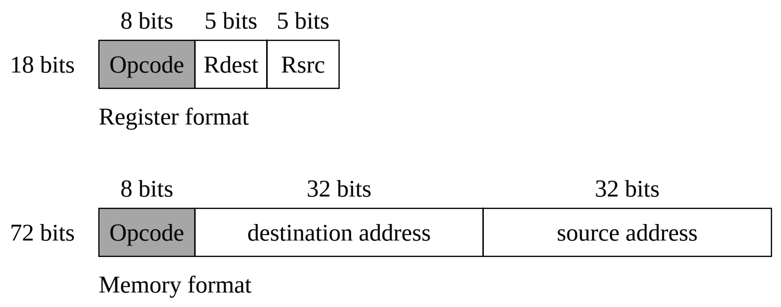
Chapter 14

RISC Processors

14–1 CISC processors provide a variety of addressing modes, which leads to variable-length instructions. In contrast to the RISC processors, CISC processors allow constants as well as operands that are either in memory or in registers. The instruction size depends on where the operands are, or whether it is a constant. For example, instruction length increases if an operand is in memory as opposed to in a register. This is because we have to specify the memory address as part of instruction encoding, which takes many more bits. Similarly, it the operand is a constant, it is stored as part of the instruction, which again increases the length of the instruction depending on the size of the constant.

The size of the instruction depends on the number of addresses and whether these addresses identify registers or memory locations. The figure below shows how the size of the instruction varies depending on whether the operands are in memory or in registers. If we use 32-bit memory addresses for each of the two addresses, we would need 72 bits for each instruction whereas the register-based instruction requires only 18 bits.



14–2 CISC designs provide a large number of addressing modes. The main motivations are (i) to support efficient access to complex data structures and (ii) to provide flexibility to access operands. For example, the Pentium provides “based-indexed addressing with scale-factor” to access complex data structures such as multidimensional arrays.

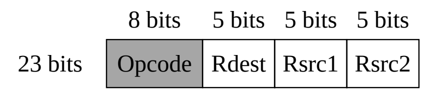
14–3 The main characteristics of RISC designs are:

- *Simple Operations:* The objective is to design simple instructions so that each can execute in one cycle. This property simplifies processor design.
- *Register-to-Register Operations:* RISC processors allow only special `load` and `store` operations to access memory. The rest of the operations work on a register-to-register basis. This feature simplifies instruction set design as it allows execution of instructions at a one-instruction-per-cycle rate.
- *Simple Addressing Modes:* Simple addressing modes allow fast address computation of operands. Since RISC processors employ register-to-register instructions, most instructions use register-based addressing. Only the load and store instructions need a memory addressing mode. RISC processors provide very few addressing modes: often just one or two.
- *Large Number of Registers:* Since RISC processors use register-to-register operations, we need to have a large number of registers. A large register set can provide ample opportunities for the compiler to optimize their usage. Another advantage with a large register set is that we can minimize the overhead associated with procedure calls and returns.
- *Fixed-Length, Simple Instruction Format:* RISC processors use fixed-length instructions. Variable-length instructions can cause implementation and execution inefficiencies. Along with fixed-length instruction size, RISC processors also use a simple instruction format. The boundaries of various fields in an instruction such as opcode and source operands are fixed. This allows for efficient decoding and scheduling of instructions.

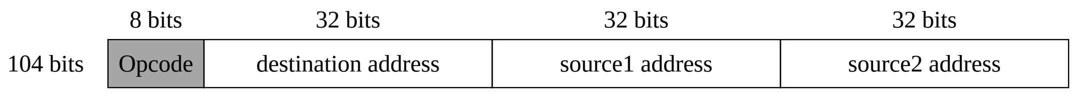
14–4 Designers make choices based on the available technology. As the technology—both hardware and software—evolves, design choices also evolve. Furthermore, as we get more experience in designing processors, we can design better systems. The RISC proposal is a response to the changing technology and the accumulation of knowledge from the CISC designs. CISC processors were designed to simplify compilers and to improve performance under constraints such as small and slow memories. We list below some of the reasons for moving away from CISC designs:

- The designers of CISC architectures anticipated extensive use of complex instructions because they close the semantic gap. In reality, it turns out that compilers mostly ignore these instructions.
- CISC ISA tends to support a variety of data structures from simple data types such as integers and characters to complex data structures such as records and structures. Empirical data suggest that complex data structures are used relatively infrequently.
- CISC designs provide a large number of addressing modes. Although large addressing modes allow flexibility, it also introduces problems. First, it causes variable instruction execution times, depending on the location of the operands. Second, it leads to variable-length instructions. For example, on the Pentium, instruction length can range from 1 to 12 bytes. Variable instruction lengths lead to inefficient instruction decoding and scheduling.

14–5 RISC processors tend to use the load/store architecture in which special load and store instructions are used to move data between the processor's internal registers and memory. All other instructions require the necessary operands to be present in the registers. Vector processors originally used the load/store architecture. This architecture reduces the size of the instruction substantially. If we assume that addresses are 32 bits long, an instruction with all three operands in memory requires 104 bits whereas the register-based operands require instructions to be 23 bits, as shown in the following figure.



Register format

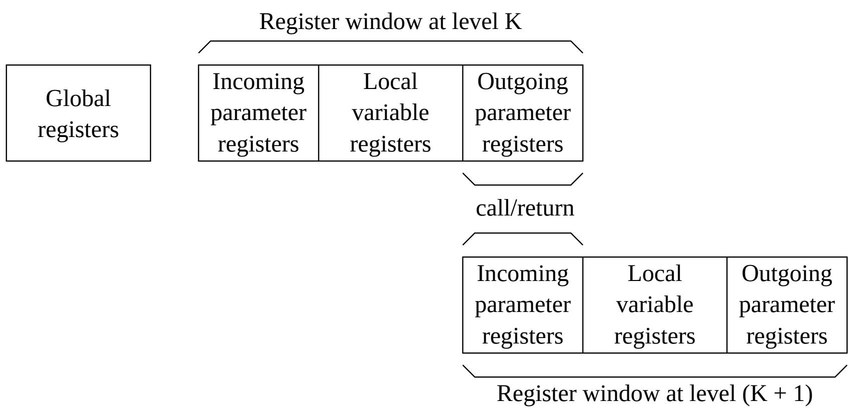


Memory format

It also allows fixed-length instructions, which allow efficient processor design and instruction scheduling.

14–6 Since RISC processors use register-to-register operations, we need to have a large number of registers. A large register set can provide ample opportunities for the compiler to optimize their usage. Another advantage with a large register set is that we can minimize the overhead associated with procedure calls and returns.

To speed up procedure calls, we can use registers to store local variables as well as for passing arguments. General-purpose registers of a processor can be divided into register windows, with each window allocated to a procedure invocation. The registers are organized as a circular buffer. To reduce overhead, outgoing parameter registers of a procedure can be overlapped with the incoming parameter registers of the called procedure as shown in the following figure.


Register window at level K
Global registers
Incoming parameter registers
Local variable registers
Outgoing parameter registers
call/return
Incoming parameter registers
Local variable registers
Outgoing parameter registers
Register window at level (K + 1)

14–7 This is somewhat similar to the flags register in the Pentium. The 32-bit register is divided into eight CR fields of 4 bits each (CR0 to CR7). CR0 is used to capture the result of an integer instruction. The 4 bits in each CR field represent “less than” (LT), “greater than” (GT), “equal to” (EQ), and “overflow” (SO) conditions. CR1 is used to capture floating-point exceptions. The remaining CR fields can be used for either integer or floating-point instructions to capture integer or floating-point LT, GT, EQ, and SO conditions. Branch instructions are available to test a specific CR field bit. Instructions can specify the CR field that should be used to set the appropriate bit.

14–8 *XER Register (XER):* The 32-bit XER register serves two distinct purposes. Bits 0, 1, and 2 are used to record summary overflow (SO), overflow (OV), and carry (CA). The OV bit is similar to the overflow flag bit in the Pentium. It records the fact that an overflow has occurred during the execution of an instruction. The SO bit is different in the sense that it is set whenever the OV bit is set. However, once set, the SO bit remains set until a special instruction is executed to clear it. The CA bit is similar to the carry bit in the Pentium. It is set by add and subtract arithmetic operations and shift right instructions.

Bits 25 to 31 are used as a 7-bit byte count to specify the number of bytes to be transferred between memory and registers. This field is used by Load String Word Indexed (`lswx`) and Store String Word Indexed (`stswx`) instructions. Using just one `lswx` instruction we can load 128 contiguous bytes from memory into all 32 general-purpose registers. Similarly, reverse transfer can be done by `stswx` instruction.

Link Register (LR): The 32-bit link register is used to store the return address in a procedure call. Procedure calls are implemented by branch (`bl`/`bla`) or conditional branch (`bc`/`bca`) instructions. For these instructions, the LR register receives the effective address of the instruction following the branch instruction. This is similar to the `call` instruction in the Pentium, except for the fact that the Pentium places the return address on the stack.

Count Register (CTR): The 32-bit CTR register holds the loop count value (similar to the ECX register in the Pentium). Branch instructions can specify a variety of conditions in many more ways than in the Pentium. For example, a conditional branch instruction can decrement CTR and branch only if CTR $\neq 0$ or if CTR = 0. Even more complex branch conditions can be tested.

14–9 Load and store instructions support three addressing modes. We can specify three general-purpose registers `rA`, `rB`, and `rD`/`rS` in load/store instructions. Registers `rA` and `rB` are used to compute the effective address. The third register is treated as either the destination register `rD` for load operations or the source register `rS` for store operations.

- *Register Indirect Addressing:* This addressing mode uses the contents of the specified general-purpose register `rA` as the effective address. Interestingly, we can also specify `0` for `rA`, which generates the address 0.

  Effective address = Contents of `rA`.

- *Register Indirect with Immediate Index Addressing:* In this addressing mode, instructions contain a signed 16-bit immediate index value `imm16`. The effective address is computed by adding this value to the contents of a general-purpose register `rA` specified in the instruction. As in the last addressing mode, a `0` can be specified in place of `rA`. In this case, the effective address is the immediate value given in the instruction. Thus, it is straightforward to convert indirect addressing to direct addressing.

Effective address = Contents of `rA` or `0` + `imm16`.

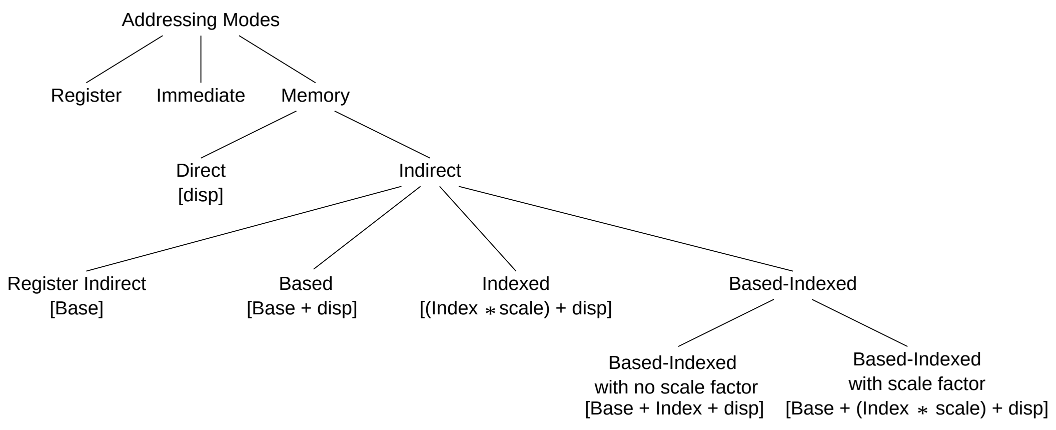
- *Register Indirect with Index Addressing:* Instructions using this addressing mode specify two general-purpose registers `rA` and `rB`. The effective address is computed as the sum of the contents of these two registers. As in the other addressing modes, we can specify `0` in place of `rA`.

  Effective address = Contents of `rA` or `0` + Contents of `rB`.

14–10 PowerPC supports three addressing modes:

1. *Register Indirect Addressing:* In this mode, the effective address is computer as follows:
   Effective address = Contents of `rA`.
2. *Register Indirect with Immediate Index Addressing:* In this mode, the effective address is computer as follows:
   Effective address = Contents of `rA` or `0` + `imm16`.
3. *Register Indirect with Index Addressing:* In this mode, the effective address is computer as follows:
   Effective address = Contents of `rA` or `0` + Contents of `rB`.

The Pentium addressing modes in the protected mode are shown in the following figure:


Addressing Modes
Register
Immediate
Memory
Direct [disp]
Indirect
Register Indirect [Base]
Based [Base + disp]
Indexed [(Index * scale) + disp]
Based-Indexed
Based-Indexed with no scale factor [Base + Index + disp]
Based-Indexed with scale factor [Base + (Index * scale) + disp]

Pentium addressing mode	PowerPC addressing mode that implements the Pentium addressing mode
Register	All instructions support this addressing mode
Direct	Limited implementation by using addressing mode 2 by specifying 0 for `rA` Limitation: only 16-bit offset
Register indirect	Addressing mode 1
Based	Addressing mode 2 Limitation: only 16-bit offset
Based indexed with no scale factor	Addressing mode 3 Limitation: only 16-bit offset

From this list, you can see that the missing Pentium addressing modes are: Immediate, Indexed, and Based indexed with scale factor.

14–11 PowerPC supports three addressing modes:

1. *Register Indirect Addressing:* In this mode, the effective address is computer as follows:

   Effective address = Contents of `rA`.

2. *Register Indirect with Immediate Index Addressing:* In this mode, the effective address is computer as follows:

   Effective address = Contents of `rA` or `0` + `imm16`.

3. *Register Indirect with Index Addressing:* In this mode, the effective address is computer as follows:

   Effective address = Contents of `rA` or `0` + Contents of `rB`.

The Itanium supports the following three addressing modes:

1. *Register Indirect Addressing:* In this mode, load and store instructions use the contents of `r3` as the effective address.

   Effective address = Contents of `r3`.

2. *Register Indirect with Immediate Addressing:* In this addressing mode, the effective address is computed by adding the contents of `r3` and a signed 9-bit immediate value `imm9` specified in the instruction. The computed effective address is placed in `r3`.

   Effective address = Contents of `r3` + `imm9`,
   `r3` = Effective address.

3. *Register Indirect with Index Addressing:* In this addressing mode, two general registers `r2` and `r3` are used to compute the effective address as in the PowerPC. As in the previous addressing mode, `r3` is updated with the computed effective address.

$$\text{Effective address} = \text{Contents of } \texttt{r3} + \text{Contents of } \texttt{r2},$$
$$\texttt{r3} = \text{Effective address.}$$

These three addressing modes appear to be similar to the PowerPC modes. The main difference is the update feature of the last two addressing modes. This update feature gives more flexibility to Itanium compared to PowerPC. For example, we can use the second addressing mode to access successive elements of an array by making `imm9` equal to the element size.

14–12 Assume that the PowerPC is configured to use the big-endian order. The code is shown below:

```
la    r4,number1   ;r4 = address of number1
la    r5,number2   ;r5 = address of number2
la    r6,result    ;r5 = address of result
lwz   r7,4(r4)     ;r7 = lower half of number1
lwz   r8,4(r5)     ;r8 = lower half of number2
addc  r8,r7,r8     ;r8 = r7 + r8
stw   r8,4(r6)     ;store lower 32 bits of result
lwz   r7,(r4)      ;r7 = upper half of number1
lwz   r8,(r5)      ;r8 = upper half of number2
adde  r8,r7,r8     ;r8 = r7 + r8 + CA
stw   r8,(r6)      ;store upper 32 bits of result
```

14–13 The PowerPC multiply instruction is slightly different from that of the Pentium in that it does not produce the full 64-bit result. Remember that we get a 64-bit result when multiplying two 32-bit integers. The PowerPC provides two instructions to get the lower- and higher-order 32 bits of the 64-bit result. First, we look at the signed integers. The instruction

```
mullw     rD,rA,rB
```

multiplies the contents of registers `rA` and `rB` and stores the lower-order 32 bits of the result in `rD` register. We have to use

```
mulhw     rD,rA,rB
```

to get the higher-order 32 bits of the result.

For unsigned numbers, we have to use `mulhwu` instead of `mulhw` to get the higher-order 32 bits of the result. The lower-order 32 bits are given by `mullw` for both signed and unsigned integers.

14–14 The BO (branch options) operand, which is five bits long, specifies the condition under which the branch is taken. The BI (branch input) operand specifies the bit in the CR field that should be used as the branch condition.

14–15 EPIC design features include the following:

- *Explicit Parallelism:* The ISA provides necessary support for the compiler to convey information on the instructions that can be executed in parallel. In traditional architectures, hardware extracts this instruction-level parallelism (ILP) within a fairly small window of instructions (or reorder buffer). In contrast, the Itanium allows the compiler to do the job. Since ILP extraction is done in software, a more detailed analysis can be done on a much larger window at compile time.

  The Itanium also provides hardware support to execute instructions in parallel by reading three instructions as a bundle. The compiler packs only instructions that have no dependencies into a bundle. Thus, the processor does not have to spend time in analyzing the instruction stream to extract ILP.
- *Features to Enhance ILP:* Since the Itanium allows the compiler to detect and extract ILP, we can use more elaborate methods to improve ILP. Two such schemes are speculative execution and predication. Speculative execution allows high-latency load instructions to be advanced so that the latency can be masked. Branch handling is improved by using predication. In some instances, branch instructions can be completely eliminated.
- *Resources for Parallel Execution:* It is imperative that to successfully exploit the ILP detected by the compiler, the processor should provide ample resources to execute instructions in parallel. The Itanium provides a large number of registers and functional units. It has 128 integer registers and another 128 floating-point registers. The large number of registers, for example, can be effectively used to make procedure calls and returns very efficient. Most procedure calls/returns need not access memory for parameter values and local and global variables.

14–16 Itanium uses registers to improve performance, particularly supports efficient procedure invocation and return. Unlike the Pentium procedure calls, an Itanium procedure call typically does not involve the stack. Instead, it uses a register stack for passing parameters, local variables, and the like. For this purpose, the 128 general register set is divided into *static* and *stacked* registers.

Static registers are comprised of the first 32 registers: `gr0` through `gr31`. These registers are used to store the global variables and are accessible to all procedures.

The upper 96 registers, `gr32` through `gr127`, are called stack registers. These registers, instead of the stack, are used for passing parameters, returning results, and storing local variables.

14–17 A predicate register indicates whether the associated instruction should be executed. Itanium has 64 1-bit predicate registers. An instruction is executed only if the specified predicate register has a true (1) value; otherwise, the instruction is treated as a NOP (no operation). If a predicate register is not specified in an instruction, predicate register p0 is used, which is always true.

14–18 The eight 64-bit branch registers, `br0` through `br7`, hold the target address for indirect branches. These registers are used to specify the target address for conditional branch, procedure call, and return instructions.

14–19 The static registers are used to store the global variables and are accessible to all procedures. The stacked registers, instead of the stack, are used for passing parameters, returning results, and storing local variables. Specifically, these registers are used to implement register-based stack frames. A register stack frame is the set of stacked registers visible to a given procedure. This stack frame is partitioned into local area and output area. The local area consists of input area and space for local variables of the procedure. The input area is used to receive parameters from the caller and the output area is used to pass parameters to the callee. The Itanium aligns the caller's output area with the callee's local area so that passing of parameters does not involve actual copying of register values.

14–20 The Itanium enables instruction-level parallelism by letting the compiler/assembler explicitly indicate parallelism by providing run-time support to execute instructions in parallel, and by providing a large number of registers to avoid register contention. By means of instruction groups, compilers package instructions that can be executed in parallel. It is the compiler's responsibility to make sure that instructions in a group do not have conflicting dependencies.

In traditional architectures, hardware extracts this instruction-level parallelism (ILP) within a fairly small window of instructions (or reorder buffer). In contrast, the Itanium allows the compiler to do the job. Since ILP extraction is done in software, a more detailed analysis can be done on a much larger window at compile time. This further enables Itanium to use more elaborate methods to improve ILP. Two such schemes are speculative execution and predication. Speculative execution allows high-latency load instructions to be advanced so that the latency can be masked. Branch handling is improved by using predication. In some instances, branch instructions can be completely eliminated. These techniques improve overall performance.

14–21 Instructions bundles are useful to convey the instruction group information to the hardware. Note that the compiler forms instruction groups such that instructions in a group do not have conflicting dependencies. By means of instruction groups, compilers package instructions that can be executed in parallel. Three instructions in a group are put together to form instructions bundles. The hardware can schedule these instructions without further analysis as they are guaranteed to be non-conflicting. An advantage of the instruction bundles is that processors with additional resources can take advantage of the existing ILP in the instruction bundle.

14–22 The main purpose of the template field is to specify mapping of instruction slots to execution instruction types. Instructions are categorized into six instruction types: integer ALU, non-ALU integer, memory, floating-point, branch, and extended. A specific execution unit may execute each type of instruction. For example, floating-point instructions are executed by the F-unit, branch instructions by the B-unit, and memory instructions such as load and store by the M-unit. The remaining three types of instructions are executed by the I-unit.

14–23 There are three techniques to handle branches:

- *Branch Elimination:* The best solution is to avoid the problem in the first place. Although we cannot eliminate all branches, we can eliminate certain types of branches. This elimination cannot be done without support at the instruction-set level.

- *Branch Speedup:* If we cannot eliminate a branch, at least we can reduce the amount of delay associated with it. This technique involves reordering instructions so that instructions that are not dependent on the branch/condition can be executed while the branch instruction is processed. Speculative execution can be used to reduce branch delays.
- *Branch Prediction:* If we can predict whether the branch will be taken, we can load the pipeline with the right sequence of instructions. Even if we predict correctly all the time, it would only convert a conditional branch into an unconditional branch. We still have the problems associated with unconditional branches.

14–24 In the Itanium, branch elimination is achieved by a technique known as *predication*. An instruction is not automatically executed when the control is transferred to it. Instead, it will be executed only if a condition is true. This requires us to associate a predicate with each instruction. If the associated predicate is true, the instruction is executed; otherwise, it is treated as a `nop` instruction. The Itanium architecture supports full predication to minimize branches. Most of the Itanium's instructions can be predicated.

To see how predication eliminates branches, let us look at the following example:

```
if (R1 == R2)                           cmp    r1,r2
    R3 = R3 + R1;                       je     equal
else                                    sub    r3,r1
    R3 = R3 - R1;                       jmp    next
                                 equal:
                                        add    r3,r1
                                 next:
```

The code on the left-hand side, expressed in C, is a simple if-then-else statement. The Pentium assembly language equivalent is shown on the right. As you can see, it introduces two branches: unconditional (`jmp`) and conditional (`je`). Using the Itanium's predication, we can express the same as

```
     cmp.eq p1,p2 = r1,r2
(p1) add r3 = r3,r1
(p2) sub r3 = r3,r1
```

The compare instruction sets two predicates after comparing the contents of `r1` and `r2` for equality. The result of this comparison is placed in `p1` and its complement in `p2`. Thus, if the contents of `r1` and `r2` are equal, `p1` is set to 1 (true) and `p2` to 0 (false). Since the `add` instruction is predicated on `p1`, it is executed only if `p1` is true. It should be clear that either the `add` or the `sub` instruction is executed, depending on the comparison result.

14–25 When two instructions access common resources (either registers or memory locations) in a conflicting mode, data dependency exists. A conflicting access is one in which one or both instructions alter the data. A control dependency exists if the execution of an instruction depends on a runtime value (e.g., result of a comparison). In general, handling control dependencies are more difficult as they need runtime support.

14–26 We discussed two types of speculative execution: one type handles data dependencies, and the other deals with control dependencies. Both techniques are compiler optimizations that allow the compiler to reorder instructions. For example, we can speculatively move high-latency load instructions earlier so that the data are available when they are actually needed.

Data speculation allows the compiler to schedule instructions across some types of ambiguous data dependencies. The read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) dependencies are not ambiguous in the sense that the dependency type can be statically determined at compile/assembly time. These dependencies can be handled statically by the compiler/programmer. However, when there is ambiguous data dependency, as in the following example, instruction reordering may not be possible:

```
sub    r6=r7,r8 ;;     // cycle 1

st8    [r9]=r6         // cycle 2
ld8    r4=[r5] ;;

add    r11=r12,r4 ;;   // cycle 4

st8    [r10]=r11       // cycle 5
```

The Itanium provides architectural support to move such load instructions. This is facilitated by advance load (`ld.a`) and check load (`ld.c`). The basic idea is that we initiate the load early and when it is time to actually execute the load, we will make a check to see if there is a dependency that invalidates our advance load data. If so, we reload; otherwise, we successfully advance the load instruction even when there is ambiguous dependency. The previous example with advance and check loads is shown below:

```
1:  ld8.a  r4=[r5]         // cycle 0 or earlier
        . . .
2:  sub    r6=r7,r8 ;;     // cycle 1

3:  st8    [r9]=r6        // cycle 2
4:  ld8.c  r4=[r5]
5:  add    r11=r12,r4 ;;

6:  st8    [r10]=r11       // cycle 3
```

When we want to reduce latencies of long latency instructions such as load, we advance them earlier into the code. When there is a branch instruction, it blocks such a move because we do not know whether the branch will be taken until we execute the branch instruction. Let us look at the following code fragment:

```
     cmp.eq  p1,p0 = r10,10   // cycle 0
(p1) br.cond   skip ;;        // cycle 0
     ld8   r1 = [r2] ;;       // cycle 1
```

```
          add   r3 = r1,r4         // cycle 3
skip:
          // other instructions
```

In the above code, without runtime support from the processor, we cannot advance the load instruction due to the branch instruction. Itanium provides speculative check (`chk.s`) to facilitate advancing such load instructions as shown below:

```
          ld8.s   r1 = [r2] ;;     // cycle -2 or earlier

           // other instructions

          cmp.eq  p1,p0 = r10,10   // cycle 0
     (p1) br.cond   skip           // cycle 0
          chk.s   r1, recovery     // cycle 0
          add   r3 = r1,r4         // cycle 0
skip:
          // other instructions
recovery:
          ld8   r1 = [r2]
          br    skip
```