CSE 2021

Computer Organization

CPU DESIGN

Pipelining

Prof. H. Roumani,
Dept of CS & Eng, York University

PIPELINING

- Basic Concepts
- Implementation
- Hazards
- Hazard Handling

The Idea

Assume delays of 1, 2, and 4 min for ordering, paying, and making a sandwich in the worst case.

What is the latency and throughput of a “single-cycle” restaurant (door opens and shuts at regular intervals allowing one customer to enter and one to leave)?

7 min, 8.6 person/hr

Is this arrangement simple? Does it cater to the slowest customer? Is it wasteful (idle employees)?

Yes. Yes. Yes.

The Idea, cont.

What is the latency and throughput of a "pipelined" restaurant?

12 min, 15.0 person/hr

Compared to the single-cycle restaurant, latency has become worse but throughput is better!

Is this arrangement as simple? Caters to the slowest customer? Is it wasteful (idle employees)?

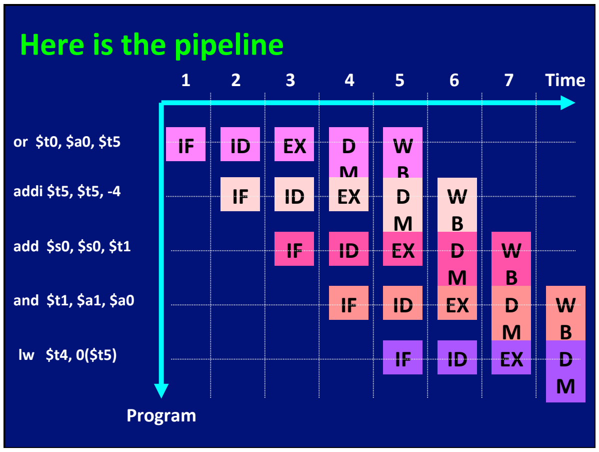
Almost (need to marshal)**. No. No.**

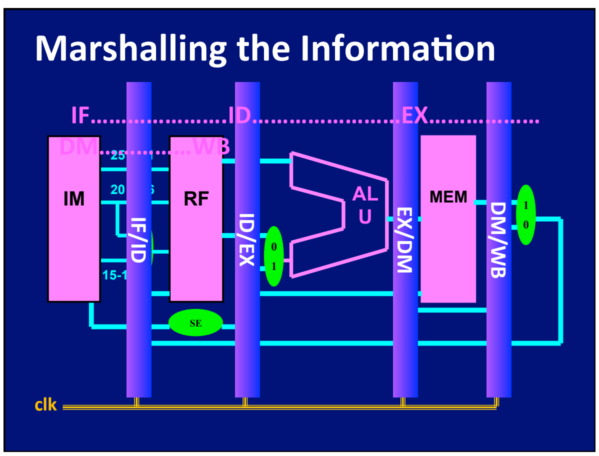
Basic Concepts

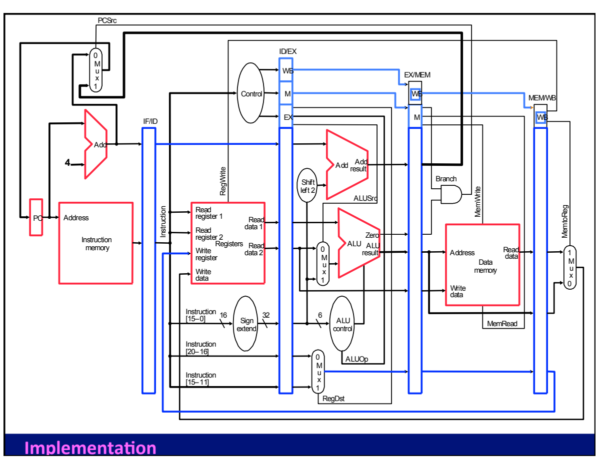
- **Assembly Line**
  Stages (Order, Pay, Make) **and Marshaling.**
- **Balancing, Latency, and Throughput**
  Balancing the stages increases the latency
- **The Performance Potential**
  Up to n-fold improvement in throughput (for n stages)
- **No Dependencies**
  - Can I add small fries?
  - I'll have the same as my friend in the front.

A Pipelined CPU

- **Assembly Line**
  Five stages:
  IF (in IM), ID (in RF), EX (in ALU), DM (in MEM), WB (in RF)
- **Balancing, Latency, and Throughput**
  IF (200), ID (50), EX (100), DM (200), WB (50)
  Balance 200. Latency is thus 1000 (not 600) or 1 GHz
- **The Performance Potential**
  Up to 5 GHz


Here is the pipeline
1 2 3 4 5 6 7 Time
or $t0, $a0, $t5
addi $t5, $t5, -4
add $s0, $s0, $t1
and $t1, $a1, $a0
lw $t4, 0($t5)
IF ID EX DM WB
Program


Marshalling the Information
IF……ID……EX……
DM……WB
IM
RF
ALU
MEM
IF/ID
ID/EX
EX/DM
DM/WB
SE
clk


PCSrc
IF/ID
ID/EX
EX/MEM
MEM/WB
Control
WB
M
EX
Add
Add result
Shift left 2
ALUSrc
Branch
MemWrite
MemtoReg
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Zero
ALU
ALU result
Address
Data memory
Read data
Write data
MemRead
Instruction [15–0]
Sign extend
16
32
6
ALU control
ALUOp
Instruction [20–16]
Instruction [15–11]
RegDst
Implementation

Exercise

During a particular clock cycle, the five highlighted instructions were be in the pipeline; in particular, the addi instruction of address 420 was in its EXecution stage. Determine the content of EX/MEM at the end of that cycle.

```
400 addi $a0, $0, 25
404 addi $t6, $0, 13
408 addi $s0, $0, 47
412 add $t5, $0, $0
416 sub $t0, $0, $a0
420 addi $t6, $a0, 22
424 lw $t7, 24($a0)
428 lw $t8, 12($a0)
...
```

FIELD	WIDTH	VALUE
WB		
M		
Add Result		
Zero		
ALU Result		
Read Data 2		
Write Register		

Answer

During a particular clock cycle, the five highlighted instructions were be in the pipeline; in particular, the addi instruction of address 420 was in its EXecution stage. Determine the content of EX/MEM at the end of that cycle.

```
400 addi $a0, $0, 25
404 addi $t6, $0, 13
408 addi $s0, $0, 47
412 add $t5, $0, $0
416 sub $t0, $0, $a0
420 addi $t6, $a0, 22
424 lw $t7, 24($a0)
428 lw $t8, 12($a0)
...
```

FIELD	WIDTH	VALUE
WB	2	10_B
M	3	000_B
Add Result	32	512_D
Zero	1	0
ALU Result	32	47_D
Read Data 2	32	13_D
Write Register	5	14_D

Exercise

During a particular clock cycle, the five highlighted instructions were be in the pipeline; in particular, the addi instruction of address 420 was in its EXecution stage. Determine the values of the wires at the end of that cycle (see next slide).

```
400 addi $a0, $0, 25
404 addi $t6, $0, 13
408 addi $s0, $0, 47
412 add $t5, $0, $0
416 sub $t0, $0, $a0
420 addi $t6, $a0, 22
424 lw $t7, 24($a0)
428 lw $t8, 12($a0)
...
```

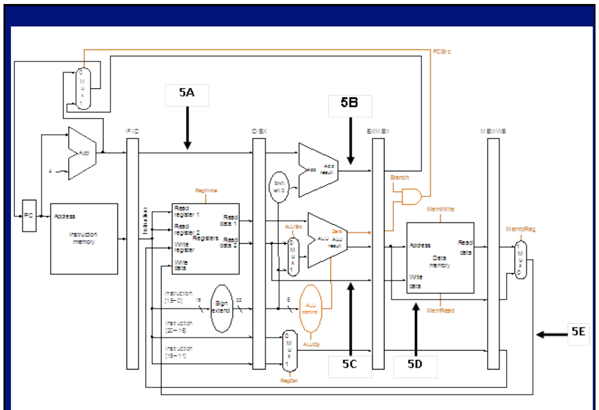
FIELD	VALUE
5A	
5B	
5C	
5D	
5E	

Answer

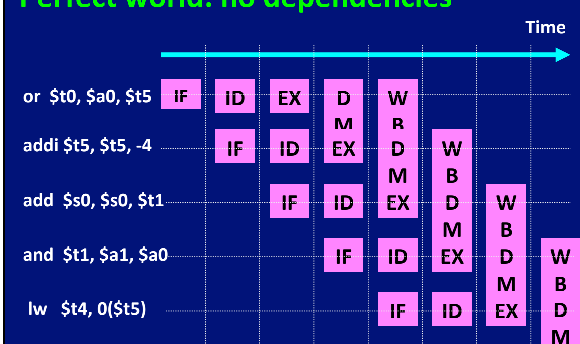
During a particular clock cycle, the five highlighted instructions were be in the pipeline; in particular, the addi instruction of address 420 was in its EXecution stage. Determine the values of the wires at the end of that cycle (see next slide).

```
400 addi $a0, $0, 25
404 addi $t6, $0, 13
408 addi $s0, $0, 47
412 add $t5, $0, $0
416 sub $t0, $0, $a0
420 addi $t6, $a0, 22
424 lw $t7, 24($a0)
428 lw $t8, 12($a0)
...
```

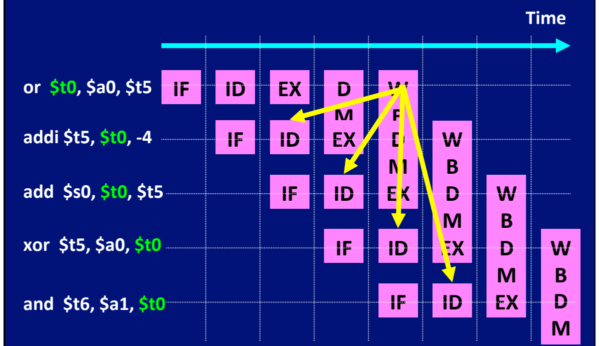
FIELD	VALUE
5A	428
5B	512
5C	13
5D	-25
5E	0



Perfect world: no dependencies



Hazards



Hazards

Types, Detection, and Avoidance

- **Structural**

  Can I add small fries (out of sequence)

  Out of sequence like addm. Never happens in MIPS

- **Data**

  I order based on how much my friend paid

  RF/ALU and ALU/DM

- **Control**

  I'll have the same as my friend in the front

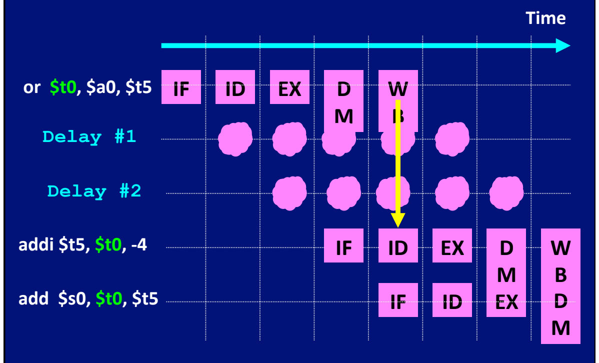
  All branches (conditional transfers)

Type 1 Data Hazard: Compute + dependency

Example:

```
addi    $t0, $0, 15
or      $s5, $s5, $t0
```

Solution #1: Create Two Delays in between



But how to populate the delay slots?

1. **Nop's**
   Have the compiler insert two nop's
   BAD! Creates DRAM access delays and crowds the cache
2. **Benign instructions**
   Have the compiler reorder the code
   May or may not succeed
3. **Bubbles**
   Have the CPU stall the pipeline
   OK but we lost ~half the gain!

None of these is good ☹

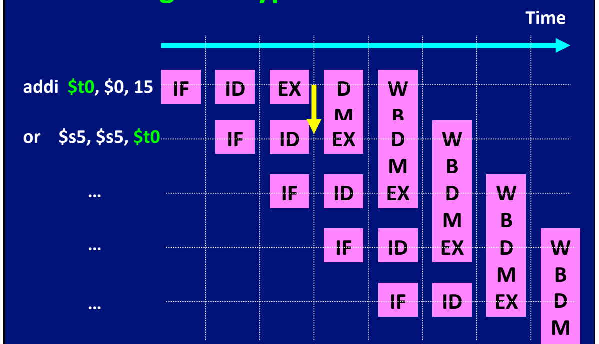
Solution #2: FORWARDING

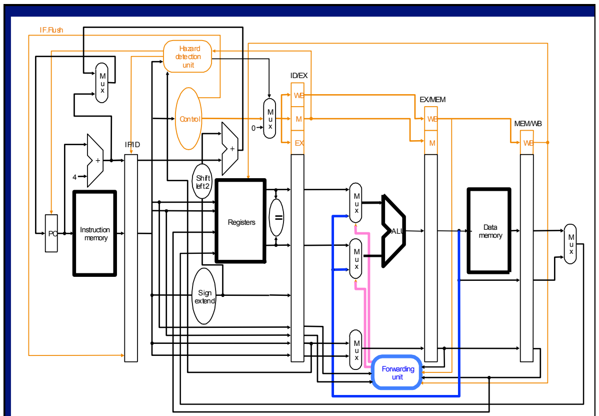
Change the datapath so that the output of the ALU is sent back to its input, i.e. forwarded directly to the next instruction.

Data is thus available just in time rather than having to wait for DM and WB.

This is a perfect solution! No stalls!

Forwarding in a Type-1 data hazard





Implementation

Type 2 Data Hazard: Load + dependency

Example:

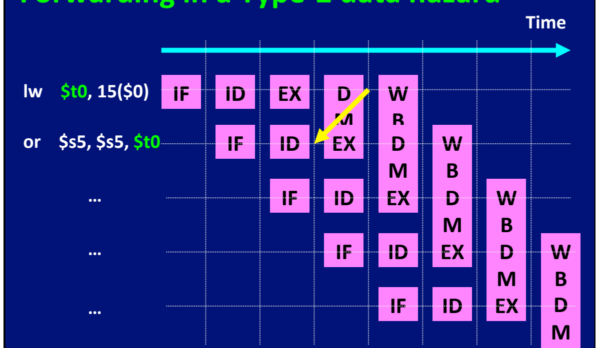
```
lw      $t0, 16($0)
or      $s5, $s5, $t0
```

Type 2 Data Hazard

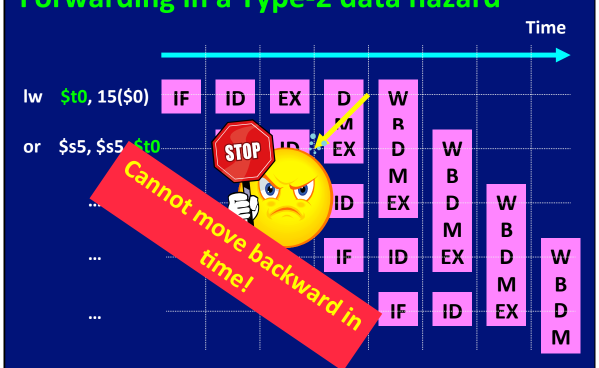
Solution #1
We can, as in Type-1, insert two delays between the dependent instructions. But this will degrade the performance.

Solution #2
Forwarding worked very well for us in Type-1. Can we employ it here?

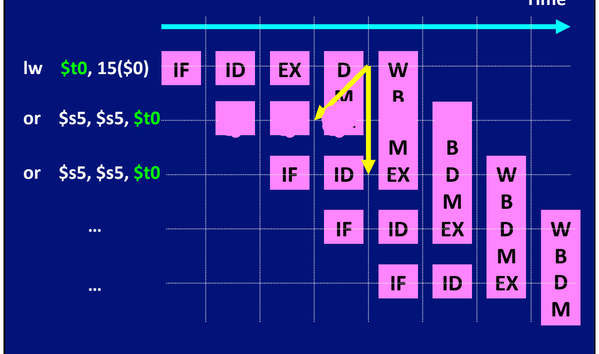
Forwarding in a Type-2 data hazard



Forwarding in a Type-2 data hazard



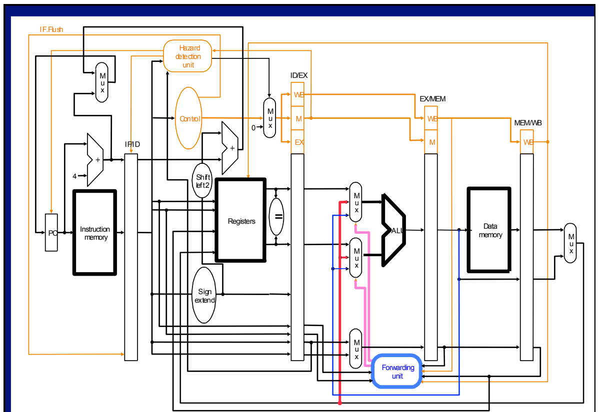
Forwarding in a Type-2 data hazard



Type 2 Data Hazard

Hence, forwarding works for Type-2 as well

But ONE stall is needed.

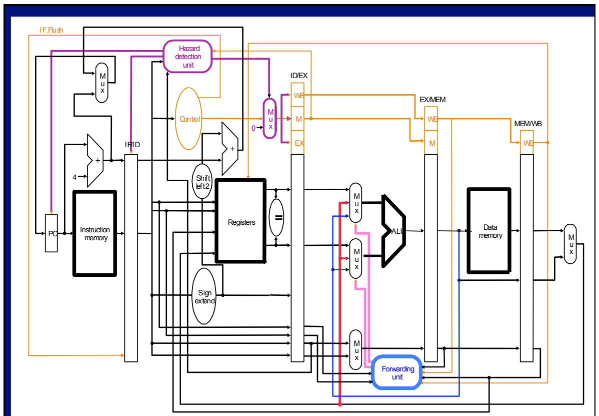


Implementation

But how can we create a stall?

When a type-2 is detected (i.e. lw in EX and some ins that depends on it in ID), create a stall between them:

- The instruction in ID must stay in ID. How?
  Disable IF/ID!
- The instruction in IF must stay in IF. How?
  Disable PC!
- A fake but benign instruction must be marshaled from
  ID to EX. How?
  Zero out the controls in ID/EX



Implementation

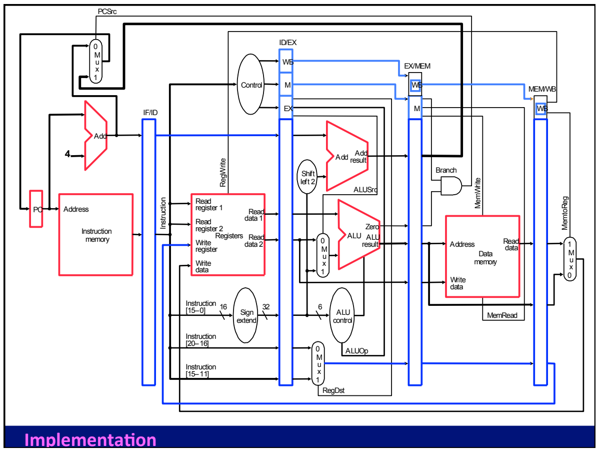
Control Hazards: branches

By the time the branch decision is taken, <u>two</u> instructions are already in the pipeline! Why?

Example:

```
beq     $t0, $s0, loop
or      $s5, $s5, $a0
addi    $t3, $t3, 1
```

Hence, two stalls are needed per branch ☹

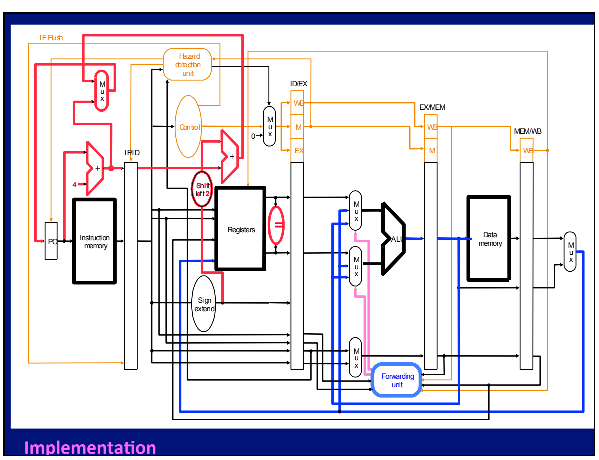


Control Hazards …

Branches account for ~25% of all instructions ➔ two stalls per branch is unacceptable.

Reduce them to <u>one</u> stall by detecting equality and computing the target in ID rather than EX

To implement this stall, flush the IF/ID register. When the clock ticks, IF/ID would hold nop, and IF would be fetching PC+4 or PC+4+target*4 (PC=branch address).



Approaches to Branch Hazards

- Always stall once
- Assume the branch will not be taken
- Predict the branch target
- Delay the branch (S/W implication)

Critique these approaches.
Assume a 100-iteration loop and analyze the factors that determine the likely penalty per branch in each approach

The Pipeline

Performance

```
do
{
      int el = ar[i];
      sum += el;
      sha ^= el;
      i++;
} while (el != last);
```

```
loop:  lw    $t5, ar($t0)
       addu  $s0, $s0, $t5
       xor   $s2, $s2, $t5
       addi  $t0, $t0, 4
       bne   $t5, $a0, loop
```

How long does it take to execute this loop on a single-cycle machine?

Assume component latencies of:
RF=50, ALU=100, and MEM (both IM and DM)=200 ps

Answer: 3.0 ns/iteration

```
do
{
      int el = ar[i];
      sum += el;
      sha ^= el;
      i++;
} while (el != last);
```

```
loop: lw    $t5, ar($t0)
      addu  $s0, $s0, $t5
      xor   $s2, $s2, $t5
      addi  $t0, $t0, 4
      bne   $t5, $a0, loop
```

How long does it take to execute this loop on a multi-cycle machine?

Assume component latencies of:
RF=50, ALU=100, and MEM (both IM and DM)=200 ps

Answer: 2.15 ns/iteration

```
do
{
      int el = ar[i];
      sum += el;
      sha ^= el;
      i++;
} while (el != last);
```

```
loop: lw    $t5, ar($t0)
      addu  $s0, $s0, $t5
      xor   $s2, $s2, $t5
      addi  $t0, $t0, 4
      bne   $t5, $a0, loop
```

How long does it take to execute this loop on a pipelined machine?

Assume component latencies of:
RF=50, ALU=100, and MEM (both IM and DM)=200 ps

Answer: 1.0 to 1.6 ns/iteration