Pipelining

Ref: Chapter 6

Multicycle Instructions

- Chop each instruction in to stages.
- Each stage takes one cycle.
- We need to provide some way to sequence through the stages:
  - microinstructions
- Stages can *share* resources (ALU, Memory).

Pipelining

- We can overlap the execution of multiple instructions.
- At any time, there are multiple instructions being executed – each in a different stage.

- So much for sharing resources ?!?

The Laundry Analogy

Non-pipelined approach:

1. run 1 load of clothes through washer
2. run load through dryer
3. fold the clothes (optional step for students)
4. put the clothes away (also optional).

Two loads? Start all over.

Pipelined Laundry

- While the first load is drying, put the second load in the washing machine.
- When the first load is being folded and the second load is in the dryer, put the third load in the washing machine.
- Admittedly unrealistic scenario for CS students, as most only own 1 load of clothes…

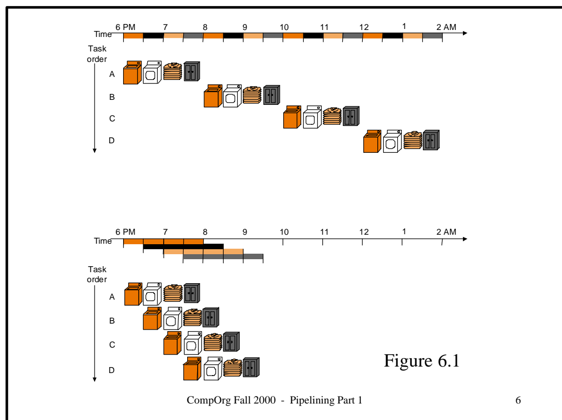


Figure 6.1

Laundry Performance

- For 4 loads:
  - non-pipelined approach takes 16 units of time.
  - pipelined approach takes 7 units of time.
- For 816 loads:
  - non-pipelined approach takes 3264 units of time.
  - pipelined approach takes 819 units of time.

Execution Time vs. Throughput

- It still takes the same amount of time to get your favorite pair of socks clean, pipelining won't help.
- However, the total time spent away from CompOrg homework is reduced.

It's the classic "Socks vs. CompOrg" issue.

Instruction Pipelining

First we need to break instruction execution into discrete stages:

1. Instruction Fetch
2. Instruction Decode/ Register Fetch
3. ALU Operation
4. Data Memory access
5. Write result into register

Operation Timings

- Some estimated timings for each of the stages:

Instruction Fetch	2ns
Register Read	1ns
ALU Operation	2ns
Data Memory	2ns
Register Write	1ns

Comparison

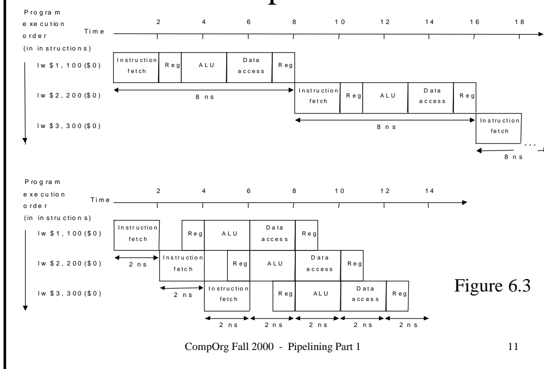


Figure 6.3

RISC and Pipelining

- One of the major advantages of RISC instruction sets is the complexity of a pipeline implementation.
  - It's more complex in a CISC processor.
- RISC (MIPS) design features that make pipelining easy include:
  - single length instruction (always 1 word)
  - relatively few instruction formats
  - load/store instruction set
  - operands must be aligned in memory (a single data transfer instruction requires a single memory operation).

Hazard

- Your pants are clean, dry and ready to wear.
  - This is know as CD&RTW.
- Your underwear is still wet (from the washing)
- The process of getting dressed "stalls" while you wait for your underwear to dry.

OK, so perhaps not all of you would wait

Pipeline Hazard

- Something happens that means the next instruction cannot execute in the following clock cycle.
- Three kinds of hazards:
  - structural hazard
  - control hazard
  - data hazard

Structural Hazards

- Two stages require the same resource.
  - What if we only had enough electricity to run either the washer or the dryer at any given time?
  - What if MIPS datapath had only one memory unit instead of separate instruction and data memory?

Avoiding Structural Hazards

- Design the pipeline carefully.
- Might need to duplicate resources
  - an Adder to update PC, and ALU to perform other operations.
- Detecting structural hazards at execution time (and delaying execution) is not something we want to do (structural hazards are minimized in the design phase).

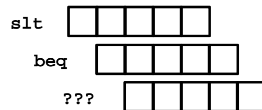
Control Hazards

- When one instruction needs to make a decision based on the results of another instruction that has not yet finished.
- Example: conditional branch
  - The instruction that is fed to the pipeline right after a **beq** depends on whether or not the branch is taken.

beq Control Hazard

```
a = b+c;
if (x!=0)
      y++;
...
```

```
      slt $t0,$s0,$s1
      beq $t0,$zero,skip
      addi $s0,$s0,1
skip:
      lw $s3,0($t3)
```



The instruction to follow the **beq** could be either the **addi** or the **lw**, it depends on the result of the **beq** instruction.

One possible solution - stall

- We can include in the control unit the ability to *stall* (to keep new instructions from entering the pipeline until we know which one).
- Unfortunately conditional branches are very common operations, and this would slow things down considerably.

A Stall

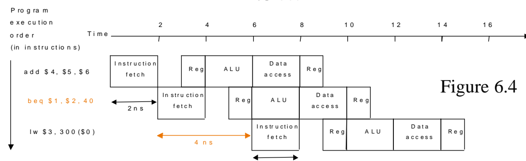


Figure 6.4

To achieve a 1 cycle stall (as shown above), we need to modify the implementation of the **beq** instruction so that the decision is made by the end of the second stage.

Another strategy

- Predict whether or not the branch will be taken.
- Go ahead with the *predicted* instruction (feed it into the pipeline next).
- If your prediction is right, you don't lose any time.
- If your prediction is wrong, you need to undo some things and start the correct instruction

Predicting branch not taken

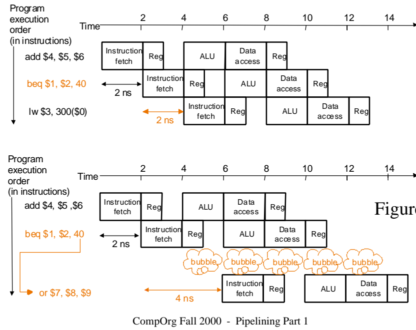


Figure 6.5

CompOrg Fall 2000 - Pipelining Part 1 22

Dynamic Branch Prediction

- The idea is to build hardware that will come up with a prediction based on the past history of the specific branch instruction.
- Predict the branch will be taken if it has been taken more often than not in the recent past.
  - This works great for loops! (90% + correct).

CompOrg Fall 2000 - Pipelining Part 1 23

Yet another strategy: delayed branch

- The compiler rearranges instructions so that the branch actually occurs delayed by one instruction.
- This gives the hardware time to compute the address of the next instruction.
- The new instruction is hopefully useful whether or not the branch is taken (this is tricky - compilers must be careful!).

CompOrg Fall 2000 - Pipelining Part 1 24

```
a = b+c;
if (x!=0)
      y++;
...
```

Delayed Branch

```
      add $s2,$s3,$s4
      beq $t0,$zero,skip
      addi $s0,$s0,1
skip:
      lw $s3,0($t3)
```

Order reversed!

beq

add

The compiler must generate code that differs from what you would expect.

Data Hazard

- One of the values needed by an instruction is not yet available (the instruction that computes it isn't done yet).
- This is like the CompOrg vs. Socks issue.
- This will cause a data hazard:

```
add $t0,$s1,$s2
addi $t0,$t0,17
```


adds $s1 and $s2

selects $s1 and $s2 for ALU op

stores sum in $t0

add $t0,$s1,$s2 — IF | Reg | ALU | Data Access | Reg

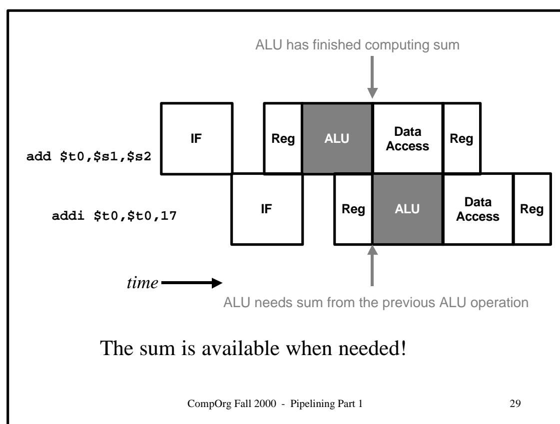
addi $t0,$t0,17 — IF | Reg | ALU | Data Access | Reg

time →

selects $t0 for ALU op

Handling Data Hazards

- We can hope that the compiler can arrange instructions so that data hazards never appear.
  - this doesn't work, as programs generally need to use previously computed values for everything!
- Some data hazards aren't real - the value needed is available, just not in the right place.



The sum is available when needed!

Forwarding

- It's possible to *forward* the value directly from one resource to another (in time).
- Hardware needs to detect (and handle) these situations automatically!
  - This is difficult, but necessary.

Picture of Forwarding

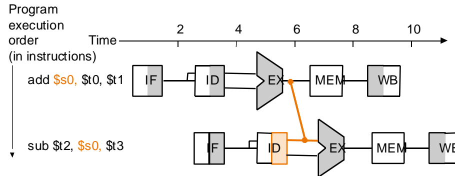


Figure 6.8

Another Example

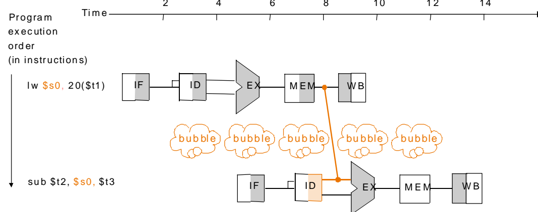


Figure 6.9

Pipelining and CPI

- If we keep the pipeline full, one instruction completes every cycle.
- Another way of saying this: the average time per instruction is 1 cycle.
  - even though each instruction actually takes 5 cycles (5 stage pipeline).

CPI=1

Correctness

Pipeline and compiler designers must be careful to ensure that the various schemes to avoid stalling do not change what the program does!

- only when and how it does it.
- It's impossible to test all possible combinations of instructions (to make sure the hardware does what is expected).
- It's impossible to test all combinations even without pipelining!

Pipelined Datapath

We need to use a multicycle datapath.

- includes registers that store the result of each stage (to pass on to the next stage).
- can't have a single resource used by more than one stage at time.

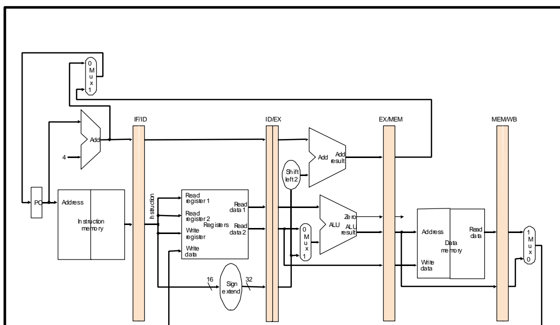


Figure 6.12

`lw` and pipelined datapath

- We can trace the execution of a load word instruction through the datapath.
- We need to keep in mind that other instructions are using the stages not in use by our **`lw`** instruction!

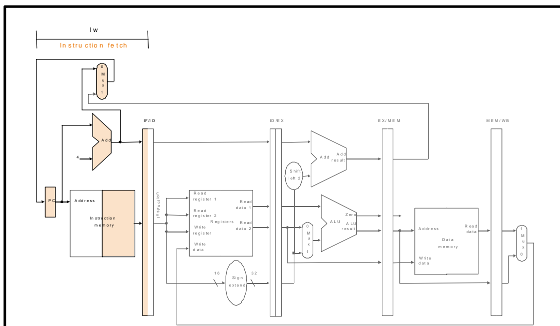


Figure 6.13 Stage 1: EX (ALU Op)

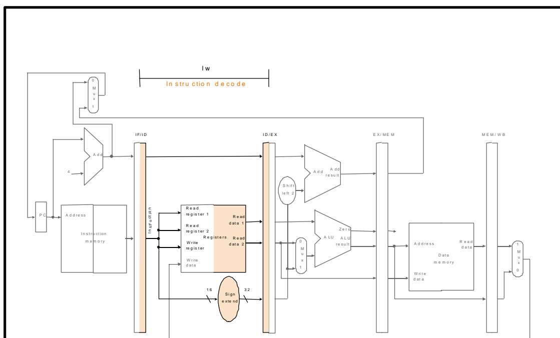


Figure 6.13 Stage 2: ID

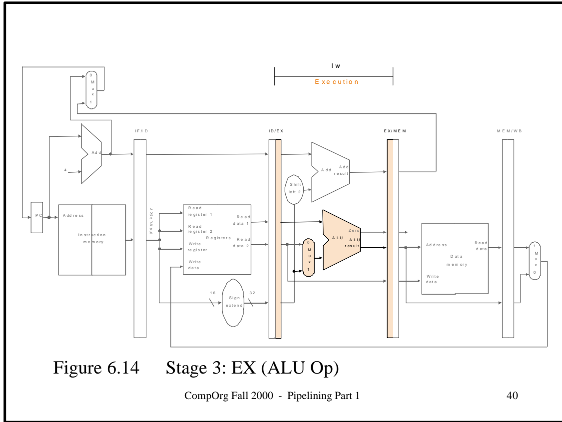


Figure 6.14 Stage 3: EX (ALU Op)

CompOrg Fall 2000 - Pipelining Part 1 40

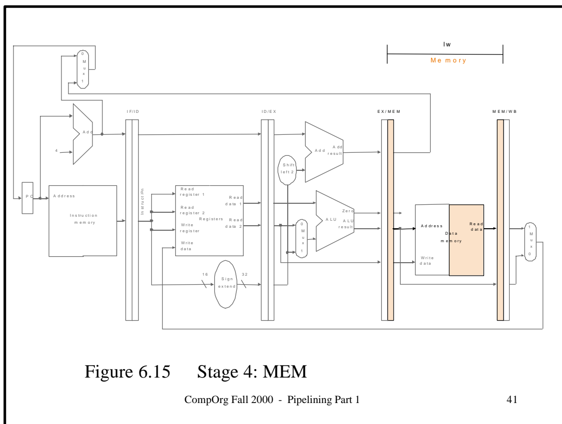


Figure 6.15 Stage 4: MEM

CompOrg Fall 2000 - Pipelining Part 1 41

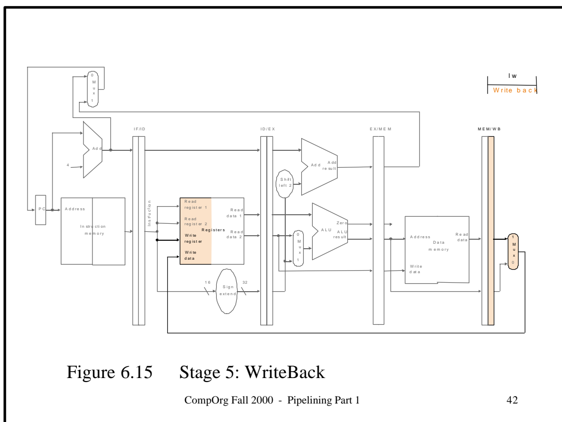


Figure 6.15 Stage 5: WriteBack

CompOrg Fall 2000 - Pipelining Part 1 42

A Bug!

- When the value read from memory is written back to the register file, the inputs to the register file (write register #) are from a different instruction!
- To fix the bug we need to save the part of the `lw` instruction (5 bits of it specify which register should get the value from memory).

New Datapath

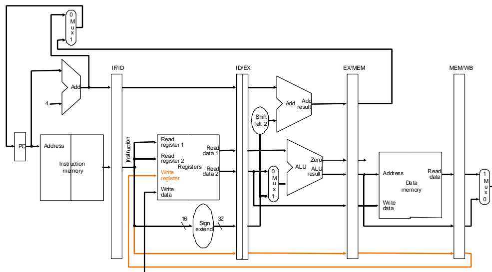
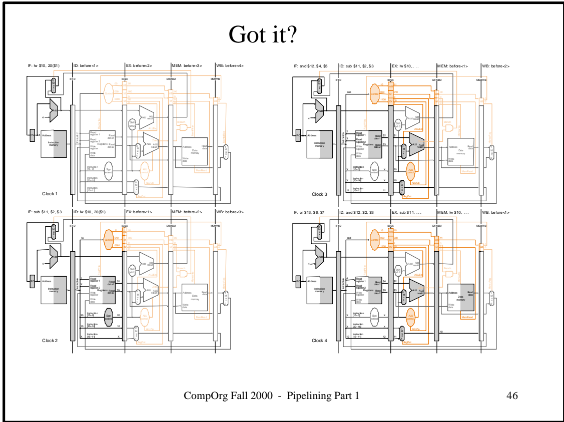


Figure 6.18

Pipeline Control System

- We need to build a new control system for a pipelined datapath.
- There are lots of complications, but the general approach is the same.
- We can learn everything we need to know about building a pipelined control system in one slide:

Got it?



Skipping Ahead

- We are not going over the details of the design of a pipelined datapath or control system.
- We will skip ahead to talk about multiple issue (superscalar), dynamic pipeline scheduling and advances in laundry technology.