Good News!

- Room change

- New lecture room from Wednesday:
  - CIC Panther Hollow Room, 4th Floor
  - Aka, the big conference room right on the left when you enter the big glass doors of the Intel Science and Technology Center after you get off the 4th floor elevators in CIC

- CIC location:
  - http://goo.gl/maps/dh4KT
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


Readings for Next Few Lectures (II)

Control Dependence Handling
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
A
B
A
C
B
BC C
--
BC
G
--
```

6 cycles

Delayed branch code:

```
A
C
BC X
```

Timeline:

```
if
A
C
A
BC C
B
BC
G
B
```

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

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

Safe?

For correctness: a new instruction added to taken path??

[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, 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.
Fine-grained Multithreading

- **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 → 800 ns to complete an instruction
  - assuming no memory access
Multithreaded Pipeline Example

Slide credit: 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

- **0x0001**: LD R1, MEM[R0]
- **0x0002**: ADD R2, R2, #1
- **0x0003**: BRzero 0x0001
- **0x0004**: ADD R3, R2, #1
- **0x0005**: MUL R1, R2, R3
- **0x0006**: LD R2, MEM[R2]
- **0x0007**: LD R0, MEM[R2]

Branch prediction:
- 12 cycles
- 8 cycles

PC: 0x0001

![Diagram](image-url)
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
0x0002
0x0003
0x0004
0x0005
0x0006
0x0007

0x0001
0x0002
0x0003
0x0004
0x0005
0x0006
0x0007
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

```
<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Rename</th>
<th>Schedule</th>
<th>Register</th>
<th>Read</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>B3</td>
<td></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>
<td></td>
</tr>
</tbody>
</table>
```

Branch condition, TARGET:

- A
- B1
- B3
- D
- E
- F
Branch Prediction: Always PC+4

When a branch resolves:
- branch target (Inst\textsubscript{k}) is fetched
- all instructions fetched since \textsubscript{inst}h (so called “wrong-path” instructions) must be flushed
Pipeline Flush on a Misprediction

Inst_h is a branch
Performance Analysis

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

probability of penalty for
a wrong guess a wrong guess
Reducing Branch Misprediction Penalty

- Resolve branch condition and target address early

CPI = [ 1 + (0.2*0.7) * 1 ] = 1.14

[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
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)

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)

taken?

PC + inst size

hit?

target address

Cache of Target Addresses (BTB: Branch Target Buffer)

Next Fetch Address

Which direction earlier branches went

Global branch history

Program Counter

Address of the current branch

Direction predictor (2-bit counters)

taken?

PC + inst size

hit?

target address

Cache of Target Addresses (BTB: Branch Target Buffer)

Next Fetch Address
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%
  - Compiler can layout code such that the likely path is the “not-taken” path

- **Always taken**
  - No direction prediction
  - Better accuracy: ~60-70%
    - 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 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:

TTTTTTTTTTTNNNNNNNNNNN → 50% accuracy
TNTNTNTNTNTNTNTNTNTNTNT → 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)

  - 20% misprediction rate
Static Branch Prediction (III)

- 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?
Aside: 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 are common disadvantages 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...)

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 number of iterations
-- Loop branches for loops will small number of iterations
  TNTNTNTNTNTNTNTNTNTNTNTNTN  →  0% accuracy

Last-time predictor CPI = \[ 1 + (0.20 \times 0.15) \times 2 \] = 1.06  (Assuming 85% accuracy)
Implementing the Last-Time Predictor

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

predict not taken

actually not taken

actually taken

predict taken

actually taken

actually not taken
Improving the Last Time Predictor

- **Problem:** A last-time predictor changes its prediction from $T \rightarrow NT$ or $NT \rightarrow T$ 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 = (N-1)/N
  TNTNTNTNTNTNTNTNTNTNTNTN → 50% accuracy
  (assuming counter initialized to weakly taken)

+ Better prediction accuracy
  2BC predictor CPI = [ 1 + (0.20*0.10) * 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
Hysteresis Using a 2-bit Counter

Change prediction after 2 consecutive mistakes
Is This 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
  - Stalling on a branch wastes instruction processing bandwidth (i.e. reduces IPC)
    - N x W instruction slots are wasted (W: pipeline width)

- How do we keep the pipeline full after a branch?
- Problem: Need to determine the next fetch address when the branch is fetched (to avoid a pipeline bubble)
Importance of The Branch Problem

- Assume a 5-wide \textit{superscalar} pipeline with 20-cycle branch resolution latency

- How long does it take to fetch 500 instructions?
  - Assume 1 out of 5 instructions is a branch
  - 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 \times 2 (wrong path) = 140 cycles
    - 40\% extra instructions fetched
  - 95\% accuracy
    - 100 (correct path) + 20 \times 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