COMPUTER ORGANIZATION AND DESIGN
The Hardware/Software Interface
5th Edition

Chapter 4

The Processor
Advanced Issues

Review: Pipeline Hazards

- Structural hazards
  - Design pipeline to eliminate structural hazards.
- Data hazards – read before write
  - Use data forwarding inside the pipeline.
  - For those cases that forwarding won't solve (e.g., load-use) include hazard hardware to insert stalls in the instruction stream.
- Control hazards – `beq`, `bne`, `j`, `jr`, `jal`
  - Stall – hurts performance.
  - Move decision point as early in the pipeline as possible – reduces number of stalls at the cost of additional hardware.
  - Delay decision (requires compiler support) – may not feasible for deeper pipes.
  - Predict – with even more hardware, can reduce the impact of control hazard stalls even further if the branch prediction (**Branch History Table**) is correct and if the branched-to instruction is cached (**Branch Table Buffer**).

Exceptions and Interrupts

- "Unexpected" events requiring attention
  - Different ISAs use the terms differently.
- Exceptions (sometimes called Traps)
  - Arises within the CPU
    - e.g., undefined opcode, overflow, syscall, divide by zero, …
- Interrupt
  - Comes from an external I/O controller.
- Dealing with them without sacrificing performance is impossible.

Handling Exceptions

- In MIPS, exceptions managed by a System Control Coprocessor (CP0).
- Save PC of offending (or interrupted) instruction in the **Exception Program Counter** (EPC).
- Save indication of the problem in the **Cause Register.**
- Jump to handler at **hard address** 8000 0018.

An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause. This is very common in embedded processors.
- Example:
  - Undefined opcode: C000 0000
  - Overflow: C000 0020
  - …: C000 0040
- Instructions either:
  - Deal with the interrupt.
  - Jump to the real handler.
  - Pass control to the OS.

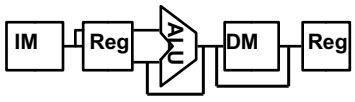
Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once.
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions.
  - “Precise” vs. “imprecise” exception approach.
- In complex pipelines
  - Multiple instructions issued per cycle.
  - Out-of-order completion.
  - Maintaining precise exceptions is difficult.

Precise vs. Imprecise Exceptions

- An interrupt that leaves the machine in a well-defined state is called a ***precise interrupt***. Such an interrupt has four properties:
  - The Program Counter (PC) is saved in a known place.
  - All instructions before the one pointed to by the PC have fully executed.
  - No instruction beyond the one pointed to by the PC has been executed.
  - The execution state of the instruction pointed to by the PC is known.
- An interrupt that does not meet these requirements is called an ***imprecise interrupt***.

Where in the Pipeline Exceptions Occur



	Stage(s)?	Synchronous?
Arithmetic overflow:	EX	yes
Undefined instruction:	ID	yes
TLB or page fault:	IF, MEM	yes
I/O service request:	any	no
Hardware malfunction:	any	no

- Multiple exceptions can occur simultaneously in a *single* clock cycle.

Extracting Yet *More* Performance

- ***Superpipelining -*** Increasing the depth of the pipeline to increase the clock rate
  - The more stages in the pipeline, the more forwarding/hazard hardware needed and the more pipeline latch overhead (i.e., the pipeline latch accounts for a larger and larger percentage of the clock cycle time).
- ***Multiple-issue –*** Fetching and executing more than one instruction at a time (expand every pipeline stage to accommodate multiple instructions)
  - The instruction execution rate, CPI, will be less than 1, so instead we use ***IPC -*** instructions per clock cycle
    - E.g., a 6 GHz, four-way multiple-issue processor can execute at a peak rate of 24 billion instructions per second with a best case CPI of 0.25 or a best case IPC of 4.

Types of Parallelism

- ***Instruction-level parallelism*** (**ILP**) – a measure of the average number of instructions in a program that a processor *might* be able to execute at the same time
  - Mostly determined by the number of data dependencies and control dependencies in relation to the number of other instructions.
- ***Machine-level parallelism*** – a measure of the ability of the processor to take advantage of the ILP of the program
  - Determined by the number of instructions that can be fetched and executed at the same time.
- To achieve high performance, we need *both* ILP and machine-level parallelism.

Instruction-Level Parallelism

- Pipelining: executing multiple instructions in parallel
- To increase ILP you need:
  - Deeper pipeline
    - Less work per stage ⇒ shorter clock cycle.
  - Multiple issue
    - Replicate pipeline stages ⇒ multiple pipelines.
    - Start multiple instructions per clock cycle.
    - But dependencies reduce this considerably in practice.

Multiple Issue

- Static multiple issue
  - Compiler groups instructions into "***issue packets***".
  - Compiler must detect and avoid hazards
    - E.g., Intel Itanium and Itanium 2 for the IA-64 ISA – EPIC (Explicit Parallel Instruction Computer).
    - 128-bit "bundles" containing three instructions, each 41-bits plus a 5-bit template field, which specifies which functional unit each instruction needs.
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle.
  - Compiler can help by reordering instructions.
  - CPU must resolve hazards at runtime.

Multiple-Issue Datapath Responsibilities

- Must handle, with a combination of hardware and software fixes, the fundamental limitations of:
  - How many instructions to issue in one clock cycle.
  - Data hazards
    - Limitation is more severe in a Superscaler/VLIW processor due to a (usually) lower ILP.
  - Control hazards
    - Must lean heavily on dynamic branch prediction to help resolve the ILP issue.
  - Structural hazards
    - A SS/VLIW processor has a much larger number of potential resource conflicts.
    - Functional units may have to arbitrate for result busses and register-file write ports.
    - Resource conflicts can be eliminated by duplicating the resource or by pipelining the resource.

Static Multiple Issue Machines (VLIW)

- Static multiple-issue processors (aka ***Very Long Instruction Word*** (**VLIW**) use the compiler to statically decide which instructions to issue and execute simultaneously:
  - Issue packet – the set of instructions that are bundled together and issued in one clock cycle – think of it as one *large* instruction with multiple operations.
  - The mix of instructions in the packet is usually restricted – a single "instruction" with several predefined fields.
  - The compiler does static branch prediction and code scheduling to reduce control or eliminate data hazards.
- VLIW's have
  - Multiple functional units.
  - Multi-ported register files.
  - Wide program busses.

Loop Unrolling

- ***Loop Unrolling*** is a loop transformation technique that attempts to optimize a program's execution speed at the expense of its binary size (space-time tradeoff). The transformation can be undertaken manually by the programmer or by an optimizing compiler.
- The goal of loop unrolling is to increase a program's speed by reducing (or eliminating) instructions that control the loop, such as pointer arithmetic and "end of loop" tests on each iteration; reducing branch penalties; as well as "hiding latencies, in particular, the delay in reading data from memory". To eliminate this overhead, loops can be re-written as a repeated sequence of similar independent statements.

Loop Unrolling Example

- A procedure in a computer program is to delete 100 items from a collection. This is normally accomplished by means of a *for*-loop which calls the function *delete(item_number)*.

Normal loop

```
int x;
for (x = 0; x < 100; x++)
{
   delete(x);
}
```

After loop unrolling

```
int x;
for (x = 0; x < 100; x+=5)
{
   delete(x);
   delete(x+1);
   delete(x+2);
   delete(x+3);
   delete(x+4);
}
```

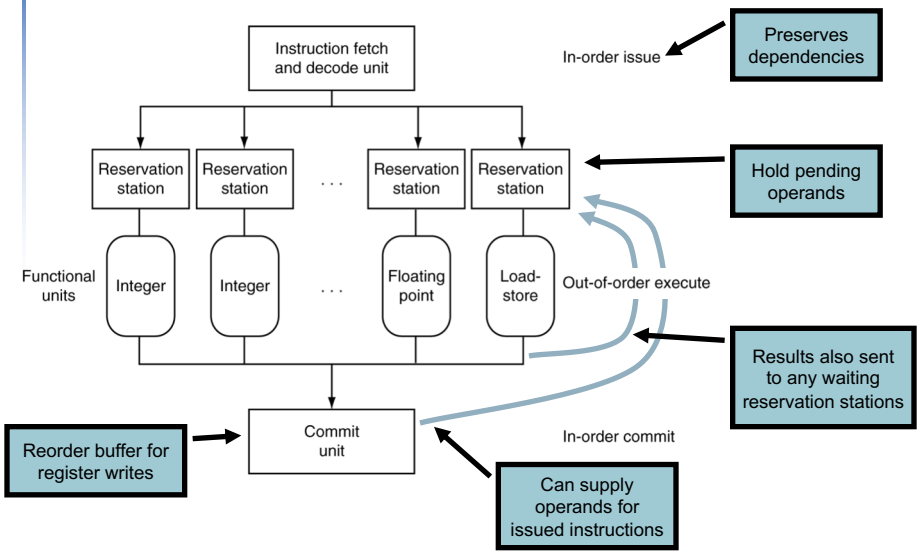
Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit results to registers in order.
- Example

```
lw    $t0, 20($s2)
addu  $t1, $t0, $t2
sub   $s4, $s4, $t3
slti  $t5, $s4, 20
```

  - Can start sub while addu is waiting for lw.

Dynamically Scheduled CPU



Register Renaming

- Reservation stations and the *reorder buffer* effectively provide ***register renaming.***
- On instruction issue to reservation station
  - If operand is available in register file or reorder buffer
    - Copied to reservation station.
    - No longer required in the register; can be overwritten.
  - If operand is not yet available
    - It will be provided to the reservation station by a functional unit.
    - Register update may not be required.

In-Order vs Out-of-Order

- Instruction fetch and decode units are ***required*** to issue instructions in-order so that dependencies can be tracked.
- The commit unit is ***required*** to write results to registers and memory in program fetch order so that:
  - If exceptions occur, the only registers updated will be those written by instructions before the one causing the exception.
  - If branches are mispredicted, those instructions executed after the mispredicted branch don't change the machine state (i.e., we use the commit unit to correct incorrect speculation).
- Although the front end (fetch, decode, and issue) and back end (commit) of the pipeline run in-order, the FUs are free to initiate execution whenever the data they need is available which can leads to out-of-order execution.
  - Allowing out-of-order execution increases the amount of ILP.

Speculation

- ***Speculation*** is used to allow execution of future instructions that (may) depend on the speculated instruction:
  - Speculate on the outcome of a conditional branch (*branch prediction*).
  - Speculate that a store (for which we don't yet know the address) that precedes a load, does not refer to the same address, allowing the load to be scheduled before the store (*load speculation*).
- Must have (hardware and/or software) mechanisms for:
  - Checking to see if the guess was correct.
  - Recovering from the effects of the instructions that were executed speculatively if the guess was incorrect.
- Ignore and/or buffer exceptions created by speculatively executed instructions until it is clear that they should really occur.

Predication

- ***Predication*** can be used to eliminate branches by making the execution of an instruction dependent on a "predicate", e.g.,

  ```
  if (p) {statement 1} else {statement 2}
  ```

  would normally compile using two branches. With predication, it would compile as:

  ```
  (p) statement 1
  (~p) statement 2
  ```

- Predication can be used to speculate as well as to eliminate branches.

Dependencies Review

- When more than one instruction references a particular location for an operand, either reading it (as an input) or writing it (as an output), executing those instructions in an order different from the original program order can lead to three kinds of ***data hazards***:
  - Read-after-write (RAW): A read from a register or memory location must return the value placed there by the last write in program order, not some other write. This is referred to as a **true dependency** or **flow dependency**, and requires the instructions to execute in program order.
  - Write-after-write (WAW): Successive writes to a particular register or memory location must leave that location containing the result of the second write. This can be resolved by **squashing** (synonyms: cancelling, annulling, mooting) the first write if necessary. WAW dependencies are also known as **output dependencies**.
  - Write-after-read (WAR): A read from a register or memory location must return the last prior value written to that location, and not one written programmatically after the read. This is the sort of **false dependency** that can be resolved by renaming. WAR dependencies are also known as **anti-dependencies**.

Dependency Example

- With out-of-order execution, a later instruction may execute *before* a previous instruction so the hardware needs to resolve both *read-before-write* ***and*** *write-before-write* data hazards

```
lw    $t0,0($s1)
addu  $t0,$t1,$s2
. . .
sub   $t2, $t0, $s2
```

  - If the `lw` write to `$t0` is executed after the `addu` write, then the `sub` gets an incorrect value for `$t0`
  - The `addu` has an output dependency on the `lw` – *write-before-write*
    - The issuing of the `addu` might have to be stalled if its result could later be overwritten by a previous instruction that takes longer to complete.

Antidependencies

- We also must deal with ***antidependencies*** – when a later instruction (that executes earlier) produces a data value that destroys a data value used as a source in an earlier instruction (that executes later).

```
R3 := R3 * R5
R4 := R3 + 1
R3 := R5 + 1
```

Antidependency
True data dependency
Output dependency

- The constraint is similar to that of true data dependencies, except *reversed:*
  - Instead of the later instruction using a value (not yet) produced by an earlier instruction (*read before write*), the later instruction produces a value that destroys a value that the earlier instruction (has not yet) used (*write before read*).

Does Multiple Issue Work?

- Yes, but not as much as we'd like.
- Programs have real dependencies that limit ILP.
- Some dependencies are hard to eliminate.
- Some parallelism is hard to expose
  - Limited window size during instruction issue.
- Memory delays and limited bandwidth
  - Hard to keep pipelines full.
- Speculation can help if done well.

Fallacies

- Pipelining is easy:
  - The basic idea is easy.
  - The devil is in the details, e.g., detecting data hazards.
- Pipelining is independent of technology:
  - So why haven't we always done pipelining?
  - More transistors make more advanced techniques feasible.
  - Pipeline-related ISA design needs to take account of technology trends.

Concluding Remarks

- ISA influences design of datapath and controller
  - Poor ISA design can make pipelining harder.
- Datapath and control influence design of ISA.
- Pipelining improves instruction throughput using parallelism:
  - More instructions completed per second.
  - Latency for each instruction is not reduced.
- Hazards: structural, data, control.
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism.
  - Complexity leads to the power wall.