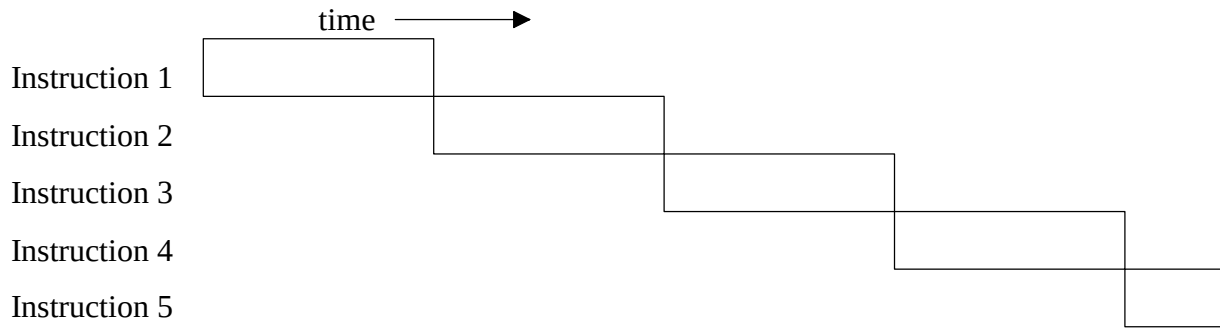
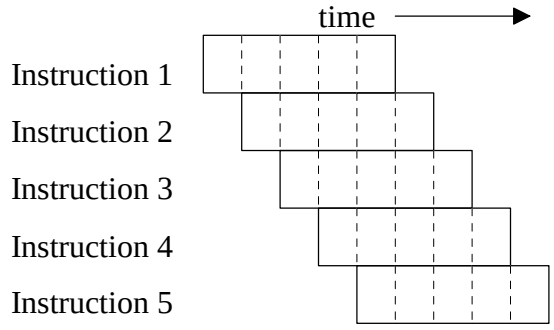
Serial Execution

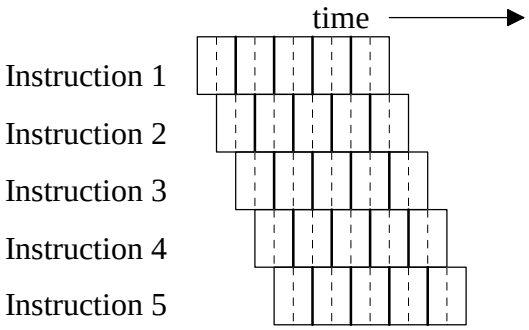


Pipelined Execution - Original RISC goal is to complete one instruction per clock cycle



Advanced Architectures - multiple instructions completed per clock cycle

1. *superpipelined* (e.g., MIPS R4000)- split each stage into substages to create finer-grain stages



2. *superscalar* (e.g., Intel Pentium, AMD Athlon)- multiple instructions in the same stage of execution in duplicate pipeline hardware

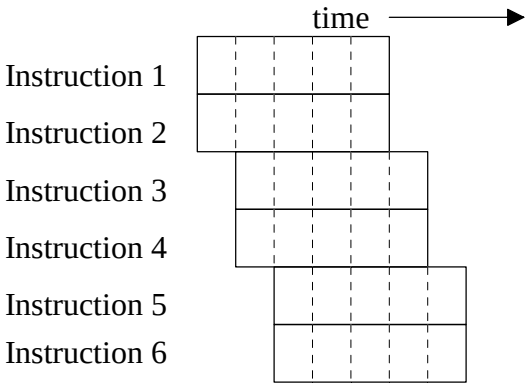
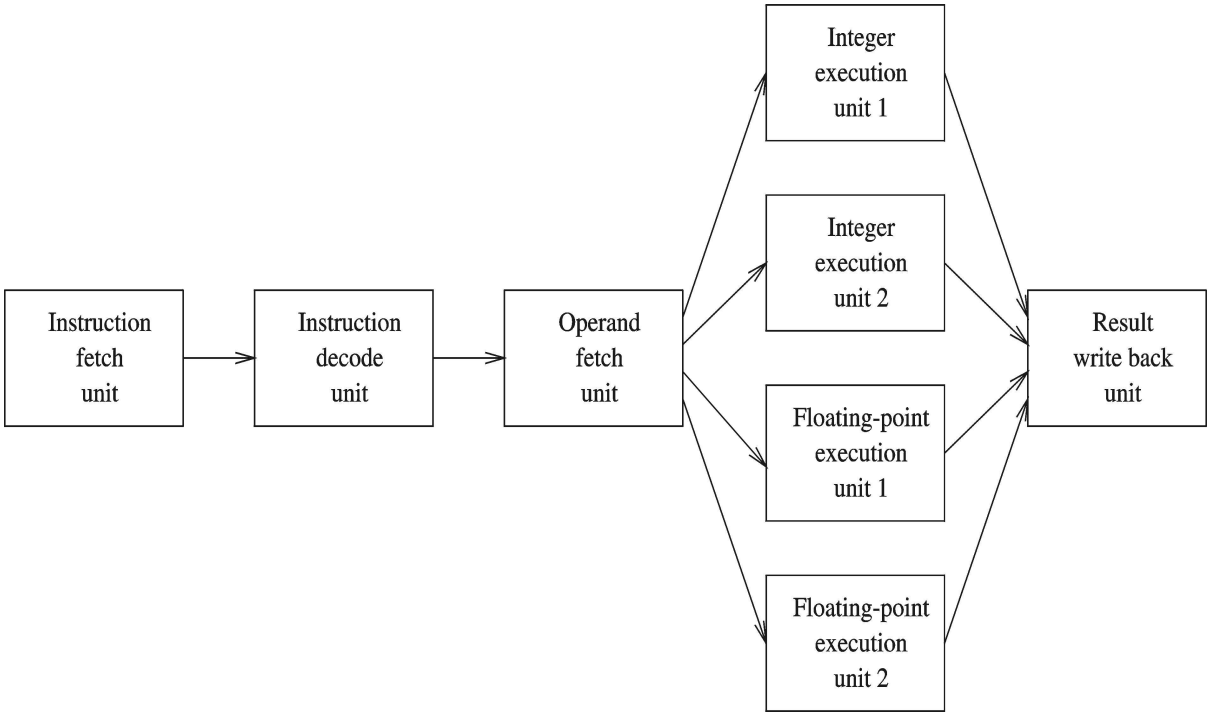


Figure 8.15 on page 288 - several instructions in the "execute" stage on different functional units



3. *very-long-instruction-word, VLIW* (e.g., Intel Itanium) - compiler encodes multiple operations into a long instruction word so hardware can schedule these operations at run-time on multiple functional units ***without analysis***

machine parallelism - the ability of the processor to take advantage of instruction-level parallelism. This is limited by:

- number of instructions that can be fetched and executed at the same time (# of parallel pipelines)
- ability of the processor to find independent instructions (the processor needs to look ahead of the current point of execution to locate independent instructions that can be brought into the pipeline and executed without hazards)

Limitations of superscalar - how much "instruction-level parallelism" (ILP) exists in the program. Independent instructions in the program can be executed in parallel, but not all can be.

1) true data dependency:
```
SUB R1, R2, R3      ; R1← R2 - R3
ADD  R4, R1, R1     ; R4← R1 + R1
```
Cannot be avoided by rearranging code

2) procedural dependency - cannot execute instructions after branch until branch executes

3) resource conflict / structural hazard - several instructions need same piece of hardware at the same time (e.g., memory, caches, buses, register file, functional units)

Three types of orderings:
1) order in which instructions are fetched
2) order in which instructions are executed (called *instruction issuing*)
3) order in which instructions update registers and memory

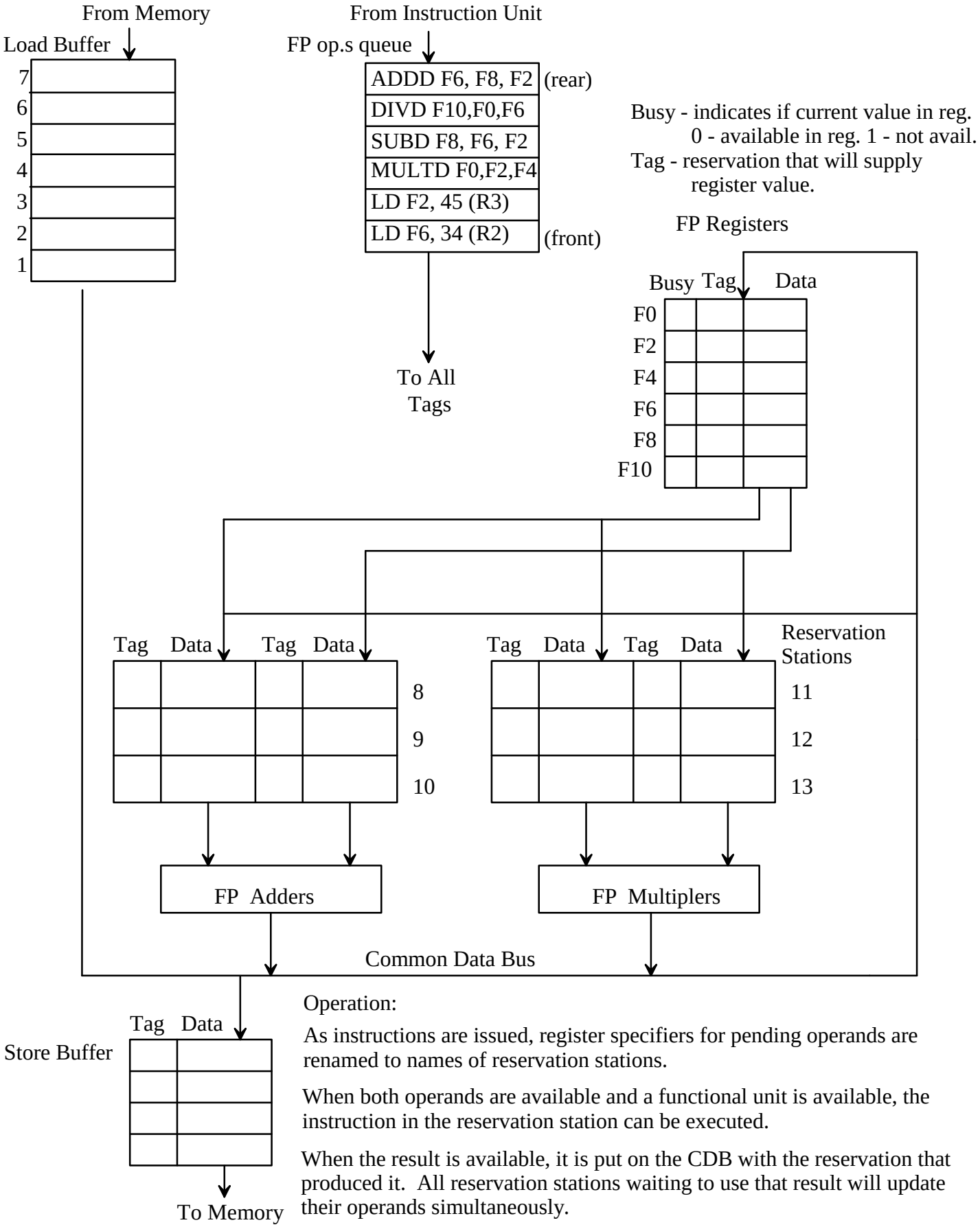
The more sophisticated the processor, the less it is bound by the strict relationship between these orderings. The only real constraint is that the results match that of sequential execution.

Some Categories:
a) In-order issue with In-order completion.

b) In-order issue with out-of-order completion

Problem: *Output dependency / WAW dependency (Write-After-Write)*

```
I1:   R3 ← R3 op R5
I2:   R4 ← R3 + 1
I3:   R3 ← R5 + 1
I4:   R7 ← R3 op R4      ; R3 value generated from I3 must be used
```



Operation:

As instructions are issued, register specifiers for pending operands are renamed to names of reservation stations.

When both operands are available and a functional unit is available, the instruction in the reservation station can be executed.

When the result is available, it is put on the CDB with the reservation that produced it. All reservation stations waiting to use that result will update their operands simultaneously.

c) Out-of-Order Issue (decouple decode and execution) with Out-of-Order Completion

Instruction window provides a pool of possible instructions to be executed:

- filled after decode
- removed when issued if (1) fn. unit is available and (2) no conflicts or dependencies

Antidependency / WAR (Write-After-Read)

I1: R3 ← R3 op R5
I2: R4 ← R3 + 1
I3: R3 ← R5 + 1 ; If executed out-of-order, then I2 could get wrong value for R3
I4: R7 ← R3 op R4

Notice that I3 is just reusing R3 and does not need its value, so it is just a conflict for the use of a register.

Register Renaming is a solution to this problem; We allocate a different register dynamically at run-time

I1: $R3_b \leftarrow R3_a$ op $R5_a$; $R3_b$ and $R3_a$ are different registers
I2: $R4_b \leftarrow R3_b + 1$
I3: $R3_c \leftarrow R5_a + 1$
I4: $R7_b \leftarrow R3_c$ op $R4_b$

Example using Tomasulo's Algorithm

Tomasulo's Algorithm is an example of ***dynamic scheduling***. In dynamic scheduling the ID-WB stages of the five-stage pipeline is split into three stages to allow for out-of-order execution:

1. *Issue* - decodes instructions and checks for structural hazards. Instructions are issued in-order through a FIFO queue to maintain correct data flow. If there is not a free reservation station of the appropriate type, the instruction queue stalls.
2. *Read operands* - waits until no data hazards, then read operands
3. *Write result* - send the result to the CDB to be grabbed by any waiting register or reservation stations

All instructions pass through the issue stage in order, but instructions stalling on operands can be bypass by later instructions whose operands are available.

RAW hazards are handled by delaying instructions in reservation stations until all their operands are available.

WAR and WAW hazards are handled by renaming registers in instructions by reservation station numbers.

Load and Store instructions to different memory addresses can be done in any order, but the relative order of a Store and accesses to the same memory location must be maintained. One way to perform ***dynamic disambiguation*** of memory references, is to perform effective address calculations of Loads and Stores in program order in the issue stage.

- Before issuing a Load from the instruction queue, make sure that its effective address does not match the address of any Store instruction in the Store buffers. If there is a match, stall the instruction queue until, the corresponding Store completes. (Alternatively, the Store could forward the value to the corresponding Load**)**
- Before issuing a Store from the instruction queue, make sure that its effective address does not match the address of any Store or Load instructions in the Store or Load buffers.

Studies have shown that superscalar machines:

- need register renaming to significantly benefit from duplicate functional units
- with renaming a larger window size is important

Branch prediction - usually used instead of delayed branching since multiple instructions need to execute in the delay slot causing problems related to instruction dependencies

Committing / Retiring Step - needed since instructions may complete out-of-order

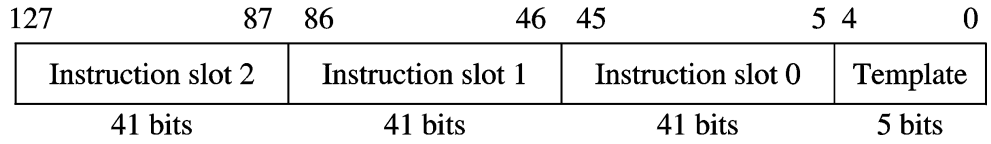
Using branch prediction and speculative execution means some instructions' results need to be thrown out

Results held is some temporary storage and stores performed in order of sequential execution.

Itanium Processor

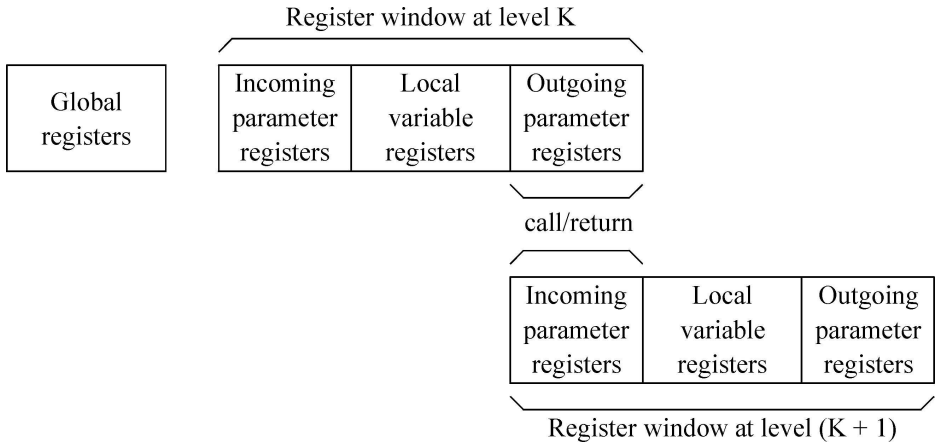
Interesting Features:

- Uses *explicit parallel instruction computing (EPIC)* from *very-long-instruction-word (VLIW)* architecture. In EPIC the compiler encodes multiple operations into a long instruction word so hardware can schedule these operations at run-time on multiple functional units without analysis. On the Itanium, a three instruction bundle is read -- Figure 14.6.



template field maps instruction slots to execution types (integer ALU, non-ALU integer, memory, floating-point, branch, and extended)

- Provides hardware support for efficient procedure calls and returns -- large number of registers with overlapping register windows (see Figure 14.1)



Itanium: first 32 registers for global variables and remaining 96 registers for local variables and parameters.

- Features to Enhance ILP: (1) Predication of eliminate branches, (2) Hiding memory latency by speculative loads, and (3) Improving branch handling by using predication

```
add r1 = r2,r3        // r1 = r2 + r3
add r1 = r2,r3,1      // r1 = r2 + r3 + 1
```

Compare instructions **- used to set predicate reg(s)**

```
cmp.eq p3 = r2,r4     // p3 set if r2 equals r4
cmp.gt p2,p3 = r3,r4  // p3 = not p2
```

Predicate instruction

```
(p4) add r1 = r2,r3   // result of add only
                      // seen if p4 is true
```

Branch instruction

```
br.cloop.sptk loop_back
```

Instruction groups - Set of instructions that do not have conflicting dependencies

- Can be executed in parallel
- Compiler/assembler can indicate this by `;;` notation

Example: Logical expression with four terms

```
if (r10 || r11 || r12 || r13) {
    /* if-block code */
}
```

can be done using or-tree evaluation

```
or   r1 = r10,r11    /* Group 1 */
or   r2 = r12,r13 ;;
or   r3 = r1,r2      /* Group 2 */
Other instructions   /* Group 3 */
```

Processor can execute as many instructions from group as it can

- Depends on the available resources

Data transfer instructions

- Load and store instructions are more complicated than a typical RISC processor

Load instruction formats

```
(qp) ldSZ.ldtype.ldhint r1=[r3]
(qp) ldSZ.ldtype.ldhint r1=[r3],r2
(qp) ldSZ.ldtype.ldhint r1=[r3],imm9
```

- Loads SZ bytes from memory
  - SZ can be 1, 2, 4, or 8 to load 1, 2, 4, or 8 bytes
  - Example:

```
ld8  r5 = [r6]
```

ldtype - This completer can be used to specify special load operations

```
ld8.a  r5 = [r6]    // Advanced

ld8.s  r5 = [r6]    // Speculative
```

ldhint - Locality of memory access

None – Temporal locality, level 1
nt 1 – No temporal locality, level 1
nt a – No temporal locality, all levels

Three techniques for Reducing Branch Penalties:

Branch elimination - Best way to handle branches is not to have branches
Possible to eliminate some types of branches

Branch speedup - Reduce the delay associated with branches
Reorder instructions
Speculative execution

Branch prediction - Discussed before (see Chapter 8)

Ambiguous Data Dependency

Ambiguous Data Dependency - dependencies between load and store instructions which use pointers to access memory

```
st8  [r9] = r6
ld8  r4 = [r5]
```

Since the pointer values (r9 and r5) are calculated at run-time and are not know by the compiler, a store instruction to memory followed by a load instruction from memory would have a dependency if two instructions referenced the same part of memory.

On the Itanium, advance load (ld.a) instruction is used to start a load well in advance of the instruction that needs the value read. Before using the value prefetched by the advance load instruction, we can check to see if a subsequent store might have caused an ambiguous data dependency using the check load (ld.c) instruction. For example,

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

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

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

The advance load (ld8.a) starts the load well in advance of the “add” instruction that needs the value loaded into r4. If the store (st8) instruction refers to the same memory as the advance load, then the value read into r4 is garbage. If such a dependency exists, the check load (ld8.c) instruction automatically reexecutes the load and the add instruction.

Control Speculation

Consider the following Itanium code:

```
                cmp.eq      p1, p0 = r10, 10        // cycle 0
        (p1)    br.cond     end_if ; ;              // cycle 0
                ld8         r1 = [r2] ; ;           // cycle 1 (loads take 2 cycles)
                add         r3 = r1, r4             // cycle 3
end_if: ...
```

We'd like to move the load (ld8) instruction earlier in the code to avoid the latency, but we don't know if the load should ultimately be performed because of the branch, **and the load will not cause an exception** (e.g., r2 points to null).

The Itanium provides a speculative load (ld.s) instruction to use in these cases.

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

                cmp.eq      p1, p0 = r10, 10        // cycle 0
        (p1)    br.cond     end_if                  // cycle 0
                chk.s       r1, recovery            // cycle 0
                add         r3 = r1, r4             // cycle 0
end_if: ...


recovery:
                code to recover from the exception
```

If the ld.s instruction causes an exception, the exception is not raised immediately, but delayed until the chk.s instruction is encountered. If the branch is taken, the hardware ensures that the results produced by the speculative load do not update r1.