Agenda for Today & Next Few Lectures

- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
  - Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
  - Issues in OoO Execution: Load-Store Handling, ...
Reminder: Readings for Next Few Lectures (I)

- P&H Chapter 4.9-4.11

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


Reminder: Readings for Next Few Lectures (II)

Reminder: Relevant Seminar Tomorrow

- **Practical Data Value Speculation for Future High-End Processors**
  - Arthur Perais, INRIA (France)
  - Thursday, Feb 5, 4:30-5:30pm, CIC Panther Hollow Room

- **Summary:**
  - Value prediction (VP) was proposed to enhance the performance of superscalar processors by breaking RAW dependencies. However, it has generally been considered too complex to implement. During this presentation, we will review different sources of additional complexity and propose solutions to address them.

Recap of Last Lecture

- Data Dependence Handling
  - Data Forwarding/Bypassing
  - In-depth Implementation
    - Register dependence analysis
    - Stalling
    - Performance analysis with and without forwarding
    - LC-3b Pipelining
  - Questions to Ponder
    - HW vs. SW handling of data dependences
    - Static versus dynamic scheduling
    - What makes compiler based instruction scheduling difficult?
    - Profiling (representative input sets needed; dynamic adaptation difficult)
    - Introduction to static instruction scheduling (e.g., fix-up code)

- Control Dependence Handling
  - Six ways of handling control dependences
  - Stalling until next fetch address is available: Bad idea
  - Predicting the next-sequential instruction as next fetch address
Tentative Plan for Friday and Monday

- I will be out of town
  - Attending the HPCA Conference

- We will finish Branch Prediction on either of these days

- Lab 2 is due Friday
  - Step 1: Get the baseline functionality correct
  - Step 2: Do the extra credit portion (it will be rewarding)

- Tentative Plan:
  - Friday: Recitation session → Come with questions on Lab 2, HW 2, lectures, concepts, etc
  - Monday: Finish branch prediction (Rachata)
Sample Papers from HPCA

  - [http://users.ece.cmu.edu/~omutlu/pub/adaptive-latency-dram_hpca15.pdf](http://users.ece.cmu.edu/~omutlu/pub/adaptive-latency-dram_hpca15.pdf)


Control Dependence Handling
Review: Control Dependence

Question: What should the fetch PC be in the next cycle?

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?
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)
Review: 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
    - Profile guided code positioning $\rightarrow$ Pettis & Hansen, PLDI 1990.
  - **Hardware:** ??? (how can you do this in hardware...)
    - Cache traces of executed instructions $\rightarrow$ Trace cache
Review: Guessing $\text{NextPC} = \text{PC} + 4$ 

- 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 $\rightarrow$ combine predicates (predicate combining)
  2. Convert control dependences into data dependences $\rightarrow$ predicated execution
Review: 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 → 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

- **Simple example:** 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**
  ```c
  if (a == 5) {b = 4;} else {b = 3;}
  ```
  ```c
  CMPEQ condition, a, 5;
  CMOV condition, b ⇔ 4;
  CMOV !condition, b ⇔ 3;
  ```
Conditional Execution in ARM

- Same as predicated execution
- Every instruction is conditionally executed
Predicated Execution

- Eliminates branches → 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:

A
B
C
BC X
D
E
F
G

Timeline:

if  ex

A
B  A
C  B
BC  C
--  BC
G  --

6 cycles

Delayed branch code:

A
C
BC X
B
D
E
F
G

Timeline:

if  ex

A
C  A
BC  C
B  BC
G  B

5 cycles
**Fancy Delayed Branching (III)**

- **Delayed branch with squashing**
  - In SPARC
  - **Semantics:** If the branch falls through (i.e., it is not taken), the delay slot instruction is not executed
  - Why could this help?

Normal code: \[ A B C D E \]

Delayed branch code: \[ A B C \text{NOP} D E \]

Delayed branch w/ squashing: \[ A B C X A D E \]
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

For correctness: add a new instruction to the not-taken path?

For correctness: add a new instruction to the taken path?

Safe?
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, no instruction is fetched 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
Idea: Switch to another thread every cycle such that no two instructions from a 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 have only 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
- No control and data dependency checking
Multithreaded Pipeline Example

Slide credit: Joel Emer
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 (PCs, register files, ...), thread selection logic
  - Reduced single thread performance (one instruction fetched every N cycles from the same thread)
  - 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
Branch Prediction: Guess the Next Instruction to Fetch

Branch prediction diagram showing the execution of instructions in a pipeline. The instructions include:

- 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]

The diagram illustrates the execution flow from PC to I-$ to DEC to RF to WB, with branch prediction impacting the pipeline stages. The instructions take 12 cycles in total, with 8 cycles for the branch prediction phase.
Misprediction Penalty

LD R0, MEM[R2]
LD R2, MEM[R2]
BR
ZERO
0x0001

Misprediction Penalty

LD R1, MEM[R0]
ADD R2, R2, #1
ADD R3, R2, #1
MUL R1, R2, R3
LD R2, MEM[R2]
LD R0, MEM[R2]

PC  I-$  DEC  RF  D-$  WB
0x0007  0x0006  0x0005  0x0004  0x0003

0x0001
ADD R2, R2, #1
BR zero 0x0001
ADD R3, R2, #1
MUL R1, R2, R3
LD R2, MEM[R2]
LD R0, MEM[R2]
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

```plaintext
A

B1  B3

D

E

F
```

Branch condition, TARGET

### What to fetch next?

- Fetch from the correct target

### Pipeline Stages

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Rename</th>
<th>Schedule</th>
<th>RegisterRead</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>B3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>E</td>
<td>D</td>
<td>B1</td>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>

Target Misprediction Detected! Flush the pipeline
Branch Prediction: Always PC+4

When a branch resolves
- branch target (Instₖ) is fetched
- all instructions fetched since instₕ (so called “wrong-path” instructions) must be flushed
Pipeline Flush on a Misprediction

Instₜ is a branch
Performance Analysis

- correct guess ⇒ no penalty  ~86% of the time
- incorrect guess ⇒ 2 bubbles

Assume
- no data dependency related stalls
- 20% control flow instructions
- 70% of control flow instructions are taken
- CPI = [ 1 + (0.20*0.7) * 2 ] = [ 1 + 0.14 * 2 ] = 1.28

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

- Resolve branch condition and target address early

Is this a good idea?

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

Always taken CPI = \[ 1 + (0.20 \times 0.3) \times 2 \] = 1.12 (70% of branches taken)
More Sophisticated Branch Direction Prediction

- Global branch history
- Program Counter
- Address of the current branch
- Which direction earlier branches went
- Direction predictor (taken?)
- PC + inst size
- hit?
- target address
- Cache of Target Addresses (BTB: Branch Target Buffer)

Next Fetch Address
Three Things to Be Predicted

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

- Third (3.) can be accomplished using a BTB
  - Remember target address computed last time branch was executed

- First (1.) can be accomplished using a BTB
  - If BTB provides a target address for the program counter, then it must be a branch
  - Or, we can store “branch metadata” bits in instruction cache/memory → partially decoded instruction stored in I-cache

- Second (2.): How do we predict the direction?
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
Static Branch Prediction (I)

- **Always not-taken**
  - Simple to implement: no need for BTB, no direction prediction
  - Low accuracy: ~30-40% (for conditional branches)
  - Remember: Compiler can layout code such that the likely path is the “not-taken” path → more effective prediction

- **Always taken**
  - No direction prediction
  - Better accuracy: ~60-70% (for conditional branches)
  - Backward branches (i.e. loop branches) are usually taken
  - Backward branch: target address lower than branch PC

- **Backward taken, forward not taken (BTFN)**
  - Predict backward (loop) branches as taken, others not-taken
Profile-based

Idea: Compiler determines likely direction for each branch using a profile run. Encodes that direction as a hint bit in the branch instruction format.

+ Per branch prediction (more accurate than schemes in previous slide) → accurate if profile is representative!
-- Requires hint bits in the branch instruction format
-- Accuracy depends on dynamic branch behavior:
  TTTTTTTTTTTTTNNNNNNNNN → 50% accuracy
  TNTNTNTNTNTNTNTNTNTNTN → 50% accuracy
-- Accuracy depends on the representativeness of profile input set
Static Branch Prediction (III)

- Program-based (or, program analysis based)
  - Idea: Use heuristics based on program analysis to determine statically-predicted direction
  - Example opcode heuristic: Predict BLEZ as NT (negative integers used as error values in many programs)
  - Example loop heuristic: Predict a branch guarding a loop execution as taken (i.e., execute the loop)
  - Pointer and FP comparisons: Predict not equal

+ Does not require profiling
-- Heuristics might be not representative or good
-- Requires compiler analysis and ISA support (ditto for other static methods)

- Ball and Larus, ”Branch prediction for free,” PLDI 1993.
  - 20% misprediction rate
Static Branch Prediction (IV)

- Programmer-based
  - Idea: Programmer provides the statically-predicted direction
  - Via *pragmas* in the programming language that qualify a branch as likely-taken versus likely-not-taken

+ Does not require profiling or program analysis
+ Programmer may know some branches and their program better than other analysis techniques

-- Requires programming language, compiler, ISA support
-- Burdens the programmer?
Pragmas

- **Idea:** Keywords that enable a programmer to convey hints to lower levels of the transformation hierarchy

  - `if (likely(x)) { ... }`
  - `if (unlikely(error)) { ... }`

- Many other hints and optimizations can be enabled with pragmas
  - E.g., whether a loop can be parallelized
  - `#pragma omp parallel`
  - **Description**
    - The `omp parallel` directive explicitly instructs the compiler to parallelize the chosen segment of code.
Static Branch Prediction

- All previous techniques can be combined
  - Profile based
  - Program based
  - Programmer based

- How would you do that?

- What is the common disadvantage of all three techniques?
  - Cannot adapt to dynamic changes in branch behavior
    - This can be mitigated by a dynamic compiler, but not at a fine granularity (and a dynamic compiler has its overheads...)
    - What is a Dynamic Compiler?
      - Remember Transmeta? Code Morphing Software?
      - Java JIT (just in time) compiler, Microsoft CLR (common lang. runtime)
Dynamic Branch Prediction

- **Idea:** Predict branches based on dynamic information (collected at run-time)

- **Advantages**
  - Prediction based on history of the execution of branches
  - It can adapt to dynamic changes in branch behavior
  - No need for static profiling: input set representativeness problem goes away

- **Disadvantages**
  -- More complex (requires additional hardware)
Last Time Predictor

- Last time predictor
  - Single bit per branch (stored in BTB)
  - Indicates which direction branch went last time it executed
    TTTTTTTTTTTNNNNNNNNNN → 90% accuracy

- Always mispredicts the last iteration and the first iteration of a loop branch
  - Accuracy for a loop with N iterations = (N-2)/N

+ Loop branches for loops with large N (number of iterations)
-- Loop branches for loops will small N (number of iterations)

  TNTNTNTNTNTNTNTNTNTNTNTNTNTN → 0% accuracy

Last-time predictor CPI = \[ 1 + (0.20 \times 0.15) \times 2 \] = 1.06 (Assuming 85% accuracy)
The 1-bit BHT (Branch History Table) entry is updated with the correct outcome after each execution of a branch.
State Machine for Last-Time Prediction

![State Machine Diagram]

- Predict not taken
- Predict taken
- Actually taken
- Actually not taken

- From predict not taken to predict taken with blue arrow:
  - Actually taken
- From predict taken to predict not taken with blue arrow:
  - Actually taken
Improving the Last Time Predictor

- Problem: A last-time predictor changes its prediction from T→NT or NT→T too quickly
  - even though the branch may be mostly taken or mostly not taken

- Solution Idea: Add hysteresis to the predictor so that prediction does not change on a single different outcome
  - Use two bits to track the history of predictions for a branch instead of a single bit
  - Can have 2 states for T or NT instead of 1 state for each

Two-Bit Counter Based Prediction

- Each branch associated with a two-bit counter
- One more bit provides hysteresis
- A strong prediction does not change with one single different outcome

- Accuracy for a loop with N iterations = \( \frac{N-1}{N} \)
  TNTNTNTNTNTNTNTNTNTN → 50% accuracy
  (assuming counter initialized to weakly taken)

+ Better prediction accuracy
  2BC predictor CPI = \( \left[ 1 + (0.20 \times 0.10) \right] \times 2 \) = 1.04  (90% accuracy)

-- More hardware cost (but counter can be part of a BTB entry)
State Machine for 2-bit Saturating Counter

- Counter using *saturating arithmetic*
  - Arithmetic with maximum and minimum values

```
pred taken 11
actually taken

pred taken 10
actually taken

pred !taken 01
actually taken

pred !taken 00
actually taken
```
Hysteresis Using a 2-bit Counter

Change prediction after 2 consecutive mistakes
Is This Good Enough?

- ~85-90% accuracy for many programs with 2-bit counter based prediction (also called bimodal prediction)

- Is this good enough?

- How big is the branch problem?
Rethinking the The Branch Problem

- Control flow instructions (branches) are frequent
  - 15-25% of all instructions

- Problem: **Next fetch address after a control-flow instruction is not determined after N cycles in a pipelined processor**
  - N cycles: (minimum) branch resolution latency

- If we are fetching W instructions per cycle (i.e., if the pipeline is W wide)
  - A branch misprediction leads to $N \times W$ wasted instruction slots
Importance of The Branch Problem

- Assume $N = 20$ (20 pipe stages), $W = 5$ (5 wide fetch)
- Assume: 1 out of 5 instructions is a branch
- Assume: Each 5 instruction-block ends with a branch

How long does it take to fetch 500 instructions?

- **100% accuracy**
  - 100 cycles (all instructions fetched on the correct path)
  - No wasted work
- **99% accuracy**
  - 100 (correct path) + 20 (wrong path) = 120 cycles
  - 20% extra instructions fetched
- **98% accuracy**
  - 100 (correct path) + 20 * 2 (wrong path) = 140 cycles
  - 40% extra instructions fetched
- **95% accuracy**
  - 100 (correct path) + 20 * 5 (wrong path) = 200 cycles
  - 100% extra instructions fetched
Can We Do Better?

- Last-time and 2BC predictors exploit “last-time” predictability

Realization 1: A branch’s outcome can be correlated with other branches’ outcomes
  - Global branch correlation

Realization 2: A branch’s outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch “last-time” it was executed)
  - Local branch correlation