Assembly Language Programming Tools for the IA-64 Architecture

Ady Tal, Microprocessor Products Group, Intel Corporation
Vadim Bassin, Microprocessor Products Group, Intel Corporation
Shay Gal-On, Microprocessor Products Group, Intel Corporation
Elena Demikhovsky, Microprocessor Products Group, Intel Corporation

Abstract

The IA-64 architecture, an implementation of Explicitly Parallel Instruction Computing (EPIC), enables the compiler to exercise an unprecedented level of control over the processor. IA-64 architecture features maximize code parallelism, enhance control over microarchitecture, permit large and unique register sets, and more. Explicit control over parallelism adds a new challenge to assembly writing, since the rules that determine valid instruction combinations are far from trivial, introducing new concepts such as bundling and instruction groups.

This paper describes Intel's IA-64 Assembler and IA-64 assembly assistant tools, which can simplify IA-64 assembly language programming. The descriptions of the tools are accompanied by examples that use advanced IA-64 features.

INTRODUCTION

The IA-64 architecture overcomes the performance limitations of traditional architectures and provides maximum headroom for future development. Intel's innovative 64-bit architecture allows greater instruction-level parallelism through speculation, predication, large register files, a register stack, advanced branch architecture, and more. 64-bit memory addressability meets the increasingly large memory footprint requirements of data warehousing, e-Business, and other high-performance server and workstation applications. Significant effort in the architectural definition maximizes IA-64 scalability, performance, and architectural longevity.

In the 64-bit architecture, the processor relies on the programmers or the compiler to set parallelism boundaries. Programmers can decide which instructions are executed in each cycle, taking data dependencies and availability of microarchitecture resources into account. Assembly can be the preferred programming language under the following situations: when learning new computer architectures in depth; when programming at a low level, such as that required for BIOS, operating systems, and device drivers; and when writing performance-sensitive critical code sections that power math libraries, multimedia kernels, and database engines.

Intel developed the Assembler and the Assembly Assistant in order to aid assembly programmers in rapidly writing efficient IA-64 assembly code, using the assembly language syntax jointly defined by Intel and Hewlett-Packard*.

The Intel® IA-64 Assembler is more than an assembly source code-to-binary translator. It can take care of many assembly language details such as templates and bundling; it can also determine parallelism boundaries or check for those given by assembly programmers. The assembler can also allocate *virtual registers* and so enable assembly programmers to write code with symbolic names, which are replaced automatically with physical registers.

The Assembly Assistant is an integrated development tool. It provides a visual guide to some IA-64 architecture features permitting assembly programmers to comprehend the workings of the processor. The Assembly Assistant has three main goals: to introduce the architecture to new assembly programmers; to make it easier to write assembly code and use the Assembler; and to help assembly programmers get maximum performance from their code. This last task is achieved through *static analysis*, a *drag-and-drop* interface for *manual optimization*, and through *automatic optimization* of code segments.

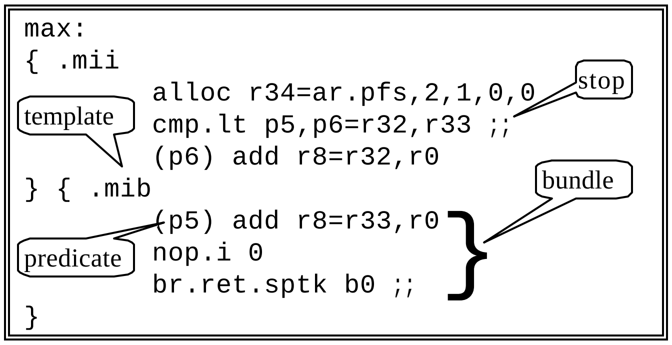
IA-64 ARCHITECTURE FEATURES FOR ASSEMBLY PROGRAMMING

The IA-64 architecture incorporates many features that enable assembly programmers to optimize their code for efficient, high-sustained performance. To allow greater instruction-level parallelism, the architecture is based on principles such as explicit parallelism, with many execution units, and large sets of general and floating-point registers. Predicate registers control instruction execution, and enable a reduction in the number of branches. Data and control speculation can be used to hide memory latency. Rotating registers allow low-overhead software pipelining, and branch prediction reduces the cost of branches.

The compiler takes advantage of these features to gain a significant performance speedup. Branch hints and cache hints enable compilers to communicate compile-time information to the processor. New compiler techniques are developed to use these features, and IA-64 compilers will continue gaining speedups utilizing these features in innovative ways.

This abundance of architecture features and resources makes assembly writing a challenging task for assembly programmers.

The IA-64 architecture requires that the instructions are packed in valid bundles. Bundles are constructs that hold instructions. Bundles come in several templates, restricting the valid combinations of instructions and defining the placement of boundaries (*stops*) for maximum parallelism. Each instruction can be placed into a specific slot in a bundle, according to the template and the instruction type. For example, the instruction *alloc r34=ar.pfs,2,1,0,0* appearing in the example below, can only be placed in an M slot of a bundle. The template defined in the example below for the first bundle is .mii meaning that the first slot can only be taken by an instruction that is valid for an M slot, and the following instructions must be valid for I slots. When no useful instruction can be placed in the bundle due to template restrictions, a special *nop* instruction, valid for the slot, must be used to fill the bundle (as observed in the case of the second slot in the second bundle; a nop.i was placed in an I slot).



```
max:
{ .mii
        alloc r34=ar.pfs,2,1,0,0
        cmp.lt p5,p6=r32,r33 ;;
        (p6) add r8=r32,r0
} { .mib
        (p5) add r8=r33,r0
        nop.i 0
        br.ret.sptk b0 ;;
}
```

Example 1: Code reflecting language syntax

Assembly programmers are expected to define groups of instructions that can execute simultaneously by inserting stops. If a stop is missing, then there is a chance that not all the instructions were meant to be executed in the same cycle. Such an instruction group may contain a dependent pair of instructions. For example, it may contain two instructions that write to the same register or a register write followed by a read of the same register.

The result of parallel execution of dependent instructions, even though not necessarily adjacent, is unpredictable and may vary in different IA-64 processor models. This situation is called a *dependency violation*. To avoid it, assembly programmers have to place the two instructions in different groups by inserting a stop.

In Example 1 we can see a *stop* after the *cmp* instruction. This *stop* will ensure that the *cmp* will not be executed in parallel with the following *add*, and it enables the *add* to use the predicate *p6* written by the *cmp* instruction without producing a *dependency violation*.

THE IA-64 ASSEMBLER

The IA-64 Assembler enables many capabilities beyond traditional assemblers. In addition to assembling, it implements full support of all architecture and assembly language features: bundles, templates, instruction groups, directives, symbols' aliases, and debug and unwind information.

Writing assembly code with bundles and templates is not trivial. Assembly programmers must know the type of execution unit for each instruction, be it memory, integer, branch, or floating-point. Another important element of the assembly language is the *stop* that separates the instruction stream into groups.

The IA-64 assembly language provides assembly writers with maximum control through the use of two modes for writing assembly code: *explicit* and *automatic*.

When writing in explicit mode, assembly programmers define bundle boundaries, specify the template for each bundle, and insert stops between instructions where necessary. The Assembler only checks that the assembly code is valid and has the right parallelism defined. This mode is recommended for expert assembly programmers or for writing performance-critical code.

The automatic mode significantly simplifies the task of assembly writing while placing the responsibility for bundling the code correctly on the Assembler. The Assembler analyzes the instruction sequence, builds bundles, and adds stops.

```
ld4   r4 = [r33]
add   r8 = 5, r8
mov r2 = r56
add   r32 = 5, r4
mov   r3 = r33
```

Example 2: Original user code

```
{ .mii
      ld4   r4 = [r33]
      add   r8 = 5, r8
      mov   r2 = r56 ;;
}
{ .mmi
      nop.m 0
      add   r32 = 5, r4
      mov   r3 = r33 ;;
}
```

Example 3: Code created after assembly with automatic mode

In the example above we can see how the user can write simple code, which the Assembler will then fit into bundles. The Assembler also adds *nop* instructions for valid template combinations and *stops* as needed to avoid any possibility of dependency violations. (A stop bit was added at the end of the first bundle to avoid a dependency violation between the first instruction and the next to last instruction on r4.)

Parallelism

One of the tasks of the Assembler is to help assembly programmers define the right parallelism boundaries. The Assembler analyzes the instruction stream, taking into consideration the architectural impact of each instruction and any implicit or explicit operands involved in the execution. However, it is hard to do the complete program analysis needed in order to detect all these conditions, statically. Consider a common case in IA-64 architecture where two instructions writing to the same register may be predicated by mutually exclusive predicates, as shown in the Example 4.

```
cmp.ne p2,p3 = r5,r0 ;;
      …
(p2) add r6 = 8, r5
(p3) add r6 = 12, r5
```

Example 4: Predicate relation

The Assembler can identify this case and ignore the apparent dependency violation between the two *add* instructions on *R6*. In this case, the compare instruction, which defines the pair of predicates, precedes their usage. However, more complicated cases may exist. For example, consider a case in which there is a function call between predicates set and usage. In this case, assembly programmers may know that the called function doesn't alter the predicates' values, but there is no way for the assembler to deduce this information, if the function is in a different file.

Another type of information known only at run-time is the program flow at conditional branches. The processor automatically begins a new instruction group when the branch is taken, and a dependency violation may occur only on the fall-through execution path.

When writing in explicit mode, assembly programmers are responsible for stops. The Assembler simply checks the code and reports errors, even when it finds only potential dependency violations. In this mode, to avoid false messages, assembly programmers can add annotations describing predicate relations at that point of the procedure.

```
.pred.rel "imply", p1, p2
(p1) mov r5 = r23
(p2) br.cond.dptk Label1
add r5 = 8, r15
```

Example 5: User annotation

The relation "p1 implies p2", in Example 5 means that if p1 is true then p2 is also true. Adding such a clue to the assembly code prevents a false dependency violation report between the second and fourth lines.

Automatic mode simplifies the programming tasks while delegating the responsibility for valid instruction grouping to the Assembler. In automatic mode, the source code contains no bundle boundaries. The Assembler ignores stops written by the assembly programmer; it builds bundles and adds stops according to the results of static analysis of instructions and program flow. In this mode, the code is guaranteed not to contain dependency violations.

The example below contains a dependency violation which is not immediately apparent. The first instruction writes to CFM, while the second instruction reads from CFM, resulting in a dependency violation. Using automatic mode, the dependency violation is automatically resolved.

```
br.ctop.dptk.many   l8
fpmin   f33=f1,f2
```

Example 6: Code containing dependency violation

```
{ .mib
      nop.m 0
      nop.i 0
      br.ctop.dptk.many   l8 ;;
}
{ .mfi
      nop.m 0
      fpmin   f33=f1,f2
      nop.i 0
}
```

Example 7: Code after automatic mode assembly

In the example above, observe how the Assembler detects a dependency violation on *CFM* between the instructions, and how it inserts a *stop* between them.

Virtual Register Allocation

Large register sets in the IA-64 architecture complement the unique parallelism features. Maintaining assembly code becomes harder when there is a need to track the assignment of many variables to registers. Modifying a procedure code might lead to variable reallocation.

The virtual register allocation (VRAL) feature solves these problems. VRAL allows assembly programmers to use symbolic names instead of registers, and it performs the task of register allocation in procedures.

To employ VRAL, assembly programmers must use a set of VRAL directives in order to communicate some register-related information to the Assembler. Assembly programmers assign groups of physical registers for virtual allocation, and they define the usage policy; i.e., whether they should be scratch registers or registers that are preserved across calls. The Assembler assigns some default families that many assembly programmers are likely to use, including integer, floating point, branch, and predicate. Assembly programmers can also isolate registers of the same type in subfamilies. For example, a user-defined family may include all local registers of a procedure.

Each symbolic name used in a procedure, called a virtual register, belongs to one of the register families. The assembly language allows redefinition of virtual registers' names, which is convenient when used in preprocessor macros.

VRAL analyzes the *control flow graph* of the procedure, and it calculates the registers' live ranges. An accurate control flow graph is very significant for this analysis. The Assembler provides appropriate directives to specify the targets of indirect branches and additional entry points. In order to find a replacement for each symbolic name, VRAL applies standard graph-coloring techniques.

The heuristic function used for allocation priorities considers both the results of the preceding analysis and the architecture constraints of registers' usage. Several physical registers may replace one symbolic name, and one physical register may be reallocated and utilized for several different symbolic names.

```
.proc foo
foo::
    alloc loc0=ar.pfs,2,8,0,0
.vreg.safe_across_calls r15-r21
.vreg.safe_across_calls loc3-@lastloc
.vreg.allocatable p6-p9
.vreg.family LocalRegs,loc3-@lastloc
.vreg.var LocalRegs,X,Y,Diff
      mov loc1=b0
      add X=in0,r0
      add Y=in1,r0 ;;
.vreg.var @pred,GT,LE
      cmp.gt GT,LE=X,Y ;;
      (GT) sub Diff=X,Y
      (LE) sub Diff=Y,X ;;
.vreg.redef GT,LE
      mov r8=Diff
      mov ar.pfs=loc0
      mov b0=loc1
      br.ret.dptk b0
.endp foo
```

Example 8: Code with virtual register

Consider the code in the Example 8 above. Starting with ***.vreg.safe_across_calls*** and ***.vreg.allocatable*** directives, we define the registers that are available for allocation. We then use the ***.vreg.family*** directive to define a family of virtual registers that will only be allocated from the local registers. We then define the virtual registers themselves and declare them to be part of the local registers' family defined earlier using the ***.vreg.var*** directive. The code itself then uses virtual registers X and Y instead of directly naming physical registers. The example also illustrates that virtual registers can be defined in the middle of the code, and then undefined with the ***.vreg.redef*** directive to allow reuse of symbolic names (used most frequently in macros).

All symbolic names defined with the directive ***.vreg.var*** are replaced with physical registers assigned for allocation by the directives ***.vreg.allocatable*** and ***.vreg.safe_across_calls***. For this simple example, in the current version of the Assembler, the registers chosen were X=R37, Y=R38, GT=P6, and LE=P7.

In order to effectively use VRAL, we plan to emit the allocation information to allow debugging using symbolic names. This enables the debugger to show the value of the symbolic name, even if the value is represented by different registers in different parts of the code. Also, by emitting the allocation information, code optimizations without the allocation constraints will be enabled.

THE IA-64 ASSEMBLY ASSISTANT

In order to further assist IA-64 assembly developers, we designed and implemented a unique development environment to have the following:

1. a tool to reduce the steep learning curve for IA-64 assembly programming and to introduce IA-64 architecture using assembly programming
2. a user friendly environment for IA-64 assembly development
3. an environment for analyzing and improving the performance of assembly-code fragments

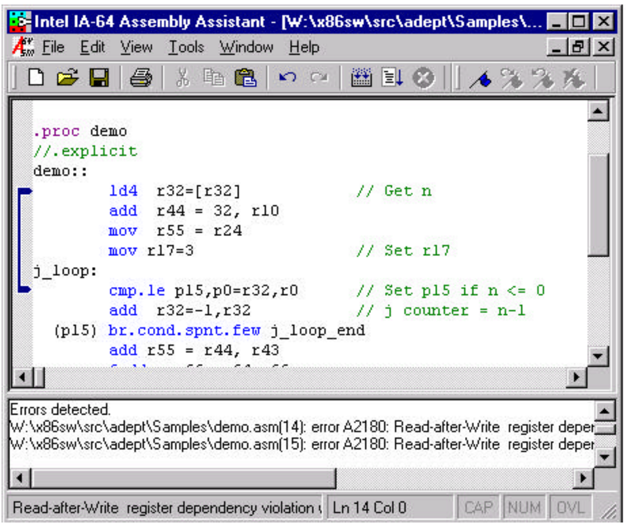
The Assembly Assistant delivers a comprehensive solution for assembly code developers, assembly language-directed editing, tools that aid the creation of new code, error reporting, static performance analysis, and manual and automatic optimizations.

Other needs such as debugging or run-time performance analysis will be addressed when the Assembly Assistant is integrated with other tools that supply these features.

The next sections describe the Assembly Assistant's editing, assembling, and analysis capabilities in detail and examine the unique features that the Assembly Assistant provides to IA-64 assembly programmers.

Editing and Assembling

The Assembly Assistant provides *syntax-sensitive coloring* that includes all components of assembly code: directives, instructions, comments, templates, and bundling. Every valid instruction format (including completers) is colored to mark it visually as a valid instruction.


Intel IA-64 Assembly Assistant - [W:\x86sw\src\adept\Samples\...
File
Edit
View
Tools
Window
Help

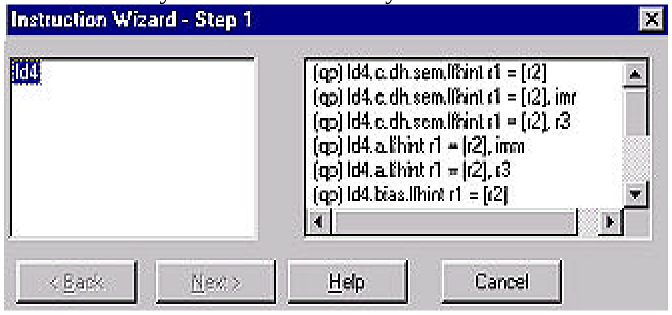
```
.proc demo
//.explicit
demo::
        ld4  r32=[r32]            // Get n
        add  r44 = 32, r10
        mov  r55 = r24
        mov r17=3                 // Set r17
j_loop:
        cmp.le p15,p0=r32,r0      // Set p15 if n <= 0
        add  r32=-1,r32           // j counter = n-1
   (p15) br.cond.spnt.few j_loop_end
        add r55 = r44, r43
```

```
Errors detected.
W:\x86sw\src\adept\Samples\demo.asm(14): error A2180: Read-after-Write  register deper
W:\x86sw\src\adept\Samples\demo.asm(15): error A2180: Read-after-Write  register deper
```

```
Read-after-Write  register dependency violation   Ln 14 Col 0   CAP  NUM  OVL
```

Figure 1: Source code window

The IA-64 instruction set is very rich, and the same mnemonic may be used in a variety of instructions when combined with different types of operands. The Assembly Assistant provides an *Instruction Wizard* to help assembly programmers select the appropriate instruction with the right set of completers and operands. It allows assembly programmers to choose between instructions in different variations, and it provides a template to select the operands and activate on-line help about the instruction. The example in Figure 2 illustrates how the instruction wizard allows you to choose a specific ld4 form (step 1) and then easily apply the correct completers (step 2). The Help includes some information from ref [1], ref [5], and the *IA-64 Assembly Language Reference Guide*.



```
ld4
```

```
(qp) ld4.c.dh.sem.llhint r1 = [r2]
(qp) ld4.c.dh.sem.llhint r1 = [r2], imr
(qp) ld4.c.dh.sem.llhint r1 = [r2], r3
(qp) ld4.a.llhint r1 = [r2], imm
(qp) ld4.a.llhint r1 = [r2], r3
(qp) ld4.bias.llhint r1 = [r2]
```

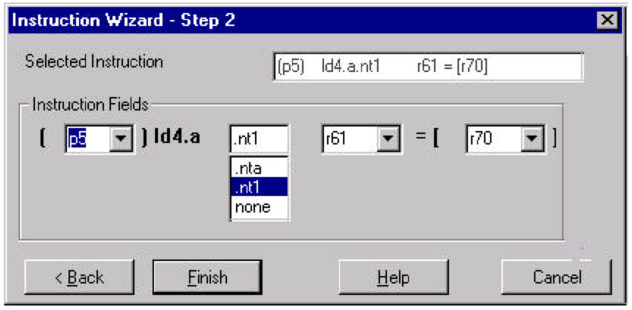


Figure 2: Instruction Wizard

Many assembly programmers need an easy way to interface assembly with other high-level languages such as C* and C++*. Procedures written in assembly must adhere to the *IA-64 Software Conventions* in order to execute correctly. The Assembly Assistant will generate the procedure linkage code given a high-level language-like function prototype. The *Procedure Wizard* generates a procedure template with information that assembly programmers can use to access input parameters and results (see Figure 3).

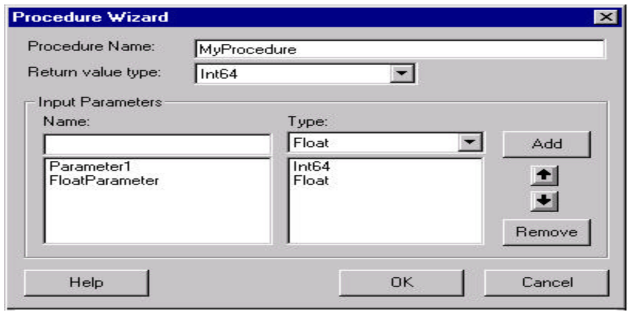


Figure 3: Procedure Wizard

The goal of the *Auto Code Wizard* is to provide an option to retain, customize, and reuse favorite code templates. An example of such a template is the code for integer multiplication. (It is provided as an example in the tool.) The IA-64 architecture does not have a multiplication instruction for general registers so the instruction for floating-point registers must be used. Assembly programmers could write an Auto Code template that moves values from general to floating-point registers, multiplies them, and moves the result back to a general register.

In general, using the source code editor together with wizards and the context-sensitive help provides a rich set of customizable tools to help both beginners and experienced IA-64 assembly programmers.

While browsing errors after compilation is a common task in all development environments, the Assembler identifies a special set of errors called dependency violations (see above). These errors can produce treacherous results, and special care is required while treating them. The difficulty is that these errors involve two instructions that may be distant from one another. When the error in the error view at the bottom of the screen is highlighted, it displays connected pointers pointing to the offending pair of instructions in the source code window (see Figure 1).

ANALYSIS WINDOW

The Assembly Assistant provides a static analysis as a guide to help assembly programmers improve performance. In this section, we discuss the *analysis window*. This window helps assembly programmers understand, browse, analyze, and optimize their assembly code.

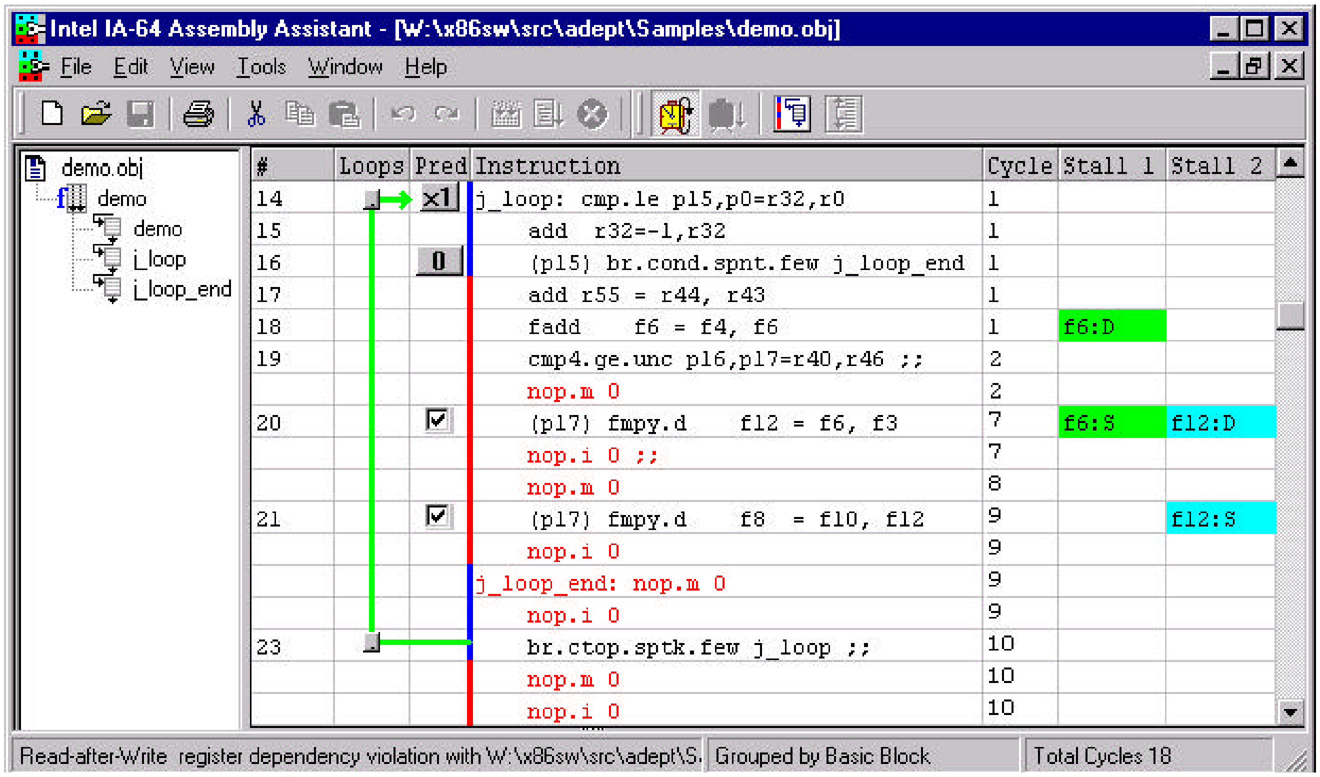
The Assembly Assistant uses static performance analysis on a function-by-function basis, without any dynamic information on the program behavior (such as register values and memory access addresses). Fast performance simulation of instruction sequences is used in order to obtain the performance information.

The main performance information displayed in the *analysis view* is *cycle count* and *conflicts*. The cycle count is the estimated relative cycle number in which the instruction enters the execution pipe in the performance simulation. This number is relative to the beginning of the function or selected block. Usually the execution path doesn’t utilize the full capacity made possible by the IA-64 architecture. Conflict indicators in the *stall* columns show the reasons for extra cycles in the execution path and processor stalls.

Assembly programmers analyze the conflicts and modify their code accordingly by manually moving an instruction earlier or later in the code sequence, selecting a different instruction sequence, etc. Automatic optimization attempts to find the best instruction scheduling. Optimizations are discussed in a later section.

As shown in Figure 4, the assembly source is presented along with line numbers, cycle counts, and conflicts, as discussed earlier. Conflicts are highlighted with different colors for each conflict, so assembly programmers can easily identify which instructions are involved in each conflict. In Figure 4, the cycle counts are based on a hypothetical machine model.

Another type of information is division of the instruction stream into several types of groups. The most interesting are bundles that are fetched together from memory and instruction groups, which assembly programmers define as candidates for parallel execution. Two additional groups for more advanced assembly programmers are issue groups (instructions that execute simultaneously) and basic blocks (for control flow analysis).


Intel IA-64 Assembly Assistant - [W:\x86sw\src\adept\Samples\demo.obj]
File
Edit
View
Tools
Window
Help
demo.obj
demo
demo
Loop
Loop_end

#	Loops	Pred	Instruction	Cycle	Stall 1	Stall 2
14		x1	j_loop: cmp.le p15,p0=r32,r0	1		
15			add r32=-1,r32	1		
16		0	(p15) br.cond.spnt.few j_loop_end	1		
17			add r55 = r44, r43	1		
18			fadd f6 = f4, f6	1	f6:D	
19			cmp4.ge.unc p16,p17=r40,r46 ;;	2		
			nop.m 0	2		
20		✓	(p17) fmpy.d f12 = f6, f3	7	f6:S	f12:D
			nop.i 0 ;;	7		
			nop.m 0	8		
21		✓	(p17) fmpy.d f8 = f10, f12	9		f12:S
			nop.i 0	9		
			j_loop_end: nop.m 0	9		
			nop.i 0	9		
23			br.ctop.sptk.few j_loop ;;	10		
			nop.m 0	10		
			nop.i 0	10		

Read-after-Write register dependency violation with W:\x86sw\src\adept\S. | Grouped by Basic Block | Total Cycles 18

Figure 4: Analysis window

Run-time information is not available during static analysis. The Assembly Assistant provides information about execution flow and predicate values. These values control the simulated execution path and may activate or deactivate instructions. As illustsrated in Figure 4, each predicated instruction is preceded by a checkbox. Marking the checkbox signifies that the qualifying predicate is true, and the instruction executes. This interface eases analysis and optimization of predicated blocks. The Assembly Assistant provides the means to control predicate values. In the future, the Assembly Assistant might also calculate predicate relations for use in the static analysis, and marking a single predicate as true or false will automatically determine the values for all of the predicates that are related to it.

The Assembly Assistant gives assembly programmers more extensive control. Assembly programmers can specify whether or not the branch is taken, and they can set the probability of taking the branch in the simulation. The Assembly Assistant uses this probability when simulating loops: it provides assembly programmers with the approximate time of loop execution together with *a-priori* knowledge of possible execution paths.

The analysis window provides more than just loop visualization. Assembly programmers may select the number of iterations to simulate. Selecting a single iteration provides performance information and shows any conflicts between the main body of the loop and the prologue code. Selecting two iterations also displays conflicts between the head and tail of the loop code section. Selecting more than two iterations provides the approximate execution time of the loop calculated by branch probabilities, as described above.

The assembly module may contain more than one function. To help assembly programmers navigate in the analysis window, the window is split into two panes, just like the Microsoft Explorer* window. The left tree pane contains a list of all the functions in the module, while the right pane displays one function's analyzed code. Clicking on a function name or icon in the tree pane displays the analysis of the selected function. Assembly programmers work with one function at a time, viewing in the tree pane the list of labels in the active function. This allows easy navigation inside the function.

We have described above how assembly programmers can analyze assembly code. But analysis is useless if assembly programmers cannot apply their insights to

* Other brands and names are the property of their respective owners.

improve the code. The Assembly Assistant allows them to do this.

Assembly programmers may want to move stalling instructions and solve conflicts. The Assembly Assistant provides a simple drag-and-drop interface so that instructions can be moved manually. It displays the instruction's move boundaries as defined by data dependencies. Assembly programmers can drop the instructions into their new position. Assembly programmers can also apply automatic optimizations that reschedule the instruction stream to improve performance.

After completing code optimization in the analysis window, the Assembly Assistant generates new, improved code in a new source window.

OPTIMIZATION AIDES IN THE IA-64 ASSEMBLY ASSISTANT

When optimizing assembly code, a tool such as the Assembly Assistant can generally use conventional compiler techniques. However, a key challenge to optimization is code analysis, since some of the information visible to the compiler does not exist or is hard to infer from such low-level representations. Examples include branch targets for indirect branches, calling conventions, memory disambiguation, and aliasing.

This information is critical for code speedup and maintenance of correct code. An assembly optimizer also has no choice but to deal with every feature of the architecture, whereas a compiler might choose not to use certain features or instructions; for example, system instructions.

The first and simplest solution is to leave optimizations to the assembly programmers but still help them with available analysis data and IA-64 processor-specific information. An advanced and friendly user interface enables assembly programmers to easily perform the optimizations.

The next level of automation requires assembly programmers to provide missing information. A user-friendly interface allows assembly programmers to interact with the optimizer to define branch targets and more. Using this information, a control flow graph is created and analyzed. Assembly programmers can also provide program behavior information such as branch probabilities, which direct the optimizer to bias its optimizations accordingly.

Automatic optimization is also used, but as mentioned earlier, it is somewhat limited due to the conservative approach to assembly-level optimizations.

Manual Optimization

Analysis provides many hints for manual optimization. The analysis of register live ranges helps assembly programmers better use the registers. The analysis of data flow provides the assembly programmers with suspected dead code, and it detects the use of registers that were not initialized in the analyzed code. Data flow analysis is also helpful when trying to attain optimal scheduling. Height reduction (the process of reducing the control dependence, for example as in ref. [4]) and strength reduction (ref. [3] p.435) are easier for assembly programmers to handle when all the dependency chain is analyzed automatically.

For software pipelined loops, automatic tracking of the rotating registers used in the loop helps assembly programmers to write modulo-scheduled loops. This can also greatly simplify modification of the code.

It is difficult to keep track of other machine resources (such as control registers, functional units, and more). The Assembly Assistant can automatically keep track of machine resources, and it can warn assembly programmers when machine resources are insufficient for the code. The Assembly Assistant shows an assembly programmer the penalty incurred by the code, and it suggests methods to overcome the limitations inherent in the microarchitecture.

To aid in speculation, when moving a load beyond ambiguous memory references or control dependencies, the Assembly Assistant shows assembly programmers the probable costs and benefits of the speculation. Load instructions that inhibit scheduling can also be identified and they can be suggested to assembly programmers as likely candidates for speculation.

Automatic Optimization

While assembly programmers are certainly capable of performing most optimizations that can be done automatically, other optimizations are difficult. This is due either to complexity or tedium. For example, scheduling the instructions for optimal performance is mostly a problem of brute force. Experienced assembly programmers who are also familiar with all the microarchitecture details can tweak the scheduling to get the best performance, but sometimes the only way to get the best scheduling is to simply try out all of the combinations. The testing would have to be repeated after every source code change. This is clearly a mission for an automated tool.

In automatic optimization mode, the Assembly Assistant schedules the instruction stream in a top-to-bottom issue-group-scheduling approach. Instructions are scheduled according to internal heuristics, taking into account critical path, code size, instruction priorities, utilization of machine resources, and more. Many possible templates are checked against the heuristics, and the best one is chosen for the issue group. The code in the example below will execute significantly slower than the same code after automatic optimization (~25%). Analyzing the example, we can observe that the instructions at lines 8 and 9 of the original code were moved up, and instructions on lines 3, 4, and 7 were moved down. This schedule was chosen considering the latencies of various functional units and in order to prevent unnecessary stalls.

```
Original code:
```

```
1  { .mii
2     ld4     r4 = [r33]
3     add     r8 = 5, r8
4     mov   r2 = r56 ;;
5  }
6  { .mii
7     add     r32 = 5, r4
8     mov     r3 = r33 ;;
9     add     r33 = 4, r3 ;;
10 }
11 { .mib
12    cmp4.lt p6,p7=r2,r33
13    nop.i 999
14    (p7)    br.cond.dpnt START
15 }
```

```
Code after automatic optimization:
```

```
1  { .mii
2     ld4            r4 = [r33]
3     mov            r3 = r33 ;;
4     add            r33 = 4, r3
5  }
6  { .mii
7     mov     r2 = r56
8     add            r8 = 5, r8 ;;
9     add            r32 = 5, r4
10 }
11 { .mib
12    cmp4.lt p6,p7=r2,r33
13    nop.i   999
14    (p7) br.cond.dpnt START
15 }
```

Example 9: A small code sample before and after automatic optimization

Optimal utilization of machine resources for parallel execution is very important, and actual results show that even code written by experienced assembly programmers can gain a speed-up of 6 – 8% from the Assembly Assistant's automatic scheduling. In the case of code written by inexperienced assembly programmers, the gain is likely to be much higher.

Assembly programmers can choose from various optimization schemes, actually changing the heuristics used. For example, scheduling for the smallest code size might incur significant penalties due to overloading of machine resources. By default, automatic optimization tries to address the issues most challenging to a human assembly programmer. It optimizes for better utilization of machine resources rather than concern itself with code size. However, this might result in inflated code and affect the instruction cache. Enabling various optimization schemes offers assembly programmers greater control over the automatic optimizations, and it allows expert assembly programmers to take advantage of automatic modes without losing flexibility.

Even for code that is not performance-critical, but still has to be written in assembly, automatic optimization can be valuable. It can be used either to speed up performance or to pack the code more tightly.

FUTURE ENHANCEMENTS FOR THE ASSEMBLY ASSISTANT

The Assembly Assistant is currently used by many IA-64 assembly programmers both beginners and experts to tune their code. We received many requests for more features and enhancements. The requests include a library of optimized special-purpose code (for example, integer divide and floating-point square root), more manual and automatic optimizations, visualization of registers' live ranges, etc. We are also looking into integrating the Assembly Assistant with other programming environments such as Microsoft Visual Studio* and the Intel® VTune™ performance analyzer.

CONCLUSION

It is strongly recommended that compilers be used in order to generate highly optimized code for the IA-64 architecture. The use of compilers also guarantees scalability and portability for future IA-64 implementations. However, we recognize the need of some developers to continue to use assembly code in their applications. We attempted to outline the difficulties faced by assembly programmers when writing for the IA-64 architecture, and we presented tools to alleviate or overcome these difficulties. The tools presented contribute to the following goals:

- Quickly familiarize assembly programmers with the new IA-64 architecture.

* Other brands and names are the property of their respective owners.

- Program in IA-64 assembly with relative ease.
- Provide a comprehensive development environment for assembly programming.
- Analyze and optimize assembly programs to utilize IA-64 unique features for optimal performance.

REFERENCES

[1] *IA-64 Application Developer's Architecture Guide*, Order number 245188.

[2] Analysis of Predicated Code, HPL-96-119.

[3] Steven S. Muchnick, *Advanced compiler design and implementation*, Morgan Kaufmann, San Francisco, California, 1997.

[4] "Control CPR: A Branch Height Reduction Optimization for EPIC Architectures, " in *Proceedings of the ACM SIGPLAN 99 Conference on PLDI*, Atlanta, Georgia 1999, pp.155-168.

[5] *IA-64 Assembler User's Guide*, Order number 712173.

AUTHORS' BIOGRAPHIES

Ady Tal received his M.Sc. degree from the Technion in 1990. He has been working for Intel Israel since 1996, and he leads the Optimizing Libraries development team. He was a member of the Assembler and Assembly Assistant development teams and has wide experience with all aspects of IA-64 architecture features and assembly optimization techniques. His e-mail is ady.tal@intel.com.

Vadim Bassin received an M.A. degree in computing from the Belorussian Radio-Engineering Institute in 1987. While he studied mostly hardware, he has worked only in software. He spent six years programming in real time and working on image processing in the Belorussian Science Academy and a small Israeli-American company before switching to network management. He finally found himself working at Intel on GUI tools for programmers. His e-mail is vadim.bassin@intel.com.

Shay Gal-On received his B.A. degree from the Technion in 1997. Since then he has lingered at Intel Israel, mainly working on assembly optimization techniques and bit manipulations' libraries. Professional interests run from optimizations to security. His e-mail is shay.gal-on@intel.com.

Elena Demikhovsky graduated from the Belorussian Radio-Engineering Institute with an M.Sc. degree in 1991. Since 1994, she has worked at Intel Israel as a software engineer. Elena has wide experience in the development of tools, especially the Assembler, for both the IA-32 and IA-64 architectures. Her e-mail is elena.demikhovsky@intel.com.