DEE 1053 Computer Organization
Lecture 6: Pipelining

Dr. Tian-Sheuan Chang
tschang@twins.ee.nctu.edu.tw
Dept. Electronics Engineering
National Chiao-Tung University
Outline

• Overview of pipelining
• A pipelined datapath
• Pipelined control
• Data hazards and forwarding
• Data hazards and stalls
• Branch hazards
• Exceptions in the pipelined datapath
• Advanced pipelining
Overview of Pipelining (6.1)

- Pipelining
  - Multiple instructions are overlapped in execution

Nonpipelined: 8 hour

Pipelined: 3.5 hours

Speedup = 8/3.5 = 2.3 ⇒ 4?
MIPS Architecture

• Each MIPS instruction take five steps
  – Instruction fetch (IF)
  – Instruction decode and register fetch
  – ALU operation or calculate the address
  – Data access in data memory
  – Register Write

• Instruction execution
  – 5 steps -> 5 stages
### MIPS Architecture

#### Class Function units

<table>
<thead>
<tr>
<th>Class</th>
<th>Instruction fetch</th>
<th>Register access</th>
<th>ALU</th>
<th>Register access</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>Instruction fetch</td>
<td>Register access</td>
<td>ALU</td>
<td>Register access</td>
</tr>
<tr>
<td>Load word</td>
<td>Instruction fetch</td>
<td>Register access</td>
<td>ALU</td>
<td>Memory access</td>
</tr>
<tr>
<td>Store word</td>
<td>Instruction fetch</td>
<td>Register access</td>
<td>ALU</td>
<td>Memory access</td>
</tr>
<tr>
<td>Branch</td>
<td>Instruction fetch</td>
<td>Register access</td>
<td>ALU</td>
<td>Memory access</td>
</tr>
<tr>
<td>Jump</td>
<td>Instruction fetch</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Assume MUX, control unit, PC access, sign extension have no delay**

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>Instruction memory</th>
<th>Register read</th>
<th>ALU operation</th>
<th>Data memory</th>
<th>Register write</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>200</td>
<td>100</td>
<td>200</td>
<td>0</td>
<td>100</td>
<td>500</td>
</tr>
<tr>
<td>Load word</td>
<td>200</td>
<td>100</td>
<td>200</td>
<td>200</td>
<td>100</td>
<td>800</td>
</tr>
<tr>
<td>Store word</td>
<td>200</td>
<td>100</td>
<td>200</td>
<td>200</td>
<td></td>
<td>700</td>
</tr>
<tr>
<td>Branch</td>
<td>200</td>
<td>100</td>
<td>200</td>
<td>0</td>
<td></td>
<td>500</td>
</tr>
<tr>
<td>Jump</td>
<td>200</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>200</td>
</tr>
</tbody>
</table>
Single Cycle v.s. Pipelined

**Instruction cycle = 800ps**

Program execution order (in instructions):

- `lw $1, 100($0)`
- `lw $2, 200($0)`
- `lw $3, 300($0)`

Time:

<table>
<thead>
<tr>
<th>200</th>
<th>400</th>
<th>600</th>
<th>800</th>
<th>1000</th>
<th>1200</th>
<th>1400</th>
<th>1600</th>
<th>1800</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Instruction fetch</td>
<td>Reg</td>
<td>ALU</td>
<td>Data access</td>
<td>Reg</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Instruction fetch</td>
<td>Reg</td>
<td>ALU</td>
<td>Data access</td>
<td>Reg</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Instruction fetch</td>
<td>Reg</td>
<td>ALU</td>
<td>Data access</td>
<td>Reg</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Instruction fetch</td>
<td>Reg</td>
<td>ALU</td>
<td>Data access</td>
<td>Reg</td>
<td></td>
</tr>
</tbody>
</table>

The time between the 1st and 4th load instruction: 2400ps v.s. 600ps

**Ideal speedup of the pipeline = pipeline stages**, Do we achieve this?
Pipelining SpeedUp

\[
\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipeline stages}}
\]

Ideal speedup = number of pipeline stages

- Conditions
  - Stages are perfectly balanced
  - Large number of instructions
- Total execution time is less important especially when large number of instructions
- So what’s problem in our previous pipeline?

- So Why Pipelining can Improve Performance
  - Improve performance by increasing instruction throughput
  - Instead of decreasing the execution time of an individual instructions
Designing Instruction Sets for Pipelining

• 1. Instructions are the same length
  – Easier to fetch instructions and decode

• 2. Only a few instruction formats
  – Source register field located in the same place in each instructions
  – Symmetry field
  – You can read the register file when determining the instruction type. If not, you have to split the pipeline stages

• 3. Memory operands only appear in loads or stores
  – This restriction enables us to use EX stage to calculate the address
  – What if not: => address, memory, execute

• 4. Operands must be aligned in memory
  – No two data memory accesses for single data transfer
Pipelining

- What makes it easy
  - all instructions are the same length
  - just a few instruction formats
  - memory operands appear only in loads and stores

- What makes it hard?
  - structural hazards: suppose we had only one memory
  - control hazards: need to worry about branch instructions
  - data hazards: an instruction depends on a previous instruction

- We’ll build a simple pipeline and look at these issues

- We’ll talk about modern processors and what really makes it hard:
  - exception handling
  - trying to improve performance with out-of-order execution, etc.
Pipeline Hazards (1)

- **Hazards**
  - Situations in pipelining when the next instruction cannot execute in the following clock cycles

- **Three types of hazards**
  - **Structural hazards**
    - Due to resource constraints
  - **Data hazards**
    - Due to data availability
  - **Control hazards**
    - Due to change of instruction flow
Pipeline Hazards (2)

• Structural hazards
  – Hardware cannot support the instructions executing in the same clock cycle
  – Limited resources
  – Eg. Memory accesses
  – Can you draw an example? Fig. 6.3
Structural Hazards

What if the 4th instruction appears?
PipeLine Hazards (3)

• Data hazards
  – An instruction depends on the results of a previous instruction still in the pipeline
  – **Data dependency**
  – Example
    ```
    add $s0, $t0, $t1; write at 5th stage
    sub $t2, $s0, $t3; read at 2nd stage
    Results: 3 bubbles (waiting cycle)
    ```

Could you draw the diagram?
Pipeline Hazards (4) Solutions to the Data Hazards

- Rely on compiler to remove these dependencies
- **Data forwarding (bypassing)**
  - Observation
    - We don't need to wait for the instruction to complete before trying to resolve the data hazards
    - Getting the needed data item early from the internal resources

![Diagram showing pipeline stages and data forwarding]

No stall after forwarding
Pipeline Hazards (5) Load-Use Data Hazard

- Still one **stall** (bubble) even the forwarding is applied
Pipeline Hazards (6) Solutions to the Data Hazards

- Rely on compiler (or DIY) to remove the these dependency
- Reordering code to avoid pipeline stalls
- Example
  A = B + E
  C = B + F

```
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t2, $t2
sw $t3, 12($t0)
lw $t4, 8($01)
add $t3, $t2, $t2
sw $t3, 12($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

```
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t2, $t2
sw $t3, 12($t0)
lw $t4, 8($01)
add $t3, $t2, $t2
sw $t3, 12($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

The hazard in blue line can be solved by forwarding
Pipeline Hazards (7) Control Hazards

- Flow of instruction addresses is not what the pipeline expected
  - What is the pipeline expected? Next instructions
- Due to Branch instruction
  - Simplest solution: **Stall on branch**
- Assume putting enough extra hardware so that can
  - Test registers, Calculate the branch address, Update the program counter
    - during the 2nd stage,
    - We still need one pipeline stall after the branch
Pipeline Hazards (8) Performance of “Stall on Branch”

• Assume all other instructions have a CPI of 1
• Branch: 13%

CPI = 1 (ideal case) + 1 (one stall) * 0.13 = 1.13

• If not resolved by the 2nd stage, the situation will be worse

• Better Solution: Branch Prediction
Pipeline Hazards (9) 2nd Solution to Control Hazards

- **Predict**
  - Use *prediction* to handle branches
    - If predict right -> no stall
    - If predict wrong -> 1 stall

- **Approach**
  - Static prediction
    - Always *untaken*
    - Some branches as taken and some as untaken,
      - *Always taken for loop branch*
  - Dynamic prediction
    - Keep a history for each branch as taken or untaken
    - Use the recent past behavior to predict the future
    - Over 90% accuracy

- **What you should concern for prediction**
  - If guessed wrong, the pipeline control should ensure that the instruction following the wrongly guessed branch has *no effect* and must *restart the pipeline* from the proper branch address
Pipeline Hazards (10) Predict “Not Taken”

Right predict

Wrong predict
Pipeline Hazards (11) Third Solution to Control Hazards

- **Delayed Branch**
  - Place an instruction into the branch delay slot that is not affected by branch
    - Handwritten or compiler
    - Used in MIPS

Without delayed branch:
```
beq $1, $2, 40
add $4, $5, $6
```

(delayed branch slot)
```
beq $1, $2, 40
add $4, $5, $6
or $7, $8, $9
```

(delayed branch)
Note on Delayed Branch

• Compiler typically fill about 50% of branch delays slots with useful instruction
  – Imply 50% of slot could be “NOP”

• If the pipeline is deeper, more branch delays slots and even harder to fill
Pipeline summary

- Exploits **parallelism** among the instructions in a sequential instructions
- **Invisible** to the programmer
- Pipelining improve the **instruction throughput**
- Instead of **individual instruction latency**
Additional: Data Hazard Classification

- Classified according to the order of read and write accesses
- RAW (Read after write)
  - J tries to read a source before I write it, so J incorrectly get the old value
    add $s0, $t0, $t1; write at 5th stage
    sub $t2, $s0, $t3; read at 2nd stage
    IF ID EX MEM WB
    IF ID EXE MEM WB
  - Some can be solved by forwarding
- WAW (Write after write)
  - J tries to write an operand before it is written by I, leaving the old value
    lw $s0, 100($t0); write at 6th stage
    add $s0, $t1, $t3; write at 4th stage
    IF ID EX MEM1 MEM2 WB
    IF ID EXE WB
  - Present only in the pipelines that write in more than one pipe stage or out-of-order execution (allowed instruction continuing even previous one is stalled)
Additional: Data Hazard Classification

- **WAR (Write after read)**
  - J tries to write a destination before it is read by I, so I incorrectly gets the new value
  
  ```assembly
  sw  $s0, 100($t0); use $t0 at 5th stage
  add $t0, $t1, $t3; write at 4th stage
  ```

  - Occurs for complex instruction sets that support autoincrement addressing and require operands to be read late in the pipeline
  - Also occurs in the out-of-order execution

- **RAR (Read after read):** this is not a hazard
Outline

• Overview of pipelining
• **A pipelined datapath**
• Pipelined control
• Data hazards and forwarding
• Data hazards and stalls
• Branch hazards
• Exceptions in the pipelined datapath
• Advanced pipelining
A Pipelined Datapath

- **5 steps** for instruction execution
- **5 stages** pipeline
  - 1. **IF**
    - Instruction fetch
  - 2. **ID**
    - Instruction decode and register fetch
  - 3. **EXE**
    - Execution or address calculation
  - 4. **MEM**
    - Data memory access
  - 5. **WB**
    - Write back
Review of Single Cycle Datapath

Pipelined execution of single cycle Datapath
3 datapath ->
Shared single datapath with registers to hold data
Pipeline Datapath with Pipeline Registers

- **Pipeline registers**
  - IF/ID  ID/EX  EX/MEM  MEM/WB

- Two basic style of pipeline figures
  - Multiple clock cycle pipeline diagram
  - Single clock cycle pipeline diagram

Why no pipeline register at the beginning and the end of the pipeline?
Pipeline Version of the Single Cycle Datapath

[Diagram of pipeline stages: IF, ID, EX, MEM, WB]
• Example for `lw`

`lw $t1, 100($t2)`

**IF**

- `PC = PC + 4`
- `IF/ID <= PC+4`
- `IF/ID <= MEM[PC]`

**ID**

- `ID/EXE <= IF/ID (PC +4)`
- `ID/EXE <= $t1`
- `ID/EXE <= $t2`
- `ID/EXE <= signext(100)`
Illustration

**EXE**

\[ \text{EXE/MEM} (\text{address}) \leq \text{ID/EXE} (t2) + \text{ID/EXE} (\text{signext}(100)) \]
MEM/WB(Data) = Data[EXE/MEM(address)]

Register($t1) <= MEM/WB(Data)
Illustration of Store

- IF and ID stages are the same as the `lw`
- `sw $t1, 100($t2)`

\[
\text{EXE/MEM (address)} \leq \text{ID/EXE ($t2)} + \text{ID/EXE (signext(100))}
\]
\[
\text{EXE/MEM (Data)} \leq \text{ID/EXE ($t1)}
\]
Illustration

MEM

Data[ EXE/MEM(address) ] = $t1

WB
Notes to the Pipeline

• To **pass** something from an early pipe stage to a later pipe stage
  – Information must be placed in a pipeline register
  – Otherwise, information is lost when the next instruction enters that pipeline

• Each component of the pipeline datapath
  – Used only **within a single** pipeline stage
Corrected Datapath

• The **bug** in the previous illustration
  – Load: **register number** used during the WB stage has to be passed through the pipeline stages
Graphically representing Pipelines

- Multiple-clock-cycle pipeline diagram
- Single-clock-cycle pipeline diagram
• What happened in the pipeline datapath at a sequence of clock cycle
Traditional Multiple Clock Cycle Pipeline

- Use the **name** of each stage instead of **physical resources**
Single-clock-cycle Diagram

- What happened in the pipeline datapath at some clock cycle

Clock cycle 5
Outline

- Overview of pipelining
- A pipelined datapath
- Pipelined control
- Data hazards and forwarding
- Data hazards and stalls
- Branch hazards
- Exceptions in the pipelined datapath
- Advanced pipelining
Pipeline Control

- Add control to the pipeline datapath
  - No separated write control for pipeline registers and PC
  - Same control value as the one in ch. 5
**Original control for single clock cycle implementation**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Memto-Reg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Reorder into five groups according to the pipeline stages**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution/address calculation stage</th>
<th>Memory access stage</th>
<th>Write back stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>RegDst</td>
<td>ALUSrc</td>
<td>ALUOp1</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Implementing the Controls

- Extend the pipeline register to include control information
Full Datapath with Control
Outline

- Overview of pipelining
- A pipelined datapath
- Pipelined control
- Data hazards and forwarding
- Data hazards and stalls
- Branch hazards
- Exceptions in the pipelined datapath
- Advanced pipelining
Pipelined Dependences

<table>
<thead>
<tr>
<th>Time (in clock cycles)</th>
<th>Value of register $2$:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CC 1</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>

Program execution order (in instructions):
- `sub $2, $1, $3`
- `and $12, $2, $5`
- `or $13, $6, $2`
- `add $14, $2, $2`
- `sw $15, 100($2)`

Hazards: And, or instructions o.k. write/read at the same cycle
Pipeline Data Dependency Problem

- Two steps
  - 1. Hazard detection
  - 2. Software or hardware solution

- 1. Software solution
  - Insert "NOP" or "other independent instruction" between "sub" and "and"
  - Problem: this really slows us down!

- 2. Hardware solution: Forwarding
  - When is the data from "sub" actually produced
    - Available at the end of EX stage
  - When is the data actually needed by the and and or
    - At the beginning of the EX stage
  - Thus, simply forward the data as soon as it is available to any unit that need it
  - But which data?
Hazard Detection

- One of the source register number (in the pipeline register ID/EX) is equal to the register number in the EX/MEM or MEM/WB stage
  - 1a. \( \text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRs} = \$2 \)
  - 1b. \( \text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRt} \)
    - Example
      - \( \text{sub} \ \$2, \ \$1, \ \$3 \)
      - \( \text{and} \ \$12,\$2, \ \$5 \)
  - 2a. \( \text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRs} \)
  - 2b. \( \text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRt} \)
    - Example
      - \( \text{sub} \ \$2, \ \$1, \ \$3 \)
      - \( \text{and} \ \$12,\$2, \ \$5 \)
      - \( \text{or} \ \$13,\$6, \ \$2 \)
  - Condition: \( \text{EX/MEM.RegisterRd} \neq 0, \text{MEM/WB.RegisterRd} \neq 0 \)
    (why, see the next page)
Harzard Detection

1. This policy is inaccurate
   - Sometimes it would forward when unnecessary
2. Some instruction do not write register
   - Check if the RegWrite signal will be active:
     - Examining the WB control field of the pipeline register during the EX and MEM stages

2. Register $0 as the destination
   - \texttt{sll $0, $1, 2}
   - $0 cannot be forwarded ($0 cannot be changed)
   - Add \texttt{(!=0)} condition to correct it
Data to be Forwarded (from the Pipeline Registers)

<table>
<thead>
<tr>
<th>Time (in clock cycles)</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value of register $2$:</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10/–20</td>
<td>–20</td>
<td>–20</td>
<td>–20</td>
<td>–20</td>
</tr>
<tr>
<td>Value of EX/MEM:</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>–20</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Value of MEM/WB:</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>–20</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Program execution order (in instructions)

- sub $2$, $1$, $3$
- and $12$, $2$, $5$
- or $13$, $6$, $2$
- add $14$, $2$, $2$
- sw $15$, 100($2$)

ALU inputs can be from any pipeline registers (add MUX to select)
Datapath without Forwarding

a. No forwarding
Datapath with Forwarding

b. With forwarding
Forwarding Control

• For MUX control, see Fig. 6.31
• Three sources for each MUX
  – ID/EX: no forwarding, just from the register file (00)
  – EX/MEM: forwarded data from the prior ALU results (10)
  – MEM/WB: from data memory or an earlier ALU result (01)
Forwarding Control

- Forwarding control will be in the EX stage because ALU forwarding MUX is in this stage
  - Pass the operand register number from the ID stage via the ID/EX pipeline register
  - We already have \( rt \) field (bits 20-16), add \( rs \) to ID/EX pipeline register

- 1. EX hazard

\[
\begin{align*}
\text{If } & (\text{EX/MEM.RegWrite} \land (\text{EX/MEM.RegisterRd} \neq 0) \land (\text{EX/MEM.RegisterRd}=\text{ID/EX.RegisterRs})) \quad \text{ForwardA}=10 \\
\text{If } & (\text{EX/MEM.RegWrite} \land (\text{EX/MEM.RegisterRd} \neq 0) \land (\text{EX/MEM.RegisterRd}=\text{ID/EX.RegisterRt})) \quad \text{ForwardB}=10
\end{align*}
\]

Select forwarded data from **EX/MEM** stage
Forwarding Control

• 2. MEM hazard

If (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd=ID/EX.RegisterRs)) ForwardA=01

If (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd=ID/EX.RegisterRt)) ForwardB=01

Select forwarded data from MEM/WB stage
Potential More Complicated Data Hazard

- Between the results in the WB stage, MEM stage and ALU source
  \[
  \begin{align*}
  \text{add} & \quad $1, \quad $1, \quad $2 \\
  \text{add} & \quad $1, \quad $1, \quad $3 \\
  \text{add} & \quad $1, \quad $1, \quad $4 \\
  \end{align*}
  \]
  (vector summation)

- In above case, two forwarding cases will occur but the MEM hazard is incorrect one due to it is old one. Select the forwarded data from EX/MEM stage.

- Modified control for MEM hazard to prevent this
  \[
  \begin{align*}
  \text{If} \quad (\text{MEM/WB.RegWrite} & \quad \text{and} \quad (\text{MEM/WB.RegisterRD} \neq 0) \\
  \quad \text{and} \quad (\text{EX/MEM.RegisterRd} \neq \text{ID/EX.RegisterRs}) \\
  \quad \text{and} \quad (\text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRs})) \quad \text{ForwardA}=01 \\
  \end{align*}
  \]

  \[
  \begin{align*}
  \quad \text{If} \quad (\text{MEM/WB.RegWrite} & \quad \text{and} \quad (\text{MEM/WB.RegisterRD} \neq 0) \\
  \quad \text{and} \quad (\text{EX/MEM.RegisterRd} \neq \text{ID/EX.RegisterRt}) \\
  \quad \text{and} \quad (\text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRt})) \quad \text{ForwardB}=01 \\
  \end{align*}
  \]

Prevent to select forwarded data from MEM/WB stage
Datapath with Forwarding
Outline

- Overview of pipelining
- A pipelined datapath
- Pipelined control
- Data hazards and forwarding
- Data hazards and stalls
- Branch hazards
- Exceptions in the pipelined datapath
- Advanced pipelining
What if Data Hazard cannot be Solved by Forwarding

- **Case: load-use data hazard**

  - Stall (need a hazard detection unit)

  ![Diagram](image)

  Program execution order (in instructions)
  - `lw $2, 20($1)`
  - `and $4, $2, $5`
  - `or $8, $2, $6`
  - `add $9, $4, $2`
  - `slt $1, $6, $7`
Hazard Detection Unit

- Operates during the ID stage
  - Insert the stall between the load and its use

If (ID/EX.MemRead and
  ((ID/EX.RegisterRd=IF/ID.RegisterRs) or
  (ID/EX.RegisterRd=IF/ID.RegisterRt)))
  Stall the pipeline

IF (instruction is a load or
  (destination register of load match either
   source register of the instruction in the ID stage))
  Stall the pipeline
How to Stall the Pipeline

• IF and ID Stage
  – Preserving the register value
  – Instruction in the IF stage will continue to be read using the same PC
  – Register in the ID stage will continue to be read using the same instruction field in the ID/ID pipeline registers

• Other stages (EX, MEM, WB)
  – Insert “NOP” instruction: do nothing
  – That is: deasserting all nine control signals (set to 0)
    • No register or memories are written if the control are all 0
Stall the Pipeline

Program execution order (in instructions)

Iw $2, 20(1)

and becomes nop

add $4, $2, $5

These instruction are delayed

or $8, $2, $6

add $9, $4, $2
Pipeline with Forwarding and Hazard Detection
Outline

• Overview of pipelining
• A pipelined datapath
• Pipelined control
• Data hazards and forwarding
• Data hazards and stalls
• Branch hazards
• Exceptions in the pipelined datapath
• Advanced pipelining
Impact of the Pipeline on the Branch

Program execution order (in instructions)

40 beq $1, $3, 28
44 and $12, $2, $5
48 or $13, $6, $2
52 add $14, $2, $2
72 lw $4, 50($7)

Remember: branch taken on the MEM stage
Two Schemes for Solving Control Hazards

• Note: **No effective scheme** to solve control hazards like the forwarding on data hazards. So just use the simpler one. (What if stall on branch?)

1. Assume Branch Not Taken
   – Continually fetch new instruction down the sequential instruction stream
   – When this strategy will **gain**
     • If branches are untaken half the time and if it costs little to discard the instructions
   – If the prediction is wrong, discard the fetched instruction
   – Discard instruction
     • **Flush instructions** in the IF, ID/ and EX stages
Two Schemes for Solving Control Hazards

2. Reducing the delays of branches
   • Concept
     – move the branch execution earlier in the pipeline, then fewer instructions need flushed.
   • How MIPS designer do?
     – Make the common case fast
     – Many branches rely only on simple tests (equality or sign)
     – Such test can be done with a few gates without full ALU
Two Schemes for Solving Control Hazards

• Move the branch execution from MEM to the ID stage

2 steps in the ID stage
• 1. **compute the branch target** (PC + offset)
  – Move the branch adder from the EXE stage to the ID stage
• 2. **evaluate the branch decision**
  – **Equality test of two registers**
    • XOR their respective bits and OR all the results
  – **New** forwarding and hazard detection logic in the ID stage for equality test
    • Forward data from ALU/MEM or MEM/WB pipeline registers
    • Stall if a data hazard occurs

• Flush instructions in the IF stage
  – New control line: IF.Flush
  – Zero the instruction field of the IF/ID pipeline registers (=>NOP)
How This Works?
How This Works?
Dynamic Branch Prediction

• Branch prediction buffer or branch history table
  – A small memory indexed by the lower portion of the address of branch instruction. The memory contains a bit that say whether the branch was recently taken or not.

• Simplest one: 1-bit prediction
  
  e.g. loops: 9 times

  Mispredict: first and last iterations
  80% prediction accuracy
Dynamic Branch Prediction

• 2-bit Prediction Scheme
  – A prediction must be wrong twice before it is changed
  – Suitable for strongly favors taken or not taken
  – Mispredicted once

• Advanced ones
  – Correlating predictors
    • Global and local branch
  – Tournament predictor
    • Multiple predictions for each branch
A is the best. Use B C when A is impossible (data dependency).
B is preferred when the branch is taken with high probability such as a loop
C is scheduled from the not-taken fall through.
It should be O.K. to execute delay slot instruction for B C cases.
Outline

- Overview of pipelining
- A pipelined datapath
- Pipelined control
- Data hazards and forwarding
- Data hazards and stalls
- Branch hazards
- Exceptions in the pipelined datapath
- Advanced pipelining
Datapath and Control to Handle Exceptions
Handle Exceptions

• Another form of control hazards
• Flush the following instructions after the faulting instructions and restart it after exception is handled
  – IF.Flush
  – ID.Flush
  – EX.Flush
• Save the offending instructions into EPC (PC + 4)
• Multiple exceptions can occur simultaneously for multiple instructions executed in the pipeline
  – Solution: to prioritize the exceptions
    • Earlier instruction in interrupted first. Hardware malfunction is handled as soon as possible
  – Precise interrupts (exceptions)
  – Imprecise interrupts (exceptions)
How it Works?
How it Works?
Outline

• Overview of pipelining
• A pipelined datapath
• Pipelined control
• Data hazards and forwarding
• Data hazards and stalls
• Branch hazards
• Exceptions in the pipelined datapath
• Advanced pipelining
ILP: Instruction Level Parallelism

- Reference for more details
  - Computer architecture: A quantitative approach
- Two methods to increase ILP
  - **Increase the pipeline depth**
    - More operations being overlapped
    - Pipeline speedup \( \alpha \) pipeline depth
    - 8 or more pipeline stages
    - To get the speedup, rebalance the remaining steps
    - Performance is potentially greater due to shorter the clock cycle
  - **Multiple issue**
    - Issue 3 to 8 instructions in every clock cycle
    - **Static** multiple issue: determined at compile time
    - **Dynamic** multiple issue: determined during execution
    - Two problems
      - How to package instruction into issue slots (by compiler or hardware)
      - Dealing with data and control hazards (by compiler or hardware)
Speculation: Find and Exploit more ILP

• Speculation
  – An approach that allows the compiler or the processor to “guess” the outcome of an instruction to remove it as a dependence in executing other instructions
    • E.g. branch, store before load

• How it works
  – Compiler or processor use speculation to
    • reorder instructions,
    • move an instruction across a branch or
    • a load across a store

• Mechanism
  – A method to check if guess right and a method to back out the effects
    • Difficulty: what if guess wrong (back-out capability)
Speculation: Find and Exploit more ILP

Recovery mechanism for incorrect speculation

• **Software approach**
  – Compiler inserts additional instructions to
    • **Check the accuracy** of the speculation
    • Provide a **fix-up** routine

• **Hardware approach**
  – Buffer the speculative results until no longer speculative
    • If correct, complete the instruction (write results to registers)
    • If incorrect, flush the buffer and reexecute the correct one
Speculation: Find and Exploit more ILP

Other possible problem: **Exception in speculative instruction**

- Speculating on certain instructions may introduce exceptions that were formerly not present
  - E.g. If executing “load” in speculative, but the address is illegal, then “exception” that should not happen will occur. (Exception should occur when load is not speculative)

- Compiler-based speculation
  - Allow such exceptions **ignored** until they should occur

- Hardware-based speculation
  - **Buffer** such exceptions until no longer speculative, then raise the exception
Static Multiple Issue

• Compiler assist **packaging instruction and handling data hazards**

• Issue packet
  – As **one large instruction with multiple operations**
  – VLIW: very long instruction word
  – EPIC: Explicitly Parallel Instruction Computer (IA-64)

• Variation: how compiler handle hazards
  – 1. Compilers handle **all** hazards, schedule code, and insert code
  – 2. Compiler handle all dependences **within** an instruction, and hardware detects data hazards and generates stalls **between** two issue packets
Two-Issue MIPS Processor

- Static two-issue pipeline (64-bits IF and ID)
  
  ALU or branch  IF  ID  EX  MEM  WB
  Load or store  IF  ID  EX  MEM  WB
  ALU or branch  IF  ID  EXE  MEM  WB
  Load or store  IF  ID  EXE  MEM  WB

- Extra hardware
  - Register file
    - 2 read for ALU, 2 read for store, one write for ALU, one write for load
    - Separated adder for address calculation of data transfers

- Performance
  - Improve up to a factor of 2 (upper bound)
  - In reality, it depends on how you schedule the instructions. Compiler takes on this role.
Two Issue MIPS Processor
Multiple Issue Code Scheduling

Original: add scalar $s2$ to array

Loop:

- `lw $t0, 0($s1)` #$t0 = array element
- `addu $t0, $t0, $s2` # add scalar in $s2
- `sw $t0, 0($s1)` # store result
- `addi $s1, $s1, -4` # decrement pointer
- `bne $s1, $zero, loop` # branch $s1 !=0

Note. The result of a load cannot be used on the next clock cycle due to load-use dependency.

Scheduled code for two-issue MIPS (4 cycles)

<table>
<thead>
<tr>
<th>ALU or branch inst.</th>
<th>Data transfer inst.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lw</code> $t0, 0($s1)</td>
<td></td>
</tr>
<tr>
<td><code>addi $s1, $s1, -4</code></td>
<td></td>
</tr>
<tr>
<td><code>addu $t0, $t0, $s2</code></td>
<td></td>
</tr>
<tr>
<td><code>bne $s1, $zero, loop</code></td>
<td></td>
</tr>
</tbody>
</table>
Loop Unrolling for 2-Issue MIPS

- To get more performance from loops: loop unrolling
- Assume the loop index is multiple of four
- Unroll four loop: register renaming to remove antidependences

Loop:

```
addi    $s1,$s1,  -16
lw      $t0,0($s1)
lw      $t1,12($s1)
addu    $t0,$t0,  $s2
lw      $t2,8($s1)
addu    $t1,$t1,  $s2
lw      $t3,8($s1)
addu    $t2,$t2,  $s2
sw      $t0,16($s1)
addu    $t3,$t3,  $s2
sw      $t1,12($s1)
sw      $t1,8($s1)
bne     $s1,$zero,loop
sw      $t3,4($s1)
```

```
for (i = 0; i < 16; i=i+4) {
    array[i] = array[i]+scalar;
    array[i+1] = array[i+1]+scalar;
    array[i+2] = array[i+2]+scalar;
    array[i+3] = array[i+3]+scalar;
}
```
Intel IA-64 Architecture

- EPIC
  - Explicitly Parallel Instruction Computer
- VLIW benefits but with more flexibility
  - Instruction groups
    - All instructions in the group without no register data dependences
    - Can be executed in parallel if sufficient hardware resources
  - Bundles
    - IA-64 instruction are encoded in bundles, 128-bit wide with 5-bit template field and 3 instructions
    - Template field specifies which five execution unit are used
- Prediction
  - Make instruction dependent on predicates rather than on branches
  - Loop unrolling works well for loop branch but not if-than-else branch
  - Something like conditional execution
Dynamic Multiple Issue Processors

• Superscalar
  – Instruction issue in order
  – 0, 1 or more instructions can issue in a give clock cycle

• To achieve good performance
  – Needs compiler to schedule instructions
  – More important: **hardware guarantees** instructions are executed correctly whether scheduled or not

• Extension: dynamic pipeline scheduling
  – Hardware support to reorder the execution order to avoid stalls

```
lw    $t0, 20($s2)
addu  $t1, $t0, $t2
sub   $s4, $s4, $t3
slt   $t5, $s4, 20
```

**sub will be executed**

In superscalar processors
Dynamic Pipeline Scheduling

- **Pipeline**
  - Instruction fetch and issue unit
  - Multiple function units
    - Includes a *reservation stations* that hold operations and operands
  - **Commit unit**
    - Decides when it is safe to release the results
    - Includes a *reorder buffer* to buffer the result and to supply operands like forwarding

- **Why dynamic scheduling (Can compiler handle every data dependences?)**
  - Not all stalls are predictable, e.g. cache miss
  - Dynamic branch prediction
  - Pipeline latency and issue width change from one to another implementation
Dynamic Pipeline Scheduling

In-order issue

Reservation station
Reservation station
Reservation station
Reservation station

Functional units
Integer
Integer
Floating point
Load/Store

Commit unit

Out-of-order execute

In-order commit
Summary

- Deeply pipelined
- Multiple issue with deep pipeline (Section 6.10)
- Pipelined
- Multiple-issue pipelined (Section 6.9)
- Multicycle (Section 5.5)
- Single-cycle (Section 5.4)

Instructions per clock (IPC = 1/CPI)