Instruction-Level Parallelism

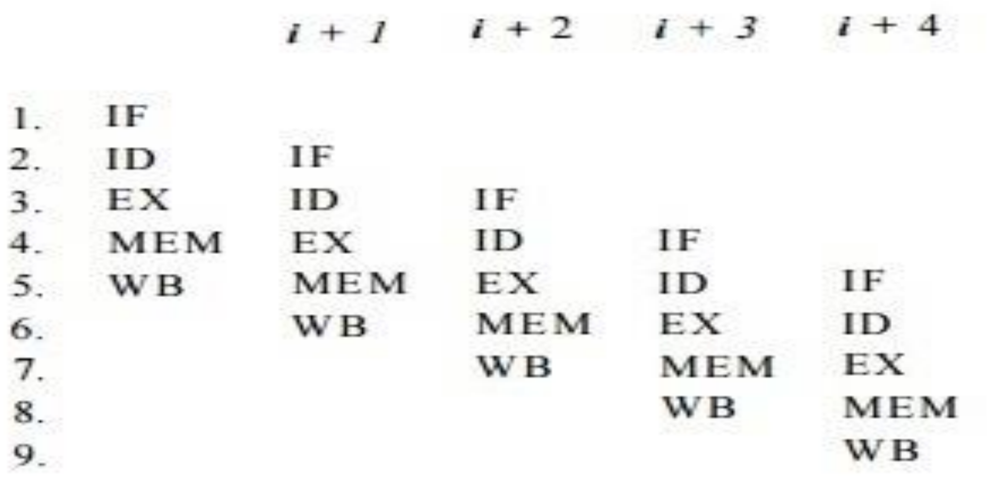
Processor Architecture

Processor Architecture

- When we think of instruction-level parallelism, we usually imagine a processor issuing several operations in a single clock cycle. In fact, it is possible for a machine to issue just one operation per clock and yet achieve instruction-level parallelism using the concept of pipelining.

Instruction Pipelines and Branch Delays

- Practically every processor, be it a high-performance supercomputer or a standard machine, uses an instruction pipeline. With an instruction pipeline, a new instruction can be fetched every clock while preceding instructions are still going through the pipeline.
- Shown in below Fig. is a simple 5-stage instruction pipeline: it first fetches the instruction (IF), decodes it (ID), executes the operation (EX), accesses the memory (MEM), and writes back the result (WB).



		i + 1	i + 2	i + 3	i + 4
1.	IF				
2.	ID	IF			
3.	EX	ID	IF		
4.	MEM	EX	ID	IF	
5.	WB	MEM	EX	ID	IF
6.		WB	MEM	EX	ID
7.			WB	MEM	EX
8.				WB	MEM
9.					WB

- The figure shows how instructions i, i + 1, i + 2, i + 3, and i + 4 can execute at the same time. Each row corresponds to a clock tick, and each column in the figure specifies the stage each instruction occupies at each clock tick.

Instruction Pipelines and Branch Delays

- If the result from an instruction is available by the time the succeeding instruction needs the data, the processor can issue an instruction every clock.
- Branch instructions are especially problematic because until they are fetched, decoded and executed, the processor does not know which instruction will execute next. Many processors speculatively fetch and decode the immediately succeeding instructions in case a branch is not taken. When a branch is found to be taken, the instruction pipeline is emptied and the branch target is fetched.
- Thus, taken branches introduce a delay in the fetch of the branch target and introduce "hiccups" in the instruction pipeline. Advanced processors use hard-ware to predict the outcomes of branches based on their execution history and to pre-fetch from the predicted target locations. Branch delays are nonetheless observed if branches are mis-predicted.

Pipelined Execution

- Some instructions take several clocks to execute.
- One common example is the memory-load operation. Even when a memory access hits in the cache, it usually takes several clocks for the cache to return the data.
- We say that the execution of an instruction is pipelined if succeeding instructions not dependent on the result are allowed to proceed.
- Thus, even if a processor can issue only one operation per clock, several operations might be in their execution stages at the same time.
- If the deepest execution pipeline has n stages, potentially n operations can be "in flight" at the same time.
- Note that not all instructions are fully pipelined. While floating-point adds and multiplies often are fully pipelined, floating-point divides, being more complex and less frequently executed, often are not.
- Most general-purpose processors dynamically detect dependences between consecutive instructions and automatically stall the execution of instructions if their operands are not available. Some processors, especially those embedded in hand-held devices, leave the dependence checking to the software in order to keep the hardware simple and power consumption low. In this case, the compiler is responsible for inserting "noop" instructions in the code if necessary to assure that the results are available when needed.

Multiple Instruction Issue

- By issuing several operations per clock, processors can keep even more operations in flight. The largest number of operations that can be executed simultaneously can be computed by multiplying the instruction issue width by the average number of stages in the execution pipeline.
- Like pipelining, parallelism on multiple-issue machines can be managed either by software or hardware.
- Machines that rely on software to manage their parallelism are known as VLIW (Very-Long-Instruction-Word) machines, while those that manage their parallelism with hardware are known as superscalar machines.
- Simple hardware schedulers execute instructions in the order in which they are fetched.
- If a scheduler comes across a dependent instruction, it and all instructions that follow must wait until the dependences are resolved (i.e., the needed results are available).
- Such machines obviously can benefit from having a static scheduler that places independent operations next to each other in the order of execution.

Multiple Instruction Issue

- More sophisticated schedulers can execute instructions "out of order."
- Operations are independently stalled and not allowed to execute until all the values they depend on have been produced.
- Even these schedulers benefit from static scheduling, because hardware schedulers have only a limited space in which to buffer operations that must be stalled. Static scheduling can place independent operations close together to allow better hardware utilization.
- More importantly, regardless how sophisticated a dynamic scheduler is, it cannot execute instructions it has not fetched. When the processor has to take an unexpected branch, it can only find parallelism among the newly fetched instructions.
- The compiler can enhance the performance of the dynamic scheduler by ensuring that these newly fetched instructions can execute in parallel.

Code-Scheduling Constraints

Code-Scheduling Constraints

- Code scheduling is a form of program optimization that applies to the machine code that is produced by code generator.
- Code scheduling subject to three kinds of constraints:
  1. Control dependence constraints: All the operations executed in the original program must be executed in the optimized one.
  2. Data dependence constraints: The operations in the optimized program must produce the same result as the corresponding ones in the original program.
  3. Resource constraints: The schedule must not oversubscribe the resources one the machine.

Code-Scheduling Constraints

- These scheduling constraints guarantee that the optimized program produces the same result as the original.
- However, because code scheduling changes the order in which the operation execute, the state of the memory at any one point may not match any of the memory states in a sequential execution.
- This situation is a problem if a program’s execution is interrupted by, for example, a thrown exception or a user interested breakpoint.
- Optimized programs are therefore harder to debug.

Basic-Block Scheduling

Basic-Block Scheduling

1. Data-Dependence Graphs
2. List Scheduling of Basic Blocks
3. Prioritized Topological Orders

Data-Dependence Graphs

- We represent each basic block of machine instructions by a data-dependence graph, G = (N,E), having a set of nodes N representing the operations in the machine instructions in the block and a set of directed edges E representing the data-dependence constraints among the operations. The nodes and edges of G are constructed as follows:

1. Each operation n in N has a resource-reservation table RT_n, whose value is simply the resource-reservation table associated with the operation type of n.
2. Each edge e in E is labeled with delay d_e indicating that the destination node must be issued no earlier than d_e clocks after the source node is issued. Suppose operation n± is followed by operation n_2, and the same location is accessed by both, with latencies l_1 and l_2 respectively. That is, the location's value is produced l_1 clocks after the first instruction begins, and the value is needed by the second instruction l_2 clocks after that instruction begins Then, there is an edge $n_1 \rightarrow n_2$ in E labeled with delay $l_1 - l_2$

List Scheduling of Basic Blocks

- The simplest approach to scheduling basic blocks involves visiting each node of the data-dependence graph in "prioritized topological order.
- Since there can be no cycles in a data-dependence graph, there is always at least one topological order for the nodes. However, among the possible topological orders, some may be preferable to others.
- There is some algorithm for picking a preferred order.

Prioritized Topological Orders

- List scheduling does not backtrack; it schedules each node once and only once. It uses a heuristic priority function to choose among the nodes that are ready to be scheduled next. Here are some observations about possible prioritized orderings of the nodes:
- Without resource constraints, the shortest schedule is given by the critical path, the longest path through the data-dependence graph. A metric useful as a priority function is the height of the node, which is the length of a longest path in the graph originating from the node.
- On the other hand, if all operations are independent, then the length of the schedule is constrained by the resources available. The critical resource is the one with the largest ratio of uses to the number of units of that resource available. Operations using more critical resources may be given higher priority.
- Finally, we can use the source ordering to break ties between operations; the operation that shows up earlier in the source program should be scheduled first