Today: Pipelining

How to apply the performance equation to our single-cycle CPU.

Pipelining: an idea from assembly line production applied to CPU design.

Why pipelining is hard: data hazards, control hazards, structural hazards.

Visualizing pipelines to evaluate hazard detection and resolution.

A tool kit for hazard resolution.
Performance Measurement
(as seen by a CPU designer)

Q. Why do we care about a program’s performance?
A. We want the CPU we are designing to run it well!
Step 1: Analyze the right measurement!

CPU Time:
Time the CPU spends running program under measurement.

Measuring CPU time (Unix):
% time <program name>
25.77u 0.72s 0:29.17 90.8%

Response Time:
Total time: CPU Time + time spent waiting (for disk, I/O, ...).
CPU time: Proportional to Instruction Count

Q. Once ISA is set, who can influence instruction count?
A. Compiler writer, application developer.

Q. Static count? (lines of program printout)
Or dynamic count? (trace of execution)
A. Dynamic.

Rationale: Every additional instruction you execute takes time.

Q. How does a architect influence the number of machine instructions needed to run an algorithm?
A. Create new instructions: instruction set architect.
CPU time: Proportional to Clock Period

Q. How can architects (not technologists) reduce clock period?
A. Shorten the machine’s critical path.

Q. What ultimately limits an architect’s ability to reduce clock period?
A. Clock-to-Q, setup times.

Time
Program \( \propto \) Time
One Clock Period

Rationale:
We measure each instruction’s execution time in “number of cycles”.
By shortening the period for each cycle, we shorten execution time.
Completing the performance equation

What factors make different programs have different CPIs?

- Cache behavior varies.
- Instruction mix varies.
- Branch prediction varies.

We need all three terms, and only these terms, to compute CPU Time!

When is it OK to compare clock rates?

“CPI” -- The Average Number of Clock Cycles Per Instruction For the Program
Consider Lecture 3 single-cycle CPU ...

• All instructions take 1 cycle to execute every time they run.

• CPI of any program running on machine? 1.0

“average CPI for the program” is a more-useful concept for more complicated machines ...
Consider machine with a data cache...

A program’s load instructions “stride” through every memory address.

The cache never “hits”, so every load goes to DRAM (100x slower than loads that go to cache).

Thus, the average number of cycles for load instructions is higher for this program.

Thus, the average number of cycles for all instructions is higher for this program.

Thus, program takes longer to run!

Seconds = Instructions Cycles Seconds
Program Program Instruction Cycle

Cal
Final thoughts: Performance Equation

\[
\text{Seconds Program} = \frac{\text{Instructions Program}}{\text{Cycles Instruction}}
\]

- **Goal is to optimize execution time, not individual equation terms.**
- **Machines are optimized with respect to program workloads.**
- **The CPI of the program. Reflects the program's instruction mix.**
- **Clock period. Optimize jointly with machine CPI.**
Pipelining
Recall: Our single-cycle processor

Challenge: Speed up clock while keeping CPI == 1

Seconds
Program = Instructions
Program

Cycles
Instruction

Seconds
Cycle

CPI == 1
This is good.

Slow.
This is bad.

Challenge: Speed up clock while keeping CPI == 1
Recall: An R-format CPU design

Decode fields to get: ADD $8 $9 $10
Reminder: How data flows after posedge

-PC 

Instr Mem

0x4

RegFile

Logic

ALU

op
Next posedge: Update state and repeat
Observation: Logic idle most of cycle

For most of cycle, ALU is either “waiting” for its inputs, or “holding” its output

Ideal: a CPU architecture where each part is always “working”.

For most of cycle, ALU is either “waiting” for its inputs, or “holding” its output

Ideal: a CPU architecture where each part is always “working”.

For most of cycle, ALU is either “waiting” for its inputs, or “holding” its output

Ideal: a CPU architecture where each part is always “working”.

For most of cycle, ALU is either “waiting” for its inputs, or “holding” its output

Ideal: a CPU architecture where each part is always “working".
Inspiration: Automobile assembly line

Assembly line moves on a steady clock. Each station does the same task on each car.
Inspiration: Automobile assembly line

Simpler station tasks → more cars per hour. Simple tasks take less time, clock is faster.
Inspiration: Automobile assembly line

Line speed limited by slowest task. Most efficient if all tasks take same time to do
Inspiration: Automobile assembly line

Simpler tasks, complex car → long line!

These lines go 24 x 7, and rarely shut down.
Lessons from car assembly lines

- Faster line movement yields more cars per hour off the line.

- Faster line movement requires more stages, each doing simpler tasks.

  To maximize efficiency, all stages should take same amount of time (if not, workers in fast stages are idle)

- “Filling”, “flushing”, and “stalling” assembly line are all bad news.
Key analogy: The instruction is the car

Pipeline Stage #1: Instruction Fetch

Stage #2: Controls hardware in stage 2
Stage #3: Controls hardware in stage 3
Stage #4: Controls hardware in stage 4
Stage #5: Controls hardware in stage 5

“Data-stationary control”
Example: Decode & Register Fetch stage

Pipeline Stage #1

Instr Fetch

SUB R10, R9, R8

Stage #2

Decode & Reg Fetch

OR R7, R6, R5

Stage #3

ADD R4, R3, R2

A sample program

ADD R4, R3, R2
OR R7, R6, R5
SUB R10, R9, R8

R’s chosen so that instructions are independent - like cars on the line.
Performance Equation and Pipelining

\[
\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}
\]

CPI == 1

Once pipe is fill, one instruction completes per cycle

Clock period is shorter
Less work to do in each cycle

To get shortest clock period, balance the work to do in each pipeline stage.
Hazards: An instruction is not a car ...

New sample program
ADD R4, R3, R2
OR R5, R4, R2

An example of a “hazard” -- we must
(1) detect and
(2) resolve all hazards
to make a CPU that matches ISA
Performance Equation and Hazards

Seconds
Program

= Instructions
Program

Cycles
Instruction

Seconds
Cycle

Instr Fetch

Decode & Reg Fetch

Stage #3

Some ways to cope with hazards makes CPI > 1 "stalling pipeline"

"Software slows the machine down"
Seymour Cray

Added logic to detect and resolve hazards increases clock period

PC

IR

PC

IR

PC

IR

RegFile

rd1

rd2

rs1

rs2

ws

wd

WE

Ext

Addr

Data

Instr

Mem

"Software slows the machine down"
Seymour Cray

UC Regents Fall 2008 © UCB
A (simplified) 5-stage pipelined CPU

1. **"IF" Stage**
   - Instr Fetch

2. **"ID/RF" Stage**
   - Decode & Reg Fetch

3. **"EX" Stage**
   - Execution

4. **"MEM" Stage**
   - Memory

5. **WB**
   - Write Back

Instr Fetch

Decode & Reg Fetch

Execution

Memory

Write Back
Sometimes, “contract” is a challenge

Sample Program
LW R4, 0(R0)
OR R5, R4, R2

... but we haven’t even started the load yet!

One approach: change the contract!
From Lecture 3: Delayed Loads ...

**Instruction Fetch**

**Instruction Decode**

**Operand Fetch**

**Execute**

**Result Store**

**Next Instruction**

- Fetch the load inst from memory
- Decode fields to get: \( \text{LW} \; \$1, \; 32(\$2) \)
- "Retrieve" register value: \( \$2 \)
- Compute memory address: \( 32 + \$2 \)
- Load memory address contents into: \( \$1 \)
- Prepare to fetch instr that follows the \( \text{LW} \) in the program. Depending on load semantics, new \( \$1 \) is visible to that instr, or not until the following instr ("delayed loads").
After we change the contract ...

Sample Program

LW R4, 0(R0)
OR R5, R4, R2

... "delayed load" contract does not guarantee new R4 is seen.

Only partially solves problem ... soon, we finish the story.
Visualizing Pipelines
Pipeline Representation #1: Timeline

Good for visualizing pipeline fills.

Sample Program

I1: ADD R4, R3, R2
I2: AND R6, R5, R4
I3: SUB R1, R9, R8
I4: XOR R3, R2, R1
I5: OR R7, R6, R5

Time: t1 t2 t3 t4 t5 t6 t7 t8
Inst
I1: IF ID EX MEM WB
I2: IF ID EX MEM WB
I3: IF ID EX MEM WB
I4: IF ID EX MEM WB
I5: IF ID EX MEM WB
I6: IF ID EX MEM WB

Pipeline is “full”
Representation #2: Resource Usage

Good for visualizing pipeline stalls.

Sample Program

I1: ADD R4, R3, R2
I2: AND R6, R5, R4
I3: SUB R1, R9, R8
I4: XOR R3, R2, R1
I5: OR R7, R6, R5

Time: t1  t2  t3  t4  t5  t6  t7  t8
Stage
IF:   I1  I2  I3  I4
ID:   I1  I2  I3
EX:   I1  I2
MEM:  I1
WB:   I1  I2  I3  I4

Pipeline is “full”
Hazard Taxonomy
Structural Hazards

Several pipeline stages need to use the same hardware resource at the same time.

Solution #1: Add extra copies of the resource (only works sometime).

Solution #2: Change resource so that it can handle concurrent use.

Solution #3: Stages “take turns” by stalling parts of the pipeline.
Structural Hazard Example: One Memory

Used by IF stage and MEM stage

“IF” Stage “ID/RF” Stage “EX” Stage “MEM” Stage WB

PC

RegFile

Mux, Logic

Mux, Logic

To branch logic

Mux, Logic

MemToReg
A solution: “Extra copies” of memory

“IF” Stage
Instr Fetch

“ID/RF” Stage
Decode & Reg Fetch

“EX” Stage
Execution

“MEM” Stage
Memory

WB
Write Back

Instr Fetch

RegFile

Din
Addr
WE

Data Memory

WE, MemToReg

Mux, Logic

I and D caches are a hybrid solution
Alternatively: Concurrent use ...

1. "IF" Stage
   Instr Fetch

2. "ID/RF" Stage
   Decode & Reg Fetch

3. "EX" Stage
   Execution

4. "MEM" Stage
   Memory

5. WB
   Write Back

ID and WB stages use register file in same clock cycle
Data Hazards: 3 Types (RAW, WAR, WAW)

Several pipeline stages read or write the same data location in an incompatible way.

Read After Write (RAW) hazards. Instruction I2 expects to read a data value written by an earlier instruction, but I2 executes “too early” and reads the wrong copy of the data.

Note “data value”, not “register”. Data hazards are possible for any architected state (such as main memory). In practice, main memory hazard avoidance is the job of the memory system.
Recall: RAW example

Sample program

ADD R4, R3, R2
OR R5, R4, R2

... wrong value of R4 fetched from RegFile, contract with programmer broken! Oops!

This is what we mean when we say Read After Write (RAW) Hazard
Data Hazards: 3 Types (RAW, WAR, WAW)

Write After Read (WAR) hazards. Instruction I2 expects to write over a data value after an earlier instruction I1 reads it. But instead, I2 writes too early, and I1 sees the new value.

Write After Write (WAW) hazards. Instruction I2 writes over data an earlier instruction I1 also writes. But instead, I1 writes after I2, and the final data value is incorrect.

WAR and WAW not possible in our 5-stage pipeline. But are possible in other pipeline designs.
Control Hazards: A taken branch/jump

Note: with branch delay slot, I2 MUST complete, I3 MUST NOT complete.

Sample Program (ISA w/o branch delay slot)

I1: BEQ R4, R3, 25
I2: AND R6, R5, R4
I3: SUB R1, R9, R8
I4: 
I5: 
I6: 

Time: t1 t2 t3 t4 t5 t6 t7 t8
Inst
I1: IF ID EX MEM WB
I2: IF ID
I3: IF
I4: 
I5: 
I6: 

EX stage computes if branch is taken
If branch is taken, these instructions MUST NOT complete!
Hazards Recap

- **Structural** Hazards
- **Data** Hazards (RAW, WAR, WAW)
- **Control** Hazards (taken branches and jumps)

On each clock cycle, we must detect the presence of all of these hazards, and resolve them before they break the “contract with the programmer”.

Hazard Resolution Tools
**The Hazard Resolution Toolkit**

- **Stall** earlier instructions in pipeline.
- **Forward** results computed in later pipeline stages to earlier stages.
- **Add** new hardware or **rearrange** hardware design to eliminate hazard.
- **Change ISA** to eliminate hazard.
- **Kill** earlier instructions in pipeline.
- Make hardware handle **concurrent requests** to eliminate hazard.
Resolving a RAW hazard by stalling

**Sample program**

ADD R4, R3, R2
OR R5, R4, R2

**Keep executing OR instruction until R4 is ready. Until then, send NOPS to IR 2/3.**

**New datapath hardware**

1. Mux into IR 2/3 to feed in NOP.
2. Write enable on PC and IR 1/2

**Freeze PC and IR until stall is over.**
The Hazard Resolution Toolkit

- **Stall** earlier instructions in pipeline.
- **Forward** results computed in later pipeline stages to earlier stages.
- **Add** new hardware or **rearrange** hardware design to eliminate hazard.
- **Change ISA** to eliminate hazard.
- **Kill** earlier instructions in pipeline.
- **Make** hardware handle **concurrent requests** to eliminate hazard.
Resolving a RAW hazard by forwarding

Sample program
ADD R4, R3, R2
OR R5, R4, R2

Instr Fetch
Decode & Reg Fetch
Execution

Just forward it back!

Unlike stalling, does not change CPI. May hurt cycle time.
The Hazard Resolution Toolkit

- **Stall** earlier instructions in pipeline.
- **Forward** results computed in later pipeline stages to earlier stages.
- **Add** new hardware or **rearrange** hardware design to eliminate hazard.
- **Change ISA** to eliminate hazard.
- **Kill** earlier instructions in pipeline.
- Make hardware handle **concurrent requests** to eliminate hazard.
Control Hazards: Fix with more hardware

Sample Program (ISA w/o branch delay slot)

I1: BEQ R4,R3,25
I2: AND R6,R5,R4
I3: SUB R1,R9,R8

If we add hardware, can we move it here?

If branch is taken, these instructions MUST NOT complete!

EX stage computes if branch is taken
Resolving control hazard with hardware

Stage #1
Instr Fetch

Stage #2
Decode & Reg Fetch

Stage #3

To branch control logic

0x4

RegFile

rs1
rs2
wd
ws
rd1
rd2
WE

IR

IR

IR

A

M

B

Ext

Addr
Data

Instr Mem

PC

D
Q

Instr Fetch

Decode & Reg Fetch

RegFile

rs1
rs2
wd
ws
rd1
rd2
WE

==

D
Q

PC

0x4

Ext
Control Hazards: After more hardware

If we change ISA, can we always let I2 complete ("branch delay slot") and eliminate the control hazard.

Sample Program (ISA w/o branch delay slot)

I1: BEQ R4,R3,25
I2: AND R6,R5,R4
I3: SUB R1,R9,R8

If branch is taken, this instruction MUST NOT complete!
From Lecture 3: BEQ $1, $2, 25

Fetch branch inst from memory

Decode fields to get: BEQ $1, $2, 25

“Retrieve” register values: $1, $2

Compute if we take branch: $1 == $2?

ALWAYS prepare to fetch instr that follows the BEQ in the program ("delayed branch"). IF we take branch, the instr we fetch AFTER that instruction is PC + 4 + 100.

PC == “Program Counter”
The Hazard Resolution Toolkit

- **Stall** earlier instructions in pipeline.
- **Forward** results computed in later pipeline stages to earlier stages.
- **Add** new hardware or **rearrange** hardware design to eliminate hazard.
- **Change ISA** to eliminate hazard.
- **Kill** earlier instructions in pipeline.
- **Make hardware handle** concurrent requests to eliminate hazard.
Resolve control hazard by **killing instr**

Sample program (no delay slot)

**J 200**
**OR R5, R4, R2**

Detect J instruction, mux a NOP into IR 1/2

This hurts CPI.

Can we do better?

Compute new PC using hardware not shown ...
The Hazard Resolution Toolkit

- **Stall** earlier instructions in pipeline.
- **Forward** results computed in later pipeline stages to earlier stages.
- **Add** new hardware or **rearrange** hardware design to eliminate hazard.
- **Change ISA** to eliminate hazard.
- **Kill** earlier instructions in pipeline.
- Make hardware handle **concurrent requests** to eliminate hazard.
Structural hazard solution: concurrent use

Does not come for free ...

ID and WB stages use register file in same clock cycle
Hazard Diagnosis
Data Hazards: Read After Write

Read After Write (RAW) hazards. Instruction I2 expects to read a data value written by an earlier instruction, but I2 executes “too early” and reads the wrong copy of the data.

Classic solution: use forwarding heavily, fall back on stalling when forwarding won’t work or slows down the critical path too much.
Full bypass network...
Common bug: Multiple forwards ...

Which do we forward from?

ADD R4, R3, R2
OR R2, R3, R1
AND R2, R2, R1
Common bug: Multiple forwards II...

Which do we forward from?

ADD R4, R0, R2

OR R0, R3, R1 AND R0, R2, R1

ID (Decode)

EX

MEM

WB

From

WB

RegFile
rs1
rs2
ws
rd1
rd2

WE

Mux, Logic

MemToReg

Data Memory

R
## LW and Hazards

<table>
<thead>
<tr>
<th>Type</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>addu, subu, addiu</td>
</tr>
<tr>
<td>logical</td>
<td>and, andi, or, ori, xor, xori, lui</td>
</tr>
<tr>
<td>shift</td>
<td>sll, sra, srl</td>
</tr>
<tr>
<td>compare</td>
<td>slt, slti, sltu, sltui</td>
</tr>
<tr>
<td>control</td>
<td>beq, bne, bgez, bltz, j, jr, jal</td>
</tr>
<tr>
<td>data transfer</td>
<td>lw, sw</td>
</tr>
<tr>
<td>Other:</td>
<td>break</td>
</tr>
</tbody>
</table>

No load "delay slot"
Questions about LW and forwarding

**ADDIU R1 R1 24**

**OR R3,R3,R2 LW R1 128(R29)**

Do we need to stall?

ID (Decode)  EX  MEM  WB

Mux, Logic  From WB

RegFile
rs1  rd1
rs2  ws  rd2
wd  WE

IR

Data Memory
Addr
Din
Dout
WE

MemToReg

R
Questions about LW and forwarding

Do we need to stall?

ADDIU R1 R1 24
LW R1 128(R29) OR R1,R3,R1

Mux, Logic

RegFile
rs1, rs2, ws, wd
rd1, rd2
WE

Mem, Data Memory
Din, Addr
Dout

ID (Decode)

EX

MEM

WB

From WB

A
32
B
32
M
32
Y
12

ALU

op

R

R1

Cal

CS 194-6 L5: Pipelining

UC Regents Fall 2008 © UCB
Resolving a RAW hazard by stalling

Sample program

\[
\text{ADD R4, R3, R2} \\
\text{OR R5, R4, R2}
\]

Keep executing OR instruction until R4 is ready. Until then, send NOPS to IR 2/3.

Freeze PC and IR until stall is over.

New datapath hardware

(1) Mux into IR 2/3 to feed in NOP.

(2) Write enable on PC and IR 1/2
Branches and Hazards

<table>
<thead>
<tr>
<th>Type</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>addu, subu, addiu</td>
</tr>
<tr>
<td>logical</td>
<td>and, andi, or, ori, xor, xori, lui</td>
</tr>
<tr>
<td>shift</td>
<td>sll, sra, srl</td>
</tr>
<tr>
<td>compare</td>
<td>slt, sli, sltu, sltui</td>
</tr>
<tr>
<td>control</td>
<td>beq, bne, bgez, bltz, j, jr, jal</td>
</tr>
<tr>
<td>data transfer</td>
<td>lw, sw</td>
</tr>
<tr>
<td>Other:</td>
<td>break</td>
</tr>
</tbody>
</table>
Recall: Control hazard and hardware
Recall: After more hardware, change ISA

Sample Program (ISA w/o branch delay slot)

I1: BEQ R4,R3,25
I2: AND R6,R5,R4
I3: SUB R1,R9,R8

If branch is taken, this instruction MUST NOT complete!

If we change ISA, can we always let I2 complete ("branch delay slot") and eliminate the control hazard.

Time: t1 t2 t3 t4 t5 t6 t7 t8
Inst
I1: IF
I2: IF
I3: ID
I4: EX
I5: MEM
I6: WB

ID stage computes if branch is taken

ID (Decode)
EX (ALU)
MEM
WB
IF (Fetch)
Question about branch and forwards:

**BEQ R1 R3 label**

**Will this work as shown?**

**To branch control logic**

**Mux, Logic**
Lessons learned

- Pipelining is hard
- Study every instruction
- Write test code in advance
- Think about interactions ...
Lessons learned

- Pipelining is **hard**
- Study every instruction
- Write test code **in advance**
- Think about interactions ... between forwarding, branch and jump delay slots, R0 issues LW issues ... **a long list!**
Control Implementation
Recall: What is single cycle control?

Combinational Logic
(Only Gates, No Flip Flops)
Just specify logic functions!

Instr Mem
Addr Data

RegFile
rs1 rs2 rd1 rd2 ws wd

RegDest
RegWr

Ext
ExtOp
ALUsrc
ALUctr

Equal

Data Memory
Addr Dout

MemWr
MemToReg

ALU

op

Combinational Logic
(Only Gates, No Flip Flops)
Just specify logic functions!

Instr Mem
Addr Data

RegFile
rs1 rs2 rd1 rd2 ws wd

RegDest
RegWr

Ext
ExtOp
ALUsrc
ALUctr

Equal

Data Memory
Addr Dout

MemWr
MemToReg

ALU

op
In pipelines, all IR registers are used

A “conceptual” design -- for shortest critical path, IR registers may hold decoded info, not the complete 32-bit instruction

Combinational Logic
(Only Gates, No Flip Flops)
(add extra state outside!)
Next Monday:

<table>
<thead>
<tr>
<th>M</th>
<th>L6: TBA</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/20</td>
<td></td>
</tr>
</tbody>
</table>

This Friday:
Ramp Gold meeting, BWRC.