Lecture 14

- Midterm structure
- Today's lecture:
  - Finish stalls
  - Branches
  - Another look at performance
- Why do we need to stall sometimes?

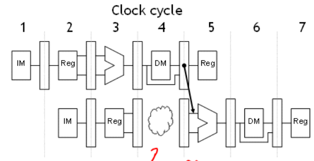
1

Stalling

- The easiest solution is to stall the pipeline.
- We could delay the AND instruction by introducing a one-cycle delay into the pipeline, sometimes called a bubble.

lw $2, 20($3)

and $12, $2, $5



- Notice that we're still using forwarding in cycle 5, to get data from the MEM/WB pipeline register to the ALU.

2

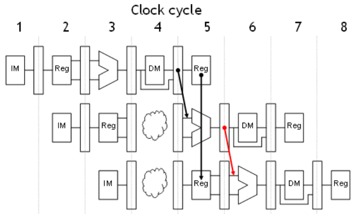
Stalling delays the entire pipeline

- If we delay the second instruction, we'll have to delay the third one too.
  - It prevents problems such as two instructions trying to write to the same register in the same cycle.
  - Also allows forwarding between AND and OR.

lw $2, 20($3)

and $12, $2, $5

or $13, $12, $2



3

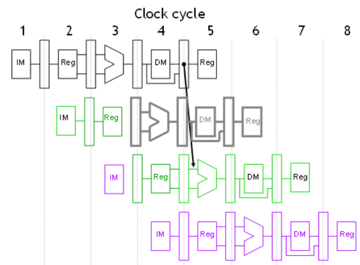
Stall = Nop conversion

lw $2, 20($3)

and -> nop

and $12, $2, $5

or $13, $12, $2



- The effect of a load stall is to insert an empty or nop instruction into the pipeline

4

Detecting Stalls, cont.

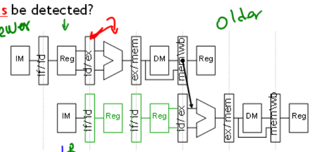
- When should stalls be detected?

newer

Older

lw $2, 20($3)

and $12, $2, $5



- What is the stall condition?

if (ID/EX.memread==1 && (ID/EX.RT == IF/ID.RS ||
ID/EX.RT == IF/ID.RT)
)

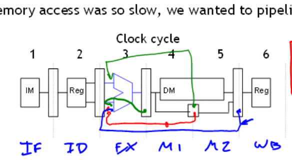
then stall

5

bypass == forwarding

Generalizing Forwarding/Stalling

- What if data memory access was so slow, we wanted to pipeline it over 2 cycles?



add $1 ←...
add $2, $1...
add $3, $1

- How many bypass inputs would the muxes in EXE have? 3
- Which instructions in the following require stalling and/or bypassing?

lw r13, 0(r11)
add r7, r8, r9
add r15, r7, r13

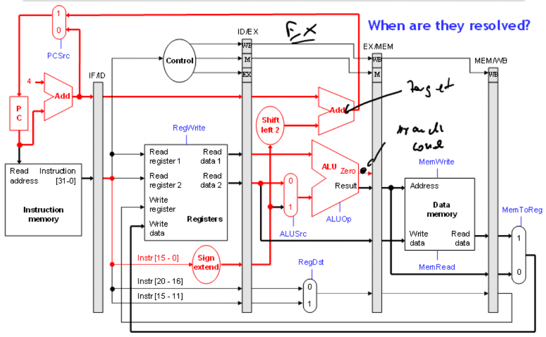
1	2	3	4	5	6	7	8	9				
IF	ID	EX	M1	M2	WB							
	IF	ID	EX	M1	M2	WB						
		IF	ID	S3	EX	M1	M2	WB				

pipe diagram

pult

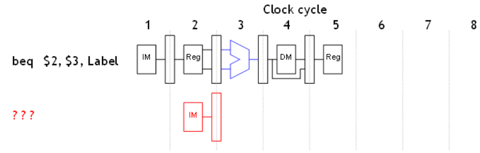
6

Branches in the original pipelined datapath



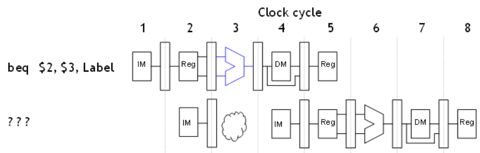
Branches

- Most of the work for a branch computation is done in the EX stage.
  - The branch target address is computed.
  - The source registers are compared by the ALU, and the Zero flag is set or cleared accordingly.
- Thus, the branch decision cannot be made until the end of the EX stage.
  - But we need to know which instruction to fetch next, in order to keep the pipeline running!
  - This leads to what's called a control hazard.



Stalling is one solution

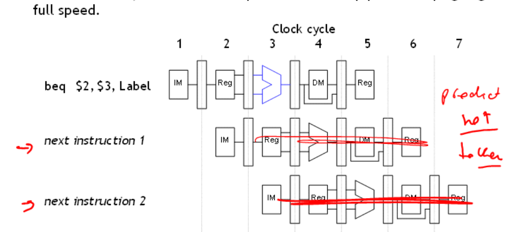
- Again, stalling is always one possible solution.



- Here we just stall until cycle 4, after we do make the branch decision.

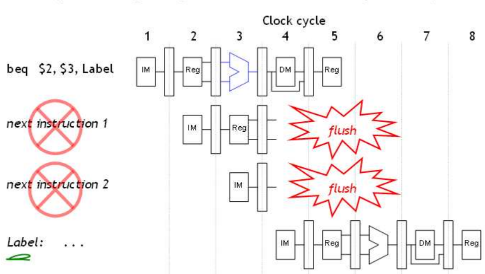
Branch prediction

- Another approach is to *guess* whether or not the branch is taken.
  - In terms of hardware, ~~it's easier to assume the branch is not taken~~.
  - This way we ~~just increment the PC and continue execution, as for normal instructions~~.
- If we're correct, then there is no problem and the pipeline keeps going at full speed.



Branch misprediction

- If our guess is wrong, then we would have already started executing two instructions incorrectly. We'll have to discard, or flush, those instructions and begin executing the right ones from the branch target address, Label.

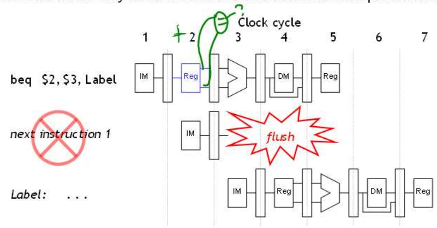


Performance gains and losses

- Overall, branch prediction is worth it.
  - Mispredicting a branch means that two clock cycles are wasted.
  - But if our predictions are even just occasionally correct, then this is preferable to stalling and wasting two cycles for *every* branch.
- All modern CPUs use branch prediction.
  - Accurate predictions are important for optimal performance. ~97%
  - Most CPUs predict branches dynamically—statistics are kept at run-time to determine the likelihood of a branch being taken. 99%
- The pipeline structure also has a big impact on branch prediction.
  - A longer pipeline may require more instructions to be flushed for a misprediction, resulting in more wasted time and lower performance.
  - We must also be careful that instructions do not modify registers or memory before they get flushed.

Implementing branches

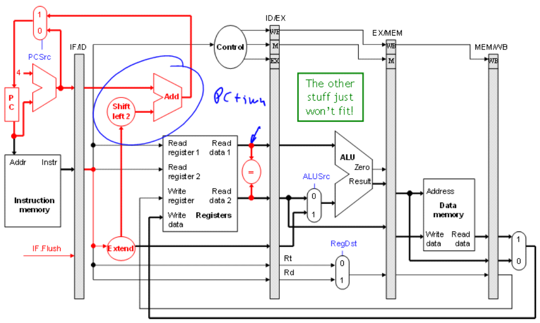
- We can actually decide the branch a little earlier, in ID instead of EX.
  - Our sample instruction set has only a BEQ.
  - ~~We can add a small~~ [illegible] ~~the~~ [illegible] the ~~source register~~ [illegible]
- Then we would only need to flush one instruction on a misprediction.



Implementing flushes

- We must flush one instruction (in its IF stage) if the previous instruction is BEQ and its two source registers are equal.
- We can flush an instruction from the IF stage by replacing it in the IF/ID pipeline register with a harmless nop instruction.
  - MIPS uses `sll $0, $0, 0` as the nop instruction.
  - This happens to have a binary encoding of all 0s: 0000 .... 0000.
- Flushing introduces a bubble into the pipeline, which represents the one-cycle delay in taking the branch.
- The IF.Flush control signal shown on the next page implements this idea, but no details are shown in the diagram.

Branching *without* forwarding and load stalls



Timing

branch delay slot

- If no prediction:

IF ID EX MEM WB
 IF IF ID EX MEM WB --- lost 1 cycle

beq
sub
add
Label
label: sub

- If prediction:
  - If Correct
    - IF ID EX MEM WB
    - IF ID EX MEM WB -- no cycle lost
  - If Misprediction:
    - IF ID EX MEM WB
    - IF0 IF1 ID EX MEM WB --- 1 cycle lost

Summary of Hazards/Stalls/Branches

- Three kinds of hazards conspire to make pipelining difficult.
- Structural hazards result from not having enough hardware available to execute multiple instructions simultaneously.
  - These are avoided by adding more functional units (e.g., more adders or memories) or by redesigning the pipeline stages.
- Data hazards can occur when instructions need to access registers that haven't been updated yet.
  - Hazards from R-type instructions can be avoided with forwarding.
  - Loads can result in a "true" hazard, which must stall the pipeline.
- Control hazards arise when the CPU cannot determine which instruction to fetch next.
  - We can minimize delays by doing branch tests earlier in the pipeline.
  - We can also take a chance and predict the branch direction, to make the most of a bad situation.

Performance

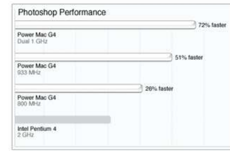
- Now we'll discuss issues related to performance:
  - Latency/Response Time/Execution Time vs. Throughput
  - How do you make a reasonable performance comparison?
  - The 3 components of CPU performance
  - The 2 laws of performance

Why know about performance

- Purchasing Perspective:
  - Given a collection of machines, which has the
    - Best Performance?
    - Lowest Price?
    - Best Performance/Price?
- Design Perspective:
  - Faced with design options, which has the
    - Best Performance Improvement?
    - Lowest Cost?
    - Best Performance/Cost ?
- Both require
  - Basis for comparison
  - Metric for evaluation

Many possible definitions of performance

- Every computer vendor will select one that makes them look good. How do you make sense of conflicting claims?



AMD

Q: *Why do end users need a new performance metric?*
A: End users who rely only on megahertz as an indicator for performance do not have a complete picture of PC processor performance and may pay the price of missed expectations.

Two notions of performance

Plane	DC to Paris	Speed	Passengers	Throughput (pmph)
747	6.5 hours	610 mph	470	286,700
Concorde	3 hours	1350 mph	132	178,200

- Which has higher performance?
  - Depends on the metric
    - Time to do the task (Execution Time, Latency, Response Time)
    - Tasks per unit time (Throughput, Bandwidth)
  - Response time and throughput are often in opposition

Some Definitions

- Performance is in units of things/unit time
  - E.g., Hamburgers/hour
  - Bigger is better
- If we are primarily concerned with response time
  - $\text{Performance}(x) = \dfrac{1}{\text{execution\_time}(x)}$
- Relative performance: "X is N times faster than Y"

$$N = \frac{\text{Performance}(X)}{\text{Performance}(Y)} = \frac{\text{execution\_time}(Y)}{\text{execution\_time}(X)}$$

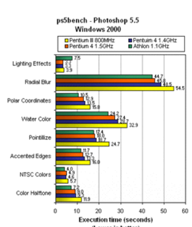
Basis of Comparison

- When comparing systems, need to fix the workload
  - Which workload?

Workload	Pros	Cons
Actual Target Workload	Representative	Very specific Non-portable Difficult to run/measure
Full Application Benchmarks	Portable Widely used Realistic	Less representative
Small "Kernel" or "Synthetic" Benchmarks	Easy to run Useful early in design	Easy to "fool"
Microbenchmarks	Identify peak capability and potential bottlenecks	Real application performance may be much below peak

Benchmarking

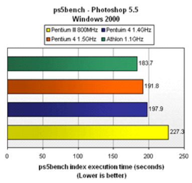
- Some common benchmarks include:
  - Adobe Photoshop for image processing
  - BAPCo Sysmark for office applications
  - Unreal Tournament 2003 for 3D games
  - SPEC2000 for CPU performance
- The best way to see how a system performs for a variety of programs is to just show the execution times of all of the programs.
- Here are execution times for several different Photoshop 5.5 tasks, from http://www.tech-report.com



Summarizing performance

- Summarizing performance with a single number can be misleading—just like summarizing four years of school with a single GPA!
- If you must have a single number, you could sum the execution times.

  This example graph displays the total execution time of the individual tests from the previous page.



- A similar option is to find the average of all the execution times.

  For example, the 800MHz Pentium III (in yellow) needed 227.3 seconds to run 21 programs, so its average execution time is 227.3/21 = 10.82 seconds.
- A weighted sum or average is also possible, and lets you emphasize some benchmarks more than others.

The components of execution time

- Execution time can be divided into two parts.
  - User time is spent running the application program itself.
  - System time is when the application calls operating system code.
- The distinction between user and system time is not always clear, especially under different operating systems.
- The Unix time command shows both.

```
salary.125 > time distill 05-examples.ps
Distilling 05-examples.ps (449,119 bytes)
10.8 seconds (0:11)
 449,119 bytes PS  =>  94,999 bytes PDF  (21%)
10.61u 0.98s 0:15.15 76.5%
```

User time

System time

"Wall clock" time (including other processes)

CPU usage = (User + System) / Total

Three Components of CPU Performance

CPU $\text{time}_{X,P}$ = Instructions executed_P * $\text{CPI}_{X,P}$ * Clock cycle time_X

Cycles Per Instruction

Instructions Executed

- Instructions executed:
  - We are not interested in the static instruction count, or how many lines of code are in a program.
  - Instead we care about the dynamic instruction count, or how many instructions are actually executed when the program runs.
- There are three lines of code below, but the number of instructions executed would be 2001.

```
         li   $a0, 1000
Ostrich: sub  $a0, $a0, 1
         bne  $a0, $0, Ostrich
```

CPI (Review)

- The average number of clock cycles per instruction, or CPI, is a function of the machine and program.
  - The CPI depends on the actual instructions appearing in the program—a floating-point intensive application might have a higher CPI than an integer-based program.
  - It also depends on the CPU implementation. For example, a Pentium can execute the same instructions as an older 80486, but faster.
- Initially we assumed each instruction took one cycle, so we had CPI = 1.
  - The CPI can be >1 due to memory stalls and slow instructions.
  - The CPI can be <1 on machines that execute more than 1 instruction per cycle (superscalar).

Execution time, again

CPU $\text{time}_{X,P}$ = Instructions executed_P * $\text{CPI}_{X,P}$ * Clock cycle time_X

- The easiest way to remember this is match up the units:

$$\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Clock cycles}}{\text{Instructions}} * \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Make things faster by making any component smaller!!

	Program	Compiler	ISA	Organization	Technology
Instruction Executed					
CPI					
Clock Cycle TIme					

- Often easy to reduce one component by increasing another

Example: Comparing across ISAs

- Intel's Itanium (IA-64) ISA is designed to facilitate executing multiple instructions per cycle. If an Itanium processor achieves an average CPI of .3 (3 instructions per cycle), how much faster is it than a Pentium4 (which uses the x86 ISA) with an average CPI of 1? (assume same freq)

  a) Itanium is three times faster
  b) Itanium is one third as fast
  c) Not enough information

Improving CPI

- Many processor design techniques we'll see improve CPI
  - Often they only improve CPI for certain types of instructions

$$CPI = \sum_{i=1}^{n} CPI_i \times F_i \quad \text{where } F_i = \frac{I_i}{\text{Instruction Count}}$$

- Fi = Fraction of instructions of type i

  - First Law of Performance:

Make the common case fast

Example: CPI improvements

- Base Machine:

Op Type	Freq (fi)	Cycles	CPIi
ALU	50%	3	
Load	20%	5	
Store	10%	3	
Branch	20%	2	

- How much faster would the machine be if:
  - we added a cache to reduce average load time to 3 cycles?
  - we added a branch predictor to reduce branch time by 1 cycle?
  - we could do two ALU operations in parallel?

Amdahl's Law

- Amdahl's Law states that optimizations are limited in their effectiveness.

$$\text{Execution time after improvement} = \frac{\text{Time affected by improvement}}{\text{Amount of improvement}} + \text{Time unaffected by improvement}$$

- For example, doubling the speed of floating-point operations sounds like a great idea. But if only 10% of the program execution time T involves floating-point code, then the overall performance improves by just 5%.

$$\text{Execution time after improvement} = \frac{0.10\ T}{2} + 0.90\ T = 0.95\ T$$

- What is the maximum speedup from improving floating point?

  - Second Law of Performance:

Make the fast case common

Summary

- Performance is one of the most important criteria in judging systems.
- There are two main measurements of performance.
  - Execution time is what we'll focus on.
  - Throughput is important for servers and operating systems.
- Our main performance equation explains how performance depends on several factors related to both hardware and software.

$$\text{CPU time}_{X,P} = \text{Instructions executed}_P * \text{CPI}_{X,P} * \text{Clock cycle time}_X$$

- It can be hard to measure these factors in real life, but this is a useful guide for comparing systems and designs.
- Amdahl's Law tell us how much improvement we can expect from specific enhancements.
- The best benchmarks are real programs, which are more likely to reflect common instruction mixes.