CS152: Computer Systems Architecture
Pipelining

Sang-Woo Jun
Winter 2023

Large amount of material adapted from MIT 6.004, “Computation Structures”, Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”, and CS 152 Slides by Isaac Scherson
Eight great ideas

- Design for Moore’s Law
- Use abstraction to simplify design
- Make the common case fast
- Performance via parallelism
- **Performance via pipelining**
- Performance via prediction
- Hierarchy of memories
- Dependability via redundancy

But before we start...
Performance Measures

❑ Two metrics when designing a system

1. Latency: The delay from when an input enters the system until its associated output is produced
2. Throughput: The rate at which inputs or outputs are processed

❑ The metric to prioritize depends on the application
  o Embedded system for airbag deployment?  Latency
  o General-purpose processor?  Throughput
Performance of Combinational Circuits

- For combinational logic
  - latency = \( t_{PD} \)
  - throughput = \( \frac{1}{t_{PD}} \)

Is this an efficient way of using hardware?

Source: MIT 6.004 2019 L12
Pipelining by adding registers to hold F and G’s output

- Now F & G can be working on input $X_{i+1}$ while H is performing computation on $X_i$
- A 2-stage pipeline!
- For input X during clock cycle j, corresponding output is emitted during clock j+2.

Assuming latencies of 15, 20, 25...

Source: MIT 6.004 2019 L12
Pipelined Circuits

F and G not doing work!
Just holding output data

<table>
<thead>
<tr>
<th></th>
<th>Latency</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unpipelined</td>
<td>45</td>
<td>1/45</td>
</tr>
<tr>
<td>2-stage pipelined</td>
<td>50 (Worse!)</td>
<td>1/25 (Better!)</td>
</tr>
</tbody>
</table>

Source: MIT 6.004 2019 L12
Pipeline conventions

_definition:
- A well-formed K-Stage Pipeline ("K-pipeline") is an acyclic circuit having exactly K registers on every path from an input to an output.
- A combinational circuit is thus a 0-stage pipeline.

Composition convention:
- Every pipeline stage, hence every K-Stage pipeline, has a register on its output (not on its input).

Clock period:
- The clock must have a period $t_{CLK}$ sufficient to cover the longest register to register propagation delay plus setup time.

$K$-pipeline latency $= K \times t_{CLK}$
$K$-pipeline throughput $= 1 / t_{CLK}$

Source: MIT 6.004 2019 L12
Ill-formed pipelines

Is the following circuit a K-stage pipeline? No

Problem:
- Some paths have different number of registers
- Values from different input sets get mixed! -> Incorrect results
  - $B(Y_{t-1}, A(X_t))$ <- Mixing values from $t$ and $t-1$

Source: MIT 6.004 2019 L12
A pipelining methodology

- **Step 1:**
  - Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.

- **Step 2:**
  - Continue to draw new lines between the terminal points across various circuit connections, ensuring that every connection crosses each line in the same direction.
  - These lines demarcate pipeline stages.

- **Step 3:**
  - Add a pipeline register at every point where a separating line crosses a connection.

**Strategy:** Try to break up high-latency elements, make each pipeline stage as low-latency as possible!

Source: MIT 6.004 2019 L12
Pipelining example

- 1-pipeline improves neither L nor T
- T improved by breaking long combinational path, allowing faster clock
- Too many stages cost L, not improving T
- Back-to-back registers are sometimes needed for well-formed pipelines

Source: MIT 6.004 2019 L12
Hierarchical pipelining

- Pipelined systems can be hierarchical
  - Replacing a slow combinational component with a k-pipe version may allow faster clock

- In the example:
  - 4-stage pipeline, T=1

Source: MIT 6.004 2019 L12
Sample pipelining problem

- Pipeline the following circuit for maximum throughput while minimizing latency.
  - Each module is labeled with its latency

What is the best latency and throughput achievable?

Source: MIT 6.004 2019 L12
Sample pipelining problem

- $t_{\text{CLK}} = 4$
- $T = \frac{1}{4}$
- $L = 4 \times 4 = 16$
When pipelines are not deterministic

- Lock-step pipelines are great when modules are deterministic
  - Good for carefully scheduled circuits like a well-optimized microprocessor

- What if the latency of F is non-deterministic?
  - At some cycles, F’s pipeline register may hold invalid value
  - Pipeline register must be tagged with a valid flag
  - How many pipeline registers should we add to G? Max possible latency?
  - What if F and G are both non-deterministic? How many registers?

If register count is wrong, results are wrong! (Results mixed between different input sets!)
FIFOs (First-In First-Out)

- Queues in hardware
  - Static size (because it’s hardware)
  - User checks whether full or empty before enqueue or dequeue
  - Enqueue/dequeue in single cycle regardless of size or occupancy
  - Does use MUX! Large FIFO has long delay
Counting cycles:
Benefits of an elastic pipeline

- Assume F and G are multi-cycle, internally pipelined modules
  - If we don’t know how many pipeline stages F or G has, how do we ensure correct results?

- Elastic pipeline allows correct results regardless of latency
  - If \( L(F) = L(G) \), enqueued data available at very next cycle (acts like single register)
  - If \( L(F) = L(G) + 1 \), FIFO acts like two pipelined registers
  - What if we made a 4-element FIFO, but \( L(F) = L(G) + 4 \)?
    - G will block! Results will still be correct!
    - ... Just slower! How slow?

L <- Latency in cycles
Measuring pipeline performance

- Latency of F is 3, Latency of G is 1, and we have a 2-element FIFO
  - What would be the performance of this pipeline?

- One pipeline “bubble” every four cycles
  - Duty cycle of $\frac{3}{4}$!
Aside: Little’s law

- \( L = \lambda W \)
  - \( L \): Number of requests in the system
  - \( \lambda \): Throughput
  - \( W \): Latency
  - Imagine a DMV office! \( L \): Number of booths. (Not number of chairs in the room)

In our pipeline example
- \( L = 3 \) (limited by pipeline depth of \( G \))
- \( W = 4 \) (limited by pipeline depth of \( F \))
- As a result: \( \lambda = \frac{3}{4} \!

How do we improve performance?
- Larger FIFO, or
- Replicate \( G \)! (round-robin use of \( G_1 \) and \( G_2 \))
CS152: Computer Systems Architecture
Processor Microarchitecture – Pipelining

Sang-Woo Jun
Winter 2023

Large amount of material adapted from MIT 6.004, “Computation Structures”, Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”, and CS 152 Slides by Isaac Scherson
Course outline

- Part 1: The Hardware-Software Interface
  - What makes a ‘good’ processor?
  - Assembly programming and conventions

- Part 2: Recap of digital design
  - Combinational and sequential circuits
  - How their restrictions influence processor design

- Part 3: Computer Architecture
  - Simple and pipelined processors
  - Computer Arithmetic
  - Caches and the memory hierarchy

- Part 4: Computer Systems
  - Operating systems, Virtual memory
How to build a computing machine?

- Pretend the computers we know and love have never existed
- We want to build an automatic computing machine to solve mathematical problems
- Starting from (almost) scratch, where you have transistors and integrated circuits but no existing microarchitecture
  - No PC, no register files, no ALU
- How would you do it? Would it look similar to what we have now?
Aside: Dataflow architecture

- Instead of traversing over instructions to execute, all instructions are independent, and are each executed whenever operands are ready
  - Programs are represented as graphs (with dependency information)

Did not achieve market success, (why?) but the ideas are now everywhere e.g., Out-of-Order microarchitecture
The von Neumann Model

- Almost all modern computers are based on the von Neumann model
  - John von Neumann, 1945

- Components
  - Main memory, where both data and programs are held
  - Processing unit, which has a program counter and ALU
  - Storage and I/O to communicate with the outside world

Key idea!
Key Idea: Stored-Program Computer

- Very early computers were programmed by manually adjusting switches and knobs of the individual programming elements
  - (e.g., ENIAC, 1945)
- von Neumann Machines instead had a general-purpose CPU, which loaded its instructions also from memory
  - Express a program as a sequence of coded instructions, which the CPU fetches, interprets, and executes
  - “Treating programs as data”

Similar in concept to a universal Turing machine (1936)
von Neumann and Turing machine

Turing machine is a mathematical model of computing machines
  - Proven to be able to compute any mechanically computable functions
  - Anything an algorithm can compute, it can compute

Components include
  - An infinite tape (like memory) and a header which can read/write a location
  - A state transition diagram (like program) and a current location (like pc)
    - State transition done according to current value in tape

Only natural that computer designs gravitate towards provably universal models

Source: Manolis Kamvysselis
Stored program computer, now what?

- Once we decide on the stored program computer paradigm
  - With program counter (PC) pointing to encoded programs in memory

- Then it becomes an issue of deciding the programming abstraction
  - Instruction set architecture, which we talked about

- Then, it becomes an issue of executing it quickly and efficiently
  - Microarchitecture! – Improving performance/efficiency/etc while maintaining ISA abstraction
  - Which is the core of this class, starting now
The classic RISC pipeline

- Many early RISC processors had very similar structure
  - MIPS, SPARC, etc...
  - Major criticism of MIPS is that it is too optimized for this 5-stage pipeline
- RISC-V is also typically taught using this structure as well

Why these 5 stages? Why not 4 or 6?
The classic RISC pipeline

- Fetch: Request instruction fetch from memory
- Decode: Instruction decode & register read
- Execute: Execute operation or calculate address
- Memory: Request memory read or write
- Writeback: Write result (either from execute or memory) back to register
Major components of a microprocessor

- **Program Binary**
- **Working data**
- **Register file**
  - x0
  - x1
  - x2
  - x3
  - x4
  - x5
  - x31
- **Program Counter**
- **ALU**
- **Main memory**

32-bit
A high-level view of computer architecture

Will deal with caches in detail later!
Designing a microprocessor

- Many, many constraints processors are optimize for, but for now:

- Constraint 1: Circuit timing
  - Processors are complex! How do we organize the pipeline to process instructions as fast as possible?

- Constraint 2: Memory access latency
  - Register files can be accessed as a combinational circuit, but it is small
  - All other memory have high latency, and must be accessed in separate request/response
    - Memory can have high throughput, but also high latency

Memory will be covered in detail later!
The most basic microarchitecture

- Because memory is not combinational, our RISC ISA requires at least three disjoint stages to handle
  - Instruction fetch
  - Instruction receive, decode, execute (ALU), register file access, memory request
  - If mem read, write read to register file

- Three stages can be implemented as a Finite State Machine (FSM)

Will this processor be fast? Why or why not?
Limitations of our simple microarchitecture

- Stage two is disproportionately long
  - Very long critical path, which limits the clock speed of the whole processor
  - Stages are “not balanced”
- Note: we have not pipelined things yet!
Limitations of our simple microarchitecture

- Let’s call our stages Fetch (“F”), Execute (“E”), and Writeback (“W”)
- Speed of our simple microarchitecture, assuming:
  - Clock-synchronous circuits, single-cycle memory
- Lots of time not spent doing useful work!
  - Can pipelining help with performance?

![Diagram showing the stages of instruction execution: Fetch (F), Execute (E), Writeback (W) for two instructions in sequence.]
Pipelined processor introduction

- Attempt to pipeline our processor using pipeline registers/FIFOs

- Much better latency and throughput!
  - Average CPI reduced from 3 to 1!
  - Still lots of time spent not doing work. Can we do better?

* We will see soon why pipelining a processor isn’t this simple

Note we need a memory interface with two concurrent interfaces now! (For fetch and execute)
Remember instruction and data caches!
Building a balanced pipeline

- Must reduce the critical path of Execute
- Writing ALU results to register file can be moved to “Writeback”
  - Most circuitry already exists in writeback stage
  - No instruction uses memory load and ALU at the same time
    - RISC!
Building a balanced pipeline

- Divide execute into multiple stages
  - “Decode”
    - Extract bit-encoded values from instruction word
    - Read register file
  - “Execute”
    - Perform ALU operations
  - “Memory”
    - Request memory read/write

- No single critical path which reads and writes to register file in one cycle

Results in a small number of stages with relatively good balance!
Ideally balanced pipeline performance

- Clock cycle: 1/5 of total latency
- Circuits in all stages are always busy with useful work
Aside: Real-world processors have wide range of pipeline stages

<table>
<thead>
<tr>
<th>Name</th>
<th>Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVR/PIC microcontrollers</td>
<td>2</td>
</tr>
<tr>
<td>ARM Cortex-M0</td>
<td>3</td>
</tr>
<tr>
<td>Apple A9 (Based on ARMv8)</td>
<td>16</td>
</tr>
<tr>
<td>Original Intel Pentium</td>
<td>5</td>
</tr>
<tr>
<td>Intel Pentium 4</td>
<td>30+</td>
</tr>
<tr>
<td>Intel Core (i3,i5,i7,...)</td>
<td>14+</td>
</tr>
<tr>
<td>RISC-V Rocket</td>
<td>6</td>
</tr>
</tbody>
</table>

Designs change based on requirements!
Will our pipeline operate correctly?
A problematic example

- What should be stored in data+8? 3, right?

```assembly
la t0 data
lw s0, 0(t0)
lw s1, 4(t0)
add s2, s0, s1
sw s2, 8(t0)
data:
> .word 1 2
```

- Assuming zero-initialized register file, our pipeline will write zero
  
  Why? “Hazards”
CS152: Computer Systems Architecture
Achieving Correct Pipelining

Sang-Woo Jun
Winter 2023

Large amount of material adapted from MIT 6.004, “Computation Structures”, Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”, and CS 152 Slides by Isaac Scherson
A problematic example

- What should be stored in data+8? 3, right?

- Assuming zero-initialized register file, our pipeline will write zero

  Why? “Hazards”
Hazard #1: Read-After-Write (RAW) Data hazard

- When an instruction depends on a register updated by a previous instruction’s execution results
  - e.g.,
    - i1: add s0, s1, s2
    - i2: add s3, s0, s4

```
i1: add s0, s1, s2
i2: add s3, s0, s4
```
Hazard #1: Read-After Write (RAW) Hazard

i1: addi s0, zero, 1
i2: addi s1, s0, 0
s0 should be 1, s1 should be 1

Cycle 1
s0 = 0

Cycle 2
s0 = 0

Cycle 3
s0 = 0

Cycle 4
s0 = 0

Cycle 5
s0 = 0

Cycle 6
s0 = 1
Solution #1: Stalling

- The processor can choose to stall decoding when RAW hazard detected

- i1: add s0, s1, s2
- i2: add s3, s0, s4

Cycle 1: i1 reads s1, s2
Cycle 2: i1 writing s0
Cycle 5: i2 not decoded
Cycle 6: i2 reads s0

Pipeline “bubbles”

Sacrifices too much performance!

Little’s law: $L = \lambda W \rightarrow 2 = \lambda * 5$
### Solution #1: Stalling

The diagram illustrates the stalling technique in a pipeline with stages: Fetch, Decode, Execute, Memory, and Writeback.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle 1</td>
<td><code>addi s0, zero, 1</code></td>
<td><code>s0 = 0</code></td>
</tr>
<tr>
<td>Cycle 2</td>
<td><code>addi s1, s0, 0</code></td>
<td><code>s0 = 0</code></td>
</tr>
<tr>
<td>Cycle 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>i2 stalled until s0 is applied</td>
</tr>
<tr>
<td>Cycle 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Cycle 3**: i2 stalled until s0 is applied.
- **Cycle 7**: i2 reads correct s0.

**Sacrifices too much performance!**
Solution #2: Forwarding (aka Bypassing)

- Forward execution results to input of decode stage
  - New values are used if write index and a read index is the same

```
i1: add s0, s1, s2
i2: add s3, s0, s4
```

```
Cycle 1
- i1 reads s1, s2
- i1 calculates s0
- But! Uses new s0 forwarded from execute
```

```
Cycle 2
- i2 reads s0, s4
```

```
Cycle 3
- No pipeline stalls!
```
Solution #2: Forwarding details

- May still require stalls for a deeper pipeline microarchitecture
  - If execute took many cycles?
- Adds combinational path from execute to decode
  - But does not imbalance pipeline very much!

![Diagram of pipeline stages: Fetch → Decode → Execute → Memory → Writeback.](image)

Question: How does hardware detect hazards?
Solution #2: Forwarding

i1: addi s0, zero, 1
i2: addi s1, s0, 0

Cycle 1
s0 = 0
Cycle 2
s0 = 0
Cycle 3
s0 = 0
Cycle 4
s0 = 0
Cycle 5
s0 = 0
Cycle 6
s0 = 1

Forwarding is possible in this situation because the answer (s0 = 1) exists somewhere in the processor!
Datapath with Hazard Detection

Not very intuitive... We will revisit with code later
Hazard #2: Load-Use Data Hazard

- When an instruction depends on a register updated by a previous instruction
  - e.g., \( \text{i1: lw s0, 0(s2)} \), \( \text{i2: addi s1, s0, 1} \)

- Forwarding doesn’t work here, as loads only materialize at writeback
  - Only architectural choice is to stall
### Hazard #2: Load-Use Data Hazard

**Instruction:**
- i1: lw s0, 0(s2)
- i2: addi s1, s0, 1

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction</th>
<th>s0 Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>i2</td>
<td>1</td>
</tr>
</tbody>
</table>

i2 is stalled until s0 is updated.

Forwarding is not useful because the answer (s0 = 1) exists outside the chip (memory).
A non-architectural solution: Code scheduling by compiler

- Reorder code to avoid use of load result in the next instruction
- e.g., $a = b + e; c = b + f;$

Compiler does best, but not always possible!
Review: A problematic example

`la t0 data
lw s0, 0(t0)
lw s1, 4(t0)
add s2, s0, s1
sw s2, 8(t0)
data:
>   .word 1 2`

← RAW hazard
← RAW hazard
← RAW hazard
← Load-Use hazard
← RAW hazard

☐ Note: “la” is not an actual RISC-V instruction
  o Pseudo-instruction expanded to one or more instructions by assembler
  o e.g., auipc x5,0x1
      addi x5,x5,-4 # ← RAW hazard!
Other potential data hazards

Dangerous if a later instruction’s state access can happen before an earlier instruction’s access

- **Read-After-Write (RAW) Hazard**
  - Obviously dangerous! -- Writeback stage comes after decode stage
  - (Later instructions’ reads *can* happen before earlier instructions’ write)

- **Write-After-Write (WAW) Hazard**
  - No hazard for in-order processors

- **Write-After-Read (WAR) Hazard**
  - No hazard for in-order processors -- Writeback stage comes after decode stage
  - (Later instructions’ reads *cannot* happen before earlier instructions’ write)

- **Read-After-Read (RAR) Hazard?**
  - No hazard within processor
Hazard #3: Control hazard

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction
    - e.g., Still working on decode stage of branch

```plaintext
i1: beq s0, zero, elsewhere
i2: addi s1, s0, 1
```

![Pipeline diagram](image-url)
Control hazard (partial) solutions

- Branch target address can be forwarded to the fetch stage
  - Without first being written to PC
  - Still may introduce (one less, but still) bubbles

- Decode stage can be augmented with logic to calculate branch target
  - May imbalance pipeline, reducing performance
  - Doesn’t help if instruction memory takes long (cache miss, for example)
Aside: An awkward solution: Branch delay slot

- In a 5-stage pipeline with forwarding, one branch hazard bubble is injected in best scenario
- Original MIPS and SPARC processors included “branch delay slots”
  - One instruction after branch instruction was executed regardless of branch results
  - Compiler will do its best to find something to put there (if not, “nop”)
- Goal: Always fill pipeline with useful work
- Reality:
  - Difficult to always fill slot
  - Deeper pipelines meant one measly slot didn’t add much (Modern MIPS has 5+ cycles branch penalty!)

But once it’s added, it’s forever in the ISA...
One of the biggest criticisms of MIPS
Eight great ideas

- Design for Moore’s Law
- Use abstraction to simplify design
- Make the common case fast
- Performance via parallelism
- Performance via pipelining
- Performance via prediction
- Hierarchy of memories
- Dependability via redundancy
Control hazard and pipelining

- Solving control hazards is a fundamental requirement for pipelining
  - Fetch stage needs to keep fetching instructions without feedback from later stages
  - Must keep pipeline full somehow!
  - ... Can’t know what to fetch

Cycle 1  Fetch PC = 0

Cycle 2  Fetch PC = ...?  Decode PC = 0
Control hazard (partial) solution

Branch prediction

- We will try to predict whether branch is taken or not
  - If prediction is correct, great!
  - If not, we somehow do not apply the effects of mis-predicted instructions
    - (Effectively same performance penalty as stalling in this case)
  - Very important to have mispredict detection before any state change!
    - Difficult to revert things like register writes, memory I/O

- Simplest branch predictor: Predict not taken
  - Fetch stage will keep fetching pc <= pc + 4 until someone tells it not to
Predict not taken example

Fetch: addi, addi, beq, sw, ret
Decode: addi, addi, beq, sw, ret
Execute: addi, addi, beq, sw, ret
Memory: addi, beq, sw, ret
Writeback: addi, beq, sw, ret

Mispredict detected!

Pipeline bubbles

Fetch correct branch

No state update before Execute stage detects misprediction (Fetch and Decode stages don’t write to register)
How to handle mis-predictions?

- Implementations vary, each with pros and cons
  - Sometimes, execute sends a combinational signal to all previous stages, turning all instructions into a “nop”

- A simple method is “epoch-based”
  - All fetched instructions belong to an “epoch”, represented with a number
  - Instructions are tagged with their epoch as they move through the pipeline
  - In the case of mis-predict detection, global epoch is increased, and future instructions from previous epochs are ignored
Predict not taken example with epochs

Epoch = 0

Fetch: 
- addi t1, zero, 3
- addi t2, zero, 3
- beq t1, t2, skip
- sw t3, 0(t0)
- ret

Decode: 
- addi (0)
- addi (0)
- beq (0)
- sw t3 (0)
- ret (0)
- sw t2 (1)
- ret (1)

Execute: 
- addi (0)
- addi (0)
- beq (0)
- sw t3 (0)
- ret (0)
- sw t2 (1)
- ret (1)

Memory: 
- addi (0)
- addi (0)
- beq (0)
- sw t3 (0)
- ret (0)
- sw t2 (1)
- ret (1)

Writeback: 
- addi (0)
- addi (0)
- beq (0)
- sw t3 (0)
- ret (0)
- sw t2 (1)
- ret (1)

Mispredict detected!
Some classes of branch predictors

❑ Static branch prediction
  o Based on typical branch behavior
  o Example: loop and if-statement branches
    • Predict backward branches taken
    • Predict forward branches not taken

❑ Dynamic branch prediction
  o Hardware measures actual branch behavior
    • e.g., record recent history (1-bit “taken” or “not taken”) of each branch in a fixed size “branch history table”
  o Assume future behavior will continue the trend
    • When wrong, stall while re-fetching, and update history

Many many different methods, Lots of research, some even using neural networks!
Pipeline with branch prediction

- Branch predictor predicts what should be the next PC
  - Typically based on the current PC as input
- Dynamic branch predictors adapt to program using feedback
- If prediction is correct, great! If not, make sure mispredicted instructions don’t effect state
  - We looked at the epoch method of doing this (2 bubbles!)
Dynamic branch prediction

- Two questions about a PC address being fetched
  - Will this instruction cause a branch?
  - If so, where will it branch to?
  - Both information are needed to predict-fetch a branch

- Two architectural entities for predicting the answer to these questions
  - Branch History Table (BHT)
    - Whether this instruction is an instruction, and if it causes a branch
  - Branch Target Buffer (BTB)
    - Which address this instruction will jump to
  - (There are many variations – This is just a common example)
Dynamic branch prediction

method Word predict(Word pc) begin
    Word next_pc = pc + 4;
    Bit#(10) lsb = truncate(pc);
    if ( bht[lsb] ) next_pc = btb(lsb);
    return next_pc;
end

Why truncate PC? BHT/BTB is typically small! (2048 elements or so)
Different branches may map to same buffer element... 😊

Execute stage updates BHT and BTB with actual behavior (if it is a branch instruction)
Back to the three questions

- Is it a branch instruction?
  - Execute updates BHT if it is a branch instruction

- Is the branch taken?
  - BHT stores if the branch was taken last time

- Where does the branch go?
  - BTB stores where it went to last time

- Of course, all three are merely predictions!
Impact of branch predictors on performance

Marek Majkowski, “Branch predictor: How many "if"s are too many? Including x86 and M1 benchmarks!” The Cloudflare Blog, 2021

```c
const char *getCountry(int cc) {
    if(cc == 1) return "Al";
    if(cc == 2) return "A2";
    if(cc == 3) return "01";
    if(cc == 4) return "AD";
    if(cc == 5) return "AE";
    if(cc == 6) return "AF";
    if(cc == 7) return "AG";
    if(cc == 8) return "AI";
    ...
    if(cc == 252) return "YT";
    if(cc == 253) return "ZA";
    if(cc == 254) return "ZM";
    if(cc == 255) return "Zw";
    if(cc == 256) return "XK";
    if(cc == 257) return "T1";
    return "UNKNOWN";
}
```
Simple example: 1-bit predictor

- BHT has one-bit entries
  - Most recently taken/not taken
  - (“Last time predictor”)
  - Does this work well?

- How many mispredicts with these taken (T), not taken (N) sequences?
  - TTTTTNNNNN  TTTTNNNNN
  - TNTNTNTNTN  TNTNTNTNTN
  - for (i = 0 ... 2) {
    for (j = 0 ... 2) {
    }
  } Mispredict at j = 0 (T), j = 2 (N)
Simple example: 2-bit predictor

- BHT has two bits – Single outlier does not change future predictions
  - 00: Strongly not taken, 01: Not taken, 10: Taken, 11: Strongly taken
  - Taken branch increases number, not taken branch decreases number
  - Counter saturates! Taken after 11 -> 11, Not taken after 00 -> 00

- How many mispredicts with these taken (T), not taken (N) sequences?
  - TTTTTNNNNN
  - TNTNTNTNTN
  - for (i = 0 ... 2) {
      for (j = 0 ... 2 ) {
      }
  } Mispredict once at i = 0 && j = 0 (T), j = 2 (N),

In reality, most SPEC benchmarks record ~90% accuracy with 2-bit predictor
Branch prediction and performance

- Effectiveness of branch predictors is crucial for performance
  - Spoilers: On SPEC benchmarks, modern predictors routinely have 98+% accuracy
  - Of course, less-optimized code may have much worse behavior

- Branch-heavy software performance depends on good match between software pattern and branch prediction
  - Some high-performance software optimized for branch predictors in target hardware
  - Or, avoid branches altogether! (Branchless code)
Aside: Impact of branches

“This code] takes ~12 seconds to run. But on commenting line 15, not touching the rest, the same code takes ~33 seconds to run.”

“(running time may vary on different machines, but the proportion will stay the same).”

Source: Harshal Parekh, “Branch Prediction — Everything you need to know.”
Aside:
Impact of branches

```c
for (int i = 0 ; i < len ; i++) {
    if (nums[0][i] * nums[1][i] != 0) {
        arbitrary++;
    }
    /* Slower because it involves two branches
    if (nums[0][i] != 0 && nums[1][i] != 0) {
        arbitrary++;
    }
    */
}
```

Source: Harshal Parekh, “Branch Prediction — Everything you need to know.”
Aside: Branchless programming

for (int c = 0; c < arraySize; ++c)
    data[c] = rnd.nextInt() % 256;

// With this, the next loop runs faster
Arrays.sort(data);

// Test
long start = System.nanoTime();
long sum = 0;

for (int i = 0; i < 100000; ++i) {
    // Primary loop
    for (int c = 0; c < arraySize; ++c) {
        if (data[c] >= 128)
            int t = (data[c] - 128) >> 31;
        sum += data[c];
        sum += ~t & data[c];
    }
}

System.out.println((System.nanoTime() - start) / 1000000000.0);
System.out.println("sum: "+ sum);

Source: Harshal Parekh, “Branch Prediction — Everything you need to know.”