Announcements

UG4 Honours project selection:
Talk to Vijay or Boris if interested in computer architecture projects
Last time: Tomasulo’s Algorithm

- Dynamic O-O-O execution
- Tags (RS #’s) used to name flow dependencies
- 5 reservation stations
- 6 load buffers
- Issue instructions to reservation stations, load buffers and store buffers
- Instructions wait in reservation stations or store buffers until all their operands are collected
- Functional units broadcast result and tag on the Common Data Bus (CDB) for all reservation stations, store buffers and FP register file

Reservation stations associated with functional units: simplifies scheduling & management of structural hazards
Last time: Summary of Tomasulo’s

Advantages

- Register renaming:
  - No need to wait on WAR and WAW (not true dependencies)
  - Can have many more reservation stations than registers
- Parallel release of all dependent instructions as soon as the earlier instructions completes
  - Common Data Bus (CDB) is a forward mechanism

Limitations

- Branches stall execution of later instructions until branch is resolved
  - This effectively limits reorder window to the current basic block (4-6 insts)
- Extending Tomasulo’s beyond just floating point operations introduces the risk of imprecise exceptions
Multiple-Issue Processors: Motivation

- Ideal processor: CPI of 1
  - no hazards, 1-cycle memory latency

- Realistic processor: CPI ~1
  - Dynamic scheduling – avoids WAR & WAW dependencies
  - Branch prediction – avoids control flow dependencies
  - Caches – minimize AMAT

- Question: can we do better than that???
Multiple-Issue Processors

- **Answer:** Yes!
  - start more than one instruction in the same clock cycle
  - CPI < 1 (or IPC > 1, Instructions per Cycle)

- **Two approaches:**
  - Superscalar: instructions are chosen dynamically by the hardware
  - VLIW (Very Long Instruction Word): instructions are chosen statically by the compiler (and assembled in a single long “instruction”)
Superscalar Processors

- Hardware attempts to issue up to \( n \) instructions on every cycle, where \( n \) is the **issue width** of the processor and the processor is said to have \( n \) **issue slots** and to be a \( n \)-**wide** processor.
- Instructions issued must respect data dependences.
- In some cycles not all issue slots can be used.
- Extra hardware is needed to detect more combinations of dependences and hazards and to provide more bypasses.
- Branches?
  - With branch prediction, we can predict branches and fetch instructions.
  - Can we execute such predicted instructions?
Speculative Execution

- **Speculative execution** – *execute* control-dependent instructions even when we are not sure if they should be executed

- Hardware undo, in case of a misprediction
  - Software recovery too costly, performance-wise

- Key Idea: Execute out-of-order but *commit* in order
  - Commit: the results and side-effects (e.g., flags, exceptions) of an instruction are made visible to the rest of the system

- Tomasulo + multi-issue + speculation
  - Foundation for today’s high-performance processors
Extending Tomasulo to Support Speculation

- Approach: buffer result until instruction ready to commit (i.e., known to be non-speculative)
  - Use buffered result for forwarding to dependent instructions
  - Discard buffered result if the instruction is on a mis-speculated execution path
  - At commit, write buffered result to register or memory

- Decouples execution (potentially speculative) from update of *architecturally-visible state* (non-speculative)
  - Architecturally-visible state: registers (R0-Rn, F0-Fn, memory, etc.)
Enabling Speculation with the Reorder Buffer

New structure: Reorder Buffer (ROB)

- Holds completed results until commit time
- Organized as a queue ordered by program (i.e., fetch) order
- Takes over the role of the reservation stations for tracking dependencies and bypassing values
  - Accessed by dependent instructions for forwarding of completed, but not-yet-committed, results
  - Reservation stations still needed to hold issued instructions until they begin execution
- Flushed once mis-speculation is discovered (mispredicted branch commits)
- Enables precise exceptions
  - Exception state recorded in ROB
  - Flushed if exception occurred on a mis-predicted path
Precise Exceptions

- Precise Exceptions require that the **architecturally-visible state** is consistent with sequential (one instruction at a time) execution
  - Architecturally-visible state: registers (R0-Rn, F0-Fn) & memory
  - Implication: all instructions before the excepting instruction are committed and those after it can be restarted from scratch (i.e., have not modified architectural state).

- **Speculation without support for precise exceptions can have nasty consequences**
  - E.g., I/O on a misspeculated path
  - E.g., program terminated after accessing memory it doesn’t have permissions to access, but does so on a misspeculated path

- **Other benefits of precise exceptions:**
  - Software debugging, simple exception recovery, easy context switching
Enabling Speculation with the Reorder Buffer

- Instructions are fetched in order from the Instruction Cache
- Instructions are executed out-of-order (with Tomasulo’s)
- Instructions are committed in order (with the ROB)
  - Enable precise exceptions and speculative execution
Tomasulo with Hardware Speculation

- **Issue:**
  - Get instruction from queue
  - Issue if an RS is free and an ROB entry is also free
  - Stall if no RS or no free ROB entry
  - Instructions now tagged with ROB entry number, not RS.id

- **Execute:**
  - Same as before: monitor CDB and start instruction when operands are available

- **Write Result:**
  - CDB broadcasts result with ROB identifier
  - ROB captures result to commit later
  - Store operations also saved in the ROB until store data is available and store instruction is committed

- **Commit:**
  - If branch, check prediction and squash following instructions if incorrect
  - If store, send data and address to memory unit and perform write action
  - Else, update register with new value and release ROB entry
In-order vs Out-of-order Superscalars Compared

**Intel SandyBridge**
- 4-wide decode/issue
- 6-wide execute

**Cavium MIPS core**
- 2-wide in-order

*Source: AnandTech*
State-of-the-Art in Out-of-Order Superscalars

Intel Haswell (circa 2014-15):

- Large Instruction Window: 192 entries
- Deep Load & Store buffers: 72 load, 42 store
- Deeper OoO scheduling window: 60 entries
- Execution ports: 8 (more execution resources for store address calculation, branches and integer processing).
VLIW Processors

- Compiler chooses and “packs” independent instructions into a single long “instruction” word or “bundle”
- Compiler responsible for avoiding hazards
  - Keeps hardware simple
  - Compiler’s schedule must be conservative to guarantee safety
- Not all portions of the long instruction word will be used in every cycle
  - Compiler must be able to expose a lot of parallelism in the schedule to attain good performance
- Example:

<table>
<thead>
<tr>
<th>MEM op 1</th>
<th>MEM op 2</th>
<th>FP op 1</th>
<th>FP op 2</th>
<th>INT op</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld f18,-32(r1)</td>
<td>ld f22,-40(r1)</td>
<td>addd f4,f0,f2</td>
<td>addd f8,f6,f2</td>
<td></td>
</tr>
</tbody>
</table>
VLIW Processors (con’d)

- Key challenge for VLIW processors:
  - find control-independent work to fill each word
  - Cover data-dependent stalls (e.g., F.DIV immediately followed by a use of the result) with independent instructions

- Solutions:
  - Get rid of control flow
    - Predication
    - Loop unrolling
  - Move code around to maximize scheduling opportunities and minimize stalls
Predication

- Idea: compiler converts control flow dependencies into data dependencies
  - In effect, branches are replaced with conditional execution
    - In practice, with conditional commit

- How?
  - Instructions on both paths of the branch are executed
    - Branch instructions are completely eliminated!
  - Each instruction has a predicate bit that is set based on the computation of the predicate
  - Only instructions with TRUE predicates are committed
Predication

Normal code (branch-based control flow)

```java
if (cond)
    b = 0;
else
    b = 1;
x = b + 1;
```

Predicated code

```
p1 = (cond)
branch p1, TARGET

mov b, 1
jmp JOIN

TARGET:
mov b, 0

JOIN:
add x, b, 1
```

```
p1 = (cond)

(!p1) mov b, 1

(p1) mov b, 0

add x, b, 1
```
Predication Pros & Cons

- **Advantages:**
  - avoid pipeline bubbles whenever there is a branch
  - compiler can hide data hazards by scheduling independent instructions

- **Disadvantages:**
  - More instructions executed (bad for power)
  - Potentially longer critical path than if only the correct side of the branch was executed
Loop Unrolling

- **Idea:** replicate loop body multiple times within one iteration of the loop

- **Advantages:**
  - Reduces loop maintenance overhead (induction variable update, condition code computation, branch)
  - Enlarges **basic block** size, thus maximizing scheduling opportunities
    - Basic block: a sequence of instructions with exactly one entry point and exactly one exit point

- **Drawback:**
  - Need extra code to detect & deal with cases when unroll factor not multiple of iteration count
    - In practice, this is a small price to pay
Safety and legality of code motion

- Two characteristics of code motion:
  - Safety: whether or not spurious exceptions may occur
  - Legality: whether or not the result is guaranteed correct

(a) safe and legal
Safety and legality of code motion

- Two characteristics of code motion:
  - Safety: whether or not spurious exceptions may occur
  - Legality: whether or not the result is guaranteed correct

![Diagram showing examples of safe, legal, unsafe, and unsafe and illegal code motion]
Multiflow and Trace Scheduling

Multiflow Computer: pioneered the VLIW design style

- Available in VLIW widths of 7, 14, and 28 ops/inst
  - Max width of 28 required a 1024-bit word
- Introduced powerful compilation techniques, particularly trace scheduling
  - Trace scheduling fused multiple basic blocks on “hot” code paths into regions called traces
  - These traces created rich opportunities for code motion
Modern-day case study: TriMedia TM 1000

- 5 issue slots, 28 functional units, 128 registers
- Each inst within a word can be predicated
  - Any one of 128 regs can serve as the predicate (LSB is used)
- No dynamic hazard detection (compiler’s job)
  - Except on cache misses, which will lock the pipeline
- Non-excepting loads enable load speculation (no virtual mem)
Superscalar vs. VLIW Processors

**Super Scalars**

+ Able to handle dynamic events like cache misses, unpredictable memory dependences, branches, etc.
+ Can exploit old binaries from previous implementations

- Complexity limits issue width to 4-8

**VLIW**

+ Much simpler hardware implementation
+ Implementations can have wider issue than superscalars

- Require more complex compiler support
- Cannot use old binaries when pipeline implementation changes
- Code size increases because of empty issue slots
What are the practical Limitations to ILP?

- Limitations on max issue width and instruction window size
- Effects of realistic branch prediction
- The effect of limited numbers of rename registers
- Memory aliasing
- Variable memory latencies (because of caches)

Available ILP in a perfect processor, with none of the above constraints.
- 6 SPEC92 benchmarks
- the first 3 are Int, the last 3 are FP

These levels of ILP are impossible to achieve in practice due to limitations above

H&P 4th ed, fig 3.1 (p.157)
Effect of Instruction Window (i.e., ROB)

H&P 4th ed, fig 3.2 (p.159)
Effect of Branch prediction (2K ROB)

H&P 4th ed, fig 3.3 (p.160)
Limits to Multiple-issue

- Fundamental limits to ILP in most programs:
  - Need $N$ independent instructions to keep a $W$-issue processor busy, where $N = W \times \text{pipeline depth}$
  - Data and control dependences significantly limit amount of ILP

- Complexity of the hardware based on issue width:
  - Number of functional units increases linearly $\rightarrow$ OK
  - Number of ports for register file increases linearly $\rightarrow$ bad
  - Number of ports for memory increases linearly $\rightarrow$ bad
  - Number of dependence tests increases quadratically $\rightarrow$ bad
  - Bypass/forwarding logic and wires increases quadratically $\rightarrow$ bad

These two tend to ultimately limit the width of practical dynamically-scheduled superscalars
Summary of Factors Limiting ILP in Real Programs

- Compared with an ideal processor
  - Limited instruction window
  - Finite number of registers (introduces WAW and WAR stalls)
  - Imperfect branch prediction (pipeline flushes)
  - Limited issue width
  - Instruction fetch delays (cache misses, across-block fetch)
  - Imperfect memory disambiguation (conservative RAW stalls)

- Implications for future performance growth?
  - Single processor has inherent limits
  - To use future silicon area, need to go to multiple processors
Acknowledgement

These slides contain material developed by Onur Mutlu (CMU) for his ECE 447 course.