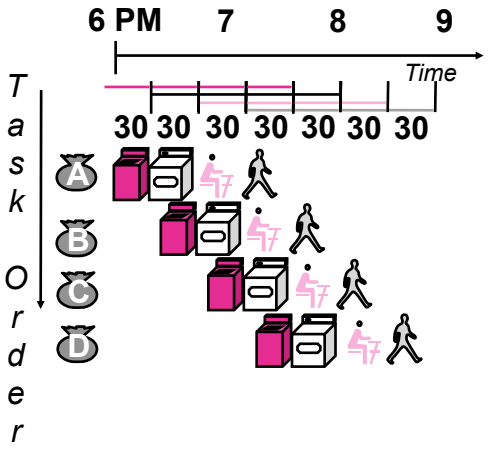
Lecture 12: Pipelined Processor

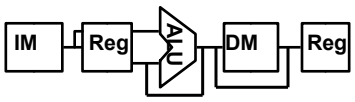
- Last time
  - Simple processor organization
  - Logic & control
  - Motivation & idea behind pipeling
- Today
  - Take QUIZ 8 over P&H 4.7-10, before 11:59pm today
  - Homework 4 due Thursday March 4, 2010
  - Pipelining in the real world

Pipelining Lessons



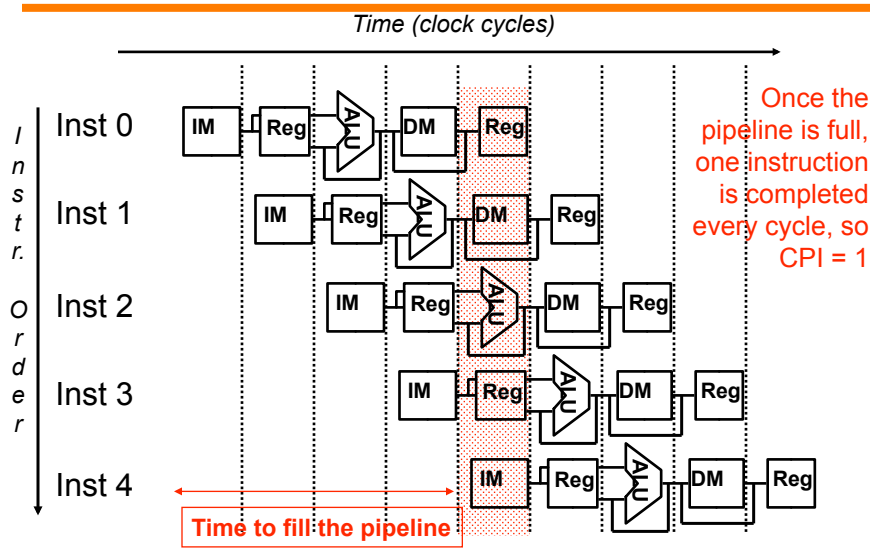
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup
- Stall for Dependences

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?

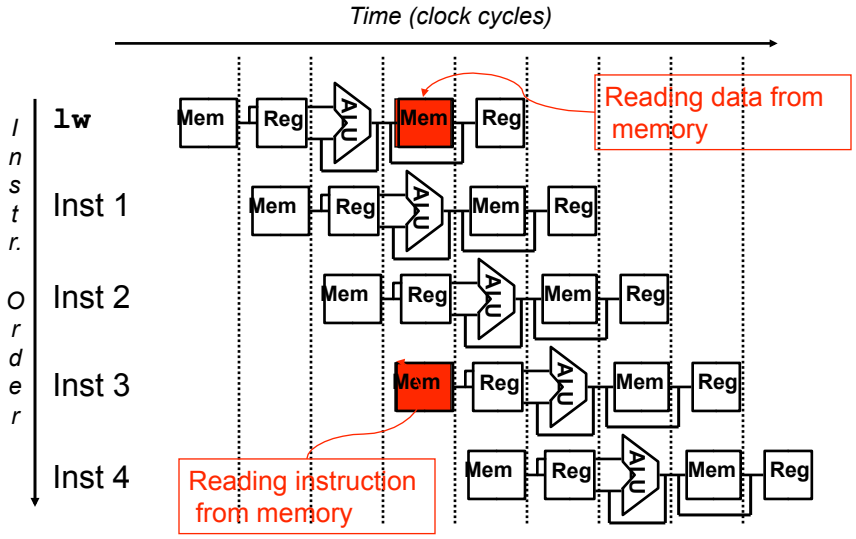
Why Pipeline? For Performance!



Can Pipelining Get Us Into Trouble?

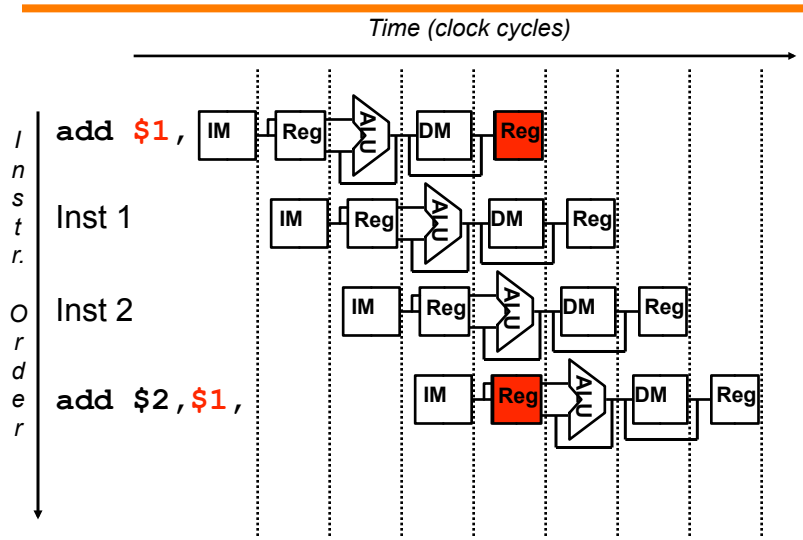
- Yes: Pipeline Hazards
  - structural hazards: attempt to use the same resource by two different instructions at the same time
  - data hazards: attempt to use data before it is ready
    - An instruction's source operand(s) are produced by a prior instruction still in the pipeline
  - control hazards: 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

A Single Memory Would Be a Structural Hazard

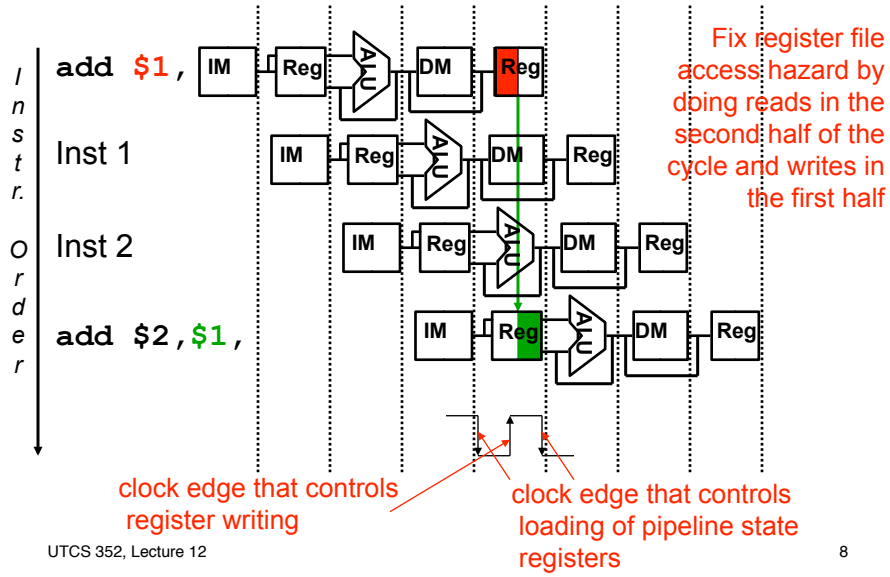


- Fix with separate instr and data memories (I$ and D$)

How About Register File Access?

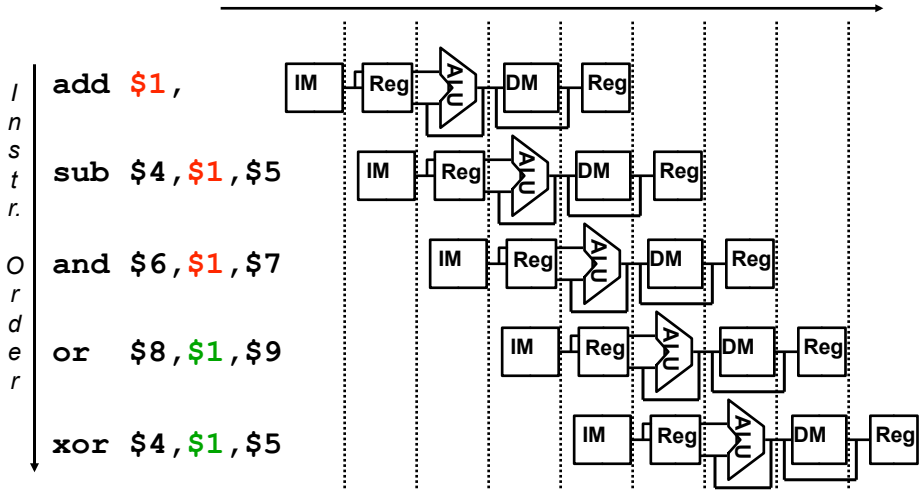


How About Register File Access?



Register Usage Can Cause Data Hazards

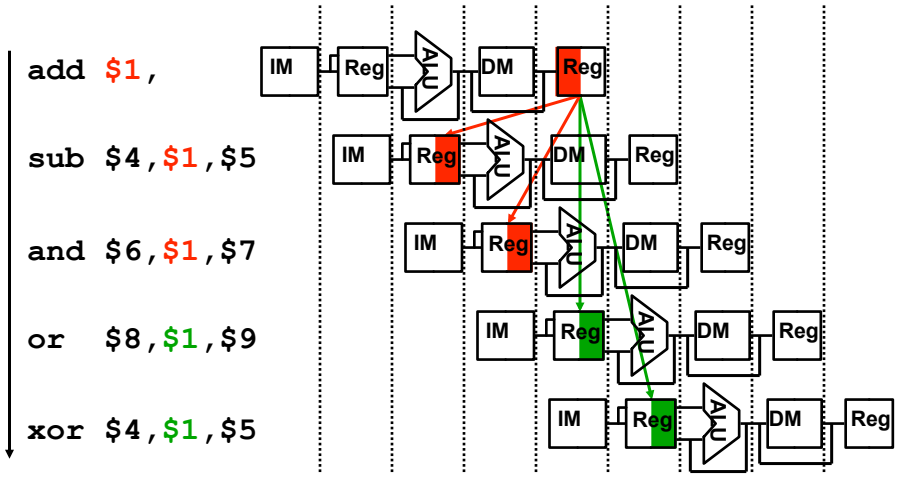
Dependencies backward in time cause hazards



Read before write is ready: data hazard

Register Usage Can Cause Data Hazards

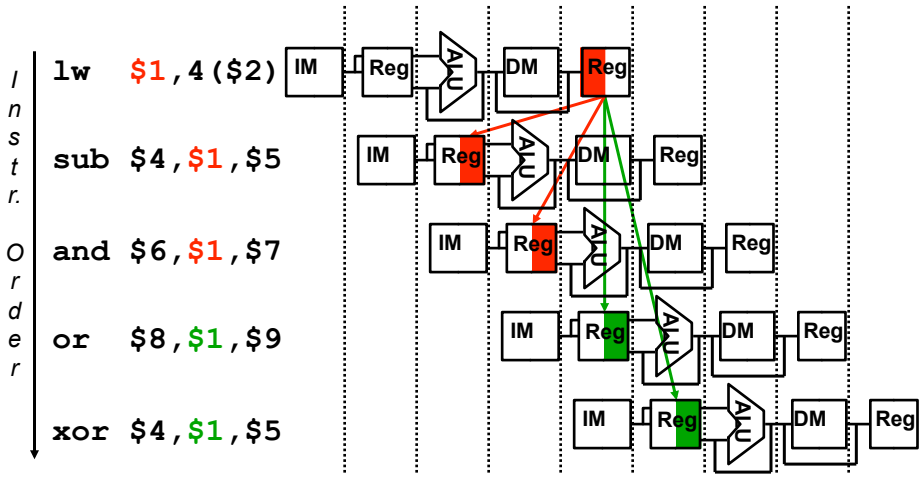
Dependencies backward in time cause hazards



Write-read data hazard

Loads Can Cause Data Hazards

Dependencies backward in time cause hazards

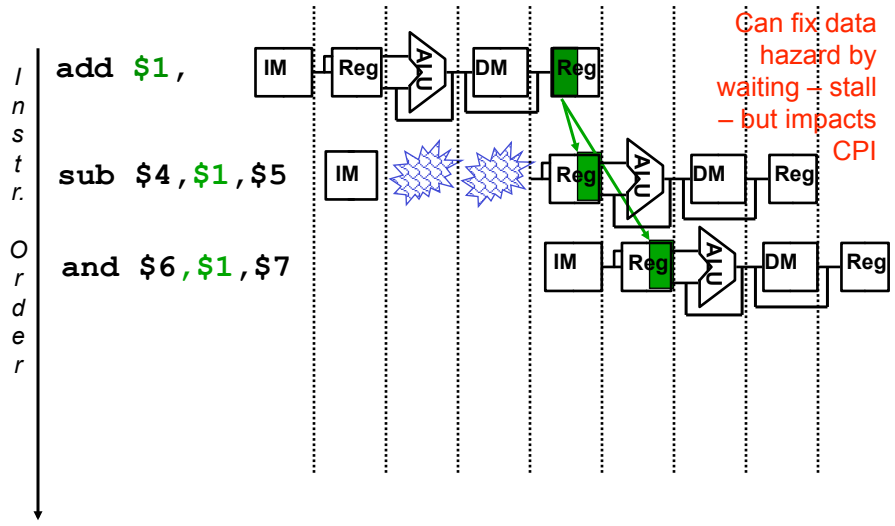


- Load-use data hazard

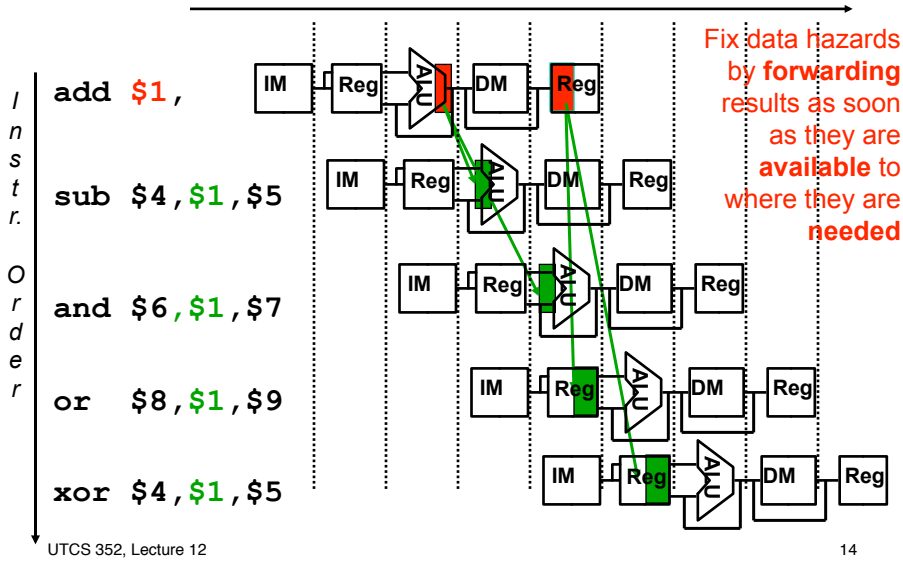
Resolving Hazards: Pipeline Stalls

- Can resolve any type of hazard
  - data, control, or structural
- Detect the hazard
- Freeze the pipeline up to the dependent stage until the hazard is resolved

One Way to "Fix" a Data Hazard



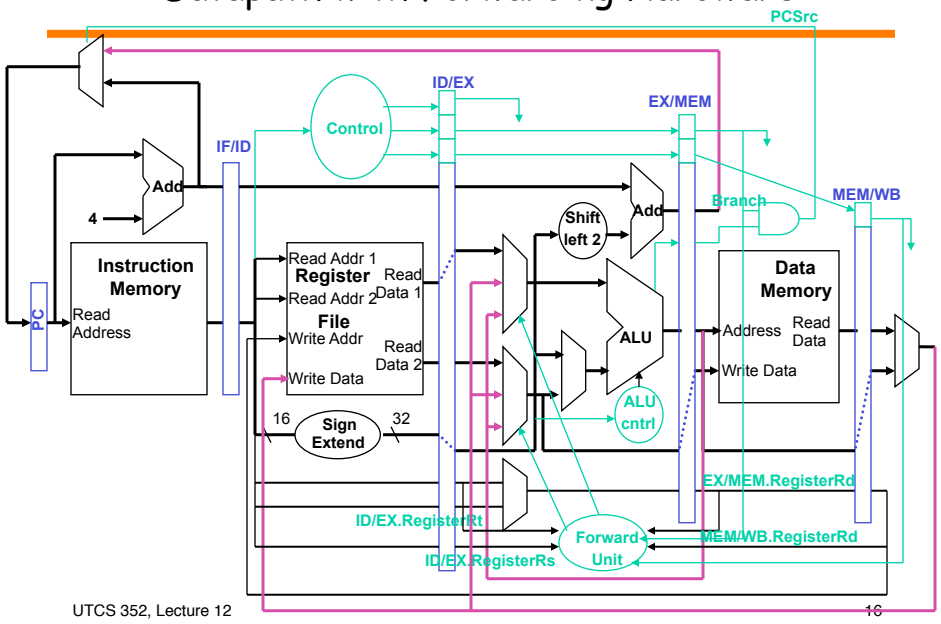
Another Way to "Fix" a Data Hazard



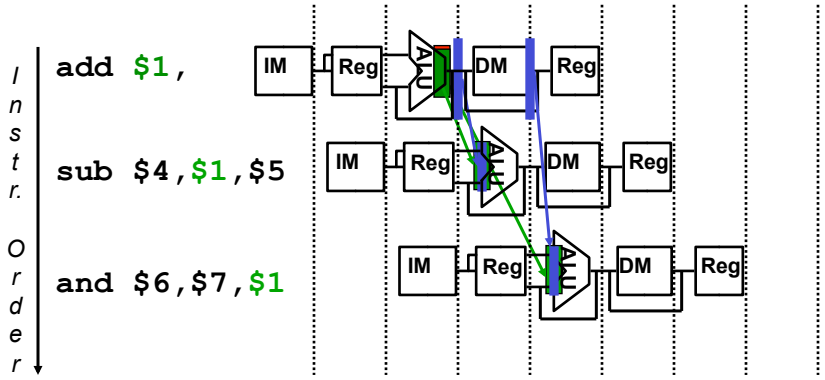
Data Forwarding (aka Bypassing)

- Take the result from the earliest point that it exists in any of the pipeline state registers and forward it to the functional units (e.g., the ALU) that need it that cycle
- For ALU functional unit: the inputs can come from any pipeline register
  - adding multiplexors to the inputs of the ALU
  - connecting the Rd write data in EX/MEM or MEM/WB to either (or both) of the EX's stage Rs and Rt ALU mux inputs
  - adding the proper control hardware to control the new muxes
- Other functional units may need forwarding logic (e.g., the DM)
- Forwarding can achieve a CPI of 1 even in the presence of data dependencies

Datapath with Forwarding Hardware



Forwarding Illustration



EX/MEM hazard forwarding

MEM/WB hazard forwarding

Data Forwarding Control Conditions

EX/MEM hazard:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 10
```

Forwards the result from the previous instr. to either input of the ALU

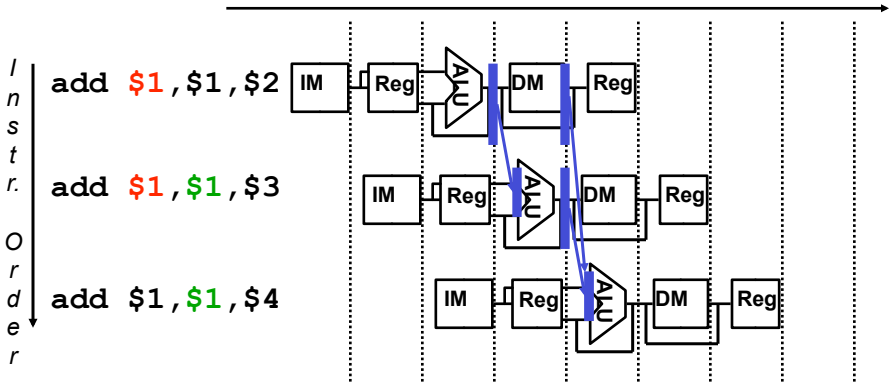
MEM/WB hazard:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 01
```

Forwards the result from the second previous instr. to either input of the ALU

Yet Another Complication!

- Another potential data hazard can occur when there is a conflict between the result of the WB stage instruction and the MEM stage instruction – which should be forwarded?

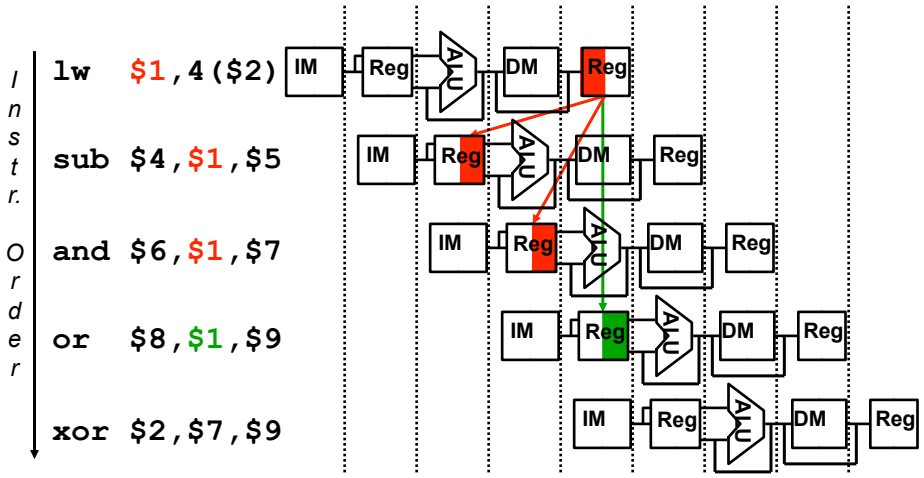


Corrected Data Forwarding Control Conditions

MEM/WB hazard:

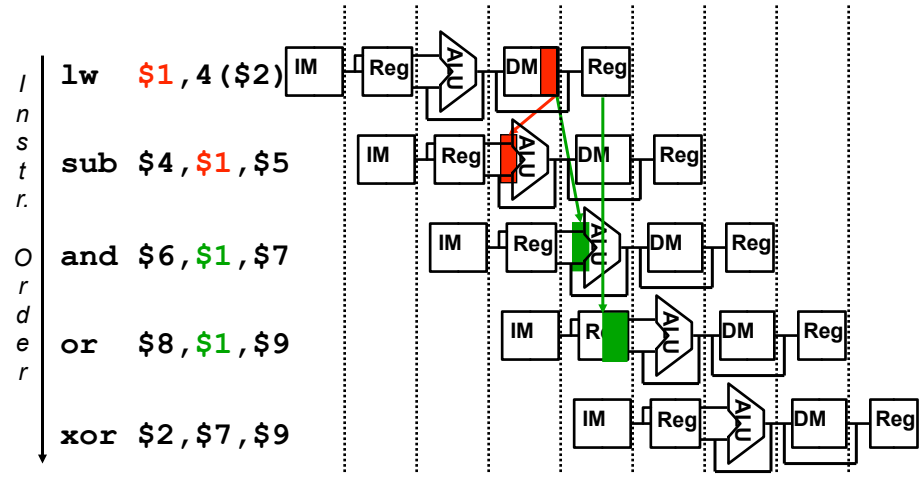
```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (EX/MEM.RegisterRd != ID/EX.RegisterRs)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (EX/MEM.RegisterRd != ID/EX.RegisterRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 01
```

Memory Data Hazards



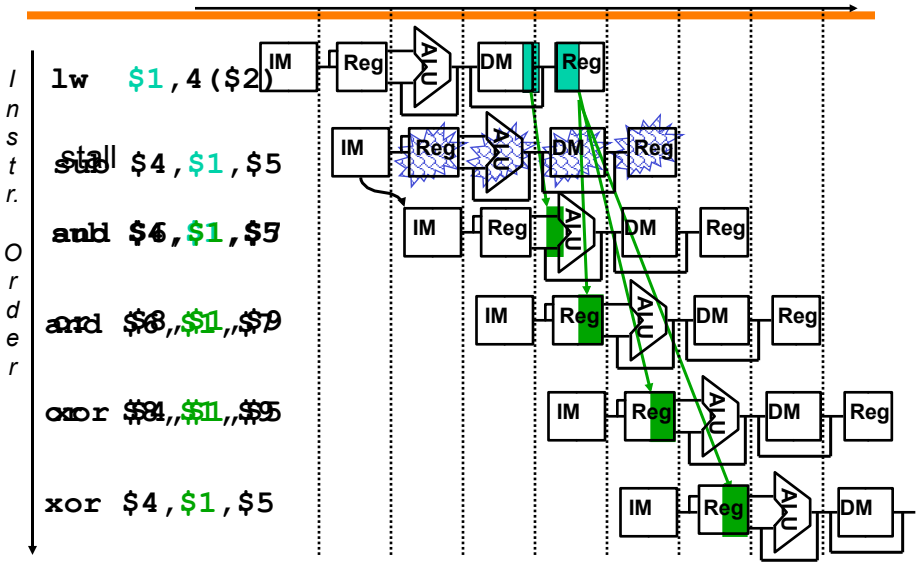
- Does forwarding solve all our problems?

Forwarding with Load-use Data Hazards



- still must stall one cycle even with forwarding!

Forwarding with Load-use Data Hazards



Load-use Hazard Detection Unit

- Need a Hazard detection Unit in the ID stage that inserts a stall between the load and its use

ID Hazard Detection

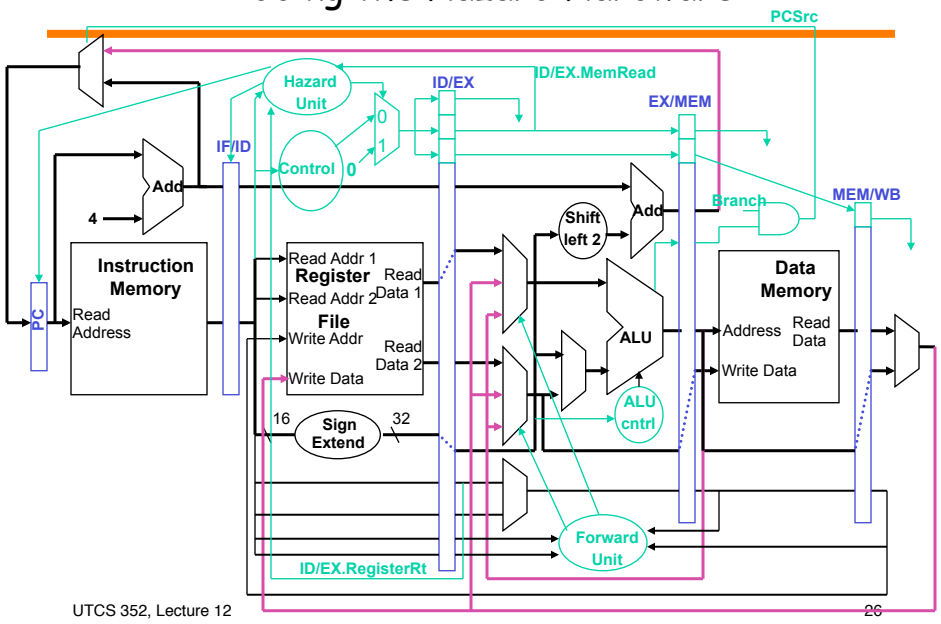
```
if (ID/EX.MemRead
and ((ID/EX.RegisterRt = IF/ID.RegisterRs)
or  (ID/EX.RegisterRt = IF/ID.RegisterRt)))
stall the pipeline
```

- The first line tests to see if the instruction now in the EX stage is a `lw`; the next two lines check to see if the destination register of the `lw` matches either source register of the instruction in the ID stage (the load-use instruction)
- After this one cycle stall, the forwarding logic can handle the remaining data hazards

Stall Hardware

- We have to detect this case & implement the stall
- Prevent the instructions in the IF and ID stages from progressing down the pipeline – done by preventing the PC register and the IF/ID pipeline register from changing
  - Hazard detection Unit controls the writing of the PC (`PC.write`) and IF/ID (`IF/ID.write`) registers
- Insert a "bubble" between the `lw` instruction (in the EX stage) and the load-use instruction (in the ID stage) (i.e., insert a `noop` in the execution stream)
  - Set the control bits in the EX, MEM, and WB control fields of the ID/EX pipeline register to 0 (`noop`). The Hazard Unit controls the mux that chooses between the real control values and the 0's.
- Let the `lw` instruction and the instructions after it in the pipeline (before it in the code) proceed normally down the pipeline

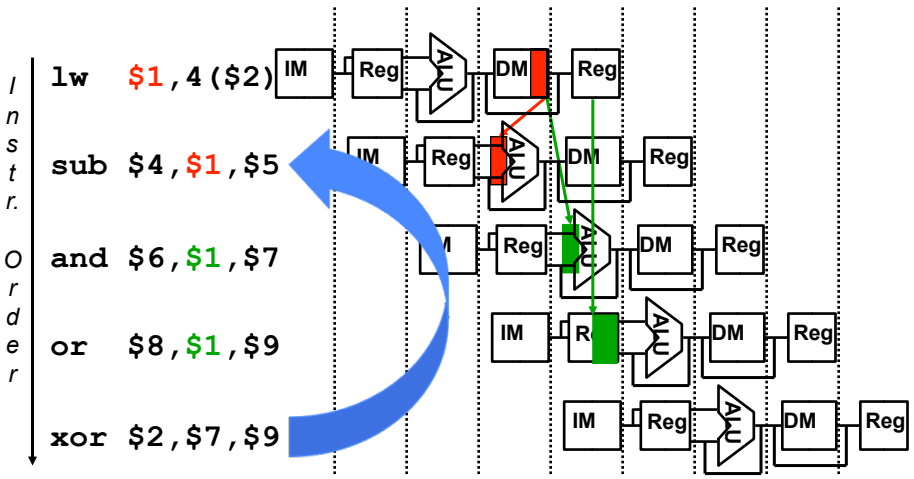
Adding the Hazard Hardware



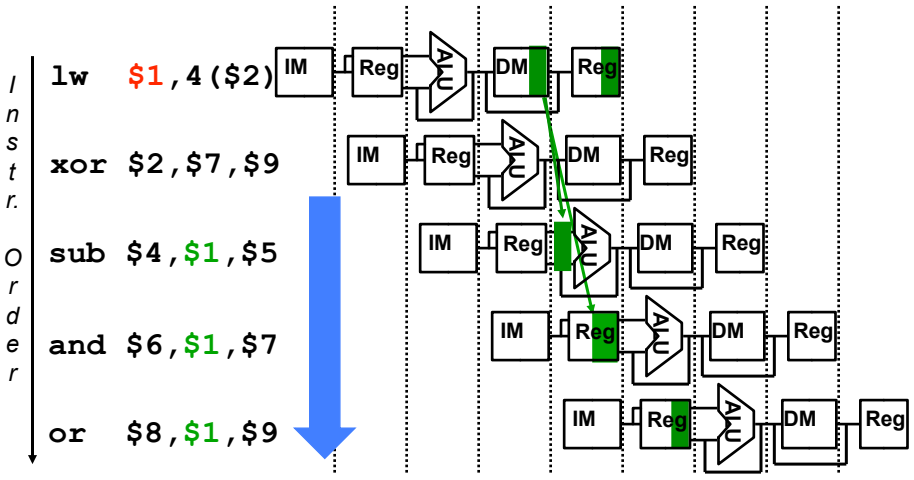
Compiler Instruction Scheduling

- The compiler can rearrange instructions, eliminating load-use hazard!
- Proebsting & Fischer (1991) show how to optimally schedule a straight line sequence of instructions, given sufficient registers and a delay of one pipeline stage.
- Approach
  - Build a dependence graph that describes the partial order of instruction definitions and uses
  - Schedule R independent loads (load; load; load; ..)
    - Each load requires a register,
    - thus R is the minimum number of live registers
  - Schedule operation independent of the previous load and another load in a pair (operation; load)

Compiler Scheduling to Avoid Load-use Data Hazards



Compiler Scheduling to Avoid Load-use Data Hazards

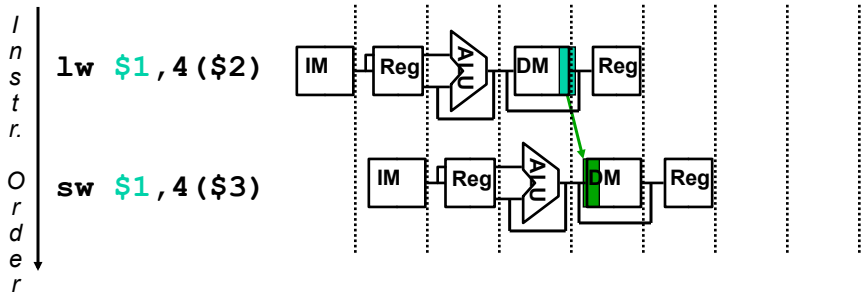


Types of Data Hazards

- RAW (read after write)
  - only hazard for 'fixed' pipelines
  - later instruction must *read* after earlier *write*
- WAW (write after write)
  - variable-length pipeline
  - later instruction must *write* after earlier *write*
- WAR (write after read)
  - pipelines with late read
  - later instruction must *write* after earlier *read*

Memory-to-Memory Copies

- For loads immediately followed by stores (memory -to-memory copies) can avoid a stall by adding forwarding hardware from the MEM/WB register to the data memory input.
  - Would need to add a Forward Unit and a mux to the memory access stage



Summary

- The real world of pipelining
  - Just stall
  - Forwarding for register and memory hazards
- Next Time
  - Prediction for control hazards
  - Multiple issue and out-of-order processors
  - Homework 4 due Thursday March 4, 2010
- Reading: P&H 4.11-15