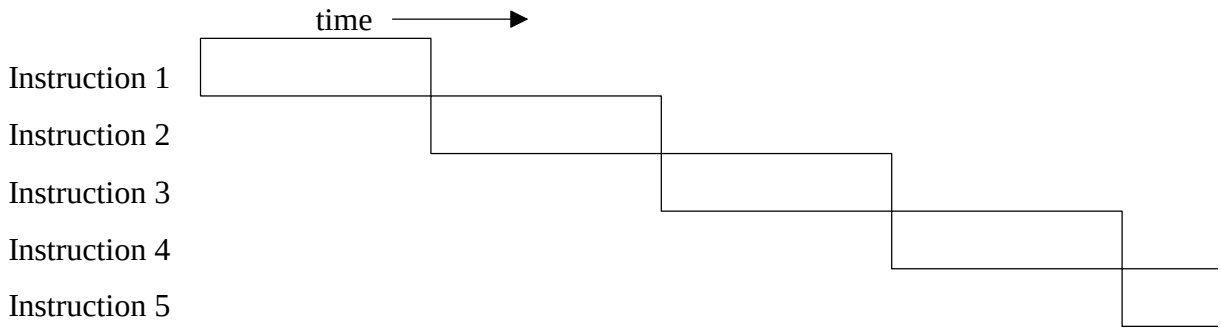
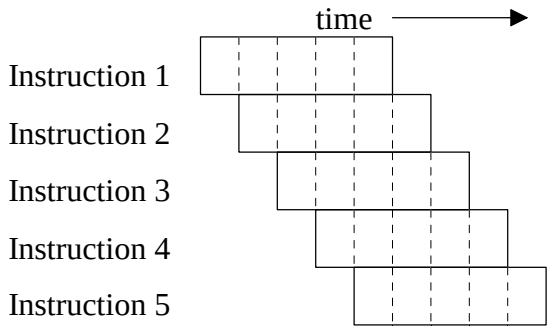
**Serial Execution**

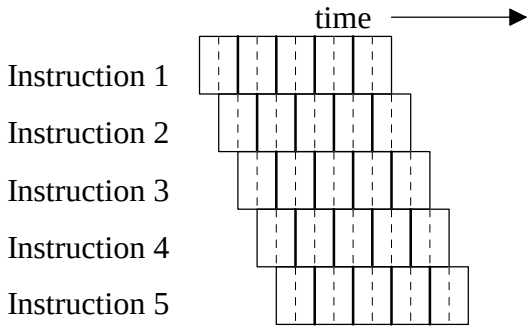



**Pipelined Execution** - Original RISC goal is to complete one instruction per clock cycle

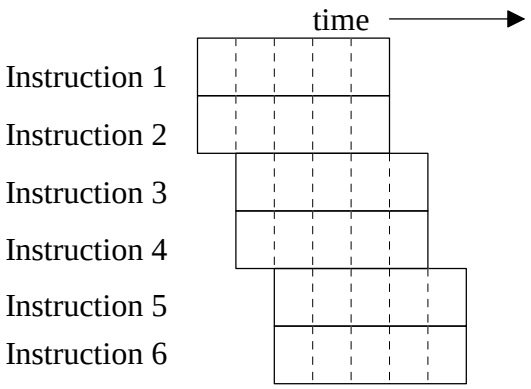



## Advanced Architectures - multiple instructions completed per clock cycle

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

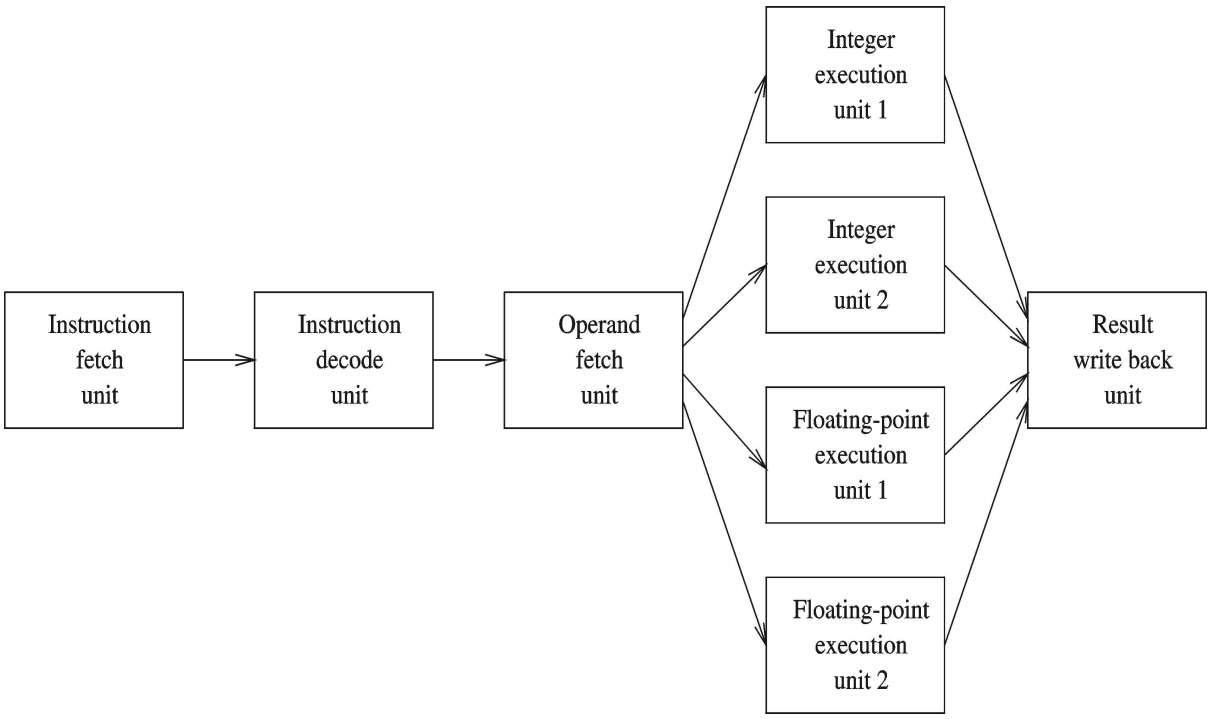


2. *superscalar* (e.g., Intel Pentium, AMD Athlon)- multiple instructions in the same stage of execution in duplicate pipeline hardware




Alternatively, several instructions in the "execute" stage on different functional units

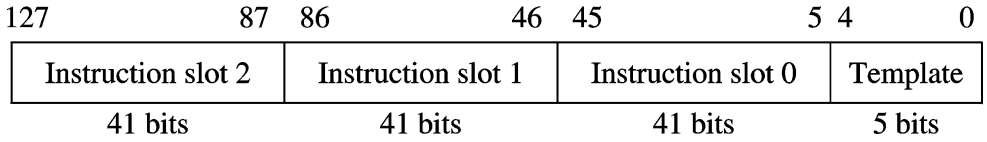
3. *very-long-instruction-word, VLIW* (e.g., Intel Itanium) - compiler encodes multiple operations into a long instruction word so hardware can schedule these operations at run-time on multiple functional units ***without analysis***



# Itanium Processor

Interesting Features:

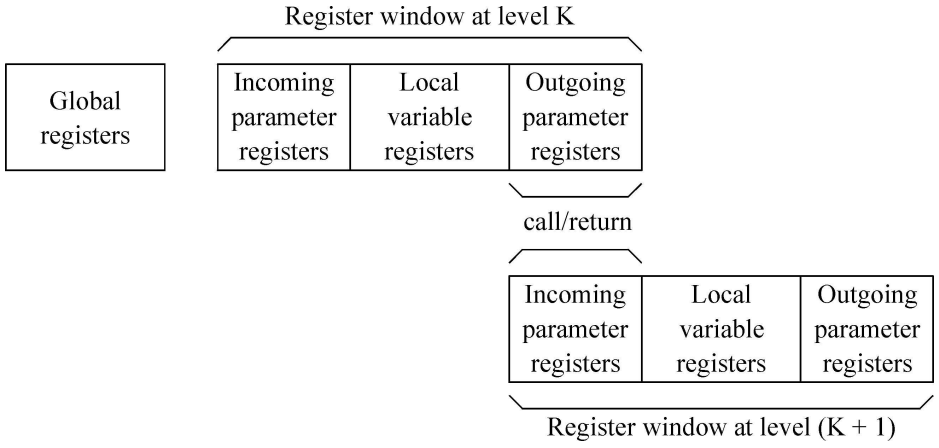
- Uses *explicit parallel instruction computing (EPIC)* from *very-long-instruction-word (VLIW)* architecture. In EPIC the compiler encodes multiple operations into a long instruction word so hardware can schedule these operations at run-time on multiple functional units without analysis, called *static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




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

| Processor | Max. instr. issue per clock | Functional units | Max. ops. per clock | Max. clock rate | Transistors (millions) | SPEC int2000 | SPEC fp2000 |
|---|---|---|---|---|---|---|---|
| Itanium | 6 | 4 integer<br>2 memory<br>3 branch<br>2 FP | 9 | 0.8 GHz | 25 | 379 | 701 |
| Itanium 2 | 6 | 6 integer<br>4 memory<br>3 branch<br>2 FP | 11 | 1.5 GHz | 221 | 810 | 1427 |

- Provides hardware support for efficient procedure calls and returns -- large number of registers (128 general-purpose and 128 fl. pt. registers) with overlapping register windows




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 - 64 1-bit “Predicate registers”,
  (2) Hiding memory latency by speculative loads (control speculation),
  (3) Hiding memory latency when branching is involved (data speculation), and
  (4) Hardware support to dynamically unroll a loop.

## Itanium AL Instruction Examples

```
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   // unconditional branch
(p5) br.cond.sptk         // branch is taken if p5 is true
```

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]
st4  r9 = [r9], 4  // postinc by 4
```

**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

| | |
|---|---|
| if (R1 == R2)<br>  R3 = R3 + R1<br>else<br>  R3 = R3 - R1<br>end if | cmp.eq p1, p2 = r1, r2<br>(p1) add r3 = r3, r1<br>(p2) sub r3 = r3, r1 |

Branch speedup - Reduce the delay associated with branches

Reorder instructions

```
sub r6 = r7, r8;;     // cycle1
sub  r9 = r10, r6
ld8  r4 = [r5];;      // cycle 2 (ld takes two cycles to fetch from L1)
add  r11 = r12, r4  // cycle 4
```

Reorder instruction

```
ld8  r4 = [r5];;      // cycle 1 (ld takes two cycles to fetch from L1)
sub r6 = r7, r8;;     // cycle2
sub  r9 = r10, r6
add  r11 = r12, r4  // cycle 3
```

Branch prediction - Discussed before

Control Speculation -- speculative load

## 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.

## Data Speculation -- 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.

Software Pipelining Example form text....