MIPS with Pipelining
The MIPS is used in

- Embedded systems,
- Cisco routers
- Nintendo 64
- Sony PlayStation, Sony PlayStation 2 and Sony PlayStation Portable
- Small computing devices
- Small consumer electronics and appliances
- Google's Honeycomb (Android 3) tablet
MIPS Controller: Sequential

Fetch:
PC outputs to memory
Instruction is loaded at the end of the state

Decode: Registers accessed

Execute: ALU computes the result

Writeback:
/load_pc and write reg_file

Memory:
opcode≠sw/

opcode=sw/mem_write and load_pc

opcode=jw or sw /

opcode=beq or opcode bne/load_pc

opcode=jr or j
/load_pc

opcode≠jr /

opcode=jr or j
/load_pc

opcode≠jr /

opcode=jw or sw /_
MIPS: Five Stages
MIPS: Five Stages

1. **IF**: instruction fetch
2. **ID**: instruction decode and register read
3. **EX**: ALU execution
4. **MEM**: data memory read or write
5. **WB**: write result back into a register
MIPS: Stage One

(1) **IF**: instruction fetch

Fetch the current instruction from memory using the Program Counter (PC) as the address
- add 4 to the PC (MIPS instruction=32 bits=4 bytes)
- and store new address as NPC

PC is a register that contains the memory address of the next instruction.
Q: Why does PC have to be incremented by 4?

A: PC points (contains) to the first byte of the instruction; each instruction is 4 bytes (32 bits) long.
(2) **ID**: instruction decode and register read

- **Determine** which instruction is given
- **Fetch** the register values  
  (2 registers in MIPS instruction set)
- **Compare** the two registers and set the EQUAL flag if equal
# MIPS: Stage Two

## MIPS Instruction Formats

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>I</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>op code</td>
<td>source 1</td>
<td>source 2</td>
</tr>
<tr>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>op code</td>
<td>base reg</td>
<td>src/dest</td>
</tr>
<tr>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>op code</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6 bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MIPS: Stage Three

(3) **EX**: ALU execution

- **Memory reference**: add the base register and the offset to form the effective address
- **Register-Register instruction**: add, multiply, logic operation
- **Register-Immediate instruction**: add, multiply, logic operation on first register and the immediate value
(4) **MEM**: data memory read or write

- **LOAD** or **STORE**: access memory
- **BRANCH**: update the PC using either PC or the output of the ALU operation
- Otherwise do nothing

Q: Why do we need Memory in MIPS?  
A:  
https://www.youtube.com/watch?v=rZev35tJaEY
(5) **WB:**
- write results back into an appropriate register
Example: \texttt{add $s0, $s1, $s2}

- **IF** get instruction at PC from memory

<table>
<thead>
<tr>
<th>op code</th>
<th>source 1</th>
<th>source 2</th>
<th>dest</th>
<th>shamt</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>10001</td>
<td>10010</td>
<td>10000</td>
<td>00000</td>
<td>100000</td>
</tr>
</tbody>
</table>

- **ID** determine what instruction is and read registers
  - 000000 with 100000 is the add instruction
  - get contents of $s1 and $s2 (eg: $s1=7, $s2=12)

- **EX** add 7 and 12 = 19
- **MEM** do nothing for this instruction
- **WB** store 19 in register $s0
Example: `lw $t2, 16($s0)`

- **IF** get instruction at PC from memory

<table>
<thead>
<tr>
<th>op code</th>
<th>base reg</th>
<th>src/dest</th>
<th>offset or immediate value</th>
</tr>
</thead>
<tbody>
<tr>
<td>010111</td>
<td>10000</td>
<td>01000</td>
<td>00000000000010000</td>
</tr>
</tbody>
</table>

- **ID** determine what 010111 is
  - 010111 is `lw`
  - get contents of `$s0` and `$t2` (we don’t know that we don’t care about `$t2`) `$s0=0x200D1C00`, `$t2=77763`

- **EX** add 16 to 0x200D1C00 = 0x200D1C10
- **MEM** load the word stored at 0x200D1C10
- **WB** store loaded value in `$t2`

Load will take the data from memory and put it into the register file
Store will take the data from register file and put it into the memory
PIPELINING
Pipelining: examples

- **Laundry**

  ![Diagram of pipelining in laundry process]

  **Serial**

  ![Diagram of serial process]

  ![Diagram of pipelined process]

  - Wash
  - Dry
  - Fold Clothes
  - Put Away
Pipelining: examples

- Factory Assembly Line

**Bad Pipelining:** Chaplin: Modern Times

**Good Pipelining:** Kia Sportage Assembly Line
Serial Microprocessor:
- Executes $n$ instructions using $s$ number of stages in $ns$ clock periods
- Total Execution Time $= ns$

Pipelined Microprocessor:
- Executes $n$ instructions using $s$ number of stages
- Total Execution Time $= s + (n - 1)$

Examples:
- $n=30$, $s=5$
- Serial Microprocessor $\implies 150$ clock cycles
- Pipelined Microprocessor $\implies 5+29=34$ clock cycles
MIPS: Five Stages with Pipeline Registers

Instruction Fetch | Instruction Decode | Execute Address Calc. | Memory Access | Write Back
IF | ID | EX | MEM | WB

- IF: Instruction Fetch
- ID: Instruction Decode
- EX: Execute Address Calc.
- MEM: Memory Access
- WB: Write Back

Diagram showing the five stages of processing an instruction in the MIPS pipeline.
Inter-stage registers are master-slave D flip-flops; the master receives new data from the previous stage of the instruction while the slave flip-flop provides data to the next stage.
Master Slave D-Flipflop

![Diagram of Master Slave D-Flipflop](image.png)
MIPS: Five Stages with Pipeline Registers

(1) **IF**: instruction fetch

   IF/ID Register

(2) **ID/RF**: instruction decode and register fetch

   ID/EX Register

(3) **EX**: ALU execution

   EX/MEM Register

(4) **MEM**: data memory read or write

   MEM/WB Register

(5) **WB**: write result back into a register
Pipelining: Without Controller

Diagram showing a pipeline with stages including Memory, Reg. Block, ALU, and MEM/WB, with signals and data paths.
Pipelining: With Controller
Pipelining: With Controller

- Controller is necessary to switch operations within and between stages

- Control information must be carried as a part of the instruction, since this information is required at different stages of the pipeline

- This can be done by adding more inter-stage storage register bits to forward control data yet to be used
## MIPS Pipelining: Example Program*

<table>
<thead>
<tr>
<th>Assembly Code</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>idle</td>
<td></td>
</tr>
<tr>
<td>lw $10, 20($1)</td>
<td>$10 = $1 + 20</td>
</tr>
<tr>
<td>sub $11, $2, $3</td>
<td>$11 = $2 - $3</td>
</tr>
<tr>
<td>and $12, $4, $5</td>
<td>$12 = $4 and $5</td>
</tr>
<tr>
<td>or $13, $6, $7</td>
<td>$13 = $6 or $7</td>
</tr>
<tr>
<td>add $14, $8, $9</td>
<td>$14 = $8 + $9</td>
</tr>
</tbody>
</table>

*Ref: Dodge, "Lecture 20: The PIPELINED MIPS PROCESSOR", UT Dallas, 2012
MIPS Pipelining: \texttt{lw $10, 20($1)}
MIPS Pipelining: \texttt{sub $11, $2, $3}
MIPS Pipelining: and $12, $4, $5
MIPS Pipelining: or $13, $6, $7
MIPS Pipelining: \texttt{add $14, $8, $9}
MIPS Pipelining: Processing
MIPS Pipelining: Processing
MIPS Pipelining: Processing
MIPS Pipelining: Processing
MIPS Pipelining: Idle

IF: Idle
ID/RF: Idle
EX: Idle
MEM: Idle
WB: Idle
MIPS Pipelining: Summary

- Pipelining replaces the serial processor with a row of five 'mini-processors', each capable of completing one part of each instruction.

- A new instruction is started every clock cycle.

- Inter-process registers store instruction information (data, write register, branch conditions) between cycles so that the instructions are passed between the pipeline stages.

- When the pipeline is filled with instructions, an instruction completes every clock cycle.
Hazards in Pipelining
MIPS Pipelining: Hazards

- Hazards occur because data required for executing the current instruction may not be available

Example:

- An instruction in the FETCH cycle may need data from a register whose value will be changed by an instruction elsewhere but still in process in the pipeline: e.g. EX, MEM, or WB cycle)

- So FETCH instruction could access a register and get incorrect data because the register data has not yet been updated by other instruction(s)
MIPS Pipelining: Types of Hazards

Two Types of Hazards

Data Hazard
- Data hazards occur when an instruction needs register contents for an arithmetic/logical/memory instruction.

Control Hazard
- Control hazards occur when a branch instruction is pending and the data necessary to initiate/bypass the branch is not yet available in the same sort of scenario.

Both occur because an instruction in the ID/RF stage of the MIPS pipeline needs register data that will be shortly updated by instructions in the EX or MEM/Bypass, or WB stage.
Here, last four instructions require data from $2$, which is changed in the first instruction

- data in $2$ will not be rewritten until cycle 4, so the AND (2nd instruction) and OR (3rd instruction) will fetch incorrect data from $2$
- ADD may not get correct information
- SW will be correct
Data Hazards: Example2

“Take the contents of $2 from the register file, add 4 to it, access the memory at that location, then move the data at that location into $1 in the register file”

\[
\text{lw } \$1, \ 4(\$2) \quad \text{IF} \quad \text{id} \quad \text{ex} \quad \text{mem} \quad \text{wb}
\]

\[
\text{add } \$3, \ \$1,\$4 \quad \text{IF} \quad \text{id} \quad \text{ex} \quad \text{mem} \quad \text{wb}
\]

Load will take the data from memory and put it into the register file

Store will take the data from register file and put it into the memory
Solution: Stalling

- One solution is to insert bubbles
- This means delaying certain operations in the pipeline

\[
\text{lw} \quad \text{ID} \quad \text{EX} \quad \text{MEM} \quad \text{WB}
\]

\[
\text{lw} \quad \text{ID} \quad \text{EX} \quad \text{MEM} \quad \text{WB}
\]

\[
\text{add} \quad \text{ID}
\]

- Another solution may require modification in the datapath, which will raise the hardware cost

- Hazards slow down the instruction execution speed

- An alternative approach to deal with this is for the compiler (or the assembler) to insert NOP instructions, or reorder the instructions
Hazard and Solution: Example

- Problem 14 Practice Test