

15. Performance, exceptions and advanced pipelining

EECS 370 – Introduction to Computer Organization – Winter 2009

Prof. Todd Austin & Prof. Marios Papaefthymiou

EECS Department
University of Michigan, Ann Arbor

© Austin & Papaefthymiou, 2009

The material in this presentation cannot be
copied in any form without our written permission

Announcements

- ❑ Exam regrade requests due today
 - Give them to me or one of the GSIs

Last Time - Control hazards (speculate and squash)

- ❑ How many cycles are wasted with speculate and squash?
 - Correct prediction?
 - Incorrect prediction?
- ❑ For the following code, assuming that only unnecessary instructions are squashed, how many cycles are wasted?
- ❑ Consider effects on data hazards given a correct, or incorrect branch prediction.

```
add   1 2 3
beq   1 5 1
nand  3 4 5
add   3 4 5
```

Today

- ❑ Performance
- ❑ Exceptions
- ❑ Advanced pipelining topics
 - Predication
 - Superscalar pipelining

Quick Review: The Basic Performance Equation

- ❑ Execution time (Time/Program) =
 - # of instr (I/P) × CPI (C/I) × cycle time (T/C)

- ❑ Multi-cycle decreases cycle time, but increases CPI

- ❑ Pipelining decreases CPI
 - Down to 1.0 if no stalls (hazards that are fixed by stalling)

Classic Performance Problem

- ❑ Program with following instruction breakdown:

lw	10%
sw	15%
beq	25%
R-type	50%
- ❑ Speculate “always taken” and squash.
 - 80% of branches taken
- ❑ Full forwarding to execute stage
 - 20% of loads stall for 1 cycle
- ❑ What is the CPI of the program?
- ❑ What is the total execution time if cycle time is T?

Classic Performance Problem (cont)

- ❑ Assume branches are resolved at Execute?
 - What is the CPI?
 - What happens to cycle time?
 - What is the total execution time?
- ❑ What if branches are resolved at Decode?

Exceptions

- ❑ Exception: When something unexpected happens during program execution
 - Example: Divide by zero
- ❑ Much harder with pipelined implementation
 - Multiple instructions executing at the same time
 - Simple Case: e.g., ALU overflow
 - “flush the pipeline after the exception”
 - “handle the exception”
 - Identify address of instruction causing exception (PC+1 in ID/EX pipeline register)
 - JALR to exception handler

Early Exceptions

- What about an early (fetch) exception?
 - Maybe a mis-speculated fetch
 - Branch is wrong, fetching “down the wrong path”
 - Solution:
 - Delay the handling of an exception until it is known to be a “real” problem.
 - Send noops down the pipeline from the point of the exception until you are sure.

Late Exceptions

- What about a late (WB) exception?
 - When does a **sw** modify state?
 - In the memory access stage which means it may have completed the write before the exception (to the instruction that logically precedes it) occurs.
 - Solution:
 - Delay the memory write until writeback.
 - What happens if the **sw** is followed by a **lw**?

Multiple Exceptions

- ❑ What about simultaneous exceptions?
 - `div 10 0 5`
 - `badop 4 4 4`
- ❑ They will generate a divide by 0 and an invalid opcode at the same cycle.
- ❑ Solution: assign priority
 - Generally deepest pipeline exception is handled, other can usually be ignored.

When good branches go bad

- ❑ Branching is expensive (slow) on pipelined processors –
- ❑ Cost of a branch can be reduced by **branch prediction**
- ❑ Not all branches are predictable; some depend on external, random data
- ❑ Many branches are for small **if** blocks; cost of the branch may exceed cost of the entire **if** block

A nasty branch

```
if (r1 != r2) r3 += r4
```

compiles to:

```
beq r1 r2 noadd  
add r3 r4 r3
```

noadd: ...

Assume that

- ❑ an add or a correctly predicted branch takes one cycle,
- ❑ a mispredicted branch takes three more cycles, and
- ❑ this particular branch is taken 50% of the time in a completely random pattern.

Branch takes an average of 2.5 cycles to execute! **Can we do better?**

A new idea...

- ❑ The conditional branch (average of 2.5 clocks in the LC2K pipeline) is relatively slow.
- ❑ The block executed conditionally (a single add, average of 0.5 clocks) is relatively fast.
- ❑ Maybe we should unconditionally do the add; that would only cost an extra 0.5 clocks
- ❑ But it will give us the wrong answer...

Generalized conditional instructions

- ❑ Suppose we can make all instructions (including our **add**) conditional.
- ❑ In addition to an opcode (and its operands), we also specify a condition (and its operands).
- ❑ Then we don't have to branch at all!

Lots of instruction bits

Nice idea, but it needs lots of register arguments:

`addifnoteq r1 r2 r3 r4 r5`

would act like:

`if (r1 != r2) r5 = r3 + r4`

We will need HUGE instructions to specify all this:
a condition (not equal) an operation (addition) and five registers!

Splitting it up

- ❑ Can we break it into two instructions? (without breaking our pipeline!)
- ❑ Let's use a boolean value to represent the true/false nature of the condition
- ❑ This boolean value can be saved somewhere (such as in a set of one-bit registers dedicated to this purpose)
- ❑ This gives us two instructions of more reasonable size:
 - Test: condition opcode, two register operands, boolean destination register
 - Action: operation opcode, up to three register operands, boolean predicate register

Branchless version

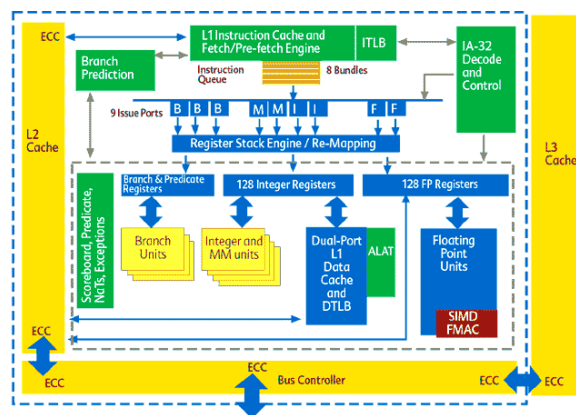
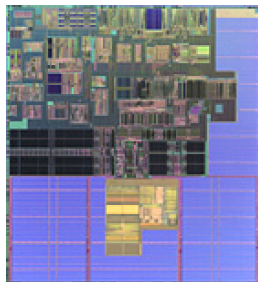
```
testeq r1 r2 bool6  
cadd r3 r4 r5 bool6
```

- ❑ Pipelines nicely; no control hazards
- ❑ Data hazard on bool6 can be easily handled by forwarding (no stalling!)
- ❑ Always runs in two cycles, whether the condition is true or false

Predication

- ❑ This idea is called **predicated execution** or **predication**
- ❑ Avoids branches
- ❑ For a hard to predict branch (e.g. 50% correct), when is branch prediction better than predication?
- ❑ Turns a control dependency into a data dependency (which is usually easier to deal with)
- ❑ Used extensively on the IA-64 (Itanium) architecture

Itanium-2 architecture



The IA-64 (Itanium) processor

- ❑ 64-bit architecture
- ❑ Superscalar, pipelined, in order execution
- ❑ 6 pipelined functional units
 - 2 load/store units
 - 2 64-bit integer units
 - 2 64-bit floating-point units
- ❑ 128 general 64-bit integer and 128 82-bit floating-point registers
- ❑ 64 1-bit predicate registers
- ❑ Fixed 41-bit instruction size
- ❑ Instructions come in 128-bit **bundles**
 - Three 41-bit instructions
 - One 5-bit template

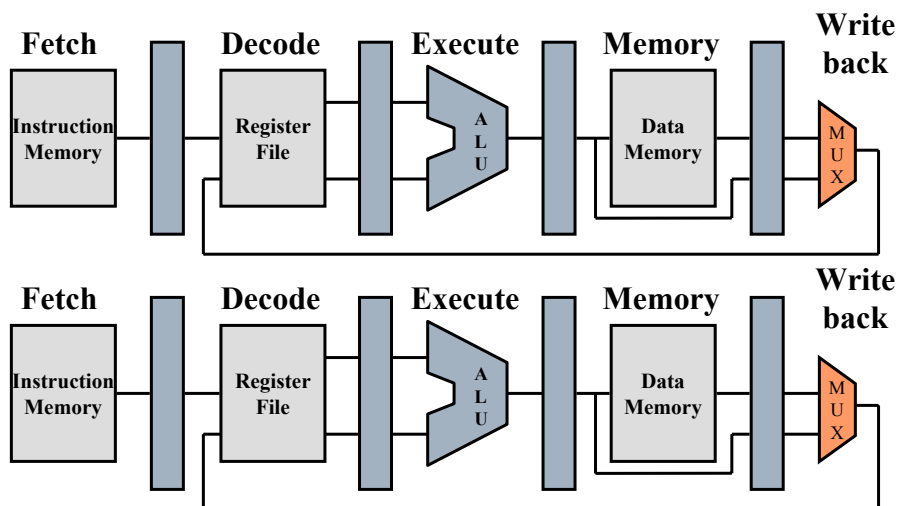
Superscalar Pipelining

- ❑ CPI for pipelining:
 - 1 (ideal case - no stalls)
 - > 1 (reality, depends on program)
- ❑ What if we want to improve performance more?
 - Want CPI as low as possible – even lower than 1
- ❑ Superscalar Pipelining
 - Build two (or more) pipelines that execute in parallel

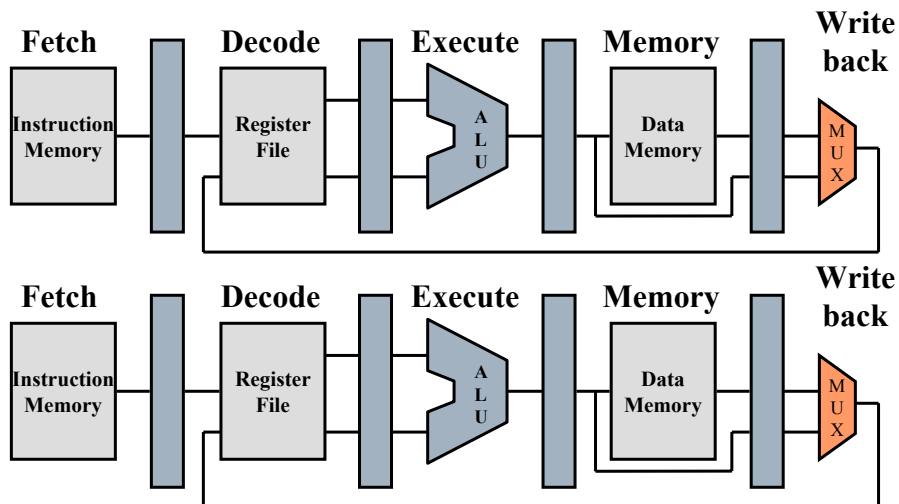
Creating two pipelines

- ❑ Have two processors
 - need two programs or a parallel program
 - does not improve single program performance
- ❑ Have two pipelines in same processor
 - pipelines need to work in tandem to improve single program performance

Superscalar Execution: Two pipelines are better than one!



Superscalar: Fetch Stage



EECS 370: Introduction to Computer Organization

The University of Michigan
© Austin & Papaefthymiou - 2009

25/39

Superscalar: Fetch Stage

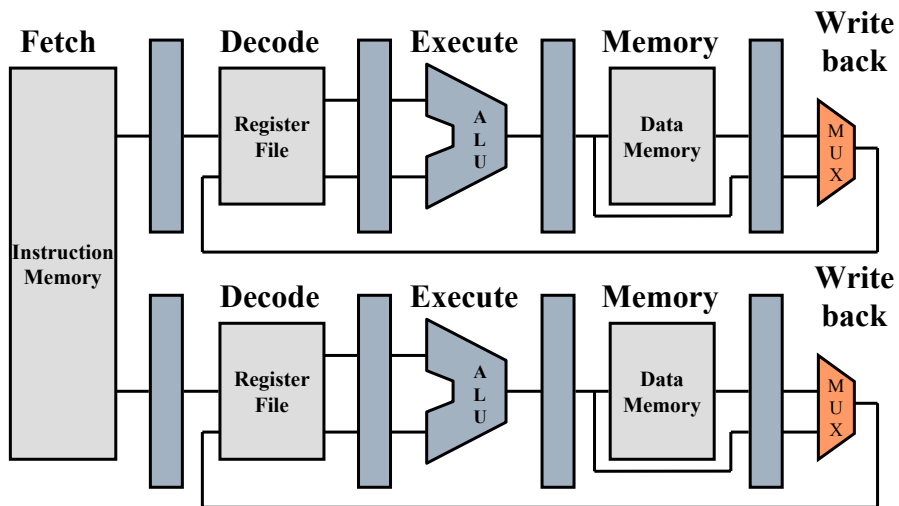
- ❑ Need to fetch two instructions:
 - Fetch 8 bytes instead of 4.
 - Increment PC by 8.
- ❑ What if first instruction is branch?
 - Already fetched PC+1
 - Cannot fetch taken target in same cycle (requires two memory accesses)
- ❑ Memory alignment could be an issue
 - Second instruction could be on different page

EECS 370: Introduction to Computer Organization

The University of Michigan
© Austin & Papaefthymiou - 2009

26/39

Superscalar: Fetch Stage

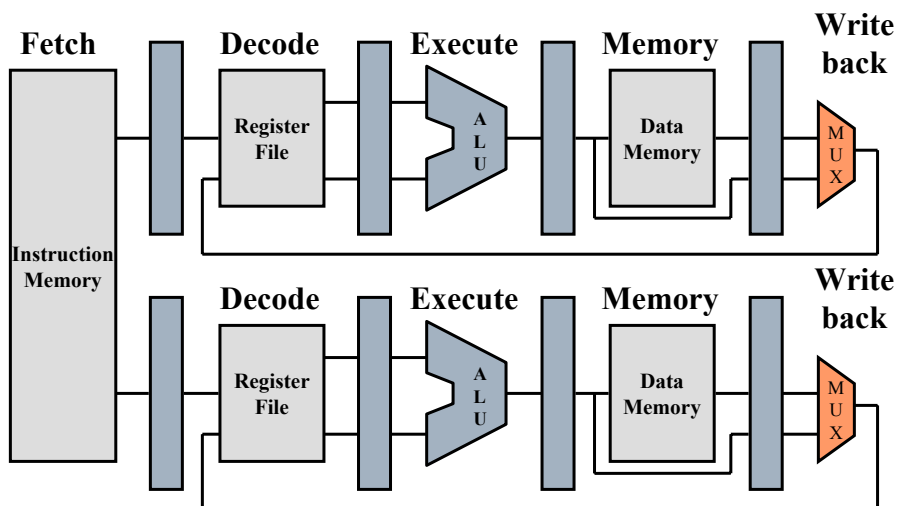


EECS 370: Introduction to Computer Organization

The University of Michigan
© Austin & Papaefthymiou - 2009

27/39

Superscalar: Decode Stage



EECS 370: Introduction to Computer Organization

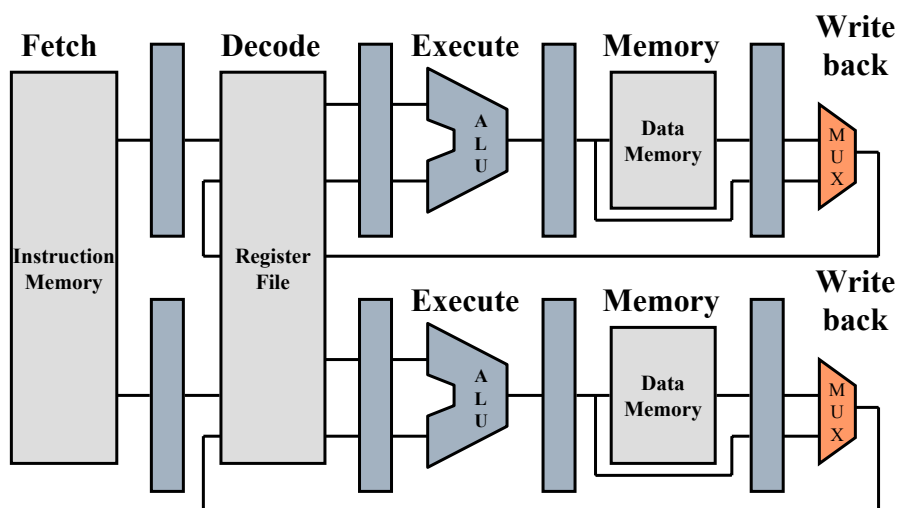
The University of Michigan
© Austin & Papaefthymiou - 2009

28/39

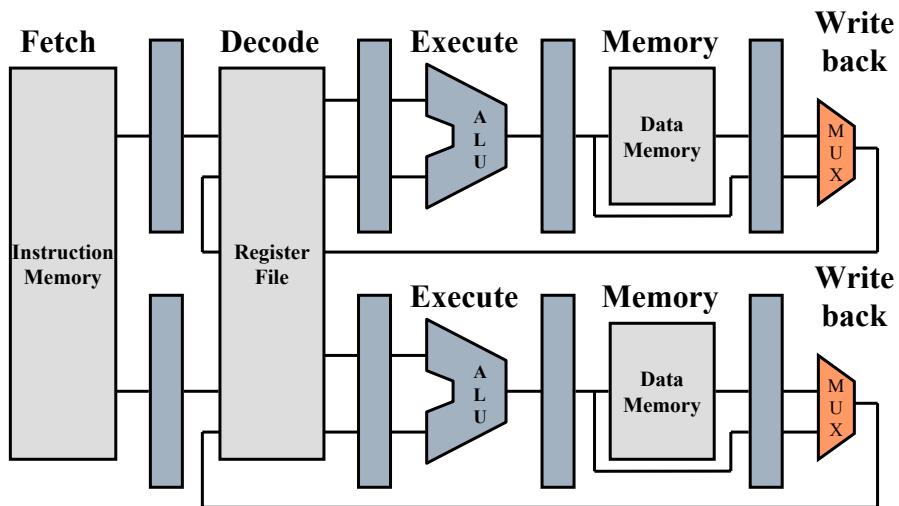
Superscalar: Decode Stage

- ❑ Register file needs to be able to support four register reads
- ❑ Hazard and forwarding logic much more complex
 - Can forward from values from both pipelines
 - Stall if second instruction depends on first
 - How do you stall a superscalar pipeline?
 - What if the instructions are variable length? (as in x86)

Superscalar: Decode Stage



Superscalar: Execute Stage



EECS 370: Introduction to Computer Organization

The University of Michigan
© Austin & Papaefthymiou - 2009

31/39

Superscalar: Execute Stage

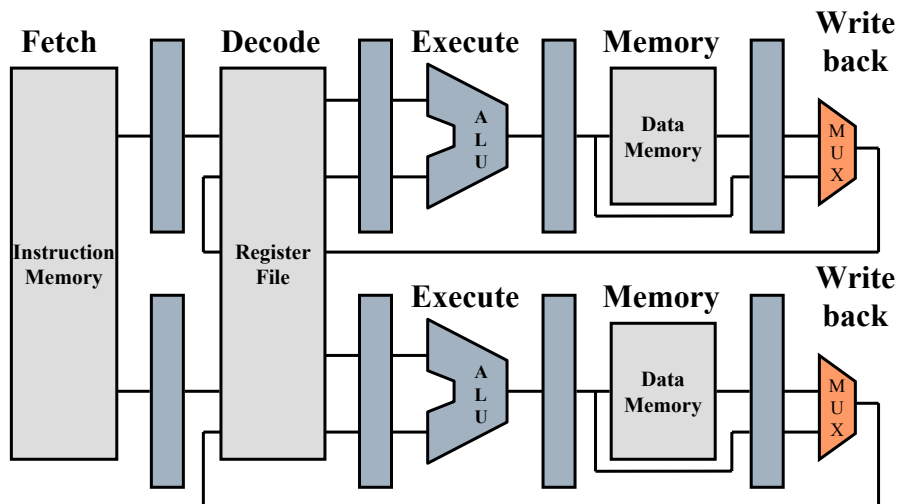
- ❑ Both pipelines can compute their values in parallel
- ❑ Each pipeline uses their own ALU

EECS 370: Introduction to Computer Organization

The University of Michigan
© Austin & Papaefthymiou - 2009

32/39

Superscalar: Memory Stage



EECS 370: Introduction to Computer Organization

The University of Michigan
© Austin & Papaefthymiou - 2009

33/39

Superscalar: Memory Stage

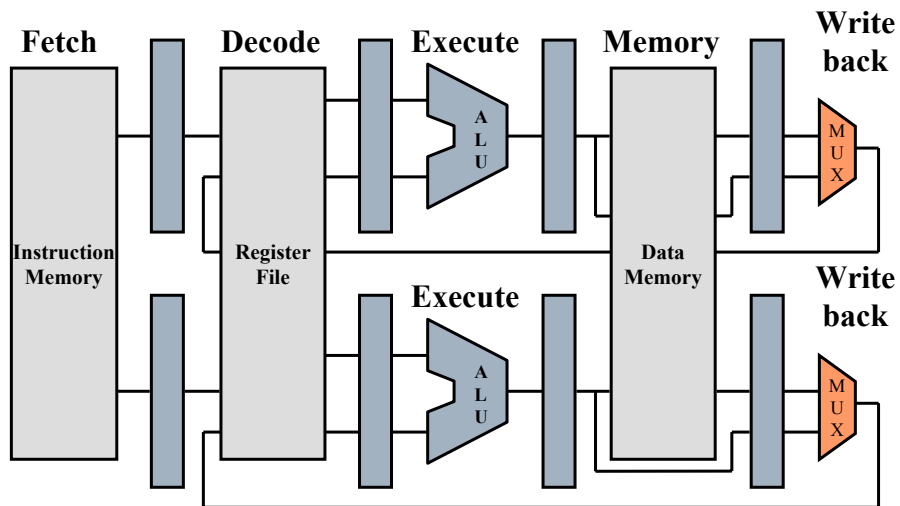
- ❑ What happens if both instructions access the same address?
 - First: load, Second: load → no problem
 - First: load, Second: store → read first
 - First: store, Second: load → write first
 - First: store, Second: store → only store second
- ❑ Register file trick (write first half, read second half) won't work here
- ❑ Can't detect address match until MEM stage

EECS 370: Introduction to Computer Organization

The University of Michigan
© Austin & Papaefthymiou - 2009

34/39

Superscalar: Memory Stage



EECS 370: Introduction to Computer Organization

The University of Michigan
© Austin & Papaefthymiou - 2009

35/39

Superscalar: Memory Stage

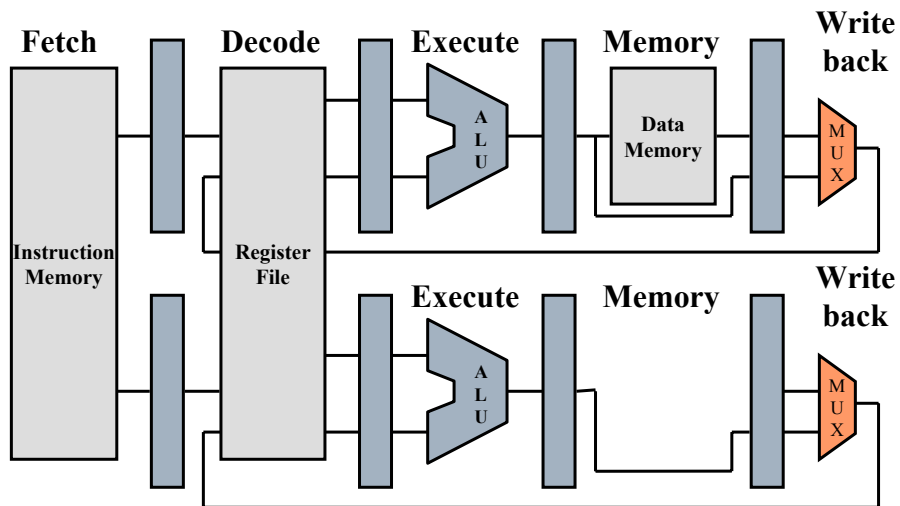
- ❑ One solution: Only allow memory accesses in one pipeline
- ❑ If memory access is encountered in the wrong pipeline, stall and execute in proper pipeline
- ❑ This type of stall is called a **structural hazard**
- ❑ Better solutions are possible

EECS 370: Introduction to Computer Organization

The University of Michigan
© Austin & Papaefthymiou - 2009

36/39

Superscalar: Memory Stage

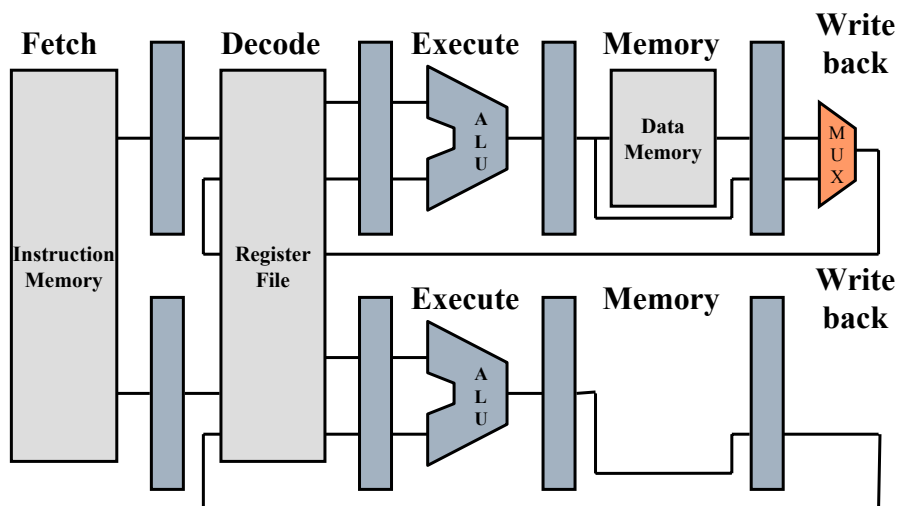


EECS 370: Introduction to Computer Organization

The University of Michigan
© Austin & Papaefthymiou - 2009

37/39

Superscalar: Writeback Stage



EECS 370: Introduction to Computer Organization

The University of Michigan
© Austin & Papaefthymiou - 2009

38/39

Superscalar: Writeback Stage

- ❑ Need to support the writeback of two registers
- ❑ What if both instructions write the same register?