Design of Digital Circuits
Lecture 14: Pipelining Issues

Prof. Onur Mutlu
ETH Zurich
Spring 2019
5 April 2019
Required Readings

**This week**
- Pipelining
  - H&H, Chapter 7.5
- Pipelining Issues
  - H&H, Chapter 7.8.1-7.8.3

**Next week**
- Out-of-order execution
  - H&H, Chapter 7.8-7.9
  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts
Agenda for Today & Next Few Lectures

- **Last week**
  - Single-cycle Microarchitectures
  - Multi-cycle Microarchitectures

- **This week**
  - Pipelining
  - Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...

- **Next week**
  - Out-of-Order Execution
  - Issues in OoO Execution: Load-Store Handling, ...
Review: 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

- Six 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
  - Detect and move it out of the way for independent instructions
  - Predict the needed value(s), execute “speculatively”, and verify
  - Do something else (fine-grained multithreading)
    - No need to detect
Stalling

lw $s0, 40($0)
and $t0, $s0, $s1
or $t1, $s4, $s0
sub $t2, $s0, $s5
Stalling Hardware

- **Stalls are supported by:**
  - adding enable inputs (EN) to the Fetch and Decode pipeline registers
  - and a synchronous reset/clear (CLR) input to the Execute pipeline register
    - or an INV bit associated with each pipeline register

- **When a lw stall occurs**
  - StallD and StallF are asserted to force the Decode and Fetch stage pipeline registers to hold their old values.
  - FlushE is also asserted to clear the contents of the Execute stage pipeline register, introducing a bubble
Stalling Hardware
Control Dependences

- Special case of data dependence: dependence on PC

- beq:
  - branch is not determined until the fourth stage of the pipeline
  - Instructions after the branch are fetched before branch is resolved
    - Always predict that the next sequential instruction is fetched
    - Called “Always not taken” prediction
  - These instructions must be flushed if the branch is taken

- Branch misprediction penalty
  - number of instructions flushed when branch is taken
  - May be reduced by determining branch earlier
Control Dependence: Original Pipeline
Control Dependence

20  beq $t1, $t2, 40
24  and $t0, $s0, $s1
28  or $t1, $s4, $s0
2C  sub $t2, $s0, $s5
30  ...
...
64  slt $t3, $s2, $s3

Flush these instructions
Introduces another data dependency in Decode stage..
Early Branch Resolution

20  beq $t1, $t2, 40
24  and $t0, $s0, $s1
28  or $t1, $s4, $s0
2C  sub $t2, $s0, $s5
30  ...
...
64  slt $t3, $s2, $s3

Flush this instruction
Early Branch Resolution: Good Idea?

- **Advantages**
  - Reduced branch misprediction penalty
    → Reduced CPI (cycles per instruction)

- **Disadvantages**
  - Potential increase in clock cycle time?
    → Higher $T_{clock}$?
  - Additional hardware cost
    → Specialized and likely not used by other instructions
Data Forwarding for Early Branch Resolution

Data forwarding for early branch resolution.
// Forwarding logic:
assign ForwardAD = (rsD != 0) & (rsD == WriteRegM) & RegWriteM;
assign ForwardBD = (rtD != 0) & (rtD == WriteRegM) & RegWriteM;

// Stalling logic:
assign lwstall = ((rsD == rtE) | (rtD == rtE)) & MemtoRegE;

assign branchstall = (BranchD & RegWriteE &
                      (WriteRegE == rsD | WriteRegE == rtD))
                   | (BranchD & MemtoRegM &
                      (WriteRegM == rsD | WriteRegM == rtD));

// Stall signals;
assign StallF = lwstall | branchstall;
assign StallD = lwstall | branchstall;
assign FLushE = lwstall | branchstall;
Doing Better: Smarter Branch Prediction

- Guess whether branch will be taken
  - Backward branches are usually taken (loops)
  - Consider history of whether branch was previously taken to improve the guess

- Good prediction reduces the fraction of branches requiring a flush
Pipelined Performance Example

- **SPECINT2006 benchmark:**
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type

- **Suppose:**
  - 40% of loads used by next instruction
  - 25% of branches mispredicted

- **All jumps flush next instruction**

- **What is the average CPI?**
Pipelined Performance Example Solution

- Load/Branch CPI = 1 when no stall/flush, 2 when stall/flush. Thus:
  - \( \text{CPI}_{lw} = 1(0.6) + 2(0.4) = 1.4 \)  \( \text{Average CPI for load} \)
  - \( \text{CPI}_{\text{beq}} = 1(0.75) + 2(0.25) = 1.25 \)  \( \text{Average CPI for branch} \)

- And
  - \( \text{Average CPI} = \)
Load/Branch CPI = 1 when no stall/flush, 2 when stall/flush. Thus:

- $CPI_{lw} = 1(0.6) + 2(0.4) = 1.4$  
  Average CPI for load
- $CPI_{beq} = 1(0.75) + 2(0.25) = 1.25$  
  Average CPI for branch

And

- $Average\ CPI = (0.25)(1.4) + (0.1)(1) + (0.11)(1.25) + (0.02)(2) + (0.52)(1)$
- $load$
- $store$
- $beq$
- $jump$
- $r$-type

$= 1.15$
Pipelined Performance

There are 5 stages, and 5 different timing paths:

\[ T_c = \max \{ \]
\[ t_{pcq} + t_{mem} + t_{setup} \]
\[ 2(t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup}) \]
\[ t_{pcq} + t_{mux} + t_{mux} + t_{ALU} + t_{setup} \]
\[ t_{pcq} + t_{memwrite} + t_{setup} \]
\[ 2(t_{pcq} + t_{mux} + t_{RFwrite}) \]
\[ \}

The operation speed **depends** on the **slowest operation**

Decode and Writeback use register file and have only half a clock cycle to complete, that is why there is a 2 in front of them
# Pipelined Performance Example

<table>
<thead>
<tr>
<th>Element</th>
<th>Parameter</th>
<th>Delay (ps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register clock-to-Q</td>
<td>( t_{pcq_PC} )</td>
<td>30</td>
</tr>
<tr>
<td>Register setup</td>
<td>( t_{\text{setup}} )</td>
<td>20</td>
</tr>
<tr>
<td>Multiplexer</td>
<td>( t_{\text{mux}} )</td>
<td>25</td>
</tr>
<tr>
<td>ALU</td>
<td>( t_{\text{ALU}} )</td>
<td>200</td>
</tr>
<tr>
<td>Memory read</td>
<td>( t_{\text{mem}} )</td>
<td>250</td>
</tr>
<tr>
<td>Register file read</td>
<td>( t_{\text{RFread}} )</td>
<td>150</td>
</tr>
<tr>
<td>Register file setup</td>
<td>( t_{\text{RFsetup}} )</td>
<td>20</td>
</tr>
<tr>
<td>Equality comparator</td>
<td>( t_{\text{eq}} )</td>
<td>40</td>
</tr>
<tr>
<td>AND gate</td>
<td>( t_{\text{AND}} )</td>
<td>15</td>
</tr>
<tr>
<td>Memory write</td>
<td>( T_{\text{memwrite}} )</td>
<td>220</td>
</tr>
<tr>
<td>Register file write</td>
<td>( t_{\text{RFwrite}} )</td>
<td>100</td>
</tr>
</tbody>
</table>

\[
T_c = 2(t_{\text{RFread}} + t_{\text{mux}} + t_{\text{eq}} + t_{\text{AND}} + t_{\text{mux}} + t_{\text{setup}}) \\
= 2[150 + 25 + 40 + 15 + 25 + 20] \text{ ps} \\
= 550 \text{ ps}
\]
Pipelined Performance Example

- For a program with 100 billion instructions executing on a pipelined MIPS processor:
  - CPI = 1.15
  - $T_c = 550 \text{ ps}$

- Execution Time
  \[
  \text{Execution Time} = (\# \text{ instructions}) \times \text{CPI} \times T_c \\
  = (100 \times 10^9)(1.15)(550 \times 10^{-12}) \\
  = 63 \text{ seconds}
  \]
Performance Summary for MIPS arch.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Execution Time (seconds)</th>
<th>Speedup (single-cycle is baseline)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-cycle</td>
<td>95</td>
<td>1</td>
</tr>
<tr>
<td>Multicycle</td>
<td>133</td>
<td>0.71</td>
</tr>
<tr>
<td>Pipelined</td>
<td>63</td>
<td>1.51</td>
</tr>
</tbody>
</table>

- Fastest of the three MIPS architectures is *Pipelined*.
- However, even though we have 5 fold pipelining, it is not 5 times faster than single cycle.
Questions to Ponder

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

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

- How does each impact different metrics?
  - Performance (and parts of the performance equation)
  - Complexity
  - Power consumption
  - Reliability
  - ...
More on Software vs. Hardware

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

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

- **How can the compiler alleviate this (i.e., estimate the unknown)?**
  - Answer: Profiling
Pipelining and Precise Exceptions: Preserving Sequential Semantics
Multi-Cycle Execution

- Not all instructions take the same amount of time for “execution”
- Idea: Have multiple different functional units that take different number of cycles
  - Can be pipelined or not pipelined
  - Can let independent instructions start execution on a different functional unit before a previous long-latency instruction finishes execution
Issues in Pipelining: Multi-Cycle Execute

- Instructions can take different number of cycles in EXECUTE stage
  - Integer ADD versus FP MULtiply

```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stage 1</th>
<th>Stage 2</th>
<th>Stage 3</th>
<th>Stage 4</th>
<th>Stage 5</th>
<th>Stage 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMUL R4 ← R1, R2</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>ADD R3 ← R1, R2</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMUL R2 ← R5, R6</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD R7 ← R5, R6</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- What is wrong with this picture in a Von Neumann architecture?
  - Sequential semantics of the ISA NOT preserved!
  - What if FMUL incurs an exception?
Exceptions vs. Interrupts

■ **Cause**
  - Exceptions: internal to the running thread
  - Interrupts: external to the running thread

■ **When to Handle**
  - Exceptions: when detected (and known to be non-speculative)
  - Interrupts: when convenient
    - Except for very high priority ones
      - Power failure
      - Machine check (error)

■ **Priority**: process (exception), depends (interrupt)

■ **Handling Context**: process (exception), system (interrupt)
Precise Exceptions/Interrupts

- The architectural state should be consistent (precise) when the exception/interrupt is ready to be handled.

1. All previous instructions should be completely retired.

2. No later instruction should be retired.

Retire = commit = finish execution and update arch. state
When the oldest instruction ready-to-be-retired is detected to have caused an exception, the control logic

- Ensures architectural state is precise (register file, PC, memory)
- Flushes all younger instructions in the pipeline
- Saves PC and registers (as specified by the ISA)
- Redirects the fetch engine to the appropriate exception handling routine
Why Do We Want Precise Exceptions?

- Semantics of the von Neumann model ISA specifies it
  - Remember von Neumann vs. Dataflow

- Aids software debugging

- Enables (easy) recovery from exceptions

- Enables (easily) restartable processes

- Enables traps into software (e.g., software implemented opcodes)
## Ensuring Precise Exceptions in Pipelining

- **Idea:** Make each operation take the same amount of time

```
FMUL R3 ← R1, R2
ADD  R4 ← R1, R2
```

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>D</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMUL R3</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>W</td>
</tr>
<tr>
<td>ADD R4</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>W</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>W</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>W</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>W</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>W</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>W</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>W</td>
</tr>
</tbody>
</table>

- **Downside**
  - Worst-case instruction latency determines all instructions’ latency
  - What about memory operations?
  - Each functional unit takes worst-case number of cycles?
Solutions

- Reorder buffer
- History buffer
- Future register file
- Checkpointing

Suggested reading

We will not cover these
Solution I: Reorder Buffer (ROB)

- **Idea:** Complete instructions out-of-order, but reorder them before making results visible to architectural state
- When instruction is decoded it reserves the next-sequential entry in the ROB
- When instruction completes, it writes result into ROB entry
- When instruction oldest in ROB and it has completed without exceptions, its result moved to reg. file or memory
What’s in a ROB Entry?

- Everything required to:
  - correctly reorder instructions back into the program order
  - update the architectural state with the instruction’s result(s), if instruction can retire without any issues
  - handle an exception/interrupt precisely, if an exception/interrupt needs to be handled before retiring the instruction

- Need valid bits to keep track of readiness of the result(s) and find out if the instruction has completed execution
Reorder Buffer: Independent Operations

- Result first written to ROB on instruction completion
- Result written to register file at commit time

What if a later operation needs a value in the reorder buffer?
- Read reorder buffer in parallel with the register file. How?
Reorder Buffer: How to Access?

- A register value can be in the register file, reorder buffer, (or bypass/forwarding paths)

```
Instruction Cache

→ Register File

→ Reorder Buffer

→ Random Access Memory (indexed with Register ID, which is the address of an entry)

→ Func Unit

→ Func Unit

→ Func Unit

bypass paths

Content Addressable Memory (searched with register ID, which is part of the content of an entry)
```
Simplifying Reorder Buffer Access

- **Idea:** *Use indirection*

- Access register file first (check if the register is valid)
  - If register not valid, register file stores the ID of the reorder buffer entry that contains (or will contain) the value of the register
  - **Mapping of the register to a ROB entry:** Register file maps the register to a reorder buffer entry if there is an in-flight instruction writing to the register

- Access reorder buffer next

- Now, reorder buffer does not need to be content addressable
Reorder Buffer in Intel Pentium III

Important: Register Renaming with a Reorder Buffer

- Output and anti dependencies are **not true dependencies**
  - **WHY?** The same register refers to values that have nothing to do with each other
  - **They exist due to lack of register ID’s (i.e. names) in the ISA**

- The register ID is **renamed** to the reorder buffer entry that will hold the register’s value
  - Register ID → ROB entry ID
  - Architectural register ID → Physical register ID
  - After renaming, ROB entry ID used to refer to the register

- This eliminates anti and output dependencies
  - **Gives the illusion that there are a large number of registers**
Renaming Example

- Assume
  - Register file has a pointer to the reorder buffer entry that contains or will contain the value, if the register is not valid
  - Reorder buffer works as described before

- Where is the latest definition of R3 for each instruction below in sequential order?
  - LD R0(0) → R3
  - LD R3, R1 → R10
  - MUL R1, R2 → R3
  - MUL R3, R4 → R11
  - ADD R5, R6 → R3
  - ADD R7, R8 → R12
In-Order Pipeline with Reorder Buffer

- **Decode (D):** Access regfile/ROB, allocate entry in ROB, check if instruction can execute, if so **dispatch** instruction
- **Execute (E):** Instructions can complete out-of-order
- **Completion (R):** Write result to reorder buffer
- **Retirement/Commit (W):** Check for exceptions; if none, write result to architectural register file or memory; else, flush pipeline and start from exception handler
- **In-order dispatch/execution, out-of-order completion, in-order retirement**
Reorder Buffer Tradeoffs

- **Advantages**
  - Conceptually simple for supporting precise exceptions
  - Can eliminate false dependences

- **Disadvantages**
  - Reorder buffer needs to be accessed to get the results that are yet to be written to the register file
    - CAM or indirection $\rightarrow$ increased latency and complexity

- **Other solutions aim to eliminate the disadvantages**
  - History buffer
  - Future file
  - Checkpointing

*We will not cover these*
Out-of-Order Execution
(Dynamic Instruction Scheduling)
An In-order Pipeline

- **Dispatch**: Act of sending an instruction to a functional unit
- **Renaming with ROB eliminates stalls due to false dependencies**
- **Problem**: A true data dependency stalls dispatch of younger instructions into functional (execution) units
Can We Do Better?

- What do the following two pieces of code have in common (with respect to execution in the previous design)?

```
IMUL  R3 ← R1, R2
ADD   R3 ← R3, R1
ADD   R4 ← R6, R7
IMUL  R5 ← R6, R8
ADD   R7 ← R9, R9
```

```
LD      R3 ← R1(0)
ADD   R3 ← R3, R1
ADD   R4 ← R6, R7
IMUL  R5 ← R6, R8
ADD   R7 ← R9, R9
```

- **Answer:** First ADD stalls the whole pipeline!
  - ADD cannot dispatch because its source registers unavailable
  - Later **independent** instructions cannot get executed

- How are the above code portions different?
  - **Answer:** Load latency is variable (unknown until runtime)
  - What does this affect? Think compiler vs. microarchitecture
Preventing Dispatch Stalls

- **Problem:** in-order dispatch (scheduling, or execution)

- **Solution:** out-of-order dispatch (scheduling, or execution)

Actually, we have seen the basic idea before:
- **Dataflow:** fetch and “fire” an instruction only when its inputs are ready
- We will use similar principles, but not expose it in the ISA

Aside: Any other way to prevent dispatch stalls?
1. Compile-time instruction scheduling/reordering
2. Value prediction
3. Fine-grained multithreading
Out-of-order Execution (Dynamic Scheduling)

- **Idea:** Move the dependent instructions out of the way of independent ones (s.t. independent ones can execute)
  - Rest areas for dependent instructions: Reservation stations

- Monitor the source “values” of each instruction in the resting area

- When all source “values” of an instruction are available, “fire” (i.e. dispatch) the instruction
  - Instructions dispatched in dataflow (not control-flow) order

- **Benefit:**
  - Latency tolerance: Allows independent instructions to execute and complete in the presence of a long-latency operation
# In-order vs. Out-of-order Dispatch

- **In order dispatch + precise exceptions:**

  \[
  \begin{array}{cccccc}
  F & D & E & E & E & E & R & W \\
  F & D & \text{STALL} & E & R & W \\
  F & \text{STALL} & D & E & R & W \\
  F & D & E & E & E & E & E & E & R & W \\
  F & D & \text{STALL} & E & R & W \\
  \end{array}
  \]

  + IMUL \( R3 \leftarrow R1, R2 \)
  + ADD \( R3 \leftarrow R3, R1 \)
  + ADD \( R1 \leftarrow R6, R7 \)
  + IMUL \( R5 \leftarrow R6, R8 \)
  + ADD \( R7 \leftarrow R3, R5 \)

- **Out-of-order dispatch + precise exceptions:**

  \[
  \begin{array}{cccccc}
  F & D & E & E & E & E & R & W \\
  F & D & \text{WAIT} & E & R & W \\
  F & D & E & E & R & W \\
  F & D & E & E & E & E & E & E & R & W \\
  F & D & \text{WAIT} & E & R & W \\
  \end{array}
  \]

- **16 vs. 12 cycles**
Enabling OoO Execution

1. Need to link the consumer of a value to the producer
   - **Register renaming**: Associate a “tag” with each data value

2. Need to buffer instructions until they are ready to execute
   - Insert instruction into reservation stations after renaming

3. Instructions need to keep track of readiness of source values
   - Broadcast the “tag” when the value is produced
   - Instructions compare their “source tags” to the broadcast tag
     → if match, source value becomes ready

4. When all source values of an instruction are ready, need to dispatch the instruction to its functional unit (FU)
   - Instruction wakes up if all sources are ready
   - If multiple instructions are awake, need to select one per FU