### Sequential RISC-V Datapath

<table>
<thead>
<tr>
<th>Phase</th>
<th>Pictogram</th>
<th>$t_{step}$ Serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td></td>
<td>200 ps</td>
</tr>
<tr>
<td>Reg Read</td>
<td></td>
<td>100 ps</td>
</tr>
<tr>
<td>ALU</td>
<td></td>
<td>200 ps</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
<td>200 ps</td>
</tr>
<tr>
<td>Register Write</td>
<td></td>
<td>100 ps</td>
</tr>
</tbody>
</table>

$\sum t_{instruction} = 800 \text{ ps}$

- **Instruction sequence:**
  - `add t0, t1, t2`
  - `or t3, t4, t5`
  - `sll t6, t0, t3`
Pipelined RISC-V Datapath

<table>
<thead>
<tr>
<th>Phase</th>
<th>Pictogram</th>
<th>$t_{step}$ Serial</th>
<th>$t_{cycle}$ Pipelined</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td></td>
<td>200 ps</td>
<td>200 ps</td>
</tr>
<tr>
<td>Reg Read</td>
<td></td>
<td>100 ps</td>
<td>200 ps</td>
</tr>
<tr>
<td>ALU</td>
<td></td>
<td>200 ps</td>
<td>200 ps</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
<td>200 ps</td>
<td>200 ps</td>
</tr>
<tr>
<td>Register Write</td>
<td></td>
<td>100 ps</td>
<td>200 ps</td>
</tr>
</tbody>
</table>

$t_{instruction} = 800$ ps

$t_{cycle} = 1000$ ps

Instructions:
- add t0, t1, t2
- or t3, t4, t5
- sll t6, t0, t3
## Pipelined RISC-V Datapath

### Instruction Sequence
- `add t0, t1, t2`
- `or t3, t4, t5`
- `sll t6, t0, t3`

### Timing

<table>
<thead>
<tr>
<th></th>
<th>Single Cycle</th>
<th>Pipelined</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Timing</strong></td>
<td><em>( t_{step} = 100 \ldots 200 \text{ ps} )</em></td>
<td><em>( t_{cycle} = 200 \text{ ps} )</em></td>
</tr>
<tr>
<td><strong>Instruction time, ( t_{instruction} )</strong></td>
<td><em>( = t_{cycle} = 800 \text{ ps} )</em></td>
<td><em>1000 ps</em></td>
</tr>
<tr>
<td><strong>CPI (Cycles Per Instruction)</strong></td>
<td>~1 (ideal)</td>
<td>~1 (ideal), &lt;1 (actual)</td>
</tr>
<tr>
<td><strong>Clock rate, ( f_s )</strong></td>
<td><em>1/800 ps = 1.25 GHz</em></td>
<td><em>1/200 ps = 5 GHz</em></td>
</tr>
<tr>
<td><strong>Relative speed</strong></td>
<td>1 x</td>
<td>4 x</td>
</tr>
</tbody>
</table>

### Diagram
- Diagram showing the pipeline stages: \( \text{IF} \), \( \text{ID} \), \( \text{EX} \), \( \text{MA} \), \( \text{WB} \)
- Timing annotations for each stage: \( t_{cycle} \)
- Instruction sequence: `add t0, t1, t2`, `or t3, t4, t5`, `sll t6, t0, t3`
Sequential vs. Simultaneous

- What happens sequentially and what simultaneously?

```
add t0, t1, t2
or t3, t4, t5
sll t6, t0, t3
sw t0, 4(t3)
lw t0, 8(t3)
addi t2, t2, 1
```

\[ t_{\text{instruction}} = 1000\text{ps} \]

\[ t_{\text{cycle}} \]
Sequential vs. Simultaneous

- What happens sequentially and what simultaneously?

- Instruction sequence:
  - `add t0, t1, t2`
  - `or t3, t4, t5`
  - `sll t6, t0, t3`
  - `sw t0, 4(t3)`
  - `lw t0, 8(t3)`
  - `addi t2, t2, 1`

- Instruction timing: `t_{instruction} = 1000ps`

- Resource use over time:

- Resource use in a particular time slot:

- Instruction cycle time: `t_{cycle}`
Pipelining Datapath
Single-Cycle RV32I Datapath

- IMEM (Instruction Memory)
- DMEM (Data Memory)
- Adder
- ALU
- Control logic
- Branch predictor
- Register file
- Bus interface

Key signals:
- Inst[31:0]: Instruction
- Inst[24:20]: Address
- Inst[19:15]: Address
- Inst[11:7]: Address
- Inst[31:20]: Immediate
- PC[+4]: Program Counter
- Address A, B, Data A, B, D
- ALUSel: ALU Select
- MemRW: Memory Read/Write
- BrEq, BrLT, BrUn: Branch
- Bsel, Asel, ALUSel: Bus Select
- RegWEn: Register Write Enable
- ImmSel: Immediate Select
- Imm.: Immediate Generation
- clk: Clock signal

RISC-V (35)
Single-Cycle RV32I Datapath

Instruction Fetch (IF)

Instruction Decode/Register Read (ID)

ALU Execute (EX)

Memory Access (MA)

Write Back (WB)
Recalculate PC+4 in M stage to avoid sending both PC and PC+4 down pipeline.

Must pipeline instruction along with data, so control operates correctly in each stage.
Pipelined RV32I Datapath

Pipeline registers separate stages, hold data for each instruction in flight.
Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation
  - Information is stored in pipeline registers for use by later stages
Pipeline Hazards
Hazards Ahead!

- **WARNING**
  - Fall Hazard.
  - Stay Clear.

- **WARNING**
  - High voltage inside.
  - Keep out!
  - Will shock, burn or cause death.

- **WARNING**
  - Beyond this point: Radio frequency fields at this site may exceed FCC rules for human exposure.
  - Failure to obey all posted signs and site guidelines for working in radio frequency environments could result in serious injury.

- **CAUTION**
  - X-RAY RADIATION
Pipelining Hazards

A **hazard** is a situation that prevents starting the next instruction in the next clock cycle

1) **Structural hazard**
   - A required resource is busy (e.g. needed in multiple stages)

2) **Data hazard**
   - Data dependency between instructions
   - Need to wait for previous instruction to complete its data read/write

3) **Control hazard**
   - Flow of execution depends on previous instruction
**Problem:** Two or more instructions in the pipeline compete for access to a single physical resource

**Solution 1:** Instructions take it in turns to use resource, some instructions have to stall

**Solution 2:** Add more hardware to machine

Can always solve a structural hazard by adding more hardware
Regfile Structural Hazards

- Each instruction:
  - Can read up to two operands in decode stage
  - Can write one value in writeback stage

- Avoid structural hazard by having separate “ports”
  - Two independent read ports and one independent write port

- Three accesses per cycle can happen simultaneously
Structural Hazard: Memory Access

Instruction sequence:

```
add t0, t1, t2
lw t0, 8(t3)
slt t6, t0, t3
sw t0, 4(t3)
addi t0, t1, t2
```

- Instruction and data memory used simultaneously
  - Use two separate memories
Instruction and Data Caches

- Fast, on-chip memory, separate for instructions and data

Diagram:
- Processor
  - Control
  - Datapath
    - Program Counter (PC)
    - Registers
    - Arithmetic-Logic Unit (ALU)
- Instruction Cache
- Data Cache
- Memory
  - Program
  - Data
  - Bytes

Garcia, Nikolić

RISC-V (46)
Structural Hazards – Summary

- Conflict for use of a resource
- In RISC-V pipeline with a single memory
  - Load/store requires data access
  - Without separate memories, instruction fetch would have to stall for that cycle
    - All other operations in pipeline would have to wait
- Pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches
- RISC ISAs (including RISC-V) designed to avoid structural hazards
  - e.g. at most one memory access/instruction
Data Hazards
Data Hazard: Register Access

- Separate ports, but what if write to same register as read?

Does `sw` in the example fetch the old or new value?

Instruction sequence:

- `add t0, t1, t2`
- `or t3, t4, t5`
- `slt t6, t0, t3`
- `sw t0, 4(t3)`
- `addi t0, t1, t2`
**Data Hazard: Register Access**

- Exploit high speed of register file (100 ps)
  1) WB updates value
  2) ID reads new value

• Indicated in diagram by shading

Instruction sequence:

- `add t0, t1, t2`
- `or t3, t4, t5`
- `slt t6, t0, t3`
- `sw t0, 4(t3)`
- `addi t0, t1, t2`

*Might not always be possible to write then read in same cycle, especially in high-frequency designs. Check assumptions in any question.*
Data Hazard: ALU Result

Value of \( s_0 \)

\[
\begin{align*}
\text{add} & \quad s_0, t_0, t_1 \\
\text{sub} & \quad t_2, s_0, t_0 \\
\text{or} & \quad t_6, s_0, t_3 \\
\text{xor} & \quad t_5, t_1, s_0 \\
\text{sw} & \quad s_0, 8(t_3)
\end{align*}
\]

Without some fix, \textbf{sub} and \textbf{or} will calculate wrong result!
Problem: Instruction depends on result from previous instruction

- `add s0, t0, t1`
- `sub t2, s0, t3`

"bubbles"

Bubble:
- Effectively `nop`: Affected pipeline stages do "nothing"
Stalls and Performance

- Stalls reduce performance
  - But stalls are required to get correct results
- Compiler can arrange code or insert nops (\texttt{addi \ x0, \ x0, \ 0}) to avoid hazards and stalls
  - Requires knowledge of the pipeline structure
Solution 2: Forwarding

Value of s0

```
add s0, t0, t1
sub t2, s0, t0
or t6, s0, t3
xor t5, t1, s0
sw s0, 8(t3)
```

Forwarding: grab operand from pipeline stage, rather than register file
Forwarding (aka Bypassing)

- Use result when it is computed
  - Don’t wait for it to be stored in a register
  - Requires extra connections in the datapath

```
add s0, t0, t1
sub t2, s0, t3
```
Data Needed for Forwarding (Example)

- Compare destination of older instructions in pipeline with sources of new instruction in decode stage.
- Must ignore writes to x0!

\[
\begin{align*}
\text{add} & \ t0, \ t0, \ t1 \\
\text{sub} & \ t3, \ t0, \ t5 \\
\text{sub} & \ t6, \ t0, \ t3 \\
\end{align*}
\]
Pipelined RV32I Datapath

Remember to forward operand B as well!