Pipelining

Politecnico di Milano
Seminar Room A. Alario
16 November, 2015

Antonio R. Miele
Marco D. Santambrogio
Politecnico di Milano
Outline

• Processors and Instruction Sets

• Review of pipelining

• MIPS
  – Reduced Instruction Set of MIPS™ Processor
  – Implementation of MIPS Processor Pipeline
  – The Problem of Pipeline Hazards
  – Performance Issues in Pipelining
Main Characteristics of MIPS™ Architecture

- **RISC (Reduced Instruction Set Computer) Architecture**
  Based on the concept of executing only simple instructions in a reduced basic cycle to optimize the performance of CISC CPUs.

- **LOAD/STORE Architecture**
  ALU operands come from the CPU general purpose registers and they cannot directly come from the memory. Dedicated instructions are necessary to:
  - load data from memory to registers
  - store data from registers to memory

- **Pipeline Architecture:**
  Performance optimization technique based on the overlapping of the execution of multiple instructions derived from a sequential execution flow.
A Typical RISC ISA

- 32-bit fixed format instruction (3 formats)
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store: base + displacement
  - no indirection
- Simple branch conditions
- Delayed branch

- Example: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3
Approaching an ISA

• Instruction Set Architecture
  – Defines set of operations, instruction format, hardware supported data types, named storage, addressing modes, sequencing

• Meaning of each instruction is described by RTL on architected registers and memory

• Given technology constraints assemble adequate datapath
  – Architected storage mapped to actual storage
  – Function units to do all the required operations
  – Possible additional storage (eg. MAR, MBR, …)
  – Interconnect to move information among regs and FUs

• Map each instruction to sequence of RTLs
• Collate sequences into symbolic controller state transition diagram (STD)
• Implement controller
## Example: MIPS

### Register-Register

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>12</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td>Opx</td>
<td></td>
</tr>
</tbody>
</table>

### Register-Immediate

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>12</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
</tbody>
</table>

### Branch

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2/Opx</td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
</tbody>
</table>

### Jump / Call

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
</tr>
<tr>
<td>Op</td>
<td></td>
<td>target</td>
</tr>
</tbody>
</table>
Datapath vs Control

- **Datapath**: Storage, FU, interconnect sufficient to perform the desired functions
  - Inputs are Control Points
  - Outputs are signals

- **Controller**: State machine to orchestrate operation on the data path
  - Based on desired function and signals
Datapath vs Control

Data path
- ALU
- Registri

Control Unit
- PSW
- IR
- PC

CPU

Memory

Control BUS
- Data BUS
- Address BUS

Data
- Instruction
- Control
The code...

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0789</td>
<td>load</td>
<td>R02,4000</td>
<td></td>
</tr>
<tr>
<td>0790</td>
<td>load</td>
<td>R03,4004</td>
<td></td>
</tr>
<tr>
<td>0791</td>
<td>add</td>
<td>R01,R02,R03</td>
<td></td>
</tr>
<tr>
<td>0792</td>
<td>load</td>
<td>R02,4008</td>
<td></td>
</tr>
<tr>
<td>0793</td>
<td>add</td>
<td>R01,R01,R02</td>
<td></td>
</tr>
<tr>
<td>0794</td>
<td>store</td>
<td>R01,4000</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Read Instruction 0789

CPU

Data path

Control Unit

Reading

ALU

PSW

IR

PC

R00

R01

R02

R03

R04

R05

Registri

Memory

Data path

Instruction

0789 load R02,4000
0790 load R03,4004
0791 add R01,R02,R03
0792 load R02,4008
0793 add R01,R01,R02
0794 store R01,4000

Data

0789 1492
0790 1918
0792 2006

Memory

0789 load R02,4000
0790 load R03,4004
0791 add R01,R02,R03
0792 load R02,4008
0793 add R01,R01,R02
0794 store R01,4000

PSW
Exe Instruction 0789

**Data path**

- **ALU**
- **Registries**
  - R00
  - R01
  - R02
  - R03
  - R04
  - R05
- **IR**
- **PC**

**Control Unit**

- **Reading**
- **PSW**

**Memory**

- **Instruction**
  - 0789 load R02,4000
  - 0790 load R03,4004
  - 0791 add R01,R02,R03
  - 0792 load R02,4008
  - 0793 add R01,R01,R02
  - 0794 store R01,4000
- **Data**
  - 4000 1492
  - 4004 1918
  - 4008 2006

**CPU**

- **CPU**
  - **Control Unit**
    - **Contro bus**
    - **Data Bus**
    - **Address Bus**
Read instruction 0790

CPU

Data path

Control Unit

Reading

load R02,4000

PSW

IR

PC

0790 +1

0789 load R02,4000
0790 load R03,4004
0791 add R01,R02,R03
0792 load R02,4008
0793 add R01,R01,R02
0794 store R01,4000

Memory

Control bus

Data Bus

Address Bus

Data

Instruction

... ... ...
0789 load R02,4000
0790 load R03,4004
0791 add R01,R02,R03
0792 load R02,4008
0793 add R01,R01,R02
0794 store R01,4000

... ... ...

... ... ...
4000 1492
4004 1918
4008 2006
... ... ...

Data path

PSW

ALU

Registries

... R05...

R04

R03

R02

R01

R00

...
Exe Instruction 0790

CPU

Data path

Control Unit

Reading

ALU

PSW

IR

PC

R00

R01

R02

R03

R04

R05

...Registri

0791

load R03,4004

4004

Memory

Instructor

Data

Data Bus

Address Bus

Contro bus

0789 load R02,4000
0790 load R03,4004
0791 add R01,R02,R03
0792 load R02,4008
0793 add R01,R01,R02
0794 store R01,4000

... ... ... ...

4000 1492
4004 1918
4008 2006

... ... ... ...
Read Instruction 0791

**CPU**

- **Data path**
  - ALU
  - PSW
  - IR
  - PC
  - R00
  - R01
  - R02
  - R03
  - R04
  - R05

- **Control Unit**
  - Reading

**Memory**

- **Instruction**
  - 0789 load R02,4000
  - 0790 load R03,4004
  - 0791 add R03,R02,R03
  - 0792 load R02,4008
  - 0793 add R01,R01,R02
  - 0794 store R01,4000

- **Data**
  - 4000 1492
  - 4004 1918
  - 4008 2006

**Data path**

- Data Bus
- Address Bus
- Control bus

**Registri**

- R00:
  - 1492

- R01:
  - 1918

- R3:
  - 0792 +1
Exe Instruction 0791

```
0791
load	R02,4000
0790
load	R03,4004
0791
add	R01,R02,R03
0792
load	R02,4008
0793
add	R01,R01,R02
0794
store	R01,4000
```

CPU

Data path

Control Unit

Memory

Instruction

Data

0789 load R02,4000
0790 load R03,4004
0791 add R01,R02,R03
0792 load R02,4008
0793 add R01,R01,R02
0794 store R01,4000

0400 1492
0404 1918
0408 2006
Read Instruction 0792

CPU

Data path

Control Unit

Reading

ALU

PSW

IR

add R01,R02,R03

PC

0792 +1

Registri

... 

R05 

R04 

R03 

R02 

R01 

R00

3410

1492

1918

... 

Memory

Instruction

0789 load R02,4000
0790 load R03,4004
0791 add R01,R02,R03
0792 load R02,4008
0793 add R01,R01,R02
0794 store R01,4000

Data

... 

4000 1492
4004 1918
4008 2006

... 

Contro bus

Data Bus

Address Bus
Exe Instruction 0792

0793 load R02,4000
0790 load R03,4004
0791 add R01,R02,R03
0792 load R02,4008
0793 add R01,R01,R02
0794 store R01,4000

Memory

Instruction
0789 load R02,4000
0790 load R03,4004
0791 add R01,R02,R03
0792 load R02,4008
0793 add R01,R01,R02
0794 store R01,4000

Data
4000 1492
4004 1918
4008 2006

CPU

Data path
Control Unit
Reading
PSW
IR
PC

Data Bus

Address Bus

Contro bus
Read Instruction 0793

Memory

Instruction

0789 load R02,4000
0790 load R03,4004
0791 add R01,R02,R03
0792 load R02,4008
0793 add R01,R01,R02
0794 store R01,4000

Data

4000 1492
4004 1918
4008 2006

CPU

Data path

Control Unit

Reading

ALU

PSW

IR

load R02,4008

0793 +1

PC

Data Bus

Address Bus

Contro bus

Registri

R00

R01

R02

R03

R04

R05

...
**Exe Instruction 0793**

### Data path
- **5416 ALU**
- **Registri**
  - R00
  - R01: 3410
  - R02: 2006
  - R03: 1918
  - R04
  - R05
- **Data Bus**
- **Address Bus**
- **Control Bus**

### Control Unit
- **IR**: add R01,R01,R02
- **PC**: 0794
- **PSW**

### Memory
- **Data**
  - 0789 load R02,4000
  - 0790 load R03,4004
  - 0791 add R01,R02,R03
  - 0792 load R02,4008
  - 0793 add R01,R01,R02
  - 0794 store R01,4000

- **InstrucBon**
  - 0789
  - 0790
  - 0791
  - 0792
  - 0793
  - 0794

- **Data Bus**
- **Address Bus**
- **Control Bus**
Read Instruction 0794

```
0789 load R02,4000
0790 load R03,4004
0791 add R01,R02,R03
0792 load R02,4008
0793 add R01,R01,R02
0794 store R01,4000
```

---

CPU

- **Data path**
  - **ALU**
  - **PSW**
  - **Registries**
    - R00
    - R01
    - R02
    - R03
    - R04
    - R05
  - **IR**
  - **PC**
  - **add R01,R01,R02**

Memory

- **Instruction**
  - 0789 load R02,4000
  - 0790 load R03,4004
  - 0791 add R01,R02,R03
  - 0792 load R02,4008
  - 0793 add R01,R01,R02
  - 0794 store R01,4000

- **Data**
  - 4000 1492
  - 4004 1918
  - 4008 2006

**Data Bus**

**Address Bus**

**Control Bus**

**PSW**

**Data path**

**Control Unit**

**Reading**
Exe Instruction 0794

CPU

Data path

Control Unit

IR

store R01,4000

PC

0795

ALU

PSW

Memory

Data path

Control Unit

IR

store R01,4000

PC

0795

ALU

PSW

Memory

Data path

Control Unit

IR

store R01,4000

PC

0795

ALU

PSW

Memory

Data path

Control Unit

IR

store R01,4000

PC

0795

ALU

PSW

Memory

Data path

Control Unit

IR

store R01,4000

PC

0795

ALU

PSW

Memory

Data path

Control Unit

IR

store R01,4000

PC

0795

ALU

PSW

Memory

Data path

Control Unit

IR

store R01,4000

PC

0795

ALU

PSW

Memory
Reduced Instruction Set of MIPS Processor

• ALU instructions:
  - add $s1, $s2, $s3  # $s1 ← $s2 + $s3
  - addi $s1, $s1, 4  # $s1 ← $s1 + 4

• Load/store instructions:
  - lw $s1, offset ($s2)  # $s1 ← M[$s2+offset]
  - sw $s1, offset ($s2)  M[$s2+offset] ← $s1

• Branch instructions to control the control flow of the program:
  - Conditional branches: the branch is taken only if the condition is satisfied. Examples: beq (branch on equal) and bne (branch on not equal)
    - beq $s1, $s2, L1  # go to L1 if ($s1 == $s2)
    - bne $s1, $s2, L1  # go to L1 if ($s1 != $s2)
  - Unconditional jumps: the branch is always taken.
    Examples: j (jump) and jr (jump register)
    - j  L1  # go to L1
    - jr $s1  # go to add. contained in $s1
Execution of MIPS Instructions

Every instruction in the MIPS subset can be implemented in at most 5 clock cycles as follows:

• Instruction Fetch Cycle:
  – Send the content of Program Counter register to Instruction Memory and fetch the current instruction from Instruction Memory. Update the PC to the next sequential address by adding 4 to the PC (since each instruction is 4 bytes).

• Instruction Decode and Register Read Cycle
  – Decode the current instruction (fixed-field decoding) and read from the Register File of one or two registers corresponding to the registers specified in the instruction fields.
  – Sign-extension of the offset field of the instruction in case it is needed.
Execution of MIPS instructions

• Execution Cycle
The ALU operates on the operands prepared in the previous cycle depending on the instruction type:
  – Register-Register ALU Instructions:
    • ALU executes the specified operation on the operands read from the RF
  – Register-Immediate ALU Instructions:
    • ALU executes the specified operation on the first operand read from the RF and the sign-extended immediate operand
  – Memory Reference:
    • ALU adds the base register and the offset to calculate the effective address.
  – Conditional branches:
    • Compare the two registers read from RF and compute the possible branch target address by adding the sign-extended offset to the incremented PC.
Execution of MIPS instructions

• Memory Access (ME)
  – Load instructions require a read access to the Data Memory using the effective address
  – Store instructions require a write access to the Data Memory using the effective address to write the data from the source register read from the RF
  – Conditional branches can update the content of the PC with the branch target address, if the conditional test yielded true.

• Write-Back Cycle (WB)
  – Load instructions write the data read from memory in the destination register of the RF
  – ALU instructions write the ALU results into the destination register of the RF.
Execution of MIPS Instructions

**ALU Instructions:** \( \text{op } \$x, \$y, \$z \)

<table>
<thead>
<tr>
<th>Instr. Fetch &amp; PC Increm.</th>
<th>Read of Source Regs. $y and $z</th>
<th>ALU OP ($y \text{ op } $z)</th>
<th>Write Back of Destinat. Reg. $x</th>
</tr>
</thead>
</table>

**Load Instructions:** \( \text{lw } \$x, \text{offset(}\$y\text{)} \)

<table>
<thead>
<tr>
<th>Instr. Fetch &amp; PC Increm.</th>
<th>Read of Base Reg. $y</th>
<th>ALU Op. ($y+\text{offset})</th>
<th>Read Mem. \text{M(}$y+\text{offset})</th>
<th>Write Back of Destinat. Reg. $x</th>
</tr>
</thead>
</table>

**Store Instructions:** \( \text{sw } \$x, \text{offset(}\$y\text{)} \)

<table>
<thead>
<tr>
<th>Instr. Fetch &amp; PC Increm.</th>
<th>Read of Base Reg. $y &amp; Source $x</th>
<th>ALU Op. ($y+\text{offset})</th>
<th>Write Mem. \text{M(}$y+\text{offset})</th>
</tr>
</thead>
</table>

**Conditional Branch:** \( \text{beq } \$x, \$y, \text{offset} \)

<table>
<thead>
<tr>
<th>Instr. Fetch &amp; PC Increm.</th>
<th>Read of Source Regs. $x and $y</th>
<th>ALU Op. ($x-$y) &amp; (PC+4+\text{offset})</th>
<th>Write PC</th>
</tr>
</thead>
</table>
MIPS Data path

IR \leftarrow \text{mem}[PC]

PC \leftarrow PC + 4

\text{Reg}[IR_{rd}] \leftarrow \text{Reg}[IR_{rs}] \text{ op}_{IRop} \text{Reg}[IR_{rt}]
# Instructions Latency

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Instruct. Mem.</th>
<th>Register Read</th>
<th>ALU Op.</th>
<th>Data Memory</th>
<th>Write Back</th>
<th>Total Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU Instr.</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>6 ns</td>
</tr>
<tr>
<td>Load</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>8 ns</td>
</tr>
<tr>
<td>Store</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>7 ns</td>
</tr>
<tr>
<td>Cond. Branch</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>5 ns</td>
</tr>
<tr>
<td>Jump</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2 ns</td>
</tr>
</tbody>
</table>
Single-cycle Implementation of MIPS

• The length of the clock cycle is defined by the critical path given by the load instruction: $T = 8 \text{ ns (} f = 125 \text{ MHz)}$.

• We assume each instruction is executed in a single clock cycle
  – Each module must be used once in a clock cycle
  – The modules used more than once in a cycle must be duplicated.

• We need an Instruction Memory separated from the Data Memory.

• Some modules must be duplicated, while other modules must be shared from different instruction flows

• To share a module between two different instructions, we need a multiplexer to enable multiple inputs to a module and select one of different inputs based on the configuration of control lines.
Implementation of MIPS data path with Control Unit
Multi-cycle Implementation

- The instruction execution is distributed on multiple cycles (5 cycles for MIPS)
- The basic cycle is smaller (2 ns $\rightarrow$ instruction latency = 10 ns)
- Implementation of multi-cycle CPU:
  - Each phase of the instruction execution requires a clock cycle
  - Each module can be used more than once per instruction in different clock cycles: possible sharing of modules
  - We need internal registers to store the values to be used in the next clock cycles.
Pipelining

- Performance optimization technique based on the overlap of the execution of multiple instructions deriving from a sequential execution flow.
- Pipelining exploits the parallelism among instructions in a sequential instruction stream.
- Basic idea:
  The execution of an instruction is divided into different phases (pipelines stages), requiring a fraction of the time necessary to complete the instruction.
- The stages are connected one to the next to form the pipeline: instructions enter in the pipeline at one end, progress through the stages, and exit from the other end, as in an assembly line.
Pipelining

- Advantage: technique transparent for the programmer.
- Technique similar to a assembly line: a new car exits from the assembly line in the time necessary to complete one of the phases.
- An assembly line does not reduce the time necessary to complete a car, but increases the number of cars produced simultaneously and the frequency to complete cars.
Sequential vs. Pipelining Execution

IF | ID | EX | MEM | WB
---|----|----|-----|-----

10 ns

IF | ID | EX | MEM | WB
---|----|----|-----|-----

2 ns

I1

I2

I3

I4

I5

Time

2 ns
Pipelining

• The time to advance the instruction of one stage in the pipeline corresponds to a clock cycle.
• The pipeline stages must be synchronized: the duration of a clock cycle is defined by the time requested by the slower stage of the pipeline (i.e. 2 ns).
• The goal is to balance the length of each pipeline stage
• If the stages are perfectly balanced, the ideal speedup due to pipelining is equal to the number of pipeline stages.
Performance Improvement

• Ideal case (asymptotically): If we consider the single-cycle unpipelined CPU1 with clock cycle of 8 ns and the pipelined CPU2 with 5 stages of 2 ns:
  – The latency (total execution time) of each instruction is worsened: from 8 ns to 10 ns
  – The throughput (number of instructions completed in the time unit) is improved of 4 times: (1 instruction completed each 8 ns) vs. (1 instruction completed each 2 ns)
Performance Improvement

• Ideal case (asymptotically): If we consider the multi-cycle unpipelined CPU3 composed of 5 cycles of 2 ns and the pipelined CPU2 with 5 stages of 2 ns:
  – The latency (total execution time) of each instruction is not varied (10 ns)
  – The throughput (number of instructions completed in the time unit) is improved of 5 times: (1 instruction completed every 10 ns) vs. (1 instruction completed every 2 ns)
## Pipeline Execution of MIPS Instructions

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>ME</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td>Instruction Decode</td>
<td>Execution</td>
<td>Memory Access</td>
<td>Write Back</td>
</tr>
</tbody>
</table>

### ALU Instructions: \texttt{op } $x, y, z$

- Instr. Fetch & PC Increm.
- Read of Source Regs. $y$ and $z$
- ALU Op. ($y \text{ op } z$)
- Write Back Destinat. Reg. $x$

### Load Instructions: \texttt{lw } $x, \text{ offset($y$)}$

- Instr. Fetch & PC Increm.
- Read of Base Reg. $y$
- ALU Op. ($y+\text{offset}$)
- Read Mem. M($y+\text{offset}$)
- Write Back Destinat. Reg. $x$

### Store Instructions: \texttt{sw } $x, \text{ offset($y$)}$

- Instr. Fetch & PC Increm.
- Read of Base Reg. $y$ & Source $x$
- ALU Op. ($y+\text{offset}$)
- Write Mem. M($y+\text{offset}$)

### Conditional Branches: \texttt{beq } $x, y, \text{offset}$

- Instr. Fetch & PC Increm.
- Read of Source Regs. $x$ and $y$
- ALU Op. ($x-y$) & (PC+4+\text{offset})
- Write PC
Implementation of MIPS pipeline
Visualizing Pipelining

Time (clock cycles)

Instr. Order

Cycle 1: Ifetch → Reg → ALU
Cycle 2: DMem
Cycle 3: Ifetch → Reg → ALU
Cycle 4: DMem
Cycle 5: Ifetch → Reg → ALU
Cycle 6: DMem
Cycle 7: Ifetch → Reg → ALU
Note: Optimized Pipeline

• Register File used in 2 stages: Read access during ID and write access during WB
  – What happens if read and write refer to the same register in the same clock cycle?
    • It is necessary to insert one stall

• Optimized Pipeline: the RF read occurs in the second half of clock cycle and the RF write in the first half of clock cycle
  – What happens if read and write refer to the same register in the same clock cycle?
    • It is not necessary to insert one stall
Note: Optimized Pipeline

- Register File used in 2 stages: Read access during ID and write during WB.
  - What happens if read and write refer to the same register in the same clock cycle?
  - It is necessary to insert one stall.

- Optimized Pipeline: the RF read occurs in the second half of the clock cycle and the RF write in the first half of the clock cycle.
  - What happens if read and write refer to the same register in the same clock cycle?
  - It is not necessary to insert one stall.

From now on, this is the Pipeline we are going to use.
5 Steps of MIPS Datapath

Instruction Fetch

Instr. Decode
Reg. Fetch

Execute
Addr. Calc

Memory Access

Write Back

IR <= mem[PC];
PC <= PC + 4

A <= Reg[IR\_rs];
B <= Reg[IR\_rt]

rslt <= A op\_IRop B

WB <= rslt

Reg[IR\_rd] <= WB

Data stationary control
local decode for each instruction phase / pipeline stage
The Problem of Hazards

ALL I WANTED WAS SOME FREAKING

HAZARD RECOGNITION
The Problem of Hazards

- A hazard is created whenever there is a dependence between instructions, and instructions are close enough that the overlap caused by pipelining would change the order of access to the operands involved in the dependence.

- Hazards prevent the next instruction in the pipeline from executing during its designated clock cycle.

- Hazards reduce the performance from the ideal speedup gained by pipelining.
Three Classes of Hazards

- **Structural Hazards**: Attempt to use the same resource from different instructions simultaneously
  - Example: Single memory for instructions and data
- **Data Hazards**: Attempt to use a result before it is ready
  - Example: Instruction depending on a result of a previous instruction still in the pipeline
- **Control Hazards**: Attempt to make a decision on the next instruction to execute before the condition is evaluated
  - Example: Conditional branch execution
Structural Hazards

- No structural hazards in MIPS architecture:
  - Instruction Memory separated from Data Memory
  - Register File used in the same clock cycle: Read access by an instruction and write access by another instruction
Data Hazards

• If the instructions executed in the pipeline are dependent, data hazards can arise when instructions are too close

• Example:

  sub $2, $1, $3 # Reg. $2 written by sub and $12, $2, $5 # 1° operand ($2) depends on sub or $13, $6, $2 # 2° operand ($2) depend on sub add $14, $2, $2 # 1° ($2) & 2° ($2) depend on sub sw $15, 100($2) # Base reg. ($2) depends on sub
Data Hazards in the Optimized Pipeline: Example

It is necessary to insert two stalls

sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
Type of Data Hazard

• **Read After Write (RAW)**
  Instr\textsubscript{J} tries to read operand before Instr\textsubscript{I} writes it

\[
\begin{align*}
\text{I}: & \quad \text{add } r1, r2, r3 \\
\text{J}: & \quad \text{sub } r4, r1, r3
\end{align*}
\]

• Caused by a “**Dependence**” (in compiler nomenclature). This hazard results from an actual need for communication.
Data Hazards: Possible Solutions

• Compilation Techniques:
  – Insertion of nop (no operation) instructions
  – Instructions Scheduling to avoid that correlating instructions are too close
    • The compiler tries to insert independent instructions among correlating instructions
    • When the compiler does not find independent instructions, it insert nops.

• Hardware Techniques:
  – Insertion of “bubbles” or stalls in the pipeline
  – Data Forwarding or Bypassing
Just an example...

I need to insert 2 bubbles or 2 stalls
Insertion of nops

sub $2, $1, $3
nop
nop
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
Insertion of bubbles and stalls

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

Contenuto di $2 10 10 10 10 -20 -20 -20 -20 -20 -20 -20 -20
Scheduling: Example

sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15,100($2)
add $4, $10, $11
and $7, $8, $9
lw $16, 100($18)
lw $17, 200($19)

sub $2, $1, $3
add $4, $10, $11
and $7, $8, $9
lw $16, 100($18)
lw $17, 200($19)
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15,100($2)
Forwarding

• Data forwarding uses temporary results stored in the pipeline registers instead of waiting for the write back of results in the RF.

• We need to add multiplexers at the inputs of ALU to fetch inputs from pipeline registers to avoid the insertion of stalls in the pipeline.
Forwarding: Example

sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
Implementation of MIPS with Forwarding Unit
Data Hazards: Load/Use Hazard

L1: \text{lw} \ $s0, 4($t1) \# \ $s0 \leftarrow M [4 + \$t1]
L2: \text{add} \ $s5, \$s0, \$s1 \# \text{1° operand depends from L1}
Data Hazards: Load/Use Hazard

• With forwarding using the MEM/EX path: 1 stall

```
lw  $s0, 4($t1)  IF  ID  EX  MEM  WB
add  $s5,$s0,$s1  IF  ID  EX  MEM  WB
```
Data Hazards: Load/Store

L1: `lw $s0, 4($t1)` # $s0 $\leftarrow M[4 + t1]
L2: `sw $s0, 4($t2)` # $M[4 + t2] $\leftarrow $s0

Without forwarding: 3 stalls
Data Hazards: Load/Store

- Forwarding: Stall = 0
- We need a forwarding path to bring the load result from the memory (in MEM/WB) to the memory’s input for the store.

```
$0 lw $s0, 4($t1)
$0 sw $s0, 4($t2)
```
Implementation of MIPS with Forwarding Unit

- EX/EX path
- MEM/EX path
- MEM/ID path
- MEM/MEM path
Data Hazards

• Data hazards analyzed up to now are:
  – RAW (READ AFTER WRITE) hazards: instruction n+1 tries to read a source register before the previous instruction n has written it in the RF.
  – Example:
    
    \begin{align*}
    \text{add } & r1, \ r2, \ r3 \\
    \text{sub } & r4, \ r1, \ r5
    \end{align*}

• By using forwarding, it is always possible to solve this conflict without introducing stalls, except for the load/use hazards where it is necessary to add one stall
Data Hazards

• Other types of data hazards in the pipeline:
  – WAW (WRITE AFTER WRITE)
  – WAR (WRITE AFTER READ)
Data Hazards: WAW

• Instruction n+1 tries to write a destination operand before it has been written by the previous instruction n
  ⇒ write operations executed in the wrong order
• This type of hazards could not occur in the MIPS pipeline because all the register write operations occur in the WB stage
Data Hazards: WAW

• Example: If we assume the register write in the ALU instructions occurs in the fourth stage and that load instructions require two stages (MEM1 and MEM2) to access the data memory, we can have:

```
lw  $r1, 0($r2)  IF  ID  EX  MEM1  MEM2  WB
add $r1,$r2,$r3  IF  ID  EX  WB
```
Data Hazards: WAW

• Example: If we assume the floating point ALU operations require a multi-cycle execution, we can have:

```
mul $f6, $f2, $f2
```

```
add $f6, $f2, $f2
```
WAW Data Hazards

- **Write After Write (WAW)**
  Instr$_j$ writes operand before Instr$_i$ writes it.

- Called an “output dependence” by compiler writers
  This also results from the reuse of name “r1”.

- Can’t happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5

- Will see WAR and WAW in more complicated pipes

```plaintext
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```
Data Hazards: WAR

• Instruction n+1 tries to write a destination operand before it has been read from the previous instruction n
  ⇒ instruction n reads the wrong value.

• This type of hazards could not occur in the MIPS pipeline because the operand read operations occur in the ID stage and the write operations in the WB stage.

• As before, if we assume the register write in the ALU instructions occurs in the fourth stage and that we need two stages to access the data memory, some instructions could read operands too late in the pipeline.
Data Hazards: WAR

• Example: Instruction sw reads $r2 in the second half of MEM2 stage and instruction add writes $r2 in the first half of WB stage $\Rightarrow$ sw reads the new value of $r2
WAR Data Hazards

• **Write After Read (WAR)**
  Instr\(_j\) writes operand **before** Instr\(_i\) reads it

  \[
  \begin{align*}
  I: \text{sub } r4, r1, r3 \\
  J: \text{add } r1, r2, r3 \\
  K: \text{mul } r6, r1, r7
  \end{align*}
  \]

• Called an **“anti-dependence”** by compiler writers. This results from reuse of the name “r1”.

• Can’t happen in MIPS 5 stage pipeline because:
  – All instructions take 5 stages, and
  – Reads are always in stage 2, and
  – Writes are always in stage 5
THANK YOU FOR YOUR ATTENTION

WE’RE GOING TO DO SCIENCE.