Writing Efficient Itanium 2 Assembly Code

Mike Burrell (mike@wizardlike.ca)

October 12, 2010

This document will likely undergo revisions through time. The date on the title page should be used to determine whether a document is most recent.

Preface

This document attempts to provide tutorial in writing efficient assembly code for the Itanium 2 processor. The primary motivation of writing this is to show off the elegance of the IA-64 architecture, to demonstrate to as wide a programming audience as possible the features which make IA-64 unique. As of the time of this writing, IA-64 seems destined to become a niche architecture, used only in high-end servers for high-performance computation, and even that small niche is not a sure thing. Even if the Itanium 2 should disappear, the ideas it presented will almost certainly be reintroduced.

The secondary, and more practical, motivation for this that since writing efficient code for the Itanium 2 depends so severely on having good compilers and since no good compilers have ever been written, it should be expected to write assembly code. It is not uncommon for hand-written Itanium 2 assembly code to be 50% more efficient than that generated by the best C compilers under the most aggressive optimization settings.

What is expected of the reader is that she has access to an Itanium 2 processor, has access to the GNU Compiler Collection (`gcc`) and the GNU debugger (`gdb`), and is comfortable with general low-level architecture concepts such as registers and branches. The IA-64 architecture likely makes a poor choice as the first architecture an assembly programmer should learn due to number of concepts that need to be learned up front. For assembly language neophytes the pacing may be brisk. Readers unfamiliar with the SPARC architecture should pay special attention to section 3.2.1.

Contents

Preface **iii**

I Introduction **1**

1 Development environment **3**

2 Simple beginnings **7**
2.1 Register-to-register instructions 7
2.2 Explicit parallelism 9
2.3 Static data and the global pointer 11
2.4 The debugger 12
2.5 Multiplication and division 13

3 Branches **17**
3.1 Branches and RAW dependencies 17
3.2 Function calls 17
3.2.1 Register windows 19
3.3 Predicates 21
3.3.1 Comparisons and predicated instructions 22
3.3.2 Conditionals 26
3.3.3 Loops 26

4 Bundles **29**
4.1 Bundle formats 29

II Optimizations **33**

5 Rotating registers **35**
5.1 Loop unrolling 37

List of Tables

3.1 The general purpose registers, r0–r127, and the conventions used across function call. See section 5.2 of Intel's conventions document [Int01b] for more information. 21

3.2 The predicate registers, p0–p63, and the conventions used across function call. See section 5.4 of Intel's conventions document [Int01b] for more information. 22

3.3 The result of performing a parallel cmp.ge.and.orcm operation from figure 3.4. 25

4.1 Bundle formats defined by IA-64. 29

5.1 Values of registers through a loop rotating registers, where each iteration we execute the instruction add r32=r33,r34. 40

List of Figures

1.1 The source file `return-42.m`, which returns a status of 42 to the operating system. . . . 3
1.2 An example Makefile, suitable for GNU make, which automatically produces ".bin" files from ".m" files. . . . 5

2.1 The source file `simple-alu.m`, which performs simple and pointless arithmetic instructions. . . . 8
2.2 The source file `simple-alu2.m`, which performs simple and pointless arithmetic instructions, with the use of an explicit stop to allow initialized values. . . . 10
2.3 The source file `add-numbers.m`, which makes use of static data. . 12
2.4 Using `gdb` to trace through the example given in figure 2.3. . . . 14
2.5 C source code for performing common arithmetic operations. . . 15

3.1 The source file `hello-world.m`. . . . 18
3.2 Register renaming of a function **x** calling function **y**. For our purposes, **x** has 1 input register (1 parameter) and 3 local registers; **y** has 2 input registers (2 parameters) and 1 local register and no output registers. As a consequence, **x** requires at least 2 output registers: in this example it has exactly 2 output registers. 20
3.3 The source file `print-arg.m`, which prints out the command-line argument if there is one; otherwise it prints out an error message. 23
3.4 The source file `between.m`, which returns a boolean value indicating whether its third argument is inclusively between its first two arguments. . . . 24
3.5 Using predicated branches for if-then-else constructs. . . . 26

5.1 The **fib** function of the source file `fib.m`, which computes the Fibonacci number in the most naïve way. . . . 36
5.2 The **main** function of the source file `fib.m`. . . . 38
5.3 Running times for three separate implementations of a simple Fibonacci number calculator on a 900MHz Itanium 2. . . . 38
5.4 The source file `fib2.m`, which performs two calculations per loop and hence one cycle per n. . . . 39

Part I

Introduction

Chapter 1

Development environment

The examples of Itanium 2 assembly code given in this document assume the use of `gcc` and were tested on the HP-UX operating system. In reality the operating system should make no difference and a Linux-based operating system should work just as well. The standard Unix macro processor `m4` will also be assumed to exist and is used to define macros that will make the assembly source code more readable and easier to maintain.

Should you wish to use the `cc` compiler bundled with HP-UX, the differences in syntax are slight. In particular, labels should two colons rather than one colon. The `+DD64` command-line option is used to mandate the 64-bit API rather than `-mlp64`.

Figure 1.1 gives the source code listing of our first program. Historically we would start off with a "hello world" program, but such a program would be too complex for assembly code.

All lines starting with a dot and written in sans-serif typeface are **compiler directives** and do not produce machine code. We will not belabour the compiler directives right now, but will take on faith that we should include them in every program. The .text directive introduces the segment of the produced object file

```
        .text
        .global main
        .proc main
main:   .prologue
        .body
        mov ret0 = 42
        .restore sp
        br.ret.sptk   rp      // return to the operating system
        .endp main
```

Figure 1.1: The source file `return-42.m`, which returns a status of 42 to the operating system.

that holds the machine code (it is usual for the code segment to be named the "text" segment); the .global directive ensures that the **main** function will be globally visible in the object file; the paired .proc and .endp directives sandwich any function which we define; and the .prologue, .body and .restore directives allow for more debugging information should we wish to run our program in `gdb`.

Comments start with a double-slash and will be written in *italic* typeface by convention. Labels end with a colon and are written in **boldface**.

This leaves two instructions: mov ret0 = 42; and br.ret.sptk rp. As one might expect, the first instruction is the return value of the **main** function. On the Itanium 2, there are registers specific to returning values from functions. Functions which return only one value—which is just about every function—will return its value in the ret0 register. The second instruction, as one might expect, returns from the function, though deciphering it may be less obvious. br.ret.sptk is read as "branch, returning from this function, and we statically predict that this branch will be taken". The "sptk" component of it is semantically unnecessary, but unfortunately syntactically required by the assembler. Almost all br.ret instructions will be of the br.ret.sptk variety. The rp operand to the br.ret.sptk instruction is the "return pointer", a register which holds the address to return to, analogous to the "link register" used in other architectures such as POWER.

The caption for figure 1.1 gives a filename of `return-42.m`. We will use the convention of having m4 source files a ".m" extension and raw assembler source files a ".s" extension; further, we will use the convention of always writing assembly code as m4 source files. Even though no m4 macros have been used in `return-42.m`, we write it as an m4 file anyway, solely for the pedagogical purpose of demonstrating the toolchain used to assemble and run the code, which the following shell commands[1] demonstrate:

```
zx624% m4 return-42.m >return-42.s
zx624% gcc -mlp64 -o return-42.bin return-42.s
zx624% ./return-42.bin
zx624% echo $?
42
zx624%
```

The usage of `-mlp64` in the invocation of `gcc` mandates the 64-bit API available under HP-UX and properly under any operating system. While the Itanium 2 is a native 64-bit processor and effectively has no "32-bit mode", the default on HP-UX is to use the 32-bit API, namely that integers and pointers stored in memory should be stored using 4-byte instructions (st4 and ld4) rather than 8-byte instructions (st8 and ld8). We settle on the 64-bit API in this document and assume all integer and pointer primitives are 64 bits wide.

[1] The Z shell, `zsh`, is used for all shell examples in this document, though the use of shell is a moot issue.

```
.SUFFIXES:      .m .s .o .bin

.m.s:
        m4 $^ >$@
.s.o:
        gcc -mlp64 -g -c $^
.o.bin:
        gcc -mlp64 -g -o $@ $^
```

Figure 1.2: An example Makefile, suitable for GNU make, which automatically produces ".bin" files from ".m" files.

The constant usage `m4` and `gcc` is tedious when trying code and, for the rest of this document, we assume the presence of a Makefile which automatically produces documents for us, as shown in figure 1.2.

Chapter 2

Simple beginnings

2.1 Register-to-register instructions

We begin with the simplest instructions of any architecture, the ALU (arithmetic and logic unit) register-to-register instructions. A full specification of *all* instructions is published by Intel [Int06b] and we will not repeat descriptions of instructions in this document, but only give examples and guides through the most typical uses of them.

One of the more famous attributes of the IA-64 is its plethora of registers, though until the next section we will use only a tiny number of them, less than 5% of the registers available to the programmer. Even among the so-called general-purpose registers, those used by ALU, we will constrain ourselves to using only a few of the 128 visible to the programmer until the next chapter. The fact of the matter is that the first 32 general-purpose registers, r0–r31, are more than sufficient for describing any simple register-to-register operations might want to perform. IA-64's extra registers are for more specialized use.

A quick description of the first 32 general-purpose registers follows. A more thorough description can be found [Int01b].

r0—Hard-coded to the value 0 (all bits 0). Unlike on other RISC architectures such as SPARC, writing to this register will not silently discard values; rather, writing to r0 will cause an "illegal instruction" error.

r1—The global pointer, to be described in later chapters. Do not touch!

r13—The thread pointer, to be described in later chapters. Do not touch!

r12—The stack pointer, to be described in later chapters.

r4–r7—Preserved ("callee-saved") global registers. These should not be modified until being stored somewhere else first so they can be restored.

r8–r11—ret0–ret3, the return registers.

```
        .text
        .global main
        .proc main
main:   .prologue
        .body
        mov  r14 = 6          // r14 will have the value 6
        add  r15 = 6,r0       // r15 will also have the value 6
        add  r16 = r17,r18    // r16 will have the value r17 + r18
        sub  r17 = 4,r18      // r17 will have the value 4 - r18
        shl  ret0 = r18,3     // return the value 8 · r18
        .restore sp
        br.ret.sptk  rp       // return to the operating system
        .endp main
```

Figure 2.1: The source file `simple-alu.m`, which performs simple and pointless arithmetic instructions.

r2–r3,r14–r31—Scratch ("caller-saved") global registers which are completely free for use.

It should be of no surprise that we will restrict ourselves to the last row, the r2–r3 and r14–r31 registers. An astute reader might recognize that there are a few usual suspects for global registers which are missing here, such as the instruction pointer, ip, or the return pointer, rp, which we used in our first example. On the Itanium 2, the ip is considered a special register, distinct from every other register which cannot be used as a general-purpose register; the rp register is considered a "branch register", which will be covered in later sections, distinct from general-purpose registers.

Figure 2.1 shows some simple ALU instructions, which should give an introduction to the flavour of the syntax used in Itanium 2 assembly code. Of course running that code yield nonsense results. The returned value is eight times r18 but r18 has never been initialized! A clever reader might try to initialize r18 first only to be given an error message as follows:

```
simple-alu.s: Assembler messages:
simple-alu.s:11: Warning: Use of 'shl' violates RAW dependency
'GR%, % in 1 - 127' (impliedf), specific resource number is 18
simple-alu.s:11: Warning: Only the first path encountering the
conflict is reported
simple-alu.s:10: Warning: This is the location of the conflicting
usage
```

Actually under `cc` bundled with HP-UX this would be an error, not a warning, and for very good reason. Trying to initialize r18 (by inserting an instruction such as "mov r18 = 2") before issuing the shl instruction is an error on the Itanium 2, unlike almost every other architecture to date, and would cause undefined results if executed.

The first warning tells us moderately clearly what the problem is: it is a RAW dependency violation and it involves r18. But what is a RAW dependency violation?

2.2 Explicit parallelism

Unlike almost every other architecture to date, and certainly unlike every processor architecture designed for general-purpose use, the IA-64 architecture requires instruction-level parallelism to be *explicit* in the machine code. Indeed before being renamed IA-64, the architecture was called EPIC (explicit parallelism instruction computing) in honour of this.

Instructions must be grouped together into **instruction groups**. For our purposes (the actual definition covers a few more cases), an instruction group is defined as a series of contiguous instructions which end with either: (a) a branch which is taken; or (b) an explicit "stop". Because we will not consider branches until section 3, we will focus on stops.

A stop is a command to the CPU that all instructions previous to that may be executed in parallel and that some instructions after the stop depend on the values which are to be computed. It is, in essence, a request to the CPU to finish executing what it is currently executing before moving on to new instructions. In reality, of course, it is more complicated than that: the CPU is free to, and will, execute instructions after a stop before instructions before the stop have finished executing, but it will place them at a different pipeline stage so that it is guaranteed the values will be computed in time.

In almost every other architecture, these dependencies would be handled *implicitly* by the CPU. The instructions "mov r18 = 3" and "mov r19 = r18" would cause either an *implicit* stall in the pipeline or an *implicit* reordering of instructions to avoid the inconsistency inherent in the fact that the first mov instruction will not finish before the second begins executing. On the Itanium 2, very little is implicit: it is borne on the programmer to *explicitly* determine which instructions have dependencies on one another. As we have seen, the assembler will assist us with some static analysis.

Even though they execute in parallel, **there is still an ordering among instructions within an instruction block**. This distinction matters for WAR (write-after-read) dependencies. The four types of register dependencies are as follows:

RAR (read-after-read)—this clearly cannot cause any problems and consequently there is no such thing as a RAR dependency violation. The following is a RAR dependency, which causes no problems:

```
mov r14 = r15       // read from r15
add  r16 = 3,r15    // read from r15 again
```

WAR (write-after-read)—it might be expected that this can cause problems, but the ordering of the instructions ensures it won't and consequently

```
        .text
        .global main
        .proc main
main:   .prologue
        .body
        mov  r14 = 6            // r14 will have the value 6
        add  r15 = 6,r0         // r15 will also have the value 6
        ;;
        sub  r17 = 8,r15        // r17 will have the value 2
        add  r18 = r15,r14      // r18 will have the value 12
        ;;
        shl  ret0 = r18,r17     // return the value 48
        .restore sp
        br.ret.sptk  rp         // return to the operating system
        .endp main
```

Figure 2.2: The source file `simple-alu2.m`, which performs simple and pointless arithmetic instructions, with the use of an explicit stop to allow initialized values.

there is no such thing as a WAR dependency violation. The following is a WAR dependency, which causes no problems:

```
mov r14 = r15          // read from r15
add  r15 = r16,r17     // write to r15
```

RAW (read-after-write)—this is the problem we ran into above and requires an end to the instruction group—e.g., by using an explicit stop—to solve. The following is illegal code:

```
mov r14 = r15          // write to r14
add  r15 = r14,r16     // read from r14. Problem!
```

WAW (write-after-write)—this also is a violation and must be solved by ending the instruction group. The following is illegal code:

```
mov r14 = r15          // write to r14
add  r14 = r15,r16     // write to r15 again. Problem!
```

The use of a double semicolon marks an explicit stop. For instance, the following would be legal code and would solve a RAW dependency:

```
mov r14 = r15          // write to r14
;;                     // stop
add  r15 = r14,r16     // read from r14. No problem!
```

One can place an explicit stop after every instruction to avoid having to deal with dependency issues. A stop is always permitted after an instruction. However, this carries grave performance issues. Unlike other processors, an Itanium 2 does not have the ability to *implicitly* parallelize machine code; it relies

on the programmer to *explicitly* do that. By placing a stop after each instruction it becomes impossible for the CPU to execute instructions concurrently.

Figure 2.2 shows an example similar to figure 2.1, but with initialized values.

2.3 Static data and the global pointer

It's a common problem on RISC architectures to load immediate values. Because every instruction is of a fixed length, usually the same length as a machine word, it's typically impossible to load a full immediate in one instruction. On the Itanium 2, instructions are actually *shorter* than machine words: 41 bits versus 64 bits. The Itanium 2 does offer double-wide instructions for dealing with 64-bit immediates, which will be dealt with later on, but they are seldom used.

The most common use for large immediates is dealing with static pointer values, usually pointers to static data in the object file or pointers to code such as external functions. Rather than force the programmer to use double-wide instructions and load 64-bit immediates for these pointers, the philosophy is to make these pointer values relative in the hopes that we can represent them in a smaller address space and thus load them in a single instruction.

Every executable binary contains with it, in addition to machine code, static data. Once loaded into memory by the loader so that the code can execute, this static data is located in the process's address space. The ELF format used for Itanium 2 processes specifies how this data is laid out [Int01a]. On most architectures the static data would be placed at the beginning of the address space; on Itanium it not necessarily so, but the global pointer, gp, provides the base address such that everything else can be offset from it.

Typically, Itanium 2 instructions allow for only 14-bit immediate values (ranging from −8192 to +8191). However, when using the first four registers, r0–r3, we are allowed to use 22-bit immediate values. The global pointer, gp, is conveniently r1, which means it can be used as on offset with 22-bit immediates in one in instruction. This is not a coincidence.

Figure 2.3 features our first example of a program using static data. The first change is that we are now using m4's `define` macro to define names for local registers. This is a convention that will be followed throughout the rest of the document. Intel's own assembler, `ias`, has an option to automatically produce local registers from symbol names, which would diminish the need to use m4. One can also see the use of explicit stop bits as a way to separate RAW dependencies within the code.

The `@gprel` assembler directive reduces a symbol down to its location in the process's address space, relative to the global pointer. Consequently, adding that offset on to gp will yield a pointer to the value in memory. The ld8 instruction—load 8 bytes—will then load a 64-bit value from that location in memory into a register, where it can be manipulated.

After the code we see the introduction of the `.sdata` section. The `.sdata` section is short for "short data" and is the first section we have seen outside

```
define(a_reg, r14)                    // m4 macros defining register names
define(b_reg, r15)
        .text
        .global main
        .proc main
main:   .prologue
        .body
        add     a_reg = @gprel(a),gp  // get the pointer to 'a'
        add     b_reg = @gprel(b),gp  // get the pointer to 'b'
        ;;
        ld8     a_reg = [a_reg]       // load the value from the object file
        ld8     b_reg = [b_reg]
        ;;
        add      ret0 = a_reg,b_reg   // return a + b
        .restore sp
        br.ret.sptk      rp           // return to the operating system
        .endp main

        .sdata
a:      data8       17
b:      data8       19
```

Figure 2.3: The source file `add-numbers.m`, which makes use of static data.

of the .text section, the section used to store machine code. Itanium 2 binaries contain two data sections which we will be dealing with in this document: the "short data" section, designed for data which is 8 bytes or smaller in size; and the "data" section, which is designed for larger data. In future sections we will see an example of the data section. Itanium 2 binaries do allow for other sections which will not be considered in this document.

The data8 assembler directive stores a 64-bit signed integer value in the object file. Should we want to use 32-bit values instead, we would use the data4 directives to store the data and the ld4 instructions to load the data.

2.4 The debugger

The use of a debugger is key for understanding any executable code, but is especially valuable with assembly code. For this document we assume the use of the GNU debugger, `gdb`. Although it will not be explicitly stated throughout the document, we assume the reader will be running any code she does not fully understand in the debugger to verify that the CPU is behaving as expected.

After producing `add-numbers.bin` from figure 2.3, one can run the command `gdb add-numbers.bin` from the shell to invoke the debugger. A warning message of "warning: Unwind base not found" means that the .prologue, .body or .restore assembler directives have not been used properly, a habit which should

be stamped out immediately.

Before running the debugger, it is recommend to create a `.gdbinit` file either in the reader's home directory, if `gdb` is going to be used exclusively for assembly language debugging on that system, or in the directory where assembly language development is being done, which should read as follows:

```
display /i $ip
break main
```

In lieu, these two commands can be issued at the beginning of every `gdb` session. The first says to display the next instruction to be executed at all times (literally, interpret the ip register as an instruction and display it every time `gdb` gives a prompt); the second says to create a breakpoint at the beginning of the **main** function, which is something the reader will typically want to do.

Figure 2.4 is a short session with `gdb` with the source code given in figure 2.3. First, the `run` (short form: `r`) command is given to execute the program from the beginning. As there is a breakpoint at the beginning of **main**, the debugger stops there and shows us the first instruction, add a_reg = `@gprel`(a), gp. In this case `@gprel`(a) is −16. By convention we use `p/x` (print in hex notation) for printing memory addresses, such as printing out gp. Unadorned `p` will print in decimal notation. The `stepi` (short form: `si`) command is used to single-step one instruction at a time. The `continue` (short form: `c`) command will continue execution until the next breakpoint, if there is one. Note that `gdb`'s convention is to prefix all register names with a dollar sign.

Further instruction on the operation of the GNU debugger can be found online [Fre10].

An observant reader might notice there have been instructions added which were not present in the original source file, specifically nop.i instructions. Other programs may generate phantom nop.m instructions. These are "no-ops" ("no operation"), operations which do nothing. The generation of these instructions is sometimes required, but they are harmless to the operation of the program. The only negative of them is that they waste the potential for more computation and thus can be considered inefficient. Their necessity and the elimination of them will be considered in chapter 4.

2.5 Multiplication and division

We have conveniently ignored two very common arithmetic operations: multiplication and division. The Itanium 2 does contain an integer multiplication instruction, but it is performed by the floating-point unit (FPU) and is beyond the scope of this chapter. The Itanium 2 does *not* contain an integer division instruction. Every Itanium 2 operating system should supply a standard function **.divI** (with the leading dot) to perform integer division. If multiplication, division or remainder are needed, it would be prudent to define one's own functions in C and then link with the assembly code to use as needed. In future sections we will look at using the FPU to perform integer multiplication and division.

```
zx624% gdb add-numbers.bin
HP gdb 6.1 for HP Itanium (32 or 64 bit) and target HP-UX 11iv2 and 11iv3.
Copyright 1986 - 2009 Free Software Foundation, Inc.
Hewlett-Packard Wildebeest 6.1 (based on GDB) is covered by the
GNU General Public License. Type "show copying" to see the conditions to
change it and/or distribute copies. Type "show warranty" for warranty/support.
..Breakpoint 1 (deferred) at "main" ("main" was not found).
Breakpoint deferred until a shared library containing "main" is loaded.

(gdb) r
Starting program: /home/mike/add-numbers.bin

Breakpoint 1, main+0 () at add-numbers.s:8
8       add-numbers.s: No such file or directory.
        in add-numbers.s
1: x/i $ip  0x4000000000000b40:0 <main>:
          addl             r14=-16,gp                                    MI,I
Current language:  auto; currently asm
(gdb) p/x $gp
$1 = 0x6000000000000018
(gdb) si
0x4000000000000b40:1          addl             r15=-8,gp;;
9       in add-numbers.s
1: x/i $ip  0x4000000000000b40:1 <main+0x1>:
          addl             r15=-8,gp;;
(gdb) p/x $r14
$2 = 0x6000000000000008
(gdb) si
0x4000000000000b40:2          nop.i            0x0
9       in add-numbers.s
1: x/i $ip  0x4000000000000b40:2 <main+0x2>:          nop.i            0x0
(gdb) p *(long *)$r15
$3 = 19
(gdb) si
0x4000000000000b50:0          ld8              r14=[r14]                                    MMI,
11      in add-numbers.s
1: x/i $ip  0x4000000000000b50:0 <main+0x10>:
          ld8              r14=[r14]                                    MMI,
(gdb)
0x4000000000000b50:1          ld8              r15=[r15]
12      in add-numbers.s
1: x/i $ip  0x4000000000000b50:1 <main+0x11>:
          ld8              r15=[r15]
(gdb) p $r14
$4 = 17
(gdb) si
0x4000000000000b50:2          nop.i            0x0;;
12      in add-numbers.s
1: x/i $ip  0x4000000000000b50:2 <main+0x12>:         nop.i            0x0;;
(gdb)
0x4000000000000b60:0          add              ret0=r14,r15                                 MIB,
14      in add-numbers.s
1: x/i $ip  0x4000000000000b60:0 <main+0x20>:
          add              ret0=r14,r15                                 MIB,
(gdb)
0x4000000000000b60:1          nop.i            0x0
14      in add-numbers.s
1: x/i $ip  0x4000000000000b60:1 <main+0x21>:         nop.i            0x0
(gdb) p $ret0
$5 = 36
(gdb) c
Continuing.

Program exited with code 044.
(gdb) q
zx624%
```

Figure 2.4: Using `gdb` to trace through the example given in figure 2.3.

```
long
mul(long x, long y)
{
        return x * y;
}

long
div(long x, long y)
{
        return x / y;
}

long
rem(long x, long y)
{
        return x % y;
}
```

Figure 2.5: C source code for performing common arithmetic operations.

The C code given in figure 2.5 would suffice. The curious reader can run the C compiler with the `-S` option, which will force the C compiler to reveal its generated assembly code.

Chapter 3

Branches

In a stored-program computer processor like the Itanium 2—like any modern architecture—the ability to branch is required to allow universal computational power. Without branching, we would not have the ability to iterate or recurse.

We have already seen one branch: the br.ret.sptk instruction. We will now consider most of the rest of the branches, with the more advanced and unique Itanium 2 branches saved for later sections.

3.1 Branches and RAW dependencies

It is a short note, but an important one, which warrants its own section. On the Itanium 2, **a branch is allowed to violate RAW dependencies**. In figure 3.1 we see that the br.ret.sptk makes use of the rp register even though there is no explicit stop between it and the "mov rp = rp_old" instruction. When we deal with predicated branches, i.e., conditional branches, we will see that we are allowed to assign to a predicate and then use that predict in a branch without an explicit stop between them.

Branches are the only instructions which allow RAW dependencies.

3.2 Function calls

The br.call instruction is used to perform function calls, though its operation requires some explanation for full understanding. Figure 3.1 shows the infamous "hello world" program, as written in Itanium 2 assembly, using the C function **puts** to print to standard out.

A lot has changed in this example compared to previous examples, beyond just the usage of the br.call.sptk instruction. The `@ltoff` directive has never been seen before; the `.data` section has never been before; the alloc instruction has never been seen before and the out and loc registers have never been seen before. The latter two of those novelties will be dealt with in subsection 3.2.1.

```
define(ar_pfs_old, loc1)
define(gp_old,     loc2)
define(rp_old,     loc3)
        .text
        .global main
        .proc main
main:   .prologue
        alloc       ar_pfs_old = ar.pfs,0,4,1,0     // set up a new register stack
        mov            gp_old = gp                  // save the global pointer
        mov            rp_old = rp                  // save the old return pointer
        .body
        add               loc0 = @ltoff(s),gp       // get a pointer in the
                                                    // offset table
        ;;
        ld8               out0 = [loc0]             // get the pointer to the string
        br.call.sptk        rp = puts                // print to the screen
        mov                 gp = gp_old              // restore global pointer
        .restore sp
        mov                 rp = rp_old              // restore return pointer
        mov              ar.pfs = ar_pfs_old         // restore the old function state
        mov               ret0 = 0                  // return success
        br.ret.sptk         rp
        .endp main

        .data
s:      stringz     "Hello world!"
```

Figure 3.1: The source file `hello-world.m`.

The .data ("data") section is only slightly different in operation as compared to the .sdata ("small data") section. Both introduce offsets relative to the global pointer, gp. The major distinction is that the data section requires an extra step of indirection. In the interest of keeping all offsets within a 22-bit immediate of the global pointer, "large" data, data in the data section, does not reside close to the global pointer, but only has a pointer to it reside close to the global pointer. The @ltoff directive gets us a pointer to the data we're looking for, relative to the global pointer. We then require a ld8 instruction to follow that pointer to the actual data.

In practice, almost all data can be stored in the small data section, and it is more efficient to do so. By Intel's conventions, a programmer *should* move data to the large data section if it is larger than 8 bytes, which our string is. In your own code you may wish to move data to the large data section only when the small data section has exceeded four megabytes in size.

Some explanation is required on why we saved the gp and rp registers, as well. The rp register was modified by the call to **puts**, and necessarily so: we required the use of the rp register so that **puts** knew where to return to. We then had to restore it before we ourselves to return to the operating system.

The gp register is less obvious, but it, too, is modified by the call to **puts**. In fact **every call to a function external to the current file changes the global pointer**. Each object file—each .o file—has its own value for the global pointer for the reason that every object file has its static data stored somewhere else in the process's address space. Thus, the global pointer should be restored after every function call.

The gp is also required to be correct—according to the current object file—before making any call to a function external to the current object file. In this case it doesn't come into play since we only make one external function call, but if we had more than one, we would be required to restore gp before making the next function call. Further, making a function call is an implicit read on the gp register. Restoring gp and making an external function call in the same instruction group is a RAW dependency and, unfortunately, one which is not caught by the assembler. Restoring gp and making an external function call without a stop between them will cause undefined behaviour.

As mentioned previously, every branch which is taken itself introduces an end to an instruction group, an implicit stop. There can be no RAW or WAW dependencies across a branch which is taken.

3.2.1 Register windows

IA-64 and SPARC are the only contemporary architectures which borrow from the original Berkeley RISC design the idea of **register windows**. Register windows allow some of a function's registers to be hidden on a function call and restored when the function returns. Further, they allow certain registers to be renamed on a function call.

Figure 3.2 shows how registers are renamed if hypothetical function **x** is calling hypothetical function **y**. Note that the operation is different from on

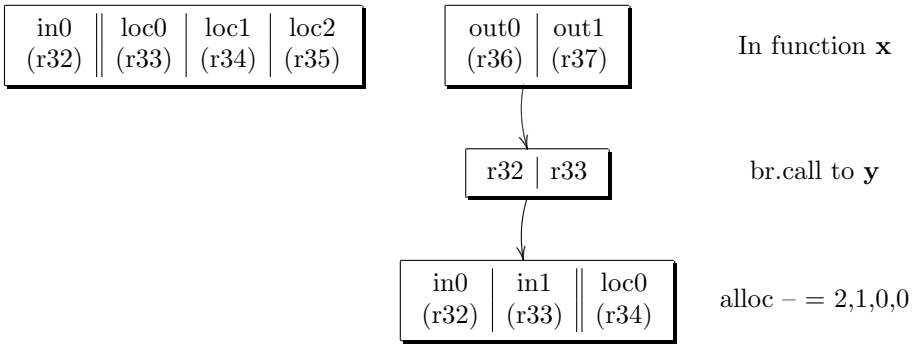

Figure 3.2: Register renaming of a function **x** calling function **y**. For our purposes, **x** has 1 input register (1 parameter) and 3 local registers; **y** has 2 input registers (2 parameters) and 1 local register and no output registers. As a consequence, **x** requires at least 2 output registers: in this example it has exactly 2 output registers.

the SPARC operation in a few ways. Firstly, the register window slides by a variable amount: the programmer chooses the number of input, local and output registers, rather than having each fixed at 8. Secondly, the renaming happens with the br.call instruction, *not* with the alloc instruction. On SPARC, the renaming only happens with the save instruction, not the call instruction. On the Itanium 2, it is the br.call instruction which renames the output registers to become the new input registers: the primary purpose of the alloc instruction is to record how many registers to save the next time a br.call instruction is issued.

What figure 3.2 shows is that the input and local registers of **x** become hidden to function **y** and thus there's no reason for **x** to save them. Further, the output registers of **x**, r36–r37, automatically become the input registers of **y**, r32–r33. This makes for very efficient function calls: no registers need to be saved in memory.

The syntax of the alloc instruction is "alloc r_d = ar.pfs, i, l, o, r" where i, l, o and r are all constants (immediates). The use of r will be dealt with in later sections and will always be treated as 0 for the beginning parts of this document. The i, l and o numbers indicate the number of input, local and output registers, respectively, used by the function. Two things to note: no register but ar.pfs may be used as the first operand; and the CPU makes no distinction between input registers and local registers. Indeed the encoding of the alloc instruction includes only two operands: $i + l$ and r. The separation of i and l is only for the benefit of the assembler, so that it knows how many input registers there and therefore which register loc0 is. The r_d is a general-purpose register that will store the previous function state such that it can be later restored: it is recommended to always store it in a local register.

The alloc instruction must always be the first instruction of an instruction group. Input, local and output registers allocated by an alloc instruction may

r0		Always 0 (all-bits zero)
r1	caller-saved	Global pointer (gp)
r2–r3	caller-saved	Scratch registers of use with 22-bit immediates
r4–r7	callee-saved	
r8–r11	callee-saved	Used for return values (ret0–ret1)
r12		Stack pointer (sp)
r13	caller-saved	Thread pointer (tp)
r14–r31	caller-saved	
r32–r127	automatically saved	Windowed registers (in0–in95, loc0–loc95 and/or out0–out95)

Table 3.1: The general purpose registers, r0–r127, and the conventions used across function call. See section 5.2 of Intel's conventions document [Int01b] for more information.

be immediately used without the need for a stop. This is not a violation of WAW dependency as alloc does not actually write to any of the register it allocates, and the registers were already *there* as soon as the br.call instruction was executed.

The register window always starts at r32 and extends up to r127. Thus, theoretically, one could allocate 96 fresh registers with an alloc instruction. Registers r0–r31 are fixed and thus are never renamed. Due to the large number of available registers, one could conceivably write a compiler specific to the Itanium 2 which does not require any spill code, i.e., which does not need to consider that there could be more values than registers and thus would need to "spill" registers into memory.

Table 3.1 gives a description of all general purpose registers and how they are treated across function calls.

3.3 Predicates

Unlike most other architectures, IA-64 offers no conditional branches. It accordingly has no integer condition codes, the familiar NZVC (negative, zero, overflow, carry) bits that are to common to almost every other general-purpose architecture to date. What it offers instead is **predication**, the ability to conditionally execute an instruction based on the value of some predicate register. Many architectures—even x86—offer some specialized predicated instructions and some architectures—most famously ARM—offer pervasive predication such that most instructions can be predicated in a general way. The IA-64 architecture is relatively unique in relying exclusively on predication for conditional execution, however.

Table 3.2 gives the IA-64's predicate registers and functional call conventions. Unlike for general purpose registers, there are no "automatic" predicate registers, registers that are automatically saved and restore with an alloc instruction. Unlike the general purpose register r0, the predicate register p0 does

p0		Always 1
p1–p5	callee-saved	
p6–p15	caller-saved	
p16–p63	callee-saved	Rotating registers, described in chapter 5

Table 3.2: The predicate registers, p0–p63, and the conventions used across function call. See section 5.4 of Intel's conventions document [Int01b] for more information.

not cause an illegal instruction error if written to; rather, the write is silently ignored.

Since the predicate registers are 1-bit in size and there are 64 of them, the entire predicate register file can be saved in an entire general purpose register, which modifies how we might treat them and nullifies the penalty of not having automatic predicate registers. If a function does not require any predicates to persist across function calls, then it should use predicate registers p6–p15, which are generally adequate for most needs. No saving or restoring is ever needed then. If any predicate is required to persist across a function call, it is generally easiest to save the entire predicate register file at call time and restore it at return time, effectively turning all 64 predicate registers into caller-saved.

The instruction "mov r14=pr" saves the entire predicate register file to register 14; the complementary "mov pr=r14" would then restore the register file.

Unlike on other architectures, an assembler for an Itanium 2 performs some static analysis, specifically in determining which predicates are mutually exclusive. Knowing that two predicates are mutually exclusive is essential for determining WAW and RAW dependencies. Two instructions in the same instruction group are allowed to have conflicting dependencies (RAW or WAW dependencies) if and only if they are predicated by mutually exclusive predicates, thus ensuring at most one of them will execute. Note that mutual exclusion is *not* the same as complementarity, though complementary necessarily implies mutual exclusion. Mutual exclusion guarantees at most one is true; complementarity guarantees exactly one is true.

- The assembler directive .pred.rel "mutex" is used to force the assembler to see two predicates as mutually exclusive. An example would be .pred.rel "mutex", p6, p7.
- The assembler directive .pred.safe_across_calls is used to force the assembler to assume that mutual exclusion properties are maintained across function calls. An example is .pred.safe_across_calls p1–p5, p16–p63, which includes all the conventionally callee-saved predicate registers.

3.3.1 Comparisons and predicated instructions

Unlike on most other architectures, there are no condition codes to set and thus the results from arithmetic operations cannot be used as conditions. The "cmp"

```
define(ar_pfs_old,      loc0)
define(gp_old,          loc1)
define(rp_old,          loc2)
            .text
            .global main
            .proc main
main:       .prologue
            alloc       ar_pfs_old = ar.pfs,2,3,1,0
            mov             gp_old = gp
            .body
            cmp.eq           p6,p7 = 2,in0              // check if argc is 2
            ;;
            mov             rp_old = rp
            (p6) add          out0 = 8,in1              // if so, get a pointer to argv[1]
            (p7) add          out0 = @ltoff(errorMsg),gp // otherwise, get a pointer to
                                                        // an error message
            ;;
            ld8               out0 = [out0]
            br.call.sptk        rp = puts
            .restore sp
            mov                 rp = rp_old
            mov                 gp = gp_old
            mov             ar.pfs = ar_pfs_old
            mov               ret0 = 0
            br.ret.sptk             rp
            .endp main
            .data
errorMsg:   stringz     "no argument given!"
```

Figure 3.3: The source file `print-arg.m`, which prints out the command-line argument if there is one; otherwise it prints out an error message.

```
define(ar_pfs_old,          loc0)
        .text
        .global between
        .proc between
between: .prologue
        alloc           ar_pfs_old = ar.pfs,3,2,0,0
        .body
        sub                 loc1 = in1,in2          // for checking if in1 ≥ in2
        cmp.le            p6,p7 = in0,in2           // in0 ≤ in2?
        ;;
        cmp.ge.and.orcm   p6,p7 = loc1,r0           // in1 ≥ in2?
        ;;
        .restore sp
        (p6) mov            ret0 = 1
        (p7) mov            ret0 = 0
        mov               ar.pfs = ar_pfs_old
        br.ret.sptk               rp
        .endp between
```

Figure 3.4: The source file `between.m`, which returns a boolean value indicating whether its third argument is inclusively between its first two arguments.

(compare) instruction is most often used to set predicate registers. Figure 3.3 gives an example of a program which uses the cmp instruction to set a predicate. Specifically, it uses the cmp.eq form, which tests for equality. If its two operands, 2 and in0, are equal, then it will set p6 to true and p7 to false; otherwise it will set p6 to false and p7 to true.

The predicates p6 and p7 are then used in the next bundle. The syntax of prefixing an instruction with "(p6)" means to only execute that instruction if p6 is true. In this case, the two add instructions will execute concurrently, though only one will actually be executed. Due to simple static analysis performed by the assembler, it is known that p6 and p7 are mutually exclusive (complementary, in this case) and thus the assembler will not emit a WAW dependency error.

The cmp instruction can be used to write to only one predicate register. In that case p0 is implicitly taken to be the second output register, which will silently discard the result.

The allowable forms of cmp instruction are eq, ne, lt, le, gt, ge, ltu, leu, glu and geu, forming all usual arithmetic comparators. Their usage does not require further explanation and the reader is directed to the instruction set reference for more information [Int06b]. The "tbit" (test bit) instruction is the other instruction commonly used to set predicate bits, and will also not be discussed here.

Figure 3.4 gives a slightly more complicated example of using conditionals. In this case the function **between** takes in three integers, a, b, c, and returns 1

Original values	**Comparison is true**		**Comparison is false**	
p6=0,p7=1	$p6 = 0 \wedge 1 = 0$	$p7 = 1 \vee (\neg 1) = 1$	$p6 = 0 \wedge 0 = 0$	$p7 = 1 \vee (\neg 0) = 1$
p6=1,p7=0	$p6 = 1 \wedge 1 = 1$	$p7 = 0 \vee (\neg 1) = 0$	$p6 = 1 \wedge 0 = 0$	$p7 = 0 \vee (\neg 0) = 1$

Table 3.3: The result of performing a parallel cmp.ge.and.orcm operation from figure 3.4.

if and only if $a \leq c \leq b$. Skipping over the sub instruction for now, the cmp.le instruction should be clear: it sets p6 to 1 if and only if $in_0 \leq in_2$ and sets p7 to 1 if and only if $in_0 \not\leq in_2$.

The cmp.ge.and.orcm is called a **parallel comparison**. It uses predicate registers p6 and p7 as both input operands and output operands. We now return to the sub instruction at the top of the function, which subtracts in1 from in2. On most architectures, a compare instruction is syntactic sugar for a subtract instruction: on IA-64 we sometimes need to explicitly simulate this fact. We perform the subtraction so that we can compare our result to 0. If $in_1 \geq in_2$, then loc1 will take on a non-negative value; otherwise it will take on a negative value. The reason we perform this explicit subtraction rather than comparing directly is that parallel comparisons require one operand to be r0; i.e., they require us to be comparing to 0.

The greater-than-or-equal comparison executes as usual. The result then is combined with the old values of p6 and p7 according to the parallel comparison type, in this case "and.orcm". This says that the first predicate register, p6, should be "and"ed with the result of the comparison and that the second predicate register, p7, should be "or"ed with the logical complement of the result of the comparison. This is explained in table 3.3. There are some things to pay special attention to in table 3.3: firstly that p6 and p7 are always complementary, a property which would not be preserved under some other parallel comparison types (e.g., plain "and" rather than "and.orcm"); secondly, and most importantly, that the resulting table ends up looking exactly like a logical "and" table, which is what we want, i.e., that p6 is true if and only if the original comparison *and* current comparison are true.

Valid parallel comparison types, which are explained further in the IA-64 ISA [Int06b], are: or, and, or.andcm, orcm, andcm, and.orcm. Comparison types with only one component, i.e., without a "dot", apply the same logical operator to both predicate registers.

Note that all comparison instructions, including parallel comparisons, can themselves be predicated. However, for non-parallel comparisons, the output predicates are set to 0 if the predicate is false, due to an oddity in the architectural design. Parallel comparisons do not modify the output predicates if the instruction predicate is false.

```
        cmp.lt                  p6 = 100,r14    // check if r14 < 100
        (p6) br.dpnt.many               else    // if r14 ≥ 100, branch
        ⋮
        do the "if" case
        br.sptk.many                    a       // finished with the "if" case
else:   ⋮
        do the "else" case
a:      both cases have merged now
```

Figure 3.5: Using predicated branches for if-then-else constructs.

3.3.2 Conditionals

As we've seen in section 3.3.1, predicated instructions can be used to perform conditionals. However, if two paths are greatly divergent in their execution, it may be prudent to use a conventional conditional branch instead of a single code path of predicated instructions.

Figure 3.5 gives an abstract example of using conditional branches to implement if-then-else constructs. The trade-off between using a single predicated code path or using two independent code paths depends on the lengths of each code path. The branch shown above is a **IP-relative branch**, meaning the target is fixed and relative to the instruction pointer. If the branch is predicted correctly, this branch carries no penalty. If the branch is not predicted correctly, the penalty is between 7 and 15 cycles, depending on depending on how many branches have been introduced to the pipeline since the mispredicted branch.

This is also our first use of a branch prediction hint other than "sptk". For branches which are not predicted, "sptk" (statically predicted to be taken) should be used. For predicted branches, the use of "dptk" (predicted to be taken, but less confidently so, using dynamic run-time prediction information) or "dpnt" (as dptk, but not taken) are generally recommended, unless the programmer has extreme confidence in whether a branch will be taken or not.

3.3.3 Loops

Conditional branches can be used to implement loops, as on conventional architectures. That is sometimes the best way to implement a loop, particularly a nested loop, but it will not be discussed in this section. Regardless of the method used to implement a loop, there are some guidelines that must be followed on the Itanium 2 to produce efficient code.

- Single-cycle loops should be avoided. The branch prediction stage of the Itanium 2 pipeline will stall for one cycle if it has seen 3 consecutive branches, i.e., 3 branches within 3 cycles. This means it takes 4 cycles to execute 3 iterations of a single-cycle loop. If a branch can be executed in a single cycle, it should be unrolled into a 2-cycle loop, as described in

section 5.1.

- Loops should generally use static prediction hints. For ctop and cloop branches, explained below, this is the norm. For 2-cycle loops, even those which do not use ctop or cloop branches, static branches are recommended: using a dynamic branch within a 2-cycle loop will cause a one cycle stall per iteration. I.e., it will take 3 cycles to execute each iteration of the loop.

- Loops should start on 32-byte boundaries. **Any branch to any target which is not 32-byte–aligned will cause a stall of 2 cycles**. This stall is typically not important for branch targets which are not loops. Guaranteeing this property will not be possible until we finish chapter 4 and can predict which instructions will be on 32-byte boundaries and which won't. We will revisit the issue at that point.

There are two classes of special loop instructions aimed at writing typical loops.

Counted loops

Counted loops, roughly corresponding to **for** loops found in higher-level languages, use the special ar.lc (loop count) and ar.ec (epilogue count) registers. Note that while the ar.ec register is part of the ar.pfs register and automatically saved by an alloc instruction, the ar.lc register is not. It is a callee-saved register and must be saved to a local register before being changed.

Chapter 4

Bundles

4.1 Bundle formats

As mentioned prior, an instruction in IA-64 is typically 41 bits in length, although a double-wide 82-bit "long" instruction can be used. Three 41-bit instructions (or one 41-bit instruction and one 82-bit instruction) are bundled together into a single instruction "bundle". An extra 5 bits are used to designate a template, making an instruction bundle 128 bits in length.

The 128-bit instruction bundle should be treated as atomic: it is not possible for IA-64 processors to issue individual instructions within a bundle. It is not specified how many bundles are issued at once: Itanium 2 processors issue two bundles per clock cycle, though any assembly code written for the IA-64 architecture should not make any assumptions about how many bundles are issued in parallel.

Table 4.1 lists the available bundle formats, simplified from table 3–10 in [Int06a]. The three letters stand for the execution unit associated with the instruction in the given "slot" in the bundle. For instance, the bundle template MBB executes one memory-unit instruction and two branch-unit instructions in parallel. Specifically, the letters designate as follows:

0 MII	1 MII,	2 MI,I	3 MI,I,
4 MX	5 MX,	6 –	7 –
8 MMI	9 MMI,	a M,MI	b M,MI,
c MFI	d MFI,	e MMF	f MMF,
10 MIB	11 MIB,	12 MBB	13 MBB,
14 –	15 –	16 BBB	17 BBB,
18 MMB	19 MMB,	1a –	1b –
1c MFB	1d MFB,	1e –	1f –

Table 4.1: Bundle formats defined by IA-64.

M — Memory unit. Can be used for loads/stores and also ALU "A-unit" integer instructions

I — Integer unit. Can be used for ALU "A-unit" integer instructions and also non-ALU "I-unit" integer instructions

F — Floating-point unit. Used for some integer instructions, such as integer multiplication

B — Branch unit. Used for branches and hinting instructions

X — Long double-wide instruction. Depending on the nature of the instruction it consumes either an I-unit or a B-unit

A comma in a bundle template is a "stop", an explicit serialization of execution of instructions.

Every sequence of instructions must be packaged into bundles and executed as bundles. The performance of IA-64 assembly code is directly dependent on how the instructions are bundled: since Itanium processors contain no out-of-order execution or implicit scheduling, the instructions are issue and executed exactly as they are packaged. The assembler will automatically package instructions into bundles, but it will not do any reordering either. If three load instructions are given in sequence, for instance, the assembler will naïvely break them across two bundles, putting two in the first bundle with a no-op and the remaining in the second bundle. This is not efficient and, consequently, it is imperative that the programmer be aware of how instructions will be bundling so that he or she can manually reorder instructions.

The assembler will allow the programmer to explicitly indicate which bundle template to use by given a compiler directory such as, e.g., .mmi. We will not explicitly use these except for pedagogical purposes, though we will from now always be cognizant of how instructions will be bundled.

The list given in table 4.1 may be daunting as it must always be kept in mind when writing assembly code, and especially as there are many template combinations which are not available. There are a few simple rules to keep in mind:

- Always mentally group instructions together in bundles of 3. You should be able to determine where the start of a new bundle is by looking at a sequence of instructions.

- Pursuant to the point above, avoid using long instructions. The use of a long instruction allows only two instructions to be issued within a bundle, which is not an efficient use of resources. Occassionally it is not reasonably avoidable.

- Pure memory instructions (loads and stores) should always be given at the very top of a bundle.

- ALU instructions should be given immediately after memory instructions. These can be slotted either in M or I slots, which appear at the front of the bundle templates.
- Branch instructions should always be given at the very end of a bundle. Further, they should, if possible, be moved to the very end of a sequence of bundles. The reason is that, if taken, the bundle introduces an implicit stop, which has performance implications. It is efficient to move a branch as far down as possible, right before an explicit stop, such that a sequence of code contains only one stop rather than two stops.

The use of intra-bundle stops is only for code density: it offers absolutely no performance benefit at all to have one bundle with a stop in it as opposed to two bundles with some no-ops.

Part II

Optimizations

Chapter 5

Rotating registers

Figure 5.1 gives the source code to our first function using rotating registers. In this case we implement the most literal definition of the Fibonacci function, namely as follows:

$$\mathit{fib}(n) = \begin{cases} 1, & \text{if } n \leq 1 \\ \mathit{fib}(n-1) + \mathit{fib}(n-2), & \text{if } n > 1 \end{cases}$$

The first thing to note is the alloc instruction. The rightmost argument is 8, indicating that we are allocating 8 rotating general purpose registers. In actuality we are only using 3, but the IA-64 architecture requires that they be allocated in groups of 8. We are also allocating 7 local registers which we never use. The assembler requires that the number of rotating registers allocated be less than or equal to—actually a subset of—the input and local registers. As we only have 1 input register, having any fewer than 7 local registers would cause an error.

We then make assignments to some "application registers", prefixed with ar, in the same category as the ar.pfs register we have seen previously. The ar.lc register is the "loop counter" and is required to use a rotating counted loop. It is callee-saved, hence the requirement to save to ar_lc_out. ar.ec is the "epilogue counter" and indicates how many iterations the loop will go through after iteration has completed. For our simple loop we require no epilogue. For reasons that are not clear, the epilogue counter is caller-saved, not callee-saved like the loop counter.

Unlike with general-purpose registers, predicate registers—as well as floating-point registers—have a fixed rotation size and thus the programmer is not allowed to indicate how many predicate registers will rotate through the course of a loop. Registers p16 through p63 are rotating. As we use p16 in the first iteration of the loop, we clear out all the rotating registers. Then, to initialize the loop, we set r32 and r33, our initial values for calculating Fibonacci.

The first iteration of the loop will do nothing since predicate register p16 is set to false. However, the br.ctop instruction set p63 to be true. Immediately after that, it will rotate all registers forwards: r32 becomes r33; r33 becomes

```
define(ar_pfs_old, r9)
define(ar_lc_old,  r10)
        .proc fib
fib:    .prologue
        alloc       ar_pfs_old = ar.pfs,1,7,0,8
        mov          ar_lc_old = ar.lc
        mov              ar.lc = in0              // set up the number of iterations
        mov              ar.ec = 0                // no epilogue needed
        mov             pr.rot = 0                // clear the rotating predicates
        mov                r32 = 1                // fib(0) = 1
        mov                r33 = 1                // fib(1) = 1
        ;;
loop:   .body
        (p16) add          r32 = r33,r34          // fib(n) = fib(n-1) + fib(n-2)
        br.ctop.sptk.few       loop
        ;;
        .restore sp
        mov               ret0 = r32
        mov              ar.lc = ar_lc_old
        mov             ar.pfs = ar_pfs_old
        br.ret.sptk            rp
        .endp fib
```

Figure 5.1: The **fib** function of the source file `fib.m`, which computes the Fibonacci number in the most naïve way.

r34; and so on. Most importantly for our purposes, p63 wraps around to become p16. This means that the next time through the loop, p16 will be true. It will then add r33 and r34 together. These registers used to be called r32 and r33, respectively, and were both initialized to the value of 1. Hence r32 will take on a value of 2. The next time through the loop, r33 will thus have a value of 2, r34 will have a value of 1, and r35 (which will never be used) will have a value of 1. Each iteration through the loop will decrement ar.lc: once it has reached 0, it will set p63 to 0 and start decrementing ar.ec. Once ar.ec has reached 0 as well, which it was initialized to in our case, the loop will finish. After n iterations of the loop, it is clear to say that register r32 will have value $fib(n)$, as desired.

The **main** function for the Fibonacci program is given in figure 5.2. The use of quotes in the define macros is required to redefine macros that have already been assigned. As we are using the 64-bit ABI, argv[1] is offset 8 bytes from argv[0]. The rest of **main** should require no explanation.

Figure 5.3 gives the running times for three implementations of calculating Fibonacci. Note that even in the most naïve assembly implementation we are considerably faster than the C implementation compiled under maximum optimization. All runs were timed on a 900MHz Itanium 2 processor. The figure demonstrates the predictability of the IA-64 architecture. The `gcc` compiler (version 4.2.3) foolishly produced a stop in the inner loop and thus we should predict exactly 2 cycles per iteration—i.e., 2 seconds to perform 900 million iterations at 900MHz—which is exactly what was seen. Similarly, for `fib2.m`, we predict 2 cycles per iteration with the number of iterations being half of n—i.e., 1 second to perform 900 million iterations at 900MHz—which is exactly what was seen.

As explained in section 5.1, `fib.m` theoretically requires $\frac{4}{3}$ cycles per iteration, which is what we see experimentally.

5.1 Loop unrolling

Normally we should expect that the inner loop of *fib* in figure 5.1 takes 1 cycle per iteration. However, there is a restriction of the Itanium 2's pipeline: should the front-end encounter branches on three consecutive cycles, it will stall by one cycle. In other words, the number of cycles it takes to perform n consecutive branches is $n + \lfloor \frac{n}{3} \rfloor$. For a large n we take this to be $\frac{4}{3}$ cycles per branch [Int04].

The solution to this is performance problem is partial loop unrolling, a common optimization technique. If we can restructure the code such that we perform two cycles of work per iteration, but only half as many iterations, we hit our minimum of time spent. Figure 5.4 provides the code for this.

We start with the original naïve loop, which contains only one instruction: add r32=r33,r34. Due to how register rotation works, this iteration's r33 is last iteration's r32, and this iteration's r34 is last iteration's r33 which was r32 two iterations ago. We can rewrite this as $f_n = f_{n-1} + f_{n-2}$, where n is the current iteration. Keeping in mind that the registers will rotate each iteration, we can see the values of our registers through each iteration, shown in table 5.1.

```
define('ar_pfs_old',        loc0)
define('ar_lc_old',         loc1)
define('gp_old',            loc1)
define('rp_old',            loc2)
                .global main
                .proc main
main:           .prologue
                alloc       ar_pfs_old = ar.pfs,2,4,3,0
                add               in1 = 8,in1                    // advance to argv[1]
                mov              out1 = 0
                mov            gp_old = gp
                mov            rp_old = rp
                ;;
                .body
                ld8              out0 = [in1]                     // n = strtol(argv[1], 0, 0)
                mov              out2 = 0
                br.call.sptk       rp = strtol
                mov                gp = gp_old
                mov              out0 = ret0
                br.call.sptk       rp = fib                       // fib(n)
                add              out0 = @gprel(formatString),gp
                mov              out1 = ret0
                br.call.sptk       rp = printf
                .restore sp
                mov                gp = gp_old
                mov                rp = rp_old
                mov           ar.pfs = ar_pfs_old
                br.ret.sptk          rp
                .endp main
                .sdata
formatString:   stringz          "%d\n"
```

Figure 5.2: The **main** function of the source file `fib.m`.

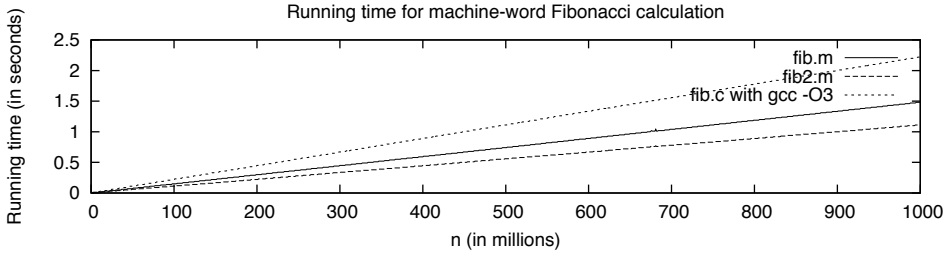

Figure 5.3: Running times for three separate implementations of a simple Fibonacci number calculator on a 900MHz Itanium 2.

```
define(ar_pfs_old, r9)
define(ar_lc_old,  r10)
        .proc fib
fib:    .prologue
        alloc          ar_pfs_old = ar.pfs,
        mov             ar_lc_old = ar.lc
        mov                ar.ec = 0
        mov               pr.rot = 0
        cmp.eq             p8,p9 = 0,in0
        add                  in0 = -1,in0        // we need to calculate
                                                 // ⌊(n-1)/2⌋
        ;;
        (p8) mov            ret0 = 1             // if n = 0, return 1
        shr.u                r14 = in0,1         // n/2
        and                  r15 = 1,in0         // not (n mod 2)
        ;;
        (p8) mov          ar.pfs = ar_pfs_old    // if n = 0, return 1
        (p9) mov           ar.lc = r14           // otherwise do half the iterations
        add                  r34 = 1,r15         // f_{n-3}
        add                  r33 = 2,r15         // f_{n-2}
        cmp.eq             p6,p7 = 1,r15         // check if n is even or odd
        ;;
        (p6) mov             r32 = 8             // if odd, f_n = 8
        (p7) mov             r32 = 5             // if even, f_n = 5
        ;;
loop:   .body
        (p16) shl            r14 = r33,1
        (p16) add            r34 = r34,r35       // f_{n-3} = f_{n-4} + f_{n-5}
        ;;
        (p16) add            r32 = r14,r34       // f_n = 2f_{n-2} + f_{n-3}
        br.ctop.sptk.few           loop
        ;;
        .restore sp
        mov                 ret0 = r33
        mov                  ar.lc = ar_lc_old
        mov                 ar.pfs = ar_pfs_old
        br.ret.sptk                rp
        .endp fib
```

Figure 5.4: The source file `fib2.m`, which performs two calculations per loop and hence one cycle per n.

	Iteration 1	Rotation 1	Iteration 2	Rotation 2	Iteration 3
r35	$-$	b	b	a	a
r34	b	a	a	$a+b$	$a+b$
r33	a	$a+b$	$a+b$	$2a+b$	$2a+b$
r32	$a+b$	$-$	$2a+b$	$-$	$3a+2b$

Table 5.1: Values of registers through a loop rotating registers, where each iteration we execute the instruction add r32=r33,r34.

Bibliography

[Fre10] Free Software Foundation, *GDB user manual*, 2010, `http://www.gnu.org/software/gdb/documentation/`.

[Int01a] Intel Corporation, *Intel Itanium processor-specific application binary interface (ABI)*, 2001, `http://download.intel.com/design/Itanium/Downloads/245370.pdf`.

[Int01b] Intel Corporation, *Itanium software conventions and runtime architecture guide*, 2001, `http://www.intel.com/design/itanium/downloads/245358.htm`.

[Int04] Intel Corporation, *Intel Itanium 2 processor reference manual*, 2004, `http://download.intel.com/design/Itanium2/manuals/25111003.pdf`.

[Int06a] Intel Corporation, *Intel Itanium architecture software developer's manual (volume 1): Application architecture*, 2.2 ed., 2006, `http://www.intel.com/design/itanium/manuals/245317.htm`.

[Int06b] Intel Corporation, *Intel Itanium architecture software developer's manual (volume 3): Instruction set reference*, 2.2 ed., 2006, `http://www.intel.com/design/itanium/manuals/245319.htm`.