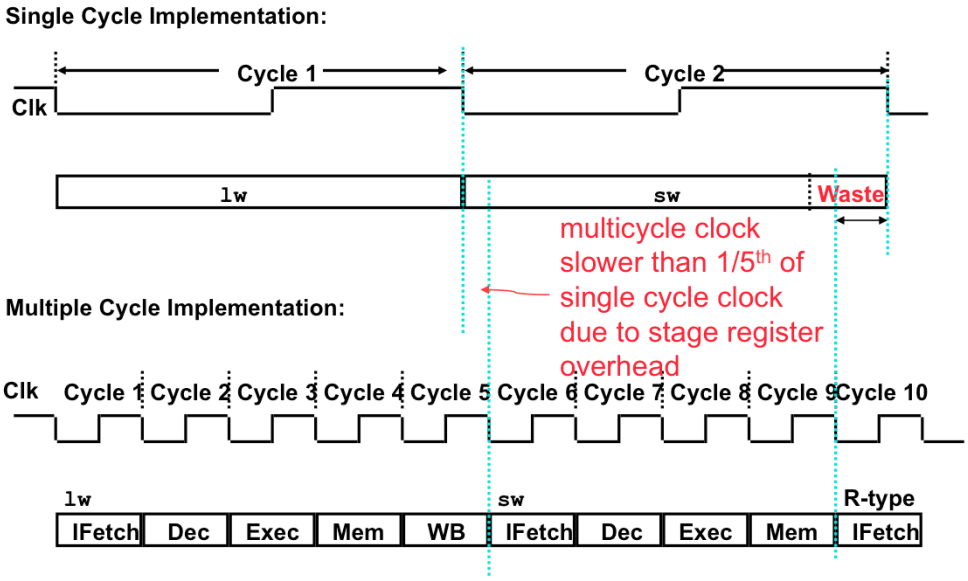
Basic MIPS Pipelining Review

[Adapted from Mary Jane Irwin for
Computer Organization and Design,
Patterson & Hennessy, © 2005, UCB]

Other handouts

To handout next time

Review: Single Cycle vs. Multiple Cycle Timing



Here are the timing diagrams showing the differences between the single cycle and multiple cycle.

In the multiple clock cycle implementation, we cannot start executing the store until Cycle 6 because we must wait for the load instruction to complete.

Similarly, we cannot start the execution of the R-type instruction until the store instruction has completed its execution in Cycle 9.

In the Single Cycle implementation, the cycle time is set to accommodate the longest instruction, the Load instruction.

Consequently, the cycle time for the Single Cycle implementation can be five times longer than the multiple cycle implementation.

How Can We Make It Even Faster?

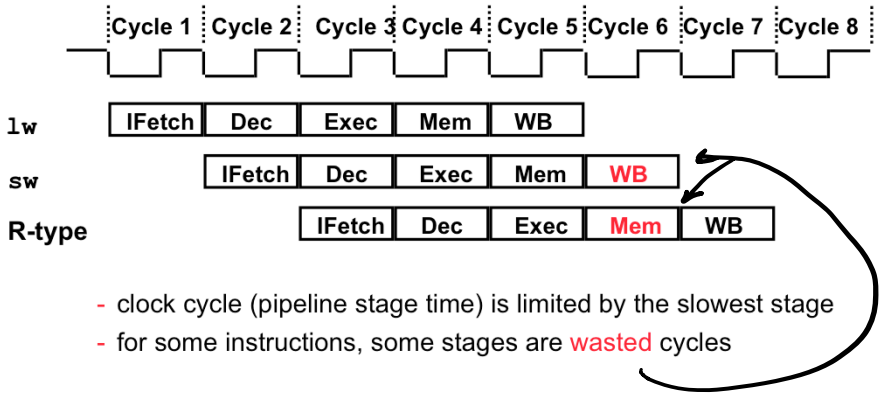
- Split the multiple instruction cycle into smaller and smaller steps
  - There is a point of diminishing returns where as much time is spent loading the state registers as doing the work
- Start fetching and executing the next instruction before the current one has completed
  - Pipelining – (all?) modern processors are pipelined for performance
  - Remember *the* performance equation:

    CPU time = CPI * CC * IC
- Fetch (and execute) more than one instruction at a time
  - Superscalar processing – stay tuned

Laundry example: collecting laundry basket, washing machine, drier, folding, storing.

A Pipelined MIPS Processor

- Start the next instruction before the current one has completed
  - improves throughput - total amount of work done in a given time
  - instruction latency (execution time, delay time, response time - time from the start of an instruction to its completion) is *not* reduced



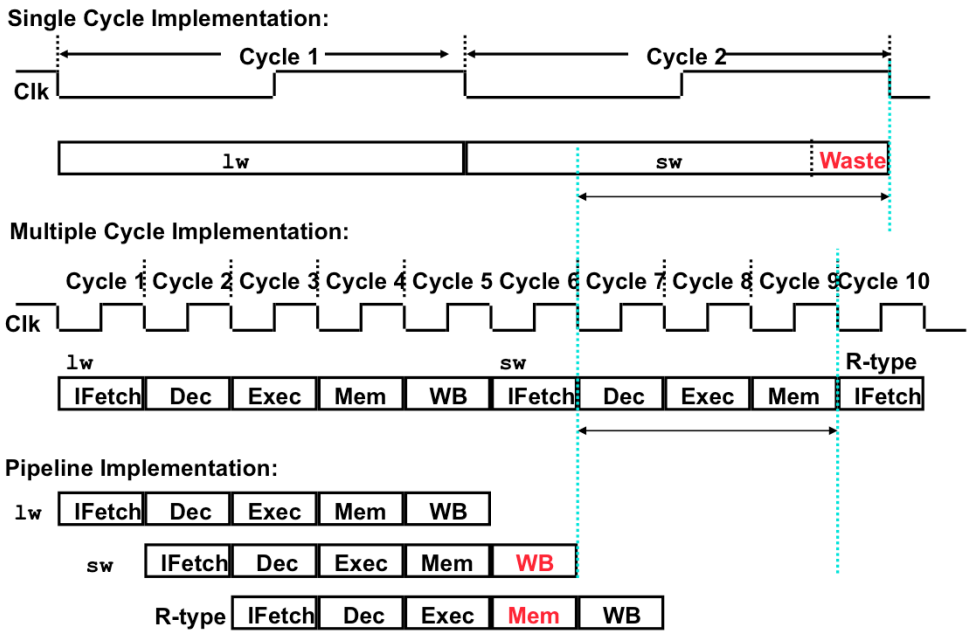
- clock cycle (pipeline stage time) is limited by the slowest stage
- for some instructions, some stages are wasted cycles

Latency = execution time (delay or response time) – the total time from start to finish of ONE instruction

For processors one important measure is THROUGHPUT (or the execution bandwidth) – the total amount of work done in a given amount of time

For memories one important measure is BANDWIDTH – the amount of information communicated across an interconnect (e.g., bus) per unit time; the number of operations performed per second (the WIDTH of the operation and the RATE of the operation)

Single Cycle, Multiple Cycle, vs. Pipeline

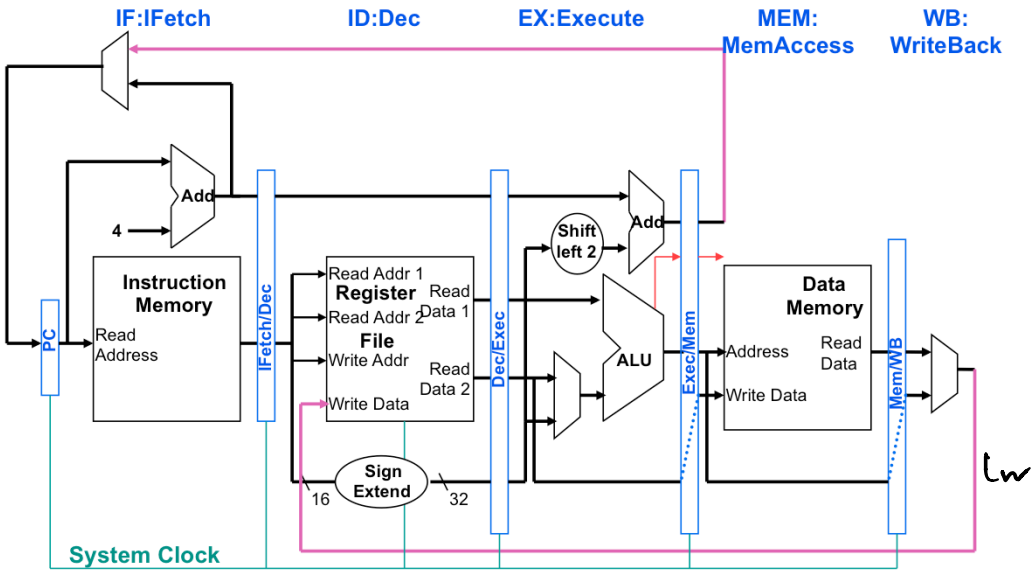


Here are the timing diagrams showing the differences between the single cycle, multiple cycle, and pipeline implementations.

For example, in the pipeline implementation, we can finish executing the Load, Store, and R-type instruction sequence in seven cycles.

MIPS Pipeline Datapath Modifications

- What do we need to add/modify in our MIPS datapath?
  - State registers between each pipeline stage to isolate them



Note two exceptions to right-to-left flow

1. WB that writes the result back into the register file in the middle of the datapath
2. Selection of the next value of the PC, one input comes from the calculated branch address from the MEM stage

Only later instructions in the pipeline can be influenced by these two REVERSE data movements.

The first one (WB to ID) leads to data hazards.

The second one (MEM to IF) leads to control hazards.

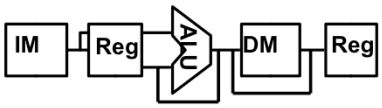
All instructions must update some state in the processor – the register file, the memory, or the PC – so separate pipeline registers are redundant to the state that is updated (not needed).

PC can be thought of as a pipeline register: the one that feeds the IF stage of the pipeline. Unlike all of the other pipeline registers, the PC is part of the visible architecture state – its content must be saved when an exception occurs (the contents of the other pipe registers are discarded).

Pipelining the MIPS ISA

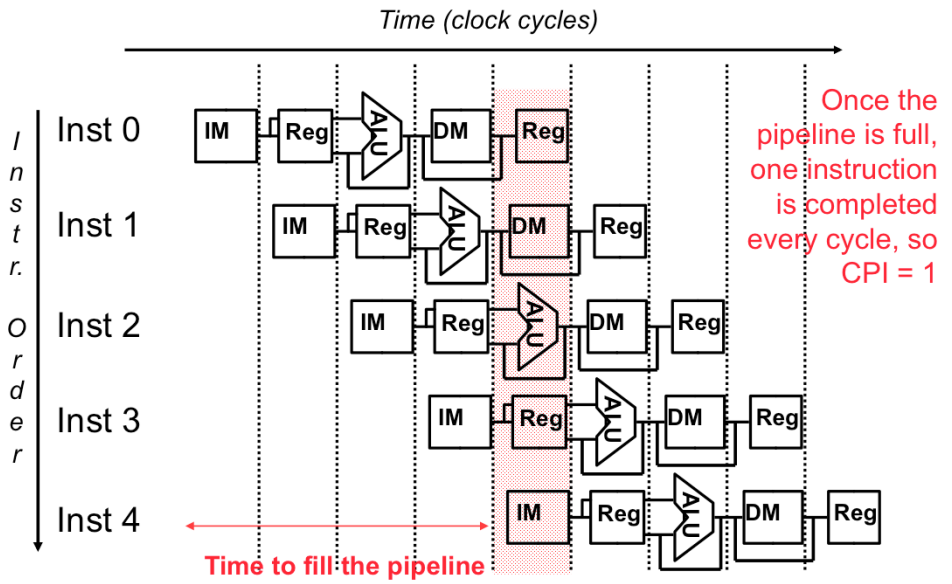
- What makes it easy
  - all instructions are the same length (32 bits)
    - can fetch in the 1st stage and decode in the 2nd stage
  - few instruction formats (three) with symmetry across formats
    - can begin reading register file in 2nd stage
  - memory operations can occur only in loads and stores
    - can use the execute stage to calculate memory addresses
  - each MIPS instruction writes at most one result (i.e., changes the machine state) and does so near the end of the pipeline (MEM and WB)
- What makes it hard
  - structural hazards: what if we had only one memory?
  - control hazards: what about branches?
  - data hazards: what if the instruction input operands depend on the output of the previous instruction?

Graphically Representing MIPS Pipeline



- Can help with answering questions like:
  - How many cycles does it take to execute this code?
  - What is the ALU doing during cycle 4?
  - Is there a hazard, why does it occur, and how can it be fixed?

Why Pipeline? For Performance!



IM (IF) = instruction memory and PC fetch

Reg not (ID) = instruction decode (ID stage) and register file read + sign extender

ALU (EX)= execution

DM (MEM)= data memory access

Reg (WB)= register file write

CPI=clock cycles per instruction

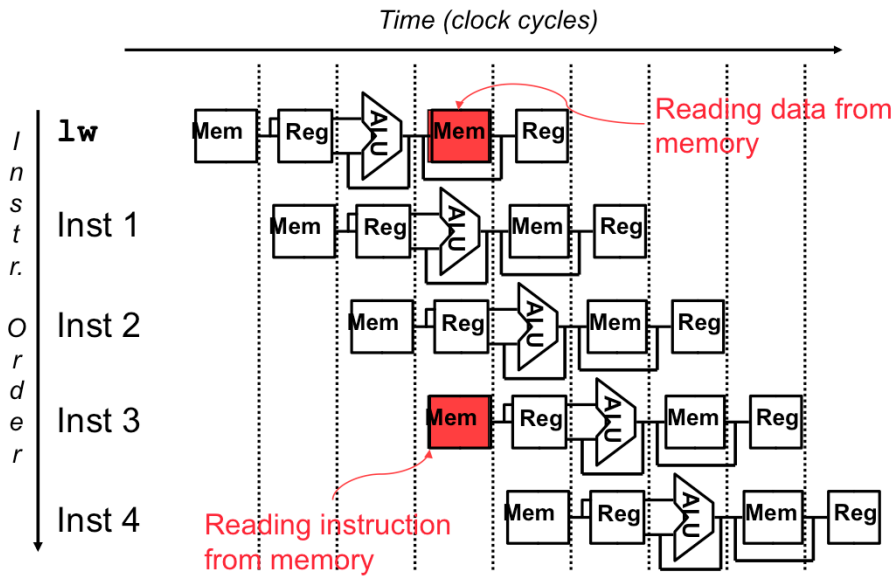
Can Pipelining Get Us Into Trouble?

- Yes: Pipeline Hazards
  - structural hazards: attempt to use the same resource by two different instructions at the same time
  - data hazards: attempt to use data before it is ready
    - An instruction source operand(s) is produced by a prior instruction still in the pipeline
  - control hazards: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
    - branch instructions

- Can always resolve hazards by waiting
  - pipeline control must detect the hazard
  - and take action to resolve hazards

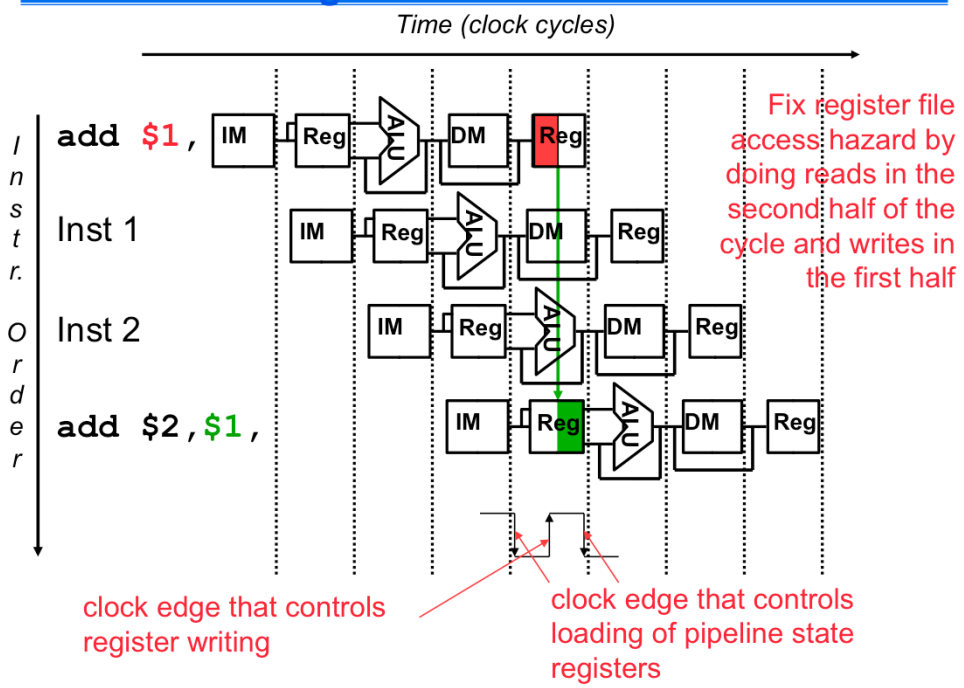
Note that data hazards can come from R-type instructions or lw instructions

A Single Memory Would Be a Structural Hazard



- Fix with separate instr and data memories (I$ and D$)

How About Register File Access?

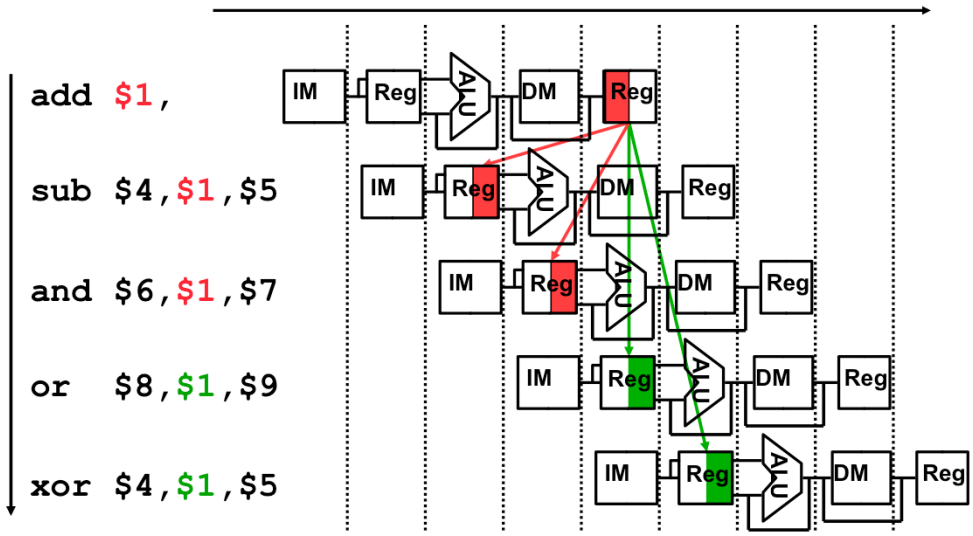


For lecture

Define register reads to occur in the second half of the cycle and register writes in the first half

Register Usage Can Cause Data Hazards

- Dependencies backward in time cause hazards

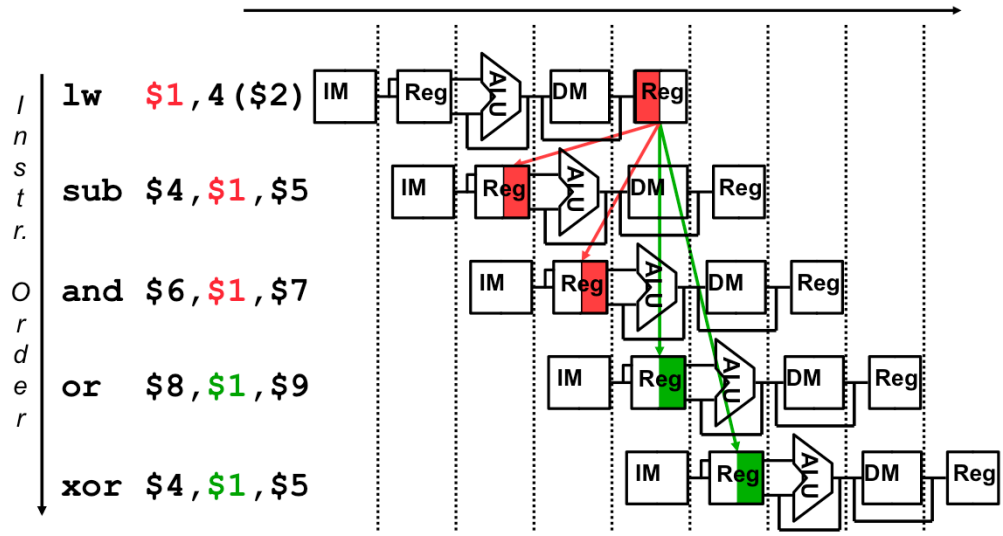


- Read before write data hazard

For lecture

Loads Can Cause Data Hazards

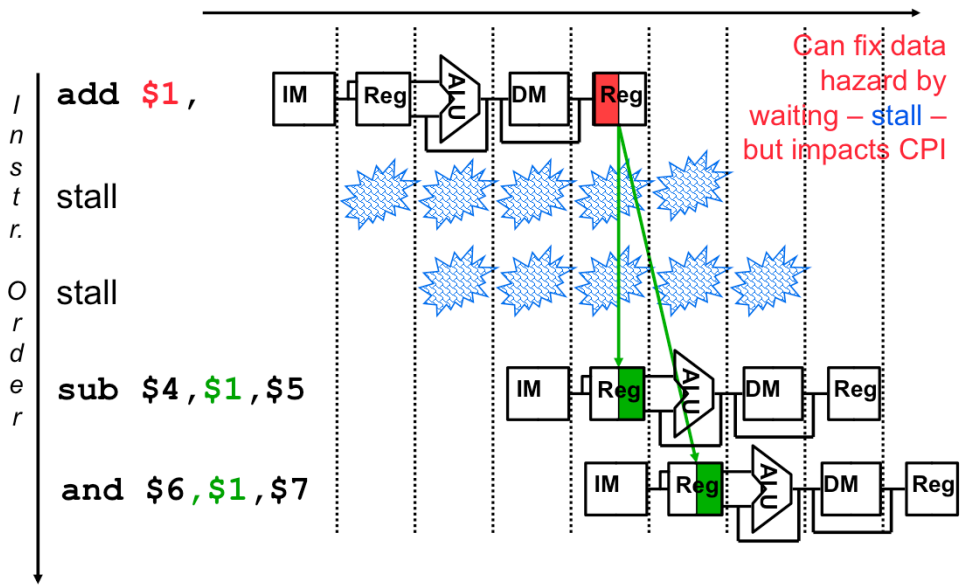
- Dependencies backward in time cause hazards



- Load-use data hazard

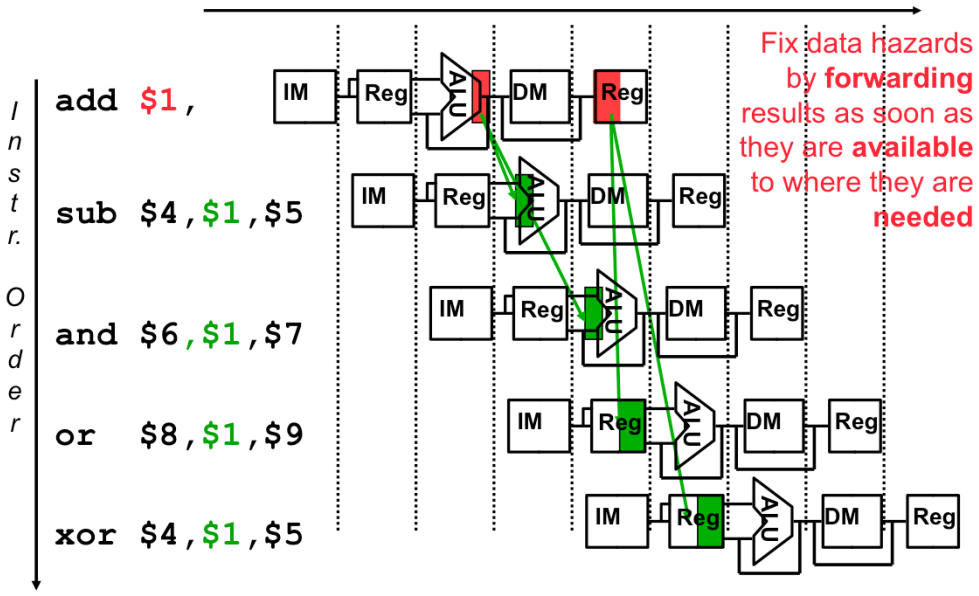
Note that lw is just another example of register usage (beyond ALU ops)

One Way to "Fix" a Data Hazard



To stall: hinhalten, verzögern, Zeit schinden

Another Way to "Fix" a Data Hazard

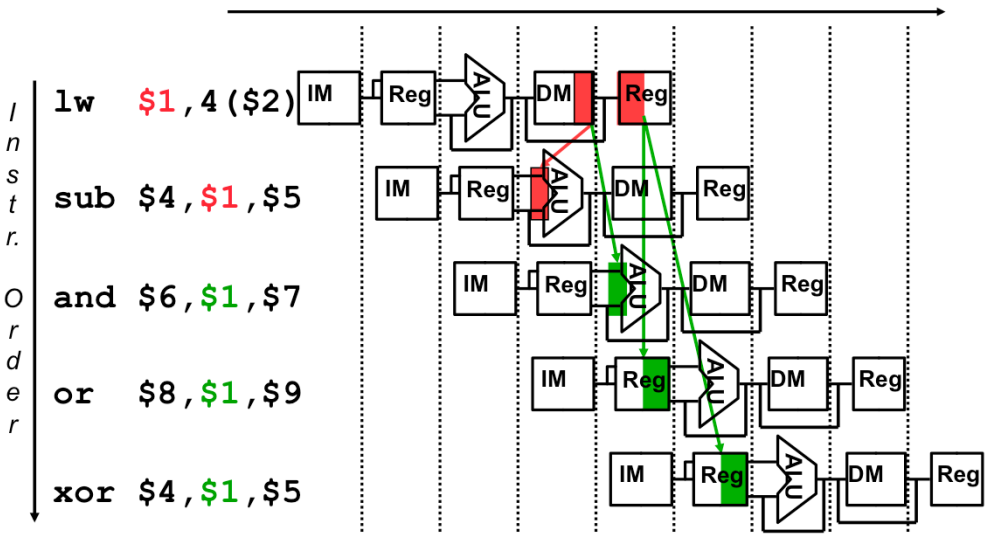


For lecture

Forwarding paths are valid only if the destination stage is later in time than the source stage.

Forwarding is harder if there are multiple results to forward per instruction or if they need to write a result early in the pipeline

Forwarding with Load-use Data Hazards



- Will still need one stall cycle even with forwarding

For lecture

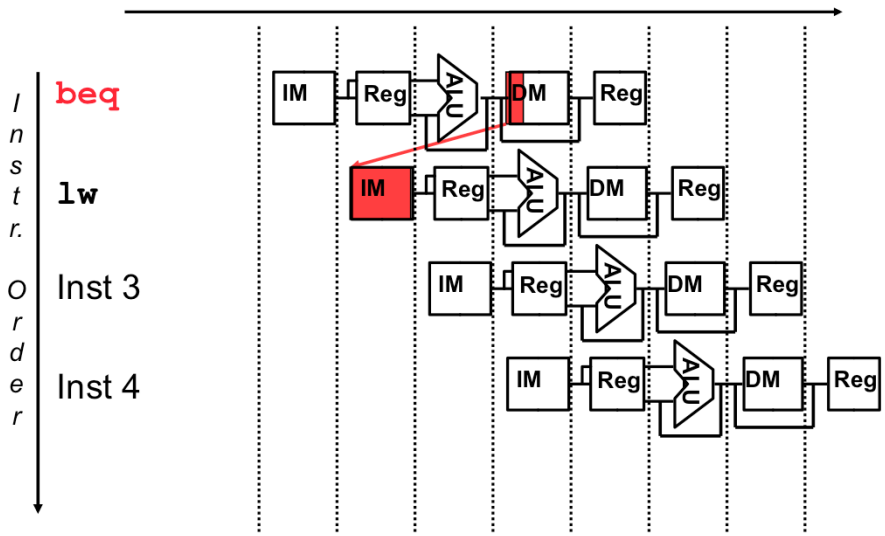
Note that lw is just another example of register usage (beyond ALU ops)

Need to stall even with forwarding when data hazard involves a load

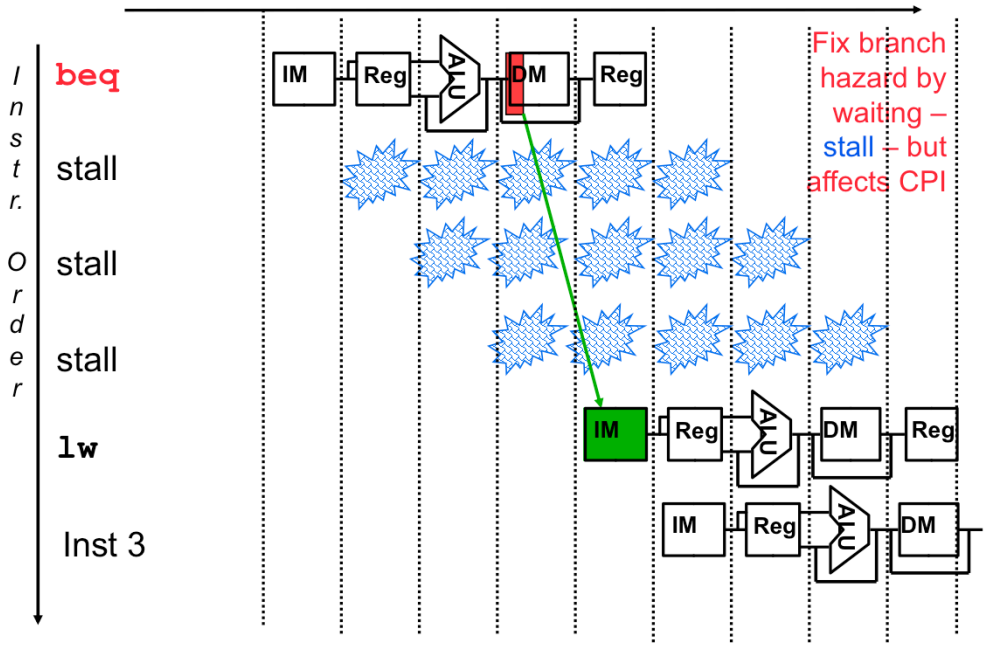
Another way to avoid stall cycles: Sometimes the order of the instruction is not fix, so we can switch them around(The compilee does this automatically)

Branch Instructions Cause Control Hazards

- Dependencies backward in time cause hazards



One Way to "Fix" a Control Hazard

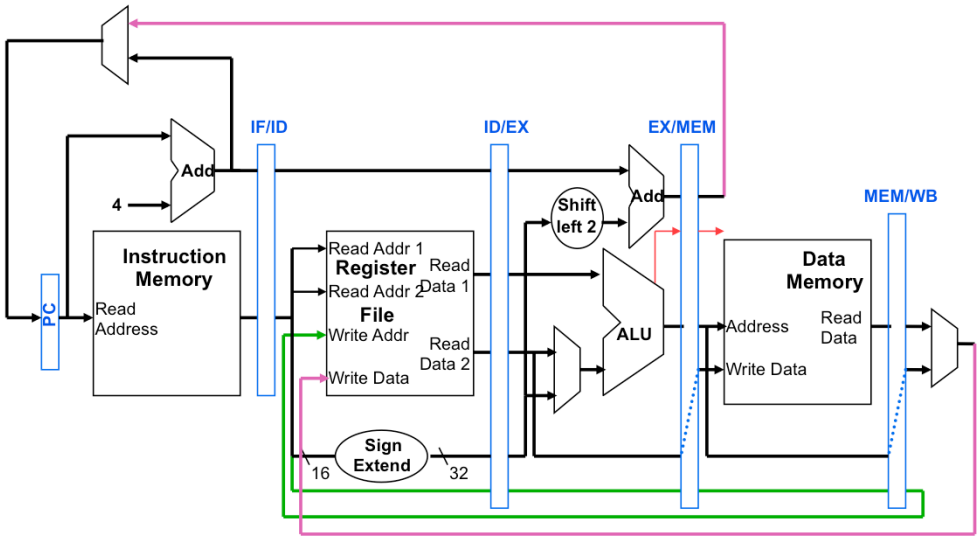


Another "solution" is to put in enough extra hardware so that we can test registers, calculate the branch address, and update the PC during the second stage of the pipeline. That would reduce the number of stalls to only one.

A third approach is prediction of branches: always predict that branches will be untaken. When right, the pipeline proceeds at full speed. When wrong, have to stall (and make sure nothing completes – changes machine state – that shouldn't have). Will talk about these options in more detail in the next lecture.

Corrected Datapath to Save RegWrite Addr

- Need to preserve the destination register address in the pipeline state registers



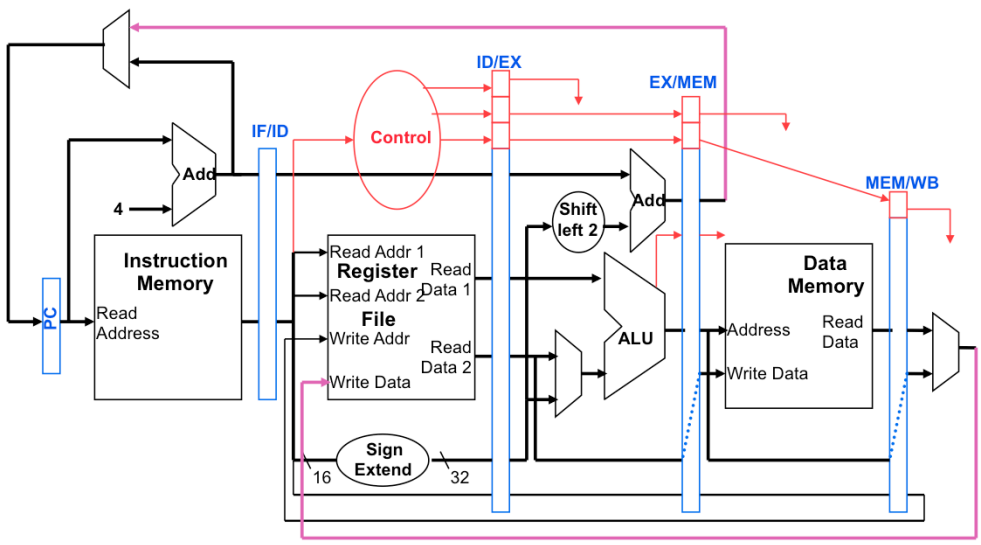
Issue for load instruction (by the time it gets to read the register, the address of the register is lost

WriteAddr geht auch durch die Pipeline. Damit liegt die richtige Adresse zum richtigen Zeitpunkt an.

WriteAddr also passes through the pipeline. Putting us in the right place at the right time.

MIPS Pipeline Control Path Modifications

- All control signals can be determined during Decode
  - and held in the state registers between pipeline stages



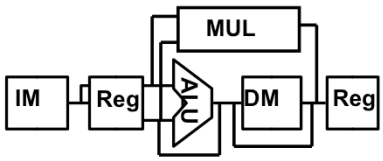
In general we must store some control registers for each stage

Control ist KEINE FSM. Nur boolsche Logik. Kann genau so wie bei der single Cycle Architektur gebaut werden.

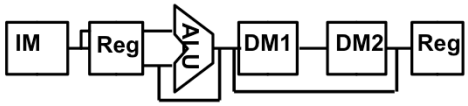
Control is NOT FSM. Only boolean logic. Can be built exactly as in the single cycle architecture.

Other Pipeline Structures Are Possible

- What about the (slow) multiply operation?
  - Make the clock twice as slow or …
  - let it take two cycles (since it doesn't use the DM stage)



- What if the data memory access is twice as slow as the instruction memory?
  - make the clock twice as slow or …
  - let data memory access take two cycles (and keep the same clock rate)



Note that we don't need the output of MUL until the WB cycle, so we can span two pipeline stages with the MUL hardware (so the multiplier is a two stage pipelined multiplier)

Sample Pipeline Alternatives

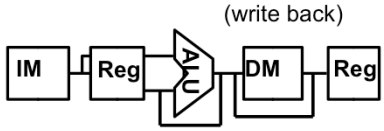
- ARM7

PC update
IM access

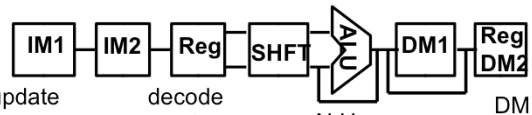
decode
reg
access

ALU op
DM access
shift/rotate
commit result
(write back)

- StrongARM-1



- XScale



PC update
BTB access
start IM access

IM access

decode
reg 1 access

shift/rotate
reg 2 access

ALU op

start DM access
exception

DM write
reg write

About branching prediction:

BTB: Branch Target Buffer: Branch hash tables mapping PC to branching target addresses. Before we had to wait until decode to know that at that PC we had a branching instruction.

Now we already have the target address too. Avoids idle stalling

Xscale is Intel‘s ARM version. Succerssor of StrongARM (still Intel).

Summary

- All modern day processors use pipelining
- Pipelining does not help latency of single task, it helps throughput of entire workload
- Potential speedup: a CPI of 1 and a fast CC
- Pipeline rate limited by slowest pipeline stage
  - Unbalanced pipe stages makes for inefficiencies
  - The time to "fill" pipeline and time to "drain" it can impact speedup for deep pipelines and short code runs
- Must detect and resolve hazards
  - Stalling negatively affects CPI (makes CPI less than the ideal of 1)