are several possible answers to this question.

- Disallow reversing over I/O calls. Declare such an action to be illegal, an error. However, this approach seems overly restrictive.

- Allow reversing over I/O instructions, but treat this as a no-op; don't actually undo the I/O action. This is like treating each process as if it were running on a separate reversible processor, but with an irreversible communications channel. This seems OK, but still limited.

- Finally, allow reversing over I/O instructions, but when this happens, actually reverse the flow of data in the data stream. Reversing over output is just like pulling back out the last thing that you output when running forwards; reversing over input is like stuffing the last thing you input back into the pipe.

In future work we may experiment with all three options, and will perhaps think of more. The third option seems the most interesting to explore, however. It raises the further question of what happens when you attempt to reverse over an output instruction, but the program at the other end of the pipe has already pulled out and used the last datum X that you sent.

One possibility is that the program B is then forcibly reversed until it gets back to the point where it received X, so that then X gets stuffed back into the pipe, and A retrieves X (running backwards). However, this seems rude; it prevents B from doing whatever it might have been planning to do with X before reversing over the input instruction that got it.

Another possibility treats the situation more symmetrically. Suppose that reversing over an instruction to output data to a stream really is exactly like getting input from the stream—just like the stream were a totally symmetric, big, stretchy hose where you can stuff things in either end, and get things from either end, but where the things inside always remain in the same order relative to each other.

Well, then if you try to get a datum out, but there is no datum inside, then the normal thing is to *block* and wait until the data is available. *I.e.,* when A tries to reverse over its "output X" instruction, it blocks until B decides to reverse over its "input X" instruction. This is just like what happens in a normal operating system when you try to input from a stream when running forwards, and there is nothing in the stream to be read.

This of course leads to still more problems. What if B pushes back into the pipe a different value than the one that A originally sent? (This can happen whenever programs have control over their own execution direction in a non-trivial way.) Program A might not be expecting to get a different value back when he reverses over his "output X" instruction; and so program A might not function as expected. (If reversibility isn't guaranteed at the hardware level, and we're not careful, program A might even fail to reverse properly, and may irreversibly dissipate information to heat.)

All we can say about this problem at this point is that a communications mechanism is not a communication protocol, and if we want multiple concurrent reversible programs to communicate meaningfully with each other along reversible channels such as this, we will have to describe more precisely what the intent of those communications would be. It would help if we had in mind some particular candidate application for which multitasking would be a useful abstraction.

9.7 Parallelism

Finally, we note that we have not said much so far about the design and programming of parallel reversible computers. Of course, one can treat a parallel computer as just a set of interacting serial computers, and program it that way. As long as reversible I/O operations are provided, nothing need be broken. And, as we discussed in earlier chapters, the processors could be organized into a mesh structure that would be very regular, relatively easy to program, and asymptotically optimal.

However, even with the right physical architecture, programming a parallel system as a set of independent interacting serial processes is very hard. It would of course be nice to have a means to express parallel reversible algorithms at a high level, and have a compiler do to the work of translating that high-level parallel algorithm into sequential algorithms to run on the individual interconnected processors.

Such a language would allow us to easily express "physical" algorithms, specifying the movements and interactions of whole "fields" of data extending through the machine. The ideal language would somehow implicitly incorporate the realities of physical constraints, such as that large amounts of data can not all be in the same place at the same time. It might describe data operations in terms that traditionally we associate more with physical processes: moving, sifting, mixing, separating, recombining, chemically reacting, etc. Such analogies become increasingly appropriate when the computing system is designed to admit the fact that information is conserved (in the sense that its transformation are reversible), takes up space, and must physically move from one place to another.

Programmers of such a "physical" programming system would use the skills of a computer systems architect, or even those of a mechanical or industrial engineer, such as an assembly-line designer, as well as the skills of a sequential program coder. The programmer's job would be to design a dynamic assembly line for information within the real physical 3-D space of the machine. He would have an advantage over the factory designer in that all parts of his "factory" (namely, the individual processors) can be reprogrammed and reconfigured dynamically at will to serve his needs. Also since the material being manupulated is just information, there are no concerns with weight, structural support, etc. But there are still concerns about flows! The information takes up space. Where does the unwanted information go when it is no longer needed? The programmer would design pathways for the flow of both useful information and garbage information through the machine.

Good programming tools might perform the detailed routing of these pathways automatically, even dynamically as the system runs. But a good programmer should still occasionally find that concerns with the physical nature of information come into his thinking during the algorithm design process. The expert programmer should not mind expanding his expertise to designing algorithms for the manipulation of information considered as a *conserved* material-like thing, embedded in 3-D space. After all, *that* is what information really is like. The ultimate, best-performing algorithms can never be discovered by those who are afraid to step out of the imaginary serial, random-access, bit-destroying world of programming that we have constructed for ourseves up to now.