Single-cycle datapath, slightly rearranged

Diagram showing the flow of data through various components such as instruction memory, registers, and ALU. The diagram includes signals like PCSrc, MemToReg, MemRead, MemWrite, RegWrite, ALUOp, ALUSrc, RegDst, and data flows.
Pipeline registers

- In pipelining, we divide instruction execution into multiple cycles
  - IF      ID      EX      MEM      WB
- Information computed during one cycle may be needed in a later cycle:
  - Instruction read in IF stage determines which registers are fetched in ID stage, what immediate is used for EX stage, and what destination register is for WB
  - Register values read in ID are used in EX and/or MEM stages
  - ALU output produced in EX is an effective address for MEM or a result for WB
- A lot of information to save!
  - Saved in intermediate registers called pipeline registers
- The registers are named for the stages they connect:
  - IF/ID     ID/EX     EX/MEM     MEM/WB
- No register is needed after the WB stage, because after WB the instruction is done
Pipelined datapath

Instruction memory

IF/ID

ID/EX

EX/MEM

MEM/WB

PC

PCSrc

Add

Instr [31-0]

Read Instruction address [31-0]

Instruction memory

Read Instruction memory

ID/EX

RegWrite

Read register 1

Read register 2

Write register

Write data

Instruction [15 - 0]

Instr [20 - 16]

Instr [15 - 11]

Sign extend

ALUOp

ALUSrc

Shift left 2

Add

Zero

Result

RegWrite

RegDst

MemWrite

Data memory

Write data

Read data

MemToReg

MemRead

PCSrc

1

0
Propagating values forward

- Data values required *later* propagated through the pipeline registers

- The most extreme example is the destination register (*rd* or *rt*)
  - It is retrieved in IF, but isn’t updated until the WB
  - Thus, it must be passed through *all* pipeline stages, as shown in red on the next slide

- Notice that we can’t keep a single “instruction register,” because the pipelined machine needs to fetch a new instruction every clock cycle
The destination register

[Diagram showing the flow of data and signal processing through different stages: IF/ID, ID/EX, EX/MEM, MEM/WB. Key processing stages include:
- **Instruction memory**: Reads the instruction address [31-0].
- **Registers**: Contains read and write operations for registers 1 and 2, and data.
- **Sign extend** and **Shift left 2** stages for data manipulation.
- **ALU** with **Zero** and **Result** outputs, driven by **ALUOp** and **ALUSrc** inputs.
- **PCSrc** and **MemToReg** signals for program control.
- **MemWrite** and **MemRead** for memory access.

Flow of signals and data through these stages, illustrating the processing pipeline.]
What about control signals?

- Control signals generated similar to the single-cycle processor
  - in the ID stage, the processor decodes the instruction fetched in IF and produces the appropriate control values

- Some of the control signals will not be needed until later stages
  - These signals must be propagated through the pipeline until they reach the appropriate stage
  - We just pass them in the pipeline registers, along with the data

- Control signals can be categorized by the pipeline stage that uses them

<table>
<thead>
<tr>
<th>Stage</th>
<th>Control signals needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX</td>
<td>ALUSrc     ALUOp     RegDst</td>
</tr>
<tr>
<td>MEM</td>
<td>MemRead    MemWrite   PCSrc</td>
</tr>
<tr>
<td>WB</td>
<td>RegWrite   MemToReg</td>
</tr>
</tbody>
</table>
An example execution sequence

- Here’s a sample sequence of instructions to execute:
  ```
  1000: lw $8, 4($29)
  1004: sub $2, $4, $5
  1008: and $9, $10, $11
  1012: or $16, $17, $18
  1016: add $13, $14, $0
  ```

- We’ll make some assumptions, just so we can show actual data values:
  - Each register contains its number plus 100. For instance, register $8 contains 108, register $29 contains 129, etc.
  - Every data memory location contains 99

- Our pipeline diagrams will follow some conventions:
  - An X indicates values that aren’t important, like the constant field of an R-type instruction
  - Question marks ??? indicate values we don’t know, usually resulting from instructions coming before and after the ones in our example
Cycle 1 (filling)

IF: lw $8, 4($29)
MEM: ???
WB: ???

**Control**

IF/ID

Add 1004

PCSrc 4

**Add**

IF: lw $8, 4($29)

ID: ???

EX: ???

MEM: ???

WB: ???

**Instruction memory**

Read Instruction address [31-0]

**Registers**

Read register 1
data 1

Read register 2
Read data 2

Write register

Write data

**Sign extend**

RegWrite (?)

Shift left 2

**ALU**

Zero Result

ALUOp (???)

**Data memory**

Address

Write data

Read data

MemWrite (?)

MemToReg (?)

**Instruction**

Memory

Write data

Read data

MemRead (?)

???
Cycle 2

IF: sub $2, $4, $5
ID: lw $8, 4($29)
EX: ???
MEM: ???
WB: ???

IF/ID
1004
1008
PCSrc
Add
P
C

Read Instruction address [31-0]
Instruction memory

ID/EX
Control

RegWrite (?)

Shift left 2

Add

ALU
Zero
Result

ALUSrc (?)

ALUOp (???)

RegDst (?)

MemWrite (?)

MemToReg (?)

Sign extend

Read register 1
Read data 1
Read register 2
Read data 2
Write register
Write data

RegWrite (?)

MemRead (?)

Write data
Read data

Data memory

Address

???

???
Cycle 3

IF: and $9, $10, $11
ID: sub $2, $4, $5
EX: lw $8, 4($29)
MEM: ??
WB: ??

Instruction memory

Read address [31-0]

Read Instruction memory

Add

1012

IF/ID

Shift left 2

RegWrite (?)

Sign extend

RegDst (0)

ALUSrc (1)

ALUOp (add)

ALU

Zero

Result

133

Data memory

Address

Write data

Read data

MemRead (?)

MemToReg (?)

Write data

Read data

MemWrite (?)

MEM/WB

MEM/EX

EX/WB

Control

Control

Read register 1

Read data 1

Write register

Write data

Read register 2

Read data 2

Write register

Write data
Cycle 4

IF: or $16, $17, $18
ID: and $9, $10, $11
EX: sub $2, $4, $5
MEM: lw $8, 4($29)
WB: ???
Cycle 5 (full)

IF: add $13, $14, $0
ID: or $16, $17, $18
EX: and $9, $10, $11
MEM: sub $2, $4, $5
WB: lw $8, 4($29)

13
Cycle 6 (emptying)

IF: ???
ID: add $13, $14, $0
EX: or $16, $17, $18
MEM: and $9, $10, $11
WB: sub $2, $4, $5

Read address [31-0]
Instruction memory

PCSrc
Add

Control

RegWrite (1)
Shift left 2

ALU
ALUOp (or)
Zero

Result

RegDst (1)

Registers

RegWrite (1)

Instruction

Shift

Add

PCSrc
Add

WC

MemToReg (0)

MemWrite (0)

MemRead (0)

Data memory

Address

Write data
Read data

Read data 1
Read data 2

Write register 1
Write register 2

Read register 1
Read register 2

Write register
Write data

Sign extend

Add

ID/EX

EX/MEM

MEM/WB

ID: add $13, $14, $0
EX: or $16, $17, $18
MEM: and $9, $10, $11
WB: sub $2, $4, $5

Control

RegWrite (1)

Shift left 2

ALU

Result

Zero

ALUOp (or)

RegDst (1)

Registers

Read register 1
Read register 2

Write register 1
Write register 2

Read data 1
Read data 2

Write data
Read data

Sign extend

Add
Cycle 7

IF: ???

ID: ???

EX: add $13, $14, $0

MEM: or $16, $17, $18

WB: and $9, $10, $11

PC Src

Add

Registers

Shift left 2

ALU

Add

Shift left 2

ALU

MemWrite (0)

MemRead (0)

MemToReg (0)

MemWrite (0)

MemRead (0)

MemToReg (0)

MemWrite (0)

MemRead (0)

MemToReg (0)

MemWrite (0)

MemRead (0)

MemToReg (0)

MemWrite (0)

MemRead (0)

MemToReg (0)
That’s a lot of diagrams there

- Compare the last few slides with the pipeline diagram above
  - You can see how instruction executions are overlapped
  - Each functional unit is used by a different instruction in each cycle
  - The pipeline registers save control and data values generated in previous clock cycles for later use
  - When the pipeline is full in clock cycle 5, all of the hardware units are utilized. This is the ideal situation, and what makes pipelined processors so fast

- See the textbook for more examples
Instruction set architectures and pipelining

- The MIPS instruction set was designed especially for easy pipelining:
  - All instructions are 32-bits long, so the instruction fetch stage just needs to read one word on every clock cycle
  - Fields are in the same position in different instruction formats—the opcode is always the first six bits, rs is the next five bits, etc. This makes things easy for the ID stage
  - MIPS is a register-to-register architecture, so arithmetic operations cannot contain memory references. This keeps the pipeline shorter and simpler

- Pipelining is harder for older/more complex instruction sets:
  - If different instructions had different lengths or formats, the fetch and decode stages would need extra time to determine the actual length of each instruction and the position of the fields
  - With memory-to-memory instructions, additional pipeline stages may be needed to compute effective addresses and read memory *before* the EX stage
Note how everything goes left to right, except ...
An example with dependencies

- There are several **dependencies** in this new code fragment
  - the first instruction, SUB, stores a value into $2
  - that register is used as a source in the rest of the instructions

- This is not a problem for the single-cycle datapath
  - each instruction is executed completely before the next one begins, so instructions 2 through 5 above use the new value of $2

- How would this code sequence fare in our 5-stage MIPS pipeline?
### Data hazards in the pipeline diagram

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub $2, $1, $3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and $12, $2, $5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or $13, $6, $2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add $14, $2, $2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw $15, 100($2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The **sub** instruction does not write to register $2$ until clock cycle 5. This causes two data hazards in our current pipelined datapath:
  - the **and** reads register $2$ in cycle 3, and since **sub** hasn’t modified the register yet, this will be the old value of $2$, not the new one
  - the **or** instruction uses register $2$ in cycle 4, again before it’s actually updated by **sub**
## Things that are okay

<table>
<thead>
<tr>
<th>Instruction</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub $2, $1, $3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and $12, $2, $5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or $13, $6, $2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add $14, $2, $2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw $15, 100($2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The **add** instruction is okay, because of the register file design
  - registers are written at the beginning of a clock cycle
  - the new value will be available by the end of that cycle

- The **sw** is no problem at all, since it reads $2 after the **sub** finishes
Dependency arrows

- Arrows indicate the flow of data between instructions
  - The tails of the arrows show when register $2$ is written
  - The heads of the arrows show when $2$ is read

- Any arrow that points backwards in time represents a data hazard in our basic pipelined datapath
Bypassing the register file

- The actual result $1 - 3$ is computed in clock cycle 3, before it is needed in cycles 4 and 5.

- If we could somehow bypass the writeback and register read stages when needed, then we can eliminate these data hazards.

- Essentially, we need to pass the ALU output from \texttt{sub} directly to the \texttt{and} and \texttt{or} instructions, without going through the register file.

<table>
<thead>
<tr>
<th></th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>sub $2, $1, $3</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>and $12, $2, $5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>or $13, $6, $2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>
Pipeline Registers to the rescue!

- Pipeline stages communicate through pipeline registers:
  
  IF/ID      ID/EX      EX/MEM      MEM/WB

- We “forward” data from pipeline registers to later instructions