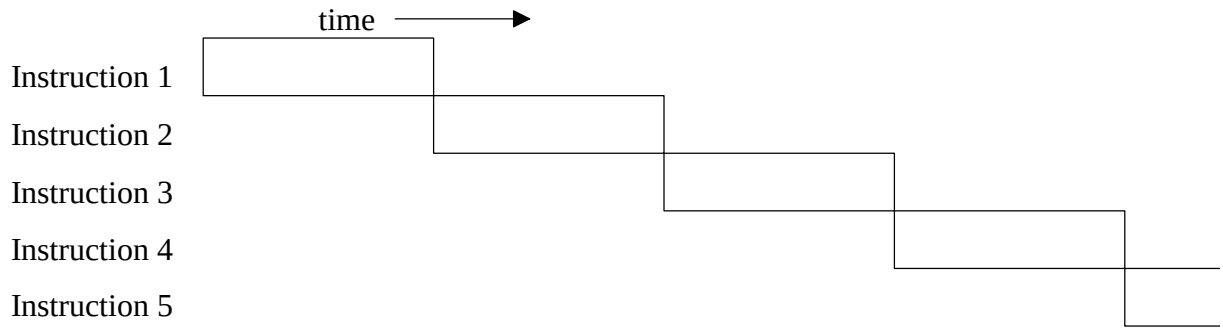
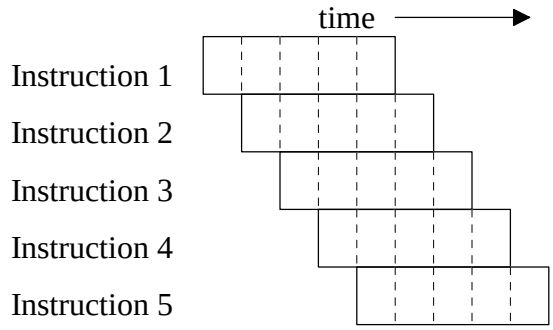
Serial Execution

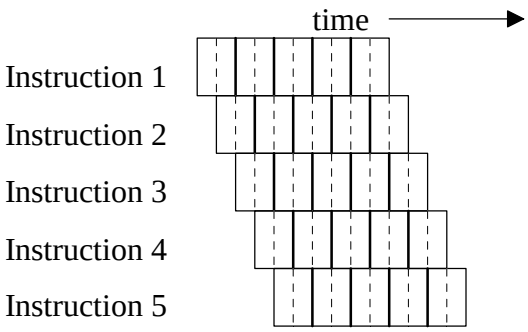


Pipelined Execution - Original RISC goal is to complete one instruction per clock cycle

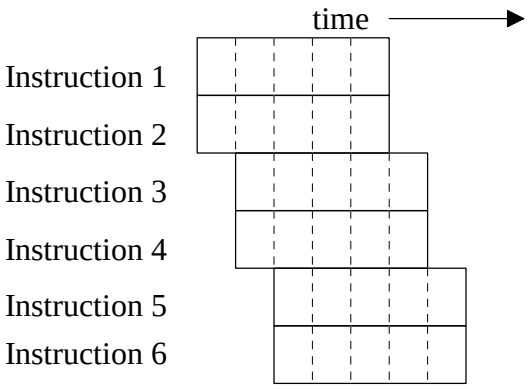


Advanced Architectures - multiple instructions completed per clock cycle

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, avoid duplicating hardware for all stages of the pipeline by only allowing several instructions in the "execute" stage on different functional units

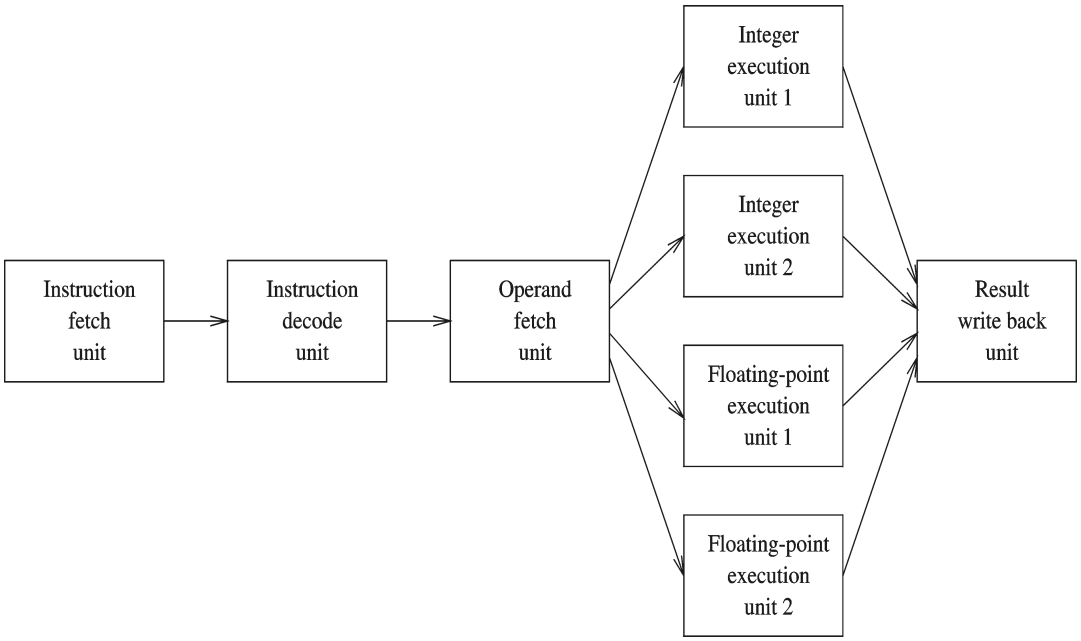
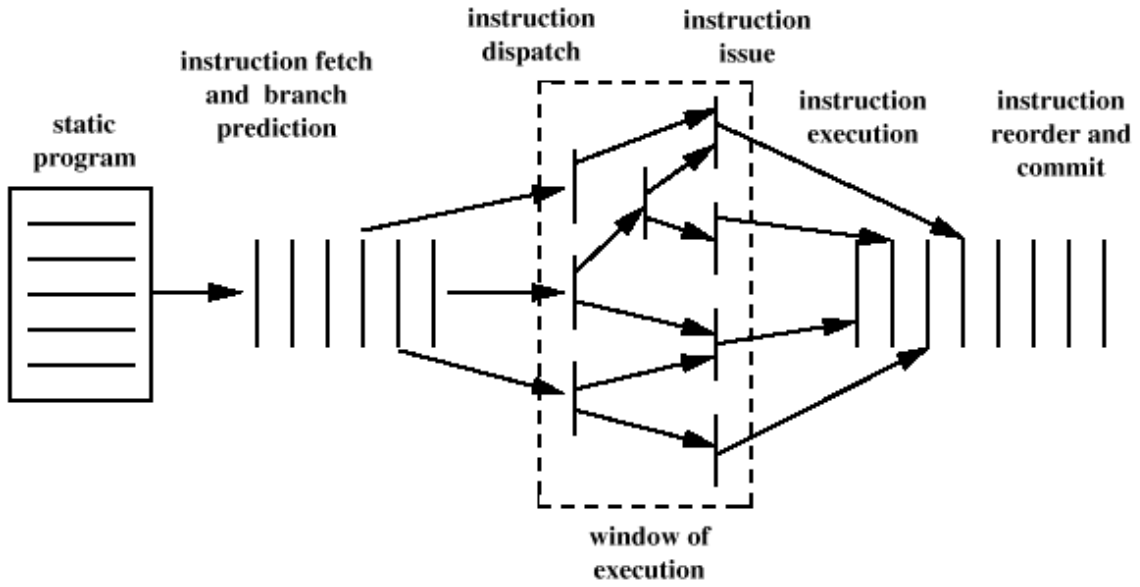


Figure 14.6: Conceptual Depiction of Superscalar Processing



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 a branch until the 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
```

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

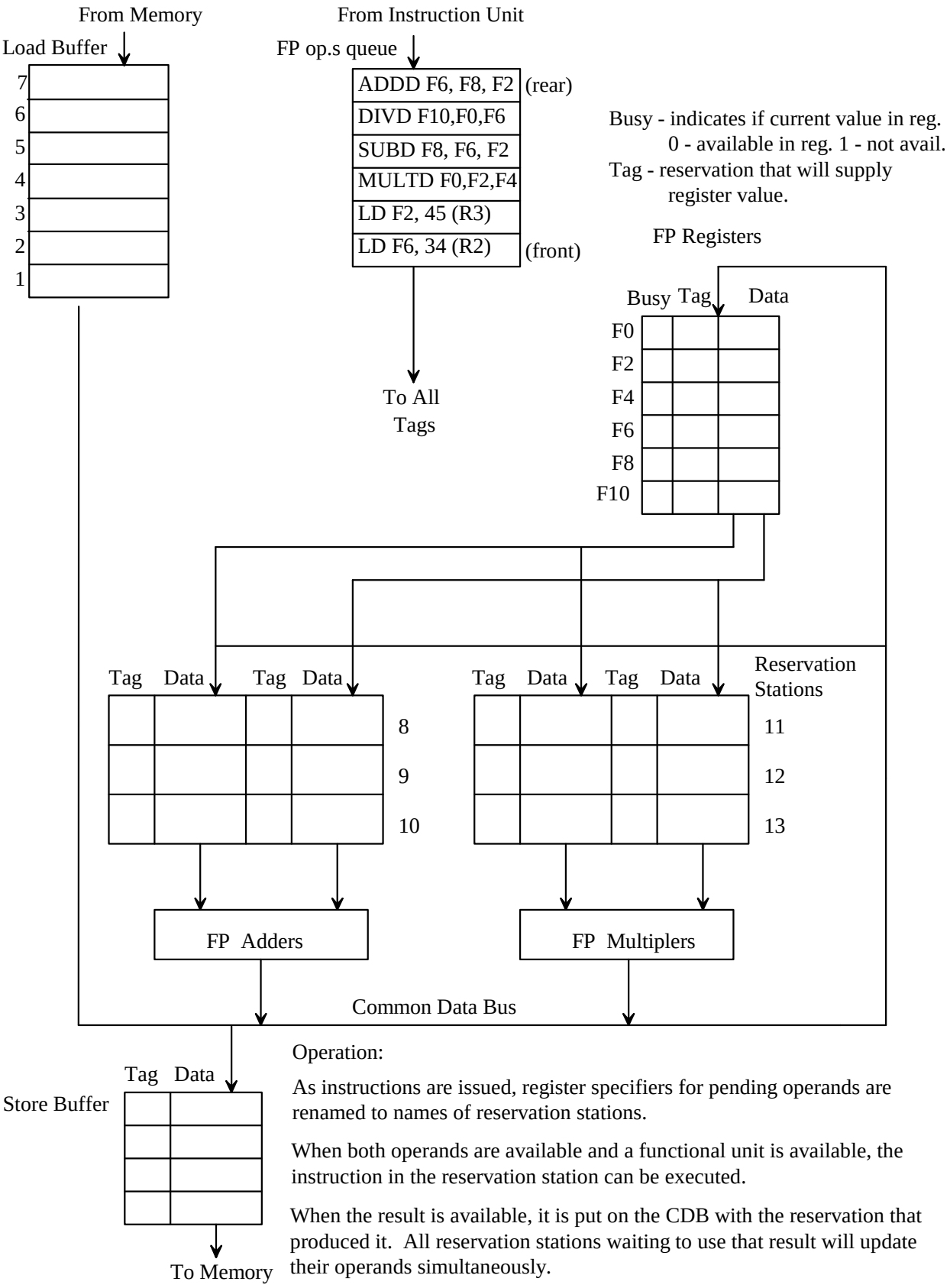
Antidependency / WAR (Write-After-Read)

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 $R4_b$



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.

Tomasulo's Algorithm is an example of ***dynamic scheduling***. In dynamic scheduling the ID - WB stages of the five-stage RISC pipeline are 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 bypassed 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.

Smith ’95 Studied the relationship between out-of-order issue, duplication of resources, and register renaming on R2000 architecture. (Figure 14.5)

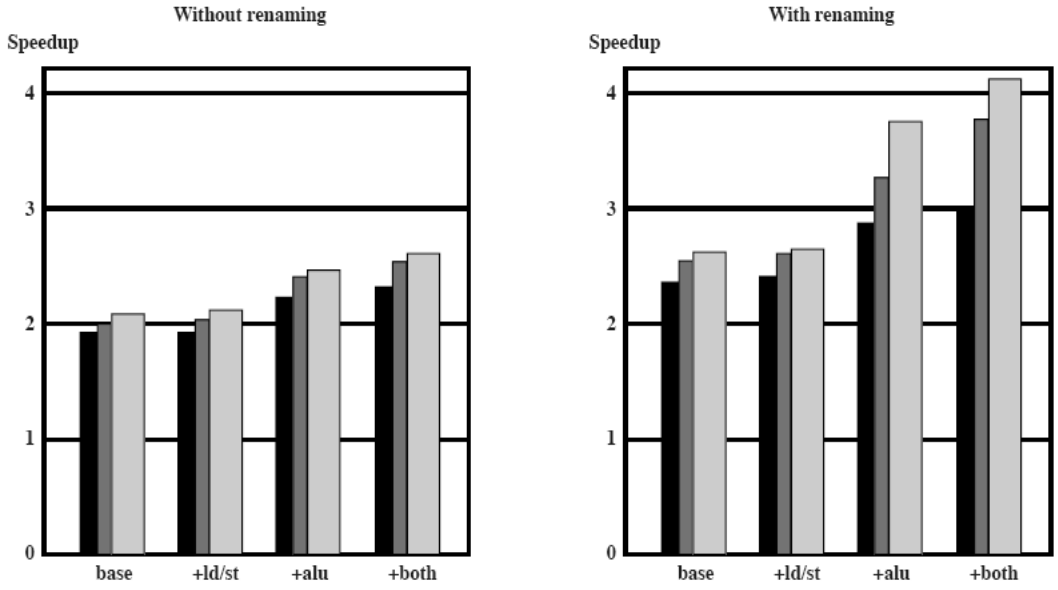


Figure 14.5 Speedups of Various Machine Organizations Without Procedural Dependencies

1) base machine - no duplicate functional units, but can issue out-of-order

2) + ld/st: duplicates load / store functional unit that access data cache

3) + alu: duplicates ALU

4) + both: duplicates both load/store and ALU

Differences shown for window sizes of 8, 16, and 32 instructions with and without register renaming

Conclusions: study shows 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

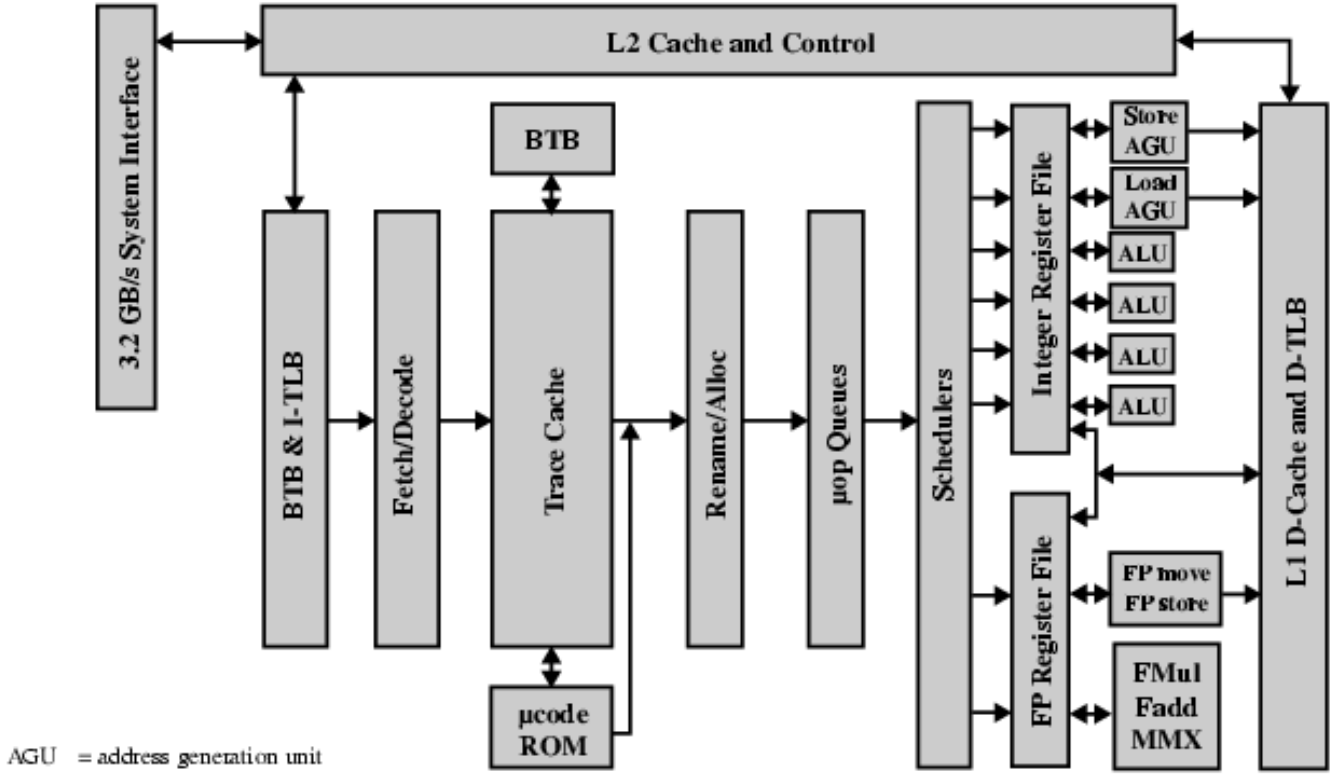
Committing / Retiring Step - needed since instructions may complete out-of-order

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.

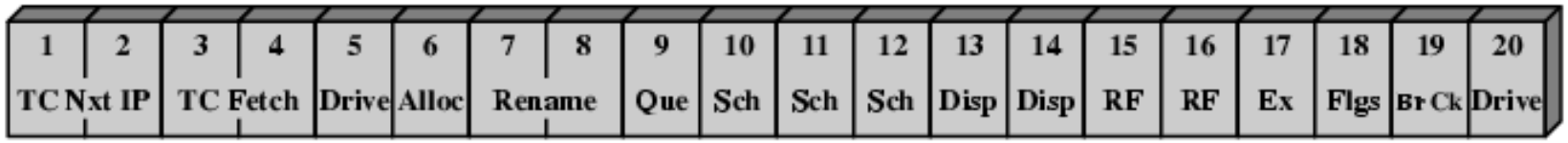
Pentium 4 Processor

- 80486 - CISC
- Pentium
  - some superscalar components
  - two separate integer execution units
- Pentium Pro – Full blown superscalar
- Subsequent models refine & enhance superscalar design

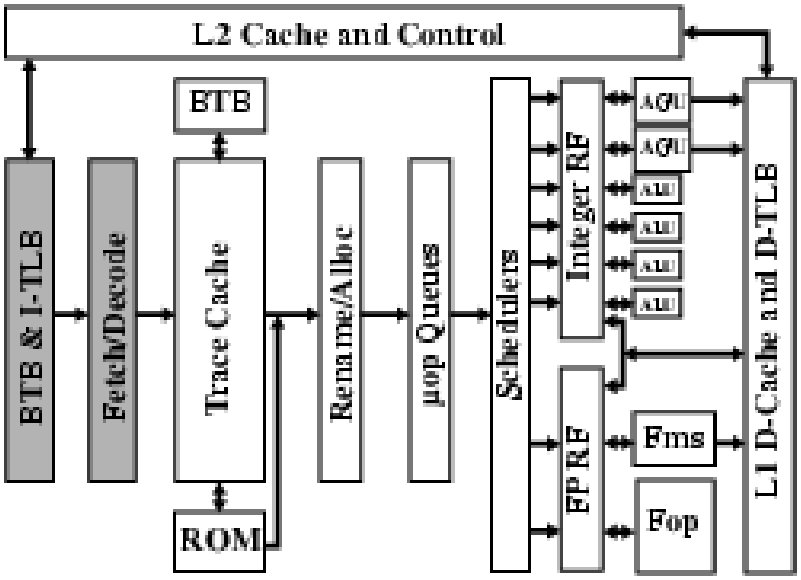


Pentium 4 Operation:

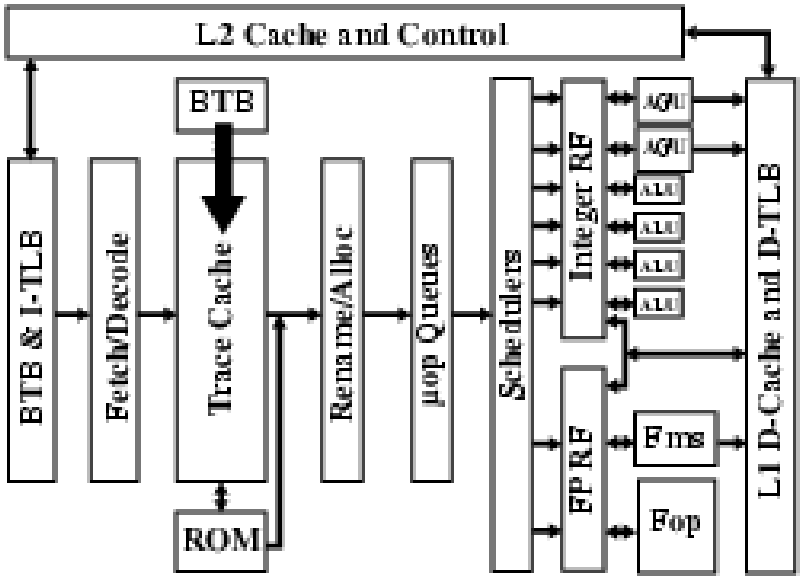
- Fetch CISC x86 instructions form memory in order of static program
- Translate instruction into one or more fixed length RISC instructions (micro-operations)
- Execute micro-ops on superscalar pipeline
  - micro-ops may be executed out of order
- Commit results of micro-ops to register set in original program flow order
- Outer CISC shell with inner RISC core
- Inner RISC core pipeline at least 20 stages
  - Some micro-ops require multiple execution stages
  - Longer pipeline than five stage pipeline on x86 up to Pentium



TC Next IP = trace cache next instruction pointer
TC Fetch = trace cache fetch
Alloc = allocate
Rename = register renaming
Que = micro-op queuing
Sch = micro-op scheduling
Disp = Dispatch
RF = register file
Ex = execute
Flgs = flags
Br Ck = branch check

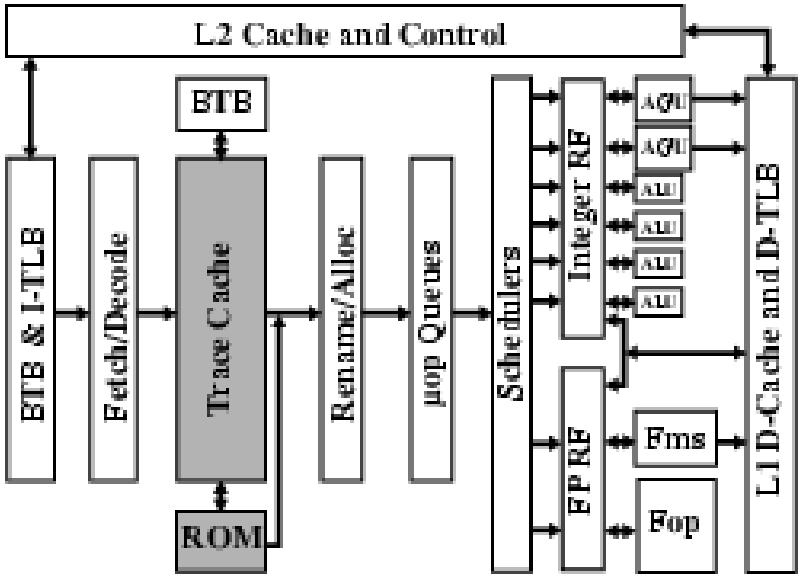


(a) Generation of micro-ops

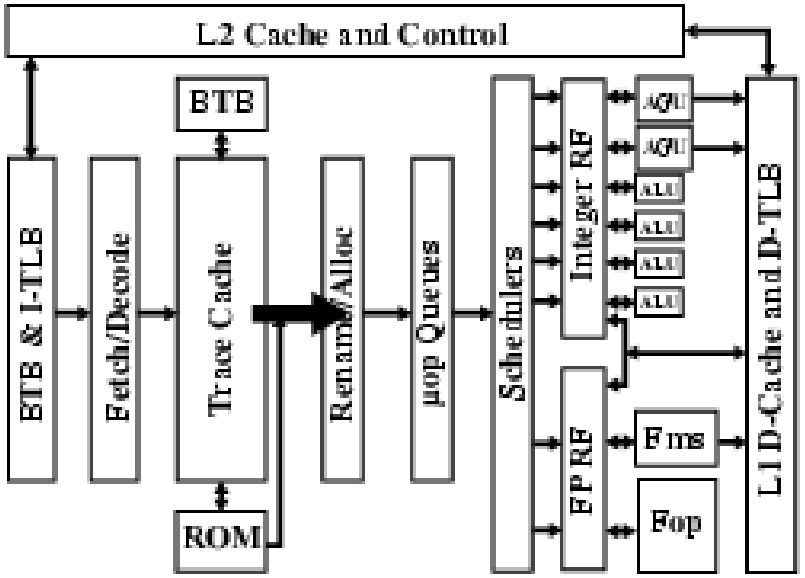


(b) Trace cache next instruction pointer

a) Fetch 64 bytes of Pentium 4 (CISC) instruction(s) from L2 cache and decode instruction boundaries and translates Pentium 4 (CISC) intructions into micro-op's (RISC)	b) Trace cache (L1 cache) stores recently executed mico-op's BTB uses dynamic branch prediction (4-bits used via Yeh's algorithm). Static prediction used if not in BTB.



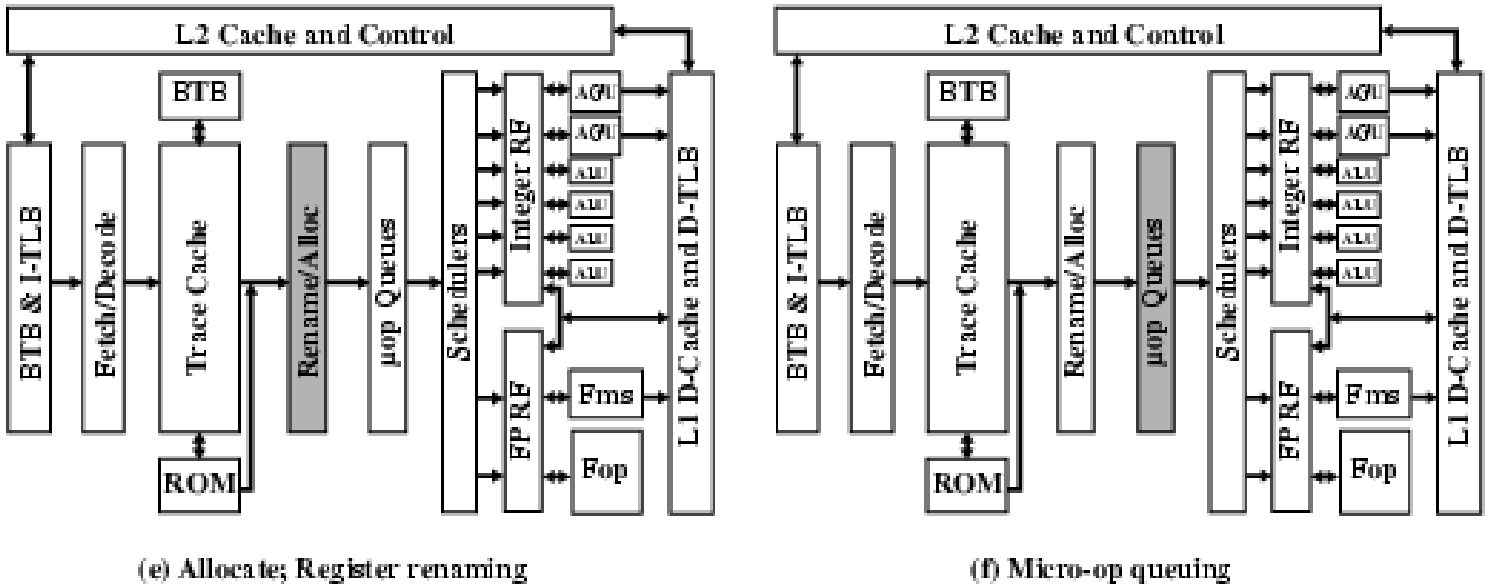
(c) Trace cache fetch



(d) Drive

c) Pulls micro-ops from cache (or ROM microprogrammed control unit for very complex instructions) in program sequence order	d) Drive delivers decoded instructions from the trace cache to the rename/allocate module.

Out-of-Order Execution Logic:

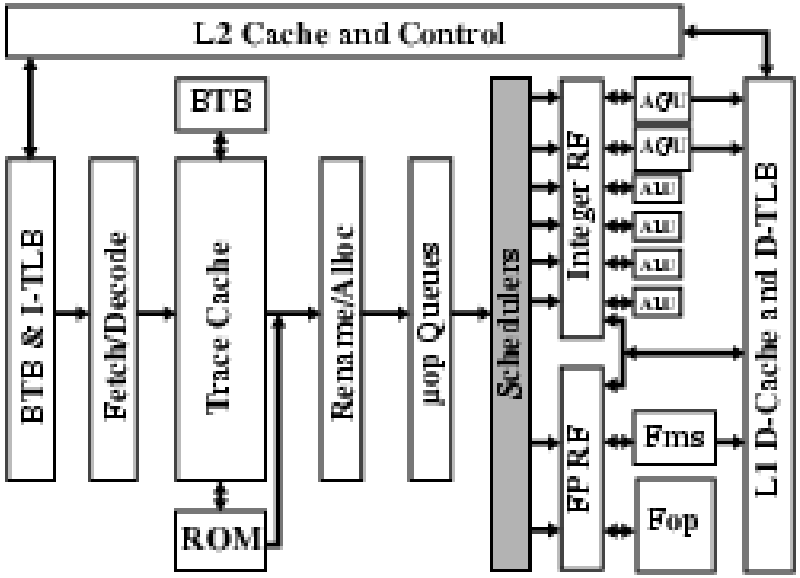


(e) Allocate; Register renaming

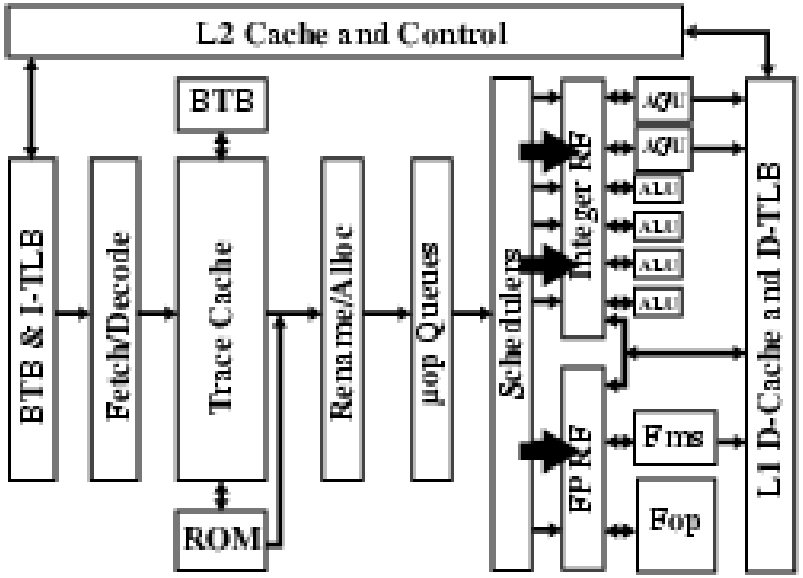
(f) Micro-op queuing

(ROB entry contains: state, memory address of generating instruction, micro-op, renamed register

Allocate - allocates resources needed for execution: • stalls pipeline if a resource (e.g., register) is unavailable • a reorder buffer (ROB) to store information about a micro-op as it executes • one of 128 integer or float registers for the result and/or one of 48 load buffers or one of 24 store buffers • an entry in one of the two micro-op queues	Two FIFO queues to hold micro-ops until there is room in the scheduler. One queue holds load or stores micro-ops One queue hold the remaining nonmemory micro-ops Queues can operate at different speeds

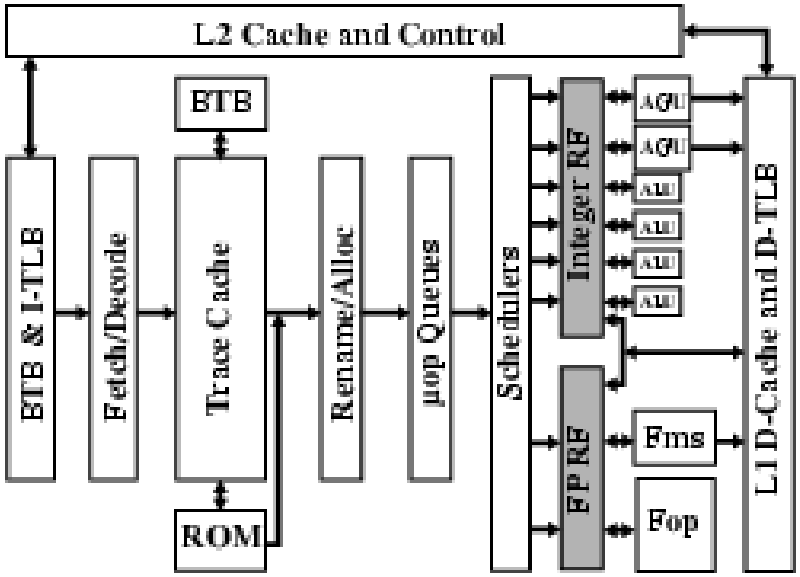


(g) Micro-op scheduling

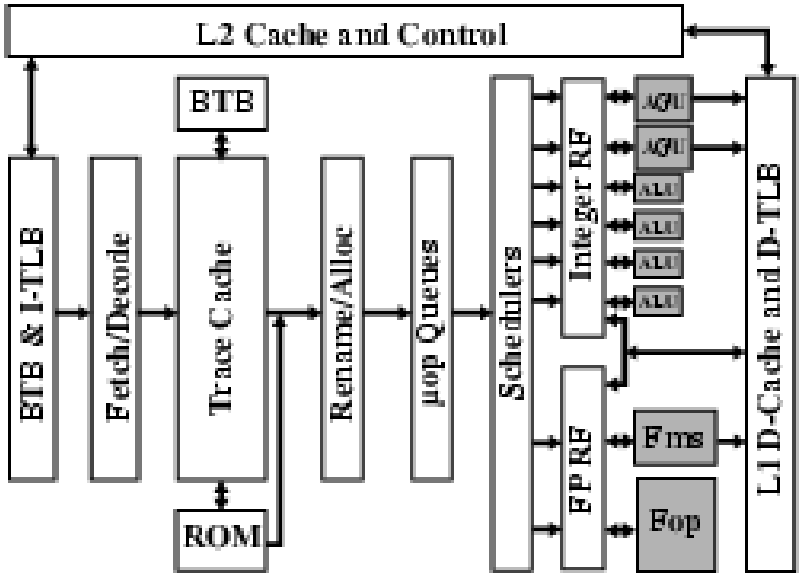


(h) Dispatch

Scheduler retrieves micro-ops from queues for dispatching/issuing for execution if all operands and execution unit are available.	Up to 6 micro-ops can be dispatched per cycle.

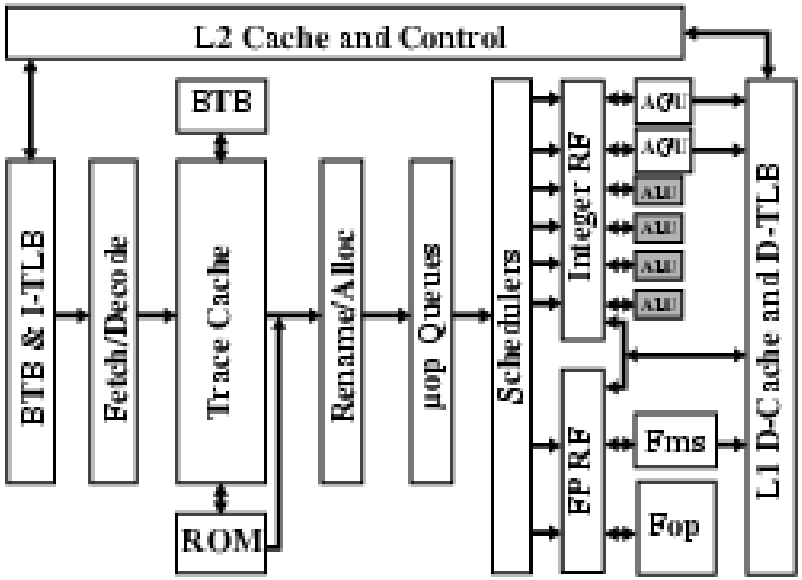


(i) Register file

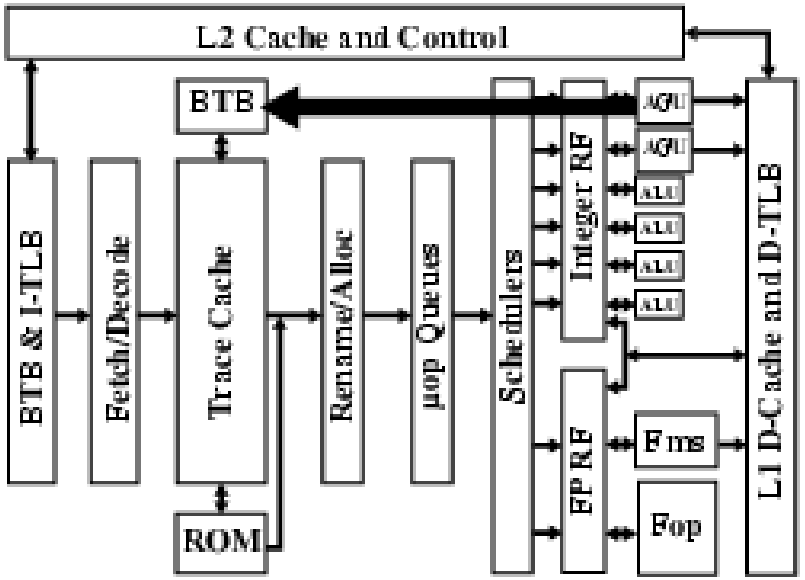


(j) Execute; flags

Execution units retrieve necessary integer and floating point registers	Compute flags - N, Z, C, V to use an input to the branches



(k) Branch check



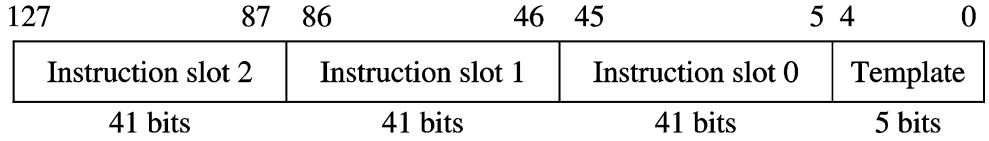
(l) Branch check result

Compares the actual branch result with the prediction.	If branch outcome does not match prediction, remove micro-ops from the pipeline. Provide proper branch destination to the BTB which restarts the whole pipeline from the correct target address.

Itanium Processor

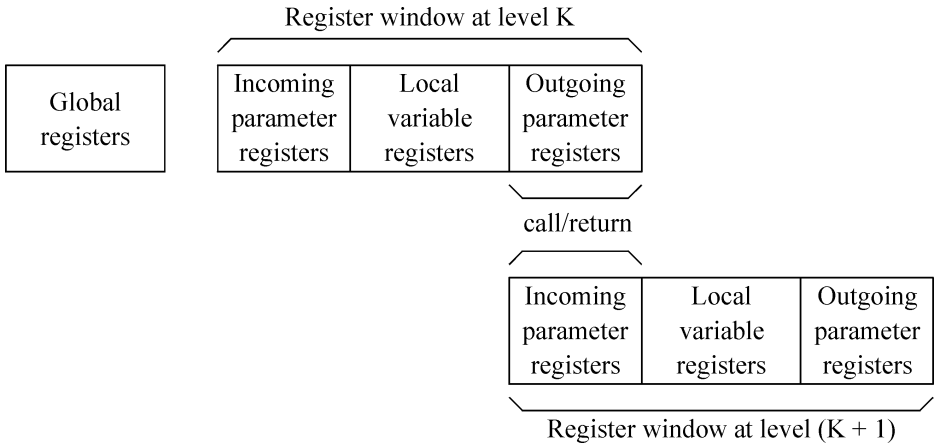
Interesting Features:

- 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