18-447
Computer Architecture
Lecture 7: Pipelining

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2014, 1/29/2014
Can We Do Better?

- What limitations do you see with the multi-cycle design?

- Limited concurrency
  - Some hardware resources are idle during different phases of instruction processing cycle
  - “Fetch” logic is idle when an instruction is being “decoded” or “executed”
  - Most of the datapath is idle when a memory access is happening
Can We Use the Idle Hardware to Improve Concurrency?

- **Goal:** Concurrency \(\rightarrow\) throughput (more “work” completed in one cycle)

- **Idea:** When an instruction is using some resources in its processing phase, **process other instructions on idle resources** not needed by that instruction
  - E.g., when an instruction is being decoded, fetch the next instruction
  - E.g., when an instruction is being executed, decode another instruction
  - E.g., when an instruction is accessing data memory (ld/st), execute the next instruction
  - E.g., when an instruction is writing its result into the register file, access data memory for the next instruction
Pipelining: Basic Idea

- More systematically:
  - Pipeline the execution of multiple instructions
  - Analogy: “Assembly line processing” of instructions

- Idea:
  - Divide the instruction processing cycle into distinct “stages” of processing
  - Ensure there are enough hardware resources to process one instruction in each stage
  - Process a different instruction in each stage
    - Instructions consecutive in program order are processed in consecutive stages

- Benefit: Increases instruction processing throughput (1/CPI)
- Downside: Start thinking about this...
Example: Execution of Four Independent ADDs

- **Multi-cycle:** 4 cycles per instruction

  ![Multi-cycle pipeline](image)

- **Pipelined:** 4 cycles per 4 instructions (steady state)

  ![Pipelined pipeline](image)
The Laundry Analogy

- “place one dirty load of clothes in the washer”
- “when the washer is finished, place the wet load in the dryer”
- “when the dryer is finished, take out the dry load and fold”
- “when folding is finished, ask your roommate (??) to put the clothes away”

- steps to do a load are sequentially dependent
- no dependence between different loads
- different steps do not share resources

Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Pipelining Multiple Loads of Laundry

- 4 loads of laundry in parallel
- no additional resources
- throughput increased by 4
- latency per load is the same

Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Pipelining Multiple Loads of Laundry: In Practice

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

the slowest step decides throughput
Pipelining Multiple Loads of Laundry: In Practice

Throughput restored (2 loads per hour) using 2 dryers

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

- **Goal:** Increase throughput with little increase in cost (hardware cost, in case of instruction processing)

- Repetition of *identical operations*
  - The same operation is repeated on a large number of different inputs

- Repetition of *independent operations*
  - No dependencies between repeated operations

- *Uniformly partitionable suboperations*
  - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)

- Fitting examples: automobile assembly line, doing laundry
  - What about the instruction processing “cycle”?
Ideal Pipelining

- Combinational logic (F, D, E, M, W) takes \( T \) psec.
  - Bandwidth: \( \sim \frac{1}{T} \)

- Takes \( \frac{T}{2} \) psec for (F, D, E) and \( \frac{T}{2} \) psec for (M, W).
  - Bandwidth: \( \sim \frac{2}{T} \)

- Takes \( \frac{T}{3} \) psec for (F, D), \( \frac{T}{3} \) psec for (E, M), and \( \frac{T}{3} \) psec for (M, W).
  - Bandwidth: \( \sim \frac{3}{T} \)
More Realistic Pipeline: Throughput

- Nonpipelined version with delay $T$
  \[ BW = \frac{1}{T+S} \] where $S$ = latch delay

- $k$-stage pipelined version
  \[ BW_{k\text{-stage}} = \frac{1}{T/k + S} \]
  \[ BW_{\text{max}} = \frac{1}{1 \text{ gate delay } + S} \]
More Realistic Pipeline: Cost

- Nonpipelined version with combinational cost $G$
  \[ \text{Cost} = G + L \text{ where } L = \text{latch cost} \]

- $k$-stage pipelined version
  \[ \text{Cost}_{k\text{-stage}} = G + Lk \]
Pipelining Instruction Processing
Remember: The Instruction Processing Cycle

1. Instruction fetch (IF)
2. Instruction decode and register operand fetch (ID/RF)
3. Execute/Evaluate memory address (EX/AG)
4. Memory operand fetch (MEM)
5. Store/writeback result (WB)
Remember the Single-Cycle Uarch
Dividing Into Stages

<table>
<thead>
<tr>
<th>200ps</th>
<th>100ps</th>
<th>200ps</th>
<th>200ps</th>
<th>100ps</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF: Instruction fetch</td>
<td>ID: Instruction decode/register file read</td>
<td>EX: Execute/address calculation</td>
<td>MEM: Memory access</td>
<td>WB: Write back</td>
</tr>
</tbody>
</table>

Is this the correct partitioning?
Why not 4 or 6 stages? Why not different boundaries?

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

5-stage speedup is 4, not 5 as predicted by the ideal model. Why?
Enabling Pipelined Processing: Pipeline Registers

No resource is used by more than 1 stage!

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

All instruction classes must follow the same path and timing through the pipeline stages. Any performance impact?

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

Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Illustrating Pipeline Operation: Operation View

```
<table>
<thead>
<tr>
<th>Inst_0</th>
<th>Inst_1</th>
<th>Inst_2</th>
<th>Inst_3</th>
<th>Inst_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_0</td>
<td>t_1</td>
<td>t_2</td>
<td>t_3</td>
<td>t_4</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>
```

Diagram showing pipeline stages at different times: t_0 to t_5.
Illustrating Pipeline Operation: Resource View

<table>
<thead>
<tr>
<th></th>
<th>$t_0$</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
<th>$t_4$</th>
<th>$t_5$</th>
<th>$t_6$</th>
<th>$t_7$</th>
<th>$t_8$</th>
<th>$t_9$</th>
<th>$t_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IF</strong></td>
<td>$l_0$</td>
<td>$l_1$</td>
<td>$l_2$</td>
<td>$l_3$</td>
<td>$l_4$</td>
<td>$l_5$</td>
<td>$l_6$</td>
<td>$l_7$</td>
<td>$l_8$</td>
<td>$l_9$</td>
<td>$l_{10}$</td>
</tr>
<tr>
<td><strong>ID</strong></td>
<td></td>
<td>$l_0$</td>
<td>$l_1$</td>
<td>$l_2$</td>
<td>$l_3$</td>
<td>$l_4$</td>
<td>$l_5$</td>
<td>$l_6$</td>
<td>$l_7$</td>
<td>$l_8$</td>
<td>$l_9$</td>
</tr>
<tr>
<td><strong>EX</strong></td>
<td></td>
<td></td>
<td>$l_0$</td>
<td>$l_1$</td>
<td>$l_2$</td>
<td>$l_3$</td>
<td>$l_4$</td>
<td>$l_5$</td>
<td>$l_6$</td>
<td>$l_7$</td>
<td>$l_8$</td>
</tr>
<tr>
<td><strong>MEM</strong></td>
<td></td>
<td></td>
<td></td>
<td>$l_0$</td>
<td>$l_1$</td>
<td>$l_2$</td>
<td>$l_3$</td>
<td>$l_4$</td>
<td>$l_5$</td>
<td>$l_6$</td>
<td>$l_7$</td>
</tr>
<tr>
<td><strong>WB</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$l_0$</td>
<td>$l_1$</td>
<td>$l_2$</td>
<td>$l_3$</td>
<td>$l_4$</td>
<td>$l_5$</td>
<td>$l_6$</td>
</tr>
</tbody>
</table>
Control Points in a Pipeline

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

Identical set of control points as the single-cycle datapath!!
Control Signals in a Pipeline

- For a given instruction
  - same control signals as single-cycle, but
  - control signals required at different cycles, depending on stage
  - decode once using the same logic as single-cycle and buffer control signals until consumed
  - or carry relevant “instruction word/field” down the pipeline and decode locally within each or in a previous stage

Which one is better?
Pipelined Control Signals

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

- Goal: Increase throughput with little increase in cost (hardware cost, in case of instruction processing)

- Repetition of **identical operations**
  - The same operation is repeated on a large number of different inputs

- Repetition of **independent operations**
  - No dependencies between repeated operations

- **Uniformly partitionable suboperations**
  - Processing an be evenly divided into uniform-latency suboperations (that do not share resources)

- Fitting examples: automobile assembly line, doing laundry
  - What about the instruction processing “cycle”?
Instruction Pipeline: Not An Ideal Pipeline

- Identical operations ... NOT!
  ⇒ different instructions do not need all stages
  - Forcing different instructions to go through the same multi-function pipe
  → external fragmentation (some pipe stages idle for some instructions)

- Uniform suboperations ... NOT!
  ⇒ difficult to balance the different pipeline stages
  - Not all pipeline stages do the same amount of work
  → internal fragmentation (some pipe stages are too-fast but take the same clock cycle time)

- Independent operations ... NOT!
  ⇒ instructions are not independent of each other
  - Need to detect and resolve inter-instruction dependencies to ensure the pipeline operates correctly
  → Pipeline is not always moving (it stalls)
Issues in Pipeline Design

- Balancing work in pipeline stages
  - How many stages and what is done in each stage

- Keeping the pipeline *correct, moving, and full* in the presence of events that disrupt pipeline flow
  - Handling dependences
    - Data
    - Control
  - Handling resource contention
  - Handling long-latency (multi-cycle) operations

- Handling exceptions, interrupts

- Advanced: Improving pipeline throughput
  - Minimizing stalls
Causes of Pipeline Stalls

- Resource contention

- Dependences (between instructions)
  - Data
  - Control

- Long-latency (multi-cycle) operations
Dependences and Their Types

- Also called “dependency” or less desirably “hazard”

- Dependencies dictate ordering requirements between instructions

- Two types
  - Data dependence
  - Control dependence

- Resource contention is sometimes called resource dependence
  - However, this is not fundamental to (dictated by) program semantics, so we will treat it separately
Handling Resource Contention

- Happens when instructions in two pipeline stages need the same resource

- **Solution 1: Eliminate the cause of contention**
  - Duplicate the resource or increase its throughput
    - E.g., use separate instruction and data memories (caches)
    - E.g., use multiple ports for memory structures

- **Solution 2: Detect the resource contention and stall one of the contending stages**
  - Which stage do you stall?
  - Example: What if you had a single read and write port for the register file?
Data Dependences

- Types of data dependences
  - Flow dependence (true data dependence – read after write)
  - Output dependence (write after write)
  - Anti dependence (write after read)

- Which ones cause stalls in a pipelined machine?
  - For all of them, we need to ensure semantics of the program are correct
  - Flow dependences always need to be obeyed because they constitute true dependence on a value
  - Anti and output dependences exist due to limited number of architectural registers
    - They are dependence on a name, not a value
    - We will later see what we can do about them
Data Dependence Types

Flow dependence
\[ r_3 \leftarrow r_1 \text{ op } r_2 \] Read-after-Write
\[ r_5 \leftarrow r_3 \text{ op } r_4 \] (RAW)

Anti dependence
\[ r_3 \leftarrow r_1 \text{ op } r_2 \] Write-after-Read
\[ r_1 \leftarrow r_4 \text{ op } r_5 \] (WAR)

Output-dependence
\[ r_3 \leftarrow r_1 \text{ op } r_2 \] Write-after-Write
\[ r_5 \leftarrow r_3 \text{ op } r_4 \] (WAW)
\[ r_3 \leftarrow r_6 \text{ op } r_7 \]
How to Handle Data Dependences

- Anti and output dependences are easier to handle
  - write to the destination in one stage and in program order

- Flow dependences are more interesting

- Five fundamental ways of handling flow dependences
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
Review: Pipelining: Basic Idea

- **Idea:**
  - Divide the instruction processing cycle into distinct “stages” of processing
  - Ensure there are enough hardware resources to process one instruction in each stage
  - Process a different instruction in each stage
    - Instructions consecutive in program order are processed in consecutive stages

- **Benefit:** Increases instruction processing throughput (1/CPI)
- **Downside:** ???
Review: Execution of Four Independent ADDs

- Multi-cycle: 4 cycles per instruction

- Pipelined: 4 cycles per 4 instructions (steady state)

Is life always this beautiful?
Review: Pipelined Operation Example

Is life always this beautiful?

Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Review: Instruction Pipeline: Not An Ideal Pipeline

- **Identical operations ... NOT!**
  - different instructions do not need all stages
    - Forcing different instructions to go through the same multi-function pipe
      → external fragmentation (some pipe stages idle for some instructions)

- **Uniform suboperations ... NOT!**
  - difficult to balance the different pipeline stages
    - Not all pipeline stages do the same amount of work
      → internal fragmentation (some pipe stages are too-fast but take the same clock cycle time)

- **Independent operations ... NOT!**
  - instructions are not independent of each other
    - Need to detect and resolve inter-instruction dependencies to ensure the pipeline operates correctly
      → Pipeline is not always moving (it stalls)
Review: Fundamental Issues in Pipeline Design

- **Balancing work in pipeline stages**
  - How many stages and what is done in each stage

- **Keeping the pipeline correct, moving, and full in the presence of events that disrupt pipeline flow**
  - Handling dependences
    - Data
    - Control
  - Handling resource contention
  - Handling long-latency (multi-cycle) operations

- **Handling exceptions, interrupts**

- **Advanced: Improving pipeline throughput**
  - Minimizing stalls
Review: Data Dependences

- Types of data dependences
  - Flow dependence (true data dependence – read after write)
  - Output dependence (write after write)
  - Anti dependence (write after read)

- Which ones cause stalls in a pipelined machine?
  - For all of them, we need to ensure semantics of the program is correct
  - Flow dependences always need to be obeyed because they constitute true dependence on a value
  - Anti and output dependences exist due to limited number of architectural registers
    - They are dependence on a name, not a value
    - We will later see what we can do about them
Data Dependence Types

Flow dependence
\[ r_3 \leftarrow r_1 \text{ op } r_2 \]  
\[ r_5 \leftarrow r_3 \text{ op } r_4 \]  
Read-after-Write (RAW)

Anti dependence
\[ r_3 \leftarrow r_1 \text{ op } r_2 \]  
\[ r_1 \leftarrow r_4 \text{ op } r_5 \]  
Write-after-Read (WAR)

Output-dependence
\[ r_3 \leftarrow r_1 \text{ op } r_2 \]  
\[ r_5 \leftarrow r_3 \text{ op } r_4 \]  
\[ r_3 \leftarrow r_6 \text{ op } r_7 \]  
Write-after-Write (WAW)
Pipelined Operation Example

What if the SUB were dependent on LW?
How to Handle Data Dependences

- Anti and output dependences are easier to handle
  - write to the destination in one stage and in program order

- 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
Interlocking

- Detection of dependence between instructions in a pipelined processor to guarantee correct execution

- Software based interlocking vs.
- Hardware based interlocking

- MIPS acronym?
Approaches to Dependence Detection (I)

- **Scoreboarding**
  - Each register in register file has a Valid bit associated with it
  - An instruction that is writing to the register resets the Valid bit
  - An instruction in Decode stage checks if all its source and destination registers are Valid
    - Yes: No need to stall... No dependence
    - No: Stall the instruction

- **Advantage:**
  - Simple. 1 bit per register

- **Disadvantage:**
  - Need to stall for all types of dependences, not only flow dep.
Not Stalling on Anti and Output Dependences

- What changes would you make to the scoreboard to enable this?
Approaches to Dependence Detection (II)

- **Combinational dependence check logic**
  - Special logic that checks if any instruction in later stages is supposed to write to any source register of the instruction that is being decoded
  - Yes: stall the instruction/pipeline
  - No: no need to stall... no flow dependence

- **Advantage:**
  - No need to stall on anti and output dependences

- **Disadvantage:**
  - Logic is more complex than a scoreboard
  - Logic becomes more complex as we make the pipeline deeper and wider (flash-forward: think superscalar execution)
Once You Detect the Dependence in Hardware

- What do you do afterwards?

- Observation: Dependence between two instructions is detected before the communicated data value becomes available

- Option 1: Stall the dependent instruction right away
- Option 2: Stall the dependent instruction only when necessary → data forwarding/bypassing
- Option 3: ...

50
Data Forwarding/Bypassing

- Problem: A consumer (dependent) instruction has to wait in decode stage until the producer instruction writes its value in the register file.

- Goal: We do not want to stall the pipeline unnecessarily.

- Observation: The data value needed by the consumer instruction can be supplied directly from a later stage in the pipeline (instead of only from the register file).

- Idea: Add additional dependence check logic and data forwarding paths (buses) to supply the producer’s value to the consumer right after the value is available.

- Benefit: Consumer can move in the pipeline until the point the value can be supplied → less stalling.
A Special Case of Data Dependence

- Control dependence
  - Data dependence on the Instruction Pointer / Program Counter
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 know whether or not the fetched instruction is a control-flow instruction?
Data Dependence Handling: More Depth & Implementation
Remember: Data Dependence Types

Flow dependence
\[ r_3 \leftarrow r_1 \text{ op } r_2 \quad \text{Read-after-Write (RAW)} \]
\[ r_5 \leftarrow r_3 \text{ op } r_4 \]

Anti dependence
\[ r_3 \leftarrow r_1 \text{ op } r_2 \quad \text{Write-after-Read (WAR)} \]
\[ r_1 \leftarrow r_4 \text{ op } r_5 \]

Output-dependence
\[ r_3 \leftarrow r_1 \text{ op } r_2 \quad \text{Write-after-Write (WAW)} \]
\[ r_5 \leftarrow r_3 \text{ op } r_4 \]
\[ r_3 \leftarrow r_6 \text{ op } r_7 \]
How to Handle Data Dependences

- Anti and output dependences are easier to handle
  - write to the destination in one stage and in program order

- 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
RAW Dependence Handling

- Following flow dependences lead to conflicts in the 5-stage pipeline
Register Data Dependence Analysis

<table>
<thead>
<tr>
<th></th>
<th>R/I-Type</th>
<th>LW</th>
<th>SW</th>
<th>Br</th>
<th>J</th>
<th>Jr</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td>read RF</td>
<td>read RF</td>
<td>read RF</td>
<td></td>
<td>read RF</td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td>write RF</td>
<td>write RF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- For a given pipeline, when is there a potential conflict between 2 data dependent instructions?
  - dependence type: RAW, WAR, WAW?
  - instruction types involved?
  - distance between the two instructions?
Safe and Unsafe Movement of Pipeline

\[ \text{dist}(i,j) \leq \text{dist}(X,Y) \Rightarrow \text{Unsafe to keep } j \text{ moving} \]
\[ \text{dist}(i,j) > \text{dist}(X,Y) \Rightarrow \text{Safe} \]
### RAW Dependence Analysis Example

<table>
<thead>
<tr>
<th>R/I-Type</th>
<th>LW</th>
<th>SW</th>
<th>Br</th>
<th>J</th>
<th>Jr</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>read RF</td>
<td>read RF</td>
<td>read RF</td>
<td>read RF</td>
<td>read RF</td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>write RF</td>
<td>write RF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Instructions** \( I_A \) and \( I_B \) (where \( I_A \) comes before \( I_B \)) have RAW dependence iff
  - \( I_B \) (R/I, LW, SW, Br or JR) reads a register written by \( I_A \) (R/I or LW)
  - \( \text{dist}(I_A, I_B) \leq \text{dist}(ID, WB) = 3 \)

What about WAW and WAR dependence?

What about memory data dependence?
Pipeline Stall: Resolving Data Dependence

Stall==make the dependent instruction wait until its source data value is available

1. stop all up-stream stages
2. drain all down-stream stages
How to Implement Stalling

- Stall
  - disable PC and IR latching; ensure stalled instruction stays in its stage
  - Insert “invalid” instructions/nops into the stage following the stalled one

Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Instructions $I_A$ and $I_B$ (where $I_A$ comes before $I_B$) have RAW dependence iff

- $I_B$ (R/I, LW, SW, Br or JR) reads a register written by $I_B$ (R/I or LW)
- $\text{dist}(I_A, I_B) \leq \text{dist}(ID, WB) = 3$

In other words, must stall when $I_B$ in ID stage wants to read a register to be written by $I_A$ in EX, MEM or WB stage.
Stall Conditions

- **Helper functions**
  - `rs(I)` returns the `rs` field of `I`
  - `use_rs(I)` returns true if `I` requires `RF[rs]` and `rs! = r0`

- **Stall when**
  - `(rs(IR_id)==dest_{EX}) && use_rs(IR_id) && RegWrite_{EX}` or
  - `(rs(IR_id)==dest_{MEM}) && use_rs(IR_id) && RegWrite_{MEM}` or
  - `(rs(IR_id)==dest_{WB}) && use_rs(IR_id) && RegWrite_{WB}` or
  - `(rt(IR_id)==dest_{EX}) && use_rt(IR_id) && RegWrite_{EX}` or
  - `(rt(IR_id)==dest_{MEM}) && use_rt(IR_id) && RegWrite_{MEM}` or
  - `(rt(IR_id)==dest_{WB}) && use_rt(IR_id) && RegWrite_{WB}`

- It is crucial that the EX, MEM and WB stages continue to advance normally during stall cycles
Impact of Stall on Performance

- Each stall cycle corresponds to 1 lost ALU cycle

- For a program with N instructions and S stall cycles, the Average CPI is \((N+S)/N\)

- \(S\) depends on
  - frequency of RAW dependences
  - exact distance between the dependent instructions
  - distance between dependences

  Suppose \(i_1, i_2\) and \(i_3\) all depend on \(i_0\). Once \(i_1\)'s dependence is resolved, \(i_2\) and \(i_3\) must be okay too
Sample Assembly (P&H)

- for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) { ....... }

```assembly
addi $s1, $s0, -1
for2tst:
  slti $t0, $s1, 0
  bne $t0, $zero, exit2
  sll $t1, $s1, 2
  add $t2, $a0, $t1
  lw $t3, 0($t2)
  lw $t4, 4($t2)
  slt $t0, $t4, $t3
  beq $t0, $zero, exit2
  ........
addi $s1, $s1, -1
j for2tst
exit2:
```

3 stalls
3 stalls
3 stalls
3 stalls
3 stalls
3 stalls
3 stalls
3 stalls
Data Forwarding (or Data Bypassing)

- It is intuitive to think of RF as state
  - “add rx ry rz” literally means get values from RF[ry] and RF[rz] respectively and put result in RF[rx]

- But, RF is just a part of a computing abstraction
  - “add rx ry rz” means 1. get the results of the last instructions to define the values of RF[ry] and RF[rz], respectively, and 2. until another instruction redefines RF[rx], younger instructions that refers to RF[rx] should use this instruction’s result

- What matters is to maintain the correct “dataflow” between operations, thus
Resolving RAW Dependence with Forwarding

- Instructions $I_A$ and $I_B$ (where $I_A$ comes before $I_B$) have RAW dependence iff
  - $I_B$ (R/I, LW, SW, Br or JR) reads a register written by $I_A$ (R/I or LW)
  - $\text{dist}(I_A, I_B) \leq \text{dist}(\text{ID, WB}) = 3$

- In other words, if $I_B$ in ID stage reads a register written by $I_A$ in EX, MEM or WB stage, then the operand required by $I_B$ is not yet in RF
  - $\Rightarrow$ retrieve operand from datapath instead of the RF
  - $\Rightarrow$ retrieve operand from the youngest definition if multiple definitions are outstanding
Data Forwarding Paths (v1)

[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Data Forwarding Paths (v2)

Assumes RF forwards internally

b. With forwarding

[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Data Forwarding Logic (for v2)

if \((rs_{EX} \neq 0) \&\& (rs_{EX} == dest_{MEM}) \&\& RegWrite_{MEM}\) then
   forward operand from MEM stage  // dist=1
else if \((rs_{EX} \neq 0) \&\& (rs_{EX} == dest_{WB}) \&\& RegWrite_{WB}\) then
   forward operand from WB stage  // dist=2
else
   use \(A_{EX}\) (operand from register file)  // dist \(\geq\) 3

Ordering matters!! Must check youngest match first

Why doesn’t \(\text{use}_\text{rs}( )\) appear in the forwarding logic?

What does the above not take into account?
### Data Forwarding (Dependence Analysis)

<table>
<thead>
<tr>
<th>R/I-Type</th>
<th>LW</th>
<th>SW</th>
<th>Br</th>
<th>J</th>
<th>Jr</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>use</td>
</tr>
<tr>
<td>EX</td>
<td>use</td>
<td>use</td>
<td>use</td>
<td>use</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>produce</td>
<td>(use)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Even with data-forwarding, RAW dependence on an immediately preceding LW instruction requires a stall
Sample Assembly, Revisited (P&H)

- for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) { ...... }
  - addi $s1, $s0, -1
  - for2tst: slti $t0, $s1, 0
  - bne $t0, $zero, exit2
  - sll $t1, $s1, 2
  - add $t2, $a0, $t1
  - lw $t3, 0($t2)
  - lw $t4, 4($t2)
  - nop
  - slt $t0, $t4, $t3
  - beq $t0, $zero, exit2
  .........
  - addi $s1, $s1, -1
  - j for2tst

exit2:
Pipelining the LC-3b
Pipelining the LC-3b

- Let’s remember the single-bus datapath

- We’ll divide it into 5 stages
  - Fetch
  - Decode/RF Access
  - Address Generation/Execute
  - Memory
  - Store Result

- Conservative handling of data and control dependences
  - Stall on branch
  - Stall on flow dependence
An Example LC-3b Pipeline
Control of the LC-3b Pipeline

- Three types of control signals

- Datapath Control Signals
  - Control signals that control the operation of the datapath

- Control Store Signals
  - Control signals (microinstructions) stored in control store to be used in pipelined datapath (can be propagated to stages later than decode)

- Stall Signals
  - Ensure the pipeline operates correctly in the presence of dependencies
<table>
<thead>
<tr>
<th>Stage</th>
<th>Signal Name</th>
<th>Signal Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>FETCH</td>
<td>MEM.PCMUX/2;††</td>
<td>PC+2 ;select pc+2 \n TARGET.PC ;select MEM.TARGET.PC (branch target) \n TRAP.PC ;select MEM.TRAP.PC</td>
</tr>
</tbody>
</table>
|            | LD.PC/1†                        | NO(0), LOAD(1) \n \n LD.DE/1†                        | NO(0), LOAD(1) \n \n DE.BR.OP/1                        | NO(0), YES(1) \n \n SR2.NEEDED/1                        | NO(0), LOAD(1) \n \n SR2.IDMUX/1                        | NO(0), LOAD(1) \n \n SR1.NEEDED/1                        | NO(0), YES(1) \n \n SR1.INSIDE/1                        | NO(0), LOAD(1) \n \n SR1.IDMUX/1                        | NO(0), LOAD(1) \n \n V.AGEX.LD.CC/1††                   | NO(0), LOAD(1) \n \n V.MEM.LD.CC/1††                   | NO(0), LOAD(1) \n \n V.SR.LD.CC/1††                   | NO(0), LOAD(1) \n \n V.AGEX.LD.REG/1††                  | NO(0), LOAD(1) \n \n V.MEM.LD.REG/1††                  | NO(0), LOAD(1) \n \n V.SR.LD.REG/1††                  | NO(0), LOAD(1) \n \n DECIDE | DRMUX/1                          | 11.9 ;destination IR[11:9] \n \n R7 ;destination R7 \n \n SR2.NEEDED/1                        | NO(0), YES(1) \n \n SR1.NEEDED/1                        | NO(0), LOAD(1) \n \n SR1.INSIDE/1                        | NO(0), LOAD(1) \n \n SR1.IDMUX/1                        | NO(0), LOAD(1) \n \n V.AGEX.LD.CC/1††                   | NO(0), LOAD(1) \n \n V.MEM.LD.CC/1††                   | NO(0), LOAD(1) \n \n V.SR.LD.CC/1††                   | NO(0), LOAD(1) \n \n V.AGEX.LD.REG/1††                  | NO(0), LOAD(1) \n \n V.MEM.LD.REG/1††                  | NO(0), LOAD(1) \n \n V.SR.LD.REG/1††                  | NO(0), LOAD(1) \n \n AGEX   | ADDR1MUX/1                       | NPC ;select value from AGEX.NPC \n \n BaseR ;select value from AGEX.SR1(BaseR) \n \n ADDR2MUX/2                          | ZERO ;select the value zero \n \n offset6 ;select SEXT[IR[5:0]] \n \n PCoffset9 ;select SEXT[IR[8:0]] \n \n PCoffset11 ;select SEXT[IR[10:0]] \n \n LSHFI/1                             | NO(0), 1bit Left shift(1) \n \n ADDRESSMUX/1                       | 7.0 ;select LSHF(ZEXT[IR[7:0]],1) \n \n SR2MUX/1                           | ADDER ;select output of address adder \n \n SR2                        | 4.0 ;IR[4:0] \n \n SR1MUX/1                           | ADDER ;select output of address adder \n \n SR1                        | 4.0 ;IR[4:0] \n \n ALUK/2                              | ADD(00), AND(01) \n \n XOR(10), PASSB(11) \n \n ALU RESULTMUX/1                     | SHIFTER ;select output of the shifter \n \n ALU                        | 4.0 ;IR[4:0] \n \n MEM                  | DCACHE.EN/1                       | NO(0), YES(1) ;asserted if the instruction accesses memory \n \n DCACHE.RW/1                       | RD(0), WR(1) \n \n DATA.SIZE/1                       | BYTE(0), WORD(1) \n \n BR.OP/1                            | NO(0), BR(1) ;BR \n \n UNCON.OP/1                       | NO(0), Uncond.BR(1) ;JMP,RET, JSR, JSRR \n \n TRAP.OP/1                        | NO(0), Trap(1) ;TRAP \n \n SR       | DR.VALUEMUX/2                    | ADDRESS ;select value from SR.ADDRESS \n \n DATA ;select value from SR.DATA \n \n NPC ;select value from SR.NPC \n \n ALU ;select value from SR.ALU.RESULT \n \n L.D.REG/1                          | NO(0), LOAD(1) \n \n L.D.CC/1                           | NO(0), LOAD(1) \n
Table 1: Data Path Control Signals
††: The control signal is generated by logic in that stage
†††: The control signal is generated by logic in another stage
## Control Store in a Pipelined Machine

<table>
<thead>
<tr>
<th>Number</th>
<th>Signal Name</th>
<th>Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SR1.NEEDED</td>
<td>DECODE</td>
</tr>
<tr>
<td>1</td>
<td>SR2.NEEDED</td>
<td>DECODE</td>
</tr>
<tr>
<td>2</td>
<td>DRMUX</td>
<td>DECODE</td>
</tr>
<tr>
<td>3</td>
<td>ADDR1MUX</td>
<td>AGEX</td>
</tr>
<tr>
<td>4</td>
<td>ADDR2MUX1</td>
<td>AGEX</td>
</tr>
<tr>
<td>5</td>
<td>ADDR2MUX0</td>
<td>AGEX</td>
</tr>
<tr>
<td>6</td>
<td>LSHF1</td>
<td>AGEX</td>
</tr>
<tr>
<td>7</td>
<td>ADDRESSMUX</td>
<td>AGEX</td>
</tr>
<tr>
<td>8</td>
<td>SR2MUX</td>
<td>AGEX</td>
</tr>
<tr>
<td>9</td>
<td>ALUK1</td>
<td>AGEX</td>
</tr>
<tr>
<td>10</td>
<td>ALUK0</td>
<td>AGEX</td>
</tr>
<tr>
<td>11</td>
<td>ALU.RESULTMUX</td>
<td>AGEX</td>
</tr>
<tr>
<td>12</td>
<td>BR.OP</td>
<td>DECODE, MEM</td>
</tr>
<tr>
<td>13</td>
<td>UNCON.OP</td>
<td>MEM</td>
</tr>
<tr>
<td>14</td>
<td>TRAP.OP</td>
<td>MEM</td>
</tr>
<tr>
<td>15</td>
<td>BR.STALL</td>
<td>DECODE, AGEX, MEM</td>
</tr>
<tr>
<td>16</td>
<td>DCACHE.EN</td>
<td>MEM</td>
</tr>
<tr>
<td>17</td>
<td>DCACHE.RW</td>
<td>MEM</td>
</tr>
<tr>
<td>18</td>
<td>DATA.SIZE</td>
<td>MEM</td>
</tr>
<tr>
<td>19</td>
<td>DR.VALUEMUX1</td>
<td>SR</td>
</tr>
<tr>
<td>20</td>
<td>DR.VALUEMUX0</td>
<td>SR</td>
</tr>
<tr>
<td>21</td>
<td>LD.REG</td>
<td>AGEX, MEM, SR</td>
</tr>
<tr>
<td>22</td>
<td>LD.CC</td>
<td>AGEX, MEM, SR</td>
</tr>
</tbody>
</table>

Table 2: Control Store ROM Signals
Stall Signals

- **Pipeline stall**: Pipeline does not move because an operation in a stage cannot complete.
- **Stall Signals**: Ensure the pipeline operates correctly in the presence of such an operation.
- **Why could an operation in a stage not complete?**

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Generated in</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICACHE.R/1:</td>
<td>FETCH</td>
<td>NO, READY</td>
</tr>
<tr>
<td>DEP.STALL/1:</td>
<td>DEC</td>
<td>NO, STALL</td>
</tr>
<tr>
<td>V.DE.BR.STALL/1:</td>
<td>DEC</td>
<td>NO, STALL</td>
</tr>
<tr>
<td>V.AGEX.BR.STALL/1:</td>
<td>AGEX</td>
<td>NO, STALL</td>
</tr>
<tr>
<td>MEM.STALL/1:</td>
<td>MEM</td>
<td>NO, STALL</td>
</tr>
<tr>
<td>V.MEM.BR.STALL/1:</td>
<td>MEM</td>
<td>NO, STALL</td>
</tr>
</tbody>
</table>

Table 3: STALL Signals