Instruction Level Parallelism
(concurrent instruction execution)

Bruce Shriver

University of Tromsø, Norway

Contemporary ILP Basics

- fetching, decoding, issuing, executing, completing, and retiring multiple operations in parallel
- in-order, out-of-order, and combinations of in-order and out-of-order
  - decoding, issuing, execution, completion, retirement
- dealing with hazards (data, procedural, resource conflicts, and others)
- exception and error handling

Basics (cont.)

- this is a processor centric view of the world; it is not a systems centric view
  - what memory architecture is required to sustain the required bandwidth for the target performance goal ?
  - how many local buses and special high bandwidth ports are needed for the target market functionality ?
  - what processor / core logic interface functionality must exist ? Does it have to be pipelined as well ?
  - etc.

Hazards Revisited

- (true) data dependencies
- procedural dependencies
- resource conflicts
- output dependencies
- antidependencies

Data Dependency

When an instruction J uses as input a value which has been produced as an output by another instruction I, we say the instruction that uses the value (J) has a *true data dependency* on the instruction which produced the value (I).

Data Dependency (cont.)

- example: scalar
- example: superscalar
- performance impact
  - zero-issue cycle
  - one-issue cycle
  - maximum-issue cycle

Data Dependency (cont.)

- need *dependency detection logic*
  - simple semantics
  - complicated semantics
- after detection
  - one solution: stall the pipeline
  - another solution: out-of-order execution
- when dependency is resolved, what happens obviously depends on solution

Procedural Dependency

- instructions following a conditional branch instruction are said to have a *procedural dependency* on the conditional branch instruction
- one solution: let the pipeline, starting with the first instruction following the conditional branch, wait until the outcome of branch is known

Procedural Dependency (cont.)

- example: scalar
- example: superscalar
- another solution: speculative execution

Speculative Execution

- predicting the outcome of the evaluation of the branch condition
- retaining machine state & undoing the effects of operations
- restoring the machine state if and when required to do so
- is it possible to support one or more threads of execution?

Resource Conflicts

- when two instructions must use the same resource at the same time, the two instructions share a *resource conflict*
- the term resource embodies the complete gamut of buses, datapaths, ports, registers, flip-flops, latches, ALUs, caches, memories, etc. the processor has direct access to or control over

Resource Conflicts (cont.)

- example: scalar
- example: superscalar
- one solution: stall one of the instructions
- do superscalar processors have an increased probability of resource conflicts?
- another solution: pipeline the resource in conflict

Resource Conflicts (cont.)

- yet another solution: provide multiple copies of the same resource
  - how will performance be impacted as you incrementally add additional copies of the resource ?
  - how many copies are needed ?
  - may be expensive all depending on the resource to be copied
- combined solution: replication + pipelining

Data Dependencies vs Resource Conflicts

- do they both effect performance in the same way?
- can both be eliminated by pipelining or replicating the resource causing the conflict?
- what if both are present at the same time in relatively they same ratio in the overall instruction mix?

Output Dependency

- when the destination of an instruction P is the same as the destination of a subsequent instruction S
- and it is possible that P would complete after S so that the value in the destination would be that of P when it ought to have been that of S. P and S have an *output dependency*.
- consider the following code sequence:

R[N] op R[J] → R[N]
R[N] + 1 → R[K]
R[J] + 1 → R[N]
R[K] op R[N] → R[L]

Antidependency

- when the destination of an instruction S is the same as one of the sources of a preceding instruction P
- and if S can give an incorrect value to P, S has an *antidependency* on P.
- consider the following code sequence:

R[1] / R[2] → R[3]
R[3] + R[4] → R[5]
R[6] + R[7] → R[3]

Different Terms For The Various Types of Data dependencies

Kogge	*Flynn*	*Johnson*
read after write	essential	data
write after write	output	output
write after read	ordering	anti

Some Clarifications

- *program dependencies* represent the actual data relationships that exist in the program
- *machine dependencies* exist because the program is executed on a machine and the machine's storage devices hold different values from time-to-time
- these are sometimes called *true dependencies* and *instruction dependencies*, respectively

Clarifications (cont.)

- atomic instructions that are issued in order and complete in order; there is a 1-1 map between storage locations and values
- when the instructions are non-atomic, or they are issued or complete out-of-order, the correspondence between storage locations and values breaks down

Clarifications

Johnson attributes to Backus the observation that antidependencies and output dependencies are more accurately described as resource conflicts. It is the attempted use and reuse of a resource (storage locations) by the instructions that causes the potential conflict with one another, even though they would in all other respects be conflict free.

Register Renaming

- one resolution to the resource conflict problem was to replicate the resource
- solution: provide additional registers and a mechanism to establish the map between the registers and the values
- this techniques is called *register renaming* and is described in Keller {1975}

Fundamental High-Performance ISA Problems

(often incorrectly couched in CISC/RISC terms)

- variability in instruction format and complexity of decode
- dependency detection
- high/low semantic content and degree of variability in instruction execution times
- the technology constraints for the target marketplace
- how will performance be impacted by implementing ILP
- exposing vs hiding possibilities

Fundamental Problems (cont.)

- revisit: M[x] + R[I] ➔M[x]
- non-pipelined scalar processor
  - atomic instructions
- for either pipelined scalar or superscalar
  - non-atomic instructions
  - each must make the other aware of its need to and its actual use of shared or contested resources

Performance Impacts of Stalls

- are dependencies more severe in superscalar processors?
  - e.g., do stalls prevent the execution of a potentially greater number of instructions
  - other examples?

The Road Ahead

- the obvious —
  - it can be difficult to determine conflicts among multiple instructions having complicated semantics
- the not so obvious —
  - excellent results abound in commercially successful RISC and CISC processors

In-Order Issue
In-Order Completion

- let's discuss Johnson's Figure 2.5
- pipeline stages are shown horizontally
- clock cycles are shown vertically
- "*This particular processor can decode two instructions, execute them in three functional units, and write two results per cycle (during the writeback stage of the pipeline). … the following constraints on parallelism:*
  - *I1 requires two cycles to execute*
  - *I3 and I4 conflict for a functional unit*
  - *I5 depends on the value produced by I4*
  - *I5 and I6 conflict for a functional unit*"

Assumptions

- what are the fundamental assumptions behind Johnson's example?
- can you make his description more precise?
- in-order issue with in-order completion is a rarely used policy in contemporary processors
- do you agree with this last statement?

In-Order Issue
Out-of-Order Completion

- let's use the example in Johnson's Fig. 2-6
- "*With out-of-order completion, any number of instructions is allowed to be in execution in the functional units, up to the total number of pipeline stages in all functional units.*"
- "*Instructions may complete out of order because instruction issuing is not stalled when a functional unit takes more than one cycle to complete an instruction.*"
- "*Consequently, a functional unit may complete an instruction after subsequent instructions have already completed.*"

Assumptions

- what are the fundamental assumptions behind Johnson's example?
- can you make his descriptions more precise?
- does the fact that instruction *I1* completes out-of-order "*improve*" the processor's "*lookahead capability*"?

In-Order Issue
Out-of-Order Completion (cont.)

- "*In a processor using out-of-order completion, instruction issuing is stalled when there is a conflict for a functional unit or when an issued instruction depends on a result that is not yet computed*."
- do you agree with this statement?

Some Implications of Out-of-Order Completion

- there must be a mechanism to ensure the results are written (completed) In-Program-Order
- dependency logic is more complicated
- why ?
- requires that functional units arbitrate for
  - result buses
  - register file ports
- why ?
- more difficult to deal with exception conditions

Implications (cont.)

- the exception condition may be associated with an instruction completing out-of-order
- subsequent instructions may also have completed out-of-order
- to what value is the Instruction Pointer set in the machine state for those exceptions that use interrupt handlers?
- it cannot be set to the instruction immediately after the exception was detected as is the normal case in atomic instruction execution
- why ?

Restarting After An Exception

- one solution: provide a mechanism that maintains a well-defined restart state *that is identical to* the state of a processor having in-order completion
- processors using this approach are said to support *precise exceptions*
- in this approach, the interrupt return address gives
  - the address of the instruction where the exception occurred
  - the address where the program should be restarted

Out-of-Order Issue
Out-of-Order Completion

- recall: "*In a processor using out-of-order completion, instruction issuing is stalled when there is a conflict for a functional unit or when an issued instruction depends on a result that is not yet computed*."
- the processor is not able to look ahead beyond the conflicting or dependent instruction

Out-of-Order Issue
Out-of-Order Completion (cont.)

- the decoder stage of the pipe needs to be isolated from the execution stage of the pipe
- use a buffer, often called an *instruction window,* which provides a pool of instructions for the instruction-issue policy
- instructions are fetched and decoded and placed in the window as long as there is room, only then is the fetch stage stalled

Out-of-Order Issue
Out-of-Order Completion (cont.)

- let's look at Johnson's Fig. 2-7
- instructions are issued from the pool of instructions in the window with little regard for the program order, but based on a dependency analysis and issue policy

Assumptions

- what are the fundamental assumptions behind Johnson's example?
- "Out-of-order issue simply gives the processor a larger set of instructions available for issue, improving its chances of finding instructions to execute concurrently."
- do you agree with this summary statement?

Revisiting The “Solutions”
(previously introduced)

- dependency detection logic
- out-of-order execution
- speculative execution
- pipeline resources in conflict
- replicate resources in conflict
- register renaming
- out-of-order completion
- precise interrupts
- out-of-order issue
- instruction window

Policy-Mechanism Separation

- what are the mechanisms used in the processor to fetch, decode, issue, execute, complete, and retire instructions?
- what are the policies that are employed to determine which instructions to fetch, when to fetch them, which to issue, when to issue them, to what function unit(s) will they will be issued, what resource(s) will they require, etc.?

Policy Implications

- intimately bound up with our list of “alternative solutions”
- example: instruction-issue policy
  - IF resource conflict THEN halt instruction fetching
    UNTIL the conflict is resolved
  - IF resource conflict THEN continue fetching instructions
    UNTIL an independent instruction is located
- *lookahead*: how many instructions ahead of the current instruction can the processor *examine*?

Bottom Line

- use the minimum complexity required that results in significant performance improvement and guarantees correct program execution
- recall the following transparency?
  - "*technology constraints*
    - *process and packaging technology*
    - *memory and cache architecture and technology*
    - *ILP theory and practice*
    - *compiler technology*
    - *knowledge of the execution behavior of complex systems (e.g., the OS & various application domains)*"
- and the one that followed it?

A Small Aside: Measures

- Stone's
- Flynn's
- Johnson's:
  - Machine Parallelism
  - Instruction Parallelism
- others
- obviously, for the moment, we'll only deal with Johnson's

Machine Parallelism

- a measure "*of the ability of the processor to take advantage of the instruction-level parallelism*." Determined by:
  - the number of instructions that can be fetched, decoded, issued, executed, completed, and retired at the same time
  - thus, it's tightly coupled to the mechanism the processor uses to identify independently executable instructions

Instruction Parallelism

- a measure of the "*average number of instructions that a superscalar processor might be able to execute at the same time*"
- determined by:
  - # of data dependencies and branches relative to the other instructions
  - latencies of the processor operations

Target Price-Performance Point

- there are obviously programs for which
  - all of the ILP cannot be realized on a given processor
  - there is so little ILP that much of the processor's resources go unused
- real systems often consist of a mix of the two
- how do you go about designing a processor to achieve balance and yield the lowest cost design giving the highest performance

Scheduling Goals

- the scheduler dispatches (issues) operations to the execution units
- it should:
  - be highly efficient to allow high clock rates
  - maximize parallel execution of operations
  - have as simple a design as possible to reduce circuit size & cost and minimize design errors