Instruction Level Parallelism in VLIW Processors

Charles Gray

May 9, 2003

Contents

1 Traditional approaches to ILP

Exploiting *instruction level parallelism* (ILP) is the main way of increasing processor performance. By executing instructions in parallel, processor performance can be increased without reducing the time taken to execute an individual instruction. The problem with parallelising instructions, of course, is the dependencies that exist between instructions. Traditionally the logic to detect dependencies and work around them has been implemented in hardware.

Pipelining is one of the most simple approaches to exploiting ILP. Pipelining involves overlapping the execution of multiple instructions through different phases. Where a dependency exists in the instruction stream a pipeline bubble, or NOP, must be inserted. Hardware ILP processors like RISC and CISC detect pipelines hazards such as these and insert pipelines bubbles to ensure that code executes correctly.

Super-scalar processors introduce the idea of multi-issue. Super-scalar processors include multiple pipelines so that multiple instructions can be issued every cycle. It is up to the hardware to decide that two instructions can safely be issued in parallel.

Register renaming is a common technique used to allow more instructions to execute simultaneously. Without register renaming there is a limit on the number of concurrent instructions based on the number of registers presented in the architecture's instruction set. Most RISC and CISC architectures have more internal registers and re-map architecture registers onto actual registers on a per instruction basis.

Branch instructions and pipelines generally do not mix well. A branch instruction causes a pipeline bubble because the destination address is not known at the time the following instruction is fetched. There are many hardware techniques employed to predict the outcome of branches so that pipelines can be kept as full as possible. To take further advantage of branch prediction speculation is used. Speculation involves executing instructions before it is known if they will violate a dependency. This allows instructions to be executed as early as possible, at the expense of possibly doing some unnecessary work.

These techniques generally fall under the topic of *scheduling*. The issue unit examines the instruction stream and schedules instructions into the processor to best take advantage of the hardware.

1.1 Problems with Super-scalar

The hardware size and complexity of modern super-scalar processors is a problem. The P4 architecture has two of its twenty pipeline stages devoted to moving data across the chip. The complexity of issue units capable of maintaining a high number of instructions per clock is such that the issue unit themselves need to be pipelined.

These problems have a negative impact on clock speed. Most ILP can be discovered statically by a compiler, and compiler techniques are increasingly important in optimising for modern super-scalar architectures. This has led some designers to look for other solutions to exploit ILP.

Floating-Point Unit	Integer ALU	Load/Store Unit	Branch Unit
div r10, r11, r12	nop	load r5, r2	nop
div r13, r14, r15	add r5, r5, 0x4	nop	nop
nop	nop	store r5, r2	jmp r10

Figure 1: Example of the 128-bit crusoe instruction format

2 The VLIW Approach

Very long instruction word (VLIW) systems are designed to achieve a high level of ILP with simpler hardware. Using simpler hardware means you can build processors with deeper pipelines and more execution units. To allow simpler hardware but maintaining high levels of ILP the VLIW technique is to expose more details of the architecture.

Exposing details of the architecture allows for a lot of simplification. If a compiler knows the latency of every instruction then the hardware need not detect pipeline hazards. Inserting NOP instructions to overcome hazards becomes the job of the compiler.

VLIW also reduces hardware complexity by making instruction scheduling and speculation explicit. VLIW architectures need only execute in-order. This removes any need for re-order buffers or out-of-order execution.

VLIW is so named because of the instruction format. Rather than the typical RISC or CISC approach of having one operation per instruction, VLIW packs many operations into a single long instruction that is executed each cycle. This essentially exposes the details of the number and types of functional units available in the hardware. An example of the Transmeta Crusoe instruction format is giving in Figure 1.

Code density can be a big problem on VLIW systems. Because there is generally not an operation for each functional unit in each instruction, many NOPs are inserted. This is essentially wasted space. Many VLIW architectures compress the instruction stream to save space. The instruction stream is decompressed on the fly either into the L1 instruction cache or from the L1 cache on a fetch.

Binary compatibility can also be a problem for VLIW architectures. If the processor designers change instructions latencies or increase the number of functional units in a later revision then programs will have to be recompiled, and possible re-optimised by hand for the new revision.

Because of these problems VLIW systems come in a number of different flavours. *Classic* VLIW systems opt for simpler hardware at the expense of binary compatibility and other advantages. Systems such as IA64 attempt to overcome these disadvantages with more complex hardware, but still preserving the VLIW philosophy of explicit parallelism. It is for this reason that some people attempt to introduce another class of processor design - the *explicitly parallel instruction computing* (EPIC) design. There is, however, no clear feature or function that can distinguish between VLIW and EPIC. For this reason the term VLIW is used here when referring to either.

C Code	Without Predication	With Predication

`if(i==0)`
`  i = 1;`
	`      cmp r0, r2    // i == 0`
	`      bne past      // if()`
	`      mov r2, 1     // i = 1`
	`past:               // ...`
		`      cmp.eq r3 = r0, r2`
		`(r3)  mov    r2 = 1`

Figure 2: An example of code using predication

2.1 VLIW Mechanisms

2.1.1 Register Naming

VLIW architectures try to simplify hardware by removing register renaming. The architecture typically exposes more registers to the compiler or programmer. Increasing the number of registers in the ISA can eliminate name dependencies.

Selecting a size for a register file has a number of interesting trade-offs. Increasing the number of registers exposed by the architecture removes more name conflicts however it comes at a cost. A larger register file requires more hardware and could potentially affect clock speed. More addressable registers also requires longer register operand fields in instructions. Increasing the number of registers will also increase context switching overheads with the number of registers that have to be spilled and filled. Some architectures implement schemes of tracking dirty or used registers so the operating system can keep context switch costs low while the architecture still supports many registers.

2.1.2 Branches

Branching is an important topic when discussing VLIW architectures since branches can typically be 17-25% of instructions. To get competitive performance an architecture must be able to exploit parallelism across basic blocks. The best profile-based static branch prediction compiler techniques still offer inferior performance to dynamic branch prediction.

Predication is a technique used by many VLIW processors to eliminate branches. Predication involves adding an extra operand to selectively execute some instructions. A predicated instruction is conditionally evaluated based on the value of a register operand. If the register operand (predicate register) evaluates to true then the instruction is executed normally. If the predicate register is false then the instruction writes no results and causes no exceptions as though the instruction was never executed. A code example using predication is shown in Figure 2

Predication effectively turns a control dependencies into data dependencies. Instruction execution depends on the data in the predicate register rather than the destination address of a branch instruction. This change has some rather important effects. Predicate instructions fill a normal instruction slot and do not introduce pipeline bubbles or branch delays. Predicated instructions therefore have less need for complex branch prediction algorithms.

Predication can also simplify execution of multi-way branching. Without predication, code with high numbers of branches requires very complex branch prediction and speculation. With sufficient predicate registers no branches are required.

There are a number of design choices to be made when including predication in an ISA. Choosing between full and partial predication is important. Most server and desktop architectures such as IA32 and Alpha have support for some predicate instructions such as conditional move. Partial predication is simpler

<table>
<tr><th>Without Speculation</th><th>With Speculation</th></tr>
<tr><td>

```
     cmp.eq r3 = r0, r2
(r3) br     over          // if(i != 0)
     ld     r4 = [r2]     //   j = *i
      // wait for the load...
over: // do stuff
```

</td><td>

```
     ld.s   r4 = [r2]
     // ...
     cmp.eq r3 = r0, r2
(r3) br     over          // if(i != 0)
     chk.s  r4, recover   //   j = *i
over: // do stuff
     br.ret
recover:
     ld     r4 = [r2]
     br.ret
```

</td></tr>
</table>

Figure 3: An example of code using control speculation

to add to an existing ISA and allows for shorter instructions because only some instructions have an extra source register.

Full predication adds a predicate operand to every instruction. Full predication allows more instructions to be predicated and thus allows removal of more branches. This flexibility comes at the cost of an extra source operand on every instruction. Full predication also simplifies the architecture somewhat. Typical instructions such as a conditional branch naturally become an unconditional branch instruction but with a predicate to control execution.

The decision as to when to stop execution of an instruction (annul) is also important for an ISA. Annulling instructions early in the pipeline keeps functional units free to do useful work however it can cause pipeline stalls if the data dependence is not yet met. Stalling the pipeline for early annulment is somewhat contrary to the philosophy of predication. The very latest an instruction can be annulled is just before the instruction takes effect. This includes writing results to registers and raising exceptions. Late annulment generally allows the pipeline to be kept full and not stall at the expense of doing extra (useless) work. In practice most architectures annul as late as possible in the pipeline. More complex architectures such as IA64 also will not issue instructions that are known to have false predicates.

Predicate registers can be all or a subset of the architecture's general registers, or a special set of predicate registers. Reducing the number of registers for predication allows shorter instructions. Using general registers for predicate operands is simpler in the ISA and does not require extra instructions to operate on special registers. Specific predicate registers allows extra read and write port on the heavily used registers and a richer set of comparison instructions.

2.1.3 Speculation

Speculation refers to the moving of high-latency memory instructions earlier in the execution to hide the latency. This is speculative because moving the instructions can sometimes fail and special care must be taken. The two types of speculation - control and data - are further explained below.

Control speculation refers to moving a load or store before a branch instruction. In VLIW architectures this is done with special speculative instructions. A speculative load or store will attempt the instruction execution. Should the instruction cause an exception, the exception is delayed until a corresponding check instruction is executed. Example code using control speculation is given in Figure 3

To implement delayed exceptions an extra bit is added to each register that can be a destination for a load. This is generally known as a poison bit. Using a register as a destination for a non-speculative instruction clears the poison bit and copying register to register copies the poison bit. A speculative load will clear the poison bit if the load succeeded and will set the poison bit if the load failed. Any attempt to use a register as

<table>
<tr><th>Without Speculation</th><th>With Speculation</th></tr>
<tr><td><pre>st [r2] = 0x37
ld r3   = [r4]
// wait for load
st [r5] = r3</pre></td><td><pre>   ld.a   r3 = [r4]
     // ...
     st   [r2] = 0x37
     chk.a r3, recover
     st   [r5] = r3
over: // do stuff
     br.ret
recover:
     ld r3   = [r4]
     br.ret</pre></td></tr>
</table>

Figure 4: An example of code using data speculation

a source operand, other than to a copy, will result in an exception being raised. A *check* instruction is used to test the value of a poison bit. Typically the exception raised on using a poison bit is unrecoverable by the operating system. The executing code has a bug because a register with a poison bit should never be used.

Data speculation involves the re-ordering of memory instructions to hide latency. Most commonly this involves moving loads before stores, as is shown in Figure 4

Re-arranging memory instructions can cause hazards if it is not known whether the memory references overlap. Whether pointers overlap is frequently not known at compile time and can cause unwanted latency to compute during execution. Instead an optimistic approach is taken.

Advanced load instructions can be issued ahead of the use of the results. All stores before the corresponding check of the load are compared against the load address. If an advanced load and store overlap then the load can be re-issued at the check instruction.

Moving stores before other instructions requires the architecture to expose a controllable write-back buffer. If a store is moved before an overlapping load then the store cannot commit until the load has read the original value.

3 Example VLIW Processors

3.1 Phillips TriMedia TM32

The Phillips TriMedia TM32 architecture is a 'classical' VLIW architecture. It is designed for embedded media processing using VLIW for increasing performance.

TM32 exposes most of the details of the processor design. This includes the cycle latency for all instructions, the number of execution units in the processor and restrictions on how many read and write ports are available to each instruction.

The TM32 has 128 general registers with r0 and r1 being fixed to the values 0 and 1 respectively. A *long instruction* is issued on every clock cycle. Each long instruction consists of five RISC-style operations. It is up to the compiler to ensure that there are the necessary NOPs in place to avoid pipeline hazards.

The TM32 is a very simple architecture in a number of ways. It has no virtual memory or supervisor mode support. The TM32 also has no form of exceptions. Illegal operations are usually silently ignored, returning an architecture defined bit-pattern.

3.2 Transmeta Crusoe

The Transmeta family of Crusoe processors use VLIW techniques to achieve low-power. The Crusoe was designed to emulate the IA32 instruction set by *morphing* IA32 instructions into the native VLIW instructions on the fly. This is achieved by loading a translation program at boot time and using instruction emulation and translation to execute IA32 code, including the system BIOS.

The internal Crusoe architecture is also very much a classical VLIW architecture. The runtime translation code is responsible for emitting code that does not have hazards. Internally the only component that need know about the specifics of the pipeline and execution units is the runtime compiler so they are free to make a very simple architecture.

Current Crusoe processors have 64 general purpose registers. Some of these registers directly mirror IA32 architecture registers. On every cycle the Crusoe issues a 64- or 128-bit *molecule*. The 128-bit molecule consists of five RISC-like instructions. The Crusoe VLIW architecture also has some instructions that for instrumentation so the runtime system and work out what needs optimising.

The Crusoe processor uses an interesting technique for handling exceptions. Exceptions on the IA32 are precise, however the Crusoe translation software re-orders instructions with no regard for correct exception semantics. Instead, the software keeps a shadow copy of the IA32 register state and has a controllable write buffer. The shadow register copy is updated and write buffer written back after each unit of translation successfully executes. On an exception the processor state is rolled back to the start of the current translation and instructions are executed in-order. When the exception is once again generated it will be precise.

The Crusoe supports data speculation internally through special instructions. The morphing code can re-schedule load instructions before stores by changing a regular load into a *load-and-protect* and any stores in between the new and old issue place to a *store-under-alias-mask*. If any of the store instructions overlaps with the protected load then an exception is raised.

3.3 Intel IA64

The IA64 is Intel's latest workstation and server architecture [IA]. IA64 focuses on a large address space and performance for scientific applications. IA64 uses VLIW (otherwise known in IA64 terms as EPIC) to try to reduce CPI. The IA64 is by no means a simple architecture like the TM32 or the Crusoe, however IA64 tries to reduce complexity to allow faster clock rates.

The IA64 register set consists of 128 floating-point registers, 128 application registers, 64 predicate registers, 8 branch registers and at least 128 general registers. The IA64 only specifies a minimum number of general registers for the system and uses a *register stack engine* (RSE) to do explicit software renaming. An implementation of the IA64 architecture can have as many general registers as it wants, however the instruction set can only address 128 registers at a time.

To overcome the problem of excessive context switching costs the IA64 has various levels of support for dirty bits on registers. The RSE keeps track of dirty registers that need to be written out in the general registers. The floating point registers are also divided into the bottom 32 registers and top 96 registers. There is a dirty bit for each part so the operating system can minimise overheads.

Instead of the typical VLIW approach of having a long fixed-width instruction issue per clock, the IA64 uses *stops*. A stop is placed in the instruction stream to delineate where one set of instructions is dependent on the output of the previous set. The IA64 is free to issue all instructions between two stops serially or in parallel. For this reason the IA64 architecture does not define the latency of instructions or require the compiler to insert NOPs to avoid hazards. Instead the compiler inserts a stop to ensure the hardware issues no more instructions before all previous results are available. This allows an IA64 implementation to have as much hardware as it needs and to modify instruction latencies. Existing binary code will be able to take advantage of faster instructions and more execution units yet will also not have a problem if instructions are slower or there are fewer execution units.

3.3.1 Branching

IA64 has a very complex branching design. Firstly, to try to eliminate branches, all instructions are predicated. With 64 predicate registers this allows many multi-way branches to be reduced to just predicated instructions with no branches.

For the case where branches are necessary IA64 also has many solutions. Unlike most VLIW architectures the IA64 actually has branch prediction hardware. While the IA64 does not actually speculatively execute it does pre-fetch instructions to be executed. In conjunction to the branch prediction hardware the compiler has an extensive range of hints it can flag on each branch.

Part of the complexity of the IA64 branch hardware stems from the compare instructions of IA64. There can be many comparison instructions executed in parallel and each instruction has a destination predicate register for the true and false part. IA64 also allows the results of multiple comparisons to be combined into a single register. For example two compares could AND their results into a single register.

IA64 also allows multiple branches to be executed in parallel. Branches are always, and can only be, conditional on predicate registers. IA64 guarantees that branches are always evaluated in instruction order so that a branch appearing earlier in the instruction stream with a true predicate will always be taken regardless of other branches appearing in parallel. This design allows simplification of complex multi-way branches by reducing the amount of predicate manipulation code that is required.

In conjunction with these two features, IA64 specifies that predicate registers on branches do not cause RAW hazards on compare instructions in the same set of instructions. In theory all these features allow the IA64 to perform efficiently on code with a high number of branches such as switch statements.

3.3.2 Speculation

IA64 provides hardware support for both control and data speculation. Control speculation is provided through the speculative load instruction (ld.s). This instruction allows a compiler to speculatively schedule a load before a branch to hide the latency of the load. If the load should cause an exception, however, it is delayed. IA64 has a check instruction (chk.s) corresponding to the speculative load. If the load failed then the check instruction causes a branch to some code to perform a regular load and take an exception if necessary. If the speculative load succeeded then execution continues as normal.

To help implement speculation the IA64 has *not a thing* (NaT) bits on all general registers. This makes general registers 65 bits wide and the RSE and operating system have to be able to deal with the consequences of this. When a speculative load is executed the destination register is marked as NaT. Any attempt to use a NaT register, other than to copy it to another general register (and copy the NaT bit) causes an unrecoverable fault. A successful speculative load clears the NaT bit so the register can be used normally.

Data speculation is handled in the IA64 with the help of the *advanced load address table* (ALAT). An advanced load (ld.a) is output by the compiler when it schedules load instructions ahead of store instructions but cannot statically determine if the pointers overlap. The advanced load stores an entry for that load into the ALAT. All store instructions are checked against the ALAT to see if they overlap with any advanced loads. When a corresponding check instruction is issued (chk.a) it will branch to fix-up code should any stores have modified the loaded value. The IA64 architecture has no support for WAW memory hazards, or re-ordering store instructions.

3.3.3 Exceptions

IA64 has configurable exception delivery. The operating system can designate some exceptions to be handled on a speculative load rather than waiting for the fix-up code to execute. This feature allows operating systems to implement some 'cheap' exceptions and handle them ahead of a check instruction.

The operating system can select on a per-exception basis whether it would like it delivered immediately or delayed until the fix-up code executes. Only non-fatal exceptions can be delivered eagerly.

4 Evaluating VLIW

VLIW systems are designed around the idea of simplifying hardware to allow faster clock rates, more execution units and longer pipelines. It is argued that this will drastically increase achievable ILP when the compiler generates a static plan of execution. It is still unclear whether this is true.

Traditionally VLIW architectures have been very good at floating point number crunching applications while they fall behind on scalar, pointer and branching code. It seems this is still somewhat true today.

4.1 Current Performance

Hennessey and Patterson [PH03] give some benchmarks results of the a TM32 processor against some similar performance MIPS processors on predominantly floating point code. They show that with a regular compiler the TM32 processor and MIPS processors have similar performance results. However, hand optimisation of the code for the TM32 processor can lead to a 10-25 times speed up over the MIPS processor.

The IA64 processors however (Itanium I and II) have no such performance advantage. While the Itanium II is the current leader in SPECfp benchmarks it is not even by a factor of two. The Itanium II is also not the leader in SPECint benchmarks. This leads us to ask why.

This question really asks what the current performance limit on implementations of VLIW architectures such as the IA64 is. Are the VLIW compilers simply not mature enough yet or are there hardware problems that mean VLIW cannot deliver the high clock rates and many instruction issues per cycle.

4.2 Performance Limit

If the current lackluster overall performance of VLIW architectures is simply a compiler or current implementation problem, the question still remains as to what the limit of VLIW performance is.

The most recent PA-RISC processor and Alpha processor have a transistor count of 300 million and 17 million respectively. Despite the enormous number of extra gates on the PA-RISC it is only marginally better than the Alpha EV7 in most benchmarks and performs worse in some. If there is an inherent limit in super-scalar performance then they may have found it. The question as to whether VLIW architectures are inherently more scalable and able to perform far better with higher gate counts is yet to be answered.

Recent performance tests on software emulation of IA32 code on the IA64 showed that an IA64 could emulate with the same performance a modern IA32 of similar clock-speed. The benchmark code used on these tests, however, was not revealed.

<table>
<tr><th>Without Software Pipelining</th><th>With Software Pipelining</th></tr>
<tr><td>
<pre>
for(i = 0; i < 1000; i++)
{
    load_regs(i);
    // delay
    operate(i);
    store_vals(i)
}
</pre>
</td><td>
<pre>
// prologue
load_regs(0)
for(i = 0; i < 1000; i++)
{
    load_regs(i+1);
    operate(i);
    store_vals(i)
}
// epilogue
</pre>
</td></tr>
</table>

Figure 5: An example of software pipelining

5 Other ILP Techniques

This section covers some techniques for further exposing ILP that are newer or less common. This section in no way offers a comprehensive reference.

5.1 Software Pipelining

Software pipelining is a technique similar to loop unrolling but designed to keep the instruction sequence dense. When a compiler unrolls loops it is difficult to tell how many iterations to unroll. Too many iterations and there will negative effects on caching performance for diminishing returns. Too few iterations and there will still be data hazards causing pipeline stalls.

Software pipelining tries to solve this problem by scheduling load instructions even sooner. In a software pipelined loop, code executes the loads for the *i+1*th iteration of the loop during the *i*th calculation. If there is a high latency on the calculation the stores can be executed one iteration after the calculation, too. An example of this is shown in Figure 5

This technique reduces the latency of loop code while maintaining a higher code density. It is important to note that technique can also require a large number of registers. Registers are required for not only the current calculation but also the next and possibly previous iteration as well.

The IA64 architecture has special instructions devoted to software pipelining. IA64 allows a rotating set of registers that shifts on ever execution of a loop instruction. This allows for even higher code density because it can simplify the code inside the loop and remove prelude and epilogue code.

5.2 Trace Caches

Trace caches are a method of optimising the critical path of specialised blocks of code, specifically inside loops. Figure 6 shows example logic inside a loop. They grey area is the most common path that is almost always taken. Trace caches are used when the un-taken branch restricts the optimisations available on such a block of code.

An example trace of this code is given in Figure 7. This shows a loop unrolling optimisation with disregard for the branch that may be taken. This branch is replaced with trace exit and trace entry points. The resulting code is heavily optimised for the common case. The trade-off involved in this is the extremely expensive cost involved when the other trace exit branch is taken. Trace caches have been shown to offer good performance improvements for some specific pieces of code.

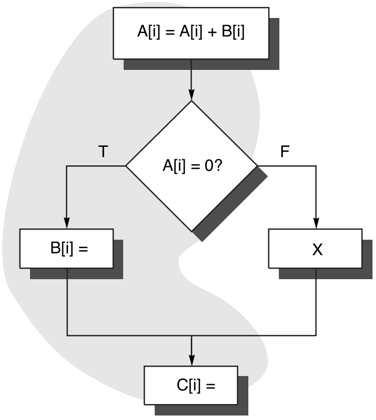

Figure 6: Code to optimised. The gray area highlights the common case.

5.3 Value Prediction

Value locality and prediction aims to increase ILP by breaking true data dependency chains [CRT99]. While typically an instruction cannot begin executing until its source operands have been computed by previous instructions, if we can accurately predict the source operands we need not wait for them to be computed. Speculative load instructions are a specific case of value prediction. Speculative loads predict the outcome of a branch instruction and attempt a load regardless.

The general topic of value prediction mainly covers further exploitation of predicting load values, however there is also work looking at the outcome of long latency instructions such as DIV and SQRT. Analysis of execution traces has shown that theoretically a speedup of approximately 70% is possible with value prediction, however this has not yet been achieved.

Examples of places to exploit value prediction are: [LWS96]

- Data redundancy in processing repetitive data
- Rarely executed error checking code
- Program constants
- Compute branches such as switch statements
- Virtual function calls
- Register spill/fill code across function calls

To predict values of load instructions a system would use a *load value prediction unit*. This unit behaves similarly to a branch-target-buffer in that it caches the result of loads instruction addresses and the instruction loaded. A load instruction that is already in the prediction unit can begin speculative execution of dependent instructions immediately. It is important to note that most load value prediction units associate the value loaded with the address of the load instruction, not the address from which to load.

Some systems propose the use of a *constant verification unit* (CVU). The CVU is responsible for predicting the outcome of what are essentially program constants that rarely change.

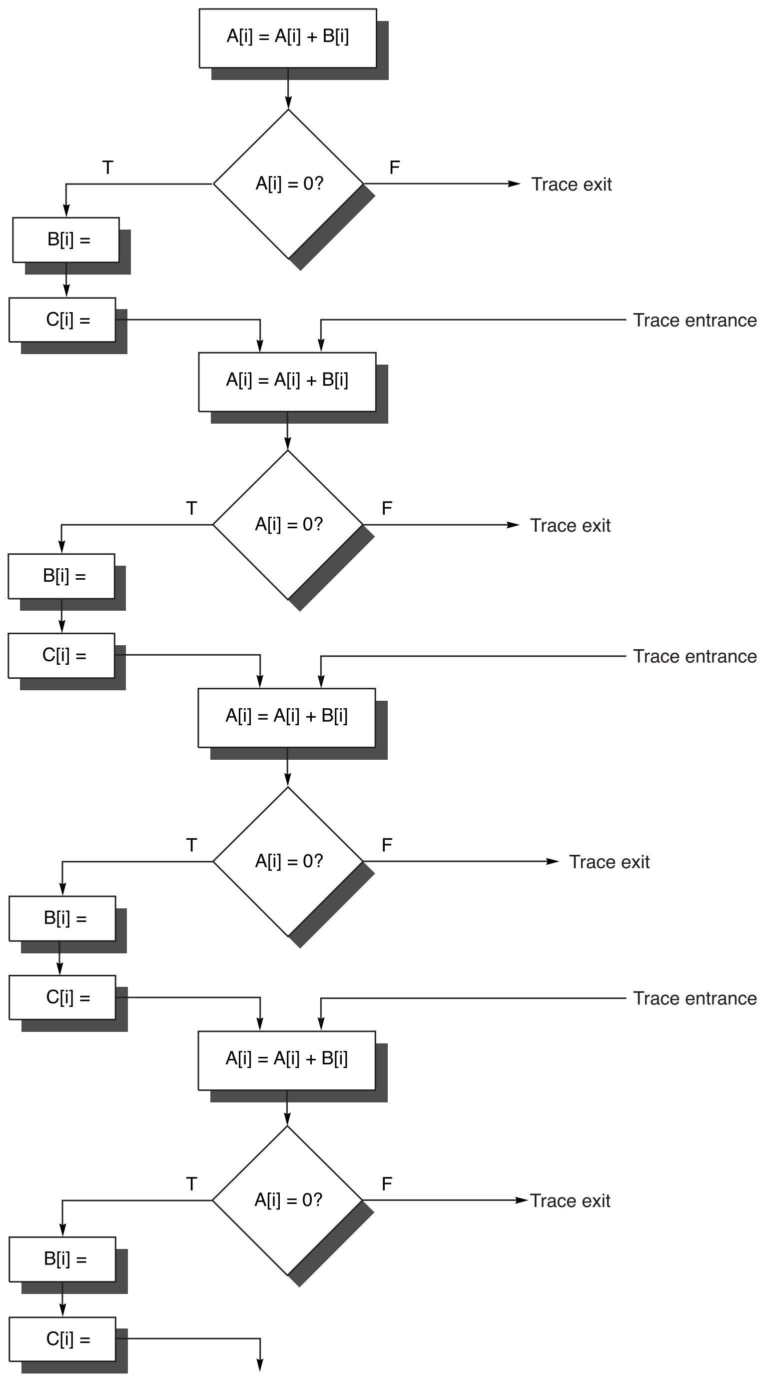

Figure 7: The unrolled trace

Value prediction is only useful when the prediction is likely to be correct. Mis-predicts are expensive and excessive numbers of them can harm performance. A value prediction system that makes good predictions offers better performance than a predictor that makes more, but less useful, predictions. Value prediction is also only useful on long latency instructions on the critical path. Predictions of fast instructions or those off the critical path offer little to no benefit.

References

[CRT99] Brad Calder, Glenn Reinman, and Dean M. Tullsen. Selective value prediction. In *ISCA*, pages 64–74, 1999. 11

[IA] Volume Ia-Application. Intel ia-64 architecture software developer's manual. 6

[LWS96] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *Architectural Support for Programming Languages and Operating Systems*, pages 138–147, 1996. 11

[PH03] David A. Patterson and John L. Hennessy. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann, 2003. 9