**Instruction Level Parallelism (ILP)**

**ILP:** The simultaneous execution of multiple instructions from a program. While pipelining is a form of ILP, the general application of ILP goes much further into more aggressive techniques to achieve parallel execution of the instructions in the instruction stream.
Exploiting ILP

Two basic approaches:

▶ rely on hardware to discover and exploit parallelism dynamically, and
▶ rely on software to restructure programs to statically facilitate parallelism.

These techniques are complimentary. They can be and are used in concert to improve performance.
Basic Block: a straight-line code sequence with a **single** entry point *and* a **single** exit point.

Remember, the average branch frequency is 15%–25%; thus there are only 3–6 instructions on average between a pair of branches.

**Loop level parallelism**: the opportunity to use multiple iterations of a loop to expose additional parallelism. Example techniques to exploit loop level parallelism: vectorization, data parallel, loop unrolling.
3 types of dependencies:

- data dependencies (or true data dependencies),
- name dependencies, and
- control dependencies.

Dependencies are artifacts of programs; hazards are artifacts of pipeline organization. Not all dependencies become hazards in the pipeline. That is, dependencies may turn into hazards within the pipeline depending on the architecture of the pipeline.
Data Dependencies

Assume that instruction $i$ executes before $j$, then:

An instruction $j$ is **data dependent** on instruction $i$ if:

- instruction $i$ produces a result that may be used by instruction $j$; or
- instruction $j$ is data dependent on instruction $k$, and instruction $k$ is data dependent on instruction $i$. 
Name Dependencies (not true dependencies)

Occurring when two instructions use the same register or memory location, but there is no flow in data between the instructions associated with that name.

There are 2 types of name dependencies between an instruction \( i \) that precedes instruction \( j \) in the execution order:

- **antidependence**: when instruction \( j \) writes a register or memory location that instruction \( i \) reads.

- **output dependence**: when instruction \( i \) and instruction \( j \) write the same register or memory location.

Because these are not true dependencies, the instructions \( i \) and \( j \) could potentially be executed in parallel if these dependencies are somehow removed (by using distinct register or memory locations).
A hazard exists whenever there is a name or data dependence between two instructions and they are close enough that their overlapped execution would violate the program’s order of dependency.

Possible data hazards:

- RAW (read after write)
- WAW (write after write)
- WAR (write after read)

RAR (read after read) is not a hazard.
Control Dependencies

Dependency of instructions to the sequential flow of execution and preserves branch (or any flow altering operation) behavior of the program.

In general, two constraints are imposed by control dependencies:

► An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch.

► An instruction that is not control dependent on a branch cannot be moved after the branch so that the execution is controlled by the branch.
Strictly enforcing dependency relations is not entirely necessary if we can preserve the correctness of the program. Two properties critical to program correctness (and normally preserved by maintaining both data and control dependency) are:

- **exception behavior**: reordering instructions must not alter the exception behavior of the program — specifically the visible exceptions (transparent exceptions such as page faults are ok to affect).

- **data flow**: data sources must be preserved to feed the instructions consuming that data.

*Liveness*: data that is needed is called *live*; data that is no longer used is called *dead*. 
Compiler Techniques for Exposing ILP

- loop unrolling
- instruction scheduling
Advanced Branch Prediction

- Correlating branch predictors (or two-level predictors): make use of outcome of most recent branches to make prediction.

- Tournament predictors: run multiple predictors and run a tournament between them; use the most successful.
Correlating branch predictors

(m, n) correlating branch predictor

- m: length of branch history
- n: number of prediction bits
Correlating Branch Prediction

The diagram shows the frequency of mispredictions for various SPEC98 benchmarks. The x-axis represents the frequency of mispredictions ranging from 0% to 16%, while the y-axis lists the benchmarks.

- **nASA7**: 1% 0% 1%
- **matrix300**: 0% 0% 0%
- **tomcatv**: 1% 0% 1%
- **dodox**: 5% 5% 5%
- **spice**: 9% 9% 9%
- **fpfppp**: 9% 9% 9%
- **gcc**: 11% 11% 11%
- **espresso**: 4% 5% 5%
- **eqntott**: 6% 6% 6%
- **li**: 5% 5% 5%

There are two types of entries: 4096 entries with 2 bits per entry and 1024 entries with unlimited entries and 2 bits per entry.
Correlating branch predictors (gshare)

Figure 3.4 A gshare predictor with 1024 entries, each being a standard 2-bit predictor.
Measured misprediction rates: SPEC89

![Graph showing misprediction rates vs. total predictor size for different predictor types: Local 2-bit predictors, Correlating predictors, and Tournament predictors. The graph displays a downward trend as predictor size increases, indicating lower misprediction rates.]
Tagged Hybrid Predictors

Figure 3.7 A five-component tagged hybrid predictor has five separate prediction tables, indexed by a hash of the branch address and a segment of recent branch history of length 0–4 labeled "h" in this figure. The hash can be as simple as an exclusive-OR, as in gshare. Each predictor is a 2-bit (or possibly 3-bit) predictor. The tags are typically 4–8 bits. The chosen prediction is the one with the longest history where the tags also match.
Performance of Hybrid Predictors

Figure 3.8 A comparison of the misprediction rate (measured as mispredicts per 1000 instructions executed) for tagged hybrid versus gshare. Both predictors use the same total number of bits, although tagged hybrid uses some of that storage for tags, while gshare contains no tags. The benchmarks consist of traces from SPECfp and SPECint, a series of multimedia and server benchmarks. The latter two behave more like SPECint.
End section 3.1-3.3

Lecture break; continue section 3.4-3.9 next class
Dynamic Scheduling

Designing the hardware so that it can dynamically rearrange instruction execution to reduce stalls while maintaining data flow and exception behavior.

Two techniques:

- Scoreboarding (discussed in Appendix C), centralized scoreboard
- Tomasulo’s Algorithm (discussed in Chapter 3), distributed reservation stations
Dynamic Scheduling

Instructions are issued to the pipeline *in-order* but executed and completed *out-of-order*.

*out-of-order execution* leading to the possibility of *out-of-order completion*.

*out-of-order execution* introduces the possibility of WAR and WAW hazards which do not occur in statically scheduled pipelines.

*out-of-order completion* creates major complications in exception handling.
Register renaming: to minimize WAW and WAR hazards (caused by name dependencies).

Reservation station: buffering operands for instructions waiting to execute. Operands are fetched as soon as they are available (from the reservation station/instruction serving the operand). The register specifiers of issued instructions are renamed to the reservation station (to achieve register renaming).

Thus, hazard detection and execution control are distributed (the targeted functional unit and reservation station determine when).

Common data bus/result bus/etc: carries results past the reservation stations (where they are captured) and back to the register file. Sometimes multiple buses are used.
Hardware Speculation

Execute (completely) the instructions that are predicted to occur after a branch w/o knowing the branch outcome. Speculate that the instructions are to be executed.

instruction commit: when the results (operand writes/exceptions) of the speculated instruction are made.

Key to speculation is to allow out-of-order instruction execute but force them to commit in order. Generally achieved by a reorder buffer (ROB) which holds completed instructions and retires them in order. Thus ROB holds/buffers register/memory operands that are also used by functional units for source operands.
Multiple-Issue Processors

Attempting to reduce the CPI below one.

1. Statically scheduled superscalar processors
2. VLIW (very long instruction word) processors
3. Dynamically scheduled superscalar processors

**superscalar**: scheduling multiple instructions for execution at the same time.
branch-target buffer or branch-target cache: predict the destination PC address for a branch instruction; can also store the target instruction (instead of, or in addition to the destination address). branch folding: overwrite the branch instr with the destination instruction.

Return address prediction.

Integrated instruction fetch units. Autonomously executing units to fetch instructions.

Comments on speculation: how much? through multiple branches? energy costs?

Value prediction: not yet feasible; not sure why it’s covered.
<table>
<thead>
<tr>
<th>Common name</th>
<th>Issue structure</th>
<th>Hazard detection</th>
<th>Scheduling</th>
<th>Distinguishing characteristic</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superscalar (static)</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Static</td>
<td>In-order execution</td>
<td>Mostly in the embedded space: MIPS and ARM, including the Cortex-A53</td>
</tr>
<tr>
<td>Superscalar (dynamic)</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Dynamic</td>
<td>Some out-of-order execution, but no speculation</td>
<td>None at the present</td>
</tr>
<tr>
<td>Superscalar (speculative)</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Dynamic with speculation</td>
<td>Out-of-order execution with speculation</td>
<td>Intel Core i3, i5, i7; AMD Phenom; IBM Power 7</td>
</tr>
<tr>
<td>VLIW/LIW</td>
<td>Static</td>
<td>Primarily software</td>
<td>Static</td>
<td>All hazards determined and indicated by compiler (often implicitly)</td>
<td>Most examples are in signal processing, such as the TI C6x</td>
</tr>
<tr>
<td>EPIC</td>
<td>Primarily static</td>
<td>Primarily software</td>
<td>Mostly static</td>
<td>All hazards determined and indicated explicitly by the compiler</td>
<td>Itanium</td>
</tr>
</tbody>
</table>
SuperScalar w/ Speculation
Branch-Target Buffer

PC of instruction to fetch

Look up

Predicted PC

Number of entries in branch-target buffer

No: instruction is not predicted to be branch; proceed normally

= Yes: then instruction is branch and predicted PC should be used as the next PC

Branch predicted taken or untaken
End section 3.4-3.9

Lecture break; continue section 3.10-3.11 next class
CPI of i7 Processors
How much ILP is available? Is there a limit?

Consider the ideal processor:

1. Infinite number of registers for register renaming.
2. Perfect branch prediction
3. Perfect jump prediction
4. Perfect memory address analysis
5. All memory accesses occur in one cycle

Effectively removing all control dependencies and all but true data dependencies. That is, all instructions can be scheduled as early as their data dependency allows.
The Power7 currently issues up to six instructions per clock.
- Up to 64 instructions issued/cycle
- Tournament predictor; best available in 2011 (not a primary bottleneck)
- Perfect memory address analysis
- 64 int/64 FP extra registers available for renaming
- Single cycle execution
Fine-grained multithreading: switch threads at each clock cycle. Examples: Sun Niagara processor (8 threads), Nvidia GPUs.

Course-grained multithreading: switch threads at major stalls such as L2/L3 misses. Examples: no commercial ventures.

Simultaneous multithreading: process/dispatch multiple threads simultaneously to the common functional units. Examples: Intel (hyperthreading, two threads), IBM Power7 (four threads).