CS G22.2233
Computer Systems Design
Spring 2007

Lecture 07: Multiple Issue Introduction

Mohammad Banikazemi

[Slides from Prof. Mary Jane Irwin, PSU Adapted from
Computer Organization and Design, Patterson & Hennessy, © 2005, UCB]
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 – \texttt{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) – not feasible for deeper pipes requiring more than one delay slot to be filled
  - Predict – with even more hardware, can reduce the impact of control hazard stalls even further if the branch prediction (BHT) is correct and if the branched-to instruction is cached (BTB)
Dealing with Exceptions

- Exceptions (aka interrupts) are just another form of control hazard. Exceptions arise from
  - R-type arithmetic overflow
  - Trying to execute an undefined instruction
  - An I/O device request
  - An OS service request (e.g., a page fault, TLB exception)
  - A hardware malfunction

- The pipeline has to stop executing the offending instruction in midstream, let all prior instructions complete, flush all following instructions, set a register to show the cause of the exception, save the address of the offending instruction, and then jump to a prearranged address (the address of the exception handler code)

- The software (OS) looks at the cause of the exception and “deals” with it
Two Types of Exceptions

- Interrupts – asynchronous to program execution
  - caused by external events
  - may be handled between instructions, so can let the instructions currently active in the pipeline complete before passing control to the OS interrupt handler
  - simply suspend and resume user program

- Traps (Exception) – synchronous to program execution
  - caused by internal events
  - condition must be remedied by the trap handler for that instruction, so much stop the offending instruction midstream in the pipeline and pass control to the OS trap handler
  - the offending instruction may be retried (or simulated by the OS) and the program may continue or it may be aborted
Where in the Pipeline Exceptions Occur

- Arithmetic overflow
- Undefined instruction
- TLB or page fault
- I/O service request
- Hardware malfunction
## Where in the Pipeline Exceptions Occur

<table>
<thead>
<tr>
<th>Exception</th>
<th>Stage(s)?</th>
<th>Synchronous?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic overflow</td>
<td>EX</td>
<td>yes</td>
</tr>
<tr>
<td>Undefined instruction</td>
<td>ID</td>
<td>yes</td>
</tr>
<tr>
<td>TLB or page fault</td>
<td>IF, MEM</td>
<td>yes</td>
</tr>
<tr>
<td>I/O service request</td>
<td>any</td>
<td>no</td>
</tr>
<tr>
<td>Hardware malfunction</td>
<td>any</td>
<td>no</td>
</tr>
</tbody>
</table>

- Beware that multiple exceptions can occur simultaneously in a *single* clock cycle
Hardware sorts the exceptions so that the earliest instruction is the one interrupted first.
Multiple Simultaneous Exceptions

- Hardware sorts the exceptions so that the earliest instruction is the one interrupted first.

Diagram showing multiple instructions and exceptions, including:
- Inst 0: IM -> Reg -> ALU -> DM -> Reg
- Inst 1: IM -> Reg -> ALU -> DM -> Reg
- Inst 2: IM -> Reg -> ALU -> DM -> Reg
- Inst 3: IM -> Reg -> ALU -> DM -> Reg
- Inst 4: IM -> Reg -> ALU -> DM -> Reg

Exceptions marked:
- D$ page fault
- Arithmetic overflow
- Undefined instruction
- I$ page fault
Additions to MIPS to Handle Exceptions (Fig 6.42)

- **Cause register** (records exceptions) – hardware to record the cause of the exceptions in Cause register and a signal to control writes to it (CauseWrite)

- **EPC register** (records the addresses of the offending instructions) – hardware to record in EPC the address of the offending instruction and a signal to control writes to it (EPCWrite)
  - Exception software must match exception to instruction

- **A way to load the PC with the address of the exception handler**
  - Expand the PC input mux where the new input is hardwired to the exception handler address - (e.g., 8000 0180hex for arithmetic overflow)

- **A way to flush offending instruction and the ones that follow it**
Datapath with Controls for Exceptions (Incomplete)
Summary

- All modern day processors use pipelining for performance (a CPI of 1 and fast a CC)
- Pipeline clock rate limited by slowest pipeline stage – so designing a balanced pipeline is important
- Must detect and resolve hazards
  - Structural hazards – resolved by designing the pipeline correctly
  - Data hazards
    - Stall (impacts CPI)
    - Forward (requires hardware support)
  - Control hazards – put the branch decision hardware in as early a stage in the pipeline as possible
    - Stall (impacts CPI)
    - Delay decision (requires compiler support)
    - Static and dynamic prediction (requires hardware support)
Extracting Yet *More* Performance

- Two options:
  - Increase the depth of the pipeline to increase the clock rate – **superpipelining** (more details to come)
  - Fetch (and execute) more than one instructions at one time (expand every pipeline stage to accommodate multiple instructions) – **multiple-issue**

- Launching multiple instructions per stage allows the instruction execution rate, CPI, to 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

- If the datapath has a five stage pipeline, how many instructions are active in the pipeline at any given time?
Superpipelined Processors

- Increase the depth of the pipeline leading to shorter clock cycles (and more instructions “in flight” at one time)
  - The higher the degree of superpipelining, the more forwarding/hazard hardware needed, the more pipeline latch overhead (i.e., the pipeline latch accounts for a larger and larger percentage of the clock cycle time), and the bigger the clock skew issues (i.e., because of faster and faster clocks)

Superpipelined vs Superscalar

- Superpipelined processors have longer instruction latency than the SS processors which can degrade performance in the presence of true dependencies

- Superscalar processors are more susceptible to resource conflicts – but we can fix this with hardware!
Instruction vs Machine Parallelism

- **Instruction-level parallelism (ILP)** of a program – 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 true (data) dependencies and procedural (control) dependencies in relation to the number of other instructions

- **Data-level parallelism (DLP)**

- **Machine parallelism** of a processor – 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, need both ILP and machine parallelism
Multiple-Issue Processor Styles

- **Static multiple-issue processors (aka VLIW)**
  - Decisions on which instructions to execute simultaneously are being made statically (at compile time by the compiler)
  - E.g., Intel Itanium and Itanium 2 for the IA-64 ISA – EPIC (Explicit Parallel Instruction Computer)

- **Dynamic multiple-issue processors (aka superscalar)**
  - Decisions on which instructions to execute simultaneously are being made dynamically (at run time by the hardware)
  - E.g., IBM Power 2, Pentium 4, MIPS R10K, HP PA 8500
Multiple-Issue Datapath Responsibilities

- Must handle, with a combination of hardware and software fixes, the fundamental limitations of
  - Storage (data) dependencies – aka data hazards
    - Limitation more severe in a SS/VLIW processor due to (usually) low ILP
  - Procedural dependencies – aka control hazards
    - Ditto, but even more severe
    - Use dynamic branch prediction to help resolve the ILP issue
  - Resource conflicts – aka structural hazards
    - A SS/VLIW processor has a much larger number of potential resource conflicts
    - Functional units may have to arbitrate for result buses and register-file write ports
    - Resource conflicts can be eliminated by duplicating the resource or by pipelining the resource
Instruction Issue and Completion Policies

- **Instruction-issue** – initiate execution
  - Instruction lookahead capability – fetch, decode and issue instructions beyond the current instruction

- **Instruction-completion** – complete execution
  - Processor lookahead capability – complete issued instructions beyond the current instruction

- **Instruction-commit** – write back results to the RegFile or D$ (i.e., change the machine state)

In-order issue with in-order completion

In-order issue with out-of-order completion

Out-of-order issue with out-of-order completion

Out-of-order issue with out-of-order completion and in-order commit
In-Order Issue with In-Order Completion

- Simplest policy is to issue instructions in exact program order and to complete them in the same order they were fetched (i.e., in program order)

- Example:
  - Assume a pipelined processor that can fetch and decode two instructions per cycle, that has three functional units (a single cycle adder, a single cycle shifter, and a two cycle multiplier), and that can complete (and write back) two results per cycle
  - And an instruction sequence with the following characteristics

\[
\begin{align*}
I1 & \text{ needs two execute cycles (a multiply)} \\
I2 & \\
I3 & \\
I4 & \text{ needs the same function unit as I3} \\
I5 & \text{ needs data value produced by I4} \\
I6 & \text{ needs the same function unit as I5}
\end{align*}
\]
In-Order Issue, In-Order Completion Example

- I1 - two execute cycles
- I2
- I3
- I4 - same function unit as I3
- I5 - data value produced by I4
- I6 - same function unit as I5

In parallel can:
- Fetch/decode 2
- Commit 2
In-Order Issue, In-Order Completion Example

I1 - two execute cycles
I2
I3
I4 - same function unit as I3
I5 - data value produced by I4
I6 - same function unit as I5

In parallel can fetch/decode 2
Commit 2

8 cycles in total

need forwarding hardware
In-Order Issue with Out-of-Order Completion

- With out-of-order completion, a later instruction may complete before a previous instruction
  - Out-of-order completion is used in single-issue pipelined processors to improve the performance of long-latency operations such as divide

- When using out-of-order completion instruction issue is stalled when there is a resource conflict (e.g., for a functional unit) or when the instructions ready to issue need a result that has not yet been computed
IOI-OOC Example

I1 - two execute cycles
I2
I3
I4 - same function unit as I3
I5 - data value produced by I4
I6 - same function unit as I5
IOI-OOC Example

I1 - two execute cycles
I2
I3
I4 - same function unit as I3
I5 - data value produced by I4
I6 - same function unit as I5

7 cycles in total
Handling Output Dependencies

- There is one more situation that stalls instruction issuing with IOI-OOC, assume
  - I1 – writes to R3
  - I2 – writes to R3
  - I5 – reads R3

  - If the I1 write occurs after the I2 write, then I5 reads an incorrect value for R3
  - I2 has an output dependency on I1 – write before write
    - The issuing of I2 would have to be stalled if its result might later be overwritten by an previous instruction (i.e., I1) that takes longer to complete – the stall happens before instruction issue

- While IOI-OOC yields higher performance, it requires more dependency checking hardware
  - Dependency checking needed to resolve both read before write and write before write
Out-of-Order Issue with Out-of-Order Completion

- With in-order issue the processor stops decoding instructions whenever a decoded instruction has a resource conflict or a data dependency on an issued, but uncompleted instruction
  - The processor is not able to *look beyond* the conflicted instruction even though more downstream instructions might have no conflicts and thus be issueable

- Fetch and decode instructions *beyond* the conflicted one, store them in an instruction buffer (as long as there’s room), and flag those instructions in the buffer that don’t have resource conflicts or data dependencies

- Flagged instructions are then issued from the buffer without regard to their program order
**OOI-OOC Example**

- **I1**: two execute cycles
- **I2**
- **I3**
- **I4**: same function unit as I3
- **I5**: data value produced by I4
- **I6**: same function unit as I5

6 cycles in total
Antidependencies

- With OOI also have to deal with data antidependencies – when a later instruction (that completes earlier) produces a data value that destroys a data value used as a source in an earlier instruction (that issues later)

\[
\begin{align*}
\text{R3} & : = \text{R3} \ast \text{R5} & \text{True data dependency} \\
\text{R4} & : = \text{R3} + 1 & \text{Output dependency} \\
\text{R3} & : = \text{R5} + 1 & \text{Antidependency}
\end{align*}
\]

- 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)
Dependencies Review

- Each of the three data dependencies
  - True data dependencies (read before write)
  - Antidependencies (write before read)
  - Output dependencies (write before write)

manifests itself through the use of registers (or other storage locations)

- True dependencies represent the flow of data and information through a program

- Anti- and output dependencies arise because the limited number of registers mean that programmers reuse registers for different computations

- When instructions are issued out-of-order, the correspondence between registers and values breaks down and the values conflict for registers
Storage Conflicts and Register Renaming

- Storage conflicts can be reduced (or eliminated) by increasing or duplicating the troublesome resource
  - Provide additional registers that are used to reestablish the correspondence between registers and values
    - Allocated dynamically by the hardware in SS processors

- **Register renaming** – the processor renames the original register identifier in the instruction to a new register (one not in the visible register set)

\[
\begin{align*}
R3 & := R3 \times R5 \\
R4 & := R3 + 1 \\
R3 & := R5 + 1 \\
R3b & := R3a \times R5a \\
R4a & := R3b + 1 \\
R3c & := R5a + 1
\end{align*}
\]

- The hardware that does renaming assigns a “replacement” register from a pool of free registers and releases it back to the pool when its value is superseded and there are no outstanding references to it
Next Lecture and Reminders

- Next lecture
  - Introduction to Multiple Issue

- Reminders
  - Lab1 part 1 due in one week March 7th
  - Lab1 part 2 due in one week March 28th