Pipelining
RISC-V
### 'Sequential' RISC-V Datapath

<table>
<thead>
<tr>
<th>Phase</th>
<th>Pictogram</th>
<th>$t_{\text{step}}$ 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>
<tr>
<td>$t_{\text{instruction}}$</td>
<td></td>
<td>800 ps</td>
</tr>
</tbody>
</table>

**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><img src="image1" alt="Pictogram" /></td>
<td>200 ps</td>
<td>200 ps</td>
</tr>
<tr>
<td>Reg Read</td>
<td><img src="image2" alt="Pictogram" /></td>
<td>100 ps</td>
<td>200 ps</td>
</tr>
<tr>
<td>ALU</td>
<td><img src="image3" alt="Pictogram" /></td>
<td>200 ps</td>
<td>200 ps</td>
</tr>
<tr>
<td>Memory</td>
<td><img src="image4" alt="Pictogram" /></td>
<td>200 ps</td>
<td>200 ps</td>
</tr>
<tr>
<td>Register Write</td>
<td><img src="image5" alt="Pictogram" /></td>
<td>100 ps</td>
<td>200 ps</td>
</tr>
</tbody>
</table>

$t_{instruction}$: 800 ps  
$1000$ ps

### Instruction Sequence

- `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`

#### Single Cycle
- **Timing**
  - $t_{step} = 100 \ldots 200$ ps
- **Instruction time, $t_{instruction}$**
  - $= t_{cycle} = 800$ ps
- **CPI (Cycles Per Instruction)**
  - ~1 (ideal)
- **Clock rate, $f_s$**
  - $1/800 \text{ ps} = 1.25$ GHz
- **Relative speed**
  - 1 x

#### Pipelined
- **Timing**
  - $t_{cycle} = 200$ ps
- **Instruction time, $t_{instruction}$**
  - $= t_{cycle} = 800$ ps (1000 ps)
- **CPI (Cycles Per Instruction)**
  - ~1 (ideal), <1 (actual)
- **Clock rate, $f_s$**
  - $1/200 \text{ ps} = 5$ GHz
- **Relative speed**
  - 4 x
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

- $t_{\text{instruction}} = 1000\text{ps}$
- $t_{\text{cycle}}$
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 $t2, t2, 1$

$t_{instruction} = 1000ps$

Resource use of instruction over time

Resource use in a particular time slot

$t_{cycle}$
Pipelining
Datapath
Single-Cycle RV32I Datapath

Diagram showing the single-cycle RV32I datapath architecture.
Single-Cycle RV32I Datapath

Instruction Fetch (IF)

Instruction Decode/Register Read (ID)

ALU Execute (EX)

Memory Access (MA)

Write Back

Instruction Fetch (IF)

Instruction Decode/Register Read (ID)

ALU Execute (EX)

Memory Access (MA)

Write Back

RISC-V (36)
Pipelined RV32I Datapath

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

\[ \text{lw } t0, 8(t3) \quad \text{sw } t0, 4(t3) \quad \text{slt } t6, t0, t3 \quad \text{or } t3, t4, t5 \]
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.
  - In accordance with Federal Communication Commission rules on radio frequency emissions of 47 CFR 15.104

- **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 and data memory used simultaneously
  - Use two separate memories

Instruction sequence:

1. `add t0, t1, t2`
2. `lw t0, 8(t3)`
3. `slt t6, t0, t3`
4. `sw t0, 4(t3)`
5. `addi t0, t1, t2`
Instruction and Data Caches

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

Might not always be possible to write then read in same cycle, especially in high-frequency designs. Check assumptions in any question.

add t0, t1, t2
or t3, t4, t5
slt t6, t0, t3
sw t0, 4(t3)
addi t0, t1, t2
Data Hazard: ALU Result

Value of $s0$

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

Without some fix, `add` and `or` will calculate wrong result!
Problem: Instruction depends on result from previous instruction

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

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 \texttt{nop} (\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

RISC-V (54)
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

\[
\begin{align*}
\text{add} & \quad s0, \ t0, \ t1 \\
\text{sub} & \quad t2, \ s0, \ t3
\end{align*}
\]
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!

```
add t0, t0, t1
sub t3, t0, t5
sub t6, t0, t3
```
Pipelined RV32I Datapath

Remember to forward operand B as well!

Forwarding control logic
Load Data Hazard
Data Hazard and Forwarding

Value of s0

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Symbol(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>s0, t0, t1</td>
</tr>
<tr>
<td>sub</td>
<td>t2, s0, t0</td>
</tr>
<tr>
<td>or</td>
<td>t6, s0, t3</td>
</tr>
<tr>
<td>xor</td>
<td>t5, t1, s0</td>
</tr>
<tr>
<td>sw</td>
<td>s0, 8(t3)</td>
</tr>
</tbody>
</table>

Forwarding: grab operand from pipeline stage, rather than register file
Load Data Hazard

lw s2,20(s1)
and s4,s2,s5
or s8,s2,s6
add s9,s4,s2
slt s1,s6,s7

Load requires one-cycle pipeline stall

Data from memory is here
Needs to be here!

1-cycle stall unavoidable!
Forward

Unaffected

lw s2,20 (s1)
and s4, s2, s5
or s8, s2, s6
add s9, s4, s2
slt s1, s6, s7
Stall Pipeline

Load requires one-cycle pipeline stall

Instruction sequence:

1. `lw s2,20(s1)`
2. `and s4, $0, s5`
3. `or s8, s2, s6`
4. `add s9, s4, s2`
5. `slt s1, s6, s7`

Stall Repeat

and and and forward

Diagram showing the pipeline stages and the stall.
Slot after a load is called a *load delay slot*
- If that instruction uses the result of the load, then the hardware will stall for one cycle
- Equivalent to inserting an explicit `nop` in the slot
  - except the latter uses more code space
- Performance loss

**Idea:**
- Put unrelated instruction into load delay slot
- No performance loss!
Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instr!

### Original Order:

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

- **Stall!**
- **Stall!**

### Alternative:

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

- **9 cycles**
- **7 cycles**
Control Hazards
Control Hazards

beq t0, t1, Label
sub t2, s0, t0
or t6, s0, t3
xor t5, t1, s0
sw s0, 8(t3)

Two stall cycles after a branch!

Executed regardless of branch outcome!

PC updated
Observation

- If branch not taken, then instructions fetched sequentially after branch are correct.
- If branch or jump taken, then need to flush incorrect instructions from pipeline by converting to NOPs.
Kill Instructions after Branch if Taken

beq t0, t1, Label
sub t2, s0, t0
or t6, s0, t3
Label: xor t5, t1, s0

Taken branch!
Convert to \text{nop}

PC updated

Convert to \text{nop}
Reducing Branch Penalties

- Every taken branch in simple pipeline costs 2 dead cycles
- To improve performance, use “branch prediction” to guess which way branch will go earlier in pipeline
- Only flush pipeline if branch prediction was incorrect
Branch Prediction

```
beq t0,t1,Label

Label :...
```

Taken branch!

Guess next PC!

Check guess
Superscalar Processors
Increasing Processor Performance

1. **Clock rate**
   - Limited by technology and power dissipation

2. **Pipelining**
   - “Overlap” instruction execution
   - Deeper pipeline: 5 => 10 => 15 stages
     - Less work per stage $\rightarrow$ shorter clock cycle
     - But more potential for hazards (CPI > 1)

3. **Multi-issue “superscalar” processor**
Superscalar Processor

- Multiple issue “superscalar”
  - Replicate pipeline stages $\Rightarrow$ multiple pipelines
  - Start multiple instructions per clock cycle
  - CPI < 1, so use Instructions Per Cycle (IPC)
  - E.g., 4GHz 4-way multiple-issue
    - 16 BIPS, peak CPI = 0.25, peak IPC = 4
  - Dependencies reduce this in practice

- “Out-of-Order” execution
  - Reorder instructions dynamically in hardware to reduce impact of hazards

- CS152 discusses these techniques!
Superscalar Processor

In-order issue

Reservation station

Functional units

Instruction fetch and decode unit

Reservation station

Reservation station

Commit unit

Out-of-order execute

Floating point

Integer

Load-store

In-order commit

P&H, p.330
Benchmark: CPI of i7

P&H, p.330

CPI = 1
“Iron Law” of Processor Performance

CPI = Cycles Per Instruction

CPI = \frac{Cycles}{Instruction} = \frac{Time}{Program} \times \frac{Program}{Instruction} \times \frac{Time}{Cycle}

Can time
Can count
Can look up

Time
Program

Instructions
Program

Cycles
Instruction

Time
Cycle

CPI = \frac{Cycles}{Instruction} = \frac{Time}{Program} \div \left( \frac{Instructions}{Program} \times \frac{Time}{Cycle} \right)
Pipelining and ISA Design

- **RISC-V ISA designed for pipelining**
  - All instructions are 32-bits
    - Easy to fetch and decode in one cycle
    - Versus x86: 1- to 15-byte instructions
  - Few and regular instruction formats
    - Decode and read registers in one step
  - Load/store addressing
    - Calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle
“And In conclusion...”

- We have built a processor!
  - Capable of executing all RISC-V instructions in one cycle each

- 5 Phases of execution
  - IF, ID, EX, MEM, WB
  - Not all instructions are active in all phases

- Controller specifies how to execute instructions
  - Implemented as ROM or logic

- Pipelining improves performance
  - But we must resolve hazards