18-447
Computer Architecture
Lecture 10: Control Dependence Handling

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2013, 2/8/2013
Reminder: Homework 2

- Homework 2 out
  - Due February 11 (Monday!)
  - LC-3b microcode
  - ISA concepts, ISA vs. microarchitecture, microcoded machines

- Remember: Homework 1 solutions were out
Reminder: Lab Assignment 2

- Lab Assignment 2
  - Due Feb 15 (next Friday!)
  - Single-cycle MIPS implementation in Verilog
  - All labs are individual assignments
  - No collaboration; please respect the honor code
  - Do not forget the extra credit portion!
Readings for Next Few Lectures

- P&H Chapter 4.9-4.11

  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts

- Recommended:
Also, ...

Tuesday, February 12, 4:30-6:30pm, Hamerschlag 1107
Dr. Richard E. Kessler, Cavium Fellow and Principal Architect
Designing Efficient Processor Cores for Multicore Networking
Abstract:
- The design of CPUs has always required a balance of performance and efficiency in power, area, and complexity. The emergence of multicore SoCs armed with accelerators for packet processing has shifted this balance from solely single-thread performance to a combination of single-thread performance and efficient parallel processing. This shift requires a new style of core with short and deterministic pipelines, caches and memory systems optimized for low latency and high bandwidth, and an architecture that scales to 48-plus cores on a chip. This talk demonstrates how continuously emerging application demands shaped the fundamental principles behind OCTEON processor cores and supporting on-chip accelerators.
Monday: IEEE Tech Talk and CALCM Seminar

- **Monday, February 11, 4:30-6:30pm, Hamerschlag 1107**
- **Dr. Richard E. Kessler, Cavium Fellow and Principal Architect**
- **Designing Efficient Processor Cores for Multicore Networking**

**Break-out Session:**
- Cavium is building a community of university and industry partners around the 32-core OCTEON II solution, with evaluation boards in use by students and professors at several universities globally. This break-out session for students will be conducted at the conclusion of the talk above to describe the evaluation board hardware, the Cavium SDK, and various semester-long student projects appropriate for upper level undergraduates or first year masters students. Other aspects of the OCTEON program will be briefly described, including a multi-university workshop planned in May for students to present their OCTEON project and compete for the OCTEON Trophy.
Last Lecture

- Data dependence handling
Today’s Agenda

- Control dependence handling
Review: How to Handle Data Dependences

- Anti and output dependences are easier to handle
  - Dependence on name, not dependence on value/dataflow
  - Get rid of them via **renaming**: You have seen a version of this!

- Flow dependences are more interesting

- **Five fundamental ways of handling flow dependences**
  - Detect and wait until value is available in register file
  - Detect and forward/bypass data to dependent instruction
  - Detect and eliminate the dependence at the software level
    - No need for the hardware to detect dependence
  - Predict the needed value(s), execute “speculatively”, and verify
  - Do something else (fine-grained multithreading)
    - No need to detect
Questions to Ponder

- What is the role of the hardware vs. the software in data dependence handling?
  - Software based interlocking
  - Hardware based interlocking
  - Who inserts/manages the pipeline bubbles?
  - Who finds the independent instructions to fill “empty” pipeline slots?
  - What are the advantages/disadvantages of each?
Questions to Ponder

- What is the role of the hardware vs. the software in the order in which instructions are executed in the pipeline?
  - Software based instruction scheduling → static scheduling
  - Hardware based instruction scheduling → dynamic scheduling
More on Software vs. Hardware

- **Software based scheduling of instructions → static scheduling**
  - Compiler orders the instructions, hardware executes them in that order
  - Contrast this with dynamic scheduling (in which hardware will execute instructions out of the compiler-specified order)
  - How does the compiler know the latency of each instruction?

- **What information does the compiler not know that makes static scheduling difficult?**
  - Answer: Anything that is determined at run time
    - Variable-length operation latency, memory addr, branch direction

- **How can the compiler alleviate this (i.e., estimate the unknown)?**
  - Answer: Profiling
Control Dependence Handling
Control Dependence

- Question: What should the fetch PC be in the next cycle?
  - Answer: The address of the next instruction
    - All instructions are control dependent on previous ones. Why?

- If the fetched instruction is a non-control-flow instruction:
  - Next Fetch PC is the address of the next-sequential instruction
  - Easy to determine if we know the size of the fetched instruction

- If the instruction that is fetched is a control-flow instruction:
  - How do we determine the next Fetch PC?

- In fact, how do we even know whether or not the fetched instruction is a control-flow instruction?
### Branch Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Direction at fetch time</th>
<th>Number of possible next fetch addresses?</th>
<th>When is next fetch address resolved?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional</td>
<td>Unknown</td>
<td>2</td>
<td>Execution (register dependent)</td>
</tr>
<tr>
<td>Unconditional</td>
<td>Always taken</td>
<td>1</td>
<td>Decode (PC + offset)</td>
</tr>
<tr>
<td>Call</td>
<td>Always taken</td>
<td>1</td>
<td>Decode (PC + offset)</td>
</tr>
<tr>
<td>Return</td>
<td>Always taken</td>
<td>Many</td>
<td>Execution (register dependent)</td>
</tr>
<tr>
<td>Indirect</td>
<td>Always taken</td>
<td>Many</td>
<td>Execution (register dependent)</td>
</tr>
</tbody>
</table>

Different branch types can be handled differently
How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:
  - Stall the pipeline until we know the next fetch address
  - Guess the next fetch address (branch prediction)
  - Employ delayed branching (branch delay slot)
  - Do something else (fine-grained multithreading)
  - Eliminate control-flow instructions (predicated execution)
  - Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)
Stall Fetch Until Next PC is Available: Good Idea?

This is the case with non-control-flow and unconditional br instructions!
Doing Better than Stalling Fetch …

- Rather than waiting for true-dependence on PC to resolve, just guess nextPC = PC+4 to keep fetching every cycle
  Is this a good guess?
  What do you lose if you guessed incorrectly?

- ~20% of the instruction mix is control flow
  - ~50% of “forward” control flow (i.e., if-then-else) is taken
  - ~90% of “backward” control flow (i.e., loop back) is taken
    Overall, typically ~70% taken and ~30% not taken
    [Lee and Smith, 1984]

- Expect “nextPC = PC+4” ~86% of the time, but what about the remaining 14%?
Guessing $\text{NextPC} = \text{PC} + 4$

- Always predict the next sequential instruction is the next instruction to be executed
- This is a form of next fetch address prediction and branch prediction

- How can you make this more effective?

- Idea: Maximize the chances that the next sequential instruction is the next instruction to be executed
  - Software: Lay out the control flow graph such that the "likely next instruction" is on the not-taken path of a branch
  - Hardware: ??? (how can you do this in hardware...)
Guessing NextPC = PC + 4

- How else can you make this more effective?

- Idea: Get rid of control flow instructions (or minimize their occurrence)

- How?
  1. Get rid of unnecessary control flow instructions → combine predicates (predicate combining)
  2. Convert control dependences into data dependences → predicated execution
Predicate Combining (not Predicated Execution)

- Complex predicates are converted into multiple branches
  - if ((a == b) && (c < d) && (a > 5000)) { ... }
    - 3 conditional branches
- Problem: This increases the number of control dependencies
- Idea: Combine predicate operations to feed a single branch instruction instead of having one branch for each
  - Predicates stored and operated on using condition registers
  - A single branch checks the value of the combined predicate
  - Fewer branches in code $\rightarrow$ fewer mipredictions/stalls
- Possibly unnecessary work
  - If the first predicate is false, no need to compute other predicates
- Condition registers exist in IBM RS6000 and the POWER architecture
Predicated Execution

- **Idea:** Convert control dependence to data dependence

- Suppose we had a Conditional Move instruction...
  - CMOV condition, R1 ← R2
  - R1 = (condition == true) ? R2 : R1
  - Employed in most modern ISAs (x86, Alpha)

- Code example with branches vs. CMOVs
  if (a == 5) {b = 4;} else {b = 3;}

  CMPEQ condition, a, 5;
  CMOV condition, b ← 4;
  CMOV !condition, b ← 3;
Predicated Execution

- Eliminates branches $\rightarrow$ enables straight line code (i.e., larger basic blocks in code)

**Advantages**
- **Always-not-taken prediction works better** (no branches)
- **Compiler has more freedom to optimize code** (no branches)
  - control flow does not hinder inst. reordering optimizations
  - code optimizations hindered only by data dependencies

**Disadvantages**
- **Useless work: some instructions fetched/executed but discarded** (especially bad for easy-to-predict branches)
- **Requires additional ISA support**

**Can we eliminate all branches this way?**
Predicated Execution

- We will get back to this...

- Some readings (optional):
How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:
  - Stall the pipeline until we know the next fetch address
  - Guess the next fetch address (branch prediction)
  - Employ delayed branching (branch delay slot)
  - Do something else (fine-grained multithreading)
  - Eliminate control-flow instructions (predicated execution)
  - Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)
Delayed Branching (I)

- Change the semantics of a branch instruction
  - Branch after N instructions
  - Branch after N cycles

- Idea: Delay the execution of a branch. N instructions (delay slots) that come after the branch are always executed regardless of branch direction.

- Problem: How do you find instructions to fill the delay slots?
  - Branch must be independent of delay slot instructions

- Unconditional branch: Easier to find instructions to fill the delay slot
- Conditional branch: Condition computation should not depend on instructions in delay slots → difficult to fill the delay slot
Delayed Branching (II)

Normal code:

Timeline:

Delayed branch code:

Timeline:

6 cycles

5 cycles
Fancy Delayed Branching (III)

- Delayed branch with squashing
  - In SPARC
  - If the branch falls through (not taken), the delay slot instruction is not executed
  - Why could this help?

Normal code:               Delayed branch code:               Delayed branch w/ squashing:

X:  
   | A  
   | B  
   | C  
   | BC X  
   | D  
   | E  

X:  
   | A  
   | B  
   | C  
   | BC X  
   | NOP  
   | D  
   | E  

X:  
   | A  
   | B  
   | C  
   | BC X  
   | A  
   | D  
   | E
Delayed Branching (IV)

- **Advantages:**
  - Keeps the pipeline full with useful instructions in a simple way assuming
    1. Number of delay slots == number of instructions to keep the pipeline full before the branch resolves
    2. All delay slots can be filled with useful instructions

- **Disadvantages:**
  - Not easy to fill the delay slots (even with a 2-stage pipeline)
    1. Number of delay slots increases with pipeline depth, superscalar execution width
    2. Number of delay slots should be variable with variable latency operations. Why?
  - Ties ISA semantics to hardware implementation
    - SPARC, MIPS, HP-PA: 1 delay slot
    - What if pipeline implementation changes with the next design?
An Aside: Filling the Delay Slot

reordering data independent (RAW, WAW, WAR) instructions does not change program semantics

within same basic block

a new instruction added to not-taken path??

Safe?

[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:
  - Stall the pipeline until we know the next fetch address
  - Guess the next fetch address (branch prediction)
  - Employ delayed branching (branch delay slot)
  - Do something else (fine-grained multithreading)
  - Eliminate control-flow instructions (predicated execution)
  - Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)
Fine-Grained Multithreading

- **Idea**: Hardware has multiple thread contexts. Each cycle, fetch engine fetches from a different thread.
  - By the time the fetched branch/instruction resolves, there is no need to fetch another instruction from the same thread.
  - Branch/instruction resolution latency overlapped with execution of other threads’ instructions.

- No logic needed for handling control and data dependences within a thread.
  - Single thread performance suffers.
  - Extra logic for keeping thread contexts.
  - Does not overlap latency if not enough threads to cover the whole pipeline.
Fine-grained Multithreading

- **Idea:** Switch to another thread every cycle such that no two instructions from the thread are in the pipeline concurrently.

- Tolerates the control and data dependency latencies by overlapping the latency with useful work from other threads.

- Improves pipeline utilization by taking advantage of multiple threads.


Fine-grained Multithreading: History

- CDC 6600’s peripheral processing unit is fine-grained multithreaded
  - Processor executes a different I/O thread every cycle
  - An operation from the same thread is executed every 10 cycles

- Denelcor HEP (Heterogeneous Element Processor)
  - 120 threads/processor
  - available queue vs. unavailable (waiting) queue for threads
  - each thread can only have 1 instruction in the processor pipeline; each thread independent
  - to each thread, processor looks like a non-pipelined machine
  - system throughput vs. single thread performance tradeoff
Fine-grained Multithreading in HEP

- Cycle time: 100ns
- 8 stages $\rightarrow$ 800 ns to complete an instruction
  - assuming no memory access
Multithreaded Pipeline Example

- Slide from Joel Emer
Sun Niagara Multithreaded Pipeline

Fine-grained Multithreading

**Advantages**
- No need for dependency checking between instructions
  (only one instruction in pipeline from a single thread)
- No need for branch prediction logic
- Otherwise-bubble cycles used for executing useful instructions from different threads
- Improved system throughput, latency tolerance, utilization

**Disadvantages**
- Extra hardware complexity: multiple hardware contexts, thread selection logic
- Reduced single thread performance (one instruction fetched every N cycles)
- Resource contention between threads in caches and memory
- Some dependency checking logic between threads remains (load/store)
How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:
  - Stall the pipeline until we know the next fetch address
  - Guess the next fetch address (branch prediction)
  - Employ delayed branching (branch delay slot)
  - Do something else (fine-grained multithreading)
  - Eliminate control-flow instructions (predicated execution)
  - Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)
Branch Prediction: Guess the Next Instruction to Fetch

```
LD R1, MEM[R0]
ADD R2, R2, #1
BRzero 0x0001
ADD R3, R2, #1
MUL R1, R2, R3
LD R2, MEM[R2]
LD R0, MEM[R2]
```

Branch prediction:
- PC: 0x0001
- Branch prediction: Guess the Next Instruction to Fetch

12 cycles
- LD R1, MEM[R0]
- ADD R2, R2, #1
- BRzero 0x0001
- ADD R3, R2, #1
- MUL R1, R2, R3
- LD R2, MEM[R2]
- LD R0, MEM[R2]

8 cycles
Misprediction Penalty
Branch Prediction

- Processors are pipelined to increase concurrency
- How do we keep the pipeline full in the presence of branches?
  - Guess the next instruction when a branch is fetched
  - Requires guessing the direction and target of a branch

Pipeline

Fetch  Decode  Rename  Schedule  RegisterRead  Execute

B3  F  E  D  B1  A

Branch condition, TARGET

A
B1
B3
D
E
F

Target Misprediction Detected! Flush the pipeline
When a branch resolves:
- branch target (\texttt{Inst}_k) is fetched
- all instructions fetched since \texttt{inst}_h (so called “wrong-path” instructions) must be flushed
Pipeline Flush on a Misprediction

Inst_h is a branch
Performance Analysis

- correct guess $\Rightarrow$ no penalty
  $\sim$86% of the time
- incorrect guess $\Rightarrow$ 2 bubbles

Assume

- no data hazards
- 20% control flow instructions
- 70% of control flow instructions are taken
- CPI = $[ 1 + (0.20 \times 0.7) \times 2 ]$
  = $[ 1 + 0.14 \times 2 ]$
  = 1.28

Can we reduce either of the two penalty terms?
Reducing Branch Misprediction Penalty

- Resolve branch condition and target address early

CPI = \[ 1 + (0.2 \times 0.7) \times 1 \] = 1.14
Branch Prediction (Enhanced)

- **Idea:** Predict the next fetch address (to be used in the next cycle)

- Requires three things to be predicted at fetch stage:
  - Whether the fetched instruction is a branch
  - (Conditional) branch direction
  - Branch target address (if taken)

- Observation: Target address remains the same for a conditional direct branch across dynamic instances
  - **Idea:** Store the target address from previous instance and access it with the PC
  - Called **Branch Target Buffer (BTB)** or **Branch Target Address Cache**
Fetch Stage with BTB and Direction Prediction

- **Direction predictor (2-bit counters)**
  - Program Counter
  - Address of the current branch
  - Cache of Target Addresses (BTB: Branch Target Buffer)
  - Always taken CPI = \[ 1 + (0.20 \times 0.3) \times 2 \] = 1.12 (70% of branches taken)
More Sophisticated Branch Direction Prediction

Which direction earlier branches went

Global branch history

Program Counter

Address of the current branch

Direction predictor (2-bit counters)

XOR

PC + inst size

taken?

hit?

Next Fetch Address

target address

Cache of Target Addresses (BTB: Branch Target Buffer)
Simple Branch Direction Prediction Schemes

- Compile time (static)
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile based (likely direction)

- Run time (dynamic)
  - Last time prediction (single-bit)
More Sophisticated Direction Prediction

- Compile time (static)
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile based (likely direction)
  - Program analysis based (likely direction)

- Run time (dynamic)
  - Last time prediction (single-bit)
  - Two-bit counter based prediction
  - Two-level prediction (global vs. local)
  - Hybrid